游戏开发经典丛书

随书附赠 1CD

没有抽象的数学公式，完全是平常的语言，一次教会你一小点办法，把复杂的概念和实际的应用方法解释的一清二楚，使你阅读、编程感到轻松。

AI Techniques for Game Programming

游戏编程中的人工智能技术

[美] Mat Buckland 著

吴祖增 沙 鹰 翻译

Premier Press

清华大学出版社

游戏开发经典丛书

- C++游戏编程
- 游戏音频编程
- JAVA2游戏编程
- J2ME游戏编程
- 在线互动游戏开发
- 人工智能游戏编程真言
- 游戏编程中的人工智能技术
- 3D游戏与计算机图形学中的数学方法
- 游戏美术：2D艺术与3D建模
- 游戏高级脚本编程
- MUD游戏编程
- Flash好莱坞2D动画革命
- 剑与电——角色扮演游戏设计艺术

ISBN 7-302-12599-6

9 787302 125990 >

定价:39.00 元(附光盘1张)

游戏开发经典丛书

游戏编程中的人工智能技术

（AI Techniques for Game Programming）

（美）Mat Buckland 著

吴祖增 沙 鹰 翻译

清华大学出版社

北 京

内 容 简 介

本书是人工智能游戏编程的一本指南性读物，介绍在游戏开发中怎样应用遗传算法和人工神经网络来创建电脑游戏中所需要的人工智能。书中包含了许多实用例子，所有例子的完整源码和可执行程序都能在随书附带的光盘上找到。光盘中还有不少其他方面的游戏开发资料和一个赛车游戏演示软件。

本书讲解的原理通俗易懂，介绍程序详细周到，很适用于游戏编程者自学之用，也可以作为游戏培训教材使用（本书实际已被国内外许多游戏培训单位用作指定教材）。 对于任何希望了解遗传算法和人工神经网络等人工智能技术的各行业人员，特别是要实际动手做应用开发的人员，都是一本值得一读的好书。

Mat Buckland

AI Techniques for Game Programming

EISBN：1-931841-08-X

北京市版权局著作权合同登记号 图字：01-2003-8205

981-265-948-X

图书在版编目（CIP）数据

游戏编程中的人工智能技术/（美）布克兰德（BuckLand，M.）著；吴祖增，沙鹰翻译. —北京：清华大学出版社，2006.5

（游戏开发经典丛书）

书名原文：AI Techniques for Game Programming

ISBN 7-302-12599-6

I. 游… II. ① 布… ② 吴… ③ 沙… III. 人工智能-应用-游戏-软件设计 IV. TP311.5

中国版本图书馆 CIP 数据核字（2006）第 014879 号

出 版 者：清华大学出版社 **地 址**：北京清华大学学研大厦

http://www.tup.com.cn **邮 编**：100084

社 总 机：010-62770175 **客户服务**：010-62776969

组稿编辑：许存权 **文稿编辑**：刘欢欢

封面设计：范华明 **版式设计**：王慧娟

印 刷 者：清华大学印刷厂

装 订 者：三河市新茂装订有限公司

发 行 者：新华书店总店北京发行所

开 本：185×260 **印张**：20.5 **字数**：447 千字

版 次：2006 年 5 月第 1 版 2006 年 5 月第1 次印刷

书 号：ISBN 7-302-12599-6/TP·8058

印 数：1～4000

定 价：39.00 元（附光盘 1 张）

关于作者

Mat Buckland 在伦敦大学学完计算机科学后，做过多年的风险管理咨询顾问。终于他开始厌倦了所有的金钱游戏和公司制度，就一把火烧掉了他原有的公司礼服，跑到一家为 Gremlin Software 开发游戏的公司工作。虽然薪水少得多，但是却很有趣，而且他可以每天穿牛仔裤去上班了！现在 Mat 同时是一个自由程序员和人工智能咨询顾问。自从 20 世纪 80 年代初第一次接触到这些技术，他就对进化计算和 AI 尤感兴趣。他是 ai-junkie.com 网站（www.ai-junkie.com）的作者，该网站提供一些有关进化算法的教程和建议。

来自系列丛书编辑的信

作为 Premier 游戏开发系列丛书的编辑，我很少有时间来自己编写技术书籍。所以，我必须找到那些真正热爱这个行业并有真才实学可以传授给别人的人来写这些书。如果你曾经阅读过我写的游戏编程书籍，你会发现，我总是用大量篇幅来阐述 AI 部分——从状态机到模糊逻辑——但我从来也没有足够的时间来写一部关于 AI 方面的专著。因此，我们开始为写出世界上最好的 AI 专著寻找合适的作者。现在，这本书已经完成了，我简直不能相信，我们真的把它完成了！Mat 不仅写出了我所希望有的这个专题著作，而且，远远超出了我的期望。他写的这本书有着超越时代的价值，并且会在游戏行业产生深远的影响，不仅如此，这本书对工程、生物计算、机器人制造、最优化理论及其他科学领域都将具有指导作用。

我从来没有看到过有哪本书能够把神经网络和遗传算法结合在一起，并用这些技术来构作真实的演示程序。在过去的 20 年里，我一直在使用 AI 技术，但我所惊讶的是，没人意识到这一切其实是那么简单——这并不是一门火箭科学（rocket science），而只是一种做事情的新方法。如果你曾经看过有关 AI 的那许多理论书籍，你会发现这些书非常难以理解——大量的数学、理论，除了让你输入一些系数，然后观察神经网络或遗传算法工作的迭代过程外，完全没有一个实际的真正可操作的程序——全都是些无用的信息。

当我开始计划该书时，我希望能找到这样一个作者：他不仅要完全了解自己的技术，并且还应是一个优秀的程序员，一个艺术家，还有最关键的是，他必须是一个完美主义者。Mat 和我在该书的目录单上花了不少时间，以决定本书应该涵盖的内容。同时，我们两个都绝对认为本书应该在每个章节都有实例并图文并茂，我们认为必须利用大量的插图（figure）、图例（illustrations）和直观化的东西（visuals）来帮助读者把概念落实。最后，我可以毫不犹豫地说“这是世上有关 AI 应用技术的最棒的书！”

我敢说任何人都无法拿出比 Mat 这本书更好地教授理论知识，并把理论知识结合实际让读者能了解并运用于现实的书。我保证当你读完这本书，无论你是一个程序员、工程师、生物学家、机器人科学家或其他专业的人，你会立即迫不及待地把这些技术运用到你的工作中去——本书就是有那样的神奇。

同时，本书也将会让你获得把 AI 技术应用到其他实际领域，诸如机器人（robotics）、工程学（engineering）、武器设计（weapon design）等的工具。我敢打赌，本书发行 6 个月左右后，网上就会出现有许多非常可怕的 Quake 机器人！！！

总之，无论你对计算的哪个领域感兴趣，你不能不阅读一下本书。当你发现你已有可能创造出有思考能力的机器时，你会感到震惊和喜悦——机器人除了构作在硅材料的数字世界里，在技能上不如我们人之外，其余和我们人并没有什么更多的差别，他们也是有生命（alive）的，就看你怎么来定义生命——数字化生物（Digital Biology）时代的到来将取决于我们——对生命的崭新定义，到底何谓生命，和由此开始的各种争论——生存在物质

世界的人类和有机体不能单方面地给生命和感知能力下片面的定义。就像 Ray Kurzweil 在“灵魂机器时代”（Age of Spirtual Machines）里说的那样，“再过 20 年，一个标准的台式机的运算能力将超越人脑。”当然，这个结论只是单纯地从莫尔定律出发得出，还没有把量子计算和其他各种未来必定要发生的革新技术考虑进去。我的预测是，到 2050 年，一个价值一便士、仅有针尖大小的芯片的计算能力将超越这个星球上的人脑运算能力的总和。可能我的预测是完全错误的，说不定这一芯片的能力还有可能大 100 万倍，不过我现在仍是一个悲观者。

总而言之，我们确实处于一个新时代的起点，这里，有生命的机器就要诞生，这是不可避免的。了解本书的相关技术是到达那里的第一步。这就是简单法则的应用、演化算法和以我们的生物为模型的基本技术，就可以帮助我们创造出这些机器，或者更讽刺些的话，创造我们未来的祖先。

Andre'LaMothe
Premier 游戏开发丛书系列编辑

引　言

考虑一下有多少傻瓜能作微积分演算，要让任何其他傻瓜去掌握看来是更困难或更繁琐的微积分技巧就令人惊奇了。

有些微积分技巧是相当容易的，但有些则非常难。那些编写了高等数学教课书的傻瓜——他们都是些最聪明的傻瓜——很少会劳神向你说明简单的微积分有多么简单。相反地，他们采用了最最艰难的方法来处理相关内容，好像就是刻意要让你牢记，他们那无比的智慧。

由于我是一个出名的傻家伙，我始终不学那些困难的技巧，我现在也要求为我的傻瓜读者提供那些不难的部分。彻底地掌握这些以后，其余的也会随之而来（follow）。我相信，一个傻瓜能干的，其他傻瓜也一定能干。

Silvanus P. Thompson，《简易微积分》导言，1910 年初版

家用电脑从 Sinclair ZX80 演变到现在已经历了很长的一段路程。硬件速度越来越快，而电脑组件的价格则越来越低。在短短几年时间里，我们所看到的游戏的图形质量有了飞速的提高。不过，到目前为止，这几乎就是游戏开发所有努力要解决的重点——提供更好的视觉享受，而在我们最喜爱的电脑对手的 AI 方面，却改进甚微。

然而，时代在改变。计算机的硬件现已到了这样的转折点，它已能为游戏开发者创建 AI 提供更多的时钟周期。同时，游戏的玩家们的品位也愈来愈复杂。人们已经不再希望在游戏中看到诸如 Doom 和 Quake 这些曾经是最心爱的早期游戏中所看到的那种蠢笨怪兽了。他们也不希望再看到计算机控制的游戏角色在那里盲目地蹒跚着寻找根本不存在的路径，不时地被卡在那些拐弯处，或是在没有资源的地方开发矿藏，傻头傻脑地撞在周围的树上。游戏玩家想要从游戏中获得更多的乐趣。他们希望从电脑生成的对手（或盟友）那里看到合情合理的、有智能的行为。

基于这些原因，我坚定地认为，在未来的几年里，AI 技术将会大幅度起飞。像 Black & White 和 Halo 这样的游戏已经让我们为其 AI 技术而倾心，游戏玩家们正期待更多的此类游戏。此外，基于 AI 技术和人造生命的全新游戏流派也在最近几年里开始出现，如 Steve Grand 的 Creatures，这个游戏的销售量超过 100 万份的事实让他本人和其他所有人都吃惊。但如果你认为这是很多了，那么你再看看由 Electronic Arts 制作的 Sims 的销售，到目前为止，Sims 及其服务器软件（add-on packs）的销售已经高达 1300 万份！这是一个巨大的销售收入，而同时也是一个重要指示：玩家对这类技术有多么浓厚的兴趣。这个趋势只可能继续。

创造智能幻影（illusion）的技术有多种，本书主要探讨两个方面：遗传算法和人工神经网络。这两项技术讨论的地方很多，它们无疑正是当前的热点论题，但是被误解的地方也相当多。就拿神经系统网络来说，经常会发现游戏开发者误认为神经网络是极其复杂的

东西，因而会占用太多的处理器时间，导致游戏速度降低。或者相反，他们可能过度地追求神经网络的能力，企图去创造一个有感觉的类似 HAL 的人工生命，其结果必然是以失败而告终。我希望本书能帮助他们减少类似的曲解。

上面我引用 Silvanus Thompson 所著的受到人们喝彩的《简易微积分导言》中的那一段话可以成为本书的完美的开端（谢谢，Silvanus!），因为，神经网络和遗传算法和微积分一样，对初学者来说，可能是非常难入门的，尤其对那些没有接受过正规专业教育的人来说更是如此。而已有的所有此类著作都是由学院式专家为学院式读者所写，其中充斥着（非本专业人）看不习惯的数学公式和难于理解的术语。因此，我写了一本是我刚开始对这些课题感兴趣时所想要读到的那种书：一本由傻瓜写给傻瓜看的书。相信我，如果我刚开始涉足该领域时能够得到这样一本书，我就用不着为搞清那些学究们到底是在讲什么而花那么多时间、受那么多挫折了！

经过这些年，我已经读了有关这个课题方面的很多书和论文，它们几乎都没有能给出真实的例子，没有东西可以让你牢靠地掌握并对自己说“啊！这就是我能拿来运用的东西！”。比如，关于遗传算法的书差不多总是向你提出这样的问题：

最小化下面的函数

$$f(x_1,\cdots,x_5) = x_1\sin x_1 + 1.7x_2 \sin x_1 - 1.5x_3 - 0.1x_4 \cos(x_4 + x_5 - x_1) + 0.2x_5^2 - x_2 - 1$$

其中

$$-100 \leqslant x_1,\cdots,x_5 \leqslant 100$$

我的意思是说，这是一个完全可以用遗传算法来解决的问题，但是上面的提法却让我们这些凡人实在难于理解其中的意义。除非你有一个很好的数学基础，这样的问题很可能会显得过于抽象，会立即让你感觉到不舒服，对继续学下去也就没有趣味。

但如果给你的问题是这样：

请允许我向你介绍 Bob。今天对 Bob 来说可不是什么好日子，他已深陷到一个迷宫中，而他的妻子正期待他能早点回家一起分享她花了整个下午制作的晚餐。我来告诉你如何使用遗传算法帮助他找到迷宫出口，以挽救他的婚姻。

你的大脑就有一个可以与之联系的着落点（anchor point）。你也立即会对这样的题目感到舒适。不仅如此，这也是一道有趣的题目。你一定很想知道我们怎样来帮他呢？由此你就会打开书本，继续学下去，在学习中感受到乐趣。

这就是我在本书中用来阐明概念所使用的一种类型的题目。如果我的做法正确有效，如何把你的想法运用到你的游戏和项目里就会变得显而易见。

对于我的读者，我只作一个假设，那就是读者了解如何编程。我不了解会有些什么样的读者，但是有时，当我买了一本书而发现其中仅有一部分我不了解，以至于必须去买另一本书来解释第一本书里我所不懂的内容，就会让我感到失望。为了防止这样的事情发生在读者身上，我努力确保这本书能解释代码所说明的一切——从 Windows GDI、矩阵和矢量数学的使用，直到物理学和 2D 图像学。我知道事情通常都有其两面性，很可能一些读者已经掌握了这些图形学、物理学和 GDI 的相关知识，那么，你们可以跳过这部分而只阅读那些你们更感兴趣的部分。

注：创建演示程序

演示程序是很容易编译生成的。首先把你所需的源程序复制到你的硬盘里（译注：因编译一般需要用到其他的一些文件，你在复制源文件时，应把源程序所在文件夹中所有的文件整个地从光盘复制到硬盘）。如果你用微软的 Visual Studio 来编译，只要点击工程的 workspace 文件（译注：即以.ws 为后缀的那个文件）就可以开始进行编译并最后生成可执行程序。如果使用其他的编译器，你应该创建一个新的 Win32 工程（确保 winmm.lib 在你的工程设置里），并在按编译按钮之前，先在工程文件夹里加上相关的源程序和资源文件。所有要做的就是这些。用不到添加额外的路径、DirectX 或 OpenGL。

在所有的例子里，我尽量使代码编写得尽可能简单。使用的是 C++语言，但是我希望 C 语言程序员也能理解我的代码。所以，出于这方面的考虑，我没有使用 C++所固有的诸如继承、多态一类的东西（stuff）。我使用了很简单的标准模板库（standard Template Library，sTL）功能，但是在我引用 sTL 功能的地方，会在边框里进行解释。使用简单代码的一个出发点是不使我所要阐述的原理显得晦涩。相信我，本书所涵盖的有些内容一开始并不容易掌握，所以我不想把因为在例子中使用了高级代码而把事情弄得更复杂。我已尽我所能恪守老管理顾问的信条：K.I.S.S（Keep It Stupidly Simple），让一切尽可能地简单。

好了，不再罗嗦了，让我们来开始冒险吧……

目　录

第 1 篇　Windows 编程

第 2 篇 遗传算法

第 3 篇 神经网络

译 者 序

智能和人工智能

本书介绍游戏编程中的人工智能，简称“游戏人工智能”。所谓人工智能，就是由人工建立的硬件或软件系统的智能，是无生命系统的智能。例如，一个机器人的智能，一个能与人下棋的软件的智能。智能是人类（大脑）智力活动的能力。它的表现形式非常广泛，例如，能利用各种知觉系统接受来自环境的各种信息，能形象思维，能逻辑推理，能理解事物之间的内在联系，能作归纳和推广来发现规律；能承认真理，能使自己适应环境，能主动学习，能根据以往经验改进提高自己，不至于犯同样的错误；能进行创新思维来从事各种创新等。智能是一个抽象的概念，一个软件或硬件系统是否有智能，只能根据它所表现出来的行为是否和人类上述的某些行为相类似来作判断。

人工智能也是一门学科名称。这门学科在电子计算机发明后不久即创立，其目的就是要模拟人类的智力活动机制来改进计算机的软件硬件结构，使它们掌握一种或多种人的智能，以便在各种领域内有效替代人的脑力劳动，特别是解决用传统软硬件方法难以解决的问题，如自然语言的翻译、定理证明、模式识别、复杂机器人的控制等。

游戏编程中的人工智能

现代游戏的制作都是十分庞大的工程，目前还无法期望利用人工智能来创作整个的游戏。这里所说游戏的人工智能，是指用来控制游戏中各种活动对象行为的逻辑，使它们表现得合情合理，如同人的行为一样。游戏中活动对象分两类，一类是在背景中，如天上飘着的云，飞过的鸟。这类对象的行为连同它们的造型要显得逼真也不容易，需要掌握 2D 或 3D 的图形和动画技术，还需要有艺术修养。但它们在游戏中无须人工干预，变化不很多，控制的逻辑并不复杂。另一类活动对象是游戏中的各种角色，或称游戏代理，如虚拟的人、兽、怪物、机器人等。这些对象的活动方式必须变化多端才行，否则游戏就不好玩，所以控制逻辑就比较复杂。尤其是玩家对手的代理最困难。比如，要开发一个猫捉老鼠的游戏，假定玩家（不妨就是你）的代理是猫，则猫的行为由你操纵，而老鼠的行为则完全需要由程序来控制。当猫不出现时，老鼠必须到处觅食或打洞，以解决生存必需的食住问题，而一旦发觉有猫出现，则必须立即躲进洞里。如果附近没有洞，则要立刻逃窜，而逃窜的方向取决于它和猫的相对位置。如果猫在老鼠的西边，老鼠应向东逃跑；如果猫从老鼠东边追来，则老鼠应向西跑；如果途中遇到障碍物挡住了去路，则应改变方向，如向南或向北跑，至少不应该回头跑，除非前面是死胡同。老鼠能遵循这样的逻辑来行动，就是游戏编程中为老鼠设计的智能，是游戏人工智能。同样，猫也需要人工智能，但很简单，那就是听话，能听从你用按键或鼠标进行的指挥。

所有角色扮演类游戏都需要有类似的智能。愈是好玩的游戏需要的智能愈复杂。但并不是所有游戏都要有人工智能，例如 Windows 提供的接龙和挖地雷游戏就没有人工智能问

题；网上供两人对弈的象棋、围棋、军棋类游戏也不需要人工智能。如果要由机器当公证人，那也只要很低级的智能。但一旦要求机器能与人对弈，那就需要很高的智能了。

人工智能的两种实现方法

人工智能在计算机上实现有两种不同的方式。一种是采用传统的编程技术，使系统呈现智能的效果，而不考虑所用方法是否与人或动物机体所用的方法相同。这种方法称为工程学方法（engineering approach），它已在一些领域内作出了成果，如文字识别、电脑下棋等。另一种是模拟法（modeling approach），它不仅要看效果，还要求实现方法也和人类或生物机体所用的方法相同或相类似。本书介绍的遗传算法（Generic Algorithm,GA）和人工神经网络（Artificial Neural Network,ANN）均属后一类型。遗传算法模拟人类或生物的遗传—进化机制，人工神经网络则是模拟人类或动物大脑中神经细胞的活动方式。为了得到相同智能效果，两种方式通常都可使用。采用前一种方法，需要人工详细规定程序逻辑，如果游戏简单，还是方便的。如果游戏复杂，角色数量和活动空间增加，相应的逻辑就会很复杂（按指数式增长），人工编程就非常繁琐，容易出错。而一旦出错，就必须修改源程序，重新编译、调试，最后为用户提供一个新的版本或提供一个新补丁，非常麻烦。采用后一种方法时，编程者要为每一角色设置一个智能系统（一个模块）来进行控制，这个智能系统（模块）开始什么也不懂，就像初生婴儿那样，但它能够学习，能渐渐地适应环境，应付各种复杂情况。这种系统开始也常犯错误，但它能吸取教训，下一次运行时就可能改正，至少不会永远错下去，不必发布新版本或打补钉。利用这种方法来实现人工智能，要求编程者具有生物学的思考方法，入门难度大一点。但一旦入了门，就可得到广泛应用。由于模拟法编程时无须对角色的活动规律作详细规定，应用于复杂问题，通常会比前一种方法更省力。

有关人工智能的书籍

目前市面上游戏编程的书已出版了不少，每一本书都会介绍一些人工智能，但采用的都是传统的方法，没有一本书像本书那样采用 GA 和 ANN 技术来实现。市面上有关 GA 和 ANN 的专著也能找到，但均因偏重理论，大量采用数学公式，缺乏实际应用例子，使缺少相关数学基础的游戏开发人员或其他应用开发人员难于入门，无法使用。本书的特色就是抛弃了通常书中所有的抽象数学公式，采用平实的语言，以一次教一小点的办法，把复杂的概念和实际的应用方法解释得一清二楚，使读者阅读起来感到轻松。译者在翻译此书的过程中就深深体会到了这一点。

本书各章内容的概述

本书正文由 3 篇组成。第一篇（第 1，2 两章）介绍 Windows 编程。这一篇的写作风格也很有特色，用不多的篇幅就把 Win32 编程的繁琐入门知识以及利用 GDI 制作图形和动画的原理交待清楚，使读者从此见到 Windows 代码不会生畏，为后面各章的阅读打下基础。同时，只要参照书中的大量例子，也不难自己动手编写出许多 Windows 程序。

第 2 篇（第 3 章～第 6 章）介绍遗传算法。这一部分需要借用生物遗传进化中的选择、

杂交、变异等概念，在算法实现时则又需要用到数学、物理和计算机软件方面的许多知识。本书首先以讲故事的形式使读者了解生物的遗传进化过程，然后介绍模拟这一过程的遗传算法的进化过程（一种特殊的算法迭代过程），再通过寻找迷宫路径、寻找最短巡回路线和月球登陆飞船的模拟等 3 个有趣例子的多种求解过程，把算法实现所需的详细知识全部串接了起来，所有的内容都很有用也极为有趣。

第 3 篇（第 7 章～第 11 章）介绍 ANN。前面已提到，ANN 实际是人类或动物大脑工作机制的一种模拟。在游戏中主要用来控制对手的游戏代理的行为，使它们像真人那样，能通过学习来认识环境、适应环境，或学会与他的对手（玩家）进行周旋和较量的本领。其中第 7 章介绍了 ANN 的基本知识和最常用的 ANN，并介绍怎样用这种 ANN 来表示一个扫雷机，即一个用于自动扫雷的虚拟机器人，再结合遗传算法来改进 ANN 的性能，也就是改进扫雷机的本领。第 8 章的例子也是扫雷机，但在环境中设置了障碍物，因而要求机器人有知觉，能识别障碍物，避免与它们碰撞。第 9 章介绍网络的训练，就像我们在从事某些工作前需要预先接受训练一样。该章首先介绍网络训练的基本方法，并以实现异或（XOR）函数的网络为例，详细介绍了利用最通用的反向传播法的训练过程。接着介绍的几个例子是训练网络来识别玩家通过鼠标所作的手势。手势用来指挥玩家自己一方的游戏代理的行动，如指挥猫怎样来追赶老鼠。在团队式的游戏中，不同的手势可用来代表指挥诸如一群战士如何冲锋陷阵，如何包围对方，如何撤退的各种命令。本章最后介绍了改进网络训练的一些有用技巧。第 10 章介绍网络的实时演化，它允许一群由 ANN 所代表的个体经常性地有诞生和死亡发生，如同人类那样（此前介绍的演化，个体只能成批地进行更新）。采用的例子是外星人工程。一群外星人想来入侵，要遭到对手（你）的射击。为避免被你杀死，他们在入侵前先要接受训练，学会躲避被你所发的子弹打中。最后一章介绍一种非常特殊的神经网络，它的结构不是固定的，而是能根据目标需要，由小到大自动形成。只要提出不同的要求，就能形成不同的网络结构，是一种生成网络的网络，一种通用的网络。

值得注意的是，本书所介绍的所有人工神经网络程序都是结合遗传算法实现的。这样，人工神经网络不但可以模拟单个的大脑功能，同时也能模拟人类或生物群体借助于自然选择和遗传进化机制而形成的、具有个体差异的许多大脑的功能。反映到游戏中，就是一群具有不同性能和不同行为方式的个体，而不是一群行为清一色的个体。

除上述 3 个部分外，本书还有 3 个附录。其中前两个附录分别介绍和游戏开发所需的 WEB 资源和参考文献。第 3 个附录则介绍随书所带光盘的内容。但此介绍很不完全。要了解光盘上究竟有些什么，最好是用资源管理器直接打开光盘来看，或者查看根目录下由译者加入的 readme.htm 或 index.htm 文件，它们完整地列出了光盘内容，还介绍了它们的用法以及使用中可能出现的问题和解决方法。

本书的局限性

需要说明，本书不是游戏编程的万宝全书。为了构建 AI 完整的应用程序，书上确实介绍了不少必要的编程知识，但重点仍然只是教会读者在程序中应用 AI，并未详细介绍与此主题无关的各种游戏编程知识。后者应由成套的游戏开发丛书完成。同时，书上介绍的例

子虽然都是完整的程序，但并不是完整的游戏程序。在一个实际可玩的完整游戏程序中，不仅需要有人工智能来控制角色，还需要考虑角色本身的造型，考虑丰富的背景、声响、脚本等。人工智能只是整个程序中的一些组成部分、一些不可缺少的模块。

其次，本书也不是人工智能的万宝全书。游戏人工智能除了用 GA 和 ANN 之外，还可用诸如有限状态机、模糊逻辑，甚至更简单地直接利用 if-else 或 switch 语句来实现。即使是 GA、ANN 领域，本书也不可能像有关专著那样讲很多。为了提供足够的篇幅来介绍实际的程序代码，对其他的内容必须有所取舍。但总的说来，本书收集的内容是丰富的。收集的材料既有基本的，也有深入的，甚至也有其他书上还见不到的很先进的材料，如最后一章介绍的 NEAT 网就是一个例子。这种类似于胚胎发育那样，能由小到大自动形成拓扑结构的新型人工神经网络，无论在理论上或实际应用上都有很大的价值，本书花大量篇幅介绍了它，而在其他书中或者完全不谈，或者只是泛泛谈一下，没有任何程序。

和任何书一样，本书也有缺点和不足。尤其是因为第一版，更少不了差错。其中多数是印刷排版错误，也有一些其他类型的差错。但已发现的差错在中译本中均已更正（译者发现的这些差错凡有疑问的，均已向原书作者提出，并在他的确认或帮助下改正了）。为了节省篇幅，这种类型的差错不再加译注，但将会在本人的个人网站 www.ggdn.net 论坛中公布。此论坛将来也要用来公布今后可能发现的新错误，包括原书新发现的错误或中译本中发现的错误。如果读者阅读中发现了任何错误也恳请到那里去贴出，以便让译者和广大读者都知道。

其次，译者对本书也有一些不够满意的地方。例如：没有为每个程序提供框图，这不利于读者掌握算法的整体概念；执行程序的使用方法只写在书上，没有写在程序的帮助菜单中，这样不看书就不知道如何操纵程序，很不方便；所有程序的演化结果都没有保存到磁盘，这样每次启动都得从头开始演化，非常费时间，也不够实用；有些例子（如扫雷机或登月飞船）的人工智能是否足够，也值得研究等。但所有这些，当读者看完书中相应部分后都可以自己解决。建议读者把这些作为练习，完成后一定大有收获。

适用对象和需要基础

本书的对象首先是游戏编程人员和教学人员（本书已被国内外许多游戏学校或培训单位用作指定教材）。同时，凡希望了解和应用 GA 和 ANN 的人员，无论属于哪一个领域（自动控制、规划设计、组合优化、图象处理、机器人、人造生命等），也无论是学生、教师、程序员、工程师……也都值得一看。此外，作者能把十分难写的题材写得如此深入浅出、通俗易懂，也是值得我们在编写各类图书，特别是自学教材时好好学习的。

阅读本书需要 C/C++的基础知识，但对 C++的要求不是很多。另外，如果想利用本书的源码来编译，需要掌握 VC 或 BCB 之类的集成开发环境的基本用法。

本书翻译的曲折经历

本书原由《Windows 游戏编程大师技巧》译者、现旅居加拿大的沙鹰先生翻译。他早在 2004 年 7 月就已把第 6 章以前的所有内容译毕。后因眼疾无法继续，委托我接替他的工作。我从 8 月下旬由 Amazon 邮购到原版图书开始动手，到年底也把后面所有内容译完，

并用 Word 按出版社提出的要求排了版，也改制了原有的光盘文件。本以为 2005 年春就能和读者见面，但到 6 月下旬突然收到原编辑高巍先生来电，叫我补译第 3 章。我询问是否弄错了？但高巍说没错，并说已得到沙鹰确认，我就翻译了这一章。应该差不多了吧？但隔了一段时间，出版社许存权先生告诉我，高巍已调离，并寄来沙鹰的全部底稿，要我按照我的格式整理他的底稿。我看了后发现底稿采用 xml 格式书写，但缺少其中引用到的 xsl 样式文件和所有图片文件，第 6 章后半部分的译稿也缺。于是直接和沙鹰联系，希望从他那里获得这些资料（我相信他是有的），但无论用 E-mail 或电传，均无回音。不能再等了，我着手重复做了各种工作。当我把他的底稿整理完毕，再次试发 E-mail 给他，希望他能把译稿校一遍（因我对他的译文有较多更动，且他原来也曾向我表示希望能负责统校），但仍无回音。没有办法了，我只得再把全书复查一遍，制作了目录和索引，向出版社交了稿。不知道沙鹰现在情况怎样，我很为他的眼睛担心。

我在翻译、校对和为本书排版的过程中，得到出版社高巍先生和许存权先生的许多指导和帮助，在此表示十分感谢！

吴祖增

2005-9-15 于复旦十舍

留下我和沙鹰的 E-mail 地址和论坛网站，欢迎各位读者提问和批评指正。

沙　鹰：yingsha@hotmail.com 或 ysha@jamdat.ca

吴祖增：zzwu@citiz.net 或 zzwoo@126.com

论坛：www.ggdn.net/bbs

前　言

欢迎阅读《游戏编程中的人工智能技术》。我想你会发现，这正是你所读到的有关游戏编程的最有用的图书之一。

Mat 最先引起我的注意大概是在 2000 年的时候，那时，他开始在 GameDev 论坛（www.gamedev.net）发布各类关于游戏人工智能的帖子，并回答网友的各种相关问题。他很快就获得了大家的关注，在跟贴中也不乏赞扬与认同，尤其是在贴出他为公众消遣而制作的有关神经网络和遗传算法的两个教程之后。Mat 发现游戏开发者需要获得 AI 技术方面的知识以期运用在游戏制作中，而他的两个教程以及在 GameDev 论坛里对提问的耐心答复无疑是满足这一需要的一个途径。我对能为这样一个专题的书写前言而感到荣幸，希望以后能有更多的此类专著。

本书内容

本书基本上是为提高游戏制作水平而写的。主要通过使计算机对手更聪明、更有能力和更接近真人来达到这个目的。这个新的知识领域只是在过去 10 年左右的时间里才开始引起真正的关注。

本书出版的同时，游戏开发者可以发现当今的游戏产业正不断扩展，吸引着更多新的玩家，并以史无前例的速度蓬勃发展。随着新的游戏机和 PC 平台不断拥入市场，游戏开发者会发现自己拥有极为充裕的物质条件：更大的记忆空间，更快的 CPU 速度，更多的连接选项和更高的视频分辨率。这些新的功能为游戏开发者提供了无限的可能，但同时也让他们面对无数的取舍和重点选择。新的游戏应该是在视频分辨率上更高一些呢，还是应该使碰撞更具有真实感些？在游戏速度方面，我们在制作一年半以后要上市的游戏时，能够在那时的主流机上做何种程度的开发呢？如何使我们的游戏区别于市场上那些竞争对手的产品？

采用大量的游戏人工智能（AI）显然是使你的游戏鹤立鸡群的重要手段之一，有关游戏 AI 专题的书和文章的源源出版能充分说明这一点。高质量的游戏 AI 已经不再是为提高出帧率（framerate）才予以考虑的东西，它现在已是和图形（graphics）或声音（sound）一样，成为游戏设计过程的极为重要的一个部分，它是促进还是阻碍游戏产品畅销的一个决定性因素。游戏开发者正竭尽所能研究新的 AI 技术，以藉此构筑更好、更聪明的游戏 AI。他们想要探索新的理念，使 AI 技术进入下一代，到那时，AI 不仅是要创造有趣的游戏对手，而且还要使这个对手能够与玩家交谈（talk），能和众多的在线冒险家周旋（interact with），能在一个一个的游戏中不断学习，使它在下一轮的游戏中变得更加聪明机智。

自然，这些新的 AI 也得帮助游戏产品更好地销售。这永远是一根底线。如果一个游戏产品不销售出去，则无论它的 AI 有多么高超也都无关紧要了。

制作更好的游戏

本书着重探讨一个比较“新奇”的技术领域（无论如何，对于游戏行业来说如此）：遗传算法和神经网络及其在游戏制作中的运用。这个领域以前一直很难使游戏开发者感兴趣，主要是有以下几个原因。大多数游戏开发者都认为他们现有的技术已经很完善且容易调试。标准的有限状态机（finite state machine，FSM）和模糊状态机（fuzzy state machine，FuSM）已经出色地提供了稳固而易于调试的 AI 技术，一些备受欢迎的游戏，诸如帝国时代（Age of Empires）到 Quake 都是采用这种技术的。的确，该技术是切实可用，而且如果有足够的编程时间，该技术可以适用于几乎所有的场合。

但是这些技术也常常面临着过多的选择，而这就是游戏开发者开始采用减少返回法则（law of diminishing returns）的地方。通过构筑 FSM 来处理一些新游戏所固有的无数的可能性会使开发者心思混乱，AI 必须估算的选项数量多得使他们不知所措。对于一个游玩人来说，可能只有二三种“明显”更好的选择，但对一个需要在星期六晚上为即将发送到出版商那里的游戏最终版编写 AI 程序的开发者，会这么认为吗？如果玩家看到，在一个将使游戏完全死机（hang）的关键抉择面前，AI 作出了错误的选择，或者更糟的是，做了一个愚蠢的选择，且这样的情况发生了几次，那么，完了！玩家就会把你的 CD 从驱动里退出并转向去玩别的游戏了。

与此相反，我们应该让玩家面对的计算机对手不存在盲点（blind spot），在一游戏中不存在那种会被玩家发现，并一旦运用就会引起 AI 脑死（AI brain-dead）的某种特殊状态组合。并要求玩家面对的 AI 能够不断适应玩家的风格，能随着玩家对游戏学习的深入，一起变得更加精明能干。

这种适应性，或者学习本领，对游戏开发者及游戏玩家来说就犹如一个圣杯（Holy Grail），无论何时，当问及玩家未来最希望看到的游戏时，回答的都会是这个。玩家希望接受能够适应他们的游玩风格的 AI 的挑战，而 AI 也能预料到玩家最有可能做些什么，并相应地采取一些策略步骤，AI 所做的就如同另一个人所做的那样。

进入未来

现在让我们来谈谈本书所包含的一些更有兴趣的 AI 技术。这些技术给通常是枯燥的、逻辑性较强的 AI 领域带来了许多生物学的思考方法，为开发者提供用以创造能和玩家一样思维的计算机对手的工具。使用这些技术，一个开发者可以构筑一个足够聪明的 AI，它将能尝试不同的事物来考虑哪个效果更好，而不是只简单地从程序员编制的菜单里挑选出某个选项。它将会分析敌对部队的相对实力和位置，找出一个最近的突击点，并且不时地重新调配自己的部队来夺取胜利。

这些技术所带来的好处不仅仅是玩家能够玩得更畅快。如果构造合理，一个有学习功能的 AI 可以对程序员所花的游戏开发和测试时间产生确实的效果，因为程序员将不再必须构筑和测试几十甚至上百的脆弱而死板的 AI 逻辑。如果 AI 可以在只给予几个基本指导之后就通过观察职业人类玩家来学习怎样玩游戏，这个 AI 就不但会更牢靠，而且无疑能把游戏玩得更好。这就是阅读关于篮球的知识和实际亲手玩篮球的效果区别所在。

是不是这样就意味着 Mat 已经完成所有高难的工作，而你所需要做的就只是复制和粘贴他的编码到你最新的项目里去构筑一个能像真人一样的 AI 玩家呢？当然不是。这里所呈献的，是为那些完全不了解这些更边缘的 AI 技术和正在为新项目寻找灵感的人提供一个指导，一个框架和一条基线。可能，你没有时间去亲自研究这些可能性，或者你想摆脱那些课本或网站上提供的过于理论性的说明，那么这本书可以为你提供所需的知识。

接下来的章节将用一种简单易懂的方式探讨这些技术。采用 Mat 一贯使用的游戏开发者互相探讨问题的方式阐述问题。

有学习功能和高适应性的 AI 技术是一门新兴的技术，它可以使游戏制作更完善，更满足游戏玩家的要求，并且，更重要的是，能更好地促进游戏发行量。

Steven Woodcock
ferretman@gameai.com

致　谢

首先，我要感谢我的爱妻 Sharon，在我写这本书的时候给予我极大的容忍、理解和鼓励。有无数次，当我从键盘前转过头去，眼神空洞地问她：“对不起，你刚才说什么来着？”时，她竟没有一次失去耐心对我扔盘子。

感谢 Premier 出版社的 Mitzi 女士在本书写作的整个过程中给予我的帮助，回答我那些常常显得很荒唐的提问（虽然她认为约克夏男人讲话像 Jamie Oliver！）。非常感谢 Jenny，我的编辑，对负责检查我的程序的 bugs 的 Andre 如此严厉，也感谢 Heather 帮我纠正了我的所有错误，并美国化了我的文章。

非常感谢 Gary “Stayin’Alive” Simmons，他第一个建议我编写此书；还有我的网上教程的支持者们，他们的邮件每天都给予我很多鼓励；感谢 Steve “Ferretman” Woodcock 为我写了前言；还有 Ken，回答了我很多关于 NEAT 的疑问。

当然，我也不能忘了 Fish 和 Scooter 两个小宝贝，每当我坐下来写作时，它们就会跳到我的膝盖上来，向我亲热，使我感到温暖。

第 1 篇　Windows 编程

本篇包括以下内容：

第 1 章　Windows 概述

计算机用户：“刚才我装好了 Windows 3.0。”
技术支持：“是。”
计算机用户：“我的计算机现在不工作了。”
技术支持：“不，你刚才说不是这样。”

1.1　历史一瞥（A Little Bit of History）

很久以前，那是人们认为《飞狼》连续剧很刺激的年代，也是人们闲逛时手里喜欢抓个魔方的年代。那是 1983 年，有一个叫做比尔·盖茨（Bill Gates）的人宣布他的公司——Microsoft（微软）——将发布一个新的操作系统。新的操作系统将被命名为 Windows。其实 Gates 原本决定把他的产品叫做“界面管理器（The Interface Manager）”，幸运的是，他手下的行销专员要他相信 Windows 这个名字会更好。不过，尽管 Gates 在 1983 年下半年已向 IBM 展示过一个 Windows 的 beta 版，公众用户仍不得不继续等待，因为最终的产品直到两年之后才正式推出。

1.1.1　Windows 1.0

Windows 1.0（如图 1.1 所示）真可怕——笨拙、缓慢、错误多，而且最要命的是很难看。而且当时几乎没有软件支持，直到 1987 年 Aldus 发布了一个产品 PageMaker。PageMaker 是供 PC 机用的第一个 WYSIWYG（What You See Is What You Get，所见即所得）桌面出版软件。之后很快就有一些其他的软件面世，如 Word 和 Excel，但 Windows 1.0 从来没有获得过最终用户的好感。

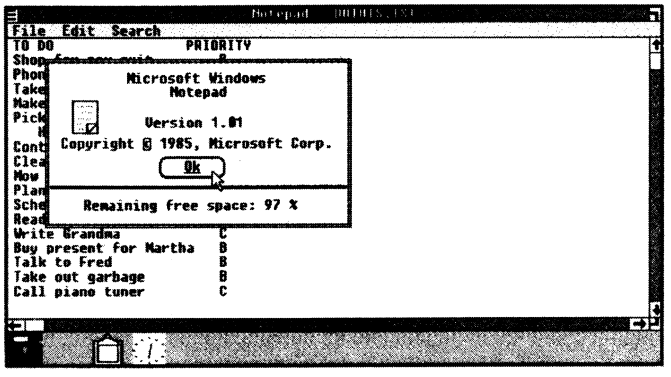


图 1.1　Windows 1.0

1.1.2　Windows 2.0

当 Windows 2.0 发布之时，用户界面与苹果公司（Apple）的 Macintosh 计算机的 GUI（图形用户界面）很相似。Apple 对这模仿者的出现显然有些气恼，并对 Microsoft 提起了诉讼，指控 Bill 剽窃了他们的创意。Microsoft 则声明他们早先曾和 Apple 有过一个协议，该协议给予了他们使用 Apple 特征的权利。4 年之后，Microsoft 赢了官司。因此，Windows 2.0（如图 1.2 所示）得以继续在商店的货架上进行销售。但由于缺乏软件开发者的支持，Windows 2.0 的销量很不理想。毕竟，一个没有兼容软件的操作系统没有什么用。

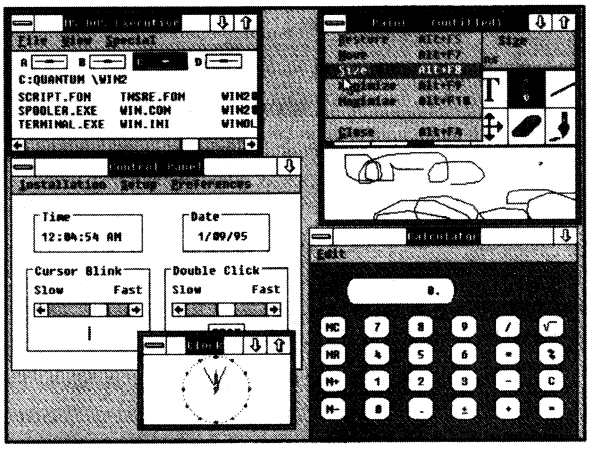


图 1.2　Windows 2.0 界面

1.1.3　Windows 3.0 和 3.1

Windows 3.0（如图 1.3 所示）是在 1990 年发布的。它有 16 种颜色和图标，并有一个改进的文件管理器和程序管理器。尽管仍存在一些问题，不知为何程序员们似乎开始喜欢上了 Windows 的这个新版本，于是开发了很多软件。Microsoft 找出了很多问题之后于 1992 年发布了 Windows 3.1。这个版本稳定多了，并开始具备了对音频和视频等内容的支持。Windows 3.1 自发布之日起两个月内，就售出了 300 万份。紧接着，Microsoft 又发布了 Windows 3.1 for Workgroups，其中增加了对网络的支持。至此，Microsoft 成功地走上了广阔的时代。

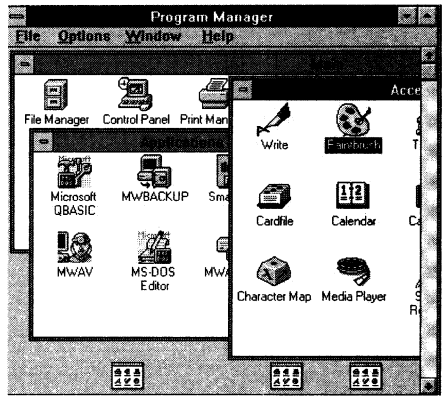


图 1.3　Windows 3.0 界面

1.1.4 Windows 95

Windows 95 是第一个无须预装 MS-DOS 就可以安装的 Windows 版本。界面很好看，而且是真正的 32 位多任务环境。笔者还记得当时在几个朋友的公司里安装它的情景。安装完毕后首先干的一件事就是在 4 个不同的窗口里同时运行同一个屏幕保护程序，接着大家的脸上都洋溢着笑容，对视片刻之后，同时发出了“Cool!”的由衷赞美。那的确是一个新纪元的诞生。游戏也开始在 Windows 下面运行得比较快了。因为在 Windows 95 面世以前，那些运行在 Windows 下的游戏都是被取笑的对象。它们又慢，又难看，而且玩起来很无聊。人人都知道好游戏都要在 DOS 下面运行，否则它就根本算不上游戏。终于，Windows 95 改变了这一切。游戏玩家再也不需要为获得可用的基本内存和扩展内存来运行某个游戏程序而整天折腾 config.sys 和 autoexec.bat 了。现在，只需安装 Windows，安装好后就可以开始游戏了。Windows 95 真是个出乎意料的好东西。

1.1.5 Windows 98 及其后续版本

此后的几代 Windows 都不同程度地基于大受欢迎的 Windows 95。Windows 不断地变得更稳定可靠、对用户变得更为友好，开发 Windows 程序也变得更容易。DOS 已经成为了遥远的过去，现在几乎所有的电脑游戏[①]都是运行在窗口环境下的。在多个界面下——Windows 98、Windows Me、Windows 2000 及 Windows XP——Windows 已是当今独占统治地位的操作系统。这就是本书的代码为什么要在 Windows 下开发的理由，也是笔者为什么一开始需要向读者介绍 Windows 编程的理由。现在就开始吧！

1.2 Hello World!

大多数编程书都是从教你如何编写一个在屏幕上显示“Hello World!”字样的简单程序开始。在 C++中，这个程序如下：

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World! \n";
  return 0;
}
```

运行时，这个简易程序会给出如图 1.4 所示的输出。

现在也保持传统，介绍一下如何在屏幕的一个窗口中显示这一句熟悉的话。

[①] 译者注：包括 Windows 和 Mac。

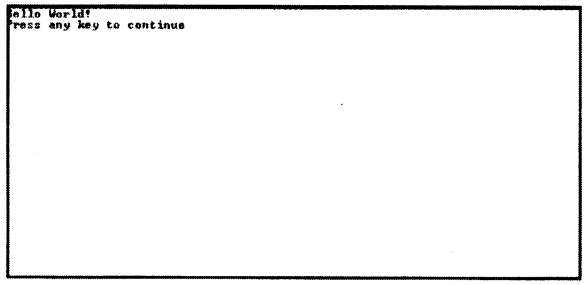


图 1.4　输出到控制台（console）中的"Hello World!"

1.3　第一个 Windows 程序

开动了！系好安全带，我们就要上路了！在开始的时候，Windows 编程可能会使你觉得头疼，但可以保证，一旦你自己动手写过几个程序之后，就不会那么头痛了。事实上，你会开始喜欢上它。好了，下面就是让屏幕的一个窗口中显示“Hello World!”字样所需的完整代码。

```
#include <windows.h>
int WINAPI WinMain (HINSTANCE   hInstance,
                    HINSTANCE   hPrevInstance,
                    LPSTR       lpCmdLine,
                    int         nCmdShow)
{
  MessageBox(NULL, "Hello World! ", "MsgBox", 0);
  return 0;
}
```

如果在集成开发环境（IDE）里输入这个程序，再编译和运行它（或者直接运行配套光盘里的可执行文件 HelloWorld1.exe），在屏幕上就会出现一个小的消息框，等待单击 OK 按钮后退出，如图 1.5 所示。如果决定自行输入源程序，在 IDE 里选择创建的 project（工程或项目）类型时，必须选择 Win32 Application，而不是 Win32 console Application。否则代码不会通过编译，你可不愿意自己的爱驹在跨第一步时就马失前蹄吧。

图 1.5　消息框

如你所见，首先不同的一点就是程序入口点不再是昔日的：

```
int main()
```

而是

```
int WINAPI WinMain (HINSTANCE   hInstance,
                    HINSTANCEhPrevInstance,
                    LPSTR     lpCmdLine,
                    int         nCmdShow)
```

下面就来解释这一步。

基本上可以忽略 WINAPI。它只是 WINDEF.h 中定义的一个宏，形式如下：

```
#define WINAPI _stdcall
```

这是使编译器知道应当以 Windows 兼容方式产生机器指令。如果忽略它，程序仍然可以编译并运行，但在编译期间会收到一条警告信息，建议一定要确保 WinMain 函数具有 WINAPI 前缀。

第一个参数 hInstance 是一个实例句柄（Instance Handle）。这基本上是 Windows 在运行时期（run time）为程序提供的一个识别标志（ID）。偶尔调用众多的 Win32 API 函数时，就需要传递实例句柄作为一个参数。Windows 用这个句柄来识别程序，使它与同时处在运行状态的其他程序区别开来。

第二个参数 hPrevInstance 也是一个实例句柄，但今日人们总把它设为 NULL。在过去，它曾被一些 16 位 Windows 应用程序使用，以达到同时打开一个程序的几个 copy 的目的。但时至今日，它已经没用了。

参数 lpCmdLine 与 DOS 程序中 main()函数的 argc 和 argv[]参数非常相似，只是用来将命令行参数传递给应用程序。LPSTR 在 WINNT.H 中被定义为字符串指针。当从命令行运行应用程序时，字符串 lpCmdLine 中就会含有所输入的、除了程序本身的名字外的所有东西。举例来说，如果执行程序的名称是 MyGame.exe，而输入的命令是 MyGame /s/d/log.txt，那么 lpCmpLine 中就会含有“/s/d/log.txt”字样。

表 1.1　nCmdShow 选项

参　数	意　义
SW_HIDE	隐藏此窗口并激活另一个窗口
SW_MINIMIZE	最小化指定窗口，并激活处在系统列表里最顶层的窗口
SW_RESTORE	激活并显示某个窗口。如果该窗口处在最小化或最大化状态，则 Windows 替它恢复原始尺寸和位置 （注：与 SW_SHOWNORMAL 意义相同）
SW_SHOW	激活窗口并以当前尺寸和位置显示它
SW_SHOWMAXIMIZED	激活窗口并以最大化方式显示它
SW_SHOWMINIMIZED	激活窗口并以图标方式显示它
SW_SHOWMINNOACTIVE	将某个窗口以图标方式显示。当前活动窗口不受影响
SW_SHOWNA	将某个窗口以当前状态显示。当前活动窗口不受影响
SW_SHOWNOACTIVATE	将某个窗口以前次尺寸和位置显示。当前活动窗口不受影响
SW_SHOWNORMAL	激活并显示某个窗口。如果该窗口处在最小化或最大化状态 （注：与 SW_RESTORE 作用相同）

最后一个参数 nCmdShow 告诉你的程序在启动后应怎样显示。有许多不同的参数可供选择，总结于表 1.1。

当用户在桌面或在开始菜单里创建应用程序的快捷方式时，他可以指定应用程序的打开方式。因此，如果用户觉得窗口运行后就应该马上最大化，那么就把 nCmdShow 设成

SW_SHOWMAXIMIZED 就好了。

好了，已经解释了 WinMain，接下来看这一行代码：

```
MessageBox(NULL, "Hello World! ", "MsgBox", 0);
```

这里简单地调用了成千个 Win32 API 函数之中的一个。它可以显示一个消息框，还可以通过一些参数来设置这个消息框的式样。这个短小的函数很方便，特别是在需要向用户显示错误信息的时候。如果写了一个函数而觉得它需要报错，你就可以这样来写：

```
if (error)
{
  MessageBox(hwnd, "Details of the error", "Error!", 0);
}
```

下面是函数的原型：

```
int MessageBox(HWND      hWnd,       // 拥有者（owner）窗体的句柄
               LPCTSTR   lpText,     // message box 中的 text 的地址
               LPCTSTR   lpCaption,  // message box 的 title 的地址
               UINT      uType);     // message box 的类型
```

其中，第一个参数 hWnd 是你要显示的消息框所从属的窗口的句柄。在进行 Windows 编程中，常常要和句柄打交道，以后的章节就会较详细地介绍它们。在 HelloWorld1 程序里，hWnd 设为 NULL，这表示消息框从属于桌面。

第二个参数 lpText 是一个以 null 结尾的字符串，它是你要显示的消息字样。

第三个参数 lpCaption 也是一个以 null 来结尾的字符串，它将作为消息框的标题显示出来。

最后，uType 是希望消息框所具有的显示样式。有许许多多可用的样式，它们定义成为一族可以相互组合产生更复杂样式的标志（参见图 1.2）。你可在 Win32 文档中找到完整的列表，如表 1.2 所示。

为了组合这些标志，可以使用逻辑或（Logical OR）。例如，为了创建一个具有 OK 和 Cancel 按钮、并带有 STOP 图标的消息框，可以将 uType 的值设为 MB_OKCANCEL | MB_ICONSTOP，非常简单。

表 1.2　消息框 uType 样式

一 般 设 置	
标志（flag）	意　义
MB_ABORTRETRYIGNORE	消息框带有三个按钮：Abort，Retry 和 Ignore
MB_OK	消息框带有一个按钮：OK。这也是默认标志
MB_OKCANCEL	消息框带有两个按钮：OK 和 Cancel
MB_RETRYCANCEL	消息框带有两个按钮：Retry 和 Cancel
MB_YESNO	消息框带有两个按钮：Yes 和 No
MB_YESNOCANCEL	消息框带有三个按钮：Yes，No 和 Cancel

续表

图标类型设置	
标志（flag）	意　义
MB_ICONWARNING	消息框中显示一个惊叹号图标
MB_ICONASTERISK	消息框中显示一个在圆圈中写着小写字母 i 的图标
MB_ICONQUESTI0N	消息框中显示一个问号图标
MB_ICONSTOP	消息框中显示一个 Stop 记号的图标

像大多数 Win32 函数调用一样，MessageBox 会给出一个返回值。在 HelloWorld1 例子里，返回值可以忽略，但经常会需要接收用户的反馈。如果没有足够的内存来创建消息框，MessageBox 函数将返回零。其他返回值简单列出如下：

IDABORT　　单击 Abort 按钮。
IDCANCEL　　单击 Cancel 按钮。
IDIGNORE　　单击 Ignore 按钮。
IDNO　　单击 No 按钮。
IDOK　　单击 OK 按钮。
IDRETRY　　单击 Retry 按钮。
IDYES　　单击 Yes 按钮。

这样第一课就算上完了。当然，必须承认，还没有具体讲述应该怎样创建一个应用程序的窗口。但下面会循序渐进地讲述。读者一定在疑惑那些变量名的奇怪前缀，如 lp、sz 和 h 等。微软的程序员们在编程时都使用一种共同的约定——匈牙利符号表示法（Hungarian Notation）。下面介绍一下匈牙利表示法。

1.3.1 匈牙利表示法

匈牙利表示法是微软雇员 Charles Simonyi 博士的发明。它之所以称为匈牙利表示法，是因为 Charles 来自匈牙利。基本上，这是一个命名约定：在每一个变量名前添加表示变量类型的字母前缀，并继以一个大写字母开头的对变量的简短描述。例如，如果需要用一个整数型变量来保存游戏中的得分，会把它命名为 iScore。匈牙利表示法的发明来自于为微软程序员建立一个可遵循的编程规范的迫切需求。如果一个公司所有的程序员各自使用不同的命名约定，一切将变得非常混乱。

尽管这个系统看上去挺繁琐，有些名字看起来似乎更像来自一个遥远国家的语言，但当你一旦接受了之后就会觉得它确实挺有用。有一些程序员厌恶这种表示法，在 Usenet 上充斥着关于匈牙利表示法的好与不好的争论帖。竟然有那么多人为此措辞严厉，情绪激动，这真是有点不可思议。这只是一个个人爱好问题。但无论持何种观点，了解这种命名约定对将来学习 Windows 编程还是有帮助的。这是一些起码的知识。这些前缀的具体含义是什么？可参看表 1.3，在表中列出了一些较常用的前缀。

表 1.3　匈牙利记号表示法前缀

前　缀	类　型
sz	指向一个以零字符结尾的字符串中的第一个字符
str	字符串
i	int
n	数或 int
ui	Unsigned int
c	char
w	WORD（unsigned short）
dw	DWORD（unsigned long）
fn	函数指针（function pointer）
d	Double
by	byte
l	long
p	pointer
lp	long pointer
lpstr	指向字符串的 long pointer
h	句柄（handle）
m_	类成员（class member）
g_	全局型（global type）
hwnd	窗口的句柄（Window handle）
hdc	Windows 设备上下文（device context）的句柄

现在，当看到诸如 g_iScore、szWindowClassName 和 m_dSpeed 的变量名，你就能够知道它们具体描述什么了。你会在笔者的代码中发现，笔者使用了个人版本的匈牙利表示法，因为觉得它极为有用。有了匈牙利表示法，就可以在阅读他人的代码的时候立即理解变量类型而不用来回查找它们的定义。还有一点必须在这里说明的是，笔者并不对每个变量都使用匈牙利表示法。如果在一个小函数里用到一个变量，那么笔者会用任何我觉得适当的名字，因为这变量是什么是显而易见的。例如，一个函数来接收一个错误字符串作为参数，并显示一个消息框，程序如下：

```
void ErrorMsg(char* error);
```

而非：

```
void ErrorMsg(char* szError);
```

另外，笔者用大写字母 C 作为所有类的前缀，大写字母 S 作为所有结构的前缀（读者可以想一下这是为什么）。在一些诸如 2D/3D 矢量和 STL vector 类的声明上，也使用了笔者所习惯的匈牙利表示法。由此，一个典型的类定义是这样的：

```
class CmyClass
{
private:
  int           m_iHealth;
  S2DVector     m_vPosition;
  vector<float> m_vecfWeights;
public:
  CMyClass();
};
```

明白了这些，让我们开始学习吧……

1.3.2 第一个窗口

在创建窗口之前，必须首先创建自己的窗口类并将它注册，这样操作系统才能知道希望此窗口以何种类型显示，并如何动作。窗口可以是任意尺寸，边界、滚动条和菜单这些都可有可无。窗口上也可以有一些任意颜色的按钮或工具条。有许多的可选项可以选择。即使是像刚才创建的最简单的消息框也是一种预定义的窗口。最常需要的是一个能够在其中显示文本和绘制图形的窗口。下面将要介绍窗口的创建方法。程序可以在光盘上的HelloWorld2 项目中找到。

1.3.2.1 注册窗口

窗口类型是由生成窗口的类的结构来定义。为此，必须编写一个 WNDCLASSEX 结构。结构形式如下：

```
typedef struct _WNDCLASSEX{
   UINT      cbSize;
   UINT      style;
   WNDPROC   lpfnWndProc;
   int       cbClsExtra;
   int       cbWndExtra;
   HANDLE    hInstance;
   HICON     hIcon;
   HCURSOR   hCursor;
   HBRUSH    hbrBackground;
   LPCTSTR   lpszMenuName;
   LPCTSTR   lpszdassName;
   HICON     hIconSm;
} WNDCLASSEX;
```

下面进行详细解释。

注意： Windows 以前使用一个称为 WNDCLASS 的结构，但微软经过改进后设计了新的 WNDCLASSEX 结构。有一些其他结构也因同样理由而具有 EX 后缀。其实老的结构也还是可用的，不过没有意义。

cbSize 用来保存 WNDCLASSEX 结构的大小，以 byte 为单位。可以这样设置：

```
cbSize = sizeof(WNDCLASSEX);
```

必须确保这一参数设置正确，否则注册类时，Windows 将会提示有错误。

style 是窗口应具有的显示样式。可以选择几个并将它们 OR（逻辑或）起来，样式最常用的设置是：

```
style = CS_HREDRAW | CS_VREDRAW;
```

这告诉 Windows API，这个窗口在用户改变窗口大小（高度或宽度）时需要重画。在 Win32 API 帮助文件中还可以找到更多样式选项。

lpfnWndProc 是一个指向 Windows Procedure（Windows 过程）的函数指针。下面将作详细讨论。

cbClsExtra/cbWndExtra：大多数情况设置为 0 即可。它们存在的意义仅仅是，如果必要的话（其实一般没有必要），可以为窗口类多分配几个字节的存储空间。

HInstance：即 WinMain 里面的 hInstance 参数。这用于设置窗口类结构。

```
hInstance = hInstance;
```

hIcon 应用程序所使用的图标的句柄。此图标在用 Alt+Tab 组合键来切换任务的时候会显示出来。可以用 Windows 的默认图标之一，也可以自定义图标并作为一个资源而包含在程序里。在下一章里将讲述如何编写程序。读者可调用 LoadIcon 来获取图标句柄。

可用下面这行语句来选用一个默认图标：

```
hIcon = LoadIcon (NULL, IDI_APPLICATION);
```

hCursor：hCursor 是应用程序所使用的鼠标光标的句柄。一般，可以把它设置为默认光标，即箭头光标。要取得光标句柄，应该采用下面的形式来调用 LoadCursor：

```
hCursor = LoadCursor(NULL, IDC_ARROW);
```

hbrBackground 是一个用以指定创建窗口的客户区背景色的域。客户区是窗口中实际绘图和写字的区域。hbr 前缀表示它是一个画刷句柄（handle to a brush）。Windows 用画刷来指定填充区域的颜色甚至也包括预定义图案（pattern）。可以定义用户自己的画刷，也可以选用 API 已经定义好的许多画刷。例如，如果希望背景以白色填充，可以这样设：

```
hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

在第 2 章介绍 Windows 的进一步编程知识时将会更详细地讨论画刷。

lpszMenuName 用来设置菜单的名字，如果不需要下拉菜单，如 edit、save 和 load，可以把这项设为 NULL。下一章里也将具体介绍。

lpszClassName 就是赋予窗口类的名字。这个名字是自定义的。

hIconSm 是一个较小的图标的句柄，该图标将在任务栏和应用程序窗口的左上角出现。和往常一样，既可以亲手设计自己的图标，并作为资源包含在程序里，也可以使用默认的某一个图标。

技巧：在这里值得一提的是，很少有程序员能够真正记得所有这些参数。大多数程序员倾向于保留一个基本的窗口模板文件，这样可以在每当要开始新项目的时候将程序复制和粘帖即可，这样做很方便。

当窗口类创建完毕之后，需要调用 RegisterClass 函数来注册它，应将一个指向 WNDCLASSEX 结构的指针传给该函数。以光盘上的 HelloWorld2 程序为例，程序如下：

```
//window 类的结构
WNDCLASSEX         winclass;

//在 window 类的结构中首先编写：
winclass.cbSize          = sizeof(WNDCLASSEX);
winclass.style           = CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc     = WindowProc;
winclass.cbClsExtra      = 0;
winclass.cbWndExtra      = 0;
winclass.hInstance       = hInstance;
winclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION);
winclass.hCursor         = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground   = (HBRUSH)GetStockObject(WHITE_BRUSH);
winclass.lpszMenuName    = NULL;
winclass.lpszClassName   = g_szWindowClassName;
winclass.hIconSm         = LoadIcon(NULL, IDI_APPLICATION);

//注册此 window 类
if (!RegisterClassEx(&winclass))
{
  MessageBox(NULL, "Class Registration Failed!", "Error", 0);

  //退出应用程序
  return 0;
}
```

这样就创建了一个窗口类：它具有白色的背景、不设置菜单、采用了默认的箭头光标和图标，并且当用户改变窗口尺寸时会重画该窗口。注意在注册类的时候，使用了 MessageBox 函数以向用户报告错误。

1.3.2.2 创建窗口

下面创建并显示窗口。这里需要调用 CreateWindowEx 函数。函数原型如下：

```
HWND CreateWindowEx(
  DWORD     dwExStyle,        // 扩展窗口式样
  LPCTSTR      lpClassName,   // 指向所注册类名的指针
  LPCTSTR      lpWindowName,  // 指向窗口名的指针
  DWORD     dwStyle,          // 窗口的样式
  int          x,             // 窗口左上角的 x 位置
  int          y,             // 窗口左上角的 y 位置
```

```
    int         nWidth,         // 窗口的 width
    int         nHeight,        // 窗口的 height
    HWND        hWndParent,     // 指向父窗口或 owner 窗口的句柄
    HMENU       hMenu,          // 菜单句柄或子窗口的标识符
    HINSTANCE   hInstance,      // 指向应用程序实例的句柄
    LPVOID      lpParam         // 指向窗口创建数据(creation data) 的指针
);
```

dwExStyle 用来设置需要的任意扩展式样。表 1.4 里列出了一些可用的式样，但在这个 Windows 程序里是不需要使用的，设为 NULL 即可。

表 1.4　扩展窗口式样

扩展式样标志	描　　述
WS_EX_ACCEPTFILES	使用该式样创建而成的窗口接受拖放文件
WS_EX_APPWINDOW	使一个可见顶级窗口出现在任务栏上
WS_EX_CLIENTEDGE	指定该窗口使用凹陷的边框
WS_EX_DLGMODALFRAME	创建一个有双线边框的窗口；此窗口可以（可选）通过在 dwStyle 参数里使用 WS_CAPTION 式样而具有一个标题栏
WS_EX_CONTEXTHELP	在窗口的标题栏里加入问号按钮。当用户单击该按钮时，光标变成一个带有问号的光标。此时若用户单击子窗口，则子窗口会收到一条 WM_HELP 消息。子窗口应当将此消息传递给其父窗口函数，由其通过 HELP_WM_HELP 命令调用 WinHelp 函数。Help 程序会显示一个包含关于子窗口的帮助信息的弹出窗口
WS_EX_WINDOWEDGE	指定该窗口使用凸出的边框

lpClassName 是字符串指针，对象是窗口类的名字。在这里以及后面的所有例子里，均被设为 g_szWindowClassName。

可以把窗口类名和应用程序名作为两个全局字符串 g_szWindowClassName 和 g_szApplicationNmae 存放在 main.h 的顶部。这样做日后修改名字更加容易，只需要看一个地方就可以了。

lpWindowName 是希望用的、显示在顶端的应用程序标题。本书的例子里被设为 g_szApplicationName。

dwStyle 包含了用于设置窗口样式（或式样）的标志。有很多可供选用的样式。表 1.5 中列出了一些较为常用的样式。

表 1.5　窗口样式

样 式 标 志	描　　述
WS_BORDER	创建的窗口具有细线边框
WS_CAPTION	创建的窗口带有标题栏（包括 WS_BORDER 式样）
WS_HSCROLL	创建的窗口带有水平滚动条
WS_MAXIMIZE	创建的窗口初始为最大化
WS_OVERLAPPED	创建的窗口是迭加的，此窗口有标题栏和边框

续表

WS_OVERLAPPEDWIN0OW	用以下式样创建一个迭加窗口：WS_OVERLAPPED、WS_CAPTION、WS_SYSMENU 、 WS_THICKFRAME 、 WS_MINIMIZEBOX 和 WS_MAXIMIZEBOX。和 WS_TILEDWINDOW 式样相同
WS_POPUP	创建一个弹出式窗口。这个式样不能和 WS_CHILD 式样一起用
WS_THICKFRAME	创建的窗口的边框可变大小
WS_VSCROLL	创建的窗口带有垂直滚动条

x、y 的值用来设置窗口的左上角的位置。在 Windows 里 y 的值是自上而下从零开始增加的，如图 1.6 所示。

nWidth 和 nHeight 指定了窗口的宽度和高度。笔者把这两个值用#define 定义在 defines.h 这个头文件里，命名为 WINDOW_WIDTH 和 WINDOW_HEIGHT。

hWndParent 是一个指向窗口的父窗口（或所有者）的句柄。如果该窗口为主程序窗口，将句柄设为 NULL，这就使 Windows 将桌面当作父窗口。

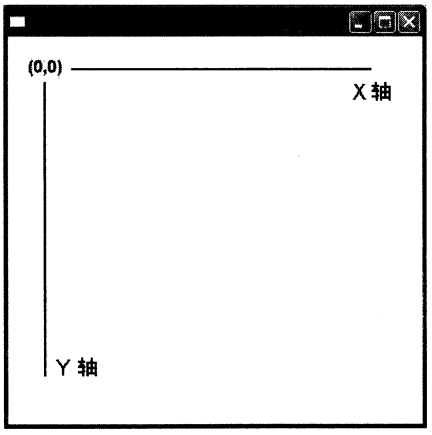


图 1.6　上下颠倒的窗口 Y 坐标轴

hMenu 是一个菜单句柄。菜单出现在应用程序顶部，我将在后面讨论菜单，现在设为 NULL 即可。

hInstance：使用从 WinMain 里的 hInstance。

lParam 只是在创建多文档界面（MDI）窗口时才会用到，这里设成 NULL 即可。

下面是 HelloWorld2 对 CreateWindowEx 的完整调用代码：

```
hWnd = CreateWindowEx(NULL,                    // 扩展式样
                      g_szWindowClassName,     // window 类名
                      g_szApplicationName,     // window 标题
                      WS_OVERLAPPEDWINDOW,     // window 式样
                      0,                       // x 初始位置
                      0,                       // y 初始位置
                      WINDOW_WIDTH,            // window 初始宽度
                      WINDOW_HEIGHT,           // window 初始高度
                      NULL,                    // 父窗口句柄
```

```
                    NULL,                    // window 菜单句柄
                    hInstance,               // 程序实例句柄
                    NULL);                   // 创建参数
```

最后，用下面这两个调用语句使窗口变为可见：

```
ShowWindow (hwnd, iCmdShow);
UpdateWindow (hwnd);
```

ShowWindow 需要两个参数。第一个是需要显示的窗口——也就是你刚才创建的窗口的句柄 hWnd。第二个参数是一个指定窗口是否可见和是否应当最小化、正常或最大化的标志。要记住，iCmdShow 使用的值，就是用户在创建快捷方式或把程序加入开始菜单时所选用的值。

然后利用 UpdateWindow 使窗口的客户区以为背景画刷指定的颜色画出来。

由于已经注册了一个自定义的窗口类，在程序退出以前，必须 Unregister（撤销）这个类。可以使用 UnregisterClass 来实现，如：

```
BOOL UnregisterClass (
  LPCTSTR lpClassName,    // 指向类名 ClassName 的字符串的指针
  HINSTANCE hInstance     // 指向应用程序实例 instance 的句柄
);
```

然后将窗口的类名 ClassName，以及它的实例句柄 hInstance 传给这个函数，操作就全部完成了。

编译并运行 HelloWorld2 例子。在运行 HelloWorld2.exe 的一瞬间，用户会看见新创建的窗口出现，但立即就消失了。因此，如果要真正使用新创建的窗口，就必须找到使窗口停留在屏幕上的方法。在调用 UpdateWindow 后再用一无限循环可达到这一目的，但无法利用正常的手段来关掉这个窗口了。因此，需要寻找另一种解决方案，这个解决方案就是非凡、奇妙的 Windows 消息循环。

1.3.3　Windows 消息循环

每当打开一个应用程序的时候，Windows（操作系统）都在后台运行——移动光标、检查是否改变了窗口大小或移动了窗口，检查是否切换了任务，检查是否要求应用程序最小化或最大化等。之所以能够处理所有这些，是因为 Windows 是一个事件驱动（Event-driven）的操作系统。事件，或者更常见地叫做消息，是在用户作各种操作时产生出来的，并被储存在一个消息队列（message queue）中，直到当前活动的应用程序来处理它们。当用户移动窗口、关闭窗口、移动光标或在键盘上按下一个键时，Windows 不断地创建新的消息。因此，程序员必须找到一种从队列中读取消息并进行处理的方法。为此，在 UpdateWindow (hWnd)之后，立即使用一个被称作 Windows message pump（Windows 消息循环）的循环过程：

```
//保存从队列中获取的任意 Windows 消息
```

```
MSG msg;
//消息循环的入口处
while(GetMessage (&msg, NULL, 0, 0))
{
  TranslateMessage (&msg);
  DispatchMessage(&msg);
}//结束消息循环
```

上面这一段代码中，变量 msg 用来保存消息，它是一个结构体，定义如下：

```
typedef struct tagMSG{
  HWND        hwnd;
  UINT        message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
}MSG;
```

hwnd 是将要包含该消息的窗口的句柄。

message 是消息标识符。存在大量可用的消息标识符，它们几乎都以 WM_开头。举例来说，当程序窗口第一次被创建时，Windows 把一条 WM_CREATE 消息放入消息队列，而当程序窗口被关闭时，则是一条 WM_CLOSE 消息。更多的标识符可在 WINUSER.H 中找到。当用户具备了更多 Windows 编程知识时，会知道如何处理更多的消息，甚至如何创建自定义的消息。表 1.6 中列出了一些常用的 Windows 消息标识符。

表 1.6　常用 Windows 消息

消　息	描　述
WM_KEYUP	当用户放开某个非系统按键时，会发出此消息
WM_KEYDOWN	和上一条类似，但是在按键按下时发出
WM_MOUSEMOVE	每次光标移动时就会发出这条消息
WM_SIZE	用户改变窗口尺寸时就会发出这条消息
WM_VSCROLL	垂直滚动条移动时就会发出这条消息
WM_HSCROLL	水平滚动条移动时就会发出这条消息
WM_ACTIVATE	有两种情况会发出这条信息，一是窗口被用户激活时，二是窗口被取消激活时——可以通过 wParam 判断到底是哪种

wParam 和 lParam 是两个 32 位参数，它们包含了消息的一些附加信息。例如，如果消息是 WM_KEYUP（按键放开），则参数 wParam 指出了是哪一个按键刚刚被放开的信息，而 lParam 给出了诸如上一个按键状态和重复次数的附加信息。

time 是消息进入事件队列的时间。

pt 是一个 POINT 结构，其中存有消息进入队列的时候的鼠标坐标。POINT 结构定义如下：

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
}POINT;
```

下面逐行地解释消息循环。

```
while (GetMessage (&msg, NULL, 0, 0))
{
```

GetMessage 是从消息队列里取出消息的函数。如果队列里一条消息也没有，它会一直等待，直到出现一条消息。GetMessage 获得消息的办法，是利用它的第一个参数，即传递一个 MSG 结构的指针让 Windows 来填充这个结构。第二个参数是需要获取消息的窗口的句柄。在这个例子里希望处理所有 Windows 收到的消息，可以把这个参数设为 NULL。第 3 和第 4 个值是暂时不需要了解的附加过滤器，因此都设为 0。如果 GetMessage 收到消息不是 WM_QUIT，则都应返回一非零值。最后，当 GetMessage 收到一条消息后，该条消息就从队列中删去。

```
TranslateMessage (&msg);
```

当一按键被按下时，用户会收到一条 WM_KEYDOWN 消息，而放开按键时，则 Windows 会产生一个 WM_KEYUP 消息。如果按下 Alt 键，那就产生了一条 WM_SYSKEYDOWN 消息，而若按住 Alt 键，再把一个键按下并松开，将得到一个 WM_SYSKEYUP 消息。如果用户不停地在键盘上按下了许多键，消息队列很容易会被消息堵塞。而 TranslateMessage 的作用，就是将这样的多个键盘消息，组合成为一个单个的消息 WM_CHAR。

```
DispatchMessage(&msg);
```

利用 DispatchMessage 这个函数可以处理从消息列中抽取的消息。注册窗口类时，需要在 lpfnWndProc 域中填入一个指向 Windows Procedure（Windows 过程）的函数指针吗，Windows 过程就是 DispatchMessage 发送消息的回调函数（Callback Function），命名为 WindowProc。由 WindowProc 对相应的消息进行处理。例如，当用户按下一按键，消息循环就收到 WM_KEYDOWN 消息，WindowProc 接着对这个按键进行必要的处理；如果用户需要拖动鼠标来改变窗口尺寸，则消息循环就会接收到一个 WM_SIZE 消息，而最终它会发给 WindowProc，这样，如果处理程序正确的话，显示窗口就被相应地调整尺寸。

这样，应用程序停留在消息循环的循环中不停地处理消息，直到用户关闭窗口、引发 WM_QUIT 消息，这时 GetMessage 返回零，于是应用程序退出消息 pump 并终止执行。

1.3.4　Windows 过程

为了掌握创建 Windows 应用程序的基础，需要了解 Windows 过程。

Windows 过程的定义形式如下：

```
LRESULT CALLBACK WindowProc(
  HWND    hwnd;          //window 的句柄
  UINT    uMsg;          //message 的标识
  WPARAM  wParam;        //第一个 message
  LPARAM  lParam;        //第二个 message
);
```

LRESULT 是窗口函数的返回类型。一般正常情况下是个非零值。

CALLBACK 用来告诉 Windows，WindowProc 是一个回调函数，每当 Windows 产生了需要处理的事件时，就会调用此函数。不需要一定要求将 Windows 过程的名称定为 WindowProc，可以使用自己所喜欢的名字，笔者在这里将 Windows 过程定为 WindowProc。

注意：可以同时打开几个窗口，各自具有不同的句柄和分开定义的 WindowProc 来处理各自的消息。

hwnd 是用户要为它处理消息的窗口的句柄。

uMsg 是待处理消息的 ID。它与 MSG 结构中 message 域是相同的。

wParam 和 lParam 则和 MSG 结构中包含的 lParam 和 wParam 相同，用于表示有关信息的其他附加信息。

下面，举例说明一个为 HelloWorld 3 程序写的简单的 Windows 过程。

```
LRESULT CALLBACK WindowProc(HWND    hwnd,
                            UINT    msg,
                            WPARAMwParam,
                            LPARAMlParam
                              )

{
 switch (msg)
 {
//当应用程序窗口首次建立时，就会送出 WM_CREATE 这一 msg
case WM_CREATE:
    {
     PlaySound("window_open.wav", NULL, SND_FILENAME|SND_ASYNC);
     return 0;
    }
   case WM_PAINT:
    {
     PAINTSTRUCT ps;
     BeginPaint (hwnd, &ps);

     //**这里留作在屏幕上画图所需之代码使用**

     EndPaint (hwnd, &ps);
     return 0;
    }
   case WM_DESTROY:
```

```
    {
      //kill 应用程序，它将送出一个 WM_QUIT 消息
      PostQuitMessage (0);
      return 0;
    }
  }//结束 switch 语句
  return DefWindowProc (hwnd, msg, wParam, lParam);
}//结束 WindowProc
```

WindowProc 可以看作是一个大的 switch 语句。上例 WindowProc 只处理了 3 种消息：WM_CREATE、WM_PAINT 和 WM_DESTROY。但运行一下程序，会发现这些代码已足够实现一个可以移动、可以最小化、最大化甚至也能够改变大小的一个窗口，如图 1.7 所示。并且，作为对坚持学习到现在的一个奖励，这段程序在打开时甚至还能播放一曲 wav 文件。

下面讲述 WindowProc 的各部分。

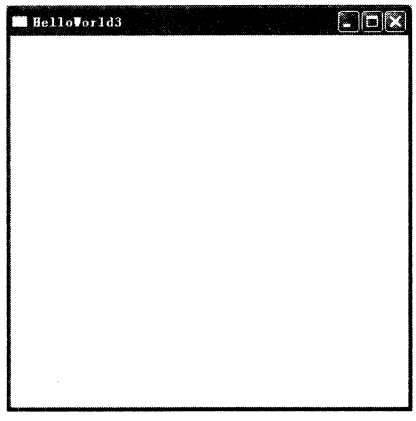


图 1.7　第一个窗口

1.3.4.1　WM_CREATE 消息

WM_CREATE 消息是窗口第一次创建时产生的。这一消息被消息 pump 俘获后，最终被发送到 WindowProc，再由后者的 switch 语句进行处理。

```
case WM_CREATE:
{
  PlaySound("window_open.wav", NULL, SND_FILENAME | SND_ASYNC);
  return 0;
}
```

在这个例子里，当程序执行到 switch 语句的 WM_CREATE 段落时，在打开窗口的同时，调用 API 里的 PlaySound 函数，播放了一曲 wav 文件。PlaySound 定义如下：

```
BOOL PlaySound(
  LPCSTR pszSound,
  HMODULE hmod,
```

```
    DWORD   fdwSound
);
```

pszSound 是一个字符串，指定了要播放的声音文件。如果它的值为 NULL，则所有当前正在播放中的声音都会停下来。

hmod 是包含该 wav 作为资源的可执行文件的句柄。下一章将会介绍各种资源。在这里把此参数设成 NULL 就行了。

fdwSound 是用来控制声音播放的一个标志。完整的列表可以查阅一些文档。笔者用了 SND_FILENAME 标志和 SND_ASYNC 标志，这样 PlaySound 就知道 pszSound 是文件名，而且声音是异步播放的。异步意味着 PlaySound 函数将开始播放声音并在播放结束后立即返回调用点。

注意：要使用 PlaySound 函数，必须用 include 语句来包含一个 Windows 多媒体库，因此，在试图编译 HelloWorld3 之前，在工程的设置中确认编译器会连接 winmm.lib。

1.3.4.2 WM_PAINT 消息

任何时候，只要系统需要重画应用程序窗口的任何部分，都会产生一个 WM_PAINT 消息。许多初学 Windows 编程的人都认为，只要创建好窗口，接下来的任何事情 Windows 操作系统都会代办。事实并非如此。每当用户移动另一个窗口遮蔽了一部分窗口，或对窗口最大、最小化和改变尺寸时，WM_PAINT 消息就会发出，但必须由用户自行重画。幸好，Windows 会自动计算需要重画的区域。如果窗口只有一部分被覆盖后又要重显，Windows 并不重画整个窗口，而只画那些恢复为可见的部分。需要重画的区通常称为无效区（Invalid Region）或更新区（Update Region）。如图 1.8 所示。当一条 WM_PAINT 消息产生，在调用 BeginPaint 之后，Windows 便知道哪些区域已经恢复为有效，所有相关的 WM_PAINT 消息（可能积累了数条）都会从消息队列中删去。

