LabVIEW开发技术丛书

LabVIEW 深入探索

gsdzone.net/community

GSD zone.net

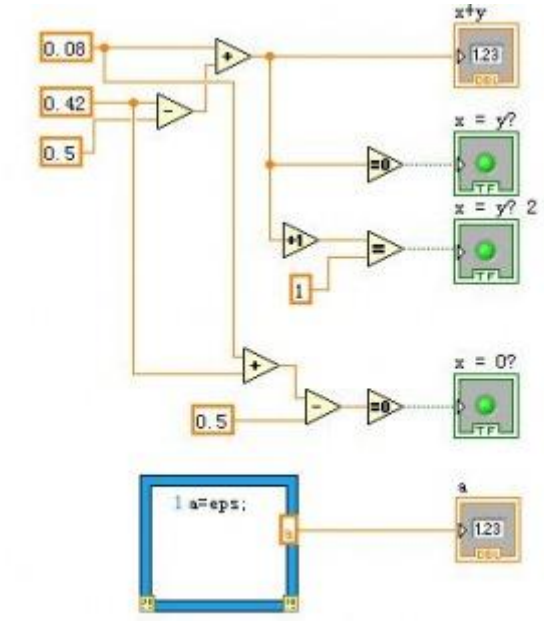
目录

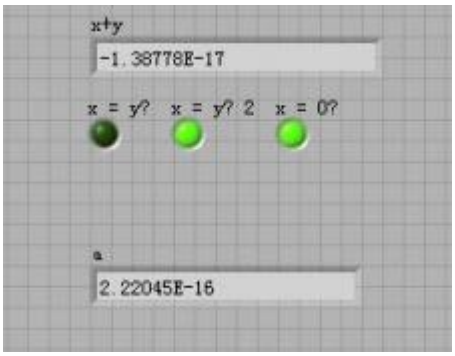
双精度数据不适于相等比较（内置函数）

日前帮朋友看一个程序，一个 WHILE 循环，退出条件是等于一个双精度数，结果程序未按照预想条件退出，由此联想到了双精度数的精度问题。

由于计算机的计算位数有限，所以双精度数都存在精度损失的问题，因此一般不宜用相等函数进行判断。

对于下面的例子





通过例图可以看出：
0.08+（0.42−0.5)=0，实际上是 x+y 的指示结果，使用等于 0 来判断，结果不等于 0（x=y?）

而下面（0.08+0.42）−0.5 =0，使用等于 0 来判断，结果等于 0（x=y?2）

将上面不等于 0 的结果在 + 1，使用相等来判断，结果却等于 1。

由此可见，双精度浮点数的确不适合用相等来判断。

顺序结构是“结构”吗?

LabVIEW 中的"结构"概念同 C 语言有很大不同，C 语言中中的结构指的是复合数据类型，而 labVIEW 中所谓的结构相当于 C 语言的程序运行结构，包括循 环、顺序结构、条件结构、事件结构等，这里面尤其需要强调的是顺序结构，无论是平铺式顺序结构还是堆叠式顺序结构，NI 都不建议使用，原因主要有以下几点：

1：强行规定的动作次序，影响了数据流的传递方式。
2：内存使用上，顺序结构比较同样性能的数据依赖关系的数据流，程序框图所占的内存空间比较大。

我们在作 VI 性能分析的时候，也能看到一个有趣的现象，LV 虽然把顺序界结构放在函数选板的结构子类中，但是在统计中，顺序结构并没有被看成是结构，而是作为一般的节点，其中每增加一个祯就增加一个节点，所以一个复杂的顺序结构会增加大量的节点，从这个角度也可以说明，从 LabVIEW 的内部来看，顺序结构并不时真正意义的运行结构。

从上面的例图可以看出：顺序结构并没有被统计为结构，而只是一般的节点，每一个 FRAME 都是一个节点。

在看下面的例图

通过上面的例子可以发现，for ,while ,case 都是结构，本身也都是一个节点。

到底什么是"节点"

即便是用了 LV 多年，有些基本概念还是非常模糊的，比如"节点"和"函数"(NODE AND FUNCTION)，我们称 LV 本身提供的函数为节点，或者节点函数，那自己做的子 VI 被调用时算不算节点，它内部包含的下一级别的子 VI 是不是节点那？控件是不是节点?装饰是不是节点?程序的结构比如顺序结构、循环结构是否是节点那?

LV 经常用节点的数量来统计 VI 的性能，所以了解节点的真正含义是非常有必要的。

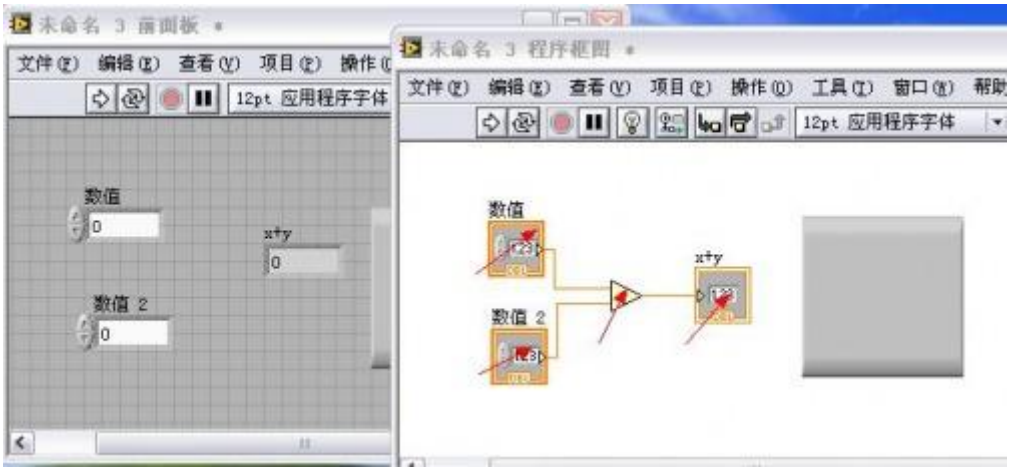
首先看看帮助文件对节点的定义：

节点是程序框图上的对象，带有输入输出端，在 VI 运行时进行运算。节点类似于文本编程语言中的语句、运算符、函数和子程序。LabVIEW 有以下类型的节点：

函数 – 内置的执行元素，相当于操作符、函数或语句。
子 VI – 用于另一个 VI 程序框图上的 VI，相当于子程序。
Express VI – 协助常规测量任务的子 VI。Express VI 是在配置对话框中配置的。
结构 – 执行控制元素，如 For 循环、While 循环、条件结构、平铺式和层叠式顺序结构、定时结构和事件结构。
公式节点和表达式节点 – 公式节点是可以直接向程序框图输入方程的结构，其大小可以调节。表达式节点是用于计算含有单变量表达式或方程的结构。
属性节点和调用节点 – 属性节点是用于设置或寻找类的属性的结构。调用节点是设置对象执行方式的结构。
通过引用节点调用 – 用于调用动态加载的 VI 的结构。
调用库函数节点 – 调用大多数标准库或 DLL 的结构。
代码接口节点(CIN) – 调用以文本编程语言所编写的代码的结构。

这里函数的概念本身就不好理解，内置的执行元素，加减运算符号算一个节点,一个文件操作函数内部包含大量的子函数或者子 VI,也是一个节点?
我们自己制作的 VI 称作子 VI,它可以理解成函数吗?从分类上看显然不是,那它到底有那些不同呢?

下面是简单的 vi 例子，有 2 个输入控件和一个输出控件，一个运算内置函数和一个装饰件。



下图是对上图的分析结果

VI	节点数量	输入控件	显示控件	属性读取	属性写入
总计	4	2	1	0	0
平均	4.00	2.00	1.00	0.00	0.00
未命名 3	4	2	1	0	0

显然,两个输入控件+一个显示控件+运算符号共四个节点,前面板的装饰当然也是控件，因为我们可以得到它的参考，进而控制它的属性，从分类上看，它也是继承于最基本的图形对象的，所以装饰是控件，但不是输入控件，也不是输出控件，换个角度看，它没有数据的流入或者流出，因此,装饰不属于节点。

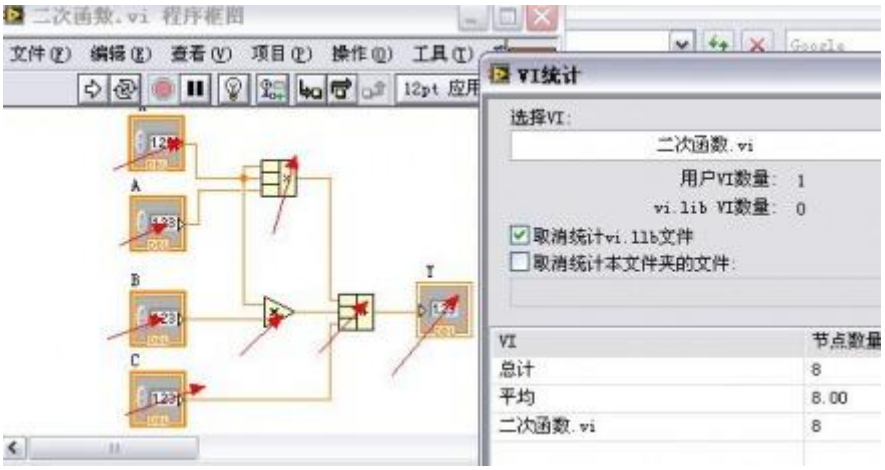
下面再看一个例子

从上图可以看出,常量,结构,属性节点都属于节点,一个属性节点可以同时控制多个属性，它仍然只算是一个节点，所以通过一个属性节点控制可以减少节点数量，全局变量和局部变量也都属于节点，因为它都涉及了数据的流入或者流出。

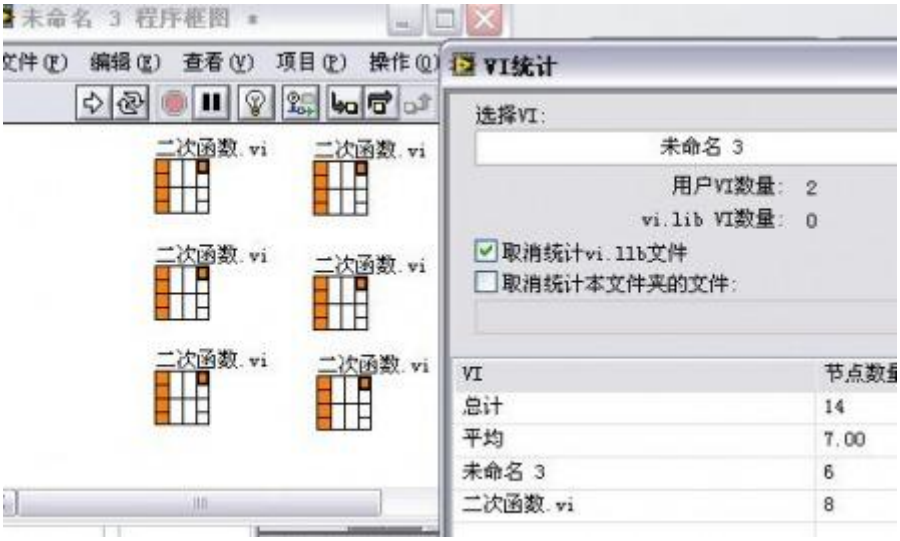
我们感兴趣的是自己做的子 VI 和内置函数有何不同之处

上图中，顶层 VI 有三个节点，但是 EXPRESS VI 内部包括 53 个节点，自己做的 SUBVI 内部包含 8 个节点，而 WRITE SPREAD SHEET 尽管内部包含大量操作，我们可以打开跟踪，但是它只是一个节点，从这里可以看出,经管 EXPRESS VI 使用非常方便，但是 LV 没有把 EXPRESS VI 称为函数，而写文本文件 VI 尽管内部非常复杂,但是仍然是一个节点。

所以使用 LV 内置的函数可以提高程序运行效率,而 EXPRESS VI 尽管使用方便，但是效率很低.



上图表明，我的计算二次函数 VI，的确内部包含 8 个节点，那么如果我多次调用它，情况会如何那?



二次函数本身 8 个节点没有变化，每多调用一次，节点增加一个，6+8=14 个节点.
对于经常使用的操作，如果作成子 VI，可以有效地减少节点数量，提高运行效率.

为什么 LV 内置的函数只是一个节点那，而我们自己做的子 VI 却包含内部使用的节点那?

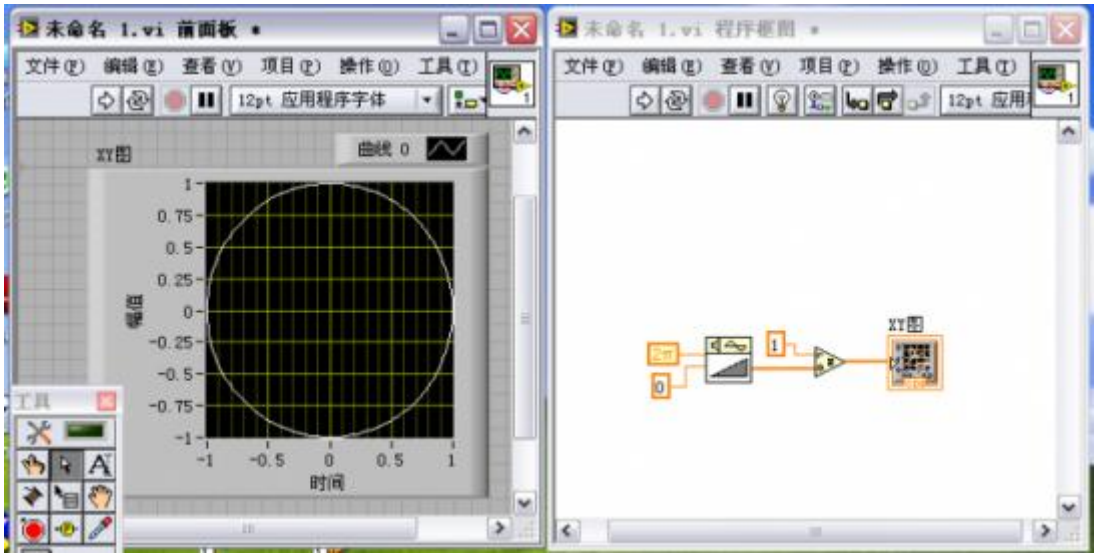
猜测 LV 启动后,很多内置的函数已经调入内存了,因此我们使用它不过是增加了一个节点,而我们自己做的 SUBVI 则不同,它是添加在程序框图中被调入内存的,所以节点数量应该包括它内部使用的节点.

XY GRAPH 的输入参数形式

本文探讨一下 XY GRAPH 可以接受的输入参数类型.

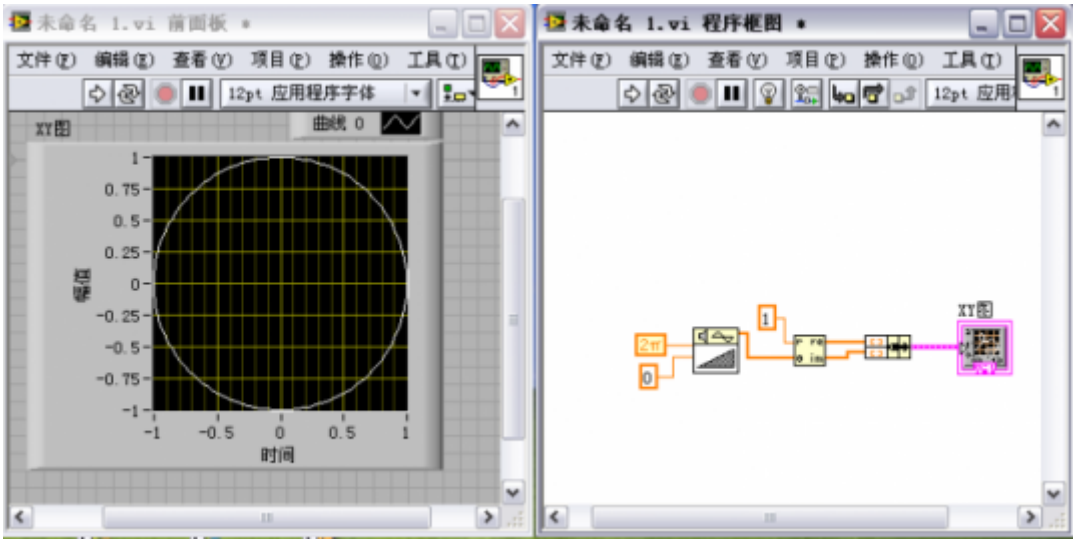
一、复数构成的数组

在 XY 坐标系中，一个（X，Y）坐标可以用一个点来表示，而复数 X+Yi 也可以表示一个点，因此，一个复数组成的数组就是一个有点的坐标组成的数组。

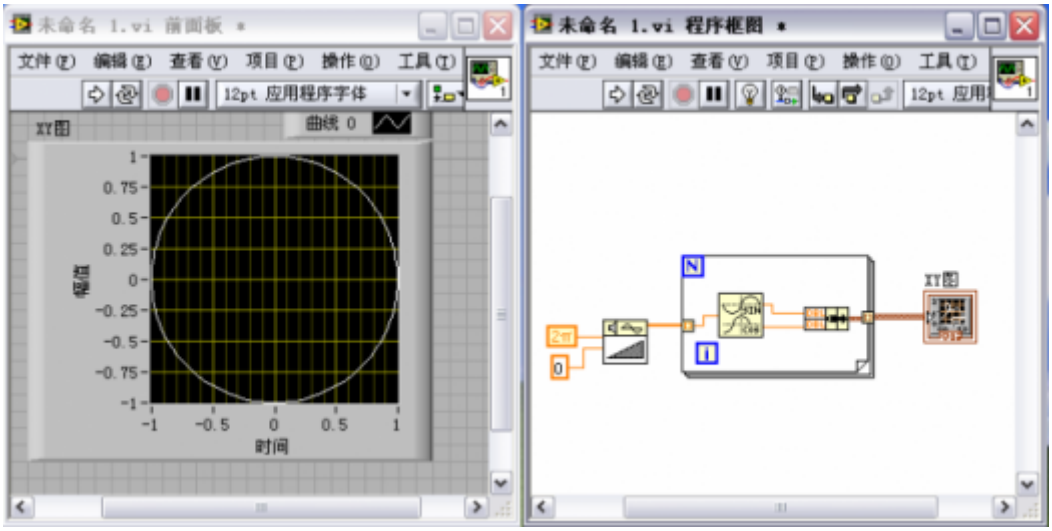


二、两个长度相同的一维数组构成的簇

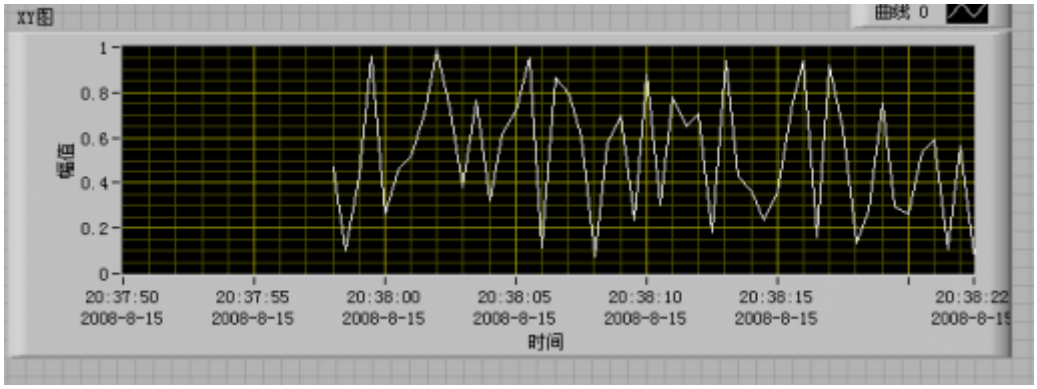
两个长度相同的一维数组打包成簇，LABVIEW 自动解包，并把两个数组的对应元素解释成点。

