第 9 章　多循环程序框架

第 8 章详细讲解了各种单循环程序框架（汇总见表 9-1），开发大多数单任务系统均可使用那些讨论到的模板。但是即使植入了事件结构，单循环程序框架还会在某个程序分支耗时较大时降低程序的响应速度。更重要的是在实际虚拟仪器项目开发过程中，往往需要同时处理多个任务，仅使用单循环程序框架并不能完美地达到目的。例如要设计程序来高效、准确、迅速地控制机械臂自由移动，如果使用单循环框架，则可能出现发送停止运动的命令时，它仍在执行上个循环中的旋转动作，从而导致机械臂或其周围物体的损坏；相反，如果机械臂的每一个可控部分都由一个与之对应的循环来控制，则可以提高用户界面对机械臂的控制速度，减少安全事故。此外，在虚拟仪器项目开发中，一边进行数据采集，一边进行其他仪器控制或数据分析等工作的应用尤为常见。因此，近年来，系统和程序可同时执行多任务、多线程和多处理的能力备受关注，在某些场合下甚至必不可少。

表 9-1　程序框架汇总

程序框架	程 序 结 构	数据通信方式	备　　注
单循环	轮询	局部变量或全局变量	效率低
	功能全局量	移位寄存器（Shift Register）	可防止数据访问冲突
	经典状态机	移位寄存器	无缓冲
		枚举（Enum）	
	消息状态机	消息数组	带缓冲
	队列状态机	队列数据	
	事件状态机	事件及其携带的数据	带缓冲； 用于响应 GUI 操作
多循环	变量控制并行结构	局部变量或全局变量	资源访问冲突
	同步多循环结构	通知器（Notifier）	无缓冲，同步
		事件发生函数（Occurrence）	
		软件触发定时源 （Software-Triggered Timing Source）	
		同步定时结构起始时间 （Synchronize Timed Structure Starts）	
		集合点（Rendezvous）	
	异步多循环结构	队列数据	带缓冲，异步
		事件及其携带的数据	用于响应 GUI 操作

多任务、多线程和多处理在 LabVIEW 程序中具体表现为多个循环。对多循环程序框架的研究有助于用户快速开发具有多任务、多线程和多处理能力的应用。多循环程序框架一般使用两个或两个以上的循环分别处理不同的任务，按照多个循环之间逻辑关系不同，多循环程序框架可分为并行结构、主 / 从结构和生产 / 消费结构。无论何种并行结构，要使多个循环之间协调工作，都需要依靠它们之间的通信数据来保证。因此，研究并行结构可从分析多个循环之间使用何种数据通信方式协调运作入手。

控制多循环协调运行的数据通信方式多种多样，但总的来说可归纳为变量控制、同步和

异步执行的循环等几大类。使用不同的控制方式创建的程序框架各不相同，各有优缺点。本章就来详细讲解多循环程序框架的构成和使用时的技术特点。首先讲解与多任务系统相关的一些基本概念。

9.1 多任务、多线程、多处理与多循环

多任务（Multitasking）是指操作系统在任务之间快速切换，实现多个任务同时运行的能力。每个应用程序在很短的时间内完成运行，然后运行下一个应用程序。例如，在 Windows 中，任务通常就是一个应用程序，如 Microsoft Word、Microsoft Excel、LabVIEW 等。多任务技术又有协同式多任务（Cooperative Multitasking）技术和抢占式多任务（Preemptive Multitasking）技术之分。使用前者的系统中，正在运行的应用程序定期将处理器的控制权转交给操作系统，而采用后者的系统中，不管应用程序运行处于什么状态，操作系统都可以在任何时候获得处理器的控制权，以更好地保证对用户的响应和更快的数据吞吐率。Windows 2000/XP 采用的就是抢占式多任务技术。

多线程（Multithreads）是将多任务的概念扩展到应用程序中，在单一的应用程序中将特定的操作划分成独立的线程，每个线程理论上能够并行运行。这样操作系统不仅可以给不同的任务（应用程序）分配处理时间，也可以给同一个任务中的每个线程分配处理时间。在进行 LabVIEW 虚拟仪器多线程项目开发时，每个线程表现为程序中的一个循环。通常可以将应用程序分为三个线程：用户界面线程、数据采集线程和仪器控制线程，并对每个线程都指定优先级独立执行，以实现并行的多任务处理能力。

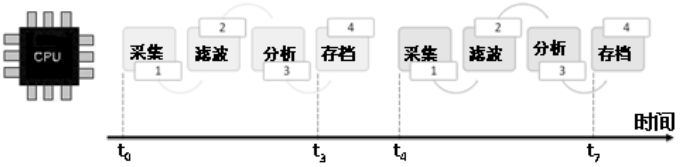
多处理（Multiprocessing）又称为多核编程，是指一台计算机有两个或两个以上的处理器，每个处理器运行一个单独的线程。单线程应用程序一次只能在一个处理器上运行，而多线程应用程序能够同时在多个处理器上运行多个单独的线程。在进行 LabVIEW 虚拟仪器多线程项目开发时，可以分别使不同的线程在不同处理器上运行，而单线程应用程序则会对系统性能造成严重影响。

