AI 角色复杂决策——行为树

行为树技术原理

行为树主要采用 4 种节点来描述行为逻辑，分别是顺序节点、条件节点、选择节点和行为节点。每一棵行为树表示一个 AI 逻辑。要执行其逻辑，需要从根节点开始遍历执行整棵树。

节点从结构上分为两类：组合节点、叶节点。

组合节点：树中间的节点。

叶节点：没有孩子的节点，一般用来放置执行逻辑和条件判断。

叶节点

叶节点包括两种类型节点，分别是条件节点和行为节点

1.条件节点

可以理解为 if 条件测试，用来测试当前是否满足某些性质或条件，例如："玩家是否在 20 米之内？"

如果条件测试结果为真，那么向父结点返回 success，否则返回 failure

2.行为节点

用来完成实际的工作，例如：播放动画、规划路径、让角色移动位置、感知敌人、更换武器、播放声音、增加生命值等。

在执行这种节点的时候，可能只需要一帧，也可能需要多帧。

绝大部分动作节点会返回 success。

组合节点

用来控制树的遍历方式，最常用的组合节点有选择节点、顺序节点、并行节点、修饰节点等

1.选择节点

也称为优先级 Selector 节点，它会从左到右依次执行所有子节点，只要子节点返回 failure，就继续执行后续子节点，直到有一个节点返回 success 或 running 为止，这时它会停止后续子节点的执行，向父节点返回 success 或 running。若所有子节点都返回 failure，那么它向父节点返回 failure。

需要注意的是，当子节点返回 running 时选择节点会"记录"返回 running 的这个子节点，下个迭代会直接从该节点开始执行（行为节点对应的代码执行时间较长，这时返回 running）

```
// C#伪代码 Selector 节点
for (int i = 0; i < n; i++)
{
    state = Tick(child(i));
    if (state == Running)
        return Running;
    if (state == Success)
        return Success;
}return Failure;
```

2.顺序节点

它会从左到右依次执行所有子节点，只要子节点返回 success，它就继续执行后续子节点，直到有一个节点返回 failure 或 running 为止，这时它会停止后续子节点的执行，向父节点返回 failure 或 running。若所有子节点都返回 success，那么它向父节点返回 success。

它与选择节点正好是相反的感觉（选择节点类似于解决怎么进入这个房间如"爆破"、"踹门"，而顺序节点类似于解决能不能成功做这一件事如"爆破"，我有没有炸弹包，引线，火源）

```
// C#伪代码 Sequence 节点
for (int i = 0; i < n; i++)
{
    state = Tick(child(i));
    if (state == Running)
        return Running;
    if (state == Failure)
        return Failure;
}return Success;
```

3.随机选择节点

前面提到的两种组合节点都是有优先级的，最左面的节点优先级最高，然而对于随机选择节点，它不是永远按照从左到右的顺序执行，而是会随机选择访问子节点的顺序。

4.修饰节点（循环执行某个节点直到达到某个条件，过滤器）

修饰节点值包含一个子节点，用于以某种方式改变这个子节点的行为。

修饰节点有很多种，其中有一些是用于决定是否允许子节点运行的，这种修饰节点有时也称为过滤器，例如 Until Success，Until Fail 等。Until Success 的行为是这样的：循环执行子节点，直到子节点返回 success 为止。

例如，检测"视线中是否有敌人"，在修饰节点的作用下，会不停地检测，直到发现敌人为止。

Limit 节点用于指定某个子节点的最大运行次数。例如：如果子节点的连续运行次数小于 3，那么如果大于等于 3 返回 failure。

Time 节点设置了一个计数器，它不会立即执行子节点，而是等待一段时间，时间到了才开始执行。

5.并行节点（Parallel）

有多个子节点，与顺序节点不同的是，这些子节点的执行是并行的——不是一次执行一个，而是同时执行，直到其中一个返回 failure（或全部返回 successs）为止。此时，并行节点向父节点返回 failu（或 success），并终止其它所有子节点的执行。