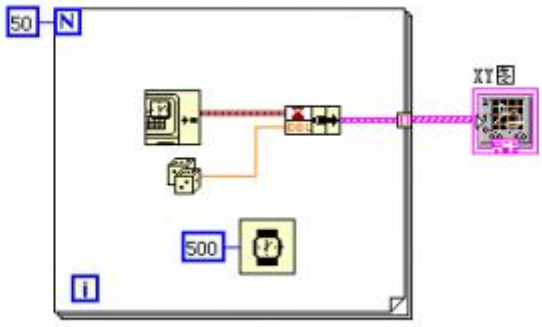


三、簇数组，每个簇代表一个点

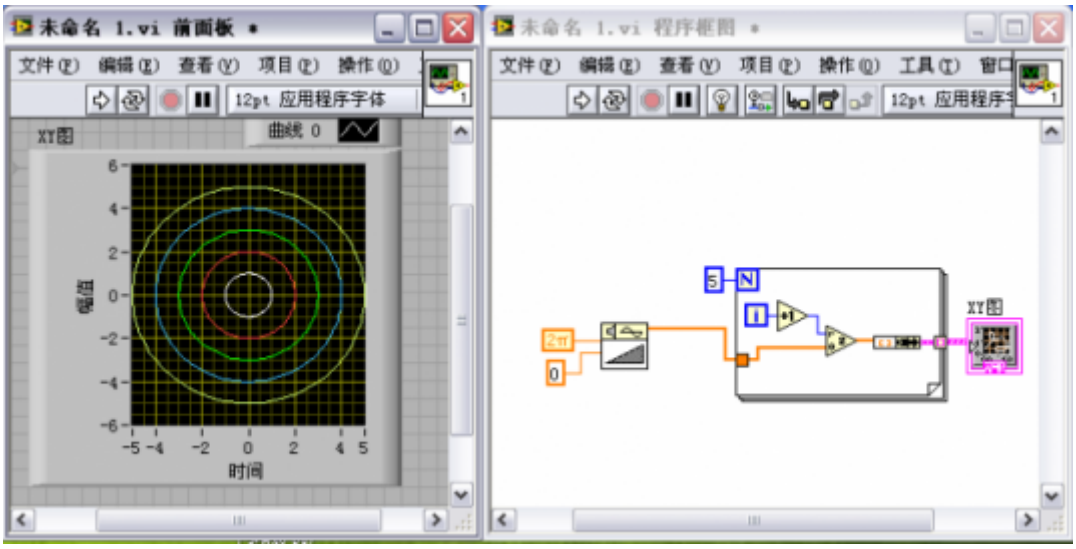


四、时间作为 X 轴

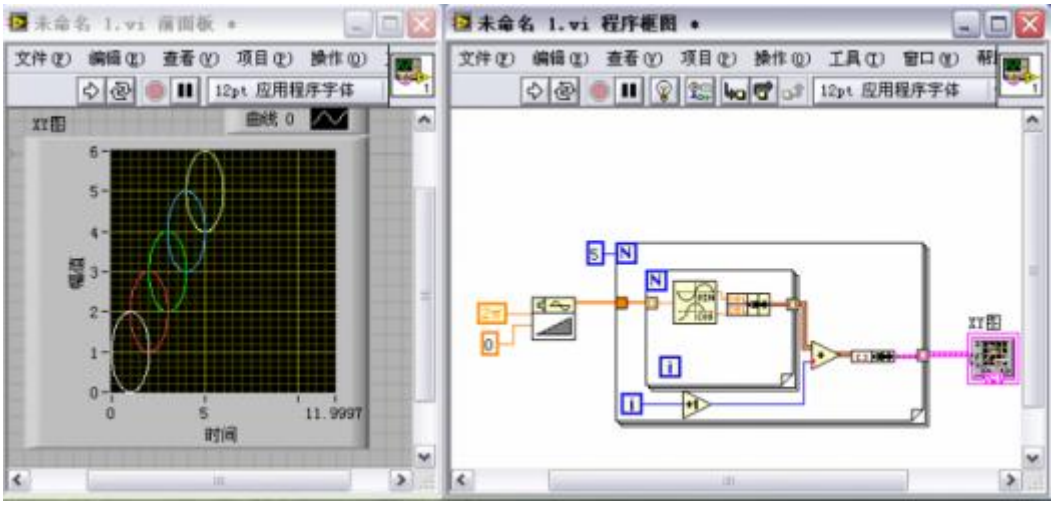




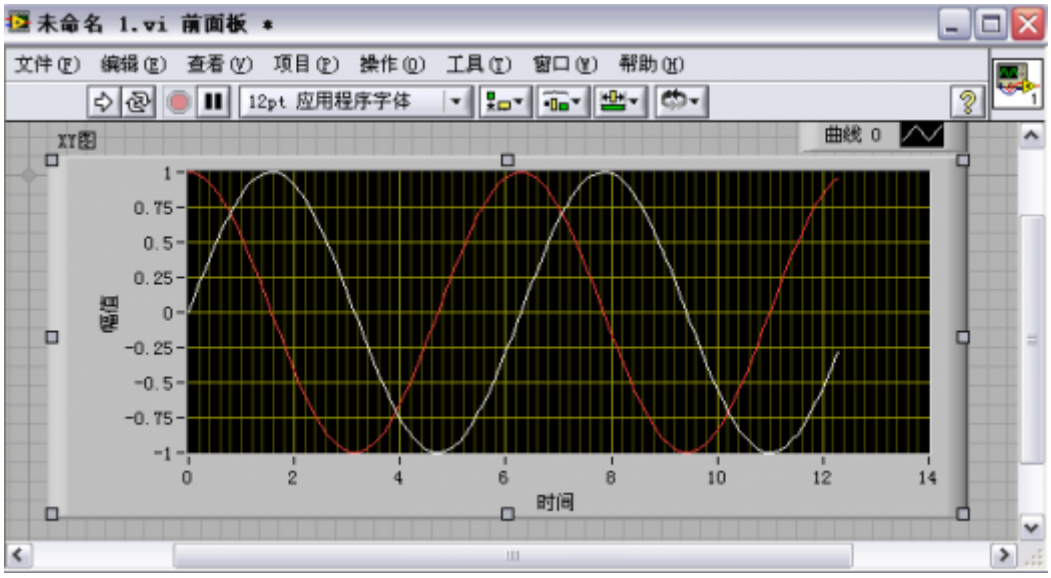
五、多条曲线输入方法：

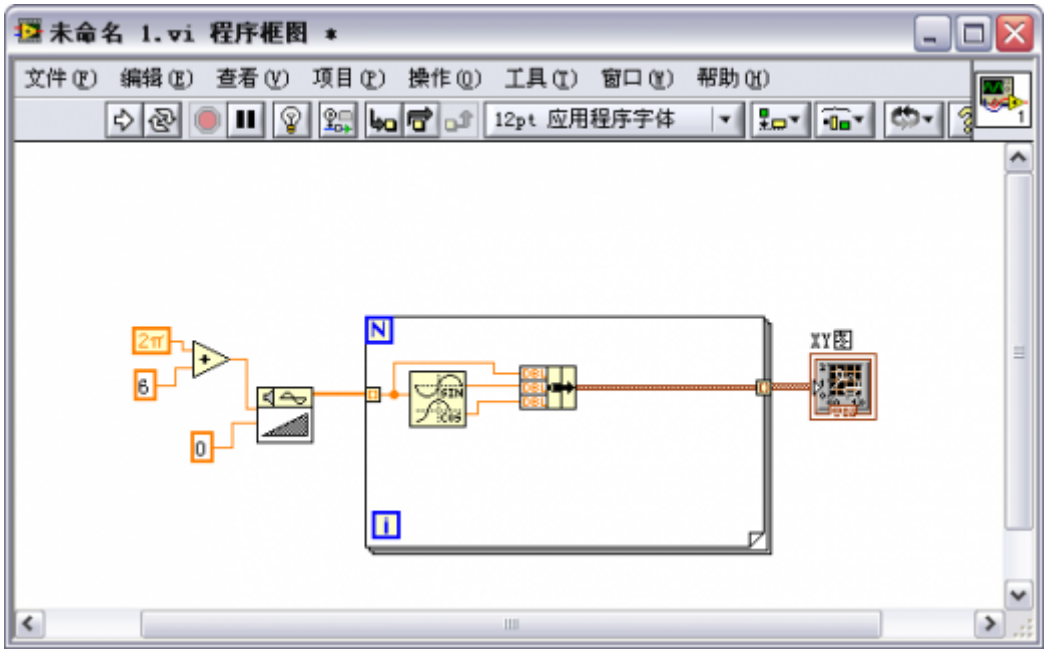


也可以采用点簇的方式

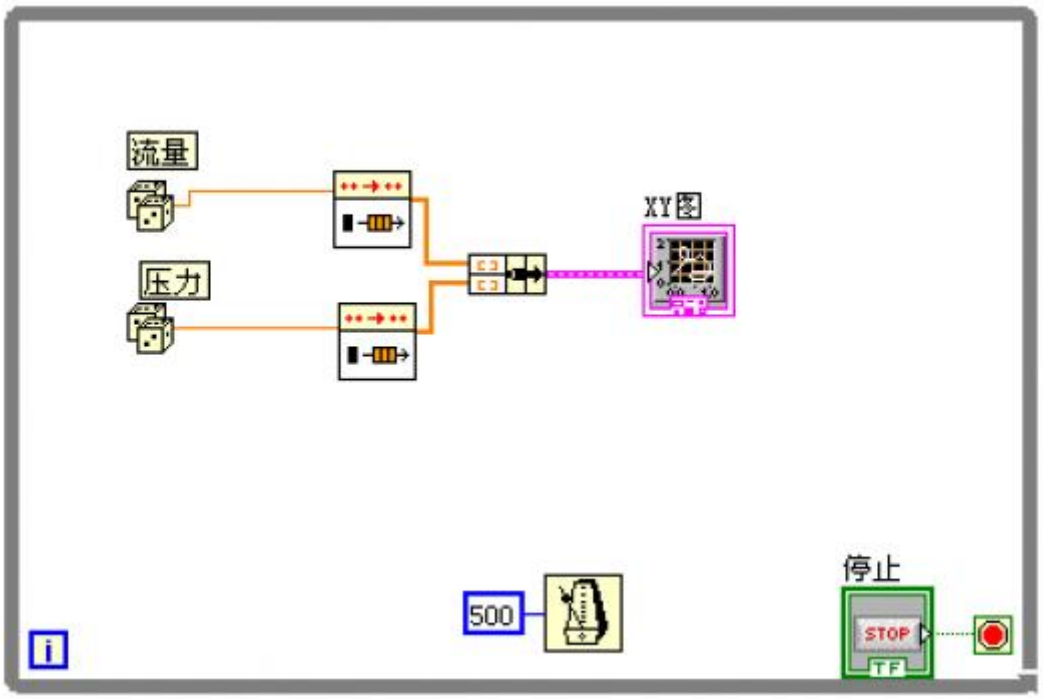


六、对于绘制两条曲线，网友试验出一种特殊用法





关于网友流量压力曲线问题:



整型数据类型和内存映射

首先要区分的是控件和数据类型的区别.

控件是数据类型的容器,或者说数据类型是控件的一个属性,控件都有一个值的属性,这个值的类型就是控件所代表的数据类型.

描述一个控件是通过类型描述符实现的,它包括控件的名称、控件类型及控件所代表的数据类型等等。

今天要谈的与控件本身无任何关系，是数据类型在内存中如何存储的，或者称作数据的内存映射。

我们知道，无符号整型数有 U8、U16、U32、U64

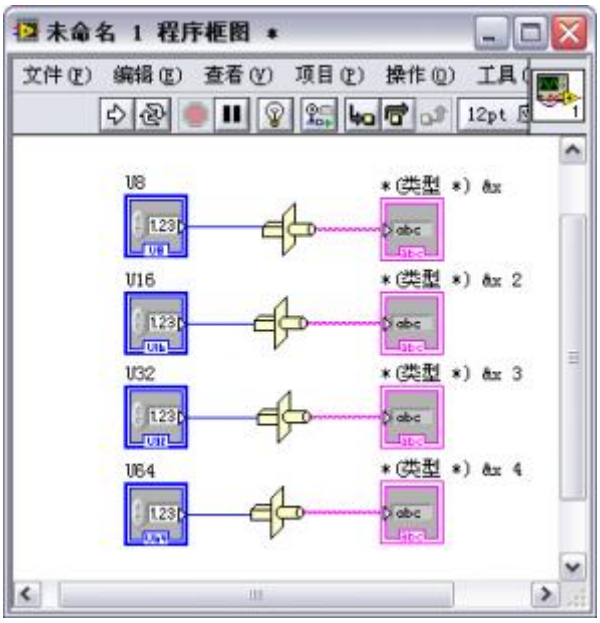
U8 是指 8 位（BIT），一个字节（BYTE），　　值范围：0X00----->0XFF
U16 是指 16 位（BIT），二个字节（BYTE），　　值范围:0x0000--->0XFFFF

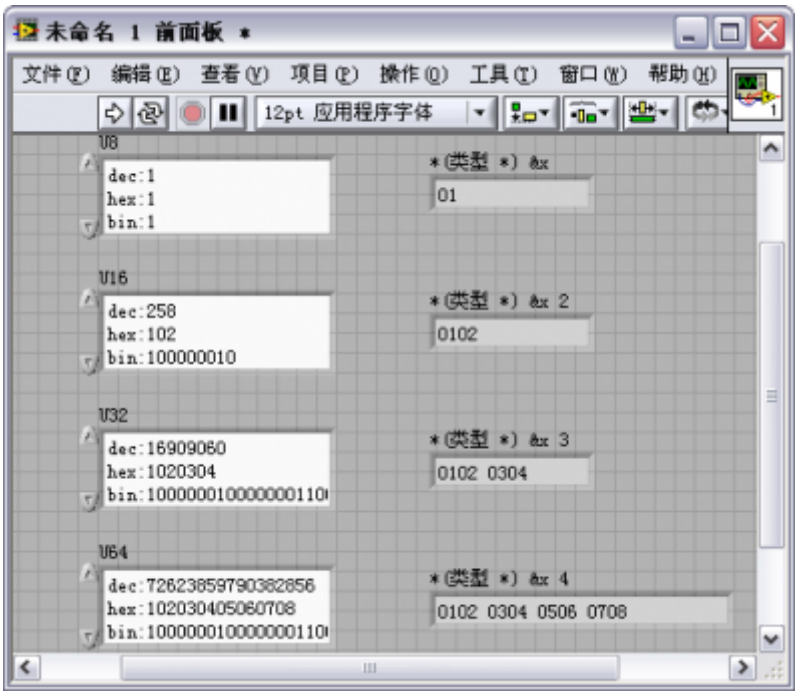
U32 是指 32 位（BIT），四个字节（BYTE），　　值范围：0X00000000---》0XFFFFFFFF

U64 是指 64 位（BIT），八个字节

标准的数据类型在内存中是连续存放的，比如 U32，是四个字节，那么一个 U32 就占用连续的 4 个字节的内存空间，同理，U16 占用 2 个字节的空间，U64 占用 8 个字节的连续空间。单精度是 4 个字节，双精度是 8 个字节。

通过下面的例子，可以清楚地看到，数据类型与字节的关系。





上面的图中的字符串是用 HEX 方式显示的，清楚地表明了 U16--》2BYTE，U32--》4BYTE，U64----》8BYTE

以 U32 为例，它由四个字节组成，而这四个字节，可以理解成很多方式，它可以是

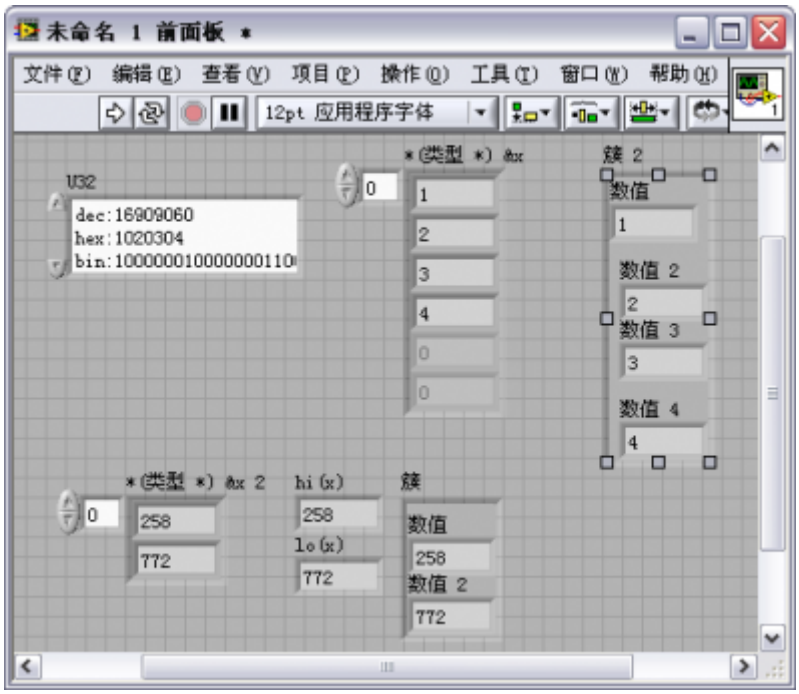
字符串------四个字符，每个字符一个字节
U8 数组------四个元素，每个元素一个字节
U16 数组---二个元素，每个元素占两个字节

