链滴

2.2 UML 状态机速成教程之二：基本概念

作者：li3p

原文链接：https://ld246.com/article/1624893159120

来源网站：链滴

<h2 id="2-2-状态机基本概念">2.2 状态机基本概念</h2>
通过对事件-动作范式（Event-Action Paradigm）进行扩展，可以明确包含对 执行下文（ Execution Context） 的依赖。事实证明，大多数事件驱动系统的行为可以被划分少量的大块，在每个单独的块内部，事件响应确实只依赖于当前的事件类型，而不依赖于过去的事件列（也就是上下文）。换句话说，事件-行为范式仍然适用，只不过被局部应用在了每个单独的大块部。
基于这一思想，一种常见且直接的行为建模方式就是利用有限状态机（Finite State Machine, FM）。在这种方式中，"行为块（Chunks of Behavior）"被称为状态（State)行为的变化（即对任意 事件（Event） 的响应）对应于状态的变化，被称为状态迁移(State Transition) 。FSM 是义系统整体行为约束的有效方法。处于某个状态，就意味着系统只需处理所有可能事件的某个子集，味着产生的响应也是全部可能响应的某个子集，进而意味着状态可以直接迁移到的目标状态是全部可状态的某个子集。
FSM 概念在编程中很重要，因为它使事件处理明确地依赖于事件类型和系统的执行上下文（即状态）。如果使用得当，状态机就会成为一个强大的 "意大面条修剪器"，它可以大幅削减代码中执行路径的数量，减少每个分支点的测试条件，并简化不同执模式之间的转换。
<h3 id="2-2-1-状态-States-">2.2.1 状态（States)</h3>
状态（State） 可以非常有效地捕捉系统历史的相关信息。比如当你在键上敲击一个键时，生成的对应字符代码是大写还是小写，取决于大写锁定键（Caps Lock）是否激活因此，键盘的行为可以分为两个块（状态）："default" 状态和 "caps_locked " 状态。（实际上大多数键盘都有一个 LED 灯，用于指示键盘处于 "大写锁定"状态）。键盘的行为只决于其历史的某些方面，即是否按过 Caps Lock 键，而不取决于之前按过多少键和具体是哪个键等情况。一个状态能够对所有可能的（但不相关的）事件序列进行排除，只捕捉有关系的事件序列。</>
把这一概念跟编程联系起来，这意味着你不需要把事件历史记录在大量的变量、标志和复杂的逻中，而是主要依靠一个状态变量，它的取值只能假定在有限的、事先确定的范围里（比如，在上面键的例子中就是两个值）。状态变量（State Variable） 的值清晰地定义了系统任何特定时刻的当前状态。状态这一概念把在代码中确定执行上下文的问题，简化成了只检测单个状变量而不是大量关联变量，从而省去了大量的条件逻辑。实际上，除了最基本的状态机实现技术外，所有的状态机实现技术中，比如第 3 章讨论的 "嵌套 switch 语句 "技术，甚至连对状态变量的显式检都从代码中消失了，这就进一步减少了 "意大利面条"式的代码（你将在后面的第 3 章和第 4 章中体到这种效果）。此外，不同状态之间的切换也大大简化了，因为你只需要重新对一个状态变量赋值，不是以后果自负的方式去修改多个变量。**
<h3 id="2-2-2-状态图-State-Diagram">2.2.2 状态图，State Diagram</h3>
状态图（State Diagram） 是 FSM 的图形化表示形式。这些图是有向图（irected Graph），其中节点表示状态（State），连接线表示 状态移 （State Transition）。
例如，图 2.2 显示了对应计算机键盘状态机的 UML 状态转换图。在 UML 中，状态被表示为圆矩形，标有状态名称。状态迁移用箭头表示，标有触发事件，后面可选地标有行动作的列表。初始迁移（Initial Transition）源于实心圆，指定了系统第一次始时的起始状态。每个状态图都应该有这样一个迁移，它没有名称标记，因为它不是由事件触发的。始迁移可以有关联的动作 （Action）。

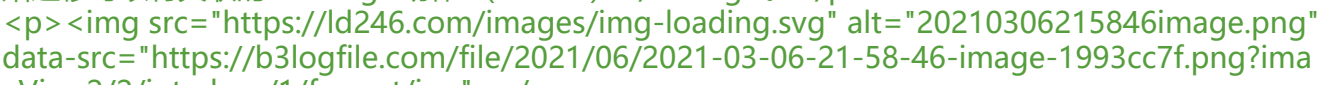
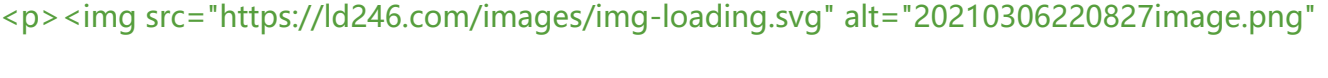