```
case WM_PAINT:
{
  PAINTSTRUCT ps;
  BeginPaint (hwnd, &ps);
  //**这里就是用来向屏幕画图的地方**
  EndPaint (hwnd, &ps);
  return 0;
}
```

处理 WM_PAINT 消息的第一件事就是创建一个 PAINTSTRUCT 结构。BeginPaint 使用此结构来传递重画窗口所需的信息。下面是 PAINTSTRUCT 的定义：

```
typedef struct tagPAINTSTRUCT {
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
```

```
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

hdc 是设备描述表（Device Context）句柄。鉴于本章目的只是产生一个空的窗口，可以忽略它，但在下一章里将开始频繁地使用设备描述表在窗口里绘图。

Ferase 用来告诉应用程序创建窗口类时是否应该用指定的颜色重画背景。如果 fErase 非零，背景就会被重画。

rcPaint 是一个 RECT 结构，告诉应用程序哪个区域已经被无效了，需要重画。RECT 结构是一个非常简单的结构，定义了矩形的 4 个顶点，格式如下：

```
typedef struct _RECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;
```

其余的参数是保留给 Windows 的，可以不用理会。

当调用 BeginPaint 时，PAINTSTRUCT 结构 ps 已经被填好，而背景一般也已重画好了。可以进行自定义绘图。最后调用 EndPaint 函数，以通知 Windows 已经绘图完毕。

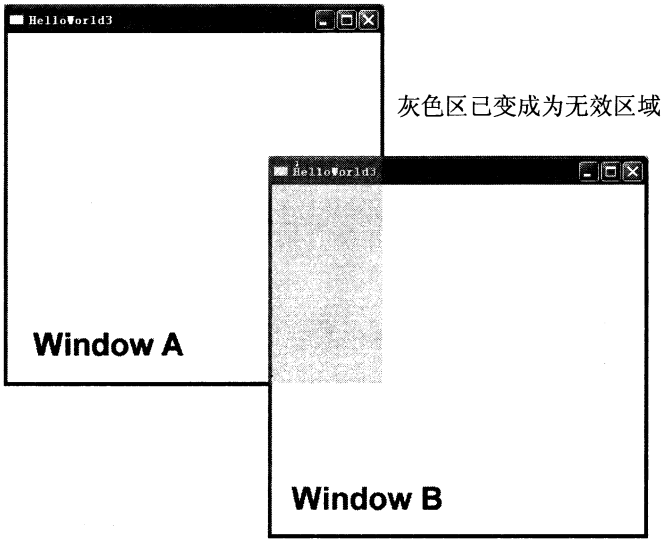


图 1.8　窗口重叠是怎样产生无效区域的

1.3.4.3　WM_DESTROY 消息

这一条消息的含义为：用户要求关闭应用程序窗口。关闭应用程序窗口的做法是调用 PostQuitMessage(0)，后者产生一条 WM_QUIT 消息并置入消息队列。切记，在消息循环中使用到的 GetMessage 函数在遇到 WM_QUIT 消息时会返回 0。这样应用程序就终止了。

```
case WM_DESTROY:
{
  PostQuitMessage(0);
  return 0;
}
```

技巧：如果不用 PostQuitMessage(0)来发出 WM_QUIT 消息，那么尽管应用程序的窗口会被关闭，程序本身仍然在运行！一些应用程序就用了这种方法来缩到任务栏里去。

1.3.4.4 其余的消息

虽然已知道如何处理上述 3 条消息，但那些发往 WindowProc 而未处理的消息怎么处理？Windows 有一个 DefWindowProc 函数，用它可以处理那些用户没有处理的消息。最后应这样从 WindowProc 返回：

```
return DefWindowProc (hwnd, msg, wParam, lParam);
```

Windows 便会处理其他消息。

技巧：尽管程序通常不直接调用窗口过程，但可以使用 SendMessage 函数自己把消息加入消息队列（从而由 WindowProc 处理）。关于 SendMessage 函数的更多细节请查阅 Windows API 文档。

1.3.5 键盘输入

结束本章之前，简单介绍一下键盘的处理。当按下某个键时，产生了一个 WM_KEYDOWN 或 WM_SYSKEYDOWN 消息，再放开这一键时，则产生了一个 WM_KEYUP 或 WM_SYSKEYUP 消息。WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息的使用对 Windows 操作系统来说比对应用程序更为重要，这些消息是在用户按下诸如 Alt+Tab 或 Alt+F4 之类的键时产生的。此时，用户就不必处理 WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息，只要让 DefWindowProc 处理它们就行了。

对于所有这些消息，wParam 包含了正被按下的键的虚拟键代码（Virtual Key Code），而 lParam 包含诸如重复次数、扫描码（scan codes）等附加信息。本书里没有用到任何 lParam 中的信息。

在过去，程序员们写程序时必须直接从键盘的硬件[①]里读取击键的代码。而每一种不同类型的键盘为每一个键产生着不同的编码；这就是所谓的扫描码（scan codes）。这意味着每个厂商都有自己的编码规则，因此，一个键盘上的 c 键的编码在另一个键盘上可能代表的却是个 1 键。幸运的是，Windows 引入了虚拟键代码，解决了这个问题。从而不再需要考察硬件就可以进行编程。表 1.7 列出了一些在游戏中经常用到的虚拟键代码。

[①] 译注：硬件就是键盘缓冲区。

表 1.7　常用虚拟键代码

虚　拟　键	描　　述
VK_RETURN	Enter（回车）键
VK_ESCAPE	Escape（退出）键
VK_SPACE	SPACE（空格）键
VK_TAB	Tab（制表）键
VK_BACK	Backspace（退格）键
VK_UP	Up Arrow（向上箭头）键
VK_DOWN	Down Arrow（向下箭头）键
VK_LEFT	Left Arrow（向左箭头）键
VK_RIGHT	Right Arrow（向右箭头）键
VK_HOME	Home（行首）键
VK_PRIOR	Page Up（翻上页）键
VK_NEXT	Page Down（翻下页）键
VK_INSERT	Insert（插入）键
VK_DELETE	Delete（删除）键
VK_SNAPSHOT	Print Screen（截屏）键

若用户按下了数字或字母键，则虚拟键代码就是该数字或字母的 ASCII 码。为了读取键盘的输入，只需要为 WM_KEYDOWN 或 WM_KEYUP 消息编写一处理代码。在 HelloWorld4 例程中，加入了一段处理 WM_KEYUP 消息的简短代码，用来检测 Escape 键是否按下。如果检测到 Escape 键已按下，就让应用程序结束。下面列出它的 WindowProc 中相关的代码：

```
case WM_KEYUP:
{
  switch(wParam)
  {
    case VK_ESCAPE:
    {
      //如果按下了 Escape 键，就退出程序
      PostQuitMessage(0);
    }
  }
}
```

由上可以看到，首先检测到 WM_KEYUP 消息，然后根据此消息的 wParam 部分来创建一个 switch 语句，来区分其中保存的不同的虚拟键代码。如果检测到的虚拟键代码是 Escape 键的代码 VK_ESCAPE，就调用方法 PostQuitMessage(0)来发送 WM_QUIT 消息并终止应用程序。

另外有一种获取键盘信息的方法，是使用 GetKeyboardState、GetKeyState 或

GetAsyncKeyState 3 个函数中的任意一个。它们都是很有用的函数，特别是在游戏编程时更是如此。利用它们，你可以在程序的任何地方来测试按键，不再需要麻烦消息循环。其中 GetAsyncKeyState 可能是最为有用的，因为可简单地调用它来检查某个特定键是否处于按下状态。其原型如下：

```
SHORT GetAsyncKeyState (
  int vKey      //虚拟键代码
);
```

要检测某个键是否已按下，将该键的虚拟键码传给函数 GetAsyncKeyState 作为参数，并检查返回值的最高有效位（Most Significant Bit，即最左的一位）是否等于 1。例如，测试向左箭头键是否已被按下的代码如下：

```
if (GetAsyncKeyState (VK_LEFT) & 0x8000)
{
  //**这里插入进行左移的代码**
}
```

如果对另外两个函数也有兴趣的话，可以查阅相关文档——在实践中多半用不着它们。

第一个 Windows 程序已经完成。虽然这个 Windows 程序除了等待作移动或关闭窗口等操作外，其他什么事情也没有做，但会使读者感到一种极强的成就感。

第 2 章　Windows 编程进阶

“我总忍不住要表达我不喜欢那些认为理解会破坏他们的经验的人们……他们怎么会知道的呢？”

Marvin Minsky

在前一章里已经学过了如何创建窗口——现在开始介绍如何利用 Windows 的绘图及文字工具。当熟悉了这些内容之后，将确切地说明什么是资源，怎样利用它们来创建自己的菜单、图标、鼠标光标等内容。学完本章后，将具备足够的 Windows 编程基本知识来理解本书其余章节所有源代码范例，然后就可以进入遗传算法和神经网络的学习了。其中偶而会涉及一些必须让读者了解的额外知识，但一定是很容易解释清楚的。

2.1　Windows 图形设备接口

Windows 负责屏幕上绘制图形的部分称为图形设备接口（Graphics Device Interface，GDI），这是许多可供调用的函数的集合。其中包括绘制 shape[①]、画线、填充 shape、输出文字、裁剪、设置颜色和画笔宽度等。可在窗口中显示的图形大体上可被归为以下 4 类：

- 文本（**text**）。GDI 中关于文本的部分显然非常重要。GDI 中提供了大量的格式化文本和输出工具，用户几乎可以得到想得到的任何一种方式在屏幕上创建和输出文字。
- 直线（**line**）、形状（**shape**）、曲线（**curve**）。GDI 对绘制直线、曲线（Bezier 曲线）、基本形（如矩形和椭圆）和多边形提供了充分的支持。多边形就是由一系列相互连接的顶点组成的 shape，最后一点和第一点相连，形成闭合图形。绘图时，首先需要创建一支用来绘图的画笔（Pen），然后用该画笔绘制需要的 shape。
- 位图（**bitmap**）。GDI 提供了许多处理位图的函数。可以加载位图、缩放位图、保存位图或从一处复制位图到另一处。位图复制通常被游戏程序员称为 blitting（图像位块传送）。
- 填充区域（**filled area**）。绘图时除了画笔，还可以创建用户自己的画刷（Brush）。画刷可用来填充屏幕上的区域（region）和 shape。

除了提供绘制和输出文本函数外，GDI 中还有许多功能用于定义区域和路径、处理裁剪、定义调色板和输出到其他设备如打印机等。如果要解释 GDI 的无数个细节，工作量实在是太大了，这里只能教一些基本的内容。

[①] 译注：可译为形、形状等，shape 在这里具体是指圆、椭圆、矩形、多边形、扇形等各种具有面积的平面几何图形，但不包括直线。因译名形、形状很难反映上述特征，所以本书今后直接使用 shape。

注意：在游戏界里，GDI 是以慢出名的。所谓慢，是与其他 API，如 OpenGL 或微软的 DirectX 相比较而言的。在例子中选用 GDI，是因为它简单好用、也好理解，使用它也足够快，而且更重要的是，在代码中不会充斥混淆视听的复杂 API 调用。

2.1.1 设备描述表

设备描述表——又称 DC——在使用 GDI 的图形文字绘制中扮演很重要的角色。在朝任何图形输出设备——屏幕、打印机、甚至是内存中的一块位图——开始绘图之前，都必须获得该设备的设备描述表的句柄。画刷、画笔、光标、桌面、窗口实例（hInstance）、图标、位图……，都有相应的句柄。句柄就像执照。要开车需要驾驶执照，要经营酒店也需要执照；类似地，也需要取得对特定类型的对象或设备的句柄，来进行想要的操作。如果向 Windows 发出请求："是否可以在这个窗口上绘图？"，则 Windows 就会赋予用户一个执照——窗口句柄——这样就可以绘图了。

2.1.1.1 如何得到句柄（Handle）？

可以有几种方法用来取得设备描述表句柄，为了简单，将用缩写 HDC 来称呼设备描述表句柄。

在第一章中处理 WM_PAINT 消息的内容时？首先创建的是 PAINTSTRUCT 结构，Windows 将其填充，以描述关于窗口的详细信息。PAINTSTRUCT 结构定义如下：

```
typedef struct tagPAINTSTRUCT {
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

现在可以发现，第一个域 hdc，就是 HDC。如果要在 WindowProc 的 WM_PAINT 部分进行绘图，便可以将 hdc 作为窗口句柄。

注意：从 PAINTSTRUCT 中获得的句柄只对在 RECT 结构 rcPaint 定义的区域中绘图有效。如果想要在此区域外绘图，就需要从其他途径获得 HDC。

当调用 BeginPaint 时，它不但返回一个 HDC，同时还填充了 PAINTSTRUCT 结构。因此，另一个获得 HDC 的方法如下：

```
case WM_PAINT:
{
   PAINTSTRUCT ps;
   HDC hdc;
   hdc = BeginPaint (hwnd, &ps);
   //**在这里使用 hdc 作图
   EndPaint (hwnd, &ps);
```

```
    return 0;
}
```

但你不需要将所有的绘图工作都集中在 WindowProc 的 WM_PAINT 部分中进行，只要从 Windows 那里得到了 HDC，你就可以在任何时候进行绘图。为了确保在任何时候从 Windows 获得一个 HDC，可以调用 GetDC 函数：

```
HDC hdc = GetDC (hWnd);
```

hWnd 为你要获取 HDC 的窗口的句柄。无论何时用这个方法创建 HDC，一定不要忘记在使用完毕之后释放它。可以使用 ReleaseDC 函数来释放它：

```
ReleaseDC (hWnd, hdc);
```

没有在 WM_PAINT 中调用 ReleaseDC 是因为 EndPaint 函数自动释放了 DC。但如果创建了 HDC 而事后没有释放，就会出现资源泄漏问题，慢慢地程序也可能出现许多不曾听说的故障。甚至可能导致系统崩溃。记住这一点很重要。

如果需要的话，也可以获取一个应用于整个窗口（包括系统菜单和标题栏域）的 HDC，而不仅仅是客户区。为此可以用 GetWindowDC (hWnd)来实现：

```
HDC hdc = GetWindowDC (hWnd);
```

也可以获得对整个屏幕的 HDC，只要在调用 GetDC 的时候用 NULL 作为参数即可。

```
HDC hdc = GetDC (NULL);
```

现在读者可以明白如何获得 HDC 了。下面讲述如何使用它。

提示：*许多初学者会忘记的事情是 Windows 并不监视窗口的重画。因此，如果在 WindowProc 函数的 WM_PAINT 部分以外进行绘图，一定要确保你画出来的东西会在屏幕需要更新的时候（例如，用户拖动一个其他窗口经过了窗口的上空，或用户把窗口最小化、或最大化）被重画。否则，很快地屏幕看起来就会一团糟了。*

2.1.2　各种绘图工具：画笔、画刷、颜色、线和形状

回想起 20 世纪 70 年代，笔者当时还是个儿童，有两年时间曾流行过一种真是非常稀罕的疯狂玩意儿。但只有极少数的一些玩意儿是真正奇怪的。而我记得的那个玩意儿就是其中之一。无论向何处去，无论朝何处看，都会看到墙上有用钉子和棉线制作的一些图案。图案有很多种，但其中最简单的一种是这样做的，准备一块长方形的木板，沿木板的左边和底边均匀地钉上钉子，然后用棉线把钉子相连。如果把钉子的排列想象成 Y 轴和 X 轴，在 X 轴的第一颗钉子和 Y 轴的最后一颗钉子间拉上棉线，又在 X 轴的第二颗钉子和 Y 轴的倒数第二颗钉子间拉上棉线，如此重复，最后会得到一幅如图 2.1 那样的曲线状图案。

读者也许在疑问：“这和 Windows GDI 有什么关系呢？”这里的第一个程序将模拟此类优美的艺术图案，如图 2.2 所示。

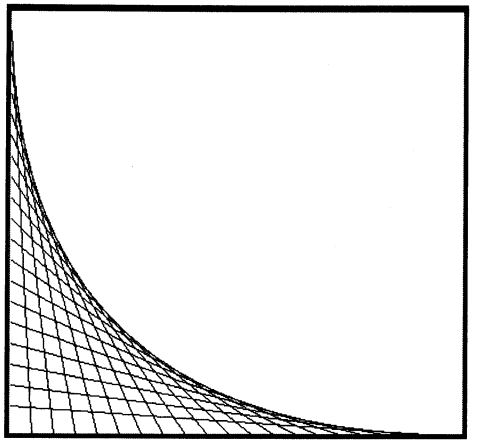


图 2.1　得到的曲线状图案

图 2.2　奇妙的线条

读者可在光盘的 GDI_Lines 文件夹下找到源代码。为了创建这些美妙的线条，只需要对第一章中你看到的窗口过程做一些修改即可。下面是修改过后的 WindowProc 的片断：

```
LRESULT CALLBACK WindowProc (HWND      hwnd,
                             UINT      msg,
                             WPARAM    wParam,
                             LPARAM    lParam )
{
    static int cxClient, cyClient;
```

这里定义了两个静态整型变量来保存窗口客户区的尺寸。客户区也就是窗口中读者将进行绘图的区域，但不包括标题栏、滚动条、边框。任何绘图所用到的运算都基于这两个值进行，这样，显示图形就能够在用户改变窗口尺寸的时候进行相应的缩放。

```
switch (msg)
{
  case WM_CREATE:
    {
     RECT rect;
     GetClientRect (hwnd, &rect);
     cxClient = rect.right;
     cyClient = rect.bottom;
    }
    break;
```

当第一次创建好窗口后，需要确定客户区的宽和高。为此，创建一个 RECT 结构，将其和一个窗口句柄一起传给 GetClientRect 函数。然后只需读取 RECT 结构中的信息，并用它们来设置 cxClient 和 cyClient。这样就可从 cxClient 和 cyClient 来知道“画布”究竟有多大了。

下面进行绘图。先请读者浏览一下 WM_PAINT 处理段，接着将逐个讲解相关的部分。

```
case WM_PAINT:
{
   PAINTSTRUCT ps;
   BeginPaint (hwnd, &ps);
   //准备从每边画出几根线
   const int NumLinesPerSide = 10;
   //根据窗口尺寸计算所画各线的步长（即间距）
   int yStep = cyClient/NumLinesPerSide;
   int xStep = cxClient/NumLinesPerSide;
   //下面是画线的代码
   for (int mult=1; mult<NumLinesPerSide; ++mult)
   {
    MoveToEx (ps.hdc, xStep*mult, 0, 0);
    LineTo (ps.hdc, 0, cyClient-yStep*mult);
    MoveToEx(ps.hdc, xStep*mult, cyClient, 0);
    LineTo(ps.hdc, cxClient, cyClient-yStep*mult);
    MoveToEx(ps.hdc, xStep*mult, 0, 0);
    LineTo(ps.hdc, cxClient, yStep*mult);
    MoveToEx(ps.hdc, xStep*mult, cyClient, 0);
    LineTo(ps.hdc, 0, yStep*mult);
   }
 EndPaint (hwnd, &ps);
}
```

可以看到，绘图代码还是相当易懂的。注意这里绘画所用到的点都是根据 cxClient 和 cyClient 计算出来的。这是适应窗口缩放所必须的。有关这一点，后面马上就会具体谈到。现在，解释一下 MoveToEx 函数和 LineTo 函数。

如果有一条从点 A 到点 B 的线段，MoveToEx 可以用来先将画笔所在位置移动到点 A，然后再利用 LineTo 以当前画笔绘制一条线段到 B 点。MoveToEx 的函数原型如下：

```
BOOL MoveToEx (
   HDC      hdc,                    // DC 句柄
   int      X,                      // 当前点位置的 X 坐标
   int      Y,                      // 当前点位置的 Y 坐标
   LPPOINT lpPoint                  // 指向前一个点的位置的指针
);
```

给这个函数传入一个设备描述表句柄和一对 X、Y 坐标，它就会将画笔移动到目标点，移动过程中不会画出线来。如果因为某种原因而不得不把画笔移回原始位置，可以从 lpPoint 取到开始绘图之前画笔的位置。这在某些情况下是有用的，但一般情况下很少用它，将它设为 NULL 即可。

一旦把画笔定位于线段的起点后，就可用 LineTo 来绘制线段：

```
BOOL LineTo (
   HDC hdc,              // DC 句柄
   int nXEnd,            // 线段终点的 X 坐标
```

```
    int nYEnd          // 线段终点的 Y 坐标
);
```

同样地，如果要传一个 HDC 给 LineTo——指明要在哪个设备上进行绘图，还需要传两个坐标值指明线段终点的位置。

下面解释 WindowProc 中的一段新内容。

```
case WM_SIZE:
{
  cxClient = LOWORD(lParam);
  cyClient = HIWORD(lParam);
}
break;
```

如果编译一个不带 WM_SIZE 消息处理功能的 GDI_Lines_1 例程，运行并试图改变窗口的大小，显示就会立即出错，因此为了允许用户改变窗口大小，必须跟踪客户区。

当用户改变窗口大小时，系统会发出了一条 WM_SIZE 消息。窗口的新尺寸存放在 32 位整数 lParam 的高、低字里面传给 WindowProc。图 2.3 是尺寸信息在 lParam 里存储的示意图[①]。HIWORD 和 LOWORD 是 Windows 用以取得 32 位整数的高、低字的宏。接着用新的尺寸数据更新 cxClient 和 cyClient，由于这两个值是所有绘图计算的基础，故新绘制的图形就能得到相应的缩放。

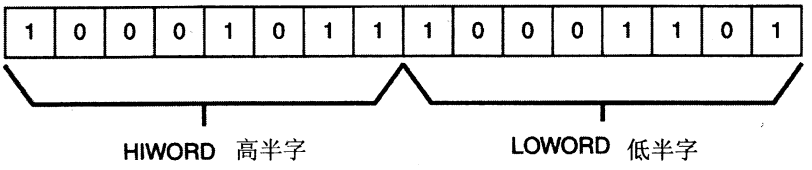


图 2.3　在 lParam 和 wParam 内部

技巧：有几种方法可以用来禁止用户改变窗口的大小，但最简单的方法就是在调用 CreateWindowEx 的时候不使用 WS_THICKFRAME 标志。如果在文档里查找 WS_OVERLAPPEDWINDOW，会发现使用它能节省一些时间，它是一些标志的组合，其中就包括 WS_THICKFRAME。因此，要达到和前面同样的结果，但除了 WS_THICKFRAME，应使用如下这些标志：

WS_OVERLAPPED | WS_VISIBLE | WS_CAPTION | WS_SYSMENU

试一试这个方法，看这样做以后在试图改变窗口大小的时候会发生什么。

2.1.2.1 创建自定义画笔

前面的例子用默认的黑色来画线，但如果希望用其他颜色和不同的线宽绘图应该怎么做？为了达到这些目的，必须用 CreatePen 函数创建你自定义的画笔。下面看一下这个函数：

[①] 译者注：图 2.3 中画出来的左右两部分只有 8 位，实际左右每边都应该有 16 位。

```
HPEN CreatePen (
  int      fnPenStyle,    // 笔的式样
  int      nWidth,        // 笔的宽度
  COLORREF crColor        // 笔的颜色
);
```

fnPenStyle 是定义画笔应如何绘线的标志。一些常用的样式如表 2.1 所示。

表 2.1　画笔绘图样式

样　式	描　述
PS_SOLID	画笔画出的是一根实线
PS_DASH	画笔画出的是虚线
PS_DOT	画笔画出的是点状线
PS_DASHDOT	画笔画一根点划线
PS_DASHDOTDOT	画一根在短划线后跟两个点的线

nWidth 设置画笔的宽度，单位是逻辑单元。可以认为这就是以像素数为单位的线宽。不过，如果把这个值设为零，画笔仍有其值=1 的线宽。

crColor 是定义画笔颜色的 COLORREF 结构。在显示器上看见的每种色彩都由红绿蓝三原色组成。利用不同强度（或亮度）的三元色组合，就可以创造出可用的整个颜色光谱。三原色在 COLORREF 中各占一个字节，这就意味可以为每一个亮度赋予从 0 到 255 范围之间的某个值，以 255 为最亮。

定义颜色时，通常按红、绿、蓝顺序设置颜色强度。例如，颜色(255,255,255)代表白色，(0,0,0)代表黑色，而(255,0,0)是红色等。Windows 提供了一个叫做 RGB 的宏来帮助创建 COLORREF 结构。例如要创建一个红色的 COLORREF，可写作 RGB(255,0,0)。

CreatePen 返回刚创建的画笔的句柄。下面尝试创建一个宽度为两个像素的红色实线画笔。

```
HPEN RedPen = CreatePen (PS_SOLID, 2, RGB (255,0,0));
```

RedPen 是一个自定义画笔的句柄。要使用这一支画笔，先要在设备描述表中选定它，以便设备知道用什么画笔进行绘图。SelectObject 函数用于选定对象。

```
HGDIOBJ SelectObject (
  HDC     hdc,              // 指向 DC 的句柄
  HGDIOBJ hgdiobj           // 指向 object 的句柄
);
```

注意：任一时刻，设备描述表中只能选定一支画笔。为了修改画笔，需要再次使用 SelectObject，为 DC 选定另一支画笔。

这是一个多用途的函数，还可以用它为设备描述表选定除画笔以外的对象，例如位图、画刷、字体和区域等。这也是为什么传递的参数是 HGDIOBJ（GDI 对象的句柄）而不是 HPEN。SelectObject 返回函数调用时 DC 里选中的对象的句柄，这样就能够保留绘图开始前的 DC 的一个备份，以便绘图结束时来恢复它。由于函数使用了 HGDIOBJ 作为参数，

用户必须作合适的强制类型转换——在本例子中就是 HPEN。下面这一行就是为设备描述表选择使用上述红笔的代码：

```
HPEN OldPen = (HPEN)SelectObject (hdc, RedPen);
```

当绘图完毕时，应把先前使用的画笔选回到 DC，也就是恢复现场。

```
SelectObject (hdc, OldPen);
```

当程序结束运行时，应当确保任何创建过的画笔（或其他对象，诸如画刷和位图）都被删除。删除画笔可以使用 DeleteObject。

```
BOOL DeleteObject(
  GDIOBJ hObject      // 指向绘图对象的句柄
 );
```

如果忘记删除 GDI 对象，就会造成资源泄漏。此外，记住不要删除当前正被 DC 选用的 GDI 对象。

以上即为创建画笔的内容。为了使用自定义的画笔，笔者已修改了 GDI_Lines1 的代码。笔者自己用的颜色可能看起来不舒服。接下来请看 GDI_Lines2 的源代码，并用读者自己创建的画笔样式来编程。

技巧：可以在屏幕上用 SetPixel 函数绘点。原型如下：

```
COLORREF SetPixel (
  HDC     hdc,          // DC 句柄
  int       X,          // 像素的 X-坐标
  int       Y,          // 像素的 Y-坐标
  COLORREF crColor      // 像素的颜色
);
```

也可以从屏幕上读取任意坐标的像素的颜色：

```
COLORREF GetPixel
  HDC hdc,              // DC 句柄
  int nXPos,            // 像素的 X-坐标
  int nYPos             // 像素的 Y-坐标
);
```

2.1.2.2 画刷

画刷是用来填充或绘制 shape 的。比如说要画一个矩形，同时希望这个矩形包围的区域填满橙色，就可以先创建一个橙色画刷，然后用它绘制矩形。

创建画刷和创建画笔很相似：先创建画刷，然后将其选入设备描述表，使用完毕后，就删除它。Windows 也有许多备用的画刷，不过它们不是全黑、全白就是灰度的（只有 NULL_BRUSH 例外，它是不可见的）。默认画刷是白色。

和创建画笔不同，创建画刷有数种方法。下面简要介绍一下这些方法。

1．创建实心型画刷

这个函数很简单。它创建了一个以 solid 方式填充一区域的画刷。

```
HBRUSH CreateSolidBrush (
  COLORREF crColor      // 画刷颜色值
);
```

只需要为函数传入一个 COLORREF 值，它便会返回自定义画刷的句柄。

2．创建阴影线型画刷

创建的画刷用由 fnStyle 定义的阴影线模版绘图。可用的阴影线模版样式，如图 2.4 所示。

```
HBRUSH CreateHatchBrush(
   int        fnStyle,          // 阴影线模版的样式
   COLORREF clrref              // 颜色值
);
```

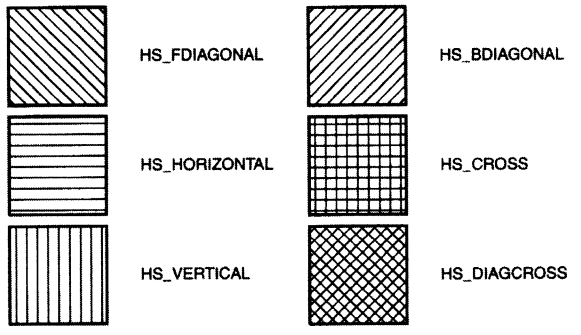


图 2.4　画刷图案样式标志

3．创建图案型画刷

也可以定义用位图来代替画刷。只需要一个位图句柄，则画刷会用该位图填充指定区域内，函数原型如下。

```
HBRUSH CreatePatternBrush (
ITMAP hbmp     // 位图句柄
);
```

此外还有 CreateBrushIndirect 和 CreateDIBPatternBrushPt 两种画刷创建法，在此不再一一详细解释，它们被用到的频率很低。

接下来介绍怎样利用 GDI 来绘制各种各样的 shape。

2.1.2.3　形状

可以通过 GDI 来绘制许多类型的 shape 图形。shape 图形通常都有一条用当前选中的画笔绘制的边界，而在 shape 内部的封闭区域则用当前画刷来填充。图 2.5 列出了一些 shape 类型。

用户需要给这个函数传入一个设备描述表句柄和矩形的左上角、右下角的坐标，如图 2.6 所示。

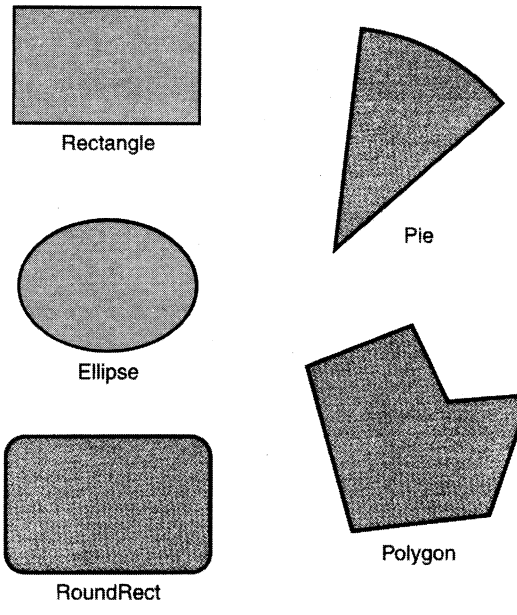


图 2.5　典型 GDI shapes

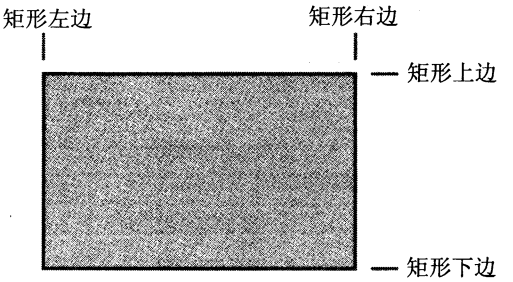


图 2.6　定义矩形的坐标

1．矩形

Retangle 是最简单的 shape 绘制函数。其函数原型如下：

```
BOOL Rectangle(
    HDC hdc,                // DC 句柄
    int nLeftRect,          // 矩形左上角的 X-坐标
    int nTopRect,           // 矩形左上角的 Y-坐标
    int nRightRect,         // 矩形右下角的 X-坐标
    int nBottomRect         // 矩形右下角的 Y-坐标
);
```

此外还有两个有用的矩形函数：FrameRect 和 FillRect。

FrameRect 用指定画刷绘出指定矩形边框。

```
int FrameRect (
  HDC         hDC,          // DC 句柄
  CONST RECT *lprc,         // 指向矩形坐标的指针
  HBRUSH      hbr           // 画刷句柄
);
```

而 FillRect 则用指定画刷绘出矩形区域。

```
int FillRect(
  HOC         hDC,          // DC 句柄
  CONST RECT *lprc,         // 指向矩形坐标的指针
  HBRUSH      hbr           // 画刷句柄
);
```

2．椭圆形

Ellipse 函数和 Rectangle 函数几乎相同。

```
BOOL Ellipse (
  HDC hdc,              // DC 句柄
  int nLeftRect,        // 椭圆包围矩形左上角的 X-坐标
  int nTopRect,         // 椭圆包围矩形左上角的 Y-坐标
  int nRightRect,       // 椭圆包围矩形右下角的 X-坐标
  int nBottomRect       // 椭圆包围矩形右下角的 Y-坐标
);
```

要画一个椭圆，就要利用恰好包围该椭圆的那个矩形的坐标，如图 2.7 所示。

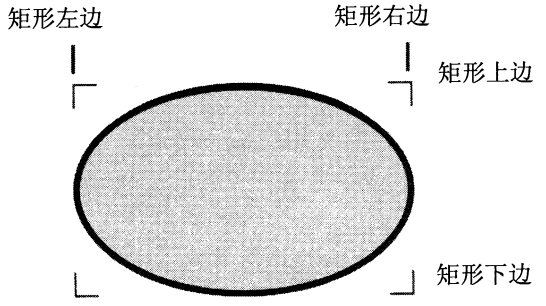


图 2.7　椭圆的形状

因此，如果要画一个（正）圆，只需要向该函数传入一个正方形的左上和右下角坐标即可。

3．多边形

一个 Polygon，定义为由线段相互连接起来的一系列点，它们围出了一个区域。最后一点总和第一点相连，如图 2.8 所示。

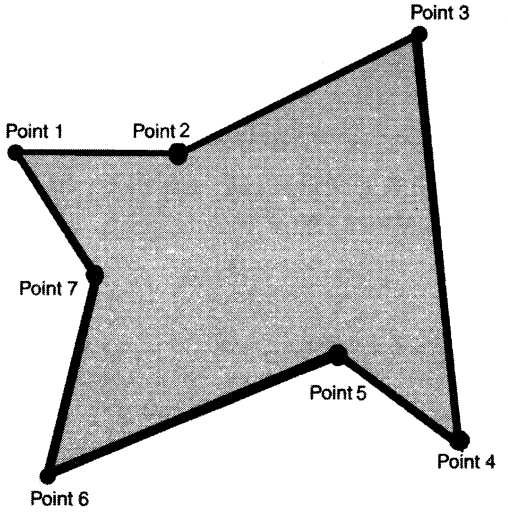


图 2.8　多边形 shape（形状）

形成多边形的点通常称作顶点。Polygon 函数原型如下：

```
BOOL Polygon (
  HDC          hdc,                    // DC 句柄
  CONST POINT *lpPoints,               // 指向多边形顶点坐标的指针
  int          nCount                  // 多边形顶点数目
);
```

这个函数的参数之一是个指向一张 POINT 结构表的指针。该表定义了每个顶点的坐标，也定义了多边形包含的顶点的总数。POINT 结构定义了空间中的一个点，如下所示：

```
typedef struct tagPOINT
{
   LONG x;
   LONG y;
)POINT;
```

笔者在这里写了一个简单程序，每次用户按空格键时都会在屏幕上新画一个多边形。可以在光盘上的 Chapter2/GDI_Polygon 目录下找到此源代码。下面列出 WindowProc 中一些相关的部分。

```
LRESULT CALLBACK WindowProc (HWND   hwnd,
                             UINT   msg,
                             wParam, lParam)
{
    //创建一些画笔供画图用
    static HPEN BluePen  = CreatePen (PS_SOLID, 1, RGB (0, 0, 255));
    static HPEN OldPen   = NULL;
    //创建一个实心型画刷
    static HBRUSH RedBrush = CreateSolidBrush (RGB (255, 0, 0));
    static HBRUSH OldBrush = NULL;
    //用来存放窗口客户区的大小数据
    static int cxClient, cyClient;
    //用来存放建立的多边形顶点
    static POINT verts [NUM_VERTS];
    static int iNumVerts = NUM_VERTS;
```

这里定义了一些绘图中要用到的变量。这里还创建了一支画笔用来画图形的轮廓和一把用来填充内部的画刷。NUM_VERTS 的值是在 defines.h 头文件中定义的。

```
switch (msg)
  {
    //当应用程序窗口首次建立时,有一个 WM_CREATE 消息送出
    case WM_CREATE:
    {
    //为了得到窗口客户区的 size,先要创建一个 RECT,然后要求
    //Windows 在 RECT 结构中填入窗口客户区的宽和高
    //然后再将它们分别赋给 cxClient 和 cyClient
    RECT rect;
```

```
    GetClientRect (hwnd, &rect);
    cxClient = rect.right;
    cyClient = rect.bottom;
    //激活一个随机数发生器
    srand((unsigned) time(NULL));
    //随机地创建一些顶点
    for (int v=0; v<iNumVerts; ++v)
    {
      verts [v].x = RandInt(0, cxClient);
      verts[v].y = RandInt(0, cyClient);
    }
  }
  break;
```

当程序收到 WM_CREATE 消息，随机数发生器就被随机化，而 verts 用随机坐标填充。每个坐标都代表多边形的一个顶点。RandInt 是 utils.h 头文件中定义的诸多随机数函数之一，它返回一个落于指定的两个数之间的随机数。

```
case WM_KEYUP:
  {
    switch(wParam)
    {
     case VK_SPACE:
      {
       //为多边形随机地创建一些新的顶点
        for (int v=0; v<iNumVerts; ++v)
        {
          verts [v].x = RandInt(0, cxClient);
          verts[v].y = RandInt(0, cyClient);
        }
        //进行刷新,可以看到我们的新 polygon
        InvalidateRect (hwnd, NULL, TRUE);
        UpdateWindow(hwnd);
      }
      break;
    }
  }
```

如果空格键按下又松开，就会产生一个 WM_KEYUP 消息。程序检测到消息，即为多边形产生一组新的随机顶点坐标。注意这里调用了 InvalidateRect 和 UpdateWindow。调用 InvalidateRect 时，若第二个参数设置为 NULL，能使 Windows 知道应该将整个客户区加入更新区域内。更新区域代表窗口客户区中必须在下次执行 WM_PAINT 时被重画的部分。而 UpdateWindow 则只是在需要更新区域时发出一个 WM_PAINT 消息（因此在此前先要调用 InvalidateRect）。它将 WM_PAINT 消息直接发给 WindowProc，而不经过消息队列。这就保证了窗口立刻得到更新。这短短两行代码的效果是客户区得到更新，因而就能看见新创建的多边形。

```
case WM_PAINT:
{
 PAINTSTRUCT ps;
 BeginPaint (hwnd, &ps);
  //首先选择一支用来作图的画笔,并将与它交换的旧画笔存放到一个备份中
  OldPen = (HPEN)SelectObject (ps.hdc, BluePen);
  //对画刷也作同样的工作
  OldBrush = (HBRUSH)SelectObject (ps.hdc, RedBrush);
  //画此多边形
  Polygon (ps.hdc, verts, iNumVerts);
  //代替成原先的旧画笔
  SelectObject (ps.hdc, OldPen);
  //和旧画刷
  SelectObject (ps.hdc, OldBrush);
  EndPaint (hwnd, &ps);
  }
  break;
```

WM_PAINT 段很容易理解。首先向 DC 内输入自定义的画笔和画刷。然后利用 vert 中的顶点调用 Polygon 函数来画图。用户可以试着修改一下上面的代码，创建一些自己的画刷和 shape，然后再继续看下一节。

2.2 文　　本

到目前为止已经学习了如何在画布上绘图，但如何在上面签名呢？下面就要解决这个问题。

Windows 有非常多的函数用来显示和操纵文本与字体。这里只是介绍基本点。

2.2.1 TextOut

朝屏幕输出文本最简单的方法就是使用 TextOut 函数。下面看一下它的原型：

```
BOOL TextOut (
  HDC       hdc,              // DC 句柄
  int       nXStart,          // 开始写入位置的 X-坐标
  int       nYStart,          // 开始写入位置的 Y-坐标
  LPCTSTR   lpString,         // 指向字符串的指针
  int       cbString          // 字符串的字符数
);
```

其中参数不难理解。首先向 TextOut 传入 DC 句柄，还有需要文本显示的位置坐标，一个字符串指针，最后是一个表示文本长度的整数。文本默认的颜色是黑色，默认的背景色是 WHITE_BRUSH。这里先不解释怎么改变默认的颜色和背景，先要介绍另一种文本显示方法。

2.2.2　DrawText

DrawText 比 TextOut 稍微复杂一点。它的原型如下：

```
int DrawText (
  HDC     hDC,          // Dc 句柄
  LPCTSTR lpString,     // 指向代写入字符串的指针
  Int     nCount,       // 字符串的长度（字符个数）
  LPRECT  lpRect,       // 指向格式化大小结构的指针
  UINT    uFormat       // 文本书写标识
);
```

这个函数在 lpRect 指定的文本框里显示文本。文本根据 uFormat 标志的设置，被格式化于该框内。Draw Text 格式化标志有许多，表 2.2 中列出了一些。

表 2.2　DrawText 格式化标志

标　志	描　述
DT_B0TT0M	文本在文本框内底部对齐。若要使用本标志，必须同时使用 DT_SINGLELINE 标志
DT_CENTER	这个标志使文本在文本框内水平居中
DT_LEFT	文本左对齐
DT_RIGHT	文本右对齐
DT_SINGLELINE	在单行里显示所有文字。按回车键文字也不换行
DT_T0P	文本顶部对齐
DT_WORDBREAK	这个标志的效果类似于字（word）的换行

2.2.3　加入颜色和透明度

在 Windows 中可以自己定义背景色和前景色，还可以设置文本的透明度。为了设置文本颜色，需要利用 SetTextColor 函数：

```
COLORREF SetTextColor (
  HOC       hdc,        // DC 句柄
  COLORREF  crColor     // 文本颜色
);
```

设置背景色用 SetBkColor 函数：

```
COLORREF SetBkColor(
  HDC       hdc,        // DC 句柄
  COLORREF  crColor     // 背景颜色
);
```

设置完成以后，前景和背景色会一直有效，直到再次修改为止。两个函数都返回当前的颜色值，用户可以记录最初的设置，以便在必要时恢复。

举例来说，为了将文本颜色设为红色而背景设为黑色，可以执行下面这些语句。

```
//设置 text 为红色
SetTextColor (ps.hdc, RGB (255, 0, 0));
//设置 background 为黑色
SetBkColor(ps.hdc, RGB (0, 0, 0));
```

除了能够设置前景和背景色以外，还可以设置透明度。这把显示到屏幕上的文本的背景部分变成了透明（例如，如果文本背后是一个图案，则看起来就像是把文字直接打印到图案上一样）。

可以用 SetBkMode 函数设置背景的透明度：

```
int SetBkMode(
  HDC hdc,          // DC 句柄
  int iBkMode       // 用以指明背景模式的 flag
);
```

只有两个标志可用：（不透明）OPAQUE 和（透明）TRANSPARENT。因此，要使用透明背景绘制文本的时候，在具体绘制之前，应设置合适的背景模式。

GDI_Text 的源代码示范了所有这些函数的使用。图 2.9 是它的一个屏幕图像。

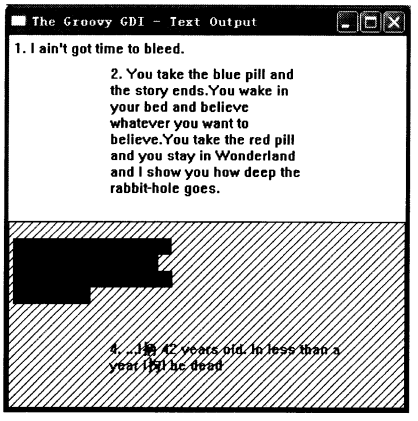


图 2.9　输出文本的各种方法

2.2.4　实时消息抽取循环

在 GDI_Polygon 例子里，每次需要显示一个新的多边形，就得按一次空格键，这可能显得有点繁琐。但怎样才能在屏幕上连续不断地显示新的多边形来催眠用户呢？仅用 GetMessage 是办不到的。如果消息队列中没有消息，GetMessage 就会等待，直到新的消息出现在队列中。因为许多游戏都是充满动作包装的，需要猛击键盘，产生场景颤动变化，利用简单的 GetMessage 消息循环通常不能符合要求。游戏编程者不希望在玩家没有新的动作时游戏就静止不动——他们希望你的怪物从后面冲出来、围攻玩家、追捕玩家。为了做到这样的效果，需要这样一种消息循环，只有在需要处理消息时才去处理消息，其余的时间都让游戏代码去自动产生激动的场面。为了做到这一点，可以使用 PeekMessage 函数。

其原型如下：

```
BOOL PeekMessage (
  LPMSG  lpMsg,              // message 结构指针
  HWND   hWnd,               // window 句柄
  UINT   wMsgFilterMin,      // 第一个 message
  HINT   wMsgFilterMax,      // 最后的 message
  UINT   wRemoveMsg          // 移去标志 (removal flags)
);
```

PeekMessage 和 GetMessage 函数很相似。惟一的差异是最后一个参数 wRemoveMsg。它的值可以是 PM_NOREMOVE——表示消息在处理后仍然保留在队列中，或 PM_REMOVE——表示消息在处理后从队列中移去，通常选择使用后者。如果队列中有消息，PeekMessage 将返回 true，若没有则返回 false。

要创建实时的消息抽取循环，就需要更复杂一些，如果只把 GetMessage 替换作 PeekMessage，那么，当队列中没有消息时，PeekMessage 返回零，应用程序会因此而终止。下面尝试用另一种形式来修改绘制多边形的例子：

```
while (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

这样只会见到应用程序窗口闪电般地打开，接着就立即关闭。而实际中需要的是一个更为强健可靠的设计，过程如下：

```
// 进入 message 抽取循环
bool bDone = false;
MSG msg;
while(! bDone)
{
  while( PeekMessage ( &msg, NULL, 0, 0, PM_REMOVE ) )
  {
    if( msg.message == WM_QUIT )
    {
      // 如果是 quit 消息,就退出循环
      bDone = true;
    }
    else
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
  }
  //下面是调用 WM_PAINT 绘制场景
  InvalidateRect (hWnd, NULL, TRUE);
```

```
        UpdateWindow(hWnd);
    }//结束 while 语句
```

这一消息抽取过程在 while(!bDone)里不断循环，直到 bDone 成为 true。每次执行循环体时，PeekMessage 函数检查消息队列中是否有待处理的消息。如果有消息，则检查它是不是 WM_QUIT 消息。如果是，则 bDone 被设置成 true 而应用程序随即退出。如果不是 WM_QUIT 消息，则像先前的例子那样处理和分发这条消息，并且从消息队列中删去这一个消息。如果没有待处理的消息，则循环体会利用 InvalidateRect 和 UpdateWindow 产生 WM_PAINT 消息以重画窗口。在用户编写的游戏程序时，这里也是供用户放置游戏主循环的场所。

从 GDI_Polygons2 例子里的消息循环的运行情况中，可以发现 WindowProc 中的 WM_PAINT 部分现在包含产生新多边形的代码。这里利用了小巧的 Sleep 函数让程序每帧有几毫秒的延迟，这样就能实际看清每个多边形。调用 Sleep 时，参数表示程序应当暂停的毫秒数，暂停过后，程序接着就自动继续运行。

2.3 如何创建后备缓冲

在编写游戏或其他任何需要在一秒钟内刷新许多次屏幕显示的程序时，经常会遇到屏幕闪烁问题。我写了一个小演示程序来告诉你我的意思是什么。在光盘的 GDI_Backbuffer 目录下，读者能找到我写的小演示程序。观察一下这个程序的执行，会看到一些球在屏幕内部弹来弹去。图 2.10 是此程序演示时的一幅快照。

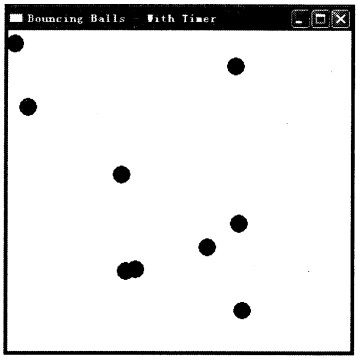


图 2.10 球

屏幕闪烁的效果不好，这个问题与显示器的工作原理有关。

在显示器的屏幕内表面覆盖着 3 种不同的荧光物质，它们在电子枪的轰击下分别放出红、绿、蓝光。由于三种颜色的各种不同的相对强度，形成了在屏幕上实际看见的各种色彩。这正是为什么先前看到的 COLORREFS 结构中定义着红、绿和蓝（RGB）的缘故。显示器后部是电子枪，它是发射高速电子束的装置。电子束在电磁场的作用下进行瞄准。在绘制图像时，电子枪从显示屏的左上角开始扫描，水平地移动到最右端，使电子束依次几击中该行荧光物质的每一个像素。当到达一行的末尾时，它移回最左边并下移一个像素，

重新开始下一行的扫描。如此这样直到屏幕的右下角，然后又回到左上角，重新开始这整个过程。图 2.11 是电子枪扫描顺序的示意图。电子枪接连两次扫描到左上角的时间间隔，称为垂直刷新周期（vertical refresh period），每秒钟该过程的重复次数称为垂直刷新率（vertical refresh rate），或帧刷新率（frame refresh rate），简称帧率（framerate）。在不会混淆时，也可以简称刷新率（refresh rate）。

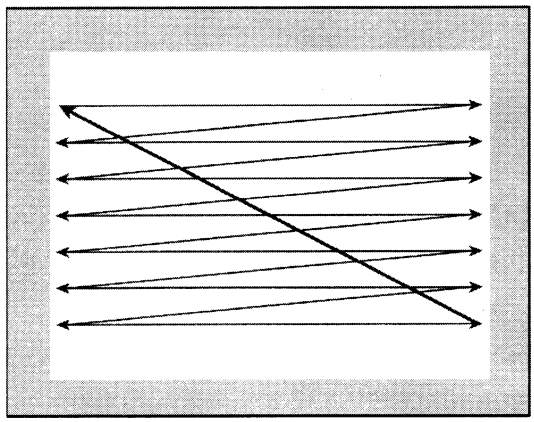


图 2.11　屏幕刷新过程

画面刷新的过程造成了闪烁——基本上，当程序向屏幕上绘图的时候电子枪仍在移动。因此造成了显示的闪烁、摇曳和撕裂。使用后备缓冲（back buffer）可以防止这些问题。

前端缓冲（Front Buffer）是一块能直接映射到显示器的内存区域。当在其上绘制任何东西时，它们都会立即显示出来。目前涉及的所有例子里，都还只是向前端缓冲中绘图。在 WindowProc 的 WM_PAINT 部分通过 BeginPaint 得到的 HDC，就是前端缓冲的 HDC。

为了防止闪烁，需要在内存中另外再创建一个区域，其格式和大小都与前端缓冲器完全相同，并在其上进行所有的绘画。这一块区域就叫做后备缓冲区（缓冲器）。由于是在一个离屏的（off-screen）缓冲区上进行绘画，因而在画的时候根本不会有任何图形显示出来。用户必须做的（在每一帧中），只是将后备缓冲区中画好的内容复制到前台缓冲区（这种操作通常称为 blitting，即显存位块传送）。由于这种传送速度极快，几乎是立刻发生的，屏幕的刷新不会由于电子枪的移动而变得混乱。这样就实现了不闪烁的显示。这一种技术一般称为双缓冲（Double Buffering）技术，有时也称为页面切换（Page Flipping）技术。

技巧：有时，游戏程序员为了使显示更加平滑，需用到第三个缓冲区。这称为三缓冲（Triple Buffering）。今日图形加速卡的出现，甚至使创建更多层的缓冲链成为可能。

2.3.1　如何实现双缓冲

如果要在内存中创建一块区域来表示前置缓冲（显示区域），首先需要的，是创建一个与显示 DC 相兼容的内存设备描述表。这个过程分 3 步。

首先，用 CreateCompatiableDC 函数创建一个内存设备描述表：

```
HDC hdcBackBuffer = CreateCompatibleDC (NULL);
```

用 NULL 作为参数时，Windows 默认创建一个与当前屏幕兼容的 DC，而这正是编程者所希望实现的。但当内存设备描述表被创建出来时，它是单色的，并且宽度和高度各为一个像素，这并不实用。因此，在开始用它绘图之前，需要创建一张大小和格式与前置缓冲完全一致的位图，并利用 SelectObject 函数将其选入内存 DC。这是第二步。

可以用 CreateCompatiableBitmap 函数来创建位图。它的函数原型如下：

```
HBITMAP CreateCompatibleBitmap (
  HDC hdc,                  // DC 句柄
  int nWidth,               // bitmap 宽度,用 pixels 来计算
  int nHeight               // bitmap 高度,用 pixels 来计算
);
```

当把显示设备的 HDC 以及它的高度和宽度作为参数传送到该函数后，它就会返回在内存中创建的一个位图的句柄。因此先要用本章先前提过的 GetDC 函数取得设备描述表的句柄：

```
HDC hdc = GetDC (hwnd);
```

然后创建相容的位图：

```
HBITMAP hBitmap = CreateCompatibleBitmap (hdc,
                                          cxClient,
                                          cyClient);
```

最后将此位图选入第一步得到的设备描述表中。当这一步也完成后，hdcBackBuffer 就完成了设置，与前置缓冲的 DC 完全一样，这样就可以开始朝它绘画了。本步工作可以通过 SelectObject 函数完成，如下：

```
HBITMAP hOldBitmap = (HBITMAP)SelectObject (hdcBackBuffer, hBitmap);
```

已经存在的 1×1 像素的单色位图的一个备份被保留，因此可以在绘图完毕时将它选回来，清理后备缓冲（之前选择画笔和画刷时也是这样处理的）。

新的弹球例程中，所有这些步骤都在 WM_CREATE 中完成。WindowProc 中相关的代码列出如下（为了简明起见，略去了初始化球的代码）：

```
LRESULT CALLBACK WindowProc (HWND      hwnd,
                             UINT      msg,
                             WPARAM    wParam,
                             LPARAM    lParam)
{
  //保存窗口客户区的大小尺寸
  static int cxClient, cyClient;
  //创建 back buffer
  static HDC        hdcBackBuffer;
  static HBITMAP    hBitmap;
  static HBITMAP    hOldBitmap;
  switch (msg)
  {
```

```
case WM_CREATE:
 {
   //获取客户区尺寸
   RECT rect;
   GetClientRect (hwnd, &rect);
   cxClient = rect.right;
   cyClient = rect.bottom;
   //创建后备缓冲区
   //先创建一个内存 DC
   hdcBackBuffer = CreateCompatibleDC (NULL);
   //获取前置缓冲区的 DC
   HDC hdc = GetDC (hwnd);
   hBitmap = CreateCompatibleBitmap (hdc,
                                     cxClient,
                                     cyClient);
   //将 bitmap 装入内存 DC
   hOldBitmap = (HBITMAP)SelectObject (hdcBackBuffer, hBitmap);
   //释放 DC
   ReleaseDC (hwnd, hdc);
 }
 break;
```

2.3.2　如何使用后备缓冲器

创建了后备缓冲后，使用它只需要按下面的步骤操作：

1. 清除后备缓冲——通常采用办法就是用背景色填充。
2. 在后备缓冲区的 hdc 中绘图、写字等。
3. 将后备缓冲中的内容复制到前台缓冲区。

下面一步一步来讲述。

要在后备缓冲区中填充实心的（solid）颜色（一般是背景色），可以使用 BitBlt 函数。这个函数通常是用来将后备缓冲区中的所有 bit 复制到另一个区域——显示内存区。参考图 2.12 中的例子。用户也能用该函数将一块内存填为某个颜色。

BitBlt 函数定义如下：

```
BOOL BitBlt (
  HDC  hdcDest,   // 目标 DC 句柄
  int  nXDest,    // 目标矩形区域左上角的 X-坐标
  int  nYDest,    // 目标矩形区域左上角的 Y-坐标
  int  nWidth,    // 目标矩形区域的宽度
  int  nHeight,   // 目标矩形区域的高度
  HDC  hdcSrc,    // 源 DC 句柄
  int  nXSrc,     // 源矩形区域左上角的 X-坐标
  int  nYSrc,     // 源矩形区域左上角的 Y-坐标
  DWORD dwRop     // 扫描操作码 (raster operation code)
);
```

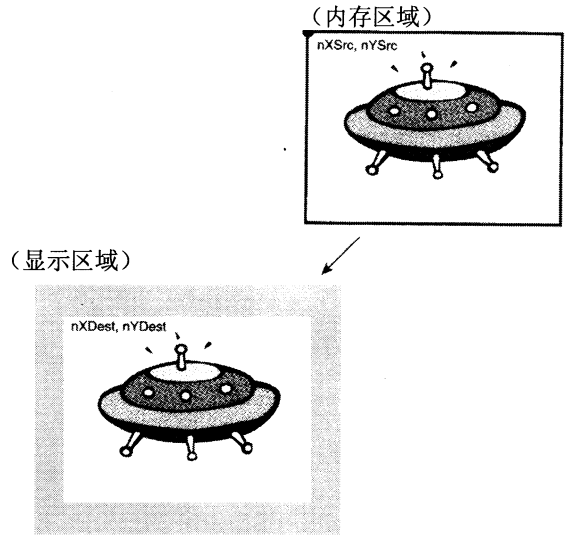


图 2.12　BitBlt 的执行结果

参数包括源区域 DC 和目标区域 DC，还有它们各自的左上角坐标和待复制区域的宽和高。最后一个参数用来指示如何将源区域中的颜色数据和目标区原有数据进行合并的标志。有大量可用的参数，利用它们可以获得各种各样的奇妙效果。但现在只考虑两个标志，即 SRCCOPY——从源区逐位地复制数据到目的区，和 WHITENESS——用白色 RGB (255, 255, 255)填充目标区域。

这里是 WM_PAINT 部分代码。为了简明起见，省略了更新和画球的代码。

```
case WM_PAINT:
  {
    PAINTSTRUCT ps;
    BeginPaint (hwnd, &ps);
    //用白色填充 backbuffer
    BitBlt (hdcBackBuffer,
            0,
            0,
            cxClient,
            cyClient,
            NULL,
            NULL,
            NULL,
            WHITENESS);
```

为了将后备缓冲区填为白色，除了向 BitBlt 传送客户区大小外，关于源区的其余参数均设为 NULL，同时用 WHITENESS 标志将所有像素设为 RGB (255,255,255)。

```
    //------------------------------------------------------------
    // 在这里执行所有的画图操作。
    //------------------------------------------------------------
```

记住，现在用的是 WM_PAINT 里创建的后备缓冲区的 hdc: hdcBackBuffer。

```
        //现在把 back buffer 内容用位块传送方法传到 front buffer
        BitBlt (ps.hdc,
                0,
                0,
                cxClient,
                cyClient,
                hdcBackBuffer,
                0,
                0,
                SRCCOPY);
        EndPaint (hwnd, &ps);
    }
```

通过这些代码，后备缓冲器中的内容被复制到了显示窗口 DC 中。这里再一次使用了 BitBlt，但这次设置了内存 DC 的相关参数，并将 dwRop 设为 SRCCOPY，这告诉函数要复制内存 DC 中的内容。利用这样的方法，用户在后备缓冲区中画出的所有内容都会在屏幕上显示出来。

这就是如何创建后备缓冲区并进行渲染的办法。不过，如果只是把上面的代码插入程序中，可能还会见到闪烁。这是因为创建窗口类时，利用到了 WNDCLASSEX 结构的填充，而后者的背景色设置成白色。因而，即使用后备缓冲进行绘图，每次调用 BeginPaint 时，API 还是会把窗口填充为白色背景，这就造成了一定的闪烁。为了解决这个问题，用户可以把 WNDCLASSEX 结构中的相应成员变量设为 NULL。

```
    winclass.hbrBackground = NULL;
```

技巧：BitBlt 在许多地方有用，但特别常用于 2D 游戏中的复制精灵（Sprite）和在 3D 游戏中复制顶级信息显示（heads-up display，HUD）或状态统计信息（Stats）等。Sprite 是个 2D 位图或一系列产生动画效果的位图，可以载入内存并复制到显示区域中的指定坐标。还记得音速刺猬索尼克（Sonic the Hedgehog）吗？它就是一个 sprite。

2.3.3　保持干净

创建了一个位图和内存 DC 后，当游戏结束时一定要删除它们，否则会造成资源泄漏。首先把 hOldBitmap 选回内存 DC，这就释放了要删除的位图。接着便可以安全地删除 DC 和位图。下面是执行此功能的 WM_DESTROY 消息处理段代码：

```
    SelectObject (hdcBackBuffer, hOldBitmap);
    DeleteDC(hdcBackBuffer);
    DeleteObject(hBitmap);
```

最后必须确保后备缓冲区也随用户改变窗口尺寸大小而改变。为此，用户必须删除已有的兼容位图，并创建一个新的具有恰当大小的位图，它是在 WM_SIZE 中完成的，代码为：

```
case WM_SIZE:
{
    //如果用户改变窗口尺寸,则应更新变量,使得所有利用
    //cxClient 和 cyClient 所作的图形都相应地改变比例
    cxClient = LOWORD(lParam);
    cyClient = HIWORD(lParam);
    //将 backbuffer 相应地改变比例
    //把 old bitmap 选回到 DC
    SelectObject (hdcBackBuffer, hOldBitmap);
    //不要忘了下面这一条,否则将会产生资源泄漏
    DeleteObject(hBitmap);
    //为应用程序获得 DC
    HDC hdc = GetDC (hwnd);
    //创建和应用程序有相同大小相同形状的另一个 bitmap
    hBitmap = CreateCompatibleBitmap (hdc,
                                      cxClient,
                                      cyClient);
    ReleaseDC (hwnd, hdc);
    //把新的 bitmap 选送进 DC
    SelectObject (hdcBackBuffer, hBitmap);
  }
  break;
```

2.4 使 用 资 源

资源（Resource）包含了游戏中可能使用的任何一种数据，它将和编译产生的代码结合在一起形成一个可执行文件。资源包括位图、声音文件、图标、光标，以及程序需要的任何其他东西，如图 2.13 所示。之所以将数据文件包含到资源中，是因为这样可以使一切都干净整洁，不再需要许多独立的图像和声音文件，也可防止他人轻易盗用艺术作品或辛辛苦苦创建出来的其他文件。

虽然可以创建自定义的资源，但通常 Windows 预定义的资源类型已经足够用了。最常用的资源类型有下列几种：

- 图标（**Icons**）：在应用程序窗口左上角可以看见的小图标，或在 Windows 资源管理器中看到的和用 Alt+Tab 组合键切换任务时显示的图标。
- 光标（**Cursors**）：可以使用任何默认光标。你也可以创建自定义的光标来用。
- 字符串（**Strings**）：把字符串列为资源之一似乎有些奇怪，但事实上，将所有的字符串集中存储是个很好的方法。因为当把它们存在一起，就很容易为游戏制作多个语言的版本，进行小规模的文字改动也变得容易，不必在繁多代码中查找了。
- 菜单（**Menus**）：极大多数情况编程者的游戏用户自己定制的用户界面，但也经常为编写一些工具程序而使用此类常规菜单。
- 位图（**Bitmaps**）：这些是包含像素阵列的图像文件。Windows 支持 BMP 文件后缀的位图。

- 声音文件（**Sound files**）：用户可以把自己的所有的声音文件（wav 文件）作为资源使用。
- 对话框（**Dialog boxes**）：既可以用预定义的对话框，如 MessageBox，也可以创建自己定义的对话框并将其储存为资源。

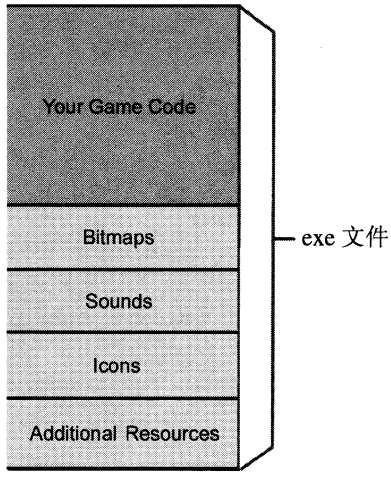


图 2.13　资源的组成

下面讲述如何创建资源。

2.4.1　图标

用户可以创建两种尺寸类型的图标：一种是 32×32 像素的大图标，一种是 16×16 像素的小图标。大的那种是在按下 Alt+Tab 组合键切换程序时看到的，而小的那种能够在 Explorer 中和在程序窗口的左上角见到。

用户可以用自己喜欢的绘图程序创建图标，并将它保存为.ico 后缀的文件，如果使用了 Microsoft Developer Studio，就可以用其内建的图标编辑器来创建图标。后者的使用方法是，在菜单里选择 Insert、Resource、Icon、New，然后一个简单的图标编辑器窗口会出现，如图 2.14 所示。

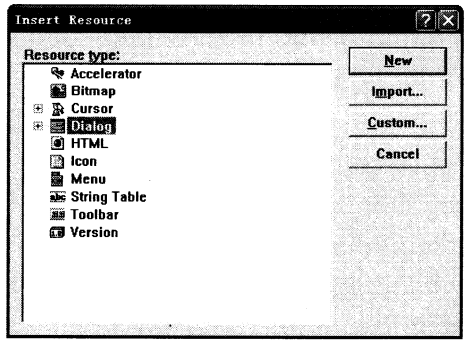


图 2.14　插入资源选项

现在开始创建图标（一大一小），并为它们起个有意义的名字，如 IDI_ICON_SM 和 IDI_ICON_LRG，将它们保存为资源：一个具有.rc 后缀的文件。你会看到 Developer Studio 自动生成了 resource.h 文件，你应当在你的主程序中用语句来包含它。

如果要显示图标，应当在注册窗口类的时候引用图标。

```
//首先填写 window class 结构
winclass.cbSize        = sizeof(WNDCLASSEX);
winclass.style         = CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra  = 0;
winclass.cbWndExtra  = 0;
winclass.hInstance    = hInstance;
winclass.hIcon    =LoadIcon(hInstance,MAKEINTRESOURCE (IDI_ICON_LRG));
winclass.hCursor = LoadCursor(NULL,IDC_ARROW);
winclass.hbrBackground    = (HBRUSH)GetStockObject(WHITE_BRUSH);
winclass.lpszMenuName     = NULL;
winclass.lpszClassName    = g_szWindowClassName;
winclass.hIconSm = LoadIcon(hInstance,MAKEINTRESOURCE(IDI_ICON_SM));
```

MAKEINTRESOURCE 是一个 Windows 定义的宏，带有一个整数参数，并将它转换成一个对 Windows 资源管理函数有意义的某种东西。如果读一下 Develop Studio 在用户为图标创建资源脚本时生成的 resource.h，就会发现 IDI_ICON_LRG 和 IDI_ICON_SM 都已被 #define 成了整数。

在光盘本章目录上的 Resources_Icons 样例程序文件示范了如何把图标作为资源使用。

注意：用户也可以载入 rc 脚本文件中定义为字符串的图标资源，但这样做没有特别的好处。因此，建议使用整数 ID，因为这是 Developer Studio 自动生成它们的方法。

2.4.2 光标

创建光标和图标的方法一样。通常光标点阵是 32×32，但也可以大至 64×64，它们是以.cur 的文件后缀来保存。在游戏中，通常会使用用户自定义的光标，如一个十字准星、一只手、一个魔符等。

除了绘制光标以外，你还需要指定一个热区（Hotspot），即光标的活动区域。这是相对左上角(0, 0)的坐标。在 Developer Studio 里，可以单击 hotspot 按钮，然后单击需要激活的区域来设置 hotspot。

在光盘本章目录上的 Resources_Cursors 程序示范了如何创建和显示一个简单的光标，如图 2.15 所示。

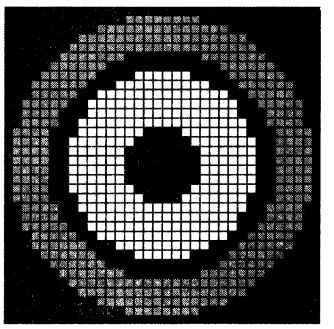


图 2.15 简单的自定义光标

技巧：如果需要一个光标能根据停留的位置不同而形状也不同，则你应当首先创建好所有要用到的光标，然后，在窗口过程中拦截 WM_SETCURSOR 消息，并相应地设置光标。

改变光标可以使用 Windows API 中的 SetCursor 函数。

2.4.3　菜单

为程序窗口添加菜单与添加光标或图标基本一样简单。创建的菜单通常会在菜单栏中出现，位置是在窗口的标题栏下面。

如果在 Developer Studio 中创建菜单，选择 Insert→Resouse→Menu 命令，如图 2.16 所示。

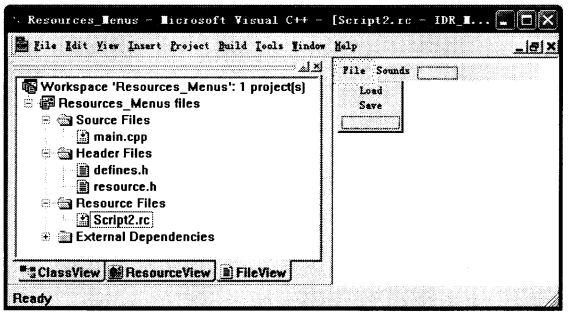


图 2.16　创建菜单

多数菜单都有几个标题项（caption），如 File、View 和 Help，在每个标题下面又有一些选项。要创建一个标题，双击第一个灰色矩形，一个选项框（option box）便会弹出。输入要用的标题并关闭选项框。现在会看到在它下方出现了第二个矩形，这是放菜单选项，比如 Save 和 Load 的场所。输入另一个标题并设置一个惟一的且易记的标识符。比如，如果标题是 Save，则 IDSAVE 是一个理想的标识符，如图 2.17 所示。

继续添加标题和标识符，直到创建了所需的菜单，然后如同图标和光标一样将其保存为资源脚本。菜单会被赋予一个默认的名字，比如 IDR_MENU1，也可以为它修改成任意的自己喜欢的名字。

然后将下面这行代码添加在窗口类定义处，使窗口使用这个菜单：

```
windass.lpszMenuName = MAKEINTRESOURCE (IDR_MENU1);
```

代码的运行效果如图 2.18 所示。

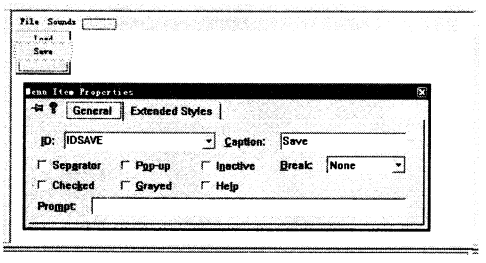


图 2.17　菜单属性

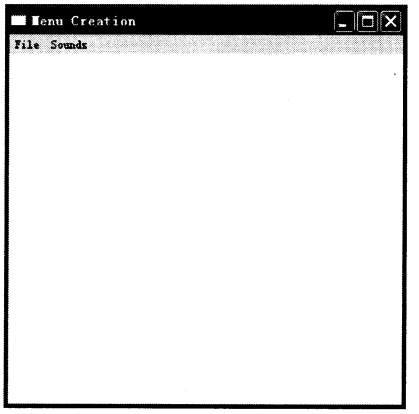


图 2.18　一个简单菜单

2.4.4 为菜单添加具体功能

现在虽然创建了一个可见的菜单，但它不能用，只有将菜单和代码联系起来，才能使它执行用户想要的功能。

当用户单击菜单的某个标题时，WindowProc 会收到一个消息。需要拦截的消息是WM_COMMAND。这条消息的 lParam 包含了发出该条消息的父窗口的句柄，而 wParam 包含了被单击的菜单项的 ID。因此，在 WindowProc 中，需要为 WM_COMMAND 创建一个 case 型语句，它根据 wParam 进行切换，再根据用户对所建的菜单项的选择来编写 case 语句。

技巧：如果使用 Developer Studio，并且安装了 Spy++，就可以在应用程序运行时用它来观察所有产生并进入消息队列的消息。消息产生不仅多而且非常快，尤其是在移动鼠标时更明显。用户可在 Developer Studio 的 Tools 菜单下找到 Spy++。这一工具有时极为有用，值得花些时间来学习一下如何正确使用它。

下面是Resources_Menus例子中WindowProc的相关代码。本例的功能是创建一个菜单，使用户能够选择播放两个不同的声音文件中的一个。

```
case WM_COMMAND:
    {
      switch(wParam)
      {
       case IDSAVE:
        {
         //进行 save 操作
        }
        break;
      case IDLOAD:
        {
          //进行 load 操作
        }
        break;
      case IDSOUND1:
        {
          PlaySound("wav1.wav", NULL, SND_FILENAME | SND_ASYNC);
        }
        break;
      case IDSOUND2:
        {
           PlaySound("wav2.wav", NULL, SND_FILENAME | SND_ASYNC);
        }
        break;
      }// 结束 WM_COMMAND 的 switch 语句
}
break;
```

创建简单的菜单比较简单，为了创建各种复杂而又功能强大的菜单，还是有无数的选项要了解，读者想进一步作更多的尝试，可以参考有关文档。

2.5 对　话　框

对话框分两种：模态（Modal）的和非模态（Modeless）的。

模态对话框是最常见的。它需要用户按下按钮或给予某种输入，才能将控制交回父窗口。About 对话框（关于对话框）就是典型的模态对话框，Developer Studio 中的查找和替换对话框也是。

非模态对话框在显示的时候，用户仍然可以操作其父窗口。此类对话框较不常见。写字板（WordPad）程序的查找和替换对话框就是非模态的例子。

本书中的例子未涉及非模态对话框使用，这里就不再讨论。简单介绍一下模态对话框。

对话框也是一种简单的窗口，通常朴实无华，标题栏可有可无，即使有，也不带最小化/最大化按钮，也没有客户区。它们和任何窗口一样有自己的窗口过程，因此也能处理消息。创建对话框时，需要指定一个对话框模板。虽然它们也可以用手工方式来创建，但毕竟十分繁琐，所以最好是利用 IDE 的资源编辑器来生成。

在 Developer Studio 中创建对话框模板，选择 Insert、Resources、Dialog Box 就会看到编辑器窗口，如图 2.19 所示。

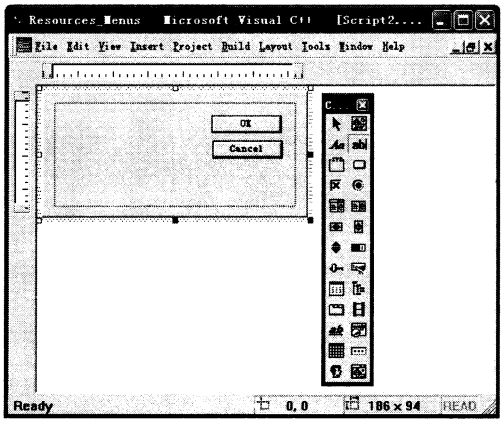


图 2.19　创建对话框

2.5.1 一个简单的对话框

首先介绍如何为上面这一例子创建一个 About 对话框。程序的代码可以在 Resources_Dialog_Box1 目录下面找到。

这个对话框只需要一个标题[①]、一些通知行文和一个单击以后就可以返回主窗口的 OK

[①] 译者注：它不是指放在标题栏中的标题。

按钮。在编辑对话框时弹出的默认对话框模板有许多其他功能，比如 Cancel 按钮和标题栏，但这里不用。为了删除 Cancel 按钮，先单击它，当它的周围出现线框的时候，按 Delete 键。然后双击标题栏，会出现一个属性对话框，如图 2.20 所示。

单击 Styles 属性页，取消选中 Title bar 单选项。同时注意对话框已经被赋予了默认的 ID，通常是 IDD_DIALOG1，而 OK 按钮的 ID 是 IDOK。用户可以使用默认值，或修改成自己较满意的名字。通常可以用默认值。

现在所要做的全部工作，就是加上一些文字，并把 OK 按钮移到合适的地方。移动按钮的方法是用鼠标选中它后进行拖放。添加文字的方法是使用工具条上的 3 个按钮之一。这 3 个按钮均对应某种文本控件。这里暂时只用静态文本（static text）按钮，这样得到的文本只能供用户阅读而不能被修改，它们和编辑框（Edit box）不同。为了添加 text，可以先单击工具条上的 Static text 按钮，然后单击准备写文本的窗口位置，文本框就会出现。双击该文本框会弹出一个属性对话框。输入需要显示的文字，并设置 Styles 属性页中显示的式样。笔者建议为每行文字单独建立静态文本框，这样比较方便操作。现在就可以制作一个如图 2.21 所示的对话框了。

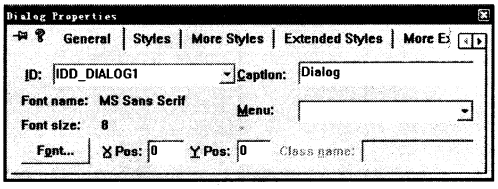


图 2.20　属性对话框

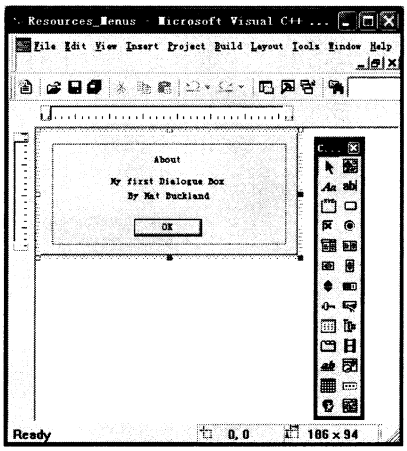


图 2.21　一个 About 对话框

一旦定义了对话框模板，就需要创建一个对话框过程来处理消息。下面的对话框过程是例程 Resources_Dialog_Box1 的形式：

```
BOOL CALLBACK AboutDialogProc ( HWND      hwnd,
                                UINT      msg,
                                WPARAM    wParam,
                                LPARAM    lParam)
{
  switch(msg)
  {
  case WM_INITDIALOG:
    {
      return true;
    }
    break;
  case WM_COMMAND:
```

```
    {
      switch(LOWORD(wParam))
      {
       case IDOK:
        {
          EndDialog (hwnd, 0);
          return true;
        }
        break;
      }
    }
    break;
  }//结束 switch 语句
  return false;
}
```

可以看出，对话框过程和窗口过程十分相似，但它们之间也有一些重要的区别。首先，它的返回值是一个 BOOL 类型而不是 LRESULT。其次，它没有利用 DefWindowProc 来管理未处理消息的功能。当 DialogProc 收到任何不能处理的消息时，它只是让函数值返回 false。最后，对话框过程没有必要处理 WM_PAINT、WM_DESTROY 或 WM_CREATE 等消息。

WM_INITDIALOG 是 DialogProc 被调用时首先收到的一个消息。如果对这一条消息的返回值是 true，Windows 将会把焦点设置在第一个子控件中，本例子即为 OK 按钮。

需要处理的其他消息只有一个，即 WM_COMMAND。WM_COMMAND 消息是用户单击 OK 按钮时发送给 DialogProc 的。按钮的 ID，即 IDOK，存放在 wParam 的低字中。因此需要测试这个值，并相应地决定是否调用 EndDialog 函数。EndDialog 函数告诉 Windows 要撤销（destroy）该对话框。

这样就创建了一个对话框模板和一个对话框过程 DialogProc，接下来要做的就是添加一些代码，当用户单击 About 菜单项时会调用这个对话框。如果读者检查样例程序里面的资源脚本的话，会发现已经添加了一个 ID 为 IDABOUT 的 About 菜单框。因此，在 WindowProc 的 WM_COMMAND 段中检测其他菜单 ID 时，检测一下 About 菜单的 ID，如果是，就调用我们的对话框。调用对话框的函数是 DialogBox：

```
int DialogBox (
  HINSTANCE  hInstance,        // 应用程序实例句柄
  LPCTSTR    lpTemplate,       // 对话框模板的 ID
  HWND       hWndParent,       // owner 窗口句柄
  DLGPROC    lpDialogFunc      // 指向 dialogbox 过程的指针
);
```

可以看出，对话框调用函数非常简单。惟一的问题是，需要将主窗口的实例句柄 hInstance 作为一个参数传递给 DialogBox。为了实现这一点，笔者在 WindowProc 起始处已经创建了一个静态的 HINSTANCE，并在 WM_CREATE 中通过以下这一行代码来取得 hInstance：

```
hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
```

有了 hInstance，利用下面的程序调用对话框：

```
DialogBox (hInstance,
           MAKEINTRESOURCE (IDD_DIALOG1),
           hwnd,
           DialogProc);
```

以下即为创建简单对话框的具体内容。

2.5.2 一些更有用的知识

上面例子里的对话框仍然不实用。最常用的对话框，应是允许用户用某种方式修改某些程序参数的对话框。下面介绍怎么做一个这样的对话框。源代码可以在光盘上的 Resources_Dialog_box2 目录下找到。

下面修改以前的弹球程序作为这种对话框一个应用例子，使用户能够改变球的个数和半径。为此，需要创建一个简单对话面板，其中应有两个编辑框：一个用来输入球的个数，另一个输入球的半径。对话框的形式如图 2.22 所示。为了简单起见，在 main.h 中建了 3 个全局变量。g_iNumBalls 用来保存球的个数，g_iBallRadius 保存球的半径，还有一个 g_hwnd 用来保存主窗口句柄的全局备份。在新的对话框过程中，它们三个将被同时用到。不过，它们并不是必须设置为全局性变量，这样设置只是为了让例子的目标能够迅速定位。

