节点参考

腾讯游戏
Tencent Games

https://github.com/TencentOpen/behaviac

目录

behaviac 有以下 5 类节点类型：

其中‘组合’，‘装饰器’类都是分支节点，‘动作’类，和‘条件’类大都是子节点，而‘条件’类中的‘或’、‘与’是分支节点。

需要指出的是，下面的有些节点类型通过其他更基本类型的节点的组合也可以实现。这里提供的节点类型只是使用起来可能会更方便，更直观些。用户可以根据自己的偏好选择使用与否。用户也可以选择扩展节点类型提供扩展功能或者快捷方式，请参考‘教程’中的相关章节。同时如果根本不打算使用某些节点类型，要可以在 `config.xml` 里配置后就不会出现在可选的列表里了，请参考[配置 Config.xml]。

1 节点共有属性

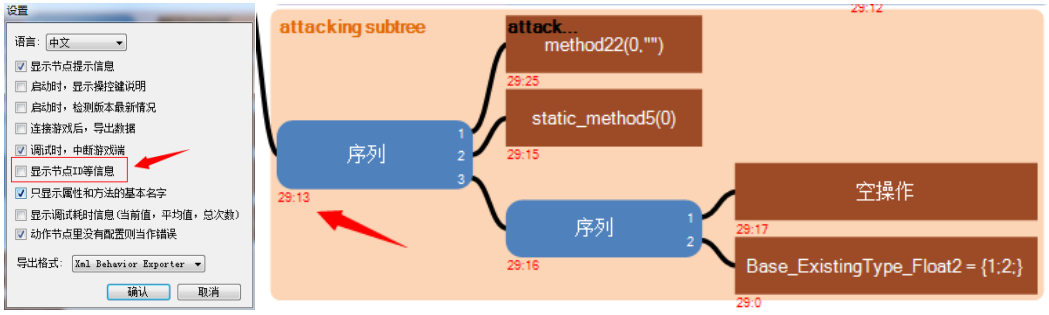
当在编辑器中选择每个节点的时候，都会有下图左侧所示的一些共有的属性。

1.1 注释和背景颜色

如上图右侧中所示的那样可以给某个部分表明一个颜色块和加上一些说明。

1.2 节点 ID

节点 ID 是节点的唯一 ID，可以用来定位节点，可以在文件->设置里打开“显示节点 ID 等信息”来在窗口里节点的左下角显示 ID。子树中的节点 ID 被加上主树节点的 ID，如下图。



进入动作，退出动作请参考[给节点配置 EnterAction/ExitAction]

2 节点状态

每个节点执行后都会返回下面状态之一：

- 成功 Success
- 失败 Failure
- 执行中 Running

在下面的说明中，涉及逻辑的时候，**返回成功意味着就是返回 true，而返回失败意味着就是返回 false**。理解这一点非常重要。
而返回成功或返回失败，意味着节点不再 Running，统称为节点结束。

执行一个 BT 的 exec 总是返回 EBTStatus 的这样的一个状态，成功、失败或运行；返回成功可以认为是返回 true，返回失败可以认为是返回 false，有的时候认为返回 true 或 false 在上下文的理解上更方便些。

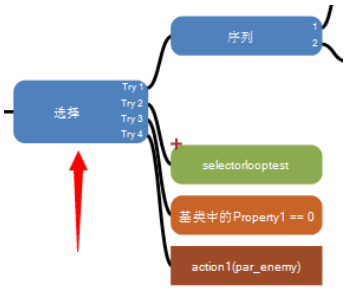
- 返回成功或失败意味着 BT 结束。下一次如果还被执行，则完全重新开始，即重新 enter，update，exit。
- 返回运行的情况则不同。如果本次执行返回运行，下次继续执行的时候，则继续执行（继续 update），如此循环直到返回成功或失败从而结束（exit）。特别的，该运行状态的节点在 BT 内部被‘记住’，下次执行的时候，‘直接’执行该运行状态的节点，而其他已经结束的节点不再被执行，理解这个处理对于把握 BT 的执行情况和效率比较**关键**。

由此可以看出 BT 就如同一个 coroutine，它 ‘知道’下一步从哪里继续。

3 组合节点

组合节点管理若干个子节点，也可以成为控制类节点。

3.1 选择（Selector）



Selector 节点是 BT 中传统的组合节点之一。该节点以给定的顺序依次调用其子节点，直到其中一个成功返回，那么该节点也返回成功。如果所有的子节点都失败，那么该节点也失败。

