# 决策支持系统简介

在之前的文档中我们已经简单了解一下 「决策支持系统」了，我们知道了状态机好比角色的大脑，而决策支持系统为角色的眼睛和耳朵，其实说白了决策支持系统就是用来收集世界信息的一套工具。

假设我们写一个足球游戏中控球队员的 AI，如果我们不考虑复杂的战术配合，一般来说，控球队员要么带球突破，要么直接射门，要么传球给会造成威胁的空位球员，考虑到足球场上瞬息万变情况来，如果单纯的 if - else，会很难罗列出全部的条件，所以我们可能会采用模糊 AI 的决策逻辑：比如势力图、分数系统、概率统计工具等等。我们来分析一下在下面这个问题中，AI 决策时候可能需要知道哪些游戏世界信息：

1. 场上己方球员的分布
2. 场上对方球员的分布
3. 球场上哪里比较有威胁（靠近球门的地方）
4. 球场上哪里比较不安全（比如有强力防守队员，或者防守队员人数很多）
5. 场地的构成（边线，禁区）
6. 球员的相关信息（比如球员能力、位置、当前行为等等）

如果把上面的信息分个类别的话，可以分成以下 4 种：

* 静态实体信息（比如 5）
* 静态抽象信息（比如 3）
* 动态实体信息（比如 1，6）
* 动态抽象信息（比如 4）

静态和动态的概念比较好理解，「静态」就是值不随着游戏的进行而变化的信息，「动态」就是随着游戏的进行会一直改变的信息，像场地信息就是静态的，不会改变的，像场上对方球员分布就是属于动态信息，因为他们的位置是一直变动的。而我这边提到的「实体」信息，指的是「真是存在」的信息，真是存在「抽象」是指「自定义」的参考信息。像场地信息，就是实体信息，因为类似禁区位置都是实际存在的信息，但像球场上哪里有威胁，那就是我们根据需要，自己定义的信息了，可以不断的调整和修正。而决策支持系统就是收集、记录上述信息的一套工具，常见的有：势力图，分数系统。

# 势力图

在 势力图的简介与应用 这篇文档中，我们已经详细地讨论过势力图的相关概念以及其在寻路上面的应用。这篇文档中，我们主要用上述的足球场例子再来讨论一下势力图是如何应用的。

首先我们按需求将游戏地图栅格化，比 10 × 10，当然，粒度的大小取决于对于精度和效率的平衡。每一个格子就相当于「帧缓冲区」中的像素，然后我们可以创建多个这样的势力图，和创建多个「帧缓冲区」一样。每个图都代表上述信息中的一项内容，图中的每个格子都根据信息的内容填入 0.0 ~ 1.0 的值 （这个值的范围可以根据需求自定）。

例如，我们要建立一个「场地威胁图」，我们定义 0 表示完全没有危险，1 表示威胁值最高，那我们就可以这张图的相应的格子中填入相应的值，而且因为这是静态信息，所以只需要在游戏开始时填入就可以了，当我们填完每一个格子的时候，我们就得到了这样一张「场地威胁图」。对于动态信息的情况，我们需要每隔一段时间对「图」中的信息做一次更新，比如「场地威胁图」，就是这样的动态「信息图」，需要根据防守队员的情况来实时更新。这样当我们填完所有的图信息后，AI 决策时就可以知道任意时刻，在地图上的任意点上的相关信息了。

小地图的情况（如上例）可以直接做栅格化，但对于地图比较大的情况，如果直接栅格化的话，更新起来性能太低，这种情况可以考虑采用层次化的图模型，先将地图分成大块的格子，在大的格子里再细分成小格子，当查看距离近的信息的，采用精细的格子信息，查看远处的时候，采用粗略的信息，这样就可以在效率上取得一些平衡。

用势力图来表示世界信息的另一个好处是，可以方便的将信息绘制出来（在地图上，或者在外部的调试工具中），而不用面对一大堆的数据，如果再将不同的值配以不同的颜色来显示的话，那将大大的降低 AI 调试的难度。

# 分数系统

为了方便说明分数系统，我们先来设计一个场景，有一个单兵作战的士兵，在丛林中，他有一些基本的行为，比如射击（shoot），移动（move），撤退（retreat）。他必须结合很多因素来决定自己的行为，比如，敌人的数量、自身的情况、弹药的情况、环境的情况等等，我们现在就来为这个士兵写 AI 来模拟他的行为。AI 常常用到一些 if - else 的逻辑判定（这种方式，在 AI 中叫基于规则的 AI，Rule - Based System），对一些简单并且确定的逻辑，这种方式十分的便捷有效。但在上面这个场景中，如果单纯的用 if - else ，会使代码很快变的杂乱和难以维护，我们可以试着写一下，假设我们需要考虑以下两个个要素：

