# 1.什么是有限状态机

有限状态机，常常被称作FSM（Finite State Machine），多年来已经作为人工智能编程者们选用的工具用于设计具有智能幻觉的游戏智能体。你会发现从视频游戏的早期开始，这种或那种FSM正是每个游戏所选中的架构；尽管更专业的智能体结构越来越普及，但FSM架构还将在今后很长时间内无处不在。为何会这样？原因如下：

* 编程快速简单。有很多方法编码一个有限状态机，并且几乎所有的有限状态机实现都相当的简单。
* 易于调试。因为一个游戏智能体的行为被分解成简单的易于管理的块，如果一个智能体开始变得行动怪异，会通过对每一个状态增加跟踪代码来调试它。用这种方法，人工智能程序员可以很容易跟踪错误行为出现前的事件序列，并且采取相应的行动。
* 很少的计算开销。有限状态机几乎不占用珍贵的处理器时间，因为它们本质上遵守硬件编码的规则。除了if-this-then-that类型的思考处理之外，是不存在真正的“思考”的。
* 直觉性。人们总是自然地把事物思考为处在一种或另一种状态。并且我们也常常提到我们自己处在这样那样的状态中。有多少次你“使自己进入一种状态”或者发现自己处于“头脑的正确状态”，当然人类并不是像有限状态机一样工作，但是有时候我们发现在这种方式下考虑我们的行为是有用的。相似地，将一个游戏智能体的行为分解成一些状态并且创建需要的规则来操作它们是相当容易的。出于同样的原因，有限状态机能够使你很容易地与非程序员（例如与游戏制片人和关卡设计师）来讨论你的人工智能的设计，能够更好地进行设计概念的沟通和交流。
* 灵活性。一个游戏智能体的有限状态机可以很容易地由程序员进行调整，来达到游戏设计者所要求的行为。同样通过增添新的状态和规则也很容易扩展个智能体的行为的范围。此外，当你的人工智能技术提高了，你会发现有限状态机提供了一个坚固的支柱，使你可以用它来组合其他的技术，例如模糊逻辑和神经网络。

历史上来说，有限状态机是一个被数学家用来解决问题的严格形式化的设备。最著名的有限状态机可能是艾伦·图灵假想的设备——图灵机，他在1936年论文《关于可计算数字》中写道：这是一个预示着现代可编程计算机的机器，它们可以通过对无限长的磁带上的符号进行读写和擦除操作来进行任何逻辑运算。

幸运的是，作为一个人工智能程序员，我们可以放弃有限状态机的正式的数学定义，一个描述性的定义就足够了：

**一个有限状态机是一个设备，或是一个设备模型，具有有限数量的状态，它可以在任何给定的时间根据输入进行操作，使得从一个状态变换到冗一个状态，或者是促使一个输出或者一种行为的发生。一个有限状态机在任何瞬间只能处在一种状态。**

因此，有限状态机背后的概念是要把一个对象的行为分解成为易于处理的“块”或状态。例如，在你墙上的灯的开关，是一个非常简单的有限状态机。它有两种状态：开或关。状态之间的变换是通过你手指的输入产生的。向上按开关，产生从关到开的状态变换，向下按开关，产生从开到关的状态变换。

关闭状态没有相关的输出或行动（除非你考虑灯泡不亮也作为一个行动），但是当它处在开状态时，允许电流流过开关并且通过电灯泡罩的灯丝点亮你的房间，见图1。



**图1 一个电灯开关是一个有限状态机**

当然，一个游戏智能体的行为常常要比一个电灯泡复杂的多。下面是一些关于有限状态机是如何应用在游戏中的例子：

* 在Pac-Man中幽灵行动的实现就是一个有限状态机。存在一个规避状态，这对所有的幽灵都是相同的，并且每个幽灵都有自己的追踪状态，追踪行动的实现对于每个幽灵来说都是不同的。玩家吃了一个强力药丸的输入就是从追踪状态变换为规避状态的条件。
* 类似雷神之槌中的角色类型也可以作为一个有限状态机来实现。它们具有诸如找到武器、找到血包、寻求掩护和走开的状态。甚至雷神之槌中的武器也实现了它们自己的小型有限状态机。例如，火箭可以实现如移动、接触物体和消失的状态。
* 运动员仿真例如足球游戏FIFA2002，是作为有限状态机来实现的。他们具有状态例如碰撞、运球、追球和盯住对方队员。此外，球队本身也常常作为FSM来实现，并且具有状态如开球、防卫，或者退场。
* NPC们（非玩家角色）在RTS（实时策略游戏）中，例如Warcraft也利用有限状态机。他们具有移动到位、巡逻和沿路经前进等状态。

