【讲古堂】行为树

Behavior Tree，BT

行为树，英文是Behavior Tree，简称BT，是由行为节点组成的树状结构

每次执行行为树，都会从根节点遍历整个树，父节点执行子节点，子节点执行完后将结果返回父节点。下面是基本的四个节点：

1 \*顺序节点（Sequence）：属于组合节点，顺序执行子节点，只要碰到一个子节点返回false，则返回false，否则返回true，类似于程序中的逻辑与。

2 \*选择节点（Selector）：属于组合节点，顺序执行子节点，只要碰到一个子节点返回true，则返回true，否则返回false，类似于程序中的逻辑或。

3 \*条件节点（Condition）：属于叶子节点，判断条件是否成立。

4 \*执行节点（Action）：属于叶子节点，执行动作，一般返回true。

行为树从本质上来说，是一颗逻辑树，它把所有的行为逻辑用树形结构串联起来，仔细观察的话，可以发现行为树的核心思想有三个方面：

逻辑分离

逻辑关联

逻辑抽象

行为树把逻辑分散在节点中，每个节点负责自己的逻辑部分，这些逻辑节点又可以被放在行为树的其他部分，也就是可以被重用。在这个基础上，行为树又抽象了三个逻辑概念，

控制逻辑，

前提逻辑，

行为逻辑，其中行为逻辑用来描述功能，控制逻辑和前提逻辑用来描述逻辑间的关联，对于逻辑关联的抽象是行为树相较于FSM的一个重大突破，它使得逻辑的关联“可视化”了，用过行为树的人都会有这样的感觉，我只要看一下树的结构，我就能知道整个AI行为是如何协作的了，也正是这样的优势使得行为树现在被越来越多的用在了AI逻辑中。

行为树就是逻辑树，是一种对于逻辑的维护和管理的架构。

只要存在复杂的逻辑，就可以用行为树的方式去做，它可以很好的帮助你理清思路，实现漂亮的逻辑代码。

在BT中，节点是有层次（Hierarchical）的，子节点由其父节点来控制。每个节点的执行都有一个结果（成功Success，失败Failure或运行Running），该节点的执行结果都由其父节点来管理，从而决定接下来做什么，父节点的类型决定了不同的控制类型。节点不需要维护向其他节点的转换，节点的模块性（Modularity）被大大增强了。实际上，在BT里，由于节点不再有转换，它们不再是状态（State），而是行为（Behavior）。由此可见，BT的主要优势之一就是其更好的封装性和模块性，让游戏逻辑更直观，开发者不会被那些复杂的，乱七八糟的连线绕晕。

行为树中的主要部分基本都有涵盖，包括前提（Precondition），选择节点（Selector），并行节点（Parallel），序列节点（Sequence）等等。

行为树的基本概念：执行每个节点都会有一个结果（成功，失败或运行）子节点的执行结果由其父节点控制和管理返回运行结果的节点被视作处于运行状态，处于运行状态的节点接下来被直接运行其中理解运行状态是理解行为树的关键，也是使用好行为树的关键。

我们先可以抽象两个概念，“单个条件”和“单个目标”。单个的条件包含“怎么算是达到条件”的一段逻辑，单个的目标也是一段逻辑，包含了“怎么算是完成”。所以这些都可以做成一个个逻辑单元，就像行为树的前提和行为节点一样。

另外根据设计的需求，接任务的条件可能有多个，完成的目标也可以有多个，所以这些单个逻辑之间就存在逻辑关联，所以我们可以借鉴行为树中控制节点的概念，把这些逻辑关联也抽象出来，成为“关联”，比如一个最简单的逻辑关联就是“并且”，这样我们就可以描述这样一个逻辑，要完成目标1，“并且”完成目标2，这里我们就把两个“单个目标逻辑”用“关联”串起来了。

最后，我们就在每个任务中定义了两颗逻辑树（换个通用的名称，其本质和行为树是一样的），一个是接这个任务的条件树，一个是完成这个任务的目标树，这样每个任务都可以用配置文件来配，我们也做了一个工具来帮助设计师来生成和编辑任务，作为程序只要维护那些可以被用到的“条件”，“目标”，“关联”就可以了。

