国际象棋程序设计(一)：引言

François Dominic Laramée/文

这是国际象棋程序设计连载的第一部分，本连载共有六部分，他们不仅仅是针对象棋【译注：以后如不特别指出，都指国际象棋】的，还可以运用到别的益智类游戏中。

在人工智能领域中，专家对象棋进行了大量卓有成效的研究，这其中就有电脑对抗卡斯帕罗夫等世界冠军的比赛。象棋在人工智能上的地位，就相当于果蝇在遗传学上的地位，举足轻重。我的连载将着重介绍人工智能的程序设计艺术，这其中包括“深蓝”(Deep Blue)等著名程序。

我的连载从2000年4月开始，每个月一次，到10月结束的时候，我会用Java写一个简单的程序来实现对弈。到时候你们可以从我的网站上随便下载，耐心地等吧。

信息完备的游戏

象棋是“信息完备”的游戏，因为游戏双方面对的局面是同一个局面，任何一方所掌握的棋子及其位置的信息是一样的。除了象棋以外，西洋跳棋(Checkers)、围棋(Go)、五子棋(Go-Moku)、西洋双陆棋(Backgammon)、黑白棋(Othello)【也有称Reversi的，可译为“翻棋”】可等也属于这一范畴。但是纸牌游戏却不是，因为你不知道对手手上的牌是什么【在中国的棋类游戏中，陆站棋(起源于欧洲)和四国大战棋也不是】。

我的连载中将提到各种算法，大多数算法对所有的信息完备的游戏都是有效的，只是细节上有所不同罢了。很明显，无论棋盘、着法、位置等因素有那些，搜索算法就是搜索算法，它不会因为游戏规则而改变。

我们需要什么?

能下棋的电脑软件至少要包括下列组件：

1. 棋盘的表示方法，即局面在存储器中的存储方法，程序是根据它来分析局面的；
2. 掌握规则，即什么样的着法是合理的，如果程序连不合理的着法都不能检测出来，那么对手就可以利用这种着法来欺骗程序；
3. 找出所有合理着法的算法，这样程序就可以从这些着法中找到最好的，而不是随便找一种着法；
4. 比较方法，包括比较着法的方法和比较局面的方法，这样程序就可以选择最佳的着法；
5. 用户界面。

本连载将涉及以上除了用户界面以外的所有内容，用户界面在所有二维棋类游戏中都是差不多的，这里就不作介绍了。接下来将对以上几个部分作逐一介绍，并且引出许多重要的概念。

棋盘的表示方法

在早期的程序设计过程中，存储器是非常有限的(有些程序只用8K或更少的存储器)，所以最简单、最节省的表示方法就是最有效的方法。一个典型的国际象棋棋盘可以用一个8x8的数组表示，棋盘上的每个格子用一个字节表示——空的格子用0，黑方的王用1，等等。

如今，象棋程序可以在64位计算机上运行了，精心设计的“位棋盘”表示诞生了。在60年代后期，位棋盘在苏联诞生，一个位棋盘由一个64位的字【“字”是计算机中一次运算所涉及的存储单元，我认为当时还没有字长为64位的计算机，所以一个位棋盘应该由多个较短的字来构成，如8个8

位的字或4个16位的字，即便是现在的个人电脑上，一个位棋盘也必须由两个32位的字构成】来表示局面中的某个状态，一位代表一个格子。例如，一个位棋盘能表示“所有被黑兵占有的格子”，或者“处于e3格的后可以走的格子”，或者“被黑马攻击的白子所处的格子”，等等。位棋盘用途广泛，并且具有很快的运算速度，因为局面分析时要做大量的逻辑运算【就是“与或非”运算，也称布尔代数】，而一个位棋盘的运算只需要一次操作就可以了。

这部分内容将在连载的第二部分作详细介绍。

着法的产生

所谓棋类游戏的规则，指的就是某一方可以走哪些着法。某些游戏很容易就找到合理着法，例如在井字棋中【Tic-Tac-Toe，在3x3的棋盘上轮流占格子，先在同一条线(横线、纵线或斜线)上占有3枚棋子者得胜】，所有空的格子都是合理着法。但是在象棋中，情况就有些复杂了，每个棋子都有它特定的着法，例如兵吃子时走斜线，而不吃子时走纵线，又例如把王走到被将军的格子是不允许的，还有诸如“吃过路兵”【法语en passant】、兵的升变、王车易位等着法只有在特殊场合才是合理的。

事实上在象棋程序设计中，着法的产生已经被证实为最复杂和最费时的事。幸运的是，着法的产生有一些预处理的技巧，我还会介绍一些数据结构，它们能显著提高着法产生的速度。

这部分内容将在连载的第三部分作详细介绍。

搜索技巧

对于计算机来说，判断一个着法是好的或坏的，并不是件容易的事。判断着法优劣的最佳办法，就是看它们的后续结果，只有推演以后一系列的着法，才能确定看那步是好的。在保证不犯错误的前提下，我们要设想对手会走出最好的应着。这一基本原理称为“最小-最大”搜索算法，它是所有象棋程序的基础。

不幸的是，“最小-最大”法的运算量是$O(b^n)$【数学上指它和b^n是一个数量级的】，b(分支因子)指平均每步的合理着法【有研究表明，在国际象棋中这个值约为38，中国象棋则更多些，为42(这是中国象棋程序“七星大师”的作者赵德志的研究结果)】，n(深度)指思考的步数(一回合有两步)。所以当步数增长时，运算量以几何级数迅速增长，如果要思考得更深一些，就必须用更为合理的算法。其中，迭代加深的Alpha-Beta搜索、NegaScout搜索和MTD(f)搜索是最基本的算法，这些会在连载的第四部分介绍。除此以外，还要依靠数据结构和启发式算法的帮助，启发式算法是增强棋力的算法，例如置换表(Transposition Tables)、历史启发和将杀启发(History/Killer Heuristic)等。

在象棋程序设计中，另一个头痛的问题是“水平线效应”(Horizon Effect)，它首先由Hans Berliner提出。假设你的程序的搜索深度是8步，并且程序算出对手会在第六步捉死你的后。按照程序自己的设想，它会献出象来延缓后的捉死(假定这样做可以让后在第10步才被捉死)，因为程序只能看到8步。从程序的角度看，后是被“保住”了，因为在它的搜索范围内后没有被捉死，但事实上却多丢了一个象。从这个例子可以看出，要把程序的工作重心放置到位，并不是一件简单的事情【意思是，某些变化没有必要搜索得太深，而关键的变化需要更深层次的搜索】，如果把每条变化都搜索到同一深度，那么结果是自取灭亡。很多技术是用来克服水平线效应，在连载的第五部分关于高级搜索的阐述中，将要介绍静态搜索(Quiescence Search)和深蓝(Deep Blue)的单步延伸(Singular Extensions)。

评价局面

最后，程序必须有一个估计局面(占优或处于劣势)的方法。局面估计方法完全取决于规则(即子的走法)——在象棋中，“子力平衡”(Material Balance)是主导因素，因为一个小小的兵的领先就可能足以保证优势一方取得胜利【而在中国象棋中，多一个兵算不了什么，这足以证明本节的观点

——局面估计方法完全取决于规则】，而在五子棋中却没什么影响，在黑白棋里，根据子力上的数值分析局面则完全会成为误导，你可能一直处于领先状态，但最后几步局面却翻了盘。

建立有效的局面评估方法，这常常会成为程序设计中的难点和焦点。连载的第六部分将详细阐述著名象棋程序的局面评估方法，其中包括Chess 4.5、Cray Blitz和Belle(尤物)。

结论

我们已经找到了完成拼版的所需要的碎片，现在是开始动手做的时候了。下个月我将介绍最常用的棋盘表示方法，敬请关注。

François Dominic Laramée，2000年4月

原文：http://www.gamedev.net/reference/programming/features/chess1/
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 棋弈软件基础——残局库对引擎棋力的负面影响
- 下一篇 国际象棋程序设计(二)：数据结构
- 返　回 象棋百科全书——电脑象棋

国际象棋程序设计(二)：数据结构

François Dominic Laramée/文

上个月我简要介绍了象棋程序设计中所需要的知识，其他信息完全的双人游戏也是一样的。现在我们将讨论一些细节——棋盘的内部表示方法。

在棋盘表示方法这个理念上，近三十年内没有多大发展，你可能会觉得很吃惊。它的发展需要智慧的推动，很早就有人提出过绝妙的方案了，但同时也受到制约，因为这些方案需要数据结构的支持，某些数据结构至今还没有实现。

尽管如此，我还是会介绍三种数据结构，尽管它们并不是必需的，但是对于提高你的下棋水平是大有帮助的。其中两种可以加快思考速度(但是其中一种需要无限多的存储器)，另一种可以加快着法产生速度。【译注：分别指后面要提到的置换表、历史表和着法生成预处理的数据库。】

在我们继续讨论之前，我提一句格言：“无论是象棋还是其他游戏，你通常使用的数据结构，应该是能达到目的的最简单的数据结构。”然而象棋程序设计者提出了很多技巧，它们能让程序运行的更快，其中相当多的还适用于其他游戏。如果你对你要设计的游戏不很了解，而且手头的资料很有限，那么你应该好好掌握我所提到的这些技巧，你可以把这些技巧试验到你的程序上。

基本的棋盘表示

在上世纪70年代，个人电脑的存储器是稀有资源，所以棋盘表示得越紧凑越好。很多人会很自信地说：用一个64字节的数组，每个字节表示棋盘上的一个格子，用一个整数就可以表示格子的位置了。(任何棋盘的数据结构都必须用一些额外的字节，来记录吃过路兵的目标格、王车易位权利等信息，但是这里我们暂且忽略它，因为这些因素可以独立处理，而且处理方法大同小异。)

