当我开始在键盘上敲打出这句话的时候，我已经使用 LabVIEW 7 年了。7 年的时间，就算天赋平平也可以积攒下一箩筐可供参考的经验了。所以我打算利用今后的闲暇时间写一些这方面的东西，既可以同大家交流，也是作为自己这七年工作的总结。

还是在上大学的时候，有一次老师让编写一段软件，用来模拟一个控制系统：给它一个激励信号，然后显示出它的输出信号。那时我就想过，可以把每一个简单的传递函数都做成一个个小方块，使用的时候可以选择需要的函数模块，用线把它们连起来，这样就可以方便地搭建出各种复杂系统。

后来，我第一次看到别人给我演示的 LabVIEW 编程，就是把一些小方块用线连起来，完成了一段程序。我当时就感觉到，这和我曾经有过的想法多么相似啊。一种亲切感油然而生，从此我对 LabVIEW 的喜爱就一直胜过其他的编程语言。

LabVIEW 的第一个版本发布于1986年，是在 Macintosh 机上实现的，后来才移植到了 PC 机上，并且 LabVIEW 从未放弃过对跨平台的支持。这也给 LabVIEW 带来了一些麻烦。最明显的就是 LabVIEW 开发环境的界面风格。它总是与一般的 Windows 应用程序有些格格不入：面板是深灰色的，按键钮是看起来别别扭扭的 3D 模样。还有一些可能不太容易发现：比如对于整数的存储，LabVIEW 即便是运行在 x86系统上，采用的也是高地址位存高位数据(big-ending)。这与我们习惯了的 x86 CPU 使用的格式正相反，这往往给编写存取二进制文件带来了不多不少的麻烦。

我接触过的最早的 LabVIEW 版本是4.0版，发布包是一个装有十几张三寸软盘的大盒子。安装的时候要按顺序把软盘一个一个塞到计算机里。尽管当时 LabVIEW 的界面不是很好看，但我还是非常喜欢它。真方便呐!比如说要画一个开关，用 LabVIEW 一拖就行了。如果要自己动手用 C 语言设计一个好看的开关,，那得费多少时间啊!我尤其喜欢它通过连线来编程的方式，尽管很多熟悉了文本编程语言的人刚开始时会对这种图形化编程方式非常不适应。

从 4.0 到现在的 8.2，LabVIEW 的一些技术革新给我留下了非常深的印象。比如说 LabVIEW 5 中实现了多线程运行; LabVIEW 6 里漂亮的 3D 控件，和对事件响应的支持;LabVIEW 7 中的 Express VI 以及 LabVIEW 8 中的工程库。这些新特性都已成为了现在 LabVIEW 版本中富有特色并不可缺少的一部分了。

我对使用 LabVIEW 编程的认识在这些年里经历了不少转变。刚开始接触 LabVIEW 的时候，第一印象就是觉得这东西编程序比 C 语言简单多了，尤其在设计界面时确实方便简易。LabVIEW 是一种真正意义上的图形化编程语言。与文本编程语言，如 C、Basic 等相比，它在编程过程中有更详细的提示信息，如函数的功能、参数类型等等，程序员再不需要去记忆这些枯燥的信息了。编写风格良好的图形程序代码要比文本代码更加清晰直观，便于阅读。

刚开始用 LabVIEW 编程时，我连一本相关的书籍都没读过，差不多完全靠自己摸索。当时，市面上几乎没有有关 LabVIEW 的中文书籍，而阅读英文资料又感觉得太累。但是，靠自己摸索的方法也有好处，最明显的就是有成就感：自己琢磨着解决了一个问题，比模仿别人的方法做更令人兴奋。再者，他人的方案并不一定是最佳的，自己独自思索就不至于被他人的方案局限住思路。

当然，我不会满足于只用 LabVIEW 编写一些简单的程序。我还希望能够用它来编写大型的软件，并且提高自己的开发效率。这时，自己的编程水平有一个质的提高，不阅读相关的书籍资料就不行了。有些问题，不读书，自己可能永远都得不到最佳的答案。同样，有些 LabVIEW 的功能，如果不阅读原始资料，自己也许永远都掌握不了。于是，我把能得到的 LabVIEW 的中高级教程都看了一遍。因为自己有了一定的基础，我就可以在读书的过程中反思自己以前的编程方法是否合理，高效。我觉得最好的 LabVIEW 教

程还是 NI 自己编写的 LabVIEW 中高级教程。但书本中一般原理讲得多，具体的编程技巧涉及得少，所以还必须大量阅读别人的代码，才能学习到更多更好的编程方法。

随着时间的流逝，我慢慢地产生了 LabVIEW 应当进一步改善的想法。作为一名忠实的 LabVIEW 语言的使用者，我衷心地期望着 LabVIEW 在日后也可以成为一种被广为使用的通用编程语言，能够在更广泛的领域与 C，Java 等争得一席之地。LabVIEW 虽然有它独特的优势，但其不足之处也很明显。我在编程过程中就曾感觉到它很多的使用不方便之处。下一步，我追求的目标就是能尽自己所能，对 LabVIEW 作一些改进和完善，使它更适应于通用编程之用。

一. 程序执行顺序

LabVIEW 是数据流驱动的编程语言。程序在执行时按照数据在连线上的流动方向执行。同时，LabVIEW 是自动多线程的编程语言。如果在程序中有两个并行放置、它们之间没有任何连线的模块，则 LabVIEW 会把它们放置到不同的线程中，并行执行。

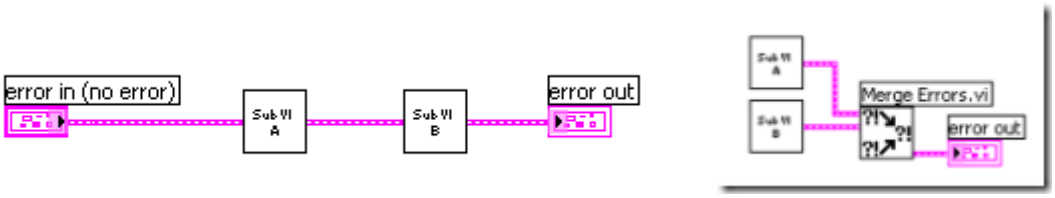


图1、2：顺序执行 和 并行执行 的例子

顺序执行（图1）：数据会从控制控件流向显示型控件，因此数据流经的顺序为“error in”控件，“SubVI A”，“SubVI B”，“error out”控件，这也是这个 VI 的执行顺序。

并行执行（图2）：“SubVI A”，“SubVI B”没有数据线相互连接，它们会自动被并行执行。所以这个 VI 的执行顺序是“SubVI A”，“SubVI B”同时执行，当它们都执行完成以后，再执行“Merge Errors.vi”。

二. 顺序结构

如果需要让几个没有互相连线的 VI，按照一定的顺序执行，可以使用顺序结构来完成（Sequence Structure）。

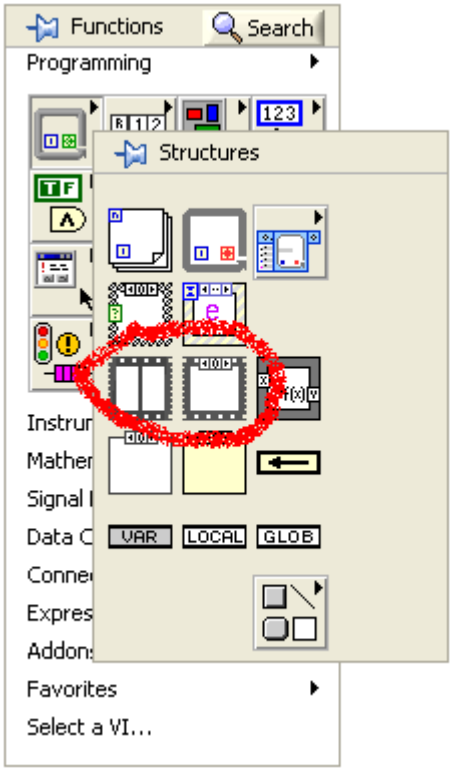


图3：Menu Palette

当程序运行到顺序结构时，会按照一个框架接着一个框架的顺序依次执行。每个框架中的代码全部执行结束，才会再开始执行下一个框架。把代码放置在不同的框架中就可以保证它们的执行顺序。

LabVIEW 有两种顺序结构，分别是层叠式顺序结构（Stacked Sequence Structure）、平铺式顺序结构（Flat Sequence Structure）。这两种顺序结构功能完全相同。平铺式顺序结构把所有的框架按照从左到右的顺序展开在 VI 的框图上；而层叠式顺序结构的每个框架是重叠的，只有一个框架可以直接在 VI 的框图上显示出来。在层叠式顺序的不同的框架之间如需要传递数据，需要使用顺序结构局部变量（Sequence Local）方可。

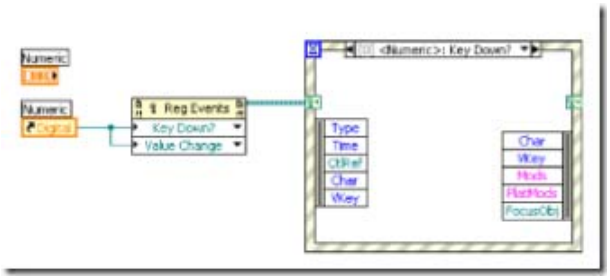


图4：层叠式顺序结构

三. 顺序结构的使用

好的编程风格应尽可能少使用层叠式顺序结构。层叠式顺序结构的优点是及部分代码重迭在一起，可以减少代码占用的屏幕空间。但它的缺点也是显而易见的：因为每次只能看到程序的部分代码，尤其是当使用 sequence local 传递数据时，要搞清楚数据是从哪里传来的或传到哪里去就比较麻烦。

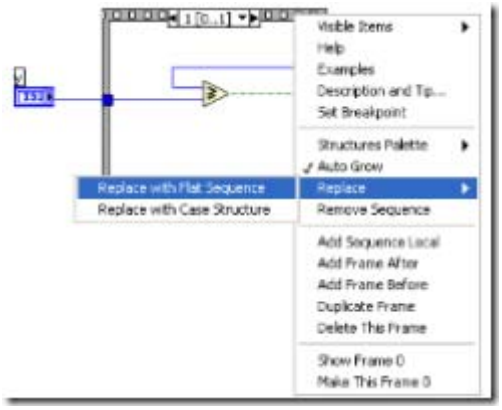


图5：转换顺序结构

使用平铺式顺序结构可以大大提高程序的可读性，但一个编写得好的 VI 是可以不使用任何顺序结构的。由于 LabVIEW 是数据流驱动的编程语言，那么完全可以使用 VI 间连线来保证程序的运行顺序。对于原本没有可连线的 LabVIEW 自带函数，比如延时函数，也可以为其包装一个 VI，并使用 error in, error out，这样就可以为使用它的 VI 提供连线，以保证运行顺序。

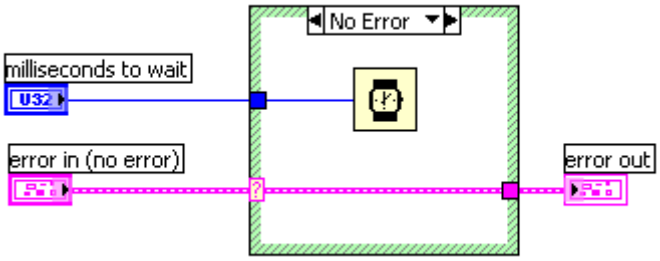

No Error
milliseconds to wait
U32
error in (no error)
error out

图6：改进的延时 VI

选择结构

选择结构相当于文本语言中的条件语句。LabVIEW 8 中新增加的 Diagram Disable Structure，Conditional Disabled Structure 类似 C 语言中的条件宏定义语句。

一. 程序框图禁用结构（Diagram Disable Structure）

在调试程序时常常会用到程序框图禁用结构。程序框图禁用结构中只有 Enabled 的一页会在运行时执行，而 Disabled 页是被禁用、即不会执行的；并且在运行时，Disable 页面里的 SubVI 不会被调入内存。所以，被禁用的页面如果有语法错误也不会影响整个程序的运行。这是一般选择结构（Case Structure）无法做到的。

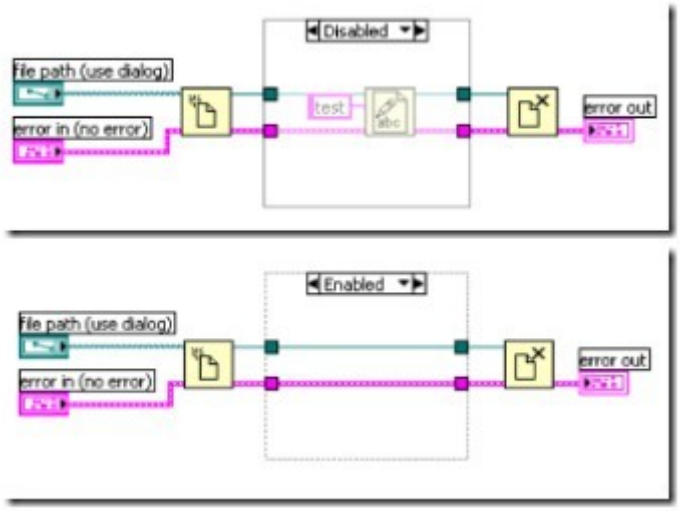


图1、2：使用程序框图禁用结构

例如图 1、2 中的示例，如果我们在运行程序的时候暂时不希望将 test 写入到文件里，但又觉得有可能以后会用到。此时，就可以使用程序框图禁用结构把不需要得程序禁用掉。需要注意的是程序框图禁用结构可以有多个被禁用的框架，但必须有且只能有一个被使用的框架。在被使用的框架中，一定要实现正确的逻辑，比如上图的例子中，在被使用的框架中一定要有连线把前后的文件句柄和错误处理联接好。

二. 条件禁用结构（Conditional Disabled Structure）

条件禁用结构则根据用户设定的符号（symbol）的值来决定执行哪一页面上的程序。其他方面与程序框图禁用结构相同。

程序中所使用的符号，可以在项目或是运行目标机器（例如“My Computer”）的属性里设置。

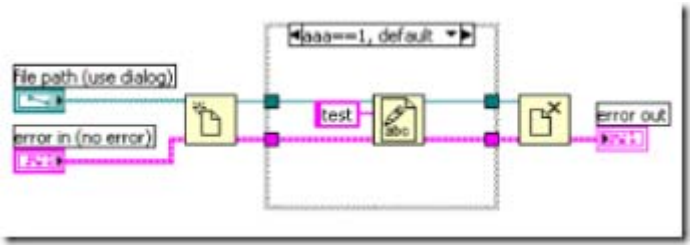


图3：条件禁用结构

值得注意的是：程序框图禁用结构与条件禁用结构都是静态的，如果需要在运行时决定执行哪一部分的程序可以使用选择结构。

程序框图禁用结构和条件禁用结构的一种实用案例可以参考：《其它常用调试工具和方法》

三. 选择结构（Case Structure）

在一般情况下，选择结构类似于 C 语言的 switch 语句。当输入为 bool 数据类型或 error 数据类型时，选择结构类似于 C 语言中的 if 语句。

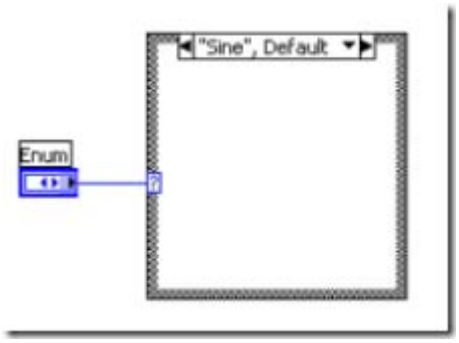


图4：枚举类型的 Case Selector

有输出时，则每一个框架中都必须连一个数据，当然也可以选择“Use Default If Unwired”。选择“Use Default If Unwired”会有一定的风险，因为你可能会忘记了连线，这时候 LabVIEW 并不会提醒你，程序就可能得到不可预料的结果。

如图5所示，鼠标右击数据输出隧道，可以选择是否使用“Use Default If Unwired”

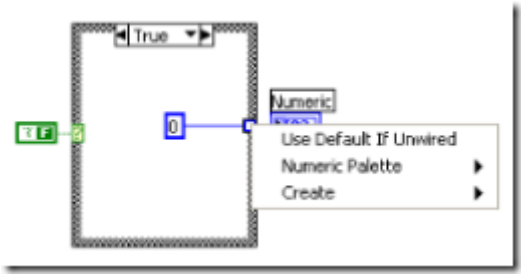


图5：选择 Use Default If Unwired

事件结构

阮奇桢

Event Structure 也是一种选择结构，程序根据发生的事件决定执行哪一个页面的程序。此时，LabVIEW 的界面编写与 Visual Basic 的界面程序有些类似。

一. 按照产出源来区分事件的种类

按照事件的产生源来区分，LabVIEW 有以下几种事件：

图1：配置事件

1. 应用程序事件（<Application>），这类事件主要反映整个应用程序状态的变化，例如：程序是否关闭，是否超时等。

2. VI 事件（<This VI>），这类事件反映当前 VI 状态的改变。例如：当前 VI 是否被关闭，是否选择了菜单中的某一项等等。

3. 动态事件（Dynamic），用于处理用户自己定义的或在程序中临时生成的事件。

4. 区域事件（Pane）和分割线事件（Splitter）是 LabVIEW 8中新添加的特性。LabVIEW 8中，用户可以把一个 VI 的前面板分割成几份，这两类事件用来处理用户对某个区域或区域分割线的状态的改变。

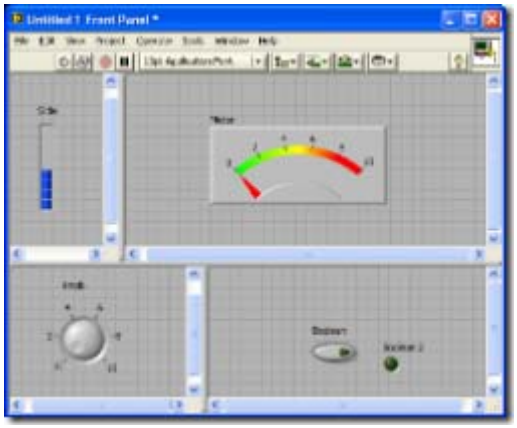


图2：面板上划分区域

5． 控件事件（Control）是最常用的一种事件，用于处理某个控件状态的改变。例如，控件值的改变，或者鼠标键盘的操作。

打开上述的“edit events”框，只要选定了某一个事件产生源，其相应的所有事件均排列在右侧 events 框中。有时候，多个事件产生源会对同一个用户操作分别产生相应事件。比如在某一控件上按下鼠标，区域事件和控件事件都会发出鼠标按下（Mouse Down）事件。LabVIEW 按以下规则顺序产生不同的事件：

键盘相关的事件（Key Down, Key Up, etc.）只在当前选中（Key Focused）的控件上产生;

鼠标相关的事件（Mouse Down, etc.）按照从外向里的顺序发出。例如，区域的鼠标按下事件先于控件的鼠标按下事件发出；结构的鼠标按下事件先于先于结构内控件的鼠标按下事件发出。

值改变事件按照从内向外的顺序发出。结构（Cluser）内控件的值改变事件先于结构的值改变事件发出。

二. 按照发出时间区分事件的种类

按照事件的发出时间来区分，LabVIEW 的事件可分为通知型事件（Notify Event）和过滤型事件（Filter Event）。

通知型事件是在 LabVIEW 处理完用户操作之后发出的，比如用户利用键盘操作改变了一个字符串，LabVIEW 在改变了该控件的值之后，发出一个值改变（Value Changed）通知型事件，告诉事件结构，控件的值被改变了。如果事件结构内有处理该事件的框架，则程序转去执行该框架。

过滤型事件是在 LabVIEW 处理用户操作之前发出的，并等待相对应的事件框架执行完成之后，LabVIEW 再处理该用户操作。这类事件的名称之后都有一个问号。例如键盘按下？事件（Key Down? Event），当用户处理该事件时，控件的值还没有被改变，因此，用户可以在该事件对应的事件框架内决定是否让 LabVIEW 先处理该事件，或改变键盘按下的值之后再让 LabVIEW 继续处理该事件。

可以明显地看出，过滤型事件比相应的通知型事件要先发出。

当同一 VI 的程序框图上有多于一个的事件结构时，通知型事件是同时被发往所有的事件结构的，而过滤型事件则是按顺序、依次发往每一个事件结构的。但是，在同一 VI 上放置多个事件结构是没有必要，而且极易引起错误的。所以应该避免在同一 VI 上使用多个事件结构。

下面举例说明如何使用通知型事件。我们经常需要使用到这样的字符串控件：控件用于输入电话号码，因此只接收数字和横线，对其他按键不起反应。LabVIEW 没有直接提供此种控件，但是它们可以利用通知型事件被方便地实现出来。

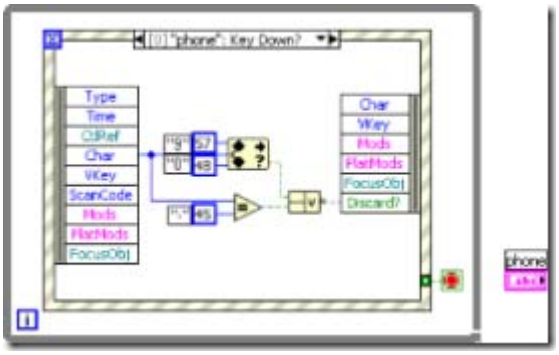


图3：利用 Key Down? 事件实现电话号码控件

这个例子可以在这里下载：Phone Number Control.vi

三. 动态事件

在初始状态下，打开事件配置（Edit Events）对话框，动态事件下的一栏是空的。因为动态事件只有注册过之后才能使用。与事件相关的操作在函数选栏的 Programming -> Dialog & User Interface -> Events 下面。

用于注册事件的节点是事件注册节点（Register For Events）。需要注册某一事件时，先为它的产生者生成一个引用节点，然后将引用节点与事件注册节点的下方区域相连，再选取所需的事件。如下例：

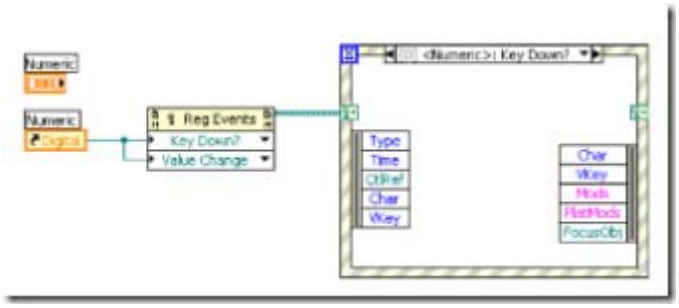


图4：注册动态事件

对于当前 VI 上的控件或区域等类型的事件，一般来说可以在事件结构中静态地被选择，所以不需要再动态注册一遍。但有时，当前 VI 的程序框图已经过于复杂，我们希望在子 VI 里去处理某些控件的事件。这时就可以把控件的引用传入子 VI，在子 VI 中动态注册所需事件。在子 VI 的事件结构中处理相应的事件。

四. 用户自定义的事件

用户自定义的事件是动态事件的一种。用户自定义的事件不基于任何一个 LabVIEW 对象，它是使用创建用户事件节点（Create User Event）生成出来的。并且，用户可以选择不同的事件数据类型。

五. ActiveX 控件的事件

ActiveX 控件的事件不能直接被 LabVIEW 的事件结构所截获。ActiveX 事件需要用注册事件回调 VI 节点（Register Event Callback）来为某一事件指定一个 VI。当事件发生时，执行被注册的 VI。我们也可以

利用注册事件回调 VI 节点为某一 LabVIEW 自身的事件注册一个回调 VI，但是出于运行效率，和程序可读性等方面的考虑，最好不要这样使用。

循环结构

LabVIEW 中的循环结构有 for 循环和 while 循环。其功能与文本语言的循环结构的功能类似类似，可以控制循环体内的代码执行多次。

一、for 循环

但是 LabVIEW 中的 for 循环的限制更多一些。

1． For 循环的迭代器只能从 0 开始，并且每次只能增加 1。

2．For 循环不能中途中断退出。C 语言里有 break 语句，但在 LabVIEW 中不要试图中间停止 for 循环。

外部数据进入循环体是通过隧道进入的，有几种方式：

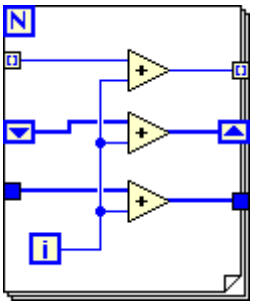


图1：For 循环结构上的隧道

图 1 所示的 For 循环结构演示了三种隧道结构，就是在 For 循环结构左右边框上用于数据输入输出的节点。这三种隧道从上至下分别是：索引隧道、移位寄存器（shift register）、一般隧道。

一般隧道，就是把数据传入传出循环结构。数据的类型和值在传入传出循环结构前后不发生变化。

索引隧道是 LabVIEW 的一种独特功能。一个循环外的数组通过索引隧道连接到循环结构上，隧道在循环内一侧会自动取出数组的元素，依顺序每次循环取出一个元素。用索引隧道传出数据，可以自动把循环内的数据组织成数组。

通过移位寄存器传入传出数据，也是数据的类型和值都不会发生变化。移位寄存器的特殊之处在于在循环结构两端的接线端是强制使用同一内存的。因此，上一次迭代执行产生的某一值，传给移位寄存器右侧的接线端，如果下一次迭代运行需要用到这个数据，从移位寄存器左侧的接线端引出就可以了。

C 语言程序员初学 LabVIEW，在使用循环结构时，常常为创建一个中间变量烦恼。为循环中的变量创建一个 Local Variable 不是好的方法。我们应当时刻记得 LabVIEW 与一般文本语言不同，LabVIEW 的数据不是保存在显示的变量里，而是在连线上流动的。LabVIEW 是通过移位寄存器把数据从一次循环传递到下一次的。

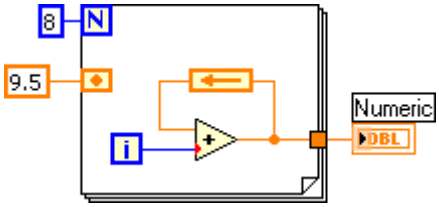


图2：反馈节点

如果单纯是为了让下一次迭代使用上次迭代的数据，也可以使用反馈节点，如图2所示。

移位寄存器左侧的接线端可以不只有一个，用鼠标可以把左侧的接线端拉出多个来，如图3所示。下面的接线端可以记录上两次、三次……的数据。

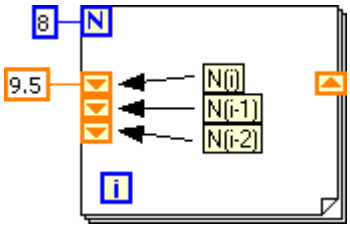


图3：多接线端移位寄存器

使用数组的隧道有一些需要注意的事项，参考：LabVIEW 代码中常见的错误。

从 LabVIEW 8.5 开始，for 循环增加了结束判断条件。for 循环也可以像 while 循环那样随时结束运行。

二、While 循环

LabVIEW 的 While 循环相当于文本语言中的 do... while... 循环。有些语言还有 while... do... 循环，LabVIEW 没有这样的循环。LabVIEW 的 while 循环至少要运行一次。

for 循环中可以用的数据传递方式，几种隧道也都可以在 while 循环中使用。所以在很多情况下，while 循环可以替代 for 循环。

While 循环比 for 循环(LV 8.5 之前)灵活的地方是可以进入循环后在决定何时循环结束。比如，希望当某一变量大于一个值时停止循环，这种情况下不能预知循环次数，所以一定要使用 while 循环。

while 循环也有不利的方面：

首先，for 循环更利于阅读。读者一眼就可以看出程序会内运行多少次。

其次，while 循环也可以使用带索引的隧道来构造数组，但是它的效率低于 for 循环。

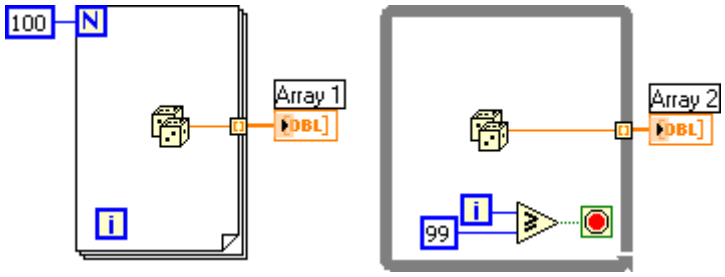


图4：使用循环构造数组

如图4，用两种循环所产生的数组大小是相同的。但是如果使用的是 for 循环，LabVIEW 在循环运行之前，就已经知道数组的大小是100，因此 LabVIEW 可以一次为 Array1 分配一个大小为 100 的内存空间。但是对于 while 循环，由于循环次数不能在循环运行前确定，LabVIEW 无法一次就为 Array2 分配合适的内存空间。LabVIEW 会在 while 循环的过程中不断调整 Array2 内存空间的大小，因此效率较低。

所以，在可以确定次数的情形下，最好使用 for 循环。

三、移位寄存器

移位寄存器除了在迭代间传递局部数据，还有其他一些功能。

首先，移位寄存器可以用于程序的内存优化。

由于移位寄存器的左右接线段使用的是同一块缓存，可以利用这一特性，显示的告诉 LabVIEW 重用某些数据的内存，并且不对数据做额外的拷贝。详细说明可以参考：LabVIEW 程序的内存优化。

移位寄存器还经常被当作全局变量来使用，比如 LabVIEW 程序中常见的功能全局变量。

定时结构

定时结构是从 LabVIEW 7.1 开始出现的。一眼就能看出来，它在外观与其它结构的风格完全不同。倒是和 LabVIEW 7 力推的 Express VI，风格一致。打开定时结构的函数面板（图1），最上面两个分别是定时循环，和定时顺序结构。下面的是与控制时间结构相关的的一些 VI。

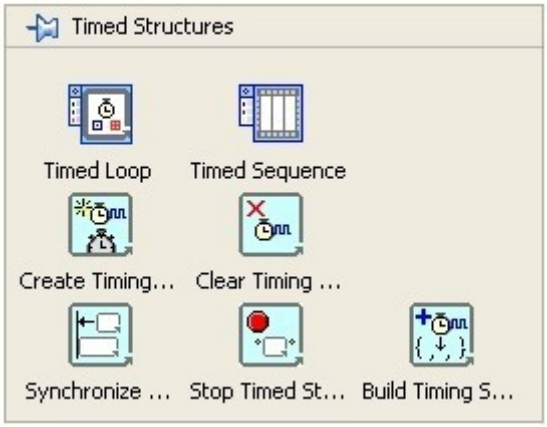


图1：时间结构的函数面板

定时结构，顾名思义，与时间控制有关。LabVIEW 中原本有一些用于延时或定时的函数，比如 Wait, Delay Time 等，他们都位于 Time&Dialog 面板中。利用这些函数，基本可以实现与使用时间结构相同的功能。定时结构的最大改进在于，它可以选择使用哪个时间源（硬件）来定时。尤其是当你的 LabVIEW 程序运行在 RT、FPGA 等设备上时，这一点就特别有用了。使用定时结构指定使用硬件设备上，而不是 PC 机上的时钟来定时，可以使使运行时序更精准。

即便同样都是在普通 PC 上使用，定时间结构的定时效果也要比 Wait 等函数精确的多。我曾经参与过的一个测试程序，开始使用 Wait 函数定时，运行一小时后，时间误差有几分钟。改用定时结构后，误差缩短到了几秒钟。

缓存重用结构

一、缓存重用

在《LabVIEW 程序的内存优化》一文中有一个利用移位寄存器来降低 VI 内存的例子。下面这个 VI 大约会占用了2.7M 的内存空间

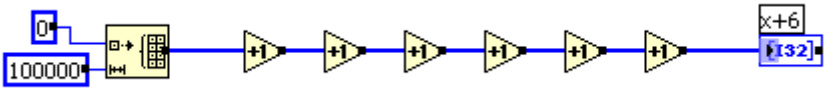


图1： 对数组进行数值运算的顺序执行程序

给它加上一个移位寄存器，如下图所示，内存占用就降低到只有不到400k 了。

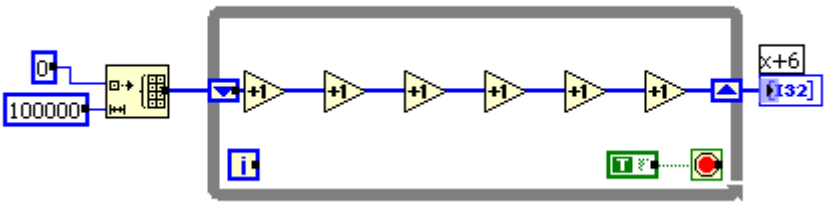


图4： 利用移位寄存器实现缓存重用

这其实是利用了移位寄存器两端接线端指向的是同一块内存这一特性，主动的告诉 LabVIEW 这段代码上的每个加法节点的输入输出数据可以使用同一块内存。避免的 LabVIEW 分配不必要的数据缓存。

但是代码还是不够完美，本来不需要循环，却非得摆上一个只执行一次的循环结构。感觉上总是有些别扭。

这个问题终于在 LabVIEW 8.5 中被解决了。LabVIEW 8.5 中多出了一个结构——缓存重用结构，专门用于告诉 LabVIEW 在某段代码上为输入输出数据做缓存重用。上面这个程序用新的缓存重用结构来写就是这样的：

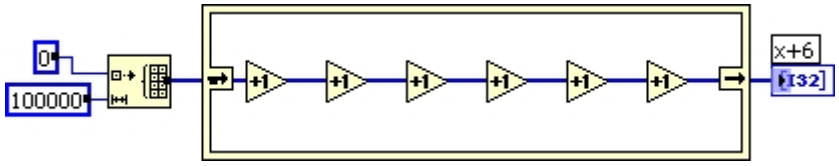


图3：利用缓存重用结构实现缓存重用

二、使用缓存重用结构

缓存重用结构与其它结构不在同一个函数选板上。这是缓存重用结构不是一个功能性、或改变程序流程的结构。它的使用不会改变代码的功能，仅仅会改变代码的效率。

要使用缓存重用结构，需要打开函数选板的 Programming->Application Control->Memory Control。第一个选项就是他了。

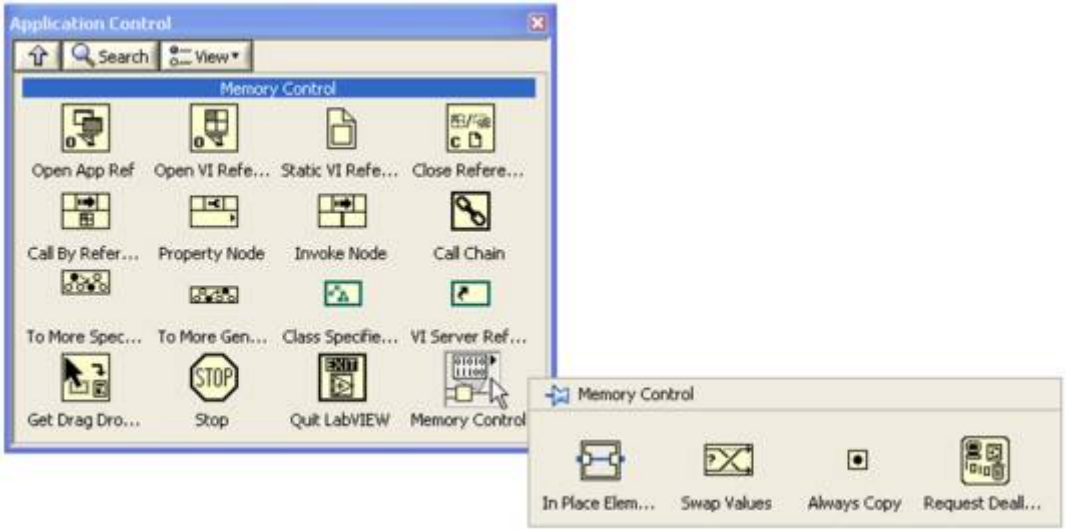


图4：缓存重用结构在函数选板上的位置

缓存重用结构为了方便使用，并不是简单的作为循环加移位寄存器的替换，它还有一些可选的边框节点，帮助编程者处理不同的数据类型。

刚刚被拖到程序框图上的是一个光滑的黄色方框，要使用它的缓存重用功能还要为打算从用的内存，根据它的数据类型选择相应的边框节点。在黄色的边框上点击鼠标右键，弹出菜单的最后几项就是可供选择的边框节点类型。如图5所示。

每种边框节点都是成对出现的，一个在输入端，另一个在输出端。

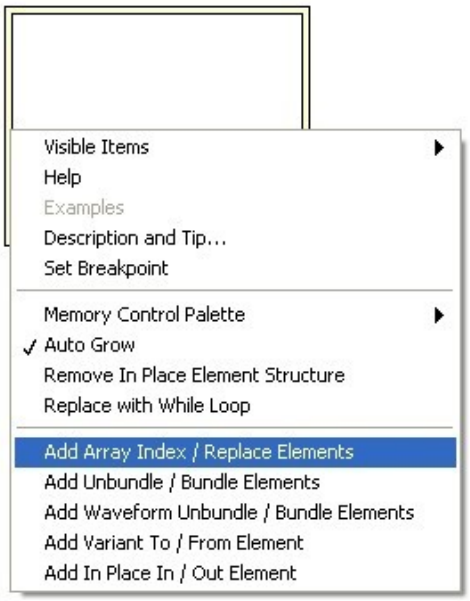


图5：添加边框节点

三、边框节点

1. 数组元素索引和替换节点

这对节点用于改变数组中某个元素的值。输入的数组数据连到缓存重用结构左面的数据索引节点上，结构内得到的数据，就是需要处理的元素的数值。

LabVIEW 中的泛型容器

Google 网站里有个 Google 实验室，有不少 Google 的产品最初就是放在这个实验室里的。现在 NI 也有 NI 实验室了。NI 实验室公布出来的项目一般是 NI 工程师利用额外时间做的一些调查研究。这些项目不是公司的正式产品，但是它们的设计很有创新或者是比较有应用潜力。与其让这些项目被埋没了，不如先看看用户对这些项目的反应，如果相当一部分用户觉得某个项目非常有帮助，或许它就值得我们为其增加投资，把它作成正式产品了。

我这里给大家介绍其中的一个项目：“LabVIEW Generic Container Map”。因为这个项目是我设计的，所以对它了解比较多一些。当时，我们打算提出这个项目的时候，主要有两个目的：第一是帮助用户编写有复杂数据结构的应用程序;第二是推进 LabVIEW 向通用编程语言方向做改进。

C++ 的程序员基本都很喜欢 STL 这个模板库。程序中常会使用数组、队列、字符串等等数据类型和结构，如果自己设计实现这些数据结构和相关的操作，是相当耗费精力的。好在 STL 实现了这些数据结构，和它们常用的操作方法。借用 STL 提供的功能，编程时很多细节方面不需要再去考虑了，这就让工作简化了许多。尝到 STL 甜头的程序员，在编写程序的时候，已经很难离开 STL 了。

STL 中非常重要的一个部分就是容器。容器用于存放数据，程序通过调用容器的结构函数保存数据到容器或者访问容器中的数据。容器也分为不同的类型，如链表、队列等。它们在数据的组织方式上，或存取方式上有所分别，以适用不同的需求。STL 中的容器和方法都是泛型的或者说是数据类型无关的，就是说这些容器可以保存和操作任何类型的数据。

其它一些常用的编程语言，如 Java、C# 也都有类似的泛型容器以方便程序员使用。

LabVIEW 的主要方针是简化工程师们编写程序的难度，以前用 LabVIEW 编写的程序大多是工业领域流程控制类型的。这种类型的程序用不到太复杂的数据结构和算法，因此，LabVIEW 中对我们在计算机课程中学到的那些经典数据结构以及算法的支持并不多。

但是在我自己用 LabVIEW 多了之后，用它比用 C++ 要顺手，任何类型的程序都喜欢使用 LabVIEW 来编写，包括一些通常用途的程序。这时候，LabVIEW 缺乏对基本数据类型支持的缺点就格外突出了。于是我和周围几个同事就想到应该在这些方面对 LabVIEW 做一些补充，做一些比较规范的泛型容器和算法，一方面方便自己，也许还可以提供给别的用户。

由于这不是正式项目，我们能投入的资源很有限，不可能一开始就做得很全面。作为开始，我们选择了 Map 容器和它最常用的几个方法。首先选择 Map 一是因为它比较常用，二是其它容器中，有些在概念上和 LabVIEW 中已有的一些函数比较接近，如果选则他们，可能会引起用户的误解。

LabVIEW 中的 Array 操作与 STL 中的 vector 是非常相似的，功能齐全，不需要考虑底层操作如内存管理等。STL 中的 deque, queue, stack 等，与 LabVIEW 中的“队列”(Queue)操作比较类似。但是 LabVIEW 中的队列存在的目的不是为了作容器，而是用于在多线程程序中通讯。在“生产者/消费者”程序模式中，经常使用队列在不同的线程中传递数据或消息。因为 LabVIEW 中队列操作主要用于不同线程间的通讯，因此它的函数并没有采用 LabVIEW 的主要传参方式-传数据，而是采用了传引用的方式。

我们实现的这个 Map(这个按字面翻译比较别扭，中文可能翻译成“字典”还比较合理)泛型容器功能与

C++ STL 中的 Map 是类似的，它主要用于程序经常需要按某一关键字查询数据的情况。

Map 已经包括了编写查询程序时常用的操作，比如把数据放到容器中、查找一个数据、删除、清空容器等。

我们的 LabVIEW Generic Container Map 内部的数据是按照平衡二叉树的方式组织存储的，它的查询复杂度比一般线性数据结构的要低。这样，在数据量很大的情况下，使用 Map 的程序效率明显高于使用数组的程序。

Map 采用的是符合 LabVIEW 风格的传数据方式，把整个 Map 中的数据在不同函数间传递。

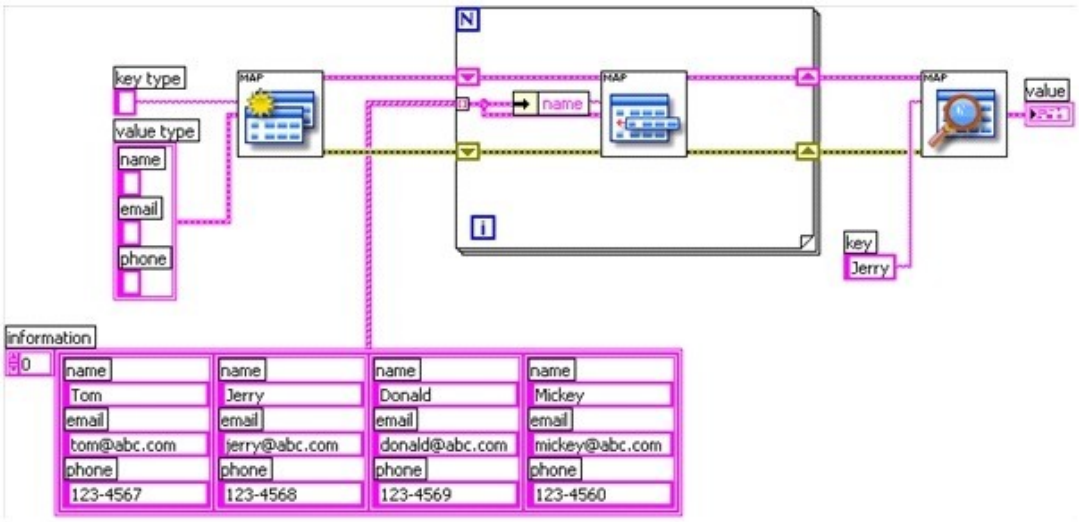
C++ 是支持泛型编程的。简单地说，泛型编程可以这样理解，就是程序员可以实现一个方法，这个方法能够应用在任何合法的数据类型上。比如，前面提到的 STL，它有个“比较”方法，你可以用它来比较整数，也可以比较字符串，或者使用户定义的类的实例等等。

支持泛型编程，程序员就可一抽象出与数据类型无关的算法，从而使代码具有更好的可重用性。

目前，用户还不能在 LabVIEW 上实现泛型编程。使用 Polymorphic VI 可以使一个方法支持某几种特定的数据类型，但不是任何数据类型。真正能做到数据无关的函数，比如说“equal”函数，都是 LabVIEW 自带的，用户无法写出这样一个函数或 VI。

我是非常想推动 LabVIEW 有朝一日也实现泛型编程的。这个 Map 容器也是在 LabVIEW 泛型编程方面做的一次探索。这个 Map 容器不是 LabVIEW 自带的东西，它是完全使用 G 代码实现的。但是它同时也是一个泛型容器，支持对任何数据类型的操作。

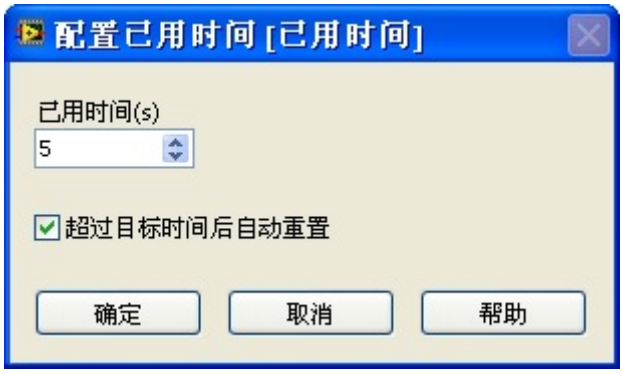
下图是使用 Map 编程的一个例子：



循环运行某个时间后退出

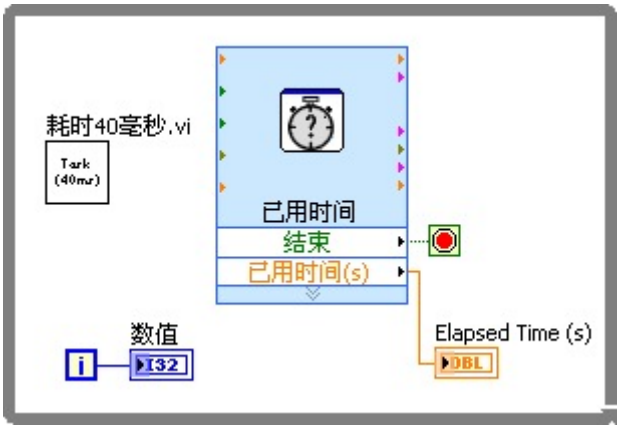
有时候需要一个循环不是在迭代多少次之后停下，而是在运行多少时间之后停下。一般，直接的解决方案就是利用“时间计数器”函数，进入循环前，记下当前时间，然后每次循环迭代都查看一下当前时间，若超过所需时间，则退出循环。这个功能也可以使用“已用时间”Express VI 来完成。

把“已用时间”Express VI 拖到程序框图上，会出现它的配置对话框。我们需要这个 Express VI 计时5秒，每次计时完成自动重置。



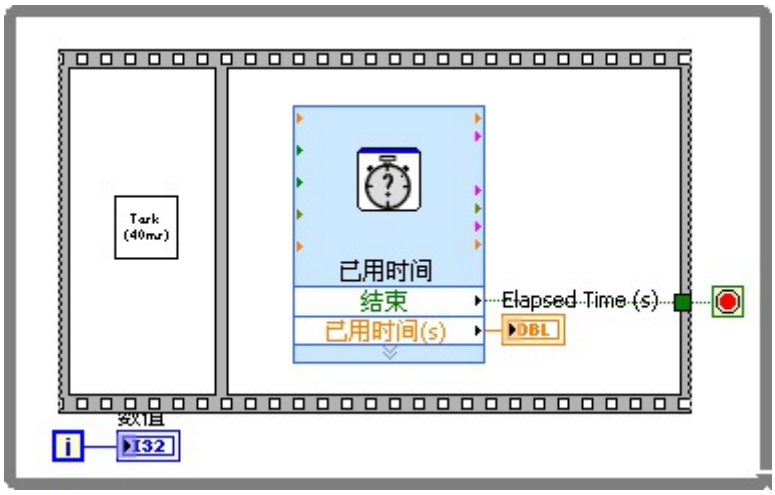
在循环里，需要用到的是“已用时间”的“结束”输出，一旦运行时间超过5秒，这个值就会被设为真。这时循环停止。

但是下图程序有个问题，就是“耗时40毫秒”这个完成功能的 VI 和“已用时间”VI 之间没有连线，它们是同时运行的，而“已用时间”运行时间几乎可以忽略，一调用它，它就立刻返回“结束”值。这时，即便“结束”值为“真”，循环也要再等大约40毫秒，待功能完成功能不部分运行结束，循环才停止。

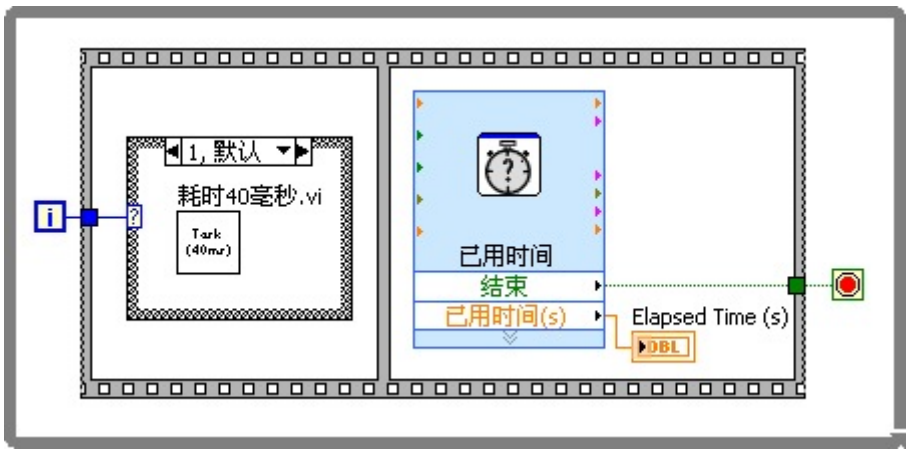


强制程序先工作，再计时间是否可以解决这个问题呢，如下图中的程序。这样也不行，因为“已用时间”是在重置后，第一次调用它的时候开始计时的。下图这个程序，循环第一次迭代，并没有马上就开始计时，

而是要等到功能 VI 完成后，已经耗用的一段时间，才开始计时。这个即时已经不精确了。



所以要完全解决这个问题，只能麻烦一点。程序中多加一个条件语句。在循环一开始，就立刻计时;而在后续的迭代中，每次功能完成再检查当前时间。或者，把“已用时间”放在前面，每次判断是否结束。如果是，则不执行“耗时40毫秒”这个 VI。



循环结构的反馈节点

如果单纯是为了让下一次迭代使用上次迭代的数据，可以使用反馈节点，如下图所示。

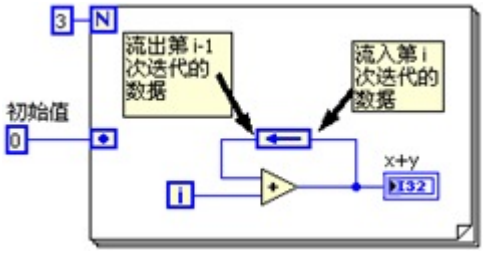


图1：反馈节点

需要使用反馈节点的时候，可以通过移位寄存器的右键菜单，把一个移位寄存器改造成反馈节点。在给循环结构内的节点的连线端连线时，如果数据流出现一个环，LabVIEW 会自动创建出一个反馈节点插在这个环中。例如图2中的“+1”函数，输入从它的输入端流入，被其加工过后再从它的输出端流出。现在试图连线，把流出的数据再引回到“+1”函数的输入端，这样就形成了数据流的环，LabVIEW 会自动在这个环上查如一个反馈节点。

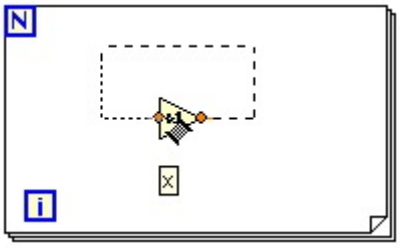


图2：连接输入输出接线端

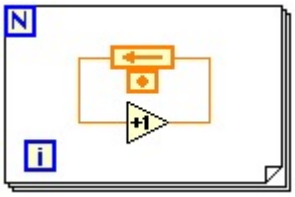


图3：通过直接连线创建出的反馈节点

反馈节点与移位寄存器在本质上是相同的，它只是改变了数据线的连线方式。把原本在循环结构两侧的连线端移到循环中间来了。

经常绘制电路原理图或者控制信号流图的用户可能会比较喜欢反馈节点。因为它这比较符合绘制这些图时的习惯。直接把数据线画成一个环就可以表示反馈了。

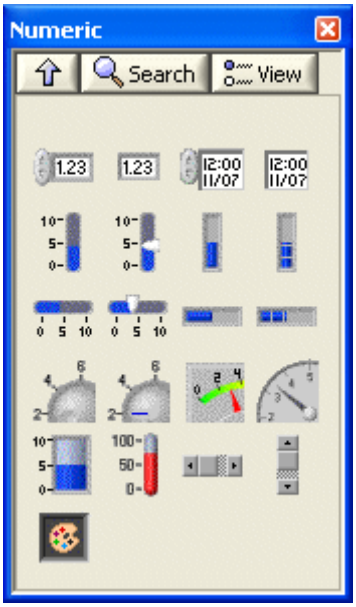
此外，反馈节点在某些情况可以缩短程序框图上的数据线，简化程序框图。但是它会导致某些连线上的数据逆向流动，从左向右流动。如果逆向数据线过长，则不如使用移位寄存器。

LabVIEW 中的数字型数据 1 - 控件和常量

一、数值型控件和常量

1. 控件

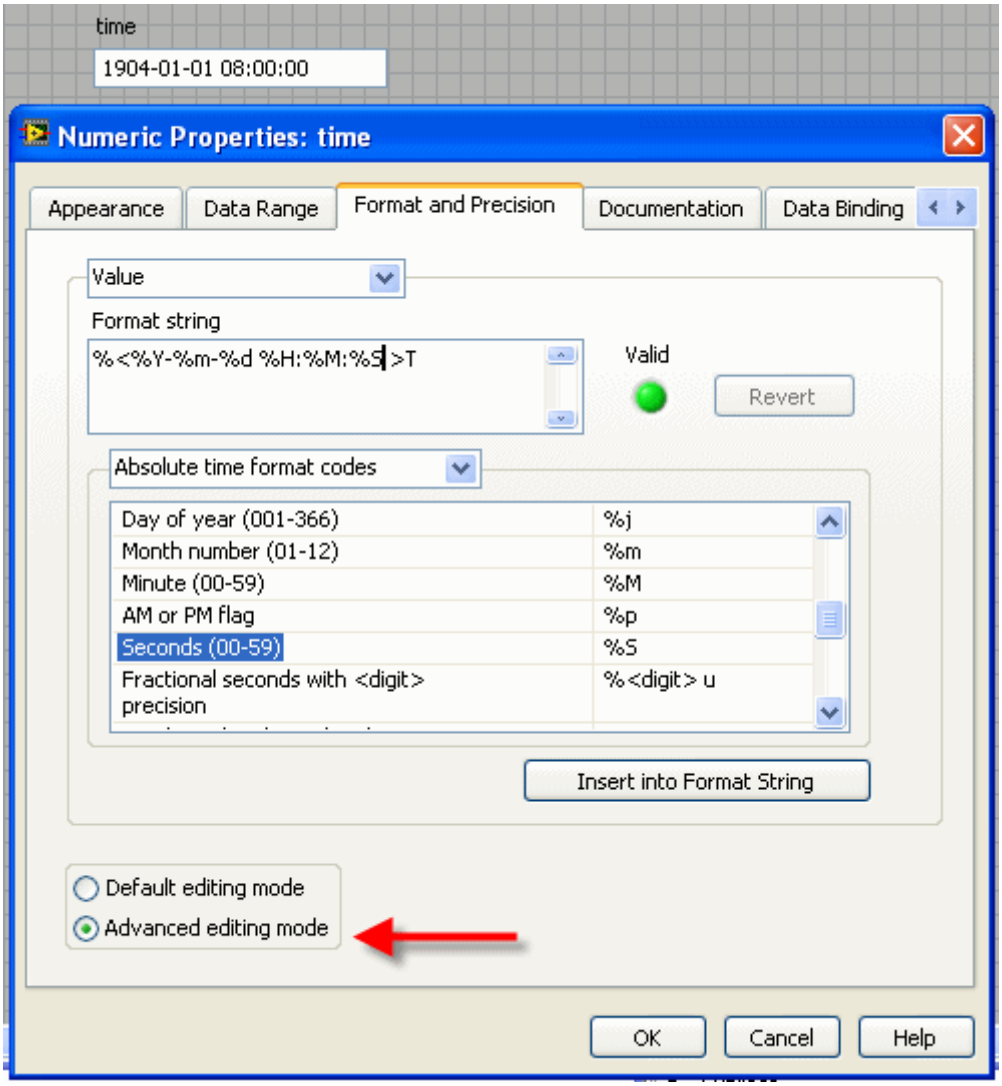
在 LabVIEW 的控件栏中有一栏是数值控件。



这一栏内的控件虽然在前面板上的外观各不相同，但是在框图中的端点都是数值类型的。我们在使用这些数值控件时可以选择适合的界面所需的旋钮、仪表盘等。还可以在数值型控件的属性对话框里设置它的数值类型、数值范围、格式和精度等，显示方法等等。

我们以最普通的数值控件为例，解释一下如何配置它的显示方式。假如，我们的界面是 Windows 风格的，那么界面上所有的控件都应使用系统风格的控件，包括数值型控件。如果这个控件用于表示时间，我们就需要对这个控件的显示方法进行高级的配置。

打开这个控件配置面板的格式与精度页，选择“Advanced editing mode”，就可以自己为控件设置显示方式了。



2. 常量

如果是常量，在设置数值类型时通常会发现“Adapt to Source”(按照输入调整)项是被选中的，作为控件时这一项不能被选中。此时如果在常量中输入一个正数，比如“34”，常量的类型会自动变为 I32整型(蓝色)，而输入“34.3”， 常量的类型会自动变为 DBL 实数型。如果一定要输入实数型34，可以输入34.0。

3. 不同表示法(Representation)的选择

数值类型包括各种长度的整型、实数型和虚数型，其中 I64和 U64类型是 LabVIEW 8.0新增加的。选用短整型数值比选用长整型数值类型节约内存。在大的数值数组中应尽量使用短类型数值以节约内存。对于单个数值，它可以节约的内存十分有限，但是使用长整型数值可以避免数值越界引起的错误，所以还是应该使用长整型数值。

你可以自己做个试验：新建一个 VI，在 VI 上放置两个值为 300 的 I16 常量，然后相乘，看看他们的积是多少。这种错误如果隐藏在一个大工程内，调试起来也是颇为费劲的。

如何学习 LabVIEW

根据我自己的观察，学习 LabVIEW 一般有以下三种方式：系统型学习方法、探索型学习方法和目标驱动型学习方法。这三种方法之间并不矛盾，可以在不同的时段使用不同的方法。每个人可以根据自己的个性特点和所处环境选择一个适合自己的学习方案。

系统型学习方案是传统的学习方法，学生学习多是按此方法。它是指按照别人制定好的学习方案一步一步学习掌握一门知识。学习效果如何，主要取决于教师和教材的水平。若选此方案学习 LabVIEW，最高效的方法莫过于参加 NI 公司的 LabVIEW 培训课程。基本上，完全没接触过 LabVIEW 的学员可以在一星期的时间内达到编写简单程序的程度。此外，现在很多大学都开有 LabVIEW 课程，方便了在校生学习。

自学也可以采用此方案。找一本教程类的书籍，按照书中指导一步一步学习。教程类的书籍应当侧重于解释 LabVIEW 的编程思想以及原理;有些书仅偏重于罗列 LabVIEW 中每个函数或 VI 的功能，则不适合用于此种学习方案。

探索型学习方法适合喜好自己钻研的人。同样一个技巧，如果是自己发现的，比从他人那里的来会更有成就感。任何一个教程都不可能覆盖到 LabVIEW 的全部功能，有心得学员不妨自己打开书中未曾介绍到的那些菜单或者函数选板，尝试一下它们都是做什么用的。在真正动手摆弄每个新东西之前，打开 LabVIEW 的即时帮助窗口，阅读一下相关说明可以大大加快学习过程。

比如，打开“应用程序控制”函数选板，发现这里有一项“选板编辑”。好像没有任何一本书里介绍过这个功能嘛，这是干啥用的呢?如果没任何提示，也是无从下手去尝试的。打开 LabVIEW 的及时帮助，可以看到它对这个功能的简单介绍。进入“详细帮助信息”，会得到更全面的说明。再自己动手实践一下，就基本可以掌握此功能了。

阅读他人代码也是一个很好的学习方法。自己的探索总是有思维局限性的，他人解决问题的方法可以大大拓宽自己思路。我介绍过的编程经验中，很大一部分都不是我自己凭空想出来的，而是借鉴与别人的 LabVIEW 代码。

目标驱动型学习方法是公司员工中最常见的学习方式了。工作后，如果不是个人有兴趣，多数人不会浪费时间去学习工作中用不到的知识。等老板布置了具体项目或者工作任务后再学习相关知识，效率更高。学也只要够解决眼前问题就行了。针对这种情况，请教身边牛人或者公司前辈是最好的学习方法。如果周

围的人不能解决问题，到论坛上发贴，寻求更广泛的帮助。

推荐一个论坛。首先是 NI 的官方论坛，这里会有 NI 的技术支持和研发工程师来回答问题。如果英文够好，最好是到它的英文版面去提问，英文讨论区人气更旺，容易找到答案。LAVA 是官方之外最大的 LabVIEW 社区，也是寻求帮助的好地方。如果平时用 Windows Live Messenger，可以加入 http://labview.groups.live.com/，这是个 msn 讨论 LabVIEW 的群。在它上面讨论问题最大的好处是可以及时得到回应。

我见过几个工程师在项目中遇到了难题，于是来报名参加 LabVIEW 的培训课程，以为上完课可以解决自己的问题。但实际上完全误解了培训课程的意义，培训课程是为了帮助那些想要系统学习 LabVIEW 知识的人，而不专注于任何一个具体问题。

LabVIEW 中的数字型数据 2 - 运算

二、运算

1. 常用函数

与数值数据相关的运算处理节点大都在函数栏的 Programming -> Numeric 项里，如图1 所示。

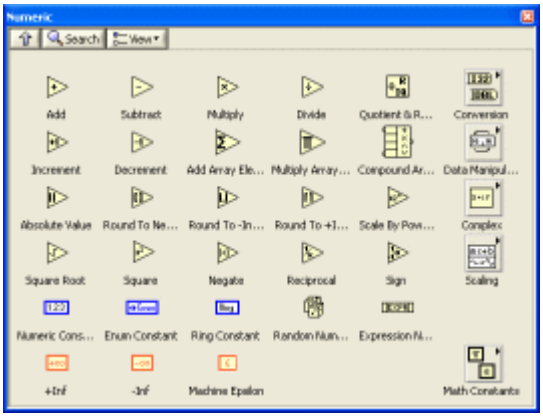


图1：Numeric Function Palette

从这些函数节点的图标一眼就可以看出它们的用处了。例如，加、减、四舍五入、求倒数等。更全面的数学运算函数在 Mathematic 函数栏。Mathematic 函数栏内的很多运算不仅是针对单个数值的，还可用于数组运算。

这里每一个公式的用途都可以在 LabVIWE 帮助文档上找到，我就不重复了。我们在这里着重讨论一下，在众多类似的运算方法中，如何选择一个适合你的程序的方案。

对于简单一次性加减乘除，自然使用基本的函数节点就够用了。但是，如果是复杂的数值运算，则需要大量函数节点。节点之间的连线可能会有转角甚至相互交叉，显得比较杂乱，不利于程序阅读和维护。这时我们可以使用其它运算节点。

