AI for Game Developers

游戏开发中的人工智能

O'REILLY®
東南大學出版社

David M. Bourg & Glenn Seemann 著
*O'Reilly Taiwan*公司 编译

O'REILLY®

游戏开发中的人工智能

你的游戏是否有角色不能任意走动？是否有角色走进障碍物？是否有非玩家角色不能按照团队运动？现在你就可以掌握高级人工智能（AI）技术以解决这些问题。不管你是编程新手或者是个仅仅想快速学习AI的熟练游戏编程人员，你都会发现《游戏开发中的人工智能》对于理解并应用AI到你的游戏中是非常合适的入门书籍。本书正是为你提供游戏开发方面高级、有用的AI技术的。如果你曾试图使用AI延长你的游戏的生命周期，让你的游戏更加具有挑战性，更重要的是让它们更加有趣，这本书就是为你准备的。

David M. Bourg（畅销书《游戏开发中的物理学》的作者）和**Glenn Seemann**将用非常直观、易懂的语言给你介绍一些诸如有限状态机、模糊逻辑和神经网络之类的技术，全书使用源代码（用C和C++编写）说明这些技术。从基本的诸如追逐、闪躲、基于模式的运动和聚集等游戏行为到玩家行为预测，这本书告诉你怎么应用AI给你的游戏角色提供可信的智能。这些技术包括了适合初级AI开发者的定性（传统的）和非定性（较新的）AI技术的混合。

其他的主题包括：

- 使用基于势函数的单一技术处理追逐、闪躲、聚集和避障等问题。
- 使用包括航点和经典A*算法解决寻径问题。
- 利用AI脚本扩充AI引擎的功能，让设计者和玩家更好地设计和玩游戏。
- 给你的游戏角色赋予基于规则的AI推理能力，包括模糊逻辑和有限状态机。
- 使用概率分析和诸如贝叶斯推理的高级技术处理不确定性问题。

ISBN 7-5641-0507-0

9 787564 105075 >

O'Reilly Media, Inc. 授权东南大学出版社出版

ISBN 7-5641-0507-0

定价：48.00 元

目录

前言

近年来3D视觉影像和仿真实境科技在软硬件层次上都有长足进步，让游戏开发人员得以创建出更吸引人、更令人沉迷其中的游戏环境。接下来要做出更能令人流连忘返的游戏就是改良人工智能（AI）。计算机计算能力的进步，特别是硬件加速处理的图形，可以让更多的CPU资源释放出来，专注在游戏软件某些精细的AI引擎。此外，有大量资源（学术论文、书籍、游戏类的文章以及网站）在探讨AI，协助每位游戏开发人员来运用高级AI技术，而不再仅限于专长是AI的专业人员了。

话虽如此，要读遍诸多技术论文、教科书和网页，对初涉游戏AI开发人员而言是令人望而却步的工作。本书把新手需要的信息收集起来，让他们能一跳就进入游戏AI开发领域。我们介绍许多主题的相关理论，整本书都以程序范例作为辅助手段。

很多普通的游戏开发书籍多少都会谈到AI，然而，那些书看待AI的方式都过于狭隘。可能是因为那些书必须谈很多其他主题，而无法深入谈AI的任何主题。虽然有很多好书专门谈游戏软件AI（我们在后文“其他资源”中列出了好几本），但多数都是针对有经验的AI开发人员，而且通常集中在特定而高深的主题。因此，新手可能还需要其他详谈游戏软件AI基础议题的书。不过，还是有其他书，详细讨论某些特定的游戏软件AI技术，但都限于讨论那些技术而已。

这本书谈了很多游戏软件AI的主题，内容深度适合初涉开发人员。所以，不论你是游戏开发新手还是资深游戏开发人员，如果需要尽快掌握AI技术，比如有限状态机、模糊逻辑、神经网络以及其他议题，这本书都非常适合你。

事前准备

本书是针对游戏软件AI的初学者，我们假定你没有任何AI背景知识。然而，我们假定

你确实知道怎么用C/C++来写程序，也假定你懂得基本向量数学在游戏中的应用，尽管如此，本书附录仍提供了简单的向量数学以便复习，以免你对此过于生疏。

本书概要

本书不打算（也无法）论述游戏AI的所有方面，因为要谈的技术太多，而且几乎每种技术都有变化以适应各式各样的游戏类型、特定游戏结构、游戏场景。我们只介绍新手必须彻底了解的“定性”（传统）和“非定性”（新式）AI技术。以下是各章概要：

第一章 游戏人工智能简介

定义何谓“游戏AI”，讨论当前AI技术的发展，以及AI技术的未来。

第二章 追逐和闪躲

讨论基本的追逐和闪躲技术，以及进级的拦截技术。我们也谈及这些技术在砖块环境和连续环境中的变化。

第三章 移动模式

许多视频游戏中经常出现固定模式的移动，比如守卫的巡逻行为、宇宙飞船的降落等。开发者可将移动模式技术应用于特定行为的程序的编写。

第四章 群聚

群聚方法（flocking method）是A-life算法的实例。A-life算法除了可以做出效果很好的群聚行为外，也是高级群体运动的基础。

第五章 以势函数实现移动

靠势能移动在游戏AI程序中还算相当新颖。这个方法最优越的地方在于可以同时处理追逐、闪躲、成群结队和避免碰撞等行为。

第六章 基本路径寻找及航点应用

游戏开发人员使用很多技术在游戏环境中寻找路径。本章要谈几种方法，包括航点应用。

第七章 A*路径寻找算法

没有谈路径寻找算法的主力——A*——就不能算完整交代路径寻找这个主题，因此，我们以一章的篇幅来讨论A*算法。

第八章 描述式AI及描述引擎

当今的程序员通常只写描述引擎，而由设计者使用工具创建内容和定义AI。本章探讨一些开发人员把描述系统应用在游戏中的技巧，以及他们所得到的益处。

第九章 有限状态机

有限状态机是游戏软件AI的基本要素。本章讨论有限状态机的基础，以及如何予以实现。

第十章 模糊逻辑

开发人员把模糊逻辑和有限状态机结合起来使用，甚至取代有限状态机。本章你将会学到模糊逻辑如何能够优于传统逻辑技术。

第十一章 规则式AI

技术上而言，模糊逻辑和有限状态机都落在基于规则的方法这个大伞之下。本章我们要谈这些方法，以及其他变化的方法。

第十二章 概率概论

游戏开发人员时常使用简单的概率，使游戏较难预测。这种简单的不可预测性让游戏开发人员可以拥有对游戏的实质性控制。本章要谈这种用途的基本概率，顺便作为更高级方法的基石。

第十三章 不确定状态下的决策：贝叶斯技术

贝叶斯技术是概率技术，本章解释如何运用，以便在游戏中做决策并适应游戏。

第十四章 神经网络

“神经网络”技术让游戏具有学习和适应的能力。事实上，从决策判断到预测玩家行为，都可以应用。我们会详谈最广泛使用的神经网络结构。

第十五章 遗传算法

遗传算法提供游戏软件AI演化的可能。虽然遗传算法还不是经常被应用于游戏中，但是它们在某些特定应用方面的潜力是值得令人期待的，尤其是结合其他方法使用时，更为如此。

附录 向量运算

这篇附录示范了如何实现一个C++类，处理2D或3D仿真程序时所需的向量运算。

本书各章可说是彼此独立。因此，你可以按你所想的次序来读各章，不用担心会漏掉前几章内容。唯一例外是探讨概率基本内容的第十二章，如果你对概率一窍不通，读第十三章的贝叶斯法之前应该先读这一章。

此外，我们也鼓励你在自己的程序中试用各种算法。如果你是游戏AI的初学者——本书预想的读者群——你应该想将本书介绍的一些技巧运用到简单的街机游戏或下棋游戏；你也可以考虑利用可扩充的AI工具写一个“队友”，借此印证你的AI构想，而不必去耗费心神撰写无关AI的程序。在第一人称射击游戏的开发领域，这种做法已经逐渐成为一种标准。

排版约定

由于本书保留了部分术语的原貌，为了避免读者的混淆，我们运用了一些字型变化，让读者能轻易辨别词汇本身的含义。

纯文本（Plain Text）
: 用于表示菜单标题、菜单选项、菜单按钮和键盘快捷键（如Alt和Ctrl）。

斜体（*Italic*）
: 用于表示新术语、URL、电子邮件地址、文件名、文件扩展名、路径名、目录和Unix公用程序。

等宽字（`Constant Width`）
: 用于表示命令、选项、分支语句、变量、属性、关键字、函数、类型、类、命名空间、方法、模块、特性、参数、值、对象、事件、事件处理、HTML标记、宏、文件的内容或命令的输出。

等宽黑体字（**`Constant Width Bold`**）
: 用于表示需要用户逐字输入的命令或者其他文字。

等宽斜体字（*`Constant Width Italic`*）
: 用于表示可用用户提供的值替换的文字。

黑体字（**Boldface**）
: 用于表示在正式的输出中相对于标量变量的矢量变量。

其他资源

虽然我们尽可能纳入各式各样的AI技术，但毕竟无法将开发游戏AI所必须知道的一切都涵盖进来。为此，我们整理了一份有用的AI网站和书籍的清单，如果你决定继续钻研游戏AI，它们将是很有帮助的信息来源。

以下是我们认为有用的几个和游戏软件AI有关的知名网站：

- The Game AI Page（*http://www.gameai.com*）。
- AI Guru（*http://www.aiguru.com*）。
- Gamasutra（*http://www.gamasutra.com*）。
- GameDev.ne（*http://www.gamedev.net*）。

- AI Depot（*http://ai-depot.com*）。
- Generation5（*http://www.generation5.org*）。
- The American Association for Artificial Intelligence（*http://www.aaai.org*）。

每个网站都提供了关于游戏AI的信息，以及相关资源的链接。

以下是我们认为有用的书籍。请注意，有些书讨论的是一般性的AI技术，而不是完全针对AI在游戏方面的应用。

- 《Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference》，Judea Pearl 著。（Morgan Kaufmann Publishers, Inc. 出版）。
- 《Bayesian Artificial Intelligence》，Kevin Korb 和 Ann Nicholson合著。（Chapman & Hall/CRC 出版）
- 《Bayesian Inference and Decision》，Second Edition,Robert Winkler 著。（Probabilistic Publishing 出版）。
- 《AI Game Programming Wisdom》，Steve Rabin 著。（Charles River Media 出版）
- 《AI Techniques for Game Programming》，Mat Buckland 著。（Premier Press 出版）。
- 《Practical Neural Network Recipes in C++》，Timothy Masters 著。（Academic Press 出版）。
- 《Neural Networks for Pattern Recognition》，Christopher Bishop 著。（Oxford University Press 出版）。
- 《AI Application Programming》，M. Tim Jones 著。（Charles River Media 出版）。

范例程序

本书宗旨是协助你完成工作。一般而言，你可以自由运用本书的程序代码，而不必征求我们的授权许可，除非涉及重制（reproduce）行为。例如，你可将本书的程序片段运用到你的程序中，而不必事先征询我们的同意，但如果你想把O'Reilly书籍的范例程序烧制成光盘贩卖或散布，则需要事先征求我们的授权。引用本书的文句和范例程序来回答问题，不需要取得许可，但是将本书的范例程序大量整合到你的产品说明书中，则需要取得授权。

当你引用本书资源时，我们希望你注明来源，虽然这不是绝对必要的，但我们会很感谢。注明来源应该包括书名、作者、出版商以及ISBN。例如："《AI for Game Developers》, by David M. Bourg and Glenn Seemann. Copyright 2004 O'Reilly Media, Inc., 0-596-00555-5."

如果你不确定是否应该征询授权，不用客气，请尽量和我们联系：*permissions@oreilly.com*。

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

O'Reilly的每一本书都有专属网页，你可以在此找到关于本书的相关信息，包括可下载的范例程序、勘误表与相关资源的链接。

http://www.oreilly.com.cn/book.php?bn=7-5641-0507-0
http://www.oreilly.com/catalog/ai/

询问技术问题或对本书进行评论，请发电子邮件到：

info@mail.oreilly.com.cn
bookquestions@oreilly.com

最后，您可以在以下站点找到我们：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

我们要感谢编辑Nathan Torkington和Tatiana Diaz，感谢他们的技能、洞察力和耐心。我们也要表示我们对O'Reilly的感激，让我们有机会写这本书。并且，要特别感谢O'Reilly的产品和技术员工，还有负责技术审读和润笔的Graham Evans以及Mike Keith，感谢他们充满灵思妙想的专业见解和意见。最后，我们要感谢我们各自的妻子，Helena和Michelle，感谢她们的支持和鼓舞，忍受我们不断地说："我这个周末得写那本书。"我们不能忘记感谢我们的孩子，一直告诉我们放手去玩。想到这里，就想起游戏中优良AI最重要的目标就是让游戏好玩。来玩吧！

第一章

游戏人工智能简介

就广义而言，多数游戏软件都具有一些人工智能（artificial intelligence，AI）的表现形式。例如，多年来开发者利用AI，让无数游戏角色看起来像是有智慧的生命，诸如经典游戏“小精灵”（Pac-Man）里的魔鬼、“Unreal”第一人称射击游戏（first-person shooter，FPS）里的虚拟队友，以及许多介于两者之间的其他游戏角色。很多类似的游戏和角色都可将之扩大解读为游戏AI（game AI）的表现形式。甚至在比较传统的科学应用领域里也认为人工智能是真实的。

有些开发者认为“路径寻找”也是游戏AI的一部分。Steven Woodcock在其《2003 Game Developer's Conference AI Roundtable Moderator's Report》报告书中指出，有些开发者甚至认为“碰撞侦测”也是游戏AI的一部分（注1）。很显然，有些人将游戏AI定义得更广泛。

我们将对游戏AI做最广泛的定义，也就是包含一切，从简单的追逐和闪躲，到移动模式，以及神经网络和遗传算法。游戏AI或许比较偏向于“弱势AI”（weak AI）（见附加说明“AI的定义”）的范畴，然而，就某方面而言，或许比弱势AI更广义。

使用AI让非玩家角色（nonplayer character，NPC）表现出不同的人格特质，或呈现人类特有的情绪或脾气（惊吓、恼怒等），是游戏设计中可考虑的选项之一。不过，并非所有的NPC都要有AI，在游戏设计上，我们不见得想让NPC拥有人类水平的智力。诸如机器人、龙、啃齿动物之类的非人类生物，可能只会按照我们预先规划的固定模式行动。再者，谁说我们一定要让NPC够精明？让某些NPC当蠢蛋，反而会让游戏的内容更多变和丰富。通常，只有在需要解决相当复杂的问题时，才会使用AI技术。

注1：　Steven Woodcock经营一个很棒的网站，专门谈游戏AI（*http://www.gameai.com*）。

AI的定义

“何谓人工智能？”这是个很难回答的问题。如果你在《美国传统辞典》(《The American Heritage Dictionary of the English Language, Fourth Edition》, Houghton Mifflin出版）里找“人工智能”这个词，可能会找到像这样的定义：“计算机或其他机器有能力执行那些通常认为需要有智慧才能做的事。”不过，有些文献则把人工智能定义成“创建拥有智慧的机器的过程或科学”。

有些专家认为，AI应该视为机器所展示的智能行为，或是智能行为背后的人工大脑。即使是这样的解释，也还不够全面。有些专家对AI的研究，目的在于探究人类智慧的本质，而不是为了建造智能机器。

这样就会引发出一个问题：“何谓智能？”对某些人而言，判定标准是AI技术产生的效果有多么接近人类智慧的表现；其他人则认为还必须满足额外条件，机器才能被视为具有智能，其中有些人主张，意识是智能的必要条件，而且情绪和意识也不可分割；另一派学者则说智能必须要能够解决问题，而且要看起来好像是由人类解决似的；但即使提出这么多的条件，也还不够全面，AI还必须有学习和适应能力，才有资格被视为有智能。

满足上述所有条件的AI，就称为“强势AI”(strong AI)。弱势AI不同于强势AI的地方，在于牵涉到比较宽广的用途和科技，让机器拥有专业化的智力。游戏AI属于弱势AI的范畴。

一言以蔽之，游戏AI的定义相当宽广而且灵活性很大。无论采取何种手段，只要能给人以某种智能程度的“错觉”，让游戏更能令人沉迷于其中、更具有挑战性，最重要的就是要好玩，那才能看作是游戏AI。就像在游戏中用到物理学那样，好的AI会使游戏更令人无法自拔，吸引住玩家，让他们在现实世界中的时间暂停。

定性与非定性AI

游戏AI技术通常分成两种，定性（deterministic）和非定性（nondeterministic）。

定性

定性行为或其表现是特定的，而且可预测，没有不确定性。其具体实例是简单的追逐算法。你可以明确地塑造一个非玩家角色，沿着XY坐标轴前进，往某目标点移动，直到该角色的XY坐标和目标点的坐标重叠。

非定性

与定性行为相反，非定性行为有某种程度的不确定性，有点不可预测（至于不确定到什么程度，则与人们对所采用的AI方法的理解的难易程度有关）。其具体实例是，让非玩家角色学习到适应玩家的作战战术。这样的学习能力可以利用神经网络、贝叶斯技术（Bayesian technique）或遗传算法而得到。

定性AI技术是游戏AI的基础。这些技术的结果是可预测的，效率高，容易实现、理解、测试和调试。虽然定性方法有很多，但是预先考虑各种场景，以及明确写出所有行为的重担都落在了开发者的肩上。再者，定性方法无法帮助NPC学习或演化，玩家只要小玩一下游戏，就可预测出NPC的定性行为。我们可以这么说，使用定性行为，会限制游戏软件的“玩命”。

非定性方法能促进学习，让玩家在玩游戏时难以预测。再者，开发者无需事先预想所有可能场景，写下所有明确的行为。非定性方法可以让NPC自己学习，并推论出新行为，甚至引发所谓的突现行为（emergent behavior），也就是没有明确指示而出现的行为。本书将谈到的群聚（flocking）和神经网络算法就是突现行为的好例子。

长久以来，开发者总是对非定性AI保持距离，但这已逐渐得到改变。由于无法预测，就很难测试和调试；你怎么测试玩家所有可能的行动，以确保游戏软件不会在某些情况下做出蠢事？游戏开发人员面对的是不断缩短的开发周期，因此，开发新科技并予以测试，使其满足成品标准，就变得格外困难。超短的开发周期，使得开发者难以全部了解最新的AI技术，并估算新技术在大量销售商业游戏软件中的作用。

另一个因素也限制了游戏AI的发展，至少是从最近开始的，那就是开发者把他们的注意力放在图画的质量上。结果，为了做出更好更快的图画技术，所耗费的人力、物力（包括硬件加速技术），早就足以提供更多资源，开发更好、更高级的AI了。基于这样的事实，再加上要做出下一个热门游戏的压力，促使游戏开发人员更全面地探索非定性技术。稍后我们会再回来谈这一点。

现有的游戏AI技术

也许，游戏中最广泛使用的AI技术就是作弊。例如，在战争模拟游戏中，由计算机控制的玩家，可以得知对手（人类）的所有信息（基地位置、军队种类、数量和所在地等），根本不用像人类玩家那样派出侦察兵去收集这类情报。这种作弊手法是很常见的，让计算机可以比人类玩家取得某种优势。然而，作弊也可能适得其反。如果玩家一眼就看出计算机在作弊，玩家说不定会认定他的努力都是白费的，从而对这个游戏失去兴趣。此外，不平等的作弊会让计算机控制的对手太强大，使得玩家无法打败计算机。同样地，

如果玩家发现他的努力白费，可能也会失去兴趣。作弊必须采取中庸之道，替玩家创建足够的挑战性，让游戏既有趣又好玩。

当然，作弊不是唯一常用的现有AI技术。有限状态机（finite state machine，FSM）是到处可见的游戏AI技术。第九章我们会详谈，基本观念是列举出计算机控制的角色，一连串动作或状态，再利用if-then条件语句检查各类情况和满足条件，据此执行动作或状态，或者在动作或状态之间做转换。

开发者时常在模糊状态机（fuzzy state machine）中用到模糊逻辑（fuzzy logic），让最后执行的动作难以预测，减少必须以if-then语句大量列举条件的重担。在有限状态机中，你可能有“if距离值为10且健康值为100，then攻击”这样的规则，但是，模糊逻辑与此不同，可以让你用不太精确的条件设计规则，比如“if靠近而且足够健康，then强力攻击”。第十章会谈模糊逻辑。

在各类游戏中，非玩家角色的基本任务，是必须有效地找出有效的路径。在战争模拟游戏中，非玩家角色的军队必须能够通过各种地形，避开障碍物，抵达敌军所在地。第一人称射击游戏中的生物，必须能通过地牢或建筑物，以便和玩家相遇或逃离玩家视线。这种场景是数不胜数的，毋庸置疑，AI开发人员会很注意路径寻找一事。第六章我们会谈一般的路径寻找技巧，到第七章才谈重要的A*算法。

这些只是现有游戏AI技术的少数几种而已，其他还有以规则为主的描述式系统，以及某些人工生命技术，种类繁多，这里不再一一罗列。人工生命技术在机器人应用领域很常见，而且得到开发人员的改编，运用到视频游戏中，获得了很大的成功。基本上，人工生命系统就是一种人造系统，可以展现出符合人性的行为。这些行为属于突现行为，其发展是结合各种低层次算法运作后的结果。本书会讨论人工生命的实例以及其他技术。

游戏AI的未来

游戏AI的下一件大事就是“学习”。游戏上市后，所有非玩家角色的行为，不再事先安排，游戏玩得愈久，游戏就会更多地演化和学习，更具适应性。这样的游戏会跟玩家一起成长，玩家也难以预测游戏行为，因此就能扩展游戏的生命周期。游戏会学习并演化，造就了本身无法预测的本质，显然地，这让AI开发者带着相当大的惶恐去探索“学习”技术。

“学习”与“角色行为反应”技术，属于上文提过的非定性AI的范围，所以有相当大的难度。明确地讲，这种非定性的“学习”AI技术，要花更长的时间开发和测试。再者，要了解AI究竟会做什么也更加困难，这就使得调试也更困难。已经证实这些因素是“学习”AI技术得以广泛应用的巨大障碍。不过，这一切都在改变之中。

几个主流游戏都用了非定性AI方法，比如“Creatures”、“Black & White”、“Battlecruiser 3000AD”、“Dirt Track Racing”、“Fields of Battle”以及“Heavy Gear”。这些游戏的成功，重新点燃了人们对“学习”AI技术的兴趣，诸如决策树（decision tree）、神经网络、遗传算法以及概率方法（probabilistic method）。

这些成功的游戏软件，运用非定性方法时，也搭配用较传统的定性方法，只有在最适合于解决问题的情况下，以及所需之处才使用。神经网络不是仙丹，无法解决游戏软件中所有的AI问题，然而，你可以在混合的AI系统中，解决特定的AI任务，借此获得令人印象深刻的结果。这就是我们提倡使用这些非定性方法的手段。这样一来，至少你可以把AI中不可预测而且难以开发、测试和调试的部分隔离出来，同时又能让你的AI系统的大部分，都保持传统的形式。

本书会讨论传统的游戏AI技术，以及相当新颖、刚出现的AI技术。我们想帮助你全面了解有哪些技术可用于游戏AI。此外，我们也想让你学会几种令人期待的新技术，让你走在游戏AI的前沿。

第二章

追逐和闪躲

本章的焦点是追逐和闪躲，这是到处可见的问题。无论你开发的是太空战机射击游戏、策略模拟游戏，还是角色扮演游戏，游戏中的非玩家角色都有机会必须试着追逐或逃离玩家角色。在动作游戏或者街机游戏中，面对的情况就是要得知敌方太空战机的行踪，然后和玩家的战机交战。在角色扮演冒险游戏中，也许需要有个巨人或者某种可爱的生物，追着你的玩家角色跑。在第一人称射击游戏和飞行模拟游戏中，你得算出导弹的轨迹，攻击玩家或玩家的飞行器。无论哪种情况，你需要使用一些逻辑手段，让NPC扮演的追击者追逐猎物，或让猎物逃避。

追逐/闪躲问题由两部分组成。首先要做出追或逃的决策判断。其次是开始追或逃，也就是让追击者追猎物，或者让猎物尽可能离追击者远一点，免得被抓到。在某种意义上，有人认为追逐/闪躲问题还包括第三部分：避开障碍物。追逐和闪躲的同时还要闪避障碍物，这确实让问题复杂化，使得算法更难写。本章不讨论避开障碍物，到了第五章和第六章我们会再来谈这个问题。本章我们把焦点放在问题的第二部分：开始追逐或闪躲。至于问题的第一部分（追或逃的决策判断），留待后文谈论到状态机和神经网络时再来讨论。

让追击者追逐猎物是最简单、最容易写而且也是最常用的方法，就是在每次的游戏循环中，更新追击者的坐标，让追击者和猎物的坐标离得愈来愈近。这种算法不去管追击者和猎物各自行进的方向和速度。虽然这种做法有直接的效果，追击者会不断往猎物的位置移动，除非被障碍物挡住，但是，这种做法有其限制，稍后再加以讨论。

除了这种最简单的方法之外，还有其他方法可以用，也许更能满足你的需求，审视你的游戏所需的条件。例如，游戏中如果整合了实时物理引擎，你就可以采用一定方法，考虑追击者及猎物的位置和速度，让追击者试着拦截猎物，而不是傻乎乎地一直追下去。

在这种情况下，相对位置和速度的信息，可作为某种算法的输入数据，由该算法求出适当的驱动力(例如，推力)，把追击者引向猎物。不过，另一种方法是利用势函数(potential function)，以某种方式改变追击者的行为，使其去追逐猎物，或者更明确地讲，是让猎物引起追击者的注意。同样地，你也可以用类似的势函数，让猎物逃离追击者，或者让追击者对猎物产生排斥感。第五章将介绍势函数。

本章我们要探索几个追逐和闪躲的方法，从最基本的方法开始。我们也会提供实现这些方法的范例程序，让你运用于砖块式(tile-based)与连续移动式(continuous-movement)环境。

基本的追逐和闪躲

如前所述，最简单的追逐算法，就是根据猎物的坐标来修改追击者的坐标，使两者间的距离逐渐缩短。这是追逐和闪躲常用的基本方法。(也可以将此方法反之运用，不再是缩短追击者和猎物间的距离，而是试着扩大该距离。)这个方法的程序代码如例2-1所示。

例 2-1：基本追逐算法

```
if (predatorX > preyX)
     predatorX--;
else if (predatorX < preyX)
     predatorX++;

if (predatorY > preyY)
     predatorY--;
else if (predatorY < preyY)
     predatorY++;
```

此例中，猎物的坐标是preyX和preyY，而追击者的坐标是predatorX和predatorY。游戏循环每运行一轮时，就比较两者的x、y坐标，若追击者的x坐标大于猎物的x坐标(可解释为追击者在猎物之前或之后，视你所用的坐标象限而定)，则递减追击者的x坐标；但如果追击者的x坐标小于猎物的x坐标，则递增追击者的x坐标；y坐标的调整逻辑也一样。最后的结果就是，每当游戏循环运行一轮后，追击者就会越接近猎物。

运用相同的方法，只要颠倒一下判断逻辑，就可实现闪躲效果，如例2-2所示。

例 2-2：基本闪躲算法

```
if (preyX > predatorX)
     preyX++;
else if (preyX < predatorX)
     preyX--;

if (preyY > predatorY)
     preyY++;
```

```
else if (preyY < predatorY)
     preyY--;
```

在砖块式构成的游戏中，游戏领域（game domain）会切割成不连续的砖块（正方形、六边形等），而玩家位置会固定在某个砖块上。移动时都是以砖块为单位，而且玩家前进的方向是有限制的。在连续环境中，则以点坐标表示游戏领域中的位置，玩家也可以往任何方向移动。

无论是砖块还是连续环境，例2-1与例2-2所示范的技巧都适用。在砖块环境中，x、y坐标就是砖格的列、行编号。也就是x、y坐标应该都是整数。在连续环境中，x、y、z（如果是3D游戏）就应该是实数，构成游戏领域的笛卡儿坐标（Cartesian coordinate）。

毋庸置疑，此法虽然简单，但确实有效。追击者会以不屈不挠的决心追逐猎物。示范程序Demo2-1（可从本书的网站下载*http://www.oreilly.com/catalog/ai*），实现了砖块环境中基本的追逐算法。相关的程序代码见例2-3。

例2-3：基本的砖块环境追逐实例

```
if (predatorCol > preyCol)
     predatorCol--;
else if (predatorCol < preyCol)
     predatorCol++;

if (predatorRow> preyRow)
     predatorRow--;
else if (predatorRow<preyRow)
     predatorRow++;
```

注意例2-3和2-1之间的类似。唯一的差别是例2-3改用列和行，不是用点的x、y坐标。

这种基本方法的缺点，在于追逐或闪躲路径过于死板。图2-1是AIDemo2-1程序中，巨人追赶玩家时所走的路径。

正如你所见，巨人会沿着对角线走向玩家，直到两坐标之一和玩家的相等（此例是横坐标）（注1）。接着，巨人会沿着另一个坐标轴往玩家方向前进，此例为纵轴。显然，这看起来很不自然。比较好的做法是让巨人走直线去追玩家。实现这种算法不会太困难，下一节我们就会谈到。

注1：在砖块环境中，沿着对角线移动时，似乎走得比较快。这是因为正方形的对角线长度是其边长的$\sqrt{2}$倍。因此，每一步沿对角线方向移动的距离，要比转直角的移动方式快$\sqrt{2}$倍。

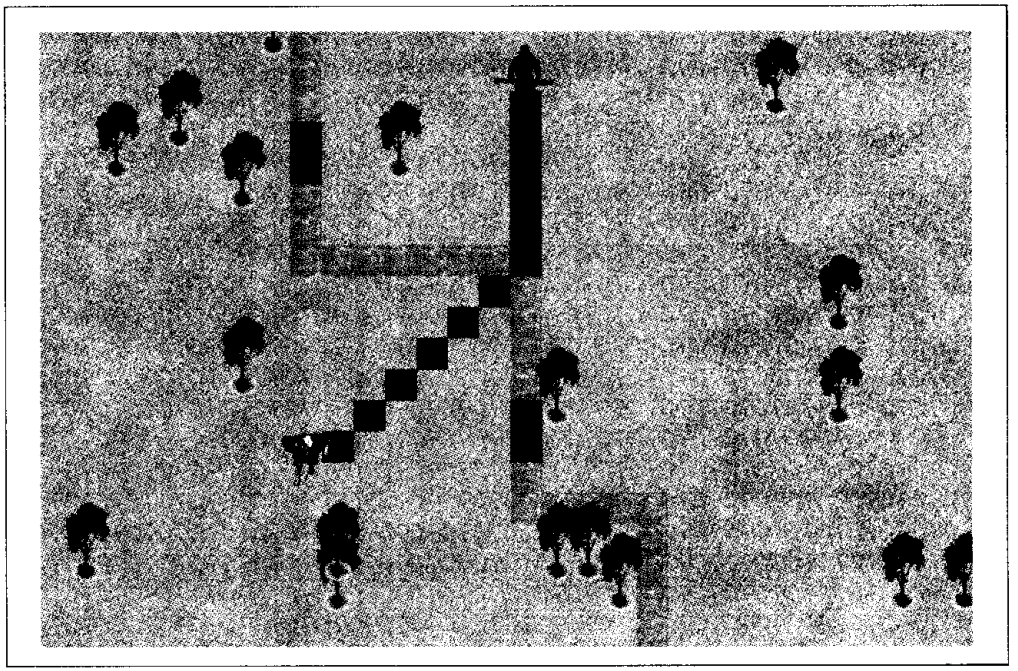


图 2-1：砖块环境中的基本追逐

视线追逐

本节介绍一些追逐与闪躲算法中的视线法（line-of -sight）。视线法主要是让追击者沿着猎物的直线方向前进，让追击者永远面对着猎物当时位置直进，当猎物站着不动时，追击者所走的路径是直的，但是当猎物移动时，路径就不一定是直线了，可能是弯弯曲曲的，如图 2-2 所示。

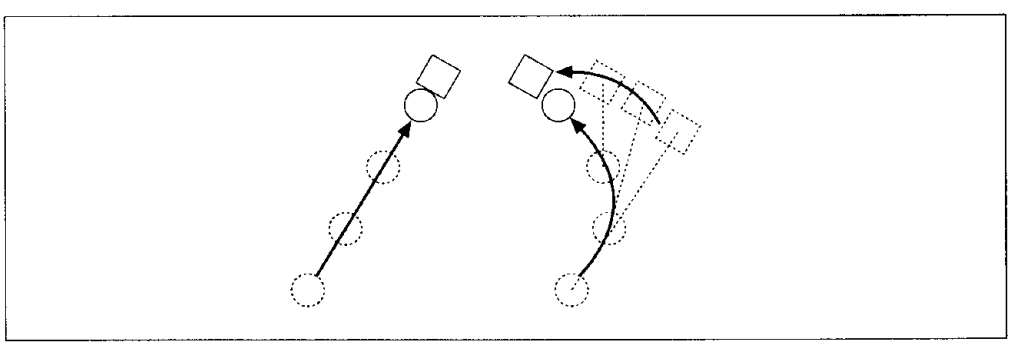


图 2-2：视线追逐

在图2-2中，圆圈代表追击者，方块代表猎物。虚线图形指的是起点和中途位置。在左边的场景中，猎物是不动的，因此追击者可以直线追击猎物。在右边的场景中，猎物不停地移动，追击者的方向也随之改变。游戏循环每运行一轮或经过一段时间，就必须重新计算追击者朝向猎物的新方向，而留下曲折的追逐轨迹。

视线追逐法的效果，显然比基本追逐法来得自然。接下来的两小节分别介绍两种实现视线追逐效果的算法；第一种适用于砖块环境，而另一种则适用于连续环境。

砖块环境中的视线追逐

在砖块环境中，任何游戏角色都必须位于砖格上，不能同时跨越不同砖格。在连续环境中，角色的位置通常以点表示，要显示在屏幕上时，才换算成最适当的屏幕图素坐标。本质上的差异，使得游戏角色在砖块环境下的移动，具有连续环境下所没有的限制。当游戏角色在连续环境中移动时，没有限定只能移动到邻近屏幕图素上，因为图素通常足够小，每次重绘画面时，多跳过几个图素，也不会牺牲动作的流畅感。

然而，在砖块环境下，由于砖块不是对应到单一屏幕图素，而是特定范围内的一整片图素，所以角色的移动路线会呈现锯齿状；为了尽量让锯齿状现象不那么明显，同时也为了避免跳格现象，游戏角色每次都只能移动到邻近砖块。可能的移动方向，取决于砖块本身的形状与排列方式，对于采用方形砖排列的游戏，就只有八种可能的移动方向，而方向准确度上的限制，会引发一个有趣的问题：从数学观点来看，没有任何一个方向可以精确表达目标的真正方向（参见图2-3）。

如图2-3所示，这八个可能的方向都无法直接到达目标。我们需要做的是找出邻近的八个砖块中，有哪一个是巨人移动后，尽可能以直线走向玩家的。

你可以用简单的追逐算法让巨人不屈不挠地追着玩家，算出来的还是走向玩家最短的可能路径。既然如此，缺点何在？其中之一和美学有关。在砖块环境中，简单的追逐方法不见得都能求出视觉上的直线。图2-4说明了这个观点。

不用简单追逐方法的另一个原因是，如果有一群追击者（诸如一群怒火中烧的巨人）往玩家处聚拢时，简单追逐方法会出现意想不到的副作用。使用简单方法时，在以目标为原点的坐标系中，他们会沿着对角线走到距离最近的坐标轴，然后再沿着该轴走向目标。这样可能会让他们排成单一纵队发起攻击。比较细致、自然的做法应该是，分别从不同方向直接向目标逼近。

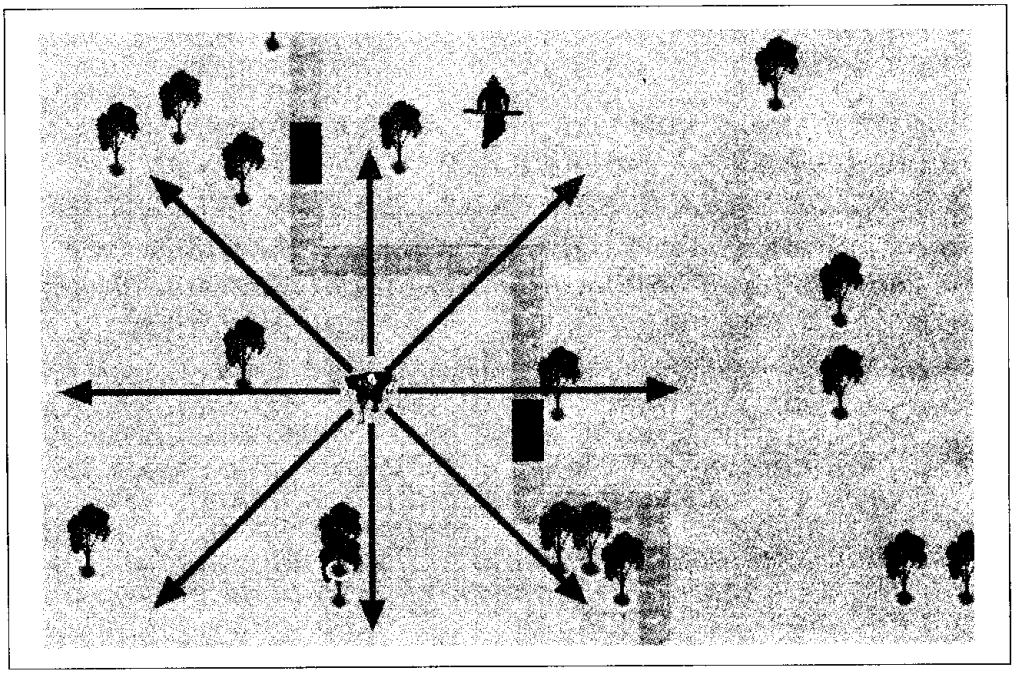


图 2-3：砖块环境中的八种移动方向

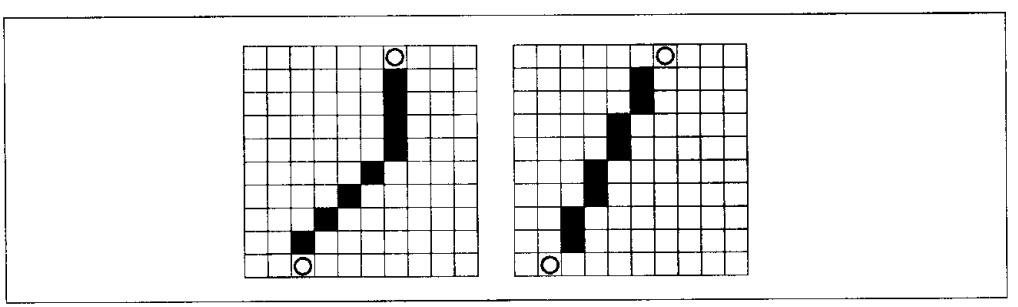


图 2-4：简单追逐与视线追逐

我们注意到一个有趣的现象，图 2-4 所示的两种路径的距离其实相等。然而，视线法似乎更自然、直接，看起来就像巨人更有智能。所以，视线法的目标就是算出一条路径，让巨人看起来是沿着直线走向玩家。

解决这个问题的方法是使用标准线段（standard line）算法，这种算法通常是在图素环境中画线段。本质上，把砖块环境中的每个砖块当成放大的屏幕图素。然而，不是在屏

幕上把图素配上颜色来画线，而是让这个线段算法，告诉我们巨人应该走哪些砖块，才能以直线走向目标。

你可以用各种方法算出线段的每个点，此例中，我们打算采用Bresenham线段算法。Bresenham算法是在图素环境中画线最有效的方法之一，但这并不是求算路径的唯一原因。Bresenham算法很有吸引力，因为和其他线段绘制算法不同，绝不会在线段最短的那个轴上画出两个相邻的图素。就我们寻找路径的需求而言，这表示巨人会在起点和终点之间走最短可能路径。图2-5中Bresenham算法（左边）和其他线段算法作比较，其他算法偶尔会在最短轴上多画出好几个图素。如果算法会算出像右边这样的线段，则巨人就会多走好几步多余的路径。虽然也可以追到猎物，但却不是最短，也不是最有效的路径。

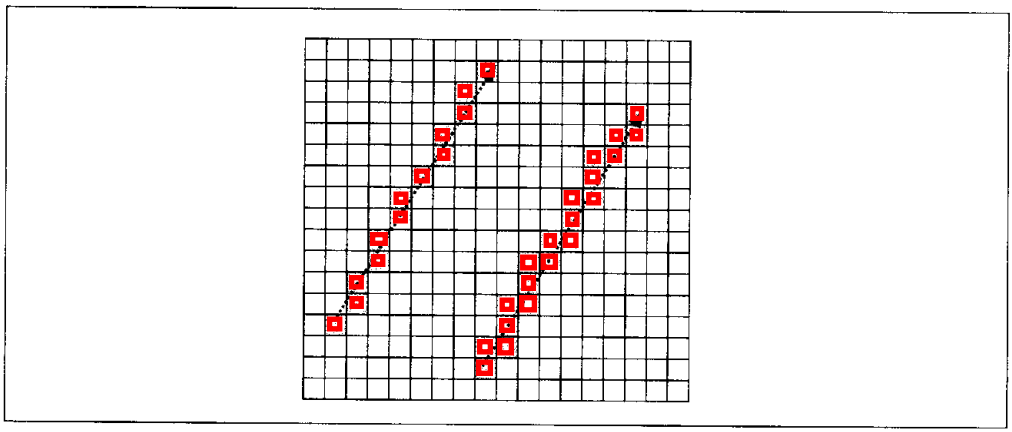


图2-5：Bresenham与其他线段演算法的比较

如图2-5所示，像右边这样的标准算法，在寻找路径时，会把起点及终点间的联结线段、彼此连接的砖块都算进来。对路径寻找的应用而言，这并非想要的结果，因为这不能找出最短可能路径。从这方面来讲，Bresenham算法能得出令人更满意的结果。

计算巨人移动方向的Bresenham算法，会以起点（也就是巨人位置的行和列）和终点（也就是玩家的位置的行和列）为数据，算出巨人要走的一连串步伐，使其能以直线走向玩家。记住一点，每次巨人的猎物（此例是玩家）改变位置时，都要调用一次这个函数。一旦猎物移动了，前一次算出来的路径就无效了，必须再重算一次。例2-4到2-7示范了如何使用Bresenham算法建立巨人走向猎物的路径。

例2-4：BuildPathToTarget()函数

```
void ai_Entity::BuildPathToTarget (void)
{
```

```
int nextCol=col;
int nextRow=row;
int deltaRow=endRow-row;
int deltaCol=endCol-col;
int stepCol, stepRow;
int currentStep, fraction;
```

如例2-4所示，这个函数使用了ai_Entity类中储存的值，建立路径的起点和终点。col和row之值是路径的起点坐标，也就是巨人当前的位置。endRow和endCol之值是路径的终点坐标，也就是猎物的位置。

例 2-5：路径初始设定

```
for (currentStep=0;currentStep<kMaxPathLength; currentStep++)
{
    pathRow[currentStep]=-1;
        pathCol[currentStep]=-1;
}

currentStep=0;
pathRowTarget=endRow;
pathColTarget=endCol;
```

从例 2-5 可知，行和列的路径数组已初始化。每次猎物的位置改变后，这个函数就会被调用，所以在计算新值前，必须把旧路径清除掉。

离开例 2-5 函数后，pathRow和pathCol这两个数组，就含有巨人走向猎物路径的每个点的坐标（行、列值）。再来更新巨人的位置就很简单了，每当巨人准备好走下一步时，从两个数组分别取出下一对坐标，更新巨人的位置即可。

如果这是画线函数，储存在路径数组中的坐标，就是构成该线段的每个图素的坐标。

例 2-6 利用先前算出的deltaRow和deltaCol决定路径的方向。

例 2-6：路径方向计算

```
if (deltaRow < 0) stepRow=-1;  else stepRow=1;
if (deltaCol < 0) stepCol=-1;  else stepCol=1;
deltaRow=abs(deltaRow*2);
deltaCol=abs(deltaCol*2);
pathRow[currentStep]=nextRow;
pathCol[currentStep]=nextCol;
currentStep++;
```

这里也会指定路径数组中的初始坐标值，此例是巨人的位置。

例 2-7 是 Bresenham 算法的精粹。

例 2-7：Bresenham 算法

```
if (deltaCol >deltaRow) {
     fraction = deltaRow *2-deltaCol;
     while (nextCol != endCol)
      {
       if (fraction >=0)
        {
         nextRow =nextRow +stepRow;
         fraction =fraction -deltaCol;
        }
       nextCol=nextCol+stepCol;
       fraction=fraction +deltaRow;
       pathRow[currentStep]=nextRow;
       pathCol[currentStep]=nextCol;
       currentStep++;
     }
    }
   else
    {
     fraction = deltaCol *2-deltaRow;
     while (nextRow !=endRow)
      {
       if (fraction >=0)
        {
           nextCol=nextCol+stepCol;
           fraction=fraction -deltaRow;
        }
       nextRow =nextRow +stepRow;
       fraction=fraction +deltaCol;
       pathRow[currentStep]=nextRow;
       pathCol[currentStep]=nextCol;
       currentStep++;
      }
     }
    }
```

Bresenham算法的原理，是计算每一点与终点之间的横轴与纵轴，然后比较两轴的长度，哪个轴比较长，就往该方向前进，如果两轴等长，则往斜边前进。所以，一开始的`if`条件语句便以`deltaCol`和`deltaRow`的值判断哪个轴比较长。如果列轴较长，则执行`if`语句后第一区块的程序代码。如果行轴较长，则执行`else`之后的程序代码。然后，这个算法将沿着较长的轴去走，算出沿着线段上的每个点。图 2-6 是使用 Bresenham 视线算法，求出巨人应该行走的路径实例。就此例而言，行轴较长，所以会执行`else`之后的程序代码。

图 2-6 所示的是巨人所走的路径，当然，前述的程序实际上没有画出路径。那些程序只是将路径上每个点的坐标分别存入`pathRow`与`pathCol`数组，由其他函数利用这些储存的值将巨人引导至猎物所在的位置。

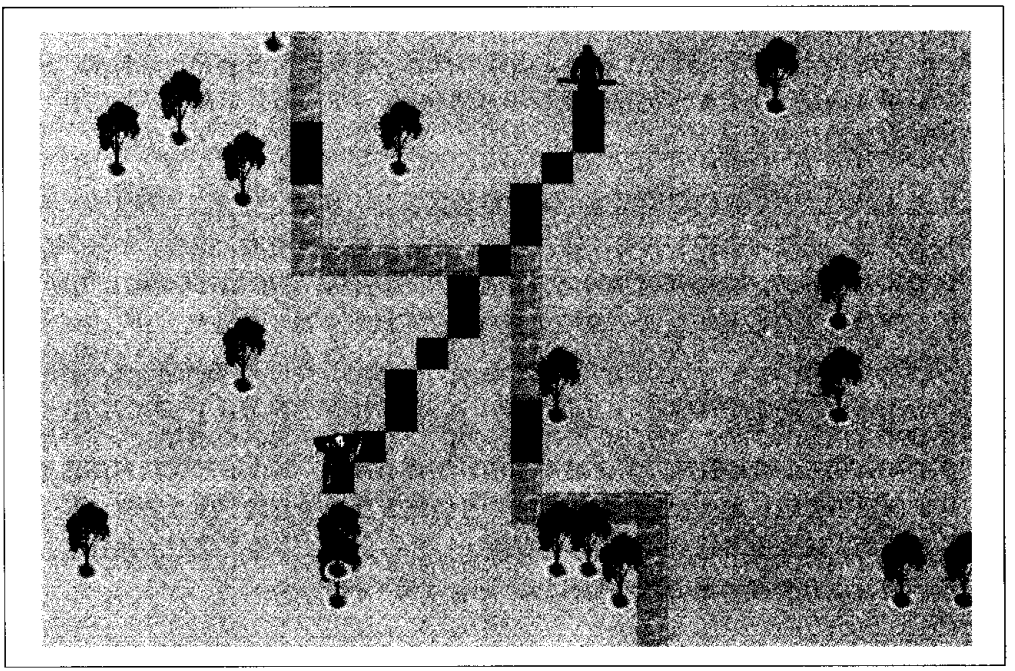


图 2-6：砖块环境下，Bresenham 演算法算出的视线追逐路径

连续环境的视线追逐

Bresenham算法对砖块环境而言是相当有效的方法。本节我们要讨论连续环境中的视线追逐算法。更精确地说，我们要教你如何实现简单的追逐算法，让你可以在结合了物理引擎的游戏中使用——游戏的个体移动都是由力和扭力（force & torque）来推动的，比如飞机、宇宙飞船、气垫船等（译注 1）。

本节要讨论的实例，采用简单的平面、刚体物理引擎，用来计算追击者和猎物的驱动力。你可以从本书网站下载此示范程序Demo2-2的完整源代码。我们会尽量谈到了解这个范例所必需的刚体物理学知识，但不会太深入。有关这个主题的完整探讨，请参考《Physics for Game Developers》（O'Reilly）这本书。

假设有一个游戏场景。玩家有向前行进的推力，和转弯所需的转向力（steering force）控制其载具。计算机也以相同的力学机制控制其载具。我们让计算机扮演追击者角色

译注 1：torque，扭力或转矩、扭矩。

(追击者)，玩家（猎物）要负责逃跑。假设追击者知道猎物的唯一信息，是猎物当前的位置。有了这个目标位置，再加上追击者当前的位置，就可以在追击者和猎物之间拉出视线。利用这条视线决定如何把追击者引导至猎物处。(下一节会示范另一种方法，以向量表示玩家的位置和速度。)

开始谈追逐算法之前，我们想说明追击者和猎物载具怎么移动。所有载具都一样，都由推力控制前进、由转向力控制转弯。要右转时，就必须施加转向力，把载具的头调向右边。同理，要左转时，就必须施加转向力，把载具的头调向左边。就此例的目的而言，我们假设转向力来自载具机头两侧的喷射推进器，可以把前端推往任何一边。图2-7说明了这两种力的操作。

在连续环境中的视线算法，主要在于控制追击者转向力的启动时机与方向，使其随时保持面向猎物的姿态。每台载具的最大速率和偏转率都有限制，因为前进或转弯的速率越快，其反向作用力也就越大；此外，载具也不可能绕着圆心一直转。因为旋转半径是线性速率的函数，线性速率越高，旋转半径就越大。这样会让每台载具采取的路径看起来更平滑而自然。若要绕着圆心转，载具应开得非常慢才行。

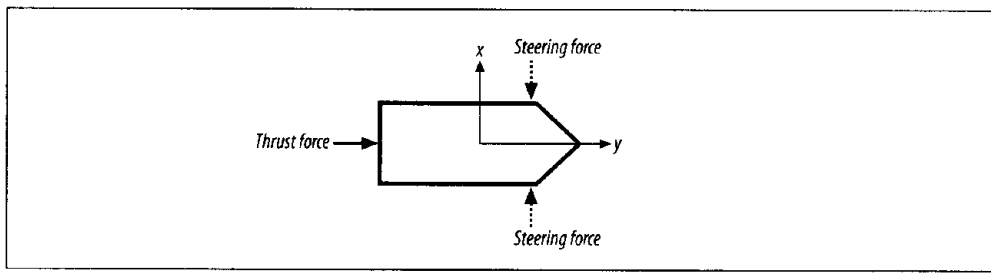


图2-7：载具动力示意图

例2-8所示的函数，会计算追击者自己和猎物之间的相对位置，并凭借调整转向力来保持面对猎物的方向。这段程序必须在每次物理引擎循环运行一轮时，就被重新执行一次，只有这样才能达到视线追击效果。

例2-8：视线追逐函数

```
void    DoLineOfSightChase (void)
{
   Vector   u, v;
   bool     left = false;
   bool     right = false;

   u = VRotate2D(-Predator.fOrientation,
                 (Prey.vPosition - Predator.vPosition));
```

```
    u.Normalize();

    if (u.x <  -_TOL)
         left = true;
    else if (u.x > _TOL)
         right = true;

    Predator.SetThrusters(left, right);
}
```

由此可见，例2-8的算法相当简单。一开始就定义了四个局部变量。u和v是追击者与猎物的向量，它们所属的Vector类，是我们规定的类（参见附录），该类提供了所有基本的向量运算，诸如向量加法、减法、内积、外积以及其他运算。另外两个局部变量left和right是一组布尔变量。它们代表该方向的转向力是否有作用，在直线前进的情况下，这两个变量的初始值都赋以false。

在局部变量的定义之后，首先要计算追击者到猎物之间的视线：

```
u = VRotate2D(-Predator.fOrientation,
              (Prey.vPosition - Predator.vPosition));
```

其中的(Prey.vPosition - Predator.vPosition)是以追击者与猎物的全局坐标计算两者之间的相对位置向量，而VRotate2D()函数将此向量转换成追击者的局部坐标。VRotate2D()函数需要两个参数，一个是局部坐标系统的基准点，另一个是全局坐标系统中的向量，它能将该向量转换成相对于局部坐标基准点的向量。

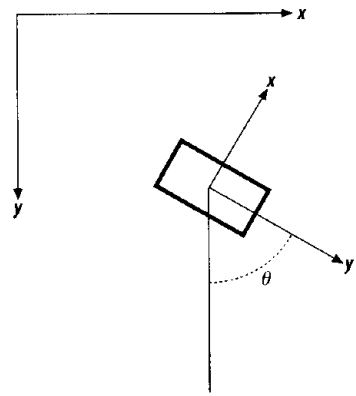
接着，我们使用Normalize()将得到的向量u标准化，也就是转换成1个单位长度的向量。

有了从追击者指向猎物的单位向量u，便可据此判断猎物是在追击者的左边、右边，还是正前方，并据此调整方向。以追击者为原点的局部坐标，其y轴是从背后指向正前方，也就是说，载具总是面对着局部坐标y轴的方向。

所以，从追击者的局部坐标系统来看，如果猎物的x坐标是负值，那么猎物位于追击者的右边，因此，应该启动左边的转向力，调整追击者的前进方向，使其能再度朝猎物方向前进。同理，如果猎物的x坐标值是正的，则位于追击者的左边，应该启动右边的转向力，调整追击者的方向。图2-8是两侧推进器应该如何启动的测试图。如果猎物的x坐标是0，则无需启动两侧的推进器，直接前进即可。

全局坐标系统与局部坐标系统

全局坐标系统的原点固定于整个空间中的特定位置，不会移动；这让我们有一个统一的标准描述物体在空间中的位置。另一方面，局部坐标系统的原点是固定在物体上，当物体在全局坐标系统中移动时，原点的全局坐标也跟着物体移动；当物体旋转时，局部坐标系统的方向也跟着旋转。



只要知道该物体相对于全局坐标系统的方位，就可以利用下列方程式，把某一点的全局坐标转换成该物体的局部坐标：

$$x = X\cos\theta + Y\sin\theta$$
$$y = -X\sin\theta + Y\cos\theta$$

这里的 (x, y) 是局部坐标，也就是对应到全局坐标的 (X, Y)。

最后一行程序调用rigidbody类（Predator对象所属的类）的SetThrusters()成员函数，该函数会设定物体左右两侧的转向力值，让仿真循环改变物体的运动方向。为了简化起见，我们只设定固定的转向力，如果要深入些，你可以依据方向夹角的大小（或是你认为合理的其他参数）调整转向力大小。

这个算法的结果如图2-9所示。

图2-9是追击者和猎物行走的路径。仿真运算开始时，追击者位于画面的左下角，而猎物位于右下角。猎物随时间往画面左上角直线移动。追击者的路径则是弯曲的，因为他不断调整自己的方向，保持朝着移动中的猎物前进。

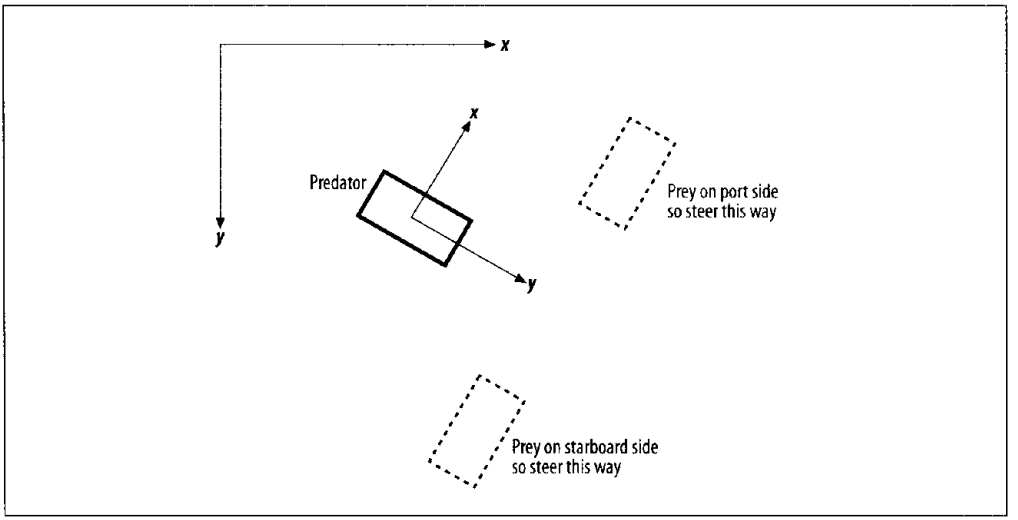


图 2-8：测试转向力

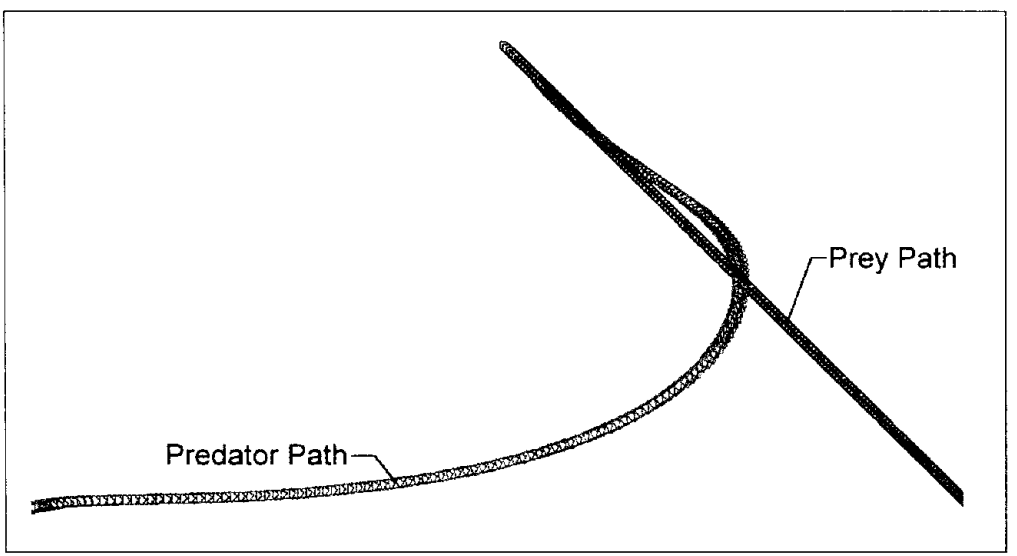


图 2-9：连续环境中的视线追逐

如同本章前面讨论过的基本算法那样，视线算法也会让追击者傻乎乎地去追逐。追击者只会笔直地朝猎物前进，一般来说最后会跟在猎物后面，除非他的速度快过于猎物，而且一旦追过头，他会再绕回来，再次朝着猎物前进。你可以设计一些控制速率的方法，让追击者在接近猎物时自动放慢速度，以免发生追过头的现象。比方说，当两者的距离

小于某个预定的距离时，就降低追击者的前进推力。两物体之间的距离，等于两者的位置向量相减所得的向量的长度。

如果你想让计算机控制的载具负责闪躲而不是追逐，所要做的就是把例2-8的大于（>）和小于（<）符号反向即可。

拦截

上一节讨论连续环境中的视线追逐算法，可以有效地使追击者一直朝着猎物方向前进。但是，这种算法的缺点是直接朝着猎物方向前进，从空间或者时间的角度来看，不一定都是追上猎物的最短路径。再者，视线算法的结果，通常都是追击者跟在猎物后面，除非追击者跑得比较快，但这种情况又会跑过头。大多数情况下（例如，飞行器发射的导弹），比较合理的解决办法，是让追击者在猎物路径上的某个点予以拦截。这样从时空角度来看，可以让追击者以最短的时间或路径追到猎物。另外，拦截算法甚至可以让速度较慢的追击者，有机会拦截到速度较快的猎物。

为了说明拦截算法怎么操作，我们要以先前提过的，结合物理机制的游戏场景作为基础。事实上，把基本的追逐算法改成拦截算法，所需做的就是在追逐函数内加几行程序代码。不过，看程序代码之前，有必要先理解拦截算法运作的原理。（其实也可以运用砖块环境中的视线追逐法。）

拦截算法的基本原理是能够预测猎物未来的位置，然后直接到那个位置去，使其能和猎物同时到达同一位置。图2-10说明了这一点。

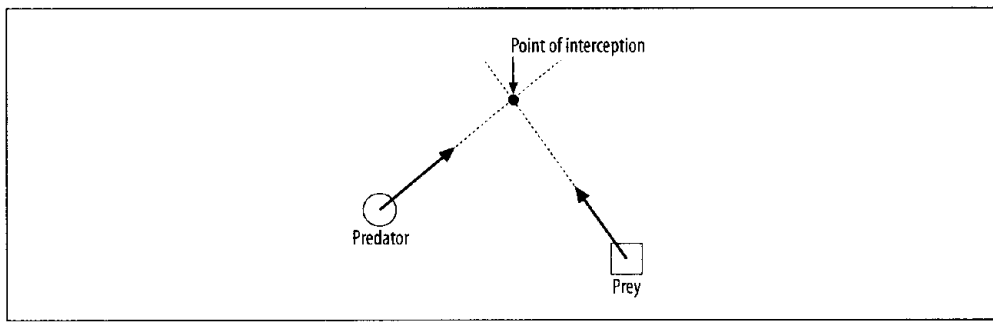


图2-10：拦截

一眼就能看出，预测的拦截点，似乎就是猎物路径上最接近追击者的点。这是计算点到直线的最短距离的问题，也就是某点到某线段的最短距离，是过此点垂直于该线段的直线距离。但这个点不一定是拦截点，因为最短距离问题，没有考虑到追击者和猎物间的

相对速度。追击者有可能在猎物之前，先到达猎物路径上最短距离的点。此时，如果要拦截猎物，追击者必须停下来，等猎物送上门。如果把追击者换成导弹，瞄准像飞行器这种移动的目标而发射时，显然就行不通了。如果场景是角色扮演游戏，当猎物看见追击者出现在自己的未来路径上，只要绕开它就行了。

为了找出追击者和猎物能同时到达的点，你必须考虑他们的相对速度。所以，不仅要知道猎物当前的位置，还要知道猎物目前的速度，也就是猎物的速率和前进方向。这些信息可以预测猎物未来某段时间的位置。预测出的位置就成为追击者前进的目标点，并予以拦截。但追击者必须持续监视猎物的位置和速度，并以自身的状态，更新预测的拦截点。如此下去，面对猎物捉摸不定的策略，追击者就可以随着猎物灵活地更新路线。当然，这是假设了追击者有改变方向的能力。

剩下的问题是，如何挑选出某个预测位置作为拦截点？这个问题相当于，哪个位置是追击者可以在最短时间内到达的。这个答案和两者之间的相对位置和速度有关（译注2）。让我们构思计算方法。

追击者要做的第一步计算，是求他和猎物间的相对速度。我们称为靠拢速度（closing velocity），就是猎物和追击者间的速度向量差：

$$\boldsymbol{V}_{r} = \boldsymbol{V}_{prey} - \boldsymbol{V}_{predator}$$

这里的相对或靠拢速度向量以 $\boldsymbol{V}_{r}$ 来表示。第二步是要计算靠拢距离（range to close），也就是追击者和猎物间的相对距离，相当于两者的当前位置的向量差：

$$\boldsymbol{S}_{r} = \boldsymbol{S}_{prey} - \boldsymbol{S}_{predator}$$

这里的 $\boldsymbol{S}_{r}$ 表示追击者和猎物间的相对或靠拢距离（范围）。现在，有足够的信息，可以轻松计算靠拢时间（time to close）。

靠拢时间是以靠拢速率（即追击者和猎物间的相对移动速率）走完靠拢距离所需的平均时间。其计算公式如下：

$$t_{c} = |\boldsymbol{S}_{r}| \,/\, |\boldsymbol{V}_{r}|$$

靠拢时间 t_{c} 等于靠拢距离向量 $\boldsymbol{S}_{r}$ 的值（长度），除以靠拢速度向量 $\boldsymbol{V}_{r}$ 的值（长度）。

译注2：复习中学物理：速度(velocity) = 位移 / 时间，位移是向量，所以速度也是向量。速率(speed)= 路程 / 时间，全部都是标量。当运动轨迹为直线时，位移大小刚好等于路程，所以速度的数值（magnitude）等于速率。

现在，知道靠拢时间后，就能预测猎物在未来时间 t_c 的所在位置。假设猎物当前位置是 S_{prey}，以 V_{prey} 速度行走。以此速度移动 t_c 时间后的平均移动距离是 V_{prey} 与 t_c 的乘积，所以 t_c 时间后的预测位置 S_t 等于目前位置 Sprey 加上平均移动距离：

$$S_t = S_{prey} + (V_{prey})(t_c)$$

这里的 S_t 就是猎物在未来 t_c 时间内的预测位置。S_t 这个预测位置现在就成为追击者的目标点了。为了拦截，追击者应该朝这个点前进，如同以视线算法朝猎物前进一样。事实上，你需要做的就是在例2-8的视线追逐函数内加几行程序代码，将之转变成拦截函数。例2-9就是这个新的函数。

例2-9：拦截函数

```
void     DoIntercept(void)
{
     Vector   u, v;
     Bool     left = false;
     Bool     right = false;
     Vector   Vr, Sr, St;    // 新增
     Double   tc ;           // 新增

     // 新增
     Vr = Prey.vVelocity - Predator.vVelocity;
     Sr = Prey.vPosition - Predator.vPosition;
     tc = Sr.Magnitude() / Vr.Magnitude();
     St = Prey.vPosition + (Prey.vVelocity * tc);

     // 将Prey.vPosition改成St
     u = VRotate2D(-Predator.fOrientation, (St - Predator.vPosition));

     // 其余部分和视线追逐法的函数一样

     u.Normalize();

     if (u.x <  -_TOL)
          left = true;
     else if (u.x > _TOL)
          right = true;

     Predator.SetThrusters(left, right);
}
```

从例2-9中的程序注释，可看出“视线追逐函数”（例2-8）与“拦截函数”之间的差别，仅在于目标点的改变，前者是以猎物本身为目标，后者是以猎物的未来位置为目标。所以新增的程序代码，基本上只是运用前述公式，从靠拢速度、距离以及靠拢时间，计算出猎物的预测位置，如此而已。

每当游戏循环或物理引擎运行一轮，都应该重新调用此函数，随时修正拦截点及拦截路径。

AIDemo2-2 整合了这个算法的结果，图 2-11 到图 2-14 是两种场景的拦截效果图。

图 2-11 所示的场景是追击者和猎物，分别从画面的左下角和右下角出发。猎物以匀速走向左上角。同时，追击者计算出预测的拦截点，并朝该点走去，同时持续更新预测的拦截点及其行进方向。图中预测的拦截点以一连串的点表示，分布在猎物的前方。最初，拦截点会随着追击者转向猎物而改变，然而，由于猎物以匀速移动，方向确定后，拦截点就固定了。

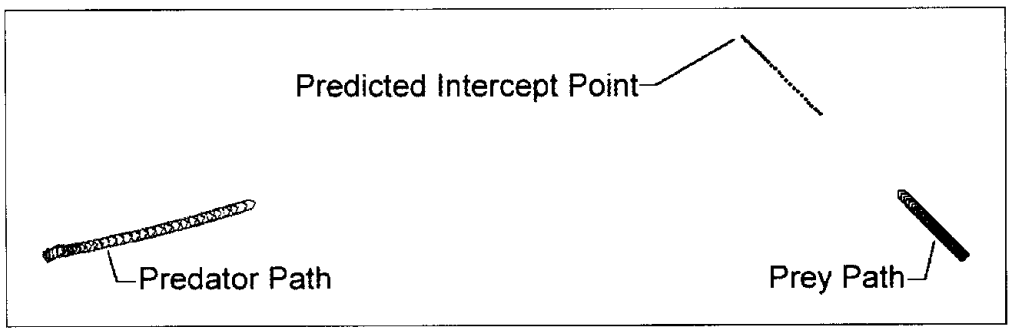


图 2-11：拦截场景 1:最初路径

过了一段时间，追击者就拦截住猎物，如图 2-12 所示。

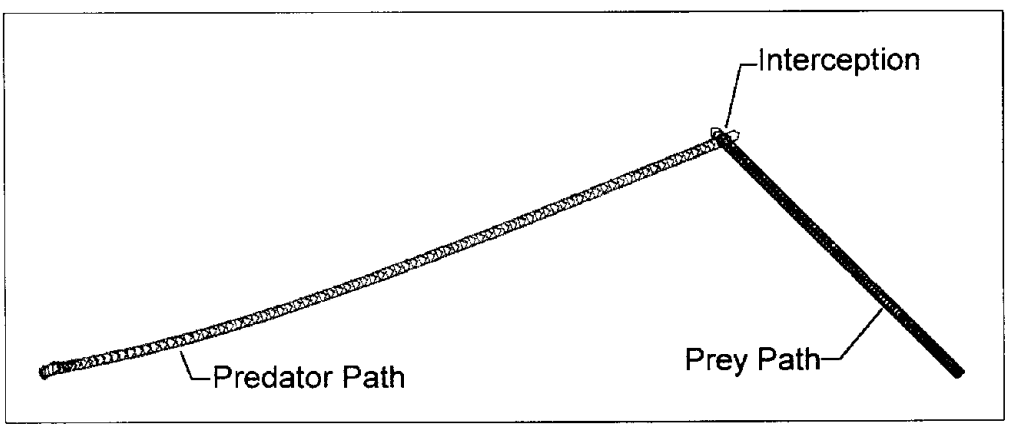


图 2-12：拦截场景 1:拦截

注意，追击者使用拦截算法，和图 2-9 中使用视线算法，所采取的路径的差别。显然，这种算法会得出比较短的路径，实际上，可以让追击者和猎物，在相同时间和空间中的同一点碰头。在视线算法中，追击者会绕圈子追着猎物跑，最后跟在猎物后面。如果追击者速度没有保持足够快，永远也追不到猎物，说不定还会被甩开。

如图 2-13 所示，猎物采取某些难以捉摸的策略时，这个算法依然有效。

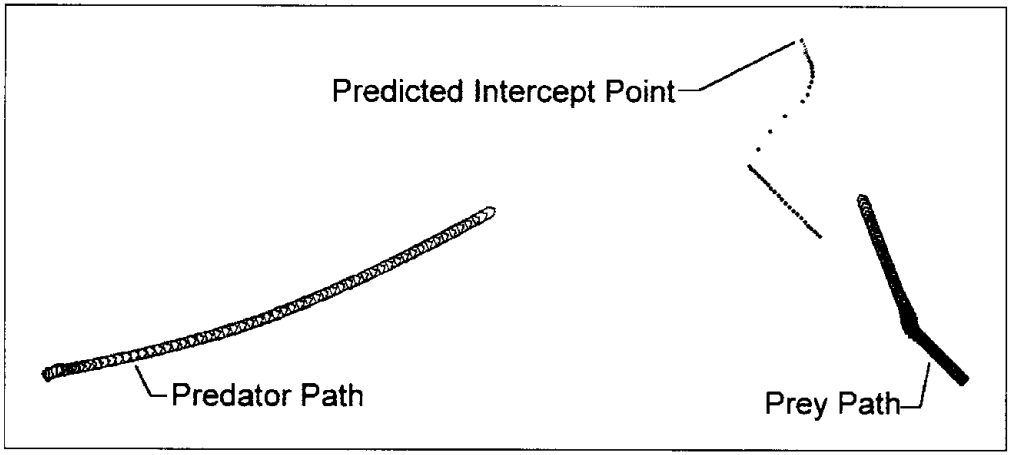


图 2-13：拦截场景 2:修正后的行动

这里你可以看到最初的预测拦截点，如猎物左前方的点所示，和图 2-11 中的相同。然而，当猎物开始往右边走时，预测的拦截点立刻被修正，而追击者也采取更新后的行动，往新的拦截点行进。图 2-14 是最终的拦截结果。

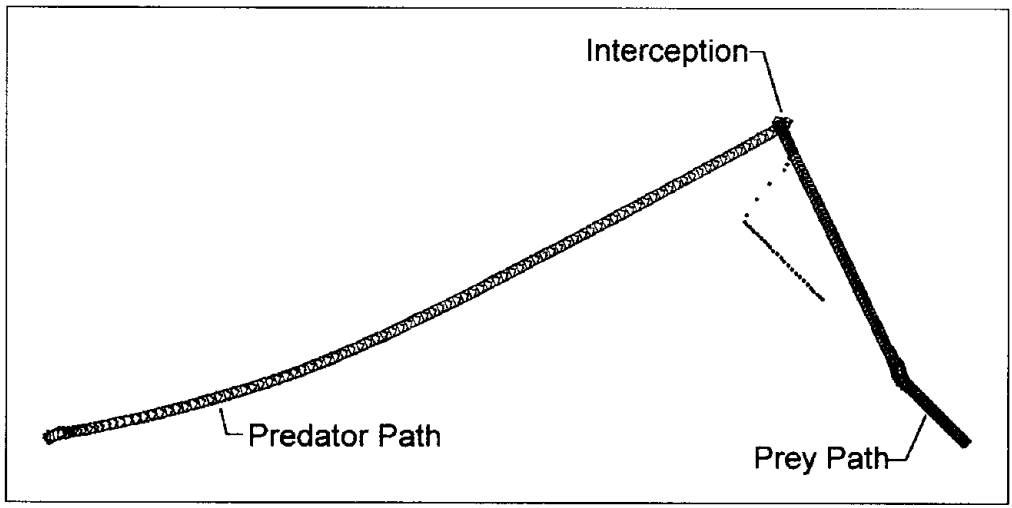


图 2-14：拦截场景 2:拦截

这里讨论的拦截算法非常经得起考验，可以让追击者不断地更新其路径，以有效地拦截。试验过例程序后，你会发现拦截几乎总能成功。

有时候拦截是不可能的，然而，只要你修改我们讨论的算法，就可以处理这些情况了。例如，如果追击者的速度比猎物慢，或是追击者的位置远远落在猎物之后，就无法拦截。追击者永远无法追上猎物或跑到前面去拦截，除非猎物耍什么花招，让追击者不再落在后面。即便如此，也可能因为靠近的程度不一，使得追击者未能以足够快的速度去拦截。

再如，如果追击者以相同或快于猎物的速度，跑到猎物的前面；则预测拦截点会持续落在两者的前头，如此一来，拦截永远不会发生，因为拦截点会不断远离他们。在这种情况下，最佳的做法是侦测出追击者何时会在猎物前方，然后让追击者绕圈子，或采取其他行动，以较佳角度拦截猎物。你可以侦测猎物是否位于追击者后面，只要在追击者的局部坐标系统，检查出猎物相对于追击者的位置即可，类似于例2-8和2-9的做法。此时检查的不是x坐标，而是y坐标，如果是负值，则猎物就是位于追击者之后，此时追击者需要转向。让追击者转向最简单的方法，就是改用视线算法，暂时不用拦截算法。这样将让追击者转至反向，再度对准猎物，此时再使用拦截算法予以拦截。

上文我们提到追逐和闪躲牵涉到两个不同的问题（或三个）：决定要追或逃，实际追或逃，以及如何避开障碍物。本章我们从几个不同的角度，讨论了第二个问题，实际追或逃。其中包括在砖块环境和连续环境中的基本追逐、视线追逐以及拦截。这些方法都已被证实，令人产生角色具有智能的错觉。而且，这些方法还可以结合其他算法，共同处理尚待解决的问题，比如，判断要追或逃以及何时开始、或是在追或逃时，如何避开障碍物等，以提升人工智能的效果。接下来几章要探讨这些算法。

此外，注意到追逐或闪躲还可以用其他算法。其中之一是利用势函数，在第五章说明。

第三章

移动模式

本章主题是移动模式（Pattern Movement）。移动模式是制造智能行为幻觉的简单方式。基本上，计算机控制的角色会根据一些预先定义好的模式移动，使其看起来好像在执行复杂而绞尽脑汁的策略。如果你较年长，还记得以前街机游戏“Galaga”，马上就会知道移动模式是怎么回事。回想一下那些外星人的战机从屏幕上方和两侧猛扑过来，既返回又转弯，同时还对着你的战机开炮。这些外星人有不同的类型，根据你所在那一关的情况会采取不同的攻击策略。这些手段就是利用移动模式算法做出来的。

虽然“Galaga”是使用移动模式的经典案例，即使是现代的视频游戏也会用到某些移动模式。例如，在角色扮演游戏或第一人称射击游戏中，敌方怪兽也许根据某些预先定义好的模式巡逻游戏世界中的某些地区。在战争模拟游戏中，敌方战机也许根据某些预先定义好的模式而执行令人难以捉摸的策略。各种不同类型的次要生物和非玩家角色会被设计成按某些预先定义好的模式移动，让人一看便知他们正在闲逛、进食或执行某些任务。

实现移动模式的标准做法是选取想要的模式，再将控制数据填入某个数组或多个数组。控制数据由特定的移动指令组成，比如向前移动再转弯，借此迫使计算机控制的物体或角色按所需模式移动。利用这些算法，你可以建立圆形、方形、蛇行、曲线以及任何类型的模式，将之编制成一组精确的移动指令。

本章将在一般条件下检视标准移动模式的算法，然后再举两个实例，来实现标准算法的变化形式。第一个实例在砖块环境中实现，而第二个实例展示如何在仿真物理环境中，实现移动模式，在这个环境中设计游戏时，应该考虑到一些需要特别注意之处。

标准算法

标准移动模式算法使用控制指令（编码过的指令清单或数组），指示计算机控制的角色，在每一轮游戏循环中如何移动。每当循环运行一轮时，数组将编入索引值，以便处理下一组新的移动指令。

例 3-1 给出了一组典型的控制指令。

例 3-1：控制指令数据结构

```
ControlData {
   double turnRight;
   double turnLeft;
   double stepForward;
   double stepBackward;
   };
```

此例中，turnRight 和 turnLeft 中存放的是右转或左转的角度值。如果是在砖块环境中，角色能够前进的方向有限，则 turnRight 和 turnLeft 的意义就是向右或向左转一格。stepForward 和 stepBackward 是向前或向后的距离或者是砖块数。

这个控制结构也可以包含其他指令，比如开火、丢炸弹、放出干扰雷达的金属片、什么也不做、加速、减速，以及其他许多适合你的游戏的行为。

通常，你可以事先定义出控制结构体类型的全局数组或者一组数组，以便储存模式数据。设定这些模式数组初值的数据，可以从数据文件加载，或直接编写在游戏程序中，这与你的编码风格以及游戏所需的条件有关。

直接在游戏程序中初始化模式数组，有可能像例 3-2 那样。

例 3-2：模式的初始化

```
Pattern[0].turnRight = 0;
Pattern[0].turnLeft = 0;
Pattern[0].stepForward = 2;
Pattern[0].stepBackward = 0;

Pattern[1].turnRight = 0;
Pattern[1].turnLeft = 0;
Pattern[1].stepForward = 2;
Pattern[1].stepBackward = 0;

Pattern[2].turnRight = 10;
Pattern[2].turnLeft = 0;
Pattern[2].stepForward = 0;
Pattern[2].stepBackward = 0;
```

```
Pattern[3].turnRight = 10;
Pattern[3].turnLeft = 0;
Pattern[3].stepForward = 0;
Pattern[3].stepBackward = 0;

Pattern[4].turnRight = 0;
Pattern[4].turnLeft = 0;
Pattern[4].stepForward = 2;
Pattern[4].stepBackward = 0;

Pattern[5].turnRight = 0;
Pattern[5].turnLeft = 0;
Pattern[5].stepForward = 2;
Pattern[5].stepBackward = 0;

Pattern[6].turnRight = 0;
Pattern[6].turnLeft = 10;
Pattern[6].stepForward = 0;
Pattern[6].stepBackward = 0;
            .
            .
            .
```

此例中，这个模式会指示计算机控制的角色向前移2个单位的距离，再向前移2个单位的距离，向右转10度，再向右转10度，向前移2个单位的距离，再向前移2个单位的距离，然后向左转10度。这个特定的模式会让计算机控制的角色，以蛇行模式（zigzag pattern）前进。

为了处理这个模式，必须持有一个该模式数组的索引值，每次游戏循环运行一轮时，就递增一次。并且，在每次循环中，都必须予以读取，并执行该模式数组当前索引值对应的控制指令。例3-3是这些步骤在程序中的概略写法。

例3-3：运行模式数组

```
void     GameLoop(void)
{
     .
     .
     .

     Object.orientation + = Pattern[CurrentIndex].turnRight;
     Object.orientation - = Pattern[CurrentIndex].turnLeft;
     Object.x + = Pattern[CurrentIndex].stepForward;
     Object.x - = Pattern[CurrentIndex].stepBackward;

     CurrentIndex++;

     .
     .
     .
}
```

正如你所见，基本的算法非常简单。当然，操作细节会因游戏结构而有所不同。

编写好几个不同模式，存放在不同数组中也是很常见的做法，然后让计算机随机选取一个模式来使用，或者按照游戏中某些其他决策逻辑来决定。这样的技巧可以强化智能的错觉，让计算机控制的角色行为有更多的变化。

砖块环境中的移动模式

对砖块环境移动模式而言，所要采用的方法，与第二章讨论砖块环境视线追逐时，所用的方法类似。在视线范例中，我们采用Bresenham的线段算法，事先算出起点和终点间的路径。本章也是用Bresenham的线段算法，计算不同的移动模式。如同第二章所讲的，需要把行和列（坐标）的位置储存在一组数组内。然后，以不同的模式移动计算机控制的角色（此例为巨人），走遍这些数组。

本章的路径比只有起点和终点的情况复杂得多。路径将由好几条线段构成。每条新线段的起始处就是前一条线段的终止处。你必须确保最后一条线段的终点，是第一条线段的起点，才能让巨人在周而复始的模式中移动。当巨人处在守卫或巡逻模式时，这种方法特别有用。例如，你可以让巨人在营区周围不断走动，当敌人移动到附近时，停止用这个模式，替换成简单的矩形模式。

你可以算出四条线段，完成这个矩形移动模式。第二章的视线函数，每次执行时会清除坐标的路径数组内容。然而，就此例而言，每条线段只是整个模式的一条线段而已，因此，每次要计算线段时，我们不必对路径数组初始化，只需把新的线段路径添加到前一条线段路径之后。此例中，在计算模式之前，我们要先初始化坐标数组。例3-4是初始化坐标路径数组的函数。

例3-4：*初始化路径数组*

```
void    InitializePathArrays(void)
{
    int i;

    for (i=0;i<kMaxPathLength;i++)
        {
            pathRow[i] = -1;
            pathCol[i] = -1;
        }
}
```

如例3-4所示，我们把两个数组的每个元素都赋值为−1。我们采用−1是因为−1在砖块环境中不是有效的坐标。一般而言，在多数砖块环境中，左上角的坐标是(0,0)。从该点开始，行和列会递增到整个砖块地图的大小。把路径数组中还未用到的元素赋值−1，

是指出该元素在路径数组中尚未用到的简单方法。当一条线段要连接到路径数组中的另一条线段时，就可看出其作用。例3-5是修改后的Bresenham视线算法，用于计算线段。

例3-5：计算线段

```
void ai_Entity::BuildPathSegment(void)

{
    int i;
    int nextCol=col;
    int nextRow=row;
    int deltaRow=endRow-row;
    int deltaCol=endCol-col;
    int stepCol;
    int stepRow;
    int currentStep;
    int fraction;
    int i;

    for (i=0;i<kMaxPathLength;i++)
        if ((pathRow[i]==-1) && (pathCol[i]==1))
            {
                currentStep=i;
                break;
            }

    if (deltaRow < 0) stepRow=-1; else stepRow=1;
    if (deltaCol < 0) stepCol=-1; else stepCol=1;
    deltaRow=abs(deltaRow*2);
    deltaCol=abs(deltaCol*2);

    pathRow[currentStep]=nextRow;
    pathCol[currentStep]=nextCol;
    currentStep++;
    if (currentStep>=kMaxPathLength)
        return;

    if (deltaCol > deltaRow)
        {
            fraction = deltaRow * 2 - deltaCol;
            while (nextCol != endCol)
                {
                    if (fraction >= 0)
                        {
                            nextRow += stepRow;
                            fraction = fraction - deltaCol;
                        }
                    nextCol = nextCol + stepCol;
                    fraction = fraction + deltaRow;
                    pathRow[currentStep]=nextRow;
                    pathCol[currentStep]=nextCol;
                    currentStep++;
                    if (currentStep>=kMaxPathLength)
```

```
                    return;
                }
        }
    else
        {
            fraction = deltaCol * 2 - deltaRow;
            while (nextRow != endRow)
                {
                    if (fraction >= 0)
                        {
                            nextCol = nextCol + stepCol;
                            fraction = fraction - deltaRow;
                        }
                    nextRow = nextRow + stepRow;
                    fraction = fraction + deltaCol;
                    pathRow[currentStep]=nextRow;
                    pathCol[currentStep]=nextCol;
                    currentStep++;
                    if (currentStep>=kMaxPathLength)
                        return;
                }
        }
}
```

大致上，这个算法与第二章例2-7的视线移动算法很类似。主要的差别是把初始化路径数组的程序代码，换成了一段新的程序代码。此例中，要让每条新线段都附加到前一条线段之后，所以每次这个函数被调用时，不必初始化路径数组。新加的一段程序代码是要判断在哪里把线段加上去。这就是用 -1 初始化路径数组的地方。你需要做的就是走过数组，找到第一个碰到数值为 -1 之处，这里就是新线段的起点。利用例3-6的函数，现在就能计算出第一个模式。我们打算采用简单的矩形巡逻模式。图3-1是我们所要的模式。

如图3-1所示，我们把矩形模式的四个角标示出来，再标出想要的移动方向。利用这些信息，就能用例3-5的 BuildPathSegment() 函数建立巨人的模式。例3-6是初始化矩形模式的内容。

例3-6：矩形模式

```
entityList[1].InitializePathArrays();
entityList[1].BuildPathSegment(10, 3, 18, 3);
entityList[1].BuildPathSegment(18, 3, 18, 12);
entityList[1].BuildPathSegment(18, 12, 10, 12);
entityList[1].BuildPathSegment(10, 12, 10, 3);
entityList[1].NormalizePattern();
entityList[1].patternRowOffset = 5;
entityList[1].patternColOffset = 2;
```

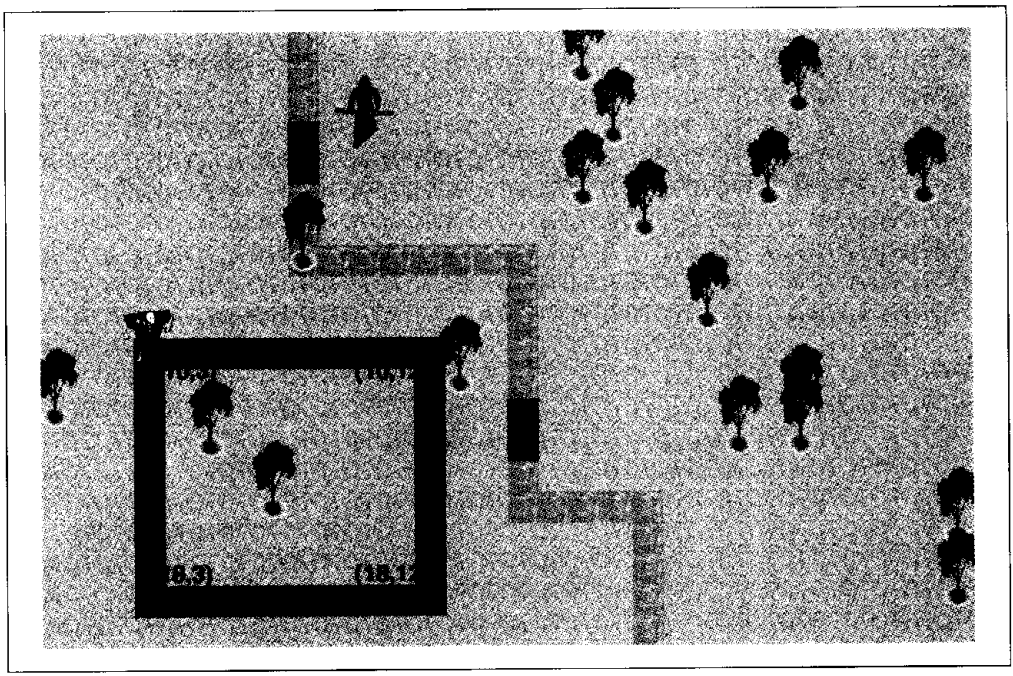


图3-1：矩形移动模式

如例3-6所示，首先调用InitializePathArrays()函数初始化路径数组。然后，利用图3-1所示的坐标，计算出四条组成矩形模式的线段。当每条线段都算出，而且储存在路径数组后，就调用NormalizePattern()将模式标准化，使其以相对坐标表示，而非绝对坐标。这样做，标准化的模式才不会在游戏领域里，和特定的起点位置绑在一起。一旦把模式建立起来，并标准化后，就能在任何游戏当中使用。例3-7是NormalizePattern()函数。

例3-7：标准化函数

```
void ai_Entity::NormalizePattern(void)
{
    int i;
    int rowOrigin=pathRow[0];
    int colOrigin=pathCol[0];
    for (i=0;i<kMaxPathLength;i++)
        if ((pathRow[i]==-1) && (pathCol[i]==-1))
              {
                 pathSize=i-1;
                 break;
               }

    for (i=0;i<pathSize;i++)
         {
```

```
            pathRow[i]=pathRow[i]-rowOrigin;
            pathCol[i]=pathCol[i]-colOrigin;
        }
}
```

如例3-7所示，要将模式标准化只需把储存在模式数组中的所有位置都减去起始位置即可。以相对坐标做出的模式，就能在任何游戏领域使用了。

现在，模式已经建立起来，我们可以通过数组，让巨人以矩形模式行走。注意到最终线段的最后两个坐标，就是最初线段最先的两个坐标。这就确保了巨人重复行走矩形模式。

你可以利用BuildPathSegment()函数建立任何数目的模式。只需求出所需模式的顶点坐标，然后计算出每条线段。当然，也可以只用最少的两条线段，如果程序资源许可的话，也可以用很多线段建立移动模式。例3-8是利用两条线段，建立起简单的来回巡逻模式。

例3-8：简单巡逻模式

```
entityList[1].InitializePathArrays();
entityList[1].BuildPathSegment(10, 3, 18, 3);
entityList[1].BuildPathSegment(18, 3, 10, 3);
entityList[1].NormalizePattern();
entityList[1].patternRowOffset = 5;
entityList[1].patternColOffset = 2;
```

利用例3-8的线段，巨人只会在坐标(10,3)和(18,3)之间来回走动。对那些在城堡大门前巡逻，或者保护桥梁附近区域的任务而言，这种模式就很有用。巨人只会一直重复使用这个模式，直到敌人出现在视线内，就切换到追逐或攻击状态。

当然，要用多少条线段制作移动模式并没有限制。像巡逻城堡周围，或在海岸线上不断行进以防守入侵者这类任务，就可以用大而复杂的模式。例3-9是一个比较复杂的模式。该范例建立了一个由八条线段组成的模式。

例3-9：复杂巡逻模式

```
entityList[1].BuildPathSegment(4, 2, 4, 11);
entityList[1].BuildPathSegment(4, 11, 2, 24);
entityList[1].BuildPathSegment(2, 24, 13, 27);
entityList[1].BuildPathSegment(13, 27, 16, 24);
entityList[1].BuildPathSegment(16, 24, 13, 17);
entityList[1].BuildPathSegment(13, 17, 13, 13);
entityList[1].BuildPathSegment(13, 13, 17, 5);
entityList[1].BuildPathSegment(17, 5, 4, 2);
entityList[1].NormalizePattern();
entityList[1].patternRowOffset = 5;
entityList[1].patternColOffset = 2;
```

例3-9建立了一个复杂的模式，把地形元素也考虑进来了。巨人从河的西岸开始走，跨越北边的桥、巡逻到南方、跨越南方的桥，然后再回到北方的起始点。接着巨人会重复这个模式。图3-2展示了这个模式，还标示了建构这个模式的顶点。

如图3-2所示，这个移动模式的做法，可以做出很长很复杂的模式。当你在建立很长的巡逻范围，绕过许多地形元素时，这种做法特别有用。

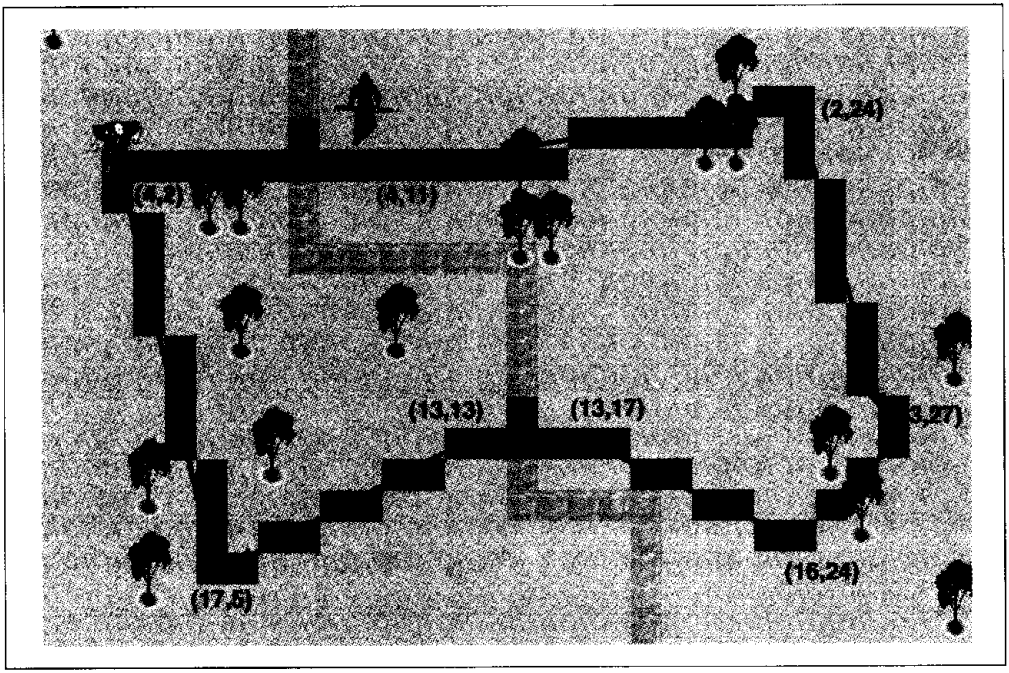


图3-2：杂乱的砖块移动模式

虽然图3-2所采用的模式方法，可以做出既长又复杂的模式，但这些模式看起来既冗长，又容易被玩家猜透。下一个要介绍的方法将在此移动模式中，加入随机因素（random factor）。在砖块环境中，游戏世界通常由二维数组表示。数组中的元素，指出每个行和列坐标中放了什么东西。下一个移动模式的做法，将用到第二个二维数组。这个模式矩阵会引导巨人沿着预先定义的路径行走。模式数组中的每个元素的值不是0就是1。只有在模式数组中元素的值为1时，巨人才可以移到相对应的行和列坐标。

实现这类移动模式首先要做的是，建立一个模式矩阵。如例3-10所示，一开始将模式矩阵的初始值全都赋为0。

例 3-10：模式矩阵的初始化

```
for (i=0;i<kMaxRows;i++)
     for (j=0;j<kMaxCols;j++)
          pattern[i][j]=0;
```

整个模式矩阵都赋成 0 之后，就可以把想要的移动模式坐标设为 1 了。在这里将使用 Bresenham 视线法的另一种算法，建立另一种模式。但是，并不把行、列坐标储存在路径数组中。而是沿着线段，把模式矩阵中相对应的行、列坐标设为 1。然后，就可以多次调用 BuildPatternSegment() 函数，建立复杂的模式。例 3-11 是建立这个模式的程序。

例 3-11：建立模式

```
BuildPatternSegment(3, 2, 16, 2);
BuildPatternSegment(16, 2, 16, 11);
BuildPatternSegment(16, 11, 9, 11);
BuildPatternSegment(9, 11, 9, 2);
BuildPatternSegment(9, 2, 3, 6);
BuildPatternSegment(3, 6, 3, 2);
```

每次调用 BuildPatternSegment() 函数时，该函数都会使用 Bresenham 线段算法，在模式矩阵中建立新线段。函数参数中的前两个是起点的行、列坐标，而后两个是终点的行、列坐标。线段中的每个点，在模式矩阵中都成为 1。这个模式以图 3-3 来说明。

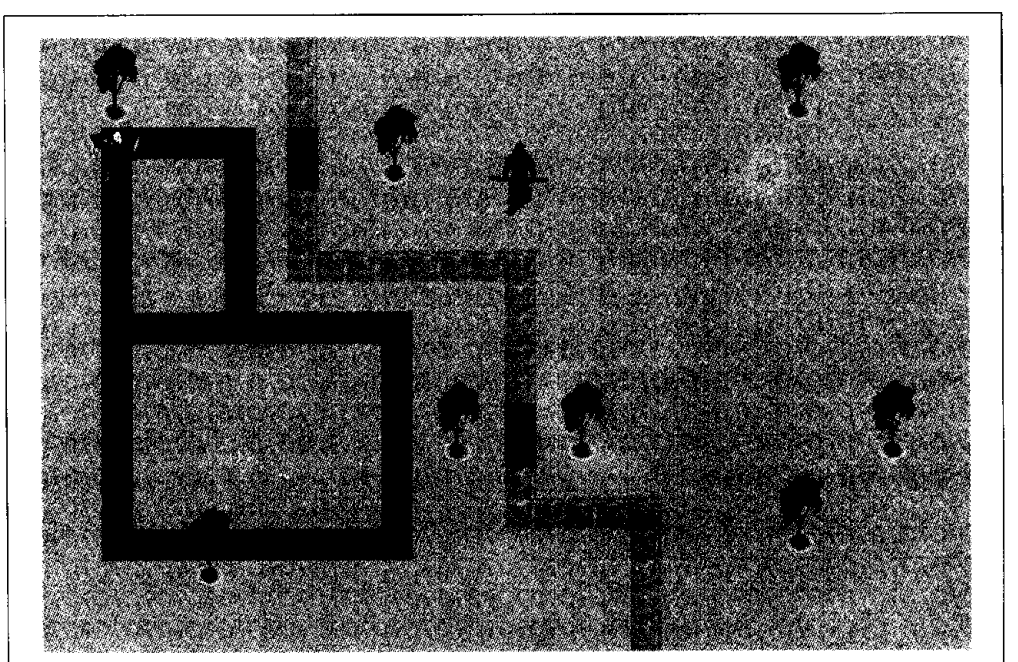


图 3-3：行进路线移动模式

图3-3标示出模式矩阵中每个含有1的点。这些点是巨人可以行走之处。但是请注意，巨人在行进路线的点上，现在有很多有效的方向可以走。因此，巨人就不会重复来回走动，也不容易被预测。

每次更新巨人的位置时，需要检查模式数组周围的八个元素，找出有效的移动点，如例3-12所示。

例3-12：沿着模式矩阵走

```
void ai_Entity::FollowPattern(void)
{
    int i,j;
    int possibleRowPath[8]={0,0,0,0,0,0,0,0};
    int possibleColPath[8]={0,0,0,0,0,0,0,0};
    int rowOffset[8]={-1,-1,-1, 0, 0, 1, 1, 1};
    int colOffset[8]={-1, 0, 1,-1, 1,-1, 0, 1};

    j=0;
    for (i=0;i<8;i++)
         if (pattern[row+rowOffset[i]][col+colOffset[i]]==1)
              if (!(((row+rowOffset[i])==previousRow) &&
                   ((col+colOffset[i])==previousCol)))
                   {
                       possibleRowPath[j]=row+rowOffset[i];
                       possibleColPath[j]=col+colOffset[i];
                       j++;
                   }

    i=Rnd(0,j-1);
    previousRow=row;
    previousCol=col;

    row=possibleRowPath[i];
    col=possibleColPath[i];
```

一开始是检查模式矩阵中巨人当前位置周围的八个点。当发现其值为1时，就把该坐标储存到possibleRowPath和possibleColPath数组中。每个点都检查过后，可以从找到有效点的数组中，随机选取新的坐标点。最后的结果，就是每次巨人抵达模式矩阵中的顶点时，不会每次都转向同一个方向。

我们注意到，例3-12的rowOffset和colOffset这两个变量的用途，是为了避免写八个条件语句。利用一个循环，把这些值加到行和列的位置，就可以走遍那八个相邻的砖块。例如，头两个元素加到当前的行和列位置之后，就是当前位置的左上角砖块。

移动巨人时还要考虑到，巨人先前的位置，也位于有效移动数组中。如果更新巨人位置时选了那个点，就会造成意外的来回移动。因此，一定要利用previousRow和previousCol变量，追踪巨人先前的位置，然后，建立有效移动数组时，就能将该点排除在外。

仿真物理环境中的移动模式

到目前为止，本章所讨论的都是指示游戏角色，在以不连续的方式前进或转弯的环境中实现移动模式。但是，对仿真物理环境而言又怎样呢？当然，你也可以在仿真物理环境中利用移动模式的优点。问题是使用仿真物理环境的优点（也就是让物理机制控制行动），无法使仿真物理环境的物体，按照特定模式移动。例如，将仿真物理环境的飞行器，在每次游戏循环运行一轮时，指定到特定的位置或方位，就失去了仿真物理机制的目的。仿真物理环境中的仿真过程，不需明确指定物体的位置。所以，为了在这种情况下实现移动模式，必须修改前面提到的算法。具体地说，此时并非在每轮游戏循环中，用模式设定物体的位置和方位，而是，必须运用适当的控制力引导物体，或从实质上加以驱动，使物体到达你想让它去的地方。

第二章的第二个实例介绍了，利用物理驱动力让追击者追逐猎物的方法。你也可以用相同的驱动力驱动仿真物理的物体，使其遵照某些模式行动。事实上，这几乎就是仿真一个智能生物，并让其操纵仿真物理的载具。从本质上讲，运用控制力，就是模拟驾驶员以某种模式驾驶他的载具。只要运用适宜的控制力，比如推力和转向力，即可做出各种形式的模式，例如捉摸不定的策略，巡逻路径，各种花招等。