使用多任务、多线程和多处理的最大优势之一在于控制多处理器计算机的能力。单线程应用程序只能在单个处理器上运行（图 9-1（a）），因此难以发挥多处理器的优势来提高性能。目前多数高端计算机都有两个或两个以上的处理器，以提高运算能力。在多线程应用程序中，可以由系统自动或手动指定不同的线程在不同的处理器上执行，真正在一台多处理器计算机上并行执行任务，从而提高整个系统的性能（图 9-1（b））。

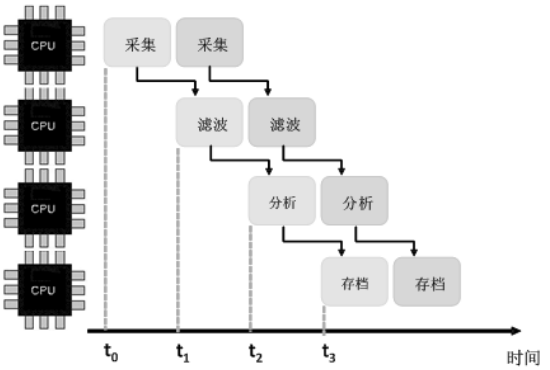
使用多任务、多线程和多处理可以更高效地使用 CPU。在许多 LabVIEW 应用中常需要同时进行多个操作，而完成每个操作又需要花费较长时间。如果使用单线程（单循环）框架，在完成某个操作之前，程序将阻碍或阻止其他操作的运行。多线程技术通过为每个操作分配一个单独的线程来消除这种阻碍，执行任务的多个线程均同时独立地运行，避免了等待。此外，任何线程空闲时，原先由其占用的处理器资源就会被释放出来参加到其余线程的处理中，大幅提高 CPU 的利用率。

使用多任务、多线程和多处理还能确保程序的可靠性，确保其他操作不影响重要操作。例如，在 LabVIEW 多线程应用中，用户界面操作一般被分配到一个固定的用户界面线程（UI Thread），而数据采集、分析以及文件操作等会在其他线程上运行。假定其他线程相对于用户界面线程更重要，则可以将用户界面线程的优先级设置为最低，以确保用户界面操作不会对 CPU 执行其他重要操作带来影响。另外，在进行高速数据采集及显示时，数据采集和显示会相互独立运行，数据采集线程连续运行，并把数据不间断地传到缓冲区，用户界面线程一

旦发现缓冲区中有数据就尽可能快地显示数据。由于数据采集线程优先于用户界面线程，所以当屏幕更新时，不会影响数据采集线程的运行，所以不会造成数据丢失，从而提高了数据的安全性和系统的可靠性。



（a）单线程程序运行



（b）多处理程序运行

图 9-1　程序运行示意图

如前所述，多任务、多线程和多处理在 LabVIEW 程序中表现为多循环。因此，对多循环程序框架的研究有助于快速开发具有多任务、多线程和多处理能力的应用。多循环程序框架一般使用两个或两个以上的循环分别处理不同的任务，循环之间的通信数据可以确保各循环之间协调有序地工作。例如，在进行数据采集和分析时，如果将两个任务和用户界面事件处理放在一个处理循环或状态机中，不仅逻辑上比较混乱，而且任何阻止循环的事件都会影响程序的响应和执行效率；相反，如果将各种任务独立开来，分别为数据采集、数据分析和用户界面事件处理分配独立的循环，并使用数据通信协调三个循环之间的工作，就可以实现响应快速、执行效率高的程序。

在实际设计过程中，根据不同循环之间的逻辑关系，多循环框架的构成可以大致分为以下三种结构。

（1）并行（Parallelism）结构。

并行结构可同时处理多个独立的任务，每个任务由一个独立循环实现，适用于一些简单的菜单型 VI （用户可从多个按钮中选择某一个按钮以执行不同的操作）。并行结构中，对某一操作的响应并不阻碍 VI 对另一个操作的响应。例如，当用户单击一个显示对话框的按钮时，并行循环仍可以继续响应 I/O 任务。这里所说的并行结构并不是完全互不相干，可以通过循环之间的数据通信协调各任务的运行，如图 9-2 所示。

（2）主 / 从（Master/Slave）结构。

主 / 从结构中每个循环以不同的速率执行任务，但一个循环作为主循环，其他循环作为从循环。主循环控制所有的从循环，并且使用数据通信技术在循环之间进行通信，如图 9-3 所示。通常，如果要求 VI 在对用户界面的控件作出响应的同时还可以采集数据，则使用

主 / 从结构比较合适。 例如，假设需要创建一个 VI，要求其每 5s 就测量并记录一次缓慢变化的电压值，并且每 100ms 显示一次采集到的数据，此外还要求用户也可以在界面上改变每次采集的参数，该应用程序就非常适合采用主 / 从结构。主循环中包含用户界面，采集电压发生在一个从循环中，画图发生在另一个从循环中。主循环负责接收用户界面的控件输入，并驱动从循环。这样不仅保证了独立的采集过程之间互不影响，而且由用户界面引起的任何延时（如显示对话框）都不会导致采集过程的循环操作产生延时。

