目　录

第一章　起步篇 11

本章概述 11

撰写本书的背景 11

如何使用本书 12

所需的 IDL 版本 12

IDL 运行期间所需颜色的数量 12

少于 150 种颜色该怎样？ 12

多于 256 种颜色该怎样？ 13

创建 IDL 的启动文件 13

本书的风格习惯 14

大写 14

注释 14

续行符 15

本书中所用的 IDL 程序和数据文件 15

安装程序和数据文件 15

获取 IDL 的主目录和当前目录 15

下载本书所用的程序和数据文件 16

确保 Coyote 目录在 IDL 的搜索路径内 16

拷贝数据文件 16

获取更多的帮助 17

使用 IDL 命令 17

IDL 命令解析 17

位置参数 17

关键字参数 18

IDL 过程和函数 18

用 IDL 命令帮助 19

创建命令日志 19

创建变量 19

动态改变变量的属性 20

注意整型变量 21

使用矢量和数组 22

创建矢量 22

数组下标的应用 23

数组的建立 23

数组中元素的存取 24

矢量和子数组的提取 24

使用 IDL 图形窗口 24

图形窗口的建立 25

确定当前图形窗口 25

使图形窗口成为当前窗口 25

删除图形窗口 25

图形窗口的位置和尺寸 26
将图形窗口设置到显示器最前面 26
在图形窗口上设置标题 26
清除图形窗口内容 27

第二章　简单的图形显示 27

本章概述 27

IDL 中简单的图形显示 27

创建线画图 28

定制线画图 30

改变线条的线型和粗细 30
用符号代替线条表示数据 31
用线条和符号来显示数据 31
创建自己的图形符号 32
用不同的颜色绘制线画图 32
限定线画图的范围 33
改变线画图的风格 34

在线画图上绘出多种数据集 35

在多个轴的图上显示数据 36

创建曲面图 36

定制曲面图 38

旋转曲面图 39
为曲面赋色 39
修改曲面图外观 40

创建阴影曲面图 41

改变阴影处理参数 41
用其他数据集为阴影处理提供参数 42

创建等值线图 43

选择等值线数目 44

修改等值线图 45

改变等值线图的外观 46
给等值线图赋色 47

创建填充的等值线图 48

在显示窗口定位图形输出 49

设置图形边缘 50
设置图形位置 51
设置图形区域 52
创建多个图形 52
给单一窗口的多幅图形留下标题空间 53
使用!P.Multi 变量创建不对称的排列 54

给图形显示添加文本 55

列出可用字体的名称.....56
用 XYOutS 命令添加文本.....57
用 XYOut 加注矢量字体.....57
排列文本.....58
删除文本.....59
改变文本的方向.....59
给图形显示添加线和符号 59
图形显示添加色彩 60
第三章　图像数据处理 62
本章概要 62
图像处理 63
显示图像.....63
调整图像数据.....65
用颜色表分段表示图像 65
在 24 位显示器上用不同的颜色表显示图像.....66
显示 24 位图像.....66
在 24 位显示器上显示 24 位图像 67
在 24 位显示器上显示 8 位图像 67
控制图像显示顺序.....68
改变图像尺寸.....68
在 PostScript 设备上改变图像大小 69
在显示窗口中定位图像.....69
用归一化的坐标来定位图像 70
从显示器中读取图像.....72
在 24 位显示器上抓屏 72
读取显示图像的一部分 72
IDL 中基本的图像处理 73
直方图均衡化.....73
平滑图像.....74
消除图像噪声 75
增强图像边缘.....76
图像的频域滤波.....77
创建图像滤波器 77
第四章　图形显示技术 79
本章概要 79
IDL 的颜色运用 79
使用索引颜色模式和 RGB 颜色模式.....79
静态与动态颜色视觉 80
在 8 位显示器上指定颜色 81
在 24 位显示器上指定分解后的颜色 81
在 24 位显示设备上指定没有分解过的颜色.....83

决定颜色分解的开与关 83
在 24 位显示设备上装载颜色表 84
获得颜色表的拷贝 84
修改和创建颜色表 85
保存自己的颜色表 86
创建自己的轴标注 87
调整轴刻度间隔 87
格式化轴的标注 88
编写刻度格式函数 88
用 IDL 处理残缺的数据 91
用 IDL 建立三维坐标系 93
建立三维散点图 93
从图形原点定位 3D 坐标轴 94
组合简单图形显示 95
IDL 中的动画图形 97
建立动画工具 97
装载动画缓冲区 98
运行动画工具 98
动画的控制 98
存储动画的像素映射图 99
其他类型图形数据的动画 99
数据网格化及显示 99
德洛内三角形法网格化 100
数据的球形网格化 102
第五章　图形显示技巧 103
本章概要 103
将光标用于图形显示 104
什么时候返回的光标位置？ 104
哪一个鼠标键和光标共同作用呢？ 104
用光标标注图形输出 105
画方框 105
在图像上使用 Cursor 命令 106
在循环中使用 Cursor 命令 107
从显示中删除注释 108
删除注释的异或法 108
删除注释的设备拷贝法 110
画一个橡皮筋方框 112
图形窗口的滚动 113
Z 图形缓冲区中的图形显示技巧 114
Z 图形缓冲区的实现 114
一个 Z 图形缓冲区实例：两个曲面 115

使 Z 图形缓冲区成为当前设备 115
配置 Z 图形缓冲区 115
将物体装入到 Z 图形缓冲区中 116
对投影面进行拍照 116
在显示设备上显示结果 116
Z 图形缓冲区的一些奇怪特点 116
用 Z 图形缓冲区使图像变形 117
Z 图形缓冲区中的透明效果 119
将 Z 图形缓冲区效果与体数据着色相结合 120

第六章 在 IDL 中读写数据 122

本章概要 122

打开文件进行读写 122
查找和选择数据文件 122
选择文件名 123
选择目录名 123
寻找文件 123
构造文件名 124
获取逻辑设备号 124
直接使用逻辑设备号 124
让 IDL 管理逻辑设备号 125
判断哪些文件和哪些逻辑设备号相连 125

读写格式化数据 125
写自由格式文件 125
读自由格式文件 126
读取自由格式文件的规则 126
读写自由格式文件的实例 128
读一个简单数据文件 128
写列格式数据文件 129
读列格式数据文件 130
创建读列格式数据的模板 131
用确定的文件格式写入 132
一些共有的格式说明符 132
写用逗号分隔的确定格式数据文件 132
读出用逗号分隔的确定格式文件 133
从字符串中读取格式数据 133

读写二进制数据 133
读取二进制图像数据文件 134
写二进制图像数据文件 134
读取带有文件头的二进制数据文件 135
二进制数据文件的一些问题 136
用关联变量存取二进制数据文件 136
关联变量的一些优点 137
定义关联变量 137

读写常用文件格式的文件 138
创建彩色 GIF 文件 139
写 GIF 文件 139
读 GIF 文件 139
创建彩色 JPEG 文件 140
写 JPEG 文件 140
读取 JPEG 文件 141
查询图像文件信息 141

第七章 图形硬拷贝输出 142

本章概要 142

选择图形硬拷贝输出设备 143

配置图形硬拷贝输出设备 143
常用的 Device 命令关键字 144
创建 PostScript 文件 145

将图形送到硬拷贝设备中 146

打印 PostScript 文件 147
在运行 MacOS 系统的计算机上打印 PostScript 文件 147
在 Windows 计算机上打印 PostScript 文件 147

生成封装的 PostScript 文件输出 148
封装 PostScript 图形的预览 148

生成彩色的 PostScript 输出 149
PostScript 中的彩色图像与灰度图像 149
真彩图像 149

在 PostScript 设备上创建高质量的输出 150
显示设备和 PostScript 设备之间的相同点 150
显示设备与 PostScript 设备之间的不同点 151
问题：PostScript 窗口可能会有不同的纵横比例 151
解决方法：让图形窗口的纵横比保持不变 151
问题：PostScript 设备有更高的显示分辨率 152
解决方法：不用设备坐标来定位图形 152
问题：PostScript 设备能使用不同的显示字体 153
解决方案：仔细设计和定位文字 153
问题： PostScript 设备使用背景颜色和绘图颜色时的不同 155
解决方法：理解 PostScript 如何处理背景颜色和绘图颜色 155
问题：PostScript 设备的颜色数目多于显示设备 156
解决方法：在 PostScript 输出中确保恰当地缩放数据 157
问题：PostScript 设备显示图像时的不同 158
解决方法：使用 TV 命令设置图像大小 160

在横向输出模式中计算 PostScript 的偏移量 163

用 PS_Form 配置 PostScript 设备 164

配置和使用打印设备 164

用打印设备定位图形.....166
用打印设备输出图像.....167

第八章IDL 编 程 基 础 168

本章概述 168

编写 IDL 批处理文件 168

编写 IDL 主程序 169
过程和与函数中变量的作用范围.....171
创建位置参数.....171
定义可选的或必须的位置参数.....172
定义关键字.....173
使用缩写关键字.....173
定义可选择的关键字.....174
关键字定义了吗？.....174
处理具有双重属性的关键字.....175
创建输出型参数.....176
用引用和传值的方法传递信息.....176
参数存在吗.....178

编写 IDL 函数 178
方括号和函数的调用.....180
用 Forward_Function 命令保留函数名.....180

使用程序控制语句 180
IDL 中表达式的真和假.....181
将多个语句处理成单个语句.....181
If…Then…Else 控制语句.....182
条件表达式.....183
FOR 循环控制语句.....183
WHILE 循环控制语句.....183
REPEAT...UNTIL 循环控制语句.....184
CASE 控制语句.....184
GOTO 控制语句.....185
错误处理控制语句.....185
ON__IOError 控制语句.....185
ON_Error 控制语句.....185
Catch 控制语句.....186
错误处理语句的优先级.....187

编译和执行 IDL 程序模块 187
程序编译规则：.....188
程序编译和自动运行规则.....188
特殊编译命令.....188

第九章编 写 IDL 程 序

..... 190

本章概述 190

基本的 ImageBar 程序 190

给程序 ImageBar 增加一个“先擦除”功能 193

向 ImageBar 程序增加颜色敏感功能 194

给 ImageBar 中的命令传递关键字 195

使用关键字继承 196

根据窗口大小改变字符大小 197

程序 ImageBar 的最终代码 198

实用程序的特点 199

在图形用户界面中包装 ImageBar 199

第十章编 写 简 单 的 组 件 程 序

..... 201

本章概述 201

组件程序的结构 201

组件程序如何对事件作出反应 202

编写组件定义模块 202

定义和创建程序组件 203

创建顶层 base 组件 203

创建菜单栏按钮 204

为程序创建图形窗口 204

在屏幕上实现组件 204

使绘图组件成为当前图形窗口 205

在绘图组件窗口上显示图形 205

保存程序运行时所需要的信息 205

使用组件用户值保存程序信息 206

创建事件循环和注册程序 206

运行程序 207

创建无阻塞组件程序 207

编写事件处理模块 207

事件结构中的公共字段 207

事件处理函数 208

将事件处理程序和组件联系起来 209

编写 Quit 按钮的事件处理程序 210

编写改变图形窗口大小的事件处理程序 211

进行小量地修改 212

添加颜色敏感 212

采用更高效的内存管理 213

使用指针储存程序信息 215

第十一章组 件 编 程 技 巧 216

本章概述 216

改变颜色表 216

保护公共块 217

一个可选择颜色表的工具 217

一个关键字继承的问题 218

编写颜色表工具的事件处理程序 219

指定 Group Leader 220

给组件程序增加 Group Leader 221

在 24 位显示器上改变颜色表 221

创建事件并将事件传递给其他程序 222

在组件程序中使用指针 223

使用 Cleanup 过程防止内存泄露 225

使用伪事件进行程序通信 226

创建一个具有“记忆功能”的程序 227

保护组件程序的颜色 229

通过组件跟踪事件来保护颜色 230

通过绘图组件事件来保护颜色 231

保存或者发布程序的图形 231

第十二章对 话 框 程 序 235

本章概述 235

创建模式对话框 235

阻塞的组件程序 235

模式组件程序 236

编写模式对话框的定义模块 236

定义一个顶级的模式 base 237

定义其他组件 238

在模式对话框中保存信息 238

创建 Info 结构 238

创建一个阻塞组件 239

从阻塞中返回 239

编写模式对话框的事件处理模块 240

测试模式对话框程序 241

创建非模式的对话框 241

编写非模式对话框程序 241

通报程序事件的组件 243

编写非模式对话框的事件处理模块 244

将事件发送给其他组件 244
测试非模态对话程序 245

附录 A组 件 的 事 件 结 构 247

事件结构的定义 247
公共字段的定义 247

基本组件的事件结构 247
base 组件 247
按钮组件 247
绘图组件 247
下拉式列表组件 248
标签组件 248
列表组件 248
滑动条组件 248
表单组件 248
插入单个字符事件 248
插入字符串事件 249
删除字符串事件 249
选择文本事件 249
选择单元事件 249
改变行高事件 249
改变列宽事件 249
无效数据事件 250
文本组件 250
插入字符事件 250
删除字符串事件 250
文本选择事件 250

复合组件的事件结构 250
CW_Animate 250
CW_Arcball 251
CW_BGroup 251
CW_Clr_Index 251
CW_Color_Sel 251
CW_DefROI 251
CW_Field 251
CW_Form 251
CW_Flisder 251
CW_Orient 252
CW_PDMenu 252
CW_RGBSlider 252
CW_Zoom 252

组件程序的事件结构 252
Xcolors 252

其他组件的事件结构 252
键盘焦点事件 253
组件退出请求事件 253
组建计时器事件 253
组件跟踪事件 253
附录 B数 据 文 件 描 述 254

前 言

第一章 起 步 篇

本章概述

本章意在解释这本书的写作目的及读者通过阅读本书能学到什么，并告诉读者如何能更方便地使用本书中所涉及的 IDL 实例。我们希望读者能通过本章掌握以下几点：

1. 本书是如何组织的；
2. 怎样使用本书；
3. 如何下载和使用本书所涉及的 IDL 文件；
4. 如何使用 IDL 的变量、关键字和命令；
5. 如何创建并运行 IDL 的矢量和数组；
6. 如何使用 IDL 的图形窗口。

撰写本书的背景

本书是笔者在多年来为科学家和工程师培训如何使用和操作 IDL（Interactive Data Language）的基础上创作的，而且笔者绝大部分时间是为 IDL 的开发者 Research Systems 公司工作。当笔者在回答了一个又一个问题之后，意识到多数问题属于同一类型。事实上，大多数人想用 IDL 做许多同样的事情。大家都非常关注的是如何分析和演示数据、写出高效率的程序来解决科学问题，并且都要求快速便捷地完成工作。而多数人并不想做的事情是阅读大量的计算机软件使用手册。IDL 是一套大型软件，并且还在不断发展壮大，随之而来的是大量的相关文档资料。笔者知道很少有人愿意去花大量时间来读这些资料。如果让某人独自探索 IDL 的奥秘，那么对 IDL 而言将是件可怕的事情，甚至对有经验的用户来说也是一样。本书意在使读者掌握 IDL，教给读者在日常运行 IDL 时所必需的 80%的知识。更为重要的是，本书的大量实例使 IDL 更容易理解，本书将用实际的例程演示如何使用 IDL。

本书的读者定位是 IDL 初学者，特别是那些需要自学 IDL 的读者。另外，深入掌握 IDL 需要一段相对长的时间，多数人只能利用业余时间学习 IDL，笔者想写一本能满足这两类人学习 IDL 的书。总之，本书为不喜欢读软件手册并想通过例子学习 IDL 的人全面介绍 IDL 的精髓。本书在 IDL 编程技术和技巧方面做了一些简要概述，而深入掌握和领会这些技术还

需要通过大量的编程实践。无论如何，这是一本笔者本人在初学 IDL 时所期望的教材。

如何使用本书

笔者曾试图使本书每章都能具有独立性，这样读者能拿起本书就可翻到任何一章去学习最需要的知识。但在安排本书章节时，或多或少是根据笔者在 IDL 教学时的顺序来安排的。对于 IDL 的初学者，按照本书的章节顺序从头开始学完本书将更合理。书中后面的几章编程教程是建立在前面几章中讲过的概念和技巧的基础上的。

所需的 IDL 版本

希望读者在学习本书时使用的是最新的 IDL 版本。本书写作时使用的是 IDL5.2 版。使用较早版本的用户可以使用本书中的大部分例程，但笔者没有试图使本书中的例程与更早的 IDL 软件版本兼容。特别是，较早版本的用户在使用长文件名（如果在 Windows 环境下）、指针（必须用句柄代替它）以及方括号来引用数组下标（必须用圆括号代替它）时存在困难。如果需要升级软件，可以从 Research Systems 公司的 WWW 网址http://www.rsinc.com/上查找关于 IDL 的最新版本和当地 IDL 代理商的信息，包括如何升级软件的信息。

IDL 运行期间所需颜色的数量

书中例程是假设 IDL 在 256 种颜色模式下运行编写的，使用的是我们通常称为索引颜色的模式（详细细节请参考 83 页的“IDL 的颜色运用”章节）。这意味着所显示的颜色是索引号或是与彩色表相关的颜色，这样当彩色表中的颜色变化时，所显示的颜色也一同变化。启动 IDL 并在 IDL 命令行键入如下 IDL 命令，就能发现当前所用的颜色模式数。

```
IDL>Print, !D.N_Colors
```

当!D.N_Colors 的值大于或等于 256 时，仍然能够使用本书中的例子，但需要对源代码做一点改变。大多数人使用的颜色值都小于 256。比较典型的颜色值介于 200 与 245 之间。笔者推测在本书中至少要用 150 种颜色。那就是说，!D.N_Colors 的值至少应在 150 ～ 256 之间。

少于 150 种颜色该怎样?

如果在 IDL 运行中少于 150 种颜色，并且计算机运行在公用桌面环境（CDE），可以不将

CDE 环境下的颜色数设置为“高”。设置为“中”或“低”的情况下，程序将运行良好。读者可以在视窗环境操作手册的在线帮助中查找如何改变这个设置。

如果不是用的共用桌面环境，颜色数也少于 150，并且不是在 PC 机或 Macintosh 计算机上运行 IDL 的话，那么很可能运行了其他应用程序，该应用程序使用了要分配给 IDL 的颜色值。网页浏览器很可能就是这样的应用程序。退出当前任务，重新登录，并在重新登录后最先启动 IDL。键入以上命令，如果仍然得到少于 150 种的颜色，那么需要联系 Research Systems 公司的技术人员，以获取更多的帮助。

如果颜色数少于 150 种，并且是在 PC 或 Macintosh 计算机上运行的 IDL，那么，检查显卡以确保设置为 256 色。一般可通过显示器的控制面板完成。细节参考计算机文档资料。

多于 256 种颜色该怎样?

如果在 IDL 运行中使用了多于 256 种颜色，并且 IDL 是运行在 X Window 环境下的计算机上，可以让 IDL 使用 8 位的假彩色显示级别。

退出 IDL，并重新启动 IDL。在做任何操作之前，键入以下命令：

```
IDL>Device, Pseudo_Color=8, Decomposed=0
```

为了确认 IDL 在使用 8 位假彩色显示级别，键入：

```
IDL>Help, /Device
```

所显示的信息使读者确信使用的是假彩色显示级别，并且所使用的颜色数为 256 或少于 256。如果想使用本书中的例子，每次进入 IDL 时都需要键入 DEVICE 命令。可以将此命令放在 IDL 启动文件中。查看 IDL 文档资料以获取更多的详细资料。

如果在 IDL 运行中多于 256 种颜色，并且是在 PC 或 Macintosh 计算机上运行 IDL，需要检查显示卡的设置参数以确保设置为 256 色。一般通过显示器或显示面板完成。细节参考计算机文档资料。修改参数后必须重新启动 IDL。

如果喜欢在 16 位或 24 位的颜色模式下工作，那么键入以下命令以确保颜色分解关键字 Decomposed 已被关闭：

```
IDL>Device, Get_Visual_Depth=thisDepth
IDL>IF thisDepth GT 8 THEN Device, Decomposed=0
```

如果在这种模式下对颜色表做些修改，记住这些修改不会在显示窗口中立即更新。必须在显示窗口中刷新图形以查看这些颜色改变是否起作用。细节参考 83 页的“IDL 的颜色运用”章节。

创建 IDL 的启动文件

记住，每次启动 IDL 来使用本书的命令时，都必须执行以上命令。为此，可以将这些命令输入到 IDL 的启动文件中。当每次 IDL 启动时，启动文件中的命令都被执行，这就像在 IDL 命令提示符下键入这些命令。为了解如何在使用的计算机中创建 IDL 开始文件，可在 IDL 命令行键入以下命令，以获取在线帮助：

```
IDL>?  Startup
```

本书的风格习惯

笔者尽量用统一的风格贯穿全书，目的是更方便读者阅读。首先，本书中在 IDL 命令行或 IDL 编辑器窗口所键入的命令总是以 Courier 字体形式来书写，如：

```
Surface, data
```

在 IDL 命令行键入的命令都显示在 IDL 提示符“IDL>”的后面，如：

```
IDL>Surface, data
```

其他的 IDL 命令都是在文本编辑窗口键入的。可以选择自己的文本编辑器或使用 IDL 提供的文本编辑器，这由读者决定。

大写

本书中常用大写形式来书写 IDL 程序。这种形式完全是任意的。 IDL 对字母的大小写并不敏感，但与操作系统打交道的命令（例如：UNIX 操作系统对 IDL 所打开的文件名的大小写敏感）和执行字符串比较命令时除外。大写可以有助于记住命令和关键字名，并且一目了然地知道命令行中哪些单词是函数名。

本书中所有 IDL 函数和关键字的第一个字母用大写。此外，任何有助于记忆的字母也用大写。例如：

```
Surface, data, charsize=2.0, Color=180
XLoadCT
Widget_Control, tlb, Set_UValue=info, /No_Copy
```

变量名的第一个字母没有用大写字母,但是当变量名中的字母有可能构成单词时使用大写。例如：

```
data=FIndGen(11)
buttonValue=thisValue
ptrToData=Ptr_New()
```

IDL 的保留字全部用大写字母，例如：

```
REPEAT test UNTIL
FOR j=0,10 DO BEGIN
ENDWHILE
```

在 IDL 命令行或文本编辑器上，当键入命令时，可以随意使用大写字母。

注释

在 IDL 命令中，分号（“;”）右边的任何文本都被视为是注释，IDL 解释器将忽略它。简言之，可在 IDL 的程序中写入注释。通常在分号的前后加上空格，并让注释行缩进三个空格。例如：

```
   ; This is the loop part of the program.
FOR j=0,10 DO BEGIN
  data=j*2
  count=count ＋j
ENDFOR
```

偶尔，会在命令行的末端看到一个注释，这是在定义 IDL 结构变量的字段时，特别这样做的，目的是对所定义的字段进行说明。例如：

```
info={r:r,$    ; The red color vector
      g:g,$    ; The green color vector
      b:b,}    ; The blue color vector
```

续行符

IDL 中的续行符是表示美元的符号“$”。这表示 IDL 语句延续到下一语句行（见上例）。在本书中将看到很多续行符。建议在 IDL 命令行中不使用续行符，应该在 IDL 命令行输入完整的 IDL 命令。IDL 命令行将忽略续行符。例如，可以用如下方法键入上述命令：

```
IDL>info={r:r, g:g, b:b}
```

在输入错误或需要修改命令时，这将使得重新键入命令变得更加简单。

有时需要完全按照书中出现的 IDL 语句输入。笔者将告知什么情况下这样做。当在 IDL 命令行想键入 For 循环时就需要这样做。在命令行中一次键入多行命令是非常便捷的做法。必须让 IDL 解释器认为这些语句为一个命令。这就需要在 IDL 的命令行上正确使用续行符($)和多行命令符（&）。

本书中所用的 IDL 程序和数据文件

笔者为使用本书的读者准备了许多 IDL 程序和数据文件。IDL 程序文件以“.pro”为扩展名，数据文件的扩展名一般为“.dat”，另外还有一些扩展名为“.txt”的文本文件。

安装程序和数据文件

建议读者创建一个名为 coyote 的子目录，并把所有的程序、文本、数据文件都放在其下。可以将 coyote 子目录放到 IDL 主目录下或其他地方（让 IDL 内部的系统变量!Dir 指向这个目录）。读者可以通过设置 IDL 的 Preferences 对话框中 Startup 工作路径来指向这个目录。

获取 IDL 的主目录和当前目录

如果不知道 IDL 的主目录的位置。启动 IDL，键入以下命令：

```
IDL>CD, Current=homeDirectory
IDL>Print, homeDirectory
```

当前目录不一定是主目录。在 IDL 运行期间，可以用同样的命令随时获得当前目录：

```
IDL>CD, Current=currentDirectory
IDL>Print, currentDirectory
```

注意，如果按上述做法装载数据文件时遇到问题，请确保是在所希望的目录下。不用 IDL 主目录（例如： Windows 版的 IDL5.2 软件中，IDL5.2 就是 IDL 的主目录）作为工作目录是一个好办法，因为这样很容易删除重要文件。

下载本书所用的程序和数据文件

书中文件可以通过 internet 以匿名 FTP 方式登录下载。在使用的网络浏览器中键入 Coyote's Guide to IDL Programming 的连接，网址是：

http://www.dfanning.com/

如果用匿名 ftp，文件可以在如下网址找到：

ftp://ftp.frii.com/pub/dfanning/outging/coyote

用文本或 ASCII 模式下载所有的程序和文本文件（例如：那些带.pro 或.txt 扩展名的文件），用 BINARY 模式下载所有的数据文件（例如：那些带.dat 扩展名的文件）。为方便起见，下载压缩文件 coyotefiles.zip 就可一次性地将所有的程序、文本文件和数据下载下来。

确保 Coyote 目录在 IDL 的搜索路径内

无论在什么地方创建 coyote 目录或储存本书的例程文件，需要确保这个目录在 IDL 搜索路径中。在 IDL 中，路径用!path 系统变量指定。以后将学到更多关于该系统变量的作用，但目前读者只需知道它指定的是一系列子目录，当 IDL 遇到不认识的子程序时就到这些子目录中查找相应的子程序。打印该系统变量可以看到当前的 IDL 搜索路径：

```
IDL>Print, !path
```

如果使用的是 PC 机，这些子目录用分号隔开；在 Macintosh 或 VMS 机器上，它们用逗号隔开；在 UNIX 机器上，它们用冒号隔开。

如果想在 IDL 搜索路径中添加 coyote 目录，只要在 IDL 的当前目录的 coyote 目录下键入 AddPath 命令即可（如果没有创建 coyote 目录，可以将 IDL 的当前路径改变为存放本书文件的目录名，然后键入 AddPath 命令）。使用 CD 命令来转换到 IDL 的当前的目录。例如，如果 coyote 目录是 IDL 主目录下的一个子目录，并且这个主目录是当前目录，可以键入如下命令来在 IDL 的搜索路径中添加 coyote 目录：

```
IDL>CD, 'coyote'
IDL>AddPath
```

如果每次运行 IDL 时都想进入 coyote 目录（或本书文件所在的目录）并且运行 AddPath 程序，也许会想到将该命令添加到 IDL 启动文件中（详细细节参考第四页的“创建 IDL 的启动文件”）。或者，想到将 coyote 目录永久性地添加到 IDL 的搜索路径中。（这取决于使用的操作系统和 IDL 的配置文件。关于设置!Path 系统变量，可参考 IDL 的在线帮助）

拷贝数据文件

如果愿意，可从计算机上其他地方拷贝本书所用到的 IDL 数据文件，不必通过匿名的 ftp 来下载。为此，可使用 CopyData 命令，这个命令是刚下载的文件之一。进入 coyote 目录（或书中文件所在的目录），如果使用的是 IDL5 版，只需键入 CopyData：

```
IDL>CopyData
```

如果运行的是 IDL 更早的版本，可以通过 Demo 关键字为 CopyData 程序提供 IDL 的演示目录（演示目录名在 IDL 先前版本中各不相同，而且不一定被安装）。如在 PC 机上演示目录经常命名为“C：\RSI\IDLDEMO4”。所以应该键入如下命令：

```
IDL>CopyData,Demo="C:\RSI\IDLDEMO4"
```

数据文件将从不同的地方被选出并拷贝到当前目录上。本书附有这些数据文件的一个列表，说明了它们的类型和大小。见 313 页的“附录 B：数据文件描述”。

获取更多的帮助

当在安装这些程序文件或在 IDL 编程的其他方面需要帮助时，查看 Coyote's Guide to IDL Progamming 网页。将找到关于本书和 IDL 基础编程的信息。如果问题还没有得到解决，也可以在那里看到一张表格，通过该表格可以直接和笔者联系。Fanning 软件顾问和 Coyote's Guide to IDLProgramming 的网址为：

http://www.dfanning.com/

使用 IDL 命令

本书是一本实践性很强的书。当阅读它时，笔者更希望读者坐在电脑前，而不是坐在火炉前。笔者希望读者键入命令并查看发生了什么。为此，本书前半部分的多数命令需要在 IDL 命令行上键入（如果想保存所键入的命令，可以创建一个日志文件来记下它们。参考第 11 页的“创建命令日志”）。

随着 IDL5.0 的问世，IDL 已越来越像程序语言了。例如，对象图形引擎并不真正地用来在 IDL 命令行上使用，而是专门设计用在 IDL 编程中。但是从命令行键入 IDL 命令中能学到很多东西。特别是，能学会画出和测试一些东西，并且可以用数据文件做实验， 这些被称为“循序渐进”，是学习 IDL 的最好方法之一。

下面是刚开始所必需知道的。首先将看到本书中有许多类似下面的命令：

```
Contour,peak,lon,lat,XStyle=1,YStyle=1, /Follow,$
   Levels=vals,C_Label=[1,0,1,0,0,1,1,0]
```

如果知道所看到的东西是什么将非常有助于学习。

IDL 命令解析

在上面的命令中，单词 Contour 是 IDL 的一个子程序。它必须被完整地拼出。一些子程序或函数名会很长，但不能缩写。命令行中 peak、lon、及 lat 是变量。它们可以用来将信息传入或传出给命令或程序。XStyle,YStyle,Follow,Levels 以及 C_Lables 为关键字。一般来讲关键字对命令来说是可选的。如同变量，它们用来将信息传入或传出 IDL 命令或 IDL 程序。

位置参数

以上命令中的三个变量 peak,lon,及 lat 称为位置参数。在这个特殊例子中，这些位置参数为输入变量（例如，它们把数据传入命令），但从上述命令中并不能辨认出它们是不是输入变量。它们也可以简单地用作输出变量（或者，在某种情况下，它们既可以是输入变量也可以是输出变量）。其命令行语法完全一样。只有通过上下文，通过阅读关于这类命令或程序的公开文档才能辨别。

一个位置参数在命令名的右边有其确定的顺序。（注意，以下讨论的关键字参数不会影响位置参数的顺序）。在这个例子中，peak 变量必须在 Contour 命令右边，在 lon 变量的左边。lon 变量必须在 peak 变量的右边，lat 变量的左边。不能遗漏第二个参数，只给定第一和第三个位置参数。

例如，下面这两条命令的格式是不正确的，会导致错误。第一条命令的位置参数顺序被改变，第二条命令遗漏了第二个位置参数。

```
Contour, lon, peak, lat, XStyle=1, YStyle=1, /Follow, $
   Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]
Contour, peak, , lat, XStyle=1, YStyle=1, /Follow, $
   Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]
```

一般情况下，命令的位置参数必须给定参数，但并不总是如此。例如，在上面正确的命令中，peak 是 Contour 命令必需的参数，但是 lon 和 lat 是可选位置参数。

关键字参数

Xstyle、Ystyle、Follow、Level 和 C_Labels 都是关键字参数。与位置参数不同，关键字参数能以任何顺序出现在命令名右边。它们甚至能出现在位置参数中间而不影响位置参数之间的相对位置。换句话说，关键字参数不能像位置参数那样对待。以下的 Contour 命令是个有效构造。

```
Contour, peak, Level=vals, lon, XStyle=1, YStyle=1, $
   /Follow, lat, C_Lavels=[1,0,1,0,0,1,1,0]
```

一般情况下，关键字参数是可选参数。像位置参数一样，它们也能成为命令的输入变量或输出变量。读者将通过本书或阅读命令的文档得知这一点。

注意在上列命令中关键字的使用方法。关键字能设置为一个特定值（例如，XStyle=1），一个变量（例如，Levels=vals），一个数组（例如，C_Labels=[1,0,1,0,0,1,1,0]），甚至可以用一个斜杠字符来设定（例如，/Follow）。

注意最后的一条语法。有些关键字有二进制特性。换句话说，它们是 on/off, yes/no, true/false, 1/0，等等。能经常发现这些关键字通过/Keyword 这种语法来设置或打开。语法/Keyword 等同于语法 Keyword=1。

事实上，以上 Contour 命令能被写成这样：

```
Contour,peak,Levels=vals,lon,/XStyle,/YStyle,$
   /Follow,lat,C_Lavels=[1,0,1,0,0,1,1,0]
```

这个命令和上面的命令是一回事。命令不能写成这样的原因是，它可能错误地暗示了 X 轴和 Y 轴关键字有二进制特性，然而事实并非如此。它们能被设置为除 0 和 1 以外的其他值。

IDL 过程和函数

这个特殊的命令 Contour 是一个 IDL 过程。IDL 命令要么是过程，像这个命令一样，要么是函数。如下的 IDL 命令 BytScl 就是一个函数：

```
scaled=BytScl(image, Top=199, Min=0, Max=maxValue)
```

注意 Contour 过程和 BytScl 函数的不同。首先，在函数命令中，位置参数和关键字放在一对圆括号中的。在过程命令中，参数和关键字仅排列在一个命令行上。但是，最重要的区别是 IDL 函数会返回一个值，等号左边的一个变量用于返回该值。这是 IDL 中函数命令和过程命令根本的区别。

函数命令总是返回一个值，这个数值必须赋给一个变量。函数返回值可能是任一种 IDL 变量，包括数值、数组或结构。在这个例子中，返回值 scaled 是一个与 image 位置参数具有相同维数的字节型数组。

有时可以看到一个函数和过程写在一起，例如，考虑一下这两个命令：

```
scaled=BytScl(image,Top=199,Min=0,Max=maxValue)
TV,scaled
```

第一个命令是一个函数命令，另一个是过程命令，此过程使用函数的返回值作为其位置参数，两个命令写成如下形式在 IDL 中很常见：

TV,BytScl(image,Top=199,Min=0,Max=maxValue)

在这种情况下，BytScl 命令首先被执行并得到一个返回值，此返回值作为 TV 命令的位置参数。

花一些时间熟悉各种 IDL 子程序，就能立即识别哪个是过程，哪个是函数，但尽量记住这一点：当正在从一个命令中寻找某个值时，要想到这个命令可能是一个函数。在本书后面将学会怎样编写 IDL 过程和函数。

用 IDL 命令帮助

IDL 有全面的在线帮助系统，能为读者提供有关 IDL 命令和参数的非常有帮助的信息。可以通过在 IDL 命令行中输入一个问号，或在 IDL 开发环境下拉菜单中选择 Help 菜单项获得在线帮助。IDL 文档集中的大部分信息都可通过在线帮助获得。为了获得 IDL 在线系统帮助，仅仅需要在 IDL 的命令行中输入一个问号，如下：

```
IDL>?
```

创建命令日志

如果希望将在命令行里面输入的命令保存为日志或记录，则需要创建一个日志文件。日志文件是一个 IDL 批处理文件（参考 205 页的“编写 IDL 批处理文件”）。在 IDL 中用 Journal 命令打开一个日志文件，并指定想打开的文件名。该文件将是一个用于写信息的新文件。从 IDL 命令行不能添加日志文件。例如，为了写一个命名为 book_commands.pro 的日志文件，键入：

```
IDL>Journal, 'book_commands'
```

随后所有在 IDL 命令行上键入的命令都将写入这个日志文件。

```
IDL>a=[3,5,7,3,6,9]
IDL>Help, a
IDL>Plot, a
```

当想关闭日志文件时，再次在 IDL 命令行键入 Journal 命令，如下：

```
IDL>Journal
```

日志文件是能编辑的一个简单的 ASCII 文本文件。如果愿意，可用任何一个文本编辑，包括由 IDL 的 PC 版本附带的编辑器。当想再次执行日志文件中的命令时，在 IDL 命令行键入@作为开头字母。例如，要执行在上面 book_commands.pro 文件中的命令，如下：

```
IDL>@book_commands
```

确定创建的每个日志文件有惟一的名称，不能添加日志到这些日志文件。所以，如果第二次建立的日志文件名和第一次相同，许多操作系统将会毫无警告地覆盖第一个日志文件。

如果每次建立日志文件时都想要一个惟一文件名，可用下列的 IDL 程序完成：

```
PRO Journal_Unique
    Journal, String('journal_',Bin_Data(SysTime()),'.pro',$
    Format='(A,I4,5I2.2,A)')
END
```

然后，用 Journal_Unique 代替 Journal，就可以建立每次都具有惟一文件名的日志文件。

创建变量

在这本书中将创建许多变量。如果以前对变量有所了解将会大有益处。变量名必须以字

母开头。它们可以包括其他字母、数字、下划线、美元符号。一个变量名最长可达 255 个字符。本书的习惯是让变量名的首写字母小写。下面是一些有效的变量名：

```
ptrToData
image2
this_image
a$handle
```

变量名有两个重要属性：数据类型和组织结构。数据类型指出属于数据类型中的哪一种。在 IDL 中有 14 种基本数据类型。在图表 1 中将看到每一种数据类型，每个类型创建的变量的字节大小、变量创建方式、数据类型之间强制转换的 IDL 函数名称。除了数据类型外，一个变量有一个组织结构。有效的组织结构有标量（例如单个数值）、矢量（真正的一维数组）、数组（最高可达八维）和 IDL 结构（能包含各种数据类型的变量和组织结构，结构中独立的组成部分称为字段）。

表 1　IDL 中的 14 种基本数据类型。表中显示了每种数据类型的字节数，创建变量的方法，数据类型之间强制转换的 IDL 函数

数据类型	字节数	创建变量	数据类型函数
字节型	1	Var=0B	thisVar=Byte(variable)
16 位有符号整型	2	Var=0	thisVar=Fix(variable)
32 位有符号长整型	4	Var=0L	thisVar=Long(variable)
64 位有符号整型	8	Var=0LL	thisVar=Long64(variable)
16 位无符号整型	2	Var=0U	thisVar=UInt(variable)
32 位无符号长整型	4	Var=0UL	thisVar=ULong(variable)
64 位无符号整型	8	Var=0ULL	thisVar=Ulong64(variable)
浮点型	4	Var=0.0	thisVar=Float(variable)
双精度浮点型	8	Var=0.0D	thisVar=Double(variable)
复数	8	Var=Complex(0.0,0.0)	thisVar=Complex(variable)
双精度复数	16	Var=Dcomplex(0.0D,0.0D)	thisVar=DComplex(variable)
字符串	0-32767	Var=''或 Var=""	thisVar=String(variable)
指针	4	Var=Ptr_New()	None
对象	4	Var=Obj_New()	None

正如我们所看到的，IDL 是一个擅长于处理矢量或数组数据的软件，所以有大量的 IDL 命令用于创建不同数据类型的矢量和数组。特别是，有许多创建各类数据类型的数组的函数，该数组的每个元素的初始值为零，而且还有许多创建各类数据类型的数组的函数，该数组的每个元素的初始值为其在数组中的索引位置。在表 2 中将看到这些函数列表。例如，创建 100x100 初始值为零的字节型数组，输入：

```
IDL>array=BytArr(100,100)
```

创建一个有 100 个元素的浮点型矢量，初始数值为从 0 到 99，输入：

```
IDL>vector=FIndGen(100)
```

读者将在本书中看到使用这些 IDL 函数的各种方式。

动态改变变量的属性

IDL 最强大的功能之一是大多数过程或函数都能在任何数据类型或组织结构上生效。这是因为 IDL 在运行时能改变变量的数据类型和组织结构（像世界上其他强大的事物一样，这种动态改变变量的属性的能力也有潜在的巨大危险！必须小心，确信知道正在使用哪种数

据)。例如，在 IDL 中，本质上讲变量是毫无意义的（像在 Fortran 或者 C 程序中），因为这种变量的数据类型很容易改变。例如：

```
num=3            ; Initialize NUM as a scalar integer.
num=num*5.2      ; Variable NUM changes to a float!
```

变量 num 被初始化为一个整数，由于数学运算的结果和重新赋值，它被动态地改变成浮点数值。这是因为 IDL 在数学计算当中为了保证最高的精度，将低精度的数据类型提升为高精度的数据类型。当 num 被再赋值（在等号的左边），它被提升为一个浮点数去保持等号右边计算的精度。思考下面这个例子：

```
result=4*x
```

在这种情形下，不可能知道变量会产生哪种数据类型和组织结构，因为对 x 变量一无所知。事实上，结果主要取决于变量 x 的数据类型和数据结构。如果 x 是 10 个元素的浮点矢量，结果将会是 10 个元素的浮点矢量。如果它是 100 x 200 的长整数数组，结果也将是 100 x 200 的长整数数组。注意，如果 x 有一个字节的数据类型，那么结果将是一个整数数据类型（在这种情形下，组织结构并没有多大影响）。这是由于和整数相乘的结果。

记住等号右边的表达式总是在将数据类型和组织结构赋予等号左边的变量前计算的。IDL 将变量提升到能保持表达式的计算精度的数据类型。

注意整型变量

关于整型变量笔者想简单地提一下，以免使用它们时遇到麻烦。有两种常见的错误。第一种涉及到整数数学操作。思考一下这个示例：

```
result=12/5
```

也许期望的是一个值为2.4浮点变量,但是结果却是一个值为2的整数。知道为什么吗？这是因为方程式右边的两个数字为整数。这是一个整数除法的例子。当然，找到这个问题的原因并不难，但有时由整型数据导致的问题会更微小，以致于难以发现。

例如，假如想知道 IDL 图形窗口的比率。窗口的大小（像素点或整数值）被储存在两个系统变量中。也许会写出如下的 IDL 代码：

```
aspect=!D.X_Size / !D.Y_Size
```

它可以花掉很长的时间找出为什么比率为零。正确的方法是写出一些代码以强制将一个整数值变成一个浮点，如下：

```
aspect=Float(!D.X_Size) / !D.Y_Size
```

现在的比率变量就是一个所期望的浮点数了。

表 2　IDL 函数可以创建矢量和多维数组，并将其每个元素初始为 0 或为它们本身的索引号码

数据类型	初始化函数	产生索引值的函数
字节型	BytArr	BIndGen
16 位有符号整型	IntArr	IndGen
32 位有符号长整型	LonArr	LIndGen
64 位有符号整型	Lon64Arr	L64IndGen
16 位无符号整型	UIntArr	UIndGen
32 位无符号长整型	ULonArr	ULIndGen
64 位无符号整型	ULon64Arr	UL64IndGen
浮点型	FltArr	FIndGen
双精度浮点型	DblArr	DIndGen
复数	ComplexArr	CIndGen
双精度复数	DComplexArr	DCIndGen
字符串	StrArr	SIndGen

指针	PtrArr	None
对象	ObjArr	None

另外一个使用整型变量时常遇到的问题是没有意识到 IDL 的整型变量在其他编程语言中被称为短整型。或者说，IDL 的一个整型只有两个字节长。整型在其他程序语言中有四个字节（四个字节的整数在 IDL 的整数中是一个长整数）。

两个字节的整数只能大到 32767。大于这个值通常由于“溢出”而被 IDL 当作为负数。用短整数会在两种情况下遇到麻烦。首先，在循环中没有考虑到短整数的因素，例如，假如想读一个数据文件，但不知道有多少行。可以写入如下代码：

```
count=0
WHILE NOT EOF（lun） DO BEGIN
    READF, lun, temp
    data(count)=temp
    count=count+1
ENDWHILE
```

如果数据文件多于 32768 行，这个代码就失败了。原因是 count 变量初始为一个短整数，这个代码更好的写法如下：

```
count=0L
WHILE NOT EOF(lun) DO BEGIN
    READF,lun, temp
    data(count)=temp
    count=count +1L
ENDWHILE
```

现在随便读取多少行都可以。

另外一个常犯这种错误的地方是在For循环中。最好是按如下写法来写出For循环命令：

```
FOR j=0L, num-1 DO...
```

第二种在使用短整型时可能会遇到麻烦是在读取用其他编程语言生成的数据时（或者反过来）。如果读取用 C 或 Fortran 程序生成的整型数据，应该确保在 IDL 中用长整型来读这些数据。同样，应该用长整型数据来写那些将被 C 或 Fortran 程序视为整型来读入的文件。

使用矢量和数组

IDL 是面向矩阵的，是一种擅长于处理矢量和数组的程序语言（IDL 的第一个版本的原形是 APL，是一种在数组运算上非常优秀的程序语言）。要成为一个高效的 IDL 程序员，必须知道怎样对数组进行数学运算。在本书中，将看到许多这方面的例子，但在开始前，需要注意两个重点。

创建矢量

在 IDL 命令行中，可以用一对方括号创建一个矢量（矢量是指一维的数组）或一个数组，如下：

```
IDL>vector=[1,2,3]
```

这是一个整型矢量，因为数据值为整型值。

可以用 Help 命令，获取关于数据类型和变量结构的信息，如下：

```
IDL>Help,vector
VECTOR     INT     =Array[3]
```

如果想从第四个元素增加到矢量中，在 IDL 中可以很轻松地完成。只需键入：

```
IDL>vector=[vector,4]
IDL>Print,vector
      1    2    3    4
```

数组下标的应用

假设打算在数组的第二和第三个元素之间添加另外一个元素，数组下标可以帮助完成。数组下标的上界和下界被冒号隔开。例如，指定上述矢量的前三个元素，如下所示：

```
IDL>Print, vector(0:2)
       1          2          3
```

注意，在 IDL 中矢量和数组下标的起始值是 0，而不是 1，并且引用矢量下标时使用圆括弧以示区别。这使得有时很难将一个函数调用和一个数组下标引用区别开来。为了解决这个问题，IDL 允许使用方括弧来引用数组下标。也就是说，当运行 IDL5.0 以上版本时，可以键入：

```
IDL>Print, vector[0:2]
```

本书中使用方括弧引用下标，以避免同函数调用相混淆。倘若正在使用 IDL 的 IDL4.x 版本，要运行此命令就得用圆括弧代替方括弧。

要用数组下标将另一个元素插入第二和第三个元素之间，可键入：

```
IDL>vector=[vector[0:1],5,vector[2:3]]
IDL>Print, vector
      1    2    5     3    4
```

矢量也可用上表中讲到的数组来创建函数。例如，建立一个值在 0 到 50 之间的 6 个元素浮点矢量，可键入：

```
IDL>vector=FIndGen(6)*10
IDL>Print,vector
      0.000000   10.0000   20.0000   30.0000 40.0000   50.0000
```

数组的建立

数组也可以在 IDL 命令行中建立。例如，可以建立一个两行三列的数组，如下所示：

```
IDL>array=[[1,2,3],[4,5,6]]
IDL>Print, array
```

输出 IDL 输出窗口中将会如下所示：

```
        1    2    3
        4    5    6
```

注意，这等同于先建立一个矢量，然后用 Reform 命令将此变形为一个三行二列的数组，如下所示：

```
IDL>vector=IndGen(6)+1
IDL>array=Reform(vector,3,2)
IDL>Print, array
```

这表明矢量和数组均是以行的顺序存储在 IDL 中的。这一点在编写 IDL 程序的过程中非常重要，因为将经常用到 IDL 这种数据存储方式的优势。

数组中元素的存取

假设想读出刚才建立的数组中位于第一列第二行的元素（元素的值为 4），可以键入：

```
IDL>Print,array[0,1]
```

注意，下标的顺序先是列标，后是行标。这正好与已习惯的线性代数中的矩阵或行列式相反（同时，行标与列标比想象的小 1，因为排列下标值的起始值是 0 而不是 1）。

列-行下标源于对海量图像数据处理的要求，IDL 最初就是为处理这种数据而开发的。数据中的一行对应图像的一个独立扫描行。这种数据存储形式使数据操作迅速而精确。当然，也有其他的软件刚好相反，采用行优先的方式，这就要求读者在不同软件之间转换数据时考虑到这一点。

可以使用一维下标来存取该数组中的每一个元素。要知道数组元素是以行顺序存储的，所以获得数组中的第四个元素。可以键入以下语句来存取：

```
IDL>Print, array[3]
```

用一维下标可以存取多维数组中的元素，这是 IDL 语言的一个强大的工具。

也可以用一维向量来做数组的下标。例如，倘若要存取数组中的第一，二，四和第六个元素，可键入：

```
IDL>indices=[0,1,3,5]
IDL>Print,array[indices]
```

矢量和子数组的提取

IDL 可很容易地从数组内提取出矢量和子数组。例如：查看这个拥有随机数据的数组：

```
IDL>data = RandomU(seed, 10, 20)
```

想提取出第 6-10 列和第 12～15 行的数据，可键入：

```
IDL>subarray = data[5:9, 11:14]
```

如果要将第 8 列的数据画出来，可以使用下标“*”代表所有的行，如下所示：

```
IDL>Plot, data[7,*]
```

要取出一个第 14 行的矢量，键入：

```
IDL>vector = data[*,13]
```

要提取出一个数据为数组中最后 5 行的数组，键入：

```
IDL>subarray = data[*,15:19]
IDL>Help, subarray
```

现在可以看到子数组是一个 10 列 x 5 行的数组。

同样可以用“*”代表剩下的所有数据。例如，用数组的最后 5 列建立一个子数组，也可键入：

```
IDL>subarray = data[5:*,*]
IDL>Help, subarray
```

通过对本书中范例的练习，读者会对数组以及数组的处理方法了解得更多。

使用 IDL 图形窗口

通过对本书中范例的练习，读者会更多地了解 IDL 的图形窗口。但在开始之前，最好先了解下面一些东西。

图形窗口的建立

首先，可直接用 Window 命令建立一个图形窗口，或是在没有窗口打开的情况下，间接通过运行图形显示命令来打开。例如，可以建立并启动一个窗口，只须键入：

```
IDL>Window
```

注意，此窗口的标题栏中有一个 0，这是此窗口的索引号。当图形窗口建立后，每个图形窗口都有惟一的一个图形窗口索引号。Window 命令如果没有任何位置参数总是创建出索引号为 0 的图形窗口，被称为“窗口 0”。在 IDL 的一次运行中，最少可同时打开 128 个图形窗口，可以为 0 到 31 号图形窗口指定一个索引号。对于 32 到 127 号图形窗口，可以用 Window 命令带上 Free 关键字(以下将谈到)来创建，IDL 将赋予为它们索引号。例如：想创建一个索引号为 10 的图形窗口，键入：

```
IDL>Window, 10
```

倘若某个索引号图形窗口的窗口已经存在，再用 Window 命令创建相同索引号图形窗口，Window 命令将首先删除旧窗口，然后建立一个带有此索引号的新窗口。

如果愿意（当在 IDL 程序中建立窗口时，这通常是一个不错的主意），可以用一个未用的索引号或者已经打开但是空白窗口的索引号来创建新的图形窗口。关键字 Free 即为此目的而设，如下所示：

```
IDL>Window, /Free
```

用关键字 Free 建立的图形窗口，将会具有一个大于 31 的索引号。关键字 Free 是建立索引号大于 31 的常规图形窗口的惟一途径。

确定当前图形窗口

现在在显示器上至少已经打开了三个图形窗口，但只有一个是当前图形窗口。当前图形窗口用于接受图形命令的输出结果。当前图形窗口的索引号总是存储在!D.Window 系统变量中。如果没有创建和打开图形窗口，系统变量!D.Window 的值为 -1。

可以创建一个图形窗口，并存储其图形窗口索引号，以便以后删除该窗口或使其成为活动窗口。可键入：

```
IDL＞Window, /Free
IDL> thisWindowIndex = !D.Window
```

使图形窗口成为当前窗口

为使一个窗口成为当前图形窗口（可在其内显示图形），可使用 Wset 命令和图形窗口索引号来设定。例如，希望当前图形窗口为 10 号窗口时，键入：

```
IDL＞Wset, 10
```

随后所有的图形命令的结果都将显示到 10 号窗口内。

注意，当一个图形窗口创建完成后，该窗口即成为当前窗口（但是，用 Widget_Draw 产生的窗口不是这样）。为了在某个窗口内绘制图形，该窗口必须是当前图形窗口。

删除图形窗口

可用 Wdelete 命令和图形窗口的索引号删除图形窗口。被删除的图形窗口不必是当前图形窗口。例如，删除窗口 10，键入：

```
IDL＞WDelete, 10
```

删除当前显示器上的所有图形窗口有一个技巧：

```
IDL＞WHILE  !D.Window NE -1 DO Wdelete, !D.Window
```

图形窗口的位置和尺寸

在创建图形窗口时，图形窗口的位置和尺寸是根据内部运算规则确定的。在 Window 命令中，用关键字可以设置图形窗口的位置和尺寸。例如，用关键字 XSize 和 YSize 创建一个宽 200 像素，高 300 像素的窗口，键入：

```
IDL＞Window, 1, XSize=200, YSize=300
```

可用相对于显示器左上角的像素坐标或设备坐标来确定窗口位置。例如，用关键字 XPos 和 YPos 将窗口的左上角位置于显示器（75，150）处，键入：

```
IDL>Window, 2, XPos=75, YPos=150
```

将图形窗口设置到显示器最前面

创建一个图形窗口时，该窗口拥有输入焦点，同时也成为当前图形窗口。也就是说，对于窗口管理器来讲，该图形窗口现在为激活窗口（仅仅因为一个图形窗口拥有窗口输入焦点，并不意味它是当前图形窗口）。为了输入一个命令，不得不将窗口焦点移回到命令输入窗口。在某些平台上，特别是在 PC 机上，这会导致图形窗口隐藏到其他窗口后面。

有时，在显示器上一个图形窗口隐藏在其他窗口的后面，想将该窗口拖到前面以便能看见。在不改变窗口输入焦点的情况下，要将一个图形窗口显示在前面，可用 Wshow 命令和图形窗口索引号来完成。

```
IDL＞Wshow, 1
```

注意，光标和窗口焦点仍在键入 IDL 命令的命令输入窗口或其他窗口内。

用 Wshow 命令将窗口显示在前面但并不将窗口改变为当前窗口。如果既想将该窗口拖到前面，又想将其变为当前窗口，那么可同时键入 Wshow 和 Wset 命令：

```
IDL＞Wshow, 2
IDL>Wset, 2
```

注意，如果输入不带参数的 Wshow 命令，在显示器上将使当前窗口被拖到前面。当不清楚哪个是当前图形窗口和只想将当前窗口拖到前面而不从 IDL 命令窗口移动开焦点时，这个命令是非常有用的。

```
IDL＞Wshow
```

注意，在 PC 机和 Macintosh 机器上，可以用 Alt-Tab 键或者 OPTION-TAB 键来循环选择已经在显示器上打开的窗口，让其可见并拥有窗口焦点。

在图形窗口上设置标题

有时希望在图形窗口上设置标题，而不仅仅是图形窗口索引号。可以使用 Title 关键字将标题设置到窗口上，键入：

```
IDL＞Window, Title='Example IDL Graphics Commands'
```

清除图形窗口内容

可以使用 Erase 命令清除当前图形窗口内容：

```
IDL＞Erase
```

如果想用一种特定的颜色索引号，去清除当前图形显示（如果在 24 位颜色模式下可以用一个 24 位颜色值），可以用 color 关键字。例如，可以用以下命令实现用炭灰色清除当前图形显示：

```
IDL>TVLCT, 70, 70, 100
IDL>Erase, Color=100
```

想清除非当前图形窗口（系统变量!D.Window 指向的窗口）的内容，必须使该窗口成为当前图形窗口，接着使用 Erase 命令。

第二章　简单的图形显示

本章概述

科学分析最基本的要求就是以简单的线画图、等值线图和曲面图来显示所研究的数据。读者在这一章中，将会了解在 IDL 中用这些方式来显示数据是多么容易。也将学会用系统变量和关键字来定位和标注简单的图形显示。

读者将学会如下几点：

1. 如何用 Plot 命令将数据显示为线画图。
2. 如何用 Surface 和 Shade_Surf 命令将数据显示为曲面图。
3. 如何用 Contour 命令将数据显示为等值线图。
4. 如何在显示窗口上定位显示图形。
5. 如何用公共关键字来标注和自定义图形显示。

IDL 中简单的图形显示

IDL 中简单的图形显示可认为是栅格图形的一个实例。也就是说，可用 Plot、Contour 或 Surface 命令通过某种算法来点亮显示窗口内相应的像素点而形成栅格图形。这种栅格图形没有永久性。换言之，一旦 IDL 显示图形和点亮相应的像素点后，IDL 就不知道自己做了些什么，刚才的操作并没有保存在内存中。这意味着，在用户重置图形窗口大小时，IDL 无法进行相应的响应。总之，在这种模式下图形显示不能被刷新，除非再次输入图形命令。

但是，栅格图形命令在 IDL 中被广泛应用，因为它们简单快捷。而且读者将会看到，在仔细地用栅格图形命令编写 IDL 程序时，可以克服许多与栅格图形命令相关的限制。本章将介绍一些关于如何用栅格图形命令写出可调节尺寸的 IDL 图形窗口或进行直接硬拷贝输出的必备概念。本章的图形命令都是 Research Systems 公司所说的 IDL 的直接图形。

另外一种被 Research Systems 公司称为对象图形的图形方式在 IDL5.0 中被引入。对象图形使用时相对复杂一点，但它在 IDL 编程方面更强大更灵活。对象图形不是为了在命令行使用而开发的，而是用在 IDL 的程序中，特别是用于带有界面的程序中（带有图形用户界面

的程序)。本书对对象图形法暂不做介绍，有兴趣的读者请参考 IDL 用户手册。

创建线画图

生成线画图最简单的方法是绘出一个矢量。可以用 LoadData 命令打开时序数据集。LoadData 命令是本书所带的一个 IDL 程序（详细细节参考第 5 页的“本书中所用的 IDL 程序和数据文件”)。它用来装载本书的例程中所需的数据。键入如下语句以查看所能使用的数据集：

```
IDL>curve=LoadData()
```

如果输入 LoadData 命令时忘掉了括号，需要在它正常工作前重新编译 LoadData 程序。原因是，IDL 在命令行会认为它是一个变量并进行相应地处理。重新编译后，“loaddata”这个函数名出现在 IDL 的函数名列表中。键入：

```
IDL>.Compile LoadData
```

时序数据是在 LoadData 数据列表上的第一个数据集。点击它，数据就被装入到 curve 变量中。另外一种选择第一个数据集的方法是，按如下方法使用 LoadData:

```
IDL>curve=LoadData(1)
```

要查看 curve 变量如何被定义，键入：

```
IDL>Help, curve
 CURVE       FLOAT     =Array[101]
```

将发现 curve 是一个具有 101 个元素的浮点矢量（或一维数组)。

要绘出该矢量，可键入：

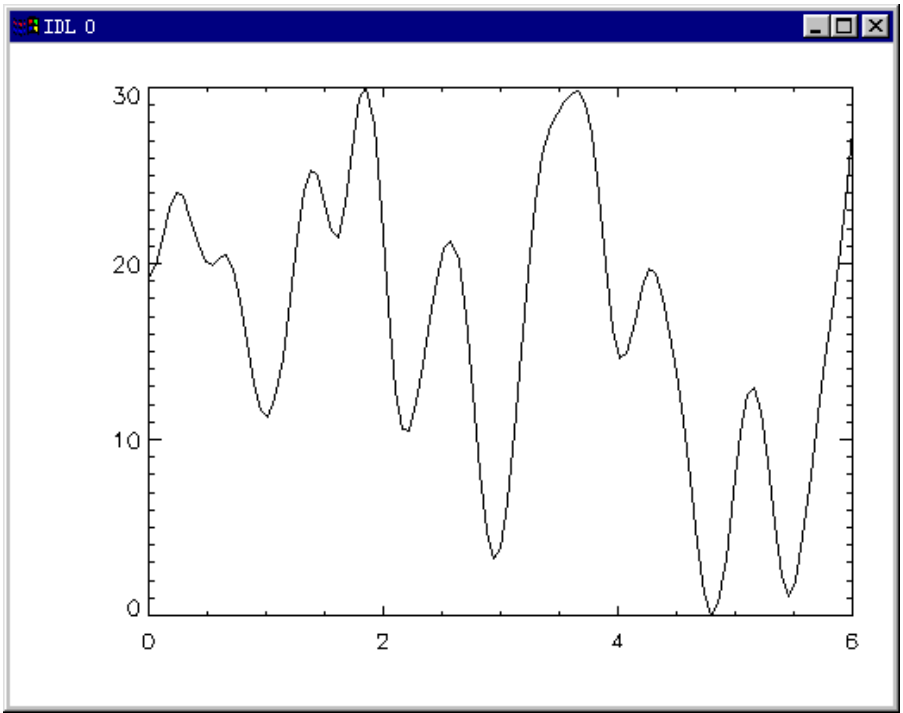
```
IDL>Plot, curve
```

IDL 试图用少量的信息尽可能地绘出漂亮的线画图。在这种情况下，x 轴或水平轴被标识为从 0 到 100，这与矢量中的元素个数相对应。而 y 轴或垂直轴则是用数据坐标来标识(它是取决于数据的坐标轴)。

但大多数情况下，线画图用于显示一组数据（独立数据）相对另外一组数据（非独立数据）的关系。例如，上面的曲线可能代表在某段时间内采集数据的信号。可能需要绘制某个时刻的信号值。在这种情况下，需要一条与该曲线矢量具有相同元素个数的矢量（这样可以获得一一对应的相关性)，并将该矢量转换为实验中所用的时间单位。例如，可以创建一个时间矢量，并绘出它与上述曲线矢量的关系图：

```
IDL>time=FIndGen(101)*(6.0/100)
IDL>Plot, time, curve
```

FIndGen 命令创建一个元素值为 0 到 100 的共 101 个元素的矢量。乘法因子按比例缩小



每个元素的大小，最后的结果是一个元素值为 0 到 6 之间的共 101 个元素的矢量。图形输出结果应与图 1 相似。

图 1　　独立数据（时间）与非独立数据（曲线）关系图

注意，在此图中的坐标轴上没有相应的标题。在图上设置标题是很容易的，只要用 XTitle 和 YTitle 关键字即可实现。例如，为此曲线图加标题，可键入：

```
IDL>Plot, time, curve, XTitle='Time Axis', $
      YTitle='Signal Strength'
```

甚至可以用 Title 关键字对整个图形设置标题，键入：

```
IDL>Plot,time,curve,XTitle='Time Axis',  $
      YTitle='Signal Strengh',Title='Experiment 35M'
```

输出结果如图 2 所示。

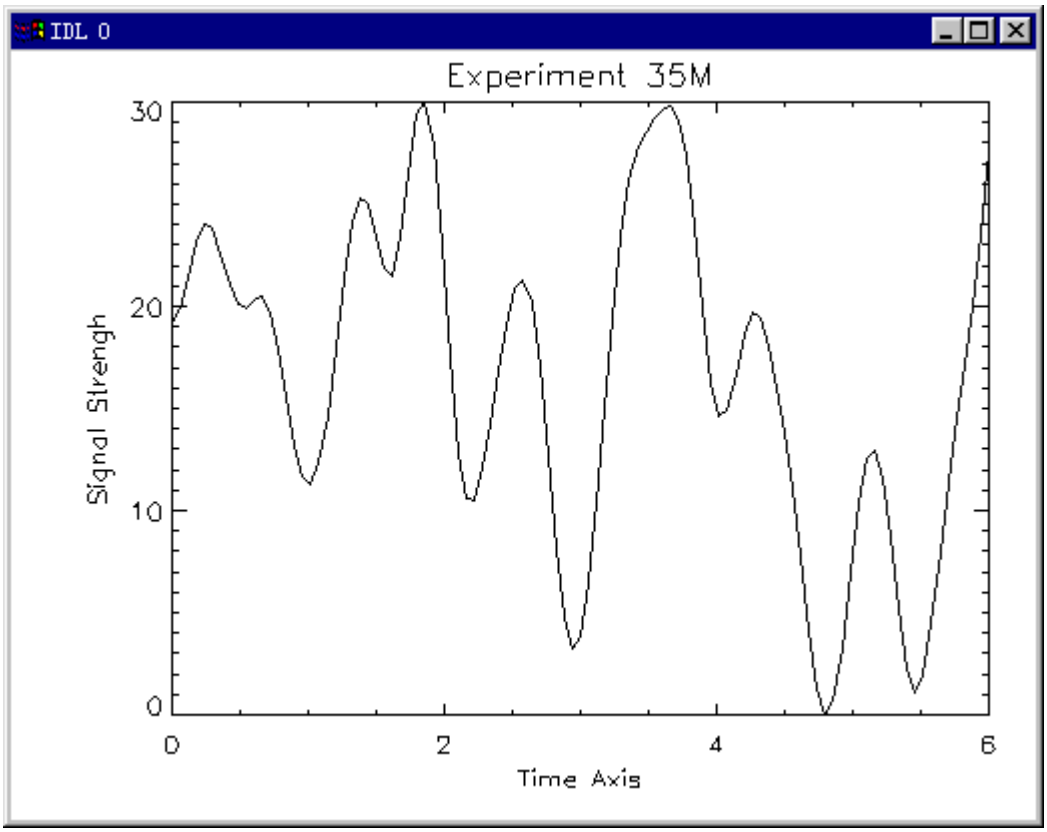


图 2　　简单的带坐标轴标题和图形标题的线画图

注意，图形显示应该为在黑色背景下的白线图，而上图显示为在白色背景下的黑线。这些插图包含在用 IDL 生成的 PostScript 文件中。一般情况下 Postscript 文件把图形颜色和背景颜色反过来显示。（参考第 189 页的“问题： PostScript 设备使用背景颜色和绘图颜色时的不同”。）

注意，图形标题稍微大于坐标轴的标题。事实上，是 1.25 倍的关系。可以用 CharSize 关键字改变所有图形注记的大小。例如，可以将坐标轴标题的字符放大 50%：

```
IDL>Plot, time, curve, XTitle='Time Axis', $
       YTitle='Signal Strength', Title='Experiment 35M', $
       CharSize=1.5
```

如果希望所有的图形显示的字符比正常情况下大，可以通过绘图系统变量上设置 CharSize 的大小，如下：

```
IDL>!P.CharSize=1.5
```

现在，所有后续的图形显示都将用较大的字符，除非用 CharSize 关键字在图形输出命令中特别地控制。

甚至可以用[XYZ]CharSize 关键字单独改变每个轴的标识字符的大小。例如，如果想使 Y 轴的注记比 X 轴的大两倍，则可键入：

```
IDL>Plot,  time,  curve,  XTitle='Time Axis',  XCharSize=1.0,  $
      YTitle='Signal Strength',  YCharSize=2.0
```

记住，[XYZ]Charsize 关键字使用当前字符的大小作为基础计算出各自的大小。当前字符的大小一般储存在!P.CharSize 系统变量中。这意味着，如果设置 XCharSize 关键字为 2，当!P.CharSize 系统变量也被设置为 2 时，字符将比平常大 4 倍。

定制线画图

上面是简单的线画图，除了数据本身外，没有多少其他信息。然而，有许多方法可用来定制和标注线画图。Plot 函数可以被 50 多种不同的关键字修饰。下面的事情也许您想做一做：

1. 改变线型或粗细；
2. 使用符号，符号之间可以有线条和没有线条存在；
3. 创建自己的绘图符号；
4. 给线图加入颜色提示重要特性；
5. 改变刻度标记的长度或刻度标记之间的间隔；
6. 使用对数来标度图形坐标轴；
7. 改变绘图范围来绘出感兴趣的数据段；
8. 删除坐标轴或改变绘图方式。

改变线条的线型和粗细

例如，想用不同的线型画出数据。如画一条线型为长虚线的线条，可以这样实现：

```
IDL>Plot, time, curve, LineStyle=5
```

对于线画图来说，可通过 LineStyle 关键字选用表 3 中列出的索引号确定不同的线型。例如，想使用虚线画出曲线，可以把 LineStyle 关键字的值设置为 2：

```
IDL>Plot, time, curve, LineStyle=2
```

表 3　　可以通过赋予 LineStyle 这个关键字不同索引号来改变线型

索引号	线型
0	实线
1	点线
2	虚线
3	划点线
4	划点点线
5	长虚线

线画图中线的粗细同样能够被改变。例如，如果想使用比正常值粗 3 倍的虚线来显示图形，可键入：

```
IDL>Plot, time, LineStyle=2, Thick=3
```

用符号代替线条表示数据

假如想用符号代替线条表示数据，就像 LineStyle 关键字一样，也存在类似的索引号供选择，以确定不同的线画图符号。表 4 给出了能通过 PSym（绘图符号）关键字来选择的索引号。例如，可以通过设置 PSym 为 2，用星号来绘图，如下：

```
IDL>plot, time, curve, Psym=2
```

输出的图形应与图 3 中的图形相似。

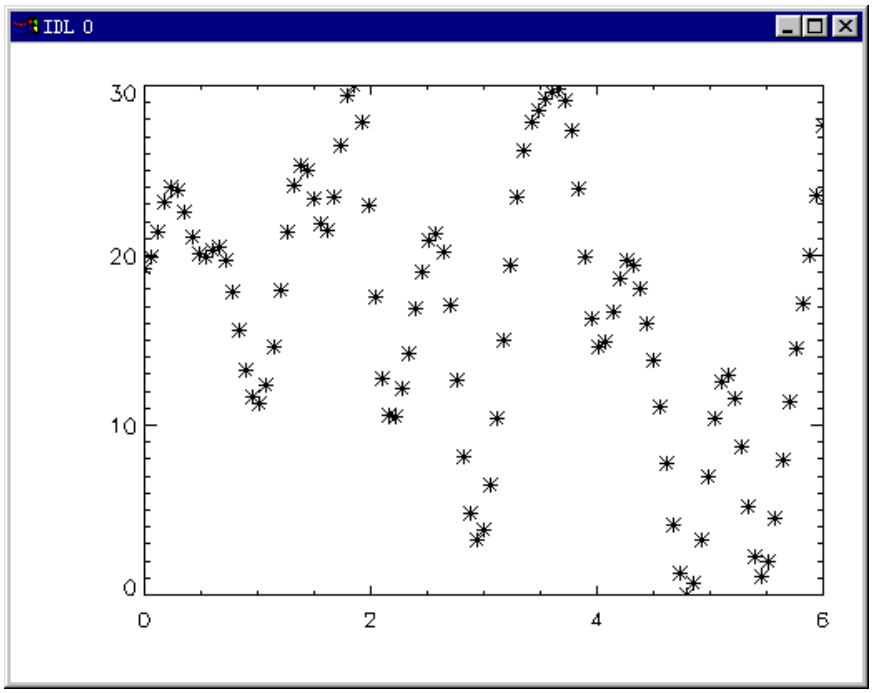


图 3　　用符号而不是线条来显示线画图

表 4　这些符号索引号可以通过 PSym 关键字来引用以便在绘图中使用不同的符号。注意，绘图符号为负值时表示用线条来连接相应的符号

索引号	绘图符号
0	无符号，通过线条连接点
1	加号
2	星号
3	点
4	菱形
5	三角形
6	方形
7	X
8	用户自定义符号（用 UserSym 过程来定义）
9	未用
10	直方图
-PSym	负值表示用线条连接相应的符号

用线条和符号来显示数据

赋予 PSym 关键字一个负值就可以用线条将图形符号连接起来。例如，可用实线与三角形符号绘出数据，键入：

```
IDL>Plot, time, curve, PSym=-5
```

为创建一个更大的符号，可用 SymSize 关键字。下面的语句画出的符号为正常的两倍。符号值为 4 时符号的大小为正常值的 4 倍，依此类推。

```
IDL>Plot, time, curve, PSym=-5, SymSize=2.0
```

创建自己的图形符号

如果您富有创造力，甚至可以创建自己的图形符号。UserSym 命令就用于此目的。在创建了一个特殊的图形符号之后，可通过设置 PSym 关键字为 8 来选择它。以下是一个创建五角星符号的例子。 x,y 矢量定义五角星的顶点，它们的值为偏离原点（0，0）的位置。可以用 UserSym 命令通过设置关键字 Fill 创建一个填充的图形符号：

```
IDL>x=[0.0, 0.5, -0.8, 0.8, -0.5, 0.0]
IDL>Y=[1.0, -0.8, 0.3, 0.3, -0.8, 1.0]
IDL>TvLCT, 255, 255, 0, 150
IDL>UserSym, x, y, Color=150, /Fill
IDL>Plot, time, curve, PSym=-8, SymSize=2.0
```

输出结果应与图 4 相似。

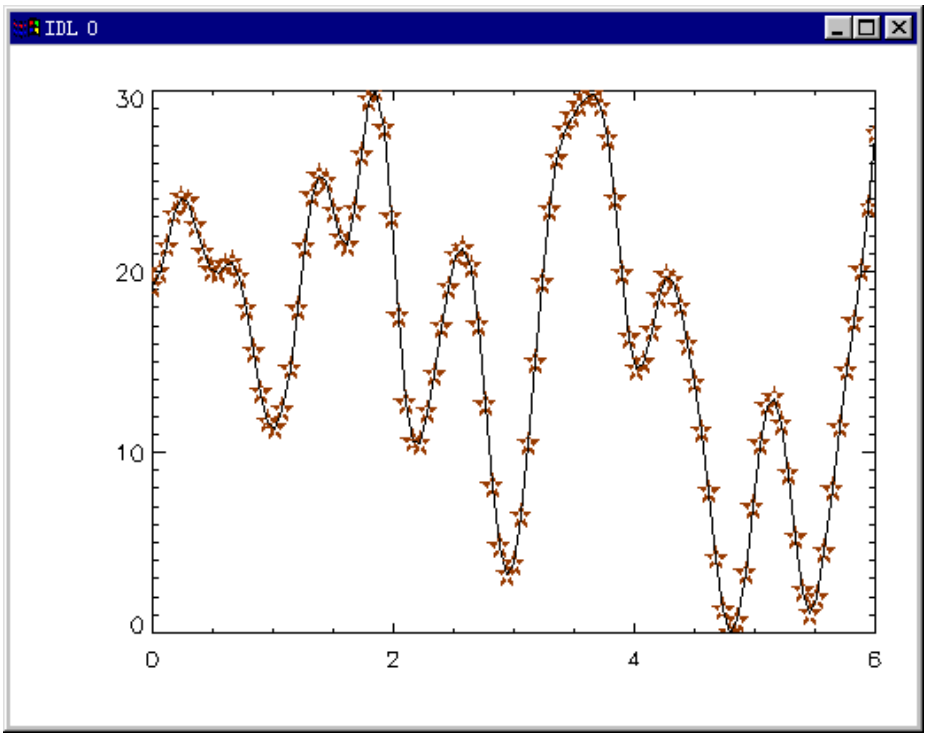


图 4　　用 UserSym 程序创建的符号来绘制线画图

用不同的颜色绘制线画图

可以用不同的颜色绘制线画图（颜色将在第 83 页的“IDL 的颜色运用”中详细讨论）。现在，只须按如下所示键入 TvLCT 命令即可，以后将学到这个命令意味着什么。实质上，该语句表示装载了三个颜色矢量，每个矢量的三个分量分别代表颜色的三个组成部分红、绿、蓝。该语句的三种颜色矢量分别表示碳灰、黄、绿色。）例如将颜色索引号 1、2 和 3 分别设置为碳灰、黄、绿色，键入：

IDL>TvLCT, [70,255,0], [70,255,255], [70,0,0], 1

在碳灰背景下绘黄色图，键入：

IDL>Plot, time, curve, Color=2, Background=1

如果只是想使线条成为不同的颜色,首先必须将 NoData 关键字打开来绘图,然后用 OPlot 命令（下面要讨论的）覆盖该图。例如,在碳灰色背景上绘制黄色外框,数据用绿色显示,键入:

IDL>Plot, time, curve, Color=2, Background=1, /NoData

IDL>OPlot, time, curve, Color=3

限定线画图的范围

并非所有的数据都必须在一个线画图中绘出，可以用关键字限定绘图的数据量。例如,可仅绘出位于 X 轴上 2 至 4 之间的数据，键入:

IDL>Plot, time, curve, XRange=[2, 4]

或者仅绘出 Y 值在 10 至 20 之间，X 值在 2 至 4 之间的部分数据图形，键入:

IDL>Plot, time, curve, YRange=[10, 20], XRange=[2, 4]

也可以通过给定关键字数据范围来反转数据的方向。例如,可将 Y 轴的 0 点设置为图形的顶端，如下:

IDL>Plot, time, curve, YRange=[30, 0]

输出结果应与图 5 相似。

如果所选择的轴的范围不适合 IDL 关于坐标轴美观标记的规定，IDL 将忽略所要求的范围。试一试如下的命令:

IDL>Plot, time, curve, XRange=[2.45, 5.64]

X 轴上显示的范围将是从 2 至 6，这并不是对 IDL 所要求的精度。为确保轴上显示的范围正如所要求的那样，可将 XStyle 关键字设置为 1，如下:

IDL>Plot, time, curve, XRange=[2.45, 5.64], XStyle=1

下一节将学到更多关于[XYZ]Style 关键字的知识。

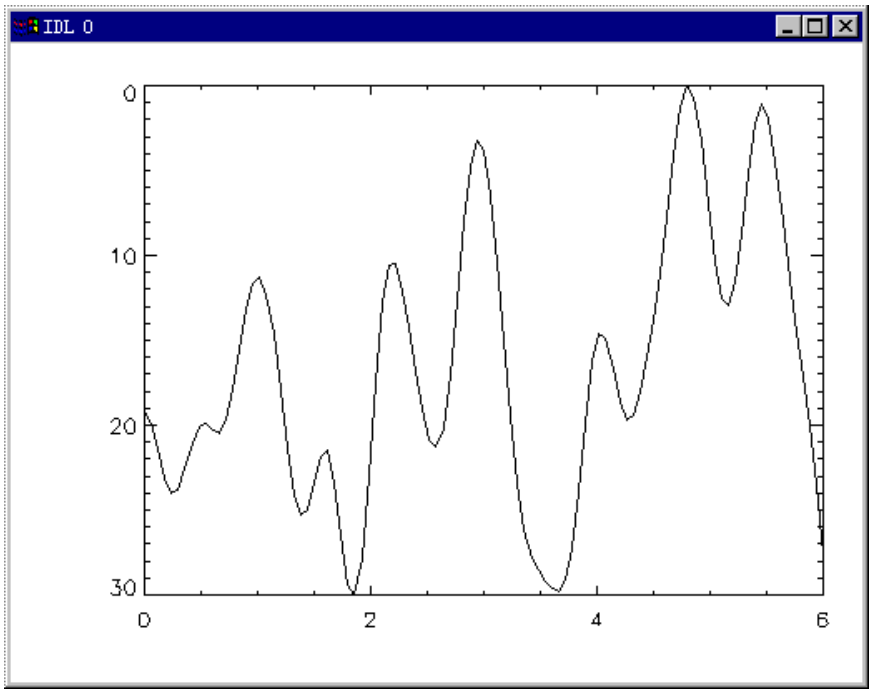


图 5　　将 Y 轴 0 点设置为图形顶端的图形

改变线画图的风格

可以方便地改变线画图的许多特性，包括它们的外观形式。例如，读者可能不需要显示线画图的方框。如果是这样，可以用[XYZ]Style 这些关键字改变线画图的特性。表 5 给出了可通过这些关键字来改变线画图风格的值。例如，为除去方框线，只留下 X 轴或 Y 轴，可键入：

```
IDL>Plot, time, curve, XStyle=8, YStyle=8
```

表 5 [XYZ]Style 关键字参数表，用于设置坐标轴的属性。注意：这些值可以累加从而设置坐标轴的多个而非单个属性

值	对坐标轴的影响
1	精确的坐标轴范围
2	扩展坐标轴范围
4	不显示整个坐标轴
8	不显示外框（只画坐标轴）
16	屏蔽 Y 轴起始值为 0 的设置（只有 Y 轴有此属性）

可以完全隐藏一个轴。例如，仅用 Y 轴显示图形，可键入：

```
IDL>Plot, time, curve, XStyle=4, YStyle=8
```

输出结果应与图 6 相似：

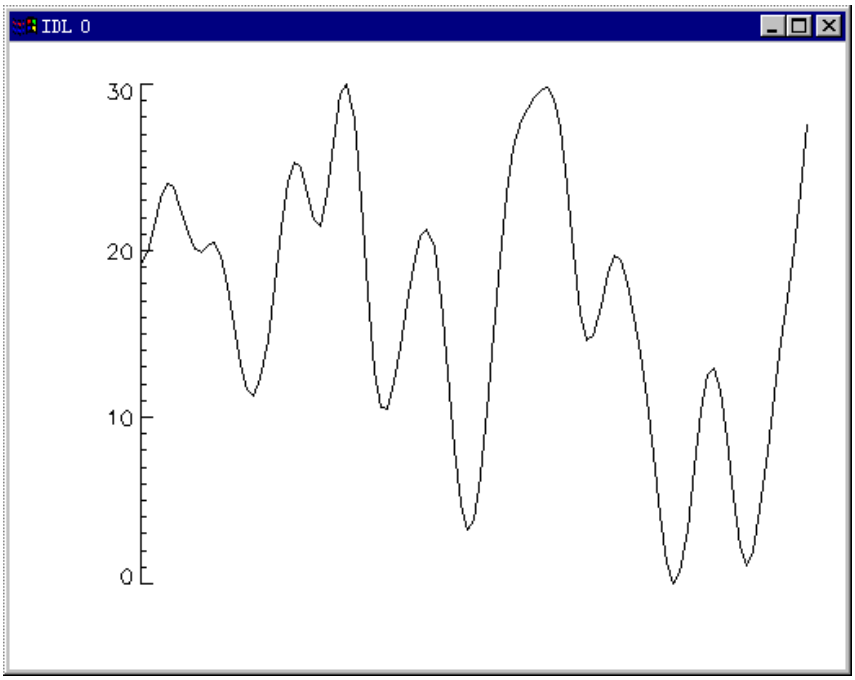


图 6 关闭 X 轴和方框只剩 Y 轴的线画图

可以用 Y 轴和 Y 方向的网格线来显示同一幅图：

```
IDL>Plot, time, curve, XStyle=4, YTickLen=1, YGridStyle=1
```

[XYZ]Style 关键字可以一次设置坐标轴的多个特性。可以通过累加适当的值来实现。例如，可以从表 5 中看出，强制使用精确的坐标轴范围的参数值为 1，而用来删除方框线的参数值为 8。为实现上述两项功能，即让 X 轴显示精确的范围又隐藏方框线，可将两个参数值相加：

IDL>plot, time, curve, xstyle=8+1, xrange=[2, 5]

在线画图上创建网格线，通常可用 TickLen 关键字来完成。如下：

IDL>Plot, time, curve, TickLen=1

将[XYZ]TickLen 关键字设置为一个负值可以创建向外的刻度标记。例如，为创建向外的刻度标记,可键入:

IDL>Plot, time, curve, TickLen=-0.03

在某个轴上创建向外的刻度标记，可将[XYZ]TickLen 关键字设置为一个负值。例如，只在 X 轴上创建向外的刻度标记，键入：

IDL>Plot, time, curve, XTickLen=-0.03

可以用[XYZ]Ticks 和[XYZ]Minor 关键字，在一个轴上选择主要的和次要的刻度标记的个数。例如，在 X 轴上创建两个主要的刻度间隔，每个主要的刻度间隔内设置 10 个次要的刻度标记，键入:

IDL>Plot, time, curve, XTicks=2, XMinor=10, XStyle=1

在线画图上绘出多种数据集

读者可以不仅仅用一组数据绘制线画图。IDL 程序允许在同一套坐标轴内显示任意多套数据。OPlot 命令就用于此。键入以下命令，输出结果应与图 7 相似:

IDL>Plot, curve

IDL>OPlot, curve/2.0, LineStyle=1

IDL>OPlot, curve/5.0, LineStyle=2

初始的 Plot 命令为以后的绘图建立数据比例(!X.S 和!Y.S 是比例参数)。或者说，!X.S 和!Y.S 系统变量告诉 IDL 如何在数据范围内取点以及如何将该点显示在设备坐标空间上。要确保初始图形有足够的轴长，以便包容以后绘制的所有图形，否则数据将被裁剪掉。可在第一个 Plot 命令中用 XRange 和 YRange 关键字来创建一个足够大的数据范围。为区别不同的数据集，可用不同的线型，不同的颜色，不同的图形符号等。Oplot 命令接受很多被 Plot 命令接受的关键字。

IDL>TvLCT, [255, 255, 0], [0, 255, 255], [0, 0, 0], 1

IDL>Plot, curve, /NoData

IDL>OPlot, curve, Color=1

IDL>OPlot, curve/2.0, Color=2

IDL>OPlot, curve/5.0, Color=3

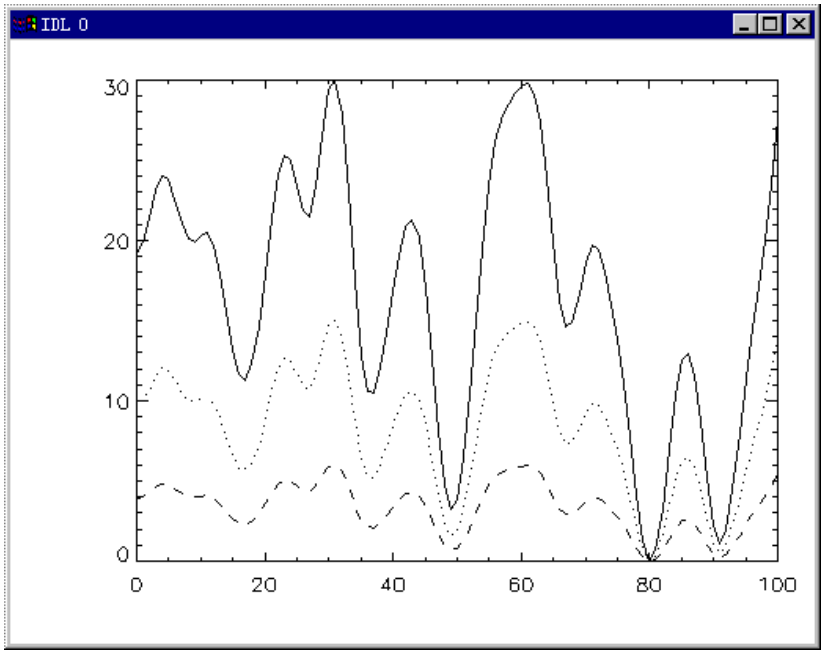


图 7　　在同一个线画图上可以绘制无限多套数据集

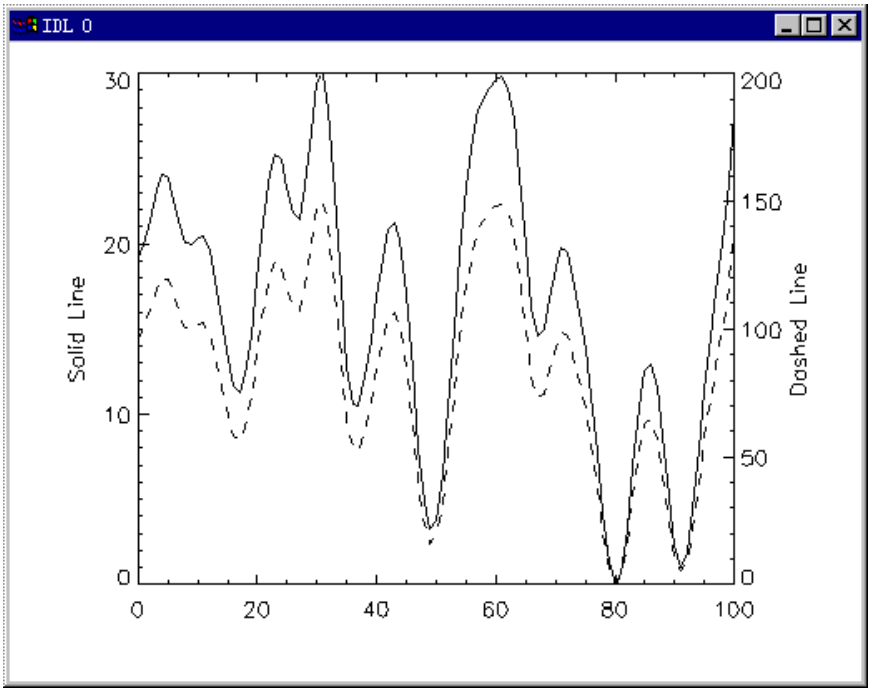


图 8　　具有两个 Y 轴的线画图。第二轴是用 Axis 命令来定位的。一定要用 Save 关键字来将数据比例保存起来

在多个轴的图上显示数据

有时，希望在同一个线画图上显示两个或多个数据集，并用不同的 y 轴表示不同的数据集。使用 Axis 命令很容易建立所需数量的坐标轴。使用 Axis 命令的关键是使用 save 关键字来存储正确的绘图比例参数（即存储在!X.S 和!Y.S 系统变量中的比例参数），以便后续图形的调用。