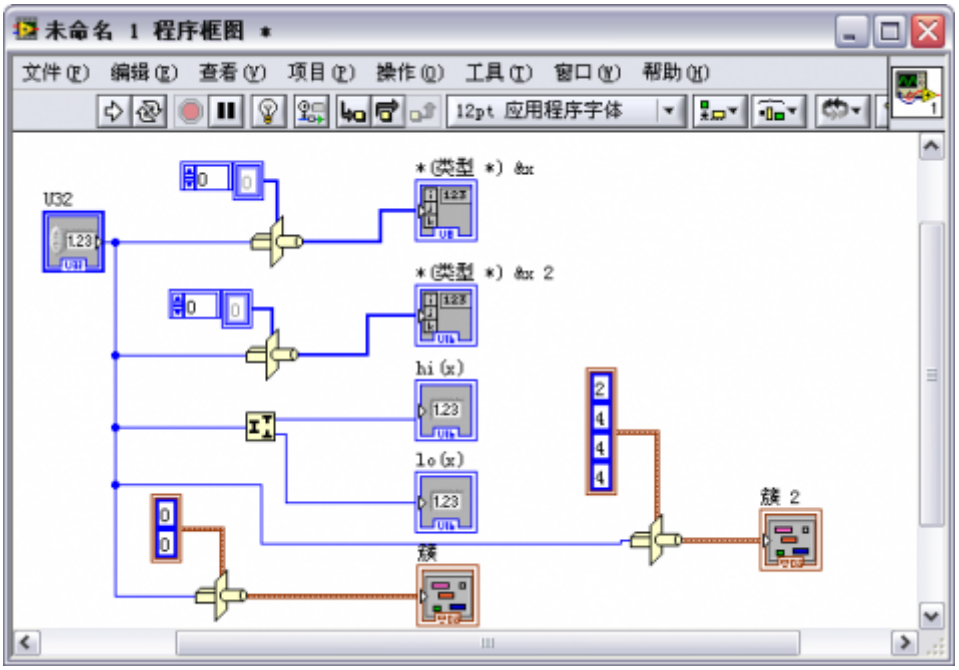
LABVIEW 的簇与 C 的结构不同，它是连续按字节存放的，未采用对齐方式，因此我们甚至可以理解成一个簇

这个簇可以是：
四个 U8 元素、2 个 U16 元素、一个 U8，一个 U16 和一个 U8，总之，只要是四个字节就可以，对内存本身来说，是无法判断它到底存储的是什么的。

LABVIEW 中有一个“CAST”函数，中文版翻译成强制类型转换，CAST 本意是铸造模型的意思，用它的本意比较合适，对于四个字节，我们可以按照我们自己的理解转换成任意数据类型，只要它的字节数相同。

如果我们熟悉数据在内存中的映射关系，用 CAST 函数可以解决一些特殊的类型转换问题。





U32 转换成 U8 数组和 U8 组成的簇结果相同
直接用 CAST 把 U32 转换成 U16 数组和 U16 组成的簇与 LV 内置的拆分函数结果相同。

状态机的基本概念

状态机不是 LABVIEW 独有的概念,早在 LABVIEW 诞生之前,就有了状态机(STATE MACHINE)的概念,只所以在 LABVIEW 编程中经常强调状态机是因为 LABVIEW 特有的图形编程方式特别适合于采用状态机模式编程,在 PLC 中 有流程图的编程方式,从本质上说,那是一种特殊的状态机。

STATE MACHINE 包括三个基本要件，STATE、EVENT 、ACTION，状态、事件和动作

状态：是一个抽象的概念在一定条件下或者一定时间内保持不变，等待一个或几个事件的发生，命名状态时，往往可以用等待--来定义。

事件：是一个瞬时的概念，表示某件事情发生了，一旦有关的事件发生了，势必要采取某种动作。

动作：表示一旦事件发生，采取何种处理方式，处理的结果就是另一个稳定的状态。

状态（等待事件）---》事件发生----》采取动作----》另一个状态。

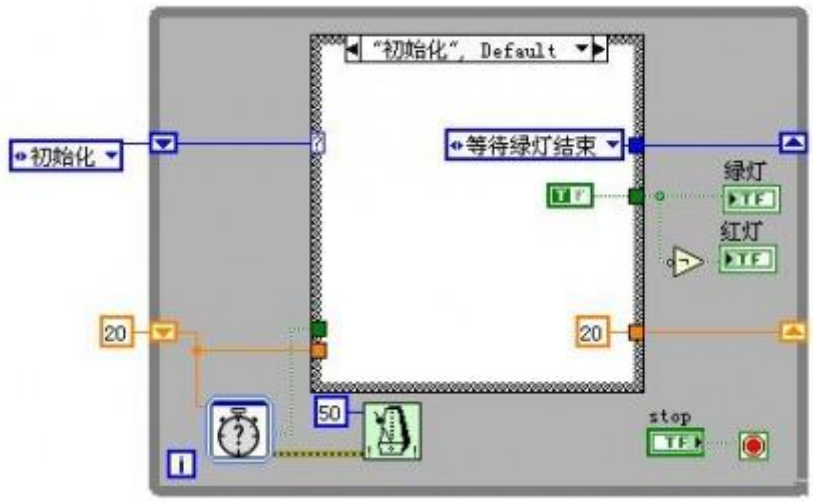
状态机的概念是非常简单的，越是简单的东西越不容易处理，原因是简单则限制少，则灵活，状态机设计的好坏完全取决于编程者的水平，这不仅仅指 LABVIEW 编程的水平，更重要的是编程者的逻辑思维，一个好的状态机的设计，关键是如何定义状态，状态少了，则意味着每一个状态中要处理的事务多了， 状态多了，则整个状态机就变的复杂了。

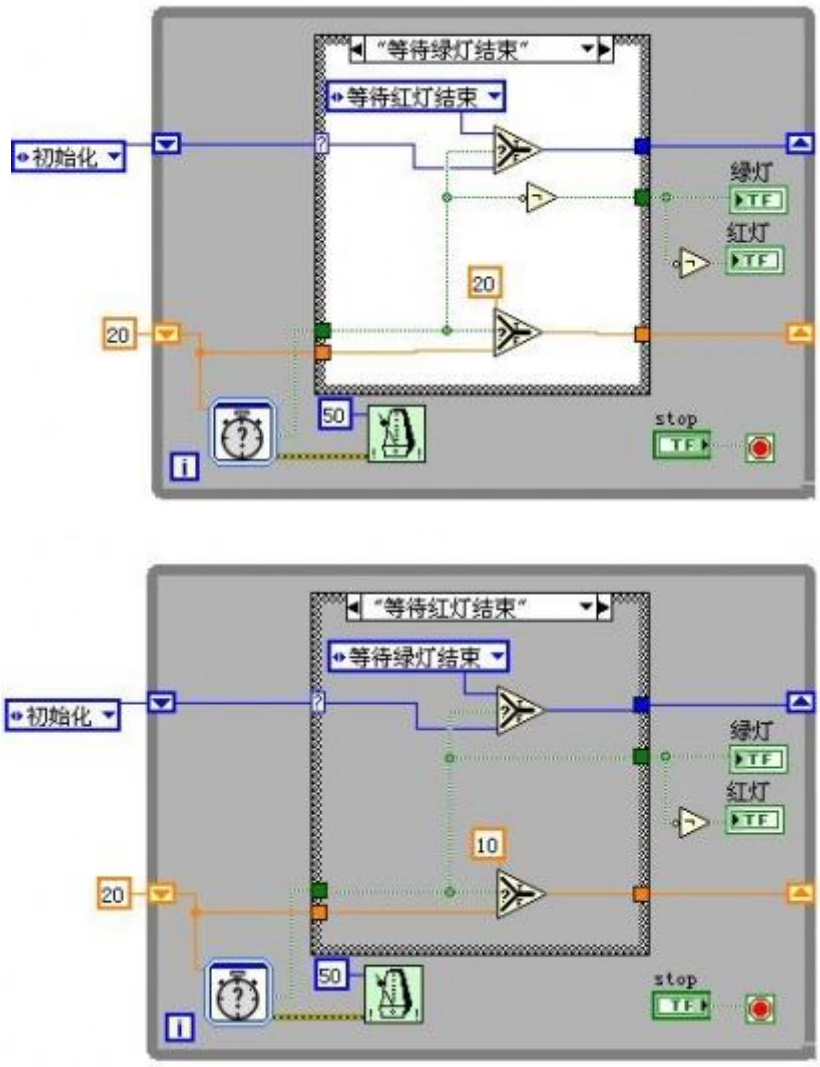
以一个简单的红绿灯控制来说，简单地说有两个状态，红色状态和绿色状态，事件是绿灯时间到和红灯时间到，绿色时间到触发的动作是，绿色灯灭，红色灯量，红色时间到触发的动作是红色灭，绿灯量。

我过去的文章中多次提到 ACTION ENGINE 的概念，它与状态机的区别是，它仅仅定义了 ACTION，而没有定义 STATE 和 EVENT，因此，它的 ACTION 完全取决于编程者，而状 态机自己本身就可以根据内部或者外部条件的变化，自动采取相应的动作，转入其它的状态，实现控制自动化。

有多种形式的 STATE MACHINE，我将在后续的文章中陆续介绍，先看一下红绿灯的实现过程，假如绿灯亮 20 秒，红灯亮 10 秒。

先定义三个基本状态：初始化（仅在第一次调用时发生），等待绿灯结束和等待红灯结束。
LABVIEW 的严格枚举数据类型是 LABVIEW 状态机定义状态的最好工具，使用状态机时，必须要使用这种数据类型，好处是增加或者减少状态，程序可以自动更新。





状态机的基本类型顺序结构

前文介绍了 LV 编程的重要概念---状态机,状态机是个基本概念或者说理论,其具体表现形式多种过样,很难具体分类,我根据个人在编程实践中的体会,归纳几种常见类型.

一、顺序结构

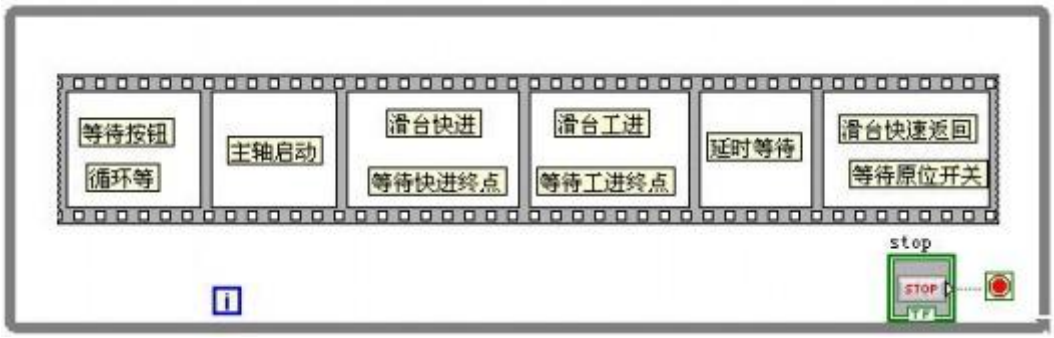
LV 本身是有顺序结构的，而且有两种方式，STACKED （堆叠）和 FLAT FRAME（平铺）。

顺序结构的状态机更象是堆叠顺序结构,不过二者的区别在于 LV 本身的顺序结构是强制的,无法中间退出的,而状态机的顺序结构是采用的循环扫描的方式.我举一个例子来说明一下:

一个加工零件的程序,过程如下:

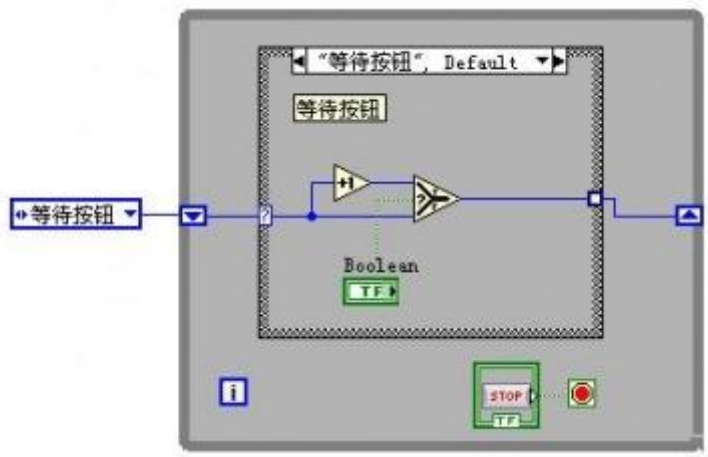
等待启动按钮--->主轴启动--->滑台快进----->滑台工作进给--->终点延时--->滑台快速返回原位--------->等待启动按钮(下一次循环)

这里的启动按钮可以是界面上的按钮也可以是操作台上的按钮.用普通顺序结构应该是(我们用平铺更容易理解)



我们注意到,每个动作(FRAME)内部都是一个循环结构,需要等待一定条件后,如果条件满足,转入下一个FRAME,我们没有任何办法改变它,比如我们 有一个急停按钮,则需要在每一个 FRAME 中检测急停,如果急停生效,退出本 FRAME,转入下一个 FRAME,依然要判断急停,直到所有的 FRAME 都 完成才能退出,
在这个过程中,外层循环需要所有动作完成后才执行下一循环,对它改造一下,就可以形成顺序状态机结构.

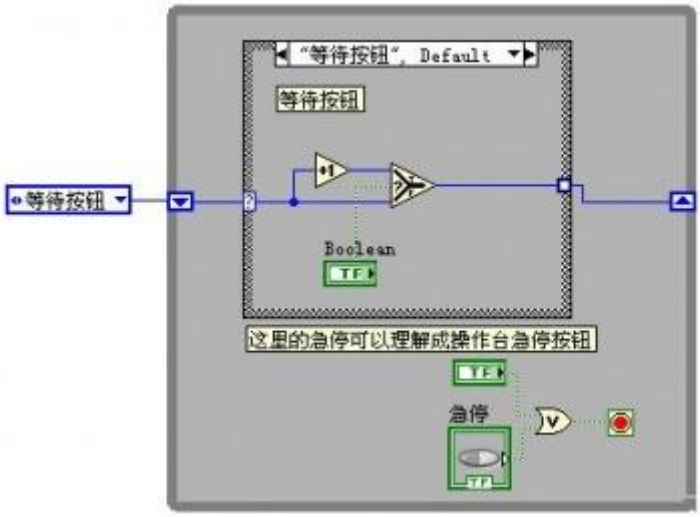
我曾经提到过,严格类型的枚举是状态机的核心要素,我们先构造一个严格类型的枚举.枚举变量有一个特点,当最后的元素执行加一操作时返回第一个元素.



这个状态机的功能和上面的顺序结构完成的功能是相同的,区别在于每个 CASE 不存在循环等待了,

整个循环过程都是在外层循环中实现的,外层循环不断地更新,如果转换条件(事件未发生),下一次循环仍然执行前一个 CASE(状态不变),如果转换条件满足(事件发生),采取加一的动作(ACTION),转入下一个状态.

这样做和上面的顺序结构比较,好处是显而易见的,整个过程只有一个循环,而上面的顺序结构实际每个 FRAME 都是一个循环,这样,如果有急停按钮,我们可以在 CASE 外面检测急停,如果急停,可以马上停止循环,而在顺序结构中,需要在每个 FRAME 检测,并逐个退出.



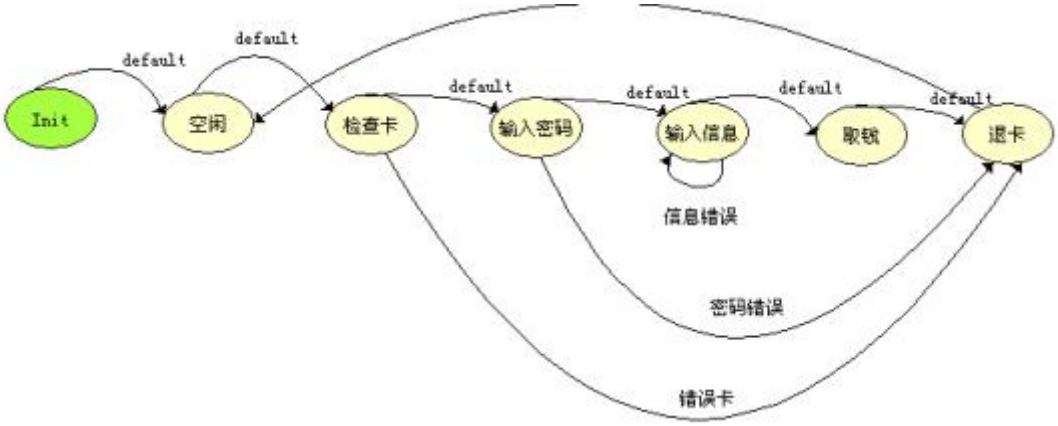
.