2. 表达式节点

对于只有一个输入和一个输出的运算，我们可以使用表达式节点(Expression Node)。

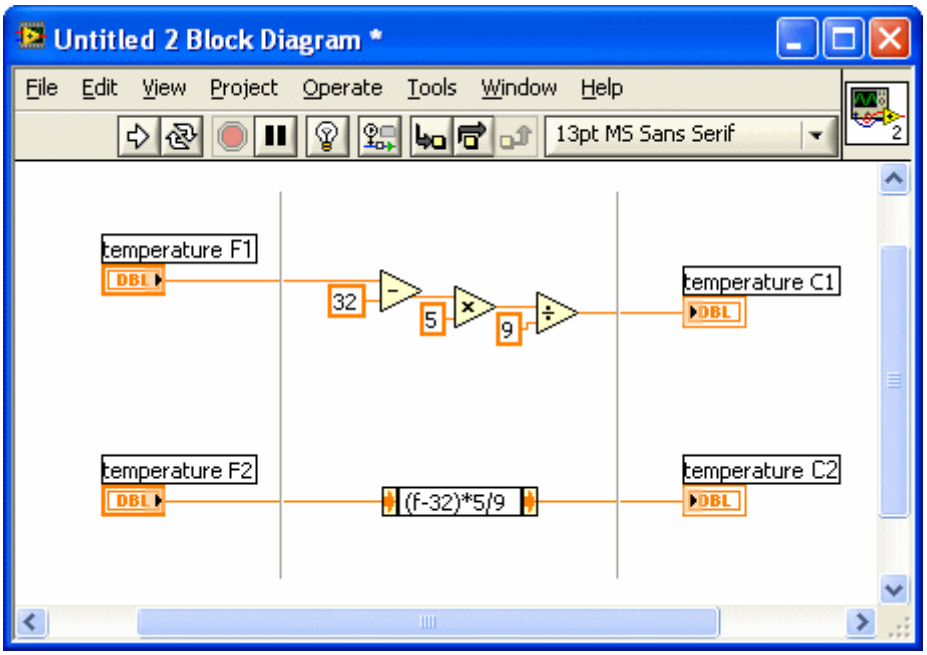


图2：Expression Node 的应用实例

图2 所示的例子中，完成把华氏温度转换为摄氏温度的计算。F1 到 C1 的转换是通过基本运算节点完成的。尽管运算并不复杂，但是阅读程序的人仍然无法立即就意识到这个运算与书中给出的公式相对比是否正确，还需要仔细地一步一步判断。这是图形化语言在表达纯数学计算时不利的一面，文字表达方式此时会更为直观易懂。表达式节点是使用文字来描述运算的。F2 到 C2 的转换就是使用表达式节点来完成的，用户可以直观地读出该节点所使用的公式。

与使用基本运算节相比较，表达式节点另一个优点是节省了框图上的空间。

在表达式节点中只允许有一个字符串，代表输入参数，例如本例中，参数用 f 表示。LabVIEW 在线帮助里列出有表达式节点所支持的运算符、函数和表达式规则。

3. Formula Express VI

如果运算有多个输入，可以使用 Formula Express VI。该 VI 在函数栏 Mathematic -> Scripts & Formulas 下。图3 是这个 Express VI 的配置面板，它看起来就像是一台高档计算器，基本不需要学习就可以使用了。

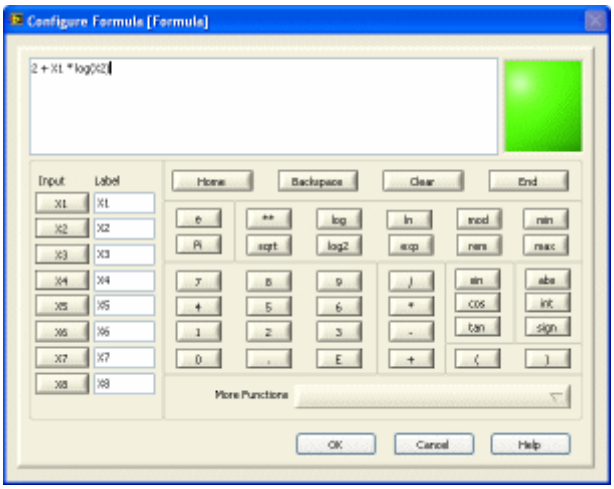


图3：Formula Express VI 的配置面板

Formula Express VI 的缺点是：他的表达式是隐藏起来的。用户需要查看，还得先调出配置面板才行。

4. 公式节点

对于更加复杂的计算，尤其是当输入变量超过一个的时候，应该使用公式节点(Formula Node)。公式节点中的表达式语法与 C 语言类似。可以把它看作是更为复杂的支持多输入输出的表达式节点。它的优点也与表达式节点相同：

在实现算法时，人们往往更习惯于文本表达方式，所以使用公式节点的可读性和可维护性更强。

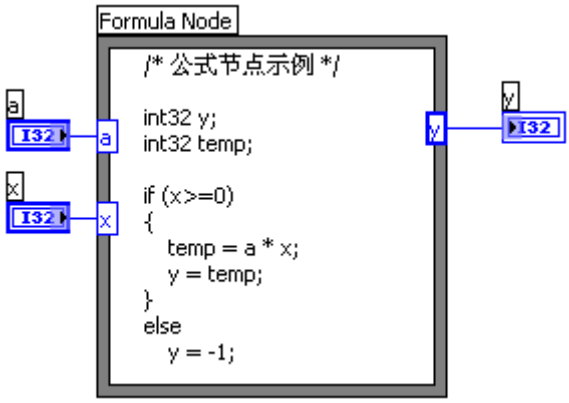


图4：Formula Node

5. MathScript，MATLAB Script 和 Xmath Script 节点

这三个脚本节点比较类似，都应用于处理更为复杂的数学运算，比如大型矩阵运算等。脚本语法使用 MATLAB 的语法或与 MATLAB 极为类似的语法。

如果是编写新程序，可以优先考虑 MathScript，因为后两种节点还需调用外部程序来解释节点中的脚本。

使用 MATLAB Script 节点 需要机器上安装有 MATLAB。MATLAB 由 M athWorks 公司开发，是数学计算工具方面最常用的软件。使用 Xmath Script 节点，需要安装 NI MATRIXx。MATRIXx 是 NI 公司进行数学运算，仿真等的产品。功能与 MATLAB 类似。

三、数值的单位

1. 数值控件上的单位

数值型控件和常量是可以带单位的。在数值型控件的快捷菜单上选择“Visible Items -> Unit Label”，就可输入数值的单位。如果你对某个单位的正确拼写没有把握，可以先任意输入一个字符，然后用鼠标右键点击单位标签，选择“Build Unit String…”。这时，LabVIEW 会弹出一个对话框，LabVIEW 所支持的单位都在这里分类排出。

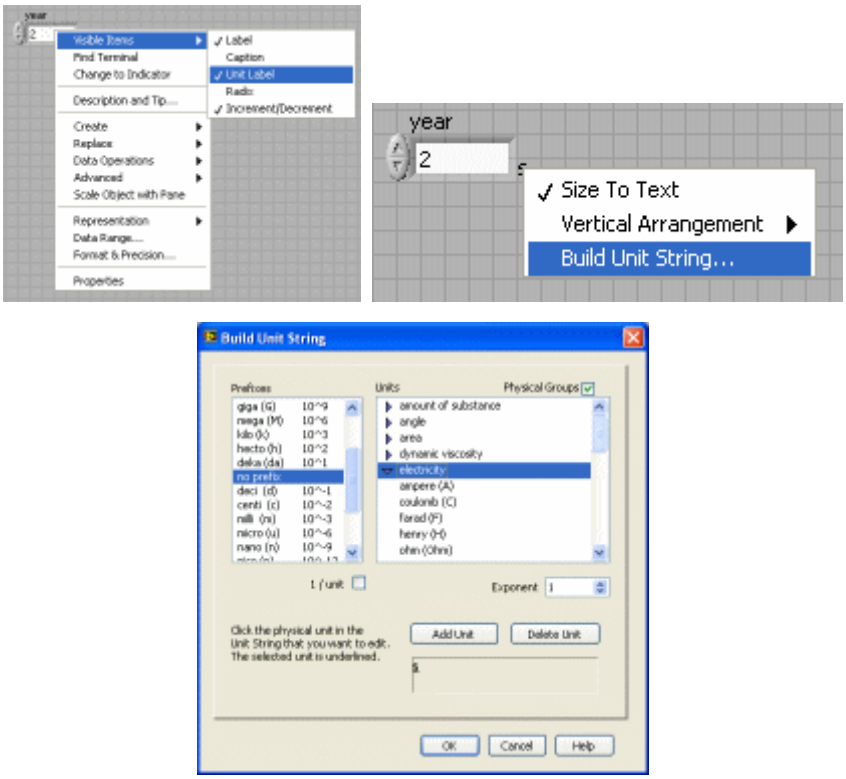


图1~3：使用数字控件的单位

例如要计算2年有多少天，可以有如下的程序：

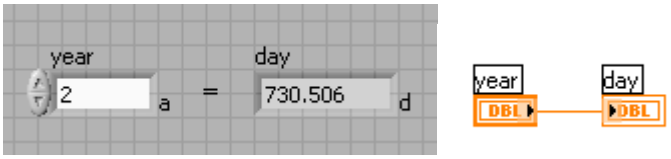


图4，5：同类型单位的空间可以由数据传递

2. 单位使数据类型检查更严格

把一个 I32 型的数据赋值给 string 型的控件肯定是一种错误行为，程序员总是希望编译器在编译时就把这种错误报告出来。虽然现在大多数编程语言都可以在编译时报告此类错误，但 LabVIEW 数值类型的单位可以让这种检查更严格：实数与字符串之间不可以互相赋值；同样是实数型的俩个数据，一个表示时间，一个表示长度，他们之间也不应当相互赋值。

在编写 LabVIEW 程序的时候，应当尽量使用带单位的数值控件。因为，如果你给一个数据设置了单位，LabVIEW 就会自动帮助你进行单位的一致性检查。比如图6 所示，当你试图把表示时间的数据和表示长度的数据相加时，LabVIEW 会禁止你连线。 着帮助你防止了编程时出现的不一致性错误。

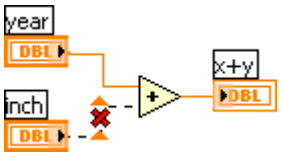


图6：不同类型的数据不能进行计算

但是，这种严格的一致性检查也可能会带来麻烦。例如，我们编写了一个子 VI，用于计算两个时间单位的和。下次当我们需要一个计算长度单位的和的子 VI 时，却不能够直接使用已有的计算时间单位的子 VI，因为它们的单位是不同的。为了解决这个问题，LabVIEW 提供了单位统配符。

在编写需要用于不同单位的子 VI 时，可以使用单位通配符。单位的通配符用 $n 表示，其中 n 是 1 到 9 之间任意一个数字。例如我们以上提到的加法，可以在子 VI 中使用通配符 $1，如果还需要另外一个执行其他运算的子 VI 中，其单位可以用 $2 表示。

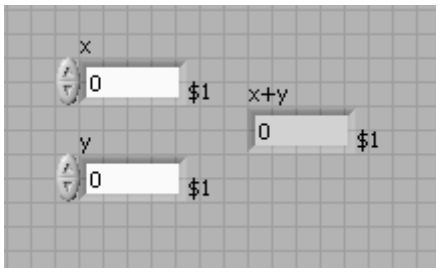


图7：使用单位通配符

3. 单位转换

使用 Numeric->Conversion->Convert Unit 节点可以把一个纯数字量转换为带有单位的数字量，或者反

过来转换。使用 Cast Unit Base 节点可以更灵活地把某一数值的单位直接转换成另一单位。需要注意的是，Convert Unit 节点的外观和表达式节点的外观一模一样，甚至快捷菜单都一样，这应该是 LabVIEW 的一个缺陷。但他们的功能完全不同，你不要试图在表达式节点中使用 build unit 菜单，它不执行单位的转换，也不指示有差错。

程序要求：输出一个字符串“LabVIEW 真好用！”，要突出强调“好用”两个字。

本程序关键是要熟悉字符串控件的属性，可以利用它的属性，选中控件中的一部分文字，并修改其字体：

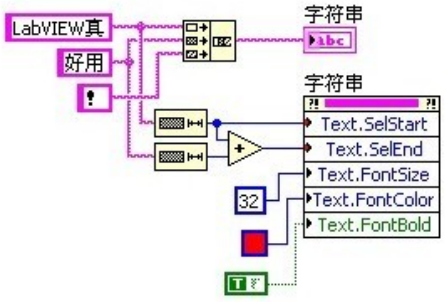


图1：程序代码

图2：程序运行结果

几种简单的测试程序流程模型

大多数测试程序主要步骤就是以下几步：采集数据、处理数据、显示数据、保存数据。这几个步骤可以顺序执行。在一次实验中，通常要多次循环这一过程，因此，这种测试程序的模型如图1所示。图1中最后一个子 VI 是用来判断实验是否结束，是否进行下一次循环用的。在这个模型中，各个程序模块是单线程顺序执行的，它的好处是程序逻辑简单，容易设计和理解。

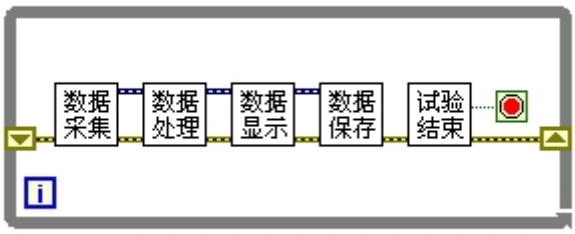


图1：顺序测试程序的模型

但是对于单线程的程序，计算机必须执行完一个任务，才能再进行下一步工作。比如，尽管数据存储是一个相对比较慢的过程，但计算机必须还是要等到它执行完，才能去做下一步的采集数据工作。

对于速度要求较高的测试程序，最好把这两样工作同时进行，以节约时间。这样，我们可以在两个循环内分别做数据采集，和其它的工作。因为数据采集的速度一般来说高于处理和存储的速度。当新数据被采集来，上次的数据可能还没处理完呢。所以可以先把每次采集到的来不及处理的数据放在缓存里。这种模型如图2所示。它实质上也就是 LabVIEW 在新建 VI 的模板中的“Producer/Consumer Design Pattern”。

这个模型的实际应用程序会更加复杂，相比第一个模型不是那么好理解和维护。

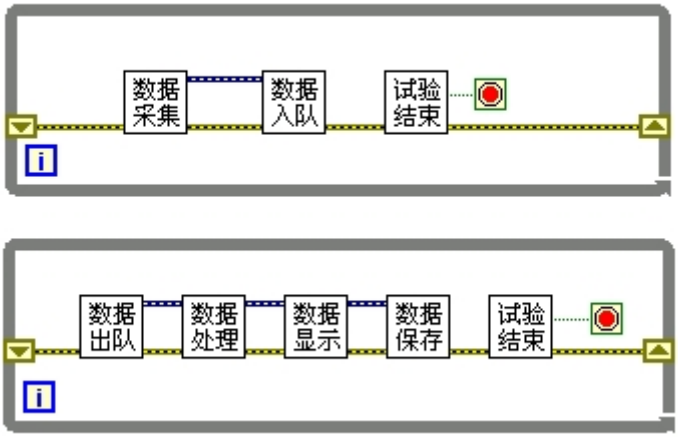


图2：数据采集和后续工作并行执行的模型

不过还有一个折衷的方案，既保证各个任务同时运行，又不至于太复杂。如图3所示，在这个模型下，所有的任务同时运行：采集新的数据、处理上一次采到的数据，显示保存上一次处理好的数据。在这个模型下，要注意第一次循环运行时处理的数据，和循环头两次运行显示存储的数据是无效的，实际循环终止

条件式也要考虑到，采集的数据再两次循环后才被保存下来。

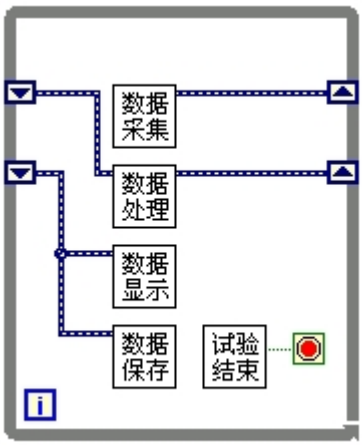


图3：并行执行每一任务的模型

用 LabVIEW 编写 Wizard 类型的应用程序 1 (LabVIEW 6.1 之前)

Wizard 向导类型的程序，指的是可以类似安装程序那样，一步一步地指导用户完成功能的应用程序。这类程序极为常见。它一边要求用户提供必要的信息，一边给用户发出指导性的意见和反馈。这样，即便是新用户也可以轻松完成任务。但是，向导类型的程序虽然方便了用户，却是增加了开发人员工作的复杂度。

在过去的七年里，我编写过各种各样的 LabVIEW 应用程序，其中大部分程序的界面都是向导类型风格的。这种风格的软件确实是用途广泛。

在这里，我把曾经用到的编写 Wizard 的各种方法作一简介。随着 LabVIEW 程序的不断改进，以前用过的有些方法今天看来或是十分笨拙、或是已经失去了意义，以后再也不会使用。我记叙下来只是作为对自己过去这一段工作的怀念吧。

一、页面为独立 VI

我在编写第一个程序时，首先想到的就是一个最直观的编程方案：为 Wizard 的每一个步骤编写一个独立的 VI。关闭当前的 VI，打开一个新的 VI，程序就自然而然从一个页面跳到另一个新页面了。完成后，就发现这并不是一个好主意。因为每个页面都是独立的，数据的交换和状态的控制都不方便。比如说，原来的面板在屏幕左边，按一下 Next 键，面板突然蹦到屏幕的右边了。

二、借助 Windows API

于是，就考虑是否可以采用框架式的结构。当时 LabVIEW 还没有 sub panel 控件，做框架只能借助 Windows API 的帮助，把一个 VI 的面板当作子窗口，插到框架 VI 的界面中去。当时公司的一些产品就是采用了这种方法。但是，我并不喜欢。它的主要缺点是只能够支持 Windows，无法跨平台使用。再者，我是一位地地道道的 LabVIEW fans，怎么能利用 Windows API 呢?我应该寻找一个纯 G 语言的解决方案。

三、变动控件位置

下面，就来介绍一个早期的纯 LabVIEW 的解决方案：通过挪动界面上控件的位置来达到界面切换的效果。具体说，就是当用户按下 Next 键，程序就把当前界面上的控件往旁边挪动一段距离，移动到用户界面的可视范围以外，用户就看不到它们了;与此同步，把那些原来在可视范围以外的、而下一页应当显示的控件，挪到可视范围内。这样，用户的感觉就是切换了一个页面。

这里有一个用这种方法编写的示例。程序的功能是把一个文件分割成几块，或者把已经分割成几块的文件再合并起来。

从范例中可以看出，这种方法十分麻烦，每一步操作都要考虑如何挪动每一个控件。如果程序太大，就难以维护了。也许现在不会再这么编写程序了，然而在当年 LabVIEW 还不支持 Tab 控件与事件处理结构(event structure)时，这个方法还相当流行的呢。

相关文章：

[1] 可互换虚拟仪器驱动程序的开发，阮奇桢. 2006.

[2]《我和 LabVIEW》的其他文章

[3] 用 LabVIEW 编写 Wizard 类型的应用程序 2 (LabVIEW 6.1 ~ 7.1)

用 LabVIEW 编写 Wizard 类型的应用程序 2 (LabVIEW 6.1 ~ 7.1)

四、Tab 控件+事件处理结构

LabVIEW 6.1 的出现才第一次大大简化了 Wizard 界面风格程序的编写。LabVIEW 6.1增加了两个非常重要的新特性，一个是 Tab 控件，一个是事件处理结构。

有了 Tab 控件，就可以把 Wizard 中每一页需要的控件分别放在 Wizard 不同的页面上，切换 Tab 的活动页面也就显示了该页面上相应的控件。

事件处理结构的应用更为广泛。有了它，编程者就不需要再添加额外的代码来监视每个控件的状态改变以及鼠标、键盘等的操作了。

这种利用 Tab 控件和事件处理结构编写的 Wizard 风格界面程序的方法现在仍然被广泛使用着。

它的功能是把一个 C 语言开发的仪器驱动程序转换为 LabVIEW 下的驱动程序。程序虽然是我编写的，但版权属于 NI 公司，所以不能把程序源代码公开给大家。

这种方法也有它的弊端。因为整个 Wizard 界面会用到的所有控件都集中在同一个 VI 上，这个主 VI 就可能特别庞大：界面可能有数十个控件，程序框图上的事件处理更为复杂，有近百个事件也不作为奇。如果需要对程序作修改，要找到相应的事件框就已经很困难了，要确定这个改动是否会影响程序的其他部分就更为困难了。

图1是我编写的一个 Tab 控件风格的向导型程序，它的主 VI 中的事件结构中，有近百个事件需要处理。对这样的程序，想找到一个相应的时间都很麻烦，处理好事件之间的关系就更困难了。

Tab 控件+事件处理结构的架构虽然大大简化了 Wizard 界面风格程序的编写，但是这样的程序很难对他的代码进行更细致的模块划分，并把模块的私用数据隐藏起来。为了使大型 Wizard 程序有更好的可读性，可维护性，还需找到一种更好的架构。

图1：使用 Tab 控件的向导型程序，事件结构中众多事件

用 LabVIEW 编写 Wizard 类型的应用程序 3 (LabVIEW 8.0)

五、SubPanel

主 VI 太过复杂，是肯定会影响它的可读性和可维护性的。所以，对向导类型程序的进一步改进的重点，就是把主 VI 进一步模块化，不但是程序代码要模块化，界面也必须模块化。代码模块化相对比较简单，多利用子 VI 就是了。但是界面的模块化，在之前的 LabVIEW 中是非常困难的，因为 LabVIEW 没办法在运行时，把不同的 VI 的界面拼在一起。是 LabVIEW 7.1 和 8.0 的一些新功能最终解决了这个问题。

对程序界面模块化，按一般的思路，第一步就是把每个页面划分成一个独立的模块。这似乎又回到了我们前文提到过的第一、二个阶段。但有所不同的是，旧版本 LabVIEW 功能不全，无法很好的管理被分为模块的页面，而新 LabVIEW 改进的对这方面的支持。

在 LabVIEW 7.1 中出现了一个新的控件 - SubPanel（子面板）。当一个 VI 运行的时候，它的 SubPanel 控件中，可以显示另一个 VI 的前面板。我们可以利用这个新的控件，我们可以使用插件框架式程序架构来编写向导型的程序。图1是这种插件框架式程序结构的示意图。

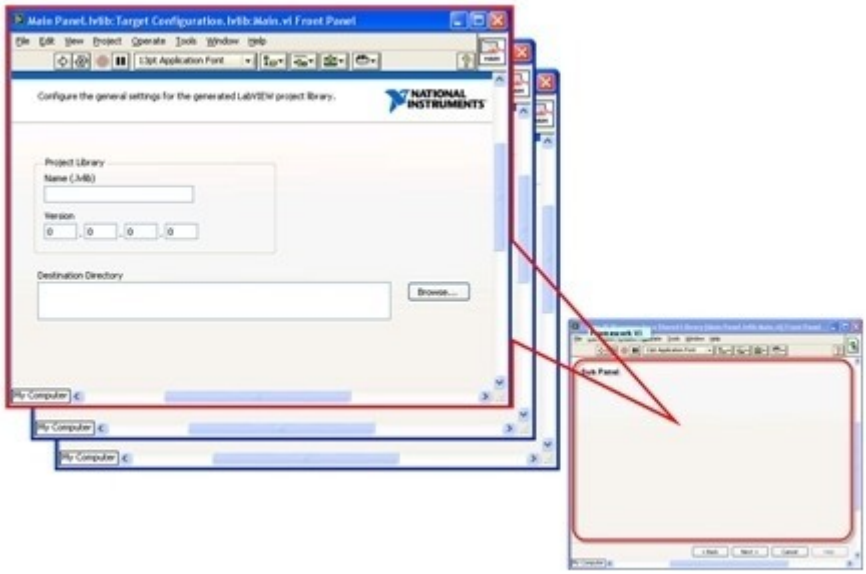


图1：插件框架式的程序结构

插件框架式程序的实现思路是，把向导的每个页面都分配到一个独立的 VI 上去，这个页面上所有的操作，都有这个页面所在的 VI 完成。图1左上部分的那些 VI 就是为每个页面编写的 VI。这些 VI 都被当作插件，在主程序需要的时候被调用显示在主程序上。

图1右下角的 VI 是主程序的 VI。它的界面上主要是一个 SubPanel 控件，这个控件用于显示页面 VI 的界面。主程序在每一步的时候，分别把对应这一步骤页面的 VI 的界面显示出来，这样就实现的向导功能。主程序的界面上还有一些公共控件，比如“上一步”“下一步”这样的按钮，这些按钮在所有步骤中都需要，所以可以放在主框架上，不需要再在每个页面中重复了。

这样的插件框架式程序在运行时，主 VI 和插件 VI 是在同时运行的。

主 VI 的运行流程大致如下：创建或注册程序运行时需要的各种事件 -> 初始化程序 -> 等待和处理事件，主要是管理插件。比如在用户按下“下一步”按钮后，主程序负责把当前的插件移出内存，把对应下一页的 VI 调入内存，运行，并显示界面。 -> 最后负责销毁创建的事件，关闭所有资源，退出。

插件 VI 的主要程序结构和主 VI 一样，采用的是事件处理结构。它在运行起来以后执行的流程也和主 VI 类似： 创建或注册插件运行时需要的各种事件 -> 初始化程序 -> 等待和处理事件，主要是用户在界面上的操作，和一些后台程序，比如数据处理等等。-> 销毁创建的事件，关闭插件。

虽然 SubPanel 在 LabVIEW 7.1 中就出现了，但是我当时却并没有在我的程序里采用上述的设计方案。只是因为当时还有一个棘手的问题没有解决。这个问题就是 VI 太多了，不好管理。

向导页面的多个插件 VI，他们的功能有很多共同之处。在以前，所有页面都在同一个主 VI 中的时候，那些相同的功能可以通过调用同一个子 VI 来完成。但是，把页面分割成独立 VI 之后，很多情况，我都不得不为每个页面做一整套子 VI，他们在每个页面上完成的功能都类似，但却不能使用同一个子 VI。

以处理事件为例，我写了一套子 VI 处理页面 VI 的事件。但是由于不同的页面可能会同时在运行，每个页面都有自己的事件，如果调用同一套处理事件的子 VI，不同页面之间会相互干扰。

另外，如果想创建一个新的页面，最方便的方法莫过于把一个已有页面的 VI、子 VI 全部复制一遍，然后在其基础上做改动。LabVIEW 以前是不允许出现同名 VI 的。把一个页面的 VI、子 VI 全部改名，还要保证调用链接不出现混乱，非常的不方便。所以上述的插件框架方案是我等到到 LabVIEW 8.0 出来以后才开始使用的。

六、Project Library

LabVIEW 8.0 作为一大升级版本，拥有了很多新特性。其中之一是“Project Library（工程库）”。

工程库是一组功能相关联的 VI 或其它文件的集合。一个工程库是把一组功能相关的 VI，和其她文件按一定结构组合封装在一起，以便于代码的管理和发布。工程库的名字也是库中 VI 的名字空间(name space)。这个名字空间与 C+ +、C#等语言中的名字空间的概念类似。有了它 LabVIEW 就可以在一个程序中使用两个同文件名的 VI，当然，它们的名字空间不能相同，也就是它们存在于不同的工程库中。

另外，工程库中的 VI 有操作安全设置， 每一个 VI 可以被设置为公有（Public，可以被库外的 VI 调用）；或者私有（Private，只能被库的成员 VI 调用）。

工程库给开发插件框架式的程序带来的很大的方便，特别是在 VI 文件的管理方面。

新的设计思路是这样的：把所有的功能模块都封装在工程库内，比如说每个页面都有一个对应的工程库。专为这个页面使用的所有子 VI 都被加在它的工程库内。并且，不想被其它库使用的的子 VI 都要标记为私有。被这样组织起来的程序，虽然 VI 数量还是很多，但模块划分清楚，不会出现不希望出现的调用关系。安全性，可维护性就大大提高了。

另外，可扩展性也得到了提高。如果需要添加一个新的页面，只需要把一个已有页面复制一份。复制出来的这一份，只要工程库的名字换个新的就行了。再也不需要一个一个的去改 VI 的名字了。

图2是我的一个程序的工程管理窗口：

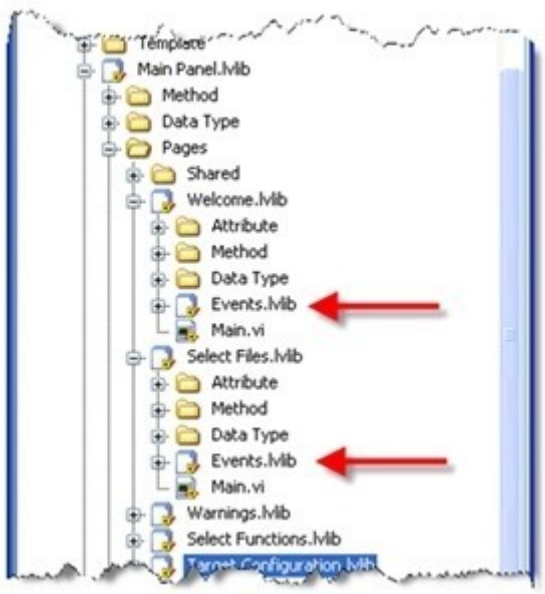


图2：采用工程库管理程序的 VI

但是，现在的程序结构还是有些令我不太满意的地方-重复的代码太多。不同页面之间，有很多类似的 VI。就比如图2中的程序，每个页面都会用到事件处理的一些 VI，他们的代码在每个页面中都是相同的。但是，利用这个工程库组织起来的程序却不能把这些重复的 VI 提取出来，变成共用的子 VI，因为在每个页面里，这些代码相同的 VI，处理的数据是不同的。并且这些数据会保留在 VI 的局部或全局变量中，不同的页面如果共用一套子 VI，会相互影响，出现数据混乱的。

直到 LabVIEW 支持了面向对象的编程之后，我们才终于找到了一个完美的解决这一问题的方案。

在 LabVIEW 中使用常量定义

如下图所示，在 C 语言里，使用#define 来定义一个常数是非常基本的用法。直接使用数字，时间一长，就不只到这个数字是哪来的了。而且，这种方法也便于修改在程序中多处使用的常量的值。在 C++一般是用 const 来达到同样的目的。

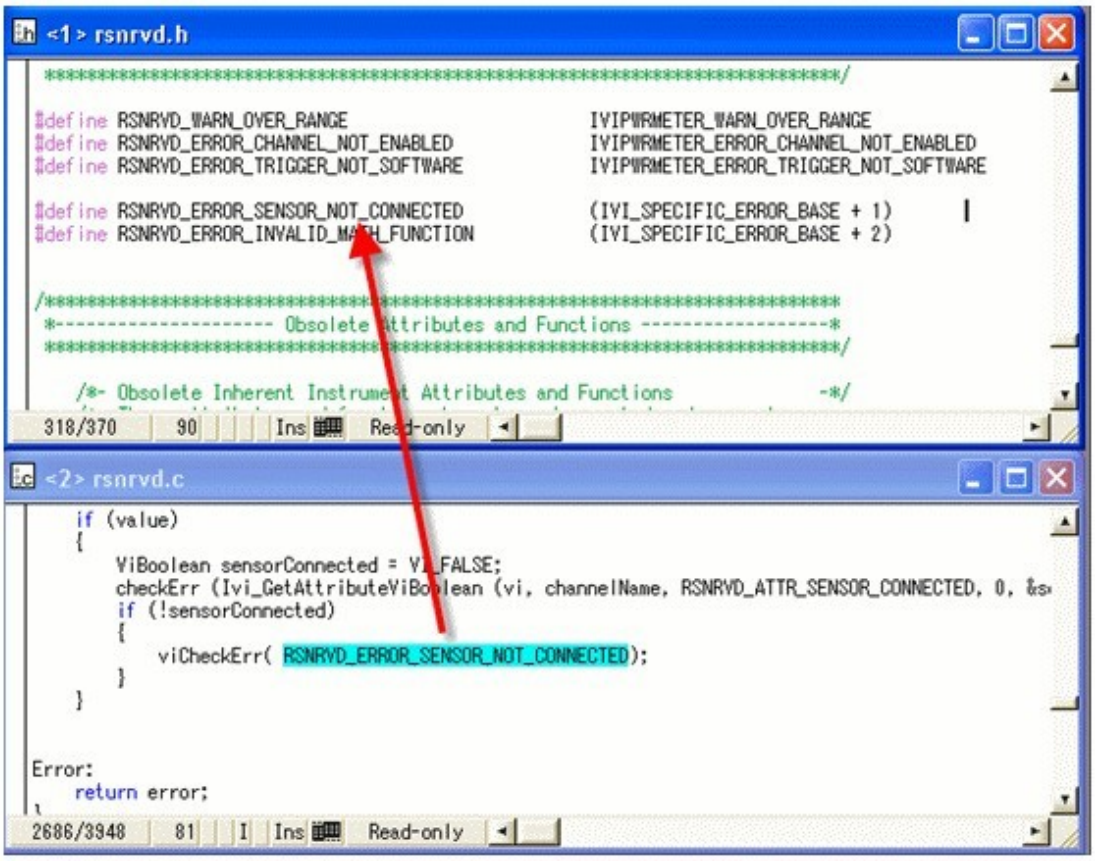


图1：C 语言中的常量定义

我以前在 LabVIEW 中编程，还从没注意过这个问题。一般哪里要用一个常数，直接就放一个 constant 在那里。如图2。

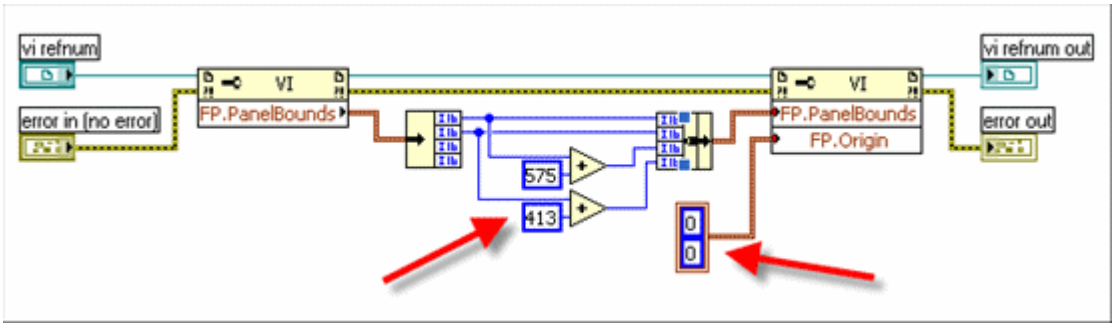


图2：在 LabVIEW 中使用常量的最普遍方法

以前编写的 LabVIEW 程序都比较小，一般是一个人开发的，所以这样写，也没有太大的麻烦。现在编写的程序规模越来越大，最近做的一个项目，VI 数量已经上千了，有4个人参与编程。程序规模大了，不规范就很难维护。所以开始考虑这个问题。

但是 LabVIEW 里面没有类似的功能，不知道为什么以前没人提意见？

下面提出几种不算太完美，但有所进步的解决方案。

一种简单的替代方法是使用type define control，自定义一个Ring control。关于 Type Def 的详细信息，可以参考《用户自定义控件中 Control, Type Def. 和 Strict Type Def. 的区别》。把要使用的常数作为Ring的值，给他个有意义的文字标签。在需要时用常数的地方，把这个带type define 的ring常数放上去，而不是直接放数值常量。这样就解决了上面提到的一个问题：可以有自带的文字说明。如图3所示。

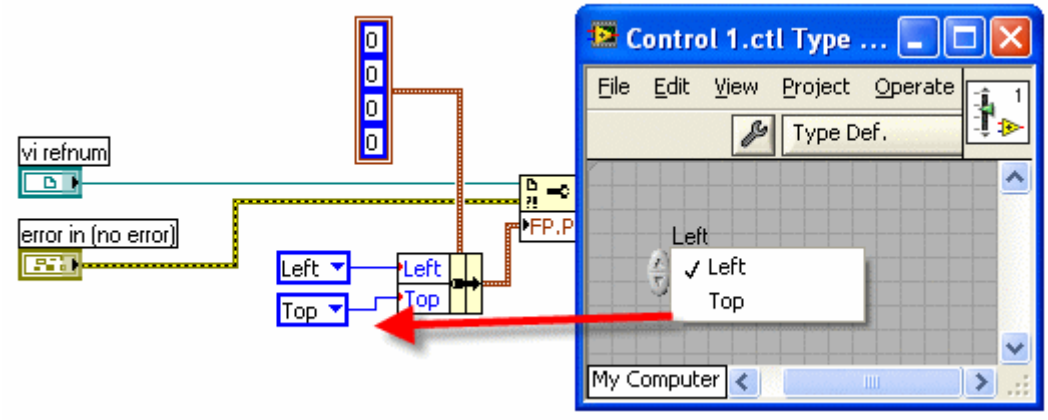


图3：利用 Type Def Ring 的解决方案

但是这样做还是有很多缺陷。首先是统一修改数值的问题。在自定义 Ring 中修改某一项的值，相关的常量不会跟着一起更新；还有一个缺陷是 Ring control 不支持多个标签是用同一数值；另外 Ring control 也没办法像 C 语言中一样使用表达式定义值。

一个改进版的解决方案是使用 Enum Type Def 把所有常量名字列出来，再写一个 VI 用于得到常量的真实值，如图4所示。这样解决了不同标签可以返回相同值的问题，也可以自动更新常量值，但是使用表达式还是不方便。

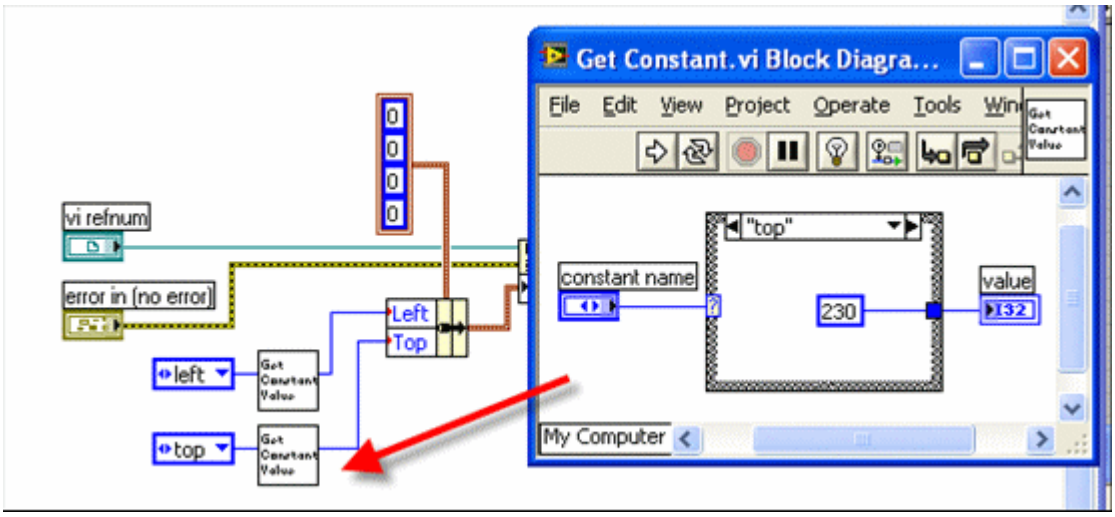


图4：利用 Enum Type Def 和 subVI

我目前在程序中使用的方法是，把所有要用到的常量，全部做成全局变量。全局变量可以用 Global ，但我喜欢用 VI 全局变量。就是把变量记载 shift regisiter 中。然后，用一个初始化的 VI 负责在程序运行开始时初始化所有的全局变量。这样，以后如果需要更改某一常数值，就只需改这一个 VI 就可以了。

不过，现在回想，还是用 Global 好一些。我以前测试过，Global 读写的速度比 VI 要慢很多，所以我不喜欢 Global 。但是，常量值在程序中用的并不频繁，所以速度不是个问题。但是数量很多，用 VI 表示就不太合适了，每个常数都要创建一个 VI 非常费事。另一个缺点是如果在后面板换用一个常量，还要再拖另一个 VI 上来，很麻烦。用 Global 会好一些，但还不是让我太满意。

要想有一个完美的解决办法，只能再造一个新东西了。@#$%^&* （此处属公司机密，删去256字）

多态 VI

一、多态 VI 的概念

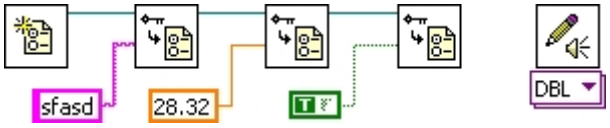


图1：一些多态 VI

LabVIEW 提供的一些 subVI，它们可以用于处理多种不同类型的数据。比如读写 INI 文件的 VI，它们既可以读写数值型的数据，也可以读写字符串、布尔等数据类型。类似的还用声音输出的 VI、数据采集的 VI 等等。

这种可以处理多种不同数据类型的 VI 被称为“多态 VI”。这个多态和 C++ 中的多态可不是一个意思，它更类似于 C++ 中的函数重载。

多态 VI 根据输入或输出的数据类型，再选择调用一个的针对这种数据类型实现功能的 VI。这些准对某种数据类型实现功能的 VI 被称作“实例 VI”。一般一个多态 VI 调用多个实例 VI。

在这种场合下，比较适合使用多态 VI：你帮助用户实现了一个算法，这个算法会应用到几种不同的数据类型上。为了用户使用方便，最好是不是给用户看到一组不同的 VI，这样他在使用前，还要根据数据类型的不同先去寻找适合的 VI；最好是指停工一个统一的接口 VI，这个 VI 可以接受不同的数据类型，这个接口 VI 自动的根据输入数据类型的不同，去调用相应的算法 VI。

二、如何实现多态 VI

比如说，我们现在需要提供给用户一个加法功能，它支持两种数据类型：整数和字符串。如果输入是两个正整数，输出就是它们的和；如果输入是两个字符串，输出就是把两个字符串连接在一起。

对于这个需求，我们需要实现一个多态 VI："add.vi"，这个 VI 支持两种数据类型：数值和字符串。这个名为"add.vi"的 VI 根据输入数据的类型，再去调用两个实例 VI：“add numeric.vi”和“add string.vi”来实现具体的加法功能。它们的调用关系如图1所示。

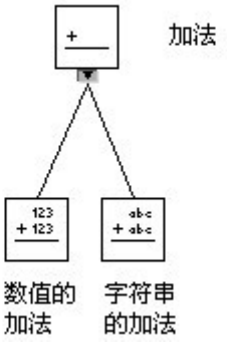


图2：一个实现加法功能的多态 VI 的调用关系

我们在实现这样的多态 VI 之前，一般先实现它的实例 VI，就是那些针对每个数据类型完成算法功能

的 VI。在这里是“add numeric.vi”和“add string.vi”。

完成了实例 VI，就可以开始创建多态 VI 了。在 LabVIEW 的菜单中选择 File->New，出现 LabVIEW 新建项目的选择对话框，再选择 VI->Polymorphic VI 就会出现一个新的多态 VI。

多态 VI 和普通的 VI 看上去不一样，没有前后面板。因为它的功能都是在实例 VI 中实现的，因此多态 VI 只要选择一下它的实例 VI 就可以了。

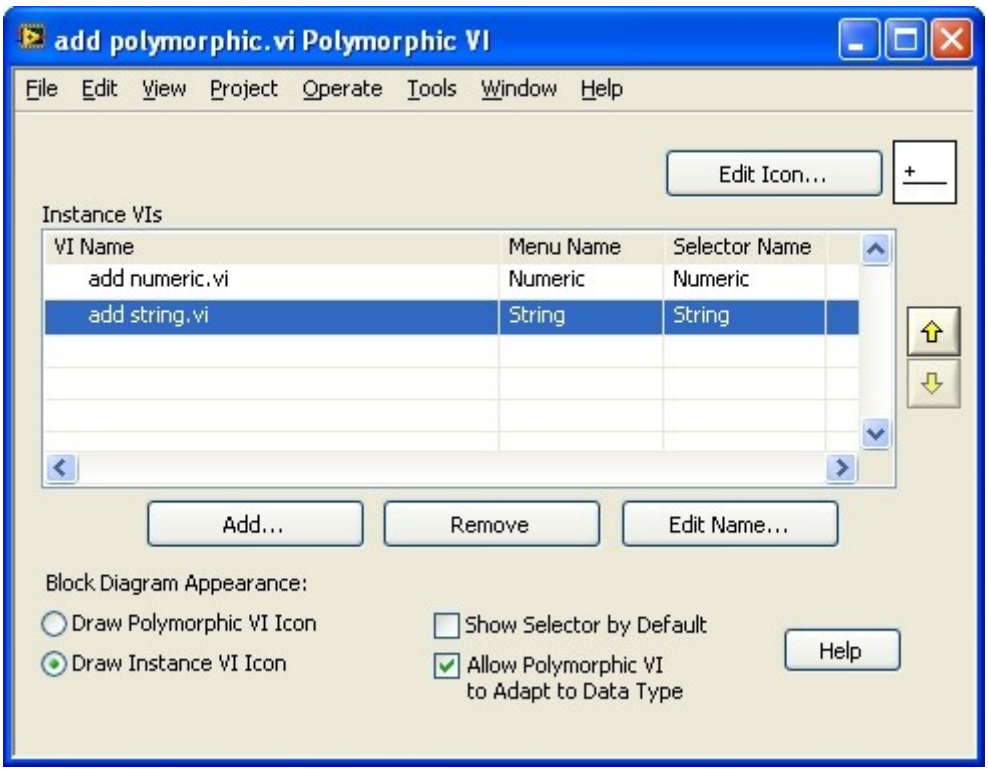


图3：多态 VI

在多态 VI 的界面上，右上方是这个多态 VI 的图标。我们可以在这里画一个图标，让用户在使用多态 VI 时，程序框图上一直显示此图标。但是，有时候使用实例 VI 的图标，可以让程序显得更清晰，那么我们可以在多态 VI 左下方配置此选项。

多态 VI 的主体部分是一张列表，通过“Add”按钮，可以把它的实例 VI 添加进来。在用户的程序框图上，多态 VI 的数据类型可以自动确定，也可以由用户通过右键菜单或选择栏（图1中多态 VI 下面那个紫色方框）来选择。实例 VI 列表中的“Menu Name”和“Selector Name”分别是在选择是代表每种数据类型的在菜单和选择栏中的文字，可以通过“Edit Name”按钮来编辑它们。

多态 VI 右下方两个选择框，“Show Selector by Default”表示当多态 VI 被拖到程序框图上的时候，就把选择框显示出来。否则，用户也可以通过右键菜单选择显示它。

“Allow Polymorphic VI to Adapt to Data Type”表示有多态 VI 根据输入数据类型的不同，自动选择调用一个相应的实例 VI。如果这项没被选中，编程者必须每次手动选择他想要的实例 VI。

三、多态 VI 的注意事项

在设计多态 VI 时，有一些事项需要注意。

多态 VI 只能处理有限种数据类型，它只能处理实例 VI 中处理了的那些数据类型。数据的类型是无限的，比如：包含两个整数的簇（Cluster）是一种数据类型，包含三个整数的簇就变成另一种数据类型了，

包含三个字符串的簇又是一种新类型。如果你想做一个多态 VI 可以像 LabVIEW 原有的加法函数一样处理无限种数据类型，是做不到的。

多态 VI 的每个实例 VI 可以是完全不同的，前面板，程序框图，使用的子 VI 等等都可以完全不同。但是，为了便于用户理解，一个多态 VI 应该就是处理某一种算法的，它的每个实例 VI 负责一种数据类型。并且，为了便于用户在不同的数据类型之间切换，每个实例 VI 的接线方块（Connector Pane）的接线方式都应当保持一致。

多态 VI 不能嵌套使用，一个多态 VI 不能作为其他多态 VI 的实例 VI。

四、小技巧

你可以把多态 VI 的右键菜单，选择某个实例 VI 那一项做成有层次的多级菜单。只要 Menu Name 中输入菜单的层次结构，使用冒号:作为层级分隔符就可以了。比如下面这个例子：

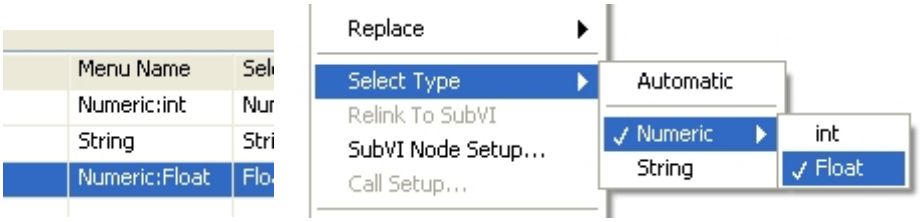


图4：有层次结构的快捷菜单

全局变量

全局变量是一种数据在不同节点，不同 VI，不同线程间传递的方式。数据被保存在某一固定的内存空间里，不随数据线流动。在需要读写数据的地方，不需要外部链接的数据线，直接通过某些节点或 VI 就可以得到目标数据，并对其操作。

在 LabVIEW 中应当尽量避免使用全局变量。全局变量看似方便，但带来的问题也很多。最主要的是它破坏了数据流顺序的逻辑关系，导致程序可读性和可维护性下降。

偶尔也有不得不使用全局变量或使用它利大于弊的情况。比如：实现子 VI 间参数传引用的机制；在不破坏程序可读性的前提下，避免一些过于杂乱的数据连线；对高层用户隐藏某些底层模块内部使用的数据。

一、全局变量（Global Variable）

此处所说的全局变量是特指图标像地球的那个 Global Variable VI。使用这种全局变量，目标数据被存放在一个只有前面板的特殊 VI 中，任何需要使用这个数据的地方，把它所在的 Global VI 拖过来即可。如同前面所述，全局变量虽然使用方便，但是缺点也十分明显。

首先，它不利于代码的可读性，破坏了数据流顺序的逻辑关系。使用全局变量难以知道数据是否在其它地方被改动过。换言之，代码上的全局变量，不能直观的反映出它的数据来源。

其次，它的安全性低。全局变量可以在任何地方被直接读写。即便知道数据在某些地方不应该被改动，也无法对其进行控制。

再次，它的效率低下。VI 每次读全局变量，LabVIEW 都要为读到的数据复制一个新的副本，这毫无疑问影响到 VI 的效率。

此外，全局变量的不合理使用还可能导致竞争状态。比如下图中的 VI，假设全局变量 Data 的值原本为 0，运行完下面这个加2减1后的代码后，Data 中的值是几呢？可能是1，也有可能是2，还可能是-1，这完全取决于程序的执行顺序。而在这种情况下，这个顺序是不确定的。

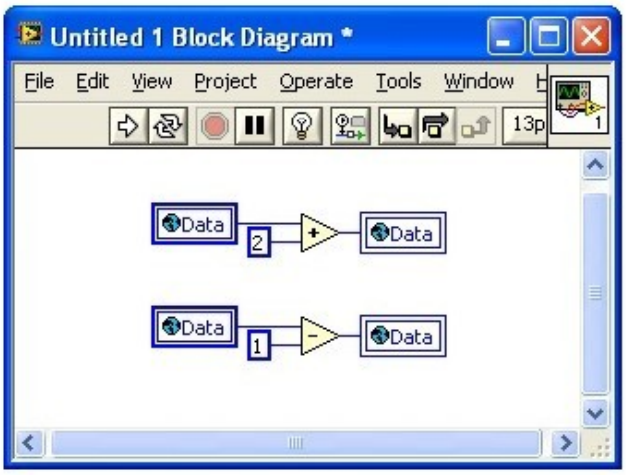


图1：处于竞争状态的全局变量

二、单进程共享变量（Single-Process Shared Variable）

共享变量有三种：单进程，网络发布，以及时间触发的共享变量。后两种主要应用于不同硬件设备、不同计算机、不同进程程序间的数据交换。在此，我们仅仅介绍与全局变量相关的第一种：单进程共享变量。共享变量的种类可以在它的属性页中进行修改。

单进程共享变量，顾名思义就是作用域为单个程序进程的共享变量。它与全局变量的性质是完全相同的。唯一的不同点是单进程共享变量带错误输入/输出端，我们可以利用错误处理连线来控制单进程共享变量的执行顺序。比如下图中的 VI，假设共享变量 Data 的值原本为 0，运行完下面这个加2减1后的代码后，Data 的值必然为1。

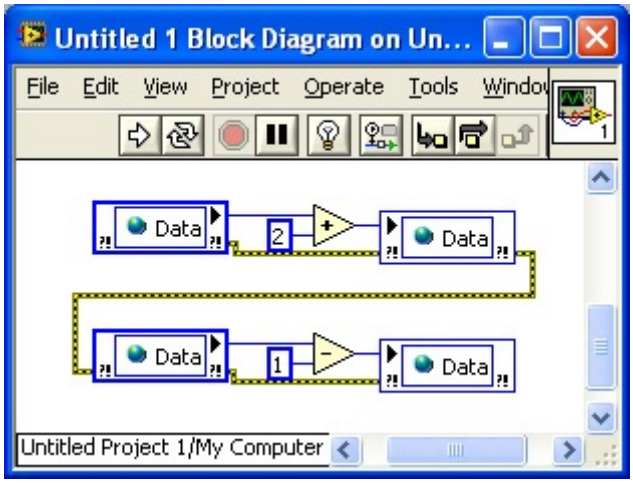


图2：共享变量的使用

这并不意味着单进程共享变量可以防止出现竞争状态。设想上图的 VI 只是程序中的一个子 VI，在其运行的同时，Data 仍然可以在其它子 VI 中被访问，因此，仍然有可能处于竞争状态。

共享变量是 LabVIEW 8 之后的一个新东西，比必须被保存在某个 Library 内部，不能独立存在。

三、功能全局变量（Functional Global）

功能全局变量与前两种全局变量完全不同，它在 LabVIEW 中就是一个普通的 VI。它把需要在全局使用的数据保存在一个没有初始化的移位寄存器中，并实现相关的访问这些数据的方法。

功能全局变量的代码结构都是类似的：主体是一个只执行一次的循环结构；内套一个选择结构；一个输入用于选择某种操作；若干用于输入和输出数据的控件。

其实，使用循环结构仅是为了利用它的移位寄存器。移位寄存器没有连初始化数据，因此每次执行这个 VI 时，它里面保存的是上一次 VI 执行结束时的数据。这样，就可以在程序的全程保存、处理或使用这一数据了。功能全局变量 VI 不可以被设置为可重入，否则在不同地方，得到的移位寄存器中的数据就不是同一份了。

从 LabVIEW 8.5 开始，VI 的程序框图增加了反馈节点了。可以使用它来替代仅执行一次的循环，以简化程序。下面两图就是完全等效的功能全局变量。

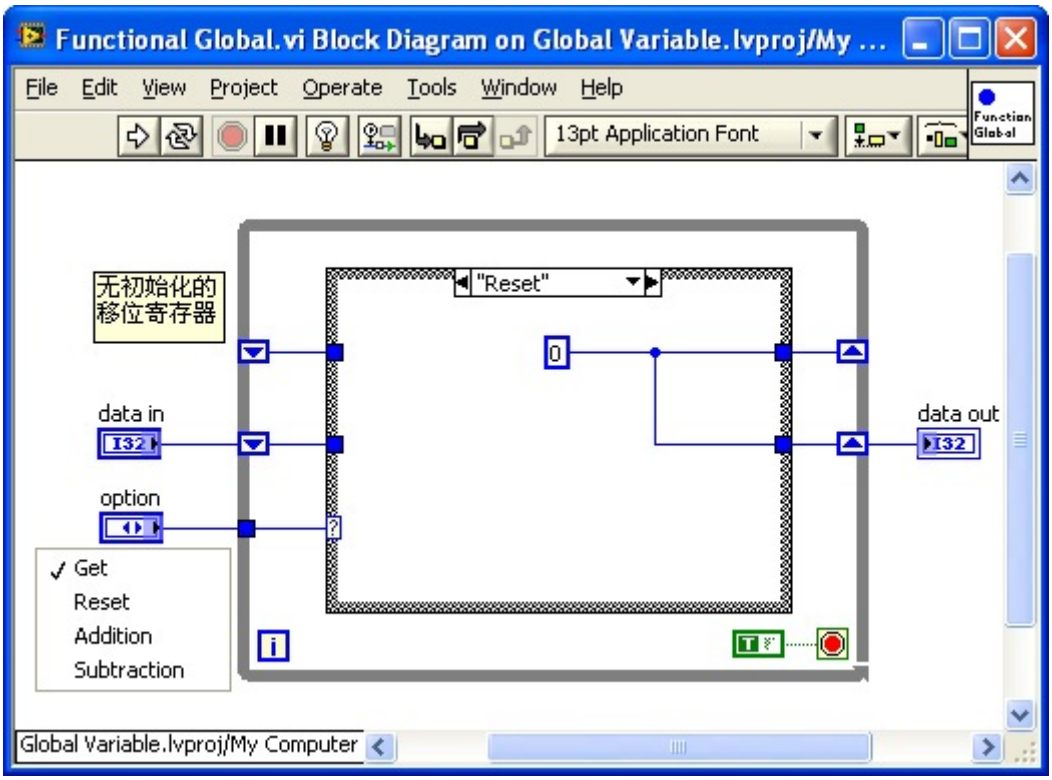


图3：实现加减法功能的功能全局变量

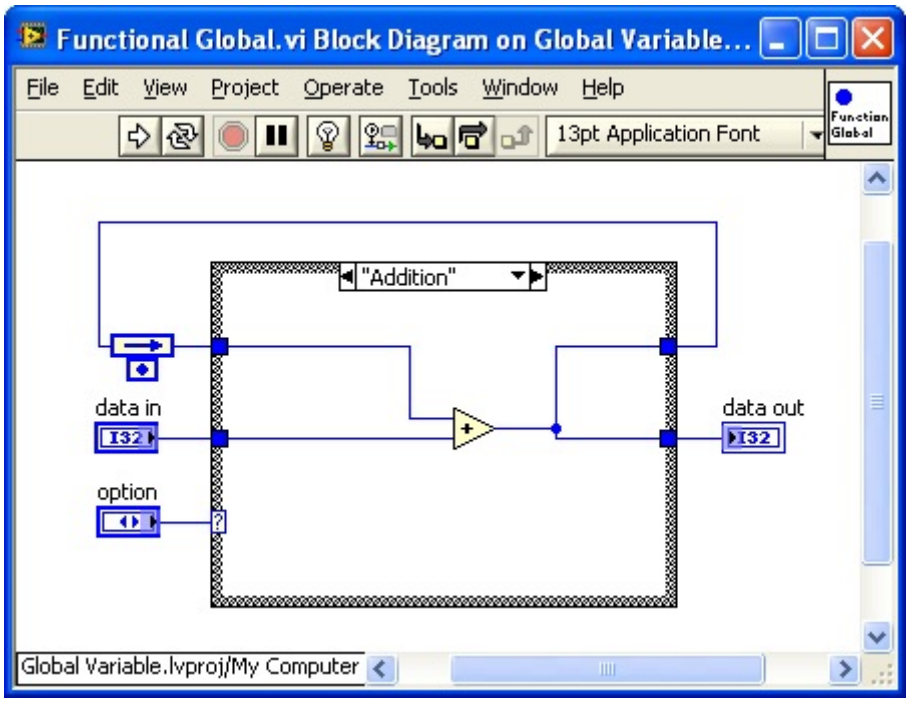


图4：等效的加减法功能全局变量

与前两种全局变量相比，功能全局变量有两项主要的优点。所以我建议，如果不得不使用全局变量，那就使用功能全局变量。

首先，功能全局变量可以防止竞争状态出现。因为功能全局变量的 VI 是不可重入的，所以把它作为子 VI 时，绝对不可能出现两个相同功能变量子 VI 同时执行的情况。因为对全局数据的所有操作都是在这个 VI 内部完成的，也就意味着，所有对数据的操作都绝

对不会被其它操作干扰。图5中的 VI，执行结束必然导致全局数据增加1，即便还有其它线程的子 VI 在同时运行也不会影响这个结果。但需要注意的是，解决了竞争状态不等于全局变量的使用顺序可以乱写，随机的顺序很可能导致错误。设计功能全局变量时可以加入出错处理的连线，以方便使用时确定全局变量的调用顺序。

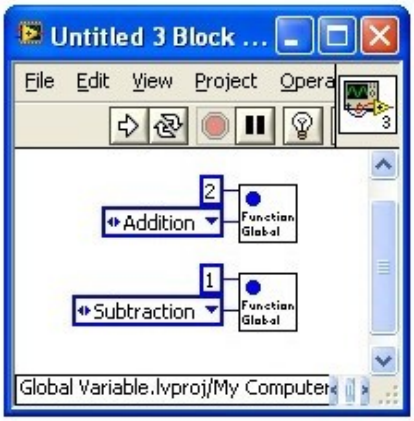


图5：功能全局变量的使用

另一优点是，功能全局变量可以封装内部数据、控制对数据的访问权限。例如图3 所示的那个功能全局变量，故意没有设置数据的方法。因此，使用这个全局变量的高层程序是无法直接修改全局变量的值的，只能使用给定的方法：复位或加减。甚至也可以不对高层程序提供直接查看全局数据的方法，只允许通过某些方法得到数据处理后的结果。这样全局数据就被很好的隔离开来，避免了被不当改动的风险。

由于功能全局变量的这两个优点，它一度受到程序员的极力推崇。我读过一本名为《A Software Engineering Approach to LabVIEW》的书，它的核心思想就是建议读者把程序模块全部写成功能全局变量的形式。

功能全局变量虽然有上述优点，但用它来作为较大功能模块的框架，还是存在很多不足的。首先，这种实现方法，模块最主要的功能和代码都在同一个 VI 中实现。这个 VI 会变得十分复杂，难以维护。其次，模块如果接口的数据较多，这个 VI 的连线就会极为复杂。再有，如果模块要增加或改动点什么功能，这个大 VI 参数可能会发生变动，引起版本不兼容的问题。

LabVIEW 从8.2 版本起，已经支持面向对象的编程，类的概念发挥了功能全局变量的优点，克服了其缺点。因此，再设计功能模块应该首先考虑 使用 LvClass。另外，利用 LVOOP 可以设计出更便于维护的功能全局变量，不过这更加复杂，我们以后在介绍 LVOOP 的时候一并介绍吧。

LabVIEW 工程资源管理器

从 8.0 开始，LabVIEW 增加了一个工程资源管理器功能。LabVIEW 工程资源管理器就是一个可以方便查看、调整程序系统结构的工作区。与 VC, VB 等语言中的 project, workspace 相类似。Project 的出现使得 LabVIEW 对于大项目的管理更加方便。

