游戏之旅——

我的编程感悟



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

游戏之旅——我的编程感悟

云风 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书忠实地记录了作者十余年来对游戏编程的所思、所感、所悟。全书按照作者本人学习和实践的过程，带着读者从基础的计算机知识到高级的编程技术，从非常专业的汇编优化到非常实际的项目管理进行了一次游戏开发的全景探索。

本书不仅适合游戏开发者阅读，也会给所有的开发者和程序爱好者带来启示。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

游戏之旅——我的编程感悟 / 云风著. —北京：电子工业出版社，2005.12

ISBN 7-121-01609-5

Ⅰ. 游… Ⅱ. 云… Ⅲ. 游戏－应用程序－程序设计 Ⅳ. G899

中国版本图书馆 CIP 数据核字（2005）第 086284 号

责任编辑：朱沐红 zsh@phei.com.cn

印　　刷：北京天竺颖华印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　邮编　100036

经　　销：各地新华书店

开　　本：787×980　1/16　印张：25.75　字数：494 千字

印　　次：2005 年 12 月第 1 次印刷

印　　数：6000 册　　定价：46.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

姚壮宪 序

PREFACE

云风，这名字在我 2000 年刚从台湾到北京之初就已经听闻了，当时觉得这名字真“武侠”，挺特别的。当时他已经是中国游戏程序界的名人了，那时我很想招募到这位高人，只可惜欠缺缘分未能如愿。现在他有更大的成就，我也很替他高兴。

在中国，我最佩服的网游公司是网易，不仅是因为其游戏运营得成功，更重要的是其赖以成功的几套网游完全是自制的。而云风是这些网游主力程序员之一，他也是我在中国最佩服的游戏开发者。看了云风的研发历程，我觉得就是一部中国的游戏程序史，从最早的 Z80，6502，PC8088，286，386...DOS，保护模式，Assembler 到C++的整个发展轨迹。这本书可以说横跨了游戏程序的过去、现在和未来。书中传达的不仅是一些实用的技术经验，更是传达一种理念——虽然研发的环境随着时代而变，但研发的精神是不变的，那就是“在实践中积累”。

一些过去早期的技术手段，也许现在不一定派得上用场，但精神和原理是相通的。云风之所以有现在的成就，不是一朝一夕得来的，而是十余年的实践积累，从早期至今的每个阶段，面对各种当时的挑战，不断

地钻研。早期用一条条汇编指令跟运行速度斗争，64K，640K DOS 时代想尽办法跟内存空间斗争，和显卡斗争，到了网游时代，跟网速斗争，甚至跟外挂斗争。很辛苦，但是我觉得云风乐在其中，而且还能仔细地把这么长的过程中的这些细节记录下来并且写成书，实在非一般人可及，我就自叹不如。

像这样的书，在游戏界确实稀少，主要原因是，有空写书的往往不是在职的重要研发人员， 而真正的在职高手工作负担都很重。像云风这批中国第一代的游戏程序高手目前几乎都在各企业担任重要工作，所以这类由本地从业人员所写，且内容实际在游戏中实践的书自然就很少。

云风花一年多写此书，可想而知是异常辛苦的，身体健康也付出不少代价。但是，我认为此书对整个中国游戏行业有承先启后的效果，可提供千千万万想成为未来游戏程序员的人很好的启蒙，对在职的工作者也能得到许多良好的经验。云风得到的只是一些稿费，而整个中国民族游戏产业得到的将是一次知识推动。

孟岩 序

PREFACE

我不是一个游戏程序员，对游戏开发可以说是一窍不通。因此，当云风邀请我给这本书写几句评论的时候，我是有些惶恐的——我能胜任吗？在读过这本书的初稿之后，我心里有了几分底气，因为我确信这本书不仅仅适合游戏开发者，同样也适合所有的开发者和程序爱好者。不同的人能从这本书里得到不同的启发，我愿意与大家分享我读这本书的几点体会。

这本书是我所看过的第一本以叙事散文体写作的技术作品，因此面目亲切成了我对它的第一印象。从作者记忆闸门里涓涓流淌而出的故事，本身就成为这本书的主线，也成为了最好的自序。本土技术图书的行文风格，历来以冰冷刻板为特征而自成一体，以至于我经常猜想，这种文体是否也是我们的以层层过滤为特点的精英选拔机制的一部分？

近年来，本版技术图书在“亲和性”方面已经做了很大的努力，很多作者，尤其是来自开发一线的作者，做了很多改进的努力，比如在文章里设定一个开发场景，讲一个故事，采用第一人称夹叙夹议，等等。

而云风的这本书，依我的有限的阅读来看，在可读性上达到了当前本土技术图书的一个新高度。行文亲切自然，态度谦和自信，记叙、议论、说明、抒情数体并用，转折无痕，恰到好处。尤其是在技术细节的表达上，非常细腻，非常深入，既无避重就轻、故弄玄虚之怯，又无居高临下、盛气凌人之骄。读着令人如此惬意的技术文字，我仿佛回到了三年前与云风品茗畅谈的那个晚上，看到了摇曳的烛光，闻到了茶气的清香，似乎他就坐在对面向我兴致勃勃而又慢条斯理地叙述着他的故事和心得。一本好书应该既是好老师又是好朋友，在这方面，我相信云风的这本书已经做了一个精彩的诠释。

当然，我并不是说技术书也应该按照文学的要求斧凿一番，本质上我们这些“理工科思维”的怪物还是会依据一本书的技术水准来给予它基本的评价。跟机器打交道的程序员自然有冷血的一面，哪怕你锣鼓喧天，如果没有真功夫，也休想赢得半声喝彩。而以“真功夫”论，本书在编程技术和游戏开发专业技术两方面都表现得极为出色。全书基本上按照作者本人学习和实践的过程，带着读着从基础的计算机知识到高级的编程技术，从非常专业的汇编优化到非常实际的项目管理进行了一次游戏开发的全景探索。

在技术上，本书既相当深入又很好地把握了重点，显然在选材上花了很多心思。同时作者非常清楚书本的局限性，很多功夫下在了书外，为读者提供了很多资源和学习方法的指导。从体裁上看，本书不是教材，但是由于作者将自己的学习经历忠实地记录了下来，所以读者如果能够在阅读本书的同时，根据自己的实际情况多动手、多思考、多实践，则完全能够走出自己的“高手”之路。因此，对于有心人来说，这本书也可以是最好的教材。更重要的是，全书完完全全来自实践，又经过了多年的沉淀，与时代同步，与读者零距离，没有食洋不化的弊端，也没有隔靴搔痒的尴尬，在目前国内游戏编程类图书中显得卓尔不群。

云风是国内著名的游戏程序员，也是网易公司著名网游《大话西游》的主力开发者，他的技术是高超的，他的事业也是非常成功的。当然，他的传奇经历不用我多说，这本书本身就是他最好的自传。从编程技术上讲，他是毫无疑问应该被归为“天才”一类的人，过人的天资加上惊人的勤奋，使他很早就成为国内游戏开发技术的佼佼者。我对于游戏开

发技术并不了解，但是在C/C++开发上与他有过一些交流，深为他扎实的基本功、对系统的透彻理解、对程序的天然敏感和刻苦的钻研精神所折服。

特别让我感到钦佩的是，作为一个别人眼中的编程天才，他似乎很清楚地知道如何权衡地使用技术力量，在技术与实践中把握平衡。比如在C++一些具体技术的使用问题上，我走过很多弯路，总是在几个极端上不断地自我否定，只是在数次挫折之后才逐渐有所体会。而云风不仅有着“先天”的敏锐嗅觉，而且能够比我快得多地达到“知”的彼岸。他既勇于探索又能够保持头脑清醒，既能够创新又能客观地看待自己的创新，从而在总体上始终能够向着最有利于实践的方向前进。这一点是非常难能可贵的，甚至对大多数人来说，必须经过长时间的实践才能在一次次的成功与挫折中悟道。

C++名人 Alexandrescu 说，十几岁的少年天才到处都有，三十多岁的优秀设计师凤毛麟角，掌握一种力量是容易的，学会恰当地使用这种力量却难得多，这就是聪明与智慧之间的差别。文武之道，有张有弛，知道什么是该做的，什么是不该做的，这是人生最大的学问。之前我经常奇怪，云风还非常年轻，他程序思想中的那种老练的智慧是从何处得来的呢？读完这本书之后，我终于明白，还是那句话：“无他，唯手熟耳”。可以说，云风的成功之路，就是一条实践之路。路上的每一步，都是他一点一点走过来的，任何一个读过本书的读者，恐怕都会对云风丰富的经历印象深刻。他付出比常人多得多的努力去实践，自然收获比常人多得多的智慧和经验，对此，我所能有的只是钦佩。

因此在我看来，这本书对我以及很多人来说更重要的启示在于，它揭示了一个程序员的成功之道——实践。由于工作的原因，我接触到很多的优秀开发者，他们在技术上各有特长，但在个人事业上却状态迥异。作为一个近距离的旁观者，固然常为成功者的意气风发所感染，却也时不时为众多高手的作茧自缚而一声叹息。是什么导致这样的差别？我觉得就是对于成功之“道”的体认，或者按照我的总结，对于“真正的实践”的体认。

这种实践不是传统文人式的闭门苦读，不是精神贵族式的居高临下的点点画画，也不是市侩式的唯利是图的混迹，而是有理想、有方向、有规划、有选择、有实干的主动的思考和行动。云风在这本书里用他自己的经历为这种实践作了出色的范例，他在自己成长的

每一个阶段都充分利用了客观环境所能提供的条件，既深入地学习，积极地实践，同时也毫不犹豫地参与社会工作，还能总结经验和教训并迈出新的步伐。这才能使他做到既深入技术细节又能把握工作全貌，既实现了技术梦想又实现了现实的事业。读者如果善于领悟，应该能够见微知著，体会作者的成功之“道”。

老子说，具体的知识越学习就积累越多，而“道”则是越追求便越简单，大道至简，至简的大道才能应变万端（为学日益，为道日损，损之又损，以至于无为，无为无不为）。知识和技术是学不完的，只有得到实践这个“道”，才能实现“无不为”的理想与抱负。这就是我从云风这本书里得到的最大的收获，愿与各位共勉。

写书是件非常辛苦的事情，尤其对云风来说，更是件非常得不偿失的事，他满可以用这宝贵的时间和精力为自己创造更多的“价值”，然而他选择了写书，把更大的价值奉献给读者。面对这沉甸甸的作品，我确实感到，这是云风用心写的书。用心写的书，当然出色。

前　言

PREFACE

终于把整本书完成了，我几乎不敢相信能完成。陆陆续续地写了一年多，其中的辛苦恐怕只有写过技术书籍的朋友能够理解了。

很早以前就有朋友跟我提过，好好写本书，把自己理解的知识传授给他人，于己于人都是颇为有益的事情。写作技术书是很困难的事，它需要作者厚积薄发，把自己多年的知识浓缩归纳，才能用简洁的文字表达出来。而知识的传授绝对不同于讲故事，文字究竟能表达出多少作者自己已经理解的，我个人觉得是很难拿得准的。

先后有几个出版社的朋友来约稿，只是一直忙于项目开发，没有将其放在心上。直到《大话西游 II》和《梦幻西游》两款游戏相继开发完毕，并取得了市场上的成功，我的工作也相对闲了下来。这时，我终于可以动笔来写这本书了。

大家现在看到的这些内容，是我在一年多的时间里，断断续续整理出来的。看惯了国外严谨的技术书的读者，肯定会挑出许多毛病来。我几乎没有一章，把某个技术点由浅入深地阐述全面；也没有将游戏制作

的某个特定领域分章断节，细细道来；它绝对是依照我自己的偏好来选材，甚至，这本书不能算是一本游戏编程的专门书，我把重点放在了编程领域，而不是游戏。

我曾经对自己这本书的定位迷茫过。它看起来好像是一个大杂烩，整体的组织没有条理，细节的描述又不够彻底。终于有一天，我自己想明白了，得以心情舒畅地将之全部完稿。

首先，这里记录的诸多技术问题都曾经启发过我自己，那么它们一定也能启发他人，必然会和一些心有戚戚的读者引起共鸣。而另一方面，我不敢随意对游戏开发，甚至软件开发中哪些方面更重要一些下出论断。虽然我自己有自己的理解，但表现于具体的文章上，只好将自己觉得会非常重要的部分多写一些（因为自己觉得这些重要，所以研究得也多一些），希望读者读到那些枝节要点却没有点明的时候，能够理解我的用意。

其次，所有的基础知识都有前人留下的教导，和读者分享自己是怎样学到的足矣。授人以鱼，三餐之需；授人以渔，终生之用。大多数问题，如果把别的书中的章节生搬硬套过来，则是对我自己的精力和读者的时间的极大浪费。不以自己懂了什么就写出来给读者炫耀一番，是我后期删改书稿的重要原则。

终其整本书，虽然参考之书籍、网站、代码众多，但我自己都一直坚持一个原则：建立在充分理解的基础上，再用自己的语言表达出来。我相信只有这样，读者才能够看到一个从独立思考角度获得的知识体系，凌乱中不失一缕联系。

这样也带来许多遗憾。因为还有一些内容我在原计划里是想写的，它们对游戏软件的编写也是至关重要的，比如设计模式。可惜我在动手的时候，发现自己这些方面的修养在项目中用用还可以，离著书立说却差得很远，只好放弃。

曾经年少轻狂。但随着时间推移，越发发现自己所知所见，只是沧海一粟。学得越多，越是不敢轻言所知。所以，希望读者相信，我是非常谨慎地来完成自己的第一本书的。孔子说过，无友不如己者。我真诚地尊重所有对本书提出批评的朋友。可以去我的个人主页（http://www.codingnow.com）留言发表你的看法，或给我发E-mail：cloudwu@gmail.com，

我个人更倾向电子邮件这种交流方式。

希望听到大家的声音。

写书这件事情终归是告一段落了，我也将把精力投入到自己新的事业中去，或者很久都不可能再以这种形式将自己的技术人生做一个总结。身处当今热门行业中，可谓衣食无忧，而在国内的游戏制作圈子里混迹多年，业内也小有名气。而这些并非我去追求的，它们自然而生。这本书也一样，我没有刻意去追求表达些什么，也从没有想过本书会给我带来些什么。我现在是中国并不成熟的游戏制作行业中的一员，游戏给了我太多，我告诉自己需要做一点事情。分享知识和经验是我的义务，别无它。

云风

2005 年 5 月 23 日晚于广州

目　录

第 1 章　计算机，游戏，我 —— 1

我，云风，从二十年前的计算机游戏萌芽阶段一路走来。计算机，并不神秘。程序，也只是表达计算机控制逻辑的符号而已。

1.1　计算机 …… 2
1.2　计算机游戏 …… 3
1.3　计算机与我 …… 7
　1.3.1　启蒙 …… 7
　1.3.2　编程 …… 9

第 2 章　算法，程序的灵魂 —— 13

算法，先于计算机存在于世，比编程语言本身更为重要。语言只是工具，算法才是灵魂。这里简单介绍了数据结构的基本知识、算法效率的评估方法、几种 NP 问题的定义。

这里对游戏中最常用的搜索算法问题之一寻路问题做了些许探讨，并展开介绍了在以后可能被广泛使用的智能算法。最后用几个实例探讨了优化问题。

2.1　程序=算法+数据结构 …… 14
　2.1.1　算法 …… 15
　2.1.2　数据结构 …… 17
2.2　搜索算法 …… 23
　2.2.1　地图寻路问题 …… 23
　2.2.2　博弈问题 …… 27
　2.2.3　更为广泛的运用 …… 28
2.3　智能算法 …… 29
　2.3.1　遗传算法（Genetic Algorithm） …… 29
　2.3.2　模拟退火算法（Simulated Annealing） …… 31
　2.3.3　禁忌搜索（Tabu Search） …… 33
　2.3.4　人工神经网络
　　（Artificial Neural Network） …… 34
2.4　优化 …… 36
　2.4.1　质数问题 …… 36
　1.4.2　俄罗斯方块竞赛 …… 37
2.5　Apple II 上的编程之路 …… 39

第3章 编程语言 45

至少要学会一种编程语言，才可能编写游戏软件。流传于世的编程语言数以百计，但真正流行的却不多。各种编程语言之间有它们的共通性，但也并非能够一通百通，甚至也不存在一种适合解决所有问题的语言。

我认为大部分游戏程序员都应该掌握工作平台的汇编语言。这对理解硬件，写出更好的代码，还有软件调试都有莫大的帮助。

3.1 C 语言……46
3.2 BASIC……50
3.3 C++……51
3.4 汇编语言……54
3.4.1 概述……55
3.4.2 程序的本质……57
3.4.3 寄存器……58
3.4.4 寻址方式……60
3.4.5 汇编指令……61
3.4.6 C/C++ 语言和汇编……62
3.4.7 小结……63
3.5 其他语言……63
3.5.1 Forth……63
3.5.2 Lisp……64
3.5.3 Java……64
3.5.4 Python、Lua、更多……65

第4章 前Windows 时代 67

现在，Windows 一统江山，它帮我们隐藏了硬件的许多东西。这一章是对10年前知识的一个回顾，它们如今已经沉寂在历史中，或是被 Windows 掩盖。但是，作为喜爱刨根问底的程序员，我们有必要对 Windows 之前的时代有一些了解。

4.1 386 保护模式……68
4.2 VGA 到 VESA……70
4.2.1 超越 BGI……70
4.2.2 VGA……72
4.2.3 VESA 标准……72
4.2.4 花絮……74
4.3 保护模式下的开发工具……75
4.4 闲话 Allegro……81
4.4.1 用 C 与汇编写成的程序库……81
4.4.2 BITMAP……82
4.4.3 Sprite……85
4.4.4 几何图形和 3D……89
4.4.5 数据文件……91
4.4.6 声音……92
4.4.7 其他的部分……93

4.4.8 小结 …… 94
4.5 cfido 中国惠多网 …… 94

第5章 Windows 编程 —— 101

如果我们在 Windows 下开发游戏，那么理解 Windows 系统就是尤为重要的事情。Windows 只是一个操作系统的特例，不属于计算机理论知识的范畴。这导致了许多已经掌握了编程知识，理论功底也很扎实的程序员，面对 Windows 下的开发无可适从。

5.1 Windows 编程入门 …… 104
5.1.1 Windows 版本综述 …… 105
5.1.2 操作系统的核心 …… 107
5.1.3 Windows API 和 DLL …… 110
5.1.4 COM …… 111
5.1.5 Windows 的窗口和消息处理与传递 …… 114
5.1.6 Windows GDI …… 125
5.2 控制游戏的速度 …… 130
5.3 浅谈 MFC …… 132
5.4 小结 …… 132

第6章 汇编优化 —— 135

汇编优化，是在高性能游戏引擎设计中很难避免的一个环节。虽然不用汇编去做优化依然可以写出高性能的程序，但优化却可以变得更好。

但是，这一章并非从汇编开始谈起。相反，一开始我就花了不少笔墨谈汇编之外的知识。即使是一些技巧性的细节，也并非必须用汇编才可以做到。

这一章着重介绍了在 C/C++ 中和汇编进行混合编程的方法和技巧，然后才具体谈到汇编指令的高效使用策略。

6.1 浅谈代码优化 …… 138
6.2 并不仅仅是汇编 …… 139
6.2.1 在算法实现时减少上下文的依赖关系 …… 139
6.2.2 低效的静态变量 …… 140
6.2.3 数据的组织 …… 141
6.2.4 消除除法 …… 142
6.2.5 避免过大的循环 …… 144
6.3 汇编和 C/C++ 的混合编程 …… 146
6.3.1 使用外置的汇编编译器 …… 146
6.3.2 使用内嵌汇编 …… 148
6.3.3 C++的函数调用规则 …… 150
6.3.4 寄存器的使用 …… 152
6.3.5 _declspec(naked) …… 152
6.4 指令使用的技巧 …… 153
6.4.1 LEA：别样的内存寻址指令 …… 153
6.4.2 BSWAP：扩展寄存器的个数 …… 154

6.4.3 缩短指令的字节数 …… 155
6.4.4 避免跳转指令 …… 159
6.5 Pentium 的汇编优化策略 …… 161
6.5.1 UV 流水线 …… 163
6.5.2 AGI 地址生成互锁（Address Generation Interlock） …… 167
6.5.3 Pentium Pro，Pentium II 和 Pentium III 的汇编优化策略 …… 168
6.5.4 Pentium 4 的汇编优化策略 …… 178
6.5.5 如何测试代码的速度 …… 181
6.5.6 汇编能干的事情 …… 183

第 7 章 SIMD 指令 —— 185

单指令多数据(SIMD, Single-Instruction-Multiple-Data)编程是现代 CPU 高速处理多媒体数据的发展方向。好的工具摆在一边不去用就是一种浪费。

这里介绍得多的是 MMX 使用技巧，这方面也是我用得比较多的，勉强算得上经验丰富。

7.1 单指令多数据编程 …… 186
7.1.1 Alpha 混合 …… 187
7.2 MMX …… 191
7.2.1 MMX 的指令概况 …… 192
7.2.2 SSE 及其对 MMX 的扩展 …… 194
7.2.3 SIMD 技术在 C 语言环境下的使用 …… 196
7.2.4 MMX 编码技巧 …… 197
7.2.5 SIMD 的应用举例 …… 201
7.2.6 SIMD 编程注意要点 …… 203
7.2.7 SIMD 的性能检测 …… 204

第 8 章 引擎中的优化 —— 205

单从内容来看，这一章是本书的独家内容。我得以成名的开源游戏引擎“风魂”的几乎所有优化策略都在这章公开出来。虽然引擎源码一直都是公开的，想研究的人自然可以通过阅读代码来学习，但是，由作者本人来解释其中的优化策略，这是第一次的文字记录。

8.1 Blit 的优化 …… 207
8.2 脏矩形优化 …… 209
8.2.1 “风魂”中改进的脏矩形算法 …… 215
8.2.2 滚动优化 …… 219
8.2.3 覆盖优化 …… 222
8.2.4 脏矩形在 3D 游戏界面中的运用 …… 224
8.2.5 时间和空间的权衡 …… 229
8.2.6 关于估算 …… 230

8.3 自己设计精灵格式 …… 231
8.4 动态数据加载 …… 234
8.5 小结 …… 236

第9章 C 和 C++ —— 237

C++ 是我这几年的最爱。如果程序员没有陷入工程项目的泥沼，单纯做理论学习研究的话，我相信很多人都会爱上博大精深的 C++ 语言。

C++ 被程序员圈子最大的误解就是效率低过 C 语言。这里，我想为 C++ 再次平反，已经有好多人干过这事，我多做一次也不为过。我想证明的是，至少在 PC 平台上，好的 C++ 程序不仅不会比同样的 C 语言实现慢，甚至会略快一些。

当然，我选择 C++不仅仅因为它的高效，还在于 C++语言的优雅。本章重点介绍两个枝节问题，一是内存的管理，二是 template 的使用，这都是广泛使用 C++后一定会遇到的问题。

9.1 从 C 到 C++ …… 239
9.2 C vs C++，效率至上 …… 243
9.2.1 全局变量 …… 243
9.2.2 函数调用的堆栈处理 …… 243
9.2.3 函数调用的参数传递 …… 244
9.2.4 临时变量的堆栈使用 …… 244
9.2.5 代码生成 …… 244
9.2.6 异常 …… 244
9.2.7 inline …… 245
9.2.8 多态性 …… 245
9.2.9 标准库的差异 …… 245
9.2.10 动态内存分配 …… 245
9.3 优雅的 C++ …… 246
9.3.1 消失了的宏 …… 247
9.3.2 const 修饰及类型转换 …… 248
9.3.3 隐藏实现 …… 248
9.3.4 引用而非指针 …… 250
9.3.5 名字空间 …… 250
9.4 C++ 的误区 …… 251
9.4.1 类层次过细 …… 251
9.4.2 滥用操作符重载 …… 252
9.4.3 滥用标准容器 …… 253
9.4.4 滥用动态类型识别和异常 …… 253
9.4.5 滥用多重继承 …… 254
9.4.6 忽视 C++高级特性的复杂度 …… 255
9.4.7 学习 C++ …… 256
9.5 动态内存分配的优化 …… 257
9.5.1 另一种优化策略：多次分配，一次释放 … 266
9.5.2 垃圾回收 …… 268

9.5.3 再论动态内存分配 …… 269
9.6 template …… 272
9.6.1 封装 C++的成员函数调用 …… 272
9.6.2 常数数量的对象复制 …… 274
9.6.3 对象计数 …… 275
9.6.4 避免重复代码 …… 277
9.6.5 选择最佳的容器 …… 279
9.6.6 延迟运算 …… 281
9.6.7 编译时计算的游戏 …… 283
9.7 小结 …… 286

第 10 章 调试 —— 287

这是更偏重于实用的章节。方便软件调试的编程技巧很容易被没有项目经验的新程序员所忽略。

这里讲解了 ASSERT 机制的使用、log 模块的重要性，以及为游戏做一个录像的机制来配合自动测试以及 bug 的重现。

最后提到的内存检测的类，是延续上一章的代码。完整的代码都在文中列出，甚至可以直接输入软件中使用。

10.1 assert 及程序错误的提前发现 …… 288
10.1.1 崩溃的程序不说谎 …… 289
10.1.2 为软件维护多个版本 …… 291
10.1.3 前置条件和后置条件 …… 292
10.1.4 自己实现一个 assert …… 292
10.1.5 不正确的 assert 使用 …… 293
10.1.6 为 assert 分级 …… 294
10.1.7 assert 的引申：检查函数返回值 …… 295
10.1.8 释放的指针需要置空吗 …… 297
10.2 让程序更利于调试 …… 298
10.2.1 录像 …… 299
10.2.2 实现一个 log 类 …… 300
10.2.3 跟踪代码运行的流程 …… 302
10.2.4 为内存分配加上一个壳 …… 302
10.3 RELEASE 版本的调试 …… 310
10.4 小结 …… 311

第 11 章　脚本 313

嵌入式脚本在现代游戏软件中已经成了必不可少的模块。除了讨论嵌入式脚本本身的优劣和适用范围外，我还选出了 Lua 作为实例介绍给大家。这里可以看到，Lua 是如何嵌入到软件中，并且怎样服务于游戏逻辑的。这里我将给出一段可以使用的代码，展现我是如何做到对象封装的。

很特别的，在本章的后半段提到了网络游戏反外挂的技术。好的嵌入式脚本是防止反编译游戏逻辑的一个基本前提。本人觉得，反外挂实在是一个不登大雅之堂的技术，所以不想给它另开辟一章专门介绍了。

11.1　为什么需要脚本 …… 314
11.2　应该用脚本做些什么 …… 316
11.3　脚本不应该做什么 …… 318
11.4　如何嵌入脚本语言 …… 319
11.5　选择一种脚本语言 …… 321
11.6　实战嵌入式脚本 Lua 5 …… 324
11.6.1　嵌入 Lua …… 324
11.6.2　使用 coroutine 驱动游戏逻辑 …… 326
11.6.3　封装 C++中的对象 …… 328
11.6.4　回调机制 …… 336
11.7　反外挂技术 …… 338
11.7.1　防止加速器 …… 338
11.7.2　防止分析伪造网络包 …… 339
11.7.3　防止逆向工程 …… 341
11.7.4　防止代码钩子 …… 344
11.7.5　惩戒外挂作弊者 …… 344
11.7.6　安全地将信息传递给服务器 …… 345
11.7.7　防止模拟输入 …… 346

第 12 章　版本控制 349

我自己在进入网易做《大话西游》系列开发之前，是没有考虑过版本控制的问题的。甚至，我见到了许多游戏开发公司都没有做好这些。在四年多中经历数个网络游戏项目后，我深深感到版本控制在一个软件项目开发中的重要性。

12.1　常用版本控制软件 …… 351
12.2　版本控制工具在项目中的使用 …… 355
12.3　不仅仅是程序才需要 …… 358

第 13 章　开发方法 —— 361

这是个人对软件开发的经验体会。我既参与过失败的游戏项目，如《大话西游》的最早版本，对于失败深有感触；也在几个游戏项目中体会到了成功的喜悦，如后来的《大话西游 II》和《梦幻西游》，我都从头到尾地目睹了它们从无到有成长到百万用户在线。

我在它们上面付出了许多的时间和精力，或多或少都悟到点什么。尽管文字的传达必然会损失掉许多色彩，但我已尽力来和读者分享。成功很难复制，失败也难回避。过去的历史却总能给我们一些借鉴。

13.1　失败的经验 …… 363
13.2　成功的经验 …… 365
13.3　对游戏开发方法的一些思考 …… 369

第 14 章　编程和游戏 —— 373

这章是本书的一个总结。本书以编程技术内容为主干，但却以非编程内容而结束。我因为爱游戏，才去制作游戏，因为制作游戏，才去学习和钻研编程技术。而最终，我是一个游戏玩家，好的游戏，而非好的游戏程序，才是我的目标。

与游戏程序员们共勉。

14.1　操作 …… 376
14.2　角色设定 …… 378
14.3　再谈技术 …… 379
14.4　浅谈网络游戏 …… 381
14.5　小结 …… 382

后记 —— 383

致谢 —— 385

第1章

计算机，游戏，我

我，云风，从二十年前的计算机游戏萌芽阶段一路走来。计算机，并不神秘。程序，也只是表达计算机控制逻辑的符号而已。

1.1 计算机

于我看来，计算机绝对是上世纪人类最伟大的发明。几千年来，人类一直在寻求借助自然之力从体力和脑力上增进自己能力的方法。各种扩展体力的工具被一一发明，而增进脑力方面，却一直走得很慢。直到最近半个多世纪才得以飞速发展。

计算机在不到 60 年的发展中，如今已经可以提在公文包中，虽只有不到一公斤的重量，却可以播放高清晰的电影，把玩逼真的三维游戏。和几十年前塞满了难看的电子管，拖着长长的电线，占据着整个房间的笨重的大家伙相比，简直是天壤之别。但是，如果我们把看起来极为不同的它们从数学上理想化，本质上却是一样的！

20 世纪 40 年代，计算机的始祖和他的小孙子，办公桌上摆放的精巧的笔记本，其实都可以简化成这样一种机器：它每次接收一个输入，然后根据自己内部状态的不同，按事先定好的规则表做出相应的操作。这，就是图灵机。

现在计算机的奠基人之一阿兰·图灵（Alan Turing）提出的图灵机的原始描述大约是一个盒子，里面有一个状态寄存器，用来保存机器当前的状态。图灵机所有可能的状态数目是有限的，并且有一个特殊的状态：停机。有一根很长的纸带通过盒子，纸带上面被划分成一个个的格子，每个格子里都标记有符号，符号的种类是有限的。图灵机拥有一个对纸带的读写头，可以读入或者改变当前格子的符号，并且能左右移动纸带。

这台机器有一套控制规则，根据机器所处的状态以及当前读写头所指的格子上的符号来确定下一步读写头的动作，并改变状态寄存器的值，如图 1-1 所示。图灵认为，只要有无限长的纸带，对于每一部分都是有限的机器（有限的状态数、有限的规则、有限的符号种类），就可以模拟人类所能进行的任何计算过程。

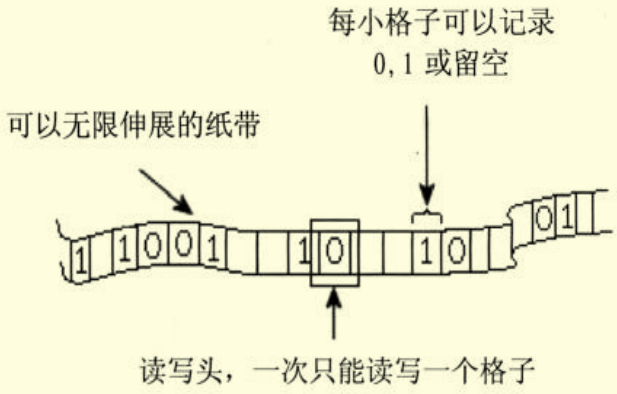


图 1-1　纸带示意图

有限的符号种类实际可以换成数字来表示，数字如果使用二进制，那么用 0，1 就可以表达出来，只需要增加读写头的移动次数。而利用机器内部规则的控制，用多个格子的 0，1 表示就可以了。如今的计算机其实还是让保存在储存器中的一串 0，1 数字，经过处理器的内部规则的处理，将结果以 0，1 的形式放在储存器的合适地方，或者去驱动别的设备罢了。

这，就是计算机。

1.2　计算机游戏

计算机游戏的历史并不比计算机本身的历史短多少。娱乐，是人的天性。早在 1952 年，Cambridge 大学的 A.S.Douglas 在他的博士论文中就实现了第一个有图形显示的计算机游戏——一个井字棋的版本。当时，这个游戏可以在阴极射线管上显示出游戏画面（如图 1-2 所示）。

井字棋，老外叫做 Tic-Tac-Toe。很多人小时候也玩过吧。用树枝在地上画个井字，把棋盘分为九格。两人轮番把自己的棋子放在盘中，如果有人可以将自己的三枚棋子连成一条直线就算胜出。如果到九格填满尚未有人达到目标，即为和棋。

闲话井字棋

（Tic-Tac-Toe）

小时候，我觉得这是个穷极无聊的游戏。稍有头脑的人就不可能在这个游戏中输棋。当然，我也从未赢过。

但这个游戏在计算机史上占有非常重要的地位。除了是计算机游戏的鼻祖外，据说，微软的创始人 Bill Gates 也是从编写这个游戏的计算机程序版本走向未来之路的。这个游戏还频繁出现于人工智能领域的研究中。如今，居然还有一个井字游戏存在于 MSN Messenger 的在线游戏菜单里，颇有些复古的风味。

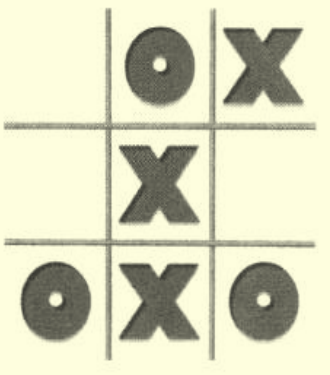


图 1-2　游戏画面

在此不久，便有了视频游戏（Video Game）的诞生。1958 年，出自 William Higinbotham 的 Tennis for Two（图 1-3）；1962 年，出自 Steve Russell 之手的著名的 spacewar!（图 1-4）。

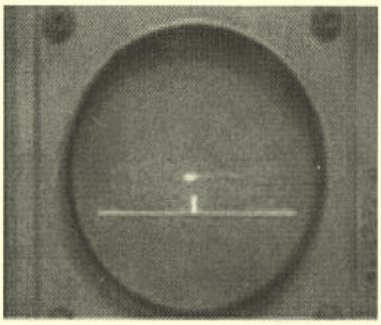


图 1-3　Tennis for Two，世界上第一个视频游戏，简单，但是很有趣

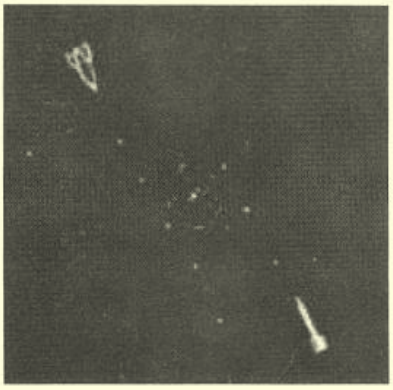


图 1-4　spacewar! 更接近现代充满竞技性游戏的作品，可以接受双人对战

计算机游戏由美国人发明，但却被日本人继承并得以极大的发展。美国人发明的 Atari 2600 虽然并非是世界上第一台家用游戏机，但它应该是广为国人所知的最早的家用游戏机了（如图 1-5 所示）。



图 1-5　Atari 2600

我的小学同学家中就有一台，虽然他家那条大狼狗经常追着我跑，让我很害怕。但是我当年经常不顾生命危险，三天两头往他家里跑，就是为了蹭一下游戏玩。

没过多久，大家的乐趣就被日本的游戏机吸引过去，那就是任天堂的红白机（如图 1-6 所示）。



图 1-6　红白机

我家里现在就还有这么一台游戏机，其寿命经久不衰。直到最近两年，我还在一些朋友家中，看到新一代的小朋友玩得不亦乐乎。我们当年也是捧着几盘游戏卡，串门聚集在一起围着一台电视机，抢夺手柄的呢。

红白机（Famicom）的主处理器是 6527，这和同时期著名的个人电脑 Apple II 采用的处理器是基本相同的（Apple II 使用的是 ROCK WELL 公司的 6502 芯片，指令集和 6527 相同）。

Apple II 是一款相当出色的个人电脑，虽然 IBM PC 发布的时间并不晚多少，性能上也远强过它，但是，无疑，Apple II 在他那个年代，是全世界最流行的个人计算机。包括著名的 DOOM 之父，John Carmark 在内的许多 PC 游戏史上著名的游戏程序员，都曾经沉迷于 Apple II 的游戏开发。我自己的家中，还有成箱的软盘装载着运行其上的游戏。



图 1-7 Apple II

历史发展到今天，这些古老的玩意已经快消失在我们的视线之中。游戏机平台如今被 Sony 的 Playstation2、微软的 X-Box，任天堂的 Game Cube 三分天下；而个人计算机则由 IBM PC 发展而来的个人电脑标准占据绝大多数的市场。游戏也慢慢分为游戏机游戏和个人计算机游戏两派。

近年来，随着技术发展，各大游戏厂商把新推出的游戏发布到各种平台上。微软也宣布在未来的 Windows 平台加强对游戏的支持。似乎，游戏平台又将归于统一。

而我，一个游戏程序员和设计者，从操作系统、硬件环境的编程枝节中走出来后，已经不那么在乎平台的问题了。游戏，就是一个娱乐我们的玩具；又或是表达设计者情感及人生哲学的一种工具；甚至于有人把它提到第九艺术的高度。这些，都暗示着我们，计算机游戏最重要的是游戏性和艺术性，与平台何干。

背景

1977 年，苹果推出了世界上第一台真正的个人电脑 Apple II，改变了整个电脑行业。其后十多年里，它一直是世界上最普及的个人电脑，孕育了无数优秀的程序员，比如世界上最出名的游戏程序员 John Carmark 就是从 Apple II 上起家的。

在中国，有好像是天津无线电二厂这样“无畏的盗版者”，仿制出完全兼容 Apple II 系列中最成功的 Apple II+ 的国产机型——中华学习机，其实仅仅只是缩减了 Apple II+ 的扩展槽的数量，以及将一个并不优秀的中文系统与一个有趣的 LOGO 语言固化在了 ROM 中。

1.3　计算机与我

1.3.1　启蒙

1984 年，刚刚进入小学的我见到了生命中的第一台电脑。我的父母都是一家国有大钢铁厂的小职工，拿着每月百多块的工资。或许是一家之主的父亲更年轻求学的时候学过一些计算机编程的知识，居然拿出了一家多年全部的积蓄，花了六百多元人民币买下了一台港产的家用电脑 Laser-310（如图 1-8 所示）。在此之前，家中唯一的电器就是一台九寸黑白电视了。



图 1-8　港产的家用电脑 Laser-310

回忆当年的情景，我带着一丝好奇，在 1986 年游戏机出现在武汉街头之前，完全不知道这玩意有什么用。只看见老爸总是很晚才睡，在我已经躺在床上后，开始鼓弄那台机器：一个小小的塑料盒子，上表面有一组被称为键盘的咖啡色按钮，屁股后面牵出一跟电线连到电视机上，而电视屏幕上不再是电视剧、动画片，而换之以一些跳动的字母。

几天之后，放学回家，父亲向我展示了被他称做电脑游戏的东西。当连接在电脑上的磁带机被按下，一段不太长时间的等待后，电视屏幕上的字母消失了，黑色的底板上出现了几个亮点。屏幕下方是四个白色小方块组成的倒 T 字，上方有三个亮点水平地滑过。那是一架飞机，两个并排的小方块组成了机身，45 度斜上有一个方块表示的机尾（如图 1-9 所示）。我被妈妈捏住手，在键盘最下面一个长长的被叫做空格键的

儿时的我，有着众多的爱好，刚刚学会写字那会，喜欢拿着红色的碎砖头趴在地上写下满地的汉字，为此，母亲买来笔墨纸砚让我学书法，可惜没多久，我就爱上了用彩色铅笔涂鸦。

母亲或许在我随处涂鸦的图案中发现了小孩子的天赋，拼命想把儿子向绘画方面培养，可惜在浪费了一盒颜料、两瓶墨水、三根毛笔、数打宣纸后，正式宣告了国画和这个小孩子无缘，那个涮笔的缸现在还积满灰尘地闲置在柜子中。

同样，我也没能继承父亲的手艺，自己扎的风筝没有飞起来；对无线电两三天就失去了兴趣；也没能学会用泡沫或者木头做模型，虽然我的童年充满了父亲亲手做的各种精致的玩具。

按键上按了一下。“嘟”的一声，倒 T 字型的炮台上发出一发炮弹，两秒之后和飞行的飞机交错而过。我迅速理解了这个新玩具的游戏规则，并在一局飞过的 20 架飞机中击落了 7,8 架。

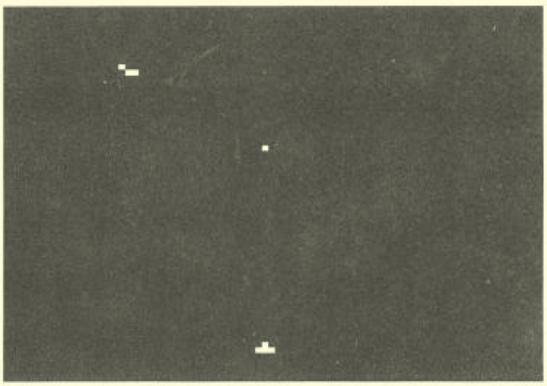


图 1-9 “打飞机”游戏

小时候，父母对儿子的评价就是，这小家伙对什么都有兴趣，就是耐不住性子好好钻研。电脑应该也是众多兴趣中的一例吧。谁也没有意识到，这玩意竟然会引导着我活过的这二十多年。

从打飞机的游戏以后，我又拥有了魔方、跳棋等游戏，同时，算算术题也变成了游戏之一。起因是，数学老师要求家长每天出 20 道计算题，最开始是 20 以内的加减法，然后扩大到 100 以内的九九乘法。这项工作居然被父亲“偷懒”交给了计算机。做错了的话，机器会发出“嘀嘟”的叫声，并用汉语拼音写出“你做错了”的字样；而如果算对了，屏幕上便会显示一副熊猫啃竹子的图片。

那只熊猫是画在坐标纸上的，大家不应该对坐标纸感到陌生，中学学代数时，老师就让我们在上面画过函数图像。

嘿！原来这玩意除了能画出几个方块以外，还可以画出图案来呀。更让我觉得惊奇的是，我在父亲的桌子上还看到了一张手绘的熊猫图片。是怎么把纸上的图案变到电视里去的呢？在一个生活在 20 世纪 80 年代中国的孩子眼里，觉得能把笔下的东西变成机器上的玩意是多么了不起呀。同班同学的一则小作文被报纸刊登，他的文字被用端正的仿宋体铅印出来，都让大家羡慕不已。如果能让自己的画也出现在电视屏幕上，那岂不是更值得炫耀的事情？

父亲从来没有放过引导儿子新兴趣的机会，在此之前我都是打游戏的兴趣，而有一天，儿子突然对这些游戏的背后有了兴趣，当然是更值

得高兴的事了。我不记得当时父亲是组织什么样的语言给我讲解的，但我知道，当时我是理解了其中的原理。一个六岁大的孩子能理解的东西，我相信即使用更理论化的语句在这里写出来，各位读者也不会有什么理解上的困难。

说明：想在计算机屏幕上表示一张图片，通常有两种方式，其一是基于点线面的描述，现今网络上流行的 Flash 中用到的矢量图形和大量充斥着现今的电子游戏市场的三维游戏都是基于此。而当时父亲给我讲解的是另外一种更常用的位图表示法。几乎所有的二维游戏，还有互联网上花花绿绿的图片都是基于位图的表示方法。而且前面提到的矢量图形和 3D 图形，到了显示的最后一步，反映在显卡中的储存器中的依旧是位图。

显示屏上的图像其实是由许多的亮点横竖整齐地排列构成的。计算机里面的一个数据块描述了每个点的状态，或亮起或熄灭，又或者是对应着每个点的颜色，这些描述图像的数据，通过某种设备表达到了屏幕上，展现在人们的眼前。

这些年，我游逛于一些关于游戏制作的 BBS，见过一些未入门的小孩子发贴询问如何显示一张 BMP 图片，或如何用 Turbo C 绘制图片？我都觉得，这些或许真是未来中国游戏产业的生力军，真可惜了他们不得门而入啊。

1.3.2　编程

在计算机编程这个领域，我们应该从基本学起，万事需要追根探源。如果总把思想停留在文件，或者某种编程工具上，如何能深入下去呢？比如计算机的图像，那就是一些数据的表述，而图像文件无非是这些数据的集合，想显示出图像，只需要把文件中的数据读入，按硬件的要求把它们重新组织，传送给显示设备就够了。

当然，当年我的父亲不会对一个刚读小学的小孩子讲什么图灵机、二进制，但他让当时的我明白了，可以用一些数据描述出画在坐标纸上

的熊猫，通过给计算机编写一些控制它的叫“程序”的东西，把图案显示在屏幕上。

当时没有鼠标可以在屏幕做画，没有扫描仪，更没有数码相机。那台用 Z80 做 CPU，只有 16KB 内存，使用磁带机做外部储存设备的 Laser-310 在进入我家后，所有的软件都是父亲自己动手编写的。他并没有制作一个用键盘绘图的软件，所以，程序中若用到复杂的图案，就是先在坐标纸上画好，再用数点的方式，把数据手工输入到机器中。

坐标纸上的熊猫引发了我对计算机游戏制作的兴趣，自然会缠着父亲教我，当时，计算机的真正玩家能干的事情只有一样，就是编程，我见到了生命中的第一个程序（如图 1-10 所示）。

世界上最大的软件公司微软的创始人比尔·盖茨就是从为一台最早的普及性微机开发 BASIC 解释器而发家的，以至于微软到现在依然对 BASIC 情有独钟。另一本软件技术商业杂志的领袖，Dr.Dobb's Journal 的诞生也是为了出版几期关于 Tiny BASIC 实现的新闻通讯。

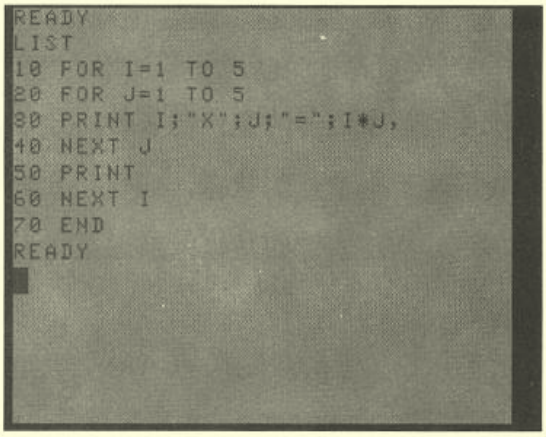


图 1-10　我见到的第一个程序

这是一个 BASIC 程序，BASIC 是计算机发展史上的一个传奇。它没有显赫的出生，最初甚至于仅仅是另一种学术界非常流行的 FORTRAN 的简化版，简直就是专门给业余计算机爱好者娱乐用的玩具。可是，这个世界如果没有 BASIC，计算机行业肯定不是现在这个样子。

熟悉 C 语言的程序员往往很看不惯 BASIC 的语法，也不屑于使用这种语言。早期纯正的 BASIC 只支持过程化的编程，甚至连函数调用都是很晚才加入标准的。但是，我以一个有着亲身经历的过来人发言，小小的 BASIC 的确是让小孩子也能理解编程的最佳选择。

Laser-310 的 ROM 里就固化了一个简单的 BASIC 解释器，不只

是这个机型，早期的家用微型计算机为了减少成本，都没有配备硬盘，取而代之的都是在 ROM 中固化一些供人娱乐的小玩意，BASIC 解释程序属于标配。只需要一开电源，机器立刻就能进入 BASIC 的人机对话环境，你可以输入 BASIC 指令立即运行，也可以一行行编写程序。

上面这个程序用了两个 FOR 循环就可以在屏幕上输出 5 以内的乘法表，有点编程基础的人即使完全不懂 BASIC，应该也能理解上面的代码吧。可惜当年的我连 26 个字母都无法顺利地读出其英文读音，早期为了学习 BASIC 语言，我是靠字母键盘上的按键位置以其汉语拼音读音来记忆指令的，好在 Laser-310 在键盘上将每个 BASIC 语句印刷在按键上，并可以直接用热键按出来，这极大地方便了不懂英文的我来学习编程。

但是对于一个还不具备清晰的逻辑思维的小孩子，上面的程序还是太复杂了，我几乎无法理解什么是变量，什么叫赋值，只是父亲在我的眼前把上面第 30 句中那个 "x" 改成 "+" 后，运行结果里出现了诸如“3 + 4 = 12”这样的显示后，让我似乎明白了点什么。只要学好编程，计算机就能听我的话，我让它算对题，它就能算对，让它按别的规则算错，它也不会抱怨，这将是个非常听话的玩具，它能做到的事情取决于控制他的人。

小学 3 年级的时候，我见到了街头的游戏机。原来游戏不仅仅是几个亮点，几张静止的图片和嘟嘟几声，游戏居然可以这样绚烂多彩。可惜家教甚严的我，只在夏天能有每天买冰棍的一角零花钱，投币的游戏机一个币需要 3 角，真是个天价。更多的时候，放学后，都只能站在街机旁看人娱乐。我时常地想，如果家里的那台计算机也能有这么好玩的游戏就好了。不知道为何，脑海接着闪过的第一个念头居然是，没有我可以自己做嘛。

找出了坐标纸，开始设计自己游戏中的角色。那个时候的男孩子都有些共同的爱好，比如打电子游戏，看动画片；对于我，还喜欢把屏幕上的角色搬到纸上。之前的点滴学习，还是粗略了解了一些基本的概念，比如计算机的分辨率，就是指屏幕上横向和纵向最多能排下多少像素。

有了这样的概念，我就知道应该在坐标纸上画多大的图，最后在屏

幕上将是多大。虽然我还不知道最后应该怎么编程，但是这个世界上没有知识是学不会的，不是吗？如果一开始学不会，就可以把问题细化分解，然后学习更基本的知识，最后，所有的问题都能变得和 1+1=2 一样简单，我需要的只是时间。

而时间，对于一个小孩子来说，是最不缺的，当时年幼的我就是这样想。家里就两本讲 BASIC 语言的书，统统被塞进书包，闲下来就看看；订的一份叫做《儿童计算机世界》的报纸，上面经常会刊登一些有趣的 BASIC 小程序，不会自己编，照着敲总会吧。那时，这种供学习用的家用电脑有很多品牌，各种机器间的固化 BASIC 也不太相同，大约我是从“移植”这些小程序开始学习编程的，模仿永远是最快的学习途径。

画了几张坐标纸，代码在课间时间、午饭后的休息期间爬满了笔记本后，处女作最终还是流产了，因为我意识到自己的能力还是不够。同时，父母终于答应我买一台当时感动着无数小孩子的任天堂红白机了。

虽然只有一盘卡带，魂斗罗，我还是不厌其烦地玩着，直到可以一次不败地通关，甚至可以蒙上眼睛顺利过第一关。稍后就开始仔细观察这个游戏，这么丰富的画面，如果一个个点地画上去，那不知道要输入多少数据，写多少程序啊。

父亲告诉我，游戏的画面其实是一块块地拼接起来的，而且，画面中的所有东西还是按一定逻辑用程序控制起来的。无论是数据还是程序，都会被重复利用。我一知半解地接受了新的知识，唯一肯定的是，我还需要更多的学习。

他们那个年代的大多数人，由于众所周知的原因，在应该读书的年龄没能上大学，只有在工作后想办法弥补（许多那个年龄的人都希望有个文凭）。

就在小学的第五年，母亲开始上成人大学的大专。那所学校离我家不远，学校里有计算机编程课，讲授的就是 BASIC，我被母亲带去一同上课了。一屋子三四十岁的成年人中混着一个认真听讲的小孩，作业做得最好的恐怕也非这个小孩莫属，不为虚无的文凭，只是想学编程，想做游戏。

在和大群的玩伴一起堆泥巴，挖蚁洞，滚铁圈，玩洋画，爬树，钓虾，打游戏机的童年中，我带着这样一个与众不同的爱好步入了中学，迈向了程序员之路。

第2章

算法，程序的灵魂

算法，先于计算机存在于世，比编程语言本身更为重要。语言只是工具，算法才是灵魂。这里简单介绍了数据结构的基本知识、算法效率的评估方法、几种NP问题的定义。

我们得以知道，现有的计算机是不可能完美解决任何问题的。

这里对游戏中最常用的搜索算法问题之一寻路问题做了些许探讨，并展开介绍了在以后可能被广泛使用的智能算法。最后用几个实例探讨了优化问题。

初中的时候，我开始进行系统的编程学习，一种以参加 NOI（全国青少年信息学奥林匹克竞赛）为目的的训练。因为我在编程方面有那么一点点天赋，在湖北省的编程比赛中，以初中生的身份参加高中组比赛取得了还算不错的成绩，所以在初二那年的暑假，参加了省里的集训队。

武汉的夏天是非常炎热的，整个夏天，每天我都要坐进蒸笼一般的公共汽车，去离家相当远的武汉测绘大学上计算机课。课堂上使用的是本科生用的数据结构的教材。在大学的教室里，老师给我重新整理了头脑中条理不清的链表、树、图这些概念。刚开始，我还带着一点点自负，可是没多久就认识了自己的不足。

第一堂课上，我被老师叫到了讲台。让我当着所有同学的面写出一个二分查找的程序。按道理说，这是个非常容易的问题，可是我居然写写改改，在黑板上磨蹭了半天才留下个坑坑巴巴的程序。这告诉我一件事情：编程的基本功，光靠理解是无法随手写出正确、清晰的代码的。

编程，同样讲究熟能生巧。

2.1 程序=算法+数据结构

程序是什么？我的诸位老师都曾经这么教导我，程序等于算法加数据结构。算法（algorithm）是解决问题的方法，如果我们把任何需要计算机解决的问题看成一个需要求解的函数的话，算法就是讲输入转换成输出的方法；数据结构（data structure）就是把需解决问题中的信息用计算机进行的数字描述方法，以及相应的对这些信息的操作。

不少程序员做软件开发，脑子里想的是如何把各种控件组合起来使用，该在哪些地方填写一些逻辑控制代码等等。

这个是我们在课堂和理论书籍上可以学到的知识。但今天，大多数以写程序来谋生的程序员却在工作中感觉到了理论的脱节。在商业软件的开发中，我们面临着需求经常变化，业务逻辑复杂，涉及各种硬件平台的使用等各种情况。书本上教授的基本算法和数据结构看似无用。

计算机语言的发展似乎也在引证同样的道理。现代编程语言，不再重视程序逻辑的实现和数据的简单描述方法，而注重问题的分化、类层

次的设计、模块的封装。如何更好地实现设计模式，是使用编程语言开发软件的重中之重。

但是，我们不可以抛弃编程的本源，尤其是在游戏程序设计中。

把整个软件化为一个个片断之后，其本质依旧是算法和数据结构。游戏程序更是没有一个通行的公式来实现，也不会有傻瓜式的编程模板。这是一个永远需要创造力的挑战。几乎每个成功的游戏，都是先有了创意，再想办法实现之。那么和前人不同的地方，总需要我们程序员想新的方法来解决。

游戏程序员是聪明的工程师，而不是学术理论大师。受限于计算机硬件的空间限制和游戏的实时性要求，并不是所有普适性的现成的程序库可以供我们直接来调用的。一个不能熟练运用各种基本算法和数据结构的程序员不可能成为优秀的游戏程序员。只有把这些基本问题烂熟于心的人才可以面对新问题时，做到游刃有余。据我个人之所见，大多数不太优秀的游戏程序，多出于程序员的编程经验不足，对相对复杂的算法或数据结构把握不够，或出于对复杂事物的恐惧心理而逃避所致。

在这一章，我们将讨论一些很基础的问题，但在阅读时请时刻注意，我们的思维不能局限其中。比如算法，我们知道，选择好的算法比优化代码更加重要，但是在研究算法的时候却不能陷入学术研究之中。

真正的学术研究应该是更加严谨，比现实的工程更加合理。可惜很多人对于学术研究往往只取其表，不取其里。给许多工程师们造成了学术研究不切合实际的错误印象。

2.1.1　算法

我们在讨论算法的时候，经常会讨论其空间复杂度和时间复杂度。

注意：理论书籍上讨论的复杂度，其实描述的是时空开销的增长情况。即，书本上关心的问题多是需要求解问题的规模扩大的时候，时间和空间开销的增长情况。而在实际项目，比如一个实际的游戏中，任何问题都是可以估计或者限制其规模的，我们程序员在预计其规模后，传统的时空复杂度的估算结果很可能不能正确表达出算法的适用度。

另一个算法设计中的关键点是对基本操作的评估。一次加减乘除或者内存访问可以算做一个基本操作；一次函数调用也可以看成基本操作。

API 的调用，对库函数的使用等都会在特定场合被认为是算法中的基本操作。大多数基本操作都是固定时间的，所以并不影响算法的时间复杂度。可是放在实际程序中，却对效率影响和速度估算有很大的影响。我们在对算法分析时，只有正确地认识所用到的基本操作的时间，才能够设计出最合适的算法。

计算机依照着摩尔定律在发展，处理器的功能和复杂性每 18 个月就增加一倍，而成本却成比例地递减。那么按一个游戏 3 年的开发周期计，我们在开发结束的时候，计算机的处理能力将扩大 4 倍，这对算法的设计有什么影响呢？能否说，我们能够处理的问题规模也扩大了 4 倍呢？令人失望的是，大多数情况下，我们还做不到。这跟算法的复杂度有关。当一个算法是线性复杂度 $O(n)$的时候，问题的规模允许扩大 4 倍。而当复杂度为 $O(n^2)$时，就只能是根号 4 了。所以，选择复杂度低的算法还是有很大的意义的。

通常用大 O 表示法（big-Oh notation）来表示问题的复杂度，它表达的是问题计算量的层次，O 即 Order 的缩写。$O(n^2)$ 表示基本操作使用的次数和问题的规模成平方关系。

常见的层次按以下表递增：

$$O(\log n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^k)<O(2^n)<O(k^n)<O(n!)$$

可惜，不少问题可以选择的算法复杂度还是太大，属于 NP 完全问题，甚至 NP 难度问题。实际上，在更多时候，我们去用近似算法求得近似解是最可行的方案。对于要求实时处理的游戏程序中，有更多的使用近似算法的需求。

首先，NP 指非多项式时间能够解出的问题是一个常见的误解。NP 其实只是指一个问题的解答可以在多项式时间被验证。一个问题属于 NP 问题并不表示这个问题很难，而 NP 完全问题可以看成 NP 问题中比较难的一部分，这些问题不可能用多项式时间得到解。

古克的论文证明，所有的 NP 问题都可以用某种方式归约成同一种 NP 完全问题。

NP 问题及其他

1971 年，古克（Stephen A. Cook）发表了 The Complexity of Theorem Proving Procedures 。把以多项式（Polynomial）时间解决为衡量标准的问题归成三大类，即 NP（nondeterministic polynomial），NP 完全（NP-complete）与 NP 难度（NP-hard）问题。

数学化的语言可能会让普通读者生厌，也超出本书应该讨论的范畴。如果需要了解更多，可以去查阅相关论文或者书籍。这里用简单的文字描述。

而 NP 难度问题，即比 NP 问题要难的问题是指无论怎样，我们既不能用多项式时间得到问题的解，也不能在多项式时间验证问题的解。比如围棋，怎样找到一种计算机必胜的策略？显然，理论上必胜的策略是存在的，但是这个策略的描述应该是我下某一步棋，对手下任一步棋的必胜应对方法。显然这套策略是否正确都不可能在多项式时间由现在的计算机来验证。问题的复杂度随着棋盘的大小（$N \times N, N=19$）的增加而剧烈增大。保守的估计也至少在（$n \times n$）!这个复杂度。

围棋这个问题中，不光是不能在多项式时间验证其解是对的。中间所需要的空间消耗也是成指数增长的。对应于时间复杂度的 NP-hard，空间消耗上也存在 PSPACE-hard 问题。Robertson 与 Munro 在 1918 年证明了围棋就是一种 PSPACE-hard 问题。目前有人估算围棋的必胜策略的计算量在 10^{600} 以上，而现在所知的宇宙原子数量却只有 10^{75} 左右。

一个好的算法应该同时在空间和时间上达到最优，这是无可挑剔的实现方法。但在很多情况下，两者互相矛盾。我们经常要考虑协调它们之间的关系达到一个平衡点。用空间换时间的做法在游戏中用得非常广泛，也就是预处理的方法；而用时间换空间的做法也不能忽视，它多采用重复计算的方式来实现。

算法的研究也离不开计算机硬件平台的特性。在目前可以看到的未来里，我们可以用到的算法至少是依赖一台非并行处理的计算机（虽然现代个人电脑已经实现了诸如超线程这样的机制，但解决问题用的算法大多并不会考虑使用并行基制来提速）；空间存储是允许随机访问的，并不会因为像以前的字带机或是磁带机那样只能按数据流来顺序存储。

本章不会就具体程序做具体代码的分析，因为由算法描述来写出高性能的代码属于程序设计方法的范畴。

2.1.2 数据结构

我们不能把数据结构狭义地理解为数据的组织方法。它还包含了针对这些数据组织方式的处理方法。在本书中，并不想对数据结构给出严

格的定义，严格的定义可以从更严谨的技术书籍中读到，定义对我们并不重要。

本章也不想做一个数据结构的教学篇，因为数据结构的专业书籍在书店的书架上可以找到许多。而且理论学习数据结构的知识也不是可以一蹴而就的，我不希望长篇累牍地堆砌概念，把读者限于枯燥无味的基础知识讲解中。所以，在阅读本章相关文字前，先学习了解数据结构的专业知识会更好。

流行的语言，例如游戏开发常用的 C++ ，在语言标准中已经实现了常见的数据结构供大家使用。STL（标准模板库）中就用现成的链表（list）、字典（map）等等。为什么我们还需要了解它们的实现方法？因为蕴涵在数据结构中的思想，是程序员的必修之课。它能启发你更多，让程序员按照更合理的方式进行编程。在特定情况下，我们还需要根据实际情况，对数据结构的实现进行改造，达到更高的效率。

1. 线形表

在数据结构中，我们通常面对的是数据的逻辑关系而非物理地址的联系。例如线性表，指的是有限的逻辑上的有序序列。这里的有序指每个元素都有自己的位置，并且有确定的前驱和后继（可以为空）。通常，每个数据都是类型相同的。

然而，根据计算机实现的不同，我们又可以把线性表分成两类：数组和链表。

数组指物理地址连续的表，可以用内存地址来检索到对应的元素。这样，我们可以用常数时间访问到指定位置的元素，但是在中间插入和删除元素时间复杂度却为线形的。

而链表每个元素的物理位置是任意的，通过指针串接起来。它的访问时间非常长，但插入删除速度却很快。而且由于需要额外的空间记录元素的前后关系，所以占用的内存也略大于数组。

这两种线性表，在 STL 中分别用 vector 和 list 来实现，实际上，我们可以根据实际情况对常规操作方法加以改进。比如数组中删除数据，就可以采用做记号的方法，而不是每次删除元素都将后面的数据进行移动。等删除的元素累计到一定数量再一次删除。链表则可以额外记录并

维护一些中间元素的位置来帮助定位。这些方法之所以没有加入标准，是因为它们带来好处的同时，也在另一方面增加了负担。必须由程序员来判断需要采取的策略。

数组和链表的结合

我在多年前，曾经构思了一个结合数组和链表优点的方案，并编程实现，后来用到一些参与开发的游戏中。

在很多问题中，我们需要保存一组有序（这里指比较意义上的有序，即数据从小到大或从大到小）的数据，这组数据需要高速的检索，并增删频繁，而且数据量很大。

解决方法有很多，例如可以使用树结构或者 hash 表的方式来保存。但是我选择了线性表，并试图结合数组和链表的优点。

当时我是这样实现的：创建一个固定大小的数组，作为链表的一个节点。检索数据的时候，检索链表确定数据在哪个节点上，然后在节点内做二分查找。增加新元素的时候，如果每个节点的数组满，就增加一个新节点，并复制老节点的一半元素去新节点上，然后再添加。

这样做的好处是，每次添加新元素时，需要移动的数据量控制在每个节点的数组大小之内。而检索数据变成二级检索，在节点内的数组上是二分查找，速度很快。整个表的冗余数据大约占 1/4 左右，如果单个元素本身尺寸不大，就比用二叉树来记录数据要节省空间得多。

这并非一个通用的解决方案，甚至我也不能证明它在大多数情况是否都是有效的。举这个例子只是说明，我们程序员的思维不能仅限于书本上的知识。

2. 堆栈、队列和串

堆栈和队列是两种特殊的线形表。

堆栈这个概念在现在的程序设计中必不可少。准确地说，在计算机术语中，堆和栈是两个不大相同的概念。这里提到的堆栈指的是**栈（stack）**。这种数据结构，即限制一个线性表的操作，只能从一头进出，先进入的数据后出来。它广泛地被用于类 C 的语言中的函数调用机制。我们在做深度优先搜索求解问题时也需要用到它。

队列（queue）和栈相反，它采取的是先进先出的策略。一般我们在实现的时候让数据从线性表的一头进入，从另一头出去。这个数据结构用来保持元素的先后顺序，被广泛用在消息通讯中，也可以用于广度搜索的算法。

由于效率因素，这两种数据结构大多使用数组来实现。C 语言中，原始的数组必须固定初始的尺寸，所以数组不适合用在初始大小的算法中，在C++中可以使用能够调整尺寸的 vector 或者 deque。事实上，C++的 STL 中，已经用 deque 实现了模板 queue 和 stack。

队列的应用非常广泛，在很多算法中，某个步骤得到了不止一个结果，而下一步骤只能处理其中一个结果时，我们都将其他结果放进一个队列中。经典的问题是图形染色问题。这个问题一般的解决方案是在图片中找到一个染色的起点，围绕这个点把可以通达的区域都染上颜色，逐步扩大范围。被染过色的区域的边界信息被放进一个队列中，依次处理。染色问题在游戏中很容易碰到，比如计算 NPC 视野，或是寻路算法就是在它的基础上完成的。

在某些情况下，我们不严格遵守先进先出的规则，而是根据数据本身的特性来决定出顺序。这样的数据结构往往也借用队列这个名字。典型的例子是 sgi 的 STL 版本中实现的**优先队列（priority queue）**。这个队列中，每个元素都有一个优先级，出队列的时候，永远是优先级最小的元素优先。优先队列并非上面提到的队列，通常也不用线性表来实现。

串（string）是一个符号的序列。这些符号一般是一些字符，所以，它通常也被称为字符串。字符串通常用字符数组来实现，这是一种最经济的实现方法。在 C 语言和诸多其他的系统中，我们把字节 0 看做一个特殊的字符，以标记字符串的结束。其实这并不是一个最合理的方案。这样，0 就被看成了一种特例，某些二进制数据流就不能正确表示了。许多软件的安全问题，也正来自于一开始不知道串的长度而产生的数据溢出问题。更合理的方案是用长度加字符数组的方式来表示。

串本质上也是一种线形表，但和堆栈及队列不同，串的操作大多不是以里面一个元素为单位，而重点在于串的全部或一部分的比较、连接、匹配等。这些操作的实现算法也相对复杂得多。

其中用得最多的是串的匹配算法，比如 Knuth-Morris-Paratt（KMP），Rabin-Karp 等。现在计算机处理的问题中，极大量的工作都是用在串处理上。

3. 树、二叉树及其他

树结构是一种非常常用的数据结构，但却因为其实现比现线形结构复杂，所以大多数游戏程序员使用它时都非常慎重，甚至回避使用。通常的代码中，树结构一般都存在于大的程序框架上，是整个软件的支柱。

树结构的复杂性和多样性，导致C++的标准库中都没有一个已有的实现。每次用它的时候，都需要专门的实现。

关于树的讨论，可以延展出非常多的话题，深入研究下去可以超过一本书的容量，这里仅仅做一个简单的介绍。更多的内容，可以参阅有关专业书籍，或者在互联网上搜索。

■ 树（tree）

简单地描述，树就是对有层次的数据集的一种组织方式。树中的每个数据节点都有或没有它所下属的数据集。而除了根节点是整棵树的根源外，每个数据节点都有一个唯一的父亲。

软件的图形用户界面，通常用树来组织这些数据。游戏中的对象管理，有时候也用到树来解决一些管理上的层次问题。复杂的数据结构往往就是程序复杂的根源。

■ 二叉树（binary tree）

二叉树严格说来并不算树的一种，因为二叉树的子树有明确的左右之分。我们可以利用这个特性把一棵树用二叉树的形式来表示和实现，即让左子树指向自己的一个儿子，而让右子树指向自己的下一个兄弟。

二叉树在表达式计算和数据压缩，以及排序查找方面都有很多的用途。C++的 STL 库中，就是用一个红黑树（一种特定形式的二叉树）来实现 map 和 set 的。

■ 四叉树（quadtree）及八叉树（octree）

这些多用于空间状态的划分，四叉树用于平面划分，而八叉树用于三维空间。值得一提的是，这里所说的空间，不仅仅局限于游戏里的场景空间。例如，计算调色盘的算法，就有利用八叉树对处于 RGB 色彩

C 语言的库函数中提供了常见的字符串处理的函数，但有时候不太够用。现在用得更多的是**正则表达式**，它可以提供更为广泛的串匹配、提取和替换操作。在众多的脚本语言中，正则表达式是内建的实现。尤其是 Perl 这样的脚本语言，简直是依之而存在。可惜它在很长时间里不为游戏程序员所重视，这点读者应该稍加关注。

空间中的诸多颜色中选出有限数量的调色盘。

■ 图（graph）

相比树来说，图要更复杂一些。图中的每个节点并没有父子关系，图是纯粹的点和边的集合。节点甚至允许有通向自己的边，但这并不常见，通常也不会使用。

节点和节点之间允许加上一些与它相关的数，称为权（weight）。带权的图一般叫做网络。节点和节点之间可以是无方向的连接，也可以是有向连接。这又会有不同的分支。

图论就是专门研究图的学问，它并不简单，绝非三言两语能够概括。但绝对值得游戏开发人员去学习。现今的网络游戏中，多服务器的设计，或者三维游戏中对于大场景的描述等，都是近年来游戏开发圈子里热门的研究题目。

■ 映射表

C++ STL 的被人用得最多的容器恐怕就是 std::map 了。它也是 STL 中实现最复杂的容器。

这就是一个映射表，可以将键值（key）和内容对应起来。典型的应用就是用字符串而不是数字做数组的下标。这样运用范围更为广泛。

C++标准中是用红黑树来实现的，这是一种准标准的实现方法。空间效率和时间效率在一个平衡点。

但是，实际更常用的方法是采用哈希表（Hash Table）。可惜的是，hash_map 现在还并不是 C++标准。虽然 sgi 版的 STL 实现了它，但使用的时候应该适当考虑一下非标准这个问题。

其实，在自己的程序中实现映射表不仅仅有以上两种方案，应当灵活处理之。如果表本身是一次创建好，而对查询效率有很高要求时，则最好也是最简洁的方法，是把数据放进线形数组，排序后对分查找。

2.2 搜索算法

我曾经天真地认为，计算机需要解决的一切问题都可以用穷举搜索的方法来解决。只要找到合适的搜索算法，配合很强的硬件，在足够的时间内就可以获得答案。

这种想法当然是错误的，避开 NP 问题不谈，在游戏中，我们就会遇到大量需要实时反应的问题，不可能用大量搜索来解决。但是不可否认，搜索依然是计算机解决问题最有利的利器。

因为任何问题都可以转化为若干个状态的连续集。我们将问题在某一时刻的情况做一个数学的描述后，再归纳出问题从一个状态到另一个状态的转移操作。由计算机模拟这些状态转移的过程，遍历问题所有状态，找到从起始状态到目标状态的可行路径，就找到了问题的解。

2.2.1 地图寻路问题

不得不说，提到搜索，许多的游戏程序员会把它与地图寻路问题混为一谈。甚至，把 A*，A，IDA*等等算法全部跟寻路算法划到一起。其实，游戏中经常用到的地图寻路算法，只是一个常见的、很容易把状态空间数字化反映出来的问题。非常适合用各种搜索算法来解决。诸如 A* 这样的算法绝对不会为了单单解决地图寻路问题而存在，而应该看成在广义上的状态空间地图上找到最佳解路径的一种方法。

通常地图都是用图的形式来描述的，或许大多数人喜欢采用二维数组来保存游戏中的地图，这看起来跟数据结构中提到的图结构相距甚远。但实际上，二维数组描述的地图，可以看成一种特殊的图。数组中的每个单元，即单个格子都是一个顶点，而边信息被隐藏在结构之中。每个格子有默认的四条上下左右连通的边。

不少对传统基于格子的寻路算法大多是改进地图状态空间的描述方法，把数据量减少。例如，设定路点（waypoint）、将路点连通，或者用四叉树来描述地图。但其基本的搜索算法都是类似的。

先来看最基本的**广度优先搜索**来解决寻路问题。

我们从起点开始，向其连通的顶点扩散搜索可以到达的新的位置，然后记录下每个新到达位置到起点的路径长度。如果地图是一个均匀的无权值的网格，那么就是每个格子路径长度加一。有时候我们需要有地形，比如山地比平原要难走一些。这样，格子之间的通路权重就不同，甚至于有些单向通道，例如可以从悬崖上跳下，但无法从悬崖底爬上去，这些都是很容易实现的。

当然，地图上的每个点都可以有不同的路径通达，这样，每次广度搜索扩展到一个新的顶点，都需要和历史上曾经达到这点的长度进行比较。路径较长的被停止扩展。这种操作，叫做**剪枝**。最终，只要目标点和起始点是可以通达的，那么随着搜索节点的不断扩展，总能找到一条最短路径。

很明显，广度优先算法需要消耗大量的内存。需要一个很大的队列来保留等待扩展的节点。被扩展的节点的数量随着起始点到目标点的距离而迅速增加。

在算法设计上，我们永远可以找到空间换时间和时间换空间的方法。**深度优先搜索**相比广度优先搜索，就可以消耗最少的内存。深度优先搜索去寻找路径，就好像在模拟人来走迷宫。一个人在迷宫中永远无法分身，他只能采取碰壁回头的方法。我们把所有走回经过的节点的情况都看成碰壁，让寻路点回头选择下一条分支，最终肯定能找到通向目标点的路径（如果理论上存在的话）或者最终返回原点。

深度优先算法无法找到最短路径，作为改进，我们一般使用**迭代加深搜索（Iterative Deepening）**。既定一个搜索树的最大深度，做深度优先搜索，如果无法找到解，就加大最大深度，重新来。这就是典型的时间换空间的策略。

以上提到的方法都属于盲目搜索策略，实现虽然简单，但在地图寻路中并不实用。

游戏的地图大多不是迷宫，如果是人来寻找路径，则往往可以根据方向等信息快速地找出一条路来，而不必缓慢地从起始点逐层展开。如果由计算机来模拟这种利用额外信息来快速找到解的算法，则统一被归

纳成**启发式搜索**。

最为游戏程序员所熟知的是**A*算法**。

A* 只是一种基本的搜索算法，是诸多算法基本功里的一项，是每个程序员在学习编程的工程中必须练习的习题之一。我们为游戏写程序，不比写其他程序更特殊。在学习编程时，应该花最多的时间去做一些看似对未来工作毫无实用价值的习题，以提高自己对算法的驾驭能力，日后处理实际问题能得心应手。

A* 算法是一定能保证找到最优解的。它最坏的情况，也只是占用的时间和空间和广度优先搜索相当，而绝大多数情况是要明显地好。所以在游戏寻路中用得最多。

设计A*算法在于设计一个估价函数，即在搜索时扩展到某个点的时候，评估这个点可能到目标的路径长度。根据评估值和起始点到这个点已知的路径长度之和，对当前待扩展的节点排序，永远先扩展最有可能更快达到目标点的节点。这样，可以在找到解之前，极大减少搜索节点扩展的量。

在设计A*算法的估价函数时，必须保证估价函数的值一定要小于实际路径的长度。保证了这一点，就保证了搜索结果一定是最优的（数学证明并不复杂，这里从略）。对于寻路问题，最简单的估价函数就是当前点到目标点的直线距离，因为不可能有比直线距离更短的路径，所以保证了估价函数的要求。

在游戏中实际使用时，往往为了提高估价函数的效率（估价函数值越接近实际路径长度，效率越高），我们往往不能完全遵守上面的准则。如果设计的估价函数值有可能大于实际存在的最短路径的长度，那我们就有可能求不到最优解，这有时是需要的。在游戏的实时环境下，速度比最优解更重要。那么，这种采用不遵从上述规定的估价函数的启发式搜索，就被称为**A 算法**。

