第21章 状 态 机

本章内容：

- 状态、转换和活动
- 对对象的生命期建模
- 创建结构良好的算法

使用交互，可以对共同工作的对象群体的行为建模。使用状态机，可以对单个的对象的行为建模。一个状态机是一个行为，它说明对象在它的生命期中响应事件所经历的状态序列以及它们对那些事件的响应。【在第15章中讨论交互；在第13章中讨论对象。】

状态机用于对系统的动态方面建模。在大多数情况下，这包括说明一个类、一个用况或整个系统的实例的生命期。这些实例可能响应诸如信号、操作或时间段这样的事件。当一个事件发生时，某些活动将依赖对象的当前状态而发生。一个活动是一个状态机中进行的非原子执行。活动最后将导致某些动作，动作是由引起模型状态改变或值的返回的可执行的原子计算组成的。对象的状态是指在这个对象的生命期中的一个条件或状况，在此期间对象将满足某些条件、执行某些活动或等待某些事件。【在第4章和第9章中讨论类；在第16章中讨论用况；在第31章中讨论系统。】

你可以用两种方式来可视化状态机：一种是强调从活动到活动的控制流（使用活动图），另一种是强调对象的潜在状态和这些状态之间的转换（使用状态图）。【在第19章中讨论活动图；在第24章中讨论状态图。】

结构良好的状态机像结构良好的算法一样：是简单、高效、可适应和可理解的。

21.1 入门

想象一下你家里的恒温器在一个晴朗的秋日里的工作情况。

在早晨最初的几个小时中，对恒温器而言事情相对平静。屋内的温度和屋外的温度都是稳定的。然而，接下去事情就变得有趣了。太阳从地平线升起，周围的温度渐渐升高。家庭成员开始醒来，某人可能翻身从床上起来，扭动恒温器的刻度盘。这些事件对于房间的加热和制冷系统是重要的。恒温器通过控制房间的加热器（加热屋内的温度）或空气调节器（降低屋内的温度）而开始像所有好的恒温器应该做得那样进行工作。

一旦所有的人都去上班或上学，周围渐渐安静下来，屋内的温度又一次稳定下来。然而，这时一个自动的程序可能启动，命令恒温器降低温度以便节约电力和冷气，恒温器又回到工作中。在这一天的晚些时候，程序会再一次启动，这次将命令恒温器升高温度，保证家庭成员回来时，有一个温暖舒适的家迎接他们。

在晚上，随着房间里有了温暖的身体，以及来自烹饪的热量，恒温器需要做许多工作，高效地运转加热器和制冷器，以使室内温度保持恒温。

最后，在夜里，一切又回到一个安静的状态。

许多软件密集系统像恒温器一样运行。一个起搏器不间断地工作，但要适应血压或活动的变化。一个网络路由器不间断地工作，静静地引导异步的比特流，某些时候会调整它的行为以响应从网络管理者那里发来的命令。一个蜂窝网电话按需求工作，响应来自用户的输入和来自本地蜂窝网的消息。

在UML中，使用像类图和对象图这样的元素对系统的静态方面建模。你可以用这些图对存在于系统中的事物，包括类、接口、构件、节点、用况以及它们的实例，还有它们相互之间的关系，进行可视化、详述、构造和文档化。【在第二部分和第三部分中讨论对系统的结构方面建模。】

在UML中，使用状态图对系统的动态方面建模。交互是对共同完成某些动作的对象群体建模，而状态机是对单个对象的生命期（不管它是类、用况，还是整个系统的实例）建模。在一个对象的生命期中，可以出现各种各样的事件，如信号、操作的请求、对象的创建和撤销、时间推移、或某些条件下的改变。在响应这些事件时，对象通过某些动作作出反应，这些动作表示一个导致对象状态改变或一个值返回的原子计算。所以，这样一个对象的行为是受过去影响的。一个对象可以接收一个事件，并通过一个动作来响应，然后改变它的状态。一个对象还可以接收另一个事件，根据响应前一个事件产生的当前状态而作出不同的响应。【在第15章中讨论可以用交互来对系统的动态方面建模；在第20章中讨论事件。】

你可以使用状态机来对任何模型元素（通常是类、用况，或是整个系统）的行为建模。状态机可以用两种方式可视化。第一种是使用活动图，关注于发生在对象内部的活动。第二种是使用状态图，关注于按事件安排的对象行为，这种方式在对反应式系统建模时特别有用。【在第19章中讨论活动图；在第24章中讨论状态图。】