Figure 2.2: UML state diagram representing the computer keyboard state machine.
<h3 id="2-2-3-状态图对比流程图">2.2.3 状态图对比流程图</h3>
状态机模式的新手经常会把状态图和流程图混淆。UML 规范[OMG 07]在这方面没起好作用，因它把活动图归入状态机包。活动图本质上是更精细的流程图。
图 2.3 显示了状态图与流程图的对比。状态机（图片左侧 A）根据明确的触发事件执行相应动作相反，流程图（图片右侧 B）不需要明确的触发事件，而是在活动完成后自动地从图中的节点迁移到一个节点。


data-src="https://b3logfile.com/file/2021/06/2021-03-06-22-08-27-image-f09cceb1.png?imaeView2/2/interlace/1/format/jpg">
Figure 2.3: Comparison of (A) state machine (statechart) with (B) activity diagram (flowchat).
从图形上看，与状态图相比，流程图反转了圆角矩形和连线的含义。在状态图中，处理过程关联连线（状态迁移）上的，而在流程图中，它与圆角矩形相关联。当一个状态机处于某个状态中等待事发生时，它是空闲的。当一个流程图处于一个节点中时，它就忙于执行活动。图 2.3 试图通过将状态的连线与流程图的处理阶段用虚线对齐来显示这种角色的反转。
你可以把流程图看作制造业中的流水线，因为流程图描述的是一些任务从开始到结束的进程（例，把输入的源代码转化为编译器输出的对象代码）。状态机一般没有这种前进的概念。例如，计算机盘在处于 "caps_locked "状态时，与处于 "default "状态相比，并不处于更高级的阶段；它只是对事的响应不同而已。状态机中的状态是一种定义特定行为的有效方式，而不是定义处理阶段的有效方式
状态机和流程图之间的区别尤为重要，因为这两个概念代表了两种截然相反的编程范式：<stron>事件驱动式编程（状态机）和转换式编程（流程图）。如果没有不地思考可用的事件，就无法设计出有效的状态机。相反，对于流程图来说，事件只是一个次要的关注（如果有的话）。
<h3 id="2-2-4-扩展状态机-Extended-State-Machine">2.2.4 扩展状态机，Extended State Machne</h3>
对于软件系统的状态，一种可能的解释是，每个状态代表整个程序内存的一组不同的有效值。即是只有几个基本变量的简单程序，这种解释也会导致一个天文数字级的状态。例如，一个单一的 32 整数可能会导致超过 40 亿个不同的状态。显然，这种解释是不实际的，所以程序变量通常与状态无。相反，系统的完整条件（称为扩展状态）是定性方面（状态）和定量方面（扩展状态变量）的结合在这种解释中，变量的变化并不总是意味着系统行为的定性方面的变化，因此不会导致状态的变化[Selc+ 94]。
带有额外变量的状态机被称为扩展状态机（Extended State Machines）相比不包含扩展状态变量的状态机，扩展状态机可以将其底层机制用于解决更复杂的问题。例如，假键盘的行为取决于截止当前在键盘上输入的字符数，并且比如在 1000 次击键之后，键盘会损坏并进最终结束状态。要在没有内存的状态机中模拟这种行为，您需要引入 1000 个状态（例如，在状态 <code>stroke123</code> 中按下一个键将导致状态 <code>stroke124</code>，以此类推），这显是一个不切实际的提议。作为另一种手段，你可以结合一个 <code>key_count</code> 递减变量构造一个扩展状态机。计数器将被初始化为 1000，并在不改变状态的情况下，每击一次键就递减一。当计数器达到零时，状态机将进入结束状态。
图 2.4 中的状态图是扩展状态机的一个例子，其中系统的完整状态（称为扩展状态，Extended Sate）是定性和定量两方面的结合；前者是指"状态"，后者是指扩展状态变量（递减变量**<code>key_count</code>**）。在扩展状态机中，一个变量的变化并不总是意味着系行为的定性改变，因此并不总是导致状态的变化。
扩展状态机的显著优势是灵活性。例如，要将 "廉价键盘"的寿命从 1000 次按键延长到 10000 按键，完全不会导致扩展状态机变复杂。唯一需要修改的是改变初始迁移中 <code>key_count</coe> 递减变量的初始值。
<img src="https://ld246.com/images/img-loading.svg" alt="16150406059379.jpg" data-sc="https://b3logfile.com/file/2021/06/16150406059379-5b25f0dc.jpg?imageView2/2/interlac/1/format/jpg">
Figure 2.4: Extended state machine of “cheap keyboard” with extended state variable ky_count and various guard conditions.
<h3 id="2-2-5-守卫条件-Guard-Conditions">2.2.5 守卫条件，Guard Conditions</h3>
然而，扩展状态机的这种灵活性是有代价的，因为扩展状态的 "定性"和 "定量"之间存在着复杂耦合。这种耦合是通过附加在迁移（Transition）上的守卫条件（Guard Condition）引入的，如图 24 所示。
守卫条件（或简称为守卫，Guard）是基于扩展状态变量的值和事件参数来动态求值的布尔表达（参见下一节对事件和事件参数的讨论）。守卫条件影响状态机行为的方式，是只有当它们求值为 "真"时会才能触发动作或迁移，而当它们求值为 "假 "时则会不会触发。在 UML 符号中，守卫条件显示方括号中 (例如，<code>[key_count == 0]</code>)。
对守卫的需求是在状态机模式中增加内存（扩展状态变量）的直接后果。尽量少用，扩展状态变