行为树，由名字就可以看到，它是一个树结构，通过各个节点相互连接，所以我先定义了节点的基类

要把树链接起来，需要在这个类中保留父节点指针，和子节点指针，我用了一个固定的数组来保存子节点指针，它的大小是16，也就是说，一个节点最多可以有16个子节点

有了这些变量的定义，我们就可以串联起一颗树了。到目前为止，这个节点类还仅仅是一个树的节点，作为行为树的节点还差了些东西，在以前的介绍中，我们知道行为树的每一个节点都可以绑定一个称为前提（Precondition）的部分，用来作为是否进入这个节点的条件，在我的实现中，我把这个前提拆分成了两个部分，一个称为“内在前提”，一个称为“外在前提”。“内在前提”是和节点类静态绑定的（也就是说，这个节点的固有前提），而“外在前提”是可以和节点做动态绑定的。这样做的原因是，由于在行为树上，节点是可以被复用的，在不同的子树上他的进入条件往往是不同的。比如，“移动”，这是一个常见的行为节点，逃跑的时候，可能需要“移动”，追击的时候也需要“移动”，但进入这个节点需要不同的“外在前提”，所以这里就需要让节点支持动态绑定的前提。“内在前提”，我用继承的方式来实现，而“外在前提”，我用了另一个类来实现

可以看到这里用到了一个叫做BevNodePrecondition的类，用来表示“外在前提”，他是一个纯虚函数，只有一个方法，先看一下它的定义，后面会有详细的讨论。

\_DoEvaluate虚方法就是需要被子类继承并实现的“内在前提”，这两种前提在Evaluate方法中被结合了起来，用来检测进入条件，当返回True时，就表示当前节点可以被运行。返回False时，就表示当前节点进入条件不满足，不能被运行。

在节点基类的中，还有两个重要的方法是:

转移（Transition）的概念是第一次出现，转移（Transition）指从上一个可运行的节点切换到另一个节点的行为。这个方法会被在节点切换的时候调用，比如，在一个带优先级的选择节点下有节点A，和节点B，节点A的优先级高于节点B，当前运行的节点是B，然后发现节点A可以运行了，但带优先级的选择节点就会选择去运行节点A，这时就会调用节点B的Transition方法，所以在这个方法中，一般可以用来做一些清理的工作。Tick方法就是通常的更新方法，就不多说了。

再来看一下这三个重要方法的参数，一共有两种类型的参数，BevNodeInputParam和BevNodeOutputParam，前者是传入参数，可以认为是行为树的输入，用const作为限定符，表示只读，后者是传出参数，可以认为是行为树的输出，可以修改。其实，从代码中可以看到，这两种类型的本质都是一样的，都是一个名为AnyData的类

1: typedef AnyData BevNodeInputParam;

2: typedef AnyData BevNodeOutputParam;

由于输入和输出参数是游戏相关的，所以这里用AnyData这个类来表示，这个类可以存放任意的数据结构，所以，这个类中真正的内容是需要玩家自己定义的。

最后来看看行为树是如何被定义和更新的（可以在示例程序中找到相关代码）

定义自己的输入和输出参数（BevInputData，BevOutputData）

创建行为树，保存根节点指针（m\_BevTreeRoot）

测试是否有可以运行的节点，如有则更新

上一次说到了节点的基类，它描述了在行为树上一个节点的基本结构。我们知道，在行为树上有两大类的节点，一种我称之为“控制节点”，像“选择节点”，“并行节点”，“序列节点”都属于此类，这类节点负责行为树逻辑的控制，是和具体的游戏逻辑无关的，属于行为树库的一部分，并且这类节点一般不会作为叶节点。还有一类称为“行为节点”，也就是行为树上挂载的具体行为，是和游戏逻辑相关的，不属于行为树库的一部分，需要自己去继承和实现，这类节点一般都作为叶节点出现。