然而，记住一点，这种情况下你没有绝对的控制权。控制仿真中的物体的各项能力（比如速率和旋转半径等）的物理引擎和模型，也会控制该物体的整体行为。例如，你的输入数据是推力和转向力的校正值，然后由物理引擎处理，而最后的行为则是所有输入到物理引擎数据的函数，而不是你的函数。让物理引擎保持控制权，我们就可以让计算机控制的物体，有某种程度的智能，而不会在该物理模型中，强迫物体做一些不可能发生的事。如果你想违反物理模型，就要冒着破坏真实物理经验世界的风险。记住，我们的目标是要具有智能，以强化玩家的注意力。

为了说明如何在仿真物理环境中实现路径移动模式，我们打算以第二章提过的在连续环境中，追逐和拦截的场景作为基础。回想一下那个场景，里面有两台载具，我们以简单的平面刚体模拟仿真运算。计算机控制一台载具，而玩家控制另一台。本章要修改那个范例，也就是示范程序AIDemo2-2，让计算机控制的载具，能按预先定义好的模式移动。修改后的示范程序叫 AIDemo3-2 所示，你可从本书网站下载。

此例所采取的方法非常类似先前讨论过的算法。我们要以一个数组储存模式信息，再绕着这个数组走，把适当的模式指令交给计算机控制的载具。我们先前讨论过的算法和这个要用在仿真物理环境的算法，有某些关键性的差异。在先前的算法中，模式数组储存的是不同的移动信息，例如，往左一步，向前一步，右转，左转等。此外，每次运行一

轮游戏循环时，模式数组索引值都会往前递增，使得下一次的移动指令可以取出来。在仿真物理环境中，你必须采取不同的方式，下一节会详谈。

控制结构

如前所述，在仿真物理环境中，你无法强迫计算机控制的载具，分别向前或向后走一步。你也无法明确告知令其左转或右转。相反地，你必须为物理引擎提供控制力信息，让计算机控制的载具，可以依照你想要的模式实际运行。此外，当控制力（如转向力）施加在仿真物理环境中时，并不会立即改变仿真中物体的运动状态。这些控制力需要一段时间才能引发所要的运动。也就是说，模式数组索引值和游戏循环之间不会同步；其实你也不希望这种事发生。如果你有一个模式数组，每轮仿真运算时，都要执行其中所含的特定指令，那么这个模式数组将变得非常大，因为仿真物理环境的仿真运算时间，间隔是非常短的。

为了避开这个问题，我们这里所用的模式数组所含的控制数据，也包含了其他信息，让计算机知道每组指令应该执行多长时间。算法的运行是这样的：计算机从模式数组中选取第一组指令，然后施加到控制中的载具。每轮仿真运算时，物理引擎会处理这些指令，直到该组指令中指定的条件满足为止。此时，模式数组中的下一组指令就会被选出并执行。这个过程将一直重复，直到模式数组走完，或者模式因某种原因而中断为止。

例3-13的程序代码是我们为此例设定的模式控制数据结构。

例3-13：*移动模式控制数据结构体*

```
struct    ControlData {
    bool      PThrusterActive;
    bool      SThrusterActive;
    double    dHeadingLimit;
    double    dPositionLimit;
    bool      LimitHeadingChange;
    bool      LimitPositionChange;
};
```

注意一点，控制数据会因为你的游戏中将要模拟的东西，及其底层物理模型如何运作，而有所改变。此例中，我们控制的载具方位，仅由两侧的推进器施加的力控制。因此，我们在配置时，只需考虑两种控制力，据此实现某种移动模式。

因此，例3-13的数据结构体，包含了两个布尔成员变量`PThrusterActive`和`SThrusterActive`，指的就是哪个推进器应该启动。接下来两个成员变量是`dHeadingLimit`和`dPositionLimit`，用于决定每组控制数据应该实施多久。例如，`dHeadingLimit`指定载具方向的改变。如果你想执行特别指令，要载具转45度角，就

把dHeadingLimit设成45。注意到，转向是相对角度，不是绝对方向。如果标号LimitHeadingChange设为true，则每轮仿真循环中，执行指定模式指令时，也会检查dHeadingLimit的值。指令继续运行前，如果载具的方向和上次方向相比已经转好了，就应该把下一条指令取出来。

dPositionLimit的意义也雷同。这个成员变量储存的是所要的位置改变，也就是这组指令运行前，相对于载具位置所要移动的距离。如果LimitPositionChange设为true，则每轮仿真循环中，载具的相对位置改变会以dPositionChange来作比较，以决定是否要从模式数组中取出下一组指令。

继续深入讨论之前，我们要强调一点，我们在这里讲的移动模式算法，是指方向和位置的相对变化。模式指令是类似往前100英尺，然后向左转45度，再向前走100英尺，然后再向右转45度，诸如此类的命令。但指令是绝对的：向前移动直到你抵达位置(x,y)0，然后转弯，直到你面向东南方，然后再予以移动，直到你抵达位置(x,y)1，然后转弯，直到你面向西南方，等等。

在位置和方向上使用相对改变，可以让你不用考虑被控制物体的位置或最初方向，就能执行已储存的模式。如果你用绝对坐标和罗盘方向，你用的模式就会被限制在那些坐标附近。例如，你可以使用某种模式巡逻地图上的特定区域，但是，以这种特定用途的模式，就无法巡逻地图上的任何区域。后一种采用绝对坐标的做法，和我们先前在砖块实例中采用的算法一致。此外，这种做法和航点导航（waypoint navigation）一致，有其自身的优点，后续各章会讨论。

由于我们这里的位置和方向的改变采用相对值，你也需要采用某种记录这些改变的方式，才能从一组模式指令中带到下一组。最后，我们定义了另一个结构体，储存了载具从一组模式指令到下一组指令其状态的改变。例3-14就是这个结构体。

例3-14：记录状态改变的结构体

```
struct    StateChangeData {
    Vector      InitialHeading;
    Vector      InitialPosition;
    double      dHeading;
    double      dPosition;
    int         CurrentControlID;
};
```

前两个成员变量InitialHeading和InitialPosition是向量，储存了一组模式指令——从模式数组中选出，当前被控制载具的方向和位置。每次模式数组的索引值向前进时，就会取出新一组指令，而这两个成员变量就必须更新。接下来两个成员变量dHeading和dPosition储存了仿真运算过程中，当前这种模式指令实施时，方向和位

置的变化。最后，CurrentControlID储存模式数组中当前的索引值，指出当前正在执行的模式控制指令是哪一组。

定义模式

现在，要定义某些模式，你必须把可以获得所需移动模式的适当转向力控制指令，填满一个ControlData结构体类型的数组。就此例而言，我们设立了三个模式。第一个是方形模式，第二个是蛇行模式。在实际的游戏中，你可以利用方形模式，让载具巡逻一个围成方形的区域。你可以用蛇行模式让载具采用捉摸不定的策略，比如海军军舰采用蛇行路径通过海洋，让敌军潜水艇很难用鱼雷攻击。任何你想要仿真的模式都可以定义出控制的输入资料，可以利用这个方法定义圆形、三角形、或任何路径。事实上，此例所引入的第三个模式就是任意形状的模式。

对于方形和蛇行模式而言，我们建立了两个全局数组，名叫PatrolPattern和ZigZagPattern，如例3-15所示。

例3-15：模式数组声明

```
#define_PATROL_ARRAY_SIZE          8
#define _ZIGZAG_ARRAY_SIZE          4

ControlData          PatrolPattern[_PATROL_ARRAY_SIZE];
ControlData          ZigZagPattern[_ZIGZAG_ARRAY_SIZE];

StateChangeData      PatternTracking;
```

正如你所见，我们也定义了一个全局变量，名叫PatternTracking，用于记录这些模式执行时在位置和方向上的改变。

例3-16和3-17表示了这两个模式以适当控制数据做初始化。我们直接在范例中给模式赋初值，然而，在实际游戏中，你也许更愿从数据文件中加载模式数据。此外，你可以利用更精确的编码方式优化数据结构，我们在这里使用的结构只是为了清楚说明问题而已。

例3-16：方形巡逻模式初始化

```
PatrolPattern[0].LimitPositionChange = true;
PatrolPattern[0].LimitHeadingChange = false;
PatrolPattern[0].dHeadingLimit = 0;
PatrolPattern[0].dPositionLimit = 200;
PatrolPattern[0].PThrusterActive = false;
PatrolPattern[0].SThrusterActive = false;

PatrolPattern[1].LimitPositionChange = false;
PatrolPattern[1].LimitHeadingChange = true;
```

```
PatrolPattern[1].dHeadingLimit = 90;
PatrolPattern[1].dPositionLimit = 0;
PatrolPattern[1].PThrusterActive = true;
PatrolPattern[1].SThrusterActive = false;

PatrolPattern[2].LimitPositionChange = true;
PatrolPattern[2].LimitHeadingChange = false;
PatrolPattern[2].dHeadingLimit = 0;
PatrolPattern[2].dPositionLimit = 200;
PatrolPattern[2].PThrusterActive = false;
PatrolPattern[2].SThrusterActive = false;

PatrolPattern[3].LimitPositionChange = false;
PatrolPattern[3].LimitHeadingChange = true;
PatrolPattern[3].dHeadingLimit = 90;
PatrolPattern[3].dPositionLimit = 0;
PatrolPattern[3].PThrusterActive = true;
PatrolPattern[3].SThrusterActive = false;

PatrolPattern[4].LimitPositionChange = true;
PatrolPattern[4].LimitHeadingChange = false;
PatrolPattern[4].dHeadingLimit = 0;
PatrolPattern[4].dPositionLimit = 200;
PatrolPattern[4].PThrusterActive = false;
PatrolPattern[4].SThrusterActive = false;

PatrolPattern[5].LimitPositionChange = false;
PatrolPattern[5].LimitHeadingChange = true;
PatrolPattern[5].dHeadingLimit = 90;
PatrolPattern[5].dPositionLimit = 0;
PatrolPattern[5].PThrusterActive = true;
PatrolPattern[5].SThrusterActive = false;

PatrolPattern[6].LimitPositionChange = true;
PatrolPattern[6].LimitHeadingChange = false;
PatrolPattern[6].dHeadingLimit = 0;
PatrolPattern[6].dPositionLimit = 200;
PatrolPattern[6].PThrusterActive = false;
PatrolPattern[6].SThrusterActive = false;

PatrolPattern[7].LimitPositionChange = false;
PatrolPattern[7].LimitHeadingChange = true;
PatrolPattern[7].dHeadingLimit = 90;
PatrolPattern[7].dPositionLimit = 0;
PatrolPattern[7].PThrusterActive = true;
PatrolPattern[7].SThrusterActive = false;
```

例 3-17：蛇行模式初始化

```
ZigZagPattern[0].LimitPositionChange = true;
ZigZagPattern[0].LimitHeadingChange = false;
ZigZagPattern[0].dHeadingLimit = 0;
ZigZagPattern[0].dPositionLimit = 100;
ZigZagPattern[0].PThrusterActive = false;
ZigZagPattern[0].SThrusterActive = false;
```

```
ZigZagPattern[1].LimitPositionChange = false;
ZigZagPattern[1].LimitHeadingChange = true;
ZigZagPattern[1].dHeadingLimit = 60;
ZigZagPattern[1].dPositionLimit = 0;
ZigZagPattern[1].PThrusterActive = true;
ZigZagPattern[1].SThrusterActive = false;

ZigZagPattern[2].LimitPositionChange = true;
ZigZagPattern[2].LimitHeadingChange = false;
ZigZagPattern[2].dHeadingLimit = 0;
ZigZagPattern[2].dPositionLimit = 100;
ZigZagPattern[2].PThrusterActive = false;
ZigZagPattern[2].SThrusterActive = false;

ZigZagPattern[3].LimitPositionChange = false;
ZigZagPattern[3].LimitHeadingChange = true;
ZigZagPattern[3].dHeadingLimit = 60;
ZigZagPattern[3].dPositionLimit = 0;
ZigZagPattern[3].PThrusterActive = false;
ZigZagPattern[3].SThrusterActive = true;
```

方形模式控制用的输入数据相当简单。第一组指令对应数组元素 PatrolPattern[0]，通知载具向前移200个单位的距离。此例中，没有施加转向力。注意，载具的向前推力已经启动，而且维持常量。你也可以在控制结构中引入该推力，包含转向力和速率变化，让模式更复杂。

下一组模式指令是数组元素 PatrolPattern [1]，通知载具启动左侧推进器右转，直到载具的方向已转过90度。元素 PatrolPattern [2]的指令和元素 PatrolPattern [0]的相同，都告诉载具持续往前走200个单位距离。剩下的元素只是重复前面三个元素而已——元素 PatrolPattern [3]再转90度，元素 PatrolPattern [4]再向前走200个单位距离，以此类推。最后就是数组中有八组指令，让载具在方形模式下运行。

实际上，你可以只留两组指令，也就是例3-16的前两组，依然可以达到方形模式的效果。唯一的差别是你得重复执行四次这两组指令，以形成方形路径。

蛇行控制数据和方形控制数据类似，载具首先往前移一点，然后转弯，再往前移一点，然后再转弯。只是，这次的转弯是从右到左，而且转弯的角度是限制在60度，而不是90度。最后的结果就是载具以蛇行模式移动。

执行模式

此例中，我们在 Initialize()函数中对模式做初始化，当程序启动时，这个函数就会被调用。在这个函数内，我们也会调用名为 InitializePatternTracking()的函数，对 PatternTracking 结构做初始化，如例3-18所示。

例 3-18：InitializePatternTracking()函数

```
void InitializePatternTracking(void)
{
     PatternTracking.CurrentControlID = 0;
     PatternTracking.dPosition = 0;
     PatternTracking.dHeading = 0;

     PatternTracking.InitialPosition = Craft2.vPosition;
     PatternTracking.InitialHeading = Craft2.vVelocity;
     PatternTracking.InitialHeading.Normalize();
}
```

无论何时，调用InitializePatternTracking()函数之后，该函数就会把计算机控制的载具Craft2的当前位置和速度向量复制一份，储存在这个状态改变数据结构中。CurrentControlID是指定模式数组中，当前的元素索引值，此时指定为0，表示第一个元素。此外，位置和方向的最初修正值都先指定为0。

当然，如果你没有一个函数实际处理这些指令，什么事都不会发生。所以，接下来，我们又定义了一个DoPattern()函数，所带的参数是指向模式数组的指针，以及该数组中的元素数目。每次运行一轮仿真运算循环时，这个函数都必须被调用，实施模式控制数据，并走过模式数组。此例中，我们是在UpdateSimulation()函数中调用DoPattern()函数，如例 3-19 所示。

例 3-19：UpdateSimulation()函数

```
void UpdateSimulation(void)
{

     .
     .
     .

     if(Patrol)
     {
          if(!DoPattern(PatrolPattern, _PATROL_ARRAY_SIZE))
               InitializePatternTracking();
     }

     if(ZigZag)
     {
          if(!DoPattern(ZigZagPattern, _ZIGZAG_ARRAY_SIZE))
               InitializePatternTracking();
     }

     .
     .
     .

     Craft2.UpdateBodyEuler(dt);
```

```
    .
    .
    .
}
```

此例中，我们有两个全局变量和两个布尔标号，指出要执行哪个模式。如果Patrol设为true，则使用方形模式；如果ZigZag设为true，则使用蛇行模式。此例中，这些标号彼此互相排斥。

使用这些标号时，如果有必要，可以让你中断某个模式。例如，如果在执行巡逻模式的过程中，其他逻辑运算侦测到敌方载具进入了巡逻区域，你可以把Patrol标号设为false，而把Chase标号设为true。这样可以让计算机控制的载具停止巡逻，开始追击敌人。

物理引擎处理所有作用在载具上的力和力矩之前，DoPattern()函数必须先被调用，否则，模式指令就不会包含在力和力矩的计算中了。此例中，调用Craft2.UpdateBodyEuler (dt)时，就开始计算力和力矩了。

从if语句中可知，DoPattern()返回的是布尔值。如果DoPattern()返回false，就表示指定模式已完全走完。在这种情况下，这个模式会重新初始化，使得载具可以继续以该模式行动下去。在真实游戏中，你可能有其他控制逻辑可以测试其他条件，再去决定巡逻模式是否应该重复下去。侦测是否有敌军出现是很好的检测。此外，依据你的游戏设计，检查燃料存储量或许也是合适的。实际上你可以检测任何东西，这完全取决于游戏的必需条件。顺带提一下，这一点和有限状态机将牵连在一块，以后会谈。

DoPattern 函数

现在，让我们仔细检视DoPattern()函数，如例3-20所示。

例3-20：DoPattern()函数

```
bool     DoPattern(ControlData *pPattern, int size)
{
    int      i = PatternTracking.CurrentControlID;
    Vector     u;

    // 检查模式数组中的下一组指令是否需要取出
    if(      (pPattern[i].LimitPositionChange &&
         (PatternTracking.dPosition >= pPattern[i].dPositionLimit)) ||
         (pPattern[i].LimitHeadingChange &&
         (PatternTracking.dHeading >= pPattern[i].dHeadingLimit)) )
    {
         InitializePatternTracking();
         i++;
```

```
        PatternTracking.CurrentControlID = i;
        if(PatternTracking.CurrentControlID >= size)
            return false;
    }

    // 计算这组指令开始运行后方向上的改变
    u = Craft2.vVelocity;
    u.Normalize();
    double P;
    P = PatternTracking.InitialHeading * u;
    PatternTracking.dHeading = fabs(acos(P) * 180 / pi);

    // 计算这组指令开始运行后位置上的改变
    u = Craft2.vPosition - PatternTracking.InitialPosition;
    PatternTracking.dPosition = u.Magnitude();

    // 求出转向力系数
    double f;
    if(pPattern[i].LimitHeadingChange)
        f = 1 - PatternTracking.dHeading /
            pPattern[i].dHeadingLimit;
    else
        f = 1;

    if(f < 0.05) f = 0.05;

    // 依照当前这组指令的赋值施加转向力
    Craft2.SetThrusters( pPattern[i].PThrusterActive,
              pPattern[i].SThrusterActive, f);

    return true;
}
```

DoPattern()函数所做的第一件事是，复制模式数组当前索引值CurrentControlID到一个临时变量i，以备后用。

接着，这个函数检查位置或方向改变值，是否已达到当前这组控制指令的限制值。如果是，就把记录结构体重新初始化，使得下组指令可以进入。此外，模式数组的索引值会递增，并测试指定模式是否已到达尾端。如果是，这个函数就会在此时返回false，否则，会继续处理这个模式。

下一段程序代码是计算当前这组指令开始执行后，载具在方向上的改变。载具的方向是从其速度向量取得的。要把方向的改变换算成角度，此例中，你要把速度向量复制到临时向量$\boldsymbol{u}$，然后将之转化成单位向量。接着，把储存在模式记录结构体中的最初方向向量，和代表当前方向的单位向量$\boldsymbol{u}$做内积。结果就存放在标量变量P中。接着，使用向量内积的定义，注意到这两个向量都是单位向量，你就可以取P的反余弦函数，计算出两向量间的夹角。这样会得到径度单位的角度，所以，你必须乘以180，再除以pi，才

能得到角度。注意到，我们对最后的角度值取绝对值，因为我们感兴趣的只有方向角度的变化量而已。

下一段程序代码是计算当前这组指令开始执行后载具在位置上的改变。求取位置的变化量是算出载具当前位置，和储存在模式记录结构体中，最初位置之间的向量差值。最后的向量数值就是距离的变化值。

接着，这个函数会求出适当的转向力系数，以施加底层物理模型所定义的载具最大可用传动推力。求推力系数的步骤是让1减去方向变化值和所需方向变化值（也就是当前这种控制指令所给予的方向限制值）之比值。这个系数会把`SetThrusters()`函数传递给刚体`Craft2`，而这个函数会把最大可用转向力乘以此系数，再把所得的推力，赋给载具左侧或右侧的推进器。

我们把最小转向力系数定为0.05，这样一来，经常有转向力存在。因为这是仿真实际的载具，盖过底层物理而强迫载具移往特定方向是不恰当的。当然，你还是可以这样做，但这会破坏最初使用物理模型的目的。所以，由于我们施加了转向力，这个力需要时间让载具转弯，而且由于载具并非固定在导引铁轨上，所以我们启动或关闭转向力时，以及载具的响应之间，会有一定的时间落差。这表示如果我们转弯时转向力太大，可能会转过头；如果我们打算转90度，根据底层物理模式，我们多少会多转一点。因此，为了避免转过头，我们一开始时要全力转弯，接着越接近目标方向时，转向力就要逐渐减少。这样一来，我们就能在方向的改变上有平滑的结果，逐渐转到我们的目标方向，而不会转过头。

和汽车转弯相比较，如果你想在车里右转，一开始会把方向盘整个转往右边，当你开始转弯后，你会让方向盘往另一方向回转，逐渐让轮胎转正。你不会一直让方向盘打着，直到你转了90度，才突然放开方向盘，还想这样不会转过头。

现在就知道了，我们限定最小转向力的原因，是为了让我们能实际达到我们所要的方向目标。使用“1减去方向变化比率”这样的公式，也就是说，力的系数在方向变化量逐渐往所需的方向变化量接近时，最后会在变化值抵达限制值时成为0。这表示我们在方向上的变化，会渐近式地转往所需的方向变化量，但实际上不会刚刚好，因为转向力最后不是太小就是0。0.05这样的系数，只是我们针对此特定模式所做的校正值而已。你得自行调整你的物理模式，找出你正在仿真的东西的合理值。

结果

图3-4和3-5是此算法的结果，分别为方形模式和蛇行模式。我们从可下载的范例程序的执行结果，把这些画面捕捉下来。

在图 3-4 中，你可以看见计算机控制的载具，实际上走的就是方形模式。你应该注意到方形的角都有良好的截角效果，不是死板的硬转。这里示范的载具的旋转半径是该物理模型，以及先前讨论的转向力推进器校正值的函数。对特定的模型而言，结果可能不同，你得调整这些仿真运算值，使其得到满意的结果。

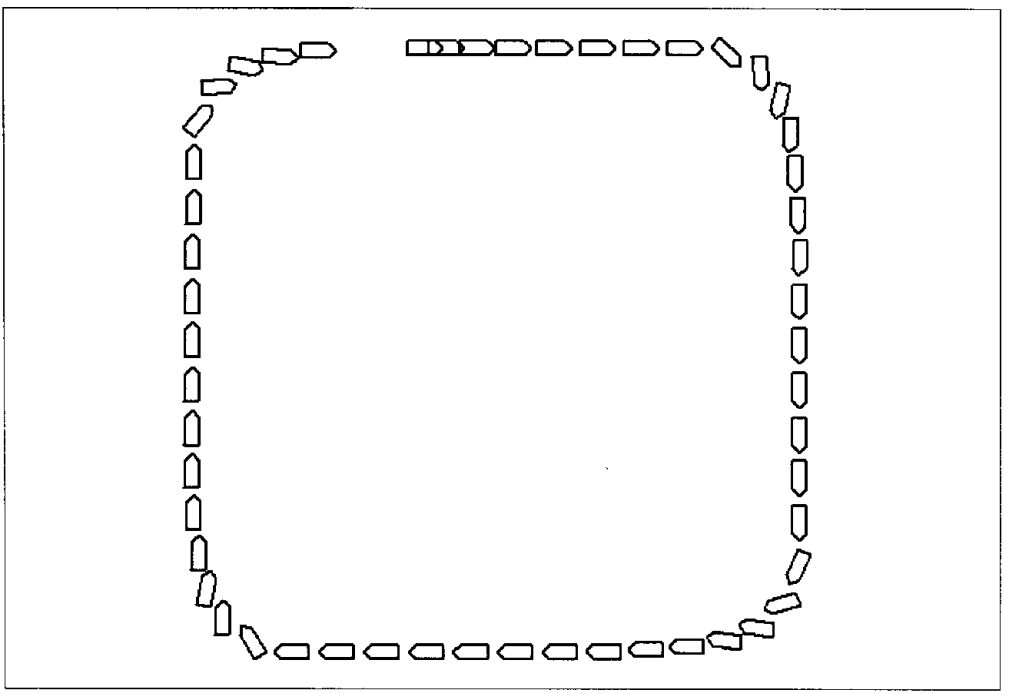


图 3-4：方形路径

图 3-5 是从范例程序中取出的蛇行模式图。同样地，注意到转弯很平滑。这会让路径看起来相当自然。如果它模拟的是飞行器，玩家会希望看到平滑的弯度。

图 3-6 的模式由 10 组指令组成，通知计算机控制的载具直走，向右转 135 度，再直走，向左转 135 度，等等，一直到此模式走出图中所示的结果为止。

只从趣味而言，我们在此例中引入了任意路径，让你知道这个算法，并没有限制你只能用诸如方形和蛇行这类简单模式。你可以编写任何你想象得到的模式，以相同方式放入一连串指令，让你可以得到看似有智能的行动。

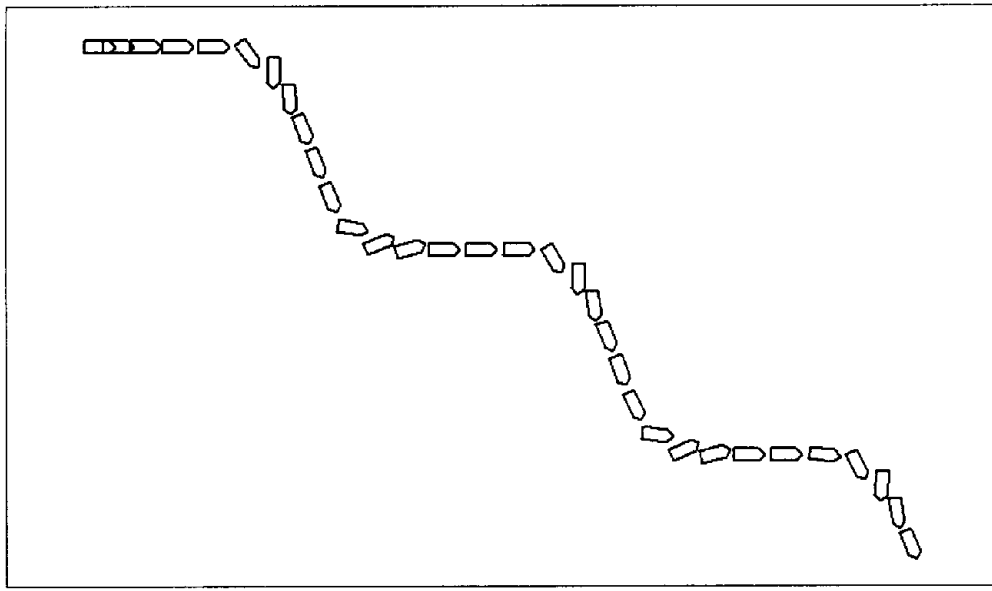


图 3-5：跑形路径

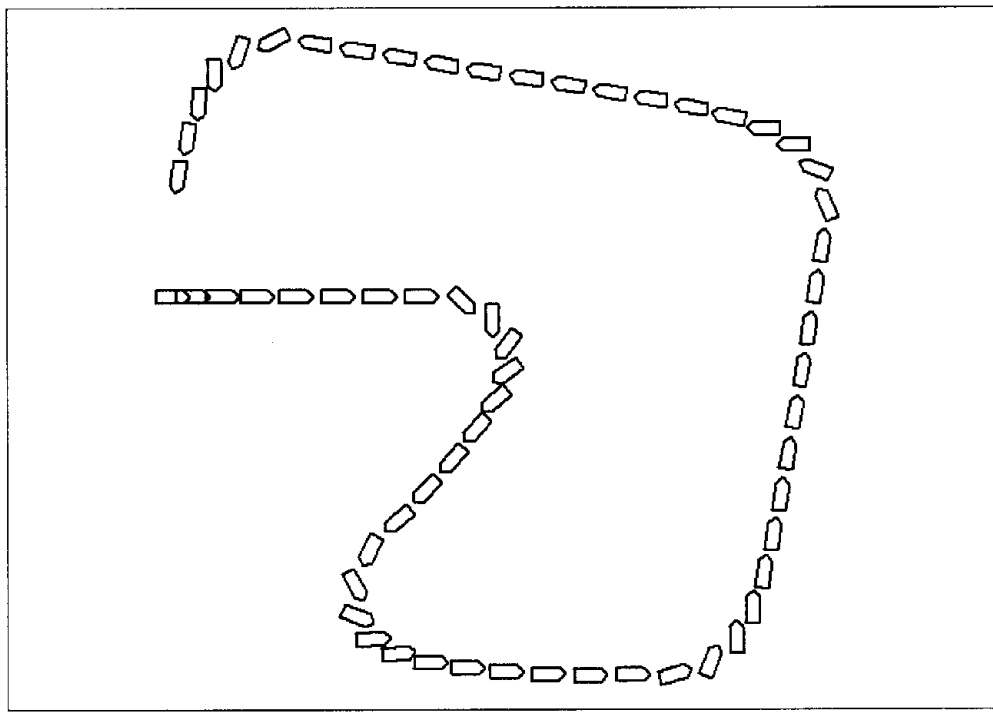


图 3-6：任意路径

第四章

群聚

通常在视频游戏中，非玩家角色必须群聚移动，而不是个别行动。我们举几个实例。假设你在写在线角色扮演游戏，在主城镇外有一片绵羊的牧草地。如果你的绵羊是一整群在吃草，而不是毫无目的地闲逛，看起来会更真实。也许在同一个角色扮演游戏中，有一群鸟会以游戏中的人类居民为猎物。同样地，那些鸟如果成群结队猎食，会比单独行动，看起来更真实一点，同时也让玩家受到挑战，该如何面对群体合作的攻击者。把这种群聚行为套用到巨蚁、蜜蜂、老鼠、或海洋生物时，也不需要做太大变动。

这些局部地区的动物成群结队移动、吃草或攻击的实例，似乎就是在游戏中使用群聚行为的明显例子。话虽如此，但群聚行为并不限于动物，事实上，也可以扩展到其他非玩家角色。例如，在实时策略模拟游戏中，你可以替非玩家角色做移动时，使用群体移动行为。这些非玩家角色可以是计算机控制的人类、巨人、魔鬼、或各式各样机械式的载具。在战争模拟游戏中，你可以把这种群体移动行为，使用在计算机控制的整群飞机上。在第一人称射击游戏中，计算机控制的敌人或友军也可以采用这种群体移动。你甚至可以把基本的群聚行为做某些调整，例如，模拟一群人在镇上的广场闲逛。

在这些例子中，我们的想法是让非玩家角色能聚集在一起移动，让人产生有某种目的的错觉。如果有一群东西在动，但各走各的，看起来不像是群体协调一致的移动，那就完全不一样了。

这种群体行为的核心就是基本的群聚算法，比如，Craig Reynolds 在其 1987 年 SIGGRAPH 论文《Flocks, Herds, and Schools: A Distributed Behavioral Model》中，提到的那个算法。你可以用 Reynolds 当初提出的算法原型仿真整群的鸟、鱼或其他生物，也可以用修改版本，仿真群体移动的物体、军队、或者飞机群。本章要详谈基本群

聚算法，教你如何修改，用来处理诸如避开障碍物这类情况。为了表示的一般性，本章接下来将以“单位”（unit）指称组成群体的个别实体，例如，鸟、绵羊、飞机、人类等。

基本群聚

Craig Reynolds提到他模拟群体时，以“类鸟群”（boids）来指称。他做出的行为非常类似水中的鱼群或成群的飞鸟。所有的类鸟群可以同时朝同一方向移动，接着下一时刻，群聚形成的形体之尖端会转弯，而群体中其余的部分也会跟着转，形成类鸟群在群体中传达转弯的行为，而形成波浪状。Reynolds的算法成果是没有领导核心的，也就是说实际没有类鸟体（boid）在领导这个群体；就某种意义而言，他们都跟着这个群体走，而这个群体似乎有其自身的心思。Reynolds的群聚算法，产生的运动给人留下的印象相当深刻。更令人称奇的是，事实上这个行为只是优雅而简单的三条规则产生的结果而已。这些规则总结如下：

凝聚

每个单位都往其邻近单位的平均位置行动。

对齐

每个单位行动时，都要把自己对齐在其邻近单位的平均方向上。

分隔

每个单位行动时，要避免撞上其邻近单位。

从这三条语句可清楚得知，每个单位都必须有比如运用转向力行进的能力。此外，每个单位都必须得知其局部的周遭情况，必须知道邻近单位在哪里、他们的方向如何，以及他们和自身有多接近。

在仿真实境的连续环境中，你可以对仿真中的单位施加转向力使其改变方向。这里你可以利用我们之前在书中提到的追逐、闪躲以及移动模式实例的相同技巧。（参见第二章，特别是图2-7以及有关如何处理改变方向的相关讨论。）我们要指出一点，你在网站上或其他出版资料上，找到的诸多群聚算法都是以粒子（particle）表示单位，但我们这里打算以刚体来表示，也就是第二章与第三章所谈的。虽然粒子很容易处理，你不用考虑旋转问题，但你的游戏的单位很有可能不是粒子。相反的，游戏的单位应该有明确的体积，有定义完善的前后表面，因此，记录其方位是很重要的，唯有如此，当他们在行动时，他们会转而面对他们要行进的方向。把单位视为刚体，就可以让你考虑方位了。

对砖块环境而言，你可以采用我们在砖块环境追逐和闪躲实例中使用的视线方法，让单位改变方向或者朝特定点移动。例如，就凝聚规则而言，你得让单位朝其邻近单位的平均位置前进，而这个平均位置以砖块坐标表示。（参见第二章的“视线追逐”一节。）

每个单位要注意其邻近单位靠近到什么程度。基本上，每个单位会注意到其局部的邻近状况，也就是说，要知道平均位置、方向及其自身和群体中，与其他最靠近的单位之间的间隔有多少。单位不需知道整个群体，在任何特定时间做些什么。图4-1是单位的局部视野。

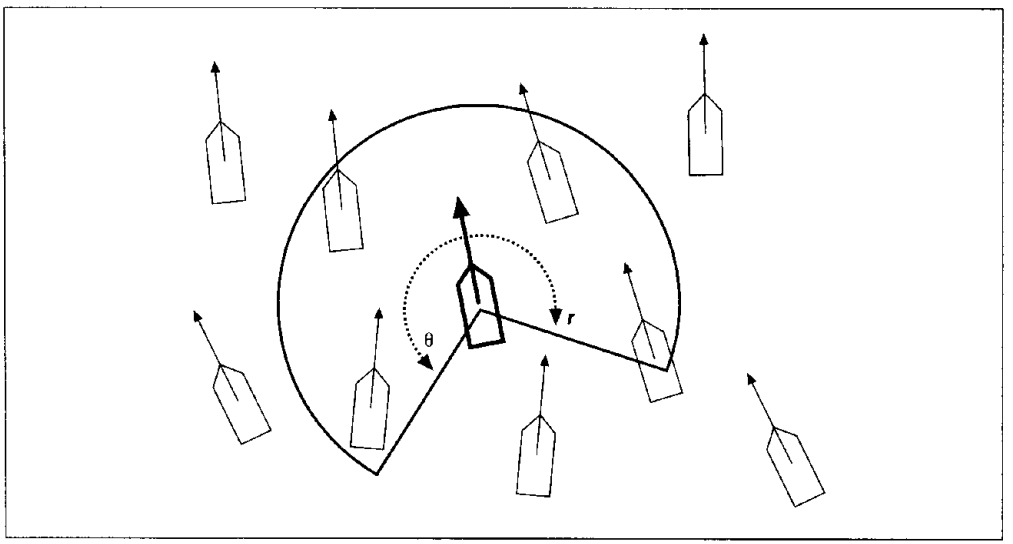


图4-1：单位的视野

图4-1是一个单位（图中用粗线表示的那个）以r为半径画弧而定出其可见视野的说明。任何其他单位落入这个弧内，都能被这个单位看见。运用群聚规则时，这些可视的单位就有用了；而其他单位都会被忽略。可见，弧由两个参数定义，也就是弧半径以及角度θ。这两个参数会影响最后的群聚行动，你可以视需求作调整。

一般而言，大半径会让单位看到群体中更多同伴，从而产生凝聚性更强的群体。也就是说，群体没有分裂成小群体的倾向，因为每个单位可以看见多数邻近单位或全部邻近单位，再据此行进。另一方面，较小的半径让整个群体分裂，形成较小群体的可能性较高。如果某些单位临时看不到他们的邻近单位，而那些邻近单位又是他们和大群体的连接点，整个群体就有可能分裂。当这种现象发生时，被拆散的单位会分裂成较小的群体，而且如果有其他单位刚好又进入他们的视线，就会再度重新聚集。通过障碍物行进也会造成群体分裂。就此而言，较大的半径可以协助整个群体再度结合为单一群体。

另一个参数θ量定了每个单位的视野范围。最宽广的视野当然是360度。有些群聚算法使用360度的视野范围，因为这做起来比较简单，然而，最后得到的群聚行为可能有点失真。常用的视野范围类似于图4-1的说明，每个单位的身后都有一块看不见的区域。这

里你还是可以调整这个参数，以满足需求。一般而言，视野宽广的话，如图4-2左侧所示，其视野角度约270度，就会得到组织良好的群体。视野较窄的话，如图4-2右侧所示，其视角约为45度，得到的群体倾向于像蚂蚁那样沿着单一路径行进。

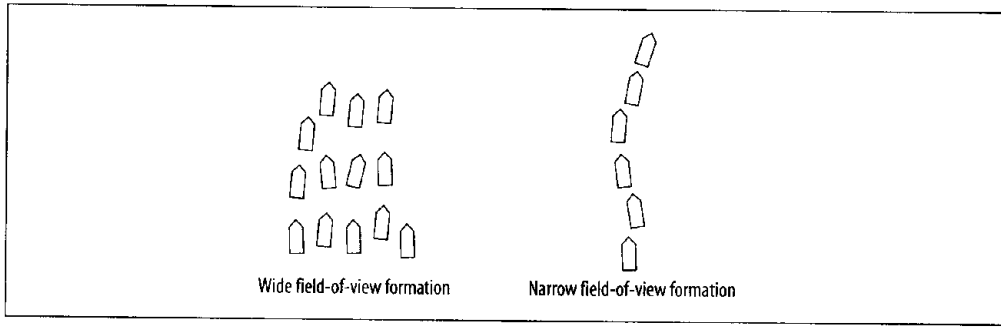


图4-2：宽视野与窄视野的队形

这两种结果都有其用处。例如，如果你正在仿真一群喷射战机，可能会用宽视野。如果仿真一支军队鬼鬼祟祟跟踪某人，你也许会用窄视野，使其前后排成一条线，当他们前进时，就不会变成宽广的目标。如果将后者和避开障碍物两者结合运用，你的单位看起来就像在避开障碍物，同时也会跟着领头的人。

稍后我们将根据这三条群聚规则（凝聚、对齐以及分隔）简化避开障碍物以及领头者。但首先，我们要介绍一些范例程序，来实现这三条规则。

群聚实例

我们要看的范例是在连续环境中仿真好几个单位。这里我们要用先前讨论追逐及移动模式范例中，使用过的同样的刚体仿真算法。这个示范程序名叫AIDemo4，可从本书网站（*http://www.oreilly.com/catalog/ai*）下载。

基本上，我们打算仿真大约20个单位，以群聚的方式移动，然后和环境及玩家互动。就此简单范例而言，和环境的互动就是避开圆形的物体。群聚中的诸多单位和玩家的互动就是去追玩家。

行进模式

就此例而言，我们要实现一个行进模式，差不多和第二章那个以物理机制为基础的范例中所用的相同。你可以参考图2-8以及相关讨论，回想一下当时所做的行进模式。基本上，我们要把每个单位视为刚体，在该单位的前端施加转向力。这个转向力会在该单位

的右侧或左侧推进，而且施加转向力的大小，是每条群聚规则应用后所得的累加值。这种做法可以让我们实现任何数目的群聚规则及其结合方式，因为每条规则都可对总转向力做出一部分贡献，而最终的结果就是，所有规则都考虑过后施加在该单位上的力。

我们也要提醒你，这种做法还需做一些调整，才能确保不是单一规则主导一切。也就是说，你不能让某已知规则所贡献的转向力过于强大，使其总是压倒其他规则所做的贡献。例如，如果我们从凝聚规则获得的转向力远大于其他规则的，而且假设我们实现了一条避开障碍物的规则，让单位试图远离该障碍物，此时如果凝聚规则主导一切，那些单位也许还会待在一块儿。因此，也就无法绕过障碍物，也许会撞上或当作没有障碍物一样直接过去。为了减少这种失衡，我们要做两件事：首先，我们要调控好每条规则所贡献的转向力；其次，我们要调整行进模式，以确保每个单位都获得平衡，至少多数时刻下都是如此。

调整是需要尝试错误的过程。如果要调控转向力，则我们需要把每条规则贡献的转向力，写成方程式或响应曲线的形式，使得贡献的力不是常量。但相反的，我们想让转向力作为已知规则中某些重要参数的函数。

先想一下避开规则（avoidance rule）。此例中，我们试图让单位不会彼此撞上，同时又要让单位能根据对齐和凝聚规则而靠在一起。我们想让单位彼此间距离够宽时，避开规则的转向力贡献要小一点；但是，当单位彼此间靠得太近而太危险时，就让避开规则的转向力贡献大一些。这样一来，当单位彼此远离时，凝聚规则就会让他们再聚在一起而形成群聚，从而不会和避开规则相抵触。此外，一旦单位形成群体，我们想让避开规则的贡献足够大，以避免单位撞在一起，尽管根据凝聚规则和对齐规则，他们有种想聚在一起的倾向。显然，此例中，单位之间的分隔距离是很重要的参数。因此，我们想将避开用的转向力，写成分隔距离的函数。你可以用无数个函数完成这项任务，然而，就我们的经验来看，简单的反函数就够用了。此例中，避开用的转向力和分隔距离是成反比的。因此，分隔距离越大，得出的避开用转向力越小；同时，分隔距离越小，得出的避开用转向力越大。

其他的规则也采用类似方法。例如，就对齐而言，我们会考虑某单位当前方向，与其邻近单位间平均方向间的角度。如果该角度较小，我们只对其方向做小幅度调整，然而，如果该角度较大，就需要做较大的调整。为了完成这样的行为，我们要把对齐用的转向力贡献，设定成和该单位当前方向及其邻近单位平均方向间的角度成正比。下面几节中，我们要看一些实现此行进模式的程序代码，并予以讨论。

邻近单位

如前面所讨论的，群体中的每个单位都必须侦测其邻近单位。至于每个单位能侦测多少个邻近单位，则是以图4-1所示的视野角度及视野半径为参数的函数。由于群体中单位所形成的排列会随时变动，因此，游戏循环每运行一轮时，每个单位都必须更新其视野。也就是说，我们必须绕行群体中的所有单位以收集必要数据。注意到，我们必须针对每个单位做这项工作，以取得每个单位各自的视野。当单位数量增多时，这种邻近单位搜寻法将十分耗费运算资源。我们稍后要讨论的范例程序只是为了清晰说明而已，但这也是做些优化工作的好起点。

示范程序名叫AIDemo4，你可从本书网站（*http://www.oreilly.com/catalog/ai*）下载，其写法类似于本书先前讨论过的范例。此例中，你会发现一个名叫UpdateSimulation()的函数，每次走过游戏循环或仿真运算循环时，就会被调用。这个函数的责任是更新每个单位的位置，并把每个单位画到画面显示缓冲区内。例4-1是此例的UpdateSimulation()函数。

例4-1：UpdateSimulation()函数

```
void     UpdateSimulation(void)
{
    double     dt = _TIMESTEP;
    int          i;

    //初始化后端缓冲区
    if(FrameCounter >= _RENDER_FRAME_COUNT)
    {
        ClearBackBuffer();
        DrawObstacles();
    }

    // 更新玩家控制的单位 (Units[0])
    Units[0].SetThrusters(false, false, 1);

    if (IsKeyDown(VK_RIGHT))
        Units[0].SetThrusters(true, false, 0.5);

    if (IsKeyDown(VK_LEFT))
         Units[0].SetThrusters(false, true, 0.5);

    Units[0].UpdateBodyEuler(dt);

    if(Units[0].vPosition.x > _WINWIDTH) Units[0].vPosition.x = 0;
    if(Units[0].vPosition.x < 0) Units[0].vPosition.x = _WINWIDTH;
    if(Units[0].vPosition.y > _WINHEIGHT) Units[0].vPosition.y = 0;
    if(Units[0].vPosition.y < 0) Units[0].vPosition.y = _WINHEIGHT;

    if(FrameCounter >= _RENDER_FRAME_COUNT)
          DrawCraft(Units[0], RGB(0, 255, 0));
```

```
    // 更新计算机控制的单位
    for(i=1; i<_MAX_NUM_UNITS; i++)
    {
        DoUnitAI(i);

        Units[i].UpdateBodyEuler(dt);

        if(Units[i].vPosition.x > _WINWIDTH)
            Units[i].vPosition.x = 0;
        if(Units[i].vPosition.x < 0)
            Units[i].vPosition.x = _WINWIDTH;
        if(Units[i].vPosition.y > _WINHEIGHT)
            Units[i].vPosition.y = 0;
        if(Units[i].vPosition.y < 0)
            Units[i].vPosition.y = _WINHEIGHT;

        if(FrameCounter >= _RENDER_FRAME_COUNT)
        {
            if(Units[i].Leader)
                DrawCraft(Units[i], RGB(255,0,0));
            else {
                if(Units[i].Interceptor)
                    DrawCraft(Units[i], RGB(255,0,255));
                else
                    DrawCraft(Units[i], RGB(0,0,255));
            }
        }
    }

    // 把后端缓冲区复制到屏幕上
    if(FrameCounter >= _RENDER_FRAME_COUNT) {
          CopyBackBufferToWindow();
          FrameCounter = 0;
    }  else
        FrameCounter++;
}
```

UpdateSimulation()完成的是平常的工作，清除即将绘制图像的后端缓冲区，处理玩家控制的单位的互动行为，更新计算机控制的单位，把一切都绘进后端缓冲区，做好之后，再把后端缓冲区复制到屏幕上。就我们的目的而言，有趣的是计算机控制单位是在何处更新的。就此工作而言，UpdateSimulation()会以循环走遍计算机控制单位的数组，对每个单位而言，都调用另一个名叫DoUnitAI()的函数。一切有趣的事都发生在DoUnitAI()内，所以，本章接下来要介绍的就是这个函数。

DoUnitAI()处理一切和计算机控制单位的移动有关的事。所有群聚规则都在此函数内实现。然而，实现这些规则之前，这个函数得先收集单位的邻近单位数据。注意到，该单位（也就是当前正在处理的单位）是以参数传递进来的。更明确地讲，是把代表当前正在处理的单位的数组索引值当作参数i返回DoUnitAI()。

例4-2是DoUnitAI()开头的一小部分。这段程序只有几个局部变量和初始化的程序代码。一般而言，我们只会概略扫视这类程序代码，但因为这个程序含有很多局部变量，而且在群聚计算中时常用到，所以值得逐一介绍，说明每个变量究竟在干些什么。

例4-2：DoUnitAI()初始化

```
void    DoUnitAI(int i)
{
        int         j;
        int         N;      // 邻近单位数量
        Vector      Pave;   // 平均位置向量
        Vector      Vave;   // 平均速度向量
        Vector      Fs;     // 总净转向力
        Vector      Pfs;    // Fs 施加的位置
        Vector      d, u, v, w;
        double      m;
        bool        InView;
        bool        DoFlock = WideView||LimitedView||NarrowView;
        int         RadiusFactor;

        // 初始化
        Fs.x = Fs.y = Fs.z = 0;
        Pave.x = Pave.y = Pave.z = 0;
        Vave.x = Vave.y = Vave.z = 0;
        N = 0;
        Pfs.x = 0;
        Pfs.y = Units[i].fLength / 2.0f;

        .
        .
        .

}
```

我们已经提过参数i代表当前正在处理的单位的数组索引值。我们要收集这个单位所有邻近单位的数据，然后再实现群聚规则。变量j代表Units数组中，其他单位的数组索引值。这些是Units[i]潜在的邻近单位。N代表邻近单位的数目，这些数目包含在当前正在处理的单位的视野内。Pave和Vave分别存放的是N邻近单位的平均位置和速度向量。Fs代表施加到处理中单位的总转向力。***Pfs***代表转向力施加的位置，以固定于个体上的坐标表示。***d***、***u***、***v***以及***w***用来储存计算函数时的各种向量值。向量值包含全局坐标和局部坐标的相对位置向量和方向向量。m是乘数变量，不是+1就是-1，用来指出我们所需的转向力施加点的方向，就是目前处理的单位的右侧或左侧。InView是个标号，指出特定单位是否位于处理中单位的视野内。DoFlock只是一个标号，指出是否使用群聚规则。此例中，你可以打开或关闭群聚功能。此外，你可以实现三种不同的可见视野模式，以观察群聚行为。这些可见视野模式叫做WideView、LimitedView以及NarrowView。最后，RadiusFactor代表的是图4-1中的r参数，每种可见视野模式的r值都不同。注意到，每种可见视野模式的视野角度也不同，稍后会提到这一点。

所有局部变量都声明之后，有几个有明确的初始值，由例4-2可知，多数都赋为0。你在那里看到的变量都是用来累计某些值的，例如，累计每条规则贡献的转向力，或者累计视野内的邻近单位数量，等等。唯一一个没有把初值设为0的是向量***Pfs***，这是用来表示当前处理中的单位要施加转向力向量的位置。这会让转向力的施力线从该单位的重心偏移出来，使得施加转向力时，该单位会转向并面对适当的方向，并向适当方向移动。

完成局部变量赋初值后，DoUnitAI()就进入一个循环，收集当前单位周遭邻近单位（如果有的话）的信息。

例4-3是DoUnitAI()中的一段，会检查所有的邻近单位并收集数据。到此时，会进入一个循环，就是j循环，在这个循环里面，Units数组的每个单位（Units[0]除外，这是玩家控制的单位，另外，Units[i]也除外，这是当前单位，现在要找的是该单位的邻近单位）都会接受测试，以确认该单位是否在当前单位的视野内。如果是，其数据将被收集起来。

例4-3：邻近单位

```
.
.
.
for(j=1; j<_MAX_NUM_UNITS; j++)
{
     if(i!=j)
     {
          InView = false;
          d = Units[j].vPosition - Units[i].vPosition;
          w = VRotate2D(-Units[i].fOrientation, d);

          if(WideView)
          {
               InView = ((w.y > 0) || ((w.y < 0) &&
                         (fabs(w.x) >
                         fabs(w.y)*
                    _BACK_VIEW_ANGLE_FACTOR)));
               RadiusFactor = _WIDEVIEW_RADIUS_FACTOR;
          }

          if(LimitedView)
          {
               InView = (w.y > 0);
               RadiusFactor = _LIMITEDVIEW_RADIUS_FACTOR;
          }

          if(NarrowView)
          {
               InView = (((w.y > 0) && (fabs(w.x) <
                              fabs(w.y)*
                         _FRONT_VIEW_ANGLE_FACTOR)));
```

```
                RadiusFactor = _NARROWVIEW_RADIUS_FACTOR;
            }

            if(InView)
            {
                if(d.Magnitude() <= (Units[i].fLength *
                                     RadiusFactor))
                {
                    Pave += Units[j].vPosition;
                    Vave += Units[j].vVelocity;
                    N++;
                }
            }

            .
            .
            .
        }
    }

    .
    .
    .
```

确定 i 不等于 j 之后（也就是我们不打算检查当前的单位），这个函数会计算当前单位 Units[i]以及 Units[j]之间的距离向量，也就是两者间位置向量的差值。所得结果会储存在局部变量 d 中。接着，d 会从全局坐标转换成固定于 Units[i]之上的局部坐标。所得结果会储存在向量 ***w*** 之中。

接着，这个函数会去检查 Units[j]是否位于 Units[i]的视野内。这项检查是依据视野角度的函数，如图 4-1 的说明。我们后面也会检查半径值，但前提是视野的检查已通过。

现在，因为此例含有三种不同的可见视野模式，因此，用三段程序做视野的检查。这些检查对照宽广视野、有限视野以及狭窄视野模式。如先前所讨论的，某单位的可见视野会影响该群体的群聚行为。你可以在此范例程序中切换各种模式，以观看其效果。

宽广视野模式提供的可见视野范围最大，因而容易形成及重组群体。此例中，每个单位能直接看到其前方、两侧以及身后两侧，除了直接在其身后的那一小片区域看不见之外的区域。图 4-3 说明了此视野模式。

检测 Units[j]是否落在此视野内的程序代码，由两部分组成。首先，从固定于当前单位的局部坐标系统来看，如果 Units[j]的相对位置使其 *y* 坐标为正值，则我们知道 Units[j]落在视野内。其次，如果 *y* 坐标是负值，则有可能落在视野内，也可能落在看不见的那块区域内，所以，还要再做另一种检查。此时，要看的是 *x* 坐标，以确认

Units[j]是否落在由两条连接可见视野弧线的直线，所形成的饼形不可见区域，如图4-3所示。如果Units[j]的x坐标的绝对值，大于某系数乘以y坐标的绝对值时，我们就知道Units[j]落在不可见区域外，也就是落在视野内。那个乘以y坐标绝对值的系数的计算，就代表先前提到连接视野弧线的那两条直线。做此检查的程序代码如例4-3所示，但为求方便，我们再次将关键处列在例4-4。

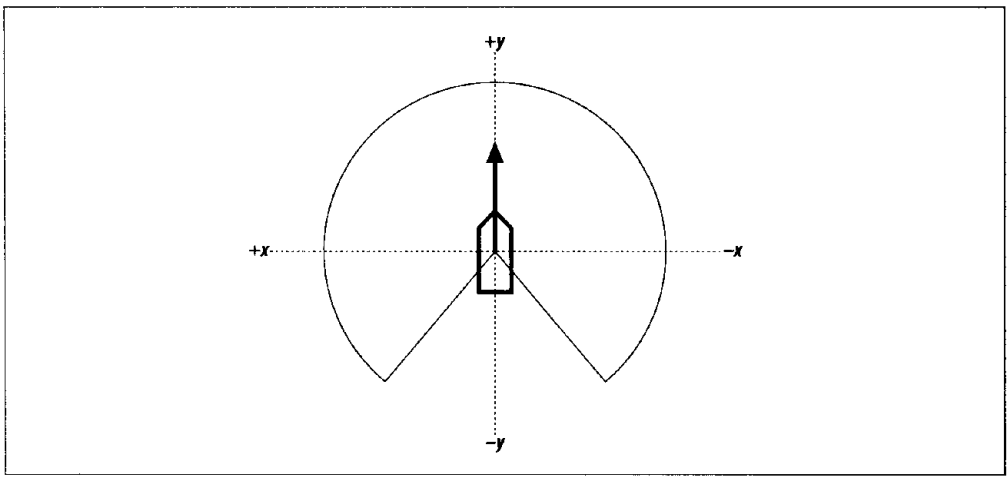


图4-3：宽广视野

例4-4：宽广视野的检查

```
.
.
.
        if(WideView)
        {
            InView = ((w.y > 0) || ((w.y < 0) &&
                        (fabs(w.x) >
                    fabs(w.y)*
                  _BACK_VIEW_ANGLE_FACTOR)));
            RadiusFactor = _WIDEVIEW_RADIUS_FACTOR;
        }
.
.
.
```

在此程序代码内，_BACK_VIEW_ANGLE_FACTOR就是视野角度系数。如果设为1，则连接视野弧线的直线和x轴的夹角就是45度。如果该系数大于1，这两条线会靠近x轴，相当于不可见区域比较大。相反的，如果该系数小于1，则这两条线靠近y轴，相当于不可见区域比较小。

你也会注意到，RadiusFactor也赋以某个预先定义的值_WIDEVIEW_RADIUS_FACTOR。这个系数控制的是如图4-1所示的半径参数。顺便提一下，调整此例的参数时，这个半径系数是获得所需行为而必须校正的参数之一。

另外两个可见视野模式的检查类似于宽广视野模式，然而，这两种模式代表的是越来越小的视野。这两个模式的说明如图4-4和4-5所示。

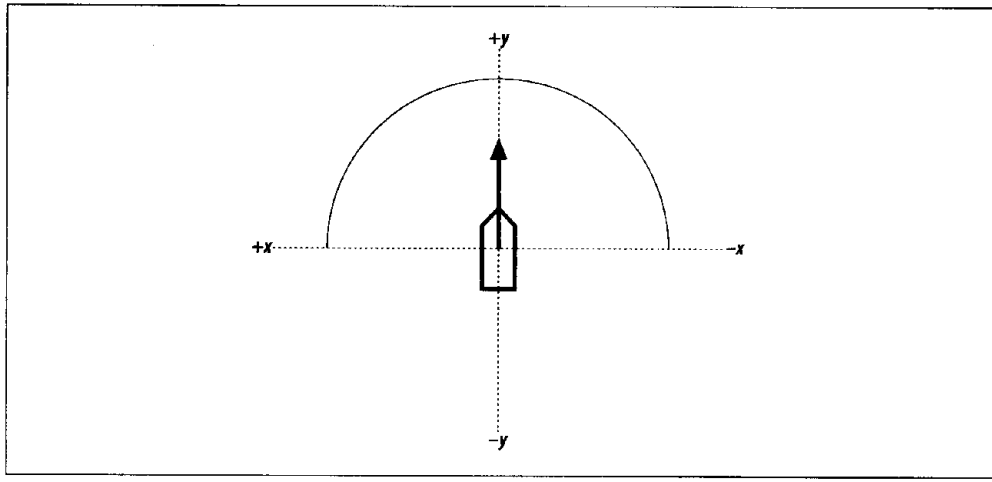


图4-4：有限视野

在有限视野模式中，可见视野弧线限制在该单位局部 +y 坐标。也就是说，每个单位都无法看到身后的任何单位。此例中，测试相当简单，如例4-5所示，你必须确定的是，从Units[i]的局部坐标系统来看，Units[j]的y坐标是否为正值。

例4-5：有限视野的检查

```
.
.
.
        if(LimitedView)
        {
            InView = (w.y > 0);
            RadiusFactor = _LIMITEDVIEW_RADIUS_FACTOR;
        }
.
.
.
```

狭窄视野模式把每个单位可见的范围限制在大约正前方，如图4-5所示。

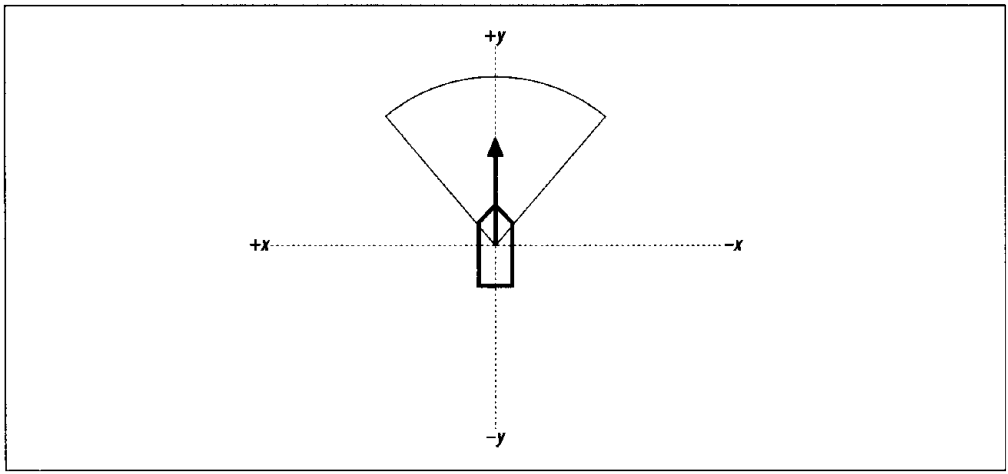


图 4-5：狭窄视野

此例的检查程序也非常类似于宽广视野的情况，其可见视野弧线也被某些系数控制住。其计算如例 4-6 所示。

例 4-6：狭窄视野的检查

```
.
.
.
        if(NarrowView)
        {
             InView = (((w.y > 0) && (fabs(w.x) <
                          fabs(w.y)*
                     _FRONT_VIEW_ANGLE_FACTOR)));
             RadiusFactor = _NARROWVIEW_RADIUS_FACTOR;
        }
.
.
.
```

此例中，系数 _FRONT_VIEW_ANGLE_FACTOR 控制了该单位前方的视野。如果此系数等于 1，则构成视野锥的两条线和 x 轴的夹角就是 45 度。如果系数大于 1，这两条线会靠近 x 轴，也就是说可见区域比较大。如果系数小于 1，则这两条线靠近 y 轴，也就是说可见区域比较小。

根据你替此例所选的视野模式，一旦上述测试都过关了，接下来还要再检查 Units[j] 是否和 Units[i] 之间保持在特定距离内。如果 Units[j] 位于视野内，而且位于特定距离内，则表示 Units[i] 看得见 Units[j]，也就会被视为邻近单位，以备后续计算所用。

例4-3的最后一个 if 区块就是测试此距离。如果向量 *d* 的数值小于 Units[i] 的长度乘以 RadiusFactor，则表示 Units[j] 和 Units[i] 够接近，可视为邻近单位。注意到，这个预定的分隔阈值是以该单位的长度乘以某系数设定的。你可以视需求而改用任何值，不过你得根据你的游戏调整出适当值，然而，我们喜欢使用半径系数乘以该单位的长度，是因为这是可大可小的设定方法。如果由于某种原因，使你决定改变游戏世界的规模（空间），包括游戏中的单位，则其可见视野也会按其比例放大，你也不必回去，根据新尺度来调整某些新的可见距离。

凝聚

凝聚意指我们想让所有单位都待在同一群体中，我们不要每个单位和其群体分开，各走各的路。如前所述，为了满足这项规则，每个单位都应该朝其邻近单位之平均位置前进。图4-6是某单位与其邻近单位关系的说明。图中的小圆点，代表的是位于该单位可见视野粗弧线内，四个邻近单位的平均位置。

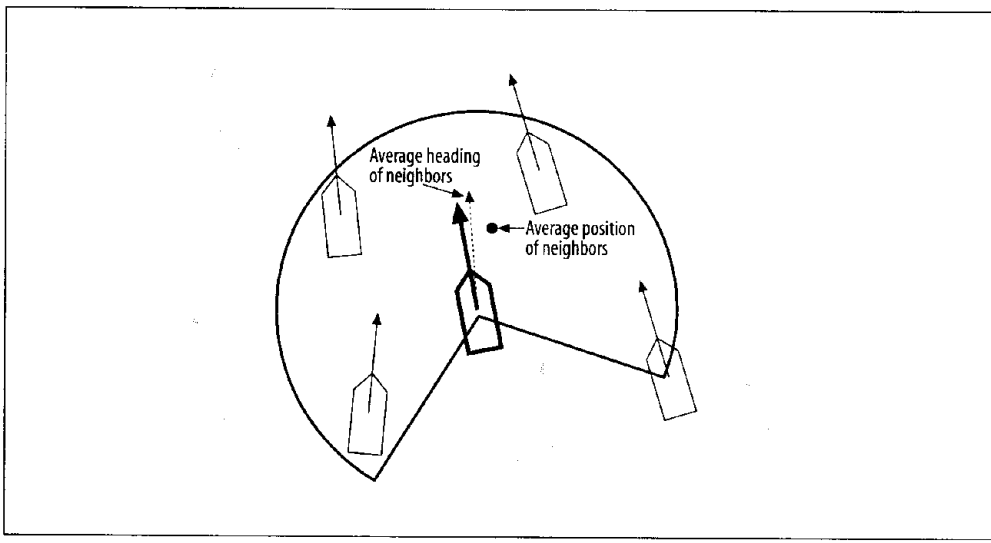


图 4-6：邻近单位的平均位置和方向

邻近单位的平均位置很容易计算。只要找出邻近单位后，其平均位置就是其各个位置的向量总和，再除以总邻近单位数（数值）。所得结果就是代表其平均位置的向量。由例4-3可知邻近单位找出来之后，其位置总和就会被算出。为了方便起见，相关程序代码在例4-7中再重列一次。

例 4-7：邻近单位位置总和

```
.
.
.
if(InView)
{
     if(d.Magnitude() <= (Units[i].fLength *
                                RadiusFactor))
     {
          Pave += Units[j].vPosition;
          Vave += Units[j].vVelocity;
          N++;
     }
}

.
.
.
```

Pave += Units[j].vPosition;这一行将所有邻近单位的位置向量相加。记住，Pave 和 vPosition 是 Vector 类的变量，此 overloaded operator（重载运算符）会替我们做向量加法。

DoUnitAI()找出邻近单位并收集信息后，就能使用群聚规则了。第一个处理的就是凝聚规则，程序代码如例 4-8 所示。

例 4-8：凝聚规则

```
.
.
.
// 凝聚规则
if(DoFlock && (N > 0))
{
    Pave = Pave / N;
    v = Units[i].vVelocity;
    v.Normalize();
    u = Pave - Units[i].vPosition;
    u.Normalize();
    w = VRotate2D(-Units[i].fOrientation, u);
    if(w.x < 0) m = -1;
    if(w.x > 0) m = 1;
    if(fabs(v*u) < 1)
        Fs.x += m * _STEERINGFORCE * acos(v * u) / pi;

}
.
.
.
```

注意到，此段程序的第一件事是检查邻近单位数量是否大于0。如果是，我们就能继续计算邻近单位的平均位置。做法是以所有邻近单位位置的向量总和Pave除以邻近单位数量N。

接着，把当前处理中的单位Units[i]方位储存在*v*中，并换算成单位向量，后续计算会用到。现在，取Pave和Units[i]位置之间的向量差值，就能算出Units[i]及其邻近单位平均位置之间的距离。所得结果储存在*u*中，并换算成单位向量。接着，把*u*从全局坐标转换成固定于Units[i]之上的局部坐标，再储存在*w*中。这样就知道，相对于Units[i]当前位置而言，Units[i]邻近单位的平均位置。

接着，要确定和转向力有关的乘数m。如果*w*的*x*坐标值大于0，则表示邻近单位的平均位置位于Units[i]的右侧，则Units[i]需左转。如果*w*的*x*坐标值小于0，则必须右转（左侧）。

最后，做一个快速检查，以得知单位向量*v*和*u*的内积是否小于1且大于−1（注1）。这一定要做，因为这个内积在计算两向量间角度时会用到，而反余弦函数的自变量须介于1和-1之间。

例4-8的最后一行就是实际计算满足凝聚规则的转向力。该行中，转向力会在Fs.x中累加，累加值为方向系数乘以预定的最大转向力，再乘以当前单位的方向，以及到邻近单位平均位置向量间的角度，再除以*pi*。当前单位的方向及其邻近单位平均位置向量间的角度，是利用*v*及*u*两向量的内积、再取反余弦函数而得到的。这源自于内积的定义。注意到*v*和*u*这两个向量都是单位向量。把所得的角度除以*pi*，就可得到一个标量系数，可代入最大转向力的运算中。基本上，累加在Fs.x的转向力是当前单位的方向，及其邻近单位平均位置向量间的角度的线性函数。也就是说，如果角度大，则转向力也会相对较大；如果角度小，则转向力也会相对较小。这正是我们想要的。如果当前单位的方向和其邻近单位的平均位置的方向相距很远，我们会想让他做大幅度的转弯。如果其方向和邻近单位平均位置的方向不太远，我们只想对其方向做小范围的修正。

对齐

对齐意指我们想让群聚中的所有单位都大致朝相同的方向前进。为了满足这条规则，每个单位都应该在行进时，试着以等同于其邻近单位平均方向的方向来前进。参见图4-6，中间以粗线表示的单位是沿着和其相连的粗箭头方向行进的。另外和其相连的虚箭头则代表其邻近单位的平均方向。因此，就此例而言，以粗线表示的单位必须朝右侧行进。

注1： 参考附录复习向量内积运算。

我们可以利用每个单位的速度向量求出其方向。把每个单位的速度向量，换算成单位向量，就可得出其方位向量。例4-7显示出收集某单位邻近单位方向数据的过程。那一行`Vave += Units[j].vVelocity;`会把每个邻近单位的速度向量累加在`Vave`中，其做法类似于`Pave`累加位置的过程。

例4-9说明了计算每个单位的对齐转向力。这里的程序代码几乎和例4-8的凝聚规则一样。此处，不再处理邻近单位的平均位置，而是把`Vave`除以邻近单位数量`N`，先求出当前单位邻近单位的平均方向。所得结果则储存在***u***中，并换算成单位向量，则为平均方向向量。

例4-9：对齐规则

```
.
.
.
// 对齐规则
if(DoFlock && (N > 0))
{
     Vave = Vave / N;
     u = Vave;
     u.Normalize();
     v = Units[i].vVelocity;
     v.Normalize();
     w = VRotate2D(-Units[i].fOrientation, u);
     if(w.x < 0) m = -1;
     if(w.x > 0) m = 1;
     if(fabs(v*u) < 1)
          Fs.x += m * _STEERINGFORCE * acos(v * u) / pi;
}
.
.
.
```

接着，当前单位`Units[i]`的方向，可将其速度向量换算成单位向量而求出。所得结果储存在***v***中。现在，把当前单位的邻近单位的平均方向，从全局坐标转换成固定于`Units[i]`之上的局部坐标，再储存于向量***w***中。行进方向系数`m`则仍如先前的做法计算。此外，如同凝聚规则那样，对齐转向力也会累加在`Fs.x`中。

就此例而言，转向力是当前单位方向，及其邻近单位平均方向间角度的线性函数。同样的，只要当前单位的方向，和其邻近单位的平均方向很接近，则只需稍微作调整，然而，如果当前单位的方向，和其邻近单位的平均方向相距太远，我们就需做大幅度调整。

分隔

分隔意指我们想让每个单位彼此间保持最小距离，即使根据凝聚和对齐规则，他们会试

着靠近一点。我们不想让那些单位撞在一起，或者更糟的是，在某个巧合的地点重叠在一起。因此，我们要采用分隔手段，让每个单位和其视野内的邻近单位保持某一预定的最小分隔距离。

图4-7有个单位和那个粗线表示的单位靠得太近了。以粗线表示的单位为中央的外层弧线是可见视野弧线，我们已讨论过。而内层弧线代表的就是最小分隔距离。任何单位只要移进此最小分隔弧线内，则粗线表示的单位就会离他远一点。

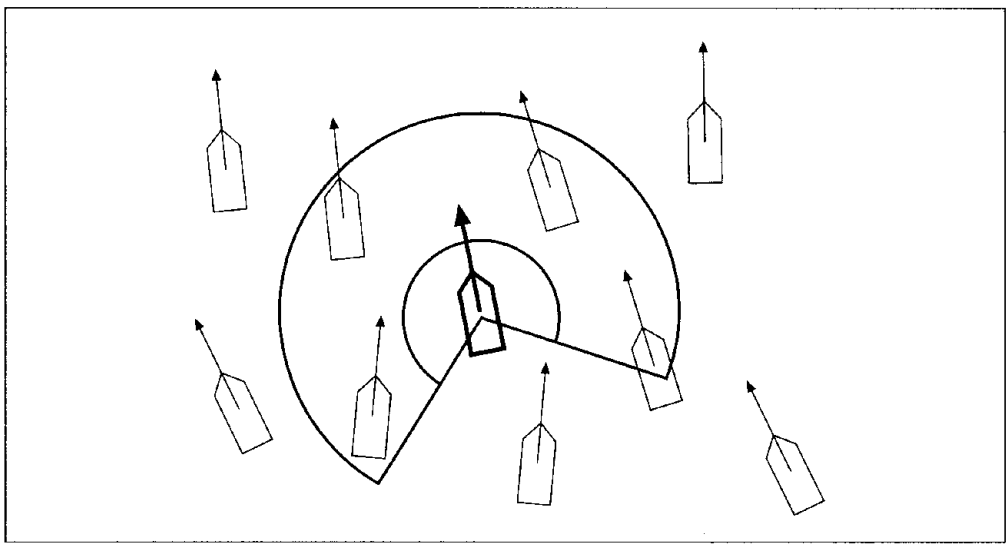


图4-7：分隔

处理分隔的程序和处理凝聚及对齐的只有一点不同，因为就分隔而言，求算适当的转向力校正值时，我们必须逐一检视每个邻近单位，而不是使用所有邻近单位的某个平均值。把分隔程序代码放在例4-3的那个j循环里很方便，邻近单位就是在那里找出来的。新的j循环加上分隔规则的操作程序代码，如例4-10所示。

例4-10：邻近单位和分隔

```
.
.
.

for(j=1; j<_MAX_NUM_UNITS; j++)
{
  if(i!=j)
  {
       InView = false;
       d = Units[j].vPosition - Units[i].vPosition;
       w = VRotate2D(-Units[i].fOrientation, d);
```

```
        if(WideView)
        {
             InView = ((w.y > 0) || ((w.y < 0) &&
                    (fabs(w.x) >
                    fabs(w.y)*_BACK_VIEW_ANGLE_FACTOR)));
             RadiusFactor = _WIDEVIEW_RADIUS_FACTOR;
        }

        if(LimitedView)
        {
             InView = (w.y > 0);
             RadiusFactor = _LIMITEDVIEW_RADIUS_FACTOR;
        }

        if(NarrowView)
        {
             InView = (((w.y > 0) && (fabs(w.x) <
                  fabs(w.y)*_FRONT_VIEW_ANGLE_FACTOR)));
             RadiusFactor = _NARROWVIEW_RADIUS_FACTOR;
        }

        if(InView)
        {
             if(d.Magnitude() <= (Units[i].fLength *
                                       RadiusFactor))
             {
                  Pave += Units[j].vPosition;
                  Vave += Units[j].vVelocity;
                  N++;
             }
        }

        if(InView)
        {
             if(d.Magnitude() <=
                (Units[i].fLength * _SEPARATION_FACTOR))
             {
                  if(w.x < 0) m = 1;
                  if(w.x > 0) m = -1;

                  Fs.x += m * _STEERINGFORCE *
                          (Units[i].fLength *
                           _SEPARATION_FACTOR) /
                           d.Magnitude();
             }
        }
   }
}

.
.
.
```

最后一个 if 区块就是新增的分隔规则程序代码。基本上，如果 j 单位位于视野内，而且位于从当前单位 Units[i] 位置算起 Units[i].fLength *_SEPARATION_FACTOR 的距离内，我们就要计算转向力校正值，并予以施加。注意到，d 是分隔 Units[i] 和 Units[j] 的距离，而且是在 j 循环一开头就算出来的。

一旦确认 Units[j] 有可能撞在一起，则程序就会接着去计算校正的转向力。首先，先确定方向系数 m，才知道所得的转向力要从哪个方向把当前 Units[i] 推离 Units[j]。就此例而言，m 的计算和凝聚及对齐规则中的算法恰好相反。

如同凝聚和对齐规则的情况那样，转向力也是在 Fs.x 之中累加。就此而言，此校正的转向力和实际的分隔距离成反比。当 Units[j] 和当前单位太近时，校正转向力就很大。同样的，注意到最小分隔距离的算法，是该单位长度及某预定分隔系数的函数。这样做的话，就能让分隔距离可大可小，如同先前讨论过的可见视野那样。

我们还应提及一点，尽管分隔力已在此算出，但那些单位并不见得总是能百分之百地彼此避开。有时候，所有转向力的总和会让某单位必须和某邻近单位相当靠近或者有一部分重叠。你可以把分隔转向力设得高一些，把其他力覆盖过去，但你会发现，当那些单位彼此间靠得很近时，其行为会变得很奇怪。此外，要让群体维持下去也变得很困难。最后，根据你的游戏的需求，还得实现某种碰撞侦测及响应算法，类似《Physics for Game Developers》（O'Reilly）所讨论的那样，处理某几个单位撞在一起的情况。

你也应该注意到，可见视野对分隔规则有很大的影响。例如，在宽广视野模式下，那些单位的分隔情况会维持得很好，然而，在狭窄视野模式下，那些单位无法维持前后左右的分隔距离。这是因为他们的视野有限，有些邻近的单位他们是看不见的。如果你在游戏中采用这种有限的视野模式，处理分隔规则时，可能得用另一种视野模式，例如宽广视野模式。你可以轻易修改此例，把最后的 if 区块的逻辑比例条件，改成宽广视野模式，以确认某单位是否在视野内，借此改用另一种模式。

一旦所有群聚规则都实现后，而且也替当前单位都算出适当的转向力，DoUnitAI()就会把最后的转向力，以及施加的点存放在当前单位的成员变量中，如例 4-11 所示。

例 4-11：Units[i]成员变量的赋值

```
void DoUnitAI(int i)

        // 计算所有的转向力 ...
        .
        .
        .

        Units[i].Fa = Fs;
        Units[i].Pa = Pfs;
}
```

DoUnitAI()返回之后，UpdateSimulation()就负责施加新的转向力，更新那些单位的位置（参见例4-1）。

避开障碍物

到目前为止，我们讨论的群聚规则可获得给人留下深刻印象的结果。然而，像这样的群聚行为如果在游戏中，这些单位以群体形态移动时，还能避免撞上游戏世界里的物体，那就更真实更有用了。事实上，加入避开障碍物的行为其实相当简单。我们所要做的就是提供某种机制给那些单位使用，让他们能看到前方，再施加适当的转向力，使其避开路径中的障碍物。

此例中，我们要考虑一个简单而理想化的障碍物，也就是说，障碍物是圆形的。你的游戏不一定如此，但你可以用我们在此所用的做法，处理其他形状的障碍物。当然，唯一的差别就是几何形状，以及你如何以数学手段，决定某单位是否会撞上障碍物。

为了侦测障碍物是否在某单位的路径内，我们要借助机器人学，替我们的单位安装虚拟触角（feeler）。基本上，这些触角会处在单位的前方，如果触角碰到某种东西，就是那些单位要转向的时候了。我们假设每个单位都能在某种程度上看得到障碍物，如此才能计算障碍物位于该单位的哪一侧。这样就知道要右转还是左转了。

我们所说的模型并非唯一可用的形式。例如，你可以替你的单位装上一个以上的触角，比如，有三个触角位于三个不同方向，不但侦测是否有障碍物，而且侦测该障碍物位于该单位的哪一侧。宽广的单位需要一个以上的触角，才能确保该单位不会和障碍物碰撞。在3D游戏中，你可以用虚拟的体积，在该单位的前方伸展出来。然后，你可以对着游戏世界中的几何图形来测试此体积，以测定是否即将和某障碍物碰撞。你可以采用的手段有很多种。

回到我们所谈的手段，看一下图4-8，以了解我们的虚拟触角如何在几何条件下操作。向量v代表的就是触角。这个触角有某个预定的固定长度，和该单位的方向在同一直线上。那个又大又暗的圆圈代表障碍物。为了求出触角是否和障碍物在某点相交，我们得用上向量数学知识。

首先，我们运算向量a，这只是该单位和障碍物位置间的差值。接着，我们取a和v的内积，将a投射到v上，由此可得向量p。把向量p减去向量a，可得向量b。现在，要测试v是否和圆的某处相交，我们得测试两种情况。首先，p的数值必须小于v的数值。其次，b的数值必须小于该障碍物的半径r。如果两者都是如此，则需要校正转向力，否则，该单位可继续沿当前方向前进。

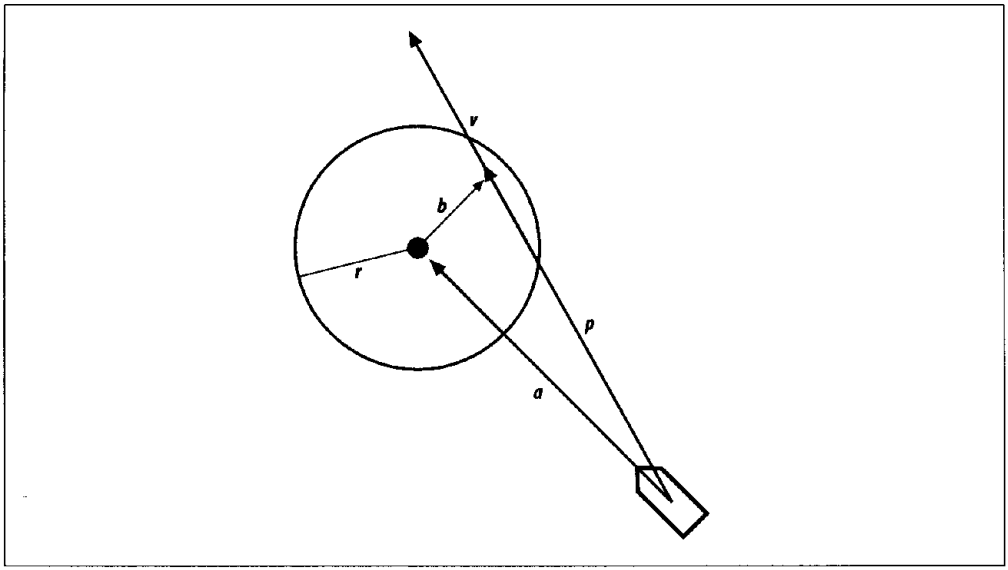


图 4-8：避开阻碍物

即将发生碰撞事件而需施加的转向力计算方法，在某种方式上类似于先前讨论过的群聚规则。基本上，必要的力的计算，是和该单位离该障碍物中心的距离成反比。更明确地说，转向力是预定最大转向力乘以 *v* 的数值和 *a* 的数值之比值的函数。当该单位比较靠近障碍物时，转向力校正就会比较大，因为避开是要紧的。

例 4-12 必须加进 DoUnitAI()，以执行这些避开运算内容的程序代码。你要把这些程序代码加在处理三条群聚规则程序代码之后。注意到，游戏世界中的所有障碍物，都会在循环中被检视，以确定是否即将发生碰撞。同样的，实际上，你会想优化这段程序代码。还注意到，校正的转向力也会累加在 Fs.x 成员变量内，和其他群聚规则的转向力加在一起。

例 4-12：避开障碍物

```
.
.
.
Vector      a, p, b;

for(j=0; j<_NUM_OBSTACLES; j++)
{
     u = Units[i].vVelocity;
     u.Normalize();
     v = u * _COLLISION_VISIBILITY_FACTOR *
         Units[i].fLength;
```

```
        a = Obstacles[j] - Units[i].vPosition;
        p = (a * u) * u;
        b = p - a;

        if((b.Magnitude() < _OBSTACLE_RADIUS) &&
            (p.Magnitude() < v.Magnitude()))
        {
             // 即将碰撞……要避开
             w = VRotate2D(-Units[i].fOrientation, a);
             w.Normalize();
             if(w.x < 0) m = 1;
             if(w.x > 0) m = -1;
             Fs.x += m * _STEERINGFORCE *
                     (_COLLISION_VISIBILITY_FACTOR *
                     Units[i].fLength)/a.Magnitude();
        }
    }
    .
    .
    .
```

如果你下载并执行此例，将发现这些单位以群体形态移动时，同时也会避开随机放置的圆形物体。以不同的可见视野模式进行实验，观察群聚碰上障碍物时的行为很有意思。对宽广可见视野模式而言，群聚会倾向于分开，从两边绕过障碍物。在某些情况下，他们会迅速重聚，但某些情况下不会。对有限及狭窄可见视野模式而言，单位倾向于排成单一纵队，平滑地绕过障碍物，而不会分开。

我们应该指出，这个避开障碍物算法，不一定保证单位和障碍物之间不会碰撞。在一种情况下会发生，例如某个单位收到彼此冲突的行进指令，也许会迫使他撞上障碍物，如果单位碰巧和邻近单位在某一边过于接近，同时又试图避开另一边的障碍物时。根据该单位和邻近单位及障碍物的相对距离而定，某个转向力可能大于另一个转向力，从而引发碰撞。同样的，明智地调整参数有助于减缓这种问题，但实际上，你还是得执行某种碰撞侦测和响应算法，以适当处理这些潜在的碰撞。

跟随领头者

对基本群聚算法的修改不必只限于避开障碍物。因为来自于各种规则的转向力都在同一变量中累加，然后一次施加，以控制该单位的方向，所以，你可以在我们已经考虑过的规则之上再叠加其他许多有效的规则。

其中一条有趣的外加规则，就是跟随领头者规则。如前所述，我们所讨论的群聚规则是没有领头者的，然而，如果我们把基本群聚算法和某些领头者AI结合起来，就可以在游戏中使用群聚功能时，开启许多新的可能效果。

此时，这三条群聚规则，似乎让群体在游戏世界中随处闲逛。如果我们在其中加入领头者，就能让群体的移动更有目的性，或者看起来比较有智能。例如，在战争模拟游戏中，计算机也许控制一群飞机追击玩家。我们可以指定计算机控制的飞机中的其中一架作为领头者，使其追逐玩家，同时让其他计算机控制的飞机，采用基本群聚规则而跟着领头者跑，看起来就像是在表演飞行术那样。在和玩家交战的同时，当天空混战发生时，就可以适时关掉群聚规则。

在另一实例中，你可能想仿真一支军队，步行巡逻丛林。你可以指定其中某个单位作为领头者，根据你想要让他们成横队或成纵队，采用宽广视野模式或有限视野模式，使其他单位采取群聚行为。

我们现在对先前讨论过的群聚实例，要做的就是加上某种领头者能力。就此而言，我们不想明确指定任何一个特定单位作为领头者，而是让某些简单规则，找出谁应该或足以担任领头者。这样一来，任何单位在任何时刻都有可能成为领头者。这种做法的好处是当领头者被除掉，或者因某种原因而脱离其群体时，整个群体不会因此而失去领导。

一旦领头者建立好之后，我们就可以执行任何数目的规则或技巧，让领头者做些有意义的事。我们可以让领头者完成某些预定的模式，或者去追逐某物，或者逃命。此例中，我们要让领头者去追逐，或拦截玩家控制的单位。此外，我们要把计算机控制的单位分成两类：普通单位和拦截者。拦截者比普通单位要快一点，会遵照书中之前讨论过的拦截算法行动。普通单位会走得比较慢，而且会遵照书中之前讨论过的追逐算法行动。你可用数不胜数的方式，定义或分类单位。我们这样做只是举例说明某些可能性而已。

例4-13是几行必须加进例4-3的语句，例4-13求出给定单位所有邻近单位数据的程序区块。

例4-13：检查谁当领头者

```
.
.
.
if(((w.y > 0) &&
    (fabs(w.x) < fabs(w.y)*_FRONT_VIEW_ANGLE_FACTOR)))
     if(d.Magnitude() <=
        (Units[i].fLength * _NARROWVIEW_RADIUS_FACTOR))
          Nf++;
.
.
.

if(InView &&
   (Units[i].Interceptor == Units[j].Interceptor))
{
```

```
        .
        .
        .
    }
    .
    .
    .
```

第一个 if 区块是做检查，使用我们谈过的狭窄视野模式，求出当前处理中单位前方视野内的单位数量。接着，这些信息将被用来确认当前单位是不是领头者。基本上，如果给定单位的前方没有其他单位，那么该单位就是领头者，其他单位就得跟着他以群聚行为行动。如果该单位前方至少有一个单位位于其视野内，则当前单位就不是领头者，而只能遵循群聚规则行动。

例4-13的第二个if区块是对InView测试做简单的修改。外加的程序代码所做的检查，是确保当前单位和Units[j]的类型相同，使得属于拦截者的单位，和其他属于拦截者的单位一起群聚行动，而普通单位和其他普通单位一起群聚行动，如此一来，这两种类型的单位就不会混在同一群体中了。因此，如果你下载此范例程序并予以执行，把其中一种群聚模式打开，至少会看见两组群体：一个是普通单位的群体，另一个是拦截者这类单位构成的群体。(注意，玩家控制的单位会以绿色显示，你可以用键盘方向键进行控制。)

例4-14是这两类计算机控制的单位领头者规则的操作内容。

例4-14：领头者、追逐和拦截

```
    .
    .
    .
    // 如果该单位是领头者，就去追逐目标
    // 注意：Nf 是当前单位前方的单位数目
    if(Chase)
    {
       if(Nf == 0)
           Units[i].Leader = true;
       else
           Units[i].Leader = false;

       if(Units[i].Leader)
       {
           if(!Units[i].Interceptor)
           {
               // 追逐
               u = Units[0].vPosition;
               d = u - Units[i].vPosition;
               w = VRotate2D(-Units[i].fOrientation, d);
               if(w.x < 0) m = -1;
               if(w.x > 0) m = 1;
```

```
                Fs.x += m*_STEERINGFORCE;
            } else {
                // 拦截
                Vector      s1, s2, s12;
                double      tClose;
                Vector      Vr12;

                Vr12 = Units[0].vVelocity -
                       Units[i].vVelocity;
                s12 = Units[0].vPosition -
                      Units[i].vPosition;
                tClose = s12.Magnitude() /
                         Vr12.Magnitude();

                s1 = Units[0].vPosition +
                    (Units[0].vVelocity * tClose);
                Target = s1;
                s2 = s1 - Units[i].vPosition;
                w = VRotate2D(-Units[i].fOrientation, s2);
                if(w.x < 0) m = -1;
                if(w.x > 0) m = 1;
                Fs.x += m*_STEERINGFORCE;
            }
        }
    }
    .
    .
    .
```

如果你打开此范例程序的追逐选项，则Chase变量会赋值true，而这里列出的程序区块就会被执行。在此区块内，会检查当前单位前方视野内的单位数目Nf，以确定当前单位是否为领头者。如果Nf为0，则表示当前单位前方无其他单位，因此，成为领头者。

如果当前单位并非领头者，什么事也不会发生，然而，如果是领头者，则会执行追逐或拦截，视其类型而定。这些追逐和拦截算法和我们先前讨论过的一样，所以这里不再讨论了。

这些新领头者规则加了一些有趣的行为到此范例程序中。在此范例程序中，任何领头者都会变成红色，你可轻易看出任何单位在仿真过程中，如何变成领头者或群聚里的单位。此外，只有这两种简单的计算机控制单位，才会产生某些有趣的战术行为。例如，在猎捕玩家的单位时，其中一组群体会尾随在玩家之后，而另一组群体似乎是从侧翼包抄，试图拦截他。这种结果很像双钳战术。

显然，你可以给领头者加其他AI，使其领导行为更聪明。此外，你可以定义并新增其他类型的单位进来，建立更高的复杂度。这种可能性无穷无尽，我们讨论的只是作为有哪些可能性的说明而已。

第五章

以势函数实现移动

本章我们要借用物理学的某些原理，做些调整以便在游戏软件AI中使用。特别地，我们要使用势函数（potential function）控制游戏里某些情况下计算机控制的单位行为。例如，我们可以在游戏中使用势函数，建立成群结队的单位，仿真群体移动，处理追逐和闪躲，以及避开障碍物问题。我们专门研究的这个势函数叫做Lenard-Jones势函数。我们会让你知道这个函数是什么，以及如何在游戏中应用。

游戏软件AI中如何使用势函数？

我们回顾第二章详谈过的追逐及闪躲问题。回想一下，我们考虑了一些不同的技巧，让计算机控制的单位，追逐或闪躲玩家控制的单位。这些技巧包括基本追逐算法（计算机控制的单位总是直接往玩家之处移动）以及拦截算法。我们可以用势函数完成这两种技巧所能达到的类似行为。这里使用势函数的优点是，只用一个函数处理追逐和闪躲，不再需要先前介绍过的算法所牵涉到的其他条件和控制逻辑。此外，这个势函数也可以替我们处理避开障碍物的问题。虽然很方便，但也有代价。后面将讨论，在游戏里的单位和对象数量增多时，一旦彼此互动起来，这个势函数算法将耗用大量的CPU资源。

势函数算法的另一个优点是操作起来很简单。你所要做的就是计算两个单位（此处即计算机控制的单位以及玩家）之间的驱动力，然后将该驱动力施加到计算机控制单位的前端，作为转向力。这里的转向模式类似第二章和第四章讨论的那几种，然而，此处的力只会指向连接处理中两个单位的作用线。也就是说，该驱动力可以指向相对于计算机控制单位的任何方向，而不仅仅是其左侧或右侧。把力施加到该单位的前端，就能令其转动，朝向力所指的方向。把该驱动力的方向逆转后，我们就可以根据需要让该单位去追

击或逃命了。注意到，转向力也对单位的驱动力有贡献，所以，当单位在移动时，你也许会发现他会加速或减速。

也许你已经猜到了，本章接下来要考虑的范例都采用仿真实境模型，像在前几章见到的那样。事实上，我们用了第二章、第三章以及第四章的范例，只做了稍微修改。像以前一样，你可以在本书网站（*http://www.oreilly.com/catalog/ai*）找到这些范例程序。

何谓势函数？

有大量的书籍讨论势能理论在各种物理现象中的应用，以及位能、力和功（potential, force, work）在物理世界中，存在着广为大家所接受的关系。然而，我们不必太深入那些理论，就能在游戏软件 AI 中使用所谓的 Lenard-Jones 势函数。对我们有意义的是这个函数的作用是什么，以及我们如何在游戏中利用该作用。

$$U = -\frac{A}{r^n} + \frac{B}{r^m}$$

这个方程式就是 Lenard-Jones 势函数。图 5-1 是指数 n 和 m 取不同值时，该函数所画出的三条曲线。

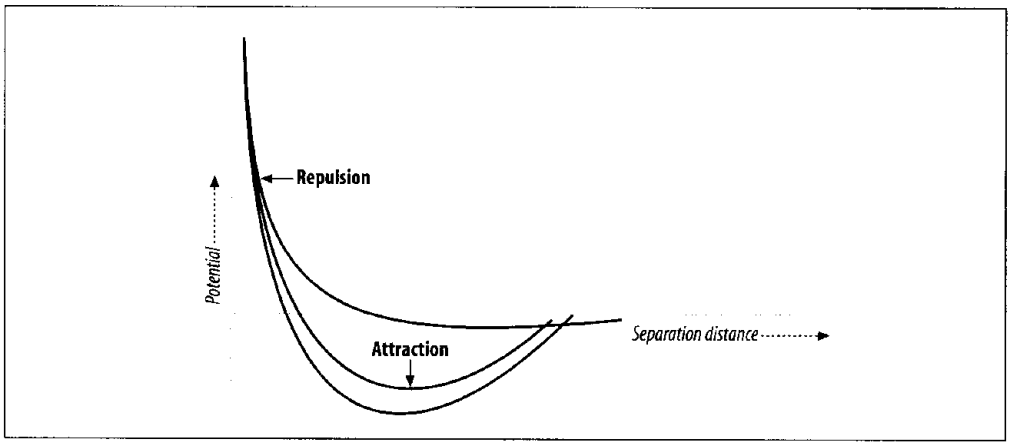


图 5-1：Lenard-Jones 势函数

在物理学中，Lenard-Jones 势能代表的是，分子间吸引和排斥的势能。这里的 U 代表的是原子内的势能，和分子间的分隔距离 r 成反比。A 和 B 是参数，与 m 和 n 这两个指数一样。如果我们取该势函数的导数（derivative），就可得到一个代表某力的函数。这个力函数根据这两个分子的接近程度，产生引力和斥力，就我们的情况而言，分子指的就

是游戏中正在行动的单位。就是这种可以表示引力和斥力的能力，能让我们受益，然而，我们要处理的不是分子，而是计算机控制的单位。

那么，我们怎么用这种能力来吸引或排斥计算机控制的单位呢？首先，我们可以用该Lenard-Jones函数让计算机控制的单位，对玩家控制的单位产生兴趣，使得计算机控制的单位去追玩家。我们也可以调整势函数的参数，让计算机控制的单位对玩家产生排斥，这样，则会使他去躲避玩家。此外，我们还可以让许多玩家控制的单位具有不同的权重，让其中某些玩家控制的单位，对计算机控制的单位更具吸引力或排斥力。这样我们就有办法优先排出目标和威胁的所在了。

除了追逐和闪躲之外，我们也可以利用相同的势函数，让计算机控制的单位避开障碍物。基本上，当计算机控制的单位靠近障碍物时，障碍物会令其产生排斥而使其远离。我们甚至可以让计算机控制的单位彼此吸引，以形成群体。然后，我们可以利用其他影响力，引导这个群体往玩家或某单位之处前进或远离，或者避开途中的障碍物。

使用Lenard-Jones势函数来做这些事最有优势的地方就是，只要一个简单的函数就能让我们做出各式各样看似有智能的行为。

追逐 / 闪躲

要实现位能生成的追逐或闪躲行为，我们只需在AIDemo2-2（参见第二章的细节）程序中加入一些程序代码。在那个范例程序中，我们在连续环境中模拟追击者和猎物这两个单位。函数UpdateSimulation()负责在游戏循环每运行一轮时，处理和玩家之间的互动并更新每个单位的状态。我们打算在该函数内加上两行，如例5-1所示。

例5-1：追逐 / 闪躲范例的UpdateSimulation()

```
void      UpdateSimulation(void)
{
    double dt = _TIMESTEP
    RECT   r;
    // 玩家控制 Craft1
    Craft1.SetThrusters(false, false);

    if (IsKeyDown(VK_UP))
            Craft1.ModulateThrust(true);

    if (IsKeyDown(VK_DOWN))
            Craft1.ModulateThrust(false);

    if (IsKeyDown(VK_RIGHT))
            Craft1.SetThrusters(true, false);
```

Lenard-Jones 势函数

下列方程式是 Lenard-Jones 势函数：

$$U = -\frac{A}{r^n} + \frac{B}{r^m}$$

在固体力学中，U 代表的是原子内的势能，它和分子间的分隔距离 r 成反比。A 和 B 是参数，m 和 n 这两个指数也是参数。要取得两分子间的原子内力，我们要取此势函数的导数的负值，得到：

$$F = -\frac{\mathrm{d}V}{\mathrm{d}r} = -\frac{nA}{r^{n+1}} + \frac{mB}{r^{m+1}}$$

同样的，这里的 A、B、m 以及 n 都是参数，在处理仿真材料的引力和斥力时，选定其值。例如，如果科学家试着仿真固体（如钢铁）和液体（如水）时，这些参数就会不一样。注意到，势函数有两项：一个是 $-A/r^n$，另一个是 B/r^m。含有 A 和 n 的项代表合力的引力分量（component），而含有 B 和 m 的项则代表斥力分量。

斥力分量只作用在相当短距离的 r 之内（从该物体算起），但 r 距离越小，其作用力越大。曲线的负值部分从纵轴往下移得很远，代表的是引力。这里，此作用力不大，但作用的范围在很宽广的分隔距离 r 之内。

你可以调整 n 和 m 两个参数，改变势能曲线或力曲线的斜率，如此就能调整斥力或引力控制的范围，同时让你在转换点上持有某些控制权。你可以把 A 和 B 分别当作引力和斥力的强度，那么，n 和 m 分别代表这两个分力的衰减。

```
if (IsKeyDown(VK_LEFT))
        Craft1.SetThrusters(false, true);

// 做Craft2的AI
.
.
.
if(PotentialChase)
        DoAttractCraft2();

// 更新每台载具的位置
Craft1.UpdateBodyEuler(dt);
Craft2.UpdateBodyEuler(dt);
```

```
    // 更新屏幕
    .
    .
    .
}
```

正如你所见，我们多加了一次检查，看PotentialChase标号是否设为true。如果是，我们就执行计算机控制的单位Craft2的AI，只是现在改用势函数了。DoAttractCraft2()替我们做这件事。基本上，该函数所做的就是用势函数算出两单位间的引力或斥力，再把所得结果当成转向力施加到计算机控制的单位上。例5-2是DoAttractCraft2()。

例5-2：DoAttractCraft2()

```
// 对Craft2施加Lenard-Jones势能所得到的力
void     DoAttractCraft2(void)
{
    Vector  r = Craft2.vPosition - Craft1.vPosition;
    Vector  u = r;

    u.Normalize();

    double  U, A, B, n, m, d;

    A = 2000;
    B = 4000;
    n = 2;
    m = 3;
    d = r.Magnitude()/Craft2.fLength;
    U = -A/pow(d, n) + B/pow(d, m);

    Craft2.Fa = VRotate2D( -Craft2.fOrientation, U * u);

    Craft2.Pa.x = 0;
    Craft2.Pa.y = Craft2.fLength / 2;

    Target = Craft1.vPosition;

}
```

此函数里的程序代码以非常简单的方式实现了Lenard-Jones势函数。进入该函数之后，首先计算的是Craft1和Craft2之间的位移向量，做法就是取两者之间位置的向量差值。所得结果储存在向量***r***内，并将其复制到向量***u***内，以备后用。注意到，***u***也换算成单位向量了。

接着，声明了好几个局部变量，以对应Lenard-Jones势函数的各个参数。变量的命名正好直接对应先前讨论的参数。唯一多出来的新参数是d。d代表的是分隔距离r除以该单位的长度，这样得到的分隔距离，就是以该单位的长度为单位换算出来的结果，而不再是该单位的位置。这样做是考虑到尺度伸缩的目的，如第四章中所说的那样。

除了把r做除法运算得到d之外，其他参数都以某些常数值代入。当然，你不必这样做，你可以从某些文件或其他来源读取那些值。我们这样做只是为了清晰起见。一旦采用实际的数值，要确认其值就要通过调整，例如，可采用试误法来调整，直到所需结果能满足为止。

其中U = -A/pow(d, n) + B/pow(d, m);这一行，会算出实际施加到计算机控制的单位的转向力。我们在此也用U这个符号，但记住一点，我们算的实际上是力。此外，也要注意U是标量，根据此力是引力或斥力，会是负值或正值。为了取得该力的向量，我们将之乘以单位向量$\boldsymbol{u}$，其方向是沿着连接两单位的作用线。然后，所得结果会转换成固定在Craft2之上的局部坐标系统，使其能以之为转向力。此转向力将施加在Craft2的前端，使其向前或远离目标Craft1。

执行此修改后的追逐程序时，我们会看见计算机控制的单位，根据我们定义的参数追逐或闪躲玩家控制的单位。图5-2是调整参数后所产生的某些结果。

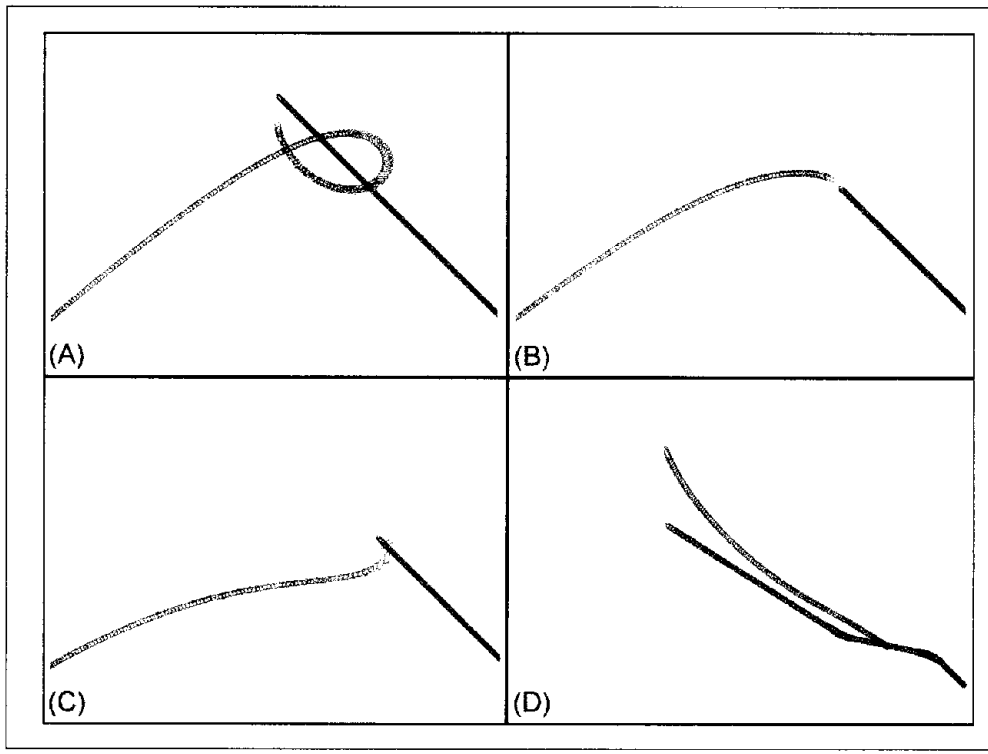


图5-2：位能追逐和闪躲

在图5-2（A）中，追击者朝猎物前进，当猎物和他擦身而过时，他会绕回来。当追击者太接近时，会突然转一下，以维持两单位间的某些分隔距离。在图5-2（B）中，我们减少引力分量的强度（把A参数的值减少一点），其结果就很像我们在第二章提到的拦截算法。在图5-2（C）中，我们增强引力的强度，而结果就很像基本视线算法。最后，在图5-2（D）中，我们减少引力，增加斥力，并调整指数参数，结果计算机控制的单位就会逃离玩家。