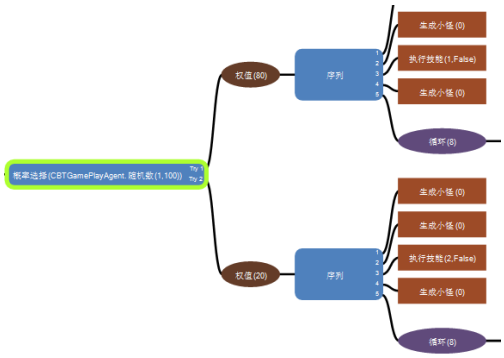
Selector 实现了‘||’的功能。我们知道表达式 R=A||B||C||D 执行的时候首先执行 A，如果 A 是 true 则返回 true，如果 A 是 false 则执行 B，如果 B 是 true 则返回 true，否则如果 B 是 false 则执行 C，如果 C 是 true 则返回 true，否则如果 C 是 false 则执行 D，并且返回 D 的值。

最一般的意义上，Selector 节点实现了一个**选择**，从子节点中选择一个。Selector 节点优先选择了排在前面的子节点。

此外，Selector 上还可以添加 Predicate 附件作为终止条件，请参考[判断（Predicate）]

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\selector.cpp

3.2 概率选择（SelectorProbability）



像 Selector 节点一样，SelectorProbability 也是从子节点中选择执行一个，不像 Selector 每次都是按照排列的先后顺序选择，SelectorProbability 每次选择的时候根据子节点的‘概率’选择，概率越大，被选到的机会越大。Selector 节点会顺序的去选择一直到成功的那个，而 SelectorProbability 的选择是‘直接’根据概率选择某个并且执行之，其他的则不会被执行。

特别需要指出的是，Selector 是按照子节点从上到下的顺序去执行子节点，一直到第一个返回成功的那个子节点然后返回成功，或者如果所有子节点都返回失败则返回失败。而

SelectorProbability 的不同之处则是，它根据概率，直接‘选择’某个子节点，执行之，无论其返回成功还是失败，SelectorProbability 也将返回同样的结果，当该子节点失败的话，SelectorProbability 也失败，它不会像 Selector 那样会继续执行接下来的子节点。

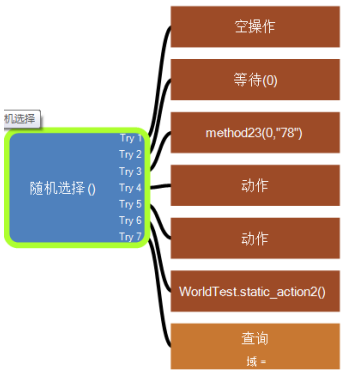
如下图，SelectorProbability 节点有随机数生成器可以配置，该随机数生成器是一个返回值为‘int’类型的函数。

该随机数生成器也可以为空，则用系统的缺省实现。

SelectorProbability 的子节点只能是‘权值’的子节点，该被节点系统自动添加。所有子节点的权值相加之和 sum 不需要是 100，子节点的概率是该子节点的（权值/sum）。

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\selectorprobability.cpp

3.3 随机选择（SelectorStochastic）



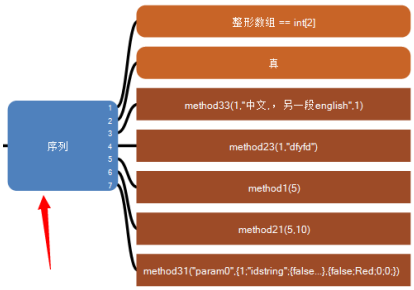
像 Selector 节点一样，SelectorStochastic 也是从子节点中选择执行一个，不像 Selector 每次都是按照排列的先后顺序选择，SelectorStochastic 每次选择的时候随机的决定执行顺序。假如 Selector 和 SelectorStochastic 都有 A，B，C，D 子节点。对于 Selector，每次都是顺序的按 A，B，C，D 的顺序选择，而对于 SelectorStochastic，有时按 A，B，C，D 的顺序选择，有时按 B，A，D，C 的顺序选择，又有时按 A，C，D，B 的顺序选择，等等。

和 SelectorProbability 相同的是，如下图，SelectorStochastic 节点有随机数生成器可以配置，该随机数生成器是一个返回值为‘int’类型的函数。

该随机数生成器也可以为空，则用系统的缺省实现。

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\selectorstochastic.cpp

3.4 序列（Sequence）



Sequence 节点是 BT 中传统的组合节点之一。该节点以给定的顺序依次执行其子节点，直到所有子节点成功返回，该节点也返回成功。只要其中某个子节点失败，那么该节点也失败。