状态机的基本类型之标准状态机

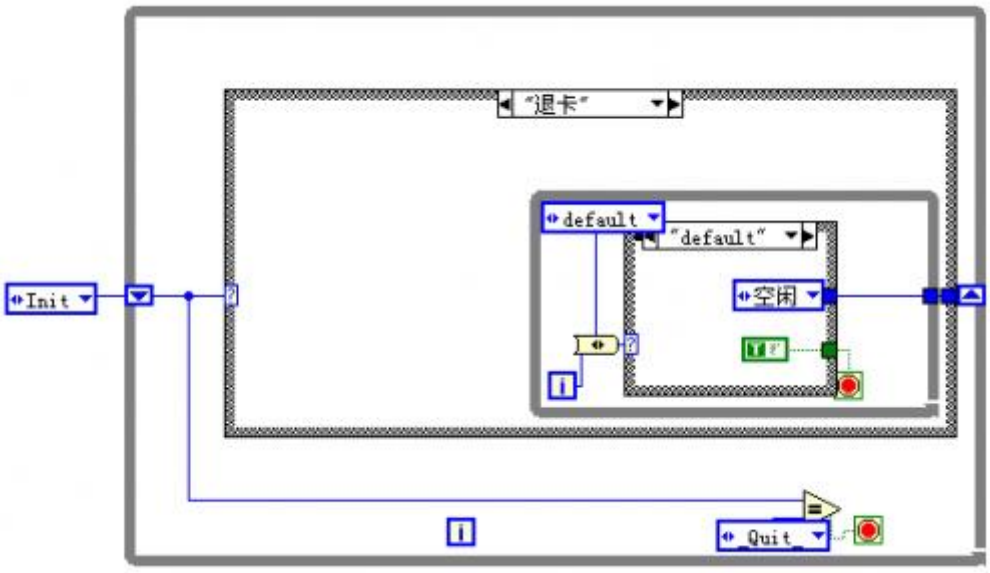
标准状态机是使最为广泛的状态机，也称为古典状态机或者经典状态机。LV 的标准状态机 VI 模板用的就是这种。

我把标准型状态机理解成条件跳转型状态机，也就是在一个状态中，根据不同的条件，跳转到另一个状态。标准型状态机非常适合用图形方式来描述，PLC 的 SFC 流程图就是我们说的标准状态机。

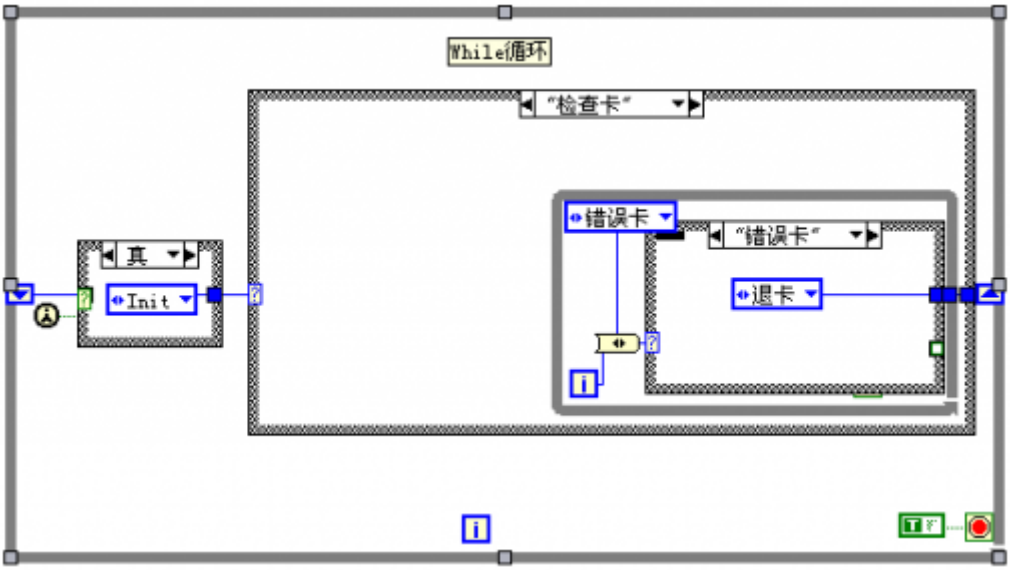
NI 提供了一个状态机的组件，它就是典型的标准状态机。看看它的用法。
NI 状态图组件,提供一个状态图编辑器,在编辑器中,可以设计各种状态和状态之间的转换条件,如下图



上面是我模拟了提款机做的示意程序,根据我们绘制的状态图,LV 可以自动生成状态机,可以选择两种方式.做为连续运行的 VI 和单次运行的子 VI

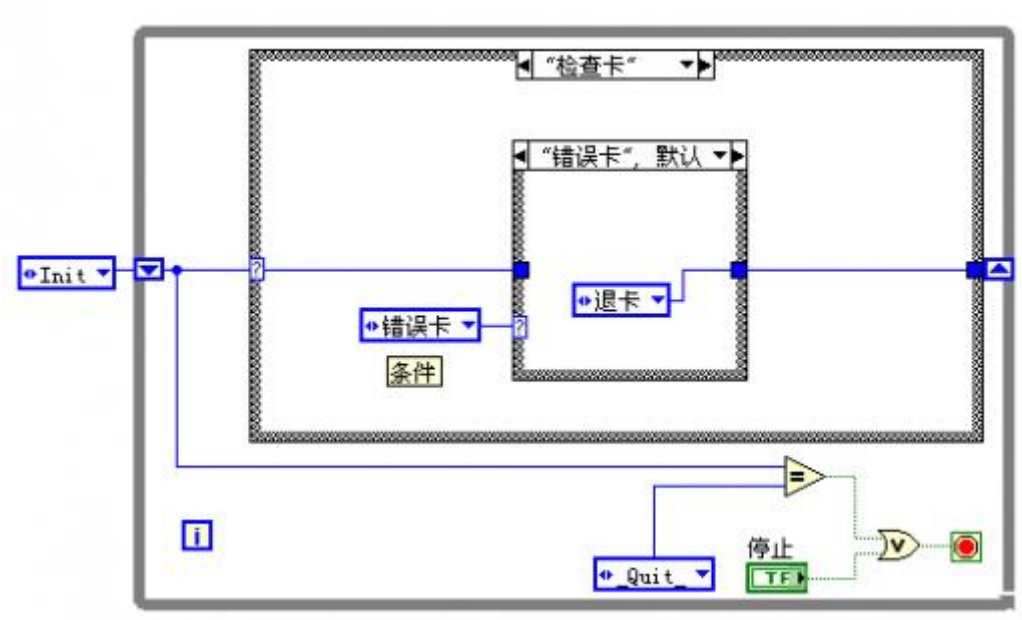


上面的图是连续运行模式.



这是单步运行方式,可以做为一个子 VI,由上一级 VI 调用

NI 的状态图组件显得比较烦琐,比较适合状态很多,条件跳转比较复杂的情况.
常用的标准状态机,自己编写用的比较多.下面已常用方式演示一下这种类型的状态机



标准类型状态机使用非常广泛,它的状态和转换条件都是可以预期的,而不随机的,对于类似于 WINDOWS 消息驱动的情况,由于状态是不可预期的,比如我们预料和控制用户对人机交互界面操作的先后次序,这种情况下,使用队列状态机是最为合适的,后续文章将专门介绍队列状态机

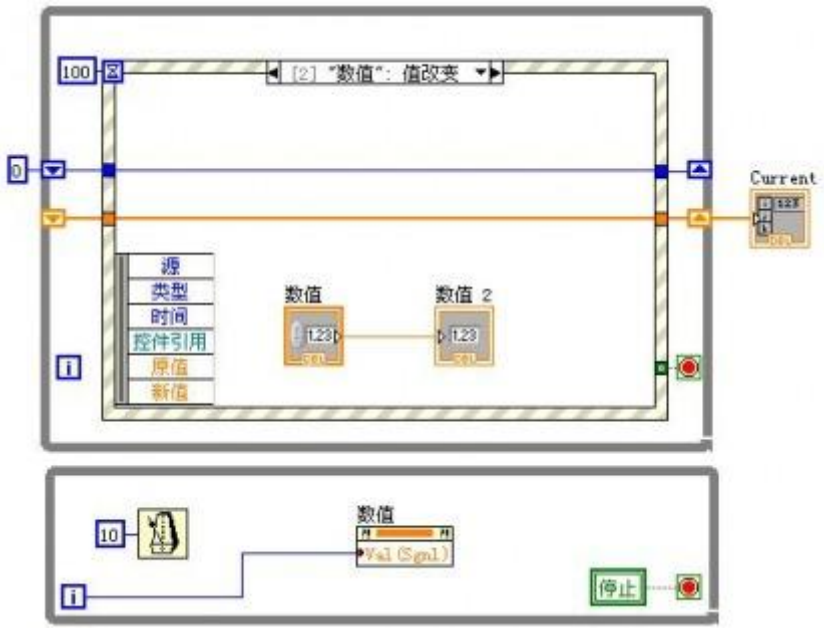
事件结构中的 TIMEOUT 进行数据采集合适吗

最近看到利用事件结构中超时 TIMEOUT 事件进行数据采集的方法,过去我也过这种方法.

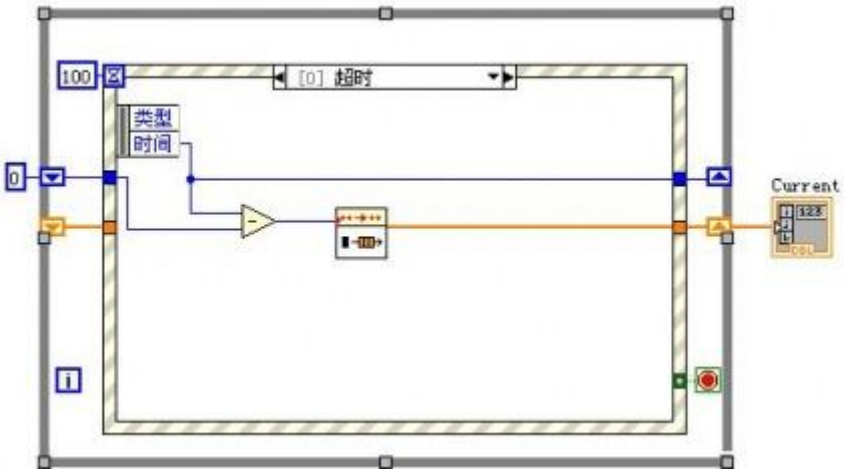
优点:不再需要单独的数据采集循环,使用 SHFIT REGISTER 就可以在其他事件中共享数据.

但是这种用法是存在一定缺陷的,假如 TIMEOUT 的设定值是 100MS,那么事件结构在 100MS 内如果没发生事件,则产生一次超时事件,但是如果 100MS 内有任何其他事件发生时,将不会响应本次的超时事件,如果在 100MS 内一直有其他事件发生,那么事件结构将永远不会产生超时事件.

看一下测试程序



由于下面的循环每隔 10MS 触发一次事件,导致根本不会产生 TIMEOUT 事件.
如果去掉下面的循环,则 TIMETOUT 事件正常产生,误差是 1MS,这也是 LV 软件定时的最高精度了

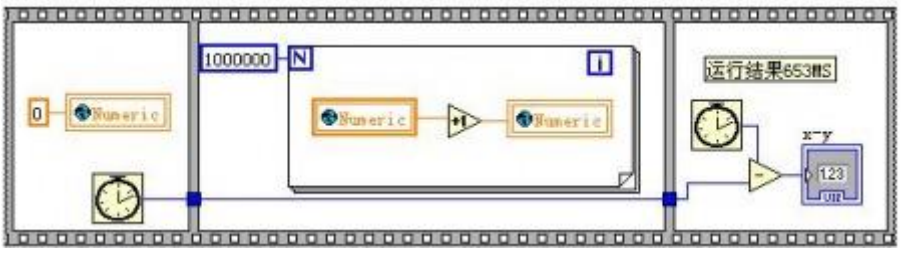


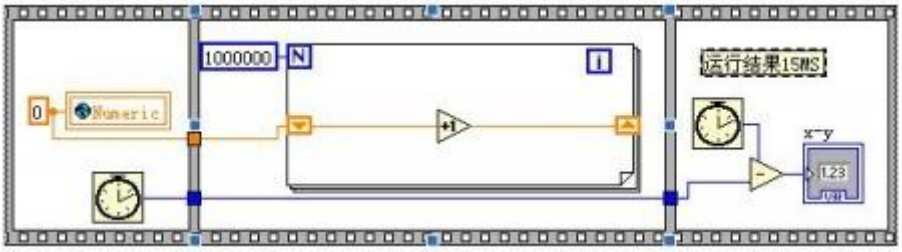
所以,如果想用 TIMEOUT 进行数据采集,一定要注意不能产生其他事件,这也是 TIMEOUT 不适合数据采集的原因.

全局变量、移位寄存器和功能型全局变量的性能比较

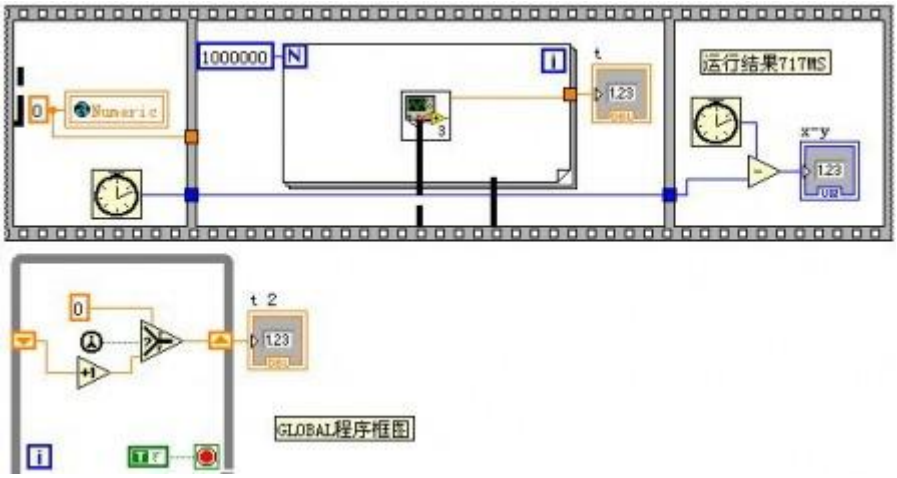
对于内置全局变量和 FUNCTION GLOBAL（LV2 GLOBAL）的性能 LV 相关书籍中的介绍各不相同，甚至是矛盾的，关于数据竞争的问题就不讨论了，FUNCTION GLOBAL 有明显的优势，今天主要看看它的运行速度问题，我在以前的文章中提到过读写 GOBAL 需要内存拷贝的问题，频繁调用内存管理器肯定要影响它的 速度，而 FUNCTION GLOBAL 虽然不存在内存复制的问题，但是它需要反复调用 SUBVI，一定程度上会影响它的速度。

首先看内置 GLOBAL 的 SHIFT REGISTER 的性能比较。

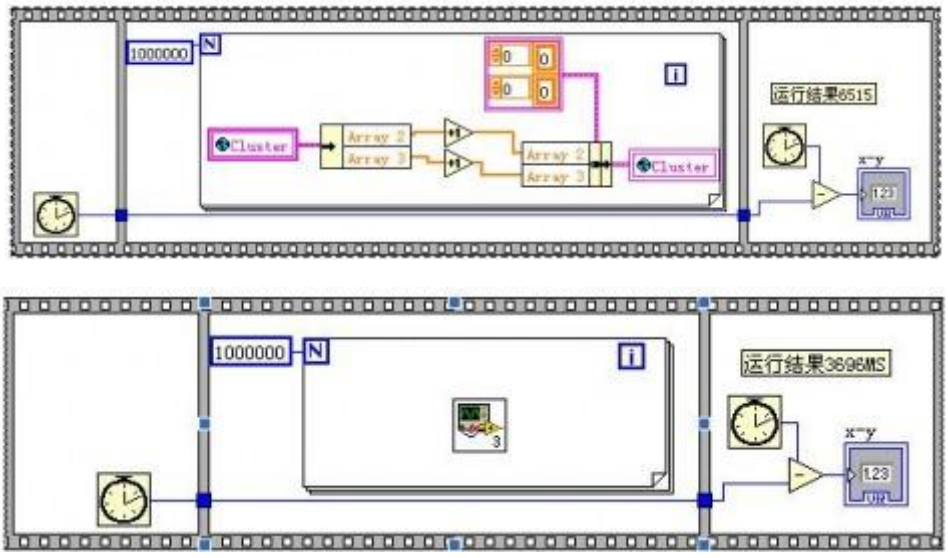


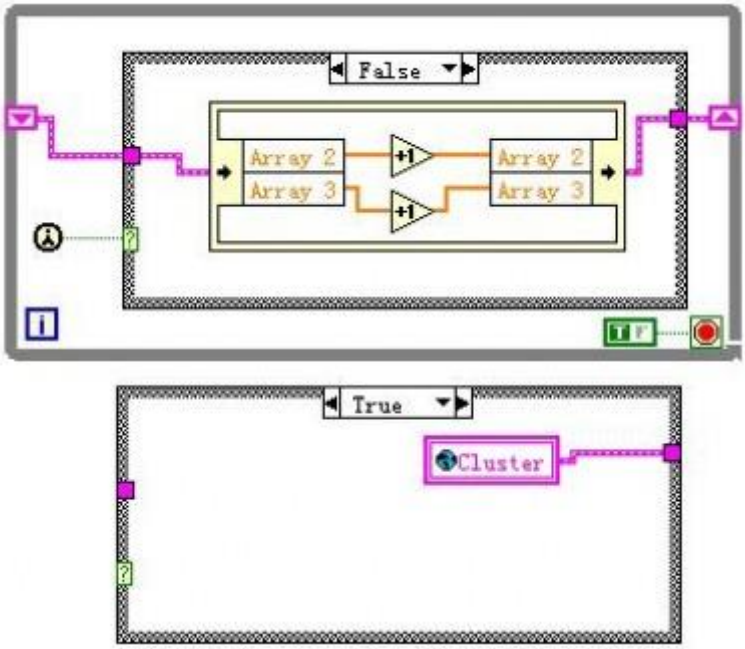


可以看出 SHIFT REGISTER 的运行速度远高于全局变量，这是可以理解的，毕竟 SHIFT REGISTER 是 LV 的核心，它的运行肯定是高效率的。



可以看出对一个简单地标量，GLOBAL 的速度还是快于 FUNCTION GLOBAL 的，不过是在一个数量级别上。





本次实验的数据类型是簇，内部包含两个元素，2 个数组长度为 100 的数组
从上面的实验结果可以得出结论，当数据结构非常复杂时，FUNCTION GLOBAL 的效率要高于内置全局变量，原因是内存复制的开销已经高于 SUBVI 调用的开销

利用 DDE 实现进程间的数据交换（一）

LABVIEW 是多线程的,在两个线程交换数据有多种方法,进程(PROCESS)和线程(THREAD)是两个不同的概念,我们启动一个执行文件实际上 就是启动一个进程,WINDOWS 的进程管理器可以观察到当前存在那些活动进程,进程间交换数据可以简单地理解成多个执行文件间交换数据.

进程间交换数据有几种方法:剪切板(CLIPBOARD),动态数据交换(DDE),内存映射文件(MAP FILE)和一般文件,当然也可以用 TCP/IP ,SHARE VARIABLE,DATASOCKET,不过这些都属于网络数据交换,用于本机进程间通讯并不合适.