调整参数让你在调控计算机控制单位的行为时有很大弹性。此外，你不需要让每个单位都使用相同参数。你可以给不同单位设定不同的参数，使其行为多样化，或者说，让每个单位有其个性。

你可以再进一步把这个势函数，和第二章讨论的其中一种追逐算法结合起来。如果你测试过AIDemo2-2，你会注意到位能追逐的菜单选项，和其他追逐算法并非彼此互相排斥。这表示你可以同时打开势能追逐和基本追逐。这样的结果会很有趣。追击者会按照所想的那样拼命追逐猎物，但当他接近猎物的某个半径范围时，会保持某个分隔距离。追击者将在猎物身旁徘徊，一直面对着猎物。如果猎物转向而朝着追击者前进，则追击者会转向而跑走，直到猎物停下来为止，接着追击者又会继续像影子一样跟着猎物。在游戏中，你可以用此行为来控制外星人的飞碟，让他们去追踪玩家的喷射战机或宇宙飞船。你也可以用这种算法，建立野狼或狮子偷偷靠近其猎物的行为，先保持安全距离，直到适当时机再出现。你甚至可以在足球游戏中用这样的行为，让守门员掩护接球员。你的想象力不限于此，此例只是说明结合不同算法以增加变化的威力，希望能让你借此做出某些有所突破的行为。

避开障碍物

你可能已经了解了，我们可以利用Lenard-Jones函数的斥力性质处理障碍物。就此而言，我们要把引力强度A这个参数设为0，只留下斥力分量。然后，我们可以调整参数B，决定斥力强度，以及指数m来调整衰减程度（例如，斥力的影响半径）。这样就能让我们有效地模拟圆形刚体。当计算机控制的单位靠近这些物体之一时，斥力就会产生，迫使该单位远离该物或绕过该物。记住一点，斥力的值是分隔距离的函数。当该单位靠近该物时，此力也许还算小，转弯就会是渐近的。然而，如果该单位很靠近了，斥力就会变大，迫使该单位紧急转弯。

在AIDemo5-1中，我们在场景中做了好几个随机放置的圆形物体。然后，做了一个计算机控制的单位，使其随机选取路径。这样做是为了了解该单位是否能避开所有障碍物。事实上，该单位避开这些障碍物的过程很顺畅，如图5-3所示。

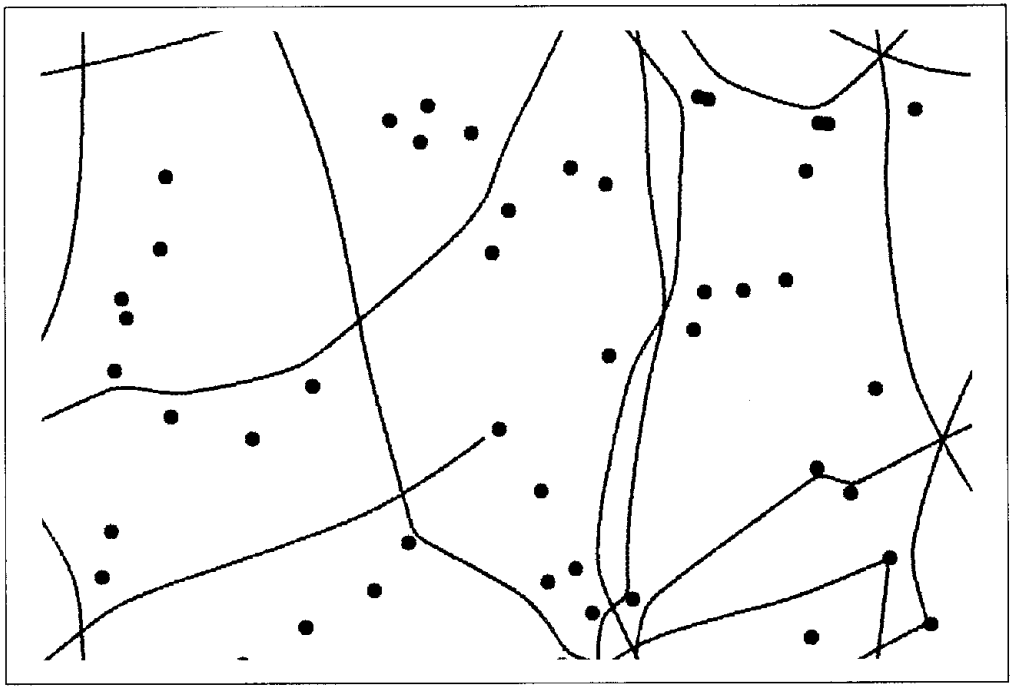


图 5-3：避开障碍物

这里的黑圆点代表障碍物，而弯曲的路线就是计算机控制的单位在通过这个场景时留下的轨迹。显然，从此图像可知，对那些有一定距离的物体而言，该单位避开时的转弯很缓和。此外，当该单位发现和某物非常接近时，就会采取断然转弯之策。这种行为非常类似第四章在群聚实例中所达到的效果。然而，我们在这里为达到这种结果采用了一个大不相同的机制。

这一切如何操作，从观念上来讲非常简单。游戏循环每运行一轮时，都会绕过储存在数组中的所有障碍物，对每个障碍物而言，都会计算其和该单位之间的斥力。对很多障碍物而言，此斥力会很小，因为这些障碍物和该单位的距离很遥远，然而，对那些和该单位距离很近的障碍物而言，此斥力就会大很多。所有贡献的斥力都会累加起来，最后的结果就当作转向力而施加到该单位上。这些计算过程如例 5-3 所示。

例 5-3：避开障碍物

```
void      DoUnitAI(int i)
{
          int               j;
          Vector            Fs;
          Vector            Pfs;
          Vector            r, u;
          double            U, A, B, n, m, d;
```

```
        Fs.x = Fs.y = Fs.z = 0;
        Pfs.x = 0;
        Pfs.y = Units[i].fLength / 2.0f;

        .
        .
        .

        if(Avoid)
        {
             for(j=0; j <_NUM_OBSTACLES; j++)
             {
                  r = Units[i].vPosition - Obstacles[j];
                  u = r;
                  u.Normalize();

                  A = 0;
                  B = 13000;
                  n = 1;
                  m = 2.5;
                  d = r.Magnitude()/Units[i].fLength;
                  U = -A/pow(d, n) + B/pow(d, m);

                  Fs += VRotate2D( -Units[i].fOrientation,
                                        U * u);
             }
        }

        Units[i].Fa = Fs;
        Units[i].Pa = Pfs;
}
```

这里展示的斥力计算，本质上和追逐范例中所用的相同，然而，此例中，参数A设为0。此外，斥力计算是针对每个障碍物而言的，因此，斥力计算是封装在一个for循环内走过Obstacles数组。

你不必把自己局限在圆形或球状障碍物。虽然斥力确实有球状影响范围，你还是可以用好几个圆形来近似任何形状的障碍物。你可以把好几个圆圈排列起来，彼此靠近，以形成一道墙，而且还可以使用不同衰减程度和强度设定值组织障碍物，使其逼近任何形状。图5-4是利用许多小型的圆形障碍物，构成一个方形墙，单位可在它的内部任意移动。

就此例而言，我们只是以AIDemo5-1为例，规则地排列障碍物，以构成方形墙。我们使用例5-3所用的算法，阻止单位离开方形墙。图5-4显示的轨迹，就是该单位在方形墙内移动时所走的路径。

当然，这是很简单的范例，但说明了你确实可以逼近任何非圆形的边界。理论上，你可以把好几个圆形障碍物排在跑道两侧，建立边界，让计算机控制的跑车能在跑道内行进。这些边界不一定为玩家所用，只用于引导计算机控制的单位。你可以结合这类边界和其

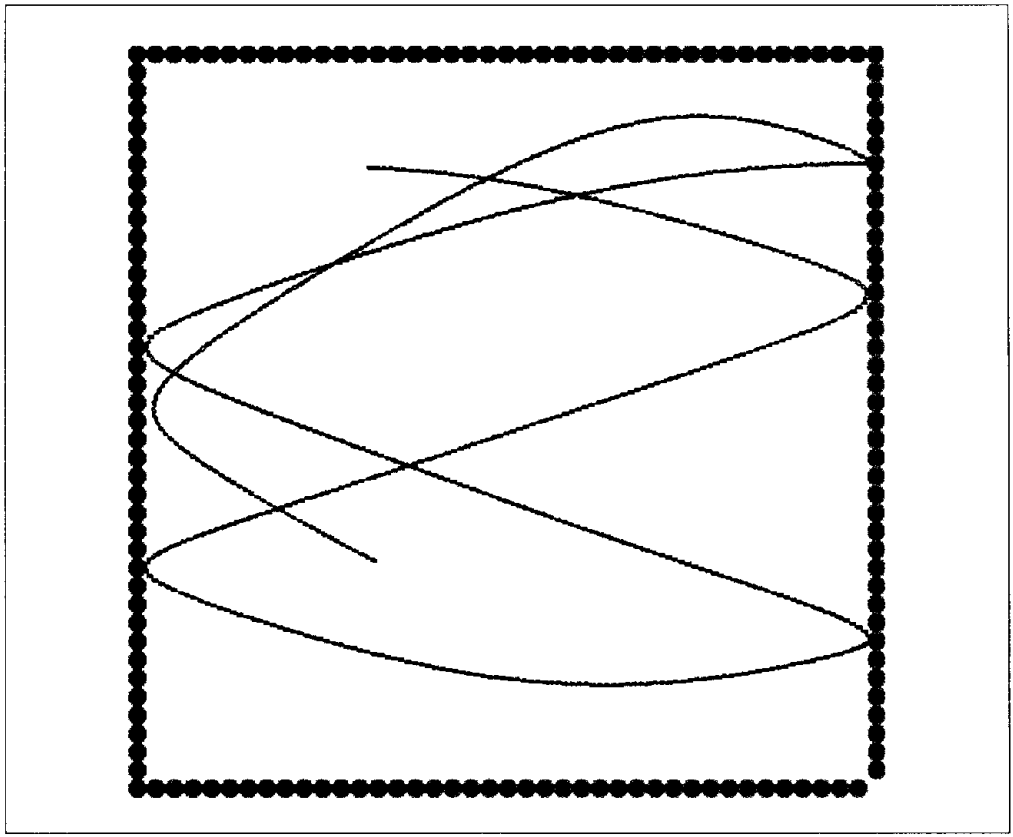


图5-4：四方墙

他只有引力的边界，然后，按策略予以排列，让计算机控制的单位特别偏爱某条跑道，或者在赛车道上来回切换跑道。后一种技巧和航点有关，稍后会谈到。

成群结队

让我们把群体行为，作为游戏软件AI使用势函数的另一个实例说明。明确地讲，让我们考虑成群结队（swarming）。这种行为和群聚很类似，然而，所得结果的行为看起来比较混乱。我们要谈的不再是一群优雅的飞鸟，而是更像一群愤怒的蜜蜂。利用势函数，仿真这种行为就很简单了。这不需要规则，不像群聚那样。我们要做的就是计算群体中单位之间的Lenard-Jones驱动力。这些力的引力分量会让这些单位靠在一起（凝聚），而斥力分量会让他们彼此远离（避开）。

例5-4是建立成群结队的群体所用的势函数。

例5-4：成群结队的群体

```
void     DoUnitAI(int i)
{

          int             j;
          Vector    Fs;
          Vector    Pfs;
          Vector    r, u;
          double    U, A, B, n, m, d;

          // 群聚AI开始
          Fs.x = Fs.y = Fs.z = 0;
          Pfs.x = 0;
          Pfs.y = Units[i].fLength / 2.0f;

          if(Swarm)
          {
               for(j=1; j <_MAX_NUM_UNITS; j++)
               {
                    if(i!=j)
                    {
                         r = Units[i].vPosition -
                             Units[j].vPosition;
                         u = r;
                         u.Normalize();

                         A = 2000;
                         B = 10000;
                         n = 1;
                         m = 2;
                         d = r.Magnitude()/
                                   Units[i].fLength;
                         U = -A/pow(d, n) +
                                   B/pow(d, m);

                         Fs += VRotate2D(
                                    -Units[i].fOrientation,
                                    U * u);
                    }
               }
          }

          Units[i].Fa = Fs;
          Units[i].Pa = Pfs;
          // 群聚AI结束

}
```

同样的，此处计算每个单位和其他单位之间的作用力的程序代码，和先前几例所用的力的计算方式都相同。这里主要的差别，是我们要计算每个单位和其他单位之间的作用力。这也就是说，只要i不等于j，我们就必须进入多层循环，绕行Units数组，计算

Units[i]和Units[j]之间的作用力。显然，当成群结队的单位数量变多时，就会导致很大的计算量。稍后我们会讨论一下如何让此程序最佳化。

就目前而言，图 5-5 说明了成群结队行为的结果。

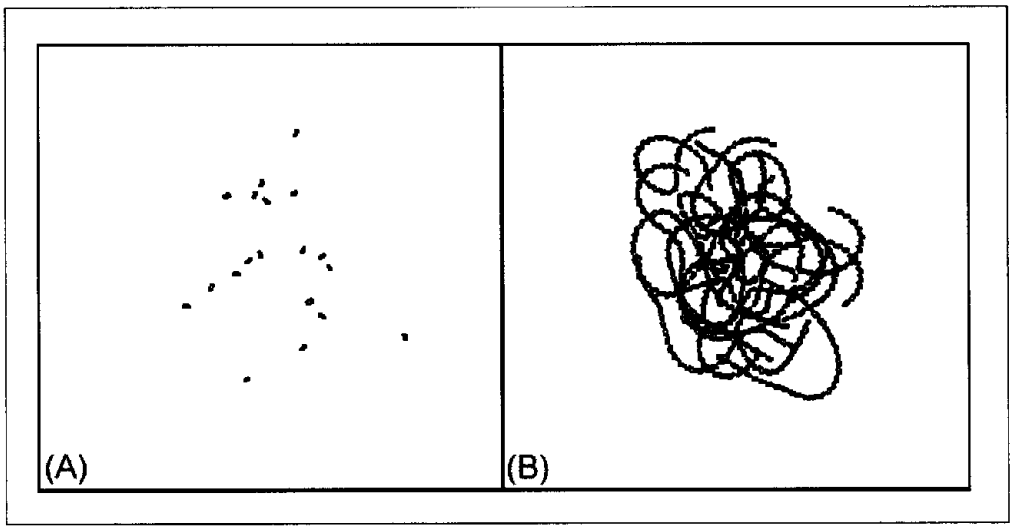


图 5-5：成群结队

只凭捕风捉影难以对成群结队的行为作评论，所以你应该下载范例程序，自己试一试，看看成群结队的行为实际是怎样的。无论如何，图5-5（A）是这些单位聚在一起的图形。图5-5（B）是每个单位所走的路径。显然，路径会缠绕交叉在一起。这样的结果很像是蜜蜂或苍蝇的集体行为。

你也可以在例5-4的成群结队算法内结合先前讨论过的追逐和避开障碍物算法。这会让你的群体不单只会成群结队而已，还会去追猎物，路上碰到障碍物时还会避开。例5-5显示了例5-4应该对函数做的修改，不仅能成群结队，还会去追猎物、避开障碍物。

例 5-5：成群结队、追逐及避开障碍物

```
void     DoUnitAI(int i)
{

          int       j;
          Vector    Fs;
          Vector    Pfs;
          Vector    r, u;
          double    U, A, B, n, m, d;

          // 群聚AI开始
          Fs.x = Fs.y = Fs.z = 0;
```

```
Pfs.x = 0;
Pfs.y = Units[i].fLength / 2.0f;


if(Swarm)
{
     for(j=1; j (_MAX_NUM_UNITS; j++)
     {
          if(i!=j)
          {
               r = Units[i].vPosition -
                   Units[j].vPosition;
               u = r;
               u.Normalize();

               A = 2000;
               B = 10000;
               n = 1;
               m = 2;
               d = r.Magnitude()/
                    Units[i].fLength;
               U = -A/pow(d, n) +
                     B/pow(d, m);

               Fs += VRotate2D(-Units[i].fOrientation,
                                 U * u);
          }
     }
}

if(Chase)
{
     r = Units[i].vPosition - Units[0].vPosition;
     u = r;
     u.Normalize();

     A = 10000;
     B = 10000;
     n = 1;
     m = 2;
     d = r.Magnitude()/Units[i].fLength;
     U = -A/pow(d, n) + B/pow(d, m);

     Fs += VRotate2D( -Units[i].fOrientation, U * u);
}

if(Avoid)
{
     for(j=0; j (_NUM_OBSTACLES; j++)
     {
          r = Units[i].vPosition - Obstacles[j];
          u = r;
          u.Normalize();
```

```
                A = 0;
                B = 13000;
                n = 1;
                m = 2.5;
                d = r.Magnitude()/Units[i].fLength;
                U = -A/pow(d, n) + B/pow(d, m);

                Fs += VRotate2D( -Units[i].fOrientation,
                                 U * u);
            }
        }

        Units[i].Fa = Fs;
        Units[i].Pa = Pfs;
        // 群聚AI结束

}
```

同样的，这里计算的作用力和先前的一样，事实上，这是我们直接从前几例的程序代码剪切过来的。

成群结队不是这个算法唯一能用的地方。你也可以用来仿真集体行为。就此而言，你得调整参数，让单位移动得更平滑一点，而不是像蜜蜂或苍蝇那样飘忽不定。

最后，我们还要提到一点，你可以结合领头者算法和这个算法，如同上一章我们在群聚算法中所做的那样。就此而言，你只需指定某个特定单位作为领头者，然后令其吸引其他单位即可。有趣的是，在此场景下，当领头者移动时，这个成群结队的群体，倾向于自我组织成某种看起来很像是上一章见过的，比较优雅的群聚行为。

关于最佳化的建议

你可能已经注意到了，当障碍物或成群结队中的单位数量增加时，我们在这里讨论的算法的计算量会变得很大。在成群结队的例子里，例5-4和5-5的简单算法是N^2阶的，显然，无法适用于大量单位的情况。因此，在游戏中实际操作这些算法时，最佳化就变得很重要。就此而言，我们要提供几条建议，让本章讨论的算法能最佳化。记住一点，这些建议是从大体而言的，实际的操作细节和你的游戏结构有密切关系。

对障碍物避开算法而言，首先能做的最佳化就是，对那些离单位很远的障碍物不要计算作用力，当作其不产生影响。你能做的就是检查给定的障碍物和该单位之间的分隔距离，如果该距离大于某些预定的距离，就跳过作用力计算。这样可以节省很多除法和指数运算。

另一种做法是把你的游戏领域分成网格，网格中的每一格都有某个固定的尺寸。然后，你可以让每一格都配置一个数组，储存每个落在该格内的障碍物数据。接着，当单位四

处行走时，你就可以轻易地确认出该单位在哪一格，然后只针对该格以及相邻格的障碍物，计算和该单位之间的作用力。现在，这些网格的实际尺寸和配置都和你的游戏有关，但一般而言，如果你的游戏领域很广，包含了许多障碍物，这种做法可以节省大量的计算资源。当然，其代价是需要更多内存来储存数据，还需要处理这些网格和障碍物之间的配置。

你也可以用这种网格方法，对成群结队算法做最佳化工作。一旦设好网格后，每一格都配有一份清单。然后，游戏循环每运行一轮时，你可以走遍储存单位的数组，确认每个单位位于哪一格。接着，把每一格的每个单位的参考点，都存进该格配置的清单里。然后，不必进入多层循环让每个单位都去比较每个单位，你只需游走每格清单中的单位，以及相邻网格的清单。同样的，处理这些配置关系也会变得更复杂，但CPU的耗用量可以节省许多。这种最佳化技巧，时常用在流体动力算法的计算里，可以有效地把N^2阶的算法差不多降到N阶。

我们能提供的最后一条建议是根据观察结果，每组单位之间的作用力大小相等但方向相反。因此，一旦你算出i和j这组单位之间的作用力时，就不必再算j和i这组了。相反的，你可以把某个力施加给i，而把该力的负值施加给j。当然，你也得记录哪些组的单位已经处理了，才不会重复施加作用力。

第六章

基本路径寻找及航点应用

寻找路径的问题有很多不同类型。没有一种解决办法可以适用各种类型的路径寻找问题。解决办法和每个游戏特定的路径寻找的需求细节有关。例如，目的地会移动还是静止不动？有没有障碍物？如果有，障碍物是否会移动？地形是什么样？最短路径解决办法是否一定是最佳解决办法？绕远路的路径，和穿越山丘或沼泽但较短的路径相比，也许更会快一点。也有可能是，路径寻找问题不需要到达某个特定的目的地。也许你只是想让某个游戏角色在游戏环境中，看似聪明地四处移动或探索。由于路径寻找问题有如此众多的类型，只选一种解决办法并不恰当。例如，A* 算法虽然是许多路径寻找问题的良方，但不适用于每种情况。本章会探索某些技巧，让你在 A* 算法不适用时使用。我们会在第七章谈重要的 A* 算法。

基本的路径寻找

从最基本的层次来讲，路径寻找只是让某个游戏角色，从其最初位置移向所需到达的目的地的过程而已。本质上，这一点和我们在第二章谈过的基本追逐算法的原理相同。例 6-1 说明了如何利用此算法达到基本的路径寻找需求。

例 6-1：基本路径寻找算法

```
if(positionX > destinationX)
       positionX--;
else if(positionX < destinationX)
       positionX++;

if(positionY > destinationY)
       positionY--;
else if(positionY < destinationY)
       positionY++;
```

此例中，游戏角色的位置以positionX和positionY变量来表示。每次这段程序执行时，positionX和positionY的坐标不是增加就是减少，使得游戏角色的位置可以移动并更靠近destinationX和destinationY坐标。这是基本路径寻找问题简单而快速的解决办法。然而，如同第二章用在追逐算法中的那样，确实有某些限制。这个方法产生的走到目的地的路径看起来并不自然。游戏角色会一直走对角线，直到其x或y轴坐标和目的地位置的坐标相同为止。然后，会走水平路径或垂直路径，直到抵达目的地。图6-1说明了可能的路径。

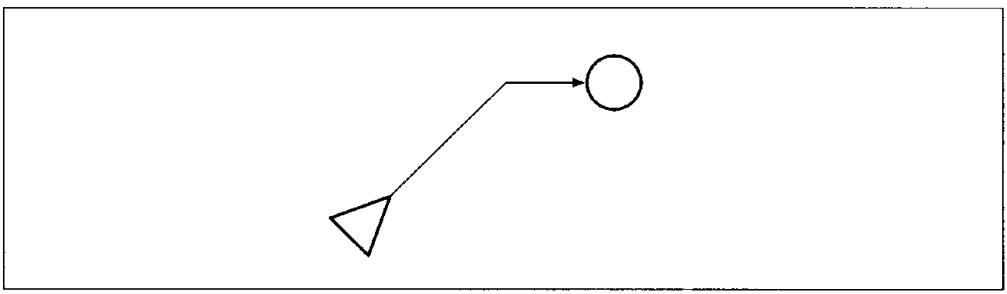


图6-1：简单的路径移动

如图6-1所示，游戏角色（三角形）会走一条很不自然的路径抵达目的地（圆形）。比较好的做法是走比较自然的视线路径。如同我们在第二章谈过的视线追逐函数，你可以用Bresenham线段算法达到这种效果。图6-2是使用Bresenham线段算法所得的视线路径，并和例6-1所用的基本路径算法所得的路径相比较。

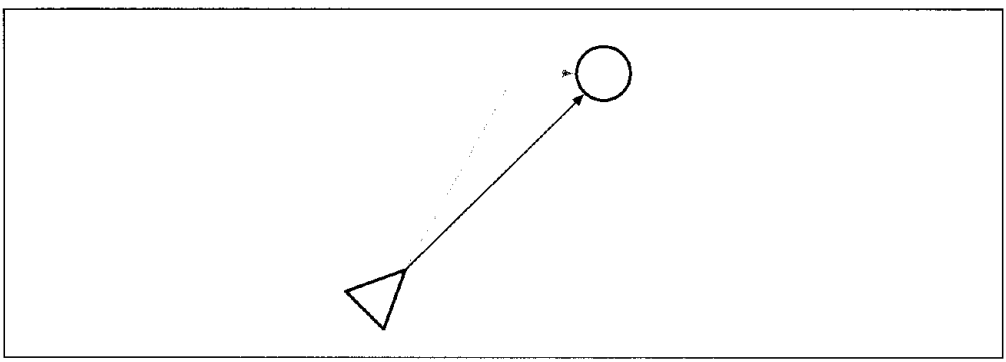


图6-2：视线路径移动

由图6-2可知，视线方法可得出一条看起来比较自然的路径。虽然视线方法确实有些优点，但这两种方法对基本路径寻找而言，都能得到精确的结果。这两种做法不但简单，而且速度相当快，所以，只要有可能，无论何时都应该予以使用。然而，这两种方法在

很多场景下却不实用。例如，游戏环境中有障碍物的时候，如图 6-3 所示，此时就需要考虑其他事项了。

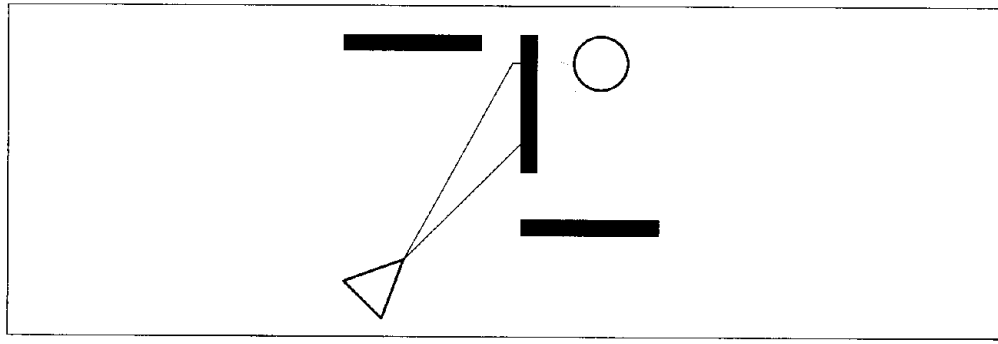


图 6-3：含障碍物的问题

随机移动避开障碍物

随机移动是简单而有效的障碍物避开方法，在那种只有少数障碍物的环境中特别有效。如图 6-4 所示，这个游戏环境中分布着树木，就是采用随机移动技巧的好对象。

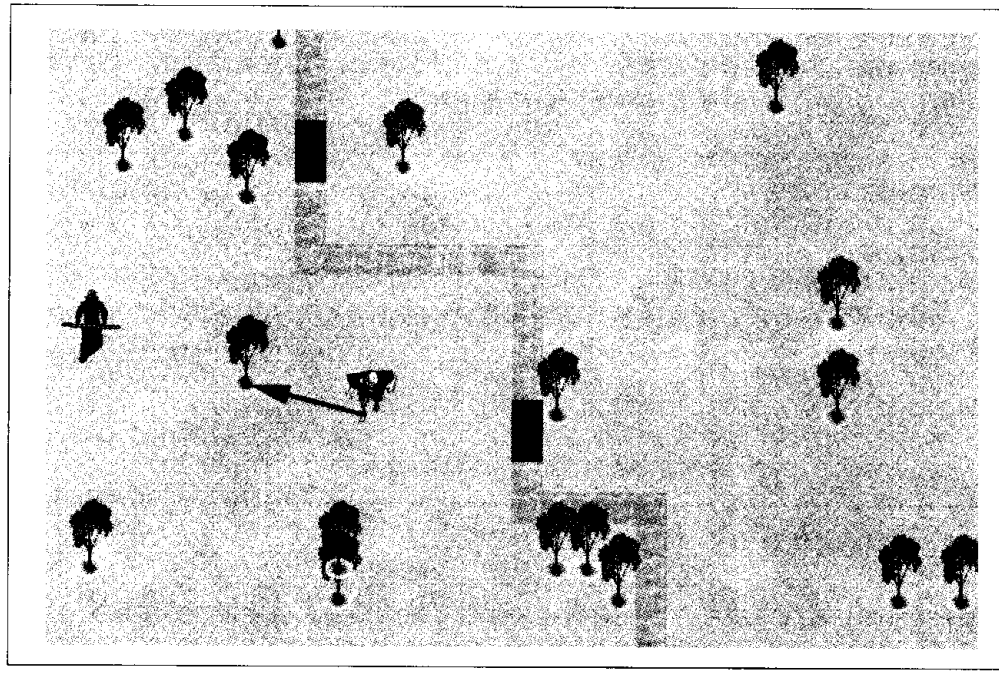


图 6-4：随机移动

如图6-4所示，玩家不在巨人的视线内。然而，由于环境中的障碍物很少，只要你随意移动一下巨人，玩家就会进入巨人的视线了。在此场景中，使用消耗大量CPU资源的路径寻找算法就过于浪费。另一方面，如果游戏环境是由许多房间组成，而每个房间之间有小通道，随机移动方式可能就不是良方了。例6-2是随机移动避开障碍物的基本算法。

例6-2：随机移动避开障碍物算法

```
if 玩家在视线内
  {
    采用直线路径走向玩家
  }

else
  {
    以随机方向移动
  }
```

如例6-2的算法所示，如果玩家在计算机控制的对手的视线内，就采用直线视线路径。如果有障碍物存在，则计算机控制的角色就采用随机方向移动。由于场景中障碍物很少，到下一轮游戏循环时，玩家可能就出现在其视线内了。

绕行障碍物

绕过障碍物是另一种相当简单的避开障碍物方法。当你要在策略游戏或角色扮演游戏中，找出一条绕过大型障碍物的路径时，比如绕过山区，这种方法就相当有效。利用此法，计算机控制的角色会采用一种简单路径寻找算法，试着抵达目标。该角色会一直沿着路径走下去，直到碰上障碍物。此时，该角色就会转换成绕行状态（tracing state）。在绕行状态下，该角色会沿着障碍物的边缘路线移动，试着沿路径经过。图6-5表示了以三角形表示假想的计算机控制角色绕行障碍物，而试图抵达以方块表示的目的地的路径。

图6-5除了显示绕行障碍物的路径外，也透露出绕行的问题之一：确定何时停止绕行。如图6-5所示，障碍物的边缘会被当成绕行路径，但也绕得太多了。事实上，几乎是绕回了起点。我们需要采用一种方式，决定何时应该从绕行状态换回简单的路径寻找状态。完成此举的做法之一，是算出从开始绕行的点到所需抵达的目的地之间的线段。计算机控制的角色会一直保持在绕行状态，直到与该线相交，到了交点时，就会换回简单路径寻找状态，如图6-6所示。

图6-5除了显示绕行障碍物的路径外，也透露出绕行的问题之一：确定何时停止绕行。如图6-5所示，障碍物的边缘会被当成绕行路径，但也绕得太多了。事实上，几乎是绕回了起点。我们需要采用一种方式，决定何时应该从绕行状态换回简单的路径寻找状态。完成此举的做法之一，是算出从开始绕行的点到所需抵达的目的地之间的线段。计算机

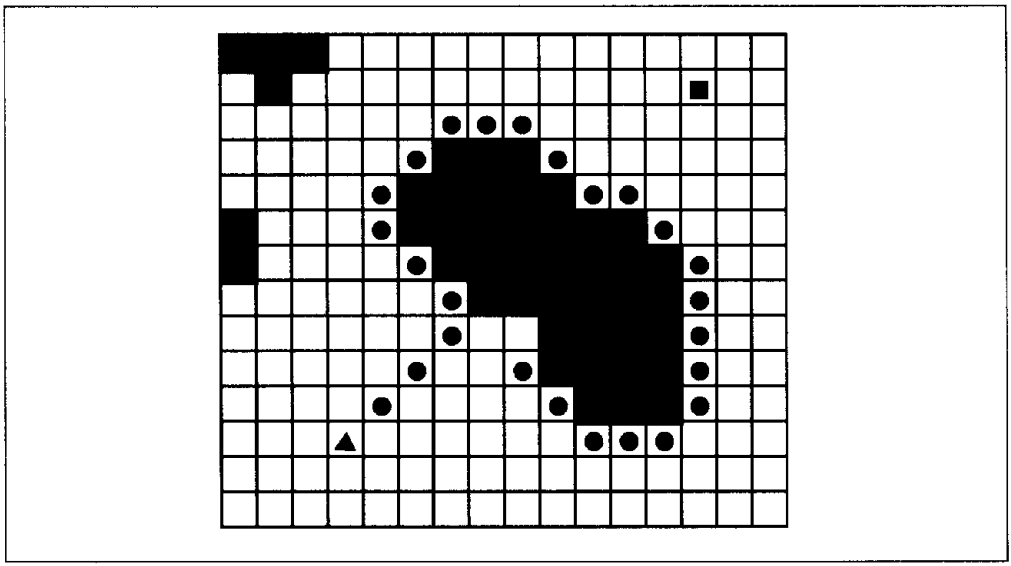


图 6-5：基本绕行

控制的角色会一直保持在绕行状态，直到与该线相交，到了交点时，就会换回简单路径寻找状态，如图 6-6 所示。

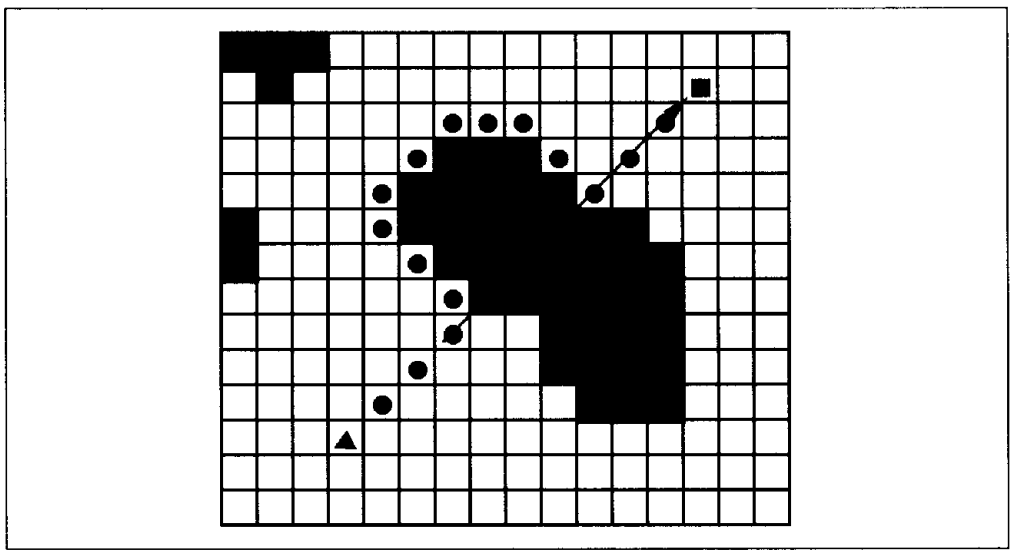


图 6-6：改良的绕行

另一种方法是在前述的绕行方法内，整合视线算法。基本上，在沿路的每一步，我们都会用视线算法，确认是否可以采用直线的视线路径抵达目的地。此法如图 6-7 所示。

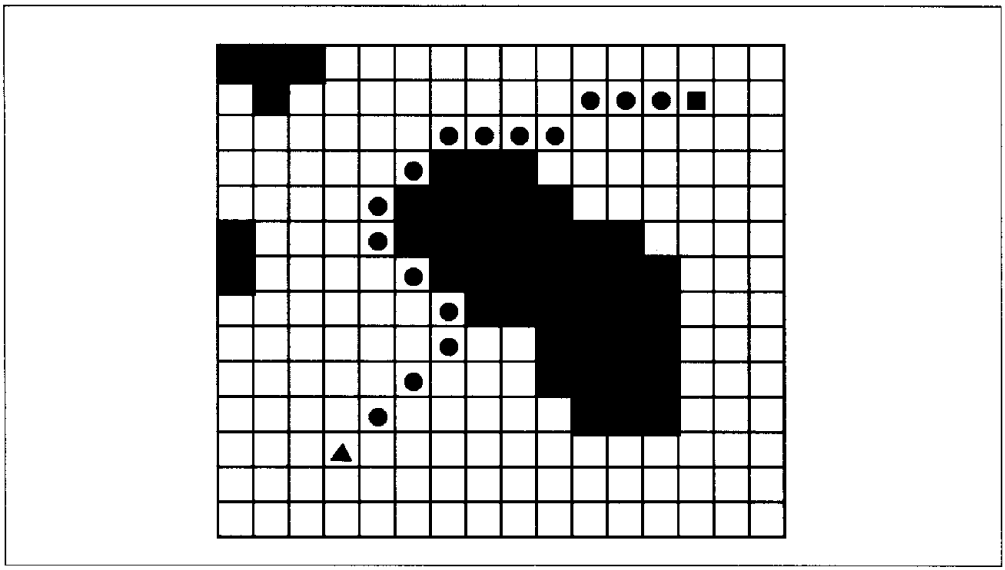


图 6-7：配合视线绕行

如图 6-7 所示，我们沿着障碍物的边缘前进，但每走一步，我们都会检查目的地是否在计算机控制角色的视线内。如果是，我们就从绕行状态切换到视线路径寻找状态。

以面包屑寻找路径

面包屑路径寻找方式可以让计算机控制角色看起来很聪明，因为是玩家在不知不觉间替计算机控制角色建立了路径。每次玩家走一步时，都会毫无所知地在游戏世界中留下看不见的标记或者说面包屑（breadcrumb）。当游戏角色碰到面包屑时，就能凭着面包屑一路走下去。游戏角色会跟着玩家的足迹，直到追上玩家。路径的复杂度以及沿路的障碍物数量根本无关紧要。玩家已经建好路径了，所以，不需要什么大不了的计算。

面包屑方法也是计算机控制角色成群移动的有效方式。你不必让群体中的每个成员，都以费时费力的路径寻找算法找路径，可以直接让成员跟着领头者留下的面包屑走。

如图 6-8 所示，玩家每走一步就会被标上一个整数值。就此而言，最多被记录下来的是 15 步。在实际游戏中，要丢下多少面包屑的数量则根据游戏类型，以及你想要让游戏控

制的角色看起来有多聪明而定。此例中，巨人会在砖块环境中随机移动，直到他在邻近位置侦测到面包屑为止。

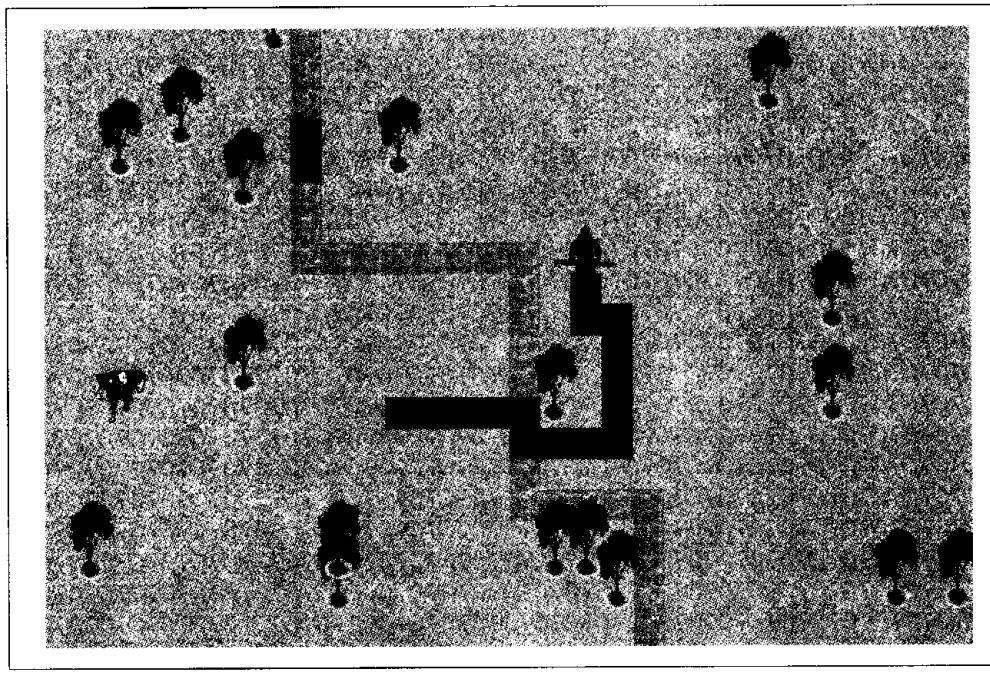


图 6-8：面包屑足迹

当然，在实际游戏中，玩家绝不会看见面包屑足迹，那是给游戏软件 AI 使用的数据。例 6-3 是我们记录和每个游戏角色有关的数据的类。

例 6-3：ai_Entity 类

```
#define   kMaxTrailLength   15

class    ai_Entity
{
       public:
       int   row;
       int   col;
       int   type;
       int   state;
       int   trailRow[kMaxTrailLength];
       int   trailCol[kMaxTrailLength];

       ai_Entity();
       ~ai_Entity();

};
```

一开始的#define命令，指定了要记录的玩家步数的最大值。然后，我们以kMaxTrailLength常量定义trailRow和trailCol数组的阈值。trailRow和trailCol数组储存的是玩家先前走过的15步的行、列坐标。

如例6-4所示，我们一开始时把trailRow和trailCol数组的每个元素指定为-1。我们用-1是因为该数值不在此砖块范例的坐标系统内。当此范例程序开始启动时，玩家还没走任何一步，所以，我们需要采用一种方式来识别trailRow和trailCol数组中有哪些元素还没被指定。

例6-4：初始化足迹数组

```
int   i;

for (i=0;i<kMaxTrailLength;i++)
   {
      trailRow[i]=-1;
      trailCol[i]=-1;
   }
```

由例6-4可知，我们绕行整个trailRow和trailCol数组，把每个元素指定为-1。现在我们可以开始记录实际的足迹了。此种做法最合理之处就是修改玩家位置的函数。这里我们要用的是KeyDown()函数，范例程序会在此处检查四个方向键，如果有某个方向键被按下的事件发生，就会更改玩家的位置。KeyDown()函数如例6-5所示。

例6-5：记录玩家位置

```
void ai_World::KeyDown(int key)
{
     int   i;

   if (key==kUpKey)
       for (i=0;i<kMaxEntities;i++)
          if (entityList[i].state==kPlayer)
              if (entityList[i].row>0)
                  {
                     entityList[i].row--;
                     DropBreadCrumb();
                  }

   if (key==kDownKey)
       for (i=0;i<kMaxEntities;i++)
          if (entityList[i].state==kPlayer)
              if (entityList[i].row<(kMaxRows-1))
                  {
                     entityList[i].row++;
                     DropBreadCrumb();
                  }

   if (key==kLeftKey)
       for (i=0;i<kMaxEntities;i++)
```

```
            if (entityList[i].state==kPlayer)
                if (entityList[i].col>0)
                   {
                      entityList[i].col--;
                      DropBreadCrumb();
                   }

   if (key==kRightKey)
       for (i=0;i<kMaxEntities;i++)
          if (entityList[i].state==kPlayer)
             if (entityList[i].col<(kMaxCols-1))
                {
                   entityList[i].col++;
                   DropBreadCrumb();
                }

}
```

例6-5的KeyDown()函数会确认玩家是否按下四个方向键的其中之一。如果是，就会走遍entityList数组，搜寻玩家所控制的角色。如果找到了，就会确定新的移动位置位于此砖块领域的边界内。如果所要的移动位置合法，位置就会被更新。下一步是实际记录位置，要调用的是DropBreadCrumb()函数，如例6-6所示。

例6-6：丢下面包屑

```
void ai_World::DropBreadCrumb(void)
{
    int   i;

    for (i=kMaxTrailLength-1;i>0;i--)
        {
           entityList[0].trailRow[i]=entityList[0].trailRow[i-1];
           entityList[0].trailCol[i]=entityList[0].trailCol[i-1];
        }
    entityList[0].trailRow[0]=entityList[0].row;
    entityList[0].trailCol[0]=entityList[0].col;

}
```

DropBreadCrumb()函数会把当前玩家位置加入trailRow和trailCol数组。这些数组含有最近玩家位置的清单。就此而言，常量kMaxTrailLength指定了将被记录的位置数量。足迹越长，计算机控制的角色越有可能发现这条足迹，并以此作为追上玩家的路径寻找过程。

DropBreadCrumb()函数一开始是把最旧的位置从trailRow和trailCol数组中删除。我们只追踪kMaxTrailLength个位置，所以每次要新增位置时，都必须把最旧的删除。我们在第一个for循环做这件事。事实上，这个循环做的是把数组中的所有位置都移位而已。循环把最旧的位置删掉，让第一个数组元素可以储存当前玩家位置。接着，我们把玩家的当前位置储存在trailRow和trailCol数组的第一个元素内。

下一步是让计算机控制角色，能实际侦测到玩家留下的面包屑足迹，并沿着该足迹走。范例程序一开始是让计算机控制的巨人在砖块环境中随机移动。图6-9是巨人在砖块环境中向八个可能方向中的任何一个方向移动的情况。

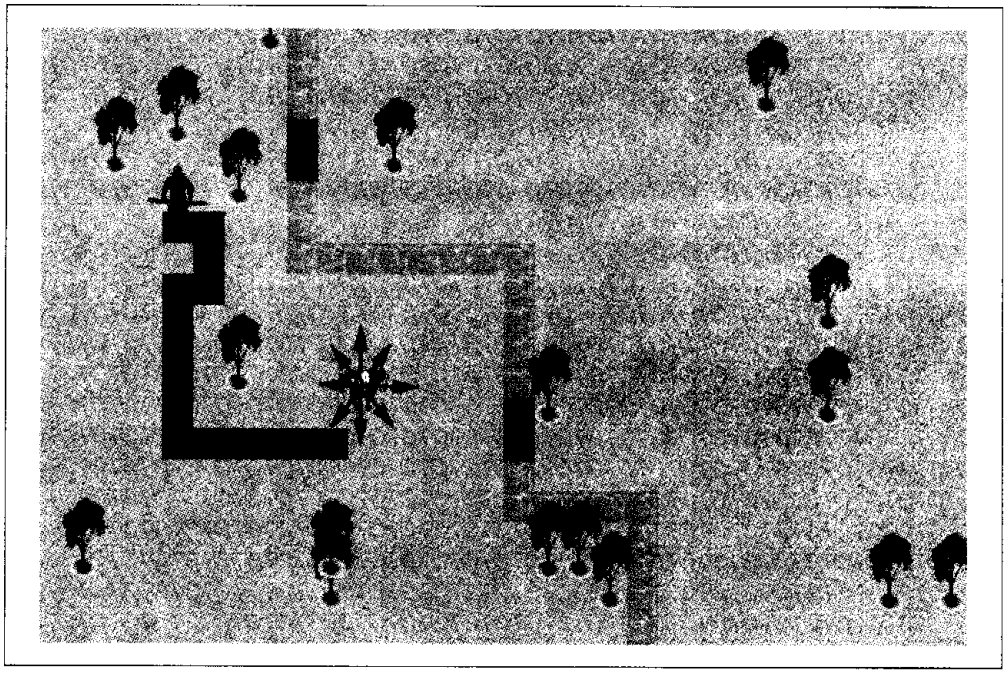


图6-9：寻找面包屑

例6-7是巨人如何侦测并跟着面包屑足迹移动的程序。

例6-7：跟着面包屑

```
for (i=0;i<kMaxEntities;i++)
    {

       r=entityList[i].row;
       c=entityList[i].col;
       foundCrumb=-1;
       for (j=0;j<kMaxTrailLength;j++)
       {
          if ((r==entityList[0].trailRow[j]) &&
              (c==entityList[0].trailCol[j]))
             {
                foundCrumb=j;
                break;
             }
          if ((r-1==entityList[0].trailRow[j]) &&
              (c-1==entityList[0].trailCol[j]))
```

```
        {
           foundCrumb=j;
           break;
        }
    if ((r-1==entityList[0].trailRow[j]) &&
        (c==entityList[0].trailCol[j]))
       {
          foundCrumb=j;
          break;
       }
    if ((r-1==entityList[0].trailRow[j]) &&
        (c+1==entityList[0].trailCol[j]))
       {
          foundCrumb=j;
          break;
       }
    if ((r==entityList[0].trailRow[j]) &&
        (c-1==entityList[0].trailCol[j]))
       {
          foundCrumb=j;
          break;
       }
    if ((r==entityList[0].trailRow[j]) &&
        (c+1==entityList[0].trailCol[j]))
       {
          foundCrumb=j;
          break;
       }
    if ((r+1==entityList[0].trailRow[j]) &&
        (c-1==entityList[0].trailCol[j]))
       {
          foundCrumb=j;
          break;
       }
    if ((r+1==entityList[0].trailRow[j]) &&
        (c==entityList[0].trailCol[j]))
       {
          foundCrumb=j;
          break;
       }
    if ((r+1==entityList[0].trailRow[j]) &&
        (c+1==entityList[0].trailCol[j]))
       {
          foundCrumb=j;
          break;
       }
}

if (foundCrumb>=0)
    {
       entityList[i].row=entityList[0].trailRow[foundCrumb];
       entityList[i].col=entityList[0].trailCol[foundCrumb];
    }
```

```
    else
        {
            entityList[i].row=entityList[i].row+Rnd(0,2)-1;
            entityList[i].col=entityList[i].col+Rnd(0,2)-1;
        }

    if (entityList[i].row<0)
        entityList[i].row=0;
    if (entityList[i].col<0)
        entityList[i].col=0;

    if (entityList[i].row>=kMaxRows)
        entityList[i].row=kMaxRows-1;
    if (entityList[i].col>=kMaxCols)
        entityList[i].col=kMaxCols-1;
}
```

我们一开始是走遍trailRow和trailCol数组。这些数组储存的是玩家最近走过的行、列位置。这些储存下来的位置是玩家留下的面包屑。实际储存的位置数量由kMaxTrailLength常量决定。我们的目标是找出计算机控制巨人所在的相邻八个可能位置中是否含有面包屑。我们用八个if语句比较每个相邻砖块和trailRow及trailCol数组中的每个元素。for循环是从数组元素0开始，因为这是玩家最新留下的位置。如果找到面包屑，其对应的数组索引值就会储存在foundCrumb这个数组索引变量中。然后，我们会让程序跳出for循环，因为我们知道不会再找到任何更接近玩家位置的面包屑了。我们知道这是因为trailRow及trailCol数组的储存顺序，是从最新的玩家位置到最旧的玩家位置。从最新玩家位置开始搜寻，也可以确保巨人跟着面包屑走向玩家，而不是越走越远。但是，巨人还是很有可能在一开始时，侦测到的是trailRow和trailCol数组中间的某个面包屑。我们必须确保巨人会跟着面包屑走向玩家。

一旦for循环完成工作之后，我们要检查是否找到了面包屑。foundCrumb变量储存的是找到的面包屑数组索引值。如果没有找到面包屑，则仍为其初值-1。如果foundCrumb大于或等于0，我们就把巨人的位置指定为trailRow及trailCol数组位于foundCrumb索引值的元素值。每次当巨人的位置需要更新时，都必须这样做一次，才能让巨人跟着玩家的足迹行进。

当我们离开for循环时，如果foundCrumb等于-1，我们就知道没有找到面包屑。就此而言，我们会随机选取一个方向令其移动。希望这样的随机移动，可以让巨人走到邻近可侦测的面包屑，以便下一次巨人的位置需要更新时，可以跟着面包屑走。

当玩家走回头路，或者和走过的路重叠，或者彼此相邻时，把玩家最新的位置储存在trailRow及trailCol数组的低位元素里，也有这个好处。这并非不常见，如图6-10所示。

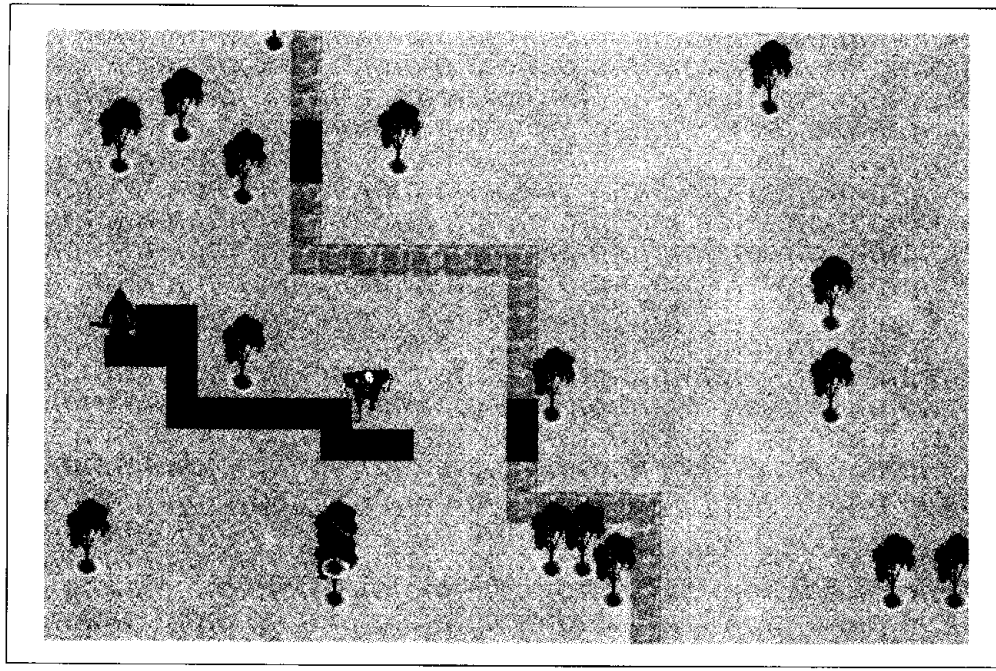


图6-10：走最短路径

就此而言，巨人不会沿着玩家的每一步行进。事实上，如果真是这样走，那会很不自然。玩家一眼就能看出，巨人是在沿着玩家的步伐前进，而不是以某种智能方式，寻找走向玩家的路径。然而，就此例而言，巨人会一直寻找含有最新面包屑的相邻砖块。最后的结果就是巨人尽可能跳过面包屑。例如，图6-10中，巨人走向玩家所在的位置时，所走的数组元素是{12, 11, 10, 9, 8, 7, 6, 2}。

遵循路径走

路径寻找通常仅被认为是从起点行走到所需到达的目的地的问题。然而，很多情况下，在游戏环境中，即使没有最终的目的地，也要让计算机控制的角色以仿真现实的方式移动。例如，在赛车游戏里，计算机控制的赛车就必须沿着赛车道前进。同样的，策略或角色扮演游戏，也许要让军队在城镇间的道路上巡逻。就像所有路径寻找问题一样，这类问题的解决办法也是和游戏环境的种类有关。在赛车游戏里，其环境就是连续的自然界。就此而言，移动起来不大可能被当成刚体。你也许会想让赛车都待在车道上，但某些情况下这是不可能的。如果有辆车试着以极高的速度转弯，就有可能冲出跑道。接着，该车会试着减速，驶回跑道的方向。

在砖块游戏环境中，你可以采取比较刚性的方式，把这种环境想象成容器问题。在此场景中，游戏角色受限制于特定的地形元素，比如道路。图 6-11 说明了此例。

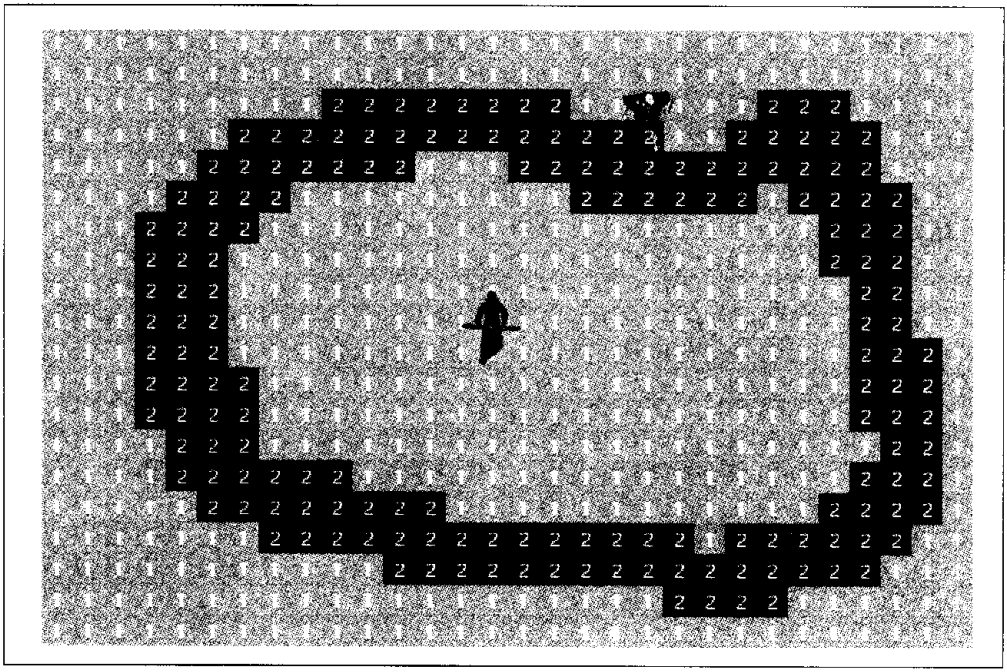


图 6-11：沿着道路走

在图 6-11 中，所有标示为 2 的地形元素都视为道路。标示为 1 的地形元素就视为道路边界以外的区域。所以，这个问题就变成了让计算机控制的巨人，待在此地形里的道路区域内。然而，我们不想让巨人只在道路上随机移动。这种结果看起来不自然。我们想让巨人看起来好像是沿着道路在行走。你也注意到，道路宽度不只有一块砖，所以，这个问题不仅仅是寻找相邻砖块中含有 2 的地方去移动。我们必须分析周围的地形，再决定最佳移动地点。砖块环境中，通常在移动时有八种可能方向可供选择。我们会检视这八个方向，再把不属于道路的部分剔除掉。然后，这个问题就变成了从剩下的方向中，决定出某一个方向去行走。例 6-8 说明了我们如何开始分析周围地形。

例 6-8：*地形分析*

```
int   r;
int   c;
int   terrainAnalysis[9];
r=entityList[i].row;
c=entityList[i].col;
```

```
terrainAnalysis[1]=terrain[r-1][c-1];
terrainAnalysis[2]=terrain[r-1][c];
terrainAnalysis[3]=terrain[r-1][c+1];
terrainAnalysis[4]=terrain[r][c+1];
terrainAnalysis[5]=terrain[r+1][c+1];
terrainAnalysis[6]=terrain[r+1][c];
terrainAnalysis[7]=terrain[r+1][c-1];
terrainAnalysis[8]=terrain[r][c-1];

for (j=1;j<=8;j++)
   if (terrainAnalysis[j]==1)
      terrainAnalysis[j]=0;
   else
      terrainAnalysis[j]=10;
```

一开始我们定义了terrainAnalysis数组。这是我们存储计算机控制巨人相邻八个砖块地形的数值的地方。我们的做法是对巨人当前的行、列位置做偏移。当这八个值都被储存之后，就进入for循环，确认是否每个值都是道路的一部分。如果不是道路的一部分，相对应的terrainAnalysis数组元素就指定为0。如果是道路的一部分，terrainAnalysis数组元素就指定为10。

现在，我们知道有哪些可能的方向了，再来考虑目前要移往哪个方向。我们想让巨人都朝着大致相同的方向前进。我们只想在必须转弯时才转弯，而到了要转弯的时候，决定左转或右转，是以能否完全改变方向为依据。就此范例程序而言，我们为每个方向分配了一个数字，让我们能记录当前方向。图6-12是配置给每个方向的数字。

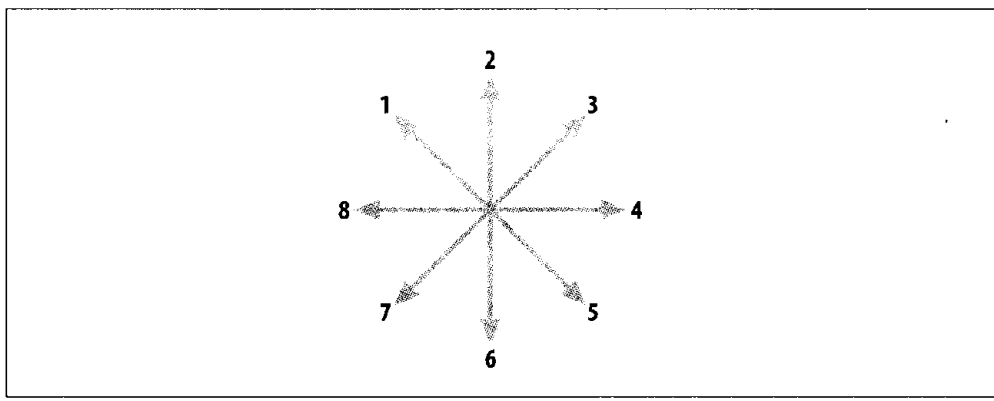


图6-12：可能的方向

我们将以图6-12的数字，记录每次更新巨人位置时所移动的方向。这样就可以在更新巨人位置时，给先前的方向增加权重。例6-9说明了该做法。

例6-9：方向分析

```
if (entityList[i].direction--1)
    {
      terrainAnalysis[1]=terrainAnalysis[1]+2;
      terrainAnalysis[2]++;
      terrainAnalysis[5]--;
      terrainAnalysis[8]+-;
    }
if (entityList[i].direction==2)
   {
      terrainAnalysis[1]++;
      terrainAnalysis[2]=terrainAnalysis[2]+2;
      terrainAnalysis[3]-+;
      terrainAnalysis[6]--;
   }
if (entityList[i].direction==3)
   {
      terrainAnalysis[2]++;
      terrainAnalysis[3]=terrainAnalysis[3]-2;
      terrainAnalysis[4]++;
      terrainAnalysis[7]--;
   }
if (entityList[i].direction==4)
   {
      terrainAnalysis[3]++;
      terrainAnalysis[4]=terrainAnalysis[4]+2;
      terrainAnalysis[5]-+;
      terrainAnalysis[8]--;
   }
if (entityList[i].direction==5)
   {
      terrainAnalysis[4]++;
      terrainAnalysis[5]=terrainAnalysis[5]+2;
      terrainAnalysis[6]++;
      terrainAnalysis[1]--;
   }
if (entityList[i].direction==6)
   {
      terrainAnalysis[2]--;
      terrainAnalysis[5]++;
      terrainAnalysis[6]=terrainAnalysis[6]+2;
      terrainAnalysis[7]++;
   }
if (entityList[i].direction==7)
   {
      terrainAnalysis[3]--;
      terrainAnalysis[6]++;
      terrainAnalysis[7]=terrainAnalysis[7]+2;
      terrainAnalysis[8]++;
   }
if (entityList[i].direction==8)
   {
```

```
        terrainAnalysis[1]++;
        terrainAnalysis[4]--;
        terrainAnalysis[7]++;
        terrainAnalysis[8]=terrainAnalysis[8]+2;
    }
```

每个if语句都会利用储存在entityList[i].direction中的当前方向，递增或递减terrainAnalysis数组的值。这样会让某些潜在的方向更能受到青睐，同时让其他方向更不受青睐。例如，第一个if语句会检查前一方向为1的情况，也就是往左上方的移动。如果前一方向是1，我们就把terrainAnalysis数组的1元素的权重提高，做法是加上2以增加其值。当然，一直往左上方移动是不可能的，也许已走到路的边缘。所以，我们得考虑剩余的可能性。接下来两个最佳可能性是往上走或直接左走。我们也要替这两个方向增加权重，所以，要把terrainAnalysis数组的2和8元素的值都加1。元素3、4、6以及7则视为不相干，所以没有改变这些元素在terrainAnalysis数组中的值。然而，最后剩下的方向是不受青睐的，就此而言，方向5是和当前方向完全相反的。这个方向将是我们最后的选择，所以，我们把terrainAnalysis数组中该元素的权重减1。此例的说明如图6-13所示。

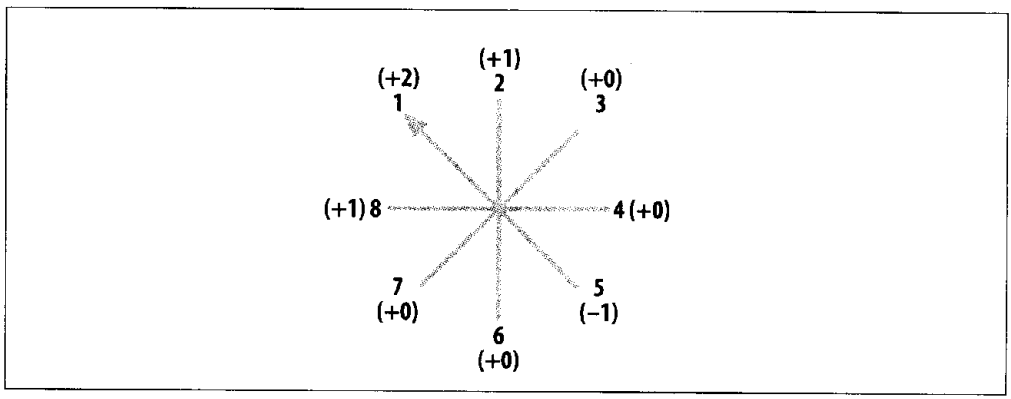


图6-13：替各个方向分配权重

如图6-13所示，当前方向为1（左上）。每个方向皆平等，但我们想让巨人持续朝该方向前进，所以，其terrainAnalysis数组的元素权重加了2。另外两个可能的方向是2和8，因为这两个方向是最轻微的转向，权重都加了1。剩余的元素则不予考虑，除了完全相反的5这个方向外。下一步是找出最佳方向，如例6-10所示。

例6-10：选择一个方向

```
maxTerrain=0;
maxIndex=0;
for (j=1;j<=8;j++)
   if    (terrainAnalysis[j]>maxTerrain)
```

```
        {
           maxTerrain=terrainAnalysis[j];
           maxIndex=j;
        }
```

如例6-10所示，我们绕行terrainAnalysis数组，找出可能方向中权重最高的一个。跳出for循环前，maxIndex变量所含的数组索引值，就是代表权重最高的方向的元素。例6-11说明了我们如何使用maxIndex变量的值更新巨人位置。

例6-11：更新位置

```
if (maxIndex==1)
   {
      entityList[i].direction=1;
      entityList[i].row--;
      entityList[i].col--;
   }
if (maxIndex==2)
   {
      entityList[i].direction=2;
      entityList[i].row--;
   }
if (maxIndex==3)
   {
      entityList[i].direction=3;
      entityList[i].row--;
      entityList[i].col++;
   }
if (maxIndex==4)
   {
      entityList[i].direction=4;
      entityList[i].col++;
   }
if (maxIndex==5)
   {
      entityList[i].direction=5;
      entityList[i].row++;
      entityList[i].col++;
   }
if (maxIndex==6)
   {
      entityList[i].direction=6;
      entityList[i].row++;
   }
if (maxIndex==7)
   {
      entityList[i].direction=7;
      entityList[i].row++;
      entityList[i].col--;
   }
if (maxIndex==8)
   {
```

```
        entityList[i].direction=8;
        entityList[i].col --;
    }
```

maxIndex的值指出了巨人的新方向。我们对八个可能方向都以if语句加以判断。一旦找出所要的方向时，我们就更新entityList[i].direction的值，下一次巨人位置需要更新时，这个方向就会成为前一个方向。然后，我们更新entityList[i].row和entityList[i].col的值。图6-14是巨人沿着道路走的路径。

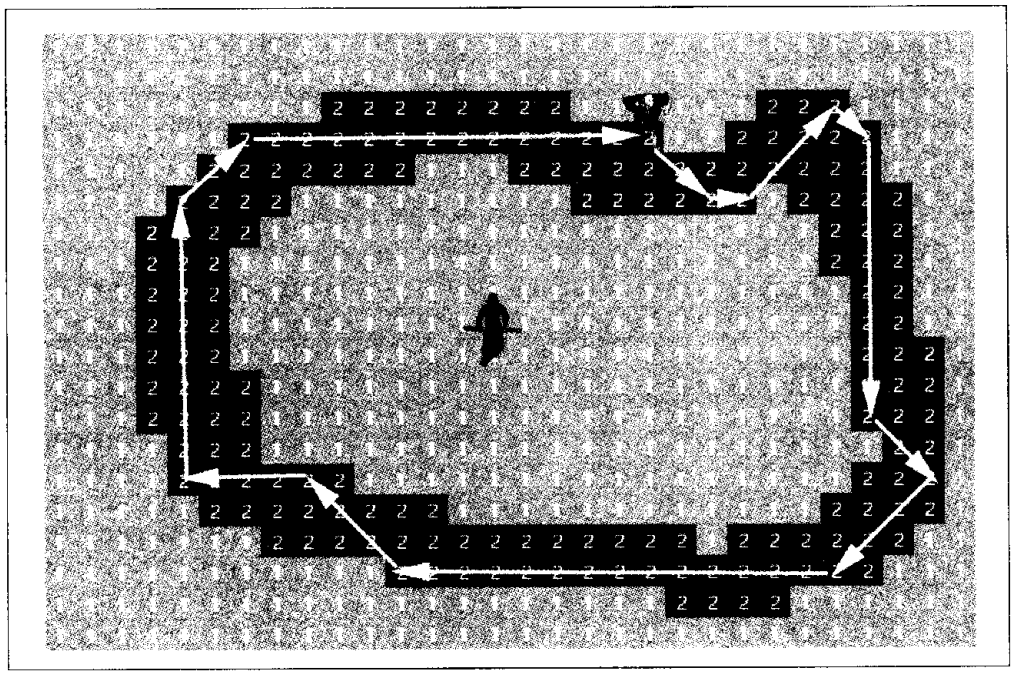


图6-14：道路的路径

如图6-14所示，巨人会持续沿着这条路走。在实际游戏中，你可以让此计算机控制的对手持续在道路上巡逻，直到遇见玩家。到时候，计算机控制角色的状态就要切换成攻击模式。

此例中，我们使用相邻砖块做出加权的方向，找出接下来要移动的方向。你也可以多检视相邻砖块以外的砖块，让这种技巧更加强大。除了相邻砖块以外，你也可以替相邻砖块以外的砖块加上权重。这样会让移动看起来更自然更聪明。

沿着墙走

开发游戏时，另一种有用的路径寻找方式就是沿着墙走（wall tracing）。和遵循道路走一样，这种方法也不会去算起点和终点之间的路径。沿着墙走更像是探索的技巧。在那种由许多小房间构成的游戏环境中最有用，不过你也可以在类似迷宫的游戏环境中使用。你也可以利用绕行障碍物的基本算法，在上一节讨论绕行障碍物时提到过。游戏很少让每个计算机控制的敌人，同时寻找路径往玩家方向走去。有时候，必须让计算机控制的角色探索环境，以搜寻玩家、武器、燃料、宝物或任何游戏角色可以接触的东西。让计算机控制角色在环境中随机移动，是游戏开发人员最常用的办法。这样会造成某种程度的不可预测性，但也有可能让计算机控制角色，长时间被困在某些小房间内。图6-15是城堡里的某一层，由许多小房间构成。

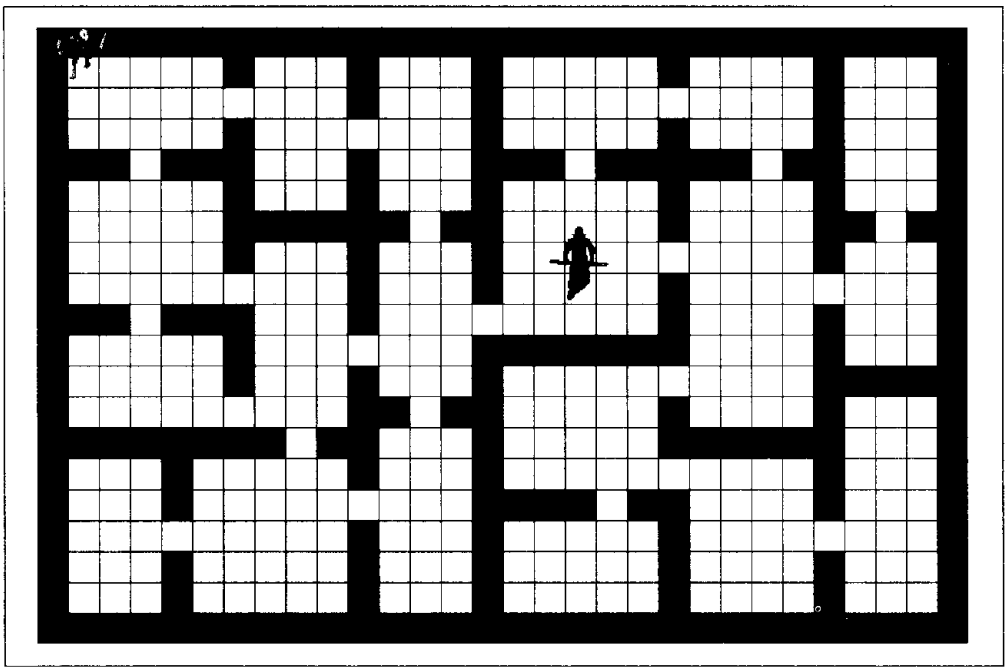


图6-15：沿着墙走

在图6-15的范例中，我们可以让巨人往随机方向移动。然而，这可能会让巨人花点时间才能离开左上角的房间。比较好的做法是让巨人系统地探索整个环境。幸运的是，有一个相当简单的办法可以使用。基本上，我们打算采用左侧移动法。如果巨人总是往左移，就会对其环境做完整的探索工作。重点是要记住巨人必须尽可能往左移，这个左侧不一定是玩家的左侧。图6-16说明了这一点。

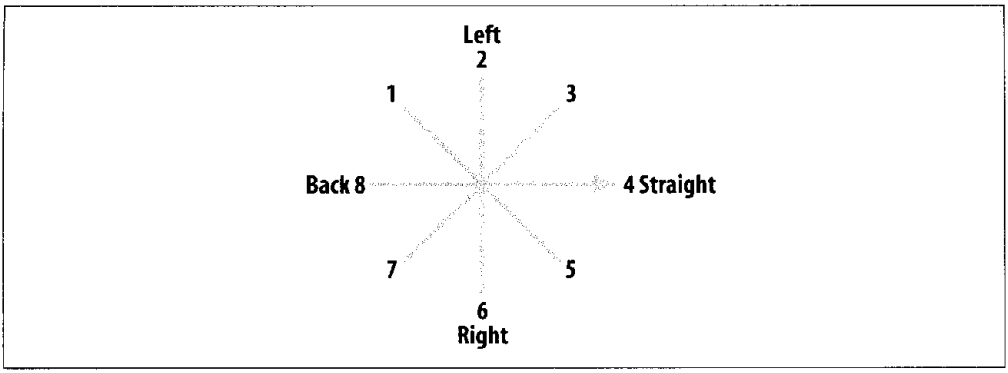


图 6-16：面对玩家的右侧

范例程序开始时，巨人面对的是玩家的右侧，也就是面对标示为4的方向，如图6-16所示。这也就是说方向2是巨人的左侧，方向6是巨人的右侧，而方向8是巨人的背后。采用左侧移动法时，巨人总是试着先往左边走。如果无法往左走，会试着往前走；如果也走不通，就会试着往右走；如果还走不通，就走回头。当此例程序开始时，从玩家的观点来看，巨人会试着往上走。如图6-15所示，有道墙挡住去路，所以巨人必须改为直走。从巨人的观点来看，就是方向4。巨人前方没有障碍物，所以可以移动。这种左侧移动法如例6-12所示。

例 6-12：*左侧移动*

```
r=entityList[i].row;
c=entityList[i].col;

if (entityList[i].direction==4)
    {
       if (terrain[r-1][c]==1)
          {
             entityList[i].row--;
             entityList[i].direction=2;
          }
       else if (terrain[r][c+1]==1)
          {
             entityList[i].col++;
             entityList[i].direction=4;
          }
      else if (terrain[r-1][c]==1)
          {
             entityList[i].row++;
             entityList[i].direction=6;
          }
      else if (terrain[r][c-1]==1)
          {
             entityList[i].col--;
```

```
            entityList[i].direction=8;
         }
   }

else if (entityList[i].direction==6)
   {
      if (terrain[r][c+1]==1)
         {
            entityList[i].col++;
            entityList[i].direction=4;
         }
      else if (terrain[r+1][c]==1)
         {
            entityList[i].row++;
            entityList[i].direction=6;
         }
      else if (terrain[r][c-1]==1)
         {
            entityList[i].col--;
            entityList[i].direction=8;
         }
      else if (terrain[r-1][c]==1)
         {
            entityList[i].row--;
            entityList[i].direction=2;
         }
   }

else if (entityList[i].direction==8)
   {
      if (terrain[r+1][c]==1)
         {
            entityList[i].row++;
            entityList[i].direction=6;
         }
      else if (terrain[r][c-1]==1)
         {
            entityList[i].col--;
            entityList[i].direction=8;
         }
      else if (terrain[r-1][c]==1)
         {
            entityList[i].row--;
            entityList[i].direction=2;
         }
      else if (terrain[r][c+1]==1)
         {
            entityList[i].col++;
            entityList[i].direction=4;
         }
   }

else if (entityList[i].direction==2)
   {
```

```
    if (terrain[r][c-1]==1)
       {
          entityList[i].col--;
          entityList[i].direction=8;
       }
    else if (terrain[r-1][c]==1)
       {
          entityList[i].row--;
          entityList[i].direction=2;
       }
    else if (terrain[r][c+1]==1)
       {
          entityList[i].col++;
          entityList[i].direction=4;
       }
    else if (terrain[r+1][c]==1)
       {
          entityList[i].row++;
          entityList[i].direction=6;
       }
}
```

例6-12有四个if语句区块。我们必须为巨人面对的四个可能的方向，分别准备不同的if区块。这是必要的，因为巨人左方的砖块是什么，由巨人所面对的方向决定，如图6-17所示。

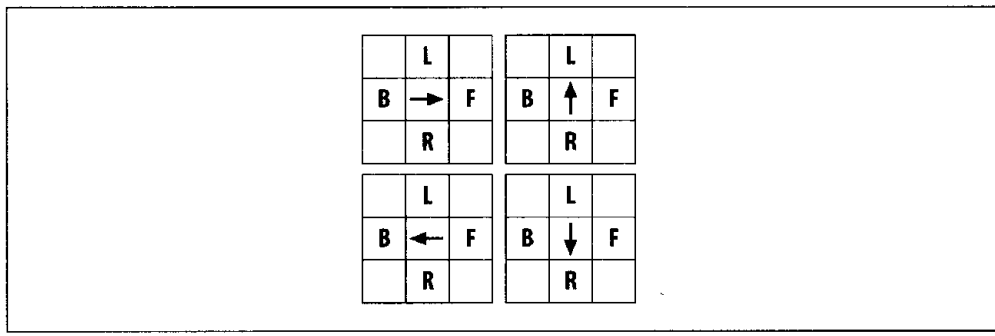


图6-17：相对方向

如图6-17所示，如果以玩家的角度来看，巨人面对右侧，则其左侧就是位于其上的砖块。如果巨人面对上方，则其左侧的砖块，实际上就是左边的砖块。如果其面对左侧，则下面的砖块就是其左侧。最后，如果其面对下方，则右边的砖块才是巨人的左侧。

如第一个if区块所示，如果巨人面对方向4，会先检查其左侧的砖块，也就是terrain[r-1][c]。如果此位置的值为1，则表示无障碍物。此时，巨人的位置会被更新，更重要的是，其方向会更新为2。也就是说从玩家的观点来看，巨人现在面对的是

上方。下一次此段程序执行时，因为其方向已改变了，所以会用到另一个 if 区块。其他的 if 区块则可以确保每个可能的方向，都会遵循相同的步骤。

如果检查 terrain[r-1][c]时侦测到障碍物，则接着会检查巨人前方的砖块。如果也有障碍物，则会检查巨人右侧的砖块，最后是其背后的砖块。最后的结果就是巨人探索了整个游戏环境，如图 6-18 所示。

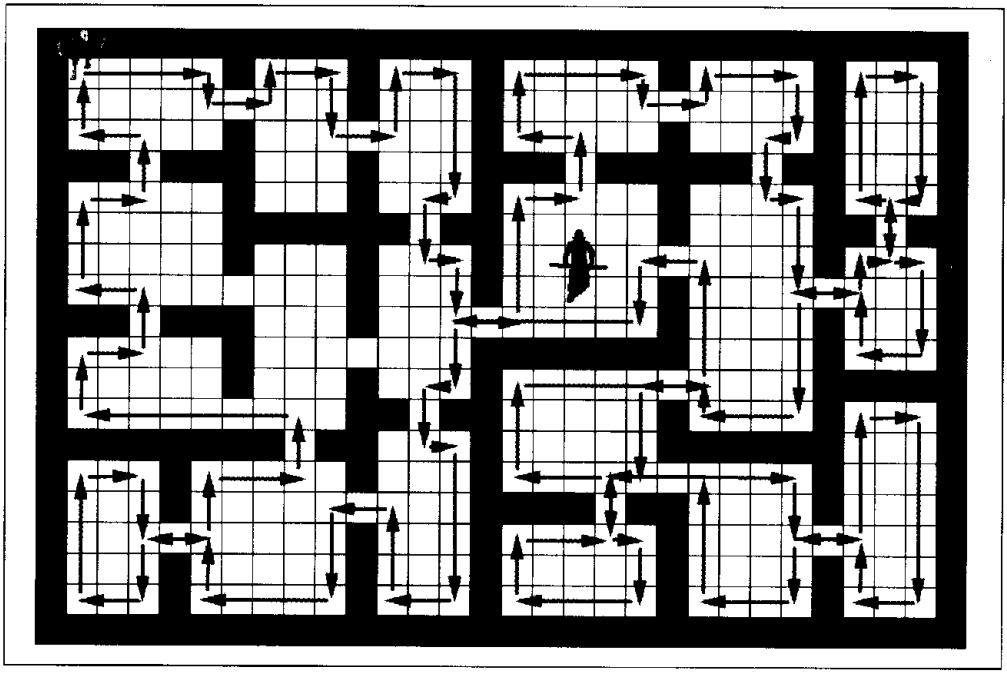


图 6-18：沿着墙走的路径

由此可见，采用左侧移动法时，巨人会进入到游戏环境中的每个房间。虽然这种做法在概念上很简单，而且多数情况下都非常有效，但并不能保证一切情况都适用。有些几何环境会使得这种方法无法让巨人抵达每个房间。

航点导航

路径寻找是一项非常耗时，且耗用CPU资源的运算工作。减少这种困扰的方式之一，就是尽可能预先算好路径。航点导航（waypoint navigation）减少这种困扰的做法就是，认真地在游戏环境中置放节点，然后，使用预先计算好的路径，或者是简单的路径寻找方法在节点之间移动。图 6-19 说明了如何在一张由七个房间构成的简单地图上置放节点。

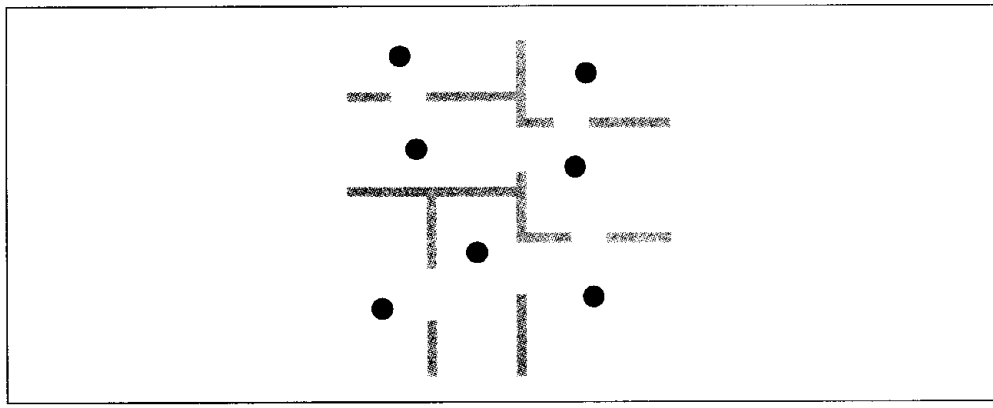


图6-19：置放节点

在图6-19中，注意图中的每个点，至少都能让另一个节点的视线看见。也就是说，每个节点也至少都在另一个节点的视线内。对以这种方式构成的游戏环境而言，游戏控制角色总是能使用简单的视线算法，到达图中的任何地点。游戏软件AI只需要知道这些节点之间的关系。图6-20说明了如何标示节点，并让图中的每个节点能连在一起。

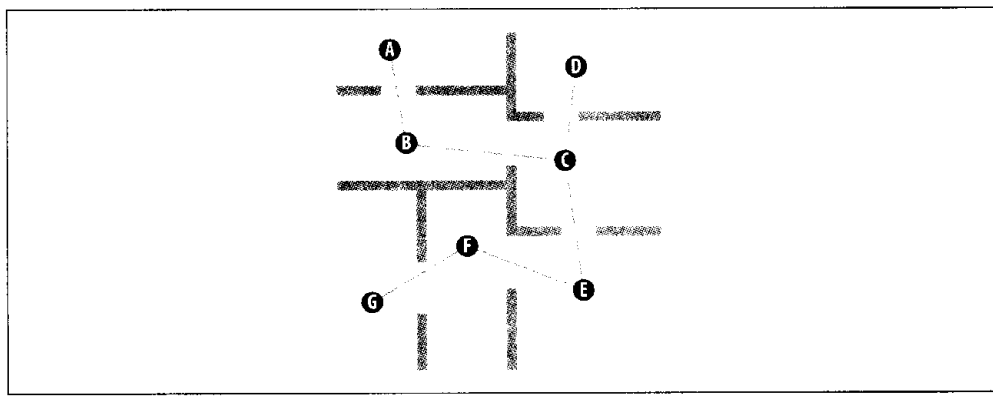


图6-20：标示节点

利用图6-20的节点标示符号以及连线，现在我们就能找出房间之间的路径了。例如，要从含有节点A的房间，走到含有节点E的房间，就必须走过节点ABCE。节点间的路径则由视线算法求算，或者也可以是一连串预先计算好的步伐。图6-21显示了由三角形表示的计算机控制角色，抵达由方形表示的玩家控制角色的路径的过程。

计算机控制角色先计算哪一个节点和他当前位置最接近，而且在其视线之内。在此例中，那个节点就是A。然后，该角色会计算哪一个节点最接近玩家当前位置，而且在玩家视

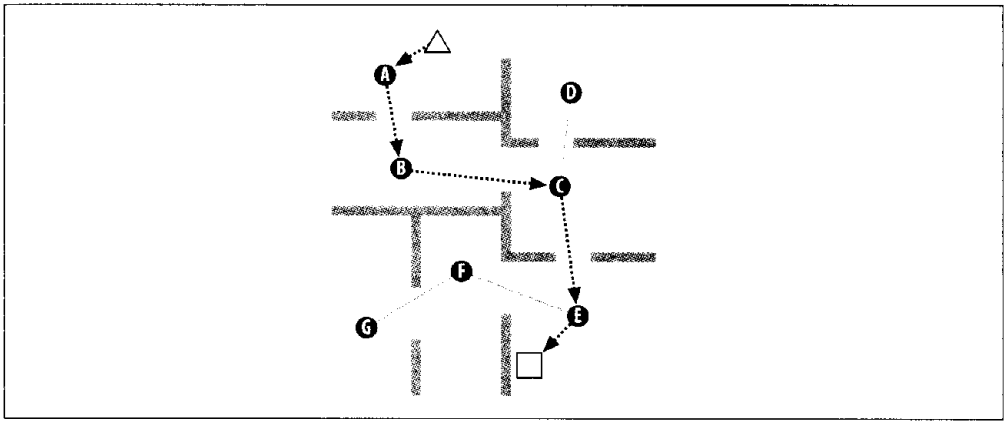


图 6-21：建立路径

线之内。那就是节点 E。然后，计算机会规划出从其当前位置到节点 A 的路径。接着，利用节点的连接关系，从节点 A 走到节点 E。就此例而言，就是 A → B → C → E。一旦抵达最终节点，就可以再算出最终节点到玩家之处的视线路径了。

这看起来似乎很简单，但计算机怎么知道要走哪些节点呢？换言之，计算机怎么知道要从节点 A 走到节点 E，就必须先经过 B 和 C？为找到答案，让我们利用一张简单的数据表格，快速而轻易地找出任意两节点间的最短路径。图 6-22 是最初空白的节点连接表。

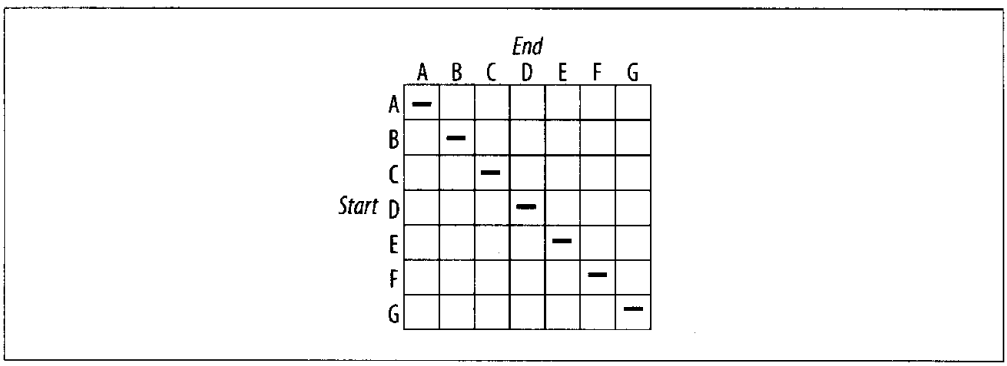


图 6-22：空白的节点表

这张表的目的是建立节点间的连接关系。填写这张表是很简单的事，要从任意起点移到任意终点时，只要找出最先要去的那个点就行。起点都列在这张表的左侧，而终点则放在顶端。我们只要找出表中起点和终点的交叉点，就能求出最佳路径。你会注意到，表中对角线那些格子都放了横线。这些表格元素不需要填，因为起点和终点都是同一点。例如，左上角的表格元素即属此类，起点和终点都是 A，你不用从节点 A 移到节点 A，

所以这个表格元素不必理会。然而，最顶端那一行该表格元素旁边的那个表格元素的起点是A，而终点却是B。现在，我们根据图6-21来决定从节点A走到节点B所需的第一步。在此例中，下一步就是移到节点B，所以，我们在最顶端那一行的第二个元素内填入B。最顶端那一行的第三个表格元素是从节点A走到节点C。同样的，图6-21让我们知道第一步是移到节点B。填这张表时，我们不必理会任意两节点间的完整路径。我们只需确认从任意节点移到其他任意节点时，最先要去的那个节点即可。图6-23显示了表格中完成的第一行。

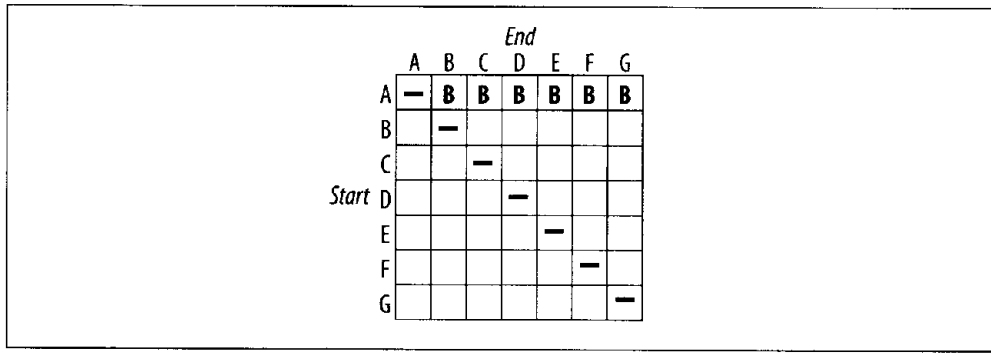


图6-23：填写节点表

从图6-23中我们知道，当你要从节点A移动到任意其他节点时，都必须先到节点B。检视图6-21时，可确认此事实。唯一和节点A相连的就是节点B，所以，要从节点A移到其他任意节点时，我们必须先通过节点B。要从节点A移到节点E，只知道必须先移到节点B，还是不能到达目的地。我们必须填完这张表。看到表中的第二行，我们发现要从节点B移到节点A，只需直接移到节点A。从节点B移到节点C，只需直接移到节点C。我们一直这样做下去，直到表中每一格都填完为止。图6-24就是完成后的节点连接表。

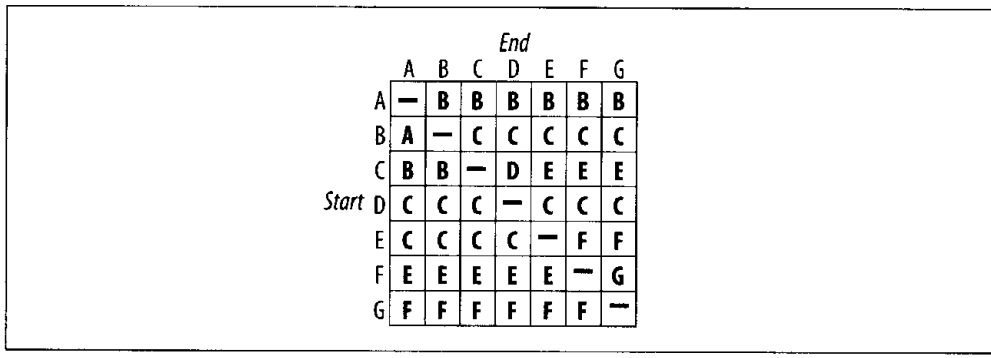


图6-24：完成的节点表

利用图 6-24 完成的表格，我们就能确认出从任意点到其他任意点时要走的路径了。图 6-25 是其中一条可走的路径实例。此图中，假想的计算机控制角色（由三角形表示）想建立一条走到玩家（由方形表示）的路径。

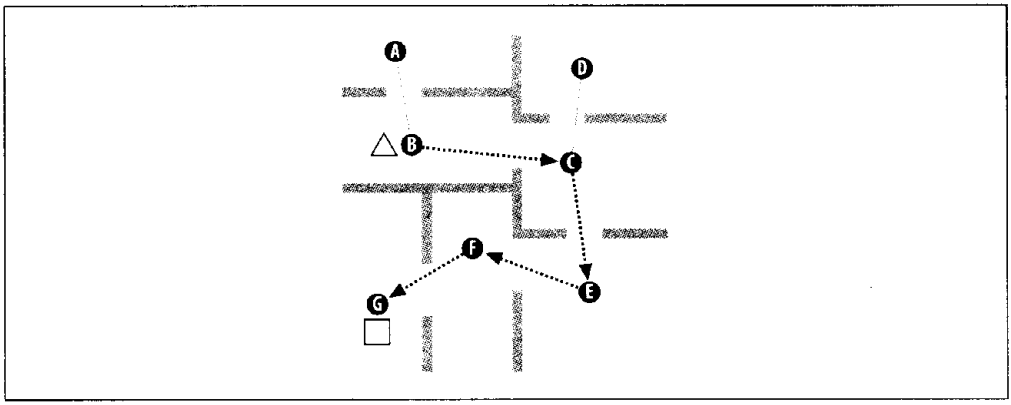


图 6-25：寻找路径

要建立此路径，我们只需用图 6-24 所完成的节点连接表。如图 6-25 所示，我们想建立从节点 B 到节点 G 的路径。一开始是找表中节点 B 和节点 G 的交叉点。表中显示节点 C 是该交叉点。所以，要从节点 B 移到节点 G 时，第一条要穿越的连线是 B → C。一旦我们抵达节点 C 之后，可再引用表格，找出节点 C 和所要到达目的地节点 G 之间的交叉点。此时，我们发现节点 E 是其交叉点。然后，我们继续走 C → E。我们重复此过程，直到抵达目的地。图 6-26 是建立从节点 B 走到节点 G 时，所经过的各个路径线段。

Goal B → G		
B → G	Table intersection → C	Move → C
C → G	Table intersection → E	Move → E
E → G	Table intersection → F	Move → F
F → G	Table intersection → G	Move → G

图 6-26：寻找路径

如图 6-26 所示，此例中计算机控制角色必须走四段路，才能抵达目的地。

我们在这里讨论的每种方法都有其优缺点，显然地，没有一种方法可适用所有可能的路径寻找问题。本章一开始提到的那种方法是A*算法，可适用大部分的路径寻找问题。A*算法是游戏中极为常用的路径寻找算法，下一章将专门讨论这种方法。

第七章

A*路径寻找算法

本章要讨论A*路径寻找算法的基本原理。路径寻找是游戏软件AI最基本的问题之一。路径寻找设计得不好会让游戏角色看起来很愚笨很不真实。要是看见游戏角色无法越过一群简单的障碍物，再也没有比此更令人对游戏产生厌倦的了。有效地处理路径寻找问题，可以让游戏变得更有趣，让玩家沉迷其中。

幸运的是，A*算法可以提供有效的办法来解决路径寻找问题。即使A*算法不是当今游戏软件开发中最为常用的路径寻找算法，也可能是相当常用的一种。A*算法之所以会如此吸引人，是因为它可以保证在任何起点及任何终点间找到最佳的路径，当然，前提是确实存在这种路径。此外，A*是相当有效的算法，因而更增添其魅力。事实上，你应该尽可能予以使用，当然，除非你是在处理某种特殊情况的场景。例如，如果起点和终点间没有障碍物，有明确的视线，那么用A*算法就大可不必了；改用既快速又有效的视线移动算法比较好。如果CPU功能不太强，A*也可能不是最佳替代方案。虽A*很有效，但仍然会耗用不少CPU运算能力，尤其是，如果你必须同时为许多游戏角色寻找路径的时候。然而，就多数路径寻找问题而言，A*是最佳选择。可惜的是，要了解A*算法如何运作，对游戏开发新手而言有一定难度。本章我们要循序渐进地讨论A*算法的内部运作过程，了解A*算法如何在起点和终点间建立路径。观察A*路径一步一步地建造出来，应该有助于揭开A*算法的神奇之处。

定义搜寻区域

路径寻找的第一步是定义搜寻区域。我们需要以某种方式表示游戏世界，让搜寻算法能借此予以搜寻，并找出最佳路径。总的来说，游戏世界必须由点来表示，让游戏角色和物体能占据其上，而找出任意两点的最佳路径并避开障碍物，这就是路径寻找算法的工

作。至于游戏世界到底以什么方式表示，则和游戏的类型有关。就某些情况而言，游戏世界必须简化。例如，游戏若使用连续环境，则可能是由大量的点构成的，让游戏角色能占据其上。对这类搜寻空间而言，用A*算法就不切实际了，因为点的数量太大了。然而，如果搜寻区域可以简化，也许还能运用。这会牵涉到在整个游戏世界中的节点置放。接着，我们就可以在节点间建立路径，不一定在游戏世界里任何可能的点之间都建立路径。这一点以图7-1来表示。

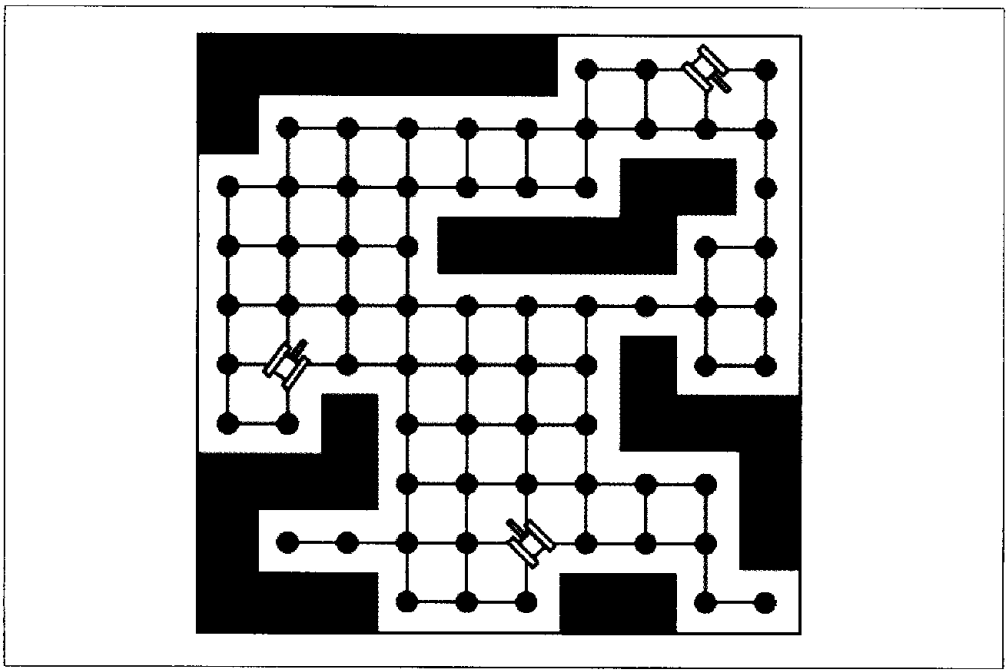


图7-1：简化搜寻区域

图7-1中的坦克可以根据其坐标系统而占据任何点位，但就路径寻找的目的而言，游戏世界可在整个游戏环境中置放节点而予以简化。这些节点并不直接对应任何可能的坦克位置，这样就需要太多节点了。我们必须把节点减少到可控制的数量内，这就是我们所说的必须简化搜寻区域。

当然，我们必须持有节点间连接关系的数据清单。搜寻算法必须知道节点是如何连接的。一旦知道节点如何连接，A*算法就能算出从任何节点到其他任何节点的路径。游戏世界中节点放得越多，路径寻找的过程就越慢。如果路径寻找时耗用了太多CPU运算资源，另一种可替代的做法就是通过少用一些节点来简化搜寻区域。

另一方面，使用砖块世界的游戏就是A* 算法的合适对象，当然，前提是这个游戏世界不会大到太离谱。因为这种世界本质上已经分成节点了，每个砖块就是搜寻区域里的节点。其说明如图 7-2 所示。