并行节点用在如下一帧发生了某种事件，需要打断这些节点的执行。这种情况显然顺序节点是无法实现的，而并行节点可以实现，因为并行必须保证所有结点为 success

子树的复用

游戏中可能有多种的 AI，需要不同的行为树，但是可能它们的行为树中的战斗系统是一样的，这时，为了避免重复工作，就可以复用战斗的子树。

使用行为树与有限状态机的权衡

行为树和有限状态机在游戏 AI 中已经十分常见，但它们并不能互相代替，所以应用的时候加以权衡，了解它们不同的适用性。

（1）对于状态机来说，每个时刻都是出于某种"状态"，等待某个事件（转换）的发生。本质上是"事件驱动"的，即周围游戏世界发生的"事件"驱动角色的"状态"变化。从实现上来看，状态机既可以采用轮询的方式实现（每帧主动查询是否发生了某种事件），也可以采用事件驱动的方式实现（例如，注册一个回调函数，每当事件发生时，调用这个函数，在其中改变状态机的状态，或是利用消息，当事件发生时，发送消息）。**状态机是事件驱动，采用轮询或者回调函数的方式**

（2）对于行为树，处理周围游戏世界的变化的任务是由条件节点来完成的，这相当于每次遍历行为树时，条件节点都向周围世界发出某种"询问"，以这种方式来监视游戏世界发生的事情。因此，这实际上是"轮询"的方式——不断的主动查询。虽然目前已经有了一

些高级的技术，能够将事件的驱动集成在行为树中，但在实现中，绝大多数行为树都是自顶向下，采用轮询的方式实现的。**自顶向下，主动查询，轮询方式**

（3）一般来说，行为树不太适合需要事件驱动的行为。例如 AI 角色需要对大量的外部事件作出反应——当 AI 角色正在向某个目标移动时，突然发生了某个事件，如同伴需要救援，玩家被击中等事件，需要立即终止这个移动过程，需要重新做出新的决策等。在遇到这种情况的时候，还是在状态机和行为树直间好好做一下权衡。**行为树不适合事件驱动**