如图21-1所示，UML提供了对状态、转换、事件以及动作的图形表示。这种表示法允许你

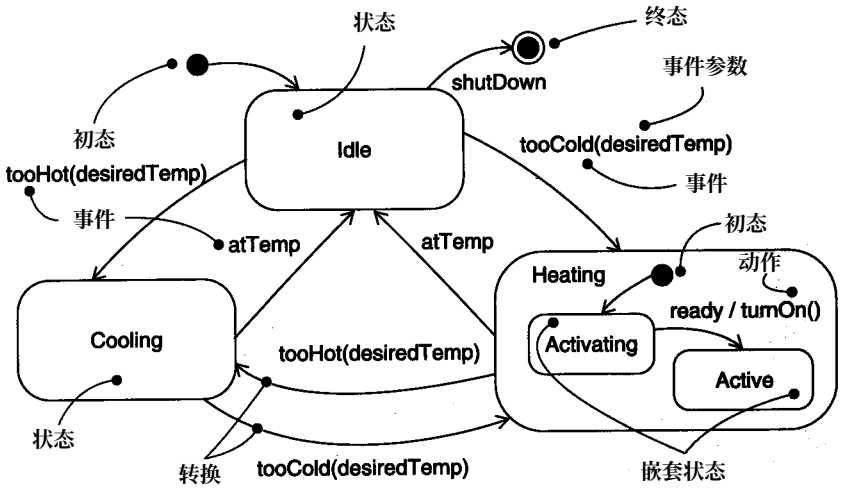

图21-1 状态机

以一种强调对象的生命期中的重要元素的方式来可视化该对象的行为。

21.2 术语和概念

一个状态机（*state machine*）是一个行为，它说明对象在它的生命期中响应事件所经历的状态序列以及它们对那些事件的响应。一个状态（*state*）是指在对象的生命期中的一个条件或状况，在此期间对象将满足某些条件、执行某些活动或等待某些事件。一个事件（*event*）是对一个在时间和空间上占有一定位置的有意义的事情的规格说明。在状态机的语境中，一个事件是一次激发的产生，激发能够触发一个状态转换。一个转换（*transition*）是两个状态之间的一种关系，它指明对象在第一个状态中执行一定的动作，并当特定事件发生或特定的条件满足时进入第二个状态。一个活动（*activity*）是状态机中进行的非原子执行。一个动作（*action*）是一个引起模型状态改变或值的返回的可执行的原子计算。在图形上，状态用圆角的矩形表示。转换用一条直的有向实线表示。

1. 语境

每个对象都有一个生命期。创建时，一个对象诞生；撤销时，一个对象终止退出。在两者之间，一个对象可以（通过发送消息）作用于其他的对象或（通过作为一个消息的目标）被作用。在许多情况下，这些消息将是简单的、同步的操作调用。例如，类 `Customer`的一个实例可能调用在类`BankAccount`的一个实例上的操作`getAccountBalance`。像这样的对象不需要一个状态机来说明它们的行为，因为它们当前的行为并不依赖于它们的过去。【在第13章中讨论对象；在第15章中讨论消息。】

在其他种类的系统中，你可能会遇到必须响应信号的对象，该信号是实例之间进行通信的异步激发。例如，一个蜂窝网电话必须响应随机的呼叫（来自其他电话）、按键事件（来自客户，开始一次电话呼叫）以及来自网络的事件（当电话从一个呼叫转到另一个呼叫时）。类似地，你还会遇到当前行为依赖过去行为的对象。例如，一个空对空导弹导航系统的行为依赖于它的当前状态，如`NotFlying`（当装载导弹的飞机还停在地上时，发射导弹并不是一个好主意）或`Searching`（在你知道要攻击的目标之前，不需要装备导弹）。【在第20章中讨论信号。】

如果对象的行为必须响应异步激发，或它的当前行为依赖于过去，那么它可以通过使用状态机得到最好地说明。这包括能够接收信号的许多类的实例，其中包括许多主动对象。事实上，一个接收信号但又没有状态机的对象，将会简单地忽略这个信号。你也可以使用状态机对整个系统的行为建模，尤其是对反应式系统，此类系统必须对来自系统外的参与者的信号作出响应。【在第22章中讨论主动对象；在第24章中讨论对反应式系统建模。】

> **注释** 多数时间里，你将使用交互对用况的行为建模，但是你也可以使用状态机来达到同样的目的。类似地，你可以用状态机来对接口的行为建模。尽管接口没有任何直接的实例，但实现该接口的类可以有实例。这样一个类必须符合该接口的状态机所说明的行为。【在第16章中讨论用况和参与者；在第15章中讨论交互；在第11章中讨论接口。】

2. 状态

状态是指在对象的生命期中满足某些条件、执行某些活动或等待某些事件时的一个条件或状况。对象在某一状态保持有限的时间。例如，房间中的一个“加热器” `Heater`可能有4种状

态：“空闲”Idle（等待开始加热房间的命令），“启动”Activating（热气打开，但等待达到某个温度），“活动”Active（热气和鼓风机都打开）和“关闭”ShuttingDown（热气关闭，但鼓风机打开，吹散系统的剩余热量）。

当一个对象的状态机处于一个给定的状态时，这个对象就被称作处于这个状态。例如Heater的一个实例可能处于Idle状态，或可能处于ShuttingDown状态。

【你可以在一个交互中可视化对象的状态，这在第13章中讨论；状态名在它的外围状态当中必是唯一的，与在第12章中讨论的规则一致；状态的后四部分在本章的后面介绍。】

一个状态有以下几个部分。

1）名称（name）	一个可以把该状态和其他状态区分开的字符串；状态可能是匿名的，即没有名称。
2）进入/退出动作（entry/exit action）	分别为进入和退出这个状态时所执行的动作。
3）内部转换（internal transition）	不导致状态改变的转换。
4）子状态（substate）	状态的嵌套结构，包括不相交（顺序活动）或并发（并发活动）子状态。
5）延迟事件（deferred event）	指在该状态下暂不处理，但将推迟到该对象的另一个状态下排队处理的事件列表。

注释 一个状态的名称中可以包含任意数量的字母、数字和某些标点符号（有些标点符号如冒号除外），并且可以连续几行。在实际应用中，状态名是取自你所建模的系统的词汇中的短名词或名词短语。通常，将状态中的每个单词的首字母大写，如 Idle或ShuttingDown。