先来看看“行为节点”的代码，我先从节点的基类继承了一个所有“行为节点”的基类

1: class BevNodeTerminal : public BevNode

2: {}

在它的Tick方法中，我做了一个简单的状态机（可以自行看代码），负责处理进入行为（Enter），更新行为（Execute），退出行为（Exit），所有的行为节点应该继承自BevNodeTerminal类，并且重写这些虚函数，在进入和退出行为里，可以做一个初始化和清理的工作：

1: class BevNodeTerminal : public BevNode

2: {

3: protected:

4: virtual void \_DoEnter(const BevNodeInputParam& input) {}

5: virtual BevRunningStatus \_DoExecute(const BevNodeInputParam& input, BevNodeOutputParam& output) { return k\_BRS\_Finish;}

6: virtual void \_DoExit(const BevNodeInputParam& input, BevRunningStatus \_ui\_ExitID) {}

7: }

值得注意的是，在Tick方法中，它有一个返回值，表示当前节点是否处理完毕，在库中，我定义了一个enum来表示节点的运行状态：

1: enum BevRunningStatus

2: {

3: k\_BRS\_Executing = 0,

4: k\_BRS\_Finish = 1,

5: ...

6: };

当返回k\_BRS\_Finish的时候，就表示当前节点已经处理完毕了，如果再次进入该节点，就认为是重新进入了。用上面描述的那个状态机的来说的话就是，如果是重新进入，会先调用\_DoEnter方法，然后调用\_DoExecute方法，如果\_DoExecute返回正在运行（k\_BRS\_Executing），那么以后再进入这个节点就会直接调用\_DoExectue，如果返回已经结束（k\_BRS\_Finish），则会调用\_DoExit，以后再进入这个节点就会重新调用\_DoEnter方法了。

对于控制节点来说，它的运行状态和子节点的运行状态是息息相关的，比如，选择节点的运行状态，就是它当前选择的这个节点的运行状态，并且，有时控制节点的控制逻辑也和子节点的运行状态有关，比如序列节点，当它前一个子节点运行结束，序列节点就会自动的切换到下一个子节点运行。所以在实现具体的行为类时，我们应该要正确的返回节点的运行状态。在例子程序中，我做的一个“空闲”（idle）的行为节点，就能很好的说明问题：

1: class NOD\_Idle : public BevNodeTerminal

2: {

3: public:

4: NOD\_Idle(BevNode\* \_o\_ParentNode)

5: :BevNodeTerminal(\_o\_ParentNode)

6: {}

7: protected:

8: virtual void \_DoEnter(const BevNodeInputParam& input)

9: {

10: m\_WaitingTime = 0.5f;

11: }

12: virtual BevRunningStatus \_DoExecute(const BevNodeInputParam& input, BevNodeOutputParam& output)

13: {

14: const BevInputData& inputData = input.GetRealDataType<BevInputData>();

15: BevOutputData& outputData = output.GetRealDataType<BevOutputData>();

16:

17: f32 timeStep = inputData.m\_TimeStep;

18: m\_WaitingTime -= timeStep;

19: if(m\_WaitingTime < 0)

20: {

21: outputData.m\_BodyColor = D\_Color(rand() % 256, rand() % 256, rand() % 256);

22: return k\_BRS\_Finish;

23: }

24: return k\_BRS\_Executing;

25: }

26: private:

27: float m\_WaitingTime;

28: };

这段代码中的某些内容不明白也没有关系，我们主要关注的是关于节点运行状态的部分。这个Idle行为做了一件这样的事，就是不停的变换自己的颜色，间隔是0.5秒，当时间一到，就会返回运行结束（k\_BRS\_Finish），并输出当前的颜色，当时间还没到，则返回运行中（k\_BRS\_Executing），并且维持当前颜色。可以看到，我们用运行状态控制了计时器的重置，选择在\_DoEnter方法中重置了计时器，当然，更合理的做法是在时间一到的时候，就重置计时器，并且永远返回运行中，不过这个例子里，我主要就是想用来演示运行状态，和\_DoEnter的相关用法。

接下去再来看看控制节点，我一共写了5种控制节点，带优先级的选择节点（BevNodePrioritySelector），不带优先级的选择节点（BevNodeNonePrioritySelector），序列节点（BevNodeSequence），并行节点（BevNodeParallel），循环节点（BevNodeLoop），这些节点的进入条件和选择逻辑都是按照在行为树中改节点的定义来做的，我想用一张表格来说明：

