有限状态机（Finite-state machine, FSM）又称有限状态自动机，简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。

**Finite State Machines(有限状态机）**

1. 基本节点是状态。他包含了一系列运行在该状态的行为以及离开这个状态的条件。  
2. 这是图。状态可以任意跳转,实现简单,但是对于大的状态机很难维护.状态逻辑的重用性低.  
3. 每一个状态的逻辑会随着一些新状态的增加而越来越复杂。维持状态的数量和状态逻辑复杂性是一个很大的难点。需要合理的分割以及重用状态。  
4. 状态机状态的复用性很差，一旦一些因素变化导致这个环境发生变化。你只能新增一个状态，并且给这个新状态添加连接他以及其他状态的跳转逻辑  
5. 状态机的跳转条件一旦不满足，就会一直卡在某一个状态（整个状态机就会卡住）

**Hierarchical FSM(层次状态机)**

1. 基本的概念和状态机一样。

2. 可以将一些状态节点的归结成一个超级状态（Super-States），共享一些状态跳转逻辑（Generalized Transitions）。相对于状态机，它主要提供了可重用得跳转条件。  
3. 绝大多数层次状态机设计的时候，每一个子状态通常只包含在一个Super-States里面。(如果A,B两个Super-State都包含相同的子状态，则A,B分别添加两个不同的子状态实例)  
4. 超级状态跳转依然需要考虑很多不同的子状态的情况。手动的修改这些状态跳转是非常恶心的一件事情。

5. 状态模块化很差，你很难在不修改代码的情况下完成新逻辑

**Behaviour Tree（行为树）**

**一个流行的AI技术，涵盖了层次状态机，事件调度，事件计划，行为等一系列技术**  
1.高度模块化状态，去掉状态中的跳转逻辑，使得状态变成一个“行为”。

2.”行为”和”行为”之间的跳转是通过父节点（Composite）的类型来决定的（例如sequence或者selector) 。比如并行处理两个行为，在状态机里面无法同时处理两个状态。

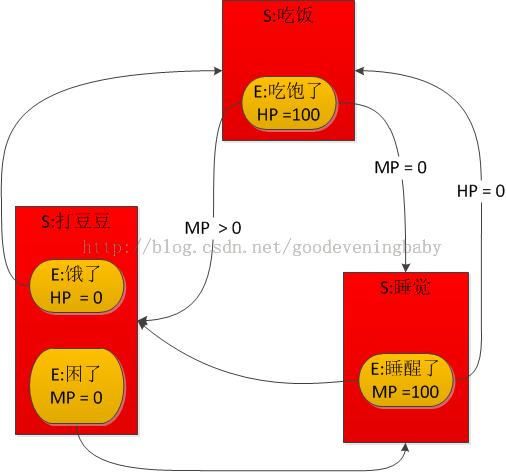
3.通过增加控制节点的类型，可以达到复用行为的目的。

4. 可视化编辑

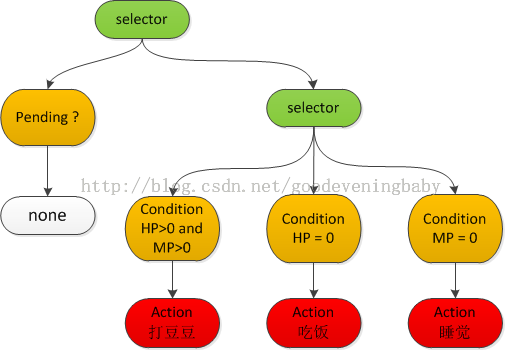
有限状态机和行为树举个粗糙的例子来比较一下两者的不同：

AI行为：吃饭 睡觉 打豆豆(很消耗体力和脑力的;)  
  
1.打豆豆 HP -= 5 / 秒    MP -= 3 / 秒  
2.吃饭 HP += 10/秒  MP -= 1 / 秒  
3.睡觉 MP += 15/秒  HP -= 2/秒   
4.吃饭和睡觉是不可打断的动作(pending)，必须执行到吃饱(HP = 100) or 睡饱(MP = 100)  
5.打豆豆是瞬发动作,每帧都可以执行一次

状态机的实现逻辑图：



行为树的实现逻辑图：



其实不管你知不知道什么是selector,condition都不要紧，至少从上图，应该可以看出来，行为树节点间的联系并不像状态机那样得“紧密”。

选择两种不同的ai实现方法，也决定了具体行为的实现逻辑。

比如对于sleep动作的实现，如果是状态机：

function sleep() = if Y == 100 then AwakeEvent() return end HP -= X MP += Y end然后每一帧执行sleep()  
  
  
如果是选择行为树:  
function sleep() local sleepTime = (100/15)--不好意思乱入了一段cocos2dx的代码 self:runAction(cc.Sequence:create(cc.DelayTime:create(sleepTime),cc.CallFunc:create(cancelPending))) local cancelPending = function() pending = false end end