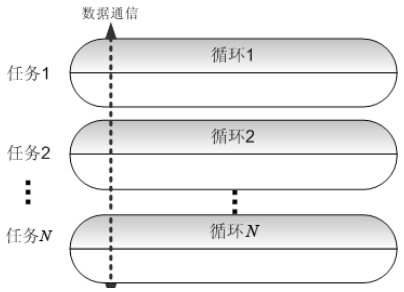


图 9-2　并行结构示意图

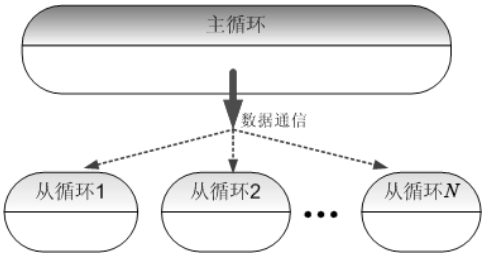


图 9-3　主 / 从结构示意图

一般来说，主 / 从结构中从循环对主循环的响应时间不能太长。如果从循环正在处理一个来自主循环的请求，而主循环传递了多个消息给从循环，那么从循环只接收最后一个消息（带缓冲的结构另当别论）。这种情况下，使用主 / 从结构会造成数据丢失。 只有在确信每个从循环的执行时间都小于主循环执行时间的情况下，才可以使用主 / 从结构。此外，在使用主 / 从结构时，还要避免共享数据冲突的问题，确保同一时间只有一个循环对数据进行访问。

（3）生产者 / 消费者（Producer/Consumer）结构。

生产者 / 消费者结构可以看作是主 / 从结构的升级版本，它提高了不同速率的多个循环之间的数据共享能力。 一方面，与主 / 从结构相似，它将“生产”数据和“消费”数据的任务分开处理，为每个任务分配单独循环；另一方面，它使用缓冲的方式（如队列、栈、消息数组或事件等）在循环之间传递数据（图 9-4）。

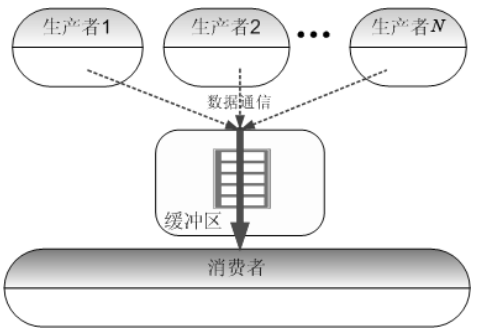


图 9-4　生产者 / 消费者结构示意图

通常，如果要按顺序处理多个数据，则应使用生产 / 消费结构。生产者 / 消费者结构在消费循环以固定速度处理数据的同时，允许生产循环继续按顺序生产数据。例如，在对网络数据进行分析时，一般需要两个同时运行但速度不同的循环。其中一个循环不断抓取数据包，另一个循环分析第一个循环抓取的数据包。使用生产者 / 消费者结构设计程序时，因为第一个循环供给第二个循环数据，因此第一个循环可以看作是数据生产者，而第二个循环分析数据，因此是数据消费者。可以使用队列在两个循环之间缓冲抓取的网络数据包。确保数据分析按顺序进行，且不丢失任何抓取的数据包。

如前所述，研究并行结构可以从分析程序使用何种数据通信方式来控制多个循环协调运作入手。控制多循环协调运行的方式可归纳为以下常用的几类。

（1）变量控制：由布尔、枚举等多种数据类型的局部变量或全局变量来控制多个循环的协调运行。

（2）同步多循环：由事件发生函数（Occurrence）、集合点（Rendezvous）、通知器（Notifier）或软件定时源等控制多个循环同步执行。

（3）异步多循环：使用带缓冲的消息数组、队列、栈或用户自定义事件等控制多个循环的异步执行。

使用变量通信方式控制多个循环的运行，可以很容易地构建多循环程序框架，但是却存在轮询和数据访问冲突的问题。同步多循环一般用于需要多个循环同时执行的情况，并可以有效避免轮询对系统资源的消耗。但是同步多循环一般每次只能对一个触发请求作出响应，而且触发不能被缓冲。带缓冲的消息数组、队列或栈可以在并行循环之间传送异步消息，并且还可以连续缓存多个消息。这种方式可以确保用户请求不会丢失。用户自定义事件也可以按照先进先出的顺序同步多个循环，并可以与消息方式一样携带数据到其他循环中。但是多数情况下事件用于处理用户界面事件，并且事件结构通常具有“一对多”的特点。例如可以使用一个独立的循环和事件结构配合来处理用户界面事件，随后再通知不同的数据处理循环进行数据处理。队列则正好相反，用于处理“多对一”的任务关系，即程序中不同位置的用户请求可以通过队列汇总在一处。下面就根据不同的控制方式讲解多循环结构。