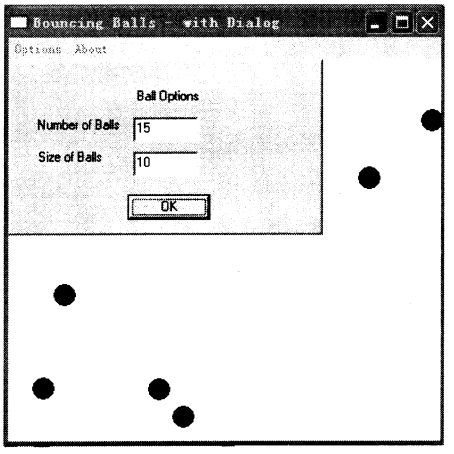


图 2.22 具有编辑框的对话框

可以和上例一样来创建一个对话框（对话面板），只是这一次需要在其中加入两个带标题（caption）的编辑框（编辑盒）。两个编辑盒的标识（identity）分别为 IDC_EDIT1 和 IDC_EDIT2。当准备好对话框模板后，就需要编写一个对话框过程。这一过程和以前所编写的对话框过程不同，因为这次需要在编辑框中显示用户可修改的参数，还要在单击 OK 按钮的时候检查是否参数被用户更改过，并相应地进行更新。通过下面的程序，看一下新的对话框过程中有哪些改动。

```
BOOL CALLBACK OptionsDialogProc(HWND        hwnd,
                                UINT        msg,
                                WPARAM      wParam,
                                LPARAM      lParam)
{
  //取出编辑控件 (edit controls) 的句柄
  HWND hwndNumBalls = GetDlgItem (hwnd, IDC_EDIT1);
  HWND hwndRadius   = GetDlgItem(hwnd, IDC_EDIT2);
```

首先记下编辑框的句柄，以便稍后使用。GetDlgItem 函数很简单，它根据对话框句柄和控件标识符返回该控件的句柄。

```
  switch(msg)
  {
  case WM_INITDIALOG:
    {
      //使用当前的球数和当前的球半径进行更新
      string s = itos(g_iNumBalls);
      SetWindowTextChwndNumBalls, s.c_str()};
      s = itos(g_iBallRadius);
      SetWindowText(hwndRadius, s.c_str());
      return true;
    }
  break;
```

对话框被调用时，当前球半径 g_iBallRadius 和个数 g_iNumBalls 应该显示在编辑框控件中。因此需要把相应参数转换成文字，并使用 SetWindowText 函数将文字写入对应的编辑框中。在这个例子里，整数可以通过 util.h 中定义的 itos 函数转换成 std::string 类型。

```
  case WM_COMMAND:
    {
      switch(LOWORD(wParam))
        {
          case IDOK:
            {
              //为每一个 edit box 收集信息,然后相应地改变对应的参数
              char buffer[5];
              //----------球数
              GetWindowText (hwndNumBalls, buffer, 5);
              //转换成一个 int
              g_iNumBalls = atoi(buffer);
              //----------球半径
              GetWindowText (hwndRadius, buffer, 5);
              //转换成一个 int
              g_iBallRadius = atoi(buffer);
```

当用户单击 OK 按钮，就需要收集文本框的内容并相应地更新参数。注意这个例子里没有错误检查。当编程者设计自己的对话框时，一定要检查可能出现的错误，否则会很麻烦。

可以通过 GetWindowText 函数取得某个编辑框控件中的文本。GetWindowText 函数原型如下：

```
int GetWindowText (
  HWND   hWnd,              // 窗口或文本控件的句柄
  LPTSTR lpString,          // 文本缓冲区的指针
  int    nMaxCount          // 要复制的字的最大数目
);
```

可以看出，赋予此函数的第一个参数是编辑框控件的句柄，第二个是用于存放文本的缓冲区地址，第三个是需要获取的字符个数。该函数若成功执行，会返回已复制的字符串的长度（不包括 null 结束符）。当一旦取到文本以后，就可以再使用 atoi 将字符串转换回整数。

```
        //把一客户消息送给 WindowProc, 这样新球就得到了
         PostMessage (g_hwnd. UM_SPAWN_NEW, NULL, NULL);
         //撤销对话框
         EndDialog (hwnd, 0);
       return true:
         }
         break;
       }
     }
     break;
   }//结束 switch 语句
   return false;
}
```

g_iNumBalls 和 g_iBallRadius 已被赋予新的值。在撤销对话框以前，需要让程序知道新的参数值，以便产生新的球的数组。为此，在消息队列中放入一条自定义的消息，并在 WindowProc 中写一个处理它的 case 语句。可以按下面的方式来自定义消息：

```
#define UM_CUSTOM_MESSAGE1 (WM_USER + 0)
#define UM_CUSTOM_MESSAGE2 (WM_USER + 1)
#define UM_CUSTOM_MESSAGE3 (WM_USER + 2)
```

在这个例子里，只需要一条自定义消息 UM_SPAWN_NEW。将消息发到消息队列可以使用 SendMessage 或 PostMessage。SendMessage 跳过消息队列直接将消息发给窗口过程，而 PostMessage 只是将消息简单地压入消息队列，使消息等待轮到它时处理。本例中使用 PostMessage。函数原型如下：

```
BOOL PostMessage (
  HWND   hWnd,              // 目标窗口句柄
  UINT       Msg,           // 要发送的消息
  WPARAM wParam,            // 消息的第一个参数
  LPARAM lParam             // 消息的第二个参数
);
```

这里送给函数的第一个参数是主窗口句柄 g_hwnd，第二个是自定义的消息 UM_SPAWN_NEW。其余两个参数均设为 NULL。这样，UM_SPAWN_NEW 消息就会进入到队列中，而对话框过程在撤销对话框后即退出。

技巧：虽然使用了全局变量来保存关于球的信息，也可以使用 PostMessage 和 SendMessage 函数的 wParam 和 lParam 参数来将球的信息传给 WindowProc。

为了完整起见，列出 WindowProc 中 UM_SPAWN_NEW 的 case 语句如下：

```
case UM_SPAWN_NEW:
    {
    //创建一个所需球数的新数组
    if (balls)
     {
       delete balls;
     }
      //创建球的数组
       balls = new SBall[g_iNumBalls];
      //把所有的球设置为随机的位置和随机的速度
       for (int i=0; i<g_iNumBalls; ++i)
       {
         balls[i].posX = RandInt(0, cxClient);
         balls[i].posY = RandInt(0, cyClient);
         balls[i].velX = RandInt(0, MAX_VELOCITY);
         balls[i].velY = RandInt(0, MAX_VELOCITY);
       }
    }
    break;
```

可以看出，这一段代码就是把原有的球的数组撤销，然后创建了一个新的符合所需大小的球的数组。整个程序就是这样的。

2.6　正 确 定 时

在开始遗传算法的讨论之前，还有最后一件事需要补充说明，这就是有关定时（timing）问题。

在编写游戏程序时，应当保证在不同配置的计算机上具有同样的速度。假如编写了一个 Pac-Man 游戏，在计算机上 Pac-Men 的行动很正常；而利用 486 运行这个程序时，老 Pac-Men 就会用每秒 3 帧的速度在屏幕上蹒跚；更糟的是，一旦把计算机升级到了快速的 AMD K15 TXL 并带有 Geforce 9 显卡时，这个小精灵会移动得很快。那样的话，游戏就出问题了。必须要有一种能够在任何计算机上都能保持同样帧数来运行程序的方法，这种方法就是采用 Windows 的定时器（timer）。

在本书中使用的 timer 类定义在 CTimer.h 和 CTimer.cpp 中。有关 CTimer 类的内部工作原理在此不作详细介绍。这里将示范如何使用这个类。这里创建了弹球程序的另一个版本，它使用了定时器，可以在光盘的 Chapter2/Bouncing Balls with Timer 目录下找到它。

如果细看 main.cpp 中的代码，你会发现我使用定时器的方法很简单。下面就是相关节的代码：

```
//创建一个定时器
CTimer timer(FRAMES_PER_SECOND);
```

首先，需要创建一个 timer 类的实例，并指定你希望 timer 运行在每秒多少帧上。我通常使用在 defines.h 中定义 FRAMES_PER_SECOND 来表示每秒帧数。

```
//启动定时器
timer.Start ();
```

然后，在你刚好进入主循环之前，调用 Start 启动定时器。

```
MSG msg;
while(!bDone)
{
  while( PeekMessage ( &msg, NULL, 0, 0, PM_REMOVE ) )
  {
    if( msg.message == WM_QUIT )
    {
      // 如果是 quit 消息就终止循环
      bDone = true;
    }
    else
    {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
  }
  if (timer.ReadyForNextFrame ())
```

接着只需要向定时器查询是否到了该处理下一帧的时间即可，方法是通过调用函数 ReadyForNextFrame，该函数当应该处理下一帧时返回 true，否则返回 false。

```
  {
//**  把更新 game 的任何代码写在这里  **
//下面是调用 WM_PAINT, 它将绘制场景
InvalidateRect (hWnd, NULL, TRUE);
    UpdateWindow(hWnd);
  }
}//结束 while 语句
```

2.7 总　　结

在本章中学习了许多的基础知识，现在应该已经能够阅读和理解本书其余部分中与 Windows 相关的代码了。第 3 章将开始讲述人工智能（AI）部分。

第2篇 遗传算法

本篇包括以下内容：

第3章　遗传算法入门

有一天，一群著名科学家聚集在一起，他们确信人类已经得到了巨大进步，现在已经不再需要上帝了。因此他们选出了一位科学家作为代表去见上帝，决定要“休”了他。

这位科学家跑到上帝那里，对他说：“上帝，我们人类已经能够自己克隆自己、能做各种各样神奇的事情，现在已经不再需要你，你为什么还不退休啊？”

上帝耐心听了这位使者的话，然后说：“很好，但是，首先让我们来进行一场制作人的比赛吧，怎么样？”

这位科学家回答说：“行啊，好极了！”

但上帝又补充说：“现在就让我们来做，就像我从前与亚当一起制作人那样。”

科学家说：“没问题，没问题，”并弯下腰去自己抓了一把泥土。

上帝看见后，说：“不，不，不，你得去拿你自己的泥土啊！”

3.1　鸟和蜜蜂

生物只有经过许多世代的进化，才能使生存与繁衍的任务获得更大成功。遗传算法也遵循同样的方式，需要经过长时间的成长、演化，最后才能收敛得到针对某类特定问题的一个或多个解。因此，了解一些有关有生命机体如何演化的知识，对理解遗传算法如何工作是有帮助的。本章的开始几页，将扼要阐述自然演化的机制（通常称为“湿”演化算法）以及与之相关的术语。即使读者以前在中学里对生物并不擅长，也无须担心。本章不会涉及过深的细节，但对于理解自然演化的基本机制已经足够。抛开以上不论，当你读完本章或下一章后，我想，你也会和我一样，深深叹服自然母亲的令人着迷！

从本质上说，任何生物机体不过就是一大堆细胞的集合。每个细胞都包含着称作染色体的相同集合的DNA链。染色体中包含的DNA分成为两股，它们以螺旋形状缠绕在一起，这就是人们所熟悉的DNA双螺旋结构，如图3.1所示。

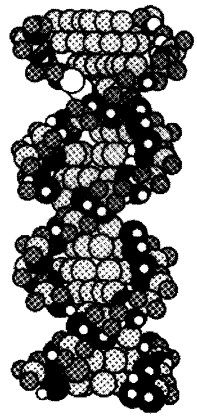


图3.1　有趣的生命双螺旋

单个的染色体是由称作基因（gene）的更小的结构模块组成，而基因则又由称作核苷酸（nucleotide）的物质组成。核苷酸一共只有4种类型，即腺嘌呤（thymine）、鸟嘌呤（adenine）、胞嘧啶（cytocine）、胸腺嘧啶（guanine）。它们常简写为T、A、C、G。这些核苷酸相互连接起来，形成若干很长的基因链，而每个基因编码了生物机体的某种特征，如头发的颜色，耳朵的样子等。一个基因可能具有的不同设置（如头发的棕色、黑

色或金黄色），称为等位基因（allele），它们沿染色体纵向所处的物理部位称为基因座位（locus）。

重要注释：等位基因不一定就限于物理形状特性的设置，某些等位基因将用来产生不同的行为模式，例如鸟类或大马哈鱼的本能性的回归行为，母亲具有抚育其新生一代的天性。

一个细胞中的染色体组（collection）包含了复制该机体所需的全部信息。这就是克隆怎样实行的秘密。你可以从被克隆施主（donor）身上，哪怕是一个血细胞中包含的信息，复制出整个生物机体，例如一只羊。新的羊将会在每一个方面和施主羊完全相同。染色体的这一集合就称为生物机体的基因组（genome）。在一特殊基因组中等位基因的一种状态称为该机体的遗传类型（genotype）。这些就是用来生成实际的生物机体（表现型，phentype本身的硬编码指令）。人都是表现型。人类 DNA 携带了人类自身的遗传类型。如将这些术语用到其他领域中，则设计汽车用的成套蓝图就是一个遗传类型；在生产线上隆隆作响的成品汽车就是一个表现型；只有设计被定型以前的，那些完全陈旧的设计，才勉强称得上是一个基因组。

现在开始讨论，怎样把所有这些应用到进化中去。对于千千万万的动物和植物——小到只有在显微镜下才能看到的单细胞生物，大到从空间卫星上也能见到的巨大珊瑚礁——地球是它们共同的家，不论它们的大小怎样、形状或颜色又怎样。一个生物机体被认为取得了成功，如果它得到了配偶并生下了一个子机体，而后者完全有希望来继续进一步复制自己。为了做到这一点，生物机体必须擅长许多工作。例如，能寻找食物和水，能面对掠食者来保卫自己、能使自己吸引潜在的配偶等。所有这些特长在某种程度上都和生物机体的遗传类型——生命的蓝图有关。生物机体的某些基因将会产生有助于它走向成功的属性，而另一些基因则可能要妨碍它取得成功。一个生物的成功的量度就是它的适应性环境。生物机体愈能适应环境，它的子孙后代也就愈多。

当两个生物机体配对和复制时，它们的染色体相互混合，产生一个由双方基因组成的全新的染色体组。这一过程就叫重组（recombination）或杂交（crossover）[①]。这样就意味着后代继承的大部分可能是上一代的优良基因，也可能继承了它们的不良基因。如果是前一种情况，后代就可能变得比它的父母更能成功（例如，它对掠食者有更强的自卫机制）；如为后一种情况，后代甚至就有可能不能再复制自己。这里要着重注意的是，愈能适应环境的子孙后代就愈有可能继续复制并将其基因传给下一个子孙后代。由此就会显示一种趋向，每一代总是比其父母一代生存和匹配得更完美。这里举一个简单的例子，假设雌性动物仅仅青睐大眼睛的雄性。这样，在追求雌性配偶的雄性中，眼睛的尺寸愈大，其获得成功的可能性也愈大。可以说动物的适应性正比于它的眼睛的直径。因此就会看出从一个具有不同大小眼睛的雄性群体出发，当动物进化时，在同位基因中，能产生大眼睛雄性动物的基因，相对于产生小眼睛雄性动物的基因，就更有可能被复制到下一代。由此可以推出，当进化几代之后，大眼睛将会在雄性群体占据统治地位。这样就可以说生物正在向一种特

[①] 译者注：crossover 一词的译法较多，如杂交、交叉、交换等，但在生物遗传学中几乎一律译为交换，而在遗传算法中则译名很不统一。

殊的遗传类型收敛。

但是，有些读者可能已经意识到，如果这是繁殖期间惟一进行的事情，那么，即使经历成千上万代后，适应能力最强的成员的眼睛尺寸也只能像初始群体中的最大眼睛一样。而根据对自然的观察中可以看到，人类的眼睛尺寸实际存在一代比一代大的趋势。之所以会发生这种情况，是因为当基因传递给子孙后代的过程中，会有很小的概率发生差错，从而使基因得到微小的改变。这多少有点像中国古老的耳语传话游戏：在一队人中，把一条消息一个个地传递下去；第一个人对着第二人的耳朵低声讲一个故事，第二个人再低声地把此故事传向第三个人，等等，直到最后那个人再把听到的故事讲出来。通常这都会弄出很多笑话，最后一个人讲出来的故事已经与原来的面目全非。这种类型的差错在把信息从一个系统传给其下一系统时实际都会发生。图 3.2 显示的一列图画是一个令人惊讶的例子。这是一次测试的结果，第一个人画出了一只鸟的图（见左上角）交给第二人，第二人看了以后重复画一个给他的下一个人，这样下去直到最后画出来的就会显现出“异化”。

有趣的事实：古代的硬币容易产生这种类型的信息丢失差错。早期厄尔利凯尔特人和条顿人所使用的硬币大量地被假冒，在开始能找到一位皇帝头像的硬币（那时已经在许多城市和乡镇用于支付）后来则变成一匹马或一碗果子的形状了。

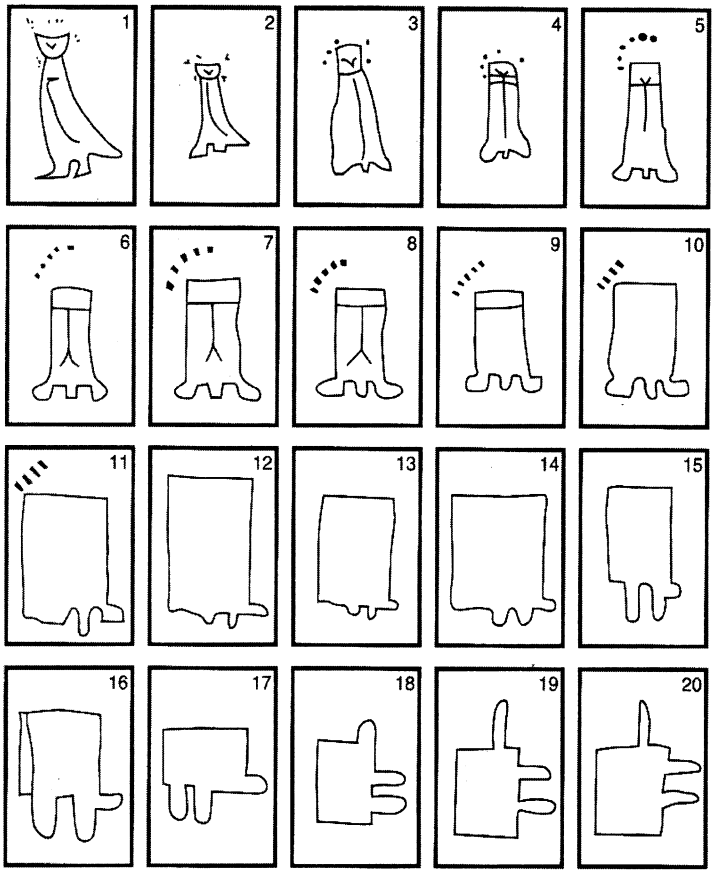


图 3.2　信息移转的一个试验（Thames 和 Hudson 提供的幻想图形）

可以认为图形或故事的细节在从一个人到另一个人的传递过程中，已经发生了变异（或突变，mutation），同样的变异在生物的繁衍过程中会在它们的基因中出现。发生变异的概率通常都很小，但在经历大量的世代之后变异就会很明显。一些变异或突变对生物将是不利的（这有最大的可能），另一些则对生物的适应性可能没有任何影响，但也有一些则可能会给生物带来一些明显的利益，使它能超过与其同类的生物。在前面讲的例子中，能使动物引起眼睛直径变大的基因突变就是一种有利的突变，它将使该动物与群体其余动物相比显得更加突出。这种趋势需要基因参与，才能使眼睛变得越来越大。当进化过程经历成千上万代之后，可以想像动物会长出一对和盛小菜用的盘子那样大的眼睛，如图 3.3 所示。

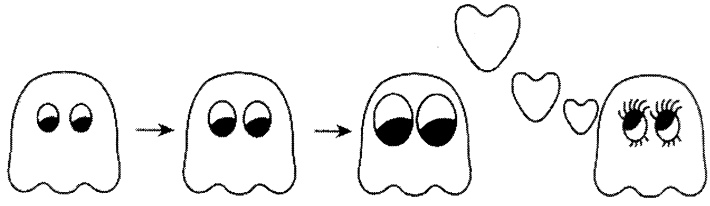


图 3.3　一个 Adonis 的进化

进化机制除了能改进已有的特征外，也能产生完全新的特征。这里再以眼睛的进化作为一个例子。可以设想，曾有一个时期动物就根本没有眼睛。那时，动物在它们的环境中航行完全是靠嗅觉和触觉来躲避掠食它们的动物。它们也相当擅长于这样做，因为它们靠这样已经经历了成千上万个世代。在那个时候，大鼻子和长手脚的雄性是受雌性的欢迎的。然而有一天，当两个动物配对时，一个基因突变发生在为皮肤细胞提供的蓝图上。这一突变使其后代在它们的头上发育出了一个具有相当光敏效应的细胞，使其后代能识别周围环境是亮的还是暗的。这样就给它带来了一个微小的优点，因为，如果一种食肉动物，比如一只鹰，来到了某个范围以内，则它将阻挡光线，这时，该动物就会感觉到，就可迅速跑到隐蔽的地方躲藏起来。另外，这种皮肤细胞还能告诉它现在是晚上或白天，或告诉它现在是在地面上或地面下，这些在捕食和吸取营养时都能为它提供方便。这一新型皮肤细胞将使这一动物与群体中其余的动物相比，具备了稍多的优点，并因此也就有更多的生存和繁殖的机会。过了一段时间，由于进化机制的作用，许多动物的染色体中都会出现具有光敏皮肤细胞的基因。

如果再作一些外推，想象这一光敏细胞基因得到了进一步的有利突变，经过了许多世代后，光敏细胞分化形成为一个区域，这个区域不断变大，产生出一些更确定的特征，例如形成一个晶体，或产生能区别颜色的视网膜细胞；还可想象一个突变使某个动物产生了两个，而不是一个光敏区域，从而就使该动物有了立体视觉。立体视觉对一个生物体来说是一个巨大的进步，因为这能精确告诉它目标离它有多远。当然，对眼睛产生不利影响的突变也可以传入那些同样的基因，但这里重要的一点是，这样生长出来的后代将不会和已具备改进眼睛的堂表亲那样取得成功，它们最终将会灭绝。只有成功的基因才会得到继承。观察自然界中存在的任何特征就能发现，它们的进化都是利用无数微小的变异发展而来的，且它们都是对拥有者有利。难以置信吧？这些重组和变异机制说明了进化怎么完成。我希

望现在你已经理解，有机体是怎么逐步形成各种不同类型的特征，以帮助它们在其生存环境中取得更大的成功。

3.2 二进制数速成

当进入更深层的学习之前，首先介绍二进制记数系统。如果已经知道二进制记数的工作原理，可以跳过这一小节。

我认为了解二进制数（基为 2 的数）的最容易的方法，就是首先查看一下你为什么使用十进制数字（基为 10 的数）和怎样使用十进制计数？

人们通常相信，人类之所以采用基数为十的记数法来计数，是因为双手共有十个手指的缘故。设想我们的一个祖先，不妨称他为 Ug，几十万年前在计算一个猛犸群中猛犸的数字。Ug 利用 2 个拳头来开始计算，当他每看到一个猛犸，就伸出一个手指。这样继续下去，直到他所有的手指都被用上为止。这样他就知道他已经算到 10 个猛犸。但因猛犸群中包含的猛犸远远超过 10 个，Ug 不得不再想一种方法来计算更大的数目。他狠抓了一下他的脑袋，就产生了一个想法：叫他的一个朋友 Frak 来帮忙。Ug 想到用 Frak 的一个手指来代表他计算到的那 10 个猛犸，然后他自己的手指就得到解脱，可重新开始用来计算第 11、12、13 个猛犸，直到 20，这时就需要使用 Frak 的另一个手指。你能看出，采用这样的过程，Ug 和 Frak 最多可以计算到 110 个猛犸，但为了统计出更多的猛犸数目，他们就不得不去招募另一位朋友了。

当人们最终学会了怎么写出数字时，就是使用类似方法来完成的。为了表示基数为 10 的数字，可以创建一系列的列（columns），每一列代表人的一双手，例如：

1000 位	100 位	10 位	个位

因此，要计数到 15，应先在个位（列）由 0 开始不断递增，直到 9，然后，因个位已不能再增，就在 10 位记 1，并重新在个位由 0 开始不断增加，直到如下结束：

1000 位	100 位	10 位	个位
		1	5

数字 15 由一个十位和 5 个个位组成。二进制数系以及其他进制数系都用同样的方式工作。但二进制计数时不用 10 个数字，而只用 2 个[①]，其中一个是 0，另一个是 1。这样，你在写二进制数时，表示数的列（在二进制中称作“bit”）的形式应为：

16 位	8 位	4 位	2 位	个位

[①] 译者注：原文误为 1 个。

现在就可以来计算 15。首先在个位（列）加 1，得：

16 位	8 位	4 位	2 位	个位
				1

这时，已经没有更大的数字可以用了（2 进制数中最大的数是 1），必须增加个位左边的那个列，并将个位数重新变为 0，因此数字 2 的形式如下：

16 位	8 位	4 位	2 位	个位
			1	0

数字 3 的形式为：

16 位	8 位	4 位	2 位	个位
			1	1

数字 4 的形式为：

16 位	8 位	4 位	2 位	个位
		1	0	0

如此这样，直到数字 15：

16 位	8 位	4 位	2 位	个位
	1	1	1	1

这就是计算 15 所要做的全部过程了。至此，应该能够转换十进制数为二进制，或把二进制转换为十进制了。二进制数字也常常写成一组有固定长度的位，特别当它与计算机联系起来讨论时如此。这就是为什么处理器常被说成是 8 位、I6 位、32 位或 64 位的原因。如果把 15 写成 8 位的二进制，应写为下面的形式，其中高位都是 0，以使整个长度达到 8：

00001111

为了加深对这一概念的理解，在你继续进入下一节以前，试回答下列问题（答案附在本章最后）：

1．把十进制 27 转换为二进制。

2．把二进制数 10101 转换为十进制。

3．把十进制数 135 表示成为一个 8 位的二进制数。

3.3　计算机内的进化

遗传算法的工作过程实质上就是模拟生物的进化过程。首先，应确定一种编码方法，使得你的问题的任何一个潜在可行解都能表示成为一个“数字”染色体。然后，创建一个由随机的染色体组成的初始群体（每个染色体代表一个不同的候选解），并在一段时期中，

以培育适应性最强的个体的办法，让它们进化。在此期间，染色体的某些位置上要加入少量的变异。经过许多代后，运气好一点，遗传算法将会收敛到一个解。遗传算法不确保一定能得到解，如果有解也不确保找到的是最优解，但只要采用的方法正确，通常都能为遗传算法编出一个能够很好运行的程序。遗传算法的最大优点就是，你不需要知道怎么去解决一个问题；仅需知道用怎么的方式对可行解进行编码，使得它能被遗传算法机制所利用。

通常，代表可行解的染色体采用一系列的二进制位作为编码。在运行开始时，首先创建一个染色体的群体，每个染色体都是一组随机的二进制位。二进制位（即染色体）的长度在整个群体中都是一样的。作为一个例子，长度为 20 的染色体的形状如下：

01010010100101001111

每个染色体都用这样的方式编码成为由 0 和 1 组成的字符串，而它们通过译码就能表示当前问题的一个解。这可能是一个很差的解，也可能是一个十分完美的解，但每一个单个的染色体都代表了一个可行解。初始群体通常都是很糟的，当一个初始的群体已经被创建好了后（不妨假设共有 100 个成员）即可开始做下面一系列工作了。

不断循环，直到寻找出一个解：

1．检查每个染色体，看它解决问题的性能怎样，并相应地为它分配一个适应性分数。

2．从当前群体选出 2 个成员。选出的概率正比于染色体的适应性，适应分愈高，被选中的概率也愈大。常用的方法就是赌轮选择（roulette wheel selection）法。

3．按照预先设定的杂交率（crossover rate），从每个选中染色体的一个随机确定的点上进行杂交（crossover）。

4．按照预定的变异率（mutation rate），通过对被选染色体的位的循环，把相应的位实行翻转（flip）。

5．重复步骤 2，3，4，直到 100 个成员的新群体被创建出来。

结束循环

算法由步骤 1 到步骤 5 的一次循环称为一个世代（或代，generation）。这里把整个的循环称为一个时代（epoch），在正文和代码中将都用这样方式来称呼。

3.3.1 什么是赌轮选择法

赌轮选择是从染色体群体中选择一些成员的方法，被选中的几率和它们的适应性分数成比例，适应性分数愈高的染色体，被选中的概率也愈大。这不保证适应性分数最高的成员一定能选入下一代，仅仅说明它有最大的概率被选中。其工作过程如下：

设想群体全体成员的适应性分数由一张饼图来代表（如图 3.4 所示），这一饼图就和用于赌博的转轮形状一样。需要为群体中每一染色体指定饼图中一个小块。块的大小与染色体的适应性分数成正比，适应性分数愈高，它在饼图中对应的小块所占面积也愈大。为了选取一个染色体，先旋转这个轮子，并把一个小球抛入其中，让它翻来翻去地跳动，直到

轮盘停止时，看小球停止在哪一块上，就选中与它对应的那个染色体。后面的章节中会讲述编写这种程序的准确算法。

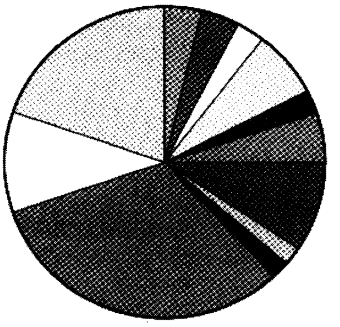


图 3.4　染色体的赌轮式选择

3.3.2　杂交率

杂交率就是用来确定两个染色体进行局部位的互换以产生两个新的子代的概率。实验表明这一数值通常取为 0.7 左右是理想的，尽管某些问题领域可能需要更高或较低的值。

每一次从群体中选择两个染色体，同时生成 0 和 1 之间之间一个随机数，然后根据此数据的值来确定两个染色体是否需要进行杂交。如果数值低于杂交率（0.7），就进行杂交，然后沿着染色体的长度随机地选择一个位置，并把在此位置之后的所有的 bit 进行互换。

例如，设给定的两个染色体为：

10001001110010010
01010001001000011

沿着它们的长度随机选择一个位置，比如说第 10 位，然后互换第 10 位之后所有位。这样两个染色体就变成了（这里在开始互换的位置加了一个空格）：

100010011 01000011
010100010 **10010010**

3.3.3　变异率

变异率（突变率）就是在一个染色体中将位实行翻转（flip，即 0 变 1，1 变 0）的几率。这对于二进制编码的基因来说通常都是很低的值，比如 0.001。

因此，无论在群体中怎样选择染色体，首先应检查是否要杂交，然后再从头到尾检查子代染色体的各个位，并按所规定的几率对其中的某些位实行突变（翻转）。

3.3.4　建议的学习方法

从现在开始直到本章结束，所有阅读材料大多数都被设计用来重读两遍。这里有很多

需要理解的新概念，且它们都是相互混杂在一起的。相信对于读者这是最好的学习方法。当你阅读第一遍时，或许只能对一些基本概念得到一些孤立的感性认识，而在阅读第二遍时，就能看到各种不同的概念是怎样相互联系起来的。而当你最后结合源程序来开始编程时，每一件事物都只是考虑怎样周密地进行安排的问题，到那时仅仅是一种如何提高知识和技能了。

注意：在本书所附的光盘的 Chaper3/Pathfinder 文件夹中，你能找到 Pathfinder 工程的全部源码。

如果你想在进一步阅读课文之前窥视一下工程的全貌，在 Chaper3/Executable 文件中有一个预先制作好的执行程序，Pathfinder.exe.

（本书各章的所有源码和执行程序都用同样方式存放在光盘相关文件夹中）

3.4 帮助 Bob 回家

寻找路径问题是游戏人工智能的一块“神圣基石”，下面就来创建一个遗传算法用在一个非常简单的场景中解决寻找路径问题。首先创建一个迷宫，它的左边有一入口，右边有一出口，并有一些障碍物散布在其中。在出发点放置一个虚拟的人鲍勃（Bob），然后要为他解决如何寻找路径的问题，使他能找到出口，并避免与所有障碍物相碰撞。下面讲述如何产生 Bob 的染色体的编码，但首先需要解释怎样来表示迷宫。

迷宫是一个二维整数数组；用 0 来表示开放的空间，1 代表墙壁或障碍物，5 为起始点，8 为出口。因此，整数数组如下：

```
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
 1,0,1,0,0,0,0,0,1,1,1,0,0,0,1,
 8,0,0,0,0,0,0,0,1,1,1,0,0,0,1,
 1,0,0,0,1,1,1,0,0,1,0,0,0,0,1,
 1,0,0,0,1,1,1,0,0,0,0,0,1,0,1,
 1,1,0,0,1,1,1,0,0,0,0,0,1,0,1,
 1,0,0,0,0,1,0,0,0,0,1,1,1,0,1,
 1,0,1,1,0,0,0,1,0,0,0,0,0,0,5,
 1,0,1,1,0,0,0,1,0,0,0,0,0,0,1,
 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
```

在屏幕中的显示效果如图 3.5 所示。

这种地图设计方法被封装在一个称为 CBobsMap 的类中，并定义为：

```
class CbobsMap
{
private:
//保存地图用的存储器(一个 2 维整数数组)
static const int map[MAP_HEIGHT][MAP_WIDTH];
```

```
static const int m_iMapWidth;   //地图的宽度
static const int m_IMapHelght;  //地图的高度
//起始点在数组中的下标
static const int m_iStartX;
static const int m_iStartY;
//终点的数组下标
static const int m_iEndX;
static const int m_iEndY;
public:
//如果需要,可以利用这一数组作为 Bob 走过的路程的存储器
int memory[MAP_HEIGHT][MAP_WIDTH];
 CBobsMap()
{
 ResetMemory();
}
//利用一个字符串来记录 Bob 行进的方向，其中每一个字符代表 Bob 所走的一步
//检查 Bob 离开出口还有多远，返回一个与到达出口距离成正比的适应性分数
double TestRoute(const vector<int> &vecPath, CBobsMap &memory);
//Render 函数利用 Windows GDI 在一个给定的作图表面上显示地图
void Render(const int cxClient, const int cyClient, HDC surface);
//画出能够存放于存储器中的任意路径
void MemoryRender(const int cxClient, const int cyClient. HDC surface);
void ResetMemory();
};
```

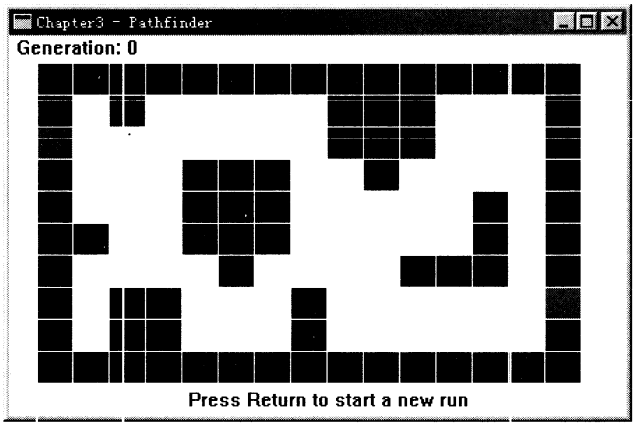


图 3.5　Bob 的迷宫

由上可知，只需要以常量的形式来保存地图数组以及起点和终点就行了。这些数据是在文件 CBobsMap.cpp 中定义的，在光盘上能找到相关的文件夹。除了存储迷宫，这个 Map 类也用来记录 Bob 在迷宫中行进的路程：memory[][]。这对遗传算法本身而言不是本质的，但为了显示目的，使读者能看到 Bob 怎样在迷宫中漫游，设置一个记录是必须的。这里重要的成员函数是 TestRoute()，它需要利用一系列的行进方向来检测 Bob 走了多远。这里不列出 TestRoute 函数的清单，只给予说明。给出一个方向向量，它的每个分量能代表向北

（North）、向南（South）、向东（East）、向西（West）4 个方向之一，让 Bob 按照它在地图中行走，TestRoute 计算 Bob 能到达的最远点的位置，然后返回一个适应性分数，它正比于 Bob 最终位置离出口的距离。他所到达的位置离开出口愈近，奖励给他的适应性分数也愈高。如果他实际已到达了出口，他将得到满分 1，这时循环就会自动结束。这时已经找到一个解。

3.4.1 为染色体编码

每个染色体必须把 Bob 的每一个行动编入代码中。Bob 的行动仅限为 4 个方向：向东，向南，向西，向北，故编码后的染色体应该就是代表这 4 个方向信息的一个字符串。传统的编码方法就是把方向变换成二进制的代码。4 个方向只要两位就够了，例如下表所示的那样：

二进制代码	十进制译码	代表的方向
00	0	向北
01	1	向南
10	2	向东
11	3	向西

这样，如果得到了一个随机的二进制字符串，就能根据它译出 Bob 行动时所遵循的一系列方向。例如染色体：

111110011011101110010101

代表的基因就是：

11,11,10,01,10,11,10,11,10,01,01,01

当把二进制代码转化为十进制时，就成为

3,3,2,1,2,3,2,3,2,1,1,1

再把这些放进一个表格中。

二进制代码	十进制译码	代表的方向
11	3	West
11	3	West
10	2	East
01	1	South
10	2	East
11	3	West
10	2	East
11	3	West
10	2	East
01	1	South

续表

二进制代码	十进制译码	代表的方向
01	1	South
01	1	South

到此，所要做的全部就是将 Bob 置于迷宫的起点，然后告诉他根据这张表中所列的方向一步步地走。如果一个方向使 Bob 碰到了墙壁或障碍物，则只需忽略该指令并继续按下一条指令去走就行了。这样不断下去，直到用完所有方向或 Bob 到达出口为止。假如有几百个这样的随机的染色体，它们中的某些就可能为 Bob 译码出到达出口的一套方向（问题的一个解），但它们中的大多数将是失败的。遗传算法以随机的二进制串（染色体）作为初始群体，测试它们每一个能让 Bob 走到离出口有多么近，然后让其中最好的那些来孵化后代，期望它们的“子孙”中能有比 Bob 走得离出口更近一点。这样继续下去，直到找出一个解，或直到 Bob 绝望地在一个角落里被困住不动为止。

因此，必须来定义一种结构，其中包含一个二进制位串（染色体），以及一个与该染色体相联系的适应性分数。这个结构称为 SGenome 结构，它的定义如下：

```
struct Sgenome
{
  vector<int>      vecBits;
  double          dFitness;
  SGenome ():dFitness(0){}
  SGenome(const int num_bits):dFitness(0)
  {
    //创造随机二进制位串
    for (int i=0; i<num_bits; ++i)
     {
      vecBits.push_back(RandInt(0,1));
     }
  }
};
```

如果在创建 SGenome 对象时把一个整型数作为参数传递给构造函数，则它就会自动创建一个以此整数为长度的随机二进制位串，并将其适应性分数初始化为零，完成对基因组的设置。

程序注释：

std::vector 是 STL（Standard Templete Library,标准模板库） 的一部分，它是一个为处理动态数组而预先建立好的类。要把数据赋给它可使用 push_back()方法。下面是一个简单的例子：

```
#include<vector>
std::vector<int> MyFirstVector;
for (int i=0; i<10; i++)
{
    MyFirstVector.push_back(i);
```

```
    cout << endl << MyFirstVector[i];
}
```

要清空一个向量，使用 clear()方法：

```
MyFirstVector.clear();
```

可以利用 size()方法来得到向量中元素的数目：

```
NyFirstVector.size()
```

使用时不需要考虑内存管理问题——std::vector 能够做这些工作。需要时，可以在整个程序中使用它。

SGenome 结构中不具备如何为染色体（vecBits）进行译码的知识，这是需要由遗传算法类自己来完成的一项任务。下面简单介绍这个类的定义，这里将其称为 CgaBob 类。

```
class CgaBob
{
  private:
  //基因组群体
  vector<SGenome>     m_vecGenomes
  //群体的大小
  int                 m_iPopSize
  double              m_dCrossoverRate;
  double              m_dMutationRate;
  //每个染色体含有多少 bits
  int                 m_iChromoLength;
  //每个基因有多少 bits
  int                 m_iGeneLength;
  int                 m_iFittestGenome;
  double              m_dBestFitnessScore;
  double              m_dTotalFitnessScore;
  int                 m_iGeneration;
  //为 map 类创建一个实例
  CBobsMap            m_BobsMap;
  //另一个 CbobsMap 对象，用来保存每一代的最佳路径的一个记录
  //这是被访问小格的一个数组，它仅为了显示目的而使用
  CBobsMap            m_BobsBrain;
  //检测运行是否仍在进行中
  bool                m_bBusy;
  void                Mutate(vector<int> &vecBits);
  void                Crossover (const vector<int>  &mum,
                                 const vector<int>  &dad,
                                 vector<int>        &baby1,
                                 vector<int>        &baby2);
  SGenome&            RouletteWheel Selection();
  //用新的适应性分数来更新基因组原有的适应性分数
  //并计算群体的最高适应性分数和适应性分数最高的那个成员
  void                UpdateFitnessScores();
```

```
    //把一个位向量译成为一个方向的（整数）向量
    vector<int>     Decode(const vector<int> &bits);
    //把一个位向量变换为十进制数。用于译码
    int             BinToInt(const vector<int> &v);
    //创建一个随机的二进制位串的初始群体
    void            CreateStartPopulation();
    public:
    CgaBob(double cross_rat,
           double  mut_rat,
           int     pop_size,
           int     num_bits,
           int     gene_len):m_dCrossoverRate(cross_rat),
                             m_dMutationRate
                             (mut_rat),
                              m_iPopSize(pop_size),
                              m_iChromoLength(num_bits),
                              m_dTotalFitnessScore(0.0),
                              m_iGeneration(0),
                              m_iGeneLength(gene_len),
                              m_bBusy(false)
      {
         CreateStartPopulation();
      }
    void        Run(HNND hwnd);
    void        Epoch();
    void        Render(int cxClient, int cyClient, HDC surface);
    //访问用的方法
     int        Generation(){return m_iGeneration;}
     int        GetFittest(){return m_iFittestGenome;}
     bool       Started(){return m_bBusy;}
     void       Stop(){m_bBusy = false;}
};
```

由上述内容可知，当这个类的一个实例被创建时，构造函数初始化所有的变量，并调用 CreateStartPopulation()。这一短小函数创建了所需数量的基因组群体。每个基因一开始包含的是一个由随机二进制位串组成的染色体，其适应性分数则被设置为零。

3.4.2　Epoch（时代）方法

遗传算法类中最为实际的内容就是 Epoch()方法。这就是本章早些时候讲过的遗传算法的那个循环。它是这个类中执行工作的部门。这一方法与所有工作或多或少都联系在一起。下面就对它进行详细讲述。

```
void CgaBob::epoch()
{
 UpdateFitnessScores();
```

在每一个 epoch 循环内所要做的第一件事情，就是测试染色体群中每一个成员的适应性分数。UpdateFitnessScores()是用来对每个基因组的二进制染色体编码进行译码的函数，而由它再把译码所得到的一系列结果，也就是由代表东、南、西、北 4 个方向的整数，发送给 CBobsMap::TestRoute。后者检查 Bob 在地图中游走了多远，并根据 Bob 离开出口的最终距离，返回一个相应的适应性分数。计算 Bob 的适应性分数程序如下：

```
int DiffX = abs(posX - m_iEndX);
int DiffY = abs(posY - m_iEndY);
```

这里，DiffX 和 DiffY 就是 Bob 所在格子的位置相对于迷宫出口的水平和垂直偏离值。观察图 3.6，灰色小格代表 Bob 通过迷宫的路程，上面写着 B 的小格是他最终所到达的地方。在这一位置上，Diffx = 3，DiffY = 0。

```
return 1/(double)(DiffX+DiffY+1);
```

上面的一行程序就是计算 Bob 的适应性分数。它把 DiffX 与 DiffY 这两个数字加起来然后求倒数。DiffX 与 DiffY 的和数中还加了一个 1，这是为了确保除法不会出现分母为零的错误，如果 Bob 到达出口，Diffx + DiffY = 0。

UpdateFitnessScores 也保持对每一代中适应性分数最高的基因组以及与所有基因组相关的适应性分数的跟踪。这些数值在执行赌轮选择时要使用。

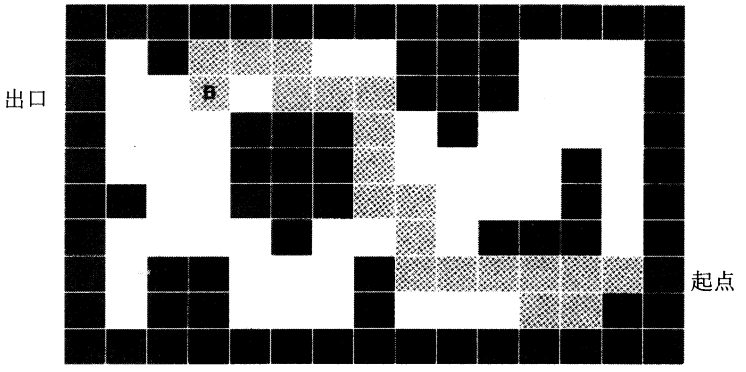


图 3.6　Bob 尝试寻找迷宫出口

重新讨论 Epoch 函数。由于在每一个 Epoch 中将会创建一个新的基因组群，因此，当它们在创建出来时（每次 2 个基因组），需要寻找一些地方来保存它们。

```
//现在创建一个新的群体
int NewBabies = 0;
//为婴儿基因组创建存储器
vector<SGenome> vecBabyGenomes;
```

现在继续讨论遗传算法循环中所处理的事务。

```
while (NewBabies < m_iPopSize)
 {
  //用赌轮法选择两个上辈（parents）
```

```
    SGenome mum = RouletteWheelSelection();
    SGenome dad = RouletteWheelSelection();
```

在每次迭代过程中，需要选择两个基因组来作为两个新生婴儿的染色体的父辈，分别称为 dad（父亲）和 mum（母亲）。一个基因组的适应性愈强，则由赌轮方法选择作为父母的几率也愈大。

```
    //杂交操作
    SGenome baby1, baby2;
    Crossover (mum.vecBitsdad.vecBits,baby1.vecBits,baby2.vecBits);
```

以上 2 行程序的工作是：创建两个空白基因组，这就是两个婴儿；它们与所选的父辈一起传递给杂交函数 Crossover ()。这一函数执行了杂交（需依靠 m_dCrossoverRate 变量来进行），并把新的染色体的二进制位串存放到 baby1 和 baby2 中。

```
    //变异操作
    Mutate(baby1.vecBits);
    Mutate(baby2.vecBits);
```

以上这两步是对婴儿实行突变。这听起来可怕，但这对他们是有利的。一个婴儿的位的突变概率依赖于 m_dMutationRate 变量。

```
    //把两个新生婴儿加入新群体
    vecBabyGenomes.push_back(baby1);
    vecBabyGenomes.push_back(baby2);

    NewBabies += 2;
}
```

这两个新生后代最终要加入到新的群体中，这样就完成了一次 Loop 的迭代过程。这一过程需要不断重复，直到创建出来的后代总量和初始群体的大小相同。

```
    //把所有婴儿复制到初始群体
    m_vecGenomes = vecBabyGenomes;
    //代的数量加 1
    ++m_iGeneration;
}
```

这里，原有的那个群体由新生一代所组成的群体来代替，并把代的数量加 1，以跟踪当前的代。

这一 Epoch 函数将无止境地重复，直到染色体收敛到了一个解，或用户要求停止时为止。上述各种操作（算子）的代码会在后面章节给出，在此首先讲述如何确定使用参数的值。

3.4.3　参数值选择

程序所用的所有参数存放在文件 defines.h 中。这些参数中大多数是一目了然的，但有

几个需要说明一下，即

```
#define CROSSOVER_RATE 0.7
#define MUTATION_RATE 0.001
#define POP_SIZE 140
#define CHROMO_LENGTH 70
```

如何确定这些变量的初值，而这是价值百万美元的问题，但至今还没有快速而有效的规则，有的只是一些原则性的指导。而且，选择这些值最终还应归结为每个用户对遗传算法的体验，用户只能通过自己的编程实践，用各种不同的参数值进行调试，看结果会发生什么，并从中选取适合的值。不同的问题需要不同的值，但是通常来说，如果在使用二进制编码的染色体，则应把杂交率定在 0.7，变异率定在 0.001，这将是很好的初始默认值。而确定群体大小的一条有用规则是将基因组的数目取为染色体长度的 2 倍。

因 70 表示 Bob 的 35 步的最大可能移动数目，所以这里选择 70 作为染色体长度，它比 Bob 为穿越地图到达出口所需的步数还要大一些。学习了以后几章的知识后可以使遗传算法变得更为有效，到时将这个长度减小即可。

历史的注释：遗传算法是 John Holland 创造的，在 20 世纪 60 年代初期，他已提出了这种想法。但不可思议的是，他没有感到需要在计算机上实际试验出结果，而是用笔和纸来修修改改。直到后来他的一名学生编写出程序并在一台个人计算机上运行后，才使人们终于看到在软件中利用他的思想能够得到什么。

3.4.4 算子函数

重新阐述一下遗传算法的各种操作（或称算子）函数——选择、杂交、变异——的代码。读者在了解遗传算法的知识时对它们具有更确切的认识。

3.4.4.1 重温赌轮选择

首先讲述赌轮选择算法，这一个函数的功能是从群体中选择一个基因组，选中的几率正比于基因组的适应性分数。

```
SGenome& CgaBob::RouletteWheelSelection()
{
   double fSlice = RandFloat()*m_dTotalFitnessScore{123456789};
```

在从零到整个适应性分数范围内随机选取了一实数 fSlice。可以将此数看作整个适应性分数饼图中的一块，如图 3.4 所示。[①]

```
double       cfTotal = 0;
int              SelectedGenome = 0;
for (int i=0; i<m_iPopSize; ++i)
{
  cfTotal += m_veceenomes[i].dFitness;
```

[①] 译者注：但并不是其中一块。

```
        if (cfTotal > fSlice)
          {
            SelectedGenome = i;
            break;
          }
      }
      return m_vecGenomes[SelectedGenome];
}
```

现在，程序通过循环来考察各个基因组，把它们相应的适应性分数一个一个累加起来，直到这一部分累加和大于 fSlice 的值时，即返回该基因组。

3.4.4.2　重温杂交算子

这一函数要求两个染色体在同一随机位置上断裂开来，然后将它们在断开点以后的部分进行互换，以形成两个新的染色体（子代）。

```
void CgaBob::Crossover (const vector<int>   &mum,
                        const vector<int>   &dad,
                        vector<int>         &baby1,
                        vector<int>         &baby2)
{
```

这一函数共传入 4 个参数，参数传递均采用引用（reference）方式，其中前两个传入父代 parent 染色体（这里染色体只是一个整数型的矢量 std::vector），后 2 个则是用来复制子代染色体的空矢量。

```
if ( (RandFloat() > m_dCrossoverRate) || (mum == dad) )
  {
    baby1 = mum;
    baby2 = dad;
    return;
  }
```

程序的作用为：首先进行检测，决定 mum 和 dad 两个上辈是否需要进行杂交。杂交发生的概率是由参数 m_dCrossoverRate 确定的。如果不发生杂交，则两个父辈染色体不作任何改变地就直接复制为子代，函数立即返回。

```
int cp = RandInt(0, m_iChromoLength - 1);
```

沿染色体的长度随机选择一个点断裂开染色体。

```
    for (int i=0; i<cp; i++)
    {
      baby1.push_back(mum[i]);
      baby2.push_back(dad[i]);
    }
    for (i=cp; i<mum.size(); i++)
    {
```

```
            baby1.push_back(dad[i]);
            baby2.push_back(mum[i]);
        }
    }
```

这两个小循环把 2 个 parent 染色体在杂交点（CP,crossover point）以后的所有位进行了互换，并把新的染色体赋给了 2 个子代: baby1 和 baby2。

3.4.4.3 重温变异算子

这一函数所做的工作是沿着一个染色体的长度，一位一位地进行考察，并按 m_dMutationRate 给定的几率，将其中某些位进行翻转。

```
void CgaBob::Mutate(vector<int> &vecBits)
{
   for (int curBit=0; curBit<vecBits.size(); curBit++)
    {
     //是否要翻转此位?
     If (RandFloat() < m_dMutationRate)
       (
         //是,翻转此位
         vecBits[curBit] = !vecBits[curBit];
       )
    }//移到下一个位
  }
```

至此，完成第一遗传算法程序。下面讲述在运行 Pathfinder 程序时，会有什么样的效果。

3.4.5 运行寻路人程序

运行 Pathfinder 程序时，程序不是每次都能找到一条通往出口的路径。Bob 有时会在一个局部地区不确定地走来走去，试着寻找他的回家路。这主要由于群体太快地收敛到一个特殊类型的染色体。这样，由于群体中的成员变得如此相似，crossover 算子的优势这时实际上已经不能发挥作用，所有发生的事情都是靠总量很少的变异操作在起作用。但因变异率设置很低，当染色体类型的差异消失后，仅仅依靠变异本身已不能去发现一个解。另外，由于赌轮选择的工作方式，使得任何一代的最合适的染色体无法保证传到下一代。这意味着，只要在适当时候，立即剔除这个染色体，遗传算法就能在群体中找到一个几乎完全的解，同时将失去拥有的所有好的基因！在后面的章节中，将会谈到这些问题，并介绍一些技术来帮助维护基因组的差异性且同时能保留那些较好的基因组。但在这里，首先介绍一些不同的编码方法，并考察它们如何和使用中可能遇到的问题类型关联在一起。这些问题将在第 4 章中作介绍。

3.4.6 二进制数转换 3 个问题的答案

（1）11011　　（2）21　　（3）10000111

3.5　练　　习

本章起每一章的后面，都会给出一些问题。不强调这对编程有多么重要，但这是惟一能使读者对那些算法产生“感性”认识的方法。而且，当开始去做复杂的题目时，这种“感性”认识将变得非常重要。

1．为杂交率、突变率、群体尺寸、染色体长度等参数设置各种不同的值来进行试验，观察它们对算法的效率有什么影响？

2．尝试去掉杂交操作，而增加突变率，会出现什么后果？如果只用杂交操作，而不利用突变，又会发生什么？

3．修改适应性分数的计算函数，使多次进入同一小格的染色体得到惩罚。这应该导致更有效的到出口的路径。

4．用别的什么办法能够使路径变得更为有效？

第 4 章　置换码与巡回销售员问题

有位过路的销售员看到一个老农夫坐在门口，他便走上前去。农夫身边是一头只有一条腿的猪。销售员还没展开拿手的推销技巧，好奇心已经攫住了他的心。

“打扰了，先生，你的猪怎么只有一条腿？”销售员问道。

“哎呀，年轻人，说来话长。当年俺在这块屋子后面 40 公顷的地里耕作时，不知为什么那拖拉机就翻了车，把俺扣在了底下。我血直流呀，心想这回死定了。谁知道这头猪就跑过来，拿鼻子拱啊拱的，把我给掘了出来，还一直拖回到了屋里。说出来谁信哪，可这头猪确实救过我的命。”

“噢，太不可思议了。”销售员说，“但我还是不明白它怎么只有一条腿呢？”

“呵呵，小伙子，要是这头聪明的猪是你的，你舍得一顿把它吃掉吗？”

第 3 章已经介绍了遗传算法的基础知识，本章将要介绍一种完全不同的遗传算法的编码方法，用来解决包含置换（Permutation）的问题。这种问题的一个很好的例子就是著名的巡回销售员问题。

4.1　巡回销售员问题

巡回销售员问题的意思是：

给定几个城市，巡回销售员必须决定一条最短的路线，使他能够访问到每个城市一次，然后返回到他的起点，如图 4.1 所示。

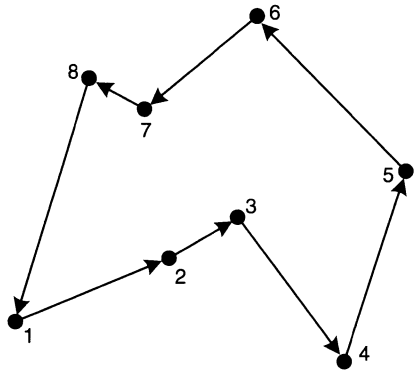


图 4.1　一个简单的 8 城市 TSP 问题

巡回销售员问题通常缩写成 TSP。它是一个非常具有迷惑性的简单问题，它属于数学家们 NP-Complete 问题集中的一个。这里没有必要纠缠“NP-完全”具体的含义是什么（要弄懂它需要很多数学知识作基础），但是基本上，难度来自于此：当城市数目不断增加时，

求解问题所需要的计算力呈指数级增长。这意味着，假设在某台计算机上某个算法可以解决 50 个城市的 TSP，但你再增加 10 个城市，采用同一个算法，就需要快一千倍以上的计算机才能解决它。在任何高速的机器上，采用同一算法，都会很快达到它的极限。

此类问题常常能够在为战略游戏（strategy game）编写人工智能时遇到。在一个游戏单元中，往往需要生成从某个站点（waypoint）出发到另一个站点结束的最短路径，沿途经过一些预设地区以获得资源、食物、补充能量等。它也可以用在 Quake 类 FPS 游戏中为 bot（游戏中机器人）作路径规划的人工智能的一部分。显然，遗传算法在今日的个人计算机上不可能对此类问题进行实时求解，但将它作为一种离线（Offline）的工具用于 AI 的开发阶段，还是极有价值的。如果游戏包含了某种随机生成的地图/水平级（map/level generation），它甚至也可能用于地图生成代码（map-creation code）的 20 级。

在你的学习过程中设置 TSP 这一内容的好处之一就是可以观察代码的改动怎样奇妙地影响到结果。通常，在编写遗传算法时，很难将一些因素可视化，也就是说，无法看到用户在算法中采用不同的变异或杂交操作时，对结果有怎样的影响，无法看到各种最优化技术的效果会怎样。但 TSP 提供了极好的可视性弥补了这个缺陷。

表 4.1 列出了一些从 19 世纪 50 年代开始，TSP 求解领域的里程碑。

当开始调试遗传算法时，解出一个具有超过 15000 个城市的 TSP 是很了不起的，但现在还是先来考察 20 个左右的城市规模。这比较适合读者的首次实验，这样的规模大体也是在游戏单元中可能用到的站点数目。

表 4.1　TSP 问题求解的里程碑

年　份	学　者	城 市 数 目
1954	Dantzig, Fulkerson, and Johnson	49
1971	Held and Karp	64
1975	Camerini, Fratta, and Maffioli	100
1977	Grötschel	120
1980	Crowder and Padberg	318
1987	Padberg and Rinaldi	532
1987	Grötschel and Holland	666
1987	Padberg and Rinaldi	2392
1994	Applegate, Bixby, Cook, and Chvátal	7397
1998	Applegate, Bixby, Cook, and Chvátal	13509
2001	Applegate, Bixby, Cook, and Chvátal	15112

4.1.1　小心陷阱

第 3 章里用过的方法已经不能解决问题了，TSP 问题的主要特点就是：任何解都是问题中所有城市的一个置换或重新排列，因此，必须确保所有的基因组都代表一个有效的排列，也就是对所有城市的一个有效周游。如果使用第 3 章 Pathfinder 程序的基因组的二进制编码法和杂交操作，会很快地遇到困难。举 8 个城市为例，如图 4.1 所示。可以将每个

城市编码为 3 位二进制数，编号为 0～7。这样，如果有两个周游顺序，可以这样编程：

周游路径顺序	二进制编码
3,4,0,7,2,5,1,6	011 100 000 111 010 101 001 110
2,5,0,3,6,1,4,7	010 101 000 011 110 001 100 111

现在在第 4 个城市之后设一个杂交点（用一个 x 来表示），看看能得到什么样的后代：

杂交前

	二进制编码	解码后
Parent 1	011 100 000 111 x 010 101 001 110	3,4,0,7,2,5,1,6
Parent 2	010 101 000 011 x 110 001 100 111	2,5,0,3,6,1,4,7

杂交后

	二进制编码	解码后
Child 1	011 100 000 111 x 110 001 100 111	3,4,0,7,6,1,4,7
Child 2	010 101 000 011 x 010 101 001 110	2,5,0,3,2,5,1,6

图 4.2 是该杂交操作的结果，读者会发现两个后代均包含了重复的城市，因此它们都不是有效的周游路径。

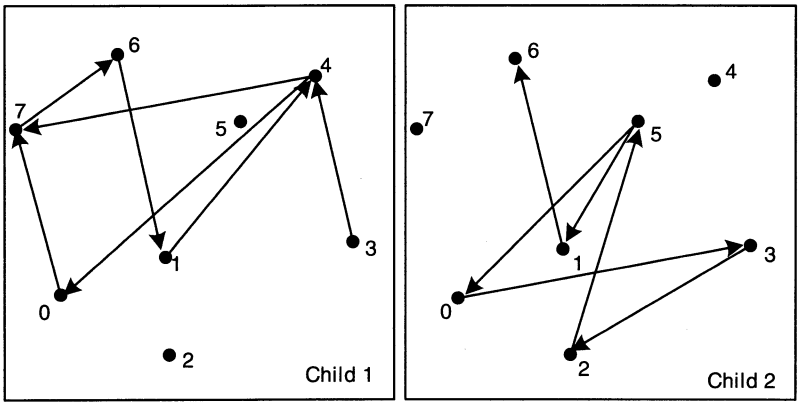


图 4.2　无效后代

为了解决城市的重复访问问题，需要提供非常繁琐的查错功能来排除所有的重复，但这样在消除重复的同时也丢失游程中积累的改进。所以，为了求解 TSP 问题，必须发明一种新的杂交操作，它只生成有效的后代。同样，对于变异操作也有类似的问题，也需要设计一种新的变异操作。为了方便起见，这里选择用整数表示城市。例如，对于表 4.1 中解 Parent 1 的游程可以简单地被表示为一个整数矢量：

3,4,0,7,6,1,4,7

就不需要浪费 CPU 周期不停地把解从十进制变换为二进（编码），再由二进换算到十进（解码）了。

4.1.2　CmapTSP，Sgenome，CgaTSP

在详细描述上面的操作之前，首先浏览一下 TSP 遗传算法程序的头文件。这个例程的源代码可以在配套光盘的相应目录下找到。

4.1.2.1　CmapTSP（类）

为了封装地图数据、城市座标以及适应性计算，创建了一个 CmapTSP 类，它的定义如下：

```
class CmapTSP
{
private:
  vector<CoOrd> m_vecCityCoOrds;
```

CoOrd 是一个保存每个城市的 x 和 y 座标的简单结构。定义如下：

```
struct CoOrd
{
  float x, y;
  CoOrd (){}
  CoOrd(float a, float b):x(a),y(b){}
}
```

下面继续讨论 CmapTSP 类：

```
  //地图中的城市数目
  int     m_NumCities;
  //客户区窗口大小
  int     m_MapWidth;
  int     m_MapHeight;
  //如果解可以计算,即保存解的长度
  double  m_dBestPossibleRoute;
  void    CreateCitiesCircular ();
```

函数 CreateCitiesCircular 把 m_NumCities 个城市排列成为一个环形，如图 4.3 所示。定义为这样的城市布局，是因为这样很容易检测最优路径（检查遗传算法的解），这也是一个将遗传算法的运行过程可视化的好方法。

```
  double  CalculateA_to_B(const CoOrd icity1, const CoOrd &city2);
```

该方法使用著名的勾股定理，也称为毕达哥拉斯（Pythagoras）定理：“直角三角形斜边的平方等于两直角边的平方和”这条定理计算两个城市之间的距离，如图 4.4 所示。

```
  void    CalculateBestPossibleRoute ();
```

该函数为环形排列好的城市计算可能的最佳（最短）周游路径。由于城市排列成了环形，答案是显而易见的。最短路径就是以环形顺序将各个城市依次连接得到的环形链，如

图 4.5 所示。

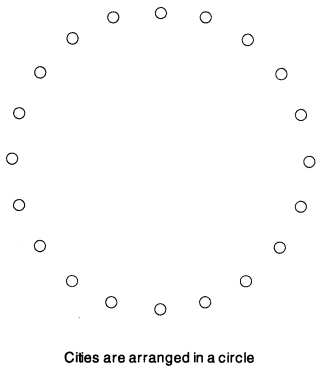


图 4.3　环形城市排列

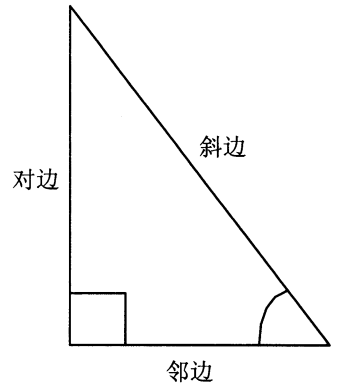


图 4.4　直角三角形的三边

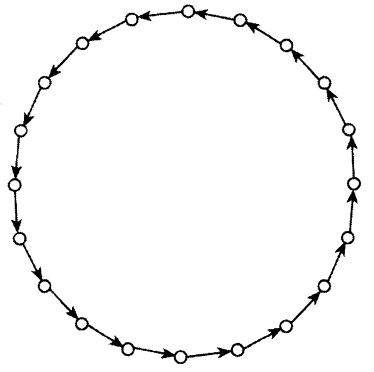


图 4.5　最终结果的路径为一个圆

其实，CalculateBestPossibleRoute 所作的工作，就是在周游路径上，将依次相邻的城市，逐对调用 CmapTSP::CalculateA_to_B 函数，计算出它们之间的距离，并把这些距离加在一起，将它们的总和作为函数值返回。

```
public:
  CmapTSP (int w, int h, int nc):m_MapWidth(w),
                                 m_MapHeight(h),
                                 m_NumCities (nc)
  {
    //计算城市的坐标
    CreateCitiesCircular ();
    CalculateBestPossibleRoute ();
  }
```

当该类的一个实例被创建时，所需数目的城市的座标就被创建，并计算出可能的最佳周游路径。城市的座标保存在 m_vecCityCoOrds 里。

```
  //一旦用户改变了窗口尺寸时使用
  void    Refresh(const int new_width, const int new_height);
```

```
    double  GetTourLength (const vector<int> &route);
```

给定一个有效的城市周游路径，GetTourLength 返回周游路径的总的路程长短它是适应性函数的关键所在。

```
    //供调用者使用的方法
    double BestPossibleRoute(){return m_dBestPossibleRoute;}
    vector<CoOrd> CityCoOrds(){return m_vecCityCoOrds;}
}
```

图 4.6 是遗传算法胜利完成一轮运行，并找到了最优周游路径之后程序的一幅截图。如果读者想再进一步学习，可以先运行一下程序，在光盘上的相关目录里可以找到预先编译好的可执行文件。

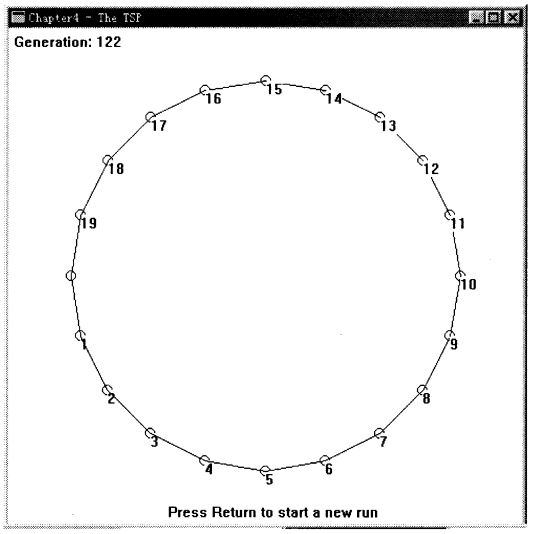


图 4.6　完成一轮运行

4.1.2.2　SGenome

基因组 sgenome 结构定义如下：

```
struct SGenome
{
    //城市周游路径（基因组）
    vector<int>       vecCities;
    //它的适应分
    double              dFitness;
    //构造函数或构造器（Constructor）
    SGenome ():dFitness(0){}
    SGenome (int nc):dFitness(0)
    {
        vecCities = GrabPermutation (nc);
    }
    //创建一个随机的城市周游路径
```

```
    vector<int>  GrabPermutation (int &limit);

    //在 GrabPermutation 中使用
    bool  TestNumber(const vector<int> &vec, const int &number);

    //重载'<',用于排序
    friend bool operator<(const Sgenome& lhs, const int rhs)
    {
    return (lhs.dFitness < rhs.dFitness)
    }
};
```

基因组由一个保存在整数向量 std::vector 里的候选周游路径 vecCities 和一个适应性分数 dFitness 组成。

当创建 SGnome 对象时，向构造函数传入一个整数 n 代表周游的城市个数，然后，构造函数就会调用 GrabPermutation 来创建一个序列（0,1,...,n）的随机置换（Permutation）并将其保存在 vecCities 中。基因组这就准备好了可以加入种群了。

GrabPermutation 函数代码如下：

```
vector<int> SGenome::GrabPermutation (int &limit)
{
   vector<int> vecPerm;
   for (int i=0; i<limit; i++)
   {
     //使用 limit-1,是因为希望整数数组从 0 开始计算
     int NextPossibleNumber = RandInt(0, limit-1);
     while(TestNumber(vecPerm, NextPossibleNumber))
     {
         NextPossibleNumber = RandInt(0, limit-1);
     }
     vecPerm.push_back(NextPossibleNumber);
   }
   return vecPerm;
}
```

4.1.2.3 CgaTSP 类

CgaTSP 是遗传算法类。在下面的类声明中，成员变量大多都有解释，没有解释的都是不言自明的。其中的成员函数将在下一节里作更详细的讲解。

```
class CgaTSP
{
private:
   vector<SGenome>   m_vecPopulation;
   //地图类的实例
   CmapTSP*          m_pMap;
   double            m_dMutationRate;
```

```
    double              m_dCrossoverRate;
    //整个种子群体的总适应性分
    double              m_dTotalFitness;
    //在此之前找到的最短周游路径
    double              m_dShortestRoute;
    //在此之前找到的最长周游路径
    double              m_dLongestRoute;
    //种子群中基因组的数目
    int                 m_iPopSize;
    //染色体长度
    int                 m_iChromoLength;
  //最新一代中适应分最高的成员
    int                 m_iFittestGenome;
    //表明已经到了哪一代
    int                 m_iGeneration;
    //帮助了解程序当前是否已进入绘图阶段
    bool                m_bStarted;
    //交换变异（Exchange Mutation）
    void                MutateEM(vector<int> &chromo);
    //部分匹配杂交（Partially Matched Crossover）
    void            CrossoverPMX(const vector<int>&mum,
                                 const vector<int> &dad,
                                 vector<int>       &baby1,
                                 vector<int>       &baby2);
    SGenome&        RouletteWheelSelection();
    void            CalculatePopulationsFitness();
    void            Epoch();
    void            Reset();
    void            CreateStartingPopulation();
public:
    //构造函数或析构器（clestrutor）
    CgaTSP ( double     mut_rat,
             double     cross_rat,
             int        pop_size,
             int        NumCities,
             int        map_width,
             int        map_height):m_dMutationRate(mut_rat),
                                    m_dCrossoverRate(cross_rat),
                                    m_iPopSize(pop_size),
                                    m_iFittestGenome(0),
                                    m_iGeneration(0),
                                    m_dShortestRoute(999999999),
                                    m_dLongestRoute(0),
                                    m_iChromoLength(NumCities),
                                    m_bStarted(false)
    {
      //设置 map
```

```
        m_pMap = new CmapTSP (map_width,
                              map_height,
                               NumCities);
        CreateStartingPopulation();
    }
    //析构函数或析构器(destructor)
    ~CgaTSP (){delete m_pMap;}
    void        Run(HWND hwnd);
    //供访问用的方法
    void        Stop(){m_bStarted = false;}
    bool        Started(){return m_bStarted;}
};
```

这里需要定义一个杂交操作、一个变异操作和一个适应性函数。就 TSP 而言，这三者中，最复杂的是杂交操作，因为杂交操作必须提供有效的后代。首先介绍杂交操作。

4.2 置换杂交操作

为了生成有效的后代，可以应用许多置换杂交，如：

- 部分映射杂交（Partially-Mapped Crossover）
- 顺序杂交（Order Crossover）
- 变位杂交（Alternating-Position Crossover）
- 最大保留杂交（Maximal-Preservation Crossover）
- 基于位置的杂交（Position-Based Crossover）
- 边重组杂交（Edge-Recombination Crossover）
- 子周游段杂交（Subtour-Chunks Crossover）
- 交点杂交（Intersection Crossover）

在本章将讨论其中最常用的部分映射杂交，常见的缩写是 PMX。第 5 章将开始描述一些其他的杂交操作，因为试验不同的操作和观察它们对遗传算法效率有什么影响对学习是很有帮助的。这里重要讲述 PMX。

假设该 8 个城市的问题已用整数编码，两个可能的父代的周游路径是：

```
Parent1: 2.5.0.3.6.1.4.7
Parent2: 3.4.0.7.2.5.1.6
```

为了实现 PMX，首先必须随机地选出两个杂交点——比如说在第 3 和第 6 个城市之后。这里把裂口（split）标记为 x，就成为下面的形式：

```
Parent1: 2.5.0.x 3.6.1 x.4.7
Parent2: 3.4.0.x 7.2.5 x.1.6
```

观察两个父代的中段，并记下其中的对应关系。在这个例子里是：

```
3 对应于 7
6 对应于 2
```

1 对应于 5

现在逐个地检查两个父代基因组中的每一个基因，每次找到和上面列出的基因之一相匹配的基因，就进行交换。一步一步地进行下去，如：

步 骤 1

Child1: 2.5.0.3.6.1.4.7

Child2: 3.4.0.7.2.5.1.6

（这一步只是直接将父代复制为子代）

步骤 2-1 [3和7交换]

Child1: 2.5.0.7.6.1.4.3

Child2: 7.4.0.3.2.5.1.6

步骤 2-2 [6和2交换]

Child1: 6.5.0.7.2.1.4.3

Child2: 7.4.0.3.6.5.1.2

步骤 2-3 [1和5交换]

Child1: 6.1.0.7.2.5.4.3

Child2: 7.4.0.3.6.1.5.2

至此基因完成杂交，得到没有重复城市的有效置换。

理解了上述内容后，可以尝试用纸笔实际模拟一下该杂交。加深理解。

有趣的事情：TSP 问题是由一个名叫 Karl Menger 的数学家、经济学家在 19 世纪 20 年代首次提出的，但直到 20 世纪 40 年代一个名叫 Merill Flood 的人开始和他在 RAND 公司的同事们展开讨论后，TSP 问题才开始普及起来。那时人们对组合问题有着极大的兴趣，数学家们喜欢 TSP，是因为 TSP 问题描述起来很简单，但要解决却是非常难。时至今日，TSP 仍然被广泛用作新的组合优化方法的测试问题。

PMX 操作的实现代码如下：

```
void CgaTSP::CrossoverPMX(const vector<int>     &mum,
                          const vector<int>     &dad,
                          vector<int>           &baby1,
                          vector<int>           &baby2)
{
baby1 = mum;
baby2 = dad;
//是否立刻返回,取决于杂交率 CrossoverRate,或者要看
//两个父辈染色体是否实际为同一个染色体
if ( (RandFloat() > m_dCrossoverRate) || (mum == dad))
{
    return;
}
```

```
    //首先选取染色体的一节（section）的开始点 beg
    int beg = RandInt(0, mum.size()-2);
    int end = beg;

    //再选一个结束点 end
    while (end <= beg)
    {
        end = RandInt(0, mum.size()-1);
    }

    //从 beg 到 end，依次寻找匹配的基因对，并
    //交换两个子染色体中的位置
    for (int pos = beg; pos < end+1; ++pos)
    {
        //这些是要作交换的基因
        int gene1 = mum[pos];
        int gene2 = dad[pos];
        if (gene1 != gene2)
        {
        //将它们从子代 baby1 中寻找出来，并进行交换
        int posGene1 = *find(baby1.begin(), baby1.end(), gene1);
        int posGene2 = *find(baby1.begin(), baby1.end(), gene2);
        swap(posGene1, posGene2);
        //再将它们从子代 baby2 中寻找出来，进行交换
        posGene1 = *find(baby2.begin(), baby2.end(), gene1);
        posGene2 = *find(baby2.begin(), baby2.end(), gene2);
        swap(posGene1, posGene2);
        }
    }//下一对
}
```

STL 注释：

find()

find 算法定义在<algorithm>中；它可以用于任何 STL 容器类上进行值的查找。它的定义是：

```
InputIterator find (InputIterator beg, InputIterator end,
const T& value)
```

什么是 iterator？简单地说，可以把 iterator 看作一个元素指针。可以用++对 iterator 作增量操作，就像对指针那样，也可以用*来访问 iterator 所对应的元素的值。Input Iterator 是一种特殊类型的 iterator，它只能逐个元素地依次增量，并且是只读的。因此，传给 find 算法的两个 iterator 分别用于定义搜索范围的开始和末尾以及需要搜索的值。find 算法在找到第一个等于该值的元素的时候就返回对应该元素的 iterator。如果没有找到匹配的元素，它就返回 end()。

begin()和 end()是容器类的成员函数，它们分别返回 iterator 代表容器中元素的开始和

末尾。end()返回的是容器最后一个元素之后的那一个位置。例如有一个 vecInts，它是由随机整数组成的 std::vector<int>向量，如果要在其中搜索一个值为 5 的元素，应首先创建正确类型的 iterator，并调用 find 来获取信息：

```
vector<int>::iterator it;
it = find(vecInts.begin, vecInts.end(), 5);
```

swap()

swap 也定义在 <algorithm> 中，用于交换容器内的两个元素。下面为它的定义：

```
void swap(T& val1, T& val2);
```

4.3　交换变异操作

讨论了 PMX 杂交操作之后，变异操作比较简单了。应注意必须提供一个只会生成有效周游路线的变异操作。交换变异操作（Exchange Mutation Operator）就能满足这个要求，它在染色体上选择两个基因并将其交换。下面举出这种染色体的例子：

5.**3**.2.1.7.**4**.0.6

变异函数随机选择了两个基因，这里是 4 和 3，并交换：

5.**4**.2.1.7.**3**.0.6

这样就产生了另一个有效排列。交换变异操作的程序如下：

```
void CgaTSP::MutateEM(vector<int> &chromo)
{
   //根据变异率 MutationRate 确定是否要返回
   if (RandFloat() > m_dMutationRate) return;
   //选择第一个基因
   int pos1 = RandInt(0, chromo.size()-1);
   //选择第二个基因
   int pos2 = pos1;
   while (pos1 == pos2)
   {
     pos2 = RandInt(0, chromo.size()-1);
   }
   //交换它们的位置
   swap(chromo[pos1], chromo[pos2]);
}
```

4.4　选择一个适应性函数

需要一个适应性函数，能够为愈短的周游路线奖励愈高的分数。可以用周游路程长度的倒数作为这个函数，但这样得到的数值分布的广度不够，使种群中最好和最差染色体的

分数差别不大。因此，采用与适应性成比例选择法时，使适应性分数最高的基因组被选上是相当难的。表 4.2 给出了一个例子。

表 4.2　TSP 周游路程长度与适应性分数

基　因	周游路程长度	适应性分数
1	3080	0.000324588
2	3770	0.000263786
3	3790	0.000263786
4	3545	0.000282029
5	3386	0.000295272
6	3604	0.000277406
7	3630	0.000275417
8	3704	0.00026993
9	2840	0.000352108
10	3651	0.000273854

一种比较理想的办法是：记下每一代中的最差的周游路线长度，然后再对种群基因组作一轮循环，从最差（即最长）的长度中减去每个基因组的长度。这样就可以使结果的相对差异变大，因此接下来使用轮盘选择法就会有效。这样做也可有效地从种群中移去最差的染色体，因为最差的适应性分数为 0，所以在选择过程中绝不会被选中，如表 4.3 所示。

表 4.3　调整过后的适应性分数

基　因	周 游 长 度	适应性分数
1	3080	710
2	3770	20
3	3790	0
4	3545	245
5	3386	404
6	3604	186
7	3630	160
8	3704	86
9	2840	950
10	3651	139

最终得到这样一个适应性分数函数：

```
void CgaTSP::CalculatePopulationsFitness()
{
 //对每一个染色体
 for (int i=0; i<m_iPopSize; ++i)
 {
    //计算染色体周游路线的路程长度
```

```
        double TourLength =
        m_pMap->GetTourLength (m_vecPopulation[i].vecCities);
        m_vecPopulation[i].dFitness = TourLength;

        //在每一代中保存最短（即最优）的路程长度
        if (TourLength < m_dShortestRoute)
        {
            m_dShortestRoute = TourLength;
        }
        //在每一代中保存最长（即最差）的路程长度
        if (TourLength > m_dLongestRoute)
        {
            m_dLongestRoute = TourLength;
        }
    }//下一个染色体
    //计算完所有周游路线的路程长度，下一步计算它们的适应性分数
    for (i=0; i<m_iPopSize; ++i)
    {
      m_vecPopulation[i].dFitness =
      m_dLongestRoute - m_vecPopulation[i].dFitness;
    }
}
```

4.5 选　　择

这里采用轮盘赌选择法，但仍有一点不同。为了使遗传算法较快地收敛，在每一个 epoch 中，在选择循环开始前，都应确保将前一代中适应性最高的 n 个基因组原样复制到新一代中。这代表适应性最好的基因组永远不会在随机过程中丢失掉。这一技术通常称为种子或精英选拔法（Elitism）。

技巧：虽然种子选拔法是遗传算法工具包（GA Toolkit）里一个很有价值的工具，且通常也是一种好的构想，但还是应谨慎使用，对于某些类型的问题可能会遇到困难，加入 Elitism 会给种群造成一种收敛过快的趋向。种群里的个体间会很快变得大同小异，从而使遗传算法只能找到非最优解，也就是通常称为的局部极小解（local minima 解）。理想的办法是，在保留种群的多样性和保留每一代中最优基因组之间，采取一些平衡措施。这些内容将在第 5 章中更详细地讨论。