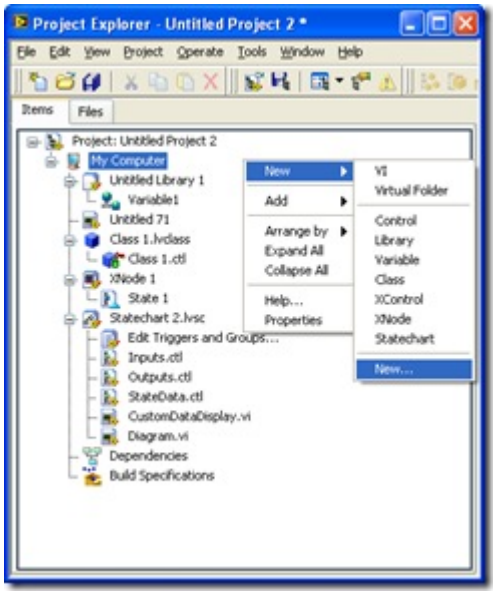


图1：LabVIEW 工程资源管理器

要想充分利用 LabVIEW 工程资源管理器带来的好处，那就需要我们改变一些以往的 LabVIEW 编程习惯。譬如说，在开始一个新的项目的时候，在 LabVIEW 8 以前的版本中，我们最先创建的是程序的主 VI; 而现在，我们首先创建的应该是一个空的工程(Project)，再在这个工程中添加相应的 VI 和各种组件。

工程管理器还有如下优点：

工程的树形结构表示了程序中 VI 的调用层次关系，利用 VI 的快捷菜单可以查看到调用该 VI 的程序，以及该 VI 的子程序，而不必再使用 VI Hierarchy 来查看。

在工程资源管理器的 File 页就可以直接调整文件存放的磁盘位置，而不必再另外打开操作系统提供的文件浏览器。

在工程资源管理器中集成源代码管理功能，不需要再使用源代码管理工具提供的界面了。(源代码管理工具是用来进行软件源代码版本控制的。大型软件开发通常需要这样的工具，用来记录每一次代码的修改、同时开发同一软件的不同版本、方便多人同时对同一段代码进行修改等。)

一、工程的结构

图1是一个工程资源管理器的截图。它用一个树形的结构来表示工程中所有的 VI、各种组件和文件设置等。

树形结构的最顶层是工程的名称。

第二层是工程运行的目标机器。假如我的机器上只装了普通台式机版本的 LabVIEW，大家只能看到一个目标："My Computer"。假如我的计算机上还装了 LabVIEW RT, FPGA 等用于其它硬件环境的 LabVIEW，那么在这一层还会出现其它那些目标设备。

第三层以下就是工程中所有使用到的文件了。用户可以添加虚拟文件夹，按自己的喜好组织文件结构。LabVIEW 从 8.0 起，文件及其它组件种类一下子丰富了许多。以前基本就只有 VI 和控件两种文件，现在又多了 Library, Class, XControl，XNode，共享变量等等。安装了其他功能模块，组件的种类还会更多。

右击树状结构中的每个条目，还可看到针对他们的更多设置。

在 LabVIEW 8 之前，若要把 VI 源文件构建成可执行文件，必须使用 Tools 菜单下的 APP Builder 工具。现在这个工具也被集成到了工程管理器中。在目标机器的最后一个条目"Build Specification"中包含了把源代码配置成为 EXE, DLL 等的信息。

在旧版本 LabVIEW 中，保存 VI 时的一些高级选项，比如添加密码、移除 VI 前面板，程序框图等选项;以及其它一些与运行有关的选项，例如禁止调试，自动弹出错误框等选项，也都被合并到此处了。在 Build Source Distribution 中可以找到相应的设置。

在旧版本 LabVIEW 中，是绝对不允许把两个文件名相同，但内容不同的 VI 同时装入内存的。这也可以理解，VI 就好比是 C 语言中的函数，如果两个函数名相同，那到时究竟应该使用哪一个呢?但这毕竟不方便，比如我们需要同时运行两个程序，他们当中都有排序的功能。为了方便，VI 的名字都起名为"Sort.vi"，但是他们的代码其实并不相同。为了让两个程序同时正常工作，我们应该允许在这两个程序内分别使用两个文件名相同，但内容不同的 VI。

在版本 8 中，LabVIEW 对此作了两点改进，一是引进了库的概念，类似给 VI 增加了名字空间。在不同的库中的 VI，即时文件名相同，它们的 VI 名字实际上也是不同的，因为 VI 名还包括库名作为前缀。(可以参考：http://ruanqizhen.spaces.live.com/blog/cns!5852D4F797C53FB6!783.entry)

二是增加了运行环境的概念。LabVIEW 中，每一个工程都是一个独立的运行环境。为每一个的程序建立一个单独的工程，它们就有了一个独立的运行环境。一个运行环境中的 VI 不会影响到其它运行环境中的程序。这样，同时运行分属不同工程的两个程序，它们之间即便用到了同名的 VI，也不会相互影响。

但在单个工程中，还是不允许出现同名又不在库中的 VI。

二、按照文件的物理结构来查工程

在需要创建一个新的类似工程，或版本备份时，程序经常被来回复制。在这个过程中，很可能会引起子 VI 的错误链接。比如，本来工程里应该使用的的是 Project One 文件夹下的一个子 VI，但实际上却链接到了 Project Two 文件夹下的一个同名 VI。我们可以把文件的真实路径同时显示在工程资源管理器的 Item，以及 File 页上，选取菜单 Project -> Show Item Paths，如图2所示。我们可以在这里检查每个文件的路径是否正确。

但是，文件比较多的时候，一条一条看下来，是比较麻烦的。这时可以按照文件的物理结构来查看工程中的文件。在工程资源管理器中选择“Files”标签页，看到的就是文件在物理硬盘上的真实结构。我们只要在这里检查一下，有没有不需要的文件夹出现，就可以判定是否出现了错误链接。

如果发现某些文件所存放的路径不恰当，需要调整，我们可以直接在工程资源管理器中来调整，而不需要打开文件浏览器去修改。

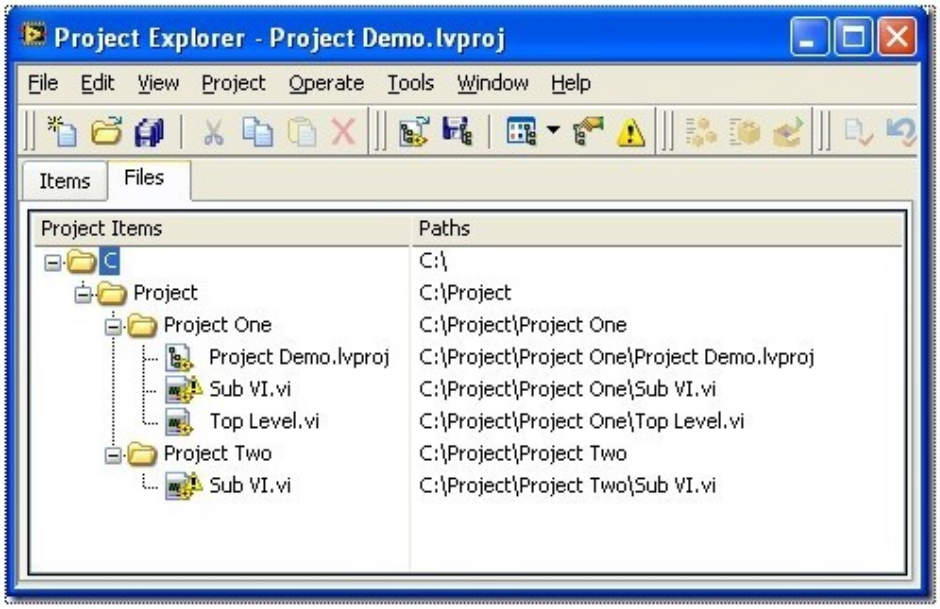


图2：按照文件的物理结构查看工程

三、VI 交叉连接

前文提到程序可能会错误的链接子 VI，这在 LabVIEW 中被称为 VI 的交叉链接。除了前面提到的可能引起交叉链接的情况外，试图调用重名的 VI，或把它加到工程里来等操作，都会引起 VI 的交叉链接。

VI 的交叉链接会引起很多问题，比如你在修改 VI 的时候，改动了不应该改动的那份;引起程序不可预知的行为等等。LabVIEW 的工程资源管理器可以帮助检查、修复 VI 的交叉链接。

当工程资源管理器发现程序中试图使用两个同名 VI 时，就会在这样的 VI 上打出一个惊叹号。如图2中所示的 Sub VI.vi。工程中试图用到了两个同名的 VI，分别在路径 Project One 和 Project Two 下。你可以到引用这些 VI 的地方去把他们一一修正，也可以让工程资源管理器来修正它们。回到“Item”页面，在有惊叹号 VI 上点击鼠标右键，选择 Resolve Conflicts。这时候，就会出现修正冲突的对话框。在这个对话框中选择应该被选用的正确子 VI，工程资源管理器就会自动更新程序，使它们链接到正确的 VI 上去。

传引用

传值是符合数据流驱动程序的传参方式，LabVIEW 中应该尽量使用这种方式。但是传引用在某些情况下是不可避免的，比如程序要在不同的线程中对同一数据进行操作，就得用到传引用。

引用在 C++ 中和指针本质上是一个东西，只是使用规则有些不同罢了。它们都是一个4或8字节的整数，这个整数表示的是目标数据所在的地址。程序代码通过这个地址来访问数据。

在 LabVIEW 中，没有指针的概念，但是我们可以通过多种形式来完成传引用的功能。下面我们就来讨论一下这些传引用的形式。

一、LabVIEW 自带的传引用数据类型

在 C++ 语言中，调用子函数时，可以指定某个参数是传值还是传引用。LabVIEW 采用的是完全不同的机制：在一般情况下，数据类型决定了这个数据是采用传值还是传引用。LabVIEW 中大部分数据类型是值传递的，一少部分数据类型专门用于传引用。例如，控件选板上的 Refnum 栏上的控件就都是传引用数据类型的控件。在程序框图上，用深色细绿线表示这类传引用的数据类型。

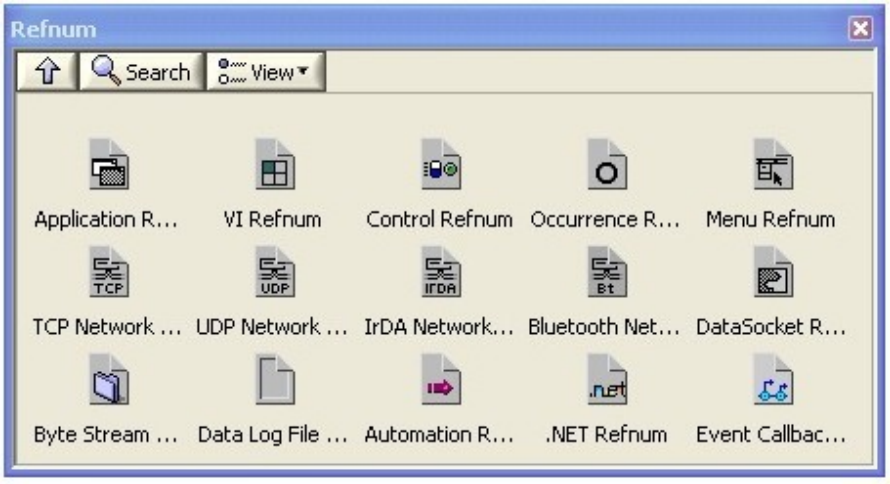


图1：Refnum 选板

使用 VI Scripting 编程时常会使用到 VI Refnum, Control Refnum 和其它对象的引用。与传值不同，在传递这些数据时，如果数据线分叉，并不意味把它们所表示的控件等复制了一份。新分出来的 Refnum 还是指向原来的那个控件。

除了各种 Refnum，LabVIEW 中还有其它一些数据类型，尽管其数据线的颜色不同，其实也属于传引用的数据类型。它们包括了硬件设备的句柄（VISA Resource, IVI Logic 等），notifier, event, queue 等等。

二、全局变量

在实际编程过程中，更常见的是我们希望把一个普通类型的数据按传引用方式传递。比

如一个数组，一个自定义的簇等。

全局变量是一种最简便的传引用的方式。全局变量的数据被保存在某一固定的内存空间里，但在不同的 VI 或线程中，都可以通过表示这个全局变量的对象来访问数据。

在使用全局变量时，直接把表示全局变量的 VI 或节点放在程序中就可以访问它的数据了。这种方式尽管有其优点，但更多的却是缺点。我们在阅读 LabVIEW 程序的时候，数据线是非常重要的线索。它为我们指明了程序执行的顺序，数据传递和加工的过程。失去数据线这一重要线索，就不容易搞清楚这个数据是从哪里来的，何时被改动。因而大大降低了程序的可读性和可维护性。

不过不要紧，下面提到的方法将会解决这个问题。

三、队列

LabVIEW 中有一套操作队列的函数。

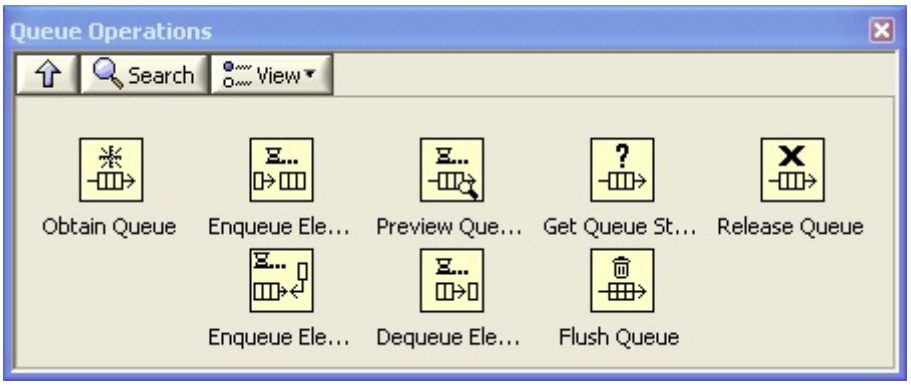


图2：队列的函数选板

LabVIEW 中的队列是双向队列，堆栈也可以使用它。它与其它语言中的队列一样，为数据数据提供了存入取出的操作。一般用于不同线程、不同设备等之间通讯时数据的缓存。

但 LabVIEW 中的队列它有其特殊性。其他语言中，队列主要作为一种数据结构，是一种便捷的在内存中保存需要顺序处理的数据的方式。在 LabVIEW 中，队列更主要的应用于在不同的线程间的数据交换。因此，队列不同于 LabVIEW 中大多数的数据类型，它是传引用的。这样才能在不同的线程内对同一个队列进行操作。

我们可以借助队列，使任意一种类型的数据按照传引用的方式传递。其思路是：创建一个新的只有一个元素的队列，把数据作为这个队列的元素。平时在 VI 间传递参数时，传递的是这个队列。需要时，再把数据从队列中取出使用。下图是初始化一个这样的传引用数据的代码：

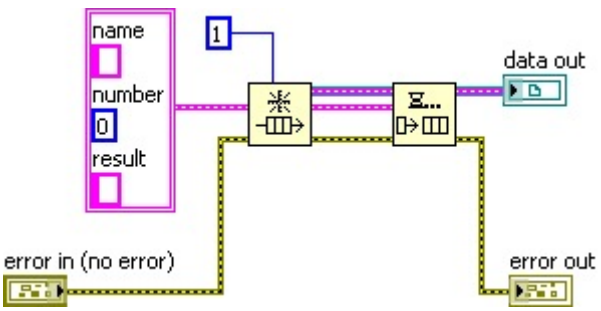


图3：创建一个传引用的数据

传引用通常都是用于在不同线程里访问同一份数据，所以在访问数据时要防止出现竞争状态。一个数据处理的 VI（假设名为 A），第一步操作就应当是用“Dequeue Element”把队列中唯一的元素取出。在 VI （A）所有工作都完成后，再让新的数据重新入队。这样一来，程序执行到 VI（A）时，队列立即被清空。其它线程内若有 VI （假设名为 B）准备同时处理同一数据，此时它已经无法从空队列中取出所需的数据。它只能暂时等待，直到 VI（A ）完成所有工作，再次把数据放回队列，VI（B） 才能继续执行。这样就避免了同一数据被同时访问而引发的竞争状态。

下图就是一段处理数据的示例代码：

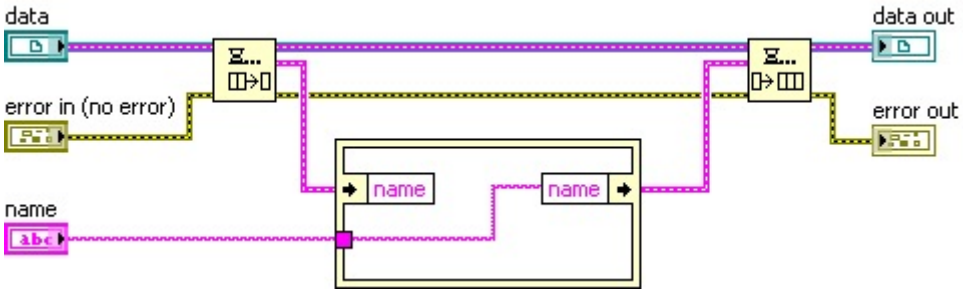


图4：清空队列、处理数据、重新入队

（有人问起，代码中那个黄色可框是什么。它是缓存重用结构。）

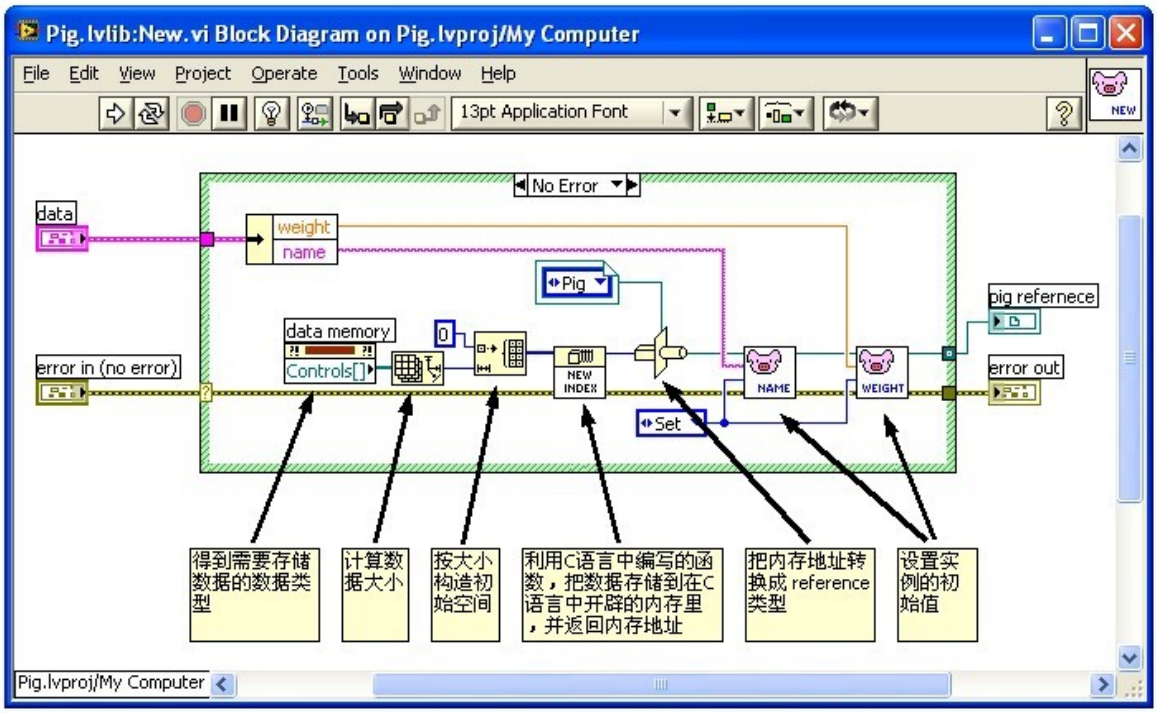
使用传引用，就必须自己管理内存了：当数据不再使用时必须把创建的队列销毁，否则可能会引起内存泄漏。

四、借助 C 语言

借助 C 语言比借助队列实现传引用要麻烦一些，所以这不是一个首选方案。但是，如果软件中已经有部分模块是使用 C 语言编写的，并且所传递的数据在 C 代码中和 LabVIEW 代码中都会使用到，也可以考虑把数据存放在 C 语言实现的模块中。

这种做法的思路是，数据存放在 C 语言开辟的内存空间里。C 语言把数据的内存地址传给 LabVIEW。平时在 VI 间传递参数时，传递的是这个地址的数值。需要时，再把数据从内存中读到 LabVIEW 里使用。

在下图的小猪的演示程序中使用到了这种传数据引用方法。下图是演示程序中创建数据引用的 VI：Pig.lvlib:New.vi



先不去考虑这个例子中的数据具体是什么，值分析一下它如何构造数据的引用：在这个 VI 的正中间是一个显示为“NEW INDEX”的 VI。它的功能是把一段数据放置在 C 语言中开辟的内存里，然后返回保存数据的内存地址。内存地址用 I32整数类型标识。以后在每个子 VI 间传递的数据就是这个内存地址的值。

你可能已经注意到，程序把这个内存地址又强制转换成了一个 Refnum 数据类型。这不仅是为了看着舒服（LabVIEW 中的传引用大多使用这种数据类型），更是因为使用自定义的 Refnum 类型，比整数有更高的安全性。例如你的程序中有不同的几块数据都采用这种放法保存在 C 语言开辟的内存中，使用不同的 Refnum 类型可以分别地将它们区分开来，避免直接把地址值传递给需要使用另一块数据的 VI。（

利用循环条件结构控制几个任务的执行顺序

循环条件结构不是一个基本结构，它是指在循环结构内套一个条件结构，这样的复合结构。这是 LabVIEW 中常见的程序结构之一。

假设需要编写这样一个测试程序，它有多个测试任务：任务 A、任务 B……，需要顺序执行每一个测试任务。这是一个典型的顺序结构的程序，可以采用上一章提到的顺序程序的编写方法。它的代码如下：

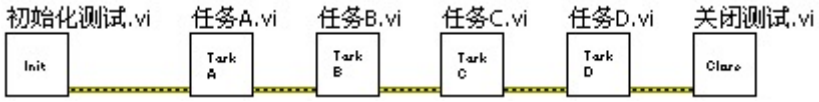


图1：顺序执行测试任务

如果程序要求更复杂一些，这个简短的顺序结构就不够灵活了。比如，有多中产品需要测试，但每种产品的测试流程不一样，有的产品需要测试任务 ABC，有的需要测试任务 CDB，等。针对不同产品编写不同的测试程序不是一种高效的方法。

高效的方法是把测试任务做为测试程序的输入，程序根据用户每次指定的测试任务顺序来调用测试任务。这个程序可以使用循环条件结构来完成。它的程序如图2所示。

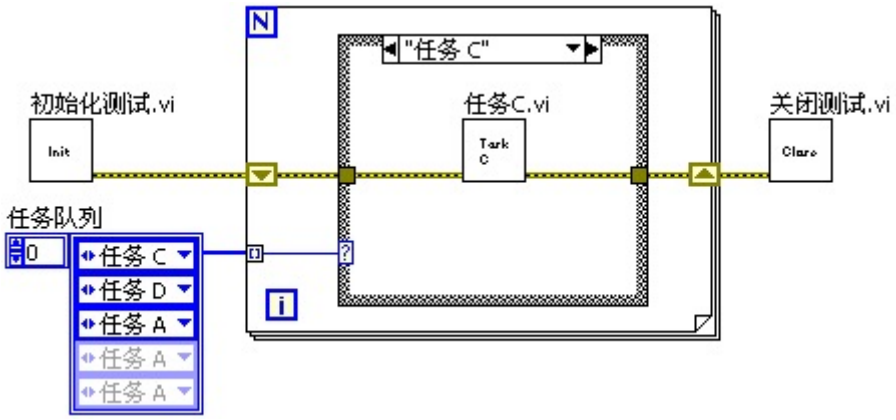


图2：按照输入的顺序执行测试任务

这个程序中的“任务队列”应该是一个输入控件，这样用户不需要改动程序，就可以改变它的输入值。但是在这里为了便于观看，把它变成了一个常量。“任务队列”是一个数组，元素按照找任务执行的顺序排列。这样在程序运行时，循环每迭代一次，循环结构从“任务队列”中取出一个任务，然后由条件结构判断该任务并进入相应的分支，执行该任务。

LabVIEW 的调试环境

1. LabVIEW 的全局选项

在 LabVIEW 8.2 中打开 Tools -> Options 菜单项，选择其中的 Debugging，会出现如下四个选项：

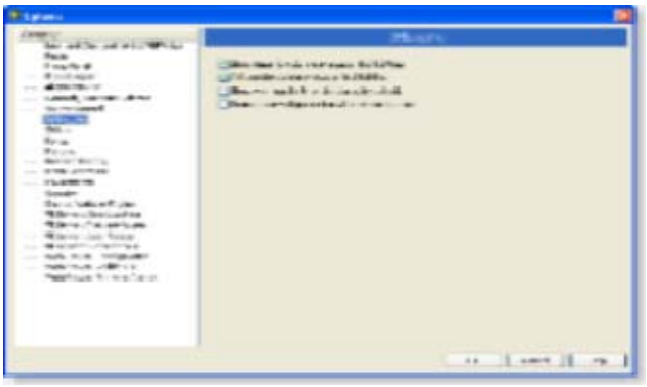


图1：LabVIEW 与调试相关的选项

a) Show data flow during execution highlighting 表示在高亮显示执行的过程中显示数据的流动。

b) Auto probe during execution highlighting 表示在高亮显示执行的过程中，数据从每个接线端流出时，显示数据的数值。

c) Show warnings in Error List dialog by default 表示在默认情况下，在错误列表的对话框中显示警告信息。

d) Prompt to investigate internal errors on startup 表示在 LabVIEW 启动时检查是否存在内部错误。

如果你仅从字面上还不能理解上述几个选项的含义，不要紧，后面的章节里会详细介绍它们的含义。

2. VI 的属性

某些 VI 的属性设置可能会导致你无法调试这个 VI。比如，VI 被设置为有密码保护，而你又不知道密码是什么；又比如，VI 被设置为不允许调试等。禁止 VI 调试可以大大提高 VI 的运行速度，降低 VI 的内存占用，所以，在 VI 发布给用户之前，最好把它设为不可调试。

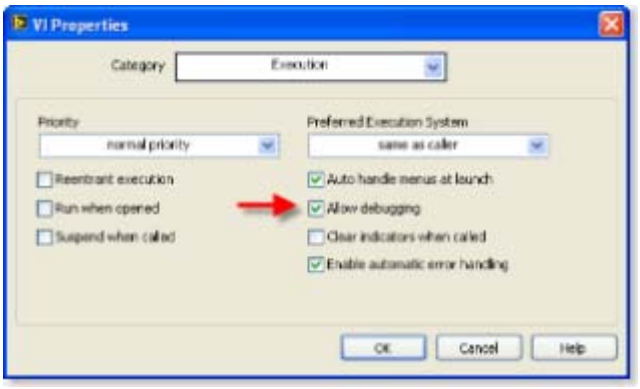


图2：VI 的属性设置

3. 调试工具

VI 程序框图上的工具栏中，某些按键是用于调试的。

图3：正在运行的一个 VI 的程序框图

图3 是一个正在运行的 VI 的程序框图。我们看到的工具栏上的按钮的图形，基本就可以猜出它的功能了。

 用于停止整个程序的执行。

 用于暂停或者继续程序的执行。

 用于启动高亮显示执行。在高亮显示执行时，LabVIEW 会放慢代码的执行速度，并且在程序执行到每一个节点时，高亮显示这个正在被执行的结点。高亮显示执行的速度非常慢，所以启用它要非常小心。如果启动高亮显示的同时，你的某个 VI 前面板是模式的（modal），那么你想中途关掉它是不可能的了，你只能非常痛苦地等待程序的结束，或杀掉整个 LabVIEW 进程。

 用于保留 VI 程序框图上数据线中的数据。

 用于单步执行，它们三个分别表示进入、跳过或跳出某个节点、结构以及子 VI。

下拉框表示 VI 的调用关系。打开下拉框，可以看到当前 VI 从低层到高层的逐级被调用关系。选择下拉菜单中的某一项，即可跳到那个 VI 被调用的地方。

 是设置断点的地方。

 是设置探针的地方。图3 上的悬浮窗口显示的就是探针所在处的数据。

在需要设置断点和探针的地方按鼠标右键，在弹出菜单里可以选择 Set Breakpoint 或者 Probe，或者通过使用工具选板（Tool Palette）上的断点和探针工具进行设置。

断点和探针

1. 断点

断点和探针是调试 LabVIEW 代码时最常用的两个工具。LabVIEW 中的断点在使用和功能上都比较简单、直观：使用工具选板上的断点工具，在想要设置或者取消断点的代码处点击鼠标即可；或直接在程序框图的节点、数据线上右击鼠标，就可以看到设置或取消断点的菜单项。

断点几乎可以设置在程序的任何部分。当程序运行至断点处，就会暂停，等待调试人员的下一步操作。很多其他语言的调试环境都有条件断点，LabVIEW 的端点没有类似的设置，LabVIEW 是使用条件探针来实现条件断点功能的。

断点是会保存在 VI 中的。关闭带有断点的 VI，程序执行至断点处还是会停下来，并且这个 VI 会被自动打开。

如果某个 VI 不允许你设置断点，很可能这个 VI 被设为不允许调试了。此时，只要在 VI 属性中重新设置一下即可。（LabVIEW 的调试环境.2）

2. 探针

探针的功能类似于其他语言调试环境中的查看窗口，用于显示变量当前状态下的数据。LabVIEW 与其他语言不同之处在于，LabVIEW 是数据流驱动型的图形化编程语言。LabVIEW 中的数据传递主要不是使用变量，而是通过节点之间的连线完成的。所以 LabVIEW 的探针也不是针对变量的，而是加在某根数据线上的。

LabVIEW 的探针也是图形化显示的。比如为一根数字类型的数据线加探针，探针一般就是一个数字型显示控件，见图1。Error Cluster 类型的数据线的探针，则看上去就像是个 Error Cluster，见图2。

图1、图2：数值型和错误信息型数据线的探针

3. 选取其他类型控件作为探针

如果你觉得 LabVIEW 默认的探针不美观或不适用，则可以在数据线上点击鼠标右键，选择 Custom Probe -> Controls -> ... 选取一个其他控件作为探针，如图3。但是要注意，你选取的控件的数据类型要与数据线的数据类型一致才可以。

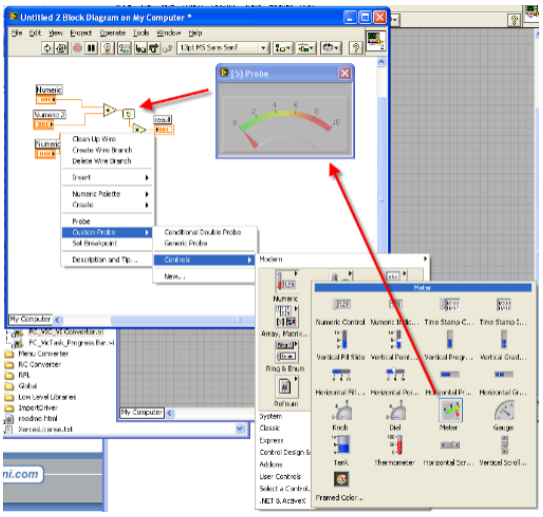


图3：使用仪表盘控件作为数值型数据线的探针

4. 条件探针

在你设置断点后，程序在每次执行到断点的时候都会停下来。但有的时候，调试者希望程序只在被监测的数据满足某一条件时，才暂停运行。比如，被监测的数据在正常情况下应大于零，调试者希望一旦数据小于零则暂停。在 LabVIEW 中，可以使用条件探针来实现这样的功能。

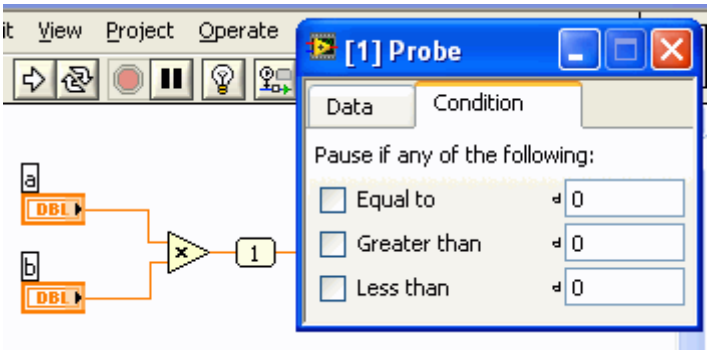


图4：数值型条件探针

以图4 为例，如果你希望程序中的循环在运行 8 次以后才停下来，就可以使用条件探针。在记录循环次数的 i 的输出数据线上点击鼠标右键，选择 Custom Probe 下以 Conditional 开头的探针，打开探针上的 Condition 页，就可以设置条件了。此时，若被探测的数据满足你所设置的条件，程序就会暂停。

5. 用户自定义探针

如果你觉得 LabVIEW 自带的探针功能还不够强大，或者你自己创建了一种数据类型，而 LabVIEW 没有适合它的探针，这时你可以自己创造一个满意的探针出来。

用户自定义的探针其实也是一个 VI。LabVIEW 自带了一些已经做好的探针，这些探针都被放置在 <lvdir>\vi.lib_probes 文件夹下。你可以打开这里面的 VI 看一看已有的自定义探针是如何做的。比如我们在图4 中所使用的 I32 型条件探针的 VI 是

ConditionalSigned32.vi。

需要新建一个自定义探针时，先在数据线上点击鼠标右键，选择 Custom Probe -> New。这时 LabVIEW 会弹出一个向导界面。按照向导的提示，输入所需信息，LabVIEW 会为你生成一个用作探针的 VI 框架，对这个 VI 稍作修改，即可成为一个新的探针。

这个探针 VI 有一个输入和一个输出。输入的是被探测的数据，输出是一个布尔类型，表示程序是否需要暂停。这个 VI 的界面也就是探针的外观。探针所实现的功能完全依赖于如何对其编程。

其它常用调试工具和方法

除了断点和探针这两种最常用的调试工具外，我们也经常要借助一些其它的工具和方法来找到程序的问题所在。

1. 性能和内存查看工具（Profile Performance and Memory）

调试的目的并不一定仅要找出功能性错误，有时是要找到程序效率低下的原因，或者潜在危险，如内存泄漏等。这时就要用到 LabVIEW 的性能和内存查看工具了。

2. 显示缓存分配工具（Show Buffer Allocation）

显示缓存分配工具是另一检查 LabVIEW 代码内存分配情况的强大工具。

3. 程序框图禁用结构（Diagram Disable Structure）

调试首先要找到问题发生的部位。有时候，我们可以使用探针一路跟踪数据在程序执行过程中的变化。如果数据在某个节点的输出与预期的不一致，这个节点很可能就是问题所在。还有些情况，不是靠这种简单方法就可以找出问题的。比如程序中出现的数组越界的错误，在错误发生后，程序可能还会正常运行一段不确定的时间，然后崩溃，或报错。这种程序报错，或者崩溃的地方有可能在每次调试时都不同，或者找到了最终出错的代码，发现他是个最基本的 LabVIEW 节点，不能再根据去调试了，而这个节点出错的可能性基本为零，错误肯定是其他地方引起的。

调试这种问题，一般就是把一部分代码禁止掉，看看程序运行是否还有问题。如果没有问题了，说明有毛病的代码被禁止运行，则在把禁止代码的范围再缩小；如果问题又出现了，说明是刚刚被放出来的代码有毛病，则对这部分代码再禁掉一部份，继续调试。知道找出引起问题的一个或几个节点，改正它们。在这个仅用部分调试代码的过程中，使用程序框图禁用结构是最为方便的了，它就好象是 C 语言中用来做注释的关键符号“/*　*/”或者“//”。使用它可以方便的把一部分代码框住，禁用，如图1。

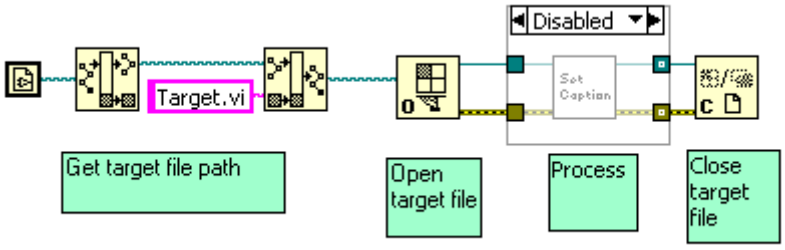


图1：程序框图禁用结构

使用程序框图禁用结构需要注意的一点是，这个结构可以有多个 Disable 的页面，同时会有一个 Enable 的页面。调试人员可能还要在 Enable 的页面作一些改动，比如为输出数据添加一些虚拟值，以使后续程序可以程序可以正确运行下去。例如图2，为了让后续的程序继续正确运行，需要把 reverence 和 error 数据线连接上。

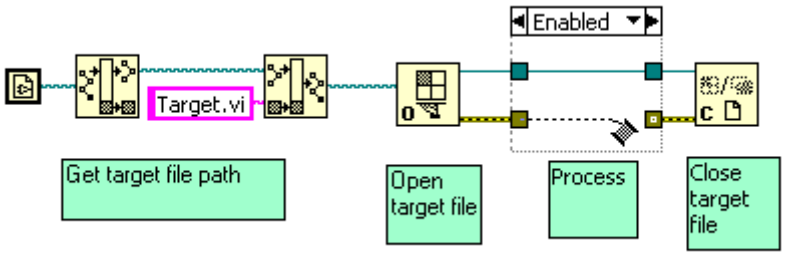


图2：修改 Enable 页面

4. 条件禁用结构（Conditional Disable Diagram）

LabVIEW 中还有一个类似于 C 语言中 #if，#ifdef 的结构，就是条件禁用结构。使用条件禁用结构可以让某些代码在特定的条件下不运行。与条件结构（Case Structrue）相区别，条件结构在运行时决定执行哪一个页面中的代码；而条件禁用结构是在编译时就已决定好执行哪一个页面的代码了，不被执行的页面的代码在运行时都不会被装入内存。

利用条件禁用结构的这一特性，可以把分别需要在调试时和发布后的代码放在不同的条件禁用结构页面内。这样，既可以在不同条件下运行不同的代码，有不会使程序留有冗余的代码。图3 的是一个条件禁用结构应用的典型例子，用户希望在开发调试时，如果错误数据线上出现错误，则探出错误信息的对话框；而在发布之后，又错误发生，也不可以弹出对话框。

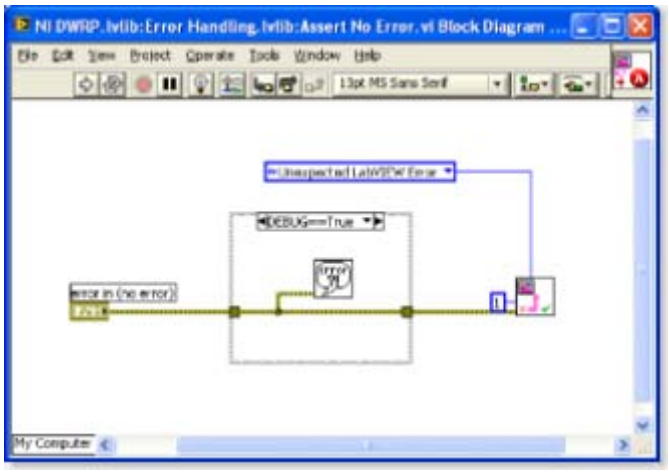


图3：使用条件禁用结构控制调试时和发布后程序的不同行为

点击条件禁用结构右键弹出菜单中的 Edit Condition For This Subdiagram... 条目可以弹出条件配置窗口，在这个窗口改变使本页运行的条件。LabVIEW 有一些预定义的符号（Symbol）可供条件禁用结构使用，比如 TARGET_TYPE 表示目标代码在什么系统下运行。如果条件是“TARGET_TYPE == Mac”表示目标代码运行在苹果机上。

如果你有工程文件“*.lvproj”，那么还可以在工程文件的属性->条件禁用符号栏下配置自己需要的符号。如图3中的例子，就是我自己在工程的属性对话框中添加了一个“DEBUG”符号，这样我就可以通过更改 DEBUG 符号的值来控制是否弹出程序的错误对话框。

5. 使用消息对话框和文件

有一些错误是在关闭了调试信息后才出现的，或者出错的代码部分不允许使用 LabVIEW 的调试环境。这时就要使用类似 C 语言中 printf() 的功能了。具体实现方法就是把可以的数据在程序中用 messagebox 显示出来，这样就可以跟踪察看程序是在哪一部分出错的。还可以把所有相关的数据都保存在一个状态记录文件中，察看这个记录文件，就可以找出可以的错误。

状态记录文件可以与第4节提到的条件禁用结构联合起来使用，设置一个调试开关，再调试运行方式下记录下所有状态信息；在正式发布后不再记录仪提高程序运行效率。

LabVIEW 代码中常见的错误

发现了程序的问题再回头去调试，在查找程序错误时就不可避免地要花大量时间。要调高开发效率，最好是在编写代码时就避免一些常见的低级错误，这样可以节约大量的调试时间。

有些编程错误差不多是每个 LabVIEW 程序员都曾遇到过的。在编写相关代码的时候，对这些问题多留心一下，就可以大大减少调试时间。

1. 数值溢出

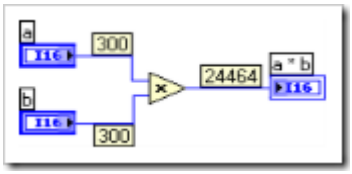


图1：数值溢出错误

图1 中的 VI 只做了一个简单乘法 300*300 ，不加思索就应该知道答案是 90000，但程序中乘法节点给出的结果却是 24464。乘法节点是不会错的，错误是由于程序中使用的数据类型是 I16。I16 能表示的最大数目只有32767，所以在乘法计算中出现了溢出。

避免此类错误的方法是，在程序中使用短数据类型时，一定要确认程序中的数据绝不会超出该类型可以表示的范围。

2. For 循环的隧道

循环相关的介绍可以参考《循环结构》。

数据传入传出循环结构可以通过移位寄存器（Shift Register）和隧道（Tunnel）两种方式。隧道又有两种类型：带索引的和不带索引的。

移位寄存器一般用在需要局部变量的情况下，循环运行一次的输出数据要作为下次运行的输入数据使用；循环外的数组数据通过带索引的隧道在循环体内就可以直接得到数组元素；除此之外，简单地在循环内外传递数据，使用一般的隧道就可以了。

值得一提的是，如果一个数据传入循环体，又传出来，那么就应该使用移位寄存器或带索引的隧道来传递这个数据，尽量不要使用不带索引的隧道。因为 For 循环在运行时，循环次数有可能为0。在循环次数为0时，大多数情况，用户还是希望传出循环的数据就是传入值，但使用不带索引隧道时，输入值有时会被丢失的。如果使用移位寄存器，即使循环次数为0，也不会丢失传入的数据。因为移位寄存器在循环上的两个接线柱指向的实际是同一块内存，而输入输出两个隧道指向的是不同的内存，数据不一定相同。

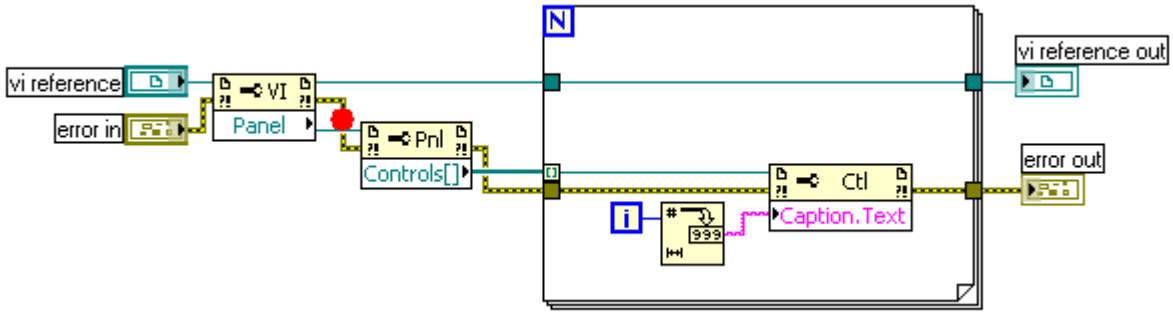


图2：For 循环上的隧道

图2中的程序，vi reference 传入，再传出循环均使用了隧道。如果循环次数为0（Controls 数组为空），vi reference 再传出循环时，信息就丢失了。这不但有可能造成后续程序的错误，而且由于 vi reference 的信息丢失，再无法关闭打开的 vi，造成了程序泄漏。

Error 数据线（黄绿色的粗线）在传入传出数组时，一定要使用移位寄存器。原因还不仅是为了防止在循环次数为0时，错误信息丢失。通常一个节点的 Error Out 有错误输出，意味着后续的程序都不应该执行。在错误的情况下继续执行程序代码，风险非常大，可能会引起程序，甚至系统崩溃。只有使用移位寄存器，某次循环产生的错误才会被传递到后续的循环中，从而及时阻止后续循环中的代码被运行。

3. 循环次数

与其它语言相比，LabVIEW 的 For 循环有一大特点，在某些情况下它并不要求一定要输入循环次数，而可以根据输入数组的大小自动决定循环次数。通过带索引的隧道，可以把数组分解成元素传递到循环体内，此时不需另行设置循环次数 N，循环的次数就是数组的长度。每次循环，带索引的隧道便给出一个元素。

循环体上可以有两个或更多的输入数组使用带索引的隧道，此种情况下容易引起错误。这时，循环的次数等于几个数组中长度最短的那个数组的长度。如果另外又设置了循环次数 N，那么循环次数就是 N 与输入数组长度这两者的最小值。调试时，如果发现一个本该运行多次的循环没有运行，那么很可能就是因为它的一个输入数组是空的。

While 循环同样也可以使用带索引的隧道，但是我不建议大家这么用——如果需要用到带索引的隧道，还是使用 For 循环更为适宜。因为 while 循环的循环次数不由数组个数决定，而是由停止条件决定的。如使用了带索引的隧道，你还需要考虑当数组大于、小于循环次数时，程序应该如何处理，所以还是在循环体内作索引比较方便。如果希望循环次数与数组大小保持一致，那自然是用 For 循环的程序更加清晰易懂了。

4. 移位寄存器的初始化

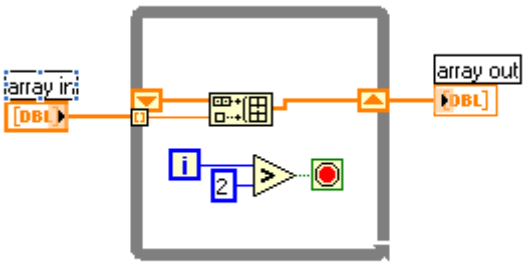


图3：没有初始化的移位寄存器

看图3中这个程序，因为它在 while 循环上使用了带索引的隧道，所以可读性不那么好。array out 的运行结果是什么，还要考虑一阵子才能给出答案。实际上这个程序，即使输入不变，每运行一次，array out 的结果都是不一样的，它的长度一直在增加。这个问题就出在没有给程序中的移位寄存器一个初始值。

没有初始化的移位寄存器，总是保存上次运行结束时的数据。这个特点在某些情况下可以被程序员利用，比如用它当作全局变量，随时把数据存入或取出（一个例子是《如何使用 VI 的重入属性》中的图4）。但多数情况下移位寄存器还是被用作为循环内部的局部变量的，这时就一定要对它初始化，以防止潜在的错误。

5. Cluster

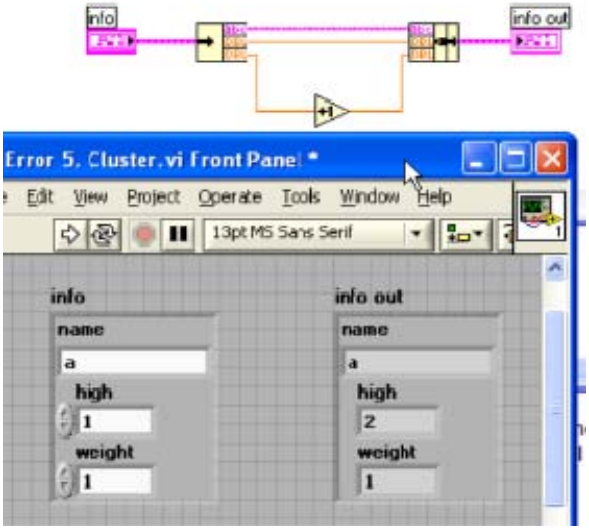


图4：Cluster 传递数据出错

图4的程序中有个奇怪的错误，明明应该是 weight 加 1 怎么运行完后的结果变成了 high 加 1 了呢？直接揭开谜底吧，原因是 Cluster 中的元素有个顺序，这个顺序可以和界面上看到的顺序不一致。分别鼠标右击程序中的两个 Cluster，选择“Reorder Controls in Cluster”，就可以看到每个元素在 cluster 中的编号。info out 中的 high 实际上编号是 2，第三个元素。

为了避免 cluster 中用可能出现的错误，以及让 cluster 应用起来更方便，使用 cluster 最好遵循以下原则：

1. 凡是用到 cluster 的地方，就为它造一个类型定义（《在程序中使用类型定义》），在程序所有要用到这个 cluster 的地方，都使用类型定义的实例。这样一是可以保证所有的 cluster 都完全一致，避免图4 这种错误；二是一旦需要变动 cluster 中的元素，只需在类型定义中更新就可以了，不必挨个 VI 修改。

2. 凡是在需要解开（unbundle）或打包（bundle）的地方统统使用 unbundle by name 和 bundle by name 来实现。使用带名字的 bundle，unbundle 可以直观的显示出 bundle 种元素的名字，这样不会因为顺序的不同而导致错误的连线。

6. 并行运行

LabVIEW 是自动多线程的编程语言，这一点在方便用户的同时，也会带来一些麻烦。比如最常见的情况，多线程会引起数据或资源的竞争错误（race condition）。

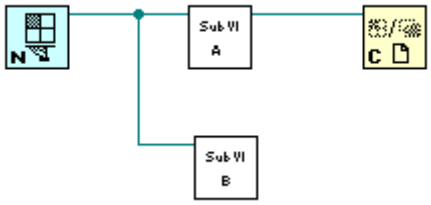
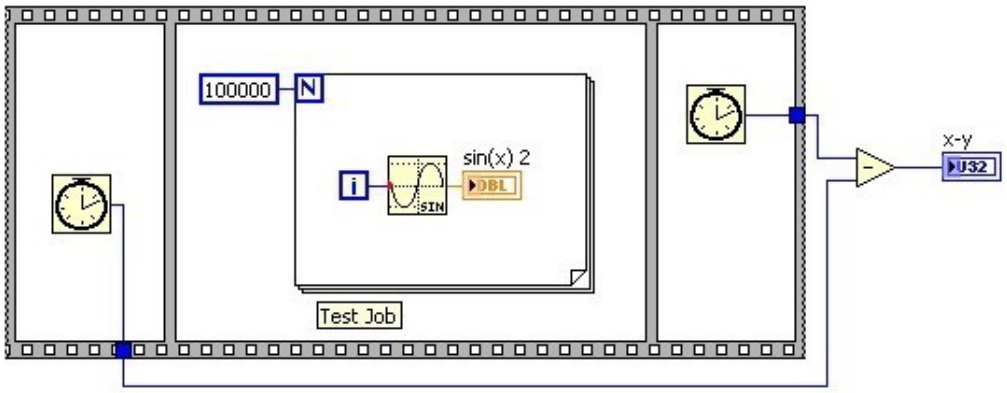


图5：两个并行运行的子 VI

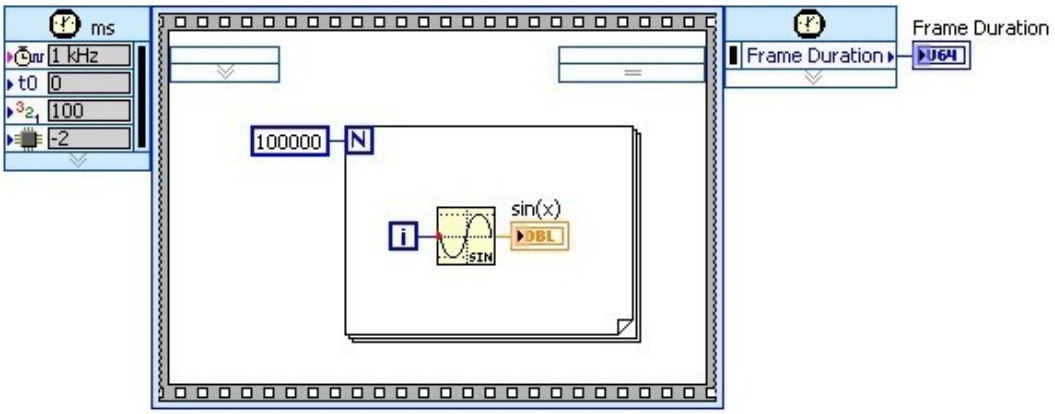
图5是一个简单的两个子 VI 并行运行的例子，在这个例子中就隐藏着一个潜在的问题。并行执行的两部分程序，先后次序是不定的。有可能关闭程序中的引用数据（绿色的线上的数据）的节点在子 VI B 结束前运行。而子 VI B 是要用到这个参考数据的，这是子 VI B 就会因为它所需要的数据失效而产生错误。

查看一段代码的运行时间

有时候调试或者测试一段程序的时候，需要查看这段程序到底运行了多少时间。这里的时间还不同于程序所消耗的 CPU 时间，而就是指程序运行一遍直观的耗时。一般，编写一段简单程序就可以完成这个工作了，如下图所示。使用一个顺序结构，在被测程序开始前，记录下当前时间，程序运行结束在查看一下当前时间，差值就是程序运行的时间。



不过还可以再懒惰一点，仅使用一个时间顺序结构也能完成同样的工作，如下图所示，Frame Duration 与上图的 x-y 含义相同。



着两种方法的最小精度都是1毫秒，更快的速度就测不了了。

如何调试 LabVIEW 调用的 DLL

问题(Frank)：

我用 Labwindow 编写了一个读文件的动态库, 即向动态库传递文件路径及文件名和某特定字符串,然后通个三个参数返回读到的值, 在 labVIEW 里调用该动态库,结果返回值老是显示打开文件失败, 不知错误出现在那里,另外在 LabVIEW 里如何调试确定传到动态库的参数是符合函数参数格式的呢?该函数在 Labwindow 里调试没有问题.请大侠指点迷津,不胜感激!

回答：

我好久没有用过 CVI 了，计算机上也没有装，不过用 CVI 来调试，应该和用 VC 来调试原理是相同的，步骤也想类似。我就以 VC 为例说明一下。首先在 Debug 模式下 build 出一个 DLL 来。 (VC 7.1 即便是 release 模式下也可以设置断点，单步运行，但别的编译器不一定行。)然后用这个新的 Debug DLL 覆盖原有的 DLL。

关闭 LabVIEW，点击 VC 菜单 Debug->Start (F5)。因为工程生成的是不可以直接执行的 DLL 文件，这是 VC 会弹出一个对话框，问你用什么运行。选择浏览，然后找到 LabVIEW.exe。(这个可执行文件也可以在工程属性中 Debugging->Command 一栏设置。)之后，VC 就会把 LabVIEW 调用起来。

在 VC 中设置好断点。在 LabVIEW 中运行想要调试的 VI。程序会停在你设置断点的地方。

怎样根据错误代码得到错误信息

大多数 VI 都会带有错误处理机制，所以 VI 的前面板上会有 error in/error out 控件。如果发现有返回错误代码，之间在空间边缘处点击鼠标右键，选择 Explain Error 就可以看到详细的错误信息。

在 Explain Error 对话框上改变错误代码，即可查看到任意一个错误代码的相关信息。也可以通过菜单 Help->Explain Error 打开这个对话框。

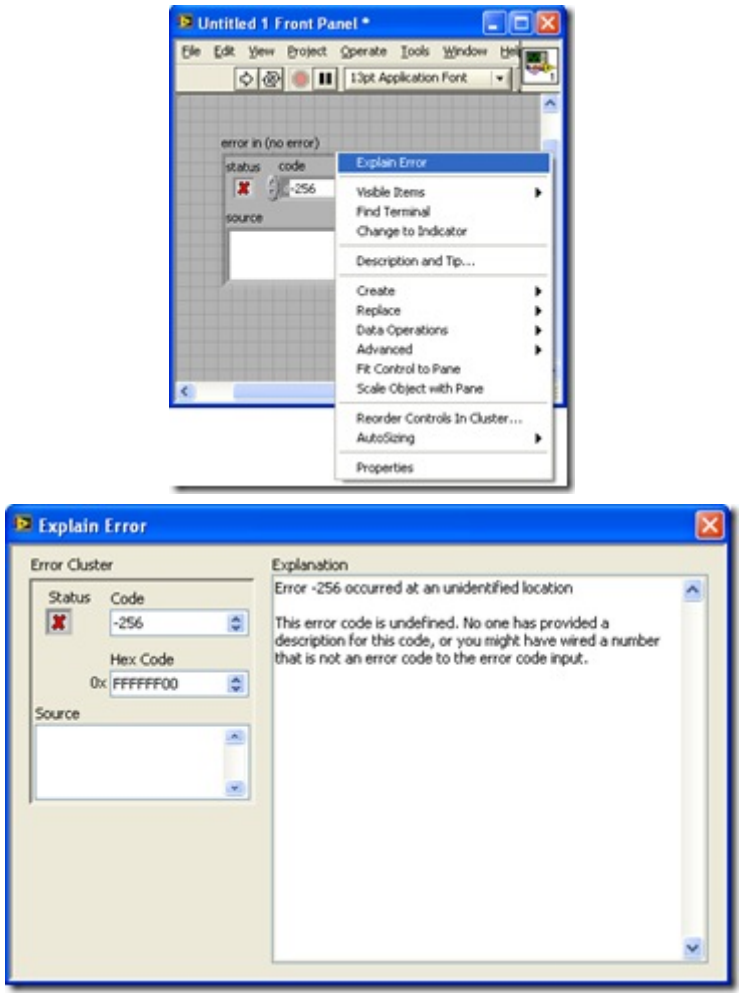


图1：使用 Explain Error 对话框

如果是在程序当中需要动态的得到一个错误代码的信息，可以使用[LabVIEW]\vi.lib\Utility\error.llb\Error Code Database.vi。这个 VI 可以根据输入的错误代码返回错误信息。

G 语言

G 语言是图形化编程语言(Graphical Programing Language)的缩写。LabVIEW 有的时候也被叫做 G 语言。我们可以这样理解：LabVIEW 是一个开发环境(类似的如 Visual Studio 也是一个开发环境)，在这个环境下编写的代码就是 G 语言代码(类似的如在 Visual Studio 下写出的 C 代码)。

目前在中国，很多工程师认为 LabVIEW 是一个应用在工业测控领域的应用软件，并不理解他是一个编程语言。原因有两个，首先是因为它和以往其它的编程语言差距太大，第一次看到它的人倒是更容易联想到电路板布线、工业总线配置软件等;其次是因为 LabVIEW 在中国使用的年头不多，大多数用户仅用到了 LabVIEW 的一小部分功能，还没有真正体验到 LabVIEW 的强大。

既然是一门编程语言，在使用 LabVIEW 的时候，就应该按照程序设计的思想来解决问题。举一个例子来说明如果用程序设计的思想来解决问题：

我们需要解决的问题是求两个正整数的最大公约数，这是一个非常常见的编程例子。

用 LabVIEW 来解决这个问题，应当与用其他语言求解这个问体的思路是一致的。按照程序设计的一般方法，解决这个问题可以三个步骤：

第一：确定问题的需求，给出需求的详细说明。对于这个求最大公约数的问题，我们在这一步需要做的就是写出程序输入输出的详细定义。如果是用普通的文本语言编程，你至少应该以文档的方式吧问题需求记录下来。但是 LabVIEW 程序员在这一步有个更方便的设计方法——直接在 VI 的前面板上定义程序输入输出：程序需要两个输入值(a, b)，用 Numeric control 代表，一个输出(x)用 indicator 代表。输入要求是正整数，我们可以把 Numeric control 的数据类型设置为 U32，并在这个控件的属性中设置最小值为1。再为 VI 和它每个控件添加上帮助信息，VI 的前面板就可以用户提供一个详细的 VI 的功能描述以及接口定义。

第二：设计解决问题的算法。一个问题通常不只会有一种解决方法(算法)，比如说我们的求最大公约数问题，你可以采用穷举算法，把1到 a 之间所有的整数都试一遍，然后找到那个最大的公约数;也可以使用 g.c.d.算法。

多数情况下这一步骤和具体的语言环境无关，比如说我们的问题不论采用哪种语言编写，g.c.d.算法的效率都高于穷举法。但是某些时候可能要考虑 G 语言不同与文本语言的特性，在 G 语言下使用不同于其他语言的算法。比如要遍历一棵树，可以使用递归的算法，也可以使用循环的算法。在 C 语言下，一般会选择递归的算法，因为递归算法的思维方式更自然，更容易掌握，实现起来也比较方便;但是在 G 语言下，递归的实现并不那么容易，效率也比较低，所以在 G 语言中，选择循环的算法更加适合。

对于我们要解决求最大公约数问题，我们还是选用 g.c.d.算法，它的运算过程如下：

Step1: If (a mod b == 0) goto Step3; else goto Step2;

Step2: (a, b) = (b, a mod b); goto Step1;

Step3: x=a; return;

第三：在 LabVIEW 下实现设计好的算法。G 语言之所以被称之为图形化的编程语言，并不仅仅是因

为它的程序又图形化的界面(前面板)，最本质的原因是因为它的代码也是通过画图的方式来编写的(程序框图)。

针对本例，可以使用 while 循环，a 和 b 分别用循环上的两对移位寄存器表示。在循环体内首先判断 a 是否被 b 整除，如果是，结束循环;否则把 b 和 a mod b 赋给两个移位寄存器，进入下一次循环。

图形化编程语言是数据流驱动(以后再解释)的，与一般文本编程语言的过程驱动机制有很大差别，因而在程序设计的思路上也与文本编程语言有所区别。尤其是有过文本编程经验的程序员开始使用 LabVIEW 的时候，会感觉 LabVIEW 缺失了很多文本语言常用的功能，比如使用局部变量、跳出循环等等，因而 LabVIEW 用起来不是太方便。另外 LabVIEW 编写出来的的代码连线乱七八糟，造成程序阅读和维护的困难。不过这些问题其实不能算是 LabVIEW 本身的问题，主要是由于编程者还没有掌握 G 语言的编程思想造成的。

LabVIEW 虽然不能覆盖所有文本语言的优点，但它具有自己的特色。在编写与工业领域设计、测量、控制等相关的程序或系统时，其开发效率大大高于其它语言。

在 LabVIEW 中可以为代码添加图文并茂的注释，再加上人类对图形的识别速度远远超过对文本的分析速度，一个优秀程序员编写的 G 语言代码的可读性要高于文本语言一个层次。

如何创建和使用 LabVIEW 中的 LLB 文件

最近接连有人问我，怎样在 LabVIEW 中创建一个 LLB 文件。于是我就把它写了下来。

通常，需要新建一个文件时，我们很自然就会想到去选中菜单“File->New”。可是，在 LabVIEW 8 中，在“新建”菜单中是看不到 LLB 这种文件类型的; 要创建或者管理一个 LLB 文件，首先要选择“Tool->LLB Manager...”。在打开 LLB Manager 之后，再在菜单中选择“File -> New LLB”。这样才能创建一个新的 LLB 文件出来。

我个人认为，在 LabVIEW8及以后的版本中，LLB 文件现在已经没有存在的必要了，使用它，弊大于利。

LLB 文件的功能就是把一组相关的 VI 以及其他文件打包存储在一起。其优点是节省磁盘空间，LLB 文件是压缩了的。但是，近年来计算机存储介质的容量迅速膨胀。LabVIEW 程序的存储空间再也不是一个需要考虑的问题了。所以这方面已经不再有诱惑力了。

