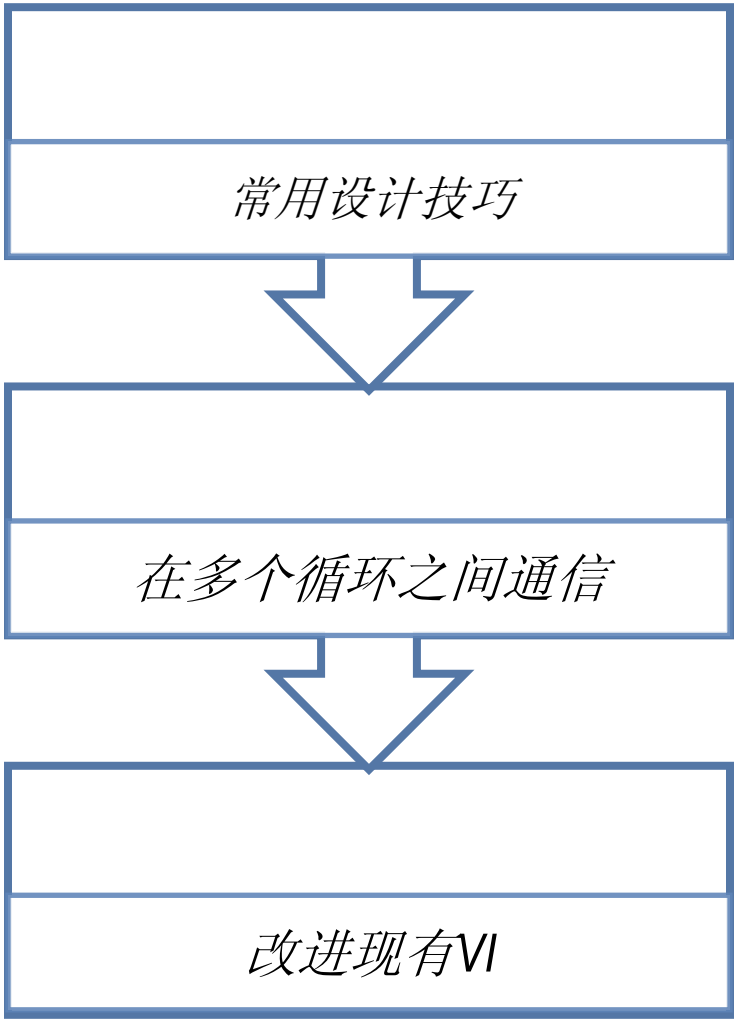
LabVIEW进阶培训（二）

课程学习导读



目标

本教程旨在帮助您：

- 有效地使用事件编程。
- 使用序列和事件的常用设计模型。
- 通过编程控制用户界面对象。
- 修改现有的代码，以增强可用性。

第1部分
常用设计技巧

A. 设计模式
B. 简单设计模式
C. 事件驱动编程
D. 多循环架构
E. 为设计模式设置定时

A. 设计模式

为什么要使用设计模式？

- 有益于软件开发。
- 无需从头开始创建应用程序。
- 更便于其他人员阅读和修改程序代码。

设计模式通常在多个开发人员的努力下逐步完善，且不断调整以增强简化性、可维护性和可读性。

B.简单设计模式

顺序编程

简单VI

通用VI

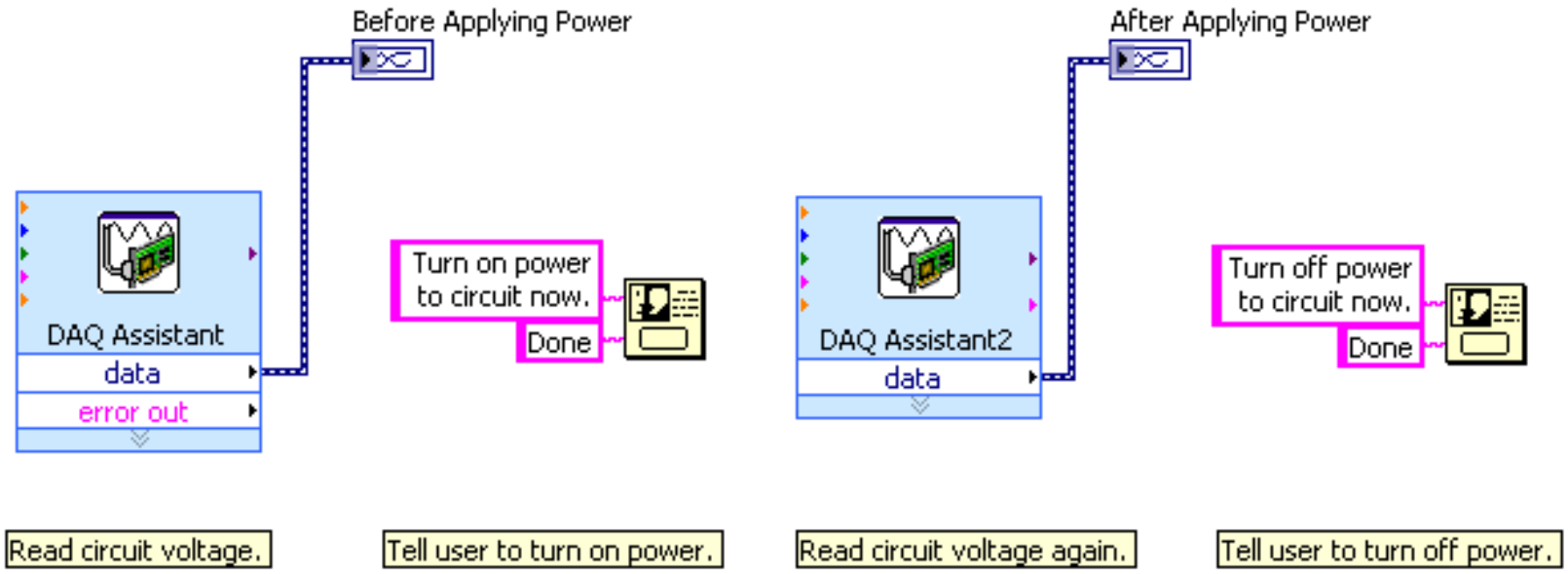
状态机

基于事件的状态机

简单状态机模板

顺序编程

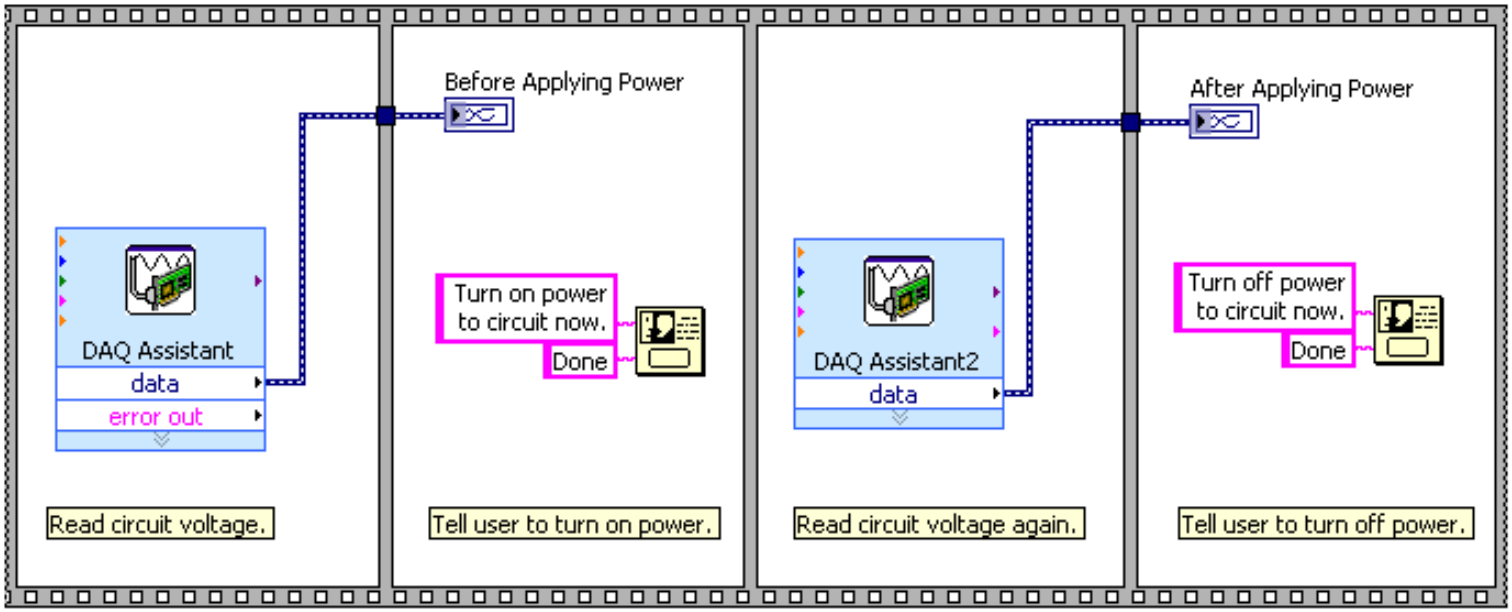
- 大多数编写的VI都是实现顺序任务



- 该程序框图中没有强行指定这些任务的执行顺序-它们中的任何一个都可以先发生

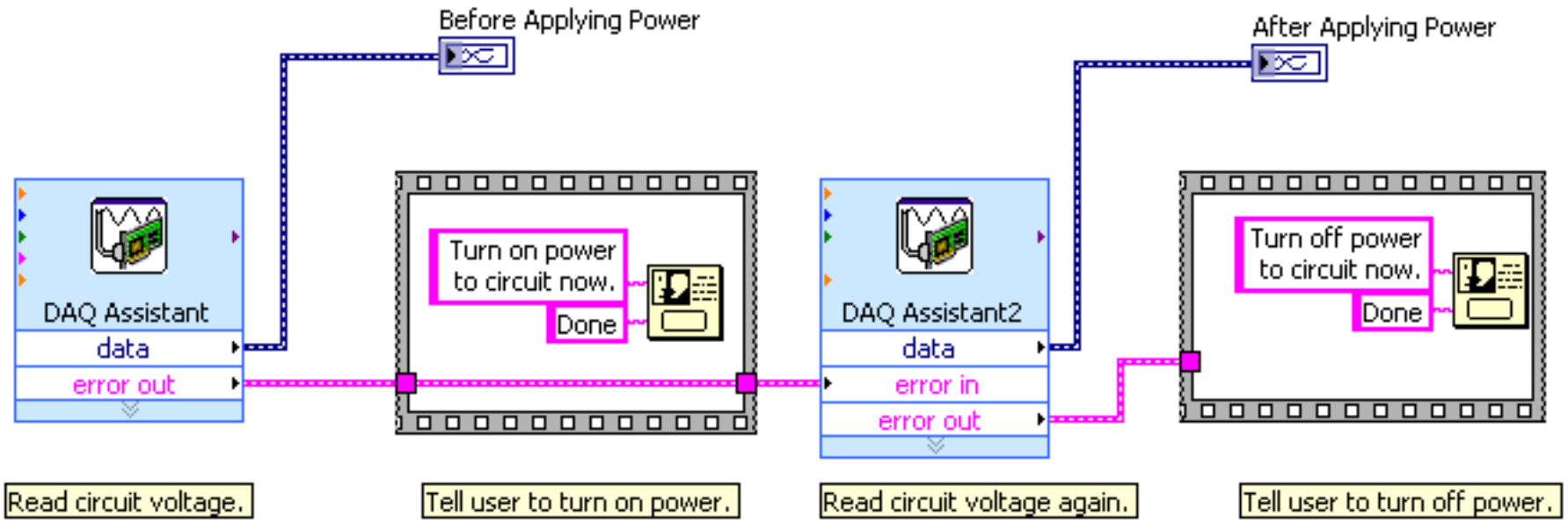
顺序编程

- 如要强行指定执行顺序，请使用顺序结构
  - 由多个帧组成的结构，每一帧均按顺序执行
  - 在第一帧没有完全执行完之前不能执行第二帧



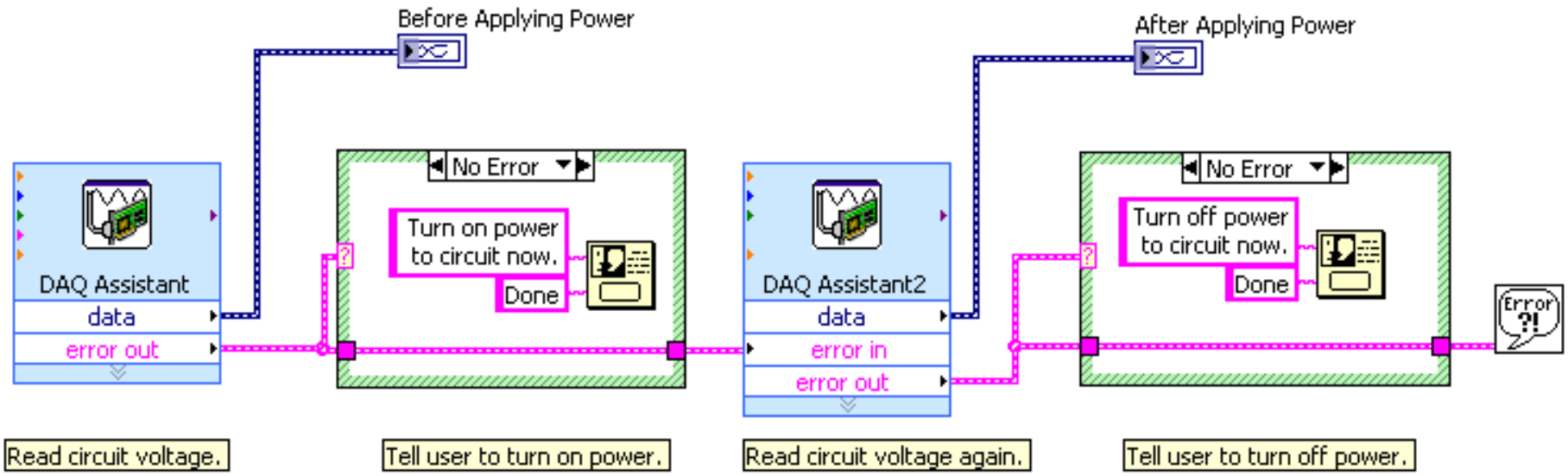
顺序编程

- 避免使用太多顺序结构
- 顺序执行的中途不能停止执行



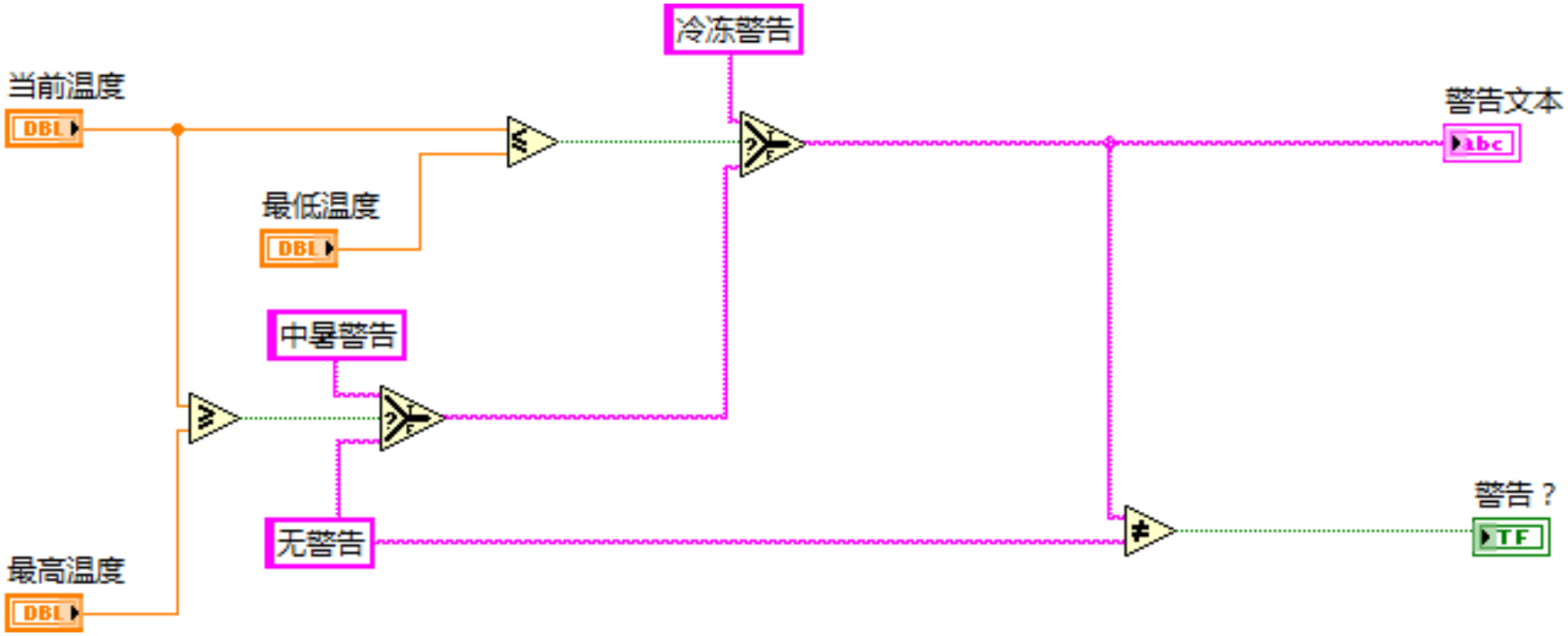
顺序编程

- 编写该VI的最好的方法是将对话框包入条件结构中，将错误簇和条件选择器相连接



简单VI模式

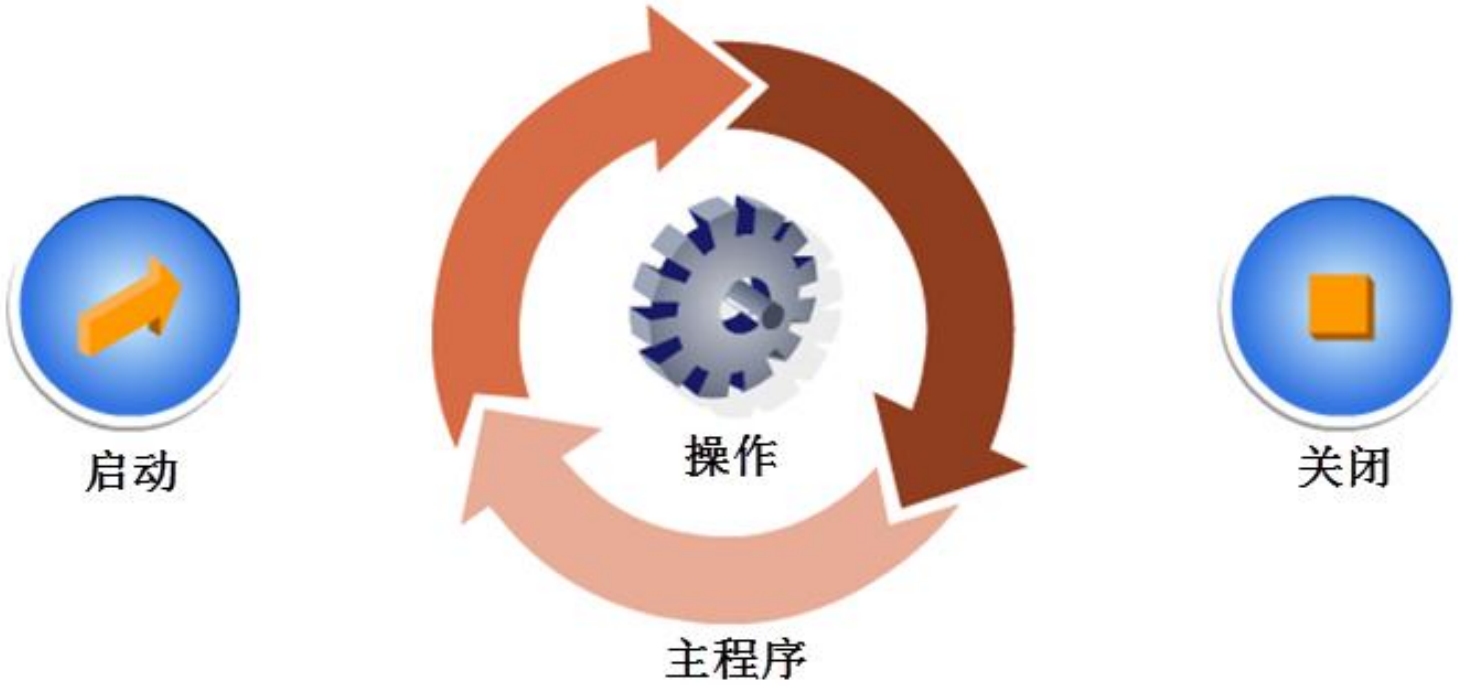
- 完成测量、计算、显示结果或将结果记录到磁盘的单个VI。
- 通常无需用户执行指定起始或停止动作。