典型的例子是，我们在地图中把障碍复杂的地段的估价权值提高，认为通过障碍的代价比绕过障碍之地要高；这样采用这种估价函数，寻路过程就会趋向于避开大量障碍之地，而避免陷入地形复杂的情况的搜索。

我时常能收到我的网站的读者来信，其中有不少朋友向我请教A* 算法寻路的问题。好像寻路算法变成了游戏编程中的一个专门课题，需要专门研究。掌握 A* 算法也成了游戏程序员的一项资本。

也有许多人来信抱怨我在网站上公布的一段实现了基本的 A* 算法寻路的代码无法通过他们的编译，并希望我提供一份可以在他们的编译环境中运行的程序。实际上，影响编译的只是一些与平台有关的输出函数，并非算法本身。

这些朋友，大多是一些在校学生，期望通过互联网以一种无需思考和学习的途径去完成作业。对于此，通常我只能无可奈何地回信：掌握算法，对于已实现的程序，应该是通过阅读来理解它，然后必须亲手重新实现出来，而不是复制过来编译运行。只想看到最后运行的结果，对于学习毫无意义。

A* 算法提高了广度优先搜索的效率，尤其是时间效率。但是，空间消耗依然是一个大问题。在游戏中，尤其是网络游戏中，引擎可能会面对大量的路径搜索请求。最简单的方法就是限制 A*算法的空间占用量，即可扩展节点的数目。一旦溢出，就在已经找到的节点中，求出最接近目标点的一个。把它作为新的起始点，放回路径搜索请求队列中。这种方法大多数情况是有效的，虽然有时候会出现自动寻路的角色塞在某处死角出不来的情况，但可以解决单次寻路过程过长，堵塞整个线程的情况。尤其是，当起始点和目标点原本就不通的情况下，无论 A* 搜索还是盲目搜索，都会扩展所有可能的节点，消耗大量的时间。这样，这种限制就显得非常重要了。

一旦我们必须找到路径，就可以考虑结合 A*算法和迭代加深算法，这就是 **IDA*算法**。把搜索过程改为了迭代加深，而在搜索过程中忽略掉估价值大于迭代上限的结点，使速度大大地加快。而且在搜索过程中，不需要对待扩展节点排序，使用更加简单。

双向搜索

我不止一次地在和人讨论寻路算法的时候都听到过双向搜索的想法。看起来，双向搜索将是单向搜索的一种加强版，而实者不然。利用双向搜索提高效率的可能性很小，只在很少的地方有效。一般，它只是简单的交替扩展节点的过程，如果需要求最优解，那么两头扩展的节点首次交汇的地方的解无法判断其是最优解。而双向搜索多只用于求可行解。但一旦两头的搜索过程擦肩而过时，找到解的速度会变得更慢。

启发式双向搜索其实价值不大。如果简单地用两个 A*算法从两头向中间搜索，则两边的第一次交汇点并不是最优解。为了保证解最优，就必须对两边扩展的每个节点之间一一做评估，这样，估价函数计算量太大，反而没有太大的意义。

在目标点固定的情况下，我们可以用预运算的方法提高双向搜索的效率。这样可以预运算扩展目标点，然后把扩展过的节点记录在一张表里面，可以提高以后正向搜索的效率。这种方法被称为周界搜索（perimeter search）。

2.2.2 博弈问题

博弈问题可能是计算机游戏中最常见的一种 AI 问题了，在回合制游戏中实现 AI 时用得很多（当然，许多游戏设计者回避了这些问题，改由一些简单的规则来表现 AI。游戏的计算机控制角色普遍智能不高，这也是经常为广大玩家所诟病的）。

其实，传统的博弈问题解决方法就是用启发式搜索解决的。我们可以把棋局的每种状态看成状态空间中的诸个节点。这样，一盘棋的所有变化，就可以用一棵博弈树来表示。树的每一层即是棋局的每一步的所有变化。

每个节点表示了棋局的一种特定局面。我们用程序给每个局面进行评估，对我方越有利，评估值就越高。如果某个局面我方必胜，那么评估值就是∞；反之，如果某个局面我方已败，评估值就是-∞。

自然，如果能准确地估算出当前每个子局面的评估值，我们就可以选择评估值最大的方案行棋。可是，单靠对盘面本身的评价很难找到公式做出准确的评价。这样，我们就需要借助搜索。

当轮到我方行动的回合时，这个时候的局面评估值就等于所有可以演化下去的子局面的评估值中的最大值（我们应该选择最佳的方案）。

而如果这个回合是敌方行动时，局面的评估值就等于所有子局面的评估值中最小的一个了（敌人会选择对你最不利的招数）。

假设我们有足够的时间和空间将整个博弈树展开，而又不存在和局的情况下，那么所有节点的评估值要么就是+∞，要么就是-∞。我们只需要选择+∞的分支下即可。

可惜，能够展开整棵树的情况不多（井字棋算一个☺），一般我们都会设置一个搜索深度，这个深度当然也可以是动态的。当逐层展开博弈树到达一定深度后，就按一定的公式直接对棋局打分。这样回溯回去后，就能估算出每步每个子状态的分数，来做出较优的决策。

在这个搜索过程中，如果完全展开每个局面的子局面，运算量必然很大。在博弈数搜索中，通常会使用 **Alpha-Beta 剪枝**。这是一种剪枝算法，使得在搜索过程中途就可以停止搜索展开部分的节点。这种剪枝

的依据就是博弈树的每层节点的估价值要么等于子节点的最大值，要么等于最小值。一旦我们发现即将扩展的节点不可能比前一个大(或者小)，就可以停止扩展。以下棋为例简而言之就是，如果我下这步棋时，对手有一种应对方式比我用另一种下法时对手的所有应对方式都糟糕，则就不用考虑下去了，直接否定这种下法。

基于 Alpha-Beta 剪枝的博弈树搜索方法是基于深度优先的策略的。我们依然可以把启发式的广度优先算法用进来。这样，每次根据对所有子局面的主观评价值（不通过搜索得到的评估值）排序，总是先展开评估值高的节点。在不考虑空间占用的情况下，往往比 Alpha-Beta 剪枝更加有效。这种算法被称为 **SSS*算法**。

SSS*算法的改进型是 **MemSS***，它为 SSS*算法使用的内存空间做出了上限的控制，一旦溢出就丢掉一些评估值低的节点（但保留评估值本身）。实践证明，这个方法很有效。

2.2.3 更为广泛的运用

搜索算法运用的范围很广，对于大多数问题都很有效，只是本身具有很高的灵活性，不太容易用好。

在游戏程序开发中，其实不光是为了解决实时性问题，有时候，还需要在开发过程中对一些数据进行预处理，都是很有可能需要用到合适的搜索算法来解决问题的。比如最近，我就碰到一个三维游戏的贴图优化问题，有许多张不同形状的小贴图需要最高利用率地填塞进若干张 256×256 的规则贴图中。怎么排放就是一个头痛的问题了。它就可以利用一些搜索技巧来借用程序自动尝试排列。

千万不要以为只有寻路才会用到搜索算法。把算法的使用和特定的问题等同起来，对于游戏程序员，将是很危险的。

2.3　智能算法

下面谈到的一些算法多出现在一些比较新的讲解人工智能的书籍和网站上。这些算法的起源都是因为一些复杂问题的存在（比如一系列的 NP 难度问题），用搜索穷举的方法很难在有限时间内找到解，所以人们逐渐地利用模拟一些自然现象的方式而不是简单穷举的方法来求解。

因为实际用到的次数很少，我对这些算法了解不是特别多。但也曾经尝试用其中一些方法解决碰到的小问题，感觉颇为有趣。或许这些会是未来游戏 AI 实现的一大方向呢。在这里跟大家分享一下，期望引起部分读者的兴趣以深加研究。书里这方面的内容甚为粗浅，若为行家所见，估计要贻笑大方了。好在其内容均是根据我个人理解所述，提到的内容基本上都包含个人的实践经验，非从其他专业书本上摘抄而来。如有错误，请行家帮我指出。

2.3.1　遗传算法（Genetic Algorithm）

从名字可以看出，这是某位前辈从生物自然法则中悟到的算法。物竞天择，适者生存，这是自然规律。上天用简单的方法进化了物种，进化的目的是得到更优秀的基因，而进化的过程，对于单体来说却是无序的。上天安排物种不断优化的过程就不需要逐个地搜索试探，也不需要刻意发展较好的单体。只需要遵循最基本的几条规则就够了。如果计算机能模拟这个过程，即使是解决非常复杂的问题，程序也可以变得很简单，甚至很通用。20 世纪 60 年代的时候，Michigan 大学的 John Holland 提出了这个遗传算法。

简而言之，就是先把需要解决的问题分解成一个个小的元素，一组元素就构成了对这个问题的一个可能解。如把某一个解看做一只小白鼠的话，它就具有了一系列问题的子单元的基因。

让我们先随机制造一大批的小白鼠，第一代小白鼠因为基因各异，相差都非常远。但是这群小白鼠的数量远远低于所有基因的全部组合数。

可以认为，小白鼠的分布是均匀分布在解空间中的。那么总有一些白鼠更接近最终我们需要的解（品质更优秀）。我们根据它们的基因优秀程度为概率来挑选出一些繁殖下一代。在基因算法中，每一代的数量是一样多的，但是，基因好的白鼠被挑出甚至重复挑出的几率大一点，如果有1000只小白鼠，我们挑1000次，只要有白鼠被挑出多于一次，那么就必然有白鼠落选，那么它的基因就会消失。

注意，这个挑选过程是随机的。只是基因的好坏决定了被挑出来的可能性（通常我们会对基因打分，然后根据得分来加权）。并非最差的老鼠一定落选，这一定程度上反应了生物学的原理，父母很糟糕并不说明其后代也同样糟糕，虽然那种可能性比较大。

一部分的白老鼠采取了无性繁殖的方式生成了下一代，保留了所有前辈的基因。而另一部分白老鼠采取了两两杂交的策略，就是说，选取两只白鼠，对它们的基因做一个交叉，生成两只杂交过的后代。根据经验，这个比例可以选取为整体的60%。这是生物界杂交优势的体现。一般说来，杂交的动物都比它们的先辈要优秀，例如杂交水稻的产量提高，又比如骡子在体力、耐力方面都大大超过了马和驴（当然也有例外，狗似乎是越纯种的越值钱☺）。

在新的一代中，我们需要对极少的单体做一些基因的变异，其方法是随机替换掉染色体上的一个基因。变异的结果大多是把优秀的单体变得很糟糕，而失去繁衍到下一代的机会。但由于优秀的群体相对较多，可以允许这样的损失；变异的结果是可以让某个单体迅速地跳离当前的群体的境界，突然变得优秀很多。

通常，变异的概率都选得非常得小，在0.3%左右。过高的变异率会导致种群的不稳定；而过低的变异率则依赖于大量的种群数量和导致长时间的繁衍。

遗传算法的程序实现非常简单，其基本框架都是一样的。用C++实现绝对不超过200行代码。感兴趣的朋友可以从互联网上找到大量的相关资料。我曾经根据上述思路实现过一个程序解决某种特定情况下求最佳调色盘的问题，也仅仅花了几个小时，且效果不错。

就我个人看来，使用遗传算法解决问题，其难点在于基因的计算机

编码方法。

某些网文提到，可以直接用解的二进制数据的逐位来表述基因，无论问题如何，都用解的二进制数据来充当遗传算法中的染色体。这显然是不实用的。

因为毫无规律的二进制数据，实际上对应了极大的解空间。这样，为了解决问题，我们必须使用相当大的种群和非常长的时间迭代才可能找到比较好的解。而且，毫无规律的基因设计方案，诸如直接用一个解数据的每个字节作为一个基因，使得杂交优势非常不明显。

好的编码方式是，将问题分成尽量而不完全独立的几个部分，每个部分列举出所有可能的情况，一一做成连续数字编码作为基因。这样解空间本身被缩小了，而且由于基因本身的独立性，解只取决于基因的组合方式，这导致了杂交优势变得非常明显。优秀的个体必然携带优秀的基因片段，成组地交换基因片段，很容易衍变成更优的个体。

2.3.2 模拟退火算法（Simulated Annealing）

退火是一种金属热处理方法，当把金属缓慢加热到一定温度，保持足够时间，然后在一定速度下逐渐冷却，可以使金属的化学成分均匀，改善其塑性和韧性等。

固体在足够高温时，分子的内能非常大，分子的布朗运动很强。而当温度缓慢下降，在每个温度有时间停留的时候，类似的分子会聚集到一个范围，运动范围逐步变小。分子热运动慢慢减弱，逐渐趋于有序，最终达到稳定。

如果将这种现象模拟出来，用在解空间里寻找我们需要的解，就好比在全局的空间里先无序地运动，以找到解可能所在的范围，并逐步缩小这个范围，直到最后把解确定在某个点上。

整个模拟过程如下。

首先，我们还是需要将问题拆分成细小的元素，正如遗传算法里的那样。然后，我们需要设计一个函数，可以不断地调整这些元素，正如一个做布朗运动的分子，可以在解空间里随机地改变一点位置。

模拟退火的求解过程一开始更像一个无序的搜索过程。先定一个充分大的温度 T 和一个初始解 S。从这个起点开始不断地改变 S，每次得到一个新的解 S'，对新解计算评估值 $C(S')$。将这个评估值和上一个解的评估值 $C(S)$相比较得到 $\Delta t'=C(S')-C(S)$。如果新的解比上一个好($\Delta t'>0$)，就接受这个新的解，否则就按一定的概率来接受。这个概率和温度 T 以及解的波动范围有关，即 $\exp(-\Delta t'/T)$。

从这里可以看出，当温度很高的时候，几乎总是以很大概率接受新的解。也就是说，解“分子”在解空间里的移动是随机的，它只是以一定概率比较接近偏优的解。这样做的好处是，如果解空间里有多个不同位置的比较好的解，则算法不会偏向某一个，而导致搜索跳不出局部最优。

在固定的温度 T 下，迭代若干次后，解所在的位置就会落在一些可能是最终解的周围了。

然后我们稍微降低温度，让解“分子”的活动范围稍微减少，即偏离最优解位置的概率减少。继续做如上操作，直到 T 减到 0。

$T=0$ 时是一个特例，迭代过程中没有了概率因素。每次迭代都只选择向更优秀的解的方向移动，这有点类似一个简单搜索的过程。

模拟退火算法具有渐近收敛性，整个模拟退火的过程已在理论上被证明是一种以一定概率收敛于全局最优解的全局优化算法。它是可行的，但算法的参数选择对实际效果和速度有很大关系，需要根据实际情况进行调整。

解的初始位置是不重要的，可以随机产生。但初始温度 T 很重要，T 越高，越能够找到最优解，但是迭代次数太多，会花费大量的计算时间。

退火速度，即每个温度下的迭代次数也很重要，一定要做得充分，才能让当前解的位置符合要求。

温度控制的问题也很复杂，一般我们都会采用简单的按比例缓慢下降的方法。

最后，和上面提到的遗传算法一样，怎样定义解的结构是至关重要的。定义得好，就可以缩小解空间，用不长的迭代时间找到最优解。定义得不好，在有限的时间同样无法找到需要的答案。和遗传算法一样，

算法本身对问题是没有特殊化处理的。面对任何问题都一样，从组成解的所有元素的全排列中找到最合适的排列。用评估值和概率来导向，逐步修正随机排列达到最优。它们对解决时间复杂度为 $O(n!)$的组合问题是很有效的。

2.3.3　禁忌搜索（Tabu Search）

在解空间里寻找最优解的时候，好比在一个平面寻找一个最高点。假设整个平面都是连续的，修改解的一小部分就看做在平面上移动一点位置。其高度会稍微高一点或者低一点。

传统的启发式搜索方法都是永远朝着相对高的地方走的，如果要找到最高点，则要么走过平面的每个点，要么在有限时间内只找到局部的高点。

模拟退火算法，是用一定的概率允许解向更低劣的方向移动，以避免纠缠于局部高点的搜索。而禁忌搜索更像是对局部领域搜索的一种扩展，是一种全局逐步寻优算法。

禁忌搜索的思想在上世纪八十年代被提出，其主导思想是模拟人的记忆过程。从某个点开始，向临近区域扩展解。经过的地方一概进入禁忌区域，防止在短期搜索回来，造成循环搜索。每个禁忌区域有个寿命，超过若干步骤后，就解除禁忌。这个步骤数叫做 Tabu Length，禁忌长度。禁忌长度过大会导致计算量增加，而过小则容易陷入循环搜索。

已求到的最优解，是可以无视这个禁忌规则的，优先多扩展一次，以期能在附近找到更优解。

记录禁忌区域的禁忌表就是模拟人的记忆，记住一段时间搜索经过的路，不使搜索过程循环。它采取队列的先入先出原则，但是，一旦发现一个历史最优解，就不会放它出去，而是重新记录下来，并从这个点开始重新找另一路探索（之所以可以避免走老路，是因为老路被记忆禁忌起来了）。禁忌表越大，禁忌搜索的效果越好，但计算量也变得很大。反之，禁忌表太小，也很容易陷入局部最优解的搜索。

做个形象点的比喻，一群蚂蚁在一块区域寻找最高点。它们任意选

了一个起点，一开始就保持队伍胡乱地向周围爬。蚂蚁的队伍遵循一个原则，就是不能回到最近经过的位置。这点只要看看是否会碰到队伍里其他蚂蚁，如果碰到，再判断碰到的蚂蚁跟队首的距离。当整个蚂蚁队伍都离开了起点后，就开始监督，保证离开的那个点是否比原有记录的最高点要高。一旦高于记录点，就在这个点驻扎起来，并从此为起点向旁边探索。周而复始，保证队伍离开的点一定不会是比已经找到的高点更高的位置。

这样，整个蚂蚁群体不会因为发现某个高点而在附近纠缠，也不会忘记曾经见过的最高点。最终，达到整个区域的最高处。

2.3.4 人工神经网络（Artificial Neural Network）

关于人工智能的模拟，计算机软件设计者们通常从两个方面去探索。传统的方法都是从人的心理角度去模拟，这也最为人所知。简单的心理模拟，体现在游戏设计中，就成了 NPC 角色对玩家动作的固定反应。这也是游戏中的人工智能被玩家指责为人工白痴的原因之一。更复杂的一些，反映在启发式搜索上面。直接被玩家所见的就是角色寻路的过程。如果电脑角色可以准确无误地延着最短路径走到目的地，就被称赞为游戏寻路做得好，高 AI 了。目前，似乎很少有游戏去挑战自然语言理解，自学习的对战战术这些真正有难度的 AI 课题。

其实，人工智能的研究还有另一个方向，即人脑的生理模拟。

人工神经网络就是对人脑系统最底层结构的模拟。它在五十多年前被提出时，更多的是考虑用电子线路来实现。它迥异于冯·诺依曼体系的计算机，是非精确数值逻辑的，允许被高度并行，并且有天生自然自学习功能的系统。现在，我们已经可以用计算机程序来模拟它。当问题通过受训练的人工神经网络而得到结果时，我们无法用数学逻辑推理来逐个神经元的分析解释为什么可以解决问题。

问题解答的正确率和速度已经和软件的实现联系不大，关键在于它自身的不断学习。这种思想已经和冯·诺依曼模型很不一样。

生物的神经网络，由海量的神经元互相连接起来。生物电流在里面传播。神经元之间的连接强度决定了信号传递的强弱，而这个连接强度是可以随训练而改变的（从局部看来，人脑对一件事情的反应的训练跟

用骨头训练狗没有多大区别）。信号可以起刺激作用，也可以起抑制作用，一个神经元接受的信号的累积效果决定了该神经元的状态。每个神经元都可以有一个“阈值”。

人工神经网络也模拟了上述的特征。我们可以用程序实现一个个神经元组建出电子版的网络。人工神经网络一般建设成一个多层的非线形有向图，有一层直接面对输入信号，另一层输出结果，中间作为隐藏层并不对外所见。

学习和训练阶段，信号正向传播或反向反馈刺激和改变神经元间的权值，使网络趋向于解决特定的问题。足够的训练后，解决问题需要的信息会被分布存放在几乎整个网络中，所以网络局部受损还是可以工作的。但是，当它完成学习后，继续学习新的东西也有可能破坏原来已经学会的知识。

在学习过程中，单个神经元的结构并不会改变，这个个体本身的特性也和要解决的问题没有直接联系。学习的作用仅仅根据神经元之间刺激或抑止的关系，改变它们的作用强度。这些神经元之间连接通路的权值就包含了通过学习得到的全部信息。

我个人仅仅读了一些人工神经网络的相关书籍，并没有尝试过运用它们到自己开发的游戏中。所以并没有什么深刻理解，这里就避免详细介绍和讨论。这是一个迷人的课题，希望有更多的人研究它。

其实，这些智能算法，本质上并不复杂，都只是提供一个思路而已。举凡大自然中存在的事物，刨根问底，又有哪点是构造在复杂的原理之上的呢？耐下性子，加上一些扎实的编程功底，几个小时就能把它们一一实现。充斥着网络的，求 XX 算法源码帖子的作者难道不该为自己写不出几百行代码而汗颜吗？

自己动手实现一下，真正了解这些方法，慢慢地就可以真正运用它们到实际中去。我们程序员平时脑力总是集中在怎么描述问题，以适应某种特定算法，那么了解更多的算法会让我们更快地解决棘手的问题。

去年我去了许多大学招聘应届毕业生，其中有一站招聘时，有学生迟到，问其缘故。答曰，参加人工智能课程的考试。当我饶有兴趣地和几个同学讨论起这方面的问题时，大为失望。

虽然许多人可以将诸如遗传算法，或是神经网络这些时髦的词挂在嘴边如数家珍。但问及对这些的想法时，却毫无建树。除了书本上的例子，没有人告诉我它们具体可以用在什么地方。包括以此为研究方向的硕士研究生，也支支吾吾说不清算法实现的细节，表示有些东西要回去翻书。

对于像遗传算法中非常重要的编码问题，都被人忽略。而只能像背书般地讲述一些生硬的算法过程。

2.4 优化

优化这个词，总是会反复刺激程序员的神经。尤其是游戏程序员，他们都希望自己的程序跑得更快。而代码的优化，永远在各种层次上进行。在本书里，后面还有许多章节谈到它们。这里，主要讲述的是算法层面的优化。

稍有经验的程序员都能意识到算法优化对程序性能的重要性。经过近二十年的编程经历，我总结下来，无非是以下四点：

（1）数学方法的改进。

（2）预运算来节省时间（空间换时间来避免重复运算）或是重复运算来节省空间。

（3）简化算法求得近似解来取代精确解（或最优解）

（4）改进数据组织方式，用更少的操作处理更多的数据，甚至避免冗余数据的处理。

逐点举例说明是很难的，因为每次成功的优化都会是各方面的。

下面我用一些真实的故事来给大家一些启发。

2.4.1 质数问题

记得上初中的时候，我顺着一本习题集编写程序。那都是些不太难的题目，基本上手不需要离开键盘就可以不停地做下去。当年我完成任务后，目光回到其中一题，求出小于等于 N 的质数的个数。

按质数的定义：一个整数只能被它自己和 1 整除的话，它就是一个质数。按照定义，我们只需要检验小于等于 N 的所有整数是否质数，然后对质数做一次计数就够了。不过我想到一些有趣的改进方法，所以就和教室里一个好友下了战书，看谁的程序能算得更快！

这是我第一次面对程序优化的挑战，最初我的策略是：要检验一个数字 N 是否质数，是不需要依次去试除小于它的所有整数的，实际上，把根号 N 以下的数字试一遍就够了。其次，除了 2 以外的偶数可以立刻

排除掉，当然也不用去试除偶数。这样，运算就可以少很多。当我用这个思路重写了程序后，我的对手也做到了，程序的速度不分伯仲。

然后可以想到的改进的方案是，我们甚至可以只去除根号 N 之内的质数，那么只需要把已经运算过的质数记录下来，后面测试质数的时候，用前面找到的质数去试就够了。可惜做到这一点后，依然没有赢得比赛。那么还有别的方法吗？如果我们能直接排除掉偶数，也就是 2 的倍数，有没有什么方法可以一次排除 3 的倍数呢？

3 的倍数的合数在整数集里也占有很大的分量呢。这个思考方向很容易想到，但是，具体的方案却不那么容易得到。在经过长时间的思考后，我找到了方法：只需要建立一个尺寸为 6 的数组，依次把偶数项和第三项填入 1 ，其他项填为 0，在依次校验质数时，以 6 为一个小循环，对应数组的数字为 1 的就直接排除掉。我把这个小数组叫做筛子，当我们把筛子做得大一些后，就可以迅速排除掉 2,3,5,7,11... 的倍数的合数，从而减少了计算机计算除法的次数。

最后让我取得压倒性胜利的却不是这个算法上的改进，而是我想到了一个近乎作弊的方法。我记录下每次运行的结果，并在内存允许的范围内，把能求到的质数也记录了下来。我的程序运算结束后并不退出，算了几次后越来越快，因为更多时候只用去搜索以前的结果了，甚至我可以用二分查找来加速。就算是很大的 N，也仅需要从曾经算过的较小的 N 的基础上算下去。

当内存不够，不足以放下所有运算得到的质数时，可以放一些关键点上的 N 对应的质数的个数（记得我们题目的要求仅仅是求出质数的个数，而不是列出所有的质数么），更多时候，程序再运算时只需要验证某个区间的整数是否是质数了。

事过多年，我把这个题目当做招聘笔试题的时候。非常遗憾，即使做出很多提示，考场上也极少有人可以给出满意的答案。

现场考试紧张可能是一个方面，但是书本上知识也带来了太多的桎梏。身为游戏程序员是要有些想像力的，要善于用各种方法解决问题。如果一切都可以从前人那里学到，那时代岂不是在倒退吗？

2.4.2　俄罗斯方块竞赛

另一个自编的小游戏对我影响颇大，这也几乎是所有业余编程玩家都实现过的——俄罗斯方块。

第一次接触这个游戏是从一盘任天堂红白机的卡带。我曾经见人彻

夜不眠地摆弄那些方块，自己试玩以后，也是异常着迷。虽然家里已经有一台任天堂游戏机，可是游戏卡带还比较昂贵，只能通过同学间的交换。所以我下定决心自己实现一个，这样随时都可以玩了。

第一版的程序是在课堂上完成的。当我有了想法后，即便在上课，也无法抑制把程序写出来的冲动，就打开笔记本用铅笔在上面编写。程序的逻辑并不复杂，加上我已经有了成打的自制小游戏的经验，只用了两节课的时间就完成了。中间审核了好几次，修改了一些细节，一到放学，就飞奔到机房，输入计算机，一个游戏就这样诞生了。

这是一个画面粗糙的游戏，工作在文本方式下。屏幕上显示的都是些用英文字母和符号拼凑起来的画面。但机房里的朋友们都很有兴趣。其中我的一个好友也来了兴趣，第二天也拿出了一个他的版本。可惜我们俩的程序都有一个相同问题，速度太慢。当时大家不约而同地都采用了一个二维数组来描述游戏中方块的状态，然后每个方块下落的时候再判断是否可以移动，当落到底时，用一个循环来检查是否有一行可以消去。

这是一个最容易想到的方法，可是由于当时我们所用的机器是Apple II，只有1MHz的主频率。编程语言是一个解释型的BASIC，速度尤其慢。所以这几个循环程序就显得非常慢了，尤其是消去判定。

而且在消去某几行后，整个方块积木的下落所造成的大面积重绘图像，都是肉眼可以感觉得到的。

在接下来的日子里，我们几乎每天都要分别拿出一个更新过的程序出来，新的程序总能比老的程序快一点点，我们的目标是让游戏真正可以爽快地玩下去。一开始，我们采用另一个一维数组记录每行已有的方块的总数，在活动方块落下后，只需要修正对应行的记数，然后判断一下这个数字就可以知道这一行是否已经填满。

而后，干脆改用一维字符串数组来保存游戏状态，这样既可以用字符串比较来取代使用二维数组里的循环比较，又可以用PRINT一次输出一行图形。后面这个改进方法使游戏速度大增，游戏已经可以非常流畅了。

这并不是一个特别好的算法优化的例子，因为它被局限在一个特定

相比当时，现在随时都可以打开计算机编程简直是一种奢侈。甚至现在的编程环境，图形化的IDE都远非当年能比，现在可以一边写出点代码，一边试着编译，找出语法错误，或者在完成了几个函数后，测试一下工作是否正常。

但是，有过纸上编写几百上千行程序经历的我，有时反而觉得，用稿纸和铅笔在桌上完成代码的初稿，用自己的头脑来检查代码中潜在的错误，也不一定是坏事。强迫自己边写边思考，和让我们写出来试试看，有问题再说，是两种不同的编程方法。前者，或许最终程序能更快完成呢。

的环境下。但仔细研究其思想，其实非常常用。

其一是尽量避免重复运算，我们记录下固定的方块的每行记数，就是为了减少重复的循环判断。

其二是尽可能地让计算机做并行操作。在 BASIC 层面上看这个问题，处理一个数字比较和处理一个字符串比较是一个数量级的问题，输出一个字符串和输出一个字符也是一个时间量级的。

利用字符串简化大量的循环操作解决了速度的根本问题。用字符串取代数组的取巧方案是在那个年代缓慢的解释型 BASIC 上的一个优化特例，放在今天编译型的 C 语言上来看，应该没有什么区别。

不过，让计算机在原本干一件事情的基础上想办法并行多干几件同样性质事情的这种思想并不落后，在后面的关于 MMX 优化的章节，我们会重温这个话题。

2.5　Apple II 上的编程之路

我真正沉迷于编程的世界，其实是从接触 Apple II 这台计算机开始的。

刚入初中不久后，我开始了系统的编程学习，或者可以说是一种训练。当时初中是没有专门的计算机课程的，学生学计算机多是以参加兴趣小组的形式。而每个学校会从中选拔选手参加 NOI（全国青少年信息学奥林匹克竞赛），几乎每个学校都很重视各种奥赛活动。

第一次见到一个房间里有那么多的计算机，那么多喜爱编程的朋友坐在一起，于我是非常兴奋的。大约花了半年的时间，我和新认识的朋友们从头系统地学习了 BASIC。有那么多同龄人可以一起讨论交流，我从未感觉到编程可以是这样简洁美妙。

学校里用的是中华学习机，一种国产盗版 Apple II 的机型，很多 70 年代出生的程序员，应该对这个名字倍感亲切吧。

它比家里的 Laser-310 要有趣得多，因为不再需要用老式的磁带机读入那几个简陋的程序，一张张 5 寸软盘中，有各式各样丰富多彩的

对于 Apple II，我并不陌生，小学的时候，有段时间父亲把铺盖拿到单位，一连几周不回家，就在一台 Apple II 前工作，晚上我坐在母亲的自行车货架上去给父亲送饭，每每在办公室里打游戏，就是那台 Apple II。

新鲜游戏。

虽然 Apple II 的 CPU 并不快，6502 只有 1MHz 的主频，甚至比不上 Laser-310 的 Z80A（3.54MHz）。但是它拥有 64KB 内存和更高的分辨率（280×192），这些使得游戏可以做得更加丰富。5 寸的软驱使软件交流更为方便。Apple II 上丰富的软件促成了它的成功。

Apple II 这台简单如玩具的计算机造就了很多人，因为它简单，所以让玩家有种一切皆在掌握中的感觉。而不像现在的电脑，复杂到用好它的软件都能成为一方专家。

Apple II 内置了 BASIC 语言，这一度是个人电脑的标准。最早的 IBM PC 也是内置 BASIC 的。那个时代的理念就是，会写程序是电脑玩家的充要条件，即使你不会写，至少也要懂得把杂志上刊出的小程序输入到机器中。Apple II 的玩家大多是 BASIC 大师吧。

BASIC 在那个年代就是编程的全部。我至今依旧认为，初中的那段编程学习经历，是我程序员生涯中最珍贵的一页。那个时候学校培养学生学计算机，是略微带一点功利性的，就是参加各种比赛。比赛的培训非常辛苦，甚至是中国传统教育式的填鸭式教学。老师教给你算法，大家就背下来，不停地训练编写，直到各种题目都能迅速无错地完成。甚至于当时的比赛平台就是在 Apple BASIC 上面，这种 BASIC 远不比现在的 VB，没有结构，没有子函数，没有局部变量，有的只有全局变量和数组。很多算法实现都是变了形的，光是递归的实现，就需要自己用数组模拟堆栈。一个简单的二叉树遍历的程序，就需要花上几天的时间去悟。

但是，我一点也不想抱怨这种教育制度，其实，一切皆在于人。我们被用最高效的方式灌输了无数用得着用不着的知识，不正是为了以后可以灵活运用吗？金庸小说中的张无忌最后成为举世无双的高手，我看大半来自于他儿时义父对他的填鸭式教学。当然还需要一点点悟性。我们可以适时地从书本中脱离出来，审视自己脑中的知识，整理借鉴。人脑内部对信息整理的优势，比需要用时才查询资料可是要有效得多了。

C++ 标准库的确已经实现了诸如快速排序、二分查找这些算法，我们现在也倡导和鼓励尽量去使用标准库，这样会减少出错的机会，得到

不错的效率。但是，作为对自身编程能力的训练，自己去多多动手实现是有百利而无一害的。

实际工作中，我们总能碰到一些问题，现成的函数库往往得不到最佳解，需要自己动手一行行地实现。多年的编程经历让我明白了一个道理：绝大多数情况下，没有解决不了的问题，只有因为平时缺少练习而惧怕问题的复杂度，畏惧的心理让我们选择避让，采取并不那么好的方案去解决问题。最后，还可以找到一个合适的理由，比如一切以稳定或一切以工期为重，以此获得心灵的安慰。

世界少有天才的程序员，更多的是勤奋的程序员。只要不停地编码编码再编码，同时在每次编码后有一些思考，编程的水平自然就会提高。如果你有和我同样的经历：被关在机房中编写那些竞赛的习题，不做完不准吃饭，那么，一定会赞同我的观点。我的编程基本功就是那样被训练出来的。

兴趣是程序员一直做下去的源动力，尤其是游戏程序员。他们可以让计算机变成娱乐自己的玩具。虽然只有少部分学习编程的人最终以编写游戏为职业，但大多数人最初都自己写过小游戏自娱。我自己当年也是这样的。

当我熟悉了中华学习机后，我简直对计算机着了魔。对于把报纸杂志上刊出的小游戏输入到计算机中这种事情早已不屑一顾。整天泡在学校的机房不回家，用 BASIC 写一些小游戏。Apple II 上跑的 BASIC 虽然慢，但是处理一些文字还是足够快的，所以，最初从键盘上诞生的游戏都是用 ASCII 字符拼出来的“象形文字”。最初的游戏角色都是类似这样的（如图 2-1 所示，好像一度非常流行的 ASCII ART）。

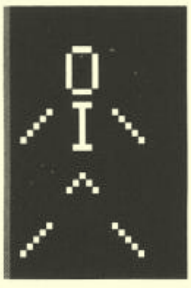


图 2-1　ASCII 字符拼出来的图形

如今几年，每年到了年底，我都会奔波于全国各大高校做招聘工作。选拔人才是公司发展中的重中之重。所以，项目上的事情多忙，我都会自己出去面对众多学习编程的学生。但不无忧虑地发现，现在，有很多学校里的佼佼者并未在算法这基本功上下足工夫。

他们把过多的时间花在了学习时髦的开发工具、新的语言、各种 API 上；又有一些人，虽然脱离了这些浮华，十分专注地去猛啃 C++、设计模式这类书，但是，却很少有人重视基本算法的实作，这方面的实际经验太少了。

在BASIC里，只需要用几行PRINT语句就能把图案表达出来。但是，一开始能让我们用ASCII字符构造出游戏世界已经很让人满足了。而且我有一帮有着相当爱好的伙伴，在那难忘的两年里，我们互相比试着谁做的游戏更有趣，更复杂，更好玩。

迷宫中穿梭着的飞行器、房间里互相格斗的小人、赛道上奔驰的赛车。那时候，大家都很高产，几天就能折腾出一个小游戏来，创意来自于街头的游戏机、Apple II上的商业游戏，以及自己的遐想。几乎每有一个念头，就立刻奔向机房去实现它。

旁人看来，那些屏幕上晃动的字符，比不上街头绚丽的游戏机，比不上家中的任天堂红白机，甚至也远远比不上Apple II上已有的丰富的图形界面的游戏。但是我们却乐此不疲，其实，更多的是享受着创造游戏的乐趣，是向别人炫耀“看，这是我做的”这种感觉。

在那个年代里，游戏还没有现在这么复杂，还没有这么多商业味道很浓的游戏，即使是专业游戏厂商生产的游戏，也同样是很少的一两个程序员在不长的时间完成的。自己动手靠个人能力制作个像样的游戏，不会给人一种可望而不可及的感觉。这或许是那个年代接触计算机的孩子们很容易萌发“我要自己做游戏”的念头的一个重要原因。现在的孩子更容易沉浸在声光影包围的游戏中，把游戏看做一种工业产品，是许多大人们做给小孩子的高级玩具，而不会动起自己做游戏的念头了。

如果有幸我的读者中有和我当年一般大的小孩子，我想在这里说，其实这些年来，游戏的确拥有了更丰富的表现力，但制作游戏本身却没有太多革命性的改变，绝大多数游戏程序还是很少的几个人用传统的方法编写出来的，游戏程序也很少能达到中等以上规模软件的水平。编写游戏程序，依然还是只要你想，就一定能做到。

在此期间花掉我最多时间的是一个双人格斗游戏，想法来源于当年街机上非常流行的“双截龙”和“战斧”。最初我希望做成一个同样的横版格斗过关游戏，但是后来发现，机器的速度无法处理很多的ASCII图案的人物，然后，我就把设计改成在一个房间里，人和计算机各控制一个小人，用复杂的格斗技能较量，后来我在街机上发现了类似形式的游

戏“Street Fighter”后，还颇为自己的创意感到自豪。

我从刚接触 Apple II 一个月后，就开始动手完成这个自己感觉规模庞大的游戏，一直到 2 年后转向 PC 平台，这个游戏的代码被重写了 3 次以上，设计方法几经翻新。最后我希望把源码打印到纸上留一份硬拷贝作为纪念，可是估算了下，需要用去厚厚一打打印纸，觉得甚是浪费，就打消了这个念头。从这里面，我领悟了许多游戏程序设计的基本方法。

首先，应该设计一些程序负责最终显示图像，然后由另外的程序调用它。那个时候的机器没有硬盘，软盘速度很慢，所以图像资源是没有额外放在外部文件中的，所谓的图像资源就是一个个字符串，可以用 PRINT 以及屏幕定位语句输出到屏幕上。

当时我用 BASIC 的实现就是做了一个个独立的子程序，里面包含了显示每个独立元素的语句，只要设定一些参数，调用对应的子程序，就可以在屏幕指定位置画一组符号出来。这些数据我是用的额外的一个工具，可以在屏幕上所见即所得，用键盘画出我希望的字符组合，并生成对应的代码保存起来。

这实际上就是将游戏的图像层和逻辑层分开的观念，游戏的图像应该有一些专门的代码负责它们的显示，这就是所谓的图像引擎，而控制游戏逻辑的代码面对的必须是完全和图像显示无关的基于游戏元素的对象。通过调用操作对象的方法来完成最终的显示。

其次，我曾碰到过一个非常棘手的问题，游戏中的人物通常完成一个完整的动作需要画一个系列的连续图片，并且在这个过程还需要做攻击判定等逻辑操作。当屏幕上两个人物同时运动时，似乎程序的逻辑需要脱离老师教给我们的三种基本程序结构：顺序、循环和分支。看起来，是两段对立的逻辑需要同步运行，它们还需要交互。

因为无法解决这个问题，我的这个游戏一度搁浅。有一天，邓给我展示了他完成的同样的程序，只是两个小人不能同时做动作，显然他也碰到了相同的问题。但是毕竟那已经是一个可玩的游戏了。我的某根神经被刺激了，发誓一定要完美解决它，我的游戏要更好玩！

当年的BASIC 无法做到对象的概念，但是，现在各种完善的高级语言都很容易实现。如果把显示和逻辑混杂在一起，程序必然会非常混乱，规模无法做大，只能做出一些需要快速实现的游戏DEMO 原型。

后来发现的方法是使用了一些状态变量，即，给游戏中的角色们赋予一个状态信息，在每次游戏循环中，查询角色的状态，根据状态来决定是去获取用户输入或者程序控制逻辑来改变状态，还是自动变化到某种新的状态中，然后在当次游戏循环结束时，根据两个角色的当前状态调用对应的绘图函数将其表现出来。不久以后，我从老师那里知道了，这叫做状态机。

有限状态机广泛地被用于游戏程序中，多用来解决类似上面的问题。有许多人可能会想到使用多线程来控制游戏中的角色，但是，多线程还是有不小的开销的，而且存在复杂的同步问题。不过，我也曾经尝试过自己实现一个轻量的用户级协作式线程，由用户自己主动控制线程的切换来减轻同步的复杂度；用简单的跳转语句以及少量的环境保存来缓解OS级线程的切换开销。

这在我参与研发的一个商业网络游戏《梦幻西游》的客户端中得到大量的使用，效果不错。另外，在GDC 2004上我听了个演讲，介绍了一种基于行为描述的语言（A Behavior Language，ABL）来解决上述问题，关于ABL可以在egl.gatech.edu找到。

当时的条件所限，我们无法通过互联网交流，查询资料，请教前辈，但即使在今天，我依然提倡碰到棘手的问题，先不要急于上Google搜索，或者在IM软件上找人问，自己的思考是最好的解决问题的钥匙。每个问题都会有无数的解决方法，每次独立解决一个问题，都是一次开拓思维的机会。

编程语言

至少要学会一种编程语言，才可能编写游戏软件。流传于世的编程语言数以百计，但真正流行的却不多。各种编程语言之间有它们的共通性，但也并非能够一通百通，甚至也不存在一种适合解决所有问题的语言。

这一章提到了最为影响我个人的几种语言：C，C++，BASIC，汇编，以及一些我感兴趣的，Forth Lisp Jave 。

我认为大部分游戏程序员都应该掌握工作平台的汇编语言。这对理解硬件，写出更好的代码，还有软件调试都有莫大的帮助。

编程语言永远不会是计算机编程艺术中的主体，它只是扮演着工具这个配角。学术界史诗般的巨作，《计算机编程艺术》（The Art of Computer Programming）创造了一种很贴近现今计算机模型的语言来阐述问题，而没有使用任何流行于今的编程语言，就是想给读者传达某种精神。

我将关于编程语言的章节编排在算法之后，也有点类似的意味。但是于我，是深知不能与前辈大师相提并论的。本书也只是描述自己对编程开发领域一些问题的看法，而项目本身必须结合到实践中去，所以我们便逃脱不了对编程语言的讨论。

我希望本章所述内容不会显得过于技术化，这样，更多的对游戏感兴趣却不了解编程的人也可以读读。其实，一门编程语言并不难学。当一些游戏策划人员突然萌发了一些学习编程的兴趣时，却往往受拒于厚厚的教科书，这真是一种遗憾啊。

3.1 C 语言

1993 年 5 月，父亲带着我从旧货市场上淘回来一台破烂不堪的 386，主频只有 16MHz，没有做浮点运算的协处理器，14 寸显示器的塑料外壳坑坑巴巴，虽然配有当时最新的 Trident 9000i 512K 显存的显卡，但分辨率却只能上到 640×480。整套机器只有 40MB 硬盘和那个弹簧机械键盘是新的，但这已经让我超乎预想地满足了。在购买一台 286 都很勉强的预算内，能够拥有一台 386，我才不会管它是新是旧呢。正是转向 PC 平台，才让我在编程方面跳出了 BASIC 的框框，让我进一步领略了程序设计的魅力。

这年，是我初三参加中考的一年。在这年，我的信息学奥赛水平也发挥到了最佳状态。由于 Apple BASIC 在解决复杂的算法面前，实在天生残疾，没有函数可以对问题分而治之，没有结构可以描述复杂的数据，没有动态内存分配可以驾驭内存，所以，几乎所有人都开始转向更完善的结构化语言，比如在学术界享有盛誉的 Pascal。

学校的中华学习机由于硬件限制，是无法跑起编译型的Pascal环境的，但随着PC渐渐进入中国，眼见着其取代Apple II成为家用计算机的主流机型，信息学奥赛也换到PC上，这样大家可以有更多的语言选择，BASIC，Pascal或者C。

Pascal和C都是当时非常流行的编译型语言，这点和BASIC大相径庭。Pascal是那个时候老师与学生的首选，可是到我选择学习哪种语言时，不知道是处于天生的那么一点点叛逆，还是因为学过一些汇编语言，对C这种更接近计算机本身模型的语言天生好感，又或许是学校里英文成绩不佳的我，对于{}这样的符号的亲切度远胜于Pascal中的begin/end，我几乎是毫不犹豫地选择了C。

但是，这个选择直接让我面对的就是，没有辅导的老师了。电子工业出版社出版的一本小册子《C语言程序设计400例》成了我的启蒙教材。在刚刚接触C的时候，我几乎只能在纸上练习，这种印有一段段小程序，并附有运行结果的书，无疑能给我莫大的帮助。直到把整本书的例子都看完，我才去读另一本系统地讲解C语言的书，这时，各种C语言的基础知识已经了然于胸。

现在回头来看，Pascal和C并没有非常严格的语义上的区别，只是在当初语言设计的哲学上略有不同。Pascal想传达一种规则的结构化观念，让使用者潜移默化地接受这种观念；而C语言则想尽可能地贴近机器的固有模型，提供一种程序员更容易接受的描述工具。

这个角度看来，Pascal提供了许多累赘，在那个年代里，更多的程序开发人员的思考方式比较会去排斥它吧。于我看来，即使不考虑C早期被用来编写UNIX系统这个重要因素，历史选择了C而不是Pascal作为主流开发语言，并非偶然。而我当初也是跟从了我自己那一类程序员的直觉。

由于C和最原始的BASIC设计理念完全不同，所以并不像很多程序员认为的，学习了一门计算机语言后，就能一通百通。或许学习了C后，去学习类C的语言是这样，但从BASIC到C的过渡却不那么容易，一切仿佛需要从头开始。依靠实例来学习新语言或许是最快上手的方式。

我很幸运能在入门之初碰到那样一本好书，到我后来上大学后，看着同学捧着一本C语言教程之类的厚砖头，用掉大半年的时间才得门而入，就非常想推荐他们我当年用一个月速成的道路，可惜书店里没能再找到合适的读物了。

花絮

刚学C语言那会儿，很难找到上机实践的地方。后来，父亲认识的一位在附近一所大学教计算机的老师介绍他的学生给我认识，这个家伙正在做毕业设计，从大学的机房里借了台PC回家用。我便每天放学后

先赶回家吃完晚饭，然后骑车去他家里做C语言的程序习题。

他姓傅，比我大不了太多，但是正在大学里学习计算机，上过C语言课程，是我当时唯一可以请教C语言问题的人，所以我称呼他为傅老师。每天在傅的家中时间有限，一般我都会在白天把想要做的事情想好，晚上一气呵成，做完习题后还有点时间打打PC上的游戏。“波斯王子”这种现在看来图像简陋的东西，在当时，已经感觉是神般的作品了。

那段时间，躺在床上一合眼，就想着如果自己有一台PC就好了。直到突然有一天，家人答应给我买一台，让我感觉在梦里一样，之后的很长一段日子，我还是会梦到一台PC，或许是当时的期盼太刻骨铭心。

上高中以后，我再也没有机会跟傅联系。在我记忆中，他也是个才华横溢的人，那个时候想做一个类似WPS的软件，并已做出了雏形。不知道他现在怎样，我向他家打过电话，电话号码已经换了，或许是搬家了吧。

多年之后，我考上了外地的大学，一个暑假，我在返家的火车上碰到一位老乡，闲聊之中，居然发现我们毕业于同一所高中，只是比我年长一些。我推算了一下年龄，发现他应该和傅同届，而傅也是那所高中毕业的。进而一问，居然是傅的同班同学，不由感叹世界之小。原来傅已经结婚生子，在一家大的国有企业工作，不知道他现在还有没有从事编程的工作呢?

C语言本身是简洁的。语言本身仅仅是提供了一种用计算机角度实现算法的符号规则而已。我们用C语言描述的就是算法的步骤，可以是顺序的，有逻辑分支的和重复循环的。所有算法细化到底，无非如此。

而为了让问题可以分而治之，C语言将所有的模块都称做函数。函数就是有多个输入参数和单个输出参数的语句集合。函数之下，语句之上，另有语句段（statement）作为组织算法层次的描述工具。

对于不需要对程序模块进行严格划分的地方，C语言一样提供了全局变量，可以让函数之间不通过输入参数来访问一些共有数据。这点是符合C语言的设计理念的，因为函数封装的概念是高于现代计算机的基础模型，冯·诺依曼体系的。C语言本身不应该限制程序员的发挥。当

然，这种设计方法是应该被指责的，并不被推荐使用。

但也并不是说，C 语言就能直接描述一切。

对于C 语言，数据运算操作本身就是被固化在语言层。语言设计者只是根据计算机可能固有存在的数据运算操作，一一定义C 语言中的运算符。比如移位这种只针对二进制的操作，虽然在理论上的数学运算中仅仅只是一种乘除法的特例，但只是因为几乎所有的计算机都能直接操作，所以被加入C 语言中。

到了今天，现代CPU 扩展了一些数值运算的操作，比如后面将用整章的篇幅讲述的 SIMD 运算，C 语言就失去了对它们在语言级的支持。

另外，协同例程（coroutine）也是不为C 语言所支持的。但它的的确确是控制逻辑的一种。

举这些例子并不是为了说明C 语言的缺陷，实际上，C 语言没有做到的，别的语言也几乎不能。好在几乎所有的C 语言编译器都给予了内嵌汇编的能力，由于C 语言天生接近机器固有模型，所以混杂C 和汇编编程是一件很容易的事情。

数据描述的方面，C 语言同样提供了简洁明了的方式。除了计算机硬件固有支持的整数、浮点数这样的数据类型外，C 语言用最直接的方式给出了结构（struct）、联合（union）、数组（array）、指针（pointer）的定义。这些让 C 语言中用到的数据结构，可以非常透明地映射于内存中。

其中，最为人所诟病的就是指针的设计。从高级语言的角度讲，程序员不需要指针。我们只需要有可以指代数据的标识之物即可，而不需要真实地将内存地址表示出来。事实上，指针是大多数程序 bug 之源。

用指针实现的数组和纯粹是方便数据组织形式转换的联合，都是为了贴合机器模型而脱离严谨的程序设计方法的，这些无不体现着C 语言的设计理念。这种理念被一直传承下去到C++。我把C++看做一个和C 语言不同的语言，正是由于设计理念的相同，让我觉得它无愧于C++名字中的C 字。

3.2 BASIC

无论我对众人心目中的那个 Visual BASIC 有多么不屑，但平心而论，BASIC 真的是一个非常伟大的语言。BASIC 足够简单，让初学者也可以轻松地学习并解决几乎所有的问题。它的名字 Beginner's All Purpose Symbolic Instruction Code 就是想传达给使用者这些理念。

BASIC 从来就不是为了发挥计算机程序效率而发明的，它就是让用的人感觉编程是如此简单。最早的 BASIC 只有十四个语句，语言只有非常少的诸如全局变量、语句行、分支跳转这些基本概念，可谓简约之极。编程者只需要一行行把自己想干的事情写下来，让计算机一步步去做即可。

BASIC 还具有简单明了的交互式特性，编程者输入一条指令，就可以立即看到结果，犹如计算器一般方便。这一点，后来的 Python 将其发展发挥，受到诸多编程者的喜爱。

BASIC 诞生之初，计算机可以解决的问题都还不算复杂，甚至不需要将问题按层次细化，大约就是第一步怎样，第二步怎样，符合条件则转到某一步。初期的 BASIC 就是用语句行号来表示这些次序，用固有的单词来表示要做的事情的，比如，输入、输出、计算，等等。计算机再用一套系统去理解这些语句，循序执行。

背景

解释型语言与编译型语言

BASIC 最初被设计成一种简单的解释型语言，BASIC 里的每个指令都只是一个代号，那些数据则是一些参数。计算机里运行着一种叫做解释器的程序，读入这些代号和参数，按预先设计的逻辑做出对应的操作，仿佛在释读那些 BASIC 代码。每运行一次，解释器就释读一次，无疑，大量的时间都花在翻译这些代号上了，其运行效率自然比计算机的原生机器语言要慢上一截。

而 C 这样的语言，被设计成在完成代码的编写后需要多做一些工作，即它们会被一次性翻译成为本机的机器代码，这样，以后运行时就不需

要做重复翻译的工作，如此提高了运行效率。粗略一看，被节省的只是翻译代号的时间，其实不然。当代码按需要一句句翻译的时候，每句代码运行完毕时，语言设计的逻辑就要求那些中间的状态必须以规定的格式保存起来，在后面的语句需要时重新被加载。

而高级语言的编译过程就好比长长的数学表达式被展开，经常会有同类项被合并，使其整体变得更为简洁，最终被翻译成机器语言的程序存在的形式不再和高级语言的语句一一对应，运行速度也在这种编译过程中得到了提升。

BASIC 并非由其发明者而是由微软推广到全世界的。正是微软，把 BASIC 改造成了一个可编译的语言。几乎所有的 DOS 用户都用过 QBASIC 吧？那是 QuickBASIC 的一个简化版。IDE 的概念也被引入 BASIC 环境中。用户可以方便地编写、调试、运行程序。至于日后的 Visual BASIC 更是引领 Windows 程序 RAD 开发的潮流，不过，BASIC 发展至此，已经只剩下几个似曾相识的关键字了。

至于嵌入几乎所有微软应用软件的 VBA（Visual BASIC for Applications），我个人用得不多。我更愿意把它看做一个有着 BASIC 部分关键字的 COM 调用的描述工具，已经和我印象中的 BASIC 语言相去甚远。

似乎在今天，由 BASIC 入门来编程的人越来越少，C 作为主流的时代也正在过去。有时我也有点怀念 BASIC，这个简约主义的编程语言。几乎每个受过逻辑学教育的人，都可以快速地理解它并帮助自己解决一些问题。从理解编程这件事情本身，我不觉得有比 BASIC 更好的选择。

3.3 C++

有了自己的 PC，每天都过得很兴奋，我在自己的机器上安装各种新潮的软件，试着各种新奇的东西。和许多同时代的 C 程序员不一样，我的开发环境并不是当时最流行的 Turbo C 2.0，而是 Borland 公司出品的更新的一个编译器 Turbo C++ 1.0。

当时我手头只有一本翻译过来的Turbo C++教程，读到class的概念时，就被深深吸引住了，把世间的事物都抽象成对象和对其的方法真的是以前没有想过的思路啊。从BASIC转到C后，我最大的不满就是字符串的处理麻烦了很多。使用class，重载了一系列的操作符后，就可以让C++按BASIC那样原生类型的方式来处理字符串了。而进一步，将有共性的对象细分，使用继承的手段表现出它们的层次，然后用virtual方法来贯穿其中，最后让整个世界都和谐统一起来，C++就是这样的有趣。

作为世界上第一个商业化的C++编译器，Turbo C++ 1.0现在还可以在Borland公司的主页上的网上博物馆中免费下载到。下面这张图3-1记录下了Borland当年的辉煌。

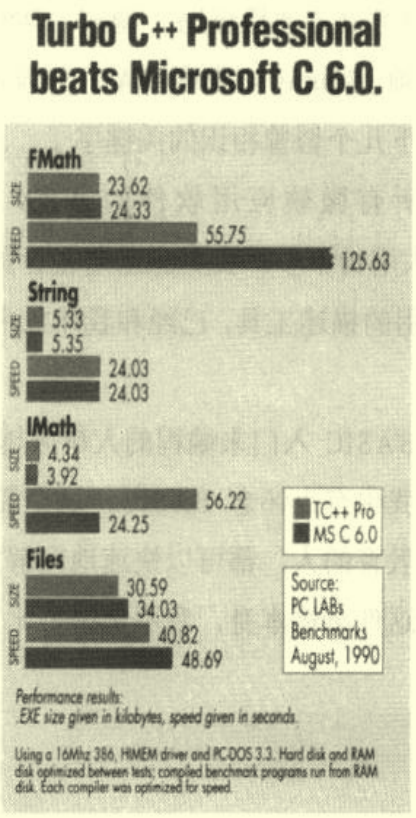


图3-1　Borland当年的辉煌

对于C++而言，它是非常简陋的，甚至template都是一个没有实现的关键字。而且，当年我的机器仅仅只有16MHz的主频，编译起C++来相当缓慢，DOS下过小的可自由使用的内存也成了使用C++的瓶颈，

不久，我还是放弃了C++的学习，重新拿起C作为编程工具，但这并不妨碍我喜爱C++。

和许多人理解的不同，C++并不等于面向对象的编程，虽然它给人的第一印象是为了面向对象而设计的。实际上，C++是一种支持各种编程范式的语言。它支持面向过程的编程、基于对象的编程、面向对象的编程，以及使用 template 实现的泛型编程。

C++ 是一个足够复杂，但因为其复杂而变得足够有趣的语言。当我从 2000 年重新沉迷于C++后，反思当年没有马上转入C++而一直使用C的原因时，得到的结论是：当年的我直觉上感到了C++的复杂度，知道需要花去大量时间才可能入门，所以做出了逃避的选择。在我的信条中，没有十足了解的东西是不会去用的，直到有一天，我看见每一行C++代码，头脑里就能反应出其编译器转化出来的汇编实现，以及每个C++对象在内存中的布局时，就毫不犹豫地投入了C++的怀抱。

在很多人的观念中，C++相对 C 会有 5%~10% 的速度损失。以我对C++的了解，是非常反对这个观点的。其实，C++和C一样，都是最贴近计算机固有模型的，C++的每一行代码，都和C一样有明确的目的，没有去干多余的事情。如果你非要说经过实际测试，C++ 的工程就是比C要慢大约10%左右，那么，这只能是C++的某种间接性或者编译器在黑暗处做了某些多余的手脚造成的。请在了解C++后仔细思考一个问题：多出来的时间，到底消耗在哪几条指令上去了？

实际上，无论参数传递的方式，对于调用栈的处理，还是编译器实现的虚函数表与自己手工实现的函数指针表的对比，以及C++异常对于C方式的 setjmp/logjmp 模式，这些语言特性在具体实现细节上，C++ 在各方面相对C都有非常微弱的速度优势而不是劣势。甚至于C++中独有的 template 推演，可以编写出C的宏指令实现不了的许多东西，对于空间换时间的策略的运用，为C远远不及。

今天，C对于C++的优势在于简捷，而非高效。当然，由于C++更多的时候容易被编译成更大的代码体积，在内存空间有限到不足以储存下程序代码本身后，还留出大块的内存应付数据的动态分配，所以它没有像C那样更适合做小内存的嵌入式系统开发。

对于大多数平台的游戏开发，C++绝对是比 C 更好的选择。因为这一点，我们在后面还会专门辟出一个章节讨论 C++。

3.4 汇编语言

提起汇编，借用上一章讲过的故事，那个俄罗斯方块游戏的优化比赛作为开篇吧。

那个故事并没有到前面记叙的地方结束。因为我们又得到了一个 Apple II 上商业化的俄罗斯方块游戏软件。自然，那个游戏并非跑在文本界面下，有着很 Cool 的图像，和我在任天堂红白机上玩到的同名游戏质量不分上下。

我并不惊诧于人家的实现如此完美，因为大家都清楚，我们只是一个刚刚入门的 BASIC 程序员。而人家，不是用这种玩具语言来创造游戏的，他们使用的是一种更贴近机器，更能发挥机器能力的汇编语言！心中虽然明了自己的差距，但触动还是很大。

如果想真正了解计算机，想以后做出真正受大家欢迎的游戏来，那么必须学会汇编，当时我很奇怪地突然萌发了这种想法。恐怕如果我晚几年接触编程，用的是 PC 平台的话，就很难这么快对汇编生出兴趣了。当时正好家里有几本关于汇编的书，我便抱起来啃了。好在遇到不懂的地方家里还有现成的老师问。

我对汇编语言并不陌生，当年父亲整夜整夜地不回家在单位工作，为一个水泥搅拌机器写控制程序用的就是它。我记得那个程序甚至可以在屏幕上显示出汉字，画出机器运行状态的图像来。父亲曾告诉过我，他用 Apple II 的 6502 汇编写这些程序。

BASIC 或者 C 这些并非计算机的母语，每种计算机都有自己与生俱来的语言，Apple II 的就是 6502 汇编，用这种语言控制它可以发挥最大的能力，做计算机可能做到的所有事情。绝大多数工作都是 BASIC 无法胜任的。

6502 这种 8 位的 CPU 设计足够简洁，我的汇编学习非常顺利。从这之后，有了汇编语言这个武器，看一些关于 Apple II 的技术资料要顺利多了，随时还可以去查看机器内部是怎样工作，一切尽在掌握之中的感觉真是好极了。

我的第一个作品就是为我的俄罗斯方块写了一个屏幕窗口下滚的汇编子程序供 BASIC 主程序调用，果然速度提高了几个数量级，几乎感觉

不到延迟。

编程好比驾驶汽车，现在的轿车设计得都非常舒适，经过短期的培训，一般人都可以轻易地控制它。而驾驶员毋须知道汽车的结构，以及引擎如何工作。恐怕，许多（而非全部）的 RAD 开发人员都在这个状态徘徊了。

但是，换做一个 F1 方程式赛车的选手，他一定对他的车了如指掌。嗯，这个例子不大好，我想我的读者中应该没人开过 F1 方程式赛车。不过卡丁车很多人玩过。开过一次的话，应该就能体会到车上每个零部件对整体的影响。身体也成了车的一部分，对车不熟悉的话，一个不小心，就会撞上道边的轮胎。

虽然今天 C 或者 C++已经非常贴近机器的原始模型，编译过的代码可以以最高的效率运行，甚至许多糟糕的汇编程序员写出来的代码不如编译器产生的，但这并不能成为拒绝学习汇编的理由。

在合适的场合用汇编写出合适的代码，可以使你的程序跑得更快；而且，绝大多数 C++编译器也无法生成一些特殊的指令集，比如 MMX，SSE，做特定的优化（这是由语言本身的特性决定的，而非编译器厂商）；了解汇编，也就能了解你的代码最终会以怎样的形态运行于 CPU，这对于你理解高级语言，比如 C/C++ 都是很有帮助的。

最后，汇编还能在紧要关头帮你的大忙，比如程序出了莫名其妙的错误，而你无法在源码级调试程序的时候。我建议每个游戏程序员或多或少地了解一下你工作平台下的汇编。

3.4.1 概述

通常，计算机的心脏是一块叫做 CPU 的芯片，正如我们上章介绍过的，它也不过是一种复杂化的图灵机，接收一些信息，按设计好的规则，利用内部保存的一些状态做出某种有意义的输出。由于设计上的方便，我们和计算机的交互，全部采取 2 进制数字，简单的 0,1 完全可以表示世界万物的所有信息；我们给 CPU 下达的指令，就是一些 2 进制数字。

由于2进制数字过于冗长，不利于人来阅读，所以通常我们使用16进制作为书写阅读的替代品，这些数字最接近程序底层，它们被称为机器语言。

人工直接用机器语言编程控制计算机几乎是不可能的，所以有了汇编语言。汇编语言其实仅仅只是用一些英文单词的缩写来表示机器语言中的数字控制指令，并将一些数字表达的规则简化成一些符号来表示。所以，用汇编语言写出来的程序，可以无二义地被翻译成对应的机器语言，两者是完全等价的。

每种类型的CPU都有它们的方言，在机器语言一级，差别很大，而在汇编语言这个级别，被人为设计得比较类似，毕竟，现今的计算机设计体系都很类似。也正因为如此，在更高层次的语言上，可以做到相同的源码被翻译到讲着不同方言的计算机上运行。

另一方面，高级语言在设计之初都是遵循着一定的语言哲学的，连最贴近现代计算机模型的C语言，也无法做到原生语义上涵盖所有类型的CPU方言体系，比如现在x86系列CPU上的MMX，SSE等指令集，标准的C语言语句就无法描述出等价的语义。

过时的8位机汇编语言已经完全不值得介绍，除非你很怀旧，想做一个任天堂红白机的模拟起。

这里，我们还是以最为流行的、PC游戏程序员最为实用的x86汇编作为蓝本。但这里也并非一个x86汇编教程，仅仅是想帮助对x86汇编语言概念模糊的人进入汇编语言的大门。如果你已经系统学习过x86汇编，或者至少在调试C++程序时，能够看懂编译器生成的目标代码汇编表示的绝大部分，那么请跳过这个部分。如果你对汇编感兴趣，但却没有学习过，希望用最短的时间有个整体的了解，那么以下内容非常适合。

这里我仅仅选出最常用的80386以上的x86体系的CPU中保护模式下的常规整数汇编指令集的主要部分进行讲解。如果希望系统学习汇编语言并用它编写实际代码，市面上有许多系统的教材可供选择。

3.4.2 程序的本质

在 32 位操作系统环境下，每个进程拥有一片连续的内存空间，程序本身和其需要的数据一起共存于此。这片空间的每个字节都可以用一个 32 位无符号整数定位，每个字节都有唯一的空间位置，我们称之为地址。这里的内存一般并非真正的物理意义上的储存设备，我们把它叫做地址空间。

不是所有地址上的字节都能够被实际操作，这取决于操作系统如何把地址空间和真实内存对应和调度。地址空间的大小为 2 的 32 次幂，也就是4G。由于程序运行于操作系统之下，或多或少地需要调用操作系统的 API，那么，操作系统本身的代码和数据必须和用户程序以及数据共存在一个地址空间下，在 Windows 环境下，用户实际直接操控的地址空间只有 2G（或者 3G）。

CPU 中有一些内部的储存单元，用于处理程序的逻辑，被称为寄存器。我们最常接触的寄存器有 EAX，EBX，ECX，EDX，ESI，EDI，ESP，EBP，EIP，PSW 这 10 个，前 9 个是 32 位的，PSW 是 16 位的。CPU 一旦加电，就开始永不停息地运作，它每个操作的过程如下。

从 EIP 寄存器里取出一个数据，其含义是一个地址空间中的一个地址。然后，CPU 加载这个 EIP 指向的若干字节的数据组成的指令，随之将 EIP 递增到下一个字节的地址处，继而按照指令逻辑作相应的操作。

一个指令分三部分：操作、参数、参数定位方式。这些对于 CPU 是用数字组合的形式表达的，但汇编语言中，我们用一些字母缩写来表示要做的操作，称之为助记符，比如 MOV 就是英文 move 的缩写，表示赋值的含义。

参数可以指定寄存器或者内存，某些改变寄存器的操作在汇编语言中是隐式表达在助记符里的，比如 RET，它实际会修改 ESP 以及 EIP 的值，但是，并没有在语句里写明参数。而参数定位的方式，汇编里可以有很多种，比如从寄存器里读取参数，从内存里读取参数，从当前指令的数据中读取参数等，在汇编语言的术语中称之为寻址方式，它是利

用一些符号和指令书写的形式来表达的。比如 MOV EAX,1 就是从指令本身中读出一个参数，这里是 1 这个数字，把它赋给 EAX；而 MOV EAX,[EDX]指把 EDX 作为一个地址，把这个内存地址中的数据赋给 EAX。这两种寻址方式就是用[]来区分开的。

3.4.3 寄存器

现在来看一下上面提到的 10 个寄存器。

EAX，EBX，ECX，EDX 四个是通用寄存器，它们的用途最为广泛。在 x86 系列 CPU 的早期，它们的分工还相对明确，但发展到 386 以后的时代，大多数情况下，只是使用惯例上的不同，实际工作中经常可以换用。只是少数指令中，这些寄存器有它们特别的用途。例如，EAX 专用于无符号乘法，EDX 用于保存乘法结果中的高位数据，ECX 用于循环计数或者移位指令。

我们可以让指令只使用 EAX 寄存器的一半，这个时候，AX 这个别名就表示了 EAX 寄存器的低 16 位。而 AX 的低 8 位和高 8 位同样可以被单独使用，其代号分别为 AL 和 AH。顾名思义，AL 指 AX 的 Low 8 位，AH 指 AX 的 High 8 位。另外三个通用寄存器有同样的特性，在书写时只需要把 A 换成 B，C，D。

ESI，EDI，EBP 三个通常被用于储存内存地址，但它们的实际可运用范围并不比上面四个通用寄存器窄。大多数情况下，也可以与通用寄存器互换。x86 指令集里有一些单一指令可以做相对复杂的操作，ESI 多被用于连续内存操作的源地址指针，EDI 被用于目标地址指针，而 EBP 用于堆栈上的数据指针。

这三个寄存器同样可以只操作低 16 位，其代号分别是 SI，DI，BP，但没有方法去独立操作低 8 位。

以上的 7 个寄存器可以根据编程的需要灵活使用，没有必要拘泥于特定的用法。我们把它们看做供代码快速操作数据的临时变量就可以了。

EIP，ESP 是两个特殊寄存器，它们没有上面的寄存器那么通用。

EIP在上文已经讲解过，修改EIP的值，在高级语言的概念中，就是程序逻辑中的跳转，在汇编助记符里，我们也不再让MOV这样的指令可以改变EIP的值，而是使用另外的代号，比如JMP，英文跳转（jump）的缩写。但在本质上，你可以把JMP xxxxxx看做MOV EIP,xxxxxx。

ESP是堆栈指针，它和前面提到的EBP不同。EBP只是在惯例上指向堆栈上的数据，实质上你也可以用别的寄存器来取代它；而ESP，许多操作堆栈数据汇编指令会隐式地改变它的值。

通常，一个进程的堆栈处于整个地址空间的最低处（这个通常指单线程程序，或者指多线程程序的主线程），用掉一定的堆栈空间，ESP就会递减，一旦ESP减到0发生环绕后，就引发了堆栈溢出。PUSH，CALL这样的指令都会把一些数据放入ESP指向的地址，并减小ESP的值，而POP，RET这样的指令会做相反的操作。

PSW是唯一的一个16位寄存器，它的作用更为特殊，里面是按位来保存CPU的一些内部状态的。其中有6个条件标志位：

- OF是溢出标志，当某条指令结束后，由于计算溢出，会把这个位置为1，反之置为0。
- SF是符号标志，当某条指令结束后，由于计算结果为负，会把这个位置为1，反之置为0。
- ZF是零标志，当某条指令结束后，由于计算结果为零，会把这个位置为1，反之为0。
- CF是进位标志，当某条指令结束后，由于计算产生了进位，会把这个位置为1，反之为0。
- AF辅助进位标志，记录运算时第3位（半个字节）产生的进位。有进位时置1，否则置0。
- PF奇偶标志，结果操作数中1的个数为偶数时置1，否则置0。

具体指令是否影响标志位，以及怎样影响标志位在相关的汇编指令手册中都可以查到。很多情况下，我们将影响标志位的指令和根据标志位做跳转的指令结合起来用，例如：

```
CMP EAX,EBX
JNZ label
```

第一句是将 EAX 和 EBX 做减法，但是结果只影响标志寄存器，而并不将结果放回 EAX。

这条指令将影响 ZF（并不仅仅影响 ZF）。

JNZ 是英文 jump not zero，如果非零则跳转的缩写。即当此时 ZF 为 1 时就修改 EIP 到地址 label 处。这里需要注意的是，条件判断语句的跳转地址不同于 JMP，它们的指令数据中是用一个有符号整数保存的相对地址。

其真正的含义是，当条件成立时，将 EIP 累加一个值，而不是将 EIP 赋予一个新值。所以，条件跳转往往不能跳过一个过大的语句块（大于 128 字节），我们一般把它和 JMP 结合起来用。庆幸的是，VC 的内嵌汇编会自动做这种转换，即在需要长跳转时，把 JNZ 转换成 JZ/JMP 的组合。

另外还有 3 个控制标志位：DF 方向标志，IF 中断标志，TF 陷阱标志。这里只介绍最常用的 DF，它一般在串处理指令中控制处理的方向，即操作完毕后，地址指针寄存器是做累加操作还是递减操作。

3.4.4 寻址方式

一条指令的寻址方式决定了这个操作如何利用指令中所携带的数据去寻找真正的操作数据。这里，为了便于理解和缩短篇幅，并没有按常规教科书上对寻址方式进行细分，而是按最直观的理解分做 4 大类。

（1）立即寻址：真正的操作数就包含在指令当中。

```
MOV EAX,1
```

这条指令的写法就是立即寻址，操作数就是 1 ，指令的含义是把 1 赋给 EAX。

（2）寄存器寻址：操作数据在寄存器内。

```
MOV EAX,EBX
```

这条指令的写法就是寄存器寻址，操作数放在 EBX 内 ，指令的含义是把 EBX 内的数据赋给 EAX。

（3）直接寻址：操作数的地址被直接写在指令内。

```
MOV EAX,[0x1000]
```

这条指令的写法就是直接寻址，操作数放在0x1000指向的内存地址内，指令的含义是把地址为0x1000的数据赋给EAX。

（4）间接寻址：操作数的地址需要经过某种运算才能得到，这种运算可能是从一个寄存器中读取，也可能是把某几个寄存器中的数据相加，等等。

MOV EAX,[EBX]含义是把EBX里的数据作为一个地址，然后从这个地址中得到数据赋予EAX。

MOV EAX,[EBX+4]含义是把EBX里的数据作为一个地址，将这个地址加4后得到新地址，从新地址获取数据赋给EAX。

MOV EAX,[EBX+EDX]含义把EBX里的数据与EDX的数据之和作为地址，从这个地址获取数据赋给EAX。

MOV EAX,[EBX+ECX*4+4]相信读者能猜出其含义了，但这里ECX * 4并非表示CPU在做寻址操作的时候可以同时做乘法，而只是将ECX扩大了4倍。在间接寻址中，写在[ ]内的最多只能是3个部分的和，你可以理解为基址、变址和常数偏移。

还可以用C语言的思想把它理解成一个数组，EBX就是数组的首地址，4是对这个首地址做的一个常数修正（设想一个二维数组，计算机内部实际会用一维线形空间储存它，由于数组的大小在声明时就已经确定，所以在访问数据时，编译出来的代码常常需要用一些根据声明的数组大小的常数对数组首地址做一些修正来操作数组中的某个变量），ECX就是需要操作的数组的序号，而*4表明了这个数组的元素都是32位长整数。x86处理器可以处理单字节、双字节、四字节、八字节的数据，所以这里在间接寻址时可以把变址扩大2，4，8倍。

3.4.5 汇编指令

原本我很希望在这里详细记录一下x86汇编系列的指令，但当我发现这将变成一个繁杂的复制、粘贴、编辑、整理的工作时，念头被打消了。

其实，有了上文的概念基础，有兴趣的人完全可以重新拿起以前枯燥无味的汇编教材重新研读，不用多久就能掌握基本的运用了。如果我在这里重复教科书的话，恐怕是徒占篇幅。

3.4.6 C/C++ 语言和汇编

C/C++ 都是非常贴近最终汇编实现的语言，它们可以很容易地和汇编共存。通常一个函数会被编译成一段汇编代码，函数指针就是指向这段代码的地址。函数参数是通过把数据压入堆栈传递给函数的，在C++里，为了提高效率，通常会把 this 指针置入 ECX 寄存器来传入函数，因为寄存器的访问速度要优于内存。而函数在运行时，编译器产生的代码通常并不会用 POP 指令将传入的参数出栈，而是修改 EBP 的值，指向当前堆栈，然后通过对 EBP 的间接寻址来访问参数，这样，在多层函数调用中，使得每层函数的参数都得到保护。同样，函数用到的局部变量也会被放入堆栈中，通过 EBP 来读写。当函数返回时，C 语言的默认做法是让调用者去处理传入的参数占据着堆栈的问题，而C++的默认做法是函数本身从堆栈中去掉这些参数。

函数的返回值一般会放入 EAX 寄存器中，某些情况会用到 EDX。如果返回的数据过大，比如是一个类或结构，许多编译器的做法是把返回值需要的数据空间由调用者在堆栈中分配好，把这个临时变量的指针传入函数，函数把结果填入指针指向的内存。

我们自己写汇编代码时如果想调用一个C++语言函数，则在调用前，如果 EAX，ECX，EDX 内的数据对你的程序的上下文有意义，那么请自己提前保存起来（C 语言可以不保存 ECX），而 EBX，EDI，ESI，EBP 这些寄存器的数据不会被破坏，ESP，EIP 则会按约定的逻辑起变化。

同样，如果你想用汇编写一个可供C++调用的函数，那么请遵守同样的约定。你的函数如果用到 EBX，ESI，EDI 等寄存器，请在函数入口处保存，并在函数返回前恢复。而 EAX，ECX，EDX 则可以任意使用（C 环境下，ECX 例外，需要得到和 EBX 一样的保护）。