和守卫构成了一个非常强大的机制，可以极大地简化设计。但是不要被这些花哨的名字（"guard"）简洁的 UML 符号骗了。当你实际编写一个扩展状态机的代码时，守卫就变成了你开始时想通过使用态机来消灭掉的 IF 和 ELSE。滥用的话你会发现自己又回到了原点（"意大利面条"），在那里，守卫底接手了系统中所有相关条件的处理。
事实上，在基于状态机的设计中，滥用扩展状态变量和守卫是造成架构劣化的罪魁祸首。通常，日常的实战中，特别是对于刚接触状态机模式的程序员来说，加个新扩展状态变量和守卫条件（另一个 if 或 an else），与把相关行为归因成系统的一个新的定性状态比起来更有诱惑力。根据我在实战中经验，这种架构劣化的可能性与添加或删除状态所涉及的开销（实际的或预感上的）成正比。（这就为什么我不是特别喜欢我在第 3 章中描述的状态表这一流行的状态机实现技术，因为添加一个新的状需要在表中添加和初始化一个全新的列）。
成为一个出色的状态机设计者的主要挑战之一是培养一种意识，即哪些行为的部分应该作为 "定性"的方面（"状态，State"）被捕获，哪些元素最好作为 "定量 "方面（扩展的状态变量，Extended Sta e Variables）。一般来说，你应该积极寻找机会将事件历史（发生了什么）作为系统的 "状态 "来捕，而不是把这些信息存储在扩展状态变量中。例如，Visual Basic 计算器使用扩展状态变量 <code>D cimalFlag</code> 来记住用户输入的小数点，以避免在同一个数字中输入多个小数点。然而，一个好的解决方案是观察到输入小数点确实会导致一个不同的状态 <code>entering_the_fractional_part of_a_number</code>，在这个状态下，计算器会忽略小数点。这个解决方案的优越性有很多原因不那么重要的原因是，它省去了一个扩展的状态变量和初始化和检测它的需要。更重要的原因是，<st ong>基于状态的解决方案更加稳健，因为上下文信息是在局部使用的（仅限于当前特定状态），一旦它变得不相关，就会被丢弃。一旦正确地输入了数字，对于计算器的后续操作来说，这数字是否有小数点并不重要。状态机会迁移到另一个状态，并自动 "忘记 "之前的上下文。另一方面，code>DecimalFlag</code> 扩展的状态变量，当信息变得无关的时候还"闲置"在那里（并有可能变过时！）。
更糟糕的是，你一定不能忘记在输入另一个数字之前重置 <code>DecimalFlag</code>，否则标志会错误地表明用户的确曾经输入过小数点，但那也许是在上一次输入数字的上下文中发生的。</ >
将行为捕捉为定量式 "状态 "有其缺点和局限性。首先，状态机里的状态和迁移拓扑必须是静态，并且在编译时是固定的，这可能限制性太强，缺乏灵活性。当然，你可以很容易地设计出会在运行自我修改的 "状态机"（当你试图将 "意大利面代码 "重新用状态机实现时，实际上经常会发生这种情）。然而，这就像在编写能自我修改的代码，在编程的早期确实有人这样做，但很快就被当作一个普的坏主意而被驳回。因此，"状态"只能捕获行为的静态方面，这些方面是先验已知的，而且在未来不可能改变 。
比如，在计算器中捕获小数点的输入作为一个单独的状态 "<code>entering_the_fractional_par _of_a_number</code>"是可以的，因为一个数字只能有一个小数点，这既是先验已知的，也不可能未来改变。然而，如果没有扩展的状态变量和守卫条件，实现 "廉价键盘 "实际上是不可能的。这个子指出了定量式 "状态 "的主要弱点：它根本无法存储过多的信息（比如范围宽的击键次数）。因此，扩展状态变量和守卫是一种为状态机增加额外运行时灵活性的机制。
<h3 id="2-2-6-事件-Events">2.2.6 事件，Events</h3>
在最普遍的术语中，一个 事件（Event） 是在时间和空间上对系统有意义行为。严格来说，在 UML 规范中，事件指的是行为的类型，而不是这个行为的任何具体的实例[OMG 07]。例如，<code>Keystroke</code> 是键盘的一个事件，但每次按键行为不是一个事件，而是 <code>Keystroke</code> 事件的一个具体实例（ Concrete Instance）。键感兴趣的另一个事件可能是 <code>Power-on</code>，但明天 10:05:36 打开电源只是 <code>P wer-on</code> 事件的一个实例。
一个事件可以携带相关参数，使得事件实例不仅可以传达一些值得关注事件的发生，还可以传达关该事件的定量信息。例如，在计算机键盘上按下一个键所产生的 <code>Keystroke</code> 事件相关参数传递的内容包括字符扫描码以及 Shift、Ctrl 和 Alt 键的状态。
一个事件实例的生命周期超过了生成它的行为瞬时，并可能在一个或多个状态机传递这个瞬间行。一旦生成，事件实例就会经历一个处理的生命周期，最多可以包括三个阶段。首先，事件实例被接并等待处理（例如，它被放在事件队列中）。随后，事件实例被分派到状态机，此时它成为当前事件最后，当状态机完成对事件实例的处理时，它被消耗掉。一个被消耗的事件实例不可再用于后续处理
<h3 id="2-2-7-动作和迁移-Actions-and-Transitions">2.2.7 动作和迁移，Actions and Transition</h3>