LLB 文件有很多弊病。

1. 内部文件没有层次关系，所有文件都是平级存放的。这样一来，文件多了，就不能直接看出他们的调用关系。此外，LLB 也允许有同名文件存在。

2. 内部文件名长度有限制，大概限制几十个字符吧，文件名太长会被自动截断。

3. 不利于版本管理。LLB 中的一个文件被修改，整个 LLB 也就被修改了。这样，一是没办法作增量存储，二是不容易定位到具体被改动了的文件上。

综上所诉，如果新建一个工程，最好不要考虑使用 LLB 文件了。同时为了方便管理工程中的文件，应当尽量利用 LabVIEW 8 的新功能 ： Project 和 Library。

LabVIEW 是编译型语言还是解释型语言

LabVIEW 和常用的 VC++、VB 一样，是编译型语言。LabVIEW 的语法定义比较严格，在程序运行之前会检查所有语句的语法，一旦查出有差错，程序会报错，不能运行。

在 LabVIEW 是否是编译型语言的问题上容易引起混淆的原因，一是用户看不到编译时生成的目标文件(在 LabVIEW 的环境中，可以直接运行一个 VI，并不生成任何其他可执行文件);二是 LabVIEW 没有编译这个按钮。此外，VI 运行前似乎也没有占用编译时间。

我们可以把 LabVIEW 和 C 语言的存储与编译方法作一比较：C 语言的原文件存储在 .c 文件中。需要编译时，要显式地告知编译器进行编译。在耗费一段编译时间后，可以看到编译后生成的含有可执行二进制代码的 .obj 文件。而 LabVIEW 的原代码是存储在 .vi 文件中的。

一个 .c 文件中通常保存了多个函数，一个由几十个函数构成的 C 语言工程，也许只由两三个 .c 文件组成。而通常情况下，一个 .vi 文件只存储一个 VI，即相当于 C 语言中的一个函数。所以，一个小型 LabVIEW 工程也可能由几十个 .vi 文件组成。

但在某些情况下，一个 .vi 文件也可能包含了某些子 VI(子函数)，即这些子函数没有他们自己的 .vi 文件。这样的子 VI 被称为实例 VI(Instance VI)。LabVIEW 7版本中出现的、目前很常用的 Express VI 就是这种 Instance VI。他们都是被存储在调用他们的 VI 中的。

.c 文件只保存程序的原代码;而 .vi 文件不仅保存了 LabVIEW 程序的原代码，也保存了程序编译之后生成的目标代码。在 LabVIEW 的工程中看不到类似 .obj 这样的文件，就是因为编译后的代码也已经被保存在了 .vi 中的缘故。

LabVIEW 在运行 VI 之前无需编译，是因为 LabVIEW 在把 VI 装入内存的时候、以及在编辑 VI 的同时进行了编译。

当把一个 VI 装入内存时，LabVIEW 先要判断一下这个 VI 是否需要被编译。一般情况下，如果不对 VI 的代码做改动，是不需要重新编译的。但是在两种情况下需要重新编译。第一种，是在高版本 LabVIEW 中打开一个用低版本 LabVIEW 保存的 VI;第二种，是在不同的操作系统下装入和打开了同一个 VI。

比如，要在 LabVIEW 8.0 中打开一个原来用 LabVIEW 7.0 编写保存的 VI，则被装入的 VI 需要被重新编译，因为不同版本的 LabVIEW 生成的目标代码会稍有不同。如果你的工程包含有上百个 VI，在新版本的 LabVIEW 中打开顶层 VI，就会明显地察觉到编译所占用的时间。第二种情况的例子是，在 Linux 中打开一个原来是在 Windows XP 下编写保存的 VI，LabVIEW 也需要重新编译。LabVIEW 为不同操作系统生成的目标代码也是不同的。

在以上两种情况下，打开一个 VI 后，会发现 VI 窗口的标题栏中的标题后面出现一个星号，这表示需要重新保存 VI。此时，虽然 VI 中的程序原代码没有改变，但是编译生成的目标代码已经变了，所以需要重新保存。

在 LabVIEW 安装了升级补丁之后(比如从8.0升级到8.01)，程序会提示你是否需要把 LabVIEW 自带的 VI 全部批量编译(mass compile)。如果你选择“是”，则可能需要占用几个小时的时间才能完成编译。

LabVIEW 在你编辑程序原代码的同时，就会对它进行编译。LabVIEW 只编译你当前正在编辑的这个

VI，它的子 VI 已经保存有已编译好的目标代码，所以不需要重新编译了。因为每个 .vi 只相当于一个函数，代码量不会很大，编译速度就相当快，用户基本上是察觉不到的。 你在编写一个 LabVIEW 程序时，假如你把两个类型不同的接线端联在一起，会看到程序的运行按钮立即断裂，它表示程序已经编译了，并且编译后的代码不可执行。程序编写完毕，所有 VI 也都已是被编译好了，程序直接运行即可。

有时会出现这种情况：打开一个 VI，VI 左上方运行按钮上的箭头是断裂的，表示 VI 不能运行。但是点击断裂的箭头，在错误列表里却没有列出任何错误信息。此时箭头断裂是由于 VI 保存的编译后的代码不能执行引起的。例如在上一次打开这个 VI 时，有一个被此 VI 调用的 DLL 文件没有找到，编译后的代码自然不能执行。而后关闭 VI 再把缺失的 DLL 文件放回去。下次打开始 VI 时，理论上 VI 应当可以运行了，但是这时 LabVIEW 没有重新编译这个 VI，VI 中保存的是上一次不可执行的代码，所以运行按钮的箭头仍然断裂。而程序原代码没有任何错误，所以错误列表中什么都看不到。

修复箭头状态的方法是按住 Ctrl + Shift 键，再用鼠标左键点击运行按钮(断裂的箭头)。在 LabVIEW 中按住 Ctrl + Shift 键 + 鼠标左键点击运行按钮表示编译，但不运行，这相当于其他语言的 Compile 按钮。

LabVIEW 采用的把可执行代码与源程序保存在同一文件，分散编译的方式，与其它语言相比是相当特殊的。它既有优点也有缺点。

它最大的缺点是不利于代码管理。比较正规的做法，程序代码需要每天都上传至代码管理服务器。因此，源代码管理需要占用大量的硬盘空间。如果只是程序代码还好，把编译好的执行代码也存在同一个文件里，这就大大加重了代码管理的负担。程序开发的时候，经常需要回头查看过去的修改历史。如果某个文件发生了变化，代码管理软件就会意识到这是代码作了修改。但是 VI 中有时只是它包含的执行代码发生的变化，因此代码管理软件无法正确的判断出是否代码有变化。

它的优点主要有两条：1. 运行子 VI 极为方便。其它语言要运行，只能从主入口进入，不能够单独运行某一个函数。而 LabVIEW 则可以直接运行任何一个 VI;2. 分散了编译时间。大型的 C++程序，编译起来很花时间，有时要用几天。LabVIEW 把编译时间分散到了写代码的同时，因此用户基本感觉不到 LabVIEW 编译占用的时间。

数据流驱动的编程语言

在面向对象的编程思想出现以前，文本编程语言主要采用的是面向过程的编程方法。面向过程有时候也被称为控制流驱动的编程方法。想我以前编程序，经常要先设计一个流程图，然后，照着流程图翻译成 C 代码就行了。

LabVIEW 程序是数据流驱动的，这与面向过程的程序还是比较相似的，但是也有一些区别。

面向过程的程序执行起来，就只有一条线，代码按照设计好的顺序一条一条执行下去。代码中的某一条语句，即便它的输入条件都已经被满足，它也要等到它前面的代码都被执行完后，才能被运行。

数据有时候不仅是在一条线上流动：数据线可能有分叉。而一个程序上也可能同时有多个数据在不同的线上流动。程序可以被扩展成一张网(有时候 LabVIEW 程序的框图线连得乱七八糟，就像一张网:)。一个节点运行完，数据从这个节点输出，会同时被传给所有用到它的其它节点去。一个节点只要它所有的输入都已经准备好了，就会被执行，不需要等待其它节点执行完。这样一来，经常有多个节点同时运行着的，LabVIEW 会自动把他们放到不同的线程中去运行。这就是数据流驱动的程序的一大特性：是自动多线程运行的。一般的文本编程语言，除非有显示的调用开辟新线程函数，否则所有代码都在同一个线程内顺序执行。

自动多线程，为编程人员带来的不少方便。但是，由于多线程程序更为复杂，可能导致出错的隐患更多，LabVIEW 不得不做一些其它语言不需要做的工作，来保证用户可以方便的用 LabVIEW 开发出安全高效的程序。

多线程程序中最常见的问题就是多个线程访问同一资源或内存时发生冲突。先以内存中的数据为例，程序运行在单线程状态下，写进这块内存的数据是什么，下次读出来一定就是你写进去的。而多线程状态下就不一定了，说不定在读写之间，内存被别的线程修改了，读出来的数据就是错误的。一般的文本语言不需要编译器来考虑如何防止用户做出类似的错误操作，因为他们默认情况下只会使用一个线程。多线程一定是在用户有意识开辟的。既然是有意识开辟的，用户在使用多线程的时候就也会留心类似的不安全问题。LabVIEW 却不能像其它语言编译器一样，不去考虑这个问题，LabVIEW 用户会在无意识的状态下就编写出多线程的程序来。如果用 LabVIEW 写出来的程序总是出错，LabVIEW 就会渐渐失去客户。

LabVIEW 采取的保护措施之一就是它的传参方式。

传值和传引用

在现在常用的文本编程语言(C++, Java, C#)中，调用子函数时的传参方式主要是传引用方式，就是说，告诉被调用的函数的是参数所在的位置，而不是参数的数据。C++ 为了保持和 C 语言的兼容，一般的简单数据还是使用值传递，但对于大块的数据，比如数组，字符串，结构，类等等，也基本上都是引用形式传递的。

值传递的方式的缺点是显而易见的：每次调用子函数的时候，需要把数据拷贝一份，耗费大量的内存。传引用的方式，不需要每次都拷贝数据，节省了内存空间，和复制数据的时间。但是传引用的安全性不如直接传值，因为传引用的时候，数据所在的内存也可以被其它函数访问，这在单线程下，问题不大。但是多线程下，就不能保证数据的安全了。

理论上，一个数据流驱动的编程语言，可以只采用值传递。数据在每一个联线分叉的地方，都做一个拷贝。这样任何一个节点所处理的数据都是它专用的，不需要担心线程之间会相互影响。在设计 LabVIEW 程序时，可以假设 LabVIEW 就是这样子工作的。但是 LabVIEW 的实际工作情况比这要复杂些，它在不违背数据流原则的前提下，做了一些优化以避免过多的复制数据。

在某些时候，一个节点得到了输入数据，LabVIEW 如果能够确认这个输入数据的内存肯定不会被其他部分的程序代码使用到，并且恰好节点的一个输出需要一块内存，LabVIEW 就不在为输出数据令开辟一块内存了，而是使用那个输入数据所在的内存。这叫做缓存重用。

这种行为实质上和传引用是一样的，告诉函数一个数据的地址，然后函数直接在这个地址上处理数据。LabVIEW 程序员是不能够直接设置某个参数是传值还是传引用的。到底采用那种传递方式，是由 LabVIEW 来决定的。LabVIEW 决定采用哪种参数传递方式的原则是：首先保证数据的安全，其次才估计效率。LabVIEW 并不能总是准确的判断出某段代码采用传引用的方式是否安全。LabVIEW 本着宁枉勿纵的原则，对凡是拿不准的地方一律不优化，全部采用传值的方式，多拷贝一份数据。

虽然不能够直接设置某个参数是传值还是传引用的，但追求效率的的程序员，可以通过改变程序风格，来帮助 LabVIEW 准确判断出那些代码可以优化，无需拷贝数据，从而让自己编写出来的 LabVIEW 代码效率最高。比如，使用移位寄存器和缓存重用结构告诉 LabVIEW 在某个地方使用传引用的方式。

LabVIEW 中有些节点的输入输出数据类型完全不一样，比如数组索引节点，输入是一个数组和索引，输出是一个数组的元素。输入和输出的数据类型一般情况下完全不同，所以必须未输出数据新开辟一块内存，根本不可能做到缓存重用。有些节点，总是有相同类型的输入输出，比如加法节点，输出值的数据类型总是和其中一个输入同类型(fixed-point 数据类型是个例外)。LabVIEW 要考虑尽量在这些节点使用缓存重用。

如果输入值是数组数据，它通过分叉的连线被同时输入到一个数组索引节点和一个加法节点。假设其它数据都已就绪，LabVIEW 作为数据流驱动的程序，理论上应该同时运行着两个节点。但实际上，为了内存优化，在类似的情况下，LabVIEW 总是运行不可能做缓存重用的节点(比如这里是数组索引节点)，然后再运行可以做缓存重用的节点(加法节点)。

原因是这样的：如果先运行加法节点或者同时运行两个节点，因为加法节点的输入数据所在的内存还要被数组索引节点读取，因而加法节点是不能够改变这块内存中的数据的，那么加法节点只好再为输入数据开辟一块新内存;相反，如果先运行完数组索引节点，在运行加法节点的时候，加法节点输入数据所在的

内存就不会再被别的节点使用了，这是加法节点就可以放心的把输入数据放到这块内存里，做到缓存重用。

LabVIEW 虽然不能设置数据传递给一个节点时，使用值传递还是引用传递，但是 LabVIEW 中有一类专门的“引用型控件”，用来保证大块的数据不被频繁复制，或者在不同的线程内对同一内存做数据操作。一般叫做 xxx refnum 的控件都属于之一类，他们所代表的数据(也可用于表示某个设备)是不随着数据线流动的。程序上的连线出现分叉，虽然 refnum 这个值本身可能会被复制，但它所指向的数据是不会被拷贝的。

另外，如果一定要在 LabVIEW 代码中实现传引用，可以通过以下的方法：做一个全局数组变量，把数据存在数组里。VI 间传递的信息是数据在数组中的索引，一个表示序号整数值，就相当于是这块数据的引用。这样，所有对块数据的操作都是在同一内存中的，并且不同线程可以同时对这块内存做修改。

VI 中的数据空间

LabVIEW 由于比其它语言采用了更多的值传递方式，这必然会影响它的运行效率，也使得 LabVIEW 在这方面要采取一些其它语言不需要的应对措施，尽量提高效率。优化之一是子 VI 中局部变量使用的内存的分配方式。

C 语言中，函数的局部变量存在于栈中。在调用某一函数时，程序才为这个子函数开辟一块空间作为用于保存函数中局部变量的栈。子函数运行结束后，栈空间即被释放。下次再调用这个函数，程序会重新非配栈空间，这时的空间可能与上次分配的并不在同一内存地址。为了节约反复开辟空间的时间，LabVIEW VI 中并没有采用栈的方式。一般情况下，静态调用 VI，每个 VI 专门有一块存数据的数据空间，这块数据空间所在的内存地址在 VI 每次运行时是不会变化的，尤其是上次 VI 运行后所留有的数据还可以被使用。

LabVIEW 这种做法最大的好处是节约了大量开辟、回收内存的开销；但它也有个严重的缺陷，这也是其他语言不采用类似措施的原因：每次函数调用没有独立的数据区，因此无法实现递归调用（LabVIEW 静态调用的情况下）。经过权衡，LabVIEW 最终牺牲了递归来换取运行效率。

对于一般的子 VI（非可重入的），不论在程序的哪里被调用时，都使用的是同一块数据区。如果主 VI 上有两个并排被调用的同一个子 VI（如图1所示的两个 Delay VI），理论上的数据流驱动语言是应该在两个线程内同时运行两份子 VI 的代码。但是，由于这两次调用会使用到同一块数据区，为了避免两次运行之间互相干扰，引起数据混乱，LabVIEW 实际上是顺序执行这两次调用的。至于那部分代码被先调用是不确定的。

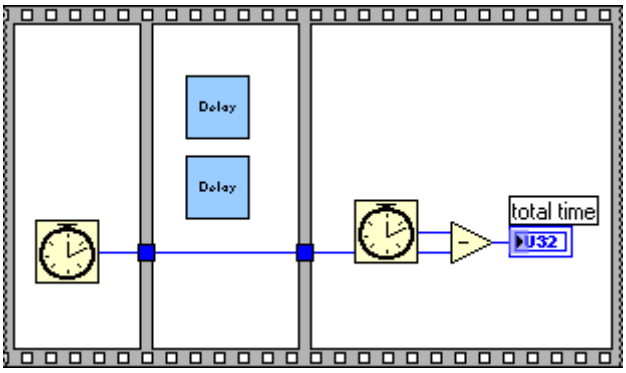


图1：并行调用同一子 VI 两次

LabVIEW 只能顺序执行这两次调用，在很多时候并不是一件坏事。比如，子 VI 中的操作是读写某一串口。LabVIEW 的这一特性恰好防止了多线程同时对这个串口读写而引发的错误。但这种行为也会引起一些糟糕的问题。比如，子 VI 是用来读写所有串口的。我在一个线程内对串口1做了操作，另一个线程要对串口2操作。读写串口是比较慢的，本来应该两个串口同时操作，来节约一点时间。但是如果串口读写子 VI 不能重入，那其中一个线程就只好慢慢等着了。

LabVIEW 解决这个问题的办法是为 VI 增加了一个可重入（reentrant）属性。非可重入的 VI 的数据区是和这个 VI 其它内容（比如执行代码、界面、源代码等）放在一起的，所以不论这个 VI 在哪被调用，使用的都是同一数据区。设置为可重入的 VI，它的数据区被开辟在调用它的父 VI 那里。在父 VI 的程序框图上每一个可重入子 VI 的图标，都意

味着父 VI 的空间内为这个 VI 开辟了一块数据区。所以，并行的两次调用同一可重入子 VI，这两次调用它们使用的是不同的数据区，所以可以同时运行而不需要担心数据被互相干扰；如果是循环内有一个子 VI，那么循环多次执行，每次调用这个子 VI，使用的还是同样的数据区。

用户界面设计 1

有些软件，一打开来就让人眼前一亮，可能是它的界面设计的非常新颖、华丽。但漂亮视觉感只能是作为锦上添花，评判一个界面好坏的最基本指标首先还是要看这个界面是否完成了它的交互功能-用户可以通过界面为程序提供必要的信息；用户可以通过界面接受到需要的信息。其次的指标是通过这个界面用户是否可以简单直观的输入或获取信息。最后才是界面的美观程度。

从这个角度说，一个好的界面，通常是不会引起用户注意的界面。多数时候，引起用户对界面的注意是因为他觉得别扭：找不到所需的信息，或输入信息的地方了。

使用 LabVIEW 开发一个项目，或编写一个软件，比较理想情况下是按照下面五个步骤顺序进行：收集需求、设计、编码、测试、发布及维护。细分设计阶段，一个项目所需的设计可能有用户界面设计，程序结构设计，接口设计，模块设计等等。对于编写 LabVIEW 程序，通常首先做用户界面设计。

先做界面设计可以使界面不受程序实现的影响。若先设计程序结构，再设计界面，难免会朝着最可能简化编码工作方向去做。但是这样的界面往往不是最方便用户使用的界面。

使用比较老的文本语言编程，设计用户界面时通常现在草稿纸上画出原型。LabVIEW 在这方面有独特的优势，它的可视化编程做的非常方便。有大量现成的控件，控件属性更改非常方便。因此，用户可以以拖拽的方式，直接用 LabVIEW 来设计界面原型。

对于界面好坏的评判，每个人都会有不同的观点。仁者见仁，智者见智。但是，好的用户界面都有一些共同的特点：一致性、使用恰当的数据类型和控件类型、控件的分类排布合理、简洁。我们在设计自己的程序界面时也要考虑到这些因素。

用户界面设计 2 - 界面的一致性

让用户迅速接受并且方便的操作一个程序界面，最关键的一点就是让这个界面保持高度的一致性。这里说的一致性包涵一下多个方面的一致：

一、程序内部的一致性

由于应用领域、面向的客户群体的不同，不同的软件可以有自己独特的风格。比如，为儿童设计的软件（例如使用乐高游戏版的 LabVIEW）几面可以加一些卡通图片，走可爱路线；为青年群体设计的软件，可以采用大量鲜艳颜色，显得活泼；LabVIEW 程序更多的时候是应用于工业领域，面向专业技术人员，这样的程序界面风格应当柔和、朴素。

不论一个程序采用了哪种风格，它内部不同界面（比如不同的对话框），同一面板上的不同控件等，它们的风格应当保持一直。一个软件采用统一的风格，才会让用户有一种“和谐”的感觉。

打开 LabVIEW 的控件选板，会发现有三种不同风格的控件：经典风格、现代风格、系统风格，如图1所示：

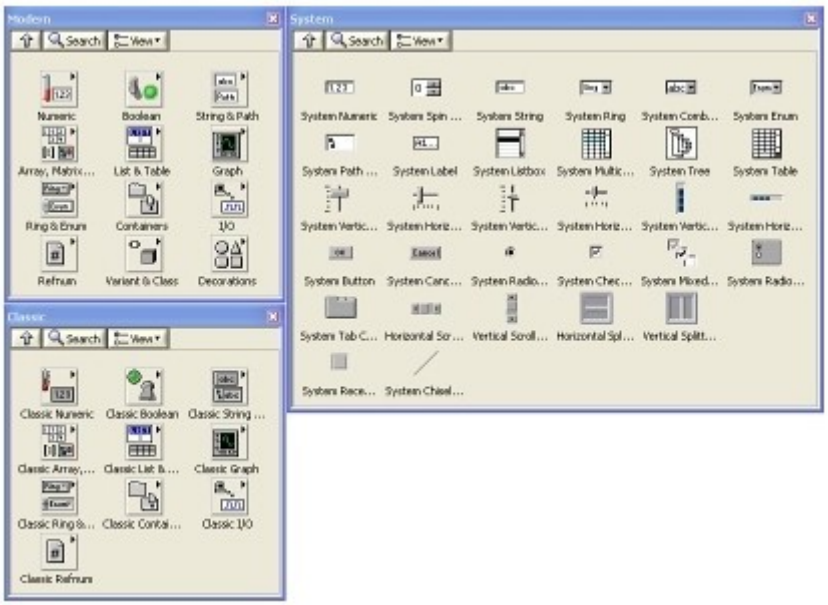


图1：三种不同风格的控件

经典风格的控件看上去比较土气，是 LabVIEW 6 之前的版本所使用的控件。一般不要用这种风格的控件了，有两种情况除外：

第一是维护老程序的时候，老程序可能还是用的这种控件，为了界面风格统一，又不想花时间改造原来的程序，那就继续用经典风格的控件。

第二是需要造一个透明控件的时候。这是一个小技巧，比如你希望有一段提示文字出现在界面上，需要使用字符串控件，但是你有希望文字直接出现在面板上，用户看不到包裹它的控件。这时候，就可以使用一个经典风格的字符串控件，然后用画笔把它的边框和背景都画为透明色即可。

LabVIEW 6 使用了一些重新设计的非常美观的立体效果控件，这就是现代风格的控件。编写测试领域的软件，可以首先考虑使用这类控件。

系统风格的控件外观与操作系统保持一致。我们编写的一般软件，希望用户比较易于接受时就可以使用这类控件。使用这类控件编写的界面，与系统自带的程序看上去风格非常一致。系统风格的控件会随着系统的不同，和系统设置的不同而随之调整。比如，把你的程序拷贝到 MacOS 的机器上，文本框会自然变成 MacOS 上圆弧角的风格。把系统颜色设为高亮反转显示，文本框也会变为黑底白字。

但是，LabVIEW 特有的控件，比如波形显示控件等，是没有系统风格的。如果你的程序整体式系统风格的，在使用这类控件时，要注意调整一下控件的颜色，使他们与其它控件的颜色保持一致。

二、与约定俗成的习惯保持一致

有很多设计或操作方法，已经被大家广为接受了。他们也许不见得美观或优化，但是一旦习惯养成了，就很难被改变了。据说我们现在使用的键盘，是当年为了延缓打字速度而精心设计出来的打字最慢的键盘排布方式。但现在大家都用习惯了，没人会为了打字快一些而换用其它按键排布方式。

与软件相关的比如，Ctrl+C 表示拷贝；Ctrl+V 表示粘贴。你如果用这两个键去干你认为更适合的工作，肯定会被用户骂死。在 LabWindows/CVI 中，查找的快捷键居然不是 Ctrl+F，搞得我只好不用它的快捷键。

对于应用程序界面，大家最习惯的就是 Windows 默认的界面风格了。简单来说，这样的界面就是：使用窗口，窗口最上方是标题栏，下面是菜单，再下面是工具条，再下面是主体内容，窗口最下方是状态栏，右面是滚动条。

如果你非要标新立异，把标题栏和滚动条的位置互换一下，那你的程序一定被用户骂死。不过，实力强大的公司也许会可以逐渐改变人们的习惯。微软今年推出的 Office 07 比以往的界面风格有了重大改变，也许是为了配合 Windows Vista。新的界面漂亮的不少，但它还是遭到了很多用户的抵制，就是因为在使用功能区（ribbon）替代了原来的菜单和工具栏之后，用户再也不能从熟悉的地方找到他们所需的操作了。

LabVIEW 默认的颜色配置和控件风格，与系统的风格也是有区别的。所以为了照顾新用户，不妨在程序里尽量使用系统风格的控件和颜色配置。

三、与真实事物保持一致

有很多程序是对现实世界的模拟或模仿，这样的程序若希望便于用户接受，最好是尽量与现实世界保持一致。比如电脑游戏，规则一定要与现实世界接近，若完全采用不同的规则，比如越练功人品越差、被人看几刀魅力值会增加等等，玩起来一定特别别扭。

LabVIEW 编写的程序大多与测量、控制等有关，在这些领域，原本也存在着一些相关的仪器或设备。因此软件的界面可以借鉴这些仪器的外观。比如需要实现的程序要完成一个类似示波器的功能，那么界面最好设计的和传统的示波器一样：一边是现实波形的控件，周围有调节垂直、水平方向范围的按钮等。这样，用户只要曾经用过示波器，不需要再学习任何知识，直接就可以使用你的软件了。

NI 公司开发的 Soft Front Panel 产品可以看作是与真实事物保持一致的一个很好范例。

四、建立并遵循界面规范

使界面保持一致性的最好办法就是在设计开发时遵循一定的规范。这个规范可以由公司内部定义，也可以遵循现有的行业规范。对于开发 Windows 系统风格的程序，可以遵循微软定义的界面规范。对于一般的 LabVIEW 程序，可以遵循 LabVIEW 程序开发规范。

用户界面设计 3 - 界面元素的关联

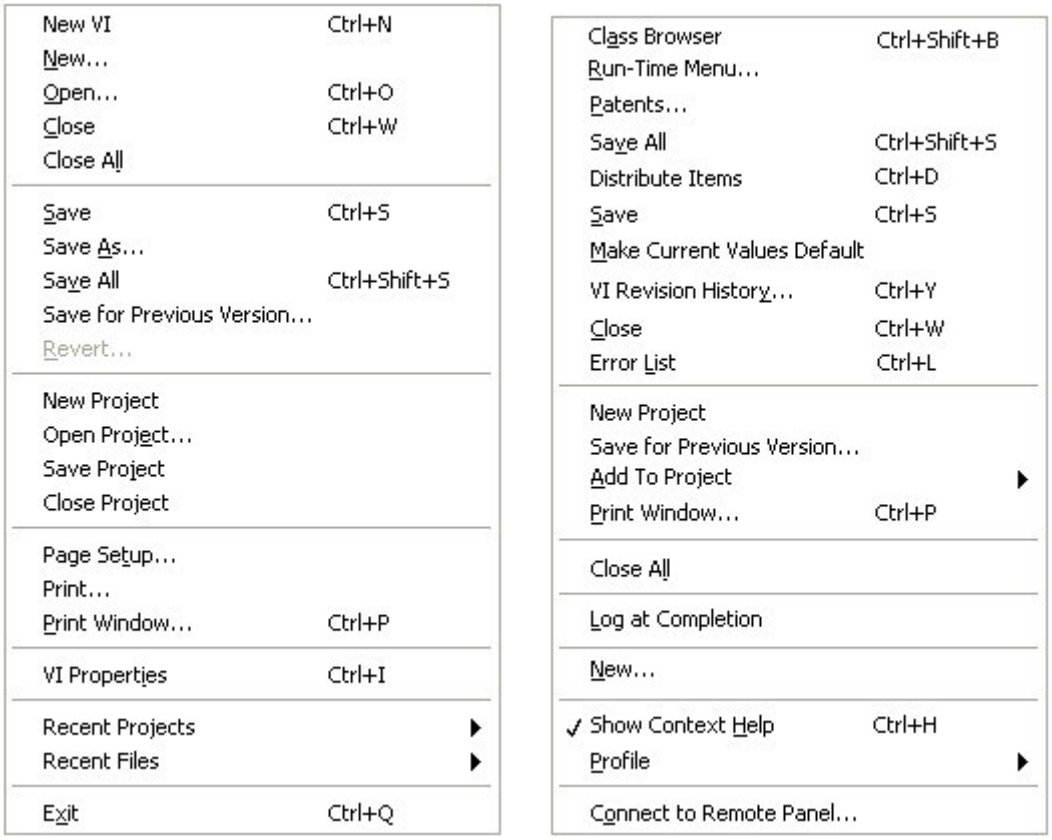


图1：两个菜单

上图左边是 LabVIEW 中的一个菜单。右边那个是我自己对它的“改进”。大家觉得那个好一些？

显然用户更喜欢组织清晰合理的那个菜单。当一个界面上的元素比较多，找到自己想要的信息就要花上一小点时间。用户常常是一眼就看到了一个与自己想要的信息有一点关联的某个元素，他这时候会期望这个元素就有一定的提示信息，帮他加速找到自己想要的东西。因此，我们要在界面上，告诉用户哪些元素是相关的，或不相关的。

有很多手段可以把界面的元素之间的关联显示给用户，比如通过元素的排布、边框、空白、颜色、字体等等方式。

我们总是在相关内容的附近去找想要的信息，所以逻辑上相关的控件或项目，应当在屏幕空间上相对临近。比如刚刚看到的菜单，Save, Save As, Save All，等等与保存相关的条目应当排在一块。

仅仅把相关内用摆在一起还不够，看看下面这个图片。

图2：小朋友的名字

这是我在网上看到的一个经典笑话：老师发作业本的时候，念小朋友们写在本子上的名字：“黄肚皮”，“鱼是虫”。但是没人答应，最后有两个小朋友没拿到本子，他们的名字分别是“黄月坡”和“鲁蛋”。小朋友们虽然把界面元素按照顺序排列了，但却没有合理的组合它们。

我们上面看到的菜单，有二十多个条目，单纯的把他们排在一起还是不利于用户查看。可以把它们按功能分成几个不同的区域，比如保存文件与 Project 的操作在功能上相对独立一些，就可以用分隔线，帮它们的项目划分开。对于面板上的控件，功能相关的几个控件可以通过被边框围住、使用分割线、采用不同的间隙等等方法，让用户直观的感觉到他们在功能上的紧密关联。

还有一种表示控件间关联性的方法值得多叙述几句，就是利用不同的颜色。球场上的两组队员，开始分列于球场两端，很容易区分他们是哪一伙的。而一旦比赛开始，这种空间上的提示就不存在了，这时大家主要靠队员衣服的颜色来区分它们属于哪支队伍。在界面设计上当然也可以使用这种方式，为不同功能的控件设置不同的颜色。

需要注意的是，颜色只能作为辅助方式，前几种方法不适用的时候，才需要用颜色来表示关联。颜色与前面提到的几种方式不同：大多数人喜欢排列整齐，布局合理的界面，但喜欢界面颜色丰富的人就不那么多了。相反，颜色艳丽、对比度高的界面会使人视觉疲劳，让人觉得反感。

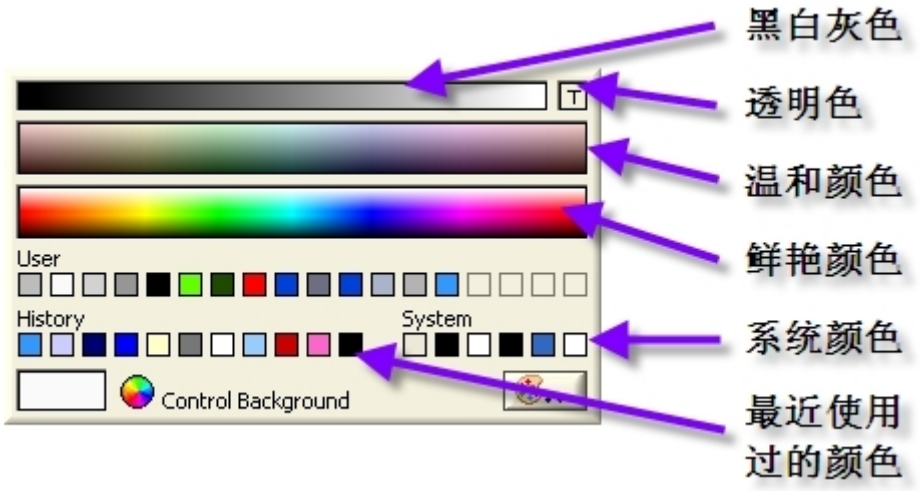


图3：LabVIEW 颜色配置

LabVIEW 配置颜色的面板上分了几类不同的颜色区域。设计系统风格的时候，需要使

用系统颜色。其他情况下，尽量使用柔和颜色，避免使用靓丽鲜艳的颜色。还要考虑到色盲、色弱的发病率也是蛮高的，界面设计时要照顾到这些用户。

所以，界面内容不多时，就尽量不要使用颜色了。只有当界面上信息量特别大的时候，颜色才会派上用场。需要使用图片的情况就不用说了，除此之外信息量较大的情况是有大量文字的时候。需要把不同的文字区分开来的时候，比如标注所有拼写错误的单词等等，就可以利用颜色来区分。当然，这时候也可以利用字体，字号等的不同来达到同样的目的。

界面设计中的很多原则，与艺术创作的原则一样，是以心理学中对人脑视觉认知的研究为理论依据的。心理学中与界面元素关联相关了一些理论可以参考《形状知觉中的分组》。

用户界面设计 4 - 帮助和反馈信息

用户界面要照顾到那些不熟悉它的用户。为了方便用户了解界面的使用方法，需要给用户提供足够的帮助信息。对于 LabVIEW，给用户提供提示主要通过以下几个手段：用户手册、在线帮助窗口、提示条、利用控件的标题、选项文字、直接把帮助文字写在界面上。

不论何时，都应该尽量使用有意义的控件名称。比如某一控件用来表示使用触发信号的上跳沿还是下跳沿作为触发条件。假若偷懒，不给这个控件起名，或者起个很简略的名称如“方式”，用户在看到这个控件时还是无法得知其确切用途。不如将名称写的详细一些，如图1所示的“边沿触发方式”，会更便于用户理解。

单有控件标题还不够完善。因为这个控件的输入值只有两种，可以使用布尔数值类型来表示。但是用户并不清楚“真”值在这里表示的是上跳沿还是下跳沿。因此，还应该把表示当前状态的布尔文本也显示出来。这样用户就可以行处的知道当前是什么方式。

图1：一个布尔型控件

提示条在一般程序中使用的非常广泛，把鼠标移到工具条的按钮上，就会出现一个黄色的小条提示这个按钮的用途。但是，由于 LabVIEW 有一个 Context Help 窗口，使用 LabVIEW 编写的程序大多数是把帮助信息写在这个在线帮助窗口里，就不需要再为界面添加提示条了。与提示条相比，在线帮助窗口面积比较大，可以写入更详细的信息，但是有时候可能影响整个界面的美观。

在线帮助窗口可以容纳的信息也还是有限的，对于十分难以掌握的界面，就只能把信息写到用户手册中去了。如果用户手册有相关内容一定要在界面上醒目的提示用户可以查看用户手册。在 LabVIEW 程序中，一般是把相关的链接写在在线帮助窗口中。

对于那些不经常被访问到的界面，比如用户可能一两年才使用某一配置界面一次，是不能指望用户记住界面上每个条目的意义的，而且这样的界面也不必做的太简洁。因此可以把帮助信息直接就写在界面上。

以 Import Shared Library 工具为例，下图是它的一页界面。在这个界面上其实用户真正需要选择的就只是“Error Handling Mode”这一项。界面下方的文字，是针对不同的错误处理模式的解释说明。用户选择不同的模式，给出的帮助文字也会相应调整。由于这个错误处理模式仅用文字描述还不够直观，因此界面中部还给出了图片作为帮助信息。

按 Ctrl+H 键，就会看到在线帮助窗口，里面有对每个界面元素的解释。如果觉得这还不够详细，可以点击界面右下方的 Help 按钮打开用户手册，阅读更为细致的解释。

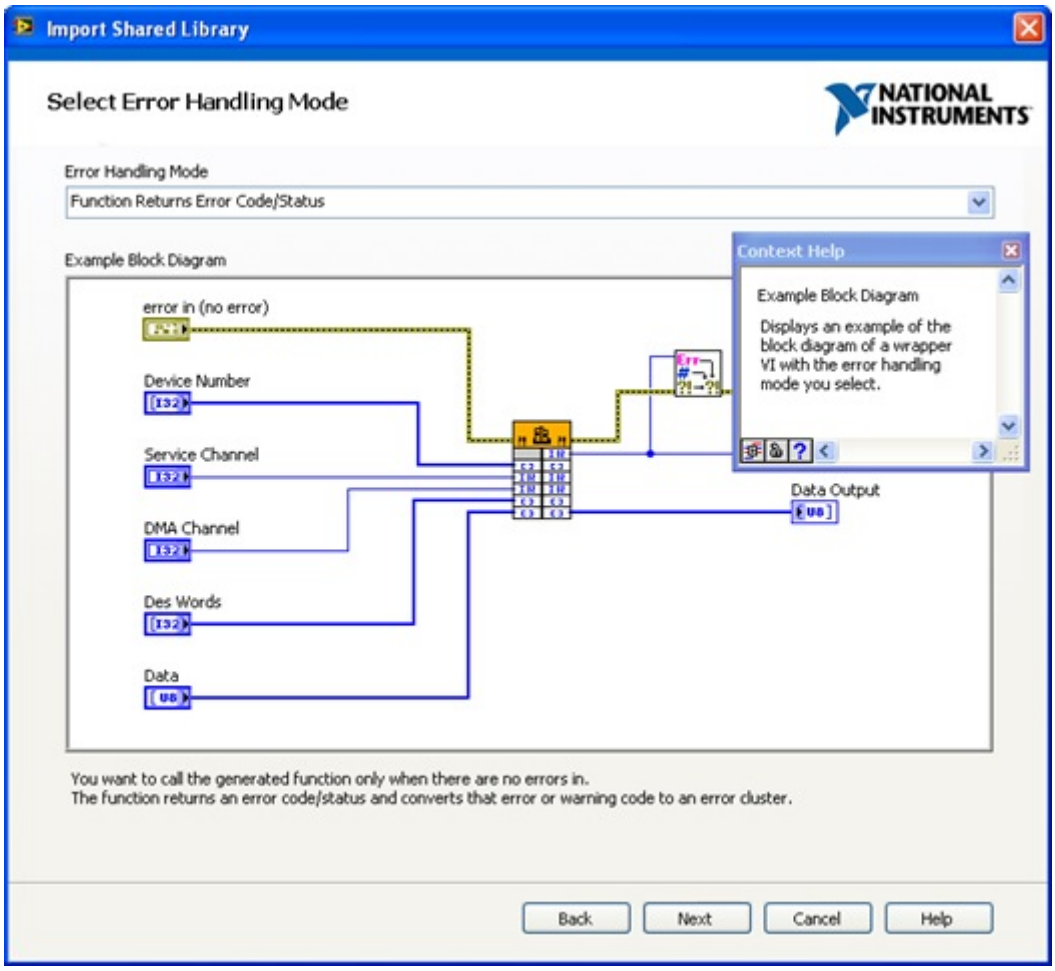


图2：Import Shared Library 的界面

保障软件的可靠性是软件开发者的责任。如果用户误操作，或者提供了错误的数据给程序，稳定的程序可以组织程序继续运行并报告错误。但这毕竟是亡羊补牢的做法，更完美的解决方案应倒是从根源上就杜绝误操作和错误的输入数据。

所以，在做界面设计时，还应考虑如何限制用户的输入数据和操作。禁止误操作出现，把输入数据都限制在合理的范围内。

一、限制输入数据

LabVIEW 的某些控件本身就带有对输入数据进行限制的功能。比如数值型控件，在它的属性对话框中的 Data Entry 页，可以设置这个控件接受的数据的范围。我有一个控件用来表示选取某个通道，可供使用的合法数据为通道0至通道3，我们就可以在这一页把控件的最大最小值分别设为3和0。如下图：

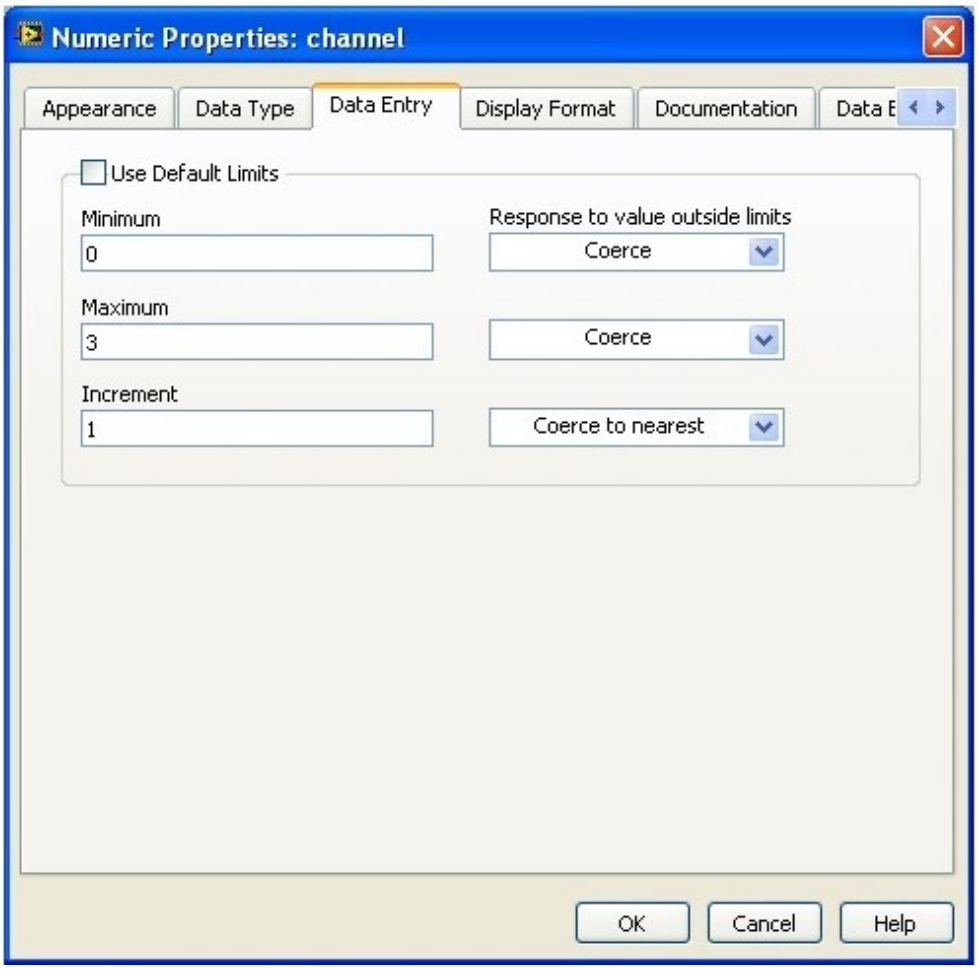


图1：数据范围限制

这样设置后，用户也许还会输入一个不合理的数值，比如99，但 LabVIEW 会立即忽略这个不合理数值。

有时，还有更好的限制方法：让用户根本没办法选择不合理的数据。比如本例，我们在设计时，可以考虑使用 Enum 或 Ring 型控件来表示通道号，这样用户只能在正确的值中选择一个。如下图：

图2：枚举型数据

除了 Enum 或 Ring 型控件，单选按钮也可以起到同样的效果。单选按钮可以直接就在界面上显示出所有可供选择的值，并且可以附带对每个选项的详细解释。不经常被用到的对话框可以采用这种控件。比如下图，是 VI 属性中设置密码的页面。

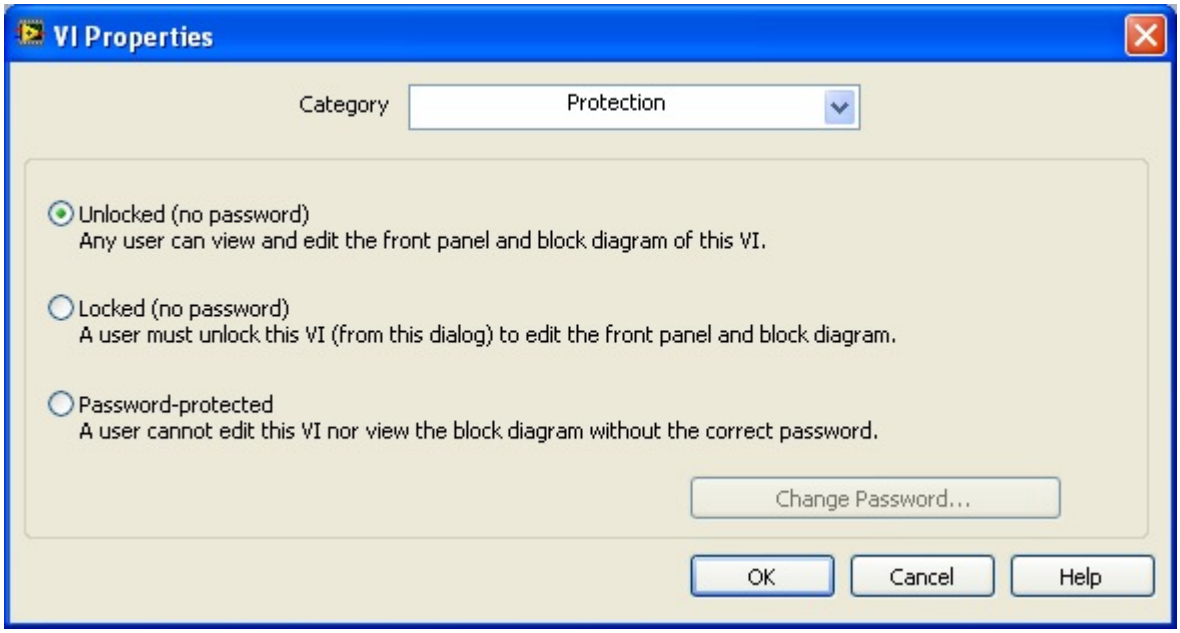


图3：使用选择按钮的界面

二、防止误操作

一个简单的规则：让所有但是不应被改动的控件都失效。大家看图3中的修改密码按钮是灰色的。因为这是用户选择的是无密码，所以当时不应出现修改密码的操作出现。与其让用户判断是否可以按这个按钮，不如直接禁止它的使用，以防用户错误的按下它发生不可预期的错误。当用户有密码设置后，再允许这个按键被使用。

Caption 和 Label 的书写规范

LabVIEW 控件的 Caption 和 Label 的特性和用途很相似，都是给了控件一个有意义的名字。因此，在很多场合没有必要刻意区分他们。

Caption 和 Label 的最主要区别在于，Caption 可以在程序运行的时候改变;而 Label 则不可以，一旦程序运行，就固定不变了。鉴于这一点，Caption 和 Label 的用途也略有区别。Label 应该是给程序自己用的，比如在程序中需要根据控件的名字找到它，那就得跟据 Label 来找，而不能用 Caption 来找;Caption 是为了给用户看的，有时控件的名字在运行到不同状态下需要发生改变，此时显示在界面上的就应该是 Caption。

推荐大家按照下面的规范使用 Caption 和 Label。

先给 VI 分一下类：

1. 底层 VI：用户不会直接使用到的 VI，作为 subVI 随程序一起发布。

2. 用户界面 VI：VI 前面板是给用户看的程序界面的一部分。

3. 程序接口 VI：VI 是提供给用户，在他们编程时，当作 API 被调用。

对于 Caption 和 Label 一个共同的书写规范是：使用有意义的文字，在使用英语短语命名时，单词之间用空格分隔，不应该有重名。

不同点列于下表：

	Label	Caption
底层 VI	显示出来	使用 LabVIEW 的默认状态，即 Caption 为空。
用户界面 VI	隐藏多语言版本中，只使用英语	显示多语言版本中，使用本地化语言
程序接口 VI	隐藏多语言版本中，只使用英语不用标注控件的默认值	显示多语言版本中，使用本地化语言在后面加一括号，括号内标注控件的默认值和数据单位

在 VI Properties -> Execution 中可以选择 VI 的 Reentrant Execution 属性(中文译为：可重入执行)。我们在《LabVIEW 程序的内存优化》一文中讨论过，尽量不要把 VI 设置为重入属性，因为这样就多占用了内存，降低了运行效率。此外，如果不加注意的话，还可能引发多线程不安全的问题。 尽管可重入 VI 在 LabVIEW 中不是必须的，但是在某些情况下使用可重入 VI 可以简化我们的程序。那么在什么情况下可以使用 Reentrant VI 呢?

首先看一下图 1 所示的程序，程序中调用的两个子 VI 是同一个 VI，并且不是可重入的 VI。LabVIEW 是自动多线程的语言，那么图中的两个子 VI 会不会同时执行呢。一定不会的。如果程序中调用的是两个不同的子 VI，LabVIEW 有可能会同时在不同的线程执行它们，但对于两次调用相同的子 VI，LabVIEW 一定要等一个执行完，再执行另一个。

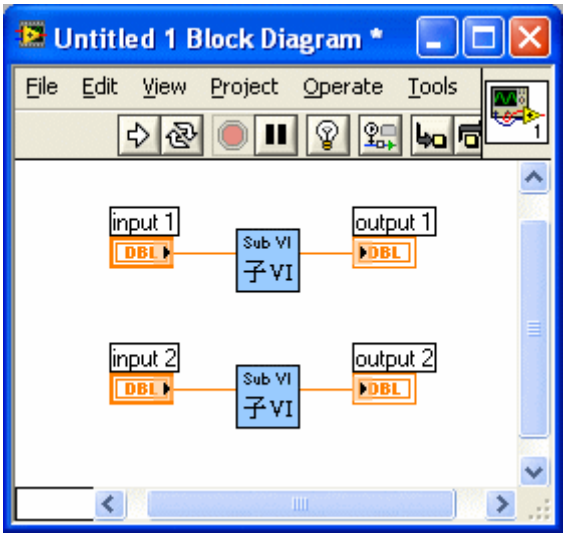


图1：并行的两个相同子 VI

其原因是，LabVIEW 会为每个 VI 都开辟一块内存用于数据存储。作为子 VI，每次被调用，它的局部变量的数据都是被存在同一地址的。与 C 语言相对照，在默认情况下，VI 是不可重入的，VI 中所有的局部变量都是静态变量。如果 LabVIEW 在不同的线程下执行同一 VI，那么两个线程就会同时对这一块数据地址进行读写，就会导致这一块地址内数据的混乱。为避免此类不安全情况的出现，LabVIEW 必须等待一个子 VI 执行结束，再执行另一个子 VI。

如果需要图1 中的两个子 VI 同时运行，比如子 VI 所做的工作是读取文件这样一类耗时多、但 CPU 占用不大的操作，则并行执行可以大大提高效率。这时，就需要把子 VI 设置为可重入了。LabVIEW 在不同的地方调用一个可重入 VI 时，会给它另外分配一个独立的数据地址空间。这样就做到了线程安全。在两个线程执行的子 VI 使用两份在不同的地址存储的数据，也就不会造成混乱。但是千万要注意， 这个“在不同的地方”调用：不可重入的 VI 的局部变量与 C 语言中非静态变量的含义是不同的。在后面提到的计数器的例子可以验证这一点。

我觉得我说得挺清楚了，出道题目给大家测试一下：

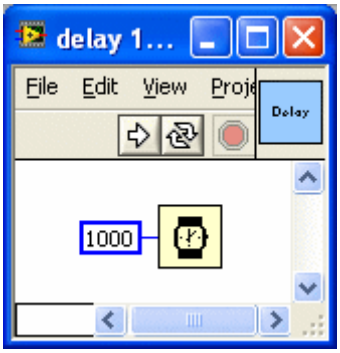


图2：延时子 VI

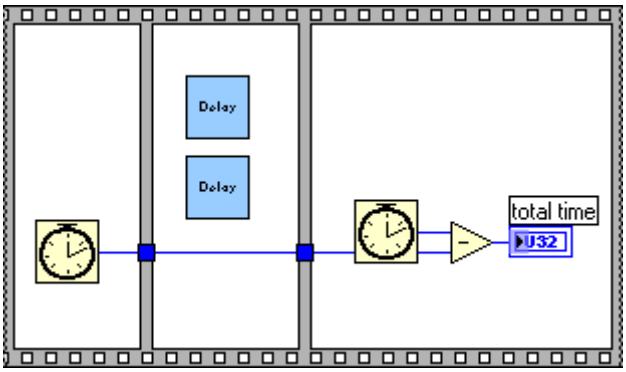


图3：计算延时的主 VI

图2 是一个子 VI 的代码，功能是延时 1000 毫秒。图3 是主 VI 的代码，并行调用同一子 VI 两次，并计算程序的执行时间。运行主 VI，total time 的值是多少?

答案在文章最后。

这是可重入 VI 的一种用途，即希望在不同的线程里同时执行同一个子 VI。

另外还有一种情况下，也可以用到可重入 VI：即需要使用到子 VI 中局部变量保存的数据，而在不同的调用处，这些数据是独立不同的。这句话可能解释得不那么清楚，看下面例子就会比较容易理解些。

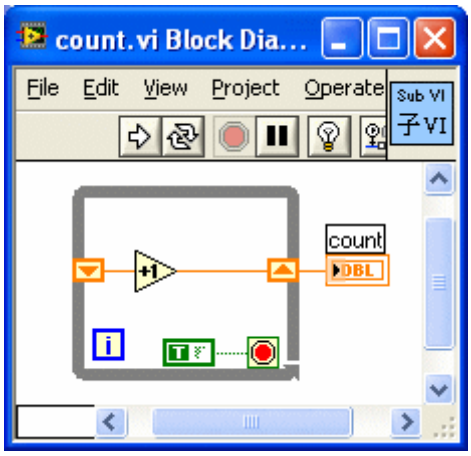


图4：计数子 VI

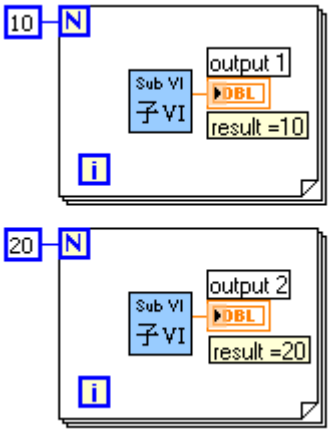


图5：测试计数的主 VI

图 4 是一个可重入子 VI 的代码，功能是计算这个 VI 被运行的次数，每运行一次，输出的 count 值就增加1。图5 是调用它的主 VI，用于演示这个计数器。执行主 VI 一次，output 1 和 output 2 的值分别是 10 和 20，表示这个子 VI 在两处分别被调用了 10 次和 20 次。

如果把图 4 中的 VI 改为不可重入，则 output 1 和 output 2 的输出值是不确定的。大家可以自己试一试，再想一下原因。

当使用递归结构时，参与了递归调用的 VI 是需要被同时调用多次的。因此这些 VI 中的变量必须是局部的，也就是说参与了递归调用的 VI 必须都被设置为可重入。参考：在 LabVIEW 中实现 VI 的递归调用

测试题目答案：如果图2的子 VI 没有设置为可重入，则 total time = 2000;如果设置为可重入则 total time = 1000。

美化程序 - 隐藏程序框图上的大个 Cluster

在编写某些程序的时候可能会遇到如图1 所示的情形：即用到了一个极为复杂的数据类型常量。 这个常量由于体积巨大，使得在程序框图无论怎么摆放都让人看起来不太舒服。如何才能把这个程序改造得美观一些呢？

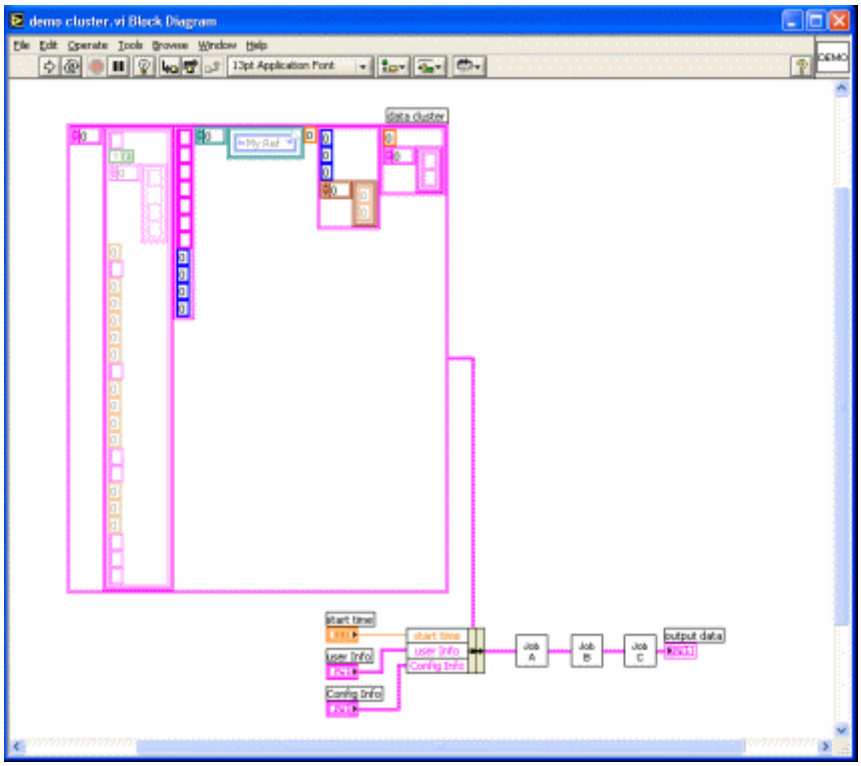


图1：体积巨大的常量会有碍观瞻

要解决这个问题，只有设法把这个常量在主程序框图上隐藏起来。通常可以用以下两种方法。

第一种方法：把这个常数变换成控件，再把控件隐藏起来。这种方法比较简单，但是也有弊病。①容易引起误解：控件一般表示有值传入，其他人读程序读到这里就可能搞不清楚这个值是从哪里传来的了；②如果要修改常量 Cluster 中某一个元素的值，操作起来比较麻烦。

第二种方法，也就是我向大家推荐的：把它隐藏到更深层的子 VI 中去。具体操作方法如下：

如图2 先给这个复杂数据类型建立一个 Strict Type Def。我的建议是为所有程序中用到的 Cluster 都建立一个 Strict Type Def。这样可以为以后的程序维护省去很多麻烦。

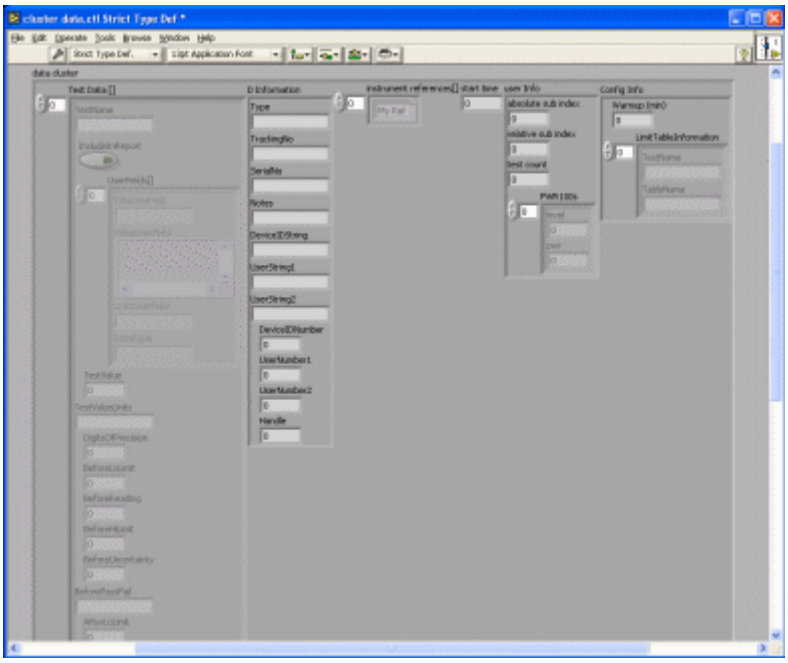


图2：Strict Type Def.