如果你使用 VC 的内嵌汇编，那么恭喜你，你不需要去考虑烦心的

寄存器保护恢复问题了，但是需要意识到，内嵌汇编在某些极端的情况下的效率略逊于手工编写的纯汇编函数，其原因也在于编译器自动生成的一些代码。

3.4.7 小结

以上是对 386 保护模式下的汇编概况的一个初步介绍，如果需要进一步学习汇编，则需要有一本详细的教材和供查阅汇编指令的手册，这些都不难在书店买到。

汇编和C++混合编程的知识，在后面的章节还会有更为具体的介绍。

通过实践多用汇编写一些程序，多阅读一下C++编译器生成的汇编代码，绝对能够加快学习的步伐。

3.5 其他语言

其实，上个世纪电脑诞生之初，编程语言和计算机硬件本身一样丰富。没有特别的主机一统天下，也无法说哪种语言是占主导地位。

今天，不少人有学会了C++或者Java别的语言就会一通百通的感觉。说这话的人，言语中往往透着一种自信。而自信若是过了头，就很容易变成自负。将所有语言看成同类，未免有点坐井观天的嫌疑。

3.5.1 Forth

还有人记得 Forth 吗？“真正”的程序员见到 Forth 时都会为它着迷。Forth 兴趣组因为 Forth 的魅力遍及全世界。20 世纪 80 年代，Forth 被介绍引入中国，甚至国内还有这方面的书出版。在我的印象中，我母校的书架上就有长长一排《Forth 语言》的黄皮教材。90 年代初，发行量很大的《电子与电脑》杂志连载了 Forth 语言的介绍。我从那个时候见识了 Forth 语言，并一直在我的心中占据着一席之地。

没听过 Forth 的人赶快去 Google 一下这个关键字吧，从那些介绍文字，你会获得阅读的乐趣。

Forth 的所有都是浑然一体的，包括用户交互、编译、编辑、语言的

扩展。如果你掌握了 Forth 的思维方法，会发现控制计算机是如此的灵巧。Forth 语言本身仅仅定义了很小的一个语句集合和语法规则。整个系统都是在此之上迭代产生的。以至于 Forth 语言很容易做到自己编译自己，而且是实时地编译成本地码。如果想做到交叉编译，即从一种 CPU 上编译到另一种 CPU 上的目标代码也非常容易实现，并且可以让系统本身来自行做这种转译工作。

Forth 实际上描述的是一种计算机模型，并且可以和现有的 CPU 指令进行方便的转换。事实上，Forth 就有自己的专有 CPU（NOVIX），该 CPU 的“汇编语言”就是 Forth。

但 Forth 并不像传统意义上的汇编语言那样的“低级”，它允许程序员在其基础上迭代出程序员想要的语言的高级形式。

3.5.2 Lisp

Lisp 往往被人嘲笑成括号比实际有用的代码还多的语言。我在儿时曾经读过一本关于 AutoCAD 的书，里面提到了 Lisp 语言，那个时候的感觉是，Lisp 是简单的。

现在看来，似乎研究 Lisp 的人都是高手，尤其是从 C++高手回头再研究 Lisp 的人，都有高深莫测的感觉。我没用过 Lisp 做过开发（所以我不是真正的高手☺），充其量看过一些网文的介绍和少许的书籍。

Lisp 是 meta-programming 的鼻祖，或许用惯了类 C 的编程语言的人很难想像怎么用语言来定义 1,2,3 这样的数字和加减这样的运算操作。在这点上，Lisp 和 Forth 有一点点相似，据说，最早就有用 Forth 实现的 Lisp。但是，语言的思考方式却大为相同。

3.5.3 Java

放在这么一个小节来谈Java而没有把它跟C++并列起来真是有所不恭。我从来都不是 Java 迷，甚至早几年，对 Java 还有一些排斥思想。但是，人的想法并非一成不变的。随着对编程的理解，不排除我日后成为 Java 迷的可能。

Java在很多人的心目中，代表着跨平台、嵌入式开发。在许多Java程序员的眼中，又是开发便易的象征。我不想提Java的OO特征又或是模块化分明这些。于我，能对Java生出好感无非是两点：

（1）Java有天生的多线程能力，这来自于它的虚拟机的设计。对于游戏的应用，虚拟机跨平台的特性远远不及在虚拟机层次上实现多线程来得有价值。只有构架出Java机，才能真正摆脱OS的限制，真正意义上地实现多线程。Python这样的脚本语言也实现了类似的机制，但感觉上不如Java来得彻底。

（2）Java有原生的垃圾回收机制的支持。这同样得益于它并不试图把代码全部翻译成本机代码，Java可以在虚拟机的层次拥有控制内存的能力。正是垃圾回收机制，才可以真正摆脱动态内存管理中内存碎片的困扰。这也是Java适合嵌入式开发的缘故。可惜的是，我在PC平台看到的Java程序，无一不暗示着大量的内存占用；或许，让整个系统都搭建到Java系统中会更美好，但我就没有多少研究了。

C#或许是为了解决Java在实际运用中产生的问题的一种解决方案。看起来，C#比Java要美妙一些。既然我两者都没怎么用过，这里给个不恰当的评价，行家看过尽可一笑了之：Java更像一个实验室研究出来，有着浓厚的学术气味，外观优美超过了它本来应该具有的人们所期望的实用性；而C#则是工程师的结晶，外加了许多商业味道，被微软强力地推广。

3.5.4 Python、Lua、更多

历史总是螺旋上升的。最早的语言是贴近机器的，用以发挥机器的全部威力。然后人们又在适用性上下工夫，创造出解释型语言，用机器语言来解释一些符号，方便编程者实现自己的思路。而后，为了追求效率，编译型的高级语言相继诞生，让高级语言尽可能地翻译成机器码。

C++将纯编译语言的理念几乎发挥到了极限。

而近年来，运行效率在许多领域不再是最大的问题。为了解决计算机固有模型对描述问题方式方面的一些缺陷，新的语言如雨后春笋般地

诞生。它们中的大多数都以解释型脚本的形式展现出来，比较有名的就有 Python，Lua，Ruby 这些。

鉴于本书后面会有一章专门阐述脚本语言在游戏开发中的运用，这里就不再展开。提到它们是想提醒读者，脚本语言会像 C++这样主体开发用语言的一种有益的补充。它们反映出来的编程思想也略有不同。并不是像某些人所想像的，C++能够解决一切。适当的时候，我们会用不同的工具解决不同类型的问题。

前 Windows 时代

现在，Windows 一统江山，它帮我们隐藏了硬件的许多东西。这一章是对 10 年前知识的一个回顾，它们如今已经沉寂在历史中，或是被 Windows 掩盖。但是，作为喜爱刨根问底的程序员，我们有必要对 Windows 之前的时代有一些了解。

386 保护模式，VESA，Watcom C，DJGPP 这些名词或许还为很多人所熟悉。它们在某些领域有各自的用途。能在这一章中占据一个位置的它们还是有价值的。尤为重要的是，这里，我们搬出了老古董，Allegro，一个开源的游戏程序库。现在，它以一个跨平台游戏程序库的身份继续被维护。

Allegro 在我的游戏程序员生涯中有着重要的位置，它引导我学习了许多游戏程序中的基本概念：位图、精灵、RLE 压缩、数据文件等等。这里，我依然通过 Allegro 把这些概念介绍给尚未入门的游戏程序员。

最后一个短小的章节，介绍了中国惠多网。这个用 DOS 系统，电话拨号连接，通过单纯的信件交流的 BBS 网络，不知道还有多少人记得。cfido 是我一段美好的回忆，相信曾经沉迷于 cfido BBS 的老一代网民们会有所共鸣。

网络上现在依然可以找到许多怀旧为主题的网站。PC 时代的早期，甚至比现在更为朝气蓬勃。爱好者以自己的方式来玩弄计算机，就好像 Apple II 的年代一样。

对硬件控制的技巧仅限于新的 3D 显卡技术方面。而当 3D 引擎逐渐变成商品以后，硬件控制本身的东西正慢慢离大多数主流游戏开发者远去，似乎与软件开发的正统偏离太远。

但这些历史有过的游戏程序员们的激动，包括我自己，让我觉得有机会记录下来，颇为欣慰。或许知识本身已经对现在的人们意义不大。但我觉得，今人读起，会有一些乐趣的。

4.1 386 保护模式

几年前，我和一位在游戏制作的圈子里奋斗了近十年的台湾程序员聊天，他提到了两次游戏编程史上的变革，每次变革都使许多程序员退出游戏编程的队伍，甚至使一些小的游戏公司从市场上消失（大陆的游戏产业发展较晚，相比台湾，并不那么明显）。

这两次变革，一次是从实模式编程到 386 保护模式的转变，另一次是 DOS 平台向 Windows 平台的过渡。可见保护模式在 PC 发展历史中的重要性。

所谓保护模式，其实从 286 开始就有了。之所以叫做“保护”，是因为可以使计算机想同时干几件事情时，各自占用的资源得以被隔离和保护起来，不会因为一个程序的错误而破坏掉整个系统。

但是，286 在这点上做得还不完善，微软得以打下天下的先头部队 Windows 3.0 就支持工作在 286 保护模式下，不过，在日渐成熟的后续版本 Windows 3.1 中就去掉了对 286 保护模式的支持，转而全面支持 386 保护模式，可以说，所谓的 Wintel 联盟就是从这里开始的。

另一方面，保护模式提供了每个程序独立的 4G 内存的访问空间，这极大地解放了程序员的创造力。玩家们更会感激。

20 世纪 90 年代开始的游戏几乎都被一个叫做 DOS4GW 的 EXE 文

件所驱动，它就是最流行的保护模式驱动程序之一，提供了一整套用于建立保护模式环境，并分配、映射、管理内存的 API。使用 386 保护模式的新游戏，每个都激动人心。当年我的最爱有 DOOM，Strike Commander，Syndicate，SimCity 2000，UFO: Enemy Unknown 等许多许多，每一个都曾让我废寝忘食过。

PC 游戏种类的多样性带给我的冲击远远大过了当年的任天堂红白机，总结其原因就在于 PC 的内存远远大过早期的 Console 游戏机，可以存放更为复杂的数据，游戏逻辑能够变得更加复杂，更方便的编程方式也使得游戏开发变得大众化。只要你愿意，每个人都可以圆一个游戏梦。

今天，我们生活在 32 位时代（有迹象表明，64 位时代正在来临）。由于内存地址空间已不那么拮据，对后来扩展的更高分辨率、更高发色数的显示模式的编程就反而变得容易多了，可以说，现在的游戏编程的技术门槛已经比当年下降了很多，这对许多期望跨入 PC 游戏制作行业的程序员来说无疑是个好消息。

可惜的是，没经历过 PC 早期程序开发过程中的那种拮据感，许多的程序员对程序内存消耗的问题不屑一顾。仿佛有操作系统在后面撑着，32 位环境下，有 2G 的空间可以随便使用。

在许多人的观念里，游戏中要用到的数据只管加载进来，反正物理内存不够了，OS 自然会负责和外存交换，只需要保证程序的逻辑运行正常就行了。更有甚者，程序出了问题，发生了内存泄露，随着时间推移，内存占用总量缓慢增加，干脆就在崩溃时要求用户重新启动一次机器算了，对于 BUG 闭上一只眼睛。

以我多年的经验，对于游戏程序的设计，除了整体框架外，考虑得最多的无非就是时间和空间。有时候我们需要空间换时间，以突破速度的极限，有时又需要时间换空间，使程序能在合适的机器配置下流畅地运行。

现在的 PC 已不是当年的单任务操作平台了，即使是游戏程序，用户在大多数情况下也需要它能占用更少的 CPU 和内存资源，而不是独占系统。PC 在这个问题上，已经很优待开发人员了。如果有朝一日转去

当时，12 点学校放学，我就飞似地骑着自行车赶回家，有时也会有几个同学挤在我的小房间里，通常会忘记了吃饭。享受完快乐的午间一小时，再在大马路上飚车赶回教室上课。因为被看到大汗淋漓地坐在教室，和那些单元测试见不得人的分数，不知道被老师训了多少次话。

但是我至今仍旧认为那是少年时期最快乐的时光，因为极大地享受了游戏的乐趣。还有许多的伙伴和我一起分享，我们讨论战略游戏中的战术、动作游戏中的技巧，评论模拟游戏中表现，这些很多都是许多成年人不能理解的快乐。

console 游戏机平台上开发，机器的内存多是有限的，甚至没有硬盘允许创建出虚拟内存空间。用完有限的资源，就意味着崩溃。

4.2 VGA 到 VESA

4.2.1 超越 BGI

Borland 公司曾为它的系列编译器制作了一套通用的图形接口，叫做 Borland Graphics Interface，简称 BGI。这是所有使用 Turbo C 的人最早能接触到的操作计算机画面的函数库，我也是从 BGI 开始操作 PC 来绘图的。从某种意义上讲，BGI 设计得很好，它和设备无关，可以用通用的 API 在不同的显示模式上画点、线，直到各种几何图形。

在这点上，BGI 很像现在 Windows 下 GDI 实现的一部分功能。其实，理论上只需要有画点的函数，就可以表现出无限丰富的画面了，不是吗？事实上，当时我也见了许多图形界面的软件，真的是依靠 BGI 来实现的。但是，这个理论的前提是，我们不必考虑效率。

刚开始用 PC 的时候，我用了大约一周的时间，希望自己实现一个类似画笔的工具，以此来熟悉 PC 这个陌生的编程环境。于我看来，PC 真的比 Apple II 复杂了很多，一时不能给我带来一种简洁透明的感觉，就算我能熟练地配置 autoexec.bat 和 config.sys，把基本内存配置到 639K，又或者使用 Norton diskedit 去手工修复损坏的分区表，这又能说明什么？仅仅只是把别人提供的工具用熟，对系统表层的东西搞清楚罢了。

从 Apple II 带来的习惯，我真正需要的是把计算机里那些程序是怎样工作的弄清楚。所以，把这个画图的工具完成后，我再也不堪忍受看到绘图过程的低反应速度，放弃了最开始苦苦寻觅支持更高颜色数和更高分辨率的 BGI 驱动的想法，直接抛弃了 BGI。

我希望让自己的认识更贴近最底层，也正是如此，在如今的 Windows 时代，操作系统已经非常复杂，即使已经熟悉了大多数的 Windows API

在玩计算机的过程中，我认识了很多朋友，大家经常一起交流经验，互相传授各种软件的使用技巧。当时 PC DOS 非常简陋，出现大量的第三方扩展或改进软件，所以，即使不玩游戏，PC 也是有很多有趣的东西可以摆弄的，也有好些朋友生出自己做游戏的想法。但是，到今天，只有我一人从事游戏行业，可能就是源于我更希望了解的是软件背后的运作原理，而不是怎样使用。

调用，我依然只会用Windows界面上的有限功能，也头痛于千奇百怪的各种软件和操作系统的各种设置。

在公司里，我的计算机里是异常干净的，除了操作系统本身和基本的编程工具，几乎一无所有，连现在这些文字，也是用记事本写成的。:)

当年只用了不到半个月的BGI就改寻它途去处理画面，也是基于同样的理念，因为它绝非直接控制最终显示在屏幕上的数据，在其背后隐藏了太多我不需要的东西。

那时，标准的显示模式是VGA 640×480 16色的模式，由于效率和16位内存地址空间等的限制，这种模式被设计得相当奇怪（至少以今天的眼光来看是很怪异的）。标准VGA是一个8位设备，显存也有明显的等待状态，为了直接操作它还不得不和诸多的寄存器打交道，还有那些艰涩的概念，索引寄存器、数据寄存器、位屏蔽寄存器、位平面，等等。

我花掉了很长的时间，翻遍了书店里能够找到的所有相关资料，才勉强把VGA的控制原理和几个不同的读写模式弄清楚，然后迫不急待地完成了一个叫C-View的作品。这个程序可以不需要加载中文平台，就能在VGA标准模式下阅读中文的文本文件，而且用到了一点VGA编程的小技巧，做到了像素级的平滑滚动，使在计算机上阅读变得轻松很多。另外，汉字库的加载也是动态的，程序本身体积并不大，即使在软盘上启动这个程序，也无须忍受过长的读盘时间。这个小程序让我自己非常得意，它是对我学习VGA编程的一项验收。

现在的Internet上还可以Google到我所作的C-View这个软件，不过已不是上文中所提的那个版本。一个有趣的点子往往可以引发一个有趣的软件，自己会编程就是这么有趣。当初我就是想让文字能在屏幕上平滑滚动，并且可以动态加载字库以及超大的文件，让计算机阅读变得更舒适一点。

后来我用保护模式重新实现了这个想法，发布了C-View 2，并突发奇想试试收取区区10元的注册费，没想到真的有不少人汇来了10元人民币。后来因为我觉得当年的游戏之作给我带来的麻烦太多，经常需要跑到邮局去兑取10元的汇单，然后还要用手写的邮件邮寄出注册码（很多用户当时并没有上Internet），就干脆把这个软件免费了。我也逐渐失

直到1999年的时候，还有一位和我同住一个城市的70多岁，一人独居的老人家，坐了一个小时的公共汽车，然后步行几公里找到我家，为了交付这10元注册费给他使用的这个软件的作者，让接待他的我的家人感动不已。当时我不在家中，让他很失望，并且一再为买了盗版光盘才获得这个软件而表示万分歉意。盗版光盘成为传播第一代国产共享软件的主要途径，恐怕这也是中国特色吧 ☺

去了继续开发的热情，终于我没有实现对 C-View 用户的承诺，始终没有出一个 Windows 版本。

4.2.2 VGA

在 PC 显示设备的发展历史中，VGA 的地位无与伦比，它是曾经有过的使用最广泛的统一标准。在 PC 显示设备进入 3D 化以来，再没有一套标准能够统一整个市场。

在 VGA 上，我们可以确保 PC 的图像模式分辨率达到 640×480 16 色，或者 320×200 256 色，或者是没有官方技术文献记载但完全标准的 360×480 256 色（传说中的 Mode-X☺），任何一台支持 VGA 显示模式的显卡都可以用完全统一的接口来编程。但是，标准的 VGA 是一个 8 位或 16 位设备，其编程接口工作于 16 位模式下，所以，在今天 32 位操作系统一统天下的时候，我们已经极少使用 VGA 编程了。

VGA 通常会把显示内存映射到 16 位模式下地址 A000:0000 到 B000:0000 的 64K 空间下，而 VGA 标准的配置有 256K 显存。这使得大部分 VGA 模式并非平坦地对应于内存地址空间，控制起来需要操作 VGA 的若干读写端口，编程相对麻烦，唯有 320×200×256 的标准模式，是在内存地址空间上与显存逐字节对应的，也为最多的游戏开发人使用。而更专业的开发人员会依赖自己的技术，发挥 VGA 的性能到极限，那就是 Mode-X。

4.2.3 VESA 标准

相对于保护模式把游戏程序员从640K 内存限制提高到了4G，VESA 标准将程序员控制显卡的能力扩展，同样具有重要的意义。

背景：

VESA （Video Electronics Standards Association）标准在 DOS 时代的末期被广泛地使用，它给程序员提供的接口通常被称做 VBE（VESA BIOS EXTENSION）。即使是一些很古老的显卡也在 BIOS 中固化了

VBE 1.2 的接口，如果没有，通常也能找到一个可以额外加载的 TSR（内存驻留）程序来提供相关的服务，这些就是显卡驱动程序的前身。

使用 VBE，我们就可以轻易地突破原来 VGA 可以管理的显存的上限，使用更高的分辨率。VBE 1.2 最高可以提供高达 1280×1024 真彩（16.8M）色的支持。在 20 世纪 90 年代中期发布的 VBE 2.0 更提供了 32 位保护模式下的接口，使用户不必在 16 位实模式下，使用那可怜的 64K 地址窗口来访问几百 K 甚至上 M 的显存。

有过 DOS 下玩游戏经历的老玩家应该记得，曾经有过一个广为流传的叫做 UNIVBE 的小程序，不少游戏需要在运行前加载它，就是用于提供 VBE 2.0 接口的，那就是一个大量显卡驱动程序的集合。

当我拿到 VBE 1.2 的接口资料时，着实让我兴奋了好几天。因为之前我一直想自己写程序显示出更漂亮的画面。640 行的高分辨率下仅有的 16 种颜色显然不够丰富，而 320 行的 256 色模式又让图片显得过于粗糙。

透过散乱在各种书籍、报纸杂志上的零星文字，显示出有一种超过 VGA 的 SuperVGA 模式可以达到更高的分辨率，但是似乎每种显卡又各自为阵，我可不想在我的机器上能运行的程序换到朋友的计算机上就漆黑一片了。似乎 VBE 这个统一接口可以帮助简化为所有可能的显卡各编制一套程序的工作。

了解了 VBE 后，另一个发现却是关于色彩方面的。我惊奇地发现，如果显存足够，显卡居然可以以一种比 256 色调色盘模式更为简洁的方式表示像素的色彩，最直接的方式就是真彩模式。它用 3 个字节直接描述每个像素的红绿蓝三个分量，然后线性地排列于显存中，直接写显存就可以修改屏幕上对应像素的颜色。这不仅比以前碰到的 16 色模式需要分层控制要来得方便，比之 256 色模式也减少了调色盘的操作。

在熟悉了那些 API 后，我动手写一些有趣的程序、一些小游戏。但是没动手多久就发现行动过于草率了，即使 VBE 的接口再容易使用，还是和最终需要干的事情，如画一条线，贴一个图有所距离。如果将这些代码和游戏逻辑处理的代码混杂在一起，只会使程序变得更混乱。而

在很长时间里，我总是觉得更高级的东西控制起来会更麻烦。但在学习对显卡的编程时，却改变了我的这个看法。如果把 16 色、256 色、真彩模式相比较的话，最简单的却是真彩模式，而 16 色模式反而最为繁琐。从这也可以看到，随着硬件的发展，存储器、控制芯片成本的下降，硬件提供商们似乎有一种趋势，就是尽量提供一种更直白通用的接口来简化软件开发人员的门槛，而不是为了挖掘硬件的潜力去设计复杂的接口。

我们有理由相信，以后的游戏设计在接近系统底层的部分，会变得比今天更简单，不再需要当年那样，被大多数的游戏程序员所关注。

且这部分代码非常具有共性，几乎每要写一个小游戏，都需要用到。多写几个程序就能总结出哪些代码是共通的，提取出来放在一起，做成一个函数库，就成了一个最原始的游戏图形引擎的雏形了。

4.2.4 花絮

高中三年，和众多同龄人一样，我面临着巨大的升学压力。在湖北，几乎历年全国统一高考中，理科都拥有全国最高的分数线。为了保证升学率，我们在高二上学期就结束了本应在三年学完的课程，然后进入漫漫的应试复习。大部分时间作业负荷都很重，每天需要早晨 7 点出发去学校，然后晚上 9 点放学，周末的休息被压缩到半天，还需要完成大量的习题。

当时我对这种强化训练般的学习非常反感，学习成绩也很糟糕，除了数学和物理能偶然考出很好的成绩外，别的科目都徘徊在及格线边缘。为了能升入大学，只能寻求别的途径，当然那也只能是信息学奥赛。

不过搞奥赛也并不轻松，一样需要残酷的习题训练，我在省内的辅导老师夏老师一直有种恨铁不成钢的感觉。当我谈起一些他当时并不了解的诸如保护模式、真彩模式的编程方法时，夏老师总是一番感慨地说，你不如考到我们学校来，以后跟着我接一些活做，肯定挺有“钱”途的，就是这竞赛怕是搞不好了，心思太野。

果然如他所料，几次的赛场发挥都不理想，直到高三快毕业时才取得了个好成绩，但是已经晚了，因为全国选拔出来的选手必须再经过一年的集训才有可能出国参加国际奥赛，而当时我已经临近高中毕业，只有把这个机会让给师弟了。然后便得到了确切的消息，不可以利用奥赛的这个成绩保送去大学。历史就是这样重演的，初中升高中时被我碰上了一次，高考前一个月又不得不放弃免试保送的幻想，全部投入心思去复习备考。

高考前，学校放假一周，让学生在家里自己复习，压抑了很久的我突然觉得非常轻松，鬼使神差地在这一周泡在屋子里打起一个刚出的 RPG 来了，白天就不停攻关，晚上看书，累了就睡在床上回忆游戏的剧

情，感慨主角的曲折命运，刚好在第七天顺利完成了这个游戏，而对于三年中学到的知识也有了更多的自信。或许就是因为这考前的放松，高考成绩出乎意料的好。

整整三年，我在班上 64 个同学间，任何一次考试都没有排到过 32 名之前，而这次却排到了 11，分数超过了重点大学线。更神奇的是，最后一门物理考试之前，不知是由于紧张，还是因为意识到苦难的高中生活即将过去而不由自主地兴奋，我经历了人生第一次失眠。带着涨红的眼睛，却在考场上春风得意，几乎拿了个满分，而那年的物理试题是被媒体评论为难度太高的。

我的班主任兼物理老师，对这个沉迷于游戏和编程的无可救药的学生刮目相看，真不知道我那几年是为游戏所累还是被之所救啊。

4.3　保护模式下的开发工具

用过 Turbo C 系列做软件的早期程序员都有一种四处受制肘的感觉，BGI 中提供的基于图片的操作之所以不能处理过大的图片，也是由于实模式下 64K 的限制。当游戏需要的数据量越来越大时，光是使用超过 640K 的内存也需要特别的技巧。

第一个被广泛使用的 32 位 C 编译器是著名的 Watcom C ，但是，在 Watcom C 把持天下的年代里，我并没有成为职业的游戏程序员，甚至没有想过以后靠写游戏程序吃饭。那个时候没有跟上潮流，却也并没有蒙受多少损失。当我意识到保护模式下编程的重要性时，Watcom C 已经逐渐退出历史舞台，放在我面前的最佳选择是 DJGPP。

关于 Watcom C/C++

PC 历史上曾经出现过 4 个同时被广泛使用的 C/C++编译器。Borland C，Microsoft C，Symantec C 和 Watcom C。在那个年代，至少在中国，Borland 的 C 被最为广泛地使用，而 Watcom C（如图 4-1 所示）在专业性方面却有最好的名声，这源于出品 Watcom C 的那家加拿大的小公司

在编译优化上下足了工夫。当时最有名的出品 DOS Extender（为 DOS 扩展支持 32 位保护模式）的厂商都指名要使用 Watcom C。Watcom C 一度被认为是可以编译出世界上最快的执行代码。



图 4-1 Watcom C/C++

不知道有多少人还记得一家叫 SciTech Software 的公司，他们曾经为许多老显卡做过新版本 VESA 的驱动程序，当年很多 DOS 游戏需要的一个叫 UNIVBE 的内存驻留显示模式驱动，就是这家公司的产品。正由于他们在显卡硬件上的造诣，曾经推出过一个叫做 MGL 的通用图形库，其在 DOS 环境下，就类似今天 Windows 下 DirectX 的地位。MGL 的最初版本要求使用 Watcom C，并被很多游戏公司用来制作游戏（MGL 至今依旧被维护，可以用于 DOS/Windows/Linux 的跨平台使用显卡的硬件加速特性）。

从这些，我们都能看到 Watcom C 一度的辉煌。可是，进入 Windows 时代后，比编译效率更重要的是易用的 framework，而 C++ framework 并非一朝一夕可以完成。为了抢占 Windows 开发市场，时间不允许自己开发一套，可供选择的只有 Borland 的 OWL 和 Microsoft 的 MFC。最终，Watcom C 向微软签约 License，让 MFC 作为其 C++在 Windows 下的 framework，同时，Symantec C++也选择了 MFC。Borland 的 OWL 立刻处于孤立的状态，没过多久就被淘汰。framework 的主导权落到对手身上时，其下场可想而知，Watcom C 被 Sybase 并购后，逐渐退出了程序员的视线。

有趣的是，这些年来，一直有网友向我询问有关 Watcom C 编译器

的事情，希望使用 Watcom C 来开发游戏。在家中整理那积满灰尘的旧书时，也曾翻出国内唯一翻译出版的 3 本一套关于 Watcom C 的参考书，不禁感慨其对于游戏程序员的影响力。有趣的是，这两年，出现了一个叫做 Open Watcom 的开源编译器，仿佛也预示着 Watcom C 的新生。看来，也只有开源项目才能对抗如今的微软帝国了。

DJGPP 是一个叫 DJ Delorie 的人想把 UNIX 平台上著名的 GNU C/C++移植到 DOS 平台上所启动的一个工程。当然，其发展不久就囊括了几乎所有的 GNU 开发工具。无疑，这得益于开源社区的力量，DJGPP 中内附了一个 DPMI Server（cwsdpmi），也是免费的，这样，使用 32 位保护模式就不需要花钱去购买别的类似 DOS4GW 这样的商业产品了。

Allegro

我第一次听说 DJGPP 是因为 Allegro 这样一个开源的游戏程序库，而关于 Allegro 的消息则是从 cfido BBS（关于 cfido BBS 见 4.5 节）的一个站台上下载的零星资料。当年在 cfido 上并没有太多关于游戏编程方面的深入讨论，甚至也没有一个游戏编程的专区。和我有同好的朋友辗转在编程区和游戏区讨论，在编程区讨论编程，在游戏区讨论游戏的设计，虽然只是些纸上谈兵，但也是兴趣盎然。而在没有互联网的时代，cfido 上的文件共享是唯一的淘金处，我在里面淘到了 Allegro 2.1。

有着自己开发过一些简单的游戏雏形程序的经历后，我对显卡等硬件的编程开发有一种敬畏的心情。直接对硬件编程真的是要经过有如地狱般的洗练，尤其是在资料欠缺的环境下。很多东西都只能靠编程尝试，死机重启那是家常便饭，甚至有些试验是要冒着损坏硬件的风险，至少很多书上这么警告，当然我从来都没有碰到过。

从文档上看，Allegro 几乎实现了所有游戏编程必须实现的东西，支持所有的显示模式，方便地控制鼠标键盘，甚至有完整的各种声卡的驱动。这些繁琐危险的工作都有人做好了，对于一个想开发游戏的程序员，可以轻松地避过技术难题，是一个多么让人兴奋的好消息呀。

在 cfido 的武汉本地站上，我和许多新认识的朋友热烈地讨论了好几天的 Allegro 这个新宝藏，但是没有人能把它用起来。因为 Allegro 是工作在 32 位保护模式下的，需要 DJGPP 这个陌生的开发环境。找到 DJGPP 的 1.x 版已经是 1997 年寒假的事情，在一张关于 Shareware 的

CD 集中，我发现了寻找很久的 DJGPP，可惜我还是无法让 Allegro 工作起来。

1997 年的秋天，我就读的大学终于向学生开放了网络中心，可以上网了（就在前一年，我还认为 Internet 遥不可及，仿佛科幻小说中才存在的东西）。从网络中心开放的第一天起，我就把所有的零花钱都换成了上机票，几乎全部的时间都泡在机房里。在身边的同学沉迷于 MUD 世界中时，我在各个 ftp 上淘金，终于下载到一套完整的 DJGPP 2 和 Allegro 3。

Linux 在中国就是那个年头风行起来的，我也不能免俗地玩起这个东西。和很多人一样，就是玩玩 shell，装点软件试试，学校里的第一个 MUD 就是我向校方争取了主机的空间安装起来的。当时的编译错误一大堆，让我调了好一阵子。可笑的是，直到 2004 年底，我都没玩过 MUD 和任何一个后来的图形网络游戏，包括自己参与开发的。

当然，青春年少的我也做过黑客梦，那种入侵主机的黑客。制作自己的字典文件去强行攻击本校和教育网上外校的主机，或者从网上搜索得来的一些关于系统漏洞的信息，来试图进入别人的主机。现在想起来，早期的互联网真的是漏洞百出（当然现在也一样）：一个小毛孩只依赖一些互联网角落里不知道谁写的所谓黑客教材和一点点编程知识，就能去众多大学的主机上签上“到此一游”。

当时我写了几个 shell 脚本，上传到学校和另外一些学校的主机，在后台下载我需要的软件。这事被老师发现了（他们通过核对学生上机记录找到我的名字），我因此被禁了两天网，之后觉得非常无趣，再也没做过这类事情。

DJGPP+Allegro 再配上 Rhide 这个类似 Borland C++ 3.1 风格的 IDE 后，可以说是 DOS 下的终极游戏开发环境了（如图 4-2 所示）。深入了解 Allegro 源于我需要在互联网上做个主页，一开始，做个人主页的确仅仅是需要，而不是兴趣。因为通过教育网上国际互联网是按流量计费的，而且费用极高。

后来，我发现了学校 BBS 的一个漏洞，可以通过这个漏洞取得主机上很高的权限。我马上通知了网络中心的老师，补上了漏洞。可是，事隔不久，就被校方管理员怀疑曾经利用漏洞做过坏事。那只是因为某人曾经在学校主机上留过我的名字！

自从那次无故蒙冤后，我就愤然辞去了在网络中心的勤工俭学工作，再没有在学校网络中心上过网。



图 4-2　开发环境

学校网络中心屏蔽了国外的 IP，而刚开始，没有一家国内的公司提供免费的 E-mail 服务，只有一家新公司，网易，承诺如果使用他们公司的另一项免费服务，制作一个个人主页，就可以附送一个 E-mail 信箱。

在网易还是一个小公司的时候，这个方法聚集了大量的人气。1997 年以前，几乎所有活跃的简体中文网络资源，都是放在网易的服务器上，我也是贡献者之一吧☺（中国大陆第一代网民大多对网易有特别的好感，我也是这样，这间接促使了后来我为网易工作，参与完成了如今网易最大的赢利项目）。

个人主页构建之初，我罗列出很多个人兴趣，比如编程、围棋、游戏，很多很多。然后马上就发现这样大而全的结果是：我一样都做不好，而且和其他人的个人主页相比毫无特色。稍加考虑，就立刻砍掉了除了游戏开发之外的所有栏目，我要做中国大陆第一个关于游戏开发的主题站。

主页建设之初，需要有一个主打的连载内容，这样才能长期吸引人浏览，很自然的，我选择了翻译 Allgero 的文档，一个 300K+的格式文档。Allegro 的作者自定义了一种文本格式，并用它书写文档，然后可以用程序处理这些文本，转换成需要的格式。为了保持原汁原味，我选择了逐行对译，并保留其中的格式信息。为了做这个翻译工作，甚至先做了一个辅助翻译工具，可以对着每行英文在下面写译文，并且自建一个词典库，把当前行的生词显示在屏幕最下方的状态条上（如图 4-3 所示）。

有时候我也想过，如果当年是搜狐提供给我一个 E-mail 信箱，是不是我现在就在 sohu 做网络游戏开发呢？事实上，那时搜狐刚刚创建的时候，也同样给了我不错的印象，他们一开始希望做一个类似 Yahoo 那样以搜索 engine 为主导的公司，在我把自己的主页提交给搜狐的时候，曾经收到过一封手工回复的 E-mail，对我的主页的内容做了少许的赞赏。

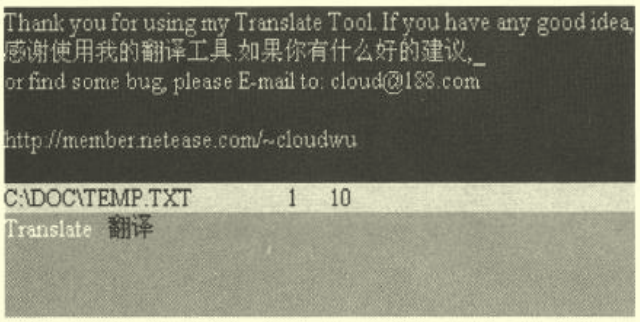


图 4-3 自制的辅助翻译工具

翻译就这样有条不紊地进行着，不太明白的地方，就阅读 Allegro 的源码。随着翻译的进行，不知不觉中，我对 Allegro 有了更为细致的了解。通过在主页上连载译文，还交到了不少的朋友，同时写的一些关于游戏编程技术方面的小文也被各处转载。个人主页慢慢有了些名气，一些国内的游戏公司主动联系过来，在那几年，一有假期，我就全国各地四处走访网上认识的从事游戏开发工作的朋友，慢慢地踏上制作游戏这条不归路。

学习新的技术，翻译一本相关的英文著作可以算是一条捷径了，比囫囵吞枣地读一遍英文原文要有效得多。因为责任感，使你必须用心地搞清楚每一个句子的意思，以免译错而误导阅读你的译作的人们。即使以前对此有所了解，翻译完成后，也一定使自己的理解更上一层楼。

其实，技术书籍的翻译，英文能力是很次要的，相关技术知识的掌握尤显重要。我在翻译 Allegro 的英文文档时，有些专业词汇不甚了解，就停下来找其他的资料把它们弄懂，学到了不少知识，这种经历使我受益匪浅。

我的英文水平，从前一直非常差，大学里第一次四级考试只考了五十九分，当我花掉一年时间翻译完这篇文档后，第二次参加四级考试居然拿了七十多分，而且中间并没有参加刻意去背单词或是复习语法，做模拟考题，连我自己对顺利通过 CET-4 都觉得有点不可思议。或许真的是零零碎碎的翻译工作潜移默化地提高了英文的阅读能力吧。真的是意外收获。

4.4　闲话 Allegro

Allegro（http://www.talula.demon.co.uk/allegro）是一个现在依旧被维护的跨平台的游戏程序库，从第 4 版开始甚至支持了 Windows 平台。它的存在和发展给过我许多的启发，教会了我很多游戏编程方面的知识，甚至我自己也参与过其中很少一部分代码的编码，很自豪地在作者名单中占据一席之地。

虽然进入 Windows 时代后，我已经不再使用 Allegro,也不向朋友推荐使用它。但是，我的硬盘上依然保留有一份 copy，有时，还需要参考它的部分代码。所以这里，我觉得有必要给读者介绍一下这个包含了许多人智慧的免费程序库。

不仅仅是介绍历史，更多是利用它来解释一些游戏编程中经常会遇到的概念。

4.4.1　用 C 与汇编写成的程序库

Allegro 是用 C 和汇编写成的，大部分代码是 C 的，这使得其具有很好的移植性。曾经有一个朋友和我通过 E-mail 交流时抱怨过 Allegro 是用 C 而不是 C++写成的。在 C 还是 C++这点上，我偏向 C++语言，因为 C++可以提供和 C 相同（甚至略高）的效率，并使程序更为干净整洁。但是，由于 C++语言的复杂性，现在各种平台各种厂商发行的 C++编译器都不能完全支持 C++标准也是事实。

在一些速度的瓶颈上，Allegro 使用了汇编（AT&T 格式），这样保证了它的效率。在使用汇编这个话题上，太多有经验的程序员陷入了高级语言的圈子中，他们认为，只要算法优秀，就能保证性能，但是，我们不得不面对在最底层的操作上，同样的算法用汇编写和用 C 语言写还是有区别的，当它变成瓶颈时，这种区别就会被放大。

比如，用一个 for 循环去复制一块内存，就永远比 memcpy 要慢，因为 memcpy 是编写 CRT 库的人手工写出的为 CPU 特别优化的汇编代码（另一个原因是大多编译器会特殊处理 memcpy 这个函数）。

4.4.2 BITMAP

作为一个游戏程序库，图像处理方面是非常重要的。Allegro 给出了 BITMAP 的概念，这也是几乎所有和图像有关的程序都支持的概念。中文将其翻译为位图，意即由位数据构成的图片。

在 DirectX 里，同样的概念被叫做 Surface，别的图形引擎中可能被叫做其他的名字。这里我们把这种包含一个线性数据空间，以等宽字节描述一张平面图形的对象通称为位图。在大多数图形引擎里，位图是一种最常见的对象，同时具有读和写的能力。即，可以把自己绘制到另一个对象上（通常是另一个位图），也允许别的对象绘制到自己上面。

其实，在实现一个有图形显示的程序中，只有极少的对象需要第二种能力，大部分的对象自身的数据不必被修改，只需要从外存加载，或者一次渲染生成。我在设计自己的图形引擎的时候更偏向于把位图按职能区分成两种，只接受别的对象绘制到自己身上的叫做画布（canvas），只会把自己绘制在画布上的叫做图片（picture）。

这种分法比按物理实现上的区分方法可以让程序编制时有更清晰的条理，虽然它们本质上可能是同一种东西。但是，也可以不同：canvas 在实现上可以并不是一个平展的位图，甚至可以只是一个绘图操作的记录器，并没有真实的数据空间。而 picture 也可以是一组被压缩的数据，甚至是一些矢量信息。在后面的章节，我们会就这种情况做进一步的讨论。

Allegro 中的 BITMAP 是一个结构，里面包含了一个矩形图片的内存，这块内存以连续字节的形式储存。当然，这块内存不一定是内存，也可以是显存，甚至直接映射了屏幕，或者可以共享另一张位图数据的一部分。

想在位图上开一个窗口，在上面绘图时不超过这个窗口，把这个窗口的映射为一张子图是个好主意。或者，我们可以为位图设置一个剪裁矩形，所有的绘图操作都会检查是否落在剪裁矩形之内。

实际上，即使不在位图上开窗口，剪裁矩形也是必不可少的，因为在计算机的内存中，物理只存在一种数据结构，就是一个一维线形的数组，想模拟出二维的数据结构，必须加上数据宽度的限制，而且一旦操作时越界，程序的错误由此而生。默认的情况下，剪裁矩形和创建出来的位图的完整尺寸一样。

不过，位图被创建出来的实际大小很可能不等于我们使用时看到的逻辑大小。这是因为在现在最流行的CPU上，按4字节（即32位）访问内存效率最高。所以，位图的每行如果不是4字节的倍数，那么会在创建时多分配一些内存出来，凑齐每行为4的位数。如果位图被创建在显存中，根据一些显卡的限制，行宽是不可以任意设置的，同样造成了逻辑宽度和物理宽度不一致的情况。

Allegro为了处理这些情况，位图在内部数据组织上，实际利用额外的一个一维数组记录了每个扫描线的起点，这样甚至可以处理一张位图分布在几处的情况。还记得以前提到的，老版本的VESA标准中，需要在内存地址空间上开一个窗口分区域映射显存的做法吗？

对于早期不能一次性映射完全的显存，这是一种变通之道。图形引擎就是要封装住这些物理上操作的不同，在逻辑上变得统一（在DirectX里，解决一些老式硬件的兼容问题的手段移到了操作系统的底层，用户看到的永远是连续的地址空间。用户的接口上换了一种更简洁表示方式，即只为Surface提供逻辑宽度和物理宽度pitch——通常翻译为跨度，供程序员使用）。

Allegro中还定义了一个特殊的位图，叫做screen，直接映射了显存中屏幕显示的那一块。操作这个位图就可以改变屏幕上的图像。甚至，screen这个位图可以比实际的屏幕大，然后调用卷动屏幕的API改变屏幕可见区域的映射起始点。

实际上，真正的游戏中，我们是不主张直接对屏幕上直接可见的部分做琐碎复杂的操作的，也极少利用改变显示区域在显存上的映射来滚

DirectX中类似的概念是主表面primary surface。但是，由于Windows下每个程序已经不能独占系统的所有资源了。考虑多个任务合作的情况，在操作主表面以及其他显存中的表面时，都需要进行Lock和Unlock的操作。Allegro自从支持Windows后，也加入了类似的操作。

动屏幕（虽然那样会很平滑）。因为前者使绘图过程被玩家看见，而后者的限制太多。况且现在显存间的数据传输已经非常快，那些硬件滚屏的小技巧已经远离我们的时代（后面的章节会看到，方法上的改进带来的速度提升远远胜过那些硬件技巧）。

1．双缓存

直接向屏幕画东西是初学者最爱犯的错误，对于我也是。正是Allegro 教会了当年作为初学者的我双缓存的概念。

我们可以在屏幕缓存之外创建一个额外等大的缓存。一般的想法是将这个缓存创建在显存屏幕之外额外的空间里。平时作图的时候，把图像绘制在看不见的这个缓存上。等一帧图片全部绘制完毕，修改屏幕和显存的映射关系，让刚才看不见的缓存显示出来，然后让原本的屏幕映射的显存变为不显示，下一帧我们将在这里作图。整个游戏中交替这个过程，动画将变得很平滑（DirectX 把这种方法用到的操作叫做翻转 flip，封装成特定的 API 做这件事情）。

当然，我们也可以不采用切换屏幕映射的方式，而是每帧都把图画在看不见的后台缓存（back buffer）上。需要显示的时候，把整个后台缓存的图像复制到屏幕，这种位图间的数据传输被叫做 Blit。显存间的 Blit 操作已经非常快，所以，现在的显卡上几乎已经分不清两种方式的区别。

理论上，从后台缓存传输数据到屏幕缓存的速度再快，也是需要一点时间的，为什么人眼察觉不到呢？这是因为常用的 CRT 显示器是以一定的屏幕扫描显存，把显存中的数据表示在屏幕上的。每扫描一次，都会有细微的停顿，我们只要不在扫描过程中传输数据，那些图像数据的变化就根本不会被表现在屏幕上。翻转操作由于硬件的限制，不会发生在屏幕回扫的过程中，但是，位图传输操作却有可能受其影响。

Allegro 提供了等待当前次屏幕回扫的 API，但就我的经验，这种等待不是必要的，即使不等待，顶多也就是偶尔出现屏幕图像稍微撕裂的现象，玩家并不会太抱怨什么（DirectX 也提供了类似的 API，但是，某些显卡在为 DirectX 写驱动的时候，并没有忠实地实现这个函数，至少我在自己的一块很古老的显卡上发现过问题。或许显卡厂商觉得必要性

不大，就偷了个懒）。

这里顺带提一句，如果你采用翻转技术实现动画。在向显卡申请翻转操作时，由于要等待屏幕回扫，两个缓存都变得不可写。有人为了节约这一丁点时间，就在第三个缓存上作图，然后让三个缓存轮转，这就是三缓存技术。传统的 VGA 是不可能支持三缓存的。但现代显卡在理论上，只要显存足够，你想创建多少个缓存轮转都可以。只是一般情况下，屏幕回扫仅仅是那么一瞬，超过三个缓存几乎是无意义的事情了。

把缓存创建在显存中真的好吗？对于不需要显卡直接在显存中渲染多边形的平面游戏来说，我的回答是“不”。

因为，如果从显卡的发展历史来看，CPU 读取显存的相对速度是越来越慢了。我们不能像操作内存数据那样自由地操作显存上的数据，做一些比较复杂的运算。即使是向显存写数据，也没有写内存快，所以把后台缓存放在系统内存中是个不错的选择（当然这一点会慢慢由于显卡技术的发展，例如 shader 技术，所逐渐弥补）。

当每帧图片绘制完毕后，再一次把整个缓存传输到屏幕。这个过程虽然要慢一点点，但是却是可以接受的，换来的灵活性是毋庸置疑。更何况采用巧妙的方法，还可以尽量减少数据传输量，把缓存尖数据传输的数据损耗降到几乎为零。当屏幕分辨率很大的时候，由于显存的限制，可能后台缓存放在系统内存中是唯一的选择，而大分辨率下，整屏的数据量非常大，游戏速度的瓶颈往往就在双缓存间的数据传输，就我原来对某个参与开发的游戏测试而言，这块消耗的时间居然超过了全部时间的一半，所以进行优化必不可免。

很多职业的游戏程序员都碰到过这个速度瓶颈，很多人放弃了把后台缓存放在显存中的做法，但是我在后面的章节会为大家证明，不是所有的图像显示的速度要求都必须寻求硬件的解决之道的。

4.4.3　Sprite

在讲解游戏编程的书以及诸多游戏图形库中都会被反复提到。在很多情况下，精灵就是用一张位图实现的，用来表示游戏画面中的某个独

立单位，在 Allegro 中，精灵也是一个重要的概念。

显然，游戏中构成画面的动态元素很少是规则的矩形，而计算机中最容易描述图片的数据结构就是一个二维数组。所以，通常图形引擎会指定一种颜色作为透明色，在位图传输的时候忽略掉这种颜色，这样一张矩形图片就可以表现出各种不同的形状了（如图 4-4 所示）。

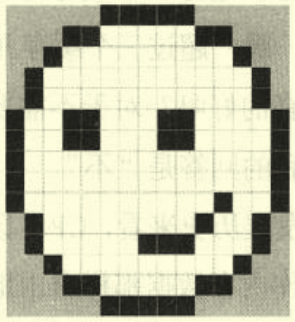


图 4-4　灰色为透明色

在 256 色模式下，通常会指定序号为 0 的点；在真彩或高彩模式下，则会指定粉红色，即红值蓝值最大绿值为零的颜色。Allegro 为精灵提供了非常丰富的操作函数，比如可以拉伸、旋转、镜像、透明以及各种光线变化。

当初我阅读了几乎全部的 Allegro 源代码，惊叹于其中各种优化方法，学到了很多东西。比如在做精灵的缩放时，Allegro 采用了一种自生成代码的做法，为调用者期望的缩放比生成一段可以工作的绘图代码出来，然后再调用这段代码进行图片缩放操作。生成代码可能需要一些时间，但是，一旦某个比例的代码被生成一次，就会被缓存下来，下次再期望使用这个缩放比的时候，就无须再次生成了。

除了这种采用透明色方式实现的精灵外，Allegro 提供了两种压缩格式存在于内存的精灵，分别是行压缩（RLE）精灵和编译（COMPILED）精灵。提到压缩的数据格式，很多人都会联想到 gif 文件、png 文件，等等。

但是，那些无损压缩格式对图形引擎的实现并无多大影响，因为它们仅仅是图像保存在外存中的形式。一旦图片被加载到内存供引擎调用，压缩的数据必须先被解码成平整的位图。而适合于引擎直接使用的压缩数据格式必须能够在绘图函数调用的时刻才实时地被解码，并且要适合

反观很多传统的图像缩放代码，或许在细节和算法上优化得很好，但是速度上却存在着某种极限，因为必须在代码中加入大量的判断、循环，还需要考虑目标位图的边界剪裁问题。一个代码自生成的方法却轻易突破了这种限制。

这里我特意提到这个，并不是特别推崇这种自生成代码的做法，而是提醒大家：很多时候，优化代码应该从一个相对高的层面看问题，执著于代码的细节反而会使得思想僵化，失去更上一层楼的机会。

做剪裁操作。

在内存中保存压缩的图形数据是一种典型的时间换空间的做法，但是，增加的计算操作并非一定会减慢程序的速度。这是因为，对于 CPU 来说，简单的运算并不比读写内存慢，甚至在很多情况下，访问内存才是速度的瓶颈。

数据被压缩后，整体的数据量变小了，在某种程度上提高了处理的速度（当然在这个级别上的优化，我们还依赖精致的汇编代码）。另外，条件判断和分支会干扰现代 CPU 的流水线工作，合理的压缩方法，可以不需要再像传统的以透明色的方式处理精灵，也就不需要每个像素去判断是否是透明色了。

如果以传统方式储存精灵，在编写精灵的绘制函数时，必不可少地需要逐点判断像素是否是透明色。这是非常影响 CPU 的流水线效率的。Intel 系列的 CPU 逐渐为此设计了很多新的指令加快这个流程：Pentium Pro 增加的 CMOV，MMX 指令集以及后来 XMM 增加的 MOVMSK 都可以改善这个操作。

另外，绝大多数显卡在处理显存上的数据时，也可以为检查透明色的图像传输做特别处理。可惜，显存非常有限，一般不可能一次把我们所有需要的图片资源全部放在里面，而显存的特性和内存又有很大区别（显存间的数据传输非常快，但是读显存却很慢，写显存也不如写内存快；而内存的读写性能却很稳定），在游戏图像引擎中考虑显存的管理是很麻烦的事情，一旦控制不好，就会产生效率颠簸，即速度忽快忽慢。

1. RLE 压缩

RLE 的全称是 Run Length Encoding，这是一种最简单的图像压缩算法，在 DOS 年代最为流行的 pcx 图像文件格式内部就是用 RLE 压缩图像的。

RLE 的压缩原理基于很多图像上有许多同色的色块（手绘的图片尤其是这样），这些重复信息可以被压缩起来。即使图像的颜色很丰富，如果图片是用来做精灵，那么至少会有大片的透明色。一般在设计图像引擎时，会把连续的透明色单独处理，这样的好处是，在做解码时，碰到

连续的透明色可以直接跳过，而不需要做更多的判断。

把图片基于水平扫描线来压缩编码，每条扫描线又分割成多段，一共有三种段：

（1）连续的透明色；

（2）连续的某种颜色的像素；

（3）不含透明色的，不同颜色的像素组。

每个段用一个字节做信息头，这个字节的高两位描述本组的类型（两个位可以表示4个状态，这里只用到3个，多余的一个可以留做扩展），低6位则描述这段包含几个像素。

如果是完全透明色的段，一个字节的信息头已经是整个段信息的全部；如果是连续相同颜色像素的段，在信息头后加上颜色信息就够了；而第三种情况，信息头后的段信息里就应该包括未编码的一组像素的颜色信息。

以256色，0为透明色的一个32字节宽的扫描线信息为例，如果位压缩的信息是这样的：

```
0 0 0 1 2 3 4 4 4 4 4 3 2 1 0 0 0 0 0 1 2 3 3 3 3 2 1 0 0 0 0
```

压缩后结果如下

```
3, (3*64+3) 1 2 3, (1*64+4) 4, (3*64+3) 3 2 1, 5, (3*64+2)
1 2, (1*64+4) 3, (3*64+2) 2 1, 5
```

这里，部分数据以（a*64+b）的形式给出，是为了方便大家分辨出字节的低6位与高2位。

RLE压缩格式的设计有很多方法，没有人规定必须怎样实现，但是，无论怎么实现，原则都是用一个表示某种像素被重复的次数的信息来表示重复的数值。这里这个方案，使原来的32字节被压缩为21字节，实际上在这里，压缩比是次要的，重要的是引擎的程序能够方便地实时解码，并且可以利用里面的信息快速地跳过不需要处理的像素，加快运行速度。

2. 编译精灵

关于如何避免在绘制精灵时过多地判断某个像素是不是透明色的问题上，Allegro还给出了一个新的解决方案，那就是编译精灵。编译精灵

直接用汇编指令保存图像数据，它使用了 mov 指令和即时数据变量来绘制图像。这样就可以让代码直接跳过一些不需要画的点。把运算的时间从运行时转嫁到制作时去做，是软件常用的优化策略，我第一次发现 Allegro 巧妙地实现了编译精灵时是很受启发的。

4.4.4　几何图形和 3D

作为一个完整的图像引擎，可以绘制一些几何图像是必不可少的，Allegro 提供了这些。画点，画线，多边形，等等。如果要制作一个平面游戏，我并不认为这些是必须的，因为，平面游戏中的一切，我们都可以用预先制作好的图片替代掉。

而 3D 的游戏，却是基于多边形的。即通过空间多边形的网格信息，把一些平面的图片，做一些变形，映射到多边形网格上。Allegro 提供了这些多边形的基本操作，当年我通过翻译 Allegro 的文档，了解了像透视映射这样的概念，也知道了矩阵这种在三维空间中坐标变换的工具。Allegro 自己带了一套矩阵运算的函数，实现很巧妙且高效。而且，还实现了一整套定点数的运算用来改善游戏中需要的数学运算速度。

在这里，我不想过多介绍 3D 有关的知识，那是一个非常大的话题，就算用一整本书来讲解都不够。只想告诉大家，Allegro 虽然在这方面已无太多的实用价值，但是却曾教给了我许多启蒙的知识，或许现在有人再读一下它这部分的文档和代码，会有许多收获也说不定。

例如，在 Allegro 的 3D 运算模块中，所有函数都提供了浮点数和定点数的版本。在当时，定点数运算的速度可以数倍于浮点运算。巧妙地使用定点数，还可以避免许多浮点误差问题，这些放在今天看都是很有价值的东西。

1．定点数

计算机中通常使用一种纯小数加指数的形式表示实数，通常被称为浮点数。浮点数的运算需要特定的硬件来完成，早期的 Intel 系列处理器用一块叫做 FPU 的芯片处理浮点数的运算。从 486 开始，FPU 又被集成到 CPU 中，但是，浮点运算和整数运算一直都是被两套相对独立的指令

集来处理。

从硬件设计的角度上讲，浮点运算比整数运算要复杂，所以速度也不如整数运算。即使速度相当，两套指令体系由于相互对立，而处理程序逻辑的指令又归属于整数运算指令，所以整数运算有天生的速度优势。为了执行效率，程序员们想到了一种利用整数运算模拟小数的方法，那就是以定点数来表示小数。

比如，我们让一个双字节的整数的高 16 位表示一个数字的整数位，低 16 字节表示小数部分，那么，这个整数实际就是我们想表示的小数的 65536 倍。小数点的位置是固定的，所以叫做定点数。

这样设定的定点数，数值范围大约在-32000 和+32000 之间。而数值精度则是大约$1/6\times10^4$，也就是说，我们至少有十进制的小数点后四位的精度，这对于游戏中大部分的应用都够了。经过简单的推导，我们可以知道，对于定点数的大小比较，以及加减运算，都可以当成整数来作，无需任何额外的操作。对于乘法或除法，仅仅需要多做一些简单的移位。

更有趣的是，对于原本比较复杂的三角函数运算，我们可以依照所需要的精度，预先计算好几张查询表即可完成。这里，使用查表的方法，以空间换取时间，是一种最常用的优化策略。

值得一提的是，最适合定点运算的角度表示方法，既不是 360° 等分圆的角度制，也不是 2π 为一个圆周的弧度制，而是更适合计算机来处理的，以 64 作为直角的表示方法。因为让 256 为一个圆周角的话，更容易化简角度运算。

即使在今天，使用定点数在很多地方也有价值。尤其在嵌入式平台上，比如手机游戏的开发，它们的处理器本身并不提供浮点运算，而是提供一个浮点运算库用软件去模拟，那样在需要大量浮点运算的场合，速度是不堪忍受的。

4.4.5　数据文件

怎样看待操作系统提供的文件系统和游戏开发者自己实现的文件系统？操作系统自带的文件系统固然经过大量优化，但它需要实现比专有系统复杂得多的功能，所以在性能上不敌游戏开发者自己的实现。

要说 Allegro 给我的启发，它提供的数据文件管理的模块必不可少。许多人写的程序都过分地依赖操作系统提供的文件系统。不光是许多业余游戏制作爱好者的作品，甚至我在许多商业渠道发行的游戏中，都曾经发现过数以千计的小文件。这真是一件糟糕的事情，文件过多使得访问效率降低，检索文件速度变慢，甚至浪费大量的磁盘空间，或许还会给玩家一种不专业的感觉。

Allegro 支持把一些小文件都打包成一个大的数据文件，然后以目录树的结构组织起来，附带的，还可以将文件以压缩形式储存。操作这些数据可以把数据文件当做一个目录，使用相应的 API 独立控制这些小文件。

其实，这个关于数据文件的模块并不难实现，但是却非常有用，使用一些简单的宏替换，就可以轻易地替换成操作系统下实际的目录和文件。这样可以在开发期，用操作系统来管理资源数据，而在游戏发行后，换成一个或几个数据包的形式。

给自己的游戏引擎做一套文件操作的界面是非常有意义的，我们的实现不仅仅限于从数据包中读取数据，或者直接从操作系统提供的文件系统上读取，甚至可以扩展到从网络或者别的地方（比如内存映射），有时候，数据还需要压缩、加密等，都可以在文件控制这个层面完成，另外，这对跨平台的开发也非常有好处。

另一方面，采用打包的数据文件还可以提高文件访问的速度，原因有两点：

（1）小文件被集中，系统文件 IO 的缓冲区利用率被提高，物理上的读盘操作减少了。

（2）打包后的文件可以不用字符串的文件名检索而使用一个唯一 ID，如果我们再对文件索引 ID 排序，在数据包文件中检索逻辑上的文件，检索速度将大大高于操作系统提供的文件系统，这在资源文件数以千计时尤为明显。

为了简化设计，事实上，我们只需要做文件读操作的接口，因为游

戏软件中需要写文件的操作实在很少，至多只有写 log 和保存存盘记录。对于写盘操作，操作系统提供的函数已经足够对付。

4.4.6 声音

一个游戏给人的第一印象的好坏，通常取决于它给人的感官上的刺激。计算机最容易模拟出来的除了图像就是声音了。

Allegro 为声卡提供了一套统一的编程接口，从硬件角度上说，主要是 MIDI、录音和采样声音的播放。前两者在现在的游戏中很少用到（由于现在内存已经越来越廉价，PC 的游戏已经直接把乐曲用数字采样的方式保存起来，很少再使用 MIDI 的形式了。倒是现在游戏机平台上出现了大量麦克风控制的游戏，有可能这个势头会延及 PC 领域）。

对于声音的处理，不像图像需要做运行时加工，声音可以看做是一种一维线形的数据，只需要分配出缓冲区填入数据，让声卡播放就行了。Windows 在这部分就提供了基本的 API，即 DirectSound。这里，Allegro 为开发者考虑，又细分出静态的声音播放和音频流的播放。

所谓音频流的播放，实现是先在声音缓存中填充一小段时间的声音数据让声卡播放，然后等部分的缓存的数据播放完后，以新的数据替换掉已经不再需要的数据，循环利用缓存，让音乐持续播放下去。这点不属于硬件的底层控制，但是它又具有通用性，所以被 Allegro 独立出来实现。

声音相关的控制在游戏引擎设计中并不比图像控制来得简单，图像是在空间上二维的数据，在时间上是一瞬间的使用；而声音在空间上是一维数据，但在时间上却需要连续性的控制。这导致了声音数据的管理对于图像在引擎一级的实现稍显复杂。

而一小段声音需要重复播放的话，可以把全部数据放在缓存中，大多数情况下我们只需要调用一下 API 播放即可。实现上，我们还可以细致地去控制左右声道的音量，模拟一些 3D 的声音效果。至于动态地修改声音缓存中的数据，造成诸如回声、颤音、震音效果，理论上是可行的，Allegro 里也提供了类似的接口，但没有去实现。实际上，我很少见到有游戏通过编程计算来控制音效的。毕竟，声音是比图像更抽象的东西，很难用程序直接产生出来。

至于这几年很流行的 3D 音效的概念，在 Allegro 风行的年代，还没有声卡在硬件上支持它，所以 Allegro 并没有后来 DirectX 中有关 3D 音

效的部分。实际上，如果没有专门处理声音的硬件的协助，模拟 3D 环境中的立体声，计算量可能比 3D 的图像还要大。

4.4.7　其他的部分

Allegro 是一个完整的游戏开发用的程序库，所以它还提供了对鼠标、键盘、时钟，甚至游戏杆的控制界面。在今天，我们都已经使用 Windows 作为主流的平台，操作系统为这些设备给出了更方便的编程接口，虽然有所差别，但区别不会太大。如果读者有机会接触更多的平台，特别是一些游戏机的平台，也不会发现太大的差异（总的来说，由于需要支持多任务，Windows 还是相对复杂的）。

这些只是对硬件设备控制的抽象，从编程角度上说没有参考价值，毕竟我们已经不必再钻研各式硬件如何编程。不过从引擎设计的角度上，还是有一些参考价值的。如果有时间和精力，可以读一下 Allegro 的文档和源码。

值得一提的是，Allegro 包含了一套 GUI 系统，不要小看界面程序的设计，一套好的图形界面模块很难编写，尤其是有鼠标控制的界面。如果你正在制作一个游戏，这里有两个选择。

一是极力简化掉界面，只有一些简单的对话框和按钮，并且硬写出这些代码凑合着用用，这样只需要大约一千行代码就可以搞定。对付一些不需要太多人机交互的游戏似乎可以应付得来。

二就是耐心做一套设计合理的 GUI 吧。以我的经验，至少需要一万行代码才能弄出个像模像样的东西来，最好能内嵌一个脚本来描述界面及其逻辑，需要有良好的界面对象的管理以及消息传递机制。这些都需要一定的编程设计功力，面向对象的思维方式和设计模式都是必修课。Allegro 虽然只是初步实现，不过在你自己做类似设计时如果遇到困难，不妨参考一下它。

4.4.8 小结

终于在有限的篇幅中，尽我的能力把 Allegro 中的精华展现给大家了。当年在花去很多精力翻译完数十万字的 Allegro 全部文档后（从我开始翻译的那天起，Allegro 也一直在发展，跟踪文档的变化和保持原文的格式，也是颇费心思的一件事，据我所知，在 Allegro 文档当时的诸多语言版本中，只有我翻译的这份中文译本完全保持了和英文原本相同的组织结构）。

我一度希望写一本关于使用 Allegro 开发游戏的书，一直未能如愿。随着 PC 的发展，Allegro 的实用价值越来越小，但是其公开源码带来的学习价值却保留了下来。能在我的这本书中花去一章的篇幅引导大家从 Allegro 中发掘出宝藏，也算是了却了一桩心愿，这里借机感谢 Allegro 的作者，我的良师益友，Shawn Hargreaves，感谢他对游戏制作爱好者们的无私奉献。

4.5 cfido 中国惠多网

时值 2004 年，我早已不再眷念曾经有过的 Apple II 时代、DOS 时代。虽然曾经有那么一段时间非常孩子气地排斥一日千里的计算机技术发展，认为程序员活在 PC 时代比 Apple II 时代悲哀，Windows 没有 DOS 更适合游戏开发，但现在已经把自己放在了时刻接受学习新技术的状态。

而在跟随计算机网络发展的道路上，1996 年初接触的 cfido（中国惠多网），却如初恋情人般迟迟不能忘怀。1997 年底开始，我就混迹于 Internet 的各种论坛、新闻组、邮件列表，但无论多么热衷于这些互联网上的文字交流，都无法像当年于 cfido 的那般投入。每每对互联网上这些交流环境有所抱怨时，心里总要去和当年的 cfido 做一番对比，然后唏嘘不已。网络技术越是发展，人们越是能更方便地接入网络，我越是感到当年的 cfido 精神在消逝，这是技术之外的事情。

1995 年的时候，我在《电脑报》上看到了国内业余 BBS 的介绍，

而在此之前就听说了美国所谓的信息高速公路的计划。虽然我当时还没有 Internet 的概念，但一直以来期盼以某种途径和外界自由交流技术，所以对 BBS 这种陌生的东西大感兴趣。直觉告诉我，这将是最廉价的交流信息的方式。由于那个时候连电话都是奢侈品，而买个 MODEM 在经济并不宽裕的情况下更显得浪费。直到 1996 年初，我才从好友那借到他换下来的一块速度仅 2400bps 的 MODEM 卡，开始我的 cfido 生活。

cfido 虽然在鼎盛时期覆盖了半个中国，拥有 200 多个站点，至少拥有几万的活跃用户。许多 id 的所有者（如 vicent qiu 求伯君，pony ma 马化腾，lei jun 雷军）现在都是 IT 圈中的名人。作为一个业余组织，比同期的 Internet 还要出名，但到新世纪已经销声匿迹，现在更多的人或许已不知道它曾经的存在。这里，做个简单的介绍。

fidonet（惠多网）是世界上最大的由业余计算机爱好者组成的拨号 BBS 网络。它依靠电话网络连接起来，用户拨号接入当地的站台，可以收发信件，参与各种话题的讨论，而各个站台之间以一定的频率靠长途电话互相转发信件。

虽然 fidonet 的中国部分属于整个 fidonet 的一部分，但是并没有参与国际间的转信，它自成一个系统，我们称之为 cfido。各地的站台通常都是由站长处于爱好和热情私人搭建，站点间的信件互转也是这些爱好者们的自发有组织的行为。在 Internet 并未普及的年代，这是让全国各地的计算机爱好者们互相通信交流最廉价和便捷的工具。

我的第一个 MODEM 只有 2400bps 的速度，也就是说每秒传输不到 300 字节。对于宽带逐渐普及的今天似乎有些不可思议。但是对于 cfido 那样的拨号 BBS 的形式，这个速度已经勉强够了。构建一个 BBS 站只需要一台可以拨号的 PC、一条电话线。然而，一个站台少则有几十用户，多则上千，这一条电话线在同一时间只能供一个用户使用，这就决定了浏览收发信息的方式和现在 Internet 上的论坛不同。

每个站点把不同的讨论话题按信区分开，用户可以订阅他关心的信区，在每次连入站台后，由站台将他订阅的信件打包压缩传输到本地，断线后再观看回复，所有回复的信件在下次连接时再打包上传。原本传输的就都只是些纯文本信息，再打包压缩后，每天新到的信息总量就不

太多了，不需要太大的通讯速率。

在1996年的那个暑假里，我近乎疯狂地在cfido上飚信（这种行为后来在Internet的论坛上被称为灌水），活跃在武汉那个最大的站台——劲捷BBS上。这个站最辉煌的时候有3根电话接入线，每天24小时上千人次接入，即使在半夜都需要3线自动轮拨近百次才得以登陆。

在cfido逐渐没落后，它还作为华中地区的一个中心，全国转信之外，和周边很多站点互转信件。在上面时常还可以看见一些远在甘肃、内蒙、江西等地的朋友慕名用长途直拨过来和武汉本地的站友交流。

起初，我主要活跃在武汉本地的信区里，聊天和讨论计算机技术，那个时候参与到BBS里交流的朋友大多是计算机迷，程序员的比例也很大。通过这个，我发布了好多个自己写的小软件，其中最有影响力的是一个电子书阅读软件，靠此交到了不少朋友，其中一个日后还成了从事游戏开发的同事。

在这里我第一次知道了Allegro，因为发现了许多志同道合的想做游戏的人的存在，所以促使我开始翻译Allegro的文档，这几乎是我正式迈向职业游戏开发的第一步。后来，有了向全国转信区的写权利后，又转而和各地的计算机爱好者交流经验。

他们的ID现在已不大记得了，只是在后来泡Internet的岁月中，时常收到陌生的E-mail询问："你是否是当年cfido上的CW"，倍感亲切。甚至，我现在干了近4年的这份工作，引我进入如今这家公司的人，也是cfido上认识的一个朋友，只要是和cfido扯上关系的人，都能给人一种亲切感。

到1997年底的时候，我就读的大学里已经可以在网络中心自由上Internet了，但是对cfido的眷恋是无法消逝的。那个时候，我托留在武汉的同学帮我登录BBS，收取信件后，把信包下载到软盘中，然后用平信邮寄到学校，然后等我回复完后再邮寄回去，每次一个来回都需要2周的时间，就这样坚持了一年。

多年以后，回想当年的情景，到底是什么让cfido有如此的魅力？我想答案就是cfido反映出来的某种精神。

首先是真实。虽然由于软件的限制，大家交流时都用的是2到3个

单词的英文 ID（cfido 战友一般用英文 ID 每个单词的首字母的连接缩写互相称呼，我的 icq 至今都是沿用当年大家对我的昵称 cw 做名字）。但是，在自我介绍时都填写了真实的名字、家庭地址、职业、性别、年龄和其他个人情况，几乎没有人去刻意作假，时常还有人组织聚会做面对面的交流。大家在这种亲切的环境里几乎无话不谈，无聊的漫骂我几乎没有见过，更多的是一些理性的争论。

其次是专业。专业并不意味着大家讨论的都是些毫无人情味的技术话题，而是说，大家都知道该怎样在合适的场合发表合适的话题，聊天的信区大家一起聊天，分门别类的技术专题信区，有感兴趣的人参与，并且可以讨论得很深入。不同于现在在线的论坛，想到了什么就写什么。

那个时候，大家全都是离线阅读和回复，写好了回复，也并不是立刻就登录发出。即使是本地不参与转信的信区，一封信从写出到被很多人看到也是有一定的周转时间的；如果是全国转信的话，传播到各地得到反馈甚至需要好几天时间。这样，大家在表达自己观点的时候都会相对谨慎，至少是经过仔细思考的了。

那种 BBS 的讨论形式和现在的论坛组织信息的方式也大不一样，并非按某个主标题为线索，而全部是一个人写给另一个人的公开信形式。大家可以饶有兴趣地观看某两人之间的对话，也可以很有礼貌地参与进去，变成一组人之间，两两地论战。那个时候，最能满足虚荣心的莫过于连接上站台后，被通知这次收到发给自己的信件的数量了。

然后是管理。由于每个站点都是由站长无偿为大家提供服务，没有任何商业性质，所以站长就有了绝对的权威。大家都认同站长的管理，而站长也希望更多人的参与，所以也不会无端赶走用户。接受站长任命的各个信区管理员也就同样被大家接受及认真为大家服务了。大到全国转信的协调上，不得不惊诧一个有上百个节点松散的民间业余组织，大家都是平等的地位，在那几年居然维持了一个良好的秩序。

虽然后来听说在许多问题上，大家也有争吵，但是至少作为我这样的一个普通用户，丝毫没有干扰过日常的使用，没有人把争吵引入到不相干的讨论中去，也很少在全国转信区看到让人不快的文字，那应该是各地的站台在本地都被人工过滤掉了。

如果说以上几点在现在某些 Internet 的论坛中还可以存在，那么剩下的最后一点，对于我们就只能是美好的记忆了，**我指的是自律**。cfido 从早期一直到鼎盛期，这都是一个显著的特点，后期因为被 Internet 那过分的自由所感染才有所弱化。首先是有些规定，其中一些以现在的眼光看过于严格，但几乎每个人都能自觉遵守。比如在专业信区不得讨论无关的话题，如果由于某个相关讨论发展到内容越界，讨论人就会写上一句，我们转移到某个另外的信区继续。关注这个话题的人可以根据这个查看相关的信件，而当事人由于信件组织结构的原因自动被转移了（前面提到过，cfido 的相关软件是以收发件人为组织信息的线索的）。

无意义的信很少出现，比如只是写上“赞同”、“反对”字样的，有时候甚至规定了回复应有的字数底线。在这点上还有“cfido 上不言谢”这个条款，就很能突出其风格，即不要仅仅因为想感谢对方的答复而发一封只含谢谢的信，这种行为同和只有“赞同”字样的信一样被认为是一种浪费资源和影响他人阅读的行为。

同样，为了节约资源，减少大家上传下载以及转信的数据量，引信也被严格规定到一定的比例，大家必须合理地删除原作者的信件内容，只保留下相关需要讨论的部分。信件的标题也被规定必须概括表达其内容，而且过长的讨论一旦主题偏离，也要求相应的改动主题。

除此之外，还有很多不成文的行为规范，比如不越权帮助管理员指出其他用户的错误和争论管理员的做法。在现在的诸多论坛中我们可以看到，这些往往成为脱离主题的帖子的导火索，版务的问题交给管理员用私下的渠道解决不会打扰到其他人通常成为最好的解决方案。避免大家起不必要的争端的另一样法宝是，如果在文字中开对方的玩笑，就在句子后面写上一个<JOKE>明确表明自己是玩笑之言不必在意，这样，绝大多数因为语言理解上的偏差造成的不快就被避免了。

另外，当时的 BBS 里书写的严谨也是有目共睹的，除了少数几个被大家公认的“通假字”（把“很”写成“狠”），很少出现现在网络上泛滥的错别字，用英文字母表达的几个简写单词，也只有被大家公认了的诸如 mm,gf 这样不多的几个。而今天，对于充斥于网络上千奇百怪的所谓网络语言，已经到了许多人看不明白的地步，除了乱造缩写，好端端的

汉语词汇都已经不仅仅局限于用同音字代替，甚至好多人以写错别字为荣，对于此，我倍感深恶痛绝。

至今，我在Internet上的行为还深受cfido的影响。在各种论坛发表言论或者回复别人的帖子时，首先想到的就是怎么给出一个合适的标题，并在正文中重述标题。写完之后，阅读一遍改正打错的字。对于回复别人的文字，斟酌哪些文字需要引用保留，哪些应该删除，并在对应的地方写下自己的观点。书写文本时，使用回车手动分行，隔行分段，以及中英文混写时空格的使用以及半角标点的使用都打下当年的烙印。对于网络上的争论尽量保持平和的心态，同时尊重管理员的判断和管理。

虽然cfido不会再有，但是我们依然可以从当年cfido精神，cfido文化中吸取点什么。如今Internet逐步成为人们生活中不可分割的一部分，成为信息来源和知识交流的重要平台，我们多一些文字书写上的自律，可以让网络多一些条理、多一份整洁。

另外，我是偏爱E-mail的，对于这个日常工作和技术交流的主要工具，还想多说点什么，一些良好的习惯会让你在同事和网友的心目中加分。

- 回复收到的每一封E-mail，即使没有时间写复言，也应该让对方知道你已经阅读了他的信件。
- 使用纯文本，发送过大的附件前最好先征得对方的同意。
- 给E-mail起一个合适的标题，能够概括内容的标题，对于回复，在讨论内容偏题时修改题目。
- 删除尽可能多的引信，尤其是对方的签名这样对主题表达无意义的字节。
- 合理地分段，多表达一些自己的意思。比“赞同”两字多写一点。
- 发出之前，读一遍，尽可能地修改掉错别字和语法错误，核对一次收信人是否正确。
- 不要随意公开转载私人交流的信件，即使转载也注明出处。