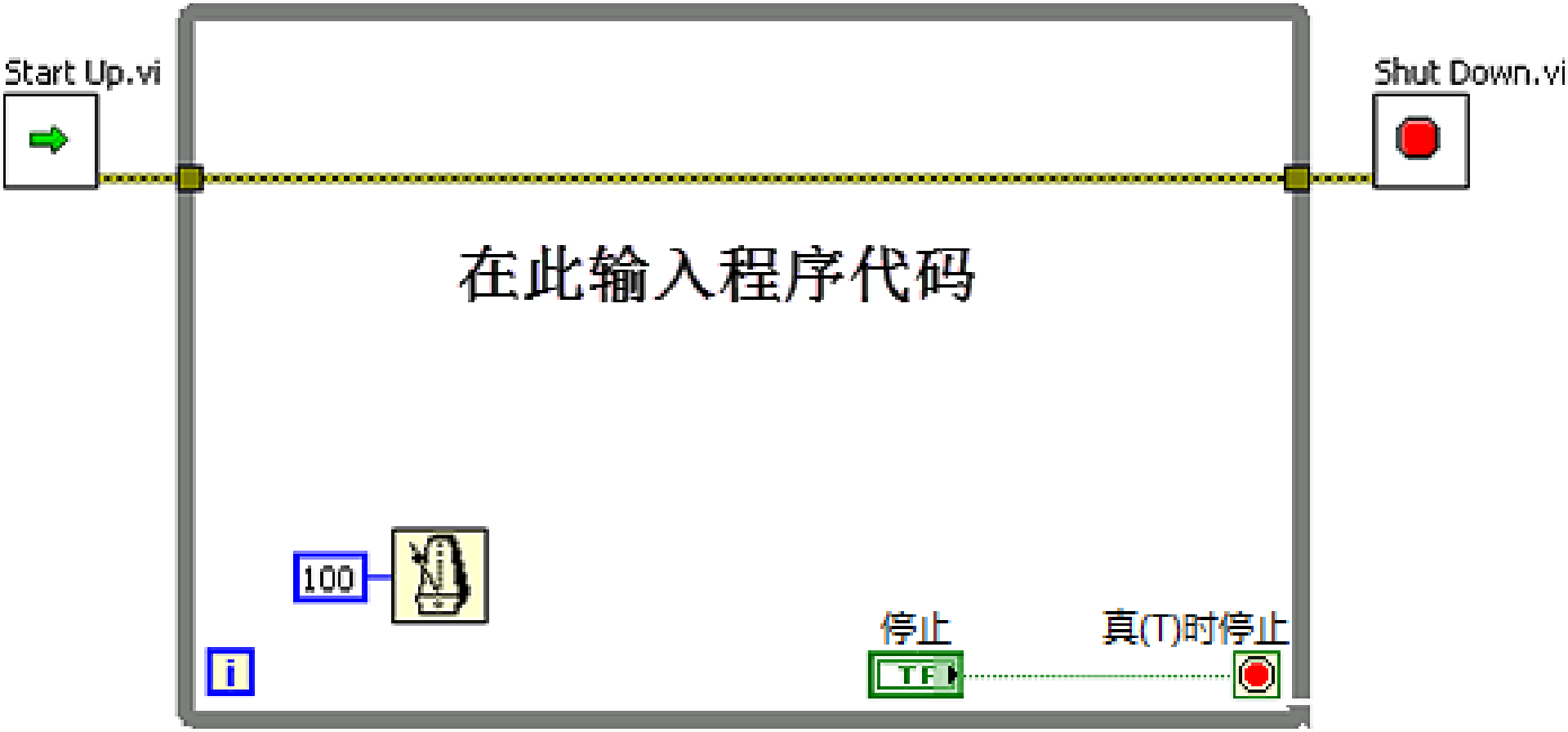
通用VI模式

三个组成部分：

- 启动
- 主程序
- 关闭



通用VI框架

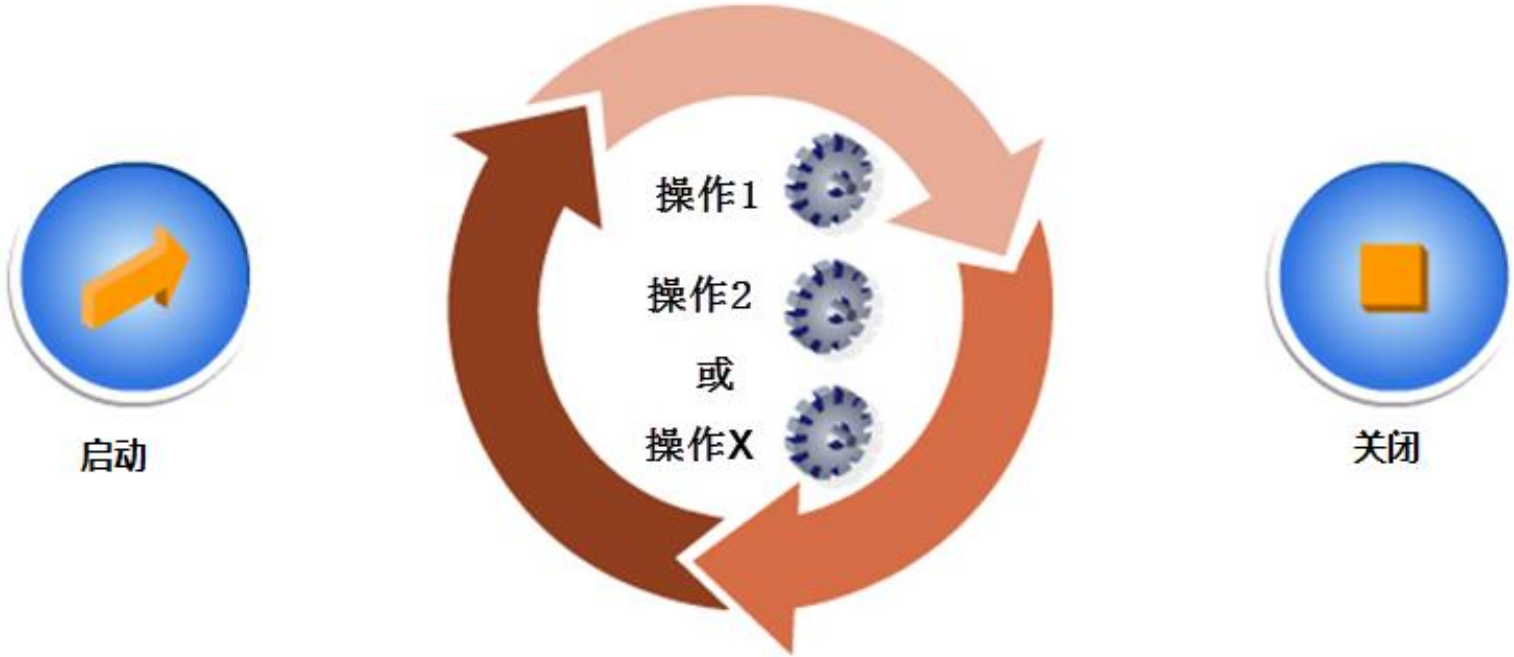


状态机模式

- 使用顺序结构和顺序连接各子VI可以实现这个任务，但这些通常不是最好的选择：
  - 如须改变执行的顺序怎么办？
  - 如须多次重复执行序列中某一帧怎么办？
  - 如须仅在满足一定条件时才执行某几帧怎么办？
  - 如须立刻停止该程序而不是等到最后一帧执行完才停止该程序怎么办？

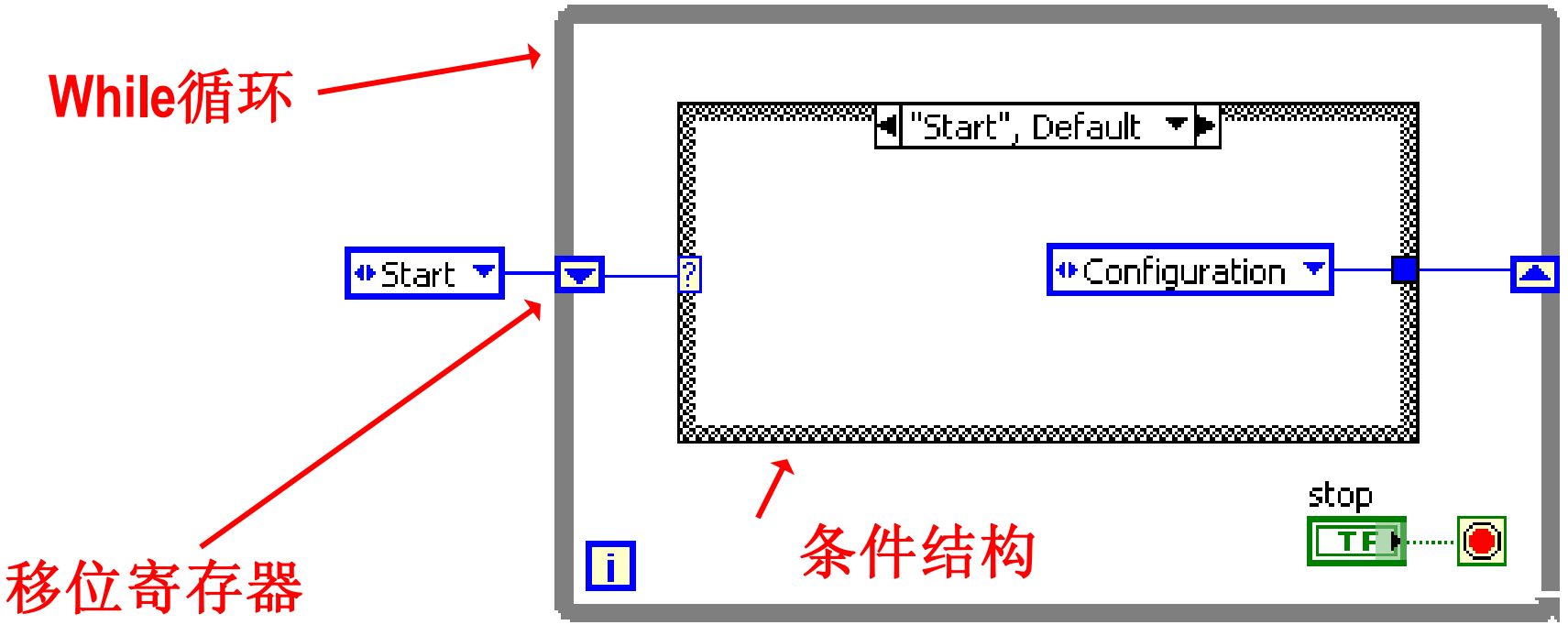
状态机模式

通常具有启动和关闭状态，但也包括其他状态。

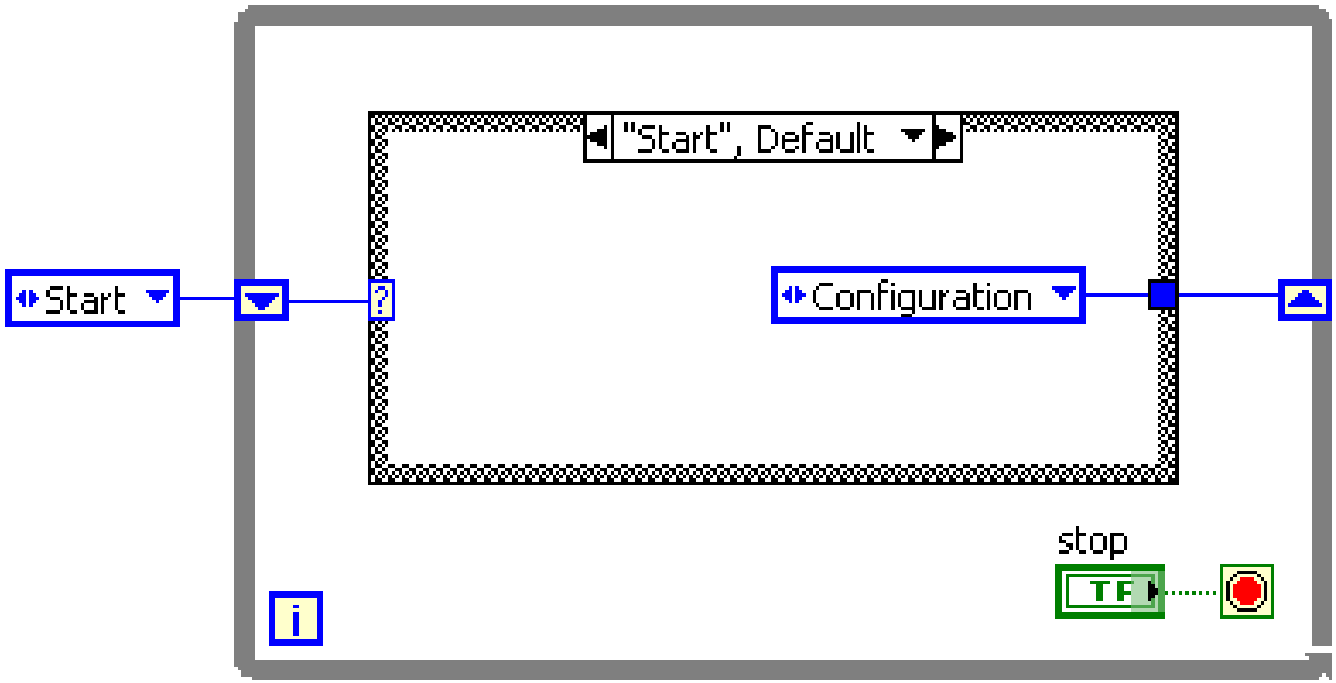


状态机 - 基础架构

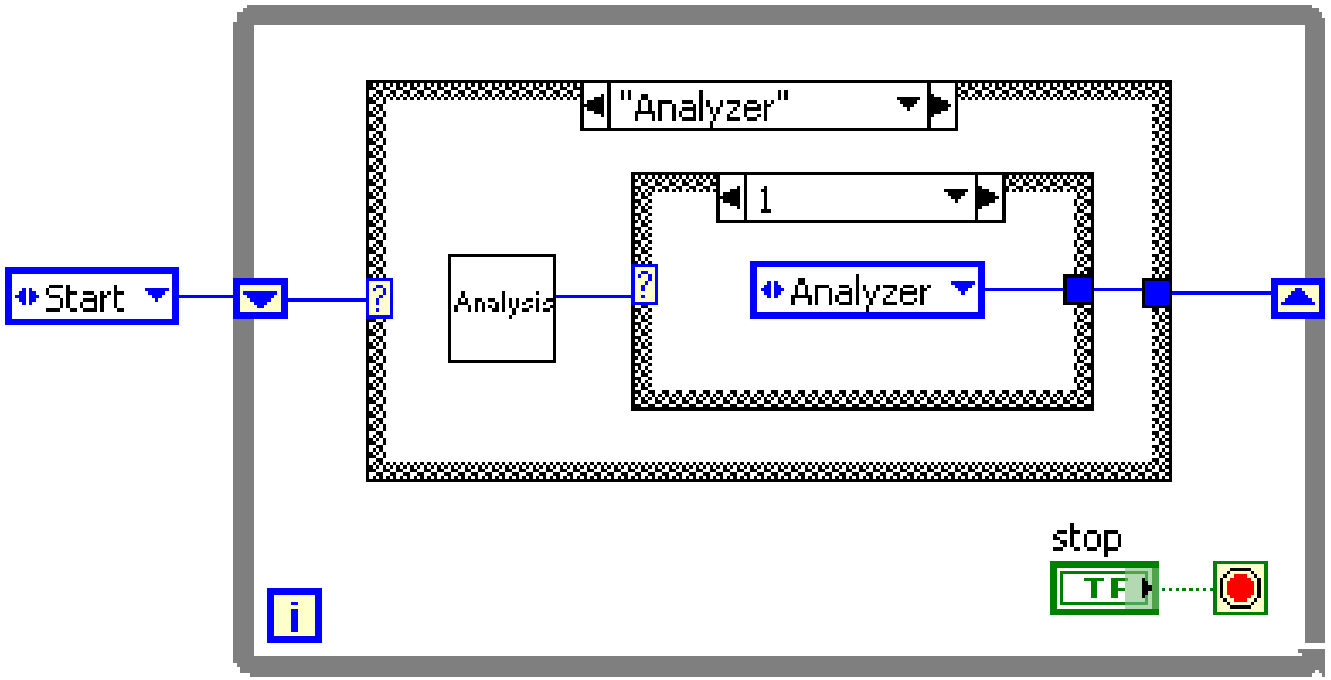
- 一个状态机包含一组状态和映射下一状态的转移函数
- 每一个状态都会导致另一个或多个状态，或者结束处理流程