然后然后再建立一个新的 VI，把我们要隐藏的这个个头巨大的常量摆放在这个 VI 中，并且连接一个 Indicator ，以把它的值传出来。VI 的接线板采用 4-2-2-4 格式的，最下层第 3 个接线端用于传出 VI 中唯一的数据，如图3 所示。

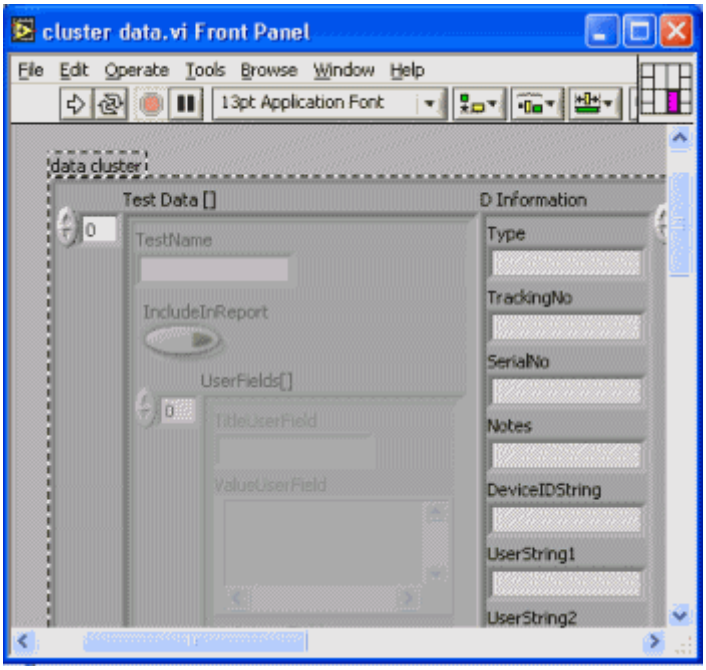


图3：用于隐藏个头巨大常量的 VI

这个 VI 的图标要做得小巧漂亮，如图4，图标不一定非要做成正方形。只要 B&W 和 256 Colors 中的图标形状一样，我们就可以画出不规则图标了。详细方法可以参考《制作不规则图形的子 VI 图标》。

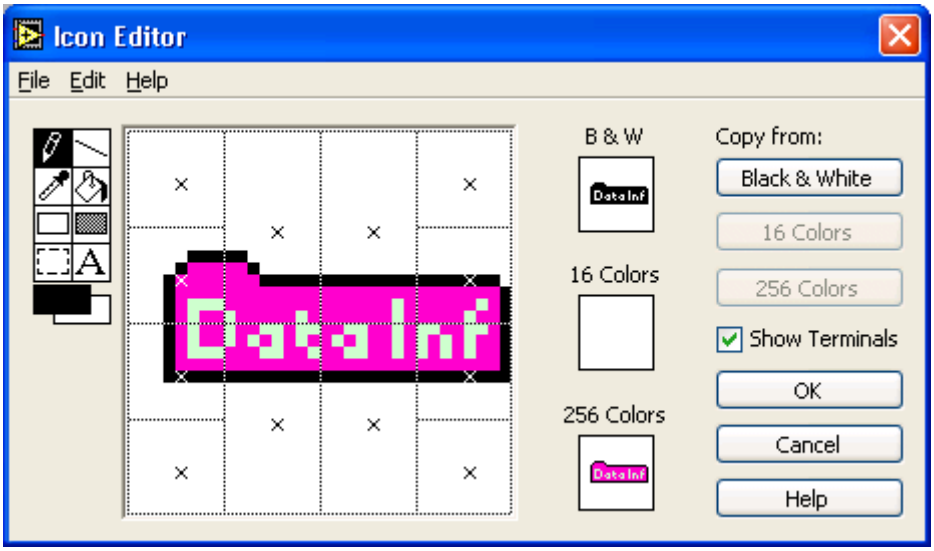


图4：常量数据 VI 的图标

把这个新造出来的常量数据 VI 拖到程序框图上，把它的输出链接到刚才链接常量的地方，再把位置摆放好。现在我们的程序是不是漂亮多了

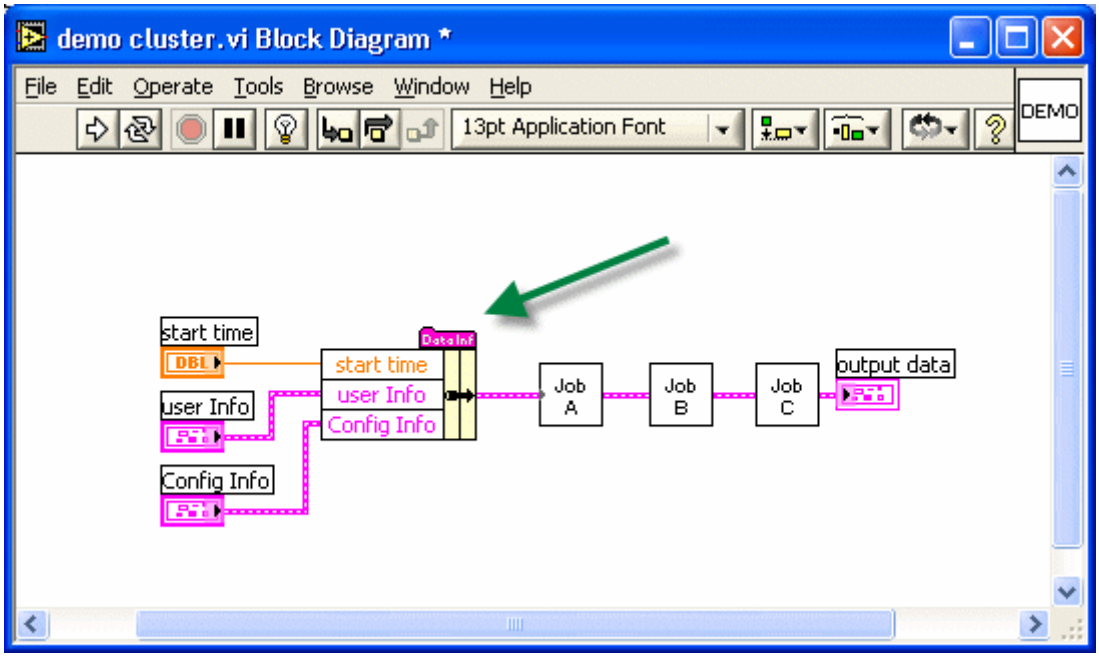

demo cluster.vi Block Diagram *
File Edit Operate Tools Browse Window Help
13pt Application Font
DEMO
start time
user Info
Config Info
start time
user Info
Config Info
Job A
Job B
Job C
output data

图5：改造后的程序框图

制作不规则图形的子 VI 图标

大多数子 VI 的图标都是一个正方块。但有时候，为了程序代码美观、易读，需要把子 VI 作成不规则图形显示在调用它的父 VI 代码上。LabVIEW 中很多自带的函数采用的就不是32×32的正方形，比如说加减乘除运算函数等。我们已可以把子 VI 作成非正方形方块，比如图1中的 My Function.vi，我把它的图标做成了三角形。

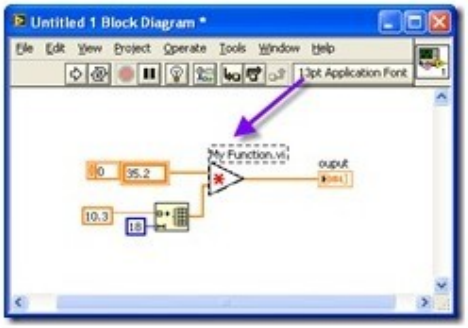


图1：调用 My Function.vi 的代码

把子 VI 的图标改为非规则图形，有一些事项需要注意。首先要考虑选取一个合适的接线方块模式。现在 LabVIEW 默认的模式是4224模式（按照每一列接线端的数量给模式命名的）。但这种模式通常不适合不规则的图标。比如说图1的 My Function.vi，它需要一个纵向排在中央的输出接线端，4224模式最右侧的4个接线端对称分布，没有一个在最中间。

在 VI 前面板的图标上点击鼠标右键，选择“Show Connector”，之后再鼠标右击，就可以挑选一个合适的模式了。对于 My Function .vi，可以使用52225这个模式。如下图所示：

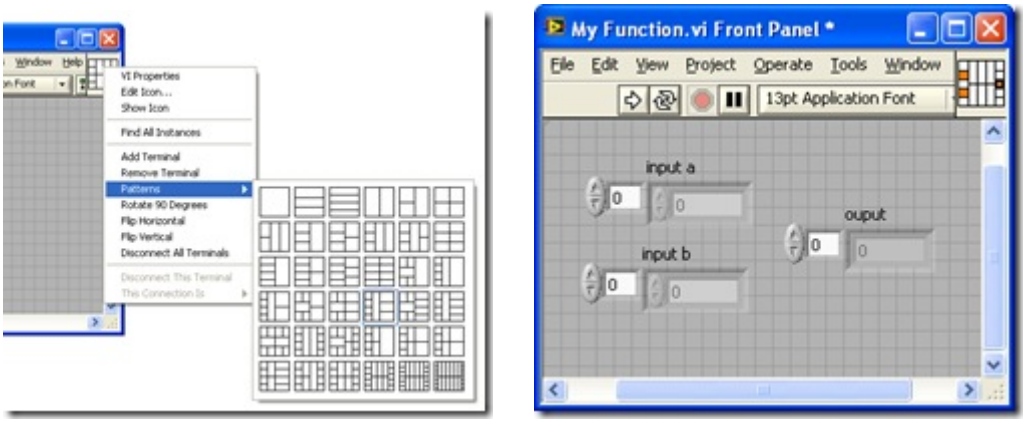


图2、3：选取接线方块的模式

接下来，就是核心工作-编辑图标了。制作不规则形状的图标最重要的两点，一是不要给图标加上边框，二是三个颜色的图标要保持一致。下图是 LabVIEW 的图标编辑器，它的中央有三个小方块，分别显示黑白色、16色、256色下的 VI 图标。用鼠标点击一下某个小方块，左边的编辑区就开始对这个颜色的图标进行编辑。

如果你的显示器是颜色数量高于256色，那么，你在程序代码中看到的始终是256色那个图标；如果少于这个颜色数量，比如只有16色，那么在程序代码上看的的子 VI 显示的就是16色的那个图标。不过现在恐怕找不到16色的显示器了，所以16色这个图标一般情况下就是

空白。黑白色的显示器虽然更罕见，但是这里的黑白图标还有另一个用处，就是确定图标中图形的轮廓。所以这里一定要有一个形状和256色图标项符合的黑白图标才能做出不规则图形图标来。一般在编辑黑白图标时，按一下“Copy from 256 Colors”的按钮，把256色的图标复制过来就可以了。

编辑不规则形状的图标最好选中“Show Terminals”，把接线方块模式在图标编辑区显示出来。这样可以方便的查看，现在的图形是否与接线方块般配。比如说，我要在接线方块右侧中央的接线端接一个输出数据，我的图形至少要覆盖到这个接线端才行。

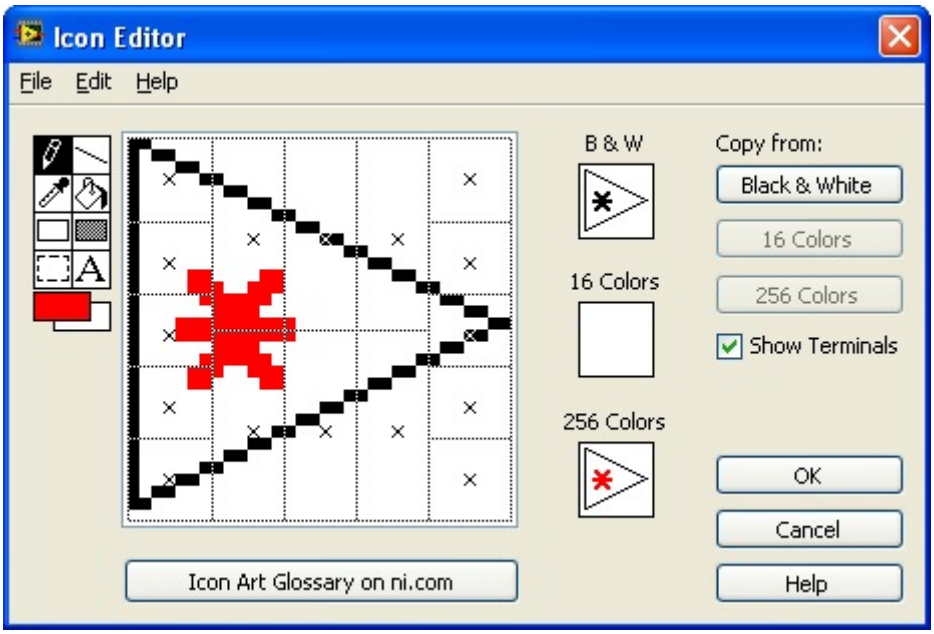


图4：图标编辑器

界面设计技巧 1 - 利用 LabVIEW 自带控件

我前面讲了一堆设计界面的规范和原则，下面介绍一些具体的技巧，可以让界面编写更快捷、美观。

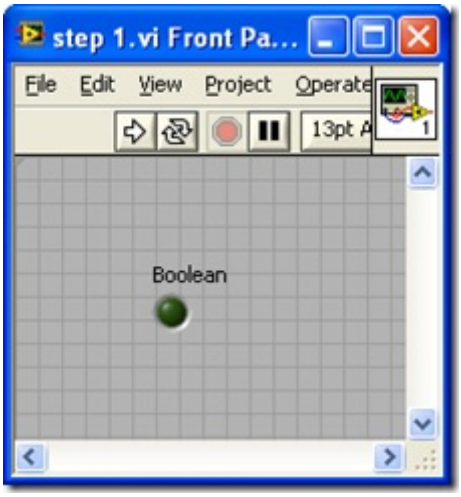
我们需要一个具体示例来帮助介绍这些的技巧，我打算以编写一个黑白棋游戏的界面为例。选择黑白棋是因为这个游戏的界面在常见棋类中比较简单，适合做范例。另外，它也是我最开始学习 LabVIEW 时的练习程序之一，比较有感情:) 黑白棋的棋盘由8×8个正方格组成，旗子为黑白两色，放置在方格中。

编写这样一个界面可以使用到多种不同的思路和技巧，我会按照从简到繁的顺序，分几次来介绍几个不同的方法。

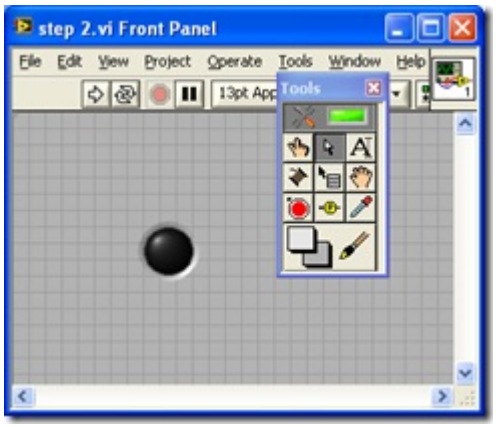
界面设计的时候，首先要调查一下看能不能使用已有的控件。借用已有控件可以大大节省我们自己的开发时间了。我们这个游戏界面上的按钮、文本框等自然可以使用 LabVIEW 自带的控件；黑白棋的棋盘棋子，也可以上网去找找看有没有别人已经做好的可供使用。

假如没有现成的棋盘棋子控件，那就要我们自己来做一个了。虽然作为整体，没有现成的东西可用，但把它细分成小的基础部分，还是有可能利用一些已有控件的。

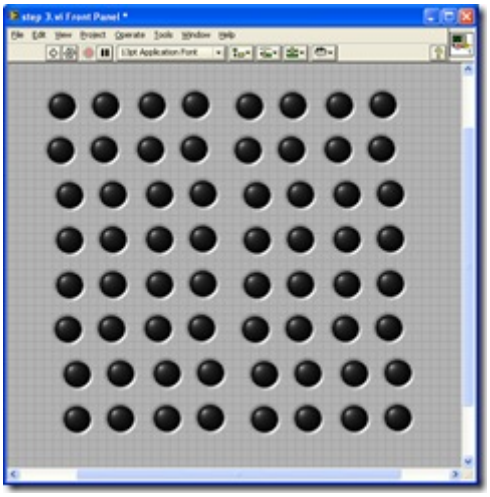
比如说棋子：这个游戏的棋子为圆形，只有黑白两色，个数最多64个。这个特点很适合用 LabVIEW 中的圆形 LED 灯泡来表示。圆形 LED 灯泡控件如下图所示：



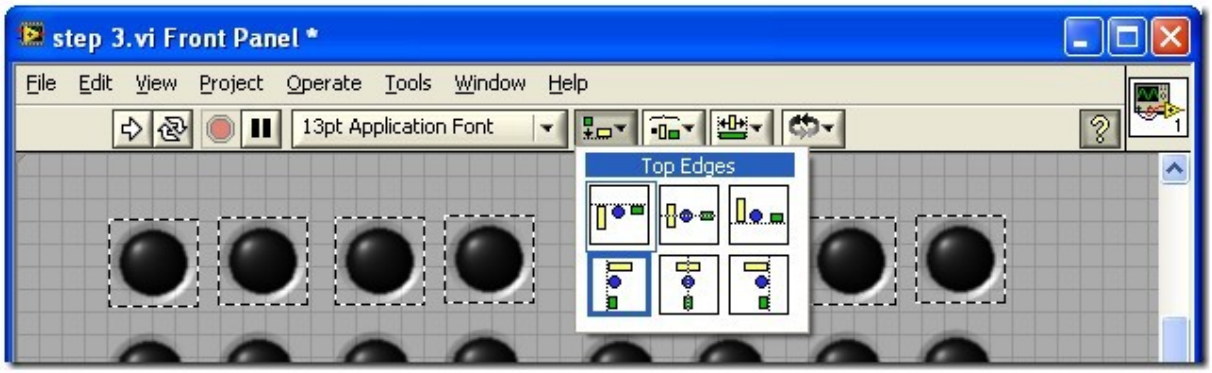
为了使它更像棋子，我们还要对他进行一下加工。首先，要把它的尺寸调大；用工具选板上的颜色画笔工具把它在“真”“假”状态下的颜色分别设置成黑色和白色；给他起一个有意义的名称-chess 0，但是在前面板上需要把这个标签隐藏起来，这个名声是为了以后编程的。改进后的棋子，如下图所示：



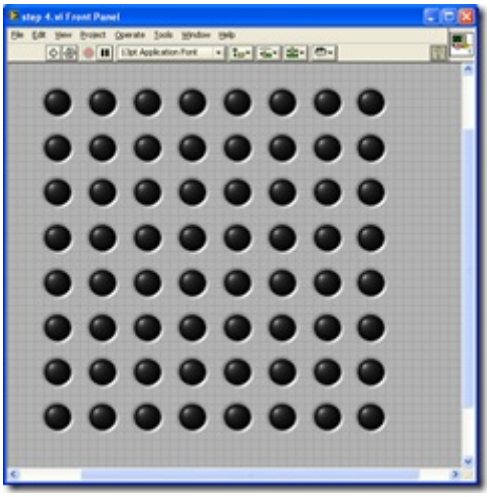
我们总共需要64个这样的棋子，排成8行8列。其它的棋子不需要再一个一个添加，以第一个棋子为模板，拷贝复制，就生成了第二个；再把两个棋子都选中，复制生成四个；重复这一过程，生成8、16、32、64个棋子。如下图所示：



下面我们要把这些棋子排列整齐。如果有耐心，可以用鼠标一个一个的调整每个棋子的位置。LabVIEW 提供了几个小工具来帮我们整理界面控件的位置和大小，它们就是工具条上，字体调整按钮右侧的四个按钮。这四个个按钮分别用于对齐控件，调整控件间距调整控件大小和控件前后次序。这几个工具在编辑界面时会经常使用到。



我们先把首先利用对齐工具把首行和首列棋子对齐、再利用间距调整按钮使它们间距均匀。再利用对齐工具让其它棋子都与首行首列对齐即可。调整好的界面如下：



到此为止，棋子的界面部分就完全设计好了。但是我们还要考虑一下相关的代码。棋子在程序运行过程中时发生变化的。

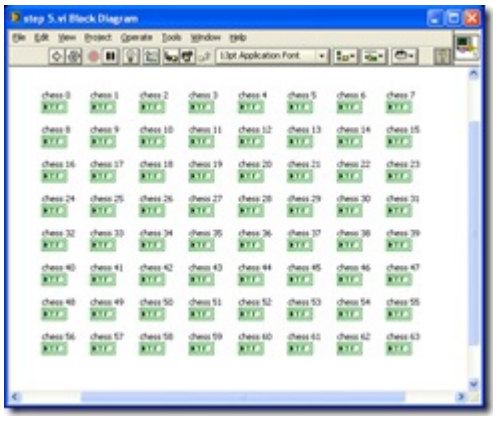
64颗棋子并不都是显示在屏幕上的。游戏一开始，屏幕上只有四颗棋子，以后每走一步多一颗棋子。LabVIEW 每个控件都有一个属性“Visible”，控制控件是否在前面板上显示出来。棋盘的某个位置还没有放棋子时，可将该位置的棋子控件隐藏。

设计界面时，经常遇到有些控件只在某种特定情况下出现。这样的问题有两种最常见解决方案，一是我们刚刚提到的，可以在不需要看见某个控件时设置它的 Visible 属性，将其隐藏。这种方法代码编写比较简单，但是不利于界面编辑。尤其当界面某一位置需要在不同情况下出现多种不同控件的情况下。几个几个控件需要在那个位置上重叠摆放，不利于对控件进行编辑调整。

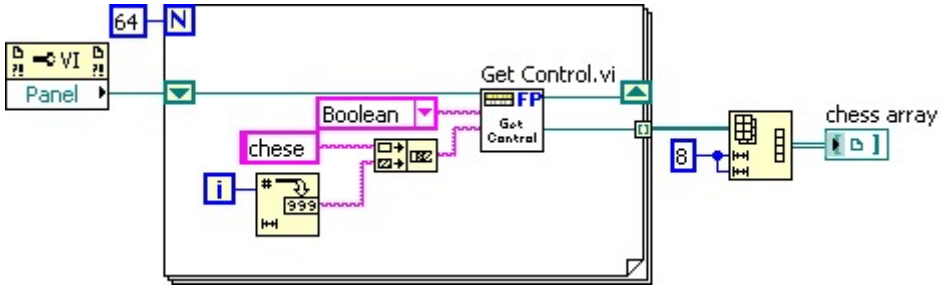
第二种方法是通过控件的 Position 属性，设置它在界面上的位置。需要显示控件时，把它设定到应该出现的位置；需要隐藏它的时候，把它挪到 VI 前面板可视范围之外的某个位置上，这样就看不到它了。使用这种方法，始终可以在 VI 前面板上找到这个控件进行编辑修改。但是编程的时候相对繁琐，需要在程序中设定控件的位置。

如果有一组控件需要同时出现或隐藏，那还可以考虑利用 tab 控件。把这组控件加在 tab 的某个页面上，然后通过调整 tab 的显示页面，控制控件出现与否。

打开程序的框图，64个控件端子排布在那里。对它们分别进行操作，程序代码将会非常杂乱难懂。为了让程序更清晰，最好把这64个控件按照在棋盘上的位置，组织成一个8×8的二维数组。之后，程序对哪个位置的棋子进行操作就一目了然了。



直接把它们组成数组的方法是：为每个控件建立一个引用，然后使用 build array 函数把它们组织起来。但是对64个控件进行一一操作还是够烦的，最好可以编程解决。由于这64个棋子的名字是有规律的，因此我们可编程，按照名字一一等到这些控件的引用。再将得到的引用转换成8×8的数组。如下图所示的代码



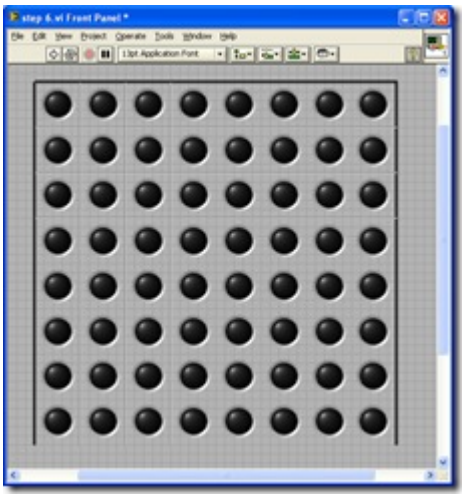
这里使用了一个关键的子 VI，Get Control.vi。这是 LabVIEW 自带的一个 VI（ [LabVIEW]\resource\importtools\Common\VI Scripting\VI\Front Panel\Method\Get Control.vi），它用来按名称得到前面板上控件的引用。

这段代码输出的 chess array 是一个8×8数组，包含了所有64个控件。之后程序再对棋子进行操作，从这里得到相应位置的棋子的引用即可对其进行操作了。

实际工作中，有些应用程序有比较复杂的界面，为了简化它的代码，对界面控件的操作被放置在子 VI 中完成。直观的做法也是：程序开始时为主程序的控件建立引用，把这些引用捆绑成一个簇，传递到子 VI 中去。但是，一旦界面发生变化，所有使用到这个簇的 VI 都可能需要被修改，相当不便。所以，这样的程序也可以使用上段文字介绍的方案，只把主 VI 的引用传递给子 VI，在使用到某个主 VI 控件的时候，按照名字得到它的引用再对其进行操作。

界面设计技巧 2 - 装饰和背景图片

现在棋子都已经摆放到位了，下面考虑如何把棋盘加上去。由于棋盘是静态不动的，所以设计起来要比棋子简单。LabVIEW 自带了这种形状的装饰组件，比如线条、方块之类的，利用这些装饰图案，很容易搭出一个棋盘来。如下图，就是由几根被画成黑色的线条搭出来的部分棋盘。

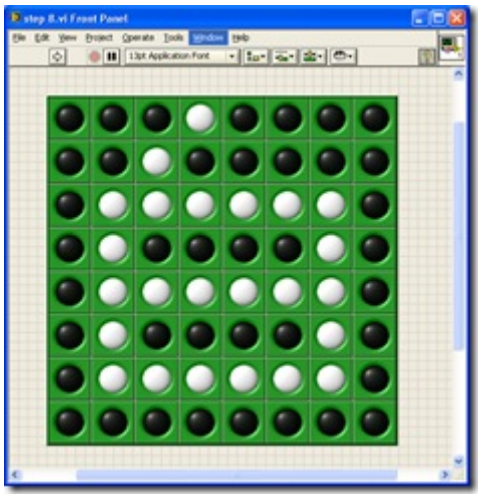


在编写程序界面时，装饰最常被用来将一组相关联的控件包围起来，或把不相关的控件个离开。

不过呢，用 LabVIEW 自带的简单图形拼出来的棋盘始终是不够漂亮。我们可以先用专业的画图工具，比如画图板（也不怎么专业吗）画一个漂亮的棋盘，保存成图片文件。然后把图片贴到 VI 的前面板，当作背景图片。这样，就可以得到一个漂亮的多的棋盘了。

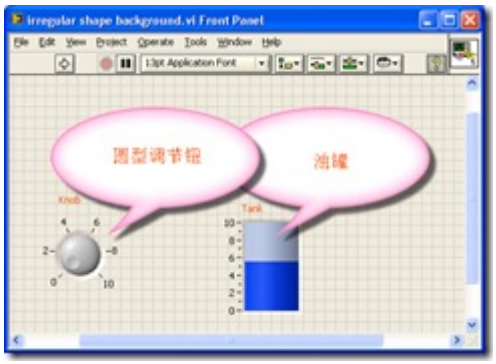
贴图这个操作，可以用 Ctrl+C，Ctrl+V，也可以直接在文件浏览器中，用鼠标把图片文件拖拽到 VI 前面板。

拖拽到 VI 上的图片，是处在界面最上层的，覆盖住了棋子。利用 Reorder 工具中的“Move to Back”，把它挪到最下层。在调整好棋子和棋盘的位置，整个一个棋子棋盘界面就做好了。如下图：

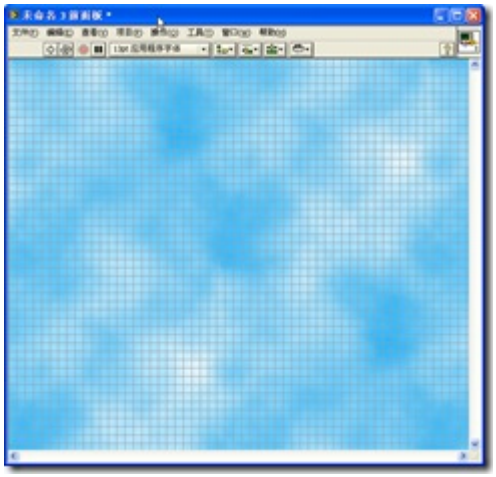


棋盘棋子都摆好之后，它们的相对位置应该固定下来。如果需要它在界面上挪动，应该是所有的棋子和棋盘一起动。先用鼠标把棋盘和全部棋子选中，再在 Reorder 工具中，选择 group 就可以把它们设定为一组。之后他们之间的相对位置就固定下来了。

咱们刚刚贴上来的图片是个矩形的，可是有时候，需要背景图片是不规则形状的。这种情况下需要使用支持透明色的图片格式，比如 gif 格式，把不规则图片空白部分设为透明即可。还有一种常用的文件格式 png 格式，支持像素点透明度的设置，利用不同的透明度设置还可以给背景图片做出阴影等效果。例如下图 VI 界面中两个带粉色的带阴影效果的解说框，使用的就是 png 文件格式的图片。



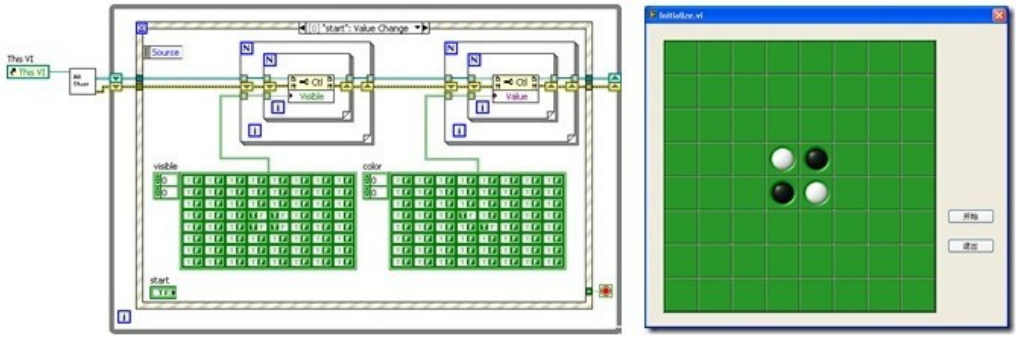
这样子贴上来的背景图片大小尺寸是固定的。有时候我们希望整个前面板被某一背景图片铺满。这个设置是在窗格的属性中设定的。鼠标右键点击前面板窗口的滚动条，选择“属性”，在“背景”设置中选择一个背景，或添加一幅自己的图片，就可以为 VI 前面板设置上背景图片了。下图是使用的“Clouds”背景的效果：



界面设计技巧 4 - 改进界面实现方法

到目前为止，棋盘棋子的界面已经基本成型。下面我们实现一小部分代码，来看看这个界面设计方案是否可行，是否可以改进。

以棋盘的初始化为例，在游戏开始时，只有两黑两白四颗子，摆在期盼最中间。实现这个操作的代码和执行结果如下图所示：



代码中的子 VI（Get All Chess.vi）中的代码，就是我们在第一节图8中看到的那段代码。它负责得到所有棋子的引用，并排列成二维数组。

这段初始化的代码并不算复杂，但是我们还是可以从中看出一些问题：棋子的布局需要用两个数组才可表达清楚，这给编程增加了负担。造成这一状况的根本原因在于：每个位置上的棋子实际上有三个状态：黑、白、无；而我们选用的灯泡控件，只有两个值：真、假。用这两个值不足以完全表达棋子的状态，所以，要两个布尔类型才能确定一个棋子的状态。

另外，棋盘只是一张背景图片，这样，判断鼠标在棋盘哪个位置上进行的点击，也比较繁琐。一旦棋盘挪动，代码也需要做相应改动。

改进的方法，就是使用一个有更多值的控件来表示棋子。在我们的程序中，要求每个棋子位置有多个不同图片，这正好可以使用 Pict Ring 控件。Pict Ring 控件包含三个值：空白图片、黑色棋子图片、白色棋子图片。这样即便是没有棋子的位置，Pict Ring 控件还在，可以感知用户的鼠标点击事件。

上面的代码还有一处不足，每个棋子都是一个独立的控件，造成界面控件太多，不好管理。对于更复杂的程序，比如围棋游戏，如果使用这种方式，界面上就有将近400个控件，这是不可接受的。

改进的方法是使用数组控件，把所有的棋子组成一个数组。

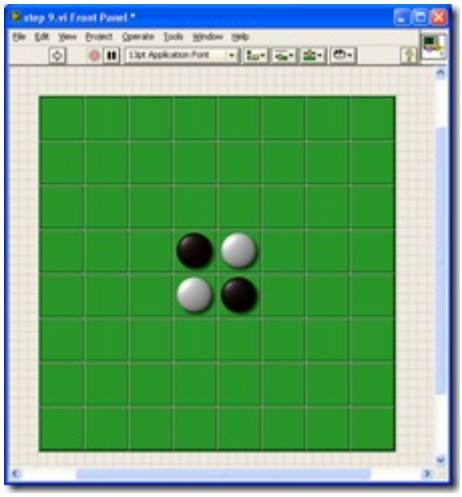
具体的实现步骤如下：

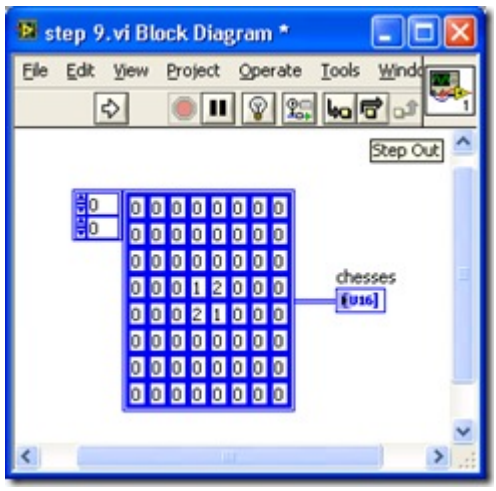
首先创建一个经典风格的 Pict Ring 控件，添加三幅图片：空白、黑棋子和白棋子。棋子采用的都是 png 文件格式的图片，以实现透明和阴影效果。

在程序中，我们不希望看见 Pict Ring 控件的边框和背景。我们可以用透明画笔，把边框和背景画为透明。

造一个二维数组用来放置棋子元素。由于界面上不希望看到这个数组的边框和背景，所以同样用透明画笔把它们画为透明。数组的标签、索引显示可以通过数组的右键菜单->显示一项进行隐藏。

把棋子元素放置于数组中，并把数组拉成8×8大小，放置在棋盘上放。一个棋盘棋子控件就做好了。而这种做法又大大简化了编码的复杂度，比如同样是初始化设置，只要一个赋值语句就可完成：




step 9.vi Block Diagram *
File Edit View Project Operate Tools
Step Out
chesses
U16

界面编程技巧 5 - 使用绘图控件

有时候需要画一个比较复杂图形或曲线，而 LabVIEW 没有提供相应的控件。可以借用 LabVIEW 已有的基本功能的控件，配上一些代码，实现一个具有特定功能的控件。

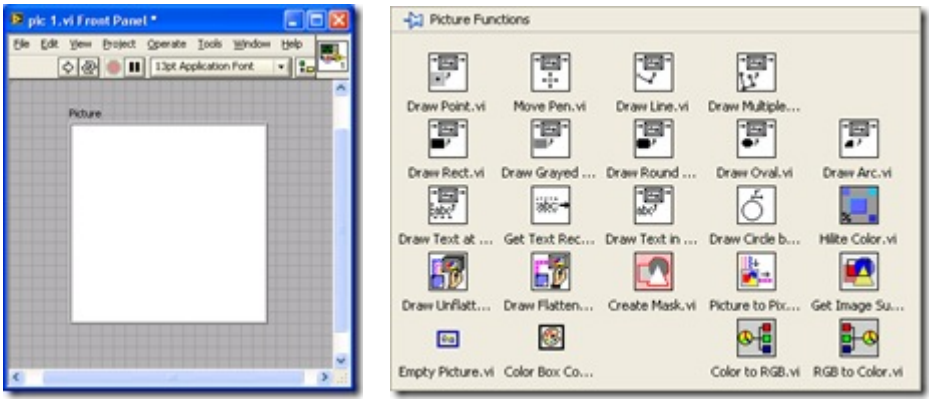
常被用来做这种基本控件有 XY Graph、3D Picture Control、Picture 控件等。

例如需要做一个绘制极坐标函数曲线的控件，就可以在 XY Graph 的基础上改造。一共一个转换用的 VI，把点的极坐标转换成直角坐标系下的值，在 XY Graph 上绘制出来就可以了。需要某个支持某种特定三维绘图方式的控件，可以通过改造 3D Picture Control 得到。

Picture 控件是个更为基础的控件，很多具有特殊效果的界面元素都可以利用 Picture 控件制作。比如，需要制作带图标的菜单，或类似 LabVIEW 函数选板的菜单等。LabVIEW 没有为它们提供现成的控件，就可以在 Picture 控件上自己把这些效果都画出来。我们前面介绍的棋盘棋子界面也可以使用 Picture 控件来制作。

下面介绍一下实现这个界面的具体过程。

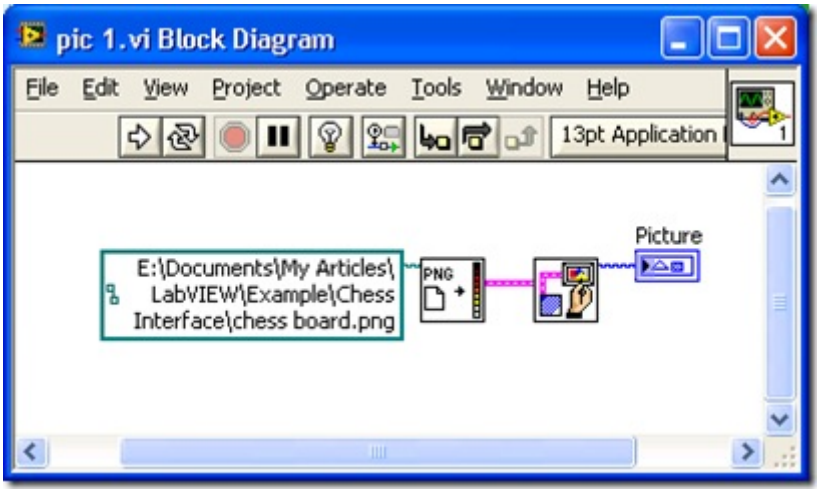
第一步，创建一个空白的 Picture 控件，针对 Picture 控件的常用操作都在 Picture Functions 函数选板中。

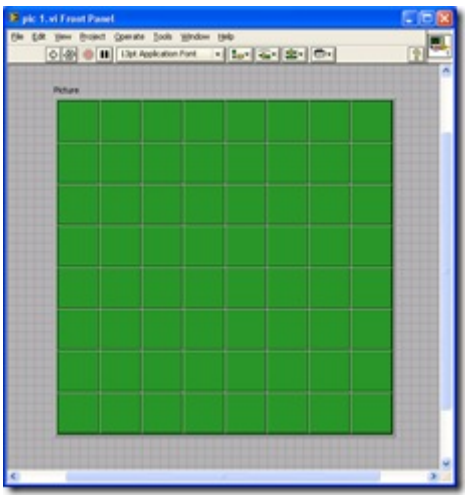


与前面介绍的方法不同，使用 Picture 控件制作棋盘棋子的过程，不是在 VI 编辑状态下进行的，而是需要在程序运行时绘制。所以下面的界面设计工作都要通过编程来完成了。先介绍一下 Picture 控件的 Erase First 属性，它有3个值：0表示从不擦除，也就是说每次传一个数据给这个控件，比如一个圆环图案，Picture 上显示的并非只有这个圆环，而是把圆环叠加在原本的内容之上。如果我想画一个有三个矩形组成的图案，可以分三次画，每一次传递一个矩形图案给 Picture 控件；2表示每次都擦除，每次传递一个图案给 Picture 控件，它都会将原来的图案擦掉，仅保留这一次的图形。擦除图案后 Picture 控件会显示默认的白色。所以，使用这种方式，用户在切换图案时会看到 Picture 闪烁一下。若非必要，尽量不要使用这种方式；1表示程序第一次运行时把 Picture 上的内容清除，等于自动帮你做了初始化工作，我们这里也使用这种方式。

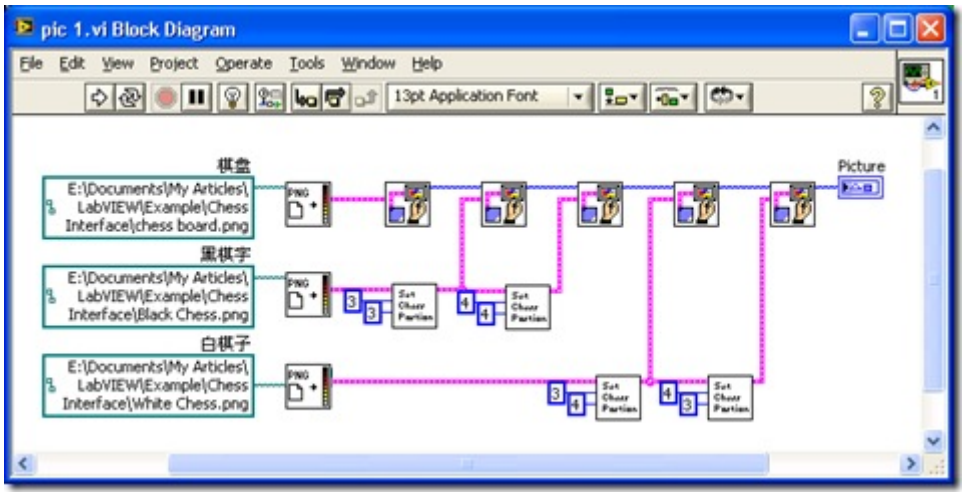
就是说，棋盘布局发生变化时，进更新发生了变动的位置。不要重绘整幅图。

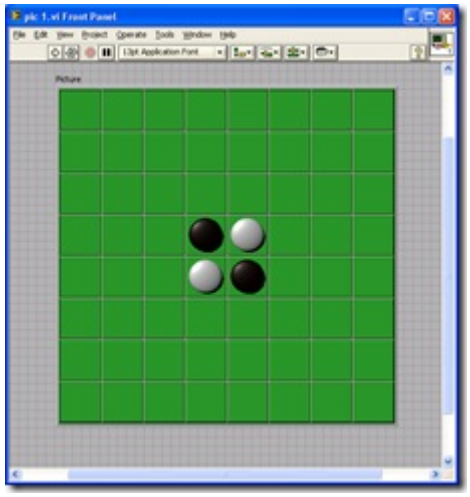
棋盘再棋子下层，所以要先画棋盘。画棋盘可以使用 LabVIEW 提供的划线函数，一条线一条线画出来。因为我们之前已经制作了棋盘的图片，所以可以直接把这张图片显示出来。代码如下：





下面再画上棋盘初始时的四个棋子。画棋子的方法与棋盘相同，可以使用画圆函数，已可以使用已经制作好的图片：





到目前为止，界面设计的几种方法就已经介绍好了。如果能够把这个黑白棋的相关界面和操作（比如放置棋子，反转棋子等）提取出来，合成一个组件，公布出来，其他有类似需求的人就可以直接利用这个组件，不再需要自己重新设计了。

然而，在 LabVIEW 8 之前是无法实现这一功能呢，因为控制棋子行为的代码分散在程序的各处，而棋盘棋子也是主 VI 的一部分，很难将它们提取出来组成独立模块。LabVIEW 8 中出现的 XControl 可以把控件的界面及行为封装在一起，成为一个既有界面，又有运行代码一个组件。

下面我就会重启一个先话题，讨论如何把这个黑白棋做成一个可以独立发布的组件。

LabVIEW 程序的内存优化 1

一. VI 在内存中的结构

打开一个 VI 的属性面板(VI Properties)，其中的“内存使用”(Memory Usage)是用来查看这个 VI 内存占用情况的。它显示了一个 VI 内存占用所包含的四个主要部分：前面板、框图、代码和数据，以及这四个部分的总和。但在打开一个 VI 时，这四段内容并不是同时都会被 LabVIEW 调入内存的。

当我们打开一个主 VI 时，主 VI 连同它的所有子 VI 的代码和数据段都会被调入内存。由于主 VI 的前面板一般情况下是打开的，它的前面板也就同时被调入内存。但是此时主 VI 的框图和子 VI 的前面板、框图并没有被调入内存。只有当主动查看主 VI 的框图或是打开子 VI 的前面板和框图时，它们才会被调入。

基于 LabVIEW 的这种内存管理的特性，我们在编写 VI 的时候可以通过以下方法来优化 LabVIEW 程序的内存使用。

第一，把一个复杂 VI 分解为数个子 VI。子 VI 的使用会增添额外的前面板和框图的空间，但并不增添额外的代码和数据空间。由于程序运行时只有代码和数据被调入内存，因此使用子 VI 不会占用额外的内存。使用子 VI 的好处还在于当子 VI 运行结束时，LabVIEW 可以及时收回子 VI 的数据空间，从而改善了内存的使用效率。

第二，在没有必要时不要设置子 VI 的重入(Reentrant)属性。重入型 VI 每次运行时都会对自己使用的数据生成一个副本，这增加了内存开销。

第三，主 VI 的面板通常就是用户界面，需要显示给用户。但是要尽量避免开启子 VI 前面板。比如，在子 VI 中使用与其前面板控件有关的属性节点(Property Node)会导致它的前面板被调入内存中，增加了内存开销，所以要尽量避免在子 VI 中使用主面板控件的属性节点来设置控件的值，而可以用局部变量等方法来替代。

第四，我们可以放心地在 VI 的前面板(对于非界面 VI)和框图里添加图片，注释等信息来帮助你编写、维护 LabVIEW 程序，这些帮助信息不会在 VI 运行时占用内存。

二. 内存泄漏。

LabVIEW 与 C 语言不同，它没有任何分配或释放内存的语句，LabVIEW 可以自动管理内存，在适当的时候分配或收回内存资源[1]。这样就避免了 C 语言中常见的因为内存管理语句使用不当而引起的内存泄漏。

在 LabVIEW 中一般只有一种情况能够引起内存泄漏，即你打开了某些资源，却忘记了关闭它们。比如，在对文件操作时，我们需要先打开这个文件，返回它的句柄。随后如果忘记了关闭这个句柄，它所占用的内存就始终不会被释放，从而产生内存泄漏。LabVIEW 中其它带有打开句柄的函数或 VI 也会引起同样的问题。

由于内存泄漏是动态产生的，我们无法通过 VI 的属性面板来查看，但可以通过 Windows 自带的任务管理工具来查看 LabVIEW 程序内存是否有泄漏。也可以使用 LabVIEW 的 Profile (Tools>>Advanced>>Profile VIs)工具来查看某个 VI 运行时内存的分配情况。

三. 缓存重用

LabVIEW 程序主要是数据流驱动型的。数据传递到不同节点时往往需要复制一个副本。这是 LabVIEW 为了防止数据被节点改变引起错误所做的一种数据保护措施。只有当目标节点为只读节点，不可能对输入数据作任何更改时，才不在这些节点处做备份。例如，数组索引节点(Index)是不会改变数组值的，LabVIEW 在这里就不为输入数组做备份。对于加减法运算等肯定改变输入数据的节点，LabVIEW 往往需要对输入或输出数据作备份。有些 LabVIEW 程序，比如涉及到大数组运算的程序，内存消耗极大。其主要原因就是 LabVIEW 在运算时为数组数据生成了过多的副本。

实际上很多 LabVIEW 节点是允许使用缓存重用的，这类似 C 语言调用子函数所使用的地址传递。通过合理设计和使用缓存重用节点，可以大大优化 LabVIEW 程序的内存使用。使用 LabVIEW 7.1的 Tool>>Advanced>>Show Buffer Allocations (LabVIEW 8.0 之后使用 Tool>>Profile>>Show Buffer Allocations) 工具可以在 VI 框图中查看缓存的分配情况。打开该工具，凡是在框图中有缓存分配的地方，都会显示出一个黑点。

下面是几个最常用节点的试验结果。LabVIEW 节点众多，不可能一一列举，文中未提及的节点读者在编程时自己可以尝试。

1. 一般顺序执行 VI 中的运算节点

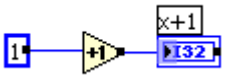


图1：简单的顺序执行程序

如图1所示，程序对一个常量加1，然后将结果输出。

“+1”节点输出端有一个黑点，表示 LabVIEW 在此处开辟了一个缓存用于保存运算结果。

其实完全可以利用输入数据的内存空间来保存这个运算结果。我们可以通过如下的方法来告知 LabVIEW 编译器，在此运算节点处重用输入数据的内存空间。

首先，用一个控制型数值控件代替图中的数值常量，然后分别将 VI 中的两个控件与 VI 的接线器 (Connector Pane)相连。

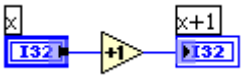


图2：实现缓存重用

图2是经过我们优化后的 VI，LabVIEW 在“+1”节点处没有开辟新的缓存。LabVIEW 中其它运算节点也有类似的性质。

2. 移位寄存器(Shift Register in the Loop Structure)

移位寄存器是 LabVIEW 内存优化中最为重要的一个节点，因为移位寄存器在循环结构两端的接线端是强制使用同一内存的。这一特性可以被用来通知 LabVIEW 在编译循环内代码时，重用输入输出缓存。

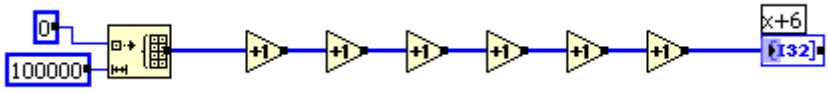


图3： 对数组进行数值运算的顺序执行程序

让我们分析一下图3所示的程序：它首先构造了一个数组，然后对这个数组进行了几次数学运算。每一步运算，LabVIEW 都要开辟一块缓存用以保存运算结果的副本。打开 VI 属性面板上的内存使用，可以看到这个 VI 大约会占用2.7M 的内存空间。其实这些副本都是不必要的，每一步运算的结果都可以被保存到输入数据的内存空间。我们可以把所用的运算节点都放到一个子 VI 中，然后利用上一段提到的方法，使子 VI 中的代码缓存重用。还有一种方法，利用移位寄存器也可以实现缓存重用。

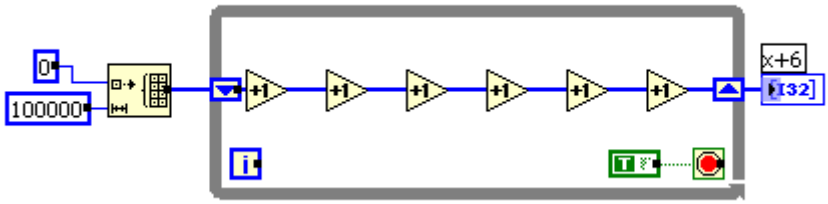


图4： 利用移位寄存器实现缓存重用

如图4，我们可以将运算代码放在一个只运行一次的循环结构内，由于运算部分的输入和输出都与移位寄存器相连，这就相当于通知了 LabVIEW，在运算的输入输出需要使用同一块缓存。因而，LabVIEW 不再为每一步运算开辟新的缓存而是直接利用输入数据的缓存保存结果。打开 VI 属性面板上的内存使用，可以查看到这个 VI 的内存占用已经减少到了原来的六分之一。

在 LabVIEW 8.5 中，有了一个新的结构——缓存重用结构，专门用于优化代码的内存使用。可以不必再使用移位寄存器来完成这项工作了。

3. 库函数调用节点(Call Library Node)

以传递整型参数为例：在参数配置面板，我们可以选择值传递(Pass Value)或选择指针传递(Pass Pointer

to Value)。

当选择了值传递时，库函数调用节点是不会改变该参数的内容的。如果我们在该库函数调用节点参数的左侧接线端引入输入数据，在输出端引出输出参数，那么输出数据其实是直接由输入数据引出的，LabVIEW 不会在这个节点处开辟缓存。

在指针传递方式时，LabVIEW 则认为传入的数据会被改变。如果输入数据同时还要发往其它节点，LabVIEW 会在此处开辟缓存，为输入数据作一个副本。选用指针传递方式，库函数调用节点的每一对接线端也同样是缓存重用的。就是说，库函数调用节点的输出值是直接存放在输入值的缓存空间的。

如果一个参数只用作输出，我们通常会在库函数调用节点的输入接线端为它建立一个输入常数，这个常数的地址空间并不能直接被利用，它只是为库函数调用节点开辟的缓存而设置的初始值。不接输入常数，LabVIEW 也会为此参数开辟一块缓存。但是，这样每次传入的参数值都会有变化。例如图5，库函数调用节点调用的函数功能是为把输入的值加1，然后输出。图5-a 中的输出值永远都是1，而图5-b，每次运行输出结果都会比前次增加1。这是因为库函数调用节点每个指针传递的参数的输入输出用的是同一块缓存，即每次运行输入值是上回的输出值。

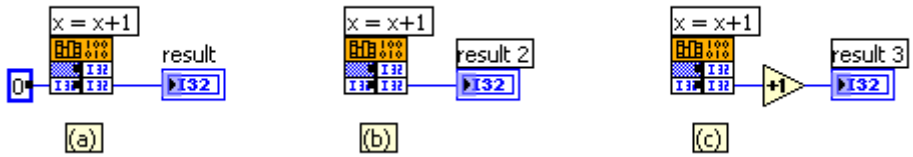


图5： 库函数调用节点

我们可以利用图5-c 的例子证明 LabVIEW 某些节点是缓存重用的。每次运行5-c 的例子，输出结果都会比前次增加2。这是因为示例中的参数接线端以及“+1”节点的输入输出端所使用的都是同一缓存。

如果，库函数调用节点中某个参数只有输入链进去，没有输出。那么，LabVIEW 是假设你调用的函数不会修改这个参数的。LabVIEW 不会为这个数据做拷贝，它会重用这个数据的缓存。但如果你调用的函数修改的这个数据，你的程序就会面临这样一个潜在的危险：这个数据可能被程序其它部分的代码使用了，在那里，你看不出这个数据有任何被改动的地方，但它在运行时却不是你期望的数值。因为这个数据所在的缓存，被程序其它一个地方的一个库函数调用节点给重用了，而这个节点又偷偷摸摸的修改了它。

在图5中的示例中，如果库函数调用节点输出的参数是个数组或者字符串，那么就必须为它相对应的输入端联入一个与输出数据大小一致的数组或字符串。否则，LabVIEW 无法知道输出数据的大小，而使用默认分配的缓存空间很容易出现数组越界错误。

四. 小结

缓存重用是 LabVIEW 内存优化的最重要的一个环节。精心设计的 LabVIEW 程序可以大大节约内存的占用，提高运行效率。但是，在编写完程序后再按照程序优化的技巧回头去优化一段已有的程序，这并不是一个好的编程方法。我们应该先熟悉理解优化的方法，在以后的开发过程中自然而然地将它们应用在编

程中。

参考文章：

LabVIEW 程序中的线程 1

LabVIEW 程序的内存优化 2 - 子 VI 的优化

LabVIEW 是编译型语言还是解释型语言

我和 LabVIEW

pdf 格式文档下载

其他技术文章

LabVIEW 程序的内存优化 2 - 子 VI 的优化

1. 子 VI 参数的缓存重用

数据在子 VI 间传入传出，如果程序设计的好，可以做到缓存重用，使得数据在主 VI 和子 VI 中都不发生拷贝，提高程序的效率。

我们先来看一下图1所示的 VI。打开 Tool>>Profile>>Show Buffer Allocations 工具查看一下这个 VI 中内存分配的情况，会发现在代码的加法函数处有一个黑点。这个黑点说明程序在这里有分配了一块内存，这个内存是用来存储加法运算结果的。s

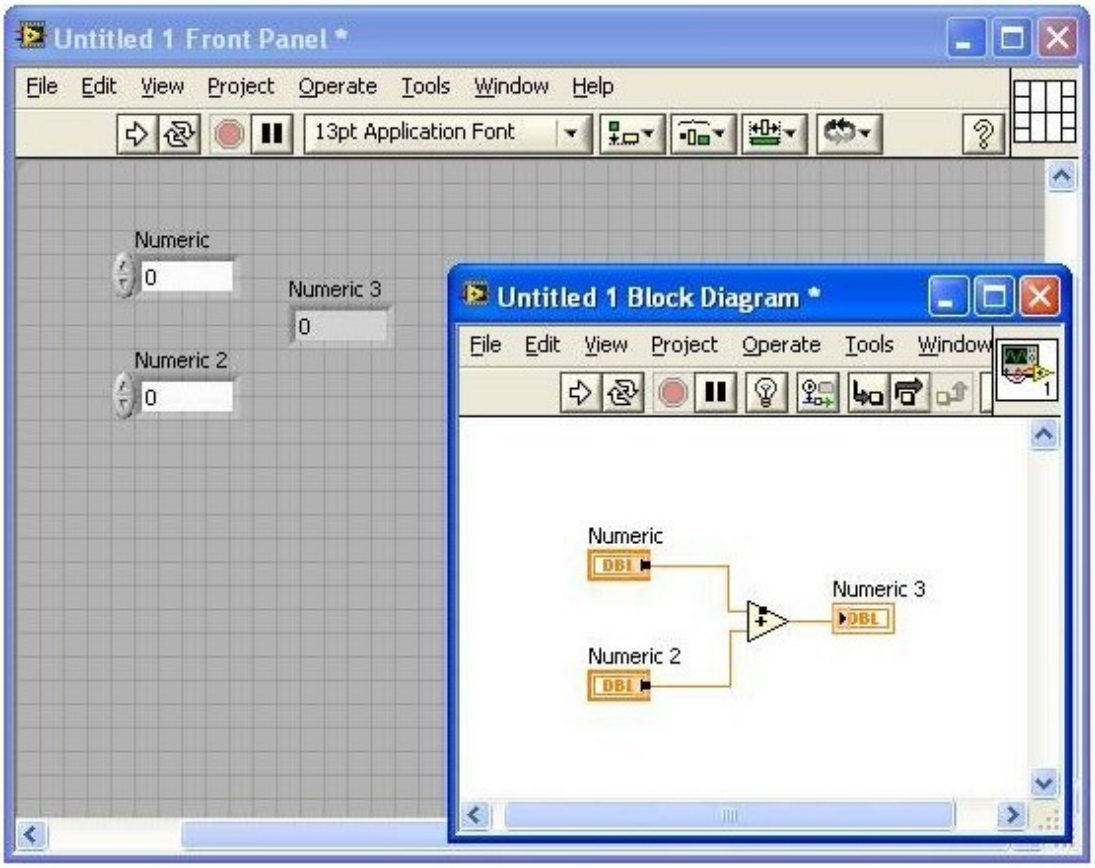


图1：控件不与接线器相连时，加法处有内存分配

为什么加法函数在这里不做缓存重用呢?利用其中一个加数的内存空间来保存计算结果。

当这个 VI 运行的时候，图2中，加数 Numeric 的数据是由 VI 前面板的控件提供的。如果用户不修改控件的值，每次 VI 运行，这个数值应该是保持不变的。如果加法函数在这里做缓存重用，加数或者说它对应的控件中的数据，就会在加法运算执行后被修改。这样程序就会出现逻辑上的错误。

所以把一个这样的控件联在 LabVIEW 的运算节点上，运算节点是不能重用控件的数据内存的。同样的道理，链接一个常量到运算节点上，节点同样不能做缓存重用。在子 VI 中，没有连到接线器上的输入

控件就相当与一个常量。

但是，如果我们让 VI 上的控件与 VI 的接线器(Connector Pane)相连，情况就不一样了。入图2所示，把三个控件连到接线器上，程序中加法节点上那个黑点就消失了，不再为运算结果分配新的内存。

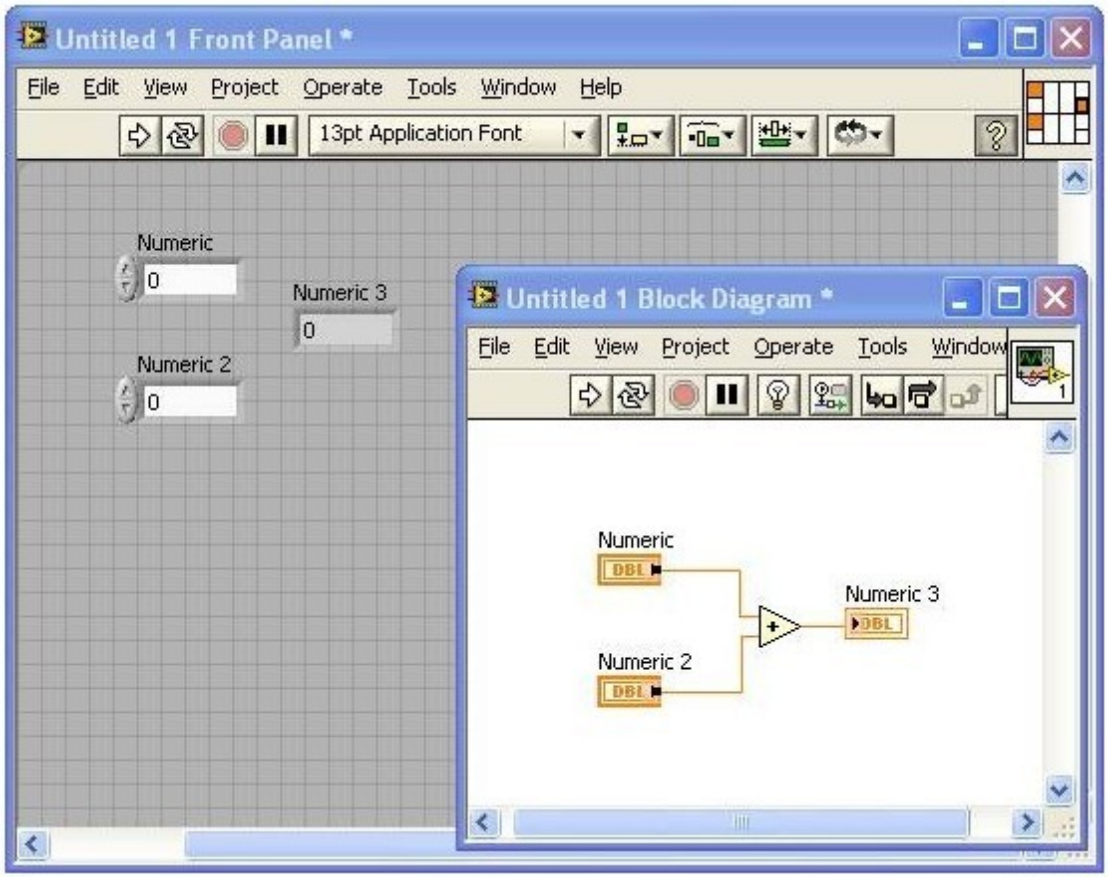


图2：控件不与接线器相连时，加法处有内存分配

这是因为，当输入控件与接线器连接后，LabVIEW 就认为这个输入值应当是由子 VI 的调用者(父 VI)提供的：连到接线器上，逻辑上，这个输入控件就不再是常量，而是一个输入变量了。既然是输入变量，子 VI 不需要记住输入的数据共下次调用时使用，因此可以把新产生的数据放在输入参数所在的内存，做到缓存重用。

你可能在想，这个输入参数的内存不一定可以被修改吧，万一它的数据还要在父 VI 中被其它节点使用呢?