第5章

Windows 编程

如果我们在 Windows 下开发游戏，那么理解 Windows 系统就是尤为重要的事情。Windows 只是一个操作系统的特例，不属于计算机理论知识的范畴。所以，学校里的课程通常不会安排也没有必要安排，这方面的教学（似乎现在许多高校为了让学生学到一些实用的知识，已经开始有这方面的课程）。这导致了许多已经掌握了编程知识，理论功底也很扎实的程序员，面对 Windows 下的开发无可适从。

RAD 的开发工具固然简单，但跟课本上学到的编程知识相比，似乎中间掉了档。这一章就是想补上一些环节。至于诸如 VB，Delphi 这些 RAD 开发工具本身没有在文中提到，因为我认为，对软件设计的更多知识，像设计模式等，了解到一定程度后，每个人都能理解 RAD 的原理。本章的末尾倒是浅谈了一下 MFC，因为我非常想让对 MFC 有偏见的人明白，MFC 既不是天使，也不是恶魔。

对于 Windows 编程的阐述，还是从基础讲起。虽然学校的计算机相关专业都会教授操作系统课程，课本上也有颇为详尽的概念阐述，但这里，我还是花去一些篇章，结合 Windows 解释了进程、线程、虚拟内存这些名词。因为就我看到的许多人，即使是科班出身，对这些概念依然认识模糊。

关于 Windows 底层，讨论了 DLL 和 COM 的概念，这些是 Windows 构架基础之基础。而对于更高一点的层次，Windows 的窗口和消息传递，则将利用一个仅仅十数行的程序对照讲解。相信这更能让不理解 Windows 编程的人来接受。

最后，跟游戏相关的部分，并没有去讨论 DirectX，而选择了 GDI。回避 DirectX 是我刻意为之。这并非我更主张使用 GDI 而非 DirectDraw，只是因为在我看来，DirectX 编程并非游戏编程的重点，希望读者能理解我的这层用意。

大二那年，我在寝室里拼凑了一台486练习编程和制作网页用。虽然Windows 95已经发布，但似乎我周围的人对此接受度不大，即使上网也是用Windows 3.1+Netscape。为了省钱，我没有给机器配硬盘，只是为机器装了8M内存，这样平时用内存模拟个虚拟盘，就可以顺利地写程序了，甚至每次用软盘启动后自动在虚拟盘上安装上Windows。这种环境持续了一年，我主要也就是写一些练习用的小程序玩。直到1998年初，把机器升级到5x86，并配备上一个旧的210M硬盘，条件才得以改善，至少可以用Windows 95的系统了。

这期间我一直都在DOS下做开发，工具是DJGPP+ Allegro。我从来不认为操作系统是一个很大的开发障碍。由于经济上不允许配备足以运行Windows 95的硬件，那么玩玩DOS也无妨，为此，曾经还在自己的个人主页的留言本上跟未曾谋面的网友大吵了一架。那个旅居海外的朋友告诫我：Windows的发展是一种不可逆转的潮流，作为当时小有名气的中文游戏开发网站却整天介绍DOS下的开发知识，误导了大批想做游戏的有志青年。当时年轻气盛的我，对这顶强加的帽子颇为愤愤不平。

如果有读者想怀怀旧，看看20世纪90年代末中国一些业余的游戏开发者关心着什么，那么可以去我的主页翻翻那些年的留言记录，现在读起来真是别有一番趣味。

或许是有那么一点酸葡萄心理，我对自己用不上的Windows 95还真有一点排斥心理，在很长一段时间都没安排它的学习计划。直到有一天，我终于重新理解了，对于编程来说，知识永远是共通的，Windows编程也无太大的不同。这个时候，我才开始放弃了在DOS下做开发，全面转向Windows。

说起来，这个转折点依旧来自Allegro。Allegro从某个版本起开始声称要支持Windows，这在我以前看来是不可能的。因为Allegro在原始设计上并不适合Windows，它没有安排窗口消息处理的机制，而Allegro的期望却是可以不修改以前任何的代码，让以前的旧代码只需要重新编译就可以运行于Windows。我考察了还在内部测试中的Allegro新版本，发现它是用多线程实现的，用额外的线程来处理窗口消息，让主线程跑程序逻辑，这样就可以不修改老代码的逻辑而正常运行在Windows下了。

在Allegro制作Windows版本的期间，我决定不再使用Allegro，而自己开发一套更适合Windows的游戏程序库。导致这个想法并使我付诸

行动的原因有三：

第一，Allegro 的 Windows 版本一直都不太稳定，很长时间都没有推出正式的版本。

第二，系统地阅读 Allegro 的代码后让我发现，Allegro 为了更加通用，兼容很多的色彩模式和显示模式，很多地方牺牲了我很看中的效率。比如，RLE 精灵为了支持各种颜色模式，还需要以透明等不同的方式绘制，采用了把绘点的部分抽象出来作为可以替换的函数的形式。而我认为，如果考虑效率，就应该为特定的模式做特定的汇编优化。通过 E-mail 的交流，Allegro 的作者同意我的看法，但是又认为 Allegro 已经是一个成熟的产品，因为要照顾到已有的代码，所以不能做结构上的改动了，建议我自己实现自己的想法。

第三，我在一个 maillist 里发现了一个叫做 RedX 的游戏库的发布，它对 DirectX 和 Windows 窗口操作做了最基本的封装，这让我发现，其实，Windows 编程并不复杂，DirectX 也使用简单，这样一下子就扫平了我对 Windows 的恐惧。

然后，就有了“风魂”。“风魂”这个名字是我当年为我们一个业余游戏制作小组起的，后来这个小组的人各奔东西。我保留了这个名字作为自己开发的游戏引擎的名字，虽然代码几经重写，名字却一直留了下来。它是我尝试学习 Windows 编程的第一个实战程序，也见证了我学习 C++和对 C++逐步理解的过程，并且包含有不少当年挑战汇编优化极限的函数。

在我的学生时代，它是最大的一个工程，正如大多数爱好编程的学生一样，我也把自己的作品免费发布。在我毕业那年的最后一个版本，现在还可以在我的个人主页上下载到。有兴趣的爱好者可以下载一个把玩。公开的那个“风魂”老旧的版本各个方面都不那么好。我接受行家的批评，但也请原谅当年的我，作为一个初窥编程领域的年轻小子，曾犯下许多的错误。

RedX 是一个用 C 完成的 DirectX 封装库，它可以用一个 Windows 下免费的 C 编译器编译（Lcc-win32）。其实，RedX 做的工作实在是非常非常简单，仅仅只是包装了一下 DirectDraw，DirectInput，DirectSound，

让完全不懂 COM 的人可以以简单的 C 接口调用而已。整个的代码根本没几行，封装的水平也谈不上有什么设计，甚至可以说比较糟糕，初通编程的人都能够在 1～2 天内写出来。事实上，这套库在其作者那个不算起眼的 maillist 发布，并被我无意间发现后，也再没有新版本更新。

仅仅只是由于我当时没有接触过 Windows 编程，觉得 DirectX 似乎是个挺神秘的事情，对于游戏底层的代码依赖了 Allegro 很长时间，习惯性地想在 Windows 上找个替代品的缘故，便如获至宝，马上在个人主页上撰文大肆宣传。

关于 RedX

不久我居然在《电脑报》上也看到了 RedX 的介绍，由于我发现 RedX 比较偶然，它也并非一个很正式的开源库，不由不让人联想，是否当时我的个人主页还真的有那么点名气，让报社的编辑都关注着转载我的发现☺

虽然我自己很快就发现，RedX 只是一个连优秀都谈不上的玩具，可惜很多人却并不会意识到。直到过去这么多年，最近我还收到一封 E-mail 向我咨询有关 RedX 的问题，让人苦笑不已。

5.1 Windows 编程入门

本章从这里将带领大家浏览 Windows 编程的诸多方面，如果你对 Windows 编程毫无了解，期望通过阅读这短短的篇章后就可以在 Windows 下游刃有余地编码，做出巧夺天工的 Windows 应用软件或者游戏，那可能会让你失望。去看看书店里那一排排的关于 Windows 编程的大部头吧，知识不可能被如此得压缩。

但是，也不要放弃阅读，即使你对 Windows 编程已经有一定的了解，你马上就可以看见，本章和大多数 Windows 编程教材不同，并没有在一开始就列出一个简单的 Windows 程序，让读者有一个感性的认识，或者用图示描述怎样使用某种开发工具（比如 Visual C++）一步步创建出一个能运行的 Windows 小程序让初学者获得那么一点满足感。

知其然而不知其所以然是无法深入下去的，容易过早地遇到学习瓶颈。对于游戏程序员，需要程序最大地发挥硬件的潜力，对掌管和控制硬件的操作系统的理解非常必要。所以，这里在最前面先讲解 Windows 的一些概念，而非列出一个范例程序，让读者囫囵吞枣地接受。如果通过本章的阅读，读者能够理清一些以前模糊的概念，我就感到非常欣慰了。许多看似和实际编码距离很远的东西，不需要多久，在 Windows 下编程实战中就能从中获益。

5.1.1　Windows 版本综述

如果要回顾 Windows 家族的历史，我们把 Windows 3.x 归结为其始祖（在之前的版本并没广泛地为世人所知），Windows 为我们带来的最重要的概念是：统一的用户界面和设备驱动程序模型。

正如我们在前面的章节看到，即使操作显卡这个工作，不同的显卡也可能不一样，作为和苹果机不同的 PC，还有大量第三方开发的设备，所以，提供一个通用的设备驱动接口是很有意义的一件事情，这样，应用程序甚至可以在新硬件被开发之前就投入开发，硬件厂商也不必担心没有软件制造商为他们的新产品制作软件。

另外，如果让各种应用程序协同起来工作，一个统一的界面接口也是很重要的，它不仅可以让用户更容易上手学习一个软件，还可以让软件之间更容易地协同工作，不需要每个软件都把功能做得大而全。作为一个多任务操作系统，资源共享是很重要的概念。

Windows 9x 是 Windows 3.x 的进化版。到 Windows 9x 后，Windows 家族才算一只脚迈入了 32 位的大门。系统中的各个进程、线程，才真正地不用用户程序自己调用特定 API 达到协作而在逻辑上并行起来，进程间也勉强做到了独立。在这个版本中，操作系统也开始企图为用户自动地完成配置的工作。

但因为为了向历史兼容，以及 Windows 9x 本身的基础并不牢固，故这是一个被开发人员非常唾弃的一个操作系统版本。不仅在其上编写软件很容易犯错误，操作系统本身也有诸多的 Bug，使得应用软件编写者需要用一些并不优美的手法绕过这些问题。而且由于设计初期，Windows 9x 期望运行在很低配置的机器上，兼容古老的设备，使得其系统构架在现在的硬件条件下反而没办法发挥出硬件的性能。

对于普通用户，Windows 9x 的诸多安全漏洞已经不是为操作系统打补丁能够解决的（系统核心决定了这一点），让互联网无处不在的今天，许多还在运行 Windows 9x 的用户被暴露在危机四伏的世界之中。

稍微晚一些推出的 Windows NT 和 Windows 3.x/9x 使用了完全不同的内核，是一个完全的 32 位操作系统。它是微软意识到老的 Windows

版本缺乏稳定性，为企业使用而开发的。虽然 NT 发展的初期还不足以适应网络运用，但它的核心完成得很好，为日后的发展打下了坚实的基础。

Windows 9x 的内存管理、资源管理方面远不如采用 NT 核心的 Windows 系列。在发挥机器性能方面，Windows 9x 唯一的优势就是可以在极小内存环境下运行（所谓的可以运行，往往只是指能够在操作系统上运行单个极其简单的应用程序），一旦硬件配备超过 64M 内存后，多任务环境下，内存却很难再得到有效的利用，被操作系统轻易地挥霍掉。

有一段时间，我无法理解为什么在中国（包括海外的一些华人社区），依然有那么多忠诚的 Windows 9x 的用户。我们经常接到投诉说我们的游戏不能运行于他们的操作系统上。后来我们在开发中增加了一台专门的 Windows 98 测试机，不断地调试排错才能够解决操作系统兼容性问题。发现的90%的问题是因为Windows 98操作系统的bug由于不应该有的限制造成的，当然也有 10% 让我们自己很惭愧，是自己程序的问题，但由于 Windows 2000 的牢固性，把这些问题化解掉了。

后来我明白了问题的所在，其一是一度存在许多软件，它们利用操作系统的漏洞做了一些特别的功能，或是做一些 Windows 9x 监督不到的非法操作。这些东西换到 Windows 2000 或其他 Windows 系统上则不能正确运行。这种软件以一些游戏修改工具、网吧管理软件为最，大多是非专业的开发人员所为。它们大多不太积极更新，使特定的用户群限在特定的操作平台下。

其二就是 Windows NT/2000 等提供了更完善的安全管理机制，因为许多用户不愿意花时间学习，那么一点酸葡萄心理使得很多人臆想 Windows 98 能比 Windows 2000 更高地发挥计算机效率，所谓的“老鸟”把这种观念传授给刚接触计算机的“新人”，“新人”又成长为“老鸟”……这个人群像滚雪球一般增大，在没有太多软件版权观念的中国，这种情况很容易被发展扩大（作为操作系统提供商，微软早已停止 Windows 95/98 的销售，并逐步停止对其的技术支持）。

Windows 2000 是 Windows NT 的进化，如果在 Windows 下用控制台查阅版本号，可以得知它是 Windows 的第 5 版。从技术层面上说，

Windows 2000 带来的革新是 COM+以及 IIS 的内置这些为网络运用的服务。对于游戏程序员而言，更重要的是它增强了 NT 系列对多媒体的支持——集成了 DirectX 7.0。这使得大量的 PC 游戏开发转向上面（之前，NT 对这些支持得并不好）。从易用性上讲，2000 比早期版的 NT 更加贴近个人用户，Windows 家族的这次革新是后来 XP 都无可比拟的。我们下面所提到的 Windows 编程技术都将假设用户的平台是 Windows 2000，当然，几乎所有的这些基础知识同样可以运用于 Windows 98 或者 Windows XP 等。

Windows Me 是对 Windows 98 的改良，可以说是微软看到了 9x 系列的种种问题，企图再次移植 NT 上的代码去补 9x 系列的漏洞，事实证明，这同样是一个失败的作品。从后来 Windows XP 的推出就可以看出，微软已经彻底放弃了 Windows 家族中这一糟糕的分支。对于普通用户来说，或许并没意识到什么不同，但于开发者，终于可以长出一口气，在可见未来的某一天，不用担心自己的程序由于不能在漏洞百出的 Windows 9x 上运行而接到用户的投诉了，更多更健全的操作系统机制可以被灵活运用而不必照顾落后的操作系统。

5.1.2　操作系统的核心

Windows 作为一个成熟的操作系统已经是如此复杂，其结构已不再是以提供一组 Windows API 为主；从开发人员来讲，也不再把程序界面可以做得多酷作为关心的首要问题。现今的操作系统已经把网络变成其结构的一部分，基于 EXE 来实现的不太依赖操作系统提供的诸多服务的游戏软件可以说是一种简单运用（简单是针对操作系统的架构而言）。

但在本书的其他部分，不会再提到跟网络有关的任何文字了。因为这并非一本讲解 Windows 编程的书，我们的目光还是被时刻带回游戏这个话题，至于如今热门的网络游戏的程序制作，如果你想尝试，那么还需要去买一本讲解网络编程的书。和 3D 技术一样，网络技术也是需要专门著作来阐述的（简单调用一下 socket APPI 倒不是一件难事，从网络上找到几个例子程序，加上零星资料就可以办到，但写好网络相关的程

序却不是件容易的事)。下面我们一起温习一下基础知识。

1. 进程、线程和虚拟内存

当一个应用程序被启动时，Windows 将创建一个进程，这个进程所拥有应用程序的一个实例和私有的内存、资源这些静态的东西。同时，在创建这个进程时，操作系统还会创建出一个线程与它相关联。线程是 Windows 中最小的执行单位，Windows 同时拥有着很多线程，每个线程都必须关联在某个特定的进程上。

自动被创建出来的线程称做主线程。许多情况下，进程和线程就这样一一对应着，但是，随着这个线程的运行，还可能有更多的线程被创建出来，一个进程拥有多个线程的情况也并不少见，只要有一个线程和进程关联着，该进程就将继续运行下去。

Windows 是一个多任务的操作系统，它让诸多线程同时运行（至少在用户的角度看，逻辑上是并行的)。但是在任意时刻，每个 CPU 只让一个线程处于运行状态（现在新的 CPU 支持超线程技术，允许同一个 CPU 并行运行多个代码序列，好像有多个 CPU 存在一样，这样，一个物理上的 CPU 就可以让多个线程处于运行状态了，我们可以把这种 CPU 看成逻辑上的多个 CPU)。操作系统为每个 CPU 维护一个线程的队列，当一个线程处于运行状态时，队列中别的线程就处于等待或就绪状态。

运行中的线程将不停地利用 CPU 执行它的代码，但不是无限期的。Windows 利用时钟定时地产生一些中断，来打断正在运行中的线程，在中断的间隙检查当前运行线程是否超过了分配给它的时间片，如果超过，运行将被强制中断。

当然，如果线程自己决定等待某个事件或者对象时，或者更高优先级却处于等待状态的线程的条件已经满足需要运行时，中断也会发生。这个时候，Windows 把该线程设置为就绪状态，系统会根据就绪线程队列中的优先级别的次序，把最高优先级的线程设置成运行状态，并切换上去运行。如果没有更高优先级的线程，就会切换到同等优先级的下一个线程。

这里反复提到了线程的优先级问题，Windows 为线程提供了 31 个不同的优先级，按优先级别决定下一次应该切换到哪个线程运行。如果

系统老在最高的优先级的几个线程上循环切换，那么操作权就永远轮不到低优先级的线程上了。为了解决这个问题，优先级低的程序会被自动临时提高优先级别，这样使得它们至少能被调度运行一个执行周期，用这种方法，大多数情况下能够保证所有的线程至少有机会取得 CPU 的使用权。

另外，为了使 Windows 的应用程序对用户的响应更加灵敏，拥有前台窗口的线程一般会被提高两个优先级别。当然这只是一般情况，如果线程本身就被设置成高优先级了，就没有了这种优待。

Windows 借助线程的概念封装了代码的运行过程，允许多段代码的运行流程在逻辑上独立且并行运行；而让各个进程间的数据可以互不影响就需要借助虚拟内存这个概念。在这方面，内存管理器是 Windows 提供的核心重要模块。

在前面的章节，我们了解到，在 32 位保护模式下，代码和数据都是存在于一段长达 4G 的地址空间里的，Windows 为每个进程都提供了这样的空间。每个进程能控制的都是自己的私有空间。而 Windows 的内存管理器，负责把物理内存映射到这 4G 空间的一部分上。

应用程序需要比物理内存更大的数据空间是很容易被满足的，这是由于正在运行的线程在任意时刻其实只会用到很小的一块相关内存，Windows 可以把暂时不用的内存数据暂时放到硬盘上，等需要使用的时候再读取回来，这是一种时间换空间的做法。不过，高效的游戏程序需要我们程序员尽量减少这种事情发生的频率。

由于操作系统需要一定的数据和代码才能工作，而 Windows 占据了 4G 空间里的高段 2G，用户程序能实际使用的只有 2G 空间了（Windows 的 Advanced Server 版本可以只留 1G 工作，这样用户程序就多了 1G 内存可以使用）。

当然，Windows 也提供一些手段使用更大的地址空间，但是限于其是一个 32 位操作系统，想做到 64 位的寻址，并没有直接的编程手法，所以依旧需要通过一些 API 调用。其实现在看来，2G 空间对于游戏来说已经足够用了，如果你在写游戏时发觉内存还不够用，首先该检讨自己的程序设计是否有问题，是否存在改进的方案。

实际上，操作系统的具体实现决定了还需要用额外的状态去表示一些中间情况：比如处于等待的备用状态，或者是正在切换时的运行状态，还有已经在逻辑上终止了运行正在注销与相关联进程之关系的状态。这些属于操作系统实现的问题，在一般的软件开发中不用关心。

正确地了解基础知识，并不一定要在编程工作中用到它们。但不理解或是不能正确理解操作系统是怎样工作的，却经常给程序员的编程带来困扰。大学里的计算机相关专业，通常会把操作系统原理列入必修课程，这种设置绝对有它的必要。

可惜的是，在我这些年从各个大学的毕业生招聘面试过程中，却很难找到几个学生可以清晰地把进程线程这些操作系统中基本概念正确地描述清楚的。对于即将成为程序员的科班出生的学子们，急切地去学习诸多新潮的开发工具、流行的新名词，是否过于急功近利了一点?

进程、线程和虚拟内存，Windows 都提供了足够丰富的管理控制接口给程序员使用。进程以及主线程属于操作平台相关，一般的应用并不需要对此有太深刻的了解；而虚拟内存的管理，对于一般的开发工具，比如 Windows 平台下的 Visual C++等，做了适当的封装，C 接口 malloc 和 C++的 operator new 关键字之确实都隐藏了和操作系统相关的知识，如果游戏程序到了必须自己管理私有堆和操作虚拟内存页面的加锁解锁等底层操作，我们就该检讨自己的工程哪里出了问题了。

Windows 多线程程序的使用，对于游戏程序，经常能带来一些意想不到的好处。但是，它毕竟属于比较高阶的 Windows 编程方法，对于没有太多实战经验的 Windows 编程新手，绝对不推荐在实际项目中使用，这个话题也留在稍后一点再去叙述。

5.1.3 Windows API 和 DLL

Windows 工作在保护模式下。所谓保护模式，是指可以保护计算机的各种资源设备，让应用程序不能随心所欲地控制它们。在 386 保护模式下，赋予运行的代码 ring0 到 ring3 四个权限级别之一。用户加载的程序代码都是运行在最低级别（ring3）下，几乎想做任何与人交互的事情，都不可能直接使用对处理器的控制指令实现，而需要经过拥有更高权限的操作系统来操作。

想像一下，一个纯粹的 100%自己编写的不调用任何他人书写的代码指令的程序，即使它内部做了再多的运算，也不可能把数据传递出来，在程序运行完后无声无息地消失，进程曾经拥有的一切就此消失，好像从来没有开始过一样，这种程序是毫无意义的。

所以，任何一个 Windows 下的应用程序都会去调用 Windows 操作系统提供的一些代码，做一些输入输出或者控制某些硬件的工作。通常，这些供应用程序使用的代码通过被称为 Windows API 的接口来调用。

最基本的 Windows API 是被写好分门别类放在一些后缀为 DLL 的文件里的，存放应用程序的 EXE 文件的文件头里描述有该文件需要使用哪些 DLL 的信息，这些信息在操作系统加载 EXE 的过程中被解析使用，

把相关的 DLL 里的代码加载到为正在启动的 EXE 分配的新进程的虚拟内存空间中，和 EXE 本身携带的运行代码绑定在一起。由于 DLL 里的代码在 EXE 加载的时候动态绑定，所以 DLL 文件被称为动态链接库。

被加载到内存空间的动态链接库可以被多个正在运行的进程所共用，在物理上可能只存在一份；但是逻辑上，每个进程空间的虚拟内存中都映射有一份独立的版本。动态链接库可以由操作系统负责帮 EXE 加载绑定，同样，这个加载的操作也开放给所有应用程序自主使用，加载动态链接库的接口 Load Library 本身也存在于 Windows 最核心的 kernel32.dll 中，所以可以说，kernal32.dll 是每个应用程序至少必须绑定的运行库，是 Windows 的核心。

在 DLL 文件里，使用一系列的可书写的字符串来表明每个 API 的位置，这种使用字符串来实现动态绑定，而不是直接保存接口地址的方法，是软件设计中减少模块间耦合度的一种常用手法，也保证了我们的应用程序可以顺利地运行在不同版本的 Windows 上，即使是 Windows 9x 和 Windows 2000 这样内部结构截然不同的操作系统。同样，除了加载 EXE 时操作系统自动做 DLL 中 API 的动态绑定外，我们也可以使用 GetProc Address 来获取 API 的地址。

在很大程度上，在 Windows 下编程，除了自身软件的运行逻辑外，无非就是调用各种 Windows 的 API，把运行结果和中间状态输出以及从外部获取输入，所以在 Windows 下编程少不了一份 Windows API 的速查手册，几乎所有的 Windows 编程老手都会向新手推荐 MSDN，我也不例外。

在 MSDN 里，几乎可以查到任何有关 Windows 开发的知识，遇到 Windows 编程问题，不要忙着问人，查阅一下 MSDN 吧。如果你没有订阅过光盘版本的 MSDN，微软的网站上提供最新的在线版，使用非常方便（http://msdn.microsoft.com）。

5.1.4　COM

COM——组件对象模型，如今已经存在于 Windows 的每个角落。在不少技术书籍里，提到 COM 就会牵扯出 OLE，ActiveX，甚至更多的概念。COM 的技术涵盖面是很广的，讲解清楚 COM 也不是件容易的事情，要全部了解学习则更加困难。对于不需要触及 Windows 太多的我们，游戏开发人员，暂时是不需要全部理解它的，这里只讲讲有关 COM 知识

的很少一部分。

先考虑一个问题：如果 Windows 以及后来不断的完善和扩展都建立在 API 的基础上会怎样？

我们将面对巨量的 Windows API，以至于最后连 API 的名字都很难取了，而且所有的系统功能都被描述在一个扁平的名字空间里。老的功能一旦升级改动太大，还需要换用新的名字，避免直接替换掉老版本使得以前的程序不能继续运行。

使用 C 语言之外的语言开发供使用的动态链接库也会变得很困难。比如，C++等支持对象概念的语言，一个函数光用函数名已经很难准确代表唯一的实现。只能通过字符串变换，使用冗长的已经难以被人阅读的接口名字来包含更多的诸如函数参数、所属类等信息。

COM 定义了一个通用的二进制层面的模型，可以用普通而便捷的方法扩展 Windows 所提供的服务，把接口、实现、对象分离，让各种语言和开发工具写出的代码互相调用，并且它保存有发行过的所有版本的内容，不用担心老的软件的兼容性问题。

COM 对于调用者来说，只能看见接口。接口之间采用单继承的模式，所有 COM 接口的基类都是 I Unknown。每个 COM 对象的实现被隐藏起来，并可以拥有多个接口。使用时，使用动态类型识别的技术把对象的 IUnknown 接口转换为需要的接口以方便调用相关的方法。

当然，这里的动态类型转换没有用 C++中的 dynamic_ cast，而是使用 IUnknown 强制要求所有 COM 对象必须实现的 Query Interface 方法。这样做的好处是，可以在二进制层面统一对象模型，甚至允许在转换到对应的接口时才临时将接口相关的代码加载进来。允许动态加载是 COM 的基本特性之一，在 COM 对象接口背后的隐藏面，我们不知道 COM 对象是否在当前进程，甚至不知道是否在本地机器。我们只需要知道在我们得到指定 COM 对象后，可以找到需要的接口进行调用。

大多数情况下，COM 依然依赖 DLL 来实现，但是我们只能把 DLL 看成 COM 的一种载体，COM 本身的这些设计相对使用 DLL 做载体的 API 体系在设计层面上是非常不同的。COM 带来了极大的灵活性，可以让系统中重用更多的东西。

对于使用 COM 组件的人来说，运行时他只需要了解他想使用的 COM 对象在哪里，想使用的 COM 对象的某个接口的名字是什么。COM 对象只在用户想使用的时候才被激活，这个 COM 服务甚至可以不在当前进程中提供，这样就存在启动服务进程的问题。在解决怎样找到需要的 COM 对象激活的问题上，需要用到 COM 服务管理器 SCM，每台支持 COM 的主机都有一个本地的 SCM，处理本地 COM 对象的激活，并且把远程 COM 对象的激活请求转发出去。

在 Windows NT 中，用于本地的 COM 对象激活是由 Windows NT Services 负责的，它比完整的 COMSCM 简单得多，其仅仅暴露了一些 API 供用户调用。激活（创建）一个 COM 对象调用 OLE32.DLL 中的 CoCreate Instance 即可。每个提供 COM 对象的人都会为这些对象起一个名字，为了方便检索，名字用一个数字 ID 来表示（同时还存在方便阅读的字符串别名）。

由于不可能全世界使用一个中心数据库来为每个想制作 COM 组件的人分配不同的 ID，COM 使用了一种叫做全局唯一标识符（GUID）的技术。GUID 是一个 128 位的巨大数字，这个数字大到即使在没有直接联系的两台机器上生成也几乎不可能相同，当然，为了保证其在统计意义上的唯一性，在生成算法上还尽量利用了时间信息、机器网卡信息等。

每个独立 COM 对象的 ID 被称为 classID——CLSID，用这个 ID 向 SCM 查询就可以得到它的 IUnknown 接口指针了。需要从 IUnknown 里查询到需要的接口，再使用 interfaceID——IID 就够了。

作为一个 COM 组件，静态保存于 DLL 中。一般只需要暴露出一个单件对象出来即可，这个对象会被实现为类工厂，从这个对象的接口中，我们可以继续创建出该组件的其他对象。

而这个 DLL 的文件名连同类工厂对象的 CLSID 存在于 Windows 的 SCM 中，被注册在注册表里的 HKEY_ CURRENT_USER\Software\Classes 或者是 HKEY_CLASSES_ ROOT 下。打开注册表编辑器，你会发现巨量的 COM 对象已经被注册到你的 Windows 中了。

跟 Windows 下游戏关联最密切的 DirectX 就是基于 COM 技术的。我们可以从 DirectX 组件中取到显卡、声卡、鼠标、键盘的控制接口，

然后对其编程。Windows 的各种 C++开发平台都带上了和 DirectX 有关的.h 描述文件，甚至为了简化使用，还在一些静态链接库中提供了 COM 之外的更为简洁的初始化方法。如果对 COM 有一定的了解，DirectX 的基本应用甚至不需要专门的学习，有一份 MSDN 作为参考已经足够。

实际上，在 Windows 下开发游戏，DirectX 并不是必修课。谈起游戏编程，就联想到 DirectX 实在是一种误解。DirectX 只是提供了一种更直接的控制硬件的渠道而已，对于平面图像显示，如果用 Windows 的 GDI，更能和其他 Windows 程序配合得体，鼠标键盘的输入走 Windows 的消息体系也不无不可，声音的播放，也早已在 winmm.dll 中提供了编程接口，至于 3D 显卡的特别性能使用，集成于 Windows 中 OpenGL 的模块也是一个不错的选择。

说这些并非主张不去学习 DirectX 的使用，更不是说 DirectX 不好，只是想说明，DirectX 并非编写 Windows 游戏的必备技能。而且，游戏程序本身，最终和操作系统的硬件级打交道的部分少之又少，作为一个游戏程序员，不要太过于注重这些技术细节了。

如果打定主意玩转 DirectX，那我觉得事先充分理解 COM 是可以事半功倍的。可惜 COM 并不是那么浅显易懂的技术。我个人至今也没有在自己的项目里大量使用 COM，可以说，对 COM 的理解相当有限。如果在这里对 COM 的介绍再多一些，恐怕就要贻笑大方了。要搞明白 COM 为什么会设计成现在这个样子，建议阅读一下 Don Box 的著作 *Essential COM*，中译《COM 本质论》。

5.1.5 Windows 的窗口和消息处理与传递

如果一个程序只有命令行的输入和对应的文件输出，那么在 Windows 下写程序和在别的操作系统下没有什么区别。但是，Windows 最初之所以叫 Windows，就是因为它的主要特色之一——大多数应用程序，都有一个图形窗口方便与用户交互。

图形界面是 Windows 应用程序的一个重要组成部分（但不是最重要的部分，我们的程序更重要的是完成特定的任务，而非设计出界面供大

家操作，不是吗？），Windows 系统的交互界面就是基于窗口的，即使你想写一个全屏的游戏，并且直接控制鼠标键盘，也需要创建至少一个窗口，这样 Windows 才可以让这个游戏和别的应用程序间能够互相切换。所以，在 Windows 下写程序，少不了跟窗口打交道。

Windows 一旦启动，就有许许多多看得见、看不见的窗口被创建出来附属于不同的进程工作着。我们不要从字面上把 Windows 的窗口理解成方方正正有标题栏、菜单、关闭按钮的东西。实际上，所有能获取外界指令，并且依照这些指令的指示来工作的对象大多数都是窗口。窗口的外在表现仅仅只是窗口对象本身按指示把自己绘制表现出来而已。所以，绝大多数按钮是窗口，任务条上的小图标是窗口，对话框上的一行文字也可能是窗口。它们能够分析和传递的指令都是一种用数字表示的叫做消息的东西。各种消息在各个进程所管辖的不同窗口间流动传递，触发相应的代码去处理，最后我们就看见了色彩缤纷的 Windows。

既然每个窗口都是一个独立的对象，从计算机语言的角度上讲，让对象发生某件事情，最直接就是去调用对象的相应方法。为什么 Windows 不这样设计呢？答案很简单：一、Windows 的应用程序可以用任何一种计算机语言书写，用数字作为消息编号，然后程序自己分析这些编码处理，这种接口最容易为所有语言描述；二、不同的窗口可能属于不同的进程，对于用户级的进程，互相之间是被隔离的，互相调用对方的代码，为常规方式所不允许，从安全角度上讲，也不被提倡。所以只能给对方窗口一个信号，由窗口本身来决定如何处理。

同样的道理，窗口对象也不能用内存对象的指针来管理。从程序设计语言上讲，指针仅仅是一个内存地址，而操作系统下，各个进程都有独立的虚拟内存空间，光用内存地址肯定是不够的。这里，Windows 引入了句柄（Handle）的概念。

可以想像一下，一个堆满杂物的抽屉，里面有或大或小各式各样的东西，检索这些东西是很困难的。如果我们给每样物品都拴上绳子并在绳子上标上号，然后想找什么东西，只要知道号码，把那根绳子一拽，就找出来了，Windows 的句柄就是这些标了号码的绳子。句柄本身毫无意义，句柄表示的对象才是有意义的。

大多数Windows API需要调用者告诉需要操作的对象是什么，全部都是通过传递一个对象句柄来实现。不光每个窗口有自己的句柄，Windows里大多数对象都有自己唯一的句柄，比如一个打开的文件、一个进程、一块分配出来的虚拟内存等等。使用消息机制让不同的对象之间协同工作，用句柄来指定不同的对象，不仅仅在Windows，几乎在所有多任务操作系统中都是这样。

下面我们来看一个很简单的，拥有一个自己的窗口的Windows程序：

严格意义上说，这个程序有很多纰漏，比如没有检查一些API是否被成功调用，但是，为了使大家更注意到重点，我删除了一些本应该写上的代码，以缩短其篇幅。

```
1.#include <windows.h>
2.int WINAPI WinMain (HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      PSTR szCmdLine, int iCmdShow)
{
3.      RegisterMyWindowClass();
4.      HWND hwnd = CreateWindow ("MyWindow",// window class name
                    "窗口的标题"),       // window caption
                    WS_OVERLAPPEDWINDOW,// window style
                    CW_USEDEFAULT,      // initial x position
                    CW_USEDEFAULT,      // initial y position
                    CW_USEDEFAULT,      // initial x size
                    CW_USEDEFAULT,      // initial y size
                    NULL,               // parent window handle
                    NULL,               // window menu handle
                    hInstance,          // program instance handle
                    NULL);              // creation parameters
5.      ShowWindow(hwnd,iCmdShow);
6.      UpdateWindow(hwnd);
7.      MSG        msg;
8.      while (GetMessage(&msg,NULL,0,0))
9.      {
10.         TranslateMessage(&msg);
11.         DispatchMessage(&msg);
12.     }
13.     UnregisterClass("MyWindow",hInstance);
14.     return msg.wParam ;
15.}
```

这并不是一个完整的可运行的程序，里面还有一个自定义函数RegisterMyWindowClass暂时没有列出来，不过没关系，那些马上

就可以看到的，还是先来看看这区区 15 行代码吧。

首先，这段程序里用到了很多 Windows 的 API 和一些预定义的类型，这些被分别定义在微软提供的各个.h 文件中。如果你无法记清楚每个具体的地方，只需要在程序最前面写一句#include<windows.h>就够了。

对于 Windows 下的 EXE 文件，文件内会描述一个程序启动的入口位置，对于 C 或者 C++语言来说，没有启动入口这个概念，所以编译器通常会帮开发人员封装一些代码，并规定一个函数为程序逻辑上第一个被调用。Windows 下，这个函数被默认为 WinMain（但并不是一成不变的，这取决于编译器的实现）。

正如 DOS 以及很多别的操作系统下的 C 编译器会在自己产生的启动代码中获取一些用户可能关心的东西，比如控制台的命令行参数以及参数个数等信息，然后传递给规定的启动函数 main 一样，用户定义的 WinMain 也将接收到一些有用的信息：当前运行的这个程序的实例句柄；同样的 EXE 前一个被运行的程序的句柄；命令行的参数字符串和程序被指定以什么窗口状态启动。

第 3 行是向操作系统注册一个自己的窗口类别，因为如前文所述，Windows 里运作着各式各样的窗口，即使你的应用程序的窗口和系统拥有的某种窗口长得一模一样，它们处理消息的方法也一定有所不同，必须向 Windows 注册一个自己的新的窗口类别。至于 RegisterMyWindowClass()是怎样工作的，放到后面再讲解，这里先关注程序的主要流程。

第 4 行创建了一个窗口，参数传入了窗口的种类，通常是我们自己注册的种类。如果想创建出系统提供好的窗口类型，比如编辑框、按钮等，也可以传入预先设置好的种类。另外还需要传入样式、名字、坐标、大小、所属进程这些对于窗口的描述参数。

这一行运行结束后，只要创建成功，属于当前进程的内存中就会被构造出一个窗口对象，API 返回了这个窗口对象的句柄，方便以后使用。其实，这里还有一个隐含的参数，那就是当前的线程，因为 Windows 的消息管理是基于线程而不是进程的，所以，每个窗口都被归属到创建它的线程。

第 5 行是一个关于窗口管理控制的 API——ShowWindow。从字面上理解，它让窗口对象按预先设置好的属性显示出来。但它实际的工作不能这样理解，实际上，这个 API 仅仅只是设置窗口的显示状态，而非驱使窗口显示。

每个窗口都会有很多的显示状态，比如隐藏起来，最大化显示，最小化显示，从最大化显示恢复到常规显示等。这个状态值被储存到窗口对象中。ShowWindow 仅仅只是改变这些值而已，没有立即引起窗口具体外观表现的任何变化。

第 6 行才是让刚才创建出来的窗口显示在屏幕上的关键语句 UpdateWindow。这个函数的本质是发送一个叫做 WM_PAINT 的消息给指定的窗口。让窗口的消息处理函数去处理遇到 WM_PAINT 情况。正如前文提到过的，Windows 操作系统以消息机制来控制协调当前机器中的各个窗口，让它们工作起来。让一个窗口做点什么，通常是发送指定的消息给它，由它自己来选择该怎么做。

发送消息的方式有两种，一种是让对应窗口的消息处理函数运行一次，立即处理这个消息，这可以通过调用 SendMessage 来实现。另一种是把消息推入窗口所属线程的消息队列，等该线程去轮循消息队列，逐个处理，那就是 PostMessage。由于 PostMessage 是延迟处理的，所以，如果需要知道消息的处理结果，就需要用 SendMessage Callback 了。这里，UpdateWindow 通过调用 PostMessage 把 WM_PAINT 消息推入消息队列实现的。

不过，我们不能自己直接调用 PostMessage 来实现和 UpdateWindow 完全相同的操作。UpdateWindow 做得更多，Windows 在图形界面的管理上做了很多的优化，尽量让每个窗口做更少的刷新重绘操作，每个窗口都有自己的需要更新区域，但 UpdateWindow 还会检查这个区域，如果为空，就不会做发送 WM_PAINT 的操作了。

到这里为止，程序还没能在屏幕上显示任何东西，下面才进入消息处理的循环，我们创建的窗口才得以工作。

第 7 行声明了一个消息的结构，由第 8 行的 while 循环中的 GetMessage 来循环获取当前线程中消息队列中的消息。如果

GetMessage 返回 0，则表示收到了一个 WM_QUIT 退出消息。

如果是用 GetMessage 获取指定窗口的消息，则可能因为窗口句柄无效而返回一个无效值-1，所以这种情况一定不要直接判断 GetMessage 的返回值为非 0，避免程序进入死循环。

通常在游戏程序里，除了让窗口处理本身自己的消息，还需要做很多额外的逻辑，不太适合用消息机制驱动，这个时候，一般会在 GetMessage 前先调用 PeekMessage，这样可以检查是否有没处理的消息。在没有消息的时候，也让程序不阻塞住而继续运行。值得注意的是，在没有接到窗口消息而试图去做别的工作的时候，我们也应该适当地调用 Sleep 这个 API 以释放线程的控制权。单纯依赖操作系统以超出时间片的规则来调度线程会浪费很多的 CPU 时间，使自己的程序不具有良好的协作性。这样的程序表现出来就是一旦运行，就占据了 100% 的 CPU 时间。

第 10 行调用了一个叫做 TranslateMessage 的 API，这个 API 是用来翻译一些键盘消息的。因为 Windows 环境下，直接对应于每个按键的消息被叫做 virtual-key 消息，而在很多情况下，知道用户按了什么键并没有了解用户做了怎样的输入操作重要。尤其是，我们在输入中文的时候，汉字是通过多个按键组合选择出来的，而不能单单依靠检测用户按了什么键。另一个普遍存在的现象是：小键盘和主键盘上都有数字符号的分布，对于按键本身来说，它们是不同的键，而对于程序逻辑，我们常常需要把它们看成是相同的逻辑。TranslateMessage 帮助我们处理这类问题，让窗口获得用户输入的逻辑含义的消息。

第 11 行 DispatchMessage 是一个关键，它所做的工作就是从当前线程的消息队列中取出一个消息，从消息的窗口句柄信息中，获知该窗口的消息处理函数的地址，然后把消息相关的参数以及窗口句柄本身作为参数调用这个函数。

注意：消息队列里不仅仅有窗口消息，还可能有线程消息。所有的线程消息中，窗口句柄被指定为 NULL。由于没有所属窗口，这种消息是会被 DispatchMessage 忽略掉的。用户可以通过 PostThreadMessage 主动向线程发线程消息。如果想处理，应该在GetMessage后判断一下MSG结构中的hwnd值是否为NULL。

第 13 行注销了我们开始注册的窗口类型，做到了资源的释放。其实，即使你的程序不释放任何曾经创建出来的资源，进程结束后，操作系统也会帮你做这些回收工作。但对于程序员来说，自己申请的资源自己释放，是一个良好的编程习惯。

下面我们来看看窗口的消息是如何被处理的：

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                         WPARAM wParam, LPARAM lParam)
{
switch (message)
{
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
```

我们需要写一个类似上面这样的消息处理函数，当其他代码直接或间接调用了 SendMessage，或者调用 DispatchMessage 处理消息队列里的消息时，等于调用了这个函数。

这里，这个函数写得非常简单，真正的应用程序会比这复杂很多。由于需要去处理不同窗口的不同消息，消息的处理也不一定是恒定不变的。所以我不赞成让其中的 switch 语句变得过于复杂，写出一个巨型的函数，那不符合软件设计之道，具体怎么设计就要看程序员的功底了。其实，使用现成的封装了这些消息处理的框架类，比如微软的 MFC，就

是一个不错的选择。

这里只处理了一个消息，即窗口的 WM_DESTROY 消息。这个消息是在窗口接收到关闭 WM_CLOSE 消息后，由缺省的窗口处理程序负责发出的。由于我们的程序只创建出了一个窗口，所以我们认为只要这个唯一的窗口被销毁，程序就应该结束运行了。基于这一点，这里的处理就是调用 PostQuitMessage，即向当前线程发出一个 WM_QUIT 消息。回忆前一段程序，GetMessage 正是在接收到 WM_QUIT 消息后退出 while 循环的。从这里我们也可以看到，Windows 下的程序多数讲究协作，而不是强制别的部分干指定的事情，这样才得以让看似分离的模块协同工作起来。

对于其他的消息，如果我们不需要特殊的处理，就一定要调用 DefWindowProc 让系统帮我们做最正常的处理。

值得一述的是，貌似简单的工作背后，往往有相当复杂的设计。这里提到的窗口的关闭工作过程就很复杂。它是这样工作的：WM_DESTROY 这个消息是在窗口关闭过程中被触发，由返回值决定能不能被销毁。这一点上，可以看出 Windows 在设计时考虑得非常周全，很多地方使用了设计模式中的模板方法，WM_DESTROY 就可以看做一种消息回调机制的使用。它给用户一个安全释放资源或者干别的一些事情的机会。所以，WM_DESTROY 消息通常不由用户去主动触发，而是在调用某些 API（例如 DestroyWindow）时由 API 主动触发的。

WM_CLOSE 这个消息却不同，Windows 对 WM_ CLOSE 的缺省处理做了许多事情，包括销毁掉我们创建出来的窗口。所以，在这个程序里，我们没有任何一个地方显式地写出了 DestroyWindow 来销毁 CreateWindow 创建出来的窗口，但在点击窗口关闭按钮或按下 Alt-F4 的时候，这些操作默认产生的 WM_CLOSE 消息的缺省处理函数，会自己去调用 DestroyWindow 函数。

但如果我们的程序并没有通过 WM_CLOSE 消息这个途径去关闭窗口，比如在菜单里加一个退出选项，就应该在菜单消息的处理函数中主动调用 DestroyWindow 避免资源泄露。

关于 SendMessage 和 PostMessage

如果消息发出的对象都是本线程创建出来的窗口，则 SendMessage 其实是直接找到窗口的 WndProc 进行调用，而 PostMessage 则把消息放到窗口所属线程的队列。

如果 SendMessage 的对象属于另外一个线程，问题就复杂多了。这个时候，实际是 SendMessage 把消息压入目的线程的消息队列，等待回应。接收方的线程进入到需要读消息队列的 API 时（通常是 GetMessage 或者 PeekMessage，WaitMessage），会立即处理这个消息，并将结果通过消息反馈回发出的线程。

在 SendMessage 等待反馈的期间，线程是等待状态的，不过，如果有另外一个线程向其发消息，可以在这段时间被处理，这样设计的好处是多个线程交互通讯的时候，减少互锁的可能。另外，我们所说的消息队列，在 Windows 内部实现并非一个单纯的单一队列结构。

实际上，为了处理这种线程间的消息通信和线程内消息发送，内部是有几个物理上队列的。通过 Post 过来的消息，和在线程间 Send 的，不同来源的消息被存放到物理上不同的队列中，方便操作系统调度处理。

另外，Windows 的系统消息在跨进程间发送时，有时会被特殊处理，这是因为消息的参数有可能携带一块内存，比如需要传递一个字符串的情况。这个时候，Windows 会把消息的内容复制到目的进程的地址空间里，让逻辑上的参数可以正确地传递。

这些是 Windows 内部的具体实现，对于开发 Windows 应用程序的程序员没有太大影响。只是，了解这一点，可以增进我们对 Windows 的理解，可以在程序出了奇怪的错误现象时，帮助我们分析问题的所在。

上面提到的消息处理函数，每个窗口都必须绑定有一个。绑定在窗口上的消息处理函数的地址可以由 SetWindowLong 来设置。但这里并没有调用 SetWindowLong，那么窗口消息处理函数是怎样绑定到窗口上的呢？

其实，我们的窗口在创建时，自动将前面注册过的窗口类型中记录的缺省消息处理函数绑定在自己身上。前面略过没列出的 RegisterMy

WindowClass()的实现就是注册这么一个我们自己的窗口类型，注意中间的 wc.lpfnWndProc 设定：

```
void RegisterMyWindowClass()
{
    WNDCLASS    wc ;
    wc.style         = CS_HREDRAW | CS_VREDRAW ;
    wc.lpfnWndProc   = WndProc ;
    wc.cbClsExtra    = 0 ;
    wc.cbWndExtra    = 0 ;
    wc.hInstance     = GetModuleHandle(0);
    wc.hIcon         = NULL ;
    wc.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_ BRUSH) ;
    wc.lpszMenuName  = NULL ;
    wc.lpszClassName = "MyWindow";

    RegisterClass (&wc);
}
```

这个函数没什么好具体解释的，因为主要就是调用 RegisterClass 注册一个叫做"MyWindow"的我们自己的窗口类别。所有的参数都是填在 WNDCLASS 结构中。其中就有上面提及的消息处理的回调函数 WndProc，有窗口的一些属性，窗口的背景用什么方式填充，默认的鼠标是怎样的，窗口的图标是什么，等等。通过查阅 MSDN 可以得到 WNDCLASS 里每一项应该填些什么的说明。

我们可以看到，WNDCLASS 结构里有 hbrBackground，里面指定了一个窗口背景的刷子。我们暂时不需要去理解刷子的含义，只需要了解，窗口需要刷新背景时，就会调用这种刷子来刷新，这里是一个白色的刷子，所以窗口会呈现为白色。

实际上，如果是一个游戏程序，如果整个窗口的背景都是游戏的画面，或者一个使用 DirectX 做全屏显示的窗口，都可以把这个设置为 NULL，系统就不会帮我们做刷新背景的操作而由我们自己来绘制。

hInstance 应该填写当前进程的句柄，知道这个窗口类别是属于哪个进程的，这里用 GetModuleHandle 来获取当前的进程句柄（暂

时这样看），其实，在WinMain函数的传入参数里也有它。甚至，这个参数很可能是为了兼容老的Windows版本才设定的，因为即使你直接填写NULL，Windows 2000也会帮你得到当前的进程句柄。

RegisterClass注册的类型只跟当前进程有关，也就是说，你的进程注册的窗口类别，别的进程是不能使用的，同样，你也不用担心别的进程会因为注册一个同名类别而导致冲突。

HINSTANCE HMODULE 和 PE 文件

查看MSDN可以发现，GetMouduleHandle返回的是HMODULE类型。WNDCLASS结构中的hInstance以及WinMain的传入参数中都提到了另一个类型HINSTANCE。这里我们的程序将它们等同了。其实，这个值只是一个内存地址而已，但这个地址究竟是什么？如果我们追根溯源的话，这个地址通常指向了一个叫做IMAGE_NT_HEADERS32的结构，这个结构在winnt.h有准确的定义。

没错，这就是PE文件的内存映射地址。PE是32位Windows的可执行文件的数据结构。在Win32中，DLL和EXE没有什么本质的不同，他们都是PE文件，PE结构描述了程序需要的代码，数据等信息。加载一个exe或者dll文件，只是对PE的解析而已。

所以，只要你愿意，甚至可以直接加载exe或者dll文件（甚至只是一片数据）而不通过相应的API，当然那需要对PE格式有一定的了解。在Windows 2000之后，提供了imagehlp.dll帮助你做这些事情。

从这个最简单的程序可以看到，窗口创建出来后，每一步的处理都不是自动的，Windows的应用程序并不比DOS下的应用程序神秘。微软也没有向程序员隐瞒什么。每个窗口都是由用户写的代码一行行运行工作起来的。我们在设计Windows程序的时候，应该经常回忆这些工作流程，防止程序不小心陷入死循环或者忘记释放某些资源。

作为游戏程序员很容易犯的错误是：并没有系统学习Windows的工作机理，就抄下书本上的范例代码让其工作，这样往往并不贴切。这也是这里详细讲解一个最基本的窗口程序逐行代码背后东西的用意了。Windows窗口系统的工作流程的每一步都有迹可寻，当写下代码时，想

想程序会按怎样的步骤去工作，遇到疑惑的时候多查阅一下MSDN自然能搞清楚。

5.1.6 Windows GDI

对于提到Windows下游戏开发就想到DirectX的人来说，我更愿意他们能先把Windows原生的东西搞清楚。其实，华丽的Windows的图像管理和显示都是基于一套通用的图形设备接口实现的（Graphical Device Interface），也就是GDI。高效的图像显示并不一定取决于用到多么底层的显示设备控制，而在于处理图像数据的方法。真正把图像数据传输到屏幕上这一简单的过程，并非总是瓶颈所在。

埋怨GDI效率过低的游戏程序员请考虑一个问题，为什么Windows自己能在很高的分辨率下，让如此多的窗口、对象都流畅地显示给用户看呢？你的游戏程序有处理这种情况的能力吗？

1. 设备相关信息（DC）和GDI对象

DC（Device Context）是GDI的一个重要部分。DC的概念被设计出来，是来源于我们的绘图操作可能被画到不同的设备上面，除了屏幕，还可能是打印机、内存或者是我们新开发出来的硬件设备。只要新设备提供一个符合Windows标准的DC，原来的软件就可以在事先不知道这种设备的情况下，依旧可以正确工作。DC是一个结构，它描述了一个设备的相关信息，让GDI的操作知道怎样把数据画在上面。

Windows程序中，用得最多的是屏幕DC。每个窗口都可以得到自己的DC，让绘图操作可以画在上面。此操作用GetDC这个API就可以办到。用完DC后，要记得使用ReleaseDC释放掉。

每个DC有自己的映射模式，可以用GetMapMode来获得映射模式，并用SetMapMode设置映射模式。所谓映射模式，它决定了在DC上作图的坐标单位。对于只对显示做操作的游戏，通常用MM_TEXT以像素为单位的映射模式就够了，坐标原点在左上角，X轴向右延伸，而Y轴向下，这样很符合一般程序员的习惯。

因为GDI允许向DC写字，作图，所以这些绘图的函数需要用的资

源需要事先放在DC里面，比如字体、画笔等。这些东西都是以GDI的对象形式存在的，一个DC在创建出来的时候，系统会给它分配一些默认的GDI对象。

当想使用新的GDI对象时，比如换一种指定的字体，就应该先用CreateFont创建出字体对象，然后用SelectObject把它选入DC。这里，选入DC的GDI对象最安全的删除方法是，先将其选出来，再调用DeleteObject删除，不过，现在大部分Windows系统都允许直接删除被选入DC的对象了。

2. 位图

在制作Windows游戏时，通常复杂花哨的GDI对象操作方法很少用到，更多的是操作位图。位图是GDI对象的一种。向DC直接写文字或是画点画线的方法在图形游戏设计中并不被推荐，这些增加了游戏程序本身的系统相关性，游戏的表现结果也变得和操作系统相关。一般是由设计游戏图像引擎的人在引擎中用自己的代码实现对位图的作图以及写文字的操作。所以在这里，也不再深入地讲解GDI更多的知识（如果需要，许多讲解Windows的书籍和MSDN里都有系统的讲解），让我们直接来看和游戏图像直接相关的位图对象。

GDI支持的位图分两类，第一类叫做设备相关位图(device-dependent bitmap DDB)。这种格式的位图只能由相关设备驱动程序支持的结构来描述它，根据设备的特性，驱动程序会为位图设计一个适合的格式，对于设备本身，位图工作得很高效。但是，如果程序员想根据自己的意愿对位图做修改的话，由于不可能去了解每种设备的结构，所有操作只能通过设备驱动程序间接地完成，这种操作通常非常麻烦和低效。游戏程序几乎不会采用这种位图结构，我们这里也不再去讨论它。

第二类是设备无关位图（device-independent bitmap DIB）。这种位图的结构简单固定，方面了程序员去直接修改位图的内容。当位图被画到DC上去后，才由系统根据DC做相应的转换。对于游戏来说，通常我们的操作对象DC就是内存，或者从内存翻转到屏幕DC去显示（关于翻转和双缓存的概念见第4章）。内存设备本身就是内存地址空间，几乎不需要去转换。而屏幕显示设备，大多只需要做一些色彩格式的转换，

这种转换也可以做得非常迅速，我们不太需要考虑此处的效率问题。

DIB 位图中保存图像本身的数据结构就和前一章提到的位图形式并没有什么差异，都是线性地保存有每个像素的颜色。除去现在制作游戏已不太常用的单色、16 色，以及 256 色模式外，高彩和真彩模式最为 Windows 游戏程序所常用，本书里也只讨论这种非调色盘颜色的 DIB 位图，如果读者需要做更多了解，可以去查阅 MSDN 和更专业的书籍。

每个 DIB 位图都有一个信息头，描述了位图的尺寸等信息。在微软提供的 Windows 开发包里，在 C 语言的.h 文件中都可以找到这个结构的定义。

BITMAPINFO 结构有两个成员：

```
BITMAPINFOHEADER bmiHeader;
RGBQUAD          bmiColors[1];
```

BITMAPINFOHEADER 则是这样构成的：

```
DWORD   biSize;
LONG    biWidth;
LONG    biHeight;
WORD    biPlanes;
WORD    biBitCount;
DWORD   biCompression;
DWORD   biSizeImage;
LONG    biXPelsPerMeter;
LONG    biYPelsPerMeter;
DWORD   biClrUsed;
DWORD   biClrImportant;
```

之所以分开定义，是因为 DIB 位图支持多种颜色格式，不同的颜色格式对应的结构长度不同。它们共有的部分是前面一个 BITMAPINFOHEADER，后面的 RGBQUAD 通常是扩展、保存调色盘或其他关于颜色的信息。

我们在创建或描述一个 DIB 位图的时候，需要填写这个结构，而且不要把参数填错。具体每个参数的含义，由于查阅资料（MSDN）可以找到，所以花文字罗列出来是浪费读者的阅读时间以及笔者的写作时间，这里只提几个容易被人忽略的问题。

biSize 填入这个结构的大小，在 Windows 定义的很多种结构中，第一个元素都是当前结构的大小，这方便了日后的扩展。以后出现新的版本时，就可以在结构后面添加新的参数，而程序可以通过结构大小的数值来判定新老版本，并对老版本缺少的参数做缺省处理。

biWidth 和 biHeight 描述了位图的宽度和高度。一般的 DIB 创建出来后，可能某些程序员会略显奇怪，因为数据排列上的第一条扫描线是位图的最底下一行，然后 Y 轴坐标是由下至上的。可能 DIB 结构设计之初，是为了符合数学上的笛卡儿坐标系统的坐标轴方向吧。如果你更习惯从上至下的位图描述方式，也没有关系，只需要把 biHeight 设置成负值就可以了，这样，位图的左上角的像素就对应到了位图数据内存地址的最前面。

如果使用 16 位颜色，当 biCompression 是 BI_RGB 的时候，高位是没有意义的，颜色为 555 模式。当想使用 565 格式时，需要把 biCompression 设置成 BI_BITFIELDS，然后在 BITMAPINFOHEADER 的结构后面跟 3 个 DWORD 写成红色、绿色和蓝色的掩码 0xf800,0x07e0, 0x001f（如果你不使用 BITMAPINFOHEADER 结构而使用 BITMAPV4HEADER 或者 ITMAPV5HEADER，可以看见结构里有 RedMask，GreenMask 和 BlueMask 的参数）。

创建一个可以对图像数据读写方便的 DIB 位图可以使用 CreateDIBSection。把 BITMAPINFOHEADER 填好传进去，就可以得到一个 DIB 位图，并可以得到指向图片数据区的数据指针，我们可以通过直接对数据指针的读写来操作位图上的图像。但是，这个 DIB 位图是不可以直接画到窗口 DC 上的，也不能选入窗口 DC。正确的做法是，先用 CreateCompatibleDC 为窗口 DC 创建出一个兼容的内存 DC，然后把这个 DIB 位图选入内存 DC。使用 BitBlt 把内存 DC 绘制到窗口 DC 上。

下面是创建一个 100×100×32bit 的 DIB 位图并把它画到窗口 hDC 的（0,0）处的代码：

```
BITMAPINFO bi;
BITMAPINFOHEADER &bh=bi.bmiHeader;

bh.biSize=sizeof(bh);
bh.biWidth=100;
bh.biHeight=-100;
bh.biPlanes=1;
bh.biBitCount=32;
bh.biCompression=BI_RGB;
bh.biSizeImage=0;
bh.biXPelsPerMeter=0;
bh.biYPelsPerMeter=0;
bh.biClrUsed=0;
bh.biClrImportant=0;

HDC window_dc=GetDC(hWnd);
HDC memory_dc=CreateCompatibleDC(window_dc);

void *buffer;
HBITMAP bitmap=CreateDIBSection(0,&bi,DIB_RGB_COLORS,
                                &buffer,NULL,NULL);
HGDIOBJ oldobject=SelectObject(memory_dc,bitmap);
// 到这里，如果以上函数调用均成功，buffer 指向的内存将映射为这个
//100x100x32bit 的位图
BitBlt(window_dc,0,0,100,100,memory_dc,0,0,SRCCOPY);
// 把内存 DC 传输到窗口 DC 上

// 在使用完毕后，记得释放掉所有的资源
SelectObject(memory_dc,oldobject);
DeleteObject(bitmap);
DeleteDC(memory_dc);
ReleaseDC(hWnd,window_dc);
```

如果你觉得这个过程比较繁琐，或者你希望位图的数据 buffer 由自己分配出来，其实也可以不创建内存兼容 DC 和 DIB 位图。有更简单的方法，就是使用 SetDIBitsTo Device。

还是上面这个例子，如果我们把一个 100×100×32bit 的图片数据预先储存在一个 DWORD 数组里，那么实现同样功能的代码就可以简化成如下：

```
BITMAPINFO bi;
BITMAPINFOHEADER &bh=bi.bmiHeader;

//对 bi 的初始化过程同上
// ...

DWORD buffer[100*100];

// 对 buffer 数组填入图像数据
// ...

HDC window_dc=GetDC(hWnd);
SetDIBitsTo Device(window_dc,0,0,0,100,0,0,0,100, buffer,&bi,0);
ReleaseDC(hWnd,window_dc);
```

关于 SetDIBitsToDevice 参数的具体描述，请查阅 MSDN。

到这里，已经结束了对 Windows 的 GDI 系统的介绍，虽然讲得很不全面，但是已经足够我们制作 Windows 游戏使用，或许在实际编程中还需要了解更多，我相信能有更多的专业书籍和资料供 Windows 游戏程序员查阅。

5.2 控制游戏的速度

PC 不同于控制台（console）游戏机，每台 PC 的性能都不相同；游戏不同于应用软件，并非运行得越快越好；所以，我们需要为 PC 游戏控制速度。

游戏通常是以每秒画面刷新多少帧来决定游戏速度的，通常我们就以帧数来标识游戏中的时间。经典的 PC 游戏《暗黑破坏神 II》就是工作在 25fps 下，即每秒刷新画面 25 次。

这样，我们都可以用帧数字来决定游戏中需要以时间为参考量的游戏逻辑。以画面刷新多少次来换算成时间是一种简洁有效的方案，因为游戏本身都是以刷新完一次画面为一个循环的终结，我们也很容易控制游戏中各个对象的互动。如果不限制帧数，让游戏程序在客户机上以最

快速度运行，则游戏中的各个对象的互动关系不会出错，但速度特别快的机器上的用户恐怕很难接受。

Windows下典型的让游戏稳定在一个固定的帧速率的方法如下：

```
constint time_per_frame=1秒的时间单位/fps;
int time=当前时刻;

while(游戏没有退出){
查询是否有新的Windows消息;
if(有新的消息){
    处理消息;
}
else{
    int t=当前时刻-time;
    if(t>time_per_frame){
        time=当前时刻-(t-time_per_frame);
        更新一帧的游戏逻辑;
        刷新画面;
    }
    else{
        Sleep(0);
    }
}
}
```

从这个框架可以看出，由于Windows程序本身需要循环分发消息，程序往往被写成循环结构，每更新一帧，游戏逻辑被放到了一个循环中。这很大地束缚了游戏程序员的思维模式。不过，如果游戏中的脚本支持coroutine（比如lua），把游戏的逻辑放入脚本中去解释，就可以把更新一帧游戏逻辑写成触发一下脚本中的coroutine切换了。关于coroutine的描述，可以参考后面关于脚本的章节中的描述。

这只是一个简单的框架，我们以最高的频率检测和处理Windows消息，这样，即使游戏被限制的帧率不高时，游戏窗口也不会很显得迟钝。而没有消息需要处理时，则以time_per_frame中的时间单位为间隔处理游戏的逻辑。我们看到，当CPU可以处理游戏逻辑的能力超过我们的需要时，调用了Sleep（0）。

这个Windows API将使当前的线程放出自己的控制权，这一点对Windows平台很重要，可以使游戏程序不至于占用全部的CPU（如果不这样，就使游戏程序几乎独占了CPU，每次切换进游戏的线程，都把可以用的时间片用完了）。

这个简单的流程还可以完善一些，比如当检测到游戏的速度总是跟不上设想的帧速率时，可以省略刷新画面这个环节，但是，游戏逻辑正

常运作。这样用户感到的只是游戏画面跳跃了一下。

有时候，某帧游戏逻辑的运行时间过长，比如进行了大量的阻塞式I/O操作，应该对time进行复位，以防止接下来的游戏运行为了赶上计时器的进度而看起来过快。

5.3 浅谈MFC

我所见到的程序员中，很多把C++/VC/MFC一锅端，这或许造成了MFC在某一类高手群体中的坏名声。而另一方面，在许多初学编程的人群中，又把MFC看得过于神圣，“会用”MFC，“会用”VC的人变成了高人。还有一些程序员，耐心地学完了C甚至C++，但是却一直不屑于使用MFC写程序，认为微软的那些东西都是垃圾。MFC的结构复杂度让他们不肯像学习C++语言那样细心地学习。

MFC既不是天使，也不是恶魔。MFC是一个框架类库，它所做的不仅仅是封装了Windows API而已，而是引导程序员去使用一些设计模式来编写Windows程序。MFC由于历史原因，携带了许多不那么现代感的C++代码，但是它依旧是一个成功的、完全基于C++语言的框架类库。其设计总的来说是合理的，规模也在容易让程序员接受的范围内，没有仔细研究过MFC的程序员没有资格评论MFC的好坏。

我也同样相信，绝大多数指责MFC的程序员没有能力完成和MFC相当能力的类库。无论MFC的优劣，在Windows下用标准C++写带有Windows GUI的软件，除了MFC还真没有太多的选择（或许WTL算一个）。

5.4 小结

Windows系统博大精深，虽然由于照顾世界上如此众多的硬件，以及在Windows 20多年历史中，为了保持向前兼容性带来的巨大历史负

担，使得某些设计不那么优雅，但是，瑕不掩瑜，Windows 仍旧是一个伟大的操作系统。想充分理解这个操作系统怎样工作是需要花很长一段时间，需要多一些编程经验的。

游戏，作为比较独立的应用，可以算是不太需要深入地和 Windows 系统打交道，如果写一些专业的 Windows 应用软件，从 Windows API 开始搭建起来则非常容易混乱，成熟的 Windows 应用程序框架是值得推荐的，比如 MFC。这些跟游戏编程关系不大，就不再专门分列一章介绍了。把这本书变成一本 Windows 下游戏编程的指导教程并非我的原意。

希望本章能为对 Windows 环境编程有恐惧感的人消除心理上的障碍，为对 Windows 编程知其然不知其所以然的游戏程序员建立一个正确的 Windows 编程观念，这才是我写作本章的真正意图。

第6章

汇编优化

汇编优化，是在高性能游戏引擎设计中很难避免的一个环节。虽然不用汇编去做优化依然可以写出高性能的程序，但优化却可以变得更好。

但是，这一章并非从汇编开始谈起。相反，一开始我就花了不少笔墨谈汇编之外的知识。即使是一些技巧性的细节，也并非必须用汇编才可以做到。

例如，消除上下文相关性、消除除法、回避静态变量、优化数据组织方式，这些是可以在 C 语言层面进行的 。

为了更好地使用汇编，这一章着重介绍了在 C/C++ 中和汇编进行混合编程的方法和技巧，然后才具体谈到汇编指令的高效使用策略。

本着追本溯源的精神，在本章的最后，会回顾 Pentium 到 Pentium 4 各代 Intel 处理器的流水线运行规则和优化策略。了解这些比了解汇编指令的用法更有价值。

另外，本章还介绍了 Pentium 以后的 x86 处理器提供的一个重要性能评测工具，RDTSC 指令。这在现代 PC 游戏中用途广泛，是一个比时钟更精确的记时工具。

热衷于用汇编来解决程序中的瓶颈部分并非我从编程之初就养成的习惯。对与程序员来说，用C++去考虑软件整体的构架和用汇编书写一个最底层函数的实现，是两种截然不同的思考方式。并非所有人都喜欢这样在宏观和微观之间的思维切换。

在我认识的好朋友里，优秀的程序员并不少见。而大多情况是，他们要么沉迷于把玩代码的细节，因为几行汇编代码的调整改善了程序的性能而欢呼雀跃；要么对coding的工作不屑一顾，只要能让人代劳的话，绝对不想干编码的活。

这两种行为都可以用一个理由来解释：现代的编译器在代码优化方面已经发展得非常完善，可以将C++代码编译成足够优化的汇编。如果只是一些机械性的优化方法，理论上说，机器不可能比人脑做得差。第一种人在用自己的大脑挑战机器，从自己的胜利中获得一种成就感；而第二种人觉得代码的细节是一种纯粹的体力活，不想浪费自己的脑力和时间。我是介乎两者之间的人，本质上说偏向前一种，任何有趣的事情都值得去付出努力。

和设计软件的框架结构一样，在汇编一级优化代码也是一件极其有趣的事情，为我启发这个兴趣的是Michael Abrash的那本《图形程序开发人员指南》。

1998年，那本书的中文版出版了以后，我迫不及待地买了一本。躺在大学寝室的床上用了3天把它读完。如果你见过这本书的厚度，该明白我当年如何沉迷于那本书的文字中。在我的记忆里，计算机方面的技术书籍，只有后来的《C++编程思想》和《C++的发展与演化》两本让我同样得如饥似渴。

那本书在完成时，很大部分已经是对历史的一种记录，大多数技术点都随着硬件的发展逐步变得没有实用意义。到了7年后的今天，书中记录的大部分代码和技巧不再可以直接抄进我们的程序。但是，在我为了写这本书而再次从旧书堆里翻出这本大部头寻求一些参考材料时，我又情不自禁地品味起作者独特的编程哲学。

或许是我和Michael Abrash在某些方面有一些共同点，他的语言总能引起一些共鸣。不过，无论是人生经历还是编程历史，他都远比我丰

富，有着更多的沉淀。所以，我在现在这个阶段，始终无法用精练的语句向读者表达我对编程，对优化的理解，这恐怕就是和大师的差距吧。对代码优化有兴趣的朋友，建议精读此书。

2004年春，我在GDC 2004上有幸见到了Michael Abrash本人，他在会上做了一个关于优化的演讲。讲演的题目是关于他的新作之一，Pixomatic这个软件3D engine。这套engine中，代码发挥了CPU的极限速度。那种对x86汇编登峰造极的运用能力，之前我从来没有见过。一个人可以如此娴熟地将各种汇编指令以最佳的方式组合到一起，完成非常复杂的功能（我指的是诸如支持各种复杂像素变换的3D运算这样的操作），实在是需要多年的经验积累。

这些也验证了一个道理：人脑在某些方面的思维能力是远远高于电脑的。按照指定的方法和逻辑，电脑的处理能力有限，在相对复杂的算法实现上，人的创造性思维能够组合出更好的机器代码。

这一章，将领略一下在x86下优化代码的乐趣。在阅读这些之前，希望读者明白一个道理：CPU的设计不是一成不变的，所有的优化策略和原则都可能在有一天产生翻天覆地的变化，以前相对快的方法变得相对慢，相对慢的方法变快。在编码生涯中，我们需要不断地学习新的知识，以适应新的硬件环境。好在大原则较小原则变化得慢，不用过于担心知识过期，对于已经写好的代码，如非必要，也不必老是从头翻新。因为一旦CPU有了革命性的变化，也意味着其绝对速度有了巨大的提高，作为游戏程序员，考虑以前代码的效率问题大多是不必要的。

作为处理器一级的优化技术，涵盖面非常广泛。这里，我们主要讨论Pentium以后的Intel x86系列处理器，并将目标放在32位保护模式下的整数运算指令集（下一章还会讨论Intel后期加上的MMX指令集）。对于基本的浮点运算集，以及后来扩展的SSE系列虽然对于现今的3D引擎的设计，也是性能改善的重点，不过限于篇幅，在本书中不具体讨论，感兴趣的读者可以从Intel的官方网站获取更为详尽的资料。这里主要希望能提供给读者一些对于处理器级别的优化策略和思想。

除了官方资料，在互联网上也可以得到一份由Agner Fog, Ph.D.维护的关于Pentium系列CPU上代码优化方法的文章，大家可以去他的个人主页：http://www.agner.org/assem阅读。本章很多的例子也是摘录于那篇文章。

6.1 浅谈代码优化

为什么同样功能的代码会拥有不同的性能？这是因为在代码的不同层面，我们都可以有不同的模块组合方式，大到设计模式，小到一个局部函数的语句实现。在了解了完成事情的方法后，程序员做的就是怎么组合一些东西，让他们联合工作的效率最高。这就好像搭积木一样，各种不同形状的积木可以搭配得紧凑，也可以松松散散。

如果不同形状的积木过多，想把它们组合得很完美，就需要事先做整体上的全局考虑。搭积木是一个模式匹配的过程，工程师同时考虑的积木数量越多，对它的大脑的思考记忆能力要求就越高。对于编码来说，把各个模块平均运行次数与模块平均运行时间的卷积做得尽量小是优化的目标，这在数学上可是个复杂的工作。

这个过程随着模块个数的增加成天文数字的增长，不仅仅让最优化变得困难，连一大堆积木堆起来保证不塌掉都成了问题。这显然说明了，模块分得过细不利于软件开发，设计师往往寻求简单的途径将更小的积木先组合起来，做成更大的块再堆起来，这样条理清晰，不容易出错。

在大多数层面上，我们用于估算效率的数学工具仅仅限于时间复杂程度，推算出一个方法的时间消耗跟处理的数据量是线形关系或是指数关系，或是对数关系。但随着描述问题的层面逐渐接近底层，这种数学方法越来越不能反映不同代码间真正的效率关系。

因为每条机器指令的速度差异、指令组合的速度差异已经和单条指令本身的运行时间放在了一个数量级上；而且内存访问速度、计算机各级缓存间的数据交换时间也变得不可忽略（甚至长于大多机器指令的运行时间）；另外，高级语言的单条语句对应于编译后的机器指令的数量也存在巨大差异，对于最贴近机器模型的 C 语言已经如此，更不用说其他高级语言了。

如果有一个函数，其实现不调用任何复杂的不可预知的系统 API，也不驱动任何复杂的硬件设备，而是完全由 CPU 和内存做数据交换，那么，我们就可以相对容易地得到它的速度精确计量值。这样的函数就是

由一些确定的机器语言代码解决了某个特定的问题。想让它运行得足够快，就是一个用机器语言指令集解决特定问题的最优化模式匹配问题。如果理论上，让编译器从代码中了解到足够多的信息，使它能从足够多的信息中明白编程者想达到的目标，那么就一定能得到最优的结果(C++就是从这方面，利用 template 等特性实现了比 C 更高的编译质量)。

遗憾的是，事与愿违，这是一个 NP 难度问题，计算机的速度没有质的提高（比如量子计算机的出现）前，是无法与人的大脑相比的。只不过这有一个前提，编码的人至少要了解足够的相关知识，而且需要有足够的经验和清晰的头脑，才有可能不比机器做得糟糕。

6.2　并不仅仅是汇编

在文字中充满汇编语句前，有必要先花一小节来说明，并非所有的处理器级的优化策略仅限于汇编。C 这样的高级语言中，不少规则依然有效。之所以编译器不会帮你做这些优化，是因为这些优化规则大多需要改变代码局部的逻辑（虽然不影响整体实现的逻辑），或是需要改变数据的组织方式，编译器是没有权利做这些的。

6.2.1　在算法实现时减少上下文的依赖关系

看如下程序:

```
double list[100];
double sum = 0;
for (int i = 0; i < 100; i++)
     sum += list[i];
```

这里，我们对 100 个 double 数求和。这段代码有优化的余地吗?从 C 语言的层面上讲，可以说没有；但是在处理器的层面，可以做如下优化：