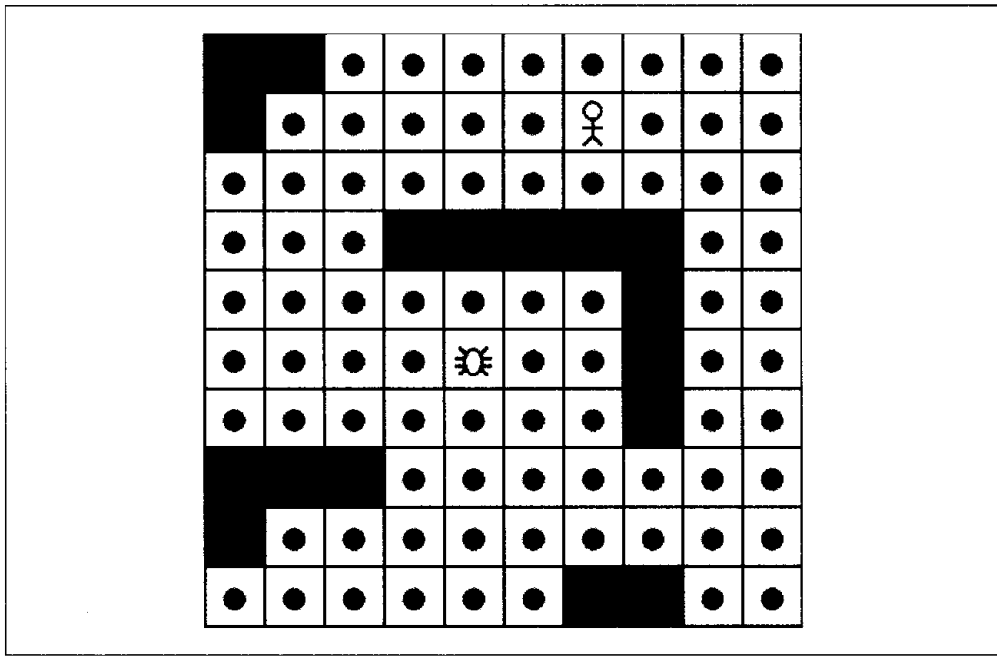


图 7-2：砖块搜寻区域

如图7-2所示的砖块环境最适合用A*算法。每个砖块都是搜寻区域里的节点。你不需要拥有节点间的连接数据清单，因为这些节点在游戏世界中都是彼此相邻的。如果有必要，你也可以简化砖块环境。你可以放一个节点以代表好几个砖块。在很大的砖块环境中，可以让路径寻找算法，只搜寻游戏世界里的某一部分。可以把它理解为大方块里的小方块。如果在小方块里找不到路径，就可以认定没有合理的路径存在。

开始搜寻

一旦我们简化了搜寻区域，使其由适当数量节点构成时，就能准备开始搜寻了。我们会以A* 算法找出任何两节点间的最短路径。就此例而言，我们要用一个小型的砖块环境。每个砖块都是搜寻区域里的一个节点，而某些节点含有障碍物。我们要运用A* 算法找出最短路径，同时避开障碍物。例 7-1 是我们要做的基本算法。

例7-1：A*伪代码

```
把起始节点加进open list
while open list不为空
  {
    当前节点= open list中成本最低的节点
    if当前节点 = 目标节点then
      路径完成
    else
      把当前节点移入closed list
      检视当前节点的每个相邻节点
      for每个相邻节点
       if该节点不在open list中
        and该节点不在closed list中
         and该节点不是障碍物then
           将该节点移进open list并计算其成本
  }
```

例7-1的伪代码有些地方让人没有看懂，但是当我们开始逐步谈此算法时，你就会一目了然了。

图7-3是我们要用的砖块搜寻区域。起点是靠近中心的蜘蛛，而想要到达的目的地则是人类角色。黑方块代表墙体障碍物，而白方块代表蜘蛛能走的地方。

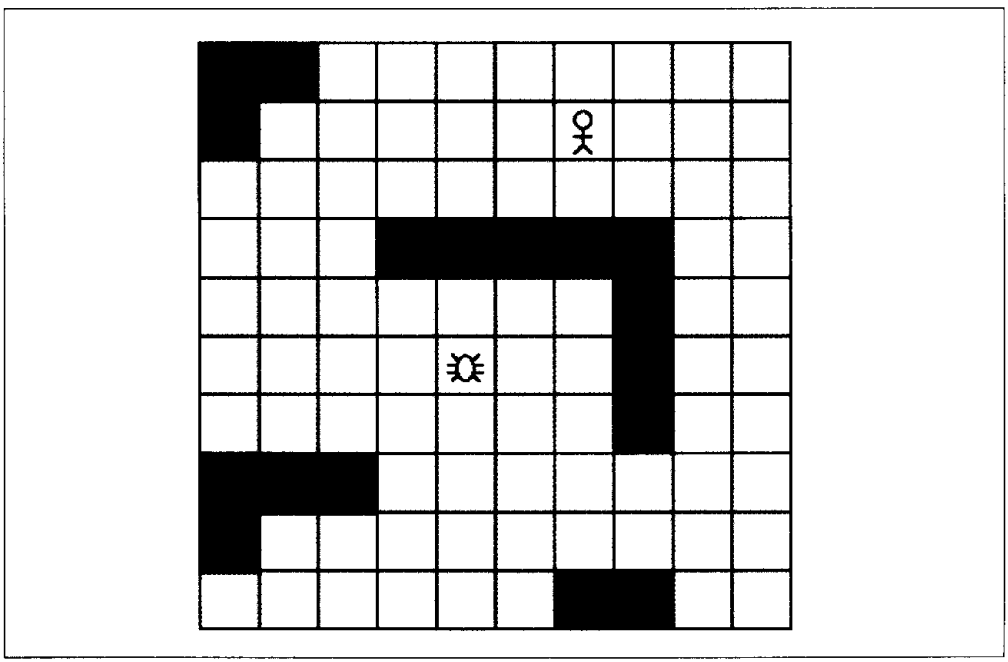


图7-3：建立砖块搜寻区域

如同任何路径寻找算法一样，A*会找出起始节点和终止节点间的一条路径，其做法是先从起始节点开始搜寻，然后再分别去搜寻周围节点。就此例而言，会先从起始砖块开始，再扩散到相邻砖块，直到抵达目的地节点。然而，开始做这种扩散搜寻技巧前，我们还需要某种方式，记录已经被搜寻过的砖块；使用A*算法时，通常把记录方式叫做*open list*。开始时，open list只有一个节点，也就是起始节点，接着会陆续把其他节点加进open list。（注意，节点和砖块这两个术语可以互换使用，同指砖块环境。）

建好open list后，接着予以检查，搜寻清单内每个砖块相邻的砖块。这背后的思想是检查每个相邻砖块，看它是否为路径上的有效砖块。基本上，我们是在检查相邻砖块能否让游戏角色行走。例如，道路砖块就是有效的，而墙体砖块就可能是无效的。我们会检查八个相邻砖块中的每一块，然后，把每个有效的砖块加进open list。如果某砖块含有障碍物，我们会直接予以忽略，不会加进open list。图7-4显示了最初位置的相邻砖块，它们都需要做检查。

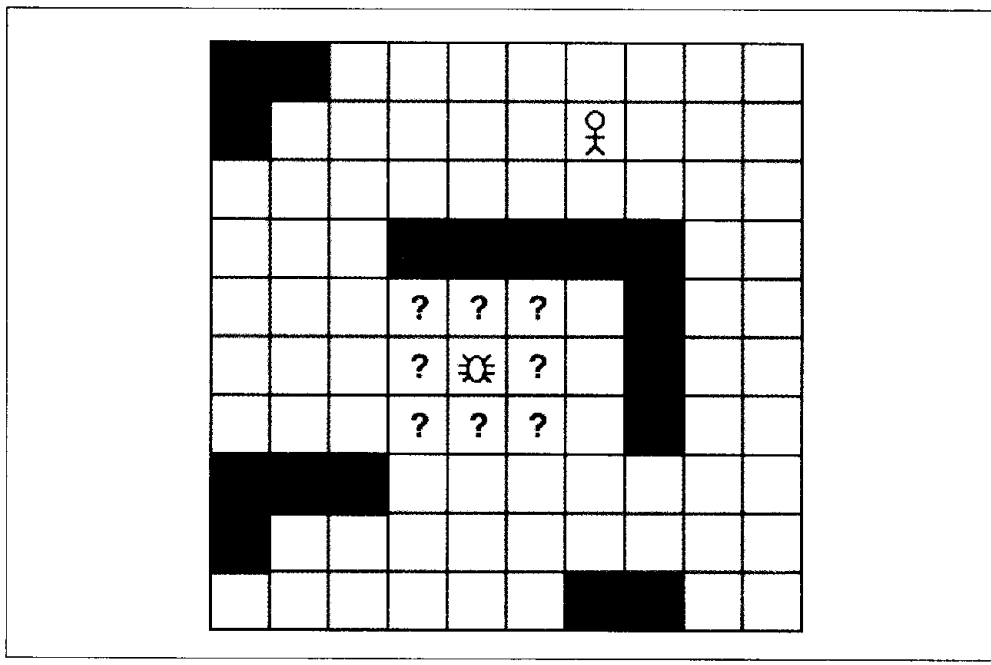


图7-4：要处理的相邻砖块

除了open list之外，A*算法也要拥有一份*closed list*。closed list里的砖块就是已经被检查过的砖块，不需要再做检视了。基本上，当某砖块的相邻砖块都已检查过后，就会将该砖块加入closed list。如图7-5所示，我们已检查过起始砖块的每个相邻砖块，所以，起始砖块会加入closed list。

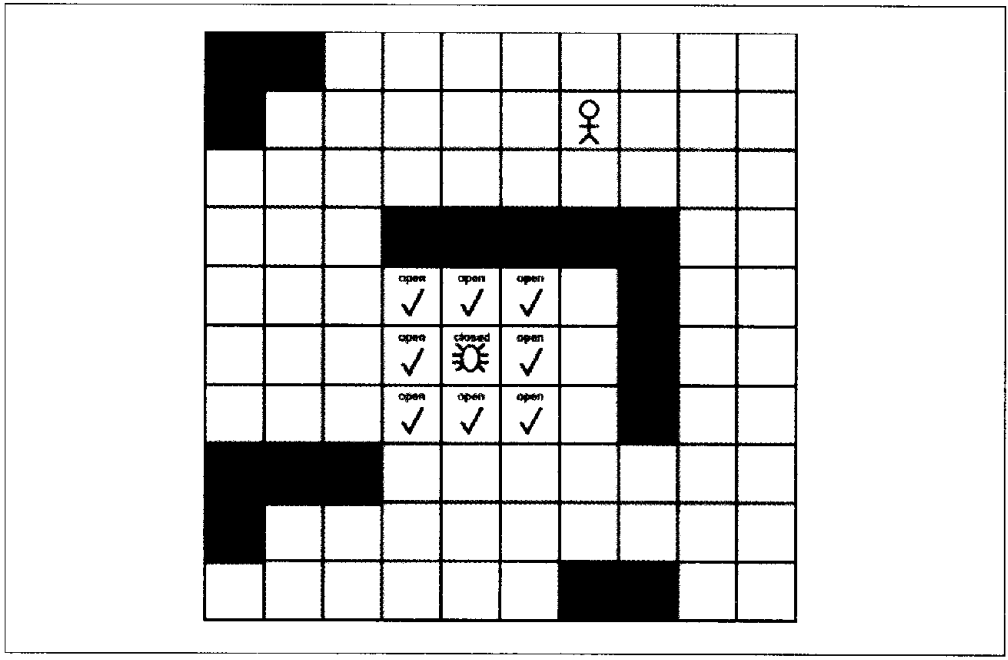


图 7-5：把起始砖块移入 closed list

所以，如图 7-5 所示，最后的结果是我们现在有八个新砖块加入 open list，而有一个砖块从 open list 中移除。话说到这里，其实就是 A* 主循环的基本检查运算，然而，我们必须再记录其他某些信息。我们必须运用某种方式把砖块连接起来。open list 中有相邻砖块清单，它能供给角色行走，但我们也必须知道这些相邻砖块是怎么连接的。我们的做法是记录open list中每个砖块的母砖块。所谓砖块的母砖块就是该角色走到当前位置前的那个砖块。如图 7-6 所示，运行第一次循环时，每个砖块都会以起始砖块作为其母砖块（以箭头指向表示）。

最后，当我们抵达最终目的地时，就会利用此母砖块的连结关系，沿路径往回倒退到起始砖块。然而，抵达目的地之前，我们仍然需要多次反复运行循环。

此时，我们要再次执行整个过程。现在，我们必须从open list中选出新的砖块进行检查。第一轮循环时，open list 上只有一个砖块，但现在清单上有八个砖块。重点就是找出要检查 open list 的某一个成员。我们确认的方式就是替每个砖块打分。

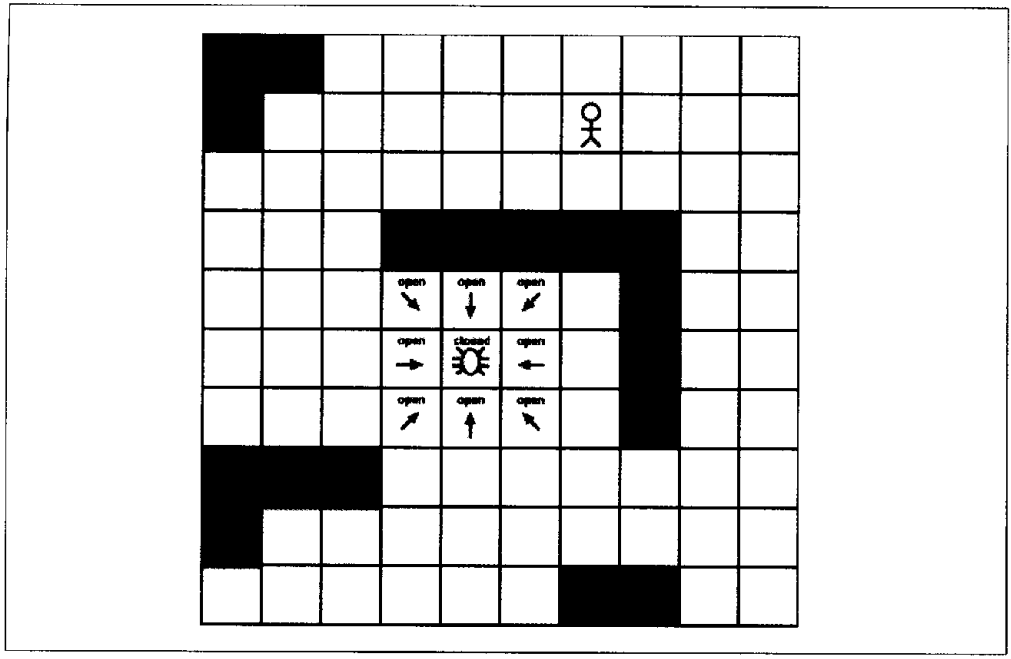


图 7-6：和母砖块连接

记分

最后，我们用路径得分（path score），找出起始砖块和目的地砖块间的最佳路径。实际给每个砖块记分时，基本上是把两个分数加在一起。首先，我们要计算从起始砖块移到任何指定砖块上的移动成本（cost）。其次，计算从指定砖块移到目的地砖块所需的移动成本。第一个得分可以说一目了然。我们从最初位置开始搜寻，再从该地扩散出去，如此一来，计算从最初位置移到扩散出去的每个砖块的移动成本时，就相当容易了。只要把每个砖块走回到最初位置所需的移动成本，各自累加起来就行了。记住，我们有储存每个砖块的母砖块连结关系。走回最初位置是轻而易举的。然而，如何求出从指定砖块移到目的地砖块的移动成本呢？目的地砖块是最终目标，但还没走到。所以，如何计算尚未求出的路径的移动成本呢？此时，我们使用启发法（heuristic）来猜测。基本上，我们根据所有信息，就能做出最佳猜测。图 7-7 是我们替任何指定砖块记分的公式。

所以，我们计算每个砖块的分数，是把从起点走到该处的移动成本，加上用启发法所得的移动成本（也就是估算的从指定砖块走向目的地砖块的移动成本）。

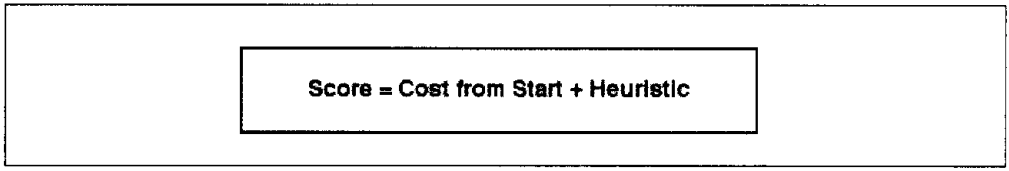


图 7-7：计算路径得分

我们用此得分决定 open list 中下一个要检查的砖块。先检查移动成本最低的那些砖块。就此而言，移动成本较低就相当于路径较短。图 7-8 是目前为止检查过的每个砖块的分数、移动成本以及启发移动成本的结果。

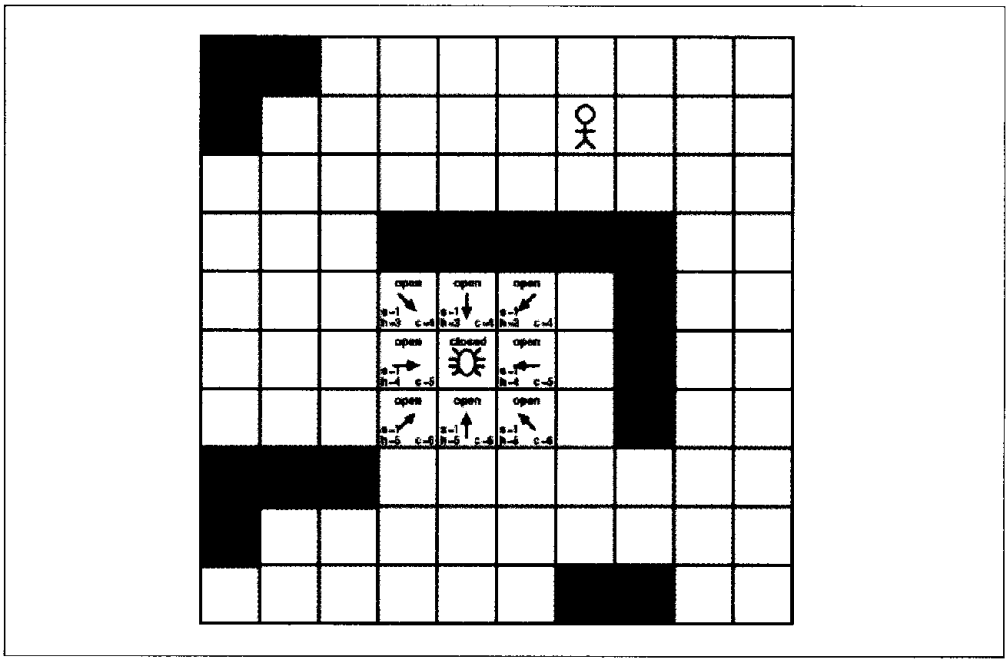


图 7-8：最初的砖块路径得分

每个被检查的砖块上的 s 值就是从起始砖块走到该砖块上所需的移动成本。就此而言，每个值都是 1，因为从每个砖块到起始砖块只是一步之遥。h 值就是启发移动成本。启发移动成本是估算的从指定砖块到目的地砖块的步数。例如，起始砖块右上角的砖块的 h 值为 3，这是因为这个砖块离目的地砖块有三步之遥。你会注意到计算启发移动成本时，我们没有把障碍物考虑进去。因为我们还没有检视当前砖块和目的地砖块之间的砖块，所以如果含有障碍物，我们也尚不知道。此时，只想算出成本，所以我们假设没有障碍物。

最终值是c，也就是s和h的总和。这就是砖块的总成本，代表的是从起点走到该处的已知成本，与从该处走向目的地的估算成本之和。

先前我们提出A*算法进入下一轮循环时，要从open list中先选出哪一个砖块进行检查的问题，而答案是有最低的c值砖块。如图7-9所示，最低的c值为4，但有此值的砖块实际上有三块。我们应该选哪一个呢？这并不重要。我们就先从位于起始砖块右上角的那个开始吧。假设我们用的是（行、列）坐标系统，搜寻区域的左上角位置的坐标是（1，1），而我们现在检查的砖块的坐标是（5，6）。

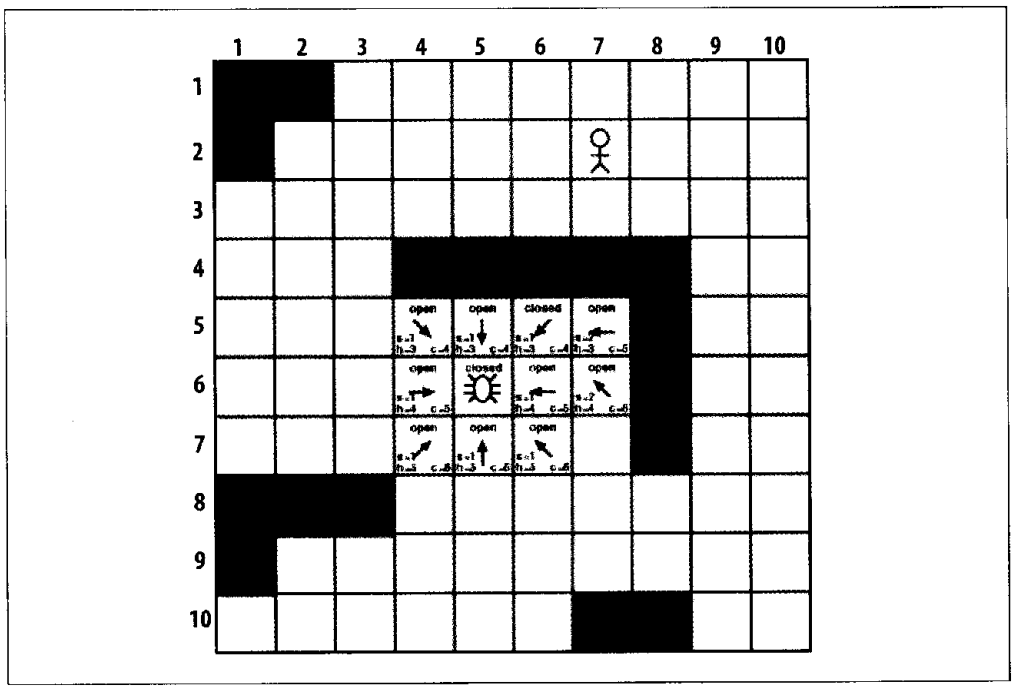


图7-9：检视位于（5，6）的砖块

图7-9中目前要检查的砖块的坐标是（5，6），位于起始位置的右上角。现在，我们要重复先前说过的算法程序，当时我们检视了当前砖块相邻的每个砖块。第一轮循环中，每个和起始砖块相邻的砖块都可用，也就是说还未被检视过，同时也不含任何障碍物。然而，到这一轮时，情况就不是这样了。寻找相邻砖块时，我们只要考虑还未检视过以及游戏角色可以行走的砖块。这也就是说要忽略open list和closed list中的所有砖块、或者含有障碍物的所有砖块。如此一来，只剩两个砖块，也就是当前砖块右侧的那个，以及当前砖块右下角的那个。这两个砖块都会加入open list。如图7-9所示，我们为每个加入open list的砖块加上了箭头符号，指向其母砖块。我们也计算了新砖块的s、h以

及 c 值。就此而言，我们计算 s 值的做法是利用母砖块连接关系往回走，这会告诉我们从起点到该处有多少步。同样的，h 值是启发移动成本，也就是估算从指定砖块到目的地的距离。另外，同样道理，c 值是 s 和 h 值的总和。最后一步是把位于（5, 6）的当前砖块放入 closed list。对我们而言，这个砖块不再有用处了。我们已经检视过其每个相邻砖块，所以，不需要再予以检视了。

现在重复此过程。我们已把两个新的砖块加入 open list，而且把一个砖块移进了 closed list。我们要再次搜寻此 open list，找出最低成本的砖块。如同第一轮循环那样，open list 中砖块的最低成本值依然为 4。然而，这一次 open list 中只剩两个成本为 4 的砖块。同样，先检视哪一个并不重要。就此例而言，我们先检视（5, 5），如图 7-10 所示。

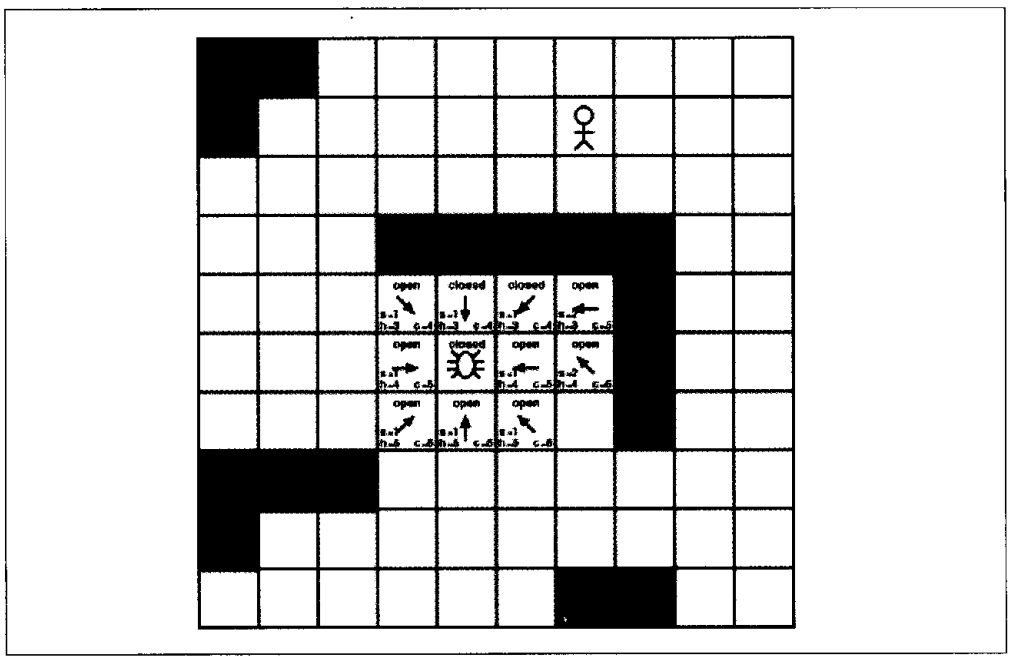


图 7-10：检视位于（5，5）的砖块

如前所述，我们要检视当前砖块的每个相邻砖块。然而，当前砖块的相邻砖块中没有可用的，不是在 open list，就是在 closed list 中，不然就含有障碍物。所以，如图 7-10 所示那样，我们只要把当前砖块移进 closed list 中，继续走下去就行了。

现在，我们要看那个在 open list 中优于其他砖块的砖块，其成本为 4，在 open list 的所有砖块中是最低值。此砖块位于（5, 4），就在起始位置的左上角。如图 7-11 所示，这就是我们接下来要检视的砖块。

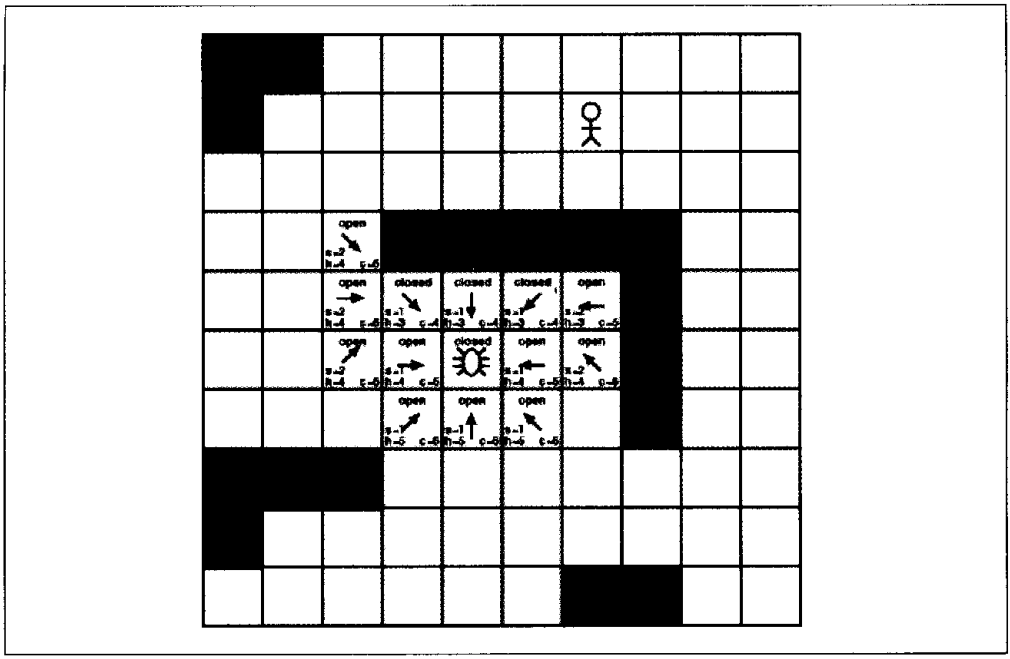


图7-11：检视位于（5，4）的砖块

如图7-11所示，我们同样要检视当前砖块的所有相邻砖块。就此而言，只有三个砖块可用：左上角、左侧以及左下角。其余的砖块不是在open list，就是在closed list中，不然就含有障碍物。这三个新砖块会加进open list，而当前砖块会移入closed list。然后，我们替此三个新砖块算出其得分，再继续执行整个程序。

我们加了三个新砖块到open list，而且把一个砖块移入closed list。上次我们检视open list时，我们发现砖块中的最低成本为4。这一次open list中砖块的最低成本为5。事实上，open list中有三个砖块之值为5，它们分别是（5, 7）（6, 6）以及（6, 4）。图7-12是检视这三个砖块后的结果。A*算法实际上每次只能检视一个砖块，如果没有更低值的新砖块可用，就会检查其他拥有相同值的砖块。然而，就此例的目的而言，只显示了检视完这三个砖块后的结果。

如同前几轮的循环那样，每个新的可用砖块都会加入open list，而每个检视过的都会移入closed list。我们也会替新砖块算出其得分。再度搜寻open list时，我们得知最低成本值变为6，所以继续检视有该值的砖块。同样的，先检查哪一个不重要。如图7-12那样，我们假设最坏的情况，也就是最佳选项最后才被选到，所以展示出检视所有分数为6的砖块的结果，如图7-13所示。

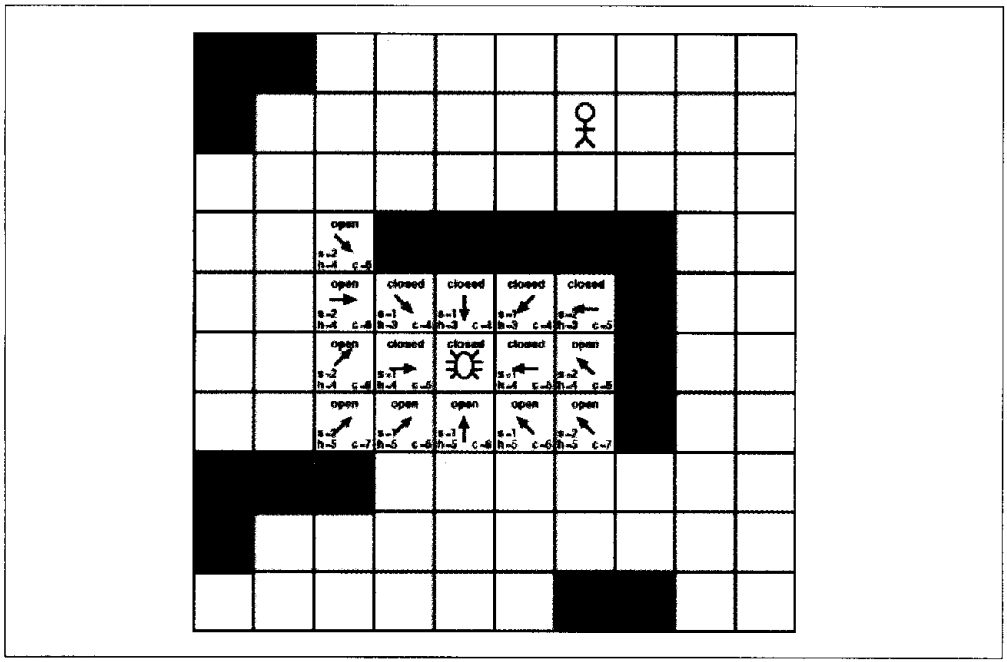


图 7-12：检视所有成本为 5 的砖块

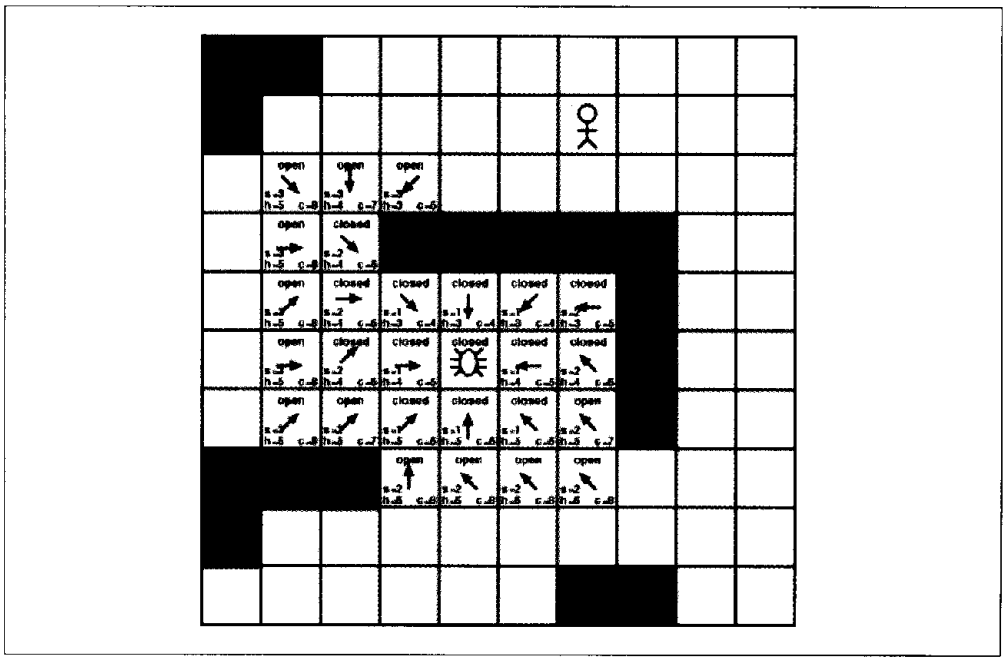


图 7-13：检视所有成本为 6 的砖块

如图7-13所示，我们检视了每个成本为6的砖块。如同前几轮循环那样，新的砖块会加入open list中，而检视过的砖块会移入closed list。同样的，新砖块的成本也会算出来。如图7-13所示，启发移动成本越来越显著。搜寻区域较低部分的砖块的启发移动成本会增加，对成本的总和具有很明显的增量。搜寻区域中较低部分的砖块仍然开放着，所以，仍有可能提供到目的地的最佳路径，然而，就目前而言，有更好的选项可以追寻。搜寻区域顶端的开放砖块有较低的启发移动成本，这才是我们更想要的。事实上，搜寻open list后，显示出当前砖块成本最低的是6。就此而言，只有一个砖块的成本值为6，也就是位于（3, 4）的砖块。图7-14是检视位于（3, 4）的砖块结果。

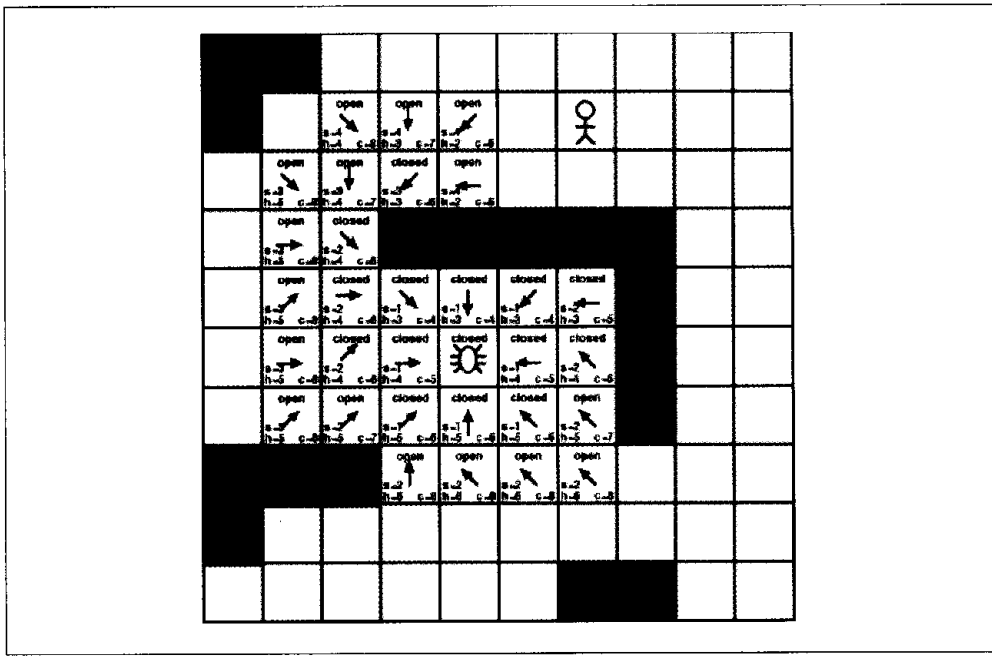


图7-14：检视位于（3，4）的砖块

检视位于（3, 4）的砖块时，会让三个新的砖块加进open list中。当然，位于（3, 4）的当前砖块会接着移进closed list。如图7-14所示，open list中多数砖块的成本值都是8。幸运的是，上一轮循环中加进来的其中两个新砖块的成本值为6。这两个砖块是我们接着要研究的。同样的，就此例的目的而言，我们要假设最坏的情况，所以有必要检视两个。图7-15是检视这两个砖块的结果。

如图7-15所示，我们终于靠近目的地了。上一轮循环在open list中又加入了五个新砖块，其中有三个砖块的成本值为6，也是当前的最低成本。如同前几轮循环那样，我们现在要检视成本值为6的这三个砖块，如图7-16所示。

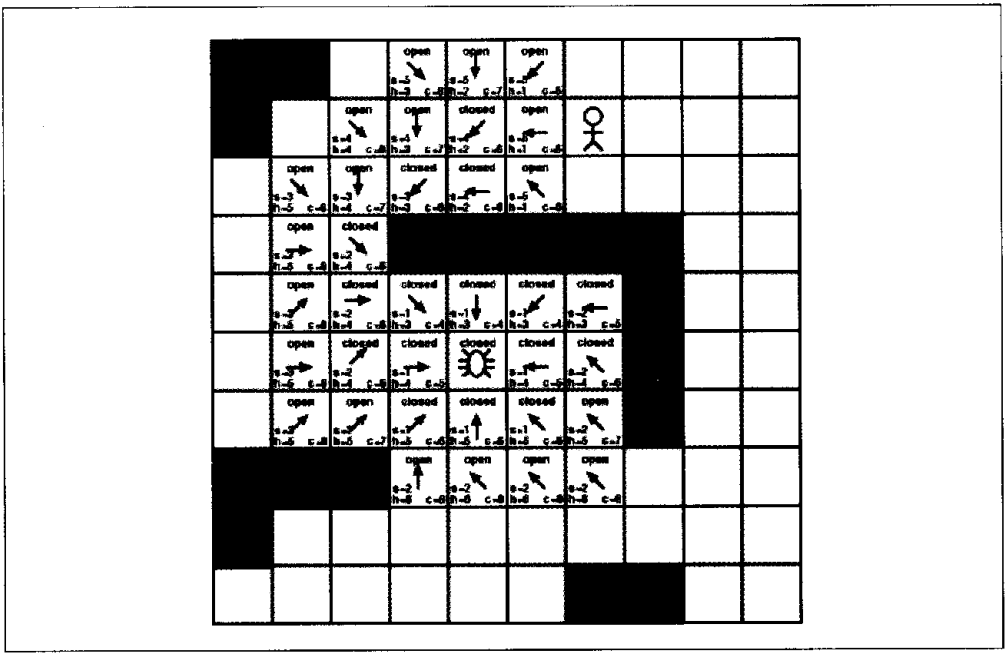


图7-15：检视位于（2，5）和（3，5）的砖块

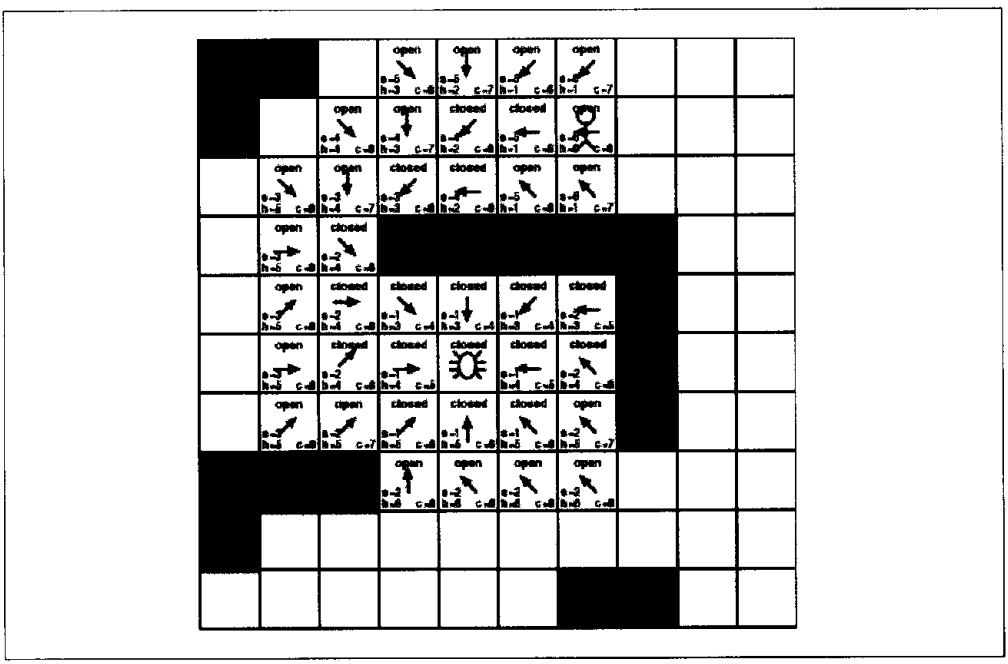


图7-16：检视位于（1，6）、（2，6）和（3，6）的砖块

如图7-16所示，我们终于抵达目的地了。然而，这个算法如何确定我们何时抵达目的地？答案很简单。当目的地砖块加进open list时，就表示路径已找到了。届时，按照母砖块连接关系走回起始点，那是轻而易举的事。现在我们关心的节点，就是带着我们走回起始节点的那些节点。图7-17表示了建立实际路径的节点。

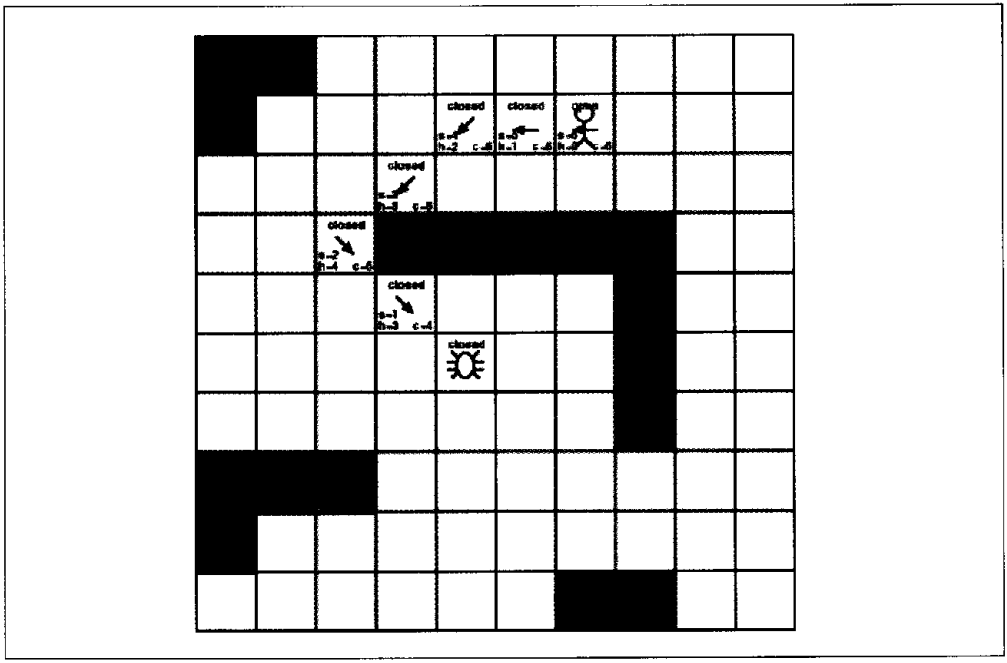


图7-17：完成的路径

一旦目的地节点也放入open list中，我们就知道路径寻找已完成。然后，按照母砖块连接关系走回起始砖块。就此而言，将产生一条路径，由（2, 7）、（2, 6）、（2, 5）、（3, 4）、（4, 3）、（5, 4）以及（6, 5）这几个点所构成。如果你按照我们这里教授的算法去做，你就会找到最短的可能路径。其他等长的路径也许会存在，但不会有更短的了。

搜寻死路

任何两个指定点之间的有效路径，也可能不存在，所以，遇上死路时，我们如何得知？最简单的方法就是监控open list。如果我们检查到最后的节点，open list中再也没有任何成员，就是遇上死路了。图7-18属于这样的场景。

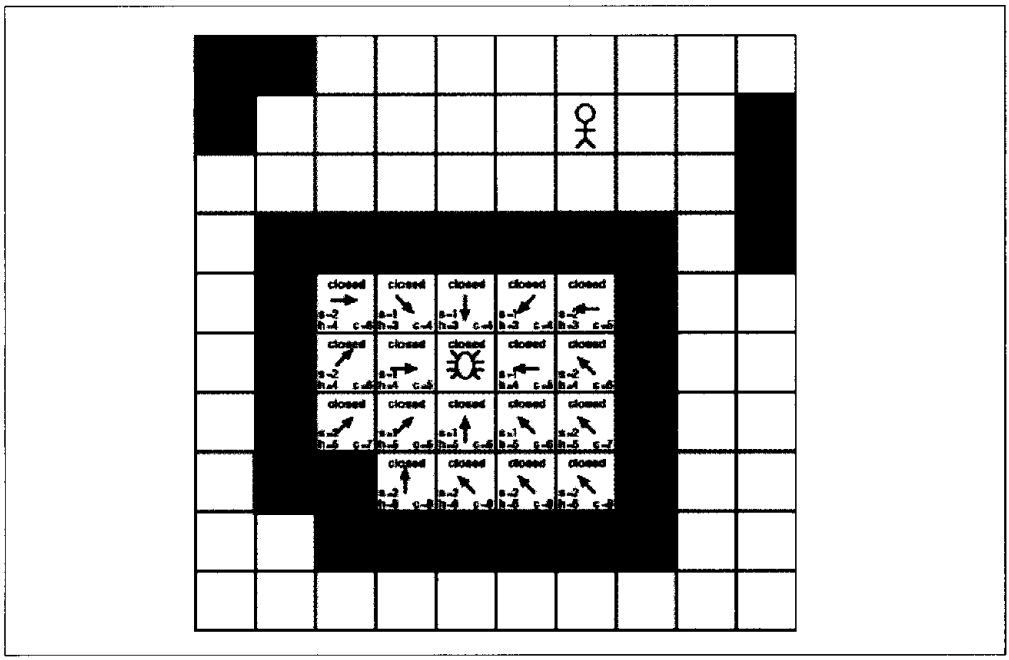


图7-18：死路

如图7-18所示，A*算法已扩散到每个可能的相邻砖块。每个砖块均被检视过，都移到closed list中了。到最后，open list中的每个砖块都被检视过，没有新的砖块可以加进来。我们就可以断定遇到找不到路的情况了，不可能从起点建立路径走向想要到达的目的地。

地形成本

如前面的例子所示，路径记分在A*算法中已是主流。标准A*算法的最基本形式，就是利用所经过的距离计算路径成本。比较长的路径就视为成本比较高，因此，也不怎么受青睐。我们常常认为，好的路径算法就是要能找出最短的可能路径。然而，有时候还需要考虑其他因素。例如，最短路径不见得就是最快的。游戏环境中可能还包含很多种类型的地形，这些地形对游戏角色的影响也不同。沿着道路长途跋涉，也许比通过路程较短的沼泽地要快一点。这就是地形成本的作用所在。前面的例子指出，我们可以计算每个节点从初始位置到其所在点之间的距离，再加上估算该点到目的地的启发距离，而得到节点的移动成本。也许并不明显，但前例基本上已经把地形成本算进去了，因为所有地形都一样，所以没有那么显著；游戏角色每走一步都会在路径成本里加1。基本上，每

个节点都有相同的成本。然而，我们可以指派不同的成本值给不同的节点。这需要对成本公式稍微做些修改。可以修改成本公式，把地形成本也算进来，如图 7-19 所示。

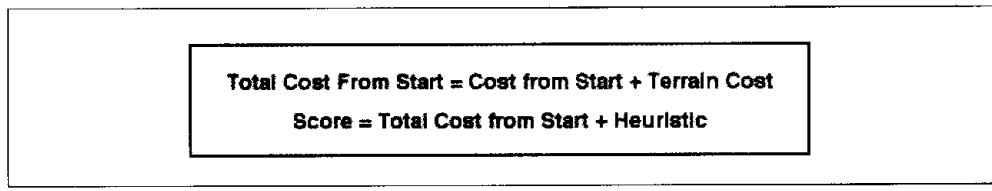


图 7-19：考虑地形成本而记分

这样会把较长但含有较简单地形的路径也考虑进来。在实际游戏中，将让游戏角色在比较短的时间内从 A 点走向 B 点，即使实际走的路径比较长。例如，图 7-20 显示了几种假设的地形类型。

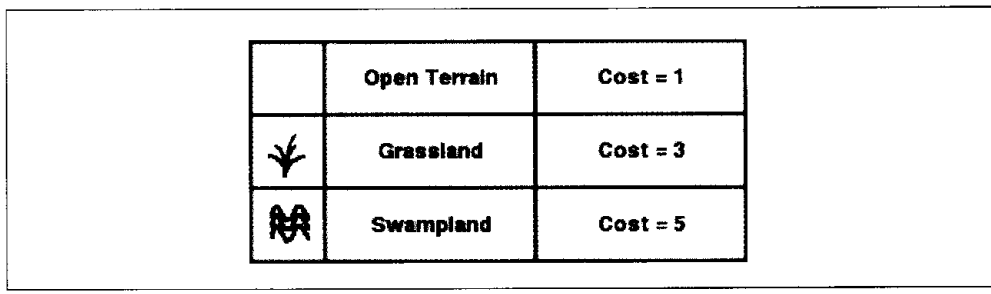


图 7-20：地形类型

前面的例子中本质上只有宽广地形而已。从一个节点移到另一个节点的成本都是 1。如图 7-20 所示，我们将引入两种新类型的地形。第一种新地形是草地，成本为 3。第二种新地形是沼泽地，成本为 5。就此而言，成本到了最后指的就是通过节点所需花费的时间。例如，如果游戏角色要花 1 秒才能通过宽广地形的节点，通过草地地形的节点则需花 3 秒，而且需花 5 秒才能通过沼泽地形的节点。实际的距离也许相等，但是通过时所花的时间却不同。A*算法总是搜寻最低成本的路径。如果每个节点的成本都相同，其结果就是最短路径。然而，如果我们令节点的成本不同，则最低成本路径也许就不再是最短路径了。如果我们把成本视同时间，则A*将找出最快的路径，而不是最短的路径。图 7-21 是和前面的例子相同的砖块配置，但引进了如图 7-20 所示的地形元素。

如图7-21所示，障碍物和游戏角色都和前面的例子一样位于相同的位置。现在唯一的差别是新增了地形成本。我们不再寻找最短实际路径，现在要找最快路径。假设通过草地所花的时间是通过宽广地形的三倍，而通过沼泽地所花时间是宽广地形的五倍。那么多

加进来的地形成本，对路径产生什么样的影响？图 7-22 表示了前面的例子所产生的路径。

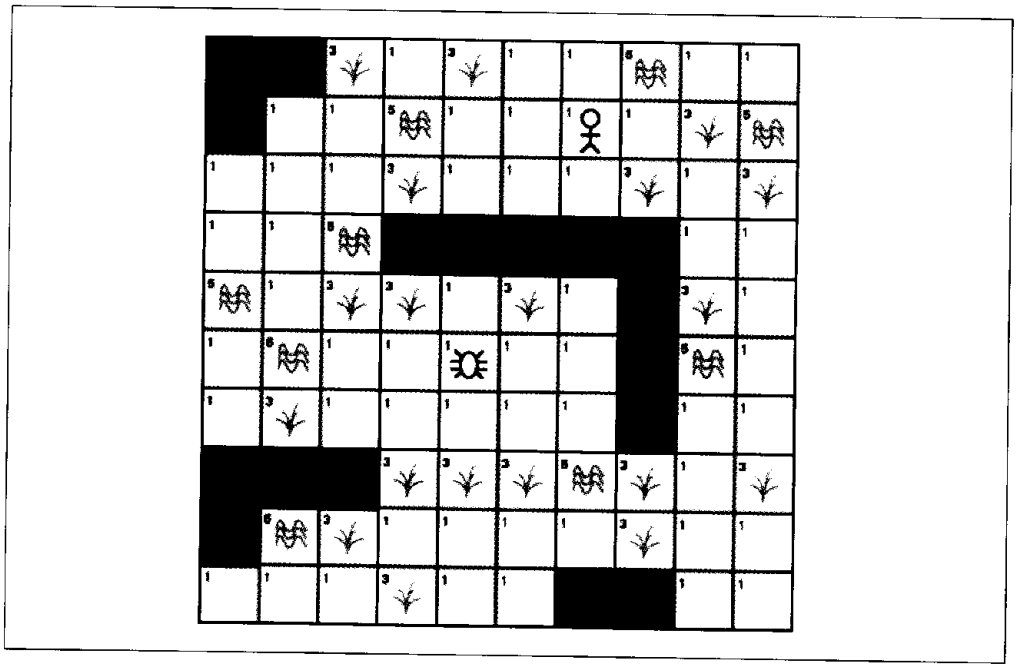


图 7-21：新增不同地形元素

如图 7-22 所示，找到的是最短路径。然而，我们注意到，该路径现在通过了好几个高成本的地形元素。这是最短路径，毋庸置疑，但它是最快的路径吗？我们与前例使用同一个 A* 算法来逐步确认，但这次要把地形成本加进每个节点的总成本。图 7-23 是按照整个算法求出的结果。

如图 7-23 所示，这条路径和前例算出的路径很像。我们以相同的扩散技巧，检视当前砖块的相邻砖块。然后利用相同的 open list 和 closed list，记录需要检视以及不再有用的砖块。主要的差别在于 s 值，也就是从起始节点移到任何指定节点的成本。前例中，每个节点都是 1。现在，要把地形成本加进 s 值。最后所得的最低成本路径如图 7-24 所示。

如图 7-24 所示，A* 算法绕过了成本较高的地形元素。我们不再有最短的实际路径，但可以确定没有比这条路径更快的路径了。或许还有其他路径，实际上短一些或长一些，花费的时间也一样长，但没有一条会比找到的这条路径更快了。

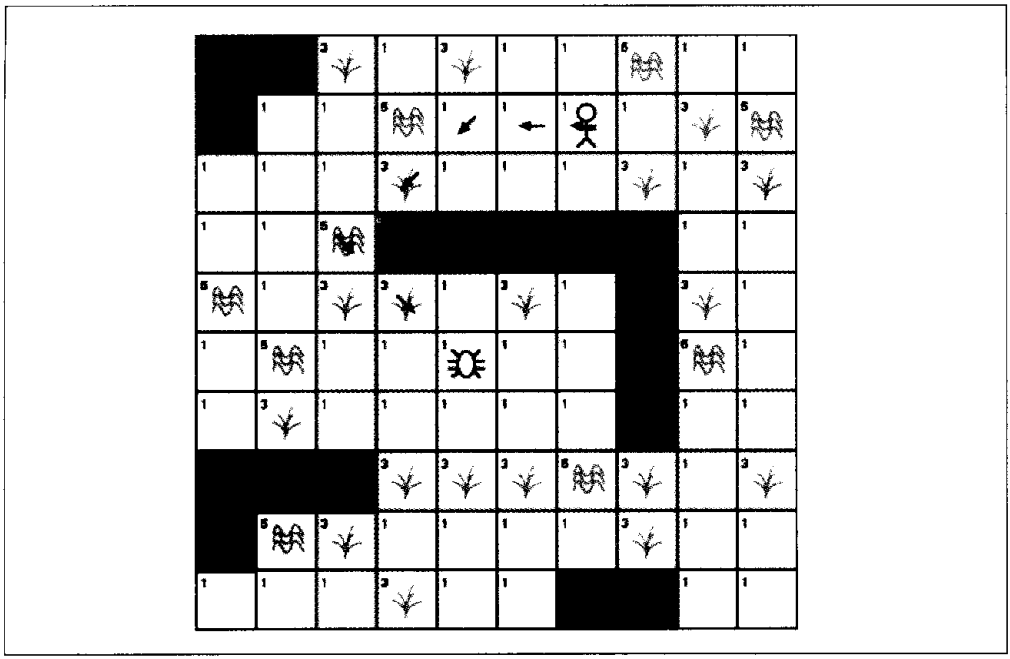


图 7-22：以原来的路径通过地形元素

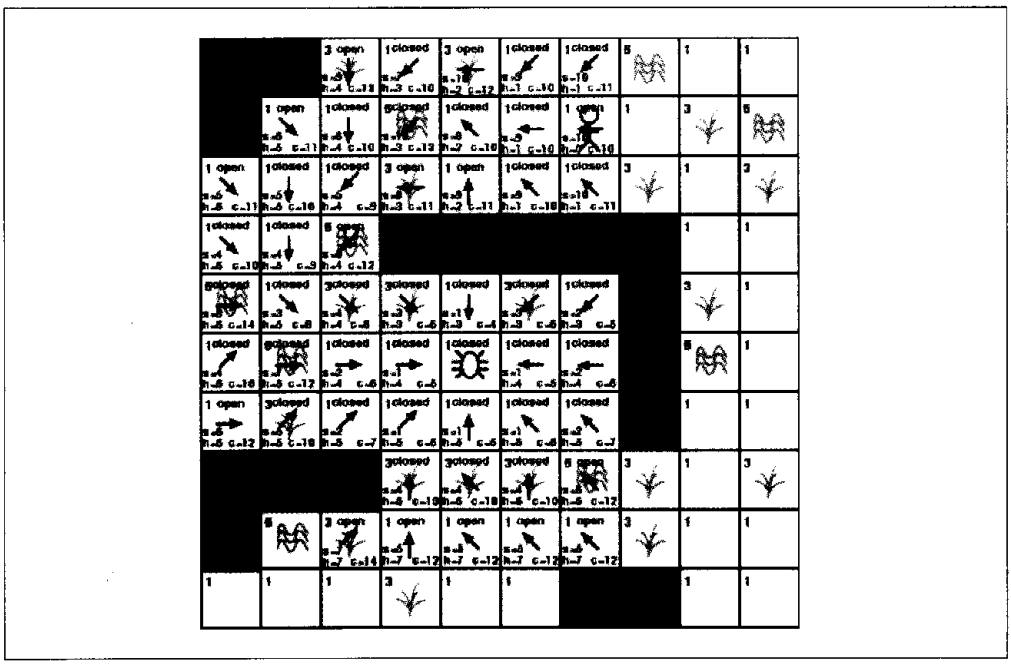


图 7-23：算出最低成本路径

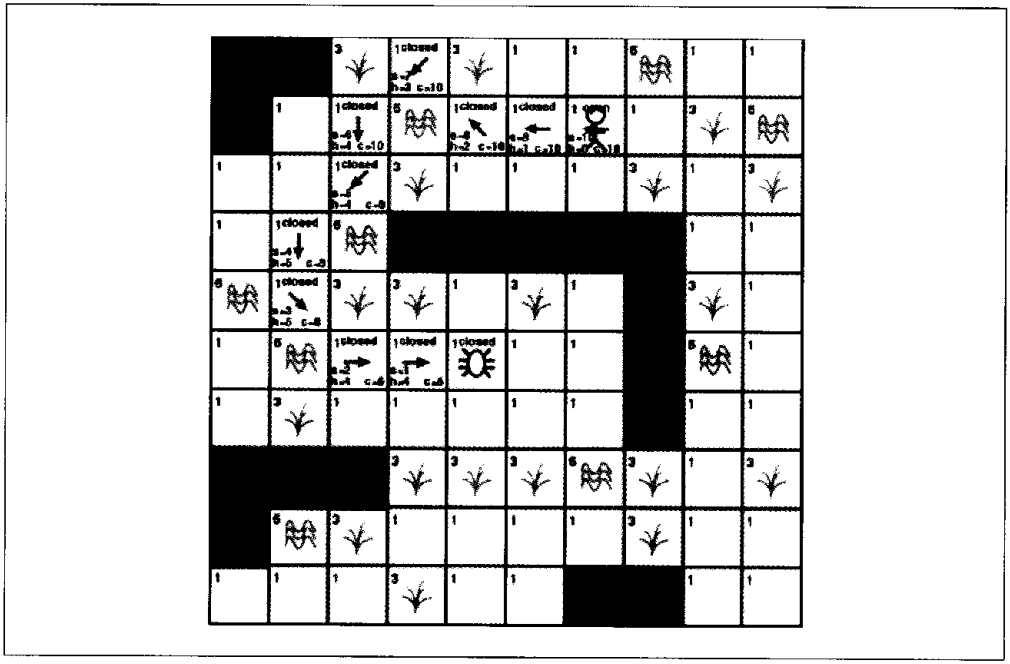


图 7-24：最低成本路径

当你把A*算法用到连续环境时，地形成本也有用处。前例是教你如何把A*算法用到砖块环境中，所有节点都是等距离的。然而，等距离的节点并非A*算法的必要条件。图7-25显示了如何把节点放在连续环境中。

在图7-25中，你会注意到，节点间的距离是不相等的。这意味着在连续环境中，当节点间的距离很遥远时，会花较长的时间才能通过。当然，这是假定节点间有等效的地形。然而，就此而言，即使地形是等效的，节点间移动的成本还是有可能不一样，因为成本就相当于节点间的距离。

我们讨论了好几种和节点间移动有关的成本类型。虽然我们倾向于把成本想象成时间或距离，但还有其他可能性存在，诸如金钱、燃料、或其他类型的资源。

影响力对应

上一节谈的是不同的地形元素，对A*算法计算路径时产生的影响。地形成本通常是游戏设计者直接编进游戏世界里。基本上，我们事先就知道草地、沼泽地、丘陵以及河流会在何处。然而，利用A*计算路径时，其他元素也会影响路径成本。例如，通过任何

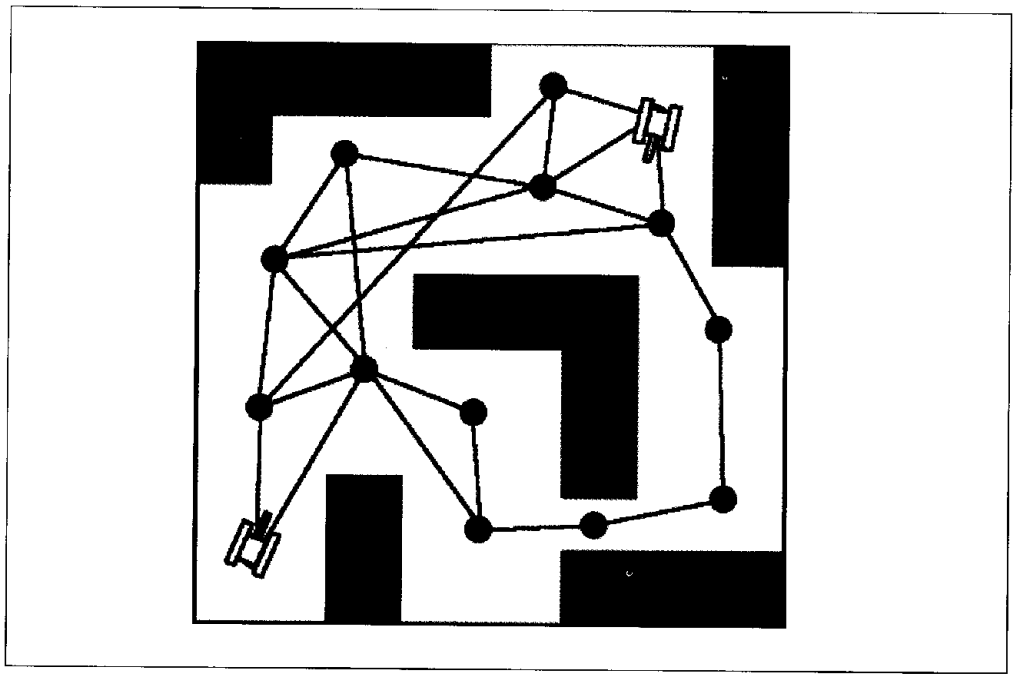


图 7-25：连续环境中置放节点

敌人的视线的节点，有较高的成本。这种成本你无法在设计游戏软件阶段时建立，因为游戏角色的位置是会改变的。影响力对应（influence mapping）是一种改变 A* 节点成本的方法，依据游戏里发生的情节而定，如图 7-26 所示。

如图 7-26 所示，我们替每个节点都指派了成本。然而，和上一节谈的地形成本不同，这种成本会受到位于（8, 4）的坦克的位置和方向的影响。这种影响力对应将随着坦克的位置和方向的改变而改变。就像上一节所讲的地形成本那样，在计算可能路径时，影响力对应成本也会加到每个节点的 s 值之中。这将造成坦克的目的地在建立路径时，可能会走一条比较长而慢的路径。然而，火力网下的砖块还是可以通过的，只是成本高一点而已。如果没有其他路可走，或者其他路的成本更高，则游戏角色就会通过火力网。

你可以在其他方面使用影响力对应，让游戏角色看起来聪明一点。你可以在影响力对应中记录个别游戏事件。就此而言，我们不再以游戏角色的位置和方向建立影响力对应。我们要改用角色做一些事。例如，如果玩家在特定路口处，不断狙击和杀害计算机控制的角色，则该路口也许就可以在成本上增加其值。然后，计算机就可以尽可能改用其他路径。对玩家而言，这可以让计算机控制的角色看起来很聪明，好像他们能从错误中汲取教训。这种技巧如图 7-27 所示。

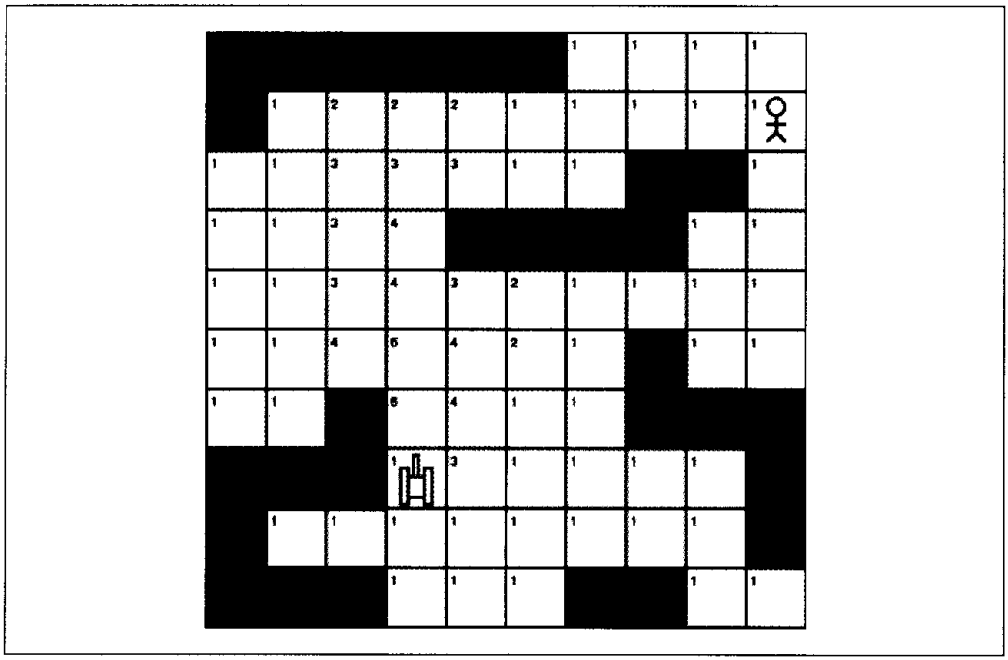


图 7-26：受到敌人火力网的影响

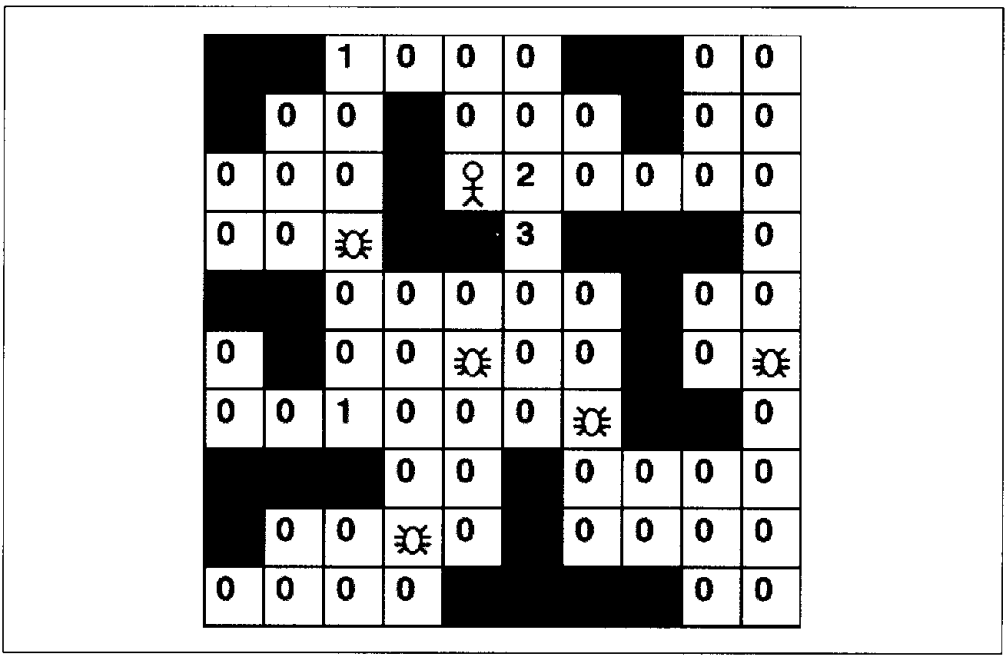


图 7-27：受被杀害数量的影响

图7-27所示的影响力对应，记录了玩家在每个节点杀害蜘蛛的数量。每次玩家杀一只，该节点就会在成本上增值。例如，也许在某个路口上，玩家发现有种狙击技巧，可以连续杀死蜘蛛。此时，就不要再让计算机控制的对手不断地通过该路口，而是可以建立也许比较长但成本较低的路径，以削弱玩家的战术优势。

其他信息

Steven Woodcock在其《2003 Game Developer's Conference AI Roundtable Moderator's Report》报告书中指出，AI开发人员本质上就是在解决路径寻找问题。各地网站上其他游戏软件AI资源，都在响应这种说法。开发人员的思想是经过验证的算法，可以在各式各样游戏场景中解决路径寻找问题，而最近多数的努力都花在如何让这些方法最佳化上。目前为止，最受欢迎的路径寻找方法是A*算法。当前发展的重点，集中在发展更快且更有效的A*算法。《Game Programming Gems》(Charles River Media)以及《AI Game Programming Wisdom》(Charles River Media)有许多有趣的文章专门探索A*的最佳化议题。

第八章

描述式AI及描述引擎

本章讨论某些技巧，让你把描述系统（scripting system）应用到游戏软件AI的问题上，以及这样做以后所能获得的好处。从最基本的层次来看，你可以把描述机制想象成非常简单的程序语言，专门为与游戏问题相关的特定工作而量身打造。描述机制可以说是游戏开发过程中，不可或缺的一部分，因为这可以让游戏设计师，而不是游戏程序员，撰写出游戏的巧妙之处，并予以精细化。玩家也可以利用描述语言，建立或修改其所处的游戏世界或等级。再进一步的话，你可以在超大型多人在线角色扮演游戏（MMORG，massively multiplayer online role-playing game）中使用描述系统，当人们实际在玩游戏时，就能改变游戏的行为。

实现描述系统时，可以采用好几种手段。例如，精致的描述系统，可以让实际所用的游戏引擎和现有的描述语言（诸如Lua或Python）衔接起来。有些游戏会建立专用的描述语言，专门设计处理个别游戏的需求。虽然有时候利用这些方法能获益匪浅，但是，让游戏分析包含描述命令的标准文字文件，会比较简单。如果采用这种手段，你就可以用任何标准文字编辑器，建立脚本（script）。在实际游戏中，当游戏开始时或者在某些特定时刻，可以读取脚本，并予以分析。例如，当玩家实际进入城堡时，控制城堡内的生物或事件的脚本，就能被读进来并进行分析。

在游戏软件AI的范围里，你可以用描述机制改变对手的属性、行为、响应方式以及游戏事件。本章要解释的就是这些用法。

描述机制技巧

游戏中实际采用的描述语言，最终还是由游戏设计师和程序员共同决定的。描述语言可以模仿过去已有的语言，诸如C或C++，也可以采用完全不同的手段；或许采用图形而

不是文字。描述系统怎么样以及如何操作，主要取决于谁会用这套描述系统。如果你的对象是终端玩家，那么比较自然的语言或者图形也许才更有优势。如果系统主要是供给设计师和程序员使用，把开发时间花在制作复杂而耗时的自然语言分析系统上，恐怕就没有好处了，快刀斩乱麻的手段也许比较好。

开发描述系统时，还应该考虑其他因素。也许你想让游戏设计师读写脚本能轻松一点，但对游戏玩家而言，就不必考虑这一点。就此而言，你可能会用加密的形式。你还可以开发描述编译器，使得最后的结果难以让人阅读。

本章我们要建立简单的描述命令，并将之储存在标准文字文件内。我们想避开复杂语言分析器，就要慎选词汇，令人相当轻松地读写脚本。换言之，我们所用的字词，要正确地反映脚本要修改的游戏的那个方面。

描述对手属性

利用某种描述机制，指定每个AI对手的所有基本属性，这是很常见也很有益处的做法。这会让我们在开发和测试过程中，能轻易调整AI对手。如果把所有重要数据都直接写进程序里，即使是最基本的修改，你也不得不重新编译。

一般而言，你可以描述对手的属性，比如智能、速率、强度、胆量以及魔法能力。实际上，可以描述的属性类型或数量是没有限制的，真正的决定因素是你正在开发的游戏类型。当然，计算机控制的友军或敌军无论何时和玩家互动时，游戏引擎最终都会用到这些属性。例如，有较高智能属性的对手，和较低智能的对手相比，行为就应该不同。也许较高智能的对手，会采用比较高级的路径寻找算法追踪玩家；而较低智能的对手，在试着走向玩家时就容易被困住。

例8-1是你可以用来设定游戏属性的基本脚本。

例8-1：设定属性的基本脚本

```
CREATURE=1;
INTELLIGENCE=20;
STRENGTH=75;
SPEED=50;
END
```

此例中，我们的脚本分析器（script parser）必须编译五个命令。第一个是CREATURE，它指的是要设定哪一个AI对手。下面三个分别是INTELLIGENCE、STRENGTH以及SPEED，就是实际设定的属性。最后的命令END，是通知脚本分析器，那个生物已设定完成，而接在END之后的任何东西，都属于另一个新区块的命令，与此无关。

直接在文件中引入数字1、20、75和50，可以轻易避开分析脚本文字的需要。这样做很有效，且开发人员经常使用，但这种做法有些缺点。首先，可读性不怎么强。其次，最重要的是，只用数字在文件中指定属性，有时候反而会让描述系统增加复杂程度，不切实际。例8-2说明了加上条件语句后，脚本就变得更加复杂了。

例8-2：以条件式脚本设定属性

```
CREATURE=1;
If (LEVEL<5)
   BEGIN
      INTELLIGENCE=20;
      STRENGTH=75;
      SPEED=50;
   END
ELSE
   BEGIN
      INTELLIGENCE=40;
      STRENGTH=150;
      SPEED=100;
   END
END
```

如例8-2所示，我们现在用条件式语句，根据当前游戏的等级，把生物属性设成不同的值。

脚本的基本分析

现在，我们已经介绍了基本属性脚本是怎么样的，我们打算进一步探索游戏如何读取脚本并予以分析。举例说明，我们要以基本脚本设定巨人的某些属性。建立一个名叫*Troll Settings.txt*的文字文件。例8-3是巨人设定文件的内容。

例8-3：设定属性的基本脚本

```
INTELLIGENCE=20;
STRENGTH=75;
SPEED=50;
```

例8-3是一个简单的范例，只设定了三个生物属性。然而，我们要编写一个程序，以便能轻易地增加其他属性，这只需对脚本分析器做稍许修改。基本上，我们打算编写自己的分析器，使其搜寻指定文件，找出特定关键字，并返回与该关键字相关的值。例8-4显示了在实际游戏程序中的内容。

例8-4：设定属性的基本脚本

```
intelligence[kTroll]=fi_GetData(Troll Settings.txt,
                                "INTELLIGENCE");
```

```
strength[kTroll]= fi_GetData(Troll Settings.txt,
                             "STRENGTH");
speed[kTroll]= fi_GetData(Troll Settings.txt,
                          "SPEED");
```

例8-4中，这三个假想的数组，可以储存生物属性。此时不要直接把这些值写进游戏程序中，而是从名为*Troll Settings.txt*的外部脚本文件中载入。fi_GetData()函数会检查这个外部文件，直到找到指定的关键词，然后，返回与该关键字相关的值。游戏设计师可以调整生物设定值，而无需在每次修改之后，都得重新编译程序代码。

现在，你已经知道可以用fi_GetData()函数设定巨人属性了，让我们再进一步来看。例8-5是这个函数完成其任务的程序。

例8-5：从脚本读取数据

```
int fi_GetData(char filename[kStringLength], char searchFor[kStringLength])

{

FILE   *dataStream;
char   inStr[kStringLength];
char   rinStr[kStringLength];
char   value[kStringLength];
long   ivalue;
int    i;
int    j;

dataStream = fopen(filename, "r" );
if (dataStream != NULL)
   {
       while (!feof(dataStream))
            {
               if (!fgets(rinStr,kStringLength,dataStream))
                   {
                      fclose( dataStream );
                      return (0);
                   }
             j=0;
             strcpy(inStr,"");
             for (i=0;i<strlen(rinStr);i++)
                if (rinStr[i]!= ' ')
                    {
                       inStr[j]=rinStr[i];
                       inStr[j+1]= '\0';
                       j++;
                    }

             if (strncmp(searchFor, inStr,
                         strlen(searchFor)) == 0)
                {
                   j=0;
```

```
                for(i=strlen(searchFor);
                    i<kStringLength;
                    i++)
                    {
                      if (inStr[i]== ';')
                        break;
                      value[j]=inStr[i];
                      value[j+1]= '\0';
                      j++;
                    }
                StringToNumber(value, &ivalue);
                fclose( dataStream );

                return ((int)ivalue);
              }
            }
          fclose( dataStream );
          return (0);
        }
    return (0);
}
```

例8-5中的函数一开始是接收两个字符串参数。第一个参数是指定要搜寻的脚本名称，而第二个是要搜寻的词汇。然后，这个函数会以指定的文件名打开文字文件。一旦文件打开了，这个函数就开始检查脚本文件，一次查一行文字。每一行都会被当作字符串而读进来。

注意，每一行都会读进变量rinStr中，再立即复制到inStr，但去掉了空白；去掉空白是让分析更加安全。如果脚本撰写者在所要搜寻的词或属性之前或之后加上了空白，这样做就可以避免脚本分析器出错。一旦把脚本的一行文字储存在字符串中，并且除去了空白，我们就能搜寻该词汇了。

回想一下，我们利用字符串变量searchFor把要搜寻的词传给fi_GetData()函数。此时，在这个函数里，我们用C函数strncmp()搜寻inStr，以期找到所要搜寻的词。

如果要搜寻的词没有找到，这个函数就会继续读取脚本文件里的下一行文字。然而，如果找到了，就会进入新的循环，把inStr变量中含有该属性值的部分，复制到名为value的新字符串中。接着再调用外部函数StringToNumber()，把这个字符串转换成整数值。然后，fi_GetData()就返回ivalue的值。

这个函数的写法很普遍。函数中没有直接写出搜寻词汇，只是通过搜寻指定文件，来搜寻词汇，再把相关联的整数值返回。如此一来我们要在程序代码中新增属性就很简单了。

此外，注意到，这里是游戏开发时检查错误的重要地方。如果除了游戏设计师之外，你还想让玩家使用描述系统，这就更具体了。你决不能假设任何分析中的脚本都是有效的。例如，你不能期望脚本撰写者所填入的数值，都会在合法值内。

描述对手行为

直接影响对手的行为，是描述机制在游戏软件AI中最常用的方式之一。前几例中已知描述属性可以对行为产生间接影响，包括修改生物智能属性的范例，大概都会在游戏中改变其行为。

描述行为可以让我们直接操纵AI对手的行动。然而，要让这有效，我们需要采取某种方式让脚本能看懂游戏世界和检查条件，以改变AI行为。为了做到这一点，我们可以新增预先定义好的全局变量到我们的描述系统里。实际的游戏引擎会替这些变量赋值，而不是由描述语言来指定。这些变量只是让脚本评估游戏世界里的特定情况。我们会在条件式描述语句中，用到这些全局变量。例如，在我们的描述系统里，我们可能有一个全局布尔变量，名为PlayerArmed，会让胆小的巨人只敢去突击没有武装的对手。例8-6给出了此脚本。

例8-6：基本行为脚本

```
If (PlayerArmed==TRUE)
   BEGIN
      DoFlee();
   END
ELSE
   BEGIN
      DoAttack();
   END
```

在例8-6中，脚本没有替PlayerArmed赋值，此变量是代表游戏引擎里的某个值。游戏引擎将评估此脚本，并把此行为连接到胆小的巨人身上。

此例中，PlayerArmed的值只是简单的布尔值，只是游戏引擎里的其中一个布尔值而已。这一点显然没什么好讲的，但当你用简单的全局变量，表示一系列更复杂的评估过程后的结果时，描述机制就更有用处了。例如，在此脚本范例中，我们要检查玩家是否武装。虽然知道对手有武装确实有用，但这并不见得表示对手在交战时是否难打。

很多因素都会对潜在对手的难度做出贡献。如果我们在游戏引擎中评估这些条件，再让结果以单一全局变量呈现给脚本使用，则我们的描述系统就更有用了。例如，我们可以用贝叶斯网络（Bayesian network）评估玩家是多么强悍的对手，然后，再将结果放在诸如PlayerChallenge之类的变量里让脚本使用。例8-7所示的脚本和例8-6一样简单，但是在游戏进行中，会产生精细得多的效果。

例8-7：行为脚本

```
If (PlayerChallenge ==DIFFICULT)
   BEGIN
```

```
      DoFlee();
   END
ELSE
   BEGIN
      DoAttack();
   END
```

就例8-7所说明的情况而言，PlayerChallenge可以表示一系列复杂的评估，而对玩家做出分级。其中有些因素是玩家是否武装、身上穿什么样的盔甲、当前玩家健康状况、该区中是否有其他玩家可以协防，等等。

可以描述行为的另一个方面就是AI角色的移动。我们可以利用某种思想，比如第三章提到的移动模式，在描述系统中得到运用。例如，游戏设计师替AI角色建立巡逻模式时，就会用得到。第三章中某些范例的做法是直接把移动模式写进程序代码中。当然，直接编写的行为有很多缺点。如果每次做出较小的修改之后都得重新编译，那就很难对游戏的设计做出调整了。图8-1是游戏设计师用描述系统实现的移动模式范例。

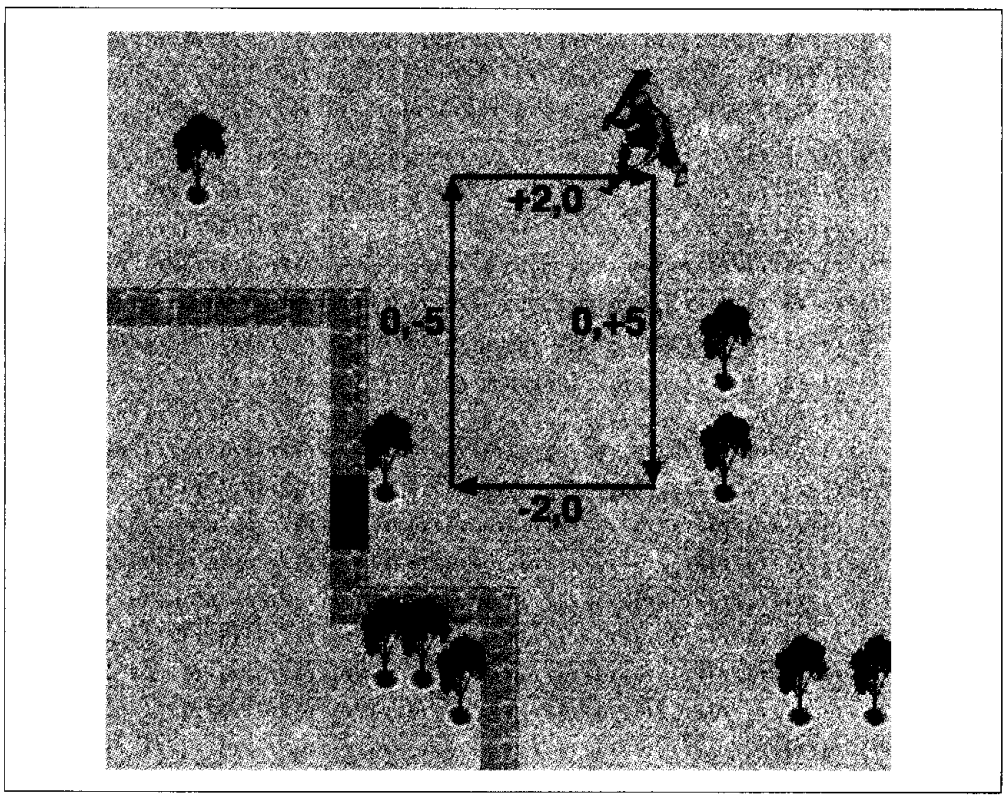


图8-1：描述式的移动模式

例8-8显示了我们如何编写一个脚本，借此实现所需的行为。

例8-8：*移动模式脚本*

```
If (creature.state==kPatrol)
   begin
      move(0,1);
      move(0,1);
      move(0,1);
      move(0,1);
      move(0,1);
      move(-1,0);
      move(-1,0);
      move(0,-1);
      move(0,-1);
      move(0,-1);
      move(0,-1);
      move(0,-1);
      move(0,1);
      move(0,1);
   end
```

由例8-8可窥见第九章要谈的有限状态机。在此描述范例中，如果AI生物处在巡逻状态，则使用指定的移动模式。每一步都是从前一位置移动一个单位。要了解移动模式技巧的细节说明，请参见第三章内容。

描述口语互动

描述机制的优点不仅仅是让AI对手变得更精致更有挑战性。许多整合智能行为的游戏，其做法都不是让玩家直接面对挑战。例如，角色扮演游戏可能会提供一系列模糊的指示，使其按照故事情节发展。描述机制是相当优秀的方法，让游戏设计师可以建立一个令人折服的故事情节，而无需修改实际的游戏程序。

智能行为可以让游戏更富有挑战性，而口语响应也属智能行为，更适用于进一步做出令玩家沉迷其中的游戏环境。口语互动（verbal interaction）的范围，可以从友善非玩家角色的辅助暗示到对手的嘲讽。口语互动似乎是最有智能的行为，对当前游戏情况来讲，也是最能引人入胜的手段。也就是说游戏软件AI必须检查一组已知的游戏参数，并据此做出响应。

例如，玩家的武装程度怎样，就是可以被检查的参数。然后，我们可以让敌方AI角色评论此种武器在接下来的战斗中，是如何无效。看起来这很有智能，也引人入胜，因为这不是随机的嘲讽，而是由当前游戏情况而定的。看起来就好像计算机控制角色，知道游戏里正在发生什么事情。例8-9是说明这种脚本的简单范例。

例8-9：口语嘲讽脚本

```
If (PlayerArmed ==Dagger)
   Say("What a cute little knife.");

      //" 好可爱的小刀 "
If (PlayerArmed ==Bow)
   Say("Drop the bow now and I'll let you live.");

      // 放下弓就让你活
If (PlayerArmed ==Sword)
   Say("That sword will fit nicely in my collection.");

      // 那把剑刚好让我当收藏品
If (PlayerArmed ==BattleAxe)
   Say("You're too weak to wield that battle axe.");
      // 你没力气挥动那把战斧
```

如例8-9所示，略微知道当前游戏的情况，能对游戏的进行增加引人入胜的效果。这一点比添加随机的普通嘲讽语，效果要好很多。

所以，描述系统的另一个重要方面，就是让脚本撰写者知道游戏引擎会发生什么事。脚本能看到的游戏元素越多越好。图8-2是一个假想的游戏场景，一个邪恶的巨人正在追逐玩家。就此而言，游戏软件AI可以使用游戏状态下特有的元素，根据当前情况，提供适当的嘲讽语。这个例子中，我们知道敌人是巨人，玩家是人类，而且玩家手上拿着一根棍子。

例8-10显示了游戏软件AI，如何在计算机控制的巨人和玩家控制的人类之间的战斗中，找出适当的嘲讽语。在实际游戏里，你可能会想替每个给定的情况多加几条响应话语，然后再在其中随机地选取。这样可以避免响应话语变得重复且可以预测。

例8-10：巨人嘲讽语脚本

```
If (Creature==Giant) and (player==Human)
   begin
      if (playerArmed==Staff)
         Say("You will need more than a staff, puny human!");

            // 你需要更多的棍子吧，小矮人!
      if (playerArmed==Sword)
         Say("Drop your sword and I might not crush you!");

            // 放下你的剑，否则我就打扁你!
      if (playerArmed==Dagger)
         Say("Your tiny dagger is no match for my club!");
            // 你的小刀抵不上我的棍子
      end
```

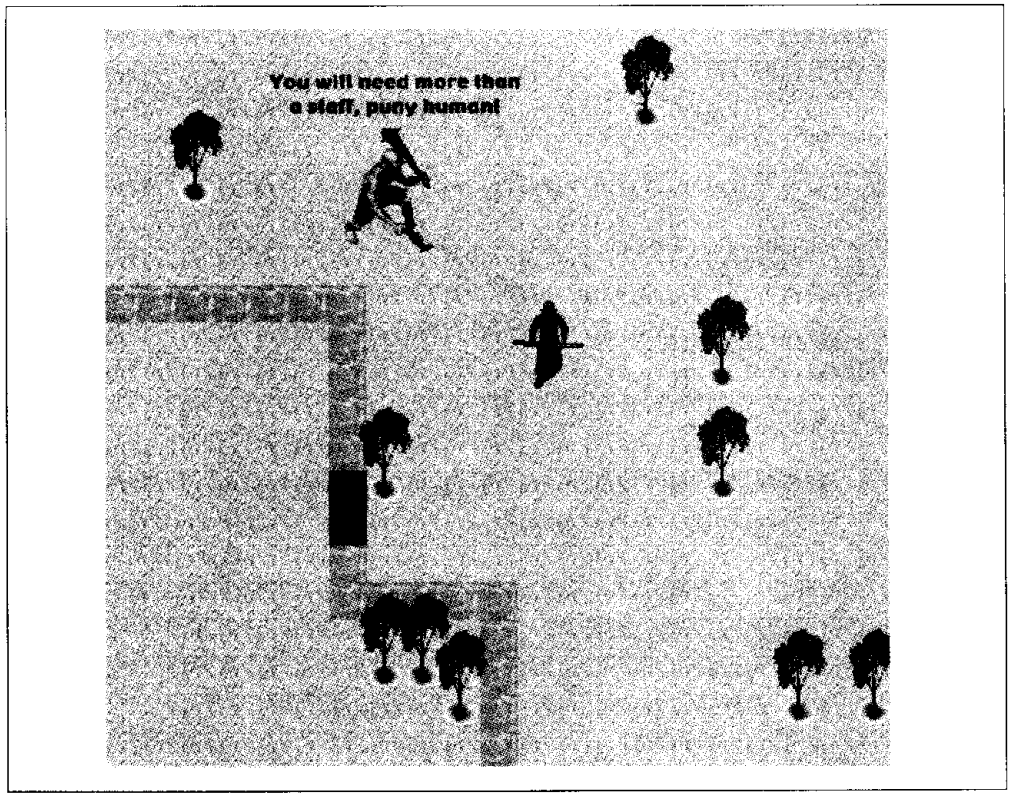


图8-2：巨人的嘲讽语

当然，这种描述机制不限于那些要杀玩家的敌方角色。慈善的计算机控制角色也能利用相同的技巧。这样可以协助脚本撰写者，建立迷人又引人入胜的情节。例8-11说明了脚本如何协助建立情节，并引导玩家的行为，走向游戏的目的地。

例8-11：慈善的AI脚本

```
If (Creature=FriendlyWizard)
   begin
      if (playerHas==RedAmulet)
         Say("I see you found the Red Amulet.
              Bring it to the stone temple
              and you will be rewarded.");
            // 你找到了红色护身符，把它拿到石庙，你将得到奖赏
   end
```

如例8-11所示，直到护身符被找到，而且玩家面对友善的法师时，有关护身符应该放在何处的重要信息才会展现出来。

前几个脚本范例让你知道了游戏软件AI可以在指定情况下做出响应，但是有时候，游戏角色还需要和玩家做某种类型的口语互动。这可能是慈善的角色，有意要提供有用的信息给玩家；或者，也可能是某个不诚实的角色，刻意要误导玩家。

在这种场景中，玩家必须以某种机制把文字输入给游戏。然后，游戏引擎再把文字字符串送交给描述系统，描述系统再分析文字，并提供适当的响应。图8-3显示了在实际游戏里的画面。

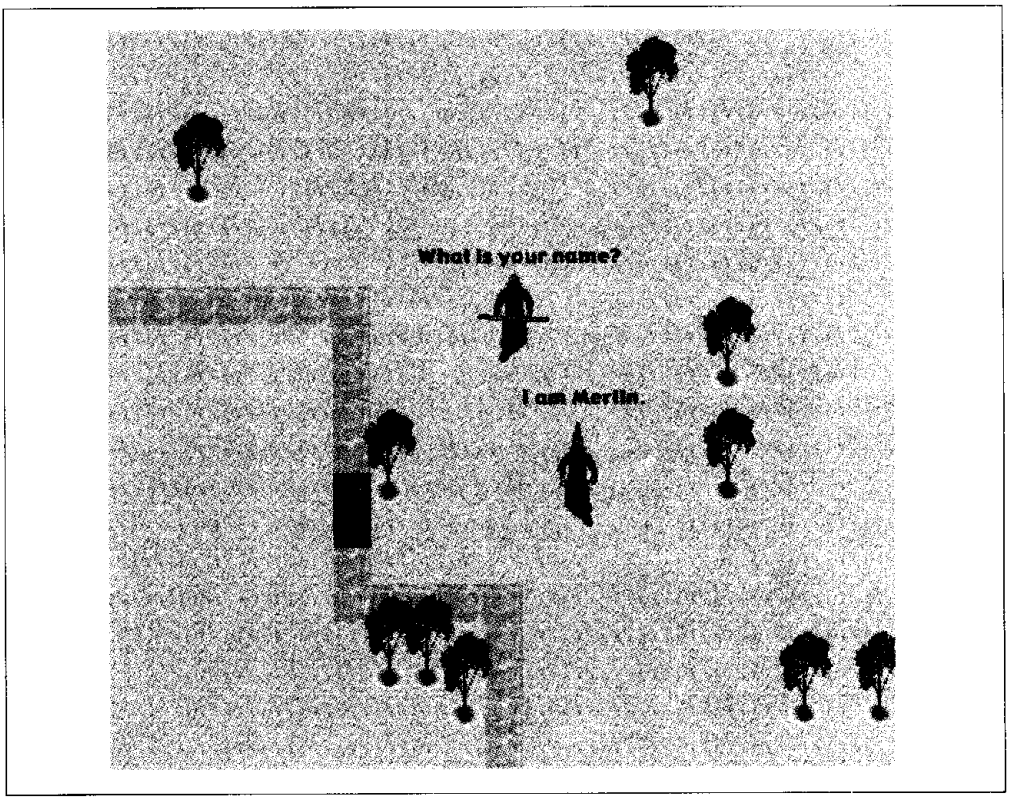


图8-3：梅林法师

就图8-3所示的情况而言，玩家输入了“What is your name?”，而描述系统做出响应的文字是“I am Merlin.”。例8-12是用于实现此种做法的基本脚本。

例8-12：基本的询名脚本

```
If Ask("What is your name?")
begin
Say("I am Merlin.");
end
```

当然，例8-12有一个严重的缺点。只有当玩家输入和脚本中的问题完全一样的文字时，才能起作用。事实上，你可以用好几种方式组成一个问题。例如，如果玩家输入例8-13中的其中一行，将会发生什么事？

例8-13：玩家输入实例

```
What's your name?
Whats your name?
What is your name.
What is thy name?
What is your name, Wizard?
Hello, what is your name?
```

由此可见，如果输入例8-13中的任何问题，例8-12的脚本都会出错，即使我们一看便知这些问题是问什么。你不但可以用许多方式提出问题，而且我们还必须考虑玩家可能不会以正确的方式提出问题。事实上，你可以发现，该范例中，有一行是以句号结束，而不是问号。我们可以严格要求玩家输入文字，但是这会让玩家对游戏失去兴趣，因为，无论玩家犯了多么不可避免的小错都是不允许的。

检查每个口语文字字符串的另一种做法是建立语言分析器，解析每个句子，确定其到底在问什么。对某些游戏而言，有一个高级的语言分析器也许很恰当，然而，对多数游戏而言，则有一个比较简单的做法。如例8-13所示，你可以用相当多的方式提出相同的问题，但如果你注意一下，就会发现他们都有一些共同的东西，比如都有“what”和“name”这些词。所以，此时不要去逐字地检查每个文字字符串，我们只需搜寻特定的关键字，并据此做出响应。就此而言，描述引擎只会检查文字字符串中是否有指定关键字存在。

如例8-14所示，脚本将检查玩家所输入的文字中，是否有“what”和“name”两个关键字存在。使用这种方法，脚本就能对例8-13中所提出的每个问题做出正确的响应了。

例8-14：关键字描述机制

```
If (Ask("what") and Ask("name") )
   begin
      Say("I am Merlin.");
   end
```

现在，我们已经教给你如何写典型的脚本了，可以根据一组给定的关键字来检查玩家输入的文字。下面我们来看看实际的游戏引擎，将如何根据指定关键字检查玩家输入的文字。例8-15就是这种做法。

例8-15：搜寻关键字

```
Boolean FoundKeyword(char inputText[kStringLength], char
searchFor[kStringLength])
```

```
{
   char    inStr[kStringLength];
   char    searchStr[kStringLength];
   int     i;

   for (i=0;i<=strlen(inputText);i++)
      {
         inStr[i]=inputText[i];
         if (((int)inStr[i]>=65) && ((int)inStr[i]<=90))
            inStr[i]=(char)((int)inStr[i]+32);
      }

   for (i=0;i<=strlen(searchFor);i++)
      {
         searchStr[i]=searchFor[i];
         if (((int)searchStr[i]>=65) &&
             ((int)searchStr[i]<=90))
            searchStr[i]=(char)((int)searchStr[i]+32);
      }

   if (strstr(inStr,searchStr)!=NULL)
      return (true);

   return (false);
}
```

例8-15是游戏引擎中实际的程序代码，当游戏设计师的脚本中使用了Ask()函数时，就会被调用。这个函数有两个参数：inputText（玩家输入的文字）以及searchFor（要搜寻的关键字）。我们在此函数中所做的第一件事，就是把字符串都转换成小写。就像许多程序语言那样，C和C++都是分大小写的。含有“Name”这个词的字符串和含有“name”这个词的字符串是两个不相等的字符串。我们不能期待玩家总是以一贯或正确的方式，使用大小写规则。最简单的解决办法就是全都转换成小写。这样一来，无论玩家都用大写、都用小写或者大小写混用，就都无关紧要了。

一旦有两个小写字符串，我们可以调用C函数strstr()来比较两字符串。strstr()函数会搜寻inStr中首次出现searchStr的地方。如果在inStr中找不到searchStr，就会返回空指针。

描述事件

现在，让我们检视其他可以让游戏的进行更加引人入胜的描述机制。前几节谈的是脚本如何改变AI角色的行为。描述行为需要费很大力才能让游戏和AI角色看起来很真实。然而，你可以利用描述机制，在其他方面让游戏具有娱乐性和真实感。本节中，我们要检视脚本如何触发与AI角色可能不太有直接关联的游戏事件。例如，也许站在特定位置上，就会触发一个陷阱。例8-16是基于文字的描述语言程序。

例 8-16：陷阱事件脚本

```
If (PlayerLocation(120,76))
   Trigger(kExposionTrap);

If (PlayerLocation(56,16))
   Trigger(kPoisonTrap);
```

如例8-16所示，描述系统可以把玩家位置和某些默认值相比较，如果二者相等，就触发陷阱。当然，你可以把触发机制做得更复杂一些，让这个过程更精致。也许，只有当玩家手中握有某种机器，或者身穿特定盔甲时，才会触发陷阱。

描述机制也是在游戏进行中增加某种气氛的有效方式。例如，你可以把某种情况或物体与特定的声音效果相连接。如果玩家在甲板上走动，就可以启用海鸥的音效。你可以将整个脚本文件，把音效连接到不同的情况下。

图 8-4 显示了玩家站在门口的情况，这是连接门轴吱吱响的声音效果的绝佳情况。

例 8-17 说明了玩家的位置或游戏的情况（比如游戏时间），如何触发相关的音效。

例 8-17：触发音效脚本

```
If (PlayerLocation(kDoorway))
  PlaySound(kCreakingDoorSnd);

If (PlayerLocation(kDock))
  PlaySound (kSeagullSnd);

If (PlayerLocation(kBoat))
  PlaySound (kWavesSnd);

If (GameTime==kNight)
  PlaySound (kCricketsSnd);

If (GameTime ==kDay)
  PlaySound (kBirdsSnd);
```