测试（Evaluate） 更新（Tick）

带优先级的选择节点（BevNodePrioritySelector） 从第一个子节点开始依次遍历所有的子节点，调用其Evaluate方法，当发现存在可以运行的子节点时，记录子节点索引，停止遍历，返回True。 调用可以运行的子节点的Tick方法，用它所返回的运行状态作为自身的运行状态返回

不带优先级的选择节点（BevNodeNonePrioritySelector） 先调用上一个运行的子节点（若存在）的Evaluate方法，如果可以运行，则继续运保存该节点的索引，返回True，如果不能运行，则重新选择（同带优先级的选择节点的选择方式） 调用可以运行的子节点的Tick方法，用它所返回的运行状态作为自身的运行状态返回

序列节点（BevNodeSequence） 若是从头开始的，则调用第一个子节点的Evaluate方法，将其返回值作为自身的返回值返回。否则，调用当前运行节点的Evaluate方法，将其返回值作为自身的返回值返回。 调用可以运行的子节点的Tick方法，若返回运行结束，则将下一个子节点作为当前运行节点，若当前已是最后一个子节点，表示该序列已经运行结束，则自身返回运行结束。若子节点返回运行中，则用它所返回的运行状态作为自身的运行状态返回

并行节点（BevNodeParallel） 依次调用所有的子节点的Evaluate方法，若所有的子节点都返回True，则自身也返回True，否则，返回False 调用所有子节点的Tick方法，若并行节点是“或者”的关系，则只要有一个子节点返回运行结束，那自身就返回运行结束。若并行节点是“并且”的关系，则只有所有的子节点返回结束，自身才返回运行结束

循环节点（BevNodeLoop） 预设的循环次数到了就返回False，否则，只调用第一个子节点的Evaluate方法，用它所返回的值作为自身的值返回 只调用第一个节点的Tick方法，若返回运行结束，则看是否需要重复运行，若循环次数没到，则自身返回运行中，若循环次数已到，则返回运行结束

可能看表格内的描述会感觉有点拗口，可以结合代码一起看，会理解的更好。特别要提一点的是，在某些控制节点的Evaluate方法中，我会修改和记录可以运行的节点索引，当调用Tick的时候，就可以用这个索引来找到可以运行的节点了。这种模式和我以前提到的行为树更新模式有点不太一样，不过本质上是相同的。

行为树最后一个要讲的地方，是关于前提（Precondition），在第一部分里，我略微提到了一下，这次我们来仔细看看，再来看看关于前提的纯虚基类的定义：

1: class BevNodePrecondition

2: {

3: public:

4: virtual bool ExternalCondition(const BevNodeInputParam& input) const = 0;

5: };

每一个前提类，都需要实现这个判断的虚函数。我在《用类来表示逻辑运算–关于行为树前提的一种实现方式》提到，我们可以用类来表示逻辑运算，这样的好处是可以做到模块化，同样的判断条件可以复用，所以在库中，我也实现了这种逻辑的表达方式，定义了基本的逻辑运算类

1: class BevNodePreconditionTRUE{};

2: class BevNodePreconditionFALSE{};

3: class BevNodePreconditionNOT{};

4: class BevNodePreconditionAND{};

5: class BevNodePreconditionOR{};

6: class BevNodePreconditionXOR{};

从这些类的名字应该就可以明显的看出这些类的含义了，和逻辑操作符一样，有些类的构造函数需要两个参数，以此来表示二元的逻辑运算（AND，OR，XOR），有些只需要一个参数，以此来表示一元的逻辑运算（NOT）。前提类被用来附在行为树的节点上（每一个节点都可以附加），默认情况下，节点上是没有前提类的，也就是不存在“外在前提”，而只有“内在前提”，这和附了一个BevNodePreconditionTRUE（永远返回True）的“外在前提”的节点是等价的。