```
double list[100];
double sum1=0, sum2=0;
for (int i=0; i < 100; i+=2) {
```

```
        sum1 += list[i];
        sum2 += list[i+1];
    }
    sum=sum1+sum2;
```

注意：这里初看是做了一次循环展开。关于循环展开是一个古老的优化技术，正如这里把 100 次循环变成了 50 次，减少了条件跳转的次数。

但是，现代 CPU 由于具有分支预测技术，条件跳转已经工作得足够快，循环展开增加了代码量，更多地占用了 CPU 内部一级代码高速缓存的空间，循环展开远远不如原来有效，反而是代码书写变得不那么优美了，我并不向读者推荐。

而这里，更重要的是，我们希望累加的是 100 个浮点数。每一次加法都依赖上一次加法运算的结果。这是一个长长的依赖关系，处理器只能线性地处理。现代处理器在微操作的级别，在不影响逻辑的情况下，早已经允许将临近的多条指令同时处理。在后一部分的优化代码中，我们将一个长长的倚赖链分化为两个。这样，处理器就有机会把两次加法同时完成了。

其实，这个问题的关键还在于处理的数据是浮点数而不是整数。这里，浮点的加法需要消耗 5 个时钟周期（现在大多数的 CPU 下），如果换成整数，加法指令可以和后面的循环变量的递增以及条件跳转指令并发，而使用浮点数这种优化效果就不再存在。

这个例子也从侧面说明了，整数运算的效率比浮点数高。虽然现代处理器处理浮点数已经足够快了，但是，整数运算更快。所以，能用整数运算解决的问题，应避免使用浮点数。

6.2.2 低效的静态变量

对于 C 语言模型不了解的人，或许会认为局部变量使用的内存在用到的时候才被分配出来，而静态变量是一开始就分配好的，所以静态变量会有更高的运行效率，这是一个天大的误解。局部变量存在于堆栈上，

对其空间的分配，仅仅只是在一个变量声明的语句段中，仅依靠单次的修改 esp 寄存器就可以实现的（一组局部变量的声明只用做一次这种操作，而不是每个变量一次）。

把变量放在堆栈上，带来的最大好处是：函数能重复使用内存。当前的函数退出了，内存空间被回收，进入新的函数中时，局部变量又重新使用了相同地址的内存。当同一块内存被反复读写，其数据就会存在于 CPU 的内部一级缓存中，访问速度非常快。绝大多数情况下，堆栈顶部的数据就符合这个条件，而零散的静态变量则没有这个优势。内存同 CPU 内部缓存的数据交换，往往成为程序的速度瓶颈。

同样的道理，临时构造一个对象，使用系统提供的 new 操作间接地调用了 malloc 是一种低效操作;注重效率的做法是写一个 placement new，而改用 alloca 在堆栈中分配对象需要的空间，然后在退出函数时直接调用对象的析构函数，而不是 delete 方法，让函数退出时的堆栈调整操作自动回收内存。

前面提到“绝大多数情况”，也就是说存在特殊情况有人喜欢局部声明巨大的数组，这既容易造成栈溢出，又使得 CPU 内部缓存的映射的内存地址不停地变动，极大破坏了 CPU 的高速缓存带来的好处。所以应尽量避免这样情况。

如果你有一组全局变量，有时候把它们聚合在一起放在一个类里面，通过成员函数来访问也是一个提升效率的方法。另一方面，对于 C++来说，这样做也符合设计模式中单件（singleton）的实现。对于运行效率来讲，把数据全部通过唯一的一个 this 指针来访问对 CPU 的缓存的有效工作有利。

这种方法也不是没有缺陷的，我们必须牺牲掉有限的寄存器中的一个来存放 this 指针。

到了 Pentium 4，访问一个 32 位地址变得比用寄存器加偏移地址的形式读写内存更加低效了，具体原因在下面关于 Pentium 4 的部分会具体解释。总而言之，无论从设计角度还是效率角度，避免使用静态变量吧。

6.2.3 数据的组织

如果几个数据是相关的，处理其中的一个，几乎在接下来的同时就

会去处理另外几个的话，那么，让这些数据物理上挨在一起是增进效率的。

最简单的例子是，在用C语言写程序时，如果希望储存一组2维坐标，有人选择定义两个数组，x[]和y[]，而有人选择先定义一个结构{x，y}，再定义这个结构的数组。由于坐标经常是同时需要x，y两个数据的，显然第2种方案更能充分利用CPU的数据缓存。

CPU在处理数据的时候，最希望数据以对齐的方式存在于内存中，大部分变量，无论是一字节的字符类型，还是双字节的short类型，或是4字节的指针都希望以4字节方式对齐（即内存地址为4的位数的地方）。另外一些比较长的类型，如double，长度是8字节，以8字节对齐则最有效率。对于一个比较大的结构，则希望以16字节对齐。

顺便提及，大多数编译器，像VC，在这点上做得还不错。VC中，在工程的设置里可以找到结构对齐的设定，默认是8字节对齐的，如果你愿意，还可以调整到16。

6.2.4 消除除法

无论对于整数运算还是浮点运算，除法都是一条很慢的指令。正因为这条语句的实现很复杂，一些RISC芯片甚至都不提供除法指令，改由库函数的形式给出。所以我们应该极力避免或减少除法运算。

对于整数运算，除2的整数幂可以被优化成移位操作，但是我们并不主张把n/8写成n>>3，因为现代编译器会帮你做这件事情。

这里需要注意的是，如果你知道n肯定是一个正数，而n又被声明成int时，写成（unsigned）n/8是有利于编译器帮你优化的。因为对于一个有符号的数字除以8并不完全等价于右移三位。

实际上，即使除数不是2的整数幂，编译器的优化依然会消除除法，变成一次乘法加一次移位操作。让编译器有优化余地的唯一前提就是，让除数是一个常数，除了直接写出除数，也可以用const int来定义。

对于浮点运算，我们有更多的方法减少除法的运算次数（以下方法在特定条件下，使用一些技巧也同样适用于整数运算）：

- 对于常数除法，如果需要对一个数字反复做除法，则可以先算出其倒数，然后用乘法替代。如果是常数除法，比如a/3.14倒不必要写成a*（1.0/3.14），因为编译器通常会帮你完成这种优化。
- 在某些情况下，可以用乘法消除掉除法运算，比如if（a>b/c）可以写成if（a*c>b）。编译器之所以不会帮你做这样的优化，是因为两者并非完全等价（如果 c 是负数，后者的要写成<;当这个方法用于整数的时候，要考虑a*c有溢出的可能）。
- 我们还可以想办法把多个除法合并起来。对于运算 a/b+c/d，（a*d+c*b）/（b*d）会更快一些，这是因为即使是算3次乘法，也快过算一次除法。

这种数学上的四则运算变形，用更多次的乘法来消除除法甚至可以推广到不相干的多个表达式上。看下面这个例子：

```
void f(double a1, double a2, double b1, double b2,
       double &y1, double &y2)
{
    y1=a1/b1;
    y2=a2/b2;
}
```

这个函数可以做如下优化：

```
void f(double a1, double a2, double b1, double b2,
       double &y1, double &y2)
{
    double temp=1.0/(b1*b2);
    y1=a1*b2*temp;
    y2=a2*b1*temp;
}
```

这种方法甚至可以推广到 4 次不相干的除法：

```
y1=a1/b1;
y2=a2/b2;
y3=a3/b3;
y4=a4/b4;
```

可以改写成：

```
b12=b1*b2;  b34=b3*b4;
temp=1.0/(b12*b34);
y1=a1*b2*b34*temp;
y2=a2*b1*b34*temp;
y3=a3*b4*b12*temp;
y4=a4*b3*b12*temp;
```

不过，再好的方法也不能滥用，如果想合并超过 4 次除法就有相反的效果了。因为消除除法的代价是增加了更多数量的乘法，太多次数的乘法消耗的时间足以超过省下来的除法运算了。

6.2.5 避免过大的循环

程序 1：

```
for (int i=0;i<100;i++) {
    do_something_1();
    do_something_2();
}
```

程序 2:

```
for (int i=0;i<100;i++) {
    do_something_1();
}

for (int i=0;i<100;i++) {
    do_something_2();
}
```

如果 do_something_1()和 do_something_2()这两个函数没有关联，则程序 1 和程序 2 的运行结果相同。哪个程序更好?

或许有很多讲究效率的程序员喜欢第一种，很多人喜欢在一个 for 循环中干太多的事情，或许他们想当然地以为，for 循环累加 i 也是一种时间消耗，如果能减少对 i 的操作也是一种优化。但实际上事与愿违，两段程序的性能比较取决于 do_something_1 和 do_something_2 的具体实现。

如果两个函数都实现得很简单，则通常程序 1 会比较有效率。而如

果两个函数都干了较为复杂的事情，那么程序2会变得比程序1高效。

导致这个结果的是CPU的代码cache机制。通常，程序代码被第一次运行的时候，速度会比较慢（比如for循环的第一次进入）。这存在一个代码从内存加载的时间，代码用到的内存数据的加载时间、还有分支预测缓存的建立，甚至还有CPU对指令的解码时间（对于现代CPU，为了兼容以前的指令集且保持高效率，在内部需要使用一套高效的微操作指令的同时，需要做从外部的指令编码到内部微指令的转换工作）。

如果循环体内涉及到的代码超过了CPU的代码缓存的容量（通常是8K，到P4加到了16K），就使得循环中代码每次的进入都要重新做一系列缓慢的操作，如果我们能尽量维持循环内的代码在一个比较小的尺寸，就能提高效率。

如果do_something_1()和do_something_2()干的事情逻辑上联系不大，仅仅是想把两个函数各自循环调用若干次，则从软件设计角度讲，程序1也是个糟糕的设计。

书写高效的C++代码几乎是所有C++程序员的期望，对于一个有经验的C++程序员来说，他可能知道何时该使用inline修饰函数定义（对于编译器熟悉者还会考虑诸如__fastcall的调用方式等），知道应该怎样避免重复的数据复制，知道应该自己实现operator new，或者为STL的容器做一个更适合的分配器来改善内存分配的速度。

对于使用C++去构建大型软件的设计师，同样会根据自己的经验决定对象体系中的粒度问题，一方面，分得过细，带来承重的函数调用负担，分得太粗，破坏了良好的设计；另一方面，注重效率的C++程序员，甚至还会斟酌使用面向对象的设计方式带来的虚函数间接调用消耗，以及采用多重继承、异常这些C++高级特性的额外消耗。

但在这里，举出的却是一些处理器相关的代码效率问题，往往被纯粹的C++程序员忽略。作为注重效率的程序员，即使你这辈子都不打算学汇编，也应该对影响效率的方方面面有所了解，即使不写汇编，也可以在C++的层面去改善。

6.3 汇编和C/C++的混合编程

如今并不主张用汇编来制作整个软件，通晓工作平台的汇编语言的目的通常有两个：提升调试的技能，使用汇编来改善核心代码的效率。

6.3.1 使用外置的汇编编译器

使用汇编书写核心代码，传统的方法是使用一个汇编编译器，编译出来的汇编代码可以和C++编译器产生的代码一起连接。

Windows平台上最普及的汇编编译器是MASM，由于和VC同为微软公司的产品，所以搭配得最好。如果选用gcc，则由于gcc产生的编译目标文件内部使用elf格式，而不是MASM和VC采用的COFF格式，这样我们就不能采用MASM做汇编编译器和gcc搭配使用。

gcc下的汇编需要兼容更多的CPU，并没有使用更符合x86汇编程序员习惯的汇编语法，而是使用略显怪异的AT&T语言（从设计面讲，AT&T汇编语法其实定义得更好），初次使用gcc的汇编编译器的程序员可能会很不习惯。为了让程序可以跨平台，跨编译器，我推荐使用NASM——一个不错的开源自由软件。语法更接近习惯，而且有相当强大的宏功能，对于最新的指令集也支持得非常好。

使用额外的汇编编译器，在使用手工书写的makefile进行工程管理时是很容易的，而对于使用VC的集成环境就稍显麻烦。好在集成开发环境对于不能识别的源文件类型，都预留了叫做自定义创建（Custom Build）的选项，在里面填上启动编译器的命令行指令即可。

C++和汇编混合编程时最麻烦的就是给汇编函数命名。在现存的连接模型中，都是以函数名——一个字符串，来定位函数的实现代码的。从C++开始，支持了函数重载，为了用唯一的字符串表示带有不同参数，存在于不同名字空间或类空间下的同名函数，采用了名字编码（name mangling）的技术。

这种经过编码的函数名很难用人工书写。通常我们可以先用C++编

译器生成一个需要的空函数体，从生成的目标代码中找到编码过的名字复制过来用。但这不是值得推荐的做法。因为C++标准对名字编码并没有统一标准，每个编译器都有自己的编码方式，这样做，汇编程序被限定在了某个固定的编译器中。

为了解决这个问题，一般汇编里书写的函数通常遵照C标准，即直接在函数名前加一个下划线，然后在头文件中为函数原型声明写上extern"C"。

如果还希望函数原型声明在纯C的环境下使用，不妨在头文件中判断一下C++编译器通常会预定义的宏__cplusplus，以决定是否加上extern"C"。如果需要为成员函数书写汇编版本的时候，由于不可能让成员函数遵从C的命名规则，我的建议是写一个C的全局函数，把this指针作为第一个参数传入，然后在类声明中把这个函数声明为友元，写一个嵌入版本的成员函数调用一下它，例如：

```
class x;

extern "C" void do_something_asm(x *self);

class x {
friend void do_something_asm(x *);
// …
void do_something() {
    do_something_asm(this);
  }
};
```

这里，我们想用汇编来实现void x::do_something()，但是并没有直接实现，而是使用了一个叫做do_something_asm的函数绕过成员函数的命名编码问题，由于现代C++编译器都可以很好地优化内嵌函数的调用，所以性能并没有损失。

使用外部汇编编译器另一个亟待解决的问题就是访问类的成员变量，汇编不能直接导入C++的头文件，所以一般我们会在汇编中创建一个结构来等价描述类的构成。对于汇编语句来说，它需要知晓的其实只是某个变量在结构中的偏移量。原生类型变量占用几个字节是很容易推

算的，但是像 STL 的库，其中，一些类型的大小却并非那么明显，而且各种编译器的 STL 实现也可能不同。所以我们不主张用汇编去访问这些并非自己写出来的结构或类，如果类中要包含 STL 中的对象，也尽量定义在靠后一些。

C++的类实现的内存布局往往不像单纯的结构那么明显，我们能肯定的是，单继承下，如果类本身和它的基类都没有用到任何的虚函数，则这个类的内存布局就和传统的 C 结构相同，只不过基类和派生类顺序连接在一起。如果有虚函数，那么一般类对象存在于内存中，前四个字节会是一个虚表指针，然后才是数据。

多重继承实现比较复杂，各个编译器也不一定完全相同，不主张在和汇编混合编程时使用。搞清楚 C++类对象的内存布局对于 C++程序员是一件有趣且有意义的事情，这不仅仅可以方便我们用汇编混合编程，也对增进调试技能大有好处。这方面，Lippman 写的 *Inside C++ Object Model* 是一本必读好书。

6.3.2 使用内嵌汇编

相对于外置的汇编编译器，使用内嵌汇编是更为方便快捷的方案。现在，几乎所有主流 C++编译器都支持在源代码中插入汇编，只是可惜每种编译器的语法标准并不统一，甚至 gcc 还使用让多数 Windows 平台汇编程序员不太习惯的 AT&T 语法规则。所以，使用内嵌汇编的代价就是源码变得和编译器相关，不利于移植。

这里只讨论 VC 下的内嵌汇编。

使用内嵌汇编就是说不需要再用汇编书写函数定义、结构定义这样的东西，甚至也不需要管函数的参数如何传递以及怎样返回值，要做的仅仅是在需要的地方用__asm 这个关键字插入汇编代码，在汇编中，可以直接使用局部变量和全局变量。使用单行汇编可以直接在__asm 后面跟上汇编语句，例如：

```
int temp=0;
__asm inc temp
```

这一句可以将temp这个临时变量累加一。__asm后直到当行回车结束，全部会被认为是汇编语句来编译，所以这里不需要用C语言的标准（分号）来结束这一行。

庆幸的是，按汇编语言的标准，分号是用来注释到行末的，所以在行末加上一个分号也不会引起编译错误。如果需要写多行汇编代码，一种方法是在每行之前都加上__asm；另外，VC也支持__asm{多行汇编语句}这样的形式（下面的例子中可以看到这种写法）。

在C函数中，无论是函数传入参数，还是局部变量，默认都是用ebp做偏移量基址来读写的，这或许是因为esp可能因为函数调用前的参数压栈操作而改变，直接用esp做偏移量的基址比较麻烦（反汇编阅读编译器编译出来的C函数，通常会在函数入口附近发现mov ebp,esp这样的汇编语句），所以，上面这句__asm inc temp实际会被编译为inc [ebp+xxx]的形式。正因为这一点，我们应该尽量避免在内嵌汇编语句中直接使用ebp这个寄存器。

虽然别的寄存器是可以随意使用的，但是出于效率，应该优先使用eax，ecx和edx。因为其他ebx，esi，edi这几个寄存器往往会用于保留编译后的C++语句上下文状态。如果你的内嵌汇编使用这些寄存器，编译器将自动产生一些代码，先把这些需要保护住的寄存器保留下来，在内嵌汇编代码之后恢复。

如果想访问类的成员变量，则应该先把类指针读到一个寄存器中，然后用偏移量来访问，相比较外置的汇编编译器，这个偏移量不需要人工计算，非常方便。

我们看下面一个例子:

```
class x {
int _member;
void inc_member() {
  __asm {
      mov ecx, this
      inc [ecx]x._member
  }
}
};
```

这个例子演示了 void x::inc_member()这个函数中的内嵌汇编如何累加成员变量_member 的。

第一步，它把 this 指针放入 ecx，这里 this 可以看成函数的一个隐式传入的参数。

第二步，以 ecx 为基地址，x._member 为偏移量累加。x._member 就是_member 变量在 class x 中的偏移。

6.3.3 C++的函数调用规则

C++的函数被调用的时候，参数是通过堆栈传递的，参数传递必须遵守一定的规则来使用堆栈。C 和 C++的缺省调用规则叫做__cdecl，这个关键字也可以被显式地加在函数原型声明中。对于遵守这个调用规则的函数，如果有多个参数，则参数是从右向左压入栈的。由于压栈操作会使 esp 递减，所以对于函数的实现体来说，第一个参数总在内存地址的低处，和进入点 esp 指针最接近的地方。而当函数返回后，曾经压入堆栈的参数数据并没有清除掉，需要调用者加上一句 add esp, xxx 退栈。

之所以这样设计，是因为 C 语言需要支持可变参数，我们用得最多的就是 printf 中的函数原型声明。这样的函数，必须从第一个输入参数开始处理，让第一个参数的位置可以通过 esp 直接得到，在未知的参数个数的情况下，这样处理最为合适。而且，函数本身有多少输入参数，调用者比函数本身更为清楚，所以让函数本身来恢复堆栈指针是不安全的。

不过，C++里已经不再提倡这种不定长参数的函数原型，严格的函数原型声明可以使用更为高效的参数传递规则，那就是__stdcall，被广泛地使用于 DLL 的导出函数。和__cdecl 不同，函数实现体本身需要自己把传入的参数退栈，对于这个操作，x86 汇编一级用一条 ret xxx 的指令就可以完成。

另外还有一种和__stdcall 类似的，但参数传递次序相反的_pascal 规则，参数从前向后压栈，今天我们已很少使用，它的存在仅

仅是历史原因（当年为了实现C和Pascal的混合编程）。

Win32 API中广泛使用__pascal规范。

编译器通常还会实现一种叫做__fastcall的参数传递方法，函数的参数用寄存器传递，这能极大地减少函数调用的消耗，但是具体会使用哪些寄存器和编译器的实现密切相关。如果我们用C/C++实现某些不适合inline却需要提升调用速度的函数时，声明成__fastcall调用规则能改善程序的性能。但如果想和汇编混合编程，__fastcall因其依赖编译器的具体实现而没有一种成文的标准，各种不同的编译器并没有保证用相同的寄存器、相同的方式传递参数，考虑到代码的可移植性以及对更新版本编译器的兼容问题，不推荐使用。

类成员函数的调用规则也依赖编译器的实现，VC中增加了一种__thiscall的参数传递规则，但不能显式指定。在__thiscall规则下，this指针会通过ecx传入函数，如果使用内嵌汇编则可以忽略这一点，因为进入函数后，ecx这个变量同样会被复制到堆栈中，可以使用this标识来访问。

函数调用时，被压入堆栈的参数往往和我们在原型声明中看到的不同，在好几种情况下会有额外的隐式参数入栈。如果成员函数被显式地加上了__cdecl或__stdcall的修饰（如果使用了不定长参数，__cdecl会被强制加上），this指针会被当做第一个参数压入堆栈。又若函数的返回值是一个类实例而不是一个简单类型，则会由调用者在堆栈上临时分配出对象需要占据的空间，并把这个空间的地址作为一个参数传入。

我们在书写可以被C++调用的汇编函数时，应该尽量使用简单的函数原型，避免不同编译器下有相异的参数传递规则的情况。汇编中如果需要调用C/C++函数，也应注意同样的问题。

在Windows下的编译器产生的代码，无论使用哪种参数传递规则，它们还有一点都需要遵循。如果一个参数小于32位，它依然会按一个双字压栈，也就是说，即使你只想传入一个字节的字符类型，被压栈的还是4个字节，而不是一个。如果参数大于32位，那么也会按32位对齐的方式处理，并且低位在前高位在后。

6.3.4 寄存器的使用

Windows 下每个可被 C 调用的函数必须遵循一个原则，就是它必须负责 EBX，ESI，EDI 和 EBP 内数据的安全，换句话说，每个函数都有义务维持这几个寄存器的值调用前和调用后一致，如果你的函数内部想借用这几个寄存器做临时变量，就应该把它们压栈并在结束时出栈。

EAX，ECX，EDX 和 CPU 后来扩展的 MMX 寄存器则可以任意使用和改变（ECX 比较特殊，在 gcc 的纯 C 环境下，同样是不能被改变的，而几乎所有的 x86 下的 C++编译器都允许自由修改 ecx。事实上，大家都使用 ecx 传递 this 指针）。

ESP 的值被期望永远是 4 的倍数，所以，不要把 16 位的数据压栈；而段寄存器在保护模式下有特殊含义，我们也不能主动去修改它们，哪怕只是临时修改，都有可能导致出错，当然实际上，我们也很少在游戏编程中需要用到它们。

状态寄存器中的串处理方向标志位默认是被清零的，所以主动调用 CLD 没有意义。但是，如果你想临时改变，那么在调用别的函数前，为了维持这个规则，你应该清除它（这个规则仅适用于 Windows，在 Linux 等操作系统下，C++的编译器似乎没有规定方向标志位的默认状态）。

函数的返回值通常被放在 eax 中，如果需要返回一个 64 位返回值，则可以把结果放在 edx:eax，这正是允许函数可以随意改变 eax 和 edx 的值的起因。对于浮点数的返回值，结果放在 st(0)中即可。综上所述，简单的函数返回值是不需要利用堆栈操作的。

6.3.5 __declspec(naked)

使用内嵌汇编编写某个函数的实现代码时，很少需要考虑哪些寄存器应该保护起来等到函数结束时恢复，其原因在上文已经提及——编译器生成了相关代码。另外，编译器还会为每个函数生成一些堆栈数据保护以及清除传入参数然后返回的代码。

绝大多数情况下，编译器产生的这些代码也符合我们的需要。但是也存在极少的可能，我们不需要编译器产生这些东西，而仅仅只需要它

为我们描述函数调用的入口。如果使用 VC，则可以用__declspec（naked）修饰一个函数来达到我们需要的效果。例如：

```
__declspec( naked ) __int64 __stdcall sqr(int x)
{
    __asm {
        mov eax, [esp+4]
        mul eax
        ret 4
  }
}
```

这个函数就可以计算 x 的平方，返回一个 64 位整数。我们没有让编译器给我们生成过多的代码，自己处理函数的返回。

__declsped(naked)存在另一个用途，这通常发生在函数需要用很多寄存器中临时变量，多一个寄存器可以使用也好的情况下。平常内嵌汇编中，我们被禁止改变 EBP 这个寄存器，而当你写 naked 汇编函数时就可以考虑直接用 esp 访问堆栈上的数据（或者不需要访问堆栈数据），而把 ebp 腾出来以做它用。

值得一提的是，gcc 到目前为止，似乎没有也不打算支持内嵌汇编函数的 naked 方式，相关的讨论可以用 Google 在 gcc@gcc.gnu.org 这个 maillist 的历史档案中找到。如果你需要写这种形式的函数，只能用外置汇编编译器了。

6.4 指令使用的技巧

6.4.1 LEA：别样的内存寻址指令

在汇编手册的定义上，LEA 是一条内存寻址的指令，只不过它的结果是地址本身而不是地址里的数据。由于 x86 CPU 的寻址方式非常丰富，导致这条指令可以完成比它字面上的意义更广泛的功能。几乎所有的汇编程序员都把 LEA 作为一条超级加法指令使用。

```
LEA 寄存器，[寄存器一+寄存器二*倍率+常数偏移]
```

这是LEA 的标准用法，倍率可以是1，2，4，8 中的任一数字。所以把 eax 加2 放到 ebx 里完全可以写在一行代码中：

```
LEA ebx,[eax+2]
```

想让 eax 自乘9 也可以用一行完成：

```
LEA eax,[eax+eax*8]
```

组合 lea 和移位等语句，可以完成比较复杂的常数乘法，而乘法指令要慢得多，并且不利于流水线工作。

注意：这个技巧一直到 Pentium Pro 都非常有效，但是正如本章开篇提到的，没有一成不变的法则。乘法指令到了 Pentium Pro 以后就开始工作得很快了（但是还是有一些限制），而 lea 到了 Pentium 4 以后，被分解成了加法以及移位的微指令，不太具有速度优势。可另一方面，Pentium 4 上乘法指令又变得相对较慢。

6.4.2 BSWAP：扩展寄存器的个数

编写汇编函数，常常需要使用很多的寄存器做临时变量，而寄存器总是不够用。CPU 提供了一些32 位寄存器，但很多情况下，16 位数字已经能满足程序的需要。那么有什么方法可以把一个 eax 寄存器当成两个16 位寄存器用呢?

遗憾的是，虽然我们可以通过别名单独操作32 位寄存器的低16 位（如 eax 的别名 ax，表示 eax 的低16 位），但目前没有方法直接访问到高16 位。

有什么办法可以解决这个问题呢？我们可以使用BSWAP。

BSWAP 的本意是解决Intel 和Motorola 格式双字数据之间的转换问题的。Intel 把数据的低位放在前面，Motorola 则相反。BSWAP 可以逆转一个 32 位寄存器里的 4 个字节。如果 eax 开始的数据是0x01020304，那么在运行完 bswap eax 之后就变成了 0x04030201。

所以，可以用一个 bswap 切换寄存器的高低 16 位，使程序员觉得好像增加了一倍的寄存器一样。虽然这是一个间接的方法，但是 bswap 语句不用访问内存，这将比把寄存器入栈出栈要好上很多了。

6.4.3 缩短指令的字节数

CPU 为了提高处理指令的速度，把最近最常使用的指令代码加载到 CPU 内部的一级高速缓存内。指令工作于此速度极快，但是，越是高速的缓存，本身的大小就越小。缓存资源极其有限，把代码加载到缓存本身是需要时间的，所以需要同样时间的指令，稍短一点的更有速度上的优势。

缩短指令长度，最容易想到的是使用 16 位立即数而非 32 位的。虽然操作 16 位寄存器的代码，指令码里的立即操作数比操作 32 字节的寄存器代码要短两个字节，但是，在 32 位模式下工作，16 位代码需要增加多余的一个预修饰符号，而且对寄存器的一部分操作往往不如操作整个寄存器有效，所以我们尽量推荐使用纯 32 位的汇编代码。

实际上，有很多缩短指令的长度以提高效率的方法，主要的原则有两点。

1. 使用寄存器而不是数值

（1）mov eax,0 ; 5 字节

这条指令有 5 个字节，其中 0 占据了 4 个。

可以用另一条语句实现相同的功能（但并非物理上等价）。

```
xor eax, eax ; 2 字节
```

或者

```
sub eax, eax ; 2 字节
```

这样，指令就只有 2 个字节了。和 mov eax,0 相比，后面的写法只影响标志位而已。对于实际的目的，把 eax 清零成功达到了。

（2）mov eax,1 ;5 字节

这条指令同样有 5 个字节，但是可以化为逻辑上等价的两条：

```
xor eax, eax ;2 字节
inc eax      ;单字节
```

一共只有 3 个字节。虽然两条指令每条都需要一个时钟周期运行，但是如果巧妙地安排上下文，鉴于 CPU 的多流水线机制，这两条指令一起并非一定要消耗两个时钟周期。

（3）
```
mov ebx,[100000]   ;5 字节
add ebx,[100004]   ;6 字节
```

这两条指令都需要读内存，但是两个内存地址靠得很近。这两条指令一共有 11 个字节。

如果写成这样：

```
mov eax,100000        ;5 字节
mov ebx,[eax]         ;2 字节
add ebx,[eax+4]       ;3 字节
```

一共就只有 10 字节了。这是因为，100000 这个数字只出现了一次，它每出现一次都会占据 4 个字节。而后面的 add ebx,[eax+4]中的+4 是一个很小的数字，用一个字节就可以描述。这种小数字的问题，下面马上就提到了。

2. 把操作数控制在-128 到+127 之间

即使是操作 32 位寄存器，很多指令的操作数如果可以用一个字节表示，那么就不需要动用 4 个字节。这些指令包括跳转、压栈、数学计算，但不包括赋值（mov）。

（1）跳转指令，包括条件跳转和无条件跳转，都支持把参数写成相对跳转的距离，而不是绝对地址。相对距离只可以是在-128 到+127 之内。用一个字节就可以描述。但绝对地址跳转指令需要 4 个字节描述地址。所以不主张代码里产生非常远距离的跳转（比如 C++中把 if 后面的语句块写得过大）。对于条件跳转，实际上不支持绝对地址，如果跳转距离过长，编译器会转换成两条语句。例如：

```
cmp eax, ebx
jz label_2
label_1:
```

```
……
label_2:
……
```

如果 eax 等于 ebx，则跳转到 label_2。否则，程序继续运行到 label_1。

如果 label_1 和 label_2 距离过大，很多编译器会把代码自动编译成如下：

有的汇编编译器不帮你做这种隐式的转换，例如nasm，这样反而可以让我们更精确地手写出最优化的代码。

```
cmp eax, ebx
jnz label_1
jmp label_2
label_1:
...
label_2:
...
```

逻辑没有改变，但是代码长了很多。语句的数量也增加了。

（2）堆栈操作

```
push 200 ;5字节
push 100 ;2字节
```

所以，刚才的 mov eax,1 这个操作，同样可以写成：

```
push 1   ;2字节
pop eax  ;1字节
```

不过，这里涉及到了内存操作（堆栈操作同样是内存操作），效果可能没有

```
xor eax,eax
inc eax
```

好。

（3）数学运算

```
add ebx, 128      ; 6 字节
sub ebx, -128     ; 3 字节
```

这是一个边界情况，加128没有减-128来得短。

不过还有一个更常见的用法：

```
mov eax, -1   ; 5 字节
```

通常会写成：

```
or eax, -1    ; 3 字节
```

3. 寄存器的区别

虽然现在大部分指令 eax，ebx，ecx，edx 都是可以通用的，但是操作 eax 在指令长度上有优势。

例如：

add eax，10000 就比 add ebx，10000 要来得短。

这恐怕是历史原因造成的，可能是因为 x86 的处理器在设计之初，只有 eax 可以用于数学计算，虽然后来每个寄存器都可以做这类事情，但是由于要兼容以前的编码方案，所以处理别的寄存器时需要额外的字节来指定寄存器的名字。或许同样有某些历史原因，导致了用做基地址的变址寻址，esp 就会产生较长的代码，ebp 其次，而别的常规寄存器会更短一些。例如：

mov eax,[ebp+12] 就比 mov eax,[esp+12] 要来得短。

而 mov eax,[ebx+12] 虽然和 mov eax,[ebp+12] 同样长，但是，mov eax,[ebx] 比 mov eax,[ebp] 要短一些。

在这方面，编译器是我们的老师。因为对于最细节的代码编译处理上，设计编译器的大师可以预先设计好一些规则，使用最优秀的指令组合来完成任务。查看编译器生成的汇编代码，往往可以在细微之处受到启发。哦，这条指令原来可以这样去用，如果这样去写汇编代码可以稍微改善性能。在阅读编译器产生的汇编代码时经常发出这种感叹。

注意： 几乎所有的编译器都能生成汇编代码，对于命令行编译器，通常是在命令行上加上一个参数。IDE 环境下，仔细寻找都能找到设置的地方。以 VC 6.0 为例。在每个 cpp 文件的设置（Settings...）中，选择 C/C++ -> Category -> Listing Files -> Listing file type，设定为 Assembly with Source Code，就可以在输出文件夹里看到一个.asm 文件了。

6.4.4 避免跳转指令

关于这个话题，准确地说，应该是避免条件跳转指令。跳转指令会影响CPU运行时的流水线，减慢代码运行速度是可想而知的。

对于无条件跳转，原因大部分来自于代码运行地址的跳转对CPU内部各种高速缓存的运作或多或少的影响。但是大多数情况下，很难避免这些跳转指令。

而对于条件跳转，则危害更大。因为条件跳转的结果受上下文的影响极大，不到最后一刻，CPU不知道下面程序该走向何方，所以一些可以预先做的操作都不能及时操作。为了提高条件分支跳转的效率，CPU的设计引入了分支预测的技术，这项技术解释其工作原理非常复杂，Pentium到Pentium 4各阶段的处理器还不大相同，限于篇幅，这里不展开详解。

但是有一点结论应该知道，分支预测成功时，代码可以保持相对高效的运行，一旦分支预测失败，时间上的损失代价非常大。如果有可能，应该尽量消除这些条件跳转指令，下面列出一些值得借鉴的方法。

1. 求有符号32位整数的绝对值

```
逻辑：if (eax<0) eax=-eax;
cdq       ；扩展eax的符号位到edx中，如果eax是正数，edx为0，
          ；否则edx为0xffffffff
xor eax，edx ；如果eax为负数，就把所有的位取反，否则不变
sub eax，edx ；如果最开始eax为负数，则把这个数字的反加一
```

这段指令之所以可以正常工作，基于这个一个事实：

```
neg eax
```

等价于：

```
not eax
inc eax
```

而，

```
not eax
```

又等价于

```
xor eax, -1。
```

2. 取两个数字中较小的一个

```
逻辑: if (ebx > eax) ebx=eax;
```

无符号整数版本:

```
sub eax, ebx
sbb edx, edx
and edx, eax
add ebx, edx
```

有符号版本:

```
sub eax, ebx
cdq
and edx, eax
add ebx, edx
```

3. 根据条件来选择赋值

```
逻辑: edx=(eax<0) ? ebx : ecx;
cdq
xor ebx, ecx
and edx, ebx
xor edx, ecx
```

除了这些技巧之外，Pentium Pro 之后的 CPU 还加入了 CMOV 指令用于条件赋值，是否应该在代码中使用，取决于程序员是否希望让他的代码工作在老式的 Pentium 或者 Pentium MMX 机上。

6.5 Pentium 的汇编优化策略

把玩汇编代码，选择最好的指令组合，编排各个指令的位置，选择合适的寄存器，这一切就好像汇编程序员的填字游戏一样，可以看成一种趣味智力题。

在x86的设计历史上，最早的8088，各种汇编指令的运行速度参差不齐，程序员可以去选择最佳的指令组合来完成任务；但从8088到80486，CPU朝着速度更快的方向发展，这个过程中，速度的进步导致了速度差别巨大的指令趋向一致，一些延缓代码运行的限制慢慢地消除，使得手工优化汇编代码越来越不合时宜，人工写出的汇编代码逐渐很难超过编译器产生的代码。

转折点从Pentium开始，CPU的设计架构有了极大的突破，CPU变得非常复杂。复杂到光是改变一些指令的次序就会带来速度的提升，编译器优化代码往往力不所及，人脑重新成为更好的优化器。或许有一天，编译器终于可以完成这个复杂的任务，不过在这一天尚未到来之前，喜爱底层代码编写的汇编程序员，玩一玩优化游戏，体验使用汇编编写函数较之C版本速度的飞升带来的成就感，是一件极为有趣的事情。Pentium使优化技术成为一种艺术的回归。

Pentium和以前的CPU不同的是，它并不单纯靠加快时钟频率提高速度，而是提供给程序并行运行代码的可能。Pentium有两条流水线，这样，在同样的工作频率下，CPU能提供1倍以上，2倍以下的性能（实际上，最慢的Pentium的总线速度也比486快得多，所以Pentium比它的前辈快了不只2倍）。为Pentium处理器编写的代码本身是不必要刻意配合两条流水线的工作的，因为Pentium设计出来就是为了在兼容以前x86系列的处理器的基础上工作得更快，而非设计一套新的指令集。不然，另外设计一款RISC处理器会工作得更快且造价便宜。

所以，CPU仅仅在一定不影响正常逻辑的情况下，让指令并发处理。甚至自修改的代码也可以在Pentium上正常工作，只不过会比较慢而已。

Pentium的两条流水线并非对等的，我们把主要的一条流水线叫做U

流水线，它可以处理任何以前 x86 系列的指令，而另一条只能处理部分简单指令，它被称为 V 流水线。两条流水线可以并行工作，有各自独立的读写缓存，这样，处理器就有可能在一个时钟周期运行两条指令。

另一个值得注意的地方是，Pentium 扩大了 CPU 内部的缓存。首先，代码缓存和数据缓存是分开的，所以，读入代码和操作数据互相不影响。数据缓存工作的时候，重复写操作甚至不用绕到外部的总线，所以在堆栈上使用少量的局部变量工作起来并不比读写寄存器慢多少。而代码缓存有一个 128 字节的指令预取队列，这样足以满足短跳转时，指令还在 CPU 的内部。加上分支预测，条件跳转大多数时候也工作得很快，不过，一旦分支预测失败，速度损失也很大。

在古老的介绍优化的书本上，很多前辈提到过循环展开这样的优化策略，也就是说，通过减少循环的次数，而在循环体内复制多次需要做的事务可以减少操作循环计数器的次数，以此来提高一点运行速度。我本人是不赞同这种方案的，即使真的能提高速度，这样做也破坏了程序的美感。到了 Pentium 时代，循环展开更是画蛇添足的败笔，因为它极大地影响了 CPU 内部缓存的工作。

CPU 的处理能力比内存总线的速度要快得多，在处理大批量数据时，读写内存经常变成了瓶颈（这也是现代 CPU 在处理简单的大片内存移动操作时，几乎没有有效的优化方法的原因），CPU 为了提升性能，对于反复操作的内存和代码，在 CPU 内部安排的高速缓存，让程序和数据都常驻 CPU 内部，所以，现代的优化方法，不能一味地追求空间换时间的策略，精简的代码和浓缩的数据很有可能工作得更好，而且，短小精悍的代码也更具有观赏价值。

在我的程序设计哲学里，程序员的水平是和其代码长度成反比的，我们开发的一款几十万人同时在线的大型网络游戏客户端的 C++部分，统计下来不超过 1M 的源码，经过几年的不停维护修改，本来风貌并无太大改变；而当我读到某韩国公司开发的一款新的网络游戏客户端代码时，长达 7M 的 C++源码简直让人头晕目眩，让维护人员整天生活在恐惧之中，每个细微的修改都担心影响其稳定性。

6.5.1 UV 流水线

Pentium 里的指令被分成了四类，第一类指令最受优化爱好者欢迎，它可以工作在任一流水线中，而且任何情况下都允许另一条流水线并行工作。

第二类指令虽然可以在UV 任一条流水线中工作，但只有它在U 流水线中工作时，才允许别的指令占用V 流水线。反之，如果它工作在V 流水线中，U 流水线就被暂停工作。

第三类指令和前一类相反，在V 流水线工作时，才能和U 流水线协同工作，反之则只能单独处理。

第四类不被推荐使用，它们是不可能和别的指令并行运行的，这种指令只能工作在U 流水线内，而且同时V 流水线一定是闲置的。

最受欢迎的第一类指令通常都是些常用的简单指令，比如mov，读写内存和寄存器时都有这种特性。nop push pop lea inc dec add sub cmp and or xor 这些指令都符合要求，大部分的数学运算都可以用这些指令完成。test 有点例外，在处理立即数与内存或者非eax 寄存器和内存时，才可以和其他指令配对运行，在这点上又显出eax 的优越性。

第二类只能在U 流水线和别人配对的指令有adc，sbb 这样的带进位加减法和移位的指令。当然，也不是所有移位指令都满足条件，循环移位指令只有在移动一位时才可以这样工作，普通移位指令也要求参数是立即数而不是cl。

第三类指令是一些短跳转指令，包括call，jmp，以及条件分支。

其他剩下的所有指令都是第四类，也就是独占CPU 的。

可以看到，支持流水线配对操作的指令非常有限，但是，这些有限的指令却能实现大部分的工作，所以应该充分利用它们。

Pentium 在处理指令流的时候，几乎总是会顺着把两条指令放进两条流水线让其并行运行，在编写汇编代码的时候，可以设想一下成对指令运行的情况。

当然，不完整地阅读一段代码的时候，不容易知道到底是哪两条指令配对在一起了。以不同的地方开始配对运行，由于第二类和第三类指

令的存在，成功配对的机会可能不一样，所以，应该尽量使用第一类指令，这样，使代码段不受上下文的影响（实际上，判断代码如何配对，只需要从每个可能跳转进来的分支跳转点找起就可以了，这是因为跳转指令要么不能与别的指令配对，要么必须放在 V 流水线中配对；从某条不循序配对的指令推算起也是一样）。

具体的配对规则如下：

（1）首先要满足指令队列中第一条指令允许 U 流水线配对，而第二条允许 V 流水线配对。

（2）上面的条件满足后，如果第一条指令写了某个寄存器，第二条指令就不能有读写这个寄存器的操作，即操作不得有依赖性。

后一条指令的操作依赖前一条指令的运行结果，当然不应该允许它们并行操作，这是一条理所当然的规定。

举几个例子:

```
mov eax, ebx
mov ecx, eax
```

后一句 ecx 的值依赖前一句的赋值结果，两条语句不能配对。但是，如果这样写：

```
mov eax, ebx
mov ecx, ebx
```

就是可以配对的，两句同时读 ebx 是被允许的。

```
mov ebx, eax
mov eax, 1
```

这样也允许配对，虽然第一句需要读取 eax，后一句又改了 eax，但是逻辑上并无依赖性。Intel 并未说明这是如何实现的，但是，Pentium Pro 及其后来的处理器系列公开了一种寄存器重命名技术，就是用来实现这种情况的并发。

同样，

```
mov ebx, eax
inc eax
```

允许配对，这里 inc eax 不仅需要写 eax 寄存器，还需要读它。

```
mov eax, 1
mov eax, 2
```

显然不能配对。因为不允许同时写 eax，这样写好像也没什么意义，下面换个有意义的指令对。

```
xor eax, eax
mov al, 1
```

这是个同时修改同一个寄存器而不能配对的例子。

需要特别说明的是，如果可以使用完整的寄存器，我们不主张使用寄存器的一部分，在许多特别的情况下，使用部分寄存器会有一点延迟，而且

```
mov al, 1
mov ah, 2
```

也是不能配对的，应该记住，ah,al,ax 这些都是 eax 寄存器的别名而不是独立的寄存器，写部分寄存器 al 和 ah 都会被看做对 eax 的修改。

（3）在上条规则里，有一个寄存器是例外，那就是标记寄存器。两条指令同时影响标记寄存器是可以配对的。甚至第一条写入标记，第二条读出标记也可以：

```
cmp eax, 2
ja label
```

是可以配对的。

（4）另一组特殊的指令是堆栈操作。

```
push eax
push ebx
```

或者：

```
pop eax
pop ebx
```

都可以配对，虽然它们都会读写 esp。甚至

```
push eax
call function
```

也是可以配对的。

（5）还有一些情况虽然可以配对，但是，成对执行消耗的时间比单一运行任何一条指令都长，这些情况通常都是由内存读写造成的。具体的时间消耗是可以精确计算出来的。阐述详细规则会花去很长的篇幅，而我们关心的也仅仅是如何编写代码更快而已，而不去关心具体的时钟消耗数量。遵循一个原则：配对的指令，应该按复杂程度，把简单的指令放在前面（U 流水线做）。判断复杂于否的简单标准就是：处理寄存器的指令比直读内存的指令简单，而需要读写内存的操作比直读内存要复杂。比如：

```
add [mem1], eax
add ebx, [mem2]
```

就不如：

```
add ebx, [mem2]
add [mem1], eax
```

因为 add [mem],reg 需要读一次内存并写回去，而 add reg,[mem]只需要把数据读出来。

另外，如果两个语句访问同一个双字的内存，也会发生延迟。例如：

```
mov al, [esi]
mov bl, [esi+1]
```

或是：

```
mov eax, [esi]
mov ebx, [esi]
```

都不能在一个时钟周期完成。

关于 Pentium 上 UV 流水线的配对规则，这里并没有把所有的细节都列出来，因为真正在运用的时候需要一种感悟和经验。另外，Pentium

以后的系列，流水线的工作方式又有变化，去抠 Pentium 上的优化规则的细节，并不一定在后面的 CPU 上同样适用，所以我们应该以大原则为先。

6.5.2　AGI 地址生成互锁（Address Generation Interlock）

如果一条语句需要访问内存，而内存地址的计算又依赖某些寄存器，当这些寄存器的数据是刚刚生成的，那么中间会发生延迟。这种现象在 Intel 的官方手册上被称为 AGI 延迟，AGI 的全称是 Address Generation Interlock，直译为中文是“地址生成互锁”，读起来非常拗口，所以下文全部直接用 AGI 来代替。

最常见的 AGI 延迟的例子是：

```
add edx, 4
mov eax, [edx]
```

在第一句运行完后，第二句不能立刻运行，这是因为第二句求地址的请求需要用到 edx 寄存器，而这个寄存器刚刚被修改过。这种情况下，我们一般会在两句之间插入一条语句（大多数可以通过调整多行语句的次序）来消除 AGI，或者简单地颠倒两句的次序也可以办到。

```
mov eax, [edx+4]
add edx, 4
```

只不过，这样的代码会变长一些了。

除了这种明显的 AGI，关于堆栈寄存器 esp 的 AGI 不那么明显被程序员发觉。

```
add esp, 4
pop eax
```

这样也会发生 AGI 的，因为 pop eax 需要用到 esp 做地址，但是，正如上文提到的，Pentium 对隐式的 esp 操作有特殊处理，所以：

```
pop ebx
pop eax
```

可以配对并发运行，而没有AGI的产生。

还可以举出另外的类似例子，如：

```
mov esp, ebp
ret
```

在ret的地方也是有AGI延迟的。

6.5.3 Pentium Pro，Pentium II和Pentium III的汇编优化策略

Intel的第六代微处理器（Pentium Pro/Pentium II）带来的重大技术革新是乱序执行。这比前一代处理器Pentium的双流水线配对并行无疑是一个重大改进，因为并行的语句并受相临的两条语句能否同时进入UV流水线的限制。

所谓乱序执行是这样工作的：CPU内部有多条流水线以及多个计算元件，比如计算加法用的元件、计算乘法用的元件等，当前语句执行需要的数据没有准备好需要等待的时候，处理器就去尝试后面的指令能否运行。

由于处理器有多个流水线，多条语句可以并发，比如一条乘法指令和一条加法指令。所以，如果后一条语句对前一条语句的运算结果没有依赖性，那么它们就能够同时执行。当然，两条乘法指令是不能并发的，因为处理器内只有一个计算乘法的元件。

这种乱序执行，旨在保证正确的程序运行逻辑下，通过打乱指令的运行次序来提高速度，但是，在支持乱序执行的处理器上，手工适当地调整各条语句的次序对于提高性能依然有效。因为合理的指令次序，可以让前后结果依赖性减至最小。

新的处理器为了实现高效，内部采用的是简单指令集的技术，因为又需要和老的指令集兼容，所以Intel采用的是由一种解码元件把长指令集分解成多条微指令（micro-operation，缩写为uop）。

在微指令级别上，处理器只能做一些非常简单的操作，比如对两个寄存器进行操作，或者读写一个单元的内存。比如add eax,[mem]这

样的指令就会被分解成两条微指令：一条是把[mem]的内容读到某个临时寄存器中；另一条就是把这个临时寄存器累加到 eax 上。当长指令序列都被分解成更多的微指令后，微指令以乱序在多条流水线中执行，即可获得可观的运行效率。

针对乱序执行引入的另一项新技术是寄存器重命名。Pentium Pro，Pentium II，Pentium III 都有 40 个内部寄存器不允许程序员主动使用，Pentium 4 更是把这个数量增加到了 128 个。这些寄存器在逻辑允许的范围内可以被临时映射到寄存器的逻辑名字上。这样就允许并行中的微指令可以保存不同的逻辑寄存器的值，在操作都完成后，再把包含逻辑上的最后一个值的寄存器复制到永久的寄存器中。通过文字上的解释似乎不太好理解，我们看一个例子：

```
1. mov eax, [mem1]
2. mov ebx, [mem2]
3. add ebx, eax
4. imul eax, 7
5. mov [mem3], eax
6. mov [mem4], ebx
```

如果第一行把 mem1 的数据读入 eax 后，第 2 行 mem2 还不在缓存中，这样执行第二行指令需要等待一小段时间加载内存。下面的第 3 行加法又依赖第 2 行读入 ebx 操作完成。所以，如果顺序执行的话，到 3 行执行完之前，eax 的值不能被破坏掉，这样，第 4 行对 eax 的乘法由于会改变 eax 的值，而不能同时执行了。

现在有了寄存器重命名技术，第 4 行的乘法可以把 eax 乘法的结果输出到一个临时寄存器中，第 5 行保存 eax 的时候，实际是把这个临时寄存器的值写入内存，而第 6 行的 ebx 的值是由之前的 eax 与 ebx 之和决定的。等到所有操作都做完后，因为最终的 eax 的值应该等同于第 4 行 imul 的运行输出临时寄存器中的值，所以这个寄存器的值最终被复制 eax 中，这个过程被称为指令引退（retirement），临时寄存器可以供下面的代码继续任意使用。

寄存器重命名不针对状态寄存器，所以状态寄存器（包括整数指令集的状态寄存器和浮点状态寄存器）看起来工作得相对慢一些。

1. 流水线操作

CPU 在执行代码时，首先会从代码缓存中按 16 字节一组（非对齐方式）读入，这些成组的代码段被称做指令预取块。在CPU 内部，指令预取块以一个双倍长缓存的形式存在。这样，加载的时候既可以按 16 字节对齐，又能保证从任何一个指定位置的连接 16 字节都能加载进来（以 16 字节对齐的 32 字节就能保证包含所要的 16 字节了）。解码器对指令预取块里的字节分析后，计算出每条指令是怎么分割开的，再通过解码翻译为微指令。

CPU 里有三个解码器，分别称为 D0，D1，D2。它们最多可以在一时钟周期内完成连续三条指令的解码。如同 Pentium 上的双流水线的结构不可能在任何时候都被充分使用，并行完成三条指令解码也是有限制的。只有 D0 可以对任何指令解码，而 D1 和 D2 都只能处理简单指令，即翻译成唯一一条微指令的指令。

除此之外，若要 D1，D2 和 D0 并发，还需要 D0 解码出来的结果少于 4 条微指令；当然这三条等待解码的指令同处于一个指令预取块中也是必须的。

所以，我们尽量把指令排列成 4-1-1 的样子比较适合高效的解码，简单说，应该把复杂指令和简单指令穿插起来。如有可能，避免使用需要分解成超过 4 条微指令的指令，这些指令一定需要 D0 单元独立工作两个或两个以上的时钟周期去解码。

在微指令的级别，已经没有了 eax，ebx 等这些面向程序员的逻辑寄存器的名字，而是通过寄存器分配表（register allocation table，RAT）把 CPU 内部的寄存器分配给这些微指令去工作，这其实就是寄存器重命名的本质。当每条微指令通过 RAT 都得到了自己的工作寄存器后，就被送入了重定序缓存（reorder buffer，ROB），在这里，微指令可以被打乱取出，乱序执行就是这样被实现的。

被安排为合理次序的微指令，会被分别送向五个通道尽可能地并行工作，有两个通道是做数学运算（跳转指令可以看成是对 eip 的运算），一个通道被用来读内存，一个写内存，还有一个用于计算内存地址。

当 ROB 中的指令运行完，根据逻辑上的次序，临时寄存器中被复

制到永久的寄存器（eax，ebx 等）上，流水线再继续下一轮循环。这样，一个流水线被分成若干个阶段，由 CPU 内部不同的单元来处理，整体处理速度就取决于最慢的那个环节了。

在解码以后的阶段，流水线每个时钟周期只能处理 3 条微指令。这个限制来自于代码通过 RAT 做寄存器重命名的阶段。这里，这个单元一个时钟只能处理 3 个寄存器。而解码阶段最多可以解出 6 条微指令，维持在每时钟周期提供 3 条微指令是一件很容易的事情，只要不用一些非常复杂的指令（大多数很复杂的指令都可以手工分解成简单指令），通常解码并非瓶颈所在。

前面提到，在解码阶段之前，有一个指令预取的阶段，会把代码缓存中的指令字节取到片内的以 16 字节为一组的指令预取块里，在没有跳转指令的前提下，这里并不会拖慢流水线的后续处理。但是，如果有跳转指令，分析到跳转时就会引发新的指令预取以生成正确的指令预取块，这会需要额外的时间。这种消耗会因为跳转点的地址是 16 字节对齐而减少，所以对于效率要求特别高的核心循环，可以在跳转入口点上做代码 16 字节对齐来改善速度。

但是，为了 16 字节对齐而插入太多的 nop 显然又成为了另一大速度杀手，在这方面，平衡是一个难题，最终，我们大多会放弃推敲这个点上的优化。如果的确需要，我们会去找一些更有意义的指令代替数量庞大的连续 nop，比如 mov ebp，ebp 或者 add ebp，0 等等。这样做或许可以改进潜在的寄存器读延迟问题（马上就要提到）。

前面还提到，微指令执行完后，需要有一个指令引退的过程，这个过程为了保证程序运行逻辑之正确，就不能按微指令执行那样打乱次序，必须依次处理。

在这个过程中，处理这件事情的单元同样可以每个时钟周期处理三个微指令，由于前面的寄存器重命名过程也有相同的限制，表面看起来，这里算不上瓶颈。但是，由于 ROB 中的微指令乱序执行和依次引退这点上，就有可能大段时间没有事情做，而另一些时间大批微指令同时到达造成流水线堵塞在这个阶段。所以从这个意义上，被合理调整过次序的代码次序依旧是处理器自动做的代码乱序运行所不能取代的。

另一方面，在我们写非常小的循环时，如果循环内的微指令数不是3 的倍数，由于跳转指令会强迫分组运行的代码复位，重新从跳转点开始分组，这样，在循环内的最后一组指令可能只有一个微操作，而使本来每周期处理三个指令引退操作变为只处理一个。

2. 寄存器读延迟

由于拥有代码乱序执行并且在执行单元有专门的地址运算通道，对于第六代处理器，前一代 Pentium 处理器的 AGI 已经不是问题。但是，依然有新的问题出现。

由于通过 RAT 选取临时寄存器的时候，一时钟周期最多可以同时通过 3 条微指令。ROB 处理这些微指令的时候，有一个从永久寄存器读入数据到临时寄存器的过程，而每个时钟周期最多只能允许读两个永久寄存器，这往往会堵塞住流水线。为了方便理解，这里举一个例子：

```
MOV [EDI+ ESI], EAX
MOV EBX, [ESP + EBP]
```

第一条指令分解成两条微指令：一条微指令会去读 EAX，而另一条需要读 EDI 和 ESI。第二条指令对应一条微指令，它需要读 ESP 和 EBP。

对于 EBX 始终是没有读操作，只有写操作的。这三条微指令一共需要读 5 个逻辑寄存器，EDI，ESI，EAX，ESP，EBP。虽然三条微指令同时通过了 RAT，但是在进入 ROB 时，每时钟周期有只能读两个永久寄存器的限制，这样就发生了延迟。

这限于读取永久寄存器的情况，一旦某个永久寄存器被读到临时分配的寄存器中，在还没有引退前，再读它是没有消耗的。下面这个例子可以说明问题：

```
MOV EAX, EBX
SUB ECX, EAX
INC EBX
MOV EDX, [EAX]
ADD ESI, EBX
ADD EDI, ECX
```

以上六条指令每条都对应一条微指令，前三条微指令读取了 EBX，

EAX两个永久寄存器，并没有超过限制，所以没有寄存器读延迟。而后三条虽然需要读EAX，EBX，ECX，ESI，EDI，但是，EAX和EBX已经预先读入，而ECX的值是前面刚刚写过的，也就是说，也有一个临时寄存器中有它的值，那么需要从永久寄存器中读入的依旧只有两个，ESI和EDI。所以，这连续的6条指令都没有因为寄存器读延迟而堵塞流水线的。

为了方便判断某条指令会被转化为若干条微指令后，某条微指令有读哪些寄存器的可能性，下面列出一些基本的规则：

（1）写内存操作会被化为两个微指令。前一条指令会读入寄存器的是准备写内存的寄存器；后一条由于需要做地址计算，所以地址中含有的寄存器会被读取。

例如：mov[esi],eax 中，前一条微指令会读入 eax，后一条读入esi。

（2）读内存并修改寄存器的操作会被化为两条微指令。前一条分析地址，后一条做数学运算。

例如：add eax,[esi]中，前一条微指令会读入 esi，后一条读写eax。

（3）读内存，修改并写回内存的操作会被化为四条微指令。第一条需要读地址相关的寄存器，第二条读出待运算的寄存器，第三条把运算结果放在临时寄存器内，第四条又需要读地址相关的寄存器了。

例如：add [esi+edi],eax 中，第一条微指令读入esi，edi，第二条读入eax，第三条指令不可能读任何永久寄存器，第四条又需要再次读esi，edi（这个时候已经没有读寄存器延迟的问题了）。

（4）压栈操作。例如 push eax，将被分解为三条微指令。第一条微指令会去读eax，而第二条微指令会读入esp以计算地址，最后一条微指令将改变esp。

（5）出栈操作。例如pop eax，将被分解成两条微指令。第一条微指令将去读esp并且会对eax做一次写操作；后一条指令会修改esp。

（6）call 操作。分解为四条微指令，但前两条都是对 eip 的修改我们不必关心（eip 不能被重命名），第三条微指令会去读 esp，最

后一条会修改 esp。

(7)ret 操作。也是分解为四条微指令，第二和第四条基于和 call 同样的理由我们不需关心，而第一条微指令会去读 esp，第三条则会修改 esp。

第六代处理器上汇编优化的极限就是让每个时钟周期都能并发三个微指令。如果想让读取永久寄存器的环节保持这个极限，并非易事。因为我们很难准确地判断，从哪里开始的三条微指令是一组的。千万不要把这个三条微指令的分组和前面提到的指令解码分组混淆，这里的分组是在解码之后，通过 RAT 分配临时寄存器时，三个一组通过的微指令。

也不要因为不容易判断微指令怎样分组而忽略寄存器读延迟，因为从 RAT 到 ROB 的队列并不长，只有十个微指令大小，顾而解码器的输出微指令流的带宽波动还是经常有可能引起寄存器读延迟的（在条件分支预测失败时，这个队列会被清空，这个时候可以明确得到微指令分组的开始位置）。

为了减少寄存器读延迟，对于汇编级优化代码，建议尽量把对相同寄存器的读操作放在一起，这样就可以在第二次读操作时不需要读取永久寄存器；并且在相邻的代码上避免读太多不同的寄存器；在写过一个寄存器后的一小段距离内（大约 10 多个微指令范围内）再次读这个寄存器是很有效率的，当然，由于要避免指令间的依赖关系，刚写完一个寄存器就读它出来是不提倡的。

最后，如果你的下文需要访问堆栈或者使用 ebp 读取变量，那么在需要写 nop 对齐代码的地方考虑用 mov esp，esp 或者 mov ebp，ebp 代替，可以缓解下面可能发生的寄存器读延迟现象。

3．对非 32 位数据操作产生的延迟

在 32 位环境下，使用完整的 32 位寄存器，或者是以四字节一组操作内存数据都是高效的方式。至少从 486 年代，就有未被 Intel 官方文档列出的，由使用字节寄存器造成的延迟。例如：

```
mov al, [mem]
add ecx, eax
```

这两行之间就会发生延迟，对于早期的 CPU，并没有官方的说法说明这种延迟发生的原因，不过我们可以想像得到，al 并非一个独立的寄存器，而是 eax 的一部分，所以，对 eax 的后续读操作并不能立刻接着运行。到 Pentium Pro 后，这种现象变得更加严重(大约 5～6 个时钟周期)，而且从上文我们已经知道，处理器内部会采用寄存器重命名技术，所以写 al 的操作只是针对一个临时寄存器，而下次想把 eax 全部读出的时候，不仅需要从永久寄存器 eax 复制数据到另一个临时寄存器，还需要合并两个寄存器的值，才能得到一个仅仅被修改了低 8 位的 eax 出来。

上面的代码是可以用另外一种形式替换掉的：

```
movzx ecx, byte ptr [mem]
and eax, 0xffffff00
or ecx, eax
```

movzx 和 movsx 这两条指令在回避这种延迟情况时显得非常有用，而且 Pentium Pro 以后的 CPU 上执行这两条指令非常快。不过，在更早的处理器上，它们是一条很慢的指令。

那个时候，我们通常用这样的写法替代 movzx eax,byte ptr[mem]：

```
xor eax，eax
mov al，[mem]
```

幸运的是，第六代处理器对这种情况做了优化，它能记住前面 xor eax,eax 把 eax 清为零了，然后对 al 这个低八位写操作时，将不会产生延迟：

```
xor eax, eax
mov al, [mem]
mov eax, eax
```

它是没有延迟的，当然，这种优化仅限于高位清零的情况。这样的代码还是有延迟问题：

```
xor eax，eax
mov ah，[mem]
```

这种问题不仅仅存在于使用字节寄存器，在混合使用 ax 与 eax 等类似情况时也会发生。

```
mov eax, eax
```

这种混合操作 8 字节和 32 字节数据造成的延迟情况不仅仅发生在寄存器上，在读写内存时也会发生，只不过引起的原因是内存数据在 CPU 内的缓存而非临时寄存器。

```
mov byte ptr [esi], al
mov ebx, dword ptr [esi]
```

这两行，前一行向一个内存地址写了一个字节，后面想连同这个字节在内的四字节一起读出，延迟就发生了。而且挺严重，大约会停止 7～8 个时钟周期。

不过反过来的情况有点特殊，如果先写一个 32 位数，再从同一地址读一个字节就没有延迟问题，但从不同地址读还是会发生延迟的：

```
mov dword ptr [esi], eax
mov bl, byte ptr [esi] ; 无延迟
mov bh, byte ptr [esi+1] ; 发生延迟
```

对于内存读写缓存的问题，有一个特殊的情况也会发生延迟，或许会让不明真相的读者迷惑不解：

```
mov byte ptr [esi], al
mov ecx, dword ptr [esi+4096] ; 发生延迟
```

实际原因很简单，CPU 内部的内存数据缓存的定位索引只有 12 位，相隔 4096 整数倍的内存地址会被映射到同一缓存空间上，虽然这种情况很少发生，但我们还是应该避免 C 语言中的类或者结构的大小正好是 4096 字节。

4. 状态寄存器的问题

状态寄存器虽然不存在寄存器重命名，但一样有被部分修改的问题，所以也应该避免这种情况。其实，只要合理编码，这种错误不会犯的。

```
inc ecx
jbe label
```

就是一种错误的代码，因为 inc 只会修改 Z（零）标志，而 jbe 需要读

Z 标志和 C（进位）标志。这会使 jbe 处发生延迟，实际逻辑上也是一个程序 bug。

正确的写法是：

```
add ecx, 1
jbe label
```

另一个值得注意的是，移位操作只在移动一位的情况下才没有延迟，而别的情况下都可能发生：

```
shr eax, 1 / jz xx ; 没有延迟
shr eax, 2 / jz xx ; 延迟
shr eax, 2 / or eax, eax / jz xx ; 没有延迟
shr eax, 5 / jc xx ; 延迟
shr eax, 4 / shr eax, 1 / jc xx ; 没有延迟
shr eax, cl / jz xx ; 延迟
shrd eax, ebx, 1 / jz xx ; 延迟
rol ebx, 8 / jc xx ; 延迟
```

5. 小结

Intel 的第六代处理器（Pentium Pro，Pentium II，Pentium III）在内部把长指令分解成微指令来处理，执行速度的极限是每个时钟周期处理三条微指令。但是，这个极限速度往往是达不到的。

由于处理器内部有乱序执行和寄存器重命名等技术，流水线又分了足够多的处理阶段，所以性能波动不那么容易发生。

但是，过度依赖硬件的本身优化，是写不出汇编一级的高性能代码的，通常我们还是应该：

- 尽量避免程序上下文过度依赖。
- 减少跳转，尤其是条件跳转的情况。
- 对于寄存器的使用，应该避免突然大量使用不同寄存器的情况；如果有可能，应该使用完整的 32 位寄存器。
- 对于状态寄存器和条件跳转，应当正确地使用。
- 理解 CPU 的工作原理，书写汇编代码的时候考虑怎么让流水线工作得更流畅。

6.5.4 Pentium 4 的汇编优化策略

Pentium 4 不同于 Pentium Pro 到 Pentium II 的改变，也不同于 Pentium II 到 Pentium III。它是一块全新的 CPU，是 Intel 的第七代微处理器。

到今年为止，讨论 Pentium 4 上的指令优化都还算是比较新的话题。虽然 Pentium 4 面市已经有一段时间了，但是，到今天（2004 年 8 月）为止，Google 上依然很难找到相关的中文资料。我本人在写这一章前对 Pentium 4 的优化技术可以说是毫无实作经验，仅有对一些文字材料的学习。所以只能以介绍为主。

其实并不仅仅是 Pentium4，从 PentiumPro 开始，互联网上已很难看到关于 CPU 汇编一级优化的研究和讨论了。难得看到的一些汉字，全部是日文网站上的。

难道华人世界的程序员都不关心这些问题吗？如果非要找出几个关心 CPU 发展新技术的人，那我在自己的朋友圈子里仅仅只是知道有那么几个人用过诸如 SSE 这样的新指令集，而关心优化技术的人没有看到。这不禁让我对华人世界的游戏程序员有那么一点高不成低不就的感觉。认识的朋友里，往高处研究各种设计模式，对程序的构架游刃有余的人屈指可数；而在底层玩弄汇编指令得心应手的人也是凤毛麟角；在游戏制作圈内摸爬滚打许多年的老人们似乎都在某个技术阶段停滞不前，重复着做过又做的工作，而不断新进来的热血青年又过分依赖前人口传身授那些旧的经验。

或许有一天，这个行业的程序员队伍达到一定的数量时，技术交流以及英文的资料翻译达到一定规模后，情况会得以改善。

很高兴现在已经看到了这种势头，越来越多的好书被翻译，越来越多的人从事编程，希望下一步可以看到的互联网上的中文资料更加丰富。

1. Pentium 4 的技术革新

Pentium 4 对以前系列的重大技术革新是，它不再每次处理指令时都通过一个解码阶段——解码成微指令集。取而代之的是：在第一次解码后，直接把微指令本身缓存起来，避免重复解码。

这样做的原因可能是以前流水线上跟解码有关的部分成为了瓶颈，CPU 的频率越来越高，解码单元的复杂度导致其很难提高速度了。

踪迹缓存（Trace Cache，TC）用来缓存微指令，TC 的空间也是有限的，有 2048 * 6 个单位。如何合理地利用 TC 是 Pentium 4 上优化的重点。汇编指令解码成微指令后，每条微指令都会占据 1 个或 2 个单位。自然，尽量产生只占用一个单位的微指令是比较有效率的；另外，Pentium 4 在处理 TC 中的微指令时，一个时钟周期也只能处理 3 个单位，而不是 3 条微指令。

直接写内存地址的操作，必须包含一个 32 位长的地址数据，使得指令无可避免地一定会占据 TC 中的 2 个单位。这就是为什么不提倡在 C++这样的高级语言中使用全局变量的原因。而当一个微操作的操作数据中含有的数字可以用有符号 16 位数表示的话，那么就只需要占据一个单元了。也就是说，如果我们能保持操作数在-32768 到+32767 最好。所以，如果把全局变量都放到一个类里面，通过 this 指针加偏移的形式，由于这个类几乎不可能大于 32K，所以这样就避免了出现 32 位地址数据。

Pentium 4 的执行单元有 4 个通道，前两个负责计算或者寄存器赋值，后两个用于读写内存。具体的分工如表 6-1。

表 6-1　4 个通道的分工

通道	执行单元	子单元	速度
0	alu0	add,sub,mov	倍速
		逻辑运算	
		保存整数	
		分支	
0	mov	赋值和保存 fp,mmx,xmm	单速
		fxch	
1	alu1	add,sub,mov	倍速
1	int	其他操作	单速
		从 fp 和 mmx 执行单元借一些子单元用	

（续表）

通道	执行单元	子单元	速度
1	fp	fp add	半速
		fp mul	
		fp div	
		fp 其他	
1	mmx	mmx 运算	半速
		mmx 移位	
		mmx 其他	
2	load	所有的加载操作	单速
3	store	保存地址	单速

表 6-1 上描述了通道的分工，还有每个通道上拥有的执行单元，以及每个执行单元下的子单元。

可以看到，0 号计算通道上，整数指令的大部分运算都可以双倍速完成，1 号通道上，整数的加减法和赋值也是一样。这说明了，理想状况下，一个时钟周期，执行单元可以并发执行六条微指令，不过，mmx 指令和浮点指令只有一半的速度。

想达到每时钟周期六条微指令的极限也并非易事。一条微指令能否开始执行不仅取决于它所依赖的数据是否已经准备好，还要求它所归属的通道、执行单元以及属下的子单元已经空闲。好在乱序执行帮助我们尽量利用它们，只是仅可能而已，还是那条反复不变的道理：如果想达到最好的效果，则不要过分依赖处理器的自动工作。

如果编码不当，产生很长的指令结果依赖链，则比前一代处理器更为影响效率，因为这次执行单元分得更细，不同的执行单元之间的结果依赖会产生更长的延迟。

滥用部分寄存器以及状态标志寄存器的问题造成的效率影响在 Pentium 4 上也更为严重，所以和第六代处理器上的优化准则一样，应该避免使用 ah 这样的寄存器（相比而言，al 比 ah 略好一些），inc eax 或者 dec eax 这样的指令也应该用 add eax,1 和 sub eax,1 来代替（这两对指令对状态寄存器延迟的影响有一些细微的差别）。

为了增强乱序执行的效果，Pentium 4 的设计还引入了一个并不一定受欢迎的技术，那就是内存预读。比如下面这段代码：

```
imul eax, 5
mov [mem], eax
mov ebx, [mem]
```

由于乱序执行，第三行把内存读入 ebx 的操作很可能先于前两行完成，但是，CPU 首先应该保证逻辑的正确。不久以后，发现 ebx 读入了一个错误的值（在这之前，ebx 的值还处于临时寄存器中没有引退，所以不破坏逻辑的正确），重新读入正确的数据在某些极端的情况下会带来十多个始终周期的延迟，所以，应该特别注意避免这种情况的发生。

还有一些关于指令的细节：Pentium 4 上因为极大加快的简单整数运算的速度，移位和整数乘法操作就显得相对较慢了，如果有可能，用加法来模拟它们或许是一个好主意，但是不要陷入牛角尖，目前我们无法预测 Pentium 4 以后 CPU 的发展。

2. 小结

对于 Pentium 4，这里只能做一个简略的介绍，肯定还有许多有趣的东西需要我们去研究。可以根据 Intel 官方提供的手册在实践中发掘。对于整个 Intel Pentium 系列的指令优化技巧的讲解也到此为止，如果有人能从我的讲解中体会到其中的乐趣，就像当年我读 Michael Abrash 的书而一头钻进这种程序员的智力游戏中一样，那么欢迎给我写信分享你的收获。

6.5.5 如何测试代码的速度

在玩优化代码的智力游戏时，能够精确地测试到一段代码的耗时是一件非常重要的事。这方面，有 Intel 官方提供的性能分析工具 VTune，可惜价格比较昂贵。

其实，从 Pentium 开始，Intel 就为我们提供了最佳的工具，那就是每块 Pentium CPU 都支持的 RDTSC 指令。这条指令可以返回 CPU 加电以来，累计的时钟周期数，这个数字是一个 64 位整数，调用 RDTSC 后

RDTSC 指令在 VC 6.0 以前的版本并没有被编译器定义出来，我们需要做一个简单的宏定义来实现。看看下一个函数，它可以在 VC 环境中方便地使用：

放在 edx:eax 内。

```
#define RDTSC  _asm _emit 0x0f _asm _emit 0x31

#pragma warning (push)
#pragma warning (disable: 4035)
inline unsigned __int64 timestamp()
{
    RDTSC
}
#pragma warning (pop)
```

编译器内嵌这个对 timestamp()的调用，产生唯一的一行汇编代码 RDTSC，这个用宏定义 RDTSC 就是用来解决 VC 6 本身的内嵌汇编还不支持这条指令的问题的。

而暂时关掉 4035 号警告，是因为这个 inline 函数并没有写返回值。不过相信我，这样做，一定不影响程序的正常工作。

使用的时候，只需要把两次 timestamp 的返回值相减，就可以知道之间指令的耗时了。由于 CPU 的流水线工作的复杂性，还有 Windows 平台的多任务处理，这样得到的时间，可能多次检测出来的并不完全相同，甚至会有比较大的出入。我的经验是，对一段代码反复检测多次，取出时间最短的那个值作为多段代码速度比较的根据。这样做的理由是，某些意外会使检测出来的时间变长，但不会因为意外使代码运行得变快。

如果简单地用 RDTSC 指令检测很短的指令序列，结果将很不准确。正如上文中所描述的奔腾系列 CPU 的工作方式。现代 CPU 都是以流水线方式处理指令，在一定的周期内，指令甚至可能是乱序执行。根本不存在单一指令运行时间这个概念。所以很短的指令序列的执行时间，一定会跟上下文有关。

一般的解决方案是在调用 RDTSC 指令前，调用一次 cpuid，迫使 CPU 开始一个新的处理周期，前后的指令都是上下文无关的。另外，RDTSC 返回值有 64 位，放在 edx:eax 中，针对短期测速的行为，我们只取 eax 即可。

Intel 官方推荐是这样的：

```
#define rdtsc  _asm _emit 0x0f _asm _emit 0x31
#define cpuid __asm __emit 0x0f __asm __emit 0xa2