下面的例子在已绘出一幅图后，用带 Save 关键字的 Axis 命令建立第二个 Y 轴。OPlot 命令中的曲线将调用通过 Axis 命令保存的比例因子，以确定其在图形中的位置。正确的命令是如下：

```
IDL>Plot, curve, YStyle=8, YTitle='Solid Line', $
     Position=[0.15, 0.15, 0.85, 0.95]
IDL>Axis, YAxis=1, YRange=[0, Max(curve*5+1)], /Save, $
     YTitle='Dashed Line'
IDL>OPlot, curve*5, LineStyle=2
```

Position 关键字用来确定第一个图形在页面内的位置。为了解更多关于 Position 关键字的知识，可参阅第 48 页的“在显示窗口定位图形输出”章节。输出图形应与图 8 相似。

创建曲面图

在 IDL 程序中，任何二维的数组都可以用 Surface 命令生成一个曲面图（经过自动消隐）。首先，必须打开数据文件，用 LoadData 命令打开 Elevation Data 数据集。键入：

```
IDL>peak=LoadData(2)
```

通过键入 Help 命令，可以发现这是一个 41*41 的浮点数组。键入：

IDL>Help, peak

这个数组可以通过 Surface 命令使之曲面化：

IDL>Surface, peak, CharSize=1.5

输出结果应与图 9 相似。

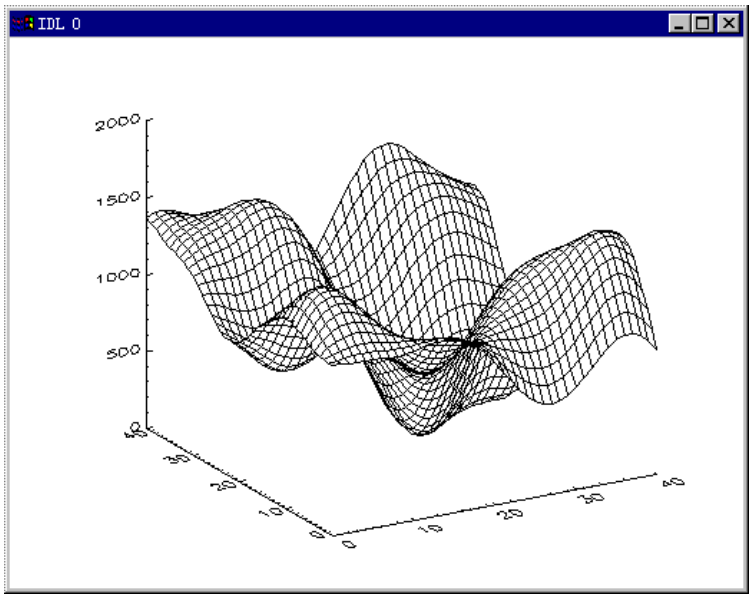


图 9　　利用高程数据生成简单的曲面图

注意，如果仅用单个数组作为变量调用 Surface 命令，它将把该数组作为其元素个数（此例在 X 和 Y 方向都为 41）的函数来绘图。（可以使用 CharSize 关键字来改变字符的大小，以便更容易看清楚）。但是，正如前面使用 Plot 命令一样，可以规定 X 和 Y 轴的数值，以便显示的图形具有实际意义。例如，X 和 Y 轴的数值可以是经纬度坐标。这里，使纬度范围为从 24 度到 48 度，经度范围为-122 度到-72 度：

```
IDL>lat=FIndGen(41)*(24./40)+24
IDL>LON=FIndGen(41)*50.0/41-122
IDL>Surface, peak, lon, lat, XTitle='Longitude', $
     YTitle='Latitude', ZTitle='Elevation', CharSize=1.5
```

输出结果应与图 10 相似。

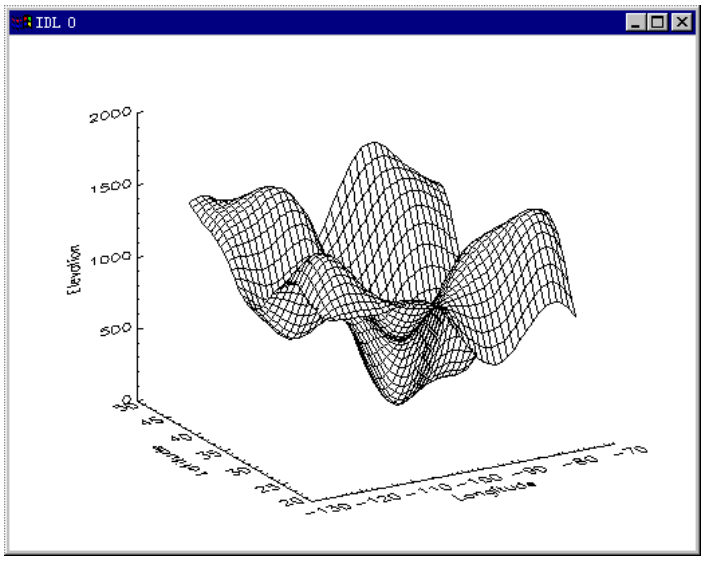


图 10　一个具有实际意义坐标值的曲面图

以上命令中的 lon 和 lat 参数是单调递增并且是规则的。它们描述了曲面网格线的位置。但网格没有必要是规则的。试想一下，如果使经度数据点不规则分布会出现什么情况。例如，可以键入以下命令模拟随机分布的经度点：

```
IDL>seed=-1L
IDL>newlon=RandomU(seed, 41)*41
IDL>newlon=newlon[Sort(newlon)]*(24./40)+24
IDL>Surface, peak, newlon, lat, XTitle='Longitude', $
        YTitle='Latitude', ZTitle='Elevation', CharSize=1.5
```

现在发现经度 X 值是没有规则分布的。尽管看起来数据被重新取样了，然而却不是。IDL 能很容易地在经度和纬度数据点指定的位置处画出曲面图的网格线。输出结果应与图 11 相似。

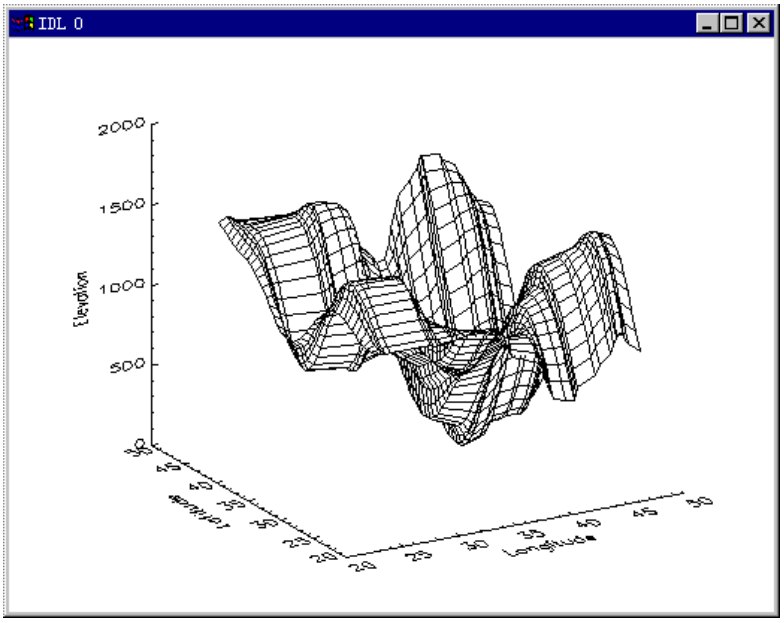


图 11　同样的曲面图，但其 X 矢量具有不规则的空间分布

定制曲面图

有 70 多个不同的关键字可以用来定制曲面图。实际上，许多关键字在 Plot 命令中已经学过。例如在上面的代码中，就使用了相同的标题关键字对曲面图的轴进行标记。然而要注意，当用 Title 关键字时，所添加的标题被旋转了，从而保证标题总是位于曲面图的 XY 平面内。键入：

```
IDL>surface, peak, lon, lat, XTitle='Longitude', $
        YTitle='Latitude', Title='Mt.Elbert', Charsize=1.5
```

但所得图形并非总是我们希望得到的。如果想使图形标题位于与显示窗口平行的平面内，就必须用 Surface 命令绘制曲面图，而用 XYOutS 命令显示标题（第 55 页有关于 XYOutS 命令的详细信息）。比如，键入：

```
IDL>Surface,  peak,  lon,  lat,  Xtitle='Longitude',  $
      Ytitle='Latitude',  Charsize=1.5
IDL> XYOutS,  0.5,  0.90,  /Normal,  Size=2.0,  Align=0.5,  $
      'Mt.Elbert'
```

旋转曲面图

在观察曲面图时可能希望能将曲面图旋转一个角度。曲面图可以用 Ax 关键字使其绕 X 轴或用 Az 关键字使其绕 X 轴旋转。当从轴上的正值向原点观察时，曲面图以逆时针方向，按某个角度值旋转。当 Az 和 Ax 关键字被忽略时其缺省值是 30 度。例如，使曲面图绕 Z 轴旋转 60 度，绕 X 轴旋转 35 度，则可键入：

```
IDL> Surface,  peak,  lon,  lat,  Az=60,  Ax=35,  Charsize=1.5
```

输出结果应与图 12 相似。

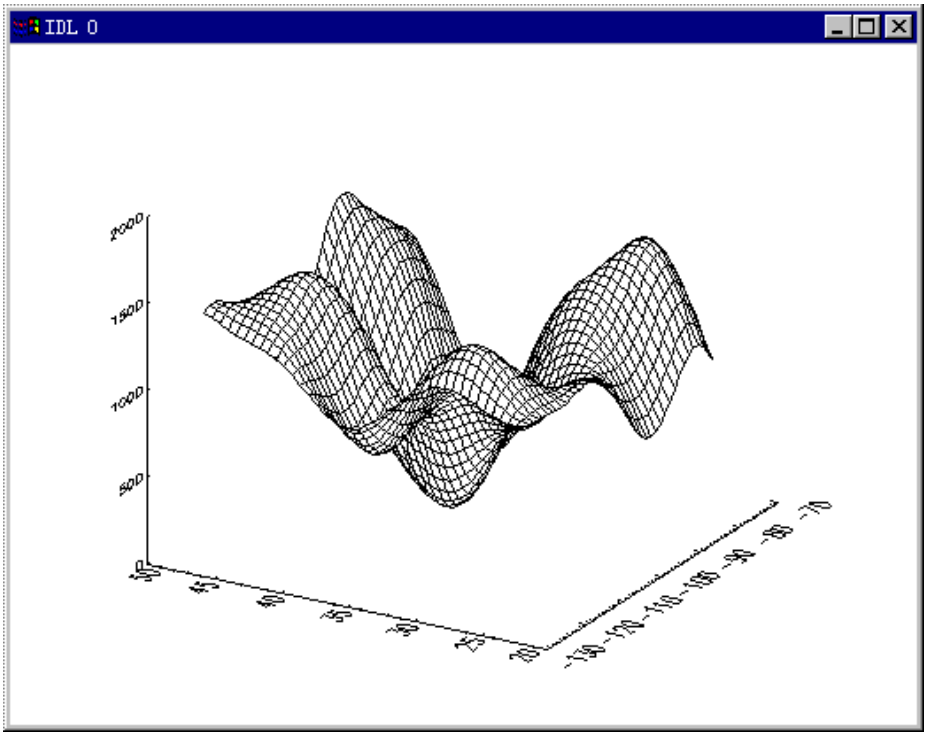


图 12　　用 Az 和 Ax 关键字使曲面图旋转

为曲面赋色

有时，可能想为曲面图上赋上颜色以强调某种特性。给曲面图着色是很简单的，只需使用在线画图中用过的赋色关键字即可。（颜色将在第 83 页的“IDL 的颜色运用”中详细讨论。现在，只须按如下键入 TvLCT 命令即可，以后将学到这个命令意味着什么。实质上，装载了三个颜色矢量，每个矢量的三个分量分别代表颜色的三个组成部分红、绿、蓝。这三种颜色矢量为碳灰、黄、绿色）。例如，在碳灰色背景上创建一个黄色的曲面图，可键入：

```
IDL> TvLCT,  [70,  255,  0],  [70,  255,  255],  [70,  0,  0],  1
IDL> Surface,  peak,  Color=2,  Background=1
```

如果想使曲面图的底面的颜色不同于顶面，比如说绿色，可以使用 Bottom 关键字来实现：

```
IDL>Surface, peak, Color=2, Background=1, Bottom=3
```

如果想将轴以不同的颜色显示，比如绿色，而不是曲面，必须键入两个命令。第一个命令使用 NoData 关键字，只将轴绘出。第二个命令是在关闭轴线后绘出曲面本身。查看第 31 页的表 5，了解[XYZ]Style 关键字的参数值及其含意：

IDL>Surface, peak, Color=3, /NoData

IDL>Surface, peak, /NoErase, Color=2, Bottom=1,XStyle=4, YStyle=4, ZStyle=4

用不同的颜色画出曲面的格网线也是有可能的，而不同的颜色代表不同的数据。比如，可用第二个数据集覆盖第一个，第二个数据集含有对第一个数据集的格网进行着色后的信息。

为了说明如何工作，可打开一个名为 Snow Pack 的数据集，并用以下这些命令将此数据作为一个曲面显示。注意，Snow Pack 数据集的大小与 peak 数据集一样，都是 41*41 浮点数组。

IDL>snow=LoadData(3)

IDL>Help, snow

IDL>Surface, snow

现在,通过用 snow 的变量值对 peak 变量的格网着色,将 snow 变量中的数据覆盖到 peak 变量中数据的上面。首先，用 LoadCT 命令装载彩色表内的一些颜色。实际的阴影处理是通过 shades 关键字完成的，如下：

IDL>LoadCT, 5

IDL>Surface, peak, Shades=BytScl(snow, Top=!D.Table_Size-1)

注意，必须用 BytScl 命令将 snow 数据集调整为使用 IDL 时的色彩数。如果调整失败，只能看到一套坐标轴，而看不到曲面显示。这是因为，数据必须调整到曲面阴影处理时所需的 0 到 255 的范围。

修改曲面图外观

有很多关键字可以用来修改曲面图的外观或形式。例如,可以显示一个带边缘的曲面图。使用 Skirt 关键字来指定边缘该画到何处。试试下面命令：

IDL>Surface, peak, Skirt=0

IDL>Surface, peak, Skirt=500, Az=60

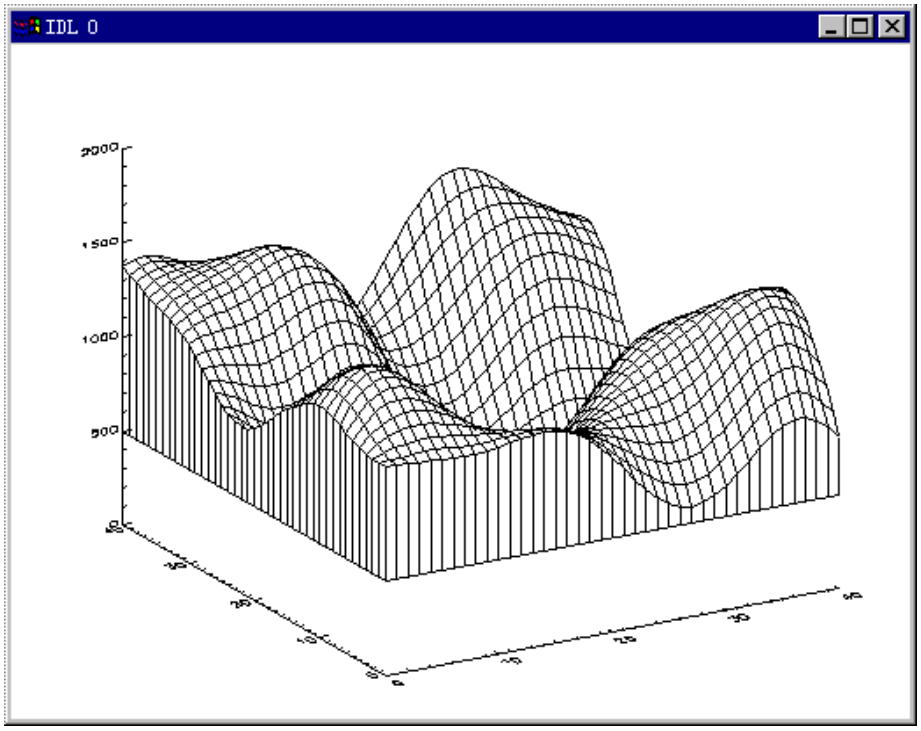


图 13　　带边缘的曲面图

上面第一个命令的输出结果应与图 13 相似。

如果仅绘出水平线，获得一种层叠线形图，比如，键入：

```
IDL>Surface, peak, /Horizontal
```

如果愿意，可以通过关键字来只显示曲面的底面或顶面，而不是两者都显示（缺省是两者都显示）。键入：

```
IDL>Surface, peak, /Upper_Only
IDL>Surface, peak, /Lower_Only
```

有时可能只想显示曲面本身，而不需要轴线。 可键入：

```
IDL>Surface, peak, XStyle=4, YStyle=4, ZStyle=4
```

创建阴影曲面图

创建阴影曲面图同样很简单，可使用 Gouraud 光源阴影算法创建阴影曲面图，键入：

```
IDL>Shade_Surf, peak
```

Shade_Surf 命令接受大多数被 Surface 命令接受的关键字。例如，如果想旋转阴影曲面，可以键入：

```
IDL>Shade_Surf, peak, lon, lat, Az=45, Ax=30
```

输出图形应与图 14 相似。

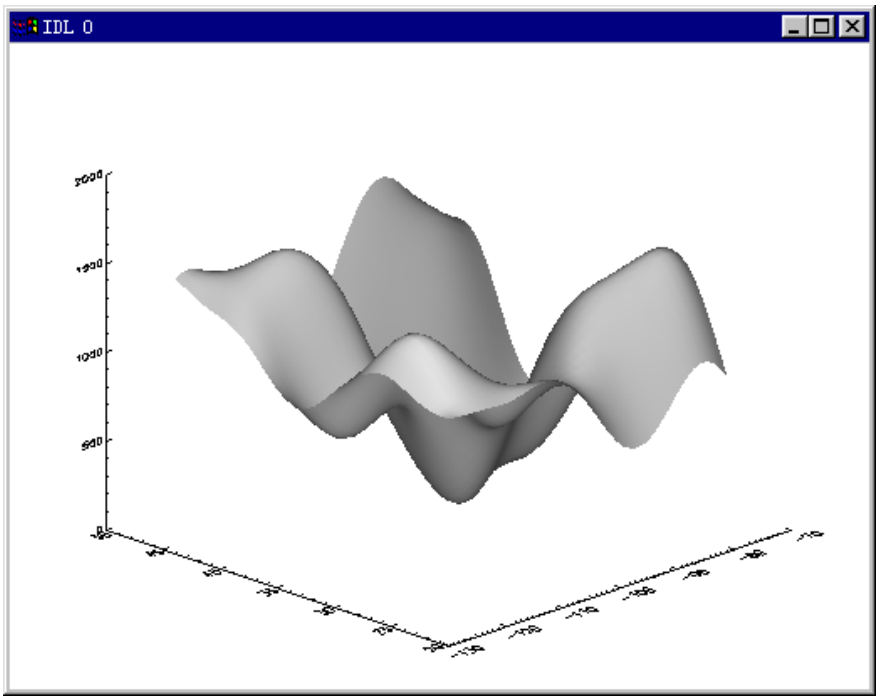


图 14 用 Gouraud 光源阴影算法生成的阴影曲面图

改变阴影处理参数

用 Set_Shading 命令可以改变 Shade_Surf 命令所使用的阴影处理参数。例如，要将光源的光线的方向从平行 Z 轴的默认值[0，0，1]改变为平行 X 轴的方向[1，0，0]，可键入：

```
IDL>Set_Shading, Light=[1, 0, 0]
IDL>Shade_Surf, peak
```

也可以从彩色表中挑选某种颜色索引号用作阴影处理。例如，想把红色彩色表（彩色表 3）装载到颜色索引号 100 到 199 之中，并将之用于阴影处理，可键入：

```
IDL>LoadCT, 3, NColors=100, Bottom=100
```

```
IDL>Set_Shading, Values=[100, 199]
IDL>Shade_Surf, peak
```

注意将光源位置和颜色参数恢复原值，否则继续操作可能会造成混乱。

```
IDL>LoadCT,5
IDL>Set_Shading, Light=[0,0,1], Value=[0,!D,Table_Size-1]
```

用其他数据集为阴影处理提供参数

首先，就象 Surface 命令一样，其他数据集也可以为阴影处理时的各数据点提供颜色值。正如前述，可以用 Shades 关键字为曲面上各点指定颜色索引号。每个像素点的阴影处理都是根据该点周围数据值通过插值求出。例如，下面是一个用 snow 变量生成的阴影曲面图：

```
IDL>Shade_Surf, snow
```

现在用这个数据集来对最初的高程数据集进行阴影处理，键入：

```
IDL>Shade_Surf,peak,lon,lat, Shades=BytScl(snow, Top=!D.Table_Size)
```

输出结果应如图 15 所示：

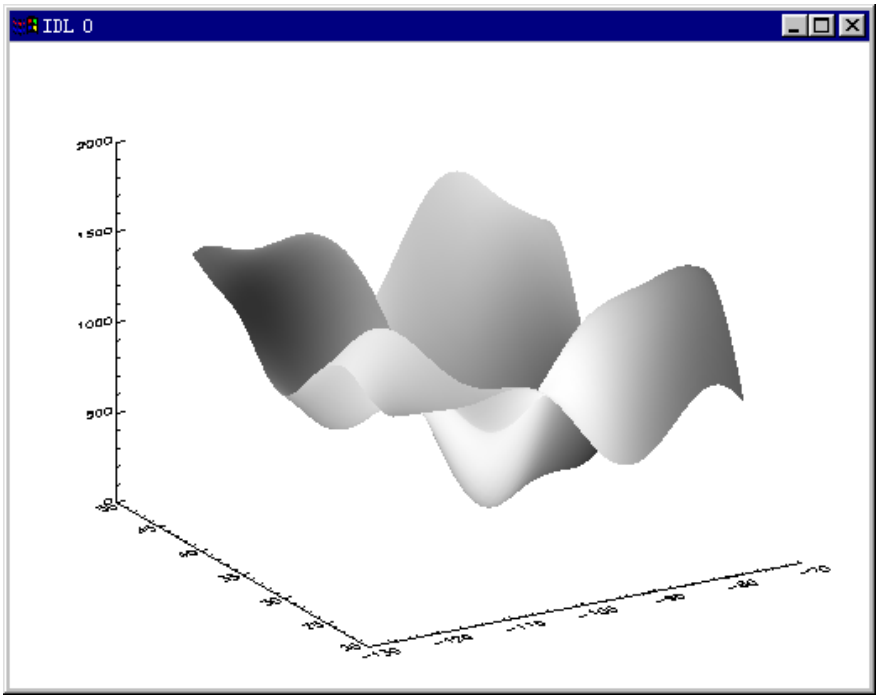


图 15 用 snow 数据集对 peak 数据进行阴影处理

如果要求根据数据点的高程值来对曲面进行阴影处理，可简单地对数据集本身进行字节比例缩放即可，键入：

```
IDL>Shade_Surf, peak, Shades=BytScl(peak, Top=!D.Table_Size)
```

将另一数据集覆盖在曲面图上是一种给数据升维的方法。例如，可将一组数据集覆盖在一个三维曲面图上，就可以直观的获得四维的信息。如果同时让两组数据集合随时间活动起来，就可以直观的获得五维信息。（关于数据动画参阅 104 页的“IDL 中的动画图形”）

有时只是想将原始曲面覆盖在经过阴影处理的曲面图上，通过结合使用 Shade_Surf 命令和 Surace 命令可轻松的做到。例如：

```
IDL>Shade_Surf, peak
IDL>Surface, peak, /NoErase
```

创建等值线图

在 IDL 中，任意二维数组都可以用一个 Contour 命令显示为等值线图。如果已经在这次 IDL 运行中定义了 peak 变量，可直接使用该变量。如果没有定义,可以使用 LoadData 命令来载入 Elevation Data 中的数据集。键入:

```
IDL>peak=LoadData(2)
IDL>Help, peak
```

这个数据集通过一个简单的命令即可显示为等值线图（图 16）：

```
IDL>Contour, peak, CharSize=1.5
```

图 16　　一个基本的等值线图，注意 X、Y 轴的标记代表该数组中的元素个数

注意，如果仅用单个二维数组作为参数调用 Contour 命令，它将把该数组作为其元素个数（此例在 X 和 Y 方向都为 41）的函数来绘图。如前述所用 Surface 命令一样，可以指定 X 轴和 Y 轴的数值，以便使其具有实际意义。例如，可以象前述一样使用经度和纬度矢量。如下所示：

```
IDL>lat=FIndGen(41)*(24./40)+24
IDL>lon=FindGen(41)*50.0/40-122
IDL>Contour, peak, lon, lat, XTitle='Longitude', $
      YTitle='Latitude'
```

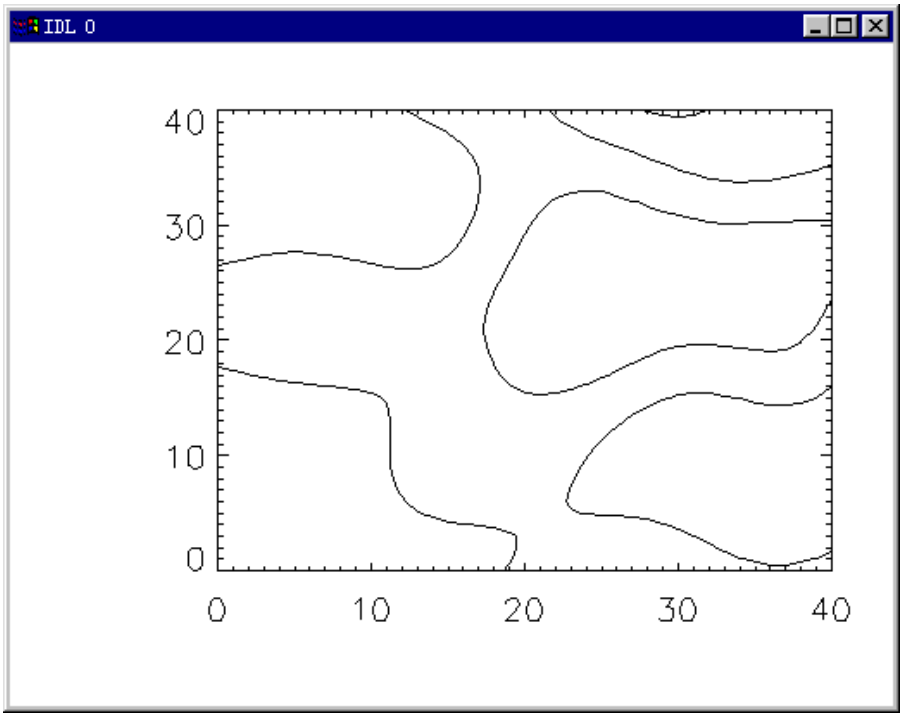
注意,轴被自动缩放了。从很多地方可以看到这一点。首先，等值线没有延伸到等值线图的边缘，其次，可以发现轴上的标记与 lon 矢量和 lat 矢量的最小值和最大值不同。

```
IDL>Print, Min(lon), Max(lon)
IDL>Print, Min(lat), Max(lat)
```

为了防止轴的自动缩放，可以设置 XStyle 和 YStyle 关键字，如下：

```
IDL>Contour, peak, lon, lat, XTitle='Longitude', $
       YTitle='Latitude', XStyle=1, YStyle=1
```

该命令得到的图形应如图 17 所示。



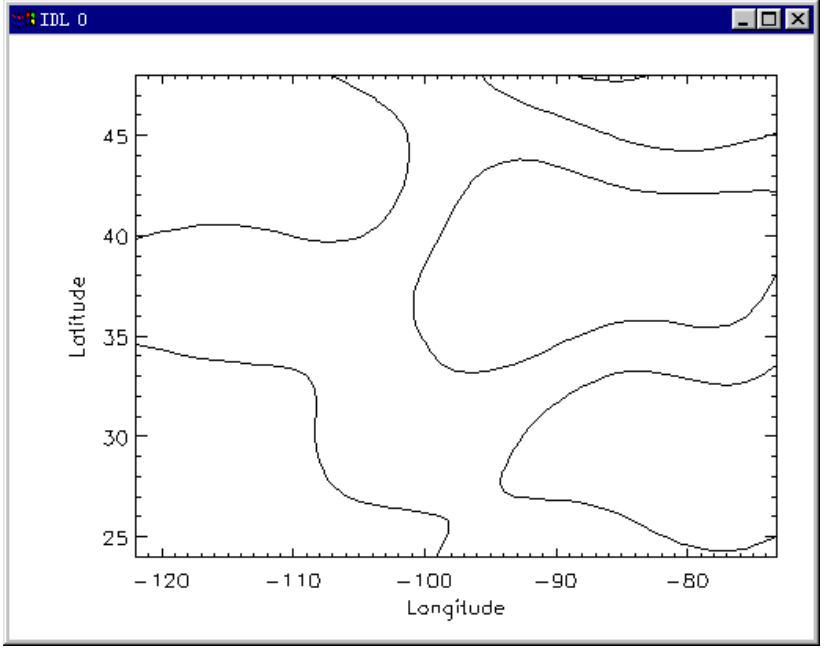


图 17　具有实际数量意义的等值线图

在早期的 IDL 版本中，Contour 命令使用所说的单元画法来计算并绘出数据的等值线。在这种方法中，等值线图是从图底画到图顶。这种方法是有效的，但是它不允许选项，比如标注等高线。而单元跟踪法被用来完整地画出围绕等值线图的每一条等值线。这需要较长的时间，但可以允许对等值线作更多的控制。例如，等值线可以断开用于等值线的标注。这种单元跟踪法可以用 Follow 关键字来调用：

```
IDL>Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow
```

从 IDL5 版本开始，Contour 命令一般都使用单元跟踪法来绘制等值线图。所以，Follow 关键字已经过时了。但该关键字仍然被使用，是因为它在自动标注其他每条等值线时的作用。

选择等值线数目

缺省情况下，IDL 选择 6 条匀称的等值线间隔（即有 5 条等值线）绘制等值线图。但是，可以用几种不同的方法改变缺省值。例如，可以用 Nlevels 关键字告诉 IDL 需要绘制多少条等值线。IDL 将计算出等间隔的等值线间隔数。例如，要绘制具有 12 条等间隔的等值线图，可键入：

```
IDL＞Contour, peak, lon, lat, Xstyle=1, Ystyle=1, /Follow, $
     Nlevels=12
```

输出结果应与图 18 相似。可选择高达 29 条的等值线。

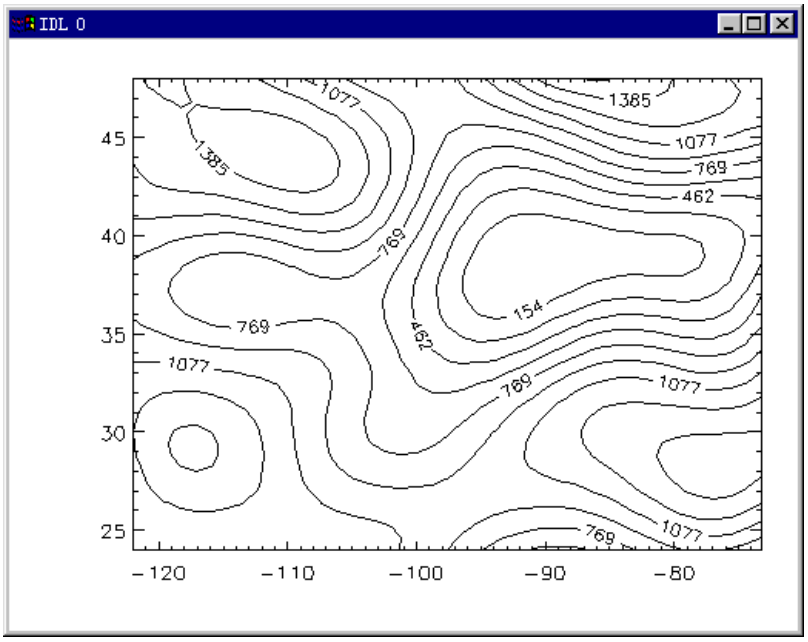


图 18 这是将等值线级别设置为 12 的等值线图。注意，每隔一条等值线都会标注一条，这是使用 Follow 关键字的一种副作用。

不幸的是，尽管 IDL 文档称 IDL 将采用给定的等间隔的等值线间隔数，但事实上不是这样。如果注意观察所创建的等值线图，会注意到 IDL 只计算出少于 12 条的间隔线。显然，NLevels 关键字的值在 IDL 中只能作为等值线选择算法中的一个“建议”。

因此，大多数 IDL 程序员都是自己计算等值线数目。例如，能精确规定哪条等值线应该画，并用 Levels 关键字传给 Contour 命令，而不是用 NLevels 关键字，如下所示：

```
IDL>vals=[200, 300, 600, 750, 800, 900, 1200, 1500]
IDL>Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
     Levels=vals
```

要选择 12 个间距相等的等值线间隔，可编写如下代码：

```
IDL>nlevels=12
IDL>step=(Max(peak)-Min(peak))/nlevels
IDL>vals=Indgen(nlevels)*step+Min(peak)
IDL>Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
      Levels=vals
```

如果喜欢，可以 C_Labels 关键字精确的指定哪一根等值线应该标注。这个关键字是一个其元素与等值线数目相等的矢量（如果元素个数与等值线数目不匹配，那么元素就不能象其他关键字那样循环使用）。如果某元素的值是 1（或更精确，只要是正数），相应的等值线就给予标注；如果某元素的值是 0，相应的等值线就不予标注。如果某条等值线没有元素值与之对应时，那么这条等值线就不标注。例如，要标注第一，第三，第六和第七条等值线，可键入：

```
IDL>Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
      Levels=vals, C_Labels=[1, 0, 1, 0, 0, 1, 1, 0]
```

要标注所有的等值线，可以使用 Replicate 命令来将 1 复制所需要的次数。键入：

```
IDL>Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $
      Levels=vals, C_Labels=Replicate(1, nlevels)
```

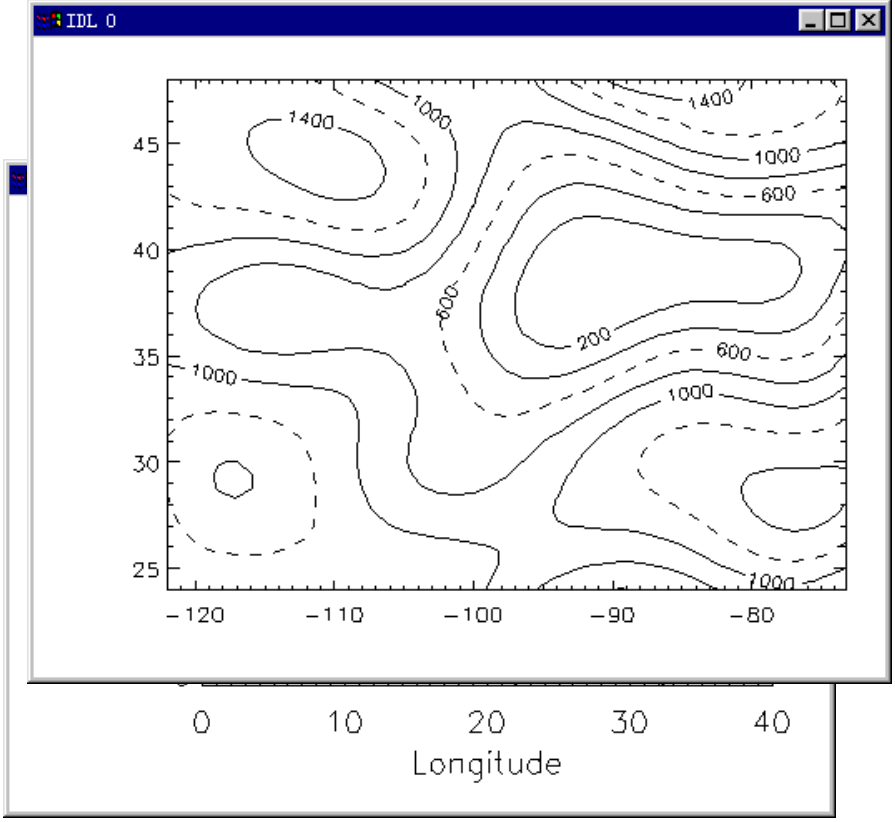
修改等值线图

等值线图可用与 Plot 命令和 Surface 命令中相同的关键字进行修改。但是仍然还有许多仅适用于 Contour 命令的关键字。它们中的大部分经常用于修改等值线本身。例如，用坐标轴标题注释等值线图，可键入：

```
IDL> contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  /Follow,
       Xtitle=’Longitude’, Ytitle=’Latitude’,  $
       Charsize=1.5,  Title=’Study Area 13F89’,  Nlevels=10
```

也可以用 C_Annotation 关键字在等值线上标注释。可以用字符串标记每一条等值线：

```
IDL> contour,  peak, Xstyle=1,  Ystyle=1,  /Follow, $
       Xtitle=’Longitude’,     Ytitle=’Latitude’,          Charsize=1.5,
Title=’Study Area 13F89’,$
       C_annotation=[‘Low’,’Middle’,’High’], Levels=[200,  500,  800]
```



输出结果应与图 19 中的图例相似。

图 19　　等值线可以用自己提供的文本标识

改变等值线图的外观

修改等值线图的外观有许多方法。这里有一些例子。能改变的特性之一是等值线的线型（见表 3 列出的可选用的线型值）。例如，为了使等值线成为虚线的线型，键入：

```
IDL> Contour, peak, lon,  lat,  Xstyle=1,  Ystyle=1,  $
             /Follow, C_LineStyle=2
```

假如需要隔二条等值线有一条虚线，可以用 C_LineStyle 关键字指定一个线型索引矢量，假如等值线数比索引号多，那么这些索引号将被循环使用或被重复使用。键入 ：

```
IDL> contour,  peak,  lon,  lat,  XStyle=1,  Ystyle=1,  /Follow,  $
       Nlevels=9,  C_LineStyle=[0, 0, 2]
```

输出结果应与图 20 相似.

图 20　可以修改等值线图的许多方面。这是隔二条等值线有一条虚线的等值线图

可以改变等值线的宽度。例如，要使等值线具有双倍的宽度，可键入：

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  $
    Nlevels=12,  C_Thick=2,  /Follow
```

通过指定一个线宽矢量，可以间隔修改等值线的宽度：

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  $
    Nlevels=12,  C_Thick=[1,  2],  /Follow
```

通过修改等值线图,可以很容易地看到等值线图的下坡方向。键入：

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  $
      / Follow,  Nlevels=12,  /Downhill
```

输出结果应与图 21 相似。

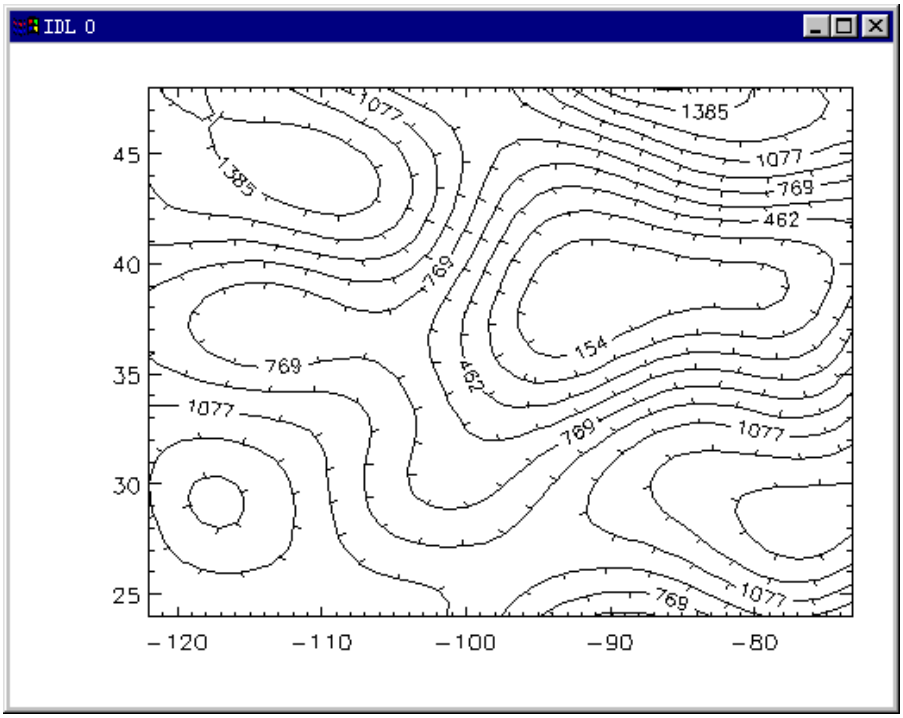


图 21　　用 Downhill 来显示等值线图的下坡方向

给等值线图赋色

给等值线图着色的方法有许多种。(颜色将在第 83 页的“IDL 的颜色运用”中详细讨论。现在，只须按如下键入 TvLCT 命令即可，以后将学到这个命令的意义。实质上，装载了三个颜色矢量，每个矢量的三个分量分别代表颜色的三个组成部分红、绿、蓝。这三种颜色矢量为碳灰、黄、绿色。）假如需要一张以碳灰颜色为背景的黄色等值线图，可键入：

```
IDL> TvLCT,  [70,  255],  [70,  255],  [70,  0],  1
IDL> Contour,  peak,  lon,  lat,  XStyle=1, YStyle=1,  $
     NLevels=10,  Color=2,  Background=1,  /Follow
```

也可以用 C_Color 关键字给等值线本身单独添上颜色。如果需要将上图的等值线变为绿色 ，键入：

```
IDL> TvLCT,  0,  255,  0,  3
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  /Follow,  $
     Nlevels=10,  Color=2,  Background=1,  C_Colors=3
```

关键字 C_Color 也可以被表达为彩色表索引号的矢量，并以循环的方式绘制等值线。也可以使用 Tek_Color 命令为等值线创建或者装入颜色，如下：

```
IDL> Tek_Color
IDL> TvLCT,  [70,  255],  [70,  255],  [70,  0],  1
IDL>Contour,  peak,  lon,  lat,  XStyle=1,  YStyle=1,  $
      Nlevels=10,  Color=2,  Background=1,  $
      C_Colors=IndGen(10)+2,  /Follow
```

可以很容易地用 C_Colors 关键字使每间隔二条等值线有一条蓝色的等值线，其余的等值线为绿色。键入：

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  $
     NLevels=12,  Color=2,  Background=1,  $
```

```
C_Colors=[3, 3, 4],  /Follow
```

创建填充的等值线图

有时用户不只是想观察等值线，也想查看填充后的等值线图。创建一张填充的等值线图，只需使用关键字 Fill 即可。首先，装入 12 种颜色于彩色表中作为填充颜色。色彩索引号由关键字 C_Colors 给出。键入：

```
IDL> LoadCT, 0
IDL> LoadCT,  4,  Ncolors=12,  Bottom=1
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  /Fill,  $
        NLevels=12,  /Follow,  C_Colors=Indgen(12)+1
```

用这种方法填充颜色还是存在许多问题，尽管从显示看不是很明显。事实上，在等值线图上有一个以背景颜色填充的”洞”。假如将背景色与图形颜色交换一下，就可以看得更清楚一些(事实上，PostScript 中就是这样做的。这也是致使许多 IDL 程序人员焦头烂额的原因)。

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  /Fill,  $
        NLevels=12,  /Follow,  C_Colors=Indgen(12)+1,  $
        Background=!P.Color,  Color=!P.Background
```

“洞”产生的原因是由于 IDL 用第一种颜色填充了第一和第二条等值线间的空间。用第一种填充颜色去填充第零条(或背景)和第一条等值线之间的空间似乎更合理。但是，要使 IDL 这样做，不得不给定自己的等值线数目，并用关键字 Levels 传送给 Contour 命令。通常可用下述代码实现：

```
IDL> step = (Max(peak) - Min(peak)) / 12.0
IDL> clevels = IndGen(12)*step + Min (peak)
```

现在，就得到了正确的等值线填充颜色。

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  /Fill,  $
        Levels=clevels,  /Follow,  C_Colors=Indgen(12)+1,  $
        Background=!P.Color,  Color=!P.Background
```

通常情况下，在填充等值线图时，经常定义等值线数目不失为一种好的方法。此外，要将填充的等值线图和色彩棒一起显示时，创建自己的等值线数目是确保等值线数目与色彩棒的级数一致的惟一方法。

有时候，需要填充有丢失数据的等值线图或者是等值线超出了图形边界的等值线图，这种情况称为“开放的等值线”。 IDL 处理这些开放的等值线时有时比较困难。填充这类等值线图的最好办法是使用关键字 Cell_Fill，而不是使用 Fill 关键字。这将导致 Contour 命令使用单元填充算法。这种算法没有 Fill 关键字使用的算法效率高，但在这种情况下可以获得更好的填充效果。假如需要将填充的等值线图放在地图投影上，使用 Cell_Fill 关键字也是个好主意。

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  $
        Levels=clevels,  C_Colors=Indgen(12)+1,  /Cell_Fill
```

单元填充算法有时会破坏等值线图的坐标轴。可以通过不带数据的等值线图的重新绘制来修复。键入：

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  $
```

```
        Levels=clevels,  /NoData,  /NoErase,  /Follow
```

有时，可能想在已填充好颜色的等值线图上看到等值线。在 IDL 中用 Overplot 关键字可以轻而易举地实现。键入：

```
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  $
        Levels=clevels,  /Fill,  C_Colors=IndGen(12)+1
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  /Follow,  $
        Levels=clevels,  /overplot
```

输出结果应与图 22 相似

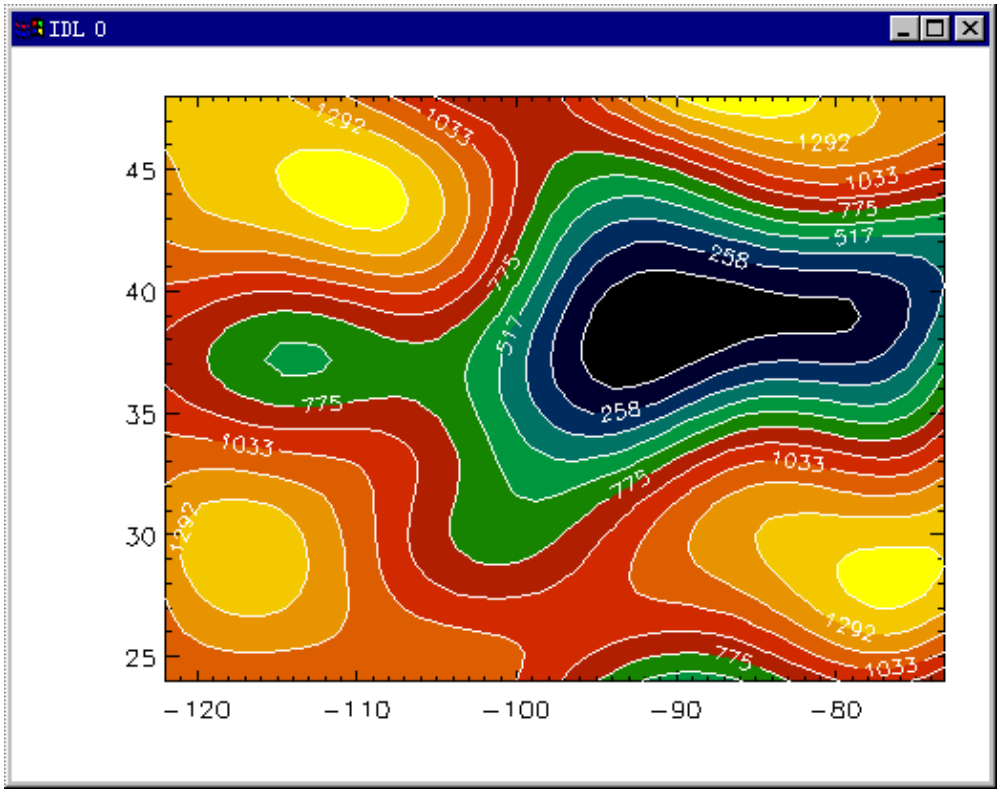


图 22　　在已填充的等值线图上覆盖等值线

注意，不要混淆 Overplot 和 NoErase 关键字。它们是相似的，但确切地说是不一样的。在等值线图上，Overplot 关键字仅仅绘出等值线，而不绘出等值线图的坐标轴。 NoErase 关键字则是绘出完整的等值线图，而不删除在屏幕上已显示的内容。

在显示窗口定位图形输出

在 IDL 内建几种在显示窗口中定位线画图、曲面图、等值线图和其他图形的方法（比如地图投影）。为了理解 IDL 怎样定位图形，了解一些定义很重要。图形位置是指在显示窗口上被图形坐标轴框起来的部分中的位置。图形位置不包括坐标轴标识、坐标轴标题或其他注释（见下面的图 23）。图形区域是显示窗口的一部分，包括图形位置，也包括环绕图形位置的空间，用来注明坐标轴标识，坐标轴标题和图标题等。图形边界定义为在显示窗口内不包括图形位置的区域。

图形位置可以用!P.Position 系统变量设置，或者用 Position 关键字对 Plot, Surface,

Contour 或其他 IDL 图形命令进行设置。整个图形区域可用!P.Region 系统变量设置，或者通过!X, !Y 和!Z 系统变量的 Region 字段来设置单个坐标轴的区域。图形边界可以用[XYZ]Margin 关键字来对 Plot, Surface, Contour 或 IDL 的其他图形命令进行设置，或者通过!X, !Y 和!Z 系统变量的 Margin 字段来设置。

在缺省值情况下，IDL 是在将图形输出到显示窗口的时候设置图形边界的。但是，正如所看到的，这并不是最好的选择。有时，使用图形定位来定位图形显示会更好，尤其是当您在一个显示窗口中显示多个命令的输出结果时。

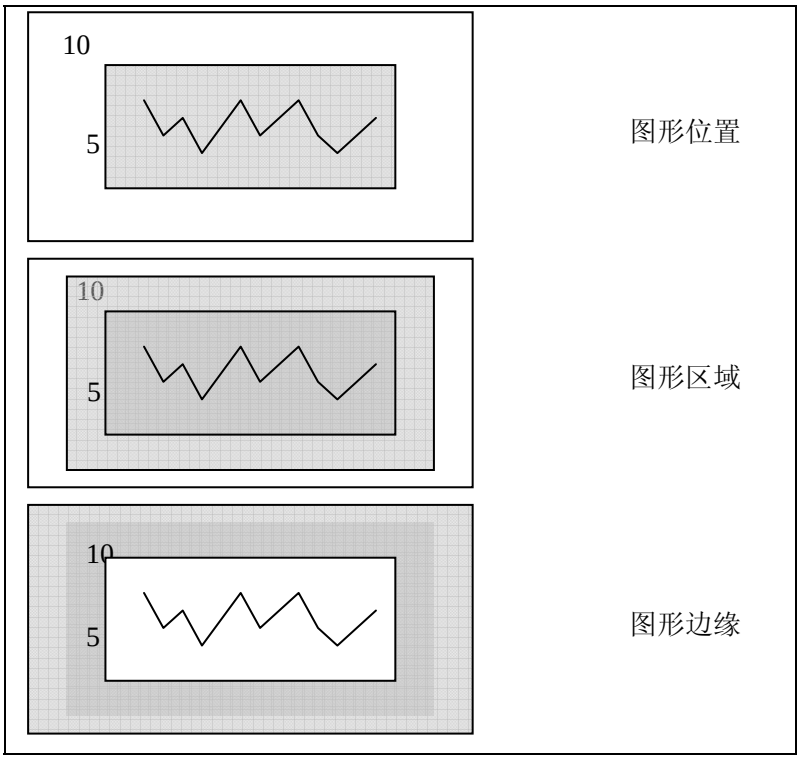


图 23　图形位置是被坐标轴包围起来的区域。图形区域与图形位置类似，但它还包括图形标题和其他注释的区域。图形边缘正好与图形位置相反。图形边缘由字符的单位确定，而图形位置和图形区域是由归一化的坐标单位确定。

设置图形边缘

图形边缘可以用图形命令中的[XYZ]Margin 关键字设置，或者通过!X,!Y 和 !Z 系统变量的 Margin 字段来设置。关于图形边缘的特殊地方在于它的单位根据字符尺寸来确定。X 方向的边缘是用两元素矢量来设置的，这两个元素分别规定左右的偏移量。Y 方向的边缘用同样的方法确定底部和顶部的偏移量。缺省边缘值是 X 轴方向为 10 和 3，Y 轴方向为 4 和 2。为了查看当前字符尺寸的设备坐标值或像素坐标值，可键入：

```
IDL> Print,  !D.X_Ch_Size,  !D.Y_Ch_Size
```

例如，在苹果机（Macintosh）中，缺省的字符尺寸在 X 方向上为 6 个像素，在 Y 方向上为 9 个像素。因此，一张等值线图的边缘就被确定为图形的左边为 60 个像素（6*10），右边为 18 个像素（6*3）。如果 CharacterSize 关键字在 Contour 命令中设置为 2，那么将会出现图形的左边边缘为 120 个像素，而图形的右边边缘为 36 个像素。

例如，为了将图形四周边缘都改变为 3 个缺省的字符宽度，可键入：

```
IDL> Plot,  time,  curve,  Xmargin=[3, 3],  Ymargin=[3, 3]
```

注意，如果同时改变字符尺寸，图形将出现非常大的差异。因为图形边界是由字符的尺寸确定的。键入：

```
IDL> !X. Margin = [3, 3]
IDL> !Y. Margin = [3, 3]
IDL> Contour,  peak,  CharacterSize=2.5
IDL. Contour,  peak,  CharacterSize=1.5
```

假如用其他的字符尺寸来做一些同样的操作，会发现，字符尺寸越大，字符将变得很大并且图形部分将变得很小，这并不是所希望看到的。当向下继续学习时，请确保将图形边界已恢复为缺省值。键入：

```
IDL> !X.Margin = [10, 3]
IDL> !Y.Margin = [4, 2]
```

注意，IDL 语言中有一些系统变量不象许多其他系统变量那样通过将其设置为零即可恢复其缺省值那样，系统边缘变量则必须直接将其设置为缺省值。假如没有键入以上的两条命令，请现在就键入将其恢复。

设置图形位置

设置图形位置需要设置一个四个元素的矢量，该矢量依次给定图形在显示窗口中的左下角和右上角坐标[X0，Y0，X1，Y1]。这些坐标值通常为归一化的值，其范围在 0 至 1 之间（如：0 常常代表显示窗口的左下角，1 常常代表显示窗口的右上角。）

设想如果需要将图形输出结果在显示窗口的上半部分显示，可以按如下设置!P.Position 系统变量并显示图形：

```
IDL> !P.Position = [0.1, 0.5, 0.9, 0.9]
IDL> Plot, time, curve
```

所有后面的图形输出定位方法都是类似的。将!P.Position 系统变量复位，以便后面的图形输出能正常地显示在窗口中。键入：

```
IDL> !p.position = 0
```

假如仅想给一张图形窗口定位，可以用图形命令的 Position 关键字规定一个图形位置。如果要在整个显示窗口的左半部分显示等值线图，可以键入：

```
IDL> Contour,  peak,  Position=[0.1, 0.1, 0.5, 0.9]
```

注意，Position 关键字可以用来在同一显示窗口输入多幅图形。只要确保在输入第二幅图形和所有的后续图形时，使用 NoErase 关键字。这可防止在显示图形时删除前面已显示的图形。对于所有的图形输出命令来说，这是一项缺省特性，但是对 TV 和 TVScl 命令则是例外。

在一张等值线图上加入一条线画图，可键入：

```
IDL> Plot,  time,  curve,  Position=[0.1, 0.55, 0.95, 0.95]
IDL> Contour,  peak,  Position=[0.1, 0.1, 0.95, 0.45],  /NoErase
```

设置图形区域

图形区域与图形位置一样，都是由归一化坐标值来确定的。同样可以通过设置!P.Region系统变量来指定。由于不存在和其他图形命令等效的关键字，因此设置图形区域没有设置图形位置方便。如果希望后续图形能正常地使用整个显示窗口，应确保已经将系统变量复位了。例如，在显示窗口上方三分之二的部分区域中显示一幅图形，键入：

```
IDL> !P.Region = [0.1, 0.33, 0.9, 0.9]
IDL> Plot,  time,  curve
```

将!P.Region 系统变量复位，以便后续图形能正常地在窗口内显示。键入：

```
IDL> !P.Region = 0
```

创建多个图形

正如所见，通过使用图形位置和图形区域系统变量以及上面所讨论的关键字可以在一个显示窗口中显示并定位多个图形（只要绘制第二个和后续的图形时使用了 NoErase 关键字）。另外一个系统变量!P.Multi 将会使在显示窗口内创建多个图形更加容易。!P.Multi 由以下五个元素进行矢量定义。

!P.Multi[0]　　!P.Multi 的第一个元素包括剩下的要在显示窗口或者 PostScript 页上绘制的图形数目。这有点不直观，下面可以看到它是如何使用的。通常设置为 0，意思是，没有剩下要在显示窗口输出的图形。接下来的图形命令将删除显示的图形，并且开始绘制新的多个图形中的第一个。

!P.Multi[1]　　此元素规定了该页上图形的列数

!P.Multi[2]　　此元素规定了该页上图形的行数

!P.Multi[3]　　此元素规定了在 Z 方向上叠加的图形数目（仅适用已经建立了三维坐标系的情况下）

!P.Multi[4]　　此元素规定了是先按行显示图形（!P.Multi[4]=0），还是先按列显示图形（!P.Multi[4]=1）。

假如想将!P.Multi 参数设置为按两行两列在显示窗口内显示四幅图形，并且，先按列显示图形，键入：

```
IDL> !P.Multi = [0, 2, 2, 0, 1]
```

显示图形时，如果要求每个图形占据窗口的四分之一位置，键入：

```
IDL> window, Xsize=500, Ysize=500
IDL> Plot,  time,  curve,  LineStyle=0
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  Nlevels=10
IDL> Surface,  peak,  lon,  lat
IDL> shade_Surf,  peak,  lon,  lat
```

输出结果应与图 24 相似

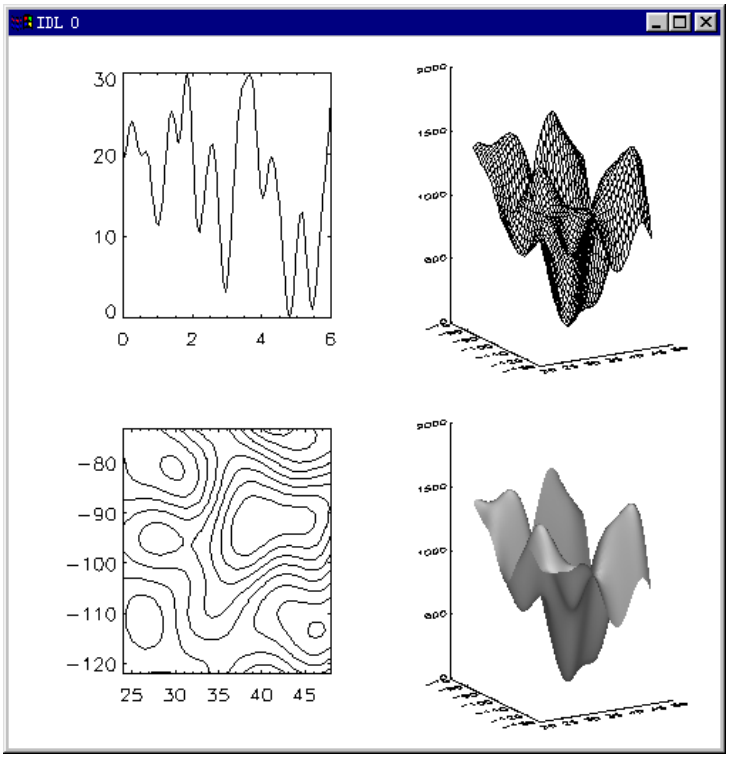


图 24　在单个显示窗口内可以绘制多幅图形

给单一窗口的多幅图形留下标题空间

当 IDL 计算图形位置时，是用整个显示窗口来决定每幅图形的大小。但是，有时想在显示窗口上有额外的空间来放图形标题或者其他类型的注释。可以通过使用!X、!Y 和!Z!系统变量的“外边缘”字段为多幅图形留出空间。外边缘字段仅仅在 P.Multi 系统变量被使用时有效。它们与正常的图形边缘一样，也是按字符单位来计算的。

如果想为刚刚创建好的四个图形的总图加上一个标题，应为标题留出空间。键入：

```
IDL> !P.Multi = [0, 2, 2, 0, 1]
IDL> !Y.Omargin = [2, 4]
IDL> Plot, time, curve, LineStyle=0
IDL> Contour, peak, lon, lat, Xstyle=1, Ystyle=1, Nlevels=10
IDL> Surface, peak, lon, lat
IDL> Shade_Surf, peak, lon, lat
IDL> XYOuts, 0.5, 0.9, /Normal, ‘Four Graphics Plots’, $
        Alignment=0.5, Charsize=2.5
```

输出结果应于图 25 相似。

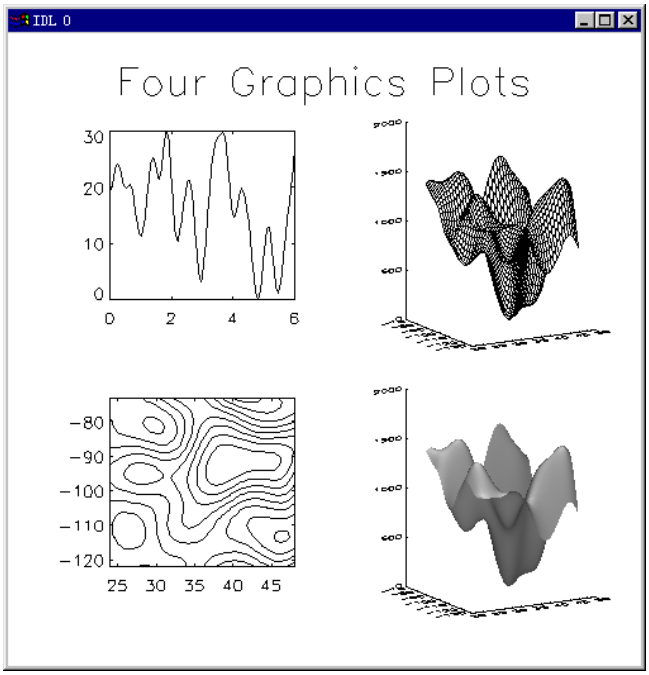


图 25　　使用关键字!Y.OMargin 在多幅图形的上方留出 4 个字符高度的空间来放标题

使用!P.Multi 变量创建不对称的排列

使用!P.Multi 变量绘图可以创建不对称的图形排列。例如，需要曲面图与阴影图一上一下地排列显示在显示窗口的左边，而显示窗口的右边是一张用同一数据生成的等值线图。可键入：

```
IDL> !P.Multi = [0, 2, 2, 0, 1]
IDL> !Y.OMargin=[0,0]
IDL> Surface,  peak,  lon,  lat
IDL> Shade_Surf,  peak,  lon,  lat
IDL> !P.Multi = [1, 2, 1, 0, 0]
IDL> Contour,  peak,  lon,  lat,  Xstyle=1,  Ystyle=1,  Nlevels=10
```

第一个!P.Multi 命令设置了一个二列二行的排列形式，第一、第二张图已制好。第二个 !P.Multi 命令设置了一个二列一行的排列形式。但要注意 !P.Multi[0]被设置为 1。结果是等值线图进入了显示窗口的第二个位置而不是第一个。结果由图 26 可以看出。

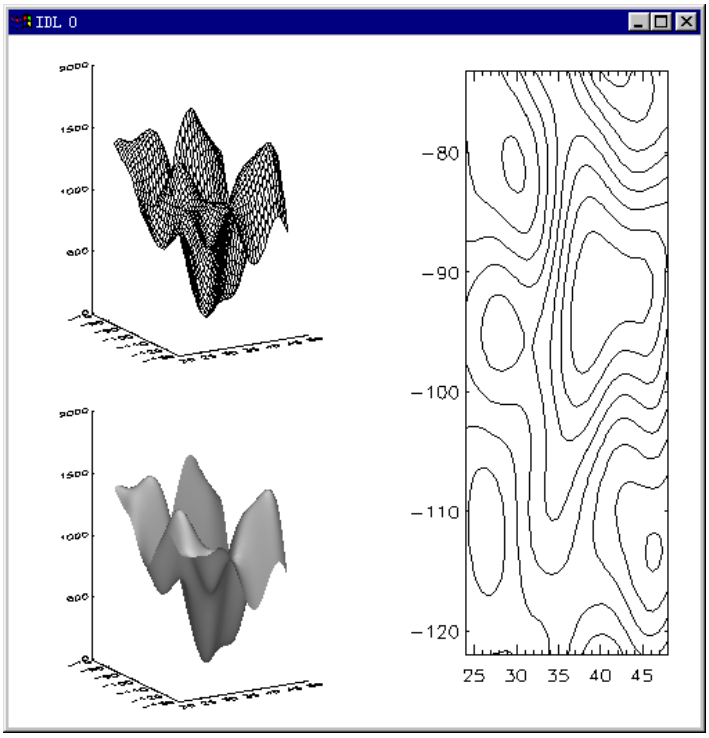


图 26　　可以使用!P.Multi 在显示窗口定位图形的不对称排列

注意：与 PLOT 和 CONTOUR 命令不同，TV 命令与!P.Multi 一起使用时无效。但是，在此书中可以用 TVImage 程序代替 TV 命令，该程序在已经下载的程序中。如果在使用 TVImage 中设置了 MULTI 关键字，!P.Multi 系统变量就有效。试试这些命令：

```
IDL> image = LoadData(7)
IDL> !P.Multi=[0, 2, 2]
IDL> FOR j=0, 3 DO TVImage,  image,  /Multi
```

确保已经将!P.Multi 复位，以便在一页上显示单个图形。象许多系统变量一样，!P.Multi 可以通过设置!P.Multi=0 重新设置为它的缺省值。

```
IDL> !P.Multi = 0
```

给图形显示添加文本

图形注释和其他文本可以通过许多方式添加到图形窗口中。最常用的方法是通过图形显示命令的关键字。被添加的文本可以三种字体“风格”中的任意一种形式出现：矢量字体（有时也称为软字体或 Hershey 字体）、TrueType 字体、硬字体。字体类型可以根据表 6 通过设置!P.Font 系统变量或者在图形输出命令中设置 Font 关键字来加以选择。

表 6 字体“风格”可以通过设置!P.Font 系统变量或者 Font 关键字为适当值来加以选择。矢量字体是直接图形命令的缺省字体，它们有不依赖于平台的优点

!P.Font	字体选择
-1	矢量字体（也叫软字体或 Hershey 字体）
0	硬字体
1	TureType 轮廓字体

在缺省值情况下，直接图形程序使用的是矢量或软字体的形式。矢量字体由矢量坐标描述。其结果是，它们是独立于平台并且极易在三维空间旋转。但是，许多人发现，对于高质量的硬拷贝输出来说，矢量字体太“瘦”了。为此，需要更丰满的字体（比如：TrueType 字体或者 PostScript 打印机硬件字体）。通过设置!P.Font 系统参数为-1 或者在图形输出命令上设置 Font 关键字为-1，就选择矢量字体了。如：

```
IDL> Plot,  time,  curve,  Font=-1,  Xtitle=’Time’,  $
        Ytitle=’Signal’,  Title=’Experiment 35F3a’
```

TrueType 字体也称为轮廓字体，这种字体是由一系列的外形轮廓描述的，这些轮廓通过创建一系列的多边形来填充。IDL 有四种 TrueType 字体家族系列：Times, Helvetica, Courier, 和 Symbol。TrueType 字体渲染需要花更长的时间，因为这种字体首先必须刻绘出来，然后创建相应的填充多边形，最后填充。并且许多人发现这种字体在低分辨率显示器上用小磅值时显示效果不好。但是它们有可以旋转的优点，并且在硬拷贝输出上较美观。TrueType 字体是 IDL 对象图形系统的一种缺省字体。

用缺省的 Helvetica TrueType 字体的外形来绘制图形，须设置 Font 关键字为 1。如：

```
IDL> Plot,  time,  curve,  Font=1,  Xtitle=’Time’,  $
        Ytitle=’Signal’, Title=’Experiment 35F3a’
```

TrueType 字体可以用 Device 命令通过 Set_Font 和 TT_Font 关键字来选择。如下：

```
IDL> Device,  Set_Font=’Courier’,  /TT_Font
IDL> Plot,  time,  curve,  Font=1,  Xtitle=’Time’, $
        Ytitle=’Signal’,  Title=’Experiment 35F3a’
```

在 IDL 中，为了掌握更多的 TrueType 字体，可以使用联机帮助系统。

```
IDL> ?  fonts
```

硬字体通过设置!P.Font 系统变量或 Font 关键字为 0 来加以选择。通常情况下，硬字体并不用于图形显示中，而是在当内容被输出到硬拷贝输出设备时使用，例如 PostScript 打印机。直到最近的 IDL 版本，硬字体都不能很好地在三维空间内旋转。因此，在使用类似于 Surface 等三维命令时，一般都不使用硬字体。

```
IDL> Plot,  time,  curve,  Font=0,  Xtitle=’Time’,  $
        Xtitle=’Signal’,  Title=’Experiment 35F3a’
```

列出可用字体的名称

可以用以下 Device 命令列出可用的硬字体名。如：

```
IDL> Device,  Font=’*’,  Get_FontNames=fontnames
IDL> For j=0,  N_Elements(fontnames)-1 DO Print,  fontnames[j]
```

只要使用 TT_Font 关键字，TureType 字体名称可用类似的方法列出。TT_Font 关键字用来选择系统上可用的 TureType 字体。（可以把自己的 TureType 字体加到 IDL 提供的四种系列类型内。如何实现可参考 IDL 的联机帮助系统。）

```
IDL> Device,  FONT=’*’,  Get_FontNames=fontnames,  /TT_Font
```

```
IDL> For j=0,  N_Elements(fontnames)-1 DO Print,  fontnames[j]
```

可用的矢量字体名称在表 7 给出。

用 XYOutS 命令添加文本

在 IDL 中,一个非常重要的命令是 XYOutS（在 XY 给定的位置，输出一个字符串）。这个命令用来在窗口的特定位置放入一个文本字符串。（XYOutS 的第一个位置参数是 X 的位置，第二个位置参数是 Y 的位置）。例如，给线画图加上一个较大的标题，键入如下命令：

```
IDL> Plot,  time,  curve,  Position=[0.15, 0.15, 0.95, 0.85]
IDL> XYOutS,  0.5,  32,  'Results: Expe riment 35F3a',  Size=2.0
```

注意，是用数据坐标来给定 X 和 Y 的位置，同时 Y 坐标在图形边界之外。在缺省的情况下，XYOutS 过程使用数据坐标系统。但是，如果选用适当的关键字，设备坐标系统和归一化的坐标系统也可使用。

（数据坐标系统由数据自身的大小所描述。设备坐标有时称为像素坐标，设备坐标系统经常和图像一起使用。归一化的坐标系统在每个方向从 0 到 1。当需要用独立于设备输出图形时，经常使用归一化坐标。）

例如，可以象如下使用归一化坐标把标题加在线画图上。当编写 IDL 程序时，用归一化坐标确定标题和其他注释等是一种很好的主意。这不仅更容易于在显示窗口定位图形，也便于在 PostScript 和其他硬拷贝输出文件中定位图形。

```
IDL> Plot,  time,  curve,  Position=[0.15, 0.15, 0.95, 0.85]
IDL> XYOutS,  0.2,  0.92,  'Results: Experiment 35F3a',  $
        Size=2.0,  /Normal
```

表 7　　Hershey 字体和其相应的在 IDL 中用于选择各字体的索引号

数值	描述	数值	描述
!3	Simplex Roman	!12	Simplex Script
!4	Simplex Greek	!13	Complex Script
!5	Duplex Roman	!14	Gothic Italian
!6	Complex Roman	!15	Gothic German
!7	Complex Greek	!16	Cyrillic
!8	Complex Italian	!17	Triplex Roman
!9	Math Font	!18	Triplex Italian
!10	Special Characters	!20	Miscellaneous
!11	Gothic English	!X	回到刚进入时的字体

用 XYOut 加注矢量字体

XYOutS 命令可用于矢量字体、TureType 字体或硬字体，只需按上述所述，简单地设置 Font 关键字值即可。这里要讨论的是关于矢量字体的，因为在直接图形命令中，此字体系统使用最频繁。表 7 中给出可获得的矢量字体和其相应的索引号，可以通过索引号来选择的特定字体。

矢量字体或 Hershey 字体的主要优点是它们的平台独立性，并且可在三维空间中缩放和旋转。例如，可以用 Triplex Roman 字体输出上图的标题，键入：

```
IDL> Plot,  time,  curve,  Position=[0.15, 0.15, 0.95, 0.85]
IDL> XYOutS,  0.2, 0.92,  '!17Results: Experiment 35F3a!X',  $
       Size=2.0,  /Normal
```

Triplex Roman 字体由!17 转换序列来选定。标题串末端的!X 将使字体转变为 Simplex Roman 字体，而 Simplex Roman 字体是在变为 Triplex Roman 字体前所使用的字体。这个转变步骤是非常重要的。否则，缺省设置将变为 Triplex Roman，并且所有接下来的串标记都将使用 Triplex Roman 字体。试试使用 Greek 字符集作为 X 轴的标题，并且按下面输出下图的标题。键入：

```
IDL> Plot,  time,  curve,  Xtitle='!17w',  $
       Position=[0.15, 0.15, 0.95, 0.85]
IDL> XYOutS, 0.2,  0.92,  'Experiment 35F3X',  size=2.0,  /Normal
```

可以在图 27 中看到结果。您可以注意到，即使没有规定该图的标题字符集，标题也是用 Greek 字符集输出的。恢复为缺省项 Simplex Roman 的惟一办法是用显式地使用 Simplex Roman 字体输出另一个字符串，例如：

```
IDL> XYOutS,  0.5, 0.5,  '!3Junk',  /Normal,  CharSize=-1
```

注意，在上面的代码中 CharSize 关键字的使用。当这个关键字值为-1 时，字符串被隐藏，不在窗口显示。

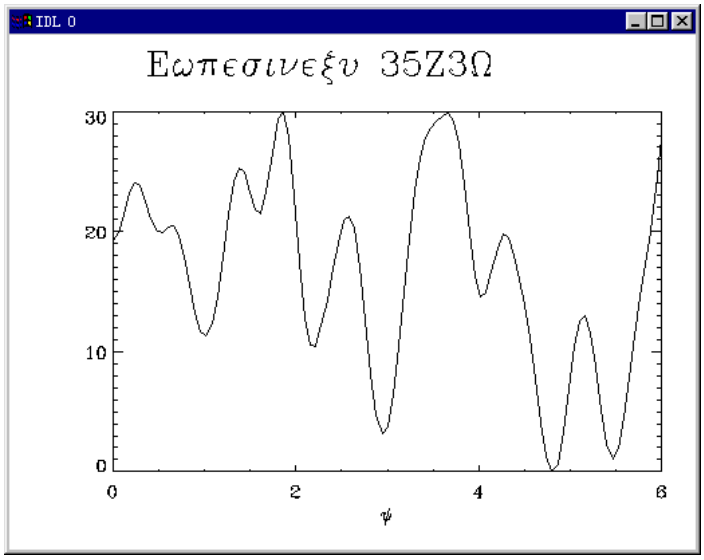


图 27　　当选择一种 Hershey 字体要注意，否则可能会显示出一个看上去象希腊字母的标题

排列文本

可以用 XYOutS 命令的 Alignment 关键字通过相对位置来定位文本。当 Alignment 的值为 0 时，字符串居左排列（这是缺省值）；当 Alignment 的值为 1 时，字符串居右排列；当 Alignment 的值为 0.5 时，将根据 X 和 Y 值所定义的位置居中排列。例如：

```
IDL> Window,  Xsize=300,  Ysize=250
```

```
IDL> XYOutS,  150,  55,  'Research',  Alignment=0.0,  $
        /Device,  CharSIZE=2.0
IDL> XYOutS,  150,  110,  'Research',  Alignment=.5,  $
        /Device,  CharSIZE=2.0
IDL> XYOutS, 150, 170,  'Research',  Alignment=1.0,  $
        /Device,  CharSize=2.0
IDL> Plots,  [0.5,0.5],  [1.0,0.0],  /Normal
```

删除文本

用 XYOutS 书写的文本有时可以通过用背景色书写同样的文本来删除。Color 关键字与!P.Background 系统变量一起使用可以达到这个目的。需要指出的是，这仅仅限于窗口中除了文本外没有其他内容的情况下奏效。通常还有别的更有效的方法来删除注释。（可参见 118 页的“从显示中删除注释”的例子）。为了理解如何用背景颜色删除注释，键入：

```
IDL> window,  Xsize=300,  Ysize=250
IDL> XYOutS,  150,  110,  'Research',  Alignment=0.50,  $
        /Device,  CharSize=2.0
IDL> XYOutS, 150, 110,  'Research'  Alignment=0.50,  $
        /Device,  CharSize=2.0,  Color=!P.Background
```

改变文本的方向

用 XYOutS 命令输出的文本可以通过 Orientation 关键字相对于水平方向上的角度来定向。Orientation 关键字可确定文本基线从水平基线开始旋转的度数。键入：

```
IDL> Window,Xsize=300, Ysize=250
IDL> XYOutS, 150, 110,  'Research', Alignment=0.5, $
        /Device, CharSize=2.0, Orientation=45
IDL> XYOutS, 150,180,  'Research', Alignment=0.50, $
        /Device, CharSize=2.0, Orientation=-45
```

给图形显示添加线和符号

给图形添加注释的另一个有效程序是 PlotS 命令，它可以用来在图形显示上添加符号或线条。PlotS 命令可在二维或三维空间中使用。

用 PlotS 程序画线，只需简单地提供含有 X 和 Y 坐标的矢量即可，矢量中的 X、Y 值是需要连接的点的 X、Y 坐标值。例如，从点（0，15）到点（6，15）在线画图上画一条基线，键入：

```
IDL> Window, XSize=500, YSize=400
IDL> Plot, time, curve
IDL> PlotS, [0,6], [15,15], LineStyle=2
```

输出结果应与图 28 相似。

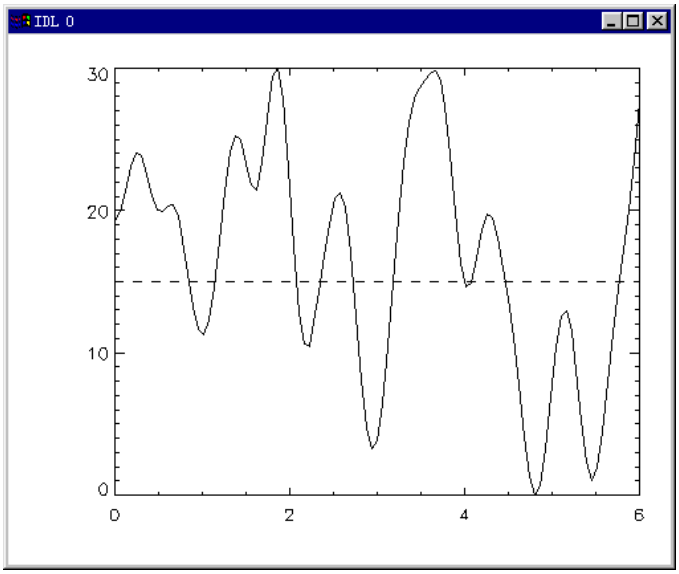


图 28　　用 PlotS 命令画一条虚线跨过图形的中部

PlotS 程序可以用来在任何位置标上符号。下面是在曲线上每五个点处标注一个菱形符号的实例。

```
IDL> TvLCT, [70, 255, 0], [70, 255, 250], [70, 0, 0], 1
IDL> Plot, time, curve, Background=1, Color=2
IDL> index = IndGen(20)*5
IDL> Plots, time[index], curve[index], Psym=4, $
        Color=3, SymSize=2
```

PlotS 命令也可以用来在图上重要信息的周围画出一个方框。通过 PlotS 命令与其他图形命令组合，如 XYOutS 命令，可以有效地注释图形显示。例：

```
IDL> TvLCT, [70, 255,0], [70,255,255], [70, 0, 0],1
IDL> Plot, time, curve, Background=1, Color=2
IDL> box_x_coords = [0.4, 0.4, 0.6, 0.6, 0.4]
IDL> box_x_coords = [0.4, 0.6, 0.6, 0.4,0.4]
IDL> PlotS, box_x_coords, box_y_coords, Color=3, /Normal
IDL> XYOutS, 0.5, 0.3, ‘Critical Zone’, Color=3, Size=2, $
        Alignment = 0.5, /Normal
```

注意，可以容易地使用 XYOutS 和 PlotS 命令为图形显示创建图例。

图形显示添加色彩

另一种有效的标注图形显示的方法是使用颜色。Polyfill 命令是一个低级的图形显示命令，它可用特殊的颜色或图案填充任意形状的多边形（无论是在二维或还是在三维环境中）。例如，可以使用 Polyfill 命令用红颜色填充上面线画图中方框：

```
IDL> TvLCT, 255, 0,0 ,4
IDL> Erase, Color=1
IDL> Polyfill, box_x_coords, box_y_coords, Color=4, /Normal
IDL> Plot, time, curve, Background=1, Color=2, /NoErase
IDL> PlotS, Box_x_coords, box_y_coords, Color=3, /Normal
```

```
IDL> XYOutS, 0.5, 0.3, 'Critical Zone', Color=3, Size=2, $
        Alignment = 0.5, / Normal
```

颜色有时代表一个数据集的另外一维的特性。例如，可以用二维圆形（或多边形）显示 XY 数据，而设置每个多边形的颜色就可表现出数据的某些附加特性，比如温度和人口密度等。查看以下是如何实现的。

IDL 没有构建圆的函数，但是很容易编写这样一个功能函数。打开文本编辑器或在 IDL 环境中键入代码来创建 IDL 的 Circle 功能。

```
FUNCTION CIRCLE, xcenter, ycenter, radius
Points = (2 * ! PI / 99.0) * FindGen(100)
x = xcenter + radius * Cos(points)
y = ycenter + radius * Sin(points)
RETURN, Transpose([x],[y])
END
```

组成圆周的 X 和 Y 值将以 2*100 数组形式返回。可以将该数组输入到 Polyfill 命令中。以 Circle.pro 保存该程序，并通过键入如下命令进行编译：

```
IDL> .Compile circle
```

然后，创建随机分布的 X 和 Y 数据。（将 Seed 设置回初始状态，这样输出结果将与图 29 看上去相似）。键入：

```
IDL> seed = -3L
IDL> x = RandomU(seed,30)
IDL> y = RandomU(seed,30)
```

将 Z 值设为这些 X 值和 Y 值的函数。键入：

```
IDL> z = (3 * ( (x-0.5)^2) + 5*((y-0.25)^2)) * 1000
```

打开窗口绘制 XY 位置，就可以看到这些数据是怎样以随机形式分布的。键入：

```
IDL> Window, Xsize=400, Ysize=350
IDL> Plot, x, y, Psym=4, Position=[0.15, 0.15, 0.75, 0.95],$
        Xtitle=' X Locations', Ytitle=' Y Locations'
```

将以不同颜色的圆显示与 XY 位置相关的 Z 数据。需要加入一张颜色表，并且使 Z 数据缩放至可获得的颜色数的范围内。键入：

```
IDL> LoadCT, 2
IDL> zcolors = Bytscl(z, Top=!D.Table_Size-1)
```

在这个例子里使用的 Circle 程序有许多弱点。主要缺点是它并非总是生成圆。假如用数据坐标系统来给定圆的坐标，圆形可能将以椭圆的形式显示，主要取决于图形长宽比例以及其他影响因素。（要获得非常棒的圆，可以从 NASA Goddard Astrophysics 的 IDL 例库中下载 TVCircle 程序，可以用浏览器通过 http://idlastro.gsfc.nasa.gov/homepage.html 来找到该例库）。为避免 Circle 程序中的这种不足，可用 Convert_Coord 命令将数据坐标转换为设备坐标。键入：

```
IDL> coords = Convert_Coord (X, Y, /Data, /To_Device)
IDL> x = coords(0,*)
IDL> y = coords(1,*)
```

最后需要使用 Polyfill 命令画出表示 Z 数据的彩色圆。键入：

```
IDL> For j=0, 29 Do Polyfill, Circle(x(j), y(j), 10), $
        /Fill, Color=zcolors(j), /Device
```

附带地说一下，最好有一个色标能够告知 Z 值和各种颜色的某些关系。可以用本书的 Colorbar 程序增加一个色标，键入：

```
IDL> Colorbar, Position = [0.85, 0.15, 0.90, 0.95], $
        Range=[Min(z), Max(z)], /Vertical, $
        Format=’(I5)’, /Right, Title=’Z Values’
```

输出结果应与图 29 相似

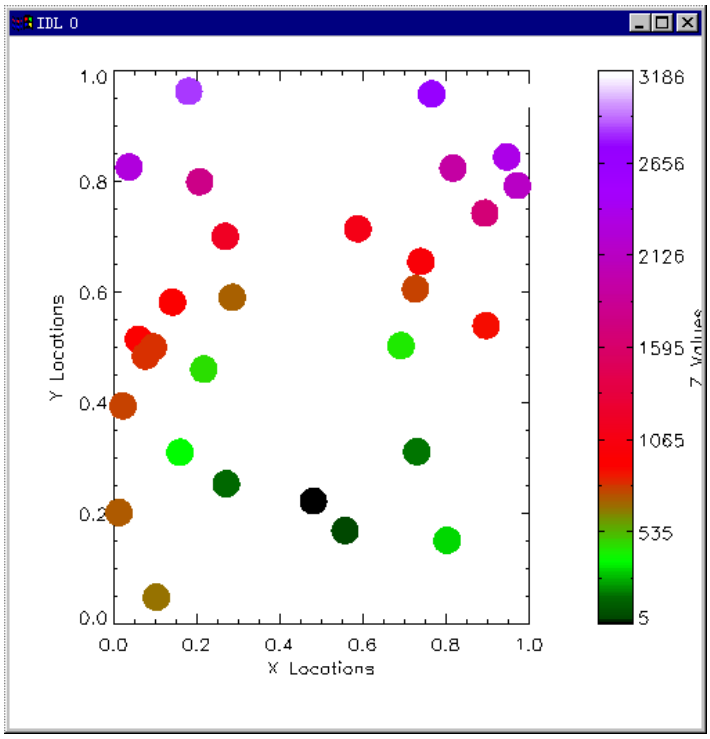


图 29　　在二维图中圆的颜色代表了第三维信息

第三章　图像数据处理

本章概要

IDL 最早是一种处理图像的语言。正因为此，世界各地的许多科学家和工程师仍在使用 IDL 语言。本章阐述了图像处理的基本工作。读者将从中学到以下几点：

1. 怎样读取和显示图像数据
2. 怎样缩放图像数据
3. 怎样在显示窗口中定位图像
4. 怎样改变图像的大小
5. 怎样从显示设备中读取图像
6. 怎样完成基本的图像处理任务
7. 怎样建立简单的图像滤波器