* 敌人的数量（Enemy Count，EC)
* 自己的负伤程度（Hit point，HP，用 0～100 来表示）

我们定义以下规则：

* 规则 1. 如果 EC = 0，那么 Move
* 规则 2. 如果 HP < 30，那么 Retreat
* 规则 3. 如果 EC > 4，那么 Retreat
* 规则 4. 如果 0 < EC <= 2 并且 HP >= 30 那么 Shoot
* 规则 5. 如果 2 < EC <= 4 并且 HP > 70 那么 Shoot
* 规则 6. 如果 2 < EC <= 4 并且 HP <= 70 那么 Retreat

这些规则可以转换成相对应的 if - else，但这样一个简单的情形（两个条件，三个结果），我们就需要 6 条规则来描述，如果我们需要其他一些考虑因素，比如：弹药、环境，那用这样的结构就很难维护了。对这种多因素判定的情况，我们就可以尝试用分数系统来做。分数系统的一个基本思想，**是为每个单独的行为打分，根据分数的高低来决定做哪个行为，而所有的因素就是打分的依据**。上面这个场景中，我们有三个独立行为，在每次的 AI 选择中，我们来为这些行为打分，下面我们来试着把这个士兵的 AI 用分数系统来做：

假设我们的分数是从 0～100，Clamp (v, l, h)是指如果 v < l，则 v = l，或者如果 v > h，则 v = h，Max 值取两者较大的那个值，用伪代码描述如下：

* Move：s = Clamp ( 100 \* (1 – EC), 0, 100)
* Retreat：s1 = Clamp (100 – HP, 0, 100)  
  s2 = Clamp (25 \* EC, 0, 100)  
  s = (s1 + s2)/2
* Shoot： s1 = Clamp (HP, 0, 100)  
  s2 = Clamp (100 – 25 \* Max (EC, 1), 0, 100)  
  s = (s1 + s2)/2

简单做一些测试：

* HP = 60, EC = 2  
  Move (0), Retreat (45), Shoot (55) –> Shoot
* HP = 20, EC = 1  
  Move (0), Retreat (52.5), Shoot (47.5) –> Retreat
* HP = 100，EC = 5  
  Move (0), Retreat (50), Shoot (50) –> Shoot or Retreat
* HP = 70, EC = 0  
  Move (100), Retreat (15), Shoot (72.5) –>Move

分数系统的打分公式，是根据情况自己定义的，但也有一些方式来帮助构建出比较合理的公式来。公式的推导就是一个函数的拟合过程，我们可以先取几个样点，然后试着找到一条函数曲线，尽可能的去使样点落在我们的这条函数曲线上。在中学的时候的实验课上，大家应该都做过这种事情。在AI 中，我们的打分公式，一般不会很复杂，可能是简单的直线、分段的折线，有时也会是二次、三次的曲线。有了这样一个基本曲线，接着再用测试样例去修正，一般就可以得到一个比较符合需求的打分公式了。

在分数系统中，如果有多个因素需要考量，比如我们上个例子中的 EC 和 HP，这就引出了分数如何组合的问题。就像我们先前对 Retreat 和 Shoot 的打分方式。常用的做法是，针对每个因素单独引入打分公式，并且归到 [min, max] 的分数区间中，然后再对所有的得分组合后，得到一个最终得分。对每个因素单独打分的好处是，我们可以跟专注于单个因素的影响，方便我们构建打分公式，并且可以方便的移除和添加新的因素，而且也能更好的复用我们的打分公式。分数的组合一般可以有两种：

* 加权平均：s = (k1 \* s1 + k2 \* s2 + … + kn \* sn)/(k1 + k2 + … + kn)
* 因子连乘：s = s1 \* s2 \* … \* sn ( 0 <= si <= 1.0 )

这两种方式有各自的适用范围，第一种方式，我们除了对打分公式要细心调整外，还需要对加权因子加以调整。第二种方式必须保证每个因子被归到 [0,1] 之间，因为用到了乘法的原因，所以体现了一种并且的关系，一旦有一个因子为 0，则整个分数就会是 0。

分数系统是一种模糊逻辑，他不像基于规则的系统那样逻辑非常清楚，所以，在分数系统中，有时会出现一种「意外点」，但对 AI 来说，出现一些意外点，也不是不可接受，有时这样也更显得 AI 很真实，AI 也会犯错的嘛。一旦使用了分数系统，debug 的过程就变成了不断调整公式和数值的过程。如果设计的好的话，AI 程序员可以提供出一套调节分数的接口，这样 design 也可以参与其中了。  
当然，有时分数系统是会和规则系统联合使用的，用来处理一些意外情况。我常觉得，在 AI 中，规则系统就是用来修 bug 的。