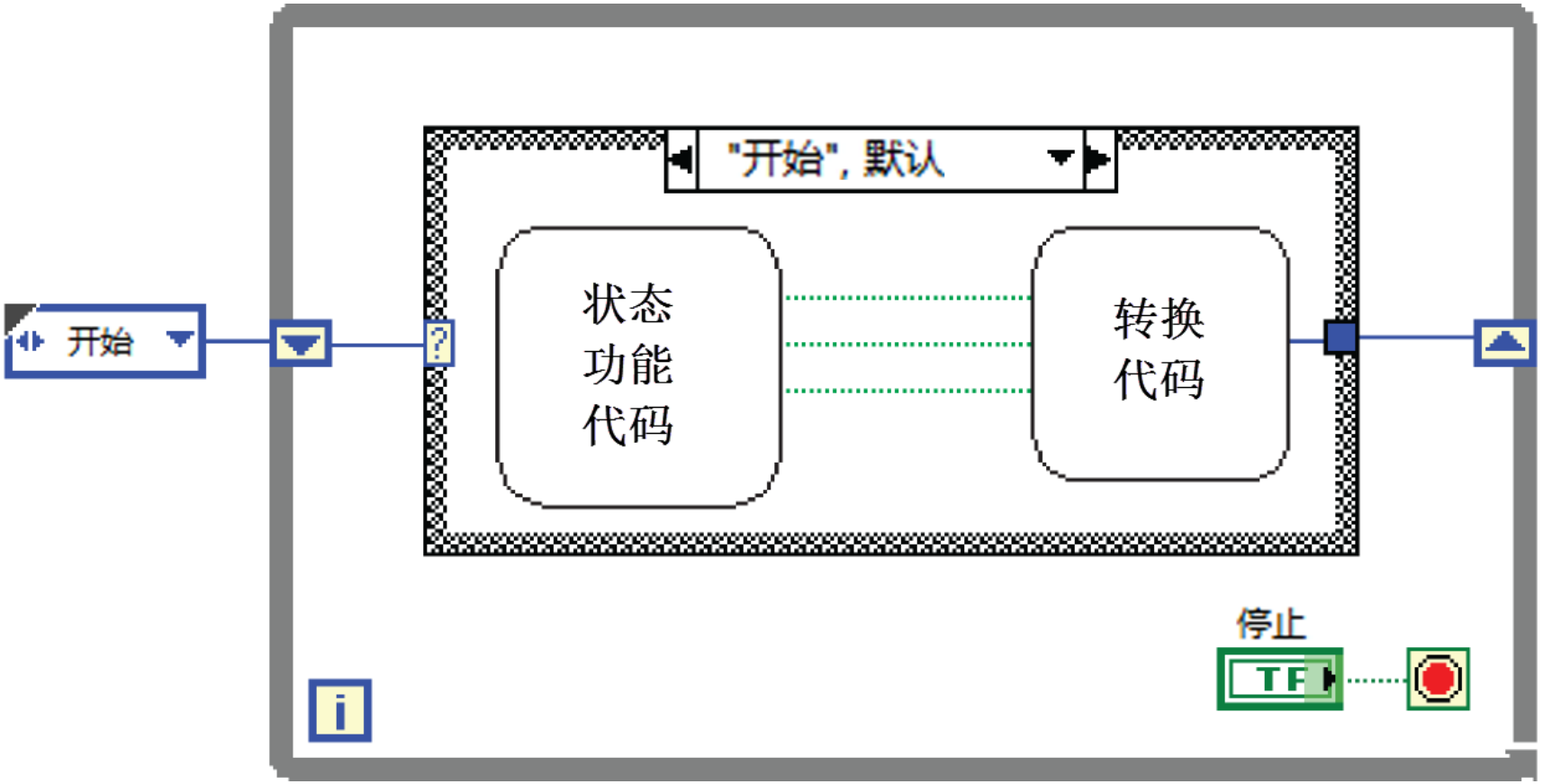
状态机 – 默认转换



C. 状态机 – 条件结构转换

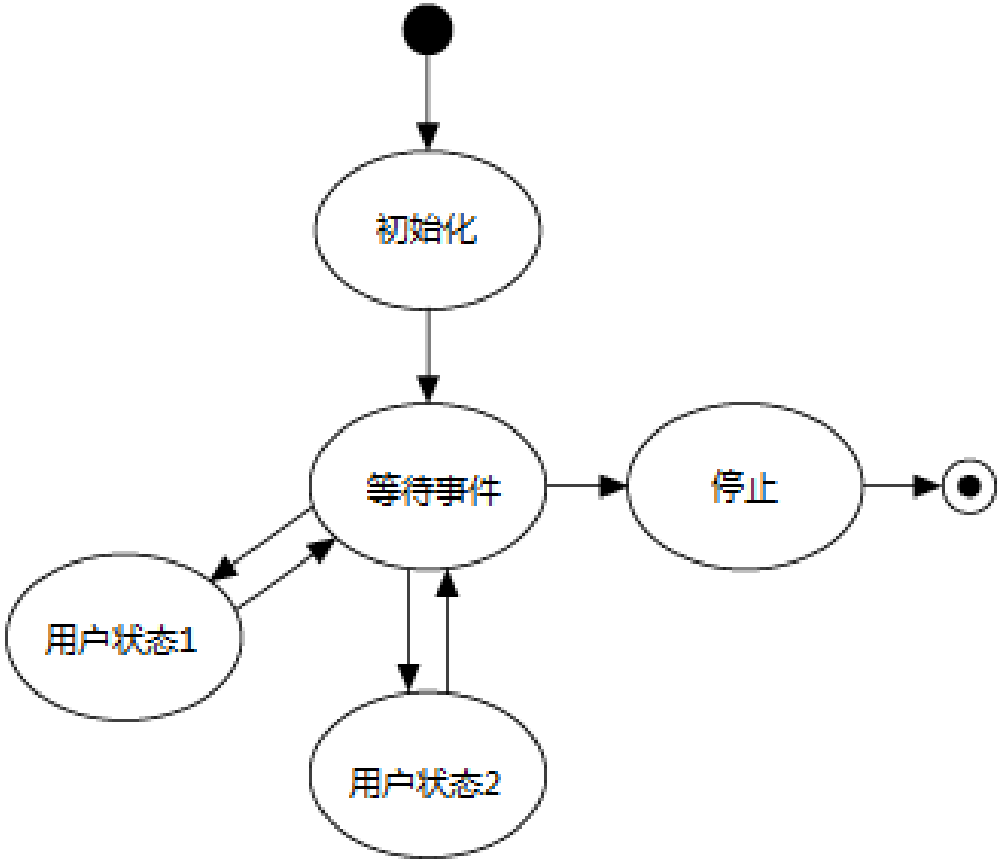


状态机框架



简单状态机模板

使用**创建项目**对话框加快实现基于事件的状态机应用程序。



C. 事件驱动编程

事件－定义

事件驱动编程－定义

轮询vs.事件结构

事件结构组成部分

配置事件结构

说明和建议

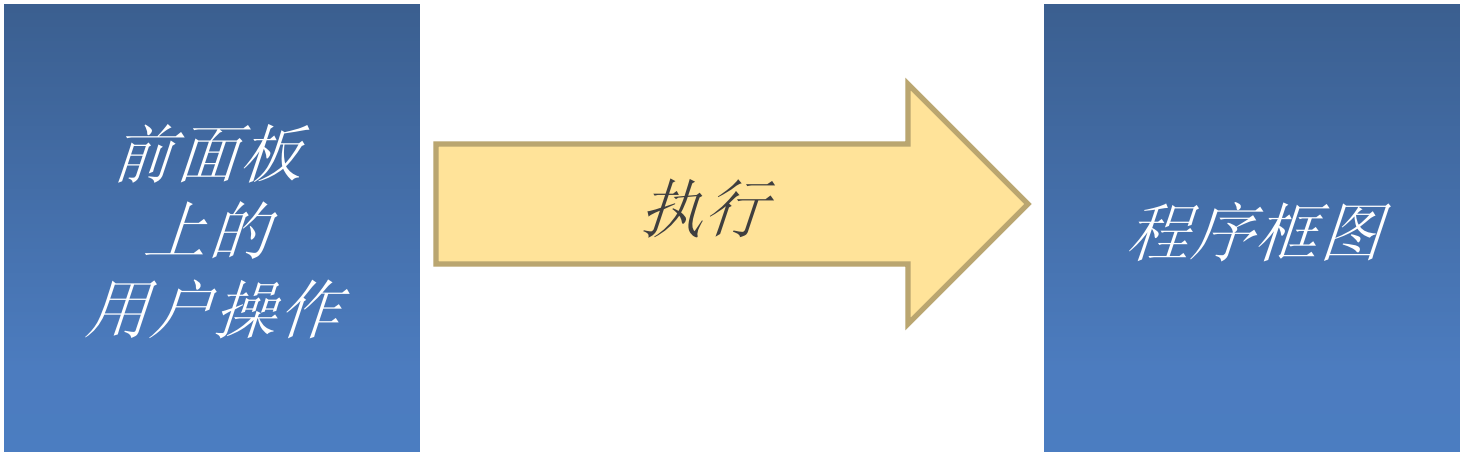
事件

事件—活动发生的异步通知

- 事件可来自用户界面、外部I/O或程序的其他部分。
- 事件对事件源执行操作。
  - 例如：前面板控件发生的值改变。

事件驱动编程

事件驱动编程——一种编程方法，其中程序执行一个或多个函数前须等待一个事件发生



轮询vs.事件结构

轮询

- 基于事件编程的方法，其中循环必须连续运行代码以检查是否发生改动。
- 轮询前面板需要大量的CPU时间。
- 改动过快时，轮询可能无法检测到改动。

事件结构

- 事件结构中的事件无需轮询前面板。
- 使用事件结构的优点包括：
  - 减少了程序对CPU的要求。
  - 简化了程序框图代码。
  - 确保了程序框图对用户的所有交互都能作出响应。