__asm {
      cpuid
      rdtsc
      mov time, eax
      … ; 需要检测的代码
      cpuid
      rdtsc
      sub eax,time
      mov time, eax
}
```

最后，读者应该知道，物理学中有测不准原则。任何测量都是会影响被测物的。所以，不要认为RDTSC的返回值单位是一个时钟周期，测量精度就可以达到那么高。

一般提高准确度的方法是反复测量，并取测到的最小值，并回避去测过短（精度有限）和过长（在多任务操作系统下，可能被操作系统切出线程）的代码段。

6.5.6 汇编能干的事情

通常，有经验的游戏程序员会在最核心的地方使用优化过的汇编来完成一些底层繁复的操作以提高效率。这些通常正是性能的瓶颈。

一个程序优化的热点往往是凭借程序员的经验估算和判断出来的，或者是通常软件的测试。无论如何，我们应该把握一个度，即在软件结构的优雅和性能优化上找到一个平衡点。

我个人相信，人的大脑一定能优化出比编译器更漂亮的代码，但也不想过分浪费我的脑力。处理细致的调整指令的次序和精心选择合适的指令外，汇编较之C的优点还在于它处理位操作时略胜C的语法一畴。所以在做一些加密算法或者是某些图像解码操作时，我在第一时间选择了汇编而不是C。另外，C/C++的语法上不支持SIMD（Single-Instruction-Multiple-Data）的语义，SIMD指令集的威力也必须靠汇编，这个话题下一章会专门介绍。

这里，尤其需要指出一个汇编优化的误区，那就是某些人喜欢用编译器编译一遍C的代码为汇编码，然后在编译器产生的结果上修改，这往往一无所获。因为人脑不可能在机器编排的代码上做出什么重大改进了。通常，当你决定用汇编书写一个函数时，就应该抛开C编译器的机器思维方式，从零堆砌代码。如果必须同时写一份C的实现，更多的作用则是用来校验汇编版本的逻辑是否正确。

不过，不要过于信任自己的汇编书写能力，在这点上，包括我在内许多我熟识的程序员都犯过错误。我自己的例子是，有一次我希望实现一个rc4算法，这个算法很简单，但是我需要极端的高效，所以使用了汇编。我对自己写汇编的经验和技巧非常自信，函数完成后，为了测试逻辑上的正确性，又写了一个C版本。当我比较两个版本的速度后，结果让人大跌眼镜。

C版本足足比汇编版快了一倍！

反复的研究终于让我找到了罪魁祸首，我为了让代码变得更短，使用了一条xchg指令交换一个寄存器和内存。这条语句需要的时间居然超过了整个内循环其他语句执行时间的总和。回头再查阅资料才知道，用xchg交换内存时，现代CPU为了解决某些并发可能引起的逻辑正确性问题，会强制触发一个LOCK#信号。在多处理器的环境下，这个信号能够确保处理器脱离任何共享内存使用造成的影响。这通常导致了缓存到实际内存的同步操作。所以整条语句在现代CPU上会产生超过100个时钟周期的延迟。这条语句允许寄存器对内存交换是为了实现多处理器环境下的线程锁，其中也包含了一些历史原因。

我们现在处理类似问题时，lock cmpxchg或许是更好的选择（lock前缀并不一定触发LOCK#信号，而xchg内存则一定会）。当我们不用在线程锁的实现时，xchg内存都应该回避使用。

这次错误固然是我对新CPU的规则了解还不甚全面造成的，但也从某个角度说明了不要太相信自己的能力，对于汇编优化经验尚浅的程序员更是这样。建议在完成一个汇编函数的同时，写一个C版本，和编译器做一个速度竞赛吧。无论是否在竞赛中落败，研究一下编译器产生的代码也会对自己优化代码的能力会有所提高的。

祝大家编码愉快！

SIMD 指令

单指令多数据(SIMD, Single-Instruction-Multiple-Data)编程是现代 CPU 高速处理多媒体数据的发展方向。因为这种理念在 C 语言诞生之初并不多见，而且，只能被看成一种借助特定硬件实现的数据处理技巧，所以并没有被传统编程语言所内置实现。

SIMD 指令现阶段必须靠汇编来使用，这一章是写给对 SIMD 不熟悉的人的。因为好的工具摆在那里不去用就是一种浪费。用 Alpha 混合图形程序中常见的操作的优化为起点，在这一章里引出了 SIMD 优化的概念，然后通过介绍 MMX 指令的使用来具体化这些内容。

这一章介绍最多的是 MMX 使用的技巧，这也是我自己用得比较多的，勉强算得上经验丰富了。至于 SSE、SSE2 介绍较少，因为触类旁通，所以就从略了。

或许是因为家中并不富裕，上了大学之后自己感觉已经成人，不想多伸手找家里要钱，所以我编程用的计算机一直都落后于时代。这样糟糕的硬件条件也刺激了自己，一定要写出尽可能发挥出计算机硬件性能的程序。

改善开发环境全部得益于网络在中国的普及。从 1997 年接触互联网，到 1998 年初，我辞去了学校网络中心的勤工俭学工作，转而投奔了学校门口的叫做“谢谢您”的第一家网吧。那家网吧是大学里一个姓谢的年轻老师开的，网吧的负责人是她的父亲，我们称呼为谢伯。

那年，是国产游戏不景气的一年，这多多少少受到了前一年臭名昭著的《血狮》的影响；而这一年，也是网上业余游戏制作团体异常活跃的一年。大家看到，一个业余水平都比不上，根本就没有也无法完成的游戏都可以充当商品贩卖，游戏制作也失去了神秘（或者说是神圣）感。几个网友在网上弄个小组声称要做游戏，这样的业余团队遍地都是，我也弄了一个。结局不用说，最终自然是销声匿迹了。

我们小组的策划去了金山接手《剑侠情缘 II》的策划案，我毕业后去了北京，小组中有那么一个人，也就是刚刚提到的 Eisen 当时所在的软件公司也想进入游戏开发行业。应该是当时他经常看我在小组的网络会议上抱怨开发机器太破，就资助了一些硬件，这样我得以跟上时代。

与此同时，我在 cfido 上认识的好友 tdlife peng（后来几年投奔创意鹰翔担任《碧血情天》的主程，现在在盛大主持《神迹》的开发）在 E-mail 闲聊中提到曾在某书店中觅到一本翻译于 Intel 官方出版的小册子：《Intel 体系结构 MMX 技术开发指南》。知道我对此方面很感兴趣，买下一本送给我。

这两件事情，促使我开始研究 Intel 的新技术——MMX 指令集，发现了汇编优化代码的一片新天地。

谢伯人不错，就是有点唠叨，他和谢老师都对我很好，让我把宿舍的机器搬到网吧中随便用，平时只需要随带帮忙解决一些网吧机器出的问题。这并不是一份正式的工作，也没有什么需要负担的责任。当时我除了帮忙架设了一个服务器，顺带自学了点 Linux 的知识外，还真没为那个网吧做过点什么。免费泡了 2 年网，电费都没有付过。现在回想起，如果没有谢伯的那个网吧，我不可能那么快拥有满中国的网友。正是无数网友中的一个，Eisen ，1998 年底从保定邮寄给我一块 Pentium 200 MMX 的 CPU 和一个 2.1G 的硬盘，帮助我升级机器。而在此之前，我一直在使用一台硬盘只有 210M 容量的 CyrixGx586/120。

7.1 单指令多数据编程

SIMD（Single-Instruction-Multiple-Data）是 Intel 的发明，它的宗旨

是在给指令加上一些限制，在损失一些操作上的灵活性的基础上，让 CPU 在有限的时钟周期内做更多的运算。其基本原理就是用一个超长的寄存器（64 位或 128 位）保存成组的多个短小的数据单位，然后用单一的数学运算指令，让这些成组的数据中每个单元做相同的运算。这样，CPU 便拥有了许多数字并行运算的能力。

这种需求最开始应该是来自游戏的图像数据处理，而后扩展到声音视频流领域。因为图像处理时，往往有大量的数据等待做相同的处理，如果有这样一种机制，可以一条指令做相邻多个数据的运算，那么处理速度将翻倍。

即使 Intel 没有为他们的处理器加上相关硬件指令集支持之前，有这种想法的人也应该有很多，我就是其中一个。其实，这种方法应当早就被广泛地用于游戏编程中了，下面以一个我早年自己独立发现的对 Alpha 混合优化的方法作为例子来讲解。

下面这个方法不能确定是否是我最先发现，但可以保证是我自己独立实现的。那是在 1998 年某天昏昏欲睡的课堂上，灵感忽然降临，让我逃掉了下一堂课赶到机房编写了出来。当天就把方案提交到 Allegro 的 Maillist 中，最后成为 Allegro 代码的一部分。非常荣幸地是，因此，我的名字被列入 Allegro 的作者名单。

7.1.1　Alpha 混合

计算机的颜色大多是用红绿蓝三个颜色的分量来表示的，如果想用一个颜色的像素覆盖到已有的一个像素上，只需要把新的颜色的数值覆盖上去即可。

但是，很多情况下，我们需要做一种混合效果，就是把新的像素以半透明的方式覆盖到旧的像素上，这就需要使用 Alpha 混合。

简单的半透明效果做起来很简单，就是把新旧像素的各颜色分量分别相加除二，取得平均值就够了。但如果我们想做到任意透明度，那么就需要将这个算法推广：设透明度 Alpha 的范围是[0,1]，新的像素颜色为（red, green, blue），老的像素的颜色为（RED,GREEN,BLUE），

而混合后的像素为（red'，green'，blue'）。

计算公式上是这样的：

```
red'=red*alpha+RED*(1-alpha)
green'=green*alpha+GREEN*(1-alpha)
blue'=blue*alpha+BLUE*(1-alpha)
```

因为，计算机处理整数比较快。真彩模式下，Alpha 一般不以浮点数存在，而是[0,256]之间的整数，上面的公式需要做相应的修改：

```
red'=(red*alpha+RED*(256-alpha))/256
green'=(green*alpha+GREEN*(256-alpha))/256
blue'=(blue*alpha+BLUE*(256-alpha))/256
```

很长时间以来，RGB565 的高彩模式广泛地用在 2D 游戏中，它被大多数显卡支持，比真彩模式节省空间，而又不需要理会麻烦的调色盘操作，是许多游戏程序员的最爱。

Alpha 混合一直是一个比较慢的操作，因为每个像素需要做 6 次乘法，565 模式下，Alpha 混合函数的传统 C 语言版本类似下面代码：

```
// alpha 的范围是 [0,32] ； x,y 分别是需要混合的两个像素
unsigned short alpha_blender(unsigned short x,unsigned short y,
                             unsigned int alpha)
{
static const unsigned int red_mask   = 0xf800;
static const unsigned int green_mask = 0x07e0;
static const unsigned int blue_mask  = 0x001f;

unsigned int red_x  = x & red_mask;
unsigned int green_x= x & green_mask;
unsigned int blue_x = x & blue_mask;

unsigned int red_y  = y & red_mask;
unsigned int green_y= y & green_mask;
unsigned int blue_y = y & blue_mask;

unsigned int red   = red_x * alpha + red_y * (32-alpha);
unsigned int green = green_x * alpha + green_y * (32-alpha);
unsigned int blue  = blue_x * alpha + blue_y * (32-alpha);
```

```
    return (unsigned short)(((red & (red_mask * 32)) | (green &
(green_mask * 32)) | (red & (blue_mask * 32)))/32);
    }
```

这个函数很容易理解，预定义了一些常量red_mask, green_mask, blue_mask作为16位像素数据中取出RGB分别的掩码。

然后把x，y的RGB分别取出来，做完alpha混合的运算之后，再用位操作将其合并起来返回。

其中，使用了6次乘法，而且分离RGB还需要用位操作。我们看到，其实每次乘法都只处理了一个很小的数字，远远没有利用全CPU 32位处理能力。如果能找到一个方法，可以同时计算RGB值，那么速度会大大加快。想到这一点，具体的算法唾手可得。

从二进制看，16位RGB565的像素是这样分布的：

```
RRRRR    GGGGGG    BBBBB
5位      6位       5位
```

而386以上CPU都有32位的寄存器，我们只需要将16bit RGB变形为：

```
00000    GGGGGG    00000    RRRRR    000000    BBBBB
5位      6位       5位      5位      6位       5位
```

储存在32位寄存器中（就是把绿色提到前16位里），由于64k色下，颜色深度是32级的，所以Alpha也只用分32级就能满足需要。那么对上面变形过的双字处理，就可以同时算RGB了。

同时，(color1*alpha+color2*(32-alpha))/32在这里简化为(color1-color2)*alpha/32+color2也是可行的。

虽然感觉上减法将产生负数，这样再算乘法时有可能出问题，但是经过测试，这样简化是没有问题的。 新的算法只用算一次乘法，而且位操作的次数也大为减少。

```
unsigned short alpha_blender2(unsigned int x,unsigned int y,
                              unsigned int alpha)
{
x = (x | (x<<16)) & 0x7E0F81F;
y = (y | (y<<16)) & 0x7E0F81F;
unsigned int result = ((x - y) * alpha / 32 + y) & 0x7E0F81F;
return (unsigned short)((result&0xFFFF) | (result>>16));
}
```

这里说明了最朴素的SIMD的思想，让CPU在做数值计算的时候一次运算同时处理了几个数据，达到了加速。

当然，在乘法还不那么快的CPU上，这种单像素点的Alpha混合函数还有别的加快速度的技巧，除了用汇编仔细优化之外，我们还用宏来产生32个Alpha为0~32之间的常数的函数，让算法成的乘法变为常数乘法运算，这样编译器可以做更多的优化。

```
#define ALPHA_BLENDER_IMPLEMENT(alpha) \
unsigned short alpha_blender_##alpha (unsigned int x,unsigned int y)\
{\
x = (x | (x<<16)) & 0x7E0F81F;\
y = (y | (y<<16)) & 0x7E0F81F;\
unsigned int result = ((x - y) * alpha / 32 + y) & 0x7E0F81F;\
return (unsigned short)((result&0xFFFF)|(result>>16));\
}

ALPHA_BLENDER_IMPLEMENT(0)
ALPHA_BLENDER_IMPLEMENT(1)
...
ALPHA_BLENDER_IMPLEMENT(31)

unsigned short (*alpha_blender3[32])(unsigned int, unsigned y)={
alpha_blender_0,
alpha_blender_1,
...
alpha_blender_31,
};
```

定义了32个函数后，就可以利用函数指针数组来调用不同透明度的Alpha混合函数了。

7.2 MMX

1997年，Intel发布了支持MMX技术的Pentium处理器。新的处理器增加了一组对64位数据组中多个数据元素操作的指令，用一条简单指令就可以并行地对分组的字节、单字或是双字做数学运算和逻辑运算。这样，通过对数据的重复运算处理的能力增强来改善程序中图像、声音等数据处理的效率。

MMX提供了8个64位通用寄存器，Intel体系结构中寄存器不足的问题得到了极大的改观。MMX的寄存器虽然是64位的，但是它是并存于现有的体系结构之下，并不需要新的处理模式，也非64位的体系结构，所以对于数据的处理还是限制在32位之下。但是，MMX却带来了两个新概念。

首先是前面已经提到的把64位寄存器中的数据当做8个字节数据或4个单字数据，又或两个双字数据同时进行操作，这样做的好处在上一节我们已经看到。

其次是支持了一种被称做饱和处理的新的算术运算方式，这在图像和声音的数据处理时特别有用。

对应于饱和方式的传统算术运算方式是环绕式处理方式，也就是如果我们对一个无符号字节类型的数据做加法，由于单个字节能够表示的数字是[0,255]，那么对于255+1这种运算，环绕式处理方式会把结果认为是0，而把是否需要进位作为标志设到标志寄存器中。但是图像和声音处理上，我们需要的往往不是这种环绕，而是需要在上溢或下溢后，按边界进行剪取。也就是说，对于字节加法，255+1的结果更希望是255而不是零。

这种处理方式称做饱和方式，典型的应用之一就是做光线叠加，把某种颜色的光叠加到已有的颜色上，简易的算法是把RGB各颜色分量相加。我们当然不希望叠加后的颜色反而变黑，所以饱和处理方式就很重要了。若以传统指令以环绕方式处理这类问题时还需要根据进位标志做条件分支处理，则大大地降低了流水线的效率。

MMX指令的另一个特点是，所有的指令都不影响标志位，这样的

工作方式不会引起标记寄存器读写延迟，也强迫用户在写程序的时候使用一些技巧避免分支，使流水线工作流畅。

7.2.1 MMX 的指令概况

最初的 MMX 指令集不算太丰富，也很容易被程序员掌握。鉴于本书并非一本 Intel 汇编开发手册，这里不一一列出每一条 MMX 指令的功能、技术指标、使用细节，那些可以从 Intel 的官方文档中找到，或者在互联网上花点心思也能搜索到。这里仅给出一个概况，增进大家对其的了解。

MMX 提供了 8 个 64 位寄存器，分别写做 mm0，mm1，...mm7。我们把 MMX 指令分为 7 类。

1. 算术运算指令

MMX 提供了数据分组的并行加减指令，既支持饱和方式的加减法，也支持环绕方式。对于饱和方式又把有符号和无符号的加减法区分开。汇编指令的助记符很有规律，不难记住，用过几次后就不需依赖文档。

环绕加减法的指令分别是 PADD 和 PSUB，如果是有符号饱和方式，就写做 PADDS/PSUBS，无符号饱和方式是 PADDUS/PSUBUS。

指令助记符的最后一个字母再说明如何分组：PADDUSB 表示按字节分组做无符号饱和方式加法，最后一个 B 表示 Byte 字节，W 表示 Word 单字，D 表示 DWord 双字，后面提到的移位操作还可能对整个 64 位字进行操作，记做 Q。

由于 MMX 指令还是 IA32 指令，所以不可能做 64 位算术运算，这些指令最多也只能处理到双字长了。饱和方式的加减法这里只支持 B 和 W，即字节和单字分组。

我们还可以利用 MMX 指令做无符号数据分组的乘法，仅限于单字的运算，这在大多数情况下都是足够的。由于 16 位无符号整数乘法运算的结果是一个 32 位数，MMX 做乘法运算时需要分开取出乘法结果的高低 16 位，PMULHW 可以把两组四个单字相乘取其结果的四个高 16 位，而 PMULLW 则相反，取出结果的低 16 位。做一些小数值的乘法的

时候，结果本身就可以用 16 位表示出来，大多数情况我们并不需要完整的 32 位结果。

如果想一次得到完整的 32 位乘法结果，可以考虑使用 PMADDWD 分组乘加指令。这条指令的作用虽然是把源操作数和目的操作数各分成四个一组的有符号单字，结对相乘，然后把四个 32 位的中间结果两两相加产生两个 32 位的双字。但我们把操作数也当成两个范围在-32K 到+32k 间的双字而不是四个有符号单字时，这条语句实际上可以看做是一条产生 32 位结果的分组乘法运算。当然乘加运算在复数乘法运算中更显威力。

2．比较指令

MMX 的所有指令都不影响标志寄存器，所以比较指令的含义也和整数指令不同。PCMPEQ 相等比较和 PCMPGT 大于比较都是把结果赋成 FF 或者 00，得到 0 1 掩码。对于数据流处理，这种方式更为适用，我们可以通过逻辑运算得到想要的结果。

3．分组和成组指令

分组指令可以把两组字节组转换成一组单字组，或者两组单字组转换成一组双字组，两组双字组转换成一个 64 位数据。这个指令把 128 位的数据转换成 64 位的，所以工作时需要指定处理高位还是低位，指令分别是 PUNPCKH 和 PUNPCKL。这个指令在装载数据的时候特别有用，后文将举出实例。

成组指令和上面相反，把相对长的字长转换成短的，这种处理都是按饱和方式工作的，我们需要指定的是有符号 PACKSS 还是无符号 PACKUS。一般，我们在做数据运算的时候，原始数据可能是字节，但是中间运算会使用到单字，而结果还是字节，这个时候就需要使用成组指令。

4．逻辑运算指令

和整数指令相同的是，MMX 提供了相应的 PAND，POR，PXOR 分别计算与，或，异或。

值得注意的是，MMX 指令集中所有运算的指令都带有两个操作数，

没有一元操作。所以，MMX 并没有提供NOT（非）操作，取而代之的是有一个 PANDN，也就是把本身取反然后做与运算。实际上 PANDN 一个-1 就可以达到取反的效果。

5. 移位指令

逻辑左移和逻辑右移都和普通整数指令一样，如果需要移位的个数不是常数，就需要把移位个数放在MMX 寄存器中。

如果需要移位时扩展符号位，就可以使用算数右移指令 PSRA，不过 PSRA 不支持对整个 64 位寄存器移位。所有的移位操作都不能作用于字节组。

6. 赋值指令

对应于整数指令中的 MOV，对 MMX 寄存器之间的操作指令是 MOVQ。如果需要在 MMX 寄存器和常规 32 位整数寄存器之间传递数据就应该用 MOVD。

movd mm0,eax 这样从 32 位寄存器传递 32 位数据到 MMX 寄存器时，MMX 的高 32 位会被填入 0。

7. EMMS

这条指令比较特殊，如果希望下面的程序中浮点运算能够正常运算，就需要调用 emms，它清空了浮点标志字。C 语言下要求，每个使用了 MMX 指令的函数都应该在函数结束时调用一下 emms。MMX 的寄存器和浮点运算用的浮点寄存器其实物理上是同一个东西，只不过取了不同的名字而已。混合使用 MMX 指令和浮点指令是非常低效的，这点应该引起程序员的注意。

7.2.2 SSE 及其对 MMX 的扩展

SIMD 技术最初是起源于 Pentium MMX 对整数的支持，AMD 公司把其扩展到了浮点数的范围，也就是 3d Now!技术。同样也是用 64 位的寄存器并行处理两个浮点数。

而后，Intel 发布了具有 SSE 指令集的 Pentium III。SSE 为 CPU 增

加了一组8个128位寄存器，用于处理分组的浮点数字。Pentium 4的年代，SSE的指令集先后经历了两次扩展，分别被称做SSE2和SSE3。SSE2使得处理double类型和64位整型变为可能。而SSE3除了增加了浮点数向整数的转换外，还增加了用于线程同步的指令，这也侧面说明了Intel希望把CPU的功能向超线程方面延伸，即在一个物理CPU上实现多个逻辑CPU的功能。

本书并非一本SIMD编程的技术手册，只是想通过短短的一章内容给读者介绍这方面的知识和SIMD编程的手法。无论是操作整数还是浮点数，SIMD的思想是不变的，如果想多了解SSE具体的知识，非常遗憾，到目前为止，我还没有发现有相关的中文专著出版。

网络上有关的中文资料也少之可怜。要在本章深入讨论也是不可完成的任务，写下那些内容可能会超过整本书的篇幅。而且我本人目前也没过多经验。所以有兴趣的读者只能求助于Intel的官方英文文档，在参考文献中都有列出下载地址。

下面讨论的内容都会集中在MMX的整数分组运算的范畴之内。值得一提的是，SSE不仅仅是扩展了一组128位寄存器用于操作浮点数，还为MMX的64位寄存器的使用扩展了一组指令。这组指令分别是一些算术运算指令和一些赋值指令：

1. SSE扩展的算术运算指令

- 计算平均数：PAVGB/PAVGW分别计算无符号字节分组和无符号单字分组的平均值。
- min/max计算：PMAXUB/PMAXSW/PMINUB /PMINSW可以对比两个MMX寄存器的数据组取较大值或者较小值。从名字上就可以看出，对于字节分组是按无符号处理，而对于单字分组是按有符号处理。
- PSADBW：计算字节分组数据的绝对差值之和，结果是一个单字。
- PSHUFW：这是用于打乱MMX寄存器中4个单字的次序的指令，用一个8位立即数表明新的次序（每2位可以描述0~3的位置信息）。

2. SSE扩展的赋值指令

- PEXTRW/PINSRW：可以把一个单字插入或解出MMX寄存器。
- PMOVMSKB：把MMX寄存器中的数据按字节分组，取每个字节的符号位组成一个8位的数据放入常规整数寄存器中。

扩展的指令目前使用起来需要注意，因为SSE需要Pentium III以后的处理器才能支持，不像MMX技术被支持得那么普遍。我自己也是考虑到这个原因，并没有广泛使用它们。同样的原因，Pentium 4的SSE2把对64位MMX寄存器的整数分组运算指令扩展到了128位的XMM寄存器上，但是使用它们也需要考虑兼容性问题。下文的讨论也不再提及这些SIMD指令扩展。

7.2.3 SIMD技术在C语言环境下的使用

由于SIMD技术着重于同一条指令操作多个数据，这个概念是C/C++标准中没有的。为了让C支持SIMD，C/C++编译器只能以函数的形式提供这些功能。

这些SIMD指令在C中看似函数调用，但实质编译器会为其做特别优化，认为这些函数是内建（intrinsic）的。

最早是Intel C++支持了SIMD的指令，而后在VC 7中也得到了支持。很自然的，这些编译器对于它们的支持，无论在类型名还是指令名上都采用了相同的标准，我们无需担心这部分的代码对编译器的兼容性。

在VC 7上，使用MMX指令需要先包含：

```
#include <mmintrin.h>
```

对于SSE指令则是 <xmmintrin.h>。

mmintrin.h中定义了__m64这种类型，每个MMX指令都被映射到了相关的 intrinsic 函数上。比如__m64_mm_and_and_si64 (__m64 m1,__m64 m2)就对应了MMX指令pand。

关于intrinsic函数

intrinsic函数并非为了解决C语言支持SIMD才诞生的。最早为了提高 memcpy 的效率，不少编译器已经把 memcpy 作为特殊的函数处理。

打开编译速度优化的情况下，memcpy操作于一个很小的常数长度数据时，不仅仅不会产生函数调用，甚至连循环都被展开了。可见

memcpy 已经被编译器看做和关键字一样的处理方式，被特殊对待。在VC的头文件中，我们可以发现#pragma intrinsic (memcpy)这样的声明，表明碰到 memcpy 这个函数调用时，应当寻求编译器内部的特殊解决方案，而不是简单地从标准库中连接一个函数。

到了有浮点处理器的年代，CPU 已经可以直接用指令处理一些浮点数学运算，比如fabs, fsin 等。许多运算指令是 C 语言标准本身没有的，为了让编译的目标代码更能发挥CPU 的性能，现代编译器也把这些数学运算的函数调用特别对待了。

今天，MMX 和 sse 指令集也受到了相同的待遇。这种对 C 语言扩展方法相信会一直延续下去。

通过查看这个头文件很容易理解每个函数的对应关系，在 MSDN 中也能查到更详尽的说明，此处就不一一列出了。

虽然通过编译器对SIMD 的支持扩展，我们可以很方便地使用SIMD（最重要的是方便调试）。但是需要用到 SIMD 的地方都是需要高速处理数据的场合，大多数情况下，自己书写汇编会更放心一些。这样更能清楚数据是如何在寄存器间工作的。我个人的习惯还是以使用内嵌汇编书写MMX 代码为主，另外也不是所有编译器都支持 SIMD 扩展（例如VC 6 就不支持）。至于两种方式孰优孰劣，还需要使用者自己来判断。

7.2.4 MMX 编码技巧

初次使用 MMX，会以常规指令的角度思考问题，往往一些技巧会被忽略。这里列出一些，供有兴趣的读者参考。

1．给寄存器赋值

（1）读内存

MMX 指令不支持64 位的立即数。所以我们必须把数据先放在内存中，然后从内存地址装载。比如希望给 mm0 赋值 0x123456789abcdef0，由于不能直接写 movq mm0,0x123456789abcdef0，所以如果使用内嵌汇编，通常可以写成：

```
static const unsigned long x123456789abcdef0[2] =
        {0x9abcdef0,0x12345678};
__asm movq mm0,x123456789abcdef0
```

读内存是一种低效的行为，尤其是用绝对地址。通常我们会把常用的64位常数放在连续的内存中，然后通过偏移量来读取。

例如，在做16位色的Alpha混合时经常会用到红色、绿色、蓝色的掩码，分别是0xf800,0x07e0,0x001f。

```
static unsigned short color_mask[3][4]={
    {0xf800,0xf800,0xf800,0xf800},
    {0x07e0,0x07e0,0x07e0,0x07e0},
    {0x001f,0x001f,0x001f,0x001f}
};

#define red_mask 0
#define green_mask 8
#define blue_mask 16
```

在MMX函数中使用时，可以先把color_mask地址加载到常规寄存器中，再使用：

```
__asm {
    lea edx,color_mask
    movq mm0,[edx+red_mask]
    movq mm1,[edx+green_mask]
    movq mm2,[edx+blue_mask]
};
```

（2）常数生成

某些常数，甚至能绕开读内存的操作。比如想将mm0寄存器赋为0那么pxor mm0,mm0已经能达到效果。

同样，让mm0全部位都为1的话（即每个字段都是-1），就应该这样操作：

```
pcmpeq mm0,mm0
```

想让每个字节都是常数1的话，可以：

```
pxor mm0,mm0
pcmpeq mm1,mm1
psubb mm0,mm1
```

想让每个单字都是2^n-1（例如3,7,15,31...）：

```
pcmpeg mm0,mm0
psrlw mm0,16-n
```

想让每个单字都是-2^n时：

```
pcmpeq mm0,mm0
psllw mm0,n
```

mmx不支持字节移位。

（3）复制数据

如果想把一个32位的整数复制成两份放在MMX寄存器中，可以这样操作：

```
mov eax,0x12345678
movd mm0,eax
punpckldq mm0,mm0
```

最后这句punpckldq mm0,mm0就把mm0的低32位数据复制到高32位了，这样的操作在常规整数指令集中是没有的，所以应该值得初涉SIMD编程的程序员注意。

同理，如果我们需要把整个mm0设置成8个相同的常数，也可以考虑这种方式。

```
movd mm0,eax ; eax是一个0~255之间的常数
punpcklbw mm0,mm0
punpcklwd mm0,mm0
punpckldq mm0,mm0
```

三条语句把一个数据复制成了一组八个。

2. 取绝对值

要求：输入mm0，按4个一组的单字，取绝对值放在mm1中。

```
movq mm1,mm0
psraw mm0,15 ; 复制符号位（正数清零，负数设为全1）
```

最大的负数0x8000的绝对值结果是0x7ffff，这是一个错误的结果，但是却比0更合理。

```
pxor mm0,mm1      ; 负数则取 1 的补码
psubsw mm1,mm0    ; 负数则加 1
```

3. 把无符号数截取到任意一个区间[low,high]

这里示范了让一个数字大于 high 的时候让它等于 high，小于 low 时等于 low 的方法。

```
paddusw mm0,0xffff-high
psubusw mm0,(0xffff-high+low)
paddw mm0,low
```

如果是有符号数，这个处理会相对麻烦一点，我们需要先把有符号数转换到无符号数的区间，做截断处理再转回来。这个问题可以作为一个智力游戏留给读者考虑。

4. 无符号数值差值的绝对值

求 pabs(mm0-mm1)

```
movq mm2,mm0
psubsb mm0,mm1
psubsb mm1,mm2
por mm0,mm1
```

这个例子利用了饱和方式的减法。让 mm0 先减 mm1，再用 mm1 减 mm0。因为两组数据，较小的数据减教大的数据时，结果会因为饱和方式减而成为零，所以 por 操作可以得到绝对值。

如果操作数是有符号数，则需要更多的技巧，这里也留给读者把玩。

5. 灵活运用掩码

MMX 指令都不影响标准寄存器，它的比较指令会产生一组掩码。我们的数据处理通常会利用这组掩码做运算。

掩码的每个数据单元要么全为 0，要么每一位都是 1，即数字-1。方便我们接下来使用 pand por 做逻辑运算。

不少初用 MMX 的程序员都抱怨过 MMX 没有提供 not 操作，而有些时候却需要使用掩码的反。前面已经提到，pandn 对-1 操作可以达到取反的效果。但大多数情况下，我们使用一些技巧并不需要得到掩码的反。

比如，如果mm1中存放的是一组字节掩码，那么：

```
pcmpeqb mm2,mm2   ; mm2=-1
pandn mm1,mm2     ; 将 mm1 中的掩码取反
pand mm0, mm1
```

这组操作并不需要做得这样复杂，一句简单的语句就可以代替：

```
psubusb mm0,mm1
```

因为pand mm0，~mm1的目的就是将mm1掩码中1的部分清零，而把零的部分留下来，psubusb可以达到同样的目的。

同理，por mm0，~mm1也可以用paddusb mm0,mm1代替。

7.2.5 SIMD的应用举例

我在自己开发的2D图像程序库“风魂”中大量使用了MMX指令在Alpha混合等图像处理的场合，事实证明在这些图像处理的运算中，MMX极大地提高了处理速度。“风魂”本身是开源代码，大家可以下载翻看，以做参考。

另一方面，在3D游戏中，SSE的分组浮点运算也能发挥很大的效能，比如skin deformation，使用SSE就比常规方法逐个顶点计算要快得多。

编写SIMD代码有时更像一个智力游戏，需要充分展开程序员的想像力才可以做得更好。这里，我想举一个例子，它使用MMX加速16位高彩模式下的Additive混合。这个程序是由我的一个网友xiaonan5010所作，他改进了“风魂”中的Additive混合的程序。

当时我看到时觉得饶有趣味，放在此处供大家欣赏。

1. Additive混合

和Alpha混合不同，Additive混合用在光线叠加的效果。Additive混合中，直接把源图和目的图上的颜色分量累加。

如果结果超过了颜色分量所能表示的最大范围，就取其最大值。也就是说，把不同的颜色多次Additive混合后，最终的颜色应该是白色，

即各个颜色分量都达到了最大值。

从算法本身看，就很适合 MMX 来描述，因为 MMX 本身就支持饱和加的模式。

输入：mm0 是源图的 4 个 16 位像素，mm3 是目标图的 4 个 16 位像素

mm6 存放的是红色加蓝色的掩码 0xf81f f81f f81f f81f

mm7 存放的是绿色的掩码 0x07e0 07e0 07e0 07e0

输出：mm0 是 Additive 混合后的 4 个像素

```
movq mm1,mm0 ; 复制源像素
movq mm2,mm3 ; 复制目的像素

pand mm0, mm6      ;mm0 = 源像素中红蓝分量保留，将绿色的位设置为 0
por mm3, mm7       ;mm3 = 目的像素中红蓝分量保留，将绿色的位设置为 1
pand mm1, mm7      ;mm1 = 源像素中绿色分量保留，红蓝色的位设置为 0
por mm2, mm6       ;mm2 = 目的像素中的绿色分量保留，并将红蓝色的位设置为 1

paddusb mm0, mm3
paddusw mm1, mm2

pand mm0, mm1
```

这段程序的巧妙之处在于最后三行，为了方便讲解，我们把前六行运行完后的寄存。：

下面希望用 Excel 做一张表。

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mm0	Rs	Rs	Rs	Rs	Rs	0	0	0	0	0	0	Bs	Bs	Bs	Bs	Bs
mm1	0	0	0	0	0	Gs	Gs	Gs	Gs	Gs	Gs	0	0	0	0	0
mm2	1	1	1	1	1	Gd	Gd	Gd	Gd	Gd	Gd	1	1	1	1	1
mm3	Rd	Rd	Rd	Rd	Rd	1	1	1	1	1	1	Bd	Bd	Bd	Bd	Bd

paddusb mm0,mm3 可以让 Rs 和 Rd 以及 Bs 和 Bd 都可以正确地做饱和加操作，并且把结果的绿色分量部分都填为 1。

paddusw mm1,mm2 则可以让 Gs 和 Gd 正确地进行饱和加，并且把结果的红色以及蓝色分量设置为 1。

最后，pand mm0,mm1 将合并结果，生成组合后的 RGB 像素值。

这就是一个巧妙地利用了饱和方式加又使用了 MMX 的单指令处理

多个数据的典型程序。

7.2.6 SIMD编程注意要点

（1）因为并非所有的CPU都支持MMX,SSE...，所以我们最好为不同的CPU写不同的代码，让不支持SSE的CPU也可以运行。

这里，需要检测应用程序所运行的CPU到底支持哪些指令集。Intel为Pentium以后的CPU都提供了一个叫做cpuid的汇编指令完成这件事情。

而微软在MSDN中提供了一个使用cpuid的例子供我们参考。例子程序中，演示了如何检测CPU时候支持MMX。例子可以在MSDN中通过关键字cpuid查询到。

但是对于游戏，要求相对较高的数据处理能力，今天我们已经可以默认用户的CPU拥有MMX指令集（但到今天，还不能默认用户的机器支持SSE，毕竟Pentium III还没有如此地普及）。因为，如果用户的CPU不支持MMX，说明它已经是一块已经过时的处理器，即使为它把每个数据处理函数写一个不需要使用MMX指令集的版本，游戏的速度也只会慢到不能接受。

（2）千万不要把MMX指令和浮点运算混用，MMX寄存器和浮点寄存器是物理上重叠的寄存器。而且两种指令的切换也会造成CPU很大的延迟。

在没有SSE的场合，我们想使用SIMD技术处理小数时，可以考虑使用定点数。

（3）对于内嵌汇编，在函数结束时，如果函数运行过程使用过MMX指令，一定记住调用emms。不然，其后一旦用到浮点运算，将会引起计算错误。

（4）MMX中操作内存时，地址应该按照8字节对齐；而对于sse，则应该保持16字节对齐。

7.2.7 SIMD 的性能检测

Intel 的 VTune 工具可以精确地分析 SIMD 的性能监测状况，包括它们对流水线的使用情况等。它是通过汇编指令 RDPMC 实现的。

RDPMC 可以得到许多 CPU 内部的工作状态，我们自己写可以使用它。不过，RDPMC 的使用详解会占用过多的篇幅，感兴趣的读者可以根据书后参考网站的地址从 Intel 官方网站下载文档。

引擎中的优化

单从内容来看，这一章是本书的独家内容。我得以成名的开源游戏引擎“风魂”的几乎所有优化策略都在这章公开出来。虽然引擎源码一直都是公开的，想研究的人自然可以通过阅读代码来学习，但是，由作者本人来解释其中的优化策略，这是第一次的文字记录。

有些东西并不新鲜，比如 Blit 优化、脏矩形技术等，但是，改进过的网格脏矩形技术，却是我自主发明的。

通过这一章，我最想让游戏程序员明白：在游戏引擎中，永远都可以有性能潜力去挖掘。优化是在各个层面进行的事情，可以进行大规模性能提升的地方往往在于高层次的设计，而并非低层面的热点上。

这一章提到了图形压缩的一些内容，关于 DXT 压缩格式应该引起大家的注意。这个由 S3 公司提出的图片压缩方案为 3D 游戏节省了大量的显存，一个好的方法真的是顶过几年的硬件发展。这值得我们程序员反思。

2000 年走出校园后，我有了大块的自由时间。似乎从来没有为工作发愁过，只要国内还有制作游戏的公司，找份兼职就不是难事。这一点得益于大学期间的游戏之作，那份公开源代码的游戏程序库“风魂”（标识如下图 8-1）。



图 8-1 “风魂”标识

虽然从头到尾都没有对其的使用收取版权费用，而且代码也 100% 毫无保留地公开，但是从中间受益是巨大的。我从维护这套代码的过程中学到了不少东西，交了许多朋友。并非出于我本人的要求，有些公司最终还是以咨询费用的方式，汇了一些钱给我。虽然不多，但是作为维持毕业后那段无业游民的生活却颇为重要。

直到 2001 年，“风魂”经历了三个大版本的全部改写。见证了我四年的学习历程。以现在我的眼光回头看，那些代码甚至是幼稚可笑的。程序中充斥着许多蹩脚的编码方法、不成熟的整体架构、非系统的编码规范和不严谨的接口定义。对于干了这么一件贻笑大方的事情，正如写这本书一样，我自己却没有后悔过。敢于把自己曾经的无知展现给世人也算是一种勇气吧。

只要自己能保持真诚和谦逊，存在的错误就能够得到理解；虽然整体上并不完善，但于后来者，依旧可以从中学到一些闪光点，而避免踏上过多的弯路。无论是从一开始公开“风魂”的源码，还是现在以并非专家的身份写这么一本大杂烩一样的书，我想，这些于己于人都还应该是有意义的事情。

关于“风魂”的新版本

我的个人网站上还能找到 2001 年版本的“风魂”下载，虽然更新

时间还很新，但是已经局限于细微处的 bug fix。许多 bug 的发现都得益于一直以来许许多多的用户，在此特表示我的感谢。许多朋友关心下个版本的推出，我只好借此一角表示道歉，或许这是一件遥遥无期不能给出承诺的事情了。

非常可惜，2002 年往后，重新写过的新的“风魂”的版本没有能公开，缘于这部分的代码已非业余作品，而全部移到工作时间维护。这些年来，我在网易工作，“风魂”的新老版本均用于商业游戏的开发。

涉及各种原因，公开新版本“风魂”源码的请求没有能获得公司的同意，这不能不说是一种遗憾。对于我最大的遗憾是，无法给网友展现和谐优美的代码。

而那些陈旧的、刺激审美神经的代码，作为个人网站的一个象征展现出来，于我，曾经几乎是一件无法容忍的事情。

最终我没有将老版的“风魂”从个人网站上删除，不过是考虑到那一万多行代码中，毕竟还是有一些可圈可点的东西。也有不少人正在使用，作为唯一署名的作者，有义务给出一个发布 bug 修订版本的地方，这才让其保留了下来。

8.1　Blit 的优化

在“风魂”设计的初期，我曾经执著于提升 Blit 的速度。不错，作为图形引擎，Blit 是最底层的操作。游戏的程序追述到底，表现给用户的，也就是一些图片的传输和绘制过程罢了，而一张图片的数据往往又以千字节为单位计，最后会转化为单调的大量循环复制数据的操作，对于 CPU 来说是一件重负荷工作，这里显而易见地会成为速度瓶颈。

在许多改良方案中，很多人选择的是绕开 CPU，而用其他的硬件设备，也就是显卡来分担这个工作。这样，只需要把图片数据一次传输到显卡的显存中，以后图片数据移动就不必再通过 CPU 了。在游戏机平台上，这几乎是标准的解决方案，但是对于通用计算机平台，并非标准。当年“风魂”没有选择这种方案有两个主要原因：

- 把数据都提交到显存中失去了很多的灵活性，而且显存十分有限，做出完善的管理和调度模块十分不容易，通常还需要调用者对其的理解，做出适当的配合。一不小心，效率会瞬间下降得很剧烈。
- 2000 年以前，我的机器上安装的仅仅是一块带有 512K 显存没有任何加速性能的 Trident 9000i 显卡，硬件环境决定了不可能调试相关的程序。

事实证明，当年的选择是正确的。如今，无论是 CPU 还是内存总线的带宽已经足够快，可以支撑十分复杂的平面游戏，过多地依赖显示设备硬件变成了一种负担。

几乎是从对 Blit 的优化过程中，我重拾对汇编的兴趣。那无非是一段数据传输的函数，用汇编写也篇幅甚短，我尝试了用 MMX 做优化，发现可以提高几个百分点的速度。现在还是能回忆起当时发现这点后的兴奋。年少的我曾经在那段日子里不停地享受着追逐 CPU 速度极限的兴奋。

关于数据块的复制

C 的标准库中提供了一对函数：memcpy 和 memmove。通常我们会使用前者（后一个是改良版本，会判断源数据区和目的区域是否有重叠，然后根据具体情况选择复制的次序）。不少初学者在刚刚接触汇编优化时，都兴奋地想用自己学会的新知识 CPU 的新指令重新写一个自己的 memcpy 版本改进速度。

可惜的是，大多数情况下，除了满足一下编码欲，这是一件毫无意义的事情。实际上，C 语言中，在复制小于 64K 的数据时，我们几乎不可能写出一个比标准库提供的 memcpy 还要快的函数。

这是因为，现代编译器为一些核心函数做了编译器级别内嵌处理的版本（不同于简单的 inline），memcpy 这样的函数实质上被看做一种关键字在处理，而不是一个简单的函数调用。本身，对数据复制这种并不复杂的操作，优化的余地就不大，而对于小数据块，用户定义的函数，会把许多的处理消耗在函数调用上。

对于大数据块（集中体现在图片数据）的复制，的确是有一定的优化余地的。一般我们会用 rep movsd 这一条汇编指令完成物理连续的数据复制。到 Pentium 下，使用浮点寄存器指令（FILD/FISTP），可以拓宽一些内存的带宽，甚至还可以循环展开两次（FILD/FILD/FXCH/FISTP/FISTP）来获得更佳效果。

进入 MMX 时代后，MMX 指令是一个新的选择。因为 MMX 寄存器也有 64 位，到 Pentium 3 以后更是有了 128 位寄存器。

下面列出的这段代码展示了如何实现：

```
; esi 源数据
; edi 目的地址
; eax 数据长度/16

        sub edi, esi
        align 16
        loop: movaps xmm0, [esi]
        movaps [esi+edi], xmm0
        add esi, 16
        sub eax, 1
        jnz loop
```

如果是单纯移动数据，而不希望被移动的数据影响 CPU 的内部缓存的话，还可以考虑 MOVNTQ 或是 MOVNTPS，因为这两条指令不影响 cache 的工作。

到了 Pentium 4 以后，还多了一个选择：使用 MOVDQA。

不过，不要对这些新的 CPU 带来的新方案抱有太大的期望。大块内存的数据复制速度极限终究被限制在内存总线的数据上。连续数据从内存条加载到 CPU 中被处理，是远比不上 CPU 的连续处理速度的，这在流水线设计的 CPU 上尤其明显。

8.2 脏矩形优化

2000 年底，以前的老朋友余雪松和吴东黎见我在北京无所事事（实

际上是先和几个哥们找到了一笔风险投资打算做手机平台的游戏，其后我又觉得做手机游戏索然无味而在签约前个人退出），极力鼓动我去他们公司做一闲职，并答应能容忍我的自由散漫。这家当年只有 6,7 个人的小公司如今已经发展成为上百人的大团队，主力产品是许多网民都知晓的可乐吧（www.kele8.com）。

我进入可乐吧的时候，老余的游戏引擎 FancyBox 已经初具规模。我十分喜爱这种完全脚本驱动的游戏引擎的模式（不过不是很喜欢那种脚本的语法）。在可乐吧的半年中，为公司做的正经事不太多，但宽松的环境让我想了很多问题。

现在的 Windows 环境已经让大家都习惯了多任务的操作环境，所有的 Windows 用户都会同时运行着许多程序，开着即时通讯软件和同事朋友保持在线的联系；启动一个 E-mail 的客户端收取电子邮件；Web 浏览器也一定打开着在互联网上冲浪，随时去 Google 查询资料；许多人还有打开 MP3 播放器听音乐的习惯；最后根据每个人的工作性质不同，还会运行着开发工具或者桌面办公软件。我突然意识到，把游戏程序看做一个特例，企图让游戏程序吃尽所有计算机的资源是一件多么愚蠢的事情，如果专门去玩游戏，还不如买台游戏机呢。

Windows 游戏的玩家会感谢游戏程序员为了提升一点点游戏帧速率而让游戏程序吃掉所有的内存并满满地占据所有 CPU 的行为吗？Windows 平台毕竟和游戏机控制台的平台有所不同，在游戏机上，用户所做的事情仅仅是同时玩一个游戏，而 Windows 下，用户期望同时做的事情更多。

Windows 下的游戏和别的应用软件并无本质的不同，我们也应该去遵循 Windows 下编写软件的约定，尽可能地让自己和别的应用程序和谐流畅地一起工作。

可乐吧的游戏作为网络休闲游戏更是如此，最初，可乐吧的引擎是作为一个 ActiveX 控件内嵌在 IE 浏览器中的，用户玩游戏就同浏览一个网页一样。尤其对于非第一人称 3D 游戏，更高的帧速率是没有意义的，稳定的帧速率下，更小的 CPU 占用率才会博得用户的欢心。

早期的 FancyBox CPU 占用率很高，经过测试，发现 40% 以上的

时间被花在了最单调的操作——从屏幕后台缓存翻转画面到显示的过程。由于可乐吧的游戏工作在窗口模式下，所以用户的桌面分辨率的不同，用户可能采取的最大分辨率也有所不同。如果窗口大小是 1024×768，采用 16 位高彩模式下，整个窗口面积的数据量达到惊人的 1.5M。

1.5M 的数据被移动，相当于 3M 的数据访问，从内存读入 1.5M，传送到显卡 1.5M 去，这是一个相当漫长的过程。似乎无论怎样优化这段代码，也不会有质的提高了。

我不禁想到 Windows 的 GDI，为什么 Windows 本身允许那么高的分辨率，却不是太在意机器的速度呢？那是因为 Windows 的显示模块并没有时时刻刻处理整个屏幕。

该联想到古老的图形优化方法“脏矩形”了。

如图 8-2，第一帧，屏幕中有一个圆形的图形和一个三角；第二帧，这个圆向右下移动了一些，而三角形没有移动。当我们已知上一帧的图像还留在屏幕上时，下一帧刷新图像数据的时候，其实只需要刷新虚线沟出的区域（见第三张叠加的示意图）。现在这个区域是用矩形的形状给出的，我们认为这个矩形区域被“弄脏”了，需要刷新，这个矩形就被称为脏矩形。

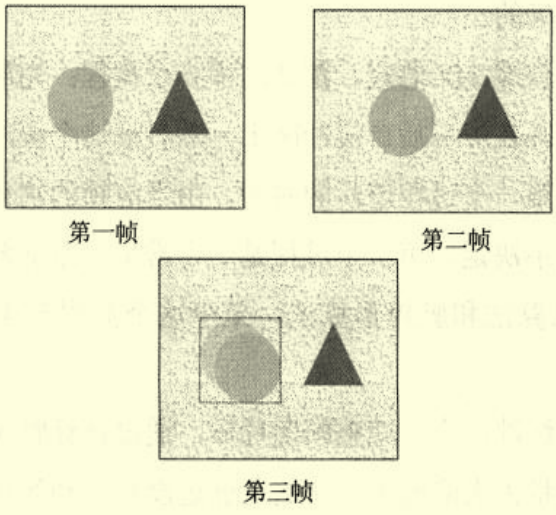


图 8-2　“脏矩形”示意

当然，在游戏的画面中，运动的画面不是一个圆形图块这么简单。游戏每帧画面都有很多的运动物体，每个物体制造了一个脏矩形，我们

就需要把所有的脏区域合并起来，在更新画面的时候集中处理。

之所以用矩形描述“脏”的区域是因为需要处理简单，如果处理的复杂度导致处理时间超过了刷新整个屏幕数据的时间，就得不偿失了。Windows 的 GDI 就是用脏矩形的方式工作的，所以即使桌面的分辨率再高，因为需要改动的区域并不多，GDI 依旧可以保证很高的效率。

让一个图形移动时，我们想到的最直接的过程是从屏幕上擦掉这个图形，然后在新的地方画出来。这仅仅在背景是纯色或者简单有规律的花纹的情况可以实现。游戏中，直接擦掉一个图片简直是不可能的事情。从这个面对显存的层次上，我们无法知道一个图形底下曾经覆盖的图像是怎么样的。

Windows 是怎样工作的呢？Windows 基于窗口对象来管理显示的桌面。它很容易知道脏矩形区域内哪些窗口需要对它进行负责，然后给这些窗口发消息让它们工作，重新绘制这块区域。我们的游戏图像引擎也一样，应该工作在可绘制的对象一级才能控制这些，使用画家算法来保证其正确性。

所谓画家算法，指的是画家在做画的时候，往往由远及近地创作。即使远处的物体被近处的物体挡住了一部分，也是先把它全部画出来。由于近处的物体后画上去，但是由于颜料会覆盖到已创作的图上，最终作品的观感是正确的。

据我所知的许多 2D 游戏，都是这样一个流程：先用画家算法把当前帧的所有物体画在屏幕后台缓冲区上，然后把这个缓冲区翻转到屏幕显示给玩家看。第一个过程消耗的时间，由当前帧的物体的数量以及每个物体图片的大小决定，第二个过程是一个常量，跟屏幕分辨率有关。而如果结合画家算法和脏矩形算法，至少这个流程的第二个步骤就会降低。

我们在后台缓冲区产生完整帧图片后，通过计算脏矩形，使需要翻转到屏幕的数据量大大降低了。而这还远远没有达到优化的目的，采用另一个简单的方法同样可以大大压缩第一个步骤的时间。那就是前面已经提到的，在对象一级制作图像引擎做一个待绘制对象的队列，在绘图时，先只把对象入队列而不真正做 Blit 处理，只合并脏矩形区域，然后

在全部对象都进入队列后，可以剔除脏矩形之外的和上一帧相比并无改变的对象。

脏矩形的原理非常简单，而且很多年以前就被人提出过，但每每一经提起，就会被游戏程序员们嗤之以鼻。理由无外乎几点：

（1）脏矩形的合并算法并不简单。如果要处理的图像太多，同样是一个相当费时的操作。

（2）合并后的脏区域并不是一个矩形，不方便做图片裁减。这样，为了用画家算法绘制把矩形区域弄脏的图片背景时，背景上原本静止不变的图片的重新绘制会扩大脏矩形的面积。

随着游戏中运动物体的增加，脏矩形区域很轻易地能扩大到整个屏幕，从而呈现比没有用脏矩形更低的效率。

（3）即使能很好地解决第2点，比如在Blit一级加强剪裁的能力，把脏矩形扩大的情况减少。一个动感十足的游戏依旧有许多的物体在屏幕上移动。

（4）没有卷动屏幕的游戏不多，而一旦整个屏幕都发生卷动，脏矩形优化就无效了。

（5）增加了图像引擎设计的复杂度，原本图像引擎只需要支持图片Blit操作，而现在却要在对象而不是图片一级工作。

对于前两点，传统的脏矩形算法是有很多问题，这或许是脏矩形算法一直难登大雅之堂的直接原因。对于此，我们在下文中会看到我使用自己的方法来解决。

第3点完全是一个错觉。不错，大多数的游戏都在屏幕上有许多运动的物体，但是，并非所有的东西都以很快的速度在运动。有的物体一秒只运动十次八次。运动快的物体也在三十帧之下，或许只有鼠标需要保持很高的帧速率才能让玩家觉得流畅（我的实际经验，四十帧的鼠标运动已经显得足够流畅了）。

那么，假如游戏的画面固定在一个帧率下，其实每帧画面相对上一帧改变的物体数量极其有限。当我们让屏幕上充满着走路的小人时，只要人的行走不是那么快，人物行走运动的帧数只有游戏帧率的一半的情况下，完全可以做到奇数帧和偶数帧各一半的小人交错运动，而玩家根

本无法察觉。

实际上，一个动感十足的游戏画面，每帧变动的画面还是能保持在整个显示区域的一半以下。不要小看这一半的数据，这意味着处理图片数据量只有原来的一半了，和引擎算法复杂度的增加多消耗的时间根本不在同一数量级。

第 4 点看起来很有道理，但只是看起来而已。其实，大部分需要滚屏的游戏，用户真正在滚屏的时候并没有程序员想当然感觉得那么多。看看游戏进行的过程，实际上，超过一半的时间是没有屏幕滚动的。对于玩家玩游戏的视觉感受来说，最舒适的还是静止的背景。

在 Windows 平台下，我们需要的是多个程序处于一种和谐共存的状态。在屏幕静止时，释放出来的 CPU 资源对其他进程也是非常有益的。尤其对于网络游戏，玩家甚至可能同时打开两个以上的游戏窗口，两个游戏通常是交互滚动，而非同时。一般用户没有操作的时刻，屏幕静止的情况也居多。

对于一些可能在后台工作的非游戏软件，CPU 资源的消耗并不和用户的交互保持一个均衡的水平，它们完成整个事务的过程不需要用帧速率来衡量而是一个 CPU 时间的总量。这样，游戏能适时释放出多余的 CPU 时间，这些程序也可以更快地结束他们需要处理的事务，对于用户来说，这是一件乐于看到的事情。

总之，在一个时间段内尽量少占用系统的资源，是优秀的 Windows 程序的模范（事实上，用一些技巧，我们可以优化掉滚动屏幕造成的全屏幕全部刷新，这个方法同样会在下文提及，引擎设计得好的话，屏幕滚动几乎不会影响速度）。

最后一点，第 5 点，简直不是一个反面理由。这也是诸多游戏程序员的编程水平一直被指责停留在 cracker 一级的原因。

其实，我们让图像引擎的管理层次高于直接处理静态的图片数据，效率上最显而易见的好处是，一旦游戏窗口被最小化，图像无需显示，大量的图像操作都可以在引擎的级别被方便地忽略掉，而只需要让逻辑层工作。

2000 年底，当我反复考虑如何提高图像引擎的速度时，反复萦绕在

脑子中的是，既然现有的做法把大部分的 CPU 时间都花在了最单调的大量数据复制上，我们能做的不应该是想办法优化数据复制的函数，而是应该减少无谓的数据处理的数量。那么脏矩形的方法应该是最好的选择，既然排除掉以上五点不采用脏矩形算法设计图像引擎的理由中的后三点非技术原因，对前两点技术问题的解决就成了当务之急。

太多的游戏程序员过于注重算法实现的细节和一些和底层密切相关的知识，而忽略了游戏程序也是一种软件的事实，不太愿意在更高的角度考虑软件构架的问题。逃避更为复杂的设计，对实现复杂的结构可能导致代码的错误，以及对错误显露后的调试复杂度的恐惧感，影响了自身的水平的提高。出于酸葡萄心理，有些人开始指责复杂的设计带来了效率下降，这就像指责 C++降低了原有 C 语言的效率一样的荒谬。

8.2.1 “风魂”中改进的脏矩形算法

在我的个人主页上公开源码的“风魂++”，留到如今依旧保有参考价值的部分就是提供了一套通用的对 2D 游戏的脏矩形解决方案。由于当年的代码写得非常草率，而后重新编写的代码由于设计版权问题又不便公开，这里就不贴出具体代码献丑，而是讲解一下实现原理，相信每个有编程经验的程序员都可以自己实现之。

注：公开源码的“风魂”版本由于非常古老，而这些年里我又对一些细节进行了重新设计，所以以下谈到的解决方案并不完全符合可以在我的主页上下载的程序代码中所展现的内容。

首先，用户所有的绘图操作都是作用于一个叫做 canvas（画布）的东西。而每个可绘制的物件都必须是一个对象，而不是一个个相异的绘图指令。

每个对象由自己的方法来决定如何把自己绘制到指定的画布上，包括 canvas 本身也是一个这样的可绘制对象，可以支持整个画布画到另一个画布上。

这些对象设计时有一个共有的基类提供一些接口供画布调用。这样，图片也就理所当然地成为这种对象。而让许多游戏程序员不习惯的是，画一条直线或者一个矩形，又或一串文字也必须做成一个对象才可以被绘制。在不少图像引擎的设计中，并没有将几何作图和文字书写和图片 Blit 操作统一起来。字符串与直线线条跟图片及精灵是同一类对象，应该有相同的基类。这是许多游戏程序员需要转变观念的一个小地方。

其次，所有的绘图操作并非立刻被执行，而是被送入canvas内的渲染管道，这个步骤并不做任何的图像数据复制和传输的操作。直到最后，需要调用canvas的刷新操作才使所有渲染管道中记录的流程被操作。这一点要求canvas的表现必须是以完整的一帧画面被产生。基于效率考虑，渲染管道内并非存放对象的实体，而仅仅是一个对象指针，甚至不是一个智能指针。这样在canvas没有被刷新之前，所有进入渲染管道的对象不能被释放，这隐式要求了禁止临时构造一个对象，比如一条直线的对象送入绘图管道。

这个设计会使许多图像引擎的二次开发人员感觉不那么舒服，毕竟异步的绘图操作不太方便调试，扩展起来也不如直接对一个显存或是系统缓存的地址指针直接操作起来那么方便直接，这或许就是增加引擎设计复杂度的一个代价。

对于使用引擎的程序员，禁止使用临时绘图对象也是一种制约，好在解决方案很多，超出了这里需要讨论的范围，感兴趣的读者自然会对此思考。

然后，有一个看似奇怪的设定，那就是，在“风魂”的设计中，绘图的过程是采用的和画家算法相逆的次序，即上面的东西最先送入渲染管道，而下面的诸如背景那样的物体最后进入。这点被我的许多朋友也指责过，认为违背了游戏程序员的常识。其实这只是一个观念问题，不影响开发，而这种设计也不是我喜欢反常规行事，而是为了另一个关于优化方案的实现，这个方案在后文会被提及。

实际上，在现在的3D显卡的API上，无论是OpenGL还是Direct3D，都建议程序员从前至后而不是从后向前地送入三角形，这是因为前面的三角形可能覆盖住屏幕的许多区域，导致后面再送入的三角形根本不需要被渲染。“风魂”采用逆向的画家算法也是基于同样的考虑。

接下来我们来看canvas的设计。

整张画布被划分成网格状。在“风魂”中，每个格子的大小为64×32。格子的大小是可以调整的，但是长宽都必须保持8的倍数以保证处理效率。由于图像缓冲区是以多条水平扫描线的形式保存，处理图像数据的时候，应该尽量保证水平方向的连续性，故而格子的宽度比高度略长为佳（如图8-3所示）。

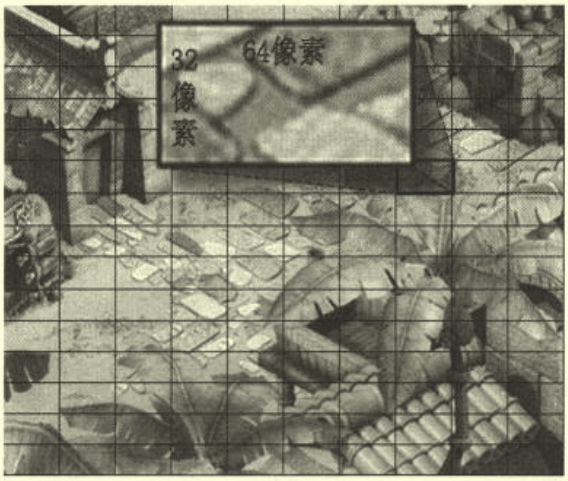


图 8-3 格子的宽度比高度略长

改良的脏矩形优化算法以固定的分格为单位，而不再是用合并任意大小位置矩形的算法。这样可以将脏矩形总面积维持在一定的范围内，并且让合并后的脏区域的形状比较规则。

每个对象被画到 canvas 上后，会由对象自己决定弄脏哪些格子，并将这些格子做上记号。大多数情况下，绘图对象有自己的外包围矩形，用这个矩形来判定自身涉及到哪些 canvas 网格就够了。在少数情况下，例如斜线对象，只用外包围矩形判定弄脏的网格会牵扯过多实际并无牵连的地方，这个时候应该做特殊处理。

如果让 canvas 自身保存上一帧所有对象的状态并且和当前帧比较，则计算负担很大，所以应该由每个对象自身记录自己的变化状态，并在绘制的时候通知 canvas。

通常，一个对象第一次绘制到 canvas 上时必然相对前一帧（并无这个对象）有所改变，对于它涉及的区域就可以设置为脏的状态。对象的移动，以及对象自身的图像变化，即缩放、旋转、动画帧切换，甚至对象相互之间的前后变化也会引起 canvas 区域弄脏。

唯一一种对象状态变化很难让对象自身通知到canvas 的是对象的消失。处理这个问题有很多个方案，在“风魂”中采用的是，每个 canvas 网格自身记录每帧画在其上的对象数量。如果数量变化，则同样也证明了当前网格被弄脏。如果在同一帧，同时有一个对象出现，一个消失，虽然数量没有变化，但新出现的对象会负担起对网格做标记的责任。

最后，在 canvas 刷新的操作时，依次查询每个网格是否弄脏。如果

是，则按渲染队列中的次序，把所有该画在此网格中的对象绘制出来。因为一个对象可能远比单个网格要大得多，每个绘图对象都应该支持剪裁，允许把自身的一部分正确地画到一个矩形位图中。当所有弄脏的网格都被处理完后，再把所有这些脏了的网格缓冲区图像数据传送到屏幕（过程示意如图 8-4 至 8-7 所示）。

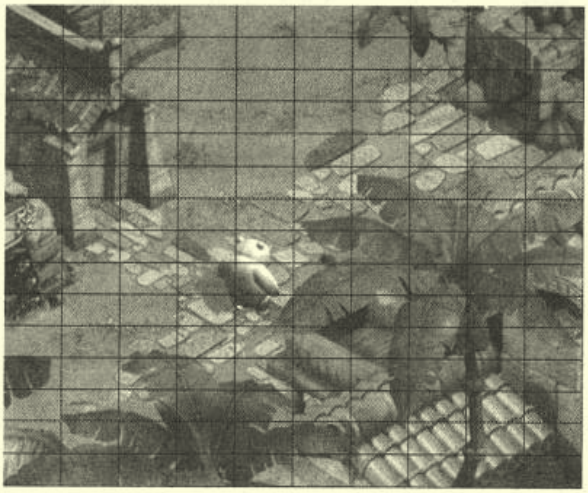


图 8-4　第一幅图

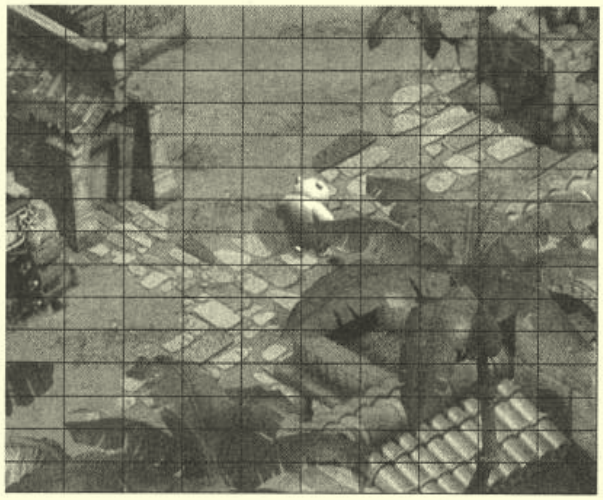


图 8-5　小熊走动

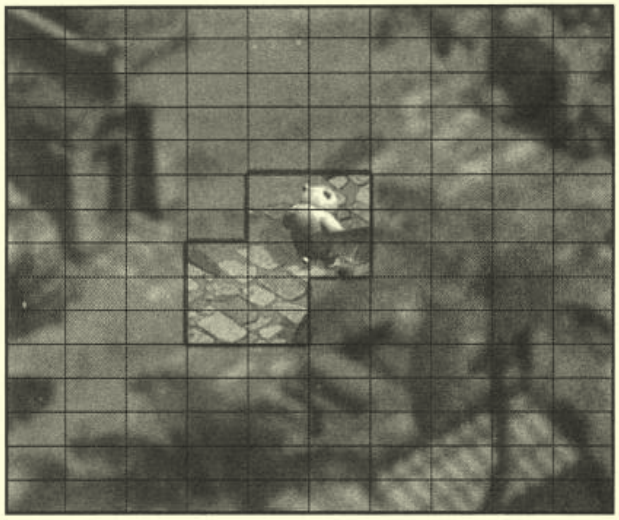


图 8-6　变化区域



图 8-7　变化层次

8.2.2　滚动优化

下一步我们来解决滚动引起的整个 canvas 都发生变化的问题。

首先应该为canvas 创建一个比实际需要显示的面积大一圈的后台缓冲区，如图 8-8 所示。

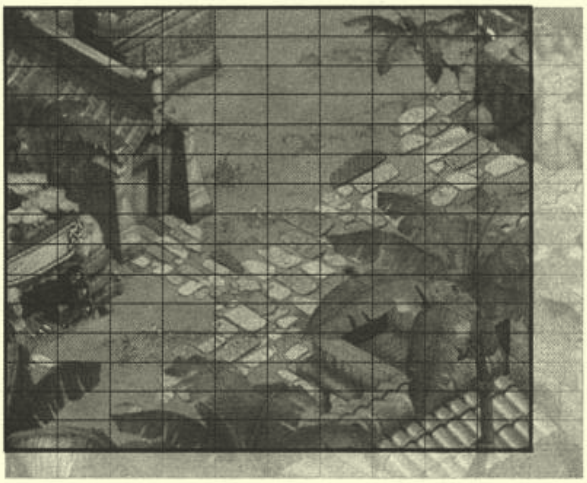


图 8-8　黑框为屏幕显示部分

这样，如果屏幕向右或向下做轻微的滚动，只需要调整缓冲区向屏幕映射的起始坐标，范围在一个网格之内，就可以不需要更新全部 canvas 的网格而做到滚动了（如图 8-9 所示）。

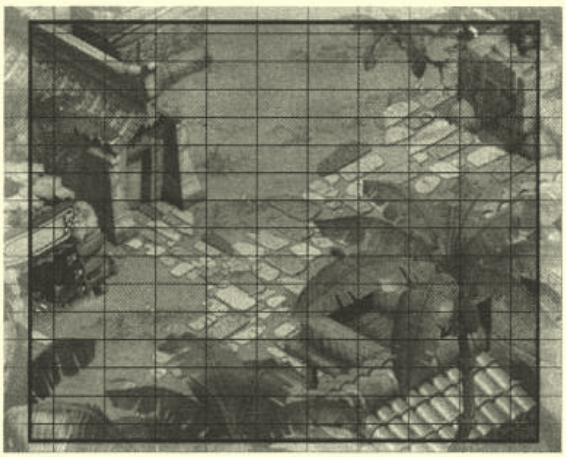


图 8-9　不更新全部 canvas 的网格而做到滚动

在滚动超过一个网格，或者向上/向左超过了 canvas 缓冲区原有的范围时，这个时候我们应该重新组合这些网格，尽量地保证更多的格子映射到滚动前的网格上，让这些网格上的图像数据不被弄脏，这样，网格到屏幕影射关系可能会被改变，因为我们在处理 canvas 刷新的时候是以网格的格子为单位处理的，这也并没有影响前面脏矩形算法的设计。

虽然由于网格和屏幕的对应关系被破坏了，在传输缓冲区的数据到屏幕时需要整个缓冲区全部重新 Blit，但没有被弄脏的格子的缓冲区数据自身却不需要重新构建出来（如图 8-10 所示）。

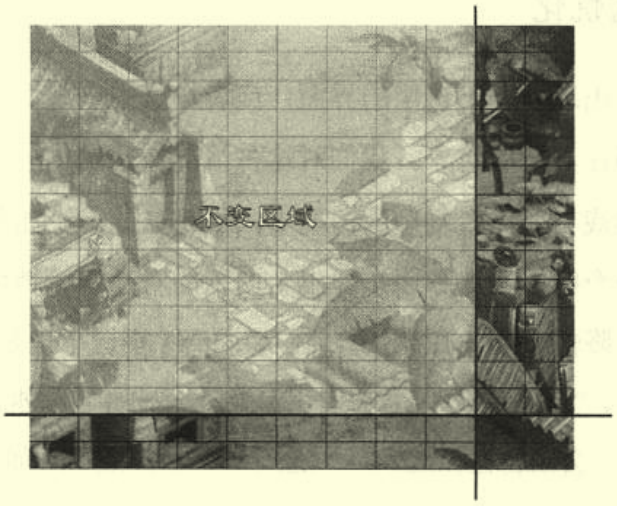


图 8-10　不变区域

这里，我们还可以利用显存间的数据可以快速传输这个特性对上面的过程进行加速。如果玩家机器配备了多余的显存，我们可以在系统内存上的 canvas 缓冲区之余，在显存上同样创建一块等大的镜像缓冲区。每次并不直接把系统内存的 canvas 缓冲区数据传输到屏幕，而是先分格加载到离屏缓冲区内，再而从离屏显存缓冲区通过显卡自身传输图像数据到屏幕。

后一过程通常可以工作得很快。当屏幕滚动时，canvas 分格的数据未弄脏区域和离屏显存中的镜像缓冲区上的对应关系并未受其影响，所以无须重新从内存上载到显存（如图 8-11 所示）。这样，滚动屏幕几乎不会影响整体的效率了。

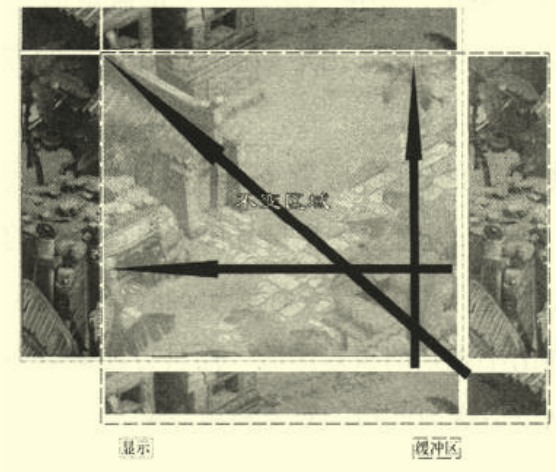


图 8-11　对应关系

8.2.3 覆盖优化

当游戏中出现大块不透明的界面，或者场景中出现大块的建筑时，都有可能遮挡住其后的对象。我们对这种情况做出优化也具有一定的意义，尤其在游戏界面比较复杂的情况下，更是如此。正如前文所述，即使让游戏在一个时段（比如用户弹出大面积的对话框遮挡住游戏场景做操作的时刻）降低 CPU 的占用率，在 Windows 平台下也是有意义的。

在这点上，“风魂”做得简洁实用。当一个物体自身拥有一个比较大的内切矩形时，会由自己把这个矩形送到 canvas，canvas 如果发现矩形完全遮挡了某些网格，这帧其后画向这个网格的对象都会被忽略而不会进入渲染管道（如图 8-12 所示）。



图 8-12　内切矩形和覆盖块

如果想完善这个系统，还应该解决一下覆盖矩形合并的问题。在“风魂”其后的版本中，淘汰了用矩形描述覆盖区域的方法而改用水平扫描线。

根据 canvas 缓冲区的高度，设置了相应数量的扫描线覆盖状态记录的对象（如图 8-13 所示）。对象在把自身传送到 canvas 的渲染管道的同时，根据需要，会以水平线为单位逐行通知 canvas 自己至少覆盖了哪些区域，这可以描述出不规则的覆盖区域，而且方便做合并。当一个网格完全被覆盖住，则以后送向此格的对象将被忽略。

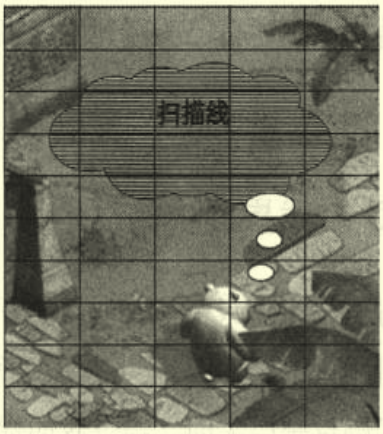


图 8-13　扫描线覆盖状态记录的对象

后一个改进方案，程序算法以及数据结构的设计稍微麻烦一些。“风魂”中的做法是以一个整数数组来保存一条扫描线的覆盖状态。由于水平扫描线一定是若干段未覆盖和被覆盖的线段交替衔接构成，数组中每个数字就表示了每段线段的起点。奇数位表示了未被覆盖的线段起点，偶数位表示了被覆盖线段起点。这个数组中的数据一定是永远有序的，所以可以方便地用对分查找查询，并且可以利用数组中的位置信息得到一段线段是否被覆盖（如图 8-14 所示）。

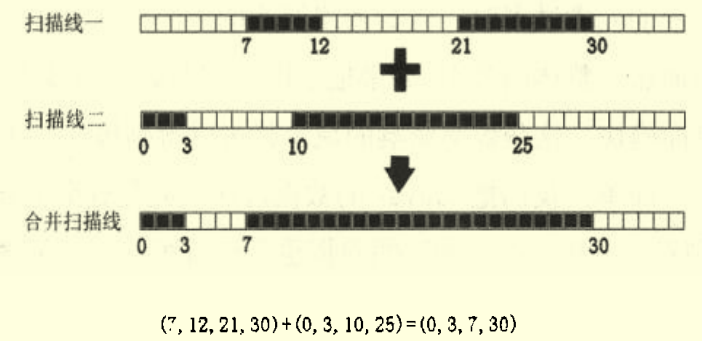


图 8-14　合并扫描线

每个网格有一个累加器，当某条水平扫描线穿过网格，并将网格所属区域完全覆盖住时，累加器加一。一旦累加器累加到网格高度的像素时，则表明整个网格被覆盖。这种设计，只需要算法实现得好，速度可以非常得快。

不过，建议对小物体不要去设置覆盖扫描线，因为覆盖算法会因为覆盖扫描线段的增加而导致处理的时间增加，反而得不偿失。同样，每

个对象也不需要非常精确地描述自己所覆盖的区域，仅需要描述至少会覆盖的区域即可。

8.2.4 脏矩形在 3D 游戏界面中的运用

最近两年，我参与了公司内两个 3D 项目的开发。无论是 3D 还是 2D 游戏，都少不了图形界面的模块。莫要小看界面模块，在游戏，尤其是网络游戏中，它的代码量是巨大的。经过我的提议，决定把公司已有 2D 游戏中的界面模块完全移植到 3D 项目中去使用，事实证明，此举极大地缩短了项目开发的工期，并减少了新写代码带来的错误隐患。而且，在某些方面获得了比传统的 3D 游戏界面模块更高的运行效率，并节省了宝贵的显存资源。

传统的 3D 游戏图像引擎的设计，界面元素都是以单独的贴图保存起来的，这样，每个按钮、对话框，甚至每个不同的汉字都会使用一些贴图。这不仅占用了大量的显存资源，对于文字，甚至还需要做一些贴图缓存管理（汉字的数量比西文字母要多出一个数量级，往往并不适合全部加载到显存中）。

当界面上的元素过多时，比如网络游戏的大面积聊天信息，还会因为绘制界面而影响整体的效率。如果把上面提到的脏矩形技术应用在 3D 游戏中的界面模块，这些效率影响的现象就可以得到极大的缓解。

做法非常简单。我们把 canvas 的数据缓区还是创建到系统内存中，在 canvas 刷新时，并非把缓冲区的数据更新到屏幕，而是传输到整张贴图上（由于 3D 游戏贴图的大小往往受到显卡硬件的限制，我们可以创建多张贴图对应整个 canvas）。

游戏界面的更新频率实际上不需要和 3D 游戏的游戏场景刷新频率保持一致，这样，我们可以以固定的低频率（例如，30 帧）更新显存贴图上的 canvas 缓冲区镜像，而以和游戏场景相同的刷新频率绘制这些贴图。由于 3D 游戏的刷新频率通常会比较高，这种让界面以较低频率刷新的方案使得非常复杂的游戏界面也不会影响游戏整体的效率了，而且界面模块的显存占用数量也是恒定的，其需求量几乎所有的 3D 显卡都可以满足要求。图 8-15 表示一个 3D 场景的游戏上面有一些平面界面元素。

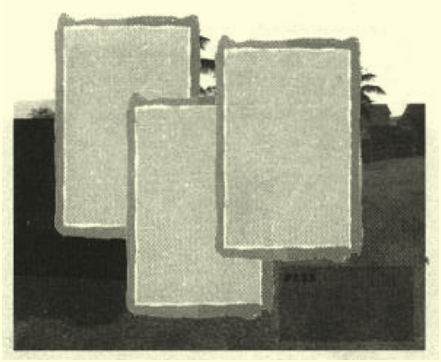


图 8-15　一个场景

不过，和 2D 游戏不同，canvas 的缓冲区数据不能再以 RGB565 的高彩像素点格式存在。这是因为，canvas 变成了游戏画面的一部分，而非全部。为了正常地显示，我们需要采用 RGBA4444 的像素格式描述 canvas 图像缓冲区。这样就可以正确地让界面的一部分透明或半透明露出其下的游戏场景。

如果界面的全部或大部分都没有物件的时候，我们还可以根据 canvas 的网格决定一个大区域不需要绘制，减少显卡的像素填充数量。这些细节，在真正引擎设计的时候应该被全面考虑。

图 8-16 表示带 Alpha 通道的界面层，一般可以用 RGBA4444 格式的贴图来作为 canvas 的映射载体。

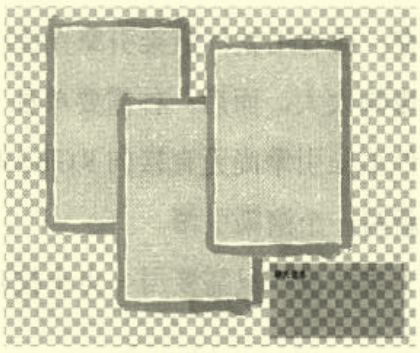


图 8-16　带 Alpha 通道的界面层

1. 界面中 3D 物件的处理方法

如果用 2D 方式处理 3D 游戏的界面，最大困难在于，界面中可能有些物件必须用 3D 渲染的方式处理。比如，前面图中的一个对话框上希望绘制一个 3D 渲染出来的角色模型（如图 8-17 所示）。

关于 RGBA4444 模式

或许有人担心 RGBA4444 像素格式，由于每个颜色通道只有 16 级深度，表现的画面会很难看，失去了很多颜色过渡。看看许多所谓 4096 色的手机，屏幕上难看的画面就是一个佐证，这其实是不懂得图像处理技巧的后果。

如果我们直接把 RGB888 的真彩图片的每个颜色通道截断后四位转换成 4096 色图片，自然会失去丰富的颜色层次，所以必须进行 dither（抖动）处理，通常的算法是误差扩散（error diffusion）。很可惜，诸多流行的图像处理软件，比如 Adobe Photoshop，都没有提供把 24bit 真彩图片抖动成 4096 色的功能。我们程序员就需要自己写一个工具交给游戏界面的美术人员使用，让他们把做好的界面图片抖动转换成 RGBA4444 的格式。



图 8-17　绘制 3D 渲染出来的角色模型

单独拿出来看，这个对话框如图 8-18 所示。



图 8-18　角色模型

因为 2D 界面处理的时候，canvas 的实际 buffer 是映射在内存中的，（否则就失去了这样做的意义），最直接能想出来的方法是，用显卡渲染出模型后，复制回内存；或者用软件渲染引擎直接在内存中渲染。

前种方法对性能影响太大，而后一种需要单独写一个渲染引擎，现在看起来，成型的软件渲染引擎尚无直接向 RGBA4444 的表面渲染的，为这个小小的应用重新写一个渲染引擎。

其实这个问题并不难解决。只需要用一点小技巧。

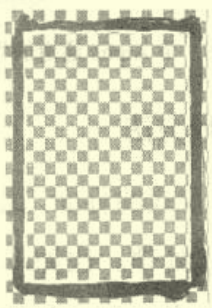


图 8-19　中间区域的 Alpha 为 0

注意图 8-19，中间的区域的 Alpha 为 0，是完全镂空的。在 canvas 的层次处理这张图片的时候，不能按 Alpha 混合处理这张图，而需要按 Alpha 值直接覆盖中心的区域。最后的效果如图 8-20。

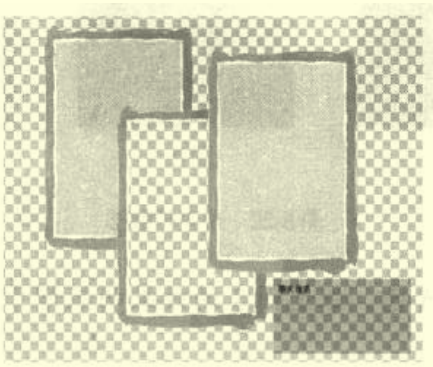


图 8-20　最后效果图

在图 8-20 上，我们可以看见，中间的对话框将左边一个对话框盖住，而中间被正确地镂空。

而 3D 模型，则应该和游戏画面一起渲染，如图 8-21。



图 8-21　3D 模型和游戏画面

当我们将界面层叠加在游戏画面上时，就得到了正确的效果，如图 8-17 所示。

误差扩散

误差扩散法（Error Diffusion）在数据处理中经常碰到，它还有另外一种称法，叫做抖动（dithering），尤其是用于图像处理中，降低色彩的深度。图 8-22 中三张图，图 1 是 256 级灰度过度，图 2 是简单地将图 1 按就近原则，转换成了 16 级灰度（直接把 8bit 灰度的后 4bit 砍掉），图

3 是将图 1 用误差扩散的方法转换成 16 级灰度。


图 1


图 2


图 3

图 8-22　误差扩散法

很明显，同是 16 色，但是图 3 的效果明显好于图 2。其原因是使用的误差扩散算法。

顾名思义，误差扩散，就是将色彩深度降低时，将像素颜色的变化误差，扩散开去。这使得肉眼在观察图片的时候，相邻的像素点集合整体的误差变小。下面举一例来说明要容易理解得多。

在原 256 级灰度图像上有一个点，灰度值是 120 (0~255)，如果我们要将这张图转换成 16 级灰度，最简单的方法是将每个像素点除以 16，那么转换后的值就是 120/16=7.5，保留整数位后就是 7。这样转换后的值就有了 0.5 的误差。最简单的误差扩散方法是将这 0.5 的误差放到这个点右边/下边的点上，我们可以按 3:2:3 的比例把它分配到右边，右下，下边的点上。即，我们把右边和下边的点加上 (0.5*16)*3/8=3，把右下的点加上 (0.5*16)*2/8=2。这样处理完整个图形，要比直接截尾的效果要好得多。实际上，3:2:3 的误差分配方案并不算好。你还可以试试 7:3:5:1 的分法。

				X 7
				3 5 1

或者 42 分：

			X	8	4
2	4	8	4	2	
1	2	4	2	1	

当然也不一定是复杂的 Filter 就一定效果更好，还要视图像而定。

我们可以很容易地把这个方法扩展到彩色图像的处理（只需要将颜色分成 RGB 三色素），如果 3D 游戏中使用 RGBA4444 的贴图色彩格式，真彩色的源图就一定要做误差扩散的处理。

RGBA4444 的色彩格式被很多游戏制作者误解，认为它只能提供非常低画质的贴图，因为每个颜色通道只能提供 16 级的色阶。实际效果也似乎证实了这点。不过原因正在于现有的常用工具（例如，Photoshop）没有提供抖动图片到 RGBA4444 的功能，很多程序员采用了偷懒的做法，直接截掉真彩色的后几位的做法来转换，那样的效果和做了抖动的效果完全不同，可以参考上面图 8-22 中的图 2 和图 3 做一个比较。

另外，不光是图像处理，这个算法在别的一些数据处理中也常用到。

8.2.5　时间和空间的权衡

从这些图像引擎中时间效率问题的讨论可以看到，算法的优化带来的速度提升效果是巨大的，往往会超过指令级的优化。

找到更好的方法解决问题，和怎样充分利用已经有的硬件环境把好的方法最佳地实现对于游戏程序员同等重要，我们不可以偏重一个方面而忽略另一个。

速度的瓶颈往往在两个方面摇摆，我们应该来回关注算法和算法的实现，让游戏程序能合理地使用用户的硬件资源。

上面讨论了游戏图像引擎中的时间效率问题，另一方面，过多的内存占用也是遭人厌恶的，即使现在 PC 上插的内存条已经数以百兆计，动不动吃干内存的程序同样会干扰用户的正常工作。

空间效率和时间效率同样重要，Windows 平台不同于专用游戏机平台，专用的游戏机平台的内存十分有限而数量固定，当物理内存使用完后，就会发生内存分配失败，迫使程序员考虑如何减少内存的消耗。而 Windows 似乎为开发者提供了无限（其实只有不到 2G）的内存资源，珍惜内存，很容易被 Windows 游戏开发人员忽略。

或许是我自己经历了内存非常小的硬件环境，一直以来非常注意程

序内存使用问题。游戏程序员应该有对这方面的敏锐的触觉。

这，应该从内存使用的估算能力培养起。

- 一张 640×480 的真彩位图占用多少内存？
- 一个平均大小为 50×100 像素的精灵，8 个行动方向，一个动作 12 帧画面。需要多少内存才可以保存下来？
- 一个 100 屏的游戏场景，采用网格信息保存场景的状态，需要消耗多少内存？

等等，这些问题应该在设计程序的时候很自然地浮现到游戏程序员的脑海中，并且做出一个估算。估算的能力表现在程序员的计算速度、准确程度、对问题的前提条件的确认等方面。好的游戏程序员应该在写程序的同时，迅速地提出相关的估算问题，找到问题需要的条件，准确地得到合适精度的答案。

有经验的游戏程序员会发现在 2D 游戏中，最消耗内存的往往不是看似面积巨大的场景，而是场景中活动的各式各样的精灵。

8.2.6 关于估算

估算是程序员必备的能力，估算并不仅限于计算程序的内存占用问题。估算一个算法消耗的时间，估算网络传输时间，估算 I/O 处理的速度，估算不同方案处理不同复杂度数据的优劣都是很重要的。

通过加强估算的训练，将这种技能带进程序设计的方方面面，发展到对事物的数量级有直觉反应的程度，就能展现出一种资深程序员比之新手独有的能力，可以更快地决定事情的可行性而省去很多无谓的劳作。对于游戏程序员，能够更加敏感地知道哪些地方需要优化，而不需要求助于 Profiler。甚至于对项目管理者，估算项目进度所需要的人员和时间也对项目成功与否相当重要。

对估算能力的训练是如此得必要，这里给出一些建议，这也是我时常无意识去做的事情。

（1）面对问题立刻对时间和空间，还有解决问题需要多复杂的代码做出反应。在任务完成后反思自己当初的估算究竟有多少偏差。

（2）准确地弄清问题，并确定会发生有关影响的前提条件。除了在设定条件下做出准确的估算外，还需要估算各种不同条件的发生几率。

（3）确定估算结果的精度需要达到怎样的程度。

（4）把问题分成更为细致的子问题，找到子问题之间的关系，确定哪个子问题最需要被估算。

（5）不要急于达成结论，多做思考。重要的或是没有碰到过的问题不要太相信自己的估算结果，应该用行动来验证。

8.3 自己设计精灵格式

许多游戏图像引擎都有自己定义的精灵格式，这一点在前面关于Allegro 的章节已有提及。不用简单的非压缩位图格式，最大的原因就是为了节约内存。

“风魂”中最常用的精灵格式是经过 RLE 压缩的。Allegro 里就提供了 RLE 精灵的实现，可以供读者参考。“风魂”中为了提高图像的质量，把精灵边缘的 Alpha 通道信息也压缩进去，这样让精灵在绘制时，轮廓可以和背景做 Alpha 混合，不会让玩家看出精灵轮廓的锯齿。采用这种技术，即使游戏画面分辨率并不太高，玩家也不会觉得其画面粗糙。

支持带 Alpha 通道的精灵，传统的方法是用带通道的位图。在硬盘上保存 tga 格式的图片，直接读入就可以用简单的方法显示出来。可是，这样的数据量实在是太大，太浪费了。如果真的以 32bit 一个像素保存精灵的图像数据，游戏中几个活动的精灵就可以吃掉大部分的内存。

“风魂”中使用 256 色模式，以 RLE 压缩方式，并单独记录精灵的带半透明轮廓信息，在内存中保存精灵图片。数据量只及传统方法的零头。

由于 RLE 压缩方式在前面的章节已经有详细介绍，这里我们来讨论另一种图片压缩方式。

一般的精灵图片都非正规的矩形，相对图片的外截矩形，边缘上有很多无效数据，在绘制的时候应该忽略掉的。这些冗余的数据不仅仅是

多占了内存，还影响了绘制的速度，对于脏矩形的算法也有影响。最容易想到的重新组织这些不规则数据的方案是把精灵的图片切分成更小的块。当我们按 16×16 像素分块保存图片时，那么整个块都可以不保存在内存。对于多帧动画图片，假若两帧间有相对的部分，也可以只保留一块图片的数据，这也是很容易实现的。

这样分块保存精灵，数据量可以减少很多，又使得精灵图片更适合脏矩形的工作机理。这个算法很容易实现，比较 RLE 压缩，图片中的像素定位更加容易。

经过分块后，每个块面积不大，一共只有 16×16=256 个像素。这让人联想到可以采用调色盘的方法对数据进行压缩。一共只有 256 个像素，自然不会去用 256 色模式，比较好的选择是 16 色。经过抖动，用 16 种颜色描述 256 个像素是很容易的事情。每个分块有单独的调色盘，比起整张大图使用唯一一个 256 色调色盘的效果更好。整个图块仅需要 32 字节的调色盘（16 个高彩的颜色）+128 字节的图像数据（每个像素 4 bit）一共 160 字节，而直接用高彩保存图块，则需要 512 字节。压缩比稳定在 31.25%。

分块并用独立的调色盘技术做图像数据压缩，看起来是一个很巧妙的方案。这是一个不对称的压缩算法。为 256 个像素找出最佳的 16 色调色盘匹配它们是一个耗时很大的操作，但是解码的速度很快。

这样，如果每个像素采用 24 bit 真彩色，一个分块就能被压缩到 4+4×3=16 字节。而完整保存 16 个像素的真彩点，则需要 48 字节，而且采用 3 字节保存一个像素还有数据对齐问题，处理速度反而下降。

DXT 压缩

我记得当时发现这个方案时非常兴奋，但是不久就发现这并非首创，其实，S3 公司最先提出的 S3TC/DXT 贴图压缩的技术就是采用的同一原理，只不过 DXT 压缩方案中，块分得更小，只有 4×4 像素，而每个块的调色盘更是被缩小到两色（另外插值出两种颜色），只需要两个位就能储存一个像素。

熟悉一下正规的 DXT 压缩格式是有必要的，现在，它们在 3D 游戏中广泛运用。比如“魔兽世界”中使用的后缀为 BLP 的文件，实际就是 DXT 压缩的贴图。

图 8-23 是 DXT1 的格式说明图，每 4×4 像素被编码成 64bit，即 4 个字节。前两个字节是两个 16bit 的调色盘颜色信息。颜色的格式为 5:6:5。

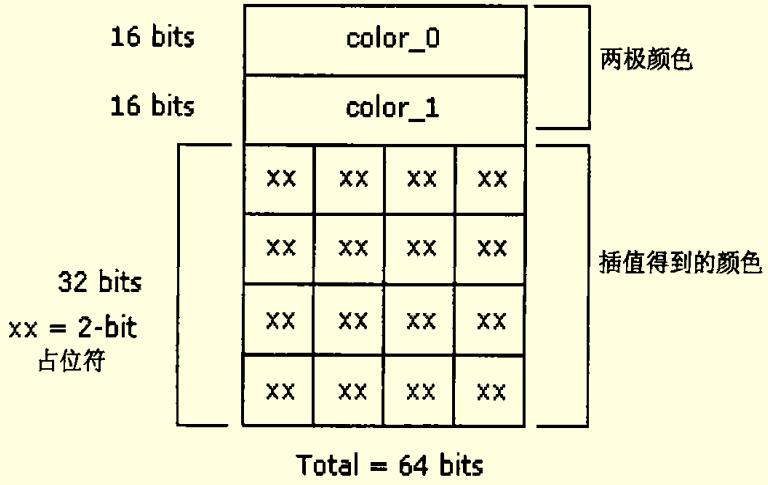


图 8-23　DXT1 的格式说明

如果把两个颜色看成两个无符号整数，那么当 color_0>color_1 时，下面的 16 个像素就可以拥有 4 种颜色。每个像素用 2 个 bit 表示。每个字节的低位到高位排列了四个像素。