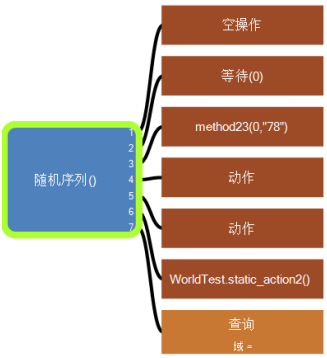
Sequencer 实现了‘&&’的功能。我们知道表达式 R=A&&B&&C&&D 执行的时候首先执行 A，如果 A 是 false 则返回 false，如果 A 是 true 则执行 B，如果 B 是 false 则返回 false，否则如果 B 是 true 则执行 C，如果 C 是 false 则返回 false，否则如果 C 是 true 则执行 D，并且返回 D 的值。

最一般的意义上，Sequence 节点实现了一个序列。实际上，Sequence 节点不仅可以管理‘动作’子节点，也可以管理‘条件’子节点。如上图的应用中，首先两个条件节点，跟着若干个其他节点，这两个条件节点实际上用作进入下面其他节点的 precondition，只有这两个条件是 true，下面的其他节点才有可能执行。

此外，Sequence 上还可以添加 Predicate 附件作为终止条件，请参考判断（Predicate）

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\sequence.cpp

3.5 随机序列（SequenceStochastic）



像 Sequence 节点一样，SequenceStochastic 也是从子节点中顺序执行，不像 Sequence 每次都是按照排列的先后顺序，SequenceStochastic 每次执行的时候随机的决定执行顺序。
假如 Sequence 和 SequenceStochastic 都有 A，B，C，D 子节点。对于 Sequence，每次都是顺序的按 A，B，C，D 的序列，而对于 SequenceStochastic，有时按 A，B，C，D 的序列，有时

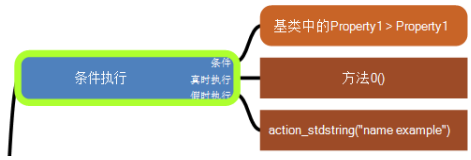
按 B，A，D，C 的序列，又有时按 A，C，D，B 的序列，等等。

和 SelectorStochastic 相同的是，如下图，SequenceStochastic 节点有随机数生成器可以配置，该随机数生成器是一个返回值为‘int’类型的函数。

该随机数生成器也可以为空，则用系统的缺省实现。

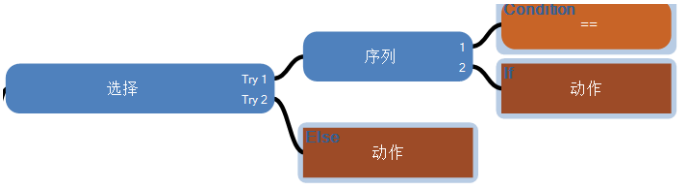
具体的执行逻辑可以查看 src\behaviortree\nodes\composites\sequencestochastic.cpp

3.6 条件执行（IfElse）

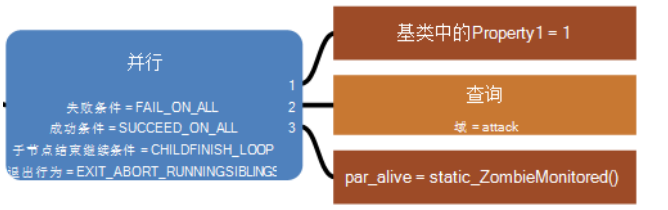


IfElse 是 behaviac 的一个扩展或快捷方式。IfElse 一定有 3 个子节点，第一个子节点是条件节点，第二个子节点是 If 分支，第三个子节点是 Else 分支。如果条件为真，那么执行“If”分支；否则，执行“Else”分支。而 IfElse 的执行结果则根据具体执行分支的执行结果来决定。

如果不使用 IfElse，完全可以用 Sequence 和 Selector 实现相同的功能，只不过显得有些臃肿。例如：

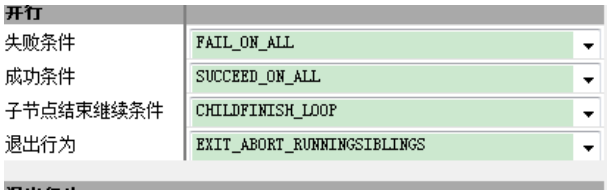


3.7 并行（Parallel）



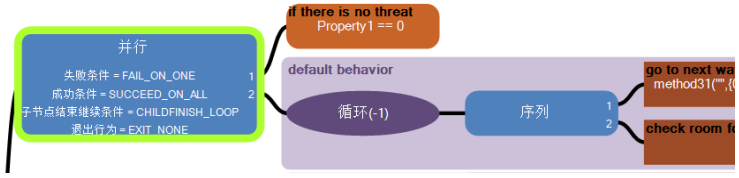
Parallel 节点在最一般意义上是并行的执行其子节点。在 Selector 和 Sequence 中，‘顺序’的‘一个一个’的执行子节点，上一个子节点执行结束后，根据其状态是否执行接下来的子节点。Parallel 节点在逻辑上是‘同时’并行的执行所有子节点，然后根据所有子节点的状态决定本身的状态。如何根据所有子节点的状态决定本身的状态呢？具体的如下图，Parallel