好了，行为树库的内容基本就是这些了。接下去我们来看看例子程序，介绍如何用库来创建行为树，例子的代码在BevTreeTest这个工程中，编译后可直接运行，这个例子分别演示了三个行为树，从简单到复杂，单击鼠标可以在这三个例子间切换。这个程序实现了这样一个功能，“在场景地图上，定时会产生一个目标点，智能体就会根据行为树的定义，用不同的行为模式移动到目标点”。

在这个程序中，我为智能体一共定义了4个行为：

1: class NOD\_Idle{}; //空闲，表现是颜色不停变化

2: class NOD\_Breathe{}; //呼吸，表现是大小规律性变化

3: class NOD\_MoveTo{}; //移动，平移到某目标点

4: class NOD\_FaceTo{}; //转向，转向到某方向

再定义了2个“外在前提”：

1: class CON\_HasReachedTarget{}; //是否到达目标点

2: class CON\_HasFacedToTarget{}; //是否朝向目标点

我就用第一个例子来说，第一例子的行为树图如下：

BevTreeTest1

这是一个很简单的行为树，根节点是一个带优先级的选择节点，所以MoveTo比Idle的优先级高，MoveTo带有一个“外在前提”，“当没有到达目标点”时，会选在MoveTo的行为，反之，则选Idle的行为。

在代码中，可以这样来定义这棵行为树

1: BevNode& ret =

2: BevNodeFactory::oCreatePrioritySelectorNode(NULL, "root");

3: BevNodeFactory::oCreateTeminalNode<NOD\_MoveTo>(&ret, "move to")

4: .SetNodePrecondition(new BevNodePreconditionNOT(new CON\_HasReachedTarget()));

5: BevNodeFactory::oCreateTeminalNode<NOD\_Idle>(&ret, "idle")

6: .SetNodePrecondition(new BevNodePreconditionTRUE());

7: m\_BevTreeRoot = &ret;

我在库中定义了一些工厂方法，帮助创建相关的节点。值得注意的是，我在这里演示了用类表示逻辑的用法。我在定义行为树的时候，会用一些格式上的缩进，来表示相应的父子结构，这仅仅是为了视觉上比较明了。当然，以后可以改进行为树的定义接口，更可以用数据文件来定义行为树。

这样定义完毕后，我们就可以用行为树来决策我们的行为了，代码相当简单

1: BevNodeInputParam input(&m\_BevTreeInputData);

2: BevNodeOutputParam output(&m\_BevTreeOutputdata);

3: if(m\_BevTreeRoot->Evaluate(input))

4: {

5: m\_BevTreeRoot->Tick(input, output);

6: }

在例子中，我尽量把行为树中要输出的变量写到BevNodeOutputParam结构中（而不是直接修改智能体的信息），这样做的好处是可以让行为树的输入和输出的接口相当清晰，做成黑盒，可以参考我在这里的讨论。

第二个例子演示了并行节点的用法，第三个例子演示了序列节点的用法，就不多说了，大家可以自行看代码。

所有的代码可以通过以下方式获得：

下载地址：

GoogleCode下载点（exe文件夹中已包含可执行文件）

也可用svn通过以下地址来得：

http://tsiu.googlecode.com/svn/branches/blogver/

编译方法：

用VS2005以上打开，选择Debug NoDx或者Release NoDx，编译后，运行BevTreeTest.

相关代码：

TAI\_BevTree.h

TAI\_BevTree.cpp

关于TsiU

TsiU是我一直在维护的一个自己用的小型的框架，我平时做的一些AI的sample，或者一些工具，都会基于这个框架，TsiU有一些基本的UI控件库，网络模块库，GDI绘图模块，D3D绘图模块等等，可以快速的做成一个小型的示例程序，很方便（具体可参考SampleAppｓ里的例子程序），并且整个架构是用Object的方式来组织，非常容易理解和扩展。整个框架很轻量化，基本就是做了一些底层的基本的功能，这样我在平时做东西的时候，就不需要重新写底层了，把精力都放在高层的实现了。以后分享代码都会基于这个框架，大家也可以通过svn来随时update到我最新的改动。下图就是TsiU里的几个工程介绍，代码不多，大家想看的也可以自己看一下:)