图像处理

事实上，任何类型的二维数据集都可认为是一幅图像。但是要在一个 8 位的显示设备上显示图像数据，就必须将图像数据调整为 0～255 之间的字节型数值。(在一个 24-bit 的显示设备上，24 位图像的 RGB 值必须调整成字节型数值)。因为图像总是以字节型数值显示，所以图像总是以字节型数组来存储。但是无论图像是怎样存储的，在 IDL 中，图像总是由两个显示图像的 IDL 命令：TV 和 TVScl 以字节型数值来完成。

要了解这两个命令是怎样工作的，需要有一些图像数据用于处理。用命令 LoadData 来打开图像数据集 Ali and Dave。将要处理这两幅图像数据中的第二幅图像。键入

```
IDL>image=LoadData(10)
IDL>image=image[*,*,1]
```

显示图像

可用 TV 和 TVScl 两个 IDL 命令中的任一个来显示图像。这两个命令几乎在各个方面都是一样的，包括能与之一起使用的关键字。仅仅在一个方面不同：TVScl 将图像数据调整为与 IDL 运行时所用颜色数目相适应的字节型数值。例如：如果在使用 IDL 时用 220 种颜色，则在图像显示之前 TVScl 将图像数据调整为 0～219 之间的字节型数值。

另一方面，TV 命令取图像数据本身的值，仅仅将其作为字节型数值直接传送到显示设备上。如果图像数据以整型或更多位数表示，则图像数据将被截断以符合字节型数值。如果图像数据不被调整到 0～255 之间，图像将很可能显示不正确。

注意，与 Plot，Surface 和 Contour 命令不同，TV 和 TVScl 命令在显示图像之前不删除窗口中已显示的内容。一般情况下这个问题影响不大，但有时候也会产生一些麻烦。如果想要一个空白的显示窗口来显示图像数据，无论当前窗口上的显示内容是什么，都可用一个简单的命令 Erase 来删除。

```
IDL>Erase
```

以下为一个实例。刚才读取的 IDL 的图像数据集已经调整在 0～255 之间。

可以键入如下语句来查看：

```
IDL>Print,Max(image),Min(image)
```

但是，如果在一个 8 位显示设备上工作，可能没有全部使用在显示器上可用的 256 种颜色。如果需要了解正在使用多少种颜色，可键入：

```
IDL>Print,! D.Table_Size
```

在一个 8 位显示器(这里指颜色表的大小)上,运行 IDL 时所用颜色的数目通常是在 210～240 之间，显然可用的颜色太少了。在一个 24 位的显示器上，可以获得 1670000 种颜色，但颜色表大小仍然是 256。以后将会学到 IDL 是怎样选择它所用的颜色数目。

打开一个显示窗口，装上灰度颜色表， 用 TV 命令显示图像:

```
IDL>Window,0,XSize=192,YSize=192
```

```
IDL>LoadCT,0
IDL>TV,image
```

所得图像应如图 30 所示。

图 30　IDL 和 Research Systems 公司的创始人—David Stern 的图像。People.dat 数据集中的另外一幅图像是 Ali Bahrami，Research Systems 公司的第一位员工。他们两人依然致力于 IDL 的开发

因为使用的是 TV 命令，所以数据没有经过拉伸就被送到显示器中显示。尽管看不出来，但图像上所有大于 IDL 运行时的颜色数目的像素值都被设为同样的值。　也就是说，比 ！D.Table_Size-1 值大的像素被以相同的颜色显示。（在这种情况下，看到的颜色是灰色明暗图。）

如果用 TVScl 命令显示图像，也许能看出差别。打开另一个窗口并将其移到第一个窗口的旁边。用 TVScl 命令显示图像：

```
IDL>window,1,XSize=192,YSize=192
IDL>TVScl,image
```

可看到两个图像的明暗程度不同。因为这幅图像数据最大值为 238，所以差别是很微弱的。如果看不出差别，可先在 0～255 之间对数据进行调整：

```
IDL>west,0
IDL>image=Bytscl(image)
IDL>tv,image
IDL>west,1
IDL>tvscl,image
```

如果仍不能看到差别，可装入颜色表。Red Temperature 颜色表可能起作用。键入：

```
IDL>LoadCT,3
```

如果要了解 TVScl 作了些什么，可调整数据并用 TV 命令显示：

```
IDL>Window,2,XSize=192,YSize=192
IDL>scaled=Bytscl(image,Top=!D.Table_Size-1)
IDL>TV,scaled
```

在窗口 2 中看到的图像应与窗口 1 中的图像一样。这就是所说的，TVScl 将数据调整为与 IDL 运行时所用颜色数目相适应的字节型数值。

注意：如果在显示窗口的图像不是用 red_temperature 颜色表显示的话，则可能是在一个 16 位或 24 位彩显上使用 IDL。在这种情况下，为了下面的练习，请确认关闭颜色分解关键字。键入如下命令：

```
IDL>Device,Decomposed=0
IDL>TV,scaled
```

如果用的是一个 16 位或 24 位显示器，为了看到新的颜色生效，在改变颜色表后，需要重新运行每个图形命令。在一个 16 位或 24 位显示器上，颜色表中的颜色没有直接被索引或连接到显示器上的彩色表。颜色表是图像用来查找每个像素该使用哪种颜色的一种方法。而像素的颜色是直接表示的。

一般来说，如果不知道数据是否被调整过，可能想用 TVScl 命令，因为这将给图像像素值以最大可能的对比度。但是如果颜色因素对您来说非常重要的话（并且它几乎总是这样），那么可能从来不必用 TVScl 命令。相反，将愿意自己拉伸图像数据，然后用 TV 命令来显示。

调整图像数据

假设您正在测量大气压，并将测量数据在一色标旁边以图像显示。可能想比较这个星期收集的图像数据和上个星期收集的图像数据。换句话说，想确定一种特定的颜色，比如红色，在这套数据中的红色和上个星期的数据中的红色表示相同的压力。

如果用 TVScl 命令显示这个星期和上个星期的图像数据，绝对不能保证特定的红色在两个数据组中能代表同一事情。

这些出入来自两个原因。第一，可能今天使用 IDL 时的颜色数目和上个星期使用 IDL 时不同。因为 TVScl 将图像数据调整到 IDL 运行时的颜色数目内，这可能会引起错误。第二，不能确保两组数据组间具有相同的数据范围。因而，用 TVScl 调整可能再次引起错误。

为解决这些问题，可用 BytScl 命令调整数据，并用 TV 命令显示。为确保 IDL 运行时所用的颜色数目不引起错误，可将数据调整到相同的颜色值内。并且，为确保数据集中数据的范围不引起错误，可以将数据调整到同样的数据范围。

可通过 BytScl 命令，应用关键字 Top、Min 和 Max 实现上述要求。例如，假设总是想以 100 种不同的灰度深浅或颜色深浅来显示数据，并且假设在任何数据集中希望最小数据值为 15，而最大的有效值为 245。可用如下 BytScl 命令实现：

```
IDL>scaledImage=BytScl(image,Min=15,Max=245,Top=99)
```

这个例子中， 数据调整之前在数据集中小于 15 的数值将设定为 15。类似地，在数据调整之前，在数据集中任何大于 245 的数值将被设定为 245。一旦数据被调整了，就可用 TV 命令显示。

```
IDL>TV,scaledImage
```

如果总是这样调整数据集（并且在 IDL 运行时总是有至少 100 个灰色级别或颜色数），那么上个星期的数据集就能直接与这个星期的数据集比较。一个特定的颜色，红色将总是表示一个特定的数据范围或压力。

可能在显示器上开了许多图像窗口。可用一个简单的命令删除所有开着的窗口。键入：

```
IDL>WHILE ! D.Window NE-1 DO WDelete,! D.Window
```

用颜色表分段表示图像

需要知道如何调整图像数据的另一个理由，是要能在使用 8 位显示器时，将数据调整到颜色表的不同部位。这使图像能用不同颜色显示出来，或者能将颜色表的特定色段用于特别的目的。例如，也许想将颜色表的一部分保留作为画图用的颜色。

注意：用 24 位彩显的一个很大的好处是能随时使用一个没有限制的颜色表。24 位彩显的缺点是，在改变颜色表之后，为了看到新颜色生效，不得不重新运行图形命令（例如：TV 命令）。在本书后面将看到如何编写程序，使得当一个新的颜色表装入后，能自动重新运行图形命令。

在大多数 8 位显示器上仅仅有一个物理颜色表，并且所有的 IDL 图形窗口都用它。但是

通过操作颜色表可以让它看上去象是同时装入几个不同的颜色表。可以通过将不同的颜色表装入到一个物理颜色表的不同部位来实现这一点。也许实现这点的最简单的方式是在LoadCT 或 XLoadCT 命令中用 NColors 和 Bottom 关键字。

例如，假设想用两个看上去不同的颜色表来显示同一幅图像。在用 IDL 打开一个图形窗口后，能通过测试系统变量！D.Table_Size 的值知道在 IDL 运行时颜色表中有多少种颜色。如果将这个数目一分为二，就知道每个图像该用多少种颜色：

```
IDL>half=!D.Table_Size/2
```

为了在同一窗口用看上去不同的两个颜色表显示图像数据 image，必须将图像数据调整为适应两个颜色空间范围的值。首先，用 BytScl 命令调整图像数据为适应第一个部分颜色表的值，生成一个新的图像 image1：

```
IDL>image1=BytScl(image,Top=half-1)
```

现在，按如下做法将图像数据调整为适应第二个部分颜色表的值，生成第二个图像 image2：

```
IDL>image2=BytScl(image,TOP=half-1)+Byte(half)
```

按如下做法将两个已调整的图像肩并肩地放在同一个窗口。注意，在使用 TV 命令。明白这是为什么吗?

```
IDL>Window,XSize=192*2,YSize=192
IDL>TV,image1
IDL>TV,image2,192,0
```

现在需要用一个灰度颜色表(颜色表索引号为 0)将左边的图像显示出来。必须将那些灰度级颜色装入颜色表中被第一个图像数据占用的部分。键入：

```
IDL>LoadCT,0,NColors=half,Bottom=0
```

如果用 XLoadCT 命令将颜色装入颜色表的第二部分，就能为右边的图像交互式地选择想要的任何颜色表。如下：

```
IDL>XLoadCT,NColors=half,Bottom=half
```

为了继续本章后面的例子，要恢复一个正常的颜色表，键入：

```
IDL>LoadCT,0
```

在 24 位显示器上用不同的颜色表显示图像

当在16位或24位显示器上运行时，使用不同的颜色表和装入颜色并显示图像一样简单。例如，如果正在一个 16 位或 24 位的显示器上运行时，可以试一试：

```
IDL>world=LoadData(7)
IDL>Window,1,Title= ‘Gray Scale Image’
IDL>LoadCT,0
IDL>TV,world
IDL>Window,2,Title= ‘Color Image’
IDL>LoadCT,5
IDL>TV,world
```

显示 24 位图像

真彩色（或 24 位）图像也能用 TV 命令显示。24 位图像总是由一个 3 维数据集构成，它的 3 个维数中的一个值设为 3。例如，数据集可以是一个 m*n*3 的数组，这种情况下，图像被认为是隔波段扫描（band-interleaved）；如果图像是 m*3*n 则被认为是隔行扫描

（row-interleaved）；如果是 3*m*n 则被认为是隔像素扫描（pixel-interleaved）。

装载一幅 24 位图像，键入如下命令：

```
IDL>rose=LoadData(16)
```

这个数据组是一个按像素扫描的图像。通过键入如下命令可知道这点：

```
IDL>Help,rose
       ROSE      BYTE       =Array[3,227,149]
```

要在一个 8 位显示器上显示一幅 24 位的图像，仅仅需要用关键字 True 来说明其用的是哪种扫描方式。True=1 为隔像素扫描；True=2 为隔行扫描；True=3 为隔波段扫描。

```
IDL>Window,XSize=227,YSize=149
IDL>TV,rose,True=1     ;Pixel-interleaved
```

注意，24 位图像在 8 位显示器上显示将表现为灰度级。要在这样的显示器上看到真彩色的图像，需要创建一幅 2 维图像以及伴随该 24 位图像或 3 维图像数据的红色、绿色、蓝色颜色表。这在 IDL 中可用命令 Color_Quan 来实现。如果使用 8 位显示器，键入如下命令：

```
IDL>image2d=Color_Quan(image24,1,r,g,b)
IDL>TVLCT,r,g,b
IDL>TV,image2d
```

现在可看到彩色图像了。

在 24 位显示器上显示 24 位图像

如果使用的是一台 24 位显示器，情形稍微复杂一点。为了正确显示一幅 24 位图像，必须打开颜色分解关键字。这在大多数真彩模式下的工作站上自动实现的，但在真彩模式 Windows 下，IDL5.2 版本却不能自动实现。为确保以正确的图像颜色显示 24 位图像，应该在 24 位显示器上键入如下命令：

```
IDL>Device,Decomposed=1
IDL>TV,image24
```

注意，下载的本书配套的程序 TVImage 自动设置正确的颜色分解关键字，这取决于要显示的图像是 24 位还是 8 位。

在 24 位显示器上显示 8 位图像

在一台 24 位显示器上，要将 8 位图像进行彩色显示则必须要用到颜色表。换句话说，一个 8 位图像的像素值被作为一个索引号，该索引号为给定的像素查找特定的红色，绿色和蓝色。这意味着如果在使用 IDL 时改变了颜色表，必须重新显示该 2 维图像来看新的颜色是否生效。这是因为在 24 位显示器上颜色是在图像被显示时决定的，同时也因为正在用 RGB 颜色模式。并且特别要注意必须关上颜色分解关键字，否则将忽视颜色表矢量，并总是用灰度色彩来显示 8 位图像。如果用的是一个 24 位显示器，键入如下命令：

```
IDL>world=LoadData(7)
IDL>Window,XSize=360,YSize=360
IDL>LoadCT,5
IDL>Device,Decomposed=0
IDL>TV,world
```

为了以另一种颜色表显示图像，装入该颜色并重新运行 TV 命令，使图像像素值遍历整个颜色表矢量。注意当只运行 LoadCT 命令时，图像颜色不变。

```
IDL>LoadCT,3
IDL>TV,world
```

控制图像显示顺序

通常，当 IDL 显示一幅图像时，习惯上图像的第 0 列和第 0 行为图像的左下角。有些人喜欢将图像的第 0 列和第 0 行作为图像的左上角。如果喜欢第二种习惯方式，可以通过设置系统变量！Order 让 IDL 使用习惯。在缺省时，！Order 设为 0。如果希望将所有图像的左上角都显示在第 0 列和第 0 行，可设置！Order=1。

如果只是希望用第二种方式显示某幅图像，可在使用 TV 或 TVScl 命令时，用关键字 Order 设置。例如，可以同时观看两种显示方式，键入：

```
IDL>Window,XSize=192*2,YSize=192
IDL>TVScl,image,Order=0
IDL>TVScl,image,Order=1,192,0
```

可能您从别人那儿得到一个图像数据文件，显示时发现图像倒过来了。这大多是因为创建数据文件的人在排放第 0 列和第 0 行时用了不同的习惯。将关键字 Order 的值反过来，看是否纠正了错误。

改变图像尺寸

IDL 提供了两个改变图像大小的命令：Rebin 和 Congrid 。

Rebin 的限制为新建图像的尺寸必须是原始图像尺寸的整数倍或整数比例。例如，变量 image 可以在 X 方向或 Y 方向上变化为 192/2 和 192*3 个元素。但不应该是 300 或 500 个元素。图像大小也可以在一个方向减小，另一个方向增大。例如，可将变量 image 重新变化为 384 列和 96 行，键入如下命令。

```
IDL>Window,XSize=384,YSize=96
IDL>new=Rebin(image,384,96)
IDL>TVScl,new
```

输出图像应与图 31 类似。

图 31 用 Rebin 命令缩放的图像其大小必须与原始图像大小有整数倍关系

在缺省情况下，当放大一幅图像时 Rebin 采用双线性插值，当缩小一幅图像时则采用最邻近平均法。如果关键字 Sample 被设定后，在两个方向上都可用最邻近采样法。双线性插值更为精确，但需要更多的计算时间。

```
IDL>Window,XSize=192/2,YSize=192/2
IDL>new=Rebin(image,96,96,/Sample)
IDL>TVScl,new
```

Congrid 与 Rebin 很相似，除了下面两个方面外，第一，在新图像中的列数和行数可以设为任意值。第二，在缺省情况下，用的是最邻近采样法。如果想用双线性插值，必须设置关键字 Interp：

```
IDL>Window,XSize=600,YSize=400
```

```
IDL>new=Congrid(image,600,400,/Interp)
IDL>TVScl,new
```

在 PostScript 设备上改变图像大小

象 PostScript 这种设备，其像素是可调节的（相对于固定像素的显示器来说），在调节图像尺寸时有所不同。（详细信息参考 185 页的“显示设备与 PostScript 设备之间的不同点”）。特别是，可不用 Rebin 或 Congrid 命令来改变图像的大小，而是用 TV 或 TVScl 命令通过关键字 XSize 和 Ysize 来改变图像的大小。

例如，当将图像输出到一个 PostScript 文件时,如果想将图像的显示比例定为 6：4 的话，也许愿意用如下代码，而不是上面的用 Congrid 命令将图像放大为 600*400。

```
IDL>thisDevice=! D.Name
IDL>Set_Plot,'PS'
IDL>Device,XSize=6,YSize=4,/Inches
IDL>TVScl,image,XSize=6,YSize=4,/Inches
IDL>Set_Plot,this Device
```

如果图像大小和位置是用下面的归一化坐标来表示的,可以写出真正的独立于显示设备的图像显示代码。

在显示窗口中定位图像

通常显示一幅图像时，IDL 将图像的左下角放在窗口的左下角。但是可通过 TV 或 TVScl 命令的附加参数来将图像移动到显示窗口中的其他位置。

例如，如果给出第二个参数，它则被视为图像在窗口中的位置。图像位置由显示窗口的尺寸和图像的尺寸计算出来的。详细算法可参阅 TV 命令的在线帮助。键入：

```
IDL>?  TV
```

位置可从显示器的左上角开始，一直到显示器的右下角。例如，在 384＊384 的显示窗口内，从显示器的左上角开始，对于 192＊192 的图像来说有四种位置。键入：

```
IDL>Window,XSize=384,YSize=384
IDL>TVScl,image,0
IDL>TVScl,image,1
IDL>TVScl,image,2
IDL>TVScl,image,3
```

通过显式地指定图像左下角的象素位置来定位一幅图像是可以的。TV 或 TVScl 命令中在图像数据名后给定两个附加参数即可以实现这点。例如，将一幅 192＊192 名为 image 的图像定位于刚刚创建的显示窗口中间。可键入：

```
IDL>Erase,Color=!D.Table_Size-1
IDL>TVScl,image,96,96
```

这样，将图像的左下角放在像素点（96,96）处。当希望为附加图留下空间时，如色标或其他的注释，这种定位图像的方法是很重要的。

例如，键入如下命令来在窗口的左边显示一个色标，在窗口的右边显示图像。显示窗口将如图 32 所示。

图 32　此图像用颜色标来显示其颜色梯度

```
IDL>Window,XSize=320,YSize=320

    IDL>ncolors=! D.Table_Size
    IDL>TvLCT,255,255,0,ncolors-1
    IDL>Erase,color=ncolors-1
    IDL>colorbar=Replicate(1B,20)#BIndGen(256)
    IDL>TV,BytScl(colorbar,Top=ncolors-2),32,36
    IDL>TV,BytScl(image,Top=ncolors-2),92,64
```

用归一化的坐标来定位图像

用归一化的坐标系来定位图像和确定图像大小是很方便的。这与其他 IDL 图 象命令使用关键字 Position 的用法相似。（详细信息参考 185 页的“显示设备与 PostScript 设备之间的不同点”）。如果要在可变尺寸的窗口内显示图像，或者要在同一显示窗口内与其他 IDL 的图形程序共同使用图像，或者希望所编写的将图像传送到一个 PostScript 文件的 IDL 程序不遇到麻烦，在这些情况下，用归一化坐标定位图像和确定图像大小是非常方便的，特别是对最后一种情况。例如，刚才键入的命令是放一个色标在图像旁。尽管这些命令在显示窗口内可以很好地工作，但如果想在 PostScript 设备上输出中得到类似的结果，上述命令是不可能的。（若有问题，参考 185 页的“显示设备与 PostScript 设备之间的不同点”）。

假设可以用关键字 Position 在窗口中确定图像的大小和位置，那么结果将如何呢？设想将图像放入一个任意的窗口内，并占满其比如 80%的空间。相对于归一化坐标来说，可将图像在窗口的位置表达为：

```
position=[0.1,0.1,0.9,0.9]
```

但这是怎样被转换成图像通常使用的设备坐标的呢？这自然要取决于显示窗口的大小。但要知道显示窗口中可视部分的大小。这是由系统变量!D.X_VSize 和!D.Y_Vsize 以设备或像素单元来给定的。

通过像素坐标，可以按如下方法计算出图像所需的尺寸和在输出窗口起始的位置：

```
xsize=(position[2]-position[0])*! D.X_VSize
```

```
ysize=(position[3]-position[1])*! D.Y_VSize
xstart=  position[0]*! D.X_VSize
ystart=  position[1]*! D.Y_VSize
```

将图像输出到显示设备和输出到 PostScript 文件的惟一区别是如何确定图像的尺寸。可以编写如下代码来显示图像:

```
IF !D.Name EQ 'PS' THEN $
      TV, image, XSize=xsize, YSize=ysize, xstart, ystart $
ELSE $
      TV, Congrid(image, xsize, ysize), xstart, ystart
```

无论是将图像输出到显示器还是输出到 PostScript 文件中，上述代码都起作用。但这样做时图像的横纵比例不能得到保证。事实上，可以让图像适合窗口的形状。这对一些应用程序来说工作的很好，但对另一些却不是这样。无论在什么情况下该问题都很容易解决，因为如果想保留图像的横纵比例时，只需固定好图像的一边，并以适当的方式调整图像另一边的坐标即可。

实现该项功能的代码已经写好，放在下载的程序 TVImage 中。无论是在显示终端，还是在 PostScript 文件中，TVImage 都用关键字 Position 来定位图像和确定图像大小，。如果希望 TVImage 程序能完好地保持显示图像的横纵比例,可以使用关键字 Keep_Aspect_Ratio。

可以用 TVImage 重新生成色标位于图像左边的上述图像：

```
IDL>Erase,color=ncolors-1
IDL>barPosition=[32,32,52,292]/320.0
IDL>imagePosition=[92, 64, 284, 256]/320.0
```

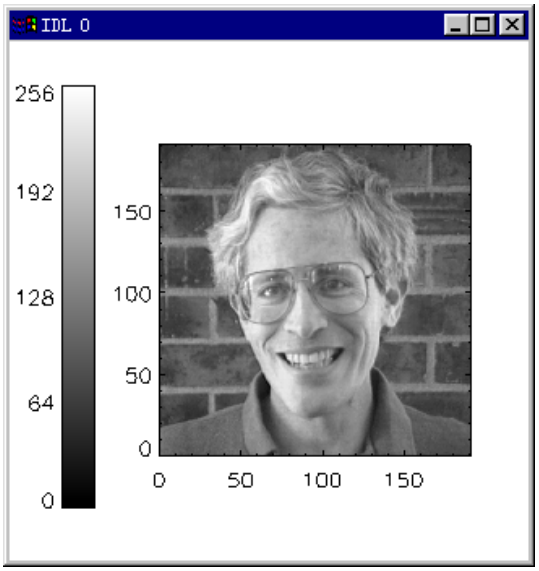


图 33　用命令 TVImage 不仅允许使用独立于设备的方法定位图像而且容易使用其他图形命令

```
IDL>colorbar=Replicate(1B, 20)#BIndGen(256)
IDL>TVImage,BytScl (colorbar,Top=ncolors-2),$
       Position=barPosition
IDL>TVImage,BytScl (image,Top=ncolors-2),$
       Position=imagePosition
```

这样做的好处，不但因为图像可以在任何尺寸的窗口或 PostScript 文件中以及显示器

上显示，而且它使得我们很容易在显示窗口中增加其他图形。例如，可以非常容易地在色标和图像周围放置外框或标记。键入：

```
IDL>TvLCT,255,255,255,ncolors-1
IDL>Plot,[0,! D.Table_Size,YRange=[0,! D.Table_Size],$
      /NoData,Color=0,Position=barPosition,XTicks=1,$
      /NoErase, XStyle=1, YStyle=1, XTickFormat='(A1)'?$
      YTick=4
IDL>Plot, IndGen(192), IndGen(192), /NoData, $
      Position=imagePosition, /NoErase, $
   XStyle=1, YStyle=1, Color=0
```

输出结果应如图 33 所示。

从显示器中读取图像

有时您可能花了许多时间，运行了许多命令，才得到了喜欢的图形显示。将图形显示读到一个图像变量中以便处理甚至硬拷贝输出是很方便的。因此，现在需要知道如何得到 IDL 图形窗口的拷屏。可以用 TVRD 命令将 IDL 图形窗口的内容读到一个 2 维 IDL 字节型数组中。

要在一个 8 位显示器上读取整个图形窗口，可键入如下命令：

```
IDL>Window,XSize=250,YSize=250
IDL>TVScl,image
IDL>new_image=TVRD()
IDL>Help,new_image
```

注意，新创建的变量现在是一个 250*250 字节的数组。

在 24 位显示器上抓屏

如果在 16 位或 24 位显示器上运行 IDL，不能象上面那样使用 TVRD 命令。16 位或 24 位 显示器有 3 个颜色通道。如果象上面不用任何参数来使用 TVRD 命令，那么得到的 2 维数组的每个像素值为该像素三个通道中的最大像素值。除非装载一个灰度彩色表（这样，每个通道有同样的值），否则所的结果将不是所期望的。要想在 24 位显示器上抓屏，只需在命令 TVRD 中设置关键字 True 即可。如果用的是 16 位或 24 位显示器则键入如下命令：

```
IDL>new_image=TVRD(True=1)
```

但是注意，得到的是一个 24 位图像，而不是一个 2 维 8 位图像。当显示该图像时就要用带关键字 True 的 TV 命令：

```
IDL>Help,new_image
IDL>Erase
IDL>TV,new_image,True=1
```

读取显示图像的一部分

如果只想读显示窗口的某一部分，可指定想要的部分窗口的左下角的像素坐标和要读取的列和行的数目。换句话说，能指定矩形区域。例如，如果只想捕获上述头像的脸部，可键入：

```
IDL>new_image=TVRD(40,30,110,130)
```

在 IDL 中，从显示器读取的 2 维数组可以象处理其他图像那样进行处理。如果需要显示，可键入：

```
IDL>Erase
IDL>TV,new_image
```

IDL 中基本的图像处理

IDL 原是作为一种图像处理的工具，所以它有很强的图像处理能力。这节中描述的是一些 IDL 中基本的图像处理工具。

直方图均衡化

如果观察图像中的像素值分布，往往会发现分布趋向集中在一个狭窄的数值范围内。实际上，图像有一个非常窄的动态颜色范围。如果像素分布开，以致使像素值的每个子范围都与这些像素值一样拥有数目大约相同的像素，则该图像的信息内容就有可能增加。将像素分布到整个颜色范围的过程叫做直方图均衡化。

例如，用 LoadData 命令打开数据集 CT Scan Thoracic Cavity。这是一幅 CT 扫描图像，该图像具有一个狭窄的动态颜色范围。

```
IDL>scan=LoadData(5)
```

要看变量 scan 的像素值分布的柱状图，键入如下命令。显示图像窗口应如图 34 所示。

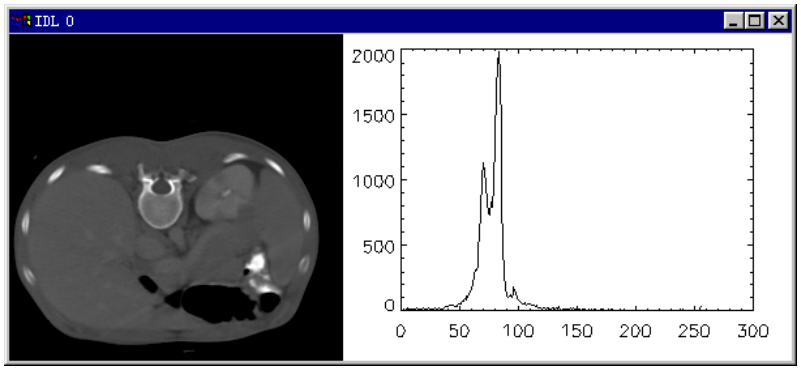


图 34　正常图像具有狭窄的像素值分布。这里的像素值集中在 50-10 之间

```
IDL>LoadCT,0
IDL>Window,0,XSize=600,YSize=250
IDL>TV,scan
IDL>Plot,Histogram(scan),/NoErase,Max_Value=5000,$
      Position=[0.5,0.15,0.95, 0.95]
```

可以看到大多数像素值落在 50～100 之间。将像素分布至整个颜色范围内，使得每种颜色值都有大致相同的像素个数，键入：

```
IDL>equalized=Hist_Equal(scan)
```

为查看新的像素分布柱状图 和 HISTOGRAM-EQUALIZED 图像，键入：

```
IDL>Window, 1, XSize=600, YSize=250
IDL>TV, equalized
IDL>Plot, Histogram(equalized), Max_Value=5000, $
```

```
Position=[0.5, 0.15, 0.95, 0.95], /NoErase
```

直方图均衡化后的图像应如图 35 所示。

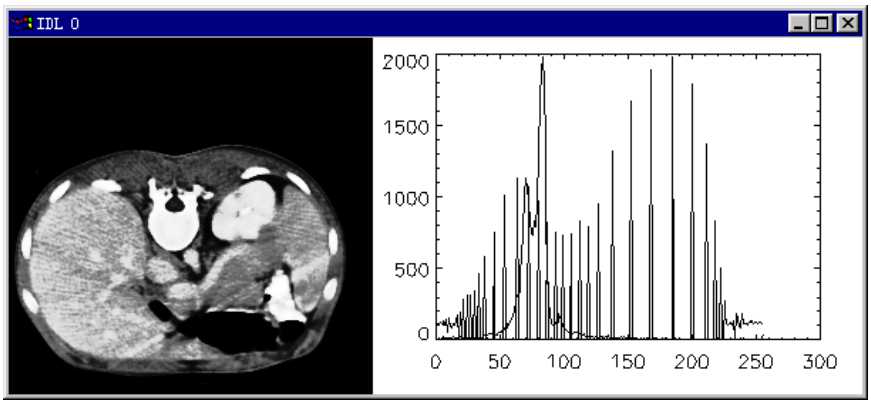


图 35　一幅直方图均衡化后的图像。象素分布扩展到了整个颜色范围

平滑图像

可以通过将每个像素值与它周围相邻像素值进行平均来平滑图像。这就是均值或核状平滑。均值平滑是由 IDL 中的功能函数 Smooth 完成的，它是在给定的奇数宽度的范围内实现等加权值平滑。例如，如果周围是 3*3 宽度，那么每个像素由它和它的周围八个像素值的平均值代替。

比较一幅没有经过平滑处理的图像和经过 5*5 核状均值平滑处理后的图像，键入：

```
IDL>Window,0,XSize=192*3,YSize=192
IDL>TV,image,0,0
IDL>smoothed=Smooth(image,5,/Edge_Truncate)
IDL>TV, smoothed, 192, 0
```

注意，与命令 Smooth 一起使用的关键字 Edge_Truncate。该关键字可复制图像边缘附近的像素，以便实现整幅图像的平滑。如果不使用该关键字，图像边缘附近的像素仅仅是简单复制，而没有平滑。

图像平滑被应用在一种称作晕光蒙片的图像处理技术中。这种技术可用作定位图像上的边缘或者是像素值突然变化的地方。这种技术非常简单：从未平滑的图像中减去平滑的图像即可。键入：

```
IDL>TV, ((image-smoothed)+255)/2.0, 2*192, 0
```

图像显示应如图 36 所示。

用 Smooth 命令，赋给相邻的像素值相等的权值来计算平均值。有时会导致出现不希望的模糊图像。另一种方式是用称为卷积的过程来平滑图像。这种技术中，一个方形滤波核和图像一起参与卷积计算。例如，在 3*3 的情况下，Smooth 命令使用的滤波核为：

```
1   1   1
1   1   1
1   1   1
```

如果给予中心像素值更大的权值，而它周围像素值的权值小一些，图像就不会那么模糊

了。例如，可以创建如下的一个核心：

```
1    2    1
2    8    2
1    2    1
```

通过 Convol 命令用上述滤波核对图像进行卷积处理，键入：

```
IDL>kernel=[[1,2,1], [2,8,2], [1,2,1]]
IDL>TV, image, 0, 0
IDL>TV, Smooth(image, 3, /Edge_Truncate), 192, 0
IDL>TV, Convol(image, kernel, Total(kernel), $
       /Edge_Truncate), 2*192, 0
```

图 36　左边为原始图像，中间为平滑处理过的图像，右边为经晕光蒙片处理后的图像

当然，可以创建任意大小的滤波核。如下是一个典型高斯分布的 5*5 滤波核：

```
1    2    3    2    1
2    7   11    7    2
3   11   17   11    3
2    7   11    7    2
1    2    3    2    1
```

可将上述滤波核应用于图像处理：

```
IDL>kernel=[[1,2,3,2,1], [2,7,11,7,2], [3,11,17,11,3], $
       [2,7,11,7,2], [1,2,3,2,1]]
IDL>TV, Convol(image, kernel, Total(kernel), $
       /Edge_Truncate), 192*2, 0
```

消除图像噪声

去除图像上的噪声是一种常规的图像处理技术。噪声来自许多方面，它经常降低图像质量。噪声的一般表现形式是黑白点相间噪声，其中一些随机的像素有极端的像素值。要了解图像平滑是怎样剔除这种噪声的，首先需要创建一幅噪声图像。用以前的图像，并键入如下的命令，将 10%的像素转换为黑白点相间噪声：

```
IDL>noisy=image
IDL>points=RandomU(seed, 1800)*192*192
IDL>noisy(points)=255
IDL>points=RandomU(seed, 1800)*192*192
IDL>noisy(points)=0
```

在原始图像的旁边创建一个窗口并显示噪声图像:

```
IDL>Window,XSize=192*3,YSize=192
IDL>TV,image,0,0
IDL>TV, noisy, 192, 0
```

IDL 中的 Median 命令是从图像上消除黑白点相间噪声的很好选择。Median 命令与 Smooth 命令类似。不同之处是 Median 命令计算相邻像素的中间值,而不是平均值。这就有两个重要作用。第一，它能删除图像中的极端值。第二，它不使那些尺寸比邻域范围大的图像边缘或特征变模糊。要看它是如何工作的，键入：

```
IDL>TV, Median(noisy, 3), 2*192, 0
```

图形显示应如图 37 所示。

图 37　左边为原始图像，中间为噪声图像，右边为用中值滤波器平滑处理后的噪声图像

增强图像边缘

一幅图像可以通过锐化或微分来增强图像边缘。IDL 提供了两个做好的边缘增强函数：Roberts 和 Sobel。还有一些其他方法也可用来增强图像边缘。例如，可以用拉普拉斯算子来对图像做卷积：

```
1    1     1
1    -7    1
1    1     1
```

因为改进了图像边缘的对比度，这也常常被称为拉普拉斯(Laplacian）锐化操作。需要了解这些方法是如何工作的，可键入：

```
IDL>TV, Sobel(image), 0
IDL>TV, Roberts(image), 1
IDL>kernel=[[1,1,1], [1,-7,1], [1,1,1]]
IDL>TV, Convol(image, kernel), 2
```

图形显示应如图 38 所示。

图 38　　三种增强图像边缘的方式。左边用的是 Sobel 方法。中间用的是 Roberts 方法。右边是用拉普拉斯算子对图像做卷积。

图像的频域滤波

频域滤波是常规的图像和信号处理技术。它可以用来平滑处理图像、锐化图像、降低图像的模糊程度，和恢复图像。

频率域滤波有如下三个基本步骤：

1. 用快速傅里叶变换（FFT）将图像从空间域转变为频率域。
2. 将转换后的图像乘以一个频率率滤波器。
3. 将滤波后的图像反变换回为空间域。

这些步骤在 IDL 中是用快速傅里叶变换（FFT 函数）完成的。（如果 FFT 函数的第二个位置参数为-1，则图像由空间域转变为频率域。如果参数为 1，则图像将从频率域反变换回空间域）。频率滤波命令的一般形式如下：

```
filtered_image=FFT(FFT(image, -1)*filter, 1)
```

在这种情况中，image 既可是一维矢量，也可以是一幅二维图像。滤波器是用来滤波图像中某些特定频率的一维矢量或二维数组。下面将详细介绍。

创建图像滤波器

在 IDL 中用基于数组的操作和函数很容易创建图像数字滤波器。许多普通的滤波器利用了所谓的频率图像或欧氏距离图的优点。一幅二维图像的欧氏距离图是一个与图像有同样大小的数组。距离图的每个像素被赋给一个值，这个值等于它到二维数组最近的角的距离。在 IDL 中的 Dist 命令用作创建欧氏距离图或频率图像。要查看一幅简单的距离图，可键入：

```
IDL>Surface,Dist(40)
```

在频域滤波中所用的滤波器一般为 Butterworth 频率滤波器。如下的方程给出一个低过 Butterworth 频率滤波器一般形式：

$$filter=1/[1+C(R/R_0)^{2n}]$$

其中，常量 C 等于 1.0 或 0.414[这个值将滤波器的幅度在 $R=R_0$ 时定义为 50%或 1/Sqrt(2)],R 为频率图像，R_0 为给定的滤波器截止频率（实际中由像素宽度代替），n 是滤波器的阶数，通常为 1。

高过 Butterworth 滤波器由如方程给出：

$$filter=1/[1+C(R_0/R)^{2n}]$$

要将频域滤波器应用到图像中，可用命令 LoadData 来打开图像 Earth Mantle Convection。这是一个 248*248 的二维数组。

```
IDL>convec=LoadData(11)
```

键入如下命令来打开一个窗口，装入颜色表 Standard Gramma II，并在左上角显示原

始图像：

```
IDL>Window,0,XSize=248*2,YSize=248*2
IDL>LoadCT,5
IDL>TV,convec,0,248
```

在频域滤波中的第一步是用函数 FFT 将图像由空间域转换为频率域，键入：

```
IDL>freqDomainImage=FFT(convec,-1)
```

通常，低频项代表一般的图像形状，高频项对应于图像的一些细节。浏览频率域的图像通常是没有意义的，但有时对观察频域图像的功率谱有用。

功率谱是一幅频域图像中不同组成部分的幅度图。与源点（通常代表图像的中心）不同的距离代表不同的频率，相对源点不同的方向代表在原图像特征的不同方向。每个位置的功率表明该频率的大小和以在图像中的方向。功率谱对分离图像中的周期性结构或噪声是特别有用的。功率谱的幅度通常用对数座标表示，因为功率从某个频率到下一频率的变化非常大。

计算出这幅对流图像的功率谱，并将其显示在原始图的相邻位置上，可键入：

```
IDL>power=Shift(Alog(Abs(freqDomainImage)),124,124)
IDL>TV,power,248,248
```

功率谱中的对称性表明这个图像上在越来越多的频率中包含了许多周期性结构。输出应该类似于图 39)。这个练习的目的是过滤掉图像中较高的频率。

接下来的一步是用频率滤波器转换图像。ButterWorth 低通滤波器用于滤出图像内的高频成分。这些高频成分为图像提供详细信息，所以最终的结果是完成图像的平滑处理。创建低通频率滤波器可键入：

```
IDL>filter = 1.0 / (1.0D + Dist(248)/15.0)^2
```

注意，截止频率宽度是 15 个像素。这是足以删除高频的一半。在图 39 中功率谱可以看到。

使用该频率滤波器，再将图像由频率域转换回空间域，最后显示滤波后的图像。键入：

```
IDL>filtered = FFT(freqDomainImge * filter, 1)
IDL>TV, filtered, 0, 0
```

为了让自己看到滤掉高频成分的图像，可以显示滤波后图像的功率谱，并在其旁显示滤波后的图像，键入：

```
IDL>filteredFreqImg = FFT(filtered, -1)
IDL>power = shift(Alog(Abs(filteredFreqImg)), 124, 124)
IDL>TV, power, 248,0
```

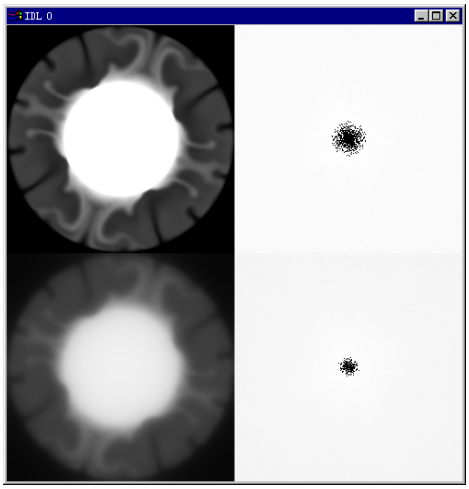


图 39 频域滤波器的图解。在图的上半部分是没有滤波的图像，左边是它的功率谱。在图的下半部分是滤波后的图像，相邻的左边是它的功率谱。注意，大约一半高频成分在滤波后的图像中已经被消除，消除了很多图像细节信息，也平滑了图像

第四章　图形显示技术

本章概要

在学会怎么显示线画图、曲面图和等值线图后，就可以用自己的想象力和创造力来显示数据了。本章给出了许多特殊的可视化技术以增强数据显示能力。笔者没有打算详细地描述 IDL 中每一种可能的技术。本章将介绍一些更普遍的技术，其目的是为读者提供工具和概念，以便帮助读者创造自己特定的数据显示。

读者通过本章将学会：

1. IDL 如何运用颜色
2. 怎样在 IDL 中创建和保存颜色表
3. 怎样按规范修改坐标轴的注记
4. 怎样用 IDL 处理坏的或残缺的数据
5. 怎样建立三维坐标系并在里面显示数据
6. 怎样组合简单图形显示
7. 怎样用动画显示图形
8. 怎样将 XYZ 数据格网化以便图形显示

IDL 的颜色运用

IDL 的颜色由三种特殊值组成。我们称这些数值为一个三色组，将其写成（R,G,B）即红、绿、蓝，其中红、绿、蓝代表红光、绿光、蓝光作用于该显示颜色时的数量，每个值的范围都在 0 到 255 之间。这样，一种颜色可由 256 级的红色，256 级绿色和 256 级蓝色组成。这就是说 IDL 能显示 256*256*256，或者说超过 167,000,000 种颜色。举例来说，黄色由亮红和亮绿组成，但没有蓝色。代表黄色的三色组写作（255,255,0）。

过去在 IDL 里常常用一种索引号通过查颜色表来获得颜色三色值。现在，由于越来越多地使用 24 位图形卡，可直接表示三色值。如果使用索引，所查寻的这个表就被称作颜色查询表（简称为颜色表）。一个颜色表由三列数组成，一列代表红色值，一列代表绿色值，一列代表蓝色值。典型地，这些数列被称为矢量。当用 IDL 装载颜色表时，所做的就是选择正确的数值放进这些列或矢量之中。请看这个概念的图解（图 40）。

使用索引颜色模式和 RGB 颜色模式

除了了解一个颜色号代表一种颜色的三色值和颜色表被用来决定三色值之外，必须意识到在 IDL 里有两种颜色模式。索引颜色模式用于 8 位显示器，RGB 颜色模式用于 24 位显示器。（IDL 在 PC 机和 Macintosh 计算机上同时使用了一种修改过的 RGB 颜色模式，这两种计算机支持 16 位颜色）。

两种模式都能用一个颜色查找表来决定用于显示的特定颜色。（当颜色分解关键字关闭时，RGB 颜色模式就用颜色查找表。否则，RGB 颜色模式就用三色值直接指定颜色）。索引颜色模式也将索引颜色号和颜色表中的特定位置联系起来，而 RGB 颜色模式直接指定颜色。被链接到特定颜色表某个位置的颜色被称作动态颜色显示。直接显示的颜色常被称为静态颜色显示。在大多数情况下（有例外），8 位显示是动态显示，24 位显示是静态显示。

动态和静态颜色显示间最重要的区别就在于，如果用动态显示并且改变装在颜色表中特定位置的数字，索引指向那个位置的像素就会立即改变颜色。而用静态显示时，像素的颜色直接被确定，在某种程度上是永久不变的。它们不会受颜色表值中后来改变的影响。（这并不总是绝对的，阅读下面关于直接颜色视觉级的讨论。）

也许这个问题显得很复杂，但它真的很有价值。它意味着采用 RGB 颜色模式的系统能同时显示所有的 167，000，000 种颜色，而采用索引颜色模式的系统只能同时显示 167，000，000 色调色板中的 256 种颜色。

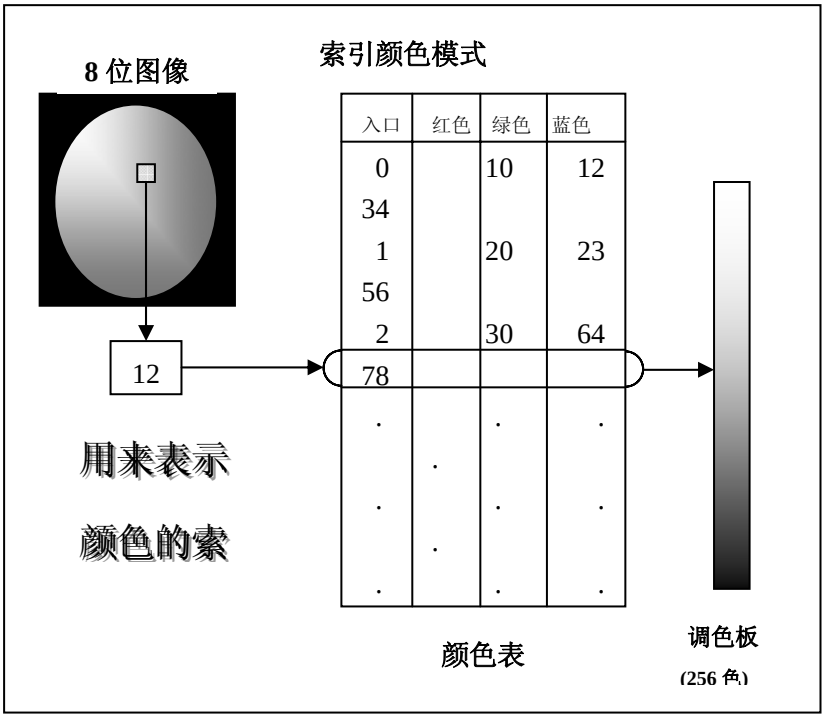


图 40　8 位像素值的索引颜色模式。像素值作为索引号输入到颜色表。在颜色表中找到的红色、绿色、蓝色值决定了与此像素值相关的或由此像素值所索引的特定三色值

通过使用 Device 命令中的新关键字，可以知道正在使用的颜色模式的类型，这些新关键字是在 IDL5.1 中新增加的。这些关键字是 Get_Visual_Depth 和 Get_Visual_Name。这些都是将值返回给指定 IDL 变量的输出关键字，键入：

```
IDL> Device, Get_Visual_Name=thisName, $
        Get_Visual_Depth=thisDepth
IDL> Print, thisName, thisDepth
      TrueColor          24
```

视觉名称通常为假彩色，直接颜色或真彩色。视觉深度通常为 8，16 或 24，它是指用于决定这个视觉级类中某个颜色的位数。

8 位假彩色视觉级表明正在使用索引颜色模式和动态颜色显示。24 位真彩色或直接颜色视觉级表明正使用 RGB 颜色模式。直接颜色视觉级有时可能是动态颜色显示，但这是窗口管理的功能，且偶尔可由用户配置。直接色视觉级色通常用静态颜色显示。真彩色视觉级总是使用静态颜色显示。（直到 IDL5.1 这个属性才保持了跨平台运行的一致性。）

静态与动态颜色视觉

假彩色视觉级是一种动态颜色视觉。这意味着如果改变色查询表的某种颜色，显示器上任何使用该颜色索引号的像素都会立即改变颜色。一般来说，装载一种新颜色表将立即改变显示器上所有图形的颜色。

真彩色视觉级是一种静态颜色视觉。这意味着改变颜色表里的一种特定颜色决不会影响已经在显示设备上的图形或颜色，因为那些颜色是直接用 RGB 三色值表达的。

直接颜色视觉级最难表述。直接颜色视觉级仅用于 UNIX 系统的机器上。每个机器制造商对直接颜色视觉级的含义都有不同的看法。但在理论上直接颜色视觉级应该集两者的优点于一身：即 24 位颜色系统表现得仿佛是一个动态颜色视觉。在实践中，很少看到它运行得很好。最普遍的问题是直接颜色视觉级通常提供私有的颜色图，要求将图形窗口作为当前窗口，并在窗口中装载正确的颜色。当完成时，其他窗口会消失。这就是常见的“颜色闪烁问题”，它是 X 窗口管理器处理颜色表的方式造成的结果。近来硬软件的发展已消除了许多这样的问题，但它们仍常常会被碰到。通常，可用 8 位假彩色视觉级或 24 位真彩色视觉级，因为这样能保证在多个平台上正确地工作。

当 IDL 启动时，其所用的视觉级正常情况下是按缺省值给定的，即可以从.Xdefaults 文件中获取信息（当 IDL 运行在 UNIX 机器上），也可以按 IDL 的一般规则来给定视觉级。（这种规则要求 IDL 查询硬件支持何种视觉级，并指定可获得的“最高”视觉级和视觉深度）。这个指定的缺省值可在 IDL 中指定视觉级和视觉深度时给定。（IDL 的微机版本是通过装在机器上的图形卡及其配置来给定这些参数的，这不能从 IDL 内部改变。）指定值必须在图形窗口打开前确定，并可作用于 IDL 运行期间。

下面是基于 UNIX 系统机器上的典型的视觉级赋值语句：

```
IDL> Device, PsuedoColor=8
IDL> Device, TrueColor=24
```

在 8 位显示器上指定颜色

如果正在使用索引颜色模式，可指定一种特定的颜色做为索引号进入颜色表。IDL 在表里寻找此颜色索引号，并在颜色表内找出红、绿、蓝色列中的值作为确定该颜色的三色值。例如，假设将代表黄色的三色值（255，255，0）装载到颜色表的第 10 项。可用 TvLCT 命令来实现：

```
IDL> TvLCT, 255, 255, 0, 10
```

如果想用黄色画图，可以用 Color 关键字指定颜色索引号，如下：

```
IDL> data = LoadData(1)
IDL> Plot, data, Color=10
```

类似地，任何值（即索引）为 10 的图像像素都将用同样的黄颜色显示。

如果正在使用索引颜色模式，可以简单地装载新的三色值进入颜色表的第 10 入口，就可容易地更改图形颜色。例如，可以像这样装载绿色：

```
IDL> TvLCT, 0, 255, 0, 10
```

显示图形的颜色立即改变了，因为此颜色已被索引到颜色表。

在 24 位显示器上指定分解后的颜色

如果用 24 位显示器，情况稍微复杂一些。当 IDL 用 RGB 颜色模式时，在缺省值情况下，采用“分解后的”颜色。这就是说 IDL 不是将颜色索引作为单独的索引号传入颜色表，而是试图将索引分解为三个单独的索引号传入颜色表。它这样处理是假设该索引为一个 24 位长整型数。IDL 使用最低的 8 位数作为红色索引，中间的 8 位作为绿色索引，最高的 8 位作为蓝色索引。那么上面命令里的数字 10 被看做在红色矢量的第 10 项，但在绿色矢量和蓝色矢量中却是第 0 项。将在图 41 里看到分解索引值的图解。

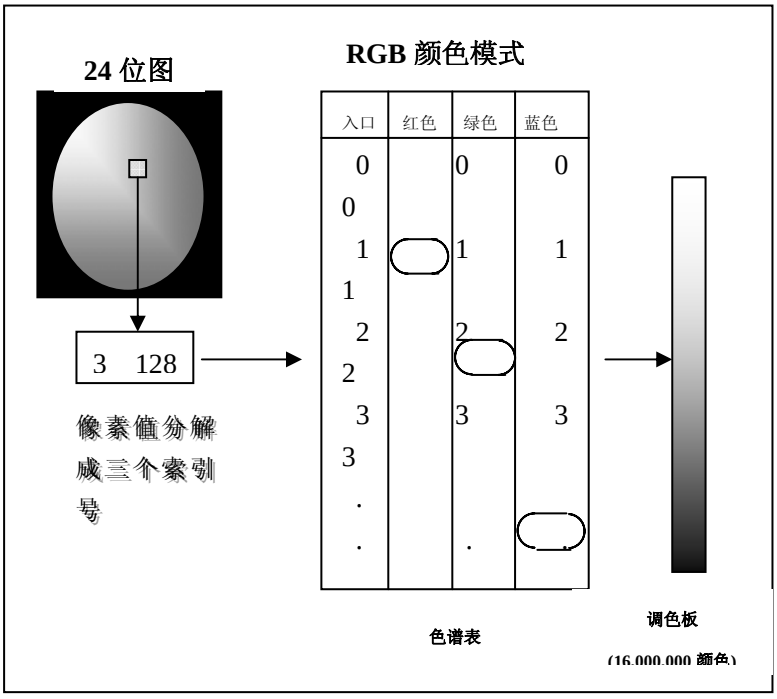


图 41　RGB 颜色模式用 24 位像素值来单独指定一种颜色的 RGB 分量。如果颜色矢量包括从 0 到 255 的值，所有的 16,700,000 种颜色都能在调色板里同时获得。

当上述颜色表被装载时（如图 41 所示），所有的 16,700,000 种颜色可立即被 IDL 存取。例如，想在 24 位系统上用黄色画图，可选择 24 位整数，其中 8 个最低位设为 1（全红），8 个中间位设为 1（全绿），8 个最高位设为 0（无蓝色）。这时，代表黄色的数字用长整数 65535 表示。为了在 24 位颜色显示设备上画出上述图形，可键入：

```
IDL> Plot, data, Color=65535L
```

因为大多数人都不能熟练地使用 24 位的字节数，此数字有时被表示成十六进制符号。那么用两个数字（ 0 － F ）就足以一次设置 8 位（也就是，256 或 2^8 能由两个十六进制数设置）。例如用十六进制符号表达全红和绿，但没有(不能表达)蓝：

```
IDL> Plot, data, Color=’00FFFF’xL
```

为了在碳灰色（70,70,70）背景上，画出含有绿色（0,255,0）标题的黄色（255,255,0）图形，可使用以下十六进制符号：

```
IDL> Plot, data, Color=’00FFFF’xL, Background=’464646’xL
IDL> XYOutS, 0.5, 0.95, Align=0.5, /Normal, ‘Plot Title’, $
        Color=’00FF00’xL
```

如果对 24 位符号或十六进制符号都不太熟悉，可能会想用程序 Color24 来获得 24 位整数值。这个程序在已经下载的本书配套程序中，可以将任何 RGB 三色值（写成三个元素的 IDL 矢量）转换成等值的 24 位整数值。例如，在 24 位系统上，如果想用程序 Color24 来画黄色图形，命令如下：

```
IDL> Plot, data, Color=Color24([255,255,0])
```

如果想写出能运行在 8 位或 24 位显示设备上的代码，代码可以如下所示：

```
Device, Get_Visual_Depth=thisDepth
IF thisDepth GT 256 THEN BEGIN
   Plot, data, Color=Color24([255,255,0])
ENDIF ELSE BEGIN
   TvLCT, 255, 255, 0, 100
   Plot, data, Color=100
ENDELSE
```

在 24 位显示设备上指定没有分解过的颜色

不能用分解过的颜色通过 RGB 颜色模式作索引。例如，可能希望以 8 位显示设备上装载颜色表的方式装载颜色表，并能在 8 位显示设备和 24 位显示设备上使用同样的代码。如果关闭颜色分解功能，这是可以实现的。可通过这样的设备命令完成：

```
IDL> Device, Decomposed=0
```

在这种情况下，IDL 是以处理 8 位颜色索引那样的方式处理像素值或颜色索引。也就是说，索引被用来存取在红、绿、蓝颜色表矢量里相同的项。这样，在 24 位显示设备上，用没有分解的颜色，也可用下面的命令画出黄色图形：

```
IDL> TvLCT, 255, 255, 0,10
IDL> Plot, data, color=10
```

但图形的颜色是直接表达的，这一点非常重要。它不被索引到颜色表里的入口位置，如果在第 10 入口处改变颜色表的值，图形颜色将完全不受影响。

```
IDL> TvLCT, 0, 255, 0, 10
```

然后用画线命令（或者，通常重新显示图形）来查看新颜色是否生效。

```
IDL> Plot, data, color=10
```

在 Windows 系统中，无论颜色分解如何设置，IDL5.1 以前的版本总是在 24 位显示设备上显示 8 位图像，就象正在使用没有分解过的颜色。也就是说，8 位像素值被用来索引颜色表中全部三种矢量的相同项。这种特点（PC 版本里的一种长期 BUG）在 IDL5.1 里得到改变，从而使得这种特性和其他平台保持了一致性。然而，这种改变使得写出在 8 位和 24 位环境中同样工作的 8 位图像显示程序时稍微困难一点，需要在 24 位环境中增加一些设置。

决定颜色分解的开与关

自从 IDL5.1.1 开始，没法保证在 24 位显示设备里的颜色分解是开还是关。这意味着如果想让分解颜色是关或者是开（例如显示 8 位图像时想要关闭颜色分解，在显示 24 位图像时想打开颜色分解），必须在调用图形显示命令前设置它：

```
IDL> Device,   Decomposed=0
IDL> TV,   image8bit
IDL> Device,   Decomposed=1
IDL> TV,   image24bit
```

Device 命令可以引入 Get_Decomposed 关键字，它用来记录目前的“分解”状态。

```
IDL> Device,   Get_Decomposed=usingDecomposed
IDL> Print,   usingDecomposed
```

注意，关于 24 位颜色方面，微机版的 IDL5.2 仍有 BUG 出现。如果正在使用 24 位显示

设备，且想用正确的图像颜色显示 24 位图像，必须装入颜色表 0 或将 Decomposed 关键字设为 1。如果将 Decomposed 值设为 0，那么即使是 24 位的图像颜色也将遍历颜色表矢量。注意，通过下载的本书附带程序 TvImage 可以根据图像是 8 位(Decomposed=0)或 24 位(Decomposed=1)来正确地设置 Decomposed 值。

在 24 位显示设备上装载颜色表

现在读者可以了解到当在 8 位显示设备上使用索引颜色模式时像素颜色被直接索引到颜色表。换句话说，如果通过在 IDL 里装载颜色表来改变颜色表里的值，那么与那些索引联系在一起的颜色也将被改变。如果想同时分别用不同的颜色表显示几个图像，必须将可获得的颜色表索引分成不同的区，每个区装载不同的颜色（参看 67 页的“用颜色表分段表示图像”的有关章节）。从实际来说，在用完索引数字前，可能至多有四或五幅具有不同颜色表的图像。

24 位显示设备的突出优点之一就是可以一次在显示设备上实际显示很多的图像，每一个图像都可用不同的颜色表。（记住只有关闭颜色分解后，在 24 位显示设备上装载颜色表才有意义。记住当 IDL 启动时，它的缺省值是打开的。）

如果装载不同的颜色表会怎样呢？想改变很多图像的颜色吗？可能不容易，因为每幅图像都是用它自己一套颜色，它们都是直接指定的。

如果有一幅图像显示在 24 位监视器上，用 LoadCT 命令或 XloadCT 工具改变颜色表，那么新的颜色不会生效，除非重新绘制图像，并将图像像素通过颜色表重新转换成特定的直接颜色。想了解如何写代码以改变 24 位显示设备上的颜色表以及让图形自动重新显示，请阅读 274 页 “在 24 位显示器上改变颜色表”的有关章节。

获得颜色表的拷贝

有两种方法取得当前颜色表中红、绿、蓝值的拷贝。一个方法是声明公共块 Colors，既可在想获得颜色表的 IDL 主程序中声明，也可在任何 IDL 程序或函数里声明。调用过程如下：

```
COMMON Colors, r_orig, g_orig, b_orig, r_cur, g_cur, b_cur
```

注意,颜色表必须在 IDL 的运行时装载，以便定义公共块中的变量。

约定从前面的三个变量中获得当前颜色表的颜色。如果想修改这些颜色，可将修改后的颜色矢量放进最后三个变量。装载颜色表用 TvLCT 命令。如果想反转颜色表里的颜色，可键入：

```
IDL> COMMON Colors, r, g, b, rr, gg, bb
IDL> rr = Reverse (r)
IDL> gg = Reverse (g)
IDL> bb = Reverse (b)
IDL> TvLCT, rr,gg, bb
```

另一个获得颜色表值的方法是用带 Get 关键字的 TvLCT 命令：

```
IDL> TvLCT, red, green, blue, /Get
```

在这个例子中，变量 red, green 和 blue 为输出变量，被赋予了颜色表中相应的值。注意这些变量拥有与正在使用的 IDL 的颜色数目一样多的元素。通过第一个打开的 IDL 图形窗口，可以确定正在使用的颜色数，键入：

```
IDL> Print, !D.Table_Size
```

注意，Windows 版的 IDL 在 24 位显示设备里面，这个数字可能不精确，因为它将取决

于颜色分解是开还是关。详细信息参考 89 页的“决定颜色分解的开与关”。

修改和创建颜色表

有两个用于颜色表操作的基本命令：XloadCT 和 Xpalette。通过这两个命令，可以修改和创建颜色表。XloadCT 允许用不同的方法扩展颜色。（进入 Function 模式并点击 Add Control Point 按钮几次。用鼠标移动控制点来看如何影响颜色表）。它也允许以交互方式进行 Gamma 矫正。（一的 Gamma 值是一个线性斜坡函数。小于或大于一的 Gamma 值是不同形状和陡峭的指数斜坡函数。）

Xpalette 命令允许通过设置滚动条的端点色和插入干涉值来修改和创造自己的颜色表。单个颜色也可在这个程序里被修改。注意是在索引颜色系统而非 RGB 颜色系统里用 Xpalette 指定颜色。当然，无论是用什么颜色系统指定颜色，被装载进颜色表中的均是红、绿、蓝色矢量。

创建自己的颜色表也很容易。下面就是一个能创造两种端点色之间的任意颜色数的颜色表，名为 Make_CT 的简单小程序（不带错误检查！）打开文本编辑器并键入：

```
FUNCTION MAKE_CT, begColor, endColor, ncolors
ScaleFactor = FindGen(ncolors) / (ncolors – 1)
Colors = BytArr(ncolors, 3)
FOR j=0,2 DO colors[*,j] = begColor[j] + (endColor [j] $
    - begColor [j]) * scaleFactor
RETURN, colors
END
```

编辑此程序须键入：

```
IDL> .Compile make_ct
```

打开 World Elevation Data 图像，并显示在窗口中：

```
IDL> image = LoadData(7)
IDL> Window, Xsize=360, Ysize=360
IDL> LoadCT, 0
IDL> TVScl, image
```

假如想要从黄色（255,255,0）到蓝色（0,0,255）的颜色表。可以用程序 Make.CT 创造并如下装载：

```
IDL> yellow = [255, 255, 0]
IDL> blue = [0, 0, 255]
IDL> TvLCT, Make_CT(yellow, blue, !D.Table_Size)
```

假如想用从黄到绿到蓝的 150 种颜色来显示此图。可以这样做：

```
IDL> scaledImage = BytScl(image, Top=149)
IDL> green = [0, 255, 0]
IDL> TvLCT, Make_CT(yellow, green, 75)
IDL> TVLCT, Make_CT(green, blue, 75), 75
IDL> TV, scaledImage
```

注意在选择颜色的过程中可能产生很多混乱数据。要当心。如果您有兴趣，可以参考 Bernice E. Rogowitz 和 Lloyd A. Treinish 在 *Computers In Physics*, 10(3):268,1996 上所著的名为 “How Not to Lie with Visualization”的论文。如果您对颜色和数据显示感兴趣，读阅 Edward Tufte 的 *The Visual Display of Quantitative Information and Envisioning Information*。这些书可能改变读者编写 IDL 程序的方法！

保存自己的颜色表

假如对刚刚创建的颜色表很满意并想保存它。可以用 TvLCT 命令得到颜色值：

IDL> TvLCT, r, g, b, /Get

查看矢量有多长可键入：

IDL> Help, r, g, b

可以看到它们和正在 IDL 里应用的颜色数目一样长。但上述颜色表的颜色只是在前 150 个值里。可用颜色值的数目重新限定矢量：

IDL> r = r(0:149)

IDL> g =g(0:149)

IDL> b = b(0:149)

现在如果愿意可以保存矢量，但大多数颜色表矢量在长度上有 256 个元素。很容易使这些矢量达到该长度：

IDL> r = Congrid(r, 256, /Interp)

IDL> g = Congrid(g, 256, /Interp)

IDL> b = Congrid(b, 256, /Interp)

如果愿意，可以将这些矢量写进文件，但用 IDL 的保存命令将它们保存在 IDL 的保存文件中则更容易：

IDL> Save, File='mycolors.sav', r, g, b

为了验证是否这些矢量已被正确保存，可装载另一个颜色表并消除这三个变量：

IDL> LoadCT, 0

IDL> DelVar, r, g, b

当准备用这些矢量时，用 Restore 命令重新恢复它们。注意它们以保存时同样的变量名返回。如果在 IDL 定义了的同名变量（象在此处做的），此变量将被覆盖。这意味着输出变量名时需要仔细考虑：

IDL> Restore, 'mycolors.sav'

IDL> Help, r, g, b

使用这些变量时，必须将这些变量重置为 IDL 运行时的颜色数目。命令如下：

IDL> r = Congrid(r, !D.Table_Size)

IDL> g = Congrid(g, !D.Table_Size)

IDL> b = Congrid(b, !D.Table_Size)

IDL> TvLCT, r, g, b

与其每次想调入所保存的一个颜色表时都要键入这些命令，不如写个小程序来自动调用更容易。如果总是用变量名 r, g 和 b 保存 RGB 矢量，可以编写名为 CT_Load 的程序。（在这个小程序里没有错误检查！）打开文本编辑器并键入：

```
PRO CT_Load, filename
IF N_Params( ) EQ 0 THEN filename = 'mycolors.sav'
Restore, filename
r = Congrid(r, !D.Table_Size)
g = Congrid(g, !D.Table_Size)
b = Congrid(b, !D.Table_Size)
TvLCT, r, g, b
END
```

保存 ct_load.pro 文件并编译它：

IDL>.Compile ct_load

现在无论什么时候想调用这个颜色表，只须键入：

IDL> CT_Load

创建自己的轴标注

IDL 缺省的基本轴标注可能不能满足您的显示要求。幸运地是 IDL 提供了许多方法来增加轴的基本注释特性。本节将讲述一些创造更复杂的轴标注的技巧。

调整轴刻度间隔

有时 IDL 内部的轴标注算法不会按照最有利于数据方式来划分的。可以用[XYZ]Ticks 关键字来控制主刻度间隔的数目。装入随本书附带的数据集 Time Series Data。键入以下命令：

IDL> curve = LoadData(1)

IDL> LoadCT, 5

IDL> Plot, curve

注意 X 轴被划分为五个主刻度间隔。可以修改为十个主刻度间隔，键入：

IDL> Plot, curve, Xticks=10

输出结果见图 42.

您会注意到小刻度的增加导致轴上刻度有点凌乱。既然对这样精细间隔的轴刻度不感兴趣，则可以改变小刻度的数目。可用[XYZ]Minor 关键字设置小刻度的数目。例如，可以只在两个主刻度之间设一个小刻度。可能想将 Xminor 关键字设为 1，以便得到每个主间隔之间有一个小刻度，但这是不正确的。如果 Xminor 关键字设为 1，所有的小刻度都会消失。为了得到想要的小刻度的数目，须将 Xminor 关键字设为比想要的数目大 1。键入：

IDL> Plot, curve, Xticks=10, Xminor=2

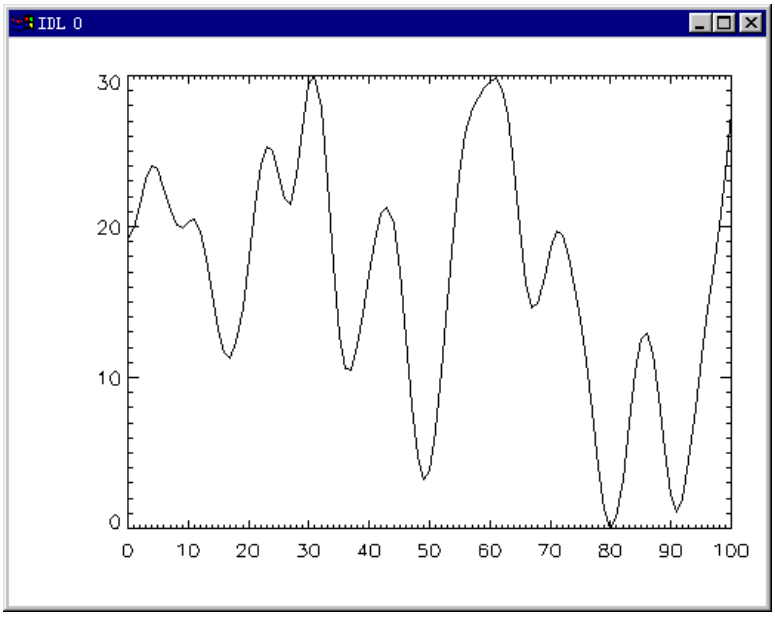


图 42　　主刻度间隔的数目随 Xticks 关键字改变

格式化轴的标注

影响轴标注的另一个方法是改变轴的标注格式。例如，X 轴标现在被表示为整数。可能想用三位数的整数来表示。可以通过将 XtickFormat 关键字设定为想要的特定格式来达到这个目的。例如，可以键入：

IDL> Plot, curve, XtickFormat='(I3.3)'

用小数点后面带两位小数的浮点值来写标注，键入：

IDL> Plot, curve, XtickFormat='(F6.2)'

也可以用特殊字符串作为刻度。这可以用 TickName 关键字完成，最多可达 30 个字符串元素。例如，可以用星期几标识图形：

IDL> labels = ['MON','TUE','WED','THU','FRI','SAT']

IDL> Plot, curve, XTickName=labels

输出结果如图 43 所示。

通过将轴刻度格式设置为(A1)，可隐藏轴标注：

IDL> Plot, curve, XtickFormat='(A1)'

编写刻度格式函数

另一个格式化刻度的方法是编写一个函数，用所想要的格式来格式化刻度。如果传给 [XYZ]TickFormat 关键字的参数是函数名，那么 IDL 将注释标注时将调用那个函数。

例如，假设在这个图形上想要的 X 轴标是日期，写作 *25 MAR 97*。可以编写名为 Date 的函数来完成格式化。这个函数必须定义三个且只能有三个位置参数。它们是轴参数，索引数，和标注值。当需要格式化轴标注时，IDL 会用这三个位置参数来调用该函数。函数的返回值必须是字符串变量。

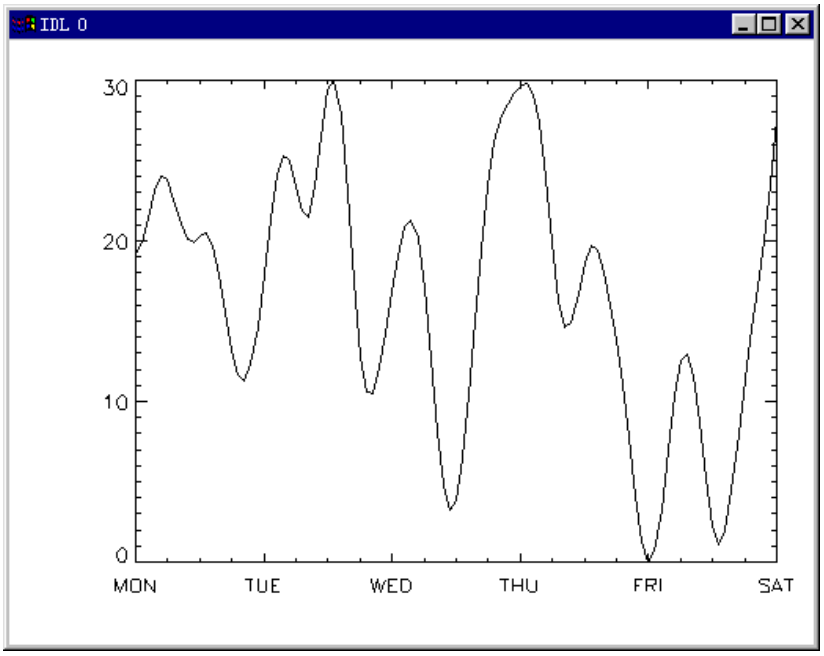


图 43 可以通过[XYZ]TickNames 关键字用字符串标识轴

轴参数为 0，1 或 2 分别表示 X、Y 或 Z 轴。索引数是特定轴标的个数。这个参数程序员很少在函数里用到。标注值是用于轴标的常规值。工作就是用标注值来计算或格式化该函数返回的新值。就是这个返回值用来为特定轴索引数标注轴的。

通过一个例子可更容易了解如何编写这个程序。打开文本编辑器键入这个简短的 Date 程序。

```
FUNCTION DATE, axis, index, value
MonthStr = [‘Jan’, ’Feb’, ’Mar’, ’Apr’, ’May’, ’Jan’, ’Jul’, $
    ‘Aug’, ‘Sep’, ‘Oct’, ‘Nov’, ‘Dec’]
CalDat, LONG(value), month, day, year
Year = StrMid(StrTrim(year,2), 2, 2)
RETURN, StrTrim(day, 2) + ‘ ‘ + monthStr (month-1) + ‘ ‘ $
    + year
END
```

编译程序 Date，以便以下的代码用来格式化 X 轴刻度标识：

```
IDL> .Compile date
```

注意这个程序的 CalDat 命令。此程序接受代表某个日期的 Julian 数值，并返回与此 Julian 数值相关的正确的日，月，年数。这个信息可被用来正确地格式化标识。为了解它是如何工作的，可键入：

```
IDL> Window, XSize=500, YSize=350
IDL> startDate = Julday (1, 1, 1991)
IDL> endDate = Julday(6, 23, 1995)
IDL> numTicks = 5
IDL> sizeCurve = N_Elements(curve)
IDL> steps = Findgen(sizeCurve) / (sizeCurve-1)
IDL> dates = startDate+ (endDate+1 – startDate) * steps
IDL> !P.Charsize = 0.8
IDL> Plot, dates, curve, XtickFormat=’Date’, $
        Xstyle=1, Xticks=numTicks, $
        Position= [0.15, 0.15, 0.85, 0.95]
```

输出结果见图 44。若想更多地了解用日期标识轴的情况，可参考 IDL 库函数 Label_Date。此函数功能很象刚刚编写的程序 Date 的功能。

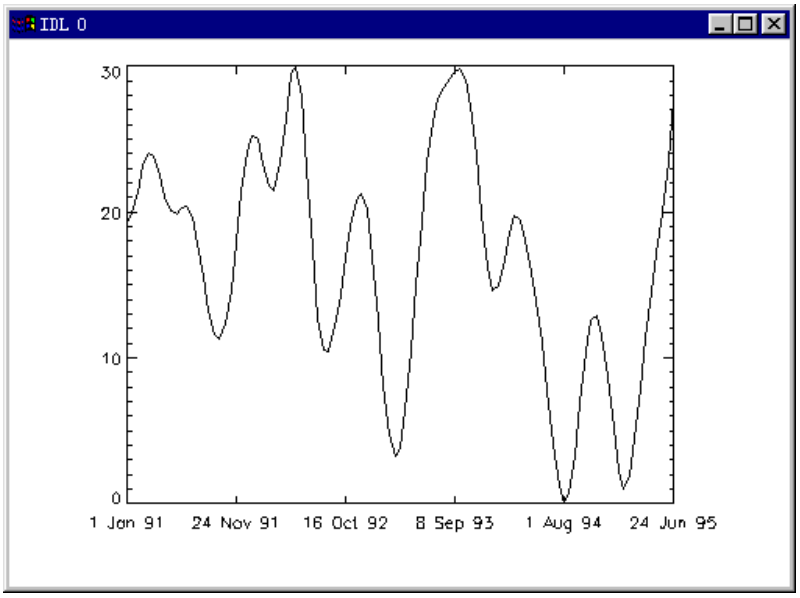


图 44　通过用户编写的函数格式化刻度

正如所见，这些标识很长，且有挤在一起的危险。可能想用另外的方法显示日期。例如，可能想将它们相对轴旋转 45 度。可是，刚刚编写的刻度格式化函数不能奏效，将不得不借助更有效的方法，用 XYOutS 命令放置标识。

然而，仍可以用程序 Date 来格式化字符串。为完成这项工作，必须画一幅 X 轴不带标注的图，并需要一个有正确刻度值的矢量。可以通过将轴的刻度格式设为(A1)来隐藏轴标注。可用 Xtick_Get 关键字以矢量的形式得到刻度值。例如图形可以这样画（Position 关键字用来给轴标留下空间）：

```
IDL> Plot, dates, curve, XtickFormat='(A1)', Xstyle=1, $
        Xticks=numTicks, Xtick_Get=tickValues, $
        Position= [0.1, 0.2, 0.85, 0.95]
```

然后，用 XYOutS 命令将标注添上。可以用![XY].Window 系统变量来找出 X 和 Y 轴端点的归一化坐标。这个信息对于正确定位标注是非常关键的。代码如下：

```
IDL> ypos = Replicate (!Y.Window[0] – 0.04, numticks+1)
IDL> xpos = !X.Window[0] + (!X.Window(1) - !X.Window[0] )* $
        FindGen (numTicks+1) /numTicks
IDL> FOR j=0,numTicks DO XYOutS, xpos(j), ypos(j), $
        Date (0, j, tickValues [j]), Alignment=0.0, $
        Orientation= - 45, /Normal
```

输出结果如图 45 所示。

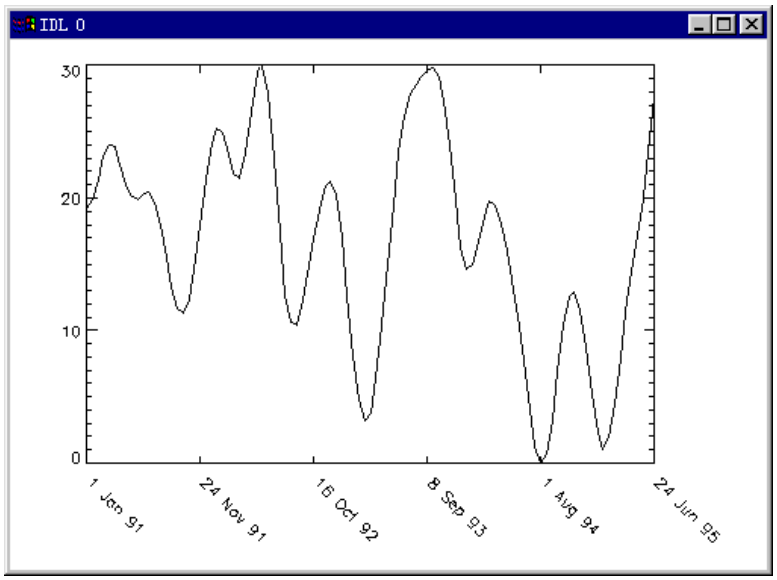


图 45 用 *XYOutS* 命令创建旋转的轴标

用 IDL 处理残缺的数据

我们知道，数据并不总是来源于性能良好的采集仪器。将原始数据处理为可用的形式通常是必要的。事实上，很多原始数据组是不完整的。许多事情都可能发生。例如，数据采集仪器关闭一小段时间；电流导致伪数据；操作者错误地操作；等等。如何用 IDL 来处理这种残缺的或坏的数据呢？

处理这种数据的一种方法是赋给它一个 NaN 值。NaN 是一种特殊的位模式，它在每种机器结构上都是不同的。运行 IDL 的机器的位模式保存在系统变量!Value 的 F_NaN 字段中。

查看它如何被使用，可用 LoadData 命令打开 Elevation Data 数据组：

```
IDL> data = LoadData (2)
IDL> !P.CharSize = 1.0
```

这个数据组是 41*41 的浮点数组。假设数据是不完整的。假设在采集数据中，在扫描此二维数据组的中间三行时，采集仪器暂时关闭。想将 NaN 值赋给这三行扫描数据，可键入：

```
IDL> badData = data
IDL> badData (*, 30:32) = !Values.F_NaN
```

现在，当显示曲面图时，IDL 将不考虑那些被标注为 *NaN* 的值，键入：

```
IDL> Surface, badData
```

输出结果如图 46 所示。

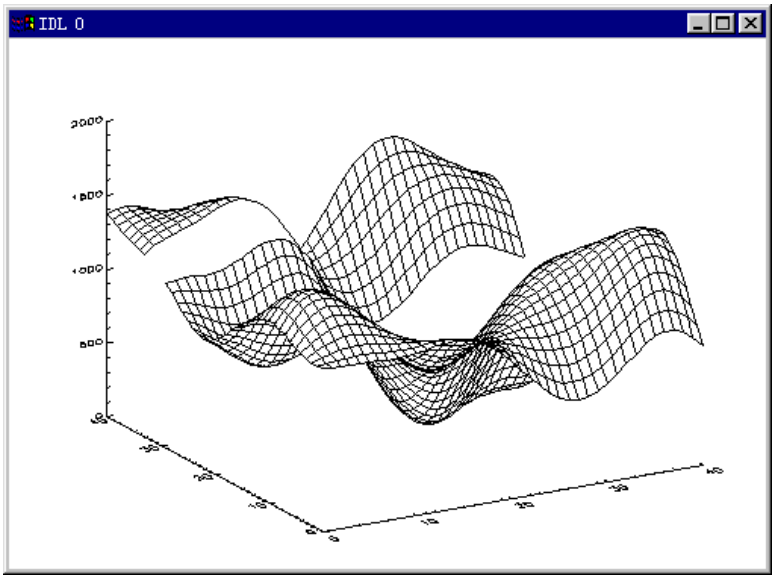


图 46　在此曲面图里，残缺的数据用 NaN 代替

除了设置 NaN 位模式外，还有另一个方法可以处理残缺的或坏的数据。这就是用大多数 IDL 图形显示命令都有的关键字 *Min_Value* 和 *Max_Value*。通过设置这些关键字，任何小于最小值或大于最大值的都可以被图形输出命令忽略。例如，在以上所用到的高程数据集里，可以只画特定范围里的那些等值线。以下是一些显示它如何工作的命令。这儿值在小于等于 400 和大于等于 1000 之间的等值线没有在图中画出：

```
IDL> Window, XSize=500, YSize=375
IDL> !P.Multi = [0, 2, 1, 0, 1]
IDL> Values = FindGen (10)*150 + 100
IDL> label = Replicate (1,10)
IDL> Contour, data, Levels=values, /Follow, C_Labels=label
IDL> Contour, data, levels=values, /Follow, C_Labels=label, $
        Min_Value=400, Max_Value=1000
IDL> !P.Multi = 0
```

图形窗口的右边部分输出结果如图 47 所示。

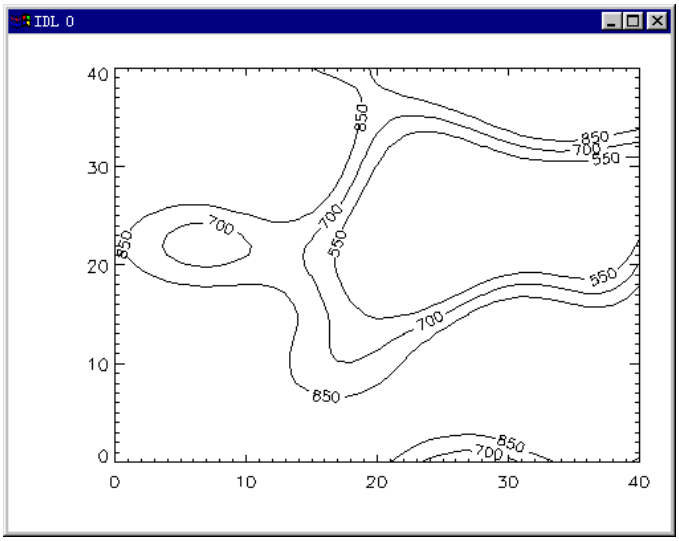


图 47　等值线可用 Contour 命令的 Min_Value 和 Max_Value 关键字消除

用 IDL 建立三维坐标系

IDL 用变换矩阵乘上三维空间的每个点，从而实现在二维显示上模拟三维坐标系。如果有这种变换矩阵，那么它将会存储在系统变量!P.T 里。如果想用 IDL 在三维空间里画图，必须首先将正确的变换矩阵装入!P.T 系统变量。接着必须确保在图形输出到显示设备之前，图形输出命令已经被该矩阵乘过。在实践中这是很容易做到的。

将三维变换矩阵装入!P.T 系统变量有几个方法。如果想严格控制矩阵的建立，可以用 T3D 命令来建立想要的三维坐标系。但除非正在做某件复杂的或超出常规的事情，否则不常用 T3D 命令。可用以下两个方法之一来装载三维变换矩阵：（1）如果想在三维空间中显示坐标轴，可用带 Save 关键字的 Surface 命令；（2）如果仅仅希望创建三维空间，而不关心坐标轴的显示问题，可以用 Scale3 命令。

建立三维散点图

假设有随机分布的三维数据，并想将它在三维空间里显示为散点图。也可以通过计算机获得随机分布的三维数据，键入：

```
IDL> seed = 3L
IDL> x = RandomU (seed, 41)
IDL> y = RandomU (seed, 41)
IDL> z = Exp (- 4 * ((x – 0.5)^2 + (y – 0.5)^2))
```

查看随机分布的数据，键入：

```
IDL> Window, XSize=400, YSize=350
IDL> plot, x, y, pSym=4, SymSize=2.0
```

在这种情况下，需要有一套坐标轴来定义三维空间。对于建立三维变换矩阵来说，用带 Save 关键字的 Surface 命令将会比较好。Save 关键字将为 Surface 命令创造的三维变换矩阵保存在!P.T 系统变量里，而不是丢掉。可以用常用的轴旋转关键字来取得想要的三维空间。NoData 关键字只显示坐标轴。建立三维空间所必需的[XYZ]Range 关键字反映了正确的真实数据范围，而不是 Surface 命令中的伪值范围。键入：

```
IDL> Surface, Dist (10), /Save, /NoData, CharSize=1.5, $
        XRange=[0,1], YRange=[0,1], ZRange=[0,1]
```

上述命令建立了常规的三个轴。也可以增添附加轴。例如，需要突出 XY 平面。可以用 *Axis* 命令增添附加的 X 和 Y 轴：

```
IDL> Axis, YAxis=1, 1.0, 0.0, 0.0, /T3D, CharSize=1.5
IDL> Axis, Xaxis=1, 0.0, 1.0, 0.0, /T3D, CharSize=1.5
```

为了在这个三维空间内画点，每个三维点都必须和变换矩阵相乘。可以通过在图形输出命令中设置 T3D 关键字来实现此项功能。在这种情况下，可以用 PlotS 命令：

```
IDL> Plots, x, y, z, Psym=4, SymSize=2.0, /T3D
```

为了使图形的立体感更强，可以使用一条线将每个点连接到 XY 平面上。事实上，还可以根据 Z 值给线条赋色，从而提供给用户更多的信息。可以键入：

```
IDL> zcolors = BytScl (z, Top=99) + 1B
IDL> LoadCT, 22, Ncolors=100, Bottom=1
IDL> FOR j=0,40 DO Plots, [x[j], x[j]], [y[j], y[j]], $
        [z[j], 0], Color=zcolors[j], /T3D
```