如图21-2所示，用一个圆角的矩形代表一个状态。

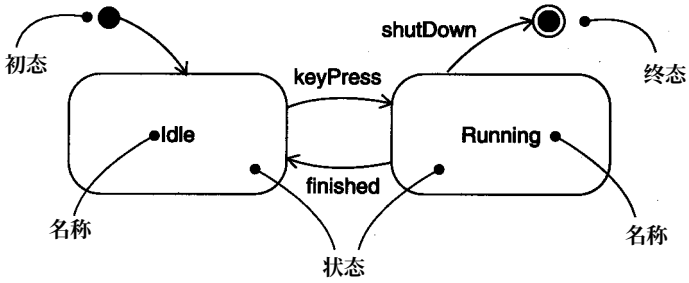

图21-2 状态

- 初态和终态

就像图中所示，在对象的状态机中有两个可能要定义的特殊状态。第一个是初态，表示该状态机或子状态的缺省开始位置。一个初态用一个实心的圆表示。第二个是终态，表示该状态机或外围状态的执行已经完成。一个终态用一个内部含有一个实心圆的圆圈表示。

注释 初态和终态实际上都是伪状态。它们除了名称外，不具有正规状态的通常部分。从一个初态到一个终态的转换可以有充分的特征补充，其中包括一个监护条件和动作

（但不包含触发事件）。

3. 转换

一个转换是两个状态之间的一种关系，表示对象将在第一个状态中执行一定的动作，并在某个特定事件发生而某个特定的条件满足时进入第二个状态。当状态发生这样的转变时，转换被称作激活了。在转换激活之前，称对象处于源状态；激活后，就称对象处于目标状态。例如，当像`tooCold`（带有参数`desiredTemp`）这样的事件发生时，“加热器”`Heater`可能从`Idle`状态转换到`Activating`状态。

一个转换由5部分组成。

1）源状态（source state） 即受转换影响的状态；如果一个对象处于源状态，当该对象接收到转换的触发事件或满足监护条件（如果有）时，就会激活一个离出的转换。

2）事件触发（event trigger） 是一个事件，源状态中的对象接收这个事件使转换合法地激活，并使监护条件满足。【在第20章中讨论事件。】

3）监护条件（guard condition） 是一个布尔表达式，当转换因事件触发器的接收而被触发时对这个布尔表达式求值；如果表达式取值为真，则激活转换；如果为假，则不激活转换，而且如果没有其他的转换被此事件所触发，则该事件丢失。

4）动作（action） 是一个可执行的原子计算，它可以直接的作用于拥有状态机的对象，并间接作用于对该对象是可见的其他对象。

5）目标状态（target state） 在转换完成后活动的状态。

如图21-3所示，一个转换用一条从源状态到目标状态的有向实线表示。自身转换是指源状态

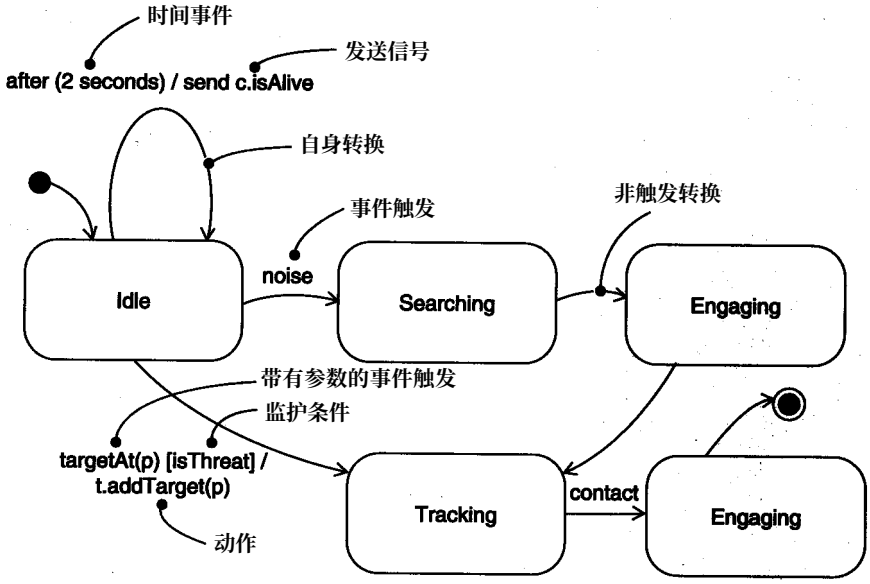

图21-3 转换

和目标状态相同的转换。

注释 一个转换可能会有多个源(在这种情况下,它表示来自多个并发状态的一个汇合),也可能会有多个目标(在这种情况下,它表示发往多个并发状态的一个分叉)。【在第19章中也讨论分叉和汇合。】

• 事件触发

一个事件是对一个在时间和空间上占有一定位置的有意义的事情的规格说明。在状态机的语境中,一个事件是一次激励的产生,激励能够触发一个状态转换。如前图所示,事件可以包括信号、调用、时间段或状态的一个改变。一个信号或一个调用可以带有参数,参数值可由包括监护条件和动作的表达式的转换所得到。【在第20章中讨论事件。】

还可能有非触发转换,由一个没有事件触发的转换表示。非触发转换也称作完成转换——当它的源状态已经完成它的活动时,它被隐式地触发。

注释 一个事件触发可能是多态的。例如,如果你定义了一个信号的族,那么,触发事件是S的转换,能被S触发,也能被S的任何子类触发。【在第20章中讨论详述一个信号的族;在第19章中讨论多个不重叠的监护条件形成一个分支。】