4.6 把一切组合在一起

若要让一切组装在一起，需要定义一两个函数来调用已经定义好的操作，并跟踪种群中最适应的成员。在这里仍然使用 Epoch。

```
void CgaTSP::Epoch()
{
  //首先设置变量,并计算每个基因组的适应性分数
  Reset();
  CalculatePopulationsFitness();
  //如果找到了一个解,就退出
  if ((m_dShortestRoute <= m_pMap->BestPossibleRoute()))
  {
    m_busy = false;
    return;
  }
  //为新种群创建一个临时矢量
  vector<SGenome> vecNewPop;
  //首先把上一代最适应的 NUM_BEST_TO_ADD 个成员（精英）加入到种群
  for (int i=0; i<NUM_BEST_TO_ADD; ++i)
  {
    vecNewPop.push_back(m_vecPopulation[m_iFittestGenome]);
  }
  //现在再创建种群其余的成员
  while (vecNewPop.size() != m_iPopSize)
  {
    //选取两个基因组作为父代
    SGenome mum = RouletteWheelSelection();
    SGenome dad = RouletteWheelSelection();
    //创建两个子代
    SGenome baby1, baby2;
    //进行杂交
    CrossoverPMX(mum.vecCities,
                 dad.vecCities,
                 baby1.vecCities,
                 baby2.vecCities);
    //再作变异
    MutateEM(baby1.vecCities);
    MutateEM(baby2.vecCities);
    //将它们加入新种群
    vecNewPop.push_back(baby1);
    vecNewPop.push_back(baby2);
  }
  //复制到下一个 generation
  m_vecPopulation = vecNewPop;
  //generation 数加 1
  ++m_iGeneration;
}
```

可以看出，本章的 Epoch 函数和第 3 章里的 Epoch 函数很相像，差异主要在于这次选择过程中加入了精英选择。

遗传算法中的主要参数定义在 defines.h 文件里，代码如下：

```
#define WINDOW_WIDTH      500
#define WINDOW_HEIGHT     500

#define NUM_CITIES        20
#define CITY_SIZE         5

#define MUTATION_RATE     0.2
#define CROSSOVER_RATE    0.75
#define POP_SIZE          40

//必须为 2 的倍数
#define NUM_BEST_TO_ADD   2
```

该参数用于设置精英的数目：在每一代都有指定数目个适应性最高的基因实例被直接复制到新的种群中。

```
//用于矫正精度误差
#define EPSILON           0.000001
```

技巧：考虑到周游路程计算需要利用浮点运算，需要定义一个 EPSILON 以便纠正引入的精度误差。例如：在一些浮点数上执行了一系列的计算，而且知道结果应该是 X。但结果往往不是 X；或多或少有些误差。因此，如果设置下面一个条件判断：

```
if (some_number == X)
{
    do something
}
```

使用效果不是很好。为此，可改用下面这样的条件判断：

```
if ( (some_number > X-EPSILON) && (some_number < X+EPSILON) )
{
    do something
}
```

需要浮点运算的地方，它都能发挥作用。

4.7 总　　结

当运行程序时，遗传算法并不是每次都能收敛到一个解上；事实上，它经常很容易地就“卡”住了。如果稍微改改程序并增加城市数目，这个例子的实际执行效果就会变得很差。并给出类似图 4.7 那样的失败路径。

改进遗传算法的有效性的方法还有许多，将会在第5章里介绍，也会涉及到一些其他可选的变异和杂交操作。读者学习完第5章“遗传算法优化”后，将能够游刃有余地使用遗传算法。

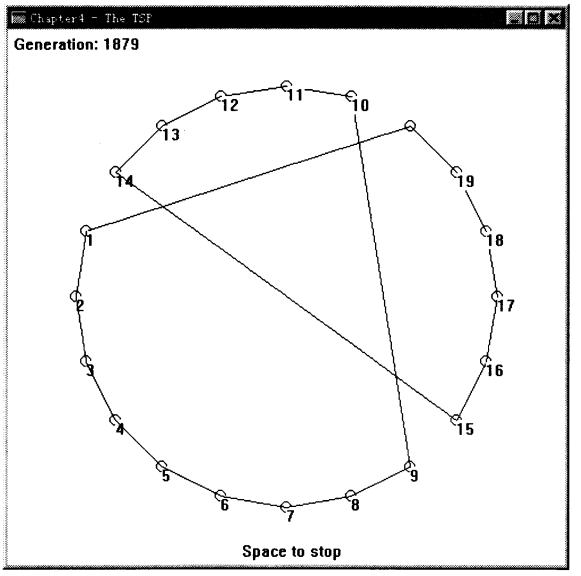


图 4.7　遗传算法没有找到（最优）解

4.8　练　　习

1．改变选择操作为精英选择类型，从前一代种群中直接复制 4 个最适应的基因组到新群中，然后在剩余个体上使用轮盘选择法。这样做算法变好还是变糟？

2．修改适应性函数，只使用周游长度的倒数。观察与原来的差异是什么。

3．完全不使用种子选拔法，观察当城市数量从十几个增加到 50 个以上时，会发生什么样的情况？

第 5 章　遗传算法优化

编程之于今日就像是软件工程师们和这个世界的一场较量。一边是软件工程师们不停地努力编写着更复杂、且不会在傻瓜用户手下崩溃的程序；一边是这个世界不停地努力制造着更多、更大，而且是更笨的傻瓜。

目前为止，这个世界暂时领先。

——Richard Cook

学习遗传算法（包括神经网络）只有通过自己动手编程，试验各种参数，才能学得更快，因为从中培养了一种感觉，知道什么可以什么不可以。这种“感觉”是很关键的。迄今，对于遗传算法几乎不存在什么“通用”的规则，这使它在科学性上平添了一份艺术性。只有时间和实践能够使编程者明白，对于特定的问题，采用多大的种子群才恰当，需要设置多高的变异率才合理等。

本章涉及的均为实例。首先，应习惯于观察各种算子，同时思考改进的方法。本章讨论一些可能会提高遗传算法性能的附加技术：例如适应性分数的各种变比技术。这里说的“可能会”提高性能，是因为每一个问题都有其特殊性，对一个问题行之有效的技术在另一个问题上可能就会有负面作用。总之，还得依靠先前提到的“感觉”。

读者可参考光盘上第 5 章的可执行文件。（图 5.1 是截图之一）。可以看出本章将要介绍的技术对解决第 4 章的 TSP 问题有很好的效果，下面就开始讲述这些技术。

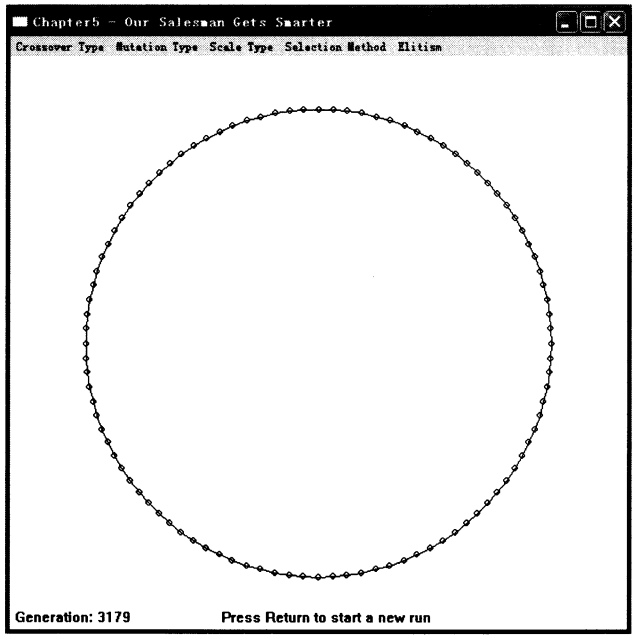


图 5.1　优化算法后的巡回销售员程序截图

5.1 TSP 用的各种算子

首先要讨论的是 TSP 问题的各种可能的变异和杂交算子。尽管这些算子中并不能让算法获得较大的改进，但为了理解生成代表城市周游路线合法排列的所有方法，还是对它们作以介绍。其中有一些算子，在观察 TSP 算法的执行过程中，提供了很有趣也很能启发思考的结果。对于每一个问题，都会有多种方法来编写算子。应多做试验，并观察它们的执行效果。这样会受益匪浅。也可以尝试发明几个自己的算子，TSP 的杂交的算子发明，确实有难度，但对于变异是可以实现的。

5.1.1 各种置换变异算子

热心遗传算法的研究者们为 TSP 问题设计了许多可用的变异算子。下面详细介绍其中效果最好的几种算子，并给出它们的实现代码。

5.1.1.1 散播变异

随机地选择一对位置，将其间的城市进行任意的移动。

0.1.2.**3.4.5.6**.7

变成

0.1.2.**5.6.3.4**.7

代码如下：

```
void CgaTSP::MutateSM(vector<int> &chromo)
{
  //是否返回由变异率决定
  if (RandFloat() > m_dMutationRate) return;
  //首先在染色体上取一个段（setion）
  const int MinSpanSize = 3;
  //下面两个整数用来保存 span（跨度）的起点和终点
  int beg, end;
  ChooseSection(beg, end, chromo.size()-1, MinSpanSize);
```

ChooseSection 是一个很小的函数，负责在一个大的区间内确定一个具有随机起点和终点的小区间，并保证小区间的跨度（span）具有最小的跨度值（MinSpanSize）。读者可参考光盘中的源代码。

```
  int span = end - beg;
  //在 beg/end 的范围内随机选择基因，进行几次交换，以打乱它们
  int NumberOfSwapsRqd = span;
  while(--NumberOfSwapsRqd)
  {
    vector<int>::iterator gene1 = chromo.begin();
    vector<int>::iterator gene2 = chromo.begin();
```

```
        //在此范围内选择两个基因位置（loci,座位）
        advance(gene1, beg + RandInt(0, span));
        advance(gene2, beg + RandInt(0, span));
        //将它们交换
        swap(*gene1, *gene2);
      }//重复
    }
```

STL 注释（STL Note）：

```
    erase()
```

erase()是某些 STL 容器的一个方法，用来从容器中删除元素。既可以为 erase()传入一个单个元素位置（作为一个 iterator）。

```
    //创建一个指向第一个 element 的 iterator
    vector<elements>::iterator beg = vecElements.begin();
    //移去第一个 element
    vecElements.erase(beg);
```

或者为 erase()传入一个待删的区间范围。该区间由一对起、止 iterators 确定。因此，为了删除一个 std::vector 的第 1 ~ 第 3 个元素，可以这样编写代码：

```
    vector<elements>::iterator beg = vecElements.begin();
    vector<elements>::iterator end = beg + 3;
    vecElements.erase(beg, end);
    insert()
```

insert()是一个用来向容器插入元素的方法。与 erase()相类似，可以选择一 iterator 来插入单个元素，也可以一次插入多个元素。下面是一个简单例子，将 vecInt1 的前 4 个元素放入 vecInt2 的第 5 个位置。

```
    vector<int> vecInt1, vecInt2;
    for (int i=0; i<10; ++i)
    {
      vecInt1.push_back(i);
      vecInt2.push_back(i);
    }
    vector<int>::iterator RangeStart = vecInt1.begin();
    vector<int>::iterator InsertPos  = vecInt2.begin()+5;
    vecInt2.insert(InsertPos, RangeStart, RangeStart+4);

    assign()
```

assign()用来将一个容器中的一串元素复制到另一容器中。举例说，如果有一个 std::vector 整数矢量 vecInts，其中含有 0 ~ 9 的所有整数，现在要建立一个新的

std::vector 但要包含 vecInts 中位置 3 ~ 6 之间的整数，可以这样做：

```
    vector<int>::iterator RangeStart = vecInt.begin() + 3;
```

```
vector<int>::iterator RangeEnd = vecInt.begin() + 6;
vector<int> newVec;
newVec.assign(RangeStart, RangeEnd);
```

也可以利用 assign 向一个 std::vector 重复添加同一个元素 n 次。例如下面的例程向一个 std::vector，vecInts 中添加了 6 次整数 999。

```
vector<int> veclnt;
veclnt.assign(6, 999);

advance()
```

advance()是一个有用的方法，可以让一个 iterator 向前推进指定数目的位置。调用方法是，传给 advance()一个 iterator 和需要向前移动的元素位置数目。

```
vector<int>::iterator RangeStart = vecInt.begin();
advance(RangeStart, 3);

sort()
```

为了对容器内的所有元素排序，可以用 sort 方法对指定区间内的所有元素进行排序，程序如下：

```
sort(vecGenomes.begin(), vecGenomes.end());
```

但要让这一排序方法工作，首先应在待排序的元素上定义某种顺序关系。在 TSP 程序里为 SGenome 结构重载了一个小于运算符<，使 SGenomes 按 dFitness 成员变量的大小排序。该算符为：

```
friend bool operator<(const SGenome& lhs, const SGenome& rhs)
{
  return (lhs.dFitness < rhs.dFitness);
}
```

5.1.1.2 移位变异

随机选择两个点，取出其间的染色体段，插入到剩余染色体的一个随机位置上。

0.1.2.**3.4.5**.6.7

变成

0.**3.4.5**.1.2.6.7

这个算子能帮助遗传算法很快收敛到一条短的路径，但为了得到解，还需要以下的代码去运行。

```
void CgaTSP::MutateDM(vector<int> &chromo)
{
  //是否返回,取决于变异率
  if (RandFloat() > m_dMutationRate) return;
  //声明一个最小的 span 尺寸
  const int MinSpanSize = 3;
```

```
    // beg, end 用来存放 span 的起点和终点
    int beg, end;
    //在 chromosome 中按 span 的起点和终点选取一段 (section)
    ChooseSection(beg, end, chromo.size()-1, MinSpanSize);
    //为起终点 beg 和 end 设置 iterators
    vector<int>::iterator SectionStart = chromo.begin() + beg;
    vector<int>::iterator SectionEnd  = chromo.begin() + end;
    //存放将要移动的 section
    vector<int> TheSection;
    TheSection.assign(SectionStart, SectionEnd);
    //将此段从当前位置上删去
    chromo.erase(SectionStart, SectionEnd);
    //把 iterator 移动到一个随机的位置以便插入
    vector<int>::iterator curPos;
    curPos = chromo.begin() + RandInt(0, chromo.size()-1);
    //插入一个 section
    chromo.insert(curPos, TheSection.begin(), TheSection.end());
}
```

5.1.1.3 插入变异

插入变异是一种相当有效的变异。IM 算子与 DM 算子很类似，惟一不同点是 IM 一次只取出一个基因并插回染色体。实验证明，IM 变异算子始终都优于前面讨论过的其他变异算子。

0.1.**2**.3.4.5.6.7

变成

0.1.3.4.5.**2**.6.7

在本章的代码中，使用插入变异作为默认的变异算子。

```
void CgaTSP::MutateIM(vector<int> &chromo)
{
    //是否返回，取决于 mutation rate
    if (RandFloat() > m_dMutationRate) return;
    //创建一个 iterator
    vector<int>::iterator curPos;
    //选择一个基因进行移动
    curPos = chromo.begin() + RandInt(0, chromo.size()-1);
    //为基因的值保存一个注释 (note)
    int CityNumber = *curPos;
    //从 chromosome 上删去此基因
    chromo.erase(curPos);
    //把 iterator 移动到插入位置
    curPos = chromo.begin() + RandInt(0, chromo.size()-1);
    chromo.insert(curPos, CityNumber);
}
```

5.1.1.4 倒置变异

倒置变异是一个十分简单的变异算子。随机地选择两点，并将其间的城市顺序颠倒。

0.**1.2.3.4**.5.6.7

变成

0.**4.3.2.1**.5.6.7

5.1.1.5 倒置移位变异

随机地选取两个点，颠倒其间的城市顺序，然后将颠倒后的城市序列移到原始染色体上的某个位置上，相当于在同一个起、止区间上依次执行 IVM 和 DM。

0.1.2.3.**4.5.6**.7

变成

0.**6.5.4**.1.2.3.7

以上最后面的两个变异算子的具体代码实现将留给读者作为编程练习。

5.1.2 各种置换杂交算子

下面列出一些较好的算子的描述和源代码。

5.1.2.1 基于顺序的杂交

为了执行基于顺序的杂交，在父代之一上随机挑选几个城市，所选出的城市的排列顺序将施加到（imposed to）另一个父代同样一些城市的排列顺序。举例如下：

Parent1: 2.**5.0**.3.6.**1**.4.7
Parent2: 3.4.0.7.2.5.1.6

在父代 Parent1 上随机选出几个位置（这里是 3 个），这些位置上的城市是 5、0 和 1，这里用粗体字标出了它们，同时记下其顺序。用 5，0，1 这个顺序调整 Parent2 上同样的城市。这样得出第一个后代 Offspring1，如下：

Offspring1: 3.4.**5**.7.2.**0.1**.6

其中城市 1 没有变动，它原来就已经处在正确的顺序上。同样顺序的操作要对另一个父代 parent1 来进行，以得出 Offspring2。注意，这里直接使用第一次已经随机选出的（3 个）位置从 parent2 上选出（3 个）元素：

Parent1: 2.5.0.3.6.1.4.7
Parent2: 3.**4.0**.7.2.**5**.1.6

由此，Parent1就变成：

Offspring2: 2.**4.0.**3.6.1.5.7

以下列出基于顺序的杂交算子的代码实现：

```
void CgaTSP::CrossoverOBX( const vector<int>    &mum,
                           const vector<int>    &dad,
                           vector<int>          &baby1,
                           vector<int>          &baby2)
{
  {
  baby1 = mum;
  baby2 = dad;
  //是否返回,取决于 crossover rate,或者
  //两个父代染色体是否同一个
  if ( (RandFloat() > m_dCrossoverRate) || (mum == dad))
  {
    return;
  }
  //保存选择的城市
  vector<int> tempCities;
  //保存所选城市的位置
  vector<int> positions;
  //首先选择第一个城市的位置
  int Pos = RandInt(0, mum.size()-2);
  //再不断随机地加入城市，直到走到无法
  //记录位置时为止
  while (Pos < mum.size())
  {
    positions.push_back(Pos);
    tempCities.push_back(mum[Pos]);
    //下一城市
    Pos += RandInt(1, mum.size()-Pos);
  }
  //从 mum 个城市中取得了 n 个城市放在 tempCities 向量
  //后,将它们的次序强加到 dad 中
  int cPos = 0;
  for (int cit=0; cit<baby2.size(); ++cit)
  {
    for (int i=0; i<tempCities.size(); ++i)
    {
      if (baby2[cit]==tempCities[i])
      {
        baby2[cit] = tempCities[cPos];
        ++cPos;
        break;
      }
    }
  }
  //反过来,从 dad 中选择同样位置的
  //那些城市,将它们的顺序强加到 mum 上
  tempCities.clear();
```

```
    cPos = 0;
    //先从dad同样的一些位置上获取（grab）这些城市
  for(int i=0; i<positions.size(); ++i)
    {
      tempCities.push_back(dad[positions[i]]);
    }
    //再使它们的顺序强加到mum上
    for (cit=0; cit<babyl.size(); ++cit)
    {
      for (int i=0; i<tempCities.size(); ++i)
      {
        if (babyl[cit]==tempCities[i])
        {
          babyl[cit] = tempCities[cPos];
          ++cPos;
          break;
        }
      }
    }
}
```

技巧：通常，如果希望为游戏中某个角色寻找最优路径（Optimum Route），只考虑移动距离是远远不够的。比方说游戏利用了一个 3D 地形引擎，那么需要考虑比如路径上的地形坡度（因为走上坡路通常较慢，而且更费燃料）和经过的地表属性（在泥泞里跋涉比行驶在沥青路面慢）等各种因素。

为了寻找最优路径，需要定义一个考虑到所有这些要素的适应性函数。这样才能在移动距离、地形坡度、地表属性之间达到一个较好的平衡。例如，可以为游戏中各种不同的地表面创建不同障碍值（或惩罚值 Penalty）的折算比（Sliding Scale）。当角色经过一块具有较高障碍值的地面，在计算站点（Waypoint）之间的距离时便会得到较高的值（注意，较高的距离值意味着较差的适应性值）。对于地形坡度也是类似的，即尽量顺坡而下，少爬陡坡。为了调整好此类平衡是需要很花时间的，但最后你会得到一个能够为各种地形单元找出最优路径而不仅仅是最短路径的遗传算法。

5.1.2.2 基于位置的杂交

类似于基于顺序的杂交，但要把使用城市的顺序，替代为使用城市的绝对位置。下面详细说明，对于和上例同样的一对父代和随机位置如何进行基于位置的杂交。

```
Parent1: 2.5.0.3.6.1.4.7
Parent2: 3.4.0.7.2.5.1.6
```

首先将 Parent1 中所选的那些城市复制到 Offspring1，位置保持不变。

```
Offspring1: *.5.0.*.*.1.*.*
```

现在，按顺序检测 Parent2 中的城市，如果某个编号的城市没有在 Offspring1 中出现过，

则依次填入 Offspring1 的空位里。本例中，这个步骤的结果如下：

Offspring1: 3.**5.0**.4.7.**1.**2.6

以同样的方式生成 Offspring2，将所选城市复制到同样的位置。

Offspring2: ***.4.0.*.*.5.*.***

然后用 Parent1 对 Offspring2 进行填充，得到

Offspring2: 2.**4.0**.3.6.**5.**1.7

代码如下：

```
void CgaTSP::CrossoverPBX(const vector<int>   &mum,
                          const vector<int>   &dad,
                          vector<int>         &baby1,
                          vector<int>         &baby2)
{
 //是否返回,取决于 crossover rate,或者
 //两个父代染色体是否同一个
 if ( (RandFloat() > m_dCrossoverRate) || (mum == dad))
  {
   //退出之前,应先确保为 baby1 和 baby2 赋予某些城市
    baby1 = mum;
    baby2 = dad;
    return;
  }
  //将 babies 初始化为-1,使今后的算法中能够
  //知道哪一个位置已经被填入
  baby1.assign(mum.size(), -1);
  baby2.assign(mum.size(), -1);
  int l = baby2.size();
  //用于保存所选城市的位置
  vector<int> positions;
  //第一个城市的位置
  int Pos = RandInt(0, mum.size()-2);
  //不断随机地加入城市,直到走到无法再记录位置时为止
  while (Pos < mum.size())
  {
    positions.push_back(Pos);
    //下一个城市
    Pos += RandInt(1, mum.size()-Pos);
  }
  //将所选的城市
  //复制到子代的同样位置上
  for (int pos=0; pos<positions.size(); ++pos)
  {
    //baby1 从 mum 接受这些城市
```

```
        baby1[positions[pos]] = mum[positions[pos]];
        //baby2 从 dad 接受这些城市
        baby2[positions[pos]] = dad[positions[pos]];
    }
    //填入空档。创建两个位置记号（markers），从而确定
    //处于 baby1 和 baby2 中的什么位置
    int c1 = 0, c2 = 0;
    for (pos=0; pos<mum.size(); ++pos)
    {
      //将 marker 前移，直到在 baby2 中到达一个空（free）位
      while( (baby2[c2] > -1) && (c2 < mum.size()))
      {
        ++c2;
      }
      //baby2 从 mum 得到下一个原来不存在的城市
      if ( (!TestNumber(baby2, mum[pos])) )
      {
        baby2[c2] = mum[pos];
      }
      //对 baby1 作同样的设置
      while((baby1[c1] > -1) && (c1 < mum.size()))
      {
        ++c1;
      }
      //baby1 从 dad 得到下一个原来不存在的城市
      if ( (!TestNumber(baby1. dad[pos])) )
      {
        baby1[c1] = dad[pos];
      }
    }
}
```

至此，通过上述算子对 TSP 问题的运行，已经勾画出遗传算法算子所涉及的范畴。在本章的剩余部分将对遗传算法的各种工具和技术进行讨论，利用它们可用来改善几乎任何一类遗传算法的性能。

5.2 各种处理工具

可以把 TSP 问题的完整可行解集想象为一片起伏不平的风景场地（landscope）。位置越低，解的适应性越高；反之，位置越高，解的适应性越低。把遗传算法想象为一个可在这种地形上四处滚动的球，直到滚入某个凹坑。陷入了这个坑，就是找到了一个稳定的解，如图 5.2 所示。显然，这里所得的不是最优解，因为还有比它更低的位置。但无论是不是最优，球一旦进入这个坑就滚不出来了。这种情况称为局部最优解，它对应的位置是一个局部极小值（Local Minima）。

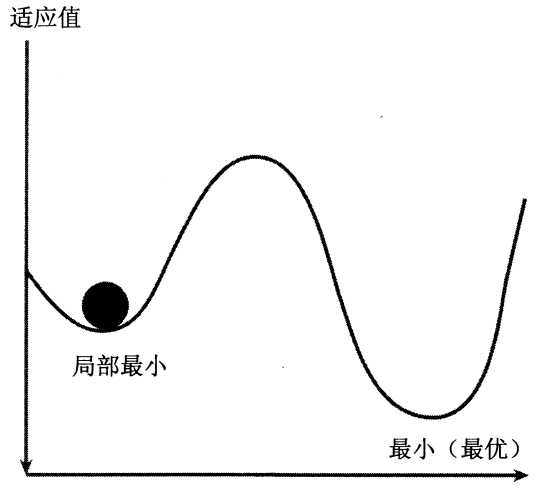


图 5.2 一个困在局部最小值的遗传算法

理想的情况是，希望这个球能在尽可能多的场地上作一些运动，直到找到最深的那个坑为止。这个结果就代表了全局最优解。也应为它提供一种能从陷入的浅坑中跳出来的办法，以便让它继续在其他场地上运动。

图 5.2 代表求解带一个参数 x 问题的适应性场景图（fitness landscape）。而图 5.3 则代表求解两个参数 x,z 的问题的适应性场景图，这个场景图需要在 3D 空间中才能表示出来。

注意：在一些书中，你会发现适应性地形和本书是上下颠倒的，其作者可能是指一个卡在局部极大值的遗传算法。无论正反，概念还是一样的。

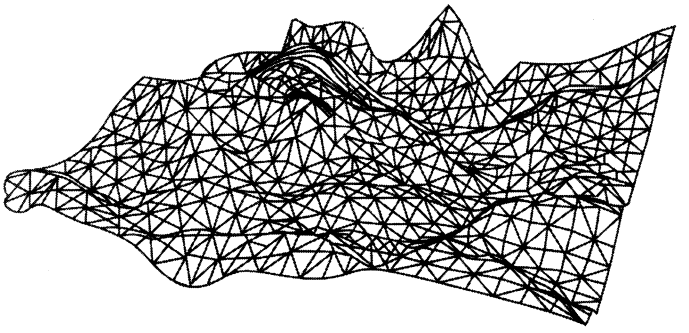


图 5.3　两参数问题的适应性地形

在图 5.3 中，x-轴和 z-轴表示两个待解参数，y-轴表示适应性大小。可以看到，在地形上布满了各种丘陵、凹槽、山脊及其他特征，遗传算法必须越过所有这些障碍才能稳定到最优解。这个图是只有两个参数的 TSP 问题的场景图。面对一个多于两个参数的问题的时候，必须具备丰富的想象力。

为了使球保持不断的滚动，需要寻找一些工具来诱使遗传算法完成你想叫它完成的工作。在本节中，将讨论有助于遗传算法更高效地收敛到一个解的各种技术和附加算子。

5.2.1 选择技术

选择，就是从群体中挑出一些个体，将其作为创建下一代个体的基因库（gene base）。有时，某些选出的个体将不作任何修改地直接送入新一代作为种子（精英，Elitism），如像第 4 章提到的那样。但较常见的情况是，将某些染色体挑选出来作为父代（parents），通过杂交、变异等过程来产生子代（offspring）。如何挑选父代是很重要的，因为它能强烈影响遗传算法的效率。如果总是挑选最强的个体，种群可能会过快地收敛到局部极小值，并停止不动。但如果只作随机挑选，遗传算法则可能需要很长时间才会收敛，甚至也有可能不收敛。选择的关键在于制定一种谋求双赢的策略——既要较快地收敛，也要保证维持种群的多样性。

5.2.1.1 精英选择

精英选择是一种确保种群中具有最高适应分的成员能过渡到下一代的方法。在第 4 章所举的例子中曾用了一小部分该方法，选送两个最佳的个体到下一代。这种方法可以推广，可以从种群中最佳的 m 个个体中选择 n 个复制到下一代，实际中总结得出，保留占种群总体 2%～5%数量的个体，效果最为理想。这种增强版精英选择函数称为 GrabNBest。该函数原型如下：

```
void GrabNBest(int              NBest,
               const int        NumCopies,
               vector<SGenome> &vecNewPop);
```

为了保留种群适应性前三名中的前两个，可以这样调用：

```
GrabNBest(3, 2, vecNewPop);
```

当围绕例子程序不断进行试验时，种子选拔法几乎可以和本章介绍的所有其他技术友好兼容，只有随机遍及取样（Stochastic Universal Sampling）例外。

5.2.1.2 稳态选择

稳态选择的原理和种子选手法有些相似，不过不是在最优的个体集中选出少量个体进入新一代，而是把当前种群中，除去少数几个表现最差的个体后，全部保留起来。被保留的个体接着要利用通常的变异和杂交算子来选择。稳态选择在解决某些问题时确实证明很有用，但对于大多数问题而言，它不被推荐使用。

5.2.1.3 适应性比例选择

这一类型的选择技术的挑选后代原则，就是把较高的被选中概率赋给那些较高适应性分数的个体，即每个个体都有一个被选中进行复制的期望值。该预期值等于该个体的适应性分数除以种群的平均适应性分数。如果有一个适应性分数为 6 的个体，而种群的平均适应性分是 4，则该个体被选中的期望次数为 1.5。

5.2.1.4　赌轮选择

轮盘赌轮选择法，或简称赌轮选择法，是实现按适应性比例选择的常用方法，但该技术也有它的缺点。由于轮盘选择法是使用随机数的一种选择法，而遗传算法的种群尺寸通常比较小（50～200 之间的种群大小是很常见的），分配给各个个体的后代数目可能远低于应有的值。更坏的情形是，轮盘选择法有可能将所有最优个体全部丢失。在使用轮盘选择法时应同时使用种子选拔法保证不会丢失最优个体。

5.2.1.5　随机遍及取样

随机遍及取样试图解决适应性比例选择应用于小种群上时出现的问题。SUS 不是用一个轮盘旋转几次来获取新种群。而是使用了 n 个均匀隔开的指针，指针的个数同于所需后代的个数，如图 5.4 所示，这样只需旋转一次，就可得到新种群的所有个体。

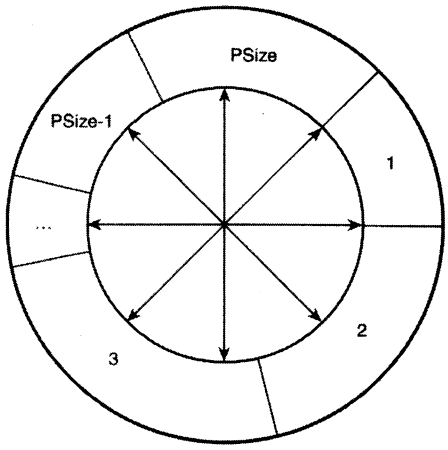


图 5.4　SUS 的概率轮盘

以下是实现该类型取样的代码：

```
void CgaTSP: :SUSSelection(vector<SGenome> &NewPop)
{
  //本算法必须要求所有适应性分数均为正值,以下的代码
  //就是用来检测和矫正相应的分数值的
  //负的适应性分将在下面即将介绍的 Sigma 定比中产生
  if (m_dWorstFitness < 0)
  {
    //重新计算
    for (int gen=0; gen<m_vecPopulation.size(); ++gen)
    {
      m_vecPopulation[gen].dFitness += fabs(m_dWorstFitness);
    }
    CalculateBestWorstAvTot ();
    }
  }
```

本章后面讨论到的一些定比技术将对某些种群个体产生负的适应性分。上面这几行代码用来检查这种可能性，并相应地调整适应性分数。如果确信适应性分数不为负值，可以略去上面的代码。

```
  int curGen = 0;
  double sum = 0;
  //NumToAdd 是用 SUS 选择法来选取的个体总数
  //注意：可能已通过精英选择法选取了其中的几个个体
  int NumToAdd = m_iPopSize - NewPop.size();
  //计算指针间隔大小
  double PointerGap = m_dTotalFitness/(double)NumToAdd;
  //为赌轮随机选择一个起始点
  float ptr = RandFloat() * PointerGap;
  while (NewPop.size() < NumToAdd)
  {
   for(sum+=m_vecPopulation[curGen].dFitness; sum>ptr; ptr+=PointerGap)
    {
      NewPop.push_back(m_vecPopulation[curGen]);
      if( NewPop.size() = NumToAdd)
      {
        return;
      }
    }
    ++curGen;
  }
}
```

如果在遗传算法里使用 SUS 选择，不推荐同时使用种子选手法，这将会导致算法冲突。当运行本章的可执行文件时，将明白打开和关闭种子选拔法对 SUS 选择的影响。

5.2.1.6 锦标赛选择

为了使用锦标赛选择，首先是在种群里随机地挑选出 n 个个体，然后将这些基因组中适应性分数最高的个体加入新种群。这一过程不断重复，直至加入的个体数目达到新种群要求的规模。任何被选中的个体并不从原种群里移走，因此有可能被重复多次选中。以下是算法的代码：

```
SGenome& CgaTSP::TournamentSelection(int N)
{
  double BestFitnessSoFar = 0;
  int ChosenOne = 0;
  //从种群中随机选择 N 个成员，与此前已发现的最好的成员进行适应性分比较
  for (int i=0;   i<N; ++i)
  {
    int ThisTry = RandInt(0, m_iPopSize-1);
    if (m_vecPopulation[ThisTry].dFitness > BestFitnessSoFar)
    {
```

```
            ChosenOne = ThisTry;
            BestFitnessSoFar = m_vecPopulation[ThisTry].dFitness;
          }
        }
        //返回冠军
        return m_vecPopulation[ChosenOne];
      }
```

本项技术使用起来非常有效，这是因为它不需要作任何的预处理，也不像赌轮选择法或其他按适应性定比的选择方法（将在本章稍后讨论）那样，往往需要对适应性分数重新进行定比。惟一的不足是锦标赛选择对某些类型的问题收敛速度过快。

这里有对该技术的另一种描述，其过程为：产生一个 0 和 1 之间的随机数。如果这个随机数小于一个预先设定的常量，比如 cT（典型取值为 0.75），那么选择适应性最高的个体作为父代之一。如果该随机数大于 cT，则选择一个较弱的个体，重复这个过程直到一个正确大小的种群完成。

趣事：NASA 以前利用遗传算法成功地计算了数条低纬度卫星轨道，最近将遗传算法用于哈勃太空望远镜的定位计算。

5.2.2　变比技术

虽然按原始（未处理的）适应性分数进行选择，能够使遗传算法开始工作（它能解决你为之特别设计的工作），但如果能在选择之前，将原始适应性分数按某种方式进行 scaling（定比，定标，变比），改变它们的评分标准，遗传算法就可以得到改进。实现变比可以采用许多种不同的方法，下面介绍其中比较优良的一些方法。

5.2.2.1　排名变比

采用按适应性分数多少的名次（rank）来改变适应性分数，可以很好地防止收敛过快的问题，特别是在运行开始时。常见的情况是：仅占总数非常小百分比的一些个体明显地优于所有其他的个体。

种群中个体的名次按适应性大小排出，再根据名次先后重新赋予一个新的适应性分数。例如，有一个大小为 5 的种群，个体的适应性分数如表 5.1 所示。那么就要将其排序，并将排名的次序，赋予个体新的适应性分数，如表 5.2 所示。

表 5.1　排名之前的适应性分数

个　体	适应性分数
1	3.4
2	6.1
3	1.2
4	26.8
5	0.7

表 5.2 排名后的适应性分数

个 体	原适应性分数	新适应性分数
4	26.8	5
2	6.1	4
1	3.4	3
3	1.2	2
5	0.7	1

当应用了按排名的名次的适应性分数之后，再使用轮盘选择法或类似的适应性比例选择方法来为下一代选择个体（在实践中不会出现只有 5 个个体的种群只是用来说明工作原理）。

该技术避免了大量后代均从极少数高适应分父代复制而来的过早收敛。排名定比能有效地保证种群中个体的多样性。这种技术虽然会使收敛速度变慢许多，但它所提供的多样性能够为遗传算法产生更好的解。

5.2.2.2 西格玛变比

如果用原始的适应性分数作为选择的基础，种群可能收敛过快；但如果像排名选择那样先行变比过，种群又可能收敛过慢。西格玛变比是一种试图在许多代的繁衍中依然保持稳定的选择压力（Selection pressure）的方法。在遗传算法开始时，适应性分数分布得很散，适应性较高的个体将以较小的期望分配到后代。在算法执行后期，当所有的适应性分数都变得大同小异，适应性较好的个体将以较大期待分配给后代。

用西格玛变比计算新适应性分数的公式是：

```
if σ = 0 then
    适应性分数 = 1
else
```

$$\text{新适应性分类} = \frac{\text{原适应性分数} - \text{平均适应性分类}}{2\sigma}$$

其中，小写希腊字母 σ（西格玛，Sigma）表示种群中个体适应性分数的标准差（standard deviation）。标准差是种群的方差（variance）的平方根。方差是一个衡量适应性分数分布宽窄的量。图 5.5 表示了一个具有较低的方差的种群。

该图水平轴上具有驼峰形突起的位置代表适应性分数的平均值（mean average）[①]。种群中绝大多数个体的分数都集中在这个驼峰附近。而适应性分数的散布范围（即方差）为这个驼峰的底的宽度。图 5.6 表示了一个具有较高方差的种群，其驼峰较矮而散布范围较为宽大。

[①] 译者注：原文误把驼峰值本身（即 Y 值）而不是峰值所在的位置（即 X 值）当作平均值；图 5.5 中的对应原文注释仍然存在如此错误。

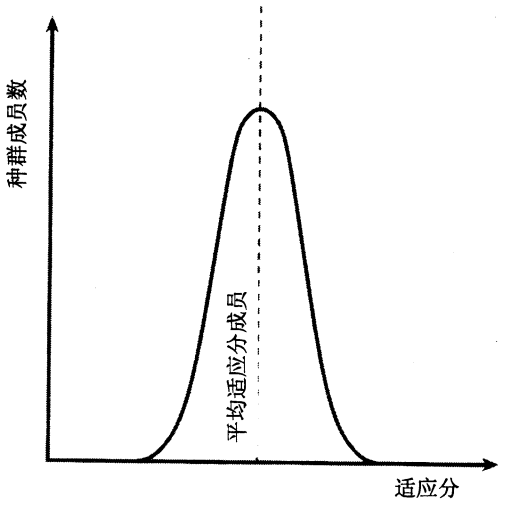


图 5.5　跨度较小的种群

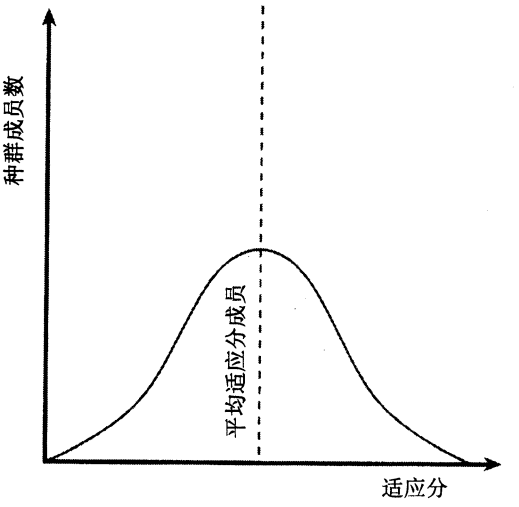


图 5.6　跨度较大的种群

下面讲述如何计算方差。假设一个种群只有 3 个个体，它们的适应性分数分别为 1、2、3。计算方差，需先计算所有适应性分数的平均值（mean）。

$$mean = \frac{1+2+3}{3} = 2$$

计算方差（variance）的公式如下：

$$方差 = \frac{(1-2)^2 + (2-2)^2 + (3-2)^2}{3} = 0.667$$

更确切的数学表示是这样的

$$方差 = \frac{\sum (f - mf)^2}{N}$$

式中，f 为当前个体的适应性分；fm 为种群的平均适应分；N 为种群大小。$\sum$ 是一个求和符号，在这里它代表将所有的 $(f - mf)^2$ 累加起来，然后除以 N 得到方差。

对方差开平方根，得到标准差：

$$\sigma = \sqrt{方差}$$

以巡回销售员问题为例，实现西格玛变比的代码如下：

```
void CgaTSP::FitnessScaleSigma(vector<SGenome> &pop)
{
  double RunningTotal  = 0;
  for (int gen=0; gen<pop.size(); ++gen)
  {
    RunningTotal += (pop[gen].dFitness - m_dAverageFitness) *
                    (pop[gen].dFitness - m_dAverageFitness);
```

```
    }
    double variance = RunningTotal/(double)m_iPopSize;
    //标准偏差即方差 variance 的平方根
    m_dSigma = sqrt(variance);
    //通过循环，为种群每一成员重算适应性分数
    for (gen=0; gen<pop.size(); ++gen)
    {
      double OldFitness = pop[gen].dFitness;
      pop[gen].dFitness = (OldFitness - m_dAverageFitness) /
                          (2 * m_dSigma);
    }
    //重新计算用于选择的值
    CalculateBestWorstAvTot ();
  }
```

最后调用的代码 CalculateBestWorstAvTot 是为对整个种群重新计算最优值、最差值和平均值，有几类选择都需要用这些值。m_dSigma 是个成员变量，使用它后当方差变成零时，可以中止算法（那时所有的适应性分数都相同，继续下去没有任何意义）。这一个函数很少能产生加速，这里写出，以便更直观地和公式相符。

西格玛变比使种群在开始几代收敛得非常快，但仍需要很长时间才能最终收敛到一个解。

有趣的事情： Quake3 游戏中的 bot（由 AI 控制的仿玩家）是用遗传算法设计的。它利用一个遗传算法为各 bot 的模糊逻辑控制器进行了优化。模糊逻辑是容许有部分真值的逻辑。用了模糊逻辑后，事物不再非真即假，也可以是灰度的。Quake3 的 bot 使用模糊逻辑判断它想做某件事情达到什么程度。它并不能确定一定要取某件特定物品，而只能确定它有 78%的可能要取枪支，有 56%的可能要取盔甲等。

5.2.2.3 波兹曼变比

使用西格玛变比使你的遗传算法在运行期间保持稳定的选择压力。但有时可能希望选择压力会发生变化。常见的情形是，在开始时需要用较小的选择压力以保证多样性得到维持；但当遗传算法开始接近收敛到解的时候，会希望只有较好的个体产生后代。

波兹曼变比使用一个连续变化的温度（temperature）来控制选择率。公式如下：

$$新适应分=e^{(原适应分/温度)} / \langle e^{(原适应分/温度)} \rangle$$

其中尖括号代表所包含项在当前一代的平均值。

在每一代中，温度（temperature）都减少一个很小的值，而这就可以导致面向较佳个体的选择压力增大。以下是 TSP 工程中的波兹曼变比的实现代码。

```
void CgaTSP::FitnessScaleBoltzmann(vector<SGenome> &pop)
{
  //逐代降低少量的温度
  m_dBoltzmannTemp -= BOLTZMANN_DT;
  //但要确保它不低于最小值
```

```
    if (m_dBoltzmannTemp< MIN_TEMP) m_dBoltzmannTemp = MIN_TEMP;
    //通过循环，寻找e^(fitness/temp)的群体平均值，并为每一个
    //个体保存一个e^(fitness/temp)记录
    vector<double> expBoltz; double average = 0.0;
    for (int gen=0; gen<pop.size(); ++gen)
    {
      expBoltz.push_back(exp(pop[gen].dFitness / m_dBoltzmannTemp));
      average += expBoltz[gen];
    }
    average /= (double)m_iPopSize;
    //现在对种群大小进行循环，计算新的期望值
    for (gen=0; gen<pop.size(); ++gen)
        pop[gen].dFitness = expBoltz[gen]/average;
    //重新计算用于选择的值
    CalculateBestWorstAvTot();
}
```

在实际解决TSP问题的程序里，temperature初始为城市个数的两倍。BOLTZMANN_DT被定义为0.05，而BOLTZMANN_MIN_TEMP被定义为1。

5.2.3 其他杂交算子

虽然，对大多数基因组编码而言，只有有限种变异算子可选择，通常还是可以采用不同的杂交操作法。下面简单介绍最普通的几种类型。

5.2.3.1 单点杂交

单点杂交仅仅将父代的基因组在随机位置切断，并交换其尾部。这种杂交算子很简单，也便于实现，对于大多数类型的问题也基本有效。

5.2.3.2 两点杂交

两点杂交需要将基因组在两个位置上断裂开来（而不是在一点断裂），并交换两点之间的基因块。因此，如果用来进行杂交的两个父代的二进制编码为：

```
Parent1: 1010001010
Parent2: 1101110101
```

而选出的杂交点是在第3个和第7个基因后，则两点杂交由下面的：

```
Parent1: 101000 1010
Parent2: 110111 0101
```

交换它们“腹部”的基因块，得到后代：

```
Parent1: 101111 0010
Parent2: 1100001101
```

图 5.7 形象地表示了这个过程。

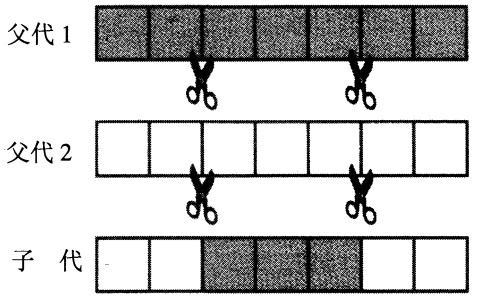


图 5.7 两点杂交

两点杂交有时很有用，因为它能创造单点杂交不能提供的基因组合。使用单点杂交的时候，尾部的基因会进行交换，这可能并不适合你要解的问题，两点杂交就排除了这个问题。

5.2.3.3 多点杂交

对于某些编码类型，当采用多个点的杂交操作时，遗传算法的性能会变得更好。为了实现多点杂交，最简单的方法就是沿着父代染色体长度，逐位逐位地按照杂交率来随机地交换基因，如图 5.8 所示。

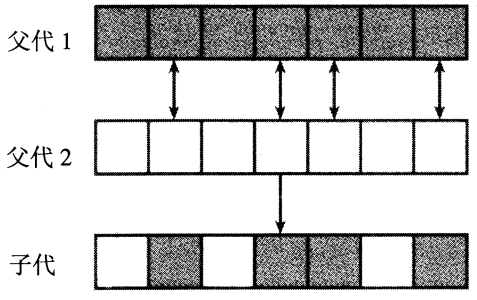


图 5.8 多点杂交

多点杂交的代码实现如下。

```
void CGenAlg::CrossoverMultiPoint(const vector<gene_type> &mum,
                                  const vector<gene_type> &dad,
                                  vector<gene_type>       &baby1,
                                  vector<gene_type>       &baby2)
{
  //对基因长度进行循环,按杂交率要求对基因进行互换
  for (int gen=0; gen<mum.size(); ++gen)
  {
    if (RandFloat() < CrossoverRate)}
    {
      //交换基因
      baby2.push_back(mum[gen]);
```

```
            baby1.push_back(dad[gen]);
        }
        else
        {
            //不需要交换基因
            baby1.push_back(mum[gen]);
            baby2.push_back(dad[gen]);
        }
    }
}
```

这类杂交有时也被称为参数归一化杂交（Parameterized Uniform Crossover）。对某些问题而言，多点杂交很有效。但使用多点杂交会过多地打乱基因，就像使用了某个过于激进的变异算子。一般地，为这种杂交算子选用的杂交率的范围在 0.5～0.8 之间。

5.2.4　子群技术

Niching 是一种通过将相似的个体划成子群以保持群体多样性的方法。最流行的 niching 技术之一是显式适应性共享（Explicit Fitness Sharing）。

通过使用该方法，首先种群中的个体被划分成子群，划分的依据是它们基因组的相似程度；接着每个个体的适应性分数都被调整，调整的方法是将自己的适应性分数和自己同属一群的成员共享。这样做就对种群里相似的个体进行了惩罚，因而保持了多样性。这里举一个通过二进制编码的基因组种群的例子，如果计算两个基因组之间的差异，只需数一下它们之间匹配的基因的位数。例如，下面这两个基因组中，有 5 位彼此相同（用粗体字符表示）。即它们的兼容性得分（Campatibility Score）为 5。为了把一个种群划分成具有相似基因组的一些子群，只需用一个样本基因组，去检测每一个基因组，从而得到它们的兼容性得分。具有相近的兼容性得分的基因组被划分到同一组，然后，用它们的原始适应性分数除以基因组子群大小来进行调整，参考表 5.3。

Genome1: 1**0100**0101**0**

Genome2: 0**0100**1010**1**

表 5.3　执行适应性共享

Genome ID	Niche ID	Raw Fitness	Adj Fitness
1	1	16	16/3=5.34
5	1	6	6/3=2
6	1	10	10/3=3.34
3	2	6	6/1=6
2	3	9	9/3=3
4	3	10	10/3=3.34
8	3	20	20/3=6.67
9	4	3	3/2=1.5
10	4	6	6/2=3

适应性共享能够有效地抑制大同小异的基因组的生成，也是使种群保持多样性的极好的方法。

5.3 总　　结

接下来的几章里，会经常使用遗传算法各种技术的一些最简组合来寻找期望的解。可以尝试和实践这些技术，理解它们对于不同类型的问题是会改进或且是妨碍进化。

5.4 练　　习

1．试将在本章学到的内容应用于第三章“遗传算法入门”里的迷宫寻路问题。

2．如何设计一个解决 8 数码问题的遗传算法？

8 数码问题就是将随机放在一个方框里的 8 个数码小块（如图 5.9 所示），通过一次滑动一个小块的办法，排列成 12345678 的有序状态。

3．设计一个遗传算法来计算 Boggle 拼字游戏盘上能获得最高分的字母组合。

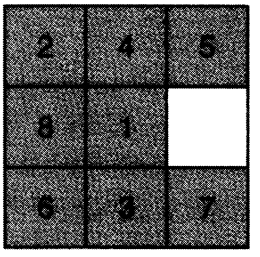


图 5.9　一个待解的 8 数码问题

第6章　登月也不难

在20世纪60年代的太空竞赛中，美国宇航局（NASA）决心开发一种能在太空舱中失重的情况下正常书写的圆珠笔。

经过大量的探索和研究，花掉了百万美元的科研经费，终于开发出了一种宇航笔（Astronaut Pen）。

这支笔确实能用了。当它返回地面后也作为一桩新鲜事物而获得不少成功。

而前苏联宇航员在遇到同样难题的时候，使用了铅笔。

在本章里，将学习如何用遗传算法对行为模式（Behavior Patterns）进行编码和演化。这种编码可以对许多游戏中的对象进行行为演化，从街机游戏中怪兽的动作，直到运动赛车的航行路线。有很多场合都能运用此类遗传算法。并且，利用此种方法在实际游戏中实现行为演化所需要的处理器功率很小。

下面的例子是为引导一个登月器、登月飞船（lunar lander）平稳地降落到一个小的着陆平台上，演化一个控制模式的序列。如图6.1所示。

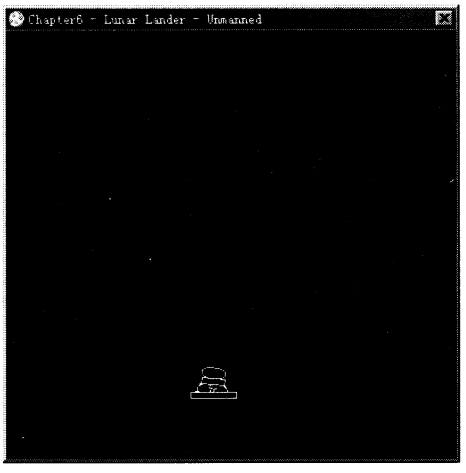


图6.1　登月飞船的一次成功着陆

读者在讨论有关的遗传算法之前，先解释一下本例或今后的例子中将要用到的（计算机）图形学技术，以及与此相关的物理和数学方法。

如果读者已经掌握矩阵、变换、矢量、牛顿物理学，可以跳过这些内容。

6.1　创建和处理矢量图形

学习图形学和有关数学的最好办法，就是一步一步地开发一个由用户（人工）控制的

登月器游戏。这样不但可以熟悉需要的技术，并且可以在学习如何编写用遗传算法来控制的登月器登陆之前，了解月球登陆问题的难度究竟有多大。

此游戏需要用到的 2D 图形是很简单的。需要一个登月器对象，一个着陆平台对象，还有一片闪烁的星空。下面介绍如何创建代表形状、形（2D shape）的数据结构。

6.1.1 顶点和顶点缓冲

Shape 是由平面（2D 空间）中一系列相互连接的点来定义的。2D 空间中的点可以通过它在 x 轴和 y 轴上的（投影）位置来表示，如图 6.2 所示。

点（Point），在计算机图形学中，一般称为顶点（Vertex）。要创建一个 shape，只需要在某种数据结构里存储所有组成该 shape 的顶点。这里举一个极简单的 shape 作为例子，如图 6.3 所示。这和当年原始版的太空入侵者（Space Invader）游戏中的 gun（枪，炮）的样子类似。

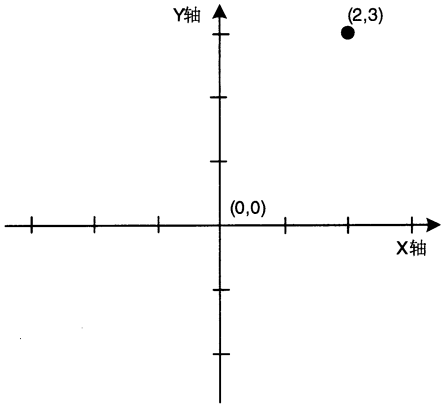


图 6.2 2D 坐标系

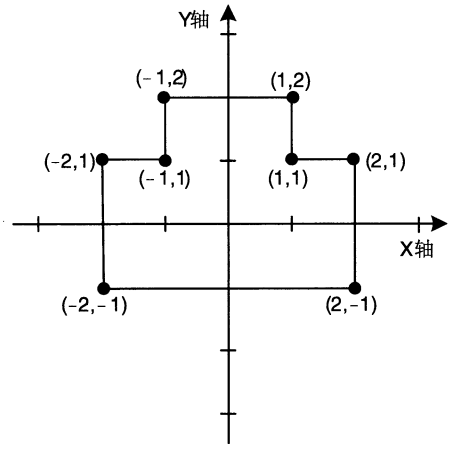


图 6.3 定义一个简单的 shape

这里将 shape 的中心放在原点。在代码中用来存放顶点的数据结构称为 SPoint，具体定义如下：

```
struct Spoint
{
 double x, y;
 SPoint (double a = 0, double b = 0):x(a),y(b){}
};
```

要保存整个 shape，只需创建一个 SPoint 数组，保存组成该 shape 的所有顶点的集合。在程序中均使用矢量 std::vector 来储存 shape。这些顶点矢量又称顶点缓冲器（Vertex Buffer），为了简单起见，用 VB 后缀来命名它们。例如，对于图 6.3 这样的 shape，定义和初始化顶点缓冲的代码如下：

```
vector<SPoint> vecGunVB;
const int NumGunVerts = 8;
const SPoint gun[NumGunVerts] = {SPoint(2,1),
                                 SPoint(2,-1),
                                 SPoint(-2,-1),
                                 SPoint(-2,1),
                                 SPoint(-1,1),
                                 SPoint(-1,2),
                                 SPoint(1,2),
                                 SPoint(1,1)};
for (int i=0; i<NumGunVerts; ++i)
  {
    vecGunVB.push_back(gun[i]);
  }
```

为了绘制这个 shape，只需写一个函数，将所有这些顶点，按照上面给出的顺序，从头到尾，依次用线段连接起来，最后一个顶点应与第一个顶点相连。

一般的做法是从数据文件中载入游戏需要用到的所有顶点数据。本例并没有用到很多的图形对象，因而我将所有的顶点定义为常量（const）数组，然后在类的构造函数里用它们初始化顶点缓冲。可以参考本书配套光盘里 Chapter6/Lunar Lander - Manned 目录下的 CLander.cpp 文件，从而理解如何初始化登月器 shape 和着陆平台 shape 的顶点缓冲器。

定义一个 shape 后，应在屏幕的正确的位置上以正确的方向绘制它。如果按定义的数据直接在屏幕上画出这一 shape，会看到一个如图 6.4 所示的形状。

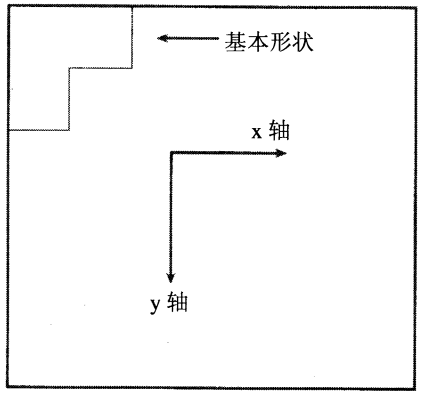


图 6.4　按定义数据画出的 gun

这是因为 gun 的原始形状数据是以原点为中心的。其中有许多顶点坐标的值是负的，这一部分的顶点在 Windows 的默认绘图模式是不支持的。另外，默认绘图模式下窗口的 y 轴也与通常的坐标系上下颠倒，因此画出的图形头朝下。

6.1.2　顶点变换

需要一种办法来调整 shape 的顶点，使其出现在正确的位置，并具有正确的方向和比

例大小。这就需要利用一系列的坐标变换（Transformation）了。需先了解一些关于游戏对象的细节。为了计算如何正确地在屏幕上放置一个对象，需要预先知道它在屏幕坐标系中的中心坐标位置（Position）、旋转角度（Rotation）以及放大比例（Scale）。因此，一个简单的游戏对象的数据结构代码如下：

```
struct GameObject
{
  double dPosX, dPosY;              //游戏对象中心的x,y坐标
  double dRotation;                 //游戏对象相对于y轴的偏转角度
  double dScale;                    //游戏对象需要的放大比例
  vector<SPoint> vecShapeVB;        //游戏对象的顶点数据
}
```

接下来，必须计算顶点缓冲区中每个顶点的新位置，确保在屏幕以正确的方式画出游戏对象。可以利用一系列的变换来实现。

6.1.2.1 平移

平移就是将一个或一组点从一个地方移动到另一个地方的过程。仍然用 gun 的例子，假如 gun 在游戏中出现的位置是坐标(5,6)，即 gun 的中心位置应该在(5,6)。

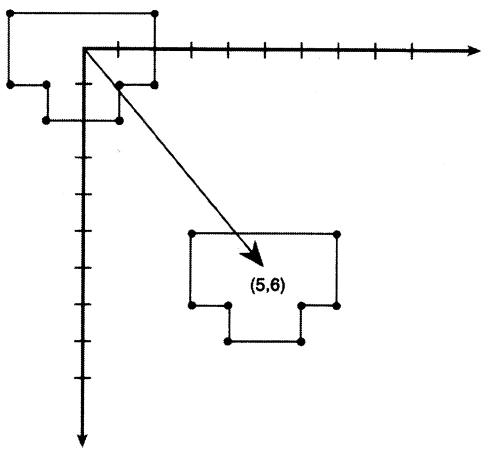


图 6.5 平移 shape

需要找到一个方法来调整 gun 的顶点缓冲中的顶点，使得在绘制时，gun 会出现在屏幕的正确位置，即中心在(5,6)，而非原点(0,0)。为此，只需写一个函数，为每个顶点的 x 坐标加 5，y 坐标加 6，各个顶点就会相应地变换到正确的屏幕位置，如图 6.5 所示（注意由于 y 轴颠倒，shape 依然是头朝下的）。

如果顶点缓冲区中每个顶点的坐标用 vertX 和 vertY 表示，这些顶点所描写的对象在游戏中的位置是 posX 和 posY，则顶点在游戏中的坐标为：

```
ScreenX = vertX + posX;
ScreenY = vertY + posY;
```

因此，要移动整个对象，只要把顶点缓冲区中的每一个顶点进行这样的平移操作，即可达到正确定位的目的。

注意：一个对象在变换前的原始坐标，称为对象的本地坐标（Local Coordinate），而它在被变换之后的坐标——即它将在游戏中出现的坐标——称为对象的世界坐标（WorldCoordinate）。

6.1.2.2　变比

变比，或比例缩放，是使对象的尺寸放大或者缩小的过程。只需将对象的每个本地坐标乘以一个比例系数即可实现。在图 6.6 中，对象 gun 被放大了两倍。

也可以在不同的方向采用不同的比例系数。例如，如需在 x 轴放大 sx 倍，y 轴放大 sy 倍，则公式为：

```
newX = x × sx
newY = y × sy
```

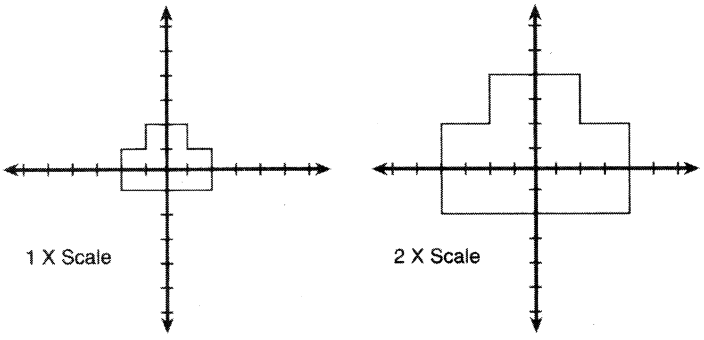


图 6.6　执行变比，使图形放大为 2 倍

6.1.2.3　旋转

旋转是较难理解的变换，因为需要用到三角知识。

在计算机的函数中使用的角度单位通常为弧度（Radian），一个圆是 360°，即 2π 弧度，π=3.14159。

假设有一点 p1，坐标为(5,3)。需要将它旋转 30°，变换到一个新的坐标 p2，如图 6.7 所示。

将某个点绕原点旋转 theta 角度的公式为：

```
newX = oldX × cos(theta) - oldY × sin(theta)
newY = oldX × sin(theta) + oldY × cos(theta)
```

将 30° 换算成弧度。每一度等于 2π/360 弧度——或 0.017453 弧度，因此

$$30° = 30 \times 0.017453 = 0.52359 \text{ 弧度}$$

将数字代入公式得到：

```
newX = 5 × cos(0.52359) - 3 × sin(0.52359) = 2.830
newY = 5 × sin(0.52359) + 3 × cos(0.52359) = 5.098
```

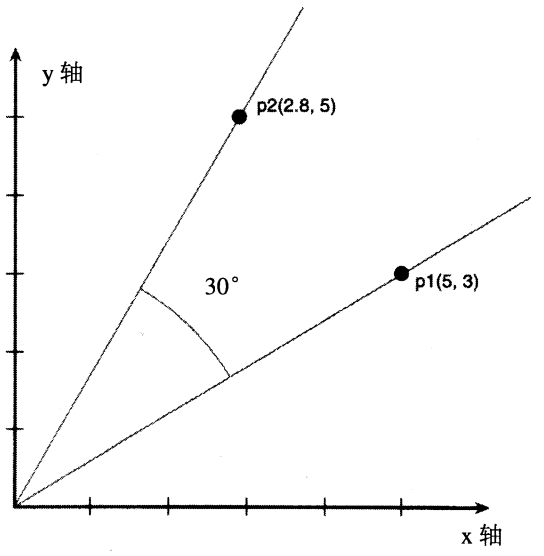


图 6.7　点的旋转

旋转的结果如图 6.7 所示。将某一个点旋转一个正的角度后，这个点将绕原点逆时针（Anticlockwise）方向进行旋转（y 轴朝上 x 轴向右）。

但在一个 y 轴朝下而 x 轴朝右的环境（如一个通常的应用程序窗口那样）中使用这条公式，旋转就应顺时针进行。如果仍需要朝相反方向旋转，可以将公式中的 sin 项符号取反，如下：

```
newX = oldX × cos(theta) + oldY × sin(theta)
newY = (-1) × oldX × sin(theta) + oldY × cos(theta)
```

或更简单些，仍然使用原先的公式，但使用负的旋转角度：

```
newX = oldX × cos(-theta) - oldY × sin(-theta)
newY = oldX × sin(-theta) + oldY × cos(-theta)
```

注意：在实际中，将角度在度和弧度单位之间来回换算对于一个游戏来说计算量过大了。只使用弧度是比较好的选择。

6.1.2.4　综合运用

对顶点进行变换后，现在只需要将 3 种变换整合成一个函数，然后在向屏幕上绘画之前调用该函数。该函数通常称为世界变换（World Transformation）函数，因为它将 shape 的顶点从本地坐标换算成世界坐标。对太空枪 shape 使用 3 种变换的，代码可在光盘上的 Chapter6/ShapeManipulation 目录下找到。可以用键盘上的方向键来改变旋转角度和缩放比例，使用 A、S、P、L 键可以使 shape 在屏幕上移动。

CGun 类的定义及 CGun::WorldTransform 函数的形式如下：

```
class Cgun
{
```

```
public:
    //Gun在世界（坐标系）中的位置
    double       m_dPosX,
                 m_dPosY;
    //它的转角
    double        m_dRotation;
    //它的大小
    double        m_dScale;
    //gun的本地坐标的顶点缓冲区
    vector<SPoint> m_vecGunVB;
    //用于保存被变换后的gun的顶点缓冲区
    vector<SPoint> m_vecGunVBTrans;
    CGun (double x,
          double y,
          double scale,
          double rot);
    void WorldTransform();
    void Render(HDC &surface);
    void Update();
};
void CGun::WorldTransform()
{
  //把所有原始顶点复制到另一个 buffer 中,以备实行各种变换
  m_vecGunVBTrans = m_vecGunVB;
  //旋转各顶点
  for (int vtx=0; vtx<m_vecGunVBTrans.size(); ++vtx)
  {
    m_vecGunVBTrans[vtx].x = m_vecGunVB[vtx].x * cos(m_dRotation)-
                             m_vecGunVB[vtx].y * sin(m_dRotation);
    m_vecGunVBTrans[vtx].y = m_vecGunVB[vtx].x * sin(m_dRotation)+
                             m_vecGunVB[vtx].y * cos(m_dRotation);
  }
  //变比,即按统一比例改变顶点位置
  for (vtx=0; vtx<m_vecGunVBTrans.size(); ++vtx)
  {
    m_vecGunVBTrans[vtx].x *= m_dScale;
    m_vecGunVBTrans[vtx].y *= m_dScale;
  }
  //对顶点进行位置的移动
  for (vtx=0; vtx<m_vecGunVBTrans.size(); ++vtx)
  {
    m_vecGunVBTrans[vtx].x += m_dPosX;
    m_vecGunVBTrans[vtx].y += m_dPosY;
  }
}
```

应注意变换的执行顺序非常重要。如果 shape 在旋转之前先进行了平移，那么下一步就是将平移后的 shape 围绕原点进行旋转。这就是为什么应该把游戏对象的中点放在原点的原因。只有这样旋转和变比才能正确实现。读者可以尝试修改一下程序，把其中的变换执行顺序改变一下，看看将产生什么后果。

而且，gun 对象需要有两个顶点缓冲区：一个保存变换前的原始顶点，另一个保存变换后的可以直接用到屏幕绘图的顶点。应有一份原始顶点用来工作。

6.1.3 矩阵

对每一个顶点，你的程序要执行 3 种计算实现对每个顶点旋转、变化和平移这 3 种不同的变换。矩阵可以将所有的变换组合成一个，而且这个矩阵适用于所有的顶点。当需要每帧变换成百上千个顶点时，使用矩阵能够节省很多的处理器时间。

6.1.3.1 什么是矩阵

矩阵是数字的阵列。它可以是一维、两维甚至是多维的，就像程序里的数组一样。变换矩阵（Transformation Matrix）总是二维（2D）的。下面是一个 2D 矩阵的例子：

$$\begin{bmatrix} 4 & 15 & 0 \\ 3 & 3 & 1 \\ 0 & 8 & 13 \end{bmatrix}$$

这个矩阵有 3 行（Row）和 3 列（Column），因此是一个 3×3 矩阵。矩阵中的每个数字都称为一个元素（Element）。例如，数字 15 是这个矩阵中的位置（1，2）上的元素。

注意：数组的下标通常从零开始，而矩阵下标是从 1 开始的。经常令人混淆。

可以对矩阵进行加法、减法、除法和乘法，但对于矩阵变换，只需要进行乘法。

6.1.3.2 矩阵乘法

举例解释如何将一行乘以一列，如下：

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = (1\times 7)+(2\times 8)+(3\times 9) = 7+16+27 = 50$$

行的元素个数和列的元素个数必须相等。如果它们不相等，就无法相乘。

矩阵 A 和矩阵 B 相乘，即将 A 中的每一行和 B 中的每一列相乘。例如：

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix} = \begin{Bmatrix} (1\times 6)+(2\times 8) & (1\times 7)+(2\times 9) \\ (3\times 6)+(4\times 8) & (3\times 7)+(4\times 9) \end{Bmatrix} = \begin{bmatrix} 22 & 25 \\ 50 & 57 \end{bmatrix}$$

计算矩阵 A×B，A 的列数必须等于 B 的行数。因此可以将下面这两个矩阵相乘：

$$[1 \quad 2 \quad 3 \quad 4]\times\begin{bmatrix}5 & 9\\ 6 & 10\\ 7 & 11\\ 8 & 12\end{bmatrix}$$

但下面的乘法就不能进行：

$$[1 \quad 2 \quad 3 \quad 4]\times\begin{bmatrix}5 & 8\\ 6 & 9\\ 7 & 10\end{bmatrix}$$

试计算前面一个乘法（答案见本章最后）。

6.1.3.3　单位矩阵

数字 1 是非常有用的。因为它是一个单位，用 1 乘以任何数字结果仍然是那个数字本身。在矩阵中也存在与数字 1 相似的东西，即单位矩阵（Identity Matrix）。矩阵的大小可以是任意的，若使用 3×3 矩阵，给出 3×3 的单位矩阵如下：

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

如果将单位矩阵乘以任何矩阵，结果还是矩阵本身。

$$\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\times[4 \quad 5 \quad 6]=[(1\times4)+(0\times4)+(0\times4)(0\times5)+(1\times5)+(0\times5)(0\times6)+(0\times6)+(0\times6)]$$

$$=[4 \quad 5 \quad 6]$$

需要使用单位矩阵作为创建其他矩阵的基础。

6.1.3.4　用矩阵变换顶点

重要性质：空间中的一点（例如 x,y）可以表示为一个矩阵

$$[x \quad y \quad 1]$$

第 3 个元素必须设置为 1。

将点表示为矩阵后，就可以将它与其他矩阵相乘来实行变换。下面讲述如何利用矩阵实行 3 种基本变换。

1．平移

对点进行平移，应在点的 x,y 坐标上加上要移动的距离 dx 和 dy，如：

$$newX = x + dx$$

$$newY = y + \mathrm{d}y$$

要利用矩阵达到同样的效果，可以设计一个平移矩阵，如下：

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mathrm{d}x & \mathrm{d}y & 1 \end{bmatrix}$$

然后乘以点(*x*,*y*)的矩阵（*x*,*y*,1）。计算过程如下：

$$\begin{aligned} &[x \quad y \quad 1] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mathrm{d}x & \mathrm{d}y & 1 \end{bmatrix} \\ &= [(1\times x)+(0\times y)+(1\times \mathrm{d}x) \quad (0\times x)+(1\times y)+(1\times \mathrm{d}y) \quad (0\times 0)+(0\times 0)+(1\times 1)] \\ &= [x+\mathrm{d}x \quad y+\mathrm{d}y \quad 1] \end{aligned}$$

矩阵（*x*,*y*,1）中的 1 使得平移的量 d*x*,d*y* 能被加到最终的和数上。

2．变比

下面的矩阵为一个变比矩阵，它使 *x* 轴放大 *sx* 倍，*y* 轴放大 *sy* 倍。

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

前面是这样变比一个点的：

```
newX = x × sx
newY = y × sy
```

对点（*x*,*y*）使用上面的变比矩阵，得到的结果将一样：

$$[x \quad y \quad 1] \times \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x\times sx \quad y\times sy \quad 1]$$

3．旋转

为了用矩阵进行旋转变换，需要设计一个矩阵来重现此前得出的旋转公式：

```
newX = oldX × cos(theta) - oldY × sin(theta)
newY = oldX × sin(theta) + oldY × cos(theta)
```

完成这一功能的矩阵如下：

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

当用该矩阵来变换一个点（x,y）时，能够正确地得到旋转后的点：

$$\begin{bmatrix} x & y & 1x \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta & x\sin\theta + y\cos\theta & 1 \end{bmatrix}$$

6.1.3.5　变换矩阵

矩阵有这样的重要性质：如果要进行一系列的变换，可以为每个变换创建一个矩阵，然后将所有这些矩阵组合成为一个单个的变换矩阵。

要组合矩阵，方法就是将它们相乘。如前所述，乘法的执行顺序是极为重要的。下面就是创建和使用变换矩阵的步骤：

1．产生一个单位矩阵。

2．产生一个变比矩阵，将其与第 1 步中生成的矩阵相乘。

3．产生一个旋转矩阵，将其与第 2 步中生成的矩阵相乘。

4．产生一个平移矩阵，将其与第 3 步中生成的矩阵相乘。

5．将第 4 步中产生的矩阵运用在 shape 的各个顶点上。

其中第 3 步的旋转和第 2 步的变比的顺序可以互换。但若其他步骤的顺序错误，结果不可想象。

可以用矩阵变换修改上一个例子。代码可以在 Chapter6/Shape Manipulation with Matrices 目录下找到。如果在编译器中运行这个程序，会发现在程序中定义了一个矩阵类 C2DMatrix 来执行矩阵运算。CGun::WorldTransformation 函数定义如下：

```
void CGun::WorldTransform()
{
  //将原始顶点复制到要执行变换的缓冲区
  m_vecGunVBTrans = m_vecGunVB;
  //创建一个变换矩阵
  C2DMatrix matTransform;
```

当创建 C2DMatrix 类的一个实例时，它被初始化为单位矩阵，这是生成变换矩阵的第一个步骤，是生成更复杂的变换矩阵的一个基础。

```
  //变比
  matTransform.Scale(m_dScale, m_dScale);
```

可以用 Scale 方法来变比一个对象，该方法的两个参数分别是 x 和 y 方向的比例因子。通常它们不一定相等，因为对象 gun 在两个轴上即使用了同样的比例因子，这对两个参数均使用同一个比例因子 m_dScale。

```
  //旋转
  matTransform.Rotate(m_dRotation);
  //平移
  matTransform.Scale(m_dPosX, m_dPosY);
```

```
//改变这个武器（gun）或飞船（ship）的顶点
matTransform.TransformSPoints (m_vecGunVBTrans);
```

上面的 TransformPoints 以引用方式传入一个顶点矢量，并乘以综合所得的变换矩阵。

由此可见，采用矩阵方法不仅使用方便，而且矩阵类的运算速度也比本章开始介绍的 Shape Manipulation 代码简单和快得多。

6.2 矢　　量

讲述矢量之前，先看一下什么是一个点。图 6.8 表现了一个点，在 2D 空间中以坐标(3,4)表示，它只是 2 维空间里的一个位置。一个 vector（矢量、向量）可以提供更多的信息，矢量表示了数量大小（Magnitude）和方向（Direction）。例如，点 *P*(3,4)代表了一个如图 6.9 所示的矢量。

比如说编写一个红色警报（Red Alert）类型的游戏，其中有一辆坦克在地图上行驶。它的速度和方向就可以表示为一个矢量。矢量的方向就是坦克的朝向，也就是坦克的前进方向；而矢量的大小（即长度）则表示坦克前进的速度。矢量越长，坦克行驶得越快。

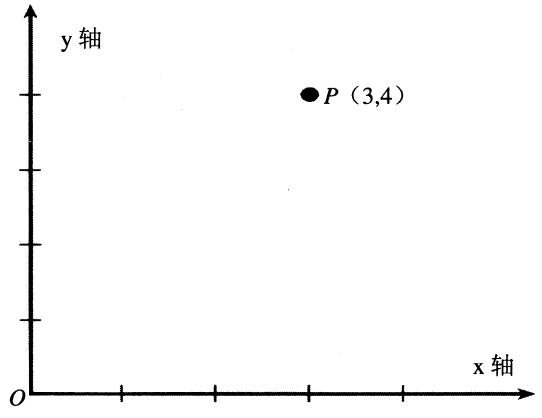


图 6.8　2D 空间中的点

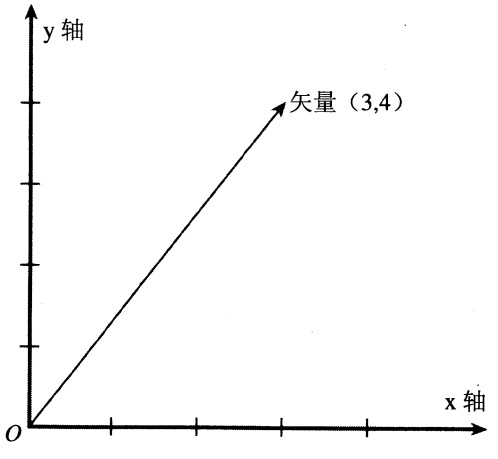


图 6.9　2D 空间中的矢量

二维矢量用 *x* 和 *y* 两个坐标表示，就与 2D 空间中的点类似。矢量的数据结构的定义也和先前提到的 SPoint 结构非常相似：

```
struct Svector2D
 {
  double x,y;
  Svector2D(double a=0.0f, double b=0.0f:x(a),y(b)){}
 }
```

如果需要在 3D 空间中编程，必须加入第 3 个分量的大小，即 *z* 坐标值来表示矢量。只要利用空间中的一个点，就可以确定一个矢量，因为矢量的另一个端点（起始点）均设置

在坐标系的原点(0,0)或(0,0,0)。

6.2.1　矢量加、减法

假如有一张藏宝图上面写了如何可以到达藏宝的地点，但它所用的语言和通常的描述不同：“从棕榈树往东走 3 步，然后朝东北走 6 步，再朝西北走 12 步……”，上面写着这样的指令：

要想找到财宝，从棕榈树开始沿下列矢量行走：(3,1), (-2,4), (6,-2), (-2,4)。

图 6.10 所示的地图标出了一条路线，按照这条路线走，就能找到财宝。

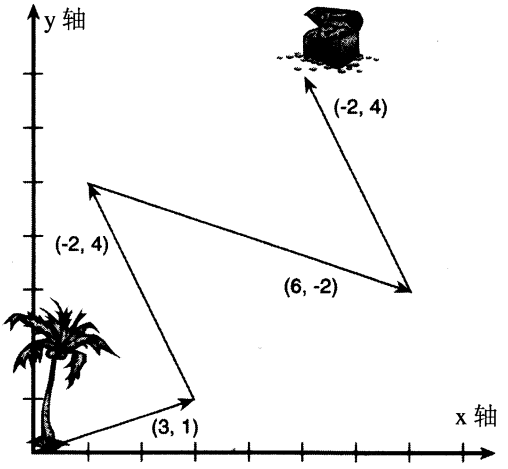


图 6.10　使用矢量表示的藏宝图

矢量很大的一个优点就是，可以把图中 4 个单独的矢量加在一起得到一个矢量，只要沿这一矢量走，就能直接找到财宝。要把矢量相加，就是把 x 和 y 分量各自相加。因此，新的矢量是：

New x = 3 + (-2) + 6 + (-2) = 5

New y = 1 + 4 + (-2) + 4 = 7

这样，所有矢量已经加在一起，得到了新矢量(5,7)，如图 6.11 所示，这一矢量直接指向了财宝储藏地。

矢量相减，是把分量各自相减而不是相加。

有趣的事实：有一种居住在沙漠洞穴里的蚂蚁，它们走起路来是这样的：先随机选一个方向，然后直线走一段时间，然后再选另一个方向走，周而复始直到找到食物为止。这时，它离开洞穴的距离保护（对于蚂蚁来说）已经很远了——有时甚至能达到数百英尺之远。

不可思议的是，当蚂蚁一旦找到食物，它将会沿一条直线直接返回地洞。可见，这聪明的小昆虫已把它所走过的各个矢量加起来成为一个矢量，然后将此总和矢量求反，找出回家的路。

