链滴

2.1 UML 状态机速成教程之一

作者：li3p

原文链接：https://ld246.com/article/1624890967764

来源网站：链滴

UML状态机速成教程

真正对我们有帮助的地方是优化IF-THEN-ELSE结构。大多数程序一开始的结构都相当好。可随着错的发现和特性的添加，IF和ELSE也被不断地加进来，一直到没人真正了解数据是如何流经一个函数的漂亮的打印能帮上些忙，但它无法减少一个嵌套15层的IF语句的复杂性。

—Jack Ganssle, “Break Points,” ESP Magazine, January 1991

传统的顺序式程序，可以借助循环和嵌套函数调用之类的标准构件，被架构成一个单一的控制流。这的程序中，大部分的执行上下文是通过程序计数器指针、函数调用树以及堆栈上分配的临时变量来表的。

而事件驱动式程序则需要一系列细粒度的事件处理函数来响应事件。这些事件处理函数必须快速执行并总是返回到主事件循环，因此在调用树和程序计数器中无法保留任何上下文。此外，所有的堆栈变在调用不同的事件处理程序之间都不复存在。所以事件驱动式程序严重依赖静态变量来保存不同事件理程序调用之间的执行上下文。

因此，事件驱动编程的最大挑战之一，就在于管理以数据形式呈现的执行上下文。主要问题是上下文据必须能以某种方式反馈回事件处理程序中的代码控制流，以便每个事件处理程序只执行与当前上下相对应的动作。传统上，这种对上下文的依赖往往会导致深度嵌套的if-else结构，后者基于上下文数（Context Data）来决定控制流。

如果你能消除哪怕是一小部分的条件分支（或者叫 "意大利面条"代码），软件都会变得更容易理解、试和维护，代码中曲折的执行路径的数量也会急剧下降，而且通常是成数量级的下降。这正是基于状机的技术的用武之地--通过大幅减少代码中的各种路径，来消除在每个分支点的条件测试。

在这一章中，我简要介绍了UML状态机，描述了这些技术在长期演进中的当前状态。给出一个完整的正式的UML状态机讨论不是我的目的，因为OMG官方规范里已经[OMG 07]全面地、正式地涵盖了些内容。相反，我在这一章的目标是通过建立基本术语、介绍基本符号和澄清语义歧义，来为后续的容打下基础。本章只限于状态机特性中那些可以说是最基本的子集。着重于UML状态机在实际、日常程中的作用，而不是数学抽象。

事件-动作范式的过度简化

目前结构化事件驱动软件的主流方法，是无处不在的 "事件-动作 "范式（Event-Action Paradigm）在这个范式中，事件被直接映射代码，以执行预期的响应。事件-动作范式是理解状态机的重要基石所以在这一节中，我简单介绍一下它在实践中是如何工作的。

我会使用一个来自图形用户界面（GUI）领域的例子，因为GUI是事件驱动系统的典范。在《用状态构造用户界面》[Horrocks 99]一书中，Ian Horrocks讨论了一个简单的GUI计算器应用程序，该程序为Microsoft Visual Basic的示例程序，发行了数百万份，他在其中发现了一些严重的问题。正如Horrcks所指出的那样，这个分析的重点并不是要批评这个特殊的程序，而是要找出其构建中所基于的常原则的缺点。

当你启动Visual Basic计算器（可从配套的网站上获得，在目录<qp>****/resources/vb/vccalc.ex中），你一定会发现，大多数时候它都能正确地进行加、减、乘、除运算（见图2.1（A））。**

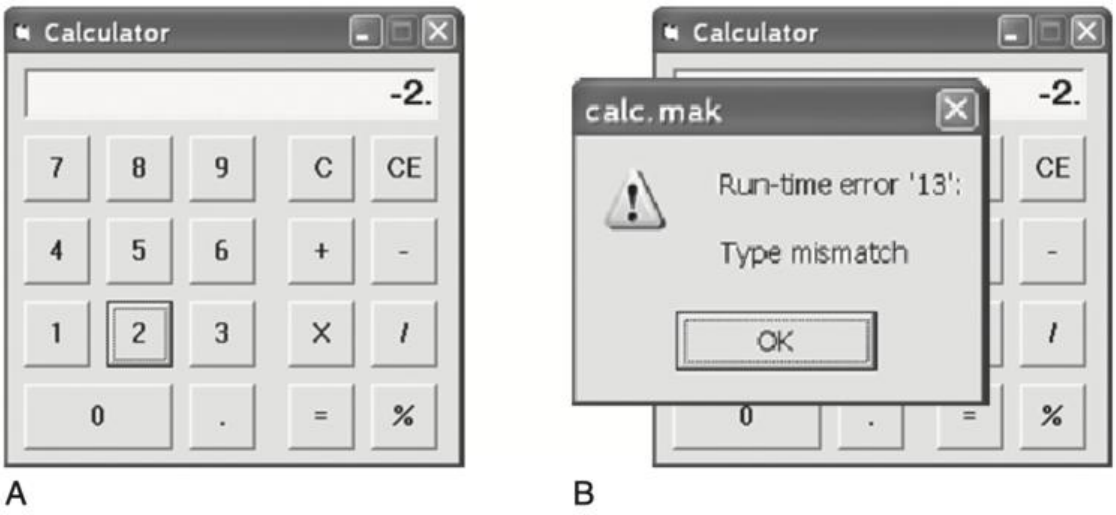

Figure 2.1: Visual Basic calculator before the crash (A) and after a crash with a runtime error (B).