虽然本章对AI描述机制的类型做了分类，但在实际游戏中，将它们结合使用的话是有益处的。例如，不要在玩家行动时触发诸如声音这样的效果，也可以触发特定生物AI巡逻模式。我们也谈到好几个范例，说明了AI生物如何响应玩家输入的文字，然而，这也是触发游戏事件中相当有用的方法。例如，玩家诵读某个咒语时，就会触发某事件。

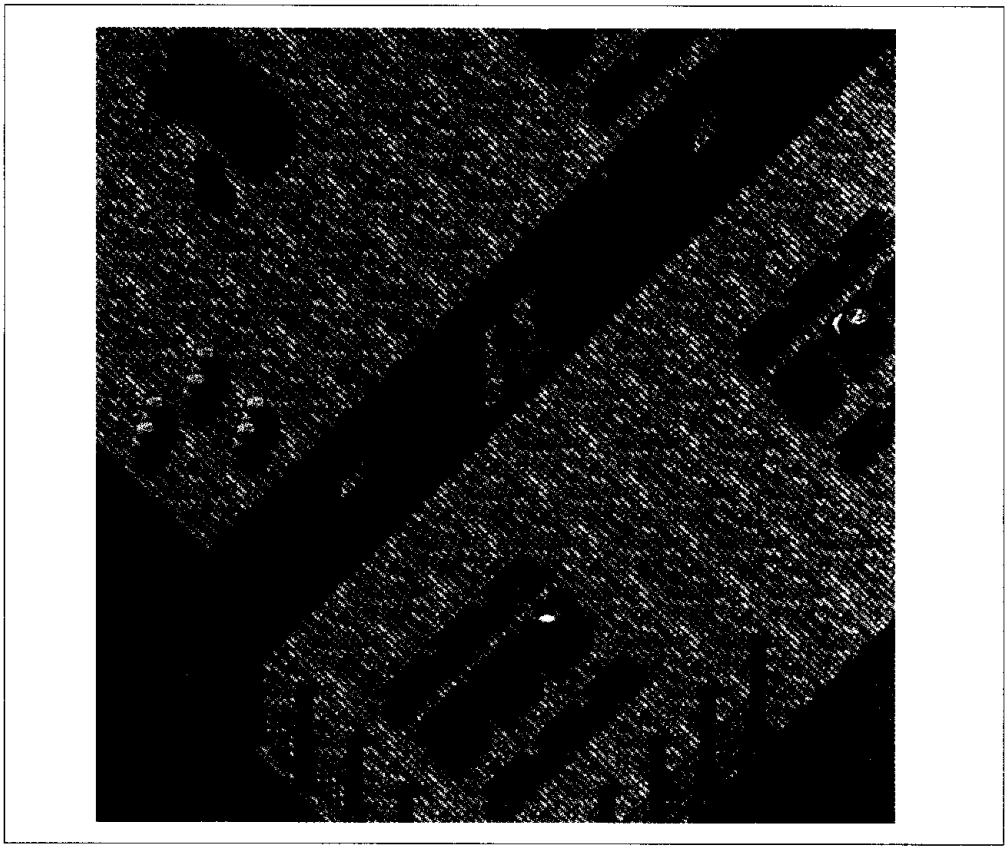


图 8-4：门轴音效脚本

其他信息

本章教你如何操作基本描述机制，让你能在主要游戏程序外修改游戏软件AI。像这样的描述机制是非常有效的。事实上，我们在一个MMORG游戏中成功地应用了这些技巧，让游戏管理者能实时修改游戏软件AI和其他游戏参数。实现一个成熟的描述引擎是很困难的，而且牵涉到其他我们还没谈到的概念。这些概念包括有限状态机（finite state machine）以及规则系统，我们在第九章和第十一章会谈到。

如果你决定继续探索本书所谈的描述机制以外的东西，下列资源对你很有用：

- 《AI Application Programming》，M.Tim Jones 著（Charles River Media）
- 《AI Game Programming Wisdom》，Steve Rabin 主编（Charles River Media）

第一本参考书中，作者Tim Jones教你从头实现描述式规则系统。他的做法结合了本章所谈到的观念，以及我们将在第十一章谈到的观念。第二本参考书中，有七篇游戏程序员所写的文章，专门谈实现游戏所用描述引擎的相关议题。

第九章

有限状态机

有限状态机（finite state machine）是一种抽象机制，是处在各种不同的预定状态下的其中一种状态。有限状态机也可以定义一组条件，以确认何时应该改变状态。实际的状态会决定状态机的行为。

有限状态机可追溯到早期计算机游戏程序设计的年代。例如，“Pac-Man”（小精灵）游戏里的魔鬼就是有限状态机。这些魔鬼可以四处游走，追逐玩家，或者闪躲玩家。在每种状态下，他们的行为都不同，而状态间的转换则由玩家的行动来决定。例如，如果玩家吃了大力丸，魔鬼的状态也许会从追逐转换成闪躲。下一节再回来谈此例。

虽然有限状态机已经存在很长一段时间了，但仍然被广泛采用，在现代游戏里也很常见和有用。由于相当容易理解、实现以及调试，这项事实使得有限状态机在游戏开发过程中，时常做出贡献。本章我们要讨论有限状态机的基础，教你如何实现它。

状态机的基本模型

由图 9-1 可知如何模拟简单的有限状态机。

如图 9-1 所示，每个可能的状态都以圆圈表示，此图中有四种可能状态：*Si*、*S1*、*S2* 以及 *S3*。当然，每个有限状态机，需要采用一种方法从一种状态转换到另一种状态。就此而言，转换函数以 *t1*、*t2*、*t3*、*t4* 以及 *t5* 来表示。有限状态机一开始是初始状态 *Si*，一直到 *t1* 转换函数提供刺激值才会改变其状态。一旦提供了刺激值，状态就转换成 *S1*。此时，你可以轻易看出从一种状态转换成另一种状态，是由哪个转换函数所提供的刺激值。在某些情况下只有一种可能，以 *S1* 为例，只有 *t5* 提供的刺激值才可以改变该机制的状态。然而，注意到在 *S3* 和 *S2* 情况下，有两种可能的刺激值可以改变状态。

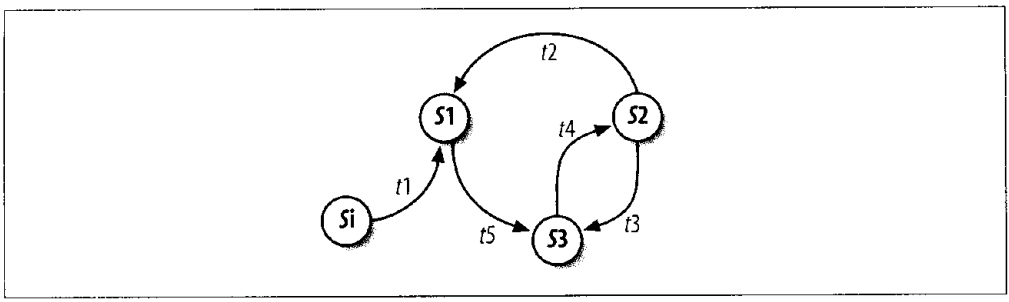


图9-1：一般有限状态机图

现在，我们已经给你提供了简单的状态机模型，让我们看一个更实际且有点复杂的范例。图9-2是实际游戏中可能出现的有限状态机。

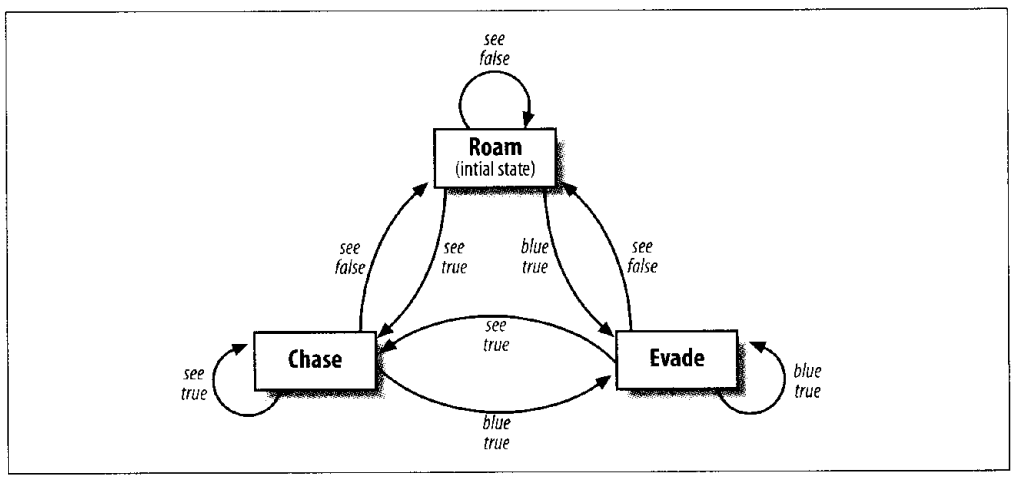


图9-2：鬼怪有限状态机图

仔细看图9-2之后，你会发现有限状态机所模拟的行为与“Pac-Man”游戏里的鬼怪很类似。每个方块代表一种可能的状态。就此例而言，有三种可能状态：游走（roam）、闪躲（evade）以及追逐（chase）。箭头表示可能存在的转换，同时也显示出哪些条件下这几种状态可以改变或维持不变。

就此而言，计算机控制的AI对手是从最初的“游走”状态开始的。有两个条件可以引发状态改变。第一个条件是“我已变为蓝色”。就此而言，当玩家吃了大力丸之后，AI对手就会变为蓝色。这样就会使其状态从“游走”转换为“闪躲”。另一个可以改变状态的条件是“看得见玩家”，也就是说游戏软件AI可以看见玩家，因此从“游走”状态转

换为“追逐”。现在，再也没有必要四处游走了，因为游戏软件AI能看到玩家而去追逐了。

此图还显示出，当有限状态机是蓝色时，会持续保持“闪躲”状态。除此之外，如果能看见玩家，则状态会转变为“追逐”。如果看不见玩家，则会回复到“游走”状态。同样的，该机制会持续保持“追逐”状态，直到其转为蓝色，此时，就会转变为“闪躲”状态。然而，如果正在追逐玩家，但后来玩家的踪影消失了，则再次转变为“游走”状态。

现在，我们已经教给你如何在有限状态机图形中仿真这种行为，接着，让我们来看一下如何实际写程序代码来实现此行为。例9-1就是该程序代码。

例9-1：魔鬼的行为

```
switch (currentState)
   {

      case kRoam:
          if (imBlue==true) currentState=kEvade;
          else if (canSeePlayer==true) currentState=kChase;
          else if (canSeePlayer==false) currentState=kRoam;
          break;


      case kChase:
          if (imBlue==true) currentState=kEvade;
          else if (canSeePlayer==false) currentState=kRoam;
          else if (canSeePlayer==true) currentState=kChase;
          break;

      case kEvade:
          if (imBlue==true) currentState=kEvade;
          else if (canSeePlayer==true) currentState=kChase;
          else if (canSeePlayer==false) currentState=kRoam;
          break;
   }
```

例9-1所示的程序不一定是此问题最有效的解法，但的确能让你知道如何用实际的程序代码，仿真如图9-2所示的行为。就此而言，switch语句会检查三种可能状态：kRoam、kChase以及kEvade。每个switch语句里的case都会检查可能的条件，以便转换状态或维持不变。注意，在每个case里，imBlue的条件有优先判断权。如果imBlue是true，则状态会自动切换成kEvade，而不管其他条件如何。只要imBlue为true，此有限状态机就会维持kEvade状态。

设计有限状态机

现在，我们要讨论某些方法，你可以运用这些方法在游戏中实现有限状态机。实际上，有限状态机对游戏软件AI的开发有很大帮助，能够以简单而合乎逻辑的方式，控制游戏软件AI行为。事实上，很多游戏可能都用上了有限状态机，只是开发人员没有发觉，用到有限状态机模型而已。

我们开始时将这项工作分成两部分。首先，讨论储存与游戏软件AI实体相关的数据的几种结构类型。然后，我们再讨论如何建立函数转换状态机的状态。

有限状态机的结构和类

使用高级语言开发的游戏，诸如C或C++，通常会把所有与每个游戏软件AI实体相关的数据，都储存在单一结构或类里。像这样的结构可以储存如位置、健康、体力、特殊能力以及财物清单等诸如此类的值。当然，除了这些元素之外，结构还可以储存当前AI的状态，而最终决定AI行为的就是这种状态。例9-2说明了典型的游戏怎样以单一类结构，储存游戏软件AI实体的数据。

例9-2：游戏软件AI结构

```
class AIEntity
   {
      public:
         int type;
         int state;
         int row;
         int column;
         int health;
         int strength;
         int intelligence;
         int magic;
   };
```

在例9-2中，类里的第一个元素参照此实体的类型，这可以是任何事物，比如巨人、人类或星际战舰。类里的下一个元素是本章我们最关注的，AI状态就储存在这里。类结构中剩余的变量，只是和游戏软件AI实体有关的特征值。

状态本身通常使用全局常量赋值。新增一种状态，就只需新增一个全局常量，如此而已。例9-3教给你如何定义这样的常量。

例9-3：状态常量

```
#define   kRoam              1
#define   kEvade             2
```

```
#define   kAttack              3
#define   kHide                4
```

现在，我们已经知道了AI状态和重要数据如何群集在单一类结构里，接下来要看的是如何替该类结构新增转换函数（transition function）。

有限状态机行为及转换函数

实现有限状态机的下一步是提供函数，以决定AI实体的行为以及何时应该改变状态。例9-4教给你如何把行为和转换函数加进AI类结构里。

例9-4：游戏软件AI转换函数

```
class   AIEntity
   {
      public:
         int type;
         int state;
         int row;
         int column;
         int health;
         int strength;
         int intelligence;
         int magic;
         int armed;

         Boolean playerInRange();
         int checkHealth();
   };
```

你可以发现我们在AIEntity类里新增了两个函数。当然，在实际游戏中，你可能会使用很多函数来控制AI行为，改变AI状态。然而，就此例而言，这两个转换函数就足够能说明如何可改变AI实体的状态了。例9-5说明了如何利用这两个转换函数，改变该机制的状态。

例9-5：改变状态

```
if ((checkHealth()<kPoorHealth) && (playerInRange()==false))
   state=kHide;
else if (checkHealth()<kPoorHealth)
   state=kEvade;
else if (playerInRange())
   state=kAttack;
else
   state=kRoam;
```

例9-5中的第一个if语句用来检查AI实体的健康值是否太低，以及玩家是否不在附近。如果这些条件都成立，则由此类结构所表示的生物，将进入“隐藏”状态。该机制可能

会一直保持此状态，直到其健康值增加为止。第二个 if 语句只是用于检查健康值的下限。如果我们运行到这个 if 语句，就表示玩家在附近；如果不在附近，第一个 if 语句的评估结果就成立。由于玩家在附近，隐藏不是上策，因为玩家可能会看见此 AI 实体。就此而言，比较恰当的做法是闪躲玩家。第三个 if 语句检查玩家是否在附近。同样的，我们知道此时 AI 的健康状况良好，否则，前两个 if 语句的其中一个就应该成立了。由于玩家在附近，而 AI 实体健康状况良好，于是其状态便改为“攻击”。如果没有其他选项可选，就采用最后的状态选项。就此例而言，我们会进入默认的“游走”状态。此例中的生物会一直保持“游走”状态，直到转换函数指定的条件指出该状态应该改变为止。

前几节谈的是为简单有限状态机建立类结构和转换函数的基础。下一节我们要运用这些概念，做出一个成熟的有限状态机实例。

蚂蚁实例

此例的有限状态机目标，将让这个有限状态机仿真两组 AI 蚂蚁。模拟蚂蚁的目的是令其收集食物并返回巢穴。仿真的蚂蚁会避开某些障碍物并遵守某些规则。首先，蚂蚁会在其环境中随机移动，试图找到食物。一旦有一只蚂蚁找到了一块食物，它就返回巢穴。当蚂蚁到达巢穴时，会放下食物，然后开始找水，而不是食物。口渴的蚂蚁会随机地到处找水。一旦蚂蚁找到水后，它又会去找更多的食物。

把食物带回巢穴之后，巢穴会生出一只新蚂蚁。只要有食物不断被带回巢穴，蚂蚁群就会不断增长。当然，蚂蚁在路上也会碰到障碍物。除了随机置放的食物外，毒物也是随机置放的。不用说，毒物能让蚂蚁一命呜呼。

图 9-3 是说明仿真蚂蚁行为的有限状态机图。

如图 9-3 所示，每只蚂蚁的初始状态都是“寻找食物”（forage）。从这里开始，其状态就只能以两种方式进行改变。其状态可以经由“找到食物”的转换函数而改成“返回巢穴”（go home），也有可能碰上“找到毒物”转换函数而被杀死。一旦找到食物，其状态就会改成“返回巢穴”。同样的，也只有两种方式可以改变此状态。一种是达到目标并找到巢穴，这由“找到巢穴”的转换箭头表示。另一种可能的转换是“找到毒物”。一旦找到巢穴，状态就会改成“口渴”。如前几种状态那样，此时也只有两种方式可以改变状态。一是“找到水”，二是“找到毒物”。如果达成目标，蚂蚁就会回到初始的“寻找食物”状态。

由此可见，蚂蚁试图完成其任务时，会处在诸多不同状态下的其中一种。每种状态都代表不同的行为。所以，我们现在要用前面提到的规则，来定义仿真 AI 蚂蚁的可能状态，如例 9-6 所示。

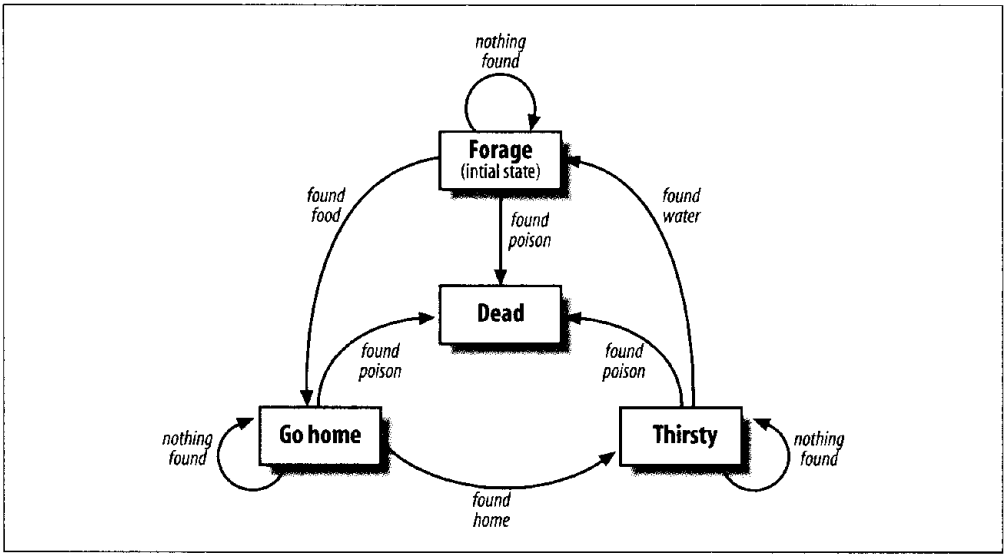


图 9-3：蚂蚁有限状态机图

例 9-6：蚂蚁的状态

```
#define   kForage          1
#define   kGoHome          2
#define   kThirsty         3
#define   kDead            4
```

此次仿真的第一条规则是蚂蚁随机寻找食物，由kForage状态定义；任何处在kForage状态的蚂蚁，都会在其环境内随机移动搜寻食物。一旦蚂蚁找到食物，就改变成kGoHome状态；在此状态下，蚂蚁会返回其巢穴，而且在kGoHome状态下，蚂蚁不再寻找食物，在返回巢穴途中，会忽略任何食物。如果蚂蚁成功返回巢穴，而且没有碰上毒物，就改变成kThirsty状态；此状态类似“寻找食物”状态，但此时寻找的不是食物，而是水。一旦找到水之后，蚂蚁将从kThirsy状态回复到kForage状态。然后重复同样的行为。

有限状态机类及结构

现在，我们已经说明了模拟蚂蚁有限状态机的基本目标，接着要看即将用到的数据结构。如例 9-7 所示，我们用了一个 C++ 类，储存所有和每个有限状态机蚂蚁相关的数据。

例 9-7：ai_Entity 类

```
#define   kMaxEntities        200
```

```
class     ai_Entity
{
   public:
   int                  type;
   int                  state;
   int                  row;
   int                  col;

   ai_Entity();
   ~ai_Entity();

};

ai_Entity      entityList[kMaxEntities];
```

如例9-7所示，我们的C++类中有四个变量。第一个变量是type，也就是这个结构所代表的AI实体的类型。

如果你还记得前面所说的，就知道仿真的蚂蚁有两组。我们可以利用例9-8中的常量来区分。

例9-8：组别常量

```
#define    kRedAnt          1
#define    kBlackAnt        2
```

ai_Entity类的第二个变量是state，储存该蚂蚁的当前状态，可以是例9-6定义的任何数值，也就是kForage、kGoHome、kThirsty以及kDead。

最后两个变量是row和col。这表示仿真蚂蚁在砖块环境中的位置；即row和col变量储存的是蚂蚁在砖块世界中的位置。

如例9-7所示，我们可以建立一个数组，储存每只蚂蚁的数据。数组中每个元素可代表一只蚂蚁。此次仿真的蚂蚁数量受到常量限制，同样定义在例9-7中。

定义仿真世界

如前所述，仿真蚂蚁会在砖块环境中占据位置。这个世界以一个二维整数数组表示。例9-9是常量和数组声明。

例9-9：地形数组

```
   #define    kMaxRows        32
   #define    kMaxCols        42

   int    terrain[kMaxRows][kMaxCols];
```

地形数组的每个元素储存的是环境中的砖块值。这个世界的大小由kMaxRows和kMaxCols常量定义。实际的砖块游戏，可能对每个砖块的可能值有很多选择。然而，就此次仿真而言，我们只用六个可能值。例9-10说明了其常量。

例9-10：地形值

```
#define   kGround          1
#define   kWater           2
#define   kBlackHome       3
#define   kRedHome         4
#define   kPoison          5
#define   kFood            6
```

砖块环境的默认值是kGround，你可以把这个值想象成空白。下一个常量kWater是蚂蚁处在kThirsy状态时要搜寻的元素。下面两个常量是kBlackHome和kRedHome。这是蚂蚁处在kGoHome状态时要找的巢穴。走到含有kPoison元素的砖块时，蚂蚁就会被杀死，并将其状态改成kDead。最后一个常量是kFood。当蚂蚁处在kForage状态，而且走到含有kFood地形元素的砖块时，则其状态会从kForage转换成kGoHome。

一旦变量和常量都声明之后，我们可以利用例9-11所示的程序代码，对这个世界做初始化。

例9-11：对这个世界做初始化

```
#define   kRedHomeRow          5
#define   kRedHomeCol          5

#define   kBlackHomeRow        5
#define   kBlackHomeCol       36
for (i=0;i<kMaxRows;i++)
   for (j=0;j<kMaxCols;j++)
      {
         terrain[i][j]=kGround;
      }

terrain[kRedHomeRow][kRedHomeCol]=kRedHome;
terrain[kBlackHomeRow][kBlackHomeCol]=kBlackHome;

for (i=0;i<kMaxWater;i++)
   terrain[Rnd(2,kMaxRows)-3][Rnd(1,kMaxCols)-1]=kWater;

for (i=0;i<kMaxPoison;i++)
   terrain[Rnd(2,kMaxRows)-3][Rnd(1,kMaxCols)-1]=kPoison;

for (i=0;i<kMaxFood;i++)
   terrain[Rnd(2,kMaxRows)-3][Rnd(1,kMaxCols)-1]=kFood;
```

例9-11把二维数组的值全部赋为kGround。记住，这是默认值。然后，我们设定两个巢穴的位置。实际的位置是由kRedHomeRow、kRedHomeCol、kBlackHomeRow以及

kBlackHomeCol常量来决定。当蚂蚁处在kGoHome状态时，这些位置就是蚂蚁移动的地点。每只蚂蚁都朝其各自颜色的地点移动。

例9-11的最后一部分是三个for循环，随机置放kWater、kPoison以及kFood砖块。每种类型的砖块数量由各自的常量决定。当然，如果改变这些值的话，就可以改变仿真蚂蚁的行为了。

图9-4所示的是初始化后的砖块世界。我们还没有让任何有限状态机蚂蚁移居到这个世界，但我们确实随机放置了食物、水和毒物。

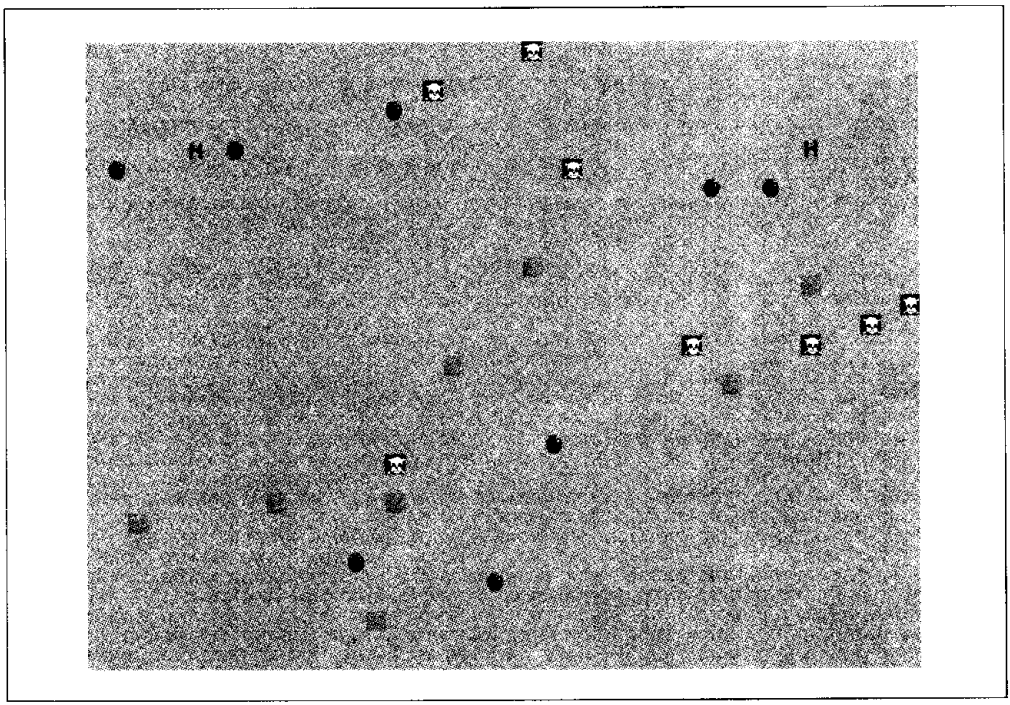


图9-4：蚂蚁世界

现在，我们已经用砖块对相关变量做初始化了，下一步就是将AI蚂蚁移居到这个世界。

移居世界

我们要做的第一件事是找到建立新AI实体的办法。为了达到这个目的，我们打算替ai_Entity类新增一个函数。例9-12就是ai_Entity类新增函数后的结果。

例 9-12：ai_Entity 类

```
class      ai_Entity
{
   public:
   int                type;
   int                state;
   int                row;
   int                col;

   ai_Entity();
   ~ai_Entity();

   void  New (int theType, int theState, int theRow, int theCol);

};
```

无论何时在此世界新增蚂蚁，都可以调用New()函数。仿真刚开始时，我们在此世界中只替每种颜色的蚂蚁各新增两只。然而，每次只要有蚂蚁把食物成功带回巢穴，就再次调用此函数。例 9-13 是此函数的全部内容。

例 9-13：New() ai_Entity

```
void ai_Entity::New(int  theType, int theState, int theRow, int theCol)
{
   int   i;

   type=theType;
   row=theRow;
   col=theCol;
   state=theState;
}
```

New()函数相当简单，只对ai_Entity实体中的四个变量赋初值，包括此实体的类型、状态以及行和列的位置。现在，让我们看一看例 9-14，来了解New()函数如何替此有限状态机仿真程序新增蚂蚁。

例 9-14：新增蚂蚁

```
   entityList[0].New(kRedAnt,kForage,5,5);
   entityList[1].New(kRedAnt,kForage,8,5);

   entityList[2].New(kBlackAnt,kForage,5,36);
   entityList[3].New(kBlackAnt,kForage,8,36);
```

如例 9-14 所示，仿真程序一开始是在此世界中新增四只蚂蚁。传给New()函数的第一个参数是指定此实体的类型。在此仿真中，是先从两只红蚂蚁和两只黑蚂蚁开始的。第二个参数是有限状态机蚂蚁的最初状态。最后两个参数是行和列的位置，也就是每只蚂蚁的起始地点。

如图9-5所示，我们使用New()函数在此仿真世界中新增了四只蚂蚁。

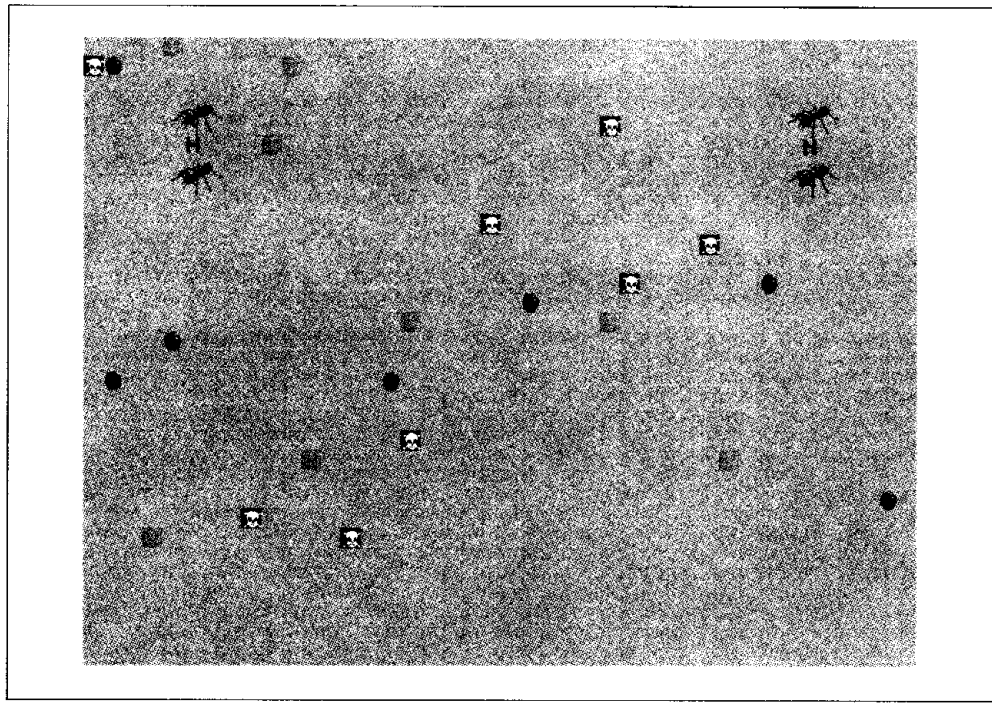


图9-5：移居世界

如图9-5所示的每只蚂蚁的初始状态都设为kForage。从初始位置开始，他们将随机在此砖块世界中移动、搜寻食物。就此例而言，食物是苹果。然而，如果踩到毒物（如图所示的骷髅头和两根交叉的骨头），就会转换成kDead状态。像水的形状的方块就是他们处在kThirsty状态时要搜寻的元素。

更新此世界

上一节我们把四只有限状态机蚂蚁移居到此世界。现在，要教你如何操作此仿真世界。这是有限状态机操作程序的关键所在。回想一下，有限状态机的基本前提是把个别独立的状态，连接到不同种类的行为。这就是我们实际让每只蚂蚁，根据其状态而从事特定行为的程序代码部分。这部分程序代码是用switch语句，检查每种可能的状态。在实际游戏中，这个switch语句在每轮主要循环进行时都会被调用。从例9-15可知这个switch语句的用法。

例 9-15：操作仿真世界

```
for (i=0;i<kMaxEntities;i++)
   {
      switch (entityList[i].state)
         {

            case kForage:
               entityList[i].Forage();
               break;

            case kGoHome:
               entityList[i].GoHome();
               break;

            case kThirsty:
               entityList[i].Thirsty();
               break;

            case kDead:
               entityList[i].Dead();
               break;

         }
   }
```

如例 9-15 所示，我们建立一个循环走遍 entityList 数组中的每个元素。每个 entityList 元素代表不同的有限状态机蚂蚁。我们以 switch 语句检查 entityList 的每只蚂蚁的状态。注意，每个可能的状态都有一个 case 语句。然后，我们把每种状态都连接到特定行为，也就是调用每个行为所属的适当函数。如例 9-16 所示，我们必须在 ai_Entity 类中新增四个新函数。

例 9-16：ai_Entity 类的函数

```
class    ai_Entity
{
   public:
   int                type;
   int                state;
   int                row;
   int                col;

   ai_Entity();
   ~ai_Entity();

   void  New (int theType, int theState, int theRow, int theCol);

   void Forage(void);
   void GoHome(void);
   void Thirsty(void);
   void Dead(void);
};
```

例9-16是更新后的ai_Entity类，增加了四个新的行为函数。每个函数都和对应的状态行为有关。

Forage函数

第一个新函数是Forage()，和kForage状态有关。回想一下，在此状态下，蚂蚁会随机在此世界中移动、搜寻食物。处在“寻找食物”的状态下时，只有通过两种方式才可以切换到另一种不同状态。第一种方式是达到目标，也就是随机找到食物。就此而言，其状态会切换成kGoHome。另一种切换状态的方式就是踩到毒物。就此而言，其状态会切换成kDead。这个行为是由Forage()函数实现，如例9-17所示。

例9-17：Forage()函数

```
void ai_Entity::Forage(void)

{
   int rowMove;
   int colMove;
   int newRow;
   int newCol;
   int foodRow;
   int foodCol;
   int poisonRow;
   int poisonCol;

   rowMove=Rnd(0,2)-1;
   colMove=Rnd(0,2)-1;

   newRow=row+rowMove;
   newCol=col+colMove;

   if (newRow<1) return;
   if (newCol<1) return;
   if (newRow>=kMaxRows-1) return;
   if (newCol>=kMaxCols-1) return;

   if ((terrain[newRow][newCol]==kGround) ||
       (terrain[newRow][newCol]==kWater))
      {
         row=newRow;
         col=newCol;
      }

   if (terrain[newRow][newCol]==kFood)
      {
         row=newRow;
         col=newCol;
         terrain[row][col]=kGround;
         state=kGoHome;
         do {
```

```
            foodRow=Rnd(2,kMaxRows)-3;
            foodCol=Rnd(2,kMaxCols)-3;
         } while (terrain[foodRow][foodCol]!=kGround);
         terrain[foodRow][foodCol]=kFood;
      }

   if (terrain[newRow][newCol]==kPoison)
      {
         row=newRow;
         col=newCol;
         terrain[row][col]=kGround;
         state=kDead;
         do {
            poisonRow=Rnd(2,kMaxRows)-3;
            poisonCol=Rnd(2,kMaxCols)-3;
         } while (terrain[poisonRow][poisonCol]!=kGround);
         terrain[poisonRow][poisonCol]=kPoison;
      }
}
```

Forage()函数一开始声明了八个变量。前两个是rowMove和colMove。这两个变量储存的是行和列两方向上要移到的距离。下面两个变量是newRow和newCol。这两个变量储存的是蚂蚁新的行和列的位置。最后四个变量foodRow、foodCol、poisonRow以及poisonCol用来储存任何可能被消化掉的食物或毒物的新位置。

然后，我们要计算出新位置。起初是在-1和+1之间随机找一个数字，赋给rowMove和colMove变量。这样可以确保蚂蚁在砖块环境中，移动到八个可能方向的其中之一。也有可能两个值都是0，此时，蚂蚁将待在原地。

一旦rowMove和colMove赋值之后，我们要将其值加进当前行和列的位置，并将结果储存在newRow和newCol之内。这就是蚂蚁的新位置了，当然，其前提是这在砖块环境中是合法位置。事实上，下一区块的if语句会检查新位置，是否位于砖块环境的合法区域内。如果不是合法位置，就跳离此函数。

现在，我们知道该位置是合法的，我们就确认蚂蚁的新位置上有什么。第一个if语句只是检查新位置上的kGround或kWater。这两个元素都不会引起状态改变，所以，我们只更新蚂蚁的row和col之值，将newRow和newCol的值填入。下次屏幕更新时，蚂蚁就会处在新位置上了。

下面就是设计有限状态机的关键部分了。这个if语句用来检查新位置是否含有食物。这一段很重要，因为有可能在此处转换状态。如果新位置有食物，我们就要更新蚂蚁的位置、去掉食物，并改变蚂蚁的状态。就此例而言，我们要从kForage改成kGoHome。if语句里最后的do-while循环，目的是用另一个随机置放的食物替换消耗掉的食物。如果我们不替换消耗掉的食物，则蚂蚁数量是不会增长的。

Forage()函数的最后一部分是另一种可能的状态转换。最后一个if语句检查新位置是否含有毒物。如果含有毒物，蚂蚁的位置就会被更新，毒物就会被删除，而蚂蚁的状态也会从kForage改成kDead。然后，我们通过do-while循环重新补充消耗掉的毒物。

GoHome 函数

现在我们要谈加进例9-16中ai_Entity类里的第二个新的行为函数，名为GoHome()，它和kGoHome状态有关。如前所述，蚂蚁只要随机找到食物，就会切换成kGoHome状态。蚂蚁会一直保持这种状态，直到成功返回巢穴，或者踩到毒物。例9-18是GoHome()函数的全部内容。

例 9-18：GoHome()函数

```
void ai_Entity::GoHome(void)

{
   int   rowMove;
   int   colMove;
   int   newRow;
   int   newCol;
   int   homeRow;
   int   homeCol;
   int   index;
   int   poisonRow;
   int   poisonCol;

   if (type==kRedAnt)
      {
         homeRow=kRedHomeRow;
         homeCol=kRedHomeCol;
      }
   else
      {
         homeRow=kBlackHomeRow;
         homeCol=kBlackHomeCol;
      }

   if (row<homeRow)
      rowMove=1;
   else if (row>homeRow)
      rowMove=-1;
   else
      rowMove=0;

   if (col<homeCol)
      colMove=1;
   else if (col>homeCol)
      colMove=-1;
   else
      colMove=0;
```

```
    newRow=row+rowMove;
    newCol=col+colMove;

    if (newRow<1) return;
    if (newCol<1) return;
    if (newRow>=kMaxRows-1) return;
    if (newCol>=kMaxCols-1) return;

    if (terrain[newRow][newCol]!=kPoison)
        {
            row=newRow;
            col=newCol;
        }
    else
        {
            row=newRow;
            col=newCol;
            terrain[row][col]=kGround;
            state=kDead;
            do {
                poisonRow=Rnd(2,kMaxRows)-3;
                poisonCol=Rnd(2,kMaxCols)-3;
            } while (terrain[poisonRow][poisonCol]!=kGround);
            terrain[poisonRow][poisonCol]=kPoison;
        }


    if ((newRow==homeRow) && (newCol==homeCol))
        {
            row=newRow;
            col=newCol;
            state=kThirsty;
            for (index=0; index <kMaxEntities; index ++)
                if (entityList[index].type==0)
                    {
                        entityList[index].New(type,
                                              kForage,
                                              homeRow,
                                              homeCol);
                        break;
                    }
        }
}
```

在GoHome()函数中所声明的变量，很类似Forage()函数的变量。然而，在此函数中，我们新增了两个变量，名叫homeRow和homeCol。我们要用这两个变量来确定蚂蚁是否成功抵达其巢穴。当在此世界中新增蚂蚁时，就会用到变量index。剩下的两个变量poisonRow和poisonCol用于替换被消耗掉的任何毒物。

我们起初要确认巢穴所在地。回想一下，我们有两种蚂蚁，即红蚂蚁和黑蚂蚁。每种颜色蚂蚁的巢穴位置都不同。每个巢穴的位置都以全局定义的常量kRedHomeRow、

kRedHomeCol、kBlackHomeRow以及kBlackHomeCol来决定。我们检查每种实体的类型，以确认其为红蚂蚁还是黑蚂蚁。然后，再以全局巢穴位置常量，设定局部变量homeRow和homeCol。现在我们知道巢穴的位置之后，就可以把蚂蚁移往该地了。

你可能会想起，这就是第二章简单追逐算法的另一种变异。如果蚂蚁当前行位置小于巢穴行位置，其行位置偏移量rowMove就设为1。如果蚂蚁的行位置大于巢穴行位置，rowMove就设为-1。如果两者相等，就没有必要改变蚂蚁的行位置，rowMove就设为0。列位置也如法炮制。如果蚂蚁的列位置小于巢穴的列位置，则colMove就设为1。如果前者大于后者，就设为-1。如果col等于homeCol，则colMove就设为0。

一旦知道行和列的偏移量，就能继续计算新的行和列的位置。我们把rowMove加到当前行位置，求得新的行位置；而把colMove加到当前列位置，求得新的列位置。

一旦我们把值赋给newRow和newCol之后，要检查新位置是否在砖块环境区域内。做这件事总是好的，就此例而言，其实是多余的。这个函数总是会把蚂蚁往其巢穴位置推进，而巢穴位置总是在砖块世界的边界内。所以，蚂蚁总是会被限制在该世界的边界内，除非全局巢穴位置常量，被修改成位于世界之外的地方。

if语句的第一部分是检查蚂蚁是否没有踩上毒物。如果新位置不含毒物，则蚂蚁的位置就会被更新。如果if语句的else部分被执行，即可得知，蚂蚁实际上踩到毒物了。因此需要改变状态：蚂蚁的位置要更新、毒物要删除，而蚂蚁的状态则从kGoHome改成kDead。然后，我们用do-while循环替换被消耗掉的毒物。

GoHome()函数中最后的if语句是检查目标是否达成，使用的是我们赋给homeRow和homeCol的值，借此确认新位置是否就是巢穴位置。如果是，则蚂蚁的位置会被更新，而状态会从kGoHome改成kThirsty。这会让蚂蚁在下次UpdateWorld()函数执行时，改用新的行为。if语句的最后部分是用来产生新的蚂蚁。回想一下，只要把食物成功带回巢穴，新的蚂蚁就会产生。我们以for循环检查entityList数组，寻找数组中尚未被用到的第一个元素。如果找到尚未用到的数组元素，我们就在巢穴位置增加新蚂蚁，并将之初始化为成kForage状态。

Thirsty函数

我们加进例9-16中ai_Entity类的下一个行为函数，和kThirsty状态有关。回想一下，蚂蚁把食物成功带回巢穴后，其状态就会切换成kThirsty。在此状态下，蚂蚁会在世界中随机移动以寻找水。然而，和kForage状态不同的是，蚂蚁完成目标后不会回到巢穴，相反的，蚂蚁找到水后，状态会从kThirsty切换到kForage。如前面几种状态那样，踩到毒物时，会自动把状态换成kDead。

我们加进ai_Entity类的第三个新行为函数名叫Thirsty()，观其名可知，当蚂蚁处在kThirsty状态下时，就会被执行。如前所述，蚂蚁成功地把食物带回巢穴后，就会切换到kThirsty状态。蚂蚁会一直保持kThirsty状态，直到找到水，或者踩到毒物。如果蚂蚁找到水，就会回复到最初的kForage状态。如果踩到毒物，就会切换成kDead状态。例9-19是Thirsty()函数的内容。

例9-19：Thirsty()函数

```
void ai_Entity::Thirsty(void)

{
   int   rowMove;
   int   colMove;
   int   newRow;
   int   newCol;
   int   foodRow;
   int   foodCol;
   int   poisonRow;
   int   poisonCol;

   rowMove=Rnd(0,2)-1;
   colMove=Rnd(0,2)-1;

   newRow=row+rowMove;
   newCol=col+colMove;

   if (newRow<1) return;
   if (newCol<1) return;
   if (newRow>=kMaxRows-1) return;
   if (newCol>=kMaxCols-1) return;


   if ((terrain[newRow][newCol]==kGround) ||
       (terrain[newRow][newCol]==kFood))
      {
         row=newRow;
         col=newCol;
      }
   if (terrain[newRow][newCol]==kWater)
      {
         row=newRow;
         col=newCol;
         terrain[row][col]=kGround;
         state=kForage;
         do {
            foodRow=Rnd(2,kMaxRows)-3;
            foodCol=Rnd(2,kMaxCols)-3;
         } while (terrain[foodRow][foodCol]!=kGround);
         terrain[foodRow][foodCol]=kWater;
      }
   if (terrain[newRow][newCol]==kPoison)
      {
```

```
            row=newRow;
            col=newCol;
            terrain[row][col]=kGround;
            state=kDead;
            do {
               poisonRow=Rnd(2,kMaxRows)-3;
               poisonCol=Rnd(2,kMaxCols)-3;
            } while (terrain[poisonRow][poisonCol]!=kGround);
            terrain[poisonRow][poisonCol]=kPoison;
        }

}
```

如例9-19所示，Thirsty()函数的开头和Forage()函数的很类似。我们声明了两个位置偏移量变量rowMove和colMove，以及两个储存蚂蚁新位置的变量newRow和newCol。剩下的变量foodRow、foodCol、poisonRow以及poisonCol，用来替换消耗掉的食物和毒物。

然后，我们计算新的行和列位置的随机偏移量。rowMove和colMove这两个变量含有随机值，介于-1和+1之间。我们把这些随机值加到当前位置以取得新位置。新位置会储存在newRow和newCol之内。if语句的区块会确认新位置是否在该环境的边界内。如果不在，就立刻离开此函数。

这个if语句会检查新位置是空白砖块还是含有食物。换言之，这两种元素都不会引起状态改变。

下一个if语句检查新位置是否含有水。如果确实含有水，则更新蚂蚁的位置、删除水，蚂蚁的状态将返回最初的kForage状态。然后，do-while循环再随机置放新的水。

如上一个行为函数所示，Thirsty()函数中最后一个if语句会检查蚂蚁是否踩上了毒物。如果是，则更新蚂蚁的位置、删除毒物，而蚂蚁的状态则变为kDead。同样的，do-while循环将替换被消耗掉的毒物。

最后结果

以上是和kForage、kThirsty、kGoHome以及kDead这几种状态相关的函数说明。你可以执行此仿真程序，观察不同的行为以及有限状态机蚂蚁，如何在状态之间做转换。

如图9-6所示，即使我们在仿真时只从四只蚂蚁做起，不需多久，蚂蚁马上就能统治这个世界了。事实上，根据特定数量的食物、水和毒物，观察蚂蚁繁殖的速度是很有趣的。

改变例9-20的值来观察其对蚁量的增长或衰减所产生的影响，也是很有趣的。

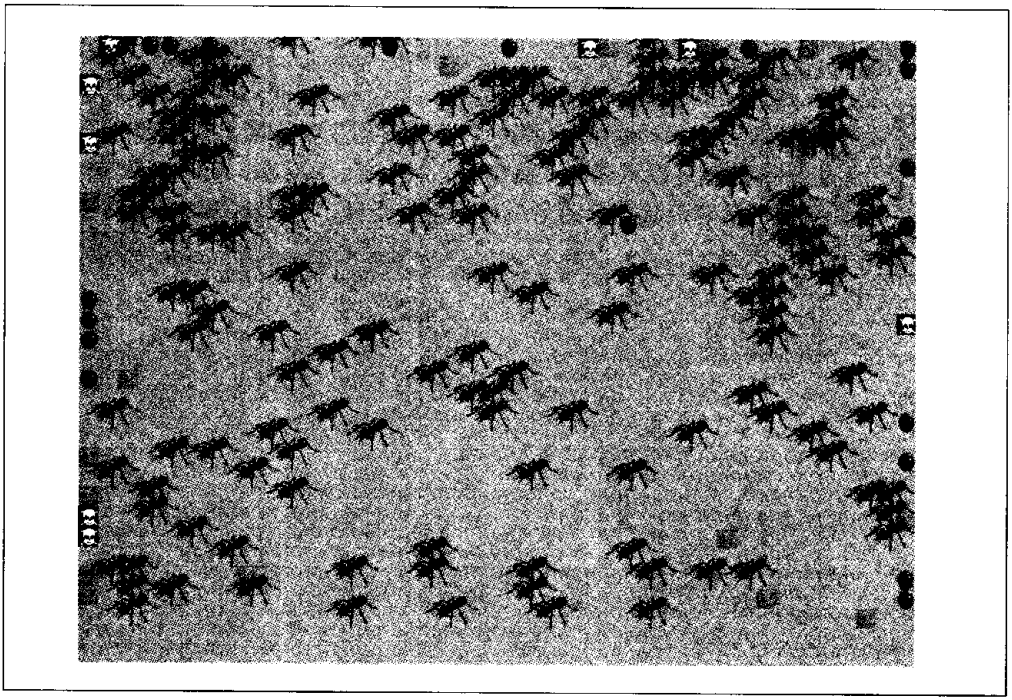


图 9-6：蚂蚁数量暴增

例 9-20：食物、水以及毒物配置

```
#define  kMaxWater      15
#define  kMaxPoison      8
#define  kMaxFood       20
```

如例 9-20 所示，改变仿真世界很简单，只需改变几个全局变量。例如，把毒物数量降得太低，蚁量会急剧增长；如果降低食物供给量，则蚁量成长速度会放慢，但不一定使其数量减少。通过调整这些值，再配上更多的其他可能的状态（从而有更多种行为），你就能让这个仿真世界更复杂而更富有观察的乐趣了。

其他信息

有限状态机在游戏中的应用很普遍。实际上，每个游戏的开发过程中都会在某种程度上用到它，这没什么好惊讶的。再者，因特网上有很多讨论有限状态机的资源。下面列举少数几个讨论有限状态机的因特网资源：

- *http://www.gamasutra.com*
- *http://www.gameai.com*

- *http://www.generation5.org*
- *http://www.aboutai.net*

如果你以“有限状态机”（finite state machine）为关键词在因特网上搜索，一定会发现好几百条信息。此外，你也可以试着以“模糊状态机”（fuzzy state machine）为关键词进行搜寻。模糊状态机是有限状态机的另一种常用的变异，在状态转换时整合了概率。本书第十二章将讨论概率。

第十章

模糊逻辑

1965年，加州柏克莱大学教授Lotfi Zadeh写了第一篇论述模糊集合理论的论文。我们发现要说明何谓模糊逻辑，除了引用模糊逻辑之父的话之外，找不到其他更好的办法。1994年，《Dr. Dobbs Journal》的Jack Woehr对Zadeh做过一次访谈，当Zadeh说道："模糊逻辑的含义是让计算机以一种接近人类行为的方式解决问题。"时，Jack Woehr引述这段话后，Zadeh接着又说道："模糊逻辑的本质是一切都和程度有关。"现在，我们要详细说明模糊逻辑的两项基本原则。

"模糊逻辑的含义是让计算机以一种接近人类行为的方式解决问题。"这句话到底指的是什么？它的意思是人类时常要在极不精确的情况下，分析情况或者解决问题。也许我们没有弄明白全部的事实，也许事实根本就不明确，也许我们只能概括事实，而无法用精确数据或测量值来说明。

例如，假设你和一群朋友在打篮球。当你衡量球场上的对手，以决定由你或其他人防守时，你会根据对方的身高和敏捷度而做出决定。你也许会认定对手又高，动作又快，因此，你最好去防守别人。也许你会说他很高，但动作有点慢，所以由你防守就绰绰有余了。一般来讲，你不会对你自己这样说："这家伙身高6英尺5.5英寸，5.7秒能跑完全场。"

模糊逻辑让你用语言意义提出问题并解决问题，类似于你说话所用的词语。理论上，你可以让计算机利用模糊逻辑告诉你，是否要防守身材高大但行动很慢的对手，等等。虽然这不一定是模糊逻辑的实际应用，但确实说明了关键所在——模糊逻辑让你在使用非常精确的工具（例如计算机）时，能按你平常所做的那样来思考。

第二项原则即"一切都和程度有关。"这同样也可以用篮球对手实例来说明。当你认为对手身高较高、普通或很高时，你心中不需要有固定的界定值，以此做区分或分类。你

可以大胆地判断那家伙较高或非常高，而不用告诉你自己说，如果他身高超过7英尺就算非常高，如果少于7英尺就只能算高。那如果他是6英尺11.5英寸呢？显然，你还是觉得他非常高，只不过如果是7英尺4英寸的话，那种非常高的程度就不一样了。你认为的高和非常高的界限其实相当模糊，而且有些重叠。

传统的布尔逻辑只能让我们定义一个点，把非常高的人和较高的人分出高矮。我们一定得说他非常高，不然就是他不算非常高。利用模糊逻辑，就能避开传统布尔逻辑这种非真即假或者非开即关的特性。模糊逻辑可以有灰色地带或者说“程度”，例如，非常高的程度。

事实上，你可以把模糊逻辑的一切都当成真，只是程度各不相同。如果我们说某事物在模糊逻辑里真的程度是1，这就是绝对真；程度为0的真就是绝对假。所以，对模糊逻辑而言，我们可以让某事物为绝对真、绝对假或介于两者之间的任何情况——也就是程度介于0和1之间。稍后我们会谈及让我们对真值程度进行量化的机制。

模糊逻辑的真值有程度上的变化，还有另一种功能，例如，就控制应用而言，对模糊输入值的响应会很平滑。使用传统布尔逻辑时，碰上某些给定的输入值时，只能用不连贯的方式切换响应状态。为了缓和极不连贯的状态转换，我们必须把输入值分割成许多足够小的范围。而利用模糊逻辑就能避开这个问题，因为响应时，会根据输入条件的真值大小程度或强度，平滑地改变其响应结果。

下面举例说明。标准的家用空调机都配有恒温器，让用户能设置特定的温度。恒温器的设计是这样的：当室内温度高于恒温器的设定值时，空调机就会打开，而当温度到达或者低于恒温器温度设定值时，空调机就会关掉。我们住在路易斯安那州南部，会因为夏天太阳的炙热使得温度上升，因空调机工作而温度冷却，这就造成空调机开开关关。而这样的开关切换对空调机的损害是很大的，时常会对空调机造成损伤。

我们可以预见这种场景，有一个模糊恒温器调控冷却扇，让温度保持理想范围。当温度上升时，冷却扇就加速，而当温度下降时，冷却扇就放慢速度，随时让温度在预定的理想值上下浮动保持平衡。这样一来，空调机就不用来来回回又开又关了。事实上，这样的系统确实存在，它是代表模糊控制的早期应用实例之一。其他应用同样受益于模糊控制，包括火车和地铁控制以及机器人控制，等等，这里不便一一罗列。

模糊逻辑的应用不限于控制系统。你可以把模糊逻辑应用在决策上。典型实例如股票获利分析或管理，也就是利用模糊逻辑制定买卖策略。任何牵涉到主观、不精确、或晦涩信息的决策，几乎都是模糊逻辑的应用对象。

传统逻辑实践家会说，你可以用传统规则方法和逻辑解决这类问题。这也没错，然而，模糊逻辑让我们使用直觉式的语汇，诸如近、远、很远等，借此建构问题、拓展规则并

评估输出结果。通常会让整个系统更具可读性，更易于了解和维护。此外，Timothy Masters在其《Practical Neural Network Recipes in C++》（Morgan Kauffman出版）一书中提到，模糊规则系统通常比传统规则系统要少50%～80%的规则，就能完成同样的任务。模糊逻辑的优点，值得游戏软件AI（时常充斥着if-then形式的规则和布尔逻辑）去探究其奥妙。基于这点动机，我们就能提出一些说明实例，以了解如何在游戏中使用模糊逻辑。

如何在游戏中使用模糊逻辑？

你可以在游戏中以各种方式使用模糊逻辑。例如，你可以用模糊逻辑控制队友，或其他非玩家角色的单位。你也可以用模糊逻辑评估玩家展示的威胁。此外，你也可以用模糊逻辑区分玩家和非玩家角色。上述只是少数几种特例，但说明了模糊逻辑，可以用在很多不同的场景。让我们更深入讨论每种应用实例吧。

控制

模糊逻辑广泛用于真实世界的控制应用上，比如控制火车、空调系统、机器人以及其他方面的应用。视频游戏也为模糊控制的使用提供了许多机会。你可以用模糊控制让游戏的单位（陆地载具、飞行器、徒步之单位等）平滑地通过航点并绕过障碍物。你也可以用模糊控制做出基本的追逐和闪躲，甚至予以改进。

假设一个单位正沿着指定方向前进，但该单位必须朝着某个静止的或移动中的特定目标前进。这个目标可能是航点、代表敌人的单位、某种宝物、基地、或游戏中任何想象得到的实体。我们可以用定性方法解决这个问题，诸如本书讨论过的那些手段。然而，回想一下，有时候我们得自行调整转向力，才能让转弯平滑。如果不调整转向力，那些单位就会突然改变方向，其行动看起来就不自然了。模糊逻辑可以让你实现平滑运动，而不必自行调整转向力。你也可以利用模糊逻辑得到其他方面的改进成果。例如，回想基本追逐的问题，那些单位最后总是沿着某个轴而跟在目标之后。之前我们采用其他方法解决这个问题，诸如视线追逐、拦截或势函数。就此而言，模糊逻辑可以达到类似拦截的效果。基本而言，我们会告诉模糊控制器，目标在左边远处、或在左边、或在前方、或在右边等，然后，令其计算适当的转向力，并施加该力，使其缓慢改变方向，以平滑的方式转向目标。

威胁评估

让我们考虑模糊逻辑在游戏中的另一个可能的应用，这是牵涉到决策，而不是直接的运动控制。

假设在你的战争仿真游戏中，计算机军队时常得配置防卫兵力，以抵抗潜在威胁的敌军。假设计算机军队知道敌军某些特定的信息。为简单起见，我们将此信息限定在敌军与计算机军队的距离，以及敌军的规模。距离可以用附近、逼近、远和很远来表示，而规模可以用零星、少量、中等、大型或巨型来表示。

有了这些信息，我们可以用模糊系统让计算机评估敌军带来的威胁。例如，威胁的程度可以视为无、低等、中等、或高等，确认威胁程度之后，计算机就能依此决定用适当的兵力部署防卫。这种模糊手段可以让我们实现下面的目的：

- 在信息不足的情况下，仿真计算机的行为。
- 让防卫兵力的规模可以平滑变化，难以预测。

分类

假设你想将游戏中的玩家角色和非玩家角色，依照他们的战斗能力予以区分等级。你可以在等级分类上以体力、武器熟练度、被击中的次数、盔甲等级以及其他诸多可供选择的因素作为基础。最后，你会结合这些因素，做出等级分类，诸如弱不禁风、不费力、平庸、顽强、可怕等。例如，玩家如果被击中次数高、盔甲等级普通、体力好、但武器熟练度低，其等级也许就是平庸。模糊逻辑可以让你决定这种等级。此外，你可以用模糊逻辑系统产生数字化的分数，表示等级或评分，再将此分数输入给游戏中其他AI流程使用。

当然，你也可以用其他方法完成这种分类，比如布尔规则、神经网络等。然而，模糊系统可以让你以少数规则按直觉法完成，而无需处理此系统。你还是得事先设立模糊规则，这在每个模糊系统中都是必要的，然而，这个过程只需做一次，而且有模糊逻辑那样的语汇素材可以协助你。稍后我们再回来谈这个议题。

模糊逻辑基础

现在你了解了模糊逻辑是做什么的了，也知道了如何在游戏中使用，下面我们要详谈模糊逻辑的运作方式以及实现方法。如果此时你对模糊逻辑的概念还是有点模糊的话，不用担心，接下来几节我们将详细叙述，那些概念会越来越清晰。

概论

模糊控制或推论过程由三个基本步骤组成。图 10-1 说明了这些步骤。

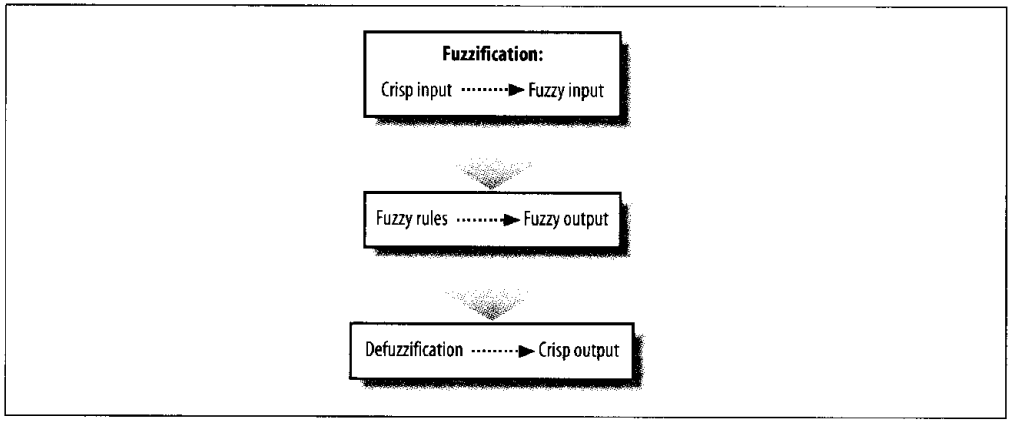


图 10-1：模糊流程概观

这个流程的第一部分叫模糊化（fuzzification）。在此步骤中，有一个对应流程，是把明确数据（crisp data，也就是实际数字），转化成模糊数据。这个对应流程牵涉到，在预定的模糊集合中寻找明确输入数据的归属程度（degree of membership）。例如，假定我们以磅来计某人的体重，我们就能确定此人体重的程度是太瘦、超重或理想。

一旦你把所有输入给此系统的数据，都以模糊集合的归属程度表示之后，就能使用模糊逻辑规则，将这些归属程度结合起来，求出每条规则的真值程度。换言之，你可以在输出之模糊集合（或行动模糊集合）中，找出每条规则的强度或归属程度。例如，以某人的体重和活动能力值作为输入变量时，我们就能定义类似下面的规则：

- 如果超重且无（AND NOT）活力，那么就常做运动。
- 如果超重且有（AND）活力，那么就控制饮食。

就此例而言，这些使用逻辑运算符结合模糊输入变量，所产生的归属程度，或者说是相对应的输出结果（行动）的程度或真值，就是建议采取常做运动或控制饮食的方法。

通常来讲，模糊输出的结果只有“常做运动”是不够的。我们还想把运动程度量化，例如，每周三小时。这个过程是把模糊输出结果的所有元素，换成相对应的明确输出数值，名为反模糊化（defuzzification）。接下来我们要仔细说明每个步骤。

模糊化

模糊系统的输入可以是明确的数字形式。这些是实际数字，用来量化输入变量，例如，某人重 185.3 磅，或者某人身高是 6 英尺 1 英寸。在模糊化的过程中，我们要将这些明确的值对应到质化模糊集合中的归属程度。例如，我们把 185.3 磅对应到微超重，而 6

英尺1英寸则对应到高。你可以利用归属函数（membership function，也叫做特征函数，characteristic function）来做这种对应。

归属函数

归属函数（或译为隶属函数）就是把输入变量对应到模糊集合中某个介于0和1之间的值，求出归属程度。如果在给定集合中的归属程度是1，我们就说该输入数据对集合而言是绝对真。如果归属程度为0，则我们说此输入数据对该集合而言为绝对假。如果归属程度介于0和1之间，则为某种范围的真，也就是某种程度为真。

探讨模糊归属函数之前，我们先谈布尔逻辑的归属函数。图10-2所示的就是这样的函数。

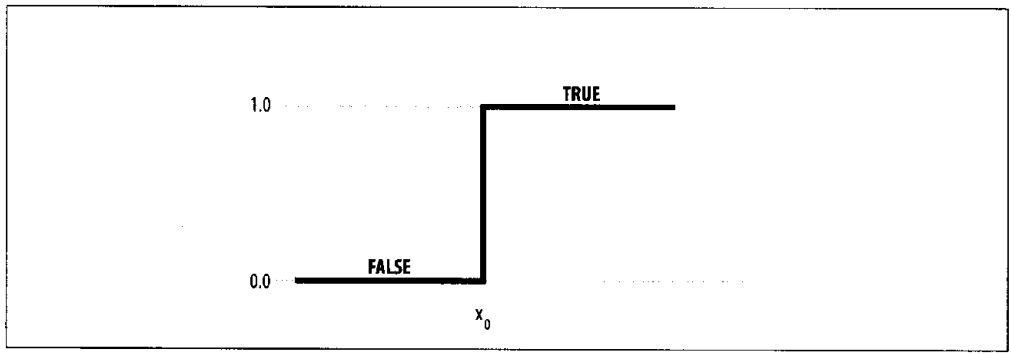


图10-2：布尔逻辑归属函数

由此可知，当输入数据小于x_0时，就对应到假（false），而当输入数据高于x_0时，就对应到真（true）。没有介于两者之间的对应。再回到前面体重的例子，如果x_0等于170磅，则任何人超过170磅就是超重（overweight），而低于170磅的就是不超重（not overweight）。即使某人体重是169.9磅，仍被视为不超重。就此例而言，模糊归属函数能让我们从false到true或者从不超重到超重之间逐渐转移。

实际上，你可以用任何函数作为归属函数，而其形式时常由所需的精确度、考虑问题的性质、经验、是否容易实现以及其他因素来决定。尽管如此，有很多常用的归属函数，已证实有广大的应用范围。我们会在本章讨论一些常用的归属函数。

先思考一下图10-3所示的归属度函数（grade membership function）。

从图10-3中可以看出，0和1之间的逐渐转移。这个函数能用的x值范围就是此函数的定义域（support）。对小于x_0的值，其归属度是0或绝对假，而大于x_1的值，其程度为1或绝对真。介于x_0和x_1之间的值，其归属度呈线性变化。

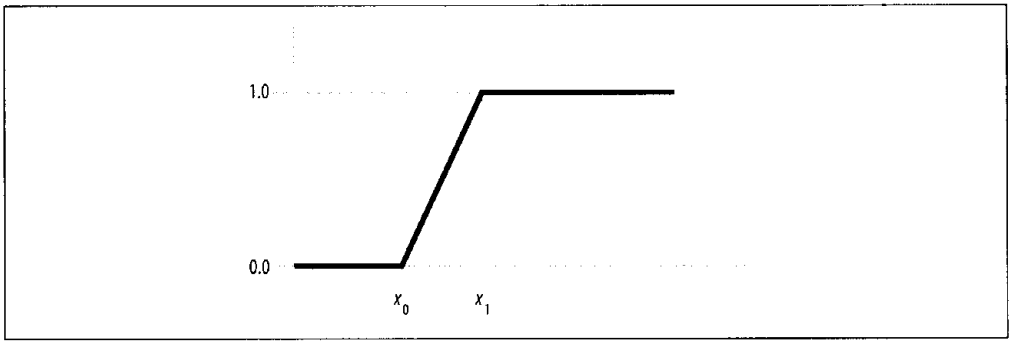


图 10-3：归属度函数

使用直线的点—斜方程式（point-slope equation），可以写出表示此归属度函数的方程式：

$$f(x)=\begin{cases}0;x\le x_0\\ \dfrac{x}{x_1-x_0}-\dfrac{x_0}{x_1-x_0};x_0<x<x_1\\ 1;x\ge x_1\end{cases}$$

回到体重例子，让该函数代表超重归属。令 x_0 等于 175，而 x_1 等于 195。如果某人重 170 磅，则他的体重和超重的归属程度为 0，也就是不超重。如果他重 185 磅，则他的体重和超重的归属程度为 0.5，也就是有点超重。

一般而言，我们感兴趣的是某个输入变量，落在几个质化集合内的某种程度。例如，我们想知道某人的体重程度是超重、太瘦或理想。就此而言，我们要设立一些集合，如图 10-4 所示。

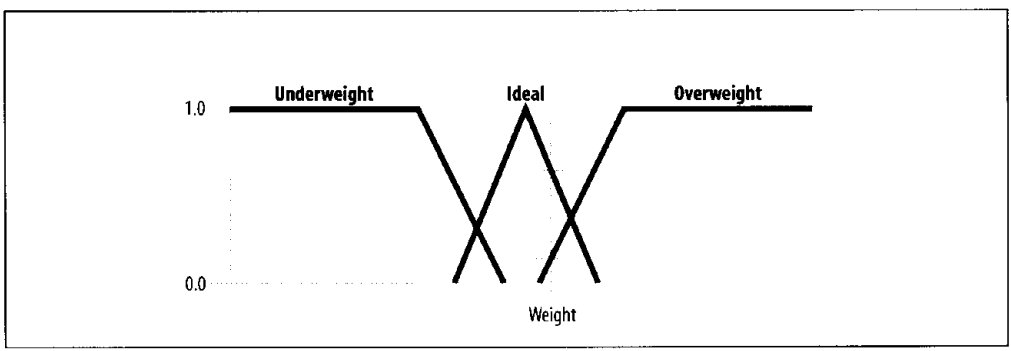


图 10-4：几个模糊集合

有了这样的集合，我们就能在这三个集合里，计算每个输入值的归属程度：太瘦、理想以及超重。就指定的重量而言，我们也许可以找到某人太瘦的归属程度为0，和理想的归属程度为0.75，以及和超重的归属程度为0.15。就此而言，我们可以推论，此人的体重实际上是理想的，也就是75%的归属程度。

直线的点——斜方程式

通过 (x_0, y_0) 和 (x_1, y_1) 这两点的直线方程式为：

$$y - y_1 = m(x - x_1)$$

其中，m 是该直线的斜率，等于：

$$m = \frac{(y_1 - y_0)}{x_1 - x_0}$$

图10-4中的三角形归属函数（triangular membership function）是另一种常用的归属函数形式。参考图10-5，就可以替此三角形归属函数写出方程式：

$$f(x) = \begin{cases} 0; x \le x_0 \\ \dfrac{x}{x_1 - x_0} - \dfrac{x_0}{x_1 - x_0}; x_0 < x < x_1 \\ 1; x = x_1 \\ \dfrac{-x}{x_2 - x_1} + \dfrac{x_2}{x_2 - x_1}; x_1 < x < x_2 \end{cases}$$

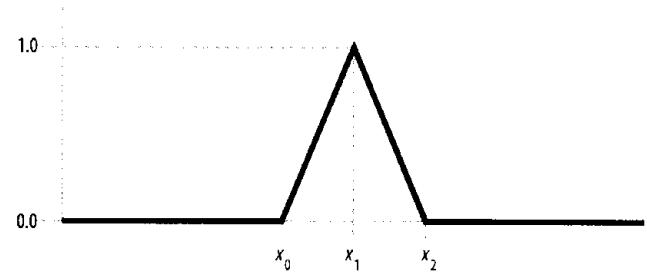


图10-5：三角形归属函数

图 10-4 也显示了“太瘦”集合的反归属度函数（reverse grade membership function）。参见图 10-6，你可以替此反归属度函数写出方程式：

$$f(x)=\begin{cases}1;x\le x_0\\ \dfrac{-x}{x_1-x_0}+\dfrac{x_1}{x_1-x_0};x_0<x<x_1\\ 0;x\ge x_1\end{cases}$$

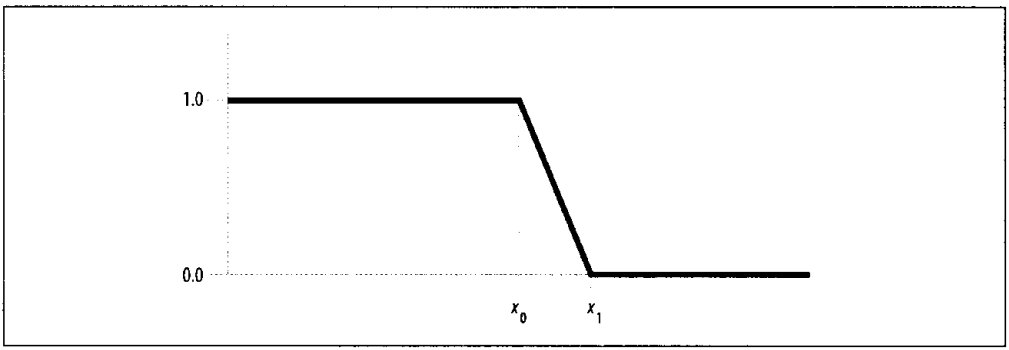


图 10-6：反归属度函数

另一种常用的是梯形归属函数（trapezoid membership function），如图 10-7 所示。

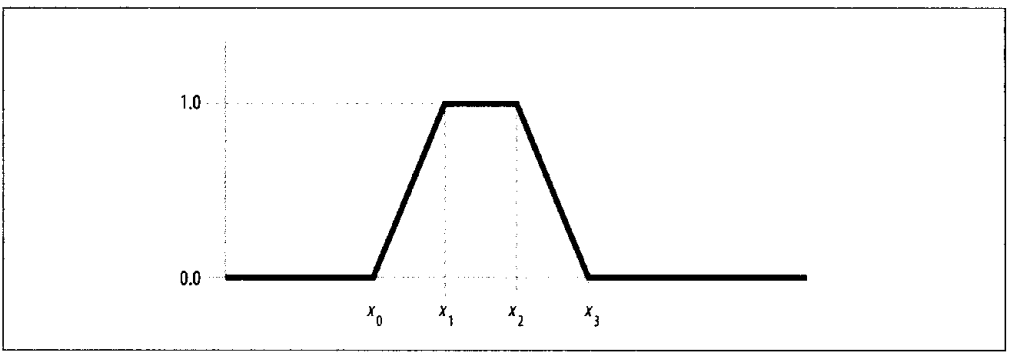


图 10-7：梯形归属函数

梯形函数的方程式如下：

$$f(x)=\begin{cases}0;x\le x_0\\ \dfrac{x}{x_1-x_0}-\dfrac{x_0}{x_1-x_0};x_0<x<x_1\\ 1;x_1\le x\le x_2\\ \dfrac{-x}{x_3-x_2}+\dfrac{x_3}{x_3-x_2};x_2<x<x_3\end{cases}$$

如图 10-4 那样，替给定的输入变量设定好几个模糊集合，大部分都是判断和试误的结果。调整集合所用的归属函数，以达到理想的或最佳的结果并非不常见。调整的时候，你可以替每个模糊集合尝试不同的函数形式，也可以尝试，多使用一些模糊集合或少用几个模糊集合。有些模糊逻辑实践家建议用七个模糊集合，全面定义任何输入变量的实际运作范围。图 10-8 就是一种这样的组合。

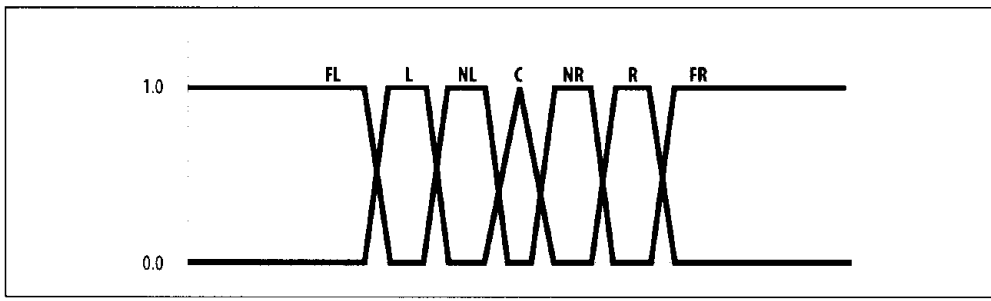


图 10-8：七个模糊集合

图 10-8 所示的七个模糊集合是中央（C）、靠右（NR）、右（R）、最右边（FR）、靠左（NL）、左（L）以及最左边（FL）。这些分类可以是任何形式，主要是取决于你的问题是什么。例如，角色扮演游戏中，要表示玩家和非玩家角色间的结盟关系，你的模糊集合也许就是：中立、稍微友好、友好、不太友好、稍微敌意、敌意及非常敌意。

注意，图 10-4 和图 10-8 的每个集合都和相邻集合彼此重叠。这对平滑转移是很重要的。一般而言，可行的原则是每个集合都应该和其相邻集合彼此重叠 25%。

到目前为止，我们讨论的归属函数都是最常用的，然而，要求较高精确度或者有非线性需求时，有时候也会用到其他函数。例如，有些程序会用到高斯曲线（Gaussian curves），有的则会用到 S 形曲线。图 10-9 做出了说明。就多数程序和游戏而言，像我们讨论过的这些线性函数就已经够用了。

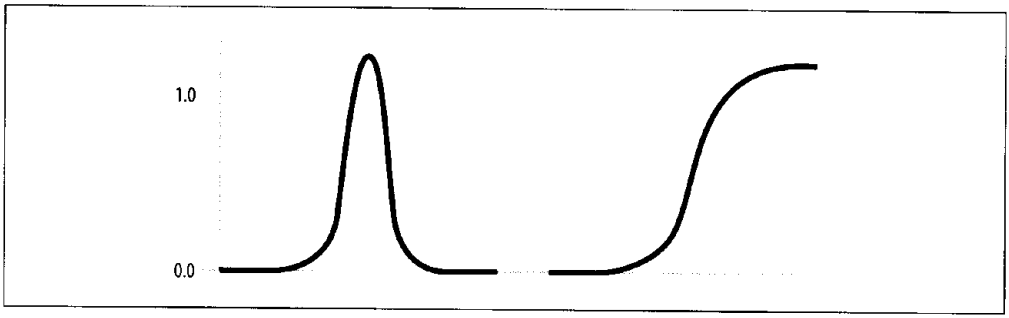


图 10-9：其他归属函数的范例

例 10-1 是我们讨论过的各个归属函数的程序表达。

例 10-1：模糊归属函数

```
double      FuzzyGrade(double value, double x0, double x1)
{
      double result = 0;
      double x = value;

      if(x <= x0)
            result = 0;
      else if(x >= x1)
            result = 1;
      else
            result = (x/(x1-x0))-(x0/(x1-x0));

      return result;
}

double  FuzzyReverseGrade(double value, double x0, double x1)
{
      double result = 0;
      double x = value;
      if(x <= x0)
            result = 1;
      else if(x >= x1)
            result = 0;
      else
            result = (-x/(x1-x0))+(x1/(x1-x0));

      return result;
}

double      FuzzyTriangle(double value, double x0,
                          double x1, double x2)
{
      double result = 0;
      double x = value;
```

```
    if(x <= x0)
        result = 0;
    else if(x == x1)
        result = 1;
    else if((x>x0) && (x<x1))
        result = (x/(x1-x0))-(x0/(x1-x0));
    else
        result = (-x/(x2-x1))+(x2/(x2-x1));

    return result;
}

double    FuzzyTrapezoid(double value, double x0, double x1,
                         double x2, double x3)
{
    double result = 0;
    double x = value;

    if(x <= x0)
        result = 0;
    else if((x>=x1) && (x<=x2))
        result = 1;
    else if((x>x0) && (x<x1))
        result = (x/(x1-x0))-(x0/(x1-x0));
    else
        result = (-x/(x3-x2))+(x3/(x3-x2));

    return result;
}
```

想求出给定的输入值在特定集合内的归属程度，只要调用例10-1的函数之一，并传递和该函数形状定义相关的数值和参数。例如，`FuzzyTrapezoid(value, x0, x1, x2, x3)`可以求出`value`在`x0`、`x1`、`x2`、`x3`所定义的函数形状（如图10-7所示）的集合里的归属程度。

藩篱函数

藩篱函数（hedge function）有时候可用于修改归属函数所返回的归属程度。藩篱函数主要是提供其他语汇素材，让你能在其他逻辑运算中结合使用。两个常用的藩篱函数是VERY()和NOT_VERY()，其定义如下：

$$\text{VERY}(\text{Truth}(A)) = \text{Truth}(A)^2$$
$$\text{NOT\_VERY}(\text{Truth}(A)) = \text{Truth}(A)^{0.5}$$

这里的*Truth(A)*就是A在某模糊集合里的归属程度。藩篱函数可以有效地改变归属函数的形状。例如，把藩篱函数用到线性归属函数上时，就会让这些归属函数的线性部分变成非线性。

藩篱函数在模糊系统中不是必要的。你可以建构归属函数以满足你的需求，而无需额外使用藩篱函数。我们在此提及它只是为了完整起见，因为藩篱函数时常在模糊逻辑的领域里出现。

模糊规则

把特定问题的所有输入变量都模糊化之后，接着要做的就是建构一组规则，以某种逻辑方式结合输入数据，来生成某些输出结果。在 if-then 形式的规则中，诸如 if A then B 这种规则，“if A”部分叫做前件（antecedent）或前提，而“then B”这部分叫做后件（consequent）或结论。我们要以逻辑方式结合模糊输入变量以构成前提，再由此产生模糊结论。结论就是指某些预定输出模糊集合中的归属程度。

模糊公理

如果我们打算以模糊输入数据写出逻辑规则，就需要某种方式能把平常的逻辑运算符用到模糊输入数据上，差不多就像我们对布尔输入数据的做法那样。明确地讲，我们要能够处理交集（AND）、联集（OR）以及补集（NOT）。对模糊变量而言，通常这些逻辑运算符的定义如下所示：

联集 (Disjunction)

$$Truth\ (A\ OR\ B) = MAX(Truth(A), Truth(B))$$

交集 (Conjunction)

$$Truth\ (A\ AND\ B) = MIN(Truth(A), Truth(B))$$

补集 (Negation)

$$Truth\ (NOT\ A) = 1 - Truth(A)$$

同样的，这里的 Truth(A)就是 A 在某模糊集合里的归属程度，其值介于 0 和 1 之间。Truth(B)也是如此。由此可见，OR 逻辑运算符被定义为操作数中的最大值，而 AND 逻辑运算符被定义成操作数中的最小值，至于 NOT 则是 1 减去所得之归属程度。

让我们考虑一个实例。假定某人对超重的归属程度是 0.7，对高的归属程度是 0.3，则前述定义的逻辑运算符的结果会如下所示：

$$超重\ AND\ 高 = MIN(0.7, 0.3) = 0.3$$

$$超重\ OR\ 高 = MAX(0.7, 0.3) = 0.7$$

$$NOT\ 超重 = 1 - 0.7 = 0.3$$

NOT 高 = 1 - 0.3 = 0.7

NOT(超重 AND 高) = 1 - MIN(0.7, 0.3) = 1 - 0.3 = 0.7

写成程序的话，这些逻辑运算相当琐细而简单，如例 10-2 所示。

例 10-2：模糊逻辑运算符函数

```
double FuzzyAND(double A, double B) {
      return MIN(A, B);
}

double FuzzyOR(double A, double B) {
      return MAX(A, B);
}

double FuzzyNOT(double A) {
      return 1.0 - A;
}
```

这些不是 AND、OR 和 NOT 唯一的定义。在某些特定应用情况下，会用其他的定义方式，例如，你可以把 AND 定义成两个归属程度的积，把 OR 定义成或然 -OR，如下所示：

或然 -OR = Truth(A) + Truth(B) - Truth(A)Truth(B)

有些其他的模糊逻辑工具，甚至可以让你自行定义逻辑运算符，使可能性无穷无尽。然而，就多数应用情况而言，这里提到的定义就已经够用了。

规则的评估运算

在传统布尔逻辑程序中，像“if A AND B then C”这种规则在运算之后，会得到绝对真或假，也就是 1 或 0。读过上一节之后，你知道了用模糊逻辑运算时显然不是这么回事。A AND B 在模糊逻辑中可以运算成 0 和 1 之间的任何数字，包括 0 和 1 在内。这样的事实，让模糊规则和布尔逻辑规则，在运算时有本质的不同。

在传统布尔系统中，每条规则会逐一运算，直到有条规则为真为止，然后，就开始运行，也就是说，执行其结论。在模糊规则系统中，所有的规则都会同时进行运算。每条规则都会运行，然而，运行的程度或强度各不相同。每条规则的前提的逻辑运算结果，会产生该规则结论的强度。换言之，每条规则的强度代表的是输出的模糊集合中的归属程度。

假设你有个视频游戏，使用模糊系统运算某生物是否应该攻击玩家。输入变量是距离、生物的健康状况以及对手等级。每个变量的归属函数看起来也许就像图 10-10 所示的那样。

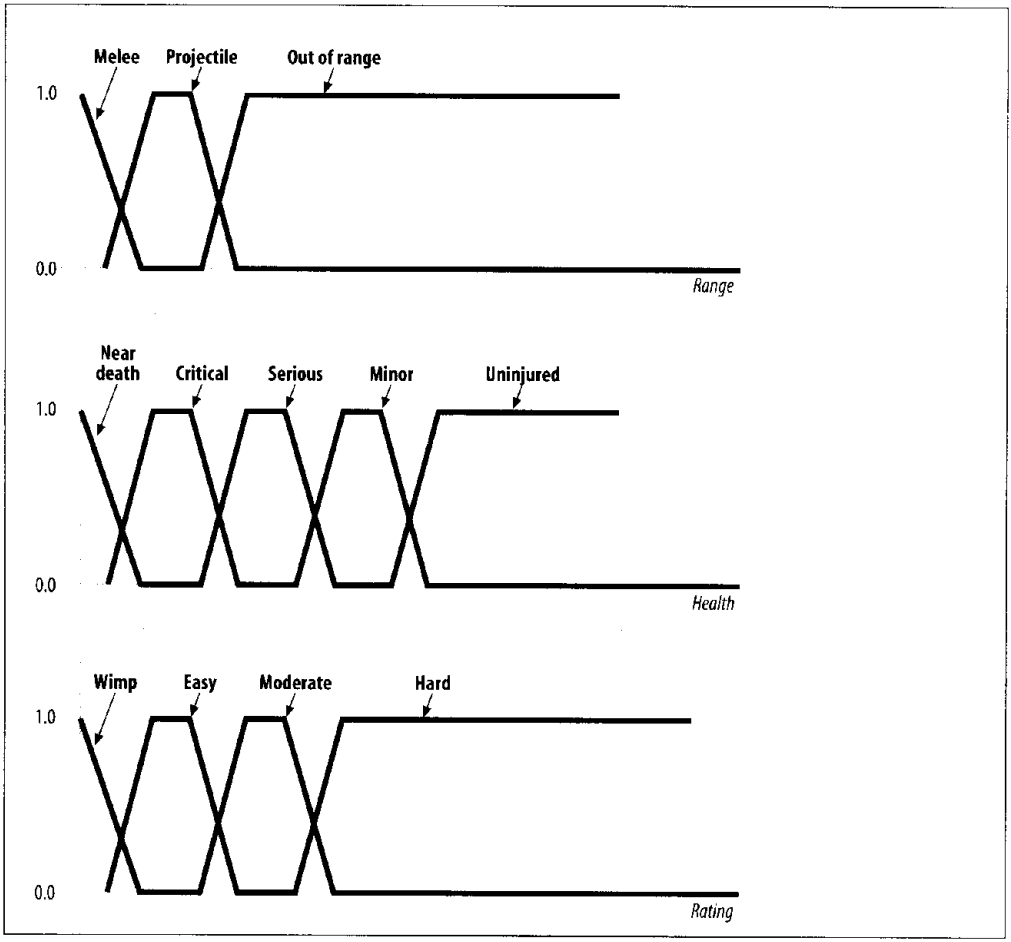


图 10-10：输入双数的归属函数

此例中的输出行动可以是逃跑、攻击或什么也不做。我们可以写出一些类似下列语句的规则：

if（位于肉搏战距离内 AND 健康）AND NOT 坚强 then 攻击

if (NOT 位于肉搏战距离内）AND 健康 then 什么也不做

if (NOT 位于距离外 AND NOT 健康）AND (NOT 懦弱）then 逃跑

你可以设定其他规则来处理其他可能性。在你的游戏中，所有的规则都会运算，并获得每个输出行动的归属程度。每个输入变量都有特定程度之后，你可能会得到如下所示的输出结果：

攻击的归属程度为 0.2

什么也不做的归属程度为 0.4

逃跑的归属程度为 0.7

在程序中，这些规则的评估运算就像例 10-3 那样，你可以从中发现和传统的 if-then 形式的规则，有显著的差异。

例 10-3：模糊规则

```
degreeAttack = MIN(MIN (degreeMelee, degreeUninjured),
                                1.0   degreeHard);

degreeDoNothing = MIN ( (1.0 - degreeMelee),
                                degreeUninjured);

degreeFlee = MIN (MIN ((1.0 - degreeOutOfRange),
                       (1.0 - degreeUninjured)),
                       (1.0 - degreeWimp));
```

输出的程度代表的是每条规则的强度。解读这些输出结果，最简单的做法就是以最高程度的行动为行动依据。就此例而言，最终行动是逃跑。

在某些情况下，你要做的不只是执行输出结果中最高程度的行动而已。例如，本章前面讨论威胁评估时，你也许想用模糊输出结果，求出要部署的防卫兵力的精确数量。要取得精确的数字作为输出结果时，你得把模糊规则所得到的结果反模糊化才行。

反模糊化

当你想用精确数值作为模糊系统的输出数据时，就需要反模糊化（defuzzification）的过程了。如前所述，每条规则都会得到某个输出模糊集合中的归属程度。就前例而言，假设此时不只要求算出某种限定的行动（什么也不做、逃离或攻击），你还想用输出结果，求出该生物采取行动时速率应该多大。例如，如果输出的行动是逃离，该生物是走着逃离，还是跑着逃离，而离开的速度有多快？要取得精确数值时，我们必须把输出强度聚集起来，也就是说需要定义输出归属函数。

例如，我们也许需要如图 10-11 所示的输出归属函数。

利用前面讨论过的数值输出（攻击归属程度 0.2，什么也不做归属程度 0.4，而逃离归属程度 0.7），最后我们可以得到合成的归属函数，如图 10-12 所示。

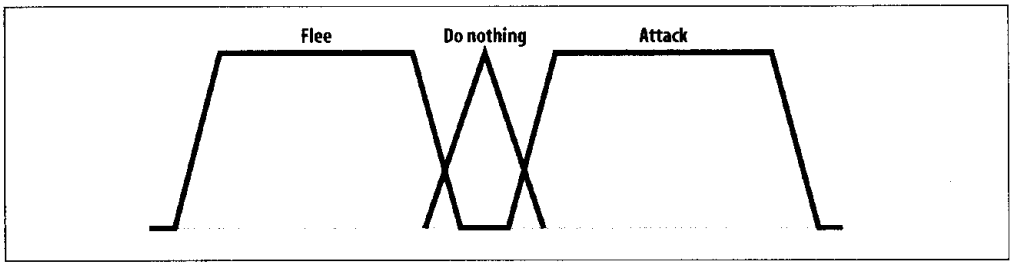


图 10-11：输出的模糊集合

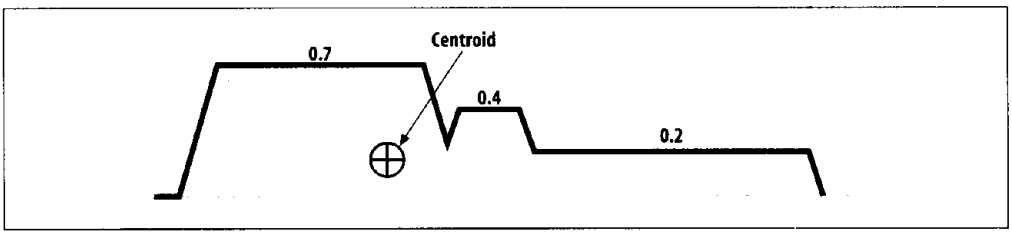


图 10-12：输出的归属函数

为了得到合成的归属函数，每个输出集合都按其规则强度，所得到的输出归属程度而裁减出来。然后，所有输出集合再利用联集结合起来。

此时，我们只有输出归属函数，还没有一丁点儿精确数字。我们可以利用许多方法，从这样的输出模糊集合中得到精确数字。最常用的方法之一是，寻找输出模糊集合所占面积的几何中心，并以该中心的水平轴坐标值，作为精确输出值。这样就可以获得所有规则之间的妥协值，也就是说，单一输出数字，其实是所有输出归属程度的加权平均值。

要找出这种输出函数的中心，你得利用数值积分手段算出曲线围成的面积。或者可以想象成多边形，再用计算机以几何方法找出中心。也可以用其他手段。无论如何，找面积的中心是很耗费运算能力的，尤其是在游戏中这种运算次数会很多。所幸，有较为简单的方法可以利用，也就是所谓的单值输出归属函数（singleton output membership function）。

单值输出归属函数，其实就像栏杆上的铁钉那样，不是一条曲线，本质上是事先就反模糊化好的输出函数。例如，我们可以给每个输出行动指派速率，比如逃离是-10，什么也不做是1，而攻击是10。然后，比如说，逃离行动最后所得的速率是默认值-10，乘以逃离输出行动为真的程度。就我们的例子而言，就是-10乘以0.7，也就是-7为逃离速率。（这里的负号只是表示逃离和攻击相反而已。）此时，计算所有输出值聚合起来的结果，只需求简单的加权平均值，而不用再求中心了。

一般而言，假设 μ 是某输出集合为真的程度，而且 x 为和此输出集合相关的精确单值，则最后聚合而反模糊化的输出结果是：

$$output = \frac{\sum_{i=1}^{n} \mu_i x_i}{\sum_{i=1}^{n} \mu_i}$$

就我们的例子而言，所得结果如下：

$$\text{输出值} = [(0.7)(-10) + (0.4)(1)+(0.3)(10)] / (0.7+0.4+0.3) = -2.5$$

这样的输出结果用来控制生物行动时，会使得该生物逃离，但不见得会让你看出端倪。

为了强化目前为止我们所讨论到的所有观念，下面我们再讲几个范例的细节。

控制实例

本章开始提到了控制实例，我们想用模糊控制，把计算机控制的单位引向某些目标。目标可以是航点、敌军等。为了做到这一点，我们要设定好几个模糊集合，描述计算机控制单位及其目标之间的相对方位。相对方位就是计算机控制单位的速度向量，以及连接计算机控制单位的位置与目标的向量之间的角度。利用前面实例中提到的技术（也就是追逐和闪躲实例），你可以求出此相对方位角度，其值为标量角度。现在，我们的目标是利用此相对方位，作为模糊控制系统的输入数据，借此求出转向力的适当值，将该值施加到计算机控制单位上，使之朝目标前进。这是一个非常简单的例子，因为只有一个输入变量，因此，只需定义一组模糊归属函数。就此例而言，我们要设的归属函数和模糊集合如图 10-13 所示。

此例中，我们设了五个模糊集合。从左到右，每个代表的是定性的相对方位，也就是最左边、左边、前方、右边以及最右边。最左边和最右边归属函数是归属度函数，而左边和右边函数是梯形函数。前方函数（ahead）是三角形函数。有了任何相对方位角度，你就可以用例10-1所示的C函数，计算出每个模糊集合的归属程度。假设在游戏中的某一时刻，相对方位角度是 +33 度。现在，我们必须计算此相对方位，落在每个模糊集合中的程度。显然，除了右边和最右边这两个集合外，其他集合的程度都是 0。然而，我们还是要做下去，写出程序代码，算出所有归属程度。例 10-4 就是该程序代码。

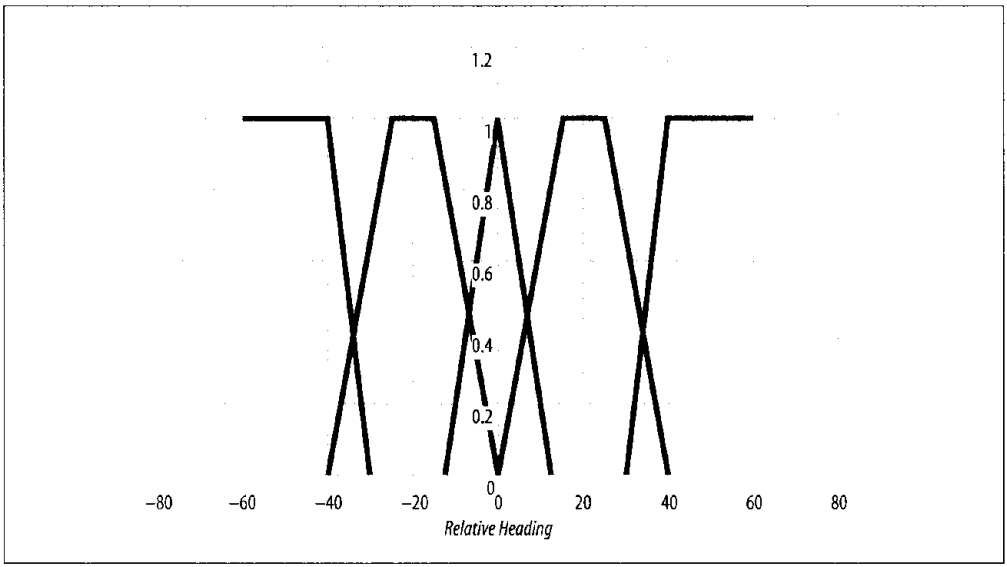


图 10-13：相对方位模糊集合

例 10-4：相对方位归属程度计算

```
mFarLeft      =   FuzzyReverseGrade(33, -40, -30);
mLeft         =   FuzzyTrapezoid(33, -40, -25, -15, 0);
mAhead        =   FuzzyTriangle(33, -10, 10);
mRight        =   FuzzyTrapezoid(33, 0, 15, 25, 40);
mFarRight     =   FuzzyGrade(33, 30, 40);
```

此例中，变量mFarLeft、mLeft等，储存的是每个预定模糊逻辑中，33度这个相对方位值的归属程度。所得结果整理如表10-1所示。

表 10-1：归属程度计算结果

模糊集合	归属程度最左边
0.0左边	0.0
前方	0.0
右边	0.47
最右边	0.3

现在，要利用这些归属程度的结果控制我们的单位时，只要以每个程度值作为转向力计算中的系数即可。假设最大向左转向力是某个常数FL，而最大向右转向力是另一个常数FR。我们令FL=-100磅，而FR=100磅。现在，就可以计算要施加的总转向力了，如例10-5所示。

例 10-5：转向力计算

```
合力 = mFarLeft * FL + mLeft * FL + mRight * FR + mFarRight * FR;
```

计算所得结果是77磅的转向力。注意，计算中没有引入mAhead。这表示前方的归属程度不需要任何转向力。从技巧上而言，我们可以不理会前方归属函数，然而，我们把它提出来是为了说明重点。

在仿真实境中，诸如本书前面所谈的那些范例，游戏循环把相对方位算出后的下一轮，就会将此转向力施加到该单位上。此转向力的作用会在游戏循环的下一轮产生，从而改变该单位的方位，然后，新的相对方位又会被计算出来。新的相对方位会以这里所讨论的同样的方式进行处理，再算出新的转向力并予以施加。最后，总转向力会逐渐减少至0，因为相对方位会变为0。就模糊的观点来看，就是前方集合的归属程度会变为1。

威胁评估实例

就本章开头提到的威胁评估范例而言，我们想再处理两个输入变量，也就是敌军所在地以及敌军规模，借此求出敌军引发的威胁等级。最后，我们想求出适当的防卫兵力数量，并加以部署以抵挡敌军之威胁。此例中我们必须设定几条模糊规则，把输出结果反模糊化，以取得精确数字，也就是要部署的防卫兵力数。然而首要任务，是替这两个输入变量定义模糊集合。图 10-14 以及图 10-15 就是此例所用的模糊集合。

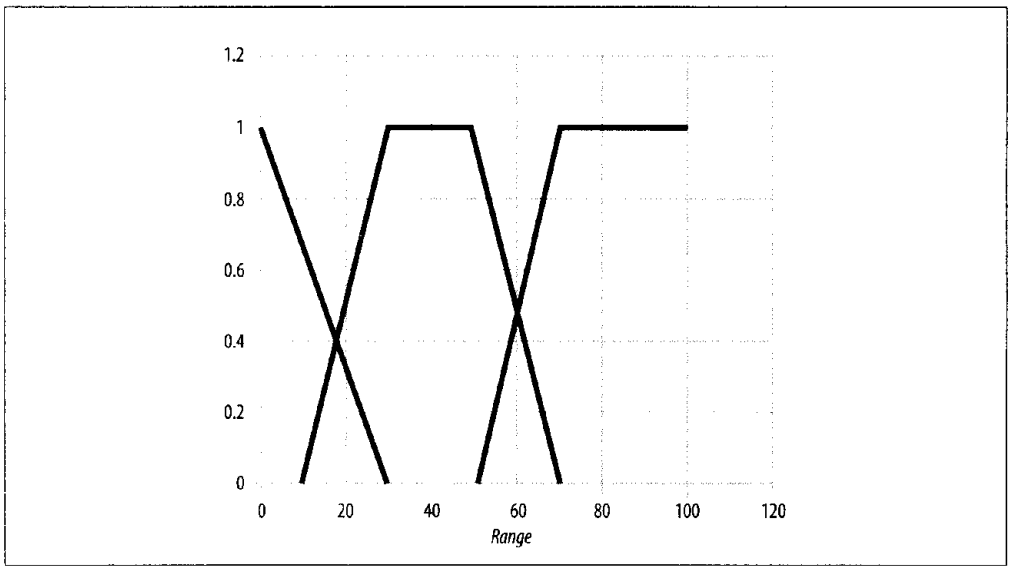


图 10-14：距离模糊集合

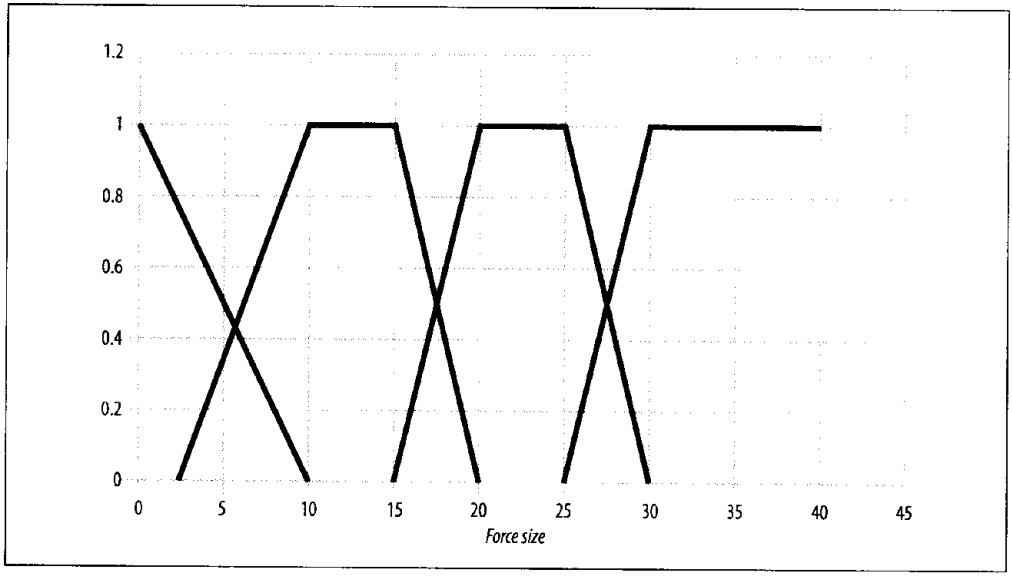


图 10-15：兵力模糊集合

参见图 10-14，从左往右看，这三个归属函数代表的是逼近、附近以及远处。其距离可以用游戏中合适的单位来指定。假设距离是以十六进制表示。

参见图 10-15，从左往右看，这些归属函数代表的模糊集合是零星、少量、中等以及大型。有了这些模糊集合，我们可以准备做计算了。

假设游戏循环在运行某一轮时，这个模糊系统被调用，借此评估位于 25 个单位距离之外，八个敌军单位所带来的威胁。所以，我们现在必须把精确的输入值模糊化，以求出这些变量在每个预定的模糊集合中的归属程度。例 10-6 是该步骤的程序代码。