后来又流行一些更优化的算法：

1. 早期的SARGON【一个象棋程序】扩展了64字节的数组，在它的外围加了两圈“虚格”，并在这些格子上作了非法标记。这一技巧能加快着法产生的速度，例如象在走棋时会延斜线滑行，直到走到虚格上才中止。这样就没有必要用复杂的运算来预先判断象到达的格子是否会超出存储器上的表示了。第二圈虚格是给马用的，例如位于盘角的马试图跳出棋盘，这就必须用两圈虚格来保护。

2. MYCHESS用了相反的方法，它只用32字节表示一个棋盘，每个字节代表一个棋子，例如代表白方王、黑方王翼马前兵【英文居然是black King's Knight's Pawn，一开始让我大惑不解】等，它存储的信息是棋盘上的位置，或者已经被吃的标记。这种技术有一个缺点，它无法描述由兵升变而来的其他棋子(同时这个棋子在棋盘上还有一个)。在后来的版本中，这个缺点被修正了。【其实这并不难办，一个字节取值从0到255，通常255表示该子已被吃，从0到63表示该子在棋盘上的位置，兵通常是升变成后的，那么从64到127可以表示该子已经升变为后，如果升变为车、马或象，则需要额外的字节来处理。】

如今，这种极端吝啬的数据结构只可能出现在掌上电脑、手机或电视机机顶盒上，它们的80~90%的资源都被操作系统占用，没有额外的空间给游戏使用。但是，某些游戏却别无选择地使用这种方法【我也不知道什么游戏非得使用这种方法不可】。

想更多地了解以前的象棋程序，可以看一下David Welsh在1984年写的《计算机象棋》(*Computer Chess*)一书。

位棋盘

用一个数组来表示棋盘，对于很多游戏来说，就找不到更好的办法了。但是对于像象棋、西洋跳棋之类在64格棋盘上的游戏来说，有一个高明的技巧——“位棋盘” (首先由苏联的KAISSA制作组提出)，在上世纪60年代末诞生了。

KAISSA是运行在64位处理器上的程序【我很怀疑当时就有64位处理器，或许当时处理器字长的概念和现在的有所不同】。“64”恰巧是象棋棋盘格子的数目，所以这就有可能让一个字来表示一个棋盘，以判断某个格子的状态是“是”或者“非”。例如，一个位棋盘可以回答棋盘的每个格子“是否有白子”。【把位棋盘运用到中国象棋上，这是我将要进行的一个课题，中国象棋的棋盘有90个格点，可以用3个32位的字来表示。】

因此，一个完整的局面需要用12个位棋盘表示：白兵、白车、黑兵等等。再加上两个用来表示“所有白子”和“所有黑子”的位棋盘，以加快运算速度。【其实只需要8个就可以了，同一兵种(不管黑白)用一个位棋盘，一共是6个，再加上代表“所有白子”和“所有黑子”的。做得更过分的是，有人把象和后并作一个位棋盘，车和后并作一个位棋盘，这样又可以减少一个。如果要表示白方的象，只要“所有白子”、“所有车或后”的补集(用“非”运算)、“所有象或后”这三个位棋盘作“与”运算就可以了。】或许你还想用一个位棋盘表示被某种子力攻击到的格子，诸如此类，这些位棋可以盘灵活运用在着法产生的运算过程中。

位棋盘之所以有效，是因为很多运算可以转化为处理器的一个逻辑指令。例如，假设你需要确认黑王被白后将军，如果用简单的数组来表示棋盘，那么你需要这样做：

1. 首先找到后的位置，这需要从64个字节中一个一个地找；
2. 然后在后所能走的八个方向找，直到找到王或者找到后走不到的格子为止。

这些运算是相当花费时间的，如果后碰巧是在数组的最后一格，更糟的是，将军只会发生在少数情况下【这种运算纯粹是白费时间！】。

如果用位棋盘，那你只要这样做：

1. 载入“白方后的位置”的位棋盘；
2. 根据这个位置，从索引数据库中找到被后攻击的位棋盘；
3. 用这个位棋盘和“黑方王的位置”的位棋盘作“与”运算。

如果结果不是零，则说明黑王被白后将军。假设被后攻击的位棋盘是储藏于存储器中的【这是上面提到的第二步的前提】，那么整个操作只需要3到4个时钟周期【通常计算机执行1条指令需要1(算术或逻辑运算)到2(寻址操作)个时钟周期】。

【这里允许我发挥一下，原作所说的“从索引的数据库中找到”(即上面提到的第二步)，其实并非是简单的一步，对于后的每个位置，都有一定的攻击格子(从边线到中心依次是21、23、25和27格)，但是要考虑被别的子阻挡的情况，程序无法为所有的情况都作索引，最多只能对某条线(横线、纵线或斜线)的棋子占有情况作索引，这也需要$2^8 = 256$ 种情况，再加后本身有64种位置，所以即使这样，数据库中也至少要保存256x64 = 16384个位棋盘。另外，横线、纵线和两条斜线的处理方法各不相同，横线只要作简单的“移位运算”就可以了，而纵线和两条斜线都要用到“位棋盘旋转”的技术，为了提高运算效率，程序的复杂程度是惊人的，细节可以参考《对弈程序基本技术》专题之《数据结构——旋转的位棋盘》一文，文中作者多次提示读者用咖啡来提神，以完成烦琐的程序。】

再举一个例子，如果在当前的棋盘上，你要产生白马的所有着法，那么你只要找到当与前位置相关联的“马能走到的格子”的位棋盘，并“与”(AND)上“所有被白方占有的格子”的位棋盘的补集(就是对这个位棋盘作“非”(NOT)运算)，因为马的着法限制仅仅在于它不能去吃自己的子。【国际象棋比较简单，而中国象棋中还有“绊马腿”的限制(还有象也是)，这种情况该怎样使用位棋盘，也是我将要研究的课题。】

如果你想更详细地了解位棋盘(也只是稍微详细一点而已)，可以去看看描述CHESS 4.5 (它是由美国西北大学开发的)的文章——Peter Frey写的《人脑和电脑的象棋技巧》(*Ches Skill in Man and Machine*)，现在至少已经出了两版了，分别出版于1977年和1981年。

值得注意的事，到今天为止，几乎还没有真正意义上使用64位处理器的个人电脑，所以位棋盘的速度优势是要打折扣的。尽管如此，位棋盘的技术仍是行之有效的。【苹果公司的Macintosh图形工作站据说是64位处理器，但不知有没有针对这种电脑的象棋软件。时隔4年，到我翻译这篇文章时，还没有什么别的64位处理器用在个人电脑上。因为毕竟没这个必要，除非你专门用它来玩国际象棋。】

置换表

在象棋里，有很多着法可以到达相同的位置。例如，不管你先走 1. P-K4 ... 2. P-Q4或者1. P-Q4... 2.P-K4，【这里K4和Q4即e4和d4的，在实战中有这样的例子，1. e4 e6 2. d4和1. d4 e6, 2. e4是形成法兰西防御一种变例的两种途径。】最终局面是一样的。有不同的路线可以达到同样位置的，这种现象称为“置换”(Transposing)。【在中国象棋中，置换现象更为普遍，通常用成语“殊途同归”来称呼这种现象。】

如果你的程序已经对1. P-Q4... 2.P-K4产生的结果竭尽全力作了搜索和评估，那么你最好记住这个结果，以避免碰到1. P-K4... 2.P-Q4时再作同样的运算。自上世纪60年代Richard Greenblatt的Mac Hack VI问世以来，所有的对弈程序都会吸纳“置换表”这一技术，这就是原因所在。

置换表存储的是已经搜索过的结果，它通常使用类似于散列表(Hash Dictionary)的数据结构来达到最快的查询速度。在搜索某个局面时，结果(包括局面分析的值、搜索深度、最佳着法等)就存储到置换表里。当搜索新的局面时，我们先从置换表里找，如果已经有这种局面，那么就可以利用它，省去重复的搜索。

这种处理有以下很多好处：

1. 加快速度。在置换情况发生得很多时(特别是残局局面里，棋盘上棋子很少时)，90%以上的局面可以在表中找到。【在中国象棋中，这优势时更为明显，因为它的子力密度小，在开局阶段就有很多“殊途同归”的现象。】

2. 任意深度。假设你需要对某个局面搜索到一个指定的深度，例如4步(也就是两个回合)，如果置换表里有这个局面而且已经搜索了6步，那么你不仅可以跳过这次搜索，还可以得到比预期更精确的结果。

3. 用途广泛。通常每个象棋程序都配有“开局库”(Opening Book)，即包含一些常见局面及其最好的着法，这些通常是从已有的记载中提炼的【例如特级大师们写的“象棋开局大全”或“象棋开局手册”之类的书，而我本人更倾向于从大量对局记录中提炼的结果】。有了开局库，程序就不必在开局阶段就做傻事了【因为前人已经做过无数次的计算了】。既然开局库的操作过程和置换表是一样的(即搜索局面)，那么为什么不在棋局一开始就把开局库装入我们的置换表里去呢？如果这样做，即使棋局暂时脱离了开局库，后来又回到开局库里的局面【要注意，这个局面可以是棋局过程中出现的局面，但更多的情况是搜索程序推演到的】，那么置换表里保留了这样的局面，我们仍旧有机会用到它。

置换表唯一的缺点就是它贪婪地占用着存储器，无论如何它至少需要存储成千上万个局面，百万甚至上亿就更好了。如果每个局面用16字节【用最紧凑的办法至少也需要32字节(用MYCHESS这种吝啬的办法)，但是这里可以存放“散列键值”，下面会提到这个技术】，那么在存储器紧缺的时候这将成为很严重的问题。

置换表还有其他用途。

CHESS 4.5还用散列表来存储其他的局面计算结果【指下面提到的兵型、子力平衡等】，这些计算结果在多数情况下是不会变动的，例如：

1. 兵型(Pawn Structure)。散列表只存储兵的位置，这需要较小的存储空间，由于兵的移动不会很频繁，所以99%的兵型局面可以在散列表中找到；【国际象棋的局面分析中，需要考虑连兵、叠兵、孤兵、通路兵等因素，这些计算是相当费时的，而中国象棋就没有这个必要了。】

2. 子力平衡(Material Balance)，即棋盘上子力的相对强弱，它只在吃子或兵升变时变动。

在CPU速度不快的今天，这些方法或许看来用处不是很大，但是它们还是有指导意义的，一些稍微花费存储器的预处理可以节省相当多的计算。【因为寻址操作(特别指在若干M的大范围区域内寻址)所占用的时间要远多于简单的运算指令，如果哪天寻址操作的速度接近于基本运算了，那么这种技术将会对程序运行速度有很大的提升。】如果你仔细研究你的程序，或许你会发现这方面是值得改进的。

为棋盘产生散列键值

上面提到的置换表的结构，是由散列表来实现的，由此引发了以下课题：你如何快速而有效地从棋盘来产生散列键值(Hash Key)?