9.2　变量控制多循环

LabVIEW 以数据流驱动的形式决定程序框图元素的执行顺序，因此与生俱来具有创建多循环程序的能力。虽然在这一点上比传统的文本编程工具简单很多，但是要使多个独立运行的循环协作完成任务，还需要循环之间的数据通信来同步。变量可以作为多循环之间最简单、直接的数据通信方式。

LabVIEW 中的变量是程序框图中的元素，通过它可以在另一位置访问或存储数据。变量的类型不同，数据的实际存储位置也不一样。变量通常可分为以下几类。

（1）局部变量：将数据存储在前面板的输入控件和显示控件中。

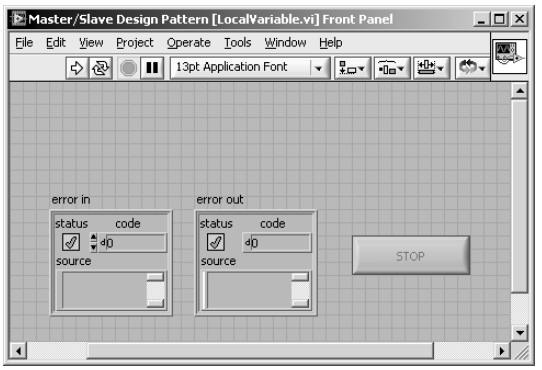
（2）功能全局变量：将数据存储在 While 循环的移位寄存器中。

（3）全局变量：将数据存储在项目中其他 VI 可访问的区域。

（4）单进程共享变量（Shared Variables）：将数据存储在多个 VI 可以访问的位置。

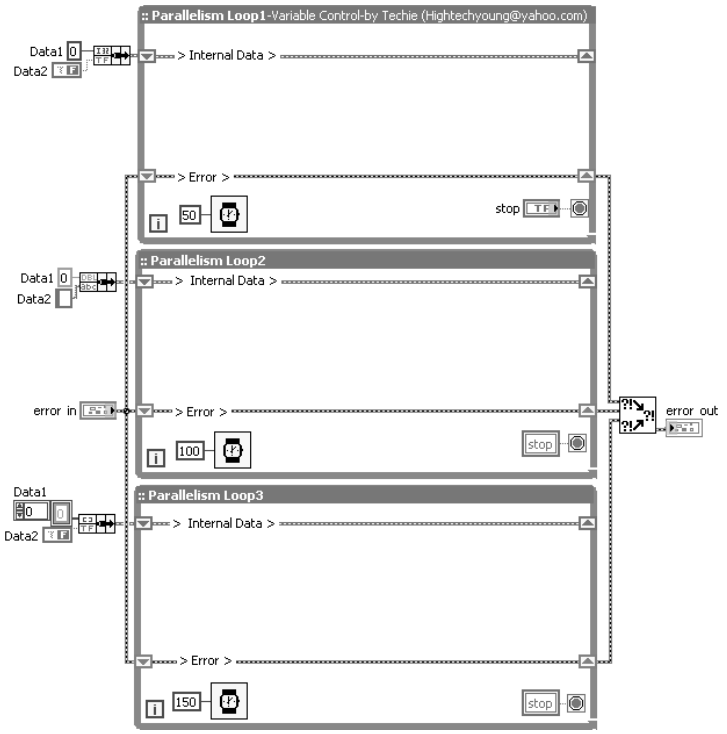
不管变量将数据存储在何处，所有的变量都可以在不使用连线连接的条件下把数据从一个地方传递到另一个地方，从而不必使用正常的数据流。

图 9-5 显示了使用局部变量在多个循环之间进行通信来控制三个循环同时结束的例子。在图 9-5（b）所示的程序中，第一个循环中的 stop 控件用于控制循环的结束，其他两个循环使用局部变量获取 stop 控件的值，以达到第一个循环退出时也能同时退出的目的。



（a）前面板

图 9-5　使用局部变量控制多循环的框架



（b）后面板

图 9-5（续）

程序框架还有以下几个需要注意的要点。

1. 每个循环中均使用簇和移位寄存器传递该循环内部的数据

簇的元素可以是任意指定的数据类型。为了说明这一点，例子中分别使用布尔量、整型、浮点型、字符串以及数组来组合在每个循环内部传递数据的簇。读者可能会觉得传递数据的簇显得多余，但是在使用这种框架开发大型项目时，这一特点的优越性就会体现出来。