那有什么不满意的呢？然而，多操作一段时间，你可以发现许多边界条件的情况下，计算器提供错误结果，冻结，或完全崩溃。

Ian Horrocks仅经过一个小时的测试，就在Visual Basic计算器中发现了10个严重错误。试着找出至一半的错误。

例如，Visual Basic计算器对-事件的响应经常会出问题，只要尝试以下操作顺序。**2, -, -, -, -, 2, =**. 用程序就因运行时错误而崩溃（见图2.1(B)）。这是因为同一个按钮(-)具有对一个数字取负值和输减法运算符两种功能。因此，对"-"按钮点击事件的正确解释取决于其发生时的上下文或模式。同样，E（取消输入）按钮偶尔也会错误地工作--尝试输入2，x，CE，3，=，观察到CE没有任何效果，尽管似乎从显示中撤销了2的输入。同样，CE在取消操作数和取消运算符时的表现应该不同。事实证明，论上下文如何，应用程序处理CE事件的方式总是一样的。此时，你可能已经注意到一个新出现的模式。**用程序特别容易受到需要根据上下文进行不同处理的事件的影响。**

这并非是说Visual Basic计算器没尝试去处理上下文。恰恰相反，如果你看一下计算器的代码(可以从站上的<qp>/resources/vb/calc.frm目录中找到)，你会发现实际上管理上下文正是这个程序的主要注点。代码中充斥着大量的全局变量和标志位，它们只有一个目的：处理上下文。例如，DecimalFlag示已经输入了一个小数点，OpFlag表示一个待定的运算，LastInput表示最后一次按钮按下的事件类，NumOps表示操作数，等等。采用这种表示方式，计算的上下文被表达地模糊不清，因此很难准确判断出应用程序在特定时刻位于哪种模式。实际上，这个程序没有任何单一操作模式的概念，取而代的是由全局变量和标志的值决定的一、堆紧密耦合并且重叠的运算条件。

代码清单2.1显示的条件逻辑中，操作符号事件（+，-，*，和/）的事件处理程序试图确定**-**（减按钮的点击，是应该被当作取反还是减法。

Listing 2.1 Fragment of Visual Basic code that attempts to determine whether the – (minus) b tton-click event should be treated as negation or subtraction

```
Private Sub Operator_Click(Index As Integer)

...
```

```
Select Case NumOps
   Case 0
      If Operator(Index).Caption = "-" And LastInput <> "NEG" Then
         ReadOut = "-" & ReadOut
         LastInput = "NEG"
      End If
   Case 1
      Op1 = ReadOut
      If Operator(Index).Caption = "-" And LastInput <> "NUMS" And OpFlag <> "=" Then
         ReadOut = "-" LastInput = "NEG"
      End If

...
```

Listing 2.1

清单2.1中举例的方法是产生 "边角情形 "行为（也就是bug）的沃土，原因至少有三。

1. 它总是导致复杂的条件逻辑（也就是 "意大利面条 "代码）
2. 在每个分支点都需要评估一个复杂的表达式。
3. 在不同模式之间的切换需要修改很多变量，从而很容易造成不一致。

像清单2.1中呈现的复杂条件表达式，散布在整个代码中，在运行时进行求值从而导致不必要的复杂性能开销。它们也是出了名的难以搞正确，即便是对是有经验的程序员也是如此，就如那些仍然潜伏Visual Basic计算器中的错误所证明的那样。这种方法很有欺骗性，因为它在最初看起来很好用，但着问题复杂性的增加，它的代码规模就没办法再上去了。显然，计算器应用程序（总体上只有7个事处理程序和大约140行包括注释在内的Visual Basic代码）是足够复杂的，以至于很难用这种方法来正处理。

上面所列举的缺陷，根源在于对事件-动作范式的过度简化。我希望 Visual Basic 计算器的例子能清地表明，**单一事件本身并不能决定该事件的响应所应执行的动作，至少当前的上下文也同样重要** 。然，流行的事件-动作范式只认识到对事件类型的依赖性，而将上下文的处理很大程度上留给了各种临技术，这些技术很容易产出为意大利面条代码。