例 10-6：距离和兵力规模变量的模糊化

```
mClose      = FuzzyTriangle(25, -30, 0, 30);
mMedium     = FuzzyTrapezoid(25, 10, 30, 50, 70);
mFar        = FuzzyGrade(25, 50, 70);

mTiny       = FuzzyTriangle(8, -10, 0, 10);
mSmall      = FuzzyTrapezoid(8, 2.5, 10, 15, 20);
mModerate   = FuzzyTrapezoid(8, 15, 20, 25, 30);
mLarge      = FuzzyGrade(8, 25, 30);
```

此例的结果如表 10-2 所示。

表 10-2：模糊化结果摘要

模糊集合	归属程度
逼近	0.17
附近	0.75
远处	0.0
零星	0.2
少量	0.73
中等	0.0
大型	0.0

考虑任何规则之前，我们先说明输出行动。就此例而言，我们想让模糊输出，指出接近中敌军的威胁等级。我们要在此例中使用单值输出函数，而以输出模糊集合（或行动模糊集合）低等、中等以及高度作为威胁等级。每个集合的单值输出值是以 10 单位、30 单位及 50 单位，作为集合的低等、中等以及高度部署兵力。

现在，我们可以确定规则了。在此例中，要显示规则的最简单方式就是利用表格，如表 10-3 所示。

表 10-3：规则矩阵表

	逼近	附近	远
零星	中等	低等	低等
少量	高度	低等	低等
中等	高度	中等	低等
大型	高度	高度	中等

第一行代表距离模糊集合，而第一列代表兵力模糊集合。表中其他方格代表的是距离和兵力两两交集后的威胁等级。例如，如果兵力是零星，而距离是逼近，则威胁等级是中等。

此例中，我们可以用很多方式设立规则并予以处理。观察表 10-3 之后，显然，我们可以利用 AND 和 OR 运算符的各种组合，并结合输入变量，替每个输出集合建立一条规则。然而，这会造成庞大的难以处理的程序代码，夹杂许多嵌套的逻辑运算。另一种极端情况是，我们可以替输入变量的每一个组合都准备一条规则并予以运算，再替每个输出集合找出最高程度，然而，这会让规则数目增多。不过，这是最简单的可读方式，所以，我们把其他的做法排除了。为了让事情再简化一些，我们只显示输入集合和归属程度非零的组合，而且我们至少做一次嵌套运算，如例 10-7 所示。

例 10-7：嵌套及非嵌套模糊规则

```
.
.
.
mLow = FuzzyOr(FuzzyAND(mMedium, mTiny), FuzzyAND(mMedium, mSmall));
mMedium = FuzzyAND(mClose, mTiny);
mHigh = FuzzyAND(mClose, mSmall);
.
.
.
```

就我们的例子而言，这些规则的运算结果是 mLow = 0.73、mMedium = 0.17 以及 mHigh = 0.17。这些是各自输出模糊集合中的归属程度。现在，我们要利用前面定义的单值输出归属函数来反模糊化这些结果，以取得代表要部署防卫兵力的单值。这种计算只需加权平均而已，如前所述。例 10-8 是此例的程序代码。

例 10-8：反模糊化

```
nDeploy = ( mLow * 10 + mMedium * 30 + mHigh * 50) /
            (mLow + mMedium + mHigh);
```

要部署的兵力的数量 nDeploy 是 19.5 单位，如果取整数，就是 20。由于敌军数量少，而且距离很近，这似乎相当合理。当然，这些结果都是调整出来的。例如，你可以通过修改我们此例中所用的单值，就能轻易调整结果。此外，各种不同输入归属函数的形状也是调整的理想对象，你可以替每个模糊集合尝试不同形状的函数，多用一些模糊集合或少用几个。一旦全部都调整好了，你将发现，无论输入值怎么变化，从一组输入变量到另一组，响应结果总是平滑地变化。此外，由于部署兵力的数目急速变化，或断点没有明确地定义，将使玩家难以预测，让游戏过程更有乐趣。

第十一章

规则式 AI

本章我们要研讨基于规则的AI系统。基于规则的AI系统可能是真实世界和游戏软件AI最广为使用的AI系统了。规则系统最简单的形式由一连串if-then规则组成，用来做推论或行动决策。从技巧上来说，我们已经在第九章有限状态机中，看过一种规则系统的形式；我们用规则处理状态的转换问题。我们在第十章谈模糊逻辑时，也看过另一种规则系统。

本章要专门谈所谓的专家系统（expert system）中最常用的规则系统。真实世界中以规则构成的专家系统实例，包括医疗诊断、防诈骗系统以及工程错误分析。规则系统的优点之一是，根据一组已知事实以及他们对某特定问题领域的知识，以人类惯用的思考和推理方式进行模拟。这种规则系统的另一个优点是相当容易编写程序和管理，因为编写在规则中的知识是模块化的，而且规则可以用任何次序编写程序。在编写系统程序以及日后修改系统时，都十分的灵活。希望你读过本章后，对这些优点的理解会更加清楚。不过，在具体谈论细节之前，我们先讨论一些可以使用规则系统的游戏范例。

想象一下，你正在制作实时策略模拟游戏，而且牵涉到常用的科技树（technology tree），借此，玩家必须训练农民，建立设施以及收割农作物。图11-1说明了科技树的构成。

我们的目标是让计算机对手，追踪玩家当前的科技状态，使得计算机对手可以据此规划并部署攻防资源。现在，你可以作弊，让计算机对手清楚地知道玩家的科技状态。虽然，让计算机获得此信息，据此推论玩家的科技状态，但是玩家以同样的方式评估计算机对手的科技状态，因此这种做法比较公平，也比较真实。玩家和计算机都必须派出侦察兵，收集信息，根据所收集到的信息做推论。我们可以利用相当简单的规则系统达到这种效果，本章会予以讨论。

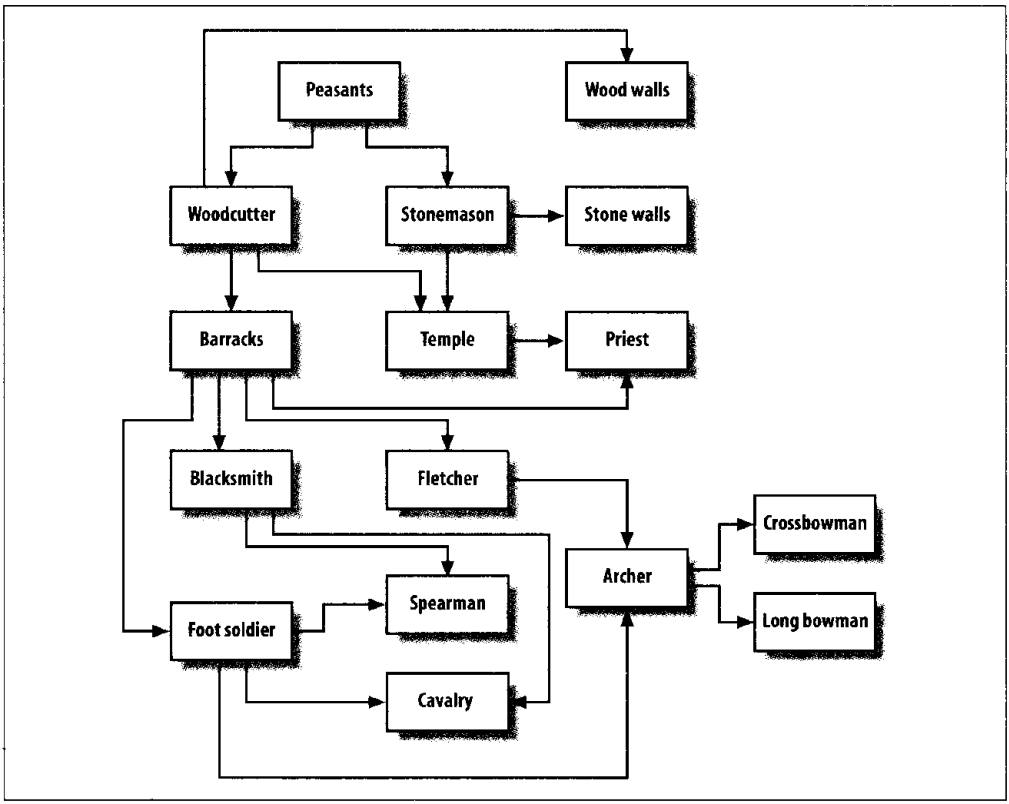


图 11-1：科技树范例

让我们考虑另一个实例。假设你在制作武术对战游戏，你想让计算机有能力预期玩家的下一招，以此做出适当的应对招式，比如反击、躲闪或避开。这里的要点是用某种方式记录玩家的招式（拳脚动作），根据先前出过的招式，推论接下来最有可能出现的是哪一招。例如，对战时，玩家先挥出一拳，再两拳合出，那么玩家接下来最有可能出的招式是再一拳，还是下踢，或上踢？我们可以利用规则系统做出这种预期。本章后面再来谈此例的细节。

规则系统基础

规则系统有两个主要部分：工作记忆（working memory）和规则记忆（rule memory）。工作记忆储存已知事实，以及由规则所做的断言，而规则记忆（简称规则）则含有if-then形式的规则，能够以储存在工作记忆里的事实运行。当规则被触发，或者以规则系统的语汇来说的话，就是规则被启动（fire）了，则这些规则就能触发某些行动，或引起状态

的改变，就像有限状态机那样；或者，这些规则也能修改工作记忆的内容，也就是新增的所谓断言（assertion）的信息。

例 11-1 是实时策略游戏科技树的工作记忆内容。

例 11-1：工作记忆示例

```
enum TMemoryValue{Yes, No, Maybe, Unknown};

TMemoryValue    Peasants;    // 农民
TMemoryValue    Woodcutter;  // 伐木工
TMemoryValue    Stonemason;  // 石匠
TMemoryValue    Blacksmith;  // 铁匠
TMemoryValue    Barracks;    // 兵营
TMemoryValue    Fletcher;    // 箭工
TMemoryValue    WoodWalls;   // 木栅栏
TMemoryValue    StoneWalls;  // 石墙
TMemoryValue    Cavalry;     // 骑兵
TMemoryValue    FootSoldier; // 步兵
TMemoryValue    Spearman;    // 矛兵
TMemoryValue    Archer;      // 弓箭手
TMemoryValue    Temple;      // 庙宇
TMemoryValue    Priest;      // 僧侣
TMemoryValue    Crossbowman; // 十字弓箭手
TMemoryValue    Longbowman;  // 长弓箭手
```

就此例而言，我们让工作记忆里的每个元素都以TMemoryValue类型声明，而且可以取下列四个值之一：Yes、No、Maybe或Unknown。其主要目的是，让计算机对手知道当前玩家对手的科技状态。Yes表示玩家有某种科技，而No表示没有。如果玩家满足所有获得某种科技的条件，但其状态尚未被侦察兵确认，则其值为Maybe。如果计算机不知道玩家对某科技的能力，则取值Unknown。

计算机可以收集玩家当前科技状态的事实，做法是派出侦察兵，并做观察。例如，如果计算机派出一名侦察兵，而侦察兵看见玩家建了庙宇，则Temple应该设为Yes。使用一组if-then规则，在侦察兵确认之前，计算机就能根据既有事实推论玩家的科技状态。例如，看到图 11-1，如果玩家有伐木工和石匠，则有能力建庙宇。就此而言，Temple就会是Maybe之值。这种情况的规则，看起来也许就像例 11-2 那样。

例 11-2：庙宇规则示例

```
if(Woodcutter == Yes && Stonemason == Yes &&
     Temple == Unknown)
          Temple = Maybe;
```

推论也能以其他方式得到。例如，如果玩家被观察到有僧侣，则计算机可以推论，玩家一定有庙宇，因此，也一定有兵营、伐木工以及石匠。这种场景的规则看起来也许就像例 11-3 那样。

例 11-3：僧侣规则示例

```
if(Priest == Yes)
{
        Temple = Yes;
        Barracks = Yes;
        Woodcutter= Yes;
        Stonemason= Yes;
}
```

你可以替此科技树写出许多规则。例 11-4 是可以写出的其他规则。

例 11-4：其他规则示例

```
if(Peasants == Yes && Woodcutter == Unknown)
        Woodcutter = Maybe;

if(Peasants == Yes && Stonemason == Unknown)
        Stonemason = Maybe;

if(Woodcutter == Yes && Barracks == Unknown)
        Barracks = Maybe;

if(Woodcutter == Yes && Stonemason == Yes &&
   Temple == Unknown)
        Temple = Maybe;

if(Barracks == Yes && Blacksmith == Unknown)
        Blacksmith = Maybe;

if(Fletcher == Yes && FootSoldier == Yes &&
   Archer == Unknown)
        Archer = Maybe;

if(Woodcutter == Yes && WoodWalls == Unknown)
        WoodWalls = Maybe;

if(Stonemason == Yes && StoneWalls == Unknown)
            StoneWalls = Maybe;

if(Archer == Yes && Crossbowman == Unknown)
            Crossbowman = Maybe;

if(Archer == Maybe && Longbowman == Unknown)
            Longbowman = Maybe;

if(Longbowman == Yes)
{
        Archer = Yes;
        Fletcher = Yes;
        FootSoldier = Yes;
        Barracks = Yes;
        Woodcutter = Yes;
}
```

```
if(Cavalry == Yes)
{
        Blacksmith = Yes;
        FootSoldier = Yes;
        Barracks = Yes;
        Woodcutter = Yes;
}

if(StoneWalls == Yes)
        Stonemason = Yes;
```

如前所述，对此例而言能写的规则不止这些。你可以开发更多规则，包含如图11-1所示的所有可能科技。这里的想法是你可以写这类规则，并在游戏中不断执行（也就是每次游戏循环运行时），以保持计算机对手看待玩家科技能力观点的最新图像。就此例而言，计算机可以将这些信息，用于其他AI系统，以决定如何部署攻防兵力。

此例也让你大致了解规则系统的运作方式，实际上就是一组if-then规则，一组事实和断言。然而，注意到，开发人员经常不用本节所用的if语句建构规则系统。我们后面会讨论其他做法，但基本而言，直接把if语句写在程序里，会让某种推论难以达到。此外，开发人员时常使用描述语言或shell语言，使他们能建立规则并予以修改，而不用修改源代码再重新编译。

规则系统的推论

前一节我们看了规则系统的主要部分，让你知道如何在实时策略游戏中，利用这样的系统进行推论。本节我们打算采取比较正式的观点，讨论规则系统的推论方式。我们这里的目标是将推论的两种基本算法进行区分，并引进一些标准的规则系统术语，以让你日后寻找规则系统的信息时，提供技术词汇上的依据。（本章结尾处会提出一些参考数据。）

演绎法

规则式系统最常用的推论算法叫做演绎法（forward chaining）。这种算法由三个基本步骤组成。第一个步骤牵涉到把规则和储存在工作记忆里的事实配对，做法是检查每条规则的if部分，是否和工作记忆里的某组事实或断言相吻合。例如，在我们的科技树范例中，如果工作记忆指出Peasants = Yes而Woodcutter = Unknown，则我们知道这与例11-4的第一条规则相吻合，有启动的可能。当某条规则启动时，其then的部分会被执行。此外，很有可能工作记忆中的某组事实，与一条以上的规则吻合。就此而言，我们得搞清楚要启动哪条规则。接着就到了所谓的冲突解决（conflict resolution）步骤。

在冲突解决步骤中，我们必须检视所有吻合的规则，找出我们想要启动的那一条。我们可以用很多方式做决定。常用的做法是启动最先吻合的规则。有时候你也可以随机选择。

其他情况下，这些规则会加权，而权值最高的会被选出来。在对战实例中我们会采取捆绑的方法。

冲突解决步骤执行过之后，有一条规则会被选出来，然后我们予以启动。启动一条规则只意味着执行其 then 的部分。该规则也许会在工作记忆中断言某些新事实，诸如例 11-4 所示的那些规则；但也许会触发某些事件，或者调用其他函数来处理某些运算。

这三个步骤都执行过后，整个过程会一直重复，直到没有规则可以启动为止。当此现象发生时，工作记忆中所存储的东西，应该足以让规则系统根据开头既定的事实做推论。如果这看起来很朦胧，不用担心，当我们谈对战实例时，就会变得清晰了。

归纳法

归纳法（backward chaining）和演绎法恰好相反。我们还是有工作记忆和规则记忆，但此时不是把规则的 if 部分拿来和工作记忆配对，而是试着把 then 部分配对。换言之，采用归纳法时，我们一开始是有某种结果或目标，然后试着找出哪些规则应该启动，才能达到该结果或目标。回到科技树范例，假设其结果是玩家的骑兵单位呈 `Cavalry = Yes` 状态。为了弄清楚玩家怎么拥有骑兵的，我们可以用归纳法，以了解哪些规则必须启动，才能把 `Cavalry` 设为 `Yes`。

观察图 11-1，我们发现要有骑兵，玩家必须要有铁匠。这种情况的规则程序代码，如例 11-5 所示。

例 11-5：骑兵规则

```
if(Blacksmith == Yes)
    Cavalry =Yes
```

接着，如果玩家有铁匠，则必须有兵营。如果玩家有兵营，则必须先有伐木工，依此类推。我们可以继续后推这类逻辑，从目标点 `Cavalry = Yes` 往科技树上方走回去，确定达成此目标所需的所有规则和事实。这就是归纳法。

事实上，归纳法是递归的，比起演绎法，要难以实现。此外，直接把范例中这些 if 语句编进程序里，就很难在归纳法时，以规则的 then 部分和工作记忆中储存的事实，互相配对了。在对战实例中，我们将谈论如何不把 if-then 规则实际编入程序，而实现规则系统。

对战游戏攻击预测

此例中，我们的目标是预测武术对战游戏中，人类对手的下一个招式。基本的假设是玩家会试着去组合各种招式，以找出最有效的套路。这些套路可以是下踢、下踢、上踢；

或者是挥拳、挥拳、强踢等，诸如此类。我们想让计算机对手，能够利用玩家最近出的招以及玩家过去所出招式的某些模式，预测玩家下次要出什么招。如果计算机可以预测下一招，就能采取适当的反击、阻挡或闪躲行动，比如往侧边跳或往后退。这会让战斗模拟游戏有更强烈的真实感，给玩家新的挑战。

为了达到这种效果，我们要实现一个有学习能力的规则系统。让每条规则加权，强化某些规则，压抑另外一些规则，借此达到学习效果。第十三章我们将讨论另一种解决此问题的方法，到时候不是用规则，而是用条件概率（conditional probability）协助预测下一个招式。

为了让范例能在讨论的掌控范围内，我们要做一些简化工作。假定玩家的招式可以分成挥拳、下踢或上踢。然后，记录这三种招式的组合。即使做了简化，我们还是有27条规则，才能找出挥拳、下踢或上踢这三种招式的所有可能的组合。我们稍后就会看到这些规则，但首先要看执行工作记忆和规则记忆所需的结构和类。

工作记忆

例11-6是工作记忆的操作方式。

例11-6：工作记忆

```
enum TStrikes {Punch, LowKick, HighKick, Unknown};

struct TWorkingMemory {
        TStrikes        strikeA; // 前、前次攻击（数据）
        TStrikes        strikeB; // 前次攻击（数据）
        TStrikes        strikeC; // 下次预测的攻击（断言）
        // 注：可以在这里加上其他元素，比如要怎么反击等
};

TWorkingMemory  WorkingMemory;  // 全局工作记忆变量
```

TStrikes是枚举类型，也就是那些可能的招式。注意，我们让其包含Unknown在内，当计算机不知道下一招是什么时，就会用此值。

TWorkingMemory是定义工作记忆的结构。这里我们有三个元素：strikeA、strikeB以及strikeC。strikeC储存预测的下一招。strikeA和strikeB会根据已知事实，通过规则的演绎法而确认其值。strikeB代表上一次的招式，而strikeA代表的是strikeB之前的招式。三招的组合依序是strikeA、strikeB、strikeC，按此次序，strikeC就是由规则系统作出的预测。

有必要的话，我们可以在工作记忆中增加其他事实或断言。例如，我们可以加进一个反击招式的元素，也可以根据预测的下一招做判断。如果预测的下一招是下踢，我们可以

用规则断言出适当的反击招式，比如后退等。此例执行工作记忆和规则的方式，可让你在工作记忆中轻易增加新元素，也可以轻易增加新规则。

规则

例11-7是此例的规则类。注意，我们没有直接写出if-then规则。相反的，我们以TRule对象数组表示规则记忆。我们可以轻松使用if-then结构，然而，这里所用的方法可以让我们轻易增减规则，让本例用到的归纳法多少能简化一些。稍后再回到这个议题上来。

例11-7：规则类

```
class TRule {
public:
TRule();
void SetRule(TStrikes A, TStrikes B, TStrikes C);

TStrikes        antecedentA;
TStrikes        antecedentB;
TStrikes        consequentC;

bool            matched;
int             weight;
};
```

TRule对象有五个成员。前两个是antecedentA和antecedentB，相当于玩家的前两招。下一个成员是consequentC，相当于要预测的下一招，也就是我们要用规则判断的招式。如果我们用标准的if语句写规则，看起来就像这样：

```
if antecedentA AND antecedentB then consequentC
```

像“if X then Y”这样的if-then形式的规则，“if X”部分是前件，或者说是前提，而“then Y”部分是后件，或者说是结论。就此例而言，假定我们的规则是由两个参数的交集（AND）构成的：antecedentA和antecedentB。规则中的then部分是consequentC，也就是根据前两招而预期的招式。

TRule的下一个成员是matched。如果规则中的前件与工作记忆中的事实相吻合，这标号就设为true。更明确地讲，就某规则而言，如果antecedentA等于WorkingMemory.strikeA，而antecedentB等于WorkingMemory.strikeB，则规则就配对起来了。可能有一条以上的规则能吻合某组事实。这个matched成员可以协助我们记录那些配对起来的规则，在冲突解决步骤中，挑选一条规则来执行。

TRule的最后一个成员是weight。这是加权因子，我们可以通过调整，以强化或压抑规则。总之，它代表的就是每条规则的强度。从另一个角度来看，加权因子就是代表计

算机的判断，某条规则比起其他吻合的规则，更适用或更不适用。在冲突解决步骤中，如果有一条以上的规则吻合，我们会以最高权重的规则作为预测招式。如果下一招出击之后，我们发现启动了错误的规则，也就是做了错误的预测，就会降低该启动的规则的权重以压抑之。此外，我们会弄清楚应该启动哪条规则，然后增加其权重以强化之。

TRule只有两个方法：SetRule()和构造方法(constructor)。构造方法只是把matched赋初值false，把weight赋以0而已。我们以SetRule()设定其他成员：antecedentA、antecedentB以及consequentC；由此，就能定义出一条规则。SetRule()方法如例11-8所示。

例11-8：SetRule()方法

```
void TRule::SetRule(TStrikes A, TStrikes B, TStrikes C)
{
        antecedentA = A;
        antecedentB = B;
        consequentC = C;
}
```

此例需要几个全局变量。第一个是WorkingMemory，如例11-6所示。例11-9是其他的全局变量。

例11-9：全局变量

```
TRule           Rules[NUM_RULES];
int             PreviousRuleFired;

TStrikes        Prediction;
TStrikes        RandomPrediction;

int             N;
int             NSuccess;
int             NRandomSuccess;
```

这里的Rules是一个储存TRule对象的数组。Rules数组的大小由NUM_RULES决定，此例指定为27。PreviousRuleFired是一个整数值，用来储存上一次游戏循环中启动的规则索引值。Prediction记录的是规则系统所做的招式预测。技术上而言，我们不需要这个变量，因为预测招式都会储存在工作记忆中。

我们打算以RandomPrediction储存随机产生的预测招式，可以将其和我们按规则而得出的预测招式相比较。我们真正要比的是，由规则得出的预测招式的成功率和随机猜测的成功率。全局变量N储存预测次数。NSuccess储存规则系统所做的成功预测次数，而NRandomSuccess储存随机猜测的成功次数。我们算成功率的方法是成功次数除以总预测次数。

初始化

仿真程序开始时，或者说游戏开始时，我们必须对所有规则和工作记忆做初始化。例11-10的Initialize()函数会替我们完成此任务。

例11-10：Initialize()函数

```
void TForm1::Initialize(void)
{
        Rules[0].SetRule(Punch, Punch, Punch);
        Rules[1].SetRule(Punch, Punch, LowKick);
        Rules[2].SetRule(Punch, Punch, HighKick);
        Rules[3].SetRule(Punch, LowKick, Punch);
        Rules[4].SetRule(Punch, LowKick, LowKick);
        Rules[5].SetRule(Punch, LowKick, HighKick);
        Rules[6].SetRule(Punch, HighKick, Punch);
        Rules[7].SetRule(Punch, HighKick, LowKick);
        Rules[8].SetRule(Punch, HighKick, HighKick);
        Rules[9].SetRule(LowKick, Punch, Punch);
        Rules[10].SetRule(LowKick, Punch, LowKick);
        Rules[11].SetRule(LowKick, Punch, HighKick);
        Rules[12].SetRule(LowKick, LowKick, Punch);
        Rules[13].SetRule(LowKick, LowKick, LowKick);
        Rules[14].SetRule(LowKick, LowKick, HighKick);
        Rules[15].SetRule(LowKick, HighKick, Punch);
        Rules[16].SetRule(LowKick, HighKick, LowKick);
        Rules[17].SetRule(LowKick, HighKick, HighKick);
        Rules[18].SetRule(HighKick, Punch, Punch);
        Rules[19].SetRule(HighKick, Punch, LowKick);
        Rules[20].SetRule(HighKick, Punch, HighKick);
        Rules[21].SetRule(HighKick, LowKick, Punch);
        Rules[22].SetRule(HighKick, LowKick, LowKick);
        Rules[23].SetRule(HighKick, LowKick, HighKick);
        Rules[24].SetRule(HighKick, HighKick, Punch);
        Rules[25].SetRule(HighKick, HighKick, LowKick);
        Rules[26].SetRule(HighKick, HighKick, HighKick);

        WorkingMemory.strikeA = sUnknown;
        WorkingMemory.strikeB = sUnknown;
        WorkingMemory.strikeC = sUnknown;
        PreviousRuleFired = -1;

        N = 0;
        NSuccess = 0;
        NRandomSuccess = 0;
        UpdateForm();
}
```

这里我们有27条规则，对应出拳、下踢、上踢这三招的所有可能组合模式。例如，第一条规则Rules[0]可以读成这样：

```
if WorkingMemory.strikeA = AND WorkingMemory.strikeB=Punch
        then WorkingMemory.strikeC=Punch
```

检视这些规则可以发现，任何时刻都有一条以上的规则可以吻合工作记忆中的事实。例如，如果招式*A*和*B*都是出拳，则前三条规则都吻合，而预测的招式可以是出拳、下踢或上踢。这里就是加权因子的用处了，它可以协助我们找出要启动哪条吻合的规则。我们只用权重最高的规则。如果有两条或两条以上的规则有相同权重，那就用最前面那一条。

所有规则都设定好之后，接下来就是工作记忆的初始设定了。基本而言，工作记忆的所有元素都先指定为Unknown。

预测招式

当游戏开始运行，每次玩家出招之后，我们都必须做招式预测。如前所述，这样可以让计算机对手预测玩家接下来会出什么招。就此例而言，我们用函数ProcessMove()处理玩家出的每一招，并预测其下一招。例11-11就是ProcessMove()函数。

例11-11：ProcessMove()函数

```
TStrikes TForm1::ProcessMove(TStrikes move)
{
        int     i;
        int     RuleToFire = -1;

    // 第一区 :
        if(WorkingMemory.strikeA == sUnknown)
        {
                WorkingMemory.strikeA = move;
                return sUnknown;
        }

        if(WorkingMemory.strikeB == sUnknown)
        {
                WorkingMemory.strikeB = move;
                return sUnknown;
        }

     // 第二区 :
     // 先处理前次预测，记录并调整权重
        N++;
        if(move == Prediction)
        {
                NSuccess++;
                if(PreviousRuleFired != -1)
                        Rules[PreviousRuleFired].weight++;
```

```
    } else {
              if(PreviousRuleFired != -1)
                      Rules[PreviousRuleFired].weight--;

                  // 归纳法以增加应该启动的规则的权重 :
                  for(i=0; i<NUM_RULES; i++)
             {
                     if(Rules[i].matched && (Rules[i].consequentC == move))
                     {
                               Rules[i].weight++;
                               break;
                     }
             }
    }

    if(move == RandomPrediction)
              NRandomSuccess++;

    // 删除旧值
    WorkingMemory.strikeA = WorkingMemory.strikeB;
    WorkingMemory.strikeB = move;

// 第三区 :
// 开始做新预测
     for(i=0; i III NUM_RULES; i++)
     {
               if(Rules[i].antecedentA == WorkingMemory.strikeA &&
                  Rules[i].antecedentB == WorkingMemory.strikeB)
                         Rules[i].matched = true;
               else
                         Rules[i].matched = false;
     }

    // 选出权重最高的规则...
    RuleToFire = -1;
    for(i=0; i<NUM_RULES; i++)
    {
            if(Rules[i].matched)
            {
                      if(RuleToFire == -1)
                           RuleToFire = i;
                      else if(Rules[i].weight > Rules[RuleToFire].weight)
                           RuleToFire = i;
            }
    }

    // 启动规则
    if(RuleToFire != -1) {
              WorkingMemory.strikeC = Rules[RuleToFire].consequentC;
              PreviousRuleFired = RuleToFire;
    } else {
              WorkingMemory.strikeC = sUnknown;
              PreviousRuleFired = -1;
    }
```

```
        return WorkingMemory.strikeC;
}
```

你可以把这个函数分成三个部分，如注释中的“// 第一区”、“// 第二区”以及“// 第三区”。我们依次分别加以讨论。

第一区

第一区是填写工作记忆。游戏开始时，在工作记忆初始化之后，任何招式出击之前，工作记忆中只有Unknown值。这样是无法做预测的，所以我们要在玩家开始出招后，从玩家那里收集资料。第一招储存在WorkingMemory.strikeA中，而ProcessMove()只返回Unknown，没有试着做预测。当第二招打出之后，ProcessMove()会再次被调用，但这次第二招会储存在WorkingMemory.strikeB中。ProcessMove()再次返回Unknown。

第二区

ProcessMove()的第二区是处理前次预测，也就是上一次调用ProcessMove()后所返回的预测招式。第二区的首要任务是确认前次预测是否有效。ProcessMove()以move为参数。move是玩家最近一次出的招。因此，如果move等于储存在Prediction的前次预测招式，则我们的预测就是成功的。此例中，我们递增NSuccess，以更新成功率。然后，我们强化上次启动的规则，因为这是利用储存在工作记忆中的招式历程，而启动的正确规则。要强化规则，只要递增该规则的权重即可。

如果前次预测是错的，也就是说move不等于Prediction时，我们就必须把前次启动的规则给压抑住。要这么做，只要递减前次启动的规则权重即可。同时，我们也想强化应该启动的规则。为了这么做，我们得找出上次ProcessMove()调用时应该启动的规则。其结果需要用到归纳法。本质上，我们已知move，因此，知道前次预测招式的后件。所以，我们要做的就是绕行上次吻合的那组规则，找出谁的consequentC等于move。一旦找到该规则，就递增其权重，这样就完成了。

ProcessMove()里第二区的剩余部分就相当简单了。下一个任务是查看前次随机预测是否正确，如果是，就递增NRandomSuccess这个代表成功地随机预测次数的参数。

最后，我们要更新工作记忆中的招式，以便做新预测。其结果只是推移工作记忆中的招式，

并把最新的move加进来。明确地讲，WorkingMemory.strikeB变成WorkingMemory.strikeA，而move变成WorkingMemory.strikeB。现在，我们可以准备替工作记忆中储存的那些招式做新预测了。

第三区

参考例 11-11 的“// 第三区”，预测过程中的首要任务是找出符合工作记忆中事实的规则。我们将在“// 第三区”注释下面的第一个 for 循环做这件事。注意，这是演绎法中所谓的配对步骤。当某规则的 antecedentA 和 antecedentB 分时别等于 WorkingMemory.strikeA 和 WorkingMemory.strikeB 时，配对就成功了。

配对步骤完成后，我们必须从那些吻合的规则中挑选一条出来。这就是冲突解决步骤。基本而言，我们要做的就是绕行吻合的规则，找出 weight 值最高者。例 11-11“// 第三区”注释下面的第二个 for 循环就是在做这件事。这个循环完成工作之后，选定的规则的索引值会储存在 RuleToFire 中。要实际启动规则，我们只要把 Rules[RuleToFire] 的 consequentC 复制到 WorkingMemory.strikeC 就行了。

ProcessMove() 把要启动的规则索引值 RuleToFire 储存在 PreviousRuleFired，下次 ProcessMove() 被调用时，会在第二区中使用。最后，ProcessMove() 会返回预测的招式。

这就是该实例大致的情况了。执行此例而模拟出击的招式时，只要按出拳、下踢和上踢按钮，我们就能看到规则系统，顺利预测下一招。通过实验，我们发现成功率在 65%～80% 之间。这和随机猜测的成功率 30% 相比，显然，这里的规则系统运行得更好。

其他信息

本章仅略谈了规则系统。虽然我们谈了所有基本概念，也让你了解了规则系统的用途，但如果你想在大规模系统中应用，其他规则系统也值得探讨。

最佳化是值得关注的一个领域。就小型的规则集合而言，演绎法不会花太多时间，然而，对大型规则集而言，很多规则都能吻合某组事实，对冲突解决步骤进行优化，就是明智之举了。就此而言，最常用的算法叫做 Rete 算法。（参见文章《Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem》，作者 C. L. Forgy，《Artificial Intelligence》期刊，1982 年。）多数谈规则系统或专家系统的教科书，都会谈到 Rete 算法。

如你所见，在对战实例中不需要在规则系统内使用 if-then 语句。你甚至不必使用枚举类型或其他类型，比如整数、布尔值等。你可以用字符串在工作记忆中表示事实，然后以字符串对比函数，确认某规则（也是字符串）的前件是否符合工作记忆中储存的事实。这种做法替编译后的程序，打开了迎接描述式规则的大门，替设计师描述 AI 规则做好了铺垫。事实上，数十年来，开发人员早就用描述语言、描述规则系统了，比如知名的

Prolog、Lisp以及CLIPS语言。(甚至还有相当新的，基于Java的语言，名叫JESS。)使用描述语言实现规则系统的另一个优点，是容易修改、删除或扩充规则，而不必去修改编译后的游戏程序代码。

你可以不用别人开发的描述语言，自己另写一套，然而，应特别注意某些事项。写描述式规则系统可以处理包含某范围内的值的事实，再配上规则中有复合式的前件和后件，而且可能引发其他事件时，这种复杂程度是远远超过写那种只有布尔值事实，和简单规则结构的规则系统的。如果你想知道这是为什么，可以看第八章到提的《AI Application Programming》一书，作者M. Tim Jones (Charles River Media出版)。注意，作者列举的范例不像Prolog和前面提到的其他语言那样，属于一般用途的描述语言，但的确能让你了解如何从头实现简单规则描述算法的过程。回想本书第八章谈的基本描述机制，你可以把相同技巧用在本章所讨论的规则系统的编写上。

至于其他资源，因特网上充斥着许许多多关于规则系统和描述式shell语言的网页。如果你到因特网上搜索“规则系统”(通常简写为RBS，rule-based system)，肯定能找到数不胜数的链接，专门讨论某种状况下的规则系统。我们发现了对初学者有益的网站，现列举如下：

- *http://www.aaai.org/AITopics/html/expert.html*
- *http://ai-depot.com/Tutorial/RuleBased.html*
- *http://www.igda.org/ai/*

第十二章

概率概论

开发人员在游戏中使用的概率（probability）可分为：击中概率、损害概率以及性格概率（比如攻击或逃跑倾向）等。游戏使用概率可以增加一些不确定性。本章我们要学习概率的基本原理，讨论如何把这些基本原理用在游戏软件AI中，以增添某种不可预测性。设置本章的另一个目的是作为下一章内容的基础，下一章要讨论不确定状态下的决策以及贝叶斯分析法。

如何在游戏中使用概率？

贝叶斯分析法（Bayesian analysis）可在不确定状态下做决策，其基础和概率密不可分。遗传算法也会在某种程度上用到概率，例如，求突变率的运算。即使是神经网络，也要用到概率方法。本书后面几章会适当地讨论这些和概率密切相关的方法。

随机性

由于我们讨论的范例严重依赖随机数的产生，我们就先谈产生随机数的程序。产生随机数的标准C函数是`rand()`，会在0和`RAND_MAX`之间产生随机整数。一般而言，`RAND_MAX`默认为32727。为了在0和99之间取得随机数，可以用`rand() % 100`。同样的，要在0和*N-1*之间取得随机数，就用`rand() % N`。别忘了在程序开始时，要调用`srand(seed)`，给随机数产生器一个种子数（seed）。注意，`srand()`携带的是一个`unsigned int`类型的参数，以此作为种子数对随机数产生器作初始设定。

举一个非常简单的实例，假设你决定要在游戏中让单位的移动多一些随机性。就此而言，当该单位面对挑战时，有25%的概率往左移，有25%的概率往右移，还有50%的概率

往后退。有了这些概率，你只需在0和99之间取随机数，再做一些测试，决定该单位要往哪个方向移动。为了做这些测试，我们要把0～24这个范围作为往左移的可能值，同样的，我们把75～99的范围作为往右移的可能值。另外，25～74（含24和74）的范围作为后退的可能值。一旦选出一个随机数之后，我们只需测试该数字落在哪个范围内，就能做出适当的移动。显然，这是一个很简单的例子，有人会说这不是智能移动，然而，开发人员恰恰常用这种技巧，来显示一些不确定性给玩家看，使得玩家面对该单位时，较难预测其移动方向。

击中概率

概率在游戏中另一种常见的用法是，用来代表生物或玩家在战斗中击中对手的机会。一般而言，游戏开发人员会根据玩家及其对手的特点，定义几种概率。例如，在角色扮演游戏里，你可以设敏捷度一般的玩家，有60%的概率可在肉搏战中以刀子刺中对手。如果玩家的敏捷度较高，就可以说他作战时能以高一点的成功率击中对手；例如，可以设有90%的机会可以击中对手。注意，本质上，这都是条件概率。我们是说，玩家有高敏捷度时，则有90%的成功概率；而敏捷度一般时，成功的概率只有60%。总之，概率对事件而言都是有条件的，尽管我们没有像上一节那样，明确地指明条件或者概率。事实上，游戏中，考虑其他因素后，时常要对诸如击中概率这类概率做调整。例如，你可以说如果玩家拿的是“快刀”，成功击中对手的概率就提高到95%。你也可以说如果对手有魔力盔甲，玩家击中的成功率就降到85%。这样的因素可以有无限多，而且可以列举出来，通常放在所谓的击中概率表格（hit probability table）里，借此，根据是否列举了那些事件，计算适当的概率。

角色能力

另一个在游戏中使用概率的实例是，定义角色类型或生物种类的能力。例如，假设你有一个角色扮演游戏，玩家可扮演剧中人物的角色有法师、斗士、游人或侠客。每个类型和其他类型相比，都有其优缺点，你可以在表中列举出技能及概率，以说明每个类型的特点。表12-1就是这种角色类型能力表的简易范例。

表12-1：角色类型能力

能力	法师	斗士	游人	侠客
法术	0.9	0.05	0.2	0.1
使剑	0.1	0.9	0.7	0.75
砍柴	0.3	0.5	0.6	0.8

表 12-1：角色类型能力（续）

能力	法师	斗士	游人	侠客
开锁	0.15	0.1	0.05	05
认陷阱	0.13	0.05	0.2	0.7
看地图	0.4	0.2	0.1	0.8
…	…	…	…	…

一般而言，像这样的角色类型表，它包含的技能比起我们这里列出的几个要多出很多。然而，该表已足以说明了每种技能，都具有相应的成功概率这件事，也就是说，角色类型的能力是有条件的。例如，法师使用法术的成功机会是90%，而斗士只有5%，等等。实际上，这些概率会在每位玩家的总类型等级上再加条件。例如，初级法师使用法术的成功率只有10%。这里的想法是随着玩家的努力挣得更高等级时，其技能也得到逐步提升，因此，其技能的成功概率也会增加，并以此反映其进展。

从计算机角度来看，诸如游戏里的AI角色，游戏世界里的所有生物，也有类似的概率表格，根据其种类而定义其能力。例如，恶龙精通的能力和巨猿的就大不相同，等等。

状态转移

你可以结合概率以及有限状态机（可以管理诸多生物状态）的状态转移，让生物的能力再进一步提高（参见第九章有关有限状态机的讨论）。例如，图 12-1 是一些生物的可能状态。

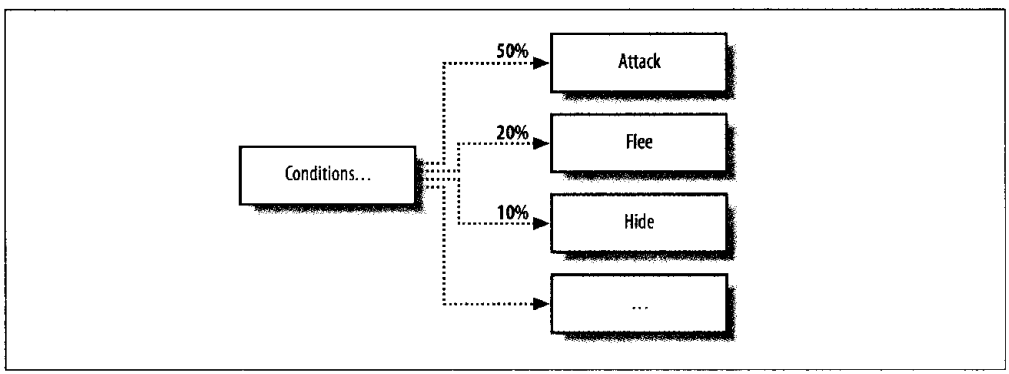


图 12-1：生物状态

假设这是有限状态机可以运行的分界线之一，也就是在计算机控制的生物遇到玩家时。图中，“条件”指的就是要检查的必要条件，也就是在有限状态机中会引发这组状态（攻

击、逃离、躲藏等）的条件。条件可以是“玩家出现在眼前，手里拿着武器”这类事件。此时，不要断然替此生物确定一种状态，可以替每种可能的状态都指定某个概率。为了说明这一点，我们让生物有50%的机会攻击玩家，有20%的机会逃离此场景，同时有10%的机会可以躲起来。如果要让变化更多，你可以让不同种类的生物有不同的概率，让某些生物比其他生物更有侵略性或更温驯，等等。此外，在每种生物种类里，你可以给个别的生物指定不同的概率，使其拥有独特的性格。

为了从中选取指定概率的某种状态，你可以在0到99之间取随机数，检查该数是否在每个概率的特定范围里。要不然，也可以采用抽签的方法。就此而言，你可以列举每种状态，例如，0是攻击，1是逃离，2是躲藏，等等，然后，根据其概率的大小，将这些值填满整个数组。例如，对攻击状态而言，你得把整个数组的一半都填满0。数组填满后，在0到数组最大数值-1之间取随机数，再以此数为数组索引值，取出选定的状态。

适应力

概率在游戏中令人较为动容的是，随着游戏的进行而更新某些概率，以促进计算机控制的单位学习或适应的能力。例如，在游戏中，你可以收集某种生物和某类玩家（例如，法师、斗士等）之间接触的次数和结果的统计资料。然后，立即计算生物和玩家遭遇后的死亡概率。基本上，这是求概率很常用的方法。有了死亡概率之后，你就可以决定以后是否和这类玩家战斗，而不是由生物决定。如果死亡率较高，表示某类玩家会杀死这种生物，就可以让这种生物开始避开该类玩家。另一方面，如果死亡率指出生物可以和某类玩家拼杀，就可以让生物去寻找该类玩家。

下一章我们会谈这类分析，根据玩家是否属于某种类型的概率，以及和此类玩家遭遇的概率，来计算死亡率这类信息。你可以再进一步，也就是不要假定生物已知玩家属于哪种类型；也就是说，生物对玩家的了解是不确定的，而生物必须在遭遇时自行推断类型，以做决策。游戏进行中，有能力收集统计资料，使用概率做决策，显然可以提供某些有趣的可能性。

到目前为止，我们讨论概率还没有给出它的正式定义。下一章谈贝叶斯方法之前，我们必须讲清楚才行。此外，我们要谈几条概率基础规则，这是你必须知道的，以便充分地领会下一章的内容。因此，本章接下来要谈概率理论的基本概念。如果你已驾轻就熟，就可以直接跳到下一章了。

何谓概率？

“何谓概率？”这个问题简单来说是没有固定定义的，其答案并不唯一。我们可以用好

几种方式解读概率，根据考虑的情况以及考虑该情况的主体来决定。下面几节我们要考虑三种常见的概率解读，这三种方式在游戏中都可找到各自的应用之处。我们多半以自然的方式讨论，使其易于了解。

标准概率

标准概率指的是事件和可能性发生的概率或可能的结果。给定一个事件 E，在总数为 N 的可能结果中，会发生 n 种结果，则此事件发生的概率 p 为：

$$p = P(E) = n/N$$

这里的 $P(E)$就是事件 E 的概率，等于 E 在 N 种可能结果中发生的次数。$P(E)$通常称为该事件的成功率，而该事件的失败率就是 $1\text{-}P(E)$。总结如下：

$$\text{成功率 } p_s = n/N$$

$$\text{失败率 } p_f = 1 - p_s$$

注意，概率的范围介于 0 到 1 之间，而成功率和失败率的总和，也就是 p_s+p_f，必然等于 1。

我们举一个简单的实例。假设你掷六面骰子，得“4”的概率是 1/6，四能从六种可能结果中出现的方式只有一种，因为只有一面为 4。此例中，事件 E 就是出现 4 的事件。掷一颗骰子时，4 只能出现一次，因此，n=1。可能结果的总数 N 是 6，因此，$P(E = 4) = 1/6$。显然，此例中，1～6 之间的任何数字出现的概率都是 1/6，因为每个数字在六种可能结果中都只有一种显示方式。

现在，考虑同时掷出两颗骰子的情况。出现的两个数字，其和为 5 的概率是多少？这里我们感兴趣的事件是，掷出的两数字其和为 5。此例中，有四种可能的方式能使其和为 5，如图 12-2 所示。

注意，第一颗骰子显示为二而第二颗骰子显示为 3，这种结果和第一颗骰子显示为 3 而第二颗骰子显示为 2 是完全不同的。就此而言，N 为 36，也就是说掷两颗骰子的可能结果有 36 种。那么，和为五的概率是其出现的次数除以 36，此处的 36 就是掷两颗骰子的可能结果总数。结果是 4/36 或 1/9 的概率。

你可以用类似的方法找出任何可能出现之和的概率。例如，和为 7 的可能方式如表 12-2 所示。

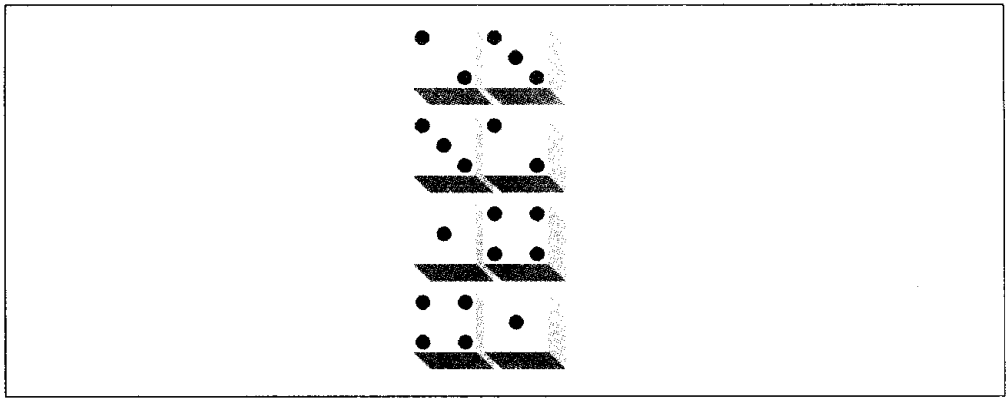


图 12-2：掷两颗骰子其和为 5

表 12-2：两颗骰子掷出和为 7 的可能情况

第一颗骰子	第二颗骰子
1	6
6	1
2	5
5	2
3	4
4	3

此例中，和为 7 的概率是 6/36 或 1/6。换言之，和为七的概率是 16.7%。我们可以把介于 0 和 1 之间的概率乘以 100%，表示成百分比的形式。

频率解读

概率的频率解读（frequency interpretation），也称为相对频率（relative frequency）或客观概率（objective probability），考虑的是事件和样本（或实验）。如果某实验做了 N 次，而某事件 E 发生了 n 次，则 E 出现的概率就是：

$$P(E) = n/N, \text{ 其中 } N \to \infty$$

注意这里有个条件，当实验次数很大时，$P(E)$才等于 n/N。对有限的实验次数而言，所得概率是近似值或经验值，因为这是从统计资料推出来的。用于计算指定的事件时，经验概率和理论概率有点不同。此外，我们假设每次实验都是各自独立的，也就是说某次实验的结果不会影响其他实验的结果。

考虑一个简单实验，掷1000次硬币。此实验的结果是“人头”面出现了510次。因此，得到“人头”的概率是510/1000：

$$P(\text{人头}) = 0.51 \text{ 或 } 51\%$$

当然，此例中，我们知道 P（人头）是0.5或50%。如果我们继续做实验，掷的次数足够多时，由经验可知，P（人头）会逼近0.5。

主观解读

主观概率（subjective probability）是一种分析手段，其值介于0到1之间，也就是说某人根据其知识、经验或判断，认为某特定事件发生的机会。当分析中的事件或实验，不会重复发生，也就是说我们无法以频率计算概率时，就可使用这种解读方式。

主观概率随处可见，我们可以说“明天可能下雨”、“我通过考试的可能性很大”、“明天圣人队可能会赢”等。就此而言，我们对此类事件的结果的信念，是根植于对这些事件的了解，无论是全面的还是片断的了解，同时也考虑了很多潜在的相关因素来做判断。例如，今天在下雨的事实，会让我们认为明天可能也会下雨。我们相信明天足球赛圣人队会赢，是因为我们知道另一队的明星后卫受了伤，所以圣人队成功的概率是要高于正常情况下的成功概率。

考虑此例：假设你要申请公司里游戏设计师领队的职位。你也许会说：“我有50%的机会得到那个位子。”因为你知道团队中，某个和你有相同资格的人也在候选人之列。另一方面，你可能会说：“我有75%的机得到那个位子。”因为你进这家公司的资历，比其他候选同事都要久。如果是这样，同时你也知道其他候选人，在不同游戏项目中有臭名远播的时程超过限度的记录，也许会倾向于修正你的信念，认为你得到该职位的机会可升高到诸如90%。简言之，贝叶斯分析法可以让我们根据新信息，更新我们对某事件的信念。下一章会深入探讨这一点。

主观概率通常很难量化，即便某些人对某事件有很不错的直觉。例如，如果你说你有可能通过考试，你说的实际概率究竟是60%、80%还是90%？你可以利用某些技术协助你，对主观概率进行量化，我们会简单谈两种方法。然而，讨论之前，必须先谈两个基本概念：赔率（odds）和期望值（expectation）。

赔率

赔率时常出现在下赌注的场合。例如，Sunflower Petals这匹马恐怕没希望赢得下礼拜的马赛，她失败的赔率是20:1。足球迷赌礼拜天巨人队会赢，也许赔率是3:1。很多情

况下，以赔率来推测概率，而不是用介于0到1之间或者百分比的某些值，会比较简单也更符合直觉。赔率能反映出概率，你可以用一些简单的关系做转换。

如果我们以某事件E的成功率来表示赔率的话，那就是$a:b$，而该事件成功的概率$P(E)$就是：

$$P(E) = a/(a + b)$$

我们可以由此把概率变为赔率。如果你知道某事件成功的概率$P(E)$，那么事件成功的赔率就是$P(E):(1-P(E))$。例如，如果你通过考试的赔率是9：1，则你通过考试的概率是0.9或90%。然而，如果你通过考试的概率是0.6或60%，因为你没像你想的那样用功念书，那么，你通过考试的赔率就是60：40或1.5：1。

期望值

以期望值来考虑概率的问题时常有帮助。数学期望值（Mathematical expectation）是指某个独立随机数X的期望值，可以由任何值（x_0、x_1、$x_2 \cdots x_n$）及其相对应的概率（p_0，p_1，p_2，…，p_n）构成。你可以替如此分布的结果计算其期望值：

$$E(X) = x_0p_0 + x_1p_1 + x_2p_2 + x_3p_3 + \cdots + x_np_n$$

类似这样的分布，你可以把期望值想象成平均值。统计学家把期望值，看成一种观察集中趋势的方法。决策理论家把期望值，看成是量度收益的方法。

举一个非常简单的实例，如果你赢100元的概率是0.12，那你的期望值就是12元，也就是100乘以0.12。

另举一例说明，假设你有一个永不停歇的网络在线游戏，你在里面监视着每晚在地方客栈聚会的玩家。假设你从监视中，建立了如表12-3所示的概率，也就是每晚玩家在客栈聚会的人数。我们再假设你计算这些基于频率的概率样本，大约都在每天同一时刻，例如，你的间谍每天晚上都潜入客栈收集情报，以防备来日入侵。

表12-3：每晚在客栈的玩家人数概率

# 玩家	概率
0	0.02
2	0.08
4	0.20
6	0.24
8	0.17

表 12-3：每晚在客栈的玩家人数概率（续）

# 玩家	概率
10	0.13
12	0.10
14	0.05
16	0.01

注意，这样的分布结果形成了一个互斥的完整的集合。也就是说，不能有 0 位或 8 位玩家同一时刻都在那儿（只能有一种情况），而所有这些结果的概率之和必为 1。就此而言，每晚在客栈的玩家数之期望值就是7.1。你可以把表格中每行的两个数字相乘再取其和，就能求得此数。因此，在任何晚上，某人可以期望在客栈平均出现七位玩家。注意到使用这种分析法时，入侵一方可以估出要派多少人马到客栈才能予以掌控。

确定主观概率的方法

如前所述，通常很难替主观概率确定明确的数字。虽然你可能对某事件的概率有一定把握，但你会发现实际用某特定值指定该事件的概率很困难。为了解决这方面的难题，有好几种常用的方法，可以辅助你替主观概率指定其值。我们简单讨论其中两种。

指定主观概率值的第一种技巧是打赌方法。我们再回到先前申请职位的例子。假设另一位同事问你，是否愿意打赌能否获得该职缺。假设该同事认为你得不到该职缺，而且愿意出 1 美元打赌，但要求 9: 1 的赔率 —— 如果你输了（没升职），就付 9 美元，如果你赢了（升职），就赚 1 美元。你会不会接受？如果你接受，就表示你认为这是公平的赌注，本质上，也就是说你得到升职的概率是 90%。如果提议你升职的赔率为 4 : 1，本质上，也就是说，你认为你会升职的概率是 4/5 或 80%。

使用此法时，前提是你认为彼此同意的赔率构成了公平的打赌。公平的打赌是一种主观的想法，也就是说某人预期的收益是 0，无论你赌的是哪一边。假设你接受 9 : 1 的赔率，觉得这是公平打赌。就此而言，你预期会赢($1)(0.9)或 90 美分。这是你会赢的金额乘以你会赢的概率。同时，你也预期会输($9)(0.1)或 90 美分，也就是你打赌的金额乘以你会输的概率。因此，你预期的净收益，就是你预期会赢的减掉你预期会输的，刚好就是 0。现在，如果你碰上 9 : 1 赔率的赌注，但你实际觉得你升职的成功率只有 80%，没有 90%，那么，你预期的收益会是：

$$(\$1)(0.8) - (\$9)(0.2) = -\$1$$

此例中，你预期会输 1 美元，也就是说这是不公平的打赌。

这里谈的打赌方法就是所谓的不赌就闭嘴（put up or shut up），你必须认真去想，愿意以事件结果打赌多少金额以求概率。这想法就是你应该对特定结果有很充足的信念。

然而，这种方法有个问题，因为个人忍受风险的能力不一。当我们谈 $1 : $9 时，输掉 $9 的想法对你没什么大不了的，而且你可能比较有那种接受打赌的倾向。然而，如果打赌是 $100 : $900，甚至是 $1000 : $9000 呢？显然，多数没有太多钱的理性的人，碰上大额金钱的打赌时，会认真想一想他们对某些结果的信念。在某些情况下，输掉这么多钱的风险会覆盖他们对特定结果发生的信念，即使他们的主观概率很准。此外，当察觉到的风险变得重要时，用这个方法也会出问题：某人的主观概率，可能因为他们察觉到的风险而受到左右。

打赌方法另一种替代法叫做合理价格（fair price）方法。此时不是打赌某事件的结果，而是你问自己对某事件的合理价格是多少。例如，假设作者的书卖得好的话，他可以赚 $30,000 分红。此外，如果书滞销，他什么也拿不到。现在，假设出版商给作者一个选项，也就是拿预付的保证稿费 $10,000，但必须同意放弃后续的分红金。现在的问题是，这本书是否畅销，从作者主观的概率来看，也就是他的信念，又是多少？

如果作者接受这种安排，我们可以推论 $10,000 大于他的期望值，也就是 $10,000 ≥ ($30,000)(p)，这里的 p 是他指定的书籍销售成功的主观概率。因此，此例中，他把该书销售成功的信念表示成 p，小于 0.33 或 33%。要求此值，我们只要解出 p，也就是 p ≥ $10,000/$30,000。如果作者拒绝接受该交易，他显然认为这书成功的概率大于 33%，也就是说他的期望值大于 $10,000。

为了明确找出作者认为书籍销售成功的概率实际是多少，我们可以直接问他能接受的保证稿费是多少，我们问的是他认为这本书的酬金是多少才算合理价格。从其回应中，我们就能利用前述期望值的公式，来计算其对该书销售成功与否的主观概率。如果 U 是他能接受的保证稿费，则该书成功的主观概率 p 就是 U/$30,000。

这种合理价格方法可以有很多种形式，和你试图估算的主观概率有关。这里的想法是想到某些情况下自身的钱有风险时（如同打赌方法那样），能够除去任何可能引起的偏见。

概率规则

正式的概率理论有几条规则，制约着概率的计算方式。我们要讨论这些规则，替下一章铺垫。虽然我们已经讨论过其中一些规则，但这里为了求完整起见，要重述一遍。下面讨论时，我们只做理论叙述，不提供特定实例。然而，下一章会看到这些规则的运用。如果你想知道每条规则的特定实例，可以参考任何概率的入门书籍。

规则 1

这条规则说的是某事件的概率 $P(A)$，必为介于 0 和 1 之间（包含 0 和 1）的实数。这条规则限制了概率值的范围。一方面，不能出现负值的概率，而另一方面，事件的概率也不能大于 1，1 指的就是该事件绝对会发生。

规则 2

此规则为规则 1 的扩充。如果 S 代表该事件的整个样本空间，则 S 的概率为 1。这样讲是因为样本空间包含所有可能结果，因此，结果之一发生的概率是 100%。这里利用范氏图（Venn diagram）就能看见样本空间和事件的关系。图 12-3 是范氏图，样本空间 S 以及位于该样本空间内的事件 A 和事件 B。

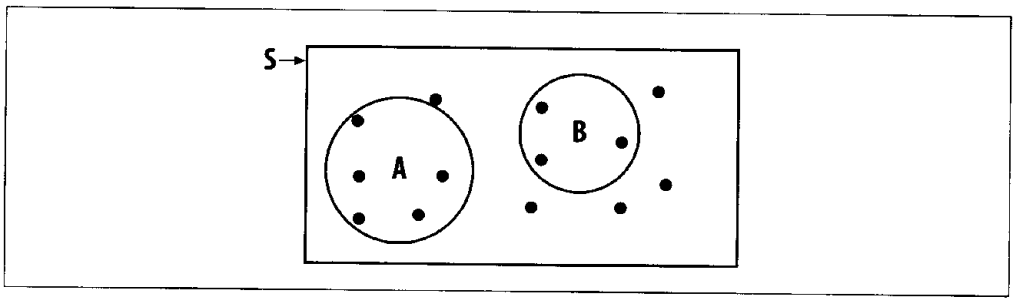


图 12-3：范氏图

点代表的是该空间里的样本，而 A 和 B 圆圈的相对大小，指的就是它们的相对概率，更明确地讲，它们的面积代表的就是它们的概率。

规则 3

如果事件 A 发生的概率是 $P(A)$，而 A 不会发生的事件记为 A'，则其概率 $P(A')$ 为 $1 - P(A)$。这条规则是说，某事件要么发生，要么不发生，而发生或不发生的概率是 1，也就是说，该事件一定会发生或不会发生。图 12-4 是事件 A 和事件 A' 的范氏图。

显然，事件 A' 包括了事件 A 以外的所有 S 样本空间。

规则 4

这条规则说的是，如果两事件 A 和 B 互斥，则任何时间，只有其中一个事件可以发生。例如，在游戏中，生物已死和生物还活着的事件彼此互斥。生物不能同时又死又活。图 12-5 显示了 A 和 B 是彼此互斥的两事件。

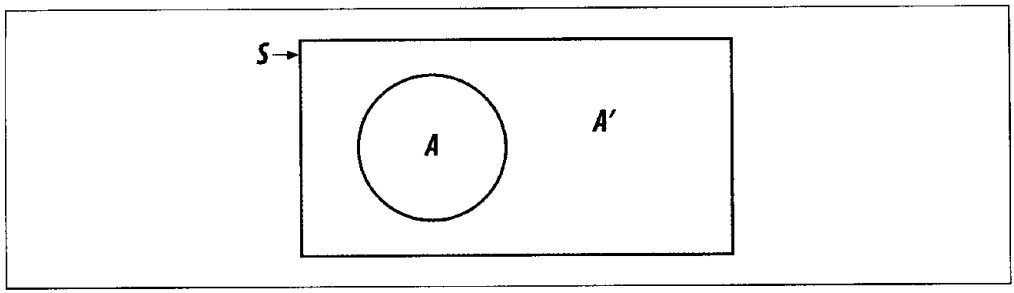


图 12-4：P(A)与 P(A′)

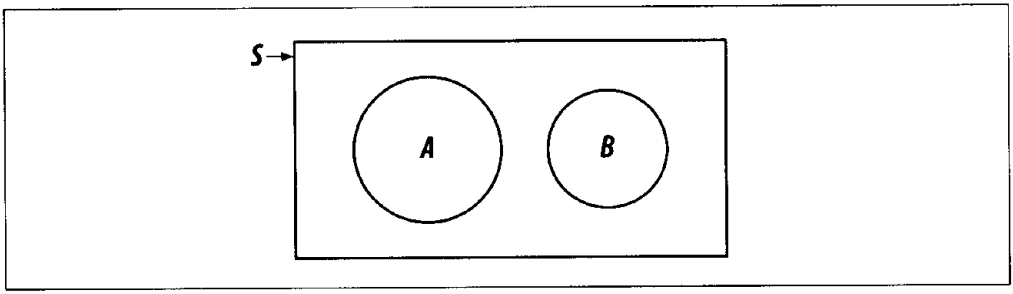


图 12-5：互斥的两事件

注意，代表该两事件的区域没有重叠。就两个互斥事件 A 和 B 而言，事件 A 或事件 B 发生的概率为：

$$P(A \cup B) = P(A) + P(B)$$

这里的 $P(A \cup B)$就是事件 A 或事件 B 的概率，也就是事件 A 或事件 B 发生的概率，而 $P(A)$及 $P(B)$分别是事件 A 和事件 B 各自的概率。

你可以推广此规则，使其不限于两个互斥事件。例如，如果 A、B、C 以及 D 是四个互斥事件，则 A、B、C 或 D 发生的概率为：

$$P(A \cup B \cup C \cup D) = P(A) + P(B) + P(C) + P(D)$$

理论上来说，你可以将此规则推广到任何数目的互斥事件。

规则 5

这条规则说的是如果考虑的事件并不互斥，我们必须修改规则 4 的公式。例如，在游戏中，某个生物可以是活着、死掉或受伤。虽然活着和死掉是互斥的，但活着和受伤就不互斥。生物可以同时活着又受伤。图 12-6 是这些并不互斥的事件。

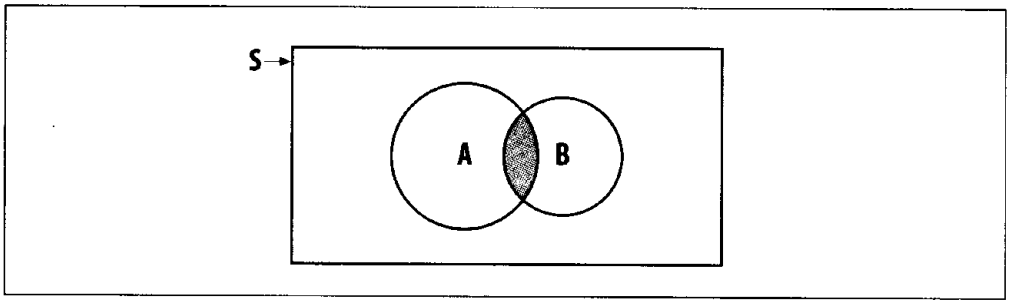


图 12-6：并不互斥的两事件

此例中，事件 A 和事件 B 的区域是重叠的，这表示事件 A 可以发生，事件 B 也可以发生，事件 A 和 B 可以同时发生。图 12-6 的阴影部分表示 A 和 B 可同时发生。因此，要计算事件 A 或 B 在此时的概率，我们可以用下列公式：

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

此公式中，$P(A \cap B)$就是事件 A 和 B 同时发生的概率。

你也可以推广此公式，使之适用于两个以上非互斥的事件。图 12-7 所示的是三个事件 A、B 和 C，彼此不互斥。

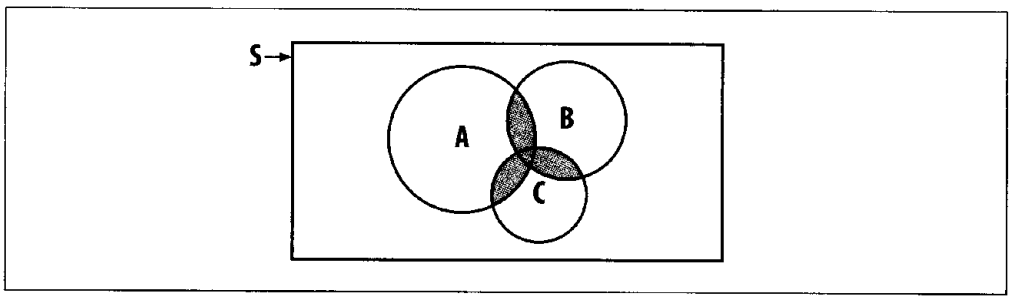


图 12-7：三个不互斥的事件

要计算事件 A、B 或 C 发生的概率，我们必须算出图 12-7 里阴影部分的概率。公式如下所示：

$$P(A \cup B \cup C) =$$
$$\{P(A) + P(B) + P(C)\} - \{P(A \cap B) + P(A \cap C) + P(B \cap C) + P(A \cap B \cap C)\}$$

规则 6

这条规则是说，如果两事件 A 和 B 是独立的，也就是说某事件的发生，不依赖另一事件是否发生，则事件 A 和 B 两者皆发生的概率为：

$$P(A \cap B) = P(A)\,P(B)$$

例如，游戏中两独立事件是，玩家碰上闲逛中的怪兽以及玩家正在生火。这两个事件中任一事件的发生与否，和另一事件是否发生无关。现在，考虑另一事件：玩家正在砍树。此例中，事件是玩家碰上闲逛中的怪兽，该事件和玩家正在砍树有些相关。这两事件并非互相独立。砍树时，玩家一定在森林里，因此增加了他碰上闲逛中的怪兽的可能性。

参见图 12-6，此概率对应事件 A 和 B 的阴影部分。

如果事件 A 和 B 并非互相独立，我们就必须处理这些事件所谓的条件概率。前述公式不适用有条件的情况。制约条件概率的规则很重要，尤其是用贝叶斯分析法时，所以，我们打算下一节专门讨论。

条件概率

当事件不独立时，就说是有条件的。例如，如果某天你回家后发现草地是湿的，那么你在上班时下雨的概率有多大？有可能是有人在你上班时打开了洒水装置，所以，结果为你的草地是湿的条件是，有没有下雨或者是否有人打开洒水装置。你可以利用贝叶斯分析法解决这类问题，下一章就会谈。但你马上就会知道，贝叶斯分析法的基础是条件概率。

一般而言，如果事件 A 依赖事件 B 是否发生，我们就不能用规则 6 适合独立事件的公式。假设有两个相互关联的事件，我们把 B 发生后，A 发生的概率写为 $P(A|B)$。同样的，把 A 发生后，B 发生的概率写为 $P(B|A)$。注意，$P(A|B)$不一定等于 $P(B|A)$。

为了找出 A 和 B 同时发生的合成概率，我们可以用下列公式：

$$P(A \cap B) = P(A)\,P(B|A)$$

这个公式是说，两个相互关联的事件 A 和 B 同时发生的概率，是事件 A 发生的概率，乘以事件 A 发生后事件 B 发生的概率。

我们可将此扩充到三个相互关联的事件，A、B 以及 C 事件，如下所示：

$$P(A \cap B \cap C) = P(A)\,P(B|A)\,P(C|A, B)$$

这个公式是说，事件 A、B 以及 C 同时发生的概率，等于事件 A 发生的概率，乘以事件 A 发生后事件 B 发生的概率，再乘以事件 A 和 B 都发生后事件 C 发生的概率。

通常，我们比较感兴趣的是某条件或事件发生后，某事件发生的概率。因此，我们通常会这样写：

$$P(B|A) = P(A \cap B)/P(A)$$

这个公式是说，事件 A 发生后事件 B 发生的条件概率等于事件 A 和 B 都发生的概率再除以事件 A 发生的概率。我们知道，$P(A \cap B)$ 也等于 $P(B)\ P(A|B)$，所以，我们可以把上述公式里的 $P(A \cap B)$ 换掉，如下所示：

$$P(B|A) = P(B)P(A|B)/P(A)$$

这就是所谓的贝叶斯规则。下一章要叙述贝叶斯规则，我们会谈一些实例。

第十三章

不确定状态下的决策：贝叶斯技术

本章要介绍贝叶斯推论和贝叶斯网络，教你怎么把这些技术应用在游戏中。明确地讲，我们要教你怎么使用这些技术让非玩家角色（NPC）在游戏世界处于不确定状态下做决策。我们也会教给你简单的贝叶斯模型，让你的计算机控制角色可以适应变动的情况。我们会用到大量的概率，所以，如果你不熟悉概率，最好先读一读第十二章，再回来读本章。

详谈贝叶斯网络之前，我们先讨论假想的一个例子。假设你正在写角色扮演游戏，要让玩家可以将宝物存放在游戏世界中散布各处的储物柜里。玩家可以用这些储物柜存放任何他们想要存放的东西，但他们要冒着被NPC抢劫这些储物柜的风险。为了阻止抢劫，玩家如果有技能和素材就能设置陷阱以保护储物柜。现在，身为游戏开发人员，你面对的问题是怎么写NPC盗贼的程序代码，让他们在找到储物柜时决定是否予以开启。

其中一种做法是让NPC试着打开找到所有的储物柜。虽然操作很简单，但这种做法没什么乐趣，也会碰到一些你不想要的结果。首先，让NPC打开所有储物柜，就把玩家设陷阱保护储物柜的用意给破坏掉了。其次，如果玩家发现NPC无论如何都要打开储物柜，那么玩家以后就会利用这个事实，替空储物柜设陷阱，以使NPC变弱，甚至是杀掉NPC，而不必直接和他们面对面战斗。

你的另一种选择是作弊，让NPC清楚知道找到的储物柜是否设有陷阱，然后让他们避开设有陷阱的储物柜。虽然这会让陷阱变成有效的拦阻手段，但可视为不公平条件。此外，这种做法毫无变化可言，不用多久，游戏就会变得很无趣。

还有一种较佳的可行方法是让NPC具有某些知识，不过并不全面，同时让他们能根据这些知识做推理。此外，如果我们让NPC有某种记忆，他们就有可能学习或适应，因此，就能避开先前提过的，设置空储物柜陷阱这种手段。本章后面会再回来看这个实例。到

时候，我们会以概率和游戏中收集的统计数据作为NPC记忆，然后以贝叶斯模型，作为NPC的推论或决策机制。

注意，此例中，我们实际上可以给予NPC全面的知识，但我们可以引入不确定性，以让事情更有趣。在其他游戏场景中，你也许无法给予NPC全面的知识，因为你根本办不到。例如，在对战游戏中，你无法确知玩家下一招是什么，因此，NPC对手也不知道。然而，你可以用贝叶斯技术和概率，赋予NPC对手预测下一招的能力（也就是预期下一招），成功率比直接猜测的要高出两倍。本章后面会详谈这个实例和其他例子。首先，我们先谈贝叶斯分析法的基础。

何谓贝叶斯网络？

贝叶斯网络是一些图形，可以替特定问题简明表示出随机变量（随机数）间的关系。这些图形有助于在面对不确定状态时，做推理或决策。这样的推理在很大程度上依赖第十二章讨论的贝叶斯规则。本章我们要用简单的贝叶斯网络，仿真特定问题场景，让NPC面对游戏世界的不确定信息，做出决策。在讲一些特定实例之前，我们先来看贝叶斯网络的细节。

结构

贝叶斯网络由代表随机变量的节点，以及代表随机变量间因果关系的弧或连线组成。图13-1是贝叶斯网络的实例。想象一个游戏，NPC碰巧遇上储物柜，也许被上锁（locked），也许没上锁。是否上锁，取决于是否存放宝物（treasure）或是否设有陷阱（trapped）。

此例中，标示为*T*、*Tr*以及*L*的节点，代表的是随机变量（在概率中称为事件）。连接每个节点的箭头代表的是因果关系。这里，可以把箭头的起始节点当成父母，箭头指向的节点当成子女，以父母产生子女的方式来想象。例如，图13-1的“上锁”节点，是因为“设陷阱”节点或“有宝物”节点才产生的，或是因为既“设陷阱”又“有宝物”的节点才有的。你应该注意一点，这种因果关系是有概率的，并不确定。例如，“设陷阱”的储物柜不一定会被“上锁”。“设陷阱”确实可能造成“上锁”，但也有可能“设陷阱”而不“上锁”。

你要从概率的观点看待事件间连接的强度。每个节点都有相关联的条件概率表，根据母事件结果的所有可能组合，给出子事件任何结果的概率。就我们的目的而言，我们只考虑独立事件。我们说的是，任何事件，也就是任何变量，都可取一组独立值中的任何一个。这些值假定互斥而且完整。例如，“上锁”节点可取*TRUE*或*FALSE*这组独立值中的任何一个。

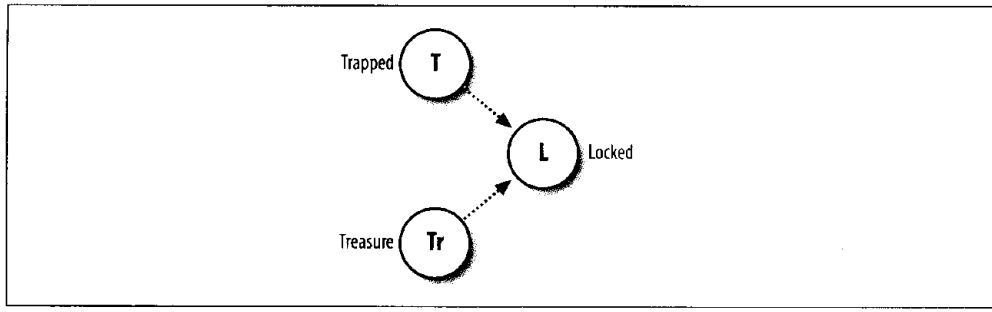


图 13-1：贝叶斯网络实例

我们假设“设陷阱”节点可以是TRUE或FALSE。同时也假设“宝物”节点可以是TRUE或FALSE。如果“上锁”节点能为TRUE或FALSE之一，则我们需要让“上锁”节点有一个条件概率表，根据“设陷阱”节点以及“宝物”节点的值的每种组合，给定“上锁”节点为TRUE的概率，同时，根据“设陷阱”节点以及“宝物”节点的值的每种组合，给定“上锁”节点为FALSE的概率。表 13-1 给出了此例“上锁”节点的条件概率表。

表 13-1：条件概率表示例

		“上锁”的概率	
“设陷阱”的值	“宝物”的值	L = TRUE	L = FALSE
T	T	P(L|T ∩ Tr)	P(~L|T ∩ Tr)
T	F	P(L|T ∩ ~Tr)	P(~L|T ∩ ~Tr)
F	T	P(L|~T ∩ Tr)	P(~L|~T ∩ Tr)
F	F	P(L|~T ∩ ~Tr)	P(~L|~T ∩ ~Tr)

此表中，前两列是“设陷阱”以及“宝物”的值的所有组合。第三列是根据“设陷阱”和“宝物”的值的每种组合，而得 Locked=TRUE 时的概率，而最后一列是根据“设陷阱”和“宝物”的值的每种组合，而得 Locked=FALSE 时的概率。“~”符号表示二元事件（binary event）的共轭对象（conjugate）。如果 $P(T)$是“设陷阱”为 TRUE 的概率，则 $P(\sim T)$就是“设陷阱”为FALSE 的概率。注意，这三个事件都只能取两值之一，故为二元事件，如此一来，“上锁”的条件概率表就是 4 乘 2 组了，如表 13-1 所示。

当这些事件的可能值数目变多时，或者特定子节点的母节点数量增加时，子节点的条件概率表中的组合数量，就呈指数增加。这是在游戏中使用贝叶斯方法的最大障碍之一。不仅难以求出这些条件概率所有可能的值，同时当网络规模增加时，计算上的需求对实时游戏而言也是行不通的。（技术上来说，贝叶斯网络被视为NP 难题（NP-hard），也就是说，有大量节点时，将耗费太多计算资源。）

记住一点，每个子节点都需要一个条件概率表。所谓的根节点（就是没有母节点的节点，此例中，就是“设陷阱”和“宝物”两节点）并没有条件概率表。相反的，根节点有所谓的事前概率表（prior probability table），包含这类事件概率的可能值。这里用“事前”这个术语，指的是尚未利用网络其他地方的新信息对概率做调整前，就决定的根节点的概率。根据新信息更新的概率称为事后概率（posterior probability）。稍后会看到这类计算的实例。

我们这里讨论的复杂度，主要的动机是为了让贝叶斯网络，能在游戏中得到简单而明确的应用。例如，理论上来说，你可以建构一个贝叶斯网络，控制NPC角色的每个方面。你可以让网络里的节点代表追逐或闪躲的决策，同时让其他节点表示左转、右转等。这样做的问题是网络会变得太复杂而难以建成、解决和测试。此外，必要的条件概率表会变得很大，使得你必须求助某种训练手段以计算出适当值，而无法明确指定其值。因此，我们不鼓励这种做法。

如第一章所说，我们建议你用贝叶斯网络做特定的决策问题，把其他AI工作留给其他比较合适的方法来做，第十四章要讨论的神经网络也持这个观点。既然有可靠简单而健全的定性方法，可以引导单位的追逐方向，何苦去用贝叶斯网络呢？我们应该用贝叶斯网络决定要追逐还是要闪躲，再让其他算法处理实际的追逐或闪躲问题。

推论

贝叶斯网络有三种基本类型的推理或推论可供利用。讨论时，我们以图13-2(A)所示的简单网络为参照物。

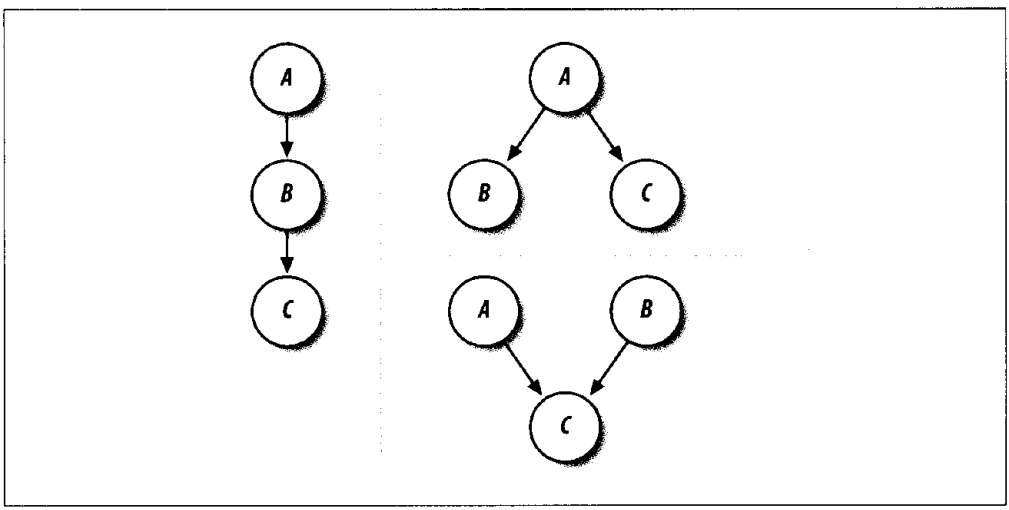


图13-2(A)：简单网络

左边的网络叫因果链（causal chain），此例为三节点因果链。右上方的网络叫做共通成因网络（common cause network），另一常用的名称是简式贝叶斯网络（naive Bayesian network）或贝叶斯分类（Bayesian classifier）。右下角的网络叫共通结果网络（common effect network）。这三种基本类型的推理形式如下：

诊断推理

诊断推理（diagnostic reasoning）可能是使用贝叶斯网络时最常用的推理形式。这种推理配合贝叶斯分类网络，广泛应用于医疗诊断中。例如，参见图13-2(A)右上角的网络，A是疾病，而B和C是病兆。根据病兆，医生就能推论出该疾病的概率。

预测推理

预测推理（predictive reasoning）是根据成因的信息推论出结果。例如，参见图13-2(A)左边的网络，如果我们知道A，而A引起了B，则我们可以推论出B发生的概率。

解释消除

解释消除（explaining away）牵涉到共通结果网络，如图13-2(A)右下角的网络所示。假设节点都是二元节点，也就是真或假。如果我们知道C为真，而且知道A和B引起了C，通过C为真这一事实，就可以推论A和B也是真的概率。然而，假设后来我们知道了B为真，这就表示A发生的概率实际上降低了。这样的贝叶斯网络会有些有趣的特征，也就是独立性和条件式相互依存。

图13-2(A)右下角的网络暗示着事件A和B彼此间互相独立。A和B之间并没有因果关系。然而，如果我们知道C，然后，也知道了A或B，确实会影响A或B的概率。就我们的例子而言，知道C为真而且B也为真，即使A和B是互相独立的事件，也会降低A为真的概率。

现在，分析图13-2(A)左边的网络。此例中，我们看见A会引起B，而B会接着引起C。如果我们知道B的状态，就能推论出C的状态，与A的状态就无关了。如果我们知道B的状态，则A对C事件就没有影响。用贝叶斯网络的术语，称为节点B阻断（block）了A对C的影响力。

贝叶斯网络中另一种形式的独立性称为D分离（d-separation）。如图13-2(B)所示，此时不像前面讨论的某个节点阻断了另一个节点，而是某个节点阻断一群节点。在图13-2(B)中，C引起了D，而D引起了E和F，但A和B引起了C。然而，如果我们知道C的状态，则A和B对D就没有影响，因此，对E和F也没有影响。同样地，如果我们知道D的状态，节点A、B、C就和E、F没有因果关系。试着解决贝叶斯网络时，这些独立性的情况将有所帮助，因为我们可以把网络的各个部分分开处理，以简化某些计算。

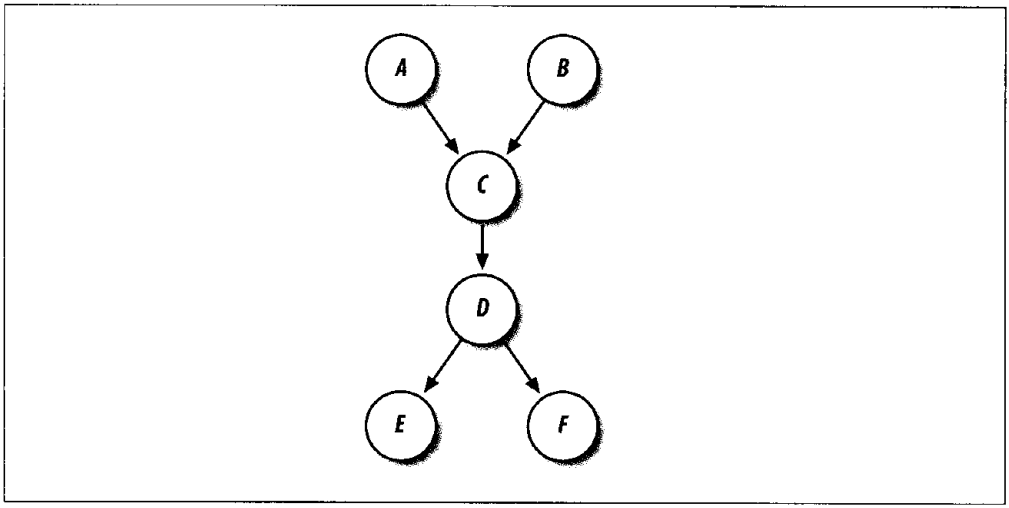


图 13-2(B)：*D* 分离

使用贝叶斯网络时，实际上解题或做推论时，必须运用第十二章谈过的规则，计算概率。我们打算教你怎么在后续简单网络范例中做这些事。不过，我们要指出的是，有些解决复杂贝叶斯网络的通用方法，我们并未提及。这类网络包括知名的信息传递算法（参见本章结尾列出的第二本参考书），以及其他近似推测法。这些方法很多看起来都不适用于实时游戏，因为其计算量都很大。这里我们再次建议你让贝叶斯网络尽可能简单，如果你打算在游戏中使用的话。当然，你可以不听我们的，但让网络保持简单，就能在最合适之处，将之用在特定任务上，而让其他方法能发挥其功效。这会让你的测试和调试简单易行，因为你可以把复杂的 AI 程序和其他 AI 程序隔离开来。

设置陷阱？

假设你正在写一个游戏，里面的NPC会抢劫可能放有玩家宝物和贵重品的储物柜。玩家可以把贵重品放在这些储物柜里储藏，而且也可以选择（如果有此能力）是否替此储物柜设下陷阱，以及是否把储物柜上锁。NPC 可以试着抢劫他们找到的这类储物柜。NPC 可以观察储物柜，确认是否被锁住，但他无法直接看出储物柜是否设有陷阱。NPC 必须决定是否尝试打开此储物柜。如果成功了，就能拿到这些赃物。如果储物柜设有陷阱，他会惹来麻烦，有可能会让他丧命。我们要用简单的贝叶斯网络，再配上一些模糊规则替此 NPC 做决策。

这类贝叶斯网络可能是其中最简单的了。这个网络是两节点链，如图 13-3 所示。

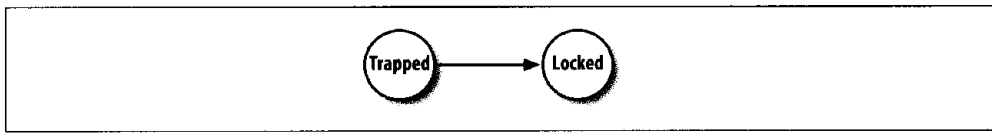


图 13-3：两节点链

每个事件（“设陷阱”和“上锁”）只能是两个独立状态的其中一种：真或假。因此，对每个事件节点而言，我们就有下列概率表，如表 13-2 和表 13-3 所示。

表 13-2：“设陷阱”概率

P(设陷阱)	
真	假
p_T	$(1-p_T)$

表 13-3：“上锁”条件概率

	P(上锁 | 设陷阱)	
设陷阱	真	假
真	p_{Lt}	$(1-p_{Lt})$
假	p_{Lf}	$(1-p_{Lf})$

在表 13-2 中，p_T 是储物柜设有陷阱的概率，而$(1-p_T)$是储物柜没设陷阱的概率。表 13-3 是条件概率，根据储物柜设陷阱的各种可能状态，定出储物柜上锁的条件概率。在表 13-3 中，p_{Lt} 代表的是储物柜设陷阱又上锁的概率，而 p_{Lf} 是储物柜没设陷阱但上锁的概率；$(1-p_{Lt})$代表的是储物柜设陷阱但没上锁的概率，而$(1-p_{Lf})$代表的是储物柜没设陷阱也没上锁的概率。

树图

有时候以树图形式协助解决问题会很有效。上述问题的树图非常简单，如图 13-4 所示。

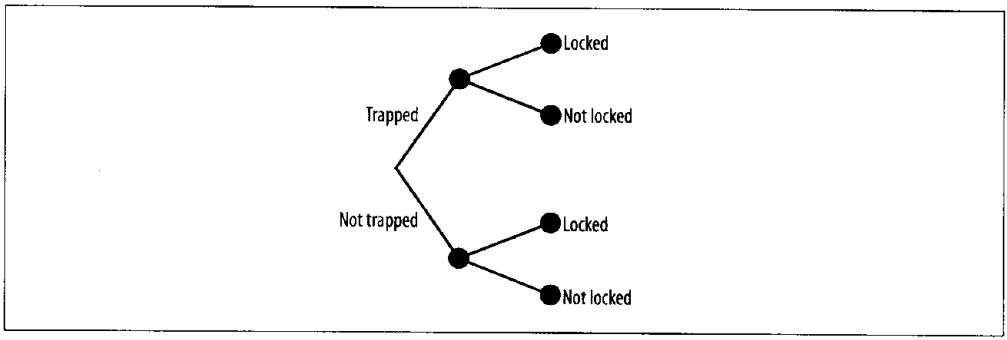


图 13-4：树图

此树图中，显然，储物柜上锁时有两种可能，没上锁时也有两种可能。树图中的每一分支都有对应的概率。这些概率和表 13-2、13-3 列出的相同。此图说明了贝叶斯网络图和树图相比，在可视的因果关系上，有其简洁之处。对事件数目庞大而且每个事件的可能状态不算少的复杂问题而言，这一点很重要。就这些情况而言，树图用来表示每个事件间的关系时，就会变得很笨重。

求概率

我们可以求出所需的概率，也就是表 13-2 和表 13-3 列出的那些概率，做法是在游戏中收集统计资料。例如，每次NPC遇到储物柜并打开时，储物柜设下陷阱的频率，和没有设陷阱的频率可以被更新。NPC可以根据经验而有效习得这些概率。你可以让每个NPC根据自身经验学习，也可以让一群NPC集体学习。你也可以收集“储物柜被锁着，同时也设了陷阱”的频率，以及“储物柜被锁着但没设陷阱”的频率，利用这些统计资料，可求取条件概率。由于我们用的是独立概率，而且每个事件只有两种状态可选，你可以建立一个只有四格的条件概率表格，如前所述。在一个游戏里，任何储物柜都处在图 13-4 所说的四种状态之一，看起来很合理。例如，玩家可以把贵重品放在储物柜中，上锁，但不设陷阱，因为他还没这种技能，或者没素材可做。玩家也可能拥有这项技能和设陷阱所需的素材，不但上锁也设陷阱，等等。因此，我们不能假定储物柜总是设陷阱，或者总是被锁上，等等。

做推论

此例中，我们要用诊断推论。我们的目标是回答问题，当NPC遇到储物柜时，储物柜设有陷阱的概率是多少？如果NPC 没看到储物柜被锁住，则储物柜设有陷阱的概率就是 p_T。然而，如果NPC 观察到储物柜是上锁的状态，我们就能根据此新信息，修改储物柜设有陷阱的概率。我们要用贝叶斯规则做此调整。贝叶斯规则如下所示：

$$P(T|L) = P(L|T)\ P(T)/P(L)$$

$P(T)$代表 Trapped=TRUE 的概率，$P(L)$代表 Locked=TRUE 的概率，$P(L|T)$代表 Trapped=TRUE 时，Locked=TRUE 的概率。这个问题以贝叶斯规则的方式表示如下：

储物柜上锁而设有陷阱的概率 =

储物柜设有陷阱而上锁的概率

× 储物柜设有陷阱的概率

÷ 储物柜上锁的概率

$P(L|T)$是从条件概率表得来的，而 $P(T)$也可从概率表得知。然而，我们必须计算 $P(L)$，也就是储物柜上锁的概率。看到图 13-4 的树图，可知储物柜上锁时可以有两种情况：1)储物柜已设有陷阱，2)储物柜没有设陷阱。我们可以用第十二章的概率规则 4 来求 $P(L)$。就此而言，$P(L)$如下所示：

$$P(L) = P(L|T)\ P(T) + P(L|\sim T)\ P(\sim T)$$

同样地，以贝叶斯规则的方式表示如下：

储物柜上锁的概率 =

储物柜设有陷阱而上锁的概率

× 储物柜设有陷阱的概率

+ 储物柜没设陷阱而上锁的概率

× 储物柜没设陷阱的概率

这里的“~”符号表示共轭对象。例如，如果 $P(T)$代表的是事件“设陷阱” = TRUE 的概率，则 P(~T)代表的就是事件“设陷阱” = FALSE 的概率。

注意，我们用了第十二章的规则 6 来求算“上锁” = TRUE 以及“设陷阱” = TRUE 的概率，也就是 $P(L|T)\ P(T)$。求“上锁” = TRUE 以及“设陷阱” = FALSE 的概率时也可用相同规则，也就是 $P(L|\sim T)\ P(\sim T)$。这也是我们在第十二章的“条件概率”一节所看到的条件概率公式的应用。

现在，我们考虑一些实际数字。假设你的游戏里的某 NPC 具有打开 100 个储物柜的经验，其中有 37 个设有陷阱。在这 37 个设有陷阱的储物柜里，29 个被锁住。在 63 个没设陷阱的储物柜中，有 18 个被锁住。根据这些信息，我们就能计算下列概率：

$$P(T) = 37/100 = 0.37$$

$$P(\sim T) = 63/100 = 0.63$$

$$P(L|T) = 29/37 = 0.78$$

$$P(L|\sim T) = 18/63 = 0.29$$

有了这些概率，我们可知储物柜设有陷阱的概率是37%。现在，如果NPC也注意到该储物柜上锁，也就是“上锁”=TRUE的情况，则储物柜设有陷阱的概率可修改成：

$$P(T|L) = (0.78)(0.37)/\{(0.78)(0.37) + (0.29)(0.63)\} = 0.61$$

因此，观察到该储物柜实际上是锁着的，会加强NPC相信该储物柜设有陷阱。因为，$P(T)$从37%升高到61%。以贝叶斯网络的术语来讲，37%概率是事前概率，而修改后的概率61%是事后概率。

现在，假设NPC观察到储物柜没上锁。就此而言，我们有：

$$P(T|\sim L) = P(\sim L|T)\ P(T)/P(\sim L)$$

可得：

$$P(\sim L|T) = 1 - 0.78 = 0.22$$

$$P(T) = 0.37 \text{（之前）}$$

$$P(\sim L) = 1 - P(L) = 0.53$$

因此，

$$P(T|\sim L) = (0.22)\ (0.37)/(0.53) = 0.15$$

这表示储物柜可能没设陷阱，因为NPC可以看到储物柜没上锁。

现在，有了这些概率，你的NPC如何利用这些概率决定是否开储物柜呢？我们回到NPC看到储物柜锁着，而储物柜设有陷阱的事后概率为0.61的场景。61%是否意味着储物柜设有陷阱的概率高？或者这是一般的概率？或者算是低的概率？我们可以建立一些布尔逻辑的if-then规则来决定，显然地，这是模糊规则最能胜任的工作，如第十章所讨论的。

使用模糊逻辑

我们可以替储物柜设有陷阱的概率建立如图13-5所示的模糊归属函数。

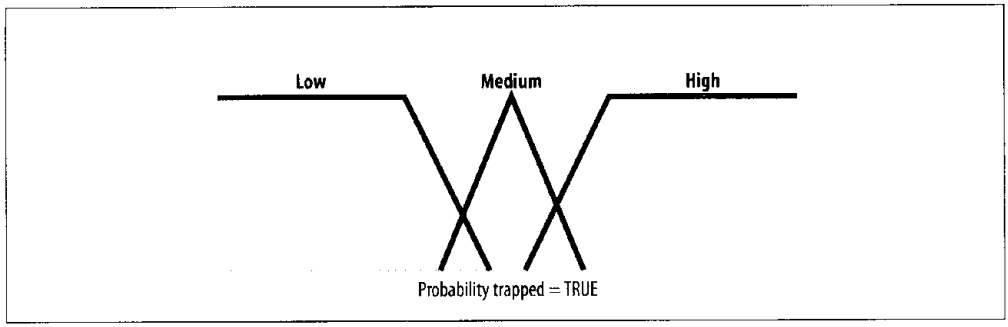


图 13-5：设陷阱归属函数

然后，使用模糊规则决定要做什么。例如，考虑如下条件和行动，做出模糊规则：

- 如果设陷阱的概率高，就不打开储物柜。
- 如果设陷阱的概率低，就打开储物柜。

然而，把其他相关信息也考虑进来就更有效。例如，碰到设陷阱的储物柜时，将对NPC造成某些损伤，因此，根据储物柜设陷阱的推论，而决定是否打开时，也要考虑其健康状况，这种做法似乎很合理。采取这种方法，我们可以设立如下规则：

- 如果设陷阱的概率高而健康值低，就不打开。
- 如果设陷阱的概率低而健康值高，就打开。
- 如果设陷阱的概率中等而健康值高，就打开。
- 如果设陷阱的概率中等而健康值中等，就不打开。

这只是一些你可以设立的规则示例而已。使用这种贝叶斯方法，再搭配模糊规则的优点是，你可以让NPC有能力做出理性决策，而不用去作弊。此外，你赋予了NPC有能力在面对不确定状态时能做决策。此外，利用此方法，NPC可以适应玩家的行动。例如，最初玩家可以锁住储物柜，但不设陷阱。然而，精明的NPC窃贼也许会由于打开储物柜的低风险，因而积极强行抢夺。如果玩家开始替储物柜设陷阱，NPC就可适应，抢劫时不会那么强硬，对是否打开储物柜也会三思。此外，玩家也许会锁住储物柜但不设陷阱，借此蒙骗NPC，但NPC仍然可以据此适应。这样就带来了另一种可能性：如果玩家试着锁住储物柜且设陷阱，但没有在储物柜中置放任何珍贵品，借此欺骗NPC时，那NPC会怎么做？玩家这么做的原因，也许是为了在攻击前先让NPC变弱。因为偷宝物是打开储物柜的动机，如果让NPC可以评估设有陷阱的储物柜，及其包含有宝物的可能性，那就很有趣了。NPC在决定打开储物柜之前，可以把两种因素都考虑进去。下一例会考虑这种情况。

宝物何在？

此例中，我们打算利用前例所建的简单贝叶斯网络。明确地讲，除了让NPC决定打开储物柜之前，先考虑设有陷阱的可能性外，也要考虑储物柜中放有珍贵品的可能性。到头来，我们要替前例的网络新增一个事件节点，变成三节点链。新事件节点就是“宝物”事件，也就是说，如果储物柜中含有宝物，则“宝物”事件为TRUE，如果不含宝物，则“宝物”事件为FALSE。图 13-6 显示了此新网络。

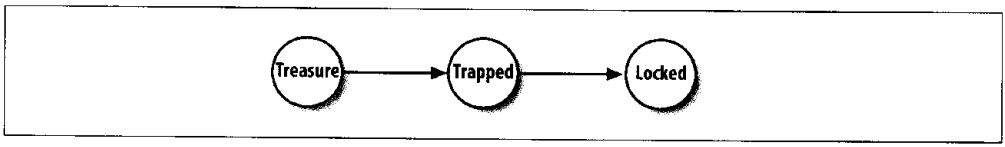


图 13-6：三节点链

就此而言，我们假定储物柜是否上锁，为储物柜是否设有陷阱的征兆，而储物柜是否设有陷阱的状态，就是储物柜是否含有宝物的征兆。

每个事件（“宝物”、“设陷阱”以及“上锁”）都只能有两种独立状态之一：真或假。因此，对每个事件节点而言，我们有下列概率表，如表 13-4、表 13-5 以及表 13-6 所示。

表 13-4：“宝物”概率

P(宝物)	
真	假
p_T	$(1-p_T)$

表 13-5：“设陷阱”条件概率

	P(设陷阱 | 宝物)	
宝物	真	假
真	p_{Tt}	$(1-p_{Tt})$
假	p_{Tf}	$(1-p_{Tf})$

表 13-6：“上锁”条件概率

	P(上锁 | 设陷阱)	
设陷阱	真	假
真	p_{Lt}	$(1-p_{Lt})$
假	p_{Lf}	$(1-p_{Lf})$

注意，就此例而言，储物柜是否设有陷阱的概率表是属于条件概率表，和储物柜含有宝物的状态有关。就“上锁”事件而言，其表基本上类似。

另一种模型

我们要指出的是，图13-6所示的模型是简化模型。储物柜含有宝物就会被锁住，这看起来很合理，不一定要和储物柜是否设有陷阱有关。这表示，“宝物”节点和“上锁”节点之间也有一条因果关系线，如图13-7所示。

就此而言，你也要根据储物柜是否含有宝物的状态，设定相对应的“上锁”条件概率。虽然这种做法很简单，你依旧可以用人工计算解决此网络，但我们还是以先前建的简单模型作为讨论依据。

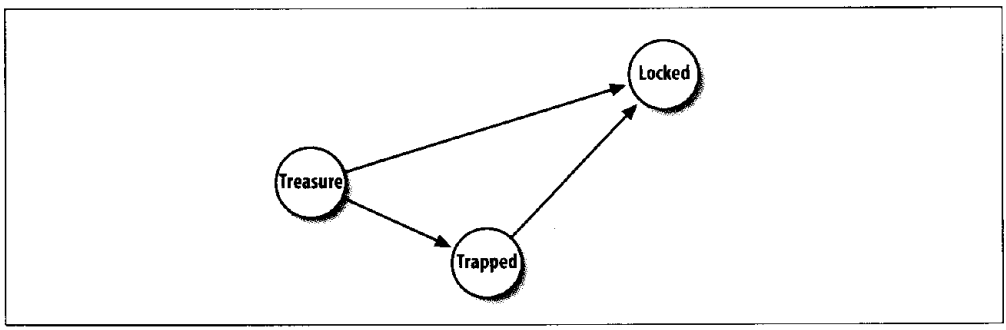


图13-7：另一种模型

做推论

此例接下来的讨论都会依据图13-6所示的模型。为了求出储物柜上锁时设陷阱的概率，我们要以前例的相似手段来计算。然而，现在我们的“设陷阱”事件也有条件概率，也就是储物柜含有宝物、设有陷阱，以及储物柜不含宝物而设有陷阱的概率。了解之后，我们就能应用贝叶斯规则了，如下所示：

$$P(T|L) = P(L|T)\ P(T)/P(L)$$

$P(L|T)$是储物柜设有陷阱而上锁的概率，来自于“上锁”事件的条件概率。然而，这里还没有$P(T)$，但我们可按下列方式计算：

$$P(T) = P(T|Tr)\ P(Tr) + P(T|\sim Tr)\ P(\sim Tr)$$

以文字表示，则为：

储物柜设陷阱的概率 =

储物柜含有宝物而设有陷阱的概率

+ 储物柜不含宝物而设有陷阱的概率

现在，$P(L)$可用下列方式求得：

$$P(L) = P(L|T)\ P(T) + P(L|{\sim}T)\ P(\sim T)$$

我们已算出$P(T)$，而$P(\sim T)$等于1-$P(T)$，所以，现在要做的就是在“上锁”事件的条件概率表中，找出$P(L|T)$和$P(L|\sim T)$，并求出$P(L)$之值。然后，把这些值代入贝叶斯规则，求出$P(T|L)$之值。