子 VI 是不需要考虑这点的，输入数据的数据被修改肯定是安全的，这一点是由父 VI 来保证的。如果输入数据不能被修改，父 VI 会把传入的数据拷贝一份再传到子 VI 中去。

比如图3中的程序，它所调用的子 VI 就是图2中那个 VI。由于与它的第一个输入参数相连的是一个常量，而常量的值是不能被改变的。所以 LabVIEW 要把这个常量的值复制一份，再传到子 VI 中去，以保证子 VI 中的运算节点可以做缓存重用。

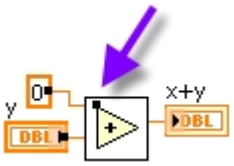


图3：父 VI 中的数据拷贝

如果图3中的父 VI，他也使用与接线器相连的输入控件为子 VI 提供输入参数，则 LabVIEW 会知道，父 VI 的这个数据是由再上一层 VI 提供的，这里也不需要需要做数据拷贝。这样，这个 VI 就也做到了缓存重用。设计合理，参数在传递多个深度后都不需要开辟新内存的。

从上面的说明中，还可以发现一个问题。就是，有时候子 VI 的改动，会影响父 VI 的行为，比如是否为传入子 VI 的数据做个拷贝等等。有时候我们发现改动了一个子 VI，它的父 VI 也需要重新保存，就是由这个原因引起的。

2. 输入输出参数的排布

在子 VI 的程序框图上，不论代码有多复杂，有多少嵌套的结构，控件终端最好按照这样的方式排布：所有输入参数(控制型控件的终端)都放在代码的最左端排成一列;所有的输出参数(显示型控件的终端)都放在代码。比如图4中的代码的风格就比较好。

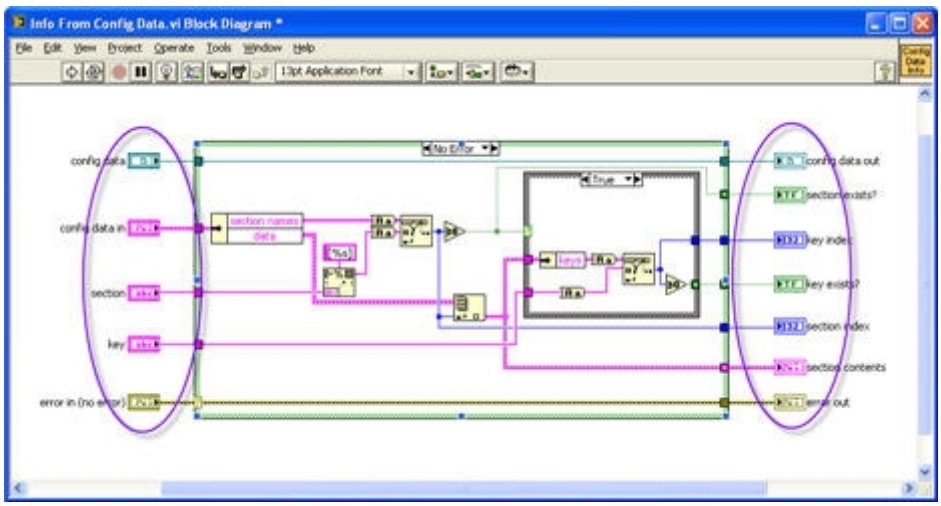


图4：控件终端整齐的排列在程序框图左右两端

这首先是为了保证程序有良好的可读性。我们在阅读 LabVIEW 代码的时候总是按照从左到右的顺序，所有的参数都排布在一起，我们就可以以数据线为线索，轻易的找的数据被读写的地方。其次，这种风格的 VI，在效率上也比较优化。

对于一个输入参数(控制型控件的终端)，如果把它放程序代码的最左侧，所有结构的外面，程序在运

行这个子 VI 之前，就可以得到这个参数的确切值了。

但是，如果这个终端是在代码的某个结构中的，在某一结构的内部，那么 LabVIEW 必须在运行到这一结构内部的时候，才可以去读这个参数的值，否则可能会引起罗技上的错误。比如说，一个控制型控件的终端是在一个循环的内部，开始时它的值是 x。在运行到第 n 次循环之前，这个终端对应的前面板上的控件被人改为一个新的数值 y。那么逻辑上，在执行第 n 次循环之前，每次用到这个参数时，它的值要保持为 x，而在第 n 次循环的时候，又要使用它的新值 y。这样的数据所在的内存，LabVIEW 显然是不能将其重用的，否则下次循环再读它的时候，数据就不正确了。

如果这个终端是在所有结构之外，LabVIEW 则可以根据数据线的链接，明确的判断出在某一节点执行完之后，程序再也不需要用到这个参数的值了，那么 LabVIEW 就可以重用它所在的内存，以避免开辟新内存，拷贝数据等操作。这样就提高了程序的内存效率。

对于一个输出参数(显示型控件的终端)，如果它位于某个条件结构的内部，LabVIEW 就要考虑，程序有可能执行不到这个条件。LabVIEW 就会多添加一些代码来处理这种情况，当 VI 没有运行到这个条件时，要给输出参数准备一个默认值。

把这个终端移到所有结构之外，就可以省去这部分 LabVIEW 自动添加上去的工作和，稍微提高一点效率:)

3. 良好的数据流结构可以优化程序内存效率

先看一个程序:

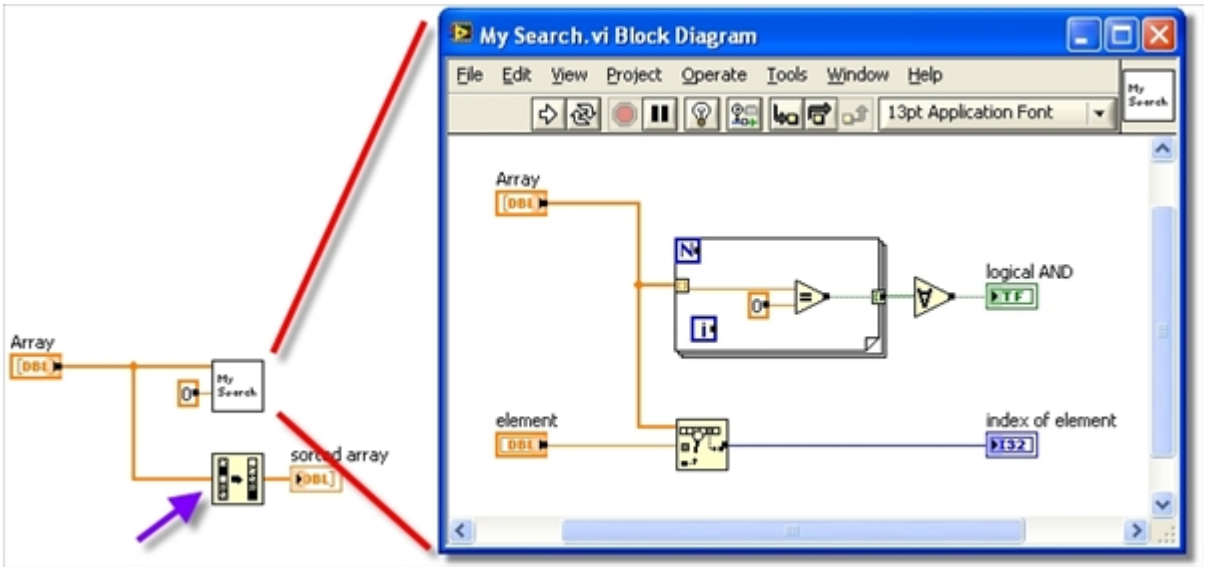


图5：程序中没有必要的数据线分枝

图5 的程序只是一个演示，不必追究它到底实现了什么功能。图中的左半部分是主 VI，在这个 VI 中对输入的数组数据 Array 进行了两次操作：一次使用 subVI“My Search” ;另一次使用了数组排序函数。图5

的右半部分是 subVI“My Search”的程序框图。

需要注意的是，主 VI 上 Sort 1D Array 函数那里有个黑点(这个黑店靠近黄色方块的中心，这里看不太清楚，和图6对比一下，就可以发现了)，说明这里做了一次内存分配。这是因为 Array 的数据被同时传递到了“My Search”和“Sort 1D Array”两个节点进行处理。这两个操作可能会同时进行，LabVIEW 为了安全(两个操作对数据的改动不能相互影响，不能同时对一块内存进行读写)，就必须为这两个节点准备两份数据在两份内存中。所以在“My Search”和“Sort 1D Array”两个节点中，如果一个节点用了原来 Array 的内存，另一个节点就需要拷贝一份数据给自己用。

不过，如果看一下“My Search”的程序框图，它其实没有对 Array 数据进行任何改动，主 VI 完全没有比要给“Sort 1D Array”开辟一块新内存。我们只要对程序稍作改动，就可以对此进行优化。图6 是改进后的程序：

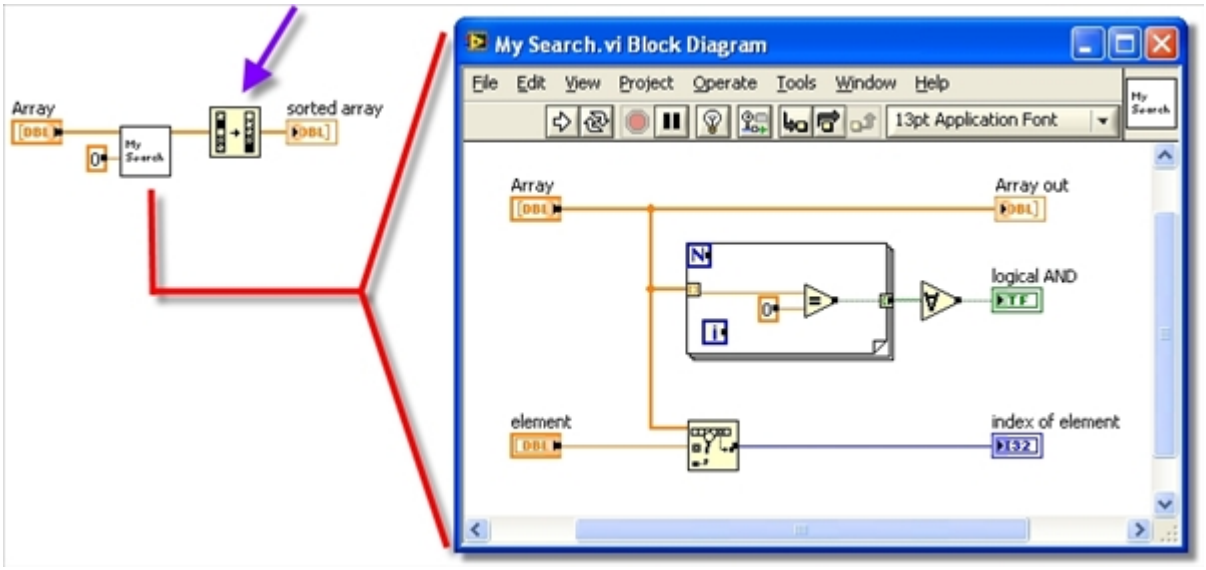


图6：符合数据流风格的主 VI

在改进后的程序中，Array 数据首先传入 subVI“My Search”，然后又传出来，继续传给“Sort 1D Array”函数。这样子看上去好像数据要多到子 VI 中转一圈，但实际上，由于子 VI 中 Array 输入输出是缓存重用的，实际上相当于只是把数组数据的引用传给了子 VI，效率是相当高的。而在主 VI 中，执行“Sort 1D Array”时，LabVIEW 知道输入数据现在是这个节点专用的，改了他也是安全的，于是也可以缓存重用。图六中，“Sort 1D Array”上的那个小黑点就消失了。

图6 中的主 VI，它的优点首先是符合数据流的风格。一个主要的数据从左到右，流经每个节点。这样的程序非常容易阅读和理解。LabVIEW 也更容易对这样的代码进行优化，所以这样风格的程序通常效率也比较高。

有的时候，利用 LabVIEW 的自动多线程特性，书写并行代码，对程序效率有利。比如，程序中某一部分的代码需要较长时间的计算或者读写时间的情况。但是并不是任何时候并行执行都好。并行书写的程序不易理解，容易出错，多线程运行也会带来额外的开销。像图5、图6中的程序，数据量较大，但是并没

有比较耗时的运算操作，或数据读写操作，这样的程序，串行运算比并行效率更高。

LabVIEW 程序中的线程 1 - LabVIEW 是自动多线程语言

一. LabVIEW 是自动多线程语言

一般情况下，运行一个 VI，LabVIEW 至少会在两个线程内运行它：一个界面线程(UI Thread)，用于处理界面刷新，用户对控件的操作等等;还有一个执行线程，负责 VI 除界面操作之外的其它工作。LabVIEW 是自动多线程的编程语言，只要 VI 的代码可以并行执行，LabVIEW 就会将它们分配在多个执行线程内同时运行。

图1 是一个正在运行的简单 VI，它由单独一个一直在运行的循环组成。在此情况下，这个执行循环的线程运算负担特别重，其它线程则基本空闲。在单 CPU 计算机上，这个线程将会占用几乎 100% 的 CPU 时间。图1 中的任务管理器是在一个双核 CPU 计算机上截取的。这个循环虽然在每一个时刻只能运行在一个线程上，但这并不表示他始终不变的就固定在一个线程上。他可能在这个时刻运行在这个线程上，另一时刻又被调度到其他线程上去运行了。(关于这一段，在看完本文第二章：LabVIEW 的执行系统，会有更深刻的理解。)

因此，图1 这个程序最多只能占用两个 CPU 内核 50% 的总 CPU 时间，两个 CPU 内核各被占用一些。

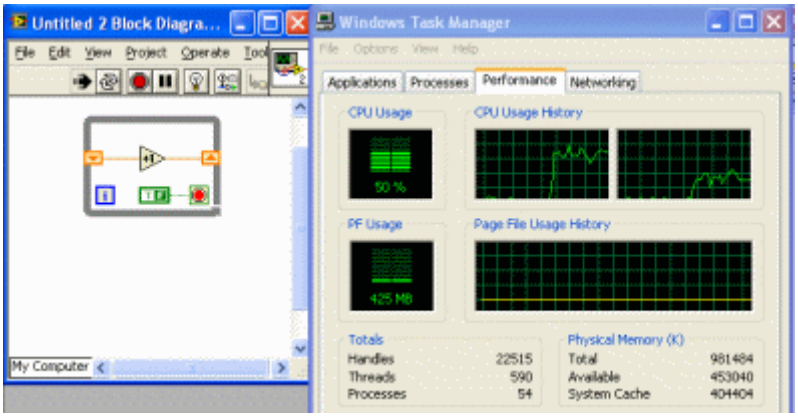


图1：双核 CPU 计算机执行一个计算繁重的任务

图2 是当程序有两个并行的繁重计算任务时的情况，这时 LabVIEW 会自动把两个任务分配到两个线程中去。这时即便是双核 CPU 也会被 100% 占用。

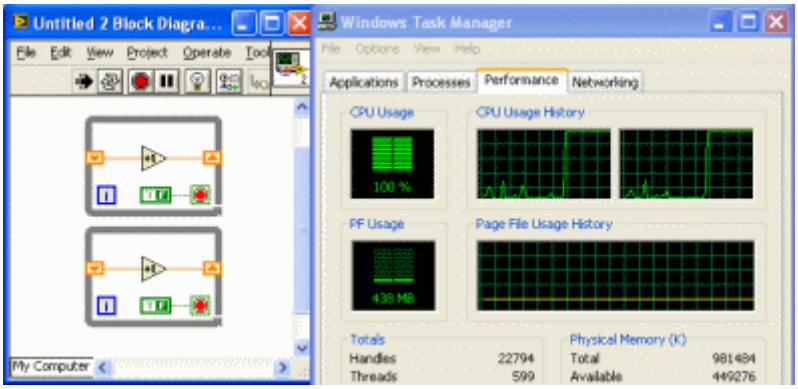


图2：双核 CPU 计算机执行两个计算繁重的任务

从上面的例子，我们可以得出如下两个结论。

1. 在 LabVIEW 上编写多线程程序非常方便，我们应该充分利用这个优势。一般情况下，编写程序时应当遵循这样的原则：可以同时运行的模块就并排摆放，千万不要用连线，顺序框等方式强制它们依次执行。在并行执行时， LabVIEW 会自动地把它们安排在在不同线程下同时运行，以提高程序的执行速度，节省程序的运行时间。今后多核计算机将成为主流配置，多线程的优势会更为明显。

特殊的情况也是有的，即用多线程时，运行速度反而慢。 以后我们再来详细介绍此类特殊情况。

2. 假如有一个或某几个线程占用了 100% 的 CPU，此时系统对其他线程就会反应迟钝。例如，程序的执行线程占用了100% 的 CPU，那么用户对界面的操作就会迟迟得不到响应，甚至于用户会误认为程序死锁了。所以在程序中要尽量避免出现 100% 占用 CPU 的情况。 目前大多数的计算机还是单核单个 CPU 的，因此要避免任何一个线程试图 100% 占用 CPU 的情况(如图1、图2 所示的程序)。

此类问题最简单的解决方法就是在循环内加一个延时。在图1、图2 的例子中，如果在每个循环内加上 100 毫秒的延时，CPU 占用率就会接近为 0。

对于总运行时间较短的循环(假如 CPU 占用总时间不足 100毫秒)就没有必要再加延时了。

在很多情况下，运行时间很长的循环往往都只是为了等待某一个任务的完成，在此类循环体的内部几乎没有耗时较多的、又有意义的运算，所以必须在循环框内加延时。

对于那些确实非常耗费 CPU 资源 的运算(如需要 100% 地占用 CPU 几秒钟甚至更长的时间)，最好也在循环内插入少量延时，从而让 CPU 至少 空出 10% 的时间给其它线程或进程。你的程序会因此而多运行 10% 的时间。 但是由于 CPU 可以及时处理其他线程的需求，比如界面操作等，其他后台程序也不会被打断，用户反而会感觉到程序似乎运行得更加流畅。反之，假如你的程序太霸道了，CPU 长期被某些运算所霸占，而别的什么都不能做，这样的程序，用户是不可能满意的。

还有这样一种情况，比如某些运算可能需要程序循环 1,000,000次，每执行一次仅需要 0.1 毫秒。此时如果在每次循环里都插入延时，即使是 1 毫秒的延时，也会令程序速度减慢 10 倍。 这当然是不能容忍的。这种情况下，就不能在每次循环都加延时了，但可以采用每一千次循环后加上 10 毫秒延时的策略。此时，程序仅减慢 10% 左右，而 CPU 也有处理其他工作的时间了。

在处理界面操作的 VI 中，常常会使用到 While 循环内套一个 Event Structure 这种结构形式。在这种情况下，就没有必要再在循环内添加延时了。因为程序在执行到 Event Structure 时，如果没有事件产生，程序不再继续执行下去，而是等待某一事件的发生。这是，运行这段代码的线程会暂时休眠，不占用任何 CPU 资源，一直等到有事件发生，这个线程才会重新被唤醒，继续工作。

LabVIEW 程序中的线程 2 - LabVIEW 的执行系统

二、LabVIEW 的执行系统

1. 什么是执行系统

早期 LabVIEW 的 VI 都是单线程运行的，LabVIEW 5.0 后才引入了多线程运行。其实，对于并排摆放的LabVIEW 函数模块而言，即使LabVIEW 不为它们分配不同的线程，通常也是“并行执行”的。LabVIEW 会把它们拆成片断，轮流执行：这有一点像是 LabVIEW 为自己设计了一套多线程调度系统，在系统的单个线程内并行执行多个任务。

LabVIEW 中这样一套把 VI 代码调度、运行起来的机制叫做执行系统。现在的 LabIVEW 有六个执行系统，分别是：用户界面执行系统、标准执行系统、仪器 I/O 执行系统、数据采集执行系统、以及其他1、其他2系统。一个应用程序中使用到的众多子 VI 可以是分别放在不同的执行系统里运行的。用户可以 VI 属性面板上选择 Execution 页面，可以在这个页面指定或更改某个 VI 的首选执行系统。

2. 执行系统与线程的关系

LabVIEW 在支持多线程以后，不同的执行系统中的代码肯定是运行在不同线程下的。用户界面执行系统只有一个线程，并且是这个程序的主线程。 这一点与其他执行系统都不一样，其他的执行系统都可以开辟多个线程来执行代码。用户除了可以设置 VI 的执行系统，还可以设置它的优先级。优先级分 5 个档次(暂先不考虑 subroutine)。在 LabVIEW 7.0 之前， LabVIEW 在默认情况下为同一个执行系统下每个档次的优先级开启一条独立的线程;而在 LabVIEW 7.0 之后,LabVIEW 在默认会默认的为每个执行系统下每个档次的优先级开启 4 条线程。当然你使用 \vi.lib\Utility\sysinfo.llb\threadconfig.vi 可以更改这一设置。但是对于普通用户来说最好不要改动它。

在用 C 语言编写多线程程序时，你还要注意不能开辟太多的线程，因为线程开辟、销毁、切换等也是有消耗的。线程太多可能效率反而更差。但是使用 LabVIEW 就方便多了。在使用默认设置的情况下，LabVIEW 最多为你的程序开辟 5 条线程：一条用户界面线程，四条标准执行系统标准优先级下的线程。五条线程不会引起明显的效率损失。

3. 用户界面执行系统

程序中所有与界面相关的代码都是放在用户界面执行系统下执行的。就算你为一个 VI 设置了其他的执行系统，这个 VI 的前面板被打开后，他上面的数据更新的操作也会被放在用户界面执行系统下运行。还有一些工作，比如利用 Open VI Reference 节点动态的把一个 VI 加载到内存的工作，也是在用户界面执行系统下运行的。

前面提到了，用户界面执行系统一个最特殊的执行系统，因为它只有一个线程(我们就给这个线程起名叫用户界面线程好了)。LabVIEW 一启动，这个线程就被创建出来了，而其他执行系统下的线程只有在被使用到时才会被 LabVIEW 创建。

在图1 中的例子中，如果是运行在其他的线程下，都会把我的双核 CPU 占满。原因参考本文第一章(LabVIEW 是自动多线程语言)的图2。但是如果我们把 VI 的执行系统改为用户界面执行系统，那么这两个循环就会运行在同一线程下，我的双核 CPU 其中一个核将被占用 100%，另一个则基本空闲。

图2 是 VI 在运行过程中的一幅截图，虽然程序在单线成下运行，两个循环仍然是并行运行的，两个显示控件的数据会交替增加。

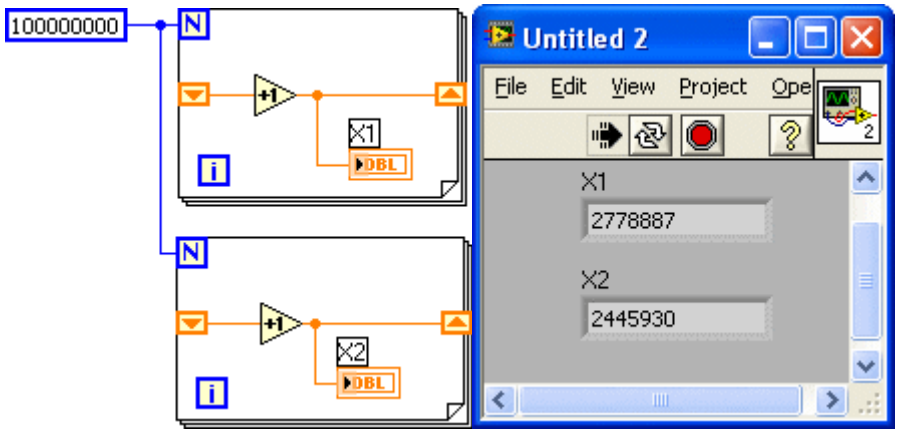


图1、2：在界面线程-单线程下运行的并行任务

因为 LabVIEW 是自动多线程的，如果一些模块不能保证多线程安全，就需要把他们设定为在用户界面线程运行。这样就等于强制他们在同一个线程下执行，以保证安全。具体例子在下一节讨论。

4. 其他几个执行系统

在 执行系统一栏还有其他几个条目可选。

“same as caller”是默认选项，它表示这个 VI 沿用调用它的上层 VI 设置的执行系统。如果顶层 VI 也选择“same as caller”，那么就等于它选择了标准执行系统。

“standard”标准执行系统是最常用的配置方式。

“Instrument I/O”仪器 I/O 执行系统一般用于发送命令到外部仪器，或从仪器中读取数据。这是程序中较为重要的操作，需要及时运行。所以仪器 I/O 执行系统中的线程的优先级比其他执行系统中的线程要高一些。

“data acquisition”数据采集执行系统一般用于快速数据采集。数据采集执行系统中的线程的数据堆栈区比较大。

“other 1”，“other 2”其他1、其他2执行系统没什么特别之处。如果你一定要让某些 VI 运行在独立的线程内，则可以使用这两个选项。

绝大多数情况下，用户使用界面执行系统、标准执行系统就已经足够了。

为了解释清楚，先定义一下要用到的概念。我们把以 .ctl 文件名定义的控件叫做用户自定义控件，把通过拖拽或打开这个 .ctl 文件在 VI 上生成的控件叫做实例。

LabVIEW 的用户自定义控件包括了三种定义形式：打开一个 .ctl 文件，在它上方的“control”下拉条中有三个选择，分别是无关联控件(Control)、类型定义(Type Def.)或者严格类型定义(Strict Type Def.)。

无关联控件是指这个控件与它的实例之间没有任何关联。例如，你制作了一个漂亮的按钮控件保存在 .ctl 文件中。需要用到它时，通过拖拽或打开这个 .ctl 文件就可以在 VI 中生成这个用户自定义控件的一个实例。这个实例一旦生成，就和原用户自定义控件无任何关联了。无论是你修改这个实例，还是修改原用户自定义控件，都不会对另一方产生任何影响。

类型定义控件是指实例控件与用户自定义控件的空间类型是相关联的。比如，你的用户自定义控件是一个数值型控件，那么它的所有实例控件也都是数值型的。如果我们在 .ctl 文件中把用户自定义控件的类型改为字符串，那么它已有的所有实例都将自动变成字符串类型。

有时候，只是类型相关联还不够。比如对于 Ring(Enum，Combo Box)这类的控件来说，如果在用户自定义控件中添加了一项内容(item)，一般总是希望它所有的实例也同时添加这一选项。如果使用类型定义控件，因为控件类型没变，还是 Ring，实例们是不会自动跟随更新的。这时就需要使用严格类型定义控件。选择严格类型定义后，不但实例与用户自定义控件的类型是相关联的，其他一些控件属性，比如颜色等等，也是相关联的。

使用严格类型定义时有一点容易被误解：严格类型定义只是与实例控件相关联，由它生成的实例常量的属性是不与之关联的。实例常量是指通过拖拽或生成常量等方法，在程序框图上生成的一个与 .ctl 文件相关联的常量。比如在 Ring 型用户自定义控件中添加了一项内容，相关的实例常量是不会发生任何改变的。很多人按常理想，认为常量也应当自动更新，但事实上不行。这也是我不采用它做常量定义的原因之一。(参见：在 LabVIEW 中使用常量定义)

LabVIEW 程序中的线程 3 - 线程的优先级

三、线程的优先级

在 VI 的属性设置面板 VI Properties -> Execution 中还有一个下拉选项控件是用来设置线程优先级的(Priority)。这一选项可以改变这个 VI 运行线程的优先级。

优先级设置中共有六项，其中前五项是分别从低到高的五个优先级。优先级越高，越容易抢占到 CPU 资源。比如你把某个负责运算的 VI 的优先级设为最高级(time critical priority)，程序在运行时，CPU 会更频繁地给这个 VI 所在线程分配时间片段，其代价是分配给其它线程的运算时间减少了。如果这个程序另有一个线程负责界面刷新，那么用户会发现在把执行线程的优先级提高后，界面刷新会变得迟钝，甚至根本就没有响应。

优先级设置的最后一项是 subroutine， 它与前五项别有很大的不同。严格的说 subroutine 不能作为一个优先级，设置 subroutine 会改变 VI 的一些属性：

设置为 subroutine 的 VI 的前面板的信息会被移除。所以这样的 VI 不能用作界面，也不能单独执行。

设置为 subroutine 的 VI 的调试信息也会被移除。这样的 VI 无法被调试。

当程序执行到被设置为 subroutine 的 VI 的时候，程序会暂时变为单线程执行方式。即程序在 subroutine VI 执行完之前，不会被别的线程打断。

以上的三点保证了 subroutine VI 在执行时可以得到最多的 CPU 资源，某些作为关键运算的 VI，又不是特别耗时的，就可以被设置为 subroutine 以提高运行速度。比如有这样一个 VI，他的输入是一个数值数组，输出是这组数据的平均值。这个运算在程序中需要被尽快完成，以免拖延数据的显示，这个 VI 就是一个蛮适合的 subroutine VI。

在设置 VI 优先级的时候有几点需要注意的。

提高一个 VI 的优先级一般不能显著缩短程序的运行时间。提高了优先级，它所需要的 CPU 时间还是那么多，但是 CPU 被它占用的频率会有所提高。

高优先级的 VI 不一定在低优先级 VI 之前执行。现在常用的多线程操作系统采用的都是抢占式方式，线程优先级别高，抢到 CPU 的可能性比低级别的线程大，但也不是绝对的。

使用 subroutine 时要格外注意，因为他会让你的程序变成单线程方式执行，这在很多情况下反而会降低你的程序的效率。比如一个 VI 并非只是用来运算，它还需要等待其它设备传来的数据，这样的 VI 就绝对不能被设置为 subroutine。现在多核 CPU 已经很流行了，在这样的计算机上，单线程运行的程序通常比多线程效率低，这也是需要考虑的。

LabVIEW 程序中的线程 4 - 动态连接库函数的线程

四、动态连接库函数的线程

1. CLN 中的线程设置

LabVIEW 可以通过 CLN(Call Library Function Node)节点来掉用动态连接库中的函数，在 Windows 下就是指 .DLL 文件中的函数。用户可以通过 CLN 节点的配置面板来指定被调用函数运行所在的线程。相对于 VI 的线程配置，CLN 的线程选项非常简单，只有两项：界面线程(Run in UI thread)和可重入方式(reentrant)。(新版本 LabVIEW 把这里的 reentrant 改为 Run in any thread 了)

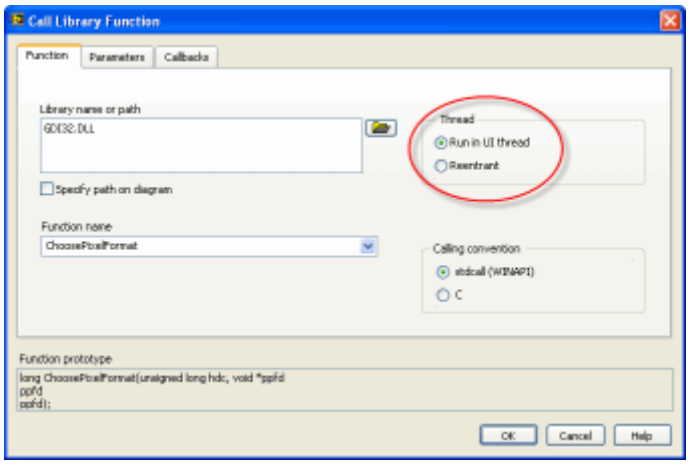


图1：在 CLN 的配置面板上选择函数运行的线程

在 LabVIEW 的程序框图上直接就可以看出一个 CLN 节点是选用的什么线程。如果是在界面线程，则节点颜色是较深的橘红色的;如果是可重入方式的，自节点是比较淡的黄色。

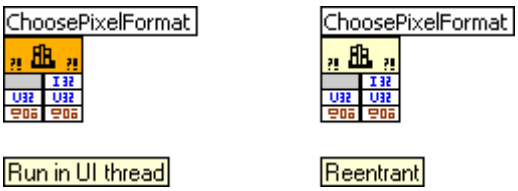


图2：不同颜色表示 CLN 不同的线程设置

2. 如何选择合适的线程

对于在 CLN 中选取何种线程，有一个简单的判断方法。如果你要使用的动态连接库是多线程安全的，就选择可重入方式;否则，动态连接库不是多线程安全的，就选择界面线程方式。

判断一个动态连接库是不是线程安全的，也比较麻烦。如果这个动态连接库文档中没用明确说明它是多线程安全的，那么就要当他是非线性安全的;如果能看到动态连接库的源代码，代码中存在全局变量、静态变量或者代码中看不到有 lock 一类的操作，这个动态连接库也就肯定不是多线程安全的。

选择了可重入方式，LabVIEW 会在最方便的线程内运行动态连接库函数，一般会与调用它的 VI 运行在同一个线程内。因为 LabVIEW 是自动多线程的语言，它也很可能会把动态连接库函数分配一个单独的线程运行。如果程序中存在没有直接或间接先后关系的两个 CLN 节点，LabVIEW 很可能会同时在不同的线程内运行它们所调用的函数，也许是同一函数。对于非多线程安全的动态连接库，这是很危险的操作。很容易引起数据混乱，甚至是程序崩溃。

选择界面线程方式：因为 LabVIEW 只有一个界面线程，所以如果所有的 CLN 设置都是界面线程，那么就可以保证这些 CLN 调用的函数肯定全部都运行在同一线程下，肯定不会被同时调用。对于非多线程安全的动态连接库，这就保证了它的安全。

3. 与 VI 的线程选项相配合

如果你的程序中大量频繁的调用了动态连接库函数，那么效率就是一个非常值得注意的问题了。

我曾经编写过一个在 LabVIEW 中使用 OpenGL 的演示程序(为了演示我们开发的“Import Shared Library 功能”)，对 OpenGL 的调用全部是通过 CLN 方式完成的。由于 OpenGL 的全部操作必需在同一线程内完成，我把所有的 CLN 都设置为在界面线程运行的方式。对 VI 的线程选项没有修改，还是默认的选项。结果程序运行极慢，每秒钟只能刷新一帧图像，CPU 占用 100%。但是作为动画每秒至少25帧才能看着比较流畅。

我开始试图用 LabVIEW 的 profile 工具来查找效率低下的 VI，结果居然查找不到。在 Profile Performance and Memory 工具上显示的 CPU 占用时间只有一点点。这个工具竟然显示不出程序中最耗时的操作在哪里，自然我也对如何优化这个程序无从下手了。后来这个演示程序被搁置了一段时间。

直到有一天我从同事给我的提供的一些信息中得到了启发，才突然想通，这些 CPU 全部被消耗在线程切换中了。我们调用 OpenGL 方法是为每个 OpenGL API 函数包装一个 API VI，这些 API VI 非常简单，程序框图就只有一个 CLN 节点，调用相应的 OpenGL 函数。由于每个 VI 都是在默认的执行线程中运行，而 CLN 调用的函数却是在界面线程下运行的。所以每次执行一次这样的 API VI，LabVIEW 都要做两次线程切换，从执行线程切换到界面线程，执行完函数，在切换回执行线程。

线程切换是比较耗时的。我的演示程序刷新一帧要调用大约两千次 OpenGL API VI，总耗时接近一秒。

解决这个问题，要么把所有 API VI 中的 CLN 都改为可重入方式，但编写程序时要保证所有被调用的函数都运行在同一线程内，这比较困难。比较容易实现的是，把程序中对 OpenGL 操作相关的 VI 也全部都设置为在界面线程下运行。我选择的就是后一种方法。改进后的程序，每秒钟画30帧图像也不会占满 CPU。

由此，我也想通了另一个问题。就是我曾经发现调用 Windows API 函数遇到的错误信息丢失的问题。在调用某一 Windows API 函数返回值为0时，表示有错误发生了。这时你可以调用 GetLastErr 和 FormatMessage 得到错误代码和信息。但是我经常遇到的问题是：前一个函数明明返回值为0，但是随后调用的 GetLastErr 函数却无法查到错误代码。

我想这一定是看上去两个函数是先后被 LabVIEW 调用的，但实际上 LabVIEW 在它们之间还要做两次线程切换才行。错误代码就是在线程切换的过程中被丢失了。解决这个问题的办法也是：把调用这三个函数的 CLN 和调用它们的 VI 全部设置为在界面线程下运行就可以了。

LabVIEW 的运行效率 1 - 找到程序运行速度的瓶颈

一、找到程序运行速度的瓶颈

想要提高程序的运行效率，首先要找到程序运行的瓶颈在哪里。LabVIEW 程序的运行也符合 80/20 定理：20%的程序代码占用了80%的运行时间。如果能找到这20%的代码，加以优化，就可以达到事半功倍的效果。

对于已经编写好的程序，可以通过内存和信息工具来查看程序中每个 VI 运行了多长时间。对程序的效率进行优化，要从最耗时的 VI 着手。

内存和信息工具可以从 LabVIEW 的菜单项 Tools->Profile->Performance and Memory 中启动。图1 是这个工具的界面。

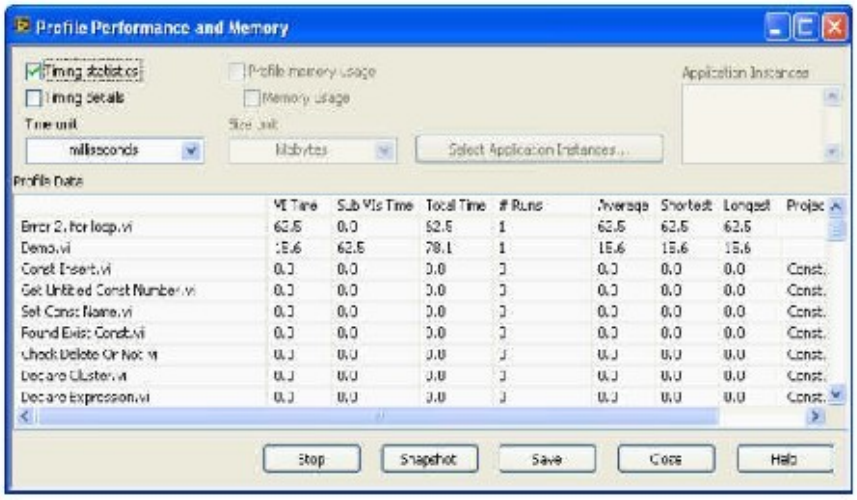


图1：内存和信息(Profile Performance and Memory)工具

在内存和信息工具中会列出一个程序中的全部子 VI。在运行这个程序之前，先按下工具界面上的 Start 按钮，工具就开始为所有的子 VI 进行统计了。你的程序运行结束后，点击工具上的 Snapshot，就会显示出每个子 VI 在刚才的运行中占用了多少 CPU 时间。按照 VI Time 降序排序，排在最前面的几个 VI 就是程序的瓶颈，是需要重点优化的对象。

一个子 VI 占用了大量 CPU 时间，有可能是因为它内部的运算较为复杂，那就需要打开它，对它的算法进行优化。但更有可能的是因为这个 VI 被程序执行的次数太多。这时，你就要考虑程序结构了，是否可以减少这个 VI 的运行次数，比如把它从某些不必要的循环中挪出去，或者拆分这个 VI 的代码，把没有必要循环执行的部分分离出去，挪到循环体外面。

并不是所有的运行效率问题都可以在内存和信息工具中体现出来的。

VI Time 列出的只是子 VI 的 CPU 占用时间，如果你的程序里存在大量的不必要延时，或者程序常

常被某些低速工作(如读写外部仪器，通过网络传输数据等)所阻塞。这样的程序效率肯定也是很低的，但是这一类的低效率因素在内存和信息工具上是体现不出来的。

有些非常耗用 CPU 的操作也无法体现在内存和信息工具上。比如我在《LabVIEW 的线程》第四章中会提到的使用 OpenGL 的例子，由于程序线程设计不当，CPU 被大量消耗在线程切换上。从系统资源管理器看，CPU 被 LabVIEW 占满，在内存和信息工具却看不到任何一个 VI 占用了如此多的 CPU 时间。

在多核 CPU 的计算机上，由于程序可以在多个 CPU 内核上同时执行，某些子 VI 虽然占用的大量的 CPU 时间，如果程序线程设置合理，是可以让这些 VI 不影响到程序的整体效率的。

LabVIEW 的运行效率 2 - 程序慢在哪里

二、程序慢在哪里?

仅仅使用内存和信息工具还不能发现所有程序效率问题的。并且一旦程序的主体部分已经完成，再对其进行修改，成本是比较高的。尤其是涉及到结构性的改动时更是如此：以前做过的测试需要重新做，构建在这个模块之上的代码需要作相应更新。如果时间紧迫，同时考虑到这种代码改动所带来的风险，完全可能在程序完成后就无法再对其性能进行优化了。

所以最有效的编写高效率程序的方法是在设计程序结构的时候，就考虑到可能会影响程序效率的所有因素，直接设计出高效率的程序。而不是在程序完成后，再回头查找程序瓶颈。

下面讨论的是一些常见的运行比较慢的程序代码部分。一个程序运行效率的瓶颈通常就出现在这些部分。所以在设计程序时，对这些部分要格外注意。

a) 读写外设、文件

相对于计算机的中央处理器、内存读写的速度而言，计算机的外围设备的处理和传输数据的速度是非常慢的。比如，GPIB 的传输速率最高也只有 1Mbps，比内存的传输速率低了两个数量级以上。在一个测试应用软件中，造成整个系统效率低下的瓶颈很可能就在于这类数据传输当中，程序的大部分时间都消耗在等待外部数据上了。

b) 界面

界面刷新和等待事件也是比较耗费时间的工作，这是由于人的反应速度远不如计算机引起的。比如你可以设置屏幕上的数据指示控件中的数值以每秒一千次的速度刷新，但是这对于用户来说毫无意义，因为人眼和大脑根本处理不了如此快速的变化。还有，在显示给用户一条信息后，等待用户的后续指令也需要等待一段时间。

c) 循环内的运算

设计循环的时候总是要格外小心些，因为就算一段代码运行得再快，循环个几千，甚至几百万次，耗费是时间也不得了了。所以越是执行次数多的循环，他内部代码的效率对整体影响越大。

d) Global Variable

全局变量不但会破坏 LabVIEW 的代码风格，并且它的代码读写速度也是特别的慢。

e) 子 VI

使用子 VI 是会有一定开销的，但是我们在其它文章(LabVIEW 程序的内存优化)里曾经讨论过，使用子 VI 利大于弊。从这一点来说，子 VI 使用得越多越好。不过需要注意的是，动态调用子 VI 的速度是非常慢的。因为他需要先把被调用的 VI 从磁盘装入到内存中，然后才能运行。而且，装载 VI 的工作一定是在界面线程(LabVIEW 的执行系统)中执行的。如果被动态调用的 VI 太大，就会迟滞界面刷新，影响用户的感觉。

f) 调试信息

这一条对于已经做成可执行文件的程序是没有意义，因为 LabVIEW 在把 VI 转换成可执行文件的时候，一定会去除调试信息的。但是还有相当一部分程序是以 .vi 文件的格式，直接在 LabVIEW 的编译环境中运行的，去除调试信息可以让这种程序降低约 50% 的 CPU 占用时间和内存。

g) 多线程和内存使用不当

LabVIEW 是自动多线程运行的，并且自动开辟、回收内存空间。这意味着对于 LabVIEW 初级用户来说，可以不去关心有关线程和内存的问题。但是对于高级用户而言，需要追求更高的效率，还是需要考虑多线程和内存对程序的影响的。

LabVIEW 对多核 CPU 的支持

以前，在计算机领域有个摩尔定律，是说每一年半，CPU 的主频都会提高一倍。但是近几年这个定律在 CPU 主频上已经失效了，我 4 年前用的计算机 CPU 主频是 2G，我前几天换了一台新电脑，CPU 主频还是 2G。

现在主要两个 CPU 生产商都意识到单纯通过提高处理器主频来提升性能的办法行不通了。他们的新策略是通过增加 CPU 的内核来提升系统整体性能。

现在双核 CPU 是商用电脑的主流配置，也有高端电脑采用了四核 CPU。Intel 更是宣称他们用不了5年就会做出有 80 个核的 CPU 来。

多个 CPU 同时工作，效率固然是高。但是，为了充分发挥多核的优势，为了发挥多核的威力，还要你的软件针对多核进行一定的优化才行。首先，你的程序至少是多线程运行的。

使用常用的文本语言，比如 C++ 编写一个多线程的程序并不是一项简单的工作。除了要非常熟悉 C++ 的基本编程方法，程序员还需要了解 Windows 多线程的运行机制，熟悉 Windows API 的调用方法，或者 MFC 的架构等等。在 C++ 上调试多线程程序，更是被许多程序员视为噩梦。

但如果使用 LabVIEW 编写多线程程序，情况就大为不同了。LabVIEW 是自动多线程的编程语言，LabVIEW 程序员可以不需要了解任何与多线程相关概念与知识。只要他在 VI 的程序框图上，并排放上两段没有先后关系的代码，LabVIEW 就会自动把这两段代码放在不同的线程中，并行运行。而在多核 CPU 的计算机上，操作系统会自动为这两个线程分配两个 CPU 内核。这样就有效地利用了多核 CPU 可以并行运算的优势。LabVIEW 的程序员不知不觉中就完成了一段支持多核系统的程序。

有操作系统来分配 CPU 也许效率还不是最高的。

比如我现在有这样一个程序(图1)，有数据采集、显示和分析三个模块。三个模块是并行执行的。我的电脑是双核的，于是操作系统分配 CPU0 先做数据采集，CPU1 先做数据显示，等数据采集做完了，CPU0 又会去做数据处理(图2)。数据处理是个相对任务较为繁重的线程，而电脑一个 CPU 做数据处理时，另一个 CPU 却空闲在那里。这种负载不均衡就造成了程序对于整体系统的 CPU 利用率不高。

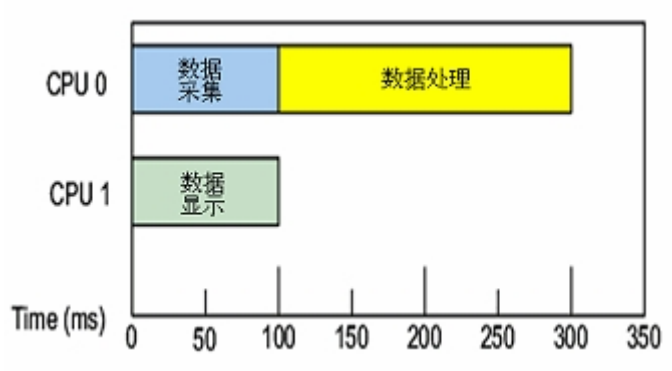


图1, 2：操作系统为多线程程序自动分配 CPU

对于效率要求极为苛刻的程序，还需要更高效的解决方案。LabVIEW 8.5 提供了一种解决方案，就是利用它的定时结构来有程序员人为指定 CPU 的分配方案。

定时结构包括定时循环结构(Time Loop)和定时顺序结构(Time Sequence)，他们的主要用于在程序中精确的定时执行某段代码，但是在 LabVIEW 8.5 中它们又多了一个新的功能，就是指定结构内的代码运行在哪一个 CPU 上。在图3中，定时顺序结构左边四边带小爪的黑方块所代表的接线柱就是用来指定哪一个 CPU 或 CPU 内核的。

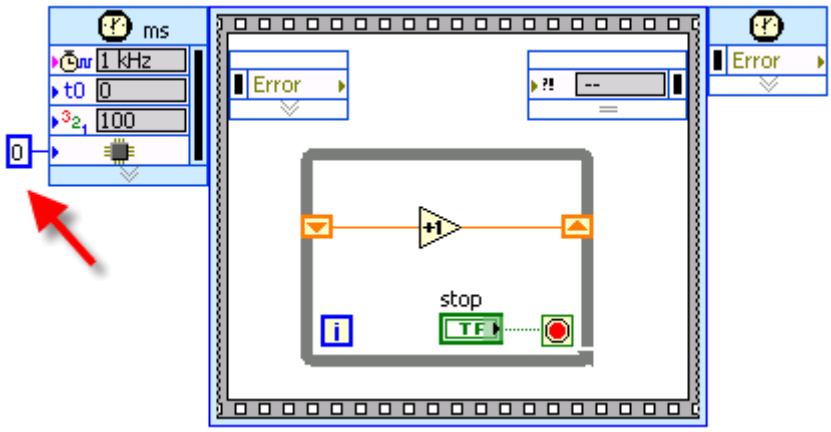


图3：一个时间数序结构

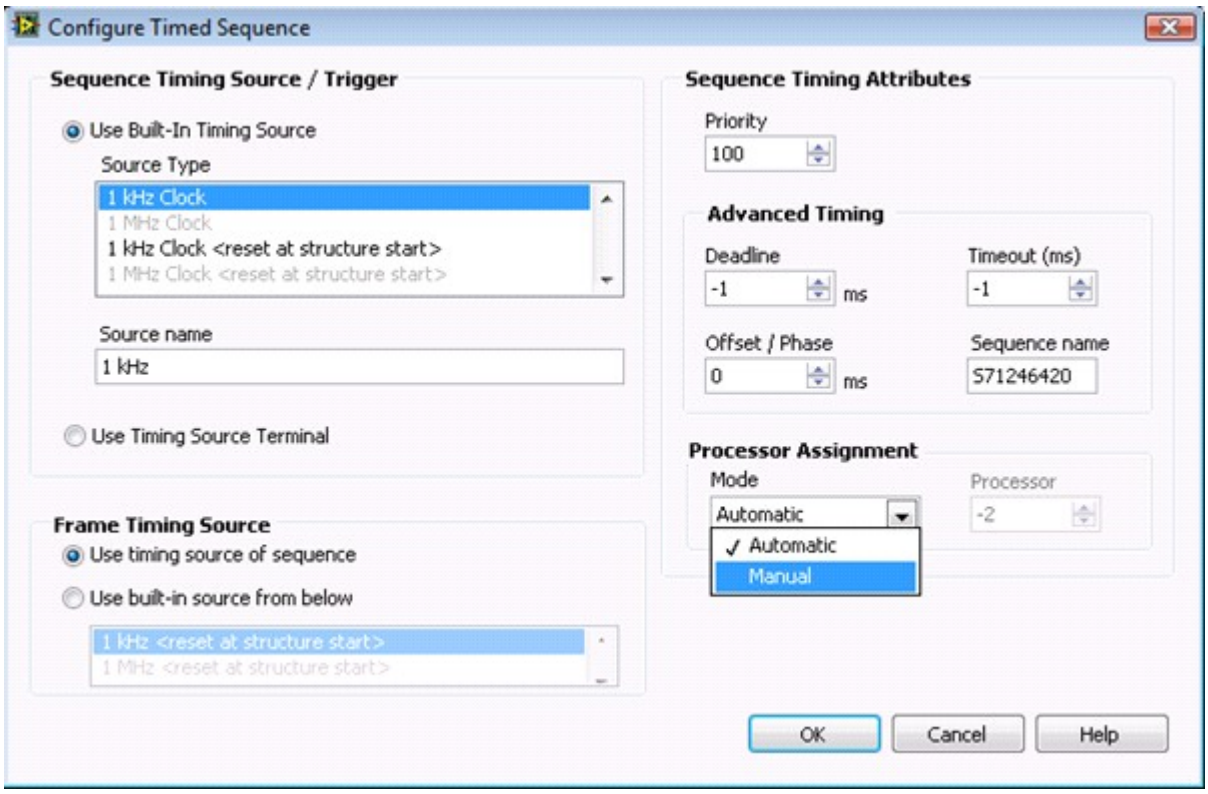


图2：时间顺序结构的输入配置面板

这个 CPU 设置可以在配置面板(图2)中静态的指定好，也可以像图1这样，在程序运行时指定。执行图1所示的程序，在0和1之间切换结构内代码运行的 CPU，就可以在系统监视器中看到指定的 CPU 被占用的情况了。

还是以刚才那段程序为例，这一次我手工为每个任务指定他们运行的 CPU。

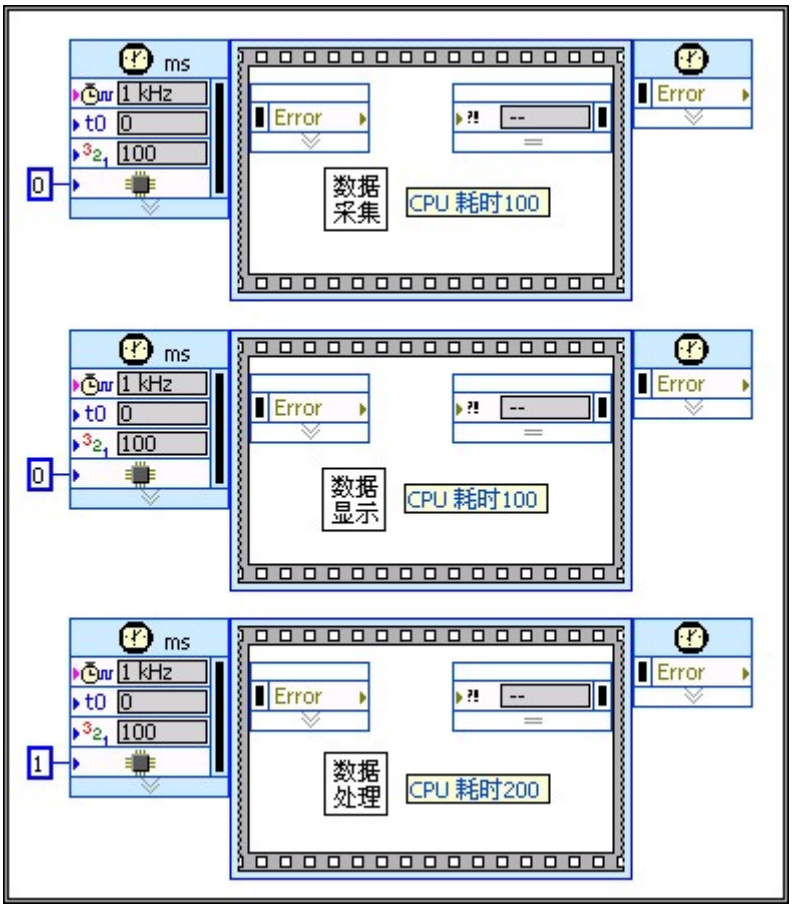


图4：手工指定每个任务运行的 CPU

这样一来，两个耗时较少的任务占用同一个 CPU，耗时较多的任务单独占用一个 CPU。不同 CPU 被分配到的任务比较均衡，程序整体运行速度大大加快，如图5所示：

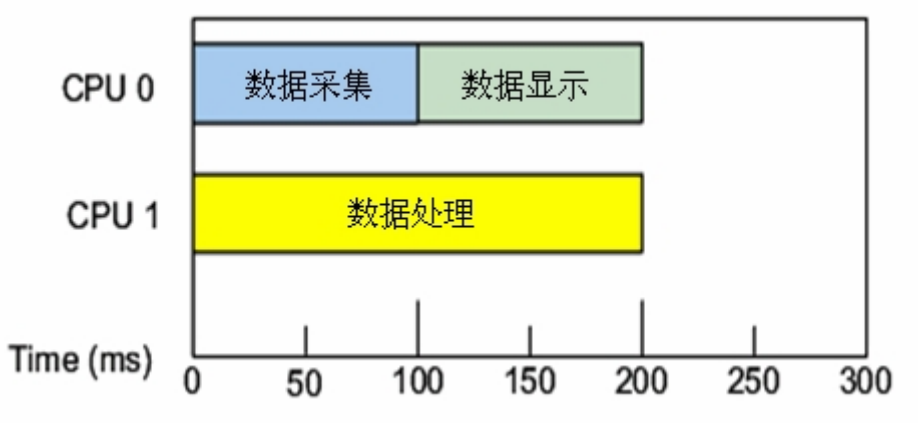


图5：两个 CPU 负载均衡

VI Server (VI 服务器)

VI Server 是指通过程序调用 LabVIEW 提供的一些服务功能的技术。使用 VI Server 可以完成如下的功能：运行远程的 VI 程序;通过互联网运行程序、传输数据;程序运行时调整 VI、控件的某些属性，主要用来调整界面;编程来创建或修改 VI;搭建插件框架式程序，等等。

VI Server 的服务端就是 LabVIEW，客户端可以通过三种方法调用服务端提供的服务：

ActiveX，LabVIEW 提供了 ActiveX 接口。在其他编程语言中，比如 VB，VC++ 中，可以通过 ActiveX 调用 LabVIEW 提供的服务。

TCP/IP，用于远程机器，通过 web 服务来调用 LabVIEW 提供的功能。

LabVIEW 编程，这是最常用的方式，也被称为 VI Scripting。通过 Property Node，和 Invoke Node 暴露出来的属性方法，在 LabVIEW 程序中就可以调用这些服务。VI Scripting 最常见于界面需要在运行时改变的程序，和动态运行某个 VI 的情况下。

后台任务

有一些任务在运行的时候，并不需要与用户交互。它们通常在不打扰用户其它工作的时候默默的执行。这样的任务叫后台任务。与之对比，前台任务就是用户看得见的。

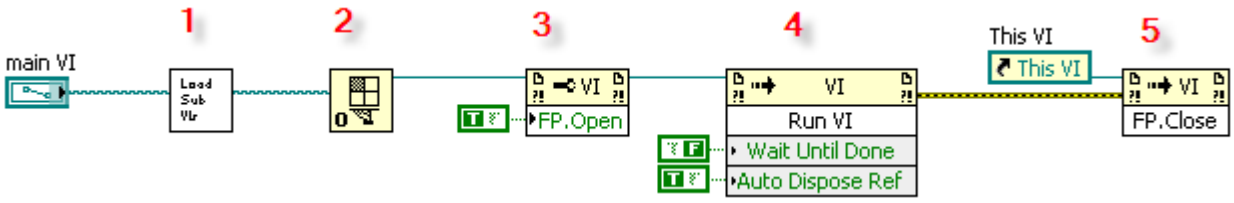
比如，一个文本编辑软件，帮助用户上编辑文字的任务都是前台执行的。如果这个软件做得好，最好还有自动保存文件的功能，这样在系统突然崩溃之后，不至于所有用户做的改动都白费。自动保存这个功能一般都被作为后台任务，他在执行时，不打扰用户其它工作。

后台任务的应用还有很多。在 LabVIEW 中实现后台任务的方法，就是使用 VI Scripting 技术，动态把负责后台的任务在新的线程里启动起来。下面以一个比较常用的功能来说明后台任务是如何实现的。

大型 LabVIEW 程序启动的时候可能会比较慢，这是因为程序在它所有的子 VI 都被装入内存之后才能执行。把大量子 VI 装入内存可能要花费几秒钟的时间，如果这段时间，什么都不显示给用户，用户可能会以为是程序出错或者死机了，从而做出一些错误的操作。好的做法是，在启动程序的时候立刻显示给用户一个提示界面，告诉用户程序正在装载，已经装载的进度。同时把所有的子 VI 装入内存，然后再启动程序的主界面。

在程序显示启动画面这个阶段，当子 VI 全部装入内存后，启动画面仍然显示在前台，而程序会再启动一个后台任务，把主程序运行起来。之后，启动画面程序把自己关闭，而主程序界面也从后台跳到前台来了。

启动画面的程序代码如下图所示：



它由5个部分组成：

第1部分负责把主程序的子 VI 都装入内存(用 Open VI Reference 函数打开这些子 VI 就可以把它们装入内存)。这部分与后台任务无关，不细说了。

第2部分 Open VI Reference 函数打开主程序的主 VI，它的输入是主 VI 的地址。后续使用到的针对这个 VI 的 Scripting 函数，都依赖于在此打开的 VI 的引用。

第3部分设置主 VI 的 Front Panel:Open 属性。这个属性值为真的时候，主程序的界面被显示出来。在我们这个程序中，主程序最终是要跳到前台来与用户交互的，所以需要显示他的界面。有些后台任务，比如自动保存任务，没有交互界面，就不需要设置此属性。

第4部分调用主 VI 的 Run VI 方法。调用这个方法之后，主程序开始运行。目前主程序还是作为后台

任务在运行。这里需要注意的是 Run VI 方法的两个参数：

Wait Until Done 参数如果设为真，启动画面程序运行到这里就会暂停下来，一直等到后台任务结束，才继续执行后面的代码。但是，作为启动画面程序，我们显然不希望出现这种情况。我们需要让前台任务和后台任务同时执行，所以这个参数在本程序中被设为“假”。后台程序在另一个线程中被执行，它的结束与否不影响当前程序的执行，我们的启动画面程序会立即去执行下一条语句。

用 Open VI Reference 函数打开的引用必须被关掉，否则，VI 被装入内存，却从来不被卸除出内存，就会引起内存泄漏。Auto Dispose Ref 参数用于决定由谁来关闭动态运行起来的 VI 的引用，是前台任务，还是后台任务自己。真对于我们的例子，后台任务是主程序，我们希望用户关闭它的同时，把它从内存中卸除。所以实例中这个参数的值是“假”，后台程序结束时自动释放引用。对于那些没有界面的后台任务，它们的启动、停止都需要由前台程序来控制。这是这个参数就需要被设为“真”。前台程序还需要这个引用来停止后台任务、卸除后台任务的 VI。

第5部分是对启动画面 VI 自身进行操作的。程序运行到这里，主程序已经运行起来，而启动画面程序的使命也已完成，应当让位给主程序了。启动画面程序在此调用 Front Panel:Close 把自己关闭。

在 LabVIEW 中实现 VI 的递归调用

LabVIEW 中使用递归调用不是很方便。不过递归并不是编程必须程序结构，任何需要使用递归调用的地方，都可以用循环结构来代替。但是在某些情况下，使用递归调用的确可以大大简化程序代码，对缩短编程时间、提高程序可读性都非常有帮助，所以学习一下递归的实现方法还是有好处的。

一、为什么 VI 不能够被静态的递归调用

LabVIEW 不能通过静态调用的方法(把子 VI 直接放到另一 VI 的程序框图上)来实现递归。

对于一个非可重入的 subVI，在每一个时间，这个 subVI 这能被运行一次。LabVIEW 需要借此来保证多线程时的数据安全。对于被递归调用的代码，是需要在它执行到中间的时候，就再次被调用的。所以默认设置下的 VI 不能被静态递归调用。

对于被设置为可重入的 VI，是可以被同时调用多次的，但也不能被静态的递归调用。

除非是通过 VI Server 动态的调用 VI，否则，LabVIEW 是在一个程序被调入内存，开始运行之前就为它的所有 VI 分配好内存空间的，包括数据区。如果一个 VI 不是可重入的，LabVIEW 会在这个 VI 运行时局部变量所在的数据区开辟在这个 VI 所在的空间内;对于可重入的 VI，LabVIEW 把它的数据区开辟在调用者 VI 上，这样就可以保证这个可重入 VI 在不同的地方被同时调用时使用不同的数据区，以防止多线程运行时数据混乱。

因此，可重入 VI 虽然可以被同时多次调用，但是被调用的次数是运行前就确定的。而递归运算时的调用次数是运行时决定的。这样，如果是静态调用， LabVIEW 根本没有办法为提前为参与递归的 VI 开辟好数据区。

二、用动态调用方法实现递归

图1 是一个采用递归算法计算阶乘的例子

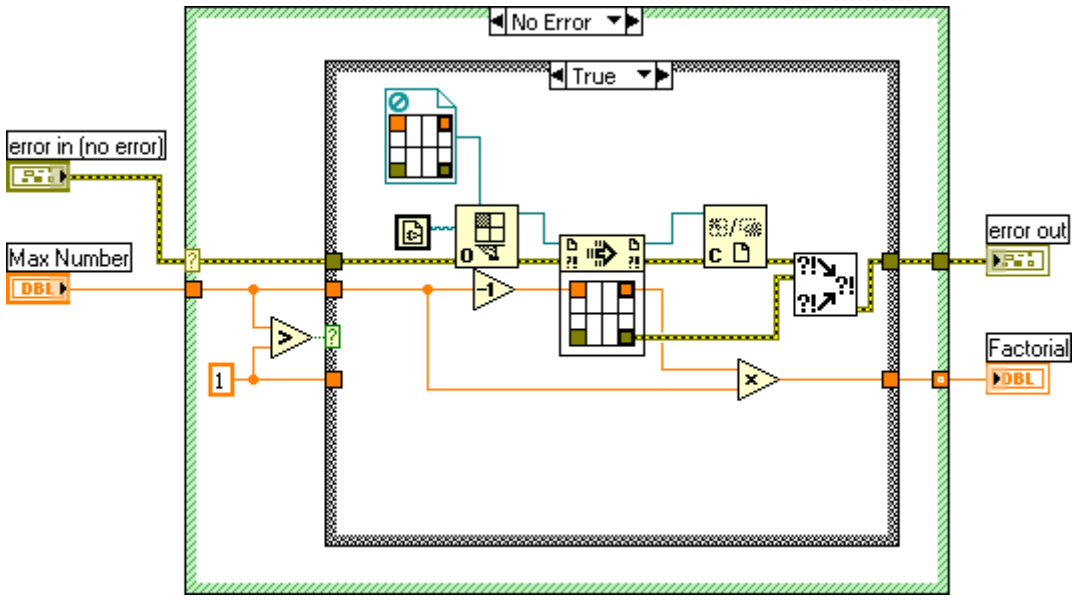


图1：利用递归结构计算阶乘

正如前文说过的，所有的递归都可以使用循环来代替，计算阶乘也可以使用循环结构，但是这里介绍的是使用递归结构的方法。因为 n!=n*(n-1)!，所以我们只要编写一个 VI 实现功能 F(n)=n*F(n-1) 就可以了。

程序中，递归调用 VI 自身的结构由三个 VI 动态调用节点实现：Open VI Reference, Call By Reference Node, Close Reference。这三个节点分别负责动态打开一个 VI(本例中就是这个 VI 自身)，运行这个 VI，再关闭它。

使用 Call By Reference Node 需要在打开 VI 句柄的时候就要知道 VI 连线板(Connector Pane)的布局，因此，我们在用 Open VI Reference 打开 VI 的时候要提供 VI 连线板的布局信息，在例子中就是 Open VI Reference 节点上方的那个常量。

三、使用递归时的几点注意事项

递归调用的退出或结束条件，本例中当输入数据小于1时，就需要结束递归调用返回最底层的值了。如果递归调用的退出条件设置不当，可能会引起程序死循环甚至崩溃。

LabVIEW 中也可以实现 A 调用 B，B 又调用 A 这种用多个 VI 相互调用的递归结构。

参与递归调用的 VI 必须被设置为可重入。

动态调用的需要把 VI 在运行时调入内存，这个过程是比较耗时的。因此递归结构的运行效率远不如可实现相同功能的循环结构，内存占用也会更大一些。决定使用递归结构之前要考虑到这些因素。

VB Script 打开一个 VI

LabVIEW 的一些服务功能是通过 ActiveX 接口提供出来的，我们在其他支持 ActiveX 的语言中可以方便的调用这些服务。比如一个最简单的例子，使用 VB Script 打开一个 VI 的前面板，这样做就行了：