• 监护条件

如前图所示,一个监护条件由一个方括号括起来的布尔表达式表示,放在触发事件的后面。一个监护条件只在引起转换的触发事件发生后才被评估。所以只要那些条件不重叠,就可能会存在来自同一个源态、具有相同的事件触发的多个转换,。

对于每一个转换,一个监护条件只在事件发生时被评估一次,但如果该转换被重新触发,则监护条件会被再次评估。在布尔表达式中,可以包含关于对象状态的条件(例如,表达式`aHeater in Idle`如果对象`Heater`当前处于`Idle`状态,则它的值为真)。

注释 尽管一个监护条件只在每次对它的转换触发时才被评估一次,但变化事件是在暗中不断地被评估着的。【在第20章中讨论变化事件。】

• 动作

一个动作是一个可执行的原子计算。动作可以包括操作调用(调用拥有状态机的对象或其他可见的对象)、另一个对象的创建或撤销或者向一个对象的信号发送。如上图所示,对发送信号有一个特殊的表示法——在信号名前加一个关键字为`send`的前缀作为一个可视化提示。【在第15章中讨论动作。】

动作是原子的,这意味着它不能被事件中断,并因此一直运行到完成。而活动则相反,它可以被其他事件中断。【在本章后面讨论活动;在第5章和第10章中讨论依赖。】

注释 你可以用一个构造型为`send`的依赖来显式地显示一个信号发送到的对象,该依赖的源为状态,目标为这个对象。

4. 高级状态和转换

在UML中,你可以只使用状态和转换的基本特征来对广泛的、各种各样的行为建模。使用这些特征,最终可以产生简单的状态机,它意味着行为模型中除了弧(转换)和顶点(状态)

之外不包括其他东西。

然而，UML的状态机具有许多可以帮助你管理复杂行为模型的高级特征。这些特征可减少你需要的状态和转换的数量，并且将你在使用简单状态机时遇到的许多通用的和有点复杂的惯用法编集在一起。这些高级特征包括进入动作、退出动作、内部转换、活动和延迟事件。

• 进入和退出动作

在许多建模情况下，每当你进入一个状态时，不管是什么转换使你进入，你都是想要执行同一个动作。同样地，当你离开一个状态时，不管是什么转换使你离开，你也都是想执行同一个动作。例如，在一个导弹导航系统中，你可能想：每当它进入 Tracking状态时，显式地声明系统是onTrack的；每当它离开该状态时，声明系统是 offTrack的。使用简单的状态机，可以把那些动作放在每个进入和退出的转换上来达到这种效果。然而，这样也存在一些错误倾向；你不得不在每次添加一个新的转换时记住增加这些动作。而且，修改这个动作意味着你不得不触及每个相邻的转换。

如图21-4所示，UML为这种惯用法提供了简捷的表示。在状态符号中包括一个进入动作（以关键字事件entry标记）和一个退出动作（以关键字事件 exit标记），并与一个合适的动作在一起。每当你进入该状态时，就执行它的进入动作；每当你离开该状态时，就执行它的退出动作。

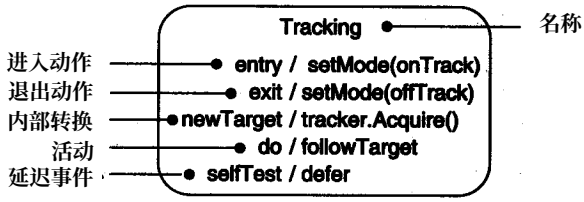

图21-4 高级状态和转换

注释 进入和退出动作不可以有参数或监护条件。然而，位于一个类的状态机顶层的进入动作可以有参数，用来表示当创建该对象时状态机接收到的参数。

• 内部转换

一旦处于一个状态内，你将遇到想要在不离开该状态的情况下处理的事件。这些被称为内部转换，它与自身转换有一些细微的不同。在一个自身转换中，像你在图 21-3所看到的，事件触发这个转换后，你就离开了这个状态，一个动作（如果有）被执行，然后你又重新进入同一个状态。因为这个转换先退出并且随后又进入该状态，所以，自身转换先执行该状态的退出动作，接着执行自身转换动作，最后执行该状态的进入动作。然而，假设你想处理一个事件，但并不想激发该状态的进入和退出动作。使用简单状态机就可以达到这个效果，但你将不得不努力记住哪个状态的转换中有这些进入和退出动作，哪个状态的转换中没有这些进入和退出动作。

如图21-4所示，UML提供这种惯用法的简捷表示（例如，对于事件 newTarget）。在状态符号中，可以包括一个内部转换（用一个事件来标记）。每当你处于该状态且该事件被触发时，其相应的动作被执行，而不必先离开然后重新进入该状态。所以，这个事件的处理没有执行状态的退出和进入动作。

注释 内部转换可以有带参数和监护条件的事件。所以，内部转换实质上是中断。

• 活动

当对象处于一个状态时，它一般是空闲的，在等待一个事件的发生。但是某些时候，你可能希望描述一个正在进行的活动。在处于一个状态的同时，对象做着某些工作，并一直继续直到被一个事件所中断。例如，如果一个对象处于 Tracking状态，只要它在该状态中，它就执行 followTarget。如图21-4所示，在UML中用特殊的do转换来描述执行了进入动作后在一个状态内部所做的工作。一个 do转换的活动可能命名另一个状态机（如 followTarget）。你也可以说明一个动作序列 —— 如do/op1（a）;op2（b）;op3（c）。动作是从不中断的，但动作序列是会中断的。在每两个动作之间（由分号分开），事件可能被一个外围状态处理，这个外围状态会导致离开此状态的转换。