节点有几个属性可以配置：



- 失败条件，决定 Parallel 节点在什么条件下是失败的
- 成功条件，决定 Parallel 节点在什么条件下是成功的
- 子节点结束继续条件，子节点结束后是重新再循环执行还是结束后不再执行
- 退出行为，当 Parall 节点的成功条件或失败条件满足而成功或失败后，是否需要 abort 掉其他还在运行的子节点
- 当子节点执行状态既不满足失败条件，也不满足成功条件，且无 Running 状态子节点时，Parallel 节点返回 Failure

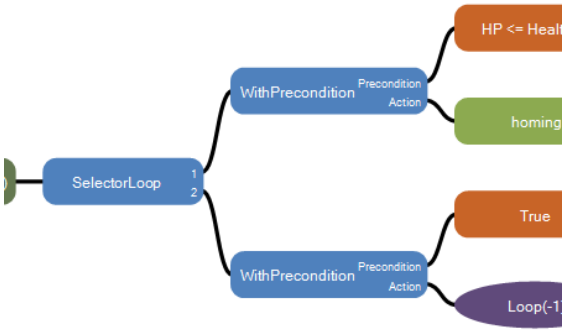
在序列 Sequence 中，当条件节点之后跟着其他节点的时候，条件节点实际上是作为其他节点的 precondition，只有条件节点是 true，下面的其他节点才有可能执行。
Parallel 节点可以用来实现所谓‘context precondition’，如下图：



Parallel 节点配置条件节点和其他节点，并且失败条件是缺省的配置 FAIL_ON_ONE ，那么只有当条件节点成功的时候其他节点才被执行，从而条件节点事实上是其他节点的 precondtion。和 Sequence 的不同之处在于，Sequence 节点实现的 precondition 只是‘进入’其他节点的 precondition，一旦‘进入’开始执行其他节点就不再检查该 precondition 了，而 Parallel 节点实现的 precondition 是‘context’的，不但‘进入’开始执行前需要检查，之后每次执行也都要检查。

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\parallel.cpp

3.8 循环选择（SelectorLoop）和条件动作（WithPrecondition）



SelectorLoop 和 WithPrecondition 作为对传统 BT 的扩展，对于处理事件比较直观和方便。

SelectorLoop 和 WithPrecondition 只能配对使用，即 SelectorLoop 只能添加 WithPrecondition 作为它的子节点， WithPrecondition 也只能作为 SelectorLoop 的子节点被添加。

- WithPrecondition 有 precondition 子树和 action 子树。只有 precondition 子树返回 success 的时候，action 子树才能够被执行。
- SelectorLoop 是一个动态的选择节点，和 Selector 相同的是，它选择第一个 success 的节点，但不同的是，它不是只选择一次，而是每次执行的时候都对其子节点进行选择。如上图中，假若它选择了下面有 True 条件的那个节点并且下面的 Loop 节点在运行状态，下一次它执行的时候，它依然会去检查上面的那个有条件 HP<=Health 的子树，如果该条件为真，则终止下面的运行节点而执行上面的 homing 子树。
- 当 Action 子树返回成功（success）的时候，SelectorLoop 节点也将结束并且返回成功，但是需要指出的是，当 Action 子树返回失败（failure）的时候，SelectorLoop 节点将继续尝试接下来的节点！

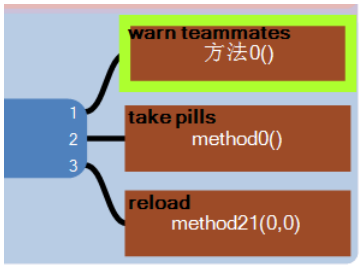
请参考 SelectorLoop 和 WithPrecondition

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\selectorloop.cpp

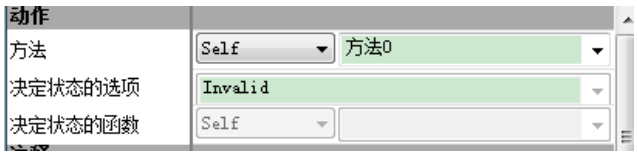
4 叶子节点

叶子节点只能作为最底层的叶子加到 BT 上去。不像组合节点是作为分支节点来控制其他节点。叶子节点必然被加到了其他组合节点上。

4.1 动作（Action）



Action 节点是 behaviac 中几乎是最重要的节点。Action 节点通常对应 Agent 的某个方法（Method），可以从下拉列表里为其选择方法。在设置其方法后，需进一步设置其“决定状态的选项（Status Option）”或“决定状态的函数（Status Functor）”，如下图所示，如果没有正确配置，则视为错误不可能被导出：