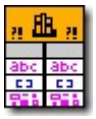
```
Set lvapp = CreateObject("LabVIEW.Application")

Set vi = lvapp.GetVIReference("C:\temp\test.vi")

vi.FPWinOpen = True
```

由于 IE 支持 VB Script，这段代码还可以嵌在 HTML 文件中实现这样的功能：页面上有一处超级链指向一个 VI，点击这个链接，就可以显示相应的 VI。其他浏览器可以使用 JavaScript。

调用动态链接库 1 - 动态链接库导入工具

在 Windows 系统上，动态链接库就是 DLL 文件。调用 DLL 是 LabVIEW 与其它语言混合使用是最重要的一种方法。比如，在一个大的项目中，用户可以用 C++ 语言实现软件的运算部分，并把这些功能构建在 DLL 文件中;再使用 LabVIEW 编写程序的界面部分，并通过调用编写好的 DLL 来调用运算部分的功能。

现在轻易就可以找到完成各种功能的 DLL，LabVIEW 通过调用它们，也几乎无所不能。



在 LabVIEW 中，通过 Connectivity -> Libraries & Executables -> Call Library Function Node(CLN)节点来调用 DLL 中的函数。

在函数选板上，它旁边的一个节点是 Code Interface Node，这个 CIN 节点也是用来与 C 语言混合编程用的。这是在 CLN 节点出现以前，LabVIEW 调用 C 函数的方法。现在有了 CLN，可以不再考虑它了。(我个人强烈建议：不要使用 CIN!因为那样会遇到很多问题，但没人能帮你解决。)

在 LabVIEW 中调用 DLL 中的函数，最大的难度就在于把函数参数的数据类型映射为相应的 LabVIEW 中的数据类型。在准备研究 CLN 中的参数如何配置之前，可以先考虑一下这个工具：Tools -> Import -> Shared Library。这个工具专门用作把 DLL 中的函数包装成 VI，升成的每个 VI 中最主要的部分就是一个 CLN 节点。它自动帮你把函数的参数都已经设置好了，有了这个工具你就不需要再费脑筋去考虑数据类型匹配的问题了。

这个工具的第一个版本是我开发的。它虽然不算完美，但大多数情况下也够用了。如果你有现成的 DLL，打算在 LabVIEW 中使用，首先应该考虑的就是用这个工具，把 DLL 中所有函数都包装成 VI。再使用起来，就方便多了。

某些特别复杂的情况下，Import Shared Library 这个工具可能无法处理，或者包装出来的 VI 不令人满意。这时就需要编程人员手工做一些设置或修改了。因此，高级用户了解这些数据类型的匹配也还是有必要的。我会在后续文章里对此做一详细的解释。

LabVIEW 在 8.0 版对控件和函数面板作了一次较大调整。LabVIEW 功能越来越强大，控件和函数面板上的东西越来越多。如果增加面板的嵌套深度，用户每次选取面板上的一个控件和函数都要多点几下鼠标，而且对不熟悉位置的东西找起来也相当费劲；如果扩充每一个面板上的容量，一个面板上图标太多，用户会眼花，也不利于查找。所以 LabVIEW 8 调整了面板的显示方式：最顶层面板所有栏目都以文字的方式显示，纵向排成一列。其中第一个栏目是默认就展开的，可以直接看到次级面板的内容；其它栏目都收起，鼠标挪上去，才看得到它里面的内容。

我现在用的是 LabVIEW 8.2 版，它的函数面板看上去是这样的：（在程序框图的空白处点击鼠标右键）

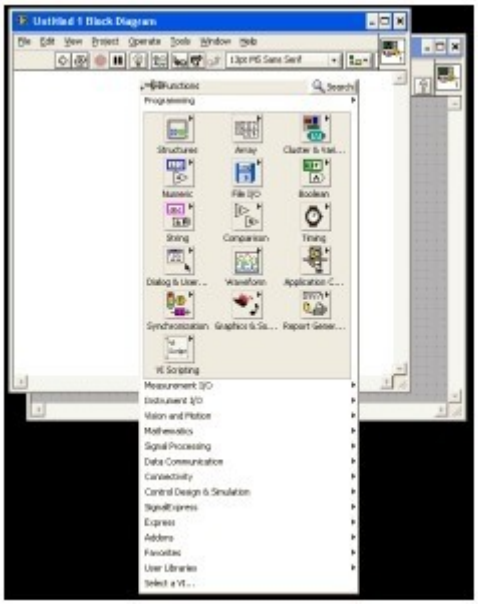


图1：用鼠标右键弹出函数面板

函数和子 VI 被分为几大类，每个类的名字被列在弹出的菜单上，其中最前面一个类是展开的。因为第一个分类中的函数最常用。

但是，有些人可能最常用的函数是在其它分类中的。这样可以调整一下，比如最常用的是 Express VI，就可以把它最常用挪到最前面来。

用鼠标点弹出菜单左上角的图钉，就可以吧弹出菜单固定住。固定后的菜单每一项左端有两个竖线，和一个三角。点击三角可以展开和收缩这个类里的图标，鼠标放到两个竖线上，鼠标就会变成带箭头的十字花。这是就可以按下鼠标拖动这个条目了。点在 Express 项目的竖线上，然后把它托到最上面。如图2、图3所示：

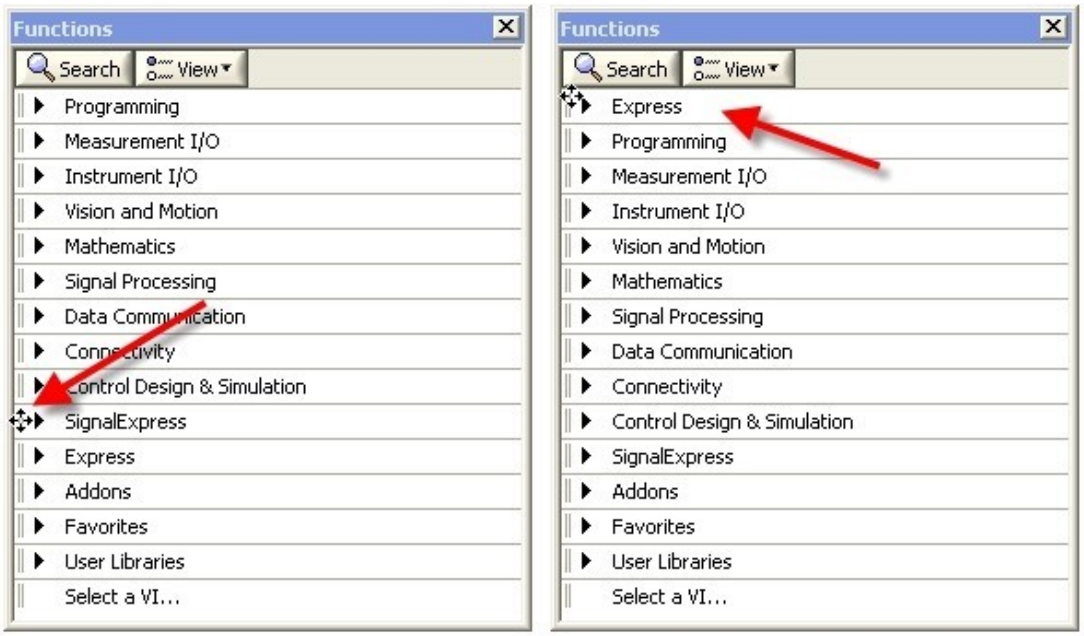


图2, 3：鼠标点在连个竖线上可以拖动这个项目

从此以后，Express 就在最上面了。看图4，再在程序框图上点鼠标右键，弹出的函数面板，最上面一栏展开的就已经变成 Express 了。

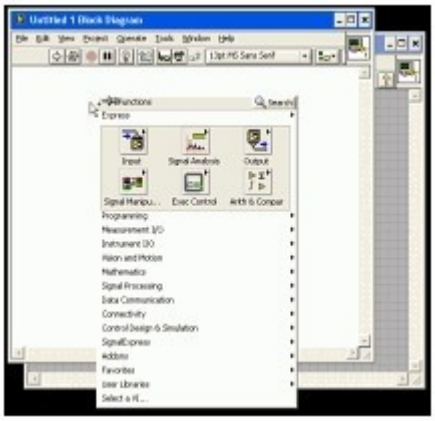


图4：新的函数面板

你可能会发现，自己的 LabVIEW 在鼠标右击程序框图后，只显示出几个分类，其他的类别统统都缩了起来。用户是可以自己制定显示或隐藏哪些分类的。在函数或控件面板钉住的状态下（如图2这种状态），点击面板最上方的“View”按钮，就会出现让你更改显示或隐藏分类的菜单。

如果你觉得 LabVIEW 的面板布局不是很合理，你要用的东西分散在不同的类别里，用哪个坐首选项都不太方便。那也没有关系，在 LabVIEW 8.5 中又多了一个类别，叫 Favorite （我的最爱）。在函数或控件面板钉住的状态下，右键点击其它类别的函数，或子面板标题，弹出菜单上有一项就是加入到 Favorite 中。把你最常用的函数都放到 Favorite 里去，再把 Favorite 移到面板的最上端作为首选项，就可以了。

调用动态链接库 2 - CLN 的配置选项

双击一个 CLN(Call Library Function Node)节点，就会出现它的配置对话框。这个对话框有四页。

第一页是被调用函数的信息。

Library name or path 是 DLL 文件名和路径。在系统路径下的 DLL，直接输入文件名即可，否则需要全路径。在这里知名的 DLL 是被静态加载的程序中的。当调用了这个 DLL 的 VI 被装入内存时，DLL 也同时被装入内存。

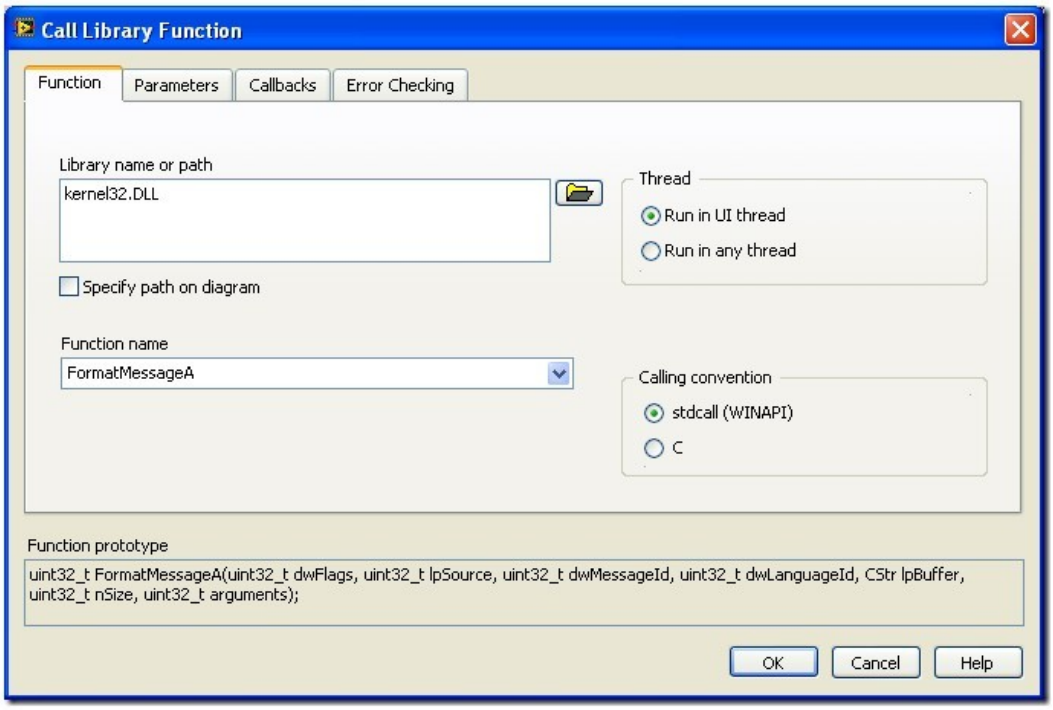
LabVIEW 也可动态加载 DLL。只要把 Specify path on diagram 选上就可以了。选择了这个选项，在 Library name or path 中输入的内容就无效了。取而代之的是，CLN 节点多出一对输入输出，用于指明你需要使用的 DLL 的路径。这样，当 VI 被打开时，DLL 不会被装入内存，只用程序运行到需要使用这个 DLL 中的函数时，才把其装入内存。

Function name 及需要调用的函数的名称。LabVIEW 会把 DLL 中所有的暴露出来的函数都列出，用户只要在下拉框中选取即可。

Thread 栏用于设定哪个线程里运行被调用的函数。它的具体含义可以参考《LabVIEW 程序中的线程 4 - 动态连接库函数的线程》。

Calling convention 用于指明被调用函数的调用约定。这里只支持两种约定：stdcall 和 C call。它们之间的区别在于，stdcall 由被调用者负责清理堆栈;C call 由调用者清理堆栈。这个设置错误时，可能会引起 LabVIEW 崩溃，所以一定要小心。反过来说，如果 LabVIEW 调用 DLL 函数时出现异常，首先就可以考虑这个设置是否正确。

(Windows API 一般使用的都是 stdcall;标准 C 的库函数大多使用 C call。如果函数声明中有类似 __stdcall 这样的关键字，它就是 stdcall 的。)



第二页是函数参数的配置，这是最复杂的部分，留待下次详细分析。

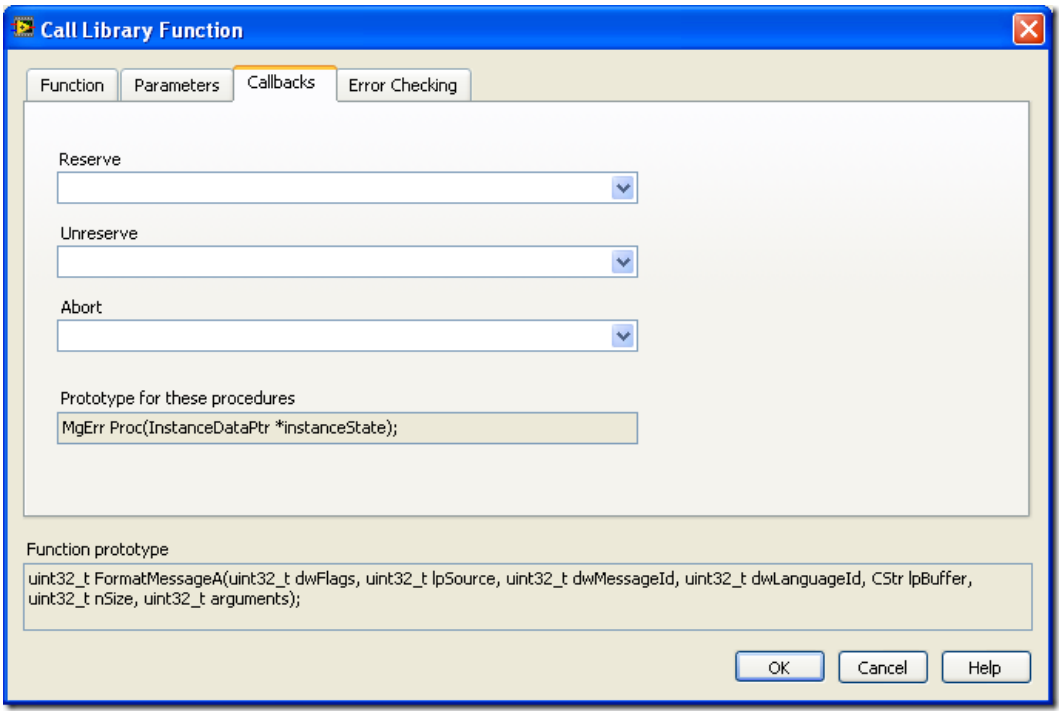
第三页用于为 DLL 设置一些回调函数，可以使用这些回调函数在特定的情形下完成初始化、清理资源等工作。

如果为 Reserve 选择了一个回调函数，那么当一个新的线程开始调用这个 DLL 时，这个回调函数首先被调用。可以利用这个函数为新线程使用到的数据做初始化工作。

如果一个线程使用了这个 DLL，在线程结束时，它会去调用 Unreserve 中指定的回调函数。

Abort 中指定的函数用在 VI 非正常结束时被调用。比如按 Abort 按钮让一个 VI 停止，而不是让他运行完。

这里的几个回调函数必须要由 DLL 的开发者按照特定的格式实现。它的原型就是 Prototype for these procedures 中列出的那个。如果你使用的 DLL 不是专为 LabVIEW 设计的，一般不会包含这样的回调函数。



第四页是错误处理方式，这上面说明写得已经够详细了，我也在补充不了什么了。不过，像我在《用户界面设计 4 - 帮助和反馈信息》里提到的，把帮助文档直接写在界面上的地方，都是极不常用的设置。所以，我们基本上可以不关心这页的设置。

调用动态链接库 3 - 简单数据类型参数的设置

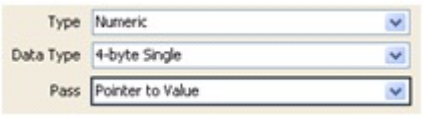
复杂问题先从简单地说起，在 DLL 和 LabVIEW 之间传递参数，最常用的三种数据类型是：数值类型、字符串、数值型数组。这几种类型的参数配置起来还是比较简单的。

1. 数值类型

LabVIEW 多种不同精度的数值类型与 C 语言中的数值类型的匹配是相当直观的，比如 4-byte Single 对应 C 语言中的 float。LabVIEW 自带的例子“LabVIEW 8.5\examples\dll\data passing\Call Native Code.llb”中详细的列出了简单数据类型在 LabVIEW 与 C 之间的对应关系。

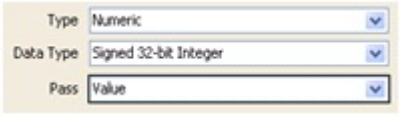
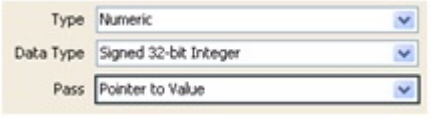
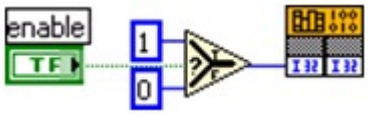
C 语言中经常把指针，或者数据的地址在函数间传递。在32位操作系统中，可以使用int32数值来表示指针。因此，当需要在 LabVIEW 中传递指针数据时，可以使用 I32或 U32 数值类型来表示这个地址类型的数据。但是，64位的程序中，数据的地址只能使用 I64或 U64来表示。这样，如果一个调用了 DLL 函数的 VI，并且函数参数中有地址型数据，使用固定数据类型的数值来表示地址，就要准备两份代码。解决方法就是使用 LabVIEW 中的新的数据类型 Pointer-sized Integer。这个数据类型的长度在不同的平台上会自动使用32位或64位长度。

如果在 C 语言函数参数声明中有 const 关键字，可以选中 Constant 选项。

输入/输出	输入	输出或兼作输入输出
C 语言声明	float red;	float* red;
LabVIEW 中的配置		
LabVIEW 的使用		

2. 布尔类型

布尔类型在 DLL 函数和 LabVIEW VI 之间传递没有专有的数据类型，是利用数值类型来传递的。输入时先把布尔值转变为数值，在传递给 DLL 函数；输出时再把数值转为布尔值。

输入/输出	输入	输出或兼作输入输出
C 语言声明	bool red;	bool* red;
LabVIEW 中的配置		
LabVIEW 的使用		

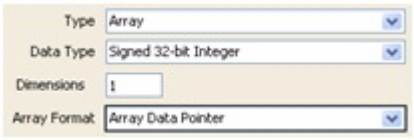
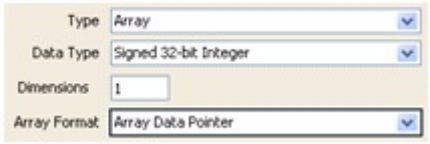
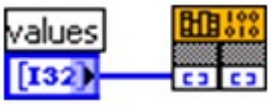
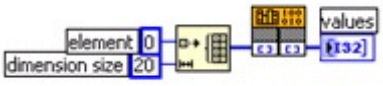
3. 数值型数组

对于数组的传递，LabVIEW 只支持 C 数据类型中的数值型数组。传递数组类型需要注意的的是“Array Format”要选择“Array Data Pointer”。这个设置中还有其他两个选项，像这种带有“Handle”的参数类型都是表示 LabVIEW 定义的特殊类型的。在第三方的 DLL 中不会使用到。

数组参数作为输出值时，要记得为输出的数组数据开辟空间。开辟数据空间的方法有两种：

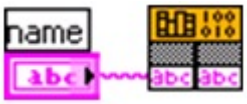
第一种方法，创建一个长度满足要求的数组，作为初始值传递给参数，输出数的数据就会被放置在输入数组的所在的内存空间内。

第二种方法是直接在参数配置面板上进行设置。在 Minimum size 中写入一个固定的数值，LabVIEW 就会按此大小为输出的数组开辟空间。在 Minimum size 中选择函数的其它数值参数，而不是固定数值。这样 LabVIEW 会按照当时被选择的参数的值的大小来开辟空间。

输入/输出	输入	输出或兼作输入输出
C 语言声明	int values[];	int values[];
LabVIEW 中的配置		
LabVIEW 的使用		

4. 字符串类型

字符串与使用与数组是非常类似的，实际上在 C 语言中字符串就是一个 I8数组。

输入/输出	输入	输出或兼作输入输出
C 语言声明	char* name;	char* name;
LabVIEW 中的配置		
LabVIEW 的使用		

调用动态链接库 4 - 结构型参数的设置

C 语言中的结构（struct），在一些简单情况下，可以和 LabVIEW 中的 Cluster 相对应。但是，对于比较复杂的情况，LabVIEW 中的 Cluster 要做适当调整，才能够对应起来。

在讨论结构型参数的映射前，一定要先了解一下字节对齐的概念。我在这里只做一个简单说明，详细内容可以查找相关的专题文章。

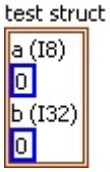
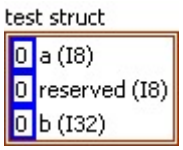
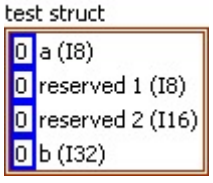
C 语言中的一个结构 typedef struct { char a; int b} MyStct; 结构中的元素 a 所在的地址是：0xAAAA0000，b 的存放地址是与结构的字节对齐设置相关的。如果采用1字节对齐，b 是紧挨着 a 存放的，b 的地址就是：0xAAAA0001；如果采用2字节对齐，b 的存放地址是紧挨着 a 的第一个偶数地址，也就是：0xAAAA0002；如果采用4字节对齐，b 的存放地址是紧挨着 a 的第一个4整数倍地址，也就是：0xAAAA0004……

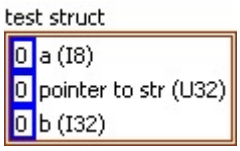
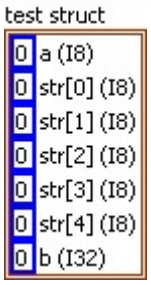
C 语言的字节对齐数可以由 #pragma pack 指令指定，也可以在工程属性里指定。但是 LabVIEW 的 Cluster 只能是1字节对齐的。因此，C 语言中，非1字节对齐的结构与 Cluster 对应时，一定要做适当调整。比如，结构 typedef struct { char a; int b} MyStct; 是2字节对齐的，那么，对应的 LabVIEW Cluster 第一个元素还应该是 I8 型的 a，但是，不能紧接着就放 b，因为 C 语言中，b 的起始地址不是紧挨着 a 的，他们中间还有一个无意义的数据，C 的结构体虽然表现不出来，LabVIEW 中却需要把它考虑进去。

如果是自己编写一个 DLL 给 LabVIEW 使用，为了方便，可以把 C 代码中所有的结构都设为1字节对齐。

C 语言的结构中如果还嵌套了数组，是不能直接对应于 LabVIEW 中嵌套了数组的 Cluster 的。在 LabVIEW 中，只能把数组的元素都拆开来放在 Cluster 中。

下面是一些对应的实例：

C	**LabVIEW**
#pragma pack (1) typedef struct { char a; int b} MyStct; MyStct* testStruct;	
#pragma pack (2) typedef struct { char a; int b} MyStct; MyStct* testStruct;	
#pragma pack (4) typedef struct { char a; int b} MyStct; MyStct* testStruct;	

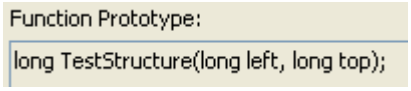
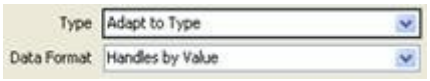
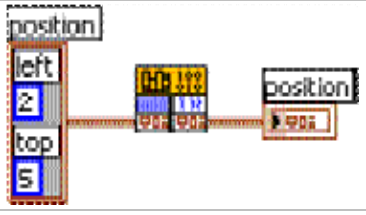
#pragma pack (1) typedef struct { char a; char* str; int b} MyStct; MyStct* testStruct;	
#pragma pack (1) typedef struct { char a; char str[5]; int b} MyStct; MyStct* testStruct;	

上面这个表中有两点需要注意的：

一是，表格中的第四个例子，结构中含有一个指针，LabVIEW 中的 Cluster 只能用一个U32数值（32位系统上，64位系统上使用 U64）来表示指针的地址。不能把指针指向的内容放到 Cluser 中去。后面的章节再讨论当我们在 LabVIEW 中得到了一个数据的地址后，如何从这个地址中把数据拿出来。

第二，上面 C 语言中声明的 testStruct 变量，是指向结构的指针。就是说 C 函数的变量类型为结构的指针时，才能在 LabVIEW 中使用 Cluster 与之对应。CLN 节点的配置面板中，没有一个专门的参数类型叫做“struct”或者“Cluster”，选择“Adapt to Type”就可以了。

如果参数的类型就是结构而非指针，考虑到 C 函数参数的压栈顺序，把一个结构体作为参数传给函数，等价于把结构中每个元素分别作为参数传递给函数。下面是一个例子：

输入/输出	输入	输出或兼作输入输出
C 语言声明	typedef struct{int left; int top;} Position; long TestStructure(Position inPos);	typedef struct{int left; int top;} Position; long TestStructure(Position *pos);
LabVIEW 中的配置		
LabVIEW 的使用		

调用动态链接库 5 - 作为函数返回值的字符串为什么不用在 VI 中先分配内存

Call Library Node 是 LabVIEW 中调用 DLL 函数的节点。如果被调用的函数有一参数数据类型为 char*，用来输出字符串。我们需要在 CLN 中这个参数对应的左侧接线端连进一个字符串，并且输入字符串的长度要保证大于输出字符串的长度。这个输入字符串的内容是没有用的，它只被用作是被开辟的内存，保存输出字符串。否则，会出现数组越界的运行错误，LabVIEW 会莫名其妙死掉。

更糟糕的是，LabVIEW 不会在刚好出现数组越界错误时死掉，而是在之后的某一部确定时候死掉。如果你意识不到自己的程序中有这种错误，或者你有几百个类似的 CLN，那你调试起来可能会类似的。

有人问我，如果函数不是用参数输出字符串而是返回字符串，CLN 返回参数是没有左接线端的。这可咋开辟内存捏?

我打开 LabVIEW 一试，可不是嘛。函数返回字符串的地方根本没法输入任何信息。自己编了一个 DLL 试了试，发现 CLN 是可以正确输出函数返回的字符串的，不需要特别指定字符串的大小。

今天早上起得太早，于是就有点发晕，心想，如果既然 LabVIEW 不需要为函数返回的字符串开辟内存，干嘛非要难为我们为参数输出的字符串开辟内存。否则可以避免多少潜在的错误啊。DLL 函数参数输出字符串是个比较常见的导致程序崩溃的陷阱。

琢磨了半天，脑袋才清醒过来。所谓返回或输出字符串是口头上的语言。换成计算机的语言来解释就清楚了:)

函数返回字符串的情况，实际上是函数返回了一个指向字符串指针。既然是函数返回的，LabVIEW 就可以得到该指针，进而就可以得到它所指的字符串。在 LabVIEW 内部，调用以下 strlen() 得到字符串的长度，开辟一个相应大小的 buffer，再调用以下 strcpy() 就把这个字符串考到 LabVIEW 控件的数据区了。

而参数输出字符串的情况并不是真的输出，而是函数要求输入一个指针。LabVIEW 必须为 DLL 函数提供这样一个指针。而 LabVIEW 自己又不能自动开辟一片缓存就把指针传给函数，因为这时候我们想要的字符串还不存在呢，LabVIEW 没办法知道应该开辟多大的缓存。只好把指定缓存大小的任务交给编程人员了:(

LabVIEW 编程如果不是考虑调用 C 编出来的函数，根本不需要内存分配回收的问题。有了内存分配就是烦啊。

调用动态链接库 6 - LabVIEW 中对 C 语言指针的处理

C 语言函数常有指针类型的参数，有时候，在 LabVIEW 中只能得到一个指向某个数据的指针。比如，在第4节里的一个例子：

```
#pragma pack (1)
typedef struct { char a; char* str; int b} MyStct;
MyStct* testStruct;
long TestStructure(MyStct* tempStct);
```

在 LabVIEW CLN 节点中，就只能返回以整数类型表示的 str 的指针。

在大多数情况下，并不需要在 LabVIEW 中得到指针指向内存的具体数据，对这些数据的操作是在 DLL 的函数中完成的。我们只需在 LabVIEW 中得到这个指针的地址，再把它传递到下一个 CLN 节点就可以了。

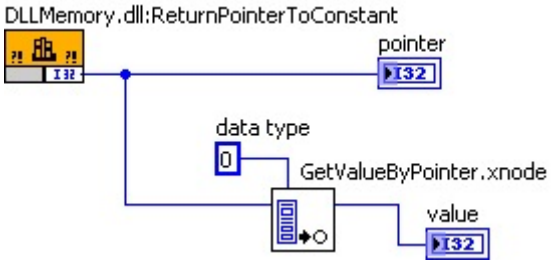
但在某些情况下，我们仍然需要在 LabVIEW 中得到指针指向的内容，这只能借助 C 语言来完成。在上面的例子，我们需要另外写一个 C 函数，把函数 TestStructure 返回的 tempStct 结构中的元素拆开成简单数据类型，作为新的函数的参数（新函数中的一个参数就是 char* str，LabVIEW 可以识别它）。在 LabVIEW 中调用这个新的函数，可以得到这些简单数据类型的数据。

有些函数需要在外部开辟的一块内存中写入数据，LabVIEW 中没有分配内存的操作，也需要再写一个 C 的函数分配好内存，给被调用的函数使用。

这种做法的缺点是针对每个需要得到内容的指针都要做个包装函数，相当麻烦。

一个减少 C 代码的方法是：编写一个 C 函数，负责把指针指向的内存中的数据以数组的形式读出，再在 LabVIEW 中把它们从新组织成合理的数据类型。这种方法其实更复杂，好在 LabVIEW 8.5 中自带的一些 VI 已经做了这个工作。如果你需要，不需要再额外编写代码，直接用 LabVIEW 提供的 VI 就可以了。

[LabVIEW]\vi.lib\Utility\importsl\GetValueByPointer\GetValueByPointer.xnode 就是用来得到指针内容的一个 VI。告诉它指针地址、数据类型，它就会返回正确的 LabVIEW 数据。参见下图中的示例：



DLLMemory.dll:ReturnPointerToConstant 返回的是一个指针，指向我在 C 语言中声明的一个整数常量。把这个指针传给 GetValueByPointer.xnode 并且告诉它数据类型是 I32，GetValueByPointer.xnode 就会得到这个指针指向的内容。

[LabVIEW]\vi.lib\Utility\importsl\ 中还有几个 VI 可以在调用 DLL 时起到帮助作用。比如，对于函数需要使用外部开辟的内存的，就可以使用 DSNewPtr.vi 开辟一块内存，然后把地址传递给这个函数。

需要注意的是，这几个 VI 不是 NI 承诺给用户使用的，所以没有什么文档，需要用户自己研究它们

的用法。

如何调试 LabVIEW 调用的 DLL

问题(Frank)：

我用 Labwindow 编写了一个读文件的动态库, 即向动态库传递文件路径及文件名和某特定字符串,然后通个三个参数返回读到的值, 在 labVIEW 里调用该动态库,结果返回值老是显示打开文件失败, 不知错误出现在那里,另外在 LabVIEW 里如何调试确定传到动态库的参数是符合函数参数格式的呢?该函数在 Labwindow 里调试没有问题.请大侠指点迷津,不胜感激!

回答：

我好久没有用过 CVI 了，计算机上也没有装，不过用 CVI 来调试，应该和用 VC 来调试原理是相同的，步骤也想类似。我就以 VC 为例说明一下。首先在 Debug 模式下 build 出一个 DLL 来。 (VC 7.1 即便是 release 模式下也可以设置断点，单步运行，但别的编译器不一定行。)然后用这个新的 Debug DLL 覆盖原有的 DLL。

关闭 LabVIEW，点击 VC 菜单 Debug->Start (F5)。因为工程生成的是不可以直接执行的 DLL 文件，这是 VC 会弹出一个对话框，问你用什么运行。选择浏览，然后找到 LabVIEW.exe。(这个可执行文件也可以在工程属性中 Debugging->Command 一栏设置。)之后，VC 就会把 LabVIEW 调用起来。

在 VC 中设置好断点。在 LabVIEW 中运行想要调试的 VI。程序会停在你设置断点的地方。

为什么在 CLN 节点中，会自动配置某些 DLL 函数的参数信息

这是一个网友问我的问题，我开始也不了解。后来跟同事打听到了一些信息。

有些 DLL，比如说是使用 LabVIEW 生成的 DLL，再 LabVIEW 中，使用“Call Library Function Node”调用 DLL 中的函数，选择好一个函数，CLN 节点自动就把这个函数的参数信息添加上去了。这样，编程者就不需要再对照着头文件去给它配置参数信息。但是，对于绝大多数 DLL，比如通过一般步骤，在 VC 下编译出来的 DLL，使用 CLN 节点选择了函数后，还要手工为其配置参数信息。它们的区别在哪呢?

那些可以识别参数信息的 DLL，是因为它们把参数定义的信息，以 IDL/ODL 文件格式，嵌入到了 DLL 文件中。LabVIEW，CVI 是可以把这些信息嵌进去的。其它编译器也许也可以把信息嵌入 DLL 中，但是具体如何操作我就没研究过了。

LabVIEW，VB 等编程语言可以识别嵌入 DLL 的参数设置信息，在这些语言下使用这种使用有参数信息的 DLL，更加便捷。

利用 LabVIEW 工程库实现面向对象编程

注意：

我写这篇文章的时候，LabVIEW 8.2 还没有出来。现在 LabVIEW 8.2 本身就以支持面向对象的编程方法，所以这里介绍的方法有点过时。我有时间会再写一篇关于新 LVOOP 的文章。

摘　要：

本文将简要介绍图形化编程语言 LabVIEW 中面向对象的编程思想。并且提出了一种实现面向对象编程具体方法，即利用 LabVIEW 8.0的新特性：工程库，来帮助实现对象的程序设计思想。

关键词：

LabVIEW，面向对象，类，工程库

Implementing Object Oriented Programming in LabVIEW with Project Library

Abstract:

This paper introduces the Object Oriented Programming in LabVIEW, which is also called as GOOP. And it also introduces a new way of implementing the GOOP application: with the help of Project Library, a new feature in LabVIEW 8.0

Key Words:

LabVIEW, GOOP, Class, Project Library

一. 背景

LabVIEW 是一个强大的编程语言，但是随着开发程序规模变大，LabVIEW 程序员可能会觉得对程序越来越难于管理和维护。其根本原因就是 LabVIEW 是面向过程的编程语言，它采用基于数据流的运行方法。而这种程序设计方式在模块划分方面有着天然的缺陷。使用 LabVIEW 编写程序时关注的是按流程完成功能，而不是程序功能模块的划分。因此 LabVIEW 程序划分出来的不同的块之间可能会公用很多子 VI，或全局变量，它们的存在使得程序各个模块无法完全独立，更糟糕的事模块之间的关系可能不为编程人员所察觉。当程序规模大到一定程度，尤其是需要多名开发人员共同参与的时候，编写出来程序会越来越显得杂乱无章，使得程序的调试、维护、和升级都变得非常困难。

解决这一问题的途径就是引入更加抽象化的面向对象的编程方法[2]。通过构造类的方

法，把不同模块之间的数据彻底分离开来，甚至把数据和操作分离开来。这样就保证了不同模块可以完全独立的开发、测试。对某一模块的修改将不会影响到任何其他模块。这样，就可以将一个大的工程分解为可以完全独立开发的多个模块，彻底解决前文所提到的开发困难。

早在1999年，NI 就曾向用户演示过在 LabVIEW 中使用面向对象的编程思想的示例。一些第三方的公司还为 LabVIEW 面向对象编程提供了一些开放工具。但是由于这些工具使用复杂，功能简单，LabVIEW 面向对象的编程思想当时并没有引起用户广泛的注意和重视。

刚刚推出的 LabVIEW 8.0版的一些新特性明显体现出面向对象的编程思想。尽管它仍然没能实现对面向对象的编程的整体支持，但是可以预见，LabVIEW 将在后续的版本中完整的实现对面向对象的编程的支持。

二. LabVIEW 工程库（LabVIEW Project Library）

LabVIEW 8.0的一个重要新特性就是“工程库”，这也是 LabVIEW 向现行对象开发语言过渡的一个重要体现。工程库是一组功能相关联的 VI 或其它文件的集合。工程库与传统的 LabVIEW 的 LLB 文件有着本质的区别。LLB 文件只是将一组 VI 打包存储的一种形式，而工程库与如何存储 VI 无关，它更关注是把功能相关的 VI 按一定结构组合封装，以便于代码的管理和发布。

工程库的一些特性可以帮我们方便地实现面向对象的编程：

1. 工程库的名字也是库中 VI 的名字空间(name space)。

名字空间是 LabVIEW 8.0的一个新特性。在8.0前的 LabVIEW 中无法打开两个文件名相同但内容不同的 VI，这就好比在 C 语言中，一个工程不能拥有两个名字相同的函数。新版本的 LabVIEW 不再有此限制，但是被同时打开两个同名 VI 必须存在于不同的名字空间，也就是在不同的工程库中的同名 VI 才能被同时打开。这与 C++、C#等语言中的名字空间的概念类似。

2. 库中的 VI 有操作安全设置， 每一个 VI 成员可以被设置为公有（Public，可以被库外的 VI 调用）；或者私有（Private，只能被库的成员 VI 调用）。
3. 使用 VI Scripting 技术，可以在运行时方便的得到库的组织结构信息。

VI Scripting 技术也是 LabVIEW 的新特性。利用它可以直接在 LabVIEW 中解析或更改 LabVIEW VI。

三. LabVIEW 面向对象编程的具体实现方法

我们可以把一组相关的数据和 VI 放在一个工程库内，借以实现类的封装功能，但是这种方法不能实现类的继承和多态。

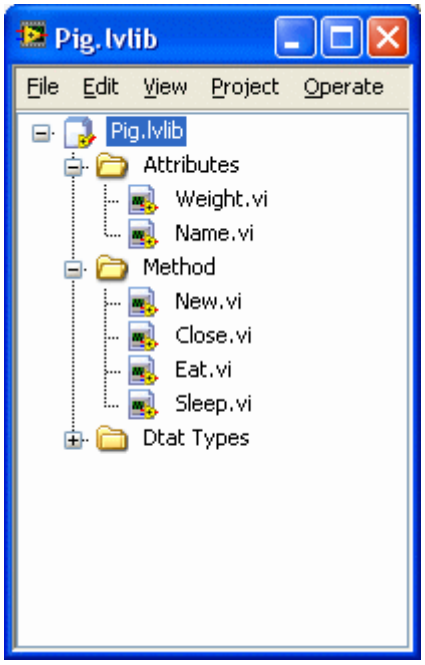


图1：LabVIEW 工程库的结构

1. 工程库的结构

例如，要建立一个表示“猪”的类，我们先要为它新建一个名为 Pig 的 LabVIEW 工程库。然后按一定的分类方法建立文件夹结构，比如将表示数据的 VI 放在 Attribute 文件夹下；把表示动作的 VI 放在 Method 文件夹下。也可以划分两个文件夹分别存放公有 VI 和私有 VI。各种分类组织方法并无本质区别，可凭个人爱好选择。

从数据和操作安全的角度考虑，需要在工程库的属性面板中设置成员 VI 的公有或私有属性。为了维护和使用方便，还应当为库设置适当的版本号、图标等属性。

2. 类的设计

LabVIEW 工程库一般是不能直接就拿来当作一个类来使用的。类是一个抽象概念，在使用时，需要类进行实例化，类的实例才是真正参与工作的。类的每个实例要保存自己的一份数据，而类中的方法则只需存有一份。因此我们需要为类编写一些用来管理数据的 VI，例如图1中的 new.vi。它就相当于这个类中构造函数。

我们可以使用两种方法来为每个实例保存一份数据。

简单的方法是在 new.vi 初始化一个结构（cluster），把所有这个类可能用到的数据都包括在这个结构里。例如，在本例中，可以做一个结构，有一个字符串和一个数字量构成，分别表示我们将用到的名字，和重量。其他类中的成员 VI 都必须使用这个结构作为传入参数，这样就保证了每份实例的数据互不影响。

另一种方法需要借助 C 语言的帮助，比较复杂，但是可以避免把一个大的数据结构作为参数传来传去。我们可以使用 C 编写一套专门处理类数据的 API 函数，生成 DLL 文件供 LabVIEW 调用。具体操作时，用 C 语言为类中所有的数据开辟一块内存空间，然后返回内存地址给 LabVIEW。我们可以在 new.vi 中把返回的内存指针强制转换为自定义的 Data Log File Refnum 数据类型，这样我们还可以为每个类定义一个专用的 reference 类型。其他类中的成员 VI 都使用这个 reference 作为主要参数。需要使用某一数据时，可以调用 C 语言编写

的 API 从内存里读出数据。使用这种方法一定要有一个类似析构函数功能的 VI，释放开始时开辟的内存。这种方式类似于 LabVIEW 中的文件操作 VI。

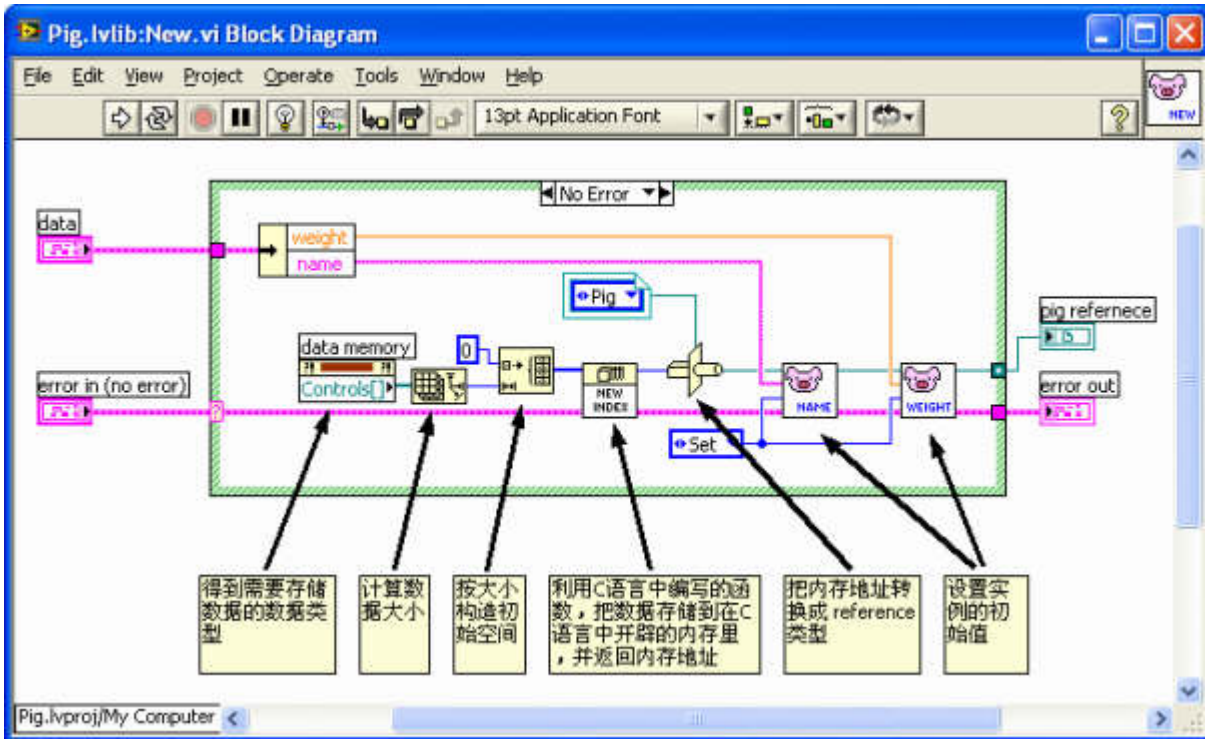


图2：借助 C 语言的帮助实现开辟多份实例

3. 类的使用

类的使用相对来说要简单得多，与面向对象的文本语言的编写方法相类似。基本步骤也是首先调用构造 VI 创建类的实例，然后对类的实例进行操作，操作结束需要调用析构 VI 释放实例占用的资源。如图3所示的例子，用我们在前文中设计的“猪”的类编写的一段程序。程序中我们创建了“两头猪”，然后经过不同的喂养方法，再比较一下他们的体重。

相信读者仅凭代码中的图标就已经可以读懂程序的功能。由此也可见面向对象编程对程序提高可读性的帮助。而使用传统方法编写类似 LabVIEW 程序，由于没有很好的数据封装，在程序框图中数据连线多且杂乱，极易引起错误。

更值得一提的是，LabVIEW 与 C++不同，使用面向对象的编程方法不会引起程序效率的损失。

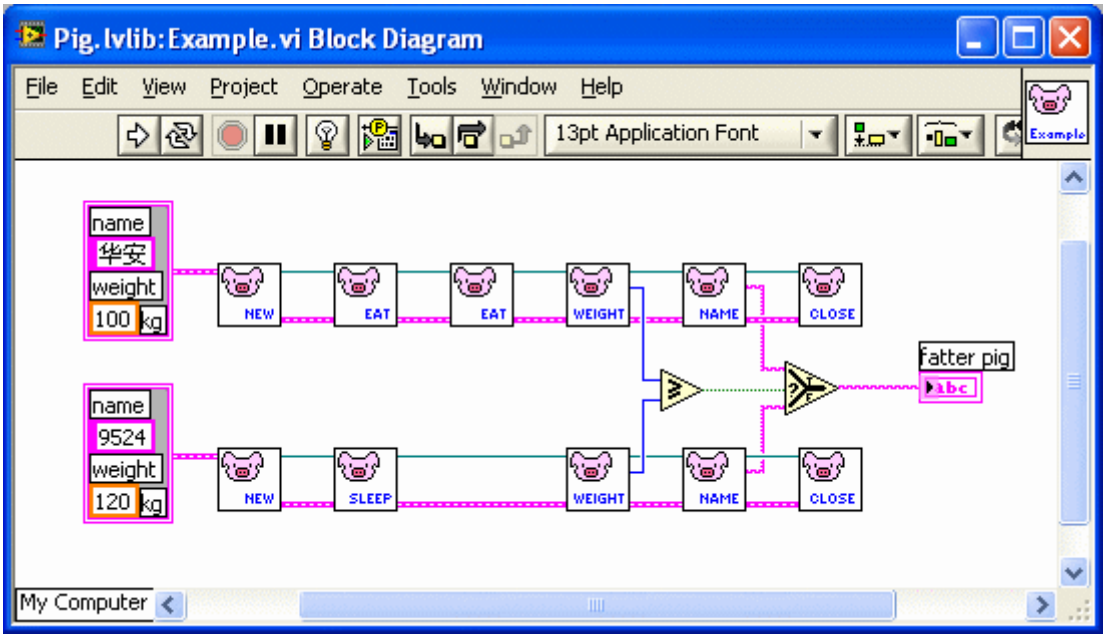


图3：面向对象编程方法的示例

4. 其它方法实现面向对象编程

除了文中提到的借助 LabVIEW 工程库实现面性对象编程的编程方法之外，我们还可以借助于 LabVIEW 8.0的其它一些新特性，比如 XControl 等帮助实现面向对象的编程方法。他们的具体实现方法留待其它文章讨论。

四. 面向对象的方法对 LabVIEW 程序设计的影响

目前，LabVIEW 程序开发的一般流程是先设计和实现顶层 VI，一般来说顶层 VI 也就是程序的主界面。然后自上而下的设计和编写 LabVIEW 程序。面向对象的编程方法由于大大提高了程序模块之间连接和搭建的效率，它使得程序员把更多的精力集中在模块的设计开发上。而不同的模块之间相对独立，可以并行的开发、测试。这就使得 LabVIEW 开发大型程序的效率大大提高。可以预见，随着面向对象的编程方法在 LabVIEW 中的推广，LabVIEW 程序的规模将有一个本质性的飞跃提高。

模块接口 API 的两种设计方案

假如你要设计一个程序模块，它的功能是读写 INI 文件。用户调用这个模块，就可以方便的把信息写入 INI 文件，或从其中读出信息。

你将如何设计这个模块的接口呢?LabVIEW 中常见的方式有两种，第一，为模块的每个方法都做一个子 VI，比如写数值型数据的方法做一个 VI，写字符串的做一个 VI，读字符串的一个 VI 等等;另一种方案：把所有的方法都放到一个子 VI 里去，用户通过一个变量来选择运行哪个方法。

这两种方案各有优缺点。第一种方案符合一般人的思维模式，更容易让用户理解和学会使用。现在 LabVIEW 中处理 INI 文件的模块采用的就是这种方案。每个用户可能用到的方法(甚至是每一种数据类型)，都有一个对应的 VI。维护起来也容易，哪个方法有 bug，到它对应的那个 VI 中去调试就可以了。

但是打开这些处理 INI 文件的 VI，他们调用了一个更底层的模块，这个模块采用的是第二种接口方案。所有对 INI 文件底层的操作，都被放到了一个子 VI(Config Data Registry.vi)里。用输入参数("function")来控制执行不同的功能。

这种方案也有它的好处，我看过一本叫做《软件工程方法在 LabVIEW 中的应用》的书，它的内容用一句话来概括，就是号召大家把模块都写成上述的第二种方案。不过我们先来说一下着第二种方案的弊端。

首先，给外部用户的感觉就不如第一种方案那么清晰易学。如果把所有方法分开成独立的 VI，用户可以只专注学习自己可能会用到的功能对应的 VI;而第二种方案，所有功能在一个接口 VI 里，那就强迫用户把所有功能都要了解一下。

其次，每种不同功能所用到的参数都不尽相同。采用第二种方案，就意味着这个唯一的接口 VI 要包含所有方法时用到的控件(参数)。所以这个 VI 上的控件会比较多。并且，有的控件在调用不同功能时，用途(或者说所表达的意思)不同。这样不但会造成用户学习的困难，在使用时，也非常容易出错。

还有一条，第二种方案的效率在某些情况下非常低下。我们把一个模块提供给用户，但用户不见得会使用这个模块中所有的功能。第一种方案，用户程序是在编译时选择使用模块中的那些方法;而第二种方案是在运行时选择使用什么方法。如果用户只用到一个模块中的一两个功能，采用第二种方案，只用用户用到的方法相关的代码才会被链接到它的程序中;而采用第二个方案，不论用户是否需要，整个模块都会被链接到它的程序中去。

这是因为这几个缺点，造成现在 LabVIEW 提供给用户的库中，几乎都是采用的第一种接口方案。

但是，着第二种方案，一度是 LabVIEW 程序设计中一个非常流行的方法，自然也有他的优点。

其一是更好的解决模块封装的问题。在 LabVIEW 8 之前，LabVIEW 本身不支持面向对象编程，也没有提供对一个模块进行封装的功能。我如果编写一个功能模块给用户，我这个模块中所有的 VI，即便是我只把它当作内部使用，都可以被用户调用。这是很不安全的，因为内部 VI 随时都可能被改变调整，从而引起客户应用程序的错误。如果所有的功能都通过一个 VI 暴露给用户，则用户更容易搞清楚只有这个 VI 他可以用，其它的 VI 都是不能被他直接使用的。并且这样也可以使自己编写的一大堆 VI 看上去也更像是一个模块或组件。

LabVIEW 的另一个问题是，它作为数据流驱动的编程语言，不像文本语言那样可以方便的使用全局或局部变量。在 LabVIEW 中使用全局或局部变量不但效率查，还会严重影响程序的可维护性。我编写的模块，它所用到的内部数据如何组织呢?全局变量既然不好，那就只能考虑使用移位寄存器了。

LabVIEW 程序如果设计的不好，数据在不同节点间传递时会产生很多份拷贝，造成效率低下。为了解决这个问题，最好是我内部使用数据，就不要再在 VI 之间传来传去了。打开 Config Data Registry.vi，你会发现这个 VI 的主体框架是一个只运行一次的循环。凡是这种只运行一次的循环，程序真正想利用的都是循环上的移位寄存器。这个 VI 里的多个移位寄存器都是既无输入又无输出的，它们的功能是用来保存模块的私有数据。

用移位寄存器保存模块的全部私有数据，模块的所有方法都在移位寄存器之间完成。这样数据始终在一个 VI 内，避免了数据在不同 VI 之间传递可能会引起的复制。这是很长一段时间内都相当流行的 LabVIEW 程序模块设计思路，不过我觉得也许现在可以放弃这个方案了。

首先，这个实现方法只适合功能简单的小模块，模块的大部分代码都放到一个 VI 中。如果模块数据功能较多，还用这个方法编出来的 VI 就很难读懂，没法维护了。Config Data Registry.vi 虽然功能并不复杂，但代码已经不那么清晰易懂了。

如果这个模块在程序中只有一个实例还好办，若要支持多个实例，那数据部分就要设计个更为复杂以确保模块不同实例之间的数据不会混乱。

最重要的是现在 LabVIEW 自身已经开始支持面向对象的功能了。在 LVClass 中，既可以有数据，也可以有方法;方法可以被定义为是私有的或共有的;另外之支持继承、多态等。所有这些都为功能模块的封装和接口提供了更好的解决方案。与其费尽心机的自己想办法把格模块包装的更合理，不如直接利用 LVOOP 已有的功能。把自己的的模块都设计为 LVClass。

LabVIEW 面向对象程序设计的简介

LabVIEW 的数据流驱动模式，与面向过程的编程思想有些类似。它们都是把程序看成是一组过程或功能的集合，LabVIEW 利用数据流控制这些功能执行的顺序。由于开发者可以随意的修改、调用这些功能模块，在程序开发的后段，模块之间的划分会变得模糊，依赖关系也变得无序。这种方式就不再适合大型程序的开发。

面向对象的编程思想是专为解决这个问题提出来的。面向对象的编程思想大大提高了编程时的灵活性和可维护性。现在的大型程序中几乎没有不基于面向对象编程思想的。LabVIEW 为了适应这一趋势，也从 8.2 版本开始引入了面向对象程序设计的思想。

面向对象有三大特征：封装、继承和多态。

封装是把高度相关的一组数据和方法组织在一起，形成一个相对独立的类。外部程序只能通过严格定义好的接口访问类所允许公开的数据和方法;而对于不需与外部发生联系的数据和方法，类会把他们隐藏和保护起来。这样就避免了编程过程中，函数模块常常被到处滥用以至于难以维护的弊病。(假如，我们的程序是模拟多只小狗的日常生活的。在设计程序时，就可以把他们抽象归为“狗”类。这个类包括了一些属性，如年龄、皮毛颜色、名字等等;还可以包含一些方法，即狗的行为，比如进食、移动、叫等。)

初一看 LabVIEW 中的 Class 就会发现它很像 Cluster，或许它就是在 Cluster 基础上发展来的。C++ 中的 Class 也是在 Struct 的基础上发展来的，而且，在 C++ 中，除了函数默认的权限不同，Class 和 Struct 是等效的。在 LabVIEW 中，二者还是截然分开的，Cluster 中只有数据，Class 中除了数据，还可以有方法。

C++ 类中的成员变量可以是私有，也可以是共有;为了安全起见，LabVIEW 中所有的数据都是私有的，必须通过公有的 VI 才能访问这些数据。

C++ 的类拥有构造函数和析构函数;LabVIEW 的类没有这两个方法。

继承是为了鼓励代码重用。不同的类可能拥有共同属性和方法，这些共性可以被抽取出来成为父类，被所有子类继承。比如，我们的程序要模拟“狗”和“鸡”两种动物的生活。它们之间其实有很多相似之处的，作为一个优秀的程序设计方案，应该把这些共同点提取出来，构成一个父类“动物”。“动物”类具有“狗”类和“鸡”类的公共属性与方法，比如年龄、进食等。构造“狗”或“鸡”类时，首先把这些公共属性、方法继承下来，再添加一些自己独特的属性方法，比如狗“看家”、鸡“下蛋”等。

C++ 的类支持多继承;LabVIEW 的类只支持单继承，这与 Java/C# 相似。

LabVIEW 中所有的类都有一个共同的祖先类，而 C++ 中没有。这点也与 Java/C# 相似。

多态最早也是个遗传学概念，源自同一祖先的不同生物会表现出多种不同形态。在面向对象中，多态是指同一个方法，在不同子类中有不同的表现方式。多态可以简化我们的编程，比如：几个子类都有同样一个继承自父类“动物”的方法“移动”，而不同的子类，狗和鸡移动的现实代码是不相同的：一个使用四两条腿，一个使用两条腿。在应用程序中只要调用父类“动物”的“移动”这个方法，一旦程序运行到这里，就会自动判断要处理的实例是属于狗还是属于鸡，然后去调用狗或鸡类中对“移动”方法的实现。

LabVIEW 的面向对象也实现了对多态的支持。

在 Windows Explore 中，鼠标右键点击某一文件夹的空白处，弹出的菜单中有“新建”一项。通过修改 Windows 的注册表，可以给这个新建列表添加一项，从而直接在文件夹下创建一个新的 VI。如图1所示。

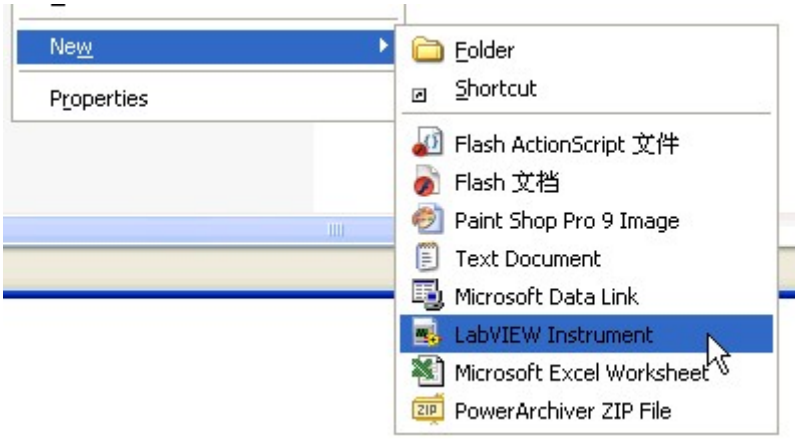


图1：在“新建”菜单中添加一项-创建 VI

下载 Create New VI.zip 文件，然后解压缩。运行里面的 Create Empty VI.reg，把它里面的内容导入到注册表。然后就会发现多出图1所示的新建 VI 的项目了。

让 Windows 多一个新建项目，只要在注册表里添加上相关内容就行了。Create Empty VI.reg 文件中的 Data 数据其实就是新建出来的文件的内容。在这个例子中，他是一个空白的 LabVIEW 8.0 的 VI。我们可以改变 reg 文件中的数据，使得产生出来的 VI 有所不同。

比如，你希望自己的新 VI 总是使用蓝色背景的，有一个特殊的图标等等，只要改变 reg 文件中的数据就可以了。ZIP 包中还有一个 Create New Reg Data.vi，这个 VI 可以读入一个文件，把它转换成注册表中实用的 Data 数据。使用者可以自己先造一个自己喜欢的 VI 作为模板，然后利用 Create New Reg Data.vi 为它创建出一个注册表数据，导入注册表。这样每次在文件夹下用鼠标右键创建出的 VI 就是模板 VI 的样子了。

面向对象与数据流驱动的结合

LabVIEW 是数据流驱动的编程语言：数据在数据线上流动，每个节点通过输入端的连线接收到数据，对其进行处理，再把结果传给输出端连线。

为了符合数据流的概念，多数情况下 LabVIEW 函数（或子 VI）使用的传参方式是值传递：就仿佛是整个数据在连线上流动，遇到一个节点，便一股脑都传到节点中去。必要时，譬如数据线分叉的时候，数据便生成一个副本，这样就有了两份同样的数据，沿着不同的分支继续传递。

也有例外的情况，比如使用 refnum 这种数据类型的时候。真正有意义的数据是存放在内存的某个地方不动的，而在节点间流动的只是一个指向这片数据的引用。这种传引用的方式破坏了数据流的概念，只在不得已的情况下才可使用（比如在多线程中对同一块数据进行操作），否则还是应当尽可能遵循 LabVIEW 一贯的值传递方式。

为了与用户熟知的数据流方式兼容（风格不一致，必然造成程序的极大混乱），LabVIEW 中的对象也是按照值传递的方式在节点间流动的。这点和其它语言有所差别，文本编程语言中传递对象时，基本都使用传引用的方式。

实际上，这两种传参方式各有特色，LabVIEW 别具一格的选择了值传递方式，是因为，在 LabVIEW 中，值传递的优势更大一些。

传引用的优点在于效率高，一个对象的数据量往往都比较大，值传递免不了要生成一些副本给被传递的数据，这类开销是相当大的。而一个引用类型的数据一般只需要占用4或8个字节，传递它们的开销远小于直接传递数据。

在多线程程序中，传引用意味着不同的线程可以访问同一块数据。在不同的线程中同时对同一数据进行读写是很危险的，它可能会产生不可预期的结果。所以，在多线程程序中常常使用临界区、信号量等方法来防止竞争状态的出现。这对于 C++ 或其他文本语言的程序员不是一个太大的问题，编写多线程程序的人员多少已经对可能出现的竞争状态有所了解。并且他们清楚地知道自己在编写多线程程序，会格外留意并采取相应措施防止错误出现。

而 LabVIEW 的使用者中，相当一部分人是非计算机专业的。为了帮助这些非计算机专业编程者利用多线程的优势，LabVIEW 采用了自动多线程的机制。LabVIEW 中，编程者并不需要告诉程序去开辟新线程；任何两段逻辑上没有先后顺序的代码，都有可能被自动的放到两个线程里去同时执行。

在这种情况下，传引用是非常危险的，编程者可能根本没意识不到程序有多个线程，因而无意中写下存在竞争状态的代码。

只有值传递才可以解决这个问题。值传递意味着数据每到一个分叉处，就变成相等但互相独立的两份数据。每个可能同时运行的数据线上的数据都是相互独立的，程序永远不会试图去同时访问同一个数据。这样就避免了无意识下造成的竞争状态。如果程序中的确需要在不同线程里处理同一对象，编程者可以在明确程序风险的前提下使用 LabVIEW 中的传引用机制，并做好多线程安全防护。

那么，类的实例作为值传递的数据，到底都包含哪些内容呢？这个数据你可以把它看成是一个簇。这个簇中可能又包含一下多个簇：

1. 一个簇包含所有类的属性；
2. 一个簇包含它父类的所有属性；
3. 一个簇包含它父类的父类的所有属性；
4. 一个簇包含它父类的父类的父类的……

……

除了这些属性数据，类的实例还包含有自身类别的信息，比如自己属于哪个类，什么版本等。这样，一个实例即便是按照它的某个祖先类来传递，也同样可以在需要的时候调用属于自己本身类的方法。