• 延迟事件

考虑一个像 Tracking的状态。如图 21-3所示，假定只有一个离开这个状态、并由事件 contact触发的转换。当处于 Tracking状态时，除了事件 contact和那些由其子状态处理的事件，其他事件都将被丢失。也就是说事件可以发生，但它将被延迟，而且不因为该事件的出现而产生任何动作。【在第20章中讨论事件。】

在各种建模情况下，你可能想识别一些事件，而忽略另一些事件。识别那些作为转换的事件触发的事件；忽略那些刚刚离开的事件。然而，在某些建模情况下，你可能想要识别某些事件，但延迟对他们的响应，直到以后才执行。例如，当处于 Tracking状态时，你可能想要延迟对像selfTest这样的信号的响应，这些信号或许是由系统中的某些维护代理发送的。

在UML中，你可以用延迟事件来描述这种行为。延迟事件是事件的一个列表，这些事件在状态中的发生被延迟，直到激活了一个使这些列表事件不被延迟的状态，此时这些列表事件才会发生，并触发转换，就像它们刚刚发生一样。如你在上图中所看到的那样，用特殊的动作 defer列出事件来描述一个延迟事件。在这个例子中，事件 selfTest可能会在Tracking状态中发生，但它被延迟直到该对象处于 Engaging状态时才出现，就像刚刚发生一样。

注释 延迟事件的实现需要有一个内部事件队列。如果一个事件发生，并被列为延迟事件，则进入队列。一旦对象进入一个不延迟这些事件的状态，这些事件就会从这个队列中被除掉。

5. 子状态

这些状态与转换的高级特征解决了许多常见的状态机建模问题。然而， UML状态机还有另一个特征 —— 子状态 —— 它甚至能帮你简化复杂的建模行为。子状态是嵌套在另一个状态中的状态。例如，一个“加热器”Heater可能处于“加热”Heating状态，且在Heating状态中还有一个嵌套状态Activating。在这种情况下，应该称这个对象既处于 Heating状态，又处于 Activating状态。

简单的状态是没有子结构的状态。一个含有子状态（即嵌套状态）的状态被称作组合状态。组合状态包括并发（正交的）或顺序（不相交的）子状态。在 UML中，表示组合状态就像表示一个简单的状态一样，但还要用一个可选的图形框来显示一个嵌套状态机。子状态可以嵌套到

任何层次。【组合状态具有与组合相似的嵌套结构，这在第5章和第10章中讨论。】

• 顺序子状态

考虑对一个ATM的行为建模的问题。这个系统可能有3个基本状态：“空闲”Idle（等待与顾客交互）、“活动”Active（处理一个顾客的事务）和“维护”Maintenance（可能是再装满现金夹）。在Active状态下，ATM的行为沿一条简单路径执行：验证顾客，选择事务，处理这个事务，然后打印收据。打印之后，ATM机返回到Idle状态。可以把行为的这些阶段表示为状态Validating、Selecting、Processing和Printing。更理想的是，能让顾客在“验证”Validating账号之后和“打印”Printing最后收据之前选择和处理多种事务。

问题在于，在这个行为的任何阶段，顾客都可能决定取消事务，使ATM返回到它的“空闲”Idle状态。使用简单状态机可以表示这种情况，但十分麻烦。因为顾客可能随时在任意点取消事务，你不得不为Active序列中的每一个状态加入一个合适的转换。这是非常麻烦的，因为很容易忘记在所有适当的位置都包含这些转换，并且许多这样的中断事件意味着，你将以许多对准同一目标状态、来自不同的源、但带有相同的事件触发、监护条件和动作的转换而结束。

如图21-5所示，通过使用顺序子状态，可用一个简单的方法来对这个问题建模。这里，Active有一个子结构，包括子状态Validating、Selecting、Processing和Printing。当顾客将信用卡插入ATM机时，ATM的状态从Idle转换到Active。在进入Active状态时，执行进入动作readCard。从子结构的初态开始，控制从Validating状态传送到Selecting状态，再到Processing状态。在Processing状态之后，控制可能返回到Selecting状态（如顾客选择另一个事务）或可能转移到Printing状态。在Printing状态之后，有一个非触发转换返回到Idle状态。注意，Active状态有一个退出动作来退出顾客的信用卡。

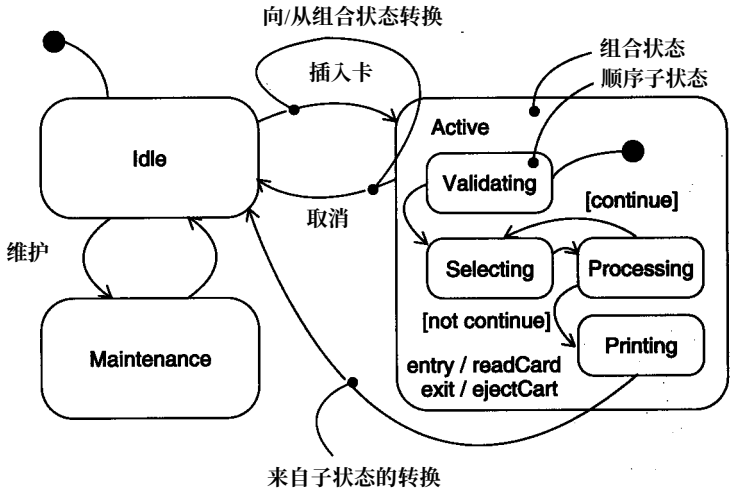

图21-5 顺序子状态

注意，从Active状态到Idle状态的转换也是由cancel事件触发的。在Active的任何子状态中，顾客都可能取消这个转换，并使ATM返回到Idle状态（但只是在退出顾客的信用卡之