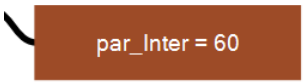


请参考动作（Action）

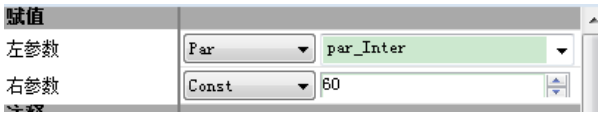
此外，Action 上还可以添加 Predicate 附件作为 Precondition，请参考判断（Predicate）

具体的执行逻辑可以查看 src\behaviortree\nodes\actions\action.cpp

4.2 赋值（Assignment）



Assignment 节点实现了一个赋值的操作，可以把右边的值赋值给左侧的参数。



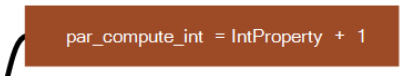
其中左参数可以是 par 或者是某个 agent 的属性，右参数可以是常数、par，其他 agent 的属性、或者是方法调用的返回值。

当需要对某个属性或 par 做一些加减乘除运算的时候，可以用节点计算（Compute）。但请注意这些操作的‘粒度’过小，大量这种小‘粒度’的操作可能对性能造成影响。请慎用。

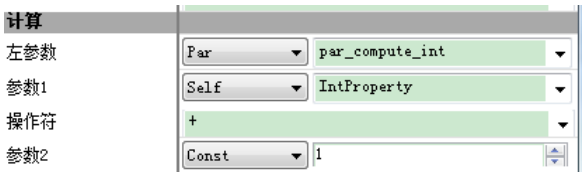
另外如果需要修改某些其他没有导出的属性，或做一些复杂的计算时，可以通过 Action 节点调用相应的函数类实现修改或计算。

具体的执行逻辑可以查看 src\behaviortree\nodes\actions\assignment.cpp

4.3 计算（Compute）



Compute 节点对属性，par 或函数的返回值做加减乘除的运算，把结果赋值给某个属性或 par。



其中左参数可以是 par 或者是某个 agent 的属性，参数 1 和参数 2 可以是常数、par，其他 agent 的属性、或者是方法调用的返回值。操作符可以是”+,-,*,/”。

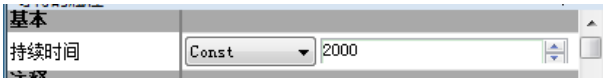
另外如果需要修改某些其他没有导出的属性，或做一些复杂的计算时，可以通过 Action 节点调用相应的函数类实现修改或计算。

但请注意这些操作的‘粒度’过小，大量这种小‘粒度’的操作可能对性能造成影响。请慎用。

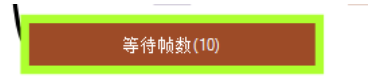
4.4 等待（Wait）

Wait 节点保持在 Running 状态持续在指定的时间（毫秒 ms），之后返回成功。

需要配置‘持续时间’，可以是常数，也可以是 par 或属性。



4.5 等待帧数（WaitFrames）



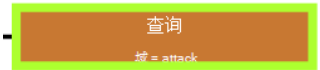
WaitFrames 节点保持在 Running 状态持续在指定的帧数，之后返回成功。

需要配置‘帧数’，可以是常数，也可以是 par 或属性。

4.6 空操作（Noop）

Noop 节点只是作为占位，仅执行一次就返回成功。

4.7 查询（Query）

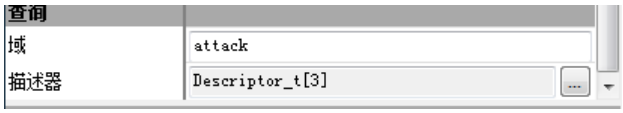


Query 节点是 behaviac 的扩展，大部分情况下请忽略。

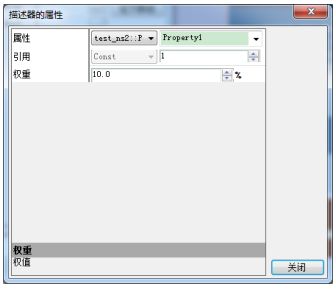
Query 节点试图提供一个动态的功能，例如，Query 节点可以用作如下场景：
当我们需要一个用于‘attack’的子树，而用于攻击的子树在创建拥有该 Query 节点的 BT 的时候可能还没有创建，可以配置该 Query 节点的 domain 为‘attack’，而所有需要用作‘attack’的 BT 的 domains 都配置上‘attack’，这样在执行的时候，Query 节点就从所有 domains 出