以下是1970年Zobrist的方案。

1. 产生12x64个N位的随机数(如果置换表能储存2^N个局面)，并把他们做成一张表。每个随机数只跟某个位置上的某种棋子有关(例如H4格的黑车)，空的格子用零表示；【因为棋子总共有12种，棋盘总共有64格，所以是12x64个数，空的格子不用随机数而用0表示。】
2. 先把散列键值设为零；
3. 搜索整个棋盘，每遇到一个子，就在原来的散列键值上“异或”(XOR)它所对应的随机数，重复这个过程直到整个棋盘被检查一遍。

这个方案有趣的一面在于，每走一步，散列键值的更新都非常容易，不需要重新搜索整个棋盘。你知道“XOR图形处理”(XOR-Graphics)吗？某个图形用XOR运算作用到背景上，然后再作同样一次运算，不就又回到背景了吗？这里也同样这么做。【如果你熟悉图形操作中的技术，就不难理解了，原文把这个过程比作“位图”(Bitmap)操作，12x64个棋子的状态就对应12x64张位图，原先的散列键值被比作“背景”(Background)，只要在背景上作位图的粘贴操作(用的是XOR处理)就可以了。】举个H1格的白车吃掉了H4格的黑兵的例子，要产生这个散列键值，先XOR“在H1格的白车”这个随机数(把这个图形擦掉)，然后是“在H4格的黑兵”(把这个图形也擦掉)和“在H4格的白车”(粘贴一个新的图形，代表新的车的位置)【其实顺序是无关紧要的】。

用相同的方法，不同的随机数，可以产生第二个散列键值，或称“散列锁”(Hash Lock)【在英语中Lock(锁)和Key(钥匙)是对应的】，它是在置换表中存储的真正有用的信息。这个技术是用来检测冲突的，如果恰巧有两个局面具有相同的散列键值，那么他们具有同样的散列锁的几率是微乎其微的。

历史表

“历史启发”(History Heuristic)是“杀手着法”(Killer Move)【杀手着法指能产生截断的着法，以后的连载会提到的】技术的衍生技术。一篇研究性的文章是这么解释的，历史表用来保存这些着法，在过去的搜中非常有意义(因为使用高效搜索技术的而对它进行了很深的搜索)，这个着法今后还可能用到。历史表由一个64x64的整数数组构成【着法的起始格和到达格，共有64x64种组合】，记录每种着法的数值，当搜索算法认为某个着法很有用时，它会让历史表增加这步的数值。表中的数值是用来对着法排序的，“历史上占优势”的着法会优先考虑。

着法产生的预处理

着法的产生(即决定特定位置下那些着法是合理的)和局面的估计一样，是象棋程序设计中计算量最大的部分。因此，在这个方面的一点预处理会对提高速度大有帮助。

我个人喜欢Jean Goulet在1984年写的《象棋的数据结构》(*Data Structures for Chess*，McGill大学出版社)一书中提到的方案，它概括为：

1. 出于着法产生的目的，棋子的颜色是无关紧要的，除了兵以外，它只朝对面走；

2. 对于基本的子力和它的位置，有64x5=320种组合【指除了兵以外的5种棋子，根据上一条，这些棋子是不分黑白的】，黑兵有48个格子可以放(他们后面一行是走不到的，并且一到第八行就会变成别的棋子)，白兵也有48个格子；

3. 从某个格子沿某个方向，可以确定一条着法“射线”，例如，后从H3格朝北走，这就是一条“射线”；

4. 每个格子上的每个棋子，都有确定的几条射线可以走，例如位于中央的王可以朝8个方向走，而位于盘角的象却只有一条逃生的路；

5. 在开始游戏之前，要计算数据库里在所有格子的所有棋子的所有射线，假设棋盘是空的(即着法只受到棋盘边缘的限制，不受其他棋子的限制)；

6. 当你为某个格子的某个棋子产生着法时，朝每个方向搜索直到碰到棋子为止。如果是对方的棋子，最后一种着法就是吃子的着法，如果是本方的棋子，最后一种着法就是不合理的。

有了恰当的数据库，着法的产生就变成简单得接近于线性的寻找了，几乎用不着什么计算。整个事情就掌握在这么几千个字节里，只是置换表的一个零头。

以上提到的所有技术(位棋盘、置换表、历史表和预处理数据库)都会反映在我自己的程序中，当我写完这个连载以后就会发布出来。下个月我会详细介绍着法产生的方法。

François Dominic Laramée，2000年6月

原文：http://www.gamedev.net/reference/programming/features/chess2/
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 国际象棋程序设计(一)：引言
- 下一篇 国际象棋程序设计(三)：着法的产生
- 返 回 象棋百科全书——电脑象棋

国际象棋程序设计(三)：着法的产生

François Dominic Laramée/文

上个月，我完成了连载的第二部分，介绍了在着法产生的预处理中涉及的数据结构。现在我们把话题回到着法产生，详细介绍两种着法产生的策略，并解释一下在特定的情况下如何在这两个策略中作出选择。

困境

不管你怎么对付象棋，它就是一个复杂的游戏，对于电脑来说就更是如此了。

在通常局面下，棋手要在30多种着法中选择，有些是好的，有些则是自杀棋。对于受过训练的人来说，他能判断出大多数愚蠢和没有目的着法，甚至初学者都知道，后处于被吃的位置时该怎么逃跑。专家(多数是通过直觉而非推理)则知道哪几步棋会比较有力。

但是，把这些信息(特别是无意识的那些)编写到计算机里去，被证明是极端困难的，连那些最强的象棋程序(除了个别程序以外，例如Hans Berliner的Hitech和它的姊妹程序)都已经放弃这条路线了，取而代之的是“蛮力”——如果你能以足够快的速度分析所有的着法，并且足够深远地预测他们的结果，无论你是否有一个清晰的思路，你最终会走出一步好棋。所以，着法的产生和搜索必须越快越好，以减小蛮力方法的花费。

搜索技术将在连载的第四和第五部分介绍。这个月，我们会关注着法产生。历史上有以下三条主要的策略：

1. 选择生成：检测棋盘，找到一部分可能的着法，把其他的全不要；
2. 渐进生成：产生一些着法，希望沿着某个着法下去，可以证明这条路线好到(或坏到)足以中止搜索的程度(而不必再去找其他的着法)【译注：即后面要提到的“截断”】；
3. 完全生成：产生所有的着法，希望置换表(在第二部分曾讨论过)会包含有关的信息，从而没有必要对这些着法进行搜索。

选择生成(由之衍生出的搜索技术称为“朝前裁剪”(Forward Pruning))，上世纪70年代中期以前，几乎所有的程序都这么做的，但是从那以后就突然消失了。另外两个方法代表了一枚硬币的两面——着法产生和搜索之间的权衡。对于那些着法产生简单并且有很多路线可以到达相同局面的游戏(或者具备其中一个特征)，例如黑白棋(Othello)和五子棋(GoMoku)，完全生成效率会更高些，而当着法产生的规则复杂的时候，渐进生成的速度会更快。不过两种策略都是很好的策略。

【这里就黑白棋发挥几句。可能原作者认为，凡是只由黑白两子构成的游戏就一定具有这两个特征了，就像围棋和五子棋。但是我曾经做过黑白棋的程序并发现两个特点，一是着法产生并不像想象中的那么容易，它有点类似于中国象棋中的炮的着法，二是殊途同归的局面比起国际象棋来说少得多，原因就在于走一步棋最多会改变18个格子的颜色，这与原作者的观点大相径庭。】

早期的策略：朝前裁剪

在1949年(确实如此)，Claude Shannon提出了两个象棋程序的算法：

1. 着眼于对于所有的着法及其应对着法，循环下去；
2. 只检查“最好”的着法，这个着法由对局面的分析来确定，然后也只选择“最好”的应对着法，循环下去。

起初，第二种选择看上去成功的可能更大。毕竟人就是这么做的，从逻辑上说在每个结点上只考察某些着法，总比考虑所有的着法要快。不幸的是，这条理论被事实推翻了，用选择搜索的程序，棋下得并不怎么样。它们最好的也只能达到中等俱乐部棋手的水平，最坏的情况下还会犯低级错误。打败世界冠军(现实一点，水平发挥得稳定一些)是遥不可及的。