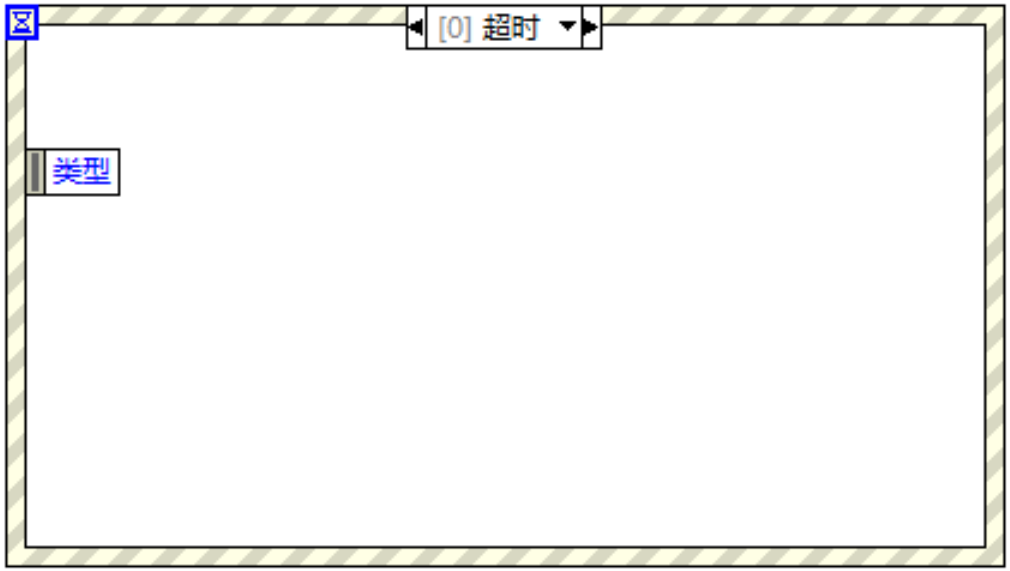
在事件驱动编程中使用事件结构

事件结构的工作原理犹如具有内置“等待通知”函数的条件结构。

使用事件结构处理用户界面（静态）事件，

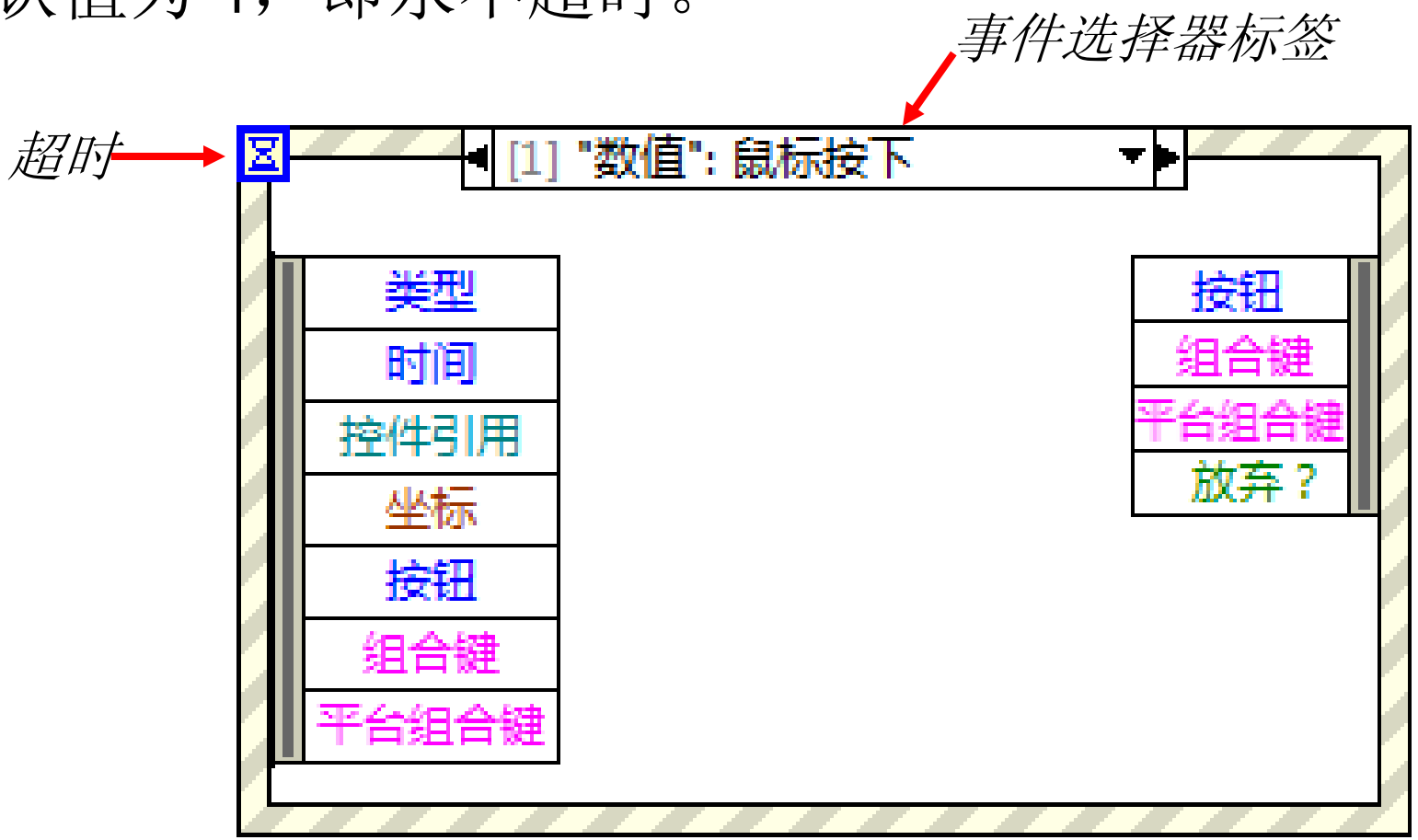
例如：

- 单击鼠标按键。
- 单击键盘按键。
- 修改数值控件的值。



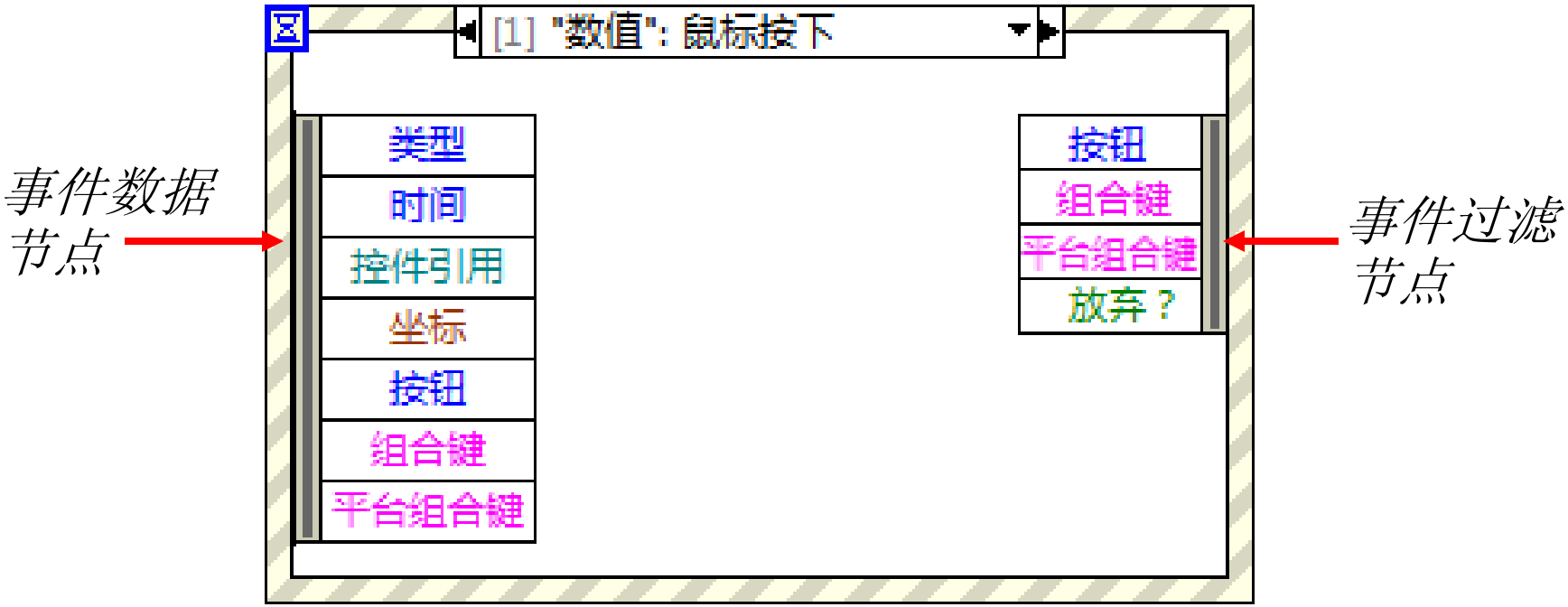
事件结构的组成部分

- 事件选择器标签－识别查看的事件分支。
- 超时－指定等待某个事件发生的时间（单位为毫秒）。默认值为-1，即永不超时。



事件结构组成部分（续）

- 事件数据节点－识别事件发生时LabVIEW提供的数据； 类似于“按名称解除捆绑”函数。
- 事件过滤节点－识别在事件数据节点中事件分支可修改的部分数据。

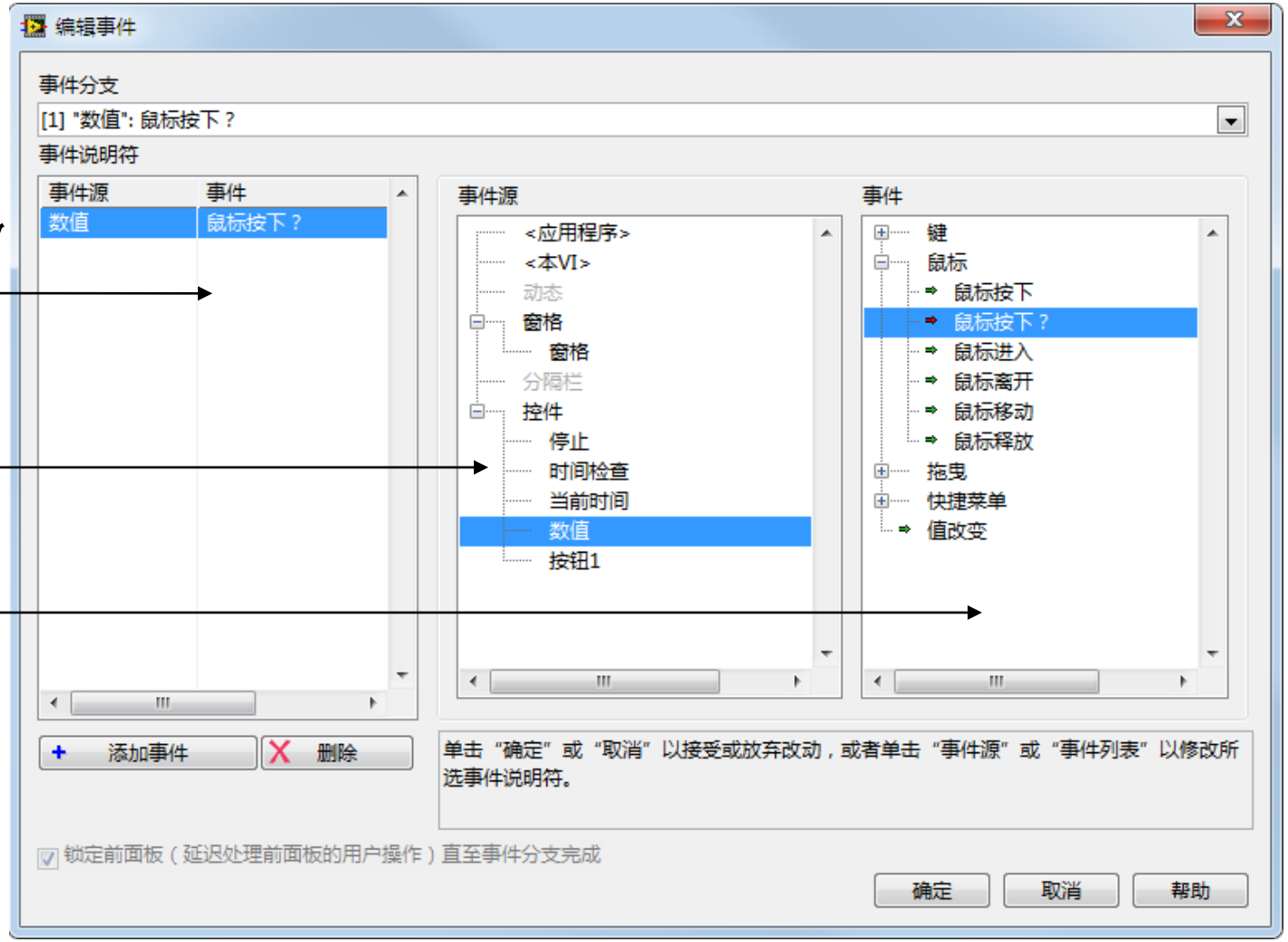


事件结构

一般情况下，事件结构位于While循环内部。

- 每次While循环，事件结构仅处理一个事件。
- 无事件发生时事件结构将进入休眠状态。

编辑事件对话框



已配置的事件

事件源

事件

说明和建议

- 避免在循环外使用事件结构。
- 一个循环中仅放置一个事件结构。
- 避免为同一个事件配置两个事件结构。
- 使用值改变事件检测值的改变。
- 保持事件简洁快速地处理代码。
- 将布尔控件接线端置于事件分支的内部，以保证触发操作正常运行。

D. 多循环设计模式

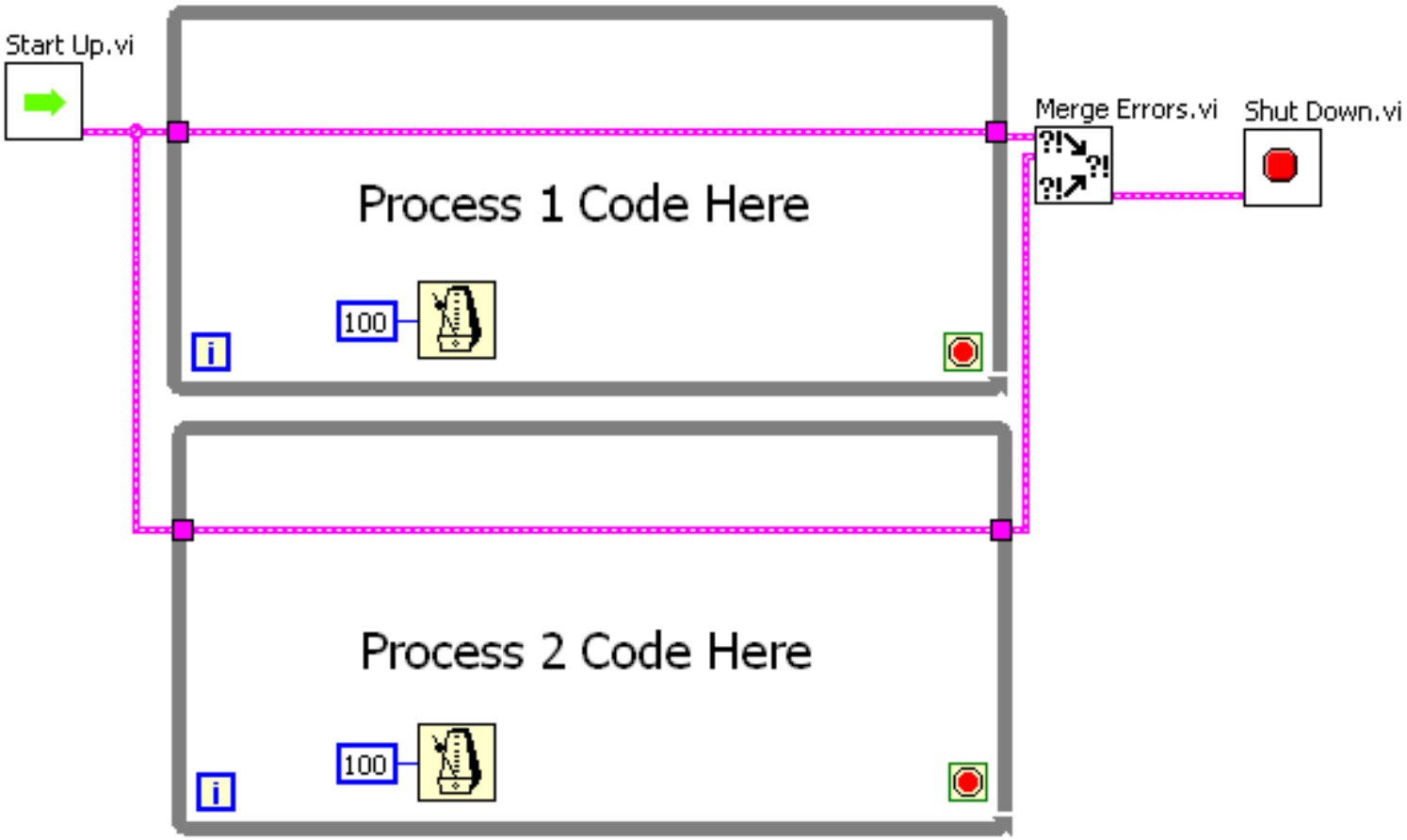
并行

主从

生产者消费者设计模式

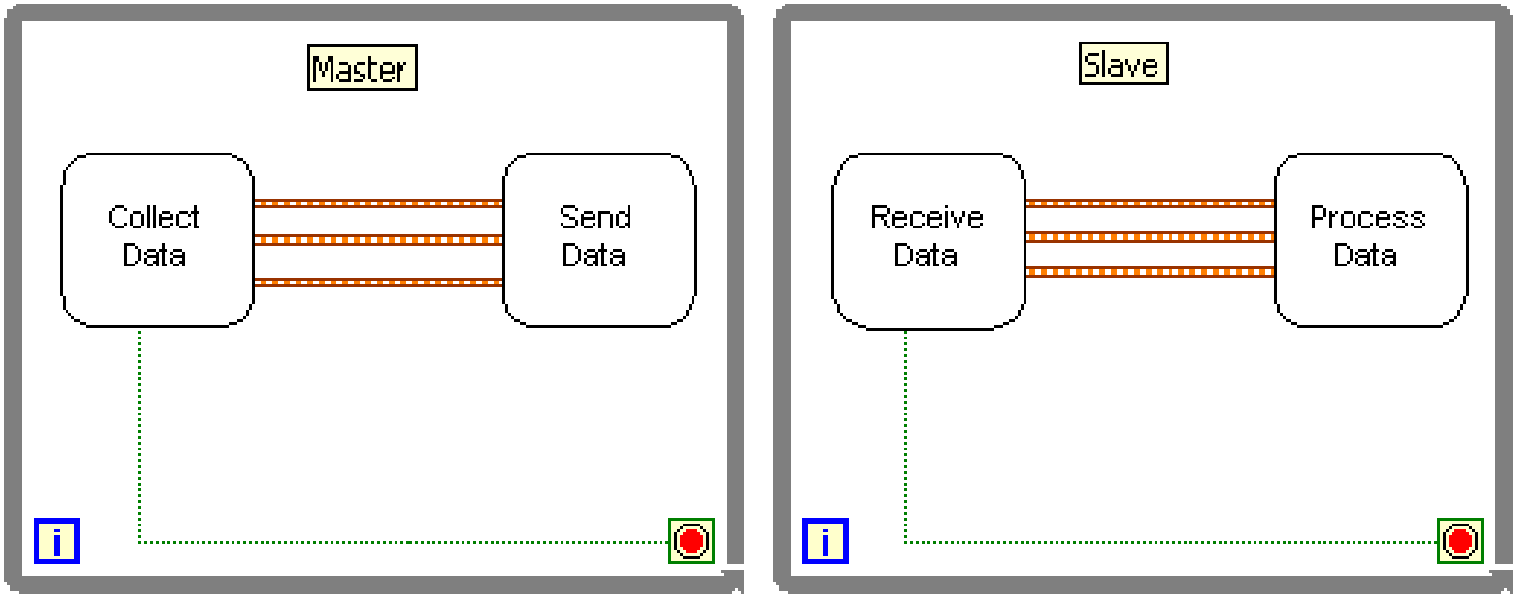
C. 多循环架构

•并行循环


Start Up.vi
Process 1 Code Here
100
i
Merge Errors.vi
Shut Down.vi
Process 2 Code Here
100
i