为了找出储物柜锁着而含有宝物的概率，我们需要再度应用贝叶斯规则，如下所示：

$$P(Tr|L) = P(L|Tr)\ P(Tr)/P(L)$$

然而，注意，“上锁”和“宝物”之间因“设陷阱”而阻断了。因此，我们可以把$P(L|Tr)$的公式写成这样：

$$P(L|Tr) = P(L|T)\ P(T|Tr) + P(L|{\sim}T)\ P(\sim T|Tr)$$

这是因为，我们的简单模型假定设陷阱的储物柜上了锁。

从上面的步骤，我们已算出$P(L)$，而$P(Tr)$也已知，所以，我们已具备求$P(Tr|L)$的所有信息了。

使用数值范例

我们现在考虑一些实际数字。假设游戏中某NPC有打开100个储物柜的经验，其中50个含有宝物。这50个储物柜里，40个设有陷阱，而这40个设陷阱的储物柜里，有28个上锁。现在，那10个没设陷阱的储物柜中，有3个锁着。并且，另外50个没有宝物的储物柜中，有20个设有陷阱。根据这些信息，我们就能算出下列概率：

$$P(Tr) = 50/100 = 0.5$$

$$P(\sim Tr) = 50/100 = 0.5$$

$$P(T|Tr) = 40/50 = 0.8$$

$$P(T|Tr) = 20/50 = 0.4$$

$$P(\sim T|Tr) = 10/50 = 0.2$$

$$P(\sim T|\sim Tr) = 30/50 = 0.6$$

$$P(L|T) = 28/40 = 0.7$$

$$P(L|\sim T) = 3/10 = 0.3$$

$$P(\sim L|T) = 12/40 = 0.3$$

$$P(\sim L|\sim T) = 7/10 = 0.3$$

假设你的NPC靠近一个储物柜。没有观察此储物柜是否锁着时，NPC会认定该储物柜含有宝物的机会是50%。现在，假设NPC观察到此储物柜锁着。那么，储物柜设有陷阱的概率是多少？含有宝物的概率又是多少？我们可以用先前的公式求算这些概率。就此而言，其值为：

$$P(T) = P(T|Tr)\ P(Tr) + P(T|\sim Tr)\ P(\sim Tr) = (0.8)(0.5) + (0.4)(0.5) = 0.6$$

$$P(L) = P(L|T)\ P(T) + P(L|\sim T)\ P(\sim T) = (0.7)(0.6) + (0.3)(1 - 0.6) = 0.54$$

$$P(T|L) = (0.7)(0.6)/(0.54) = 0.78$$

现在，我们来求 $P(Tr|L)$：

$$P(Tr) = 0.5$$

$$P(L) = 0.54$$

$$P(L|Tr) = P(L|T)\ P(T|Tr) + P(L|\sim T)\ P(\sim T|Tr) = (0.7)(0.8) + (0.3)(0.2) = 0.62$$

$$P(Tr|L) = (0.62)(0.5)/(0.54) = 0.57$$

从此例中，我们可知，观察到储物柜锁着时，把储物柜设有陷阱的概率从60%提高到78%。而且，储物柜含有宝物的概率也从50%提高到57%。

有此信息，你就能用类似此例的模糊逻辑，决定NPC是否开启该储物柜。例如，你可以替储物柜设陷阱以及含有宝物的事件，建立模糊归属函数，然后按照下述方式建立规则：

- 如果设陷阱概率高、含有宝物概率高、建康值低，就不打开。
- 如果设陷阱概率低、含有宝物概率高、建康值不低，就打开。

这只是你可能在规则集合中加入的一些情况。利用此做法，你的NPC会根据好几个因素做决策，即使面对不确定状况也行得通。

空战或陆战

接下来的实例是，假设你正在写战争模拟游戏，玩家可以由陆地、空中或陆空同时对计算机控制的敌军发动攻击。我们的目标是估算玩家选择攻击方式后，赢得战争的机会。然后，用玩家获胜的概率，求出计算机控制的敌军，应该给予哪些攻击目标和防卫点以较高的优先权。例如，假设每次游戏结束，你都让计算机记录赢的一方，是玩家或计算机，同时记录玩家的攻击方式，也就是空中、陆地或陆空双击。然后，你可以不断收集统计数据，以求出玩家采用的攻击模式而获胜的概率。假设你的游戏发现玩家如果采用空战，最有可能赢。也许是玩家发现了计算机的空防弱点，或者运用了其他获胜的战术。如果是这样，计算机就应该给予建造空中防卫力量，以较高优先权。此外，计算机也可以给敌方飞行器生产地点，以较高的攻击目标优先权。这样的做法可以让计算机调整攻防策略，因为它可从过去战史中学习而得知。

模型

此简例的贝叶斯网络如图 13-8 所示。

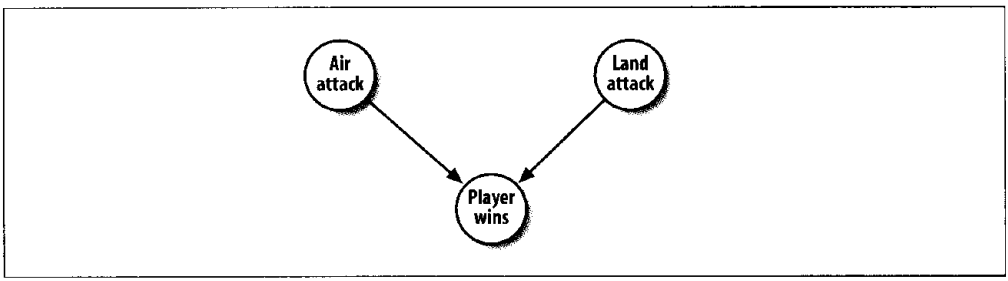


图 13-8：攻击模式网络

注意，玩家赢的结果有两个因素：空战和陆战。我们假定这些事件并非互斥，也就是说，玩家可做陆战、空战或陆空攻击双管齐下。每个事件节点都为两值之一：真或假。例如，“空战”可以是真或假，而“陆战”也可以是真或假，依此类推。

相反的，如果我们只接受陆战或空战的一种，而不允许陆空双管齐下，这个网络就会如图 13-9 所示的那样。

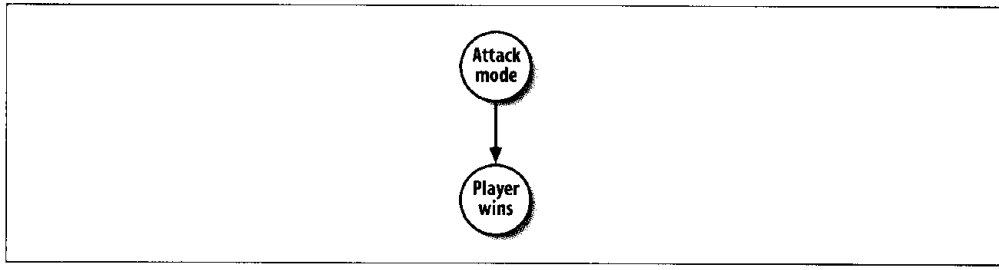


图 13-9：另一种攻击网络

就此而言，“攻击模式”可以取的值是“空战”或“陆战”。在这里，两值是互斥的。下面的讨论，我们会以前者较一般的情况为例。

计算概率

如前所述，我们必须收集一些统计资料，才能估算推论运算所需的概率。随着游戏的进行，我们必须收集如下统计数据：

- 游戏总共进行了 N 场。
- 玩家赢了 N_w 场。
- 玩家发起空战而赢了 N_{pa} 场。
- 玩家发起陆战而赢了 N_{pl} 场。
- 玩家发起空战有 N_a 场。
- 玩家发起陆战有 N_l 场。
- 玩家发起陆空合战而赢了 N_{pla} 场。

我们可以利用这些资料计算下列概率：

$$P(A) = N_a/N$$
$$P(L) = N_l/N$$
$$P(P_w|A \cap \sim L) = N_{pla}/N_w$$
$$P(P_w|A \cap \sim L) = N_{pa}/N_w$$
$$P(P_w|\sim A \cap L) = N_{pl}/N_w$$
$$P(P_w|\sim A \cap \sim L) = 0$$

前两个概率分别是玩家发起空战和陆战的概率。后四个概率是条件概率，也就是玩家在

“空战”和“陆战”事件下所有组合的概率。在这些公式中，*L* 代表的是“陆战”，*A* 代表“空战”，而 *Pw* 代表“玩家赢”。另外，注意一点，我们假设如果玩家没有发起任何攻击，则其赢的概率是 0。

根据这些信息，就能求出玩家在新一场游戏中获胜的概率，方法是总结玩家能获胜的所有方式的概率。就此而言，玩家可以用四种不同方法的任何一种获胜。计算上，每种情况用的都是联合的概率公式，如下所示：

$$P(P_w \cap A \cap L) = P(A)\ P(L)\ P(P_w|A \cap L)$$

玩家赢的可能方式如表 13-7 所示，表中填有相关的概率资料。

表 13-7：玩家赢的条件概率表

空战	陆战	P(Pwi A ∩ L)	P(Pw)	标准化 P(Pw)
$P(A) = N_a/N$	$P(L) = N_l/N$	N_{pla}/N_w	$N_aN_lN_{pla}/N_wN2$	$P(P_w)/\Sigma P(P_w)$
$P(A) = N_a/N$	$P(\sim L) = 1\text{-}P(L)$	N_{pa}/N_w	$(N_aN_{pa}/N_wN)(1\text{-}N_l/N)$	$P(P_w)/\Sigma P(P_w)$
$P(\sim A) = 1\text{-}P(A)$	$P(L) = N_l/N$	N_{pl}/N_w	$(N_lN_{pl}/N_wN)(1\text{-}N_a/N)$	$P(P_w)/\Sigma(P_w)$
$P(\sim A) = 1\text{-}P(A)$	$P(\sim L) = 1\text{-}P(L)$	0	0	0
			$\Sigma P(P_w)$	1.0

上表看起来有点复杂，其实是一目了然。前两列是“空战”和“陆战”可能的状态组合。第一列是“空战”每种状态（真或假）的概率，而第二列是“陆战”每种状态（真或假）的概率。第三列是“空战”和“陆战”状态的每种组合下、“玩家赢”=TRUE 时的条件概率。第四列 *P(Pw)* 代表的是事件“空战”、“陆站”以及“玩家赢”的联合概率。你会发现，这一列的值只是前三列的值相乘的结果，而把乘积放在第四列而已。把第四列的值累加起来，就是玩家获胜的边缘概率（marginal probability）。

看一下第五列。第五列放的是“玩家赢”标准化（normalized）概率。你会发现此列之值就是第四列的每个值除以第四列所有值的总和。这样会让第五列的所有值的总和是 1。（这就好像把向量变成单位向量那样的过程。）第五列里的结果基本上告诉我们，哪种“空战”和“陆战”状态的组合是玩家最可能赢的。

数值实例

我们来考虑一些数字。假设有足够的统计数据可以产生表 13-8 前三列的概率。

表 13-8：玩家赢的条件概率表示例

空战	陆战	P(PwlA ∩ L)	P(Pw)	标准化 P(Pw)
$P(A) = 0.6$	$P(L) = 0.4$	0.167	0.04	0.15
$P(A) = 0.6$	$P(\sim L) = 0.6$	0.5	0.18	0.66
$P(\sim A) = 0.4$	$P(L) = 0.4$	0.33	0.05	0.2
$P(\sim A) = 0.4$	$P(\sim L) = 0.6$	0	0	0
			0.27	1.0

这些数字指出玩家在游戏中的获胜率是27%。（由第四列的总和可知。）此外，观察第五列可知，如果是玩家赢，最可能的攻击模式是空战，不包含陆战。因此，此例中，计算机给予空防系统较高优先权，并把玩家的空战资源视为首要打击目标，才是精明的做法。

现在，假设新一轮游戏开始了，而玩家会以空战攻击，我们想找出此例中获胜的概率。概率统计如表 13-9 所示。

表 13-9：修正后的概率表

空战	陆战	P(PwlA ∩ L)	P(Pw)	标准化 P(Pw)
$P(A) = 1.0$	$P(L) = 0.4$	0.167	0.07	0.18
$P(A) = 1.0$	$P(\sim L) = 0.6$	0.5	0.3	0.82
$P(\sim A) = 0.0$	$P(L) = 0.4$	0.33	0.00	0.0
$P(\sim A) = 0.0$	$P(\sim L) = 0.6$	0	0	0
			0.37	1.0

我们这里的做法是把$P(A)$的值换成1.0，把$P(\sim A)$的值换成0.0，然后重算第四列和第五列的值。就此例而言，我们可得玩家获胜的新边缘概率，反映出我们的假设，也就是玩家会采用空战攻击。这里我们发现玩家获胜的概率提高到37%。此外，我们发现，如果玩家赢，则有82%的机会是发动空战而赢的。这会进一步强化先前的结论，也就是说计算机应该把优先权给予空防武力，并把玩家的空中武力资源视为优先攻击对象。必要时，我们可以继续调整概率以查看结果。例如，我们可以把$P(L)$的值设为1.0，重新算出假定采用陆战时$P(Pw)$的值，等等。

功夫游戏

最后一个实例是假设在做对战游戏，我们想试着预测玩家要出的下一招是什么。这样一

来，我们可以让计算机控制的对手试着预期招式，并据此防卫或反击。为了让事情简单化，我们假设玩家可以出三种招式之一：挥拳、下踢或上踢。此外，要记录这三种招式的组合。根据前两招的招式，计算接下来出招的概率。这样可以让我们找出三种招式的组合。你可以轻易记录更多的招式组合，但会导致更高的记忆需求和计算成本，因为条件概率表会变得比较大。

模型

此例要用的贝叶斯网络如图 13-10 所示。

在此模型中，把这个招式组合事件的第一招叫*A*，第二招叫*B*，而第三招叫*C*。我们假定无论怎么组合，第二招*B*出招的依据是第一招*A*。此外，假定第三招*C*是根据第一招*A*和第二招*B*而决定的。组合可以是任意的，诸如挥拳、挥拳、上踢，或者是下踢、下踢、上踢，等等。

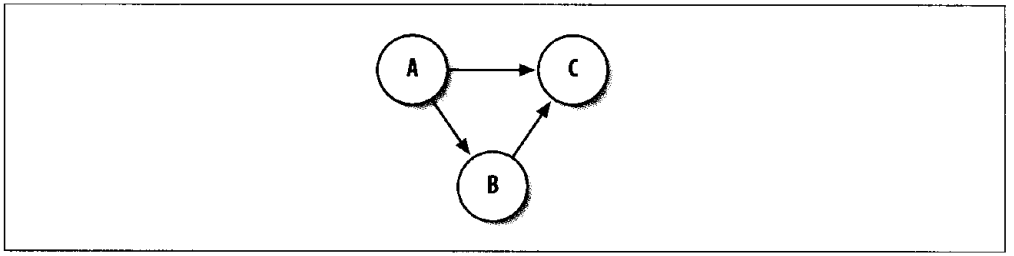


图 13-10：招式网络

计算概率

一般而言，我们必须算出*A*的概率，以及出了*A*招后*B*的条件概率，还有*A*和*B*都出招后，*C*的条件概率。然而，我们会发现*A*和*B*的事前概率和此例无关。因此，只需计算*A*和*B*出招后的每一组合下，*C*的条件概率。由于对*A*招式和*B*招式而言，每一事件都有三种招式状态，因此，我们要记录*A*和*B*的九种可能组合。

我们再度采用频率方式，求这些条件概率。玩家每出一招，根据前两招以及此招的组合，相对应的计数器也会递增（就是指后面程序里的 NAB 数组）。最后，可以得到如表 13-10 所示的条件概率表。

表13-10：出招的条件概率表

		C招的概率		
A招	B招	挥拳	下踢	上踢
挥拳	挥拳	P_{00}	p_{01}	P_{02}
挥拳	下踢	p_{10}	p_{11}	P_{12}
挥拳	上踢	p_{20}	p_{21}	P_{22}
下踢	挥拳	p_{30}	p_{31}	P_{32}
下踢	下踢	p_{40}	p_{41}	P_{42}
下踢	上踢	p_{50}	p_{51}	P_{52}
上踢	挥拳	p_{60}	p_{61}	p_{62}
上踢	下踢	p_{70}	p_{71}	p_{72}
上踢	上踢	p_{80}	p_{81}	p_{82}

上表是A和B招式的每种组合下，招式C为三值之一（挥拳、下踢或上踢）的概率。这里的每个概率都有标示索引值，代表这个矩阵的行和列。稍后出现的范例程序中会用到这些索引值。

要计算这些概率，我们必须记录总出招数。然后，就能算出下列概率：

$$P(A = \text{挥拳}) = N_{\text{挥拳}}/N$$

$$P(B = \text{挥拳} \mid A = \text{挥拳}) = N_{\text{挥拳 挥拳}}/N$$

$$P(B = \text{挥拳} \mid A = \text{下踢}) = N_{\text{下踢 挥拳}}/N$$

$$P(C = \text{挥拳} \mid A = \text{挥拳}, B = \text{挥拳}) = N_{\text{挥拳 挥拳 挥拳}}/NAB$$

这只是一些例子而已，我们用这种方式能算出所有条件概率。在这些等式中，N指的是出招式的次数，而带有下标文字的N，指的是特定招式组合的次数。例如，

$N_{\text{挥拳 挥拳 挥拳}}$指的就是A、B和C同为挥拳的次数。

实际上，你不用储存这些概率，这些频率足够在必要时算出概率。就此例而言，我们要把招式组合的频率储存在9 × 3的矩阵中。这个矩阵代表的是C对应A和B的九种组合时的所有结果。我们也需要一个包含九个元素的数组，以储存A和B的九种组合所发生的次数。

招式预测

现在，要做下一招C的预测，先看最近出的两招A和B，然后再从条件概率表中的组合

找C招。基本而言，就是使用A和B在条件概率矩阵中，找出要考虑的是哪一行，然后，再挑出概率最高的招式作为C招，也就是找出条件概率最高的那一列。

我们写了一小段范例程序来测试此做法。我们有个窗口，里面有三个按钮，对应挥拳、下踢和上踢。用户可以以任何次序，按这些按钮来仿真对打招式。打出这些招式时，刚才提过的条件概率会更新，接着会对下一招式做预测。例13-1是此程序中执行运算的核心函数。

例13-1：招式预测

```
TStrikes ProcessMove(TStrikes move)
{
        int     i, j;

        N++;
        if(move == Prediction) NSuccess++;

        if((AB[0] == Punch) && (AB[1] == Punch)) i = 0;
        if((AB[0] == Punch) && (AB[1] == LowKick)) i = 1;
        if((AB[0] == Punch) && (AB[1] == HighKick)) i = 2;

        if((AB[0] == LowKick) && (AB[1] == Punch)) i = 3;
        if((AB[0] == LowKick) && (AB[1] == LowKick)) i = 4;
        if((AB[0] == LowKick) && (AB[1] == HighKick)) i = 5;

        if((AB[0] == HighKick) && (AB[1] == Punch)) i = 6;
        if((AB[0] == HighKick) && (AB[1] == LowKick)) i = 7;
        if((AB[0] == HighKick) && (AB[1] == HighKick)) i = 8;

        if(move == Punch) j = 0;
        if(move == LowKick) j = 1;
        if(move == HighKick) j = 2;

        NAB[i]++;
        NCAB[i][j]++;

        AB[0] = AB[1];
        AB[1] = move;


        if((AB[0] == Punch) && (AB[1] == Punch)) i = 0;
        if((AB[0] == Punch) && (AB[1] == LowKick)) i = 1;
        if((AB[0] == Punch) && (AB[1] == HighKick)) i = 2;

        if((AB[0] == LowKick) && (AB[1] == Punch)) i = 3;
        if((AB[0] == LowKick) && (AB[1] == LowKick)) i = 4;
        if((AB[0] == LowKick) && (AB[1] == HighKick)) i = 5;

        if((AB[0] == HighKick) && (AB[1] == Punch)) i = 6;
        if((AB[0] == HighKick) && (AB[1] == LowKick)) i = 7;
        if((AB[0] == HighKick) && (AB[1] == HighKick)) i = 8;

        ProbPunch = (double) NCAB[i][0] / (double) NAB[i];
```

```
        ProbLowKick = (double) NCAB[i][1] / (double) NAB[i];
        ProbHighKick = (double) NCAB[i][2] / (double) NAB[i];

        if((ProbPunch > ProbLowKick) &&
           (ProbPunch > ProbHighKick))
             return Punch;
        if((ProbLowKick > ProbPunch) &&
           (ProbLowKick > ProbHighKick))
             return LowKick;
        if((ProbHighKick > ProbPunch) &&
           (ProbHighKick > ProbLowKick))
             return HighKick;

        return (TStrikes) rand() % 3; // 最后的手段
}
```

这个函数以一个名为move、类型为TStrikes的变量作为单一参数。TStrikes只是枚举类型，如例13-2所示。

例13-2：Tstrikes枚举类型

```
enum TStrikes {Punch, LowKick, HighKick};
```

这个move参数代表的是玩家最近打出的招式。ProcessMove()也会返回一个类型为TStrikes的值，代表的是预测玩家出的下一招。

记录

进入ProcessMove()之后，全局变量N会递增。N是玩家打出招式的总数。此外，如果最近打出的招式move等于上次预测的招式Prediction，则NSuccess成功预测数也会递增。

ProcessMove()下面的任务是，根据最近move招式以及储存在二维数组AB中的前两招，更新条件概率表。AB的定义如例13-3所示。

例13-3：全局变量

```
int             NAB[9];
int             NCAB[9][3];
TStrikes        AB[2];
double          ProbPunch;
double          ProbLowKick;
double          ProbHighKick;
TStrikes        Prediction;
TStrikes        RandomPrediction;

int             N;
int             NSuccess;
```

由于条件概率表储存在9 × 3的NCAB数组中，我们要根据最新的一招以及前两招，在此表中找出适当的行和列所对应的项目，以便递增。把NACB称为条件概率表并不完全正确。我们储存的并非概率，而是频率，必要时再用这些频率计算概率。

无论如何，ProcessMove()中的第一块九个if语句，会检查AB数组中储存的所有可能招式组合，求出需更新NCAB的那一行。下一块三个if语句求出需更新NCAB矩阵的那一列。现在，可以根据刚才求出的行和列，更新NCAB元素。同时也要根据刚才求出的行更新NAB元素。NAB储存的是A和B任意组合的次数。

下一步是把AB数组中的元素移位。要把B位置（数组索引值为1）的招式移到A位置（数组索引值为0），把先前储存在A位置的值删除。然后，把最近招式move放入B位置，以便于我们做预测，以及运行下一轮时让此函数能予以使用。

做预测

此时，可以对下一招做预测了。下一块九个if语句是根据AB数组中储存的新招式组合，找出我们要考虑NCAB矩阵中的那一行。利用找出来的那一行，来寻找NCAB矩阵里，三种可能招式的每一种频率。记住一点，这些频率是有条件的，都是根据AB数组储存的招式组合而定。

下一步是计算实际的概率ProbPunch、ProbLowKick以及ProbHighKick，做法是把每一招式的频率除以AB数组储存的招式组合中已被打出的总数。最后，这个函数会以概率最高的那一招，作为下一招的预测值。如果所有概率都相等，不太可能，但我们就直接返回随机猜测的值。技术上而言，应该再多写一点检查程序，处理三招中有两招概率相等，但又高于第三招的情况。就此例而言，可以在概率相等的两招中随机猜测。

经过重复测试，我们发现运用此方法时，计算机预测下一招的成功率在60%～80%之间。如果计算机每次收到招式时都只做随机猜测，则成功率只有30%。此外，如果玩家碰巧发现某种最喜爱的组合，而且时常使用，则计算机很快就会摸清底细，则成功率会提高。当玩家发现计算机防御其他招式组合时越来越顺手，玩家就会调整其招式组合，那么成功率一开始会降低，但是，当玩家持续使用新的招式组合时，成功率又会回升。这样的循环会不断持续下去，迫使玩家不断改变技巧，以应对计算机对手的适应力。

其他信息

本章介绍了贝叶斯技术，教你怎么在游戏软件AI中使用简单贝叶斯模型，在不确定状态下做决策，同时利用概率达到某种程度的适应能力，希望我们能达到预定的目标。实际

上，我们只谈了这些很有效的手段的九牛一毛，还有很多信息值得你去研究，你应该下定决心去多学一点这些技术。为了方便你学习，我们准备了一些自认为有用的参考书，如下所示：

- 《Bayesian Inference and Decision》，2nd，Robert Winkler（Probabilistic Publishing 出版）。
- 《Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference》，Judea Pearl（Morgan Kaufmann Publishers, Inc. 出版）。
- 《Bayesian Artificial Intelligence》，Kevin Korb 和 Ann Nicholson（Chapman & Hall/CRC 出版）。

第一本参考书特别好，以白话文形式全面讨论了概率、贝叶斯推论以及不确定状态下的决策。如果你决定追寻复杂而难以用简单计算解决的贝叶斯模型，一定要看第二本参考书，其介绍的方法，可用来解决一般推论的复杂贝叶斯网络。

因特网上有数不胜数的贝叶斯资源，以下是我们觉得有用的资源链接：

- *http://bndev.sourceforge.net/*
- *http://www.niedermayer.ca/papers/bayesian/*
- *http://www.cs.ualberta.ca/~greiner/bn.html*
- *http://www.research.microsoft.com/research/dtg/*

第一个链接指向“Bayesian Network Tools in Java”网站，里面有做贝叶斯分析的工具，以及 Java 开源码工具箱的信息（有关开源码的定义，可参考 *http://www.opensource.org*）。第二个链接，有贝叶斯网络的简介，不专门针对游戏方面的应用。该网页上也有链接至其他谈论贝叶斯网络的因特网资源。第三个链接，包含几份教学文件以及很多其他资源的网络链接。第四个链接是微软的“Decision Theory and Adaptive Systems”搜寻网页，包含许多不确定性和决策辅助科技的资源，包括贝叶斯网络在内（还有其他的资源）。除了这四个网址外，因特网上还可以找到很多其他资源。你只需用“贝叶斯网络”（Bayesian networks）作为关键词搜寻，就能找到好几百条链接。

第十四章

神经网络

我们的大脑由数十亿个神经元（即神经细胞）组成，每个神经元都和其他数千个神经元连接，形成了拥有强大运算能力的复杂网络。人工神经网络试图模仿大脑的运算能力，尽管其规模要小很多；下面的人工神经网络也会简称为神经网络或网络。

可以这么说，信息是通过轴突和树突在神经元之间传递。轴突把活化神经元的电位或活化电位，传到其他相连的神经元。活化电位经由树突的接收器而获得。突触间隙（synaptic gap）是化学反应发生之处，会对神经元收到的活化电位予以刺激或压抑。图14-1是神经元的图示。

成人大脑约有10^{11}个神经元，而每个神经元可经由突触，接收约10^4个其他神经元的输入电位。如果所有输入电位的结合效果足够强，神经元就会活化，将其活化电位传给其他神经元。

相比之下，我们在游戏中所用的人工神经网络相当简单。就很多应用情况而言，人工神经网络仅由十几个神经元组成而已，和大脑相比非常简单。某些特殊的应用所用到的网络也许由数千个神经元组成，即使如此，和大脑相比还是很简单。此时，我们并不期望利用人工神经网络，可以接近人类大脑的运算能力，然而，就特定问题而言，我们的简单网络就相当够用了。

这是神经网络的生物学观点。有时，不仅从生物学观点来认识神经网络会比较有用。明确地讲，你可以把神经网络想象成数学函数那样。网络的输入值代表的是独立变量，而输出值代表的是相关变量。网络本身就是函数，可以根据输入值，给出独特的一组输出值。就此而言，这个函数很难写成方程式，但所幸我们也不必写出来。再者，这个函数是高度非线性函数。稍后我们会回到这种想法。

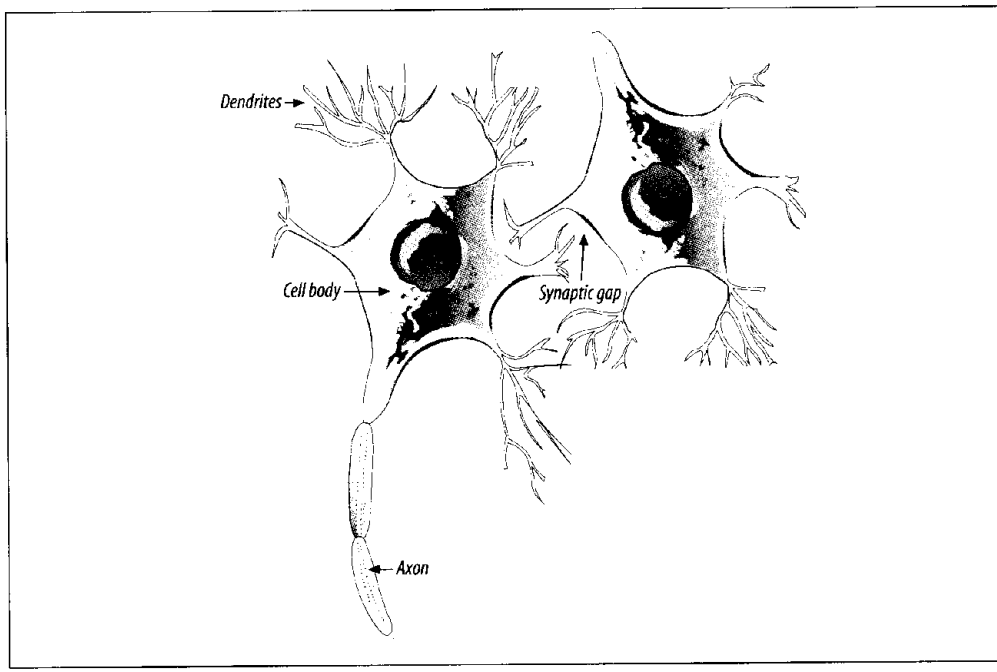


图 14-1：神经元

就游戏而言，和比较传统的AI技术相比，神经网络提供了某些关键的优点。首先，使用神经网络，可以让游戏开发人员简化复杂状态机或规则系统的程序编写，做法是把某些关键的决策流程，交给一个或数个已训练过的神经网络。其次，神经网络有个可能性，可以让游戏进行中，游戏的AI也随之适应。这是相当吸引人的地方，本书写作时，在游戏AI社群里，这也是非常受人关注的话题。

尽管有这些优点，神经网络在视频游戏中还没有得到广泛使用。游戏开发人员在好几个知名游戏中，都用到了神经网络，但总体而言，神经网络在游戏中的应用还是有限的。这可能是由几个因素导致的，下面我们谈两个关键因素。

首先，神经网络处理高度非线性问题很有用，这些问题你很难用传统方法轻松解决。有时候神经网络就这样被贴上标签，而且运算过程难以控制，也让所谓的测试人员不知所措。其次，神经网络会产生什么输出结果，通常也难以预测，尤其是编制的网络会在游戏中学习或适应时。例如，和有限状态机的测试和调试相比，这两个因素让神经网络的测试和调试异常困难。

再者，早期某些尝试在游戏中应用神经网络的做法是整个AI系统都用，也就是说，组成巨型神经网络，处理多数游戏生物或角色常碰见的AI任务。把神经网络当作整个AI系

统，或者可以这么说，整个大脑。我们并不鼓励用这种做法，因为牵涉到可预测性、测试和调试，问题只会更恶化。相反的，就像我们的大脑有很多区域，专职于特定任务，我们也建议你用这种方式，让神经网络处理特定游戏软件 AI 部分，使其成为也用传统 AI 技巧的整合性 AI 系统的一部分。如此一来，AI 系统大体而言都是可预测的，而困难的 AI 任务、或者你想运用学习和适应优点来应对的任务，都可用经过严格训练的特定神经网络专职于该项任务。

AI 社群使用很多不同种类的神经网络来解决各种问题，从金融到工程问题等各方面都有。神经网络通常和其他技术结合在一起，比如模糊系统、遗传算法以及概率方法，这里不再一一列举。这个主题太广，难以用一章的篇幅道尽来龙去脉，所以我们要把焦点锁定在神经网络特别有用的那一部分。我们要把注意力放在一种名叫多层前馈网络（multilayer, feed-forward network）的网络上。这种网络相当有用，能够处理多种问题。谈这种网络的细节前，我们先从大方向探讨如何在游戏中应用神经网络。

控制

在机器人应用领域里，神经网络时常用作神经控制器。就此而言，机器人的感应系统，会给神经控制器提供相关的输入数据，而神经控制器的输出结果（由一个或多个输出节点组成），会把适当的响应传给机器人的发动机控制系统。例如，机器人坦克的神经控制器，也许获得了三个输入数据，分别指出机器人前方或两侧是否感应到障碍物。（每个感应到的障碍物的距离也可以作为输入数据。）神经控制器可以有两个输出结果，控制其左右两条履带的移动。其中一个输出节点，可以把左侧履带设为往前或往后，而另一输出节点，可以把右侧履带设为向前或向后。输出结果结合在一起，就是让机器人往前、往后、左转或右转。这种神经网络看起来就像图 14-2 那样。

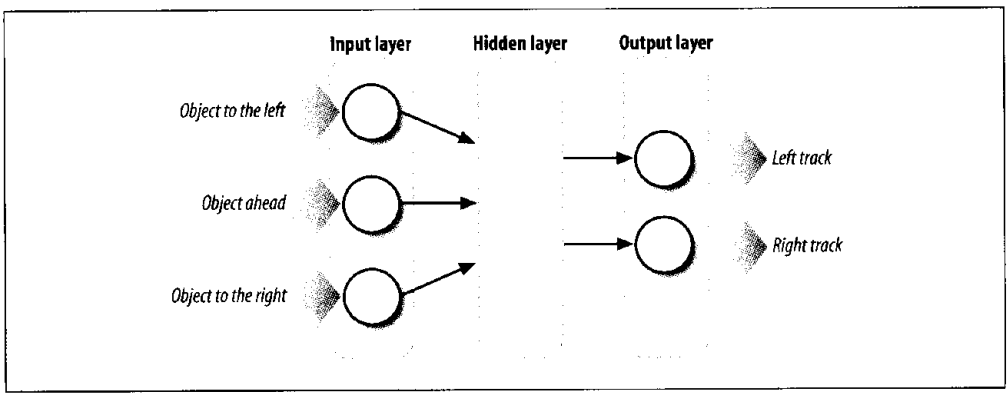


图 14-2：机器人控制神经网络示例

游戏中也有类似的场景。事实上，你可以在游戏中，设计计算机控制的半履带机械化单位。你也可以用神经网络，控制宇宙飞船或飞行器的飞行。无论哪一种做法，你都可以用一个或多个输入神经元，以及一个或多个输出神经元，控制单位的动力、车轮、履带或你模拟的任何移动方式。

威胁评估

举另一个例子，假设你在写策略模拟类游戏，玩家必须运用技术，训练某些单位以抵挡或攻击计算机控制的基地。假设你决定用神经网络，在游戏进行中的任何时刻，让计算机控制的军队能用某种方式，预测玩家的威胁类型。可能的神经网络之一如图14-3所示。

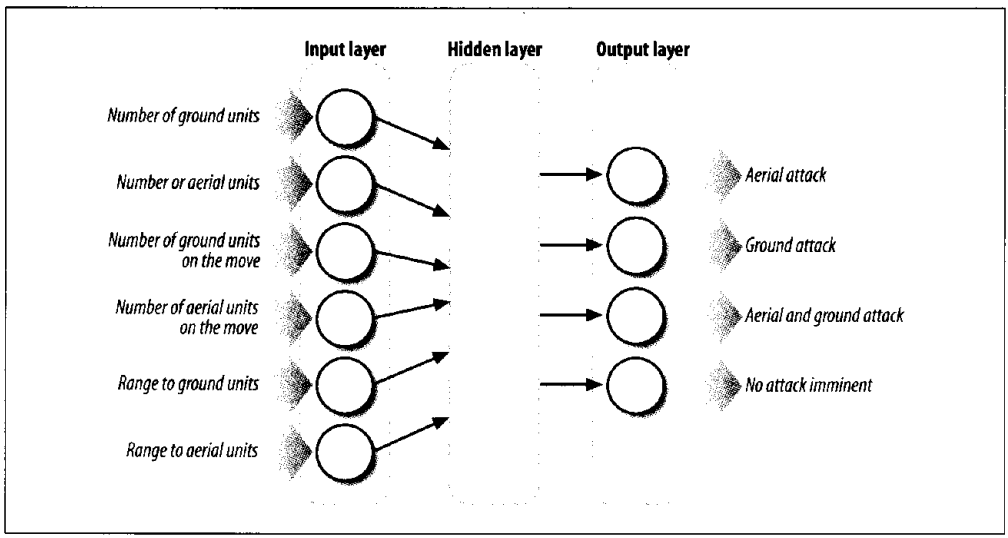


图 14-3：评估威力等级的神经网络示例

这个网络的输入数据包括敌方（玩家）地面单位数量、敌方空中单位数量、地面单位是否移动的迹象、空中单位是否移动的迹象、地面单位的距离以及空中单位的距离。输出结果由神经元组成，指出四种威胁的其中一种，包括空中威胁、地面威胁、陆空威胁或者无威胁。利用游戏中适当数据以及评估网络成效的方法（稍后会谈训练），你可以用这种网络预测近期是否有某种攻击。一旦评估出威胁，计算机就能采取适当行动，包括地面或空中武力的部署、在海岸边建立防线，或者让步兵待命，不设威胁时，就保持平常状态。

这种方法在游戏中，需要进行网络训练和验证，但可以根据玩家的玩法自行调整。此外，如果你用规则系统，或有限状态机类型的结构做这种工作，就可以减轻计算所有可能的场景和阈值的任务。

攻击或逃离

举最后一个例子，假设你有一个在线角色扮演游戏，决定用神经网络控制游戏中某些生物的行为。现在，假设你打算用神经网络控制生物的决策过程，也就是说，该生物是否攻击、闪躲或群聚，取决于该生物附近是否有敌人（玩家）。图 14-4 所示的就是这样的神经网络。注意到，你只用这个网络决定是否攻击、闪躲或群聚，至于所需的行动，则是用其他游戏逻辑执行，比如前面讨论过的追逐和闪躲技术。

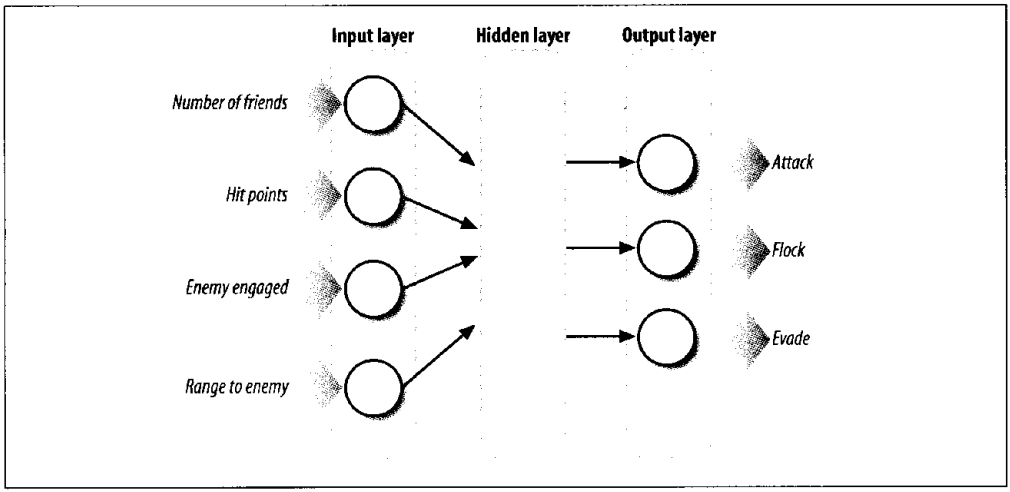


图 14-4：攻击决策神经网络示例

此例有四种输入数据：有多少类似生物在当前正在做决策生物的附近（作为该生物采取单独行动或群聚行动的指标）、该生物被击中的次数或健康状态值、敌人是否和其他生物正在交战中、以及敌人的距离。

我们可以加一些输入数据，比如敌人的等级、敌人是法师或斗士等，借此让这个例子再精细一点。生物的攻击策略和防卫，如果都比较适合某种等级或类型的敌人，做这种考虑是很重要的。你可以通过“作弊”得知敌人的等级，或者更好的做法是，利用另一个神经网络或贝叶斯分析法，预测敌人的等级，在整个过程中增加一些不确定性。

分析神经网络

本节我们打算详谈三层前馈神经网络，了解每个部分的功能，为何重要，以及如何运行。这里的目标是清楚而明确地揭开神经网络的奥秘。我们将采取比较实用的方法进行说明，把某些学术性的题材，留给同主题的其他书籍说明；本章将提及几本这样的参考书。

结构

本章的焦点是三层前馈神经网络。图 14-5 显示了这种网络的基本结构。

三层网络由输入层、隐匿层以及输出层（input、hidden、output layer）组成。每一层的神经元数目不限。输入层的每个神经元，都和隐匿层的每个神经元相连接。而且，隐匿层的每个神经元都和输出层的每个神经元相连接。此外，除了输入层之外，每个神经元还有一个额外的输入值叫偏差项（bias）。图 14-5 中的数字代表三层中的每个节点。当我们写公式计算每个神经元的值时，就会用到此数字编号系统。

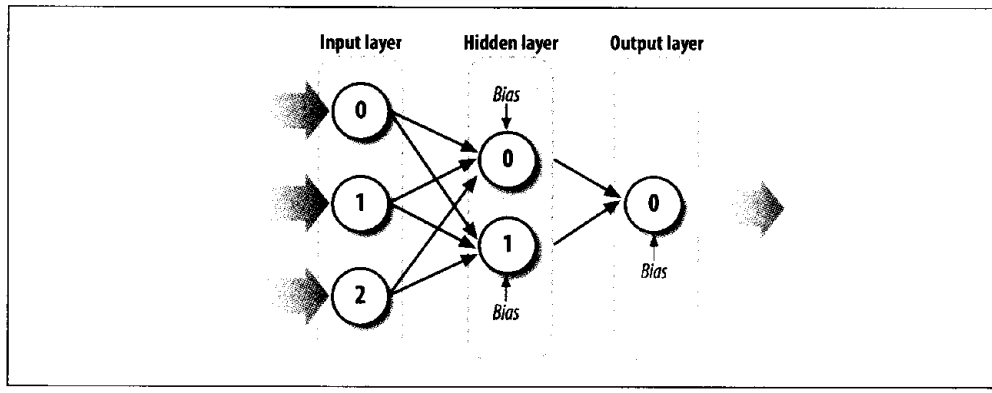


图 14-5：三层前馈神经网络

计算网络的输出值，先从提供给每个输入层神经元的输入值开始。接着，权衡这些输入值后，传给隐匿层的神经元。接着重复此过程，从隐匿层传给输出层，也就是说，隐匿层神经元的输出值，变成输出层的输入值。从输入层到隐匿层再到输出层的过程，就是前馈流程。下列各节我们要详谈这种网络的各个部分。

输入层

神经网络的输入数据显然非常重要，没有输入数据，神经网络就不可能进行运算。显然，我们需要输入数据，但你应该如何选择输入数据？需要提供多少输入数据？输入数据的形式是什么？

输入数据：什么数据？多少数据？

把什么数据当作输入数据应根据问题而定。要看所要解决的问题是什么，而且选择什么样的游戏参数、数据和环境特征值，对要解决的任务而言都有很大影响。例如，假设你正在设计神经网络，替角色扮演游戏中的玩家角色分类别，让计算机控制的生物可以决

定是否和玩家交战。你要考虑的输入数据，可能包括玩家的装备数据、拿什么武器（如果有的话）以及任何目击的行为，例如，是否刚下咒语。

如果你把输入神经元的数量控制在最少，训练神经网络的工作（稍后会谈）就会大为简化。然而，在某些情况下，要选择的输入数据是什么，不见得都能一目了然。就此而言，通用的做法就是把你认为可能重要的数据，当作输入数据，然后，让神经网络自行弄清楚哪些才是重要的。面对所需输出的结果，神经网络擅长排列输入数据间的相对重要性。然而，记住一点，你放进来的输入数据愈多，准备训练网络的数据也就愈多，游戏中要做的计算就愈多。

通常，你可以合并或转换重要信息，改成某种比较简洁的形式，借此减少输入数据的数量。举一个简单实例，假设你试着用神经网络，控制游戏中宇宙飞船登陆星球事件。宇宙飞船的质量是变量，除了其他因素之外，该星球的重力加速度显然也是重要因素，应该将之视为输入数据，提供给神经网络。事实上，你可以替每个参数都建立一个输入神经元，一个给质量，另一个给重力加速度。然而，这种做法会迫使神经网络增加额外的工作，以弄清楚宇宙飞船质量和重力加速度之间的关系。比较好的输入方式是，把这两个重要参数合并成一个神经元，也就是以宇宙飞船的重力，作为此神经元的输入数据，即质量和重力加速度的乘积。当然，除了这个神经元之外，还有其他输入神经元，例如，还需要飞行高度以及速率的输入数据。

输入数据：什么形式？

你可以用各种形式的数据，作为神经网络的输入数据。对游戏而言，这类输入数据通常由三种类型组成：布尔、枚举以及连续类型。神经网络以实数为对象，所以，无论你有什么数据类型，都必须先转换成适当的实数，才能作为输入数据。

考虑图 14-4 的实例，“敌方是否交战”的输入数据显然是布尔类型 —— 如果敌方已在交战，其值为`true`，反之则为`false`。然而，我们不能把`true`或`false`传给神经网络的输入节点。相反的，我们要以 1.0 表示`true`，而以 0.0 表示`false`。

有时候，输入数据也许是枚举类型。例如，假设你设计的网络是为了对敌人进行分类，而其中一个考虑因素是所使用的武器种类。武器种类选项可能像是短剑、劣质剑、长剑、刀、十字弓、短弓或长弓。这里先后的次序不重要，而我们假定这些可能性是互斥的。一般而言，在神经网络里处理这种数据，用的是所谓的“n 分之一”（one-of-n）的编码方法。基本而言，你替每种可能性都建立一个输入值，然后根据每个特定的可能性是否为`true`，把输入值设为 1.0 或 0.0。例如，如果敌人持有刀，则输入向量为{0, 0, 0, 1, 0, 0, 0}，1 就是刀这个输入节点的值，而 0 是其他可能节点的值。

事实上，你的资料是浮点数或整数的情况很常见。无论是哪种情况，只要是在某种实用的上下限之内，这类数据类型通常可以带任何值。你可以直接把这些值，输入到神经网络内（游戏开发人员时常这么做）。

然而，这将造成某些问题。如果输入值前后在大小上的变化很大，则神经网络可能会给那些比较大的输入值较大的权重。例如，如果某个输入值的范围在0～20之间，而另一输入值的范围在0～20,000之间，后者可能会把前者的影响力抹掉。因此，就此而言，把这类输入数据的范围调整成能够伸缩的，使其前后输入时能有比较的余地，这一点是很重要的。一般而言，你可以用0～100的百分比或者0～1之间的值，控制这类数据的伸缩范围。以这种方式调控伸缩范围，会根据不同的输入资料，使得游戏世界大小不等。然而，调控伸缩范围时要谨慎。必须确定用于训练网络数据的伸缩范围，和网络实际反映游戏世界时的数据是一致的。例如，距离输入值的伸缩范围由屏幕宽度控制，以此作为训练数据，则网络实际在游戏中运行时，也要用相同的屏幕宽度，调控输入数据的伸缩范围。

权重

人工神经网络里的权重，就好像生物神经网络里的突触连接。权重会影响特定输入数据的强度，可以抑制或激活。实际上可说是权重在决定神经网络的行为。再者，求出这些权重之值的任务，是训练神经网络或让神经网络演化的主题。

神经元和另一神经元的连接都有相匹配的权重，如图14-5所示。于是，神经元的输入值，就是每个输入值和该神经元相连的权重，乘以自身的输入值的总和再加上偏差项（后面会讨论）。最后的结果就是神经元的总输入值。下列方程式是特定神经元j的总输入值，就是从i个神经元的一组输入值计算而得来的。

$$n_j = \sum n_j w_{ij} + b_j w_j$$

参见图14-5，你可以发现神经元的每项输入值，都会乘以那两个神经元之间连接的权重，外加偏差项。我们看一个简单实例（后面会看这些计算的源代码）。

假设你想计算图14-5隐匿层中第0个神经元的总输入值。利用上述公式，我们可以得到隐匿层第0个神经元的总输入值的公式：

$$n^h{}_0 = n^i{}_0 w_{00} + n^i_1 w_{10} + n^i{}_2 w_{20} + b^h{}_0 w_{b0}$$

在此公式中，n代表神经元之值。就输入神经元而言，这些就是输入值；就隐匿神经元

而言，就是指总输入值。上标 h 和 i 代表的是神经元所属的层次——h 指隐匿层，而 i 指输入层。下标指的是每一层的节点。

注意到，某特定神经元的总输入值只是其他神经元加权后输入值的线性组合。如果是这样，神经网络怎么像我们前面提到的那样，能逼近高度非线性函数呢？关键就在于总输入值如何转换成某神经元的输出值。明确地讲，活化函数（activation function，或译为激发函数）把总输入值以非线性方式对应到相应的输出值。

活化函数

活化函数可以接收神经元的总输入值，予以处理，再产生神经元的输出结果。活化函数应该是非线性的（除了稍后会谈到的一种特例之外）。如果活化函数不为非线性，则神经网络就会降低为线性函数的线性组合，其设计就无法逼近非线性函数及非线性关系。

最常用的活化函数是 logistic（罗吉斯）函数，或称为 S 型函数（sigmoid function）。图 14-6 就是 S 型函数。

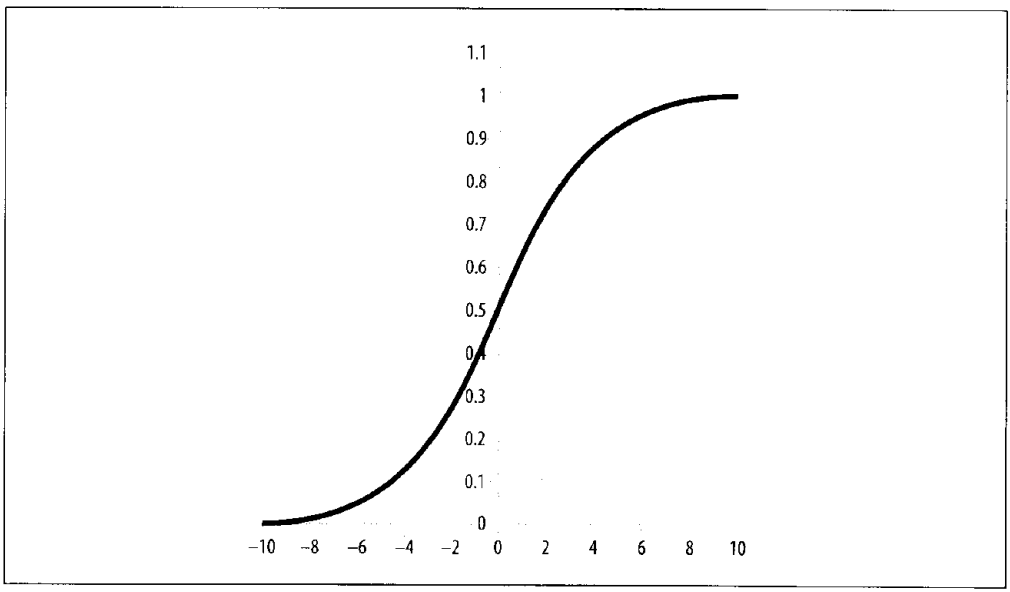


图 14-6：罗吉斯活化函数

logistic 函数的公式如下：

$$f(x)=\frac{1}{1+e^{-x}}$$

有时候，这个函数会写成：

$$f(x)=\frac{1}{1+e^{-x/c}}$$

就此而言，c 用来改变函数的形状，也就是沿着水平轴伸缩函数。

注意到，输入值是水平轴，对所有x值而言，此函数的输出值都介于0～1之间。实际上，可用的范围好像是0.1～0.9。0.1左右的值，表示神经元没有活化，而0.9左右的值表示神经元被活化。注意一点，无论x值多大（正或负），logistic函数永远不会达到1.0或0.0，这一点很重要；logistic函数只能渐近这两个值。训练神经网络时，这要牢记在心。如果你试着训练网络，起始值对特定输出神经元给予的值是1，则永远也得不到结果。比较合理的值是0.9，锁定此值，会让训练速度加快许多。如果试着训练网络使其输出的值是0，结果也一样，此时应该用0.1左右的值。

还有其他活化函数可用。图14-7和图14-8是另外两个为人所熟知的活化函数：阶跃函数（step function）以及双曲正切函数（hyperbolic tangent function）。

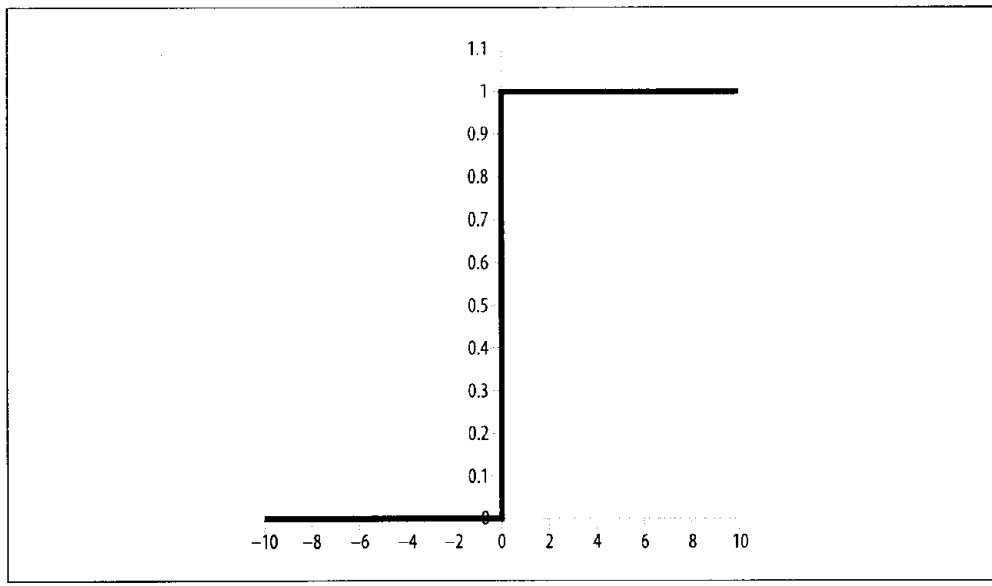


图14-7：阶跃活化函数

阶跃函数的公式如下：

$$f(x)=\begin{cases}0;x\le 0\\1;x>0\end{cases}$$

阶跃函数用于早期神经网络发展时期，但由于缺少导数（无法微分），难以训练网络。训练网络时需要导数，而 logistic 函数正好有易于微分的导数，我们很快就会谈到。

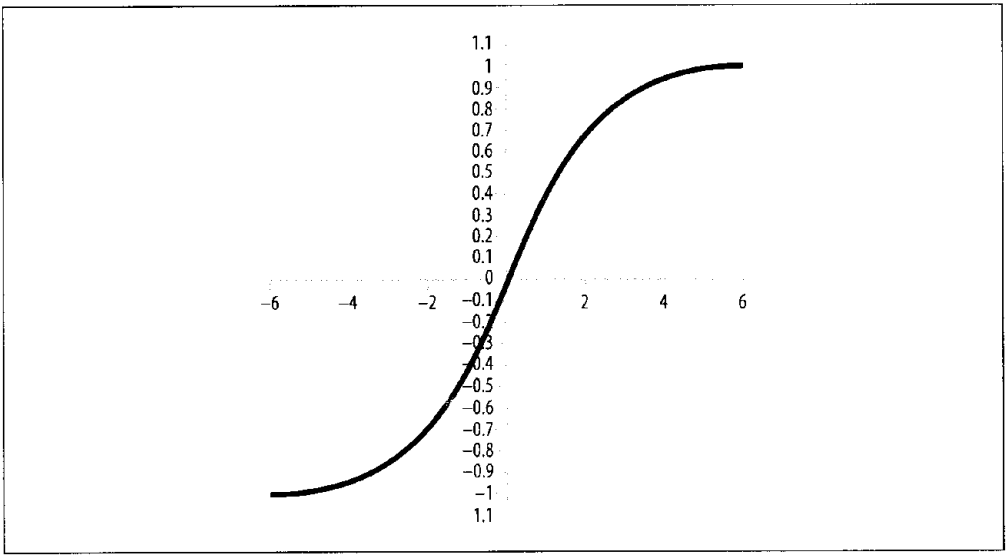


图 14-8：双曲正切活化函数

双曲正切函数的公式如下：

$$f(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

双曲正切函数偶尔会用到，目的是加快训练。此外，还有其他活化函数用于神经网络，用作各式各样的用途，然而，我们不会谈那么远。一般而言，logistic 函数似乎是最广为使用的，而且适用很广的应用范围。

图 14-9 显示的是有时会用到的另一个活化函数，称为线性活化函数（linear activation function）。

线性活化函数的公式就是：

$$f(x)=x$$

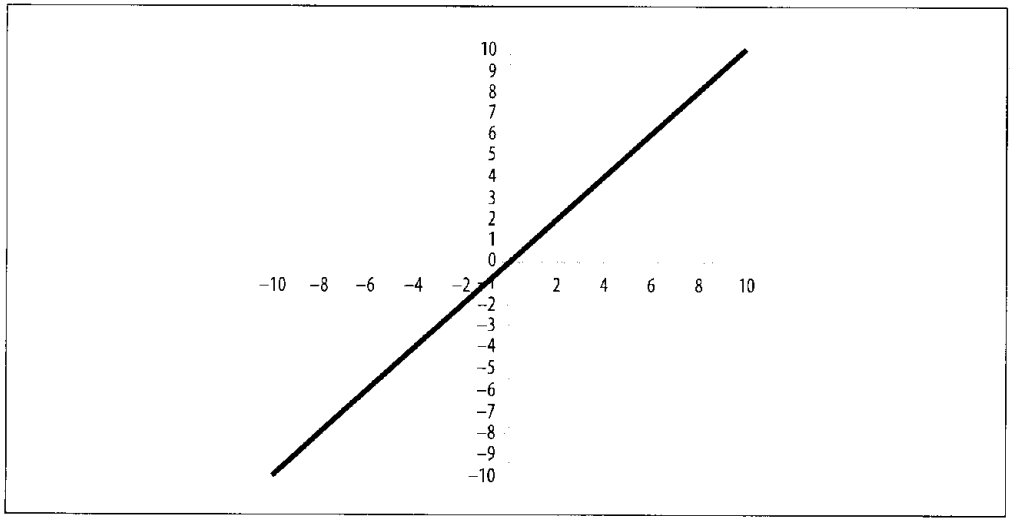


图 14-9：线性活化函数

这就是说，神经元的输出值就直接充当总输入值，即所有与之相连的输入神经元的加权输入值的总和，再加上偏差项。

线性活化函数偶尔会当作活化函数，作为输出神经元的值。注意到，如果网络不打算降低为线性函数的线性组合，则隐匿神经元，就必须使用非线性活化函数。当你不想把输出值限定在0和1之间的区间时，采用这样的线性输出神经元就有用了；但就此而言，仍然可以使用 logistic 输出活化函数，只要把输出值，伸缩调整成所需的实际值就行了。

偏差项

谈到如何计算神经元总输入值时，我们曾提到每个神经元都配有一个偏差项（bias）。这个偏差项就是每个神经元的偏差值和偏差权重，前面所述的总输入值公式中，已表现出来，但为了方便起见，这里再次列出来：

$$n_j = \sum n_i w_{ij} + b_j w_j$$

b_j 是偏差值，而 w_j 是偏差权重。

要了解偏差项做什么，需要检查，用来替神经元的总输入值，产生输出值的活化函数。基本而言，偏差项是把总输入值沿着活化函数的水平轴移动，有效改变神经元活化的阈值。偏差值总是设为 1 或 −1，而其权重则经由训练调整，如同其他权重那样。本质上，这让神经元可以习得每个神经元活化的适当阈值。

有些人总是把偏差值设为1，或总是设为-1。就我们的经验而言，使用1或-1其实无关紧要，因为经由训练后，网络会调整偏差权重以符合你的选择。权重可以是正或负，所以，如果神经网络认为偏差项应为负的，就会调整权重使其满足，无论选择1还是-1。如果选择1，神经网络就会找出适当的负值权重，而如果选择-1，则会找出适当的正值权重。当然，你是经过训练网络，或者让网络演化而达到这些目标的，本章稍后就会谈到。

输出层

就像输入层那样，你替网络选择的输出神经元，也和所需解决问题有关。一般而言，最好把输出神经元数量保持最少，以减少计算和训练时间。

设想一个网络，给了特定的输入值后，想让输出结果用来分类输入值。也许你想判断一组特定输入值是否落入某类型内。就此而言，你可以使用一个输出神经元。如果该神经元受到活化，则结果为`true`，如果没有受到活化，则结果为`false`，也就是说输入值没有落在所考虑的类型内。如果用logistic函数作输出活化运算，0.9左右的输出值表示受到活化或`true`，而0.1左右的输出值表示没受到活化或`false`。实际上，你也许无法刚好得到0.9或0.1的输出值，例如，也许得到0.78或0.31。因此，必须定义一个阈值，以便评估特定输出值是否表示活化。一般而言，可以直接在两极值间，选一个输出阈值。就logistic函数而言，可以用0.5。如果输出值大于0.5，则结果就是受到活化或`true`，否则就是`false`。

当你感兴趣的是某类输入值，是否落在多种类型中的其中一种时，就可以使用一个以上的输出神经元。参考图14-3的网络，我们根本上是想区分出敌人引起的威胁类型，这些类型是空中威胁、地面威胁、陆空威胁或者没有威胁。每一种类型都有一个输出神经元。就这类输出形式而言，假定高输出值指的是受到活化，而低输出值指的是没受到活化。每个节点的实际输出值，可以包含在某范围之内，至于是什么范围，就和网络是如何训练，所用输出活化函数的种类有关了。有了一组输入值，以及每个输出节点的结果，其中一种了解输出值是否活化的方式，就是找出拥有最高输出值的神经元。这就是所谓的“赢家通吃”（winner-take-all）法。有最高活化值的神经元就是最后所得的类型。本章后面会举一个用此法的实例。

通常，你要的神经网络，是根据一组输入值而得到单一值。时间连续预测，就是使用单一输出神经元的情况；需要根据随时间演变的历史资料，预测下一个值。就此而言，输出神经元的值就是你感兴趣的预测值。然而，记住一点，如果用的输出活化函数会产生某范围内的值，比如logistic函数，或者要预测的值的大小，会落在某个范围内，也许必须通过伸缩以调整其结果。

就其他情况而言，如图14-2所示，也许有一个以上的输出神经元，用来直接控制其他系统。就图14-2所示的实例而言，输出值控制了半履带机器人的每条履带的运动。该例中，使用双曲正切函数，求输出神经元之值，使输出值介于−1和+1之间，也许就很有用。那么，负值就表示向后移动，而正值就表示向前移动。

有时候，你可能需要一种网络，让输出神经元数量和输入神经元数量相等。这类网络通常用来做自动联想（样式辨认）以及数据压缩。这里的目标是输出神经元，应该反映出输入值。就样式辨认而言，这样的网络应该训练成输出其输入值。训练数据集，应该由许多感兴趣的样式样本组成。这里的想法是如果提供了一组不太明确的样式，或者无法在训练数据集中找到完全吻合的样式，则网络应该可以输出训练数据集中，和该输入样式最为接近的样式。

隐匿层

到目前为止，我们讨论了输入神经元、输出神经元，还有如何替神经元计算总输入值，但还没专门讨论隐匿层。在三层前馈网络中，有一层神经元放在隐匿层里，夹在输入层和输出层之间。

如图14-5所示，每个输出神经元都和每个隐匿神经元连接在一起。而且，每个隐匿神经元会把其输出传到每个输出层的神经元。顺带提一下，神经网络并不是只有这种结构，还有其他各种各样的形式，比如有一个以上的隐匿层、回馈层，以及完全没有隐匿层。然而，三层前馈网络是最常用的一种形态。无论如何，隐匿层是网络处理输入数据的特征所不可或缺的。隐匿层神经元越多，网络能处理的特征就越多；相反的，隐匿层神经元越少，网络能处理的特征就越少。

那么，所谓的特征是什么？为了了解这个特征的含义，可以把神经网络当作函数的近似值的方式来思考。假设有一个函数，如图14-10所示，噪声很多。

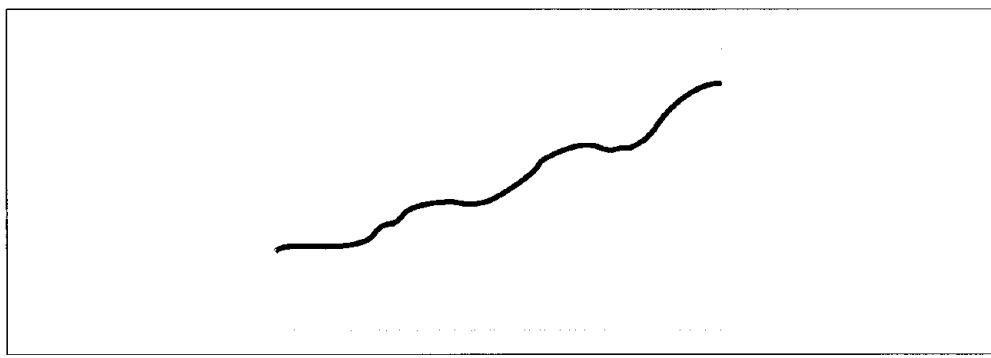


图14-10：噪声函数

训练神经网络来近似噪声很多的函数时，如果使用很少的隐匿层神经元时，可能会得到如图 14-11 所示的结果。

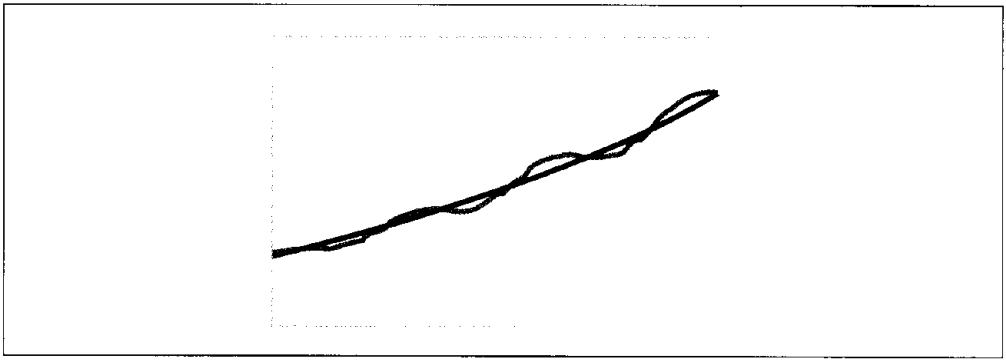


图 14-11：使用很少的神经元来近似噪声函数

如图 14-11 所示，近似的函数有捕捉输入数据的趋势，但漏掉了各处噪声的特征。某些情况下，比如噪声消减应用，这正是你想要的，然而，就其他问题而言，可能就不想要这种结果了。如果走另一个极端，选择很多隐匿层神经元，近似后的函数除了函数整体趋势外，也抓住了各处的噪声特征。在某些情况下，这可能就是你想要的，然而，在其他情况下，你的网络可能被训练过度，只要新的输入数据不在训练数据集内，就无法概括在内。

特定应用情况下，究竟多少隐匿神经元才适当，很难固定。一般而言，要用试误法决定。然而，有条原则你会觉得有所帮助；如果你的三层网络不是用于自动联想，隐匿层的适当数量，大约等于输入和输出神经元数量乘积的平方根。这只是粗略值，但的确是不错的起试点。记住一点，尤其是碰上CPU用量很大的游戏，隐匿层神经元数量越多，计算网络输出值的时间就越长。因此，试着把隐匿层神经元数量缩小到最少，这是有帮助的。

训练

到目前为止，我们不断提及训练网络，却没有实际告诉你该怎么做的细节。本节要说明这一点。

训练的目标是找出连接所有神经元之值的权重，让输入资料可以产生所需的输出值。你大概也想到了，除了选出权重值外，还有其他要训练的。本质上，训练神经网络就是一种最佳化过程，在这个过程里，你试着找出最佳权重值，让网络能产生合适的输出。

训练方式可以分成两类：指导训练（supervised training）以及无指导训练（unsupervised training）。仅靠一章无法详尽描述某些常用训练方法，更别提全部都谈了，因此，我们要集中讨论指导训练法中最常用的一种：倒传递法（back-propagation）。

倒传递训练

重申一次，训练的目标是找出连接所有神经元之值的权重，让输入资料可以产生所需的输出值。为达此目的，你需要一个训练数据集，由输入数据和对应该输入数据所要的输出值组成。下一步是利用某些技术的其中一种，不断重复地替整个网络找出一组权重值，使得该网络可以产生符合训练数据集内，每组数据所要的输出结果。做好之后，就可以让网络启动，提供不在训练数据集内的新数据，使其产生合理的输出结果。

因为训练是一种最佳化的过程，我们需要用某种方法做最佳化。就倒传递法而言，是实施试误法，把误差最小化。有了输入值以及产生的输出值后，必须将产生的输出值和已知想要的输出值作比较，量化两结果间的吻合度，例如，算出误差。有很多误差的算法可用，这里用最常见的：均方误差（mean square error），也就是计算所得的输出值，和想要的输出值之间的差值的平方的平均值。

如果你学过微积分，也许想得起来，要对函数最小化或最大化，就必须对该函数微分。因为要把误差最小化，以得出最佳的权重，必须在某处微分，自然无需惊讶。明确地讲，我们必须对活化函数微分，而这就是为什么logistic 函数非常好用的原因所在，我们可以轻易用解析方法求出其导数。

如前所述，找出最佳权重是个不断重复的过程，就像这样：

1. 一开始是一个训练数据集，包含输入数据和相对应的所希望的输出值。
2. 替神经网络的权重设初值，设成某些随机的较小的数值。
3. 把每组输入数据输入给网络，算出输出值。
4. 比较算出的输出值和所希望的输出值，算出误差。
5. 调整权重以减小误差，再重复上述过程。

这个过程的执行可以通过两种方式。一种是算出误差值，替每组输入数据和所希望的输出数据调整权重，然后继续试误下一组输入/输出数据。另一种方式是算出训练数据集内，所有组数据的输入和所希望输出数据的累加误差量，然后调整权重，再重复这个过程。每一次的重复就叫做一轮（epoch）。

步骤 1～3 相当清楚，稍后我们会看到实例。不过，现在先细谈步骤 4～5。

计算误差

要训练神经网络，你得给定一组输入数据，使其产生某些输出。要根据某组输入值比较计算所得到的输出值和所希望的输出值时，必须计算其误差。这样不但可以确认计算所得的输出值是对是错，也可以确认其对错的程度。最常用的误差就是均方误差，也就是所希望的输出值，和计算的输出值之间的差值的平方的平均值：

$$\varepsilon = \frac{\sum (n_c - n_d)^2}{m}$$

在此公式中，ε是此训练资料集的均方误差。n_c和n_d分别是计算所得的输出值和所希望的输出值，也就是所有输出神经元之值；而m是每轮的输出神经元的数量。

目标是通过不断调整网络中连接所有神经元的权重值，使误差值尽可能小到在实际中可行。要知道权重需做多大调整，每轮进行时，也必须算出输出层和隐匿层中，每个神经元的误差。输出神经元计算误差的公式如下：

$$\delta_i^0 = \Delta n_i^0 f'\left(n_{ci}^0\right)$$

这里的δ_i^o是第i个输出神经元的误差，而Δn_i^o是第i个输出神经元，其计算的输出值和所希望的输出值间的差值，此外，$f'(n_{ci}^o)$是活化函数的导数，代入第i个输出神经元之值。前面说过必须在某处计算导数，就是在这个地方了。这就是为什么logistic函数好用的地方，其导数形式相当简单，很容易以解析方法算出。使用logistic函数的导数重写此公式后，就得出下列计算输出神经元误差的公式了：

$$\delta_i^0 = \left(n_{di}{}^0 - n_{ci}{}^0\right) n_{ci}{}^0 \left(1 - n_{ci}{}^0\right)$$

此方程式中，$n_{di}{}^0$是第i个神经元所希望的输出值，而$n_{ci}{}^o$是第i个神经元计算的输出值。

就隐匿层神经元而言，误差公式有点不同。就此而言，和每个隐匿神经元有关的误差如下：

$$\delta_i{}^h = \left(\sum \delta_j^0 w_{ij}\right) f'\left(n_{ci}{}^h\right)$$

注意，这里每个隐匿层神经元的误差，是该隐匿神经元连接的每个输出层神经元的误差，乘以连接关系上的权重值的函数。也就是说，要计算此误差，并依序调整权重，你必须从输出层往回走向输入层。

此外，注意到，这里再次需要用到活化函数的导数。假设用 logistic 活化函数，最后会得到下列结果：

$$\delta_i^h = \left(\sum \delta_j^0 w_{ij}\right) n_{ai}^h \left(1 - n_{ai}^h\right)$$

最后一点，输入层没有所谓的误差，因为那些神经元之值是我们给定的。

调整权重

算出误差后，可以继续算出网络中每个权重要做的适当校正值。每个权重的校正值如下：

$$\Delta w = \rho \delta_i n_i$$

在此公式中，ρ 是学习率（learning rate），δ_i 是考虑中神经元的误差，而 n_i 是考虑中神经元之值。新权重值就是旧权重值加上 Δw。

记住一点，每个权重都有校正值，而且校正值各不相同。更新连接输出神经元和隐匿神经元之间的权重时，由输出神经元的误差及其值来计算权重的校正值。当更新连接隐匿层神经元和输入层神经元之间的权重时，则用隐匿层神经元的误差及其值。

学习率是个常数，影响每个权重要被调整的多少，通常设成小值，比如 0.25 或 0.5。这是你必须调整的几个参数之一。如果设得太高，权重的最佳化也许会过头；但如果设得太低，训练可能得花较长的时间。

动量

许多倒传递实践家会对我们刚才谈过的权重校正值做一点修改。修改技巧名叫增加动量（adding momentum）。我们在谈怎么增加动量之前，先讨论为何要增加动量。

在任何通用最佳化过程中，目标不是最小化就是最大化某个函数。更明确地讲，我们感兴趣的是在某个输入参数的范围内，找出特定函数整体的最小值或最大值。问题是很多函数展现出来的，都是所谓的局部最小值或局部最大值。基本上，就是函数的谷和峰，如图 14-12 所示。

此例中，这个函数在图中的范围内，有整体最小值和最大值，此外，还有好几个局部最小值和最大值，由小谷小峰表示。

就此而言，我们感兴趣的是把网络的误差最小化。明确地讲，是找出最佳权重，使其产生整体最小误差，然而，也许会得到局部最小误差，而不是整体最小误差。

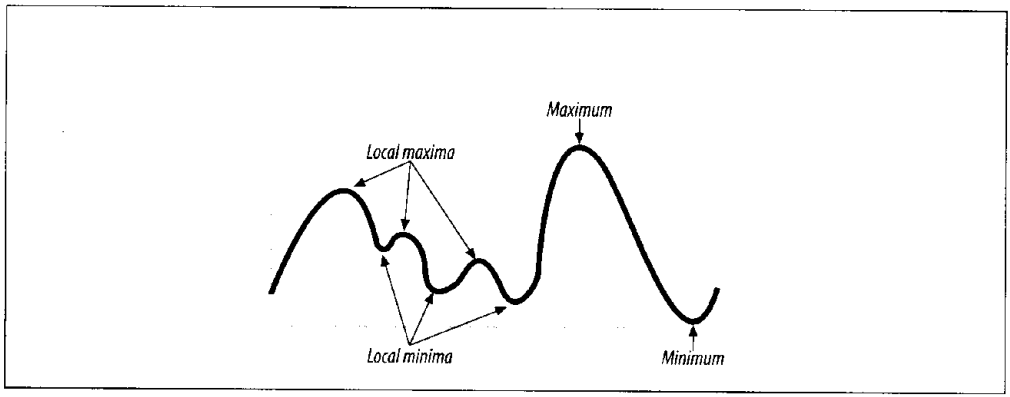


图 14-12：局部最小值和最大值

当网络开始训练时，我们把权重设成某些随机小值。此时，不知道这些值和最佳权重有多接近，因此，也许在设初值时让网络比较接近局部最小值，而不是整体最小值。我们的目的不是要谈微积分，但更新权重的方法是一种叫做梯度下降法（gradient descent）的技巧，也就是用函数的导数，试着取得最小值，就我们的情况来讲，就是最小误差值。问题是不知道得到的是整体最小值，还是局部最小值，一般而言，神经网络所谓的误差范围（error-space）内总是会碰到局部最小值。

这类问题在所有最佳化技巧中是很常见的，而且有很多不同的方法试着解决这种难题。动量法就是这种用于神经网络的技巧。动量方法，并没有去掉缩小至局部最小值的可能性，但这种方法有助于远离局部最小值，而往整体最小值靠近，这也就是为什么名称中有动量两个字的原因。基本而言，我们要在权重校正值中再加上一个小值，也就是前一轮中，权重校正值的函数。这样会给权重校正值一个动力，如果逼近的是局部最小值，这样的算法也许可以跳过局部最小值，往整体最小值靠近。

所以，用上增加动量后，计算权重校正值的新公式便如下所示：

$$\Delta w = \rho \delta_i n_i + \alpha(\Delta w')$$

在此公式中，$\Delta w'$是上一轮的权重校正值，而α是动量系数。动量系数也是另一个必须调整的参数，通常是设成某个介于 0.0 和 1.0 之间的小数值。

编写神经网络的程序

终于要来看一些实现三层前馈神经网络的源代码了。下面几节是两个实现这种网络的 C++类。本章稍后会看一个利用这几个类的实作范例。如果你想先知道这个神经网络的运行过程，再看其内部细节，可先跳到“用大脑解决追逐和闪躲之决策”那一节。

我们必须替三层前馈神经网络实现两个类。第一个类代表的是通用层，可用作输入层、隐匿层和输出层。第二个类代表由这三层组成的整个神经网络。下面几节会介绍每个类的完整源代码。

层次类

NeuralNetworkLayer类实现了多层前馈网络中的通用层，负责处理该层内所包含的神经元。其执行的任务包括，配置和释放储存神经元之值、误差和权重的内存、对权重赋初值、计算神经元之值以及调整权重。例 14-1 是该类开头的内容。

例 14-1：NeuralNetworkLayer 类

```
class NeuralNetworkLayer
{
public:
     int               NumberOfNodes;
     int               NumberOfChildNodes;
     int               NumberOfParentNodes;
     double**          Weights;
     double**          WeightChanges;
     double*           NeuronValues;
     double*           DesiredValues;
     double*           Errors;
     double*           BiasWeights;
     double*           BiasValues;
     double            LearningRate;

     bool              LinearOutput;
     bool              UseMomentum;
     double            MomentumFactor;

     NeuralNetworkLayer*     ParentLayer;
     NeuralNetworkLayer*     ChildLayer;

     NeuralNetworkLayer();

     void     Initialize(int NumNodes,
                         NeuralNetworkLayer* parent,
                         NeuralNetworkLayer* child);
     void     CleanUp(void);
     void     RandomizeWeights(void);
     void     CalculateErrors(void);
     void     AdjustWeights(void);
     void     CalculateNeuronValues(void);
};
```

层次之间彼此的连接方式是采用上下关系。例如，输入层是隐匿层的上层，而隐匿层是输出层的上层。此外，输出层是隐匿层的下层，而隐匿层是输入层的下层。注意到，输入层没有上层，而输出层也没有下层。

这个类的成员主要是由数组组成，用来储存神经元权重、内含值、误差以及偏差项。此外，还有一些成员含有一些设定值，控制此层的行为。成员如下所示：

NumberOfNodes

这个成员储存的是层次类实体中，神经元（或者说节点）的数量。

NumberOfChildNodes

这个成员储存的是层次类实体中，位于下层内神经元的数量。

NumberOfParentNodes

这个成员储存的是层次类实体中，位于上层内神经元的数量。

Weights

这个成员是一个指针，指向某个双精确度之值的指针。基本而言，代表的是连接上下层节点的二维权重数组。

WeightChanges

这个成员也是一个指针，指向某个双精确度之值的指针，也就是可以动态存取已配置内存的二维数组。就此而言，储存在此数组中的值，就是要对权重值做调整的校正值。我们需要这些校正值才能实现前面讨论过的动量法。

NeuronValues

这个成员是指向某双精确度之值的指针，也就是可以动态存取已配置内存的数组，储存的是该层内神经元计算所得的值（或活化值）。

DesiredValues

这个成员也是指向某双精确度之值的指针，也就是可以动态存取已配置内存的数组，储存的是该层内神经元所要的值（或目标值）。这些值是给输出数组使用的；我们会根据计算所得的输出值以及训练数据集的目标输出值来计算误差。

Errors

这个成员是指向某双精确度之值的指针，也就是可以动态存取已配置内存的数组，储存的是该层中和每个神经元相关的误差。

BiasWeights

这个成员是指向某双精确度之值的指针，也就是可以动态存取已配置内存的数组，储存的是该层中和每个神经元相连接的偏差权重值。

BiasValues

这个成员是指向某双精确度之值的指针，也就是可以动态存取已配置内存的数组，储存的是该层中和每个神经元相连接的偏差值。注意到，此成员并非必要，因为我们通常把偏差值设为+1或−1之后，就不予理会了。

LearningRate

此成员储存的是学习率，用于计算权重校正值。

LinearOutput

此成员储存的是一标号，指出此层的神经元是否使用线性活化函数。只有该层是输出层时才能用。如果标号为false，则改用logistic活化函数。默认值为false。

UseMomentum

此成员储存一标号，指出调整权重值时是否使用动量。默认值是false。

MomentumFactor

此成员储存的是动量系数，如前所述。如果UseMomentum标号为true，才用到这个变量。

ParentLayer

此成员储存的是指向代表此层的上层NeuralNetworkLayer实体的指针。对输入层而言，此指针设为NULL。

ChildLayer

此成员储存的是指向代表此层的下层NeuralNetworkLayer实体的指针。对输出层而言，此指针设为NULL。

NeuralNetworkLayer类包含七个方法。我们逐一详谈，先从例14-2的构造方法谈起。

例14-2：NeuralNetworkLayer类的构造方法

```
NeuralNetworkLayer::NeuralNetworkLayer()
{
     ParentLayer = NULL;
     ChildLayer = NULL;
     LinearOutput = false;
     UseMomentum = false;
     MomentumFactor = 0.9;
}
```

这个构造方法非常简单，就是为我们刚才讨论过的一些成员赋初值，而例14-3的Initialize()还用得比较多。

例14-3：Initialize()方法

```
void NeuralNetworkLayer::Initialize(int NumNodes,
                                    NeuralNetworkLayer* parent,
                                    NeuralNetworkLayer* child)
{
     int     i, j;

     // 配置内存
     NeuronValues = (double*) malloc(sizeof(double) *
                                     NumberOfNodes);
```

```
DesiredValues = (double*) malloc(sizeof(double) *
                                 NumberOfNodes);
Errors = (double*) malloc(sizeof(double) * NumberOfNodes);

if(parent != NULL)
{
     ParentLayer = parent;
}

if(child != NULL)
{
     ChildLayer = child;


     Weights = (double**) malloc(sizeof(double*) *
                                 NumberOfNodes);
     WeightChanges = (double**) malloc(sizeof(double*) *
                                       NumberOfNodes);
     for(i = 0; i<NumberOfNodes; i++)
     {
          Weights[i] = (double*) malloc(sizeof(double) *
                                        NumberOfChildNodes);
          WeightChanges[i] = (double*) malloc(sizeof(double) *
                                              NumberOfChildNodes);
     }

     BiasValues = (double*) malloc(sizeof(double) *
                           NumberOfChildNodes);
     BiasWeights = (double*) malloc(sizeof(double) *
                            NumberOfChildNodes);
} else {
     Weights = NULL;
     BiasValues = NULL;
     BiasWeights = NULL;
     WeightChanges = NULL;
}

// 确定全部为0
for(i=0; i<NumberOfNodes; i++)
{
     NeuronValues[i] = 0;
     DesiredValues[i] = 0;
     Errors[i] = 0;

     if(ChildLayer != NULL)
          for(j=0; j<NumberOfChildNodes; j++)
          {
               Weights[i][j] = 0;
               WeightChanges[i][j] = 0;
          }
}

// 指定偏差值和权重的初值
if(ChildLayer != NULL)
```

```
        for(j=0; j<NumberOfChildNodes; j++)
        {
            BiasValues[j] = -1;
            BiasWeights[j] = 0;
    }
}
```

Initialize()方法的任务是替那些动态数组配置内存，以便储存该层内神经元的权重、值、误差以及偏差值和偏差权重值，此外，还替这些数组赋初值。

这个方法有三个参数：该层的节点数（或神经元数量）、指向上层的指针以及指向下层的指针。如果该层是输入层，则上层指针之处应传递NULL。如果该层是输出层，则下层指针之处应该递NULL。

此方法开头，NeuronValues、DesiredValues和Errors数组的内存都会被配置好。这几个数组都是一维数组，元素个数由该层节点数决定。

接着，设定上下层指针。如果下层指针是NULL，则此层为输入层或隐匿层，而且必须配置相连接的权重的内存。因为Weights和WeightChanges是二维数组，我们必须分步骤配置内存。第一步是配置内存，储存double类型数组的指针，其元素个数则由该层节点数决定。接着，对每一元素，也要配置另一块内存，储存实际数组值，至于个数是多少，则由下层的节点数决定。输入层或隐匿层的每个神经元，都和其相关联的下层内每个神经元相连接，因此，权重数组和权重校正值数组的总量，就等于该层内神经元的数量，乘以下层内神经元数量。

我们也要配置偏差值和偏差权重值数组的内存，其总量等于相连接的下层内神经元数量。

内存配置好之后，再来替数组赋初值。就大部分情况而言，我们要让数组的值都是0，除了偏差值数组之外，把偏差值数组都设为-1；注意到，如前所述，也可以都设为+1。

例14-4是CleanUp()方法，负责把Initialize()方法里配置的内存都释放掉。

例14-4：CleanUp()方法

```
void NeuralNetworkLayer::CleanUp(void)
{
    int     i;

    free(NeuronValues);
    free(DesiredValues);
    free(Errors);

    if(Weights != NULL)
    {
        for(i = 0; i<NumberOfNodes; i++)
        {
```

```
            free(Weights[i]);
            free(WeightChanges[i]);
        }

        free(Weights);
        free(WeightChanges);
    }

    if(BiasValues != NULL) free(BiasValues);
    if(BiasWeights != NULL) free(BiasWeights);
}
```

这个程序一目了然，只是使用 free 释放所有动态配置的内存而已。

前面我们提过，神经网络权重只有初始化成某些随机小值，才能开始训练。例 14-5 的 RandomizeWeights()方法替我们处理这件事。

例 14-5：RandomizeWeights()方法

```
void NeuralNetworkLayer::RandomizeWeights(void)
{
    int     i,j;
    int     min = 0;
    int     max = 200;
    int     number;

    srand( (unsigned)time( NULL ) );

    for(i=0; i<NumberOfNodes; i++)
    {
        for(j=0; j<NumberOfChildNodes; j++)
        {
            number = (((abs(rand())%(max-min+1))+min));

            if(number>max)
                number = max;

            if(number<min)
                number = min;

            Weights[i][j] = number / 100.0f - 1;
        }
    }

    for(j=0; j<NumberOfChildNodes; j++)
    {
            number = (((abs(rand())%(max-min+1))+min));

            if(number>max)
                number = max;

            if(number<min)
                number = min;
```

```
            BiasWeights[j] = number / 100.0f - 1;
    }
}
```

这个方法所做的事就是替Weights数组中，每个权重在−1和+1之间随机选取数。BiasWeights数组中的偏差权重也是如此。开始训练前才调用这个方法。

下一个方法CalculateNeuronValues()的任务是，利用先前提到的神经元总输入值和活化函数的公式，计算该层内每个神经元的活化值或内含值。例14-6是此方法的程序。

例14-6：CalculateNeuronValues()方法

```
void NeuralNetworkLayer::CalculateNeuronValues(void)
{
    int          i,j;
    double       x;

    if(ParentLayer != NULL)
    {
        for(j=0; j<NumberOfNodes; j++)
        {
            x = 0;
            for(i=0; i<NumberOfParentNodes; i++)
            {
                x += ParentLayer->NeuronValues[i] *
                     ParentLayer->Weights[i][j];
            }
            x += ParentLayer->BiasValues[j] *
                 ParentLayer->BiasWeights[j];

            if((ChildLayer == NULL) && LinearOutput)
                NeuronValues[j] = x;
            else
                NeuronValues[j] = 1.0f/(1+exp(-x));
        }
    }
}
```

在此方法中，以嵌套for语句来绕行所有权重值。j循环绕行下层节点，而i循环则绕行上层节点。在这些嵌套循环中，总输入值会算出来，存储在x变量内。该层中每个节点的总输入值是上层（i循环）所有和每个j节点相连之值的加权总和，再加上j节点的加权偏差值。

算出每个节点的总输入值后，可以利用活化函数算出每个神经元之值。每一层都是用logistic函数，但输出层除外，因为要根据LinearOutput标号的值来决定是否使用线性活化函数。

例14-7的CalculateErrors()方法负责使用先前讨论过的公式算出每个神经元的误差。

例 14-7：CalculateErrors()方法

```
void NeuralNetworkLayer::CalculateErrors(void)
{
     int          i, j;
     double       sum;

     if(ChildLayer == NULL) // 输出层
     {
          for(i=0; i<NumberOfNodes; i++)
          {
                Errors[i] = (DesiredValues[i] - NeuronValues[i]) *
                            NeuronValues[i] *
                            (1.0f - NeuronValues[i]);
          }
     } else if(ParentLayer == NULL) { // 输入层
          for(i=0; i<NumberOfNodes; i++)
          {
               Errors[i] = 0.0f;
          }
     } else { // 隐匿层
          for(i=0; i<NumberOfNodes; i++)
          {
               sum = 0;
               for(j=0; j<NumberOfChildNodes; j++)
               {
                    sum += ChildLayer->Errors[j] * Weights[i][j];
               }
               Errors[i] = sum * NeuronValues[i] *
                           (1.0f - NeuronValues[i]);
          }
     }
}
```

如果该层没有下层，也就是该层为输出层时会有此情况，此时就要使用计算输出层误差的公式。如果该层没有上层，也就是该层为输入层时也会有此情况，此时就要把误差设为 0。如果该层同时有上下层，则其为隐匿层，而应使用隐匿层的误差计算公式。

例 14-8 的 AdjustWeights()方法负责计算每个相连权重的校正值。

例 14-8：AdjustWeights()方法

```
void NeuralNetworkLayer::AdjustWeights(void)
{
     int          i, j;
     double       dw;

     if(ChildLayer != NULL)
     {
          for(i=0; i<NumberOfNodes; i++)
          {
               for(j=0; j<NumberOfChildNodes; j++)
               {
```

```
                    dw = LearningRate * ChildLayer->Errors[j] *
                          NeuronValues[i];
                    if(UseMomentum)
                    {
                         Weights[i][j] += dw + MomentumFactor *
                                          WeightChanges[i][j];
                         WeightChanges[i][j] = dw;
                    } else {
                              Weights[i][j] += dw;
                    }
               }
          }

          for(j=0; j<NumberOfChildNodes; j++)
          {
               BiasWeights[j] += LearningRate *
                                 ChildLayer->Errors[j] *
                                 BiasValues[j];
          }
     }
}
```

如果该层有下层，则权重必须调整，也就是该层为输入层或隐匿层的情况。输出层没有下层，因此没有相连接的权重可以调整。嵌套 for 循环绕行该层以及下层的节点。记住一点，每层中的每个神经元都和下层的每一节点相连接。在这些嵌套循环中，权重校正值的计算运用前述的公式。如果增加动量，则动量系数乘以上一轮权重校正值，也会加入此权重校正值内。然后，此轮的权重校正值就会储存在 WeightChanges 数组内，以供下一轮使用。如果没有增加动量，则权重校正值就不会加上动量，也就没必要储存权重校正值了。

最后，偏差权重也以类似连接权重的调整方式来调整。就每个和子节点相连的偏差权重而言，其校正值就是学习率乘以下层神经元误差，再乘以偏差值。

代表神经网络的类

NeuralNetwork 类包括三个 NeuralNetworkLayer 类实体，分别是网络中的每一层：输入层、隐匿层以及输出层。例 14-9 是此类开头的内容。

例 14-9：NeuralNetwork 类

```
class NeuralNetwork
{
public:
     NeuralNetworkLayer     InputLayer;
     NeuralNetworkLayer     HiddenLayer;
     NeuralNetworkLayer     OutputLayer;
```

```
    void    Initialize(int nNodesInput, int nNodesHidden,
                       int nNodesOutput);
    void    CleanUp();
    void    SetInput(int i, double value);
    double  GetOutput(int i);
    void    SetDesiredOutput(int i, double value);
    void    FeedForward(void);
    void    BackPropagate(void);
    int     GetMaxOutputID(void);
    double  CalculateError(void);
    void    SetLearningRate(double rate);
    void    SetLinearOutput(bool useLinear);
    void    SetMomentum(bool useMomentum, double factor);
    void    DumpData(char* filename);
};
```

这个类只有三个成员，对应三个层次实体。然而，这个类有13个方法，我们要逐一说明。

例14-10是Initialize()方法。

例14-10：Initialize()方法

```
void NeuralNetwork::Initialize(int nNodesInput,
                               int nNodesHidden,
                               int nNodesOutput)
{
    InputLayer.NumberOfNodes = nNodesInput;
    InputLayer.NumberOfChildNodes = nNodesHidden;
    InputLayer.NumberOfParentNodes = 0;
    InputLayer.Initialize(nNodesInput, NULL, &HiddenLayer);
    InputLayer.RandomizeWeights();

    HiddenLayer.NumberOfNodes = nNodesHidden;
    HiddenLayer.NumberOfChildNodes = nNodesOutput;
    HiddenLayer.NumberOfParentNodes = nNodesInput;
    HiddenLayer.Initialize(nNodesHidden,&InputLayer,&OutputLayer);
    HiddenLayer.RandomizeWeights();
    OutputLayer.NumberOfNodes = nNodesOutput;
    OutputLayer.NumberOfChildNodes = 0;
    OutputLayer.NumberOfParentNodes = nNodesHidden;
    OutputLayer.Initialize(nNodesOutput, &HiddenLayer, NULL);

}
```

Initialize()带三个参数，对应构成此网络三个层次的每一层中所含的神经元数量。这些参数用来对输入层、隐匿层以及输出层的层次类实体做初始化。Initialize()也会把层次间正确的上下关系连接做好。接着，继续替连接的权重值赋随机值。

例14-11的CleanUp()方法只是调用每个层次实体的CleanUp()方法而已。

例 14-11：CleanUp()方法

```
void NeuralNetwork::CleanUp()
{
     InputLayer.CleanUp();
     HiddenLayer.CleanUp();
     OutputLayer.CleanUp();
}
```

SetInput()用来设定特定输入神经元的输入值。例 14-12 是 SetInput()方法。

例 14-12：SetInput()方法

```
void     NeuralNetwork::SetInput(int i, double value)
{
     if((i>=0) && (i<InputLayer.NumberOfNodes))
     {
          InputLayer.NeuronValues[i] = value;
     }
}
```

SetInput()带两个参数，对应需要设定输入值的神经元的索引值，以及输入值本身。然后，这条信息会设定特定的输入值。这个方法在两个地方会用到，一是在训练时要设定训练数据集的输入数据，二是网络实际运用时，要设定输入数据，使输出值能计算出来。

一旦网络产生一些输出值，我们要以某种方式取得。GetOutput()方法的用意就在于此。例 14-13 是 GetOutput()方法。

例 14-13：GetOutput()方法

```
double     NeuralNetwork::GetOutput(int i)
{
     if((i>=0) && (i<OutputLayer.NumberOfNodes))
     {
          return OutputLayer.NeuronValues[i];
     }

     return (double) INT_MAX; // 指出错误
}
```

GetOutput()带一个参数，也就是想要取得其输出值的输出神经元的索引值。这个方法会返回指定的输出神经元的内含值或活化值。注意到，如果你指定的索引值，不在有效输出神经元范围之内，则会返回 INT_MAX 表示指出错误。

训练期间，我们必须比较计算所得的输出值以及所希望的输出值。层次类让这类计算轻松许多，同时也可以储存所希望的输出值。如例 14-14 所示，SetDesiredOutput()方法可以轻松地根据某组输入值，指定需要的输出值。

例 14-14：SetDesiredOutput()方法

```
void NeuralNetwork::SetDesiredOutput(int i, double value)
{
     if((i>=0) && (i<OutputLayer.NumberOfNodes))
     {
          OutputLayer.DesiredValues[i] = value;
     }
}
```

SetDesiredOutput()带两个参数，对应要设定所想要的输出值的输出神经元的索引值，以及所想要的输出值本身。

为了实际让网络能根据一组输入值来产生输出，我们必须调用例 14-15 的 FeedForward()方法。

例 14-15：FeedForward()方法

```
void NeuralNetwork::FeedForward(void)
{
     InputLayer.CalculateNeuronValues();
     HiddenLayer.CalculateNeuronValues();
     OutputLayer.CalculateNeuronValues();
}
```

这个方法只是依次调用输入层、隐匿层以及输出层的CalculateNeuronValues()方法。一旦这些调用都执行完后，输出层将包含计算所得的输出值，然后，就可以调用 GetOutput()方法来检视了。

训练期间，一旦输出值算出后，我们必须利用倒传递技巧，调整连接的权重值。BackPropagate()方法可以做这项工作。例 14-16 是 BackPropagate()方法。

例 14-16：BackPropagate()方法

```
void NeuralNetwork::BackPropagate(void)
{
     OutputLayer.CalculateErrors();
     HiddenLayer.CalculateErrors();
     HiddenLayer.AdjustWeights();
     InputLayer.AdjustWeights();
}
```

BackPropagate()首先调用输出层和隐匿层的CalculateErrors()方法，输出层在前，然后是隐匿层。接着，BackPropagate()会调用隐匿层和输入层的AdjustWeights()方法，隐匿层在前，然后是输入层。这里的顺序很重要，而且必须按照例 14-16 所示的顺序，也就是我们倒着经过网络，而不是顺着向前走，像 FeedForward()方法那样。

当你使用的网络有好几个输出神经元时，而且再度采用“赢家通吃”法，求出被活化的输出神经元，此时，必须找出输出值最高的输出神经元。例14-17的GetMaxOutputID()就是此用意。

例 14-17：GetMaxOutputID()方法

```
int     NeuralNetwork::GetMaxOutputID(void)
{
     int          i, id;
     double     maxval;

     maxval = OutputLayer.NeuronValues[0];
     id = 0;

     for(i=1; i<OutputLayer.NumberOfNodes; i++)
     {
          if(OutputLayer.NeuronValues[i] > maxval)
          {
               maxval = OutputLayer.NeuronValues[i];
               id = i;
          }
     }

     return id;
}
```

GetMaxOutputID()只是绕行所有输出层的神经元，找出有最高输出值的神经元，而拥有最高值的神经元的索引值会被返回。

先前我们讨论过，必须算出某组输出值的误差。必须这么做的原因是为了做训练。CalculateError()方法会替我们做误差计算。例14-18是CalculateError()方法。

例 14-18：CalculateError()方法

```
double NeuralNetwork::CalculateError(void)
{
     int          i;
     double     error = 0;

     for(i=0; i<OutputLayer.NumberOfNodes; i++)
     {
          error += pow(OutputLayer.NeuronValues[i]-
                    OutputLayer.DesiredValues[i], 2);
     }

     error = error / OutputLayer.NumberOfNodes;

     return error;
}
```

CalculateError()会利用先前讨论过的均方误差公式，计算所得输出值和想要的输出值之间的误差。

为了方便起见，我们给出了SetLearningRate()方法，如例14-19所示。可以用来设定网络中每层的学习率。

例14-19：SetLearningRate()方法

```
void NeuralNetwork::SetLearningRate(double rate)
{
     InputLayer.LearningRate = rate;
     HiddenLayer.LearningRate = rate;
     OutputLayer.LearningRate = rate;
}
```

例14-20所示的是另一个便捷的方法SetLinearOutput()。你可以用来替网络中的每一层设定LinearOutput标号。然而，注意到，此实例中只有输出层会用到线性活化函数。

例14-20：SetLinearOutput()方法

```
void     NeuralNetwork::SetLinearOutput(bool useLinear)
{
     InputLayer.LinearOutput = useLinear;
     HiddenLayer.LinearOutput = useLinear;
     OutputLayer.LinearOutput = useLinear;
}
```

你可以用SetMomentum()方法，设定UseMomentum标号以及网络中每一层的动量系数，如例14-21所示。

例14-21：SetMomentum()方法

```
void     NeuralNetwork::SetMomentum(bool useMomentum, double factor)
{
     InputLayer.UseMomentum = useMomentum;
     HiddenLayer.UseMomentum = useMomentum;
     OutputLayer.UseMomentum = useMomentum;
     InputLayer.MomentumFactor = factor;
     HiddenLayer.MomentumFactor = factor;
     OutputLayer.MomentumFactor = factor;
}
```