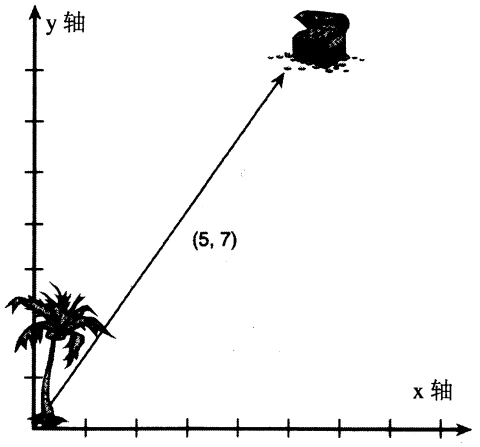


图 6.11　运用矢量加法

6.2.2　计算矢量大小

找到了财宝后，但应计算财宝箱离棕榈树有多远。为此，需要计算矢量的模（Magnitude），也就是矢量的长度。

矢量的大小幅度可通过著名的毕达哥拉斯定理来计算。如图 6.12 所示，*AB* 长为：

$$AB = \sqrt{x^2 + y^2}$$

代入数字得出：

$$AB = \sqrt{5^2 + 7^2} = 8.6023$$

最后，计算得出从棕榈树到财宝箱的距离为 8.6023 单位。

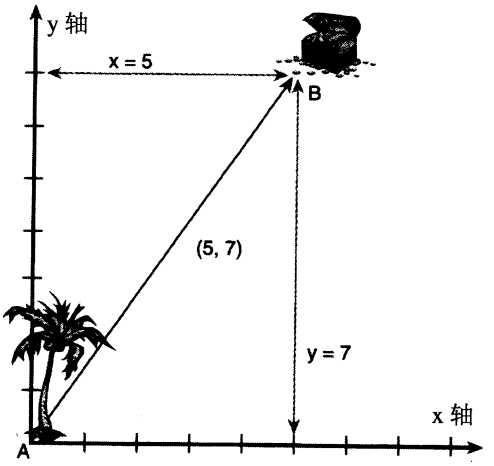


图 6.12　计算矢量的大小

6.2.3　矢量的数乘

将矢量与一个数相乘，即将此矢量的每一个分量各自乘以这一个数。例如，矢量(1,2)乘以 4 为：

$$(1,2)\times 4=(4,8)$$

计算得到的矢量与计算前的矢量方向相同，但大小是原来的 4 倍，如图 6.13 所示。

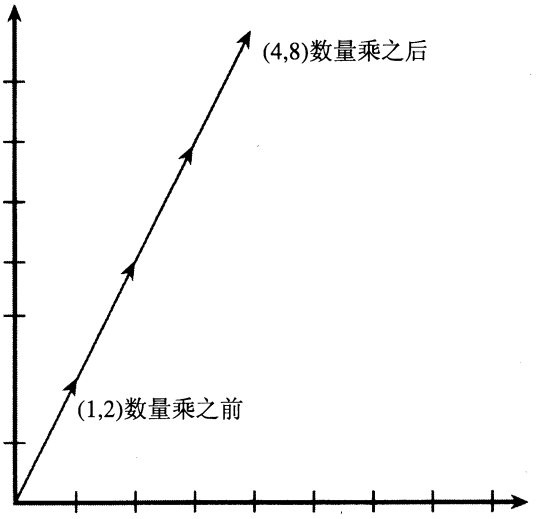


图 6.13　矢量的数量乘法

6.2.4　矢量的规范化

一个规范化矢量（Normalized vector，或标准化矢量）即大小为 1.0 的矢量，也称为单位（Unit）矢量。若将矢量规范化，只需将矢量的所有分量除以矢量的长度。这样得到的矢量方向保持不变，但长度变为 1.0。

例如，矢量 ***V***=(3,4)。其幅度或长度为：

$$Length=\sqrt{3^2+4^2}=\sqrt{25}=5$$

既然已经求出了长度，用它除矢量的各分量：

$$3/5=0.6$$
$$4/5=0.8$$

因此，规格化矢量 ***N***=(0.6,0.8)。

技巧：矢量通常为粗体大写字母，比如 ***V***。矢量的长度为|***V***|。因此，规格化矢量可以表示为：$\boldsymbol{N}=\boldsymbol{V}/|\boldsymbol{V}|$。

规格化矢量有时也用小写字母表示，或在大写字母后加星号，例如 $\boldsymbol{V}$ 的规范化矢量可以记为 $\boldsymbol{v}$ 或 $\boldsymbol{V}^*$。

6.2.5 矢量分解

矢量的一个有价值的操作就是可以将矢量分解（Resolve）为分量 x 和 y。图 6.14 中有一辆小车，假设这辆车以 50 千米的时速沿图中的方向行驶。

可以将速度矢量分解为两个单独的矢量，分别代表汽车沿 x 和 y 方向的速度。这两个矢量即为速度矢量的 x 和 y 分量。利用三角知识，可知直角三角形满足下列公式：

$$\cos\delta = \text{邻边} / \text{斜边}$$
$$\sin\delta = \text{对边} / \text{斜边}$$

图中角度为 30°，斜边为矢量的模（汽车的速度）50 kph（千米每小时），求解 x 和 y 分量，即为求解邻边和对边。计算如下：

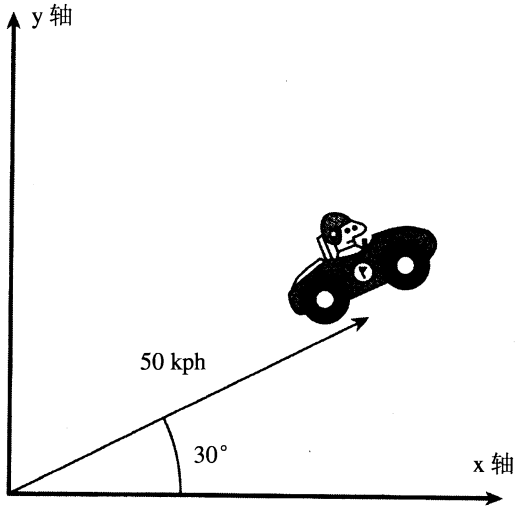


图 6.14 速度矢量

$$x\text{分量} = 50 \times \cos 30° = 43.3\text{kph}$$
$$y\text{分量} = 50 \times \sin 30° = 25\text{kph}$$

求得这辆车每小时沿 y 轴行驶 25 千米，沿 x 轴行驶 43.3 千米。矢量分解的这种用法在很多时候都非常有用。

假设在设计坦克大战游戏中，用户控制的坦克的位置用 SVector2D 结构表示，其旋转角度用一个双精度实数表示：

```
SVector2D vTankPos;
double    dTankRotation;
```

当用户踩下油门，编程员如果希望坦克以 10kph（每小时 10 千米）的速度向正前方开

去。应将 10kph 的速度分解为 x 和 y 分量，然后将对应的分量加到坦克的位置，就可得出坦克下一个小时的位置。代码如下：

```
double xComponent = 10*cos(dTankRotation);
double yComponent = 10*sin(dTankRotation);
vTankPos.x += xComponent;
vTankPos.y += yComponent;
```

此后的代码里会频繁使用这种方法。

6.2.6　矢量的点积

两个矢量的点积（Dot product）有许多优点，在游戏编程中会频繁使用。给定两个矢量 $\boldsymbol{U}$ 和 V，它们的点积的计算公式如下：

$$\boldsymbol{U} \cdot \boldsymbol{V} = \boldsymbol{U}x\boldsymbol{V}x + \boldsymbol{U}y\boldsymbol{V}y$$

或

$$\boldsymbol{U} \cdot \boldsymbol{V} = |\boldsymbol{U}||\boldsymbol{V}|\cos\theta$$

若参与点积的两个矢量均为规格化矢量，第二个等式就简化为：

$$\boldsymbol{U} \cdot \boldsymbol{V} = \cos\theta$$

将等式 1 代入等式 2，就得到了下面的公式：

$$\boldsymbol{U}x\boldsymbol{V}x + \boldsymbol{U}y\boldsymbol{V}y = \cos\theta$$

从而求得两个矢量的夹角 θ。

6.2.7　SVector2D 实用工具类

为了方便 2D 矢量运算，相关代码参见 SVector2D.h，此后所有的代码项目里几乎都包含该头文件。在此头文件中重载了运算符，以便对矢量进行乘法、除法、加法和减法。此外也有一些很有用的矢量运算，例如：

```
inline double Vec2Dlength(const Svector2D &v);
inline void Vec2Dnormalize(Svector2D &v);
inline double Vec2DDot(Svector2D &v1,  Svector2D &v2);
inline int Vec2DSign (Svector2D &v1,  Svector2D &v2);
```

最后一个函数 Vec2DSign 当 v1 按顺时针方向转到 v2 时返回 1，若逆时针则返回-1。这在实际应用中很有用。

SVector2D 中还定义了 SPoint 结构。通常对顶点使用 SPoint，对速度和对象位置则使用 SVector2D。

6.3 相关的物理知识

随着游戏的日益复杂和生成游戏所用的场景越来越真实化，对程序员的要求也越来越高了。程序员不仅要会编写代码，还需要掌握有关人工智能方法、数学、3D 图形学中诸如 BSP 树（Binary Space Partition tree，二叉空间树）的数据结构和快速算法、纹理映射及处理、以及快速碰撞检测等技术。但如果在程序中不按照物理学的基本规律来做，即使用尽上面所有的技术，也无法让游戏具有真实感。

物理学是关于物质和能量及其关系的科学。

物理学定律刻画了人们生活的宇宙中万事万物的行为和特性，学习物理就是要认识周围的世界。

当程序员学会如何将物理写入游戏世界后，游戏世界不但有静态的形象真实，而且也开始会有感觉的真实。物体的运动是按它应有规律运动，车辆在转弯角上滑动，子弹和导弹则在空气中划出优美的弹道，烟雾粒子冉冉上升并像真的烟雾一样产生涡旋，物体有轻重不同的感觉。

本章中的登月飞船是运动的，并且它是受引力（重力）作用的，只需考虑运动和重力的物理学。下面介绍一些用来刻画物理对象和物理行为的一些基本单位的定义。

6.3.1 时间

时间的单位是秒（Second），在编写游戏程序时，通常会用一秒钟的一小部分作为单位来工作，因为代码在每一帧中都要进行相应的更新计算，而大多数游戏至少要运行在每秒 30 帧以上。

有趣的事实：在人们发明铁路之前（这还不是很久），许多国家，甚至是许多村镇，都有自己独特的时间单位。在伦敦对好了表，但到了牛津，时间就差了好几分钟。这种事情让人感到很奇怪——人们旅行时每走几英里就必须调一下手表——但纵然如此，以前的世界就是这个样子。

6.3.2 长度

长度的标准单位是米（Meter）。

在设计游戏的时候，距离通常是使用任意的单位度量的，因为这与图形显示设备的分辨率有关。

6.3.3 质量

质量是度量某个东西包含物质的多少的物理量。由于一个物体必须通过测量它的重量（weight）才能算出它的质量，而重量又不是质量的一个单位，所以质量的正确测量十分

困难。质量是物质（matter）的一个单位。而重量则是重力（gravity）作用在某个东西上产生多少大小的力（force）的度量。

重力作用于放在笔者面前的镇纸的石头在不同场所是不同的。例如，放到月亮上它将很轻。由于地球不是一个精确的球，地层或山脉岩石在各处又会有不同的密度，故在地球上的不同地点的镇纸石也将有不同的重量。但无论镇纸石放在哪里，它的质量不变，总是有相同的数值。

质量的标准是放在法国巴黎的一个由铂铱合金制作而成的金属圆柱体（cylinder）。物理学家们已同意把此圆柱体的质量定义为 1 公斤（或千克，kilogram）。任何别的东西都得以此标准来进行度量。

6.3.4　力

牛顿为力所下的定义：

力是施加于一个物体（body）使其改变状态——或者静止，或者形成等速直线运动——的一个行为（action）。

力是改变物体的运动速度（speed）或运动路线（line）的某种事物。某些力是极为明显的，例如，一个足球运动员用来踢球的力、烤面包机弹出烤好面包的力、使用电梯时所感受到的力等。但另外有一些力就没那么明显，因为它们可能相互作用但又相互抵消，尽管力正在作用着，但物体并不移动。例如，重力施加在放在一张桌子上的一个苹果的力（即苹果的重力），就被桌子向上施加到苹果的力相抵消。如图 6.15 所示。这两个力大小相等而方向相反。

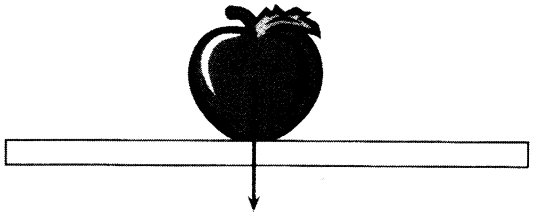


图 6.15　作用在稳态物体（桌子）上的（重）力

力的单位是牛顿（newton），一牛顿的力被定义为：使 1kg 质量的物体由静止而产生每秒 1 米的速度所需的力。

利用力的这一定义可以定义质量，而不再需要参照重力。

6.3.5　运动－速度

速度是物体位置随时间的变化速率，通常是以每秒米（m/s）为单位。但如果正在乘一辆汽车外出旅行，则速度就会用每小时几米（mph）或每小时几千米（kph）的单位来度量。在游戏中，游戏对象的移动速度则通常是用它在屏幕上每秒移动的像素数目（pps）来计算的。

如果已知一辆汽车行驶的时间，也已知车子的平均速度，则可以利用下面的公式来计算在这一段时间内汽车已经走完了多少路程：

新位置＝旧位置＋速度×时间

例如，观察图 6.16 中的汽车。

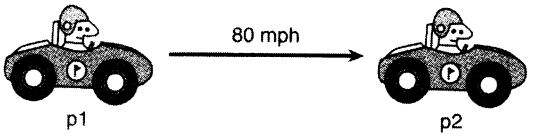


图 6.16　移动着的汽车

如果汽车旅行了两个小时，则 pl 和 p2 之间的距离为：

距离＝0＋80×2＝160英里

注意：这里要涉及 pl 和 p2 的精确位置的选择测量一个物体的位置时，通常是以它的质心（center of mass）来度量的。一个物体的质心可以理解为这一物体的平衡点。

6.3.6　运动－加速度

速度是距离的变化率，而加速度是速度的变化率，并用每秒每秒米数（m/s^2）度量。[①]

假设图 6.16 的汽车由静止开始，并以 5 m/s^2 的等加速度加速，则它的速度每隔 1 秒就增加 5m/s，如表 6.1 所示。

图 6.17 以图形方式表示了这一情况。

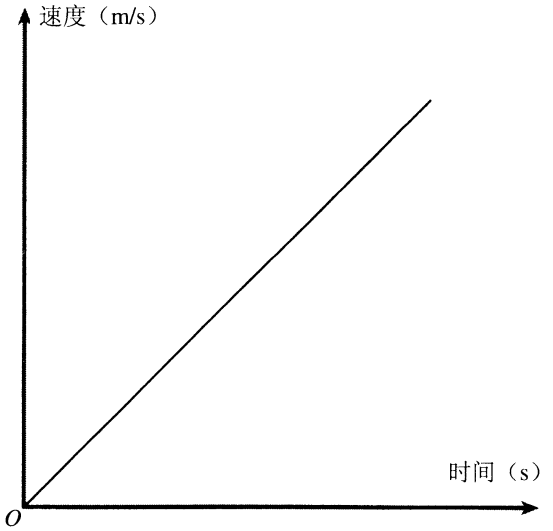


图 6.17　以等加速运动的汽车速度

[①] 译者注：原文误写成“每秒米”，而正确的应是“每秒每秒米”。

等加速运动的一个极好的例子是坠落的岩石，它是等加速度运动，因为对它起作用的力只有地球的重力。

表 6.1　由加速度引起的速度的变化

时 间	加速度（m/s^2）	速度（m/s）
1	5	5
2	5	10
3	5	15
4	5	20
5	5	25

物体也可以以非等加速度运动。例如，一个赛车的运动就是如此。它的速度可以近似用图 6.18 表示。开始时速度迅速增加，随后速度的增加就缓慢下来，最后则稳定于某个速度，此时加速度等于 0 了。

为了计算在时间 t 中以等加速度 a 航行的物体的距离，可以使用下列公式：

$$\text{航行距离} = ut + \frac{1}{2}at^2$$

其中 u 代表物体的初始速度。

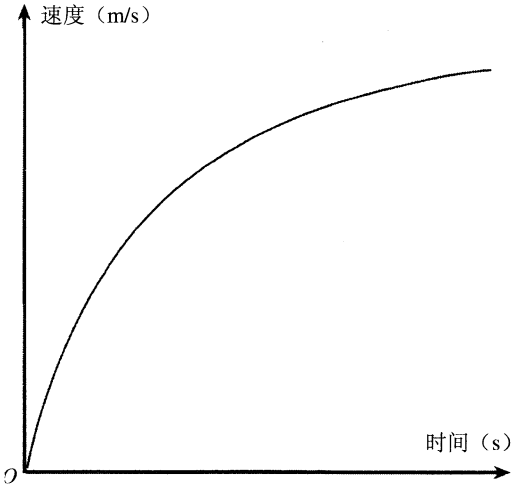


图 6.18　非等加速运动的速度变化

使用前面的例子，汽车由静止开始以 $5m/s^2$ 的加速度行驶了 5 秒，利用公式可以算出它已经行驶了多远。

$$\text{行驶距离} = 0\times5 + 5\times5\times5/2 \ ^{①}$$

$$= 62.5\text{（米）}$$

① 译者注：原文此行表达式有排印错误。

注意：字母 a 通常用来代表加速度，但由重力引起的加速度则通常都用 g 来表示。在地面上，g 的平均值为 9.8 m/s^2。

6.3.7 力、质量、加速度三者的关系

力、质量、加速度三者之间的关系可以用下式来表示：

$$力 = 质量 \times 加速度$$

由上式可知，施加在物体上的力与它的质量和加速度成正比。它是物理学中的一个很重要的公式，今后会经常使用它，特别是下面这个形式：

$$加速度 = 力 / 质量$$

因此，当给出了一个游戏对象的质量和作用在它上面的力后，就很容易计算它的加速度，并相应地去更新物体的速度和位置。游戏的每一帧中，都将会做类似下面这样的一些计算工作：

$$新的加速度 = 旧的加速度 + (力 / 质量)$$
$$新的速度 = 旧的速度 + 新的加速度 \times 帧时间$$
$$新的位置 = 旧的位置 + 新的速度 \times 帧时间$$

其中的帧时间（FrameTime）是指从上一帧到当前帧之间所经历的时间。

6.3.8 引力

引力是两个物体之间相互吸引所产生的力。设给定两个物体（如图 6.19 所示）其中一个的质量为 m，另一个质量为 M，它们之间的距离为 r，则计算它们之间的引力的公式是:

$$F = GMm / r^2$$

其中 G 是引力常数 6.673×10^{-11}。

因此，两个物体之间的引力大小与这两个物体的质量成正比，而与它们之间的距离的平方成反比。作为一个试验，可以用这个方程来计算地球的质量。首先，设想有一个质量为 m 的网球在地球上，地球的质量为 M。

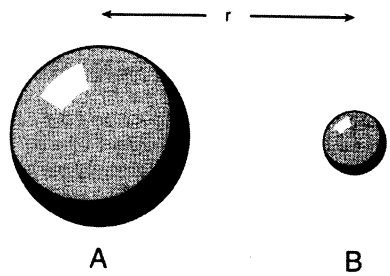


图 6.19 引力

由于作用于一个物体上的力等于它的质量乘以加速度（$F=ma$），将 F 代入上面的公式，这样可得：

$$ma = GMm / r^2$$

将等式两边的小 m 消去，可得：

$$a = GM / r^2$$

最终得出一个计算地球质量的公式：

$$M = ar^2 / G$$

已知 a 即为重力引起的加速度 g（近似 9.8m/s^2）。G 是重力常数（6.673×10^{-11}），而网球与地球的中心距离 r 与地球的半径（6378000m）相同（相比之下网球的半径可以忽略）。代入数据，则有：

$$M = 9.8\times(6378000)^2 / (6.67\times10^{-11}) = 5.974\times10^{-24}$$

求得的数据为地球质量的近似值。

6.4　人工控制的登月工程

登月工程的代码能够在 Chapter6/Lunar Lander-Manned 文件夹找到。不过，在继续进行下面的学习之前，可以使用其中的执行文件，看是否能安全地登陆？你会发现，这没有那么容易。

工程中的两个重要的类是 CLander 和 CController。CLander 与 CGun 类相似，但其功能比 CGun 要更复杂。它包含所需要的有关登月器和它的顶点的所有信息。CController 是 Windows 和 Lander 类之间的一个接口类。它同样也要处理登陆点（pad）的形状和背景里的星星。下面介绍 Ccontroller 的头文件。

6.4.1　Ccontroller 类的定义

指向 Ccontroller 类实例的全局指针在 WM_CREATE 中被初始化，然后在需要的地方就可以调用它（参考 main.h 的注释）来显示并更新场景。

```
Class CController
{
prvate:
    //用户能控制的登月飞船
    CLander*            m_pUserLander;
    //如果成功地登陆,则为 true
    bool                m_bSuccess;
    //存放星星的顶点缓冲区
    vector<SPoint>      m_vecStarVB;
```

```
    //存放登陆点形状的顶点缓冲区
    vector<SPoint>  m_vecPadVB;
    //登陆点的位置
    SVector2D       m_vPadPos;
    //保存窗口尺寸的记录
    int             m_cxClient,
                    m_cyCIient;
    void WorldTransform(vector<SPoint> &pad);
    void RenderLandingPad(HDC &surface);
public:
    CController (int cxClient, int cyClient);
    ~ CController ();
    //在窗口消息循环中调用 CLander::CheckForKeyPress()检测用户输入,
    //并相应地更新登月飞船的位置
    bool Update();
        //初始化新一轮的执行
    void NewRun();
    //由 WM_PAINT 调用,以显示在场景中的所有对象
        void Render(HDC &surface);
};
```

映射模式：映射模式是 GDI 用来配置绘图坐标的方式。通常在一窗体上绘图时 Y 轴坐标指向下方。由于本例程序谈及的都是有关重力和飞行器的效应，使用默认的窗体映射模式将与感觉违背。需要的坐标系统应是：当飞行器向上飞时其 Y 坐标应相应增加，以便让登月舱的高度正确表示出来。如果采用默认映射方式，则当登月器高度增加时，在窗体上画出时反而会落到较低的位置。Windows 允许用户自定义屏幕的映射坐标。采用下列方法就可以将影射模式改变成为 Y 轴指向上方：

```
SetMapMode (surface, MM_ANISOTROPIC);
SetViewportExtEx(surface, 1, -1, NULL);
SetWindowExtEx (surface, 1, 1, NULL);
SetViewportOrgEx (surface, 0, m_cyClient, NULL);
```

这里的函数不再详细介绍，如果有兴趣，可以去查看相关的文档。

为了恢复影射模式，可以调用下面的一些函数：

```
SetMapMode(surface, MM_ANISOTROPIC);
SetViewportExtEx(surface, 1, 1, NULL);
SetWindowExtEx(surface, 1, 1, NULL);
SetViewportOrgEx(surface, 0, 0, NULL);
```

6.4.2 CLander 类的定义

CLander 类中保存着一个记录，其中包含了需要知道的有关登月飞船的所有信息，也包括绘制登月飞船、从用户得到输入信息以及进行物理更新的各种方法。下面为 Clander 类的头文件。

```
Class Clander
{
private:
    //在世界坐标系中的位置
    SVector2D m_vPos;
    //在世界坐标系中的旋转
    double m_dRotation;
    //飞船的质量
    double m_dMass;
    //飞船的速度
    SVector2D m_vVelocity;
    //为了碰撞检测,需要知道登陆地点在哪里
    SVector2D m_vPadPos;
    //存储飞船顶点的缓冲区
    vector<SPoint> m_vecShipVB;
    //为绘制飞船所需的比例因子
    double m_dScale;
    //保持坐标变换后的顶点的缓冲区
    vector<SPoint> m_vecShipVBTrans;
    //保存喷气外形的顶点
    vector<SPoint> m_vecJetVB;
    vector<SPoint> m_vecJetVBTrans;
    //使用这个布尔量确定是否显示飞船的喷气
    //(如果用户按下(thrust 推进)按钮,则显示出喷气)
    bool m_bJetOn;
    //顾客窗体尺寸的本地备份
    int m_cxCIient;
    int m_cyCIient;
    //用来标识我们是否已经作过成功测试
    bool m_bCheckedIfLanded;
    //返回真,如果用户已经满足所有的登陆条件
    bool LandedOK ();
    //检测船的任何顶点是否在登陆平台水平面的下面
    bool TestForImpact (vector<SPoint> &ship);
    //利用该函数变换轮船最高点因此能显示它们
    void WorldTransform(vector<SPoint> &ship);
public:
    CLander (int        cxClient,  //用此来保存一个窗体大小
             int        cyClient,  //的局部记录
             double     rot,       //登月飞船的初始转角
             SVector2D  pos,       //登月飞船的初始位置
             SVector2D  pad);      //登陆点位置
    void Render(HDC surface);
    //为开始一个新尝试重置所有相关的参数
    void reset(SVector2D &NewPadPos);
    //根据用户按键更新飞船
    void UpdateShip ();
};
```

6.4.3 UpdateShip 函数

如何从用户端接收输入，如何更新登月船的位置和速度，都是在 UpdateShip 函数中完成的。以下为 update 函数在每一个帧中功能的概括：

- 检测用户是否按下了一个键。
- 相应地更新登月飞船的速度、加速度和旋转角。
- 更新登陆船的位置。
- 变换登陆船的顶点。
- 检测登陆船的顶点是否在地平面（ground level）以下。
- 如果登陆船已到达地平面，检测登陆是成功或者失败。

以下是实现这些功能的代码：

```
void CLander::UpdateShip (double TimeElapsed)
{
```

这一函数带有的参数 TimeElapsed 代表自前一帧之后（到当时为止）的延续时间。采用这样的做法是因为，虽然可以把定时器设置成每秒固定的帧数，但这样会产生问题。例如，若程序把出帧率设置在每秒 60 帧（60 fps），而程序将在一台动行较慢的计算机上运行，仅仅能达到 40 fps。如果是这样，所有物理学计算都将被搞乱了，游戏中的太空船（或赛车、坦克车）在达不到固定出帧率的机器上运行，就会在完全不同的位置上停下来。所以采用一种更安全的计算方法，这就是利用前一帧之后的延续时间。CTimer 类中包含了一种方法可以完成这一工作，这个方法名为 GetTimeElapsed。

```
//如果飞船已经坠毁或着陆
if (m_bCheckedIfLanded)
{
return;
}
```

如果程序检测飞船已登陆（在 TestFoImpact 函数中），则 m_bCheckedlfLanded 被设置为真，Update 函数就没有做任何事，直接返回。

```
//把喷气的图形关闭
m_bJetOn = false;
```

每当用户按下推进键（空格键），代表推进喷气的一个小图形就会在登月飞船的后面被画出来，上述标志设置为 ON 或 OFF，用来指示函数是否应将图形显示出来。

```
//测试用户输入,并作相应的更新
if (KEYDOWN(VK_SPACE))
 {
 //登月飞船每一帧（tick）的加速度是根据推进器所施加的力、登月飞船
 //的质量以及从上一帧开始起所经历的时间决定
 double ShipAcceleration = (THRUST_PER_SECOND * TimeElapsed) / m_dNass;
 //分解此加速度为 x,y 分量并加入登月飞船的速度向量
```

```
m_vVelocity.x += ShipAcceleration * sin(m_dRotation);
m_vVelocity.y += ShipAcceleration * cos(m_dRotation);
//将喷气图形设置为 ON
m_bJetOn = true;
}
```

当用户按下空格键时，由推进而产生的加速度的正确数值必须计算出来并应用于登月飞船。首先，此加速度是在此时间段中根据作用于飞船质量的力来计算的（加速度=力/质量）。然后加速度向量分解为在 x 和 y 方向的分量，再加入飞船速度的相应分量中。

注意：在 WindowProc 中，实际并没有检测按键动作，这里定义了一个宏，其形式为：

```
#define KEYDOWN(vk_code)
((GetAsyncKeyState(vk_code) & 0x8000 ? 1:0 )
```

可以在 Clander.cpp 文件的开始找到此定义。

```
if (KEYDOWN(VK_LEFT))
{
    m_dRotation -= ROTATION_PER_SECOND * TimeElapsed;
     if (m_dRotation < -PI)
     {
      m_dRotation + = TWO_PI;
     }
}
if (KEYDOWN(VK_RIGHT))
{
   m_dRotation += ROTATION_PER_SECOND * TimeElapsed;
   if(m_dRotation> TWO_PI
     {
     m_dRotation-= TWO_PI
     }
 }
//加入重力矢量
 m_vVelocity.y += GRAVITY * TimeElapsed;
//更新登月飞船的位置
m_vPos += m_vVelocity*TimeElapsed * SCALING_FACTOR;
```

这里，登月飞船模块的速度是根据物理学定律来更新的。应注意这个常数值 SCALING_FACTOR。加入此常数的原因目的是使游戏更加有趣。

在编写游戏程序时，距离是以像素为单位，而不是用米为单位来度量的。登月飞船开始降落时，是在登陆点的上方 300 个像素处，因此在真实世界中这代表 300 米。下面计算飞船在月球重力（1.63m/s^2）的影响下，从上空 300 米处到降落登陆点，需要花多长时间。

根据方程：

$$d = ut + \frac{1}{2}at^2$$

这里 u（初始速度）是零，因此可以简化为：

$$d=\frac{1}{2}at^2$$

上式可以进一步转化为：

$$t=\sqrt{\frac{2\times d}{a}}$$

代入数据，可得：

$$t=\sqrt{\frac{2\times 300}{1.63}}=19.18\ （秒）$$

到达着落点需要超过 19 秒，这样的时间太长了，为了弥补，引入了一个比例因子。等效于让登月飞船在较低的高度开始着陆。这样仍然能感觉与实际情况一致，但现在登陆就变得容易控制和有趣了。

回到更新函数的讨论：

```
//检查边界
if(m_vPos.x > WINDOW_WIDTH)
{
  m_vPos.x = 0;
}
if (m_vPos.x < 0)
{
  m_vPos.x = WINDOW_WIDTH;
}
```

上面程序确保了当登陆飞船在屏幕上向左或向右飞得太远时，就会自动进行回绕（wrap）。

下面的程序检测登陆船是否已经坠毁或已经成功登陆。

```
//在变换登陆飞船的顶点之前,应先创建顶点的一个备份
m_vecShipVBTrans = m_vecShipVB;
//变换顶点
WorldTransform(m_vecShipVBTrans);
```

要检测飞船是否已经到达地平面，必须先把它的顶点坐标转换成世界坐标。

```
//如果我们飞得比地平线还低,则我们就可以结束本次运行了
if(TestForImpact(m_vecShipVBTrans))
 {
```

TestForImpact 是一个用来检测飞船所有顶点中是否有某些已在地平面之下的函数。如果发现某个顶点在地平面之下，则程序检查飞船模块是着陆了，或者还是坠落到地面了。

```
  //检查用户是否已经着陆
  if(! m_bCheckedIfLanded)
```

```
    {
        if(LandedOK ())
         {
         PlaySound("landed" , NULL, SND_ASYNC|SND_FILENAME);
         }
        else
         {
         PlaySound("explosion", NULL, SND_ASYNC|SND_FILENAME);
         }
        m_bCheckedIfLanded = true;
      }
   }
   return;
}
```

LandedOK 是一个测试登月飞船模块是否已满足成功登陆的所有要求的函数。然后UpdateShip 播放一个适当的音乐文件并返回。

LandedOK 函数的形状如下：

```
bool CLander::LandedOK ()
{
//计算登月飞船与着落点的距离
double DistFromPad = fabs(m_vPadPos.x - m_vPos.x);
//计算登月飞船的移动速度
double speed = sqrt((m_vVelocity.x * m_vVelocity.x)
                    +(m_vVelocity.y * m_vVelocity.y));
//检查是否成功登陆
if((DistFromPad < DIST_TOLERANCE) &&
     (speed < SPEED_TOLERANCE) &&
     (fabs(m_dRotation) < ROTATION_TOLERANCE))
   {
     return true;
   }
 return false;
}
```

计算成功登陆所用的各种允许误差（或公差，tolerance）可以在文件 defines.h 中找到。由此可知，为了成功登陆，登月飞船必须用低于 SPEED_TOLERANCE 的速度飞行，飞船离开登陆点中心的距离必须小于 DIST_TOLERANCE，而它（相对于 y 轴）的偏转角度的绝对值必须小于 ROTATION_TOLERANCE。

6.5 遗传算法控制的登月飞船

解决登月飞船控制问题的关键，在于正确定义下面的 3 件事情：

- 为可能出现的所有候选解进行编码。

- 确定意义明确的变异和杂交操作。
- 设计一个准确的适应性函数。

6.5.1 为基因组编码

候选解可以用二进制的位串进行编码，或者利用整数（数列）的一个置换（permutation）来编码，也可用一系列实数来进行编码。但如果为非常复杂的候选解进行编码时，不论基因组有多长，都应为它们指定变异和杂交操作，甚至有可能利用非常复杂的数据结构来作为基因。但目前必须保证变异和杂交操作能以一种有意义的方式应用到问题中。

一共有 4 种不同的方法来控制登月飞船：

- 可以使用一个向前推进的力。
- 可以使用一个向左转动的力。
- 可以使用一个向右转动的力。
- 不施加任何力，即让它进行漂移或滑翔。

这 4 种控制动作的每一种都在某个时间周期中应用，该时间周期用它更新每一帧所需的秒数（分数值）来表示。因此，在每一个编码中必须同时表示出行动（action）和持续时间（duration）两种数据。图 6.20 说明了数据是怎样编码的。可以看出，每个基因包含了一对数据。其中前半个基因代表飞船应该采取的行动，后半个基因则表示该行动应该持续的时间长度。

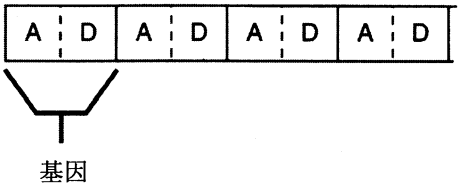


图 6.20 基因组的编码

参考文件 defines.h 可知，每个基因的一次行动的最大持续时间由 MAX_ACTION_DURATION 定义为 30 个帧。

下面为基因组结构的代码形式：

```
//为登月飞船能实行的不同行动确定一个枚举类型：
    enum action_type(rotate_left,
                     rotate_right,
                     thrust,
                     non);

struct Sgene
{
  action_type    action:
  //行动持续时间以帧为单位来计算
int          duration;
SGene()
```

```
{
 //创建一个随机行动
 action = (action_type)RandInt(0,3);
 duration = RandInt(1, MAX_ACTION_DURATION);
 }
 SGene(action_type a, int d):action(a), duration(d){}
 //需要重载 == 操作,从而能够测试动作是否相同 (在 GA 的杂交过程中使用)
 bool operator == (const SGene &rhs) const
  {
    return (action == rhs.action) && (duration == rhs.duration);
  }
};
```

以下为定义基因组的代码：

```
  struct Sgenome
  {
    vector<SGene>  vecActions;
    double         dFitness;
    SGenome():dFitness(0){}
    SGenome(const int num_actions):dFitness(0)
    {
      //创建一个随机的行动向量
      for (int i=0; i<num_actions: ++i)
       {
       vecActions.push_back(SGene());
       }
     }
     //重载 '<' 操作符,用于排序
     friend bool operator<(const SGenome& lhs, const SGenome& rhs)
     {
       return (lhs.dFitness < rhs.dFitness);
     }
};
```

假设遗传算法的初始群体是由一些随机的基因组即随机的行动和持续时间串组成的，容易看出每个动作如何能够依次应用于指定的时间周期中，由此就可以控制登月飞船了。

6.5.2　杂交和变异操作

下面的程序将采用多点杂交，两个基因组的多点杂交，就是沿着基因组长度，随机地选取一些点位，将对应基因进行互换，直到把整个基因组处理完毕。可参考第 5 章“创建更好的遗传算法”中的图 5.8。

变异算子则是沿基因组长度在某些点位上将基因的两个组成部分进行随机变化。首先按照变异率，把其中的动作随机地改变成另外的动作（动作可能保持不变）。然后将动作的持续时间改变为不超过 MAX_MUTATION_DURATION 的另一个持续时间。持续时间同时还要被限制在零到 MAX_ACTION_DURATIDN 之间。

以下为变异算子的代码：

```
void CgaLander::Mutate(vector<SGene> &vecActions)
{
  for (int gene=0; gene<vecActions.size(); ++gene)
  {
    //判断是否实行动作变异
     if (RandFloat() < m_dMutatonRate)
     {
       vecActions[gene].action = (action_type)RandInt(0,3);
     }
    //是否实行持续时间的变异
    if (RandFloat() < m_dMutationRate/2)
    {
    vecActions[gene].duration += RandomClamped()*MAX_MUTATION_DURATION;
    //限定持续时间
    Clamp(vecActions[gene].duration, 0, MAX_ACTIOR_DURATION);
    }
  }//下一基因
}
```

6.5.3 适应性函数

适应性函数常常是遗传算法最难确定的一个部分，这是因为问题常常是多目标的。在本例中登月飞船要成功着陆，在着陆之前必须同时满足几个目标，这就是：

- 登月飞船与登陆地点中心的距离必须在某个范围之内。
- 登月飞船落到登陆点的速度必须在某个速度之下。
- 登月飞船偏离 y 轴的转角必须在某个预定范围之内。

利用上述 3 个目标来编写适应性函数时，登陆动作显得过于简单。在游戏中，飞船从天上笔直着陆，并采用相当精确的转动角度和推进力着陆。因此加入人工的干预操作，又加入了一个目标：飞船在空中停留的时间愈长，适应性分数也增加得比别人愈多。算法就变得比较完善，飞船的行为变得比较真实。

下面为适应性函数的代码：

```
void CLander::CalculateFitness(int generation)
{
  //计算飞船离开着落点的距离
   double DistFromPad = fabs(m_vPadPos.x - m_vPos.x);
   double distFit = m_cxClient-DistFromPad;
  //计算飞船的速度
  double speed = sqrt((m_vVelocity.x*m_vVelocity.x)
                    +(m_vVelocity.y*m_vVelocity.y));
  //计算由于转角而得到的适应性分数
  double rotFit = 1/(fabs(m_dRotation)+1);
  //计算因在空中停留时间所得的分数
  double fitAirTime = (double)m_cTick/(speed+1);
```

cTick 是用来保存登月飞船从开始下降时刻起所经历的帧数的一个计数器。把它的值除以登月飞船的登陆速度，就得到一个综合空中停留时间和登月飞船最终速度的奖励分。

```
    //计算适应性分数
m_dFitness = distFit + 400*rotFit + 4*fitAirTime;
```

需要使用一些系数（1、400、4）把几个适应性分数组合成最终的总体适应分，这是设计多目标遗传算法时必须采用的一种典型技巧。在这一例子中，登月飞船因靠近登陆点目标而能得到的最大分数是 400（窗口的宽度），因旋转角度能得到的最大分数是 1，因此使用 400 这个比例系数作为 rotFit 的系数；又因从登月飞船的速度和空中停留时间所得的分数大体为 100 这一平均值，因此使用 4 作为乘数。利用这样的办法，每一个目标最大都能为总适应性分近似地提供 400 分。

```
    //检查是否已经成功登陆
    if( (DistFromPad          < DIST_TOLERANCE) &&
        (speed                < SPEED_TOLERANCE) &&
        (fabs(m_dRotation)    < ROTATIDN_TOLERANCE) )
    {
      m_dFitness = BIG_NUMBER;
    }
  }
```

适应性函数最后检查登月飞船是否已通过为安全登陆所需要的全部要求，如果是，就把一个很大的数赋给适应性分数，这样程序就知道已经得到一个解。

6.5.4　更新函数

下面介绍 GA 版登月船程序的 update 函数，因为实际（phisics）必须使用与前面不同的处理方法。当演化候选解时，应使计算机运行得尽可能快，以便快速找到一个解。由于人工控制版的 update 函数在每一帧中都加入了最低延迟时间，即使用每秒 5000 帧的标准来运行它，登月飞船仍然像是在作实时运动。

为了让代码能够用加速方式运行（由重力引起的加速度），在 defines.h 中预先利用出帧率（每秒产生的帧数）计算得出推力和旋转速度。

```
#define GRAVITY_PER_TICK    GRAVITY/FRAMES_PER_SECOND
#define THRU5T_PER_TICK     THRUST/FRAMES_PER_SECOND
#define ROTATION_PER_TICK   ROTATION/FRAMES_PER_SECOND
```

有了这些值就可用来实现帧的实际（phisics）更新。当程序以每秒 FRAMES_PER_SECOND 帧的速度运行时，任何事物都会按它应有的形状显露出来。因此，只要能不限速度地让计算机尽快运行，实际（phisics）也就会和任何其他事物一起加速。这样就会让遗传算法尽快得到一个解。

在 update 函数中使用这种方法还能够使更新函数变得更快。因为计算工作减少了。缺点是，如果要在人工控制版中使用这种技术，而程序是在一台慢速的计算机上的运行（不

能达到所需的出帧率），则在不同出帧率的计算机上将会感到有不同的实际世界。

下面是修改后的 UpdateShip 函数的第一部分代码：

```
bool CLander::UpdateShip ()
{
    //如果飞船已经坠毁或成功登陆,则返回 false
    if(m_bCheckedIfLanded)
    {
      return false;
    }
    //这是当前的行动
    action_type action;
   //检查是否仍有一个行动要执行。如果没有,就让登月飞船自动漂流（滑翔）
   //直到它落到地面
   if(m_cTick >= m_vecActions.size())
    {
    action = non;
   }
   else
    {
     action = m_vecActions[m_cTick++];
    }
```

在每一个 epoch 之前，所有个体的基因组都用解码函数 Decode 转化为一个行动向量。利用这种办法很容易将帧记数器用作动作数组的下标，以确定在每一帧中哪一个行动需要执行，如图 6.21 所示。

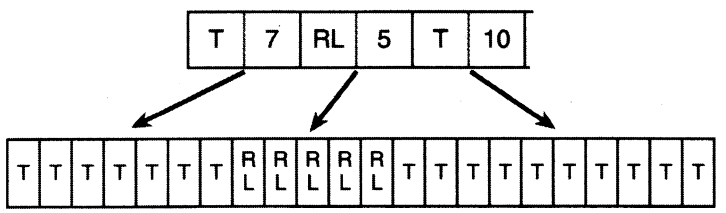


图 6.21　把基因组译码成为一个动作向量

```
    //关闭绘制喷气图形的功能
   m_bJetOn = false;
    switch (action)
    {
     case rotate_left:
     m_dRotation -= ROTATIOR_PER_TICK;
     If (m_dRotation < -PI)
     {
         m_dRotation += TWO_PI;
     }
       break;
```

```
        case rotate_right:
         m_dRotation += ROTATION_PER_TICK;
         If (m_dRotation > TWO_PI)
         {
            m_dRotation -= TWO_PI;
         }
         break;
      case thrust:
         //根据推进器的作用力和登月飞船的质量,计算登月
         //飞船每帧的加速度
         double ShipAcceleration = THRUST_PER_TICK/m_dMass;
         //把加速度向量分解为它的x,y分量
         //并加入到登月飞船的速度中
         m_vVelocity.x += ShipAcceleration * sin(m_dRotation);
         m_vVelocity.y += ShlpAcceleration * cos(m_dRotation);
         //将喷气作图切换为true
         m_bJetOn = true;
         break;
      case non:
         break;
   }//结束switch语句
   //加入到重力矢量
   m_vVelocity.y += GRAVITY_PER_TICK;
   //更新登月飞船的位置
   m_vPos += m_vVelocity;
```

利用上述方法可以减少物理计算。比例因子（scaling factor）60 被设置成和出帧率（framerate）60 一致，所以，利用数学方法，它们可以相互抵消（60/60=1）。

这里省略了其余的代码（它和前面最后版本的代码非常类似）。程序最后用来做一些范围检查——检查登月飞船是否已经在地面之下，并更新它的适应性分数。

6.5.5 运行程序

当运行 GA 控制的登月飞船的可执行程序时，整个群体的所有成员都会在屏幕上立即显示出来。按一次 B 键，可以把整个群体的显示切换为仅仅显示由上一代以来适应分数最高的个体，再按一次 B，则进行反向切换。按 F 键可用来加速，按 R 键则为开始一轮新的运行并将所有参数进行复位（reset）。为了找到一个解，遗传算法平均需要运行 100～300 代。如果运行的代的数目达到一个预先规定的最大值（定义为 500），遗传算法也会自动复位，并重新开始新一轮的运行。

本工程的遗传算法设置采用轮盘赌选择法来选拔精英，没有使用适应性比例选择。因此可以对许多参数进行改进试验，以提高它的性能。

由于能同时观察到所有个体，从而能确切得到群体有怎样的差异，进而得知不同的选择方法、不同的变异算子或不同的杂交算子，对性能会有什么样的影响。

6.6 总　　结

学习完本章，相信读者会有许多遗传算法的各种想法，也可能对它们有了一些初步的认识。

6.7 习　　题

1．尝试使用第 5 章中学到的登月飞船代码中的各种不同方法，观察不同的技术会怎样改变收敛速度，思考它们任意一种是否都能改进遗传算法？（通常，加速运行遗传算法使用 F 键，判断群体是收敛还是发散使用 B 键）

2．在程序中加入第 5 个目标。例如，登月飞船可能以固定数量的燃料开始，每使用一次推进操作，燃料就要烧掉一个固定的数量。这意味着飞船现在必须在所有的燃料被耗尽之前登陆月球。

3．利用遗传算法来解决 2 维空间中若干行星围绕一颗恒星转动的问题。着手这一问题时，你必须解决行星的轨道和它们的月球轨道，思考一下用二进制表示星系？（这一练习将帮助你巩固到目前为止已经学习到的任何一种知识，包括遗传算法技术、数学和物理学。）

4．创建一个“空间入侵者”（space invader）类的游戏，这里每一种不同异类的行为都要利用一个遗传算法来演化生成。

矩阵乘法问题的答案：

$$[1 \quad 2 \quad 3 \quad 4]\times\begin{bmatrix}5 & 9\\ 6 & 10\\ 7 & 11\\ 8 & 12\end{bmatrix}=[70 \quad 110]$$

第3篇 神经网络

本篇包括以下内容：

第7章 神经网络概述

第8章 为机器人提供知觉

第9章 有监督的训练方法

第10章 实时演化

第11章 演化神经网络的拓扑

第 7 章　神经网络概述

因为我们没有很好了解大脑，我们经常试图用最新的技术作为一种模型来解释它。在我童年的时候，我们都坚信大脑是一部电话交换机（否则它还能是什么呢？）我当时还看到英国著名神经学家谢林顿把大脑的工作挺有趣地比作一部电报机。更早些时候，弗罗伊德经常把大脑比作一部水力发电机，而莱布尼茨则把它比作了一台磨粉机。我还听人说，古希腊人把大脑功能想象为一付弹弓。显然，目前要来比喻大脑的话，那只可能是一台数字电子计算机了。[①]

John R.Searle

7.1　神经网络介绍

曾有很长一段时期，人工神经网络对我来说是完全神秘的东西。当然，有关它们我在文献中已经读过了，我也能描述它们的结构和工作机理，但我始终没有能“啊哈!”一声，如同你头脑中一个难于理解的概念有幸突然得到理解时的感觉那样。我的头上好像一直有个榔头在敲着，或者像电影 Animal House（中文片名“动物屋”）中那个在痛苦地尖叫“先生，谢谢您，再给我一个啊!”的可怜家伙那样。我无法把数学概念转换成实际的应用。有时我甚至想把我读过的所有神经网络的书的作者都抓起来，把他们绑到一棵树上，大声地向他们吼叫：“不要再给我数学了，快给我一点实际东西吧!”。但无需说，这是永远不可能发生的事情。我不得不自己来填补这个空隙……由此我做了在那种条件下惟一可以做的事情。我开始干起来了。

这样几个星期后，在一个美丽的日子里，当时我在苏格兰海边度假，当我越过一层薄雾凝视着狭长的海湾时，我的头脑突然受到一个冲击。我一下子悟到了人工神经网络是怎样工作的。我得到“啊哈!”的感觉了！但我此时身边只有一个帐篷和一个睡袋，还有半盒子脆玉米片，没有电脑可以让我迅速写出一些代码来证实我的直觉。Arghhhhh！这时我才想到我应该买一台手提电脑。不管怎样，几天后我回到家了，我立刻让我的手指在键盘上飞舞起来。几个小时后我的第一个人工神经网络程序终于编成和运行了，并且工作得挺好！自然，代码写得有点乱，需要进行整理，但它确实已能工作了，并且，更重要的是，我还知道它为什么能工作。我可以告诉你，那天我是多么地得意。

我希望本书传递给你的就是这种“啊哈!”的感觉。当我们学完遗传算法时，你可能已尝到了一点感觉，但你希望这种感觉是美妙的话，那就要等把神经网络部分整个学完！

---

① 译者注：引自 John R.Searle 的 *MINDS,BRAIN AND SCIENCE*，P44。John R.Searle 是美国当代哲学-心理学家，写过大量有关大脑和意识本质方面的书。

7.2　一个生物学的神经网络——大脑

因人工神经网络是仿照生物的大脑来工作的，为了帮助理解，这里介绍大脑灰质（gray matter）。

大脑的外层是起皱的，这一层组织称为皮层（cortex）。大脑有两层：灰色的外层（这就是“灰质”一词的来源，但没有经过福尔马林处理的新鲜大脑实际是粉红色的）和白色的内层。灰色层只有几毫米厚，其中紧密地压缩着几十亿个被称作神经细胞（或神经元 neuron）的微小细胞。白色层在皮层灰质的下面，占据了皮层的大部分空间，是由无数神经细胞相互连接组成。皮层的皱褶与光滑的皮层相比，能容纳更多的神经细胞。人的大脑大约含有 100 亿个这样的微小处理单元；一只蚂蚁的大脑大约也有 25 万个。

表 7.1 显示了人和几种动物的神经细胞数目。

表 7.1　动物神经细胞数目的比较

动 物	神经细胞的数目（数量级）
蜗 牛	10 000 （$=10^4$）
蜜 蜂	100 000 （$=10^5$）
蜂 雀	10 000 000 （$=10^7$）
老 鼠	100 000 000 （$=10^8$）
人 类	10 000 000 000 （$=10^{10}$）
大 象	100 000 000 000 （$=10^{11}$）

在胎儿发育的 9 个月内，这些细胞以每分钟产生 2.5 万个的惊人速度被创建出来。神经细胞和人身上任何其他类型细胞十分不同，每个神经细胞都长有轴突（axon），它的长度有时可以达到几厘米[①]，用来将信号传递给其他的神经细胞。神经细胞的结构如图 7.1 所示。它由一个细胞体（soma）、一些树突（dendrite）和一根可以很长的轴突组成。神经细胞体是一颗星状的球形物，里面有一个细胞核（nucleus）。树突由细胞体向各个方向长出，本身可有分支，它们是用来接收信号的。轴突也有许多分支。轴突通过分支的末梢（terminal）和其他神经细胞的树突相接触，形成所谓的突触（synapse），一个神经细胞通过轴突和突触把产生的信号送到其他神经细胞。

每个神经细胞通过它的树突与大约 1 万个其他神经细胞相连。这就使得人的大脑中所有神经细胞之间的连接总计可能有 l0000000 亿个。这比 100 兆个现代电话交换机的连线数目还多。所以毫不奇怪为什么我们有时会产生头疼毛病！

有趣的事实：曾有人估算过，如果将一个人大脑中所有神经细胞的轴突和树突依次连接起来，并拉成一根直线，则可以从地球连到月亮，再从月亮返回地球。如果把地球上所有人脑的轴突和树突连接起来，则可以伸展到离我们最近的星系！

① 译者注：人在整个身体内的神经细胞最长的可以超过 1m，即 100cm。

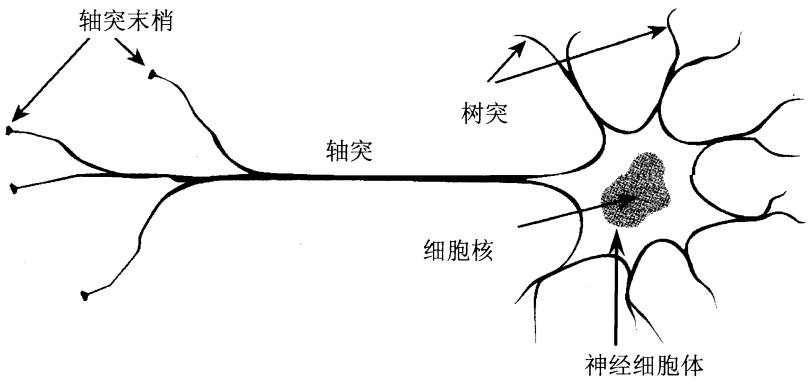


图 7.1　生物学神经细胞

神经细胞利用电化学过程交换信号。输入信号来自另一些神经细胞。这些神经细胞的轴突末梢（也就是终端）和本神经细胞的树突相遇形成突触，信号就从树突上的突触进入本细胞。信号在大脑中传输的方式是一个相当复杂的过程，把它看作与计算机一样，利用一系列的 0 和 1 来进行操作。大脑的神经细胞也只有两种状态：兴奋（fire）和不兴奋（即抑制）。传输信号的强度不变，变化的仅仅是频率。神经细胞利用一种我们还不知道的方法，把所有从树突上突触进来的信号进行相加，如果全部信号的总和超过某个阈值，就会激发神经细胞进入兴奋（fire）状态，这时就会有一个电信号通过轴突传给其他神经细胞。如果信号总和没有达到阈值，神经细胞就处于抑制状态。

正是由于数量巨大的连接，使得大脑具备难以置信的能力。尽管每一个神经细胞仅仅工作于大约 100Hz 的频率，但因各个神经细胞都以独立处理单元的形式并行工作着，使人类的大脑具有下面这些非常明显的特点：

- **能实现无监督的学习**。大脑能够自己进行学习，而不需要导师的监督教导。如果一个神经细胞在一段时间内受到高频率的刺激，则它和输入信号的神经细胞之间的连接强度就会按某种过程改变，使得该神经细胞下一次受到激励时更容易兴奋。这一机制是 50 多年以前由 Donard Hebb 在他写的 *Organination of Behavior* 一书中阐述的。他写道：“当神经细胞 A 的一个轴突重复地或持久地激励另一个神经细胞 B 后，则其中的一个或同时两个神经细胞就会发生一种生长过程或新陈代谢式的变化，使得作为激励 B 细胞之一的 A 细胞，它的效能会增加。”
  相反，如果一个神经细胞在一段时间内不受到激励，那么它的连接的有效性就会慢慢地衰减。这一现象就称为可塑性（plasticity）。
- **对损伤有冗余性（tolerance）**。大脑即使有很大一部分受到了损伤，它仍然能够执行复杂的工作。一个著名的试验就是训练老鼠在一个迷宫中行走。然后，科学家们将其大脑一部分一部分地、越来越大地加以切除。他们发现，即使老鼠的很大一部分的大脑被切除后，它们仍然能在迷宫中找到行走路径。这一事实证明了，在大脑中，知识并不是保存在一个局部地方。另外所做的一些试验则表明，如果大脑的一小部分受到损伤，则神经细胞能把损伤的连接重新生长出来。

- **处理信息的效率极高。**神经细胞之间电化学信号的传递，与一台数字计算机中 CPU 的数据传输相比，速度是非常慢的，但因神经细胞采用了并行的工作方式，使得大脑能够同时处理大量的数据。例如，大脑视觉皮层在处理通过视网膜输入的一幅图像信息时，大约只要 100ms 的时间就能完成。考虑到你的神经细胞的平均工作频率只有 100Hz，100ms 的时间就意味只能完成 10 个计算步骤！想一想通过我们眼睛的数据量有多大，你就可以看到这真是一个难以置信的伟大工程了。
- **善于归纳推广。**大脑和计算机不同，它极擅长的事情之一就是模式识别，并能根据已熟悉信息进行归纳推广（generalize）。例如，一个人能够阅读另一个人所写的手稿上的文字，即使以前从来没见过他所写的东西。
- **它是有意识的。**意识（consciousness）是神经学家和人工智能的研究者广泛而又热烈地在讨论的一个话题。对于意识究竟是什么，至今尚未取得实质性的统一看法。我们甚至不能同意只有人类才有意识，或者包括动物王国中人类的近亲在内才有意识。一头猩猩有意识吗？你的猫有意识吗？上星期晚餐中被你吃掉的那条鱼有意识吗？

因此，一个人工神经网络（Artificial neural network，ANN）所要完成的任务，就是要在当代数字计算机现有规模的约束下，模拟生物学大脑的这种大量的并行性，并在实现这一工作时，使它能显示许多和生物学大脑相类似的特性。

7.3　数字版的神经网络

生物的大脑是由许多神经细胞组成的，模拟大脑的人工神经网络 ANN 也是由许多称为人工神经细胞（Artificial neuron，也称人工神经原，或人工神经元）的细小结构模块组成。可以看作人工神经细胞是真实神经细胞的一个简化版，但采用了电子方式来模拟实现。一个人工神经网络中需要使用多少个数的人工神经细胞，差别可以非常大。有的神经网络只需要使用 10 个以内的人工神经细胞，而有的神经网络可能需要使用几千个人工神经细胞。这完全取决于这些人工神经网络准备实际用来做什么。

有趣的事实：Hugo de Garis 曾在一个雄心勃勃的工程中创建并训练了一个包含 10 亿个人工神经细胞的网络。这个人工神经网络被他非常巧妙地建立起来了，它采用蜂房式自动机结构，目的就是为一计算机客户定制一个叫做“CAM 大脑机器”的计算机（cellular automata machine，CAM）。此人曾自夸地宣称这一人工网络机器将会有一只猫的智能。但雇用他的公司在他的梦想尚未实现之前就破产了。时间将会告诉我们他的思想最终是否能变成实际有意义的东西。[①]

[①] 译者注：Hugo de Garis 现在为犹他州立大学教授，有关他和他的 CAM 机器，可在该校网站的一个网页上看到报道，其上有真实的照片，见 http://www.cs.usu.edu/~degaris。

图 7.2 为一个人工神经细胞的示意图，左边几个灰底圆中所标字母 w 代表浮点数，称为权重（weight，或权值、权数）。进入人工神经细胞的每一个输入（input）都与一个权重 w 相联系，正是这些权重将决定神经网络的整体活跃性。假设所有权重都被设置成-1 和 1 之间的一个随机小数。因为权重可正可负，故能对与它关联的输入施加不同的影响，如果权重为正，就会有激发（excitory）作用，权重为负，则会有抑制（inhibitory）作用。当输入信号进入神经细胞时，它们的值将与它们对应的权重相乘，作为图中大圆的输入。大圆的“核”把所有这些新的、经过权重调整后的输入全部加起来，形成单个的激励值(activation value)，用于激发神经细胞进入兴奋态（输出 1）或不进入兴奋态（输出 0）。激励值也是一浮点数，且同样可正可负。如果激励值超过某个阀值（作为例子我们假设阀值为 1.0），就会产生一个值为 1 的信号输出；如果激励值小于阀值 1.0，则输出一个 0。这就是人工神经细胞最简单的一种激励方式。在这里，从激励值产生输出值是由一个函数来完成，这个函数就叫阶跃（型）激励函数[①]。看一看图 7.3 后你就能猜到为什么有这样的名称。

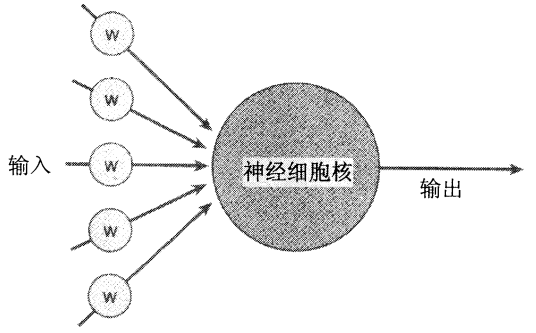


图 7.2　一个人工神经细胞

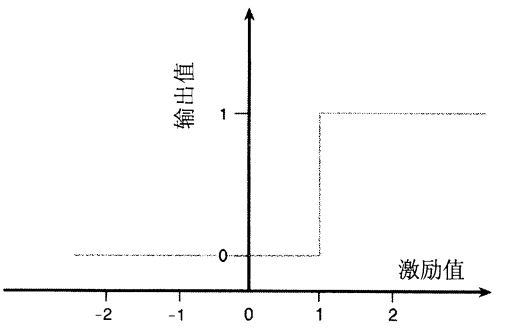


图 7.3　阶跃激励函数

7.3.1　相关的数学知识

一个人工神经细胞（简称“神经细胞”）可以有任意 n 个输入（n 代表输入总数）。可以用下面的数学表达式来代表所有 n 个输入：

$$x_1, x_2, x_3, x_4, x_5, \cdots, x_n$$

同样 n 个权重可表达为：

$$w_1, w_2, w_3, w_4, w_5, \cdots, w_n$$

激励值就是所有输入与它们对应权重的乘积的总和，因此可以写为：

$$a = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + \cdots + w_nx_n$$

这种求和式，简化为：

[①] 译者注：激励函数在图 7.2 以及后面出现的更详细的神经细胞模型中都未表示出来。

$$a=\sum_{j=0}^{j=n} w_j x_j$$

注意：神经网络的各个输入，以及为各个神经细胞的权重设置，都可以看作一个 n 维的向量。

假设输入数组和权重数组均已初始化为 $x[n]$和 $w[n]$，则求和的代码如下：

```
double activation = 0;
for(int i=0; i<n; ++i)
   {
     activation += x[i] * w[i];
   }
```

图 7.4 以图形的方式表示了此方程。如果激励值超过了阈值，神经细胞就输出 1；如果激励值小于阈值，则神经细胞的输出为 0。这和一个生物神经细胞的兴奋和抑制是等价的。假设一个神经细胞有 5 个输入，它们的权重 w 都初始化成±1 之间的随机值（$-1<w<1$）。表 7.2 说明了激励值的求和计算过程。

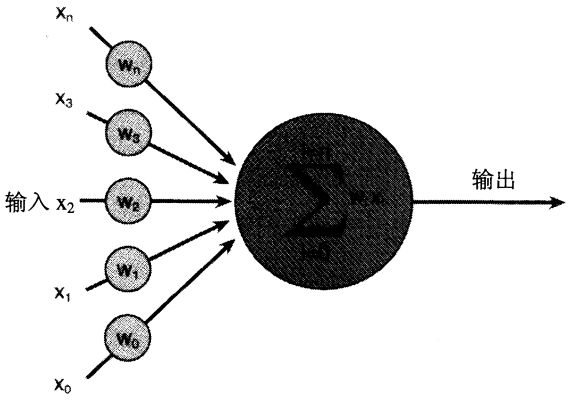


图 7.4　神经细胞的激励函数

假定激活所需阈值=1，则因激励值 1.1 > 激活阈值 1，所以这个神经细胞将输出 1。

表 7.2　神经细胞激励值的求和计算过程

输　　入	权　　重	输入×权重	运行后总和
1	0.5	0.5	0.5
0	-0.2	0	0.5
1	-0.3	-0.3	0.2
1	0.9	0.9	1.1
0	0.1	0	1.1

7.3.2　神经细胞的用途

大脑里的生物神经细胞和其他的神经细胞是相互连接在一起的。为了创建一个人工神

经网络，人工神经细胞也要以同样方式相互连接在一起。为此可以有许多不同的连接方式，其中最容易理解并且也是最广泛地使用的，就是如图 7.5 所示那样，把神经细胞一层一层地连结在一起。这一种类型的神经网络称为前馈网络（feedforward network）。这是因为网络的每一层神经细胞的输出都向前馈送（feed）到它们的下一层（在图中是画在它的上面的那一层），直至获得整个网络的输出。

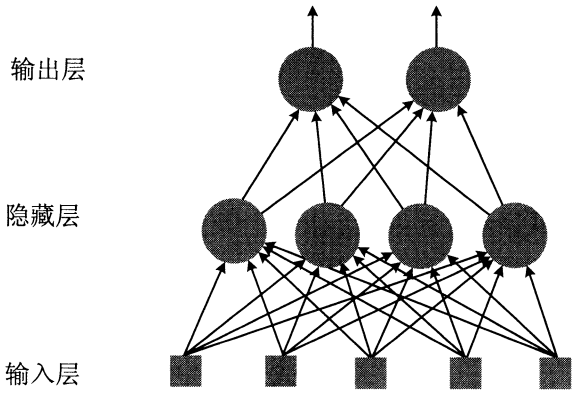


图 7.5　一个前馈网络

由图可知，网络共有 3 层[①]。输入层中的每个输入都馈送到了隐藏层，作为该层每一个神经细胞的输入；然后，从隐藏层的每个神经细胞的输出都连到了它下一层（即输出层）的每一个神经细胞。图中仅仅画了一个隐藏层，作为前馈网络，一般地可以有任意多个隐藏层。但在处理的大多数问题时一层通常是足够的。事实上，有一些问题甚至根本不需要任何隐藏单元，只需将那些输入直接连接到输出神经细胞就行了。图 7.5 选择的神经细胞的个数也是完全任意的。每一层实际都可以有任何数目的神经细胞，这完全取决于要解决的问题的复杂性。但神经细胞数目愈多，网络的工作速度也就愈低，由于其他一些原因，网络的规模总是要求保持尽可能的小。

神经网络常常用来作模式识别，这是因为它们善于把一种输入状态（它所企图识别的模式）映射到一种输出状态（它曾被训练用来识别的模式）。

这里以字符识别为例。设想有一个由 8×8 个格子组成的一块面板。每一个格子里放了一个小灯，每个小灯都可独立地被打开（格子变亮）或关闭（格子变黑），这样面板就可以用来显示 10 个数字符号。图 7.6 显示了数字“4”。

要实现这个问题，必须设计一个神经网络，它接收面板的状态作为输入，然后输出一个 1 或 0：输出 1 代表 ANN 确认已显示了数字“4”，而输出 0 表示没有显示“4”。因此，神经网络需要有 64 个输入（每一个输入代表面板的一个具体格点）和由许多神经细胞组成的一个隐藏层，还有仅有一个神经细胞的输出层，隐藏层的所有输出都馈送到它。

一旦神经网络体系创建成功后，必须接受训练来认出数字“4”。可采用这样一种方法：先把神经网络的所有权重初始化为任意值。然后给它一系列的输入，在本例中，就是代表

[①] 译者注：输入层不是神经细胞，神经细胞只有两层。

面板不同配置的输入。对每一种输入配置，检查它的输出是什么，并调整相应的权重。如果送给网络的输入模式不是“4”，网络应该输出一个 0。因此每个非“4”字符时的网络权重应进行调节，使得它的输出趋向于 0。当代表“4”的模式输送给网络时，则应把权重调整到使输出趋向于 1。

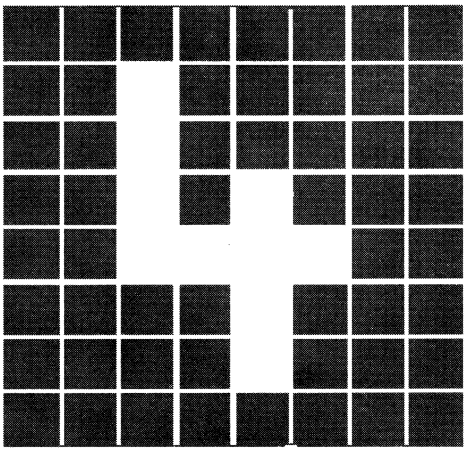


图 7.6　用于字符显示的矩阵格点

然后通过训练，就可以使网络能识别 0～9 的所有数字。还可以进一步增加输出，使网络能识别字母表中的全部字符。这实质上就是手写体识别的工作原理。对每个字符，网络都需要接受许多训练，使它认识此文字的各种不同的版本。最后，网络不但能认识已经训练的笔迹，还可以具有显著的归纳和推广能力。如果所写文字换了一种笔迹，它和训练集中所有字迹都略有不同，网络仍然有很大几率来认出它。正是这种归纳和推广能力，使得神经网络已经成为能够用于无数应用的一种无价的工具，从人脸识别、医学诊断，直到跑马赛的预测，另外还有电脑游戏中的 bot（作为游戏角色的机器人）的导航，或者机器人（robot）的导航。

这种类型的训练称为有监督的学习（supervised learning），用来训练的数据称为训练集（training set）。调整权重可以采用许多不同的方法。对本类问题最常用的方法是反向传播（backpropagation，简称 backprop 或 BP）方法。有关反向传播问题，将在训练神经网络来识别鼠标走势时进行讨论。下面讲述一种不需要任何导师来监督的训练，即无监督学习（unsupervised learning）。

7.4　扫雷机游戏

这里介绍的第一个完整例子，是如何使用神经网络来控制具有人工智能的扫雷机的行为。扫雷机工作在一个很简单的环境中，里面只有扫雷机以及随机散布的许多地雷。

在图 7.7 中，这些图形显示为黑白色，但当运行程序时性能最好的扫雷机将显现为红色。地雷显示为一些小方形。工程的目标是创建一个网络，它能够自己进行演化（evolve）

去寻找地雷。为了实现这一功能，网络的权重将被编码到基因组中，并用一个遗传算法来演化它们。

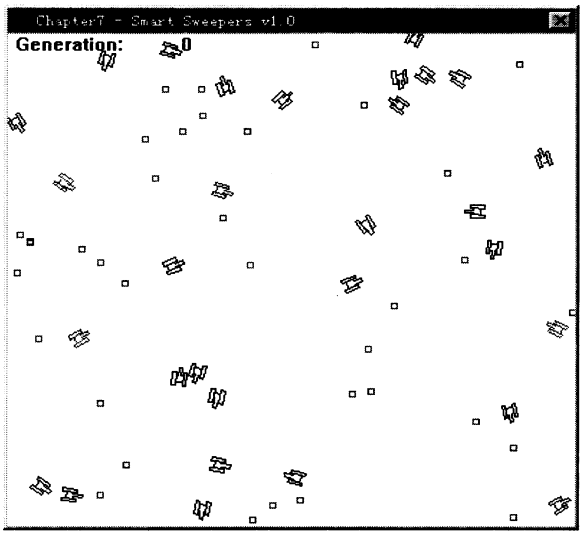


图 7.7　运行中的演示程序

提示：如果不了解怎样使用遗传算法，在进一步阅读下面的内容之前，你应学习一下有关遗传算法的内容。

下面解释人工神经网络（ANN）的体系结构，首先需要决定输入的数目、输出的数目，还有隐藏层和每个隐藏层中隐藏单元的数目。

7.4.1　选择输出

可以把扫雷机想象成与坦克车一样，通过左右两个能转动的履带式轮轨（track）来行动，如图 7.8 所示。

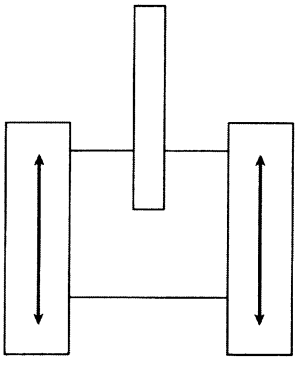


图 7.8　扫雷机的控制

扫雷机向前行进的速度，以及向左、向右转弯的角度，都是通过改变两个履带轮的相对速度来实现的。因此，神经网络需要两个输入，一个为左侧履带轮的速度，另一个为右

侧履带轮的速度。

但是……，读者也许要问，如果网络只能输出一个 1 或一个 0，我们怎么能控制车轨移动的快慢呢？你是对的；的确，利用以前描述的阶跃函数来决定输出，就根本无法控制扫雷机实际移动。如果把激励函数的输出由阶跃式改变成为在 0～1 之间连续变化的形式，就可以供扫雷机神经细胞使用了。有几种函数都能做到这一点，这里使用一个被称为逻辑斯蒂 S 形函数（logistic sigmoid function）[①]。该函数所实现的功能，从本质上说，就是把神经细胞原有的阶跃式输出曲线钝化为一个光滑的曲线，后者绕 y 轴 0.5 处点对称[②]，如图 7.9 所示。

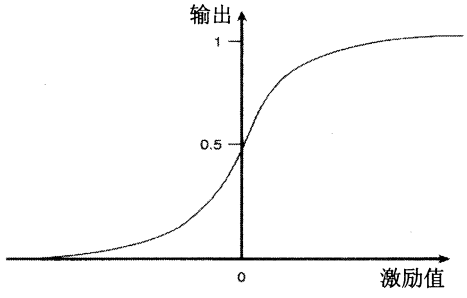


图 7.9　S 形曲线

当神经细胞的激励值趋于正、负无穷时，S 形函数分别趋于 1 或 0 。负激励值对应的函数值都小于 0.5；正激励值对应的函数值都大于 0.5。S 形函数用数学表达式写出来则为：

$$output = \frac{1}{1+e^{-a/p}}$$

其中 e 为数学常数，近似等于 2.7183，a 是神经细胞的激励值，它是函数的自变量，而 p 是一个用来控制曲线形状变化快慢或陡峭性的参数。p 通常设定为 1。当 p 赋以较大值时，曲线就显得平坦，反之，就会使曲线变得陡峭。如图 7.10 所示。较小的 p 值所生成的函数就与阶跃函数近似。P 值的大小用来控制何时使神经网络由低变高开始翻转有很大作用，但是在本例子中我们将它保持为 1。

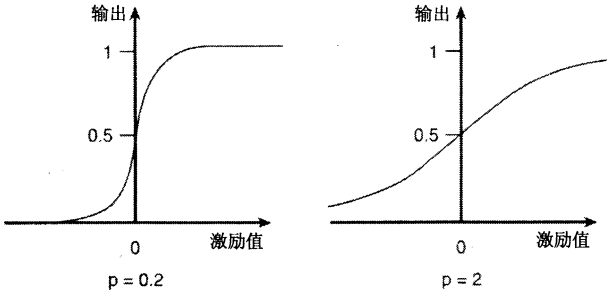


图 7.10　不同的 S 形响应曲线

[①] 译者注：逻辑斯蒂有“计算的”或“数理逻辑的”等意思在内，和“逻辑的”(logic)意义不同。

[②] 译者注：点对称图形绕对称点转 180° 后能与原图重合。若 f(x)以原点为点对称，则有 f(-x)=-f(x)。

注意：“S 型”的英文原名 Sigmoid 或 Sigmoidal 是根据希腊字 Sigma 得来的，它也可以说成是曲线的一种形状。

7.4.2 选择输入

设置完输出后，应设置网络需要的输入。设想一下扫雷机的具体细节，即应需要什么样的信息才能使它朝地雷前进。第一个输入信息清单如下：

- 扫雷机的位置（$x1$,$y1$）。
- 与扫雷机最靠近的地雷的位置（$x2$,$y2$）。
- 代表扫雷机前进方向的向量（$x3$,$y3$）。

这样一共得到 6 个输入。但是，要网络使用这些输入，工作起来就非常困难，因为，网络在像我们希望的那样执行工作之前，必须寻找所有 6 个输入之间的数学关系，而这有相当的工作量。可以把此作为一个练习倒是理想的：去试试如何给出最少数量的输入而仍能为网络传达解决问题所需要的全部信息。网络使用的输入愈少，网络所要求的神经细胞数目也愈少。而较少的神经细胞就意味更快速的训练和更少的计算，有利于网络更高速度的工作。

只要稍作一些额外考虑，就能够把输入的个数将原有的 6 减少为 4，这就是如图 7.11 所示的两个向量的 4 个参数。

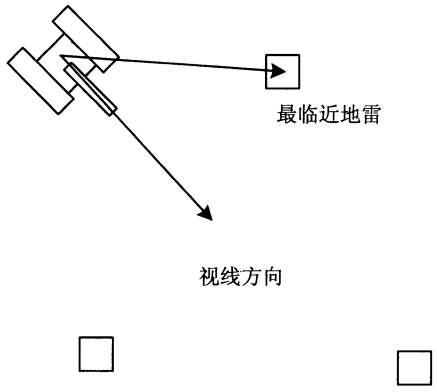


图 7.11 选择输入

把神经网络的所有输入进行规范化是一种好方法。并不是说每个输入都要改变大小使它们都在 0～1 之间，而是说每一个输入应该受到同等重视。以扫雷机输入为例，瞄准向量或视线向量（look-at vector）总是一个规范化向量，即长度等于 1，分量 x 和 y 都在 0～1 之间。但从扫雷机到达其最近地雷的向量就可能很大，其中的一个分量甚至有可能和窗体的宽度或高度一样大。如果这个数据以它的原始状态输入到网络，网络对有较大值的输入将显得更灵敏，由此就会使网络性能变差。因此，在信息输入到神经网络之前，数据应预先定比（scaled）和标准化（standardized），使它们大小相似（similar）。在本例中，由扫雷机引到与其最接近地雷的向量需要进行规范化（normalized），这样可以使扫雷机的性能得到改良。

技巧：如果将输入数据重新换算（rescale）一下，使它以 0 点为中心，就能从神经网络获得最好的性能。这一小技巧值得一试。但我在扫雷机工程中没有采用这一方法，这是因为我想使用一种更直觉的方法。

7.4.3　确定隐藏的神经细胞数目

到此我们已把输入、输出神经细胞的数目和种类确定下来了，下一步是确定隐藏层的数目，并确定每个隐藏层中神经细胞必须有多少？但遗憾的是，还没有一种确切的规则可用来计算隐藏层的数目以及每个层中神经细胞必须有多少，主要还要靠自己的不断尝试和从失败中获得经验。但遇到的大多数问题都只要用一个隐藏层就能解决。问题的关键在于如何为这一隐藏层确定最合适的神经细胞数目。数目少的神经细胞能够造就快速的网络。通常为了确定出一个最优总数，我总是在隐藏层中采用不同数目的神经细胞来进行试验。我在本章所编写的神经网络工程的第一版本中一共使用了 10 个隐藏神经细胞，应围绕这个数字的附近来调试，并观察隐藏层神经细胞的数目对扫雷机的演化会产生什么样的影响。具体程序可以在本书所附光盘的 Chapter7/Smart Sweepers v1.0 文件夹中找到，它们为即将介绍的所有程序的源码。

注意：本章代码不使用以前用到的由 #define 语句定义的常量。这是因为这里描述的程序非常复杂，包含大量需要演化的参数。由此在这里使用一个由 static 型成员变量组成的类代替一系列的#define 语句。当这一个类的一个实例被创建时，它的各成员变量将由一个名为“ini”的文件自动初始化。采用这样的方式来实现初始化，可以省去重新编译的时间，不论参数是否已经改变。所要做的全部工作只是修改 ini 文件。这一个类的名称是 CParams，ini 文件的名称为 params.ini。如果需要获得更清晰的了解，可以参看光盘上这一个类的代码。

7.4.4　CNeuralNet.h（神经网络类的头文件）

在 CNeuralNet.h 神经网络类的头文件中，定义了人工神经细胞的结构、人工神经细胞的层的结构以及人工神经网络本身的结构。首先介绍人工神经细胞的结构。

7.4.4.1　SNeuron（神经细胞的结构）

人工神经细胞的结构中必须有一个正整数记录它有多少输入，还需要有一个向量 std:vector 来表示它的权重。而且，神经细胞的每一个输入都应有一个对应的权重。

```
Struct SNeuron
{
    // 进入神经细胞的输入个数
    int m_NumInputs;

    // 为每一个输入提供的权重
    vector<double> m_vecWeight;
```

```
    //构造函数
    SNeuron (int NumInputs);
};
```

SNeuron 结构体的构造函数形式如下：

```
SNeuron::SNeuron(int NumInputs): m_NumInputs(NumInputs+1)
{
    // 要为偏移值也附加一个权重,因此应将输入数目加 1
    for (int i=0; i<NumInputs+1; ++i)
    {
        // 将权重初始化为任意的值
        m_vecWeight.push_back(RandomClamped());
    }
}
```

由上可知，构造函数把送进神经细胞的输入数目 NumInputs 作为一个变元，并为每个输入创建一个随机的权重。所有权重值在-1 和 1 之间。

这一个附加的权重十分重要。下面结合一些数学知识解释为什么应附加一个权重。激励值是所有输入x权重的乘积的总和，而神经细胞的输出值取决于这个激励值是否超过某个阈值(t)。可以用如下的方程表示：

$$w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n \geqslant t$$

上式是使细胞输出为 1 的条件。因为网络的所有权重需要不断演化（进化），阈值的数据也应做相应的演化。应将阈值转换为权重的形式。从上面的方程两边各减去 t，得：

$$w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n - t \geqslant 0$$

这个方程可以再用另一种形式写出来，如下：

$$w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + t(-1) \geqslant 0$$

阈值 t 可以看作一个始终乘以输入为-1 的权重。这个特殊的权重通常称偏移（bias），这就是为什么每个神经细胞初始化时都要增加一个权重的原因。演化一个网络时，就不必再考虑阈值问题，因为它已被内建在权重向量中。带偏移的人工神经细胞的结构如图 7.12 所示。

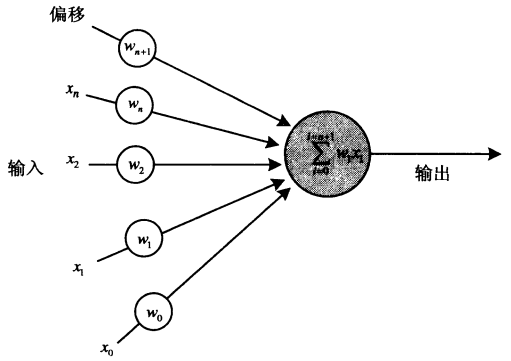


图 7.12　带偏移的人工神经细胞

7.4.4.2　SNeuronLayer（神经细胞层的结构）

神经细胞层的结构中定义了一个如图 7.13 所示的由虚线包围的神经细胞所组成的层。

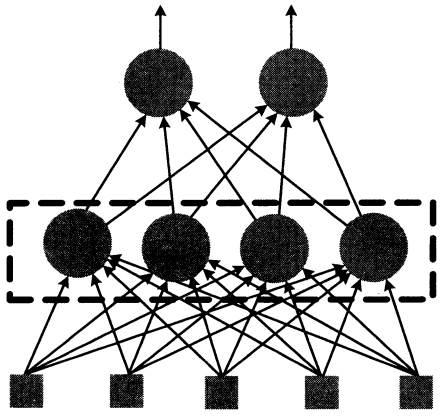


图 7.13　一个神经细胞层

层定义的源代码如下：

```
struct SNeuronLayer
{
  // 本层使用的神经细胞数目
int              m_NumNeurons;

  // 神经细胞的层
vector<SNeuron>   m_vecNeurons;

SNeuronLayer (int NumNeurons, int NumInputsPerNeuron);
};
```

7.4.4.3　CNeuralNet（神经网络类）

神经网络类用于创建神经网络对象。其定义如下：

```
class CNeuralNet
{
private:
   int                    m_NumInputs;

   int                    m_NumOutputs;

   int                    m_NumHiddenLayers;

   int                    m_NeuronsPerHiddenLyr;

   // 为每一层（包括输出层）存放所有神经细胞的存储器
   vector<SNeuronLayer> m_vecLayers;
```

所有 private 成员均是顾名思义的。需要由本类定义的是输入的个数、输出的个数、隐藏层的数目以及每个隐藏层中神经细胞的个数等参数。

```
public:

    CNeuralNet ();
```