2. 三个循环的优先级由各个循环中的等待时间决定

LabVIEW 中多个循环并行执行时，相对来说循环的等待时间越少，循环的优先级别越高。在图 9-5 的三个循环中，自上而下的三个循环的优先级依次降低。

3. 使用多循环结构中典型的错误捕获方法

三个循环中的错误捕获分别由三个移位寄存器完成，并在循环结束后由一个“错误合并”函数合并。

使用变量协调并行循环运行的实现方式较为简单。通常在应用程序中，当各循环的执行速度相当，并且处理的任务不是很复杂时，多使用这种结构。但是由于涉及多个循环中对一个变量的访问，因此资源冲突的问题不可避免。此外，如果循环的执行速度差异较大，则执行较慢的循环可能会遗漏对某些变量值变化的监测。如图 9-5 所示的例子中，假定第一个循环执行速度较快，第二和第三个循环由于执行某些任务相对耗时较长，那么如果用户单击 Stop 按钮时，可能恰巧第二和第三个循环还未完成其单次循环要处理的任务。但是当其完成任务读取 Stop 的值时，用户单击 Stop 按钮的动作早已结束，因此会遗漏对用户请求的监测。这一问题的本质其实还是资源访问冲突的问题，实际开发中应尽量避免。如果由于各种条件限制，一定要在多个循环中对某个变量进行访问时，则可以使用“信号量”（Semaphore）规避数据访问冲突的问题。

信号量的思想来源于“旗语”。旗语是古代一种主要的通信方式，常用于战场上调兵遣将。现在海军、铁路等领域还能见到。图 9-6 是海军常用旗语，不同的旗子代表不同的意思（字母、数字或其他意思）。使用时，事先约定不同旗子或其组合代表的意思即可。例如，我国海军约定图中 A 旗代表“潜水员工作中，保持距离”，只要挂出该旗子，其他舰艇就会绕道而行。

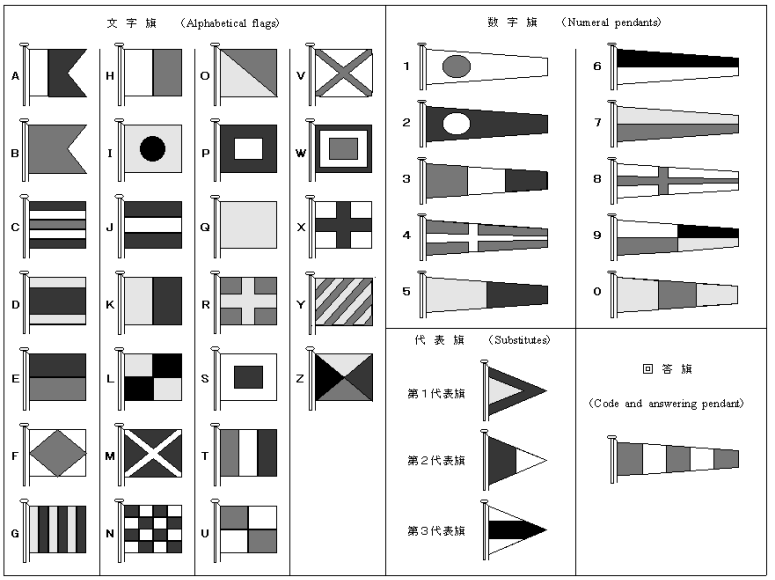


图 9-6　海军常用旗语

类似的方式在早期铁路信号系统中也很常见。如图 9-7 所示，使用多个不同颜色的信号灯将铁路线划分为多个区段，各种颜色的信号灯表示不同含义，就可以通过控制信号灯来有效组织运营。例如，车辆运行时，如遇到绿色信号灯则继续进入下一个区段；遇到黄灯则表示列车可以继续进入下个轨道区段，但是下个轨道区段前面的一个区段中可能有另一辆列车正在运行；列车遇到红灯时，必须停止，以免发生撞车事故。

图 9-7　早期铁路信号系统中信号灯的意义

例子中的轨道区段可以被看作是一种共享资源，那么通过控制信号灯就可以控制对这种资源的使用权。同理，我们可以使用信号量有效解决对某个共享变量访问时的冲突问题。LabVIEW 对信号量的操作提供完整的函数集，如图 9-8 所示。

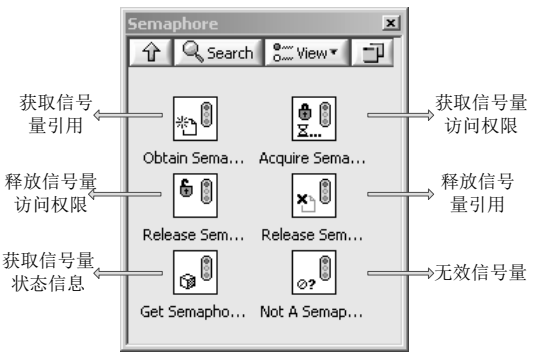


图 9-8　LabVIEW 中信号量的操作函数

LabVIEW 中操作信号量遵循下面步骤：

（1）获取信号量引用（Obtain Semaphore Reference）：要使用信号量，首先要获取对已有信号量的引用，或创建新的信号量。只有获得信号量的引用，才能继续进行信号量的相关操作。

（2）获取信号量的访问权限（Acquire Semaphore）：获得信号量的引用后，就可

以申请该信号量的使用权。获得使用权后，就可以用被占用的信号量来表示某个共享资源正在被使用，避免其他操作对共享资源的影响。可以使用“获取信号量状态”（Get Semaphore Status）函数获取信号量的当前状态。