输出结果如图 48 所示。

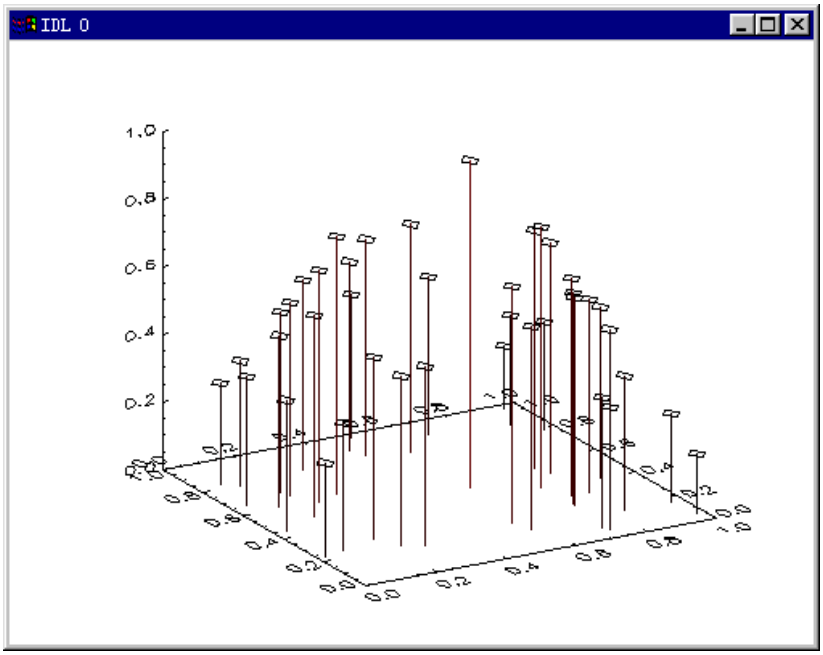


图 48　用 *Surface* 命令在三维空间里建立的散点图

可以用本书中的程序 *Colorbar* 来给图形添加颜色。键入：
本书中的程序 *Colorbar* 来给图形添加颜色。键入：

```
IDL> Colorbar, Position=[0.25, 0.9, 0.85, 0.95], $
        Range=[Min(z), Max(z)], Ncolors=100, Bottom=1, $
        Color=255, Title='Z Values'
```

从图形原点定位 3D 坐标轴

这是另一个建立三维坐标系的例子。在这个例子中，有一批跨越原点的数据，希望定义数据的轴线能通过原点。因为不想显示跟此数据相关的坐标轴，这时可用 Scale3 命令去建立三维空间。Scale3 命令与 Surface 命令使用相同的旋转矩阵，但前者在缺省情况下不画轴线，而且将变换矩阵装入!P.T 系统变量中。

为此例子可以生成一批数据，可用 LoadData 命令装载 41* 41 的 Elevation Data 数据集，如下所示：

```
IDL> data = LoadData (2)
IDL> data = data – (Max(data) / 2.0 )
IDL> x = FIndGen(41) – 20.0
IDL> y = FindGen(41)*2.0 – 41.0
```

用 Scale3 命令建立三维空间。为了使输出图形尽可能地大一些，可以关闭图形边缘。在此之前，将当前设置的系统变量保存，以便以后的恢复。键入：

```
IDL> Window, Xsize=500, Ysize=350
IDL> Save, /System_Variables, File='system.sav'
IDL> !X.Margin = 0 & !Y.Margin =0 & !Z.Margin = 0
IDL> Scale3, Xrange=[Min(x), Max(x)], Ax=45, Az=45, Yrange=[Min(y), Max(y)], $
        Zrange=[0, Max(data)]
```

现在可绘制数据的曲面图。务必用带有关键字[XYZ]Style 的 Surface 命令关闭坐标轴的显示，并强行要求 Surface 命令使用刚建立的 3D 变换矩阵，而不是 Surface 自己生成该矩阵。

键入：

IDL> Surface, data, x, y, /T3D, Xstyle=4, Ystyle=4, Zstyle=4

然后画过原点的坐标轴，键入：

IDL> TvLCT, 255, 255, 0, 1

IDL> Axis, 0, 0, 0, Color=1, /T3D, Charsize=2, Xaxis=1

IDL> Axis, 0, 0, 0, Color=1, /T3D, Charsize=2, Yaxis=1

IDL> Axis, 0, 0, 0, Color=1, /T3D, Charsize=2, Zaxis=1

输出图形如图 49 所示。

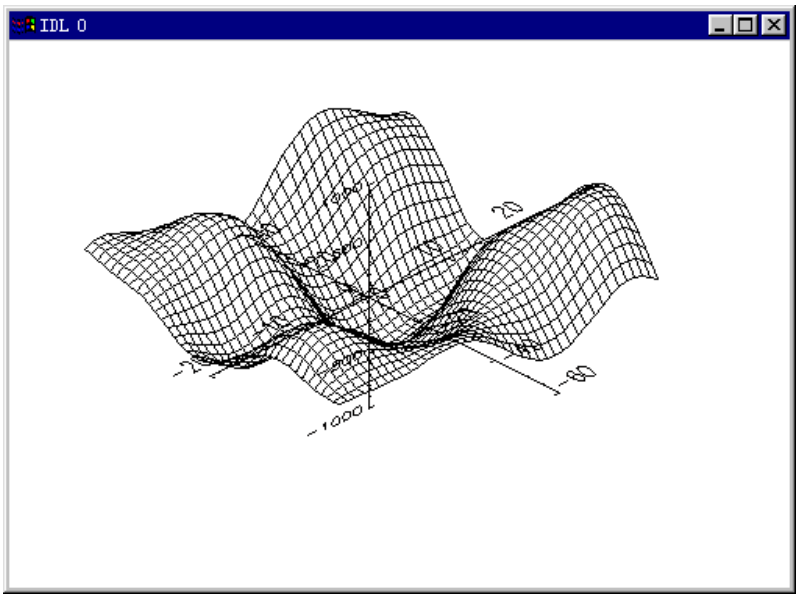


图 49　坐标轴穿过原点的曲面图。三维空间用 Scale3 命令创建

注意，在这幅图中，Y 轴没有延伸至图形的边缘。这是因图形的旋转而产生的错觉。能说服自己这真的是错觉吗？

在进入下一节之前，一定要将系统变量恢复为初始值。键入：

IDL>Restore,　'system.sav'

组合简单图形显示

利用所掌握的图形定位和创建三维坐标系的知识，可以很容易地用不同的方法组合图形显示。例如：您可以试着将同一数据的图像和等值线图形显示在一个窗口中，这通常是很有趣的。为说明做这是如何的容易，打开 512*512 的 Brain X-Ray 数据集，键入：

IDL> brain = LoadData(9)

用下载的 TVImage 命令可以让此数据显示在当前图形窗口 80%的范围内。但是记住，如果当前图形窗口不是正方形的，图像可能会产生变形。为了保证图像的横纵比例，在将图像缩放到大约 80%的图形窗口范围内时，设置关键字 Keep_Aspect_Ratio，如下：

IDL> Window, Xsize=450, Ysize=350

IDL> LoadCT, 3

IDL> thisPosition =[0.2, 0.2, 0.8, 0.8]

IDL> TVImage, brain, position=thisposition, Keep_Aspect_Ratio=1

当 TVImage 命令设置了关键字 Keep_Aspect_Ratio, 关键字 Position 就变成了一个输出参数。换句话说，变量 thisPosition 保存了图像在窗口中的规一化定位坐标。这些坐标不同

于输入的坐标，因为为保证图像横纵比例不发生变化，定位坐标不得不改变。通过打印 thisPosition 变量，能看到新的定位坐标。

IDL> print, thisPosition

可以用这些新的定位坐标，正好在已存在于显示窗口内的图像上画一幅等值线图。务必用关键字 NoErase，以防止等值线图抹去已存在的图形。关键字 XStyle 和 Ystyle 对避免轴线比例尺的自动调整是也必要的。键入：

IDL> Contour, brain, Xstyle=1, Ystyle=1, Nlevels=8, Position=thisPosition, /NoErase

输出结果如图 50 中的图片所示。

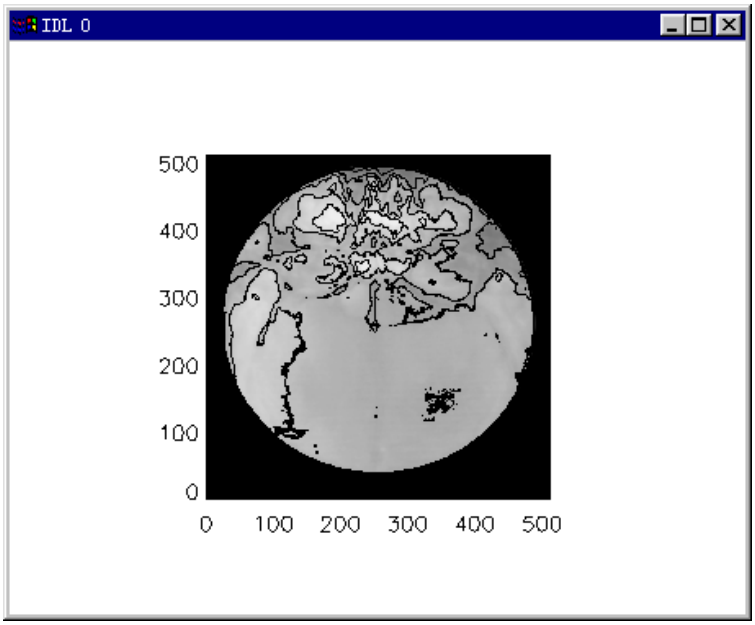


表 50　　在 IDL 中很容易将图像与等值线图进行组合

曲面图与等值线图的组合也是很简单的。用 LoadData 命令打开 41×41 Elevation Data 数组：

IDL> peak = LoadData(2)

也许希望在看到等值线图的同时，也能够看到由该数据产生的阴影曲面图。使用 Shade_Surface 命令，在 IDL 中可以非常轻松地创建三维坐标系，这个坐标系可用来定位等值线图。输入如下代码：

IDL> Window , Xsize=400, Ysize=400

IDL> TVLct, [70, 0], [70, 255], [70, 0], 0

IDL> LoadCT, 5, Ncolors=!D.Table_Size-2, Bottom=2

IDL> !P.Charsize=1.5

IDL> Shade_Surf, peak, /Save, Background=0, Color=1, Shades=BytScl(peak, Top=!D.Table_Size-2) + 2B

也许想在这个图形的右侧添加一个 Z 轴，键入：

IDL> Axis, Zaxis=0, 40, 0, 0, /T3D, Color=1

最后要做好添加等值线图的准备，务必使用关键字 NoErase，以防止将已经显示的内容抹去。关键字 Zvalue 在 Z 轴上定位等值线图。赋给关键字 Zvalue 的值是归一化坐标。Zvalue 的值为 1.0 时，将会把等值线图定位到已创建的三维坐标系空间的顶部。

IDL> Contour, peak, Nlevels=12, Color=1, Zvalue=1.0, /T3D, /NoErase, /Follow

注意，有时当图形位于三维坐标空间的边缘时，有些线条会被裁掉。这往往是将图形置于三维空间时，由计算时的舍入误差所导致。如果某些等值线图中的等值线看起来不完整，可以将关键字 NoClip 加到 contour 命令中，如下所示：

IDL> Contour, peak, NLevels=12, Color=1, Zvalue=1.0, /T3D, /NoErase, /Follow, /NoClip
输出结果如图 51 所示。

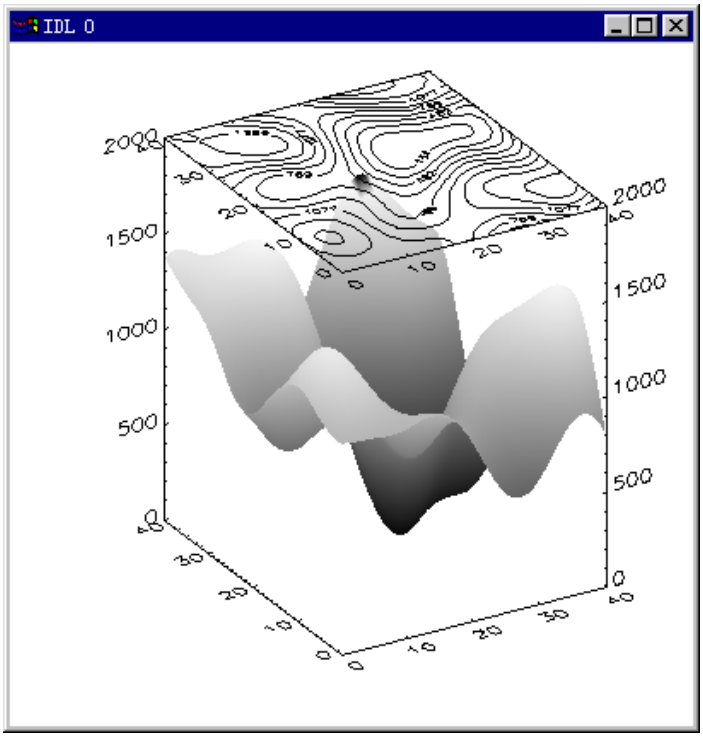


图 51 一个阴影曲面图与等值线图的组合

您还可以充分发挥您的想象力，组合使用其他的 IDL 图形命令，作出更多有趣的图形来。

IDL 中的动画图形

另一种强大的可视化图形技术就是数据动画。大多数情况下，在动画过程中可以得到用其他图像显示过程中难以或不可能获得的信息。倘若您还未打开一套 3D 数据集，可以用 LoadData 命令启动 80*100*57 的 MRI Head 数据集，如下所示：

IDL> head = LoadData(8)

在 IDL 中显示动画是用 XInterAnimate 命令来完成的。这个命令实际上调用了一个作为 IDL 主要动画工具的组件程序。XInterAnimate 必须调用三次：（1）第一次用来建立动画工具，特别是用来确定动画帧缓冲区的大小；（2）第二次用来装入动画工具；（3）最后一次用来播放动画序列。

在制作好 3D 动画数组后，设置和运行 XInterAnimate 就非常容易了。例如，可以在 Z 方向上动画显示 MRI Head 数据。那就是说，动画将从头像的底部顺着 Z 轴移动到头像的顶部。此时，在 Z 方向的每一帧图像都是一个 100*80 的数组。对于一个好的动画来说这有点小，但我们后面就会处理好这个问题。

建立动画工具

首先，建立动画工具。在此动画中将会有 57 帧，每帧的规格是 80*100 的图像。帧缓冲区应设置为 80* 100* 57。注意，使用关键字 Showload，就可以观察到动画的装入过程。当然，您也可以选择不显示这些过程。键入：

```
IDL> XInterAnimate, Set= [80, 100, 57], /Showload
```

装载动画缓冲区

下一步是将数据载装到动画缓冲区。这里，将已经打开的 3D 图像数组装入到动画工具缓冲区中。这个命令总是存在于循环中，如下所示：

```
IDL> FOR j=0, 56, DO XInterAnimate, Frame=j, Image=head[*,*,J]
```

当执行这个命令时，可以看到每帧被装入到动画缓冲区的过程。但动画过程还没有开始。

运行动画工具

最后一次键入 XInterAnimate 命令，动画就开始运行，如下所示：

```
IDL> XInterAnimate
```

动画工具应该看起来如图 52 所示。

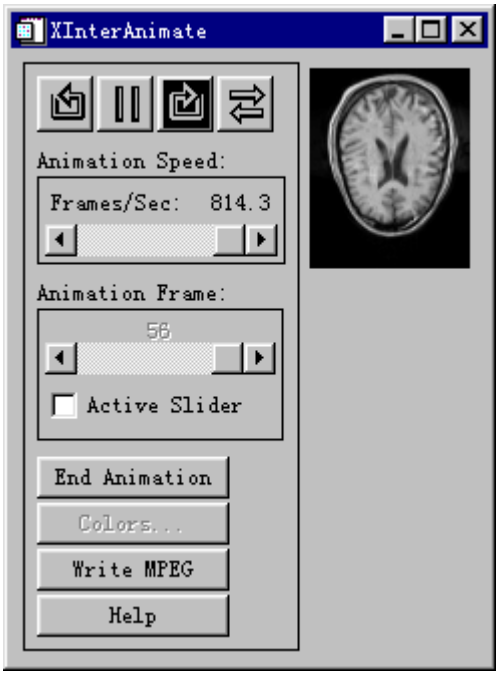


图 52　已装入 MRI Head 数据的 XInterAnimate 动画工具

动画的控制

可用某些动画控制来做实验。可以决定动画帧如何循环、动画的速度，甚至可在动画运行时改变颜色表。注意，倘若按了 End Animation 按钮，那就得再一次输入所有的三个动画命令以使动画得以运行。在使用滑杆指定动画帧之前，必须让动画停下来。

通常，动画以机器所允许的最快速度运行。（一个组件定时器事件负责取得序列中的下一个动画帧）。动画速度按钮使此定时器事件增加延迟。如果希望动画一开始就以低速运行，可以给最后的 XInterAnimate 命令赋一个速度参数。这个值可以在 0 到 100 之间。例如，要使动画以中等速度运行，可以输入：

```
IDL> XInterAnimate, Set=(80,100,57), /Showload
IDL> FOR j=0, 56    DO    XInterAnimate, Frame=j, image=head[*,*,j]
IDL> XInterAnimate, 50
```

存储动画的像素映射图

您也许发现，动画工具运行得非常快，原因之一是它使用了像素映射图和设备拷贝技术去实现动画。（关于这种技术的详细技术参考 120 页的“删除注释的设备拷贝法”。）通常当点击 End Animation 按钮时，这些像素映射图就被删掉了。倘若将这些像素图保存在内存中，只须键入 XInterAnimate, 就可以在任何时候立即启动一个新的动画。将像素映射图保存到内存中的方法是当启动动画时，使用关键字 Keep_Pixmaps，如下所示：

```
IDL> XInterAnimate, Set=(80,100,57), /Showload
IDL> FOR    j=0, 56    DO    XInterAnimate, Frame=j, image=head[*, *, j]
IDL> XInterAnimate, 50, /Keep_Pixmaps
```

可以退出动画程序，然后再一次启动，输入：

```
IDL>XInterAnimate
```

当完成任务后，确保已将像素映射图删除，输入关键字 Close 即可完成，如下所示：

```
DL>XInterAnimate , /close
```

其他类型图形数据的动画

三维数据并不是惟一一种能够在动画工具中制作成动画的数据。事实上，多数情况下，想制作动画的内容就是显示在 IDL 图形窗口中的内容。XInterAnimate 工具能够快速拍下 IDL 的图形窗口，然后将它们作为动画帧存储到动画缓冲区中。完成此项任务可用关键字 Window, 而不是 Image。

下面可以看到这一步是怎样通过已经打开的数据来完成。显示的 80*100 图像非常小，也许想使它变大一点，但又不想再创建一个更大的数组来存储数据。这样做会占用更多的 IDL 内存。可以在每个图像被显示时，用重置的方法使它们变大。这不会占用额外的 IDL 内存。

假设要求每个动画帧的尺寸是 240*300。创建如下的动画代码，严格按下面代码输入。倘若在 FOR 循环中出现输入错误，请从循环的起点再次输入：

```
IDL> XInterAnimate, Set=[240, 300, 57], /Showload
IDL> FOR j=0,56 DO BEGIN $
     TV, Rebin(head[*, *], 240, 300) $
     & XInterAnimate, Frame=j ,Window=!D.Window & ENDFOR
IDL> XInterAnimate, 50
```

可以在此循环中输入任何 IDL 图形命令。这给了读者相当大的灵活性，让读者能够对各种各样的数据进行动画处理。

数据网格化及显示

许多 IDL 图形显示程序（如：Surface、Contour、Shade_Surf 等）要求数据按 2D 网格数组排列。（通常，数据不一定要按规则网格排列。）但是，我们有时也会没有这种符合要求的网格数据，而只有一些离散点数据。例如，“点位数据”就是来自于随机定位的采集点。此类数据必须在显示之前进行网格化处理。

装载此类随机分布的 XYZ 数据来作练习，可用 LoadData 命令安装 Randomly Distributed (XYZ)数据集即可。此数据集包含了 3 个含 41 个元素的矢量，这些矢量代表站点的纬度和

经度坐标以及采样值。输入：

```
IDL> data = LoadData(14)
IDL> lon = data[0, *]
IDL> lat = data[1,*]
IDL> value= data[2,*]
```

可以在一个网格上画出纬度和经度矢量，就能看出它们实际上是随机分布的。输入：

```
IDL> LoadCT, 0
IDL> TvLCT, [70, 255, 0], [70, 255, 255], [70, 0, 0], 1
IDL> Window, Xsize=400, Ysize=400
IDL> plot, lon, lat, /YnoZero, /NoData, Background=1
IDL> plots, lon, lat, PSym=5, Color=2, SymSize=1.5
```

输出结果如图 53 所示。

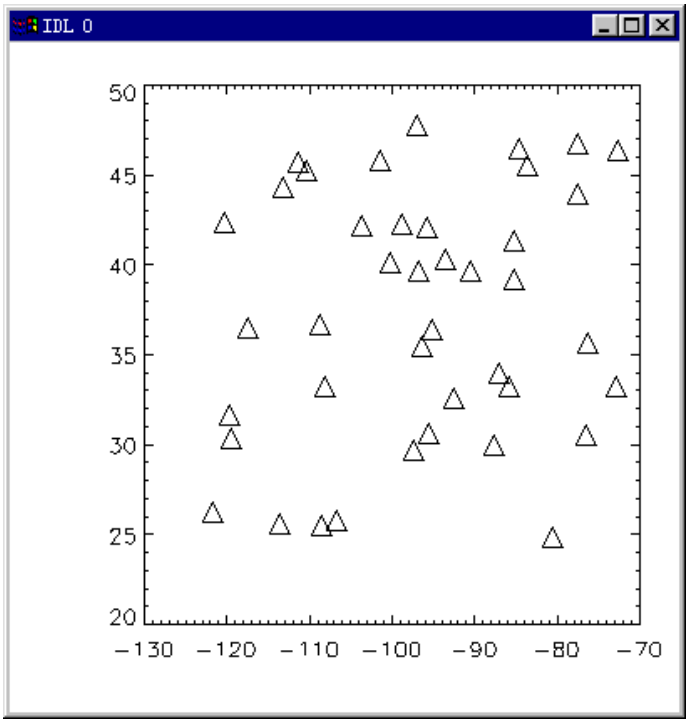


图 53　随机分布的纬度和经度坐标

德洛内三角形法网格化

IDL 所用的数据网格化的方法叫做德洛内（Delaunay）三角法。这种算法并非是对数据进行网格化的最好方法。但是作为一种常用的网格化方法，它具有速度快，相对简单和被人们广泛使用的优点。它需要从 X 和 Y 坐标上建立一组德洛内三角形，即任意一个三角形的边界范围内都不含其他三角形的顶点。德洛内法就是用这些三角形顶点上的值插出规则网格点的值。

Triangulate 命令用来建立德洛内三角形组。此命令的输入参数是离散点的 XY 坐标。此命令形式如下，其中 angles 是输出变量，保存了计算返回的德洛内三角形组。

```
IDL> Triangulate, lon, lat, angles
```

注意，也可以用 Triangulate 命令返回这些点的凸形外络线。只须在上述命令中增加第 4 个变量参数，这组点的凸形外络点就会被返回到此参数中。凸形外络线在许多算法操作中都是很有用的。

使这组三角形可视化（此数组中有 70 个三角形），输入命令：

```
IDL> FOR j=0,69 DO BEGIN t= [angles[*, j], angles[0, j]] $
    & plots, lon[t], lat[t], Color=3 & ENDFOR
```

输出结果如 54 所示。

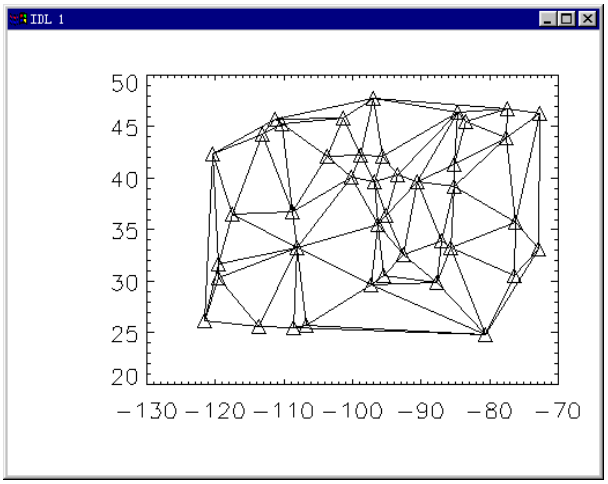


图 54　用 Triangulate 命令返回的德洛内三角形组

为了网格化数据，只需将从 Triangulate 返回的三角形组数据传递到 TriGrid 程序即可。可以设置网格的端点或边界，以及网格间隔。还可以用关键字 Missing 来为位于三角形外部的数据指定值。此外，可用输出关键字 XGrid 和 YGrid 中获得数据网格化后的新纬度和经度矢量。例如，输入：

```
IDL> latMax = 50.0
IDL> latMin = 20.0
IDL> lonMax = -70.0
IDL> lonMin = -130.
IDL> mapBounds = [lonMin, latMin, lonMax, latMax]
IDL> mapSpacing = [ 0.5, 0.25 ]
IDL> gridData = Trigrid( lon, lat, value, angles, mapSpacing, mapBounds, $
    Missing=Min(value), XGrid=gridlon, YGrid=gridlat)
```

现在可以在需要的 IDL 命令中使用网格数据了。

```
IDL> Contour, gridData, gridlon, gridlat, / Follow, $
NLevels=10, XStyle=1, YStyle=1, Background=1, color=2
```

输出结果如图 55 所示。

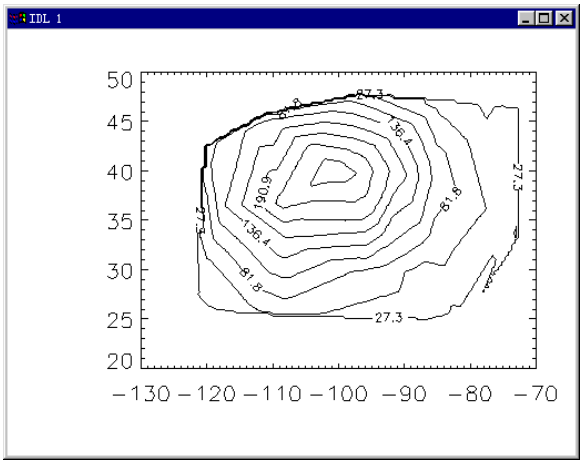


图 55　用 TriGrid 命令生成的结果显示的等值线图

注意，可用关键字 Quintic 或 Smooth 来光滑曲面（两者作用类似）。键入：

```
IDL> gridData = Trigrid( lon, lat, value, angles, mapSpacing, mapBounds, $
   Missing=Min(value), XGrid=gridlon, YGrid=gridlat, /Quintic)
IDL> Contour, gridData, gridlon, gridlat, / Follow, NLevels=10, XStyle=1, $
   YStyle=1, Background=1, color=2
```

有时用关键字 Extrapolate 对三角形边缘外部的值进行推断，可得到更好的结果。键入：

```
IDL> Triangulate, lon, lat, angles, hull
IDL> gridData = Trigrid( lon, lat, value, angles, mapSpacing, mapBounds, $
   Missing=Min(value), Extrapolate=hull)
IDL> Contour, gridData, gridlon, gridlat, / Follow, NLevels=10, $
   XStyle=1, YStyle=1, Background=1, color=2
```

输出结果类似于图 56 所示。

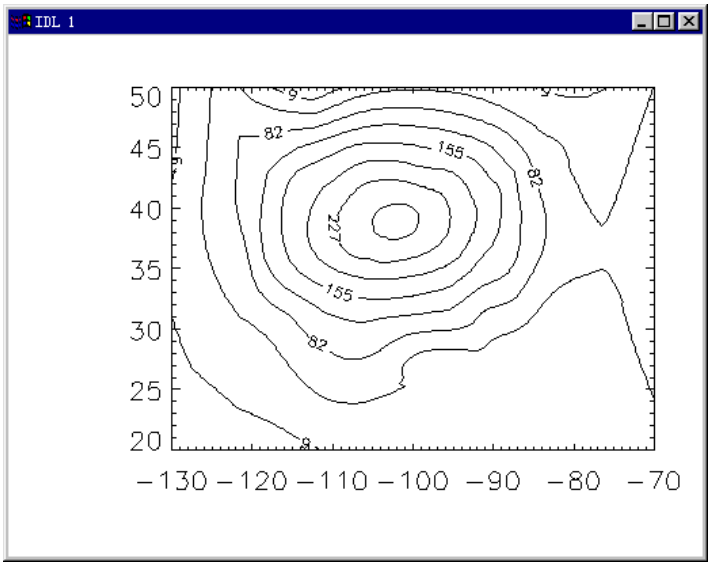


图 56 使用关键字 Extrapolation 和 Smooth 后的 TriGrid 结果图

数据的球形网格化

如果这是真实的纬度和经度数据，您不会愿意在平面上对它进行网格化。地球的表面更象球形。网格化程序 Triangulate 和 TriGrid 同样允许对数据进行球形网格化。这样，生成出来的三角形就成了球形三角形。

将此数据放到地图投影上，可输入：

```
IDL> Map_Set, /Orthographic, /Grid, /Continents, /Label, /Isotropic, 35, -100, Color=1
IDL> Plots, lon , lat, PSym=4, Color=2
```

尽管这一次在命令中使用了不同的关键字，但在球形网格化中，前面的许多相同参数仍可用来对数据进行网格化。确保设置了关键字 Degrees，否则这些命令将用弧度来计算球形三角形。命令如下所示：

```
IDL> Triangulate, lon, lat, Sphere=angles, FValue=value, /Degrees
IDL> gridData = Trigrid(value,Sphere= angles, mapSpacing, mapBounds, $
   Missing=Min(value), /Extrapolate, /Degrees)
IDL> Map_Set, /Orthographic, /Grid, /Continents, /Label, /Isotropic, 35, -100, Color=1
IDL> Contour, gridData, gridlon, gridlat, / Follow, NLevels=10, XStyle=1, YStyle=1,$
```

Color=2, /Overplot

输出结果应与图 57 中的图片相似。

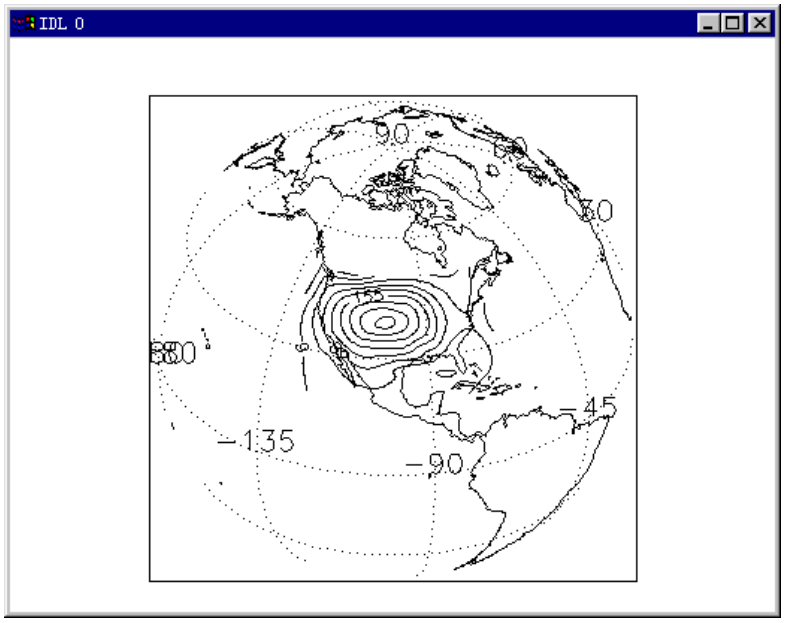


图 57　在地图投影顶部以等值线图形显示的球形栅格数据

第五章　图形显示技巧

本章概要

在上一章节学习了一些图形显示技术。在这一章节将学习几个新的图形显示技巧，以便让图形显示具有专业的视觉效果。

具体来说，读者通过本章将学会：

1. 怎样让鼠标交互作用于图形显示
2. 怎样从图形显示中删除注释
3. 怎样在图形显示上画“橡皮条”
4. 怎样在图形显示技巧中使用 Z 图形缓冲区

将光标用于图形显示

数据可视化显示后的另一需求是用户可用不同的方式对数据进行交互式的操作。用户也许需要用光标去选择或者标注部分数据。这种交互作用在 IDL 中用 Cursor 命令很容易完成。

用 LoadData 命令装入 Time Series 数据集，可看到 Cursor 命令是如何工作的。

```
IDL> curve = LoadData (1)
```

输入下述命令，显示曲线：

```
IDL> Window, Xsize = 400, Ysize = 400
IDL> LoadCT, 0
IDL> TvLCT, 255,255,0,1
IDL> Plot, curve
```

Cursor 命令接受两个参数。这些参数记录的是鼠标键按下时光标位置的变量。Cursor 命令要求光标位于当前图形窗口中。（即被!D.Window 系统变量指向的窗口。）例如，如果输入这个命令，IDL 将会等待光标被移动到当前图形窗口（如果输入的是上述命令，就是 0 号索引窗口）并单击鼠标键。当执行上述动作后，IDL 将光标位置返回到变量 xLocation 和 yLocation 中。输入：

```
IDL> Cursor, xLocation, yLocation
```

如果打印出这些变量的值，将发现这些值被赋予的是数据坐标空间。xLocation 的数值从 0 到 100，yLocation 的数值从 0 到 30。（如果是在图形边界内点击的鼠标，它们至少是这么多。如果不是在图形边界内点击的鼠标会怎么样？）缺省时，Cursor 命令返回数据坐标位置。

```
IDL> Print, xLocation, yLocation
```

什么时候返回的光标位置？

从上面的命令看，似乎鼠标键被按下时返回光标位置，但并非总是这样。事实上，Cursor 命令什么时候报告光标的位置是由 Cursor 命令的关键字所决定的。这些关键字是：

Change 当光标位置发生改变或用户移动光标时，返回光标位置。

Down 当鼠标键被按下时，返回光标位置。

NoWait 当 Cursor 命令执行时，光标位置被立即返回。没有任何延迟或等待鼠标的按键。这个关键字有时用于当对象正在显示窗口中被移动时的循环中。

UP 不是在鼠标键被按下时，而是放开或释放后返回光标位置。

Wait Cursor 命令等待鼠标键被按下后返回光标的位置。只要鼠标键被按下，此关键字对 Cursor 命令的作用就类似于用 NoWait 关键字调用 Cursor 命令。此关键字是 Cursor 命令的缺省状态。

在 Cursor 命令中，小心使用合适的关键字，特别是在循环过程中使用 Cursor 命令。用户有时习惯地认为 Cursor 命令的缺省属性是只有鼠标键被按下时才返回光标的位置。其实不然，缺省属性只是等待一个单击动作，以后的行为就和 NoWait 关键字一样。在循环中这个区别是至关重要的。

哪一个鼠标键和光标共同作用呢？

除了设置光标属性外，有时还想知道哪个鼠标键用于对 Cursor 命令作出反应。例如，想要用鼠标左键做某件事，而做另外不同的事情要用鼠标右键 Cursor 命令作出的反应。可

以检查系统变量!Mouse 中的 Button 字段，来判断哪一个鼠标键在和 Cursor 命令共同作用。（老版本的 IDL 是用系统变量!Err 的值来判断的。）这个字段是一个整型位映象。Button 这个字段的有效值及其意义如下：

!Mouse.Button = 0　当前没有按键被使用

!Mouse.Button = 1　左键用于 Cursor 命令

!Mouse.Button = 2　中间键用于 Cursor 命令

!Mouse.Button = 4　右键用于 Cursor 命令

用光标标注图形输出

使用 Cursor 命令的一种方法是允许用户交互地在线画图上放置符号标记。例如，正确无误地输入下列命令。当输完最后一个回车键后，在当前的图形窗口上单击鼠标五次。五个符号将放置在窗口中。（如果在输入下列代码时出现打字错误，必须从头开始重新输入。）输入：

```
IDL> For j = 0, 4 DO BEGIN $
IDL> Cursor, xloc, yloc, /DOWN & $
IDL> Plots, xloc, yloc, Psym = 4, SymSize = 2, Color = 1 & ENDFOR
```

画方框

有时可能为了选取图形显示中的某部分，而在它的周围画上方框。这里有些命令可用来选择由 Cursor 命令产生的方框的对角，画出该方框，并将图形缩放到该方框坐标范围。首先画图：

```
IDL> Plot, curve
```

接着，使用光标选择想画的方框的一角。要确保在当前图形窗口上点击光标。为确定哪个是当前窗口，并让它不被隐藏，可输入：

```
IDL> WShow
```

现在键入第一个 Cursor 命令。在图形轴的范围内某处点击：

```
IDL> Cursor, x1,y1, /DOWN; Select one corner of box.
```

接着输入第二个 Cursor 命令。在图形轴的范围内某处点击：

```
IDL> Cursor, x2,y2, /DOWN; Select diagonal corner of box.
```

上述 Cursor 命令返回的坐标是数据空间坐标。按如下画方框：

```
IDL> Plots, [x1,x1,x2,x2,x1],[y1,y2,y2,y1,y1], color = 1
```

输出结果应类似于图 58 中所示，尽管实际的图形上方框取决于在窗口中点击的位置。

为了放大这部分图形，必须保证方框坐标的正确顺序。这是非常必要的，因为可能先选择的是方框的右下角，然后是左上角，这样 x1 将大于 x2。还可以想象其他的假设。为了适应所有的情况，键入：

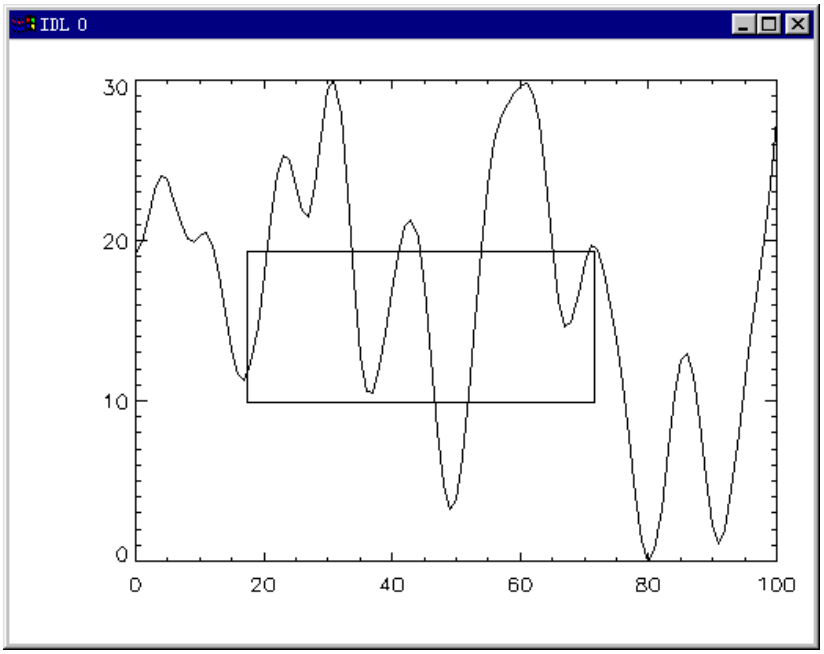


图 58　在部分数据周围画上方框的线画图。用 Cursor 命令选择方框的坐标，用 PlotS 命令画方框。

```
IDL> Xmin = Min([x1,x2], Max = xmax)
IDL> Ymin = Min([y1,y2], Max = ymax)
```

最后，已经为放大对方框内的数据做好了准备。除了正确地设置数据范围外，还必须设置<XY>Style 关键字。知道为什么吗？如果不知道，可在不使用这两个关键字的情况下试试下面的命令。将会发生什么呢？

```
IDL> Plot, curve, XRange = [xmin, xmax], Yrange = [ymin, ymax], $
    Xstyle = 1, Ystyle = 1
```

在图像上使用 Cursor 命令

通常当在处理图像数据时使用 Cursor 命令，希望用设备坐标而不是数据坐标返回光标位置。这是因为设备坐标和图像中对应的位置之间通常存在一种简单的关系（大多数是一对一的关系）。为了解如何工作的，可用 LoadData 命令打开 360*360 的 World Elevation 数据集，键入：

```
IDL> image = LoadData(7)
```

显示图像，并装入某些颜色。如下：

```
IDL> topColor = !D.Table_Size-1
IDL> LoadCT, 3, Ncolors = !D.Table_Size-1
IDL>TvLCT, 255,255,0, TopColor
IDL> Window, XSize = 360, YSize = 360
IDL> TV, BytScl (image, Top = !D.Table_Size-2)
```

利用光标在图像中选择某一特定行和列。注意 Cursor 和 PlotS 命令中的 Device 关键字。

这是确保返回的坐标是设备坐标而不是数据坐标。在该位置上画一个十字线。（确保在输入 Cursor 命令后，在图像窗口中点击一下。）键入：

```
IDL> S = Size(image)
IDL> Cursor, col, row, / Device; Click in the window!
IDL> Plots, [col, col], [0,s (2) ], / Device, Color = topColor
IDL> Plots, [0,s (1) ], [row, row ], / Device, Color = topColor
```

注意，在图像中某一特定的行和列上获得图像数据是多么的容易。例如，可以轻易地绘制出图像中行和列的数据剖面，键入：

```
IDL> Window, 1, Xsize = 500, Ysize = 300
IDL> !P.Multi = [0, 2, 1]
IDL> Plot, image [*, row], Title =’ Row Profile’
IDL> Plot, image [col, *], Title =’ Column Profile’
IDL> !P.Multi = 0
IDL> Wset, 0
```

输出结果类似于图 59 所示。

在循环中使用 Cursor 命令

有时想在循环中使用 Cursor 命令。例如，当用光标选择图像上的单个像素时，可能想知道它的像素值。下面是个简单的循环程序，它将一直执行下去，直到单击右键或中键退出。打开文本编辑器，准确无误地输入如下代码。

```
TopColor = !D.Table_Size-1
LoadCT, 3, Ncolors = !D.Table_Size-1
TvLCT, 255, 255, 0, topColor
TV, BytScl (image, Top =!D.Table_Size-2)
!Mouse.Button = 1
REPEAT BEGIN
   Cursor, col, row, /Down, /Device
   Print, ‘Pixel Value:’, image[col, row]
ENDREP UNTIL !Mouse.Button NE 1
END
```

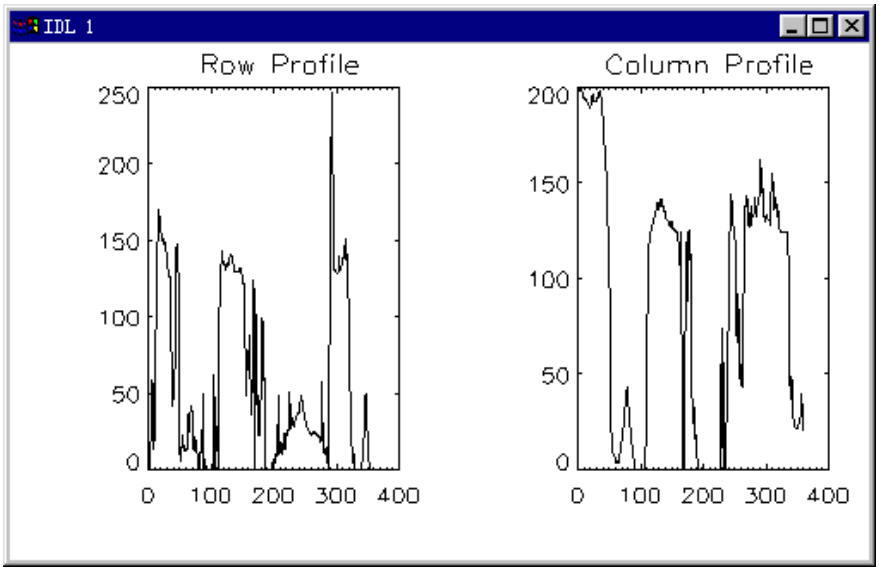


图 59　　用 Cursor 命令在图像中选择的行和列的剖面

保存到 loop1.pro 文件中（此文件已经存在于下载的本书配套程序之中）。编译，并运行这个小主程序，输入：

IDL> .RUN loop1

移动光标进入图像窗口，开始单击左键。图像像素值就会出现在日志窗口中，直到点击其他键，而不是左键。

如果在 Cursor 命令中使用除了 Down 之外的其他关键字会发生什么呢？实验一下，找出答案。

从显示中删除注释

当使用光标在图形显示上按照刚才使用的方法来添加注释时，也许会问：“怎样才能删除刚才放置在那儿的注释呢？”有两种较好的方法删除注释。称之为异或法和设备拷贝法。笔者认为在两者之中设备拷贝法更能给出专业的感观效果。两种方法都列举出来，但重点将集中在设备拷贝法上。

删除注释的异或法

异或法是在图形函数的基础上起作用的。图形函数是两个数的位操作。这两个数分别与已经显示出来的像素（称作所谓的目标像素）以及希望放置在同一位置的像素（称作所谓的源像素）相关联。

通常，IDL 使用的图形函数称作源。在这种图形函数里，IDL 忽略了目标像素的值，仅仅在该像素位置上放置源像素的值。但如果这种图形函数变成 XOR（异或法），IDL 将目标像素和源像素进行逐位比较。这会产生反向目标像素的效果。换句话说，如果像素的二进制表示为 01100101，那么执行 XOR 命令后，像素的二进制表示为 10011010。

（实际的 XOR 过程远比这复杂，因为只有 IDL 在颜色索引表中的邻近位置上有 256 种颜色时，它才按这种方式运行，而这是一种少见的情形。大多数人只是认为 XOR 法是用“相反的”颜色画，并保留原来的。在实际的 XOR 法中，可能预测将会使用哪种颜色来画，但在大多数情形下并不是这样。这就是为什么绝大多数 IDL 专业程序员宁愿采用设备拷贝法。）

在任何时侯，图形函数的作用效果都是由 Device 命令和 Set_Graphics_Function 关键字

设置的。源模式下图形函数为 3。XOR 模式下图形函数为 6。此时 IDL 处于系统缺省的源模式下。当处于这种模式时，重新显示图像窗口中的图像。输入：

```
IDL> TV, BytScl (image, Top =!D.Table_Size-2)
```

现在，选择 XOR 模式：

```
IDL> Device, Set_Graphics_Function = 6
```

在图像中画一个方框：

```
IDL> Plots, [0.2, 0.2, 0.8, 0.8, 0.2], Color = topColor,$
    [0.2, 0.8, 0.8, 0.2, 0.2], /Normal
```

注意，方框线的颜色不是预测的黄色。取而代之是多彩的，尽管它显得也很合理。在这个模式下线下的像素已经被翻转了。

要删除方框，只须将底下的像素值翻转回它们的原值。再次用 PlotS 命令很容易完成。

```
IDL> Plots, [0.2, 0.2, 0.8, 0.8, 0.2], Color = topColor,$
    [0.2, 0.8, 0.8, 0.2, 0.2], /Normal
```

可以反复执行上面的命令，让方框随心所欲地出现或消失。在继续下面的练习前，确保将图形函数设回到源模式。键入：

```
IDL> Device, Set_Graphics_Function = 3
```

在 IDL 程序中可容易地利用图形函数。例如，打开以前写的 loop1.pro 主程序，按如下修改它。这个程序将在图像窗口中每次点击的位置上画一个大的十字线。以 loop2.pro 命名保存程序。键入：

```
topColor = !D.Table_Size-1
LoadCT, 3, NColors=!D.Table_Size-1
TvLCT, 255, 255, 0, topColor
TV, BytScl(image, Top=!D.Table_Size-2)
!Mouse.Button = 1
; Go into XOR mode.
Device, Set_Graphics_Function=6
; Get initial cursor location. Draw cross-hair.
Cursor, col, row, /Device, /Down
PlotS, [col,col], [0,360], /Device, Color=topColor
PlotS, [0,360], [row,row], /Device, Color=topColor
Print, 'Pixel Value: ', image[col, row]
; Loop.
REPEAT BEGIN
; Get new cursor location.
Cursor, colnew, rownew, /Down, /Device
   ; Erase old cross-hair.

PlotS, [col,col], [0,360], /Device, Color=topColor
PlotS, [0,360], [row,row], /Device, Color=topColor
Print, 'Pixel Value: ', image(colnew, rownew)
; Draw new cross-hair.
PlotS, [colnew,colnew], [0,360], /Device, Color=topColor
PlotS, [0,360], [rownew,rownew], /Device, Color=topColor
   ; Update coordinates.
col = colnew
row = rownew
```

```
ENDREP UNTIL !Mouse.Button NE 1
;Erase the final cross-hair.
PlotS, [col,col], [0,360], /Device, Color=topColor
PlotS, [0,360], [row,row], /Device, Color=topColor
; Restore normal graphics function.
Device, Set_Graphics_Function=3
END
```

保存到文件 loop2.pro 中。（此文件已经存在于下载的本书配套程序之中。）编译，并执行这个主程序，键入：

```
IDL> .RUN loop2
```

将光标放在图像窗口中，点击左键数次。在每个光标位置上应该有一个十字线。按右键或中间键终止程序。

删除注释的设备拷贝法

设备拷贝法利用像素映射窗口来删除已显示屏幕上的注释。像素映射窗口和其他 IDL 图形窗口完全一样，除了它不显示在显示器上外。事实上，它存在于显示设备的视频随机存储器中。换句话说，它存在于存储器中。从其他任何方面来说，它就象一个正常的 IDL 图形窗口一样：可用 Window 命令创建，用 WSet 命令激活，用 WDelete 命令删除，等等。在像素映射窗口内画图和在 IDL 正常的图形窗口内画图是完全一样的（例如，用 Plot、Surface、TV 和其他图形输出命令）。

设备拷贝技术就是从一个窗口（称为源窗口）拷贝一个矩形区，然后将矩形区传入另一个窗口（称为目的窗口）。源窗口和目的窗口有时可以是同一个窗口，这点等会会看到。图 60 是设备拷贝技术的图解。

实际的拷贝是通过 Device 命令和 Copy 关键字（设备拷贝技术的名字由此而来）完成。命令的一般形式如下：

```
Device, Copy = [sx, sy, col, row, dx, dy, sourceWindowID]
```

在这个命令中，Copy 关键字的组成元素是：

sx, sy 源窗口矩形区左下角的设备坐标（矩形区是从源窗口拷贝的）

col 从源窗口中拷贝的列数。这是矩形区的宽度。

row 从源窗口中拷贝的行数。这是矩形区的高度。

dx, dy 目标窗口中矩形区左下角的设备坐标。（目标窗口是要将矩形区拷贝到的窗口。目标窗口总是当前图形窗口。）

sourceWindowID 源窗口的窗口索引号。矩形区从此窗口拷贝到当前图形窗口（由!D.Window 系统变量指定）。源窗口可以是当前图形窗口，但多数情况下它是一个非当前图形窗口的窗口。它通常是一个像素映射窗口。

要看其是如何工作的，先创建一个像素映射窗口，在里面显示一幅图像。用带 Pixmap 关键字的 Window 命令来创建像素映射窗口，键入：

```
IDL> Window, 1, /Pixmap, XSize = 360, YSize = 360
IDL> TV, BytScl (image, Top =!D.Table_Size-2)
```

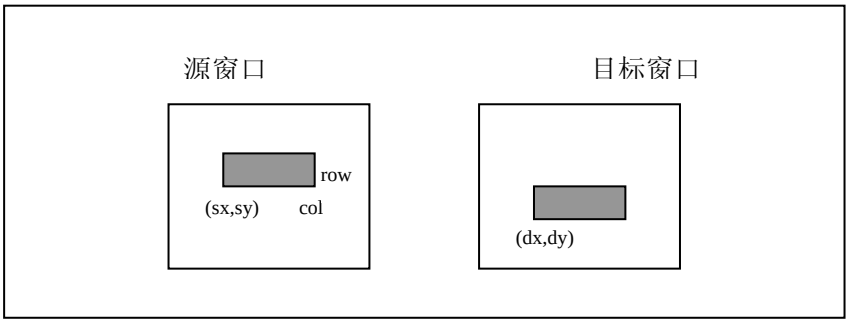


图 60　设备拷贝技术是将源窗口的一个矩形区拷贝到目标窗口的某个位置。实际上可以拷贝整个窗口，而且源窗口和目标窗口可以是同一个窗口。

注意，当输入这些命令时没有任何可见的线索表明发生了什么。这是因为像素映射窗口仅存在于视频随机存储器中，而不是在显示屏上。为证明这个窗口内存有的内容，可打开第三个窗口，并将像素映射窗口的内容拷贝到第三个窗口中。如果第三个窗口看起来象原来的图像窗口，那么已经输入的命令是正确的。键入：

```
IDL> Window, 2, Xsize = 360, Ysize = 360
IDL> Device, Copy =[0, 0, 360, 360, 0, 0, 1]
```

注意，已将像素映射窗口的全部内容拷贝到这个新的窗口中。这种操作除了在速度上快几个数量级之外，其他方面与在此新窗口中重新显示图像类似了。将像素映射窗口中的全部内容拷贝到了新的显示窗口是一种常见的做法，即使只是修改部分的显示窗口。

删除最后创建的两个窗口（包括像素映射窗口），如下：

```
IDL> Wdelete, 1, 2
```

当对像素映射窗口操作完成后，删除它们是很重要的。它们会占用存储空间，这些存储空间可用于其他用途。某些窗口管理器为这些像素映射窗口分配一个定量的存储空间。如果像素映射窗口超过了视频存储器的容量，可以使用虚拟内存。X 终端给像素映射窗口留有非常小的存储空间。

要了解设备拷贝技术在实际中是如何运用的，修改主程序以前写的主程序文件 loop2.pro。可以拷贝到另一个文件，以 loop3.pro 命名。修改代码如下：

```
TopColor = !D.Table_Size-1
LoadCT, 3, Ncolors = !D.Table_Size-1
TvLCT, 255, 255, 0, topColor
TV, BytScl (image, Top =!D.Table_Size-2)
!Mouse.Button = 1
   ; Create a pixmap window and display image in it.
Window, 1, /Pixmap, Xsize = 360, Ysize = 360
TV, BytScl (image, Top =!D.Table_Size-2)
   ; Make the display window the current graphics window.
Wset, 0

; Get initial cursor location. Draw cross-hair.
Cursor, col, row, /Device, /Down
Plots, [col, col], [0, 360 ], / Device, Color = topColor
Plots, [0, 360 ], [row, row ], / Device, Color = topColor
Print, 'Pixel Value:', image[col, row]
```

```
; Loop.
REPEAT BEGIN

; Get new cursor location.
   Cursor, colnew, rownew, /Device, /Down
; Erase old cross-hair
Device, Copy = [0, 0, 360, 360, 0, 0, 1]
Print, 'Pixel Value:', image[colnew, rownew]

; Draw new cross-hair.
Plots, [colnew, colnew], [0, 360 ], / Device, Color = topColor
Plots, [0, 360 ], [rownew, rownew ], / Device, Color = topColor
ENDREP UNTIL !Mouse.Button NE 1
   ; Erase the final cross-hair.
Device, Copy = [0, 0, 360, 360, 0, 0, 1]
END
```

以 loop3.pro 命名存档。（此文件已经存在于下载的本书配套程序之中）编辑，并运行主程序，输入：

```
IDL> .RUN loop3
```

放置光标于图像窗口中，点击左键数次。单击右或中间键终止程序。注意十字线是用黄色绘制。

在继续下一次练习之前，删除像素映射窗口。输入：

```
IDL> Wdelete, 1
```

画一个橡皮筋方框

设备拷贝技术非常适用于在屏幕上画橡皮条选择方框和其他形状。（橡皮条方框是指一角固定，另一角随着光标的变化而变化的方框）。事实上，修改程序 loop3.pro 可以很容易实现。将 loop3.pro 拷贝到文件 rubberbox.pro。（loop3.pro 文件已经存在于下载的本书配套程序之中。）作如下修改，查看创建一个橡皮条方框是多么容易。

```
 topColor = !D.Table_Size-1
LoadCT, 3, NColors=!D.Table_Size-1
TvLCT, 255, 255, 0, topColor
TV, BytScl(image, Top=!D.Table_Size-2)
!Mouse.Button = 1
; Create a pixmap window and display image in it.
Window, 1, /Pixmap, XSize=360, YSize=360
TV, BytScl(image, Top=!D.Table_Size-2)
;Make the display window the current graphics window.
WSet, 0
; Get initial cursor location (static corner of box).

Cursor, sx, sy, /Device, /Down
; Loop.
REPEAT BEGIN
   ; Get new cursor location (dynamic corner of box).
```

```
    Cursor, dx, dy, /Wait, /Device
  ; Erase the old box.
    Device, Copy=[0, 0, 360, 360, 0, 0, 1]
  ; Draw the new box.
    PlotS, [sx,sx,dx,dx,sx], [sy,dy,dy,sy,sy], /Device, $
  Color=topColor
ENDREP UNTIL !Mouse.Button NE 1
;Erase the final box.
Device, Copy=[0, 0, 360, 360, 0, 0, 1]
END
```

运行程序，输入：

```
IDL> .RUN rubberbox
```

在继续下一次练习之前，删除像素映射窗口。输入：

```
IDL> Wdelete,1
```

图形窗口的滚动

设备拷贝技术的另一个有效应用是实现图形窗口的滚动。在这个示例中将运用设备拷贝技术让图形显示窗口中的图像滚动起来。图像从左至右每次滚动四列。使用的算法如下：（1）将窗口右边最后四列拷贝到一个仅有四列宽和 360 行高的小像素映射窗口。（2）将整个显示窗口的内容（减去已经拷贝的四列）在同一窗口（也就是说，源窗口和目标窗口是同一个窗口）内向右边移动四列。（3）将像素映射窗口的内容拷贝到显示窗口左边头四列。打开文本编辑器，输入下列命令。以 scroll.pro 命名（此文件已经存在于下载的本书配套程序之中）。

```
    ; Open a pixmap window 4 columns wide.
  Window, 1, /Pixmap, XSize=4, YSize=360
  FOR j=0,360/4 DO BEGIN
    ; Copy four columns on right of display into pixmap.
  Device, Copy=[356, 0, 4, 360, 0, 0, 0]
; Make the display window the active window.
  WSet, 0
; Move window contents over 4 columns.
  Device, Copy=[0, 0, 356, 360, 4, 0, 0]
; Copy pixmap contents into display window on left.
  Device, Copy=[0, 0, 4, 360, 0, 0, 1]
  ENDFOR
END
```

运行程序，输入:

```
IDL> .RUN scroll
```

程序每次滚动一次。想再次运行程序，输入：

```
IDL> .Go
```

能修改程序让它一直滚动，直到让它停止吗？可以。

在继续下一次练习之前删除像素映射窗口。输入：

```
IDL> Wdelete,1
```

Z 图形缓冲区中的图形显示技巧

可以把 IDL 中的 Z 图形缓冲区想象成一个三维盒子，3D 对象可以被堆积在里面而不用关心它们的“实体”性质。这个盒子能用 16 位深度缓冲区来记录每个对象的深度。盒子的一边是投影平面。可以想象光线通过投影平面的每一个像素最终遇到盒子内实体对象。光线遇到的像素值就是投影到投影平面的值。用这种方法，Z 图形缓冲区就可以处理曲面和线条的自动消隐。图 61 就是此概念的图解说明。

Z 图形缓冲区

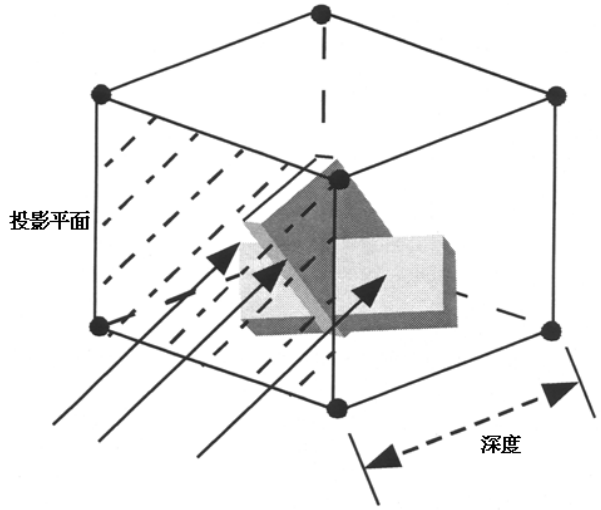


图 61　　Z 图形缓冲区可以被想象作一个能保留深度信息的 3D 盒子。光线射到 Z 图形缓冲区中的物体时，它们的像素值反过来被投影到投影平面上

其概念是，当物体被装入三维盒子里时，就可以得到投影平面的一幅快照或一幅图像。这是盒子中三维物体的二维投影。处在其他物体后面的物体就不被显示。（这种属性可以在一些 IDL 图形命令中用 Transparent 关键字来改变，以后将会看到。）快照实际上就是利用 TVRD 命令对投影平面的屏幕转储。

Z 图形缓冲区的实现

在 IDL 中，Z 图形缓冲区是用软件方式作为另一中图形输出设备实现的，类似于 PostScript 设备或 X 窗口，Win 或 Mac 设备。因此，要在 Z 图形缓冲区中画图形，必须用 Sep_Plot 命令将其变成当前的图形输出设备。和其他图形输出设备一样，Z 图形缓冲区可用 Device 命令和适当的关键字来配置。

常用于 Z 图形缓冲区的两个关键字是 Set_Colors 和 Set_Resolution。这两个关键字定义如下：

Set_Colors	Z 图形缓冲区中的颜色数。在缺省情况下，Z 图形缓冲区使用 256 种颜色。在 IDL 运行中，这是个非常少的颜色数目。如果希望 Z 图形缓冲区的输出具有和显示设备相同的颜色数目，就须设置这个关键字。
Set_Resolution	Z 图形缓冲区投影平面通常设置为 640 像素宽和 480 像素高。如果要在图形窗口显示 Z 图形缓冲区的输出结

果，就应该将 Z 图形缓冲区的大小设置成图形窗口的大小。例如：

DEVICE, Set_Resolution=[400,400]

一个 Z 图形缓冲区实例：两个曲面

要了解 Z 图形缓冲区是怎样工作的，先按如下创建两个名为 peak 和 saddle 的物体。（完成此例的命令可以在下载的本书配套程序中的文件 twosurf.pro 中找到。）

```
IDL>peak=shift(dist(20,16),10,8)
IDL>peak=Exp(-(peak/5)^2)
IDL>saddle=shift(peak,6,0)+shift(peak,-6,0)/2B
```

要在 Z 图形缓冲区中组合两个 3D 物体，首先查看这两个物体的各自形状。可以让它们在两个窗口中以不同的颜色表显示。首先，在颜色查询表中的不同部位装入蓝色和红色颜色表：

```
IDL>colors=!D.Table_Size/2
IDL>LoadCT, 1, ncolors=colors
IDL>LoadCT, 3, ncolors=colors, Bottom=colors-1
```

创建一个窗口，显示第一个物体的阴影曲面图。要注意的是，Set_shading 命令是用来将阴影处理的值限制在颜色查询表中特定部位。键入：

```
IDL>window, 1, xsize=300, ysize=300
IDL>set_shading, value=[0,coloe-1]
IDL>shade_surf, peak, zrange=[0.0,1.2]
```

接着将第二物体显示在它自己的显示窗口中。用颜色查询表的不同部位作为阴影处理参数。键入：

```
IDL>window, 2, xsize=300,ysize=300
IDL>set_shading, value=[colors, 2*colors-1]
IDL>shade_surf, saddle, zrange=[0.0,1.2]
```

使 Z 图形缓冲区成为当前设备

为了在 Z 图形缓冲区中组合两个物体，必须使 Z 图形缓冲区成为当前的图形显示设备。这可用 Set_Plot 命令来实现。Copy 这个关键字可将当前的颜色表复制到 Z 缓冲区中。保存当前图形显示设备的名字，以便能方便地返回。键入：

```
IDL>thisDevice=!D.Name
IDL>Set_Plot,' z',/Copy
```

配置 Z 图形缓冲区

接下来，必须将 Z 图形设备配置成具体的规格。在这个例子中，要限制颜色的数目，还要使缓冲区的分辨率与当前图形显示窗口尺寸相等。键入：

```
IDL>Device,Set_colors=2*colors, Set_Resolution=[300,300]
```

将物体装入到 Z 图形缓冲区中

现在，将两个物体装入到 Z 图形缓冲区中。要注意的是，键入这些命令时，将看不到任何事发生。因为输出已进入内存中的 Z 图形缓冲区里面，而不是显示设备。

```
IDL>Set_shading,values=[0,color-1]
IDL>Shade_surf,peak,zrange=[0.0,1.2]
IDL>Set_shading,values=[colors,2*colors-1]
IDL>shade_surf,saddle,zrange=[0.0,1.2],/noerase
```

对投影面进行拍照

接着，对投影表面进行快照。可通过 TVRD 命令实现。

```
IDL>picture=TVRD ()
```

在显示设备上显示结果

最后，返回到显示设备。打开一个新的窗口来显示“图像”，如下：

```
IDL>Set_Plot, thisdevice
IDL>Window, 3, xsize=300,ysize=300
IDL>TV, picture
```

输出结果应如图 62 所示。

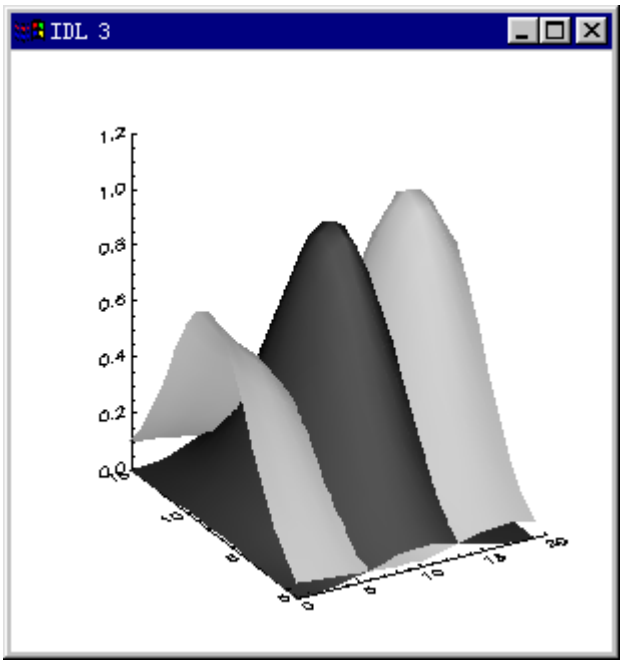


图 62　Z 图形缓冲区可以通过曲面的自动消隐来组合 3D 物体

Z 图形缓冲区的一些奇怪特点

仔细查看窗口 1 和 3 的输出。特别注意观察那些坐标轴的标记。可以注意到窗口 3（也就是来自 Z 图形缓冲区的输出）的轴的标记要稍微大些。由于某种原因，Z 图形缓冲区使用了一种缺省的不同于 IDL 在显示窗口中显示图形时所用的字符尺寸。

这种简单的事实，如果不意识到的话，会导致在 Z 图形缓冲区中建立一个 3D 坐标空间以及组合 IDL 图形命令时，造成大量的时间浪费。这主要是因为图形边缘的建立是基于缺省的字符尺寸，并且图形边缘在显示设备上和在 Z 图形缓冲区中是有差异的。它们差不多是一样的。但就是 “差不多”将使读者焦头烂额。

一种笨规则非常有用，就是，当要在 Z 图形缓冲区中画图时，总是设置!P.Charsize 系统变量。例如：

IDL>!P.Charsize=1.0

这里只是提供一个例子，查看图 62 中的图示。这幅图上的轴不是在 Z 图形缓冲区中创建的，因为那时它们将以屏幕分辨率被着色处理（也就是说,作为一幅图像）而且可用 PostScript 的分辨率对它们进行着色处理。如果!P.Charsize 关键字没有在对阴影曲面进行着色处理和随后坐标轴的添加之前被设置的话，就不可能在最后的输出中使坐标轴线对应在正确的位置上。

用 Z 图形缓冲区使图像变形

使用 Z 图形缓冲区另一种更强大的技术是用于三维数据的切片显示。这是可行的，因为 Z 图形缓冲区具有将图像变形到一个多边形平面上的能力。要了解是如何工作的，先用 LoadData 命令打开 80*100*57 的 3D MRI Head Scan 数据集。如下：

```
IDL>head=LoadData(8)
```

也许希望能在输入代码时开一个日志文件去截获这些命令，因为这些命令数量很多，而且必须正确无误地找到它们。日志文件将使读者很轻松地改变并重新运行这些命令。（这种日志文件已经创建好了，可以下载的本书配套程序中找到 warping.pro 文件。）

```
IDL>Journal, 'warping.pro'
```

一般来说，在 IDL 中用 Size 命令可以获得变量的维数和大小。为定义正确的图像平面，需要知道三维的尺寸。在这个例子中，“尺寸值”将比真实的维数值小 1，因为要用这个数值作为数组的索引号。IDL 的索引是号从零开始的。

```
IDL>s=Size(head)
IDL>xs=s[1]-1
IDL>ys=s[2]-1
IDL>zs=s[3]-1
```

假若想在这个数据集的中心显示三个正交切片，也就是说通过三维点（40,50,27）。可以用下述方法定义这些点：

```
IDL>xpt=40
IDL>ypt=50
IDL>zpt=27
```

接着，可以构建用来描绘这三个图像切片或平面的各个多边形。此例子中，每个多边形将是有四个点（矩形的四个角）的简单矩形。矩形的每点都要用一个（x,y,z）三位值描述。换句话说，每个平面都将是 3*4 的多边形。键入：

```
IDL>xplane=[ [xpt,0,0],[xpt,0,zs],[xpt,ys,zs],$
      [xpt,ys,0] ]
IDL>yplane=[ [0,ypt,0],[0,ypt,zs],[xs,ypt,zs],$
      [xs,ypt,0] ]
IDL>zplane=[ [0,0,zpt],[xs,0,zpt],[xs,ys,zpt],$
      [0,ys,zpt]]
```

下一步是获得与每个图像平面相对应的图像数据。这在 IDL 中用数组下标很容易做到。

```
IDL>ximage=head[xpt,*,*]
```

```
IDL>yimage=head[*,ypt,*]
IDL>zimage=head[*,*,zpt]
```

要注意的是这些图形都是 3D 图像（其中有一维是 1）。所关心的是与每个图像平面相关的 2D 图像，因此必须用 Reform 命令将这些 3D 图像转换成为 2D 图像格式。在此例中，Reform 命令将图像重新格式化成 80*100*1 的图像。当最后的一维为 1 时，IDL 将会舍弃它。这里的结果是一个 80*100 的图像。

```
IDL>ximage=reform(ximage)
IDL>yimage=reform(yimage)
IDL>zimage=reform(zimage)
```

要正确显示这些图像，需要确保图像数据已正确地缩放到与显示颜色数相匹配的范围内。相对于整个数据集的范围来调整数据是非常重要的。调整数据并装入颜色：

```
IDL>mindata=min(head,max=maxdata)
IDL>topcolor=!D.table_size-2
IDL>Loadct, 5, ncolors=!D.table_size-1
IDL>Tvlct, 255, 255,255,topcolor+1
IDL>ximage=bytscl(ximage,top=topcolor,max=maxdata,$
      min=mindata)
IDL>yimage=byscl(yimage,top=topcolor,max=maxdata,$
      min=mindata)
IDL>zimage=byscl(zimage,top=topcolor,max=maxdata,$
      min=mindata)
```

下一步要准备建立 Z 图形缓冲区。 Erase 命令将删去先前留在缓冲区内的任何内容。此例子中，用白色擦除，使得显示更加清晰。

```
IDL>thisdevice=!d.name
IDL>Set_Plot,’z’
IDL>device, Set_colors=topcolor, set_resolution=[400,400]
IDL>erase, color=topcolor+1
```

用 Scale3 命令创建 3D 坐标空间。这里的坐标轴是用每个维度的大小标记的。

```
IDL>Scale3, XRange=[0,xs], YRange=[0,ys], ZRange=[0,zs]
```

最后将要在 Z 图形缓冲区中对这些切片进行着色处理。为此需要用 Polyfill 命令。Pattern 关键字将用于设置想显示的图像切片。Image_Coord 关键字包含一系列与多边形的每个顶点相关联的图像坐标。Image_Interp 用于当图像被包裹到多边形上时，指定用双线性插值，而不是在最近邻域内重采样。T3D 关键字通过将三维变换矩阵应用到最后输出中，从而确保多边形出现在 3D 空间内。键入：

```
IDL>polyfill,xplane,/t3d,pattern=ximage,/image_interp,$
      image_coord=[ [0,0],[0,zs],[ys,zs],[ys,0] ]
IDL>polyfill,yplane,/t3d,pattern=yimage,/image_interp,$
      mage_coord=[ [0,0],[0,zs],[xs,zs],[xs,0] ]
IDL>polyfill,zplane,/t3d,pattern=zimage,/image_interp,$
      image_coord=[ [0,0],[xs,0],[xs,ys],[0,ys] ]
```

最后，将投影平面快速拍照，并显示结果。如下：

```
IDL>picture=tvrd()
IDL>set_Plot,thisdevice
IDL>window,xsize=400,ysize=400
IDL>tv,picture
```

如果已打开了日志文件，现在将其关闭：

IDL>Journal

输出结果应类似于图 63 所示。如果不是，可用文本编辑器修改日志文件中的代码来解决问题。要重新运行代码，保存文件并键入：

IDL>@warping

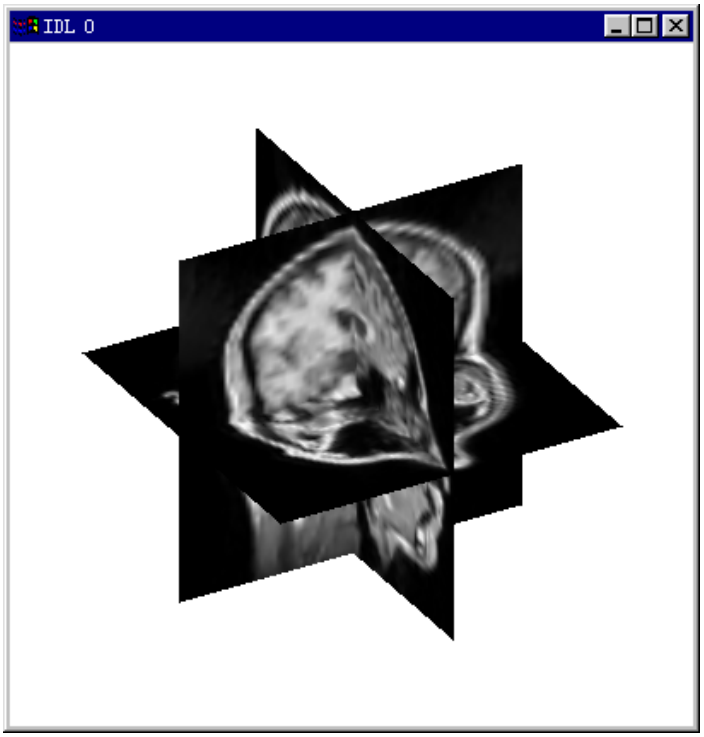


图 63　　在 Z 图形缓冲区将图像数据显示到平面中的一个实例

Z 图形缓冲区中的透明效果

注意图 63 中的每个切片的外部边缘有许多黑块。 它不属于图像的一部分，而是黑色背景噪音 。

Z 图形缓冲区的一个优点是可以在其中运用透明效果。例如，如果将命令 PolyFill 中的 Transparent 关键字设置为大约 20 或 25，那么图像中低于这个值的像素将是透明的。可以键入试试查看是什么效果。（也许想启动另一个日志文件。如果已经关闭了上次的日志文件，应给这次的文件赋一个新名。不幸的是，在 IDL 中还不能添加日志文件。给新的日志文件命名为 transparent.pro。这个日志文件的备份可以在下载的本书配套程序中找到。）

```
IDL>journal, 'transparent'
IDL>Set_plot, 'z'
IDL>erase, color=topcolor+1
IDL>polyfill, xplane,/t3d,pattern=ximage,/image_interp,$
      Image_coord=[[0,0],[0,zs],[ys,zs],[xs,0]],$
      Transparent=25
IDL>polyfill, yplane,/t3d, pattern=yimage,/image_interp,$
      Image_coord=[[0,0], [0,zs],[xs,zs],[xs,0]],$
      Transparent=25
IDL> polyfill, zplane,/t3d, pattern=zimage,/image_interp,$
      Image_coord=[[0,0], [xs,0],[xs,ys],[0,ys]],$
      Transparent=25
IDL>picture=tvrd()
```

```
IDL>set_plot,thisdevice
IDL>window,/free,xsize=400,ysize=400
IDL>erase,color=topcolor+1
IDL>tv,picture
IDL>journal
```

如果输入出现差错并且最后的输出图像看起来不正确，可对日志文件中的错误进行更改，然后重新启动日志文件，键入：

```
IDL>@transparent
```

将 Z 图形缓冲区效果与体数据着色相结合

Z 图形缓冲区效果经常和体数据的着色技术相结合，从而使数据的可视化更生动。例如，假设想创建这套数据集的等值面（一个等值面是一个在任何地方都有相同值的曲面。它就象数据的一个三维等值线图。）启动一个名为 isosurface.pro 日志文件。（此日志文件的备份可以在下载的本书配套程序中找到。）

```
IDL>Journal,'isosurface.pro'
```

要创建一个等值面，首先用 Shade_Volume 命令创建一组描述曲面的顶点和多边形。在下面的命令中，设置 Low 关键字以便所有比此等值面值大的值将被此等值面包围。变量 vertices 和 polygons 都是输出变量。它们将用在随后的 PolyShade 命令中来对曲面着色。看一下值为 50 的头部数据的直方图，是一个很适合等值级别。键入：

```
IDL>plot,   histogram(head), Max_value=5000
IDL>shade_volume, head, 50,vertices, polygons , /low
```

等值面是用 PolyShade 命令来进行着色处理的。必须确保使用前面建立的 3D 变换，键入：

```
IDL>scale3, xrange=[0,xs],yrange=[0,ys],zrange=[0,zs]
IDL>isosurface=polyshade(vertices,polygons,/t3d)
IDL>loadct, 0, ncolors=topcolor+1
IDL>tv,isosurface
```

现在，将这个等值面和穿过数据的 Z 切片组合起来，这个 Z 切片就是前面在 Z 图形缓冲区中获得的。注意现在在 Z 方向上截掉头部数据。键入：

```
IDL>shade_volume,head(*,*,0:zpt),50,vertices,$
      polygons,/low
IDL>isosurface=polyshade(vertices,polygons,/t3d)
IDL>isosurface(where(isosurface eq 0))=topcolor+1
IDL>tvlct, 70, 70, 70, topcolor+1
IDL>set_plot,'z', /copy
IDL>tv,isosurface
IDL> scale3, xrange=[0,xs],yrange=[0,ys],zrange=[0,zs]
IDL>polyfill, zplane,/t3d,pattern=zimage,/image_interp,$
            Image_coord=[[0,0],[xs,0],[xs,ys],[0,ys]],$
            Transparent=25
IDL>picture=tvrd()
IDL>set_plot,thisdevice
IDL>tv,picture
```

输出结果应类似于图 6 4 所示。如果不是，修改日志文件并重新运行该文件，如下：

```
IDL>@isosurface
```

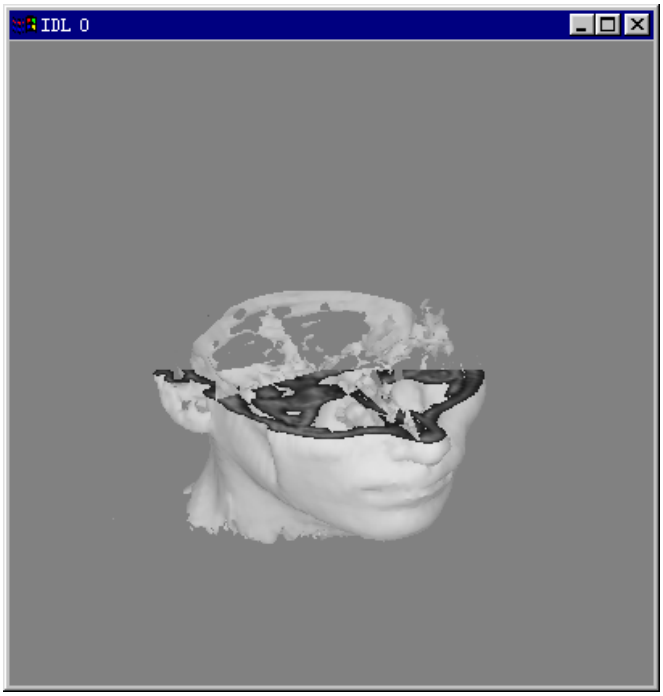


图 64　　等值面和数据切片在 Z 图形缓冲区中的组合

第六章 在 IDL 中读写数据

本章概要

本章旨在介绍 IDL 中的常用的输入和输出程序。IDL 中的基本原则是：“只要有数据，就可以将其读进 IDL”。IDL 对格式没有要求，也没有特别要求在将数据带入 IDL 时对数据进行准备。这使得 IDL 成为目前功能最强、最灵活的科学可视化分析语言。

具体来说，将学习：

1. 如何打开文件进行读写
2. 如何查找文件
3. 如何获得文件 I/O 的逻辑设备号
4. 如何获得机器的独立文件名
5. 如何读写 ASCII 或格式化的数据
6. 如何读写二进制的或二进制数据
7. 如何处理大型数据文件
8. 如何读写通用的文件格式，如 GIF 和 JEPG 文件

打开文件进行读写

IDL 中的所有输入和输出都是通过逻辑设备号完成的。可以把一个逻辑设备设想为一个管道，这个管道连接着 IDL 和要读写的数据文件。要从一个文件中读写数据，必须首先把一个逻辑设备号连接到一个特定的文件。这就是 IDL 中三个 Open 命令的作用：

openr	打开文件进行读。
openw	打开文件进行写。
openu	打开文件进行更新（也就是说，读和/或写）。

这三个命令的语法结构是完全相同的。首先是命令名，后面是一个逻辑设备号和要与该逻辑设备号相连的文件名。例如，将文件名 temp596.dat 和逻辑设备号 20 相连以便可以在此文件里面写入内容。如下：

```
OpenW, 20,'temp596.dat'
```

将会看到 Open 命令更常用的书写方式。例如，可能会看到类似于如下的 IDL 代码：

```
OpenR, lun, filename
```

此例中，变量 lun 保存了一个有效的逻辑设备号，变量 filename 代表一个机器特定的文件名，这个文件名将和此逻辑设备号联系起来。

注意，变量 filename 是一种机器特定的格式。这意味着如果它含有特定的目录信息，它必须用本地机器的语法来表达。而且它在某些机器（比如，UNIX 机器）上具有大小写敏感性，因为在这些机器上文件名有大小写敏感性。

查找和选择数据文件

IDL 被广泛使用的原因之一，是 IDL 可以在许多不同的计算机操作系统中运行。但由于不同的操作系统有不同的文件命名习惯（而且，特别用确定子目录的不同方式），这在以

独立于机器的方式指定文件名方面提出了挑战。幸好，IDL 提供了一些工具可让这项工作变得容易些。

选择文件名

也许获得机器独立文件名最容易的方法是用 Pickfile 对话框。IDL 命令允许用机器上自身的选择文件的图形对话框来交互式地从文件名列表中选择一个文件名。例如，从本地目录.pro 文件列表中选择一个文件名，可以键入如下命令：

```
IDL>filename=Dialog_Pickfile(Filter='*.pro',/Read)
```

注意，这个命令在 IDL5.0 以前的版本中命名为 Pickfile。

IDL5.2 版通过关键字 Multiple，赋予 Dialog_Pickfile 选择多个文件名（若它们存在于同一个目录下）的能力。使用了正常的依赖于平台的选择文件方式。例如，在用 WinDOws 操作系统的计算机上，通常先选择第一个文件，接着用 Shift 键和鼠标点击来选择在第一个文件和第二个文件之间的所有文件，或者用 Control 键和鼠标点击来选择一个额外的文件。

```
IDL>filename=Dialog_Pickfile(Filter='*.pro',/Read,/Muitiple)
```

如果要打开文件来写而不是去读，在对话框中，可用 Write 关键字代替 Read 关键字。甚至可以推荐一个缺省的文件名，键入：

```
IDL>outfile=Dialog_Pickfile(File='default.dat',/Write)
```

从这个对话框中返回的是带绝对路径的文件名，其形式与运行 IDL 的机器有关。也就是说，它使用机器自身的文件命名语法。键入以下命令就可以看到：

```
IDL>Help, filename, outfile
```

注意，Dialog_Pickfile 对话框中有一个“取消”按钮。若选择“取消”按钮，对话框会返回一个空字符串。所以在打开文件读写之前，总是希望检查返回的名字是否为空。

```
IDL>IF outfile EQ '' THEN Print, 'Whoops!'
```

选择目录名

在 IDL5.2 中，Dialog_pickfile 得到改进，因而它也能用于选择目录名而不仅是一个文件名。设置 Directory 关键字，在选择窗口内只列出目录而没有文件。

```
IDL>directory=Dialog_Pickfile(/Directory)
```

寻找文件

另一个有用的命令是 FindFile 命令。此命令返回一个包含所有符合给定文件要求的文件名的字符串数组。这在 IDL 程序中用于自动匹配并打开文件的任务中非常有用，或者是在任何时候不知道一个目录下有多少个文件的情况下，用于在该目录下创建一批文件的任务中也是非常有用的。

例如，要打印出当前目录下所有数据文件的长度（按字节计），可键入 IDL 代码：

```
Files=findfile('*.dat',count=numfiles)
If numfiles eq 0 then message,'no data files here!'
FOR j=0,numfiles-1 DO BEGIN
Openr, 10, files(j)
fileinfo=fstat(10)
print,fileinfo.size
Close, 10
```

```
ENDFOR
```

需要重点指出的是：FindFile 命令中文件说明('*.dat'）是以相对路径名给出，而不是绝对路径名。因此，命令返回也是相对路径名，而不是绝对路径名。这不同于 Pickfile 对话框，Pickfile 总是返回绝对路径名。

构造文件名

获得机器独立文件名的第三个很有用的 IDL 命令是 Filepath 命令。例如，假设要打开文件 galaxy.dat，它在 IDL 主目录下的 examples/data 子目录中。可以为此文件构造一个机器独立的绝对路径名，键入：

```
IDL>galaxy=filepath('galaxy.dat',Subdirectory=['examples','data']
```

如果想从其他的目录开始而不是 IDL 主目录，可以用 Root_Dir 关键字指定开始的目录名。例如，要在当前目录的子目录 coyote 中构造此文件的一个路径名，可以键入：

```
IDL>cd,current=thisdir
IDL>galaxy=filepath('galaxy.dat',root_dir=thisdir,$
        Subdirectory='coyote')
```

注意：Filepath 命令并没有实际找到此文件，它只是构造了一个文件路径名。构造的文件名甚至在机器中可以不存在。

获取逻辑设备号

在 IDL 中所有文件输出和输入都是在一个逻辑设备号上完成的。一个 Open 命令的作用是将一个特定的文件（通过其文件名来指定）和一个逻辑设备号相关联。有 128 个逻辑设备号可供使用。它们被分成两类。

表 8　128 种逻辑设备号被分成两类。一类可以直接使用，另一类用 Get_Lun 和 Free_Lun 命令获取和管理

逻辑设备号	用途
1-99	这些号可以在 Open 命令中直接使用
100-128	这些号通过 Get_Lun 和 Free_Lun 命令获取和管理

直接使用逻辑设备号

要直接使用逻辑设备号，只能在 1-99 中选择一个号，并用 Open 命令使用它。要做的是选择一个当前没有使用的号。例如，可用逻辑设备号 5 来打开当前目录的子目录 coyote 中的 galaxy.dat 文件，键入：

```
IDL>CD,current=thiedir
IDL>filename=Filepath(Root_Dir=thisdir,$
        Subdirectory='coyote','galaxy.dat')
IDL>Openr, 5, filename
```

一旦 1-99 中的某个逻辑设备号分配给一个文件后，它就不能再分配了，直到该逻辑设备号被关闭或退出 IDL（这将自动关闭所有打开的逻辑设备号）。

当完成了对逻辑设备号的操作（也就是说，不想再对文件进行读写），可用 Close 命令关闭它，并使其可以重新使用：

IDL>Close, 5

让 IDL 管理逻辑设备号

多数情况下（特别在 IDL 程序中），最好让 IDL 管理逻辑设备号。可使用 Get_Lun 和 Free_Lun 命令完成此项功能。有两种方法让 IDL 返回一个逻辑设备号。可以直接使用 Get_Lun 命令。如：

IDL>Get_Lun,lun

IDL>OpenR,lun,filename

或者，用带关键字 Get_Lun 的 Open 命令来间接完成：

IDL>OpenR,lun,filename,/Get_Lun

这个命令运用隐含的 Get_Lun 命令将作为结果的逻辑设备号存入变量 Lun 中。这是逻辑设备号最常用的获取方法，特别是在 IDL 程序中。（注意，变量名不一定是 Lun。可以给出喜欢的名字。如果打开了几个文件，就需要几个不同的名字。）

当完成了对逻辑设备号的操作（也就是说，不想再对文件进行读写），可用 Free_Lun 命令关闭它。如：

IDL>Free_Lun，Lun

使用 Get_Lun 和 Free_Lun 命令的好处在于不必记住哪个逻辑设备号是可用的或没被使用的。Get_Lun 程序保证返回一个有效的逻辑设备号（假设同时已经打开的文件少于 28 个）。如果是在过程和函数中打开文件，一般最好使用 Get_Lun 命令。因为如果直接选择某个特定的逻辑设备号，不能保证它是可用的或没有被使用的。

判断哪些文件和哪些逻辑设备号相连

使用带 Files 关键字的 Help 命令，可以很容易判断哪些文件和哪些逻辑设备号相连。

IDL>Help,/Files

读写格式化数据

IDL 在读写格式化数据方面有两种格式化文件之区分：自由文件格式和确定的文件格式。格式化文件有时叫做 ASCII 文件或者纯文本文件。

自由文件格式	自由格式文件用逗号或空白（tab 键和空格键）分开文件中的每个元素，这没有确定的文件格式正规。
确定的文件格式	确定的格式文件是用格式说明按照给定的规范进行编排的。IDL 格式说明和 FORTRAN 或 C 程序中的格式说明类似。

写自由格式文件

在 IDL 中，写一个自由格式文件极其容易。只要用 PrintF 命令将变量写入文件即可。这和在显示窗口上用 Print 命令打印变量的形式几乎相同。IDL 在写文件时自动在数据的元素间加入空格。

例如，键入下面这些命令，创建数据并写入文件：

IDL>array=FIndGin(25)

IDL>vector=[33.6,77.2]

```
IDL>scalar=5
IDL>text=[‘array’, ’vector’, ’scalar’]
IDL>header=’Test data file.’
IDL>created=’created: ’ + SysTime()
```

接着，将一个逻辑设备号（让 IDL 自己选择一个并放到变量 lun 中）和特定文件相连来打开文件写入，键入：

```
IDL>OpenW , lun, ’test.dat’,/Get_Lun
```

最后，将数据写入文件，并关闭数据文件，键入：

```
IDL>PrintF, lun, header
IDL>PrintF, lun, created
IDL>PrintF, lun, array, vector, scalar, text
IDL>Free_Lun, lun
```

用一个文本编辑器打开文件检查。类似于如下所示：

```
Test data file.
created: Tue Nov 28 15:50:58 2000
     0.000000      1.00000      2.00000      3.00000      4.00000      5.00000
      6.00000      7.00000      8.00000      9.00000      10.0000      11.0000
      12.0000      13.0000      14.0000      15.0000      16.0000      17.0000
      18.0000      19.0000      20.0000      21.0000      22.0000      23.0000
      24.0000
      33.6000      77.2000
       5
array vector scalar
```

注意：IDL 在数组变量的每个元素间设置空白区，并使每个新变量另起一行开头。IDL 在缺省情况下使用 80 列宽度。如果需要不同的列宽度，可以用带 Width 关键字的 OpenW 命令设置。

读自由格式文件

许多 ASCII 文件是自由格式文件。例如，从电子数字表程序中保存的文件大多是自由格式文件。一种特殊的自由格式数据是从键盘或标准输入读取的数据。

在 IDL 中，有两种命令可以读取自由格式文件。

Read　　读取从标准输入或键盘上读入自由格式数据

ReadF　　从文件中读入自由格式数据

读取自由格式文件的规则

无论是从键盘上还是从文件中读取自由格式的数据，IDL 遵循下列 7 种规则：

1.　如果读入到字符串变量中，那么，在当前行剩下的所有字符都将读入该变量中。

观察上面数据文件的第一行，此行内有三个单词。此项规则意味着，一旦 IDL 开始为字符串变量读入数据，那么 IDL 将一直读到行末，且不会停止。其原因是在 IDL 的所有数据类型中，字符串变量可以是任意尺寸的，而且空格符也是 ASCII 字符。

打开该数据来读入，试着只读入一个单词，如下：

IDL＞Open, lun, ‘test.dat’, /Get_Lun
IDL>word = ‘’
IDL>ReadF,lun,word
IDL>Print, word

所看到的是整个第一行都被读入到 word 变量中。

Test data file.