该构造函数利用 ini 文件初始化所有的 Private 成员变量，然后再调用 CreateNet 创建网络。

```
// 由 SNeurons 创建网络
void CreateNet ();

// 从神经网络得到（读出）权重
vector<double>    GetWeights ()const;
```

由于网络的权重需要演化，所以必须创建一个方法来返回所有的权重。这些权重在网络中是以实数型向量形式表示的，这些实数表示的权重将被编码到一个基因组中。谈论本工程的遗传算法时，将确切说明权重如何进行编码。

```
// 返回网络的权重的总数
int GetNumberOfWeights()const;

// 用新的权重代替原有的权重
void PutWeights(vector<double> &weights);
```

这一函数所做的工作与函数 GetWeights 所做的相反。当遗传算法执行完一代时，新一代的权重必须重新插入神经网络。完成这一任务的为 PutWeight 方法。

```
// S 形响应曲线
inline double  Sigmoid(double activation, double response);
// 当已知神经细胞所有输入×权重的乘积之和时，这一方法将它送入 S 形的激励函数
// 根据一组输入计算输出
vector<double> Update (vector<double> &inputs);

}; // 类定义结束
```

1．CNeuralNet::CreateNet（神经网络类::创建神经网络的方法）

下面是 CNeuralNet 的两种方法的更完整的代码。第一种是网络创建方法 CreateNet。它的工作就是把由细胞层 SNeuronLayers 所收集的神经细胞 SNeurons 聚在一起来组成整个神经网络，代码如下：

```
void CNeuralNet::CreateNet ()
{
  // 创建网络的各个层
  if (m_NumHiddenLayers > 0)
   {
```

```
    //创建第一个隐藏层①
    m_vecLayers.push_back(SNeuronLayer (m_NeuronsPerHiddenLyr,
                                        m_NumInputs));

    for( int i=0; i<m_NumHiddenLayers-1; ++i)
    {
      m_vecLayers.push_back(SNeuronLayer (m_NeuronsPerHiddenLyr,
                                          m_NeuronsPerHiddenLyr));
    }
    //创建输出层
    m_vecLayers.push_back( SNeuronLayer (m_NumOutput,
                                         m_NeuronsPerHiddenLyr));
    }
  else //无隐藏层时,只需创建输出层
    {
    //创建输出层
     m_vecLayers.push_back(SNeuronLayer (m_NumOutputs,m_NumInputs));
   }
}
```

2．CNeuralNet::Update（神经网络的更新方法）

Update 函数（更新函数）可称为神经网络的“主要劳动力”。这里，输入网络的数据 input 是以双精度向量 std::vector 的数据格式传递进来的。Update 函数通过对每个层的循环来处理输入×权重的相乘与求和，再以所得的和数作为激励值，通过 S 形函数来计算出每个神经细胞的输出。Update 函数返回的也是一个双精度向量 std::vector，它对应的就是人工神经网络的所有输出。

下面为 Update 函数的代码：

```
vector<double> CNeuralNet::Update (vector<double> &inputs)
{
  // 保存从每一层产生的输出
  vector<double> outputs;

  int cWeight = 0;

  // 首先检查输入的个数是否正确
  if (inputs.size() != m_NumInputs)
    {
      // 如果不正确,就返回一个空向量
      return outputs;
  }

    // 对每一层,……
    for (int i=0; i<m_NumHiddenLayers+1; ++i)
```

① 译者注：如果允许有多个隐藏层，则由 for 循环创建其余的隐藏层。

```
    {
      if (i>0)
        {
          inputs = outputs;
        }
      outputs.clear();

      cWeight = 0;

      // 对每个神经细胞,求输入×对应权重乘积的总和。并将总和赋给S形函数
      // 以计算输出
       for (int j=0; j<m_vecLayers[i].m_NumNeurons; ++j)
        {
         double netinput = 0;

         intNumInputs = m_vecLayers[i].m_vecNeurons[j].m_NumInputs;

         // 对每一个权重
         for (int k=0; k<NumInputs-1; ++k)
         {
         // 计算权重×输入的乘积的总和
          netinput += m_vecLayers[i].m_vecNeurons[j].m_vecWeight[k]
                                        *inputs[cWeight++];
         }

         // 加入偏移值
         netinput+= m_vecLayers[i].m_vecNeurons[j].
                    m_vecWeight[NumInputs-1] * CParams::dBias;
```

每个神经细胞的权重向量的最后一个权重实际是偏移值，一般将它设置为–1。ini 文件中已经包含了偏移值，可以改变它，考察它对创建的网络的功能有什么影响。不过，这个值通常是不应该改变的。

```
    // 每一层的输出产生之后,就应将它们保存起来
    // 但用Σ累加在一起的激励总值首先要通过S形函数的过滤,才能得到输出
    outputs.push_back(Sigmoid(netinput,CParams::
                              dActivationResponse));
        cWeight = 0;
      }
    }
    return outputs;
  }
```

7.4.5 神经网络的编码

这里介绍一个用实数编码的具体例子。为了设计一个前馈型神经网络，编码是很容易的。先从左到右读每一层神经细胞的权重，读完第一个隐藏层，再向上读它的下一层，把

所读到的数据依次保存到一个向量中，这样就实现了网络的编码。因此，如果有图 7.14 所示的网络，则它的权重编码向量将为：

0.3, -0.8, -0.2, 0.6, 0.1, -0.1, 0.4, 0.5

在这一网络中，为了简单起见，并没有把偏移值的权重包括进去。但在实际实现编码时，必须包含偏移值这个权重，否则无法获得所需要的结果。

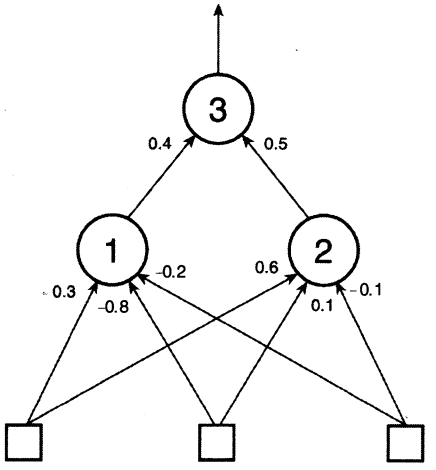


图 7.14　为权重编码

下面讲述如何用遗传算法来操纵已编码的基因。

7.4.6　遗传算法

至此，所有的权重已经像二进制编码的基因组那样，形成了一个串，就可以应用遗传算法。遗传算法（GA）是在扫雷机已被允许按照用户指定的帧数（下文将帧数称为滴答数，英文为 ticks）运转后执行的。可以在 ini 文件中找到这个滴答数（iNumTicks）的设置。

下面是基因组结构体的代码。

```
Struct SGenome
 {
    vector <double>    vecWeights;
    double             dFitness;
    SGenome():dFitness(0) {}
    SGenome(vector<double>w,double f):vecWeights(w),dFitness(f){}

    //重载'<'的排序方法
     friend bool  operator<(const SGenome& lhs, const SGenome& rhs)
       {
           return (lhs.dFitness < rhs.dFitness);
       }
};
```

从上面的代码可以看出，这一 SGenome 结构和其他的 SGenome 结构几乎完全一致，惟一的差别在于这里的染色体是一个双精度向量 std::vector。因此，可以和通常一样来应用杂交操作和选择操作。但突变操作则稍微有些不同，这里的权重值是用一个最大值为 dMaxPerturbation 的随机数来干扰的。这一参数在 ini 文件中已作了声明。另外，作为浮点数遗传算法，突变率也被设定得更高一些。在本程序中，它被设为 0.1。

下面为扫雷机游戏遗传算法类中的突变函数的形式：

```
void CGenAlg::Mutate(vector<double> &chromo)
{
  // 遍历权重向量,按突变率将每一个权重进行突变
  for (int i=0; i<chromo.size(); ++i)
  {
    // 是否干扰这个权重
    if (RandFloat() < m_dMutationRate)
    {
     // 为权重增加或减小一个小的数量
     chromo[i] += (RandomClamped() * CParams::dMaxPerturbation);
    }
  }
}
```

如同以前的工程那样，笔者已为 v1.0 版本的 Smart Minesweepers 工程保留了一个非常简单的遗传算法。这样就能留下许多余地，可以利用以前学到的技术来改进它。如同大多数别的程序一样，v1.0 版只用轮盘赌方式选拔精英，并采用单点式杂交。

注意：当程序运行时，权重可以被演化成为任意的大小，它们不受任何形式的限制。

7.4.7 扫雷机类

这一个类用来定义一个扫雷机。与登月飞船类一样，扫雷机类中有一个包含了扫雷机位置、速度以及如何转换方向等数据的记录。类中还包含扫雷机的视线向量(look-at vector)；它的两个分量被用来作为神经网络的两个输入。这是一个规范化的向量，它是在每一帧中根据扫雷机本身的转动角度计算出来的，并且指示了扫雷机当前是朝哪一个方向，如图 7.11 所示。

下面为扫雷机类 CMinesweeper 的定义：

```
class CMinesweeper
{
private:
    // 扫雷机的神经网络
    CNeuralNet      m_ItsBrain;

    // 扫雷机在世界坐标里的位置
    SVector2D       m_vPosition;
    // 扫雷机面对的方向
```

```
SVector2D      m_vLookAt;

// 扫雷机的旋转
double         m_dRotation;

double         m_dSpeed;

// 根据 ANN 保存输出
double         m_lTrack,
               m_rTrack;
```

m_lTrack 和 m_rTrack 根据网络保存当前帧的输出。这些就是用来决定扫雷机的移动速率和转动角度的数值。

```
// 用于度量扫雷机适应性的分数
double         m_dFitness;
```

每当扫雷机找到一个地雷，它的适应性分数就要增加。

```
// 扫雷机被画出时的大小比例
double         m_dScale;

// 扫雷机最邻近地雷的下标位置
int            m_iClosestMine;
```

在控制器类 CControl1er 中，有一个属于所有地雷的成员向量 std::vector。
而 m_iClosestMine 就是代表最靠近扫雷机的那个地雷在该向量中的位置的下标值。

```
public:

    CMinesweeper ();

    // 利用从扫雷机环境得到的信息来更新人工神经网络
    bool Update (vector<SVector2D> &mines);

    // 用来对扫雷机各个顶点进行变换,以便下一步可以画出扫雷机
    void  WorldTransform(vector<SPoint> &sweeper);

    // 返回一个向量到最邻近的地雷
    5Vector2D GetClosestMine(vector<SVector2D> &objects);

    // 检查扫雷机看它是否已经发现地雷
    int       CheckForMine(vector<SVector2D> &mines, double size);

    void      Reset();

    // ----------------- 定义各种供访问用的函数
    SVector2D Position()const { return m_vPosition; }
    void      IncrementFitness(double val) { m_dFitness += val; }
    double    Fitness()const { return m_dFitness; }
```

```
    void      PutWeights(vector<double> &w)
    { m_ItsBrain.PutWeights(w); }
    int       GetNumberOfWeights()const
    {return m_ItsBrain.GetNumberOfWeights(); }
};
```

需要更详细地给予说明的 CMinesweeper 类的方法只有一个，即 Update 更新函数。该函数在每一帧中都要被调用，以更新扫雷机神经网络。下面为这个函数的定义：

```
bool CMinesweeper::Update (vector<SVector2D> &mines)
{
    //这一向量用来存放神经网络所有的输入
    vector<double> inputs;

    //计算从扫雷机到与其最接近的地雷（两个点）之间的向量
    SVector2D vClosestMine = GetClosestMine(mines);

    //将该向量规范化
    Vec2DNormalize(vClosestMine);
```

首先，该函数计算了扫雷机到与其最接近的地雷之间的向量，然后使它规范化（记住，向量规范化后它的长度等于 1）。但扫雷机的视线向量（look-at vector）不需要再作规范化，因为它的长度已经等于 1 了。由于两个向量都有效地转化为同样的大小范围，可以认为输入已经得到标准化。

```
    //加入扫雷机到最近地雷之间的向量
    Inputs.push_back(vClosestMine.x);
    Inputs.push_back(vCIosestMine.y);

    //加入扫雷机的视线向量
    Inputs.push_back(m_vLookAt.x);
    Inputs.push_back(m_vLookAt.y);

    //更新大脑，并从网络得到输出
    vector<double> output = m_ItsBrain.Update (inputs);
```

然后把视线向量以及扫雷机与它最接近的地雷之间的向量都输入到神经网络。函数 CNeuralNet::Update 利用这些信息来更新扫雷机网络，并返回一个 std::vector 向量作为输出。

```
    //保证在输出的计算中没有发生错误
    if (output.size() < CParams::iNumOutputs)
    {
        return false;
    }

    // 把输出赋值到扫雷机的左、右履带轮轨
    m_lTrack = output[0];
    m_rTrack = output[1];
```

在更新神经网络时，若检测到确实没有错误，程序把输出赋给 m_lTrack 和 m_rTrack。这些值代表施加到扫雷机左、右履带轮轨上的力。

```
// 计算驾驶的力
double RotForce = m_lTrack - m_rTrack;

// 进行左转或右转
Clamp(RotForce,-CParams::dMaxTurnRate, CParams::dMaxTurnRate);
 m_dSpeed = (m_lTrack + m_rTrack);
```

扫雷机车的转动力是利用施加到它左、右履带轮轨上的力之差来计算的。并规定，施加到左轨道上的力减去施加到右轨道上的力，就得到扫雷机车辆的转动力。然后就把此力施加给扫雷机车，使它实行不超过 ini 文件所规定的最大转动率的转动。而扫雷机车的行进速度即为它的左侧轮轨速度与它的右侧轮轨速度的和。确定了扫雷机的转动力和速度，它的位置和偏转角度就可以相应地更新。

```
//更新扫雷机左右转向的角度
m_dRotation += RotForce;

// 更新视线角度
m_vLookAt.x = -sin(m_dRotation);
m_vLookAt.y = cos(m_dRotation);

// 更新扫雷机的位置
m_vPosition += (m_vLookAt* m_dSpeed);

// 如果扫雷机到达窗体四周,则让其进行环绕,使它不至于离开窗体而消失
If (m_vPosition.x > CParams::WindowWidth) m_vPosition.x = 0;
If (m_vPosition.x < 0) m_vPosition.x = CParams::WindowWidth;
If (m_vPosition.y > CParams::WindowHeight) m_vPosition.y = 0;
If (m_vPosition.y < D) m_vPosition.y = CParams::WindowHeight;
```

为了简单起见，程序设置扫雷机在碰到窗体边框时就环绕折回（wrap）。采用这种方法程序就不再需要做任何碰撞—响应方面的工作。

```
    Returen true;
}
```

7.4.8　CController Class（控制器类）

CController 类是和一切都有联系的类。图 7.15 显示了其他的各个类和 CController 类的关系。

下面为这个类的定义：

```
class CController
{
private:
```

```
// 基因组群体的动态存储器（一个向量）
 vector<SGenome>          m_vecThePopulation;
```

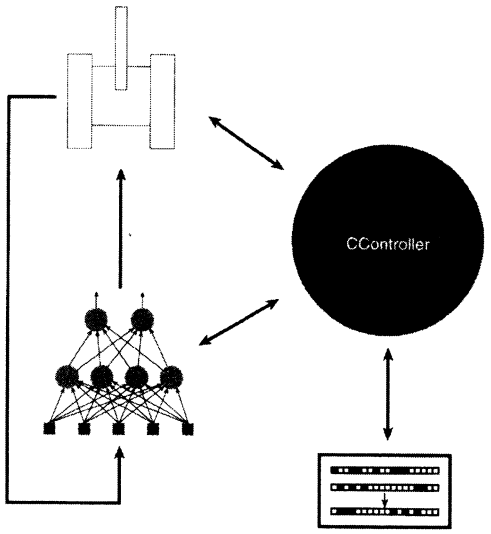


图 7.15　扫雷机游戏的程序流程图

```
// 保存扫雷机的向量
vector<CMinesweeper> m_vecSweepers;

// 保存地雷的向量
vector<SVector2D>          m_vecMines;

// 指向遗传算法对象的指针
CGenAIg*                   m_pGA;

int                        m_NumSweepers;

int                        m_NumMines;

// 神经网络中使用的权重值的总数
int                        m_NumWeightsInNN;

// 存放扫雷机形状各顶点的缓冲区
vector<SPoint>             m_SweeperVB;

// 存放地雷形状各顶点的缓冲区
vector<SPoint>             m_MineVB;

// 存放每一代的平均适应性分数，供绘图用
vector<double>             m_vecAvFitness;

// 存放每一代的最高适应性分数
```

```
    vector<double>          m_vecBestFitness;

    // 使用的各种不同类型的画笔
    HPEN                    m_RedPen;
    HPEN                    m_BluePen;
    HPEN                    m_GreenPen;
    HPEN                    m_OldPen;

    // 应用程序窗口的句柄
    HWND                    m_hwndMain;

    // 切换扫雷机程序运行的速度
    bool                    m_bFastRender;

    // 每一代的帧数（滴答数）
    int                     m_iTicks;

    // 代的计数
    int                     m_iGenerations;

    // 窗体客户区的大小
    int                     cxClient, cyClient;

    // 本函数在运行过程中画出具有平均值和最优适应性值的图
    void                    PlotStats(HDC surface);

public:

    CController (HWND hwndMain);

    ~CController ();

    void          Render(HDC surface);

    void          WorldTransform(vector<SPoint> &VBuffer,
                                 SVector2D      vPos);

    bool          Update ();

    // 几个公用的访问方法
    bool      FastRender() { return m_bFastRender; }
    void      FastRender(bool arg){ m_bFastRender = arg; }
    void      FastRenderToggle() { m_bFastRender = !m_bFastRender; }
};
```

当创建 CController 类的某个实例时，会有一系列的事情发生：

- 创建 CMinesweeper 对象。

- 统计神经网络中所使用的权重的总数，然后此数字即被利用来初始化遗传算法类的一个实例。
- 从遗传算法对象中随机提取染色体（权重）并插入到扫雷机的神经网络中。
- 创建大量的地雷并被随机地散播到各地。
- 为绘图函数创建所有需要用到的 GDI 画笔。
- 为扫雷机和地雷的形状创建顶点缓冲区。

所有的一切现都已完成初始化，由此 Update 方法就能在每一帧中被调用来对扫雷机进行演化。

控制器更新方法 CController::Update 方法（或函数）在每一帧中都要被调用。当调用 Update 函数时，函数的前一半通过对所有扫雷机进行循环，如发现某一扫雷机找到了地雷，就更新该扫雷机的适应性分数。由于 m_vecThePopulation 包含了所有基因组的备份，相关的适应性分数也要在这时进行调整。如果为完成一个代（generation）所需要的帧数均已通过，本方法就执行一个遗传算法的时代（epoch）来产生新一代的权重。这些权重被用来代替扫雷机神经网络中原有的旧的权重，使扫雷机的每一个参数被重新设置，从而为进入新一代做好准备。

```
bool CController::Update ()
{
    // 扫雷机执行一循环,总数为 CParams::iNumTicks 次
    // 在此循环中,每个扫雷机的神经网络不断利用它周围的环境信息进行更新
    // 而从神经网络得到的输出使扫雷机实现所需要的动作
    // 如果扫雷机遇见了一个地雷,则它的适应性分数将相应地被更新
    // 同样地更新它对应基因组的适应性分数
    if (m_iTicks++ < CParams::iNumTicks)
      {
       for (int i=0; i<m_NumSweepers; ++i)
        {
       //更新神经网络和位置
       if (!m_vecSweepers[i].Update (m_vecMines))
        {
        //处理神经网络时出现了错误,显示错误后退出
          MessageBox(m_hwndMain, 'Wrong amount of NN inputs!",
                     "Error", MB_OK);
          return false;
        }

       // 检查这一扫雷机是否已经发现地雷
       int GrabHit = m_vecSweepers[i].CheckForMine(m_vecMines,
                                                   CParams::dMineScale);
       if (GrabHit >= 0)
       {
         // 扫雷机已找到了地雷,并增加它的适应性分数
         m_vecSweepers[i].IncrementFitness();
```

```
            // 去掉被扫雷机找到的地雷,用在随机位置放置的一个新地雷来代替
            m_vecMines[GrabHit] = SVector2D(RandFloat() * cxClient,
                                            RandFloat() * cyClient);
          }
          // 更新基因组的适应性值
          m-vecThePopulation[i].dFitness = m_vecSweepers[i].Fitness();
        }
      }
      // 完成又一个代
      // 以下的程序为运行遗传算法并用它们新的神经网络更新扫雷机
      else
      {
        // 更新状态窗口中的状态
        m_vecAvFitness.push_back(m_pGA->AverageFitness());
        m_vecBestFitness.push_back(m_pGA->BestFitness());

        // 增加代计数器的值
        ++m_iGenerations;

        // 将帧计数器复位
        m_iTicks = 0;

        // 运行遗传算法创建一个新的群体
        m-vecThePopulation = m_pGA->Epoch(m_vecThePopulation);

        // 在各扫雷机中重新插入新的（有希望）被改进的大脑
        // 并将它们的位置进行复位等
        for(int i=0; i<m_NumSweepers; ++i)
        {m_vecSweepers[i].m_ItsBrain.PutWeights(m_vecThePopulation
                                                [i].vecWeights);
         m_vecSweepers[i].Reset();
        }
      }
      return true;
    }
```

概括起来，程序为每一时代（epoch）做的工作步骤如下：

1. 为所有扫雷机和 iNumTicks 个帧组织循环，调用 Update 函数并根据情况增加扫雷机适应性分数。

2. 从扫雷机神经网络提取权重向量。

3. 用遗传算法演化出一个新的网络权重群体。

4. 把新的权重插入到扫雷机神经网络。

5. 转到第 1 步进行重复，直至获得理想的性能。

表 7.3 列出了 Smart Sweepers 工程 v1.0 版所有默认参数的设置值。

7.4.9 运行此程序

运行程序时，F 键用来切换两种不同的显示状态，一种是显示扫雷机如何学习寻找地雷，另一种是显示在运行期中产生的最优的与平均的适当性分数的统计图表。

当显示图表时，程序将会加速运行。

7.4.10 功能的两个改进

尽管扫雷机学习寻找地雷的“本领”较完善，这里仍有两件事情能进一步改进扫雷机的性能。

7.4.10.1 改进之一

首先，单点 crossover 算子有许多可改进的地方。按照它的规定，算子是沿着基因组长度任意地方切开的，这样很有可能使个别神经细胞的基因组在权重的中间被一分两段。

表 7.3 Smart Sweepers v1.0 工程的默认设置

神 经 网 络	
参　数	设 置 值
输入数目	4
输出数目	2
隐藏层数目	1
隐藏层神经元数目	10
激励响应	1
遗 传 算 法	
参　数	设 置 值
群体大小	30
选择类型	旋转轮
杂交类型	单点
杂交率	0.7
突变率	0.1
精英设置（on/off）	On
精英数目（N/copies）	4/1
总 体 特 性	
参　数	设 置 值
每时代的帧数	2000

为清楚起见，对图 7.16 所示的网络权重进行分析。

在这里，杂交算子可以沿向量长度的任意一处切开，这样，就会有极大几率在某个神

经细胞（如第 2 个）的权重中间断开，也就是说，在权重 0.6 和-0.1 之间某处切开。这可能不是我们想要的，因为，如果把神经细胞作为一个完整的单元来看待，则它在此以前所获得的任何改良就会被干扰。这样的杂交操作有可能与断裂性突变（disruptive mutation）操作所起的作用非常相似。

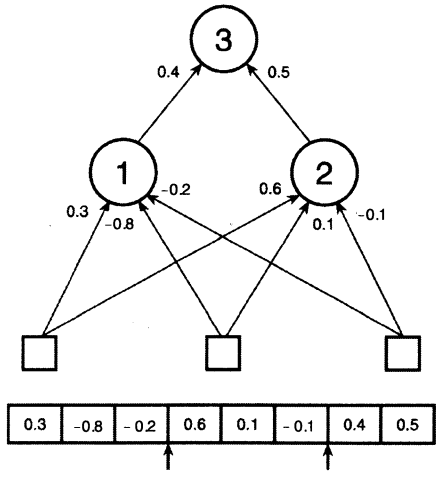


图 7.16　简单的网络

与此相反，创建另一种类型的杂交运算，它只在神经细胞的边界上切开。在图 7.16 中，就是在第 3、4 或第 6、7 的两个基因之间切开（如图中小箭头所示）。为了实现这一算法，在 CNeuralNet 类中补充了另一个切割方法：CalculateSplitPoints。这一方法创建了一个用于保存所有网络权重边界的矢量，它的代码如下：

```
vector<int> CNeuralNet::CalculateSplitPoints () const
{
  vector<int> SplitPoints;

  int WeightCounter = 0;

  // 对每一层
  for (int i=0; i<m_NumHiddenLayers + 1; ++i)
   {
    // 对每一个神经细胞
    for (int j=0; j<m_vecLayers[i].m_NumNeurons; ++j)
    {
     // 对每一个权重
     for(int k=0; k<m_vecLayers[i].m_vecNeurons[j].
                                                m_NumInputs; ++k)
       {
          ++WeightCounter;
       }
      SplitPoints.push_back(WeightCounter - 1);
     }
```

```
    }
    return SplitPoints;
}
```

这一方法是CController类构造函数在创建扫雷机并把断裂点向量传递给遗传算法类时调用的。它们被存储在一个名为 m_vecSplitPoints 的 std::vector 向量中。然后遗传算法就利用这些断裂点来实现两点杂交操作，其代码如下：

```
void CGenAlg::CrossoverAtSplits(const vector<double>  &mum,
                                const vector<double>  &dad,
                                vector<double>        &baby1,
                                vector<double>        &baby2)
{
  // 如果超过了杂交率,就不再进行杂交,把两个上代作为两个子代输出
  // 如果两个上代相同,也把它们作为两个下代输出
  if ( (RandFloat() > m_dCrossoverRate) || (mum == dad))
    {
     baby1 = mum;
     baby2 = dad;

     return;
    }

  // 确定杂交的两个断裂点
  int index1 = RandInt(0, m_vecSplitPoints.size()-2);
  int index2 = RandInt(Index1, m_vecSplitPoints.size()-1);

  int cp1 = m_vecSplitPoints[Index1];
  int cp2 = m_vecSplitPoints[Index2];

  // 创建子代
  for (int i=0; i<mum.size(); ++i)
  {
    if ( (i<cp1) || (i>=cp2) )
      {
        // 如果在杂交点外,则保持原来的基因
        baby1.push_back(mum[i]);
        baby2.push_back(dad[i]);
      }

   else
      {
        // 对中间段进行交换
        baby1.push_back(dad[1]);
        baby2.push_back(mum[1]);
      }
  }
```

```
        return;
    }
```

根据经验得知，在进行杂交时，如果把神经细胞当作一个不可分割的单位，那样比在染色体长度上任意一点分裂基因组，能得到更好的结果。

7.4.10.2　改进之二

第二种性能改进，是用另一种方式来观察网络的那些输入。在前面的例子中为网络使用了 4 个输入参数：2 个用于表示扫雷机视线方向的向量，另外 2 个用来指示扫雷机与其最靠近的地雷的方向的向量。然而，有一种办法，可以把这些参数的个数减少到只剩下一个。

扫雷机为了确定地雷的位置，只要知道从它当前的位置和朝向出发，需要向左或向右转动多大的一个角度，这一信息即可。计算出扫雷机的视线向量和从它到最邻近地雷的向量，再计算它们之间的角度（θ）就比较简单，利用这两个向量的点积就可以实现，在第 6 章“使登陆月球容易一点”中已讨论过，如图 7.17 所示。

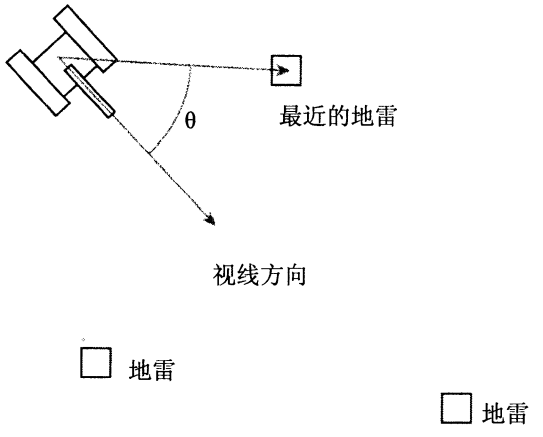


图 7.17　从视线方向转动到最邻近地雷方向的角度 θ

但点积仅仅给出角度的大小，它不能指示这一角度是在扫雷机的哪一侧。[①]因此编写了另一个向量函数返回一个向量相对于另一个向量的正负号。该函数的原型如下：

```
inline int Vec2DSign(SVector2D &v1,SVector2D &v2);
```

如果对它的原理感兴趣，可以在文件 SVector2D.h 中找到它的源码。但它的基本点在于：如果 v1 至 v2 是按顺时针方向转的，则函数返回 +1；如果 v1 至 v2 是按逆时针方向转，则函数返回-1。

把点积和 Vec2Dsign 二者结合起来，就能把输入简化，网络只需接受一个输入即可。下面为新的 CMinesweeper::Update 函数有关段落的代码：

[①] 译者注：改用叉积就可以给出大小，同时给出左右方向。

```
// 计算到最邻近地雷的向量
SVector2D vClosestMine = GetClosestMine(mines);

// 将其规范化
Vec2DNormalize(vClosestMine);

// 计算扫雷机视线向量和它到最邻近地雷的向量的点积,它给出了扫雷机要面对
// 最邻近地雷所需转动的角度
double dot = Vec2DDot(m_vLookAt, vClosestMine);

// 计算正负号
int sign = Vec2DSign(m_vLookAt, vClosestMine);

Inputs.push_back(dot*sign);
```

运行一下光盘 Chapter7/Smart Sweepers v1.1 目录下的可执行程序 executable，就可以体验经过以上两个改进，能够使演化过程提速。

需要注意的是，带有 4 个输入的网络要花很长时间进行演化，因为它必须在各输入数据之间找出更多的关系才能确定它应如何行动。网络实际就是在学习如何做点积并确定它的正负极性。因此，当设计自己的网络时，应仔细权衡一下，是由自己预先计算许多输入数据（它将使 CPU 负担增加，但导致演化时间缩短）还是让网络处理输入数据之间的复杂关系（它将使演化时间变长，但能使 CPU 减轻负担。

7.5 总　　结

我希望你已享受到了第一次攻入神经网络这一奇妙世界的快乐。我打赌你一定在为如此简单就能使用它们而感到惊讶吧，对吗？我想我是猜对了。

在下面几章里我将要向你介绍更多的知识，告诉你一些新的训练手段和演绎神经网络结构的更多的方法。但首先请你利用本章下面的提示去玩一下游戏是有意义的。

7.6 练　　习

1. 在 v1.0 中，不用 look-at 向量作为输入，而改用旋转角度 θ 作为输入，由此可以使网络的输入个数减少成为 1 个。思考这对神经网络的演化会产生什么影响，对此有什么看法？

2. 试以扫雷机的位置（$x1$,$y1$）、扫雷机最接近的地雷的位置（$x2$,$y2$）以及扫雷机前进方向的向量（$x3$,$y3$）等 6 个参数作为输入设计一个神经网络，使它仍然能够演化去寻找地雷。

3. 改变激励函数的响应，试用 0.1～0.3 之间的低端值，它将产生和阶跃函数非常相像

的一种激励函数。然后再试用高端值，它将给出较为平坦的响应曲线。思考这些改变对演化进程具有什么影响。

4．改变神经网络的适应性函数，使得扫雷机不是去扫除地雷，而是要演化它，使它能避开地雷。

5．理清有关遗传算法的各种不同设置和运算中仍不清楚的地方。

6．加入其他的对象类型，比如人。给出一个新环境来演化扫雷机，使它能避开人，但照样能扫除地雷。

第 8 章　为机器人提供知觉

盲人：我已经好了！主人已把我治愈了！
布莱恩：我没碰过他啊！
盲人：我是瞎子，但我现在能看了。哈哈！[一声轰响]

Monty Python: The Life of brian [①]

本章将讨论如何用神经网络来解决普通游戏中的两个人工智能问题：避开障碍物和探索环境。作为基础，将沿用第 7 章的代码，但在创建的游戏环境中已散布了要求扫雷机去躲避的许多障碍物。障碍物的顶点坐标存放在一个缓冲区中，可以像任何其他游戏对象一样画出来。图 8.1 显示了扫雷机的这个“新世界”。

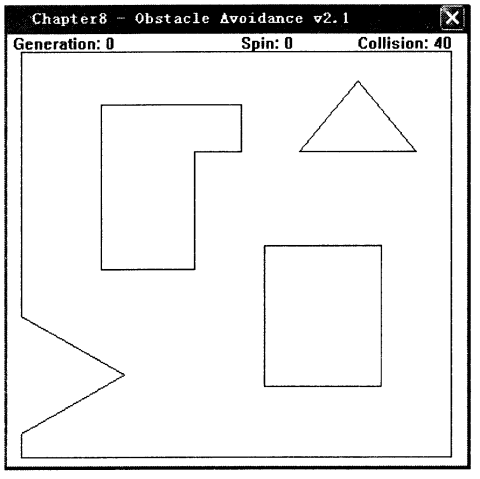


图 8.1　美丽的新世界

本章的目的，就是介绍怎样来创建许多机器人（bot），它们仍要完成原来的扫雷工作，但当它们在图 8.1 所示的新游戏世界里巡航时，必须避开所有的障碍物。下面先讨论如何使它们避免与障碍物相碰撞。

8.1　回避障碍物

回避障碍是游戏中一种非常普通的人工智能工作。这是游戏代理人（game agent）认识其周围环境，并在游戏世界中自由航行不与其他对象相碰撞的一种本领。只有极少数的游戏才有例外，它们在某种程度上可以不要求有这种能力。

[①] 译者注：The Life of brian（布莱恩的一生）由小说改编的一部美国影片，中译“万世魔星”。

要成功实现障碍物的回避，游戏代理人必须能做到：

- 考察周围的环境。
- 采取行动避免潜在的碰撞。

首先，思考如何为游戏代理人提供“视觉”（sense of vision）。

8.1.1　探测环境

直到目前为止，我们介绍的扫雷机还只有极为有限的认识能力。第 7 章所介绍的扫雷机除了能看见最邻近的那个地雷外，其余的就什么也看不见了。要使扫雷机能够避开障碍，必须为它们提供一种能力，使它们能够看见它们周围的世界。这里为每个扫雷机装上许多触觉器（sensors）。所谓触觉器，实际就是从扫雷机身上由中心向外辐射出来的几根线段。如图 8.2 所示。

触觉器（线段）的数目以及它们的长度都是可以调整的，默认时，触觉器个数是 5，向外辐射出去的长度是 25 个像素。在每一帧（frame，即一个变动的画面）中，都要调用一个函数来检测每个触觉器是否与其周围世界的障碍物的边界线相交。每一辆扫雷机都带有一个触觉器的缓冲区 m_vecdSensors，这是一个用来记录扫雷机和可能遇到的任何障碍物的距离的一个向量 std::vector。距离用 0 与 1 之间的一个小数来表示。障碍物和扫雷机的距离愈近，由触觉器返回的读数也愈接近 0。图 8.3 显示了一个扫雷机的 5 个触觉器可能返回的近似读数。

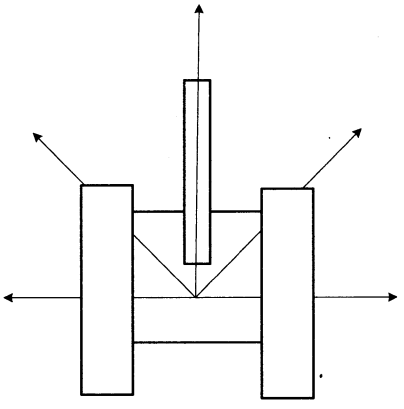


图 8.2　扫雷机的触觉器

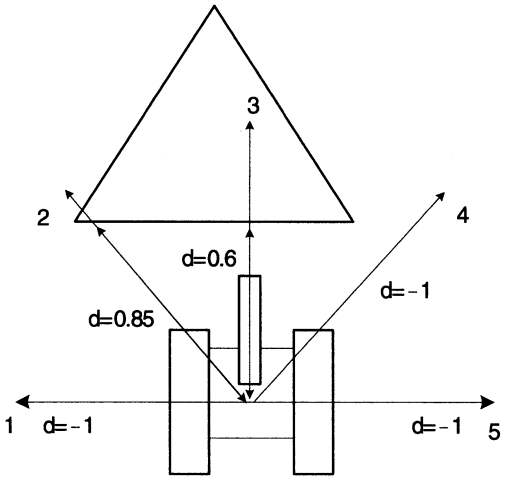


图 8.3　触觉器的典型读数

从图中可以看出，如果触觉器未碰到任何障碍物的边界线，则返回的读数被设置成了 -1。为了指示扫雷机是否实际已碰撞到了物体（仅仅是为了检测），需要做一次检测，检查每个触觉器返回的读数是否都在碰撞距离 dCollisionDist 以下，此值定义在头文件 Cparams.h 中，它是根据扫雷机本身的大小和触觉器长度的比例来估算的。这是相撞检测的一个比较粗糙的办法，但它的计算速度很快，并且，作为一个 demo 程序，其目的已能满足要求。

下面即为完成所有这些测试的代码：

```
void CMinesweeper::TestSensors (vector<SPoint> &objects)
{
```

从本函数传入的 SPoints 向量 objects 描述了需要由该扫雷机去探测的所有障碍物（对象）。这些对象定义在文件 CController.cpp 的开始部分。

```
m_bCollided = false;
```

这是一个标识，用以告诉扫雷机它是否已和任何对象碰撞。

```
//首先将触觉器的相对坐标变换为世界坐标
m_tranSensors = m_Sensors;
WorldTransform(m_tranSensors, 1);
```

代表触觉器的线段由创建触觉器的方法 CMinesweeper::CreateSensors 所创建并存放在顶点缓冲区 m_Sensors 中。因此，正如任何其他游戏对象一样，在每一帧中，这些触觉器需要转换成世界坐标，才能检测它们是否与组成障碍物边界的线段相碰撞。转换后的触觉器的端点坐标存放在 m_transSensors 中。

```
//清空触觉器
m_vecdSensors.clear();

//检测每个触觉器与其他对象的关系
for (int sr=0; sr<m_tranSensors.size(); ++sr)
{
  bool bHit = false;
  //这一标识的置位,表明触觉器与一个障碍相交

  double dist = 0;

 for (int seg=0; seg<objects.size(); seg+=2)
  {
   if(LineIntersection2D (SPoint(m_vPosition.x, m_vPosition.y),
                          m_tranSensors[sr],
                          objects[seg],
                          objects[seg+1],
                          dist))
     {
       bHit = true;

       break;
     }
}
```

本段代码利用循环为每一个触觉器线段调用函数 LineIntersection2D 进行相交性测试。可以在 collision.h 和 collision.cpp 文件中找到它的代码（有兴趣的话，可参看光盘上 FAQ

文件夹下的有关 comp.graphics.algorithms 的 FAQ)。[①] 如果检测到有一个相交，就应立刻退出循环，以避免进一步的不必要的计算。

```
    if(bHit)
  {
      m_vecdSensors.push_back(dist);

      //实现非常简单的碰撞检测
      if (dist < CParams::dCollisionDist)
      {
        m_bCollided = true;
      }
    }
```

如果已经检测到了触觉器与障碍物之间的一个相交，则碰撞已经被确定，因此应将碰撞标志 m_bCollided 置为 true。

```
    else
    {
      m_vecdSensors.push_back(-1);
    }
  }//下一个触觉器
}
```

这一函数是在 CMinesweeper::Update 开始处被调用的。所得的触觉器读数即作为输入值送进了扫雷机的神经网络。

8.1.2　适应性函数

这里的适应性函数必须反映扫雷机与障碍物的碰撞情况。适应性分数愈高，说明扫雷机闪避障碍物的本领愈好。为此，一种可用的方法就是每次检测到扫雷机与障碍物相撞，就处罚该扫雷机。用这样的记分方式是可以正常工作的，但要产生负的适应性分。当然，也可以利用负适应性分来工作，如同使用正适应性分一样。但从经验来看，当采用负的分数工作后，在编写程序时，代码容易引入难以捉摸的漏洞。因此，最好使用能永远产生正分的适应性函数。

第一个适应性函数就是简单地奖励那些在每一帧都能通过而不发生任何碰撞的扫雷机，其代码形式如下：

```
If (!Collided)
{
  Fitness += 1;
}
```

[①] 译者注：在原书光盘上没有 FAQ 文件夹，也找不到任何地方有 comp.graphics.algorithms 的 FAQ（计算机图形算法常见问题和解答集）。这是一个疏忽。译者已向作者提出了这个问题，根据作者的回复，已从网站 http://www.faqs.org/faqs/graphics/algorithms-faq/ 上下载了一个 FAQ 放在中文的光盘上。

思考一下这一类型的适应性函数会产生怎样的行为，可以参考光盘中的可执行程序 SmartSweepers v2.0。

注意：当运行本章执行程序时，扫雷机和它们的触觉器都用虚线画出。如果选择 elitism（精英统治）方式为 ON（默认方式即为 ON），扫雷机中的精英们将用黑实线显示出。如果任一触觉器和障碍物相交，则该触觉器显示为红色。如果检测到了碰撞，则整个扫雷机以红色显示。

F 键在这里同样是用来加速扫雷机的运行，R 键用来将程序复位。

必须找出一种方法来阻止扫雷机的疯狂打转，需要做的全部工作就是给在一帧中旋转次数不到某个数值的扫雷机一个奖励分。其代码如下：

```
If (fabs(Rotation) < RotationTolerance)
 {
  Fitness += 1;
 }
```

如果再运行一下改进后的 2.1 版的可执行程序 SmartSweepers v2.1，将会看到这一改进已经产生了合理许多的行为。

在屏幕上将显示出的适应性分数分为两部分，即有关旋转的奖励分 m_dSpinBonus 和有关碰撞奖励分 m_dCollisionBonus。但为了计算最终的适应性分数，它们在每一个代之后将结合在一起。运行程序时，在屏幕顶部看到的是分别计算的两种奖励分。图 8.4 显示了正在行动中的扫雷机的一幅快照。

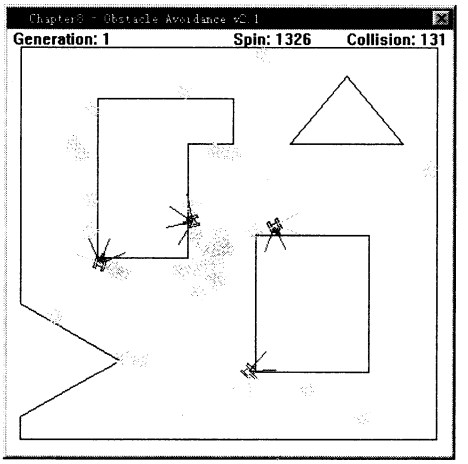


图 8.4　正在学习躲避障碍物的扫雷机

由于这种记分法所产生的适应性分与群体的分布紧密相关，所以在遗传算法中已使用锦标赛选择作为选择法。只要使用与适应性分成比例的选择技术，则演化算法几乎必定会用某种方式来产生好的结果。

读者将会发现，学习避开障碍物是神经网络相当容易学会的一种工作。在本例中，性能最好的扫雷机经过不多的几代后就会使它们的适应性分数达到最大。

表 8.1 为本程序的默认设置值。

表 8.1　Smart Sweepers v2.1 工程的默认设置

神 经 网 络	
参　　数	设　置　值
输入数目	5
输出数目	2
隐藏层数目	1
隐藏层神经元数目	6
激励响应	1

遗 传 算 法	
参　　数	设　置　值
群体大小	40
选择类型	锦标赛选拔
航行竞争对手数目	5
杂交类型	2 点
杂交率	0.7
突变率	0.1
精英设置（on/off）	On
精英数目（N/copies）	4/1

总 体 特 性	
参　　数	设　置　值
触觉器数目	5
触觉器长度（pixel 数目）	25
每一代（epoch）的滴答（ticks）数	2000

可以看到，当扫雷机学习避免障碍物时，实际并没有做很多事情。它们通常或者只是沿着障碍物的一个边走，或者是从一个障碍物跳到另一个障碍物。因为它们没有任何动机去做其他事情。扫雷机除了学习避开环境中的障碍物之外，另一种更为有用的行为就是去学习探索（explore）环境。为了做到这一点，必须使它们具有记忆功能。

8.2　为机器人提供记忆器

Bot 的记忆器（memory）用来记忆它所在的那个环境，可以用代表环境的一个简单的数据结构来表示。在本例中，将环境划分成许多同等大小的小方格或小房间，如图 8.5 所示。每个方格存放一个整数，而所有方格中的数据可以在一个二维向量 std::vector 中保存。

这个二维向量就可以用来作为保存相关信息的一张记忆地图（memory map）。在本例中被记录的是扫雷机在每一方格访问的滴答（tick）数。利用这个二维向量，扫雷机就能知道，在每一个小格，在此以前它是否曾经到过那里，或曾经呆过了多少时间。为了探索环

境，扫雷机必须演化神经网络，去优先访问那些没有访问过的小方格。

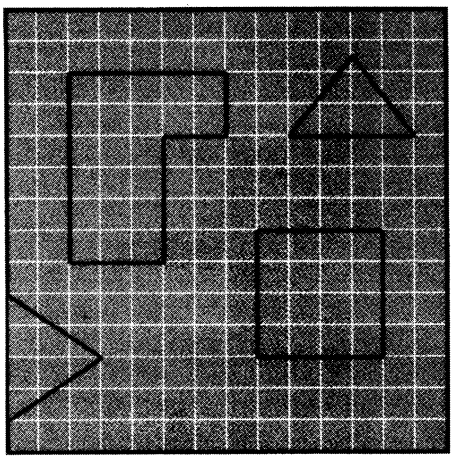


图 8.5　记忆方格

记忆地图由称为 CMapper 的类来实现。下面为它的定义：

```
class CMapper
 {
 private:

   //记忆小方格的二维向量
   vector< vector<SCell> > m_2DvecCells;
```

其中 SCell 是一个很简单的结构体，它保存一个记录着小方格坐标的矩形结构 RECT，一个记录了在那里花去多少时间的整数 iTicksSpentHere。SCell 结构体中也包含用来增加和清除 iTicksSpentHere 内容的方法。

```
    int m_NumCellsX;
    int m_NumCellsY;
    int m_iTotalCells;
    //每个方格的尺寸大小
    double m_dCellSize;

public:
    CMapper ():m_NumCellsX(0),
               m_NumCellsY(0),
               m_iTotalCells(0)
    {}

    //下面的方法在该类的一个实例被创建之后调用。它为所有方格设置了坐标
    void     Init(int MaxRangeX, int MaxRangeY);

    //下面的 Update 方法在每一帧中都要被调用,用来更新指定位置方格的访问时间
    void     Update(double xPos, double yPos);
```

```
    //返回花在此位置（方格）的时间长短（滴答数）
    int      TicksLingered(double xPos, double yPos) const;

    //返回被访问过的方格总数
    int      NumCellsVisited()const;

    //返回在给定位置的方格是否已经被访问或未被访问过
    bool     BeenVisited(double xPos, double yPos) const;

    //这种方法将访问的任何方格画为红色。被访问的次数愈多，红色愈深
    void     Render(HDC surface);
    void     Reset();
    int  NumCells(){return m_iTotalCells;}
};
```

扫雷机有了记忆器，就需要有一种利用记忆器来记住它们曾经在哪里的办法。由于在扫雷机的前一版本中已经对触觉器的端点位置作过计算，所以这一步骤的实现是很简单的。这些触觉器的末端可以在记忆地图上到处进行“探索”并从中取得信息。这和一个昆虫使用其“天线”进行探索的方法完全类似，如图 8.6 所示。

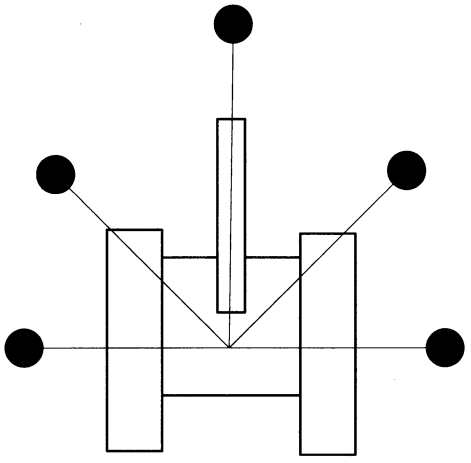


图 8.6　扫雷机的天线

这些触觉器得到的读数保存在一个称为 m_vecFeelers 的 std::vector 向量中，然后与触觉器原有的坐标参数一起被送入神经网络。

完成本工作的代码可以在 CMinesweeper::TestSensors 方法中找到。下面是新增加的几行代码：

```
//检查扫雷机在当前位置的小格中已经访问过多少时间
int HowOften = m_htemoryNap.TicksLingered(m_tranSensors[SP].X,
                                          m_tranSensors[SP].y);

if (HowOften == 0)
```

```
{
    m_vecFeelers.push_back(-1);

    continue;
}

if (HowOften < 10)
{
    m_vecFeelers.push_back(0);
    continue;
}

if (HowOften < 20)
{
    m_vecFeelers.push_back(0.2);

    continue;
}

if (HowOften < 30)
{
    m_vecFeelers.push_back(0.4);

    continue;
}
if (HoNOften < 50)
{
    m_vecFeelers.push_back(0.6);

    continue;
}
If (HowOften < 80)
(
    m_vecFeelers.push_back(0.8);

    continue;
}

m_vecFeelers.push_back(1);
```

由于把输入标准化后再送入网络是一种好的办法，所以把加入 m_vecFeelers 的任何值 v 都定义在-1 < v < 1 的范围之内。如果一个小方格以前从来没有被扫雷机访问过，则触觉器由此小方格读出的数值为-1。如果小方格已被访问过，则触觉器由此小方格读出的数值将在 0 和 1 之间。在小方格中呆过的时间愈长，从它得到的读数 v 也愈大。

这种具有滑动（slide）范围的记分法的意义是什么？观察下面的两个图，通过它们的对照就能得到最好的解释。先假设触觉器仅仅返回-1 或 1 两种读数，分析图 8.7 的情况。

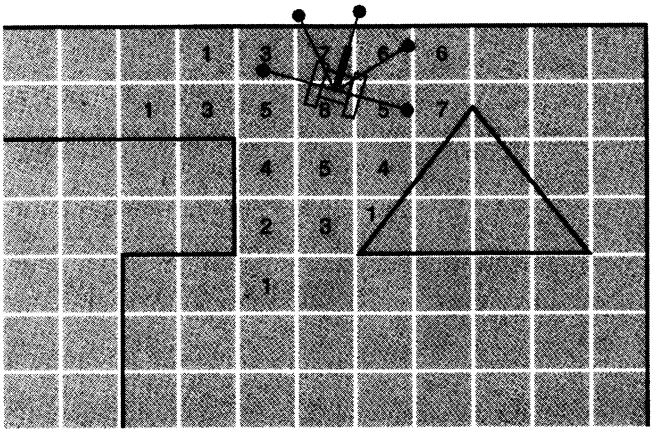


图 8.7　扫雷机被“粘”住了

图中的数字显示了扫雷机在每一方格所经历的时间多少。没有标明数字的方格代表从未访问的小方格。这一幅图对于一个扫雷机来说是一个非常普通的场景图（scenario）。由于触觉器只能为每一访问过的小方格读出 1，这样，图中的扫雷机就会由于无法获得寻找出路的线索而被“粘”在这个位置。它的触觉器无论触到哪里，所得到的读数都是相同的一个值：1。

但是当触觉器读数采用在一个范围内可以滑动变化的数据时，则神经网络通过学习就有可能指导扫雷机朝着数值较少的方格走并最终找到出路，如图 8.8 所示。

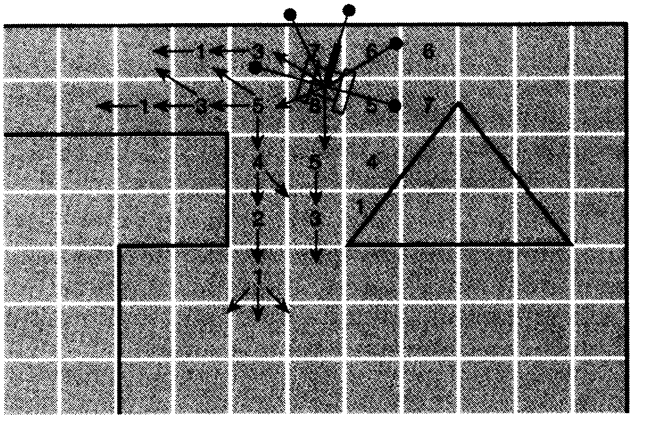


图 8.8　探索出路

这里已将其他的输入包含到进神经网络，即成员变量 m_bCollided。该变量能够明确告知扫雷机它目前是否已和一个障碍物相撞，并对其如何完成操作有帮助（当利用本程序来玩游戏时，可以尝试删除这一输入，考察网络的演化时间的变化）。

技巧：某些工作采用少量的短期记忆（short-term memory）是有益的。短期记忆可以通过把神经网络的输出直接连接到输入的办法来实现。例如，对于扫雷机，若要创建一个网络，使该网络带有两个附加的输入端，然后把前一个代的输出（m_ltrack 和 m_rtrack）

作为两个附加输入端的输入，如图 8.9 所示。这种类型的网络称为递归网络（recurrent network）。

这一想法还可以推广，把前一代的任意多个输出实行反馈。但是，这将大大减少网络的处理速度，因而应尽量避免这样做。在一个游戏中，总是希望网络尽可能快的。

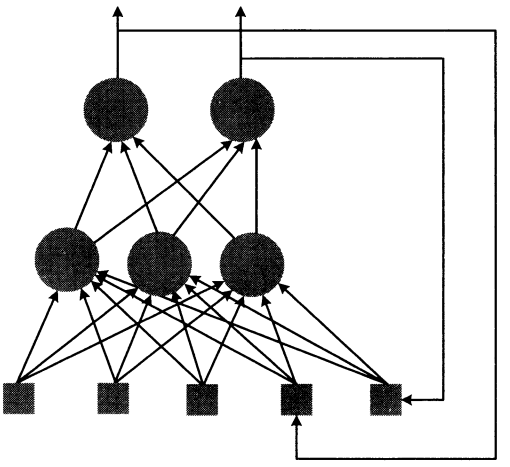


图 8.9　递归网络

我们可以把 2.1 版的适应性函数，与记忆在小方格中的分数结合在一起，来构造新的适应性函数。但完全可以仅考察小方格中的分数，这样还能更快地得到结果。为了尽可能多地访问小方格，扫雷机将会自动地去学习怎样避免与障碍物碰撞和避免绕圈子打转，因为要完成这两种动作的任何一个，都会减慢扫雷机的速度，从而产生较低的分数。

现在可以运行版本 2.2 可执行文件。可以看到，当遗传算法执行到 100～150 个代之后，扫雷机的性能就已经非常好了。

表 8.2 显示了本程序中所用的默认值设置。

表 8.2　Smart Sweepers v2.2 程序的默认设置

神 经 网 络	
参　　数	**设　置　值**
输入数目	11
输出数目	2
隐藏层数目	1
隐藏层神经元数目	10
激励响应	1

遗 传 算 法	
参　　数	**设　置　值**
群体大小	45
选择类型	锦标赛式

续表

遗 传 算 法	
参　　数	设　置　值
竞赛参与者数目	5
杂交类型	2 点
杂交率	0.7
突变率	0.1
精英设置（on/off）	On
精英数目（N/copies）	4/1

总 体 特 性	
参　　数	设　置　值
触觉器数目	5
触觉器长度（pixel 数）	25
每一代（epoch）的帧数目	2000

8.3 总　　结

需要说明的是，利用神经网络来解决的问题不一定都是大事情，如用来控制 FPS 游戏[①]中机器人（bot）的每一个动作那样。要构造一个网络来解决这样复杂的控制有点乐观主义了。只需用它们来处理游戏代理的人工智能的一部分的工作即可。

神经网络最好以模块的方式来加以应用。例如，可以为游戏中诸如追击（pursuit）、逃跑（flee）、侦察（explore）、聚会（gather）等特殊种类的行为分别进行训练，然后用另一个神经网络作为一个状态机，来选择在每一个给定的时间内与哪一种行为相关。甚至也可以设计一些网络，然后使用一台简单的有限状态机在每一个给定的时间选择一种合适的网络。

一个出色的应用就是利用神经网络来为 Quake 型[②]第一人称射击类游戏中的机器人（bot）计算射击瞄准。凡是玩过目前已有的 Bot 游戏的人几乎总能得出这样的一个结论：瞄准的人工智能需要大大加以改进。当 bot 们采用较高级水平游戏时，它们发出的 30% 的射击太精确了。简直就可以和电影 A Fistful of Dollars!中的 Client Eastwood[③]进行对赛了。但可以利用神经网络来对它进行改进，可以设计一个带有可见度等级（明亮/模糊/黑暗）、可见目标的数量、到达目标的距离以及当前选择的武器等参数作为输入，一个确定与目标中心的距离半径作为输出的网络。这样的神经网络经过训练后就能给出更为实际的瞄准行为。

同样可以在一个汽车赛游戏中用神经网络来控制汽车。在 Colin McRae Rally 的第 2 版[④]

① 译者注：FPS 为 First-Person Shooter 的速写，FPS 游戏即第一人称射击类游戏。

② 译者注：Quake 为游戏名，中译为“雷神之锤”。

③ 译者注：A Fistful of Dollars 原为一美国电影，中文译为“反恐精英”。Client Eastwood 在片中扮演了男主角。

④ 译者注：Colin McRae Rally 为一车赛游戏，中译为“科林麦克雷拉力赛车游戏”，目前已出 5 版。

中，赛车就是由一个神经网络来驾驶的，它已被训练到在赛道上奔跑。

有些思想利用遗传算法演化网络的权重是很难实现的。有时采用监督式的训练是解决问题的最好方法，这就是第 9 章中要谈论的一种方法。

8.4 练　　习

1. 将程序的一些参数设置加入到基因组中，使它们能被遗传算法所演化。例如，使能对触觉器的长度的个数等参数进行演化。

2. 在游戏世界中再加入几项东西让扫雷机去寻找。

3. 演化扫雷机，使它能避免与其他扫雷机相互碰撞。

4. 创建一个神经网络，为第 6 章中的登月飞船实现导航。

5. 演化神经网络，根据 Tron[①]，来玩光圈游戏（light circle game）。这一游戏如果你是第一次碰到，要实现决不容易。

6. 把可视化加入程序，使用户能实时看到神经网络和权重的变化。

[①] 译者注：Tron 是一款由电影“星际争霸”改变的空中飞车类游戏，由人—机进行比赛，现在的最高版为 6.2。而 light circle game 是仿照 Tron 构作的已经有的简单双人游戏。

第9章　有监督的训练方法

世界上有10种人。这就是……
知道二进制的人和不知道二进制的人。

本章介绍一种完全不同的训练方法来训练网络。在此之前一直使用的训练方法是无监督的(unsupervised)训练方法。现在要介绍的另一种训练方法则是有监督的(supervised)训练方法。有监督训练法适用于这样的情况：已经具备了可以用来训练网络的各种数据例子。第7章中已介绍网络如何训练来识别字符。它的工作过程如下：把一个输入模式送给网络，考察它这时的输出，并将它的输出与目标输出进行比较。如果实际的输出和目标输出不同，则网络所有的权重都要作稍微的改变，使得下一次再用同样模式输入时，网络的输出能和期望的结果接近一点。这一过程必须对要求网络学习的每一种输入模式进行重复，直到网络能正确识别每一种模式为止。

为了便于理解权重是怎样调节的，下面将用一个简单的数学函数：XOR函数作为例子来进行说明。

9.1　异 或 函 数

对于不熟悉布尔逻辑的读者，XOR 函数（异或函数）的功能最好利用表 9.1 来进行描述。

表9.1　XOR函数的功能

A	B	A XOR B
1	1	0
0	0	0
1	0	1
0	1	1

XOR 函数在神经网络的发展历史上曾起过重要作用。Marvin Minsky 在 1969 年证明了，一个神经网络如果只有一个输入层和一个输出层，就不能实现这个简单的函数。这是因为 XOR 函数是所谓线性不可分（linearly inseparable）这一类函数中的一个。一个逻辑函数（或其他二值函数）是线性不可分的，则它的两种不同函数值的集合在一张二维图上不能以一条直线来划分。图 9.1 显示了 XOR 函数和 AND 函数（“与”函数）的 0、1 两种函数值的分布图。AND 只有在两个输入都是 1 时才输出一个 1，它是线性可分函数的一个很好的例子。

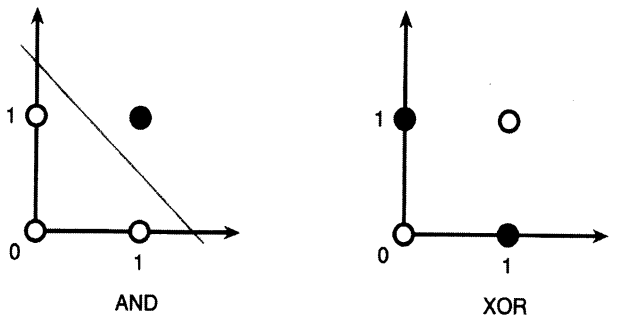


图 9.1　XOR 函数和 AND 函数

注：灰色直线说明 AND 函数是线性可分的

有趣的事情：布尔代数或布尔逻辑是乔治·布尔（George Boole）于 19 世纪中期发明的。他是在研究满足方程 $X^2=X$ 的数值解时，发现了这个以他的名字来命名的特殊逻辑分支的。满足这一方程的数只有 1 和 0，或 on（开）和 off（关），它们是组成现代数字计算机的各种基本元件所具有的两个状态。这也就是为什么谈到计算机时会经常谈到布尔操作的原因。

虽然人们从理论上已经知道，在输入与输出层的中间加入一些层，就可以解决这种类型的问题，但没有人知道如何来训练这样的多层的神经网络。在这种情况下，所谓连接主义（connectionism）的观点盛行起来，神经网络的研究一度走向了衰落。直到 20 世纪 70 年代中期，一个名叫 Werbos 的人指出，多层网络可以用反向传播（backpropagation）学习方法来训练。但直到 80 年代初该领域出现了大量业余爱好者时计算机科学家们才重新把神经网络当作一桩“事情”来加以研究。

反向传播的工作原理

反向传播 Backpropagation，简称为 backprop[①]的工作过程如下：首先创建一个含有一层或多层隐藏神经细胞的神经网络，并随机地为这些神经细胞的权重赋值，例如赋给-1 和 1 之间的任意一个实数值。然后把一个输入模式馈送到网络的输入端，并考察网络的输出值。这一输出值和要求达到的目标输出值之间的差称之为误差值（error value）。利用这一误差值就可用来调整来自输出层底下的那个层的输出的权重，使得当用同样的输入模式再次送入网络时，其输出能向正确答案接近一些。一旦当前层的权重调节完毕，就可以为它前面的那个层来做同样的事情等，这样由输出层开始，一层一层地向输入层方向推，直到到达第一个隐藏层为止，使所有层的权重都获得少量的调整。如果这一工作正确做完，则输入模式再一次被送入时，网络的实际输出将会向目标输出靠近一点。这样的全部过程要对所有不同的输入模式重复进行许多次，直到误差值降低到所处理问题可接受的一个极限值以内。这时就可以说网络已经被训练好（trained）了。

为了便于理解，将神经网络学习 XOR 函数所需要的训练数据用一系列的向量来表示，

① 译者注：通常进一步简称为 BP。

如表 9.2 所示。

表 9.2　XOR 函数训练集

输　入　数　据	输出数据（目标）
（1，1）	（0）
（1，0）	（1）
（0，1）	（1）
（0，0）	（0）

由这些输入 / 输出相匹配的模式组成的集合按下列的步骤去训练网络：

1．将权重初始化为随机的小的值。

2．为每一个模式，重复步骤（1）到（5）：

（1）将它送入到网络，并计算网络的输出 o。

（2）计算输出 o 与目标输出值（t）之间的误差。

（3）调整输出层权重。为每个隐藏层重复步骤（4）和（5）。

（4）计算隐藏层误差。

（5）调整隐藏层权重。

3．重复步骤 2，直到步骤（2）中的误差进入可接受范围内。

这就是为什么这种学习方法称为反向传播：计算的误差是从输出层到输入层反向传播的，如图 9.2 所示。

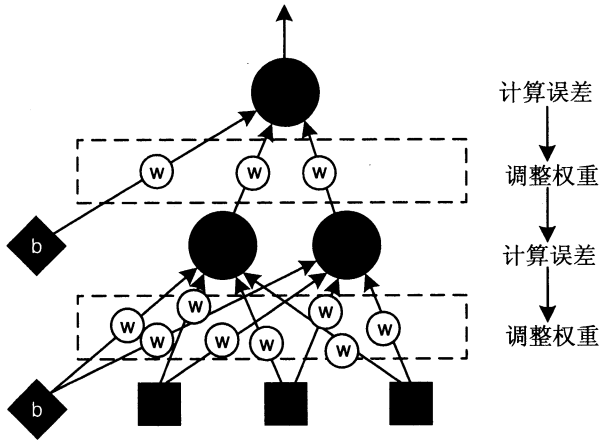


图 9.2　反向传播的工作过程

要了解反向传播或 backprop 学习算法所用到的计算公式的由来（推导过程）没有一点微积分知识是困难的，但这里不给出具体的微积分知识。这里只是说明应使用哪些公式以及如何使用这些公式。如果对这一算法的理论方面感兴趣，可以在本书后面所附的参考书目中找到很多好的阅读材料。

首先介绍所需的公式，然后利用实际数据针对 XOR 问题完整地执行一遍，就能了解 backprop 的工作过程。

需要的公式主要有两组：一组用作输出层误差的计算和权重调整的计算，另一组则用作隐藏层的误差计算和权重调整计算。为了避免复杂化，假设讨论的网络仅包含一个隐藏层。这对于了解过程并不失一般性，而且对于将来遇到的大多数问题而言，一个隐藏层是足够的。但若一旦需要两层或更多层时，去改变一下代码以适应新的需要，也不会特别困难。

1. 调整输出层的权重

首先讲述用来调整进入输出层的权重的计算公式。这里把第 k 个输出神经细胞的输出记作为 o_k，而第 k 个输出神经细胞的目标输出记作为 t_k，则第 k 个输出神经细胞开始时的误差值 E_k 为：

$$E_k = (t_k - o_k) \times o_k (1 - o_k)$$

为了改变在隐藏层第 j 个神经单元和输出层第 k 个神经单元之间的权重，使用下面的公式：

$$w_{jk} += L \times E_k \times o_j$$

其中 L 是学习率（learning rate），这是一个正的小数。学习率 L 的数值愈大，权重调整得也愈多。为了使网络获得最佳性能，这个数字必须人工调整。有关学习率的问题，下面会作出更多的说明。

2. 为隐藏层调整权重

计算隐藏层第 j 个神经细胞的权重调整的公式与此类似。首先应计算第 j 个神经细胞的误差：

$$E_j = o_k (1 - o_k) \times \sum_{k=1}^{k=n} E_k W_{jk}$$

其中 n 为输出层神经单元的个数。

确定了误差值后，从隐藏层 j 单元到输入层 i 单位的权重调整可用下式得到：

$$w_{ij} += L \times E_j \times o_j$$

这样的整个过程要对所有的训练模式进行重复，直到总的误差减少到一个可以接受的水平为止。

3. 实例

知道了计算公式是什么后，下面对 XOR 问题用实际的数据来完整地快速执行一遍，有助于了解 backprop 的工作过程。能够用来解决 XOR 问题的最小网络如图 9.3 所示。这一网络有些特别，它是一个全连接的网络：输入层不但连接到隐藏层的神经细胞，也直接连接到了输出层的神经细胞。这种网络虽然不是经常创建的一种网络类型，但因为它的尺寸小，能够把 backprop 所需的计算过程完整地列出，所以采用它作为例子。

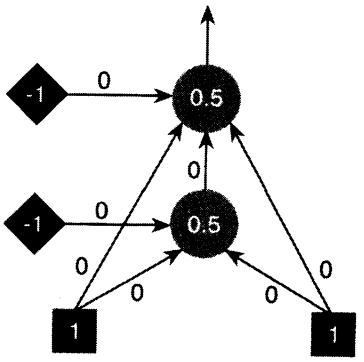


图 9.3　训练一个 XOR 网络

假设网络已被初始化，所有权重已设置为 0（通常，它们可以设置为随机的小的值）。为了演示的缘故，仅从输入模式中选择一个模式，即表 9.2 中的第一组模式（1,1），来执行一遍计算过程，因此，这时的目标输出是一个 0。另外，输出神经细胞中的数字 0.5 代表该神经细胞的激励值。不要忘记，S 形激励函数在所有输入为 0 时的输出为 0.5。

如前面所讨论过的那样，训练将按下列步骤进行：

1. 计算输出神经细胞的误差值。
2. 根据步骤 1 的结果与学习率共同调整权重。
3. 计算隐藏神经细胞的误差值。
4. 根据步骤 3 的结果与学习率共同调整权重。
5. 重复以上步骤，直到误差达到可接受的极限以内。

将数据代入，进行具体的计算。

1. 目标输出 t_k 为 0，而网络的实际输出 o_k 为 0.5，这样，利用方程：

$$E_k = (t_k - o_k) \times o_k (1 - o_k)$$

可得输出的误差：

$$error = (0 - 0.5) \times 0.5 \times (1 - 0.5) = -0.125$$

2. 调整进入输出层的权重应利用公式：

$$w_{jk} += L \times E_k \times o_j$$

从左到右计算进入输出层的 4 个权重的新值，并使用 0.1 的学习率。

新的偏移的权重$weight(bias) = 0 + 0.1 \times (-0.125) \times (-1) = 0.0125$

新的第一个权重$weight1 = 0 + 0.1 \times (-0.125) \times 1 = -0.0125$

新的第二个权重$newweight2 = 0 + 0.1 \times (-0.125) \times 0.5 = -0.00625$

新的第三个权重$newweight3 = 0 + 0.1 \times (-0.125) \times 1 = -0.0125$

3．利用公式：

$$E_j = o_k(1-o_k)\times\sum_{k=1}^{k=n} E_k W_{jk}$$

来计算隐藏层输出的误差，可得：

$$error = 0.5\times(1-0.5)\times(-0.125)\times(-0.00625) = -0.000195$$

在这里利用了第二步中更新了的权重来计算误差。如果我们首先计算好所有的误差然后再来调整各个权重也是合理的，两种方法都可以工作。之所以这样做，是因为这样能使代码更直观些。

4．调整进入隐藏层的权重。为此，再次从左到右利用公式：

$$w_{ij} += L\times E_j \times o_i$$

得：

新的偏移的权重$weight(bias) = 0 + 0.1\times 0.000195\times(-1) = -0.0000195$

新的第一个权重$newweight2 = 0 + 0.1\times 0.000195\times 1 = 0.0000195$

新的第二个权重$newweight2 = 0 + 0.1\times 0.000195\times 1 = 0.0000195$

在经历学习方法的这一轮循环以后，网络就变成图 9.4 所示的形式。

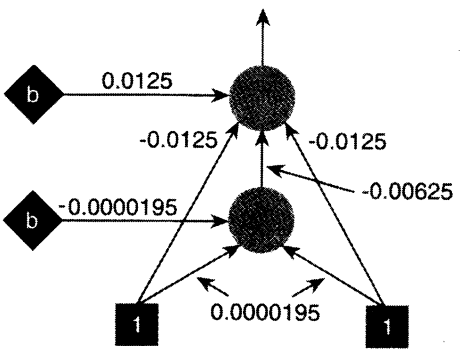


图 9.4　重复了一次 backprop 后的 XOR 网络

这一更新后的网络给出的输出值是 0.496094。它与原来的输出值 0.5 比较，和目标输出 0 接近了一点。

5．转到步骤一。

用反向传播解决 XOR 问题的思路就是不停地重复上述学习过程来逐步减少误差，直到误差小于一个可接受的值。重复的次数可以很大，这和学习率的大小有一种比例关系：学习率愈小，要求反向传播的次数愈多。可以利用加大学习率的方法来提高速度，但要冒着使算法掉入局部最小的风险。

例子仅仅显示了算法对输入模式（1,1）的执行过程。实际上，算法对每一种输入模式都要执行上述重复过程，算法的全部过程如下：

1．创建网络。
2．把权重初始化为平均为 0 的小的随机值。
3．为每一个模式，执行以下操作：
（1）计算输出层神经细胞的误差。
（2）调整输出层权重。
（3）计算隐藏层神经细胞的误差。
（4）调整隐藏层权重。
4．重复步骤 3，直到误差小于一个可接受的数值。

4. 转换为 CNeuralNet 代码

为了实现反向传播，神经网络类 CNeuralNet 以及有关的结构必须作少量修改，以适应新的训练方法。第一个要改变的是神经细胞的结构 SNeuron，使它能记录每个神经细胞的误差值和激励值。这些值是被算法频繁存取的。

```
struct SNeuron
{
    //进入神经细胞的输入个数
    int              m_iNumInputs;

    //每个输入的权重（矢量）
    vector<double>   m_vecWeight;

    //该神经细胞的激励值
    double                m_dActivation;

    //误差值
    double           m_dError;

    //构造函数
    SNeuron(int NumInputs);
};
```

神经网络类 CNeuralNet 也已改变到能适应新的学习算法。下面是新版本的 header（首部），对改变的部分已作了注释。为了清晰起见，已删去了与题无关的那些方法（它们供最后两章使用）。

```
//为输入输出向量定义了一个类型（用于训练方法）
typedef vector<double> iovector;
```

一个训练集由一系列的双精度标准向量 std::vector 组成。这样的定义能使代码更便于阅读。

```
class CNeuralNet
{
private:
```

```
    int          m_iNumInputs;

    int          m_iRumOutputs;

    int          m_iNumHiddenLayers;

    int          m_iNeuronsPerHiddenLyr;

    //为 backprop 规定一个学习率:
    double       m_dLearningRate;

    //网络的累计误差(所有(输出值-期望值)的总和)
    double       m_dErrorSum;

    //如果网络已被训练,为真
    bool         m_bTrained;

    //时代(epoch)计数器
    int          m_iNumEpochs;

    //为保存包括输出层在内的各层神经细胞的存储向量
    vector<SNeuronLayer> m_vecLayers;

    //给定一个训练集,下面的方法将对 backpropagation 算法重复执行一次
    //训练集由一系列的输入向量和一系列的输出预期值向量组成
    //如果方法执行中出现了一个问题,就返回 false
    bool      NetworkTrainingEpoch(vector<iovector> &SetIn,
                                   vector<iovector> &SetOut);
    void      CreateRet();

    //将所有权重设置为小的随机数值
    void      inltializeNetwork();

    //S 形响应曲线
    inline double  Sigmoid(double activation, double response);

public:

    CNeuraINet::CReuralNet(int          NumInputs,
                           int          NumOutputs,
                           int          HiddenNeurons,
                           double       LearningRate);

    //从一组输入值计算输出值
    vector<double> Update(vector<double> inputs);

    //对给出的训练集来训练网络。如果数据集有一个错误就返回 false
```

```
    bool            Train(CData* data, HWND hwnd);

    //访问用的方法
    bool            Trained()const{return m_bTrained;}

    double          Error()const {return m_dErrorSum;}
    int             Epoch()const {return m_iNumEpochs;}
};
```

在转入讨论第一个代码之前，先列出实现 backprop 算法的实际代码。backprop 算法需要对每一个训练集（它是一系列的 double std::vector 向量，表示出了每一个输入向量和它匹配的输出向量）重复执行一遍。对训练集的积累误差的记录保存在 m_dErrorSum 中。这个误差值是目标输出减去实际输出之差的平方的总和。在文献中，用这种误差计算方法算出的误差通常缩写为 SSE（Sum of the Squared Enors）。

CNeuralNet::Train 方法必须重复调用 NetworkTrainingEpoch 方法，直到 SSE 低于预定的某个极限值。此时，就可以说网络已经被训练好了（trained）。

```
bool CNeuralNet::NetworkTrainingEpoch(vector<iovector> &SetIn,
                                      vector<iovector> &SetOut)
{
  //建立几个迭代变量
  vector<double>::iterator curWeight;

  vector<SNeuron>::iterator  curNrnOut, curNrnHid;

  //用下面的变量保存训练集的累计误差
  m_dErrorSum = 0;

  //通过网络使每一个输入模式得到处理,计算网络输出误差,并更新对应权重
  for (int vec=0; vec<SetIn.size(); ++vec)
  {
    //首先通过网络对这一组输入向量进行计算,并获取其相应的输出
    vector<double> outputs = Update(SetIn[vec]);

    //如果遇到了错误,即返回
    If (outputs.size() == 0)
     {
      return false;
     }

    //为每个输出神经细胞计算误差并调整相应的权重
    for(int op=0; op<m_iNumOutputs; ++op)
    {
      //首先计算误差值
      double err = (SetOut[vec][op] - outputs[op]) * outputs[op]
                   *(1 - outputs[op]);
```

```
      //更新误差总值(当该总值低于一预设阈值时,即可知训练完成)
       m_dErrorSum += (SetOut[vec][op] - outputs[op]) *
                      (SetOut[vec][op] - outputs[op]);

     //用一个记录来保存误差
      m_vecLayers[1].m_vecNeurons[op].m_dError = err;
      curWeight = m_vecLayers[1].m_vecNeurons[op].
                                              m_vecWeight.begin();
      curNrnHid = m_vecLayers[0].m_vecNeurons.begin();

      //更新每一个权重,但不包括偏移
      while(curWeight !=
        m_vecLayers[1].m_vecNeurons[op].m_vecWeight.end()-1)
    {
    //基于 backprop 规则计算新权重
  *curNelght += err * m_dLearningRate * curNrnHid->m_dActivation;

      ++curWeight; ++curNrnHid;
      }

  //计算该神经细胞的偏移值
     *curWeight += err * m_dLearningRage * BIAS;
     }

     //**向后移动到隐藏层**
     curNrnHid = m_vecLayers[0].m_vecNeurons.begin();

     int  n = 0;

     //为隐藏层的每个神经细胞计算误差信号,并调整相应权重
     while(curRrnHid != m_vecLayers[0].m_vecNeurons.end())
    {
     double err = 0;

     curNrnOut = m_vecLayers[1].m_vecNeurons.begin();

     //为了计算此神经细胞的误差,需要对它所连结到的每个输出层细胞进行重复
     //并对它们的误差×权重求总和
     while(curNrnOut != m_vecLayers[1].m_vecNeurons.end())
       {
          err += curNrnOut->m_dError * curRrnOut->m_vecWeight[n];
          ++curNrnOut;
       }

     //计算错误误差
     err *= curNrnHid->m_dActivation * (1 - curNrnHid->m_dActivation);
```

```
      //对该神经细胞的每个权重,根据误差信号和学习率计算新权重
      for (int w=0; w<m_iNumInputs; ++w)
      {
        //根据 backprop 规则来计算新权重
      curNrnHid->m_vecWeight[w]+=err*m_dLearningRate*SetIn[vec][w];
    }

      //和偏移值
      curNrnHid->m_vecWeight[m_iNumInputs] +=
                                    err * m_dLearningRate * BIAS;

        ++curNrnHid;
      ++n;
     }

    }//下一个输入向量
    return true;
  }
```

下面用另一个有趣工程来说明如何实现它。

9.2　RecognizeIt——鼠标手势的识别

设想你正在玩一种实时战争游戏，必须把有关队伍攻击和防守的各种队列形式记忆下来。你不希望从键盘上敲击大量的代码来命令你的士兵排列成各种各样的队列形式，而是利用鼠标做的各种手势来进行指挥。你做一 V 形手势，你的士兵就立刻排成 V 形队伍；几分种后他们可能会受到威胁，这时如果你做一个方框形的手势，他们的剑与盾都向外，排成了一个方形队列。当再用鼠标连扫两次，士兵们马上拆开成两个队列……。

注意：如果想在进一步阅读更多内容之前尝试一下演示程序，可以在光盘的 Chapter9/Executable/RecognizeIt 1.0 文件夹找到可执行文件。

程序启动后就自动开始进入训练。必须等待到网络训练结束再开始输入手势。为了输入一个手势，必须按下鼠标右键，并在压下右键的状态下移动鼠标，然后释放鼠标。

图 9.7 是所有预定义的手势。如果网络能够识别手势，则手势的名称将在窗口左上角的蓝色横条中显示出来。如果网络不能确定，则它将会做出一个猜测。

在本书所附光盘中还可以找到 RecognizeIt 的其他的改进版本，在本章的后面也会介绍一些其他的方法。

这样的工作可以通过神经网络识别用户利用鼠标所作的各种手势的训练来达到，从而不必像普通游戏那样，需要有一个记录一切操作的“单击集合”。另外，也很容易让用户自己来定义自己的手势，而不需要被内建的手势束缚。

为了解决这个问题，必须做下面的工作：

1．寻找一种手势的表示方法，使它们能成为神经网络的一种输入数据。

2．利用上一步表示的方法，对某些预定义的手势来训练神经的网络。
3．指出一种了解用户何时在做手势以及怎样记录手势的方法。
4．指出一种将原始记录的鼠标数据转换成神经网络能识别的数据的方法。
5．使用户能够加入自己的手势。

9.2.1 用向量表示一个手势

第一个工作是寻找一种能把鼠标手势数据输入神经网络的方法。可以有几种不同的方法来实现。但这里选择的方法是把鼠标路径表示成一系列（共 12 个）的向量。图 9.5 说明了如何将一个“向右”的鼠标手势划分为一系列的向量。

为了帮助训练，这些向量在转换为训练集的一部分之前，需要进行规格化。所有进入网络的数据要和上例一样预先被标准化。这也给出了附加的优点，当处理用户所做的手势时，利用均匀散布的向量表示手势模式，有助于人工神经网络的识别过程。