过去的一篇文章中已经介绍过如何利用剪切板进行通讯,今天介绍一下动态数据交换(DDE)

DDE（Dynamic Data Exchange），即动态数据交换，是 Windows 平台上的一个完整的通信协议，它使

应用程序能彼此交换数据和发送指令。DDE 过程是两个程序的对话 过程，一方向另一方提出问题，然后等待回答。提出问题的一方即申请告知信息的应用程序，称为顾客（Client），回答的一方即提供信息的应用程序，称为 服务器（Server）。一个应用程序可以同时是顾客和服务器：当它向其他程序请求数据时，它充当的是顾客；当有其它程序需要它提供数据时，它又成了服务 器。但就某一确定的时刻而言，一个应用程序只能充当顾客或服务器。

DDE 对话的内容是通过 3 个标识进行约定的：①服务器名(Service Name)：DDE 源的每个应用程序有一个唯一的服务器名，通常为不带后缀的可执行文件；②话题(Topic)：对源程序有意义的一些数据单元即对话的议 题，许多应用程序将文档名作为 DDE 会话的话题；③项目(Item)：DDE 会话中，两个应用程序间真正传递的数据。建立 DDE 之前，客户程序必须填写服 务程序的 3 个标识名。

DDE 链接有 3 种类型：①热链接(hot link):服务器发送专门为 DDE 对话而设定项目中的数据，当这些数据发生变化时，链接将实时动作，自动更新数据；②冷链接(cold link):当数据发生变化时，客户必须明确地提出更新要求，数据才会被更新；③暖链接(warm link)：服务器在数据发生变化时，通知客户，客户根据自己的要求决定是否更新数据。

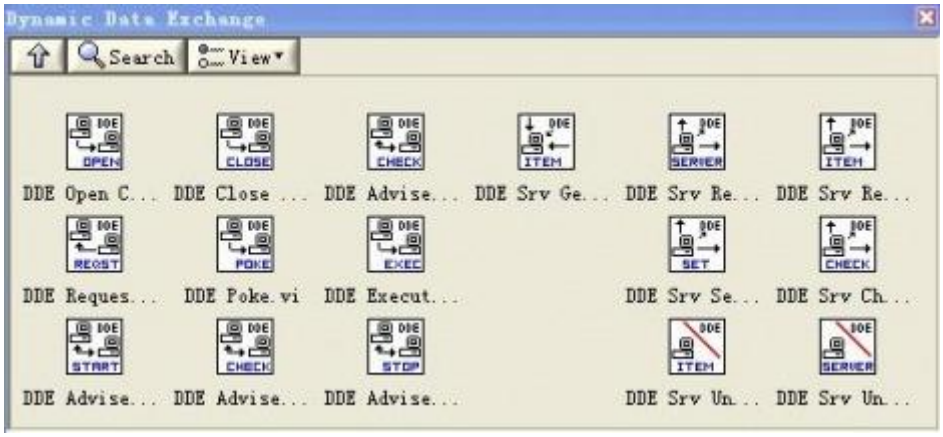
DDE 本质上是通过发送消息实现的,在 VC 和 CVI 中,可以注册事件回调函数,实现自动数据交换,但是遗憾的是 LABVIEW 并没有对 DDE 提供事件驱动方式,正如它的串口操作一样,都是通过轮询(POLLING)方式进行的,因此就涉及到两个进程 DDE 速度协调的问题。

DDE 是 WINDOWS 早期进程间通讯的重要方式，现在用的不多了，但是很多应用程序，比如 OFFICE，MATLAB 等，包括各种流行的组态软件，依然提供对 DDE 的支持，所以有必要了解一下。

7。1 以后的 LABVIEW 在模板中是找不到 DDE 库的，需要手动添加到 USER LIB 中。

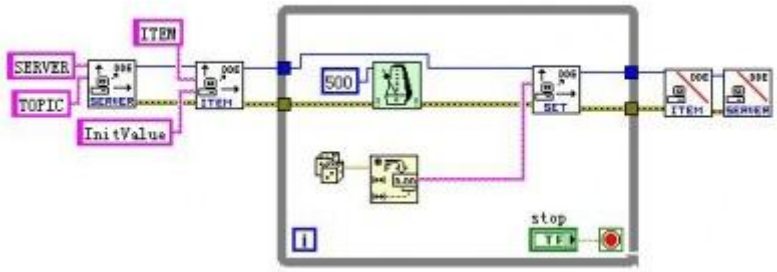
C:\Program Files\National Instruments\LabVIEW 8.5\vi.lib\Platform\dde.llb

主要分成客户机和服务器两部分,客户机和服务器 VI。

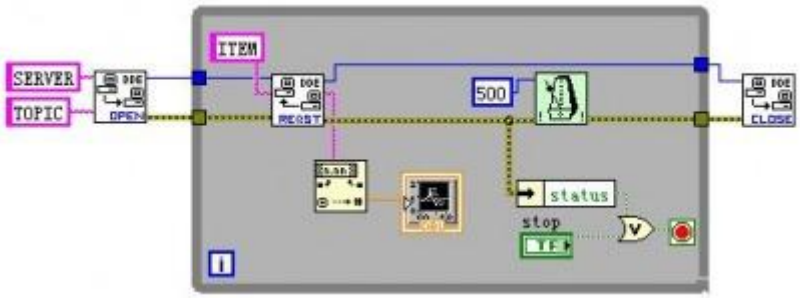


DDE 数据交换首先要启动服务器，否则客户机无法连接。

服务器操作过程是：
注册服务器---》注册 ITEM----》设定 ITEM 值-----》取消 ITEM 注册---》取消服务器注册

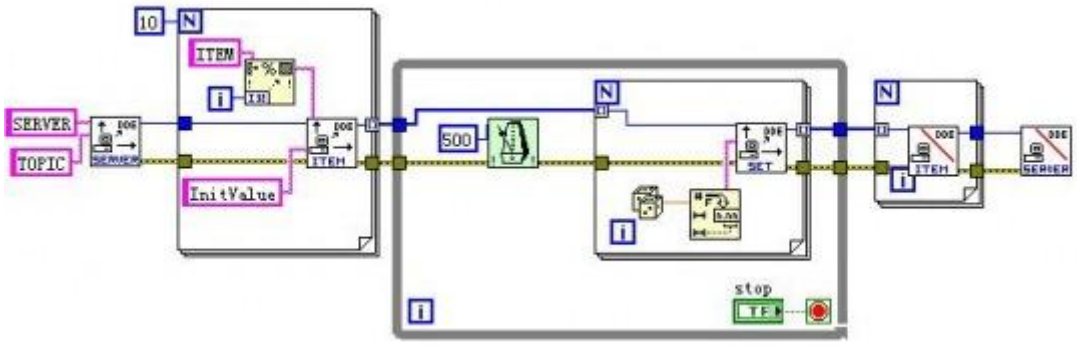


相应客户端的读数据框图

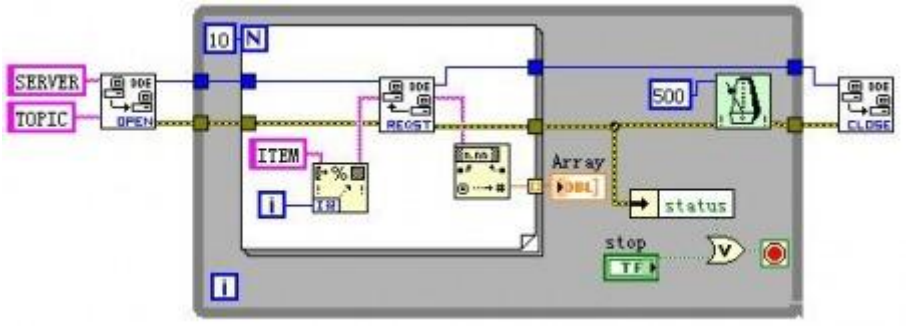


注意到服务器的循环每 500MS 更新一次，客户端每 500MS 更新一次，服务器和客户端基本保持同步,如果客户端速度高于服务器,将导致对服务器的同一数 据读多次，同理，如果服务端运行速度快，客户端运行速度慢，将导致数据丢失，这正是没有事件响应的缺点，很难保证发送和接受的同步。因此，上面的程序仅适 用于对数据交换要求不高的情况，比如监控等。

DDE 是一个层次结构，SERVER--》TOPIC---》ITEM
一个 SEVER 可以包括多个 TOPIC，（类似与组），每个 TOPIC 又可以包括多个 ITEM（项目），我们可以通过循环注册多个 TOPIC 和多个 ITEM，实现批量数据交换。



上图中，通过循环为 TOPIC1 同时注册了 10 个 ITEM，分别是 ITEM0----》ITEM9，在主循环中分别向 ITEM0--》ITEM9 写入数据。



同理，我们可以对我们有通讯的数据详细分类成多个 SERVER 和多个 TOPIC，进行大量的数据交换。

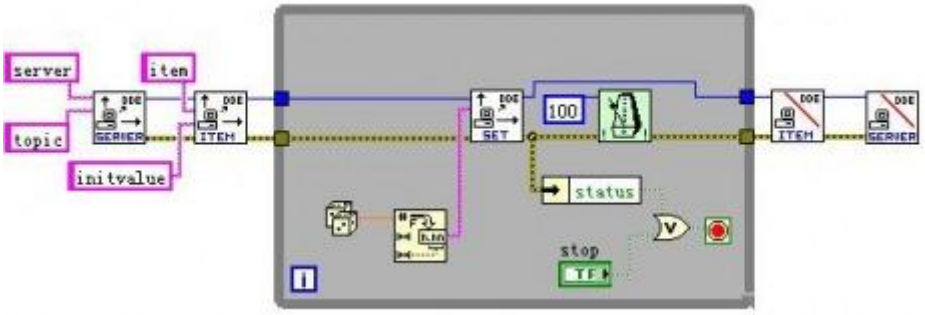
后续文章中将进一步介绍 DDE 的高级同步技术。

利用 DDE 实现进程间的数据交换（二）

我在"利用 DDE 实现进程间的数据交换之一"中,谈到了服务器端发送数据和客户端接收数据的方法.

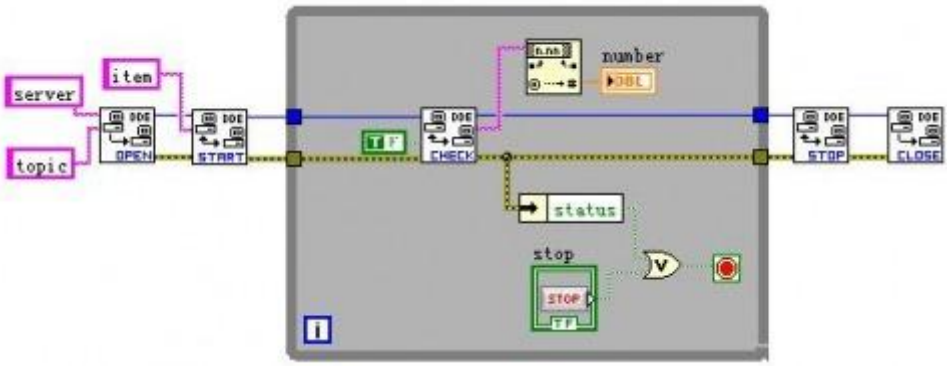
客户端采用 REQUEST 方式是无法实现服务器发送和客户端接收同步的,如果想要实现,可以采用 ADVISE 方式,这种方式下,客户端类似于中断方式,服务器发送端 ITEM 没有数据变化时,接收端一直处于等待状态,一旦服务器端发生数据变化,立即执行数据接收.

服务器端程序框图:(发送数据,100MS 更新一次)



客户端接收数据未采用任何 DELAY,同样保持 100MS 的接收速度.

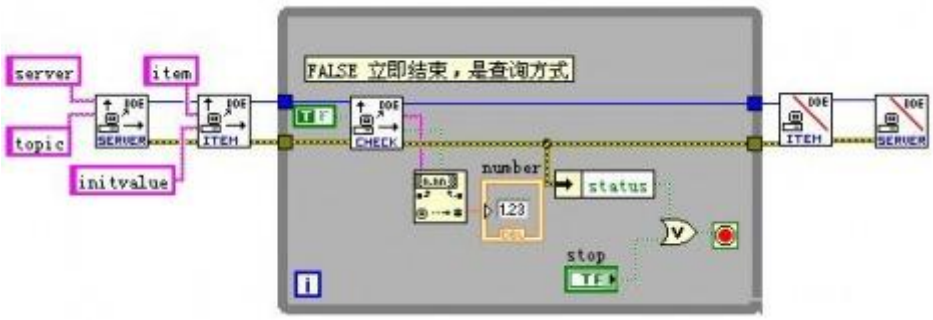
打开对话-->START ADVISE ITEM--->CHECK ADVISE--->STOP ADVISE-->关闭对话



这样就实现了发送和接收的数据交换同步.

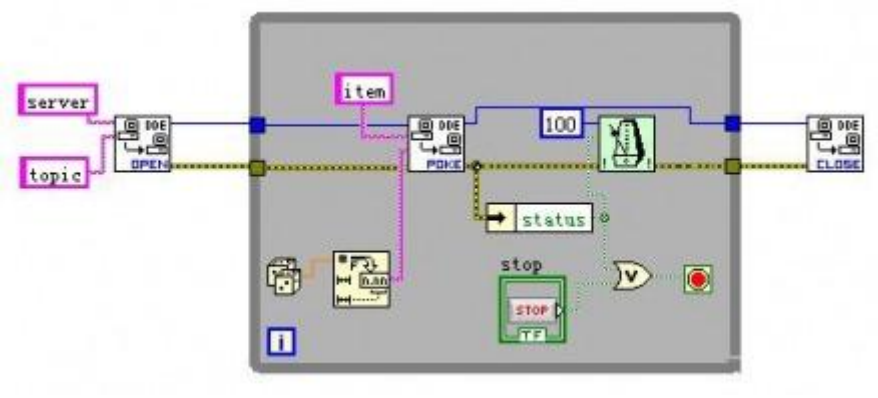
我们知道服务器和客户端的数据交换是相互的,如何实现客户端发送数据,服务器接收数据那,这需要客户端使用 POKE VI, 服务器端使用 CHECK ITEM VI.

服务器端程序框图



如果设置成 FALSE,则立即结束本次循环,实际是查询方式.

客户端程序框图:



当客户端没运行 POKE 时,服务器处于等待状态,这样就实现了双方的同步.

DDE 本身还支持握手方式通讯,不过非常复杂,很少使用.

另外,客户端还可以向服务器发送命令字符串,要求服务器执行命令.不过 LABVIEW 的 DDE 不支持命令,就不举例了.比如 EXCEL 可以作为服务器,客户端可以发送"OPEN" "SAVE"要求 EXCEL 打开和存储文件.

OPC 系列之基本概念

OPC 自从 1994 年制定标准,迄今已经 10 多年了,越来越多设备制造商和仪器制造商都开始支持 OPC 了,下面以 NI 的 OPC 服务器为例,介绍以下 OPC 的基本概念和使用方法。

NI 的 OPC SERVER 是 WINDOWS 32 位应用程序，它为 PC 用户提供了访问外部设备数据和信息的通道或者说手段，通过 OPC，设备变成了 PC 网络的一个成员。

OPC 的概念

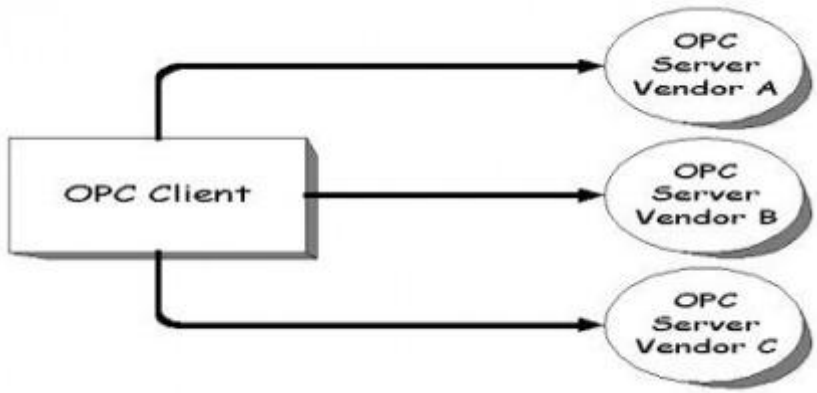
OPC 是 OLE FOR PROCES CONTROL 的英文缩写，直译是 OLE 用于过程控制，OLE 是 WINDOWS 的一个基本概念，是对象嵌入链接的缩写，过程控制实际是工业自动化控制的概念，目前，PLC 技术和 NC 技术、CAD CAM 技术以及工业控制总线已经成了工业自动化控制的核心，因此可以说 OPC 本身就是 PC 在

工业自动化控制领域的扩展。

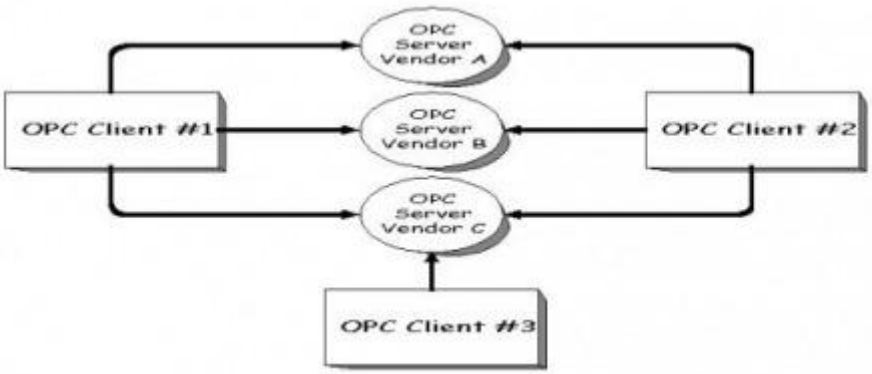
1994 年，世界一些知名的工业设备制造商成立了一个专业组织，宗旨是为各种各样的工业设备指定一个统一的软件数据通讯（不是物理层的通讯）标准，这就是后来为大家熟知的 OPC。

OPC 的一个主要目的是避免 PC 客户为工业设备开发通讯驱动程序，为一个特定设备开发驱动程序是极其复杂和耗时的，因为设备千差万别，硬件接口也是多种多样的， 对于一般的软件开发人员是很那作到的，一个更好的办法是制定一个统一的数据访问标准，而硬件驱动的部分有硬件厂商或者专门 OPC 开发人员负责，这样，PC 用户就可以依据这个标准，和外部工业设备无逢连接，这个数据访问标准就是 OPC。

通过 OPC，一个 PC 客户（OPC 客户）可以访问多个外部设备



多个 OPC 客户（可能是网络上的）可以访问多个外部设备



对于 OPC CLIENT，OPC SERVER 提供了几个高层对象供 CLIENT 访问，分别是 SERVER，GROUP 和 ITMES，这非常类似 DDE 通讯。