（如果想将第一行分解成单个的单词，可以用 IDL 的字符串处理程序来完成。）这种可以读到行末的功能是 IDL 的一个优点。为什么这样说呢？首先，将文件指针重置到文件的开头。可用 Point_Lun 命令实现此项功能。键入：

IDL>Point_Lun, lun, 0

现在可将数据文件的前两行读入到 header 变量中。键入：

IDL＞header = StArr(2)
IDL>ReadF, lun, header
IDL>Print, header

这些命令将读出文件的前两行，并将文件指针定位到数组的起始处。可以在同一行上同时输出头两行的内容。

Test data file. created: Tue Nov 28 15:50:58 2000

2. 输入数据必须用逗号或空白分隔（空格键或 tab 键）。

在 test.dat 数据文件中所用到的就是这种格式的数据。IDL 在数组变量中的每个元素间插入 5 个空格。

3. 输入通过数字变量完成。数组和结构都可作为数字变量的集合。

这意味着，如果正在读入的变量中，比如说，10 个元素，IDL 将从数据文件中读入 10 个分开的数值。它将采用以下的两条规则来确定这些数据存在于文件的何处。

4. 如果当前读入行是空的，并且还有变量要求输入，则读取另一行。
5. 如果当前读入行不是空的，但是没有变量要求输入，则忽略此行剩下的数。

为了解其含意，可键入：

IDL＞data = FltArr(8)
IDL>ReadF, lun, data
IDL>Print, data

将看到：

0.00000 1.00000 2.00000 3.00000 4.00000 5.00000
6.00000 7.00000

在此例中，IDL 从文件中读入 8 个分开的数据值。当读取到第一行数据的末端时，它自动进入到第二行（规则 4），因为还有更多的数据要求读入。当数据读入到第二行的中部时，规则 5 起作用了。如果现在还要读入更多的数据，那将从数据的第三行开始，因为第二行的其他部分被忽略了。键入：

IDL＞data = FltArr(3)
IDL>ReadF, lun, vector3
IDL>Print, vector3

将看到：

12.0000　　13.0000　　14.0000

变量 vector3 包含的数值为 12.0、13.0 和 14.0。现在，文件指针定位在文件中的第四数据行上（规则 5）。

6. 尽量将数据转换为变量所希望的数据类型。

要了解这是什么意思，将第四、第五行数据读入到一个字符串数组，以便让文件指针定位在文件中的第六行数据（即起始数值为 33.6000 的那行）。键入：

```
IDL＞ dummy = StrArr(2)
IDL> ReadF,  lun,  dummy
```

假设想读入两个整型值。IDL 尽量将数据（在此情况下，为浮点数）转换为整型。

键入：

```
IDL＞ ints = IntArr(2)
IDL> ReadF,  lun,  ints
IDL> Print,  ints
```

将看到：

33　　77

注意，浮点数被简单的截取成整数。在转换处理过程中并没有采用四舍五入的规则来保证最接近的整数值。

7. 复数数据必须有实数和虚数两部分，用逗号分隔，并用括号括起来。如果仅仅提供了单个数值，它将被认为是实数部分，而虚数设置为 0。例如，可通过键盘读入复数数据：

```
IDL>value = ComplexArr(2)
IDL> Read,  value
: (3, 4)
: (4.4, 25.5)
IDL> print,  value
```

在结束这部分学习之前，确保已将 test.dat 文件关闭。键入：

```
IDL＞Free_Lun,  lun
```

读写自由格式文件的实例

学会用 IDL 读写数据最容易的方法是看一些实例。下面实例说明在读入文件头、处理以列排列的数据以及处理已读入 IDL 的数据这些方面的常用 IDL 技巧。

读一个简单数据文件

从读刚创建的 test.dat 文件中的数据开始。首先，创建用于读文件中数据的变量。

```
IDL>header=strarr(2)          ; two header lines.
IDL>data=fltarr(5,5)          ; floating point array.
IDL>vector=intarr(2)              ;  two-element integer vector.
IDL>scalar=0.0                    ;  floating-point scalar.
IDL>string_var=’’                 ;  a string variable.
```

注意，此时数据变量将是 5*5 的数组。而在文件中是以 25 个元素的矢量保存的。用这种方法读入数据相当于读入 25 个元素的矢量，并将其重新格式化为 5*5 的数组。记住 IDL 中的数据是按行存储的。

根据规则一，不能将文件末端的一行文本读到三个元素的字符串数组中。不得不将其读入单个字符串变量，然后用 IDL 的字符串处理命令将该文本字符串分解到后续的变量中。

可立即从文件中读出所有数据，键入：

```
IDL>openr,  lun,  'test.dat',  /Get_Lun
IDL>ReadF,  lun,  header,  data,  vector,  scalar,  string_var
IDL>free_Lun,  lun
```

要将包含文件末行文本的字符串变量转换成三个元素的字符串数组，首先要将变量前后两头的空白字符除去。键入：

```
IDL>thisstring=strtrim(string_var,2)
```

有时若将字符串转换成字节型数组，字符串的处理就会容易些。可以先处理字节型数组，最后再将其转换成字符串。当然也可使用其他方法，但是这种方法在这里更好一些。

```
IDL>thisarray=Byte(thisstring)
IDL>help,thisarray
```

这是 19 个元素的字节型数组。要知道空格的 ASCII 码字符，可用 IDL 查出：

```
IDL>blank=Byte(" ")
IDL>print,blank
IDL>help,blank
```

注意，Byte 命令处理一个字符串后的返回值总是一个数组。在此例中，是一个元素的数组。为了以后不至于混淆，将其转换成一个数值。

```
IDL>blank=Blank[0]
```

输出空白字符的 ASCII 码值是 32。

可用 Where 命令显示字节型数组中的空白字符。

```
IDL>vals=Where(thisarray EQ blank)
```

最后，将字符串转换成三个元素的数组。

```
s=strarr(3)
s[0]=string(thisarray[0:vals[0]-1])
s[1]=string(thisarray[vals[0]+1:vals[1]-1]
s[2]=string(thisarray[vals[1]+1:*]
```

写列格式数据文件

在文件中数据按列储存是不稀奇。需要了解如何用 IDL 读写这种数据。IDL 将会给粗心的程序员一个惊喜。要知道究竟是怎么回事，可以写一个列格式数据文件。用下列命令将数据读入 IDL：

```
IDL>data=LoadData(15)
```

这个数据是一个有三个字段的结构：lat, lon 和 temp。每个字段都是 41 个元素的浮点矢量。可用如下命令从结构中提取矢量：

```
IDL>lat=data.lat
IDL>lon=data.lon
IDL>temp=data.temp
```

接着，打开一个写入数据文件，键入：

```
IDL>OpenW,lun,'column.dat', /Get_Lun
```

需要将三列数据写入这个文件，通过自由格式输出。这可以在一个循环中完成。

```
IDL>printf, lun, 'column data: lat, lon, temp'
```

```
IDL>FOR j=0, 40 DO printf, lun,lat[j], lon[j], temp[j]
IDL>free_lun, lun
```

column.dat 文件的前四行应如下：

```
Column data: lat, lon, temp
33.9840    –86.9405    36.9465
26.2072    –121.615    20.1868
42.1539    –103.733    231.604
```

读列格式数据文件

到目前为止一切正常。当试着将列格式数据读入 IDL 时问题就出来了。可能得按如下操做。首先，创建要读入数据的变量。

```
IDL>header=''
IDL>thislat=fltarr(41)
IDL>thislon=fltarr(41)
IDL>thistemp=fltarr(41)
```

打开 column.dat 文件读首行：

```
IDL>OpenR, lun, 'column.dat' ,/Get_Lun
IDL>ReadF, lun, header
```

由于是用一个循环将数据放进文件的，所以也可能会用一个循环从文件中将数据读出。

```
IDL>FOR j=0, 40 DO ReadF,  lun,  thislat[j],  thislon[j],  thistemp[j]
```

但这不奏效。虽然上面的命令没有错误，但没有数据读入变量（如打印变量值，它们将是零）。

其原因是 IDL 中有一个严格的规则，即不能用带下标的变量来读入内容。原因是 IDL 将带下标的变量作为值而不是作为变量的引用传递给象 ReadF 这样的 IDL 程序。以数值传递的数据不能在被调用的子程序中改变，因为被调用的子程序只是获得该数据的备份，而不是获得该数据的指针。要改变这种属性需要对 IDL 进行大的改动，然而这是不可能的。

有两种方法解决这个问题。第一种是将数据读入一个循环中的临时变量。这种方法最好是运用文本编辑器将命令输入文件中来完成，因为很难在 IDL 命令行上编写多行循环。可用 Point_Lun 命令将文件指针返回到数据文件的起始处。

```
IDL>Point_Lun, lun, 0
```

在文本文件 loopread.pro 中输入下列命令。

```
temp1=0.0
temp2=0.0
temp3=0.0
ReadF, lun, header
FOR j=0, 40 DO BEGIN
    ReadF,lun, temp1, temp2, temp3
    Thislat[j]=temp1
    Thislon[j]=temp2
    Thistemp[j]=temp3
ENDFOR
END
```

执行文本文件中的代码：

```
IDL>.Run loopread
```

将原始矢量的值和刚读入矢量的值打印出来，将发现它们是相同的。例如：

```
DL>Print, lat, thislat
```

虽然这个方法奏效，但它不是最好的方法。主要是因为它用了一个循环，循环在 IDL 中很慢。41 次循环的速度也许无所谓，但如果是 41,000 次循环，执行速度将是个麻烦问题。

较好的方法是将数据一次性读到 3*41 的数组内，然后在用 IDL 提供的数组处理命令将矢量从这个较大数组中提出。要知道这是如何实现的，可用下面命令将数据文件指针返回到头：

```
IDL>Point_Lun, lun, 0
```

接下来，将数据一次性读到 3*41 的浮点数组：

```
IDL>header=''
IDL>array=fltarr(3,41)
IDL>ReadF, lun, header, array
```

用数组下标将大数组的矢量分离出来：

```
IDL>thislat=array[0,*]
IDL>thislon=array[1,*]
IDL>thistemp=aarry[2,*]
IDL>Free_lun, lun
```

注意，这些新的矢量是列矢量（也就是说，是 1*41 的二维数组。）键入：

```
IDL>Help, thislat, thislon, thistemp
```

要使这些列矢量转换成更熟悉的行矢量，可用 Reform 命令将 1*41 的数组转换成 41*1 的数组。当多维数组的最后维为 1 时，IDL 便舍弃这个维。键入：

```
IDL>thislat=ReFORm(thislat)
Idl>thislon=ReFORm(thislon)
Idl>thistemp=ReFORm(thistemp)
Idl>Help, thislat, thislon, thistemp
```

创建读列格式数据的模板

由于许多人都用有列格式数据文件，IDL 引进了一个新的程序以便更容易读此类数据文件。提供帮助的是两个新命令：ASCII_Template 和 Read_ASCII 命令。ASCII_Template 命令是个组件程序，可以引导读者按步骤定义自己的列数据。可以给每列数据命一个名字，告诉 IDL 数据类型，甚至可以跳过一些列。运行该程序的结果是用结构变量的形式生成数据文件的模板。这个模板可以传给 Read_ASCII 命令，数据就会按模板规范来读取。其结果是一个根据模板的规范读出来的带有字段名的 IDL 结构变量。要知道它如何运用到上面文件中，可键入：

```
IDL>fileTemplate=ASCII_Template(‘column.dat”)
```

按照出现在显示器上的组件对话框模式指导进行操作。其形式有三页，在第一页上可以看到数据文件的样本行，左边有它们的行号。在标有 Data Starts at Line 的文本组件中：输入 2。这允许在文件中跳过一行的文件头。在组件的右下角选择 Next 按钮进入下一页。

注意在这页里，每行的字段数列出的是三个，并可选择 White Space 按钮作为数据分隔符。若信息正确，就点击 Next 按钮进入最后一页。

在这页中，数据组可以被命名并且可以被指定数据类型。如果想在数据中跳过一列或多列，可在界面的右上角的下拉式列表框 Type 中将其设为 Skip Field 类型。通过界面右上角的 Name 文本组件，将三个字段分别命名为 Latitude，Longitude 和 Temperature。这些字段都是浮点类型。

当完成命名和选择数据列的数据类型后，选择界面上的 Finish 按钮。运行结果是一个用于描述文件中数据的 IDL 结构变量，它们可被用作 Read_ASCII 命令的输入。

```
IDL>help, fileTemplate, /structure
```

要读文件中的数据，可使用 Read_ASCII 命令。

```
IDL＞data = Read_ASCII (‘column.dat’,  Template= fileTemplate)
```

数据立刻被读取，结果是包含三个字段 latitude, longitude, temperature 的 IDL 结构变量。

IDL>Help, data, structure

如果要提出结构中的矢量，可键入：

IDL>thislat=data.latitude

IDL>thislon=data.longitude

DL>thistemp=data.temperature

ASCII_Template 和 Read_ASCII 命令既可用于自由格式的数据文件，也可用于下面将要讨论的确定格式的数据文件。

用确定的文件格式写入

读写确定文件格式可同样用 ReadF 和 PrintF 命令，它们刚才已用于自由格式文件，但现在文件格式已由 Format 关键字明确声明。（在读写标准输入和输出时，也可将 Format 关键字用于 Read 和 Print 命令）。

Format 关键字的语法和在 FORTRAN 程序中使用的格式规则类似。尽管格式规则很复杂，这儿却有许多共同之处。如果使用过 FORTRAN 代码来读写数据的话，便会很快地熟悉它们。

一些共有的格式说明符

在 IDL 中有许多格式说明符，有一些是共有的。如矢量数据的定义：

IDL>data=Findgen(20)

I 修饰整型数据。将数据按整数输出，以每个两位，每行 5 个，每个之间有两个空格的形式打出：

IDL>thisFormat=’(5(I2,2x),/)’

IDL>Print, data, Format=thisFormat

F 修饰浮点数据。将数据按浮点数输出，以小数点后两位，每行一个的形式打出。（这个数要为小数点留出一位）

IDL>thisFormat=’(f5.2)‘

IDL>Print,data,Format=thisFormat

D 修饰双精确数据。将数据按双精度输出，按每行写 5 个，每个数之间有 4 个空格，小数点右边有 10 位的形式打出。

IDL>thisFormat=’(5(D13.10, 4x))’

IDL>Print, data*3.2959382, Format=thisFormat

E 用科学记数法描述浮点数据(如：116.36E4)

IDL>thisFormat=’(E10.3)’

IDL>Print, data*10E3, Format=thisFormat

A 描述字符型数据。将数字转换成字符串，并以四个字符长的字符串形式写出，每个字符串用两个空格分开，每行 4 个字符串：

IDL>thisFormat=’(4(A4, 2x))’

IDL>Print,StrTrim(data,2), Format=thisFormat

NX 跳过 n 个空格字符。

写用逗号分隔的确定格式数据文件

有时数据文件必须用确定格式书写，以方便它们被其他软件读取。用逗号分隔的数据文件就是这类文件的典型代表。例如，要把上面读到的数据写成此类文件。下面就是如何实现。可用

Format.dat 作为写入的文件名，键入：

```
IDL>OpenW, lun, 'Format.dat', /Get_Lun
```

创建一个逗号字符串变量，如：

```
IDL>comm.=' , '
```

以确定格式写出文件，用 10 位宽并有三位小数的浮点数，后接一个逗号和两个空格，键入：

```
IDL>thisFormat='(F10.3, A1, 2x, F10.3, A1, 2x, F10.3)'
IDL>FOR j=0,40 DO PrintF,lun, thisLat[j],comma,$
    thisLon[j],comma,thisTemp[j],Format=thisFormat
IDL>Free_lun,Lun
```

数据文件前三行如下所示：

```
48.000,      -121.128,      36.946
44.843,      -108.133,      163.027
29.865,      -109.668,      89.870
```

读出用逗号分隔的确定格式文件

要读取刚创建的确定格式数据文件，键入：

```
IDL>OpenW, lun, 'Format.dat', /Get_Lun
IDL>thisFormat='(2(F10.3, 3X), F10.3)'
IDL>array=Fltarr(3, 41)
IDL>ReadF, lun, array, Format=thisFormat
IDL>Free_lun, lun
```

这很简单，注意只要按上面的格式把逗号当作空格跳过即可。

从字符串中读取格式数据

ReadS 是一个有用的 IDL 命令，可以从字符串变量而不是从文件中为自由格式或确定格式读取数据。ReadS 运用了和命令 Read 和 ReadF 相同的读取格式数据规则。也就是说，使用 ReadS 就象从数据文件中读取一样，所不同的是所读的对象是一个字符串变量。

当大量信息需从文件头部读取时，此命令特别有用。例如，假设 ASCII 数据文件的第一行说明了数据文件的行数和列数，随后是采集数据的时间。例如：

```
10      24500      12 June 1996
```

此文件头可以从文件中读取，并且可创建一个大小正确的数组来读取数据。如下：

```
firstLine=''
ReadF,lun,firstLine
columns=0
rows=0
date=''
ReadS,firstLine,columns,rows,date
dataArray=FltArr(columns,rows)
```

读写二进制数据

迟早，数据会越来越多。若这样，就要开始思考更好的数据储存方法。二进制数据比格式化数据紧凑得多，经常用于存储大数据文件。有两种命令读写二进制数据，它们与早期用来读取格

式数据文件的 ReadF 和 Print 命令等效。它们是 ReadU 和 WriteU 命令。

二进制数据文件基本是以一长串的二进制字节存在文件中。这些字节的含义（也就是说，这些字节如何翻译成特定数据类型和结构的）很难描述的，除非刚开始就知道文件写入的是什么内容。在文件里面，各字节都很相似。将字节读入正确类型和结构的变量中就可理解了。原则上这很容易做到，因为多数数据类型有给定的字节长度。例如，每个浮点值有 4 个字节。IDL 整数有两个字节，等等。

要读取非格式数据文件，简单定义变量，打开文件读取，并用 ReadU 命令将字节一个接一个地读入变量中。如果给定了变量的数据类型和组织结构，每个变量按其要求从文件中读出相应的字节数。例如，一个 5 个元素的浮点矢量将从文件中读取五（元素数）乘四（一个浮点值的字节数）共二十个字节。

读取二进制图像数据文件

例如，假设想读取储存在 coyote 子目录下的几个非被格式化图像数据文件中的一个。这些文件碰巧包含的是字节型数据，但它们也能容易地包含整数与浮点数。下面的一些命令被用来打开其中之一，galaxy.dat 文件。星系图像的字节已被组织成一个 256*256 字节的数组。（这个代码已假定 coyote 子目录存在于 IDL 主目录中。）

```
IDL>filename=Filepath(Root_Dir=!Dir, $
    Subdirectory='coyote','galaxy.dat')
IDL>OpenR, lun, filename, /Get_Lun
```

此幅图像的结构为一个 256*256 的数组。如果事先不知道这些，为正确地读取这个数据，将会花费很大的精力。因为在数据文件里没有信息提示这些字节是如何被组织起来的。

在研究二进制数据文件之前，如果不知道它有多大，那么只有一件事情要做。这也就是要知道它包含多少字节。例如，FStat(文件的状态)命令能告知文件的字节数。

```
IDL>fileInfo=Fstat(lun)
IDL>Print,fileInfo.size
    65536
```

在这个文件里有 65536 个字节，但这没有被告知这个数据的结构是怎样组织的。例如，这些字节能被组成 128*512 的二维数组，也可以被组成 64*64*16 的三维数组。没有办法知道这些。程序员在毫无办法的情况下，有时会尝试用各种数组组合来显示数据。如果这个看起来不对，就试着用另外一个，等等。

在此例子中，这些数据被组成 256*256 字节型数组。因此图像变量可以这样设定：

```
IDL>image=BytArr(256,256)
```

现在，从文件里读取数据，然后关掉文件，如下：

```
IDL>ReadU, lun, image
IDL>Free_Lun, lun
```

显示数据，键入：

```
IDL>Window,Xsize=256,Ysize=256
IDL>Tvscl, image
```

写二进制图像数据文件

假设在图像上进行了一些处理，想将结果保存在另一个数据文件里。例如，在图像上进行了一个 Sobel 边界增强操作，如下：

```
IDL>edge=Sobel(image)
```

Sobel 操作不仅可以帮助图像增强它的边缘，而且被返回的图像不再是字节类型。事实上，它是整数数据，键入：

IDL>Help, image, edge

在 IDL 中整数占两个字节，因此如果这个数据被写入一个数据文件，这个文件的大小将会是原来的字节型数据文件的两倍。这些对于那些试图读取处理后的图像数据文件的人来说可能有些混淆。所以可以考虑在文件里给出一些信息，告诉用户关于这种数据的相关类型以及怎样组织的。这个文件信息可以按如下定义：

IDL>fileInfo =’Sobel Edge Enhanced, 256 by 256 INTEGERS’

这些字符串可以在文件里写在图像数据的前面。

但是，字符串 fileInfo 也是一串字节。在此例中，它是一个有 31 个字节的字符串。如果不知道关于此二进制文件的这一点，以后从文件里读取数据将会非常困难。例如，设或许认为信息字符串是 30 个字节长。这样，在读完文件信息字符串之后，继续读出的每一个整数（两个字节）将会完全是错误的。

为避免这么多的限制，大多数二进制数据文件的文件头有着固定的大小（通常是 256 的倍数）。例如，假设决定所有的图像文件都有一个 512 个字节的头文件。可用 IDL 里的 Replicate 和 String 命令创造一个有 512 个字节长的空格字符串。

IDL>header = String (Replicate (32B, 512))

字节值 32 是一个空格字符的 ASCII 值。把文件信息字符串插入这个长一点的文件头字符串中，从而建立具有一个正确尺寸的头文件。可以用 IDL 里的 StrPut (把一个字符串放进另外一个字符串)命令。如下：

IDL>strPut, header, fileInfo, 0

最后，将文件头和数据写入一个新的二进制数据文件中。如下：

IDL>OpenW, lun, ‘process.dat’

IDL>WriteU, lun, header, edge

IDL>Free_Lun, lun

当字符串被写进二进制的文件时，也就相当于，字符串含多少个字符，就有多少个字节被写进文件。

读取带有文件头的二进制数据文件

假设想读取上面刚建立的文件，里面有 512 个字节的头文件信息，紧接着是 256*256*2 个字节的图像数据。可以将头信息看成是一个字符串，因为它含有关于文件里保存的数据类型的文本信息。

对于格式化的数据，文件头的变量可以被创建成空一个字符串或一个空字符串数组，以便数据文件的全部行能一次读入到文件头变量。对于二进制的文件这是不可能的。事实上，二进制字符串数据的规则是当从文件里读取字符串时，仅仅只是读取确定个数的字节去填满该字符串目前的长度。这样，一个文件头被定义为一个空字符串，将不会从二进制的文件里读出什么。

这意味着在从文件中读取字符串之前，必须知道正在读的字符串长度。在刚创建的 process.dat 文件里，文件头有 512 个字节长。可以用 String 和 Replicate 命令建立一个合适长度的空白字符串去读入，象前面那样。但是更容易的方法是把文件头读进一个字节型数组变量，然后把它转变成一个字符串。键入：

IDL>OpenR, lun, ‘process.dat’, /Get_Lun

IDL>header =BytArr(512)

IDL>ReadU, lun, header

IDL>Print, String(header)

Sobel Edge Enhanced,256 by 256 INTEGERS

从这条信息里面，就能够建立正确的数据数组，并从文件里读出图像数据并显示它们。如下：

```
IDL>edgeImage =IntArr(256,256)
IDL>ReadU, lun, edgeImage
IDL>Free_Lun, lun
IDL>Window, XSize=256,YSize=256
IDL> TV, edgeImage
```

二进制数据文件的一些问题

不幸的是，虽然处理二进制的数据较方便，但同时在使用这种数据是也有一些相关的问题。首先，二进制的数据与机器类型有很大关系。在 SUN 计算机上写的数据，如果不做任何处理的话，在 SGI 或者 HP 计算机上经常读不出来，而在 PC 或者 Macintosh 计算机上是肯定读不出来的。（ByteOrder 命令能够用来解决许多这类问题。IDL5.1 版将新的 Swap_If_Big_Endian 和 Swap_If_Little_Endian 关键字引入到 Open 命令中，可用于在各种各样的机器结构上编写代码来读取二进制数据。）

为了能在不同的机器结构上传递二进制数据，IDL 支持 XDR（eXternal Data Representation，外部数据表示）文件格式。XDR 格式是 Sun Microsystems 创建的公用的数据格式。在几乎所有的现代化计算机上都可用。它在二进制文件里存储了少量的元数据（数据本身的一些附加信息）。但是 XDR 文件仍然很简洁。

如果文件是用 XDR 二进制的形式写的，数据文件在计算机之间很容易传递。换句话说，XDR 二进制文件成为跨机器结构的文件格式。

要读写 XDR 格式的文件，必须用 XDR 关键字打开。例如：把上面的 process.dat 文件写成 XDR 文件，可以键入：

```
IDL>OpenW,  lun,  'process.dat', /Get_Lun,  /XDR
```

常规的 WriteU 命令用来把数据写进文件：

```
IDL>WriteU, lun, header, edge
IDL> Free_Lun, lun
```

在 XDR 文件里字符串的长度被存储起来，并随着字符串本身一起被恢复。这意味着不必要象一般的二进制文件那样，每次都初始化一个正确长度的字符串变量。例如，打开读取 XDR 文件里的信息，可以键入：

```
IDL>OpenR, lun, 'process.dat',/XDR
IDL> thisHeader = ''
IDL> thisData =IntArr(256,256)
IDL> ReadU, lun, thisHeader, thisData
IDL> Free_Lun, lun
```

用关联变量存取二进制数据文件

大型的二进制数据文件通常都有一系列的重复单元组成。例如，一个卫星每隔半小时就拍摄一幅 512*600 像素的浮点图像，并将这些图像一个接一个地存储在一个数据文件里，这个文件每隔一定的时间被下载一次。在数据文件里包含 50-100M 的数据是很寻常的。一个 IDL 关联变量通常是处理这种数据形式的最好方式（有时候是惟一的方式）。

IDL 关联变量是把一个 IDL 数组或结构变量的组织结构映射到数据文件的内容上。文件被看作是这些重复单元的一个数组。 第一个单元的索引号是 0，第二个单元的索引号 1 等等。关联变量不象常规变量那样将整个数据组都存储在内存里。而是当一关联变量被引用时，IDL 仅对需要的部分数据执行相关的输入或输出请求，这部分数据就是要读入内存的。

关联变量的一些优点

关联变量有以下几个优点：

1. 当该变量被用于表达式时，才产生文件的输入和输出动作。不需要单独的读或写命令。
2. 数据集的大小不受内存容量的限制，因为有时它可处理大型的数据集。对于物理存储器来说是太大的数据，通过把此数据分成块就能很容易地处理。
3. 不必提前声明用于映射该数据的数组或结构的数量。
4. 关联变量是效率最高的 I/O 形式。

定义关联变量

定义和使用关联变量，可按通常的方式打开数据文件，然后用 Assoc 命令创建关联变量。例如，打开位于 IDL 主目录下的 coyote 子目录中的 abnorm.dat 文件。

```
IDL> filename = Filepath(Subdir='coyote', 'abnorm.dat')
IDL> OpenR, lun, filename, /Get_Lun
```

这个文件里含有 16 幅图像或 16 帧画面，每幅都是 64*64 个字节型数组。为这些数组创建关联变量：

```
IDL> image = Assoc(lun, BytArr(64,64))
```

Assoc 命令的第一个参数是与 image 变量相关联的文件的逻辑设备号。第二个参数是文件中被重复的单元的描述。

这些文件在重复文件单元前通常有文件头信息，尽管此文件没有。如果这样的话，Assoc 命令的第三个位置参数是给定文件头的大小或文件头在文件中的偏移量。例如，假设 abnorm.dat 文件的前 4096 个字节是文件头信息，并且希望跳过文件头，那么 Assoc 命令可以写成这样：

```
IDL> image2 = Assoc(lun, BytArr(64,64), 4096)
```

注意，现在有两个变量，image 和 image2，与同一个数据文件关联。这在 IDL 中完全合法，事实上，这对于存取重复单元不一致的二进制数据文件是一种好办法。通过改变在文件中的偏移量，可以用关联变量来实现随机读写数据。

显示上面 image 变量里的第五幅图像或画面，键入：

```
IDL> TvScl, image (4)
```

从数据文件里把数据读入一个临时变量，在其被显示后，被 IDL 删除。没有显式的 ReadU 命令，也不需要常规情况下 IDL 处理这幅图像所需要的永久内存。如果想从关联变量中创建一个变量，可按通常的方式建立这个变量。例如，可以键入：

```
IDL> image5 = image (4)
IDL> TV, Rebin (image5, 256,256)
```

数据文件里重复单元的形式没有必要只是一个简单的二维图像数组。它可以是一个复杂的结构。例如，每一个重复单元可以包含 128 个字节的文件头，两个 100 个元素的浮点型矢量和一个 100*100 的整型数组。如果是这样，可以建立一个文件关联变量。键入：

```
OpenR, 10, 'example.dat'
```

```
Info = BytArr (128)
xvector = FltArr (100)
yvector = FlatArr(100)
data = IntArr (100,100)
struct = {header: info, x: xvector, y: yvector, image: data}
repeatingUnit = Assoc (10, struct)
```

因为映射到此数据文件相的变量是一个结构变量，因此在它的引用被删除之前，必须对此结构变量进行临时拷贝。例如，显示文件第三个重复单元的图像部分，可以键入：

```
TempVariable = repearingUnit (2)
TvScl, temPvariable.image
```

按照通常的方式，可以用 Free_Lun 或 Close 命令将关联变量和文件之间的联系关闭。如下：

```
Free_Lun, lun
Close, 10
```

表 9　IDL 能够读写许多常用的数据文件格式。一般情况下通过用 IDL 语言写的库程序或动态连接模块(DLM)来完成的，DLM 在运行时可以添加到 IDL 中。CDF、netCDF 和 HDF 文件格式是著名的科学数据格式，有它们自己的 IDL 接口和库程序

文件格式	读此类文件的 IDL 程序	写此类文件的 IDL 程序
BMP	Read_BMP	Write_BMP
CDF	参考 CDF 库	参考 CDF 库
DICOM	IDLffDICOM 对象	IDLffDICOM 对象
DXF	IDLffDXF 对象	IDLffDXF 对象
GIF	Read_GIF	Write_GIF
HDF	参考 HDF 库	参考 HDF 库
HDF-EOS	参考 HDF 库	参考 HDF 库
Interfile	Read_Interfile	无
JPEG	Read_JPEG	Write_JPEG
netCDF	参考 netCDF 库	参考 netCDF 库
PICT	Read_PICT	Write_PICT
PBM/PPM	Read_PPM	Write_PPM
PNG	Read_PNG	Write_PNG
PostScript	无	PS 或打印设备
Sun Rasterfiles	Read_SRF	Write_SRF
SYLK	Read_SYLK	Write_SYLK
TIFF/GeoTIFF	Read_TIFF	Write_TIFF
WAVE	Read_WAVE	Write_WAVE
X11-bitmap	Read_X11_Bitmap	无
XWD	Read_XWD	无

读写常用文件格式的文件

到目前为止，本章已经介绍了 IDL 读写数据文件的一般方法。这种底层的能力已经可以用 IDL 读写许多数据文件了。但是可能还想知道许多其他文件格式如何读写。这些文件格式中就有 GIF 和 JPEG 文件格式，它们常常在不同办公室之间或全球范围内被用来共享数据。当想在硬拷贝中出版图形输出时，可能想知道也必须知道如何建立 PostScript 文件。

IDL 可以读写许多常用文件格式，这些文件格式已在表 9 中列出。

创建彩色 GIF 文件

GIF 文件常被用来在万维网上发布图形信息。如果想和同事共享图形结果，迟早要读写 GIF 文件。

要看其是如何完成的，可装入一些数据，然后在图形窗口中显示这些数据。键入：

```
IDL> Window, XSize=300, YSize=300
IDL> data= LoadData (1)
IDL> TVLCT,   [100,255,0], [100,255,255], [100,0,0], 0
IDL> Plot, data, /NoData, Color=2, Background=0
IDL> OPlot, data, Color=1
```

写 GIF 文件

下面的命令生成一个大小为 300*300 像素的 GIF 文件。首先，将图形窗口的内容复制到一个 2D 字节型图像变量中。如果在使用 8 位显示器，TVRD 命令可用来实现这个目的。

```
IDL> image = TVRD ()
```

如果正在 16 位或 24 位彩色显示器运行 IDL，那么需要用 TVRD 命令获取一幅 24 位的图像，然后用 Color_Quan 命令将它压缩成一幅带有正确彩色表矢量的 2D 图像，命令如下。只有是在 16 位或 24 位显示器上运行 IDL，才用下面的命令代替上面的命令：

```
IDL> image = TVRD (True=1)
IDL> image = Color_Quan (image24, l, r, g, b)
```

假如已经有一个 2D 字节型数组，就没有必要再拷贝图形窗口。

GIF 文件格式要求将彩色表随图像数据一起存储到 GIF 文件内。如果使用 8 位显示器，那么用 TVCL 命令和 Get 关键字就可以得到由红、绿、蓝三种颜色矢量组成的当前彩色表：

```
IDL> TVCL, r, g, b, /Get
```

假如正在 16 位或 24 位的显示器上运行，而没有必要键入上面的命令。可以在上面 Color_Quan 命令里得到相关图像的彩色表矢量。

色彩矢量必须是 256 个元素。如果在 8 位显示器上运行 IDL，这些矢量可能就没有这么长，但是不必担心。如果在写 GIF 文件时这些色彩矢量不够长的话， IDL 将会加长色彩矢量。如果想充分利用 256 种颜色，可考虑在 Z 图形缓冲区装载彩色表，然后得到颜色矢量，缺省情况下可获得 256 种颜色。参考 125 页的“Z 图形缓冲区中的图形显示技巧”。

最后，用 Write_GIF 命令将图像和颜色矢量写进名为 test.gif 的 GIF 文件：

```
IDL> Write_GIF, ‘test.gif’, image, r, g, b
```

以上就是所有要做的。没有必要获取逻辑设备号或其他东西。所有这些细节都是 IDL 库程序 Write_GIF 命令来处理的。如果对具体如何实现感到好奇的话，可以检查源代码。

如果读者有某个应用程序能打开和读取 GIF 文件，试着读一下刚建立的文件。许多万维网的浏览器都支持读取 GIF 文件。看一下，如果浏览器有一个 Open File 按钮，用它试试看能否读取这个 GIF 文件。

读 GIF 文件

要读刚建立的 GIF 文件，可按下面简单地用 Read_GIF 命令读 GIF 文件里的图像和颜色矢量。

```
IDL> Read_GIF, ‘test.gif’,thisImage, rr, gg, bb
```

清除图形窗口，装载一个灰色级调色板，可以看到将发生什么。键入：

```
IDL> Erase
IDL> LoadCT, 0
```

现在，显示刚从文件里读取的图像，如下：

```
IDL> TV,thisImage
```

有时候在图形窗口里什么都看不到，这是因为 GIF 图像使用的颜色还没有装入。必须装载和 GIF 图像相关的彩色表，以便这个图像能够正确地显示。键入：

```
IDL> TVCT, rr, gg, bb
```

将在显示窗口里看到原始图像。

假设在 16 位或 24 位上的显示器上，必须关掉颜色分解。为了看到正确的颜色，必须在装入颜色表矢量之后重新显示这个图像。

```
IDL> Device, Decomposed=0
IDL> TV, thisImage
```

创建彩色 JPEG 文件

另外一个常被用在万维网上共享图形结果的文件格式是 JPEG 格式。这个 JPEG 格式被称为有损压缩格式。也就是说，当图像数据被压缩后存到文件时，数据的一些信息内容会被丢失，并且不能被恢复。压缩比例，丢失的信息量以及输出图像的质量通常可用质量索引值来设定，质量索引值的范围从 0（丢失许多信息内容的，质量差）到 100（很少或根本就没有信息丢失，质量好）。

通常，质量索引值被设置为 75，即保证一个适当的压缩比，没有丢失很多信息且图像质量损失不大。

一幅彩色 JPEG 图像一般是 24 位的图像。也就是说，这个图像是 3D 字节型数组。在这个数组中，维数之一将为 3。这个维数的位置将决定图像是隔像素扫描（3, m, n），隔行扫描（m, 3, n），还是隔波段扫描（m, n, 3）。在很多情况下，所拥有的图像是 8 位的图像，而不是 24 位的图像，希望将其转变成一幅 JPEG 文件。例如，图像可能是图形窗口的屏幕转储，象上面的 GIF 例子一样。下面是如何从 8 位图像创建隔像素扫描的 24 位图像的实例。

首先，打开一个图形窗口，在色棒旁显示一幅图像。代码中的 TVImage 和 Colorbar 命令在已下载的本书配套程序中。键入：

```
IDL> Window, Size=400,Ysize=300
IDL> LoadCT, 3
IDL> image = LoadData(7)
IDL> TVImage, image, Position=[0.1, 0.1, 0.75, 0.9]
IDL> Colorbar, Position=[0.8, 0.1, 0.86, 0.9], /Right, $
      /Vertical, Division=5, Format='(F5.1)'
```

写 JPEG 文件

接下来，对图形窗口进行拍照，获得图形输出的一幅二维图像，并用这幅图像创建隔像素扫描的 24 位图像。24 位图像是通过用颜色表矢量建立的，实质上是将显示图像的颜色进行分离。IDL 代码如下：

```
IDL> image = TVRD()
 IDL> image = BytArr(3, 400, 300)
 IDL> TVLCT, r, g, b, /Get
 IDL> image3D[0, *,*]=r[image]
```

IDL> image3D1, *,*] =g [image]

IDL> image3D[2, *, *] = b[image]

注意，如果是在 16 位或 24 位显示器上，用一个简单的命令就可以得到 24 位的图像。不需要键入上述的命令，只需键入：

IDL> image3D=TVRD(true=1)

最后，用 Write_JPEG 命令，将此幅 24 位的图像用较好的图像质量和适当的压缩比输出到 JPEG 文件。键入：

IDL> Write_JPEG, 'test.jpg', image3D, true=1, quality=75

如果读者有某个应用程序能打开和读取 JPEG 文件的话，试着读一下刚建立的文件。许多万维网的浏览器都支持读取 JPEG 文件。看一下，如果浏览器有一个 Open File 按钮，用它试试看能否读取这个 JPEG 文件。

读取 JPEG 文件

用 Read_JPEG 命令就可以读取并显示一个 JPEG 文件。例如，如果打算在 8 位显示器上显示 24 位图像，可以用下面这个命令：

IDL> Read_JPEG, 'test,jpg', thisImage, colortable,$

Colors=!D.Table_Size, Dither=1, /Two_Pass_Quantize

关键字 Colors 指明 24 位图像应该量化到多少种颜色，它的值应该是从 8 到 256。关键字 Dither 选择 Floyd-Steinbeig 抖动法，它把颜色量化时的错误分散到旁边的周围的像素中去，从而获得高质量的图像。关键字 Two_Pass_Quantize 将颜色量化分为两步进行处理，同样也可以获得更好的颜色量化效果和更高的图像质量。

显示数据，键入：

IDL> Erase

IDL> Tv, thisImage

所看到的结果可能有些奇怪。这是由这幅图像的颜色量化方法引起的。为了看到输出结果到底是什么，必须装载与图像相关的颜色表。这个颜色表返回到变量 colortable 中。装载颜色表，键入：

IDL> TVLCT,colortable

假设正在 8 位显示器上显示 24 位的图像，应该使用 TV 命令里的关键字 True：

IDL> TV, thisimage, true=1

查询图像文件信息

常用图像文件格式的查询程序已在 IDL5.2 版中提供。这些程序允许在没有真正读取其数据的情况下，就可以查询图像文件。这些程序可以存取随着图像数据文件一起存储在文件里的元数据（关于数据的一些信息）。

下面是新的图像查寻程序列表：

- Query_BMP
- Query_DICOM
- Query_GIF
- Query_JPEG
- Query_PICT
- Query_PNG
- Query_PPM

- Query_SRF
- Query_TIFF

所有这些查询命令都是以同样的方式工作。它们都是返回 0 或 1 的函数，通过返回值确定是否成功地（返回值为 1）读取了图像文件里的元数据。如果它们成功地读取了文件，将保存文件信息的 IDL 结构变量作为输出命令返回给用户。用户通过存取这个结构里面的字段从而获取文件的有关信息。

例如，查询刚创建的 JPEG 文件，将文件的返回信息返回到变量 fileinfo，可以键入：

```
IDL> ok=Query_JPEG('test, jpg', fileinfo)
```

查看返回的是什么信息，键入：

```
IDL> Help, fileinfo,  /Structure
```

可以看到打印输出的信息：

```
**Structure<1364998>,  7 tags,  length=36,  refs=1:
CHANNELS        LONG              3
DIMENSIONS      LONG        Array[2]
HAS_PALETTE     INT               0
IMAGE_INDEX     LONG              0
NUM_IMAGES      LONG              1
PIXEL_TYPE      INT               1
TYPE            STRING      'JPEG'
```

能够看到此文件（Num_Image=1）里有一幅图像，它是字节型数据（Pixel_Type=1），是一个 24 位的图像（Channels=3）。这个图像的大小能够通过打印维数字段可以看到，如下：

```
IDL> Print, fileinfo.dimensions
          400   300
```

其他的图像查询程序在返回结构里含有类似的字段。

第七章 图形硬拷贝输出

本章概要

在使用 IDL 的时候，如何以硬拷贝形式再现屏幕中的图形是最复杂，也是最难理解的问题。然而，这却是大多数献身科学的人的需求，但很少有令人完全满意的方法来和同事共享这些科学结果。

本章将集中在 PostScript 输出，因为 PostScript 是普遍接受的一种输出媒介，大部分使用 IDL 的程序员都能使用 PostScript 打印机。所有关于 PostScript 的内容同样适用于其他输出设备，比如 HP 绘图仪和 PCL 打印机。

具体来说，将学习：

1. 如何选择硬拷贝输出设备
2. 如何配置硬拷贝输出设备
3. 如何将图形输出直接传送到打印机
4. 如何将图形输出传送到一个文件中
5. 如何为硬拷贝输出设备产生图形输出

6. PostScript 输出与显示器的输出有什么不同
7. 如何在 PostScript 页面上定位图形和图像
8. 如何产生能包含在其他文档中的图形输出
9. 如何编写能很容易地转化为硬拷贝输出的程序
10. 如何在 PostScript 种使用颜色

选择图形硬拷贝输出设备

与设置其他图形显示设备一样，在 IDL 中，仍然使用 Set_plot 命令来设置图形硬拷贝输出设备：

```
Set_Plot, 'option'
```

其中的 option 是下列的任何一种，注意 option 总是一个字符串，因此要使用单引号括起来。与 IDL 其他大多数字符串不一样，option 对大小写不敏感。

CGM 输出写入 CGM（计算机图形元文件）格式的文件中，CGM 也是一种独立于设备的文件格式，用于交换图形信息。CGM 文件能以三种形式之一的编码表示：（1）文本，（2）二进制数据，（3）NCAR 二进制数据。

HP 输出以惠普图形语言（HP-GL）格式写入一个文件，它适用于各种各样的 HP-GL 笔式绘图仪。

PCL 输出以惠普打印机控制语言（PCL）格式写入一个文件，它适用于各式激光和喷墨打印机。

PRINTER 输出以任何适合于默认打印机的方式直接传送到该打印机。

PS 输出以 PostScript 格式写入一个文件中。

Z 输出被写入 Z 图形缓冲区。

在打印完毕后，应再次使用 Set_plot 命令将输出设备改回图形显示设备的类型，以下是一些常用的显示设备：

WIN 使用微软 Windows 或 NT 操作系统的个人计算机。

MAC 使用 MacOS 操作系统的计算机

X 使用 X Window 系统的计算机。

只有一种设备能成为当前图形设备，可以通过检查!D.Name 系统变量来确定当前的设备是哪种，如下：

```
IDL>Print,!D.Name
```

注意，当设定设备名时，设备名对大小写不敏感，但当在代码中使用该名字时，就不一定不敏感了。存储在!D.Name 系统变量中的图形设备名是以大写字母形式存储的。这在下面的字符串比较语句中尤为重要：

```
IDL>IF !D.Name EQ 'PS' THEN Print,'Using PostScript…'
```

配置图形硬拷贝输出设备

一旦选定了图形输出设备，所有设备具体的配置参数都用 Device 命令通过关键字来控制。Device 命令可用的关键字主要取决于当前的设备。但打印设备（总是和默认的打印机相连）的设置也可以使用 Dialog_PrinterSetup 命令来设置(详见 201 页的“配置和使用打印设备”)。

测定当前的设备配置

使用 Help 命令，可以知道当前硬拷贝输出设备所设定的配置参数，如下：

```
IDL>Help,/Device
```

将能看到一系列的有关当前图形设备的当前设置参数及其参数值。这些信息可以用来配置设备。关于设备可用的颜色数，IDL 使用的是哪种图形函数以及当前选择的硬件字体等等信息，都取决于所设定的当前设备是何种设备。

注意，这些信息的显示随着每个硬拷贝输出选项的不同而不同。例如，键入下面这些命令来看 PostScript 输出设备缺省配置如何：

```
IDL>thisDevice=!D,Name
IDL>Set_Plot,'PS'
IDL>Help, /Device
IDL>Set_Plot, thisDevice
```

以下为 Help 命令的显示结果(在 Windows NT 机器上)：

```
Available Graphics Devices: CGM HP NULL PCL PRINTER PS WIN Z
Current graphics device: PS
    File: <none>
    Mode: Portrait, Non-Encapsulated, EPSI Preview Disabled, Color Disabled
    Offset (X,Y): (1.905,12.7) cm., (0.75,5) in.
    Size (X,Y): (17.78,12.7) cm., (7,5) in.
    Scale Factor: 1
    Font Size: 12
    Font Encoding: AdobeStandard
    Font: Helvetica TrueType Font: <default>
    # bits per image pixel: 4
    Font Mapping:
        (!3) Helvetica                  (!4) Helvetica-Bold
        (!5) Helvetica-Narrow           (!6) Helvetica-Narrow-BoldOblique
        (!7) Times-Roman                (!8) Times-BoldItalic
        (!9) Symbol                     (!10) ZapfDingbats
        (!11) Courier                   (!12) Courier-Oblique
        (!13) Palatino-Roman            (!14) Palatino-Italic
        (!15) Palatino-Bold             (!16) Palatino-BoldItalic
        (!17) AvantGarde-Book           (!18) NewCenturySchlbk-Roman
        (!19) NewCenturySchlbk-Bold     (!20) <Undefined-User-Font>
```

常用的 Device 命令关键字

大部分输出设备能允许以下关键字被用于 Device 命令（Z 设备例外）。以下为想要知道的关键字。对于某个特定输出设备所使用的其他关键字可以查阅 IDL 在线文档资料。例如，PS 设备能接受将近 50 种不同的关键字。

Close_Document 这个关键字在刷新了输出缓冲区后关闭图形文档。它被用于从打印机中排出打印页（使用 Printer 设备时）。

Close_File 这个关键在刷新了缓冲区之后关闭该图形输出文件。

Filename 图形输出设备如果是将输出写入一个文件时有一个缺省文件名。如

	果没有指定文件名时就使用该文件。一般情况下，该文件名为 idl.option，option 是所选择的硬拷贝输出设备类型。但也可以使用此关键字指定一个文件名来更改它。例如： IDL>Device,Filename='surface.eps'
Inches	如果设置了这个关键字，那么关键字 XSize、YSize、XOffSet 和 YoffSet 及其设置都被认为是以英寸为单位而不是以缺省的厘米为单位来给定的。 IDL>Device,XSize=4.0,/Inches 若要回到以厘米为单位来设定尺寸和偏移量，用： IDL>Device,Inches=0
Landscape	该关键字表示在纸的横向上输出。
Portrait	该关键字表示在纸的纵向上输出。这是缺省值。
XOffSet	该关键字确定输出的显示窗口的左下角，在纸上的 X 方向位置上（在纵向模式下）。关于横向模式下的位置详见 199 页的“在横向输出模式中计算 PostScript 的偏移量”。
XSize	该关键字确定输出显示窗口在纸上的宽度。
YOffSet	该关键字确定输出的显示窗口的左下角，在纸上的 Y 方向位置上。关于横向模式下的位置详见 199 页的“在横向输出模式中计算 PostScript 的偏移量”。
YSize	该关键字确定输出显示窗口在纸上的高度。 IDL>Device,XSize=4.0,YSize=7.0,/Inches

注意一旦在图形输出设备上设定了某个关键字的值，该参数将一直有效，直到显式地更改它或退出 IDL。

XSize,YSize,XOffSet 和 YOffSet 这些关键字一般是用于在输出页面上定位“图形窗口”。IDL 命令使用图形窗口的方式和图形输出使用位于显示设备上的图形窗口的方式完全一样。详细细节参考 184 页的“显示设备和 PostScript 设备之间的相同点”。

创建 PostScript 文件

当前的图形设备总是储存在系统变量!D.Name 中，所有图形命令将指向它。因此,实际上，尤其在 IDL 的程序中，选择一个硬拷贝输出设备的代码一般都类似于下面这个例子，在这个例子中，先创建数据然后送到名为 output.ps 的一个 PostScript 文件中。注意，设备如何选择，如何配置，如何在图形命令写入文件后关闭。

如果在关闭被打印机处理的文档时失败，它就不能将该页从打印机中排出，因为该文档缺乏 PostScript 的 Showpage 命令。该命令用于将该页排出。如果使用的是一台慢速打印机，这点就可不必关心。可以在打印机工作的时候，站起来喝一杯咖啡。

```
IDL> data= LoadData(1)
IDL> thisDeivce=!D.Name
IDL> Set_Plot, ‘PS’
IDL> Device, Filename=’output.ps’, XSize=4, Ysize=4,$
      /Inches, Xoffset=2.25, Yoffset=3.5
IDL> Plot, data
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

将图形送到硬拷贝设备中

对于制作硬拷贝输出的一般概念是，用 Set_plot 命令选择硬拷贝设备,，用 Device 命令并按要求设置该设备（或者在使用打印机的情况下，有时可用 Dialog_PrinterSetup），执行和用于输出到显示设备相同的 IDl 命令。然而这些命令将输出到文件或打印机，而不是输出到显示设备上。当完成调用 IDL 的图形命令后，关闭输出文件或打印任务，并以便把文件传送指向所选择的打印机或绘图仪。如果是使用打印机，路由传送是自动完成的。（详见 181 页的“打印 PostScript 文件”的关于将 IDL 产生的 PostScript 文件传送到打印机的章节）。

例如，要创建一个 PostScript 文件，可以键入如下命令：

```
IDL> thisDevice=!D.Name
IDL> Set_Plot, ‘PS’
IDL> Device, Filename=’plot.ps’, XSize=4, Ysize=4, $
        /Inches, Xoffset=2.25, Yoffset=3.5
IDL> Plot, LoadData(1)
IDL> Device, /Close_File
```

如果在 UNIX 机器上产生这个 PostScript 文件，可以调用一个简单的 lpr 命令来将文件导向打印机（或者，机器上任何的等效命令）。例如，从 IDL 中，可以键入这条命令：

```
IDL> Spawn, ‘lpr plot.ps’
```

广义上讲，在显示设备上的输出和 PostScript 文件中的输出并非完全不同。（麻烦在于其细节）。也就是说，Plot，Surface，Contour 以及其他 IDL 图形命令无论是在显示设备上还是在 PostScript 文件中操作几乎一样。

一方面，它们在如何进入文件这方面相似。例如，如果调用 Plot 或 Surface 命令，并且显示设备为当前工作的图形设备，当前窗口的内容将被擦除，建立一个新的图形显示。在 PostScript 设备为当前图形设备时，类似的情况也会发生。每个命令将擦除显示设备上的窗口，启动一个新的 PostScript 输出页面。

相应地，每个图形命令，例如 Oplot 或 XoutS，将在当前显示窗口中被执行，修改当前 PostScript 输出页面。例如，要显示一幅用 XYOutS 命令创建的带标题的图和一幅曲面图，可按如下键入：

```
IDL> Plot, LoadData(1), Position=[0.1,0.1,0.9,0.8]
IDL> XYOutS,0.5,0.9,’Simple Plot’,Align=05,/Normal
IDL> Surface,Dist(41)
```

如果这些命令被用到 PostScript 设备上而不是显示设备，将得到有两页输出的一个文件。要将这些命令的结果送到一个 PostScript 文件，可以按如下键入命令：

```
IDL> thisDevice=!D.Name
IDL> Set_Plot, ‘PS’
IDL> Device, XSize=3, Ysize=3, /Inches
IDL> Plot, LoadData(1), Position=[0.1,0.1,0.9.0.8]
IDL> XYOuts,0.5,0.9,’Simple Plot’, Align=0.5,/Normal
IDL> Surface, Dist(41)
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

注意，无论是在显示窗口还是在 PostScript 窗口，都可以用关键字 Position 和 Normal 来放置图形显示单元。这是一种用于创建在显示窗口的输出和在 PostScript 窗口的输出相同的好方法。关于此一会将学到更多的东西。

有个技巧可以用于强制让 PostScript 文件前进一页，这就是使用 Erase 命令。如果要将几幅图像放在一个 PostScript 文件中，这也很方便。正常情况下，TV 或 TVScl 命令不擦

除前面窗口的内容，所以，多个 TV 命令将简单地在前一幅图像上面叠放下一幅。Erase 命令可以将多幅图像放在同一文件中的不同页面。如下：

```
IDL> thisDevice=!D.Name
IDL> Set_Plot, 'PS'
IDL> TV, LoadData(5)
IDL> Erase
IDL> TV, LoadData(7)
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

打印 PostScript 文件

从 UNIX 机器上打印 PostScript 文件的最简单方法是调用 lpr 命令（或机器上其他任何的等效命令）。但很可能犯一个的错误，企图在 IDL 中用类似如下的命令来打印 PostScript 文件：

```
Set_Plot, 'PS'
Device, Filename='new_plot.ps'
Plot, my_data, Title='Awful Nice Plot'
Spawn, 'lpr new_plot.ps'
```

不幸的是，这样做行不通。原因是在试图用 UNIX 命令 lpx 打印 PostScript 文件之前，忘了关闭它。正确的顺序是：

```
Set_Plot, 'PS'
Device, Filename='new_plot.ps'
Plot, my_data, Title='Awful Nice Plot'
Device, /Close_File
Spawn, 'lpr new_plot.ps'
```

上面第一组命令失败的原因是，PostScript 文件需要一个 PostScript Showpage 命令来从打印机中排出该页面。然而，只有使用 Close_File 命令或退出 IDL，Showpage 命令才被插入到 PostScript 中。

在个人电脑上，在 IDL 中打印用 PostScript 设备产生的 PostScript 文件则有一点难度。事实上，大部分人都不介意，因为现在已有无数适合这种机器的工具来打印 PostScript 文件。

在运行 MacOS 系统的计算机上打印 PostScript 文件

在一个 Macintosh 机器上或是运行 MacOS 操作系统的计算机上，将一个 PostScript 文件送到打印机的最好方法是从 Bare Bones 软件公司下载一个免费的小程序 Drop・PS。
可以从一般的 Macintosh 匿名的 ftp 软件网站下载。Macintosh 打印机总是附带一些打印工具，也能直接将 PostScript 文件传送到打印机上。

在 Windows 计算机上打印 PostScript 文件

在 Windows 95 和 Windows NT 操作系统上，将一个 PostScript 文件送到打印机的最好方法是下载一个免费的软件 GhostView 或 GhostScript。这些工具是 PostScript 文件浏览器，可以在打印前预览输出。GhostView 有一个漂亮的图形化界面，可以把 PostScript 文件直接

传入打印机。详细资料可用通过 WWW 浏览器访问下列网址：

http://www.cs.wise.edu/~ghost/

GhostScript 也有运行于 UNIX 和 MacOS 操作系统上的版本。

另外，还有一个很有意思的工具叫 PrintFile。这个小工具实现了拖放功能，而且能将 PostScript 文件传送到本地或网络打印机。它甚至还能打印封装的 PostScript 文件。可以在下列网址找到这个程序：

http://heml.passagen.se/ptlerup/prfile.html

生成封装的 PostScript 文件输出

要生成能包含在其他文件（比如杂志文章和书等）中的 PostScript 输出，在输出图形到 PostScript 文件前，必须选择封装选项：

```
Set_Plot, 'PS'
Device, /Encapsulated
```

IDL 的封装 PostScript 文件可以成功地放到 LaTeX，Microsoft Word，FrameMaker 和其他一些文字处理文档中。

注意，PostScript 封装文件不能在 PostScript 打印机上通过自己来进行打印，尽管打印机在不停地动。因为，封装文件缺少 PostScript 的 Showpage 命令，此命令用于从打印机上排出该页面。这些文件必须被包含或‘封装’在其他文件中进行打印。

同时也要注意，当将封装文件输入另一个文件中时，可能无法看到该图形，直到打印出来。除非有一个 PostScript 预览器或者用下面将要描述的 Preview 关键字。

关闭封装 PostScript 文件，可以将 Encapsulated 设置为 0，如下；

```
Device, Encapsulated=0
```

封装 PostScript 图形的预览

在正常情况下，封装的 PostScript 文件不能在包含它的文档中显示。也就是说，拥有此输入文件的方框总是白色或灰色。然而，PostScript 图形在整个文档被送到 PostScript 打印机时，可以正确的打印。

如果想让文档中的该图形能看见，必须设定 Preview 关键字。此关键字能让 PostScript 驱动程序包含一幅该图形的位图和 PostScript 描述。这幅位图将显示在文档中的方框内，在文档被打印时就使用 PostScript 描述。

```
Device, /Encapsulated, /Preview
```

注意并非所有的文字处理程序都能显示位图预览图像。例如，预览图像在 Microsoft Word 5.1 或 Macintosh 机器上的 FrameMaker 4.0 上不能很好地显示。但在 Windows NT 机器上的 FrameMaker 5.1 就能很好地显示。用自己的文字处理软件试一下，查看显示如何。

要将预览关闭，将关键字 Preview 设为 0 即可。如下：

```
Device, Preview=0
```

注意，在 IDL5.2 中预览功能在 Macintosh 和 Windows 中有很大的提高。把 Preview 关键字值设为 2，便可以创建一个带有一幅 TIFF 预览图像的封装 PostScript 交换文件。文件的 PostScript 部分用于在 PostScript 打印机上打印。

生成彩色的 PostScript 输出

IDL 中支持彩色的 PostScript 输出。要想有彩色的输出，在 PostScript 设备上使用 Color 关键字：

```
Set_Plot, 'PS'
Device, Color=1
```

颜色关键字的设置自动地将当前彩色表复制到 PostScript 文件中。（类似于下面 Set_Plot 命令中的 Copy 关键字。）注意 PostScript 设备几乎总是支持 256 色，通常多于在显示设备上使用的颜色数。这将影响输出。详见 191 页的“问题：PostScript 设备的颜色数目多于显示设备”。

另一个自动装载彩色表的方法是在将图形设备设置为 PostScript 时，使用带 Copy 关键字的 Set_Plot 命令：

```
IDL> Set_Plot, 'PS', /Copy
```

这个命令在文件被打开的第一次操作时，自动地将当前的颜色矢量复制到 PostScript 文件中。注意，是显示彩色表被拷贝到 PostScript 文件中。通常这些彩色表的颜色数目和 PostScript 文件的彩色表的数目不同。详见 189 页的“问题： PostScript 设备使用背景颜色和绘图颜色时的不同”。

一旦设定 PostScript 设备为当前图形设备，可以用归一化的彩色表装载命令来装载彩色表。例如，可以键入如下命令：

```
IDL> LoadCT, 5, Ncolors=200
IDL> TVLCT, [70,255],[70,255],[70,0],200
```

要将颜色选项关闭，可将 Color 关键字设为 0，如下：

```
Device, Color=0
```

PostScript 中的彩色图像与灰度图像

缺省情况下，PostScript 设备为每一图像像素保存 4 位的信息。这对 16 色或灰度级是足够了。如果想在 PostScript 输出中能有更多的颜色，Device 命令的 Bits_Per Pixel 关键字能设置到 8 位。例如，要输出一幅使用了全部 256 色的图像，可以如下设置设备：

```
IDL> image=LoadData(7)
IDL> thisDevice=!D.Name
IDL> Set_Plot, 'PS'
IDL> Device, Color=1, Bits_Per_Pixel=8
IDL> TVSCL, image
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

真彩图像

读者的 PostScript 设备也许能支持 24 色或真彩图像。真彩图像是一个 3D 的数组，其中有一维是 3。例如，一幅 m*n 的真彩图像可以是隔像素扫描(3,m,n)，也可以是隔行扫描(m,3,n)，还可以是隔波段扫描(m,n,3)。

真彩图像可以以显示在显示器上的相同方式来显示在 PostScript 中。就是说，在 TV 或 TVScl 命令中使用 True 关键字，以表明真彩图像是如何扫描的。确保将 Bits_Per_pixel 关键字的值设为 8。例如，一幅隔像素扫描的真彩图像可以送到一个真彩 PostScript 设备上：

```
IDL> image3d=LoadData(16)
IDL> thisDevice=!D.Name
IDL> Set_Plot, 'PS'
IDL> Device, Color=1, Bits_Per_Pixel=8
IDL> TV, image3d, True=1
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

在继续阅读本章的内容前，确保当前的图形输出设备是显示设备。若不能肯定，使用以下命令：

```
IDL> Set_Plot, 'X' ;或  'Win'  或  'Mac'
```

来进行确认。

在 PostScript 设备上创建高质量的输出

创建看上去类似于显示设备输出的高质量硬拷贝输出的秘诀在于要理解显示设备和输出设备之间的共同点与不同点，比如，考虑显示设备和 PostScript 设备之间的相同点。

显示设备和 PostScript 设备之间的相同点

最明显的相同点就是在每种设备上为显示图形而创建的图形窗口，尽管在每种设备上使用的方法不同。在一般的显示设备上可能会以下面的代码来创建图形窗口：

```
Window, XSize=300, YSize=400, XPos=100, YPos=200
```

创建了一个 X 方向 300 像素和 Y 方向 400 像素的图形窗口。此窗口的左下角位于显示器（比如，显示器的分辨率为 1024*768）的（100，200）处。

而在 PostScript 设备上创建一个图形窗口的操作是类似的。区别在于不是使用 Windows 命令来创建。（当 PostScript 设备为当前图形输出设备时，Window 命令是个无效的命令，这点在写程序的时候必须记住。）而是用 Device 命令来告诉 PostScript 设备要创建的窗口的大小。例如：

```
Device, XSize=3, YSize=4, XOffset=1, YOffset=2, /Inches
```

可以这么想，PostScript 页面类似于整个显示设备，而用 Device 命令创建的区域类似于显示设备上的图形窗口。换句话说，调用上面的命令的含义是在显示设备上或在将要输出图形的页面上创建一个位置。

IDL 使用归一化原则把图形放入任意一个窗口。那么，:当键入以下这样一个命令后会怎么样？

```
Plot, FindGen(11)
```

IDL 使用归一化原则来将图形定位于窗口中。在此例中，IDL 用设备坐标来计算字符的大小，并通过它来决定图形缺省的边缘。该图形在这个边缘的基础上被置在窗口内，一般是正好填满整个图形窗口。

但是在显示设备上的图形与在 PostScript 设备上的输出图形是否一样呢？尽管会很相似，很可能不是。原因在于在显示设备上对图形的解释方式与 PostScript 设备不一样。

显示设备与 PostScript 设备之间的不同点

显示设备与 PostScript 设备之间有几个不同点，这对于要想在 PostScript 设备上输出与显示设备上几乎一样的图形是非常关键的。有一两个例外的情况，它们的差别不是很大，或者这些差别看上去不重要。但是以笔者个人的经验来看，若没有理解这些差异，要想生成高质量的硬拷贝输出将会花费很大的精力。

问题：PostScript 窗口可能会有不同的纵横比例

首先，一个相对较小的不同点，在显示设备上创建的图形窗口与在 PostScript 页面上创建的图形窗口的纵横比例可能不同。这并不奇怪，因为两个窗口的创建方法不同：显示设备上是使用 Windows 命令，PostScript 页面是使用 Device 命令。

事实上，大部分使用者在 IDL 中显示图形时不用 Windows 命令。而是简单地使用 Plot 或 Surface 命令以及它们打开的一个窗口。缺省窗口的大小随机器不同而不同，而且可由用户设置。在工作站上，缺省的窗口大小为 640*512 像素大小。在 PC 上，缺省的窗口大小通常为显示设备尺寸的四分之一。在 PostScript 设备上，缺省的窗口大小为 7*5 英寸。三种情况下的纵横比（Y/X）为 0.800，0.750 和 0.714。

很清楚，同样 Plot 命令的输出在三个窗口中不同，因为三个窗口纵横比不一样，而且，IDL 将填满 Plot 命令所获得的窗口。

解决方法：让图形窗口的纵横比保持不变

所以，创造完全一样的输出的第一条原则就是确保显示窗口和 PostScript 窗口具有相同的纵横比。这很容易做到。只要计算当前显示窗口的纵横比，并将 PostScript 窗口设为一致即可。例如（假设在显示器上已经有一个打开的窗口），可以键入：

```
IDL> aspectRatio=Float(!D.Y_Vsize)/D.X_Vsize
IDL> thisDevice=!D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=5, Ysize=5*aspectRatio, /Inches
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

这样，在显示窗口与 PostScript 输出中看上去相同的可能性就比以前大多了。

笔者喜欢用 PSWindow 程序来创建一个与当前显示窗口有相同纵横比的 PostScript 图形窗口。（pswindow.pro 在已经下载的本书配套文件之中）。该程序返回在 PostScript 页面上创建所能创建的最大图形窗口时所必须的尺寸和偏移量（默认以英寸为单位），所建立的窗口与当前图形窗口具有相同的纵横比。返回值用来设置 Device 命令的相应关键字，通常是它的_Extra 关键字。（关于_Extra 关键字详细信息见 240 页的“一个关键字继承的问题”）

查看它是如何使用的，首先打开一个图形窗口，并显示一幅线画图。

```
IDL> Window, XSize=400, YSize=300     ; Aspect Ratio=0.75
IDL> curve=LoadData(1)
IDL> Plot, Curve
```

现在用相同的纵横比创建一个 PostScript 窗口，并在上面画图。键入：

```
IDL> rightSize=PSWindow()
IDL> thisDevice=!D.Name
```

```
IDL> Set_Plot, 'PS'
IDL> Device, _Extra=rightSize, /Inches, File='test.ps'
IDL> Plot,curve
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

如果有 PostScript 打印机或 PostScript 预览软件，将此文件传给它。比较输出内容和显示窗口中的内容。一样吗？

不同？但相似吧？请继续！

问题：PostScript 设备有更高的显示分辨率

在默认状态下，IDL 在字符尺寸的基础上计算出图形的边缘，从而决定将图形的坐标轴放在图形窗口的什么地方。但用于计算边缘的字符尺寸在 PostScript 设备上和在显示设备上不一样。

原因是 PostScript 设备有一个比显示设备更精细的分辨率。可以检查 IDL 系统变量!D.X_PX_CM（决定每厘米的像素个数）和!D.X__CH_SIZE(以设备坐标决定缺省字符 X 方向上的尺寸)，查看有何不同。输入：

```
IDL> thisDevice=!D.Name
IDL> Print, !D.X_PX_CM, !D.X_CH_SIZE
IDL> Set_Plot, 'PS'
IDL> Print, !D.X_PX_CM, !D.X_CH_SIZE
IDL> Set_Plot, thisDevice
```

例如，在 Macintosh 计算机上的数据为：

```
Mac:     28.35        6
PS:       1000.00     222
```

换句话说，在显示屏上的一个像素，在 PostScript 上就有大约 35 个像素。而且，两种设备的字符尺寸相对于分辨率的比率也不一样的。Macintosh 是:0.212 和 PostScript 是 0.222。

马上就明白了，在 IDL 中用设备坐标或像素坐标来定位任何一个图形是个不好的方法，除非将分辨率因素考虑进去。例如，假设想在显示器上围绕一个 X 方向像素从 100 到 200，Y 方向像素从 150 到 250 的图像周围画一个方框，可能会这样画：

```
xBox=[100,100,200,200,100]
yBox=[150,250,250,150,150]
PlotS, xBox, yBox, /Device
```

在一个 400*400 像素的显示窗口上，方框与窗口的大小之比为 1:16。在 10*10 厘米的 PostScript 窗口上，方框与窗口的大小之比将为 1:10,000！这是相当小的盒子，肯定不是所要的。

解决方法：不用设备坐标来定位图形

创建与实际一致的图形输出的第二条原则是，确保在输出窗口中使用数据或归一化坐标而不是用设备坐标来在图形窗口中定位图形。

例如，如果按下面这样定义上面的方框，它将在显示窗口和 PostScript 窗口两者中包含同样的相对区域：

```
xBox=[0.250, 00,200,200,100]
yBox=[150,250,250,150,150]
PlotS, xBox, yBox, /Device
```

字符尺寸与分辨率的比率影响输出的另外一种方式是，在图形窗口中放置图形输出的方法。回想一下，缺省情况下，IDL 使用边缘来在窗口中定位图形以及基于字符尺寸计算边缘。如果字符尺寸在显示设备上和在 PostScript 输出中不同，这将稍微影响图形输出。

但可以用 Position 关键字来定位图形从而弥补这一点。 无论是使用显示窗口还是 PostScript 窗口，都可用归一化坐标来将坐标轴定位在准确的地方。（详见 50 页的“设置图形位置”。）

上面简单的图可以使用以下命令以相同的方式放到任何一个窗口中。

```
Plot, Load Data(1), Position=[0.1, 0.1,0.95,0.95]
```

问题：PostScript 设备能使用不同的显示字体

缺省时，IDL 使用 Hershey 字体输出图形。Hershey 字符集是一种典型的矢量字体。这些字体是用一些矢量描述的，显示时象被着色一样。使用矢量字体有两大优点：它们能很容易地在 3D 空间中缩放和旋转，而且与设备无关。矢量字体最大的缺点是：在象 PostScript 打印机这类高分辨率输出设备上，其质量不如真实的 PostScript 字体。

正因为如此，许多人趋向于用 PostScript 字体来输出 PostScript 输出图形。这就产生了在显示屏上与在 PostScript 输出中稍有差异，因为在 PostScript 打印机上没有与 Hershey 矢量字体一一对应的字体。PostScript 字体必须代替 Hershey 字体。这导致字体字符的大小不同，还可能导致文字排列的问题。

解决方案：仔细设计和定位文字

这个解决方法就是要注意如何设计文本的输出。例如，如果可能的话，标题应以点为中心或者在 XYOutS 命令中用 Alignment 关键字来特意布置在某些点上。可以从图 65 中看出 Hershey 和真实的 PostScript 字体的区别。

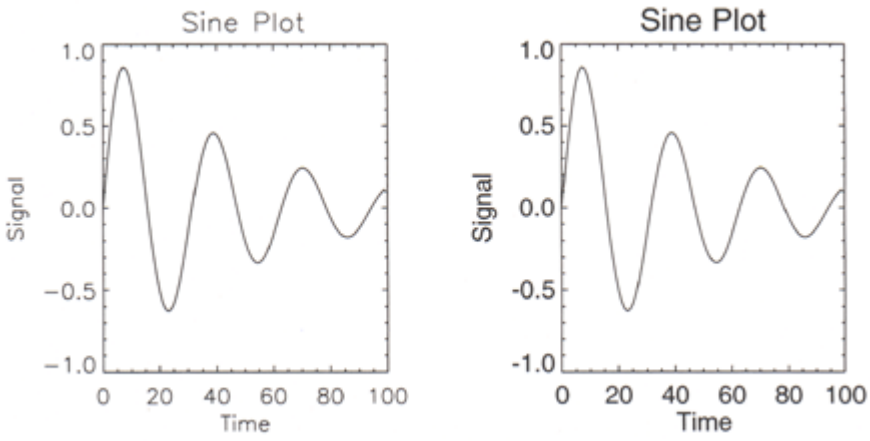


图 65　左边的图是用 Hershey Simplex Roman 字体创建的。右边的图是用 PostScript Helvetica 字体创建的。两幅图看上去类似，但不同

要选择一种真实的 PostScript 字体，可将系统变量!P.Font 或 Font 关键字设为 0。若没有其他信息的话，IDL 将根据表 10 的映射关系将 Hershey 字体映射到 PostScript 字体。可以

使用以下命令随时查询 IDL 中的这种映射关系：

```
Set_Plot, 'PS'
Help, /Device
```

注意，下表中的正常的默认字体 Simplex Roman 被映射到 Helvetica。在 PostScript 的输出中，在同样的字体尺寸下，Helvetica 的字体稍大于 Simplex Roman。也就是说，应在图形输出时，使用代替字体定位文本必须小心，要用一种合理的从显示到硬拷贝的过度方式。

实际中，这就意味着要将调节输出文本左右或中心对称的坐标归一化。例如，下面的代码是在图上生成一个图例：

```
Plots, [0.2,0.3], [0.7,0.7], /Normal
Plots, [0.2,0.3],[0.6,0.6], LineStyle=3, /Normal
XYOutS,0.32,0.7, 'Normal bias', /Normal, Alignment=0.0
XYOutS,0.32,0.6, 'No bias', /Normal, Alignment=0.0
```

这个代码用来在显示窗口和 PostScript 窗口两者相同的相对位置处定为文本。

注意，能用 Device 命令改变字体映射关系。如：字体!4 一般被映射为 Helvetica-Bold PostScript 字体。若要改为 Palatino-Bold-Italic 字体，应键入：

```
Device, /Palatino, /Bold, /Italic, Font_Index=4
```

将 Palatino-Bold-Italic 字体用于上述的图例中，可键入。

```
Plots, [0.2,0.3], [0.7,0.7], /Normal, Font=0
Plots, [0.2,0.3],[0.6,0.6], LineStyle=3, /Normal
XYOutS,0.32,0.7, '!4Normal bias!X', /Normal, Alignment=0.0
XYOutS,0.32,0.6, '!4No bias!X', /Normal, Alignment=0.0
```

在上面的字符串中的!X 将字体返回为使用!4 之前的字体。关于 XYOutS 命令的用法详见 55 页的“在图形显示添加文本”。

表 10　缺省的 Hershey 字体与 PostScript 字体之间的映射关系。Hershey 字体一般用于显示设备上。PostScript 字体一般用于 PostScript 输出

序号	Hershey 字体	PostScript 字体
!3	Simplex Roman	Helvetica
!4	Simplex Greek	Helvetica-Bold
!5	Duplex Roman	Helvetica-Narrow
!6	Complex Roman	Helvetica-Narrow-BoldOblique
!7	Complex Greek	Times-Roman
!8	Complex Italic	Times-BoldItalic
!9	Math and Special	Symbol
!10	Special	ZapfDingbats
!11	Gothic English	Courier
!12	Simplex Script	Courier-Oblique
!13	Complex Script	Palatino-Roman
!14	Gothic Italian	Palatino-Italic
!15	Gothic German	Palatino-Bold
!16	Cyrillic	Palatino-BoldItalic
!17	Triplex Roman	AvantGarde-Book
!18	Triple Italic	NewCenturySchlbk-Roman
!19	Undefined	NewCenturySchlbk-Bold
!20	Miscellaneous	Undefined

问题： PostScript 设备使用背景颜色和绘图颜色时的不同

另外一个显示设备与 PostScript 设备的不同是 PostScript 处理颜色的方式不同。例如，PostScript 设备把正常的背景颜色和绘图颜色反过来。在显示器上输出时，背景色是由!P.Background 系统变量控制的。当启动 IDL 时，该变量被设为 0，意味着 IDL 使用当前彩色表中的第 0 号颜色作为背景颜色。IDL 带的大多数的彩色表的 0 号色都是黑色。黑色的背景下，当把图形输入 PostScript 设备时，将耗用很多色粉。因此，当将 PostScript 设备设为当前图形设备时，IDL 自动将此变量的值改为 255。

但是 IDL 的 PostScript 输出处理比这更隐蔽，因为，无论用什么颜色索引值替换 255，都只能获得一份白色背景的 PostScript 输出。也就是说，PostScript 实际上除了白色，其他任何背景颜色都被忽略。即使将背景颜色设为不是 255 的值，上面这个规则仍然起作用。例如，可以尝试用以下命令在获得一个具有碳灰色背景的绿色图形：

```
TVLCT,[0,70],[255,70],[0,70],100
Plot, LoadData(1), Color=100, Background=101
```

然而，这两个命令在显示屏上输出正确，但在 PostScript 输出中是一白色背景的绿色图形。

解决方法：理解 PostScript 如何处理背景颜色和绘图颜色

在 PostScript 输出中得到其他颜色的背景的惟一方法是，把某种特定的颜色作为背景图来着色处理。例如，可能用 PolyFill 命令，如下：

```
IDL> thisDevice=!D.Name
IDL> Set_Plot, ‘PS’
IDL> Device, /Color, Bits_Per_Pixel=8, File=’example.ps’
IDL> TVLCT, [0,70],[255,70],[0,70],100
IDL> PolyFill,[0,1,1,0,0], [0,0,1,1,0], /Normal, Color=101
IDL> Plot, LoadData(1), Color=100, /NoErase
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

绘图颜色同样可以被 PostScript 设备改变。系统变量!P.Color 通常是设为彩色表中的最后一种颜色。在显示设备上等于!D.Table_Size-1。例如，运行 IDL 时有 220 种颜色，!P.Color 的值就为 219。而如果选择了 PostScript 设备，!P.Color 的值总是 0。

这意味着，如果用 IDL 缺省时装载的灰度彩色表在显示设备上画图，将看到黑色背景下的白色图形。如果在 PostScript 文件中做同样的事情，将看到白色背景下的黑色图形。如图 66 所示。

绘图色总是受到 PostScript 设备的尊重，这点不象背景色。所以，能够用!P.Color 指定一个除 0 以外的颜色，就可以在显示器上和 PostScript 中用该颜色着色图形。也可以使用 Color 关键字和一个给定的值来画图，这个颜色将同时受到显示设备和 PostScript 的尊重。例如，下面两行命令总是绘出红色图形，不管是显示器上还是在 PostScript 输出中：

```
TVLCT, 255,0,0, 100
Plot, LoadData(11), Color=100
```

注意，在灰度打印机上，颜色将由经过抖动处理的线条代替，至于是显示点还是虚线，取决于打印机的分辨率。实际中，这意味着如果要在灰度打印机上输出，最好确保绘图颜色是黑色。

显示设备上的颜色　　PostScript 设备上的颜色

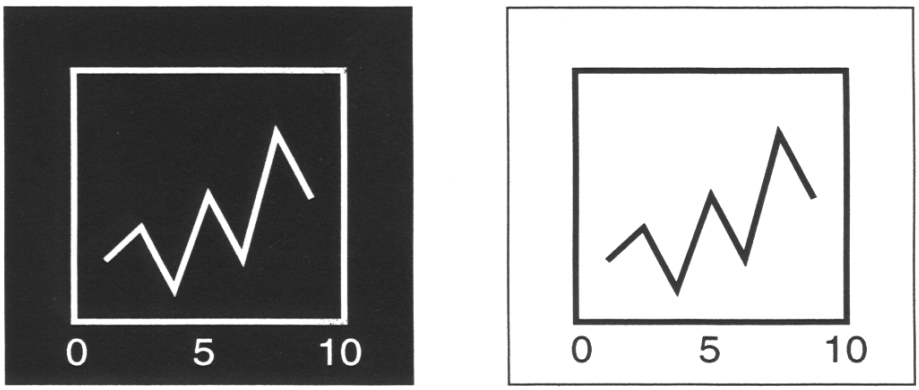


图 66　PostScript 将背景色和绘图色反色。因此，在显示上是黑底白图，但在 PostScript 上是白底黑图。

问题：PostScript 设备的颜色数目多于显示设备

PostScript 设备与显示设备的另外一个不同的方面是它们所使用的颜色数。PostScript 通常至少能显示 256 种颜色。一般情况下，用户在显示设备上使用的颜色少于 256 种。可以通过打开一个图形窗口并打印系统变量!.D.Table_Size 的值来得知现在 IDL 所使用的颜色数量，如下：

```
IDL> Window
IDL> Print, !D.Table_Size
```

通常，这个颜色数量在 200 到 240 之间。如果在一个 8 位显示卡的 PC 机上运行 IDL，这个值总是少于 256 色。在具有 8 位显示卡的其他计算机上，这个值将会更少，除非有自己的颜色表。如果在显示数据时不注意，这个差别就会影响输出。

例如,当在 IDL 运行中使用 200 种颜色，而且想用一个灰度彩色表来显示图像。可以按如下装入彩色表：

```
IDL> LoadCT, 0
```

这个命令在彩色表文件中查找组成灰度彩色表的红、绿、蓝颜色矢量，并重采样这些矢量，以便它们可以代表 IDL 运行时所使用的颜色。在此例子中，被载入实际的色表中的矢量为 200 元素。要显示图像，可按如下这样：

```
IDL> image=LoadData(7)
IDL> TVScl, image
```

它看上去像图 67 中左边的图像。

如果想将这幅图像存到 PostScript 文件中，可以将 PostScript 设备设置为当前图形设备（用 Copy 关键字将当前的颜色表拷贝到 PostScript 文件中），并从新调用上面的 TVScl 命令。例如，可按如下操作：

```
IDL> thisDevice=!D.Name
IDL> Set_Plot, 'PS' , /Copy
IDL> Device, XSize=3, Ysize=3, /Inches, /Color, $
        Bits_Per_Pixel=8, File='image.ps'
IDL> TVScl, image
IDL> Device, /Close_File
```

```
IDL> Set_Plot, thisDevice
```

然而，如果这样做，可能会对结果很失望。输出可能类似于图 67 中右边的图像。这就是说，PostScript 输出上的灰度阴影不同于显示设备上的灰度阴影。

产生这种现象的原因在于彩色表装入到 PostScript 设备时的方式。当调用 Set_Plot 命令时，IDL 将显示彩色表的前 200 种颜色复制到 PostScript 彩色表中相应的颜色中去。（无论是否使用 Copy 关键字与否都是这样。）但它并不影响颜色索引 号在 200 以上的颜色，这些颜色早就被初始化为灰度级颜色。查看体现在 PostScript 文件中的红色矢量的图就知道发生什么了。这个矢量应该是线性型的，但在图 68 中，索引号为 200 时极不连续。那么图像中像素值大于 199 的像素在 PostScript 输出中将用不正确的颜色显示。

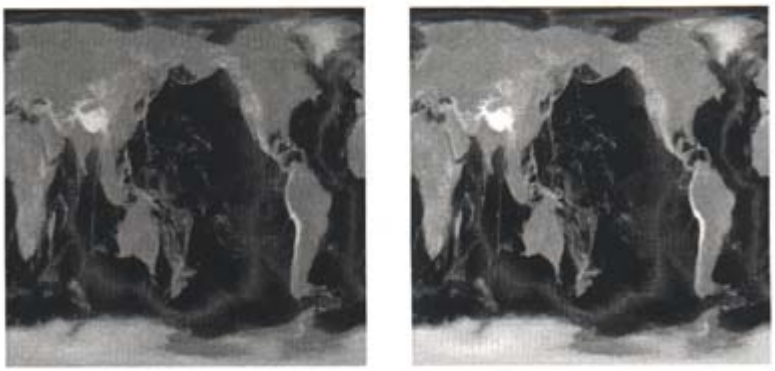


图 67　如果不注意颜色，显示设备上的输出（左图）将不同于 PostScript 文件输出（右图）。特别是，像素值大于显示设备所使用的颜色数的像素将着色不正确。在此例中，很多像素显示的太亮。

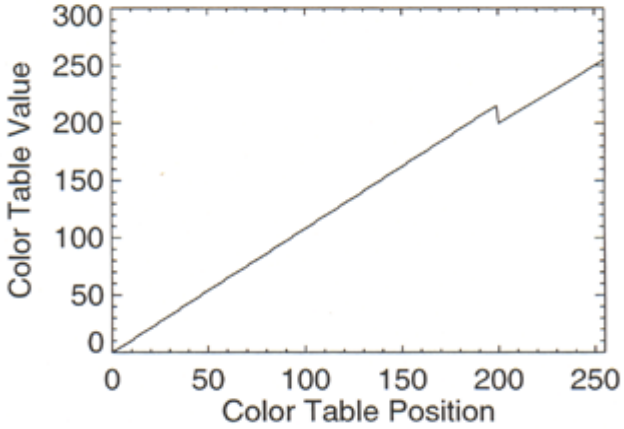


图 68　在将 PostScript 设备设置为当前图形设备后的红色矢量图。彩色表中的前 200 种颜色是从显示设备的彩色表中拷贝过来的。所以，像素值大于 199 的像素在 PostScript 将显示的不正确 c

解决方法：在 PostScript 输出中确保恰当地缩放数据

这个问题可以用两种方法来解决。第一，一旦将 PostScript 设备设置为当前图形设备时，可以重新装载彩色表。或者，确保将图像数据缩放到显示设备所能得到的颜色范围内。重新装载彩色表将使显示设备上的输出与 PostScript 输看上去几乎一样。为了使输出完全一样（当然，是在各种颜色发生技术的约束条件下），有必要将数据缩放到显示设备所能得到的颜色数量范围内。如果彩色表和数据一样，输出也将一样。（关于正确缩放数据参见 66 页的“改变图像尺寸”。）

注意，在缺省情况下，在 PostScript 图像中每个图像像素只保存四位信息。这意味着，即使 PostScript 设备能够显示 256 色，但在输出图像中将只能看到 16 色。如果想看到全部 256 色，必须储存 8 位的像素信息。可以用 Bits_Per_Pixel 关键字在 Device 命令中设置，如下：

```
Device, Bits_Per_Pixel=8, Color=1
```

问题：PostScript 设备显示图像时的不同

显示设备与 PostScript 设备的另外一个不同点是显示图像时的区别。尤其是，显示设备具有固定尺寸的像素，而 PostScript 设备具有可变的像素尺寸。换句话说，在 PostScript 中一个像素实际上可以是任意矩形尺寸。这会影响图像输出到 PostScript 文件中的方法.