神经网络的输出个数是应与要识别模式的数目相同的。例如，如果预先定义的手势只有向左、向右、向上、向下 4 种（如图 9.6 所示）那么神经网络将有 24 个输入（代表 12 向量）和 4 个输出。

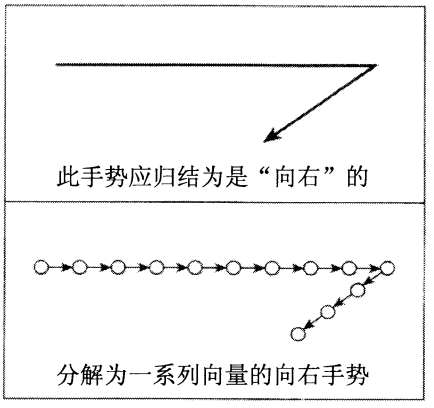


图 9.5　把手势看作一系列的向量

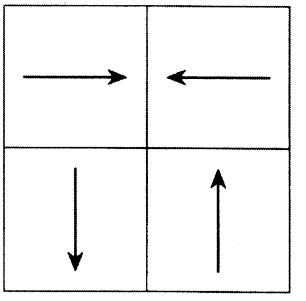


图 9.6　向右、向左、向下、向上 4 个手势

为这些模式设置的训练集如表 9.3 所示。

表 9.3　学习向右、向左、向下、向上手势的训练集

手　势	输 入 数 据	输 出 数 据
向右	(1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0)	(1,0,0,0)
向左	(-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0)	(0,1,0,0)
向下	(0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1)	(0,0,1,0)
向上	(0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1,0,-1)	(0,0,0,1)

由表可以看出，如果用户做一个向右的手势，神经网络应该从其第一个输出神经细胞输出一个 1，其余神经细胞均输出 0。如果用户做一个向下的手势，则神经网络应从其第 3

个输出神经细胞输出一个 1，其余神经细胞均输出 0。但实际上，这样的输出类型是很罕见的，因为由用户得到的数据总是每次都有些不同的。即使是重复做一个简单的向右手势，用手工来画也几乎每次都不可能拉成一根完美的直线！因此，网络为了思索送给它的是什么模式，它将扫瞄所有的输出，将把具有最高分数的输出当作最有可能的候选者。如果某个神经细胞分数最高，但只有 0.8 分，则很可能实际的手势并不是网络认为的那个手势。但如果输出分数在 0.96 以上（即在代码中作为默认值 MATCH_TOLERANCE 的定义值），则很有可能就是网络认为的那个手势了。

程序所有的训练数据封装在一个被称为 CData 的类中。这个类根据预定义模式（作为常数，定义在 CData.cpp 的开始处）创建一个训练集，如果用户要添加自定义手势，则也由它来处理训练集的改变。这里不再列出 Gdata 的源码，如果读者希望进一步了解该类是怎样来创建训练集的，可以查看光盘上的源代码，可以在光盘上的 Chapter9/RecognizeIt 1.0 目录下找到为手势识别的第一个尝试所提供的全部源码。

9.2.2　训练网络

现在已经用一系列向量来代表一个手势，并创建了一个训练集，但这仅仅是训练网络的一小部分工作。训练集将要传递给 CNeuralNet::train 方法，而后者则要反复调用 backprop 算法训练数据，直到误差平方的总和（SSE）小于用#define 语句定义的误差阈值 ERROR_THRESHOLD（默认值是 0.003）时为止。其代码形式为：

```
bool CNeuralNet::Train(CData* data, HWND hwnd)
{
    vector<vector<double> > SetIn  = data->GetInputSet();
    vector<vector<double> > SetOut = data->GetOutputSet();

    //确保训练集为有效的
    If ((SetIn.size()     != SetOut.size())||
       (SetIn[0].size() != m_iNuminputs) ||
       (SetOut[0].size() != m_iNumOutputs))
    {
      MessageBox(NULL, "Inputs != Outputs", "Error", NULL);

      return false;
    }

    //将所有权重初始化为小的随机数
    InitializeNetwork();

    //利用 backprop 进行训练,直到 SSE 小于用户定义的阈值
    while( m_dErrorSum > ERROR_THRESHOLD )
    {

      //如果出现任何问题,则返回 false
```

```
            If ( !NetworkTrainingEpoch(SetIn, SetOut))
            {
              return false;
            }

            //调用显示程序来显示误差总和
            InvalidateRect(hwnd, NULL, TRUE);
            UpdateWindow(hwnd);
          }
          m_bTrained = true;
          return true;
      }
```

当把源程序装进自己的编辑器后，必须设置好学习率才能开始运行。学习率的默认值是0.5。其中较低的学习率会使学习过程变慢，但几乎总能保证收敛。较大数值的学习率虽能加快学习进程，但有可能使网络（的输出值）跌入一个局部最小，甚至更坏，使网络根本不能收敛。因此，如同其他许多参数一样，宁可花时间以求获得正确的平衡是值得的。

图9.7显示了运行本程序时网络所学习的各种预定义手势。

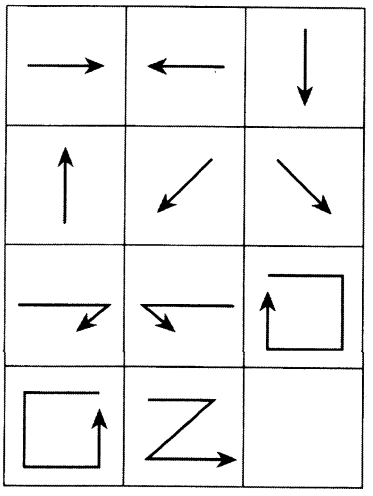


图9.7　预定义的各种手势

9.2.3　记录并变换鼠标数据

为了做一个手势，用户按住鼠标右键，再移动鼠标来画出一个模式。当用户放开鼠标右键时，手势就完成了。手势是用一系列点的方式直接记录在 std::vector 中。点的结构 POINTS 是在 windef.h 中定义的，其形式为：

```
typedef struct tagPOINTS {
    SHORT x;
    SHORT y;
} POINTS;
```

这一个向量可以有大大小小各种不同的尺寸，它完全取决于用户按住鼠标按键的时间长短。送入神经网络的输入个数必须是固定的。所以，需要寻找一种方法来减少原始输入的点数，使它们达到预先规定的数目。当完成这一工作时，还要对鼠标路径数据进行某种意义的“光滑”，以消除用户在做手势时由于各种不规则动作而引起的数据问题。这样做能帮助用户制作出更为一致的手势。

在举出的例子里，程序使用的网络共有代表 12 个向量的 24 个输入。而为了制作 12 个向量，你必须要有 13 个点（如图 9.5 所示）。因此，原始的鼠标数据必须用某种方法进行压缩，使它减少成为 13 个点。程序中采用的方法是通过对所有点的循环，找出具有最小跨度[①]的那一对邻点，然后在这最短跨度中间插入一个新点，再删去跨度的两个端点。这一过程一次只能减少 1 个点。必须重复进行，直至剩下的点数达到需要的个数时为止。

能完成这一工作的代码可以在 CController 类中找到，其形式为：

```
bool CController::Smooth()
{
    //确保工作的点达到足够的数目
    If (m_vecPath.size() < m_iNumSmoothPoints)
     {
     //返回
     return false;
     }

    //复制原始未加工的鼠标数据
    m_vecSmoothPath = m_vecPath;

    //当点数过多时,通过对所有点的循环,找出最小跨度,在它原有位置中间
    //创建一个新点,并删除原有的两个点
    while (m_vecSmoothPath.size() > m_iNumSmoothPoints)
    {
      double ShortestSofar = 99999999;
      int PointMarker = 0;

      //计算最短跨度（即相邻两点间的距离）
      for(intSpanFront = 2; SpanFront < m_vecSmoothPath.size()-1;
          ++SpanFront)
      {
       //在这些点之间计算距离
       doublelength = sqrt(( m_vecSmoothPath[SpanFront-1].x -
                             m_vecSmoothPath[SpanFront].x )*
                           ( m_vecSmoothPath[SpanFront-1].x -
                            m_vecSmoothPath[SpanFront].x )+
                           ( m_vecSmoothPath[SpanFront-1].y -
                            m_vecSmoothPath[SpanFront].y)*
                           ( m_vecSmoothPath[SpanFront-1].y -
```

[①] 译者注：span，也就是两点间的距离。

```
                                    m_vecSmoothPath[SpanFront].y) );

        If (length < ShortestSoFar)
        {
        ShortestSoFar = length;
        PointMarker = SpanFront;
        }
      }
//找出最短跨度,然后计算跨度的中点,作为新点的插入位置,并删除跨度
//原来的两个端点
      POINTS newPoint;
      newPoint.x = (m_vecSmoothPath[PointMarker-1].x +
                    m_vecSmoothPath[PointMarker].x)/2;
      newPolnt.y = (m_vecSmoothPath[PolntMarker-1].y +
                    m_vecSmoothPath[PointMarker].y)/2;
      m_vecSmoothPath[PointNarker-1] = newPoint;
      m_vecSmoothPath.erase(m_vecSmoothPath.begin() + PointMarker);
   }
   return true;
}
```

这种点数削减方法并不完美，因为它没有把手势的形状特征考虑进去，例如，没有考虑路径是直的还是带有转弯角度的。因此，当用户画一个带有角度的手势时，在经过光滑处理后的鼠标路径中，角度就变成圆滑的。但这种算法的优点是速度快，而为了成功地识别模式，利用光滑处理后所保留的信息作为输入数据来训练神经网络是足够的。

9.2.4 增加新手势

本程序也允许用户定义自己的手势。只要定义的手势惟一，这一步并不难，但这里涉及一个如何在训练集中加入数据的重要问题。如果已经训练了一个神经网络，而要求这个网络去学习一个其他的模式，如果仅为新加的一个模式再次试用 backprop 算法，通常并不理想。当需要增加新模式时，应首先把数据加入到原先已经存在的训练集中，然后删去已训练的网络权重数据，再重新开始对整个训练集进行训练就行了。

当用户需要增加一个新的手势时，首先需要在键盘上敲击一下 L 键，然后用鼠标去做一个手势。这时程序就会向用户提问他（或她）对输入的手势是否满意。如果用户满意，程序就开始对手势数据进行光滑处理，再将结果加到当前的训练集，并重新开始对网络进行训练。

9.2.5 控制器类

还可以对程序进行一些改进，这里先列出 CController 类的头文件。CController 类是一个与其他所有类有联系的类。所有的处理方法、变换方法、测试鼠标数据的方法等，都能在这里找到。

```
class CController
{

private:

    //该神经网络
    CNeuralNet*   m_pNet;

    //该类保持所有训练的数据
    CData*            m_pData;

    //用户原始的和光滑处理后的鼠标手势路径
    vector<POINTS>    m_vecPath;

    vector<POINTS>    m_vecSmoothPath;

    //将光滑处理后的路径转换为向量
    vector<double>    m_vecVectors;

    //如果用户正在作手势时为 True
    bool              m_bDrawing;

    //网络产生的最大输出,是一个最有可能的手势匹配候选者
    double            m_dHighestOutput;

    //根据最大输出 m_dHighestOutput 确定手势的最好匹配
    int               m_iBestMatch;

    //如果网络找到了一个模式（pattern）,那么这即为匹配者
    int               m_iMatch;

    //原始鼠标数据光滑处理时需要达到的点数
    int               m_iNumSmoothPoints;

    //数据库中模式（pattern）的数目
    int               m_iNumValidPatterns;

    //程序的当前状态
    mode              m_Mode;
```

程序可以处在下面的 4 种状态之一：

- TRAINING 状态：网络正处在训练时代。
- ACTIVE 状态：网络已训练好，程序准备去识别手势。
- UNREADY 状态：网络未受训练的状态。
- LEARNING 状态：用户正在输入一个自定义手势时的状态。

```
    //应用程序的本地备份
    HNND              m_hwnd;

    //清除鼠标数据向量
    void              Clear();

    //给定一系列点之后,用此方法创建一条规范化的向量的路径
    void              CreateVectors();
    //把鼠标数据预处理为固定数目的点数
    bool              Smooth();

    //通过对神经网络的查询为一预学习手势检测一个匹配者
    bool              TestForMatch();

    //对话框过程。当用户输入一个新手势时弹出一个对话框
    static BOOL CALLBACK DialogProc(HUND   hwnd,
                                    UINT    msg,
                                    WPARAM wParam,
                                    LPARAM lParam );

    //这一临时变量用来保存任何新创建模式的名称
    static string m_sPatternName;

public:

    CController(HNND hwnd);
    ~CController();

    //调用此函数,用 backprop 算法为当前数据集训练网络
    bool TrainNetwork();

    //画出鼠标手势的图形并显示相关数据,包括训练达到的代数和现有的误差
    void Render(HDC &surface, int cxClient, int cyClient);

    //返回鼠标当前是否处于 drawing 状态
    bool Drawing()const{return m_bDrawing;}

    //下面这个 Drawing 函数是当鼠标右键按下或释放时都要调用的一个函数
    //如果其中的第一个参数 val 为 true,则标明鼠标右键已按下。鼠标原有数据
    //均要被清除,为接受下一个手势做好了准备

    //如果 val 为 false, 则手势已完成。这时手势或者就被添加到当前数据集
    //或者测试它是否与已经存在的某个模式匹配

    //第 2 个参数 hInstance 用来创建一个对话框,它是主应用程序 main 的一个
```

```
        //子窗体。对话框用于获得用户打入的手势名称
        bool Drawing(bool val, HINSTANCE hInstance);
        //清除屏幕,把应用程序设置为 learning 模式,准备接受用户定义的手势
        void LearningMode();

        //调用本函数把一个点加入鼠标路径
        void AddPoint(POINTS p)
        {
            m_vecPath.push_back(p);
        }
};
```

9.3　一些有用的技术和技巧

有许多技巧能使网络学习得更快，或帮助它使其具有更好的归纳、推广能力，下面讲解其中比较流行的几种。

9.3.1　增加动量

反向传播算法应在神经网络在每一代都减少一点误差。可以想象网络有一幅反映其误差分布的场景图（landscapes），类似遗传算法相应的场景图。每迭代一次，反向传播算法根据误差分布图来确定当前点的误差梯度，并朝着全局最小的方向移动，如图 9.8 所示。

但如大多数的误差分布情况并没有图 9.8 那样光滑，而更可能像图 9.9 所示的曲线那样，有许多局部的极小点。因此，算法很容易收敛到一个局部极小而不能“自拔”。

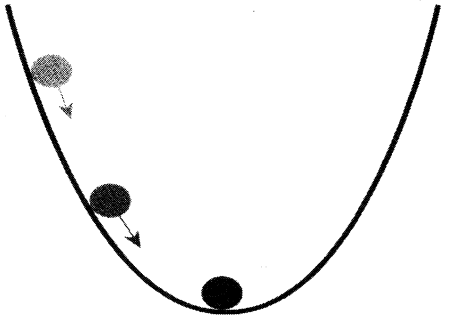


图 9.8　在误差 landscape 图中寻找全局最小值

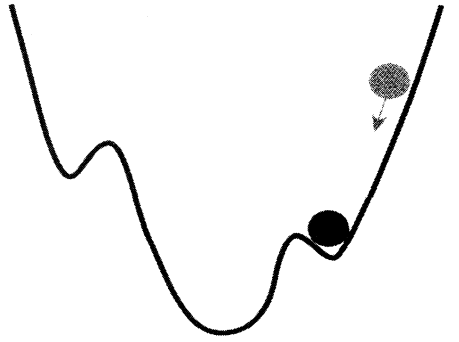


图 9.9　在局部最小点被卡住

防止发生这一情况的一种办法就是把动量（momentum）加到权重更新中。为了实现这一点，只要简单地把前一步的权重更新量的一部分加入当前的权重更新中即可。这将能帮助算法在误差分布图中压缩掉任何小的波动，从而为寻找全局性最小值提供了一个较有利的机会。使用动量还可以减少网络训练的代数。对于本例，加入动量后，可以把代的数目由 24000 个左右减少到 15000 左右。

计算权重原来使用的公式是：

$$W_{ij} += L \times E_j \times o_i$$

加入动量后的计算公式为：

$$W_{ij} += L \times E_j \times o_i + m \times \Delta W_{ij}$$

其中 ΔW_{ij} 是前一步的权重更新量，m 代表被加入的分数。m 通常设定为 0.9。

动量是相当容易实现的。首先改变 Sneuron 的结构，在其中增加一个 std::vector 双精度向量，用来存放前一步的权重更新量。然后，为了使 backprop 自我训练，需要另外增加一些必需的代码。可以在光盘 Chapter9/RecognizeIt v2.0(with momentum)的文件夹中找到这些源码。

9.3.2 过拟合

神经网络本质上就是一个函数的近似发生器。给定了一个训练集后，神经网络就要去寻找一个函数，使它拟合由训练集给定的输入数据到输出数据的函数关系。神经网络的问题之一，就是它们有时会学习得“太好”，从而失去归纳推广的能力。为了便于理解，设想有一个网络正在学习如何逼近图 9.10（a）所示数据所代表的函数关系。

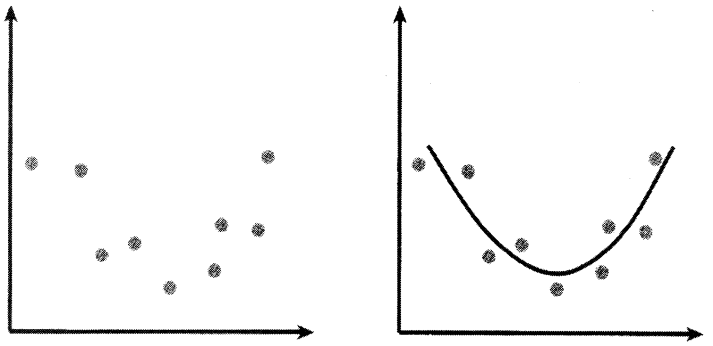


图 9.10　发现一个数据设置的最好匹配

显然，图 9.10（b）所示的曲线就是拟合了这些数据的最简单的函数曲线，而这也应该是理想中希望网络学习的曲线。但是，如果网络设计不合理，就很可能使曲线与数据产生过度拟合（overfitting）。如果发生这种情况，网络最终会学习成如图 9.11 中曲线所示的一个函数。

这一种网络尽管在拟合用于训练的数据集上做了大量工作，但对任何新加入的数据却很难实行拟合。如何防止过拟合？下面就是一些可以尝试的技术。

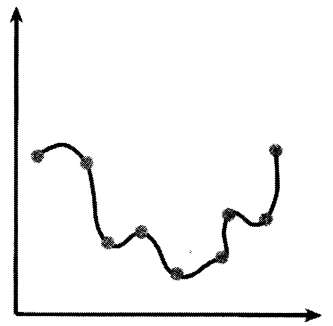


图 9.11　数据集的过拟合

1．把神经细胞的数目减到最小

应该做的第一件事情就是把网络中隐藏神经细胞的数字减少到最小。以前已经提到，还没有一种公认的可靠公式可以用来计算隐藏神经细胞的数目需要多少。因此，只能凭借过去已有的成功经验去试，如果失败再改。在 RecognizeIt 应用程序中使用的隐藏单元是 6 个，这是从 12 个开始不断减少下来得到的数字，再下去性能就开始退化了。

2．加入摇摆（jitter）

这里的摇摆是在训练集中加入的噪声。它能帮助网络提高推广或归纳能力。噪声是平均值为 0 的小的随机数据。这种方法可以阻止网络在拟合时过分靠近指定的数据点，从而能帮助防止过度拟合。从光盘 Chapter9/RecognizeIt v3.0(with jitter)文件夹所给的例子可以发现，已经在 CneuralNet::Update 方法中增加几行代码，用来把噪声加入到输入数据中。允许加入的噪声数据的最大数目量定义为 MAX_NOISE_TO_ADD。但是，在鼠标手势应用程序中仅依靠噪声加入只能使结果产生很小的变化。可以看到，噪声的加入对于使用大训练集的例子能得到更好的效果。

3．提前终止

另外一种简单技术就是提前终止程序的运行，这是一种应用大训练数据集时很有用的一种方法。将训练数据分成为两个集合：一个为训练集，另一个为验证集（validation set）。然后，使用小比例的学习率，并且使用有很多隐藏神经细胞的网络对训练集来训练网络。不必再担心隐藏神经细胞太多，但必须周期性地利用验证集来进行对照测试。思路是：当 SSE 误差已减少到一个预定值、而从验证集测试到的误差却相反开始增加时，就停止训练。当有一足够大的数据集，可以将它划分成两个集时，这一方法可以工作得很好并且速度很快。

9.3.3　柔性最大激励函数

类似于鼠标手势应用程序一类的问题属于分类问题（classification problem）。这就是说，给定一些数据后，网络的工作就是要把它放置到若干个类或范畴（categories）中某一个类中去。在 RecognizeIt 程序的例子中，神经网络必须确定用户输入的鼠标手势应归到哪一个预定义的模式类中。以前总是选择具有最高分数值的输出来代表最优匹配，这是可以的，但有时让输出代表“数据落到对应范畴的概率”就会更方便些。为了表示一个概率，首先要把所有可能的输出值加在一起。另外，输出神经细胞还必须使用一个完全不同的激励函数：softmax 激励函数。它的工作如下：

对于一个有 n 个输出单元的网络，输出神经细胞 o_i 的激励值由公式

$$output = \frac{\exp(w_i x_j)}{\sum_{0}^{n} \exp(w_n x_n)}$$

给出，其中 $w_i x_i$ 是进入该神经细胞中的所有输入×权重的乘积的总和。

这是一个相当容易混淆的公式[①]，为了得到从任何一个特定输出神经细胞的输出，首先必须把神经网络每个输出神经细胞的权重x输入的乘积累加起来。这里把总和记为 total A。一旦已做完了这一求和之后，为了计算输出，还要对每个输出神经细胞进行重复，让这一神经细胞的 A 的指数值（expA）去除以所有输出神经细胞的 A 的指数值的总和。下面是这一计算的代码：

```
double expTot = 0;

//首先计算输出的指数值的总和
for (int o=0; o<outputs.size(); ++o)
{
expTot += exp(outputs[o]);
}

//依次调整每个输出
for (o=0; o<outputs.size(); ++o)
{
outputs[o] = exp(outputs[o])/expTot;
}
```

如果在光盘的 Chapter9/RecognizeIt v4.0（softmax）文件夹下查看 RecognizeIt 第 4 版源程序，就能发现在 CNeuralNet::Update 方法中作了一些修改，使它能适应 softmax 激励函数。

尽管可以和往常一样利用误差平方总和函数（SSE）作为误差计算函数，但在使用 softmax 时可以使用一个更好的误差函数，交叉熵（cross-entropy）误差函数：

$$E = \sum_{j=0}^{n} t_j \log y_j$$

其中 n 为输出神经细胞的个数，t 为目标值，而 y 为实际的输出值。

有关这个计算公式的来源不再介绍，当网络输出为概率时，这是一个更好的误差函数。

9.4 监督学习的应用

当已有一系列输入模式需要映射到匹配的输出模式时，有监督的训练方法就能起作用。因此可以将这种技术用到任何地方去，包括弹跳类游戏、打杀类游戏，直到赛车类游戏等。

作为一个例子，假设需要制作一款赛车类游戏，希望用一个神经网络来驾驶一辆赛车，可以在赛车奔驰的过程中，将相关数据记录下来，创建一个训练集。对于每一帧结构（或每 N 帧结构），为训练集输入记录的信息可能为：

- 到左边界线（left curb）的距离。
- 到右边界线（right curb）的距离。

[①] 译者注：确实如此，而且还有错误。但因作者已指出了，所以从公式开始的本段内容，译者将一字不改地照原文译出。幸好下面的解释能帮助你弄懂实际要做的计算。

- 车子当前的速度。
- 当前轨道段（track segment）的曲率（curvature）。
- 下一轨道段的曲率。
- 到最优驾驶线（best drive line）的向量。

而为输出训练集记录了司机的响应：

- 向左或向右驾驶的总量（amount of steering left or right）。
- 油门总量（amount of throttle）。
- 制动总量（amount of brake）。
- 齿轮换挡（gear change）情况。

经历了几圈或经历了几个不同路程段之后，就积累了足够多的数据可用来训练神经网络，使它也能出现相似的行为。当给定足够的数据和进行正确的训练，神经网络应该能够归纳推广它所学到的本领，从而能自动行驶在以前从未到过的跑道上。

9.5　一个现代寓言

在我结束本章以前，我想为你留下一个小故事。故事看来是真实的，但我没有能够证实它。但不管怎样，当你在训练自己的神经网络时，请你在头脑中记着这个故事，因为我相信你一定不想犯故事中军事家们所犯的同样错误。

从前有一天，几个聪明人想出了一个主意：在一辆坦克上绑了一架照相机，然后用它来扫查周围环境中潜在的威胁，例如什么什么……，然后在一棵树后面隐藏了另外一辆巨大的坦克。他们知道，计算机擅长做重复的工作，计算机从来不会感到疲劳和自满，从来不会感觉厌烦，从来不需要休息。所以他们自认为这是一个了不起的想法。聪明人当然也知道，在图像目标的识别方面计算机很不擅长，但他们已经知道有神经网络了。他们听到了有关这一新技术的各种优点，他们准备在它上面花费大笔大笔的钱，由此就干起来了。

第二天，聪明人就宣称两组图像已经制作好了。一组图像中树后有部分隐藏的坦克，另外一组图像中则没有坦克，只有高高的树。聪明人看了看图像，感觉质量很不错。于是从每一个图像集中抽出一半的图像妥善保管在一间门窗都紧闭的暗房中——聪明人是特别重视安全的。

第三天，一架艺术型的大型计算机买来了，它高大的机柜由起重机吊起来后放落到一间为此专门盖起来的机房中。聪明人的一个手下轻轻拨动了一个开关，这台巨型机器就嗡嗡地叫了起来，与此同时，还启动了一个有 5 吨重的空调设备和一款艺术型的金光闪亮的制咖啡机。一群利用高昂费用借用来的聪明极顶的程序员从世界各个角落飞落到这里。程序员们检查了机器，根据它的许多闪动的指示灯、呼呼作响的磁带和许多发着亮光的终端机，他们确信计算机是正常的。

第四天，程序员们根据要求的类型带来一个人工神经网络程序。经过许多小时调试、保证它工作正常没有 bug 后，他们开始向神经网络输入具有坦克和树的图像。每当向程序输入一幅图像时，它必须猜测树中是否有一辆坦克。起先，程序工作得很不好，但当聪明

的程序员根据出现的错误对它进行了惩罚之后，系统几乎不需要任何时间地飞速获得了改进。第四天结束时，程序对每一幅图片都能得到正确答案。啊，生活真美好！

这时的咖啡已经煮得变成又浓又苦了。

第五天早上，程序员们再次验证了结果，然后打电话通知聪明人。聪明人来了后，看着计算机精确地识别了坦克，感觉它工作得确实不错。然后就下命令将暗房的门打开，把保留的图像也拿来进行试验。程序员们虽然知道这样的时刻迟早要到来，但由于不知道要期待什么，所以心里总是有点担心。神经网络软件会工作得好还是不好？他们不可能清楚说出来，因为，尽管他们已经设计了网络，但确实不知道它内部发生的线索。这样，他们用颤抖的手，把新的图像一个一个地送进了机器。

确实没有错！程序员们全部解脱了，他们高兴地看到每个答案都正确，他们从心底里感觉到了许多快乐。

第六天的黎明，一些聪明人又关心起来了，因为他们知道在这个该死的世界里事情从来也没有如此顺利过。就决定去拍摄一组新的图片来进行重复试验，他们把图像全速送进了计算机。但令他们感到恐怖的是，答案竟是完全随机的！“哦，不会吧！”，聪明的程序员们都叫起来了：“我们确实已经把螺丝拧紧了！”

一个聪明人用他瘦骨嶙峋的食指指向突然一下子变得不那么聪明的所有程序员，他们像一阵风似地迅速解散了。

第七、第八天，以及在此后的更多更多天的时间里，聪明人与一组新雇佣来的聪明程序员都在考虑以前出现的错误究竟是怎样产生出来的。但谁也猜不透其中的原因。直到有一天，一位机警的程序员注意到了，在初次使用的那组图片中，所有包含坦克的图片都是在多云的天气下拍摄的，而没有坦克的图像都是在阳光灿烂的天气下拍摄的。原来，计算机仅仅学会了晴天和阴天之间差别的识别！

9.6 练　　习

1. 把学习率和其他参数作一些修改，看它们对训练网络有什么影响。完成这一工作时，尝试 S 形函数激励响应的改变，以了解响应曲线改变如何影响学习效率。

2. 训练一个神经网络来玩乒乓游戏。首先找出一种来训练它有监督的方法。然后仿照本章两个例子编写一些代码来演化网络玩此游戏。

3. 按照上面的方法训练网络玩井字游戏。

第10章 实时演化

一旦我们开始了编程，我们会吃惊地发现，要使程序正确并不像我们想象的那么容易。程序必须经过调试才能发现各种各样错误。我能确切记得，我当时意识到了，从此之后我将要为在自己的程序中查找错误而花费我生命中一大部分时间。

——Maurice Wilkes 发现 debugging（调试），1949

通过一代一代的改进过程去演化行为的这种方法，适合于允许用非在线的方式或者可以利用游戏的自然停止等期间（如在进行升级时），来改进游戏代理的行为。在这样的一些情况下，这种方法都是很适宜的。但下面要介绍的另外一种简单方法允许正在游戏时演化基因组。这种方法适合于有很多很多的游戏代理，它们经常是一些在毁灭，另一些在诞生出来。这种演化类型的优点是，它能适用于接纳各种动态的游戏对象，如不同人的游戏玩家，或者切换到了可能不会遇到离线的游戏环境。例如，在实时战斗游戏中，坦克就有可能要根据它的对手的游戏方式来立即改变自己的行为。

下面将要介绍的技术实际只是实现过程中的一个小技巧。简单地说，当需要进行在线演化群体时，所要做的全部工作，就是设置一个基因组池（pool），来保存群体中所有的基因组，并使它们始终保持有序。游戏中使用的个体就是从这一个池中孵化出来的。当游戏时某一个体被杀死之后，就立刻会有另一个替身从池中孵化出来。替身是由群体中性能较好的一些分子中（例如从分数最高的 20%中）选出经过变异得到。利用这样的演化方式，群体不会出现一代一代演化时所出现的波浪式变化，而是不断发生着经常性的个体生、死周期。由于这一原因，为了正常工作，这一技术需要游戏代理的快速更替。如果你的游戏代理按太低的比率死去，则演化不太可能获得令人满意的步伐。但如果你的游戏代理能像打杀类游戏（shoot-em-up）或某些实时军事类游戏那样实现快速更替，则利用本演化方法就可以获得很好的效果。

10.1 外星人游戏

为了讲述上面的原理，下面举例介绍，这样的技术如何能应用到类似于太空入侵者（Space Invader）中外星人动作的演化。

这里将例子进行了简化。屏幕中只有一帮外星人在飞行，另外就是他们的敌人——玩家。外星人必须学习怎样尽可能长地生存着而不被玩家杀死。如果他们被你击中或者飞离屏幕的顶部或底部那么都要死去。他们活得愈长，他们的适应性分数也愈高。

程序使用了两个外星人收容所。第一个是 std::multiset，这是一个保存外星人的有序池，第二个是 std::vector，它用来保存当前活跃于游戏中的入侵者，如图 10.2 所示。

图 10.1　外星人游戏界面

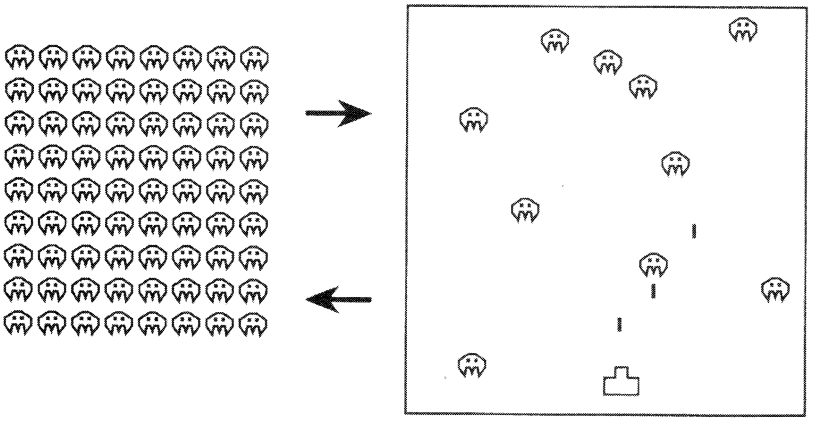


图 10.2　实时演化

当一外星人死去时，他就从游戏的画面中移走；如果他的适应性分数高于群体中表现最差者的分数，则他的基因组就被加入基因组池中，而他在游戏中的位置就由迄今为止表现较好的分子经过突变得到的一个外星人所代替。

10.1.1　程序实现

可以在光盘“Chapter10/Brainy Aliens”文件夹中找到本程序的源码。程序中有关游戏对象的类共有 3 种：枪类（CGun），子弹类（CBullet）和外星人类（CAlien）。此外同样还包含通常都会有的神经网络类 CNeuralNet 和控制器类 CController。CGun 和 CBullet 类的结构很简单，利用代码中所加的注释就能完全看懂。但为了能正确地理解每一桩事情是怎样

工作的，必须更详细地描述一下 CAlien 和 Ccontroller 两个类的结构。首先介绍怎样控制外星人。

10.1.1.1 外星人大脑的解剖

在描述外星人头脑的内部工作以前，可以快速浏览一下外星人类 CAlien 的定义，如下所示：

```
class CAlien
{

private:

    CNeuralNet        m_ItsBrain;

    //外星人在世界坐标系中的位置和速度
    SVector2D         m_vPos;

    SVector2D         m_vVelocity;

    //它的大小
    double            m_dScale;

    //它的质量
    double            m_dMass;

    //它的年龄（=它的适应性分）
    int               m_iAge;

    //它的最小包围矩形（bouding box,供碰撞检测用）
    RECT              m_AlienBBox;

    //外星人局部坐标的顶点缓冲区
    vector<SPoint>    m_vecAlienVB;

    //存放外星人变换后顶点坐标的顶点缓冲区
    vector<SPoint>    m_vecAlienVBTrans;

    //当设置为 true 时，一个警告（warning）会被显示出来，通知有一个
    //输入的大小与神经网络不匹配
    bool              m_bWarning;

    void              WorldTransform();

    //检查是否和任何活动着的子弹相撞，如果检测到相撞，返回 true 值
    bool      CheckForCollision(vector<CBullet> &bullets)const;

    //更新外星人的神经网络，并且返回它的下一个行动
```

```
    action-type  GetActionFromNetWork
      (const vector<CBullet> &bullets, const SVector2D  &GunPos);

    //重载'<',用于排序
    friend bool operator<(const CAlien& lhs, const CAlien& rhs)
     {
      return (lhs.m_iAge > rhs.m_iAge);
     }

public:

    CAlien ();
    void Render(HDC &surface, HPEN &GreenPen, HPEN &RedPen);

    //查询外星人大脑,并相应地更新它位置
    bool Update(vector<CBullet> &bullets, const SVector2D &GunPos);

    //复位任何有关的成员变量,准备进行一次新的运行
    void Reset();

    //将外星人神经网络中的连接权重实行突变
    void Mutate();

    //-----------------------------------访问方法
    SVector2D    Pos()const{return m_vPos;}
    double          Fitness()const{return m_iAge;}
};
```

外星人的大脑结构如图 10.3 所示。默认时，屏幕上任何时候只有 3 个子弹。为了检测子弹在什么地方，一个外星人的神经网络有 6 个（3 对）输入，每两个（一对）输入代表由它到子弹的向量。另外，每个神经网络还有两个输入，用来表示从外星人到发射子弹的枪（gun，或架设枪的战车 turret）间的向量。如果子弹不活动（没有进入到屏幕上），则神经网络的输入也接受从它引向战车（枪）的向量。

外星人具有质量，并受到重力的影响。为了移动，他们能引发出冲刺或推进动作，由此使他们突然向上、向左或向右移动。外星人在每一帧中都有 4 种可以选择的动作，这就是：

- 向上推进（thrust_up）
- 向左推进（thrust_left）
- 向右推进（thrust_right）
- 漂移（drift）

一个外星人的神经网络有 3 个输出，每一个输出都如上面列表中前 3 种动作之一那样进行切换。要切换到 ON，输出必须具有大于 0.9 的激励输入。如果存在多个的输出超过 0.9，则选择分数最高者。如果所有输出处于 Off 状态，则外星人就受重力作用而下落。

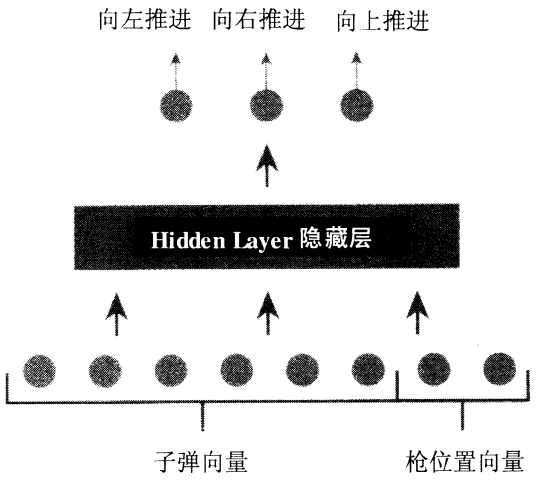


图 10.3　一个外星人大脑的内部

4 种动作组成了 action_type 枚举类型，定义如下：

```
enum action_type{thrust_left,
                 thrust_right,
                 thrust_up,
                 drift};
```

例如，图 10.4 的左图显示的输出将引发 thrust_right（向右推进）行动，而图 10.4 右边的图显示的输出将引发漂移（drift）动作。

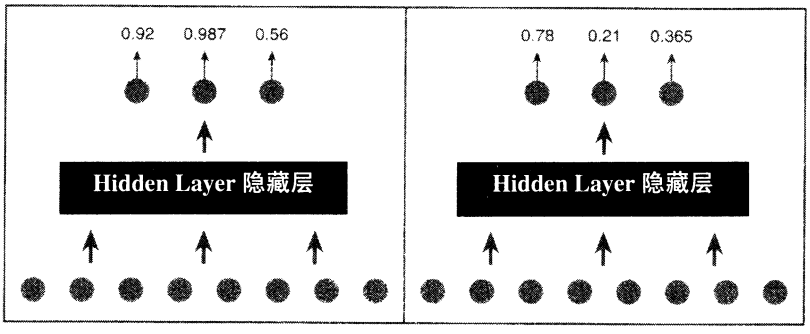


图 10.4　外星人的行动例子

下面为更新并从外星人大脑接收指令的方法的代码：

```
action_type CAlien::GetActionFromNetwork
      (const vector<CBullet> &bullets, const SVector2D &GunPos)
  {
   //送进到网络的输入
   vector<double> NetInputs;

   //用来保存从神经网络产生的输出
```

```
    static vector<double> outputs(0,3);

    //指向战车位置的向量
    int XComponentToTurret = GunPos.x - m_vPos.x;
    int YComponentToTurret = GunPos.y - m_vPos.y;

    NetInputs.push_back(XComponentToTurret);

    NetInputs.push_back(YComponentToTurret);

    //对任意一个子弹进行操作
    for (int blt=0; blt<bullets.size(); ++blt)
    {
      if (bullets[blt].Active())
        {
        double xComponent = bullets[blt].Pos().x - m_vPos.x;
        double yComponent = bullets[blt].Pos().y - m_vPos.y;

        NetInputs.push_back(xComponent);
        NetInputs.push_back(yComponent);
        }

    else
        {
        //如果子弹未动,则输入一个指向枪台(战车)的向量
        NetInputs.push_back(XComponentToTurret);
        RetInputs.push_back(YComponentToTurret);
        }
}

//将输入送进网络并获取输出
outputs = m_ItsBrain.Update(NetInputs);

//如果更新中出现了任何问题,则将其设置为 true
if (outputs.size() == 0)
    {
      m_bNWarning = true;
    }

//确定本帧中最高值输出超过 0.9 的那个动作有效,如果没有超过 0.9 的值输出
//则由重力确定其动作为漂移
  double BiggestSoFar = 0;
     action_type action = drift;

      for(int i=0; i<outputs.size(); ++i)
     {
          if((outputs[i] > BiggestSoFar) && (outputs[i] > 0.9))
```

```
            {
                action = (action_type)i;

                BiggestSoFar = outputs[i];
            }
        }

    return action;
}
```

由于只有当 S 形函数到达上升沿时，程序才能作出响应，故在 params.ini 中 S 形函数的激励响应被设置为 0.2，这比一般的设置要低些。这样可以使响应曲线变得更陡峭一点，从而使网络对连接权重的变化感觉更灵敏，有助于加速演化。

现在已经知道了神经网络是怎样构造出来的，下面开始解释演化机制是如何进行工作的。

10.1.1.2　外星人的演化

控制器类 CController 是和一切类都有联系的类，但这一次其中不再包含 epoch 函数。所有的孵化和突变操作这里均由 Update 方法进行处理。下面完整地给出 CController 类的定义。

```
class CController
{
private:
    //游戏者用的枪
    CGun*               m_pGunTurret;
```

游戏者可以利用键盘上的 left 键和 right 键向左或向右移动用来发射子弹的枪（台）或战车。发射子弹利用 Space bar 键。如果查看 CGun 类，能发现另外有一种方法，称为 AutoGun。它将使战车无规则地向左或向右移动，并随机地快速发射一系列的子弹。这是因为程序开始运行时外星人的行动显得相当迟钝，利用 AutoGun 的加速行动可以快速孵化外星人，直到外星人群体规模达到要求的数目（默认值 200 个）为止。

```
    //外星人孵化池
    multiset<CAlien> m_setAliens;
```

这是一个用来孵化所有外星人的基因组池。multiset 是一个用于存放并使所有存放成员保持有序的 STL 容器。下面的附加代码能够帮助进一步了解 multisets 是怎样使用的。

```
    //当前活动的外星人
     vector<CAlien>  m_vecActiveAliens;
```

这些是在游戏中活动着的外星人。当其中一个死去时，它就被由 m_setAliens 中适应性分数更高的成员中突变产生的一个外星人所代替。

```
    int                 m_iAliensCreatedSoFar;
```

这一变量在开始运行的一段时间内用来存放在此以前所有新创建的外星人数目。每一个新外星人首先需要在游戏环境中争取到适应性分数，然后才能被加入到 multiset 中。当这一变量数目达到要求的大小时，外星人就可以从 multiset 中进行孵化。

```
    int                 m_iNumSpawnedFromTheMultiset;

    //存放背景星空中各星体顶点的缓冲区
    vector<SPoint>      m_vecStarVB;

    //保存窗体的大小的数据
    int                 m_cxClient,
                        m_cyClient;

    //让程序运行尽可能地快的标志
    bool                m_bFastRender;

    //用来勾画游戏对象的用户画笔
    HPEN                m_GreenPen;
    HPEN                m_RedPen;
    HPEN                m_GunPen;
    HPEN                m_BarPen;
     void      WorldTransform(vector<SPoint> &pad);

    CAlien     TournamentSelection();

public:

     CController (int cxClient, int cyClient);

     ~CController ();

     //Update 是程序的“主要劳动力”。由它来更新所有的游戏对象，并孵化新的外星
     //人加入到外星人群中
     bool Update();

     void Render(HDC &surface);

     //复位所有的控制变量，创造一个新的外星人群体，准备一次新的运行
     void Reset();

    //------------------------访问函数
     bool FastRender( ){return m_bFastRender;}
};
```

STL 注释： 一个 multiset 是一个标准模板库容器类（STL container class），它能根据成员所要求的排序类型，使成员保持有序（默认的排序类型是 < 操作）。它和 std:: set 很像，

只是 multiset 允许包含复合数据类型，set 则不允许这样。但要使用一个 multiset，必须用 #include 语句包含 set 这个必要的头文件。

```
#include <set>
multiset<int> MySet
```

要把一成员加到一个 multiset（或 set）中，可利用插入（insert）操作：

```
MySet.insert(3);
```

由于 set 和 multisets 都是利用二叉树来实现的，无法把所有的成员按先后顺序列出，所以，他们的元素不允许直接访问。只能利用 iterator（迭代用指针变量）来代替它访问其元素。

下面是一个应用 STL 的例子，它把 10 个随机整数插入到一个 multiset 中，然后在屏幕上有序地显示这些元素。

```
#include <iostream>
#include <set>

using namespace std;
 int main()
  {
    const int SomeNumbers[10] = {12,3,56,10,3,34,8,234,1,17};
    multiset<int> MySet;
    //首先加入这几个数字
    for (int i=0; i<10; ++i)
     {
        MySet.insert(SomeNumbers[i]);
     }

    //创建一个迭代指示变量
    multiset<int>::iterator CurrentElement = MySet.begin();

    //用迭代指示变量来访问各成员
    while (CurrentElement != MySet.end())
     {
        cout << *CurrentElement << ", ";

        ++CurrentElement;
     }
     return 1;
  }
```

当运行这一程序时，其输出是：

```
1, 3, 3, 8, 10, 12, 17, 34, 56, 234
```

10.1.1.3　CController::Update 方法（控制器类的更新方法）

这是程序的“主要劳动力”。在更新了战车（向外发子弹的地方）和天空中的星星之后，

该方法对存放在 m_vecActiveAliens 中的所有外星人重复调用他们的更新函数 CAlien::Update。更新函数查询每一个外星人的大脑，看他们在这一时间步骤中采取什么行动，并相应地更新他们的位置。如果外星人已被子弹击中，或在行动时超出了窗体的边框之外，则它就应退出游戏，并加入到外星人的群体池内（它的适应性分数值就是它存活的滴答数）。由于池中是用一个 std::multiset 作为容器，故任何新加入的外星人都自动插入到已经（根据适应性）正确排序的队伍之中。如果所需要的外星人的人口数目已经达到要求，则程序就要删除 multiset 中最后的那个成员（最差的外星人）以保持池中总的外星人人数不变。

程序的下一步是用一个新的外星人来代替现已经死去（离开）的外星人。为了实现这一点，程序中使用了锦标赛选择法，从外星人池中分数最高的 20%（默认值）当中选出来。然后按突变率对这些个体的权重实行突变，再把结果加入到 m_vecActiveAliens 中。因此 m_vecActiveAliens 的大小永远保持常量。外星人个数的默认值显示在屏幕上，可以在任何时候用参数 CParams:iNumOnScreen 进行设置。

注意： 在本程序中已经省略了杂交操作，因为笔者想证明没有杂交操作同样能够成功演化，并且，使用这一技术可以命令孵化代码运行尽可能地快。

在第 11 章里，将会用另一个理由来解释，为什么在演化神经网络时删去杂交操作是一种好办法。

下面的代码将能帮助读者清楚地理解这个过程：

```
bool CController::Update()
{
    //如果孵化出来的子孙足够,就把 autogun 方式关掉
    if (m_iNumSpawnedFromTheMultiset > CParams::iPreSpawns)
      {
        m_pGunTurret->AutoGunOff();

        m_bFastRender = false;
    }

    //从游戏者那里获取枪炮台,战车（Gun turret）位置的移动数值并进行更新;
    //如果子弹已从 Gun turret 发出,则子弹位置也要进行更新
    m_pGunTurret->Update();

    //移动天空中的星星
    for (int Str=0; Str<m_vecStarVB.size(); ++Str)
    {
      m_vecStarVB[str].y -= 0.2;

      if (m_vecStarVB[Str].y < 0)
      {
          //创建一颗新的星
          m_vecStarVB[str].x = RandInt(0, CParams::WindowWidth) ;
          m_vecStarVB[str].y = CParams::WindowHeight;
```

```
        }
    }

    //更新外星人
    for(int i=0; i<m_vecActiveAliens.size(); ++i)
    {
    //如果外星人已"死"去,就用一个新的来代替
    if ( !m_vecActiveAliens [i].Update(m_pGunTurret->m_vecBullets,
                                       m_pGunTurret->m_vPos))
      {
      //首先需要在繁殖群体中进行重新插入,所以死去外星人的适应性分数和代
      //的数目需要记录下来
      m_setAliens.insert(m_vecActiveAliens [i]);

      //如果需要的外星人数已达到,删除 multiset 中性能的最差的那个外星人
      if (m_setAliens.size() >= CParams::iPopSize)
      {
       m_setAliens.erase(--m_setAliens.end());
      }
      ++m_iNumSpawnedFromTheMultiset;

      //如果为运行的早期,则仍然需要添加新的外星人
      if(m_iAliensCreatedSofar <= CParams::iPopSize)
      {
          m_vecActiveAliens [i] = CAlien ();

          ++m_iAliensCreatedSoFar;
      }

      //否则,从 multiset 中选取一个,并对其应用突变
      else
      {
          m_vecActiveAliens [i] = TournamentSelection();

          m_vecActiveAliens [i].Reset();

        if (RandFloat() < 0.8)
          {
              m_vecActiveAliens [i].Mutate();
          }
    }
      }
    }//下一个外星人

  return true;
}
```

当用此程序进行游戏时将会发现，外星人会想尽一切办法来保持自己尽可能地活得长久，并学习如何来对付玩家想杀死他的企图。

10.1.2 运行 BrainyAliens 程序

当运行 BrainyAliens（聪明的外星人）程序时，它一开始启动就以加快了的速度来运行。这样可以使外星人的群体在你开始杀死他们以前进行少量的演化和繁殖。这一预孵化群体包含的外星人数默认为 200，屏幕底部的蓝色进程条指示了程序的进行过程。在这一短时间内，程序进入自动射击（autogun）模式，发射台进行着一种随机的毫无目的的射击，但其过程在屏幕上是无法看到的。当蓝条伸展到屏幕右边时，预孵化完成，程序就把发射台和发射子弹的控制权转交给玩家，这时玩家就可以按照自己的想法操纵发射台和发射子弹。

表 10.1 显示了默认的 Brainy Aliens 程序设置。

表 10.1 Brainy Aliens 工程的默认设置

神经网络的参数	
参　数	设 置 值
隐藏层数目	1
隐藏层神经元数目	15
激励响应	0.2

影响演化的参数	
参　数	设 置 值
突变率	0.2
最大突变干扰	1
外星人培育池大小	200
适应于孵化的百分比	20%
锦标赛竞争者数目	10
预孵化数目	200

其 他 参 数	
参　数	设 置 值
子弹速度	4
被显示外星人的最大数目	10
最大能用子弹数目	3

10.2　练　　习

1. 用不同数目的隐藏层和不同数目的隐藏单元来试验，弄清它们对外星人的性能有什么影响。

2. 演化单独的神经网络，控制每一个外星人向发射台丢炸弹。

3. 尝试不同类型的适应性函数。

4. 创建一个有外星人“波动”的游戏。可以在波动之中将通常的 GA 技术结合到种群的演化中，从而可从两个世界之中得到最好的。

第 11 章　演化神经网络的拓扑

看来，我们已经到达了计算机技术可能到达的那种极限了，虽然说这样话需要小心一些，但在这 5 年中它们确实已经变得相当笨拙了。

John Von Neumann，1949

神经网络的体系结构（architecture）或者说拓扑（topology）对网络的有效性起着十分重要的作用。但同时也应看到，如何为网络体系结构选择各种参数更多地是一种艺术而不是一门科学。这样通常就会使程序蕴藏许多麻烦。虽然可以为此开发一种“感觉”，但使用设计出来的网络演化寻找最优拓扑并不是很理想。要让一个网络学习东西并不难，但要它在学习过程中能合理地进行归纳、不失去推广能力，就不是那么简单了。

当使用演化算法开发神经网络的拓扑时，可以设想有一幅可以用来代表适应性大小的、起伏不平的场景图（landscape），在这一场景图中，每一个点代表了网络结构的一种类型。所谓演化型人工神经网络 EANN（Evolutionary Artificial Neural Network）的目的，就是要在这一块场地上尽可能广泛地进行搜索，直到找到它的全局性最优解。

注意：曾经有人利用某些非进化型的方法处理了这一问题。研究者们在试图创造网络时，有时用建设性的方法，有时则用破坏性的方法。破坏性方法是在超过要求的一个大网络上开始进行的，它有许多神经细胞、细胞层和链接（links）。破坏性方法就是通过训练过程，有系统地进行剪枝来压缩这个网络结构。而建设性的过程则是用相反的方式来得到所要求的结果，它从一个最小的网络开始，通过训练，不断加入神经细胞和细胞间的链接。但这些方法都已发现了有相同的问题，即所得到的网络会收敛到局部最优。并且，在网络结构上也有相当大的局限。也就是说，在所有可能的网络拓扑结构谱中，通常只有其中的一小部分能够利用这些技术来产生。

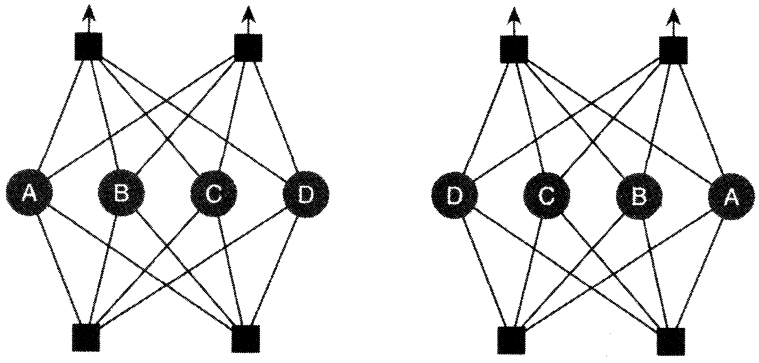
已有不同领域的众多研究人员把大量时间和精力花在这个问题上。我将利用本章的第一部分来描述许多可用技术之中的某一些，而在本章的第二部分介绍一种比较好的简单实现方法。

如同处理其他所有利用演化算法来处理的问题一样，任何可行的解决方案都必须指出网络的编码方法、适应性分数的计算方法以及为执行基因组的突变操作与/或杂交操作的有效方法。我说“或杂交”操作，是因为有些方法完全放弃了这种有潜在麻烦的操作，而宁可全部依靠突变操作在搜索空间进行搜索。因此在介绍一些流行的 EANNs 之前，先说明为什么这里进行的杂交操作会遇到一些问题。

11.1　竞争约定问题

为候选网络进行编码的主要困难之一是所谓的竞争约定问题（competing convention

problem），有时称为结构-功能映射问题（structural-functional mapping problem）。简单地说就是一个编码系统，有可能为呈现完全相同功能的网络而提供若干个不同形式的编码。例如以图 11.1 所示的一对简单网络作为例子，一种简单的编码方式可能会把其隐藏层中神经细胞出现的先后次序作为网络的编码。



网络 1　　网络 2

图 11.1　一种简单编码方式

如果使用这样的简单编码，则网络 1 应编码为：

A B C D

而网络 2 则编码为：

D C B A

但如果仔细地观察一下这两个网络的图形结构就会发现，尽管神经细胞的排列顺序不同，它们的基因组也不相同，但两个网络实质上完全相同。这两个网络将呈现完全精确一致的行为。这就是问题的所在，因为如果把一种杂交操作用于这两个网络，比如在基因组的中间点断裂进行单点杂交，则产生的子孙将是:

A B B A　或　D C C D

而这显然是一个不希望有的结果：两个子代一方面都从父母继承了双份的神经细胞，而另一方面都又失去了父母的 50%的功能（functionaliry），这不是一种性能的改进。

网络的结构越大，这样的问题也会遇到的越频繁。而这种结果在基因组群体上会产生更大的负面效应。因此，这是研究人员在设计编码方案时要尽最大努力去避免的一个问题。

注意：对竞争约定问题还有一些人持不同看法。这些研究人员相信，为避免这个问题而采取的任何措施实际上可能产生更多的问题。他们认为，不如简单地忽略这个问题、允许演化过程处置这样的“残废”网络，或者完全抛弃杂交操作并完全依赖突变操作来遍历搜索空间。

11.2 直接编码

EANN 编码有两种方法：直接编码和间接编码。直接编码，就是直接利用基因组中的神经细胞数目、连接数目等信息的编码来描述网络的准确结构。而间接编码则利用了生长规则（growth rules），这种规则甚至可以用方式来定义网络结构。下面先介绍直接编码的一些例子。

11.2.1 基因子

GENITOR 是已找到的几种最简单的编码方法中的一种，而且也是最早被发现的一种。所谓的 GENITOR 方法也有不同版本，典型的版本是用一个二进制数串来作为基因组的编码。每个基因用 9b 编码。其中第一位用来指示神经细胞之间是否存在有效连接，其余的位代表权重（-127～127）。例如，给定图 11.2 所示的网络，则基因组的编码为：

```
1 10010000  0 00000010  1 01000011  0 00000101  1 10000011
   -16          2           35          5           -3
```

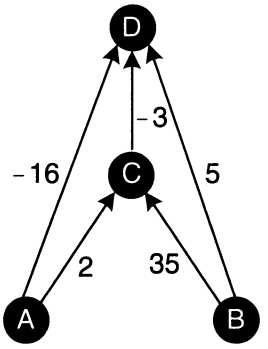


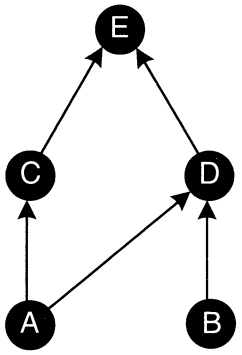
图 11.2 GENITOR 编码

其中用粗体显示的位就是指示联接性（connectivity）的位。编码下面的数字是与该二进制编码对应的十进制数。这是为了易于和图中的数字进行对照而添加的。

这种编码方法的缺点在于，它和迄今已探讨过的许多编码技术一样，为了在基因组内表示所有可能出现的连接，必须为牵涉到的每一个问题预先设计好一个最大的网络拓扑结构。另外，采用这一种编码如果要避免竞争约定问题也是很麻烦的。

11.2.2 二进制矩阵编码

直接编码的一种流行方法是使用一个二进制的邻接矩阵（adjacency matrix）。作为一个例子，观察图 11.3 所示的网络。



	A	B	C	D	E
A	0	0	1	1	0
B	0	0	0	1	0
C	0	0	0	0	1
D	0	0	0	0	1
E	0	0	0	0	0

图 11.3　一个简单的 5 节点网络的二进制矩阵表示

可以看出，这一网络中神经细胞之间的连接关系可以用一个二进制数字（0,1）的矩阵来表示，其中 1 代表对应行和列的神经细胞之间有连接，而 0 代表它们不连接。这样，整个染色体的编码只需将矩阵每一行（或列）的数字赋值给基因即可。如：

```
00110 00010 00001 00001 00000
```

但是，由于所显示的网络为前向网络，这种编码所用代码是很浪费的，因为矩阵元素中总有一半全为零。认识到这一点，可以如图 11.4 那样，仅考虑矩阵的一半，把染色体编码简化成为下面的形式：

```
0110 010 01 1
```

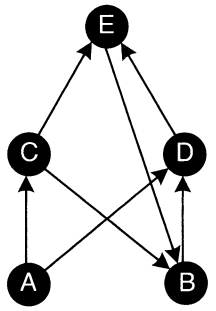
这一编码形式是非常高效的。

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	0	1	0
C	0	0	0	0	1
D	0	0	0	0	1
E	0	0	0	0	0

图 11.4　调整后的矩阵

一旦编码完成之后，二进制数字串就可以通过遗传算法去演化拓扑。在每一个代中对染色体进行译码，所得网络以随机的权重实行初始化。然后训练这一网络，并相应地赋给它一个适应性分数。如果训练机制采用了 backprop 方法，适当性函数的输出可以与所产生的误差成比例关系，为了使网络尺寸达到最小，可以对连接的增加采用附加惩罚。

显然，如果训练方法能够处理任何形式的连接性，不仅是前向网络，整个矩阵都有可能要用来表示相应的连接关系。图 11.5 所示就是一个这样的例子。采用遗传算法的训练方式就可以应用到这一类的网络，但采用标准的反向传播方法则不行。



	A	B	C	D	E
A	0	0	1	0	0
B	0	0	0	1	0
C	0	1	0	0	1
D	1	0	0	0	1
E	0	1	0	0	0

图 11.5　有返回连接的网络

已经证明，当使用矩阵编码（和另一些形式的直接编码）时，随着染色体尺寸的不断增加，演化的性能会变坏。且因尺寸的增加和神经细胞个数的平方成比例，性能变坏的速度极快。这就是可伸缩性问题（scalability problem）。另外，为了创建矩阵，用户必须首先确定最大的网络所用神经细胞的数目。并且，这种类型的表示方法也无法提交和解决前面讨论过的竞争约定问题。当使用这编码时，很有可能两个或更多个的染色体显现相同的性能。如果这些染色体进行配对杂交，则产出的子代的适应性很难有机会超过它们的父母一代。因而一种相当普遍的做法就是完全放弃杂交操作。

11.2.3　基于节点的编码

基于节点的编码处理问题的方法是，在基因中为每个神经细胞要求的全部信息进行编码。对于每一个神经细胞（或节点），它的基因所包含的信息是：与它连结的其他神经细胞，以及与这些连接相联系的权重。在基于节点的编码中，有的编码很复杂，把相关的激励函数和学习率也包括在其中（当网络采用反向传播那样的梯度下降法来训练时，就需要利用到学习率）。

由于本章的代码工程采用了基于节点的编码，稍后我会非常详细地来介绍这一技术，下面举例说明如何为网络的连接关系进行编码。

图 11.6 显示了两个简单的网络和它们的染色体。每个基因包含一个节点标识和一个进入该节点的节点连接表。如果编写代码，一个简化的基因和基因组结构将有下面的形式：

```
struct SGene
{
    int             NodeID;
    vector<Node*>   vecpNodes;
}

struct SGenome
{
    vector<SGene>   chromosome;
    double          fitness;
};
```

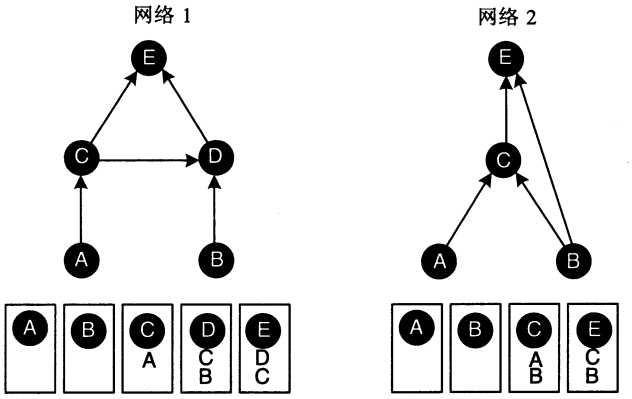


图 11.6　基于节点的编码

使用这种编码的突变操作可以有不同类型，并且也都很容易实现。它们可能的突变包括：增加一个链接、删除一个链接、增加一个节点、删除一个节点。但杂交操作与之不同。为了保证杂交所产生的子代确实有效，避免任何神经细胞出现既无输入又无输出的这种问题，必须非常谨慎地来从事这种操作。

图 11.7 显示了由图 11.6 中的两个染色体的第 3 个基因（“C”基因）配对后所产生的子代。

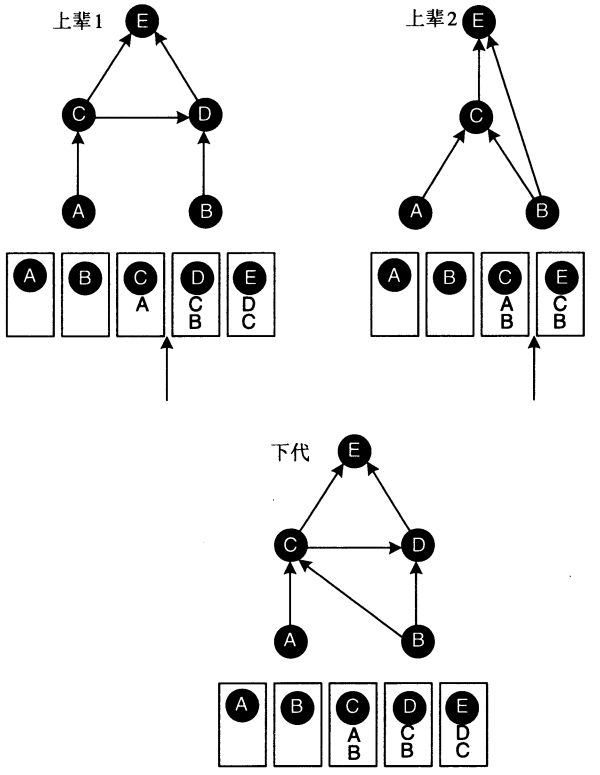


图 11.7　杂交操作过程

有效的遗传算法操作一旦定义了之后，利用所描述方式编码的神经网络可按下列的步骤来进行演化（假设它们采用了和类似反向传播那样的梯度下降算法相关的训练集）：

1．创建染色体的一个随机的初始群体。

2．训练各个网络，并根据每个网络的总体误差值来分配适应性分数（误差即目标输出减去训练最佳的输出）。也可以给尺寸增长得快的网络打一个惩罚分。这有利于具有较少神经细胞和链接的群体。

3．使用喜欢的选择法（如锦标赛选择或与适应性成比例的选择等）选取两个父代。

4．在合适的场合使用杂交操作。

5．在合适的场合使用突变操作。

6．重复步骤 3、4 和 5，直到新的群体创建出来。

7．转到步骤 2 重复，直到演化出满意的网络时结束。

在本章后面将介绍如何利用基于节点的编码去演化拓扑并同时演化连接权重。

11.2.4 基于路径的编码

用基于路径的编码定义神经网络结构的办法是：为每一条从输入神经细胞到输出神经细胞之间的路径进行编码。例如，已知图 11.8 所描述的网络时，则所需路径是：

1 → A → C → 3
1 → D → B → 4
1 → D → C → 3
2 → D → C → 3
2 → D → B → 4

因每条路径总是从一个输入神经细胞出发，并总是到一个输出神经细胞结束，这种类型的编码确保了在染色体中不会出现无用的神经细胞。重组所用的操作是两点杂交（这保证染色体总是以输入和输出神经细胞为边界点）。突变操作有下面几种典型的：

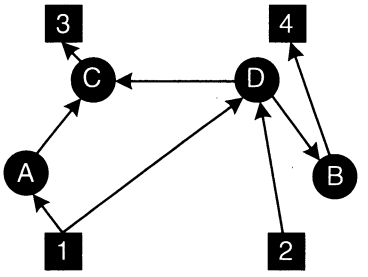


图 11.8 基于路径的编码

- 创建一条新路径并插入到染色体。
- 选择一节（section）路径并加以删除。
- 选择一段（segment）路径并插入一个神经细胞。
- 选择一段（segment）路径并删除一个神经细胞。

由于利用这种编码类型所定义的网络可以不限于前向网络（也就是说，链接可以是循环的），为了确定理想的连接权重，必须使用遗传算法那样的训练方法。

11.3　间 接 编 码

间接编码更精密地模仿了生物系统中由基因型映射到表现型的方式，并通常也导致产生更为紧凑的基因组。在生物机体中，每一个基因并不能单独引发一个物理特征；而是靠不同排列的基因之间的相互作用表征出来的。间接编码技术就是要在染色体中通过一系列生长规则（growth rules）来模拟这种机制。这些规则通常要同时指定许多连接。下面具体介绍两个这样的技术，这样可以获得对它们工作原理的一些感性认识。

11.3.1　基于语法的编码

其于语法的编码使用了可以表示为语法规则的一系列开发规则。这种语法由一系列的左手边符（Left-handside symbol,LHS）和右手边符（right-hand side symbol,RHS）组成。在开发过程中，如果遇到 LHS 标志，就要用几个 RHS 标志来代替它。开发过程是从一个称为起始符（start symbol）的 LHS 标志开始的，然后从一套产生规则（production rules）中选择一个来创建一组新的符号集，并再把生产规则用到这一组新符号，这样不断下去，直到遇到一组终止符（terminal symbol）为止。到此，开发过程就停止进行，而最终的符号即代表一个表现型。

这里是用图表来举例说明这种难于理解的方法。图 11.9 是一套生产规则的例子。

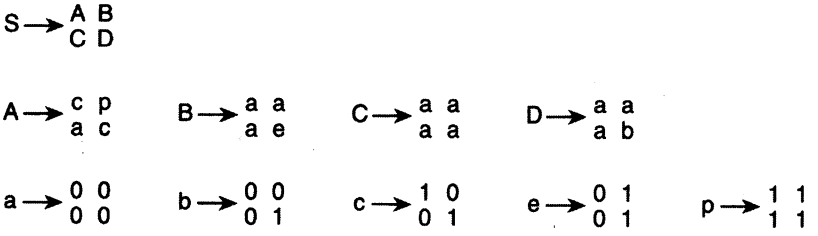


图 11.9　基于语法的编码产生规则的例子

其中 S 是起始符，1 和 0 都是终止符。观察一下如何从起始符 S 开始，利用上述的语法规则来进行不断的替代，直到达到结束符号。这里一共进行了 4 步就完成了。从图中可以清楚地看到，最终得到了一个可以构造表现型的二进制矩阵。

利用了一个遗传算法对生长规则进行演化。在染色体中，每条规则是用与此规则的 RHS 符所对应的 4 个位置来表示的。规则沿染色体长度的实际位置（它的基因位置，loci）决定它的 LHS 符。非结束符可以采用任何字母来表示。本项技术的发明者使用了从 A 到 Z 和从 a 到 p 间的各个字符。因包含结束符的规则的 RHS 符已规定为 0,1，故染色体只需对由非结束符组成的规则进行编码。由此，图 11.10 所示例子的染色体应为：

ABCD cpac aaae aaaa aaab

其中，最前面的 4 个位置对应于起始标志 S，接下去的 4 个位置，即 cpac 占有的位置，对应于 LHS 标志 A 等。

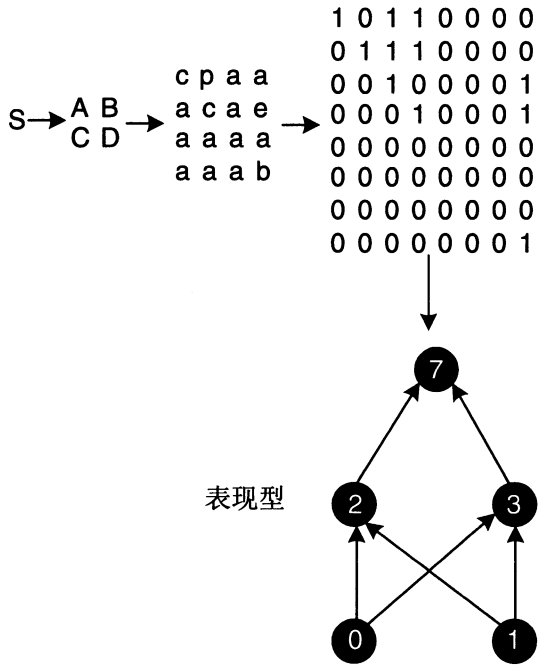


图 11.10　按生长规则来进行生长

11.3.2　二维生长编码

二维生长编码是一种特殊的编码类型。神经细胞用平面上具有固定位置的点来表示，而算法利用了一种能实际生长出许多轴突（Axon）的规则，它们就像植物的卷须一样通过空间到达另一些神经细胞。当轴突接触到另一神经细胞时就产生了一个连接。这里用图 11.11 举例说明以了解发生的过程。

图 11.11 的左边显示了神经细胞的所有轴突正在向外生长，而右边的图则显示了其中已经建立起连接的地方。

这一技术的设计者们使用一种带有 40 个块（block）的基因组编码，每一个 block 代表一个神经细胞。基因组的前 5 个 block 用来代表输入神经细胞，最后 5 个代表输出神经细胞，其余 30 个 block 作为隐藏神经细胞使用。

每一个 block 有 8 个基因：

- Gene1 决定神经细胞是否存在。
- Gene2 是神经细胞在 2D 空间里的 X 位置。
- Gene3 是神经细胞在 2D 空间里的 Y 位置。
- Gene4 是轴突生长规则的分叉角。轴突每次分枝时都使用这个角度。

- Gene5 是每个轴突段的长度。
- Gene6 是连接权重。
- Gene7 是偏移值。
- Gene8 是一个神经细胞类型基因。这一基因在原始的试验中用来确定输入神经细胞所代表的是哪一个输入。

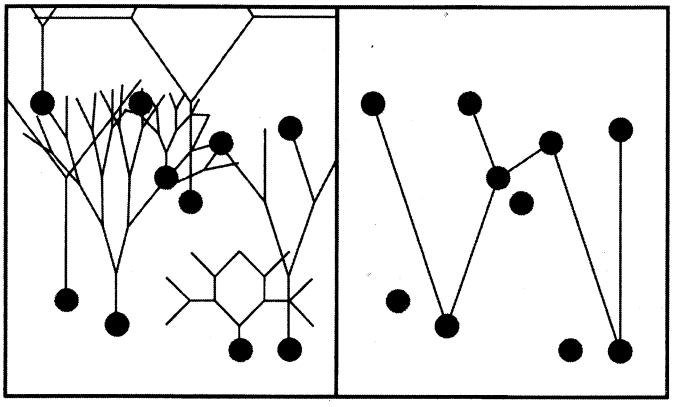


图 11.11　轴突从固定在 2D 空间里的神经细胞向外生长

可以想象，这种技术实现起来非常困难并且演化速度也相当慢。因此，它没有实际的用处。

下面将要介绍一种利用基于节点编码的奇特方法，它可以从头开始长出一个神经网络。

11.4　拓扑扩张的神经演化

NEAT 是拓扑扩张的神经演化（Neuro Evolution of Augmenting Topologies）的简称。这是得克萨斯州大学 Kenneth Stanley Owen 和 Risto Miikkulainen 合作开发的一种网络。它采用基于节点的编码来描述网络的结构和连接权重。当创建新的节点和链接时，它借助于所产生的历史数据，避免了竞争约定问题，是一种很好的方法。NEAT 也企图把它所生成的网络的尺寸减到最小。为此，在演化开始时，它让群体中每个神经网络都有最小的拓扑结构，然后在演化进行中，始终都是一个一个地在网络中添加神经细胞和连接。由于这种过程和自然界所有的生物机体的生长过程一样——随着时间的推移，不断发育长大、不断增加复杂性——因而，这是一种非常吸引人的解决方法，而这也是我在本章中要特别着重介绍这种技术的部分原因。

为了使这一思想得到实现，需要利用相当多的代码。当我描述 NEAT 演化规则的每一部分时我都会把有关代码列出来。我将按照程序固有的顺序来进行讲解，这样做，源程序代码本身就能帮助你加强对注释文本的理解、帮助你快速掌握概念。本章的所有源程序均可在光盘的 Chapter11/NEAT Sweepers 文件夹下找到。

首先让我描述网络怎么编码。

11.4.1 NEAT 基因组

NEAT 基因组的结构包括一张神经细胞基因（neuron gene [①]）表和一张链接基因（link gene）表。一个链接基因的信息包括：它所连接的两个神经细胞、与此连接相关联的权重、用来说明链接是否已启用（enabled）的一个标帜、用来说明链接是否为返回（recurrent）的一个标帜，此外还有一个创新号（innovation number）。神经细胞基因描述了该神经细胞在网络内的功能，它可以是一个输入神经细胞[②]，或者是一个输出神经细胞，或者一个隐藏神经细胞，再或者是一个偏移神经细胞。每个神经细胞基因也拥有一个惟一的标识号（ID）。

图 11.12 显示了一个简单网络的基因组的两种基因表。

11.4.1.1 链接基因的结构

链接基因的结构称为 SlinkGene，它放在文件 genes.h 中，其定义如下：

```
Struct SlinkGene
{
  //该链接（link）所连结的两个神经细胞的标识（ID）
  int         FromNeuron,
              ToNeuron;
  double      dWeight;
```

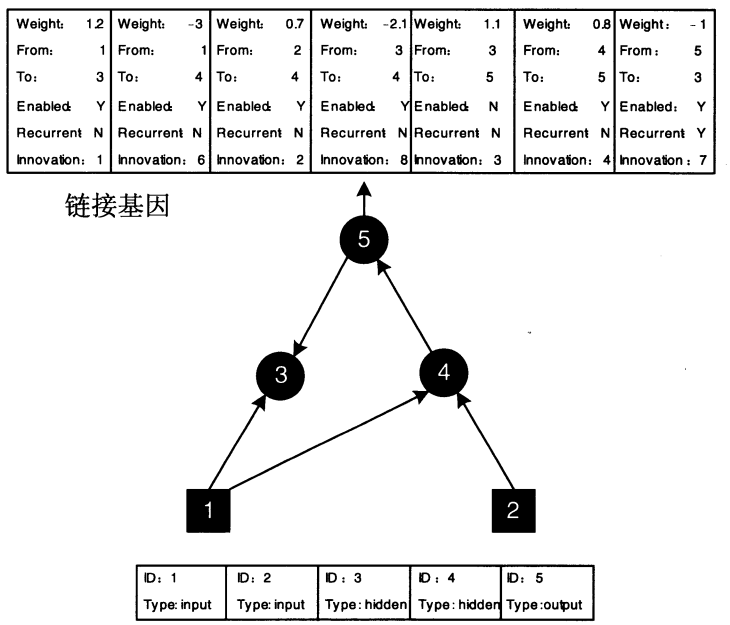


图 11.12 为 NEAT 网络编码

[①] 译者注：由注 1 可知，如果把“神经细胞基因”改称“节点基因”就可以更确切些。

[②] 译者注：“输入神经细胞”和“偏移神经细胞”都是 NEAT 的演化对象，在演化结果中都不代表神经细胞。

```
//指明本 link 当前是否为 Enabled 的标志（flag）
bool     bEnabled;

//指明本 link 是否为 Recurrent 的标志
bool     bRecurrent;
//下面具体介绍这一个值
int      InnovationID;
SLinkGene(){}
SlinkGene (int       in,
           int       out,
           bool      enable,
           int       tag,
           double    w,
           bool      rec = false):bEnabled(enable),
           InnovationID(tag),
           FromNeuron(in),
           ToNeuron(out),
           dWeight(w),
           bRecurrent(rec)
   {}
   // 重载“<”运算符用于排序（以创新标识 ID 作为应用范畴）
   friend bool operator<(const SLinkGene& lhs, const SLinkGene& rhs)
    {
      return (lhs.InnovationID < rhs.InnovationID);
    }
};
```

11.4.1.2　神经细胞基因结构

神经细胞基因的结构体记为 SNeuronGene，其代码可在 genes.h 中找到，下面是它的定义：

```
struct SNeuronGene
{
    //它的标识号码
    int      iID;

    //它的类型
    neuron_type NeuronType;
```

这是一个枚举类型，其值为 input、hidden、bias、output 和 none。有关 none 类型如何使用，在讨论 innovations（创新）时就会知道。

```
    //它是 Recurrent 的吗?
    bool bRecurrent;
```

在 NEAT 中，一个自返神经细胞（recurrent neuron）定义为存在一个环形自返的链接的神经细胞，如图 11.13 所示。

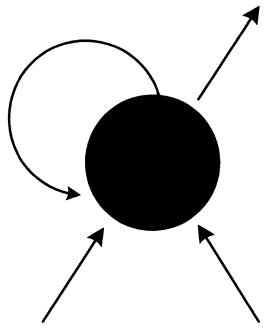


图 11.13　带有两个进入链接的神经细胞：一个向外的链接和一个环形自返链接

```
//设置 S 形函数的曲率（弯曲性）
double dActivationResponse;
```

这里，每个神经细胞的 S 形函数的激励响应也应分别地进行演化。

```
//在网格中的位置
double dSplitY, dSplitX;
```

如果要在一个二维格点上布置（laid out）一个神经网络，就必须了解每个神经细胞在格子上的坐标。利用这一信息，还可以在显示器上画出这个网络，这能为用户提供视觉上的帮助。

当一基因组最初建立时，所有神经细胞都指定了一个平面坐标（SplitX,SplitY）。现在只讨论它们的 Y 坐标值 SplitY，但 X 坐标值 SplitX 也可以用类似方法进行计算。每个输入神经细胞的 SplitY 值指定为 0，每个输出神经细胞的 SplitY 值指定为 1。当一个神经细胞加入到输入输出之间时，就应有效地断裂开一个链接，新的神经细胞的 SplitY 值取为它上下两个邻近接点的中点值。图 11.14 可以帮助阐明这一点。

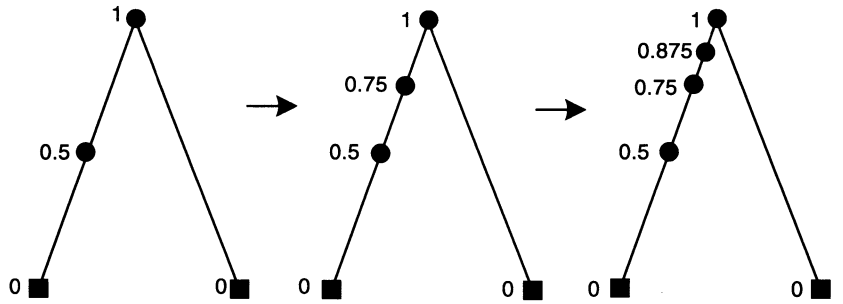


图 11.14　计算 SplitY 深度值的几个例子

这一信息除了可用在网络绘图程序中计算显示坐标外，在计算整个网络的深度（depth）以及确定一个新加入的链接是否返回链接也是极有用的。

```
SNeuronGene (neuron_type type,
             int         id,
             double      y,
```

```
                double      x,
                bool        r = false ):iID(id),
                                        NeuronType(type),
                                        bRecurrent(r),
                                        pReuronMarker(NULL),
                                        dSplitV(y),
                                        dSplitX(x)
    {}
};
```