LabVIEW 中的类

一、创建一个空的类

在 LabVIEW 工程窗口里，鼠标右键菜单的新建栏中有一项，是创建类。类的结构和 LabVIEW 工程库是比较相近的：类的名字也作为名字空间；也可以为类中的 VI 设置访问权限等。类在硬盘上被保存在一个 .lvclass 文件中。这个文件其实是一个 XML 格式的文本文件，它的格式与 .lvlib 类似。

类是一个抽象的定义，符合这个类的实体，叫做类的实例。这有点类似数据类型和数据之间的关系。

我们先来创建一个名叫 Animal 的类吧，用它来描述一些动物的属性和行为。现实中，通过特定的属性和方法（行为）来定义某一类事物；与之对应的 LabVIEW 中的概念是类的数据和 VI。

动物类是一个类，符合这个概念的任何一个实体比如某一只小猫，一条小狗就是这个类的实例。程序中处理的都是这些实例。

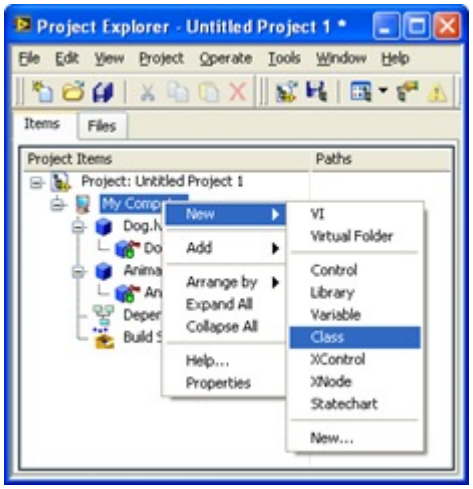


图1：创建类的菜单

二、类的属性

在工程窗口中可以看到，每个类包含数个 VI 和一个与类同名的 .ctl 项。尽管它的面板与设置方法与用户自定义控件类似，但它实际上并不是一个独立的用户自定义控件。此外，类的.ctl 项必须是一个 Cluster。Cluster 中的元素就是这个类所使用的数据，相当于 C 语言的类中的变量。通过改变 Cluster 中的元素的默认值，你可以在这里设置类的属性的初始值。

与 C 语言不同之处是，LabVIEW 类中数据只能是私有的。

公有数据是最容易被滥用的。为了自己使用方便，非常专业的编程人员常常倾向于把类中的数据都设置为公有，可以方便随时随地访问它。但这样一来就完全破坏了类的封装性，不加控制地访问类中的数据增加了模块间的耦合度，使得可读性和可维护性都大大降低。

通过类的方法访问类中数据就安全得多。比如我们可以在方法中添加对写入数据的合法性检查，在数据越界时报错等。

这样也有利于调试。比如我们需要跟踪某个类的数据的变化，如果数据是公有的，程序运行时就无办法预知它是在那里被改变的。若数据是私有的，我们就可以确定它只在类中设置它的 VI 中被改变。只要在这个 VI 上加个断点，就可以在调试时，令程序在数据被改变之前暂停运行。

LabVIEW 相当一部分用户是非计算机专业的人员。对于他们来说，概念越简单越好。类的数据强制为私有类型，可以避免他们接触更多的程序设计概念，而直接引导他们使用最佳的程序设计方法。

这样的设计方法唯一不足之处是：即便是的确需要被类之外的 VI 直接访问的数据，也必须给他们创建一个公用的方法，通过这个方法间接访问这个数据。幸好，类的右键菜单中有一项专门为数据创建访问 VI 的选项（VI for Data Member Access...）。通过它，可以便捷地创建出数据访问 VI 以供使用。

现在，回到我的 Animal 类：它有两个属性，分别是动物的年龄和颜色。于是我在 Cluster 中放了两个分别表示年龄和颜色的控件。

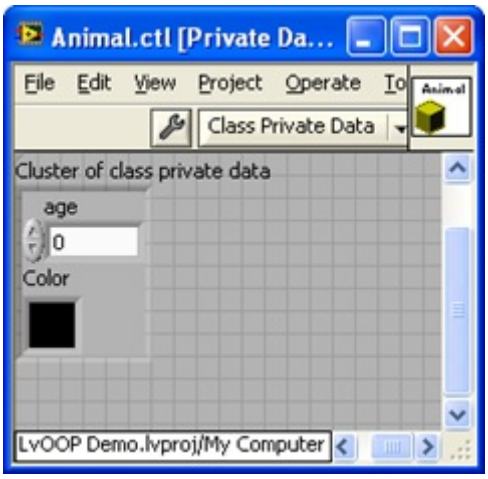


图2：添加类的数据（类的属性）

三、类的方法

鼠标右键点击在类上，就可以为类创建 VI，也就是类的方法。

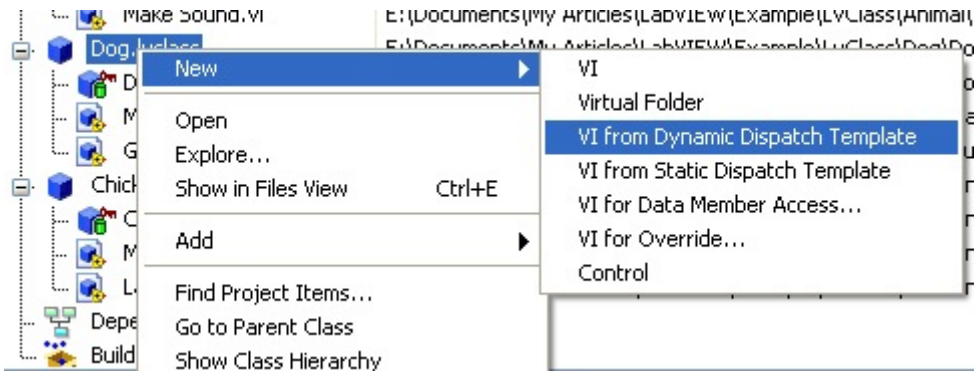


图3：创建新方法

在上图新建这一栏下可以看到很多条目：

VI，就是指创建一个普通的 VI。

Virtual Folder，是文件夹。如果类中的方法很多，可以把它们归类到不同的文件夹中，便于管理。

VI from Dynamic Dispatch Template，所创建出来的 VI 类似于 C 语言中的虚函数。应用程序再调用这个 VI 的时候，可能实际执行的是某个子类中的同名方法。

VI from Static Dispatch Template，所创建的 VI 比普通 VI 多了类方法最常用的代码框架。程序员可以省去一些自己画错误处理选择框的时间。它与 VI from Dynamic Dispatch Template 唯一的区别在于：类输入输出接线端子（这个例子中是“Animal in/out”）不是动态调度的。（参见图4：动态调度的接线端子）

VI for Data Member Access...，因为类的数据全部是私有的，所以需要借助公有 VI 来访问他们。这个选项可以帮你快速建立读写类中数据的 VI。

VI for Override...，这个选项是专门给子类用的。用来创建覆盖父类方法的 VI。

Control，创建用户自定义控件，这一条与类的概念不相关，仅为了方便用户。

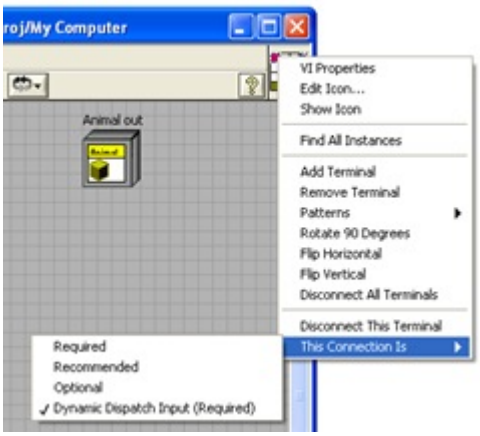


图4：动态调度的接线端子

在类的类的属性面板中可以设置类中每个 VI 是公有的还是私有的。这与工程库中 VI 的设置是类似的。

可能你已经发现了，与其它语言不同，LabVIEW 中的类没有构造和析构函数。构造函数在一个类的实例（数据为这个类的一个变量）生成时被自动调用，析构函数在它被销毁时自动调用。

在 C 语言中，你可以明确地知道一个变量的生存周期。全局变量在程序启动时被创建，程序结束时被销毁；函数的局部变量在函数被调用时创建，退出函数时销毁，等等。这些都是程序在运行时的行为。但是在 LabVIEW 中，变量的生存周期不一定是在运行过程中。LabVIEW 的变量通常对应有一个前面板上的控件，控件包含的数据在编辑状态下就已存在了，程序运行结束也不会被销毁。这就使得构造函数和析构函数失去了原有的意义。比如，构造函数和析构函数一个最常见的用法是在构造函数内预留某一资源，以供类中的方法使用，在析构函数内释放这个资源。LabVIEW 若有类似功能，则 VI 被打开时，资源就被霸占住了，这在逻辑上是错误的。

没有构造函数和析构函数，我们可以把预留释放资源一类的工作放在普通的类的方法中实现。只是在使用这个类的实例的时候，需要程序员自己调用这些方法。

四、类的继承

为了让演示程序更有意义，再分别为狗和鸡创建两个类。这两个类应为动物类的子类。进入类的属性对话框，在 Inheritance 一栏中选择 animal.lvclass 作为它的父类，这两个类便成了 animal 的子类。可以注意到，LabVIEW 中所有的类都有一个共同的父类“LabVIEW Objet”。

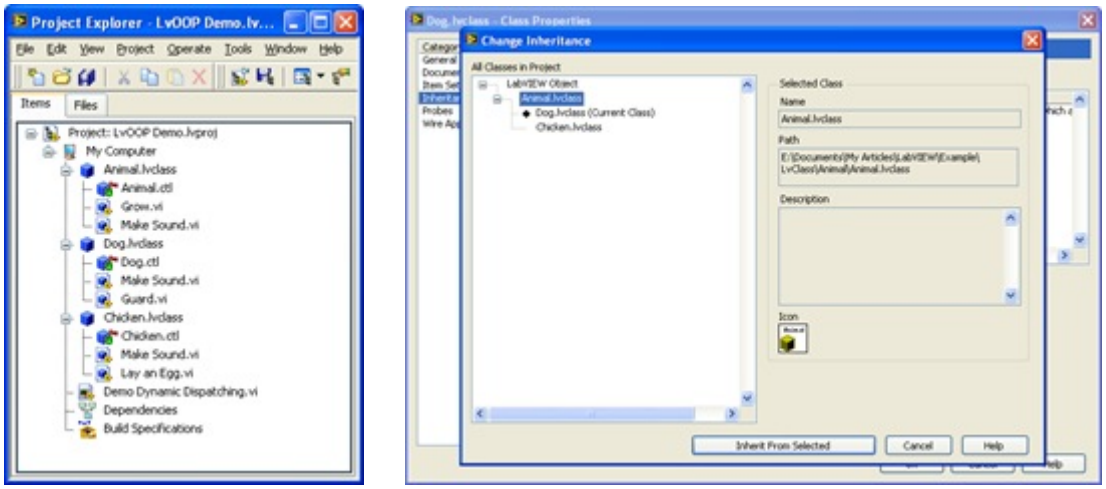


图5：设置类的继承关系

在这个设置面板上可以看到，LabVIEW 所有的类都有一个共同的祖先类 LabVIEW Object。LabVIEW Object 是个空类，既没有方法也没有属性。那么它存在的意义是什么呢？

这要先介绍一下泛型编程的概念。理论上，设计程序模块时，越抽象越好。这样同一段代码可以被应用到更多的具体问题中去。本着这个原则，程序中算法和数据类型应该是独立的。比如，一段排序算法的代码被完成后，应当可以被应用在各种数据类型上，既可以用来给一组整数排序，也可以给一组字符串排序。这就是泛型编程。

LabVIEW 暂时支持泛型编程，一个算法 VI 写好，它作为传递参数的控件的数据类型也就定死了。不能够直接使用在其它数据类型上。但是类的实例作为一个数据在 LabVIEW 不同节点间传递时，它的数据类型可以在它本身的数据类型，以及它的任意一个祖先类之间进行切换。比如在处理一只狗小狗的时候，可以把它当作是狗，也可把它当作是动物，还可以把它当作是 LabVIEW Object。

我们再实现一个算法的时候，使用 LabVIEW Object 作为它的参数的数据类型。这样这个算法就可以被应用到人和一种“类”的数据上。Java，就是采用了类似的机制来实现泛型编程的。但是 LabVIEW 并没有因此获得泛型编程的能力。与 Java 不同，LabVIEW 不能直接把一个普通数据类型（比如整数，字符串等）转换成某种“类”。所以，LabVIEW 编写的算法还是不能支持任何数据类型。

五、其它辅助性设置

设置好继承关系，再为子类创建几个属性和方法，我们的演示类就搭建完成了。为了让应用程序美观易读，我们可以修改这几个类的数据线外观。否则，所有的类的数据线千篇一律，很容易就混淆了。数据线的外观也是在类的属性对话框中配置的。

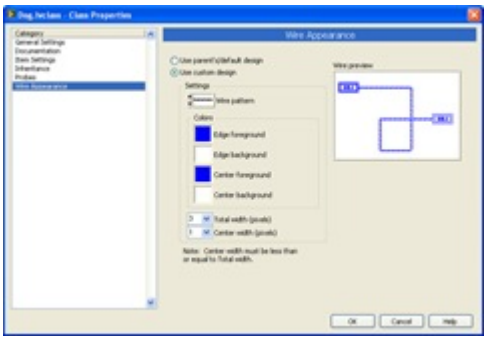


图6：配置类数据线的外观

六、演示程序

我们再简要介绍一下类的多态：在动物类中先用虚函数方法（VI from Dynamic Dispatch Template）创建一个“叫唤”方法：Make Sound.vi。因为狗和鸡的叫声不一样。因此，在两个子类中，我们用 VI for Override... 重新实现这个方法，使其覆盖父类中的“叫唤”。应用程序中有几个不同动物的实例，程序的任务就是让它们每个实例叫一声。借助类的多态特性，应用程序不需要判断实例数据所属的子类，再根据不同子类编写不同代码的。它可以把所有实例用他们共同的父类的类型来传递，代码中也只是用父类的方法。而程序执行到父类的方法时，会自动执行已经覆盖了它的相应的子类的方法。从而让不同的动物发出不同的叫声。

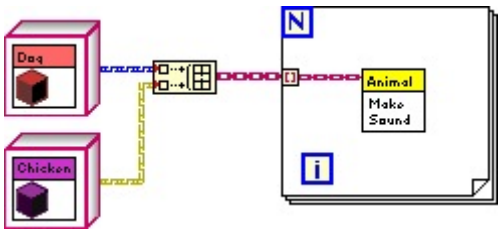


图7：动态调用的示例

一个 XControl 的实例

XControl 与 .ctl 用户定义控件相比，其最大的提高就在于它不但可以定义控件的外观，还可以定义控件的行为。

在 XControl 出现之前，同样可以在程序中编写代码，控制程序的行为。在《用 XControl 实现面向组件的编程》一文中提到了，这种方法在程序模块划分上有缺陷。如果用户想发布一个带有特定行为的控件也是不可能的，因为控制控件行为的代码，是同其它代码混杂在一起的。

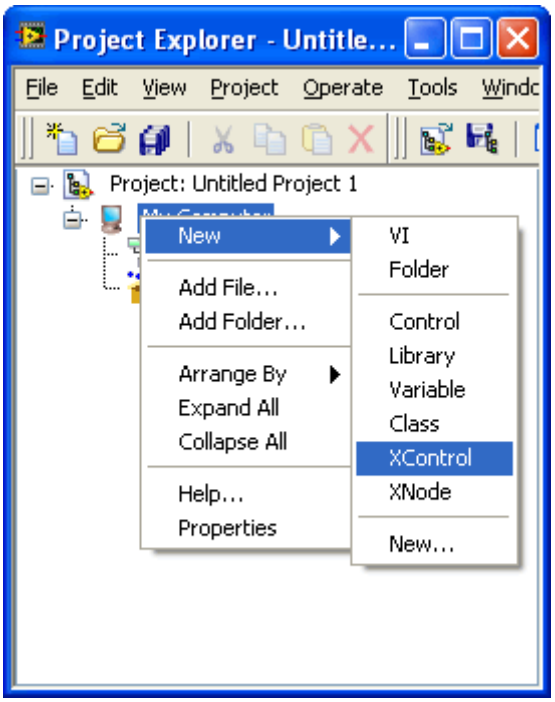
利用 XControl 可以解决上面提到的问题，这里以一个例子说明一下如何利用 XControl 实现一个有特定行为的控件。

Windows 风格的工具条上的按钮有一个特点，就是当鼠标移动到按钮上方，按钮就会变亮或浮起。LabVIEW 中默认的按钮没有这样的特性，但是实现这一点是很容易的。

以鼠标移上，按钮变亮为例：在程序中，当按钮的 Mouse Enter 事件发生时，把按钮的颜色设置为浅颜色；当按钮的 Mouse Leave 事件发生时，把按钮的颜色设置为深色即可。现在把界面上的按钮和控制颜色的代码都封装在一个 XControl 中。这样，其他人在使用这个 XControl 时，就无需修改他的代码，而直接获得这种颜色变化的特性了。

一、简单行为的 XControl

首先创建一个空的 XControl。



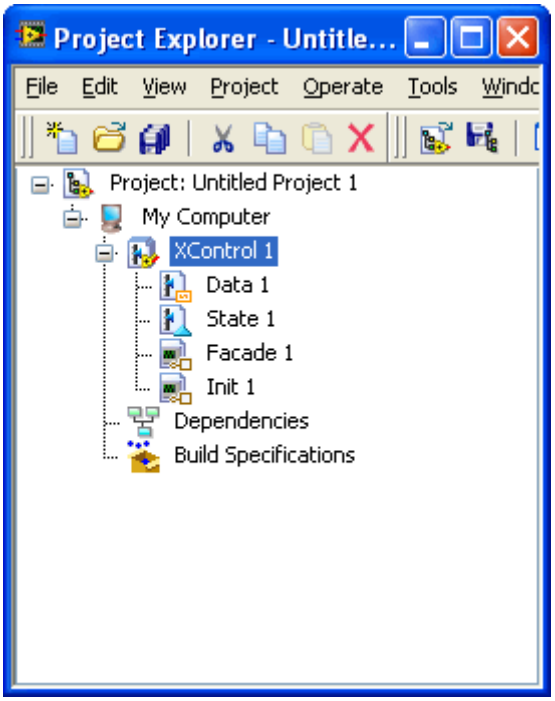


图1、2：创建一个新的 XControl

新的 XControl 中有四个 VI。

Data.ctl 定义 XControl 的数据类型。比如我们要做一个按钮，数据类型应该是布尔型。如果要作一个工具条，数据类型就应该是布尔型数组了。

State.ctl 定义 XControl 内部要用到的一些数据，类似于类的私有变量。我们这个简单的例子用不到任何变量，所以可以不去动它。

Init.vi 类似于类的构造函数。在我们这个简单的例子中也不需要去改变它。

Facade.vi 是最主要的 VI，XControl 的外观和行为都是在这个 VI 中定义的。Facade.vi 的界面就是 XControl 控件的外观。控制控件行为的代码也是放在这个 VI 的程序框图上。

我们要做的是个按钮，所以就在 Facade.vi 的前面板上放一个按钮。如果希望用户在使用这个 XControl 时可以调整它的大小，在我们这个简单例子中，只要设置 Facade.vi 窗口尺寸属性中的“在窗口尺寸变化时，按比例调整控件大小”这个选项就可以了。对于复杂的 XControl 控件，要另写代码，在窗口尺寸变化后重新计算每个控件的大小和位置。

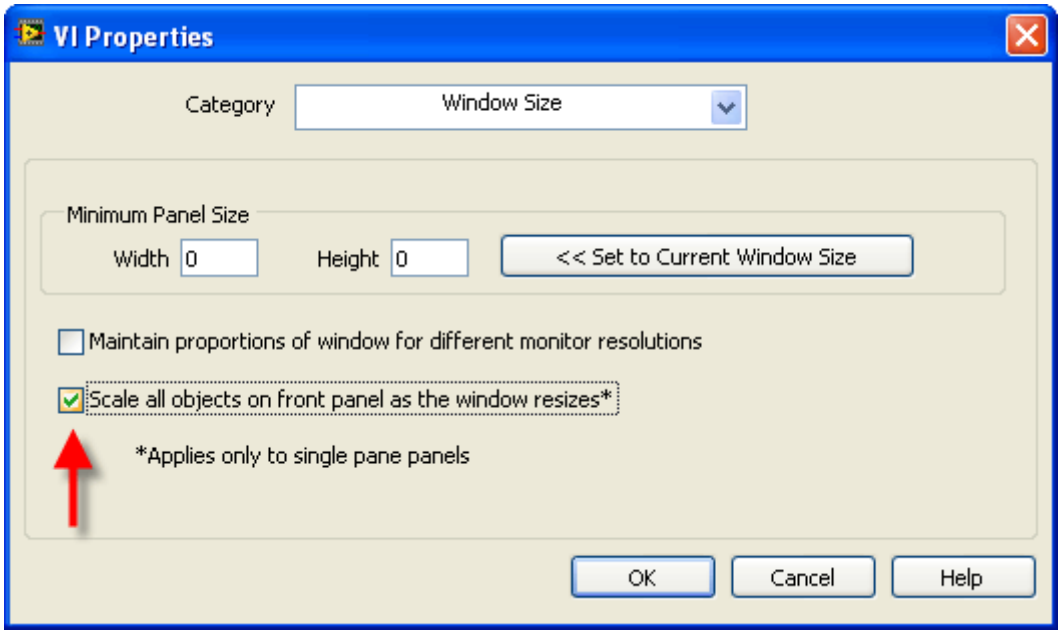


图3：窗口尺寸属性设置

控制按钮颜色的代码也需要放在 Facade.vi 中：把前文提到的按钮的 Mouse Enter 和 Mouse Leave 放在这里即可。具体实现方法，可以参考文章结尾给出的范例程序。

二、有持续运动的 XControl

Facade.vi 不能够持续运行，只有在有事件发生时，LabVIEW 才会调用这个 VI。处理完这个事件，Facade.vi 就会停止运行。不要试图让 Facade.vi 持续运行，否则会导致整个 LabVIEW 被挂起。

有时候，需要控件能够循环地或者持续一段时间地作一个动作。比如说，需要做一个不停闪烁的小灯。控制灯光闪烁的代码就不能够放在 Facade.vi 中。实现这种功能的一个方法是：

把定时控制小灯颜色的代码放在一个可重入 VI 中，通过小灯控件的引用参考来定时更改它的颜色属性。在 XControl 的 Init.vi 中把这个定时 VI 动态加载并以异步方式运行；在 XControl 的 Uninit.vi 中再把这个定时 VI 卸载即可。Uninit.vi 不是一个必须的 XControl 功能定义 VI（Ability VI），新建的 XControl 没有这个 VI。可以在工程浏览窗口，鼠标右击这个 XControl 来为它添加新的功能定义 VI。

范例在这里，它只能在 LabVIEW 8.5 下打开。

XControl 是可以在 VI 的面板上放多个实例的，每个实例小灯的闪烁频率可能不同。我在这个例子里，每个 XControl 实例都有自己的一个专用定时 VI，因为这些 VI 是可重入的。定时的方法我采用的是加延时。

我做了一下测试，发现现在的 XControl 有个问题，就是在程序面板上放多个 XControl 实例之后，定时就变得非常不准确了，小灯闪烁速度明显减慢。这也许是 XControl 的 bug，也许是 LabVIEW 延时函数的问题。解决这个问题的方法就是使用一个定时 VI 控制所有的实例，当然这样的实现方法会比较麻烦一些。

XControl 是 LabVIEW 8 中出现的新功能

面向组件的编程(Component Oriented Programming ，COP)技术建立在对象技术之上，它是对象技术的进一步发展。“类”这个概念仍然是组件技术中一个基础的概念，但是组件技术更核心的概念是“接口”。组件技术的主要目标是复用—粗粒度的复用，组件的核心是接口。

LabVIEW 为我们提供了大量漂亮的控件，可以让我们非常方便地就搭建出一个程序界面来。然而，对于追求完美的用户而言，LabVIEW 提供的为数有限的控件是远远不够的。比如图1，是 LabVIEW 8.2 的一个新功能—导入导入共享库向导的界面。在它右上方有四个按钮，这四个按钮有着特殊的外观图标，在 LabVIEW 中并没有直接提供这样的按钮。要拥有这样的按钮，并保存下来以供再次使用，就只能自己来制作这样一个自定义控件。关于(用户自定义控件可以参考文章《用户自定义控件中 Control, Type Def. 和 Strict Type Def. 的区别》)

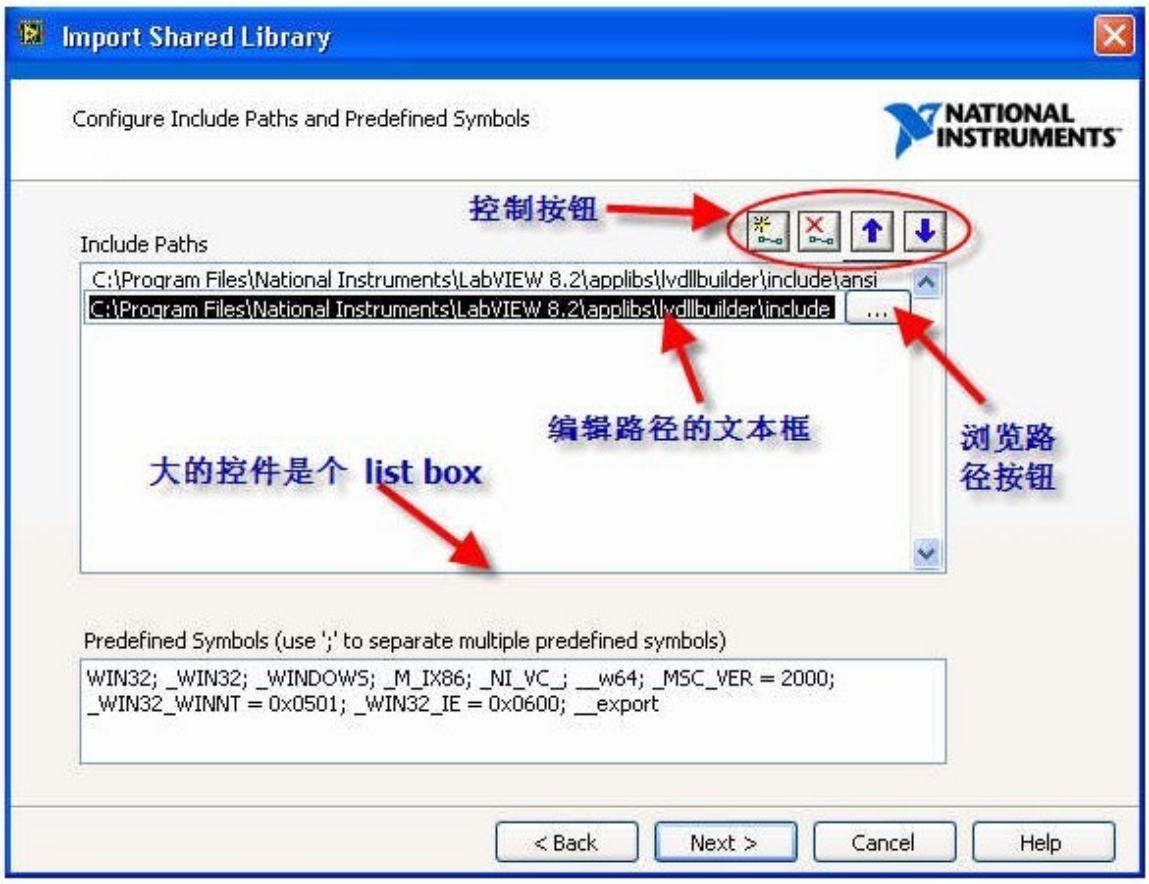


图1：LabVIEW 8.2 中 Import Shared Library 的界面

自定义控件虽然可以定义控件的外观，但无法定义控件的行为。功能复杂一点的控件，.ctl 文件就爱莫能助了。还是以图1 为例，它的 Include Paths 控件是“一个”功能比较复杂的控件，它比 LabVIEW 自带的列表框多了编辑功能。用户添加或编辑一个路径时，这个控件要为用户在所编辑的路径项目上，显示出供编辑使用的文本框和浏览路径按钮。

实际上这个编辑功能是由三个 LabVIEW 提供的标准控件合作完成的：一个 Listbox、一个 String 和一个 Button 控件。它们的行为(位置、是否可见，值，等等)是在程序运行时决定的：当用户选择编辑控件中某一路径时，程序就把 String 和 Button 挪到 Listbox 上需要编辑的那一项，并遮挡住 Listbox 原本的内容。这样，用户只能在 String 控件内输入内容，或者点击浏览按钮选择一个路径。编辑完成，程序把 String 控件的值写到 Listbox 上相应的项目中。

我们虽然看不见图1 例子中的程序框图，但是可以想象，上述的一系列操作，如判断 String 和 Button 应当显示的位置;然后挪动它们、把 String 值传给 Listbox;处理用户对他们操作的消息等等，都会为这个程序添加不少复杂的代码。这些代码应该是与程序的其它部分没有任何直接关系的。但是如果把它们也写在这个界面 VI 的程序框图上，一方面影响了程序的可读性;另一方面，编程人员有可能在更改程序其它问题时，一不小心改变了这部分代码，降低了代码的安全性。

从逻辑关系上来看，图1 中上半部分的 Listbox、String、浏览按钮以及右上方四个操作按钮，合作共同完成一个功能，与它们之外的界面控件没有什么直接关联，所以他们七个应当被作为一个整体，或者说是一个组件。这个组件需要与程序其它模块之间的接口就只是一个字符串数组—用于输入或输出一组路径。其它的数据和操作，都应当是组件私有的、外部不可见的。

在 LabVIEW 8 之前，想分离和封装出这样一个组件是非常困难的。因为既然这七个控件都在这个 VI 的面板上，对它们的操作和相应的代码必须放在这个 VI 的程序框图上，无法与其他代码隔离开。当然也不是说绝对没有办法，比如你可以使用 sub-panel、动态注册事件等方法，强行地把它们的代码分隔开。但是这些方法既不简单直观，使用它们又可能会让程序变得更为复杂、难以阅读和维护。

XControl 的出现终于为这个问题提供了一个比较完美的解决方案。图1 中我们提到的七个应当划分在同一组件的控件可以被制作成一个 XControl。这个 XControl 的外观就是图1 中上半部分七个控件组合在一起的样子。XControl 与用户自定义控件相比，它不仅定义了控件的外观，更重要的是，开发人员可以通过编写 LabVIEW 代码定义 XControl 的行为，这些代码是对外隐藏的。开发人员还可以定义 XControl 的属性和方法，通过 Property Node 和 Invoke Node 在程序中使用这些属性和方法。

同样完成选取一组路径这个功能，可以有各种不同的界面。比如各种 C++ 编译器都会提供类似的功能，但它们外观各不相同。你可以利用 XControl，编写多个外观、行为大相径庭的组件。但是，只要他们的接口相同—都是一个字符串数组，用户就可以在这些组件内任意互换，选用自己喜欢的组件，而不需改动程序的任何其它部分!

现在，XControl 仍然不太令人满意的地方是它还不支持用户自定义的事件。

XControl 具有封装的特性。因此我在《利用 LabVIEW 工程库实现面向对象编程》一文中提到，同样可以使用 XControl 来达到面向对象的编程方法。但是 XControl 不具备继承和多态的特性。与之相比较，Library 和 LVClass 只能够把程序中的某些功能封装成模块，而涉及到包含界面的模块，就无能为力了。XControl 则非常适合制作有界面的程序组件。

开发 XControl 1 - 设计

XControl 是 LabVIEW 8 开始出现的一个制作 LabVIEW 控件的工具。与之前的用户自定义控件相比，用户自定义控件只能定义控件的界面，而 XControl 还允许通过编写代码来定义控件的行为。因此 XControl 功能更加强大。

XControl 的主要优点是可以把界面元素与相关的代码封装在一起，从而方便发布和重用这些界面组件。

XControl 也有比用户自定义控件不足的地方，它开发起来更加困难;设计不合理的 XControl 会导致程序更加严重的问题。

需要开发一个新的控件之前，首先要考虑一下以何种方式实现这个控件。

如果这个控件极为特殊，只会用在某个特定的程序中，那么也许没有必要将其作为单独的控件;

如果这个控件需要被多次使用，那么就应该考虑把它做成可重用的独立控件。这个控件也许不包含任何特殊行为，比如一个用于表示坐标位置的控件由两个数值控件组成，程序只是使用它的值就可以了;或者一个新型按钮，进外观与旧按钮不同，其它行为都与传统的按钮一模一样。这样的控件适合使用用户自定义控件来制作。

如果新的控件需要重用，行为与已有其它控件又有较大差别，那么就要考虑 XControl 了。比如：制作一个新按钮，但它比传统按钮多一个状态;或者它的界面带有动画效果;制作数值类控件但是用中国本土度量单位;基于图片控件，专用于绘制某种特殊曲线等。

我们前面提到的黑白棋的控件，既有特殊界面，又有特殊行为，又可以应用于不同软件中，非常适合做成 XControl。我们先来具体设计一下这个 XControl 所需的界面和行为。

它的界面部分前面已经设计好了，直接拿来用就可以了。不过在前文提到的几个设计方案中，我个人觉得 Pict Ring 数组控件的那个解决方案，最能简化编程代码，所以我们采用这个界面方案。

XControl 在应在程序框图上的端点的输入输出数据应该是应用程序最经常需要与 XControl 交互的数据。本例中，应用程序最常使用的数据就是棋盘的布局信息。因此，这个 XControl 的数据应当是一个8×8的整型数组，表示棋盘上棋子的布局。

黑白棋控件的属性应当包括：当前该下什么颜色的子、可落子的位置、盘面上每种颜色的子数、上次落子的位置。

它的方法有：落下一子(这个方法需要包含以下具体的操作：在新位置放置一个棋子;翻转被吃掉的棋子;更新数据和所有属性的值)。

它还要在当用户在交互界面上摆下一子之后，发个事件通知应用程序。

开发 XControl 2 - 创建

在项目浏览器上，点击鼠标右键，选择“新建->XControl”，就可以创建一个新的 XControl。

XControl 在结构上是一种特殊的库，他包含一些特定的更能 VI，和一些可选的属性、方法 VI 及其它相关文件。在新建的 XControl 上已经包含了4个必须的功能 VI(控件)：数据、状态、外观、初始化。XControl 还有两个可选的功能 VI：反初始化和转换状态以保存。

简要介绍这几个功能 VI 的功能是：

数据：用来定义 XControl 的数据类型。

状态：定义所有 XControl 内部使用到的数据。

外观：这是 XControl 中最主要的功能 VI，用以实现 XControl 的界面和界面上的行为。

初始化：设置 XControl 的初始状态。

反初始化：负责清理工作。

转换状态以保存：用于把 XControl 内部的某些数据保存在使用它的 VI 中。

下面对 XControl 做一些基础的设置，比如修改它的图标、版本号等，然后保存。

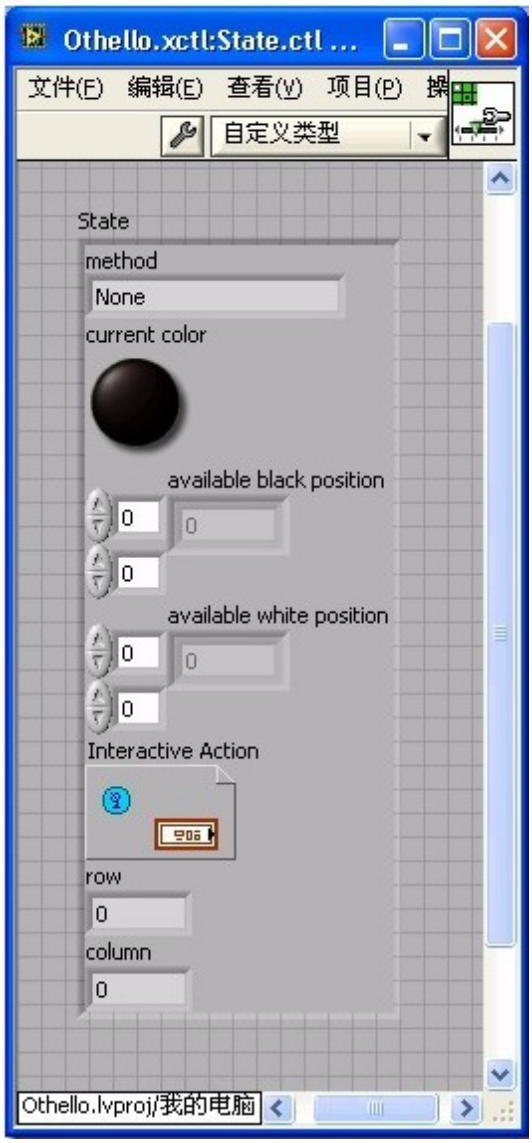
XControl 功能 VI 的文件名并不一定与其功能名相同。比如，为了方便更多人使用，我使用了英文名称来保存我的 XControl：


项目浏览器 - Othello.lvproj
文件(F) 编辑(E) 查看(V) 项目(P) 操作(O) 工具(T) 窗口(W)
项
文件
项目: Othello.lvproj
我的电脑
Othello.xctl
Data.ctl
State.ctl
Facade.vi
Init.vi
依赖关系
程序生成规范

开发 XControl 3 - 实现功能控件

XControl 有两个功能控件，本别定义 XControl 的数据类型，和 XControl 使用到的内部数据的数据类型。

首先考虑数据功能控件，它用于定义 XControl 的接线端的数据类型。我们使用一个二维的 U16数组表示棋盘布局，所以在数据功能控件中要使用一个二维数组。



其次就要来考虑状态功能控件，这个控件类型的数据在 XControl 的功能 VI 中又被称为显示状态。但它实际上的用途并不局限于帮助显示，实际上，XControl 运行所需的全部变量，都应当被包含在这个功能控件中。

左面这幅图就是我所列出的运行一个黑白棋 XControl 所需的一些变量。

在我们编写的黑白棋程控件中，将会用到一下内部数据：

method，当用户运行一个 XControl 的方法时，设置这一变量。这一变量对应每个方法有不同的值。这样，在 XControl 的外观功能 VI 中，就可以知道用户调用的是什么方法了。

current color，用于表明当前应该落什么颜色的棋子。

available black position，黑色棋子可以防止的位置。

available white position，白色棋子可以防止的位置。

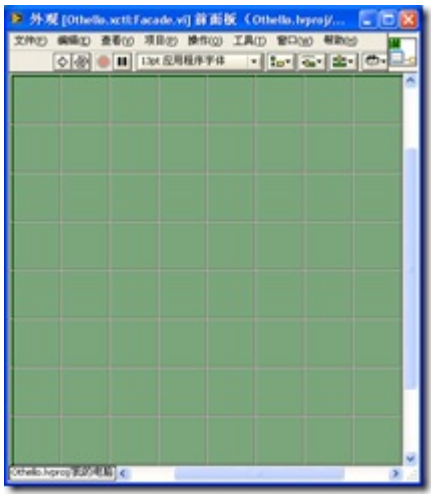
Interactive Action，是一个用户自定义事件。当用户在棋盘上落下一子时，XControl 就产生这个事件，通知使用了它的 VI。

row 和 column 用于记录上次落子的位置。

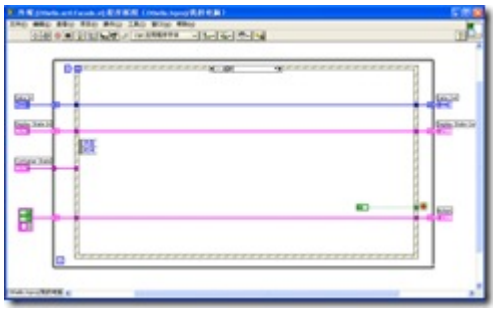
先不需了解这里边每一个数据具体的含义和用法。在后面使用到它们的时候还会详细介绍。实际在编写 XControl 的时候，也不需要一次把状态功能控件就设计好。可以一边实现 XControl 的功能，一边对其进行补充。

开发 XControl 4 - 外观功能 VI

外观功能 VI 用于定义 XControl 的界面，这里可以直接使用我们之前已经设计好的界面。把界面设计技巧 4 - 改进界面实现方法中设计好的棋盘棋子界面拷贝过来就可以了。这个外观功能 VI 窗口的大小，就是将来 XControl 控件的大小。所以这个 VI 的大小要刚好包裹住棋盘。



外观功能 VI 的程序框图定义了当有事件发生时，XControl 的反应。这是一个典型的事件处理结构。但是大家要注意到它的超时事件处理（下图），程序在这里设置了退出循环。实际上，这个程序框图并不是持续运行的，只有当 XControl 有事件发生时，LabVIEW 才调用这一程序框图，在处理完这个事件后，立即退出这一程序框图。所以千万不要试图在这里添加持续执行的代码，比如控制 XControl 上的动画等等。



从上图中还可以看到，外观功能 VI 有三个输入，和三个输出：

Data In / Data Out，是 XControl 的数据，它的数据类型由数据功能控件定义。外观功能 VI 的程序框图开始运行时，Data In 输入的是 XControl 当前的值。程序框图运行过程中可以对这一值进行修改。修改后的值由 Data Out 输出，返回给 LabVIEW。

Display State In / Display State Out，是 XControl 运行是用到的内部数据，在这里我们就把它成为状态。它的数据类型由状态功能控件定义。它也可以在程序运行中被改变，的输入输出方式与 Data 类似。

Container State，是一个簇，用于表明 XControl 实例在 VI 面板上的状态。它有三个元素：Indicator? 表明 XControl 实例是否是一个显示控件，它的值为假时，表明 XControl 实例是

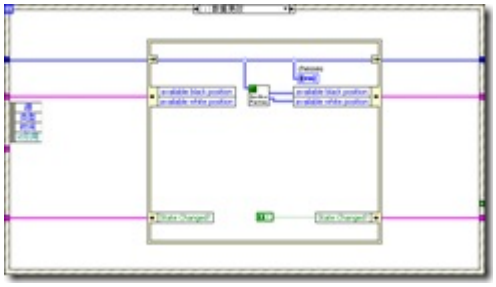
一个控制控件。Run Mode? 表示 XControl 实例所在的 VI 是否处于运行状态。Refnum 是指向 XControl 实例的引用。

Action 用于通知 LabVIEW 程序在这次执行中对 XControl 所做的修改。它有三个元素：Data Changed?，State Changed?，Action Name。如果我们在程序中改变了 Data，那么就一定要把 Data Changed?设置为真，通知 LabVIEW，这样改变的数据才会生效。同样，如果改变了 State，则一定要把 State Changed?设置为真。Action Name 是一个字符串，可以给他输入一个表明这次程序运行的简短文字。这段文字会在 LabVIEW 的菜单项“编辑->撤销”中出现。

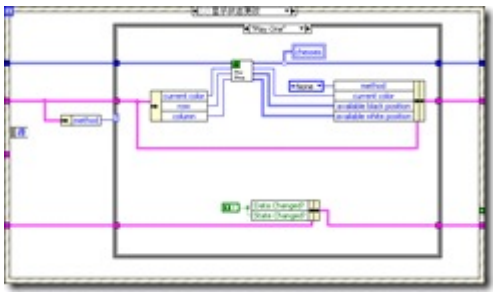
外观功能 VI 中的事件处理结构主要处理两类事件，一类是针对 XControl 的特殊事件，另一类是用户在界面上操作产生的事件。

针对 XControl 的特殊事件有4个：数据更改、显示状态更改、方向更改、执行状态更改。

当把一个数值输入给 XControl 的实例时，就会触发数据更新事件。对于数据更新事件的处理一般是根据新的数据更新界面上的控件，和 XControl 的状态（Display State）。下图是黑白棋控件对数据更新事件的处理：根据新的期盼布局，刷新棋盘在界面上的显示；重新计算黑白子可以落下的位置。由于我们在处理这一事件的过程中，更新了 XControl 的状态，所以一定要把 State Changed?设置为真。

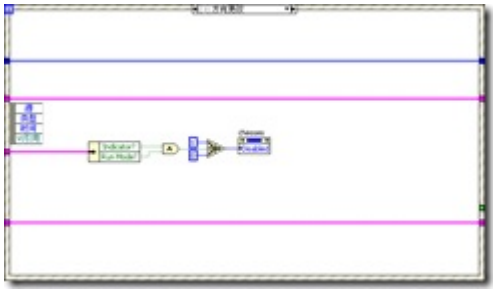


如果通过调用 XControl 的属性和方法，改变了 XControl 的状态的值，就会触发显示状态更改事件。对于数据更新事件的处理一般是根据新的状态值更新界面上的控件，XControl 的数据和状态。下图是当用户调用黑白棋控件的“走子”这个方法，放下一颗新棋子后，外观功能 VI 对其的处理：放下新棋子，更新棋盘，计算新布局下黑白棋子可以放置的位置。因为这个操作及更新了黑白棋 XControl 数据，也更新了它的状态，所以 Data Changed?和 State Changed?都要被设置为真。

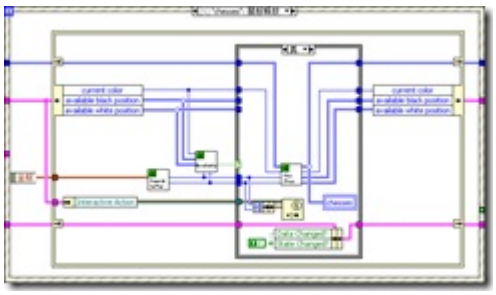


当 XControl 实例控件由控制控件变为显示控件，或反过来变化的时候，就会触发方向更改时间。当 XControl 实例控件所在的 VI 由运行态变为编辑状态，或反向变化时，就会触发执行状态更改事件。对这两个事件的处理是类似的：在某些状态下，需要禁止用户在界面

上的操作。在我们的黑白棋控件中，对这两个事件的处理是相同的。当控件为显示控件，并在运行状态时，禁止用户对界面点击。

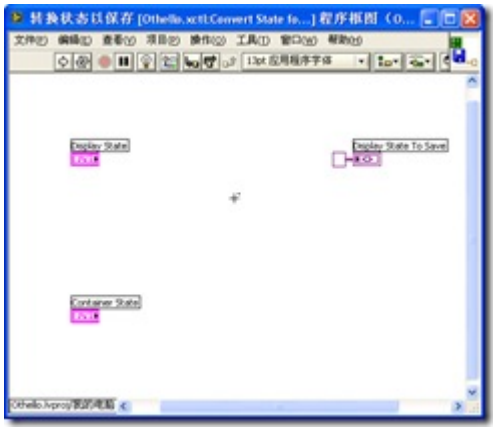


我们的黑白棋 XControl 只处理一个用户界面事件：当用户在棋盘上合法的位置点击鼠标时，走一步棋。当用户在即面上点击，首先判断这里可否落子，如果可以，则落下一子，更新 XControl 的数据和和状态，并产生一个事件。

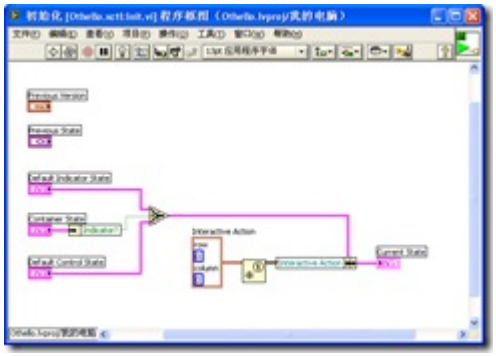


开发 XControl 5 - 其他功能 VI

转换状态以保存功能 VI 用于保存 XControl 的状态数据。默认情况下，XControl 外观功能 VI 中的状态(Display State)全部都会被保存在使用它的 VI 中。如果状态数据比较大，无疑会增加 VI 的大小。但是，这些状态也许并不需要保存。有些控件的状态，比如控件颜色，尺寸等信息，需要在 VI 关闭后仍然记得，在下次打开时，还可以保持上次的修改。但是有些状态中的数据，只是临时使用的，不需要保存。比如，我们的黑白棋控件状态中的任何数据，当前颜色，可落子的位置等等都是每次重新计算出来的，不需要保存下来共下次打开 VI 使用。所以，在转换状态以保存功能 VI 中，可以丢弃所有数据，保存一个空数据就可以了。



初始化 VI，有两个作用，一是把保存在使用 XControl 的 VI 中的状态读取出来，付给 XControl 的状态。而是打开或初始化 XControl 需要使用到的资源。对于我们的黑白棋控件，由于没有保存任何状态数据在 VI 中，所以不需要读任何数据出来。而我们的黑白棋控件使用到了一个用户事件，所以需要在初始化功能 VI 中创建这个事件。



反初始化功能 VI 负责关闭 XControl 中打开的资源，我们在初始化功能 VI 中创建了一个事件，所以在这里要销毁它。

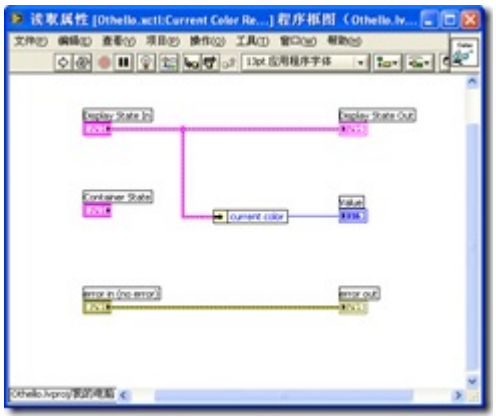

反初始化 [Othello.xctl:Uninit.vi] 程序...
文件(F) 编辑(E) 查看(V) 项目(P) 操作(O) 工具(T) 窗口(
13pt 应用程
Display State
Interactive Action
Container State
Othello.lvproj/我的电脑

开发 XControl 6 - 属性

在程序中，可以通过控件的属性节点来读取或设置一个控件的某些属性，比如它的位置，颜色等等。你可以为你的 XControl 实例控件添加自定义的属性，以供程序运行时使用。

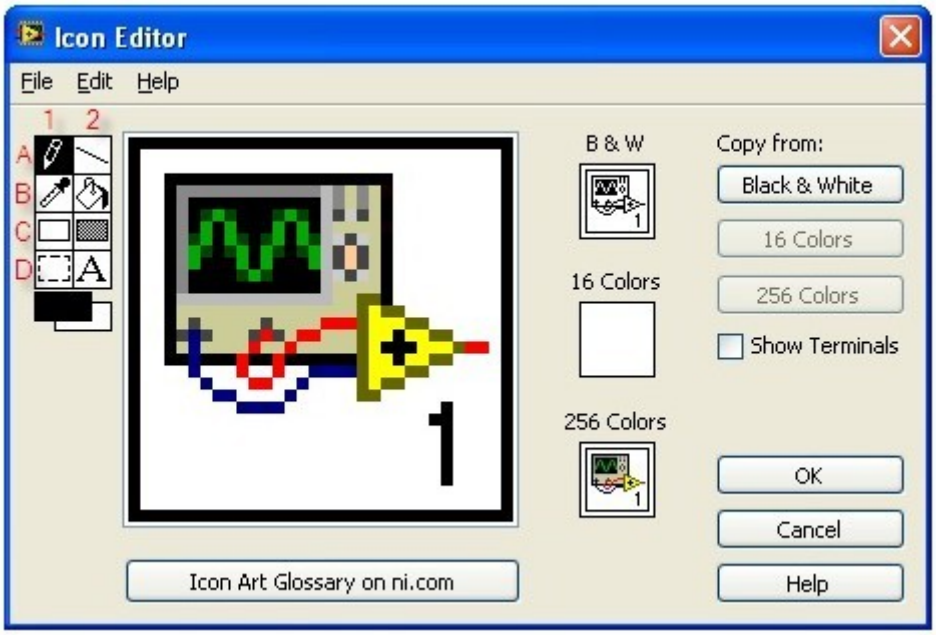
在项目浏览窗口的 XControl 上点击鼠标右键，选“新建->属性”，即可为 XControl 添加属性。每个属性对应两个 VI，分别用于读写属性。去掉其中一个 VI，属性就变成只读或只写的了。属性读写 VI 中的代码非常简单，基本上就是读出 XControl 状态中的某个数据，或者把某个数据写到 XControl 状态中去。

我们的黑白棋控件中有一个只读属性是得到当前该下什么颜色的棋子。他的实现如下：



图标是 LabVIEW 语言很有特色的一个东西，其它语言大概都不需要给它的函数设置一个图形标识。认真给每一个 VI 加上一个有意义的图标是非常必要的，它可以帮助程序人员快速理解一段程序的含义。一个风格良好的 VI，只要扫一眼它的程序框图，就可以了解他的功能了，而不必要像文本语言那样一行行把程序读下来。LabVIEW 程序比文本语言的程序可读性要高得多，当然这是指在编程风格良好的情况下进行比较。

有意义的图标是良好程序风格中非常重要的一项。编辑图标应该是一个很常见的工作，双击一个 VI 的图标就会出现图标编辑器，如下图所示。



上图这个这个图标编辑器界面左上方是绘制图标的工具栏，里面工具被我编了号。不知道大家有没有试过鼠标双击这几个工具，我是经常双击他们的。

编辑一个 VI 的图标，第一件事就是把他的默认图标删除，留一个空白框在上面就可以了。做这件工作不需要选中图标中的内容再删除，只要双击一次 C2这个工具就可以了。

如果不需要清楚图标中的内容，只想给他添个边框，那就双击 C1这个工具。

如果想选中整个图标，以备拷贝，按 Ctrl+A 是没用的，可以双击 D1工具。

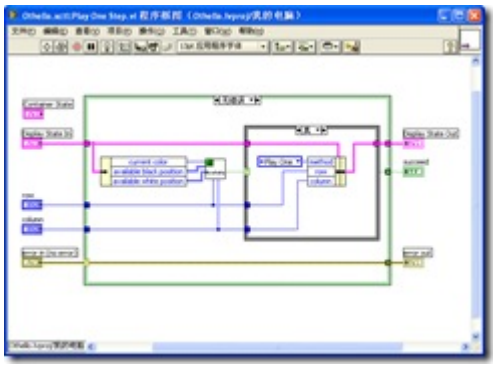
双击 D2，会出现一个字体设置对话框，在这里改变写到图标上的字体。如果要在图标上写英文，最适合的字体是“Small Fonts”，字号8。

开发 XControl 7 - 方法

方法与属性类似，它在控件的调用节点中出现。与属性不同，属性通常就是指某一个数值，而方法可以有多个参数，同时读写多个数值。

方法的创建和实现方法都和属性类似。它对应的 VI 所作的工作也是读写 XControl 的状态。

黑白棋中有一个方法：“走一步棋”。它的实现如下：



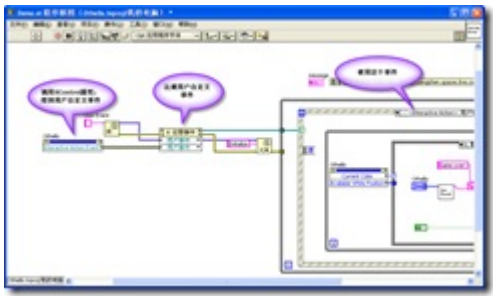
首先，判断落子的位置是否合理。如果是，则修改状态中相应的数据，落子位置。走一步棋之后，控件的数据和外观都需要做相应修改的。这部分修改没办法在方法 VI 中完成，只能在外观功能 VI 中实现。当方法 VI 修改了 XControl 的状态后，外观功能 VI 的“显示状态更改”事件会立刻被触发，所以相应代码可以放在外观功能 VI 的显示状态更改事件处理分支中。

开发 XControl 8 - 事件

非常遗憾的的是，XControl 实例控件的事件不能够自定义。我们只能够通过用户自定义事件来实现这一功能。实现的方法是，先造一个用户自定义事件，在 XControl 的状态中把它保存下来，为它写一个 XControl 属性，这样用户就可以在程序中得到这个自定义的事件。用户在程序中把这个事件注册到需要接收事件的事件处理结构上，以后就可以接收来自 XControl 控件的事件了。

事件的生成和抛出在前面两节中介绍过了（开发 XControl 5 - 其他功能 VI，和开发 XControl 4 - 外观功能 VI）。下面看一下如何在用户界面中使用这个事件：

通过 XControl 的属性得到在 XControl 中创建的用户自定义事件，在用户应用程序中注册这个事件，然后就可以接受 XControl 抛出的该事件了。

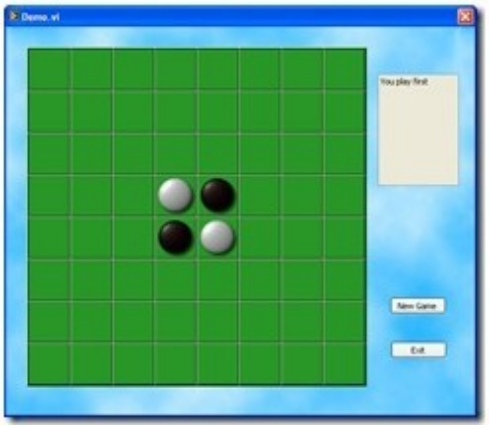


开发 XControl 9 - 使用 XControl

关于 XControl 还有好几个方面的知识点没有介绍到，包括：版本控制、错误处理、得到调用 VI 的信息、调试、调整界面大小、发布快捷菜单、动画的实现、一些注意事项等。不过，这些细节问题在这个黑白棋控件中没有体现出来。所以以后有机会再讨论。

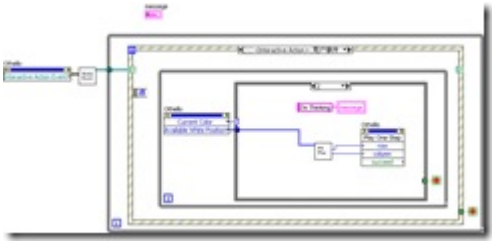
作为这一组 XControl 话题的结束篇，介绍一下演示使用黑白棋控件的范例。

这是演示程序的界面，只有黑白棋 XControl 控件和必要的几个控件。



其程序框图如下，这是一个典型的事件处理结构。

首先程序注册必要的事件：一个 XControl 的事件，在用户走子后通知应用程序;一个用户自定义事件，这里仅用于初始化。



程序初始化，与用户点“New”按钮做的事情相同，都是调用黑白棋控件的 New Game 方法，开始一盘新的游戏。

之后就等待用户(黑方)在棋盘上走一子。之后，程序判断应该黑方走还是白方走。如果轮到白方走，程序就在所有可以落子的地方随机选出一个位置，走一白子。(程序没有实现人工智能部分)

黑白都不可走时，程序计算输赢。

估算项目工时

一个项目在前期调研的时候就要估计一下项目开发的周期大约有多长。有很多不同的估计方法，适合不同的项目类型。我平时设计 LabVIEW 编写的应用程序中用到过三种估计方法：代码量度量（Size-Based Metrics）、工作量估计（Effort Estimation）和专家估计（Wideband Delphi Estimation）。

代码量度量的估计方法就相当于使用其它文本编程语言时的代码行估计法。一个软件需要多商行代码、每行代码要花多少时间，是相对来说比较容易统计的。所以代码行估计法是最流行的估计项目工时的方法之一。LabVIEW 的代码不是按行来计算的，它以节点数为计量单位。

在 LabVIEW 的菜单上选择 Tools->Profile->VI Metrics 就可以调出如下的面板。这个 VI 度量面板可以帮你统计的的 LabVIEW 代码中总共有多少个节点。

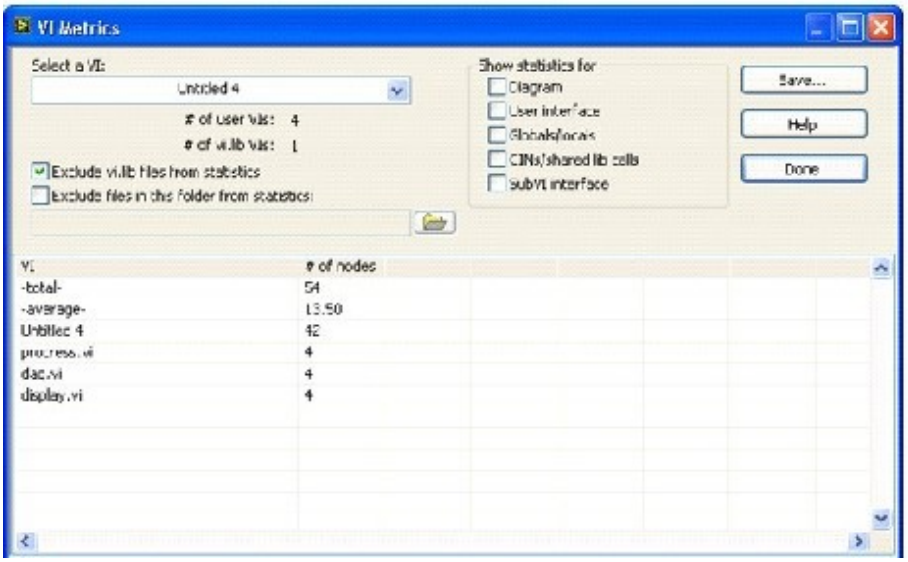


图1：VI 度量工具

利用代码量度量方法估算工时的具体实施步骤大致如下：

首先，先把项目拆分成小的模块。功能单一的小模块更容易进行准确估算。然后估计实现每个小模块需要多少 LabVIEW 代码（节点个数）。所有模块节点数的和就是整个项目所需的节点数，每个节点所需工时是已知的，所以整个项目的工时也就估计出来的。

利用代码量度量方法估算工时，是需要有一些历史经验才行的。比如某种规模的功能模块到底需要多少节点，只有有过项目经验，统计过，才能心里有谱；写一个节点需要多少时间，对于不同类型的公司，不同经验的程序员，这一数值都是不同的。自己公司的每节点编程耗时，也只有做过之后才有数。

如果缺少历史统计数据，可以使用精确度稍差一些的工作量估计法来估算项目工时。工作量估计法与代码量度量方法是很类似的。首先要把项目拆分成便于估算的小模块。但是，

由于不便于对程序节点数进行估算，就只能评直觉，估计每个模块所需的工时，然后累加出项目总工时。

Wideband Delphi Estimation 方法也可以用于缺少历史统计数据的情况，并且它的结论比工作量估计法要精确。只是这种方法实时起来比较麻烦，一般只有比较重要的项目，我们才会用此方法。

Wideband Delphi 方法的实时过程大致如下。首先组织一个十人左右的团队，队员是来自不同部门但与此项目紧密相关的人，比如开发者、文档人员、测试人员等等。

让所有队员各自估计一下项目所需的时间。

把所有人凑到一起，把估计的结果汇总，让大家看看估计值的分布情况。然后由每个队员讲一下影响自己做估计的因素，包括有利的因素和不利的因素。比如有人会提出，我们的项目与某某项目很类似，可以借用那个项目中的很多代码。另一个人说，我们的项目用到一个什么非常难掌握的技术，可能会花费很多时间等等。

等大家讨论过后，再各自重新估计项目的工时。

然后在汇总，在讨论……

经过几轮讨论估计后，大家估计出来的工时的差距就比较小了。取大家的平均值作为最终结果即可。

Palette API

对函数和控件面板的调整，并不是只有手工调整着一个方法。LabVIEW 8.6 中提供了在程序中修改函数和控件面板的 API。通过这些 API VI，用户就可以在程序里读取或设置 .mnu 文件里的内容，从而通过编程来改变函数和控件面板。