SERVER 对象提供的是服务的有关信息，同时有是 GROUP 对象的容器。

GROUP 对象提供的性能相似的分类信息，OPC CLIENT 可以配置是否允许 GROUP 和 GROUP 组数据的更新频率，同时也提供了如果数据访问失败的错误信息。GROUP 同时也是 ITEM 对象的容器。

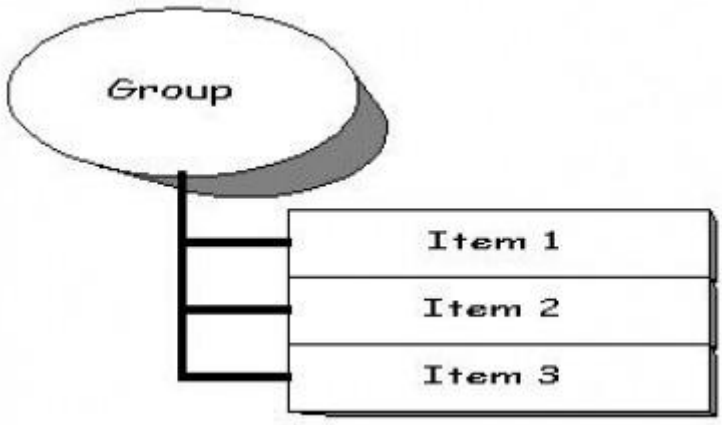
ITEM 对象是每个特定的数据项目，比如可能是设备一个特定的寄存器。

这是一个典型的分层结构，如果我们要访问一个 ITEM，途径必须是

SERVER----》GRUOP----》ITEM
OPC CLIENT 是没有办法直接访问具体的 ITEM，这样有效地实现了数据的封装。

有两种形式的 OPC GROUP，公有或者私有（也称做局部），公有可以被所有的 OPC CLIENT 访问，私有只能被特定的 OPC CLIENT 访问。

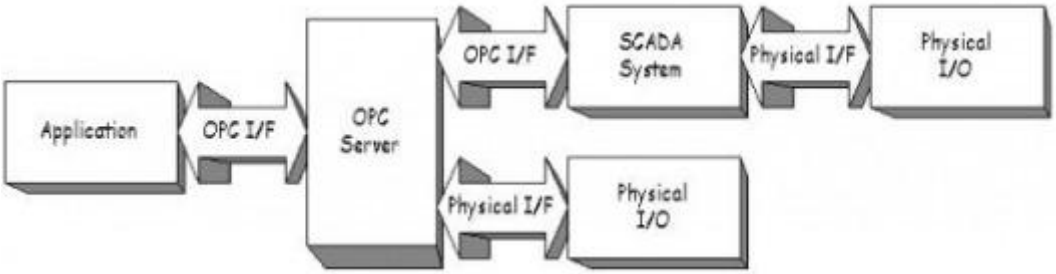
每一个 GROUP 对象都包括多个 ITEM 对象。



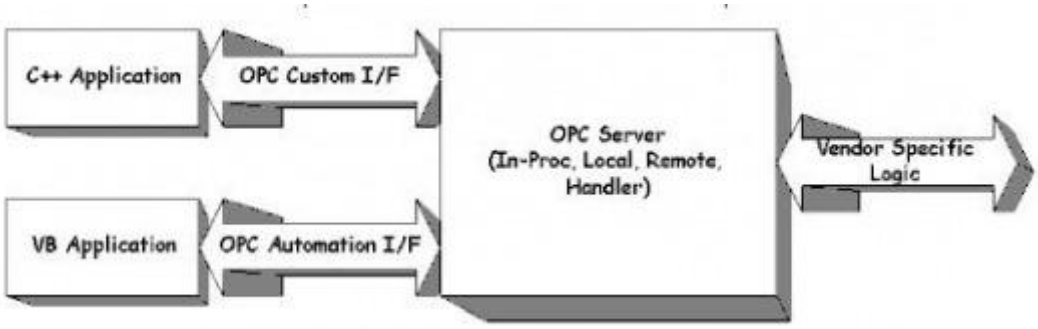
对于每一个具体的 ITEM，又由三部分组成，VALUE，QUALITY 和 TIMESTAMP

VALUE 是变体数据，表示 ITEM 当前值，QUALITY 与总线形式有关，TIMESTAMP 是时间戳。

虽然 OPC 主要用于网络设备的数据访问，但是在单机控制设备时也经常使用。



OPC 服务器的内核是 COM 的 DCOM，对于 PC CLIENT 提供了两种接口，一种是针对 C++客户，可以直接访问，速度较快，另外一种是通过 AUTOMATION 自动化服务器，这是 VB 和脚本语言用户使用的接口，因为 OPC 需要对其进行解释，相对速度较慢。



LabVIEW 与回调函数

回调函数是 WINDOWS 编程(API 编程)的核心内容之一,在许多高级编程语言,如 VB,VC(MFC)中已经封装了回调函数,取而代之的是事件响应函数,但是,追溯其本质,实际就是回调函数.

所谓 WINDOWS 回调函数,就是按照 WINDOWS 的规范,编写的(CALLBACK)函数,当 WINDOWS 检测到事件发生时,自动调用的函数,WINDOWS 是通过函数指针调用的,因此,回调函数的内容是由用户决定的,而何时调用是由操作系统决定的.

我们看一下 CVI 中的一般回调函数的定义

int callback aaaa(int panel,int control,int event1,int event2,callbackdata *data);

回调函数的参数是有操作系统提供的,比如上面的回调函数,

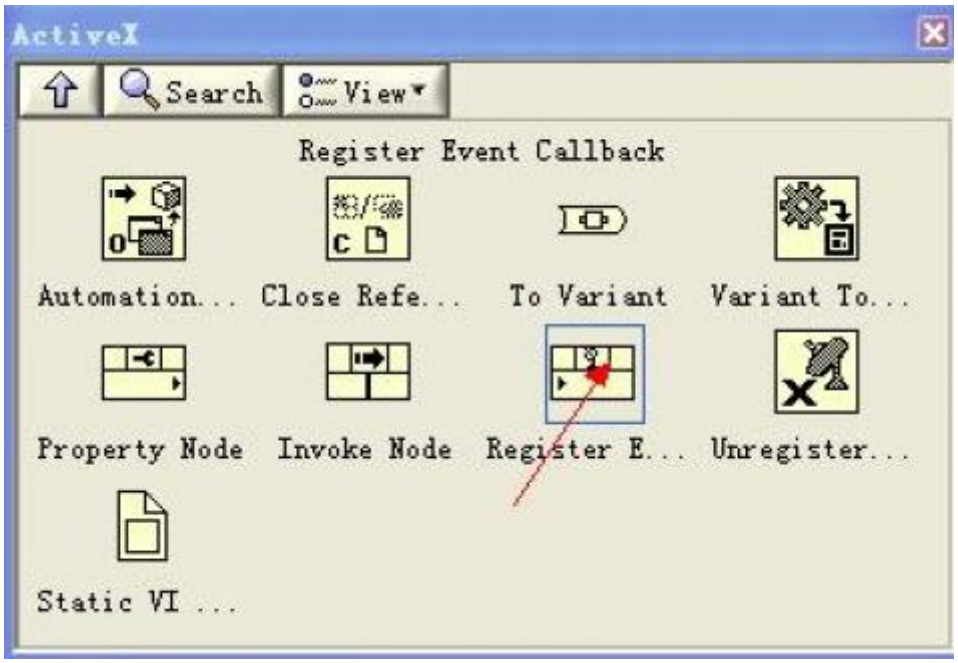
panel---表示的哪个面板(窗口)发生的事件

control---表示的面板上哪个控件发生的事件

event1 event2 表示事件的类型和相应数据,比如鼠标坐标等

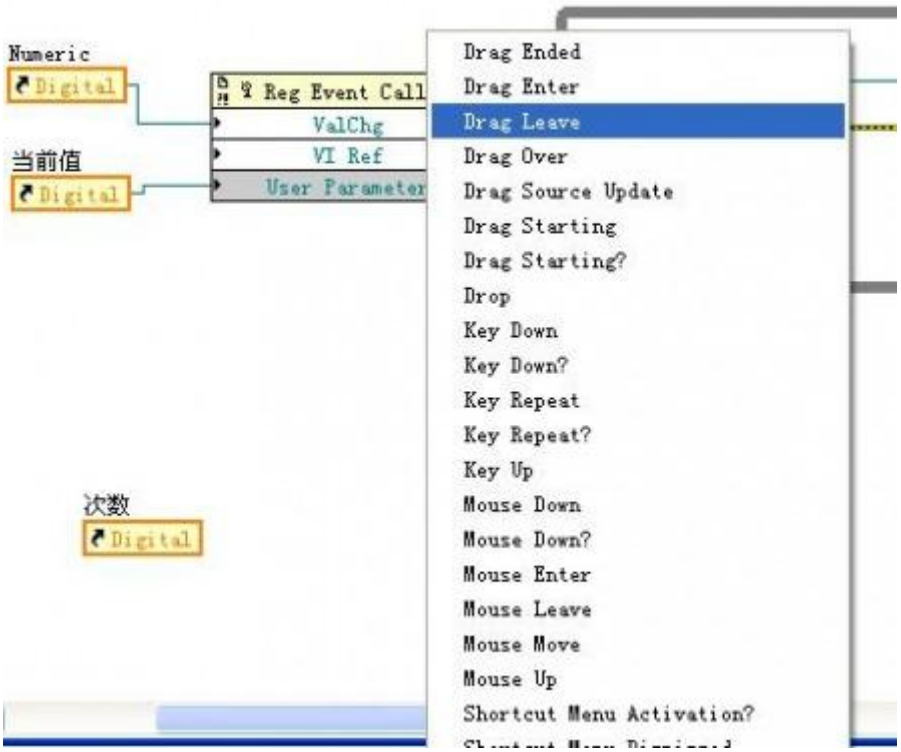
回调函数是一般高级编程语言的基本功能,但是,在 LABVIEW8.X 之前是不支持的,这极大限制了 LABVIEW 功能的扩展,因为 ACTIVEX,.NET 都需要回调函数.

8.X 中,增加了回调函数的功能,主要用于 ACTIVE,.NET 和 LABVIEW 自身控件,LABVIEW 例子程序中提供了几个例子,是有关 ACTIVEX 和.NET 调用的,下面,我们通过 LABVIEW 自身控件说明一下回调函数的使用方法.



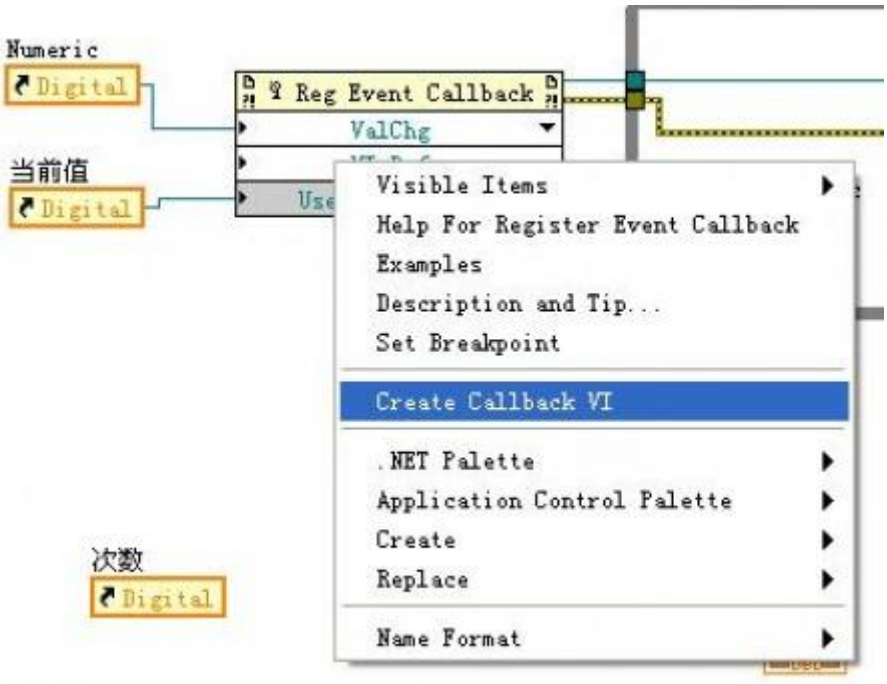
在.NET 摸板中也提供了这个节点,从分类上就可以看出,注册回调函数主要是用于 ACTIVEX 和.NET 的.

下面我们做一个简单的回调函数的程序,有两个功能,返回当前值的变化和记录控件被点击的次数



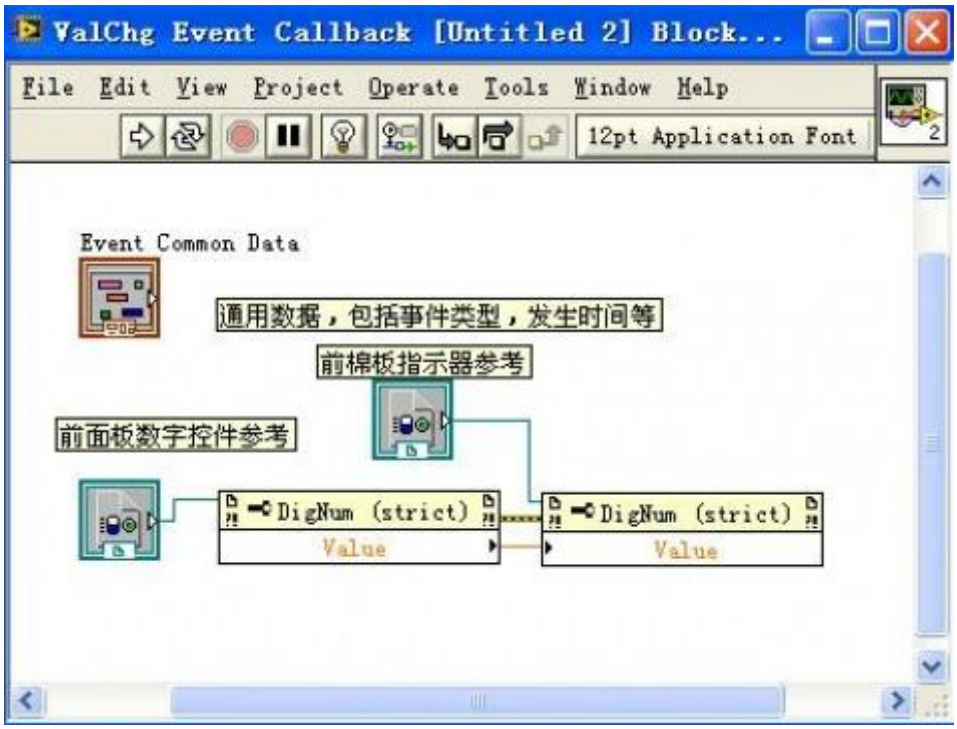
注册回调函数需要三个参数:控件参考,用户参数和自动生成的回调函数,有了控件参考,我们就可以选择事件的类型,用户参数主要是用于返回结果,因为回调函数是由操作系统调用的,没有办法通过数据流返回处理结果.