DumpData()是便捷的方法，只是把网络的某些重要数据放到某个输出文件罢了。例14-22是DumpData()方法。

例14-22：DumpData()方法

```
void NeuralNetwork::DumpData(char* filename)
{
     FILE*     f;
     int          i, j;
```

```
f = fopen(filename, "w");

fprintf(f, "-------------------------------------------\n");
fprintf(f, "Input Layer\n");
fprintf(f, "-------------------------------------------\n");
fprintf(f, "\n");
fprintf(f, "Node Values:\n");
fprintf(f, "\n");
for(i=0; i<InputLayer.NumberOfNodes; i++)
     fprintf(f, "(%d) = %f\n", i, InputLayer.NeuronValues[i]);
fprintf(f, "\n");
fprintf(f, "Weights:\n");
fprintf(f, "\n");
for(i=0; i<InputLayer.NumberOfNodes; i++)
     for(j=0; j<InputLayer.NumberOfChildNodes; j++)
          fprintf(f, "(%d, %d) = %f\n", i, j, InputLayer.Weights[i][j]);

fprintf(f, "\n");
fprintf(f, "Bias Weights:\n");
fprintf(f, "\n");
for(j=0; j<InputLayer.NumberOfChildNodes; j++)
     fprintf(f, "(%d) = %f\n", j, InputLayer.BiasWeights[j]);
fprintf(f, "\n");
fprintf(f, "\n");
fprintf(f, "-------------------------------------------\n");
fprintf(f, "Hidden Layer\n");
fprintf(f, "-------------------------------------------\n");
fprintf(f, "\n");
fprintf(f, "Weights:\n");
fprintf(f, "\n");
for(i=0; i<HiddenLayer.NumberOfNodes; i++)
     for(j=0; j<HiddenLayer.NumberOfChildNodes; j++)
          fprintf(f, "(%d, %d) = %f\n", i, j,
                    HiddenLayer.Weights[i][j]);
fprintf(f, "\n");
fprintf(f, "Bias Weights:\n");
fprintf(f, "\n");
for(j=0; j<HiddenLayer.NumberOfChildNodes; j++)
     fprintf(f, "(%d) = %f\n", j, HiddenLayer.BiasWeights[j]);

fprintf(f, "\n");
fprintf(f, "\n");

fprintf(f, "-------------------------------------------\n");
fprintf(f, "Output Layer\n");
fprintf(f, "-------------------------------------------\n");
fprintf(f, "\n");
fprintf(f, "Node Values:\n");
fprintf(f, "\n");
for(i=0; i<OutputLayer.NumberOfNodes; i++)
     fprintf(f, "(%d) = %f\n", i, OutputLayer.NeuronValues[i]);
fprintf(f, "\n");
```

```
    fclose(f);
}
```

传送到指定输出文件的数据有网络中各层的权重、内含值以及偏差权重。当你想检视网络的内部细节时，这样做很有用。当你在调试或者用某种工具程序训练网络，想把训练后的权重直接用到实际游戏中，节省在游戏开始时，又必须从头训练的时间，那么这个方法就更有用了。就后者的目的而言，你得修改此处的 NeuralNetwork 类，以便从外部资源轻松载入权重。

用大脑解决追逐和闪躲之决策

本节要讨论的范例是第四章已讨论过的群聚和追逐范例，只不过做了一些修改。那一章我们讨论的一个例子是，一群单位追逐玩家控制的单位。此例做的修改是，计算机控制的单位改用神经网络，决定是否追逐或闪躲玩家，或者和其他计算机控制单位聚在一起。如果你的游戏中有生物或单位可以和玩家对战，这个范例就是这种游戏场景的理想情况或近似情况。我们不要让生物总是去攻击玩家，也不要用有限状态机的“大脑”，现在改用神经网络，不但替生物做决策，也利用生物和玩家交战的经验，使其行为有所适应。

下面说明简单实例如何运作。大约有 20 个计算机控制的单位在屏幕上移动，会攻击玩家、逃离玩家或者和其他计算机控制单位聚在一起。所有这些行为，都可用前几章介绍的定性算法来处理，然而，要采取什么行为，则是由神经网络做决策。玩家可以随心所欲地在屏幕上四处移动。当玩家和计算机控制的单位彼此接近到指定半径范围内时，我们就假定他们会发生战斗。我们不想在此实际模拟战斗，而是用基本的系统，当计算机控制单位和玩家处在战斗距离内，每次游戏循环运行一轮时，就会减掉一些容许击中次数。玩家在战斗范围内，也会减掉某些容许击中次数，以和计算机控制单位的数量成比例。当某单位的容许击中次数变为零时，就会死掉，而后自动重生。

所有计算机控制单位都共享同一个“脑”，也就是神经网络。我们打算让这个“脑”随着计算机控制单位和玩家互动习得经验而能够演化。这必须在游戏中实行倒传递算法，实时调整网络权重，来达到这个目的。我们也假定计算机控制单位会集体演化。

我们希望能看见计算机控制单位，学会在和玩家对战毫无胜算时，就避开玩家。相反的，我们希望看见，当计算机控制单位，学习到他们的玩家对手很弱时，就变得比较有攻击性。另一种可能是，计算机控制单位学会群体行动，如此一来，打败玩家的机会就会高一点。

初始化以及训练

利用第四章的群聚实例作为起点，必须做的第一件事，就是新增一个全局变量TheBrain，来表示神经网络，如例14-23所示。

例14-23：新全局变量

```
NeuralNetwork     TheBrain;
```

程序开始时，必须对神经网络做初始化，包括神经网络的配置和训练。Initialize()函数是从前例移过来的，显然是用来对神经网络做初始设定的，如例14-24所示。

例14-24：初始化

```
void     Initialize(void)
{
     int i;

     .
     .
     .
     for(i=0; i<_MAX_NUM_UNITS; i++)
     {
          .
          .
          .
          Units[i].HitPoints = _MAXHITPOINTS;
          Units[i].Chase = false;
          Units[i].Flock = false;
          Units[i].Evade = false;
     }

     .
     .
     .
     Units[0].HitPoints = _MAXHITPOINTS;

     TheBrain.Initialize(4, 3, 3);
     TheBrain.SetLearningRate(0.2);
     TheBrain.SetMomentum(true, 0.9);
     TrainTheBrain();
}
```

这个版本的Initialize()的程序代码和前例的大部分都相同，所以，例14-24中省略了一些内容。此例所新增的程序代码用来处理把神经网络整合进来的问题。

注意到，我们必须替此刚体结构新增一些成员，如例14-25所示。新的成员包括容许的击中次数，以及指出该单位是否在追逐、闪躲或群聚的标号。

例 14-25：RigidBody2D 类

```
class RigidBody2D {
public:
     .
     .
     .
     double   HitPoints;
     int      NumFriends;
     int      Command;
     bool     Chase;
     bool     Flock;
     bool     Evade;
     double   Inputs[4];
};
```

另外注意到，我们也加了一个 Inputs 向量。当用神经网络决定该单位要采取什么行动时，就可以储存输入值。

回到例 14-24 的 Initialize()方法，当单位都初始设定好之后，再来处理的就是 TheBrain。要做的第一件事是调用 Initialize()方法设定神经网络，传递代表每层神经元数量的值。就此例而言，我们有四个输入神经元、三个隐匿神经元以及三个输出神经元。这个网络类似图 14-4 的图示。

接着，下一件事是把学习率设成 0.2。我们通过试误法调整此值，目的是控制住训练时间，同时确保有一定精确度。然后，调用 SetMomentum()方法，表示我们想在训练时使用动量，再把动量系数设为 0.9。

现在，网络已初始设定好了，可以调用 TrainTheBrain()函数加以训练。例 14-26 是 TrainTheBrain()函数。

例 14-26：TrainTheBrain()函数

```
void     TrainTheBrain(void)
{
     int           i;
     double        error = 1;
     int           c = 0;

     TheBrain.DumpData("PreTraining.txt");

     while((error > 0.05) && (c<50000))
     {
          error = 0;
          c++;
          for(i=0; i<14; i++)
          {
               TheBrain.SetInput(0, TrainingSet[i][0]);
               TheBrain.SetInput(1, TrainingSet[i][1]);
```

```
                TheBrain.SetInput(2, TrainingSet[i][2]);
                TheBrain.SetInput(3, TrainingSet[i][3]);

                TheBrain.SetDesiredOutput(0, TrainingSet[i][4]);
                TheBrain.SetDesiredOutput(1, TrainingSet[i][5]);
                TheBrain.SetDesiredOutput(2, TrainingSet[i][6]);

                TheBrain.FeedForward();
                error += TheBrain.CalculateError();
                TheBrain.BackPropagate();

            }
            error = error / 14.0f;
        }

        TheBrain.DumpData("PostTraining.txt");
}
```

开始训练网络之前，先把网络的数据放到某个文字文件，如此一来，调试时才能予以参考。接着，进入一个while循环，使用倒传递算法训练网络。while循环会一直运行，直到计算的误差小于某指定值，或者直到饶行次数达到指定的最大阈值。后者的情况是避免while循环，因为误差标准达不到而永远在那里重复循环。

具体讲while循环之前，先看用来训练网络的训练数据。名为TrainingSet的全局数组用来储存训练数据。例14-27是此训练资料。

例14-27：训练资料

```
double TrainingSet[14][7] = {
//# 友军 , 容许击中次数 , 敌人是否交战 , 距离 , 追逐 , 群聚 , 闪躲
  0,        1,          0,            0.2,    0.9,    0.1,    0.1,
  0,        1,          1,            0.2,    0.9,    0.1,    0.1,
  0,        1,          0,            0.8,    0.1,    0.1,    0.1,
  0.1,      0.5,        0,            0.2,    0.9,    0.1,    0.1,
  0,        0.25,       1,            0.5,    0.1,    0.9,    0.1,
  0,        0.2,        1,            0.2,    0.1,    0.1,    0.9,
  0.3,      0.2,        0,            0.2,    0.9,    0.1,    0.1,
  0,        0.2,        0,            0.3,    0.1,    0.9,    0.1,
  0,        1,          0,            0.2,    0.1,    0.9,    0.1,
  0,        1,          1,            0.6,    0.1,    0.1,    0.1,
  0,        1,          0,            0.8,    0.1,    0.9,    0.1,
  0.1,      0.2,        0,            0.2,    0.1,    0.1,    0.9,
  0,        0.25,       1,            0.5,    0.1,    0.1,    0.9,
  0,        0.6,        0,            0.2,    0.1,    0.1,    0.9
};
```

训练数据中包含14组输入和输出值。每组数据有四个代表该单位的友军数量、其容许击中次数、敌人是否交战以及和敌人之间距离的输入节点，此外，还有三个输出节点的数据，对应追逐、群聚和闪躲的行为。

注意到，所有数据值都位于0.0～1.0之间。如前所述，所有输入数据都调整成在0.1～1.0的伸缩范围内，因为用的是logistic函数，所以每个输出值的范围也在0.0～1.0之间。稍后会谈到怎么增减输入数据。作为输出值，要得到0.0或1.0是不切实际的，所以，我们以0.1表示无活化之输出值，而以0.9表示活化之输出值。此外，注意到，这些输出值代表的是相对应的那组输入数据所想要的输出值。

这组训练资料的选法全凭经验。基本而言，我们任意假定了一些输入值情况，然后，指出该输入值情况，应该有什么合理的响应，并据此设定输出值。实际上，你可能会考虑得更周详，可能会用更多的训练数据集，不像我们替此简例所做的那样。

现在，回到例14-26，处理倒传递训练的`while`循环。进入`while`循环时，误差设成0。我们打算每轮都计算该误差，而总共有14组输入和输出值。就每组数据而言，要设定输入神经元的值以及所想要的输出神经元的值，然后调用网络的`FeedForward()`方法；之后，就能计算误差了。要计算误差，需要调用网络的`CalculateError()`方法，在误差变量中累加结果。接着，调用`BackPropagate()`方法调整连接的权重值。就一轮循环而言，这些步骤都完成后，就能求出这一轮的均方误差，也就是把误差除以14（14是每轮中数据集的数量）。训练完成后，网络的数据会被放到某个文字文件，以备稍后使用。

此时，神经网络已准备好。可以直接使用训练后的连接权重值。这样可以让你免去写有限状态机或类似方法的程序代码，处理所有可能的输入情况。这个网络比较引人注目的应用是可以实时学习。如果单位根据网络的决策而表现得很好，我们就可以强化该行为。另一方面，如果单位表现不佳，我们就可以重新训练网络，压抑不良决策。

学习

本节我们要继续探讨实现此神经网络的程序代码，包括使用倒传递算法在游戏中学习的能力。看一下例14-28的`UpdateSimulation()`函数。你会发现这是第四章讨论过的`UpdateSimulation()`函数的修改版。为了清晰起见，例14-28只列出此函数的修改部分。

例14-28：修改后的UpdateSimulation()函数

```
void     UpdateSimulation(void)
{
     .
     .
     .
     int          i;
     Vector       u;
     bool         kill = false;
```

```
.
.
.
// 计算当前和目标物正在交战的敌军单位数量
Vector d;
Units[0].NumFriends = 0;
for(i=1; i<_MAX_NUM_UNITS; i++)
{
     d = Units[i].vPosition - Units[0].vPosition;
     if(d.Magnitude() <= (Units[0].fLength *
                          _CRITICAL_RADIUS_FACTOR))
          Units[0].NumFriends++;
}

// 减少目标物的容许击中次数
if(Units[0].NumFriends > 0)
{
     Units[0].HitPoints -= 0.2 * Units[0].NumFriends;
     if(Units[0].HitPoints < 0)
     {
          Units[0].vPosition.x = _WINWIDTH/2;
          Units[0].vPosition.y = _WINHEIGHT/2;
          Units[0].HitPoints = _MAXHITPOINTS;
          kill = true;
     }
}

// 更新计算机控制单位
for(i=1; i<_MAX_NUM_UNITS; i++)
{
     u = Units[0].vPosition - Units[i].vPosition;
     if(kill)
     {
          if((u.Magnitude() <= (Units[0].fLength *
                                _CRITICAL_RADIUS_FACTOR)))
          {
               ReTrainTheBrain(i, 0.9, 0.1, 0.1);
          }
     }

     // 处理单位的容许击中次数，必要时学习
     if(u.Magnitude() <= (Units[0].fLength *
                          _CRITICAL_RADIUS_FACTOR))
     {
          Units[i].HitPoints -= DamageRate;
          if((Units[i].HitPoints < 0))
          {
            Units[i].vPosition.x=GetRandomNumber(_WINWIDTH/2
                                    -_SPAWN_AREA_R,
                                    _WINWIDTH/2+_SPAWN_AREA_R,
                                    false);
```

```
            Units[i].vPosition.y=GetRandomNumber(_WINHEIGHT/2
                                 -_SPAWN_AREA_R,
                                 _WINHEIGHT/2+_SPAWN_AREA_R,
                                 false);
            Units[i].HitPoints = _MAXHITPOINTS/2.0;
            ReTrainTheBrain(i, 0.1, 0.1, 0.9);
        }
    } else {
        Units[i].HitPoints+=0.01;
        if(Units[i].HitPoints > _MAXHITPOINTS)
            Units[i].HitPoints = _MAXHITPOINTS;
    }

    // 取得新命令
    Units[i].Inputs[0] = Units[i].NumFriends/_MAX_NUM_UNITS;
    Units[i].Inputs[1] = (double) (Units[i].HitPoints/
                                   _MAXHITPOINTS);
    Units[i].Inputs[2] = (Units[0].NumFriends>0 ? 1:0);
    Units[i].Inputs[3] = (u.Magnitude()/800.0f);

    TheBrain.SetInput(0, Units[i].Inputs[0]);
    TheBrain.SetInput(1, Units[i].Inputs[1]);
    TheBrain.SetInput(2, Units[i].Inputs[2]);
    TheBrain.SetInput(3, Units[i].Inputs[3]);
    TheBrain.FeedForward();

    Units[i].Command = TheBrain.GetMaxOutputID();
    switch(Units[i].Command)
    {
        case 0:
            Units[i].Chase = true;
            Units[i].Flock = false;
            Units[i].Evade = false;
            Units[i].Wander = false;
            break;

        case 1:
            Units[i].Chase = false;
            Units[i].Flock = true;
            Units[i].Evade = false;
            Units[i].Wander = false;
            break;

        case 2:
            Units[i].Chase = false;
            Units[i].Flock = false;
            Units[i].Evade = true;
            Units[i].Wander = false;
            break;
    }

    DoUnitAI(i);
    .
    .
    .
```

```
    } // i循环结束

    kill = false;
    .
    .
    .
}
```

这个修改过的UpdateSimulation()函数，所做的第一件事就是，计算当前正和目标物交战的计算机控制单位数量。就我们的简例而言，如果某单位和目标物处在指定距离内，就被视为和目标物正在交战。

一旦我们求出交战中单位的数量，就根据该数量的大小，按比例减去目标物的容许击中次数。如果目标物的容许击中次数变为零，则目标物就被视为已死，并在屏幕的中间重生。此外，kill标号也会设为true。

下一步是处理计算机控制单位。就此工作而言，我们进入一个for循环，轮流运行所有计算机控制单位。进入该循环时，计算出当前单位和目标物之间的距离。接着，检查目标物是否已死。如果是，就检查当前单位和目标物之间的位置关系，即是否在交战范围内。如果是，就重新训练神经网络以强化这种追逐行为。本质上，如果该单位和目标物交战，而目标物死了，假定该单位正在做某些正确的事，我们强化追逐行为，使其更具攻击性。

例14-29的函数负责重新训练网络。

例14-29：ReTrainTheBrain()函数

```
void     ReTrainTheBrain(int i, double d0, double d1, double d2)
{
    double     error = 1;
    int          c = 0;

    while((error > 0.1) && (c<5000))
    {

        c++;
        TheBrain.SetInput(0, Units[i].Inputs[0]);
        TheBrain.SetInput(1, Units[i].Inputs[1]);
        TheBrain.SetInput(2, Units[i].Inputs[2]);
        TheBrain.SetInput(3, Units[i].Inputs[3]);
        TheBrain.SetDesiredOutput(0, d0);
        TheBrain.SetDesiredOutput(1, d1);
        TheBrain.SetDesiredOutput(2, d2);
        //TheBrain.SetDesiredOutput(3, d3);

        TheBrain.FeedForward();
        error = TheBrain.CalculateError();
        TheBrain.BackPropagate();
```

```
    }
}
```

ReTrainTheBrain()只是再次操作倒传递训练算法而已，但这次是以该单位的输入值，以及指定的目标物输出值作为训练数据。注意到，不要把while循环的最大绕行次数阈值设得太高。如果设太高，当新训练过程开始时，动作中或许有明显的暂停现象。此外，如果试着重新训练网络以得到很小的误差，网络就变得适应得太快了。你可以改变误差和最大循环次数阈值，以控制网络适应的速度。

UpdateSimulation()函数的下一步是处理当前单位的容许击中次数。如果当前单位处在目标物的交战范围内，就把该单位预设的容许击中次数减少。如果该单位的容许击中次数变为零，即假定其死亡，此时，就令其在随机之处重生。我们也认定该单位正在做什么蠢事，所以，重新训练该单位要闪躲，不要追逐。

现在，继续往下看，我们要使用神经网络替该单位做决策，也就是说，在当前条件下，该单位应该追逐、群聚或闪躲。首先，要把输入值提供给神经网络。第一个输入值是当前单位的友军数量。把友军数量调整成可伸缩的值，也就是除以那些单位的最大数量。第二个输入值是该单位的容许击中次数，调整成可伸缩值的方式是，除以容许击中次数的最大值。第三个输入值是指出目标物是否在交战。如果目标物正在交战，此时就设为1.0，反之，就设为0.0。最后，第四个输入值是和目标物间的距离。就此而言，当前单位和目标物之间的距离，是调整成除以屏幕宽度（假设800个图素）的伸缩值。

所有输入值都设定好之后，再来调用FeedForward()传递网络。该方法调用之后，就能检视网络的输出值，以推导出正确的行为。就此而言，我们选择拥有最高活化值的输出，也就是调用GetMaxOutputID()方法所得的结果。然后，这个ID值会用在switch语句中，替此单位设定适当的行为标号。如果ID为0，该单位应该追逐。如果ID为1，该单位应该群聚。如果ID为2，则该单位应该闪躲。

以上就是修改后的UpdateSimulation()函数的内容。如果你运行此范例程序（可由本书网站下载：*http://www.oreilly.com/catalog/ai*），你会看见计算机控制单位的行为，实际上会随着仿真程序的运行而做出适应。你可以用数字键控制目标物对那些单位造成的伤害。1键是指轻微伤害或无伤害，而8键是指重度伤害。如果目标物死时，没有对那些单位造成什么伤害，你会发现，他们很快就会适应时常攻击。如果你让目标物调整成可以对那些单位造成重度伤害，你会看见那些单位开始适应，尽量避开目标物。他们也会开始时常以群体交战，而较少以单兵迎战。最后，他们会适应尽一切可能避开目标物。此例呈现的有趣行为是那些单位会形成群聚，而领头者会突显出来。通常会形成一个群聚，当中间或尾部的单位跟着领头者的时候，那些领头的单位也许会追逐或闪躲。

其他信息

如本章开头所述，神经网络这个主题太广，难以在一章交代完全。因此，我们要列出一份少量但非常不错的参考书单，当你决定继续研究下去时，就会发现这些书很有用。书单如下：

- 《Practical Neural Network Recipes in C++》（Academic Press 出版）
- 《Neural Networks for Pattern Recognition》（Oxford University Press 出版）
- 《AI Application Programming》（Charles River Media 出版）

此外还有很多讨论神经网络的书可以参考，然而，上面列出来的书我们认为很有帮助，尤其是第一本《Practical Neural Network Recipes in C++》。这本书有很多实用技巧和建议，都是有关神经网络的各种应用及其程序设计。

第十五章

遗传算法

替玩家做出有挑战性的游戏环境是游戏设计师的职责。事实上，游戏在开发时大部分都是在平衡这个游戏世界。游戏必须让玩家觉得有足够的难度，不然游戏看起来太简单，他们很快就会失去兴趣。另一方面，如果设计得太难了，玩家会受挫。有时候玩家会发现一些漏洞或窍门，说白了就是作弊。这可能是开发人员，处理某项游戏设计议题时有过失而引起的。另一个游戏设计问题是，来源于玩家的技巧水平的差异；要替不同技巧水平的玩家，做出一个真正平衡而又有挑战性的游戏，那真是令人感到胆怯的任务。所幸，遗传算法（genetic algorithm）可助一臂之力。

本章我们不打算模拟一个真正的基因系统。真正的基因模型对实际计算机游戏而言，可能不切实际，或者没有好处。相反的，我们打算让讨论的系统，只是受到生物基因系统的启发而已。两者在某些方面都很类似，但如果有助于游戏设计的流程，我们会毫不犹豫地改写规则。

在真实世界中，物种会不断演化，试着在其环境中很好地适应。这些物种都是最适宜继续存活下去的生物。1859年，达尔文在其著名的《物种起源》一书中提出这个规则。最能够在当下环境中生存下去的物种，就能将其特征传递给下一代。个体的特征都编写在染色体内。到了下一代，这些染色体会通过名为交叉（crossover）的过程，把染色体结合起来。交叉是后代重组染色体的方式。图15-1说明了此过程。

在图15-1中，我们使用随机字母来表示染色体。正如你所见，双亲中的每一位，都会将其一半基因传给子代。然而，在真实世界中，不见得会安分守己地走这种交叉的过程。随机突变也会发生，如图15-2所示。

随机突变是大自然尝试新事物的方式。如果某项随机突变改良了该物种，就会传给未来的后代。如果没有，就不会传递下去。

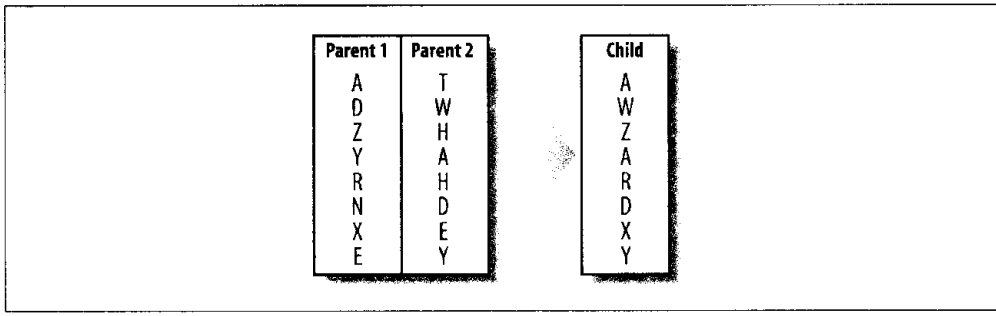


图 15-1：交叉

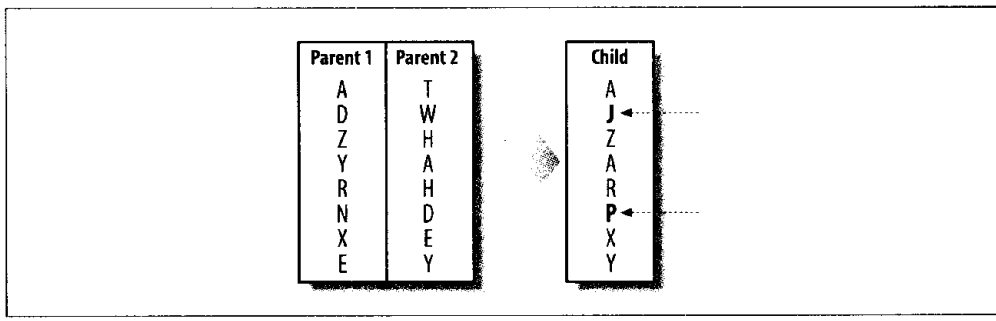


图 15-2：随机突变

上一代最优良的生物做这种固定的染色体重组，再配合随机突变，就能使生出的后代的生存适应能力更强，从而在其环境中繁衍不息。你可以在游戏中应用相同的概念。如同在生物世界中那样，游戏世界中的元素也可以演化，并适应变动的形势。

演化过程

你可以把游戏中遗传算法的操作方式分成四个步骤。图 15-3 说明了这四个步骤的过程。

如图 15-3 所示，第一步是建立第一代。整个族群会载入一组起始特征。一旦该族群开始和其环境互动后，必须有某种方式对个体做等级分类。这个过程就是适合度（fitness）分等。这将告诉我们族群中哪些个体最为优良。适合度分等的过程可在下一步选择（selection）时，协助我们。在选择过程中，要选出族群中的某些个体繁衍后代。本质上，要用上一代中最为优良的特征繁殖下一代。结合这些特征以生出适应力更强的下一代的实际过程，称为演化（evolution）。遗传算法实际上就是一种最优化过程，从中，我们试着找出适应力最强的一组特征，也就是说我们要找特定问题的最佳解决方案。

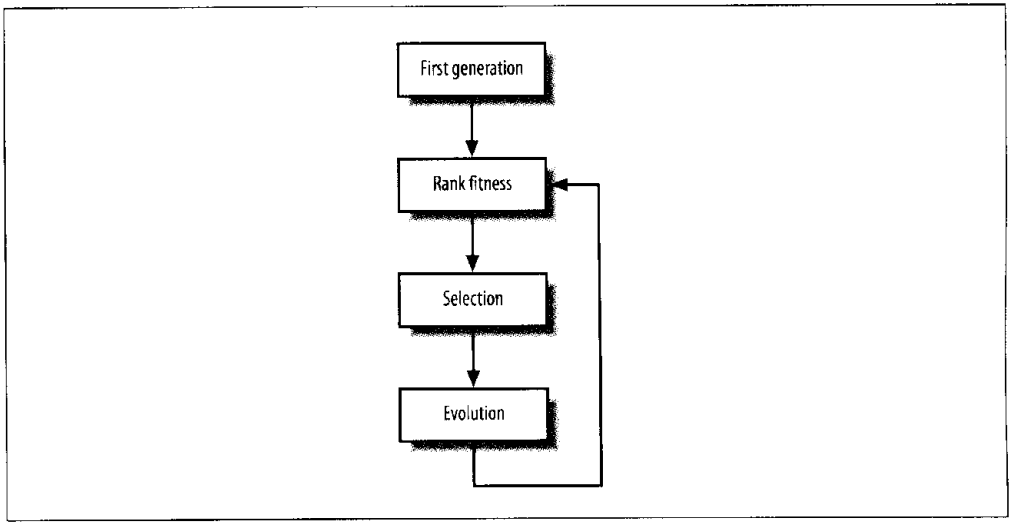


图 15-3：演化过程

第一代

第一代中的每个个体，都代表当下问题的可能解决方案之一。建立第一代的方法之一，是随机配置染色体。然而，在游戏环境中，随机配置染色体，恐怕不是最好的办法。如果游戏设计师已经知道哪种染色体的组合，最有可能生出的适应力强的个体，也许随机组合就不会用了。然而，一开始让族群个体多样化还是很重要的。如果个体彼此长得太像，基因过程就没什么用了。

编码是把染色体储存在计算机中某数据结构的过程。当然，程序员可以选用任何结构。遗传算法常用的是字符串，但数组、列表以及树结构也常用。

图 15-4 是花朵的第一代实例。这些假想的花，含有一些随机染色体，从而影响其在环境中的生长。

适合度分等

这一步在演化过程中是评估族群中的每一个个体。这里我们试着找出族群中最优良的个体，通常是用所谓的适合度函数（fitness function）完成工作。适合度函数的目的是替族群中的个体分等，继而得知，哪些个体是解决当下问题的最佳答案。

图 15-5 是花朵分等的结果。在此，假设长得最高的，就是最能适应的花朵。

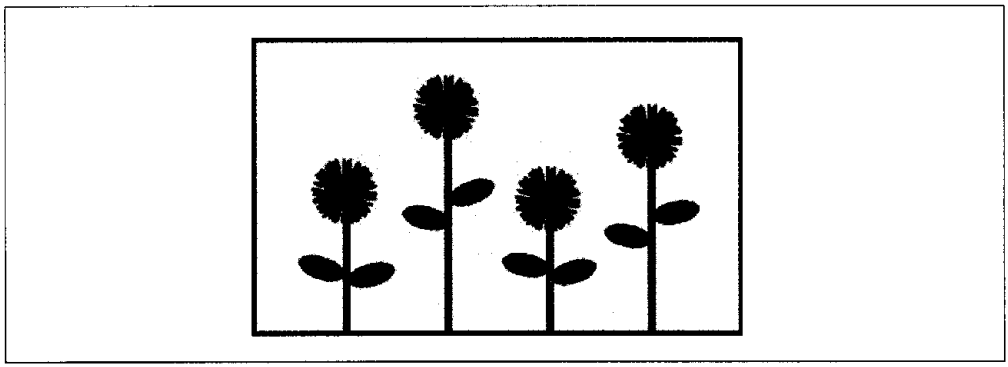


图 15-4：第一代

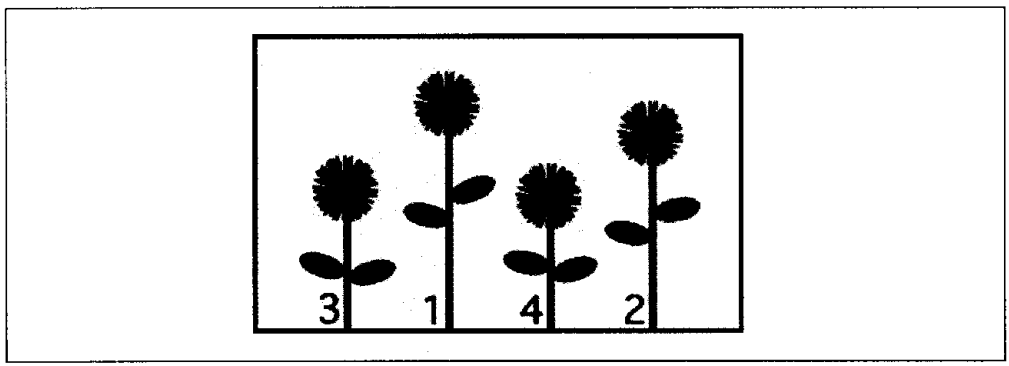


图 15-5：适合度分等

选择

在选择（selection）步骤中，我们要选出某些个体，使其特征能传递到下一代身上。在选择过程中，一般而言，我们是调用适合度函数，找出要用哪些个体繁殖下一代。在生物世界中，通常是父母双亲贡献染色体给后代。当然，在游戏开发中，双亲要如何组合都可以。例如，可以组合最优等的前两个、五个或十个个体的特征。

图 15-6 是利用适合度函数算出的分等情况，找出要用的个体产生下一代。就此例而言，就是选出两朵长得最高的花。

演化

最后一步是利用选择步骤建立新个体，放置到游戏环境里。我们取出族群里适合度最佳的个体的个别染色体，然后开始组合他们的染色体。此时，引入随机突变也很重要。一旦演化过程完成后，就再回到适合度分等的步骤。

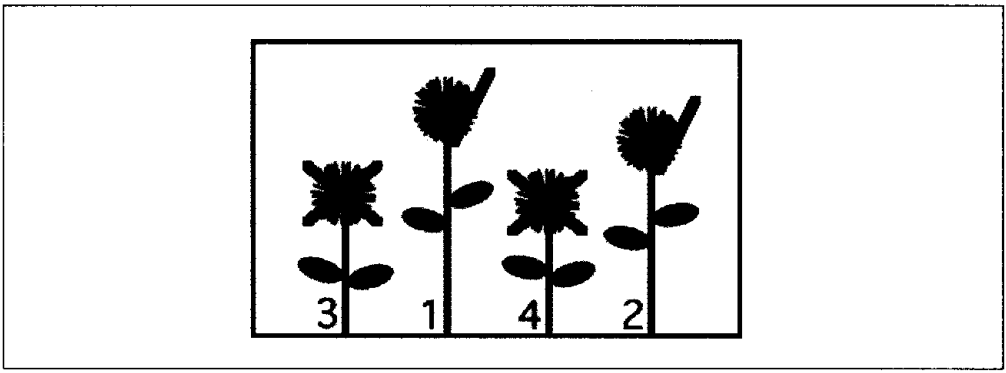


图 15-6：花朵选择

演化步骤就是交叉发生之处，也是组合适合度最佳个体染色体的地方。就此例而言，我们组合了两朵最高的花的染色体，建立新花朵，在此过程中也引入了两个随机突变。如图 15-7 所示。

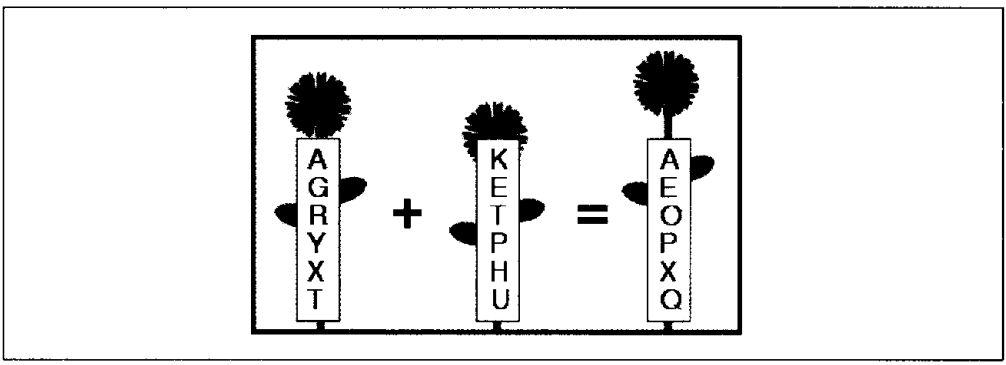


图 15-7：花朵演化

植物生命的演化

第一个实例是，如何应用遗传算法让花朵不断地繁衍后代，在其环境中生长下去。我们定义了一系列假设的环境条件，让花朵必须在其中生长。然后，每朵花都包含基因信息，表示出其理想的生长环境。花朵指出的理想生长环境和实际条件最接近的，就会长得最高。最高的花朵会被视为适合度最好，其基因信息就会传递给后代。这应该会让花朵繁衍后代时，在高度上不断地有所增长。

替花朵数据编码

我们一开始定义了六个假设的环境条件，视为花朵生长环境的实际条件，如例15-1所示。

例 15-1：编码

```
Class ai_World
{
public:

      int currentTemperature;
      int currentWater;
      int currentSunlight;
      int currentNutrient;
      int currentBeneficialInsect;
      int currentHarmfulInsect;

      ai_World();
      ~ai_World();
};
```

如例15-1所示，这六种情况是currentTemperature、currentWater、currentSunlight、currentNutrient、currentBeneficialInsect以及currentHarmfulInsect（目前温度、水质、阳光、养分、益虫、害虫）。

编码是把染色体储存在计算机结构中的过程。当然，程序员可以选择任何结构。例15-2是我们用在花朵演化实例中的结构。

例 15-2：条件

```
#define kMaxFlowers   11

Class ai_World
{
public:

      int temperature[kMaxFlowers];
      int water[kMaxFlowers];
      int sunlight[kMaxFlowers];
      int nutrient[kMaxFlowers];
      int beneficialInsect[kMaxFlowers];

      int harmfulInsect[kMaxFlowers];
      int currentTemperature;
      int currentWater;
      int currentSunlight;
      int currentNutrient;
      int currentBeneficialInsect;
      int currentHarmfulInsect;

      ai_World();
      ~ai_World();
};
```

如例 15-2 所示，我们用六个数组表示六种环境条件，包括 temperature、water、sunlight、nutrient、beneficialInsect 以及 harmfulInsect。每个条件都含有一个染色体，指出每朵花的理想情况。

第一代花朵

如同所有遗传算法一样，首先必须在这个世界中填入最初的一代。如果把遗传过程，当作搜寻问题的最佳解决方案，第一代就必须由所能猜测出的解法集组成。同时也必须确保，拥有各种组合的可行方案。例 15-3 是建立第一代的程序代码。

例 15-3：第一代花朵

```
void ai_World::Encode(void)
{
   int i;

   for (i=1;i<kMaxFlowers;i++)
       {
          temperature[i]=Rnd(1,75);
          water[i]=Rnd(1,75);
          sunlight[i]=Rnd(1,75);
          nutrient[i]=Rnd(1,75);
          beneficialInsect[i]=Rnd(1,75);
          harmfulInsect[i]=Rnd(1,75);
       }

    currentTemperature=Rnd(1,75);
    currentWater=Rnd(1,75);
    currentSunlight=Rnd(1,75);
    currentNutrient=Rnd(1,75);
    currentBeneficialInsect=Rnd(1,75);
    currentHarmfulInsect=Rnd(1,75);
}
```

如例 15-3 所示，一开始是随机替花朵的染色体编码。我们以六个数组表示花朵族群中，每个成员的理想生长条件。这些数组有 temperature、water、sunlight、nutrient、beneficialInsect 以及 harmfulInsect。每个数组的值介于 1～75 之间，这个范围是根据此例调整出来的。小于 75 的值，会让演化过程快一点。然而，如果只演化了几代，可能没什么好观察的。同样的，使用大一点的值，演化过程会变慢，找到最佳解决方案之前，需要多传几代。

当实际条件最接近编写在花朵染色体内理想的生长条件时，花朵长得最好。我们用 for 循环设定每个数组中的随机值。这样可以确保花朵族群足够多样化。一旦 for 循环执行完成后，就给当前条件赋值，包括 currentTemperature、currentWater、currentSunlight、currentNutrient、currentBeneficialInsect 以及 currentHarmfulInsect。

花朵适合度分等

就此基因仿真程序的目的而言，假设适合度最高的花朵，就是在当前环境条件下最茁壮的花朵。个别花朵里的染色体资料，编入了各自理想的生长条件。本质上，要估算每朵花的理想条件，和实际条件有多接近。那些条件最接近的花，长得最高，如图15-8所示。

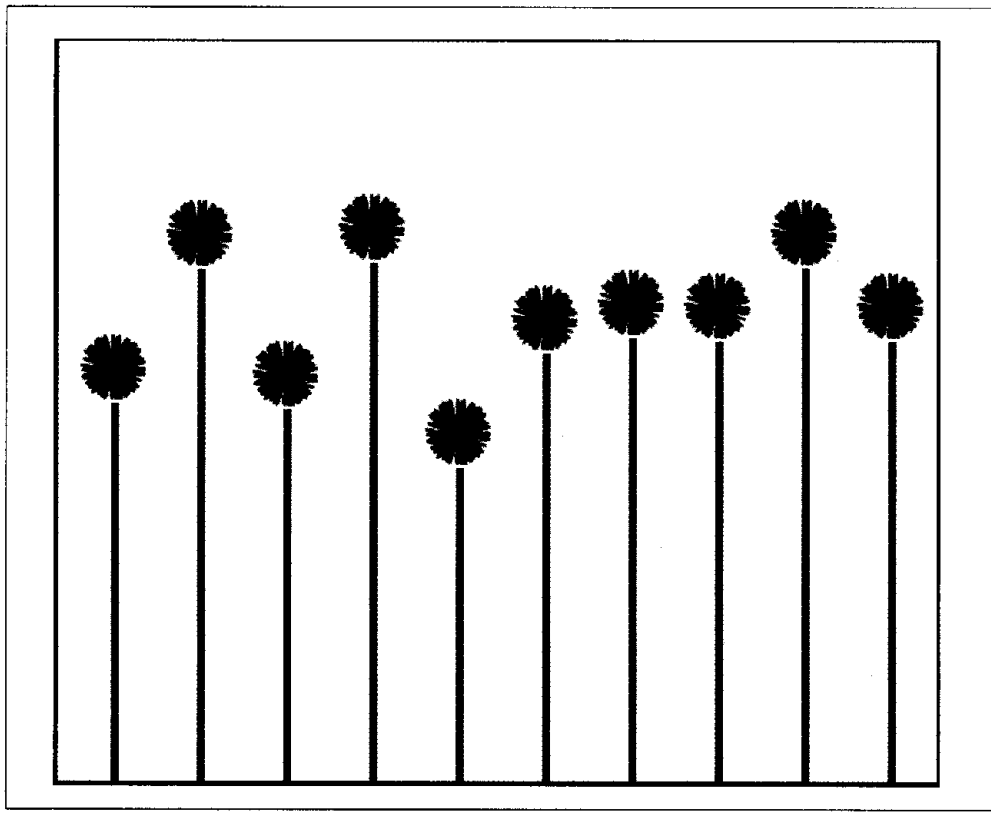


图15-8：初始花朵族群

图15-8是花群中最初的差异。在当前条件下最适合生长的，就是长得最高的。如图所示，某些花在此环境中长得特别的好。接着，要如何实际找出族群中，适合度最佳的成员，则要用如例15-4所示的函数。

例15-4：花朵适合度函数

```
int ai_World::Fitness(int flower)

{
     int theFitness=0;
     theFitness = fabs(temperature[flower] - currentTemperature);
```

```
    theFitness = theFitness+fabs(water[flower] - currentWater);
    theFitness = theFitness+fabs(sunlight[flower] -
                                 currentSunlight);
    theFitness = theFitness+fabs(nutrient[flower] -
                                 currentNutrient);
    theFitness = theFitness+fabs(beneficialInsect[flower] -
                                 currentBeneficialInsect);
    theFitness = theFitness+fabs(harmfulInsect[flower] -
                                 currentHarmfulInsect);

    return (theFitness);
}
```

如例 15-4 所示，我们用 Fitness()函数计算当前环境条件与每朵花要茁壮生长的理想条件之间的总偏差量。一开始，把 theFitness 变量设为 0。然后，增加 theFitness 的值，也就是把每朵花的理想条件和当前条件之间的差值的绝对值加上去。这样就能得到所有生长条件的总偏差量了。

花朵演化

任何遗传算法的最终目标，都是繁殖出比其前代适应力更强的后代。第一步是建立最初的族群，然后求出每个个体的适合度。适合度分等过程中，可以选出族群中最佳的成员。最后一步是使用前代最优等的特征，实际建立后代。除了让适合度最佳花朵的特征能交叉之外，还引入了随机突变。例 15-5 的 Evolve()函数有交叉和随机突变两个步骤。

例 15-5：花朵演化

```
void ai_World::Evolve(void)

{
    int fitTemperature[kMaxFlowers];
    int fitWater[kMaxFlowers];
    int fitSunlight[kMaxFlowers];
    int fitNutrient[kMaxFlowers];
    int fitBeneficialInsect[kMaxFlowers];
    int fitHarmfulInsect[kMaxFlowers];
    int fitness[kMaxFlowers];
    int i;
    int leastFit=0;
    int leastFitIndex;

    for (i=1;i<kMaxFlowers;i++)
        if (Fitness(i)>leastFit)
            {
               leastFit=Fitness(i);
               leastFitIndex=i;
            }

    temperature[leastFitIndex]=temperature[Rnd(1,10)];
    water[leastFitIndex]=water[Rnd(1,10)];
```

```
    sunlight[leastFitIndex]=sunlight[Rnd(1,10)];
    nutrient[leastFitIndex]=nutrient[Rnd(1,10)];
    beneficialInsect[leastFitIndex]=beneficialInsect[Rnd(1,10)];
    harmfulInsect[leastFitIndex]=harmfulInsect[Rnd(1,10)];

    for (i=1;i<kMaxFlowers;i++)
        {
           fitTemperature[i]=temperature[Rnd(1,10)];
           fitWater[i]=water[Rnd(1,10)];
           fitSunlight[i]=sunlight[Rnd(1,10)];
           fitNutrient[i]=nutrient[Rnd(1,10)];
           fitBeneficialInsect[i]=beneficialInsect[Rnd(1,10)];
           fitHarmfulInsect[i]=harmfulInsect[Rnd(1,10)];
        }

    for (i=1;i<kMaxFlowers;i++)
        {
           temperature[i]=fitTemperature[i];
           water[i]=fitWater[i];
           sunlight[i]=fitSunlight[i];
           nutrient[i]=fitNutrient[i];
           beneficialInsect[i]=fitBeneficialInsect[i];
           harmfulInsect[i]=fitHarmfulInsect[i];
        }

    for (i=1;i<kMaxFlowers;i++)
         {
            if (tb_Rnd(1,100)==1)
               temperature[i]=Rnd(1,75);
            if (tb_Rnd(1,100)==1)
               water[i]=Rnd(1,75);
            if (tb_Rnd(1,100)==1)
               sunlight[i]=Rnd(1,75);
            if (tb_Rnd(1,100)==1)
               nutrient[i]=Rnd(1,75);
            if (tb_Rnd(1,100)==1)
               beneficialInsect[i]=Rnd(1,75);
            if (tb_Rnd(1,100)==1)
               harmfulInsect[i]=Rnd(1,75);
        }

}
```

也可以用很多方式实现交叉函数。游戏开发人员不必受生物世界的局限性的制约。在生物世界中，交叉牵涉到适应双亲的染色体。在游戏开发中，交叉可以牵涉到任何数目的父母。就此花朵演化实例而言，要找出族群中最少适合度的成员。第一个 for 循环调用 Fitness()函数，以找出族群中最少适合度的成员。然后，把最少适合度花朵的特征，重新赋给花群中随机成员的特征。接下来两个 for 循环随机混合花群的特征。本质上，已重新赋给最少适合度花朵的特征，所以此时，花群整体应该比上一代要进步。可惜的是，个别特征不会比上一代更好，因为传递的还是相同的特征。现在需要一种方式试着

去超越前代。我们的做法是随机突变。在最后一个for循环中，每朵花的每个特征有1%的随机突变机会。如果突变成功，则其特征可能传给下一代。如果突变结果使得某朵花，变成该族群中最少适合度的成员，将被淘汰。图15-9是几代演化后的结果。

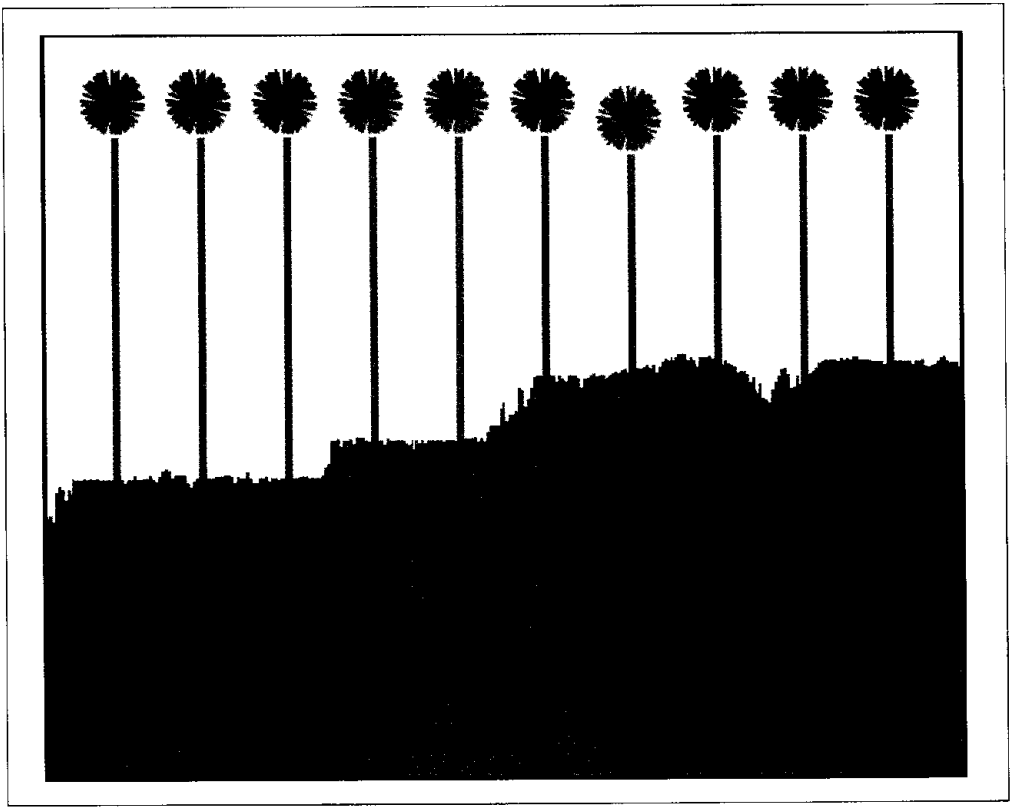


图15-9：花朵族群演化的结果

如图15-9所示，所有花朵都长到最大高度处或者很接近该处。花朵下面的区域也画出了每一代的适合度。如图所示，每一代的适合度大致呈向上的趋势。然而，并非每一代都比上一代进步。图中显示也有向下的时候。这是因为在族群中引入了随机突变。然而，如图所示，基因过程最终会找到问题的最佳解决方案。

遗传在游戏开发中的应用

对游戏而言，遗传算法只是一种寻找问题的最佳解决方案的方法。当然，游戏软件AI中有很多问题要解决，但并非所有问题都适用遗传算法。例如，寻找路径也能用遗传算法

解决，然而，这种问题通常用诸如A*算法的方法解决更合适。当问题的因素有某些不可预测性时，遗传算法最合适不过了。这可以让游戏软件AI，适应游戏设计师没有办法预测的情况。游戏设计时，游戏环境中最不可预测的因素就是玩家。就某种程度而言，游戏设计师必须要预测玩家的行为，以建立有挑战性的敌人。可惜的是，实际很难预测并响应每个可能的玩家行为。

基本而言，遗传算法牵涉到试误法。本质上，你让游戏世界有好多种可能解法，然后找出那些表现得最好的解法。当然，解法并非对每位玩家都是一成不变的。这正是遗传算法的优势所在。游戏软件AI能去适应各个玩家。

角色扮演实例

无论什么场景，只要计算机控制的敌方，必须根据玩家的行为而响应并修改其响应方式，遗传算法就有其用处。例如，考虑一个假想的多人角色扮演游戏。在此游戏中，玩家可以从多种角色类型和能力中选取一种角色。这也就是说，计算机控制的角色，必须对各种玩家控制的角色展现游戏的挑战性。玩家可能选择当战士，主要以暴力对打。当然，当这类角色出现时，玩家也可以从一堆武器中择一使用。玩家可以用剑、斧或其他任何武器进行攻击。斗士这种角色也可以穿上某种盔甲。另一方面，玩家可能选择另一种完全不同的角色类型。当法师的玩家，其行为会全然不同。角色类型和武器种类的各种组合，会让游戏设计师难以建立单一计算机控制的个体，让每种玩家控制角色都感受到挑战性。在多人游戏中，情况甚至更复杂。在这类游戏里，计算机必须对一群四处分散，但联手合作的玩家展现挑战性。可能的组合数量之多马上就让游戏设计师难以应对。

数据编码

我们搜寻某种计算机控制角色，让某位玩家或一群玩家感受到挑战性，但这种搜寻无法事先计算。每位玩家或一群玩家的行为都不同，必须确认某些情况和响应，会增加或减少族群中的适合度。例如，可能的情况之一是玩家以法器攻击计算机控制角色。对于玩家的行动，可以建立好几种可能的响应。可以用攻击玩家、试图逃跑或试图隐藏作为响应。我们可以把这种情况和响应赋给某个染色体。如果此染色体被设定成在此情况下做攻击响应，则玩家使用法器时将被攻击。然而，如果计算机控制的角色总是被打败呢？就此而言，计算机控制的角色会被视为不适应，因此，不可能把其特征传递给下一代。几代传下来，这种场景将产生不同的行为，比如当玩家手拿法器时，这些角色就会撤退。例15-6列出了在此假想例子中可能碰到的某些场景。

例 15-6：可能的场景

```
#define kAttackedbyFighter                  0
#define kAttackedbyWizard                   1
#define kAttackedbyGroup                    2
#define kHealerPresent                      3
#define kAttackedByBlade                    4
#define kAttackedByBlunt                    5
#define kAttackedByProjectile               6
#define kAttackedByMagic                    7
#define kAttackerWearingMetalArmor          8
#define kAttackerWearingLeatherArmor        9
#define kAttackerWearingMagicArmor          10
#define kImInGroup                          11
```

例15-6是一些可能的情况，族群的成员会有不同的行为。我们以不同的染色体，储存每种情况的响应。先从kAttackedbyFighter常量开始。这个染色体会储存计算机控制角色被玩家控制的斗士攻击时的响应。和斗士角色交战时，某些生物类型也许比较有效或比较无效。同样的，kAttackedbyWizard染色体会储存被玩家控制的法师攻击时的响应。kAttackedbyGroup染色体储存被一群玩家攻击时的响应行为。kHealerPresent染色体指出当有治疗能力的玩家出现时，应该采取的行为。如果玩家可以不断地被治疗，这种情况可能没什么希望获胜，撤退反而是最适当的。

下面四个染色体分别是kAttackedByBlade、kAttackedByBlunt、kAttackedByProjectile以及kAttackedByMagic，可以根据玩家手拿的武器的类型求出响应。下面三个是kAttackerWearingMetalArmor、kAttackerWearingLeatherArmor以及kAttackerWearingMagicArmor，可求出在玩家穿戴不同防御盔甲时的最佳响应。某些计算机控制角色可以选武器。例如，刀可能有助于攻击皮革盔甲。最后一个染色体是kImInGroup，用于求出当计算机控制角色和一个群体对战时，所做的最佳响应。某些类型的生物集体战斗时比较有效，比如狼群。

实际的游戏比起这里列出的可能场景要多得多，但就此例而言，我们列出的清单应该足够了。例 15-7 是每种场景的可能响应。

例 15-7：可能的行为

```
#define kRetreat                0
#define kHide                   1
#define kWearMetalArmor         2
#define kWearMagicArmor         3
#define kWearLeatherArmor       4
#define kAttackWithBlade        5
#define kAttackWithBlunt        6
#define kAttackWithMagic        7
```

例15-7的每个常量定义了可能的角色行为，每个计算机控制的角色都会根据例15-6的每个场景，赋给某种行为。我们从kRetreat常量开始。如其名称所暗示的，这项行为使得计算机控制角色远离玩家角色。第二个常量kHide是让计算机控制角色进入一种试着避开侦测的状态。下面三个常量kWearMetalArmor、kWearMagicArmor以及kWearLeatherArmor让计算机控制角色分别切换成个别的盔甲。最后三个常量kAttackWithBlade、kAttackWithBlunt以及kAttackWithMagic定义的是计算机控制角色应该对玩家采取的攻击类型。

现在要建立一个数组，包含例15-6的每种可能场景的响应行为。每种可能的场景都有一个数组元素相对应。我们以简单的C++类结构来做这件事，如例15-8所示。

例15-8：*编码结构*

```
#define kChromosomes    12

Class ai_Creature
{
        public:

        int chromosomes[kChromosomes];

        ai_ Creature ();
        ~ai_ Creature ();
};
```

如例15-8所示，我们建立一个ai_Creature类，包含一个染色体数组。这个数组的每个元素，都代表计算机控制角色的一项特征或行为。先定义每个可能行为，然后，把每个染色体和每个行为连接起来。这里用了12个数组，是因为例15-6中有12种可能情况。

第一代

到目前为止，已经定义了用于基因测试的可能场景，也定义了每个场景相关的可能行为，而且建立了一个结构来储存基因数据。下一步是建立族群。我们的首要任务是扩充例15-8的程序代码，如例15-9所示的修改。

例15-9：*定义族群*

```
#define kChromosomes        12
#define kPopulationSize    100

Class ai_Creature
{
public:

        int chromosomes[kChromosomes];
```

```
        ai_ Creature ();
        ~ai_ Creature ();
};

ai_Creature   population[kPopulationSize];
```

如例15-9所示，新增了kPopulationSize常量，定义生物族群的数量。也增加了一个ai_Creature类型的数组，其阈值设成kPopulationSize之值。下一步是把个别行为和例15-6的每种可能场景连接起来。做法是新增一个新的函数到ai_Creature类中。

例15-10：定义createIndividual()

```
#define kChromosomes        12
#define kPopulationSize    100

Class ai_Creature
{
public:

      int chromosomes[kChromosomes];

      void createIndividual (int i);

      ai_ Creature ();
      ~ai_ Creature ();

};

ai_Creature   population[kPopulationSize];
```

新函数createIndividual()会替族群的新成员设初值。然而，我们不想用预定的一组常量，设定每个个别成员的初值，而是想尽可能让族群多样化。族群不够多样化，找出最佳解决方案的可行性降低。建立多样化族群的最佳方式，就是以随机方式给行为赋值。然而，不能直接把例15-7的行为赋给例15-6的各种情况。某些行为不适用列举的情况。我们的解决办法是利用条件语句，如例15-11所示。

例15-11：随机给染色体赋值

```
void ai_ Creature::createIndividual(int i)

{

   switch (Rnd(1,5)) {
       case 1:
          ai_Creature[i].chromosomes[kAttackedbyGroup]=kRetreat;
        break;

       case 2:
          ai_Creature[i].chromosomes[kAttackedbyGroup]=kHide;
       break;
```

```
        case 3:
           ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                kAttackWithBlade;
        break;

        case 4:
           ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                kAttackWithBlunt;
        break;

        case 5:
          ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                kAttackWithMagic;
        break;
        }

    switch (Rnd(1,5)) {
       case 1:
          ai_Creature[i].chromosomes[kHealerPresent]=kRetreat;
       break;

       case 2:
          ai_Creature[i].chromosomes[kHealerPresent]=kHide;
       break;

       case 3:
          ai_Creature[i].chromosomes[kHealerPresent]=
                                                kAttackWithBlade;
       break;

       case 4:
         ai_Creature[i].chromosomes[kHealerPresent]=
                                                kAttackWithBlunt;
     break;

     case 5:
        ai_Creature[i].chromosomes[kHealerPresent]=
                                                kAttackWithMagic;
     break;
     }
```

第一个switch语句，给kAttackedbyGroup染色体赋了一个随机行为。就此染色体而言，可从五种可能行为（kRetreat、kHide、kAttackWithBlade、kAttackWithBlunt以及kAttackWithMagic）中找出一种。目的是试着确认这些行为在一群玩家攻击计算机控制角色时，是否会出现显著的优点或缺点。例如，演化的结果也许显示出，被一群玩家攻击时，最佳行为应是撤退。

第二个switch语句是给kHealerPresent染色体赋值。同样的，要确认如果建立了一个多样化的族群，对此情况的响应会随成员而异，如此一来，是否会给某些个体带来优点或缺点。如同kAttackedbyGroup染色体的情况，用的是这五种响应：kRetreat、kHide、kAttackWithBlade、kAttackWithBlunt以及kAttackWithMagic。

现在，要考虑把可能的行为，连接到玩家能用的各种攻击类型。同样的，对每种可能攻击手法都随机赋予适当行为，如例15-12所示。

例15-12：攻击响应

```
switch (Rnd(1,5)) {
     case 1:
        ai_Creature[i].chromosomes[kAttackedByBlade]=kRetreat;
     break;

     case 2:
        ai_Creature[i].chromosomes[kAttackedByBlade]=kHide;
     break;

     case 3:
        ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                kWearMetalArmor;
     break;

     case 4:
        ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                kWearMagicArmor;
     break;

     case 5:
        ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                kWearLeatherArmor;
     break;
  }

  switch (Rnd(1,5)) {
     case 1:
        ai_Creature[i].chromosomes[kAttackedByBlunt]=kRetreat;
     break;

     case 2:
        ai_Creature[i].chromosomes[kAttackedByBlunt]=kHide;
     break;

     case 3:
        ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                          kWearMetalArmor;
     break;

     case 4:
        ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                          kWearMagicArmor;
     break;

     case 5:
        ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                          kWearLeatherArmor;
     break;
  }
```

```
    switch (Rnd(1,5)) {
       case 1:
          ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kRetreat;
       break;

       case 2:
          ai_Creature[i].chromosomes[kAttackedByProjectile]=kHide;
       break;
       case 3:
          ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kWearMetalArmor;
       break;

       case 4:
          ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kWearMagicArmor;
       break;

       case 5:
          ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kWearLeatherArmor;
       break;
    }

 switch (Rnd(1,5)) {
       case 1:
          ai_Creature[i].chromosomes[kAttackedByMagic]=kRetreat;
       break;

       case 2:
          ai_Creature[i].chromosomes[kAttackedByMagic]=kHide;
       break;

       case 3:
          ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                kWearMetalArmor;
       break;

       case 4:
          ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                kWearMagicArmor;
       break;

       case 5:
          ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                kWearLeatherArmor;
       break;
 }
```

如例15-12所示，我们考虑玩家的四种可能攻击类型（kAttackedByBlade、kAttackedByBlunt、kAttackedByProjectile以及kAttackedByMagic）。每种可能攻击都与其染色体相连，在个别的case语句中予以赋值。每种攻击都会随机赋予五种

可能响应中的其中一种：kRetreat、kHide、kWearMetalArmor、kWearMagicArmor以及kWearLeatherArmor。

这些染色体会协助确认，面对每种可能的玩家攻击，哪种盔甲最合适。此外，也会让我们知道撤退或隐藏，是不是某些攻击的最佳响应。例如，如果族群的成员被魔法攻击时，总是会败下阵来，最后，撤退可能就会变成最佳响应。

现在，要考虑玩家穿戴的各种可能盔甲，所造成的可能影响。我们采用类似的case语句结构，如例15-13所示。

例15-13：盔甲响应

```
switch (Rnd(1,5)) {
     case 1:
        ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                              kRetreat;
     break;

     case 2:
        ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                              kHide;
     break;

     case 3:
        ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                              kAttackWithBlade;
     break;

     case 4:
        ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                              kAttackWithBlunt;
    break;

    case 5:
       ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                             kAttackWithMagic;
    break;
}

switch (Rnd(1,5)) {
    case 1:
       ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                             kRetreat;
    break;

    case 2:
       ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                             kHide;
    break;
```

```
        case 3:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                    kAttackWithBlade;
         break;

         case 4:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                    kAttackWithBlunt;
         break;

         case 5:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                    kAttackWithMagic;
         break;
   }

   switch (Rnd(1,5)) {
      case 1:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                    kRetreat;
      break;

      case 2:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                    kHide;
      break;

      case 3:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                    kAttackWithBlade;
      break;

      case 4:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                    kAttackWithBlunt;
      break;

      case 5:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                    kAttackWithMagic;
      break;
   }
}
```

例15-13使用三个switch语句，针对玩家穿戴的各种盔甲，随机给响应赋值。希望这样可以帮助我们，找出针对各种盔甲的最佳攻击类型。可以考虑的盔甲有kAttackerWearingMetalArmor、kAttackerWearingLeatherArmor以及kAttackerWearingMagicArmor。每种盔甲都会被随机赋予五种可能响应中的其中一种，包括kRetreat、kHide、kAttackWithBlade、kAttackWithBlunt以及kAttackWithMagic。

适合度分等

到了某一时刻，必须试着找出族群中，适合度最高的那些成员。记住，要找出族群成员中，对玩家最有挑战性的成员，必须用某种方式，量化和评定挑战性的等级。我们可以考虑几种不同的方法。角色扮演游戏，通常会把容许击中次数赋给每个角色。当角色在战斗中受伤，容许击中次数就会降低。当容许击中次数变为零时，角色就会死亡。所以，量化挑战性等级的其中一种方法，就是记下对玩家造成损害的击中次数的累加次数。族群中的每位成员，都要记录其给予玩家损害的总击中次数。相反的，也要记下玩家对族群中成员所造成损害的次数。例15-14是扩充ai_Creature类，引入变量记录造成的损害以及受到的损害。

例15-14：记录损害的击中次数

```
#define kChromosomes        12
#define kPopulationSize    100

Class ai_Creature
{
public:

      int chromosomes[kChromosomes];
      float totalDamageDone;
      float totalDamageReceived;

      void createIndividual (int i);

      ai_ Creature ();
      ~ai_ Creature ();

};

ai_Creature   population[kPopulationSize];
```

如例15-14所示，在ai_Creature类里增加了两个新变量。第一个是totalDamageDone，每次计算机控制角色对玩家造成损害时，就会递增；造成损害的击中次数的总量算出时就会递增。相反的，当玩家损伤计算机控制角色时，totalDamageReceived变量就会递增。就totalDamageDone而言，递增的值等于损害的击中次数。

当然，当计算族群中个体的适合度时，还应该考虑其他游戏因素。例如，玩家杀死的敌人总数，也是一个不错的指标。

选择

演化过程的下一步是搜寻出族群里适合度最高的成员。这些个体会展示出要传给下一代的特征。同样的，要再次扩充ai_Creature类。我们要计算造成的损害和受到的损害的

比值。要在totalDamageDone和totalDamageReceived变量中持续记录，造成的损害和受到的损害。fitness变量会存储造成的损害和受到的损害的比值。例15-15是更新后的类。

例15-15：加上适合度记录

```
#define kChromosomes        12
#define kPopulationSize    100

Class ai_Creature
{
public:

      int chromosomes[kChromosomes];
      float totalDamageDone;
      float totalDamageReceived;
      float fitness;

      void createIndividual (int i);
      void sortFitness (void);

      ai_ Creature ();
      ~ai_ Creature ();

};

ai_Creature   population[kPopulationSize];
```

如例15-15所示，现在，我们有一个变量，可用来量化个体的实际适合度，就是使用fitness变量计算个体的适合度，然后，按最优等到最差等的次序排列族群。我们也在ai_Creature类中新增了sortFitness()函数。计算和排序如例15-16所示。

例15-16：适合度排序

```
void ai_ Creature:: sortFitness (void)

{
int   i
int   j;
int   k;
float temp;

for (i=0;i<kPopulationSize;i++)
      ai_Creature[i].fitness = ai_Creature[i].totalDamageDone /
                               ai_Creature[i].totalDamageReceived;

for (i = (kPopulationSize - 1); i >= 0; i--)
    {
       for (j = 1; j <= i; j++)
           {
             if (ai_Creature[j-1].fitness < ai_Creature[j].fitness)
                 {
```

```
                temp = ai_Creature[j-1].fitness;
                ai_Creature[j-1].fitness=ai_Creature[j].fitness;
                ai_Creature[j].fitness = temp;

                temp = ai_Creature[j-1].totalDamageDone;
                ai_Creature[j-1].totalDamageDone =
                                ai_Creature[j].totalDamageDone;
                ai_Creature[j].totalDamageDone = temp;

                temp = ai_Creature[j-1].totalDamageReceived;
                ai_Creature[j-1].totalDamageReceived =
                            ai_Creature[j].totalDamageReceived;
                ai_Creature[j].totalDamageReceived = temp;
                for (k=0;k<kChromosomes;k++)
                    {
                      temp = ai_Creature[j-1].chromosomes[k];
                      ai_Creature[j-1].chromosomes[k] =
                                    ai_Creature[j].chromosomes[k];
                      ai_Creature[j].chromosomes[k] = temp;
                    }
            }
        }
    }
```

sortFitness()函数一开始是计算族群中，个体造成的损害和受到的损害的比值，也就是在第一个for循环中完成。实际的比值会储存在fitness变量中。一旦算出个体的适合度比值，就能为整个族群数组排序。排序是利用for循环的嵌套结构完成的。这只是标准的起泡排序法的应用。最后的结果是利用fitness变量，排出整个族群的顺序，从最适存者排到最不适存者。

演化

现在，有一个简单的方式，可以找出族群中最优等的个体。调用sortFitness()函数可以确保ai_Creature数组中，较低的位置就是适合度最好的几个个体。然后，用低位数组元素的个体特征，产生下一代。图15-10说明了每个数组的染色体如何结合以生成新个体。

如图15-10所示，建立新个体时，采用交叉过程。现在，我们更新ai_Creature类来包含新的crossover()函数。例15-17是更新后的ai_Creature类。

例15-17：新增crossover()函数

```
#define kChromosomes        12
#define kPopulationSize    100

Class ai_Creature
{
public:
```

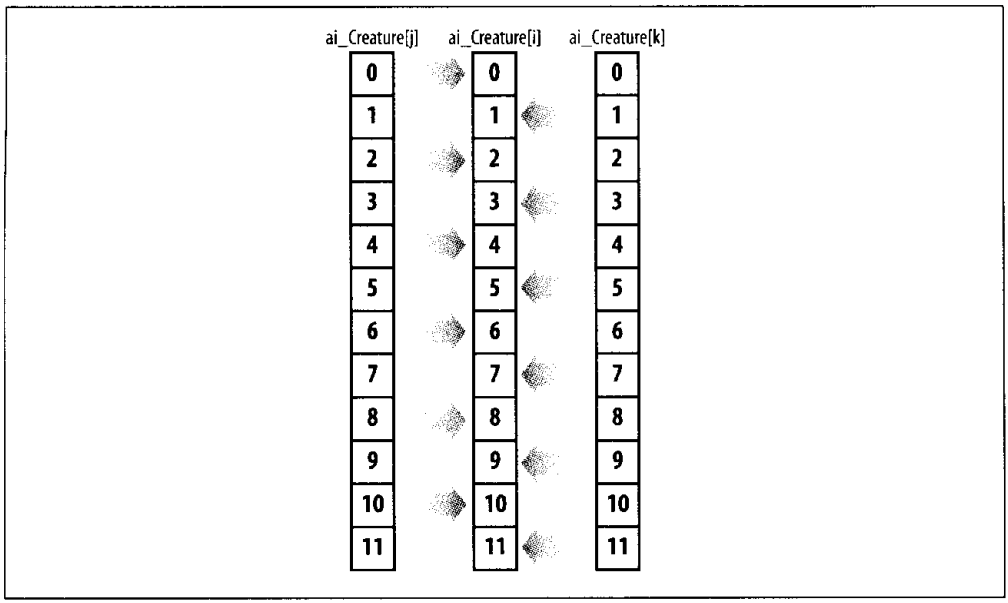


图 15-10：交叉

```
        int chromosomes[kChromosomes];
        float totalDamageDone;
        float totalDamageReceived;
        float fitness;

        void createIndividual (int i);
        void sortFitness (void);
        void crossover(int i, int j, int k);

        ai_ Creature ();
        ~ai_ Creature ();

};

ai_Creature   population[kPopulationSize];
```

新的crossover()函数会取两个个体的特征，使之结合以建立第三个个体。例15-18就是此函数。

例 15-18：crossover()函数

```
void ai_ Creature:: crossover (int i, int j, int k)

   {
        ai_Creature[i].chromosomes[0]=ai_Creature[j].chromosomes[0];
        ai_Creature[i].chromosomes[1]=ai_Creature[k].chromosomes[1];
        ai_Creature[i].chromosomes[2]=ai_Creature[j].chromosomes[2];
        ai_Creature[i].chromosomes[3]=ai_Creature[k].chromosomes[3];
```

```
        ai_Creature[i].chromosomes[4]=ai_Creature[j].chromosomes[4];
        ai_Creature[i].chromosomes[5]=ai_Creature[k].chromosomes[5];
        ai_Creature[i].chromosomes[6]=ai_Creature[j].chromosomes[6];
        ai_Creature[i].chromosomes[7]=ai_Creature[k].chromosomes[7];
        ai_Creature[i].chromosomes[8]=ai_Creature[j].chromosomes[8];
        ai_Creature[i].chromosomes[9]=ai_Creature[k].chromosomes[9];
        ai_Creature[i].chromosomes[10]=ai_Creature[j].chromosomes[10];
        ai_Creature[i].chromosomes[11]=ai_Creature[k].chromosomes[11];
        ai_Creature[i].totalDamageDone=0;
        ai_Creature[i].totalDamageReceived=0;
        ai_Creature[i].fitness=0;
    }
```

如例15-18所示，crossover()函数接收了三个变量，三个都是数组索引值。前两个是双亲，其染色体会结合起来建立新个体。第三个变量是新个体的数组索引值。每一行都在j和k的数组索引值上切换。本质上，建立后代时，会混合双亲的染色体。

虽然混合适存双亲的染色体，应该可以建立新的适存个体，我们也想在上一代的基础上做些改进。做法是引入随机突变。一开始是在ai_Creature类中引入随机突变函数，如例15-19所示。

例15-19：加入随机突变函数

```
#define kChromosomes        12
#define kPopulationSize    100

Class ai_Creature
{
public:

      int chromosomes[kChromosomes];
      float totalDamageDone;
      float totalDamageReceived;
      float fitness;

      void createIndividual (int i);
      void sortFitness (void);
      void crossover(int i, int j, int k);
      void randomMutation(int i);

      ai_ Creature ();
      ~ai_ Creature ();

};

ai_Creature   population[kPopulationSize];
```

如例15-19更新的ai_Creature类所示，现在必须新增一个随机突变函数。随机突变函数可以建立一个适存的个体，但这是猜测的，我们猜这会使其适存得更好。例15-20是此随机突变函数。

例 15-20：随机突变

```
void ai_ Creature::randomMutation(int i)

{
   if (Rnd(1,20)==1)
      switch (Rnd(1,5)) {
           case 1:
              ai_Creature[i].chromosomes[kAttackedbyGroup]=kRetreat;
           break;

           case 2:
              ai_Creature[i].chromosomes[kAttackedbyGroup]=kHide;
           break;

           case 3:
              ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                  kAttackWithBlade;
           break;

           case 4:
              ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                  kAttackWithBlunt;
           break;

           case 5:
             ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                 kAttackWithMagic;
           break;
           }

   if (Rnd(1,20)==1)
      switch (Rnd(1,5)) {
           case 1:
              ai_Creature[i].chromosomes[kHealerPresent]=kRetreat;
           break;

           case 2:
              ai_Creature[i].chromosomes[kHealerPresent]=kHide;
           break;

           case 3:
              ai_Creature[i].chromosomes[kHealerPresent]=
                                                  kAttackWithBlade;
           break;

           case 4:
              ai_Creature[i].chromosomes[kHealerPresent]=
                                                  kAttackWithBlunt;
           break;

           case 5:
              ai_Creature[i].chromosomes[kHealerPresent]=
                                                  kAttackWithMagic;
           break;
         }
```

```
if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
        case 1:
           ai_Creature[i].chromosomes[kAttackedByBlade]=kRetreat;
        break;

        case 2:
           ai_Creature[i].chromosomes[kAttackedByBlade]=kHide;
        break;

        case 3:
           ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                kWearMetalArmor;
        break;

        case 4:
           ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                kWearMagicArmor;
        break;

        case 5:
           ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                kWearLeatherArmor;
        break;
   }

if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
        case 1:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=kRetreat;
        break;

        case 2:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=kHide;
        break;

        case 3:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                kWearMetalArmor;
        break;

        case 4:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                kWearMagicArmor;
        break;

        case 5:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                kWearLeatherArmor;
        break;
   }

if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
        case 1:
```

```
            ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kRetreat;
         break;

         case 2:
            ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kHide;
         break;

         case 3:
            ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kWearMetalArmor;
         break;

         case 4:
            ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kWearMagicArmor;
         break;

         case 5:
            ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                kWearLeatherArmor;
         break;
   }

if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                kRetreat;
         break;

         case 2:
            ai_Creature[i].chromosomes[kAttackedByMagic]=kHide;
         break;

         case 3:
            ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                kWearMetalArmor;
         break;

         case 4:
            ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                kWearMagicArmor;
         break;

         case 5:
            ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                kWearLeatherArmor;
         break;
   }

if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
         case 1:
```

```
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                    kRetreat;
         break;

         case 2:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                    kHide;
         break;

         case 3:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                    kAttackWithBlade;
         break;

         case 4:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                    kAttackWithBlunt;
         break;

         case 5:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                    kAttackWithMagic;
         break;
   }

if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                    kRetreat;
         break;

         case 2:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                    kHide;
         break;

         case 3:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                    kAttackWithBlade;
         break;

         case 4:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                    kAttackWithBlunt;
         break;

         case 5:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                    kAttackWithMagic;
         break;
   }

if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
```

```
            case 1:
               ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                   kRetreat;
            break;

            case 2:
               ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                   kHide;
            break;

            case 3:
               ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                   kAttackWithBlade;
            break;

            case 4:
               ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                   kAttackWithBlunt;
            break;

            case 5:
               ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                   kAttackWithMagic;
            break;
      }
}
```

例15-20重新随机给染色体赋值。每个特征有5%的随机突变概率，做法是利用每行条件语句`if (Rnd(1,20)==1)`。就像`createIndividual()`函数那样，每个特征能赋的值有限。我们以`switch`语句保证，只有合法值才可以赋给每个特征。

你可以用其他方法在多人角色扮演游戏中整合遗传算法。上例主要集中在如何根据玩家的行为改变响应的行为，然而，游戏设计的其他领域也能得益于遗传算法。例如，角色扮演游戏通常会分类角色的能力，替每种能力赋予某点数的等级。巨人也许有100个点数机会，可以分散到好几种属性上，比如体力、法力、敏捷度以及抵抗法力的能力。此时，不要替族群中的每个巨人都赋相同值，最好是多样化。例如，有些可能体力好，而有些可能抵抗法力的能力强一点。让点数的分布不同，然后再对族群的适合度做分等，这可以协助找出点数分布的最佳平衡点。于是，传了几代巨人之后，就会演化成为对玩家而言，比较有挑战性的对手了。

其他信息

如前所述，实现交叉和突变有很多策略可用。此外，除了这里讨论的问题以外，还有很多其他游戏问题，可以有效地使用遗传算法。如果你有兴趣进一步研究遗传算法，也想看其他策略和其他实例，我们建议你参考下列书籍：

- 《AI Application Programming》（Charles River Media出版）。
- 《AI Techniques for Game Programming》（Premier Press出版）。
- 《AI Game Programming Wisdom》（Charles River Media出版）。

Mat Buckland的《AI Techniques for Game Programming》一书提到了遗传算法和神经网络，甚至教你如何用遗传算法演化或训练神经网络。这是我们在第十四章谈的“倒传递训练法”有趣的替代方法。

附录

向量的运算

本附录将介绍Vector类的操作方式，此类封装了撰写2D或3D刚体仿真程序时，所有需要用到的向量运算。虽然Vector类使用了3D的向量，但是你也可以把它当作2D向量来使用；只要在操作时忽略所有的z值或将z值都赋为0即可。

Vector类

Vector类定义了三个变量x、y、z，以及几个操作基本向量运算的函数和向量运算符。此类有两个构造式，其中一个将所有的分量变量初始化为0，另外一个则将分量变量指定为传入的参数。

```
//-----------------------------------  ----------------------------------
// Vector类和其中封装的函数
//-----------------------------------  ----------------------------------
class Vector {
public:
     float x;
     float y;
 float z;

     Vector(void);
 Vector(float xi, float yi, float zi);

     float Magnitude(void);
     void Normalize(void);
 void Reverse(void);

     Vector& operator+=(Vector u);
     Vector& operator-=(Vector u);
     Vector& operator*=(float s);
 Vector& operator/=(float s);
```

```
 Vector operator-(void);

}

// 构造式
inline Vector::Vector(void)
{
     x = 0;
     y = 0;
     z = 0;
}

// 构造式
inline Vector::Vector(float xi, float yi, float zi)
{
     x = xi;
     y = yi;
     z = zi;
}
```

长度

Magnitude函数仅根据以下公式计算向量的模：

$$|v| = \sqrt{x^2 + y^2 + z^2}$$

这是一个以零为基准的向量，也就是说其分量都是相对于原点的。向量的模也就是向量的长度，如图 A-1 所示。

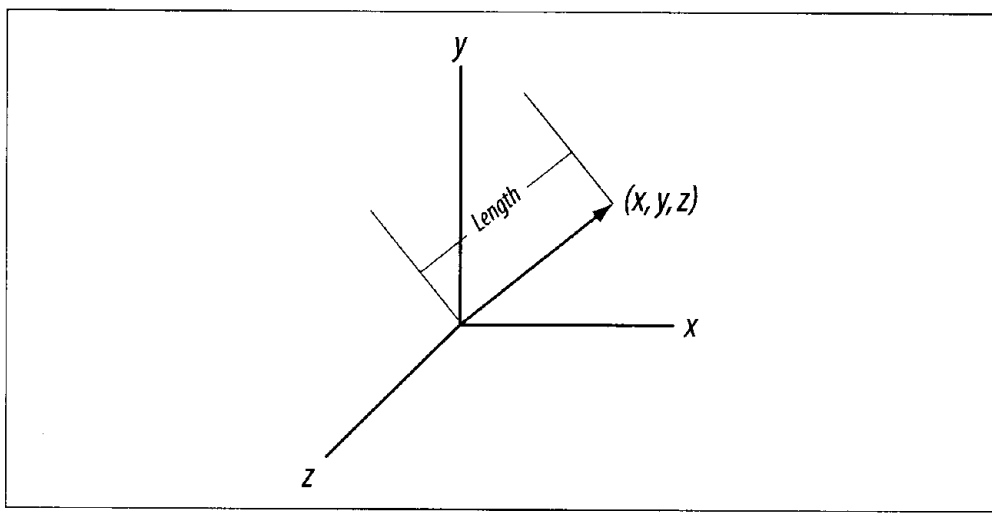


图 A-1：向量的长度（大小）

以下是 Vector 类中计算向量长度的程序代码：

```
inline float Vector::Magnitude(void)
{
    return (float) sqrt(x*x + y*y + z*z);
}
```

如果知道一个向量的长度和所有的方向角（direction angle）就可以求所有的分量。方向角的定义是向量与坐标轴的夹角，如图 A-2 所示。

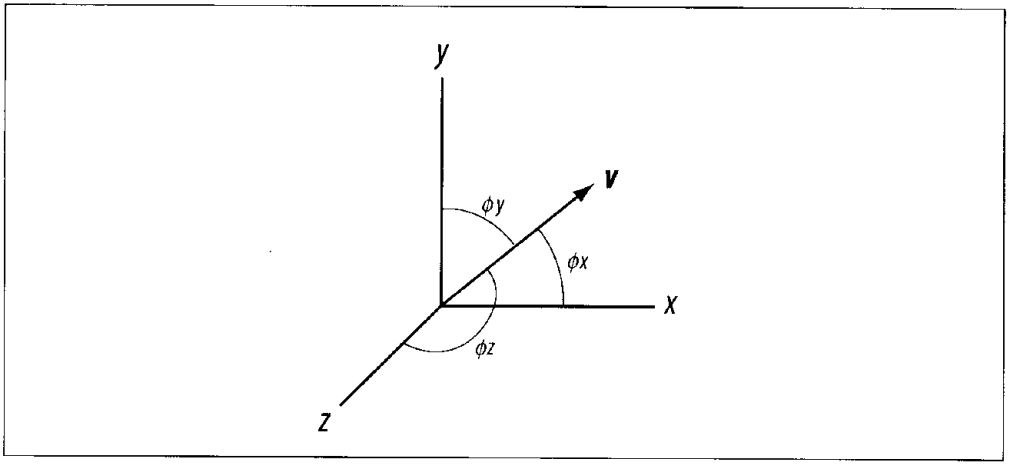


图 A-2：方向角

图中向量的分量可以这样表示：

$$v_x = |v|\cos\varphi_x$$
$$v_y = |v|\cos\varphi_y$$
$$v_z = |v|\cos\varphi_z$$

公式中方向角的余弦称为方向余弦（direction cosine）。这些方向余弦的平方和一定等于 1：

$$\cos^2\varphi_x + \cos^2\varphi_y + \cos^2\varphi_z = 1$$

标准化

Normalize 函数会将向量标准化，或者说将向量转换成单位向量，以满足下列公式：

$$|v| = \sqrt{x^2 + y^2 + z^2} = 1$$

换句话说，被标准化后向量的长度为1。假设 v 是一个非单位向量的向量，并有三个分量 x、y、z，其单位向量 u 可用以下公式从 v 中求得：

$$u = v/|v| = (x/|v|)\mathbf{i} + (y/|v|)\mathbf{j} + (z/|v|)\mathbf{k}$$

这里的 $|v|$ 是前面所述向量 v 的大小，或称为长度。

以下程序代码可将Vector类的向量转换成单位向量：

```
inline void Vector::Normalize(void)
{
     float m = (float) sqrt(x*x + y*y + z*z);
     if(m <= tol) m = 1;
     x /= m;
     y /= m;
 z /= m;

     if (fabs(x) < tol) x = 0.0f;
     if (fabs(y) < tol) y = 0.0f;
     if (fabs(z) < tol) z = 0.0f;
}
```

此函数中的tol是浮点类型的容许值，例如：

```
float       const tol = 0.0001f;
```

反向量

Reverse函数反向量的方向，做法是取所有分量的负数。调用Reverse后，向量的方向会指向在调用Reverse之前的反方向。

```
inline    void Vector::Reverse(void)
{
     x = -x;
     y = -y;
     z = -z;
}
```

图A-3说明此函数的运算结果。

向量的加法：+= 运算符

这个加法运算符专用于向量的加法，将传入向量的每个分量分别加到目前向量的每个分量中。在图A-4中可以看到，向量是以“头连尾”的方式相加。

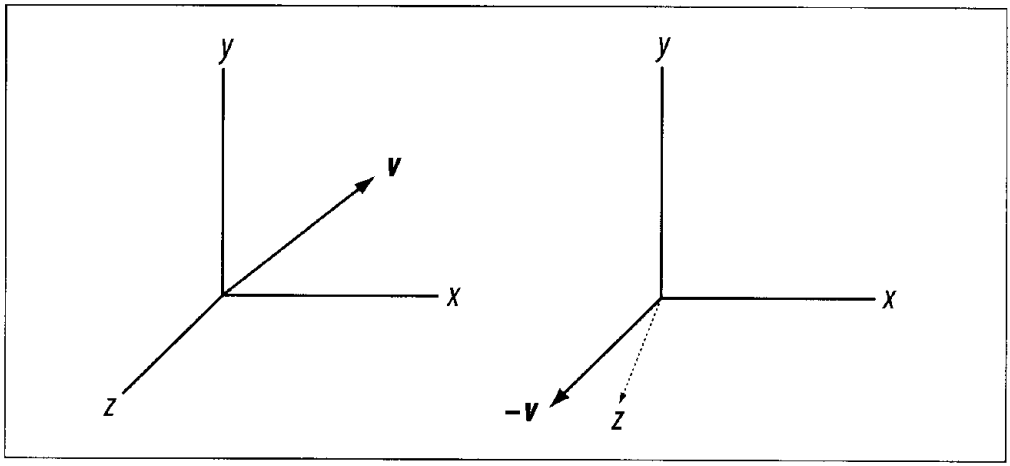


图 A-3：向量的反转

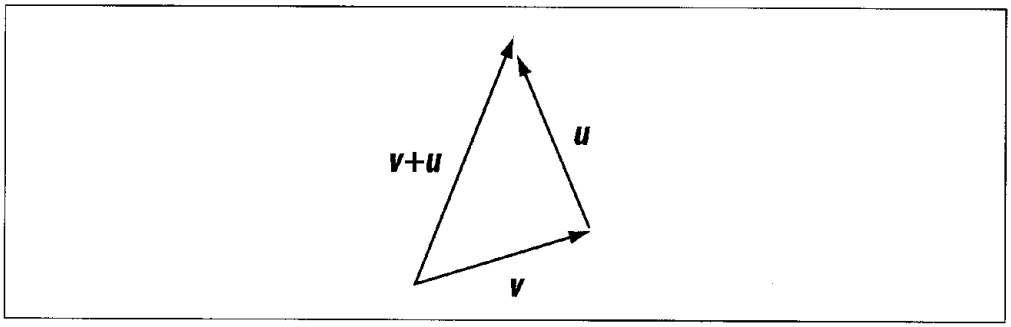


图 A-4：向量的加法

以下是将向量 *u* 加到 Vector 向量的程序代码：

```
inline   Vector& Vector::operator+=(Vector u)
{
     x += u.x;
     y += u.y;
     z += u.z;
     return *this;
}
```

向量的减法：-= 运算符

这个减法运算符用来将目前的向量减去传入的向量，也是使用分量对分量的方式来计算的。向量的减法与向量的加法很类似，但向量减法会先取得第二个向量的反向量再加到第一个向量中。图 A-5 表示向量的减法。

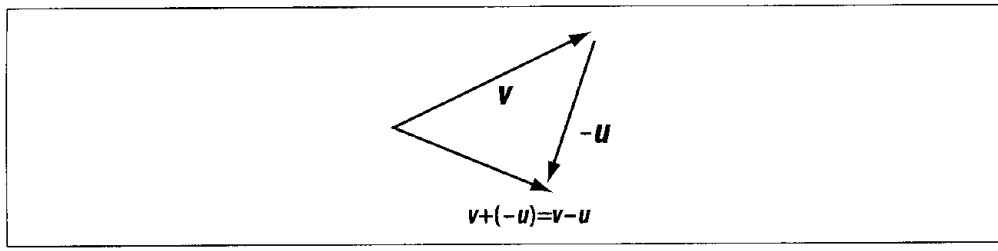


图 A-5：向量的减法

以下是从 Vector 类的向量减去向量 *u* 的程序代码：

```
inline    Vector& Vector::operator-=(Vector u)
{
     x -= u.x;
     y -= u.y;
     z -= u.z;
     return *this;
}
```

向量与数量的乘法：*= 运算符

向量与数量的乘法运算符将向量乘以传入的数量，会使向量的长度缩放。将向量乘以数量时，事实上是将每个分量分别乘以这个数量。所得到的新向量的方向与旧向量的方向相同，但是长度就不同了（除非相乘的数量是 1）。图 A-6 说明向量的变化。

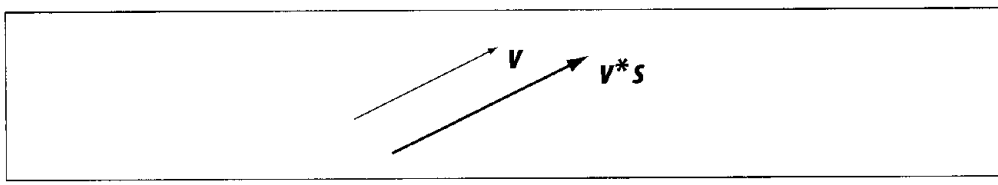


图 A-6：向量与数量的乘法

以下是 Vector 类的向量乘以数量的程序代码：

```
inline    Vector& Vector::operator*=(float s)
{
     x *=s;
     y *=s;
     z *=s;
     return *this;
}
```

向量与数量的除法：/= 运算符

向量与数量的除法与向量与数量的乘法很类似，只是数量的除法是将每个分量除以传入的数量。

```
inline   Vector& Vector::operator/=(float s)
{
     x /=s;
     y /=s;
     z /=s;
     return *this;
}
```

向量的共轭：－运算符

共轭运算符仅是取每个分量的负数，可用于向量的相减或求向量的反向量。共轭运算符和之前所说的反向量运算符的作用是相同的。

```
inline Vector Vector::operator-(void)
{
     return Vector(-x, -y, -z);
}
```

向量函数与运算符

以下函数和多载运算符可以用在两个向量间，或是向量与数量之间的运算，而其中的向量是 Vector 类实体。

向量的加法：+ 运算符

此加法运算符用以下的公式将向量 $\boldsymbol{v}$ 加到向量 $\boldsymbol{u}$ 中：

$$\boldsymbol{u}+\boldsymbol{v}=(u_x+v_x)\mathbf{i}+(u_y+v_y)\mathbf{j}+(u_z+v_z)\mathbf{k}$$

以下是程序代码：

```
inline   Vector operator+(Vector u, Vector v)
{
     return Vector(u.x + v.x, u.y + v.y, u.z + v.z);
}
```

向量的减法：－运算符

这个减法运算符用以下的公式将向量 u 减去向量 v：

$$u - v = (u_x - v_x)\mathbf{i} + (u_y - v_y)\mathbf{j} + (u_z - v_z)\mathbf{k}$$

以下是程序代码：

```
inline   Vector operator-(Vector u, Vector v)
{
     return Vector(u.x - v.x, u.y - v.y, u.z - v.z);
}
```

向量的外积：^运算符

此外积运算符计算向量 u 和向量 v 的外积：$u \times v$，并返回一个与两向量 u 和 v 垂直的向量。公式如下：

$$u \times v = (u_y * v_z - u_z * v_y)\mathbf{i} + (-u_x * v_z + u_z * v_x)\mathbf{j} + (u_x * v_y - u_y * v_x)\mathbf{k}$$

所得到的向量会垂直于向量 u 和向量 v 所构成的平面。这个向量的方向可以由右手定则来决定。也就是你先将向量 u 和向量 v 如同图 A-7 一样尾端对尾端放好，再将右手的手指（除大拇指以外）从向量 u 的方向弯曲到向量 v 的方向，此时大拇指的方向就是所求外积向量的方向。

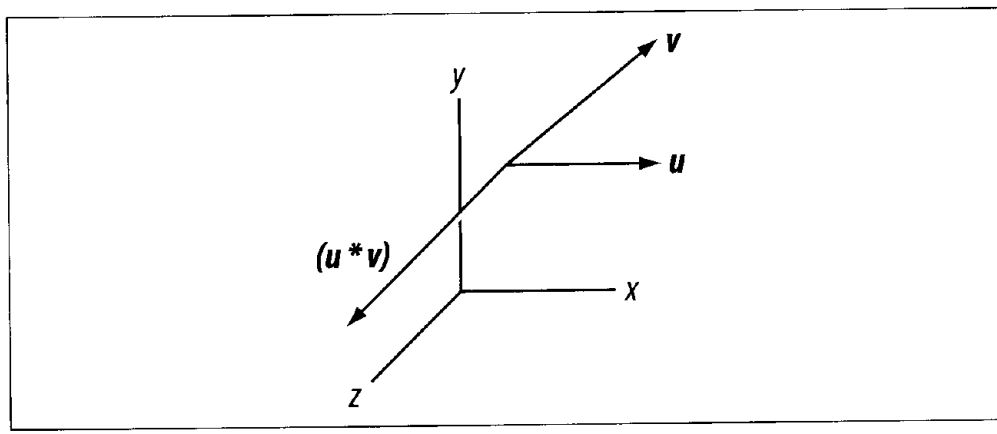


图 A-7：向量的外积

图 A-7 中所得到的向量指向 z 轴的方向，因为向量 u 和向量 v 位于 x 轴与 y 轴所构成的平面上。

如果两向量平行，其外积向量是0。这在判断两个向量是否平行时是很有用的。外积的运算符合分配律，然而却不符合交换律：

$$u \times v \neq v \times u$$
$$u \times v = -(v \times u)$$
$$s(u \times v) = (s)(u) \times v = u \times (s)(v)$$
$$u \times (v + \mathbf{p}) = (u \times v) + (u \times \mathbf{p})$$

以下是求向量 u 和向量 v 的外积的程序代码：

```
inline Vector operator^(Vector u, Vector v)
{
     return Vector(   u.y*v.z - u.z*v.y,
          -u.x*v.z + u.z*v.x,
                     u.x*v.y - u.y*v.x );
}
```

使用向量的外积来求法向量（垂直向量）是很方便的。例如进行碰撞侦测时，需求出多面体平面上的法向量。可利用多面体的顶点来得到构成平面的向量，再求这些向量的外积便得到平面上的法向量。

向量的内积：*运算符

此运算符根据以下的公式求得向量 u 和向量 v 之间的内积：

$$u \cdot v = (u_x * v_x) + (u_y * v_y) + (u_z * v_z)$$

向量 u、v 的内积表示向量 u 在向量 v 上的投影乘以 v 的长度，如图 A-8 所示。

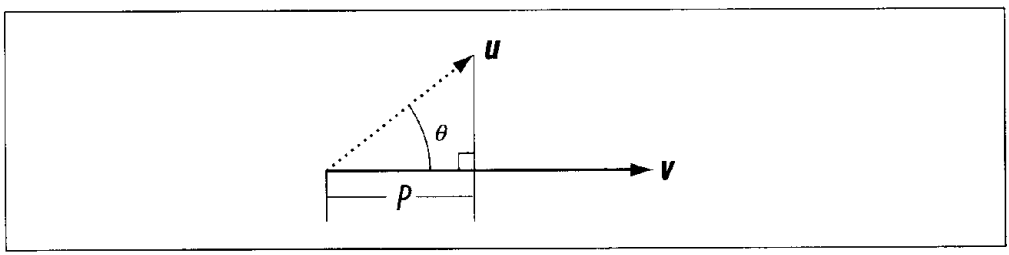


图 A-8：向量的内积

在图中，P 是所求的内积（若 v 为单位向量，则 P 即为 u 在 v 方向上的投影），且是一个数量。你也可以由夹角求得两向量的内积：

$$P = \boldsymbol{u} \cdot \boldsymbol{v} = |\boldsymbol{u}||\boldsymbol{v}|\cos\theta$$

以下是求向量 $\boldsymbol{u}$ 和向量 $\boldsymbol{v}$ 之间内积的程序代码：

```
// 向量的内积
inline float operator*(Vector u, Vector v)
{
    return (u.x*v.x + u.y*v.y + u.z*v.z);
}
```

利用向量的内积可求得一向量在另一个向量上的投影向量。回到碰撞侦测的范例，常常需要求多面体（物体1）上的顶点A到另一个多面体（物体2）上某一个面的最短距离。你可以这样做：先找出一个向量（从物体2某面的任一顶点指到A点），此向量与物体2平面法向量的内积就是A点到物体2上某个平面的距离（如果平面法向量不是单位向量，你必须要将结果除以这个法向量的长度）。

向量与数量的乘法：*运算符

这个运算符计算向量 $\boldsymbol{u}$ 乘以数量 s 的结果 —— 每个分量分别乘以数量 s。以下是依据数量与向量的顺序而衍生的两个不同版本，当然结果是相同的：

```
inline Vector operator*(float s, Vector u)
{
    return Vector(u.x*s, u.y*s, u.z*s);
}

inline Vector operator*(Vector u, float s)
{
    return Vector(u.x*s, u.y*s, u.z*s);
}
```

向量与数量的除法：/运算符

这个运算符会求出向量 $\boldsymbol{u}$ 除以数量 s 的结果 —— 将每个分量分别除以数量 s：

```
inline Vector operator/(Vector u, float s)
{
    return Vector(u.x/s, u.y/s, u.z/s);
}
```

向量的混合积

此函数使用以下的公式来求向量 $\boldsymbol{u}$、$\boldsymbol{v}$、$\boldsymbol{w}$ 之间的混合积：

$$s = \boldsymbol{u} \cdot (\boldsymbol{v} \times \boldsymbol{w})$$

这里求出来的 s 值是一个数量。程序代码如下：

```
inline float TripleScalarProduct(Vector u, Vector v, Vector w)
{
     return float(  (u.x * (v.y*w.z - v.z*w.y)) +
                    (u.y * (-v.x*w.z + v.z*w.x)) +
                    (u.z * (v.x*w.y - v.y*w.x)) );
}
```

索引

本篇索引可以帮助读者迅速找出特定信息在本书中的页码。所有项目都是依照字母顺序排列，所以读者可以像使用英文字典那样使用本索引。符号“……”代表主关键词，例如：

access map（访问表）
　以……检查客户端，151，151-155
　检查……之后的动作，152
　配置范例，153
　正则表达式，153

在此例中，“以……检查客户端，151-155”表示读者可在 151 到 155 页找到关于“以访问表检查客户端”的信息，其中 152 页描述“检查访问表之后的动作”，而 138 页提供了范例。在 158 页可找到如何在访问表里使用正则表达式。

为了方便检索，也为了编排上的考虑，原则上，顶层关键词不翻译，只加注中文解释（如果有的话），对于比较简单的字词，甚至不加注中文解释。第二层与第三层的关键词则不保留原文或不翻译，视情况而定。

希望本节的说明有助于读者使用本篇索引。如果读者对本公司书籍的索引编排方式有任何意见，请用 E-mail 告诉我们，好让我们知道如何改进。

欢迎提出改进索引方面的建议。请发邮件至 *info@mail.oreilly.com.cn*。

F

G

H

I

K

L

T

U

V

W

Z

作者简介

David M. Bourg擅长于计算机仿真领域，并开发分析工具评估，比如，气垫船效能以及海浪对船只运动的影响。目前在新奥尔良大学造船及海洋工程学院任教。David在游戏开发和顾问咨询上有其专业，同时，也是《游戏开发物理学》(O'Reilly) 一书的作者。David也在游戏学会 (Game Institute) 开网络课程，名为“Physics for Game Developers”。David就要拿到工程及应用科学的博士学位了。他目前的研究牵涉到开发计算机程序来计算仿真液体动力行为，同时把AI技术用来解决商业和工程问题。

Glenn Seemann是游戏程序员，有十几个Mac和Windows系统的游戏都是他的杰作。他是Crescent Vision Interactive这家游戏开发公司的创办人之一，专门开发跨平台游戏。

封面介绍

本书封面上是一只环尾狐猴（学名是Lemur catta）。环尾狐猴只有在非洲东南方的马达加斯加岛上才找得到。

环尾狐猴的尾巴毛浓而独特，黑白环相间，尾巴伸长最多有25英寸。环尾狐猴的鼻口又黑又尖，在不同种类的狐猴之间是很常见的特征。

环尾狐猴喜爱空旷地区，比如、岩石密布的平原以及沙漠地区。环尾狐猴通常在地上行走，不过偶尔会在树上的大树枝上走动。环尾狐猴和其他种类的狐猴不同，其他种类的狐猴喜爱丛林茂密之地，并且几乎只在树上行动。

环尾狐猴像猫那样，眼球后面有一层反射层，让它们在夜间有很好的视力。它们的尾巴很臭，可用来警告同伴有危险正在逼近。尾巴也是交配不可或缺的一部分。公狐猴会让尾巴发出臭味以吸引母狐猴，而狐猴群中时常突然爆发激烈的“臭味大战”。

环尾狐猴以5～30只群居，阶级分明，通过成员间经常发生的挑衅意味十足的冲突而制定出来。母狐猴掌管猴群，一生都待在族群里。公狐猴会时常更换族群，在其生涯中至少有一次。

生活在干燥地区，环尾狐猴只能靠水果的水分来解渴，此外，也吃树叶、花朵、昆虫和树脂。

就像多数狐猴那样，环尾狐猴只会养一个狐猴宝宝，不过，食物充足时也常见两只或三只。新生宝宝相当脆弱，能够自行拉住母猴的毛之前只能由母猴咬在嘴里四处走动；平常都是躲在母猴背上。约三周后，小狐猴会自行攀爬，通常六个月之后就会站立。在野外，环尾狐猴最久可活到27岁。