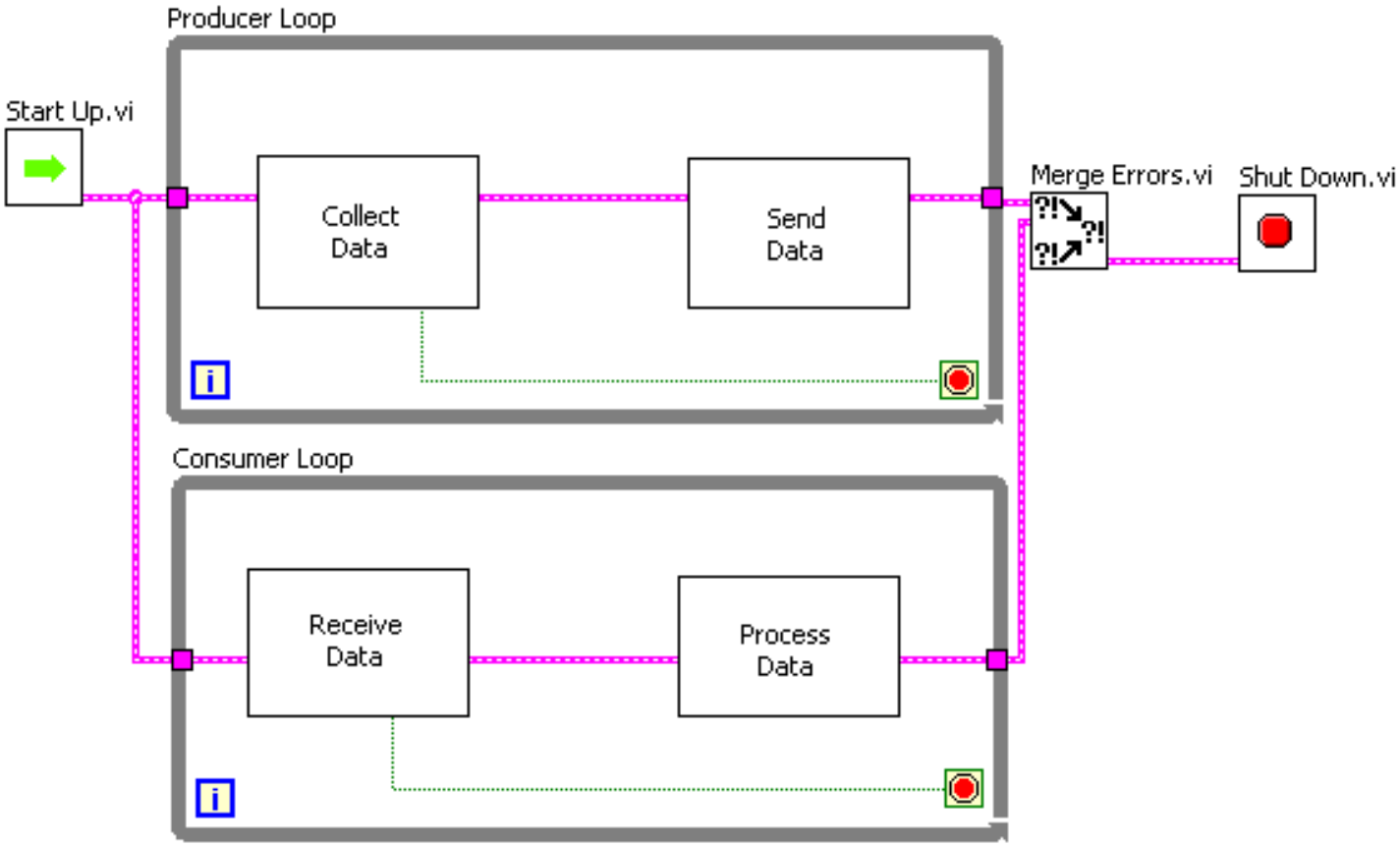
C. 多循环架构

主/从模式


Master
Collect Data
Send Data
Slave
Receive Data
Process Data

C. 多循环架构

- 生产者/消费者模式



E. 为设计模式设置定时

执行定时

软件控制定时

- 同步超时
- 获取日期/时间（秒）

为设计模式设置定时

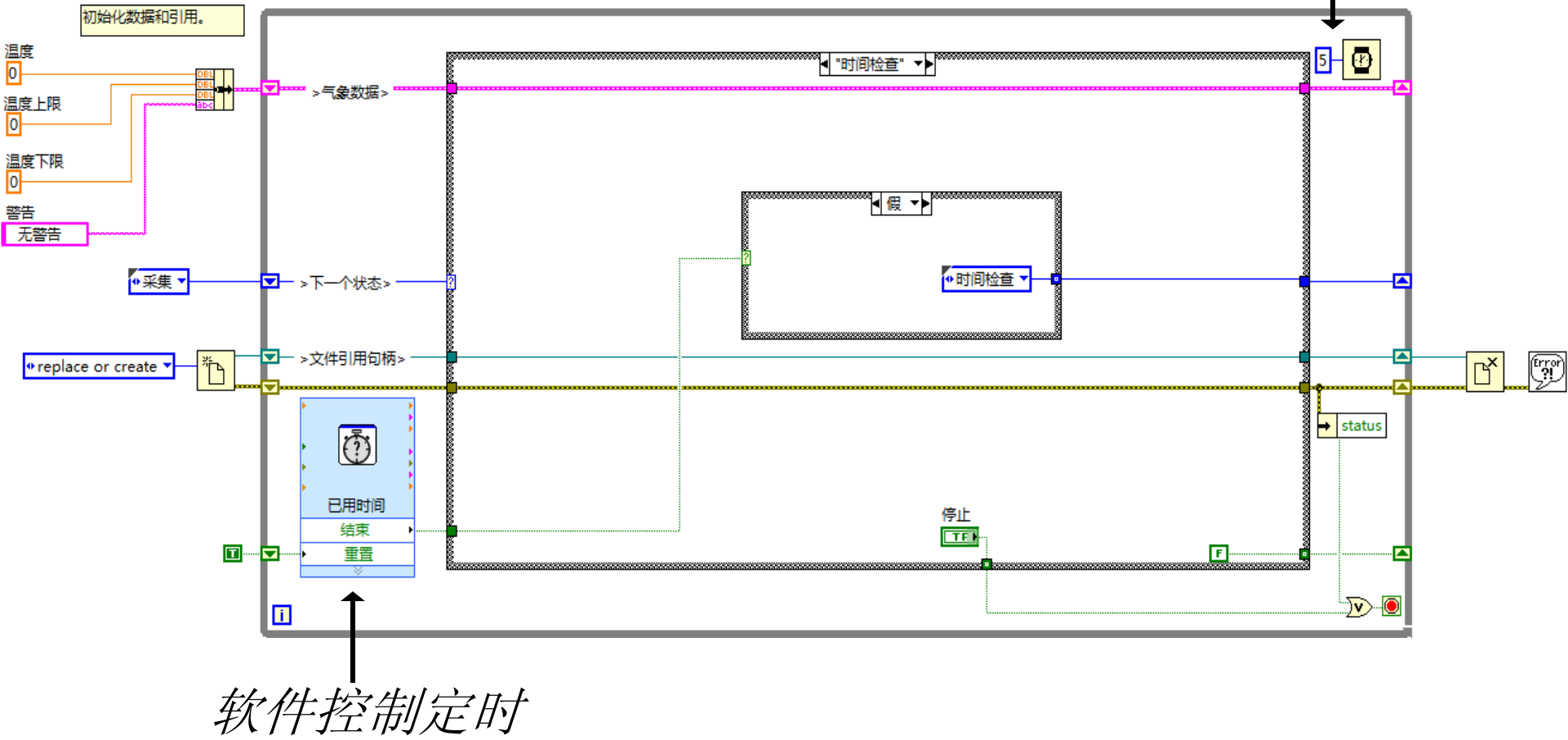
执行定时

- 为设计模式提供一个函数，使处理器有时间处理其他任务。

软件控制定时

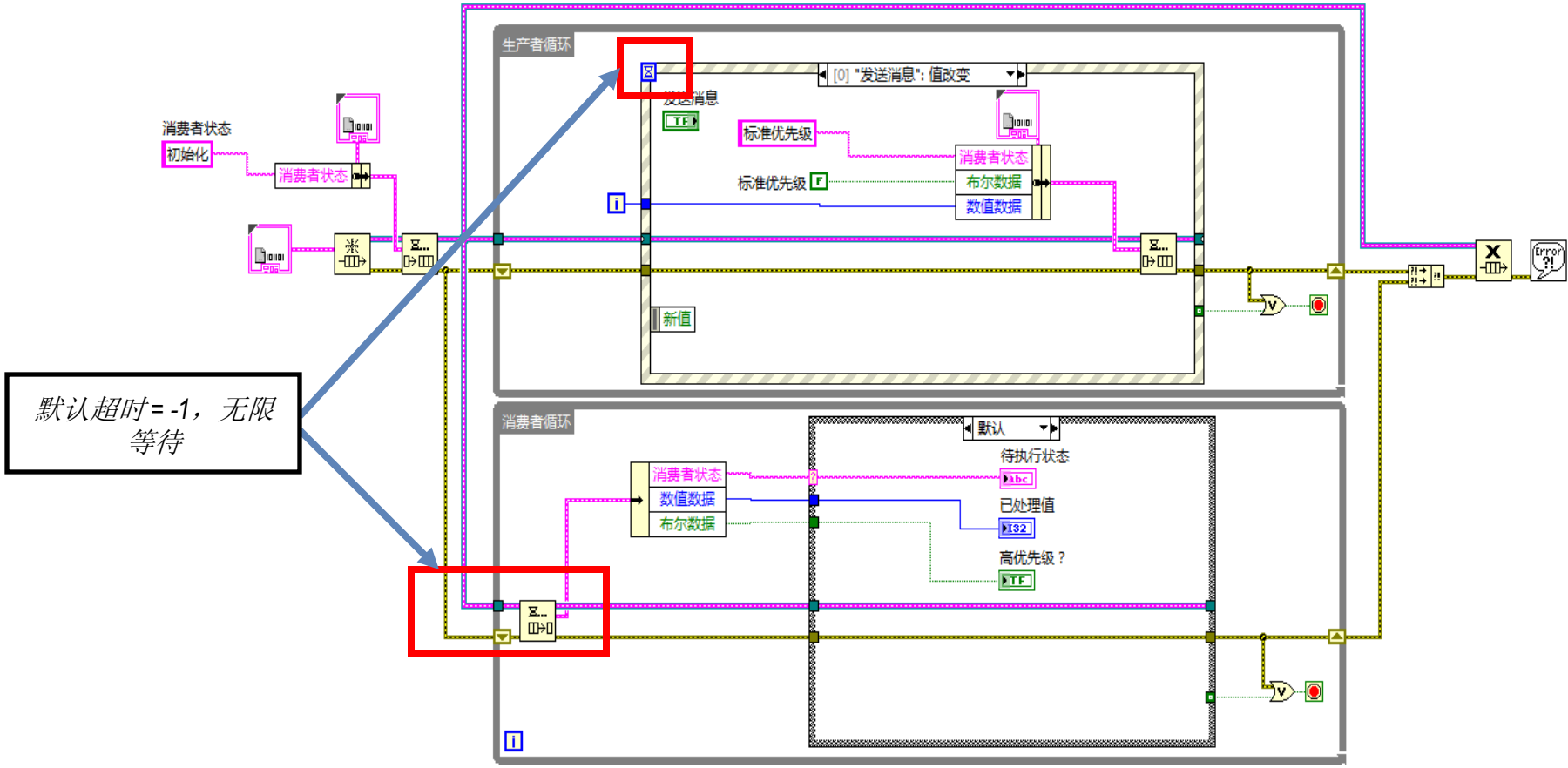
- 为实际的操作定时，以在一定的时间段内执行。
- 控制循环的执行频率。

为设计模式设置定时

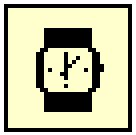
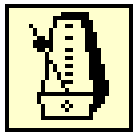

执行定时
初始化数据和引用。
温度
0
温度上限
0
温度下限
0
警告
无警告
采集
>下一个状态>
replace or create
>文件引用句柄>
>气象数据>
"时间检查"
5
假
时间检查
status
已用时间
结束
重置
停止
TF
F
软件控制定时

执行定时

定时基于事件发生的设计模式无需执行定时。



软件控制定时

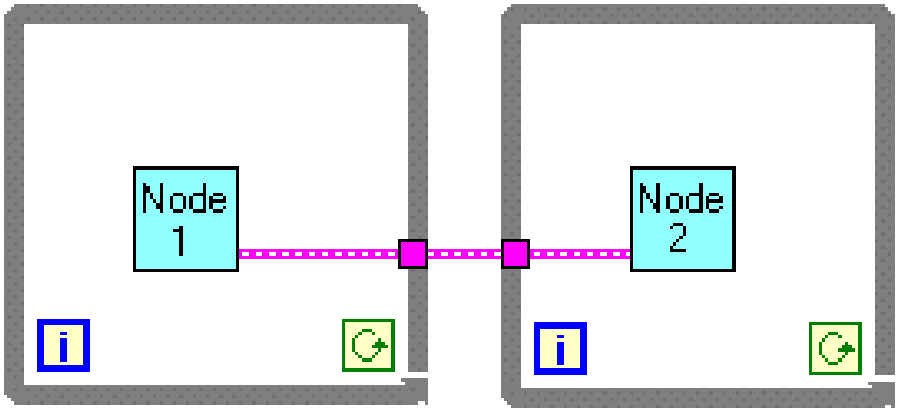
- 软件控制定时必须让设计模式连续运行，无需停止。
- 相较于软件控制定时，下列函数更适合执行定时。
  - 等待(ms) 
  - 等待下一个整数倍毫秒 

第2部分
在多个循环之间通信

A. 变量
B. 功能全局变量
C. 竞争状态
D. 队列

A. 变量

- 用连线不能在并行循环间传递数据
- 变量可以在不使用连线连接两个地方的条件下把数据从一个地方传递到另一个地方，因此不需要使用标准数据流



A. 变量

•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

•变量可以是下面任一种类型：

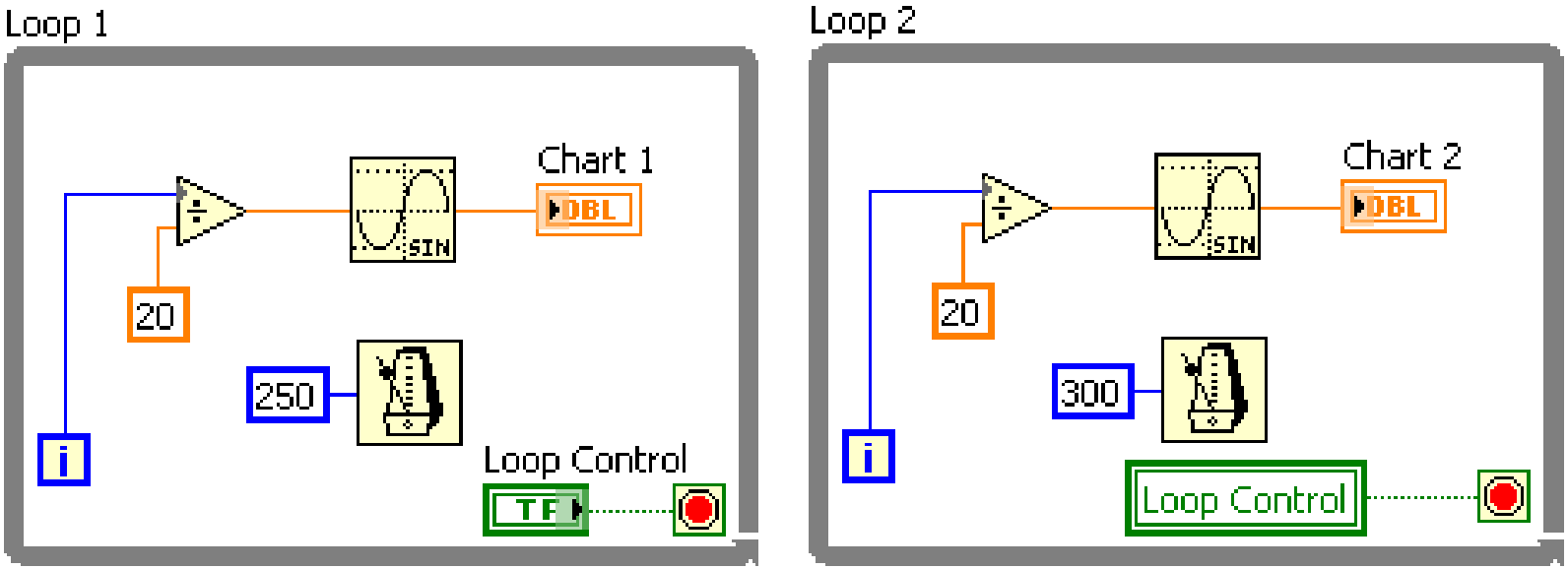
– 局部变量：将数据存储在前面板的输入控件和显示控件中

– 全局变量：将数据存储在多个VI可以访问的特殊库中

– 功能性全局变量：将数据存储在While循环的移位寄存器中

A. 变量 – 在单个VI中使用变量

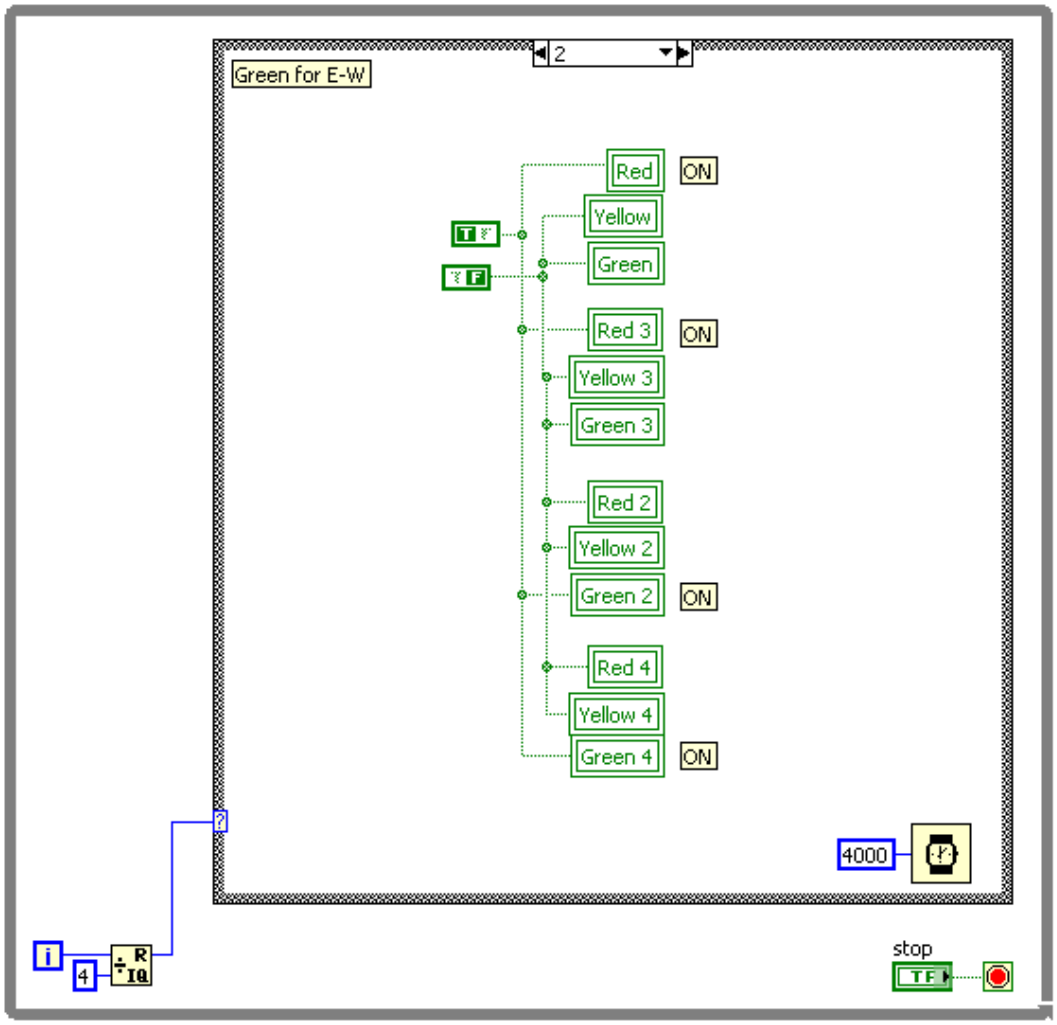
•使用局部变量在单个VI中传递数据



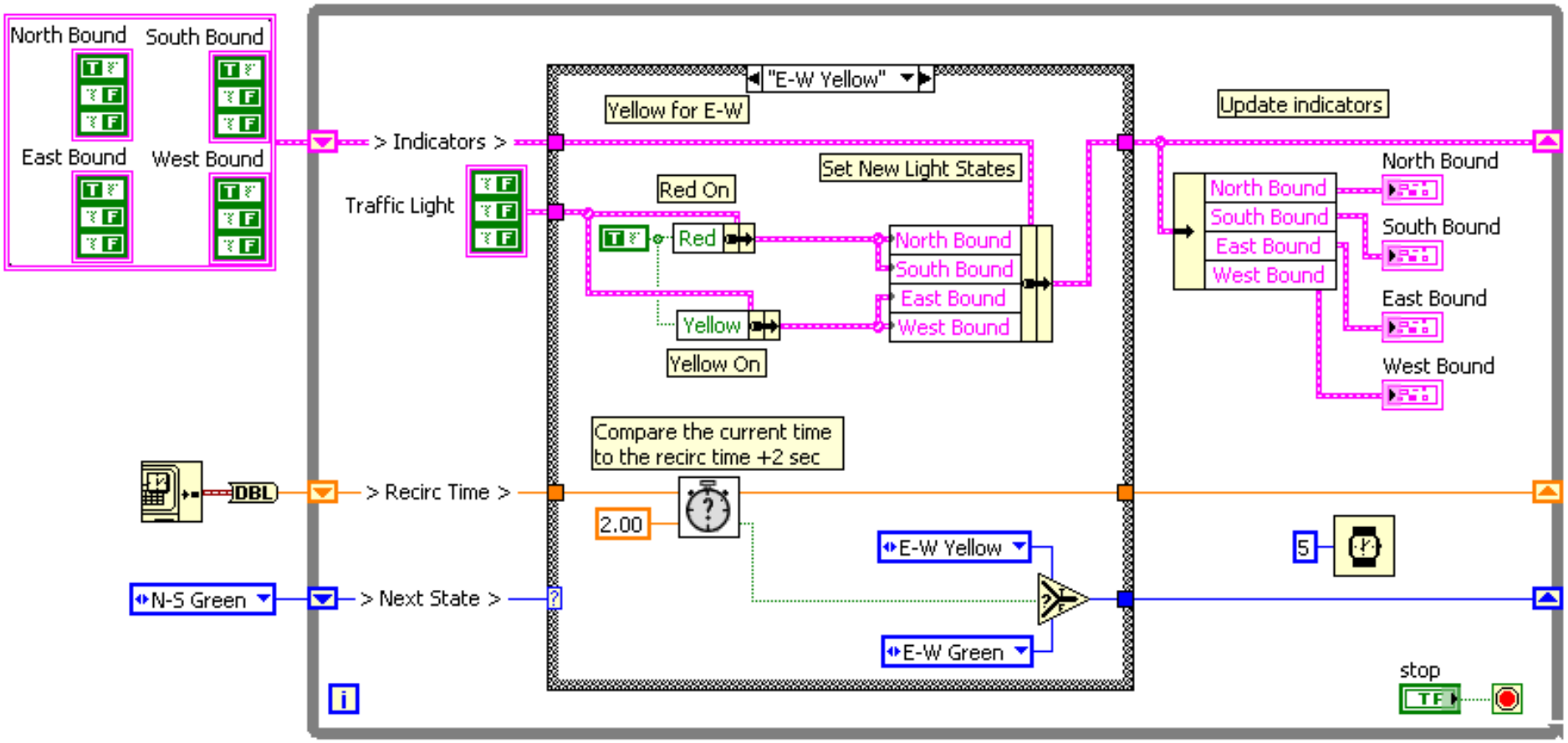
A. 变量 – 在多个VI间使用

- 使用一个全局变量或单进程共享变量在多个VI之间共享数据
  - 在同一台计算机上的多个VI之间共享数据时，请使用全局变量，尤其是在没有使用项目文件的情况下
  - 如果以后需要在多台计算机上的多个VI之间共享变量的信息，请使用单进程共享变量

A. 变量 – 谨慎使用

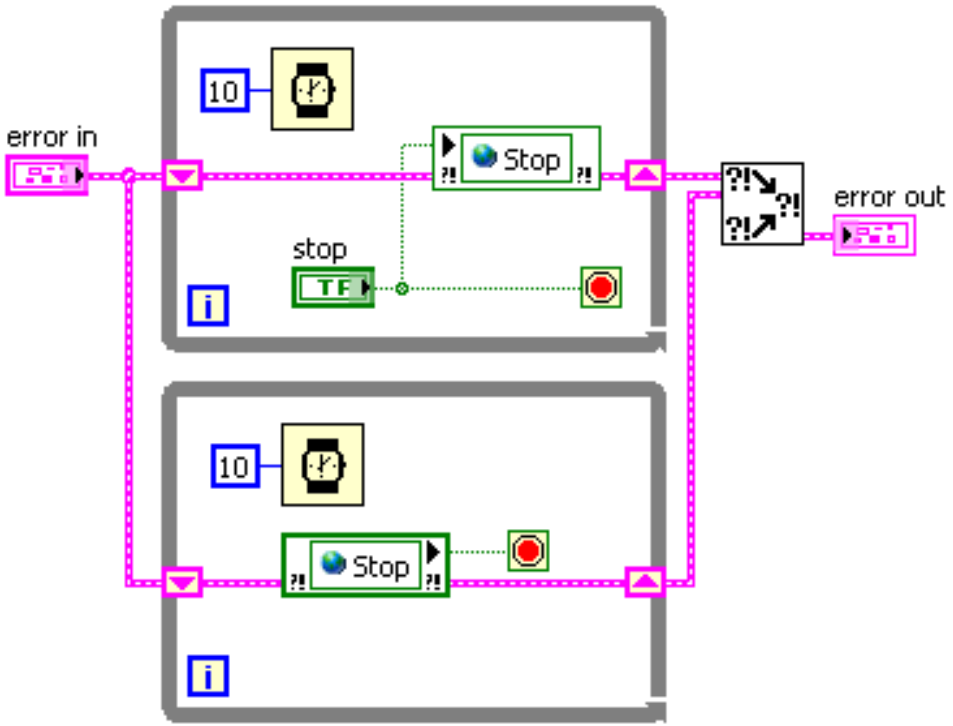

Green for E-W
2
Red
ON
Yellow
Green
Red 3
ON
Yellow 3
Green 3
Red 2
Yellow 2
Green 2
ON
Red 4
Yellow 4
Green 4
ON
4000
stop
TF
i
4