后，它是离开Active状态时执行的退出动作，不管是什么原因导致了离开该状态的转换都是如此）。如果没有子状态，在每个子结构状态中，你都将需要一个由 cancel触发的转换。

像Validating和Processing这样的子状态，被称作是顺序的或不相交的子状态。在一个封闭的组合状态的语境中给定一组不相交的子状态，对象被称为处在该组合状态中，且一次只能处于这些子状态（或终态）中的一个子状态上。因此，顺序子状态将组合状态的状态空间分为不相交的状态。

从一个封闭的组合状态外边的一个源状态出发，一个转换可以以组合状态为目标，也可以以一个子状态为目标。如果它的目标为一个组合状态，则这个嵌套状态机一定包括一个初态，以便在进入组合状态后或执行它的进入动作（如有）后，将控制传送给初态。如果它的目标是一个嵌套状态，在执行组合状态的进入动作（如有）和子状态的进入动作后，将控制传送给嵌套状态。

一个导致离开组合状态的转换，可能以一个组合状态或一个子状态来作为它的源。在每种情况下，控制都是首先离开嵌套状态（如有则执行它的退出动作），然后离开组合状态（如有则执行它的退出动作）。一个源是组合状态的转换本质上切断（中断）了这个嵌套状态机的活动。

注释 一个嵌套的顺序状态机最多有一个初态和一个终态。

· 历史状态

状态机描述对象的动态方面，该对象的当前行为依赖于过去。从效果上看，一个状态机描述一个对象在它的生命周期中所经过的合法的状态序列。

除非特别说明，当一个转换进入一个组合状态时，嵌套状态机的动作就又处于它的初态（当然，除非这个转换的目标直接指向一个子状态）。然而，在许多情况下，你对一个对象建模，想要它记住在离开组合状态之前最后活动着的子状态。例如，在对一个通过网络进行计算机备份的代理的行为建模时，如果它曾被中断（例如，被一个操作员的查询中断），你希望它记住是在该过程的什么地方被中断的。

使用简单的状态机也可以对这个问题建模，但显得很麻烦。对于每一个顺序子状态，你都需要让它的退出动作给局部于组合状态的某些变量赋一个值。然后这个组合状态的初态将需要一个到每个子状态的转换，它带有一个监护条件来查询变量。采用这种方法，离开组合状态将导致最后的子状态被记住；进入组合状态将转换到合适的子状态。这是十分麻烦的，因为它要求你记住去触及每个子状态，并设置一个适当的退出动作。它留给你这样一大堆转换：这些转换是从同一个初态出发，到带有非常类似（但不同）的监护条件的不同的目标子状态。

在UML中，对这种惯用法建模的一个比较简单的方法是使用历史状态。一个历史状态允许一个组合状态包含顺序子状态，以记住来自组合状态的转换之前的最后活动着的子状态。如图21-6所示，用一个内部包含符号H的小圆圈来表示一个浅的历史状态。

如果你想用一个转换来激活最后的子状态，就显示来自这个组合状态的外边而直接指向该历史状态的一个转换。当第一次进入一个组合状态时，它没有历史。这也就是从历史状态到一个顺序子状态（如Collecting）的单个转换的含义。这个转换的目标描述了嵌套状态机被首次进入时的初态。接下去，假定处于 BackingUp和Copying状态中，事件query被发出。控制离开Copying和BackingUp（按照需要执行它们的退出动作），并返回到Command状态。当

Command的动作完成后，非触发转换返回到组合状态BackingUp的历史状态。这次，因为有了这个嵌套状态机的一个历史，控制传回 Copying状态 —— 绕过了Collecting状态 —— 因为Copying是来自Backingup的转换之前的最后一个活动子状态。

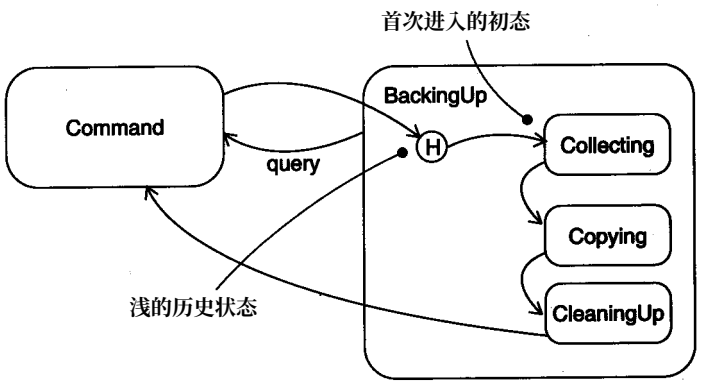

图21-6 历史状态

> 注释 符号H表示一个浅历史，只记住这个直接嵌套的状态机的历史。你也可以说明深历史，用包含符号H*的小圆圈表示。深历史会在任何深度上记住最深的嵌套状态。如果你仅有一层嵌套，那么深的和浅的历史状态在语义上是等同的。如果你有多于一层的嵌套，那么浅历史只会记住最外层的嵌套状态；而深历史则可以在任何深度上记住最深层的嵌套状态。