添加了这两个参数后,就可以自动生成回调函数了

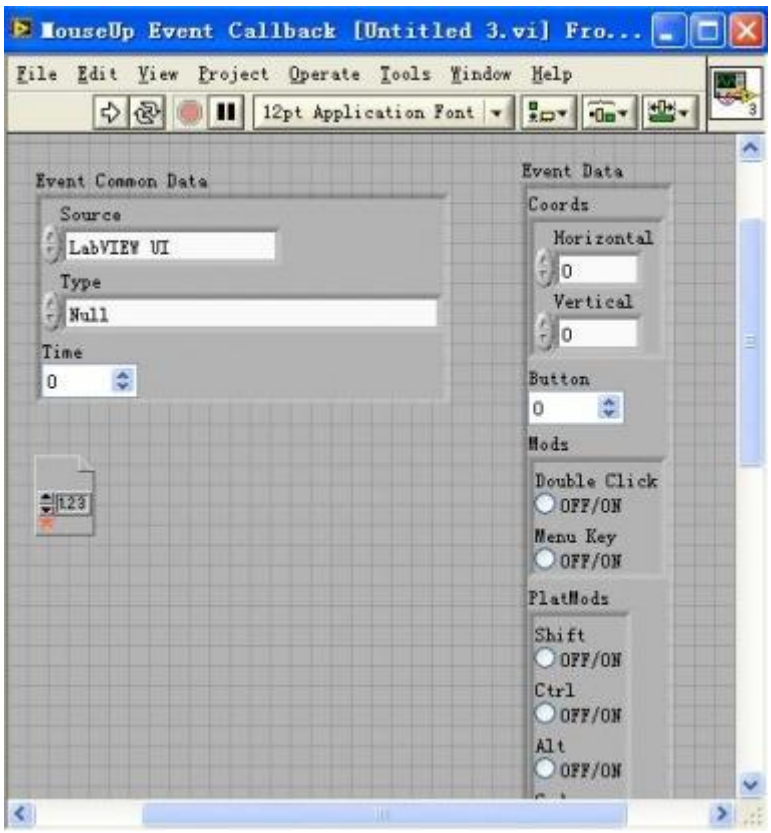


回调函数如下图所示

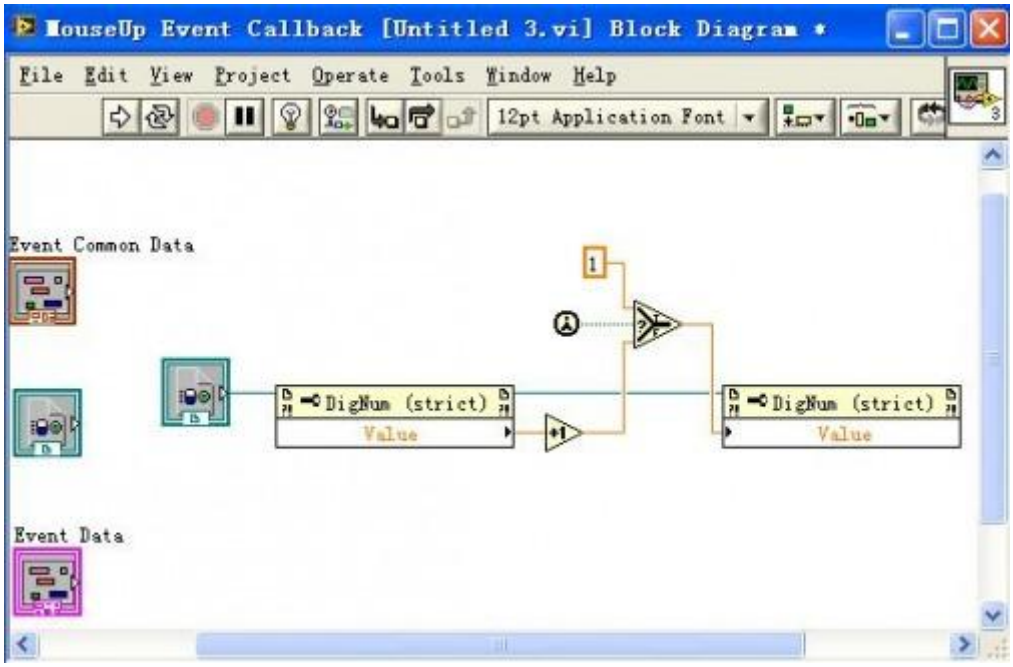
简单编程,CONTROL 的值传递给 INDICATOR



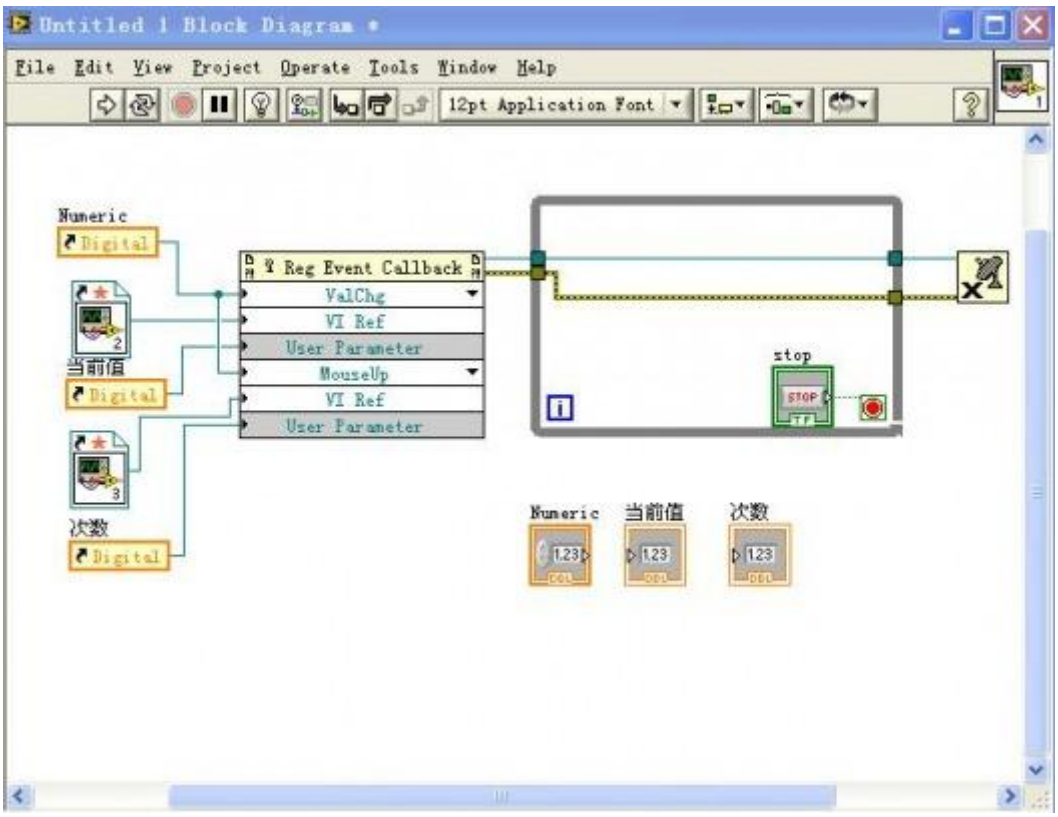
这样值变化的回调函数完成了,下面我们通过鼠标 UP 事件来记录被点击的次数



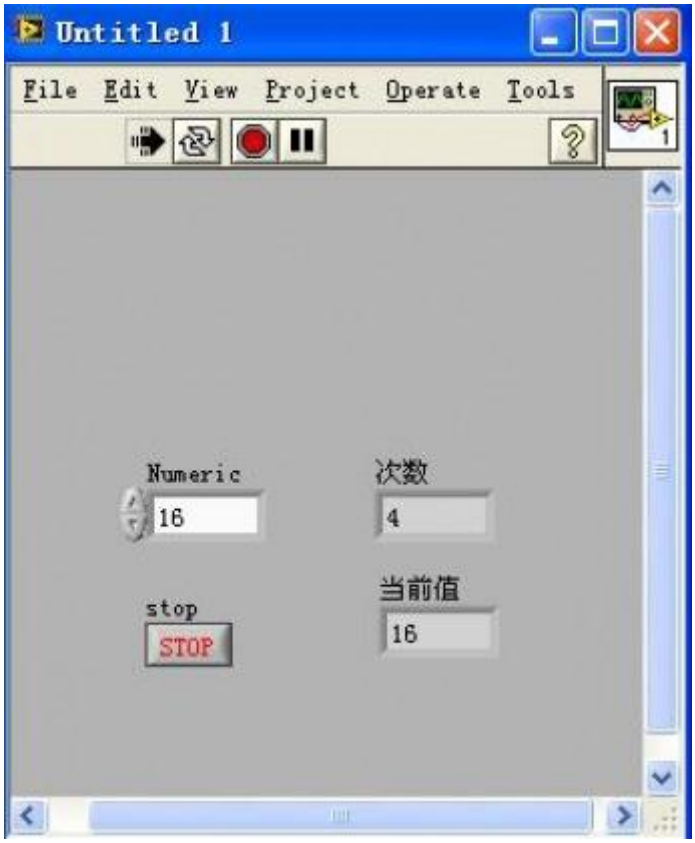
可以看出,这次,我们注册的是鼠标抬起事件,因此,系统传递了鼠标的坐标等信息



最后看一下主程序框图



在主程序循环中,没有对控件和指示器做任何操作,所有操作响应都是通过回调函数完成的



数据库连接的几个基本概念

NI 公司对通用数据库提供了 LABVIEW 的组件,通过 ADO 提供了对数据库的完全支持.
ADO 是通过连接字符串打开数据库的,因此有必要了解一下数据库连接的几个基本概念.

1.odbc,oledb,ado,adox 的关系：

odbc: 曾经的数据库通信标准

oledb: 在一切对象化的趋势下,ms 打算用它取代 odbc.

oledb 分两种：直接的 oledb 和面向 odbc 的 oledb,后者架构在 odbc 上，这样没有自己的 oledb 提供者的数据库也可以使用 oledb 的特点了。

ado: 其实只是一个应用程序层次的界面，它用 oledb 来与数据库通信。

adox: 对 ado 的安全性，维护性（如：创建一个数据库）进行了扩展。

2.用 odbc 连接数据库：

odbc 中提供三种 dsn,它们的区别很简单：用户 dsn 只能用于本用户。系统 dsn 和文件 dsn 的区别只在于连接信息的存放位置不同：系统 dsn 存放在 odbc 储存区里，而文件 dsn 则放在一个文本文件中。

它们的创建方法就不说了。

在 asp 中使用它们时，写法如下：

A.sql server:

用系统 dsn: connstr="DSN=dsnname; UID=xx; PWD=xxx;DATABASE=dbname"

用文件 dsn: connstr="FILEDSN=xx; UID=xx; PWD=xxx;DATABASE=dbname"

还可以用连接字符串（从而不用再建立 dsn):

connstr="DRIVER={SQL SERVER};SERVER=servername;UID=xx;PWD=xxx"

B.access:

用系统 dsn: connstr="DSN=dsnname"

(或者为：connstr="DSN=dsnname;UID=xx;PWD=xxx")

用文件 dsn: connstr="FILEDSN=xx"

还可以用连接字符串（从而不用再建立 dsn):

connstr="DRIVER={Microsoft Access Driver};DBQ=d:\abc\abc.mdb"

3.用 oledb 连接数据库：

A.sql server:

connstr="PROVIDER=SQLOLEDB;
DATA SOURCE=servername;UID=xx;PWD=xxx;DATABASE=dbname"

B.access:

connstr="PROVICER=MICROSOFT.JET.OLEDB.4.0;
DATA SOURCE=c:\abc\abc.mdb"

4.使用 UDL 文件：

UDL 文件是用来存放数据库连接信息的一个文本文件，有点象文件 DSN，不过 UDL 是针对 OLEDB（直接的和面向 ODBC 的）的。

UDL 的创建方法：

右击桌面或资源管理器–》新建–》microsoft 数据连接

其中的设置工作应该比较清楚了。

UDL 的用法：

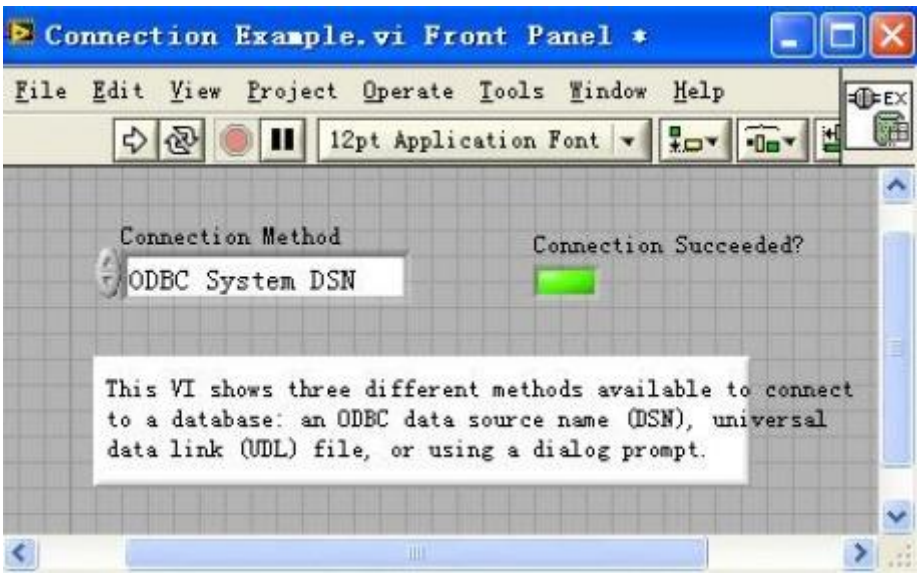
connstr="file name=e:\abc\abc.udl"

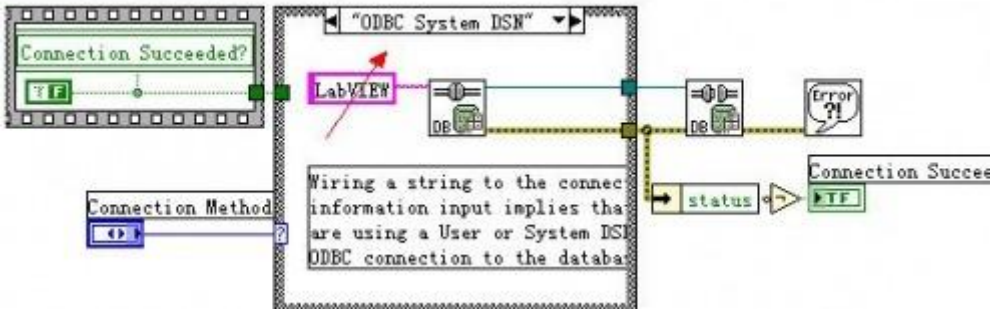
通过上面的介绍可以看出 ADO 操作数据库的层次结构

ADO----->OLEDB----->数据库

ADO----->OLEDB----->ODBC----->数据库

对于系统 DSN,可以直接给出 ODBC 名称,ADO COM 自动到 ODBC 系统区查找,看 LV 本身的连接例子

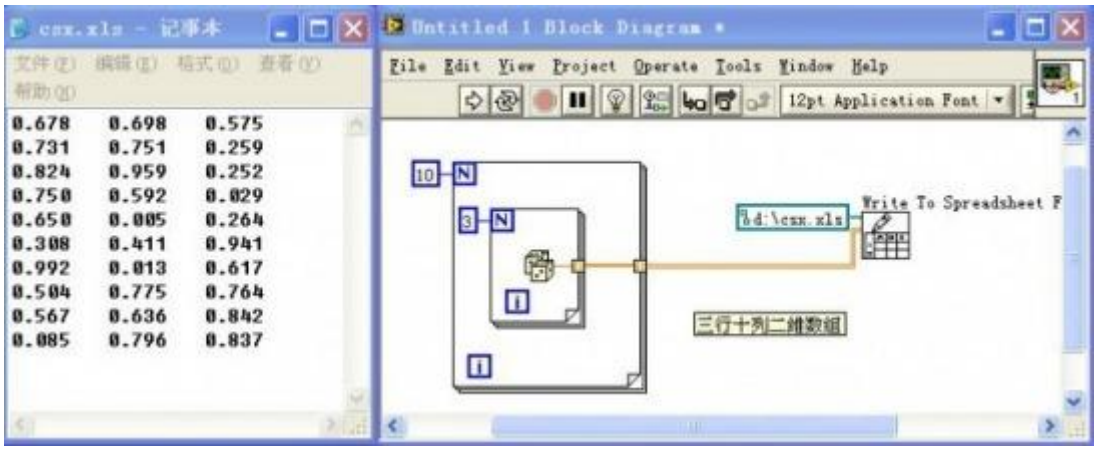




文件系列之写电子表格文件

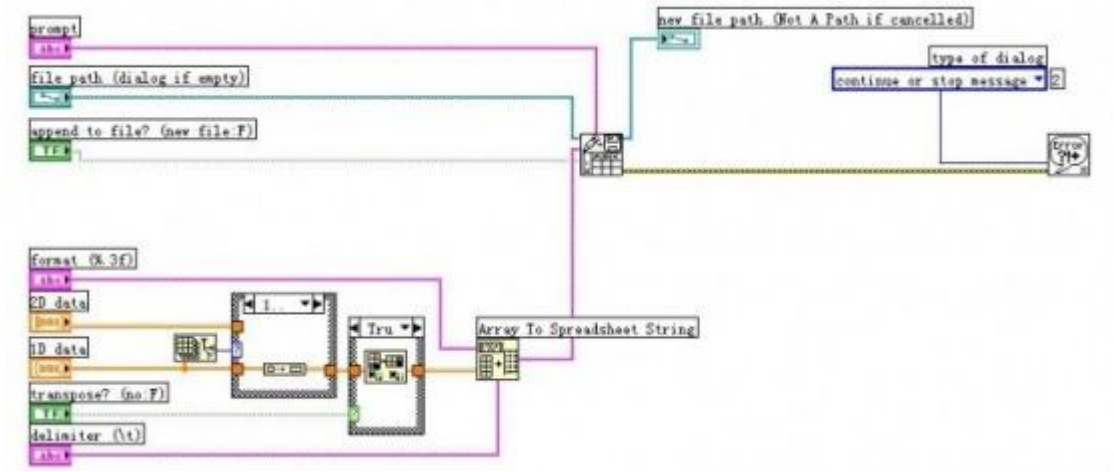
LabVIEW 文件操作种类非常丰富,最简单易用的是 WRITE TO SPREAD SHEET FILE(通常翻译成写电子表格文件),这种所谓的电子表格文件其实是文本文件,并非真正的电子表格文件,真正的电子表格文件是有格式的,一般文本编辑 器,比如 NOTEPAD 是打不开的(显示乱码),只能用 EXCEL 打开,LABVIEW 操作这种有格式的电子表格文件只能通过 AUTOMATION,自动 化服务器实现,或者利用 NI 公司的 OFFICE TOOLKIT(实际也是利用 AUTOMATION,不过是重新封装了一下,功能非常强大).

WRITE TO SPREAD SHEET FILE 是以 TAB 为分隔符号(默认)的纯文本文件,通过一个简单的例子来看一下它个格式.

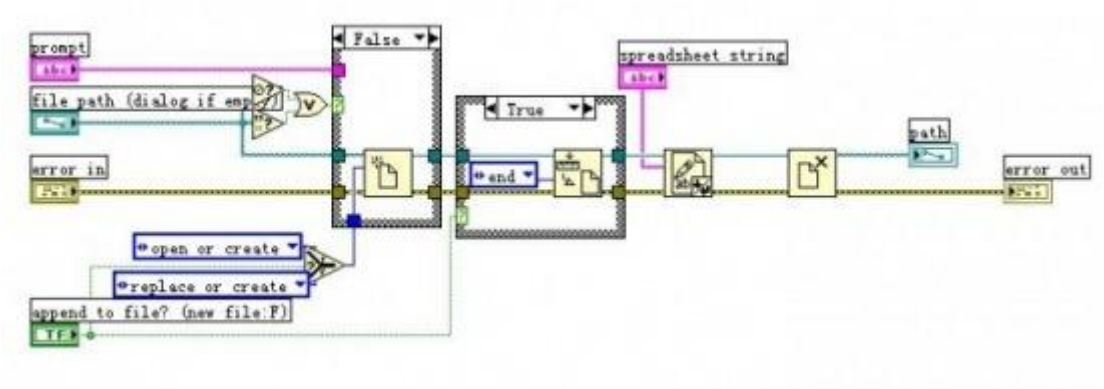


WRITE TO SPREAD SHEET FILE VI 是个多态 VI,输入可以是一维或者二维,类型可以是 INTEGER 、DOUBLE 数组或者字符串数组，从实质来说,它的输入是字符串数组,DOUBLE 或者 INTEGER 数组不过是它自动转换成字符串数组然后写入文本文件的.

有不少初学者喜欢用这个 VI,不过它不是基于磁盘流的,每次写入都包括了打开,写入,关闭三个过程,因此速度很慢,不适合于连续写入,这个 VI 源代码是公开的,我们跟踪一下就清楚了.



可以看出,无论那种数据类型,其实都是通过 ARRAY TO SPREAD SHEET 写入一个字符串,然后写入文本文件,我们继续跟踪一下写入文件的过程.



这里就非常清楚了,每次写入都包括打开,写入和关闭的过程,因此它非常适合于一次性写入,而不适合于连续写入的操作.

通过上面的分析,实际上也间接地说明了如何连续写入文本文件的问题,只要把文件打开和关闭的过程放在循环外面就可以实现连续写入.

打开文件--->循环(文件指针指向末尾--->写入数据)---->关闭文件.

值得说明的是WRITE TO SPREAD SHEET FILE VI,分割符号是可以定义的,因此我们可以选择自己的分隔符号,比如逗号等,这个非常实用,尤其是读取其它编程语言写的文本文件,它们往往有自己特定的文件分割符号.