PostScript 设备根据 PostScript 画图窗口的尺寸和图像的纵横比来决定图像的大小。例如，如果 PostScript 的绘图窗口为 2*2 英寸，并且要输出的图像为 360*360 像素，那么一个简单的 TV 命令就能输出 2*2 英寸的 PostScript 图像：

```
IDL> thisDevice=!D.Name
IDL> image=LoadData(7)
IDL> Set_Plot, 'PS'
IDL> Device, XSize=2, Ysize=2, /Inches, /Encapsulated
IDL> PlotS, [0,1,1,0,0],[0,0,1,1,0], /Normal
IDL> TV, image
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

上述命令产生的输出如图 69 所示。

图 69　PostScript 设备用可变的像素来使图像适应输出窗口的尺寸。这里的尺寸为 2*2 英寸。

然而，如果输出窗口尺寸与原图像的纵横比不同时，图像将改变尺寸以保证自身的纵横比，其中有一方向将完全填满输出窗口。例如，同样使用上述图像，这里的输出窗口为 X 方向 1 英寸，Y 方向 2 英寸。

```
IDL> Set_Plot, 'PS'
IDL> Device, XSize=1, Ysize=2, /Inches, /Encapsulated
IDL> PlotS, [0,1,1,0,0],[0,0,1,1,0], /Normal
IDL> TV, image
```

```
IDL> Device, /Close_File
```

这些命令得到的结果见图 70。注意，此图像只有 1*1 英寸，只填充了输出窗口的一半。

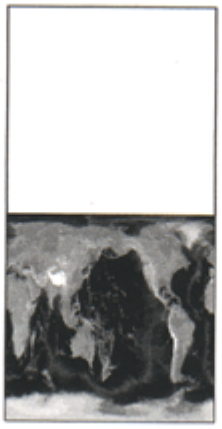


图 70　当输出窗口和原图像具有不同的纵横比时，图像将改变尺寸以维持自身的纵横比，并且其中的一个方向将充满整个输出窗口

类似地，如果有一个 2*1 英寸的输出窗口，如下：

```
IDL> Set_Plot, 'PS'
IDL> Device, XSize=2, Ysize=1, /Inches, /Encapsulated
IDL> PlotS, [0,1,1,0,0],[0,0,1,1,0], /Normal
IDL> TV, image
IDL> Device, /Close_File
```

结果见图 71。

图 71　此图类似于图 70，除了输出窗口的 X 方向是 Y 方向的两倍外

如果 PostScript 绘图窗口是 X 方向 1 英寸和 Y 方向 3 英寸，那么 TV 命令输出的结果是 1*1 英寸的图像。

事实上，图像总是根据输出窗口的尺寸以及原图像的纵横比来确定大小可能会造成困难。例如，假设有一个 500*500 像素的显示窗口，并且想将图像显示在 400*400 像素大小的窗口的中心。更进一步假设，要在图像的周围画一外框。可能会用以下命令在窗口中定位显示图像：

```
IDL> image=LoadData(7)
IDL> image=Congrid(image, 400, 400, /Interp)
IDL> Window, XSize=500,Ysize=500
IDL> TV, image, 0.1, 0.1, /Normal
IDL> Plot, FindGen(100), /NoData, /NoErase, $
      Position=[0.1,0.1,0.9,0.9]
```

如果当前图形窗口为显示设备时，可以看到图 72 所示的输出。

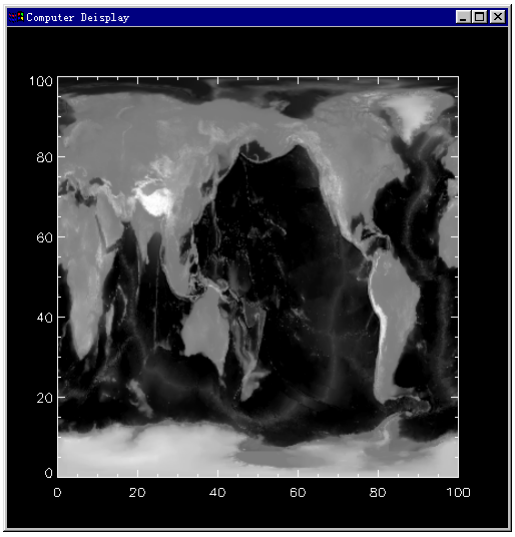


图 72　在显示设备上带边框的图像

但如果在 PostScript 设备上运行这些命令（不是用 Window 命令），将得到非常不一样的结果。尤其是，图像根据输出窗口尺寸改变大小，很可能导致图像的外框的位置不对，如图 73。

解决方法：使用 TV 命令设置图像大小

设置将进入 PostScript 输出中的图像尺寸的正确方法是在 TV 命令中使用的 XSize 和 YSize 关键字。例如，要在 PostScript 输出中得到与图 72 具有相同输出的正确方法如下：

```
IDL> thisDevice=!D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=3.5, Ysize=3.5, /Inches, /Encapsulated
IDL> TV, image, 0.525, 0.25, XSize=2.8, Ysize=2.0, /Inches
IDL> Plot, FindGen(100), /NoData, /NoErase, $
        Position=[0.15,0.10,0.95,0.90]
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

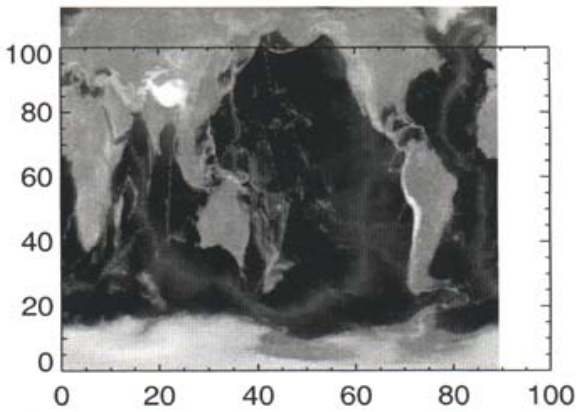


图 73　在 PostScript 输出中，图像的尺寸是根据输出窗口的尺寸来决定的，这可能并非用户想要的，如本图所示

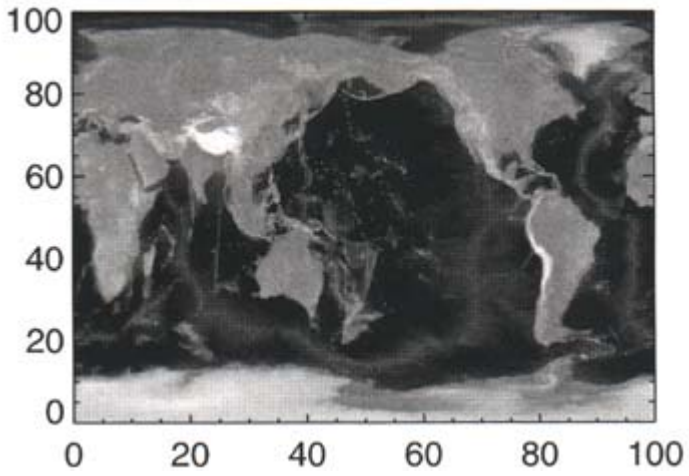


图 74　在 PostScript 窗口中，缩放和放置图像的正确方法是利用 TV 命令的缩放和定位的能力。将此图与图 73 比较一下

如果想编写一个通用的 IDL 程序，如同上面这个，无论窗口大小如何变化都能正常工作，无论是在显示设备上还是在 PostScript 文件中，都同样能工作。这时候，也许需要计算图像在显示窗口中基于设备坐标的大小和位置。在 PostScript 设备上和在显示设备上工作时，惟一的真正区别在于如何计算图像的尺寸。程序 imageax.pro 就是用于此目的（此程序在下载的本书配套程序中）。

```
PRO ImageAx, image, Position=position

    IF N_PARAMS() EQ 0 THEN Message, 'Must pass image argument.'
    IF N_ELEMENTS(position) EQ 0 THEN $
    position = [0.2, 0.2, 0.8, 0.8]

        ; Get the size of the image in pixel units.

    s = SIZE(image)
    imgXsize = s(1)
    imgYsize = s(2)

        ; Calculate the size and starting locations in pixels.

    xsize = (position(2) - position(0)) * !D.X_VSize
    ysize = (position(3) - position(1)) * !D.Y_VSize
    xstart = position(0) * !D.X_VSize
    ystart = position(1) * !D.Y_VSize

        ; Size the image differently in PostScript.

    IF !D.NAME EQ 'PS' THEN $
        TV, image, xstart, ystart, XSize=xsize, YSize=ysize ELSE $
    TV, Congrid(image, xsize, ysize, /Interp), xstart, ystart

        ; Draw the axes around the image.
    Plot, FIndGen(100), /NoData, /NoErase, Position=position
```

```
END
```

打开几个不同尺寸的窗口运行该程序，输出依次显示在每一个窗口中。注意，图像的纵横比不再保持了。但是，它在窗口的位置保持不变。

```
IDL> image=LoadData(9)
IDL> Window, XSize=400, YSize=400, /Free
IDL> ImageAx, image
IDL> Window, XSize=300, YSize=500, /Free
IDL> ImageAx, image
IDL> Window, XSize=600, YSize=300, /Free
IDL> ImageAx, image
```

可以运行这个程序，将输出结果传送到任何窗口，无论是显示设备还是 PostScript 设备都可以。例如，可以用下面的命令将输出送到 PostScript 文件中。

```
IDL> Set_Plot, 'PS'
IDL> Device, XSize=3.5,YSize=2.5,/Inches, /Encapsulated
IDL> ImageAx, image, Position=[0.15,0.15,0.95,0.95]
IDL> Device, /Close_File
```

输出结果见图 75。

想以完全独立于设备的方式来显示图像，笔者偏爱用 TVImage 程序（已下载的本书配套程序）。它不仅能用 Position 关键字按上面 ImageAx 的风格来在显示窗口中定位图像，同时如果愿意，还能保持图像的纵横比。关于 TVImage 命令详见 72 页的“用归一化的坐标来定位图像”。

```
IDL> Window, XSize=600, YSize=400, /Free
IDL> TVImage, image, Position=[0.15,0.15,0.95,0.95], $
     /Keep_Aspect_Ratio
```

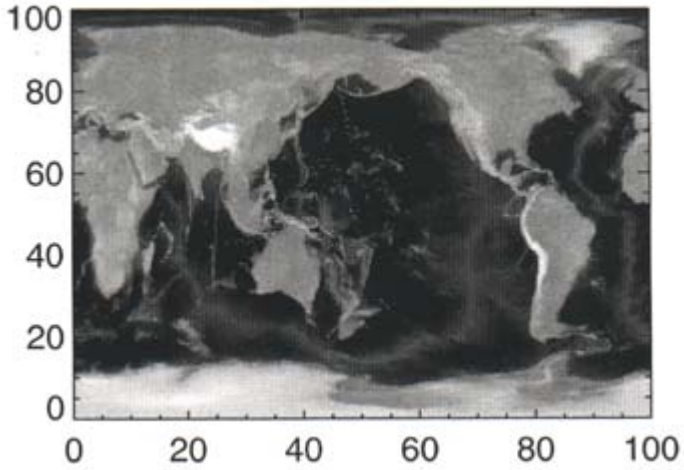


图 75　在 3.5*3.5 英寸的输出窗口中运 ImageAx 程序。注意，图像的纵横比不再保持了，尽管保留了其在窗口中的位置。

在 PostScript 中显示图像的另一个极其重要的地方是，缺省情况下，PostScript 设备对每个图像像素只保留四位的信息。这对 16 色或是灰度级的图像已经足够了。如果想要 256 色，应该将关键字 Bits_Per_Pixel 设为 8，照如下输入：

```
IDL> Set_Plot, 'PS'
IDL> Device, Bits_Per_Pixel = 8, Color = 1
```

在横向输出模式中计算 PostScript 的偏移量

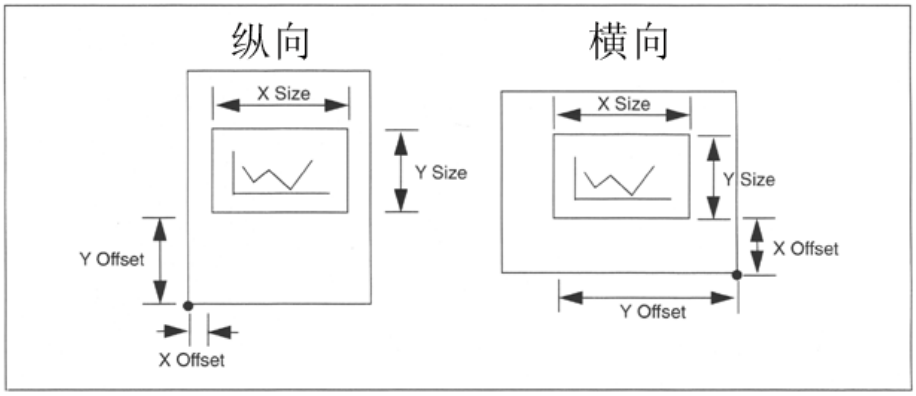


图 76　纵向和横向模式下窗口的尺寸和偏移量。注意，在横向模式下，整个页面被旋转了 90 度，并且偏移量（不是窗口尺寸）也随着一起旋转

纵向模式下的 PostScript 文件偏移量为 X 方向上 0.75 英寸，y 方向上 5 英寸。这就将图形输出到页面的上半部分。因此，非常容易就看出偏移量是基于页面的左下角计算出来。（见图 78。）键入下列语句就可以看出缺省的偏移量：

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Help, /Device
IDL> Set_Plot, thisDevice
```

然而当把图像横向输出为时，整页已被旋转了 90 度，包括页面的左下角！可以在图 76 中看到它们的缺省值。

如果没有意识到偏移点已随页面旋转了，可能设置的偏移量会使图形超出页面。例如，想要使 X 和 Y 方向的偏移量都为 1 英寸，可以这样做：

```
Set_Plot, 'PS'
Device, XOffset = 1.0, YOffset = 1.0, /Inches, /Landscape
Plot, data
```

可以在图 77 中看到想象中作出的图形和实际上作出的图。确信自己明白了偏移量在横向模式下是怎样工作的。

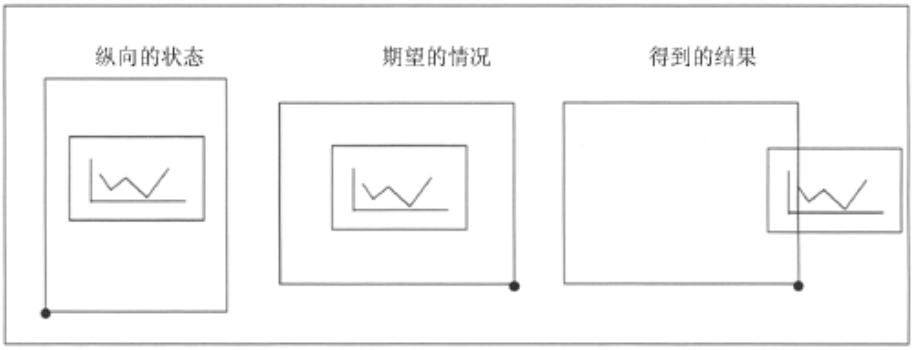


图 77　　如果不注意横向模式下偏移量是怎样工作的，图形将被旋转偏出页面右边

用 PS_Form 配置 PostScript 设备

已下载的本书配套程序中有一个名为 PS_Form 的程序。这个程序的目的是让用户能够交互式地决定将图形放在 PostScript 输出窗口的哪里，以及设置 PostScript 设备其他的配置。可以在图 78 中看到 PS_Form 的图示说明。

在右上角的绘图组件内的黄色方框代表 PostScript 页面。黄色框内的绿色框是 PostScript 页面上输出窗口的位置。用鼠标左键在页面内移动绿色方框。用鼠标右键画一个新的绿色框。

当设置好之后，点 Accept 按钮。PS_From 将返回一个结构，此结构里面的字段都是 Device 命令的有效关键字。以下上 PS_Form 被用来画一个简单图形的例子。

```
deviceKeywords = PS_Form (Cancel = canceled)
If canceled NE 1 THEN Begin
   currentDevice = !D.Name
   Set_Plot, 'PS'
   Device, _Extra = deviceKeywords
   Plot, LoadData (1)
   Device, /Close_File
   Set_Plot, currentDevice
ENDIF
```

注意 PS_Form 的一个优点是，当设备被设置为横向时，用户不必考虑偏移量的旋转问题。对用户来说，偏移量好像总是基于左下角算出来的。

PS_Form 另一个优点是，它可以记住上次设置。例如，像下面这样调用 PS_Form，并更改它的配置。结束以后点击 Accept 按钮。

```
IDL> setup = PS_Form ()
```

要看设置的内容，键入：

```
IDL> Help, setup, /Structure
```

要用刚才的设置内容来启动 PS_Form，键入：

```
IDL> newSetup = PS_Form (Defaults = setup)
```

要看 PS_Form 是如何被应用的，可以试着调用 XWindow 程序，它也是下载的本书配套程序之一。XWindow 是一个“智能化”的图形窗口，它可以自我调整大小，可以载入仅供它自己使用的彩色表，也可以将它的输出送到 PostScript 文件中。可以这样来调用它：

```
IDL> XWindow, 'Shade_surf', LoadData(2), /Output, /XColors
```

尝试一下用 XWindow 程序将窗口里面的内容制作成一个 PostScript 文件。这本书余下的大部分内容将讨论如何编写一个类似于 Xwindows 的程序。

配置和使用打印设备

打印设备在 IDL5.0 中被引入介绍，最初不像其他图形输出设备能用 Device 命令来配置。Dialog_PrinterSetup 命令是用来存取计算机上缺省打印机的配置参数。解释默认打印机如何安装和配置已经远远超出了本书要讨论的范围，但一般来说，打印机配置对话框提供了较多配置打印机自身的选项，但对于如何定位图形输出的选择相对较少。比如说，PostScript 设备。

为了让用户在打印设备上有更多的选项来定位图形输出，Research Systems 公司在 IDL5.1.1 中为 Printer 设备引入了 Device 关键字。（注意，这些关键字仅适用于向打印机发送直接图形命令时。）这些关键字 XSize，YSize，XOffset 和 YOffset 和其他硬拷贝输出设备中同名关键字很象，尽管不完全是。下面将指出它们的一些不同之处。

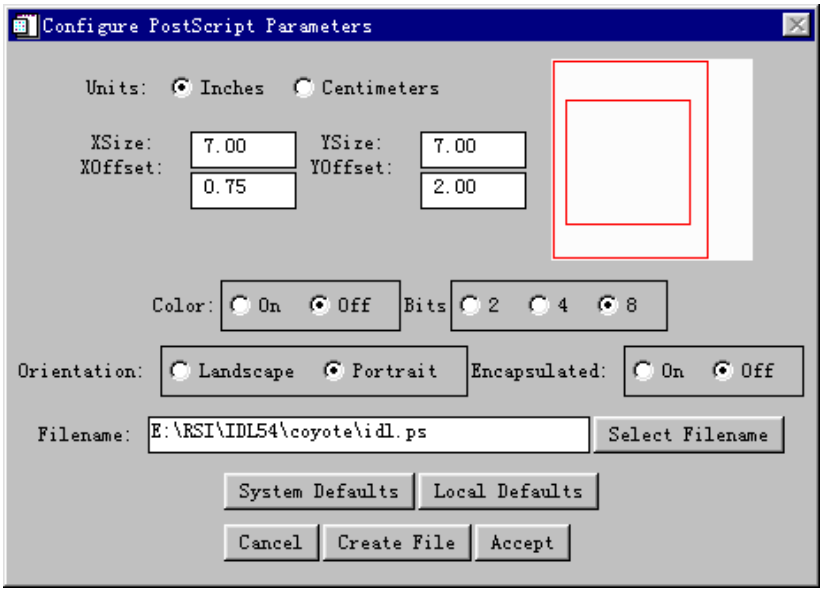


图 78　弹出式组件程序 PS_Form。这个程序为用户配置 PostScript 设备提供了交互式方法。此图为如何配置 PostScript 设备来生成本书的大部分图形

要存取默认打印机的配置，键入：

```
IDL> ok = Dialog_PrinterSetup ()
```

此对话框在 Windows NT 上如图 79 所示。

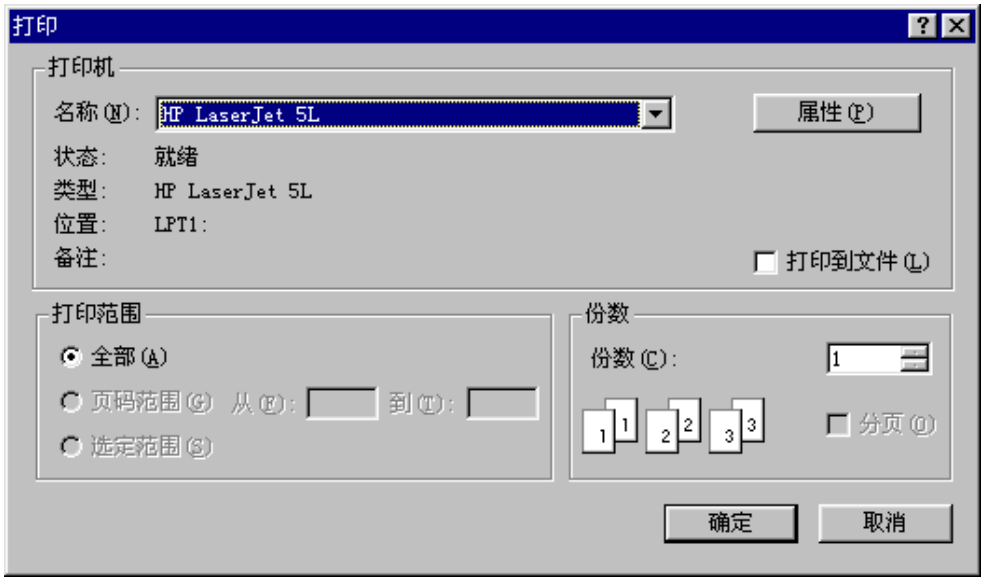


图 79　在 Windows NT 机器上的 Dialog_PrinterSetup 的对话框

在使用打印设备时，重要的是要知道，只有关键字 Close_Document 被用于 Device 命令时，输出内容才会被送到打印机上。例如，生成一幅线画图命令的正确顺序类似于下面的代码。关闭打印机文档是必须的。如果忘了这条代码，打印机不会输出任何东西。由于下面的代码中有一个 IF 循环，所以下面的代码必须放在一个文本编辑器中编辑，就像一个 IDL 主程序一样。将文件存为 sendprinter.pro。可以在下载的程序中找到这个程序。

```
data = LoadData(1)
ok = Dialog_PrinterSetup()
IF ok THEN BEGIN
    thisDevice = !D.Name
    Set_Plot, 'PRINTER'
    Plot, data
    Device, /Close_Document
    Set_Plot, thisDevice
ENDIF
END
```

如果想运行这个主程序并且把结果输出到默认打印机上，可以这样做：

```
IDL>. Run sendprinter
```

用打印设备定位图形

在 Printer 设备的第一版中，当将图形输出到默认打印机时，输出的图形常常充满了整页纸，看起来根本不象显示设备上的图形。实际上，不能控制图形从打印机出来后在页面上的位置。例如，图像按设备精度打印时，其左下角位于页面的左下角。一个 256 乘 256 的图像用 600 dpi 的像素分辨率打印到一个 PostScript 打印机上时，经常只有 0.5 平方英寸，除非应用了合适的比例放大因子。像素在 PostScript 设备上，不会按比例地缩放到纸上。TV 或 TVscl 命令中的 XSize 和 YSize 关键字用于 PostScript 时同样不能缩放像素。（例子见 71 页的“在 PostScript 设备上改变图像大小”）

这一点在 IDL5.1.1 种作了改进，对 Device 命令增加了关键字，可以和 Printer 设备一起被用来定位图形在纸上的位置和比例。像在 PostScript 设备中的同名关键字一样，Printer 设备的关键字默认时用厘米单位。（也可以设为英寸，如果 Inches 关键字被使用的话。）

在纵向输出模式下，XSize, YSize, XOffset 和 YOffset 的默认值是（用英寸）：

XOffset: 0.75 inches
YOffset: 5.0 inches
XSize: 7.0 inches
YSize: 5.0 inches

在横向输出模式中，默认值为：

XOffset: 0.75 inches
YOffset: 0.75 inches
XSize: 9.5 inches
YSize: 7.0 inches

读者立即会发现这些缺省值在 Printer 设备上和 PostScript 设备上给出了相同的相对输出尺寸。但是也要注意，偏移量常常从页面的左下角计算出来。这是显而易见的，但是横向模式下 PostScript 偏移量不是这样计算的（见 199 页的“在横向输出模式中计算 PostScript 的偏移量”）。就意味着在编写一个既能创建 PostScript 文件又能将图形显示直接送到打印机的程序时，要格外注意横向模式下的偏移量。

为了帮助读者正确地计算这些关键字的值，Research Systems 公司同时为 Device 命令引入了 Get_Page_Size 关键字，它可以用来返回一个包含打印设备页面的 X 方向尺寸和 Y 方向尺寸的两维矢量。奇怪的是，只能用设备坐标返回页面的尺寸，尽管页面和偏移量的关键词是用英寸或厘米来表示。因此，要想在页面上获得精确的输出结果，必须做些计算。

例如，如果想让一个图形在输出时占页面的 80%，也可以键入这些命令：

```
IDL> thisDevice =!D.Name
IDL> Set_Plot, 'PRINTER'
IDL> Device, Get_Page_Size = myPage
IDL> Device, XSize = myPage[0]*0.8, XOffset = myPage[0]*0.1, $
       YSize = myPage[1]*0.8, YOffset = myPage [1]*0.1, /Device
IDL> Plot, Findgen (11)
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

用打印设备输出图像

向 Printer 设备输出一张图像稍不同于 PS 设备向 PostScript 文件输出一张图像（见 193 页的“问题：PostScript 设备显示图像时的不同”。）主要区别在于通过 Printer 设备输出的图像没有保持纵横比，而用 PostScript 设备输出时却保留了。但是，和 PostScript 设备一样，Printer 设备也能在 TV 或 TVScl 命令中用 XSize，YSize 关键字来恰当的改变图像的尺寸。图像的偏移量可以用 Device 命令中的关键字 Xoffset，YOffset 关键字来设置。

例如，假设想在页面的中间放一个常规的图像，在页面上的纵横比为 2/3，可以键入下面这些命令：

```
IDL> thisDevice =!D.Name
IDL> Set_Plot, 'PRINTER'
IDL> Device, XOffset = 1.25, YOffset = 3.5, /Inches
IDL> image = LoadData (7)
IDL> TV, image, XSize = 6, YSize = 4, /Inches
IDL> Device, /Close_Document
IDL> Set_Plot, thisDevice
```

第八章　IDL 编程基础

本章概述

本章的目的是学习 IDL 基本编程技巧，具体的来说，可以学到以下方面内容：

1. IDL 批处理文件、主程序、过程和函数之间的区别；
2. 如何在 IDL 程序中输入和输出信息；
3. 如何在 IDL 程序中使用位置参数和关键字；
4. 如何编译和运行 IDL 程序；
5. 程序中的常用控制语句语法

如果把 IDL 程序看作一系列 IDL 命令的话，那么 IDL 程序——或称为 IDL 程序模块——可以分为四类：（1）批处理文件（2）主程序（3）过程（4）函数。

编写 IDL 批处理文件

最简单的程序是一个 IDL 批处理文件。一个批处理文件由一系列命令组成，这与在 IDL 命令行敲入的命令完全一样。大多数人用批处理文件是为了自动执行自己在 IDL 命令行一次又一次敲入的命令。

例如，假设要在 IDL 中打开并显示一组图像，如果已经将图像数据读入到变量 image 中，那么用来显示图像的命令可以如下所示：

```
IDL> thisImage = BytScl (image, Top = 199)
IDL>LoadCT, 5, NColors = 200
IDL> s = Size (image)
IDL> Window, /Free, XSize = s[1], ysize = s[2]
IDL> TV, thisImage
```

这五行代码并不多，但键入三、四次之后，可能已决定把他们放在一个名为 ImageOut.pro 文本文件中。这文件就是所谓的批处理文件。

要执行该文件中的命令，必须把@放在 IDL 命令行的开头，其后再加上文件名即可。（.pro 为默认扩展名。如果加了其他扩展名，那么应该把文件名称写完整）如下所示：

```
IDL> @ImageOut
```

注意，文件名没有在引号中，这与 IDL 文件名的一般规则是不一致的。

IDL 会严格执行批处理文件中的命令，就像在命令行上键入一样。这意味着有必要用行续字符（$）和其他命令行语言来让 IDL 确认键入的命令。如果在文件中的命令输入错误，那么出现的错误结果和在命令行键入命令出现的错误结果是一样的。

假设要打开 8 到 10 个图像文件，因而不得不分别打开每个图像文件，读取数据，然后运行批处理命令将每一个图像显示在窗口中。具体地说，可以这样自动进行读取数据和显示图像过程：

```
theseFiles = FindFile (‘*.img’, count = numFiles)
Print, ‘number of files found: ‘, numfiles
For j = 0, numFiles-1 Do Begin
    Openr, lun, theseFiles[j], /get_lun
    Image = BytArr (512,512)
```

```
        Readu, lun, image
        Free_lun, lun
        thisImage = BytScl (image, top =199)
        LoadCT, 5, Ncolors =200
        S = size (image)
        Window, /Free, XSize = s[1], YSize = s[2]
        TV, thisImage
    Endfor
```

但是这个文件不适合于作批处理文件，因为 FOR 循环里面有多行语句。如果没有续行符($)和命令连接符(&)的话，这种程序是很难写到命令行中的。为了自动执行由多行控制语句组成的命令，最好使用 IDL 主程序。

编写 IDL 主程序

IDL 主程序和批处理文件在很多方面很相似，但也存在着很大的区别。像批处理文件一样，一个主程序也包含一系列命令。但与之不同的是，这些命令必须以 END 语句结束。例如，上面自动读取数据和显示图像的程序可以写成一个 IDL 主程序：

```
    theseFiles = FindFile ('*.img', count = numFiles)
    Print, 'number of files found: ', numfiles
    For j = 0, numFiles-1 Do Begin
        Openr, lun, theseFiles[j], /get_lun
        Image = BytArr (512,512)
        Readu, lun, image
        Free_lun, lun
        thisImage = BytScl (image, top =199)
        LoadCT, 5, Ncolors =200
        S = size (image)
        Window, /Free, XSize = s[1], YSize = s[2]
        TV, thisImage
    Endfor
    End
```

批处理文件和 IDL 主程序最大的区别就是主程序的命令语句先由 IDL 编译器编译成程序模块，然后才执行代码。这就是为什么在主程序中可以有多行控制语句的原因。

如果将上面的代码保存在文件 ImageOut.pro 中，键入如下命令就可以编译并运行这个程序模块：

```
IDL> .RUN ImageOut
```

现在，主程序就已经驻留在 IDL 内存中了。同一时间只能有一个主程序驻留在 IDL 内存中。如果没有重新编译该代码而要重新运行这个程序，可以使用可执行命令.GO 来实现，如下：

```
IDL> .Go
```

可执行命令(.Go,Compile,Run,等等)只能应用在 IDL 开发环境中，而不能用在 IDL 过程和函数中（尽管有很多方法可以在过程和函数中编译程序。详细情况请参阅 P230“特殊编译命令”）

编写主程序的最大优点是在程序中定义的变量可以在 IDL 开发环境中使用，IDL 的命令都是在那里键入并被解释的。换句话说，在主程序中定义的变量的作用范围是 IDL 整个开发环境，因而它可以被 IDL 其他命令使用。

更为通常的是，往往希望限定变量的作用范围，以使它们不致于占用大量的内存。在编程过程中，使用大量的全局变量会使内存的使用效率大大降低。为此，大部分应用程序都用过程和函数来编写。

编写 IDL 过程

IDL 过程和 IDL 主程序很相似，同时也存在很大的区别。首先，在编写一个过程时，实际上所做的是创造另外一个 IDL 命令，一个构造在 IDL 系统语言之上的新命令。在 IDL 命令行和程序中，可以使用所创造的命令，就像使用 IDL 系统内置的命令一样。

过程看起来和主程序非常相似，不过过程是以过程定义语句开始的。定义语句的目的就是为过程命名（如果喜欢，也可以称之为命令名称）和定义过程参数。过程的参数既可以是位置参数也可以是关键字。待一会儿将学到更多有关过程参数如何定义方面的知识。

例如，想将上面的主程序写成一个名为 ImageOut 的 IDL 命令，可以这样编写这个过程：

```
PRO Imageout
theseFiles = FindFile (‘*.img’, count = numFiles)
Print, ‘number of files found: ‘, numfiles
For j = 0, numFiles-1 Do Begin
    Openr, lun, theseFiles[j], /get_lun
    Image = BytArr (512,512)
    Readu, lun, image
    Free_lun, lun
    thisImage = BytScl (image, top =199)
    LoadCT, 5, Ncolors =200
    S = size (image)
    Window, /Free, XSize = s[1], YSize = s[2]
    TV, thisImage
Endfor
End
```

如果这些代码被保存在一个名为 ImageOut.pro 文本文件中，在命令行上键入此程序的名字，它就会自动编译和执行。如下所示：

```
IDL> Imageout
```

有时候想在运行之前清楚显式地编译一个过程或函数。(例如，在代码中有一个错误，在修改完毕之后，运行之前希望将程序重新编译。）如果想要显式地编译和运行上面的代码，可以这样键入：

```
IDL>.Compile ImageOut
IDL> Imageout
```

注意，在上述编译语句中的 ImageOut 是想要编译的文件的文件名（IDL 默认.pro 为文件扩展名）。文件名并没有加引号，但是它可能会区分大小写，这取决于当前的操作系统。.Compile 命令将编译该文件中所有的程序模块，但不会运行其中的任何一个。（在这个例子中只有一个程序模块。编译方面的详细信息请参阅 P228“编译和运行 IDL 程序模块”。）文件名不一定必须和过程名称相同，但通常情况是这样做的。上面的第二行语句是想要运行的程序或程序模块的名称。

过程和与函数中变量的作用范围

关于过程和函数很重要的一点是在过程和函数中创建的变量是局部变量。也就是说，只有过程和函数内部的命令才能调用他们内部创建的变量。

例如，在上面的过程中，在 IDL 命令行中是不可能识别变量 theseFiles，thisimage 或者 s 的，因为这些变量的作用范围只限于过程。而且，当 IDL 退出该过程时（在这种情况即为执行到 end 语句），这些变量被清除，它们所占用的内存也被释放。

如果所有的变量的作用范围都只是局部的，这就极其不方便了。因为那样就不可能在过程、函数之间，或命令行与过程及函数之间进行信息交流和数据传递了。因此，可以采用多种办法来扩展变量的作用范围，或在过程、函数之间传入或传出信息通常情况下，信息（例如变量等），是以过程和函数的参数形式来传递的。

创建位置参数

位置参数是在过程的定义语句中被定义的。一般来说，位置参数的数量是没有限制的，但对它们的顺序有严格的要求。第一个定义的位置参数（“定义”的意思是把它放在过程名称的右边）是参数 1，紧接着定义的是参数 2，以此类推。

例如，希望指定 ImageOut 过程通常在哪一个子目录下查找图像文件。可以通过字符串参数 lookHere 把信息传入到 ImageOut 过程中去，修改后的 ImageOut 过程如下所示，增加部分已用黑体字标出。

```
PRO ImageOut, lookHere
Cd, lookhere
theseFiles = FindFile (‘*.img’, count = numFiles)
Print, ‘number of files found: ‘, numfiles
For j = 0, numFiles-1 Do Begin
    Openr, lun, theseFiles[j], /get_lun
    Image = BytArr (512,512)
    Readu, lun, image
    Free_lun, lun
    thisImage = BytScl (image, top =199)
    LoadCT, 5, Ncolors =200
    S = size (image)
    Window, /Free, XSize = s[1], YSize = s[2]
    TV, thisImage
Endfor
End
```

若要调用修改以后的过程，应该重新编译它。之后，就可以这样来调用它：（假设在 Windows 操作系统中。下面的代码仅起示范使用）

```
IDL>. Compile ImageOut
IDL> ImageOut, ’c:\rsi\mydatafiledir’
```

字符串 c:\rsi\mydatafiledir 被拷到参数（有时也叫形参或局部变量）lookhere 后，命令 CD 根据这个将工作路径改变到与之同名的指定路径中。

这个字符串也可以作为变量传递到过程 ImageOut 中。如下面所示：

```
IDL> myimagedirectory = ‘c:\rsi\mydatafiledir’
IDL> ImageOut, myimagedirectory
```

在这里，变量 myimagedirectory 通过引用方式来传递的。它的意思以后将会更详细讨论到。但其中一点就是，这个变量现在可以在主程序和过程 ImageOut 中被使用。这意味着如果在过程内部改变它的值，这个变化将会反映到 IDL 命令行上来。换句话说，IDL 命令行上的 myimagedirectory 变量和 ImageOut 程序中的变量 lookhere 是相同的。

也就是说，这个变量有两个名字，这两个名字都指向同一个数据或 IDL 中的物理地址。（就好比说，一个人在美国时人们叫他为乔，而当他回到家乡智利时是人们叫他乔丝。两个名字所指的是同一个人，但是一个地方的人只知道他的一个名字。）

但是如果在调用 ImageOut 时忘记了传递参数会怎样呢？如果键入如下所示会发生什么事？

```
IDL> ImageOut
```

在这种情况下，什么都不会传递到变量 lookhere 中去，所以在过程中变量就没有定义（但是，没有定义也是变量的一个有效类型）。遗憾的是，这会带来很大的麻烦，因为在过程中，CD 命令语句把 lookhere 变量作为它的参数。而在 cd 命令中使用一个没有定义的变量会导致错误的。因此，清楚地知道过程如何被调用是非常有必要的。

定义可选的或必须的位置参数

IDL 提供的 N_Params 命令可以返回过程在被调用时位置参数的调用个数（不包括关键字）。这个命令如下所示：

```
Numparams = N_Params ()
```

知道了过程在调用时输入了多少个位置参数，就有机会对所传入的参数是可选择的还是必选的进行判断。例如，想把参数 lookHere 作为一个必须的参数，在过程的前几行应增加如下几行：

```
Pro ImageOut, lookHere
Numparams = N_Params ()
If numparams EQ 0 Then $
    Message, 'must supply one parameter to this procedure.'
Cd, lookHere
```

这里，message 命令会产生一个错误，导致 IDL 停止执行过程中的命令，同时错误信息被送到命令记录窗口或输出窗口。这个结果和在没有数据参数的情况下调用 plot 命令是一样的。

也许在 ImageOut 过程中把参数 lookHere 作为一项可选参数更合情合理。

如果没有提供参数，那么 lookHere 变量的值就会设置为当前子目录。实现这部分功能的代码如下所示：

```
Pro ImageOut, lookhere
Numparams = N_Params ()
If numparams EQ 0 Then CD, current = lookhere
Cd, lookHere
```

在这里，如果在调用 ImageOut 时没有提供 lookhere 参数，那么 cd 命令就和关键字 Current 一起使用，并将当前工作目录放在 lookhere 变量中（关键字 current 在这里是一个输出性质的关键字，关于这个，读者等下将会了解更多。）

在 IDL 过程和函数中，定义参数的习惯性规则是，参数是指必须的参数，关键字是指可选参数。这种规则常常因为参数而破例，而几乎不会因为关键字而破例。也就是说，经常可能会出现参数是可选的，却很少有关键字是必选的情况。

定义关键字

和位置参数一样，关键字也是在过程的定义语句中加以定义的。只要愿意，关键字可以和夹杂在参数中定义，（这对参数的相对位置没有任何影响），但一般地，关键字在参数定义后再定义。

如果读者看了过程 ImageOut 的代码，就会发现程序中硬性地载入了颜色表 5。但是，如果允许用户指定颜色表，并且只有在用户没有指定的情况下才将颜色表 5 作为默认的颜色表，那样就更符合情理了。这便是关键字所起的作用。关键字定义语法如下:

Keywordname = keywordsymbol

等号左边的是关键字名字，这是关键字使用时的名字；等号右边是在过程和函数中关键字所赋的值。例如，下面是为 Imageout 过程定义关键字 colortable 的例子：

Pro ImageOut, lookHere, colortable = thiscolortable

关键字 colortable 是连同颜色表值一起调用 ImageOut 过程时即将使用到的名称。例如，想用 Red-Temperature 色表，即色表 3，来浏览这些图片，可以这样调用过程：

IDL> ImageOut, ‘c:\data’, colortable = 3

使用缩写关键字

关键字名称，colortable，在使用过程中不必全部拼写，只要有足够多的字母将它和其他关键字区别开来就可以了。因为 colortable 是 ImageOut 中惟一的关键字，所以用 c 已足够来定义这个关键字。过程 ImageOut 也可以这样调用。

IDL> ImageOut,’c:\data’, c = 3

或者这样：

IDL> ImageOut, ‘c:\data’, color = 3

上述三种调用方法的作用是相同的。

关键字右边的变量，thiscolortable，是用来给程序中关键字赋值的。在这里值赋为 3。

现在可以重新编写 ImageOut 的代码了（修改部分已用粗体字标出）：

```
Pro ImageOut, lookHere, colortable = thiscolortale
Numparams = N_Params ()
If numparams EQ 0 Then CD, current = lookHere
CD, lookHere
theseFiles = FindFile (‘*.img’, count = numFiles)
Print, ‘number of files found:’ numfiles
For j = 0, numFiles-1 do begin
    Openr, lun, theseFiles[j], /get_lun
    Image = BytArr (512,512)
    Readu, lun, image
    Free_lun, lun
    Thisimage = BytScl (image, top = 199)
    LoadCT, thiscolortablel, ncolors = 200
    S = size (image)
    Window, /Free, XSize = s (1), YSize = s (2)
    TV, thisimage
Endfor
End
```

但是如果变量 thiscolortable 没有定义呢？换句话说，如果用户这样调用 ImageOut 过程会怎么样呢？

```
IDL> ImageOut
```

由于关键字 colortable 没有被使用，所以在变量 thiscolrotable 里面什么也没有。这话也可以这样理解：变量 thiscolortable 在过程中没有定义。如果这样，那么在上述代码中 LoadCT 命令以下的三分之二的编码都会无效，因为它不能接受一个没有定义的变量作为参数。

这使读者想起上次曾经遇到的情况，即在调用 ImageOut 时应该知道参数是否被使用。在这里读者应该知道，在调用 ImageOut 时关键字是否被使用的情况。

定义可选择的关键字

关键字是可选择的参数。这意味着，在调用含有关键字的过程和函数时，关键字很可能不会被使用。但是赋有关键字的值的变量还是会在程序中被使用的。这意味着要对每一个已经定义的关键字赋予默认值。实际上如果不这样做的话，IDL 应用程序迟早会出错。

但是怎样才知道哪些需要定义默认值呢？当然，如果用户已经赋值那就没有必要来定义默认值。

但是怎样知道用户是否已赋值了呢？如果是参数，可以用 N_Params 命令来提供这方面的信息。遗憾的是，N_Params 只对参数才起作用，而不能提供关于关键字的任何信息。

很多人把这个问题想象为“我想知道关键字使用了还是没有？”。如果是这样的话，要查明关键字是否使用比其最初出现的时候要难多了。我们经常期盼有好的结果出现，但是现实却未必如此。我们会问，“这个变量已经赋予了关键字的值还是没有？”

关键字定义了吗？

对一个关键字，比如 colortable，很可能被赋予很多可能的值，用 N_elements 这个 IDL 命令可以知道在过程和功能程序中，变量 thiscolortable 是否已经定义。尽管 N_ements 在 IDL 中还有其他作用，但是在这个情况下，使用这个函数的功能是：如果 N_ements 的参数没有定义，则函数返回值为零。否则将返回参数的元素个数。该参数可以是任意数据类型。

在这种情况下，过程 ImageOut 在用户没有提供色表的情况下，将会定义默认的色表值，它的前几行如下所示：

```
Pro ImageOut, lookhere, colortable = thiscolortable
Numparams = N_Params ()
If numparams eq 0 hen cd, current = lookhere
Cd, lookhere
If N_Elements (thiscolortable) EQ 0 Then thiscolortable = 5
```

注意，一定要把关键字的名字（比如，它在 IDL 命令行中是如何使用的）和赋有关键字的值的变量区分开来。关键字的名字在关键字定义的左边，赋值的变量在右边。N_elements 的参数必须是关键字的变量，而不是关键字的名字。

这点经常被误解，因为一些 IDL 程序员（包括作者在内）喜欢把关键字名字和变量拼写成一样。换句话说，如果作者正在为自己编写以下程序，作者会这样写：

```
Pro ImageOut, lookHere, colortable = colortable
Numparams = N_Params ()
If numparams EQ 0 Then CD, current = lookHere
Cd, lookHere
If N_Elements (colortable) EQ 0 Then colortable = 5
```

这里不存在正确与否，但还是存在区别。IDL 会清楚地区别这些。建议读者也这样做。

记住，检查的应是关键字变量，不是关键字名字。作者喜欢把关键字变量和名字拼成一样，因为这样可以减少拼写错误。我可以认为我是在检查关键字名字，事实上我在检查关键字变量。

处理具有双重属性的关键字

如果仔细检查 ImageOut 的代码，将会发现图像数据在显示之前已经被拉伸了。这个特殊代码行是：

```
Thisimage = BytScl (image, top = 199)
```

图像数据文件很可能已被拉伸过，因此这个步骤并不是必须的。如果这样的话，就可以定义一个名为 scale 的关键字，当过程带该关键字调用时，则拉伸图像，如果不带该关键字调用，则忽略图像的拉伸。这样的关键字具有双重属性，它可以设置也可以不设置。其他关键字同样具有双重属性，也就是说它们可以是真或假，是或者否，1 或者 0。

IDL 提供了一个特殊的命令来处理类似的关键字，即 Keyword_Set 命令。和 N_elements 一样，Keyword_Set 的参数是关键字变量而不是关键字名字。但 Keyword_Set 有一点不一样就是，如果 Keyword_Set 的参数没有定义或者值为 0，那么它将会返回 0。否则返回 1。所以，Keyword_Set 的返回值只有两种：0 或者 1。

许多程序员错误运用了 Keyword_Set，用它来检测关键字是否使用。其实不是这样的，用它来验证一个关键字是否已经使用最终会导致 IDL 应用程序的错误。只能在关键字具有双重属性时才能使用它。

在确定只有设置了关键字 scale 情况下才调用图像拉伸步骤，这时的 ImageOut 代码可以这样来写：

```
Pro ImageOut, lookhere, colortable = thiscolortable, Scale = scaleIt
Numparams = N_Params ()
If numparams eq 0 then cd, current = lookhere
Cd, lookhere
Thesefiles = findfile (‘*. img’, count = numfiles)
Print, ‘number of files found: ‘, numfiles
For j = 0, numfiles-1 do begin
    Openr, UN, thesefiles (j),    /get_lun
    Image = BytArr (512,512)
    Readu, lun, image
    Free_lun, lun
    If Keyword_Set (scaleIt) then $
        Thisimage = BytScl (image, top = 199) else $
        Thisimage = image
    LoadCT, thiscolortable, ncolors = 200
    S = size (image)
    Window, /Free, XSize = s (1), YSize = s (2)
    Tv, thisimage
Endfor
End
```

现在，希望对图像进行拉伸，可以这样调用过程 ImageOut：

```
IDL> ImageOut, /scale
```

记住/keyword 只是表示 keyword=1。这种设置关键字的简单表示方法在使用具有双重属性的关键字时经常碰到。

创建输出型参数

到目前为止，在过程 ImageOut 中创建的关键字或参数都是输入型参数。换句话说，信息是从程序以外传入进来的。但是，还经常需要把信息输出到此程序外部。

比如说，ImageOut 的用户想要知道被打开的文件名。在过程内部这些文件名被储存在变量 thesefiles 中。对 ImageOut 来说，这是个局部变量。也就是说，该变量的作用范围仅限于过程本身。一个用户怎样才能从过程外部获得这些信息呢？

有许多方法可以采用，比如说，一个公用模块可以扩大变量的作用范围，但在通常情况下，不需要公用模块。一个简单的方法是使用输出型参数。

用引用和传值的方法传递信息

209 页中，在“创建参数”中讨论了关于用引用传递参数的问题。通过引用的方式，参数的作用范围比局部变量作用范围要大。也就是说，通过引用来传递参数事实上是——在低级意义上——指向 IDL 内存中的地址，这从而使得在所有具有引用权限的程序中该参数都可以被引用。在程序之间通过引用方式进行传递数据的，而实际上，任何程序对该数据的修改都将影响其他程序中该数据的结果。

另一个传递信息的方法是传值。就是说，传递到程序中的不再是指向数据地址的指针，而是数据值的拷贝。在程序模块中对数据的修改不会对原有数据有任何改变。

在 IDL 中，所有的变量都是以引用的方式传递的。其他的，比如下标变量、系统变量、表达式、结构，常量，都是通过传值的方式进行的。

这里举一个简单的例子来说明这一点。这是一个用来调整图片的大小并将它显示在一个 150*150 的窗口中的程序。在文本编辑器中键入如下所示，并保存为 resizeit.pro。

```
Pro resizeit, image
Image = congrid (image, 150,150)
Window, 0, xsize = 150, ysize = 150
Tvscl, image
End
```

现在用 loadata 来载入世界海拔高度数据，如下：

```
IDL> image = loaddata (7)
```

用 help 命令检查变量：

```
IDL> help, image
Image                byte          =          array (360,360)
```

现在用 resezeit 程序调入该数据，再次检验图片变量。这时，变量 image 已经变成了一个 150*150 的字节型数组。

```
IDL> resizeit, image
IDL> help, image
Image                byte          =          array (150,150)
```

变量 image 已经发生变化，这是由于在传入过程 resizeit 中时，它是以引用方式被传递的。换句话说，它在 resizeit 程序中具有使用权限。在这里，变量 image 既是输入型变量（即传入到程序中的信息），又是输出型变量（输出到程序外的信息）。变量可以是输入型变量、输出型变量，或者两者都是。这完全取决于在 IDL 程序中怎样编写它的代码。

假如想把 image 数据传入到程序，但不想在程序内部对它有所改变，也可以选择下面两种方法之一：（1）重新编写 resizeit 程序，不改变 image 变量；（2）用传值的方式将变量 image 传入到当前程序中。如果是第一种方法，可以这样重新编写程序：

```
Pro resizeit, image
Window, 0, xsize = 150, ysize = 150
Tvscl, congrid (image, 150,150)
End
```

在第二种情况下，也可以用表达式作为参数，如下所示：

```
IDL> rsizeit, image + 0b
```

有时候也会遇到相反的问题，比如，希望在过程或函数中改变某个值而实际上却没有。这通常是因为输入的参数不是变量，而可能是变量的一部分。例如，它可能是结构中的一个字段。因为结构、系统变量，任何一种表达是都是用传值方式传递，而不是引用。只有变量（并且是完整的变量）才是通过引用方式来传递的。

例如，假设上面的变量是一个自定义的系统变量：

```
IDL> image = loaddata (7)
IDL> defsysv,'image', image
```

如果这样调用 resizeit 程序：

```
IDL> resizeit, !image
```

这时，就不可能将新尺寸的图片输出到程序外和传递到系统变量中，因为这是系统变量！image 总是以传值方式传递的而不是以引用的方式。正确的命令如下：

```
IDL> thisimage = !image
IDL> resizeit, thisimage
IDL> !image = thisimage
```

为了解决从 ImageOut 过程中获得文件名的问题，选择输出型关键字是比较恰当的。关键字，和参数一样，可以通过传值或引用方式来传递。带有关键字 filename 的新程序代码如下，变化已经用粗体标出：

```
Pro ImageOut, lookhere, colortable = thiscolortable, Scale = scaleit, $
    Filenames = theseFiles
Numparams = N_Params ()
If numparams eq 0 then cd, current = lookhere
Cd, lookhere
Thesefiles = findfile ('*.img', count = numfiles)
Print 'number of files found:', numfiles
For j = 0, numfiles-1 do begin
    Openr, lun, thesefiles (j),    /get_lun
    Image = bytarr (512,512)
    Readu, lun, image
    Free_lun, lun
    If Keyword_Set (scaleit) then $
        Thisimage = bytscl (image, top = 199) else $
        Thisimage = image
    Loadct, thiscolortable, ncolors = 200
    S = size (image)
    Window, /free, xsize = s(1), ysize = s(2)
    Tv, thisimage
Endfor
End
```

如果用户希望返回文件名，只需简单使用关键字名字，并将它赋值给某个变量。如下所示：

IDL> ImageOut, filenames = myfiles

尽管在过程 ImageOut 被调用时变量 myfiles 可能还没有定义，但当过程返回时它会被定义为一个字符数组。如果变量 myfiles 在调用 ImageOut 时已经定义，它会被 ImageOut 再定义一次，而它的初始值将会丢失。

参数存在吗

使用输出型参数，常常希望知道该参数或是关键字是否存在。例如，某个程序在计算数据时很费时间，但计算结果并不需要。因此希望，只有在用户需要输出参数时才会进行计算。

为了更生动的说明这个问题，怎样才知道当用户这样调用程序 ImageOut 时，参数是否存在？

IDL> ImageOut

或这样：

IDL> ImageOut, filenames = myfiles

答案是无法确定的。在过程 ImageOut 内部，可以用 N_emements 或 Keyword_Set 来检查为与关键字 filenames 相对应的的变量（即变量 thesefiles）的赋值情况，遗憾的是，这些命令最多只能说明这些变量是否已经定义。但这并不同于变量是否存在。如果变量 myfiles 没有定义，那么 N_elements，Keyword_Set 都无法区别上述两种调用程序方法的差异。

为了帮助人们知道一个关键字是否已经使用，IDL5.0 引入了新的函数 arg_present。arg_present 将返回 1，如果参数（变量）是以关键字或参数传递的（换句话说，关键字和参数被使用了），并且变量以引用的方式传递。后面一点极其重要。否则，arg_present 将返回 0。如果关键字或参数已经使用了，但参数还以传值方式传递，这时 arg_present 将返回 0，就好像参数没有被使用或者并不存在一样。在使用这个新命令时应该倍加小心。

编写 IDL 函数

函数的定义和过程非常相似。也就是说，函数的参数和关键字的定义与过程中的一样。但也有两点不同：（1）函数定义以 Function 而不是以 pro 开头；（2）函数通常返回一个单一的、特定的 IDL 变量给函数调用者。被返回的变量被称为函数的返回值，它可以是任何一种有效的变量类型或结构。例如，它可以是一个很大的结构变量。实际上，这意味着在所有函数的 return 语句中必须有一个参数来保存函数的返回值（一个没有 Return 语句的函数将隐性地返回零）。

例如，为了把过程 ImageOut 改成一个函数，只需做一点改动，如下所示。变化部分已经用粗体字标出。

```
Function ImageOut, lookhere, colortable = thiscolortable, Scale = scaleit, $
    filename = thesefiles
Numparams = n_paras ()
If numparams eq 0 then cd, current = lookhere
Cd, lookhere
Thesefiles = findfile (‘*.img’, count = numfiles)
Print,’ number of file found:’, numfiles
For j = 0, numfiles-1 do begin
    Openr, un, thesefiles[j], /get_lun
    Image = bytarr (152,512)
    Readu, lun,image
```

```
        Free_lun, lun
        If Keyword_Set (scaleit) then $
            Thisimage = bytxl (image, top = 199) else $
            Thisimage = image
        Loadct, thiscolortable,l ncolors = 200
        S = size (image)
        Window, /free, xsize = s[1], ysize = s[2]
        Tv, thisimage
    Endfor
    End
```

在 IDL 中，函数的调用语法不一样。函数总是会返回一个值，因此把用来接收返回值的变量放在等号左边，函数调用名放在等号右边，而函数的所有参数和关键字都放在括号内，如下所示：

```
IDL> thisvalue = ImageOut ('c:\data', filenames = mydatafiles)
```

由于函数 ImageOut 没有 Return 语句，所以隐性地返回值为 0。

但是如果希望 ImageOut 返回打开和显示的文件的数目，可以这样编写程序。更改的地方已经用粗体字标出：

```
    Function image, lookhere, colortable = thiscolortable, Scale = scaleit, $
        filenames = thesefiles
    Numparams = N_Params ()
    If numparams eq 0 then cd, current = lookhere
    Cd, lookhere
    Thesefiles = findfile ('*.img', count = numfiles)
    Print, 'number of files found:', numfiles
    For j = 0, numfiles-1 do begin
        Openr, lun, thesefiles[j], /get_lun
        Image = bytarr (512,512)
        Readu, lun, image
        Free_lun, lun
        If Keyword_Set (scaleIt) then $
            Thisimage = bytscl (image,top = 199) else $
            Thisimage = image
        Loadct, thiscolortable, ncolor = 200
        S = size (image)
        Window, /free, xsize = s[1], ysize = s[2]
        Tv, thisimage
    Endfor
    Return, numFiles
    End
```

如果想要显示和打印出某个特定子目录下的所有图片文件的文件名，可以键入如下代码：

```
    IDL> numfiles = ImageOut ('c:\data', filenames = mydatafiles)
    IDL> for j = 0, numfiles-1 do print, mydatafiles[1]
```

不管有 10 个还是 100 个文件，这段代码照样运行得很好。

更多的函数是用来获得对变量操作的结果。例如，想得到一个矢量或一组数组的平均值，可以这样在文本编辑器中编写 average 函数：

```
Function average, data
Averageavalue = total (data)/n_elements (data)
Return, averagevalue
End
```

调用形式如下：

```
IDL> thisdata = [3,5,6,2,9,5,4]
IDL> avg = average (thisdata)
IDL> print, avg
4.85714
```

函数的返回值可以作为另一个过程或函数的参数，所以也可以把上面的三行代码写为一行：

```
IDL> print, average ([3,5,6,2,9,5,4])
4.85714
```

方括号和函数的调用

当调用一个 IDL 函数时，可能会有些迷惑。要分辨是函数调用，还是引用数组的下标，仅仅是通过检查 IDL 代码是很困难的。比如，上述代码就显示出这方面的问题。如果没有其他信息，Average 也可以认为是一个 IDL 变量。（一个函数和一个变量是可能同时名取为 average 的，但这是一种不明智的做法。）

为了增加代码的可读性，Research System 公司在 IDL5 中引入了方括号来指定下标变量。从而就可以这样定义名为 average 的变量了，如下所示：

```
IDL>average=[4,6, 3,8,2,1]
```

它还可以这样引用：

```
IDL>number=average [3]
```

这种语法就可以将变量 average 和函数 average 区分开来了，因为函数 average 要求其参数用括号括起来。

用 Forward_Function 命令保留函数名

为了确保 IDL 编译器能区分是函数调用而不是带下标的数组，可以用 Forward Function 命令来预先声明函数名称并为之保留。例如，如果想保留上述函数 Average，可以键入：

```
IDL>Forward_Function Average
```

注意：保留的函数名不必用引号括起。

现在，如果在编译程序模块中，IDL 编译器碰到 average，并且跟有参数，那么它就认为这是函数调用而不是对数组元素的下标引用。

使用程序控制语句

与其他程序语言一样，IDL 程序可以通过控制语句来控制程序语句的执行顺序。大多数情况下，一个程序控制语句包括布尔测试（通常是程序变量的一种表达式）、当测试为真时执行的命令和为假时的执行命令。或者，它包括一个计数器和反复执行某个语句的方式。重复执行某个语句的情况，取决于计数器的值。

IDL 中表达式的真和假

在 IDL 中，一个表达式的真或者假取决于该表达式中的数据类型。真的情况可以概括如下：

奇数，非零的字节型、整型和长整型；

非零的浮点型、双精度型和复数类型（包括单精度和双精度）；

非空的字符串类型；

在 IDL 中，通过布尔逻辑运算为非真的表达式，都为假。

注意，指针和对象不是通过真或假来衡量，而是分别由 Ptr_valid 或 Obj_Valid 命令来检查是有效还是无效。

第一种类型的例子是典型的 IF…THEN… ELSE 控制语句。在 IDL 中是这样表述的：

```
IF test THEN statement1
```

其中，test 通常是一个布尔表达式，stattement1 是 IDL 命令。

注意，test 必须总是数值并且必须是在表达式调用前已经定义。例如，下面的表达式就会出错，因为变量 coyote 没有定义 。

```
IDL> IF coyote EQ ‘tricky’ THEN Print,    ‘Missed him!’
```

第二种类型的例子是典型的 FOR 循环控制语句。在一个 FOR 循环语句中，语句反复地被执行，直到计数器达到某个预设定的值。如：

```
FOR j=0,10, DO statement1
```

这里，j 是计数器，statement1 是将要执行的 IDL 命令.

例如下面的例子中，Print 命令将执行 11 次,打印数值 0 到 10.

```
IDL> For j=0,10, Do Print, j
```

将多个语句处理成单个语句

大多数控制语句（见上面）在语法中，要求是单个语句，因为对 IDL 解释器而言，所有的命令看上去都是一个命令。但这常常不是所希望的。例如，在某种测试的基础上，如果这种测试是真，就执行 4 个语句，如果为假就执行 15 个语句。

在 IDL 中，通过使用 BEGIN 和 END 语句块，可以清楚地让 IDL 解释器将多个语句看作是单个语句。语句块以 BEGIN 始，以 END 结束。

例如，若在 FOR 循环中执行多个语句，上面的循环应写做：

```
FOR j=0, 10 DO BEGIN
    Statement1
    Statement2
    Sratement3
    …
END
```

当与 BEGIN 相匹配的多个 END 语句结束时， 程序变得难于阅读和交互操作。这是因为在 IDL 中，使用控制语句的地方都可以以 END 语句结束。例如，在上述控制语句中，通常用 ENDFOR 命令代替 END 命令，因为可以用 ENDFOR 命令结束 FOR 循环。

```
FOR j=0,10 DO BEGIN
    Statement1
    Statement2
    Statement3
    ...
```

```
ENDFOR
```

有效的 END 语句可以是 ENDIF、ENDELSE、ENDFOR、ENDWHILE、ENDCAS 和 ENDREPEAT，下面将更详细地讲述这方面的知识。

If…Then…Else 控制语句

IF…THEN…ELSE 控制语句是应用最广泛的控制语句之一。它是以布尔测试或可以用“真”和“假”来衡量的表达式为基础的。(详细参见 220 页中的“IDL 中表达式的真和假”)下例是一个简单的例子。

```
If (num GET 10) THEN index =2 ELSE index=4
```

在 IF…THEN…ELSE 中，ELSE 是可有可无的，上述的语句也可以表示为：

```
If (num GET 10) THEN index =2
```

在使用多行语句时，IF…THEN…ELSE 控制语句的语法就比较灵活了。注意，这多行 IDL 语句对 IDL 编译器来说必须是个简单 IDL 命令。如果在 IDL 命令行中,要编写一个多行命令,就必须用行连续符和行连接符。例如,IF…THEN…ELSE 控制语句可以写作：

```
IDL>IF (num GT 10) THEN BEGIN $
    Index=2&$
    Num=0&$
ENDIF ELSE THEN BEGIN $
    Index=4&$
    Num=-10&$
ENDELSE
```

如果代码在文件中，那么就完全没有必要使用行连续符和行连接符了。例如，在 IDL 主程序的过程或函数中,代码可以写成：

```
IF (num GT 10) THEN BEGIN
    Index=2
    Num=0
ENDIF ELSE  BEGIN
    Index=4
    Num=-10
ENDELSE
```

与在 IDL 解释器中相反，对 IDL 编译器而言，BEGIN 和 END 语句已经暗示行连续符和行连接符。但是，在 IDL 编译器中，IF...THEN...ELSE 的语法也不是任意的。

例如，一些程序员喜欢将 BEGIN 语句和 END 语句对齐，这样可以一目了然地知道程序所指的什么，比如:

```
IF (num GT 10) THEN $
BEGIN
    Index=2
    Num=0
ENDIF ELSE
BEGIN
    Index=4
    Num=-10
ENDELSE
```

但是在 IDL 中，代码不能那样写，因为在编译中不能恰当地分开 IF…THEN…ELSE 控制语句，从而不能将其作为一个独立语句。如果想把用上述格式，就必须在代码中应用行连续

符来连接，如下：

```
IF (num GT 10) THEN $
BEGIN
    Index=2
    Num=0
ENDIF ELSE $
BEGIN
    Index=4
    Num=-10
ENDELSE
```

条件表达式

在 IDL5.1 中，引入了一种新的条件表达式(?:)。用它可以代替 IF…THEN…ELSE 控制语句。如变量 num 大于或等于 10, 设变量 index=2，否则设变量 index=4，可表述为:

```
Index=(num GE 10)? 2:4
```

在这个语句中,先进行条件判断(num GE 10)，如果判断结果为真,变量 index 就设为问号右边和冒号左边的变量（或表达式）的值；如果为假，变量 index 就设为冒号右边的变量(或表达式) 的值。

FOR 循环控制语句

FOR 循环运用计数器来多次执行一个或多个语句，如：

```
A =1
FOR j=1, 10 DO BEGIN
    Print, j
    A = a * j * 2
ENDFOR
```

在这种情况下,计数器 j 从 1 开始直到 10，这个语句将执行 10 次。如果希望计数器 J 的步长不是 1，就应当明确指定步长。例如，让 j 的步长为 2，可参见下例:

```
A =1
FOR j=1, 10 DO BEGIN
    Print, j
    A = a * j * 2
ENDFOR
```

WHILE 循环控制语句

WHILE 循环控制语句循环不断地执行一条或多条语句，只要判断条件为真。如:

```
WHILE (number LT 100) DOES BEGIN
    Index =amount * 10
    Number =number +index
ENDWHILE
```

在进入 WHILE 循环之前，首先进行条件判断。

REPEAT...UNTIL 循环控制语句

REPEAT...UNTIL 循环语句与 WHILE 循环相似，不同之处在于，REPEAT...UNTIL 循环语句在循环末尾进行条件判断，而不是在循环的开始处。下面是一个简单的例子:

```
REPEAT BEGIN
    Index =amount * 10
    Number =number +index
ENDPEP UNTIL number GT 100
```

CASE 控制语句

有时会遇到一个判断条件，其判断结果可能有一些不同的值，无法用真和假来表示，这时就应该使用 case 语句。

case 语句常用于响应程序的不同的事件从而执行相应的代码。下面的代码是一个响应按钮事件的例子:

```
; What button caused the event?
Widget Control, eventide, Get Value=button Value
; Branch based on button value
CASE button Value OF
‘Sober’:   TVScl, sobel (image)
‘Roberts’: TVscl, Roberts (image)
‘Boxcar’:  TVScl, Smooth (image, 7)
‘Median’:  TVScl.  Median (image, 7)
‘Original Image’  :TVScl, image
‘Quit’:    WIDGET Control, event. Top, / Destroy
Elae: ; Do nothing at all
ENDCASE
```

注意，当判断条件与所列的条件不匹配时，ELSE 语句定义了默认操作，在这里 ELSE 语句让事件进入事件处理代码中。在使用 ELSE 时，后面必须有一个冒号。实际上，ELSE 语句不一定得出现在 case 语句中，但是如果判断条件与所有列出的条件都不匹配时，程序就会产生错误。

如果在 case 语句中使用 BEGIN 和 END 语句，一定要小心，END 语句既可以是块的结束语句，也可以是 case 的结束语句。有时多个 CASE 语句可以写成如下所示：

```
CASE this TEST OF
    0:x=5
    1: BEGIN
        X =5
        Y =x *2
    END
    ELSE: x=0
ENDCASE
```

注意，当有一个 case 语句满足条件时,case 总是跳出来。换句话说，一旦一个 case 语句为真，程序将跳转到 case 语句的下一条程序语句。这和在 C 程序不一样。

GOTO 控制语句

和所有的计算机高级语言一样，IDL 也有 GOTO 语句。IDL 程序员只有在的确需要时 ，才使用 GOTO 语句。任意使用 GOTO 语句会使简单的程序变得复杂。

GOTO 语句指定一个程序标识符，当程序执行到 GOTO 语句时，程序就跳转到程序标识符所在的程序行。比如，可以用 GOTO 语句来打断一个 FOR 循环，见下：

```
FOR j =0, n DO BEGIN
    Index= j * ! PI * 5.165 * thisTESTValue
    IF index GT 3000.0 THEN GOTO, jump out
ENDFOR
Jump out: Print, index
```

程序标识符后面要有一个冒号。在程序标识符所在行应当是可执行语句，如这个例子所示，但也可以不是可执行语句。如果该语句无效，IDL 程序将跳转到标识符下面的一个可执行语句。

错误处理控制语句

在 IDL 中，有多个错误处理语句，通常使用到的有：ON_IOError,ON_Error 和 Catch。

ON__IOError 控制语句

ON__IOError 语句与 IDL 中的 GOTO 语句相似，它可指定一个标识符，当出现任何输入输出错误时，程序就跳转到标识符所在的行。

例如，ON__IOError 语句可以在读取数据时进行相应的处理，如下所示：

```
ON_IOError,    Problem
OpenR, lun, filename, / Get_Run
Data =Bytarr 9256,256)
ReadU, lun, data
Free_Lun , iun
...
RETURN
Problem: Print, 'Problem reading data.    Returning...'
IF N_Elments (lun) NE 0 THEN Free_Lun ,lun
RETURN
END
```

在 GOTO 语句中一样，注意程序标识符后的冒号，程序标识符所在行不一定必须是有效的 IDL 语句。

ON_Error 控制语句

在程序运行过程中出现错误时，ON_Error 控制语句指明了 IDL 应该怎样做。与其他的控制语句不一样，ON_Error 控制语句并不执行一个新的 IDL 语句，而是指出错误出现时应采取的措施。ON_Error 命令的有效参数为 0，1，2，3。表 11 表明了程序出错时采取的措施。

表 11　程序出错时 ON_Error 控制语句采取的措施

值	行动
0	当程序内容模块引起错误时立即停止。这是错误的行动。
1	立即停止返回主程序。主程序在命令进入程序行的地方。
2	立即停止返回程序模块称为模块错误。
3	立即停止返回程序模块，登记环境（这可能不是当前模块。）

当程序出现错误时，IDL 新手通常不之所措。因为默认的行为（ON_Error 设为 0）是在程序出错的地方停止下来。由于受 IDL 提示的影响，许多用户认为错误出现在 IDL 主程序中。当用“HELP”命令时，他们非常失望，因为所有的变量已经消失，不存在了。

用户实际看到是位于已经破坏了的程序中的变量。使用命令 RETALL（返回 IDL 主程序）将会恢复消失的变量，从而获得它们的实际值。

RetAll 是用户最重要的工具。在程序不能正常退出时，有经验的用户常常自动用 RetAll 命令来恢复。特别是在编写模块程序时。如果忘记了这个重要的命令，程序常常变得让摸不着头脑，尤其是在编写 IDL 程序的时候。

为了避免用户在使用这个程序时感到迷惑，最好把 On_Error,1 加到正在调试的程序中。如果程序出错了，IDL 将隐性地执行 RetAll 命令，从而返回到 IDL 主程序上。

Catch 控制语句

第三种控制语句，即 Catch 控制语句，可能是最有效的，它与 GOTO 语句相似，但也不完全一样。Catch 控制语句调用如下：

```
Catch,error
```

这里的 error 是一个变量名。（当然，变量名称可以取自己喜欢的）。当执行到这条语句时，IDL 为该特定程序模块记录了一个 Catch 错误处理语句，同时，将该变量（即 Error）设为 0。

若在程序运行过程中出错，且在该程序中有相应的 Catch 错误处理语句，则该变量被赋予相应的错误值（每个错误都有与之相对应的数值），然后程序跳转到 Catch 语句后的第一条语句。

在实际操作中，CATCH 语句行包括一块错误处理代码。例如，要读一个文件，这不可避免会出现许多错误，就可以用 Catch 语句来进行避免错误的保护。

```
Catch ,error
IF error NE 0 THEN BEGIN
    Catch, / Cancel
    Print, ‘Problem reading data file .Returning…’
    IF N_Elements (lun) NE 0 THEN Free_Lun ,lun
    RETTURN
ENDIF
OpenR, lun, filename, /Get_Lun
Data=BytArr (256,256)
ReadU, lun, data
Free_Lun, lun
Catch, /Cancel
```

注意，Catch 错误在任何时候都可以忽略。上个特殊的例子中，在错误语句代码中的第

一行就忽略了所出现的错误。如果在错误保护程序中也有错误，这绝对是不允许的。后面，程序可能会产生新的错误，并且可能会有任意多个 catch 错误，但是在任何时候，只有一个错误处理语句。注意：catch 语句也可以用来捕获输入输出错误。那和 On_INError 有什么关系呢？

错误处理语句的优先级

错误处理语句有个优先级关系，程序模块中将会出现的命令取决于这个优先级关系和在该程序模块中使用的错误捕获语句。可以根据这个关系，找到恰当的错误处理语句。它们的关系如下所示：

On_Error→catch→on_error

如果在程序模块中使用了 On_IOError，那么 On_IOError 将屏蔽其他的错误处理语句。如果使用的是 Catch 语句，那么它将屏蔽 On_Error 语句。

要是没有任何错误处理语句，那么默认的 On_Error 将会在程序模块中起作用。

编译和执行 IDL 程序模块

在 IDL 编译程序中有三个可执行的命令，它们是.Run、.Rnew 和.Compile。其中，.Run 命令是最早的可执行命令。早期版本的 IDL 中，是没有过程和函数的，只是一些命令的简单组合。.Run 命令是用来编译和执行那些称为 IDL 主程序的命令组合。

自从主程序有了自身的变量后，内存的分配就显得较为重要了。尤其是其他程序编译后，每个主程序都有自己的变量。这样，就引入了.Rnew 命令。Rnew 命令与.Run 命令很相似，不同之处在于，Renew 命令在主程序编译和运行之前将删除所有已经存在的变量。

后来，IDL 中引入了程序和函数的概念。实际上，.Run 和 Rnew 也用来编译新的程序模块，仅是编译而不运行。IDL 新手对于.Run 命令几乎什么都不运行（除了偶尔运行主程序外）感到很迷惑。其实，在 IDL4 中引入的.Compile 命令所做的，是以前版本中.Run 命令所做的事，即编译过程和函数。

如今，人们已经普遍接受用.Compile 命令编译过程和函数，用.Run(.Rnew)命令编译和运行主程序。编译名为 Cindex.pro 文件可以用以下几种形式。

```
IDL>.Run cindex
IDL>.Rnew cindex
IDL>.Compile cindex
```

注意，使用这三个命令时，文件名不必用引号括起，也没有必要用扩展名.pro（如果文件中用了一个不同的文件扩展名，就必须指明文件名和扩展名）。几乎所有的 IDL 文件以 Pro 为扩展名。在不同的操作系统上，扩展名是区分大小的，比如说 Unix，文件名称也失去分大小写的。一般地，在区分大小写的操作系统上，最好将文件名称都写成小写，这对文件的自动编译是非常有必要的。（关于自动编译的详细信息请参见 230 的程序编译和自动运行规则）。

要运行一个已经编译的程序，只要 IDL 命令输入行中健入程序模块名。例如，用上述已编译过的程序模块，应健入：

```
IDL>CIndex
```

在 IDL 内部，模块名称是不区分大小写的。

程序编译规则:

对于一个文件中有多个程序模块，程序编译规则对他们的顺序作出了要求。文件中的程序模块是一个接一个地编译，直到满足某个条件。下面的规则列出了这些条件。

规则 1：编译到主程序后，编译就会停止，接着编译和运行主程序。

这个规则表明了在一个文件中,只允许有一个主程序模块。如果要编译所有的程序模块，主程序模块必须是该文件中最后一个程序模块。

规则 2：编译到与文件同名的程序模块时，文件将停止编译。在这种情况下，与文件同名的程序模块被编译后，立即执行该模块。

这就是说，如果想编译文件中的所有程序模块，与文件同名的程序模块应当放在最后。换句话说，如果最后一个程序模块是一个名为 ImageOut 的函数，文件名应为 ImageOut.pro。

规则 3：编译到文件末尾或适合其他规则时，文件将停止编译。

这个规则表明，如果此文件中没有一个主程序或一个与文件同名的程序模块，文件中所有的模块将会被编译，并且不会运行任何一个程序模块。

程序编译和自动运行规则

当过程或函数出现在 IDL 命令，或 IDL 命令行，或 IDL 代码中，它们会自动地被编译和运行。例如：

过程或函数所在的文件存在于当前工作路径或由！PASH 系统变量指定的路径中。

过程或函数名与文件名相同。在区分大小写的操作系统中，如 UNIX，文件名必须用小写。

这就意味着，在实践中，那些非常重要的、经常被其他程序调用的程序模块应该放在自己的文件或与之同名的文件中。其他任何模块应位于主程序模块之前，或者它们是主程序的辅助性模块。

注意，IDL 的系统过程，如 Plot，Surface 等，的优先级比其他命令高，因此，应当尽量避免自己的程序与 IDL 内置命令同名。

特殊编译命令

在 IDL 程序模块中，有两个特殊的编译命令。可以在程序模块中调用它们来编译其他的程序模块，这就是 Resolve_Routine 和 Reserve_All。

Resolve_Routine 命令，以 IDL 程序模块名作为参数，编译与之同名的文件。换句话说，它好比是使用.Compile 命令来编译文件。Resolve_Routine 命令的优点在于可以用在 IDL 程序模块里来编译其他程序模块，而 Compile 命令只适用于 IDL 命令行。例如：要编译 Cindex.pro 文件中的所有程序模块，可以健入：

IDL>Resolve，‘Cindex’

如果程序模块是一个函数而不是过程，应该使用 IS_Function 关键字，如：

```
IDL> Resolve_Routine‘Congrid’,  /IS_Function
```

编译时，不管它以前是否已编译过，程序仍然将被编译。

Resolve_All 命令的功能与之很相似，不同之处在于，它不是仅编译一个特定的文件。Resolve_All 命令将交互式地搜索 IDL 内存中任何未编译的程序模块并同时编译它们。Resolve_All 也遵循自动编译的一般规则，这意味着如果模块不能自动编译，Resolve_All 命令将无法查找到该模块，当然也就无法编译了。

若准备将一个应用程序的 save 文件运行在其他计算机上，因为其他的计算机上配置可能不一样，这时，Resolve_All 命令非常方便。例如，如果应用程序名为 BigApp，可以用 Resolve_All 命令来编译，并将与这个程序相关的库文件和程序文件也一起编译。如下所示：

```
IDL>.Compile BigApp
IDL>Resolve_All
IDL> Save, /Routines, File='bigapp.sav'
```

然后，其他用户只需要恢复这个 save 文件即可，同时所有的程序模块就已经编译完毕，可以使用了。而用户的计算机上就不必有源代码文件。如下：

```
IDL>Resolve 'bigapp'
IDL> BigApp
```

注意，用 Save 命令编译的过程和函数在 IDL 的其他版本中未必能够正常使用。（但 Save 文件中的数据和变量在不同 IDL 的版本是一致的。）因此，一般来讲，用户必须用创建 save 文件版本的 IDL 来恢复 save 文件。

第九章　编写 IDL 程序

本章概述

尽管 IDL 是一个程序语言，但在官方的 IDL 手册里可能找不到太多如何编写 IDL 程序的技巧。当然，这并不意味着只有一个正确的方法。任何人，只要了解那些和我一样的 IDL 程序员，都知道一个优秀的 IDL 程序员和一个不是那么优秀的程序之间的差距是很明显的。作为和 IDL 程序初学者长期打交道的人，作者见过了很多不是很好的程序。

出现这个问题，可以肯定是由于对 IDL 信息缺乏了解。因为大多数这样的人毕竟是科学家，而不是电脑程序员。他们很聪明，并且在干自己的本行，但他们并不是去编写优秀的电脑程序。

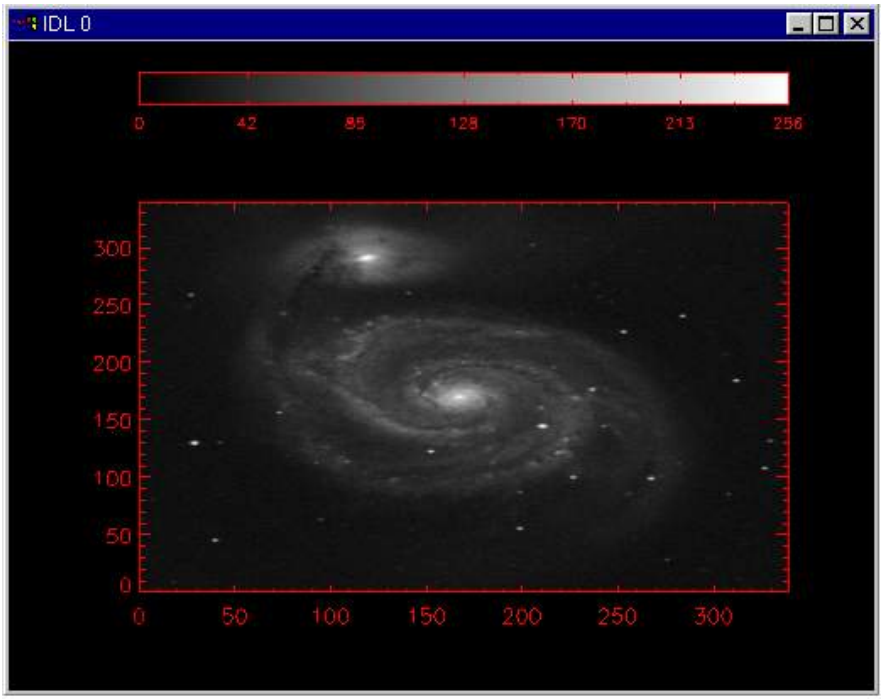
如果只要遵循几条基本的原理，他们编写的程序就会很出色，而且对他们自己也更有好处。因此，本章就是阐述这几个原理。

本章的任务就是展示怎样编写一个合理的复杂图形演示程序。而且这个程序能从 IDL 命令行上调用。同时也希望该程序能够将数据显示在可改变大小的图形窗口中，从 IDL 命令行上直接打印，或者直接传送到 PostScript 文件中。此外，这个程序能够轻松将数据文件保存成 GIF 或 JPEG 文件。即使这写程序采用不同的颜色，这个程序应该是具有颜色敏感功能，而且能够和其他程序共存。而且，在程序中增加一个图形界面应该很简单，即使那些对该程序一知半解的用户也可以容易掌握。

再者，这个程序应该维护简单，易扩展。简而言之，该程序应当以模块化方式来编写。尽管可能不清楚为什么要这样做，这里还是有必要介绍一下面向对象编程的概念，特别是自身模块和方法的概念。如果已经了解这个程序的原理 ，那么就能够毫无困难地理解在 IDL5 中引进的对象类和方法方面的复杂知识。

基本的 ImageBar 程序

这个基本程序的思路很简单，即显示图像，在图像周围显示坐标轴。并在此图像的上方绘制一个颜色栏，用来表示图像色彩与其值的相关性。将这个程序取名为 Imagebar。结果与图 80 相似。



图片 82　此程序显示了一个由轴环绕的图象，图象上方的色彩栏标出了图像值的范围

这个程序的基本框架很简单。图像用一个类似于 Tvimage 的命令来显示。（程序 Tvimage，如果带关键字 Position 就可用来定位显示图像。此外，它还可以根据图像输出设备的不同而输出不同大小的图像。详细信息请参阅 72 页的“用归一化的坐标来定位图像”。）坐标轴是用带关键字 NoData 的 Plot 命令绘制，色彩栏用 Colorbar 命令显示的。（程序 Coloebar 是并与本书配套使用的程序之一。）

注意，如果是在 24 位颜色环境中运行这个程序，色彩分解应处于关闭状态。详细信息请参阅 87 页的“在 24 位显示器上指定分解后的颜色”。要使颜色分解处于关闭状态，键入：

```
IDL>Device, Get_Visual_Depth=thisDepth
IDL>IF thisDepth GT 8 THEN Device, Decomposed=0
```

如果愿意，也可将上述代码添加到下面的 Imagebar 程序中。

新建一个文本编辑窗口，并且给程序命名为 ImageBar。其定义如下所示：

```
PRO ImageBar, image
```

其中，image 是要显示的二维图像。

作者喜欢编写简单易懂程序，即使那些不清楚程序如何工作的用户也能够看懂。如果那样，即使用户不知道要将图像作为程序的第一个参数，程序也可以让用户有机会打开并读取一个图像文件。

编写这个程序的目的之一就是能够在 Z 图形缓冲区或在 PostScript 文件中输出显示图形。那两个图形输出设备既不支持窗口，也不支持组件程序。因此，程序要生成一个命令，类似于 Window 或一个在支持窗口的图形设备中创建窗口的命令。

为达到这个目的，可以使用!D.Flags 系统变量。如果在这个变量的第八位的值不为 0，那么当前图形设备就支持窗口。用 256 和这个系统变量进行逻辑 AND 运算，如果返回值为 0，那么当前的图形设备就不支持窗口。