（3）释放信号量的访问权限（Release Semaphore）：对共享资源操作完成后，可以释放信号量使用权，以便允许其他操作申请信号量的使用，从而对共享资源进行操作。

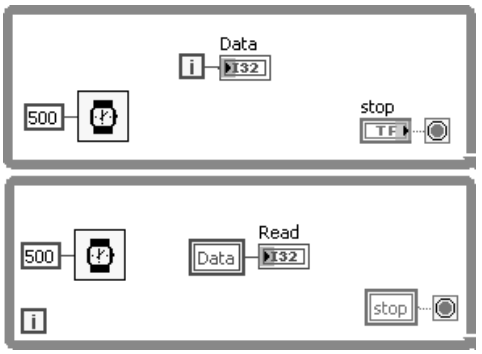


图 9-9　数据访问冲突示例

（4）释放信号量引用（Release Semaphore Reference）：信号量使用完成后，释放其引用。

下面用一个简单的例子来说明 LabVIEW 中使用信号量的重要意义和使用方法。图 9-9 所示的程序中，一个循环对指示器 Data 写入数据，另一个循环读取 Data 的数据，并显示在指示器 Read 中。然而，由于两个循环并行执行，在第二个循环读取 Data 值时，可能第一个循环已经改写了 Data 的值。在设计过程中这种“共享数据”访问冲突的问题极为常见，而且往往会导致程序运行异常（虽然对这个小程序不会带来多么严重的影响）。

图 9-10 显示了如何使用信号量来解决如图 9-9 所示的数据访问冲突问题。在例子中，首先创建一个名为 Sign 的信号量，并返回其引用值。在随后并行执行的两个循环中，均先使用“获取信号量”函数申请信号量的访问权，然后再执行数据读写的操作。只有在获得信号量的使用权后，对应的读写操作才能被执行。数据读写完成后，信号量的访问权将被释放。由于两个循环申请的是同一个信号量的使用权，这就使得其中一个获得权限后，另一个循环必须等待其释放信号量使用权后才能执行。这样就有效避免了并行循环中数据访问冲突的问题。

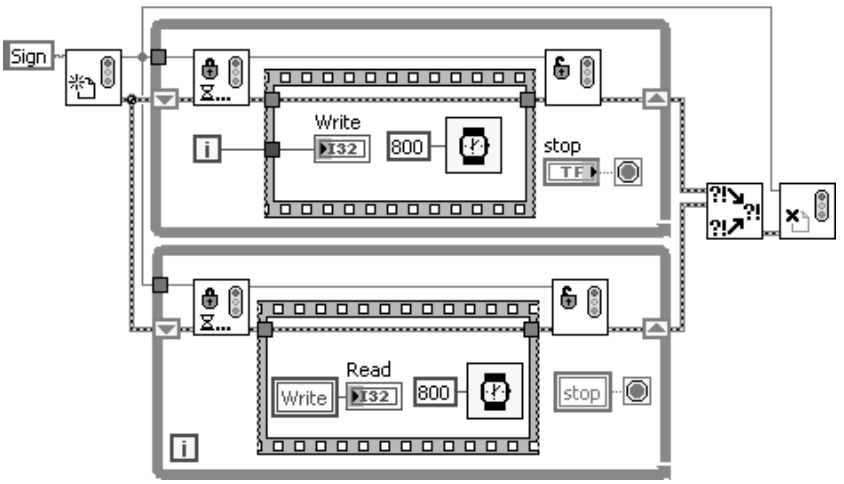


图 9-10　使用信号量解决数据访问冲突的例子

9.3　同步多循环

LabVIEW 提供以下几种方案来同步多个循环的运行：

（1）事件发生函数（Occurrence）。

（2）通知器（Notifier）。

（3）软件触发定时源（Software-Triggered Timing Source）。

（4）同步定时结构起始时间（Synchronize Timed Structure Starts）。

（5）集合点（Rendezvous）。

由于这几种方式都不对消息进行缓冲，因此它们本质上是一种“软触发”的控制方式。

9.3.1　事件发生函数控制多循环

LabVIEW 提供事件发生函数集，如图 9-11 所示。其中包括：

（1）创建事件发生函数（Generate Occurrence）：用户创建可传递至“等待事件发生”函数和“设置事件发生”函数的事件。

图 9-11　事件发生函数集

（2）等待事件发生函数（Wait on Occurrence）：用于等待“设置事件发生函数”设置某个事件发生。

（3）设置事件发生函数（Set Occurrence）：用于设置指定的事件发生，使所有正在等待该指定事件发生的节点停止等待。

使用事件发生函数控制多循环的同步运行方法比较简单，我们以图 9-12 所示的框架为例进行说明。在图 9-12 的程序中，首先使用“创建事件发生”函数生成程序要使用的事件，并将其传递至三个并行执行的循环。三个循环中一个为主控循环 Master，另两个为被控循环 Slave。主控循环中的 Start 按钮用来控制“设置事件发生”函数是否设置事件发生。被控循环中的“等待事件发生”函数一旦发现事件被设置，则立即将循环次数赋值给 Numbers 1 和 Numbers 2 指示器，否则就挂起循环的执行，进入等待状态。如果主循环中的 Start 按钮一直为按下状态，则“设置事件发生”函数就连续设置事件发生。被控循环中的“等待事件发生”函数也会相应地不断执行赋值操作，直到切换 Start 按钮的状态或退出循环为止。程序退出后，必须再运行一次“设置事件发生”函数，以便程序能顺利退出。