行为树节点（Behavior Tree Nodes）：

· 选择节点（Selector）: 顺序执行子节点，只要它的一个子节点返回true，则整个分支返回true，反之返回false，类似程序中的逻辑或（OR）。

· 顺序节点（Sequence） ：顺序执行子节点，只要它的一个子节点返回false，则整个分支返回false，反之返回true， 类似程序中的逻辑与（AND）。

· 条件节点（Condition） : 属于叶子节点（该种节点不允许有任何子节点），用于描述一个条件是否成立。

· 行为节点（Action） : 属于叶子节点（该种节点不允许有任何子节点），用于描述一个最终执行的动作，一般返回true。

其中选择节点（Selector）， 顺序节点均属于组合节点。组合节点是一个分类概念，并非指一个节点的实例。

实现行为树（Behavior Tree）：

状态机（State Machine）

　　说到状态机的概念，也许很多程序员都相当熟悉。就拿上面的行为树来说，仅仅是描述了一个怪物的AI，但其中涉及的状态可谓不少。在传统的设计思路中，玩家进入怪物的视野时，怪物的状态将会马上发生改变而从主动攻击玩家。（感知系统）

　　状态（State），指的是对象的某种形态，在当前形态下可能会拥有不同的行为和属性。

状态机（State Machine），指控制对象状态的管理器。对象的状态不会无端端改变，它需要在某种条件下才会变换。比如上面定义的行为树中的[巡逻]行为，它就规定了必须是在怪物视野中找不到目标的情况下才会变更的状态。一旦有玩家进入了它的视野，它将马上切换为攻击状态。总而言之，状态会在某个事件触发之后变更。不同的状态也有可能决定了对象的不同属性和行为。

---------------------------------------------------------------------

行为树(Behavior Tree)具有如下的特性：

它只有4大类型的Node：

\* Composite Node

\* Decorator Node

\* Condition Node

\* Action Node

任何Node被执行后，必须向其Parent Node报告执行结果：成功 / 失败。

这简单的成功 / 失败汇报原则被很巧妙地用于控制整棵树的决策方向。

---------------------------------------------------------------------

先看Composite Node，其实它按复合性质还可以细分为3种：

\* 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: 一False则返回False，全True才返回True。

Parallel Sequence Node: 一True则返回True，全False才返回False。

Parallel Hybird Node: 指定数量的Child Node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。

---------------------------------------------------------------------

再看Decorator Node，它的功能正如它的字面意思：它将它的Child Node执行

后返回的结果值做额外处理后，再返回给它的Parent Node。很有些AOP的味道。

比如Decorator Not/Decorator FailUtil/Decorator Counter/Decorator Time...

更geek的有Decorator Log/Decorator Ani/Decorator Nothing...

---------------------------------------------------------------------

然后是很直白的Condition Node，它仅当满足Condition时返回True。

---------------------------------------------------------------------

最后看Action Node，它完成具体的一次(或一个step)的行为，视需求返回值。

而当行为需要分step/Node间进行时，可引入Blackboard进行简单数据交互。

---------------------------------------------------------------------

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Action

Node为Behavior Node，而Composite Node和Decorator Node为Decider Node。)

更强大的是可以加入Stimulus和Impulse，通过Precondition来判断masks开关。

通过上述的各种Nodes几乎可以实现所有的决策控制：if, while, and, or,

not, counter, time, random, weight random, util...