现了‘attack’的 BT 中挑选，然后按照配置的描述器来进行排序从而选择最合适的那个。

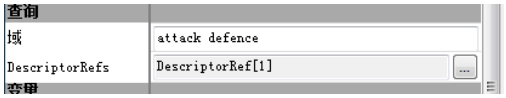
具体工作原理如下：
Query 节点有如下两个属性：



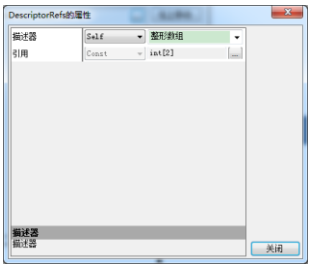
- 域 domain 是个字符串，用来表明类似标签一样的某种目的，比如分类，比如特征等。这里只是一个字符串。
- 描述器是个数组，每个元素是如下的一个结构，属性是某个 agent 的属性，引用是相应类型的参考值，权重是个百分比。



而每个 BT 也有如下的两个相关属性：



- 域 domains 是个字符串，用来表明类似标签一样的某种目的，比如分类，比如特征等。这里可以是多个空格分隔的字符串，表明该 BT 的某个特征，比如分类。
- DescriptionRefs 是参考值得数组。每个元素如下图：



Runtime 的执行逻辑如下图，对于每一个加载的 BT，查看 Query 节点中配置的 domain 是否出现在该 BT 的 domains 中，如果出现了，则对该 BT 用配置的描述器计算一个相似度，最后选择那个最相似的 BT 来执行。

```
const Query* pQueryNode = Query::DynamicCast(this->GetNode());

if (pQueryNode)
{
    const Query::Descriptors_t& qd = pQueryNode->GetDescriptors();
    if (qd.size() > 0)
    {
        const Workspace::BehaviorTrees_t& bs = Workspace::GetBehaviorTrees();

        BehaviorTree* btFound = 0;
        float similarityMax = -1.0f;

        for (Workspace::BehaviorTrees_t::const_iterator it = bs.begin();
            it != bs.end(); ++it)
        {
            BehaviorTree* bt = it->second;

            const behaviac::string_t& domains = bt->GetDomains();

            if (pQueryNode->m_domain.empty() || domains.find(pQueryNode->m_domain) != behaviac::string_t::npos)
            {
                const BehaviorTree::Descriptors_t& bd = bt->GetDescriptors();

                float similarity = pQueryNode->ComputeSimilarity(qd, bd);

                if (similarity > similarityMax)
                {
                    similarityMax = similarity;
                    btFound = bt;
                }
            }
        }

        if (btFound)
        {
            //BehaviorTreeTask* pAgentCurrentBT = pAgent->btgetcurrent();
            //if (pAgentCurrentBT && pAgentCurrentBT->GetName() != btFound->GetName())
            {

                const char* pReferencedTree = btFound->GetName().c_str();
                pAgent->btsetcurrent(pReferencedTree, TM_Return);

                return true;
            }
        }
    }
```

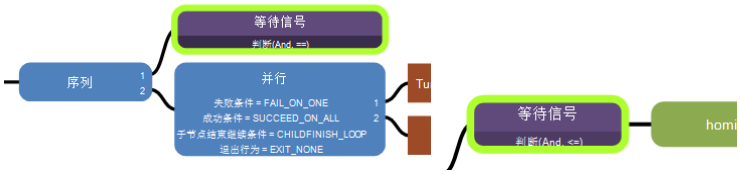
相似度是通过下述的代码计算获得，也就是对配置在 Query 节点上的每一个属性在 BT 上查找是否也配置了，如果配置了则比较在 Query 节点上的配置的参考值和 BT 上配置的参考值以及其权值得到一个数值即相似度。

```
const Property* Query::FindProperty(const Query::Descriptor_t& q, const BehaviorTree::Descriptors_t& c)
{
    BehaviorTree::Descriptors_t::const_iterator it = std::find_if(c.begin(), c.end(), PropertyFinder_t(q));
    if (it != c.end())
    {
        return it->Descriptor;
    }

    return 0;
}

float Query::ComputeSimilarity(const Query::Descriptors_t& q, const BehaviorTree::Descriptors_t& c) const
{
    float similarity = 0.0f;
    for (size_t i = 0; i < q.size(); ++i)
    {
        const Descriptor_t& qi = q[i];

        const Property* ci = FindProperty(qi, c);

        if (ci)
        {
            float dp = qi.Attribute->DifferencePercentage(ci);

            BEHAVIAC_ASSERT(dp >= 0.0f && dp <= 1.0f, "dp should be normalized to [0, 1], please check its scale");

            similarity += (1.0f - dp) * qi.Weight;
        }
    }
```