A. 变量 – 谨慎使用

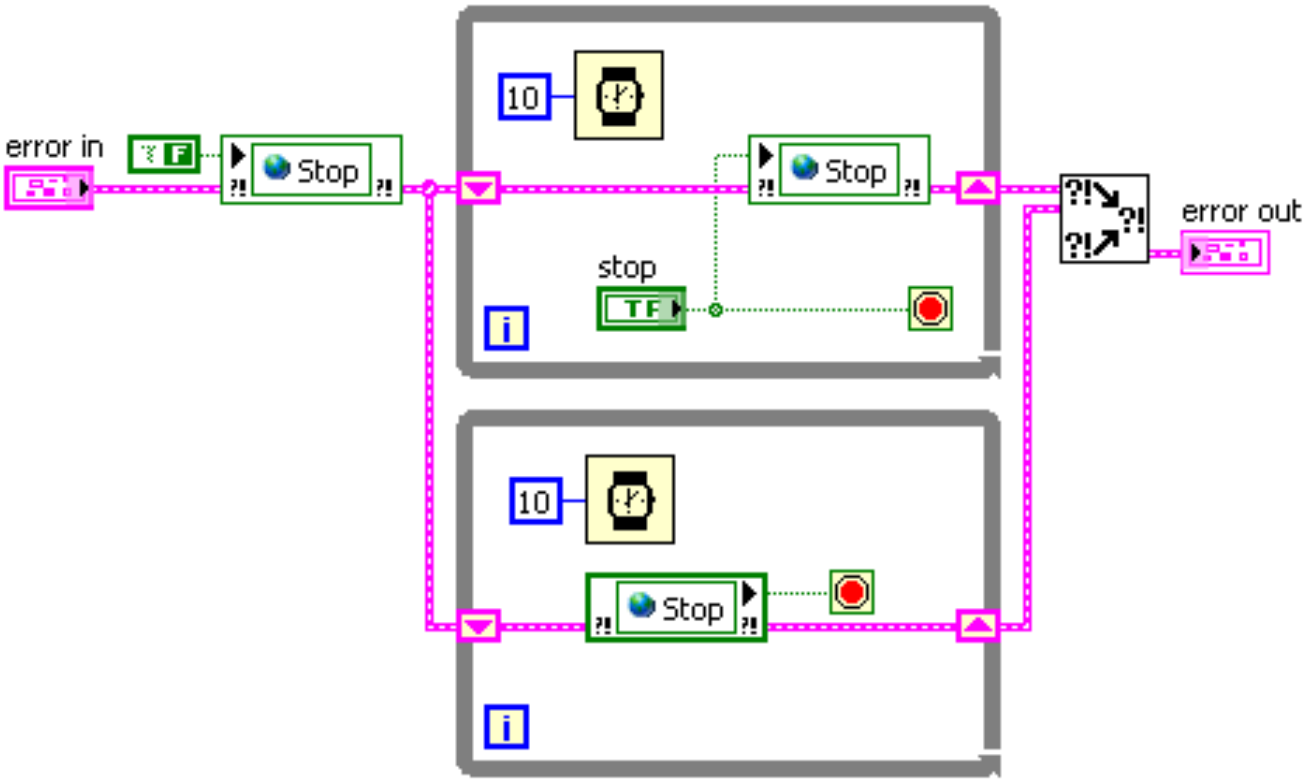

North Bound
South Bound
East Bound
West Bound
> Indicators >
Traffic Light
"E-W Yellow"
Yellow for E-W
Set New Light States
Red On
Red
Yellow
Yellow On
North Bound
South Bound
East Bound
West Bound
Update indicators
North Bound
South Bound
East Bound
West Bound
DBL
> Recirc Time >
Compare the current time
to the recirc time +2 sec
2.00
N-S Green
> Next State >
E-W Yellow
E-W Green
5
stop
TF

A. 变量 – 初始化

- 确保VI运行前变量含有已知的数据值
- 如在VI第一次
读取变量之前，
没有将变量初始化，
则该变量含有的是
  - 相关前面板对象的
  默认值

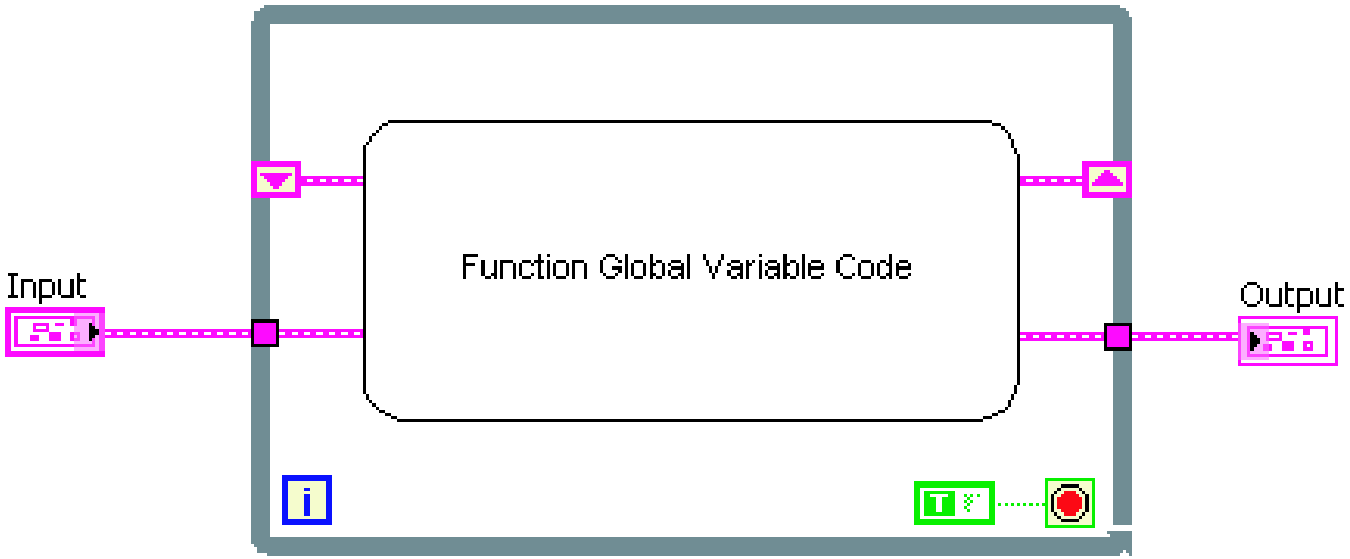


A. 变量 – 初始化



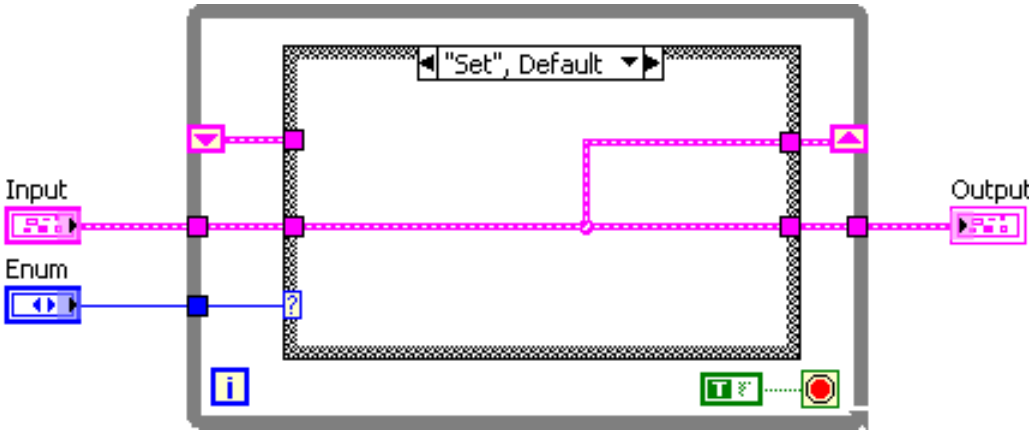
B. 功能全局变量

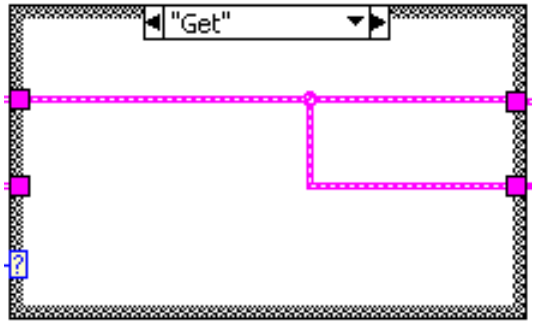
- 一个功能全局变量的一般形式包括一个未进行初始化的移位寄存器和单个迭代循环（For循环或While循环）



B. 功能全局变量

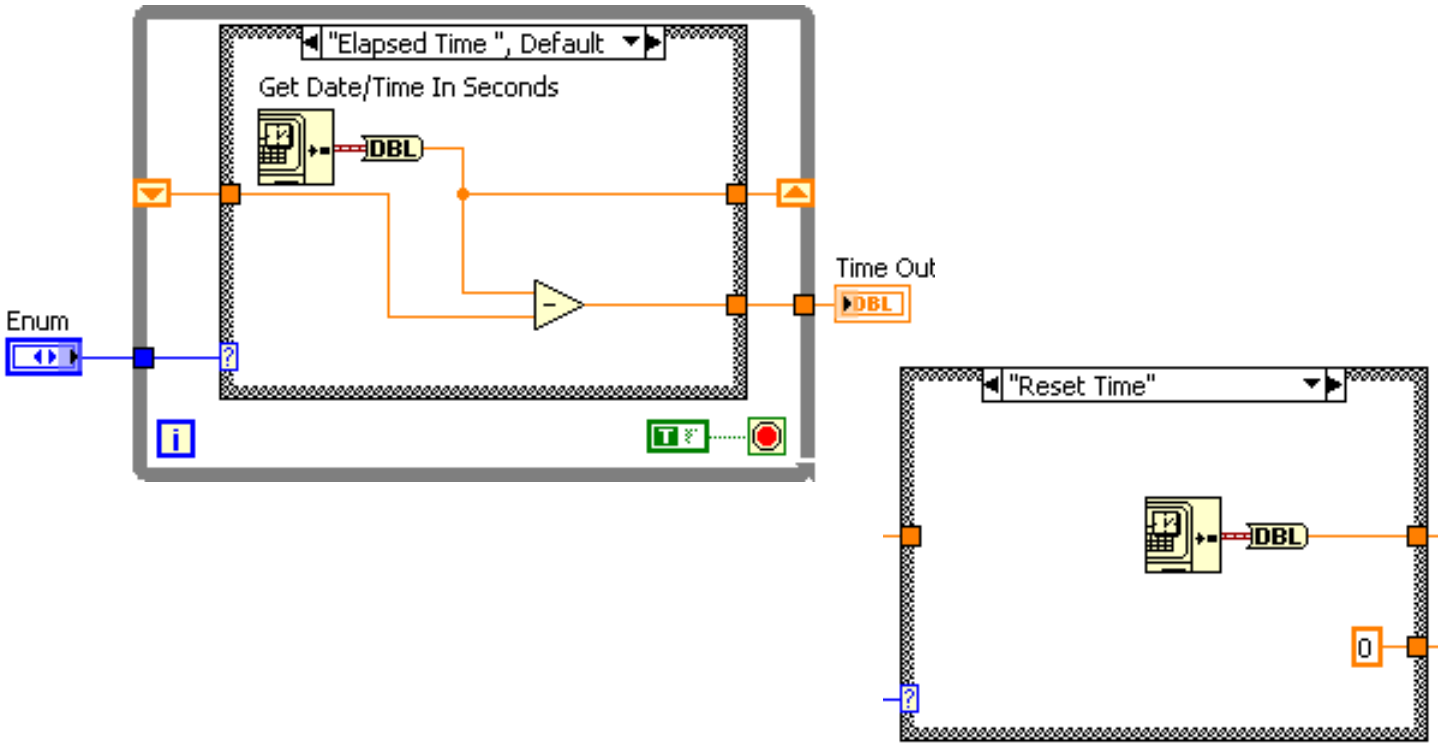
- 一个功能全局变量通常有一个**action**输入参数来指定VI执行的任务
- 这个VI在While循环中使用了未进行初始化的移位寄存器来保持操作结果





B. 功能全局变量 – 定时

- 适用于执行自定义的已用时间测量



C. 竞争状态

- 在竞争状态中，事件的定时或任务的安排可能会无意中影响一个输入或数据值
- 对于那些并行执行多个任务并在任务之间共享数据的程序，竞争状态是一种常见问题

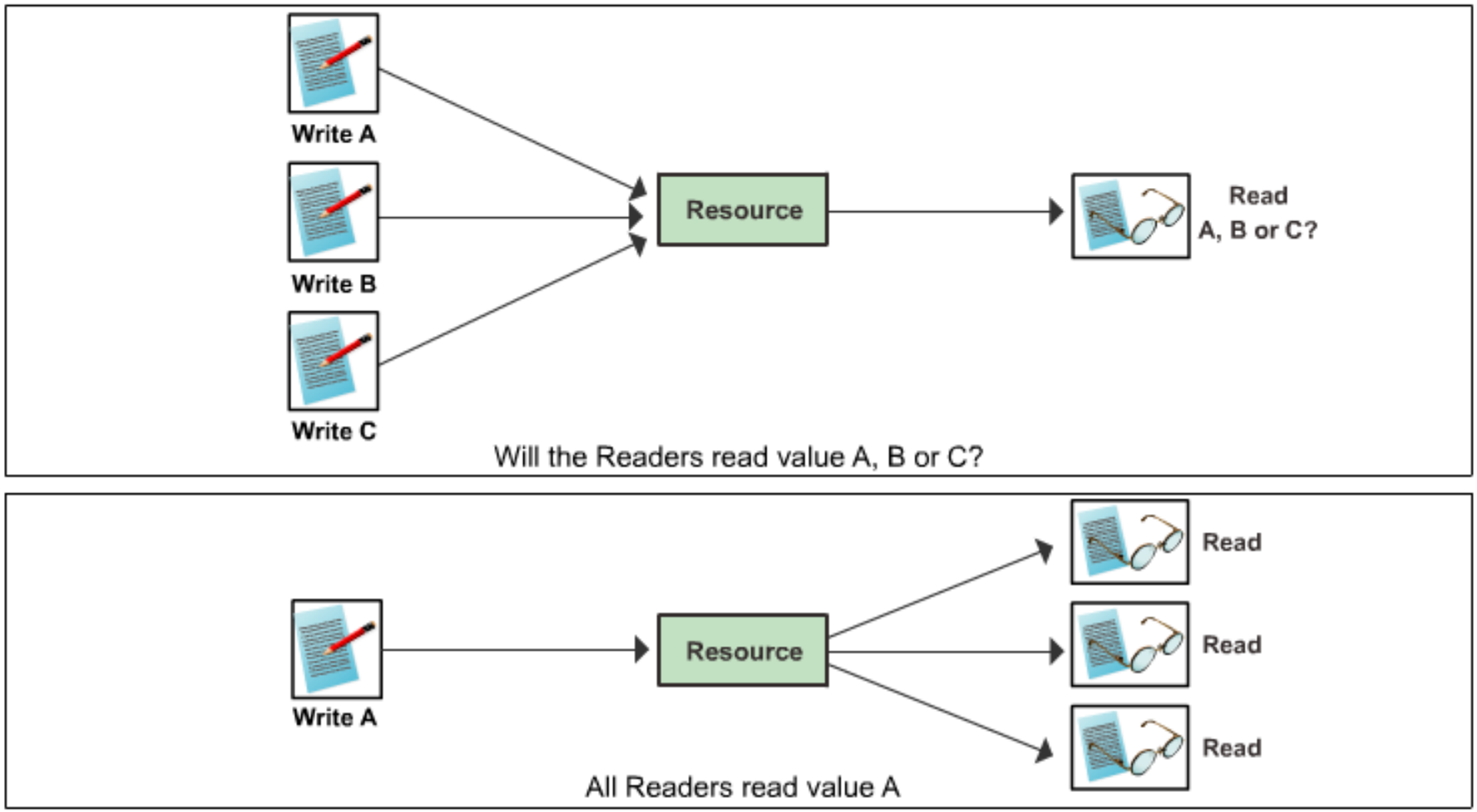
竞争状态

观看讲师演示竞争状态。

C. 竞争状态

- 竞争状态不容易识别和调试
- 很多情况下，带有竞争状态的代码会在数千次测试中返回相同的结果，但仍然可能会返回一个不同的结果
- 避免竞争状态的方法：
  - 控制共享资源
  - 合理安排指令顺序
  - 识别和保护代码中的重要部分
  - 减少变量的使用

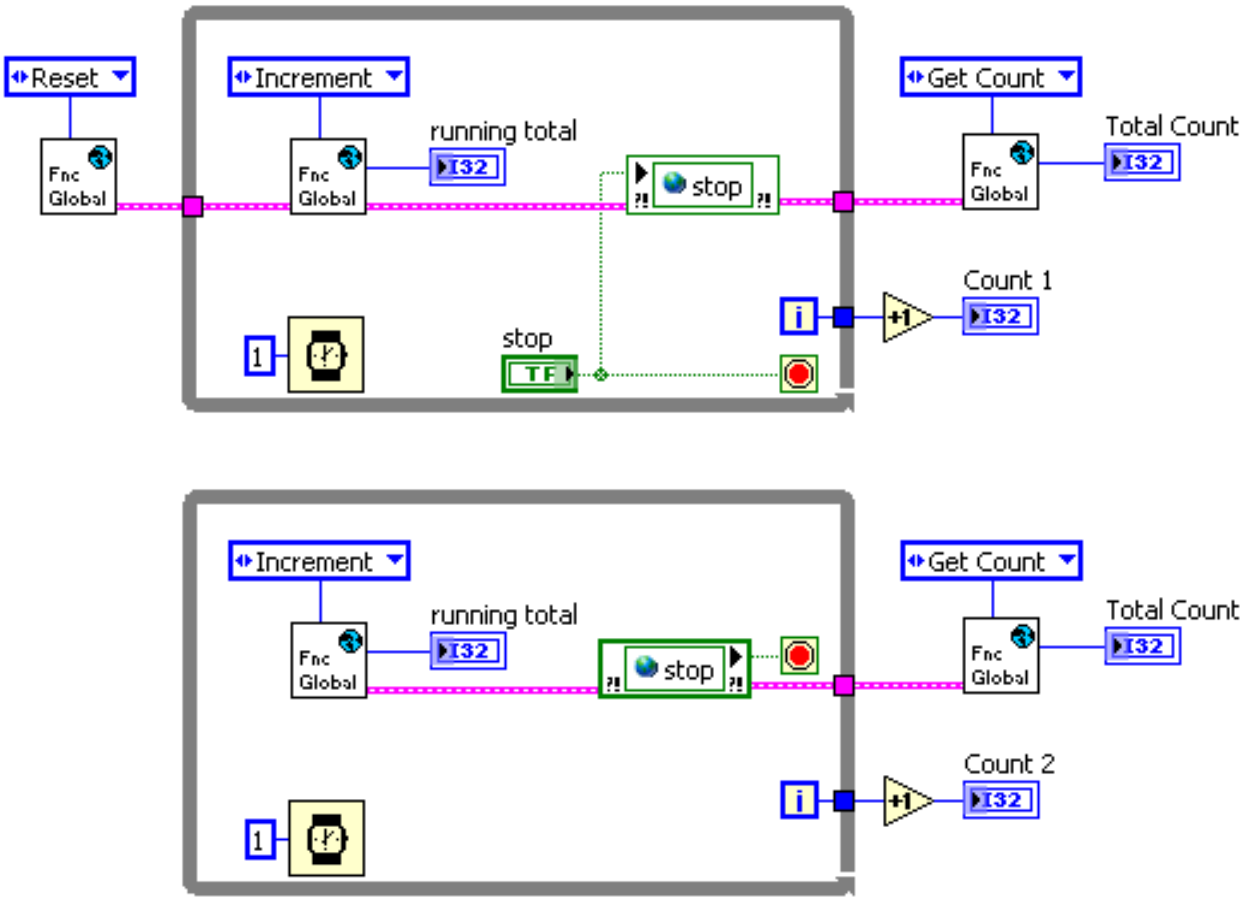
C. 竞争状态 – 共享资源


Write A
Write B
Write C
Resource
Read
A, B or C?
Will the Readers read value A, B or C?
Write A
Resource
Read
Read
Read
All Readers read value A