11.4.1.3　基因组类

下面为基因组类 CGenome 的定义，其中包含的某些方法和成员可能现在还无法理解，现在只需快速浏览一下这个类，然后就转到下一节去（为了简单起见，已删去了访问方法）。

```
class CGenome
{
private:

    //它的标识（ID）
    int                 m_GenomeID;

    //组成此基因组的所有神经细胞
    vector<SNeuronGene> m_vecNeurons;

    //所有的链接
    vector<SLinkGene>   m_vecLinks;

    //指向它的表现型的指针
    CNeuralNet*         m_pPhenotype;

    //它的原始适应性分数
    double              m_dFitness;

    //当它已被放置进一物种并已作了相应调整后的适当性分数
    double              m_dAdjustedFitness;

    //要求孵化的下一代的子孙数目
    double              m_dAmountToSpawn;

    //分别用来保存输入和输出数目的两个记录
    int                 m_iNumInputs,
                        m_iNumOutputs;

    //保存该基因组进入的物种的轨迹（仅用于显示目的）
    int                 m_iSpecies;

    //如果指定的链接已是基因组的一个部分，返回 true
```

```
    bool DuplicateLink(int NeuronIn, int NeuronOut);

    //给定一个神经细胞 ID 时,本函数就能找到它在 m_vecNeurons 中的位置
    int GetElementPos(int neuron_id);

    //测试传入的 ID 是否与已经存在的某个神经细胞 ID 相同
    //这一测试在 AddNeuron 中需要使用
    bool AlreadyHaveThisNeuronID (const int ID);

publlc:

    CGenome ();
    //本构造函数创建了一个最小的基因,它有输出与输入神经细胞,但每个输入神
    //经细胞都连结到每个输出神经细胞
    CGenome (int id, int inputs, int outputs);

    //此构造函数利用一个 SLinkGenes 向量,一个 SneuronGenes 向量
    //和一个标识号 ID 来创建一个基因组
    CGenome (int                          id,
             vector<SNeuronGene>          neurons,
             vector<SLinkGene>            genes,
             int                          inputs,
             int                          outputs);
~CGenome ();

    //复制构造函数
    CGenome (const CGenome& g);

    //为 operator 赋值
CGenome& operator = (const CGenome& g);

    //由基因组创建神经网络
    CNeuralNet*      CreatePhenotype(int depth);

    //删除神经网络
    int              DeletePhenotype();

    //按突变率在基因组中增加一个链接
    void                  AddLink(double      MutationRate,
                                  double      ChanceOfRecurrent,
                                  CInnovation &innovation,
                                  int         NumTrysToFindLoop,
                                  int         NumTrysToAddLink);

    //增加一个神经细胞
    void               AddNeuron(double      MutationRate,
                                 CInnovation &innovation,
```

```
                int             NumTrysToFindOldLink);
    //这一函数对连接权量实行突变
    void              MutateWeights(double mut_rate,
                                    double prob_new_mut,
                                    double dMaxPertubation);

    //干扰神经细胞的激励响应
    void     MutateActivationResponse(double mut_rate,
                                      double MaxPertubation);

    //计算本基因组和其他基因组之间的兼容性分
    double   GetCompatibilityScore(const CGenome &genome);

    void         SortGenes();

    //重载 "<"运算符用于排序,排序的方法按照由最合适到最不合适的顺序
    friend bool operator<(const CGenome& lhs, const CGenome& rhs)
    {
      return(lhs.m_dFitness > rhs.m_dFitness);
    }
};
```

11.4.2　算子和创新

下面讨论基因组可用的突变方法。在 NEAT 的实现（implementation）过程中，基因组中使用了 4 种突变操作（算子）：加入一个链接基因的突变、加入一个神经细胞基因的突变、干扰一个连接权重的突变、能够为每个神经细胞改变激励函数响应的突变。连接权重的突变操作和其他部分的突变操作类似，这里不再介绍。它是根据突变率预先规定的范围对各个连接权重进行干扰的。但有一个差别，即这里有可能用一个全新的权重来替代原有权重。这种情况的出现概率可以用参数 dProbabilityWeightReplaced 来设置。

当有一个新的结构加入基因组时，就会出现一个创新（inovation）。创新可以通过增加一个链接基因来实现，也可以通过增加一个神经细胞基因来实现。创新实际就是这种变化的记录。所有的创新由一个全局数据库来维护，每一个创新有它自己惟一的标识号（identification number）。当每次有一个链接或神经细胞加入时，都需要查看数据库，以检查该创新以前是否已经建立，如果它已经存在，则新基因就被分配一个已存在的创新的标识号。如果库中原来不存在，则要创建一个新的创新，并加入数据库，而基因则用新的创新的 ID 来标注。

假设正在演化一个具有两个输入和一个输出的网络。图 11.15 中左边的图描述了这种网络的基本结构。基因组群体中每一个成员开始都具有这样的简单结构。右边所画的网络代表发生了在原有网络中加入一个神经细胞的突变之后的结果。当加入神经细胞 4 的时候，3 个创新被创建了：一个是加入神经细胞 4 的创新，另外两个是在神经细胞 1 和 4 与 4 和 3 之间加入链接的创新。（原有 1 和 3 之间的链接在基因组中仍然存在，但它已被不可用了）。

每一个创新均被记录在 SInnovation 结构中，这一结构的定义如下：

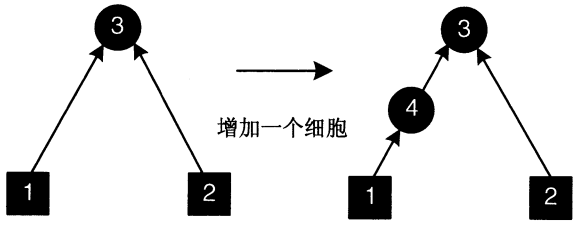


图 11.15　添加一个神经细胞的突变

```
struct SInnovation
 {
    //创新类型是加入新神经细胞还是新链接
    innov_type    InnovationType;
    int           InnovationID;
    int           NeuronIn;
    int           NeuronOut;
    int           NeuronID;
    neuron_type   NeuronType;
    /*构造函数和无关的成员均已略去*/
 };
```

创新的类型既可以是 new_neuron（新神经细胞），也可以是 new_link（新链接）。创新结构 Sinnovation 和创新类 CInnovation 的定义可以在文件 CInnovation.h 中找到，它们保存了所有创新的变化轨迹（track）。

因 NEAT 扩张结构靠增加神经细胞和增加链接实现，所以在初始群体中所有的基因组都从代表同一最小拓扑的结构开始（但是用不同的连接权重）。当各基因组被创建时，程序自动为所有的初始神经细胞和初试链接定义了创新。作为其结果，在如图 11.15 所示的突变发生之前的创新数据库将具有表 11.1 所示的形式。

表 11.1　增加神经元之前的创新

创新 ID	创 新 类 型	输　　入	输　　出	神经细胞 ID	神经细胞类型
1	new_neuron	-1	-1	1	input
2	new_neuron	-1	-1	2	input
3	new_neuron	-1	-1	3	output
4	new_link	1	3	-1	none
5	new_link	2	3	-1	none

为避免混淆，新神经细胞创新的 In 和 Out 被赋予-1 值。同样，新链接的 Neuron ID 也被分配了-1 值。

当图 11.15 添加了神经细胞 4 以后，创新数据库的内容将有所增长，如表 11.2 所示，它包含了 3 个新的创新。

表 11.2　增加神经细胞之后的创新

创新 ID	创 新 类 型	输 入	输 出	神经细胞 ID	神经细胞类型
1	new_neuron	-1	-1	1	input
2	new_neuron	-1	-1	2	input
3	new_neuron	-1	-1	3	output
4	new_link	1	3	-1	none
5	new_link	2	3	-1	none
6	new_neuron	1	3	4	hiden
7	new_link	1	4	-1	none
8	new_link	4	3	-1	none

如果将来任何时候有一个不同的基因组偶然出现与此相同的突变（加入神经细胞 4），程序就要参考创新数据库，并为最新创建的基因分配一个正确的创新标识号。这样，该基因就包含了任何结构变化的历史记录。为了设计一个有效的杂交操作，这一个信息是极为有用的。

下面浏览一下加入链接和加入神经细胞，这两种突变的代码为 AddLink 和 AddNeuron。

11.4.2.1　CGenome::AddLink 方法

本算子加入下列 3 种不同链接中的一种：

- 正向链接（forward link）。
- 返回链接（recurrent link）。
- 自环的返回链接（looped recurrent link）。

图 11.16 是各种不同类型的链接的例子。

下面是在基因组中增加 link 的代码。笔者在需要的地方已经添加了更多的注释。

```
void CGenome::AddLink (double          MutationRate,
                       double          ChanceOfLooped,
                       CInnovation     &Innovation,    //创新数据库
                       int             NumTrysToFindLoop,
                       int             NumTrysToAddLink)
{
  //根据突变率来确定是否立即返回
  if (RandFloat() > MutationRate) return;

  //确定要连接的两个神经细胞的持有者。如果要连接的是两个有效的神经细胞
  //这些值将变成 >= 0
  int ID_neuron1 = -1;
  int ID_neuron2 = -1;
```

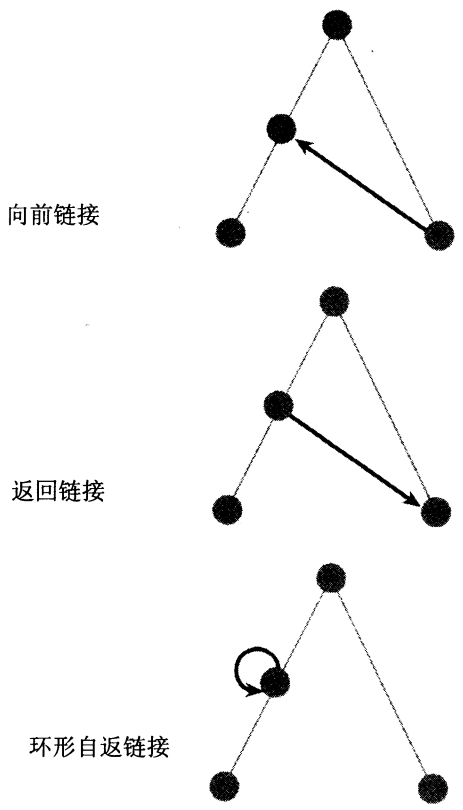


图 11.16　不同类型的链接（link）

```
//如果选择加入循环链接,则此标志为 true
bool bRecurrent = false;
//首先检查需要创建的链接是否返回到同一个神经细胞本身
if (RandFloat() < ChanceOfLooped)
  {
  //如果是,则重复试验 NumTrysToFindLoop 次,寻找一个既不是输入也不是
  //偏移且没有一个环形自返链接的神经细胞
   while(NumTrysToFindLoop--)
   {
   //任取一个神经细胞
   int NeuronPos = RandInt(m_iNumInputs+1, m_vecNeurons.size()-1);

   //做检查以确保神经细胞没有自返链接,也不是一个输入或偏移神经细胞
    if (!m_vecNeurons[NeuronPos].bRecurrent &&
       (m_vecNeurons[NeuronPos].NeuronType != bias) &&
       (m_vecNeurons[NeuronPos].NeuronType != input))
    {
    ID_neuron1 = ID_neuron2 = m_vecNeurons[NeuronPos].iID;

    m_vecNeurons[NeuronPos].bRecurrent = true;
```

```
        bRecurrent = true;

        NumTrysToFindLoop = 0;
       }
     }
   }
```

首先，程序检查是否有可能加入了一个环形自返链接。如果是，程序就要重复试验 NumTrysToFindLoop 次，去寻找一个相应的神经细胞。如果没有找到神经细胞，程序继续寻找两个不链接的神经细胞。

```
    else
    {
    //如果为否：试着去寻找两个不链接的神经细胞。一共要尝试 NumTrysToAddLink 次
     while(NumTrysToAddLink--)
      {
```

由于某些网络可能在所有神经细胞之间都已经存在链接，所以当代码试图寻找两个不链接的神经细胞时，必须保证不进入一个无限循环。为防止这种情况发生，程序在寻找两个不链接神经细胞时只试验 NumTrysToAddLink 次。这一数字在 CParams.cpp 中设置。

```
      //选择两个神经细胞,第二个不能是输入或偏移神经细胞
      ID_neuron1 = m_vecNeurons[RandInt(0,
                              m_vecNeurons.size()-1)].iID;
      ID_neuron2 = m_vecNeurons[RandInt(m_iNumInputs+1,
      m_vecNeurons.size()-1)].iID;
      if (ID_neuron2 == 2)
       {
         continue;
       }

      //保证这两个神经细胞没有链接,且也不是同一个神经细胞
      if (!(DuplicateLink(ID_neuron1, ID_neuron2) ||
                                    (ID_neuron1 == ID_neuron2)))
        {
          NumTrysToAddLink = 0;
        }

       else
        {
          ID_neuron1 = -1;
          ID_neuron2 = -1;
        }
       }
     }

     //如果寻找链接不成功,则立刻返回
     if ( (ID_neuron1 < 0) || (ID_neuron2 < 0))
```

```
    {
return;
    }

  //检查这一创新是否此前已经创建过了
  int Id = Innovation.CheckInnovation (ID_neuron1, ID_neuron2,
                                       new_link);
```

这时代码检查创新数据库，以弄清此链接是否已在其他基因组中出现。如果链接到的是新的创新，CheckInnovation 返回-1，否则 CheckInnovation 返回原有创新标识号。

```
//检查此链接是返回的吗?
if (m_vecNeurons[GetElementPos(ID_neuron1)].dSplitY >
    m_VecNeurons[GetElementPos(ID_neuron2)].dSplitY)
 {
   bRecurrent = true;
 }
```

这里，通过两个神经细胞的 splitY 值的比较，以确定链接是向前的还是向后的。如果 neuron1 的 SplitY 大于 neuron2 的 SplitY，则链接为向后的，构成返回链接。

```
if(id < 0)
{
//需要创建一个新的创新
innovation.CreateNewInnovation(ID_neuron1,ID_neuron2, new_link);

//创建新的基因
int id = innovation.NextNumber() - 1;
```

如果程序进入代码的这一段，则说明创新是一个新的创新。在创建新的基因前，应先把创新加到数据库并寻找出一个标识号，新基因就用此标识号来标注。

```
    SLinkGene NewGene( ID_neuron1,
                       ID_neuron2,
                       true,
                       id,
                       RandomClamped(),
                       bRecurrent);

  m_vecLinks.push_back(NewGene);
}

   else
    {
      //此创新已存在,下面要做的就是使用已存在的创新标识来创建新基因
      SLinkGene NewGene(ID_neuron1,
                        ID_neuron2,
                        true,
                        Id,
```

```
                        RandomClamped(),
                        bRecurrent);

        m_vecLinks.push_back(NewGene);
      }

    return;
}
```

11.4.2.2　CGenome::AddNeuron 方法

要把一个神经细胞加到网络，首先应选择一个链并让它 disabled（禁止使用）。然后创建两个新的链接把新神经细胞和它的两个邻近细胞链接起来，如图 11.17 所示。

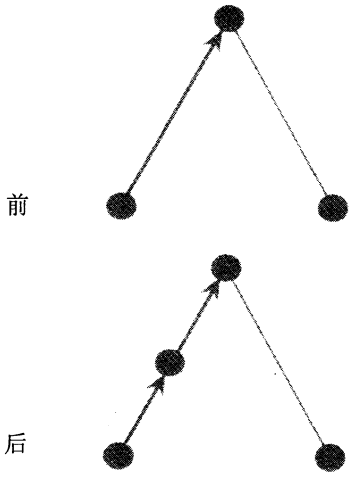


图 11.17　把一神经细胞加到网络

这意味着，每次加入一个神经细胞，共需建立 3 个创新：一个是为神经细胞基因，两个是为链接基因，如果它们已经存在，则需要复制。

```
void CGenome::AddNeuron (double          MutationRate,
                         CInnovation     &innovations, //创新数据库
                         int             NumTrysToFindOldLink)
{
    //根据突变率来确定返回与否
    if (RandFloat() > MutationRate) return;

    //如果找到了要插入新神经细胞的有效链接,则此值将设置为 true
    bool bDone = false;
//这将用来保存所选链接基因在 m_vecLinks 中的索引（数组下标）
int ChosenLink = 0;
```

在网络演化早期，有可能出现这样一个问题：同一链接被不断地要求断裂、插入、断裂、插入，从而形成一个“链条”状的效果，如图 11.18 所示。

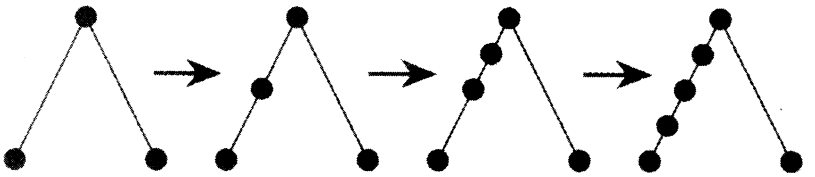


图 11.18　链接形成了链条状

显然这是我们不希望的，所以下面的代码要检查基因组中神经细胞的个数，判断其结构是否在某一阈值之下。如果是，就可选择旧链接而不选较新的链接。

```
//首先选择一个进行断裂的链接。如果基因组很小,则代码必须对原有的旧链接实
//行断裂,以保证不出现一连串的链条连接。这里规定,如果基因组包含的隐
//藏神经细胞少于 5 个,就认为它是太小了,就不能在 NumGenes-1 个链接中作随机
//选择,必须采取其他选择法
 const int SizeThreshold = m_iNumInputs + m_INumOutputs + 5;
 if (m_vecLinks.size() < SizeThreshold)
   {
    while(NumTrysToFindOldLink--)
     {
       //在基因组中选择一个相对于原有链接有偏移的较早的链接
       ChosenLink = RandInt(0, NumGenes()-1-(int)sqrt(NumGenes()));

       //保证该链接已被 enabled 并且它不是一个返回链接或带有偏移输入
 int FromNeuron - m_vecLinks[ChosenLink].FromNeuron;

     if ((m_vecLinks[ChosenLink].bEnabled) &&
         (!m_vecLinks[ChosenLink].bRecurrent) &&
         (m_vecNeurons[GetElementPos(FromNeuron)].NeuronType
                                                  != bias))
      {
         bDone - true;
         NumTrysToFindOldLink = 0;
      }
     }
    if (!bDone)
     {
       //寻找下一链接的工作失败
       return;
     }
  }

 else
 {
   //基因组具有足够的尺寸去接受任何链接
   while (!bDone)
    {
```

```
    ChosenLink = RandInt(0, NumGenes()-1);

    //保证链接已被 enabled,且它不是链接,也不是一个有偏移输入
    int FromNeuron = m_vecLinks[ChosenLink].FromNeuron;

    if ((m_vecLinks[ChosenLink].bEnabled) &&
        (!m_vecLinks[ChosenLink].bRecurrent) &&
        (m-vecNeurons[GetElementPos(FromNeuron)].NearonType
!= bias) )
    {
      bDone = true;
    }
   }
  }

//到此,链接已选中,下一步进行神经细胞的插入。首先禁止掉该链接基因
m_vecLinks[ChosenLink].bEnabled = false;

//再从该基因取得权重(用它作为新加入的一个链接的权重,这样可使链接的
//断裂不至于扰乱神经网络已经学习得到的东西)
double OriginalWeight = m_vecLinks[ChosenLink].dWeight;
```

当原有的一个链接被禁止，而新的两个链接建立后，原有链接的权重将用来作为新链接中的一个的权重，而另一个新链接的权重则设置为 1，如图 11.19 所示。这样可以使添加神经细胞之后不至于对网络原来已经学习到的行为有太大影响。

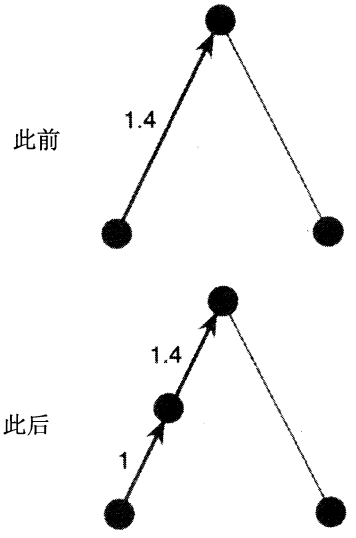


图 11.19　把原有链接的权重赋给一个新的链接基因

```
    //标识这个链接所连接的两个神经细胞
    int from = m_vecLinks[ChosenLink].FromNeuron;
    int to = m_vecLinks[ChosenLink].ToNeuron;
```

```
//计算新的神经细胞的深度和宽度,利用深度来确定链接的向前或向后
double NewDepth = (m_vecNeurons[GetElementPos(from)].dSplitY +
                   m_vecNeurons[GetElementPos(to)].dSplitY)/2;

double NewWidth = (m_vecNeurons[GetElementPos(from)].dSplitX +
                   m_vecNeurons[GetElementPos(to)].dSplitX)/2;

//检查这一创新是否以前已被群体中其他成员创建过
int id = innovations.CheckInnovation (from,
                                      to,
                                      new_neuron);

/*  NEAT 可能重复做的事情有下列几种:

   1. 寻找一个 link。这里假设选择的是 link1 到 link5 中的一个
   2. 禁止这个 link
   3. 增加一个新的神经细胞和两个新的 link
   4. 如果后一个基因组也有同样的 link 但没有被禁止的话,由第 2 步禁止的 link 有可
      能在此基因组与另一基因组重组时被重新启用
因此,下列的代码用来检查一个神经细胞标识号是否已经在使用。如果是,则函数要为神经细胞
创建一个新的创新 */
if (id >= 0)
 {
  int NeuronID = innovations.GetNeuronID(id);
  if (AlreadyHaveThisNeuronID (NeuronID))
   {
    id = -1;
   }
 }
```

如果此基因组中已存在一个具有同样标识号的神经细胞，AlreadyHaveThisNeuronID 返回 true。如果是这种情况，则应创建一个新的创新，因此 id 被设置成-1。

```
if (id < 0) // 这是一个新的创新
 {
  //为新的神经细胞加入创新
  int NewNeuronID = innovations.CreateNewInnovation(from,
                                                    to,
                                                    new_neuron,
                                                    hidden,
                                                    NewWidth,
                                                    NewDepth);

  //创建新的神经细胞基因并将它加入基因组
  m_vebNeurons.push_back(SNeuronGene (hidden,
                                      NewNeuronID,
                                      NewDepth,
                                      NewWidth));
```

```
    //需要两个新的链接创新。当基因断裂而创建两个新链接时,每一个新的链接都需
    //要一个链接创新

    //-----------------------------------第一个链接

    //产生下一个创新标识号
    int idLink1 = innovations.NextNumber();

    //创建新的创新
    innovations.CreateNewInnovation(from,
                                    NewNeuronID,
                                    new_link);

    //创建新的基因
    SLinkGene link1(from,
                    NewNeuronID,
                    true,
                    idLink1,
                    1.0);

    m_vecLinks.push_back(link1);

    //-----------------------------------第 2 个链接

    //产生下一个创新标识号
    int idLink2 = innovations.NextNumber();

    //创建新的创新
     innovations.CreateNewInnovation(NewNeuronID,
                                     to,
                                     new_link);
    //创建新基因
     SLinkGene link2(NewNeuronID,
                     to,
                     true,
                     idLink2,
                     OriginalWeight);

     m_vecLinks.push_back(link2);
}

else //存在着的创新
{
    //因该创新已经建立,故可从创新数据库得到相关的神经细胞和链接信息
    int NewNeuronID = innovations.GetNeuronID(id);
```

```
        //为两个新链接基因生成创新标识号
        int idLink1 = innovations.CheckInnovation (from, NewNeuronID,
                                                   new_link);
        int idLink2 = innovations.CheckInnovation (NewNeuronID, to,
                                                   new_link);

        //下面的情况应该永远不会发生,因为创新*应该*已经出现
        if ( (idLink1 < 0) || (idLink2 < 0) )
        {
          MessageBox(NULL, "Error in CGenome::AddNode",
                     "Problem!", MB_OK);
          return;
        }

        //创建两个新基因来代表新的链接
        SLinkGene link1(from, NewNeuronID, true, idLink1, 1.0);
        SLinkGene link2(NewNeuronID, to, true, idLink2,
                        OriginalWeight);

        m_vecLinks.push_back(link1);
        m_vecLinks.push_back(link2);

        //创建新的神经细胞
        SNeuronGene NewNeuron(hidden, NewNeuronID, NewDepth, NewWidth);

        //并且将它加入基因组
      m_vecNeurons.push_back(NewNeuron);
      }
      return;
  }
```

11.4.2.3 如何利用创新辅助设计一个有效的杂交操作

在本章一开始时就讲到，EANN 的突变操作经常会有许多麻烦。除了要保证突变不产生无效网络之外，还必须小心地避免竞争约定问题。EANN 网络的设计者们已经设法避开了这两个问题，这就是把创新标识号作为基因的历史标记来使用。因每一个创新都有一个惟一的标识号，基因可以按年代的先后来追踪，这意味着在不同基因组中的类似基因可以在进行杂交之前进行安排。图 11.20 就可以帮助我们弄清楚这一问题。

图中显示的基因是为生成每个表现型的链接基因。可以看出，表现型有很不相同的拓扑结构，但利用基因组中创新号的匹配，能很容易地从它们通过适当的基因交换来产生子代，如图 11.21 所示。

在基因组中间，那些不能匹配的基因称为脱落基因（disjoint gene），而在基因组末端不能匹配的基因称为过量基因（excess gene）。杂交的进行和以前讨论过的多点杂交有些相似。当操作进行时，随着每个基因组长度的不断下降，子代随机地继承了相匹配的上一代基因。脱落基因与多余基因仅从适应性最高的父母一代得到继承。

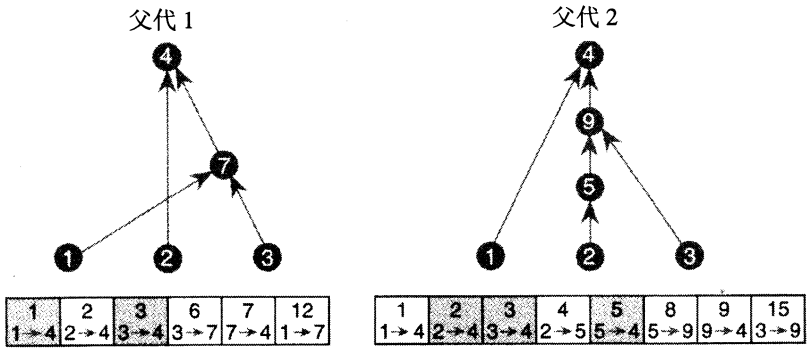


图 11.20　具有不同创新的两个表现型

图注：其中灰底色的基因是禁止使用的基因，每个基因顶部的数字是基因的创新标识。

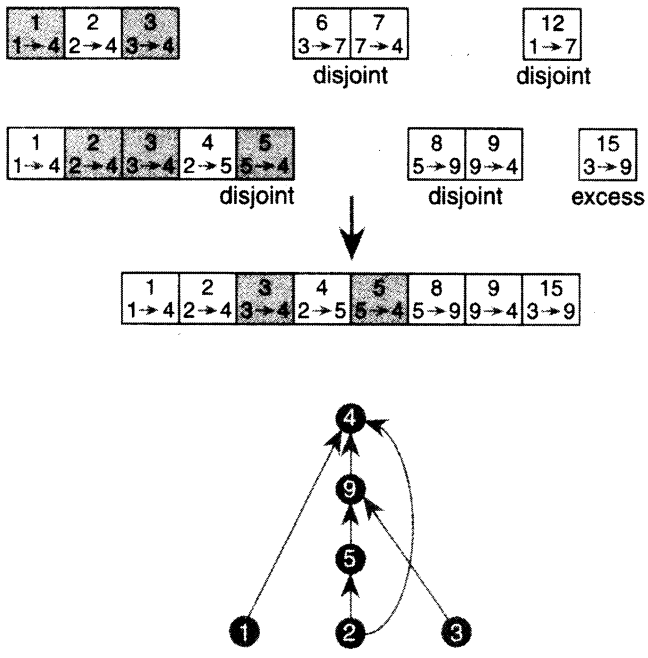


图 11.21　杂交操作进行过程

图注：标有 disjoint 的是脱落基因，　标为 excess 的是多余基因。

利用这种办法，NEAT 所生成的子代可以保证为有效基因，并能避开竞争约定问题。下面是有关杂交操作的完整代码。

```
CGenome Cga::Crossover(CGenome& mum, CGenome& dad)
{
  //首先计算用来产生 disjoint/excess 基因的基因组。这是适应性最好的
  //基因组。如果它们有相同的适应分，则选用较短者（因为希望网络保持尽可能小）
   parent_type best;

   if (mum.Fitness() == dad.Fitness())
```

```
    {
      //如果它们有相同的适应性又有相同的长度,则按随机方式来选择一个
      if (mum.NumGenes() == dad.NumGenes())
      {
        best = (parent_type)RandInt(0, 1);
      }
    else
      {
       if (mum.NumGenes() < dad.NumGenes())
         {
            best = MUM;
         }
       else
        {
           best = DAD;
        }
      }
}

else
{
    if (mum.Fitness() > dad.Fitness())
      {
        best = MUM;
      }
    else
      {
        best = DAD;
      }
  }
  //这些向量将保存子代的神经细胞和基因
  vector<SNeuronGene>    BabyNeurons;
  vector<SLinkGene>  BabyGenes;
  //用于存放所有被加入神经细胞的标识号的临时向量
  vector<int>            vecNeurons;
  //创建两个迭代变量,这样可以一步步通过每一个父代(父,母)基因,并把
  //两个迭代变量设置为每一父代的第一对基因
  vector<SLinkGene>::iterator curMum = mum.StartOfGenes();
  vector<SLinkGene>::iterator curDad = dad.StartOfGenes();
  //下面的变量用来保存在每一步中希望加入的基因的一个备份
  SLinkGene SelectedGene;

  //一步步通过每个父代(父母)基因,直到到达二者的结尾
  while (!( (curMum == mum.EndOfGenes())&&
  (curDad == dad.EndOfGenes()) ) )
  {
    //妈妈基因的结尾已经到达
```

```
  if ((curMum == mum.EndOfGenes())&&(curDad !=
                                              dad.EndOfGenes()))
   {
     //如果爸爸是最适应的
     if (best == DAD)
       {
          //加入爸爸基因
          SelectedGene = *curDad;
       }
     //考察爸爸的下一个基因
     ++curDad;
   }
  //已经到达爸爸基因的结尾
  else if( (curDad == dad.EndOfGenes()) &&
           (curMum != mum.EndOfGenes()) )
   {
     //如果妈妈最适应
     if (best == MUM)
      {
         //加入妈妈基因
         SelectedGene = *curMum;
      }
     //移动到妈妈的下一个基因
     ++curMum;
   }
 //如果妈妈的创新标识小于爸爸的创新数标识
 else if (CurMum->InnovationID < curDad->InnovationID)
 {
   //如果妈妈是最适应者,则加入妈妈基因
   if (best == MUM)
   {
     SelectedGene = *curMum;
   }
   //移动到妈妈的下一基因
   ++curMum;
 }
 //如果爸爸的创新号小于妈妈的创新号
 else if (curDad->InnovationID < curMum->InnovationID)
 {
   //如果爸爸是最适应者,则加入爸爸基因
   if (best = DAD)
   {
    SelectedGene = *curDad;
   }
   //移动至下一个爸爸基因
   ++curDad;
 }
```

```
//如果爸爸妈妈的创新号一样
else if (curDad->InnovationID == curMum->InnovationID)
{
  //爸爸、妈妈二者都取出基因
  if (RandFloat() < 0.5f)
  {
    SelectedGene = *curMum;
  }
  else
  {
    SelectedGene = *curDad;
  }
  //移动到每一个上辈的下一基因
  ++curMum;
  ++curDad;
}
//如果原来未曾加入所选择的基因,则现在将它加入
if (BabyGenes.size() == 0)
{
  BabyGenes.push_back(SelectedGene);
}
else
{
  if (BabyGenes[BabyGenes.size()-1].InnovationID !=
                                    SelectedGene.InnovationID)
  {
  BabyGenes.push_back(SelectedGene);
  }
}
//检查(vecNeuron)是否已经有所选基因 SelectedGene 所涉及(关联)的神经细胞?
//如果没有,就需要(在 vecNeurons 中)加入这些神经细胞
AddNeuronID(SelectedGene.FromNeuron, vecNeurons);
AddNeuronID(Selectedeene.ToNeuron, vecNeurons);
}//结束 while 循环
//创建所要的全部神经细胞,首先将它们排序
sort(vecNeurons.begin(), vecNeurons.end());
for (int i=0; i<vecNeurons.size(); i++)
{
 BabyNeurons.push_back(m_pInnovation->CreateNeuronFromID
                                              (vecNeurons[i]));
}
//最后创建基因组
CGenome babyGenome(m_iNextGenomeID++,
                   BabyNeurons,
                   BabyGenes,
                   mum.NumInputs(),
                   mum.NumOutputs());
```

```
        return babyGenome;
    }
```

11.4.3　物种形成

当把结构加入到一个基因组时，或者是加入一个新的链接，或者是加入一个新的神经细胞，这一过程就和一个新个体由一个蹩脚演员转变到有机会去演化并最终站入到群体之中的发展过程非常类似。但这意味着新个体在尚未有机会去演绎任何潜在有意义的行为之前，就有很高的几率被淘汰掉。这显然不是我们希望见到的，为此，人们必须考虑一些能够在演化的早期保护新的基因创新的方法。这就需要应用模拟物种形成（Speciation）。

物种形成，顾名思义，是由一个种群向几个不同的物种（species）的分离（separation）过程。至于确切地什么是物种的问题，这仍然是生物学家（和其他领域的科学家）们正在争论的事情，但一个流行定义是这样的：物种是有相似特性的一个群体，它们能成功地进行繁殖、产生健康并能再繁殖的后代，但与其他物种是繁殖隔离的。

在自然界，形成物种的一个普通机制就是通过地理环境的改变。设想有一种分布很广的动物群体，不妨称它们为“critter”（野生动物），这个群体因它们生存环境的差别，最后会形成一些地理性的变化。例如生长在高山上的 critter 和生长于平原的 critter 就可能会有不同。这样，当经历很长一段时期后，由于不同的自然选择作用和它们染色体中的基因变异，一个群体最终就会呈现多样化。生长在高山上的那个群体的皮毛就开始变厚以应对寒冷的气候，而长在另一些地方的群体可能对潜伏在那里的掠食动物产生出更强的适应能力。最后，两个地区的 critter 的相互差别可能变得很大，以至于它们一旦有机会再接触时，也不可能成功地交配和产生后代。这时就可以认为它们是两个不同的物种了。

NEAT 模拟了物种的形成机制，为任何新的拓扑改变提供了演化场所。采用这种方法，相似的个体只在它们自己内部进行竞争，而不与群体其余部分竞争。这样或多或少可以防止它们在未成熟就被淘汰的情况发生。为了实现这一想法，创建了一个可以用来保存所有物种的记录，称为 Cspecies。在每一代中，每一个个体都要对各个物种的第一个成员进行对照测试，计算出一个所谓的兼容性距离（compatibility distance）。如果某一个体与某一物种的兼容性距离在某个界限之内，则此个体就被加入到该物种里。如果此个体与当前已存在的所有物种都不能兼容，则就要创建一个新的物种，然后将此个体加入到该新物种中去。

11.4.3.1　兼容性测试

兼容性距离的计算就是测量不同个体的基因组之间的差别。这里再一次用到创新标识号。因为和突变操作所做的一样，可以对基因进行匹配，并对过量基因和脱落基因进行统计。统计所得的数目愈大，则差异也愈大。另外，连接的权重也要进行比较，然后把一个总的误差绝对值进行记录。因此这个有 3 个指标：

- 过量基因的数目（E）
- 脱落基因的数目（D）
- 连接权重的差值（W）

一旦这些数值确定下来，最终的兼容性距离利用下面的公式来计算：

$$c.dist = c_1 E / N + c_2 D / N + c_3 W$$

其中 N 是在较大的基因组中的基因数目（使大小规范化），c_1、c_2 和 c_3 是关系到最终估价的 3 个系数。如果这最终的数值在兼容性阈值之下，两个基因组就被认为属同一物种。如果它高于兼容性阈值，则基因组就被认为代表了不同的物种。这一用来计算兼容性距离的方法在程序中称为 CGenome::GetCompatibilityScore，它的形式如下：

```
double CGenome::GetCompatibilityScore (const CGenome &genome)
{
  //通过逐步减少每个基因组的长度来计算脱落基因、过量基因和匹配基因的数目
  double NumDisjoint = 0;
  double NumExcess = 0;
  double NumMatched = 0;
  //它记录了匹配的基因中权重差的总和
  double  WeightDifference = 0;
  //指向每个基因,当一步步减少基因组的长度时,它们是递增的
  int g1 = 0;
  int g2 = 0;
  whlle ( (g1 < m_vecLinks.size()-1) ||
          (g2 < genome.m_vecLinks.size()-1) )
  {
  //已经到达 genome1 的结尾处,但还没有到达 genome2 的结尾,所以应
  //递增过量的分数
  if (g1 == m_vecLinks.size()-1)
    {
       ++g2;
       ++NumExcess;
       continue;
    }
  //反之亦然
  if (g2 == genome.m_vecLinks.size()-1)
    {
       ++g1;
       ++NumExcess;
       continue;
  }

    //获得每一个基因此时的创新标识号
    int id1 = m_vecLinks[g1].InnovationID;
    int id2 = genome.m_vecLinks[g2].InnovationID;

    //如果创新号相同,则增加匹配分数
    if (id1 == id2)
     {
       ++g1;
```

```
            ++g2;
            ++NumMatched;
            //得到这两个基因之间的权重差
            WeightDifference += fabs(m_vecLinks[g1].dWeight -
                                     genome.m_vecLinks[g2].dWeight);
          }
          //如果创新号不同,则应增加脱落基因的分数
          if (id1 < id2)
          {
            ++NumDisjoint;
            ++g1;
          }
          if (id1 > id2)
          {
            ++NumDis:joint;
            ++g2;
          }
        }// while 结束
        //得到最长的基因组的长度
        int longest = genome.NumGenes();
        if (NumGenes() > longest)
        {
          longest = NumGenes();
        }
        //下面是应与最终分相乘的系数
        const double mDisjoint = 1;
        const double mExcess = 1;
        const double mMatched = 0.4;
        //最后计算总分
        double score = (mExcess * NumExcess / ( double)longest) +
                       (mDisjoint * NumDisjoint / (double)longest) +
        (mMatched * WeightDifference / NumMatched);
        return score;
      }
```

11.4.3.2　物种的类

一个个体一旦被指定为某一物种后，它就只能与和它属于同一物种的成员进行配对。但是，单独利用这种物种的形成机制还不能保护群体中新的创新。为了做到后一点，必须设法寻找一种能帮助年幼个体调整其适应性的方法，使得在一段合理长度的时期内，能有更多的不同的基因组保持为活跃状态。NEAT 中用来实现这一想法的技术称之为显式适应性共享（explicit fitness sharing）。

在第 5 章中已经介绍过了适应性共享，这就是，通过共享具有相似基因组的个体的适应性分数来保留差异。在 NEAT 中，适应性分数是由同一物种的成员所共享的。这实际上就是，在作任何选择前，每一个个体的分数都应除以物种的大小。这样，已经长大的物种的进一步生长就会受到它们的尺寸的惩罚而得到约束，而相反，小的物种在演化竞赛中则

受到奖励。

注意：在 NEAT 原始的实现中，设计者虽然也结合使用了这种“物种内部”的匹配，但将这种发生的概率设置得很低。尽管笔者从来没有观察到使用它时有任何值得注意的表现优化，但对于读者来说，当开始自己实现这一方法时，这仍然是值得一试的练习。

此外，年轻的物种在适应性共享计算中被赋予了更易加速增长的适应性分数。同样，老的物种则受到了惩罚。如果某一物种经历了几代（默认为 15 代）都不能显示一点改进，它就会被杀死而被淘汰。与此不同的一个例外是，如果某个物种包含了到目前为止所发现的功能最好的那个个体，则此物种就被允许继续生存下去。

为了帮助阐明刚提到的所有内容，下面就开始讲述计算所有适应性调整的方法。首先列出物种类 CSpecies 的定义：

```
class CSpecies
{
private:
    //为该物种的第一个成员保留一个本地备份（local copy）
     CGenome              m_Leader;
    //指向该物种内所有基因组的指针
     vector<CGenome*>     m_vecMembers;
    //物种需要的一个标识号
     int                  m_iSpeciesID;
    //物种至此找到的最高适应性分
    double                m_dBestFitness;
    //种类的平均适应性分数
    double                m_dAvFitness;
    //适应性分数改进之后的代的数目，必要时可用这一信息来杀死一个物种
    int                   m_iGensNoImprovement;
    //物种的年龄
    int                   m_iAge;
    //这一物种的下一代群体必须孵化出来多少后代
     double               m_dSpawnsRqd;
public:
    CSpecies(CGenome &FirstOrg, int SpeciesID);
    //这一种方法可以增加年轻基因组的适应性分数，惩罚年老基因组的适应性分数，
    //然后对该物种的所有成员实行适应性分数的共享
    void AdjustFitnesses();
    //把一个新个体加入到物种里
    void      AddMember(CGenome& new_org);
    void      Purge();
    //计算该物种需要孵化多少后代
    void      CalculateSpawnAmount();
    //在所选物种的最好的 CParams::dSurvivalRate 百分比中，随机选择
    //一个，来孵化出一个个体
    CGenome   Spawn();
    //-------------------------------------供访问用的各种方法
    CGenome   Leader()const{return m_Leader;}
```

```
    double    NumToSpawn()const{return m_dSpawnsRqd;}
    int       NumNembers()const{return m_vecMembers.slze();}
    int       GensNoImprovement()const
              {return m_iGensNoImprovement;}
    int       ID()const{return m_iSpeciesID;}
    double    SpeciesLeaderFitness()const
              {return m_Leader.Fitness()};
    double    BestFitness()const{return m_dBestFitness;}
    int       Age()const(return m_iAge;)
    //按照适应性的好坏对物种进行排序,最大的排在第一位
    friend bool operator<(const CSpecies &lhs,
                          const CSpecies &rhs )
     {
        return lhs.m_dBestFitness > rhs.m_dBestFitness;
     }
};
```

下面为调整适应性分数的方法：

```
void CSpecies::AdjustFitnesses()
{
  double total = 0;
  for (int gen=0; gen<m_vecMembers.size(); ++gen)
  {
    double fitness = m_vecNembers[gen]->Fitness();
    //如果物种是年轻的,则增加它的适应性分数
    if (m_iAge < CParams::iYoungBonusAgeThreshhold)
    {
       fitness *= CParams::dYoungFitnessBonus;
    }
    //惩罚年老的物种
    if (m_iAge > CParams::iOldAgeThreshold)
    {
       fitness *= CParams::dOldAgePenalty;
    }
    total += fitness;
    //把适应性共享并应用到调整后各个适当性分数
    double AdjustedFitness = fitness/m_vecMembers.size();
    m_vecMembers[gen] ->SetAdjFitness(AdjustedFitness);
  }
}
```

11.4.4　Cga 换时代方法

因为群体是划分为物种的，NEAT 的 Epoch 方法的代码和前面的各种 Epoch 方法的代码有所不同，而且还相当繁琐。Epoch 是 Cga 类的一部分，而 Cga 类是一个操纵着所有基因组、所有物种和所有创新的类。

下面介绍 epoch 方法。通过 epoch 方法的讨论，就能正确了解在演化过程的每个阶段将要做什么操作。

```
vector<CNeuralNet*> Cga::Epoch(const vector<double>
                               &FitnessScores)
{
  //首先进行检测,以保证有正确数量的适应性分数
  if (FitnessScores.size() !a m_vecGenomes.size())
  {
    MessageBox(NULL,"Cga::Epoch(scores/genomes mismatch) !",
               "Error", MB_OK);
  }
  ResetAndKill();
```

程序首先把前一代中所创建的任何表现型进行删除。然后再依次检查每一个物种，并把它的所有成员除了功能最好的一个外全部加以删除（这一留下来的个体在计算兼容性距离时将作为与之对照测试的标准基因组）。如果一个物种经历过了 Cparams::iNumGens AllowedNoImprovement 代之后仍然不能使适应性分数得到一些改善，则此物种就要被淘汰。

```
  //用上一次运行获得的适应性分数来更新基因组
  for (int gen=0; gen<m_vecGenomes.size(); ++gen)
  {
    m_vecGenomes[gen].SetFitness(FitnessScores[gen]);
  }
  //对基因组排序,并为最好的表演者作一记录
  SortAndRecord();
  //把群体分成具有相似拓扑的各种物种,调整它们的适应性分数,并计算孵化级别
  SpeciateAndCalculateSpawnLevels ();
```

函数 SpeciateAndCalculateSpawnLevels()所做的工作由计算每个基因组相对于每一现存物种的代表性基因组的兼容性距离开始。如果此兼容性距离的数值不超过某个允许范围，则此个体就被归并到这一个物种中。如果没有找到任何一个相匹配的物种，则就应创建一个新的物种，并把基因组加入这一新的物种中。

当所有的基因组都已被分配到物种后，SpeciateAndCalculateSpawnLevels()函数就要调用另一个函数 AdjustSpeciesFitnesses 去调整和共享适应性分数，如前面已经讲过的那样。

然后，SpeciateAndCalculateSpawnLevels()用来计算每一个个体在新一代中预期要孵化多少子代。这是一个用浮点数表示的值，它是用整个群体调整后的平均适应性分数去除以每个基因组的调整后的适应性分数得到的。例如，如果一个基因组调整后的适应性分数为 4.4，而群体的平均分是 8.0，则该基因组应孵化 0.55（55%）的子代。当然，对于一个单个的生物机体来说，不可能要求它去孵化出孵化量为一个分数的子代，但是对于整个的一个物种，可以把它的所有成员的个体孵化量全部累加起来，计算出物种的总体孵化量。表 11.3 可以帮助你弄清楚你对这一过程可能产生的任何混淆。它显示了包含 20 个个体的小群体的

典型孵化值。现在 Epoch 函数可以通过对每个物种的循环来孵化所要求的子代　总数。

表 11.3　不同物种的孵化值总数

物种 0			
基因组 ID	适 应 性 值	适应性调整	孵　化　值
88	100	14.44	1.80296
103	99	14.3	1.78493
94	99	14.3	1.78493
61	92	13.28	1.65873
106	37	5.344	0.667096
106	34	4.911	0.613007
107	32	4.622	0.576948
105	11	1.588	0.198326
104	7	1.011	0.126207
本物种孵化的后代总数（增长倍数）：9.21314			
物种 1			
基因组 ID	适 应 性 值	适应性调整	孵　化　值
112	43	7.980	0.99678
110	43	7.985	0.99678
116	42	7.8	0.973599
68	41	7.614	0.950419
111	37	6.871	0.857695
115	37	6.871	0.857695
113	17	3.157	0.394076
本物种孵化的后代总数（增长倍数）：6.02704			
物种 2			
基因组 ID	适 应 性 值	适应性调整	孵　化　值
20	59	25.56	3.19124
100	14	6.066	0.757244
116	9	3.9	0.4868
本物种孵化的后代总数（增长倍数）：4.43529			

下面继续讨论 epoch 方法。

```
//这将用来保存基因组的新群体
vector<CGenome> NewPop;
//从每一个物种产生子代。待孵化的子代数目是一个双精度实数,需要将它转换成
//为一个整数
int NumSpawnedSoFar = 0;
CGenome baby;
//通过对每个物种的循环,选择要配对杂交和突变的子代
for (int spc=0; spc<m_vecSpecles.size(); ++spc)
{
```

```
//从每个物种得到的孵化总数是一个浮点数,需要四舍五入化为整数,
//而这有可能导致孵化总数的溢出。本语句确保不会出现这种情况
if (NumSpawnedSoFar < CParams::INumSweepers)
{
  //这是该物种要求孵化的下一代个体总数。Rounded 把 double 型浮点数改大或
  //改小化成整数
  int NumToSpawn = Rounded(m_vecSpecies[spc].NumToSpawn());
  bool bChosenBestyet = false;
  while (NumToSpawn--)
  {
  //首先从该物种中找出表现最好的基因组,并将它不作任何变异地转移到新群体
  //从而为每个物种提供了精英
  if (!bChosenBestYet)
    {
      baby = m_vecSpecies[spc].Leader();
      bChosenBestYet = true;
    }
else
{
     //如果本物种仅包含一个个体,则只能执行突变操作
     if (m_vecSpecies[spc].NumMembers() == 1)
     {
      //孵化一个后代
      baby = m_vecSpecies[spc].Spawn();
     }
     //如果大于 1,则可以使用杂交操作
     else
     {
      //孵化 1
      CGenome g1 = m_vecSpecies[spc].Spawn();
      if (RandFloat() < CParams::dCrossoverRate)
        {
       //孵化 2,保证它不是 g1
       CGenome g2 = m_vecSpecies[spc].Spawn();
       // 寻找一个不同基因组时的尝试次数
       int NumAttempts = 5;
       while ( (g1.ID() == g2.ID()) && (NumAttempts--) )
       {
           g2 = m_vecSpecies[spc].Spawn();
       }
       if (g1.ID() != g2.ID())
       {
          baby = Crossover(g1, g2);
       }
```

由于一个物种中的个体数目可能很小，同时也因为只有性能最好的 20%（默认值）被保留成为父代，这使它有时不能（或速度很慢）找到用于匹配的第二个基因组。下面显示

的代码在寻找不同基因组时将试验 5 次，如果不成功就放弃寻找。

```
        }
      else
        {
          baby = g1;
        }
      }
    ++m_iNextGenomeID;
    baby.SetID(m_iNextGenomeID);

    //已存在一个孵化出来的子代,对它进行突变。首先应考虑
    //加入一个神经细胞的几率
     if (baby.NumNeurons() < CParams::iMaxPermittedNeurons)
     {
        baby.AddNeuron( CParams::dChanceAddNode,
                        *m_pInnovation,
                        CParams::iNumTrysToFindOldLink);
     }
     //加入链接的几率
     baby.AddLink(CParams::dChanceAddLink,
                  CParams::dChanceAddRecurrentLink,
                  *m_pInnovation,
                  CParams::iNumTrysToFindLoopedLink,
                  CParams::iNumAddLinkAttempts);
     //对权重实行突变
     baby.NutateWeights(CParams::dMutationRate,
                        CParams::dProbabilityWeightReplaced,
                        CParams::dMaxWeightPerturbation);
    //对激励响应实行突变
baby.MutateActivationResponse(CParams::dActivationMutationRate,
                        CParams::dMaxActivationPerturbation);
    }
    //根据创新号对新生基因排序
    baby.SortGenes();
    //将新基因加入到新群体
    NewPop.push_back(baby);
    ++NumSpawnedSoFar;
    if (NumSpawnedSoFar == CParams::iNumSweepers)
      {
        NumToSpawn = 0;
      }
    }//结束 while 循环
    }//结束 if 语句
    }//下一个物种
//如果这时因舍入误差而使所有物种孵化总量加起来出现下溢,并使子代总数小于群
//体的规模,则必须创建附加的子代并加入到新群体。这可以应用锦标赛方式,从群
//体的所有个体中选择得到
```

```
if (NumSpawnedSoFar < CParams::iNumSweepers)
{
    //计算要求增加的子代数目
    int Rqd = CParams::iNumSweepers - NumSpawnedSoFar;

    //捕捉它们
    while (Rqd--)
    {
      NewPop.push_back(TournamentSelection(m_iPopSize/5));
    }
  }
//用新群体代替当前群体
m_vecGenomes = NewPop;
//创建新的表现型
vector<CNeuralNet* > new_phenotypes;
for (gen=0; gen<m_vecGenomes.size(); ++gen)
  {
    //计算最大网络深度
    int depth = CalculateNetDepth(m_vecGenomes[gen]);
    CNeuralNet* phenotype = m_vecGenomes[gen].CreatePhenotype(depth);
    new_phenotypes.push_back(phenotype);
  }
  //代计数器加 1
  ++m_iGeneration;
  return new_phenotypes;
}
```

11.4.5 将基因组转变为表现型

有关物种演化的所有内容都已经作了介绍，但还需要了解如何把一个基因组转变成一个表现型。表现型的结构和基因组结构不同，它们使用了不同的神经细胞和链接。表现型的结构可以在文件 phenotype.h 中找到，其形式如下。

11.4.5.1 链接的结构

链接的结构很简单。它就是几个指针，分别用来指向它所链接的两个神经细胞和指向链接的权重。代码中的布尔变量 bRecurrent 是神经网络类 CNeuralNet 中绘图用来帮助在一窗口中绘制网络的。

```
struct SLink
{
  //指向与本 link 相链接的两个神经细胞的指针
  CNeuron* pIn;
  CNeuron* pOut;
  //链接的权重
  double dWeight;
  //这一链接是否为反复出现的链接
```

```
  bool bRecurrent;
  SLink (double dW, CNeuron* pIn, Cneuron* pOut, bool bRec):dHeight(dW),
                                                            pIn(pIn),
                                                            pOut(pOut),
                                                            bRecurrent(bRec)
  {}
};
```

11.4.5.2　神经细胞的结构

由 SNeuron 定义的神经细胞与比起它的小兄弟 SNeuronGene 相比包含了更多的信息。它另外包含了所有输入×权重形式的乘积之总和（这一值是在激励函数完成之后的值，它来自这一神经细胞的输出）和两个 std::vectors 向量（一个用来存放神经细胞内的链接，另一个用来存放神经细胞外部的链接）。

```
struct Sneuron
{
  //进入该神经细胞所有的链接
  vector<SLink> vecLinksIn;

  //离开它的链接
  vector<SLink> vecLinksOut;

  //权重×输入的乘积的总和
  double        dSumActivation;

  //由这一个神经细胞产生的输出
  double        dOutput;
  //这一神经细胞的类型
  neuron_type   NeuronType;

  //它的标识号
  int           iNeuronID;
  //设置 S 形的函数的弯曲率
  double    dActivationResponse;

  //用于表现型的可视化制作
  int           iPosX, iPosY;
  double    dSplitY, dSplitX;

  //-- 构造函数
  SNeuron(neuron_type type,
          Int          id,
          Double       y,
          double       x,
          double      ActResponse):NeuronType(type),
                                   iNeuronID(id),
                                   dSumActivation(0),
```

```
                                        dOutput(0),
                                        iPosX(0),
                                        iPosY(0),
                                        dSplitY(y),
                                        dSplitX(x),
                                dActivationResponse(ActResponse)
  {}
};
```

11.4.5.3 把一切组合在一起

在程序中，实际为一个表现型创建所有要求的 SLinks 和 SNeurons 两种结构的方法是 CGenome::CreatePhenotype。该方法通过对基因组的循环创建了所有需要的神经细胞和要求指向这些神经细胞的所有链接，然后再创建 CNeuralNet 类的一个实例。分析下列的代码后，将开始讨论 CNeuralNet 这个类。

```
CNeuralNet* CGenome::CreatePhenotype(int depth)
{
//首先应确保删除该基因组原来存在的表现型
DeletePhenotype();

//用于保存表现型所要求的所有神经细胞
vector<SNeuron*> vecNeurons;

//创建所有要求的神经细胞
for (int i=0; i<m_vecNeurons.size(); i++)
  {
  SNeuron* pNeuron =
           new SNeuron(m_vecNeurons[i].NeuronType,
           m_vecNeurons[i].iID,
           m_vecNeurons[i].dSplitY,
           m_vecNeurons[i].dSplitX,
           m_vecNeurons[i].dActivationResponse);
  vecNeurons.push_back(pNeuron);
  }

//再创建链接
for (int cGene=0; cGene<m_vecLinks.size(); ++cGene)
{
//在链接创建之前,要保证链接基因已被打开 (enabled)
if (m_vecLinks[cGene].bEnabled)
   {
    //产生指向有关的各个神经细胞的指针
    int element = GetElementPos(m_vecLinks[cGene].FromNeuron);
    SNeuron* FromNeuron = vecNeurons[element];
    element  = GetElementPos(m_vecLinks[cGene].ToNeuron);
    SNeuron* ToNeuron  = vecNeurons[element];
```

```
      //在这两个神经细胞之间创建一个链接,并为存入的基因分配权重
      SLink  tmpLink(m_vecLinks[cGene].dWeight,
                     FromNeuron,
                     ToNeuron,
                     m_vecLinks[cGene].bRecurrent);

      //把新的链接加入到神经细胞
      FromNeuron->vecLinksOut.push_back(tmpLink);
      ToNeuron->vecLinksIn.push_back(tmpLink);
    }
  }

  //每一个神经细胞都已经包含了所有的连接性信息,然后利用它们创建一个
  //神经网络
  m_pPhenotype = new CNeuralNet(vecNeurons, depth);
  return m_pPhenotype;
}
```

11.4.5.4　神经网络类

神经网络类比较简单，它包含一个组成该网络的所有神经细胞的向量 std::vector、一个用于更新该网络并取回其输出的方法，另外还有一个在用户指定窗口中画出代表网络结构的图形的方法。成员变量 m-iDepth 代表网络的深度，这是根据神经细胞基因的 splitY 值计算出来的。下面将会讲述如何使用这一个值。枚举类型 run_type 特别重要，因为它可以允许用户选择网络采用怎样的更新方式。下面给出这个类的定义并进行详细的介绍。

```
class CNeuralNet
{
private:
    vector<SNeuron*> m_vecpNeurons;

  //网络的深度
  int m_iDepth;

public:

  CNeuralNet(vector<SNeuron*> neurons,
             int              depth);
  ~CNeuralNet();

  //更新网络时你需要从 snapshot 与 active 两个参数类型中选择一个。如果
  //选择 snapshot（快照方式）,则网络深度值用来控制从输入开始对整个网络刷
  //新。如果选择了 active（激活方式）,则网络在每一个时间步骤（time-step）
  //中获得更新
  enum run_type (snapshot, active);

  //为本时钟周期（clock cycle）更新网络
```

```
  vector<double> Update(const vector<double> &inputs,
                        const run_type type);

  //在用户指定的一个窗口中绘制网络的图形
  void   DrawNet(HDC &surface,
                 int cxLeft,
                 int cxRight,
                 int cyTop,
                 int cyBot);
};
```

直到现在为止，所见到过的所有网络都是从它的输入开始，一层一层地向前执行(run)，通过整个网络直到产生输出为止。但对于 NEAT，一个网络可以有任意的拓扑结构，它的 link 可以向前链接，也可以向后链接，甚至还允许自己连向自己。这就使它几乎不可能采用任何基于层的更新方法，因为实际上根本就不存在任何的层。由于这一缘故，NEAT 的更新方法改用了下面的两种模式：

1．Active（激活）模式: 当使用 Active 更新模式时，每个神经细胞都要把它所有进入神经细胞在前一时间步骤（time-step）中计算得到的激励值合计在一起。这意味着，激励值的计算不同于通常人工神经网络那样在每一时间步骤中对整个网络进行刷新，而只是从一个神经细胞进入到它的下一个神经细胞。为了得到和基于层的方法的相同结果，这一过程必须多次重复，重复的次数应与网络深度相同，这样才能通过整个网络来刷新所有神经细胞的激励值。如果是动态地使用网络，则此模式是适宜采用的（例如用于控制扫雷机）。

2．snapshot（快照）模式：如果需要 NEAT 的更新函数表现得与通常神经网络的更新函数一样，就必须保证从输入神经细胞开始直到输出神经细胞的所有神经细胞的激励都加以刷新。为便于这样，更新函数通过所有神经细胞的循环次数应和网络生成输出之前的深度相同。这就是为什么计算那些 splitY 值有如此重要的原因。如果需要用一个训练集训练 NEAT 网络（如第 9 章“有监督的训练方法”中用来进行鼠标手势识别的程序那样），那么就可以使用这一更新类型。

下面为更新网络所用的代码 CNeuralNet::Update，它有助于弄清这一过程。

```
vector<double> CNeuralNet::Update(const vector<double> &inputs,
                                  const run_type        type)
{
  //创建一个用来存放 outputs 的向量
  vector<double> outputs;
  //如果模式为 snapshot（快照模式），则要求所有的神经细胞被重复通过
  //和网络深度一样多的次数。如果模式为 active（激活模式），则此方法只要
  //经过一次迭代就可以返回一个输出
  int FlushCount = 0;
  if (type == snapshot)
     {
       FlushCount = m_iDepth;
     }
  else
```

```
    {
      FlushCount = 1;
    }

  //对网络重复循环 FlushCount 次
  for (int i=0; i<FlushCount; ++1)
    {
    //清除输出向量
    outputs.clear();

    //这是当前神经细胞的一个下标
    int cNeuron = 0;

    //首先把“input”神经细胞的输出设置成和函数中传入的 inputs 值相等
    while (m_vecpNeurons[cNeuron]->NeuronType == input)
     {
        m_vecpNeurons[cNeuron]->dOutput = inputs[cNeuron];
        ++cNeuron;
     }

    //将偏移的输出设置为 1
    m_vecpNeurons[cNeuron++]->dOutput = 1;

    //然后用每次改变一个神经细胞的办法来遍历整个网络
    while (cNeuron < m_vecpNeurons.size())
    {
    //这个 sum 用来保存所有输入×权重的乘积的总和
    double sum = 0;

    //通过对进入该神经细胞的所有链接的循环,将该神经细胞各输入值加在一起
    for(intlnk=0;lnk<m_vecpNeurons[cNeuron]->vecLinksIn.size();++lnk)
    {
    //得到第 lnk 个链接的权重
    doubleWeight = m_vecpNeurons[cNeuron]->vecLinksIn[lnk].dWeight;

    //从该链接的进入端神经细胞得到输出
    double NeuronOutput =
          m_vecpNeurons[cNeuron]->vecLinksIn[lnk].pIn->dOutput;

    //将此输出加入总和 sum 中
    sum += Weight * NeuronOutput;
    }

    //现在让总和输入激励函数,并把其结果赋给这一神经细胞的输出
    m_vecpNeurons[cNeuron]->dOutput =
      Sigmoid(sum, m_vecpNeurons[cNeuron]->dActivationResponse);
```

```
      if (m_vecpNeurons[cNeuron]->NeuronType == output)
        {
          //加入到输出
          outputs.push_back(m_vecpNeurons[cNeuron]->dOutput);
        }
        //下一个神经细胞
        ++cNeuron;
      }
    }//进入通过网络的下一次迭代
    //如果执行了这种类型的更新,网络输出需要进行复位(reset),否则由它建立的
    //网络可能会和训练数据输入的顺序有关
    if (type == snapshot)
     {
       for (int n=0; n<m_vecpNeurons.size(); ++n)
        {
          m_vecpNeurons[n]->dOutput = 0;
        }
     }
    //返回输出
    return outputs;
  }
```

应注意的是，如果需要采用快照式更新，则网络的输出在函数返回前必须进行复位。这是为了防止对训练数据的输入顺序的依赖性(训练数据通常是一连串地依次送入网络的，如果随机送入网络将会明显减慢学习的速度)。

假设需要送入的训练数据集由一个圆周上的许多点组成。如果网络预先没有复位，则当 NEAT 加入新的链接时可能会利用上一次更新后所保存下来的数据而形成循环形式的链接。当然，如果想把输出再反馈影射到输入也可以，但通常要求的网络不会是这样的形式。

11.4.6 运行 Demo 程序

为了实际演示 NEAT 程序，已把第 8 章“为你的 Bot 提供知觉”中的扫雷机代码外挂了（plugged in）进去，读者可以自己编译源程序再来运行，也可以直接运行执行程序 Sweepers.exe。源程序和可执行程序均能在光盘相关的文件夹中找到。

和以前一样，F 键用来加快演化过程，R 键用来重新开始（复位），按下数字键 1～4 可以显示扫雷机的“足迹”。

与第 8 章不同，这一次建立了一个附加的窗口，在其中画出了被创建出来的 4 个最好的扫雷机的表现型，如图 11.22 所示。

激活的前向连接用灰线画出，抑制的前向连接用黄线画出。激活的连接用红线画出，抑制的连接用蓝线画出。从偏移神经细胞来的任何连接显示成绿色。连接线的粗细指示了权重的大小。

表 11.4 列出了本工程的默认设置值。

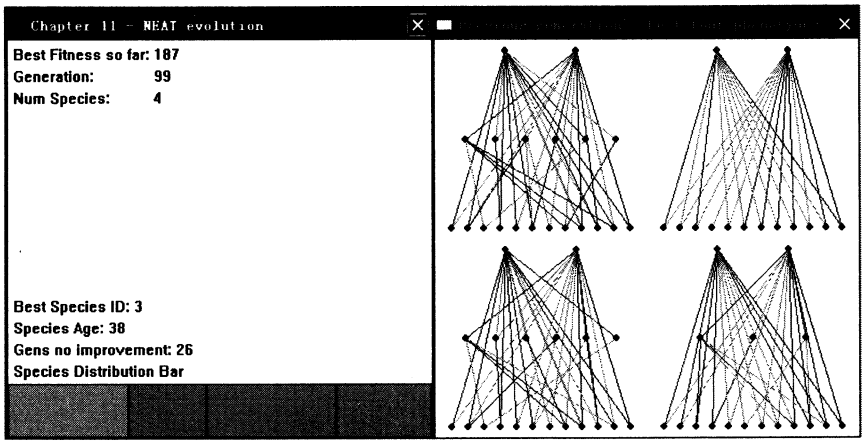


图 11.22　NEAT 扫雷机在行动

表 11.4　NEAT 扫雷机工程的默认设置

扫雷机参数	
参 数 名 称	**默认设置值**
触觉器数目	5
触觉器探测距离（单位:pixel）	25
扫雷器数目	50
最大产出（turn）率	0.2
大小（Scale）	5
演化用参数	
参 数 名 称	**默认设置值**
每 1 代（epoch）的“滴答”（tick）数	2000
加入 1 个链接（link）的几率	0.007
加入 1 个节点（node）的几率	0.003
加入 1 个链接的几率	0.005
杂交（crossover）率	0.7
权重突变率	0.1
最大突变干扰率	0.5
权重被置换概率	0.1
激励响应突变率	0.1
物种相容性阈值	0.26
物种老龄期阈值	50
物种老年惩罚	0.7
物种青春期阈值	10
年轻物种奖励	1.3

11.5 总　　结

本章的内容较深，在此过程中介绍许多内容。为了帮助理解，在本章所介绍的 NEAT I 程序已经作过了简化，可以学习一下 Ken Stanley 和 Risto Miikkulainen 的原始程序，从那里能够获得有关 NEAT 的更完整的理解。读者可以在下面的网站上找到有关 NEAT 的源码和其他文章。

http://www.cs.utexas.edu/users/kstanley/

11.6 练　　习

1. 在原有代码工程中加进一段代码，使物种的数目可以由用户规定其大小范围。
2. 设计一些不同的突变操作。
3. 在原有代码工程中增加种内配对杂交。
4. 为本章前面描述的其他的一种网络拓扑演化方法进行编码。

附录A　WEB资源

为了获得人工智能的相关信息，Internet 网无疑是最大的一个资源。下面列出的是最好的一些网站。如果什么地方有困难，就可以到这些网站去看看，几乎总有一个或几个能帮你忙。

A1　相关的URL地址

www.gameai.com

这是一个专门奉献给游戏人工智能的大型网站，由永远受欢迎的“探索人”Steve Woodcock 在经营维护。对于任何与人工智能（AI）和人工生命（Alife）游戏有关的问题，这是一个极恐怖的讨论起点。

www.ai-depot.com

另一个巨大资源，一个大场所。有关 AI 的任何新鲜事物它都会及时登载，它同时还包含了许多和 AI 各个方面有关的有用教程。

www.generation5.org

这一网站的选材不严格限制和游戏相关，但它包含了丰富的教程和有用信息。

www.citeseer.com

The Citeseer Scientific Literature Digital Library（Citeseer 科学文献数字库）是一个令人吃惊的文档源。如果需要寻找一篇论文，这里就是你开始寻找的最好地方。

www.gamedev.net

这一 Web 站点包含许多已经存档的文章和教程。它在 Internet 上主持的一个 AI 论坛是最好的 AI 论坛之一。

www.ai-junkie.com

这是笔者个人的小网站。这一网站早先称为“Stimulate”（“刺激”）网站，但感到它需要一个新的名字和一种新的面貌。如果读者对本书的技术方面的叙述有任何问题，就可放心地在其中的论坛里提问。

www.google.com

由于许多人似乎仍然不知道怎样使用这一网站，所以在这里必须包含这个搜索引擎。笔者在 Internet 网上探索的每一个事情几乎总是从这里开始链接。如果你还没用使用它，那么就开始用它吧。

A2 新 闻 组

Usenet（网上新闻组）往往被游戏编程者所忽视，但它可能是一个极有价值的信息来源，读者可以从中得到帮助，最重要的是可以得到灵感。那么下面所列的新闻组是非常有价值的：

comp.ai.neural-nets
comp.ai.genetic
comp.ai.games
comp.ai.alife

附录 B　参考书目及推荐读物

B1　技术书[①]

Neural Networks for Pattern Recognition，《用于模式识别的神经网络》

Christopher Bishop

这是神经网络当前的一本“圣经”。

An Introduction to Neural Networks，《神经网络引论》

Kevin Gurney

这是神经网络的一个伟大的简介。Kevin 在书中介绍了一个目前最流行的网络结构。他尽了最大努力来避免数学，但仍然需要知道些微积分才能读这本书。

Neural Computing，《神经计算》

R Beale & TJackson

包含了一些有趣的页面。

Genetic Algorithms in Search, Optimization and Machine Learning，《用于搜索、最优化和机器学习中的遗传算法》

David E.Goldberg

Nuff 称它是遗传算法的圣经。

An Introduction to Genetic Algorithms，《遗传算法引论》

Melanie Mitchell

一本写的很好并且非常流行的遗传算法的入门介绍。如果你喜欢遗传算法理论方面的循序前进式的介绍，这是一本理想读物。

The Natural History of the Mind，《心灵的自然历史》

Gordon Rattray Taylor

建立在大脑和思维的生物学基础上的一本巨著。

The Blind Watchmaker，《瞎子钟表匠》

Richard Dawkins

本书以及作者的另一本书，The Selffish Gene，是进化机制的两本很有特色的介绍性

[①] 译者注：为便于阅读，译者把原文书名译成中文，但这并不意味原书已有中文译本，另外译名也仅作参考。

读物。

Programming Windows 5th Edition，《Windows 编程，第 5 版》
Charles Petzold
Windows 编程的圣经。

The C++ Standard Library，《C++ 标准库》
Nocolai M Josuttis
STL（标准模版库）的圣经。这是一本极好的书。Josuttis 把一枯燥的主题写得使人着迷。

The C++ Programming Language，《C++编程语言》
Bjarne Stroustrup
C++的圣经。

B2 论　　文

Evolution of neural network architectures by a hierarchical grammmer-based genetic system
Christian Jacob and Jan Rehder

Genetic Encoding Strategies for Neural Networks
Philipp Koehn

Conbining Genetic Algorithms and Neural Networks: The Encoding Problem
Philipp Koehn

Evolving Artificial Neural Networks
Xin Yao

Evolving Neural Networks through Augmenting Topologies
Kenneth O. Stanley and Risto Miikkulainen

Evolutionary Algorithms for Neural Network Design and Training
Jurgen Branke

'Genotypes' for Neural Networks
Stefano Nolfi & Domenico Parisi

Niching Methods for Genetic Algorithms
Samir W.Mahfoud

Online Interactive Neuro-Evolution
Adrian Agogino, Kenneth Stanley & Risto Miikkulainen

B3　能激发思想的书

Gödel Esher Bach, An Eternal Golden Bruid,
《Gödel Escher Bach，一根永恒的金色发辫》
Douglas Hofstadter

The Minds I,《头脑一号》
Douglas Hofstadter

Metamagical Themas,《元逻辑命题》
Douglas Hofstadter

由 Douglas Hofstadter 编写的任何一本书都很精彩。他（在其他的主题中）采用极为引人入胜的和发人深思的方法来探索心灵、知觉和人工智能。*Gödel Esher Bach, An Eternal Golden Braid* 这本书刚出了再版。

Artificial Life,《人工生命》
Stephen Levy

如果只买一本人工生命方面的书，这一本很值得读。Levy 是一位优秀作家，本书中虽然没有非常大的深度，但用了极为轻松的风格覆盖了许多基础的东西。

Creation（Life and How to Make It）,《造物者：生命和生命的制作》
Steve Grand

写的有点含糊，并且有时有点散乱，但仍是一本值得一看的书。Grand 就是 Creatures 的编程者，本书解释了 Norms（他把他游戏中的 Creatures 称为 Norms）的原理，另外还解释了 Steve 的有关生命和意识的思想。

Emergence（from Chaos to Order）,《意外事件：从混屯到秩序》
John H Holland

一本不错的书，有些章节很有趣。

Darwin amongst the Machines，《站在机器中的达尔文》
George Dyson

这和 Levy 的书有点类似，但更多地聚焦在早期的计算机和人工生命的历史，也是一本巨著。

The Emperor's New Mind，《皇帝的新脑》
Roger Penrose

本书收集了许多理由解释作者为什么相信机器决不会有意识。可以对他的结论表示不同意，但它仍是一本很有趣的读物。[①]

B4 值得一读的科幻小说

下面是笔者经常阅读的著名科幻小说，供读者需要某种轻松读物时使用的。

The Skinner，《剥皮者》
Neal Asher

Gridlinked，《连着的网格》
Neil Asher

The Hyperion Series of Books，《亥伯龙系列丛书》
Dan Simmons

Altered Carbon，《另一种碳》
Richard Morgan

K-PAX I,II & III，《I,II,III 号 K-交换机》
Gene Brewer

最后还要加上 Iain M. Banks 写的所有的科幻小说。

[①] 译者注：本书已由许明贤、吴忠超二位译成中文，由湖南科技出版社 1996 年出版。

附录C　光盘中的内容

在随书附带的光盘上收集了每一个演示程序的源代码，为了使读者尽快地看到程序的效果，在光盘上同时也编写了预先已编译好的可执行程序。因每章都有它的自己的文件夹，因此寻找相关工程文件时不会遇到任何困难。

构建这些演示程序比较简单。首先，必须将文件复制到硬盘上。如果读者使用微软的Visual Studio来开发，那只要在相关的Workspace上单击即可。如果使用其他编辑器，则先要创建一新的win32工程，且务必把winmm.lib加入到程序设置中，然后在单击compile按钮前，从程序文件夹把相关的源文件和资源文件加入进去。

Colin McRae Rally 2[①]赛车游戏程序的一个演示版也收集在光盘中。这一游戏不但充满趣味，而且还是利用神经网络来控制计算机驱动的对手的。

下面就是该游戏幕后的人工智能专家 Jeff Hannan 在和第五代的[②]James Matthews 的一次会见中回答对方时所说的话：

问：从人工智能设计和可游戏性方面考虑，神经网络能够带来什么样的可适用性？神经网络能控制人工智能的一切方面吗？

答：最大的挑战显然就是如何实际地把一辆汽车成功地快速驾驶到轨道中。这一步一旦达到，那么就几乎总可以任意调整跑道线，把所有的性能都添加到驱动器上。神经网络能在新的路线上行驶，而无须进行任何新的训练。

神经网络的构造是以保持汽车在轨道线上行驶作为其简单目的的。它们可以有效地执行这个技能。至于一些高级驾驶功能，如追超一辆车子或从撞车后恢复正常行驶，我认为必须与这些核心作业分开。事实上我能作的是执行这些任务的相当简单的规则。

问：你认为，在现在或未来，哪些类型的游戏，最有可能渗入神经网络、遗传算法等“主流人工智能”技术？

答：神经网络和遗传算法都是强有力的技术，一般地说，它们可以用到任何合适的问题中，不光是人工智能。任何游戏类型都可以使用它们，不在一个困难问题中考虑它们的应用是没有道理的。当然，经验通常是需要的，它们可以帮助你找到解决方案，但不是把它放在一个盘子上端给你。

---

① 译者注：Colin McRae Rally 中译科林麦克雷拉力赛，目前已出 Colin McRae Rally 2005。

② 译者注：James Matthews 也是一位著名人工智能专家；第5代原来应指具有新一代体系结构的未来计算机，但在这里“第5代”（generation5）是指从事第5代计算机结构研究的专家们所组成的一个学术团体。上面引用的一段对话就是从这一机构的官方网站的一个网页上摘录下来的，它的完整的原文可参考：www.generation5.org/content/2001/hannan.asp。

从我个人使用神经网络的经验来看，我会说，神经网络在技能上特别高超。当人们执行一种技能时，这是一种不需要高级推理的自发式运动。大脑已经学到了一种功能，能自动响应环境状况来产生正确行为。由此我认为，运动类游戏是未来最有可能的候选者。

技 术 支 持

神经网络和遗传算法对于初学者来说都是很容易混淆的主题。与你有类似思想的人讨论一下你的想法往往会有帮助。你可以把你的问题在下面的论坛上张贴出来并参与讨论：

www.ai-junkie.com

本书包含的源码的任何更正可以在下面的网页中找到：

www.ai-junkie.com/updates

后　记

一个木工徒弟可能只想要一把榔头和一个锯子，而一个木匠师傅则需要使用许多精密工具。为了应付实际应用的复杂性，计算机编程同样需要各种各样的工具，而且只有实际操作这些工具，才能在他们的使用中逐步增进技巧。

Robert L. Kruse，《数据结构和程序设计》

到此，我们已经到达我所希望的一次有刺激性的并让人回味思索的长途旅行的终点。我希望您在读这本书时能够得到和我写这本书时一样多的趣味。

到现在，你应该已经充分了解怎样在你自己的工程中，“在适当的地方”，开始实施神经网络和遗传算法。我用了引号注明“在适当的地方”几个字，是因为我经常看到有人企图把神经网络作为灵丹妙药滥用在一个游戏中所有需要 AI 的地方。就是说，有一些热心人，因受到新发现知识的兴奋而一时激动起来，企图用神经网络来控制一个复杂游戏代理程序中的全部人工智能。他把网络设计成有好几打的输入和许许多多的输出，并期待一个吉祥日子它会像 Arnold Schwarzenegger 一样来完成所有的工作！

不幸的是，奇迹很少发生，而经常发现的是，同样是这些人，当程序迭代几十兆代后他们的 BOT 仍在圈内打转时，就从此不相信神经网络的功能而摇起头来。

一种最好的办法就是把遗传算法和神经网络当作你的 AI 工具箱中的另一种工具去看待。随着你使用它们的经验和信心的增长，你将会看到有更多的领域，在其中，这些工具中的一个或两个可以被利用上去并产生良好效果。这可以是一种非常明显的用法，如 Colin McRae Rally 2 所做的、利用一个前馈网络应用程序来控制汽车的人工智能；也可能是一种十分微妙的用法，如在黑白棋中用单个神经细胞来模仿人所希望的东西。你也可以在游戏开发阶段发现它们的用途。现在已有许多开发人员利用遗传算法来牵引住他们的游戏代理人的性情。我甚至还听说过有些开发者，在他们的游戏环境中，让松散的神经网络控制的游戏代理人，来测试物理引擎。如果在你的游戏程序有任何缺点或漏洞，一个由遗传算法驱动的神经网络将是一寻找它们的极有价值的方法。

如果你读了本书有关技术后而编写出了可以感到骄傲的某种程序，我愿意听到从你发来的消息。能看到各种不同的人们在继续利用这些技术，是对我在这键盘上敲掉的时间的最大的奖励。因此，别害羞，在 fup@ai-junkie.com 上和我联系吧。

尤其重要的是，玩得开心！

Mat Buckland，2002 年 7 月

读者建议反馈表

1. 姓名_____ 2. 性别_____ 3. 年龄_____ 4. 电话_____
5. 单位_____ 6. 职务/职称_____
7. 通信地址_____ 邮编_____
8. 电子信箱_____ 单位网站_____
9. 您的文化程度: ☐中专以上 ☐大专 ☐本科 ☐研究生以上
10. 您所学专业: ☐通信电子 ☐计算机类 ☐机电控制 ☐数学类 ☐其他
11. 您所在行业: ☐商业网站 ☐硬件开发 ☐邮政银行 ☐软件企业 ☐系统集成 ☐服务行业 ☐科研院校 ☐政府机关 ☐网络通信 ☐制造业
12. 您的工作性质: ☐设计开发 ☐大学教学 ☐普通培训 ☐学生
13. 您使用计算机在: ☐办公室 ☐实验室 ☐网吧 ☐宿舍和家 ☐笔记本电脑
14. 您每季度买书在: ☐五十元内 ☐一百元内 ☐二百元内 ☐可以报销
15. 您购买本书在: ☐新华书店 ☐校园书店 ☐科技书店 ☐网站 ☐其他
16. 您使用编程语言: ☐C/C#/C++ ☐Java ☐Delphi ☐VB ☐其他
17. 您使用数据库为: ☐Oracle ☐DB2 ☐SQL Server ☐Sybase ☐其他
18. 您的开发平台为: ☐.Net ☐JavaOne ☐Websphere ☐Weblogic ☐其他
19. 您认为本书作者应该创作其他哪些书籍，如何写？

_____

_____

20. 您认为市面上类似书籍的特点有哪些？

_____

_____

21. 您对本书的建议和意见:

_____

_____

22. 您今后需要本类的哪些书籍？

_____

_____

表格填好后请寄:

有关计算机/电子/通信类等书籍投稿意向请按如下方式联系:

地址: 北京清华大学校内金地公司 邮编: 100084
电话: 010-62788951/62791977 转 219 传真: 010-62788903
信箱: xucq@tup.tsinghua.edu.cn

有关本书的建议和意见或邮购本丛书请按以下方式联系:

地址: 北京清华大学校内金地公司 邮编: 100084
电话: 010-62770384 传真: 010-62788903
公司网址: www.thjd.com.cn 公司电子信箱: thjd_support@hotmail.com