Query 节点上还可以添加若干判断 Predicate 附件作为重新 Query 的条件。

具体的执行逻辑可以查看 src\behaviortree\nodes\composites\query.cpp

4.8 等待信号（WaitforSignal）



WaitforSignal 节点模拟了一个等待某个条件的‘阻塞’过程。在上左图所示的情况下，该 WaitforSignal 节点‘阻塞’直到它上面附加的 Predicate 判断的条件是 true 的时候才结束‘阻塞’，从而继续序列中后面的节点。而如上右图中，则是它上面附加的 Predicate 判断的条件是 true 的时候，结束‘阻塞’而执行其子节点。

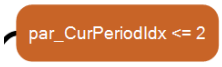
WaitforSignal 节点返回 Running，一直到它上面附加的 Predicate 判断的条件是 true 的时候，如果有子节点，则执行其子节点并当子节点结束的时候返回子节点的返回值，如果没有子节点则返回成功。

具体的执行逻辑可以查看 src\behaviortree\nodes\actions\waitforsignal.cpp

5 条件节点

条件节点不会返回 Running，条件节点的执行被认为是“即时”的，一个 exec 的执行要么返回成功 Success，要么返回 Failure，返回成功的时候视作 true，返回失败的时候视作 false。

5.1 条件（Condition）



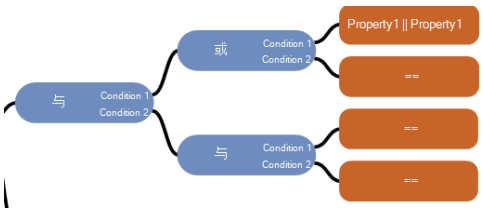
Condition 节点对左右参数进行比较，如果结果为 True，返回成功，如果结果为 False，返回失败，Condition 节点不可能返回 running。通常左参数是 par、Agent 的某个属性或 Agent 某个有返回值方法的调用，用户可以从下拉列表里选择，右参数是相应类型的常数、par 或 Agent 的某个属性。

Condition 节点没有提供取反的属性。如果需要取反，请用装饰节非（Not）。

参考条件（Condition）

具体的执行逻辑可以查看 src\behaviortree\nodes\conditions\condition.cpp

5.2 或（Or）及与（And）



Or 节点接受两个条件子节点，执行一个逻辑或，只要一个条件子节点返回值为成功（true），则返回成功，两个条件子节点都返回为失败（false）的时候返回失败（false）。
And 节点接受两个条件子节点，执行一个逻辑与，只要一个条件子节点返回值为失败（false），则返回失败，两个条件子节点都返回为成功（true）的时候返回成功（true）。

5.3 真（True）及假（False）

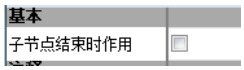
True 节点和 False 节点往往也是作为占位。在某些需要一个 Condition 的时候，可以拿 True 或 False 来占位或测试。

True 节点总是返回成功。False 节点总是返回失败。

6 装饰节点

装饰节点作为控制分支节点，必须且只接受一个子节点。装饰节点的执行首先执行子节点，根据自身的控制逻辑以及子节点的返回结果决定自身的状态。

装饰节点都有属性DecorateChildEnds可以配置：



如果 DecorateChildEnds 配置为 true，则仅当子节点结束（成功或失败）的时候，装饰节点的装饰逻辑才起作用。

6.1 输出消息（Log）

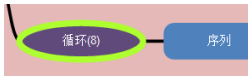
执行完子节点后，输出配置的消息。Log 节点可以作为调试的辅助工具。

6.2 非（Not）

该装饰器将子节点的返回值取反。如果子节点失败，那么此节点返回成功。如果子节点成功，那么此节点返回失败。如果子节点返回 Running，则同样返回 Running。

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratornot.cpp

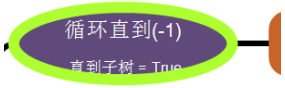
6.3 循环（Loop）



Loop 节点循环执行子节点指定的次数。当指定的次数到达后返回成功。在指定的次数到达前一直返回 Running。如果指定的次数是-1 则无限循环，总是返回 Running。

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratorloop.cpp

6.4 循环直到（LoopUntil）



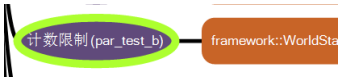
LoopUntil 节点除了像 Loop 节点可以配置循环的次数，还有一个属性‘直到子树’需要配置。

LoopUntil 节点有两个结束条件，指定的‘循环次数’到达或子树返回值和‘直到子树’值一样，只要一个条件满足就结束。

- 指定的‘循环次数’到达的时候，返回成功。
- 指定的‘循环次数’是-1 的时候，是无限循环，等同于只检查子树的返回值是否满足。
- 子树的返回值满足的时候：
  - 当‘直到子树=true’的时候，意味着直到子树返回成功，也返回成功。
  - 当‘直到子树=false’的时候，意味着直到子树返回失败，也返回失败。