这些代码让用户有机会在支持窗口的图形设备上打开图像文件。从这个意义上说，参数 image 可以说是一个可供选择的参数。如果设置不支持窗口，参数 image 就是一个必须的参数了。增加了逻辑判断的代码如下：

```
IF N_Params( ) EQ    0 THEN BEGIN
    IF (!D.Flags AND    256)    NE 0 THEN BEGIN
```

```
        Image    = GetImage ( Cancel= canceld,    'm51.dat',$
              Xsize=340,Ysize=440)
        IF    CANCELD then return
    ENDIF else begin
        Message, 'Please supply an image argument,',/Continue
        RETURN
    ENDELSE
    ENDIF
```

注意，GetImage 命令用来打开图像文件。程序 getimage.pro 是和本书配套使用的文件之一。GetImage 是用来读取无格式数据文件的一个通用程序。它是一个对话框程序。详细信息请参阅 291 页的“创建模式对话框”。

下一步，就是检查图像参数是必须是一个 2 D 图像。注意，当出现错误时，用 Message 命令而不是 Dialog_Message 命令来提示错误信息。这是因为 Dialog_Message 是个组件命令。同时在 Z 或 PostScrippt 设备里用 Dialog_Message 会导致错误。

```
    S= Size (image)
    IF S [0]NE 2 THEN BEGIN
        Message, 'Image argument must be 2D.',    /Continue
        RETURN
    ENDIF
```

最后，要定义 X 轴和 Y 轴矢量的默认值。（在现实的应用程序里，这些矢量可能代表确切的物理含义，并且作为固定参数传递到程序中。但是，如果在这里也这么做，只会使例子更加复杂，而对增加程序原理的理解毫无裨益。）

```
    X=FindGen ( [1])
    Y=FindGen( [2])
```

下一步，就可以绘制图形了。 这个图像显示在窗口的偏下部分，而窗口的上部分用来显示颜色栏。

注意，窗口不是专门为显示图形而建立的。这是又能输出到 Postscript 文件，又能输出到可改变大小的窗口的图形程序的重要特点。如果程序要能够在任何图形显示窗口或设备上正常工作，应当如下所示：

```
    imagePos=[ 0.15,0.15.0.9,0.75]
    TVImage, image, Position=imagePos
```

接着，运用带关键字 NoData 的 Plot 命令图像周围画轴。记住，要使用相同的位置参数，以确保坐标轴的范围与图像范围一致。此外，必须设置关键字 NoErase，否则 Plot 命令会擦除刚才显示的的图像。键入：

```
    Plot,X,Y,    /NoDATA,xstyle=1,Ystyle=1,Position=imagePos,    /NoEase
```

最后，在图像上方绘制颜色栏。通过变量 imagePos 可以计算出颜色栏的位置，还应该稍微地延伸图像的长度，并把颜色栏定位在图像上方。键入：

```
    barPos=[imagepos[o],(imagepos[3]+0.15,imagepos[2],$
    (imagepos[3]+0.15)+0.05]
    Colorbar,Position=barpos
    END
```

整个 ImageBar 程序的源代码如下所示。（在与本书配套使用的文件中，可以找到本程序的源代码，文件名称为 ImageBar.1.pro.）

```
    PRO ImageBar ,    image
    IF N_Params ( ) EQ 0 THEN BEGIN
        IF (! D. Flags AND 256 )NE 0 THEB BEGIN
```

```
        Image = Get Image ( Canceled, 'm 51.dat'. $
            XSize=340, Ysize=440)
        IF canceled THEN RETURN
    ENDIF ELSE BEGIN
        Message, 'please supply an image argument.',   ?Contimue
        RETURN
    ENDELSE
    ENDIF
    S =Size (image)
    IF s [0] NE 2 THEN BEGIN
        Message, 'Image argment must be 2D.' , /Continue
        RETUEN
    ENDIF
    X =FIndGen (s [1] )
    Y= FIndGen (s [2] )
    Image Pos = [0.15,   0.15,     0.9,   0.75 ]
    TVImage, image, Position =image Pos
    Plot, x, y, /NoData, XStyle =1, YStyle=1,$
    Position = imagePos,   / NoErase
    BarPos = [imagepos[0],   (imagepos[3]+0.15), imagepos[2], $
        (immagepos [3] +0.15)+0.05]
    Colorbar, Position =barPos
    END
```

保存并编译这个程序，如果程序编译失败，在继续之前应该修订错误。

```
IDL>   .Compile ImageBar
```

程序编译完后，打开银河系图像 M51 和程序一起运行，键入：

```
IDL>image =LoadData(12)
IDL>Window
IDL>ImageBar, image
```

如果程序在运行时出现程序崩溃，需要对程序作一些修改。在每次修改程序时，要保存并重新编译文件。

注意，这个程序可以在任何尺寸的窗口中显示图像，如下所示：

```
IDL>Window, /Free, XSize=500, YSize =250
IDL> ImageBar, image
IDL>Window, /Free, XSize=300, YSize =600
IDL> ImageBar, image
```

给程序 ImageBar 增加一个“先擦除”功能

注意，如果只是用鼠标改变图形窗口（而不是新建一个窗口），并重新执行 ImageBar 命令，这时图像绘制在原有窗口的上面。这与 TV 或 TVScl 命令的效果相似，但在这里并不合适。在这个程序的 TVImage 命令之前，可以增加一个永久性的擦除命令，但是在每次程序运行时都将擦除窗口里的内容。在显示设置上，这可能不是问题，但是如果是在创建 PostScript 文件，这个永久性的删除命令就会将输出结果输出到第二页中。这样，每次创建一个 ImageBar 程序的 PostScript 文件，就有输出两页，第一页是空白的。

为避免这一点，可以把擦除功能作为可选择的关键字增加到程序中。通过修改 ImageBar

程序的定义语句，可以给擦除关键字定义如下：

PRO ImageBar, image, **EraseFirst=erasefirst**

这个 EraseFirst 关键字可以设置也可以不设置，即有双重属性，这也意味着可以用 Keyword_Set 命令来检查它是否存在。（详细信息请参阅 213 页的“处理具有双重属性的关键字”。）在 TVImage 命令之前增加该行语句，如下所示。增加部分已用粗体字标出。

```
ImagePos = [0.15,  0.15.  0.9,  0.75 ]
IF Keyword_Set (eradefurst) THEN Erase
TVImage, image, Position =imagePos
```

保存并重新编译 ImageBar，并重新执行该程序，如下：

```
IDL> ImageBar, image  ，/EraseFirst
```

注意，这是可以用鼠标来改变图形窗口的大小，并重新执行 ImageBar 命令来重画图形。当进入到下一章时，就可以用一个方法自动处理这个过程。因此，当窗口改变大小时，图形就自动重画。

向 ImageBar 程序增加颜色敏感功能

颜色识别，是作者定义的一个术语，就是程序可以确切地知道哪部分颜色表可以使用。（详细信息请参阅 67 页的“用颜色表分段表示图像”）Xload 或 Xcolors 是颜色识别的最好例子。这两个例子分别通过使用关键字 Ncolors 和 Bottom 来计算出当前应用程序使用多少色彩以及这些色彩索引的开始位置。

```
IDL> Xcoloes, Ncolors=100 Bottoms=100
```

颜色识别命令可以通过不同方式一起使用来开发整个颜色识别的 IDL 应用程序。要使 ImageBar 程序能识别色彩。首先，给程序中的关键字 Ncolors 和 Bottom 下定义，如：

```
PRO ImageBar, image, EraseFirst=erasefirst, Ncolors=ncolors, Botttom =bottom
```

下一步，检查关键词，若需要定义它们的默认值，键入如下：

```
IF N –Elements (Ncolors)EQ 0 THEN $
    Ncolors = ! d. .Table_Size
IF N_Elements (bottom)EQ 0 THEN bottom=0
```

这时，再定义绘制颜色这个关键字或许是个好主意。现在，并将它作为彩色表的最顶层颜色。然后将下面一行代码紧放在上述两行代码的后面：

```
drawColor = ncolors –1 + bottom
```

下一步，修改程序 TVImage，使得图像合适地对应到指定的颜色数。键入下面的黑体部分：

```
TVImage,BytScl (image,Top=ncolors-1)+bottom,Position=ImagePos
Plot,X,Y, /NoData,XStyle=1, Ystyle=1,Psition=imagePos, /NoErase, Color=drawColor
```

最后，修改 ColorBar。幸运的是，ColorBar 已经是按照颜色识别的方式写的，只要将 nColors 和 bottom 变量传递到程序中即可，如下所示：

```
ColorBar,Postion=barPos,Ncolors=ncolors,Bottom=bottom,Color=drawColor
```

这时程序 ImageBar 变成如下所示。（在与本书配套的文件中可以找到这个程序的源代码，名称是 ImageBar2.pro）

```
PRO ImageBar ,image, EraseFirst=erasefirst ,Ncolors=ncolors, Botttom =bottom
If N_Params() EQ 0 Then Begin
    IF (!D.Flags AND  256)  NE 0 THEN BEGIN
        Image  = GetImage ( Cancel= canceld,  ‘m51.dat’,
        Xsize=340,Ysize=440)
        IF  Canceld  THEN RETURN
```

```
        ENDIF ELSE BEGIN
            Message, 'Please supply an image argument,',/Continue
            RETURN
        ENDELSE
    ENDIF
    S =Size (image)
    IF s [0] NE 2 THEN BEGIN
        Message, 'Image argment must be 2D.' , /Continue
        RETURN
    ENDF
    X =FIndGen (s [1] )
    Y= FIndGen (s [2] )
    Image Pos = [0.15,   0.15,    0.9,   0.75 ]
    TVImage, image, Position =image Pos
    Plot,X,Y,   /NoData,XStyle=1, Ystyle=1, Position = imagePos,   / NoErase
    BarPos = [imagepos[0],   (imagepos[3]+0.15), imagepos[2], $
    (immagepos [3] +0.15)+0.05]
    Colorbar, Position =barPos Ncoloes=ncolors,Botttom =bottom, Color=drawColor
    END
```

查看这时程序是否能做色彩识别，保存并重新编译，然后键入下面的命令：

```
IDL>LoadCT,1,Ncolors=75,Bottom=0
IDL>LoadCT,3, Ncolors=75,Bottom=75
IDL>window,1
IDL>ImageBar, image, Ncolors=75, Bottom=0
IDL>Window, 2
IDL>ImageBar, image, Ncolors=75, Bottom=75
```

这时，第一个窗口的图像应该是蓝色的，第二窗口是红色的。

要改变第一窗口的颜色，可以键入：

```
IDL>Xcoloes, Ncolors=75, Title='Window 1 Colors'
```

要改变第二窗口的色彩，键入：

```
IDL>Xcoloes, Ncolors=75, Bottom=75, Title='Window 2Colors'
```

注意，这时应该可以单独改变这两个窗口的颜色。如果结果不是这样，也许在程序中还存在错误。仔细检查程序代码以及键入在 IDL 命令行上的命令，然后再试一遍。

给 ImageBar 中的命令传递关键字

在上面的程序中，把变量 ncolors 和 bottom 传递给命令 ColorBar 时，就提出了这样一个问题，即一般来讲，怎样给程序内部的命令传递关键字变量呢。例如，假设想要保存程序 ImageBar 中图像外观比例，就必须为 ImageBar 程序内部的 TVImage 命令设置关键字 Keep_Aspect_Ratio。

要做到这一点，可以为程序 ImageBar 定义一个关键字 Keep_Aspect_Ratio，然后将关键字的值传递给 TVImage 命令。例如，修改 ImageBar 程序定义语句，如下所示：

```
PRO   ImageBar ,image, EraseFirst=erasefirst ,Ncolors=ncolors,Botttom =bottom，$
      Keep_Aspect_Ratio=keepaspect
```

将变量 keepaspect 的值传递给 TVImage 命令，如下：

```
TVImage, BytScl (image, Top=ncolors-1)+bottom,Position=imagePos,$
```

Keep_Aspect_Ratio=Keyword_set (keepaspect)

保存并重新编译程序，键入下面命令，看程序的运行结果如何：

IDL>ImageBar, image, /keep,/Erase

知道为什么可以用缩写的关键字 Keep 和 Erase，而不必拼写完全吗？。如果不清楚，请参阅 211 页的“使用缩写关键字”。

使用关键字继承

定义关键字并将它们的值传递给程序内部的 IDL 命令的方法，简单易懂，但是也很容易就看出这将是一个负担。例如，运行 ImageBar 程序后，可能会觉得使用轴标题会更好，或者加上图像名称，或者将颜色表划分成更多部分。事实上，很快就会觉得对这个程序的修改将是永无止境了。简而言之，如果要定义和检查大量的关键字，一个简单的程序也会就变得庞大和复杂。（仅仅 Plot 命令就有 50 多个关键词可以使用。）

一个好的解决方案就是充分利用 IDL 中的关键字的继承。

在 IDL 中使用关键字的继承就和为程序模块定义一个 Extra 关键字一样简单。（下划线是非常重要的。）例如，可以通过修改 ImageBar 程序的程序定义语句给它增加关键字的继承。如下所示：

```
PRO   ImageBar ,image, EraseFirst=erasefirst ,Ncolors=ncolors, Botttom =bottom，$
    Keep_Aspect_Ratio=keepaspect，_Extra=extra
```

注意，关键字名称是_Extra，而且关键字变量是 extra。这个关键词的名字必须是_Extra，但是关键字变量可以随便命名。

当一个程序模块定义了关键字_Extra 时，IDL 将所有没有定义的关键字以及它们相应的值收集到一起，并且把他们放入一个匿名的结构中。在这个结构里，关键字名称是结构的字段，而关键字变量的值则是相应字段的值。然后这个匿名结构就被赋值给关键字_Extra 的相应变量。（例如，这里是变量 extra。）

例如，假设用下面形式调用 ImageBar 程序：

```
IDL>ImageBar, image,Xtitle='Width (mm)',XcharSize=1.3
```

尽管关键字 Xtitle 和 Xcharsize 对程序 ImageBar 而言是没有定义的，但是它们对于 Plot 命令却是有效的。由于关键字_Extra 已经被定义，IDL 根据这些没有定义的关键字建立一个匿名结构，这个结构定义如下：

```
extra={ Xtitle:'Width (mm)',XcharSize:1.3}
```

接下来的是将这些关键字传递给 Plot 命令。这点可以通过使用已经为 Plot 命令定义了的关键字_Extra。（事实上，所有 IDL 系统程序和函数都定义了关键字_Extra。）修改 ImageBar 程序中的 Plot 命令，如下所示：

```
Plot,X,Y, /NoData,XStyle=1, Ystyle=1,Psition=imagePos,/NoErase, $
    Color=drawColor, _Extra=extra
```

在给 ImageBar 程序和 Plot 命令定义了关键字_Extra 后，保存并重新编译程序。用下面形式调用这个程序时，结果会怎样呢？

```
IDL>ImageBar, image,Xtitle='Width (mm)',XcharSize=1.3
```

如果带关键字 Division，程序 ColorBar 的一个关键字，来调用这个程序，会出现什么样的结果呢？

```
IDL>ImageBar, image,Xtitle='Width (mm)',XcharSize=1.3 $Divisons=5
```

如果是在 IDL5 的环境下运行，什么也不会发生。匿名结构 extra 中的 divisions 字段会被 Plot 命令忽略。如果是在 IDL4 的环境下运行，程序将会崩溃，这是因为在 IDL4 中，IDL 命令不能忽略不适用它的关键字。关键字 Divisions 的值也能通过关键字_Extra 传递到 ColorBar 命令中。可以这样来修改 ImageBar 程序，见下：

Colorbar, Psition=imagePos, /NoErase, Color=drawColor, **_Extra=extra**

保存并重新编译此程序，如下调用它，会发生什么呢？

IDL>ImageBar, image,Xtitle='Width (mm)',XcharSize=1.3, Divisons=5

在这个例子中，ColorBar 命令忽略了关键字 Xtitle 和 CharSize，并接受了关键字 Divisions。然而，Plot 命令恰好相反。这样，通过关键字继承，可以给包含在其他程序的 IDL 命令传递不同关键字，每个命令只接收与自己相关的关键字。

注意，在 ImageBar 代码中，无论是 Plot 命令还是 ColorBar 命令都使用了关键字 Color，然而在 ImageBar 程序中关键字 Color 并没有定义。如果象这样调用 ImageBar 程序会发生什么呢？

IDL>TVLCT,0,255,0,101

IDL>ImageBar, image, Ncolors=100,color=101

实际上，这样一来会使 Plot 命令和 ColorBar 命令调用时带有两个 color 关键字，（一个来自于本身，另一个来自于_Extra 机制）每一个关键字的的值都不相同。这时，IDL 怎样取舍呢？在这种情况下，IDL 就使用通过_Extra 机制传递过来的 color 关键字。

在大多数情况下希望是如此，然而却未必经常希望这样。比如，要是 ImageBar 程序这样调用会出现什么结果呢？

IDL>ImageBar, image,Xstyle=4

对于在某些关键字必须设置的情况，就必须自己来截取_Extra 结构并修改相应的数值了。要从匿名结构中获取个别字段，命令 Tag_Names 就非常有用了。例如，如果想知道 Plot 命令在调用时关键字 Xstyle 是否已设置，可以如下编写 IDL 代码：

```
IF N –Elements (extra) GT 0 THEB BEGIN
    theseFields=Tag_Names (extra)
    index=Where (theseFields EQ  'XSTYLE' , count)
    IF count GT 0 THEB extra.xstyle =extra.xstyle OR1
ENDIF
```

注意，关键字_Extra 继承机制是通过传值而不是引用来传递关键字及其相应值的。通过传值来传递变量也就是说，传递到程序中的只是变量的一份拷贝，而不是变量本身。这样，程序对变量所做的改变只是局部有效，而对全局没有影响。因而，这就使得关键字_Extra 不能作为输出型的关键字。由于这个限制的原因，结果在 IDL5.1 中一个新的关键字继承机制产生了，这就是通过引用来传递关键字。这个机制使用关键字_Ref_Extra 来定义，在使用时，可以使用_Extra 或_Ref_Extra 机制，但是注意这两种机制不能同时使用。

根据窗口大小改变字符大小

当改变显示窗口的大小并重新执行 ImageBar 命令时，图像和颜色栏能够适当改变大小，而轴和颜色栏的注释却没有。这也就意味着，当在小窗口里显示时，文本显示太大；当在大窗口中显示，文本又显得太小。

要改变字符大小，可以用关键字 CharSize，但是怎样做才能根据窗口大小来选择合适的字体大小呢？这个问题的答案是，用标准化坐标单位来表示字体的大小。然而遗憾的是，在 IDL 中这是不可能的，因为字符大小是以字符单位来表示的。即使如此，仍有办法解决。

通过使用 XYOuts 命令可以获得在标准化坐标下的字符串的宽度。先给关键字 charsize 赋予一个负值，然后用 XYOuts 计算出字符串的宽度，而非真正地向显示窗口输出字符串。例如，假设想知道文本串“A Sample String”的宽度，可以键入：

```
IDL>thisSize=-1
IDL.XYOuts,0.5, 0.5 'A   Sample String' ,/normal,$
    Width=thisWidth,CHARsize=THISSIZE
```

现在，比较一下 thisWidth 和目标宽度。例如目标宽度是 0.25，也就是说字符串长度可以达到显示窗口的 25%。如果目标宽度比这个 thisWidth 值大，可以增加字符大小然后再试一次直到字符大小合适为止。代码如下：

```
IDL>thisSize=-1
IDL.XYOuts,0.5, 0.5 'A   Sample String' ,/normal,$
    Width=thisWidth,CHARsize=THISSIZE
```

最后，thisWidth 将在目标宽度的微小误差内。如果 ThisWidth 比目标宽度长，可以用类似的算法来解决。

这种算法已经开发出来了，这便是 Str_Size 程序，这个程序可以在与本书配套的文件中找到。程序 Str_Size 参数是个字符串和标准化坐标下的目标宽度。作者发现，如果要显示的字符串为“A Sample String”以及目标宽度为 0.25，则在作者常用的显示窗口中可以很好。

```
IDL>thisSize = Str_Size ('ASample String', 0.25)
```

为了给 ImageBar 程序增加自动调整字符大小的功能，在程序代码的 TVImage 命令前增加下行：

```
thisSize = Str_Size ('ASample String', 0.25)
```

接着，给 Plot 和 Colorbar 命令定义关键字 CharSize。注意，Colorbar 上的字符大小是图像上字符大小的 75%。代码如下所示：

```
Plot,X,Y,   /NoData,XStyle=1, Ystyle=1,$
    Psition=imagePos, /NoErase, Color=drawColor,$
    _Extra=extra, CharSize=thisSize
BarPos=[ imagepos[0], (imagepos[3]+0.15), $
    Imagepos[2], (imagepos[3]+0.15)+0.15 ]
ColorBar, Position=BarPos,Bcolors=ncolors, Bottom=bottom,$
    Color=drawColor, _Extra=erxtra, Charsize=thisSize*0.75
```

保存并重新编译程序。试着以各种大小的显示窗口来运行，看运行结果有什么变化。

程序 ImageBar 的最终代码

最后的 ImageBar 程序代码如下所示。（在与本书配套的文件中可以找到其源代码，文件名成为 ImageBar.3.pro）

```
PRO   ImageBar ,image, EraseFirst=erasefirst , Ncolors=ncolors, Botttom =bottom，$
    Keep_Aspect_Ratio=keepaspect，_Extra=extra
IF N_Params ( ) EQ 0 THEN BEGIN
    IF (! D. Flags AND 256 )NE 0 THEB BEGIN
        Image = Get Image ( Canceled, 'm 51.dat'. XSize=340, Ysize=440)
        IF   Canceld   THEN RETURN
    ENDIF ELSE BEGIN
        Message, 'Please supply an image argument,',/Continue
        RETURN
    ENDELSE
ENDIF
S =Size (image)
IF s [0] NE 2 THEN BEGIN
    Message, 'Image argment must be 2D.' , /Continue
    RETURN
ENDIF
```

```
X =FIndGen (s [1] )
Y= FIndGen (s [2] )
IF N Elements (Ncolors)EQ 0 THEN Ncolors = ! d. .Table_Size
IF N_Elements (bottom) EQ 0 THEN bottom=0
drawColor = ncolors –1 + bottom
ImagePos = [0.15,   0.15.   0.9,   0.75 ]
IF Keyword_Set (eradefurst) THEN Erase
ThisSize= Str_SIZE (''A Sample String', 0.25)
TVImage,BytScl (image,Top=ncolors-1)+bottom, Position=ImagePos,$
      Keep_Aspect_Ratio=keywoed_Set(keepaspect)
Plot,X,Y, /NoData,XStyle=1, Ystyle=1, Psition=imagePos, /NoErase, $
      Color=drawColor, _Extra=extra, CharSize=thisSize
BarPos = [imagepos[0],   (imagepos[3]+0.15), imagepos[2], $
      (immagepos [3] +0.15)+0.05]
Colorbar, Position =barPos Ncoloes=ncolors, Botttom =bottom, Color=drawColor ,$
      CharSize=thisSize*0.75
END
```

实用程序的特点

下面是本程序的特点，这使得程序非常实用。

首先，自包含性。给出一个二维图像，程序可以以正确方式执行。这个图像不必有特别的大小。

其次，设备无关性。通过避免当前窗口或正使用的命令（除了在支持他们的设置上），程序可以在不同的图表输出设置上运行，包括 PostScript 设置和 Z-图表缓冲区。

然后，窗口无关性。通过运用正规协调的命令可以确定窗口中图表的位置，程序可在窗口中以任何大小执行，包括 PostScript 窗口。

第四，颜色敏感功能。程序使用不同的颜色，这使得此程序和其他可以共享颜色表的程序运作变得更有利。

最后，程序的可扩展性。建立在程序里的关键词继承特性，可以以一种自然方式设置 Plot 和 Colorbar 命令参数。

在图形用户界面中包装 ImageBar

一般地说，编写程序 ImageBar 方法的意外收获就是，它能够满足程序 Xwindow 调用其他程序的标准。程序 Xwindow 可以在与本书配套使用的文件中找到。Xwindow 的三个标准是；（1）程序本身并不打开自己的显示窗口；（2）程序至多有三个参数；（3）程序通过定义关键字_Extra 来进行关键字继承。

Xwindow 是一个具有可改变大小的图形窗口的窗体程序。在以后的章节里可以学到更多的关于窗体程序如何工作的信息，但到现在为止，只可以用它显示 ImageBar 程序。除了具有可改变大小的图形窗口的优点外，Xwindow 也可以将窗口中的内容（例如，ImageBar 程序）输出为 JEG, GIF, TIFFF 和 PostScript 文件，并且能够交互式地改变颜色。

（Xwindow 程序还要求其他三个程序，它们都在与本书配套的文件中。这些程序是 Pswindow.pro、Xcolors.pro 和 Ps_form.pro。在键入以下命令之前，确保这些程序下载到本地机上。

```
IDL>Xwindow,'ImageBar',image, /EraseFirst,m/Output, /Xclolors
```

对 Xwindow 的属性来做试验。试着将窗口图形输出为 GIF 或 TIF 文件，然后用浏览器来浏览输出文件。载入不同的颜色表并察看输出结果。

关闭 Xwindw 程序。要查看程序 ImageBar 的优越性，键入如下所示：

```
IDL>LoadCT, 3, Ncolors=75
IDL>LoadCT,4, Ncolors=75,Bottom=75
IDL>Xwindow,'ImageBar',image, /EraseFirst,m/Output, /Xclolors=[75,0],Ncolors=75
IDL>Xwindow,'ImageBar',image, /EraseFirst,m/Output,/Xclolors=[75,75], $
    Ncolors=75, Bottom=75
```

注意，可以独立地改变两个窗口的颜色。

在下面的章节，将要引入一些概念和编写技巧来创建功能较为强大的程序。

第十章　编写简单的组件程序

本章概述

本章目的是演示如何通过使用组件开发工具包开发一个简单的、具有图形用户界面的 IDL 程序。“组件”一词是指用来表示用户与程序输入输出交互作用的图形元素。按钮、滑动条和文本框都是组件。组件工具箱是一套 IDL API 命令的集合，它主要用于开发图形界面程序。在本章中，将学到以下方面内容：

1. 组件定义模块和组件事件处理模块的区别；
2. 如何编写组件定义模块；
3. 如何编写组件事件处理模块；
4. 如何解释和理解组件的事件结构；
5. 如何创建不同类型的组件，比如顶层 base 组件，绘图组件和按钮组件；
6. 组件程序模块之间在不使用公共块的情况下如何传递信息；
7. 如何编写一个具有可改变大小的图形窗口的组件程序；
8. 如何向组件程序中添加简单控制按钮；

组件程序的结构

许多 IDL 程序（比如前一章的 ImageBar 程序）只是由单一的程序模块组成，而组件程序至少包含两个程序模块。组件定义模块是一个必不可少的程序模块，它可以是过程，也可以是函数。组件本身的创建或定义就是在组件定义模块中进行的。并且，还至少有一个事件处理模块（可以是过程也可以是函数），它对组件事件、触发器或用户的交互作用做出反应，并加以处理。（请看下图 81）

甚至简单的组件程序也有几个事件处理模块，每一个模块都对不同的事件或触发器做出反应。这就要求在事件处理模块和组件定义模块之间对程序的某些信息进行共享。组件编程的主要技巧在于如何在构成组件程序或应用程序的各种程序模块之间传递程序信息。

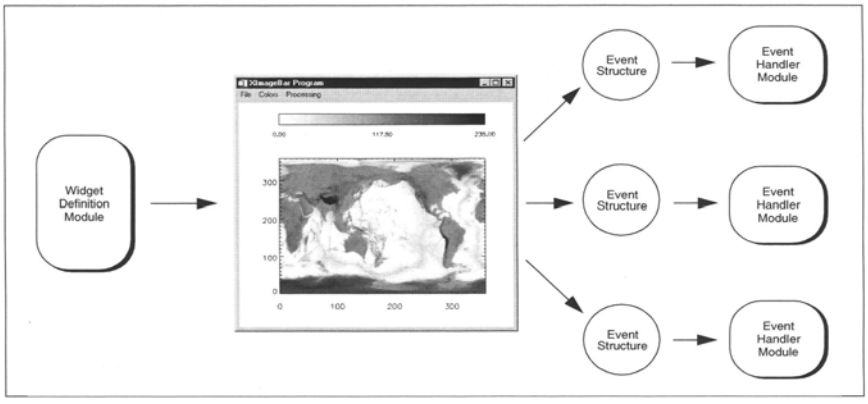


图 81　组件程序中的信息流从定义模块，通过事件结构传递到事件处理模块。一般而言，定义模块只执行一次，而事件处理模块却是重复执行。

组件程序也被称为事件驱动程序，它意味着程序事先并不知道程序执行的顺序。当用户对程序的组件元素进行交互作用时就会产生事件，然后事件处理模块根据事件产生的顺序

分别对每个事件进行处理。

一般而言，组件程序的界面或可视部分在定义模块中产生，而组件程序的行为是在事件处理模块中产生。事实上，大多数组件程序的定义模块代码只被执行一次，而事件处理模块被多次，反复执行。每次事件产生时，相应的事件处理模块都会被执行一次。

组件程序如何对事件作出反应

除了创建组件界面外，定义模块的另外任务是建立被称为“事件循环”的组件，这个名字并不确切，因为根本就没有真正的循环。当然，也可以将组件程序想象为一种等待状态，等待组件发生什么。而事实上也确实如此。但是是哪个组件，或什么在等待呢？

其实是窗口管理器在等待。在组件定义模块建立了事件循环的同时，它就将程序控制权转交给窗口管理器。（如下面所示，这两项工作都是由 Xmanager 来完成的）窗口管理器对属于自己管理的事件或用户的交互操作进行管理。

当事件产生后，窗口管理把关于事件的信息压缩成包，然后传递给 IDL。IDL 把事件信息解压后重新装入称为事件结构的里面。然后，IDL 调用合适的事件处理模块来对该事件进行处理。在调用事件处理模块时，IDL 把事件结构作为一个参数，也是惟一的一个参数传递给事件处理模块。（详细信息请参阅附录 A：传递给事件处理模块的事件结构。）

尽管每一事件结构的专有字段各有不同，这取决于产生事件的组件，但是每个事件结构都拥有三个专用字段：ID、Top 和 Handler。这一点是相同的。事实上，正是 IDL 结构变量中这三个字段的出现，使得此结构成为有别于其他结构的事件结构。下面将会很快学到这三个字段的重要性及用法。

编写组件定义模块

组件程序中需要的第一件就是组件定义模块，它有三个目的（1）创建和定义共同组成组件程序图形用户界面的组件；（2）建立程序的事件循环；（3）注册到负责事件处理的窗口管理器的程序。组件定义模块和其他模块一样，可以是过程也可以是函数。也可以用平常方式定义参数和关键字。

在这里编写的程序将对上章的 ImageBar 程序进行包装。并取名为 XimageBar。（程序名称前面的 X 是 IDL 表明该程序是组件程序的习惯。但有时候也并非如此。）

为保证程序的简单明了，起初就为 ImageBar 定义一个它所需要的参数。打开一个文本编辑窗口，键入如下所示：

```
Pro XimageBar,image
```

也可以像程序 ImageBar 一样对图像参数进行类似的处理，键入：

```
If N_Params() EQ 0 Then Begin
    Image=GetImage(Cancel=canceled, ‘m51.dat’, Xsize=340, Ysize=440 )
    If Canceled Then Return
EndIf
If Size(image, /N_Dimensions) NE 2 Then Begin
    Message, ‘Image argument must be 2D.’, /Continue
    Return
EndIf
```

定义和创建程序组件

简单或基本的组件通过使用组件工具箱 API 种的组件创建程序来建立。一般来讲，这些程序都是一些函数，它们的语法规则如下所示：

widgetID=Widget_NAME(parentID)

其中 NAME 是所创建组件的类型，可能的取值有：Base、Button、Draw、DropList、Label、List、Slider、Table 和 Text。变量 parentID 是组件的父亲或在组件层次上处于上一级的组件的标示号。（请参阅图 82 的组件层次的例子）

组件结构类似于家谱图。这样，就可以讨论组件它们好象有家庭关系一样。我们经常说到组件的父亲、孩子或兄弟。我们这样描述的其实就是组件之间的相互关系。定义模块的目的便是创建每个组件并通过参数 parentID 来描述它与程序中其他组件的关系。

每个组件结构图都有一个顶层的 base 组件。base 组件是其他组件的容器。顶层的 base 组件容纳了所有共同组成程序图形用户界面的其他组件。在组件程序中，只有顶层的 base 组件才不需要 parentID 参数。在组件创建函数中，所有其余的组件都要求有单一的 parentID 参数。

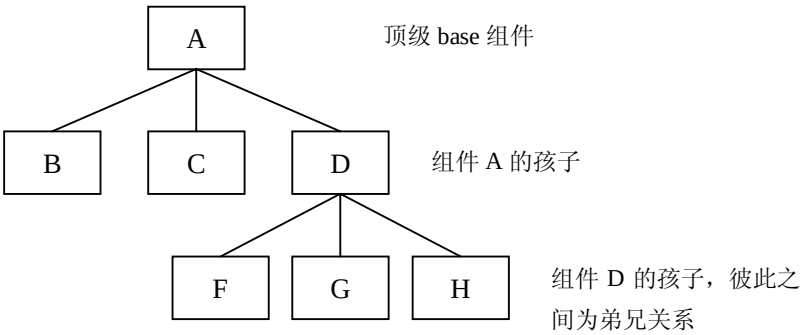


图 82　组件层次结构图。组件 A 为顶层 base 组件，同时也是其他组件的容器。组件 B, C, D 是组件 A 的孩子，它们彼此为兄弟关系。组件 F, G, H 是组件 D 的孩子，彼此是兄弟关系。

注意，作为一个函数，组件创建程序总是返回一个值。WidgetID 的值是创建指定组件的标示符，它是一个惟一的长整型数值。如果想从现实世界中找到一个比喻，可以把此整数看成社会保险或纳税号码。这个数值可以将组件区别于其他所有组件，即使它们的名字相同。如果以后要对这个组件进行处理，就必须在程序中保存这个数，同时这也是程序区分组件的方法。

创建顶层 base 组件

事实证明，在组件程序中创建最重要的组件就是顶层 base 组件。顶层 base 组件容纳了程序中所有其他组件。事实上，它就是出现在屏幕上的、包含了程序组件界面的活动窗口。尽管顶层 base 组件在创建函数中不需要 parentID 参数，但是它可能需要通过关键字来设置组件的其他属性。

这种情况下，希望顶层 base 组件有一个标题，希望子组件被创建后以列的形式存在，希望可以改变顶层 base 组件的大小，并希望该程序和在电脑中看到的用到的的其他许多程序一样可以利用按钮来控制程序操作。定义这样一个顶层 base 组件，可以键入：

```
tlb=Widget_Base(Title='XimageBar Program', Column=1,$
    Tlb_Size_Events=1,Mbar=menubarID)
```

注意，关键字 Title、Column、TLB_Size_Events 均为输入型关键字（信息传递到程序 Widget_Base），而关键字 Mbar 就是一个输出型关键字（信息从程序 Widget_Base 中传出来），

因为输入型和输出型关键字的语法规则完全相同，所以必须从上下文及相关文档中来区别它们。（关键字的详细信息请参阅 211 的“定义关键字”）

在上述的例子中，变量 menubaseID 未被定义直到 Widget_Base 命令赋值给它。变量 menubaseID 使得在顶层 base 组件的菜单栏内设置按钮成为可能，因为 menubaseID 变量可以作为菜单栏按钮的父亲标示符。

创建菜单栏按钮

接下来，定义一个下拉菜单 file，其中有一个 Quit 子菜单。其代码如下所示：

```
fileID=Widget_Button(menubarID,Value='File', Menu=1)
quitID=Widget_Button(fileID,Value='Quit')
```

注意，变量 menubaseID 作为菜单项 File 的父亲是如何使用的。File 按钮，反过来又是 Quit 按钮的父亲（通过标示符 fileID）。按钮组件通常是不允许作为其他按钮的父亲。之所以 File 可以成为其他组件的父亲，是因为关键字 Menu 在起的作用。File 按钮带了关键字 Menu 就变成了下拉式菜单，因而可以有孩子。

注意，在菜单项 File 的创建函数中，并不是一定要求有 Menu 关键字，尽管虽然设置它没有什么坏处。菜单栏的任何孩子（通过顶层 base 创建程序中的关键字 Mbar），从定义上来说，都是一个菜单及下拉式菜单。菜单栏按钮本身从来就不产生事件，它们仅仅显示子菜单，即与之相关的菜单项。

这些按钮的值就是按钮上的文本。一般而言，如果要对按钮的值进行处理，必须记得这些字符串是区分大小写的。

为程序创建图形窗口

下一步将要创建的组件是程序图形窗口。这是一个绘图组件，绘图组件同普通的 IDL 图形窗口虽然不是完全相同但却很相似。例如，虽然可以在常规图形窗口中使用 Cursor 命令，但并不想在绘图组件窗口中使用 Cursor 命令。

（组件程序中，在绘图组件中使用 Cursor 命令会导致异常奇怪的现象，一些看起来无论如何都与 Cursor 命令无关的现象。在绘图组件中使用 Cursor 命令是完全没有必要的，因为在绘图组件的事件中已经包含了要从 Cursor 命令获取的信息。）

创建一个绘图组件并使之成为顶层 base 组件的孩子，键入如下所示：

```
drawID=Widget_Draw（tlb，Xsize=400，Ysize=400）
```

注意，绘图组件的大小，和常规 IDL 图形窗口一样是以象素为单位的。

在屏幕上实现组件

当组件由组件创建函数创建时，它们只是存在于 IDL 内存中，并没有在屏幕上显示出来。它们如果要显示出来，就必须被实现。实际上，顶层 base 被实现后其他所有在层次关系上属于它的组件都将实现。

实现组件是与组件交互作用的方式之一。在 IDL 中，所有组件的交互作用都是由 Widget_Control 命令产生。这个命令要求将交互操作的组件的标示符传递给它。Widget_Control 命令有六十多个不同的关键字，可以以不同的方式与组件进行交互作用。

为实现整个组件层次，可以键入以下命令：

```
Widget_Control, tlb, /Realize
```

使绘图组件成为当前图形窗口

不同于常规 IDL 图形窗口，绘图组件创建后不会成为当前图形窗口。事实上，在把图形绘制进去之前，就必须特别指定绘图组件窗口为活动窗口或当前的图形窗口。和操作一个常规 IDL 图形窗口一样，在操作时，可以用 WSet 命令和窗口的图形索引号来实现这个。但必须记住，绘图组件的标示符（如上面的 drawID）并不是绘图组件图形索引号。

事实上，绘图组件并不会被赋予一个图形索引号，直到它被实现。并且，绘图组件的图形窗口索引号就是绘图组件的值。这和绘图组件标示符（好比是它的保险或纳税号码）差别很大。注意一定不要混淆这两个。

因为作为一个顶层 base 的子组件，绘图组件由上面的命令实现后，IDL 就将一个图形窗口索引号赋给它。获得图形窗口索引号后，将绘图组件设置为当前图形窗口，键入如下所示：

```
Widget_Control, drawID, Get_Value=wid
Wset, wid
```

注意，关键字 Get_Value 为输出型关键字，绘图组件的图形窗口索引号存放在变量 wid 中。

在绘图组件窗口上显示图形

只需要调用以前编写的程序 ImageBar，并把这里收集的参数传递给它，就可以轻易地将图形显示在绘图组件窗口中。（如果没有自己编写这个程序，可以在与本书配套的文件中找到程序 ImageBar3.pro，并在运行之前编译它）这时，该明白这样编写程序的优点了吧。因为程序可以进入到当前图形窗口而都不必考虑窗口大小。调用方式如下：

```
ImageBar, image
```

保存程序运行时所需要的信息

要程序成功运行，几乎每个组件程序都需要某些信息，这可能是组件标示符、程序数据、图形窗口索引号数和其他信息或变量。通常，这种信息在一个程序模块创建中可以获得。例如，大多数组件标示符在组件定义模块中创建，但在程序发生行为的事件处理模块中使用。

大量组件编程技巧都来自知道如何将信息从一个程序模块传递到另一个模块，这可以通过多种方法来实现。如许多程序通过使用公共块来使程序模块得到这类信息。

但是定义公共块会导致组件程序更加没有实用价值，因为在没有公共块冲突的情况下，任何时候只能有一个程序的实例在运行。设想一下，比如一个图象处理程序，它将图像数据保存在一个名为 data 的公共块中，这个程序的两个不同的实例在初始化时载入了不同的图像数据，那么第一个程序处理的数据就不再是自己的数据了。两个程序都只能处理后被载入到公共块的图象中。由于这个以及其他原因，作者不喜欢用公共块来编写组件程序。

为避免使用公共块，作者使用自称为 info 的结构来存储必要的程序信息。这是个匿名的结构变量，它包含了在组件程序中所需要的任何信息。长期经验表明，在创建结构时最好是结构字段名称和输入的变量名称相同。虽然不是要求有这个习惯，但它却可以防止混淆。作者尽力在组件程序中给变量取相同的名字，这样使得在一段程序代码中看见变量名时，可以容易地知道程序正在处理什么。

在这个程序中，作者至少需要三条信息：图象数据、绘图组件的图形窗口索引号（使得在绘图之前可以将它设置为当前窗口）和绘图组件标示符（当需要时可以将它实现）。info

结构将这样定义，将以下三行代码输入到程序文件中：

```
Info={image: image, $        ;The image data.
      Wid: wid, $             ;The window index number
      DrawID:drawID }         ;The draw widget identifier
```

有用的程序信息。必须保存在全局内存中，这样需要这个信息的组件程序模块就可以获得。变量 info 到现在为止还是局部变量，当组件定义模块代码执行完毕后，将被清除和销毁。如果不把 info 结构保存在全局内存中，当事件处理模块需要这个信息时它将不再存在。

使用组件用户值保存程序信息

事实上，组件在创建后也是被存储在全局内存中的。更重要的是，每个组件创建时都有一个内置的分类容器，称为用户值（user value），那里可以存储任何类型的变量。不管选择何种用途，都可以获得这个用户值。这种情况下，可以使用组件的用户值来储存 info 结构的信息。

但是，应该使用哪种组件的用户值呢？事实是，选择哪一个都无所谓。要达到这个目的，可以使用程序中未用过的组件用户值来实现。实际中按照惯例，一般使用顶级 base 的用户值来实现这点。这个选择的优点待会就可以看到。

将 info 结构信息拷贝并存储顶级 base 的用户值中，在程序中键入如下所示：

```
Widget_Control, Tlb, Set_Uvalue=Info
```

创建事件循环和注册程序

组件定义模块中的最后一步就是创建组件事件循环并将程序注册到窗口管理器中。如果用 C 语言编写这个程序，也许要花费半个下午和一页的代码才能完成这项任务。而在 IDL 中只需简单调用一个 XManager 命令即可。

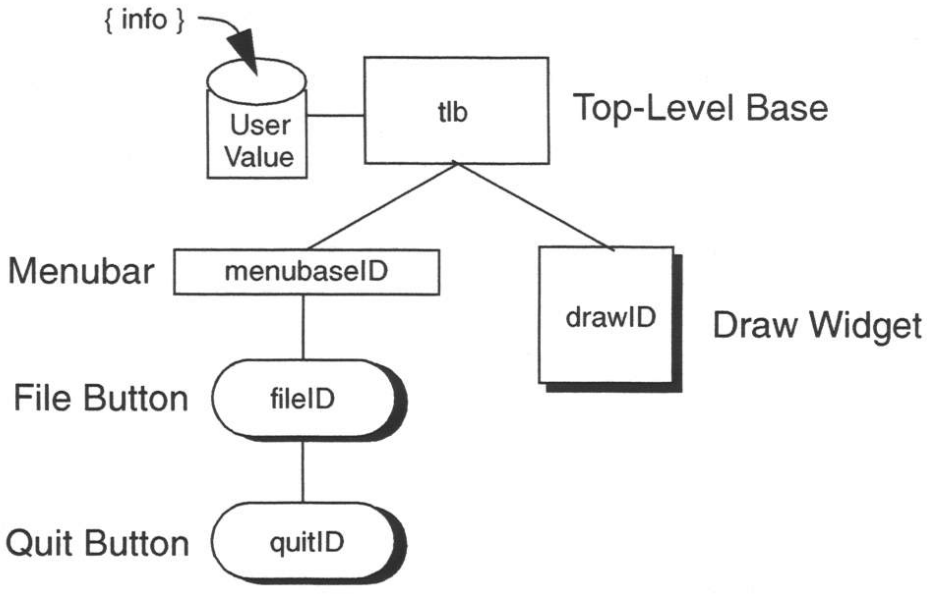


图 83　zXImageBar 组件定义模块创建的组件结构示意图。info 结构存储在顶级 base 的用户中

XManager 命令第一个参数是要注册的程序名称。一般而言，这和组件定义模块文件的名称是一致的，但并不是一定要如此。其实它可以为任何名字。在拼写名称时，最好统一是大写或小写，在以后因为某种原因需要查寻时，名字是区分大小写的。

命令的第二个参数是一级基础组件的组件识别。知道一级基础识别，XManager 就可以知道结构图中所有其他的组件。

注册程序并结束组件定义模块，键入下列两个命令：

```
Xmanager, 'Ximagebar', tlb
End
```

运行程序

到了这一步，就可以编译并运行程序 XimageBar 了，但是注意不要产生任何事件。

```
IDL>.Compile XImageBar
IDL>XImageBar
```

如果产生事件会出现什么情况呢？试着解释一下原因。

在这个程序中，只有两个组件能产生事件：一个是顶级 Base，它在用户改变窗体大小后会产生，另一个是 Quit 按钮，它在用户选择后也会产生事件。绘图组件也能产生事件，但它们必须明确地被指定，但在这里没有设置。File 按钮不能产生事件，因为它是菜单按钮（例如，它被设置了关键字 Menu）。

如果企图在 XImageBar 程序中产生一个事件，那么就会产生错误。这是因为并没有为这个程序编写事件处理模块。等一会儿即将完成这项工作。

创建无阻塞组件程序

另一件值得注意的是，这个程序是一个阻塞程序。也就是说，当运行这个程序时，就无法使用 IDL 命令行。IDL5 的最大的优点之一是，在组件程序运行的同时可以使用 IDL 命令行，但默认的设置是不具有这种能力。为使组件程序成为无阻塞程序，就必须在 XManager 命令中设置 No_Block 关键字。在程序中找到 XManager 命令的位置，并作如下修改：

```
Xmanager, 'xImageBar', /No_Block
```

这个程序的定义模块可以在与本书配套使用的文档中找到，文件名为 xImageBar1.pro。

编写事件处理模块

事件处理模块的目的就是处理程序所产生的事件。回忆一下，是窗口管理器对事件进行管理的。当一个事件发生后，窗口管理将事件的信息打包并传递给 IDL。IDL 将事件信息解包后重新打包到一个事件结构中，然后直接将事件结构作为事件处理模块的一个也是惟一的一个位置参数传递过去。

事件结构内的字段依产生事件的组件不同而不同（事件结构的详细信息请参阅附录 A：事件的结构）。但是，所有组件的事件结构有三个字段是相同的：ID、Top 和 Handler。理解这三字段的意义非常重要。

事件结构中的公共字段

事件结构中的 ID 字段总是产生事件的组件的标识符。例如，在正在编写的程序中，如果用户选择了 Quit 按钮，那么事件结构中的字段 ID 就被设置为 Quit 按钮的标识符。（如在上面的代码中，即为 quitID 的值。）

产生事件的组件往往存在于其他组件的层次结构中。Top 字段是用来标识位于该层次结

构中最顶级的组件。换句话说，在 ID 所标识的组件所在的层次结构中，Top 的值就是顶级 base 的标识符。在上面的例子中，它应该是 tlb 的标识符。

Handler 字段稍微有些复杂。每个组件程序至少有一个事件处理模块，但真正的组件程序经常有好几个事件处理模块。事实上，可以为所创建的每一个组件编写一个事件处理程序，但是程序一般是不会这样编写的。例如，如果 Quit 按钮有单独的事件处理程序，顶级 Base 的改变大小事件也有单独的事件处理程序，那么编写 XImageBar 程序也许会简单些。如果这两个单独事件对应的操作在实现过程中所采用的算法差距很大，那么就经常采用单独的事件处理程序。

程序中创建的每一个事件处理程序都必须与某一个特殊的组件联系起来。处理产生事件组件的事件处理程序，具有与之相关的组件标识符。Handler 包含的值其实就是组件标识符，一个与事件处理程序相关联的组件标识符。

下面的图例将有助于加深理解这点。如图 84 中所示，组件 A-G 已经定义，与顶级 base，组件 A，相关联着一个事件处理程序。假设组件 G 产生一个事件，IDL 看是否有一事件处理程序与组件 G 相关联。在这里，是没有的，所以事件在组件层次结构图中“上浮”一层。IDL 再看组件 F，是否有事件处理程序与之相关联。仍然没有。事件继续上浮到组件 C，…等，最后，事件找到了与组件 A 相关联的事件处理程序。

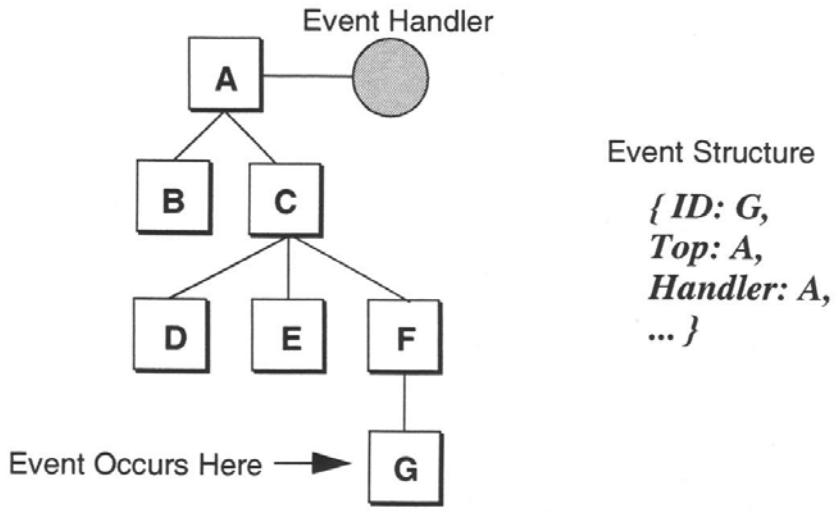


图 84　　一个简单的组件程序。组件 A-G 已经定义。与顶级 base 相关联着一个事件处理程序（填充圆所示）。如果组件 G 产生一个事件，事件就会上浮至与组件 A 相关联的事件处理程序。

然后事件结构就这样填写。ID 字段标识产生事件的组件，即组件 G。Top 字段标识组件 G 所在的层次结构中的顶级组件，即组件 A。Handler 字段标识与处理来自组件 G 事件的事件处理程序相关联的组件，仍然是组件 A。

如图 85 所示，查看一个稍微复杂的组件程序。这里有两个事件处理程序：一个与组件 A 相关联，一个与组件 F 相关联。如果组件 G 产生一个事件，那么事件就“上浮”至与组件 F 相关联的事件处理程序。这时，事件结构的 Handler 字段标识就是组件 F 而不是组件 A 了。

当组件 G 产生的事件进入到与组件 F 相关联的事件处理程序时，产生什么结果就与组件 F 的事件处理程序有关了。

事件处理函数

事件处理模块可以是过程，也可以是函数。如果与组件 F 相关联的事件处理模块是个过程，那么事件会被那个事件处理程序“吞掉”。也就是说，事件处理后就被终止。然后程

序就等待下一个事件的发生。但是假设，事件处理程序是个函数，而不是过程。回想一下，函数，是要返回值的。如果事件处理函数的返回值是个结构，且这个结构中有三个已经定义的长整型字段 ID、Top 和 Handler，那么 IDL 将把返回值作为一个事件结构上浮至层次结构图中的下一个事件处理程序。如果返回值不符合事件结构的标准，那么这个事件也称为被事件处理函数吞掉了。

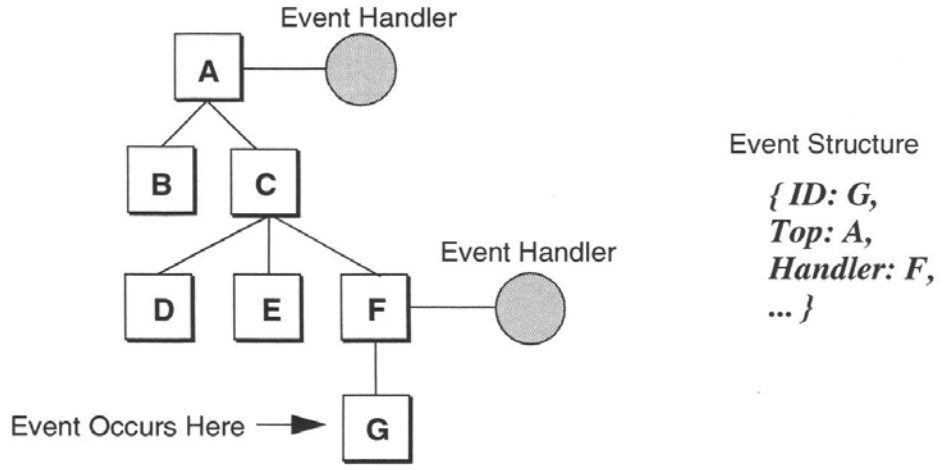


图 85　一个带有两个事件处理模块的、稍微复杂些的组件程序，产生于组件 G 的事件由与组件 F 相关联的事件处理程序来处理

因而有选择性地返回事件或处理事件及创建自己的事件是有可能的。例如，假设想将一个绘图组件用起来像一个按钮，就应该将绘图组件事件打包成一个按钮事件，然后将事件结构传递给层次结构中的下一个事件处理程序。

伪按钮事件的事件处理程序的结束或许会像如下所示：

```
Event={PSEUDO_BUTTON_EVENT,ID:Event.handler,TOP:Event.Top,$
    Handler:0L,Select:1L}
Return
End
```

注意，Handler 字段由长整型零填充，一个无效的组件标识符。IDL 将获取这个返回值，检查到 ID、Top 和 Handler 字段为合适的长整型，并在 Handler 字段中填入适合的值。因而不必在最初时为查找一个合适的 Handler 而担忧。最为重要的是，为字段 ID 和 Top 找到合适的组件标识符。例子里显示的值非常具有代表性。

将事件处理程序和组件联系起来

如何才能将事件处理模块和特定的组件联系起来呢？最简单的方法就是，在组件创建时，用关键字 Event_Pro（如果事件处理模块是一个过程）或 Event_Func（如果事件处理模块是个函数）来指定适当的事件处理模块。

例如，为 quit 按钮创建一个单独的事件处理过程，可以在组件定义模块中按钮创建程序中做如下修改：

```
QuitID=Widget_Button(fileID, Value='Quit', Event_Pro='XimageBar_Quit')
```

最好为事件处理模块取个独一无二的名字。作者喜欢以组件定义模块名作为事件处理模块的开端，这样就不会混淆。注意，名字不区分大小写。

一个组件除外，Event_Pro 和 Event_Func 关键字可用于将事件处理模块和其他任何组件联系起来，这个组件就是由 Xmanager 直接管理的那个组件（如，顶级 base 的标识符就是 Xmanager 命令的参数之一）。Xmanager 直接管理的顶级 base 已经有它们的事件处理程序，

它是通过 Xmanager 命令中的 Event_Handler 关键字来设定的。与顶级 base 相联系的事件处理程序必须是一个过程而不是函数。

如果在调用 Xmanager 命令时没有使用 Event_Handler 关键字，那么与顶级 base 相关联的事件处理过程就是程序所注册的名字加上_Event。例如，程序 XImageBar 默认的事件处理过程就是 XimageBar_Event，因为注册名是'xImageBar'（Xmanager 命令后的第一个参数）。注意，事件处理程序不区分大小写。

在 XImageBar 程序中，最好给与顶级 base 相关联的事件处理程序一个比较形象的名字。如它用来改变组件程序大小的程序，可以取名为 XimageBar_Resize。修改组件定义语句，见下：

```
Xmanager, 'xImageBar', tlb, /No_Block, $
    Event_Handler='XimageBar_Resize'
```

现在，图 86 所示，给组件程序增加了一个事件处理程序。

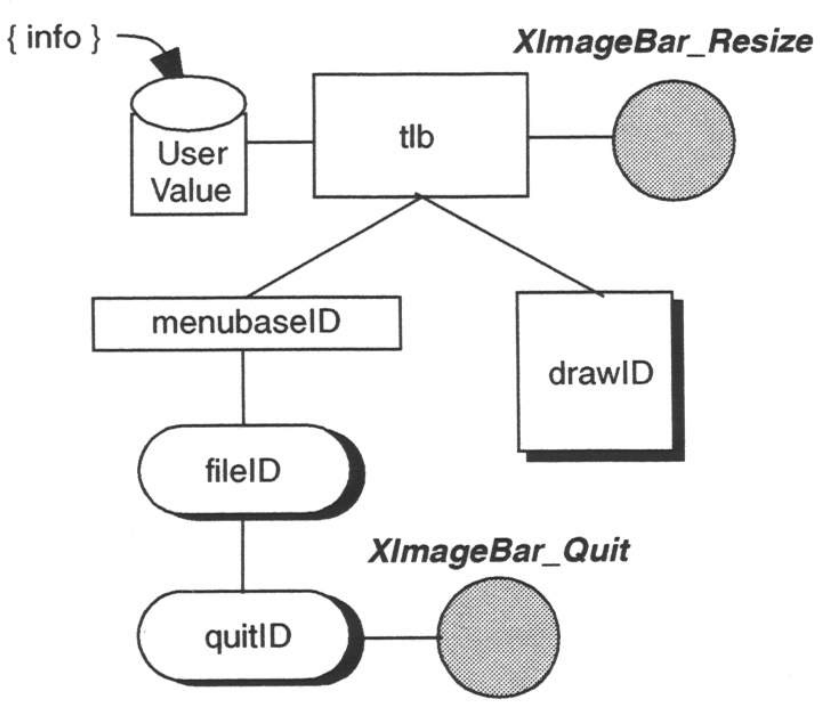


图 86　　在 XImageBar 程序示意图中，事件处理模块与特定的组件联系在一起

编写 Quit 按钮的事件处理程序

编写 Quit 按钮的事件处理程序很简单，在本程序中，它是一个过程，其惟一的功能就是销毁所有组件并从屏幕上删除。如果销毁了顶级 base，那么程序将自动地销毁该层次结构中的所有组件。

惟一的技巧是如何获得顶级 base 的标识符，tlb，一个局部变量。由于变量 tlb 是组件定义模块中的局部变量，当组件定义模块一旦执行完毕，这个变量就被销毁了。在这个模块中，可以获得等同于 tlb 的标识符，即事件结构中的 Top 字段。

所有的事件处理程序，都有一个，也只有一个位置参数。如下所示。在文件 xImageBar.pro 中的组件定义模块之前键入以下代码：

```
Pro XimageBar_Quit, Event
    Widget_Control, Event.Top, /Destroy
End
```

注意事件结构不一定非得命名为 event，它可以随意取名。最好将它取名为 event，以便

在组件程序代码中看到它时可以知道那是代表什么。

编写改变图形窗口大小的事件处理程序

与 Quit 按钮的事件处理程序相比，改变图形窗口大小的事件处理程序稍微复杂一些。想想看，如果顶级 base 改变大小后，应该作些什么。

1. 需要知道图形窗口改变后的大小，这个信息可以来自于顶级 base 的事件结构中的 X 和 Y 字段。（base 的事件结构请参阅 305 页的附录 A：组件的事件结构）
2. 需要将绘图组件（即图形窗口）拉伸到合适的尺寸。如果知道了绘图组件的标识符，用 Widget_Control 命令就能做到。在这里，绘图组件标识符就是 drawID。
3. 必须将绘图组件设置为当前图形窗口。如果已经知道绘图组件的图形窗口索引号用 WSet 命令即可实现这个。在本程序中，绘图组件的图形窗口索引号就是 wid。
4. 必须重新绘制图形。如果知道数据参数，使用 ImageBar 命令即可完成。

第一个要求所需的信息将来自事件结构本身。顶级 base 改变大小的事件件结构如下：

```
Event={ID: 0L, Top:0L, Handler:0L, X:0L, Y:0L}
```

其中，字段 X 和 Y 分别为 base 最后的大小，以像素为单位。

第 2-4 所要求的信息存于结构的信息。记住，info 结构保存于 IDL 中的全局变量中。因而必须知道，获取该变量后，将它拷贝至事件处理程序中的一个局部变量中。Info 结构保存于顶级 base 的用户值中，其中顶级 base 由事件的 Top 字段标识。

打开文件 xImageBar.pro，并在文件的首端输入以下两行代码。第一行是事件处理过程的定义行；第二行代码将顶级 base 的用户值所保存的 info 结构拷贝至一个名为 info 的局部变量中。键入如下：

```
Pro XimageBar_Resize, Event
Widget_Control, Event.Top, Get_Uvalue=info
```

下一步，通过使用关键字 Draw_XSize 和 Draw_Ysize，改变绘图组件的尺寸，以便保证它和顶级 base 大小一致。注意，事件结构中返回的顶级 base 的大小既不包括菜单栏区域，也不包括窗口标题和边框的大小。这时，不仅需要获取保存在局部变量 info 中的绘图组件的标识符，还需要事件结构中正确尺寸。键入如下所示：

```
Widget_Control, info.drawID, Draw_Xsize=Event.x,$
    Draw_Ysize=Event.Y
```

再下一步，将绘图组件设置为当前图形窗口，以便将图形显示在窗口上。

```
Wset, info.wid
```

这个重要的步骤再怎么强调都不为过。在组件程序中，必须确保知道当前所在的图形窗口，否则，将会导致在其他组件程序窗口绘制图形的操作结束。在绘图组件窗口中绘制任何图形之前一定得调用 WSet 命令。

最后，可以重新显示图形了。在本程序中，设置 ImageBar 命令中的 EraseFirst 关键字非常重要，否则在改变大小后，将会看到窗口中的残留显示信息。键入如下：

```
ImageBarm, info.image, /EraseFirst
End
```

改变大小的事件处理程序的最终代码如下所示：

```
Pro XimageBar_Resize, Event
Widget_Control, Event.Top, Get_Uvalue=info
Widget_Control, info.drawID, Draw_Xsize=Event.X,$
    Draw_Ysize=Event.Y
Wset, info.wid
ImageBar, info.image, /EraseFirst
```

End

现在可以编译和运行这个程序了。检测图形窗口是否正确改变大小以及 Quit 按钮是否工作正常。（这些代码可以在与本书配套使用的文档中可以找到，文件名为 xImageBar2.pro）

```
IDL>.Compile XImageBar
IDL>XImageBar
```

如果是在 Windows 95 或 Windows NT 操作系统上运行 IDL，可以看到窗口改变大小时会缓慢闪烁，这是因为在显示时，设置了“拖动窗口时显示其内容”。打开控制面板并选择显示项。在控制面板的“显示设置”区中取消对“拖动窗口时显示其内容”的选择。

进行小量地修改

XImageBar 程序编写得还不错，但做一些部分修改效果会更好。

添加颜色敏感

首先，要使 ImageBar 程序具有颜色敏感功能是比较麻烦的（请参阅 238 页的“向 ImageBar 程序增加颜色敏感功能”）。颜色敏感在这儿不明显。一般来说，程序具有颜色敏感功能是通过向程序添加 NColors 和 Bottom 关键字，并适当地处理这些值来实现的。

但是在 ImageBar 程序中，这些关键字也被定义，正如关键字 EraseFirst 和 Keep_Aspect_Ratio 一样。所以确保这些关键字已经传递给 ImageBar 程序方法之一就是使用关键字继承。（关键字继承的详细信息请参阅“给 XimageBar 中的命令传递关键字”）在这里，只要简单地将_Extra 关键字添加到 XImageBar 过程定义行即可。

在文件 xImageBar.pro 中的组件定义模块中找到下列行：

```
Pro XimageBar, image
```

并修改如下所示：

```
Pro XimageBar, image, _Extra=extra
```

收集和存储在匿名结构 extra 中的任何关键字，都将传递到 ImageBar 程序中。在组件定义模块中查找到下列几行：

```
ImageBar, image
```

并作如下修改：

```
ImageBar, image, _Extra=extra
```

ImageBar 程序现在具有颜色敏感功能了，但是改变大小事件的处理程序中的 ImageBar 命令又如何？颜色敏感功能不能持续很久，或效果不是很好。但请记住，如果想让在事件处理程序中的 ImageBar 命令也具有颜色敏感的功能，就必须有存储在 info 结构中的 extra 信息，以便可以在事件处理模块中获取它。

和以前做的一样，可以在 info 结构中创建一个 Extra 字段来保存变量 extra 的值，但是有一种情况更复杂：如果调用 XimageBar 时没有使用关键字，那么在组件定义模块中 extra 变量就没有定义。在 IDL 的表达式中，包括结构定义语句，如程序所创建的 info 结构，使用未定义的变量会产生错误。（然而在 IDL 中，通过关键字_Extra 传递未定义的变量不会导致错误。）

要解决这个问题，有多种方法可以采用。比如，如果 extra 变量没被定义，可以创建一个虚拟的 extra 结构，假设有一个整型的字段 Junk，如下：

```
If N_Elements(extra) NE 0 Then extra={Junk:0}
```

在 IDL5 中，通过关键字_Extra 将像这样的一个结构传递给 ImageBar 程序是不会导致错误的，因为所有未识别的关键字（Junk 当然也包含在内）都被忽略了。

将 extra 字段添加到 info 结构，在组件定义模块中找到 info 结构定义行。

```
Info={image:image,$          ;The image data
      Wid:wid,$              ;The window index number
      DrawID;drawID}         ;The draw widget identifier
```

修改如下：

```
If N_Elements(extra) Then extra={Junk:1}
Info={image:image,$          ;The image data
      extra:extra,$          ;The extra ImageBar Keyword
      Wid:wid,$              ;The window index number
      DrawID;drawID}         ;The draw widget identifier
```

继续下去，在 XimageBar_Resize 事件处理程序中找到下行：

```
ImageBar, image, /EraseFirst
```

这样修改：

```
ImageBar, image, /EraseFirst, _Extra=info.extra
```

测试程序的颜色敏感功能，保存文件并键入如下列命令。一个窗口应显示蓝色，另一个窗口应显示红色。当窗口改变大小时颜色应当保不变。

```
IDL>.Compile XimageBar
IDL>LoadCT,1, N_Colors=75
IDL>LoadCT,3,N_Colors=75, Buttom=75
IDL>XimageBar, N_Colors=75
IDL>XimageBar, N_Colors=75, Bottom=75
```

如果发现当改变一个窗口的颜色，另一窗口的颜色也改变。仔细核查代码确定程序无错误。再次运行前应改正错误并重新编译程序。

采用更高效的内存管理

关于组件程序的内存管理，需要再补充一点，尤其是在要为 info 结构分配内存时。前面就已经指出，组件定义模块中的 info 结构是个局部变量。然后将局部变量里的信息从这个局部变量拷贝到顶级 base 的用户值或全局内存中，可以用以下命令：

```
Widget_Control, tlb, Set_Uvalue=info
```

这里的关键字眼是拷贝，接着事件处理程序 XimageBar_Resize 从顶级 base 的用户值中取出信息并拷贝放置于事件处理模块中的另一局部变量中（如，名字也为 info）：

```
widget_Control, event.Top, Get_Uvalue=info
```

事实上，程序就拥有同一信息的三份拷贝。根据图 87 图解所示。如果内存管理对程序很重要，则这样做并不好（内存对程序而言，应当都是非常重要）。但应该怎么做呢？

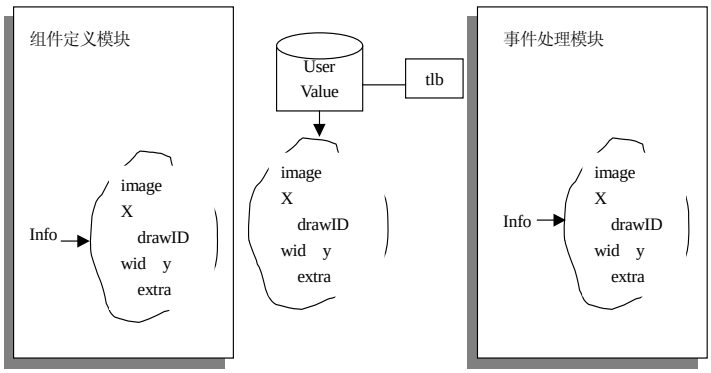


图 87　如果不细心，程序就会有同一信息的三份不同拷贝。这并不是希望的结果。为避免此事发生，将信息存入或取出顶级 base 的用户值时，可使用 No_Copy 关键字。

IDL 有一个的奇怪概念，那就是不将信息拷贝到变量中。更正确地讲，是变量名（在 C 语言中，这是一个相当复杂的指针）重新指向另外一个已经存在于内存中的信息。不将信息拷贝到变量中，其功能可以通过程序中的关键字 No_Copy 来实现。

用此种方法传递信息的有趣之处在于以前指向内存区的变量不再指向任何区。这是另一种变量未被定义的说法。这一般不是问题，因为转移总发生在 XManager 前，这是被执行的组件定义模块的最后一行。

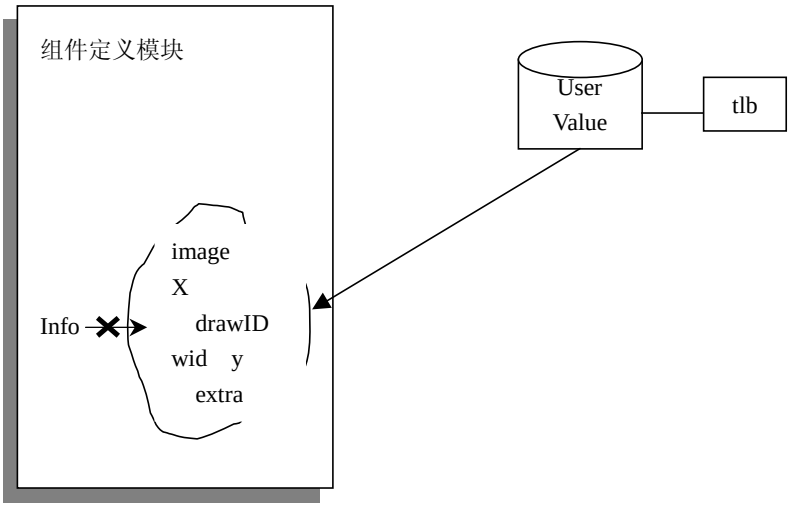


图 88　将来自组件定义模块的结构转移到带有 No_Copy 关键字的用户值可以使用户值指定内存区，内存区已以信息分配，但局部变量 info 未被转移过程定义

为使用 XImageBar 程序中的不拷贝方式，在组件定义模块中找到下一行：

Widget_Control, tlb, Set_Uvalue=Info

并作修改如下：

Widget_Control, tlb, Set_Uvalue=Info**, /No_Copy**

现在可以从下图 88 的图解中了解这个操作的影响。

相似的，如果不想将来自用户值的信息拷贝到局部变量中去。在 XimageBar_Resize 事件处理程序中找到第二行：

Widget_Control, event.top, Get_uValue=info

这样修改：

Widget_Control, event.top, Get_uValue=info**, /No_Copy**

再一次可以从下面图 89 发现此操作的影响，此操作的影响是未定义用户。不过这也不是问题，因为只需关心局部 info 结构的信息。

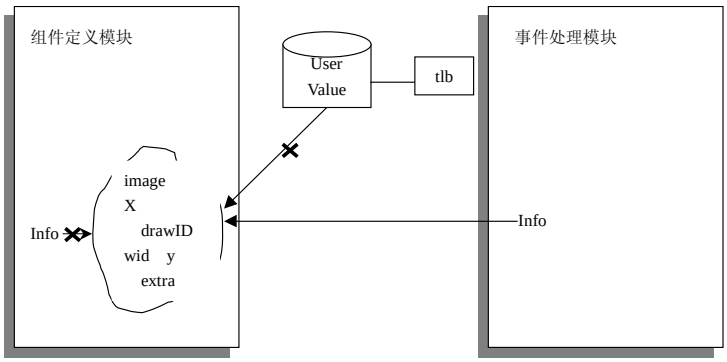


图 89　将来自用户值的信息转移到带有 No_Copy 关键字的事件句柄中的局部结构可以使变量指定原内存区，但不定义用户值

但是，如果只做这些的话，将遇到不少麻烦。注意，事件处理程序是一次性的事件处理程序。每次执行代码总有合适的事件被处理。当事件处理完后,IDL 退出事件处理模块，清除任何局部变量和指定的内存。这将包括局部 info 结构和内存定义中的信息。

一般来说，这个事件处理程序只能运行一次，处理一个事件。在接下来的事件发生时，程序将寻找顶级 base 的用户值中的程序信息，但在那里找不到（这时用户值为没有定义）。这时程序将提示错误信息“info 必须是一个结构”。

为避免这个问题，在退出事件处理模块之前，将 info 结构带回并输入带有 No_Copy 关键字的一级基础用户值。

就在 XImageBar 事件处理中的 END 陈述之前添加以下行：

```
Widget_Control, event.top, Set_uValue=info, /No_Copy
```

重新编写程序、观察是否如期望的那样仍然工作。

```
IDL>.Compile XImageBar
IDL>LoadCT,4
IDL>XImageBar
```

最终的程序代码可以在与本书配套使用的文档中找到，文件名为 xImageBar.3.pro 文件。

使用指针储存程序信息

避免存储顶级 base 用户值中 info 结构导致的内存管理问题的另一种方法是将 info 结构以指针方式存储。然后，存储在顶级 base 用户值中的仅仅是轻量级的指针。因为由定义可知，指针，也即是堆栈变量，将数据存储在全局范围内，因而不必担心使用关键字 No_Copy。

将 info 结构存储在指针内，然后将指针保存在顶级 base 的用户值中。实现代码如下：

```
ptr=Ptr_New(info, /No_Copy)
Widget_Control, tlb, Set_Uvalue=ptr
```

在事件处理模块中，要使用该结构，只需简单地获取指针，然后和通常一样使用。例如，获取指针，然后在事件处理程序 XimageBar_Resize 中调用程序 ImageBar，可用如下代码实现：

```
Widget_Control, event.top, Get_Uvalue=ptr
ImageBar, (*ptr).Image, /EraseFirst, _Extra=(*ptr).extra
```

使用指针，不必担心在事件处理程序之前将指针返回顶级 base 的用户值。但是，在组件程序销毁时应当考虑将指针释放，以免内存泄漏。此时用 Cleanup 过程最简单，此过程在279 页的“使用 Cleanup 过程防止内存泄漏”中会讨论到。

第十一章　组件编程技巧

本章概述

这一章将介绍一些重要的、经常使用的组件编程技巧。和前一章介绍的技巧一起使用，应该可以编写出功能强大的组件程序。在本章中，将学习以下内容：

1. 如何扩充组件程序的功能；
2. 如何保护具有公共块的组件程序；
3. 在组件程序中如何使用指针；
4. 组件程序之间或者是组件程序模块之间的通信新方法；
5. 如何编写适合在 8 位和 24 位的显示器上运行的程序；
6. 如何保护程序的颜色；
7. 如何将图形输出为适合广域网传输的文件和 PostScript 文件；

事实上，一个不能容易扩充或添加的组件程序是一个糟糕的程序。任何一个组件程序，首先必须保证能被简单扩充。实现扩充最简单的方法是让组件程序能够调用其他组件，或被其他组件调用。在很大程度上，这意味着用模块化或面向对象的方式来编写程序，可以提高程序的聚合度。

回想一下上一章中所写的 XimageBar 程序，作为示例，它的功能非常有限。实际上，它仅仅实现了适于某一种特殊显示设备的、可改变大小的图形窗口，对其稍加扩充便是一个很棒的程序。本章将引导读者对其进行扩充（如果需要一个起点，可以利用与本书配套使用的文档中找到 xImageBar.3.pro）。

改变颜色表

IDL 中，改变颜色表常用的工具是组件程序 XloadCT，这是一个很不错的颜色表加载程序，并可以在 IDL 命令行中调用。而它的局限性在于它是个组件程序在组件，其主要不足是它将颜色向量存储在公共块中。这意味着，在任何时刻只能有一个 XloadCT 组件可以使用。例如，输入如下三行命令：

```
IDL>XloadCT
IDL>XLoadCT
IDL>XloadCT
```

无论输入多少次，仅有一个 XloadCT 出现在屏幕上。这便是它与其他组件的不同之处，如 XimageBar。试着输入以下命令：

```
IDL>img1=LoadData（7）
IDL>img2=LoadData（11）
IDL>XimageBar, img1
IDL>XimageBar, img2
```

这时可以看到两个 XimageBar 程序同时出现在屏幕上。（当他们显示时，必须将其移开，因为程序每次调用时都出现在同一位置）其实，可以获得任意多个的组件程序。这两个程序间有那些区别呢？

保护公共块

区别在于，XloadCT 采用在任意时刻确保只有一个程序运行的方法来保护公共块。这是任何一个使用公共块的组件程序所必须的要求，同时也是防止其他组件使用该公共块的最好方法。它的缺点就是使组件程序的适用性降低。

XloadCT 在它的组件定义模块中使用了 Xregistered 命令来实现保护公共块的目的。Xregistered 命令将程序的名字作为参数，如果那个名字的程序已经注册到 Xmanager，Xregistered 命令返回 1，否则返回 0。

如果检查一下 XloadCT 的源代码（它在 IDL 安装目录的 lib 子目录下），可以在组件定义模块的顶部附近，发现这样一行，任何组件被创建之前：

IF Xregistered（‘xloadct’）NE 0 THEN RETURN

换句话说，如果名为‘xloadct’的程序已经注册，XloadCT 程序将不创建任何组件而直接返回。程序名就是与 Xmanager 一起调用时的注册名，它是区分大小写的。（这便是为什么当在用 Xmanager 来注册程序时全用大写或全用小写来拼写程序名是个好主意的原因。）

一个可选择颜色表的工具

在屏幕上能够显示一个程序的多个实例，是一大优势。例如，XimageBar 是一个颜色敏感的应用程序，它可以很好地在屏幕上显示两个不同的实例，不同程序实例使用不同颜色表。

IDL>LoadCT， 4，Ncolors=75

IDL>XImageBar，img1， Ncolors=75

IDL>LoadCT，3，Ncolors=75，Bottom=75

IDL>XimageBar，img2，Ncolors=75，Bottom=75

如果要在这些窗口中改变颜色将会怎样？可以用 XloadCT 改变一个或另一个窗口的颜色，而不是同时改变这两个窗口的颜色。

IDL> LoadCT, Ncolors=75

或者

IDL> XloadCT, Ncolors=75, Bottom=75

实际上，如果这里的每个图形窗口能够只是改变自己的颜色，并且是通过自己的颜色表工具来实现，那将会更好。其实就有这样一个名叫 Xcolors 的工具，在与本书配套使用的文档中可以找到它。

Xcolors 用它的标题来作为它的注册名字，这样，只要每个程序实例有不同的标题，就可拥有多个 Xcolors 实例了。按如下方法试一下：

IDL> Xcolors, Ncolors=75, Title=’Window 1 Colors’

IDL> XimageBar, img1, Ncolors=75

IDL> Xcolors, Ncolors=75, Bottom=75, Title=’Window 2 Colors’ 这个信息

IDL> XimageBar, img2, Ncolors=75, Bottom=75

现在每个 XImageBar 程序都有自己的 XColors 程序，并能改变所在窗口的颜色。

但是怎样从 XImageBar 程序中调用 Xcolors 呢？这很容易实现。打开 xImageBar.pro 文件，在组件定义模块里找出下行：

quitID = Widget_Button(field, Value=’Quit’, Even_Pro=’XimageBar_Quit’)

接着，在上面一行的下面添加如下几行，用以在菜单栏里创建一个 Colors 按钮，Colors 下面再创建一个 Image Colors 按钮，并将 XimageBar_Colors 事件处理程序和 Image Colors 按钮联系起来.

```
Colors = Wudget_Button (menubaseID, Value =’Colors’)
 icolors = Widget_Button (colors, Value =’Image Colors’, $
     Event_Pro=’XimageBar_Colors’)
```

一个关键字继承的问题

在编写事件处理程序之前，还需解决一个小问题。很显然，事件处理程序XimageBar_Colors 将要调用 XColors 程序，要正确地调用它，就必须知道应该加载多少种颜色以及在哪里加载颜色表。换句话说，需要知道关键字 Ncolors 和 Bottom 的值。

在 XImageBar 程序中，如果 extra 结构变量里有 Ncolors 和 Bottom 信息存在的话，Ncolors 和 Bottom 是可以通过关键字继承机制获得的。（详细信息请参阅 240 页的“一个关键字继承的问题”）。那几个变量是不会产生的，除非调用 XImageBar 程序时出现不认识的关键字。然而，在 XImageBar 程序里 Ncolors 和 Bottom 要能够方便使用，并且应该将它和其他有用的程序信息一起存储到 info 结构中。

获取这些信息的方法之一就是从 extra 变量获得，如果调用 XimageBar 程序时使用了 Ncolors 和 Bottom 关键字，那么变量 extra 就会包含这方面的信息。代码如下所示。（请不要键入这些代码，因为这不是解决问题的最好方法。这里只是显示它的不足之处。）

```
IF N_Elements (extra)  EQ  0  THEN  BEGIN
    extra ={Junk : 1}
    ncolors = !D.Table_Size
    bottom = 0
ENDIF ELSE GEGIN
    fields = Tag_Names (extra)
    dummy = Where (fields EQ ‘NCOLORS’, count)
    IF count GT 0 THEN ncolors = extra.ncolors ELSE $
        ncolors =! D.Table_Size
    dummy = Where (fields EQ ‘BOTTOM’, count)
    IF count GT 0 THEN bottom = extra.bottom ELSE $
        bottom = 0
ENDELSE
```

这些代码中，使用了 Tag_Names 和 where 命令来查找 extra 变量里是否存在 Ncolors 和 Bottom 字段。它是解决这种问题的一个可行方法，但是它有最基本的缺陷。如果严格按照上面 Where 命令中的拼写方式，那么就可以找到这些字段。换句话说，当 XImageBar 程序调用如下，这种方法就能够正常运行。

```
XimageBar, Ncolors = 200, Bottom = 100
```

如果 XImageBar 程序像下面一样调用，就不能正常运行了。

```
XimageBar ,Ncol =200 ,Bot = 100
```

通过关键字继承，关键字 Ncol 和 Bot 传递给 XImageBar 程序后被接受了，但代码还是不能正常运行。原因是它们不能够以一种可靠的方式从 extra 变量里传出，除非知道它们的确切拼写方式。

关于关键字继承，有很重要的一点值得一提。要把关键字的值从一个程序模块传到另外一个程序模块，关键字继承是一个非常有效的方法，但是如果需要获得在不同程序模块里的特定关键字的值，最好是在每个单独的模块中定义它们。

例如，这里的最好的解决方法就是，在 XImageBae 组件定义模块中直接添加 NColors 和 Bottom 关键字。找出 XImageBar 程序第一行，增加这些关键字，并在没有定义的情况下，设置这些关键字的默认值。如下：

```
PRO XimageBar , image, _EXTRA = extra
```

作如下修改：

```
PRO XimageBar , image, _EXTRA = extra, Ncolors =ncolors, $
    Bottom = bottom
IF N_Elements (ncolors) EQ 0 THEN $
    NCOLORS =! D.Table_Size
IF N_Elements (bottom) EQ 0 THEN bottom = 0
```

把变量 ncolors 和 bottom 存储到 info 结构中，在组件定义模块底端找出下面几行：

```
ImageBar, image, _Extra = extra
Info ={image : image, $          ; The image data.
    extra : extra,   $           ; The exta ImageBar keywords.
    wid : wid,   $               ; The window index number.
    drawID :drawID}              ; The draw widget identifier.
```

使用这两个变量，通过如下修改代码添加到 info 结构:

```
ImageBar, image, _Extra = extra, Ncolors = ncolors, Bottom = bottom
Info = { image: image,   $                ; The image data.
    ncolors: ncolors,   $                 ; The number of colors.
    bottom: bottom,   $                   ; The starting color index.
    extra: extra,     $                   ; The extra ImageBar keywords.
    wid: wid   $                          ; The window index number.
    drawID: drawID }                      ;The draw widget identifier.
```

在 XimageBar_Resize 事件处理程序里，还需要修改调用 ImageBar 程序的那一行程序，找出此行：

```
ImageBar, info.image, /EraseFirst, _Extra = info.extra
```

修改如下：

```
ImageBar, info.image, /EraseFirst, _extra = info.extra,   $
    Ncolors = info.ncolors, Bottom = info.bottom
```

编写颜色表工具的事件处理程序

最后准备编写 XimageBar_Colors 事件处理程序，这个非常简单。从存储器里取出 info 结构，根据适当的 ncolor 和 sbottom 信息，调用 Xcolors 程序，然后将 info 结构放回。确保文件中事件处理程序放在组件定义模块之前，编写如下所示：

```
PRO XimageBar_Colors, event
    Widget_Control, event.top, Get_UValue = info, /No_Copy
    XColors, Ncolors = info.ncolors, Bottom = info.bottom, $
        Title = 'XimageBar Colors'
    Widget_Control, event.top, Set_Uvalue = info, /No_Copy
END
```

保存并编译程序。程序能运行吗？要使程序正常运行，可能需要检查并修正其中的错误。仔细检查代码以免有问题。

```
IDL> . Compile XimageBar
IDL> image = LoadData(7)
IDL> LoadCT, 4, Ncolors = 100, Bottom = 50
IDL> XimageBar, image, Ncolors = 100, Bottom = 50
```

如果同时运行两个 XimageBar 程序，会发生什么情况？

```
IDL> LoadCT, 4, Ncolors = 75
IDL> XimageBar, image, Ncolors = 75
IDL> LoadCT, 3, Ncolors =75, Bottom = 75
IDL>XimageBar, image, Ncolors = 75, Bottom = 75
```

可以同时运行一个以上的程序吗？

答案是否定的。在任何时候只能有一个 XColors 程序在屏幕上运行，因为这两个程序实例使用了相同的标题来表示 XColors 程序。现在要做的是给每个程序一个独一无二的名字，最好是在程序里使用一个独一无二的数字来表达。最好是使用窗口索引号，那是绝对独一无二的，而且它也在 info 结构中，使用非常方便。

修改程序里的 XColors 命令，如下：

```
Xcolors, Ncolors = info.ncolors, Bottom = info. Bottom, $
    Title = 'XimageBar Colors ('+ strTrim(info.wid, 2) +')'
```

如下，又会出现什么情况呢？

```
IDL> LoadCT, 4,   Ncolors = 75
IDL>XimageBar, image, Ncolors = 75
IDL>LoadCT, 3, Ncolors = 75, Bottom = 75
IDL>XimageBar, image, Ncolors = 75, Bottom = 75
```

每个窗口都有一个（也只有一个）颜色变换工具，但是屏幕上可以有许多颜色转换工具。

指定 Group Leader

这个程序几乎能正常工作了。试一下这个。使 XImageBar 和 XColors 程序同时出现在屏幕上。（运行 XImageBar，再按一下 Image Colors 按钮.）

```
IDL> LoadCT, 4
IDL> XimageBar, image
```

现在选择 Quit 按钮关闭程序，会出现什么情况？

XColors 程序仍然在屏幕上，这并非是所想要的，因为从某种意义上来说，程序 Xcolors 是“属于”程序 XImageBar 的。当然，假设一个程序是由另一个程序产生的，如果另一个程序已经消失，那么这个程序也应该随之消失。

IDL 用一个 Group Leader 的概念来处理这种问题。组件程序可以有 Group Leader，即为另外一个组件（通常是顶级 base）。当 Group Leader 死亡或销毁，所有属于它的组件也会死亡或销毁。