不论那种情况，如果一个嵌套状态机到达一个终态，则会丢失它储存的历史，就像它未曾第一次进入一样。

· 并发子状态

顺序子状态是一种最常见的嵌套状态机。然而，在某些建模情况下，你可能想要描述并发子状态。这些子状态使你可以说明在一个封闭的对象的语境中并发执行的两个或多个状态机。

> 注释 对并发建模的另一种方式是使用主动对象。这样，不是把一个对象的状态机分成两个（或更多）并发子状态，而是定义两个主动对象，每个主动对象负责一个并发子状态的行为。如果这些并发流中的一个流的行为被其他流的状态所影响，则使用并发子状态来对之建模。如果这些并发流中的一个流的行为被送往和来自其他流的消息所影响，则使用主动对象来对之建模。如果并发控制流之间很少或没有通信，那么使用哪种表示完全是凭兴趣的事，尽管多数时间里使用主动对象会使你的设计决策更显而易见。【在第22章中讨论主动对象。】

例如，图21-7显示了对图21-5中的Maintenance状态的一个扩展。Maintenamce被分解为两个并发子状态Testing和Commanding，它们在Maintenance状态中嵌套显示、并用一条虚线分开。这些并发子状态中的每个并发子状态都被进一步分解为顺序子状态。当控制从Idle状态传送到Maintenance状态时，控制就分叉为两个并发流 —— 这个封闭的对象将处在

Testing状态和Commanding状态中。而且，当处于Commanding状态时，这个封闭的对象还将处于Waiting或Command状态。【第19章中讨论分叉和汇合。】

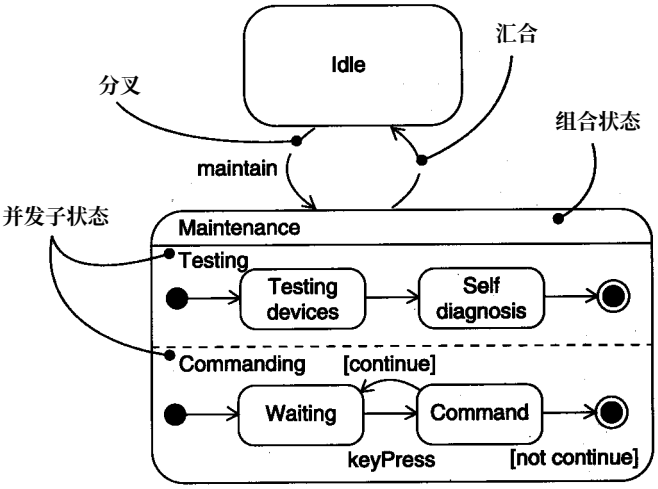

图21-7 并发子状态

注释 这就是顺序子状态和并发子状态的区别所在。在同一层次给出两个或更多的顺序子状态，对象将处于这些子状态中的一个子状态或另一个子状态中。而在同一层次给出两个或更多的并发子状态，对象将处于来自每一个并发子状态的一个顺序状态中。

这两个并发子状态的执行是并发的。最终，每个嵌套状态机都到达它的终态。如果一个并发子状态先于另一个到达它的终态，那么先到的子状态的控制将在它的终态等待。当两个嵌套状态机都到达它们的终态时，来自两个并发子状态的控制就汇合成一个流。

每当一个转换所到的组合状态被分解为多个并发子状态时，控制就分成与并发子状态一样多的并发流。类似地，每当一个转换来自一个可被分解为多个并发子状态的组合状态时，控制就汇合成一个流。这在所有情况下都成立。如果所有的并发子状态都到达它们的终态，或者有一个离开封闭的组合状态的显式的转换，那么，控制就重新汇合成一个流。

注释 嵌套的并发状态机没有初态、终态或历史状态。但是，组成一个并发状态的顺序子状态可以具有这些特征。

21.3 普通建模技术

对对象的生命期建模

使用状态机最通常的目的是对对象（尤其是类、用况和整个系统的实例）的生命期建模。交互对一起工作的对象群体的行为建模，而状态机对单个对象的整个生命期的行为建模，就像

你使用用户接口、控制器和设备所发现的那样。【在第13章中讨论对象；在第4章和第9章中讨论类；在第16章中讨论用况；在第31章中讨论系统；在第15章中讨论交互。】

当对对象的生命周期建模时，主要描述以下3种事物：对象能响应的事件、对这些事件的响应、以及过去对当前行为的影响。对对象的生命期建模，还包括决定该对象有意义地响应事件的顺序，从对象的创建时开始，一直到它被撤销。