## 有限状态机的实现

有一些方法来实现有限状态机。一种幼稚的方法就是使用一系列的if-then语句或者对switch语句稍加整理。使用一个具有枚举类型的switch语句来表达状态的代码如下所示。

enum StateType { RunAway, Patrol, Attacki};

public void UpdateState (StateType CurrentState)

{

 switch (CurrantState)

 case StateType.RunAway :

 EvadeEnemy ();

 if (Safe ())

 ChangeState (StateType.Patrol) ;

 break;

 case StateType.Patrol:

 FollowPatrolPath ();

 if (Threatened ())

 {

 if(StrongerThanEnemy())

 {

 ChangeState(StateType.Attack);

 }

 else

 ChancjeState (StateType.RunAway) ;

 }

 break;

 case StateType.Attack :

 if (WeakerThanEnemy ())

 {

 ChangeState (StateType.RunAway) ;

 }

 else

 {

 BashEnemyOverHead();

 }

 break;

 } //switch结束

}

虽然初看之下，这个方法是合理的，但当实际应用到任何一个比最简单的游戏对象稍复杂的情况，switch/if-then解决方法就变成了一个怪物，它潜伏在阴影中等待着发难。随着更多的状态和条件被加入，这种结构就像意大利式面条一样跳转得非常快，使得程序流程很难理解并且产生一个调试噩梦。此外，它不灵活，难以扩展，而扩展在实际编程时常常需要的。当你第一次计划状态机时，除非你设计一个有限状态机来完成非常简单的行为（或者你是一个人才），你几乎肯定会发现，在设计的行为实现你认为将要得到的结果之前，你总要调整智能体来应对计划外的情况。

此外，作为一个人工智能编程者，当处于初始进入状态或者退出状态时，你会常常需要一个状态完成一个指定的行动（或多个行动）。例如，当一个智能体进入逃跑状态，你可能会希望它向空中挥动着胳膊并且喊道“啊！”；当它最后逃脱了并且改变成为巡逻状态，你可能希望它发出一声叹息，擦去额头的汗水，并且说“呦！”。这些行为只能是在进入或退出逃跑状态时出现的，而不会发生在通常的更新步骤中。因此，这个附加的函数必须被理想地建立在你的状态机架构中。要想在switch或if-then架构中做到这些，你必定会咬牙切齿，恶心反胃，并写出非常糟糕的代码。

### 状态变换表

一个用于组织状态和影响状态变换的更好的机制是一个**状态变换表**。就如名称所示的那样：它是一个条件和那些条件导致的状态的表。下表显示的例子是前面所述例子的状态和条件映射图。

**一个简单的状态变换表**

|  |  |  |
| --- | --- | --- |
| 当前状态 | 条件 | 状态转换 |
| 逃跑 | 安全 | 巡逻 |
| 攻击 | 比敌人弱 | 逃跑 |
| 巡逻 | 受到威胁并比敌人强 | 攻击 |
| 巡逻 | 受到威胁并比敌人弱 | 逃跑 |

这个表可以被一个智能体在规则的间隔内询问，使得它能基于从游戏环境中接受到的刺激进行必需的状态转换。每个状态可以模型化为一个分离的对象或者存在于智能体外部的函数，提供了一个清楚的和灵活的结构。这个表与前面讨论的if-then/switch方法相比，将少有出现意大利面条式的失控情况。

有人曾经告诉我一个生动且天真形象化的方法，能够帮助人们来理解这样一个抽象概念。让我们看它是否有效。

假想一个机器人小猫。它闪闪发光、娇小可爱，长着金属的胡子并且在它的胃罩有个插槽，可以插入模块（用以控制它的状态）。这些模块都是用逻辑编程的，使小猫完成特殊的行动集。每个行动集编码一个不同的行为。例如，“玩线绳”、“吃鱼”，或者“趴在地毯上”。如果没有一个模块，小猫就会是个死气沉沉的金属雕塑品，只会像金属米老鼠那样静坐着，看起来很可爱。