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratorloopuntil.cpp

6.5 计数限制（CountLimit）



CountLimit 节点不同于循环节点。CountLimit 节点在指定的循环次数到达前返回子节点返回的状态，无论成功失败还是 Running。
在指定的循环次数到达后不再执行，CountLimit 节点的 onenter 直接返回 false。
如果指定的循环次数是-1，则无限循环，等同于什么操作都没有，只是执行子节点并且返回子节点的返回值。

此外，CountLimit 上还可以添加 Predicate 附件作为重新开始条件，请参考判断（Predicate）

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratorcountlimit.cpp

6.6 返回成功直到（FailureUntil）/返回失败直到（SuccessUntil）

FailureUntil 节点在指定的次数到达前返回失败，指定的次数到达后返回成功。如果指定的次数是-1，则总是返回失败。

SuccessUntil 节点在指定的次数到达前返回成功，指定的次数到达后返回失败。如果指定的次数是-1，则总是返回成功。

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratorfailureuntil.cpp 或 src\behaviortree\nodes\decorators\decoratorsuccessuntil.cpp

6.7 总是成功（AlwaysSuccess）/总是失败（AlwaysFailure）/总是运行（AlwaysRunning）

AlwaysSuccess 节点无论子节点返回什么，它总是返回成功。
AlwaysFailure 节点无论子节点返回什么，它总是返回失败。
AlwaysRunning 节点无论子节点返回什么，它总是返回 Running。

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratoralwayssuccess.cpp 或 src\behaviortree\nodes\decorators\ decoratoralwaysfailure.cpp 或 src\behaviortree\nodes\decorators\ decoratoralwaysrunning.cpp

6.8 时间（Time）及帧数（Frames）

在指定的时间内或帧数内执行子节点，返回 Running，当超出了指定的时间或帧数后，则返回成功。

具体的执行逻辑可以查看 src\behaviortree\nodes\decorators\decoratortime.cpp 或 src\behaviortree\nodes\decorators\decoratorframes.cpp

7 附件

附件类节点只能作为‘附件’添加到其他节点上。附件类节点不可能独立的添加到 BT 上去。

7.1 判断（Predicate）

Predicate 附件的功能就像是 Condition 节点，它的功能是做出比较，实际是个条件判断，返回成功（true）或失败（false）。

在节点类的代码里，可以提供下面的函数AcceptsAttachment来决定是否可以接受某类附件：

```
public override bool AcceptsAttachment(Type type)
```

如下图，每个Predicate附件有个Assocation用来表明本Predicate是如何和后续的Predicate共同起作用的。

需要指出的是，多个Predicate运算的时候，总是先计算最左（前面）的那个Predicate，

如：P1 And P2 Or P3，首先计算P1，

如果P1=true，由于接下来的是And，则计算P2，

如果P2也是true，则（P1 And P2） = true并且接下来的是Or，则直接返回true。

如果P2也是false，则（P1 And P2） = false，由于接下的是Or，则计算P3，如果P3=true，则返回true，否则返回false。

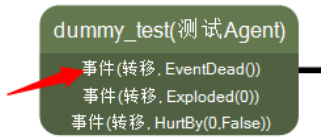
如果P1=false，由于接下来的是And，则直接返回false

目前 Action, Sequence，Selector, Query，CountLimit 和 WaitforSignal 可以接受 Predicate 附件，

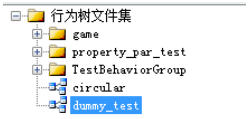
其他类型的节点暂不接受。

- 对于 Action 节点，附加的 Predicate 附件作为 precondition 的条件。只有当该 precondition 条件为 true 的时候，Action 节点配置的 Method 才会执行，如果该 precondition 条件为 false 的时候，Action 节点失败。特别需要说明的是，该 precondition 是每次执行 method 之前都会首先被执行。
- 对于 Sequence 和 Selector 节点，附加的 Predicate 附件作为打断序列的条件，当顺序的执行完一个子节点后则检查附加的 Predicate 节点，如果是 false，则返回失败，不再继续后面的子节点。
- 对于 Query 节点，附加的 Predicate 附件作为 ReQuery 的条件。
- 对于 CountLimit 节点，附加的 Predicate 附件作为重新计数的条件。
- 对于 WaitforSignal 节点，附加的 Predicate 附件作为‘阻塞’执行的条件。

7.2 事件（Event）



用鼠标拖拽 BT 到节点就可以添加 Event。当添加 Event 的时候，BT 的 Agent 类型需要保持‘一致’。如下图，可以用鼠标拖拽 dummy_test 到打开的某个 BT 窗口内的某个节点上添加 Event。



请参考 Event 作为附件