行为树执行时的协同（Coroutine）

也就是所谓的行为节点的多帧执行。在 Unity 中就用 StartCoroutine 开启协同程序就好了

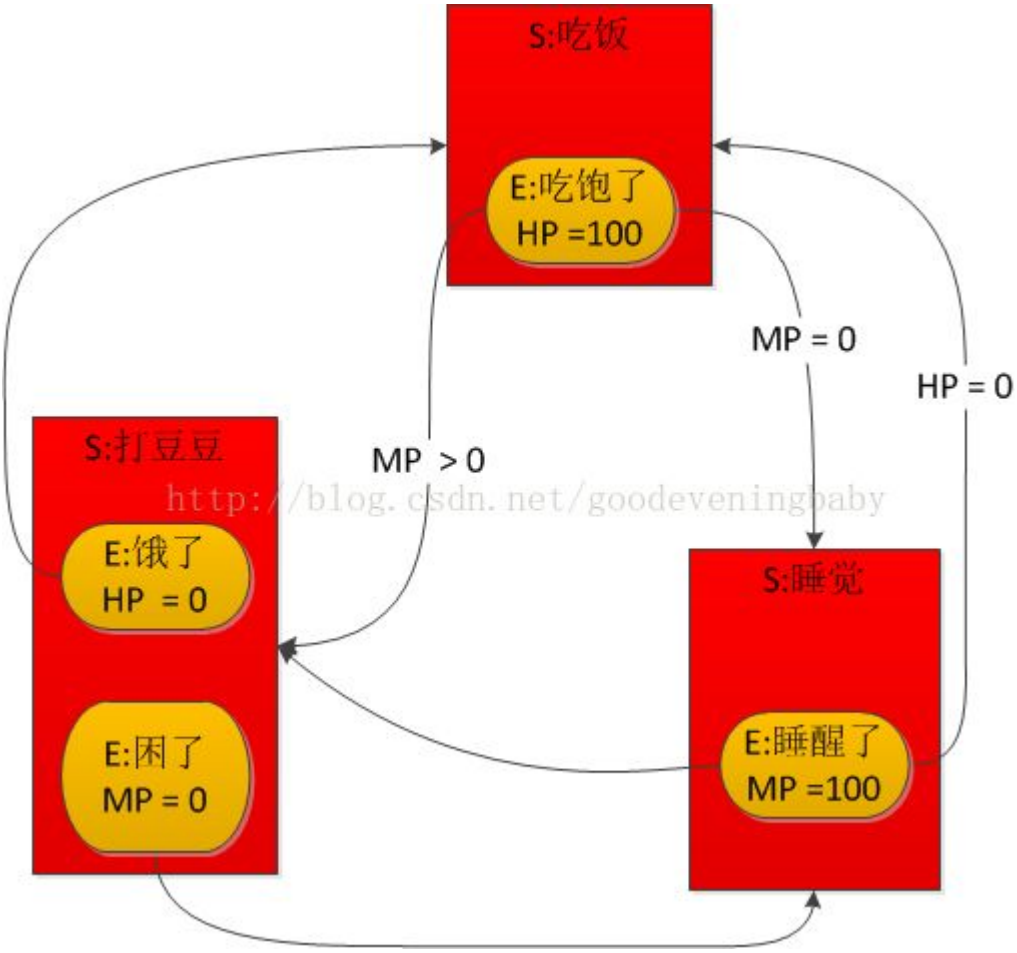
行为树与状态机取舍

行为树，实现 AI 的过程更加得有技巧，框架设计者较为全面考虑了我们可能会遇到的种种情况，把每种情况都抽象成了一个类型的节点，而我们要做的就是按照规范去写节点，然后把节点连接成一颗行为树。更加得具有面向对象的味道，行为模块间的藕合度相对较低。

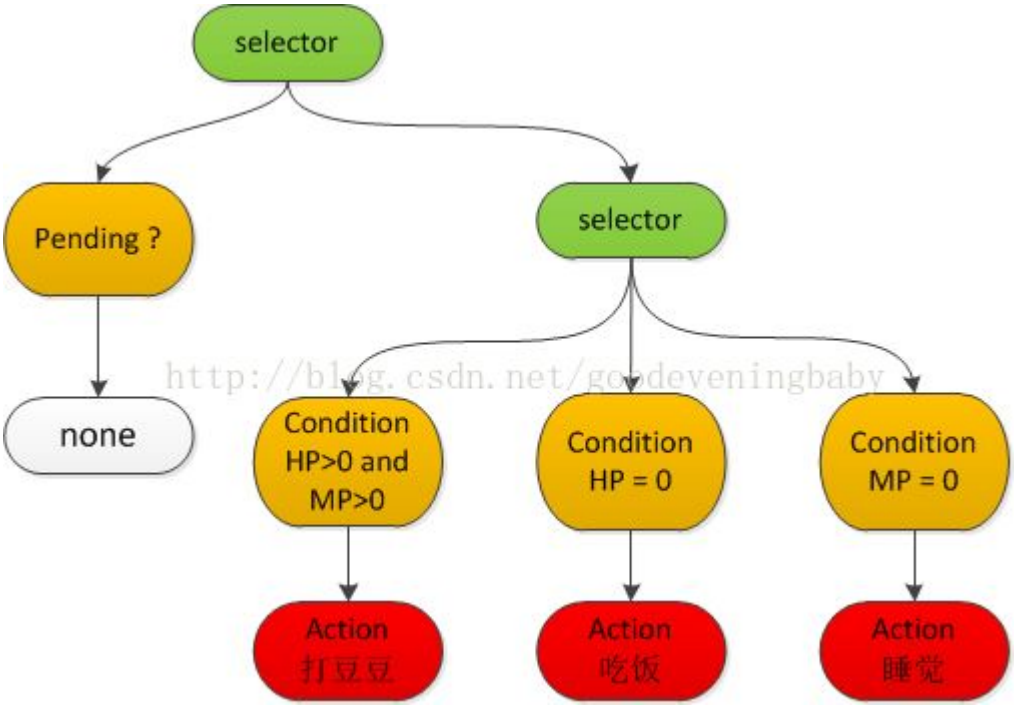
举个粗糙的例子来比较一下两者的不同：
AI 行为：吃饭 睡觉 打豆豆(很消耗体力和脑力的;)

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”。