小猫是非常灵巧的，并且它具有自治能力来调换模块，我们只要指示它这样去做。通过提供指示何时应该转换模块的规则，就可以把能够产生所有类型的有趣复杂的行为的模块的插入序列串在一起。这些规则被编程放在置于小猫头内的一个微小的芯片上，与我们前面讨论过的状态转换表类似。芯片与小猫的内部函数交流来找到必要的信息以处理规则（例如小猫有多么饿或它觉得有多么好玩）。

因此，状态转换芯片可以用类似这样的规则来编程：

IF Kitty\_Hungry ANDNOT Kitty\_Playful

SWITCH\_CARTRIDGE eat\_fish

所有在表中的规则在每个时间步进中都进行测试并且将命令送给小猫来相应地转换模块。

这类结构非常灵活，可以容易地通过增加新的模块来扩展小猫的指令表。每当—个新的模块被加入的时候，主人只需要用螺丝起子打开小猫的头部来取出芯片并重新编程状态转换规则。无需干扰任何其他的内部电路。

### 内置的规则

另一种方法就是将状态转换规则嵌入到状态本身的内部。应用这个概念到机器小猫，就是省掉状态变换芯片而把规则直接嵌入模块。例如，模块“玩线绳”可以监控小猫饥饿的等级并且当它感到饥饿等级上升时指导它转换状态到“吃鱼”。“吃鱼”模块接着可以监控小猫的碗并且当它感到排便的等级达到危险的高度时，指导它转换到“存地毯上排便”。

虽然每个模块可以意识到任何其他模块的存在，但每一个模块是一个独立的单位并不依赖任何外部的逻辑来决定它是否应该允许自己变换到一个替代状态。因此增加状态或者用一个完全新的集合来交换整个的模块集是简单易行的（可能是使得小猫的行为像个猛禽的新集合）。这样就无需再用螺丝刀打开小猫的头，只要改变一些模块本身即可。

让我们看看在一个视频游戏的环境中这个方法是怎样实现的。就像小猫的模块一样，状态被封装成对象，包含推动状态变换需要的逻辑。此外，所有的状态对象共享一个称为State的抽象类。

public abstract class State

{

public abstract void Execute(Troll troll);

}

现在想象一个TroIl（巨魔）类，它具有成员变量表示特征，例如健康、发怒、体力等，还有方法允许客户获取或设置那些值。一个Troll类可以被赋予有限状态机的功能性，只要增加一个指向State类继承类的对象和允许用户改变对象的方法。

class Troll

{

State \_currentState;

public void Update()

{

\_currentState.Execute( this);

}

public void ChangeState(State newState)

{

\_currentState=newState;

}

}

当Troll类的更新方法被调用时，它反过来用this指针调用当前状态类型中的Execute方法。当前的状态就可以使用这个this对象获取对象的信息，来调整对象的属性，或者产生一个状态转换。换句话说，一个Troll类当更新时有怎样的行为可以完全依赖于它当前状态的逻辑。用实例可以最佳地阐明这一点，让我们创建一对状态，使troll在感到危险时从敌人身边逃跑，并且当它感到安全时变为睡觉。

//-------- -------逃跑状态

class State\_RunAway : State

{

public void Execute(Troll troll)

if(troll.isSafe )

troll.ChangeState(new StateSleep());

else

troll.MoveAwayFromEnemy();

}

//-------- -------睡觉状态

class State\_Sleep : State

{

public void Execute(Troll troll)

if(troll.isThreatened)

troll.ChangeState (new State\_RunAway () );

else

troll.Snore() ;

}

正如你看到的，当更新时，一个troll将依赖于状态中\_currentStatc指向哪里而产生不同的行为。两个状态都作为对象封装并且它们都给出了影响状态变换的规则。一切都很简洁严谨。

这个结构被称为**状态设计模式**，它提供了一种优雅的方式来实现状态驱动行为。尽管这偏离了数学形式化的FSM，但它是直觉的，编码简单，并且很容易扩展。它也可以非常容易地为每个状态增加进入和退出的动作：你需要做的所做的事就是创建Enter和Exit方法并相应地调整智能体的ChangeState方法。