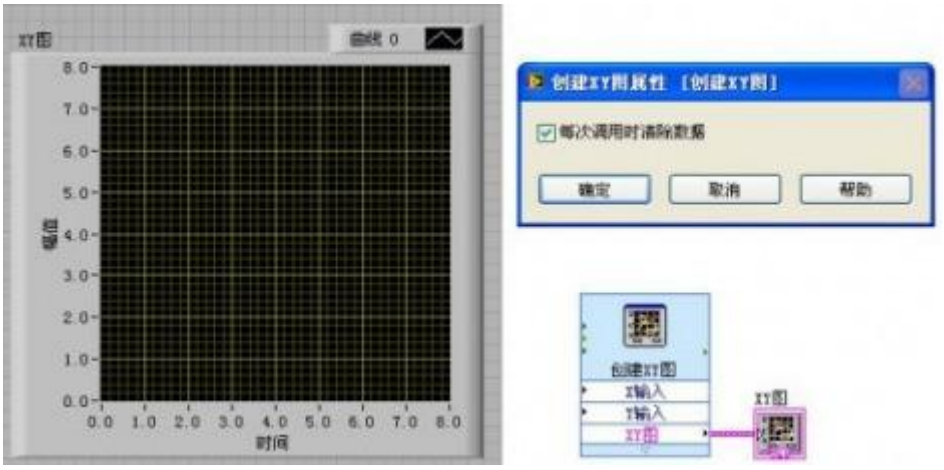
正确理解 Express XY Graph

XY 图可以说 LABVIEW 数据图形显示控件最为灵活的控件，可以替代波形图和波形图表的绝大部分功能，其输入参数形式有许多变化，在下面的两篇文章中， 具体讨论了 XY GRAPH 的多种用法。

其中一篇讨论的是利用数据缓冲区技术，如何连续显示数据的问题。快速 VI（EXPRESS VI）是 LABVIEW 提供的一套可以采用对话框快速配置的 VI,非常有利于初学者使用。在快速 VI 中,提供了快速 XY GRAPH，该 VI 可以实现连续的数据显示，但是与我提及的数据缓冲区有很大区别。

我提及的数据缓冲区连续显示数据类似于 LV 的波形图表，本身保持一个设置为固定长度的数据缓冲区，当超过所设长度时，新的数据进入缓冲区时，原有的数据被 丢弃。这样 XY 图始终显示的最新数据，而且长度不变，控件显示的是示波器的效果。

快速 XY 图则不同，它有两种不同的运行方式。



每次调用时是否清除数据决定了 EXPRESS XY GRAPH 的工作方式。

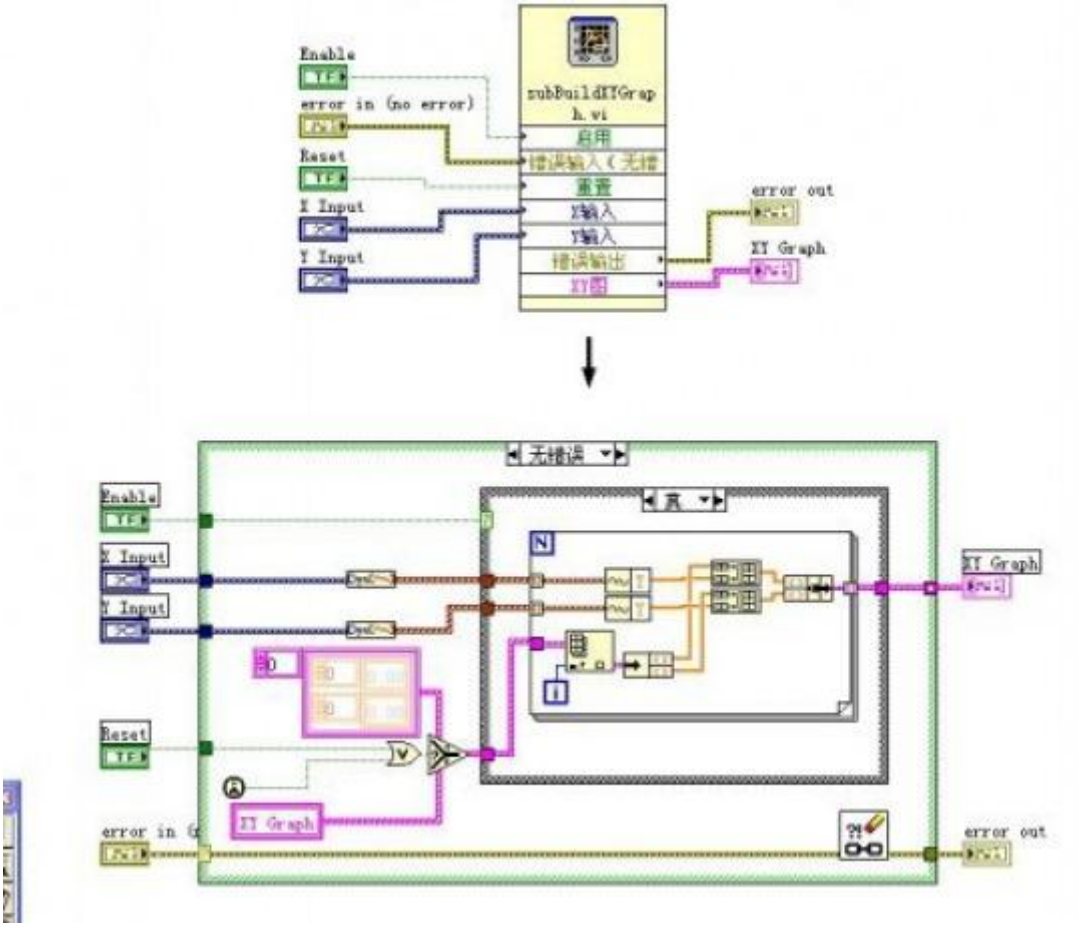
1、每次调用时清除数据，则 EXPRESS XY GRAPH 与一般的 XY GRAPH 没有明显的区别，输入参数采用了两个一维数组，这是 XY GRAPH 常见的工作方式。

2、每次调用不清除数据，则 EXPRESS GRAPH 内部记录不断累积的数据，也就是说它内部保持两个不固定长度的一维数组，用来保存 X 数据和 Y 数据。

在每次调用不清除数据时要特别注意，与数据缓冲方式不同，EXPRESS XY GRAPH 不是以移动的方式显示数据，而是不断增加数据，所用内存是不断增加的。

为了正确理解快速 XY GRAPH,我们分析一下，首先把快速 XY VI 转换成一般的 VI，分析一下它的工作原理，通过快捷菜单，选择打开前面板。

下面跟踪它的程序框图。



从程序框图可以看到，在连续显示增加数据的情况下，它利用了 XY 图的局部变量，取出原来的数据，利用 BUILD ARRAY 函数不断增加 XY 图中的数据，这样在长时间运行后，很容易导致内存滥用，而不释放的情况，同时由于使用了局部变量，内存的使用是加倍的。这与我 们所说的数据缓冲区是完全不同的。

另外，由于快速 VI 采用了动态数据，因此不断地在进行数据类型转换，这也是快速 VI 效率不高的原因之一。

从程序框图上也可以看出，如果使用连续显示的方式，我们必须监测累积数据的大小，在达到一定程度时，通过 RESET，清除数据，释放所用内存。

需要强调的是快速 VI 是在 LV7.0 后推出的，主要是简化编程难度，但是运行效率显然是不高的，同时也牺牲了灵活性，这也是熟悉 LV 的编程者一般不愿意使 用快速 VI 的重要原因。

LV2009 新增功能之数据值引用

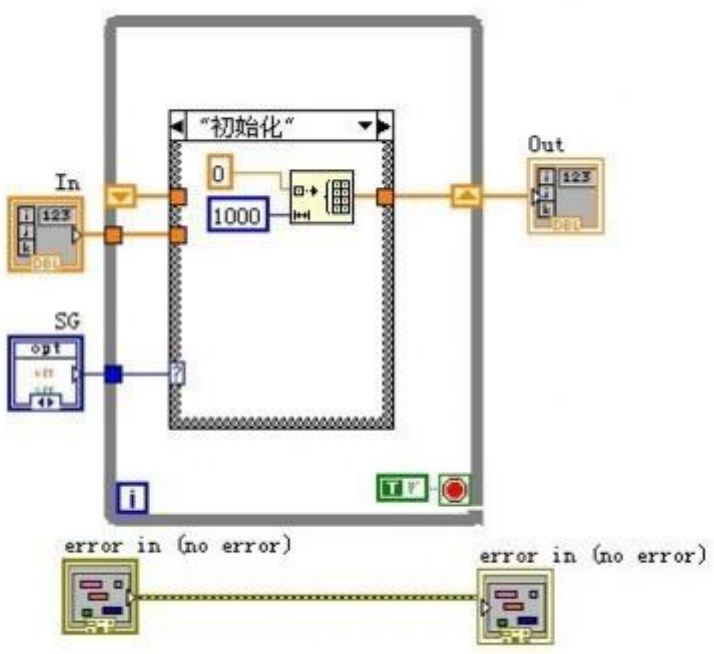
LV 是数据流驱动的，这意味着通常 LV 是采用值传递的。我们知道 C 语言传递数据有两种方式，值传递和传递引用（指针）。如果 C 函数参数采用值传递方式，会构造一个新的临时变量，函数内部对变量的修改不会影响到传入的变量，相当于自动创建了一个新的变量并完全复制传入变量的值。引用传递则完全不同，它传入的是外部变量的地址，因此在函数内部修改该变量，相当于直接修改外部变量。

LV 数据流驱动采用值传递方式，这意味着但数据连线多分枝时，数据连线可以粗略理解成 C 的变量，多分枝时则可能要创建多个同样值的临时变量。当然由于 LV 采用了缓冲重用技术，多分枝连线时可能不需要对所有的分支创建新的“临时变量”。

如果我们传递一个大型的数据结构，比如一个大的数组，值传递的运行效率远低于引用传递。因为引用传递运行中只需要传递一个指针，几个字节和一个大型数组， 显然传递指针要高效的多。另外一个重要问题是数据共享的问题，当多个地方需要处理相同的共享数据时，采用值传递显然是不合适的。

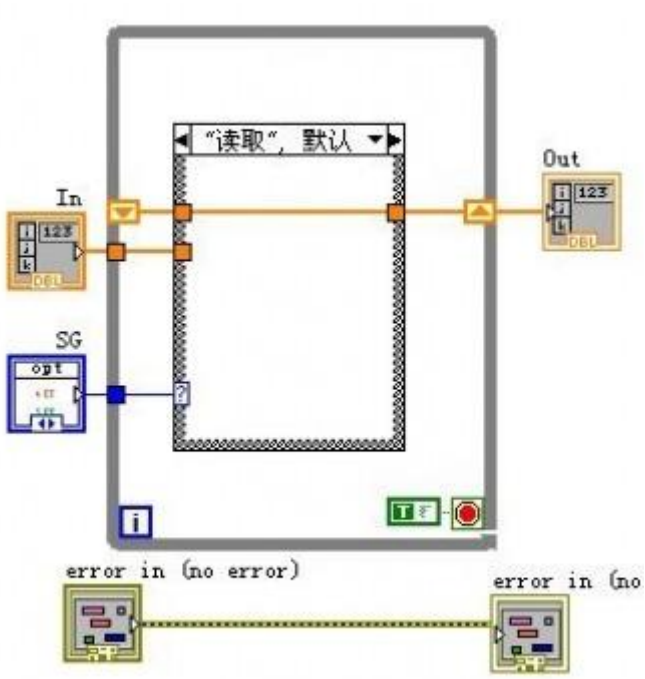
LV2009 新增了数据值引用函数，有效地解决数据引用传递问题。

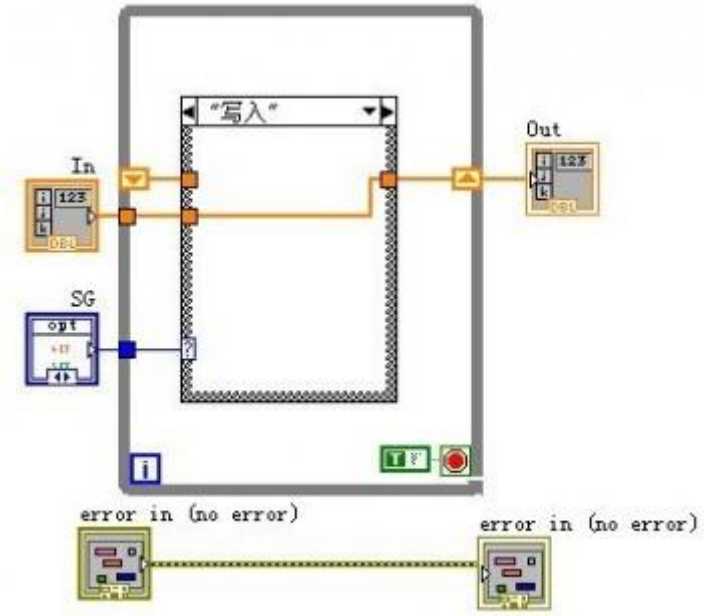
首先看一下如何在 LV 中实现共享数据，通常的做法是采用 LV2 函数全局变量，我们创建一个动作机，分配一段内存空间并实现这段内存空间的读写功能



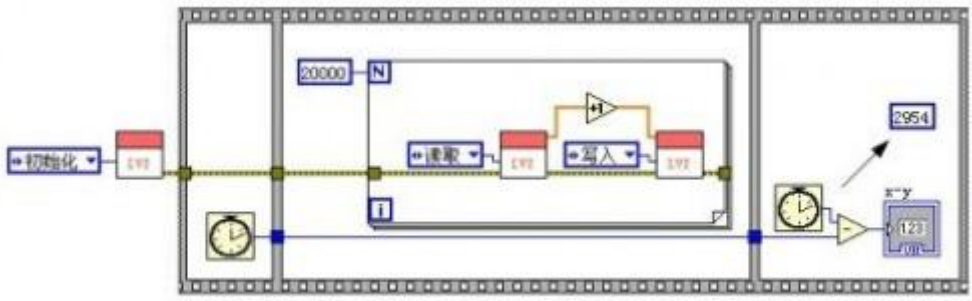
在动作机的初始化动作中，我们创建了包含 1000 个元素的数组，并存入移位寄存器中。

动作机的读、写动作如下图所示：

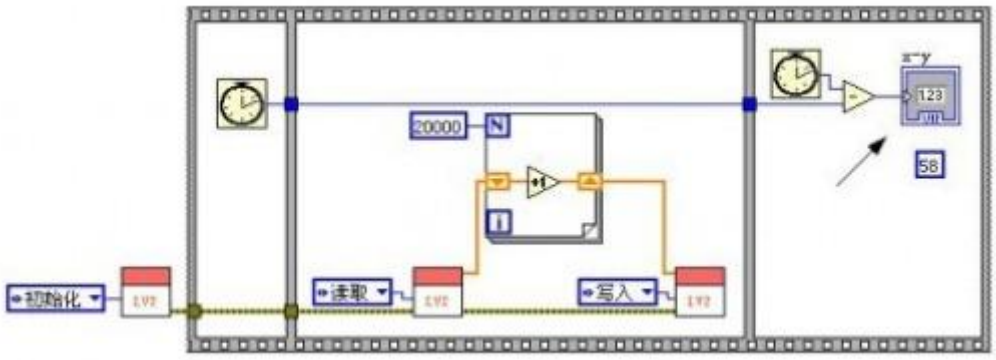




这样通过动作机，在 LV2 中封装包含 1000 个元素的双精度数组作为共享数据。具体调用方法如下图所示：

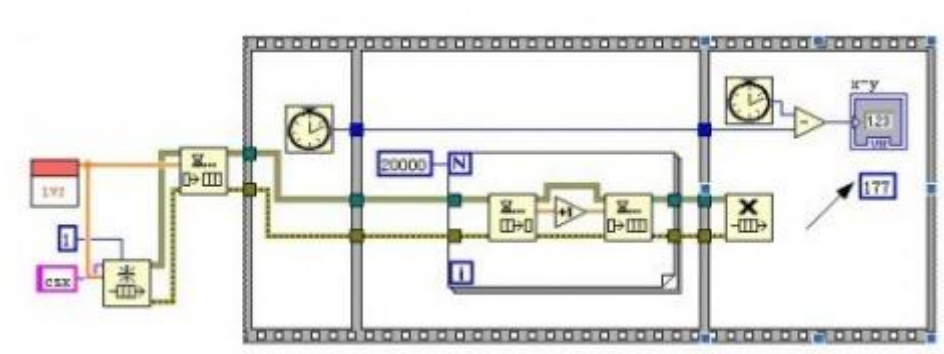


上图模拟了多处多次调用共享数据的方法，运行时间约 2954 毫秒。如果是在一个循环中连续多次调用，使用移位寄存器可以实现共享数据的同地址操作，如下图所示：



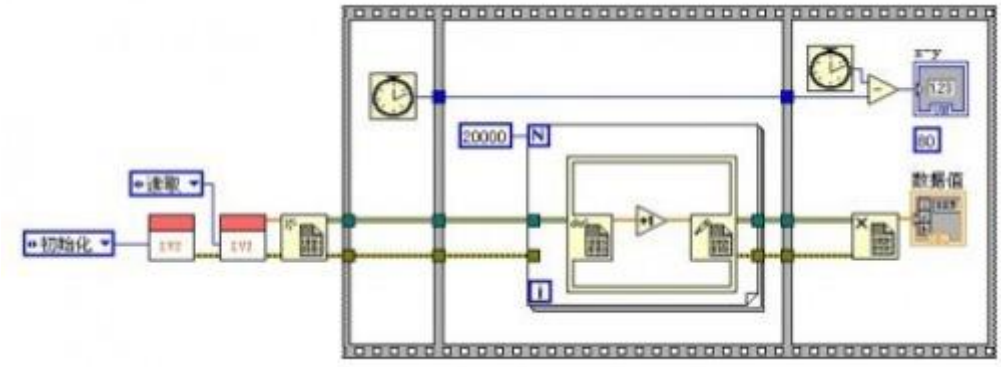
上图运行时间仅为 58 毫秒，速度远高于一般的值传递方式，上图中的方式仅适合在循环中多次调用的情形，对多处调用只能采用值传递方式。

在 9.0 以前，通过队列也可以实现引用传递，把数组放在队列中，队列本身传递的是引用，如下图所示:



上图队列之包含一个元素，目的是保证多线程运行时，某一时刻只能有一个线程操作队列中的元素，避免了多线程中数据竞争的问题。从上图可以看出，通过引用操作大型数据，速度远高于数据流方式。

下面采用 LV 新增的数据值引用函数。



采用新的引用方式传递数据，速度相当于直接采用移位寄存器，远高于普通的值传递方式。