当一个事件实例被 分派（Dispatch） 时，状态机通过执行 动作（Action） 来做出响应，如改变一个变量、执行 I/O、调用一个函数、生成另一个事件实例或改变为另一状态。与当前事件相关联的任何参数值对该事件直接触发的所有动作都是可见的。
<p>从一个状态切换到另一个状态称为 状态迁移（State Transition） ，而引状态迁移的事件称为 触发事件（Triggering Event） ，或者干脆称为 <strong触发器（Trigger）。在键盘的例子中，如果当键盘处于 "<code>default</code>"状态按下 <code>Caps Lock</code> 键时，则键盘将进入 "<code>caps_locked/code>** "状态。然而，如果键盘已经处于 "<code>caps_locked</code> "状态，按 Caps Lock 将导致不同的迁移--从 "<code>caps_locked</code> "状态到 "<code>default</code>"状态。这两种情况下，Caps Lock 按下都视为触发事件。</p>
在扩展的状态机中，一个 迁移（transition） 可以有一个守卫，这意味着有当守卫条件求值为 "TRUE "时，迁移才能 "触发"。一个状态允许有多个迁移来响应同一个触发事件只要它们有互不重叠的守卫条件；然而，这种情况可能会在共同触发事件到来时，因为守卫的求值顺产生问题。UML 规范有意不对顺序做出任何特殊规定；相反，UML 把这一设计负担放在设计者身上以求值顺序顺序无关的方式来设计守卫条件。实际上，这意味着守卫表达式应该没有副作用，至少不改变其它具有相同触发事件的守卫条件的求值。
<h3 id="2-2-8-运行至完成执行模型-Run-to-Completion-Execution-Model">2.2.8 运行至完成行模型，Run-to-Completion Execution Model</h3>
在 运行至完成（RTC，Run-to-Completion） 执行模型中，系统以离散、不可分割的 RTC 步骤处理事件。新传入的事件不能中断当前事件的处理，必须被存储起来（通常在事件队列中），直到状态机再次成为空闲状态。这些语义完全避免了单一状态机内的任何内部并发题。RTC 模型还从概念层面解决了处理与 迁移（Transition） 关联的动作的问，在这种动作（Action）的持续时间内，状态机并不处于一个定义良好的状态（它处于两个状态之间。在事件处理期间，系统是无响应的（不可观察），所以这段时间内定义不清的状态没有任何实际意。
但请注意，RTC 并不意味着状态机必须独占 CPU，直到 RTC 步骤完成。抢占限制只适用于当前执行处理事件的状态机的任务上下文。在多任务环境中，其他任务（与执行中的状态机的任务上下文关）可以运行，并可能会抢占当前执行的状态机。只要当前状态机与其他状态机之间不共享变量或其资源，就不存在并发危险。
RTC 处理的主要优点是简单。其最大的缺点，是状态机的响应性取决于其最长 RTC 步骤的执行间。5 而实现较短的 RTC 步长往往会使实时设计显著复杂化。