罗列一下行为树的概念：

对于有限状态机而言，必须明确 状态的转换方式；对于行为树，必须明确状态前提：前提条件  
每一个行为必须有“前提条件” ，这决定了该行为是否被选择。  
行为树的运算也是通过帧循环的update来驱动，不一定是每帧都update，但是要周期性update。  
每一次run从根节点(root)开始，每一运行都会选择一个可行的子节点运行，这种选择可以是随机方式，也可以是预设好优先条件  
行为树由叶子节点和中间节点组成，叶子节点是最基本的行为(如跑动，攻击)，中间节点代表逻辑单元(巡逻，逃跑)。  
当一个叶子节点被选择后，就会激活其对应的基本的行为  
最基本的行为可能执行成功也可能失败。  
高等级的行为（中间节点）是否执行成功依赖于他们的孩子节点是否执行成功。  
一个子节点失败可能导致父母节点选择另外一个孩子。  
除了选择(selector)一个单独的子节点行为，一个节点还可能顺序(sequence)or并行(concurrent)得运行他的所有子节点。  
一个行为除了有前提条件，可能还有上下文条件(父节点or孩子节点可能存储一定的状态变量)。  
高优先级的行为可能抢占低优先级的行为

 行为树(Behavior Tree)，有4大类型的Node：

(1) **Composites Node**组合节点，包括经典的：Sequence，Selector，Parallel

 \* Selector Node  
    当执行本类型Node时，它将从begin到end迭代执行自己的Child Node：  
    如遇到一个Child Node执行后返回True，那停止迭代，  
    本Node向自己的Parent Node也返回True；否则所有Child Node都返回False，  
    那本Node向自己的Parent Node返回False。  
      
  \* Sequence Node  
    当执行本类型Node时，它将从begin到end迭代执行自己的Child Node：  
    如遇到一个Child Node执行后返回False，那停止迭代，  
    本Node向自己的Parent Node也返回False；否则所有Child Node都返回True，  
    那本Node向自己的Parent Node返回True。  
    
  \* Parallel Node  
    并发执行它的所有Child Node。  
    而向Parent Node返回的值和Parallel Node所采取的具体策略相关：  
Parallel Selector Node: 有一个子节点True返回True，否则返回False。  
Parallel Sequence Node: 有一个子节点False返回False，否则返回True。  
Parallel Fall On All Node: 所有子节点False才返回False，否则返回True。  
Parallel Succeed On All Node: 所有子节点True才返回True，否则返回False。  
Parallel Hybird Node: 指定数量的子节点返回True或False后，才决定结果。

    Parallel Node提供了并发，提高性能。  
    不需要像Selector/Sequence那样预判哪个Child Node应摆前，哪个应摆后，  
    常见情况是：  
    (1)用于并行多棵Action子树。  
    (2)在Parallel Node下挂一棵子树，并挂上多个Condition Node，  
       以提供实时性和性能。  
    Parallel Node增加性能和方便性的同时，也增加实现和维护复杂度。

  PS：上面的Selector/Sequence准确来说是Liner Selector/Liner Sequence。  
  AI术语中称为strictly-order：按既定先后顺序迭代。  
    
  Selector和Sequence可以进一步提供非线性迭代的加权随机变种。  
  Weight Random Selector提供每次执行不同的First True Child Node的可能。  
  Weight Random Sequence则提供每次不同的迭代顺序。  
  AI术语中称为partial-order，能使AI避免总出现可预期的结果。

(2) **Decorator Node**装饰节点，顾名思义，就是为仅有的一个子节点额外添加一些功能，比如让子task一直运行直到其返回某个运行状态值，或者将task的返回值取反等等

(3) **Actions Node**行为节点，行为节点是真正做事的节点，其为叶节点。Behavior Designer插件中自带了不少Action节点，如果不够用，也可以编写自己的Action。一般来说都要编写自己的Action，除非用户是一个不懂脚本的美术或者策划，只想简单地控制一些物件的属性。

(4) **Conditinals Node**条件节点 ，用于判断某条件是否成立。目前看来，是Behavior Designer为了贯彻职责单一的原则，将判断专门作为一个节点独立处理，比如判断某目标是否在视野内，其实在攻击的Action里面也可以写，但是这样Action就不单一了，不利于视野判断处理的复用。一般条件节点出现在Sequence控制节点中，其后紧跟条件成立后的Action节点。

  整棵行为树中，只有Condition Node和Action Node才能成为Leaf Node，而也只有Leaf Node才是需要特别定制的Node；Composite Node和Decorator Node均用于控制行为树中的决策走向。(所以有些资料中也统称Condition Node和ActionNode为Behavior Node，而Composite Node和Decorator Node为Decider Node。)

 unity 插件Behavior Designer对于变量的共享做了如下处理：

在同一个Behavior Tree（一般一个GameObject有一个Behavior Tree）的Task间共享的局部变量可以直接在编辑器的Variables添加；另外也支持在不同Behavior Tree之间共享的全局变量；还支持Task与非Task（游戏系统中的其他脚本）之间进行变量传递，通过下面代码进行：

 behaviorTree.GetVariableName("MyVariableName");

*behaviorTree.SetVariableName("MyVariableName", value);*