在上世纪70年代中期，著名的美国西北大学Slate和Atkin的小组决定放弃复杂的“最佳着法”生成办法，后来证明，他们绕过复杂分析所省下来的时间，足以进行全面的搜索(检查所有的着法)。无论怎么说，这项改革善意地把“朝前裁剪”埋葬了。

下面介绍一下鲍特维尼克的工作。

上世纪70年代到80年代早期，在前世界冠军鲍特维尼克(Mikhail Botvinnik)的领导下，苏联发展了一个特别的朝前裁剪的例子。鲍特维尼克认为，计算机要达到特级大师级水平，唯一的途径就是像特级大师一样，即只考虑少量着法，但是要足够深远足够细致。他的程序试图判定世界级选手才能掌握的局面，还要实现很高水平的作战计划。尽管这个过程中诞生了一些吸引人的著作，揭示了只有鲍特维尼克本人才具备的特级大师级思路，但是这项工作最终没有达到预期的目标。

产生全部着法

自从朝前裁剪被淘汰以后，最直接的实现完全搜索的方法是：

1. 找到局面中所有合理的着法；
2. 对他们进行排列，想要提高搜索速度就必须选择最佳的顺序；
3. 对他们逐一进行搜索，直到全部搜索完成或者被截断【运用Alpha-Beta等搜索方法，可以在特定的情况提前中止搜索，以后的搜索就没有必要，这种情况就称为“截断”(Cut-off)，连载的第四部分会介绍这类搜索方法】。

早期的程序(例如Sargon)每次都扫描棋盘的每个格子，找有没有可以移动的棋子，然后计算可能的着法。当时存储器是稀有矿产，额外花费CPU的时间每次把着法重新计算一遍，是别无选择的蠢事。

如今，预处理的数据结构(就像我上个月描述的那个)可以避免大量计算，而复杂的代码会多花费几十KB的空间。当这个超快的着法产生方法和置换表结合在一起的时候，程序员眼前又多了一条思路——如果某些着法已经被搜索过，它的价值(保存在置换表中)足以产生截断，那么根本就不需要搜索任何着法了。很明显，置换表越大，并且置换可能越多(它由游戏规则决定)，置换表的作用就越明显。

这个技术不仅在概念上简单，而且普遍适用于其他游戏(但着法却不是普遍适用的，象棋着法可以分为吃子和不吃子，其他游戏像黑白棋就不那么容易分类了)。因此，如果试图让你的程序不止能玩一种游戏，你应该尽可能地用这个技术来代替下面要介绍的一个技术。

一次只产生一步棋

老牌的象棋程序(至少是CHESS 4.5以前的)则采取了相反的策略——一次产生少量着法，然后搜索，如果产生截断，就不需要产生剩下的着法了。

这种方法的流行是因为下列因素结合在一起了：

1. 搜索是不需要太大的存储器的。上世纪70年代以前的程序只能处理很小的置换表，也没有预处理的数据结构，这些都限制了完全搜索方案(就是上一节提到的方案)的运用。
2. 在象棋着法产生的预处理比其他游戏复杂得多，有上王车易位、吃过路兵，不同的棋子要区别对待。
3. 通常，这个方案的说服力(就是某步棋可以产生截断的例子)在于吃子。例如，某棋手送吃他的后，对手就会毫不犹豫地吃掉它，棋局也因此结束了。因为一个局面中能吃子的着法不多，而且产

生吃子着法相对要容易一些，所以计算吃子的着法有很多产生截断的机会。

4. 吃子是静态搜索(Quiescence Search，这个将在后面几部分提到)中需要检查的着法(不是唯一一类着法【可能除了吃子以外就是将军了】)，所以单独产生吃子的着法是非常有效的。

所以，很多程序都是先产生吃子的着法的，而且从吃最有价值的子开始，来找到最快的截断。有些技术是专门提高吃子着法的产生速度的，多数运用了位棋盘的技术：

CHESS 4.5 用了两个组64个的位棋盘，棋盘上的每一格对应一个位棋盘。一组由 “这个格子上的子可以攻击的格子” 的位棋盘组成，另一组正好相反，由 “可以攻击这个格子的棋子所占的格子” 的位棋盘组成。所以，如果程序去找吃黑后的着法，它先从基本位棋盘的数据库中找到黑后的位置，然后用这两组位棋盘来确定【其实只要用后一组就可以了】攻击后的棋子的位置，并且只产生这些子的着法。

每走一步都需要维护这些位棋盘，这就需要引进很多技术，有一个称为 “旋转的位棋盘” (Rotated Bitboard)的作用最显著。对于第一组位棋盘的产生办法，我见过的最好的论述是由James F. Swafford写的，这篇文章是在网上找到的，网址是：
http://sr5.xoom.com/_XMCM/jswaff/chessprg/rotated.htm。

【可以参阅我从Allan Liu那里整理的译文《对弈程序基本技术》专题之《数据结构(二)——旋转的位棋盘》，现在Swafford教授的那个网站关了(这至少是5年以前的网站了)，但事实证明，他的那套方案并不那么有效，有误导之嫌。】

对着法排序以加快搜索速度

我们下次要提到，搜索效率取决于搜索着法的顺序。着法排列的好坏直接影响到效率 好的排列方法可以产生很多的截断，大大减小搜索树的大小减小，其结点数只有用最坏的排列的搜索树的平方根那么多。

不幸的是，可以证明，最好的顺序应该从最佳着法开始。当然，如果你知道哪个着法是最佳的，你还有必要做搜索吗？不过仍旧有一些 “猜” 的方法能确定可能比较好的着法。例如，你可以从吃子、兵的升变(它会大幅度改变子力平衡)，或者是将军着法开始(它的应对着法很少)。紧接着是前面搜索过的在搜索树的同一层【肯定是在搜索树的不同分枝上】上产生截断的着法(即所谓的 “杀手着法”)，然后是剩下的。这就是迭代加深(Iterative Deeping)的Alpha-Beta搜索(下个月会详细讨论的)和历史表(上次讨论过了)的原理。要注意的是，这些技巧区别于朝前裁减——所有的着法最终是被检测的，那些坏的着法只是排在后边，而不排除在考虑范围以外。

最后要注意的是，在象棋中，有些着法因为被将军而是非法的。但是这些状况毕竟是罕见情况，可以证明判断着法的合理性需要花费很多运算量。更有效的办法是所有的着法都搜索完了再来作检查，例如某步棋走完后有个吃王的应对着法，这时才来裁定这步棋为非法，搜索就此中止。很明显，如果搜索到这步棋以前，就产生截断了，那就不会对这步棋的合法性作出判断了【这部分的时间不就省下来了吗】。

【这里就本节提到的 “杀手” 着法作一些发挥：大多数程序往往不是生成全部着法的，而是先判断杀手着法的合理性(判断着法合理性所做花的时间要比产生全部着法少得多)，如果是合理着法就先搜索这些着法。因为杀手着法是产生截断几率很高的着法，所以很有可能不需要生成着法了。

另外，排序技术也非常有讲究，因为排序所花的时间可能会比着法产生的时间更多。排序的算法很多，常用的有冒泡排序、Shell排序、快速排序等，我个人倾向于最慢的冒泡排序，原因是冒泡排序每扫描一趟会产生一个最大值，希望它能产生截断而没有必要对后面的着法再进行排序。】

我的选择

在我的象棋程序里，我选择产生所有的着法，只是因为以下几个原因：

1. 我想让这个程序作为其他游戏的基础，这些游戏没有像象棋一样的吃子着法；
2. 我有足够的存储器来运行这个游戏；

3. 这个技术需要的代码写起来比较容易，也比较看得懂；
4. 已经有很多免费的程序，可以实现只产生吃子的着法，有兴趣的读者可以去看Crafty的源程序，还有James Swafford的Galahad。

我的程序的整个表现似乎比别人的稍差了些【我想可能就是受了James Swafford的误导吧】，我的程序(是用Java写的，没有用别的)不想和深蓝去比，所以我感觉这并不算太坏。

下一个月

现在我们准备探索象棋程序的核心部分——搜索技术了。这是一个很大的专题，需要两篇文章。我们从所有游戏最基本的搜索算法开始说起，然后才是最新发展起来的专门针对象棋的优化方法。

François Dominic Laramée，2000年7月

原文：http://www.gamedev.net/reference/programming/features/chess3/
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 国际象棋程序设计(二)：数据结构
- 下一篇 国际象棋程序设计(四)：基本搜索方法
- 返　回 象棋百科全书——电脑象棋

国际象棋程序设计(四)：基本搜索方法

François Dominic Laramée/文

这个烦琐、没有代码、像胶水一样的连载，已经进入第四部分了，你们还在看下去吗？如果是的，就写eMail给我，我就可以找到写这些东西的理由了。

无论如何，这个月的主题是策略类游戏的敌对搜索(Two-Agent Search)【译注：意思是站在两个立场上的搜索，我也不知道该用什么比较确切的术语】，要了解它们有什么作用，如何来做，对电脑的程序意味着什么。我们要讨论的技术在所有的游戏中都很重要，但是仅仅靠它们是不足以下象棋的。下个月，我会介绍一些高级的技术，它们能显著增强棋力，同时提高搜索效率。

为什么要搜索?