C. 竞争状态 – 重要代码

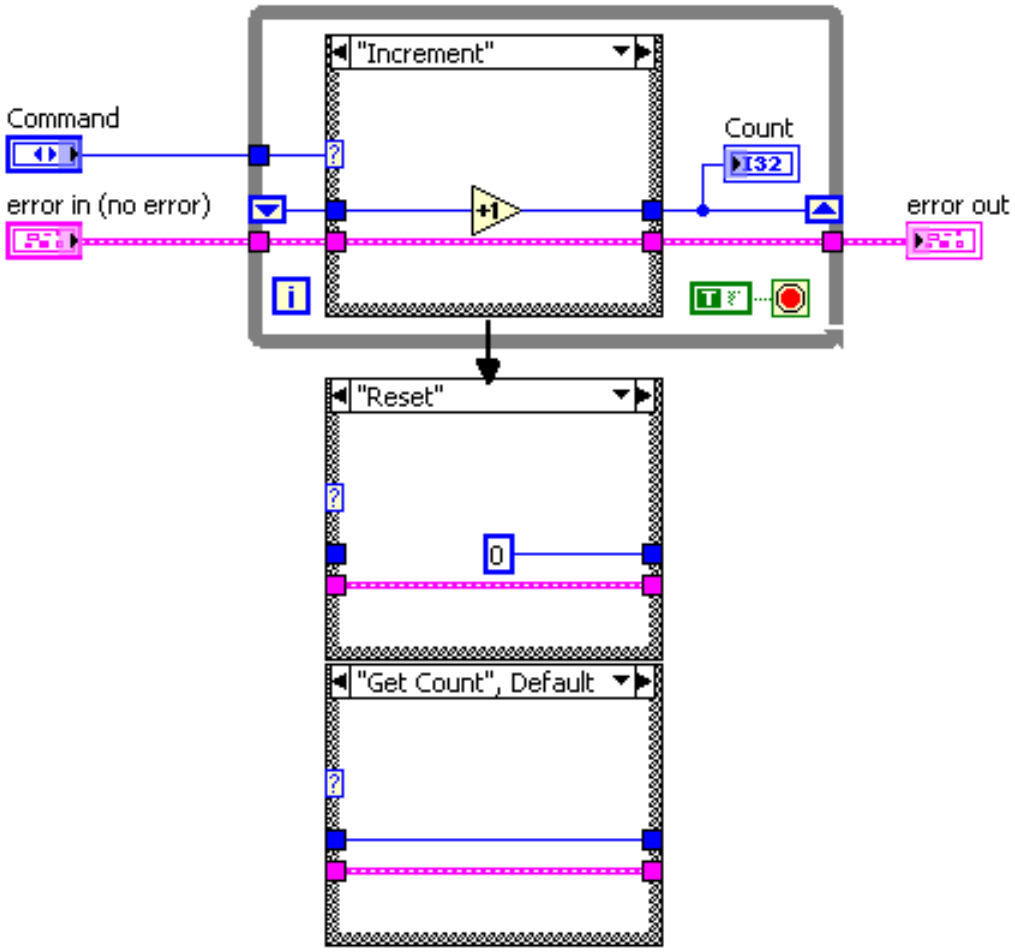
- 代码的重要部分是那些在运行时如果改变一些共享变量就会导致不一致行为的代码
- 如果一个循环在另一个循环执行重要部分代码时中断了该循环，那么竞争状态就会发生
- 为了消除竞争状态，可通过以下对象识别和保护重要部分代码：
  - 功能全局变量
  - 信号量

C. 竞争状态 – 重要代码

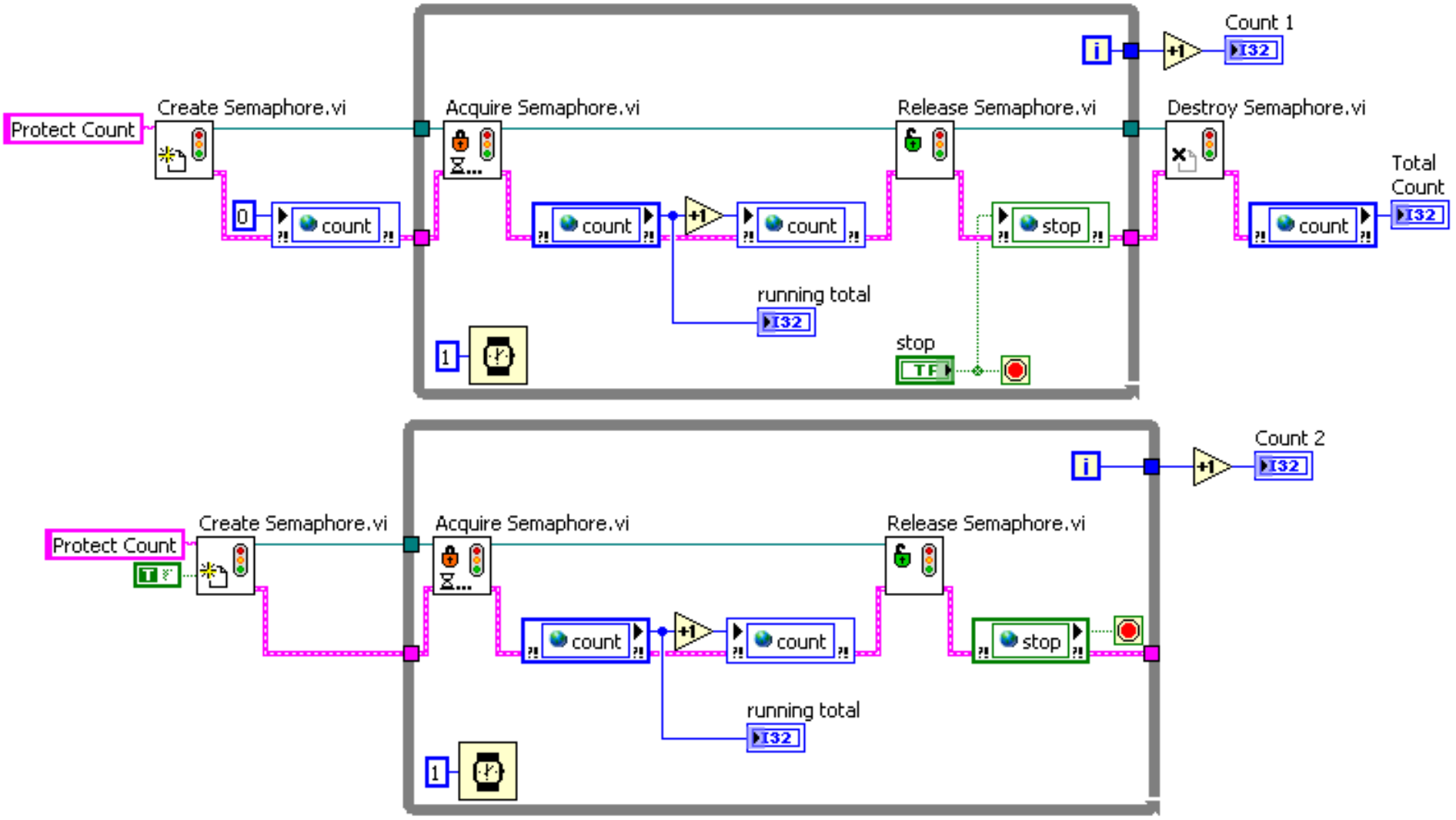

Reset
Increment
Get Count
Fnc Global
running total
I32
stop
Total Count
Count 1
1
stop
TF
i
+1
Count 2

C. 竞争状态 – 重要代码

•功能性全局变量可用于保护重要代码：

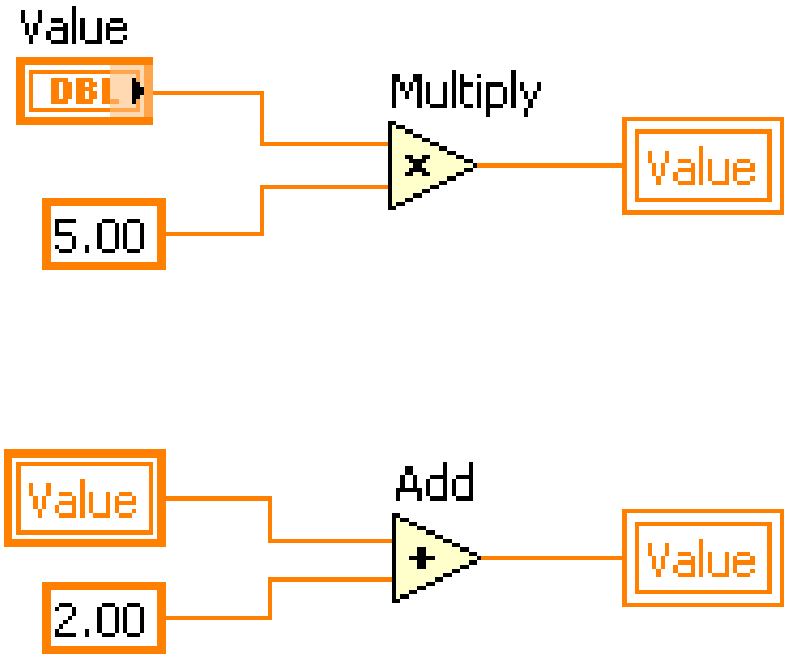


C. 竞争状态 – 重要代码


Protect Count
Create Semaphore.vi
Acquire Semaphore.vi
Release Semaphore.vi
Destroy Semaphore.vi
Count 1
Total Count
count
stop
running total
I32
TF
Protect Count
Create Semaphore.vi
Acquire Semaphore.vi
Release Semaphore.vi
Count 2
count
stop
running total
I32

C. 竞争状态 – 排序

- 最后的值是多少？
- 四种可能的结果：
  - Value = (Value * 5) +2
  - Value = (Value + 2) * 5
  - Value = Value * 5
  - Value = Value +2



B. 队列

队列

队列操作

生产者 / 消费者（数据）设计模式

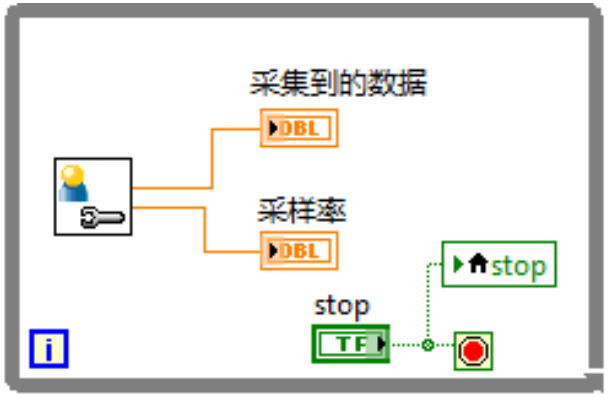
队列

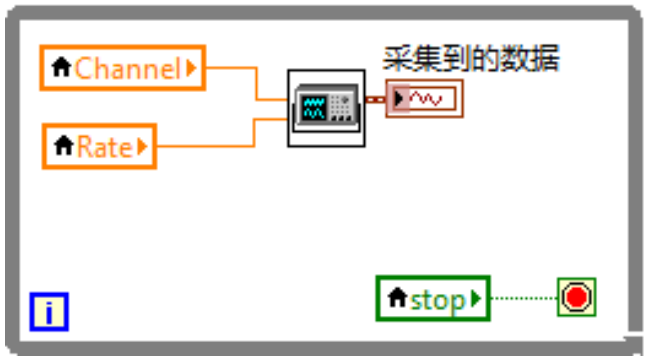
- 用于并行循环之间的数据通信。
- 存储多组数据（即缓冲数据）。
- 默认情况下，以FIFO（先进先出）方式执行。
- 可保存任何类型的数据。

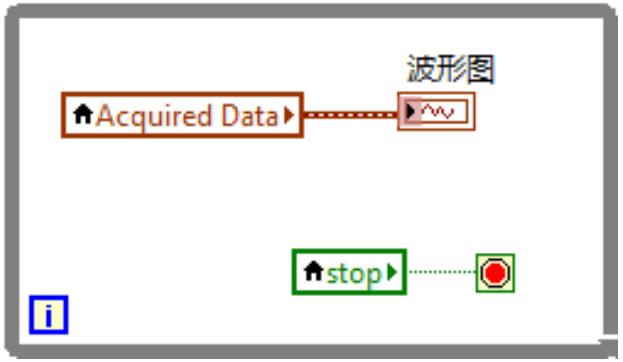
变量的不足之处

使用变量在循环之间进行通信的不足之处包括：

- 可能读取重复数据。
- 可能丢失数据。
- 用户可创建读取－修改－写入竞争状态。





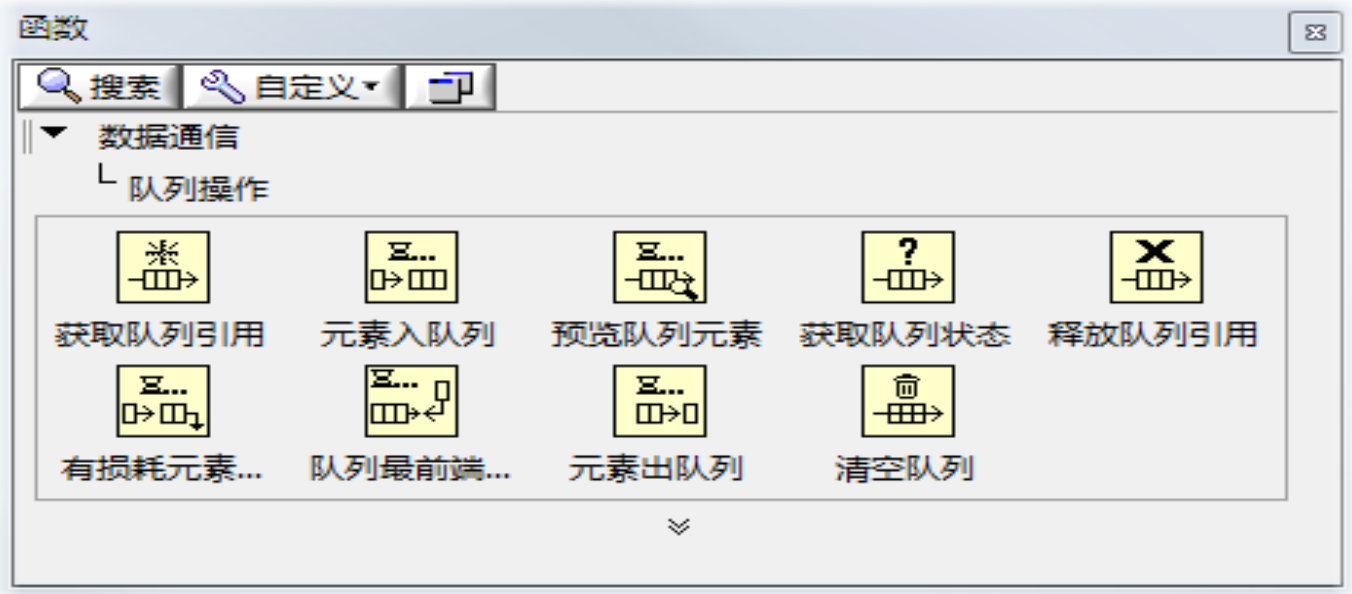


队列操作

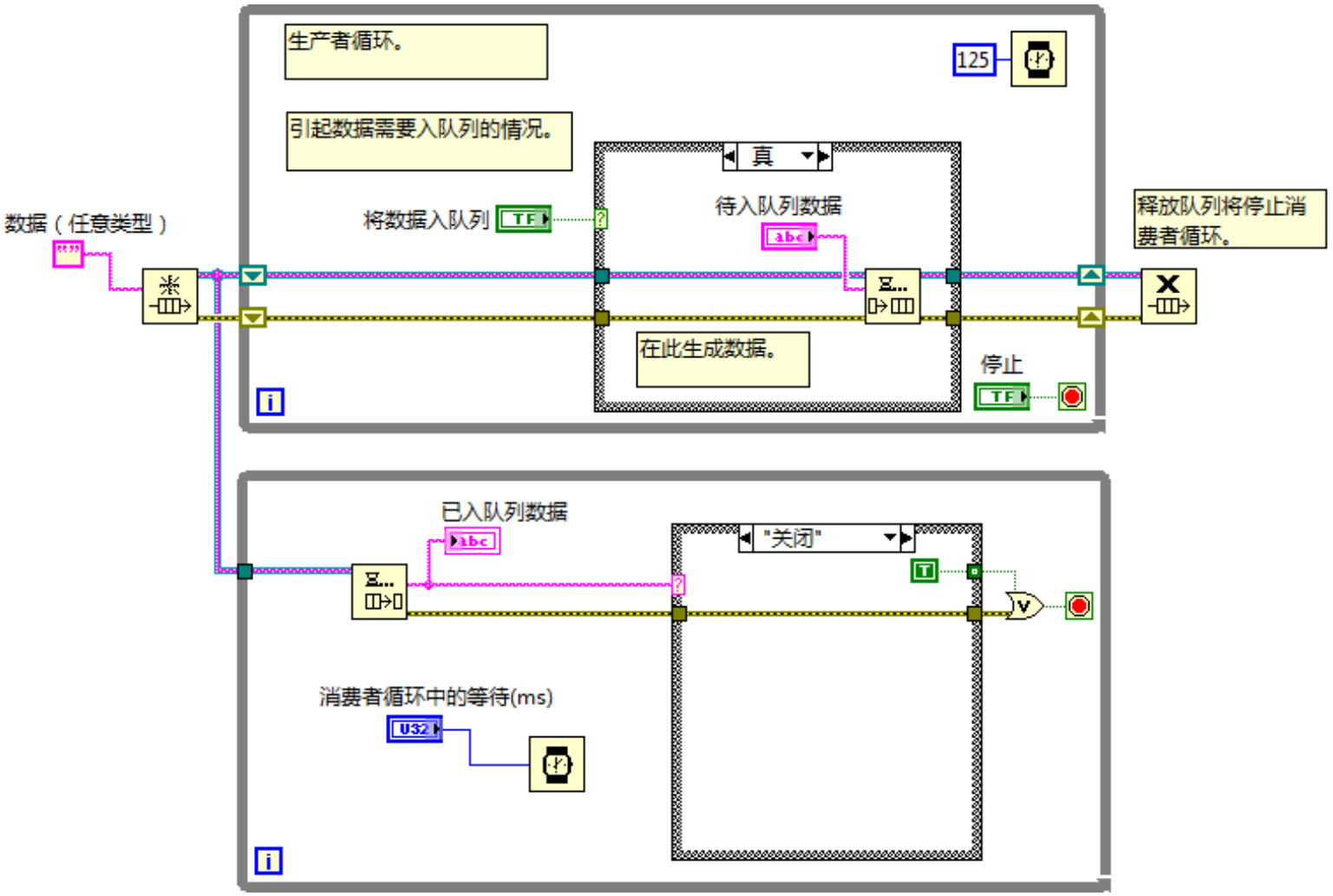
使用“队列操作”函数
创建和使用下列各项之间进行数据通信的队列：

VI的不同部分。

不同VI。



生产者 / 消费者设计模式（数据）



第3部分
改进现有VI

重构继承代码

典型问题

A. 重构继承代码

定义

重构时间

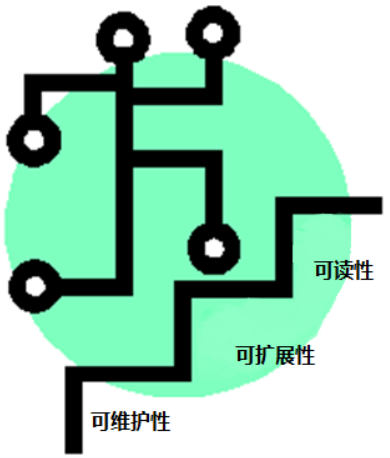
重构过程

重构继承代码

继承的VI可能设计上有缺陷，使得日后难以为VI添加功能。

重构：

- 软件重构的过程使软件更具可读性并降低了维护的难度，从而保证了修改软件的成本不随时间增加。
- 修改VI的*内部*结构，
  使其更具可读性和可维护性，
  但不会改变VI的可视化操作。