```
00 : color_0
01 : color_1
10 : 2/3 color_0 + 1/3 color_1
11 : 1/3 color_0 + 2/3 color_1
```

可以看出，10 和 11（二进制表示）两种颜色是由 color_0 color_1 插值出来的，原始数据中并不存在。

当 color_0<=color_1 时，这个 4×4 的块就表示存在透明色了。透明色是 11，而能够表示颜色的状态只剩下三个：

```
00 : color_0
01 : color_1
10 : 1/2 color_0 + 1/2 color_1
```

DXT 还有别的版本，对于压缩色彩信息的原理是一样的，只是多加

了 Alpha 通道而已，并且对 Alpha 通道的编码方式有所区别。详细资料可以查阅 MSDN。

8.4 动态数据加载

一旦游戏中用到的数据超过一个数量级，就应该引起我们的警惕了。不错，操作系统给了每个进程 2G 的地址空间，我们的程序可以随便使用，如果超过物理内存允许的范围，数据自然会被交换到硬盘上。但是，依赖操作系统做虚拟内存的交换是一件低效的事情。

（1）操作系统做内存数据交换的时候，需要把可以交换出去的数据写到硬盘上，然后再从硬盘上读入期望的数据。实际上，内存中大多数的图片数据根本不需要写盘保存下来。

无论何时，我们扔掉这些数据，接下来在需要使用的时候都可以再从硬盘的数据文件包中读回。甚至，一些数据在硬盘的数据文件中是被压缩过的，重新加载压缩的数据并解码，速度要快于非压缩数据的加载。

（2）许多用过一次，很少再有机会再使用资源数据，如果不主动清理出内存，它会一直占据在那里，当物理内存不够的时候，操作系统并不能结合游戏的实际逻辑情况把最不可能再使用的数据移出物理内存。

（3）操作系统只能基于最后使用时间或者使用频率来推测数据以后的使用情况，这样极有可能在游戏程序的几个模块分别需要大块数据的时候，例如管理游戏场景和管理精灵以及声音流解码，在物理内存很匮乏的时候，互相争夺内存资源，数据被频繁地交换在内存和硬盘，造成硬盘颠簸。

（4）操作系统的内存管理底层是基于分页来管理的，每一页是 4K，而这个数量大于游戏中许多对象的体积。当程序中的小对象分布凌散时，一个被交换到物理内存之外的对象被访问就会引起一个整页的重新加载。所以，管理大块资源数据时，自己做缓冲区管理通常能工作得更流畅。

（5）进程的地址空间是有限的，一个规模庞大的游戏所用到的所有

数据资源的总和有可能超过这个限度。

在我参与开发的网络游戏《大话西游 II》以及《梦幻西游》的客户端程序中，虽然两者的代码完全不同，但都采用了原理相似的资源调度机制。场景图像资源和精灵图像资源被分别放在两个限制大小的内存缓冲区之内。

对于场景，划分成一定大小的图块，连同图像之外的数据都按块为单位存放，根据用户的运动方向，预测可能会被优先使用的图块，采用多线程方式加载。

在场景数据缓冲区被填满之前，所有的数据都不被释放，一旦数据量超过缓冲区，就将最久没使用的数据释放掉。这样保证了这个模块用到的数据不会超过一个限额。

对于精灵，将精灵对象和资源数据分离定义，允许资源数据不在内存中。除非必须要绘制出来，才强迫读入内存。然后把数据分为三类，用三个链表管理起来。

（1）本帧内强制放在内存中，即已经进入渲染管道而尚未渲染的。

（2）最近可能被使用，但允许资源不在内存中的。

（3）对象已经被释放，但是缓冲区大小允许数据暂时不清出内存的。

这样，我们把当前要用的精灵全部放在第 2 条链中，一旦进入渲染流程，就把进入渲染管道的精灵移到第一条链表，如果资源数据没有被加载则强制加载。等 canvas 刷新后，再将 1 号链表合并进第 2 链。一旦可以删除某个精灵，它的图片资源则移动到第三条链。

每次创建新的精灵时，我们应该从内存中寻找是否有已被加载的资源，已免重复加载。而另一方面，则定期检查缓冲区是否超出预定的容量，超过后，则可以删除整个第 3 条链上的资源数据。当删除完依旧缓冲区溢出，还可以依次删除 2 号链中的精灵所附的资源数据。

这项数据加载和缓存的工作并不复杂，但很容易被游戏程序员忽略。一类人没有意识到缓存用过而又清除的数据并不重要，以为操作系统的磁盘缓存能够做得很好；另一类人，对于资源采用只加载不释放的策略，完全依赖操作系统做的虚拟内存交换。这都是不正确的。

作为游戏程序的设计者，你要比操作系统更清楚程序对资源数据的

需求情况。

8.5 小结

设计一个好的游戏软件，只求其正常工作是不够的。多数时候，能想到更好的解决方案，这需要我们不断地思考，本章仅作抛砖引玉，希望能启发读者更多的灵感。

C 和 C++

C++ 是我这几年的最爱。如果程序员没有陷入工程项目的泥沼，单纯做理论学习研究的话，我相信很多人都会爱上博大精深的 C++ 语言。

C++ 被程序员圈子最大的误解就是效率低过 C 语言。这里，我想为 C++ 再次平反，已经有好多人干过这事，我多做一次也不为过。我想证明的是，至少在 PC 平台上，好的 C++ 程序不仅不会比同样的 C 语言实现慢，甚至会略快一些。

当然，我选择 C++ 不仅仅因为它的高效，还在于 C++ 语言的优雅。这些是许多 C 语言程序员没有体验过的。对于 C++ 程序员的圈子，这一章的相关内容显得有些浅薄，即使让我专门写 C++，也能再写出一本书来。但对于游戏圈子，我看到许多老的程序员成天埋头在项目中，没有精力关注学术界的变化和这些年来 C++ 的发展。他们仅仅把 C++ 当成一个带类的 C 来用，甚至还用得不那么规范，这让我觉得有必要花一些语言来阐述我对 C++ 的些许看法。

这一章重点介绍的两个枝节问题，一是内存的管理，二是 template 的使用。这些都是广泛使用 C++ 后一定会遇到的问题。

我学习C++始于1992年，和学习C语言是在同一年。那个时候在上初中，学校里懂C语言的同学本就不多，而从中学老师到父亲都是C语言的门外汉，学校书架上的一本Turbo C++的手册是唯一的启蒙读物。我第一眼看到C++就被其面向对象的思想所吸引，每读完一章，就迫不及待地找到同学分享获得新知识的快乐。

可惜在那个年代，对C++的实现是一无所知，根本无法理解面向对象编程是怎样实现的。现在反思，在没有工程经验的情况下，的确很难对面向对象的理念融会贯通。而且对于面向对象与基于对象这两种不同的编程方法，也很容易被初学者混淆。

C++是不适合小内存环境的。在DOS下，只有几百K内存可以直接使用，使用C++编程并非明智的选择。所以当时，虽然我很快就熟悉了C++的语法，但是对于C++编程却马上放弃了。

除了硬件条件，另一个重要原因是，我始终未能理解C++的类到底是怎样工作的。直到2000年底，我还是把C和C++分得很开，大多数程序都是用纯C来编写，只在极少环境下用C++（主要是试用过一阵子Borland C++ Builder）。不像许多C++的初学者所想，既然C只是C++的一个子集（到了C99之后已经不是了），那么C应该是一个非常简单的易掌握的语言。

其实，C的简单不过是相对于C++的博大精深而已，C语言中的setjmp()/longjmp()以及信号等概念并非所有C程序员都已掌握。C里面malloc()/free()背后的细节也被大多数程序员所忽略。我见过的大多数C程序员都或多或少地有滥用malloc的倾向，这些往往又导致了他们的程序效率低下。

甚至于许多在大学里修过C语言课程的本科毕业生，到毕业时也不一定能讲得清C语言中函数指针的运用场合。所以，我很庆幸我有很长的一段对C语言反复学习的时期。

从1992年到2000年，漫长的8年，中间没有间断过编程。这让我有足够的时间慢慢地领悟怎样用C语言编写软件，了解在C语言的背后，机器到底干了些什么，C语言有哪些不足之处，并且可以时刻和已经学习过的C++知识做一个印证。从1997年之后，我在自己的个人主页上，

就不断地公开自己编写的代码（之后又删掉了许多），对这些代码编程风格的指责也不断地夹杂于朋友发来的 E-mail 中。

有许多朋友很不习惯纯 C 的风格，如果使用.c 文件而不是.cpp 作为源文件在他们的眼中是一种不合时宜的东西。而我，不厌其烦地跟每个人解释，C++对于我还不达到郢匠运斤的境界，使用 C++只能让我害怕切掉鼻子。

2000 年夏，我在北京逛书店时发现了《C++编程思想》，那天几乎是站着读完前几章的，然后毫不犹豫地买了下来，接连两天手不离卷。以前从来没有看见过一本书把 C++诠释得如此清晰，将 C++的点滴认识汇集起来，那真是一种醍醐灌顶的感觉。从读完这本书后，一夜之间我就放弃了用 C 来写程序。

后来我在网上看这本书的书评，发现其翻译被人百般指责，颇为译者鸣不平。对于技术书籍的翻译，或许我的要求很低，只要译者自己在所翻译的内容面前有所研究，让翻译的文字不影响我的思考就行了。

理解 C++并非一日之功，也不是单靠阅读一本专著可以完成的，读者不应该把自己对书中内容的不甚理解而归咎到译文遣词造句的问题中去。

2000 年以后，关于 C++,关于编程方面的好书，翻译出版的越来越多。书店里不再全是开发工具的速成教材。网络上，程序员们的“好书太少”,“烂书太多”的言论逐渐消失。这无疑是一件好事。现在的初学者真是幸福了很多啊。

9.1 从 C 到 C++

C 通常都被看成 C++的一个子集，学习过 C++后再回头看 C，它真的是足够简单。

首先，C 只提供了 CPU 本来就提供的运算操作，把它们映射到更容易让人来理解和描述的书写形式上。为了方便程序员使用，引出了栈上临时变量和堆上动态变量的概念，而不是让程序员直接面对寄存器。

对于大块的数据，使用数据指针来表达。根据数据的组织方法，同类的数据结合在一起就成了数组，异类数据描述为结构，同一数据多种表述方法则被当做联合。程序的逻辑以代码行联合成代码段，再以代码段为单位，允许程序员以顺序、分支、循环的逻辑控制结构，使多个代码段合成一个个的函数。

C 语言的程序就是这样一个个函数组织起来的，函数之间可以按层次调用，去处理那些不同结构的数据。在我学习编程的日子，“程序=算法+数据结构”金科玉律出现在所有的教材中，C 语言就是它的完美体现。

C 语言是对汇编语言的一种抽象，正如汇编语言是对机器底层指令的一个最小幅度的抽象一样，是为了人类能更方便地控制机器。但是，

人在使用C语言控制机器时，并没有比汇编语言有观念上的不同，也就是说，程序员依旧是按机器模型来思考问题。在面对问题时，问题的描述往往并不符合机器思考的模型，程序员的工作也就是在两者之间搭上一座桥梁，让问题的描述可以用机器的思维来解决。C语言程序员干的就是这种事情。

所以，C语言是容易学的，因为它非常接近机器模型，而机器模型非常简洁、有条理（不然就无法用低成本制作出来）；同样，C语言也很有效率，正是因为它接近机器固有的模型，可以被有效地翻译过去。但是它同样导致了在解决问题时的困难性。C语言靠近机器太近，离实际问题的描述太远，在问题的描述和机器模型间找到对应关系，随着问题的复杂性提高而急剧增加。

有些高级语言为了解决用机器模型对问题进行建模的复杂度，改为对问题本身抽象，把问题归为特定的类别，然后，语言本身只解决这些被抽象化的问题，这样来减低编程的难度。这往往只能针对特定类型的问题才适用。

比如，通用语言lisp就是这样，它把所有的问题最终都归结为表。如果你把你的思考方式转换过去，把所有的问题都转换为表的处理，然后语言本身再帮你把表处理转换到机器语言的模型中，问题同样得到解决。这好像看起来很麻烦，但是实际做起来，会发现这也是一个不错的方法。

而C++则采用了另一套方法来减轻程序员解决问题的复杂度，那就是面向对象。面向对象的方法并没有试图要求程序员把需要解决的问题放在特定的“世界观”下去建模，而是为程序员提供了在描述问题时表示各种事物元素的工具。这是一种通用的方法，面对问题，我们把问题中的事物都表示为“对象”，对象有自己的行为，对自己遇到的问题提出符合机器模型的解决方案，或者把自己划为更小的对象集合。

这样，程序员就可以在对象的级别用对象本身的术语描述问题，而这些术语最终都会被直接对应到机器模型上。程序员遇到新的问题，只需要添加新的对象类型，使程序可以根据问题本身不断地进行术语调整，使得程序员总可以用问题本身的术语来描述问题。

这里想插上几句的是，多了解几门设计哲学完全不同的计算机语言对于程序员来说是件好事，虽然作为C++程序员，这辈子你都可能不会用lisp写程序，即使在UNIX平台下工作你也不想学会用emacs的elisp，但是了解一下lisp这样完全和C的思维方式不同的语言对开拓思维还是很有益处的。

而且，从某种意义上说，lisp比C对计算机软件开发技术的起源和发展意义更加重大。垃圾回收、数据结构、虚拟机等诸多概念都起源于lisp，把C或者C++作为编程世界的一切的话，程序员是很容易变成井底之蛙的。

对于非通用语言，游戏的控制脚本有时候也设计得比较特殊，比如starcraft里的关卡编辑，就是用不同的trigger组合起来的，表面上看起来只是让编辑人员设置一些状态，实际上也可以看做一种特殊的专用语言。

这些相异于机器模型的语言，通常对于特定的问题都是很好的解决方案，但是离开相应问题的领域，就会变得很难用。lisp绝对不适合写游戏的图像引擎，即使是对lisp爱好者，这也是一个不能否认的事实。

这种方法既给程序员提供了强大的语言抽象的能力，又保持了底层符合机器模型的优势，从而在设计变得简单的前提下不失其效率。

在面向对象的方法里，所有的东西都成了对象。对象是一种特殊的变量，并非机器模型中那种单一的数值。它在内部可以是一组数值的集合，也可以没有具体的数据，程序员只需要从外观上感受它是一个单体，表述一种概念。程序可以对之提出要求，要求它进行对于自身的运算。程序就是一组对象，让对象之间发出请求，互相通知该做点什么。而每个对象都有自己管辖的数据空间，这样自身不会被外界干扰。对于使用它的程序员来说，复杂性被隐藏在里面，在对象暴露给外部的简明接口之后。

光把事物分成一个个对象单独处理还不够，这仅仅是基于对象的思考方式。基于对象的设计方法中，仅仅只是让数据和所属的处理方法聚合在一起。而运用面向对象的设计方法时，我们把对象分为不同的类型，每个对象是这种类型的实例。类型的设计是有层次的。就好像动物是一种大类型，而哺乳动物和爬行动物都属于动物，却都是从动物分出的子类型。哺乳动物还可以再向下细分。对于每个类型层次上都有适用于所有这种类型对象的方法。

这就是说，如果我们创建了一个狗的类型，并让它可以接受一种在动物这个级别可以接收的消息“移动”；再对另外一种叫做乌龟的类型，同样实现了“移动”的方法；以后我们编写与“动物”来通讯的代码时，它可以自动地拥有对“狗”和“乌龟”进行处理的能力。

有些语言是完全采用纯粹的面向对象的设计思想，比如诞生于C++之前的smalltalk（smalltalk同样也值得C++程序员学习，如果只会使用一种语言，将极大地限制我们的大脑）。有一些人对面向对象可以解决一切问题提出了异议，尤其是世间万物皆对象颇受争议。例如：排序是一个很有共性，经常在解决问题中碰到的问题，但“排序”本身是一种对象吗？提出这些异议的语言设计者提倡结合各种方法来设计程序，这就形成了多重编程范式的程序设计语言概念，C++就是其中最成功的一例。

游戏程序员和C程序员有很多重叠之处，许多游戏程序员就是一个C程序员。如果一个C语言程序员不是一个游戏程序员，那么他和游戏

程序员最大的共性可能来自于对程序效率的敏感性。C 语言是追求高效的，而 C++却往往被世人误解。在众多的教科书上宣称，C++的程序会比 C 程序要慢，但是差别不大，仅在 10%以内。

就是这点差别，也经常被使用 C 的游戏程序员作为武器捍卫 C 语言在游戏程序编写上的地位。我不知道现在是否 C++已经成为开发游戏的主流语言，但我几年前认识的国内游戏制作圈子，还是以 C 语言开发为主。我指的不是用纯 C 而是在 C++编译器环境下去使用 C++中等同 C 的那个子集。

关于 10%的效率差别，我没有得到过确切的实例去证实，也没有见过有人用 C++去完全重新写以前用 C 完成的大项目并做一个性能测试。或许这只是一个臆测的数字，实际上效能还差不到那么大。因为认真学习过 C++的人都可以理解，完全可以用 C++写出 C 风格的程序，两者的速度差别在于解决问题的思想不同引起的编码风格差异。

而 C++编译器不断的改进以更适合语言特性本身，使得用 C++语言原本支持的编程范式写出的代码可能得到最好的优化，C 语言想使用面向对象或者泛型编程时，由于得不到编译器良好的支持（主要体现在编译器无法利用更多的信息做优化），性能会损失很多。

另一方面，CPU 厂商为了提高处理器的性能，往往在世界上已经存在的代码用得最多的编码形式下下工夫，提高性能。例如，利用堆栈传递参数为大多数高级语言的实现所必须，CPU 在这么多年的发展中，数据出入栈的速度已经被提高到了足够快的水平了。

随着 C++开发的软件逐年增加，对于 C 语言忠实爱好者的另一个不幸的消息是，一些原本 C 语言常用的方法生成的代码之速度已经开始落后于当年看似多了层间接性会更慢一些的 C++代码了。

在前面的章节我们已经见到了一个明显的例子，把一个全局唯一对象做成单件，比之把对象中的数据都申明为全局变量这种 C 语言中常用的方法要更加有效。

9.2 C vs C++，效率至上

大多数C++程序员都不会认为C++比C效率低，即使有效率差异，其差别也被认为可以忽略。到底C能比C++快多少，是C程序员更关心的。

我作为一个C程序员的历史远比C++程序员要长，甚至现在许多小项目我还是用纯C来写。但是这里想证明的却是：即使同样采用面向过程的软件设计方法，C编译器产生的代码效率会比C++编译器要低而不是高。

9.2.1 全局变量

全局变量在C中随处可见，而在C++中是会被C++程序员尽量回避的。C++程序员更多地会把一类全局变量写成一个类并做成单件，先取得单件的指针，再使用单件的方法来访问这些数据。表面上看起来C++中的方法会多一次间接性，但是现代CPU，在对待数据时，把数据集中却有助于提高数据高速缓存的使用效率，并且直接访问全局变量会产生一个32位地址数据，处理速度比用一个this指针加一个短偏移量要低。

9.2.2 函数调用的堆栈处理

C语言是用函数名的字符串本身来帮助连接器区分函数的，它不支持函数重载，也无法区分不同的函数调用形式。为了支持printf这样的不定参数个数的函数调用，标准的C语言实现强迫所有函数都选择同一种调用协议，即，由调用函数本身的代码来把压入堆栈的参数出栈。这比让函数自身弹参数出栈要稍微低效一点。

而大多数的C++编译器可以让不同的函数使用不同的调用方式，默认情况下不再使用C语言中那种略微低效的函数调用协议，只在需要支持变长参数的函数时做特殊处理（实际上，良好设计风格的C++接口不再需要变长参数）。

9.2.3 函数调用的参数传递

语言的标准并没有限制如何在函数调用时传递参数。虽然不少 C 编译器支持了使用寄存器传递参数，但并没有标准。在准标准下，所有的参数都必须被压入堆栈。

而 C++里类被广泛应用，成员函数的 this 指针成为一个默认必须传递的参数，为了提高效率，准标准中，这个指针是使用寄存器传递的（x86 环境下，几乎所有的编译器，在参数固定的情况下，都使用 ecx 传递 this 指针）。

9.2.4 临时变量的堆栈使用

C 语言中，临时变量必须放在一个语句块的头部（C99 对这点放宽了限制）。这使得局部变量可以一次预留出空间，但也导致了堆栈可能消耗得过快。这使得堆栈上的数据更容易变得分散，而分散的数据不利于处理器内部数据缓存的工作效率。C++只在使用到的时候才申明局部变量，这样虽然增加了一点点堆栈指针的操作指令（有时也会被编译器优化掉），但是增强了堆栈部分的内存使用效率。

9.2.5 代码生成

C 语言中用宏来做代码的自动生成，在不少代码空间换运行时间的场合有很好的效果。但是 C++更胜一筹，使用模板技术代替宏替换，不仅仅是增强了编译时的类型检测减少程序员犯错误的可能，更是增加了编译器在编译时的推导能力，让很多的工作在编译时完成，为运行期生成最佳的代码。

9.2.6 异常

C 语言中用 setjmp/longjmp 来实现异常状态的处理，这样给运行期带来了额外的负担。而 C++内建了异常，让编译器产生许多额外的代码，尽力实践代码空间换时间的策略，提高运行时间效率。

9.2.7 inline

C 语言原本是不支持 inline 的（不过在 C99 把 inline 纳入标准前，几乎所有的 C 商业编译器都支持了它），而 C++为了减少太多层次的调用消耗，很早就把 inline 收进标准。事实上，inline 在 C++中不可或缺，C++的编译器也把 inline 技术发挥得淋漓尽致。这使得看似复杂的 C++程序可以飞快地运行。

9.2.8 多态性

即使不采用面向对象的设计方法，有时候多态性也是必须实现的。C 语言中的方法是使用函数指针数组。这使使用变得麻烦，而且程序员手工书写的代码，即便完全模拟 C++编译器使用的虚表技术，也很难达到编译器的效率。编译器总是能了解更全面的信息做出局部的优化。

9.2.9 标准库的差异

一个各种讲解 C++标准库的书本上最常见的例子是 std::sort 和 C 标准库中的 qsort 的比较。qsort 要求客户程序员编写一个数据比较的函数让 qsort 调用。这比之 std::sort 多了很多的函数调用的消耗。我见过的使用 C 语言的游戏程序员在效率敏感的模块中都不直接使用 qsort 排序，就是这个原因。而 std::sort 使用模板技术，让比较函数内联，消除了这个函数调用消耗。

9.2.10 动态内存分配

许多 C 语言的程序员会非常在意从他们的键盘中诞生的代码的性能，而忽略标准库中函数的消耗。典型的案例是使用 malloc()/free()动态分配和释放内存。

在许多需要大量内存分配的模块中，标准的 malloc 几乎肯定成为速度瓶颈。C++语言设计时就注意到了这一点，所有 new 和 delete 操作符是可以以诸多方式重载由用户定义的。有经验的 C++程序员很容易

注意到这点，而很少有C程序员认真写过HEAPHOOK。

自从转向由C++做项目开发以来，我的个人体验是，几乎从来没有感受到用C去写程序可以比用C++来写运行得更快，如果有人发现C++的运行效率低劣，更多的可能是他并没有理解C++。

那么我们是否应该排斥用C语言做开发？那也未必。我从来都没有完全放弃过C语言。

使用C，就是纯粹的C，而不是混杂着C++语法的C形式的代码，可以获得许多速度之外的好处。

首先是更广泛的可移植性，几乎所有的开发平台都支持C语言，而不少平台的C++编译器都不那么完善。其次是使用C的思考方式，会让人不至于陷入面向对象或者是泛型编程的泥沼。这并不是说面向对象或泛型编程不好，而是即便是有经验的C++程序员也容易钻进设计方法的牛角尖，导致项目失败。

良好的C语言接口也可能使得模块的使用方法更容易理解（当然也有可能因为错误的设计变得糟糕），对多人合作或是多语言混合编程是一件好事。而C++要做到这一点很不容易，那需要更有经验的程序员，如果你在国内的游戏开发圈子干过几年，就知道一个很有经验的C++程序员是多么可遇不可求。

最后有一个非常现实的却被许多程序员忽略的问题。C语言是如此得简洁，以至于编译速度是可以如此之快。试着全部用.c做后缀名的源文件完成一个项目。你会感受到飞一般的编译速度。或许你再也不能忍受编译C++工程时一边看书一边等待的时光了。

9.3 优雅的C++

C++对于C是优雅的，这来自于C++把C语言中的诸多概念都予以细化，而又不增加运行开销，让编译器在编译时帮助做更多的检查，防止错误的发生。

C++继承了C的优良传统，也就是信任程序员，C++语言的语法对

程序员只起监督作用，并不阻止程序员干他们想干的事情。而程序员应该配合这点，让编译器尽可能地发挥其监督作用，优雅的程序从此而来。

9.3.1 消失了的宏

宏在C语言中有极其重要的地位，而到了C++，程序员却尽力回避它。宏在C语言的编程中里主要有四大用途：定义常数，代码生成，内联代码和对编译流程做出选择。

在C++里，定义常数通常用const类型来取代，例如用const float PI=3.1415927f来取代#define PI 3.1415927f。

C语言中不这样做往往是语言的限制，比如：

```
const int MAX_SIZE=1000;
int array[MAX_SIZE];
```

在C语言标准中得不到编译通过（这点在C99中已经得到改进），而在C++中是被允许的。

其实，在更多情况下，我们选择用一个内联函数返回一个常数值来取代直接定义一个全局的常数，这种更灵活的方案被许多经验丰富的C++程序员所采纳，因为C++对inline函数优化得很好，最终编译器会把它优化成一个常数，而没有函数调用的消耗。

inline（内联）函数在C++里运用得非常广泛，配合template可以完成许多以前只有宏才能完成的事情。比如经典的min：

```
#define min(a,b) ((a)<(b)?(a):(b))
```

到了C++里，我们通常写成了：

```
template <typename T> const T& min(const T &a,const T &b) {
return a<b?a:b;
}
```

这里涉及了C++中模板的使用。这是个有趣而复杂的东西，可以帮C++扩展许多奇妙的功能。包括取代宏的几乎所有功能，template就是宏的最佳替代品，我们在下面会看到更多的介绍。

9.3.2 const 修饰及类型转换

判断一个 C++程序是否由有经验的 C++程序员写出来，最简单的方法之一就是看看代码中 const 出现的频率。

几乎所有的 C++程序员在描述字符串常量的时候都会写 const char*而不是 C 语言程序员喜欢写的 char*。

成员函数的定义最后加上 const，实际上是指出隐式传入的 this 指针是一个 const 指针。

C++程序总是把函数的参数尽可能地写成 const 的，表示这个函数不会修改它。同样，成员变量也经常会被修饰成 const，这样就只能在构造时构建它们。另外，为成员函数增加一个 const 修饰表达这个函数不会修改类的成员变量也使得程序变得优雅。

尽可能地使用 const 使软件在设计阶段的思路更清晰，让编译器更加理解程序员的想法，以便在程序员不小心犯错误时即时指出。

const 这个关键字虽然从 C 时代就已经存在，但从来没有像 C++中这样被重视过。

另一个在 C++中被加强的是类型转换。

C 语言里面几乎所有类型之间都可以任意转换，这缘于 C 语言中汇编的根。在汇编里，所有的数据类型都被看做了内存地址上的数据。

而 C++里为了避免程序员犯错误，使用 static_cast 这个关键字以模板的语法表达逻辑上看起来可以互转换的类型变换。对于没有继承关系的类型，static_cast 直接会导致编译失败，这对监督程序员是一件好事。

对于 const 和非 const 之间的转换，C++里规定更为严格：只能通过 const_cast 转换。这又一次说明了 const 在 C++中的重要，

事实上，严格地描述每个对象的 const 性质，帮助过我无数次在编译期发现错误。

9.3.3 隐藏实现

优雅的 C++程序把每个类写到不同的编译单元中，以.cpp 实现文件和.h/.hpp 的头文件结对出现。C++中通常会使用 private 关键字来把类实现的细节隐藏起来，而只把对外的方法用 public 暴露出来。

这样，使用类的程序员所需要关心的就仅仅是类的public方法，不会因为不慎失误而错误破坏了类自身的处理逻辑。

好的C++程序员会把所有的数据都声明成private的，尽量在同一个类中暴露过多的public方法，而protected应该慎用，至少尽量不用在成员数据上。

作为良好的面向对象设计，继承类很少去扩展基类的功能，这样子类作为具体实现被隐藏在基类的接口之后，结构变得干净整洁。

作为进一步的隐藏实现的设计，我们还能看到把类中的所有数据和一些私有操作放到一个类里面并只在公开的头文件中留下一个指针（或引用），这样，甚至不需要暴露设计的细节。例如“风魂”中的canvas类有类似这样的设计：

```
class canvas : bitmap {
 class data;
 data *internal_data;
public:
    ...
};
```

canvas用一个bitmap来实现（这里使用了私有继承，仅表示了这个类用这个基类来实现），canvas本身需要比较复杂的数据结构和其内部实现，这些，全部封装在canvas::data中。在给canvas这个类的客户程序员的头文件里，完全隐藏了其具体的实现，这避免了我修改canvas的内部结构，导致重新编译仅仅只是使用canvas这个类的代码。另外，关于canvas这个类的头文件也变得很简洁，这能增加程序员的好感，减少整个工程的编译时间。

好的C++程序员都会让头文件的依赖关系变得最小，合理地使用类声明，而不是包含进整个类的定义。有经验的C++程序员可以让整个工程不那么混杂成一锅粥，各个类清晰明了，即使不用预编译头文件，也获得很高的编译速度。

更进一步的，程序员还可以把基类写成一个纯虚类，而且不包含任何成员数据，具体的实现可以通过继承这个基类来完成，这样的基类被称为接口类。实现类可以对客户程序员完全隐藏，只让使用它们的客户

程序员看到接口类，这同样是C++面向对象设计中的常用手法。

让程序员只看到该看到的，只能用允许使用的，这样使得编译器可以最大可能地指出程序员无意中犯的错误。

9.3.4 引用而非指针

C++继承了C的所有东西，包括指针。虽然很多程序员不愿意承认，指针也是最容易导致资源被遗漏忘记释放的罪魁祸首。其原因就在于，当指针被赋予新的值的时候，旧的值就消失了。如果消失了的值是唯一指向着某个对象的地址，那么这个对象控制着的资源将永远存留在内存中直到进程结束被系统回收。

但是指针这个概念又是不可或缺的，传递对象如果全部靠复制，程序将像蜗牛一样地爬行，更不用说某些对象干脆被设计成禁止复制的。C++语言增加了引用的概念。

引用和指针在实现本质上是完全相同的，但是引用在语法上只能在构造的那一刻被赋值，并永远不能修改。优雅的C++程序甚至会在类的设计时，用引用代替成员变量中的指针，而减少实现类的时候不小心犯错误。对于参数传递，如果既希望指针本身指向的位置不被修改，而且指向的内容本身也不被修改，则C++程序员不再需要C里面蹩脚的`const T * const`的写法，只需要写`const T &`。传递`const T &`对于编写函数实现的人来说就和传值`T`有相同的观感。

9.3.5 名字空间

当C语言构建的项目膨胀到几万行之后，程序员将开始苦恼全局变量和函数的名字越来越难取。然后就出现了诸多形如`xxx_xxx_xxx`的名字。

C++为解决这个问题引入了名字空间（namespace），名字空间比类更灵活，它可以随意被打开和关闭，引用外部的名字空间到当前位置。这样有效地防止了不同的库之间名字冲突的问题。

类库的开发人员大多仿效标准库，和标准库使用了一个叫做`std`的

名字空间一样，他们都把自己的库放进专有的名字空间中。std::vector虽然显得有点冗长，但是比起standard_vector或者是STD_vector要清楚得多了。

9.4 C++的误区

C++提供了太多的灵活性，这些灵活性中补满了陷阱，一不小心就会让初学者限进去。工具越方便，就越容易被滥用，我们应该时刻警惕这一点。

以我个人的经历，总结出一条不被很多人接受，但自己却十分受用的准则：尽可能地使用结构最简单的工具来完成任务，直到这个工具不合适。

能用纯C写的程序不用C++，能用C的原生数组的情况下不用std::vector。能用std::vector的情况不用std::map，能自己写的代码不用第三方库。但是，请不要误解这条原则。首先，世界上没有绝对的原则，如果它们存在，我们就可以按手册来编写完美的程序了。完美的程序不存在，绝对的原则也同样不存在。

其次，我不主张重写C++的标准库，重写MFC或者任何已经存在的广为大众所使用的代码;也不主张去用C去实现面向对象的设计方法，或是拿宏去取代模板这些回到历史中去的事情。C++的每一个特性都有它们存在的理由，大多数程序员都没有达到非议标准的技术层次，提出上面的准则需要运用者把握其中的度。这个度就是使用工具的人对工具本身的了解程度。

9.4.1 类层次过细

类层次过细是初学面向对象编程的程序员最容易犯的错误。世间万物都是对象看起来很美好，把一切实现隐藏到接口之后也很美好，但是这些东西可能造成的间接调用引起的消耗在特定环境下就不那么美好

了。就好像游戏的图像以很多像素点组成，而每个像素点又有很多格式和很多绘制方法。

看起来，为像素定义一个基类，提供一些接口，以处理每个像素点对象的方式处理整个图像，表面上看来有非常大的弹性，是一个很好的面向对象设计，但是实现出来，恐怕没有人愿意忍受这样速度的游戏了。

9.4.2 滥用操作符重载

如果统计学过BASIC和C的程序员转向C++后最爱做的一件事情，莫过于设计一个字符串类了。BASIC及其诸多其他语言都提供了丰富的字符串处理功能。

让字符串之间任意地赋值，用加号连接字符串，在C中是一件完成不了的任务。而到了C++，类可以提供操作符重载，不仅仅是运算符号，连强制转换都可以重载了。这使得C++程序可以变得更加人性化而不是沉闷的函数调用。

虽然 std::string 已经实现了一个很好的字符串类，MFC 的 CString 也干得不错，但是，还是有众多C++初学者们对实现一个字符串类乐此不疲。

从字符串类的实现开始，带有游戏之心的C++程序员们往往进一步地思考怎样给他们新做出来的类重载一些操作符，让使用起来更加“漂亮”。

实际上，操作符重载只是给函数起了个有趣的名字，并且用某种符合这个操作符的形式来调用这个函数而已。在操作符重载被滥用的环境下，它给客户程序员带来的好处仅仅只是减少了程序员的击键次数，而随之而来的对代码的理解困惑，对程序的含义混淆，以及让编译器隐式地做了易被人忽视的动作，危害更大。

为了使用户写出的类适用于现存的模板，让用户构造的对象和C++原生的类型有相同的表现，操作符重载是有意义的。如果仅仅是为了程序看起来紧凑，或是类使用起来“好玩”，那就是对其的滥用了。

9.4.3 滥用标准容器

用过 std::vector 后就不想再碰语言中原本提供的数组，用过 std::string 后就不知道 const char *为何物，这是许多C++程序员的通病。

不错，许多C++权威都告诫大家尽量使用C++标准库，但在使用它们的时候有没有想过，有时有更简单的选择呢？

更加容易被滥用的是 std::map，有人几乎是离开了 std::map 程序就写不下去。如果只是为了创建一张 key-value 的对应表可供查询，那么这些人几乎从来没有考虑过，我们完全可以先用数组把这些键-值对应关系记录下来，只做一次排序，在检索时就可以使用二分查找了。这是一种更加快捷而又节省内存的方法。

滥用标准库或者是第三方库的例子真是举不胜举。对于玩弄C++模板等所谓高阶特性的人们，可曾想过，那些紧凑的代码后面隐藏了些什么？

9.4.4 滥用动态类型识别和异常

C++提供了RTTI动态类型识别的特性，可以在运行时知道一个对象的类型。这更多地被用于捕获异常的过程中，它也很容易在别的地方被滥用。

使用 if/else 或者 switch/case 结构对于不用类型的对象做分别处理是一件非常危险的事情。程序虽然可以正确地运行，却导致了设计上的难堪。

多考虑一下不同的设计模式，这些需要动态类型识别的地方都是可以用增加类层次来解决的，虽然会增加一些代码量，但是程序也会变得更加容易扩展。

既然已经提到了异常，那么异常作为一种特殊的函数返回方法正在受到广大C++程序员所钟爱。使用异常，在异常状态不发生的时候几乎是零消耗的（这里说的是“几乎”，实际上每个 try 语句块，都还是有一点点的速度消耗的，另外使用异常会使代码膨胀许多），它甚至比判断

函数返回值更加高效。

但是，它也会带来更多的设计复杂度。异常是一把双刃剑，既非常容易把整个软件的设计带进清晰明了的优雅境界，也容易将其引入危机四伏难以维护的绝地。

规范化异常（Exception Specifications）看起来会让情况好很多，既在函数申明中除了注明返回值之外，同时写明该函数可能抛出的异常。当函数抛出了未指定的异常时，让程序自己崩溃方便开发者查错。可惜的是，现有的C++编译器对它支持得还不够好，至少目前VC是不完全支持规范化异常的。

已经存于世的许多类库也未按它的要求书写调用接口，如果使用了规范化异常，一个函数万一抛出了非它指定的异常，程序的解决方案是直接退出，而不去调用任何其他的析构函数，这就是一种崩溃，这仅仅有利于开发期，而不利于软件发布后的环境。

而很多情况下，我们几乎很难描述一个函数可能抛出的异常种类，这不仅仅是使用未使用规范化异常的第三方库会造成的，在使用模板这另一个C++高级特性时，由于我们无法预知模板中类型的具体实现，就无法清楚地知道异常抛出的情况了（其实使用简单的C++特性，比如使用函数指针，也会有同样的问题）。

规范化异常是一个怪胎，就我个人的看法，如果让编译时就让不遵守异常抛出规范的代码直接引起编译错误，我们则会少犯很多错误。但是规范化异常必须保持和原有的异常标准兼容，并提供足够的灵活性。所以，虽然原本规范化异常是用来解决异常安全，这个C++设计中其中一个最让人头痛的问题的，但是我们看到，由此引入的新问题或许让事情变得更糟。

对于异常，谨慎用之。

9.4.5 滥用多重继承

为什么让C++存在多重继承？这个问题被争论过无数次。自然界的万物大多不是从一条继承线上继承下来，正如我们有父亲也有母亲一样，这也是多重继承存在的意义。

有无数人反对使用多重继承，因为它的实现过于复杂。几乎所有的多重继承问题都可以化为组合方式来解决，用好多重继承比放弃它换用别的解决方案要难得多。

许多有多年C++开发经验的程序员一直都没有搞清楚过虚继承的含义就是一个旁证。复杂的菱形继承是学习C++中最难搞明白的概念之一，该种情况下，连书写构造函数都可能困扰程序员。到我们压抑不住使用多重继承这个C++高级特性以求得问题表述上的完美时，再留一些思考时间吧，看看是否滥用了它们。

```
class base {
          public base (int);
};
class class_a : virtual public base {
            public class_a(int);
};
class class_b : virtual public base {
          public class_b(int);
};
class myclass : public base_a, public base_b {
          public myclass(int);
     };
```

上面这段代码中 myclass 的构造函数该怎么写，以及具体会怎么工作，恐怕许多C++程序员都很难说清楚。

9.4.6 忽视C++高级特性的复杂度

C++中有很多高级特性是最近几年才被挖掘出来的，如果你沉浸进去，会发现一片有趣的天地。通过阅读大师们的专著，那些C++技巧往往能让人瞠目结舌。

这些都容易使高智商的程序员们把C++当做一种高级智力玩具。但是，一句忠告：请不要忽视它们的复杂度。把两种复杂的C++特性运用在项目中时，其复杂度不是两倍，而是平方。有经验的C++读者可以回忆一下关于模板的异常安全问题的讨论，会赞同我这个观点的。

9.4.7 学习 C++

本章并非一个 C++的教学，也不是一份 C++的入门读物。我作为一个 C++爱好者，对 C++研究多年，学得越多，越是发现 C++的博大精深。

许多对 C++的认识也是螺旋上升的，以前认为糟糕的方法有一天又可能重新拿回来使用，一些精彩的东西也会在几年之后放弃。不断地学习，不断地认识，终有一天会进入 C++大师的境界。我现在也无法把自己学会的东西，理解的东西借助文字教给新入门者，因为许多的知识需要自己在实践中领悟。过多的细节，太快地展现出来反而会增加学习的复杂度，落得无法消化。

以上，我尽力用简短的文字把 C++美好的一面展现给大家，这里只是 C++语言的一个很小的角落，甚至只是我对 C++了解的部分中的一角。如果读者期望 C++成为自己开发中的利器，就应该多用 C++做项目，尽可能地体验更多的设计方法，用心去写程序，而不是单单去实现而已。自己写出的代码，多多思考，对感觉不好的部分重新设计。

学习 C++，多看这方面的专著是必修课。这个方面，我推荐以下的阅读次序：

学习 C++首先应该学会 C，至少也是 C++中类似 C 的那个部分。如果完成了这个阶段而想做一个 C++入门，首先推荐的还是《C++编程思想》，然后是 *C++ Primer* 和 C++祖师爷的专著 *The C++ Programming Language*。后两本都是大部头，或许没有前一本读得快，但也绝对是值得耐心看完的好书。

无论何时，一份 C++标准文档都是很具有参考价值的，在你对 C++的语法点产生疑惑的时候，最好的方法不是去查某本 C++教材，而是直接翻阅标准文档。

只是不像前面三本书，C++标准文档（最新的一份是 1998 年制定的）没有中文译本，而且似乎也只有电子档。不过好在通常我们也不需要躺在床上一页页地阅读，电子档更适合检索。

记得当初买到此书后还真的是废寝忘食呢，一日一夜读完）

深入了解 C++应该从其设计的根源及 C++编译器实现方法入手。同样是 C++的发明者的著作《C++的发展与演化》对于众多 C++爱好者都

是一本不可放过的好书。而另一本由 *C++ Primer* 作者所著的《深度探索C++物件模型》对追求高效的C++程序员尤为有用。

这两本书的中译本都翻译得非常流畅，读起来是一种享受。

另外在C++的进阶学习上，还推荐 *Effective C++*、*More Effective C++* 以及《C++沉思录》，这三本书也有中译本，不过都需要有一定的C++开发经验才适合阅读。

近年来，泛型编程非常流行，我不主张其被滥用，但是作为C++爱好者也一定是要了解的。侯捷先生的大作《STL 源码剖析》是首先值得推荐的，虽然对于泛型技术的初学者有点难，不过花上点时间也能读懂，而后能获得巨大的收获。

如果你对C++中 `template` 这个关键字的使用有足够的自信，对泛型编程自我感觉良好，那么可以去读一下 《C++设计新思维——泛型编程与设计模式之应用》，我保证这本书会让你叹为观止。一本颇有些难度的书哟，阅读之前请做好心理准备。

如果只是想更加精通 C++的标准库，暂时还没有想过研究 boost 甚至 loki 的源代码，那么我先推荐《C++标准程序库》，这简直是一本C++程序员案头必备的参考手册，一旦使用 C++标准库进行开发，迟早你会买上一本的。除此之外，《Effective STL 中文版》会和它的前辈 *Effective C++*、*More Effective C++* 一样，让人产生阅读的快乐。

9.5 动态内存分配的优化

动态内存分配在程序设计中经常会用到。在 C 语言中，通常偏向于静态地使用数据，即使需要动态内存分配，有经验的C 程序员也是大块内存集中管理。

如果因为需要实现一个简单的动态数据结构，去管理许多但每个尺寸都不大的数据，而频繁地调用 `malloc()/free()` 必然会让速度瓶颈移到这两个函数上。

关于操作系统如何分配内存的问题，已经有很多相关的专著讲解其

理论，我们这里只需要知道，作为一个普适性的内存分配算法，需要许多操作已经对每个内存块准备一些额外的数据空间进行管理。如果被分配的块很大，则额外的空间显得微乎其微；而大面积的数据处理必然要消耗大量的时间，那么处理分配和释放的操作时间也是可以忽略的。

问题出在小内存分配上，作为小内存分配的要求，在游戏（或者别的什么软件）中需求一般很大，它具有特殊性：体积小，花在其中数据本身的处理时间不多。

这样，内存分配的操作时间很容易变得相对很长，额外需要的内存量也会占据大量的资源。这是很值得被优化的模块。借助 C++，我们可以把它实现得很漂亮，而且 C++的类允许 new 的重载，这样我们还可以很方便地为需要大量创建出来的对象搭配上优化过的小内存分配程序。

下面看看我的实现：

```
class smallobject_allocator {
public:
 enum {
      chunk_limit=16384,
      max_number=64,
      align_size=8,
      chunk_number=chunk_limit/align_size,
 };
private:
 struct memory_list {
      memory_list *_next;
 };
 struct chunk_list {
      chunk_list *_next;
      memory_list *_data;
 };
 memory_list *_free_list[chunk_number];
 long _guard[chunk_number];

 chunk_list *_chunk_list;
 long _chunk_guard;


 static smallobject_allocator* _instance;
 static long _singleton_guard;
```

```
 static bool _singleton_destroyed;
 static void create_instance();
 static size_t chunk_index(size_t bytes) {
     size_t idx=(bytes - 1) / align_size;
     assert(idx>=0 && idx<chunk_number);
     return idx;
 }

 smallobject_allocator();
 memory_list* alloc_chunk(size_t idx);
public:
 ~smallobject_allocator();
 static smallobject_allocator& instance() {
     if (!_instance) {
         create_instance();
     }
     return *_instance;
 }
 void* allocate(size_t size);
 void deallocate(void *p,size_t size);
};

smallobject_allocator *smallobject_allocator::_ instance=0;
long smallobject_allocator::_singleton_guard=0;
bool smallobject_allocator::_singleton_destroyed= false;

void smallobject_allocator::create_instance()
{
 thread_guard guard(&_singleton_guard);
 if (_instance)
     return;
 assert(!_singleton_destroyed);
 static smallobject_allocator obj;
 _instance=&obj;
 }

 smallobject_allocator::smallobject_allocator()
 {
 _chunk_list=0;
 _chunk_guard=0;
 ::memset(_free_list,0,sizeof(_free_list));
```

```
 ::memset(_guard,0,sizeof(_guard));
}

smallobject_allocator::~smallobject_allocator()
{
 int s=0;
 chunk_list *temp=_chunk_list;
 while (temp) {
     ++s;
     temp=temp->_next;
 }

void **chunk=reinterpret_cast<void**>(malloc(s*sizeof (void *)));
 temp=_chunk_list;
 int i=0;
 while (temp) {
     chunk[i]=temp->_data;
     ++i;
     temp=temp->_next;
 }
 for (i=0;i<s;i++) {
     ::free(chunk[i]);
 }
 ::free(chunk);

 _singleton_destroyed=true;
 _instance=0;
}

inline size_t _min(size_t a,size_t b)
{
 return a<b ? a : b;
}

smallobject_allocator::memory_list*
smallobject_allocator::alloc_chunk(size_t idx)
{
 const size_t node_size=(idx+1)*align_size;
 const size_t chunk_size=_min(chunk_limit/node_
       size*node_size,node_size*max_number);
```

```
 thread_guard guard(&_chunk_guard);
 memory_list * &current_list=_free_list[idx];
 if (current_list)
     return current_list;
 memory_list *ret=current_list=reinterpret_cast
                              <memory_list*>(::malloc(chunk_size));
 memory_list *iter=ret;
 for (size_t i=0;i<=chunk_size-node_size*2;i+=node_size) {
     iter=iter->_next=iter+(idx+1)*align_size/sizeof (*iter);
 }
 iter->_next=0;

 return ret;
}

void* smallobject_allocator::allocate(size_t size)
{
 size_t idx=chunk_index(size);
 assert(idx<chunk_number);

 thread_guard guard(&_guard[idx]);

 memory_list * &temp=_free_list[idx];

 if (!temp) {
     memory_list *new_chunk=alloc_chunk(idx);

     chunk_list *chunk_node;
     if (chunk_index(sizeof(chunk_list))==idx) {
         chunk_node=reinterpret_cast<chunk_list *>(temp);
         temp=temp->_next;
     }
     else {
         chunk_node=reinterpret_cast<chunk_list*>
                    (allocate(sizeof(chunk_list)));
     }

     thread_guard guard(&_chunk_guard);
     chunk_node->_next=_chunk_list;
     chunk_node->_data=new_chunk;
     _chunk_list=chunk_node;
```

```
 }

 void *ret=temp;
 temp=temp->_next;

 return ret;
}

void smallobject_allocator::deallocate(void *p,size_t size)
{
 size_t idx=chunk_index(size);
 assert(idx<chunk_number);

 memory_list * free_block=reinterpret_cast<memory_ list*>(p);

 thread_guard guard(&_guard[idx]);

 memory_list * &temp=_free_list[idx];
 free_block->_next=temp;
 temp=free_block;
}
```

为了在多线程环境下正确地使用，上面这个程序用到了一个线程锁的类 thread_guard。定义我放在了下面，如果在单线程环境下使用，可以方便地把这个类的构造及析构函数定义为空，让编译器优化掉相关代码。当然也可以把 smallobject_allocator 写成模板类，让用户来决定使用单线程版本还是多线程版本。

```
class thread_guard {
 long *_mutex;
public:
 thread_guard(long *);
 ~thread_guard();
};

inline thread_guard::thread_guard(long *m)
{
 _mutex=m;
 while (::InterlockedExchange(_mutex,1)) {
     // wait for unlock
     ::Sleep(0);
```

```
 }
}

inline thread_guard::~thread_guard()
{
 ::InterlockedExchange(_mutex,0);
}
```

这里，我们用到的是一种叫做 free list 的小内存分配优化的方案。以分配 8 字节的内存空间为例，当用户第一次请求 8 字节的内存空间时，程序并不是直接向系统申请 8 个字节的空间，而是申请大小为 chunk_size 的一整页（这里是 2K）。然后只把前 8 个字节返回给请求者，下面的部分备用。这些剩下的部分，用一个链表串起来，而因为数据空间还没有被使用，那么链表指针可以直接占用数据空间本身。

在小内存分配器里保留着这个链表的头指针，如果接下来又接到了分配 8 个字节的请求，那么只需要从链表头那出一个结点供请求者使用，直到整个备用链表使用完，即链表的头指针为空（如图 9-1 所示）。这个使用，只需要再次向操作系统申请新的一页空间，并重新按链表的形式组织起来重复上面的工作就好了。

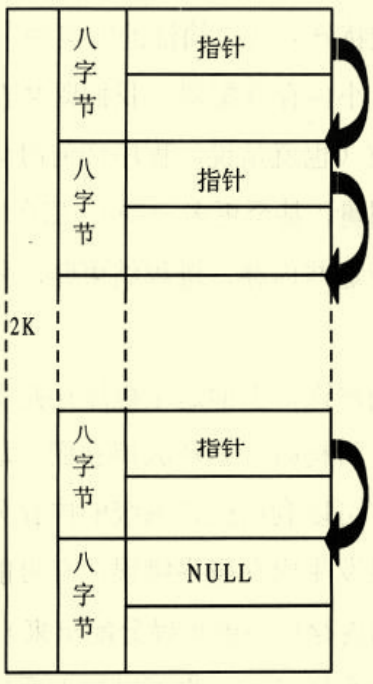


图 9-1 链表的头指针为空

当用户使用完了这 8 个字节，把内存交还，操作比分配更简单，我们不需要真正把空间交还给系统，迟早又会有新的分配请求发进来的，不是吗？这时只需要把需要回收的 8 个字节单元链接到备用链表的头部就可以了。

在小内存分配器中可以保存许多这样的备用链表，这样就可以满足几乎所有的小内存分配需要。

由于在 x86 系统上，数据被以 8 字节（或是 4 字节，16 字节）对齐比较高效，而且为了提高备用链表的复用性，我们会把小内存分配需求圆整到 8 的倍数，即，如果用户请求 7 个字节，实际我们分配给他的也将是 8 个。而如果请求 9 个字节的话，则返回一个 16 字节的空间。

这个方案也有诸多问题并是可以被改进的，比如，内存其实是只分配和没有真正释放过的，充其量是做到部分的内存重用而已。所以软件必须被良好地设计，不要在某个峰值时间里使用大量的相同大小内存块。改进方案可以通过在内存不够时，想办法真正回收掉不再使用的内存，这里没有做这种深入的展开，除了文字篇幅的原因外，在实际运用中其实意义并非很大。

另外，对于内存块不能被页大小整除时，浪费也比较大，对于提高程序的性能，特殊问题特殊对待，简洁的方案经常是最好的。

对于实现这么一个小内存分配器，我们既然在整个程序运行过程中都不对内存做真正释放（也就是说，程序任何时刻占用的内存都是曾经发生过的峰值内存占用量，甚至更多一些），那么实际上，我们甚至不需要在程序结束时释放掉这些内存。进程结束时，所有的内存都一定被操作系统清空，不是吗？

这个程序里，我们并没有使用额外的空间记录这一系列内存指针，而是利用了这个分配器本身，这需要一点点技巧，尤其在支持多线程方面，这样的设计让我头痛了好一阵子。

但出于程序设计的严谨，上面这个程序还是保留了历史上分配出来的内存页的地址，并在析构时真正释放掉它们。这样做，符合了模块对资源管理的完备性，并可以利用操作系统的内存分配释放流程发现一些潜在的程序错误（一旦发生内存越界错误，有可能会导致内存块不能正常地释放，甚至使新的内存块不能正确分配出来）。

最后，就是现在这个样子了。你可以把上面的代码抄进你自己的项目中去用，即使还有进一步优化的余地，我这里也能保证它已经非常高

效了。或者你也有兴趣仔细研究一下，代码本身还是有点意思的。

值得多说一句的是代码中间的多线程支持的设计，我在这里尽可能考虑多线程并发时的效率，让同时分配不同大小的内存时尽可能不需要互相等待，所以用到了多个不同的线程锁。

但是，需要让不同的线程在同时分配新的内存页时又不至于出错。这个问题是如何解决的，可以参考源码。需要注意的是，如果是单CPU上模拟出来的多线程并行，这种方式比直接在分配内存和释放内存时简单地加一道线程锁更高效。因为同一时间只有一个线程在运行代码，这边的线程等待时，总有另一个在运行。更多的线程锁定操作，只会浪费更多的资源。

smallobject_allocator作为一个单件（singleton）的设计也有一定的参考意义。smallobject_allocator::instance()的设计是满足多线程安全又不失其效率的，而且一旦发生设计上的失误，在单件被析构后，又有地方企图调用它分配内存，代码会用assert对程序员发出警告。在C++里，对象的销毁问题总是最能烦恼程序员的，我们应该对其产生足够的警惕性。

在C++中使用这个分配器很简单，我们只需要再做一个简单的封装。

```
struct small_alloc {
 static void* operator new(size_t size) {
     return smallobject_allocator::instance().allocate (size);
 }
 static void operator delete(void *p,size_t size) {
     smallobbject_allocator::instance().deallocate (p,size);
 }
};
```

让需要使用小内存分配的类，继承上面这个类，分配和释放就可以调用小内存分配器了。

需要多说一句的是，C++标准中，重载delete可以在两个版本中选一个。分别是接收一个指针的，和接收一个指针和一个长度信息的。

把类的尺寸作为一个对象的默认大小是错误的，因为每个类都有可能被继承，而使尺寸变大。但是编译器会知道这些细节，为对象的

delete 操作传递正确的大小。不过这在某些编译器中实现得并不好，比如 VC 6.0，编译器是无法检测到被继承过的多态的对象指针的正确大小的。

只要你在类声明中把析构函数定义为虚的，VC 6.0 会判断你将可能继承这个类，如果你此时只提供一份带 size 的 operator delete，编译器也会警告你。

对于这类情况，我们可以让分配出来的内存多消耗 4 个字节记录内存块本身的大小，这样就不需要在释放时主动告诉分配器这个信息了。这样的实现在下一章提供的代码中将会被实现。在下一章关于 C++程序调试的章节，我们会对内存动态分配的话题做进一步的展开，不光是为了效率，还为了更实用，更方便排错。

9.5.1 另一种优化策略：多次分配，一次释放

我从我的一个朋友庞鑫（国内最早的游戏程序员之一）那里听过一则小故事，他曾经在他的游戏中用粒子系统实现了一个爆炸的特效。据我所知，那是国内比较早的 3D 游戏之一，在上个世纪末，PC 还不够快，显卡也不够强，所以，在爆炸的一瞬间，游戏变得像慢动作一样，无数的碎片飞散开后，速度才恢复正常。

但是庞没有把其实现的效率低下归咎于硬件太慢，根据性能软件测试，发现速度的瓶颈在于大量粒子对象构造时的内存分配操作。在随后的内部测试版里，他根据具体情况自定义了一个内存分配函数，解决了这个问题。

我没有从他那听到具体的解决方案，但是这里我想写一下针对这个问题我自己的解决方法，这就是使用 placement new 做到多次分配一次释放。

```
class particle {
 particle();
 ~particle();
 friend class particle_allocator;
public:
```

```
  /* ... */
};

class particle_allocator {
 enum { chunk_size=1024 };
 typedef unsigned char byte;
 std::list<byte *> _chunks;
 size_t _particle_number;
public:
 particle_allocator() {
     _particle_number=0;
 }
 ~particle_allocator() {
     clearall_particle();
     for (std::list<byte *>::iterator iter=_chunks. begin();
          iter!=_chunks.end();++iter) {
       delete[] (*iter);
     }
 }

 particle* alloc_particle() {
     byte *temp;
     if (_particle_number==_chunks.size()*chunk_size) {
       temp=new byte[chunk_size*sizeof(particle)];
       _chunks.push_front(temp);
     }
     else {
temp=(*_chunks.begin())+_particle_number%chunk_size* sizeof(particle);
     }
     ++_particle_number;
     return new(temp) particle;
 }
 void clearall_particle() {
     std::list<byte *>::iterator iter=_chunks.begin();
     particle *temp=reinterpret_cast<particle *>(*iter);
     for (int i=0;i<_particle_number%chunk_size;++i, ++temp) {
       temp->~particle();
     }
     for (++iter;iter!=_chunks.end();++iter) {
       temp=reinterpret_cast<particle *>(*iter);
       for (int i=0;i<chunk_size;++i,++temp) {
           temp->~particle();
```