简单地说，因为我们还没有聪明到可以抛弃搜索的地步。

一个真正聪明的程序可以只依靠棋盘局面来确定，哪一方处于领先以及领先多少，并且制定作战计划让这个优势变为胜果。不幸的是，需要考虑的局面类型太多了，再加上那么多的规则和特例，就是再聪明的程序也不擅长做此类事情。而它们擅长的则是计算。因此对象棋程序来说，与其只根据棋盘的局面来决定好的着法，不如使用它们的蛮力——考虑每一种着法，然后为对手考虑这些着法的所有应对着法，循环下去，直到处理器吃不消为止。

比起掌握复杂的策略，深入的搜索是教电脑下棋的比较简单的方法。例如，考虑马的捉双问题，走一步棋就可以同时攻击两种棋子(例如一个车和一个后)。掌握这种类型的走法是需要花一些时间的，更复杂些，我们还要判断这个格子有没有保护。然而，一个愚钝的3步的搜索就可以学会捉双了，程序最终会尝试让马走到捉双的格子，并探索对手对于捉双的回应，然后吃掉没有逃跑的那个棋子，从而改变了子力平衡。由于全面搜索可以看到每一步，所以不会错过任何机会。如果经过5步棋后可以产生杀棋或捉死后(无论怎样不容易看到)，只要电脑的搜索得足够深，就会看到这个着法。因此，搜索得越深，电脑就会施展越复杂的作战计划。

爷爷时代的“最小-最大”原理

所有双向搜索算法的最基本的思想都是“最小-最大”(Minimax)原理。它可以追溯到中世纪(Dark Ages)，但我相信它最早是由冯-诺依曼(Von Neumann)【应该是John von Nuoma，1903-1957，美籍匈牙利数学家】在60年前完整描述的：

1. 假设有对局面评分的方法，来预测棋手甲(我们称为最大者)会赢，或者对手(最小者)会赢，或者是和棋。评分用数字表示，正数代表最大者领先，负数代表最小者领先，零代表谁也不占便宜；
2. 最大者的任务是增加棋盘局面的评分(即尽量让评分最大)。
3. 最小者的任务是减少棋盘局面的评分(即尽量让评分最小)。
4. 假设谁也不会犯错误，即他们都走能让使局面对自己最有利的着法。

那么这该怎么做呢？设想一个简单的游戏，每方只走一步，每步只有两种着法可供选择。评分函数只和最后棋盘的局面有关，即评分是最大者和最小者着法综合的结果。

最大者的着法	最小者的着法	评分
A	C	12
A	D	-2

B	C	5
B	D	6

最大者假设最小者会下出最好的棋，因此他知道，如果他选择着法A，那么他的对手会回应D，使最终评分变成-2(即获胜)。但是，如果最大者走的着法B，那么他就会获胜，因为最小者的最佳着法仍然是正数(5)。所以按照“最小-最大”算法，最大者会选择着法B，即使他选择A并且最小者走错时评分还会更高。

“最小-最大”原理有个弱点，从简单的例子中还不能明显地看出来——要检查的各种路线的数量是指数形式的，这就意味者工作量会以几何级数增长。

1. 每方有多少种可能的着法，这个数称为分枝因子(Branch Factor)，用b表示；
2. 考虑的深度用n表示，通常说“n层”，n是整数，“层”(Ply)表示一个棋手走的一步棋。例如在上面介绍的迷拟游戏中，搜索深度是2层。每个棋手走1步。

例如在象棋中，通常在中局阶段分枝因子大约为35种着法，在黑白棋中大约为8。由于“最小-最大”算法的复杂度是$O(b^n)$，所以对一个局面搜索4层就需要检查150万条路线，这是多大的工作量！再增加上去，5层就会把搜索树膨胀到5000万，6层则达到18亿！【原作者这里写的是8层150万、9层5000万、10层18亿，不知为何多算了4层。】

幸运的是，有办法能在精确度不打折扣的情况下大幅度削减工作量。

Alpha-Beta搜索——让“最小-最大”法成为现实(也只是有一点点现实)

设想你在迷拟游戏中已经搜索了着法B，结果你知道最大者在整个游戏中最高得分是5。

现在假设你开始搜索着法A了，并且一开始寻找的路线是A-D，这条线路的得分是-2。对于最大者来说，这是非常糟糕的，如果他走了A，那么结果肯定会是-2，因为最小者总是走得最好的。这是因为，如果A-C比A-D更好，那么最小者会选择A-D，如果A-C更坏(比如说-20)，那么最小者就会选择这条路线。所以，没有必要再去看A-C以及其他由A产生的路线了——最大者必须走B，因为到此位置的搜索已经能证明，无论如何A是个更糟的选择。

这就是Alpha-Beta算法的基本思想——只要你有一步好的着法，你就能淘汰其他可能导致灾难的变化，而这样的变化是很多的。如果再跟前面介绍的置换表结合起来，当不同路线的局面发生重复时可以节省下分析局面的时间，那么Alpha-Beta就能产生无限的能量——在最好的情况下，它处理的结点数是纯粹的“最小-最大”搜索的平方根的两倍，从1500万可以减少到2500。

【要说明Alpha-Beta搜索的结点数是死办法(即不用Alpha-Beta搜索的办法)的平方根的两倍那么多，可以分别计算搜索树中两种类型的结点——Alpha结点和Beta结点。

Alpha-Beta搜索是完全搜索，如果某个结点是完全搜索的，那么这个结点称为Alpha结点，显然根结点是Alpha结点。那么Alpha结点的分枝又是什么呢？至少有一个Alpha结点，这是肯定的，最好的情况下，剩余的结点都可以产生截断，这些结点称为Beta结点。Beta结点有个特点，只要它的分枝中有一个Alpha结点产生作用，那么剩下的结点就没有搜索的必要了，我们还是取最好的情况，分枝中只有一个Alpha结点。

那么如何计算Alpha结点的个数呢？一个Alpha结点下面有$b-1$个Beta结点，每个Beta结点下面又有1个Alpha结点，这样深度每增加了两层结点数才扩大b倍，因此总的Alpha结点数就是$b^{n/2}$。同样道理，Beta结点也这么计算，得到的结果也是$b^{n/2}$，因此总结点数就是$2b^{n/2}$。】

对着法排序来优化Alpha-Beta搜索

可是，我们如何才能达到预期的效果呢？我们是否还需要做其他事情？

的确是的。只要Alpha-Beta搜索可以找到比其他着法好的着法，它就能对搜索树作出非常有效的裁减，这就意味着，关键在于首先搜索好的着法。当我们在搜索其他着法以前先搜索到最好的着法，那么最好的情况就发生了。然而最坏的情况，搜索着法的顺序是按评分递增的，即每次搜索到的着法都比曾经搜索的着法要好，那么这种情况下的Alpha-Beta搜索就无法作出任何裁减，这种搜索将退化为极其浪费的“最小-最大”搜索。【这就是前一章的标题中写道“也只是有一点点现实”的原因。】

对搜索进行排序是相当重要的。让着法随便排列肯定不行，我们必须找到更聪明的办法。不幸的是，如果有简单的办法知道最好的着法，那还有搜索的必要吗？因此我们必须用“猜最好的着法”来对付。

有很多技术可以让着法的顺序排列成尽可能好的顺序：

1. 用评估函数对着法打分，然后排序。直觉上这会起到作用，评估函数越好，这个方法就越有效。不幸的是在象棋中它一点也不起作用，因为下个月我们将了解到，很多局面是不能准确评估的。

2. 找到在置换表中已经存在的局面，如果它的数值足够好，就会产生截断，这样就不必再进行其他搜索了。

3. 尝试特定类型的着法。例如，后被吃掉肯定是最坏的想法，所以先检查吃子的着法是行之有效的。

4. 把这个思路进行拓展，关注已经在同一层深度产生过截断的着法。“杀手启发”(Killer Heuristic)是建立在很多着法是次序无关的基础上的。如果你的后被攻击了，不管你把H2兵挺一格还是两格，对手都会吃掉你的后。因此，如果在搜索H2-H3着法时，“象吃掉后”的着法会产生截断，那么在搜索H2-H4着法时，“象吃掉后”很有可能也产生截断，当然应该首先考虑“象吃掉后”这一步。

5. 再把杀手启发拓展到历史表上。如果在前面几回合的搜索中，发现G2-E4的着法很有效，那么很有可能现在仍然很有用(即便原来的格子是象而现在变成了后)，因为棋盘其他位置的情况不太可能有多少变化。历史启发的的实现非常简单，只需要一个64x64的整数数组，它可以起到很可观的效果。

现在已经谈了所有宝贵的思想，然而最有效的方法却稍稍有背于人的直觉，这个方法称为“迭代加深”(Iterative Deepening)。

迭代加深的Alpha-Beta搜索

如果你正在搜索6层的局面，那么理想的着法顺序应该由同样层数搜索的结果来产生。既然这明显是不可能做到的，那么能否用稍浅的搜索(比如说5层)来代替呢？

这就是迭代加深方法背后的思想——一开始搜索2层，用这样种搜索产生的着法顺序来搜索3层，反复如此，直到到达规定的深度。

这个技术会受到一般人的质疑——大量的努力是重复的(8到10次乃至更多)，不是吗？

那么考虑一下搜索树的深度n和分枝因子b好了。搜索树在第1层的结点数为b，第2层是b^2，第三层是b^3，等等。如果b很大(记住中局时在35左右)，那么绝大多数工作量取决于最后一层。重复搜索$(n - 1)$的深度其实是小事一桩，即便迭代加深一点也不起作用，它也只会多花4%的时间(在通常的局面上)。

但是，它的优势是非常可观的——由浅一层搜索的结果排列得到的顺序，在层数加深时可以大幅度增加截断率。因此，迭代加深的Alpha-Beta搜索实际上找的结点数，会比直接的Alpha-Beta搜索的少很多。在使用置换表后，它的收效就更可观了——重复搜索浅的部分就几乎不需要时间了，因为这些结果在置换表里都有，没有必要重新计算了。

【需要指出的是，实际运用中通常只对根结点进行迭代加深，这样搜索的结点数是$1 + b + ... + b^n$，比b^n大不了多少。如果每个结点都用迭代加深，则需要搜索的结点数就是$(1 + b)^n$，刚才提到在最好的情况下分枝因子为合理着法数的平方根，即b在6左右，而6^n和7^n是有很大区别的。