重构时间

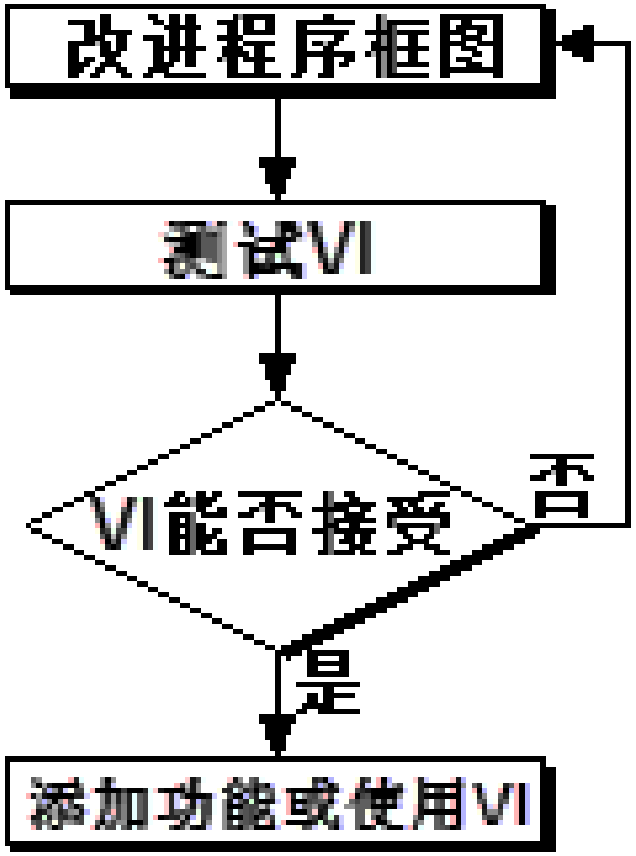
- 为VI增加新功能或调试VI时。
- 需要全部重构的情况包括：

-不能运行的VI。

-仅满足部分需求的VI。

重构过程

重构改进程序框图时，请先调整外观，然后处理较严重的重构问题。



B. 典型问题

杂乱无章
命名不恰当
过于复杂
重复逻辑
数据流断开
过时的方式

杂乱无章

程序框图过于混乱、过大或包含过多嵌套结构

解决方法：

- 在程序框图内移动对象。
- 创建子VI以缩小和组织程序框图。
- 添加注释以增强可读性。

命名不恰当

程序框图使用了不正确的对象名称和不合适的图标。

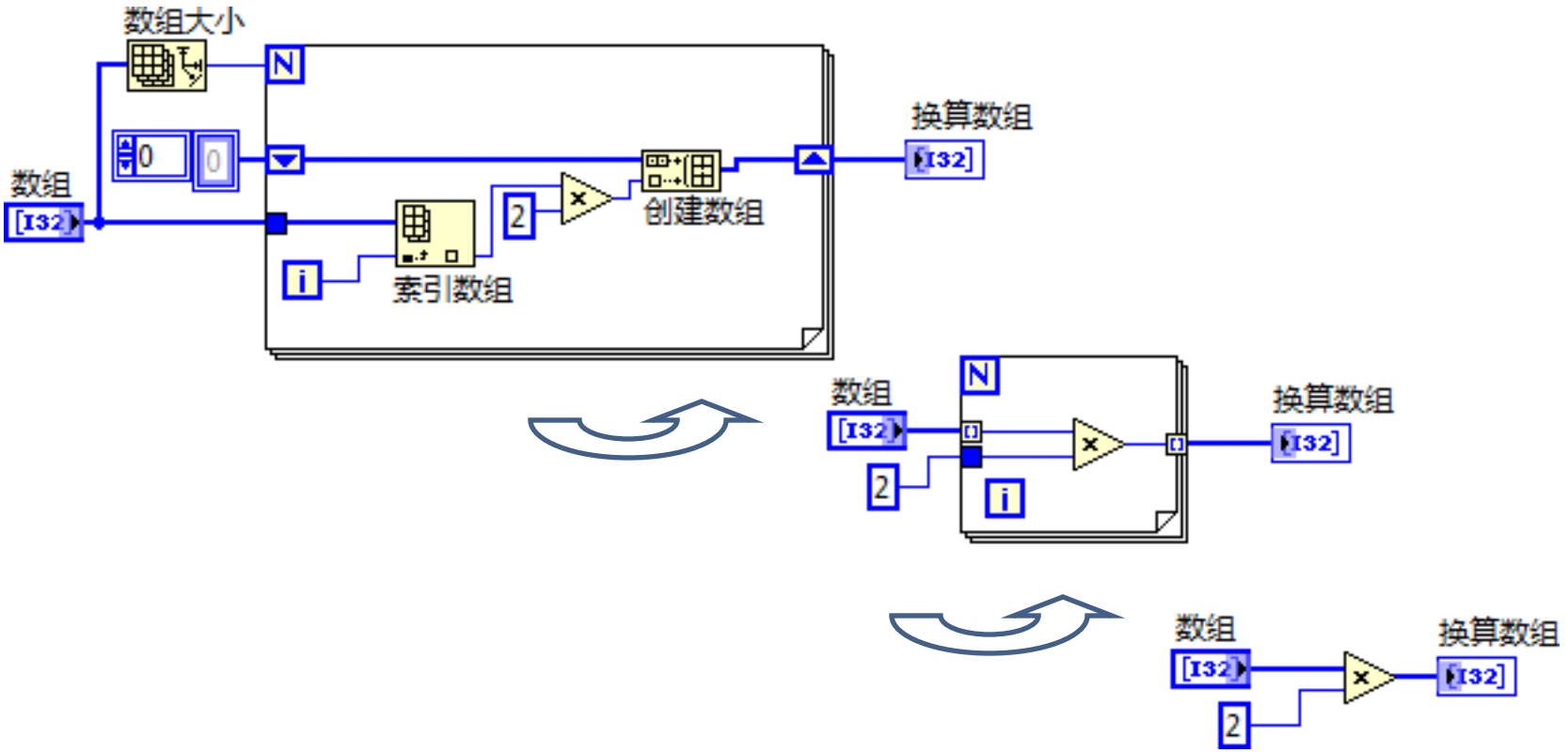
好

较好

最佳

过于复杂

程序框图使用了复杂或不必要的逻辑。

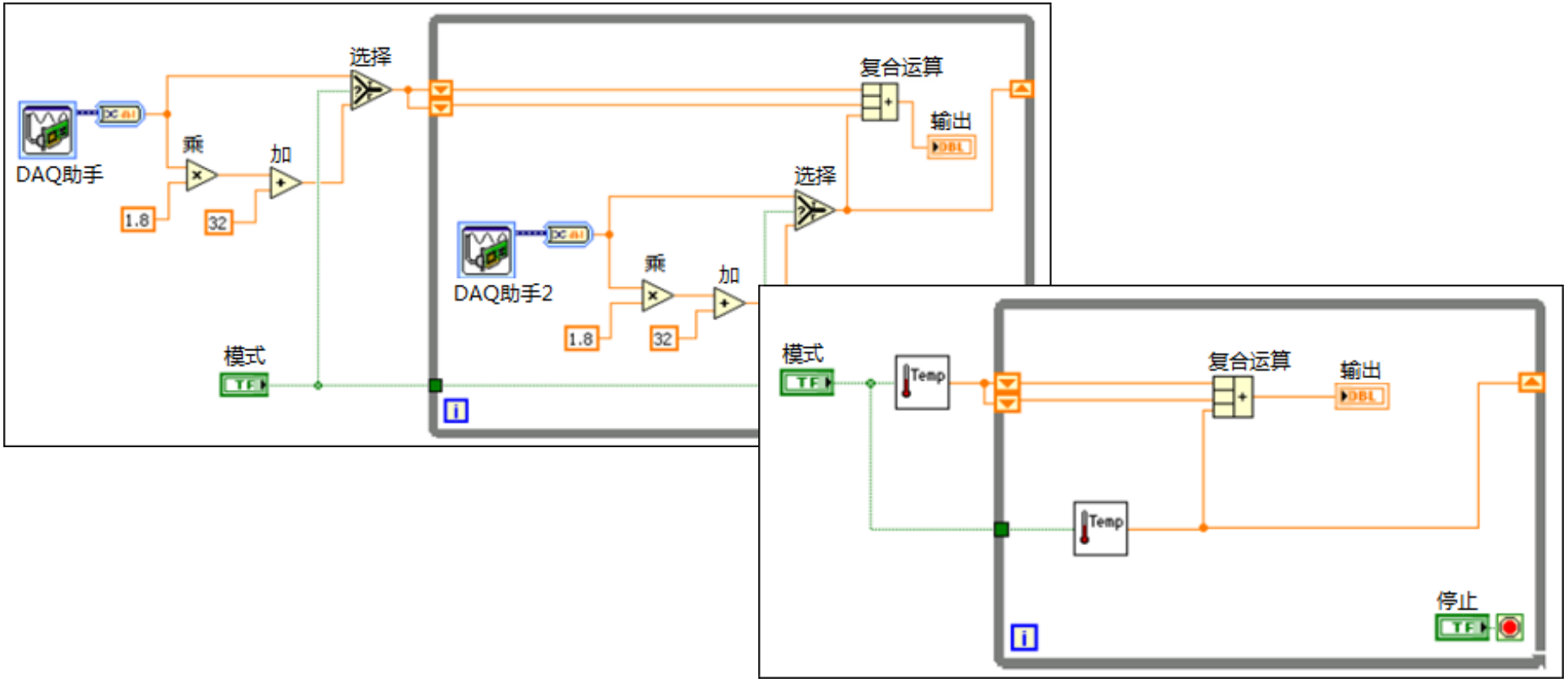


重复逻辑

程序框图使用了重复逻辑。

解决方法：

为重复逻辑部分创建子VI，实现VI的重构。

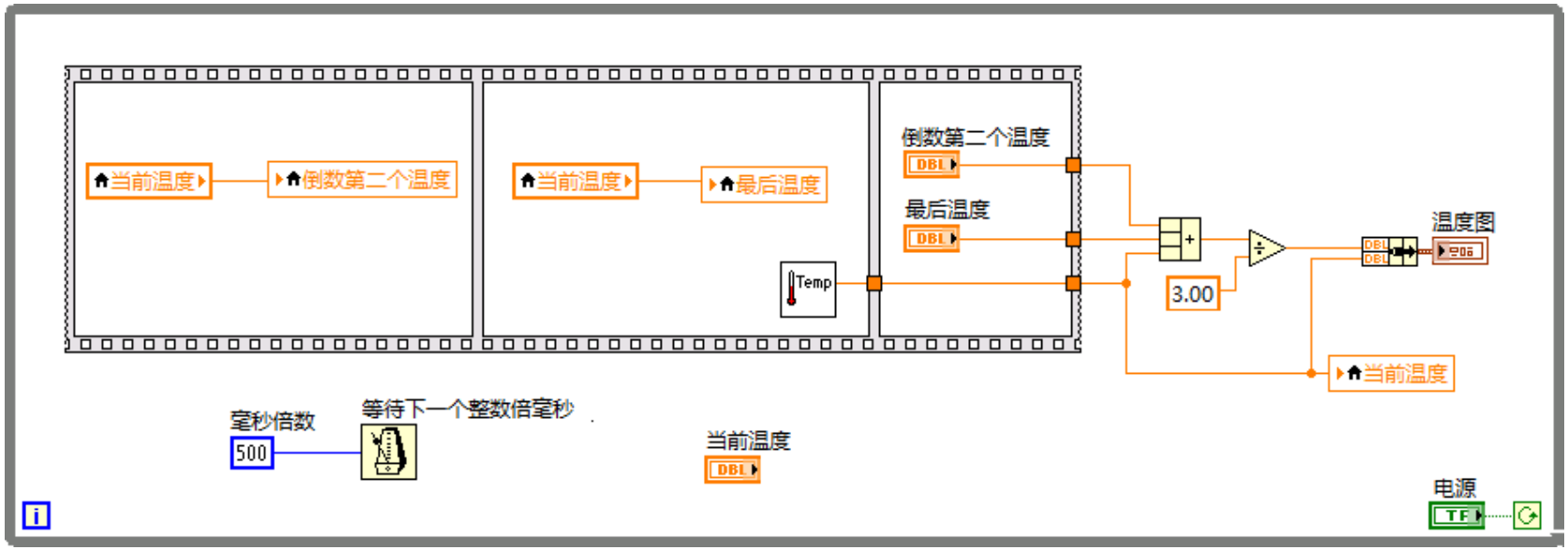


数据流断开

程序框图未使用数据流编程。

解决方法：

- 用状态机替换顺序结构。
- 删除局部变量，直接连线输入控件或显示控件。

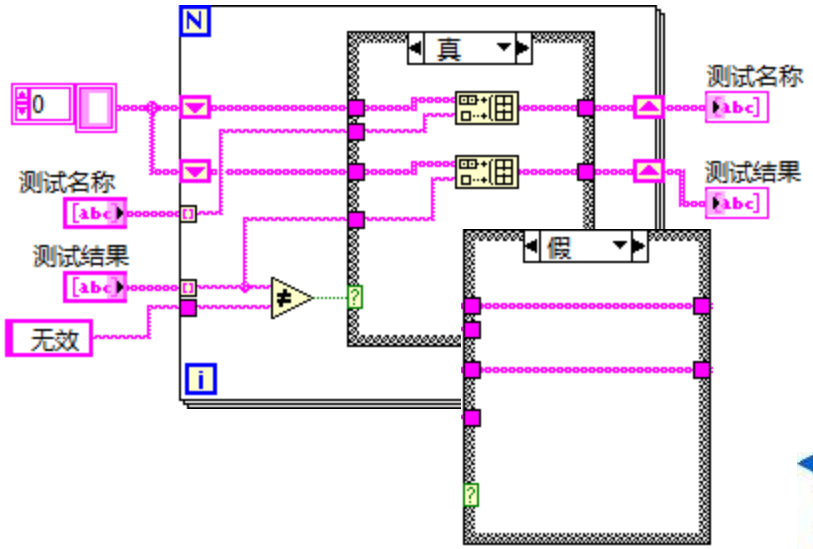


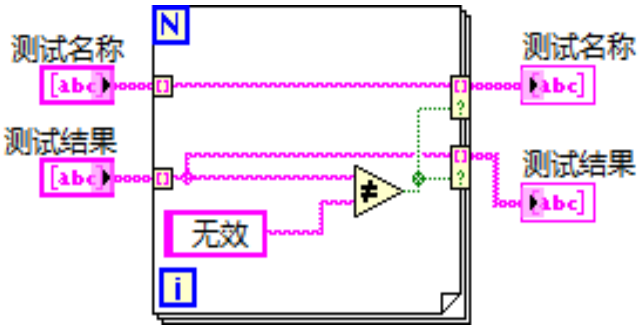
过时的方式

VI用较早版本的LabVIEW创建，其中包含一些过时的方式。

解决方案：

- 用基于事件的设计替换基于轮询的设计。
- 添加简化代码的新功能。





继续学习LabVIEW

- 面授培训
  - LabVIEW基础教程（二）：将学习并行循环、变量、属性节点和创建可执行程序
  - 硬件课程（如数据采集和信号处理）
  - 在线课程（如机器视觉和LabVIEW实时）
- 定制：各种设计用于自学的指导材料和工具

初级用户 | 中级用户 | 高级用户

LabVIEW核心教程（一）
LabVIEW核心教程（二）

LabVIEW核心教程（三）

LabVIEW互连接口
LabVIEW面向对象设计和编程
LabVIEW性能

LabVIEW中的软件工程管理
LabVIEW高级架构

认证

LabVIEW助理开发工程师考试

LabVIEW开发工程师考试

LabVIEW程序架构师考试

其他课程

LabVIEW Real-Time 1
LabVIEW Real-Time 2

LabVIEW仪器控制
LabVIEW模块化仪器

LabVIEW FPGA
数据采集和信号调理

如何在课堂外学习更多相关知识

- ni.com/support
  - 可查找到产品使用手册、知识库、范例代码、教程、应用指南和论坛
  - 请求技术支持
- Info-LabVIEW: www.info-labview.org
- Alliance Program: ni.com/alliance
- 出版物：ni.com/reference/books/
- 实践！