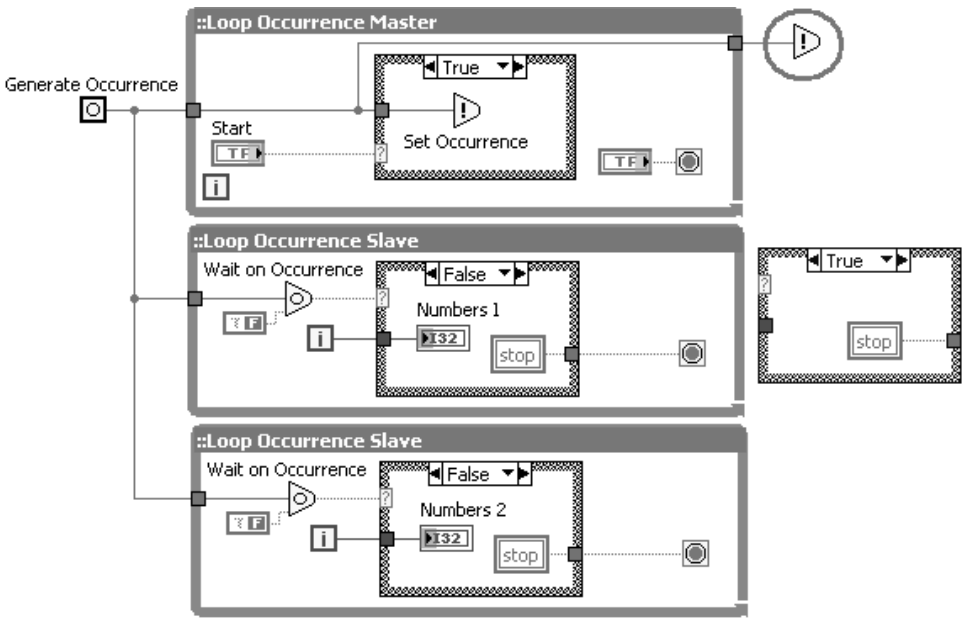


图 9-12　事件发生函数控制多循环

由于程序在被控循环中设置了“等待事件发生”函数的“忽略上一个值”参数为FALSE，因此函数将检查之前是否存在另一个节点已设置过事件发生。如果存在，函数就会执行；相反，如果函数的“忽略上一个值”参数为TRUE，并且另一个节点在该函数开始执行之前已经设置了事件发生，则函数将忽略上一个事件发生而等待另一个事件发生。

需要注意的是，“等待事件发生”函数每次只处理一个事件发生，并不会在内存中保存多个事件发生。而且使用事件发生控制多循环的方法是LabVIEW早期版本使用的技术，目前完全可以使用通知器（Notifier）技术来取代。

9.3.2 通知器控制多循环

在LabVIEW中，通知器操作函数用于挂起部分程序的执行，直到从程序的另一部分收到“通知消息”后才重新启动挂起部分程序的执行。LabVIEW提供的用于实现通知器操作的函数集如图9-13所示。

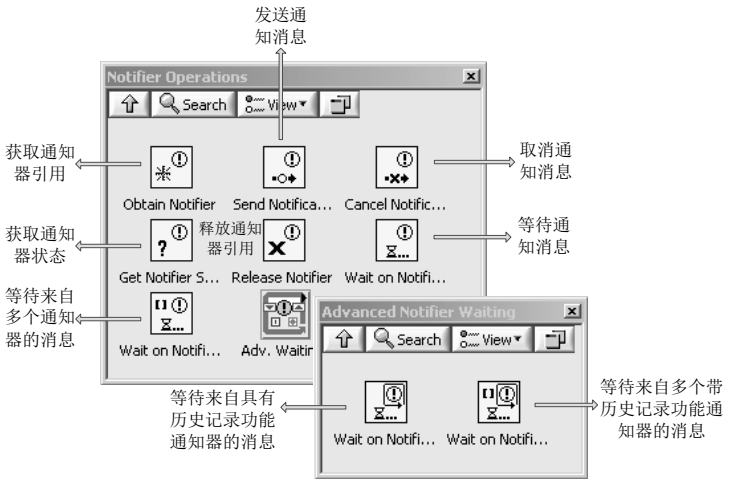


图9-13 LabVIEW实现通知器操作的函数集

通知器的使用方法一般遵循以下步骤：

（1）获取通知器引用（Obtain Notifier）。要对通知器进行操作，必须先获得它的引用。

（2）等待通知消息（Wait On Notification/ from Multiple）。获得通知器的引用后，就可以使用等待通知消息函数来部署需要同步的多个程序块。等待函数会挂起程序执行，直到接收到通知器发来的通知消息为止。为了适应不同的应用，LabVIEW提供四种等待通知消息函数，如表9-2所示。