事实上，要在每个结点上都使用迭代加深，只要检查置换表就可以了，因为从根结点处做深一层的搜索时，除了新增加的一层结点以外，其他各层结点上都有最好的着法存储在置换表中了，这些着法应该优先考虑，绝大多数着法能产生裁剪，而不需要检查杀手表或者做产生产生。

另外，迭代加深可以大幅度提高历史表的效率，在前一次N层的搜索中历史表已经有相当丰富的历史着法信息了，在做$N+1$层搜索时，这些历史信息可以让着法有更好的顺序。】

电脑下棋的风格

迭代加深的Alpha-Beta搜索和置换表(并且在历史表的推动下)能让计算机对局面搜索到相当的深度，并且可以下国际象棋了。应该这么说，它的祖先“最小-最大”算法决定了那台电脑(曾经在世界上有惊人之举的那台电脑【即冯-诺依曼的ENIAC】)下棋的风格。

例如，设想电脑能对一个局面搜索8层，通常在考虑某一步时，这种让人不解的模式会让让它战胜对手。只有少数一些看不清的、复杂的、迷惑人、超出直觉局面，才能让对手抓住一线机会取得胜利，而对于人类棋手(甚至大师)来说，这样的局面陷阱已经可以写到书里去了。

然而，如果电脑找到了一个导致和棋的无聊着法，它就会绕过陷阱，因为它假设对手会作出正确的应对，无论那种可能性有多小，只要电脑认为平局是最高的期望。

结果你可能会说，电脑下棋会极端谨慎，就像对手都是世界冠军一样。如果考虑到电脑算不到出人类棋手布置的陷阱那个深度，人类就会绞尽脑汁布设陷阱，让电脑犯错误。(人类棋手会研究对手的下棋风格，如果卡斯帕罗夫有机会和深蓝下一百盘棋，他可能会找到深蓝的弱点并且打败它。但是我们不知道有没有这种可能性。【现在看来，卡斯帕罗夫战胜深蓝是不在话下的，因为近几年他一直在和水平更高的电脑较量。】)

下一个月

在第五部分，我们会讨论直接或改变深度的Alpha-Beta搜索的局限。我们还会讨论如何利用一些技术，诸如空着启发(Null-Move Heuristic)、静态搜索(Quiescence Search)、期望搜索(Aspiration Search)、MTD(*f*)搜索，以及让深蓝名声显赫的“单步延伸”(Singular Extensions)技术，它们会提高下棋水平。坚持住，我们快要结束了！

François Dominic Laramée，2000年8月

原文：http://www.gamedev.net/reference/programming/features/chess4/
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 国际象棋程序设计(三)：着法的产生
- 下一篇 国际象棋程序设计(五)：高级搜索方法
- 返 回 象棋百科全书——电脑象棋

国际象棋程序设计(五)：高级搜索方法

François Dominic Laramée/文

哇，看上去好像有很多人在看我的连载，我的脸都红了！

这是倒数第二篇文章，我们会介绍和搜索有关的高级技术，他们既能提高速度，又能增强棋力(或者只有一个作用)。他们中有很多，概念上(程序代码可能不行)可以运用到任何2人游戏中，然而让它们用到一些具体问题上，对很多读者来说还需要加工。

干吗要那么麻烦?

到此为止，我们知道的所有搜索算法，都把局面推演到固定的深度。但是这未必是件好事。例如，假设你的程序最多可以用迭代加深的Alpha-Beta算法搜索到5层，那么来看下这几个例子：

1. 沿着某条路线，你发现在第3层有将死或逼和的局面。显然你不想再搜索下去了，因为游戏的最终目的达到了。搜索5层不仅是浪费时间，而且可能会让电脑自己把自己引入不合理的状态。

2. 现在假设在5层你吃到了兵。程序可能会认为这个局面稍稍有利，并且会这么走下去。然而，如果你看得更深远些，可能会发现吃了兵以后你的后就处于被攻击的状态，完了！

3. 最后，假设你的后被捉了，不管你怎么走，她会在第4层被对手吃掉，但是有一条路线可以让她坚持到第6层。如果你的搜索深度是5，那么后在第4层被吃掉的路线会被检测出来，这些情况会评估成灾难局面，但是唯一能使后在第6层(超出了搜索树)捉到的那条路线，对于电脑来说被吃是不能被发现的，因此它会认为后很安全，从而打一个较高的分数。现在，为了让后被吃的情况排除在搜索树以外，唯一的办法就是调虎离山，这样做是很冒险的——牺牲一些局面，还是有可能让对手犯下错误让你的后溜掉的。那么如果你要通过牺牲一个车来延缓后的被吃呢？对电脑来说，在第4层丢车要比丢后损失小，所以在这个搜索水平上，它情愿丢一个那么大的子，来推迟那个可怜的后的被吃了。(当然在随后一回合里，电脑会发现无论怎么走，它的后会在第4层被吃掉，并且车丢得莫名其妙。)很早以前Hans Berliner就把这个情况称为“水平线效应”，这在很大程度上可以通过后面即将介绍的“静态搜索”来改善。

最后要说一句：象棋中的很多局面(其他游戏也一样)太不可预测了，实在很难恰当地评估。评价函数只能在“安静”的局面下起作用，即这些局面在不久的将来不可能发生很大的变化。这将是我们介绍下一个内容。

请安静!

有两种评估局面的办法——动态评估(看局面会如何发展)和静态评估(看它像什么样子，不去管将来怎样)。动态评估需要深入的搜索。我们刚刚提到，局面在不久的将来不可能发生很大的变化的情况下，静态评估才是可行的。这些相对稳定的局面称为“安静”(Quiet)或“寂静”(Quiescent)的局面，它们需要通过“静态搜索”(Quiescence Search)来达到。

静态搜索的最基本的概念是指：当程序搜索到固定深度时(比如6层)，我们选择性地继续各条路线，只搜索“非静态”的着法，直到找到静态着法为止，这样才开始评估。

找到静态局面是需要游戏知识的。例如，什么样的着法可能引起棋盘上子力平衡的巨大改变？对于象棋来说，子力平衡会导致局面的剧烈变化，所以任何改变子力的着法就是——吃(特别是吃主要棋子)、兵的升变都是，而将军也是值得一看的(仅仅是能导致将死的将军)。【译注：我认为任何将军都应该考虑进去，因为它会导致抽吃、长将等决定性局面的产生】。在西洋棋里，吃子和升变【西洋棋的棋子分兵棋(Man)和王棋(King)，兵棋冲到底线就变成王棋，因此我断定它是国际象棋的

前身】都是选择。在黑白棋中，每一步都必须吃子，并且“子力平衡”【仅仅指目前棋子的多少，它和最终棋子的多少没多大联系】在短时间内翻覆无常，所以可以说它根本不存在“静态局面”！

我自己的程序用了简单的静态搜索，它只考虑所有带吃子着法的线路(在*x*层完全搜索以后)。由于通常局面下没有太多合理的吃子着法，所以静态搜索的分枝因子非常小(平均在4-6，双方在吃子后会迅速下降到0)。但是，静态搜索算法要分析大量的局面，它可能会占用整个处理器一半以上的时间。当你的程序使用这个方案以前，你要确定你是否需要用它。

当没有吃子发生时，我的程序才开始评价局面。其结果就是将层数固定的搜索树作选择性的延伸，它能克服大多数由“水平线效应”产生的后果。

重要的空着

有个加快象棋程序速度的有效方法，就是引入空着的概念。

简而言之，空着就是自己不走而让对手连走两次。大多数局面中，什么事都不做显然不是办法，你总是必须做点事情来改善局面。(老实说，有一些“走也不是，不走也不是”的局面，空着确实是你的最佳选择，但不能走，这种 “被迫移动”(Zugzwang)局面是没有指望的，所以不必对电脑感到失望。)

在搜索中让电脑走空着，可以提高速度和准确性。例如：

1. 假设局面对你来说是压倒性优势，即便你什么都不走，对手也无法挽回。(用程序的术语来说，你不走棋也可以产生Beta截断。)假设这个局面本来准备搜索*N*层，而空着取代了整个搜索树(你的所有合理着法用空着取代了)，并且你的分枝因子是*B*，那么搜索空着就相当于只搜索了一个*N*-1层的分枝，而不是*B*个这样的分枝。在中局阶段通常*B*=35，所以空着搜索只消耗了完整搜索所需的3%的资源。如果空着搜索表明你已经强大到没有必要再走棋(即会产生截断)的地步，那么你少花了97%的力气。如果没有，你就必须检查合理的着法，这只是多花了3%的力气。平均来说，收益是巨大的。【当然，空着搜索对于处理“被迫移动”局面还是有负面作用的，特别是在残局中，这个作用相当明显。可以参考《对弈程序基本技术》专题之《高级搜索方法——空着裁剪》一文。】
2. 假设在静态搜索中，你面对一个只有车吃兵一种吃子着法的局面，然而接下来对手就会走马吃车。你最好不去吃子而走其他不吃子的着法对吗？你可以在静态搜索中插入空着来模拟这种情况，如果在某个局面下空着比其他吃子着法有利，那么你继续吃子就是坏的选择。并且由于最佳着法是静态着法，所以这个局面就是评估函数可以作用的局面。

总的来说，空着启发会减少20%到75%的搜索时间。这当然值得，特别是当你把这个方法用在静态搜索算法上的时候，就像改变“走子的一方”这种代码一样简单，用不了十行就行了。