对对象的生命周期建模，要遵循如下的策略：

- 设置状态机的语境，不管它是类、用况，或是整个系统。
  1）如果语境是一个类或一个用况，则收集相邻的类，包括这个类的所有父类和通过依赖或关联到达的所有类。这些邻居是动作的候选目标或在监护条件中包含的候选项。
  2）如果语境是整个系统，则将你的注意力集中到这个系统的一个行为上。理论上，系统中每个对象都可以是系统的生命周期的模型中的一个参加者，而且除了最微小的系统，建立一个完整的模型将是非常棘手的。【在第27章中讨论协作。】
- 建立这个对象的初态和终态。为了指导模型的剩余部分，可能的话，分别声明初态和终态的前置和后置条件。【在第10章中讨论前置和后置条件；在第11章中讨论接口。】
- 决定这个对象可能响应的事件。如果已经说明，你将在对象的接口处发现这些事件；如果还没说明，就要考虑在你的语境中哪个对象可能与该对象交互，然后发现它们可能发送哪些事件。
- 从初态开始到终态，列出这个对象可能处于的顶层状态。用被适当的事件触发的转换将这些状态连接起来，接着向这些转换中添加动作。
- 识别任何进入或退出的动作（尤其当你发现它们所覆盖的惯用法被用于状态机时）。
- 如果需要，通过使用子状态来扩充这些状态。
- 检查在状态机中提到的所有事件是否和该对象接口所期望的事件相匹配。类似地，检查该对象的接口所期望的所有事件是否都被状态机所处理。最后，找出明显地想忽略这些事件的地方。
- 检查在状态机中提到的所有动作是否被闭合对象的关系、方法和操作所支持。
- 通过状态机（不管是手动还是通过工具）跟踪，以便再次检查事件的顺序和它们的响应。尤其要努力地寻找那些未达到的状态和导致机器停止的状态。
- 在重新安排状态机后，按所期望的顺序再一次检查，以确保你没有改变该对象的语义。

例如，图21-8描述了用于家庭安全系统中控制器的状态机，它负责监视房间周围的各种传感器。

在这个控制器类的生命期中有4个主要的状态：“初始化”Initializing（控制器开始运行）、“空闲”Idle（控制器准备好，并等待警报或来自用户的命令）、“命令”Command（控制器正在处理来自用户的命令）和“活动”Active（控制器正在处理一个警报条件）。当第一次创建这个控制器对象时，首先进入Initializing状态，然后无条件地进入Idle状态。这两个状态的详细信息并不显示，但要显示Idle状态中带时间事件的自转换。这种时间事件在嵌入式系统中是常见的，它通常有一个心跳定时器，每隔一段时间就检查一下系统的健康状况。

当接收到一个“警报”alarm事件（包括参数s，识别发生错误的传感器）时，控制从

Idle状态传送到Active状态。进入Active状态时，setAlarm被作为进入动作执行，控制首先传送到Checking状态（验证这个警报），然后传送到Calling状态（呼叫警报公司以登记这个警报），最后传送到Waiting状态。仅当“清除”clearing警报时，或是用户向控制器发“注意”attention信号以通知可能发布一个命令时，状态Active和Waiting才退出。

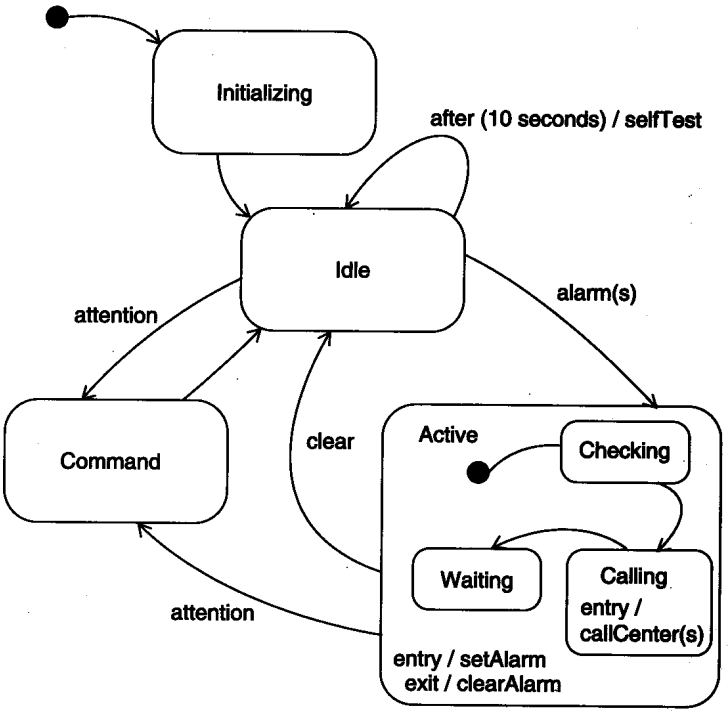

图21-8 对对象的生命周期建模

注意这里没有终态。这在嵌入式系统中也是常见的，希望系统不间断地运行。

21.4 提示和技巧

在UML中对状态机建模时，要记住每个状态机表示的是单个对象的动态方面，该对象通常代表类、用况，或整个系统的实例。一个结构良好的状态机，应满足如下的要求：

- 是简单的，因而不包含任何多余的状态或转换。
- 具有清晰的语境，可以访问所有对闭合对象可见的对象（仅当对执行由状态机描述的行为是必须的时候才使用这些邻近对象）。
- 是有效的，因而应该根据执行动作的需要，取时间和资源的最优平衡来完成它的行为。
- 是可理解的，因而应该使用来自系统的词汇中的词汇，来命名它的状态和转换。【在第4章中讨论对系统的词汇建模。】

- 不要太深层地嵌套（一层或两层的嵌套子状态能够解决大多数复杂行为）。
- 像使用主动类那样节约地使用并发子状态常常是更好的选择。

当在UML中绘制状态机时，要遵循如下的策略：

- 避免交叉的转换。
- 只在为使图形可理解所必须的地方扩充组合状态。