表9-2 LabVIEW提供的四种等待通知消息函数

等待通知消息函数	通知器个数	有无上次消息记录
等待通知函数（Wait On Notification）	单个	无
等待具有历史记录功能通知器的消息函数（Wait On Notification with Notifier History）	单个	有
等待来自多个通知器的消息函数（Wait On Notification from Multiple）	多个	无
等待来自多个具有历史记录功能通知器的消息函数（Wait On Notification from Multiple with Notifier History）	多个	有

表 9-2 中的四个函数可以通过“是否支持消息记录”和支持通知器的个数来区分。支持消息记录功能的等待函数会对通知器的最新消息和时间标识符进行跟踪，以防止在与不同的通知器重复进行通信时，发生消息遗漏的情况。另一方面，如果需要同时监测多个通知器的消息，则可以使用支持多个通知器消息的等待函数。

（3）发送通知消息（Send Notifications）。如果要使等待函数停止等待并转为继续执行，则可以通过“发送通知消息”函数向所有等待该通知器通知消息的函数发送通知消息。

（4）释放通知器引用（Release Notifier）。在使用完通知器后，必须释放为其分配的内存等系统资源，可以通过“释放通知器引用”实现。

图 9-14 为使用通知器控制多循环的一种简单程序框架。

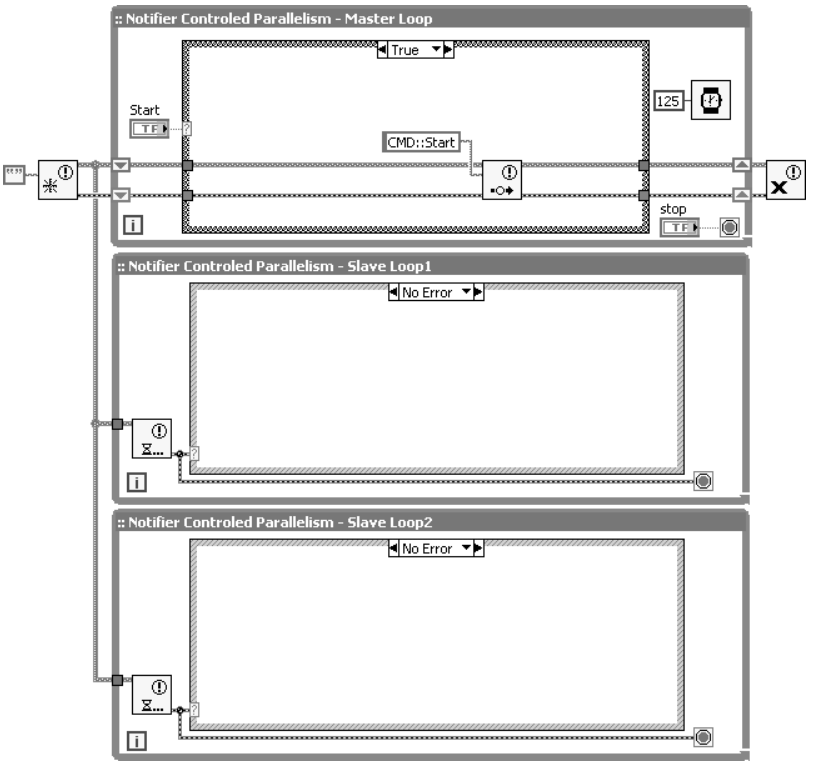


图 9-14　使用通知器控制多循环的一种简单程序框架

程序一开始先使用“获取通知器引用”函数为通知器分配资源，随后将通知器的引用同时传送给三个并行执行的循环。三个循环中第一个循环为主控循环，其他两个为受控循环。当主控循环中的 Start 按钮被按下时，其中的“发送通知消息”函数会发出 CMD::Start 消息。两个受控循环中的“等待通知消息”函数平时处于等待通知器通知消息的状态，并挂起循环的执行。一旦收到主控循环发送的通知消息，就会停止等待、继续执行。同时，选择结构中的代码也将被执行。如果用户按下 Stop 按钮后，主循环退出运行，循环外的“释放通知器引用”函数将释放通知器在内存中的资源。两个从循环中的“等待通知消息”函数将出错，因此两个循环将停止执行，这样就达到了结束程序运行的目的。

LabVIEW 的模板中也提供类似的程序框架，但是比图 9-14 所示的程序少了一个受控循环。这里用同步三个循环的例子来说明通知器的使用，主要的目的是希望读者可以理解这种框架能控制的循环数并不局限在一两个，而是可以扩展到多个循环。

我们还可以对图 9-14 中的程序框架进行一些改进，使其更接近实际项目开发的需求。改