【很多书上把“空着”这一技术称为“空着启发”(Null-Move Heuristic)，本文就是这个意思，事实上在历史表、迭代加深等启发的作用下，空着启发已经意义不大了。现在绝大多数程序都使用了称为“空着的向前裁剪”(Null-Move Forward Pruning)的搜索(它跟空着启发是有区别的)，尽管是一种不完全搜索，但它却是诸多向前裁剪的搜索中最有效的一个。】

期望搜索和MTD(*f*)

普通的老式Alpha-Beta搜索对某个局面最终的“最小-最大”值没有假设。看上去它考虑到任何情况，无论有多反常。但是，如果你有一个非常好的主意(例如由于你在做迭代加深，从而想到前一次的结果)，你就会找出那些和你预期的差得远的路线，预先把它们截断。

例如，假设一个局面的值接近于0，因为非常均衡。现在来假设对一个内部结点作先前的评价，它的值在+20,000【这里的单位应该是“千分兵值”， 即1000相当于一个兵的价值，那么马和象等于3000，车5000，后9000，其他因素也折算成这个值，而UCI协议中则用“百分兵值”，因为没有必要过于精确】，那么你可以有充分信心对它截断。

这就是“期望搜索”(Aspiration Search)背后的思想，一个Alpha-Beta搜索的变种，开始时用从负无穷大到正无穷大来限定搜索范围，然后在期望值附近设置小的窗口。如果实际数值恰好落在窗口以内，那么你赢了，你会准确无误地找到路线，并且比其他的路线快(因为很多路线都被截断了)。如果没有，那么算法就失败了，但是这个错误是很容易被检测的(因为“最小-最大”值就是其中一条边界)，你必须浪费一点时间，用一个更大的窗口重新搜索。如果前面的情况比后面的情况多，那么总体上你还是赢了。很明显，你预先猜的数值越好，这个技术的收效就越大。

在上世纪90年代中期，研究员Aske Plaat把期望搜索拓展为一个逻辑问题：如果你把带期望的Alpha-Beta搜索的窗口大小设定成0，将会发生什么事？它当然永远不会成功。但是如果它成功了，那速度将是惊人的，因为它把几乎所有的路线全都截断了。现在，如果失败意味着实际数值低于你的估计，那么你用稍低点的宽度为零的窗口再试一次，重复下去。这样，你就等于用Alpha-Beta搜索来做某个“最小-最大”值的拆半查找(Binary Search)，直到你最终找到那个宽度为零的窗口。

这个伟大的设想发表在一个网站上：http://theory.lcs.mit.edu/~plaat/mtdf.html，它的具体实现称为MTD(*f*)搜索算法，只有十多行。加上Alpha-Beta搜索和置换表的运用，MTD(*f*)呈现出惊人的效率，还善于做并行计算。它在“粗糙”(简单且快速)的局面分析中运行得更好，很明显，如果局面评估的最小单位越大(例如从0.001个兵增加到0.1个兵)，它搜索的步数就越少。

在Alpha-Beta搜索的变种当中，还有很多具有广泛用途的算法(例如名声狼藉的NegaScout，我宁可给白痴讲广义相对论，也不想给你们讲这些)【之所以说NegaScout名声狼藉，是因为它的发明者Reinefeld首次发表该算法时，程序中有一个致命错误，导致搜索效率大幅度降低，甚至低于普通的Alpha-Beta搜索，如今这个算法更多地被PVS(主要变例搜索)取代，因为它更容易理解】，但是Plaat坚持认为MTD(*f*)是至今为止效率最高的算法。我就信了他的话，所以我的程序里用了MTD(*f*)，你们可能会感叹这个算法是多么简短啊！

【MTD(*f*)在整个过程中只使用极小窗口，并且每次都从根结点开始的，这个过程极大程度地依赖于置换表，称为“用存储器增强的试探驱动器”(Memory-enhanced Test Driver，简称MTD)，它只需要传递两个参数(深度*n*和试探值*f*)，故得名为MTD(*n*,*f*)，缩写为MTD(*f*)。实际运作中MTD(*f*)是以迭代的形式收敛的，而不是原作者所说的拆半查找。

在Plaat的文章中，MTD(*f*)的代码有10行，而跟它异曲同工的算法PVS，则只比普通的Alpha-Beta多了5行左右，因此很奇怪原作者(Laramée)为什么如此看好MTD(*f*)。MTD(*f*)在并行计算上确实比PVS有优势，由于Plaat等人拿MTD(*f*)和PVS算法的比较是在并行机上完成的，才得出MTD(*f*)优于PVS的结论，而事实上大部分的程序用的都是PVS。】

单步延伸

在我们结束这个主题以前，这是最后一个话题。在象棋中，有些着法明显比其他的好，这样就可能没必要搜索其他的变化了。

例如，假设你在迭代加深过程中正在做深度为$N-1$的搜索，发现某步的评分为+9000(即你吃了对方的后)，而其他都低于0。如果像比赛一样想节约时间，你会跳过前面的N层搜索而对这步进行N层搜索【对于这步来说，搜索加深了一层，对于优势局面来说，优势应该是越来越大的，所以加深一层后评分应通常要高】，如果这步额外搜索的评分不比预期的低，那么你可以假设这步棋会比其他着法都好，这样你就可以提前结束搜索了。(记住，如果平均每层有35种合理着法，那么你就可能节省97%的时间！)

深蓝的小组发展了这个思想并提出了“单步延伸”(Singular Extension)的概念。如果在搜索中某步看上去比其他变化好很多，它就会加深这步搜索以确认里边没有陷阱。(实际过程远比这里说的要复杂，当然基本思想没变。)单步延伸是耗费时间的，对一个结点增加一层搜索会使搜索树的大小翻一番，评估局面的计算量同时也翻一番。换句话说，只有深蓝那种硬件水平才吃得消它，我那笨拙的Java代码肯定不行。但是它的成效是不可否认的，不是吗？【原作者的意思可能是指，单步延伸技术会明显提高棋力，同时也会增加搜索时间。】

下一个月

在第六部分中，我们会着重讨论局面评估函数，它才真正告诉程序一个局面是好是坏。这个主题具有极其广泛的内容，可以花几年时间来改进评估方法(也确实有人这样做)，因此我们必须对这些内容进行彻底讨论，包括它们的可行性和重要程度。【在这篇普及型的连载中，作者怎么可能给你们讲那么多呢？】如果任何事情都按照计划进行，我就该用一些Java代码来给你们填饱肚子，但是这很难办到，不是吗？

François Dominic Laramée，2000年9月

原文：http://www.gamedev.net/reference/programming/features/chess5/
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇　国际象棋程序设计(四)：基本搜索方法
- 下一篇　国际象棋程序设计(六)：局面评估函数
- 返　回　象棋百科全书——电脑象棋

www.xqbase.com

国际象棋程序设计(六)：局面评估函数

François Dominic Laramée/文

已经六个月了，我知道你们有人觉得我不会闭嘴，但是你们错了，这是我的国际象棋程序设计连载的最后一篇。还有，我的Java象棋程序(看上去很烂)已经做好了，和这篇文章一起上传给了GameDev网站，这个程序还是可以验证这些连载内容的。

这个月的话题——局面评估函数——确实很特别，搜索技术大同小异，着法产生可以只根据规则来处理，而局面的评估则需要很深入彻底的分析。可以想象，如果我们不知道某个局面会导致哪方获胜，那么评价这个局面的相对强弱将是不可能的。因此，很多要讨论概念对于其他游戏来说就不是这么回事了，有的则根本不能用。对于你自己的游戏，考虑用什么样的评估方法，这是你作为程序员应该知道的。【译注：对于中国象棋要特别注意，很多概念不能从国际象棋照般，我写过《中国象棋和国际象棋比较研究》一文，详细阐述过这个问题。】

抓紧时间，让我们来看一些局面评价的尺度，并且需要了解它们有什么作用。

子力平衡

简单地说，子力平衡(Material Balance)就是指双方各有哪些棋子。根据象棋文献，后的价值是900分，车500，象325，马300，兵100，王是无价的。计算子力平衡是非常简单的，每方的子力价值就是：

$$MB = Sum\ (Np \times Vp)$$

这里Np是棋盘上这种类型的子的数目，Vp是子的价值。如果你在棋盘上的子力价值比对手多，那么你的形势好。【说句提外话，外国人把“吃”说成“抓获”(Capture)，意思是棋子并非消失了，而是暂时退出战场了，所以说子力价值时一定要强调“棋盘上”。】

看上去非常简单是不是？但是这和象棋的局面评价还差很大距离呢。CHESS 4.5的作者估计了很多位置上的因素，例如机动性和安全性差不多接近1.5个兵。实际上，用不着考虑别的，这就足够可以下出好棋了。【注意，在把“兵值”这个概念运用到中国象棋的时候，最好也把车设成500，这时机动性和安全性的因素就可以参考国际象棋，也不超过150分。具体的子力价值可参考《中国象棋和国际象棋比较研究(二)——子力价值》一文。】

的确，有些情况下为了通过子力交换让局面得到进展，你必须牺牲某些棋子(甚至是后)。然而最好是通过搜索来发现，例如弃后可以导致3步杀，你的搜索函数不需要额外的代码就会找到杀棋(前提是它看得足够深远)。如果你硬是要在评价函数里写特别的代码，来确定是否应该弃子，这种事情将是噩梦。

很少有程序会用像我开始所说的那个评价函数。由于计算过于简单，人们会在上面增加点东西，都是这么做的。举一个众所周知的例子，当你子力上处于领先时，交换相等的子力是有利的。交换一个兵是不错的选择，因为它为你的车开放棋盘，但是你必须在棋盘上保存一些兵直到残局，以构筑防御屏障或者升变成后。最后，如果开局库里走出弃子的开局，你不用为程序担心，因此你要把“藐视因子”(Contempt Factor)加到子力平衡的评价中。例如当子力落后150分甚至更多时，你仍然让程序觉得是领先的。