---------------------------------------------------------------------

总的来说，行为树具有如下几种优点，确实是实现AI框架的利器，甚至是一种

通用的可维护的复杂流程管理利器：

> 静态性

越复杂的功能越需要简单的基础，否则最后连自己都玩不过来。

静态是使用行为树需要非常着重的一个要点：即使系统需要某些"动态"性。

其实诸如Stimulus这类动态安插的Node看似强大，

但却破坏了本来易于理解的静态性，弊大于利。

Halo3相对于Halo2对BT AI的一个改进就是去除Stimulus的动态性。

取而代之的做法是使用Behavior Masks，Encounter Attitude，Inhibitions。

原则就是保持全部Node静态，只是根据事件和环境来检查是否启用Node。

静态性直接带来的好处就是整棵树的规划无需再运行时动态调整，为很多优化

和预编辑都带来方便。

> 直观性

行为树可以方便地把复杂的AI知识条目组织得非常直观。

默认的Composite Node的从begin往end的Child Node迭代方式就像是处理一个

预设优先策略队列，也非常符合人类的正常思考模式：先最优再次优。

行为树编辑器对优秀的程序员来说也是唾手可得。

> 复用性

各种Node，包括Leaf Node，可复用性都极高。

实现NPC AI的个性区别甚至可以通过在一棵共用的行为树上不同的位置来

安插Impulse来达到目的。

当然，当NPC需要一个完全不同的大脑，比如70级大BOSS，

与其绞尽脑汁在一棵公用BT安插Impulse，不如重头设计一棵专属BT。

> 扩展性

虽然上述Node之间的组合和搭配使用几乎覆盖所有AI需求。

但也可以容易地为项目量身定做新的Composite Node或Decorator Node。

还可以积累一个项目相关的Node Lib，长远来说非常有价值。

---------------------------------------------------------------------

using System;

public enum State

{

INIT, //初始化

PENDING, //正在执行

FINISH, //成功

FAIL //失败

}

//基类BTNode

public class BTNode

{

State state;

ComponentNode parent;

public virtual void reset()

{

state = State.Init;

}

//执行方法，用于重载override

public virtual void execute(AIControl agent)

{

}

public virtual void finish(AIControl agent, State stat)

{

this.state = stat;

if(this.parent!=null)

{

//把state传给父节点

this.parent.tryMoveNext(agent, stat);

}

}

}

//复合节点

public class ComponentNode:BTNode

{

List<BTNode> childs; //子节点

bool isFirstExe; //是否第一次执行，用于初始化

addChild(){};//添加子节点

removeChild(){};//移除子节点

override finish(){base.finish();}

moveNext()

{

if(CanMoveNext())

{

nextChild.execute();

}

else

{

moveEnd();

}

}

//先判断条件是否能执行下一个子节点再执行

tryMoveNext(State stat)

{

if(checkState(stat))

{

moveNext();

}

}

override execute()

{

if(isFistExe){isFirstExe = false;beforeFirstExecute();}

moveNext();

}

virtual beforeFirstExecute(){}//用于初始化

virtual checkState(){}//用于override

virtual moveEnd(){} //用于override

}

public class SequenceNode : ComponentNode

{

override bool checkState(State stat)

{

if(stat == State.Fail)

{

finish(State.fail);

return false;

}

return true;

}

override void moveEnd()

{

finish(State.finish);

}

}

public class SelectorNode : ComponentNode

{

override bool checkState(State stat)

{

if(stat == State.finish)

{

finish(State.finish);

return false;

}

return true;

}

override void moveEnd()

{

finish(State.Fail);

}

}

public class BehaviorNode:BTNode

{

}

//条件节点

class ConditionNode : BehaviorNode

{

override void execute()

{

if(condition())

{

finish(State.finish);

}

else

{

finish(State.fail);

}

}

virtual bool condition(){}

}

//行为节点

class ActionNode:BehaviorNode

{

virtual void execute()

{

finish(State.pending);

}

}

//具体实现的选择节点

class xxx:SelectorNode

{

override void beforeFirstExecute(){}

override void execute(){}

}

//具体实现的顺序节点

class yyy:SequenceNode

{

override void beforeFirstExecute(){}

override void execute(){}

}

//具体实现的条件节点

class yyy : ConditionNode

{

override bool condition(){}

}

//具体实现的行为节点

class zzz:ActionNode

{

override void execute(){}

}

//AIControl存放一颗AI行为树的根节点，每隔一段时间从根节点遍历行为树，根据当前的情况实现不同的分支叶子节点，从而实现具体的游戏逻辑

class AIControl

{

root = 读表根节点

think()

{

每隔一段时间 root.execute(this);

}

}

状态机，行为树，黑板，模糊逻辑，神经网络，贝叶斯

使用有限状态机(FSM)，分层有限状态机(HFSM)，

决策树(Decision Tree)