在本程序中，最好让 XImageBar 程序的顶级 base 成为 XColors 程序的 Group Leader。幸运的是，Xcolors 可以通过 Group_Leader 这个关键字来接受 Group Leader 所指定的组件标识符。利用这一点，使 XimageBar 成为 Xcolors 的 Group Leader，修改如下：

```
Xcolors, Ncolors = info.ncolors,   Bottom = info.bottom,   $
    Title = 'XimageBar Colors (' + StrTrim(info.wid, 2) + ') ' ,$
Group_Leader = event.top
```

知道在这个事件处理程序中，event.top 所标识的是哪一个组件吗？如果还不知道，请参阅 256 页的“公共字段的定义”。

保存和编译并运行程序。看看出现什么情况？这时，当 XImageBar 程序选择 quit 按钮时，XColors 也会消失。

给组件程序增加 Group Leader

对于组件程序，指定一个 Group Leader 是特别有用的。如果能够使任一组件程序成为其他某一组件程序的成员，那么就能够轻易地将组件程序组合在一起。设想一下，如果在其他组件程序中，XimageBar 的功能显得非常实用的。如果能为 XImageBar 指定一个 Group Leader，那么在其他程序中增加它的功能就如同定义一个按钮和添加一个调用 XImageBar 程序的事件处理程序一样简单了。当调用程序被销毁，程序 XimageBar 也被销毁。

怎样指定一个顶级 base 作为 Xcolors 的 Group Leader 呢？

非常简单。查找到 XImagebar 程序的定义行，可以找到如下几行：

```
PRO Xcolors, Ncolors = ncolors, Bottom = bottom,   $
    Group_Leader = group
```

更进一步，上面代码中变量 group 所标识的组件被设置为 Xcolors 程序的顶级 base 的 Group Leader。代码如下：

```
Xmanager,  'xcolors',  tlb,  Group_Leader = group
```

幸运的是，group 变量不一定必须得定义，因此，不必在变量 group 没有定义的情况下给它赋一个默认值。

在作者看来，每一个组件程序都应该定义一个 Group_Leader 关键字，这样就可以在其他任一程序中都可以调用它。这是一个简单但是很有用的扩充组件功能的方法。

添加这个功能到 XImageBar 程序，找出这个程序的定义行。

```
PRO XimageBar, image, _Extra, Ncolors = ncolors, $
    Bottom = bottom
```

修改后如下：

```
PRO XimageBar, image, _Extra = extra, Ncolors = ncolors,   $
    Bottom = bottom, Group_Leader = group
```

并在组件定义模块的底部找出 Xmanager 命令行：

```
Xmanager, 'xImageBar',   tlb, /No_Block,    $
    Event_handler = 'XimageBar_resize'
```

并作如下修改：

```
Xmanager, 'xImageBar',   tlb, /No_Block,    $
    Event_Handler = 'XimageBar_Resize', Group_Leader = group
```

修改后的程序可以在与本书配套使用的文档中找到，文件名为 xImageBar.4.pro。

在 24 位显示器上改变颜色表

这个程序最终能够在使用颜色索引模式的 8 位显示器上正常运行了，在这种环境中，图像像素颜色与颜色表向量中的当前值连在一起。但是，在使用 RGB 模式的在 24 位显示器上程序工作就不一样了，其中的颜色由 RGB 值直接指定，而与颜色表向量的当前值无关。在这些显示器上，如果颜色表向量发生变化或它们与颜色表向量的当前值不一致时，图形或图像必须重新显示。（详细信息请参阅 84 页的“使用索引颜色模式和 RGB 颜色模式”。）

在 24 位显示器上也能在窗口里重新显示图形，简单得如同执行 ImageBaar 命令。问题是并不知道什么时候去做它。

在颜色表发生变化后，图形应该重新显示。但是怎样知道它什么时候发生变化呢？答案是，无法知道它什么时候改变。颜色表或许已经被其他组件程序完全改变了，但是无法将颜色表已经改变的事件返回给程序。

当 XColors 程序改变颜色的时候，它应该能传递消息（或称为事件）给 XImageBar 程

序。事实上，XColors 程序已经是这么编写的了。

创建事件并将事件传递给其他程序

这里的关键是，使用关键字 NotifyID，将一个的具有两个元素的组件标示符数组（或一个二维数组，如果有不止一个组件需要通知）传递给 XColors 程序。数组的第一个元素是当颜色表发生变化时要通知的组件的标志符。它可以是任何组件程序中任何组件的组件标志符。数组的第二个元素是包含了 info 结构的组件标示符。（实践中，第二个元素常常被设置为 Event.Top，但也不一定非得如此。）

例如，假设传递到关键字的数组定义如下：

```
array = [event . id, event . top]
```

其中，event.id 标示一个特定的组件（在这里即为产生事件的组件），event.top 标示的是顶级 base，即为第一个组件所在层次结构中的顶级 base，同时也是 info 结构存储的地方。

当颜色表改变时，Xcolors 就使用这个信息建造一个伪事件，伪事件定义如下：

```
colorEvent = {XCOLORS_LOAD,   $
    ID : array[0],   $
    Top : array[1],   $
    Handler : 0L,   $
    R : !D .Table_Size,   $
    G : !D . Table_Size,   $
    B : !D .Tabe_Size}
```

其中，字段 ID 被设置为数组的第一个元素，字段 TOP 设置为数组的第二个元素，R、G 和 B 字段的值则是从当前颜色表向量中获得。

使用 Widget_Control 命令中的关键字 Send_Event，将伪事件被放置于事件队列，如下所示：

```
Widge_Control,array[0],Send_Event=colorEvent
```

这个事件结构很像由组件程序产生的其他事件结构。然后也和它事件一样按顺序处理。事件处理程序在获取一个事件后做什么完全取决于事件处理程序本身。

看一下 XimageBar 程序能做什么。在 XimageBar_Colors 事件处理程序中找出下行：

```
Xcolors,Ncolors = info.ncolors,Botton = info.bottom,$
    Titile = 'XimageBar Colors ('_ StrTrim(info.wid,2) +')', Group_Leader = event.top
```

增加 NotifyID 关键字，如下所示：

```
Xcolors, Ncolors = info.ncolors, Bottom = info.bottom,$
    Title = 'XimageBar Colors ('+StrTrim(info.wid,2)+')',$
    Group_Leader = event.top,NotifyID = [event.id, event.top]
```

在本程序中，event.id 标识的是 Image Colors 按钮。（选择此按钮，原始事件就进入到 XimageBar_Colors 事件处理程序）这样，当 Xcolors 改变颜色表时，Xcolors 伪事件就会被送到相同的程序处理程序中。因而必须修改 Ximage_Colors 事件处理程序，以便接受这个事件。

当前的 Ximage_Colors 的事件处理程序如下：

```
PRO   XimageBar_Colors, event
Widget_Control, event.top, Get_Uvalue = info, /No_Copy
Xcolors, Ncolors = info.ncolors, Bottom = info.bottom,   $
    Title = 'XimageBar Colors (' + StrTrim(info.wid, 2) +')', $
    Group_Leader = event.top, NotifyId = [event.id, event.top]
```

```
Widget_Control, event.top, Set_Uvalue = info, /No_Copy
END
```

这个事件处理程序是为了对 Image Colors 按钮作出反应而设置的，现在需要对两种事件作出反应：一个是按钮事件和一个从 XColors 送来的事件。使用命令，通过使用 Tag_Names 命令中的 Structure_Name 关键字，可以找出进入这个事件处理程序的事件类型。记住，Tag_Names 命令总是返回一个大写的字符串。如事件来自 Xcolors 程序，检查一下看显示器是否是 24 位颜色（景深大于 256）。如果是，重新显示图形。

修改事件处理程序如下：

```
PRO XimageBar_Control, event
Widget_Control,  event.top,  Get_UValue = info, /No_Copy
ThisEvent=Tag_Names(event, /Structure_Name)
CASE  thisEvent  OF
'WIDGET_BUTTON': BEGIN
Xcolors,  Ncolors = info.ncolors,  Bottom = info.bottom,  $
Title = 'XimageBar Colors (' + StrTrim(info.wid,2) +')',  $
Group_Leader = event.top,  NotifyID = [event.id, event.top]
ENDCASE
'XCOLORS_LOAD':BEGIN
Device,Get_Visual_Depth=ThisDepth
IF thisDepth GT 8 THEN BEGIN
West,info.wid
ImageBar, info.image,Ncolors=info.ncolors, $
Bottgom=info.bottom,/EraseFirst,_Extra=info.extra
ENDIF
ENDCASE
ENDCASE
Widget_Control,event.top,Set_Uvalue=info,/No_Copy
END
```

如果有一台 24 位真彩显示器，可以试着运行一下这个程序，它现在可以顺利地运行在 8 位及 24 位显示器上，在与本书配套使用的文档中，可以找到该程序的源代码，文件名为 xImageBar.5.pro。

在组件程序中使用指针

不管如何，在组件程序中已经用到了指针。使用顶级 base 的用户值是一个指针技术，尽管准确地讲，应当说是句柄，因为“指针”（用户值）没有直接指向信息（必须将用户信息拷贝到一个局部变量中）。

但是指针在组件程序里还有其他重要作用。例如，当 inf 结构中的被引用的数据类型或数据组织改变，指针就常用于 info 结构。

如果程序增加一项功能，使它能装入一个新的图象到 XiamageBar 程序中，那将会是个好例子。如在 File 按钮底下增加一个 Open 按钮，在组件定义模块中找到如下代码：

```
fileID=Widget_Button(menubaseID,Value='File',Menu=1)
quitID=Widget_Button(fileID ,Value='Quit', Event_Pro='XimageBar_Quit'
```

在这两行代码之间增加一行代码，用于创建 Open 按钮，并使得它有自己的事件处理程序，即 XimageBar_Open。当使用它时，给 Quit 按钮添加一个分隔符，用户就会知道在 File 菜单栏下这个按钮与其他按钮作用有些不同，键入：

```
fileID=Widget_Button(menubaseID,Value='File',Menu=1)
openID=Widget_Button(fileID,Value='Open…',$
    Event_Pro='XimageBar_Open')
QuitID=Widget_Button(fileID,Value='Quit'$
    Event_Pro='XimageBar_Quit',/Separator)
```

很显然，如果打开一个文件，把其他图象读入此程序，需要存储图像到局部变量中。当前图像存储在 info 结构中，为什么不放在此处？

只要新老图像的尺寸和数据类型都一致，程序就可以正常运行。但是通常情况不是这样的，在没有完全重新构建 info 结构之前，原始 info 结构中的图像数据所分配的内存是不能改变的。（info 结构重新构建后可以容纳更大的图像，因为 info 结构是个匿名结构而不是命名结构，匿名结构就可以这样操作）与创新构建 info 结构相比（那样会陷入另一个 info 结构的定义中，也增加了代码的复杂性，从而会导致操作更加困难），一个好的解决办法就是定义一个指针，用于指向图像数据。指针所指向的变量可以动态的改变它的数据类型和组织，就像其他 IDL 变量一样。

在 info 结构中，要使 image 字段成为指针，就必须使用 Ptr_New 命令。在靠近组件定义模块的底端，定义 info 结构的地方找出下面这段代码：

```
info={image:image,$;     The mage data.
```

换成指针，如下：

```
info={image:Ptr_new(image),$;   The mage data.
```

接下来，在所有程序实例中，将 info.image 转换为指针形式。

```
Info.image 必须转换为*info.image
```

搜索字符串 info.image，并用字符串*info.image 代替它，就会发现有一行出现过两次：一次在 XimageBar_Colors 事件处理程序中，一次在 XimageBar_Resize 事件处理程序中。替换它们：

```
ImageBar, *info.image,   Ncolors = info.ncolors,   $
    Bottom = info.bottom,   /EraseFirst, _Extra = info.extra
```

至此，就可以编写 XimageBar_Open 事件处理程序了。可以使用 Get_Image 命令，允许用户打开一个图像数据文件（如果想知道文件的大小，请参阅附录 B：数据文件描述）。实际上，可以使用与 XImageBar 组件定义模块顶部的代码相似的代码。在文件中组件定义模块代码之前添加这些代码。

```
PRO XimageBar_Open,   event
Image = GetImage(Cancel = canceled, parent = event.top)
IF canceled   THEN   RETURN
s= Size (image)
IF s [0]    NE   2      THEN     BEGIN
    Message,   'Image   argument   must   be   2d.',   /Continue
    Return
ENDIF
```

注意在命令 GetImage 中关键字 Parent 的用法。在模式组件程序中，关键字 Parent 是必须要的。这在 291 页“创建模式对话框”中已经讨论过。

程序进行到这里，可以假设有一个有效的图像，这个新图像必须存储在指针内，并在窗口中显示。添加下行代码，从存储器里取出 info 结构，替换 info 结构中的图像数据并显示图像，然后将 info 结构返回给存储器。

```
Widget_Control,   event.top,   Get_Uvalue = info, /No_Copy
*info.image = image
Wset,   info.wid
```

```
    ImageBar,  *info.image,  Ncolors = info.ncolors,  $
        Bottom = info.bottom,  /EraseFirst,  _Extra = info.extra
    Widget_Control,  Event.top,  Set_Uvalue = info, /NO_Copy
    END
```

编译并运行程序，看变化后能否正常运行？

```
    IDL> .compile  XimageBar
    IDL> .XimageBar,  LoadData(7)
```

还有最后一个问题要解决。当退出程序时，在程序中使用的指针必须释放。指针数据是全局范围的并且在 IDL 的当前任务中会一直存在，除非它们被显式地被释放。

使用 Cleanup 过程防止内存泄露

为防止内存泄露，必须在程序销毁之前清除那些会导致内存泄露的东西（如指针，对象，位图窗口等）。有多种方法可以用来清除内存，其中可以采用在顶级 base 的 Xmanager 命令中设置 Cleanup 关键字的方法。

给 XImageBar 程序增加一个 Cleanup 过程，在程序代码底部找到下行：

```
    Xmanager, 'ximanager', tlb, /No_Block, Group_Leader=group, $
        Event_Handler = 'XimageBar_Resize'
```

并作如下修改：

```
    Xmanager, 'xImageBar',  tlb, /No_Block,  Group_leader = group,  $
        Event_Handler = 'XimageBar_Resize', Cleanup = 'XimageBar_Cleanup'
```

当顶级 base 从屏幕上移走后，但还没有完全销毁前，将调用 Cleanup 过程。

事实上，在顶级 base 调用 Cleanup 过程时，惟一能做的就是将保存于顶级 base 的用户值中的 info 结构取出来。但这还不够，因为是 info 结构里的信息才需要清除。尤其是要将图像数据在堆栈中分配的内存释放掉。

Cleanup 过程与 IDL 自动调用的事件处理程序相似，但它不是一个事件处理程序。Cleanup 过程必须有惟一的位置参数，一个用来标识与 Cleanup 过程相联系的组件的参数。换句话说，参数用来标识顶级 base 组件。Cleanup 过程的前两行编写如下（将这个程序模块放在组件定义模块之前）：

```
    PRO  XimageBar_Cleanup,  tlb
    Widget_Control,  tlb,  Get_Uvalue = info,  /NO_Copy
```

无论顶级 base 何时被销毁，Cleanup 过程都将被调用。在顶级 base 销毁时，info 结构才可能从顶级 base 的用户值中释放出来。

当在开发自己的应用程序时，这将是一个特殊情况。程序的错误会导致应用程序在事件处理程序中崩溃。当用鼠标销毁组件时，IDL 仍然会调用该程序模块。

如果 info 结构被释放，那么这时就不能再将它放回原处，内存泄露就几乎是不可避免的了。幸运的是，在这种极少数的情况下，可以使用 Heap_GC 来清除内存垃圾。Heap_gc 找出存储在堆栈中的数据，并将那些无效引用的数据删除，XimageBar_Cleanup 程序的剩下部分代码如下所示：

```
    IF N_Elements(info) EQ 0 THEN Heap_GC ELSE $
        Ptr_Free, info.image
    END
```

编译并运行程序，看它是否能运行。

```
    IDL>.Compile XImageBar
    IDL>XimageBar, LoadData(11)
```

修改后的 XimageBar 代码能在与本书配套使用的文档中可以找到，文件名为

xImageBar.6.pro。

使用伪事件进行程序通信

另外一个有用的组件编程技巧就是，要知道什么时候以及怎样使用伪事件。伪事件是自己创建的事件，用于组件程序模块之间，尤其是事件处理程序模块之间的通讯。

从 Xcolors 命令的关键字 NotifyID 中，就能够简单地了解到这个技巧（请参阅 274 页的“将事件发送给其他组件”）。在下一章 301 页的“通报程序事件的组件”中，将会有这方面的更详细介绍。下面将会看到这种技巧是多么实用。

如果 XimageBar 程序除了能够在可改变大小的图形窗口中显示图像，还具有其他一些功能，那么它将会更加实用。如果想在程序中添加一个图像处理功能，需要创建一个新的菜单项，并在下面增加几个图像处理功能的菜单按钮。

在 XimageBar 的组件定义模块中找到下面一行：

```
drawID = Widget_Draw(tlb,Xsize=400,Ysize=400)
```

并在其前添加如下代码：

```
processID = Widget_Button(menubaseID,Value='Processing',$
    Event_Pro = 'XimageBar_Processing'/Menu)
SmoothID = Widget_Button(ProcessID,Value='Smooth')
EdgeID = Widget_Button(processID,Value = 'Edge Enhance')
EqualID = Widget_Button(processID,Value='Histogram Equal')
OriginalID = Widget_Button(processID, Value='Original')
```

注意，ProcessID 所标识的组件已经指定了程序 XimageBar_Processing 作为其事件处理程序。由于其他按钮都是这个按钮的子按钮，因而这些按钮产生的事件将会“上浮”到 XimageBar_Processing 这个事件处理程序中。

事件处理程序的头两行很标准，并将这个事件处理程序添加到组件定义模块之前。

```
PRO  XimageBar_Processing,  event
Widget_Control,  event.top,  Get_Uvalue = info,  /No_Copy
```

在刚才创建的 5 个按钮中，只有一个按钮不能返回事件，这就是与事件处理程序相关联的那个按钮。记住，菜单项按钮（带有关键字 Menu 的按钮或菜单栏）是不能产生事件的。但是，哪一个按钮能产生事件呢？

产生事件的按钮在事件处理程序里被标识 event.id。通过获取按钮的值来找到 Event.id 所标识的是那个按钮是个不错的办法。按钮的值就是在按钮创建时通过关键字 Value 所指派的字符串。因而可以通过对按钮值进行分支，然后根据值的不同采取不同的操作。在这个例子中，所采取的操作就是对影像数据进行不同的图像处理。其代码如下：

```
Widget_Control,  event.id,  Get_Value = buttonValue
CASE  buttonValue  OF
    'Smooth': thisImage = Smooth(*info.image, 7)
    'Edge  Enhance': thisImage = Sobel(*info.image)
    'Histogram Equal': thisImage = Hist_Equal(*info.image,  $
        Top = info.ncolors-1) + info.bottom
    'Original' : thisImage = *info.image
ENDCASE
```

注意，在 CASE 语句中使用获取的按钮值的技巧只适用于按钮值是字串的情况。在 IDL 中，按钮的值还可以是位图。如果是这样的话，按钮的用户值通常存储标识该按钮的字符串。

下一步，将绘图组件设置为当前图形窗口（这点特别重要），并通过调用 ImageBar 命令将处理后的图像显示在窗口中。事件处理程序的余下代码如下所示：

```
Wset,  info.wid
ImageBar,  thisImage,  Ncolors = info.ncolors,  $
    Bottom = info.bottom,  /EraseFirst,  _extra = info.extra
Widget_Control,  event.top,  Set_Uvalue = info,  /No=Copy
END
```

保存、编译并运行该程序。查看结果如何。

```
IDL>.Compile XImageBar
IDL>XImageBar, LoadData(9)
```

按钮正常运行，显示也漂亮极了。但是如果窗口中有一个处理后的图像，再改变窗口的大小结果会怎样呢？

创建一个具有“记忆功能”的程序

不管窗口里有什么内容，当窗口改变大小时，原始的图像数据都会在窗口中显示。这并不是程序想要的结果。当窗口尺寸发生变化时，不管窗口中的当前内容如何，都应该还在那里不变。但是怎样才知道在任一时间里窗口里有什么东西呢？没有洞察力，当然很难知道在任一时间里窗口中有什么了。它完全取决于不可预知的用户用该程序时所进行的操作。

不要绝望。无论显示窗口中有什么，它都是通过 XimageBar_processing 事件处理程序进入的。如果能使事件处理程序记住它最后一次操作，那么当改变窗口的尺寸时，就可以让事件处理程序再重复一次它的最后操作。

事实上，要知道最后一次操作，事件处理程序就必须记住最后一个它所处理的按钮事件对应的标识符。简单地说，就是 event.id。但是，又将这个记忆存在哪里呢？当然在 info 结构里。

修改 info 结构，增加一个 action 字段。注意，最初的操作是将原始图像显示出来。因此 action 字段的初始值就是 original 按钮的标识符。在组件定义模块中找出如下行：

```
info = {image: Ptr_New(image),  $ ;The image data.
    ncolors: ncolors,  $  ;The number of colors.
    bottom: bottom,  $  ;The starting color index
    extra: extra,  $  ;The extra ImageBar keywords
    wid: wid,  $  ;The window index number.
    drawID: drawID}  ;The draw widget identifier.
```

修改为如下所示：

```
info = {image: Ptr_New(image),  $  ;  The image data.
    ncolors: ncolors,  $  ;  The number of colors.
    bottom: bottom,  $  ;  The starting color index
    extra: extra,  $  ;The extra ImageBar keywords
    action:originalID,  $  ;The last program action.
    wid: wid,  $  ;The window index number.
    drawID: drawID}  ;The draw widget identifier.
```

接着修改事件处理程序 XimageBar_Processing，当它处理一个事件时，更新 action 字段。在事件处理模块中找出如下行：

```
Wset,  info.wid
ImageBar,  thisImage,  Ncolors = info.ncolors,  $
    Bottom = info.bottom,,  /EraseFirst,  _Extra = info.extra
    Widget_Control,  event.top,  Set_Uvalue = info,  /No_Copy
END
```

修改为：

```
Wset,  info.wid
ImageBar,  thisImage,  Ncolors = info.ncolors,  $
    Bottom = info.bottom,,  /EraseFirst,  _Extra = info.extra
Info.action = event.id
Widget_Control,  event.top,  Set_Uvalue = info,  /No_Copy
END
```

现在好了，程序能够记住最后一次操作了，但是程序是怎样重复最后一次操作的呢？

答案是创建一个伪事件结构并将它传递到事件处理程序中。在这个事件结构中，ID 字段包含了存储在事件结构 action 字段中的组件标识符。

在 XimageBar_Resize 事件处理模块中找出如下所示两行代码：

```
Wset,  info.wid
ImageBar,  *info.bottom,  Ncolors = info.ncolors,  $
    Bottom = info.bottom,  /EraseFirst,  _Extra=info.extra
```

删除这两行代码，并用下列代码代替它们：

```
psedoEvent = { WIDGET_BUTTON, ID: info.action,  $
    Top: event.top,  Handler:0L,  Select:1}
Widget_Control,  info.action,  Send_Event= psedoEvent
```

现在完整的 XimageBar_Resize 事件处理程序如下：

```
PRO  XimageBar_Resize,  event
Widget_Control,  event.top,  Get_Uvalue = info,  /No_Copy
Widget_Control,  info.drawID, Draw_Xsize = event.x,  $
    Draw_Ysize = event.y
PseudoEvent = {WIDGET_BUTTON,  ID:info.action,  $
    Top: event.top, Handler:0L, Select:1}
Widget_Control, info.action, Send_Event = pseudoEvent
Widget_Control, event.top, Set_Uvalue = info, /No_Copy
END
```

其实，只有一两处会发生改动。

当颜色变成 24 位颜色显示时，XimageBar_Colors 事件处理程序也可以调用 ImageBar 程序来重新显示图像。在 XimageBar_Colors 事件处理模块中找出如下代码：

```
Device,  Get_Visusl_Depth = thisDepth
IF thisDepth  GT  8  THEN  BEGIN
    Wset,  info.wid
    ImageBar,  thisImage,  Ncolors = info.ncolors,  $
    Bottom =info.bottom,  /EraseFirst, _Extra = info.extra
ENDIF
```

并修改如下：

```
Device,  Get_Visusl_Depth = thisDepth
IF thisDepth  GT  8  THEN  BEGIN
    PseudoEvent = {WIDGET_BUTTON,  ID:info.action,  $
        Top: event.top,  Handler:0L,  Select:1}
    Widget_Control,  info.action,  Send_Event = pseudoEvent
ENDIF
```

最后，XimageBar_Open 事件处理模块也调用 ImageBar 程序，在 XimageBar_Open 事件处理模块中找出下面几行：

```
*info.image = image
Wset, info.wid
ImageBar,  thisImage,  Ncolors = info.ncolors,  $
      Bottom = info.bottom,  /EraseFirst,  _Extra = info.extra
```

修改如下：

```
*info.image = image
PseudoEvent = {WIDGET_BUTTON,  ID:info.action,  $
    Top: event.top,  Handler:0L,  Select:1}
Widget_Control,  info.action,  Send_Event = pseudoEvent
```

保存，编译并运行程序，结果怎么样？

修改后的程序 XimageBar 在与本书配套使用的文档中可以找到，文件名为 XimageBar.7.pro。

保护组件程序的颜色

XImageBar 程序基本上能正常运行，但是也有一个明显的弱点。要看将会发生什么，请键入如下命令：

```
IDL> LoadCT, 3, NColors = 100
IDL> XimageBar, LoadData(9), NColors = 100
```

一切正常，图象数据被分成颜色表的前 100 种颜色。如果输入如下命令又会出现什么情况呢？

```
IDL > LoadCT, 0
```

程序里的颜色全变了。使颜色恢复正常的惟一方法是使用 Image Colors 按钮来恢复它们。

在 IDL5.0 中，当程序正在运行时，也可以对 IDL 命令行进行操作。这使得可以打开更多的程序，同时也增加了编程的复杂性。其中之一就是编写的程序要能够保护自己的颜色。在编写的程序可能用于自己没有去控制的环境时，这尤其正确。

要保护程序的颜色，有多种方法可以采用。他们各有优缺点。在这里，将简要地给出两个方法，对特定的应用程序应选择哪个方法，读者自己去选择。

所有的方法中，程序都必须知道调用了哪个颜色表。因为从程序开始执行，程序对颜色就是敏感的。到现在为止，程序在任一时间调用了哪个颜色表还不能记忆。不过，这个问题很容易解决，只要将颜色矢量保存在 info 结构中。

在 XimageBar 组件定义模块中找到 info 结构的定义语句，如下：

```
info = {image: Ptr_New(image),    $   ;   The image data.
    ncolors: ncolors,        $       ;   The number of colors.
    bottom: bottom,      $           ;   The starting color index
    extra: extra,    $               ;The extra ImageBar keywords
    action:originalID,  $            ;The last program action.
    wid: wid,  $                     ;The window index number.
    drawID: drawID}                  ;The draw widget identifier.
```

修改这段代码，使包含应用于程序的红，绿，蓝颜色向量。修改后的代码如下：

```
    TVSCL,r,g,b,/Get
    r=r[bottom:ncolors-1+bottom]
    g=g[bottom:ncolors-1+bottom]
    b=b[bottom:ncolors-1+bottom]
    info = {image: Ptr_New(image),    $       ;   The image data.
```

```
r : r,  $                    ;  The red color values.
g : g,  $                    ;  The green color values.
b : b,  $                    ;  The blue color values.
ncolors: ncolors,      $     ;  The number of colors.
bottom: bottom,        $     ;  The starting color index
extra: extra,          $     ;  The extra ImageBar keywords
action:originalID,     $     ;  The last program action.
wid: wid,              $     ;  The window index number.
drawID: drawID}              ;  The draw widget identifier.
```

如果在程序执行过程中颜色变了，必须更新这些颜色向量。在程序中，已经有处理颜色变化事件的程序了（通过程序 Xcolors），因此可以修改 XimageBar_Colors 事件处理程序来更新 info 结构。

在事件 XimageBar_Colors 处理程序中找出如下代码：

```
'XCOLORS_LOAD' : BEGIN
    Device,  Get_Visual_Depth = thisDepth
    IF thisDepth  GT  8  THEN  BEGIN
```

并修改如下：

```
'XCOLORS_LOAD' : BEGIN
    info.r = event.r (info.bottom : info.ncolors – 1 + info.bottom)
    info.g= event.g (info.bottom : info.ncolors – 1 + info.bottom)
    info.b = event.b (info.bottom : info.ncolors – 1 + info.bottom)
    Device,  Get_Visual_Depth = thisDepth
    IF thisDepth  GT  8  THEN  BEGIN
```

现在应该知道了如何保护程序的颜色不受其他程序的破坏。

通过组件跟踪事件来保护颜色

当光标通过组件边界时，组件就会产生跟踪事件，产生的事件结构定义如下：

```
event = { WIDGET_TRACKING, ID :0L, Top : 0L, Handler : 0L, Enter : 0 }
```

在这里，当光标进入组件边界时，Enter 字段为 1，离开时则为 0。

理论上，通过为组件设置关键字 Tracking，组件就会产生跟踪事件。但实际上，IDL 的最新版本中（在编写本书时，最新版本为 5.2），在一些操作系统里的顶级 base 不能产生跟踪事件。同时，只有在 X window 中跟踪事件才真正可靠。在其他一些开发环境中（如 Windows 98 或 Windows NT），跟踪的概念还没有定义，因而只是假的。由此造成的结果就是有些跟踪事件被忽略了，这取决于光标进入或离开组件边界时的速度了。

将程序的绘图组件的跟踪事件打开，可以看到跟踪事件运行很正常。在 XImagBar 组件定义模块中找到下行：

```
drawID = Widget_Draw(tlb, Xsize = 400, Ysize = 400)
```

修改这个语句，使跟踪事件打开，并将产生的事件让 XimageBar_Protect_Colors 事件处理程序中去处理。修改结果如下：

```
drawID = Widget_Draw(tlb, Xsize = 400, Ysize=400，$
    /Tracking, Event_Pro='XimageBar_Protect_Colors')
```

XimageBar_Protect_Colors 程序很简单，当事件检测到光标进入绘图组件边界时，就重新装入颜色表向量。代码如下所示：

```
PRO  XimageBar_Protect_Colors,  event
IF  event.enter  NE  1  THEN  RETURN
```

```
    Widget_Control,  event.top,  Get_Uvalue = info,  /No_Copy
    TVLCT,  info.r,  info.g,  info.b  info.bottom
    Widget_Control,  event.top,  Set_Uvalue = info,  /No_Copy
    END
```

保存并编译程序。键入这些命令，因为这些命令使用了重叠的颜色表。当将光标从一个窗口移到另一个窗口时，两个窗口里的颜色都应该改变。然而，当光标在一个窗口里，此窗口的颜色就应该正确。它是这样工作的吗？能说它已经出错了吗？两个程序能够很好地共存吗？

```
IDL> LoadCT,  3, NColors = 100
IDL> XImageBar,  LoadData(9),  NColors = 100
IDL> LoadCT,  5,  NColors = 100,  Bottom = 50
IDL> XimageBar,  LoadData(7),  NColors = 100,  Bottom = 50
```

如果在 Windows 操作系统下，如果飞快地将光标移入到绘图组件的边界，看跟踪事件是否被记录下来了。

通过绘图组件事件来保护颜色

另外一种保护颜色的方法并不依赖跟踪事件，而是寻找其他类型的事件。在当前程序中绘图组件并不产生事件，因此比较适合实现这个功能。在绘图组件事件中，可以找到鼠标键按下的事件（绘图组件的事件结构请参阅 305 页的“基本组件的事件结构”）。鼠标键按下的事件可以通过事件结构的 Type 字段检测到，Type 为 0 表示按钮按下。

在 XImageBar 组件定模块中找到下行：

```
drawId = Widget_Draw(tlb,  Xsize = 400,  Ysize = 400,  $
    /Tracking,  Event_Pro = 'XimageBar_Protect_Colors')
```

作如下修改，并设置绘图组件的按钮事件

```
drawId = Widget_Draw(tlb,  Xsize = 400,  Ysize = 400,  $
    /button_Events,  Event_Pro = 'XimageBar_Protect_Colors')
```

在 XimageBar_Protect_Colors 事件处理程序中找到下行：

```
IF  event.enter  NE  1  THEN  RETURN
```

修改如下：

```
IF  event.type  NE  0  THEN  RETURN
```

保存并编译程序，键入这些命令。必须单击绘图组件窗口，以保护程序的颜色。这时程序是怎样工作的？这两个程序共存效果如何？与跟踪事件处理相比，哪一个更受欢迎？

```
IDL> LoadCT,  3, NColors = 100
IDL> XImageBar,  LoadData(9),  NColors = 100
IDL> LoadCT,  5,  NColors = 100,  Bottom = 50
IDL> XimageBar,  LoadData(7),  NColors = 100,  Bottom = 50
```

修改后的程序可以在文件 xImageBar.8.pro 中找到。

保存或者发布程序的图形

一个组件程序还不能说就已经完成了，除非它的图形输出能够以某种格式保存起来，然后打印或发布在广域网上。幸运的是，我们编写 ImageBar 和 XImageBar 程序的方法，可以很容易将图形输出保存为 PostScript，GIF 或 JPEG 文件。

首先，在文件菜单栏下创建一个 Save As 按钮，并在它的下级创建三个按钮，用于分别

输出 PostScript、GIF 和 JPGE 格式的文件。在 XimageBar 的组件定义模块中找到如下代码：

```
openID = Widget_Button (fileID, Value =’Open…’,  $
    Event_Pro =’XimageBar_Open’ )
QuitID = Widget_Button (fileID, Value = ‘Quit’,  $
    Event_Pro = ‘XimageBar_Quit’, /Separator)
```

并修改如下所示：

```
openID = Widget_Button (fileID, Value =’Open…’,  $
    Event_Pro =’XimageBar_Open’ )
SaveID = Widget_Button(fileID,  Value =’Save As’,  /Menu)
Psid = Widget_Button(saveID,  Value = ‘PostScript File’, $
      Event_Pro=’XimageBar_PostScript’)
JpegID = Widget_button(saveId, Value=’JPEG File’, $
      Event_Pro =’XimageBar_GIF’)
QuitID = Widget_Button (fileID, Value = ‘Quit’,  $
    Event_Pro = ‘XimageBar_Quit’, /Separator)
```

先从 XimageBar_GIF 事件处理程序开始，因为它最容易编写。这里的思路是这样的，简单地给显示窗口拍一个快照，获取当前颜色表向量，然后用 Write_GIF 命令写文件。（输出彩色的 GIF 文件的详细信息请参阅 154 页的“创建彩色 GIF 文件”）。整个 XimageBar_GIF 事件处理程序的代码如下：

```
PRO  XimageBar_GIF,  event
ThisFile = Dialog_Pickfile(/Write,  File = ‘idl.gif)
IF  thisFile  EQ ‘ ‘THEN  RETURN
Widget_Control, event.top,  Get_Uvalue = info, /no_Copy
Wset,  info.wid
Device, Get_Visual_Depth = thisDepth
IF thisDepth  EQ  8  THEN  BEGIN
    Snapshot = TVRD()
    TVLCT, r, g, b, /Get
ENDIF ELSE BEGIN
    Image24 = TVRD(true = 1)
    Snapshot = Color_Quan(image24, 1, r, g, b)
ENDELSE
Write_GIF,  thisFile, snapshot, r, g, b
Widget_Control, event.top, Set_Uvalue = info, /No_Copy
END
```

XimageBar_JPEG 事件处理程序并不是很复杂。惟一的技巧就是彩色的 JPEG 图像必须是 24 位图像（关于输出彩色 JPEG 文件，详情请参阅 155 页的”创建彩色 JPEG 文件”）。整个 XimageBar_JPEG 事件处理程序代码如下：

```
PRO  XimageBar_JPEG,  event
ThisFile = Dialog_Pickfile(/Write,  File = ‘idl.jpeg)
IF  thisFile  EQ ‘ ‘THEN  RETURN
Widget_Control, event.top,  Get_Uvalue = info, /no_Copy
Wset,  info.wid
Device, Get_Visual_Depth = thisDepth
IF thisDepth  EQ  8  THEN  BEGIN
    Snapshot = TVRD()
```

```
        TVLCT, r, g, b, /Get
        S = Size(snapshot)
        image24 = BytArr(3, s[1], s[2])
        image24[0, *, *] = r[snapshot]
        image24[1, *, *] = g[snapshot]
    ENDIF   ELSE   Image24 = TVRD(true = 1)
    Write_JPEG, thisFile, image24, True = 1,   Quality = 75
    Widget_Control, event.top, Set_Uvalue = info, /No_Copy
    END
```

创建 PostSript 文件输出可以很复杂也可以很简单。例如，如果调用 PS_From 程序，用户就能交互地配置 PostScript 设备（详细信息请参阅 200 页的“用 PS_Form 配置 PostScript 设备”）。在这种情况下，假设用户想类似在显示器输出那样输出,但是以灰度图的形式。

XimageBar_PostScript 事件处理程序的第一部分和其他的一样，键入如下命令：

```
    PRO XimageBar_PostScript, event
    Thisfile = Dialog_Pickfile(/Write, File =’idl.ps’)
    IF thisFile EQ ‘’ THEN RETURN
    Widget_Control , event .top ,Get_Uvalue = info, /No _ Copy
    Wset, info.wid
```

可以用与本书配套使用的 PSWindow 程序来创建一个同显示窗口一样比例的 PostScript 窗口（注意，PSWindow 返回的值是以英寸为单位的）。这将使显示尽可能地接近显示器效果。键入：

```
    keywords = PSWindow（）
```

下一步，保存当前图象设备设置，并将 PostScript 设备设为当前图形设备，用命令 Device 设置成想要的设置，键入：

```
    thisDevice = !D.Name
    Set_Plot, ‘PS’, /Copy
    Device, File = thisfile, _Extra=keywords,/Inches, $
    Bits_Per_Pixel=8
```

注意，Copy 关键字是用来将当前颜色表拷贝到 PostScript 设备，Bits_Per_Pixel 关键字被置成 8，这样保证输出为灰度图形。其缺省值是 4，它只可获取 16 位灰度图像。（创建 PostScript 输出的详细信息，请参阅 175 页的“图形硬拷贝输出”）

下一步，创建一个伪事件，并发送到 XimageBar_Processing 事件处理程序中，它具有显示窗口的记忆功能。

```
    PseudoEvent = {WIDGET_BUTTON, ID:info.action,$
        Top: event.top,Handler:0L, Select : 1}
```

可以回调那些不能被直接调用的事件处理程序。其实，当事件发生时它们总是被在 IDL（实际就是 XManager）中调用。迄今为止，在创建一个伪事件时一般是遵循这一规则，即通过 Widget_Control 命令的 Send_Event 关键字来调用这个事件。这个规则不错，且很可靠，应当被推广。然而与其他优秀的规则一样，总有例外的时候，下面便是一个例外。

如果想直接调用 XimageBar_Processing 事件处理程序，必须小心。使用 Widget_Control 命令中的 Send_Event 关键字将伪事件发送给其他事件处理程序的原因之一就是，这种技术消除了在顶级 base 的用户值校验 info 结构时的所有可能问题。因为事件是一个接一个的处理，如果每个事件处理程序都校验 info 结构，并在事件处理程序结束之前将它返回到用户值中，那么下一个事件处理程序也同样可以获得 info 结构。如果从一个事件处理程序中直接调用另一个事件处理程序，那么就必须对 info 结构负全部责任。实际上，在 XimageBar_Processing 事件处理程序被调用之前，必须确保 info 结构在它的存储单元中，键

入如下所示：

```
Widget_Control, event.top, Set_Uvalue=info, /No_Copy
XimageBar_Processing, pseudoEvent
```

最后，关闭 PostScript 文件，返回并将显示设备设为当前图形设备，键入：

```
Device, /Close_File
Set_Plot, thisDevice
END
```

如果现在编译运行，编译程序会提示出错，

```
%WSET: Routine is not define for current graphics device.
```

错误源于 XimageBar_Processing 事件处理程序。实际上，程序中有个 Wset 命令，当显示图形时，它就将显示窗口设置为当前图形窗口，但是现在并没有在窗口中显示图形，而是要在 PostScript 文件中显示图形。因此，必须确保只有在适当的时候才运行这个命令。这可以用系统变量!D 中的 flags 字段来分辨当前图形设备是否支持窗口。（在程序 ImageBar 中已经使用过了这个命令。）

在 XimageBar_Processing 事件处理程序中找出下行：

```
Wset，  info.wid
```

将其修改为：

```
IF (!D.Flags AND 256) NE 0 THEN WSet, info.wid
```

如果现在编译并运行程序，就可将图形输出为 PostScript 文件，但效果并不好，因为无论显示器上设置什么背景，PostScript 文件的背景都是为白色。（详细信息请参阅 185 页的“显示设备与 PostScript 设备之间的不同点”）图形输出一般是采用颜色表顶端的颜色来显示，这些颜色通常很淡，在白色背景下效果不好。

有几种方法可以解决这个问题，不过都要求增加 info 结构的字段以包含更多的信息。

解决这个问题的一个简单方法就是，返回到 imageBar 程序中，如果当前图形设备为 PostScript 设备，那么只要简单地将绘图颜色加深一点即可。

打开 imageBar 程序并找到下行：

```
drawColor = ncolors – 1 + bottom
```

并修改为：

```
IF !D.NAME EQ ‘PS’ THEN drawColor = !P.Color
```

保存并编译 imageBar 和 XimageBar 程序，然后运行 XimageBar 程序。

在与本书配套使用的文档中，可以找到 imageBar 和 XimageBar 程序的最终代码，文件名分别为 ImageBar.4.pro 和 xImageBar.9.pro。

第十二章　对话框程序

本章概述

本章主要讲解两种编写对话框程序的方法。对话框是用于接收用户信息，并把信息传递到另一个程序模块，或程序中。其中，将学到以下几个方面的内容：

1. 如何编写模式对话框；
2. 如何编写非模式对话框；
3. 如何在组件程序中用指针存储信息；
4. 如何在组件程序中使用伪消息；
5. 如何在独立的组件程序之间传递信息；

创建模式对话框

在上一章的 XImageBar 程序中，已经调用过一个模式对话框程序，即 GetImage，它允许用户选择并读入一个影像文件。在 XImageBar 程序中，这个对话框被调用过两次。一次是在组件定义模块中，当一个影像数据未被传入到程序时；另一次则是在 Open Image 按钮被按下的时候。（如果在上一章中没有编写 XImageBar 程序，可以调用与本书配套使用的文档 XImageBar.9.pro。）在第一个程序实例中调用方式如下：

```
image = GetImage    (Cancel = canceled)
```

在第二个程序实例中调用如下：

```
image = GetImage    (Cancel = canceled, Parent = event. Top)
```

在这两次调用过程中有几个微小的区别，不过调用时读者可能没有注意到。如果想编写对话框程序，就必须了解这两次调用的差别以及它们是如何工作的。

阻塞的组件程序

在上述例子中，第一次调用 GetImage 时，GetImage 程序运行时就好比是阻塞的组件程序。要看它是如何工作的，在 IDL 命令行中键入如下所示：

```
IDL > image = GetImage ( )
```

注意到 IDL 命令行要么是消失了要么是变灰了。这时要在 IDL 命令行上键入命令并执行它们是不可能的。我们就称 IDL 命令行是阻塞了。（当命令行正处于消失或变灰状态时，可以键入命令，但只有在命令行解除了阻塞后命令才可以执行。）

一般而言，无论运行哪一种 IDL 程序都会阻塞 IDL 命令行。换言之，只有等到当前的命令执行完毕后才有可能键入并执行另一命令。IDL 5 以前的版本中，所有的组件程序也都是如此运行的。一旦运行了一个组件程序，只有等到这个组件程序执行完毕后才可进入 IDL 命令行。

但在 IDL5 中，刚才的现象已经改变了。现在，可以运行一个组件程序并立即在 IDL 命令行上输入命令然后执行它。我们称这种组件程序为无阻塞组件程序。要创建一个无阻塞的组件程序，只需要程序中的 XManager 命令使用关键字 No_Block 即可。这正如先前编写的程序 XimageBar 中所做的那样。

然而，在程序 GetImage 中的 XManager 命令是没有关键字 Bo_Block 的。因而，它是一种阻塞程序。

在这里的“阻塞”，是指在执行 Xmanager 命令的同时，IDL 就停止执行 GetImage 组件定义模块中的代码。组件定义模块中任何在 XManager 命令下面的代码（其中也有一些可以，待会可以看到）都不能被执行，直到组件程序被销毁后才被执行，也就是说这是阻塞被解除了。

这对像 GetImage 这样的程序是合适的，因为阻塞给用户提供了足够的时间来填写表格或对话框中的信息。当用户添完表格，他们就会点击 Cancel 或 Accept 按钮。无论哪种情况，IDL 都将销毁组件，并解除阻塞，程序也可以用所收集的信息继续工作。对于 GetImage 而言，收集的信息是指关于数据文件、打开文件读取数据，将结果返回给用户。所有收集，读取，返还这一系列的操作都是在组件定义模块中的 XManager 命令执行完毕后才进行。

优点就这些。但是，阻塞也有个细微方面容易被忽视。那就是，只有第一个调用 XManager 为阻塞的组件程序才可以成为阻塞。后来所有的组件程序都将越过阻塞好像他们是无阻塞的组件程序一样。

这种行为对于像 GetImage 这样的程序而言，完全是一个灾难，因为程序将在用户获得填写表格的机会之前就开始读取数据文件。从一个自己对它一无所知的文件中读取数据会带来一些小麻烦。

模式组件程序

如果想确保在任何条件下一个组件程序都会阻塞，而不是仅仅在第一次侥幸地调用 Manager 后才成为阻塞，那么就必须建立一个模式的组件程序。模式的组件程序总是在执行 XManager 命令时处于阻塞状态，直到组件被销毁。

在 IDL5 以前的版本中，编写一个模式的组件程序相对简单些。只要简单地在 XManager 命令后设置关键字 Modal 即可。但在 IDL5 中，Xmanager 命令中的关键字 Modal 已经被废弃了。取而代之的是，在创建顶级 base 时的 Widget_Base 函数中增加一个 Modal 关键字。

这里有一个很小却很重要的补充。如果为一个 base 组件设置 Modal 关键字，同时也必须为那个 base 组建设置一个有效的 Group Leader（通过设置 Group_Leader 关键字）。这对要编写一个既能在 IDL 命令行中运行，又能在程序中运行的对话框程序的难度就更大了。等一会编写程序时或许就会明白这个意思。

编写模式对话框的定义模块

程序 GetImage 是一个对话框程序的绝好例子，但也是相当复杂的。如果是从一个更简单的例子开始，那么就能够更容易理解创建对话框的规则。由于在与本书配套使用的文档下的 Coyote 目录中，有许多类似 2D 字节数组的数据文件，那我们就从编写一个简单对话框程序开始，用于打开和读入 Coyote 目录下的文件。通过调用这个程序，就可以获得这个文件的名字以及在 X 和 Y 方向上的大小，暂且称这个程序为 GetData 吧。

Coyote 目录下的数据文件以及文件大小的详细信息请参阅附录 B：数据文件描述。

如果希望让用户将某些信息输入到表格中，一般来说，这办法不是很好，因为如果（~~我们对~~）用户对此一点也不了解的话，或许他们就无法输入。如果可能的话，让用户选择一个文件名或用鼠标点击选择文件的大小将会是个更好的主意。

假定确实要让用户输入，那么让用户输入的越少越好。解决上述问题一种方法就是，在输入框内提供一些默认值，这些值可能是对的，因此用户也不需要输入。同样，也希望在程序调用时用户能够指定一个文件名或文件的大小。因此，如下打开一个文本编辑窗口，将对

话框的定义语句定义如下：

```
Function GetData, filename, XSize = xsize, YSize = ysize, $
    Cancel = cancel, Parent = parent
```

关键字 Cancel 是一个输出型变量，它用来标示是对话框中的 Cancel 按钮还是 Accept 按钮被按下了。关键字 Parent 包含了模式组件程序的 GroupLeader 的标示符。（喜欢关键字 Group_Leader 的名字吧）记住在定义模式的 base 组件时必须有 Group Leader。

接下来，决定程序在出现错误时怎么处理（作者喜欢将程序控制返回给调用者），以及在没有提供关键字时设置关键字的默认值。增加如下所示：

```
On_Error, 2; Return to caller.
IF N_Elements（filename) EQ 0 THEN $
    filename = Filepath    (Root_Dir=Coyote(), 'ctscan.dat') ELSE $
    filename = Filepath    (Root_Dir=Coyote(), filename)
IF N_Elements (xsize) EQ 0 THEN xsize = 256
IF N_Elements (ysize) EQ 0 THEN ysize = 256
```

这里的 Coyote 命令是用于查找 Coyote 目录的。如果这个目录存在，那么 Coyote 目录将是默认的路径，否则将在当前的目录中查找文件。Filepath 命令将返回一个与设备无关的文件名称。默认的文件为 ctscan.dat，一个有 256X256 的字节数组。

接着，定义对话框的偏移值，此时它被定位于屏幕的中心，键入如下：

```
Device, Get_Screen_size = screenSize
XCenter = FIX (screenSize [0] / 2 . 0)
YCenter = FIX (screenSize [1] / 2 . 0)
Xoff = xCenter - 150
Yoff = yCenter – 150
```

定义一个顶级的模式 base

接下来的就是为这个模式对话框创建一个顶级的 base。在定义一个顶级的模式 base 必须有一个有效的 Group Leader，这在前面已经强调了多次。如果把一个 Group Leader 通过关键字 Parent 传递给程序，那将没什么问题。但事实并非总是如此。例如：如果用户想在 IDL 命令行中调用该程序，一般来讲，这是不太可能获得一个有效的 Group Leader 的。没有有效的 Group Leader，因而就不得不依赖它是一个阻塞组件。再者，如果在 IDL 命令行上第一次调用 Xmanager，这个程序将会阻塞，这时再调用程序也不成问题。

当 Group Leader 没有定义或者程序 GetData 的调用者是它自己本身时，程序就会出现麻烦。作者也不清楚该如何走出这进退两难的境地。更令人不喜欢的是，在另一个程序中如果要正确调用该程序，参数 Group Leader 是必不可少的。此外，当在 IDL 命令行调用它时，Group Leader 参数就不应该是一个必须的参数。

不管如何，如果指定了 Group Leader，就可以创建顶级的模式 base。如果没有指定 Group Leader，那么只有指望在 IDL 命令行上调用该程序了，如下所示：

```
IF N_Elements (parent ) NE 0 THEN $
Tlb = Widget_Base(Column =1 , Xoffset = xoff, Yoffset = yoff, $
Title = 'Enter File Information…', /Modal, $
Group_Leader=parent, /Floating, /Base_Align_Center ) Else $
Tlb=Widget_Base(Column=1, Xoffset=xoff, Yoffset=yoff ,$
Title='Enter File Information…', /Base_Align_Center)
```

注意，设置在顶级的模式 base 上的关键字 Floating。一个浮动组件总是浮现在 Group Leader 程序之上。这可以防止程序隐藏在其他窗口之后。关键字 Base_Align_Center 确保顶

级 base 的子组件在顶级 base 中以居中方式对齐。

定义其他组件

定义一个子 base，来包含文件名域和文件大小域。子 base 并不是真的需要，但使用子 base 可以将在窗体周围设置一个框，并把它与窗体底部的按钮分隔开来。键入如下：

```
Subbase = Widget_Base (tlb, Column = 1)
Filesize = Strlen (filename) * 1.25
FileID = CW_Field (subbase, Title = ‘ Filename:’ ,$
    Value = filename, XSize = filesize)
XsizeID =CW_Field (subbase, Title =’ X Size:’, $
    Value=xsize, / Integer)
YsizeID = CW_Field (subbase, Title = ‘Y Size: ‘ , $
    Value =ysize , / Integer)
```

注意在这里使用的是复合组件 CW_Field。CW_Field 其实就是在一个可编辑文本框旁放置一个标签组件。而它的事件处理函数能够处理大量的细节。例如：设置 CW_Field 的输入值为整数，那么用户只能在输入域输入整数。另外，当使用这个文本框时，它的返回值就是一个整数，而不是字符数组。这样一来使用这些文本组件就变得更加容易了。

接着，建立一个包含 Cancel 按钮和 Accept 按钮的 base 垒，键入：

```
Butbase = Widget_Base(tlb, Row=1)
Cancel = Widget_button (butbase, Value = ‘Cancel’)
Accept = Widget_Button ( butbase, Value= ‘Accept’)
```

现在已经建立了所有所需的组件，因此可以实现该程序：

```
Widget_Control, tlb, / Realize
```

在模式对话框中保存信息

对话框的作用就是从用户那儿收集信息，然后当用户按下 Accept 按钮时将信息返回给用户。（或根据这些信息做相应的操作）。但是，当用户按下 Accept 按钮时，对话框就被销毁，阻塞也被解除。于是问题就出现了：程序所收集的信息应该保存在哪里，进而处理该信息还是将信息返回给用户呢？

很显然，不要将信息保存在程序内部（比如说，保存在用户值中），因为当程序被销毁时信息也会被销毁。所以，信息必须保存在程序外部。公共块是一种方法，但作者喜欢将信息保存在指针内。在程序中创建一个指针，键入如下所示：

```
ptr = ptr_New ({Filename:’ ‘, Cancel : 1, XSize :0, YSize: 0})
```

这个指针指向一个匿名结构，里面包含了所有希望从窗体中收集的信息。注意有一个名为 Cancel 的字段，是用来表明用户是按下了 Cancel 按钮还是 Accept 按钮。这个信息非常重要，因为程序根据不同的值来采取不同的操作。

创建 Info 结构

与其他组件程序一样，本程序也要创建一个 Info 结构来保存程序运行过程中所需的必要的信息。从这一意义上来说，需要保存的信息有包含文件信息的组件标示符以及信息存储的位置。info 结构定义如下，并将它保存在顶级 base 的用户值中。

```
info = { fileID: fileID, xsizeID: xsizeID, $
```

```
        ysizeID:ysizeID, ptr:ptr}
    Widget_Control, tlb, Set_UValue=info, /No_Copy
```

创建一个阻塞组件

可以用 Xmanager 命令注册程序了。注意，必须确保这各程序是一个不能使用 No_Block 关键字的阻塞程序。如果变此程序为无阻塞程序，那么可以在 IDL 命令行上使用它而不必具有有效 Group Leader。键入如下所示：

```
XManager, 'getdata' , tlb, Event_Handler = 'Getdata_Events'
```

从阻塞中返回

在程序执行到上述代码时，IDL 已经停止执行组件定义模块中的代码了。程序的所有操作都发生在时间处理模块中。IDL 不会从阻塞中返回，进而执行组件定义模块的编码，直到用户按下 Cancel 或 Accept 按钮后程序被销毁，阻塞被解除。当这发生后，对话框信息就会保存在指针内。

在程序将信息返回给用户之前，要做的事情就是获取并处理信息。在本程序中，要做的事情就是获取文件名、数据文件大小，然后读取文件并将影像数据返回给程序调用者。

首先，获取指针内的信息。由于正在操作指针，所以可以删除它，如下：

```
fileInfo = * ptr
ptr_free, ptr
```

现在已经有了打开和读取数据文件所需的信息。当然，也可以认为，现在拥有了程序所需要的信息。事实上，程序可能只是获得了不正确的信息。用户可能忘记了键入文件名，或者在输入文件大小的文本框中多加了一位数字，或者他们根本就不知道文件大小而只是猜测而已。事实上，当开始读取这个数据文件时，各种意想不到的事都可能发生，因此最好有个心理准备。

建立一个 Catch 错误捕获语句来捕捉所有可能发生的意想不到的事，以及在读取数据文件时产生的错误。（Catch 错误捕获语句的详细信息请参阅 227 页的“Catch 控制语句”。）键入以下命令：

```
Catch , error
IF error NE 0 THEN BEGIN
    Catch, /Cancel
    Ok = Dialog_Message (!Err_String)
    Cancel = 1
    IF N_Elements (lun) NE 0THEN FREE_LUN, LUN
    Return, -1
ENDIF
```

这个错误处理语句将错误信息显示给用户，（以让他们知道出现了一些情况。）设置 Cancel 标识（因而用户不必使用函数的返回值），如果文件已打开则关闭文件，并返回-1。

接下来，检查一下用户是否按下了 Cancel 按钮。如果是，设置 Cancel 标识并返回-1。键入如下语句：

```
cancel=fileInfo.cancel
IF cancel Then Return, -1
```

好了，可以开始去读数据文件。

```
Image=BytArr(fileInfo.xsize,fileInfo.ysize)
OpenR, LUN, fileInfo.fileName, /Get_Lun
```

```
ReadU, LUN，Image
Free_LUN, LUN
```

如果到了这儿，就已是大功告成了。返回影像数据：

```
Return, Image
End
```

该程序的源代码可参见ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote/getdata.pro文件。

编写模式对话框的事件处理模块

现在可以编写这个模式对话框程序的事件处理模块了。事件处理模块的思路非常简单，因为只需要关心 Cancel 和 Accept 按钮的事件，而对其他事件则是忽略。（当用户回车时，CW_FIELD 组件会产生事件。）如果用户按下了 Cancel 按钮，对话框被销毁，阻塞因而也被解除。如果用户按下 Accept 按钮，程序也要做相同的事，但是同时还得在对话框销毁之前将窗体信息保存在指针内。

事件处理模块的前几行可能向如下所示。（确保将事件处理模块增加到组件定义模块之前。）

```
Pro GetData_Events, Event
EventName = Tag_ Names（Event, /Structure_Name）
If EventName NE ‘WIDGET_BUTTON‘ Then Return
```

注意，Tag_Names 命令将所有不是按钮的事件都输出到屏幕上。

程序只对按钮事件进行处理，获取 info 结构并找出产生事件的按钮。键入如下所示：

```
Widget_Control, Event.Top, Get_Uvalue=info, /No_Copy
Widget_Control, Event.ID, Get_Value=buttonValue
```

如果是 Cancel 按钮，那么只需要销毁对话框即可。这是因为指针初始化时已经设置了适合的值。（例如：将指针结构中的 Cancel 字段设置为 1）。键入：

```
Case buttonValue Of
‘Cancel’: Widget_Control,Event.Top, /Destroy
```

如果是 Accept 按钮，那就必须在对话框销毁之前获取窗体的信息，并将它保存在指针内。Accept 按钮事件处理的代码如下所示：

```
‘Accept’:Begin
    Widget_Control, Info.fileID, Get_Value=fileName
    Widget_Control, Info.xSizeID, Get_Value=xsize
    Widget_Control, Info.ySizeID, Get_Value=ysize
    (*info.ptr).fileName=fileName[0]
    (*info.ptr).xsize=xsize
    (*info.ptr).ysize=ysize
    (*info.ptr).Cancel=0
    End
EndCase
End
```

注意在存入指针之前，变量 fileName 要用下标引用。这是因为即使文本框内只有一个字符串，文本框返回的值还是一个字符串数组。因而必须确保传入到指针的只是一个字符串。

同样注意，指针结构是怎样用下标来引用的。要引用指针内的字段，那么指针一定要用括号括起。

最后，请注意，不一定非得将 info 结构返回给顶级 base 的用户值中，因为这个顶级

base 在任何情况下都会被销毁。

用户可以从以下地址获得 GetData 的源代码：

ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote/getdata.pro

测试模式对话框程序

把程序保存为 GetData.Pro，编译并调用、测试它。如下：

```
IDL>.Compile GetData
IDL>image=GetData（）
```

运行结果看上去像图 90 中的例子。在附录 B 的数据文件描述中，可以找到合适的文件名称以及其大小，用以输入到程序对话框中。

通过上一章的 XImageBar 程序，调用 GetData，查看它是如何工作的。（如果在上一章中自己没有编写 XimageBar 程序，也可以调用与本书配套使用的文档中的 XimageBar.9.Pro）在程序 XimageBar 中查找用到了 GetImage 命令的地方，这应该有两处，并将 GetImage 替换为 GetData。然后保存并编译该程序。运行结果和想象中一样吗？（改变后的 XimageBar.Pro 已经保存为 XimageBar.10.Pro，在与本书配套使用的文档中可以找到）当 GetData 已经显示在屏幕上而准备退出 XimageBar 程序时，结果会如何？结果会是所期望的吗？

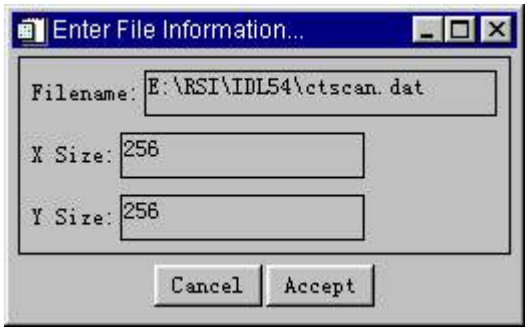


图 90　　GetData 对话框程序

如果程序运行正确，它会浮在 XImagebar 程序之上。当点击 XImagebar 程序时，可以听到一声短而尖的声音，以提醒在使用另一程序之前必须销毁该程序。

完整的 GetData 程序可以在与本书配套使用的文档中找到。

创建非模式的对话框

有时候，不希望在从用户那儿获取信息时所有的事情都停下来。更希望显示对话框是为了方便用户，而不阻塞与之同时显示的程序的使用。这种情况就更难解决了，因为不知道什么时候用户能够准备好这些信息。很可能在用户使用之前该对话框就已经在屏幕上呆了数小时。

在这种情况下，就不得不在程序中采用多个方法来告诉程序用户已经准备好了。为做到这一点，我通常是采用一个 Apply 按钮，这和前面编制的程序练习中的 Accept 相类似。同样，可以用一个等同于前面练习中的 Cancel 的 Dismiss 按钮，其目的在于将对话框销毁。因此，无阻塞对话框看上去如图 91 所示，即后面即将编写的程序 GetFile 的运行结果。

编写非模式对话框程序

GetFile 的组件定义模块和先前的 GetData 程序的定义模块几乎是一样的。但存在着很

大的区别，在下面例子中已用黑体字标出。下面是 GetFile 程序定义模块的完整代码。

```
PRO GETfile, notifyID, Filename=filename, XSize=xsize, $
    YSize=ysize, Parent=parent
On_Error, 2
IF N_Params( ) EQ 0 THEN BEGIN
    Ok = Dialog_Message (‘ Widget ID parameter required.’ )
    Return
ENDIF
IF N_Elements (filename) EQ 0 THEN $
    filename= filepath (Root_Dir=Coyote (), ‘ctscan.dat’ ) ELSE $
    filename=filepath (Root_Dir=Coyote(), filename)
IF N_Elements(xsize) EQ 0 THEN xsize=256
IF N_Elements(ysize) EQ 0 THEN ysize=256
Device, Get_Screen_Size=screenSize
Xcenter=FIX(screenSize(0)/2.0)
Ycenter=FIX(screenSize(1)/2.0)
Xoff=xCenter-150
Yoff=yCenter-150
IF N_Elements(parent) NE  0 THEN $
    Tlb =Widget_Base (Column=1, XOffset=xoffset=xoff, YOffset=yoff,$
    Title=’Enter File Information...’,$
    Group_Leader=Parent,/Base_Align_Center) ELSE $
    Tlb =Widget_Base (Column=1, XOffset=xoffset=xoff, YOffset=yoff,$
    Title=’Enter File Information...’,$
    Subbase=Widget_Base(tlb,Column=1, Frame=1)
Filesize=strlen(filename)*1.25
FileID= CW_Field(subbase, Title= ‘Filename:’,$
    Value=filename, xsize=filesize)
    XsizeID=CW_Field (subbase,Title= ‘ XSize:’,$
        Value=xsize,/Integer)
    YsizeID=CW_Field (subbase,Title= ‘ YSize:’,$
        Value=ysize,/Integer)
    Butbase=Widget_Base(tlb,row=1)
    Dismiss=Widget_Button(butbase, Value= ‘Dismiss’)
    Apply=Widget_Button(butbase, Value= ‘Apply’)
    Widget_Control, tlb,/ Realize
    Info={fileID:fileID, xsizeID: xsizeID, $
         YsizeID:YsizeID, notifyID:notifyID}
    Widget_Control, tlb, set_uvalue=info,/no_copy
    Xmanager,’getfile’, tlb, Event_Handler= ‘getfile_events’, /no_block
    END
```

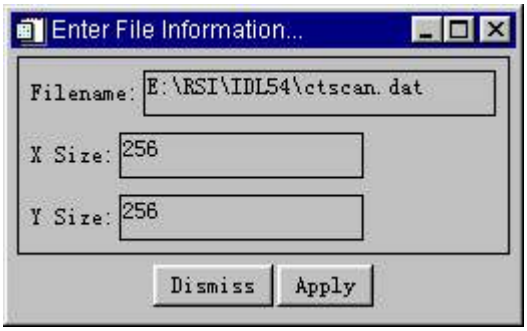


图 91　无阻塞对话框程序 GetFile，它类似于图 90 的阻塞对话框 GetData。按钮上的文本表明了这是一个无阻塞程序

通报程序事件的组件

代码中最大的变化是在组件定义模块的第一行，以前是函数现在变成了过程。

```
PRO GetFile, notifyID, Filename=filename, Xsize=xsize, YSize=ysize, Parent=parent
```

这个变化的原因在于此时不能像函数一样返回信息，因为当用户填完表格后，程序不会停止或阻塞。因而必须找到另一种技术来告诉程序调用者或其他程序，可以获取用户信息了。

在这里，通过参数 notifyID，一个组件标识符的矢量来通报事件是可行的。当用户按下 Apply 按钮时，窗体中的信息被收集后，通过向对话框中所有的组件发送一个含有窗体信息的事件，因而参数 notifyID 所标识的组件将被告知。（这种技术在 274 页中的“创建事件并将事件传递给其他程序”已经讨论过。）

参数 notifyID 是一个含有组件标识符的 2*N 数组。，第一个列是当按下 Apply 按钮时即将被告知的组件标示符，第二个列是第一列所标示的组件的顶级 base 的标示符。在实际中，通常在同一个事件处理模块中调用程序 GetFile。如下所示

```
GetFile, [event.id, event.top]
```

第二个变化是在组件定义模块中将参数 notifyID 变为一个必须参数，见下：

```
IF N_Params() EQ  0 THEN BEGIN
    Ok=Dialog_Message (‘ Widget ID parameter required.’)
    RETURN
ENDIF
```

顶级 base 的定义方式因此也相应地变了：

```
IF N_Elents (parent) NE 0 THEN $
    Tlb=Widget_Base(Column=1.XOffset=xoff, YOffset=yoff, $
    Title= ‘Enter File Infoemation...’, $
    Group_Leader=Parent, / Base_Align_Center) ELSE $
    Tlb=Widget_Base(Column=1.XOffset=xoff, YOffset=yoff, $
    Title= ‘Enter File Infoemation...’, / Base_Align_Center)
```

注意，由于这是一个非模式对话框件，因而不需要使用关键字 Modal 或 Floating 了，参数 Parent 如果定义了，那它仍然作为对话框的 Group Leader 来使用。用户可以通过选择 Parent 关键字来决定该组件程序是否作为另一组件程序的成员。

尽管按钮的名字变成了 Dismiss 和 Apply，但它们的功能与 GetData 程序中的 Cancel 和 Accept 按钮功能相类似。

```
Dismiss=Widget_Button ( butbase, Value= ‘ Dismiss’)
Apply=Widget_Button (butbase, Value= ‘ Apply’)
```

改变按钮名字是对用户来讲是一个更为直观的功能提示。换言之，看到 Cancel 和 Accept

按钮，用户就会期望这是模式对话框。看到 Dismiss 和 Apply 按钮，用户期望的就是非模式对话框。如果将界面尽可能保持一致的风格，用户就会少一些困惑。

程序中没有使用指针，因为没有必要使用全局的内存。但是，组件的标识符矢量（用来标识即将被通知的组件）必须存储在 info 结构中，因为在事件处理模块中需要用到这些信息。

```
Info={fileID:fileID, xsizeID:xsizeID, $
    YsizeID:ysizeID, notifyID:notifyID}
```

组件定义模块中的最后一个变化是通过 XManager 命令中的 No_Block 关键字将阻塞组件程序转换为无阻塞组件程序。如下：

```
XManager, ‘ getfile’, tlb, Event_Handler= ‘GetFile_Events’, /No_Block
```

编写非模式对话框的事件处理模块

程序 GetData 和 GetFile 的最大差别在于当用户按下 Accept 或 Apply 按钮时，事件处理模块是如何工作的。而其余的事件处理方式基本是相同的。下面是事件处理代码的前半部分，细微差别已用黑体字标出：

```
Pro GetFile_Events, event
EventName=Tag_Names (event, /Structure_Name)
IF eventName NE’ widget_button’ then return
Widget_Control, event.top, Get_UValue=info, / No_COPY
Widget_Control, event.id, Get_UValue=buttonValue
CASE buttonValue Of
‘Dismiss’: Widget_Control, event.top, /Destroy
```

将事件发送给其他组件

当用户点击 Apply 按钮时，事件处理程序将对话框中的信息收集，并提示 notifyID 所指定的组件事件已经发生了。这里是通过建立伪事件，并用 Widget_Control 命令的 Send_Event 关键字发送事件来完成的。第二部分如下所示，不同之处已用黑体字标出：

```
‘Apply’: BEGIN
Widget_Control, info.fileID, Get_Value=filename
Widget_Control, info.xsizeID, Get_Value=xsize
Widget_Control, info.ysizeID, Get_Value=ysize
S=Size(info.notifyID)
If s[0] EQ 1 THEN count=0 ELSE count=s[2] -1
FOR j=0, count DO BEGIN
   PseudoEvent={GETFILE_EVENT, ID:info.notifyID[0,j], $
               Top:info.notifyID[1,j], Handler:0L, $
               Filename:filename[0], XSize:xsize, YSize:ysize}
   IF Widget_Info (info.notifyID[0,j], / Valid_ID) THEN $
       Widget_Control, info.notifyID[0,j], Send_Event=pseudoEvent
ENDFOR
Widget_Control, event.top, Set_UValue=info, / No_Copy
END
ENDCSAE
```

```
END
```

伪事件是一个名为 GETFILE_EVENT 的事件结构，字段 ID 包含了接收这个事件的组件标识符。字段 TOP 标识了组件结构层次图中的顶级 base。Handler 字段是空的，但在事件传递到组件之前，IDL 自己能够正确地填写。在这里所做的只是创建几个长整型的字段，即 fileName、Xsize 和 Ysize，它们包含了从对话框中所获得的信息。

如果接收到事件结构的组件仍是个有效组件（即组件仍然存在），那么伪事件就将通过 Widget_Control 命令中的 Send_Event 关键字发送给这个组件。

注意，当按下 Apply 按钮时，对话框不会被销毁，仍将在屏幕上显示，直到按下了 Dismiss 按钮。由于对话框未被销毁，因而在退出事件处理程序之前必须把 info 结构放回它的存储位置中去。

GetFile 程序在与本书配套使用的文档中可以找到，文件名为 GetFile.Pro。该程序的源代码在ftp://ftp.dfanning.com/pub/fanning/outgoing/coyote/getdata.pro。

测试非模态对话程序

将程序保存为 GetFile.Pro，然后在 XImageBar 程序中选择 File/Open...按钮，调用这个程序，看它是否运行正常。

然而，在做这之前，必须对 XimageBar_Open 事件处理程序作相应的修改。尤其是，必须将来自于 File/Open...按钮的按钮事件和来自于 GetFile 的伪事件区分开来。

现行的 XimageBar_Open 事件处理代码编写如下：

```
PRO XimageBar_Open, event
Image=GetData (Cancel=celed, Parent=event.top)
IF canceled THEN RETURN
S = Size(image)
IF s [0] NE 2 THEN BEGIN
    Message, ‘ Image argument must be 2D.’, / Continue
    Return
ENDIF
Widget_Control, event.top, Get_UValue= info, / No_Copy
*info.image=image
pseudoEvent={WIDGET_BUTTON, ID: info.action, $
    Top; event.top, Handler:0L , Select:1}
Widget_Control, info.action, Send_Event=pseudoEvent
Widget_Control, event.top, Set_UValue=info, / No_Copy
END
```

修改后的代码如下，修改部分已用粗体字标出：

```
PRO XimageBar_Open, event
Widget_Control, event.top, Get_UValue= info, / No_Copy
ThisEvent=Tag_Names(event, / Structure_Name)
IF thisEvent EQ ‘WIDGET_BUTTON’ THEN BEGIN
    GetFile, [event.id, event.top], Parent=event.top
    Widget_Control, event.top, Set_UValue=info, / No_Copy
    RETURN
ENDIF
IF thisEvent EQ ‘ GETFILE_EVENT’ THEN BEGIN
    Image=BytARR(event.xsize, event.ysize)
```

```
            OpenR, lun, event.filename, / Get_Lun
            ReadU, lun, image
            Free_Lun, lun
            *info.image=image
            pseudoEvent={ WIDGET_BUTTON, ID:info.action, $
                Top:event.top, Handler: 0L, Select:1}
            Widget_Control, info.action, Send_Event=pseudoEvent
            Widget_Control, event.top, Set_UValue=info, / No_Copy
            RETURN
        ENDIF
        END
```

现在，这个时间处理程序可以处理两种不同类型的事件：一种是来自 File/Open...按钮的按钮事件，一种是来自 GetFile 程序的伪事件。如果事件来自 GetFile 程序，那么有关即将打开和读取的数据文件的信息包含在事件结构中。

运行一下程序。它是如何工作的呢？能不能在销毁 GetFile 程序之前打开其他影像数据？（与 GetFile 程序一起使用的 XimageBar 程序可以在与本书配套使用的文档中找到，文件名为 XimageBar.11.Pro。）

NotifyID 技术是一个在组件程序之间进行数据交流的有效方法，而不必使用公共块。

附录 A　组件的事件结构

事件结构的定义

事件结构包含了特定组件的相关信息。每个事件都产生于它自身的、特定的事件结构中，然后被发送到事件处理模块。事件结构可以是命名的也可以是匿名的 IDL 结构变量。使事件结构与其他结构区分开来的是，事件结构有 ID、Top 和 Handler 三个字段。这三个字段都是长整型。下面列举了由 IDL 组件创建或返回的事件结构。

公共字段的定义

字段 ID，通常是长整型，是产生事件的组件的惟一标示符，

产生事件的组件通常是组件层次结构中的一部分。那么字段 Top 就是该层次结构中顶级 base 的惟一标示符，它通常也是一个长整型的数。

事件产生的事件结构都要发送到事件处理程序中。每个事件处理程序都与某一个组件联系在一起。而字段 Handler，就是那个与事件处理程序相联系的组件的标识符。它通常是个长整型的数.

这些公共字段的详细信息以及它们是如何定义的，请参阅 256 页的“公共字段的定义”

基本组件的事件结构

base 组件

```
{WIDGET_BASE, ID:0L, Top:0L, Handler:0L, X:0, Y:0}
```

只有顶级 base，并且是在用户改变其大小的情况下才会产生事件。要产生事件，关键字 TLB_Size_Events 必须被显式地指定。字段 X 和 Y 是顶级 base 的尺寸大小，以像素为单位。Base 的尺寸大小并不包含窗口的任何边框。

按钮组件

```
{WIDGET_BUTTON, ID:0L, Top:0L, Handler:0L, Select:0}
```

如果按钮被选中，那么字段 Select 被设置为 1；如果按钮被释放，那么就被设为 0。一般的按钮在被释放时并不产生事件，因此 Select 字段一直为 1。然而，单选按钮对选中和释放动作会分别产生事件。

绘图组件

```
{WIDGET_DRAW, ID:0L, Top:0L, Handler:0L, Type:0, X:0, Y:0, Press:0,
```

Release:0,Clicks:0}

字段 Type 将告诉事件的类型，它可能的值有：鼠标按下（0）；鼠标松开（1）；鼠标移动（2）；视点滚动（3）和显示（4）。上面所有的事件都必须显式地指定，否则相应的事件是不会产生的。

字段 X 和 Y 给出了事件发生时的设备或屏幕坐标，绘图组件的左下角为坐标原点。字段 Press 和 Release 都是位掩码，当鼠标被按下或释放时，它们的值就分别存在了。其中 1 表示鼠标左键，2 表示鼠标中键，4 表示鼠标右键。如果是个鼠标移动事件，那么 Press 和 Release 都被设置为 0。

当鼠标单击时，字段 Clicks 返回 1，双击时返回 2。

下拉式列表组件

{WIDGET_DROPLIST, ID:0L, Top:0L, Handler:0L, Index:0L}

字段 Index 返回所选择列表项的索引号。通过它，可以对最初赋予该下拉式列表的数组进行引用。（数组应该存放于组件的用户值中或其他什么地方。）

标签组件

标签组件本身并不产生事件，但可以通过它来设置 Timer 事件。

列表组件

{WIDGET_LIST, ID:0L, Top:0L, Handler:0L, Index:0L, Clicks:0L }

字段 Index 返回所选项的索引号，通过索引号，可以对最初赋给列表的数组进行引用。字段 Clicks 返回 1 或 2，这取决于列表项是如何选择的。如果列表项被双击选中，Clicks 返回 2。注意，在事件处理程序中，单击和双击的信息都可以获得。

滑动条组件

{WIDGET_SLIDER, ID:0L, Top:0L, Handler:0L, Value:0, Drag:0}

字段 Value 返回滑动条的新值。当滑动条在拽动过程中，字段 Drag 返回 1，拽动操作结束，则 Drag 返回 0。注意，只有在 UNIX 操作系统上，并且设置了 Drag 关键字，滑动条才会产生 Drag 事件。

表单组件

插入单个字符事件

{WIDGET_TABLE_CH, ID:0L, Top:0L, Handler:0L, Type:0, Offset:0L, CH:0B, X: 0L, Y:0L}

字段 Offset 是包含的是一个以 0 为基数的插入点位置，插入的字符在插入点之后。字段 Ch 是当前插入字符的 ASCII 码，字段 X 和 Y，则表示的是当前单元格在表单中的位

置，也是以 0 为基数的。

插入字符串事件

{WIDGET_TABLE_STR, ID:0L, Top:0L, Handler:0L, Type:1, Offset:0L, Str:", X: 0L, Y:0L }

字段 Str 是要插入的字符串。

删除字符串事件

{WIDGET_TABLE_DEL, ID:0L, Top:0L, Handler:0L, Type:2, Offset:0L, Length:0L, X: 0L, Y:0L }

字段 Offset 是要删除的第一个字符所在的位置，它是以 0 为基数的。同时它也是下一个字符的插入点。字段 Length 表示删除字符的个数。

选择文本事件

{WIDGET_TABLE_TEXT_SEL, ID:0L, Top:0L, Handler:0L, Type:3, Offset:0L, Length:0L, X: 0L, Y:0L }

当插入点的位置发生变化时，这个事件就会被引发。字段 Offset 是所选文本的第一个字符的位置，以 0 为基数。字段 Length 为所选字符的个数。如果 Length 为 0，则表明没有选中，且字符插入点也被设置为 Offset 值。

选择单元事件

{WIDGET_TABLE_CELL_SEL, ID:0L, Top:0L, Handler:0L, Type:4, Sel_Left:0L, Sel_Top:0L, Sel_Right:0L, Sel_Bottom:0L}

当前所选择的单元格发生变化时，这个事件会被引发。所选择的单元格范围由字段 Sel_Left、Sel_Top、Sel_Right 和 Sel_Bottom 确定，它们都是以 0 为基数。当所选择的单元格在取消选择的事件发生时（选择发生改变或点击表单左上角会导致这个事件的发生），字段 Sel_Left、Sel_Top、Sel_Right 和 Sel_Bottom 的值都为-1。

注意，这就意味着，当选择焦点从一个已选中的单元格到一个新单元格时，会产生两次 WIDGET_TABLE_CELL_SEL 事件。在程序中，要将选择和取消选择区分开来。

改变行高事件

{WIDGET_TABLE_ROW_HEIGHT, ID:0L, Top:0L, Handler:0L, Type:6, Row:0L, Height:0L}

当用户改变表单给定行的高度时会引发这个事件。字段 Row 包含了所选定行的值(以 0 为基数)，字段 Height 包含的是改变后的高度。

改变列宽事件

{WIDGET_TABLE_COLUMN_WIDTH, ID:0L, Top:0L, Handler:0L, Type:7, Column:0L,

Width:0L}

当用户改变表单给定列的宽度时会引发这个事件。字段 Column 包含了所选定列的值(以 0 为基数)，字段 Width 包含的是改变后的高度。

无效数据事件

{WIDGET_TABLE_INVALID_DATA, ID:0L, Top:0L, Handler:0L, Type:8, Str:", X:0L ,Y:0L}

当这个事件发生时,所在的单元格的数据保持不变。用户输入的无效数据保存在字段 Str 中。单元格的位置由字段 X 和 Y 来确定。

文本组件

插入字符事件

{WIDGET_TEXT_CH,ID:0L, Top:0L, Handler:0L, Type:0, Offset:0L, Ch:0B}

字段 Offset 是当前插入点的位置（以 0 为基数)。字符 Ch 为插入字符的 ASCII 码。

插入字符串事件

{WIDGET_TEXT_STR,ID:0L, Top:0L, Handler:0L, Type:1, Offset:0L, Str:" }

字段 Str 为插入的字符串。

删除字符串事件

{WIDGET_TEXT_STR,ID:0L, Top:0L, Handler:0L, Type:2, Offset:0L, Length:0L }

字段 Offset 为要删除的第一个字符串的位置（以 0 为基数)，它同时也是下一个字符的插入点。字段 Length 为要删除的个数。

文本选择事件

{ WIDGET_TEXT_STR,ID:0L, Top:0L, Handler:0L, Type:3, Offset:0L, Length:0L }

插入点的改变会引发这个事件的产生。字段 Offset 为所选择的第一个字符的位置（以 0 为基数)。字段 Length 为要所选择字符的个数。Length 表明没有选中字符，并且插入点被设置为 Offset 的值。

复合组件的事件结构

CW_Animate

{ID:0L, Top:0L, Handler:0L, Action:0L}

字段 Action 中惟一允许的字符串就是“DONE”

CW_Arcball

{ ID:0L, Top:0L, Handler:0L, Value:FltArr(3,3)}
字段 Value 包含了新的旋转矩阵。

CW_BGroup

{ ID:0L, Top:0L, Handler:0L, Select:0, Value:*}
字段 Select 包含了新的旋转矩阵。字段 Value 可以是按钮的 Index、ID、Name 和按钮用户值，这取决于组件创建时情况。

CW_Clr_Index

{CW_COLOR_INDEX, ID:0L, Top:0L, Handler:0L, Value:0}
字段 Value 为所选择的颜色的索引号。

CW_Color_Sel

{COLORSEL_EVENT, ID:0L, Top:0L, Handler:0L, Value:0}
字段 Value 为所选择的颜色的索引号。

CW_DefROI

这个复合组件有内部事件处理程序，并且不会产生任何外部事件。

CW_Field

{ ID:0L, Top:0L, Handler:0L, Value:'',Type:0, Update:0}
字段 Value 包含了文本组件中的值。字段 Type 指定了数据的类型，它可能的值有：0（字符串），1（浮点数），2（整数）或 3（长整型）。如果没有更新，则字段 Update 返回 0，否则返回 1。

CW_Form

{ ID:0L, Top:0L, Handler:0L, Tag:'',Value:0, Quit:0}
字段 Tag 包含了那些发生了变化的字段标签。字段 Value 则是包含了变化后的新值；字段；如果 Quit 被设置，那么 Quit 返回 0，否则返回 1。

CW_Flisder

{ ID:0L, Top:0L, Handler:0L,Value:0.0, Drag:0}

字段 Value 返回滑动条的当前值。当滑动条在拽动过程中，字段 Drag 返回 1，拽动操作结束，则 Drag 返回 0。

CW_Orient

当组件内部不同于子组件改变变换矩阵时，这个复合组件产生不同的事件。一般来讲，这个组件在字段 ID 第一次发生变化时，将内部组件所产生的的事件直接传送到本组件的顶级 base 中。大多数事件处理程序都忽略该复合组件的事件，因为系统变量!P.T 在任何时候都是自动更新的。

CW_PDMenu

{ID:0L, Top:0L, Handler:0L, Value:*}

字段 Value 既可以是按钮组件的 Index、ID、Name，也可以是按钮的 Full_Name，这取决于按钮组件是如何创建的。

CW_RGBSlider

{ID:0L, Top:0L, Handler:0L, R:0B, G:0B, B:0B}

字段 R、G、B 分别代表所选择颜色的红、绿、蓝的值。

CW_Zoom

{ZOOM_EVENT,ID:0L, Top:0L, Handler:0L, XSize:0L, Ysize:0L,
X0:0L, Y0:0L, X1:0L, Y1:0L }

字段 Xsize 和 Ysize 包含了缩放后图像的大小，字段 X0，Y0，X1，Y1 分别包含了原始图像左下角和右上角的坐标。

组件程序的事件结构

Xcolors

{XCOLORS_LOAD, ID:0L, Top:0L, Handler:0L, R:BytArr(!D.Table_Size), G:BytArr(!D.Table_Size), B:BytArr(!D.Table_Size), Index:0}

Xcolors 将向由 NotifyID 所标识的组件发送事件。字段 R、G、B 分别包含了当前颜色表的红、绿、蓝的向量值。字段 Index 设置为-1，或者在载入了颜色表后设置为颜色表的索引值。

其他组件的事件结构

下面所列出的是组件产生的其他事件。

键盘焦点事件

{WIDGET_KBRD_FOCUS, ID:0L, Top:0L, Handler:0L, Enter:0L}

某些组件，如文本框，在设置了关键字 KBRD_FOCUS_EVENTS 后可以产生键盘焦点事件。当这些组件获得键盘焦点时（字段 Enter 设置为 1）或失去键盘焦点时（字段 Enter 设置为 0）会产生这个事件。

组件退出请求事件

{WIDGET_KILL_REQUEST, ID:0L, Top:0L, Handler:0L}

设置了关键字 TLB_KILL_REQUEST_EVENTS 的顶级 base 在窗口管理器销毁该组件（比如，用户用鼠标关闭窗口，而不是点击 Quit 按钮）时就会接收到这个事件。不管销毁事件是来自 Quit 按钮，还是来自窗口管理器，调用 CleanUp 过程来处理任何组件的销毁事件会更加简单。（CleanUp 过程的详细信息请参阅 278 页的“使用 CleanUp 过程防止内存泄漏”）

组建计时器事件

{WIDGET_TIMER, ID:0L, Top:0L, Handler:0L}

事件处理程序在接收到计时器事件时可以做任何事情。字段 ID 是用来标识设置计时器的组件。

组件跟踪事件

{WIDGET_TRACKING, ID:0L, Top:0L, Handler:0L, Enter:0}

每次鼠标进入到组件（字段 Enter 设置为 1）或从组件出来（字段 Enter 设置为 0）时都产生这个事件。对于指定的组件，如果要产生这个事件，则必须设置关键字 Tracking_Events。注意，在 Windows 操作系统下，跟踪事件运行得不是很好。

附录 B　数据文件描述

在这里，列举了本书所涉及到的有关数据文件。这些文件可以从下面网站中下载到：
ftp://ftp.dfanning.com/pub/dfanning/outgoing/coyote
同时，这些文件也可以在 IDL 的目录下找到。详细信息请参阅 7 页的“拷贝数据文件”。

文件名称	描　述	类型	X	Y	Z
abnorm.dat	开有小孔的血小球	字节	64	64	15
cereb.dat	大脑的 X 射线影像	字节	512	512	1
convec.dat	地幔对流图像	字节	248	248	1
ctscan.dat	胸腔 CT 扫描图像	字节	256	256	1
galaxy.dat	银河系图像	字节	256	256	1
head.dat	人头的 MRI 切片组	字节	80	100	57
hurric.dat	Gilbert 飓风数据	字节	440	340	1
image24.dat	彩色的世界海拔高程数据	字节	3	360	360
jet.dat	水利模拟	字节	81	40	101
m51.dat	M51 星系图像	字节	340	440	1
people.dat	RSI 公司的创始人图像	字节	192	192	2
worldelv.dat	世界海拔高程数据	字节	360	360	1

封底