要注意，子力平衡在象棋和西洋棋里具有很高的地位，但是在黑白棋里却是欺骗性的。的确，棋局最后你必须控制更多的格子才能赢，但是你最好在中局阶段控制子的个数。在其他游戏像五子棋【Go-Moku确实应该是五子棋，我想不出还有什么不吃子的黑白棋类了】及其变种*N*子棋(Connect-*N*)【具体怎么下我也不知道】，子力平衡是不会变化的，因为始终没有被吃的子。

机动性和对棋盘的控制

将死的特征之一是没有合理的着法。直觉上，好像选择余地越大约好，比如选手在30种着法内找出一步好棋，这种可能总要比限定在3步内高。

在象棋中，机动性是很好评估的 计算同一局面下每方的合理着法，你已经能做到了【还要复习一下《着法的产生》吗？】。然而这个数字仅仅占一小部分。为什么呢？就因为很多着法是漫无目的的。例如你每次让车向后退一格，着法是合理的，但不产生作用。另外，不惜代价试图限制对手的机动性，会让程序进行一系列漫无目的的将军，同时摧毁自己的防线。由于解除将军通常只有3到4种合理着法，所以基于机动性的程序会走出劣着去将军对方，过会儿就会发现什么事也没干成，并且自己的兵力也成一盘散沙了。【我认为这种结果的根源在于没有把将军着法考虑进“静态搜索”中，如果考虑进去了，那么将军的时候不做局面评价，根本不存在对手机动性很低的假象。】

但是，有些明智的机动性评估方法，会起点作用。我的程序就“坏象”给予罚分，即象的行动路线被一系列同色格的兵阻挡的情况，当马太靠近棋盘边缘时也同样处理。另一个例子是车，在开放或半开放线(即没有兵或只有一个兵的纵线)上价值更高。

和机动性有密切联系的是棋盘的控制能力。在象棋中，如果一方对某个格子产生攻击的棋子数量超过对方，那么这一方就控制了这个格子。走到受控制的格子通常是安全的，走到被对方控制的格子则是危险的。(有一些例外，把后走到被敌方兵攻击的格子里，不管你有多少子可以吃他的兵，这无论如何不是好主意。还有，你故意把子送到对方嘴里，但是对方有更重要的格子需要保护。)在象棋中，控制中心是开局的基本目标。但是，控制能力在某种程度上是很难计算的，它需要在任何时候都更新数据库，来记录棋盘上所有被攻击的格子。很多程序都这么做，但我的没有。

机动性对于象棋程序来说并不是非常重要的，但在黑白棋里却非常重要(在残局中有很少子可走的一方会陷入很深的困境)。而对围棋而言，对棋盘的控制则是胜利的标致。

局势发展

有个教条，下象棋时轻子(象和马)要尽早投入战斗，王要尽早易位，车和后尽量少动，直到关键时刻作出决定性的攻击。这是有很多原因的，马和象(还有兵)能控制中心，来支持后的火力，把它们调动出去能为底线疏通车的路线。随着棋局的进行，车就可以跑到第七行(即开始对对方的兵动手)，形成巨大的破坏力。

我的程序用了一些因子来衡量局势的发展。首先，任何王和后前没有动过的兵都要扣分，阻挡车的马和象也要扣分，这样可以让后在其他子力出动后才开始动，如果对手还有后，那么王易位到安全位置时则给予很大的加分(有易位权但是没有易位的也给予少量加分)。

你可以看到，局势发展的因素在开局阶段很重要，但是马上将失去意义。在大约10回合以后，要衡量的因素基本上都发生过了【即能加分的都加了，要扣分的都扣了】。

注意，在西洋棋之类的游戏中，倾向于局势发展可能很不好。事实上，先走的一方要空出后面一行的空位，这就已经吃亏了，而在这些地方避免局势发展，通常是不错的选择。

兵形

象棋大师门常说，兵是象棋的灵魂。因此新手很难体会的是，强手在对局时会因为一个兵的损失而早早认输。

象棋书籍中提到很多兵的类型，有些是有利的，有些是有害的。我的程序考虑了以下几点。

1. 叠兵或三叠兵，一方两个或多个兵在同一列上是很坏的，因为它们的移动相互阻碍了；
2. 对兵，双方两个兵“头碰头”互相阻挡去路，会造成很大的障碍；
3. 通路兵，当兵进展到不会有对方兵攻击或阻碍时【即同一列和相邻列里都不能有对方的兵】，就会变得非常强大，因为它们更容易到达底线实现升变；
4. 孤兵，两边都没有同伴兵保护的兵，最容易受到攻击，最需要寻求保护；
5. 满兵，棋盘上有太多的兵会影响机动性，为车开放一条纵线才是好主意。

最后兵形上要注意的是：后面跟了车的通路兵是非常危险的，因为任何吃掉它的子都会成为车的盘中餐。如果有一个车在通路兵的后面(同一列)，那么我的程序给予极高的加分。

王的安全和取向

我们很早就谈到过王的安全了，在开局和中局里，保护王是最重要的，易位是最好的办法。

但是在残局里，王身边的很多子都没了，他转眼间变成了攻击性子力！把它留在兵阵的后面简直是浪费资源。

“取向”(Tropism)在这里衡量的是一个子攻击对方王的难易程度，通常用距离来衡量。取向的计算规则跟兵种有关，不过通用的规则是，你越靠近对方的王，你对他的压力就越大。

确定权重

现在我们确定了有那些因素是需要衡量的，但是我们怎么决定它们的权重呢？

这是个非常有价值的问题，可以花几年时间(也确实有人在做)用线性组合的办法来调整评估函数，有时把机动性设得多一些，有时更强调安全性，等等。我当然希望有一个绝对的答案，可惜没有，因为再好的评价函数都会碰到麻烦和失误。如果你的程序足够强，那很好。如果不是，那么试试别的方案，让它和你原来的程序下，如果能赢下大多数，那么新的方案就是一个进步。

有三种方法可以试试：

1. 通过优化评估函数取得的成绩，要达到加深一层搜索所达到的同样效果，这是非常困难的。如果你很疑惑，那么宁可把评估函数设得简单些，把更多的处理器资源留给Alpha-Beta搜索。【有关资料表明，加深一层搜索大约可以使棋力提高200分(ELO)，这是相当可观的。】

2. 除非你想和世界冠军去比，否则你的局面评价函数不必要特别有效。

3. 如果你的程序确实很快，你可以花些时间用适当的算法来改进评估方法。但是对象棋来说，花上几千年时间都是有可能的。【这里指用适当的算法来调整权重，例如线性回归分析、神经网络算法等，每做一步调整，都需要用大量的对局来试验，所以工作量非常大。】

【最后想注明我的观点，即局面评估函数是整个象棋程序的核心。人们越来越把注意力集中到局面评估函数上，好的象棋程序往往用上万行的程序来分析局面，但它们的作者很少会透露出其中的细节。一个最简单的办法就是去看一些公开了的象棋程序，例如：Crafty的评估函数有4000多行，主要包括以下四方面的内容：

(1) 子力价值，仅仅是简单地把每个子的价值加在一起；

(2) 兵的价值，不仅考虑兵的相对位置，也考虑通路兵(特别在残局中)；

(3) 跟每个子位置有关的子力价值，即子力的灵活性、协调性等因素；

(4) 王的安全性，既考虑王周围的兵阵，也考虑可以攻击王的其他棋子。

以上译自Crafty源程序的第一段，其中的细节可以参考这个程序的其他片段。】

下一个月

好了，没有下一个月了，就这些了。

如果我要把这个连载拖得长一些，我可以写一些关于开局库(Opening Book)、残局库(Endgame Library)、特别针对象棋的硬件，以及很多很多其他的内容。我当然写得出来，但是我才不写呢。我打算把一部分内容保留到我今年秋天要写的书里，其他内容我也不知道到底有没有用。还有一个原因，我也懒得写下去了。

我希望你们喜欢看我的东西，希望你们能学到两三个有用的技巧。如果真的学到了，那么明年看我写的别的东西吧(GDC或E3)【我也不知道是什么学科，GDC好像是图形显示控制芯片，反正和象棋没任何关系】，并且在你公司里的工程师或程序员面前好好夸我，好吗？

太高兴了！

François Dominic Laramée，2000年10月

【译后注：总的来说Laramée写的东西还是比较浅显的，没有代码和过于专业的知识(即便是有，我也已经在译注中说明了)。但是或许是我英语水平的问题，他的某些表述并不能理解，因此即便很别扭地翻译出来了，还是要在译注中加上自己的理解，这让我非常吃力。不过总算一切都结束了。

与此同时，我还收集了很多专题方面的文章，大概有三十来篇，本来是想翻译的，以展示我在程序设计方面的造诣，但是没有时间了(况且在Laramée的文章里我也发挥过瘾了)。我相信读者看了这六篇连载后一定觉得还不够，至少对于想设计象棋引擎的读者来说，仅仅看这么点东西是远远不够的，因此我把这些原文放在我的网站上，相信对于学过程序设计的读者来说，看这些文章应该是没有问题的。

如果你读到有疑问的地方，不妨和我交流，或许你的疑问是很就价值的。如果你找到文中明显的错误(包括错别字)，一定要写信给我，我的工作是无偿为象棋爱好者和程序设计师做的，所以也需要你们的帮助。】

原文：http://www.gamedev.net/reference/programming/features/chess6/
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇 国际象棋程序设计(五)：高级搜索方法
- 下一篇 概述
- 返　回 象棋百科全书——电脑象棋