```
            }
        }
        _particle_number=0;
    }
};
```

注意看，particle 只能通过 particle_allocator::alloc_particle 分配出来。而 particle_allocator 每次会分配 1024 个 particle 需要的空间，如果这片空间使用完了，会申请一片新的，并用链表连接起来。所有的空间都在 particle_ allocator::~particle_allocator()中真正释放。

而每次，如果需要清空 particle_allocator 中分配出来的 particle，可以通过调用 particle_allocator::clearall_particle()，这个函数会逐个调用每个由它分配出来的 particle 的析构函数。当然，如果 particle 是一个 POD 类型，更可以省掉这一步。

我们在这里省略了释放内存的时间，甚至把申请内存的时间也压缩到零。这比上面的小内存分配的解决方案更快，究其原因，我们针对了特殊情况做出了特殊的优化。

技术问题讨论完了，还想补充的是，这个小节一开始讲述的那个故事还有一个小插曲。该游戏有一个赞助商，就是著名的 Intel 公司（Intel 公司经常资助一些游戏开发公司，让他们的新游戏为 Intel 最新 CPU 中的新技术做一些优化）。自然，每个内部测试版也有交到 Intel 公司。当爆炸时特效的粒子系统代码被完美地优化后，庞接到了一个 Intel 的工作人员打来的电话，一个略带责备的声音说，“那个很酷的爆炸时慢动作特效怎么在新版中被去掉了?”

9.5.2 垃圾回收

许多 C 或者 C++程序员对垃圾回收不屑一顾，认为垃圾回收肯定比自己来管理动态内存要低效，而且在回收的时候一定会让程序停顿在那里，但如果自己控制内存管理的话，分配和释放时间都是稳定的，不会导致程序停顿。最后，很多 C/C++程序员坚信在 C/C++中无法实现垃圾回收机制。这些错误的观点都是由于不了解垃圾回收的算法而臆想出来的。

其实，垃圾回收机制并不慢，甚至比动态内存分配更高效。因为我们可以只分配不释放，那么分配内存的时候，只需要从堆上一直地获得新的内存，移动堆顶的指针就够了；释放的过程被省略了，自然也加快了速度。现代的垃圾回收算法已经发展了很多，增量收集算法已经可以让垃圾回收过程分段进行，避免打断程序的运行了。而传统的动态内存管理的算法同样有在适当的时间收集内存碎片的工作要做，并不比垃圾

回收更有优势。

而垃圾回收的算法的基础通常基于扫描并标记当前可能被使用的所有内存块，从已经被分配的所有内存中把未标记的内存回收来做的。C/C++中无法实现垃圾回收的观点通常基于无法正确扫描出所有可能还会被使用的内存块，但是，看似不可能的事情实际上实现起来却并不复杂。

首先，通过扫描内存的数据，指向堆上动态分配出来内存的指针是很容易被识别出来的，如果有识别错误，也只能是把一些不是指针的数据当成指针，而不会把指针当成非指针数据。这样，回收垃圾的过程只会漏回收掉而不会错误地把不应该回收的内存清理。其次，如果回溯所有内存块被引用的根，只可能存在于全局变量和当前的栈内，而全局变量（包括函数内的静态变量）都是集中存在于 bss 段或 data 段中。

垃圾回收的时候，只需要扫描 bss 段，data 段以及当前被使用着的栈空间，找到可能是动态内存指针的量，把引用到的内存递归扫描就可以得到当前正在使用的所有动态内存了。

如果肯为你的工程实现一个不错的垃圾回收器，提高内存管理的速度，甚至减少总的内存消耗都是可能的。如果有兴趣的话，可以搜索一下网上已有的关于垃圾回收的论文和实现了的库，开拓视野对一个程序员尤为重要。

9.5.3 再论动态内存分配

在现在的32位架构上，即使我们主机上能插很多的内存，即使我们可以使用很多的硬盘空间去模拟虚拟内存，但我们的程序始终受限于4G的地址空间。

按C++的设计理念，对象在内存中是不方便物理上移动的，那牵扯到指针的重定位。所以，当我们动态分配出一块内存给一个对象使用，而之后，对象销毁，释放掉占用的内存（准确地说是地址空间）时，内存并不一定完完整整地交还给了系统，而是在内存中留下了碎片。

很多程序员不关心并非自己实现的代码，进而不关心动态内存分配是怎样实现的。所以很多人以为，只要我分配出来的内存被正确地释放

掉，一个周而复始的程序，就永远不可能出现内存不够的情况。似乎所有的内存满都是不小心内存泄露造成的。

其实在小内存环境下（比如许多嵌入式系统），或是网络游戏的服务器程序，即使在系统中事实上仍然有许多空闲内存时，内存碎片还会最终导致出现内存用完的情况非常常见。一个不断产生内存碎片的系统，不管产生的内存碎片多么小，只要时间足够长，就会将内存用完。好的内存分配器可以尽可能地避免这种情况的发生，但程序员的设计仍然可以左右结果。

“内存碎片”指一个系统中所有不可用的空闲内存。这些资源之所以不能被使用，是因为这些空闲内存太小，以不连续的形式出现在不同的位置，导致内存分配器无法使用它们。

许多数据可以在编译时决定，我们把它们写成了静态变量，编译器会帮你从地址空间正确地分配出来，并且安排得紧凑。而动态内存分配器则会浪费一定内存。

浪费在于：

（1）额外的内存开销，例如记录内存块的大小等。

（2）内部碎片，指用户在请求一块内存大小时，分配器多给的一些额外的空间。

（3）外部碎片，指用户释放掉一个动态分配出来的内存块，这个内存块由于前后的内存都在使用，而卡在中间很难被利用上。

内存碎片通常为了解决外部碎片的问题而存在，适当地将每次请求的内存圆整到一个合适的值，有利于内存的复用。而外部碎片是程序长期稳定运行的大敌。

2004 年，我们公司游戏部门从全国各大高校招聘了一大批毕业生进入技术部。在诸多培训和考察计划中，我设计了一次程序比赛。比赛的题目很简单，编写一个内存分配器，以一个 dll 的形式提供，dll 的实现和运行不得依赖任何库，全部必须自己实现。而我的测试程序会调用 dll 的导出函数来测试。

比赛持续了 2 个月，最后收到了将近 30 份合格的答卷，按照约定，

我用一些从实际游戏项目中截取的内存分配数据进行加工，制作了12组测试数据进行评分。

评分的标准很简单，首先，要求内存分配模块尽可能多地完成这些数据，尽量保证不出现内存满的情况。其次，在大家对测试数据的完成度相同的基础上，按照速度排序。每组数据都会对每份答卷有个排名，最终的排名取决于排名总和。

当然，程序出现任何bug，是取消比赛资格并接受惩罚的。

测试平台给出的模拟环境是8M 内存，只有一组数据的内存使用峰值达到过7M 以上，那组数据几乎是不可能全部完成的。大多数数据的内存峰值在4M 左右。

最终的结果很有意思，速度最快的几个程序没有能拿到前三名。他们几乎都在超过半数的测试数据中取得了好成绩，但是在其中两组数据上栽了跟头。那两组数据的共性是测试数据非常多，在一千万次以上，模拟了一个游戏长期运行的情况。虽然内存使用峰值不高，但是反复的分配和释放却很多。速度快的内存分配器同样制造了大量的内存碎片，是碎片导致了过早的内存不足。

最后获得一台最新款游戏机奖品的程序居然在各项测试中综合排名只有第4 第5 的水准。但其稳定的发挥却使得没有一个测试的分数靠后。

比赛技术后的研讨会上，我们让大家反思了一个问题，为什么很多运行速度快的分配器，在第一次内存不足时企图整理内存碎块，合并一些连续的碎片获得新的自由空间，却不能成功。

道理其实很简单，虽然长期运行的程序，内存释放的次数很多，内存外部碎片也会很多，表面看起来碎片被合并的概率还是很大的。但是，当内存不够的时候，往往需要的最大的那一块内存碎片正好能和前面或者后面的碎片合并。这个概率就非常得小了。

如果读者有兴趣，也可以自己写一个内存分配器。从一大块给定的内存块上分配出用户提交的内存。这真是一个理解动态内存分配的捷径呢。以后，你或许就会更加小心地使用动态内存。

9.6 template

template是C++中最难掌握的一部分，最初template只为取代C语言的宏而设计，为宏增加了语法检测的功能。后来，template被赋予了特例化的特性，可以针对某些特别的类型做特殊的操作。这种特质被一些C++程序员们充分挖掘，衍生出各种诡异的用法，让编译器在编译时期做出更多的决策，让许多运算都在编译期完成，为最终解决问题生成最精练的代码，这些技巧（到今天已经不再是一种玩弄C++语法的技巧，而是一门学问了）是初学C++的程序员很难想像的。

9.6.1 封装C++的成员函数调用

C++在C with Classes的年代有一个特性，它允许用户在调用一个类的成员函数之前和之后都会执行一段被定义的代码。这两段预代码以定义在类中的叫做call()和return()的函数实现。在《C++语言的设计与演化》中提到了它们最初的形式：

```
class monitor : object {
 /* ... */
 call() { /* grab lock */ }
 return() { /* release lock */ }
 /* ... */
};
```

这种机制在某些方面会有价值，但是最终并没有被加入C++标准中。不过今天，我们不必借助语言的标准，用最基本的template就可以实现它：

```
template <typename T>
class wrap {
 T* _pointer;
 wrap(const wrap &);
 wrap& operator=(const wrap&);
 class call_proxy {
     T* _pointer;
     call_proxy(T *p) : _pointer(p) {}
```

```
 public:
     ~call_proxy() {
          _pointer->after();
     }
     T* operator->() const {
          return _pointer;
     }
     friend class wrap<T>;
 };
public:
 wrap(T& x) : _pointer(&x) {}
 call_proxy operator->() const {
     _pointer->before() ;
     return call_proxy(_pointer) ;
 }
};
```

写个测试程序看一看：

```
class object {
public:
 void test() {
    puts("test");
 }
 void after() {
    puts("after");
 }
 void before() {
    puts("before");
 }
};

int main()
{
 object obj;
 wrap<object> x(obj);
 x->test();
}
```

输出：

```
before
```

```
test
after
```

注1：wrap类中的operator=与copy ctor没有被定义，这是防止客户程序员的误操作。这是C++中的一种常用技巧。

注2：call_proxy是以template中嵌套类的形式定义的，这是被template class支持的一种用法。call_proxy的定义是被wrap私有的，同样为了防止被误用。

注3：封装类成员函数的调用，在Bjarne Stroustrup个人主页上的一篇文章中有更加详细的阐述，讨论得更加深入和实用。

9.6.2 常数数量的对象复制

C/C++中最高效的数据块复制是使用memcpy，因为这是一个编译器内嵌函数，如果复制数据块的长度是常数的话，会依据具体情况做优化。比如：

```
memcpy(s,d,4);
```

既不会有一次函数调用，也不会产生一个循环来复制地址s到地址d的数据，而是一条汇编语句复制一个双字就够了。

不过memcpy是一个C语言函数，它类型不安全，数据块的长度也是按字节计而不根据按指针类型变化的，这非常容易使程序员犯错误。在C++中，我们可以写一个template更好地工作：

```
template <int size>
struct object_copy {
 template<typename T>
 object_copy(const T *s,T *d) {
     struct dummy {
          T data[size];
     };
```

```
        *reinterpret_cast<dummy*>(d)=*reinterpret_cast< const
dummy*>(s);
    };
};
```

现在就可以使用 object_copy<size>(source_address,target_address) 来代替 memcpy(source_address,target_address,size*sizeof(*source_address))了。

而且当源地址指针和目标地址指针不同类型的时候，编译器会报告编译错误提醒程序员注意。

这个模板在需要复制的对象重载了赋值符号的情况下可以正常工作的同时，对于POD类型，也能够让编译器做尽可能的优化。

9.6.3 对象计数

有一次，在一个项目中发生了内存使用不断扩张，直到内存溢出的现象。但是通过重载全局的 new 检测却没有在程序退出时有未释放的内存，这样，只好为项目中的每一个类加入对象计数器，检测每种对象在程序运行时曾经最多分配到多少个了。最后利用统计信息，我们以不断的被构造出来的对象为线索，找到了内存泄露的位置。

统计一类对象被构造的次数，这个工作很适合用template来实现，下面是从我参加的某个项目中抽出的代码：

```
class counter_data {
 int _counter;
 int _max;
public:
 counter_data(const char *name);
 ~counter_data() {}
 void inc() {
     if (++_counter>_max) {
         _max=_counter;
     }
 }
 void dec() {
     --_counter;
```

```
        assert(_counter>=0);
    }
    int current_count() const {
        return _counter;
    }
    int max_count() const {
        return _max;
    }
};

template <typename T>
class counter {
public:
    counter() { _data.inc(); }
    ~counter() { _data.dec(); }
    counter(const counter &) { _data.inc(); }
    int current_count() const { return _data.current_count(); }
    int max_count() const { return _data.max_count(); }
private:
    static counter_data _data;
};

template<typename T> counter_data
    counter<T>::_data=counter_data(typeid(T).name());
```

我们只需要把 counter 这个 template class 放到需要统计的类中作为一个私有成员变量就可以了。

```
class object {
    counter<object> _counter;
    /* ... */
};
```

它的工作原理很简单，object 被构造的时候，counter <object> 被构造一次。计数器把记数记录在静态成员变量中，当 object 析构时，计数器也能得到通知。这里，可以统计对象当前被构造出来的个数，并统计历史上的峰值使用数量。

程序使用了 RTTI 用对象的名字做统计记录输出，这个统计输出部分的代码将在下面列出来。

注1：counter的copy ctor是必须被定义出来的，设计过类似统计类的程序员很容易忽略它。如果没有这个构造函数，并不能保证每个object被构造时，counter<object>都得到计数，这种特殊情况发生在object被拷贝构造时，如果没有定义counter(const counter &)，编译器将不为counter的复制产生任何代码，包括counter::counter也不会被调用。

注2：counter<T>::_data这个静态变量可以被放在counter模板类的定义头文件中。不像普通类的静态成员变量的定义必须唯一地出现在一个编译单元里，模板类的静态成员变量的定义是被连接程序特殊处理的，这符合C++标准。

下面来看看上面被忽略了的：

```
counter_data::counter_data(const char *name)
counter_data::counter_data(const char *name)
{
 counter_log::instance().add(name,this);
}
```

它把自己注册到一个叫做counter_log的单件里面，并为自己起了一个名字，即所统计的类的名字。counter_log最终会把所有注册进来的counter_data的统计信息作为log输出，counter_log的具体实现很简单，限于篇幅，在这里省略不列。

9.6.4 避免重复代码

几乎任何一个有经验的程序员都明白，复制一段写过的代码，贴在另一个地方改动几句让它可以正常工作，这种令人作呕的行为，无论何时都应该被避免。

因为这简直是大部分程序错误的根源，在多个地方表达相似的概念，意味着日后改动一个地方，就必须记得改动相似的所有地方。而直接复制这些代码，导致编译器并不知道这些地方的相似性，不能为你提供这种

帮助。而人，随着项目的扩大，几乎不可能记住自己干过多少这种复制的动作。

程序员有时或许没有意识到他们在重复，有时是出于偷懒，这些都可以用加强对代码糟糕味道的嗅觉敏感度和提高作为编码者的责任心来达到。

但有时程序员会觉得是别无选择，比如下面一个例子。

我们需要写一个图像的 blit 函数，里面有一个模块是复制一块连续内存的。通常被写成这样：

```
void _blit(pixel *dst,const pixel *src,size_t s)
{
      for (size_t i=s;i!=0;--i,++dst,++src) {
       *dst=*src;
      }
}
```

就算改动永远不会发生，这种复制-修改的代码生成方法，也极大地破坏了代码的美感。程序员对这种方法的依赖，使得项目的源码像滚雪球似地膨胀，一些号称年产上几百万行代码的所谓“高产”程序员就是这样产生的，IDE的方便快捷更加加速了这种势头。正像当年的大跃进时期大办钢铁一样，可以想像产量背后的质量。

注意：我知道这样写不够高效，更高效率的做法是直接调用 memcpy，但这里只是一个为了说明问题的例子。

现在，需要另一个 blit 函数，它可以处理透明色，如果一个 pixel 的颜色是 mask_color，就应该被忽略掉。最简单的做法是再写一个函数：

```
void _blit_mask(pixel *dst,const pixel *src,size_t s)
{
      for (size_t i=s;i!=0;--i,++dst,++src) {
        if (*src!=mask_color)
              *dst=*src;
        }
}
```

看这两个函数，我们做了重复的工作，不是吗？ 第二个函数只是在第一个函数的基础上增加了一个判断而已，这真是太糟糕了。处于避免代码重复的原则，我们可以把两个函数合二为一：

```
void _blit(pixel *dst,const pixel *src,size_t s,bool mask_blit)
{
      for (size_t i=s;i!=0;--i,++dst,++src) {
        if (!mask_blit || *src!=mask_color)
              *dst=*src;
        }
}
```

但是，总是把效率放在第一位的游戏程序员会排斥第二种做法，因为它太低效了。在循环内部增加了一个判断，而条件分支又最影响现代CPU效率。

这个时候，拆分成两个函数，似乎是唯一的选择。

但是，我们可以考虑一下用模板来解决这个问题：

```
template <bool mask_blit>
void _blit(pixel *dst,const pixel *src,size_t s)
{
      for (size_t i=s;i!=0;--i,++dst,++src) {
        if (!mask_blit || *src!=mask_color)
              *dst=*src;
        }
}
```

为了方便使用，可以再多一次封装：

```
void blit(pixel *dst,const pixel *src,size_t s)
{
     _blit<false>(dst,src,s);
}

void mask_blit(pixel *dst,const pixel *src,size_t s)
{
     _blit<true>(dst,src,s);
}
```

mask_blit 作为模板参数，在编译期已经确定，所以那次比较会在编译期完成，不消耗运行时的时间。

9.6.5 选择最佳的容器

很多语言都提供了一种未定义类型的变量，可以用来保存各种不同

类型的变量。而 C++没有这样的类型可以使用，但是我们可以通过 template 来模拟一个，也就是设计一种容器可以存放不同类型的对象。这里并不列出它的实现代码，而是讨论另外一个问题。

我们有时候会在容器中到底是保存对象指针，还是保存一个对象值之间做一个选择。对于体积比较大的对象，保存对象指针更高效率，因为对象的复制可以做得很快。但是对于小对象，我们又希望保存对象的值，因为出于时间效率的考虑，保存值可以减少间接性；对于空间效率，我们可以减少一个指针的空间占用。

这种选择其实可以利用模板在编译期间由编译器自动进行：

```
template <typename T,class VC=value_container,
          class PC=pointer_container>
class container {
    typedef typename VC _v;
    typedef typename PC _p;
    template <bool small>
    struct _selector { typedef _v ret; };
    template <> struct _selector<false> { typedef _p ret; };
public:
    typedef _selector<sizeof(T)<=4>::ret best;
};
```

我们事前定义了 value_container 作为值容器，pointer_container 作为指针容器，这里只需要做一个选择器。

这里 container::_selector 就是一个选择器，它带有一个模板参数 bool small，利用模板特例化，当 small 是 true 的时候，选择 value_container 作为最价容器；当 small 为 false 时，选择 pointer_container。由于 container::_selector 根据 sizeof(T) 来决定，而 sizeof(T)<=4 的值可以在编译期计算出来。

```
container< my_type >::best a;
```

可以根据 my_type 的大小选择一个最佳容器。

9.6.6 延迟运算

如果我们自己定义了一种类型my_type，想让它可以支持更自然的运算法则，例如：

```
my_type A,B,C;

A=B+C;
```

这种情况下，我们需要重载operator+和operator=。一般都会做如下的定义：

```
my_type operator+(const my_type &lhs,const my_type &rhs);
my_type& my_type::operator=(const my_type &v);
```

重载赋值符号倒没什么，只是重载加号通常都会对性能产生影响，因为需要返回一个临时常量，还需要把这个临时变量拷贝到另一个变量中。通常，重载operator+=更符合性能要求，因为它不需要临时构造出一个对象，那么A=B+C，应该被写成A+=B，A+=C。

有没有办法既合A=B+C的习惯写法，又消掉这个临时对象呢？答案是：没问题！

我们可以观察到，如果没有赋值运算，单独的B+C在逻辑上是毫无意义的，只有把结果赋予A，加法操作才变得有意义。我们只需要把加法推迟到赋值的时候再做，并且由于已知存放结果的位置，所以临时变量是不需要的。

这样我们就有办法用一个模板把一种对象的operator+操作转换为operator+=操作了。

```
template<typename T>
class add_type {
 const T& _lhs;
 const T& _rhs;
public:
 add_type(const T &lhs,const T &rhs) :_lhs(lhs),_rhs(rhs) {}
 T& calculate(T &result) const {
     result=_lhs;
     result+=_rhs;
```

```
        return result;
    }
};

template <typename T>
add_type<T> operator+(const T &a,const T &b)
{
 return add_type<T>(a,b);
}
```

以假想的自定义类型 my_type 为例：

```
class my_type {
    /* ... */
    my_type& operator+=(const my_type &v);
    my_type& operator=(const add_type<my_type> &v) {
           return v.calculate(*this);
    }
    /* ... */
}
```

可以看到，关于 my_type 的 operator+这次只返回了一个很简单的对象 add_type<my_type>，虽然这也是个临时对象，不过体积很小，只有两个引用，而且利用了返回值优化。真正的加法运算是在 my_type 的 operator=中完成的（由 my_type::operator=调用 add_type<my_type>::calculate 间接调用了 my_type::operator+=。

注： 这种技术被叫做 Expression Templates，在一个高效率的 C++数值计算函数库 Blitz++中被广泛使用。以前 C 在数学运算领域要比 Fortran 慢上 20%左右，而 Blitz++正是依靠 template 技术把这个速度差抹平。

关于 Expression Templates，在 http://osl.iu.edu/~ tveldhui/ papers 可以找到更详细的论述。

9.6.7 编译时计算的游戏

```
template<int N>
class factorial {
public:
    enum { value = N * factorial<N-1>::value };
};

template<> class factorial<1> {
public:
    enum { value = 1 };
};
```

这个程序可以把N的阶乘的值作为一个编译期的常量。例如3的阶乘可以写做Factorial<3>::value。

这种让编译器在编译期递归运算的技巧还可以继续发展下去。再引入下一个小程序前，这里讲一个小故事：我在初中时有一次参加中学生编程比赛，有一个题目的第一步要求对三个数字进行排序。本来排序是小事一桩，如果不追求效率的话，我想大家都可以写出两重循环工作的冒泡排序程序：

```
inline void compare_swap(int &a,int &b)
{
 if (a>b) {
     int t=a;
     a=b;
     b=t;
 }
}

void sort(int *data,int n)
{
for (int i=0;i<n-1;i++) {
     for (int j=i+1;j<n;j++) {
         compare_swap(data[i],data[j]);
     }
 }
}
```

其实，如果n足够小，别的诸如快速排序的算法也不可能快过冒泡

法。

但是这里 n 实在是太小了，小到了 3。这迷惑了我，潜意识认为，只为 3 个数字就做两重循环简直是太浪费了，然后我就开始使用 if/else 的组合企图直接把 3 个数字的次序理清楚。或许是赛场上的紧张，居然弄错了好几次，浪费了大量时间才狼狈不堪地把排序程序做好。

其实，出于效率角度讲，用 if/else 展开循环的方法没有错，但是把代码搞得混乱就大为不该了。提高的效率远远比不上让代码混乱的代价。不过今天，我们有了模板这个武器，这种循环展开的活终于可以让编译器自动完成了：

```
template<int N>
struct inner_loop {
 static inline void expand(int* data) {
     compare_swap(*data, data[N]);
     inner_loop<N-1>::expand(data);
 }
};

template<> struct inner_loop<0> {
 static inline void expand(int*) {}
};


template<int N>
struct sort {
   static inline void expand(int* data) {
      inner_loop<N-1>::expand(data);
      sort<N-1>::expand(++data);
   }
};

template<> struct sort<1> {
   static inline void expand(int* data) {}
};
```

现在我们可以用这个模板来排序了。

```
int main()
{
 int a[]={3,2,1};
 const int len=sizeof(a)/sizeof(a[0]);
 sort<len>::expand(a);
 for (int i=0;i<len;i++) {
     printf("%d,",a[i]);
    }
}
```

如果你把编译器的优化选项开到最大，你会发现编译器在编译时就会把数组的次序排好。即使需要排序的数组并非编译时就可以决定，那样也不会有循环出现，两重循环全部被编译器展开大约如下：

```
compare_swap(*data,data[2]);
compare_swap(*data,data[1]);
++data;
compare_swap(*data,data[1]);
```

template 的活用技巧还有很多，如果读者真的感兴趣，可以先研究一下 STL 的源码，然后进一步可以去看 boost 的源码甚至 Loki 的。

但是，无论这项技术多么引人入胜，编译时计算多么诱人，都请不要滥用。在你想在项目中大规模地使用复杂的模板技巧时，请先了解下面几点：

（1）编译时的模板推导会使编译速度几何级数下降，直到令人发疯的地步，而且很容易就会引起编译器的抱怨（超过编译器的某些限制，甚至引起编译器内部出错）。

（2）许多模板技巧都是用生成更多的代码来切合不同的运行时情况以加快运行速度的，这是典型的空间换时间策略，很容易引起代码膨胀。

（3）不是所有编译器都支持所有的模板特性，当你的项目需要换一种编译环境时，很有可能代码不能正确地被编译了。

（4）相对于普通的 C++代码，模板非常不好调试。不光是运行时调试，对于没经验的模板编写/使用者来说，错用了模板引起的编译错误都不那么好找到缘由。

（5）模板直到客户程序员用到了的时候才实例化出来，这就有可能

使模板设计者不那么容易发现自己的错误。当你发布了一个模板库，有可能接踵而来的就是客户程序员不能正常编译程序的投诉。

9.7 小结

C++绝对是一门易学难用的语言，这一章的内容对于介绍 C++来说都实在是显得单薄，这里仅仅记录了一些我初学 C++的那段时间看到，想到，悟到的点滴。

而后的漫长时间里，C++反复提醒我，用好它需要一个相当长的时间，如果你或者你的项目组正在开始从 C 转向 C++，我绝对赞同这种做法。但是一定要做好心理准备，学习 C++初期，构建项目不出现设计错误的天才几乎是不存在的。

在学习 C++一段时间以后，反观这些错误，几乎出现在每个人的身上过。尤其是感觉到面向对象设计之优势的初期，想把软件结构设计得更好的念头几乎和出现设计错误的频率成正比。如果项目组里有一个有经验的专家固然好，若是缺少专家，也不应该抱怨 C++本身。

这段时期终归需要面对，早一天犯错误，就可以早一天修正错误。

第 10 章

调　试

这是更偏重于实用的章节。方便软件调试的编程技巧很容易被没有项目经验的新程序员所忽略。

这里讲解了 ASSERT 机制的使用、log 模块的重要性，以及为游戏做一个录像的机制来配合自动测试以及 bug 的重现 。

最后提到的内存检测的类，是延续上一章的代码。完整的代码都在文中列出，甚至可以直接输入软件中使用。

作为程序员，我们的目标首先是，写出无错的程序，然后才是写出高性能的程序。程序跑得再快，结果是错的，性能对用户也就毫无意义了。人脑想的事情多了，各种事物混杂在一起难免会犯错误，而计算机只知道忠实地执行程序指令，并不知道指令背后的逻辑含义，机器很难阻止我们犯错误。

好在现代的操作系统借助硬件可以给代码限制权限，这样在某个权限下，不能干的事情被限制起来，让危害减到了最小。可惜，硬件也只能做到如此，它只能判断单条语句是否合法，而无法理解一组指令的含义。所以，它只能阻止机器干错误的事情，而不能阻止它去干不合逻辑的事。用汇编直接编写软件就几乎只能依靠程序员的天才大脑记住前后的代码关系，不至于把程序写错了。

C 这样的高级语言诞生是一大进步，我们可以让编译器检查语法了。至少语法错误根本通不过编译，不能被运行。这样写错程序的可能性大大降低了，机器至少会按一些基本规则帮我们检查点什么。到了 C++，虽然语言设计者的理念还是相信程序员，但是已经加入了足够多的语法，让程序员可以尽可能多地依赖编译器帮他们发现错误。

毕竟，指定一些不应该被违反的规则在前，然后遵守它们，让机器来监督；比起把规则记在脑海中，以自律来遵守，其效果大不一样。如果规则本身不影响行事的效率，那就更加美妙了。无论是 C 还是 C++都本着这种哲学去设计，无非是在 C++下，程序员可以为自己利用编译器，设定更多的规则而已。

语法检测其实是远远不能保证我们的程序不出错的，大部分的错误出在程序员编写代码的逻辑上。这些只能在运行过程中去监督，程序员有设置这些监督规则的义务，在 C/C++中，就是使用 assert。

10.1 assert 及程序错误的提前发现

assert 是 C 语言中的一个标准，它以函数的形式出现，同时是用一个宏来实现的。assert 被定义在 assert.h 里，如果是 C++中想使

用，通常是在源码前面加上：

```
#include <cassert>
```

看起来这是接受一个 bool 变量为参数的函数，如果参数为真，程序继续运行；如果参数为假，就把这件事情通知给用户，在一些实现中，程序就此结束。

如果把 assert 翻译成中文，就是“断言”意思，写下代码 assert(a>0)从字面上理解含义即“（我）断言（现在）a>0”。这就是给程序的运行逻辑指定一个规则了。

只有 a>0 的条件成立，程序才允许继续运行，否则告诉程序员，程序违反了这个规则，并终止程序等待调试。程序破坏 assert 条件的时候，通常都是程序员设计程序犯了错误。

10.1.1　崩溃的程序不说谎

很多人对自己的程序没有信心，觉得出错是理所必然的事情。所以他们用尽一切办法在错误发生的时候予以挽救。比如下面这个类：

```
class vector_int {
 int *_data;
 int _size;
public:
 vector_int(int size) {
    _size=size;
    _data=new int[size];
}
~vector_int() {
    delete[] _data;
}
int& operator[] (int idx) const {
    return _data[idx];
}
};
```

这用一个 vector_int 类模拟一个 int 数组。嗯，我知道这是一个很不严谨的程序，读者可以从里面找到诸多不合理的设计，比如没有

生成 copy ctor，没有重载 operator=；我们还需要考虑分开提供 const 以及非 const 版本等。

避开这些设计上的问题不谈，这里需要一个足够短小的程序容易帮助讲解问题。

如果这个类真的被人使用，而使用者不去选择直接用 C++原生的数组，我想有一个原因会是做成这样一个类更具定制性。一个经常见到的用法是在 operator[]中加入越界检测。

```
int& vector_int::operator[] (int idx) const {
 if (idx<0 || idx>=_size)
    idx=0;
 return _data[idx];
}
```

这样好吗？的确，使用 vector_int 取代简单的整数数组类型，几乎不可能再出现数据越界破坏掉别的数据的错误了，程序在出错的状态下还可以以最小损失继续运行。

有些很无奈的情况下，这样做或许有它的意义，但是对于严谨的开发者来说。这样让程序用治标不治本的方法自动修正错误，只会种下更大的隐患。因为越界参数的调用，已经说明了程序前面的逻辑发生了错误，隐瞒得越久，越不容易找到真正导致错误的位置。至少，我们应该在检测到错误发生时通知开发人员。

对于开发期，更好的方法是写一个 assert。

```
int& vector_int::operator[] (int idx) const {
 assert(idx>=0 && idx<_size);
 return _data[idx];
}
```

在 assert 中可以直接中断程序，要知道，让程序直接“死”掉，对于开发人员是最好的方案了。一个崩溃掉不能继续运行的程序最好地揭示了程序中存在的问题，并且崩溃点离真正出错的位置越近，越易于修正问题；而用欺骗的手法让程序继续运行，欺骗了机器，欺骗了自己。

10.1.2　为软件维护多个版本

我们知道，Microsoft Visual C++ IDE 在创建新的工程时，都默认创建了两个版本，一个叫做 DEBUG（调试版），一个叫做 RELEASE（发行版）。这两个版本公用一份源代码，逻辑正常的情况下也有相同的行为。只是调试版拥有和发行版不同的编译选项，一般它被设置成不做编译优化，让代码中包含调试信息，方便开发人员做单步跟踪调试，其调用的标准库也会对用户的行为做更多的监控，对不合法的用法做尽量多的警告。实际上，调试版往往会包含更长的不在发行版中被编译的代码，这主要依靠宏和编译期指令来控制。

编译器通常提供一些选项，允许用户定义一些宏，这些定义不需要写在源文件里，而直接通知编译器。一个大家都承认的标准是，对于调试版本，应该定义一个叫做_DEBUG 的宏，而发行版则定义另外一个 NDEBUG。这样我们可以把一些只在调试版中才需要出现的代码写在（#ifdef_DEBUG/#endif）之间。

为一个工程创建不同生成版本，拥有不同的编译选项、不同的预定义宏，并非 VC 的专利，只不过 VC 的 IDE 把这一点做得很好，并让调试版与发行版分开的观念深入人心而已。

如果有一天我们不用 IDE 工作，或者转移到 UNIX 等平台，使用别的开发环境，当新的开发环境并没有为我们自动产生多个版本时，我们也应该主动把这件事情做好。

另外，DEBUG 和 RELEASE 仅仅只是为工程的编译版本起一个名字而已，虽然不值得推荐，我们还是在特定情况下，把 RELEASE 版加上适当的调试信息。

assert 就利用了这一点，它可以在发行版中什么都不干以提高效率，而在调试版中对用户的代码逻辑作出监督，让开发者尽早地发现错误。

有时候，仅有一个标识版本的宏是不够的，我们还可以做更多的针对版本的配置，在这一点上，我个人的建议是把所有的配置信息都写在一个 config.h 里，并要求工程中所有的源文件都第一个包含这个头文件。如果你养成这种习惯，总有一天会受益的。当你想为工程中的所有

编译单元加上点或者修改点什么，都很容易完成。

10.1.3 前置条件和后置条件

我们通常会在一个函数运行前检查一下参数是否都正确，对象的内部环境是否符合要求；在函数功能执行完后，再检查一下是否正确地实现了功能。这是两个逻辑含义上不同的事情，虽然它们都可以用 assert 下断言来完成，但还是推荐使用自定义的 require()和 assure()分开实现。require()在代码中使用得非常的频繁，因为我们不容易确保调用函数之前的所有代码都正确，所以常常在调用参数不正确时做出警告。assure()相比起来就不那么重要，因为函数本身的实现就在附近，程序员很容易确保其正确性。

assure()使用的更多环境是在模板方法中，即，在基类中提供了一个方法，但是算法中间的一个环节调用了一个虚函数，这个虚函数会由派生类去实现。因为基类在设计时很难保证派生类的实现没有错误，所以就可以在调用这个虚函数之后用 assure()确认一下事情是否正确。

尽量多地给函数加上运行时条件检查，就能够尽可能早地发现隐藏的问题。如果你使用了一个第三方库，它频频弹出 assert 窗口，永远不要抱怨。你应该感谢库的设计人员，是他们的尽责，指出了你自己代码中的问题。

或许在发生 assert 的那一刻，你无法发觉附近的程序哪写错了，觉得 assert 毫无道理，甚至希望就地做一些判断阻挠断言的再次失败。哦，那可真是一个危险的念头，真的这样干了，下次发生更为严重的错误时就不会是 assert 窗口的跳出，而是程序默名其妙地崩溃了。

10.1.4 自己实现一个 assert

虽然 assert 是由 C 的标准库提供的，但是我十分推荐实现一个自己的 assert。这样做使得 assert 的行为可以得到更多的定义。

你可以直接用对话框的形式通知用户；也可以把 assert 信息写到记录文件中；甚至可以通过网络发出。除了中断程序，也可以让程序继

续运行，或者直接进入调试状态等等。

assert 的实现也不难，一般是类似定义的：

```
#ifdef_DEBUG
 #define ASSERT(c) if (c) 0; else \
 _myassert(#c,__FILE__,__LINE__)
#else
 #define ASSERT(c) 0
#endif
```

你还可以想出更多更有创意的定义方法，比如 C++中提供了 template，现代的 C++编译器还会对 inline 的函数做优化，有很多方法可以让 assert 不影响发行版的效率。

10.1.5　不正确的 assert 使用

assert 的使用一定不能影响正常的程序逻辑，也就是说，所有的 assert 语句必须是可以被去掉的，在 assert 的条件中改变变量的值都是不允许的。

例如，assert(++a>0)就是错误的写法，因为++a 很可能在发行版中不被运行。

导致断言失败的原因必须是程序的错误，一个典型的错误用法是这样的：

```
printf("are you sure? (Y/N)");
scanf("%c",&c);
assert(c=='Y' || c=='N');
```

程序员企图用 assert 去断言用户的输入，殊不知 c 的值取决于程序使用者的输入而非开发人员所能决定，用户的输入错误只能由真正肯定被运行的代码来检查，并且做出处理。

由于工作原因我曾经接触过某个韩国网游的源代码的一部分，我们需要对其做测试，找出一些潜在的问题。其中一个关于 assert 误用的错误足可让人吐血。

那是一段工作在游戏服务器上的程序，处理客户端发出的数据包，其中一段代码居然用 assert 检查数据包的合法性（._.!）。当客户端程序不写错，理论上是不会导致服务器上这段程序的断言失败的。

这个断言看似可以帮助程序员找出客户端程序的错误，但是，即使所有客户端程序的错误都被找出，在真正运营游戏的时候，我们无法保证玩家不会修改客户端，甚至模拟客户端的行为写出恶意的代码向服务器发出错乱的数据包。

这个时候，如果服务器跑的正是发行版的程序，assert 全部被忽略，这样错误的数据包得不到监督，将很可能导致服务器程序的崩溃。试想一下作为一个网络服务程序，由于一个用户恶意发出一个错误的数据包，就会导致整个服务的停止，是多么可怕的事情。这就是典型的 assert 误用。

10.1.6 为 assert 分级

对于 assert 的使用有一个很大的误解，就是我们应该在调试版本中留下足够多的断言，并在发行版中全部关掉它们。这种论断通常基于几个美好的前提：调试软件的时候，开发人员有比最终用户好得多的机器和耐心，他们不会在意程序运行的多慢。而断言如果要发挥作用，就会消耗一定的代码空间和运行时间，而它们检查的又是绝对不应该发生的事情。一旦我们软件通过了所有的测试，发布给用户用的时候就应该关掉这些断言让程序跑得更快。

可惜这些前提很难真正被满足，assert 有时真的会让程序如蜗牛一样地爬，严重影响我们的工作效率；而另一方面，保证写出一个完全没有错误的程序又是异常困难的。

而且我们并不能保证发行版和调试版逻辑上 100% 一致，测试也会针对发行版本来做，没有断言，一些错误就没有那么快显现出来。

所以，我们应该对我们的断言分级，并且可以选择关掉一部分非常影响效率的部分，或者是在发行版中留下一些不影响效率但很重要的断言。

例如：写一个模拟数组行为的类，就好像本节开头列出的那个 vector_int 一样，在 operator=中断言不会越界的情况非常有用。而标准的 STL 中提供的 std::vector 却没有做这个工作，原因之一就是这会极大地影响调试版的运行效率，我们应该可选关闭它。最简单的 aseert 分级的方案是按影响效率的程度人为分级，这里有一个简单的实现方案：

```
#if defined(ASSERT_LEVEL)
 #define ASSERTP(C,P) \
    if (P>=ASSERT_LEVEL) { \
        if (!C) {    \
            _assert(#C,__FILE__,__LINE);   \
        }   \
    }
#else
    #define ASSERTP(C,P) 0
#endif
```

我们使用 ASSERTP(C,P)来代替 assert(C)，增加一个参数 P 来描述级别。然后在 config.h 中定义 ASSERT_LEVEL 的值来决定开关什么级别的断言。

10.1.7　assert 的引申：检查函数返回值

许多函数在设计时，返回值的含义通常都代表着函数是否成功被执行。检查每个函数的返回值并做出反应看起来是一个好习惯，不过几乎没有人很认真地永远做到这一点，那种程序写起来会看得很糟糕，满篇的 if/else。直觉告诉我们，这样不太符合对代码的审美要求。C++中引入了异常来解决这种事情（C 语言有 setjmp/longjmp），附带提一句，异常安全问题是 C++语言中最复杂的一个问题，如非必要，我不推荐使用异常。

但是，实际上结合上下文，有些函数可以由程序员自己判断，错误的返回是不可能发生的。为了避免我们的判断出了问题，应该对函数的返回值做一个断言。

```
assert(foo()==0);
```

断言 foo()的返回值一定是 0，上面这个写法明显错误，因为 assert 括号里面的东西很可能在发行版不被编译，这种错误甚至在开发期中的很长时间都不被发现（如果开发人员没有测试发行版的习惯的话）。作为一种常见的 assert 用途，我推荐写一个新的宏来断言此事。

```
#ifdef_DEBUG
#define CHECK(LINE,C) {\
    int r=LINE;\
    if (r==C) 0; else \
      printf("%s(%d):%s(%d) return %d", __FILE__, __LINE__,
            #LINE, C,r); \
    }
#else
  #define CHECK(LINE,C) LINE
#endif
```

上面只是做一个示范，定义了一个叫做 CHECK 的宏，它可以记录下 LINE 这条语句的执行结果，并且和断言的结果做一个比较。无论是哪个版本，LINE 都会被编译进执行程序中。这样不会犯上面直接用 assert 检查函数的错误。

还有个小问题，这个宏默认了函数的返回值是 int 类型的，如果是别的类型，可能不太适用，为了解决这个问题，我们可以做一个模板。

```
#ifdef _DEBUG
template <typename T>
class _check_t {
 T _result;
public:
 explicit _check_t(const T& r) : _result(r) {}
 void check(const T&c,const char *file,size_t line) {
    if (_result!=c)
        std:cerr << file << '(' << line << ") : "
                          << c << "!=" << _result << std::endl;
 }
};
template <typename T>
void _check(const T& r,const T& c,const char *file,size_t line)
```

```
{
 _check_t<T>(r).check(c,file,line);
}
 #define CHECK(LINE,C) _check((LINE),(C),__FILE__,__LINE__)
#else
#define CHECK(LINE,C) LINE
#endif
```

这里，_check 接收的参数变成了任意类型 T，而且当返回类型和用户提供的断言类型不同的时候，根本通不过编译。

10.1.8　释放的指针需要置空吗

看下面一个程序片段：

```
class A {
OBJECT *p;
/* ... */
~A() {
    delete p;
    p=0;
  }
};
```

在 A::~A()中，delete p 之后，加上了一句 p=0，这是有必要的吗？

对于正常的程序逻辑来说，这一句没有必要。因为对象析构后，整个对象的数据都是未定义的，正常的程序是不会再访问到其中的变量的。

所以，一个 A 对象析构后，里面的 p 是多少不应该有人会关心。

不过问题就在这里，万一程序出了错误，虽然某个 A 对象被析构，而另一处指向它的指针仍可以工作（这是一个错误），那么一旦我们把 p 设置成空指针。它访问对象中的 p 的时候很可能就导致程序崩溃了。空指针比一个依旧指向可能合法地访问的地址要更容易把错误表现出来。

重视效率的程序员可能会这样写：

```
A::~A()
{
  delete p;
#ifdef _DEBUG
```

```
  p=0;
#endif
}
```

好，这样让发行版中不编译 p=0 这一句了，不过代码变得很丑陋。更好的办法是再定义一个宏：

```
#ifdef _DEBUG
  #define FORDEBUG(LINE) LINE
#else
  #define FORDEBUG(LINE) 0
#endif
```

然后把上面的代码写成这样:

```
A::~A()
{
  delete p;
  FORDEBUG(p=0;)
}
```

这里引出的一个小问题就是千万不要滥用#ifdef_DEBUG 这样的宏指令，这往往是引出问题的根源之一，滥用这些，很容易导致多个版本的行为不一致，错误很难查出。我们应该根据实际想表达的意思分类，把#ifdef 出现的次数降低，在合适的地方还可以利用一些模板的技巧代替频繁使用条件编译。

10.2 让程序更利于调试

一旦程序出错，我们必须找出错误。改进程序的结构设计，仔细地阅读代码，是找到引发错误的代码行的最有效方法。但是，总会有很难通过静态分析代码找到错误的时候，这就需要调试。可惜程序大多是不可逆运行的，程序崩溃的时候，错误已经发生了。

也就是说，我们要么从崩溃点靠大脑回溯执行逻辑，找到真正引起错误的地方；要么想办法再来一次，提前于犯错误的时刻把程序中断下

来，跟踪程序运行的流程，找到不合理的地方。

前面的关于 assert 的小节就是讨论尽量让错误尽早地表现出来，使程序崩溃点离引发错误的代码距离更接近，这样大部分情况我们都可以通过人脑的分析找到出错位置。如若问题得不到解决，就只好想办法可以让错误无限次重现了。

10.2.1　录像

当软件的输入是固定的，例如，读一个文件并做出分析计算输出结果，那么即使有一次运行出错，那么每次运行都一定会有同样的表现。在 Windows 下，系统会被每个进程分配一个“干净”的环境，保证输入不变的前提下，包括所有的内存分配在内的操作，每次都能有相同的结果。

而游戏，往往是没有固定输入的，每次都会因为人的操作的不同而有不同的表现。对于网络游戏，还涉及到网络数据传输这个不定的因素在内。

好在这个不定性是很容易克服的。输入如果需要被准确地再现，就需要在同样的时刻输入完全一样的数据。游戏往往会被限制为一帧帧地运行，那么每个输入的时刻非常适合用帧作为单位，而输入数据无论是鼠标键盘还是网络，都是很容易记录下具体数值的。

理解了这一点，我们在设计游戏软件之初，就应该考虑给游戏增加一个“录像”的功能。即，把所有输入设备的接口抽象出来，真正游戏的时候，会根据当前游戏的运行时刻所在帧，把当前帧内的所有输入数据都记录在文件中。等我们需要重现的时候，不再从具体的设备输入数据，而是从文件在合适的时间，读取曾经记录下的数据，让以前发生过的一切事情忠实再现。

著名的 RTS 游戏“星际争霸”就支持把正常战局记录在文件中，日后可以重播战况。有人会发出感叹，为什么小小的录像文件就可以记录那么长时间的战局，答案很简单，录像模块仅仅只是把通讯中的网络包记录了下来而已，重播录像时，就是用这些记录模拟当初对战时的情况，并没有记录任何的图像信息。这也是新版本游戏往往不支持老版本录像

的原因。

我们在设计游戏的时候，给游戏加上这个模块尤其重要，一旦任何一个测试人员的机器运行游戏时发生错误，至少我们可以重现它的历史，一次调试找不到问题，可以调试几次，错误永远在哪里不会跑掉，只要有足够的耐心，就可以修好每个被人发现的bug。

10.2.2 实现一个log类

写log是古老的调试程序的方法，即使在今天拥有强大功能调试器的时候，它还是非常有效。我们经常需要把一些软件运行的内部状态写出来给开发人员分析查看，这些log信息，或许是直接用printf写到控制台上，或许是在Windows下用OutputDebugString输出到调试窗口，又或者是写到一个本地文件，甚至是把信息通过网络发送出去。

C++环境下，很多人第一反应是实现一个叫做log的流，可以非常柔性地做以上提到的工作。我不太喜欢用内部结构过于复杂的东西，所以还是用的C式的printf格式。

为了效率，log可以被完全关闭不产生任何的代码，所以，基本的log类被写成如下：

```
#ifdef_LOG
  #define LOG_INTERFACE virtual
#else
  #define LOG_INTERFACE
#endif

class log {
  static void create_instance();
  static log *_instance;
  static long _singleton_guard;
  static bool _singleton_destroyed;
public:
  static log& instance();
  LOG_INTERFACE ~log() { _singleton_destroyed=true; _instance=0;}
  LOG_INTERFACE void printf(const char *format,...) {}
  LOG_INTERFACE void dump((const void *buffer,size_t size) {}
};
```

```
#ifndef _LOG

inline log& log::instance() {
  return static_cast<log*>(0);
};

#endif
```

这里为log定义了两个方法，printf输出一串文字，而dump出一块内存的16进制数字。

如果没有定义_LOG这个宏，log::instance().printf()和log::instance().dump()的调用都因为是inline的空函数而被编译器优化掉，打开了_LOG宏后，printf和dump变成了虚函数，log::instance()会去调用log::create_instance()创建log单件的唯一实例。

这份实例可以是log的具体实现，用来干更多有意义的事情。例如，一个写本地文件的log类可以如下所示：

```
class log_file : log {
  FILE *_file;

  log_file();
  ~log_file();
  void printf(const char *format,...);
  void dump(const void *buffer,size_t size);
};

/* ...
   log_file的具体实现略
   ...
 */

log* log::_instance=0;
long log::_singleton_guard=0;
bool log::_singleton_destroyed=false;

void log::instance()
{
  if (!_instance) {
```

```
        create_instance();
    }
    return *_instance;
}

void log::create_interface()
{
thread_guard guard(&_singleton_guard);
if (_instance)
    return;
assert(!_singleton_destroyed);
static log_file obj;
_instance=&obj;
}
```

同理，我们还可以把log_file换成log_network，log_console等。

10.2.3 跟踪代码运行的流程

由于软件规模的扩大，我们无法精确地知道代码运行了哪些分支，没有运行哪些分支，所以想办法把代码的流程在记录文件中展现出来或许很有帮助。

下面一个宏往往能为调试提供方便：

```
#ifdef ENABLE_TRACE
  #define CODE_TRACE log::instance().printf("TRACE: %s (%d)",
                                    __FILE__,__LINE__);
#else
  #define CODE_TRACE
#endif
```

现在我们可以在任何需要跟踪代码流程的地方加入 CODE_TRACE 了。

10.2.4 为内存分配加上一个壳

内存泄露是 C++程序员最容易犯的一个错误，C++里可以灵活使用指针，所以很容易让某些指针指向的对象没有释放掉。直接用 operator

new是检测不出这种情况的，我们需要在内存分配的时候做一些记录。

```
#ifdef _DEBUG
#define DEBUG_MEMORY_ALLOCATOR
#endif

struct memory_cookie;

struct memory_debug {

#ifdef DEBUG_MEMORY_ALLOCATOR
int _id;
long _lock;
int _break_id;
memory_cookie *_list;
memory_debug();
~memory_debug();
#endif

};

class memory : public memory_debug {
memory() {}
void debug_malloc(memory_cookie *);
void debug_free(memory_cookie *);
static void create_instance();
static memory *_instance;
static long _singleton_guard;
static bool _singleton_destroyed;
public:
~memory();
void *malloc(size_t size);
void free(void *p);
void set_break(int id);
static memory& instance();
};

#ifndef DEBUG_MEMORY_ALLOCATOR

inline memory& memory::instance()
{
```

```
    return *static_cast<memory*>(0);
}

inline void memory::set_break(int id) {}

#else

inline memory& memory::instance()
{
if (!_instance) {
    create_instance();
}
return *_instance;
}

#endif

------------------------------

template <int debug>
struct memory_debug_cookie {
memory_cookie *last;
memory_cookie *next;
int id;
};

template <> struct memory_debug_cookie<0> {};

struct memory_cookie : public memory_debug_cookie< (sizeof(memory)
sizeof(memory_debug_cookie<0>)) > {
size_t size;
};

#ifndef DEBUG_MEMORY_ALLOCATOR

inline void memory::debug_malloc(memory_cookie *) {}
inline void memory::debug_free(memory_cookie *) {}

#else

memory_debug::memory_debug()
```

```
{
_id=0;
_lock=0;
_break_id=-1;
_list=reinterpret_cast<memory_cookie*>(smallobject_allocator::
      instance().allocate(sizeof(memory_cookie)));
_list->id=-1;
_list->size=0;
memory_cookie *temp=reinterpret_cast<memory_cookie *>
(smallobject_allocator::instance().allocate(sizeof(memory_cookie)));
_list->last=_list->next=temp;
temp->id=-1;
temp->size=0;
temp->last=temp->next=_list;
}

memory_debug::~memory_debug()
{
smallobject_allocator::instance().deallocate(_list,
                                  sizeof(memory_cookie));
smallobject_allocator::instance().deallocate(_list->last,
                                  sizeof(memory_cookie));
}

void memory::set_break(int id)
{
    _break_id=id;
}

void memory::debug_malloc(memory_cookie *mc)
{
thread_guard guard(&_lock);
assert (_id!=_break_id);
mc->id=_id;
mc->last=_list;
mc->next=_list->next;
_list->next->last=mc;
_list->next=mc;
++_id;
}
```

```
void memory::debug_free(memory_cookie *mc)
{
thread_guard guard(&_lock);

assert(mc->id>=0);

mc->last->next=mc->next;
mc->next->last=mc->last;

mc->id=-1;
}

memory *memory::_instance=0;
long memory::_singleton_guard=0;
bool memory::_singleton_destroyed=false;

void memory::create_instance()
{
thread_guard guard(&_singleton_guard);
if (_instance)
    return;
assert(!_singleton_destroyed);
static memory obj;
_instance=&obj;
}

memory::~memory()
{
if (_list->next==_list->last) {
    log::instance().printf"No memory leak\n");
}
else {
    log::instance().printf("Memory leak:\n");
    memory_cookie *iter=_list->next;
    while (iter!=_list->last) {
         log::instance().printf(buffer,"id=%d size=%d\n",
          iter->id,iter->size-sizeof(memory_cookie));
         iter=iter->next;
    }
}
_singleton_destroyed=true;
```

```
_instance=0;
}

#endif


void *memory::malloc(size_t size)
{
if (!size) {
    return 0;
}

memory_cookie *ret;

size+=sizeof(memory_cookie);

if (size>smallobject_allocator::chunk_limit) {
    ret=reinterpret_cast<memory_cookie*>(::malloc(size));
}
else {
    ret=reinterpret_cast<memory_cookie*>
        (smallobject_ allocator::instance().allocate(size));
}
ret->size=size;
debug_malloc(ret);
return ret+1;
}

void memory::free(void *p)
{
if (!p) {
    return;
}

memory_cookie *block=reinterpret_cast<memory_ cookie*>(p)-1;

debug_free(block);

if (block->size>smallobject_allocator::chunk_limit) {
    ::free(block);
}
```

里面用到了的thread_guard类和smallobject_allocator类都在上一章中提到过，log类在上一节就有列出代码。

```
else {
smallobject_allocator::instance().deallocate(block,block->size);
}
}
```

这里实现了一个memory类，可以用于内存分配的调试。

对于release版，memory类的表现很简单，它仅仅包含了内存分配和释放函数的，作为单件，它什么实例都不需要创建出来，因为对于内存分配释放函数，this指针是无意义的。我们用return *static_cast<memory*>(0);创建出一个假的instance指针，代码就可以工作了。

对于需要做内存调试时，memory对象用一个双向链表记录了所有正在使用的内存块，以及一些额外的调试用信息，并在相应的分配（memory::malloc）和释放（memory::free）中对这些信息进行记录。

为了更加实用，我们用上一章中提到的thread_guard类实现了多线程安全，并对于小块内存，利用smallobject_ allocator进行分配。

用过MFC的程序员对MFC的内存泄露检测印象很深刻，调试运行完毕后，所有的内存泄露都被输出到调试窗口，并且列出了源文件名和行号。

MFC是通过重载全局new和delete，并用宏重定义了new，让用户在调用new时把当前的源文件名、行号传入调试版的内存分配函数实现的。

我们这里当然也可以利用这种方法实现同样的功能，但是我给出的这段代码却没有记录源文件名和行号的行为，为什么？

因为前文可知，如果软件设计得合理，重现任何一次运行的过程都是可行的，所以我们不必记录每次内存分配的源代码位置，只需要记录内存块对应着是第几次内存分配就足够了。如果一块内存块没有被释放，那么只要我们重现这个流程就能知道任何一次内存分配发生在什么地方。

可以看到，每个经过memory分配出来的内存块都加入了一些额外的信息，其中就包括一个id(memory_ cookie::id).memory::_id

在每次分配内存时被累加，然后被赋予新分配出来的内存块的头部信息中。额外的，我们在释放内存时会把memory_cookie::id设置为-1。

这样做的好处是，一处内存被多次重复释放会被很快地查出来。

在程序结束时，memory这个单件析构的期间，根据memory中保留的所有正在使用的内存块链表，我们就可以找到没有释放的内存块及它的id了。

另外，我们还可以写一个函数检查正在使用的内存块的合法性，这通常被用于一些以外越界写数据的操作，最简单的方法莫过于检查所有内存块的id是否被修改成非法的值了。

```
void memory::check_all()
{
memory_cookie *iter=_list->next;
while (iter!=_list->last) {
    assert(iter->id>=0 && iter->id < _id );
    iter=iter->next;
}
}
```

这个函数检查所有正在使用的内存块的id是否在[0, memory::_id]区间内。如果你还不放心这种检查，可以在memory_cookie结构内多放一个标记标量，在分配时赋一个特殊值，释放后清掉。然后memory::check_all()就可以检查所有的标记是否被人篡改了。

在memory中提供了一个set_break方便调试，其作用很简单，设置一个断点在某次内存分配处，一旦某次程序运行发现第n次内存分配没有释放，我们再运行时就可以先调用 memory::instance().set_break(n)；等再次运行到第n次内存分配操作时，断言失败，程序中断下来等待调试。

为了让memory类可以真正工作，我们还是需要重载全局的new和delete，但我不推荐把重载::operator new 和::operator delete 写死在库函数中。因为是否需要重定义内存分配操作应该是由具体项目开发人员决定的，而不应该由库的开发人员决定。

我们在项目的config.h中这样写：

```
inline void* operator new(size_t size)
{
return memory::instance().malloc(size);
}

inline void operator delete(void *pointer)
{
memory::instance().free(pointer);
}

inline void* operator new[](size_t size)
{
return memory::instance().malloc(size);
}

inline void operator delete[](void *pointer)
{
memory::instance().free(pointer);
}
```

并把重载::operator new做成一个可选项。

重载内存分配和释放操作，并不是全为检查内存泄露或越界而写的，某些时候会发生意想不到的用途。比如我们希望在游戏的运行过程中设置一个断点，但没想好设在哪里，只知道大约在什么时间，由于内存分配的操作在C++项目中非常频繁，我们可以改为估计大约在第几次内存分配处中断下来。

这样就可以利用 memory::set_break()完成这个任务了。只要问题能够重现，多试几次，就能把问题发生的时机定位到特定的两次内存分配之间了。

10.3 RELEASE 版本的调试

很多人的程序在调试版中不出错，到了发行版本中就错误连连。这大多是软件设计的问题，但是问题一旦发生，许多程序员束手无策。

其实调试发行版并不可怕，仅需要一些技巧。

（1）打开可以打开的 assert。这需要你有自定义的 assert。可以随意地开关不受版本的约束。

（2）生成 map 文件。即使是优化过的 RELEASE 版本，也是可以生成一份 map 文件的，如果你使用的是 Microsoft Virtual C++，可以在连接选项中找到 Generate mapfile 的选项。它并不往你的 exe 文件中添加任何东西，仅仅只是把编译连接时的所有函数的入口地址记录在一个后缀为 map 的文件中。程序崩溃的时候，都可以得到一个崩溃时的 EIP 地址，通过这个地址，从 map 文件中就可以很容易地查到崩溃所在的函数是哪个了。

（3）RELEASE 版本也是可以设置断点的，当然，因为没有编译进调试信息，直接在 IDE 中设置断点是不可能的。不过我们可以使用一种取巧的方法，在希望设置断点的地方写上__asm int 3。这是一个调试中断，如果在调试环境下，即使是不带调试信息的发行版执行文件，运行到这一句也会中断下来。

（4）熟悉汇编和 C++的对象模型。这需要经验的积累，毕竟看懂 C++编译出来的汇编并可以进行调试不是能速成的技能。但是我可以向你保证，这是可行的。甚至每种 C++编译器有它固有的编译模式，看得多了，可以做到在脑海里从汇编指令原样还原出 C++代码。

（5）使用第三方的调试器，事情或许能变得简单一点。例如 IDA Pro，Soft-ice。

（6）最后，你可以关掉发行版中的一些优化选项，生成调试信息。虽然不被推荐，但可以留作后路。

10.4　小结

作为程序员，我们都不希望自己的代码出现 bug。要避免出错，首先是程序本身的结构设计要好，良好的设计会减少 bug 的产生。然后我们要时刻保留清醒的头脑，写代码时应该警惕那些感觉不好的地方，不能只是因为可以实现而采用味道极坏的形式，从而埋下隐患。

软件的各个模块应该提早积极地做全部的测试，并时刻检查这些测试是否还能通过，把错误的显露控制在代码规模比较小的范围内，避免bug的扩散。

因为人的失误无可避免，对于我们写的程序模块、函数，都尽可能多地加上断言，那些是留在我们不清醒的时候，做出提醒用的。或是作为库的编写者提醒他人。断言使错误提早被人知道，永远不要抱怨一个断言失败对话框的弹出。发现bug应该及时确认修正。

最后，无错程序不可能一次写成，调试代码也是程序员的必修课。为了我们调试的方便，应该做足预备工作。对于输入不定的软件，如游戏，一定要在项目初期，就做出游戏录像的功能，保证日后的bug可以方便地重现。内存分配模块的重载也是检测和修复bug有效的利器，应该充分地利用。

第 11 章

脚　本

嵌入式脚本在现代游戏软件中已经成了必不可少的模块。除了讨论嵌入式脚本本身的优劣和适用范围外，我还选出了 Lua 作为实例介绍给大家。这里可以看到，Lua 是如何嵌入到软件中，并且怎样服务于游戏逻辑的。Lua 作为我的个人爱好之一，我期望通过本章节为 Lua 的一些被人诟病之处平反——许多人认为 Lua 不够面向对象，甚至不能良好地封装 C++ 对象。这里我将给出一段可以使用的代码，展现我是如何做到对象封装的。

很特别的，在本章的后半段提到了网络游戏反外挂的技术。这似乎有点偏题。但是，好的嵌入式脚本是防止反编译游戏逻辑的一个基本前提。我在这里发挥了很多，都是和脚本本身关系不大的。本人觉得，反外挂实在是一个不登大雅之堂的技术，所以不想给它另开辟一章专门介绍了。

11.1 为什么需要脚本

为什么游戏软件需要内嵌一个脚本语言?这个问题我被人问过好多次，也问过自己好多次。看起来，脚本语言能做的事情，C++都能做到，而 C++能做的事情，虽然脚本也能有办法做到，可是效率上相差甚多。为什么要选择一个效率低下的解决方案呢？看起来无外乎以下几点原因。

（1）灵活性。

游戏不比通用软件，需求会随着项目的进展时刻变化，我们必须随时改动游戏中的许多东西。如果在程序中写死这些内容，日后的细微变动都会引起执行文件的重生成，这在项目非常小以及开发人员比较少时，是可以容忍的。一旦项目扩大，试想一下一个上十兆的执行文件的编译过程，仅仅是连接的时间就足够让人回封 E-mail 了。

早期的许多游戏并没有内嵌一个可运行的脚本语言，但至少，它们常常会把一些需要变动的数据写在人可阅读的文件里，这个文件被看成配置文件。可以随时由人手工编辑，而不需要改动执行文件。我们可以将其看做是一个静态数据描述的脚本。后来，这种方法被可运行的脚本语言代码所替代，只被有限地遗留下来。在 Windows 平台下，甚至有处理 ini 文件的 API 直接使用，或者稍微麻烦一些的，我们可以使用 XML，XML 解释器通常不需要自己从头编写。

（2）减少和底层的耦合度。

从上一点延伸，如果游戏中的一些东西经常被修改，那么它们出错的几率就会提高，这些模块应该减少和那些不被经常修改的部分的耦合度，才能保障软件的稳定性。

使用脚本而不是另一个 C++模块，强迫我们实施这种低耦合的结构。而且在脚本解释层很容易做到捕获错误脚本的异常情况，部分脚本的bug不至于让整个程序崩溃。作为开发人员，也容易在扩大游戏设计的规模的同时，维持基本模块的简洁。

让程序尽量不因为 bug 而立刻崩溃，看起来和上一章提到的，尽可能地提前崩溃掉程序的原则相违背。

其实不然，和开发者不同，最终的用户没有修正 bug 的义务，他们需要即使在出错的环境下，也能安全地运行到可以保存游戏记录的时刻。

（3）增加代码复用性。

如果我们把所有游戏逻辑有关的东西都放进了脚本中，那么脚本以下的模块实际上可以被很多游戏项目复用。这是低耦合度带来的好处，如果采用一套 C++代码实现两个不同的游戏，第一个游戏制作的经验可以让我们了解程序能够做到哪些，哪些需要加强，哪些问题应该避免。从脚本层重写出新的游戏不是件难事，而且针对新游戏对 C++层的改动会让两个项目都受益，而不是迫使开发人员分出两个程序版本分开维护。

（4）降低开发门槛。

C++真的是一门极其复杂而容易犯错误的语言。即使你可以熟练地使用，但是也很难保证整个团队的程序员都是 C++高手，这个时候，一个容易学习的脚本语言降低了开发门槛，方便我们往项目中增加人手，而不至于把整个项目搅乱。

如果脚本处理的事务比较简单，我们甚至可以教会不懂编程的策划人员也来参与写脚本，如果你担心非程序员的非计算机思维方式会把程序弄得混乱，也可以选择写一些文本处理工具，要求策划人员按格式填写一些数据，然后用工具把这些格式数据转化为脚本程序。对，这些工具同样也可以用脚本语言来做，通常，脚本语言的文本处理能力都比 C++强。

（5）方便更新。

在游戏机世界中，游戏软件被制作在卡带，光盘中，发行之后就无须也不能打补丁。PC 上的单机游戏，会通过网站发布一些补丁程序，修正初次发行游戏时未解决的 bug。

而到了网络游戏时代，几乎不可能存在不需要打补丁的游戏程序了。不仅仅是 bug 需要解决，还需要按用户的需要增加新功能，让用户永远保留一份新鲜感。这样，如果用脚本语言来编写大部分的游戏逻辑，更新程序就简单得多。甚至可以利用网络直接传输新的脚本到用户的机器

我近年参加了两个网络游戏客户端程序的开发，一个叫做《大话西游 II》，另一个是《梦幻西游》。在后一个游戏《梦幻西游》开发的初期，虽然游戏的玩法甚至界面都和前一作设计得很不一样，只是由于游戏逻辑很好地放在了脚本中去处理，所以两个游戏是公用完全一致的客户端 EXE 文件的。这个做法让新项目进展得很顺利，游戏设计人员不用熬过那段程序不稳定的时期，可以自由发挥制作。维护和测试人员的工作量也轻松了很多。

不过后期，因为需要完成一个构架更牢固，更易于扩展新功能，以及找到了一些新的优化方案可以减少客户端的 CPU 占用率和内存空间，所以我们最终还是重写了整个客户端程序。

新老版本过渡得很好，这得益于两个版本都大规模地使用了内嵌脚本。

运行。

（6）阻碍用户对游戏做逆向工程。

这是在网络游戏年代才特别显露出来的脚本语言的一个重要用途。网络游戏外挂程序盛行，其原因之一在于客户端代码既然可以在用户机器上运行，就一定可以被知道其运行的逻辑。正好像没有不可能被破解的软件防盗版加密方法一样，对客户端程序加壳防止人做反编译和跟踪调试是行不通的。

而如果在程序中广泛地使用脚本，除减轻了开发人员开发的难度以外，同样加大了破解者的阅读难度。脚本通常可以被编译成一些字节伪码，特殊的编码方式没有现成的工具反编译；如果在运行时利用 Soft-ice 等调试工具，观察运行的机器代码，看到的只是脚本虚拟机的运作而非真正的程序逻辑。这就好比把油画贴近鼻子观看，我们只能看到颜料的颗粒，无法看清画的是什么一样。

11.2 应该用脚本做些什么

游戏程序中的脚本应该完成什么样的工作，每个游戏项目的程序开发者们的看法都不一样。有人主张使用一种简单的脚本，读读配置数据，显示一下游戏中人物的对话，控制一些情节点的触发就行了。而另一部分人，把除开图像显示等需要高效的模块用 C++编写后，给脚本留下驱动的接口，干脆所有的并非速度瓶颈的一切都用内嵌脚本实现。

前一种人只为了让游戏中经常变更的部分可以方便地改变而已，这种脚本往往会嵌在游戏制作中的诸如地图编辑器内，可以给非程序员使用。而后一种人把内嵌脚本语言真正当做一种实现游戏的语言，脚本的使用变成了对游戏底层模块的二次开发，这种脚本必须由程序员来编写。在现有的游戏中，脚本语言的使用不一定在这两个极端，脚本的使用程度各不相同。我不能断言哪种方式更好，只是来谈谈自己的一些经验供借鉴。

（1）如果游戏的界面比较丰富，请一定为这个模块设置一个脚本驱

动接口。

实现一个好的游戏界面是一件相当复杂的事情。不要忽视图像化人机交互界面设计的难度，即使有 MFC 那种较为漂亮的框架类库后，对 MFC 了解得不够充分，错误地使用 MFC 把软件界面模块的代码写得一团糟的程序员不在少数。我见过多个独立设计，但都不约而同地模仿 MFC 为游戏设计一套图形界面的类库。采用一种简单的界面描述语言。这种语言用文本来描述对话框的外观，就如同 Windows 的开发环境提供了描述对话框外观的 RC 文件一样。最后用 C++去写每一个游戏里用到的对话框的功能实现。

因为游戏是时常需要更新的网络游戏，经常需要增加新的界面，并对老界面进行修改，这种 C++写对话框的方式让代码迅速膨胀。虽然制作新的对话框有现成的工具，甚至有向导生成代码，但如此多的 C++代码，以及编写这些代码的人并非资深 C++程序员，这直接导致了程序不稳定。

其实作为网络游戏，更加适合把所有的界面逻辑都放在脚本中，这可以让界面剥离出游戏工程。无论是 3D 还是 2D 游戏，只需要写一个不到 100 行的接口，把界面系统嵌到游戏画面中以供显示。当游戏开发公司同时维护多款网络游戏时，甚至可以由同一个小组的程序员专门编写维护界面的脚本代码。

一个典型的公开可以供参考学习的例子是《魔兽世界》。它用 XML 来描述界面布局，用 Lua 来实现界面的功能逻辑。甚至把界面扩展的功能开放给用户自己来完成。这样便有了 cosmos 这样强大的第三方游戏辅助工具，甚至获得了 GameSpy 2004 的最佳游戏工具奖。这充分说明了网络游戏界面可以作为独立的工程来开发。有兴趣的读者可以去 http://www.cosmosui.org 下载阅读源码。

《魔兽世界》的插件采用未编译的 Lua 文本源码进行开发，任何人都可以从已有的插件源码中学到更多，并加以完善。据我所知，有国人同样开发出一套功能不错的《魔兽世界》的界面插件，可惜当我打开 Lua 文件的时候，看到的是经过故意扰码的代码。相比 cosmos 这未免也太小家子气了一点。

（2）网络游戏的客户端网络包的处理应该使用脚本。

对于网络游戏来说，大多数客户端是采用的包驱动的工作方式。即，客户端的行为是根据每个收到的网络包的内容做出反应的。这种包协议几乎会根据游戏需要经常增加新内容。

而处理网络包的内容不会花掉太多时间。客户端的速度瓶颈通常还

是落在显示图像上面。所以我们应该把这个部分放进脚本。

（3）游戏的情节和游戏规则应该用脚本驱动。

如果是网络游戏，有许多规则玩法放在服务器程序内，这些经常会被改动，而网络服务通常会长达一周甚至几周不停歇。二进制可执行代码不方便卸载，而脚本十分利于修改后重新加载。即使是单机游戏，更新不那么频繁，也没有运行时重载逻辑模块的需要，我们同样需要把这些游戏逻辑相关的部分放进脚本。这样在测试阶段会很有效率，而且脚本也利于非程序员根据需要在开发期变动，能够快速看到改动后的效果。

（4）游戏的配置应该写在脚本中。

这里指的是更广义的脚本，有时候配置文件可能仅仅是一些简单的文本数据。应该避免用二进制的数据来描述配置信息，或者用一些除了编写程序的程序员之外谁也不懂的数字，当然直接把配置信息写在代码里面更是下下之选。用一个更具柔性的脚本语言描述配置信息，使之拥有一定的容错性，是个不错的主意。

11.3 脚本不应该做什么

应该说，好的脚本语言，作为一种计算机语言，理论上它可以代替C++做任何工作。唯一让我们拒绝使用脚本的环境只是考虑到了运行效率，在游戏中，这又是不得不考虑的重要因素。

如果你作为一个游戏程序员，给游戏项目添加了一个内嵌脚本，并开始考虑脚本不应该做什么而不是我能用脚本做点什么的时候，相信程序的结构应该是不错的。至少引擎已经做得很结实，这个时候是考虑效率的问题了。

我们需要找到影响效率的热点，通常是一些频繁计算的地方，或者被频繁调用的函数。通常，完成相同的功能，脚本的运行速度会比编译过的C++慢上数十倍，这可能超出许多不太追究效率的人的想像。

但这是事实，即使脚本被“编译”为字节码，运行时不需要做词法分析等字符串处理，脚本语言本质上还是通过解释运行的，一个简单的

数学运算，都会变成解释加处理的过程。脚本通常也不会做 C++编译器做的那些代码优化，它只求代码的运行逻辑正确。对于脚本中的变量，许多脚本的实现依靠的是字符串到对象的映射，处理起来远比编译过的 C++代码一个直接的地址来得低效。

发现脚本已经成为制约速度的瓶颈后，就应该考虑是不是应该把一些代码移出脚本了。

11.4　如何嵌入脚本语言

不少游戏程序员对把一个脚本语言嵌到游戏程序中显得很迷茫。显然，C++和任何一种脚本语言都有不大相同的问题描述方式。怎么把两种东西有机地融合在一起？

有些程序员选择为自己的游戏编写特定的脚本，这种脚本能干的事情可能很简单，只能也只需要完成一些指定的任务。比如只是和 NPC 对话弹出一个对话框而已。这样，脚本和 C 语言编译出的本地代码的交互是单向的，只由本地代码调用脚本。脚本中能做的事情都完全在语言设计时写死。这样做的局限性很大，但也是最容易想到及实现的方案。

如果选择一种通用脚本语言，那么脚本都会提供如何调用 C 函数的接口，我们用 C 语言写出符合脚本调用规则的函数，把这个函数提交到脚本的虚拟机中（通常是把 C 语言函数映射到一个字符串上，虚拟机通过这个字符串可以找回函数地址），在脚本里就可以调用它了。

这里，脚本语言仅仅是一种实现了某种特定逻辑的解释语言而已，它所做的任何事情都依赖这种 C 语言编译出的本地代码。同样，脚本语言还提供了 C 语言访问虚拟机内部数据的接口，可以取出虚拟机内的数据，调用脚本写出的函数，让脚本的全部或部分代码运行起来。

去考虑如何使用脚本，我们就必须考虑脚本在整个游戏程序中占一个怎样的角色的问题。

一种方案是让脚本全权代理游戏中的一切，只让效率关键点回调 C 函数。抛开游戏的范畴，网页浏览器就很类似这个模式。打开一个网页

后，加载全部的页面信息后，当前页的 JavaScript 就开始运行，指导网页表现的逻辑。这种方式比较适合逻辑较为单一的游戏，用一个数据描述段描述所有对象的状态，然后写出每个对象响应各种消息时应该做出的反应的代码，这些消息可以是用户的操作、网络消息，或者是定时器等等。

这是一种典型的消息驱动模式，不太符合我们在 C/C++语言这些更接近机器模型的编程中惯用的循环、分支、循序结构。把脚本的应用实现成这个样子，需要编写脚本的游戏程序员转变一些编程的观念。这种做法不乏成功的例子，这些年很流行的网页休闲游戏网站www.kele8.com就是在自己的游戏引擎中实现了这样一套脚本机制。

让脚本全权代理游戏的整个运行逻辑，而且直接把它作为 C/C++语言的一个延伸，也并非不可能，最大的难点是，游戏通常是以循环结构为主体的，在 Windows 环境下更是如此，因为 Windows 本身需要应用程序循环处理窗口消息。脚本的虚拟机解释运行脚本时却被看做是一个持续的操作，执行完需要执行的脚本操作就退出返回，这和游戏逻辑的循环处理是一个矛盾。

想比较完美地解决这个问题，需要在游戏引擎中实现一个 coroutine（协同例程）的机制，这是 C 语言本身不提供的，就是说，我们要求游戏的底层代码（C/C++书写的部分）和脚本层逻辑代码可以并行运行，这种并行不是同时运行，而是允许底层代码运行完一个阶段后可以切换到脚本层代码运行，脚本运行到一个阶段后可以主动切回，并在虚拟机保留住切出点。等底层再次切回虚拟机内部时，脚本代码可以从上次切出点继续。这样，C/C++编写的底层代码唯一需要的只是启动虚拟机，并循环切入虚拟机中运行脚本代码。

coroutine 在 C 语言中没有被支持，我们自己用汇编实现一个。我在《梦幻西游》中就实现了这样一套 coroutine，可以方便地让若干函数交错并行运行。这并不困难，无非是在函数的某一点希望切出的时候，保留住当前的位置，并跳转到下一个可能的切入点。即使不用汇编，使用 C 语言中的 setjmp()/longjmp()同样能实现，只是效率和调试方面要差一些。由于代码版权原因，在本书中不方便公布我自己实

现的 coroutine 类，不过从网络上利用搜索引擎是可以查到一些前人的实现的。

在 C/C++中实现的 coroutine 最严重的问题是无法监测某个 routine 的堆栈使用情况，堆栈溢出不能得到及时通知，并且 C++的异常机制也是很难和 coroutine 协同工作的。

如果脚本自己支持 coroutine 的机制，允许脚本主动从虚拟机中跳出，那么问题要容易得多。Lua 第 5 版以后就支持这个特性，在下面的章节中，将会展示我们可以如何使用它。

另一种常用的脚本运行机制是把脚本用做单纯的事件处理，游戏程序的主流程函数交给本地代码完成，这是一种比较符合游戏程序员思考方式的做法。一些需要灵活度的地方，不再调用某个 C++模块中的函数，而是驱动一个脚本。

这种做法的推广在网络游戏的服务器端运用得更为广泛一些，历史悠久的 MudOS 中，实现的一套脚本语言就是类似的机制，在接收到每一个网络包的时候，本地代码负责把网络包按类型分发到脚本层中的相应函数驱动游戏逻辑。

而 MudOS 又提供一些读写游戏中数据的接口供脚本调用，并允许脚本发出信息给客户端。虽然 MudOS 是为了实现文字 MUD 而实现的一套引擎，但其脚本系统的工作原理却可以供各种类型的网络游戏借鉴。加上本身公开源代码，经过改动是可以适应图形化网络游戏的。

11.5　选择一种脚本语言

Python

Python 是现在最流行的脚本语言之一，似乎近两年在 C++程序员圈子里，大家都在谈论着 Python。这是一个设计优美的语言，能和 C++结合得很好，解决许多 C++的语言模型中看起来很繁杂的事务。Python 的第三方模块也非常丰富，几乎所有你认为应该存在的模块都有人写出来了。

在我的感觉中，许多 C++程序员似乎很排斥 Java 但却非常能接受 Python。我们可以从 http://www.Python.org 下载到最新版的 Python 源代码以嵌入自己的游戏中。

Lua

Lua 是我见过运行效率最高的脚本语言了，同时也是最轻量的一个。你可以不喜欢 Lua 的语法，可以埋怨它和 C 的接口设计得有些奇怪，更可以指责 Lua 写出来的程序不那么优雅。但是你应该理解它，这些不尽如人意的地方都在为运行的高效率做出贡献。http://www.Lua.org 现在可以下载到第 5 版，正是这个版本提供了许多较之第 4 版全新的特性，让人更加喜爱它。

JavaScript

在互联网络发展的初期诞生于 Netscape 的浏览器中，虽然名字里有一个 Java，但是经过使用，我个人宁愿相信它更接近 C 的风格一些。

JavaScript 也算是一种使用方便，容易被程序员接受的通用脚本语言，Netscape 也公开了它的源码。我们可以在 http://www.mozilla.org 找到名叫 SpiderMonkey 的 javascriptengine。

Java

很多程序员不把 Java 看做一种脚本语言，的确 Java 算不上一种真正意义上的脚本语言，它介于解释语言和编译语言之间，由 Java 的编译器把 Java 源码编译成一种 Java 虚拟机可以识别的二进制代码，再由 Java 虚拟机运行它们。因为 Java 本身提供了本地接口规范（Java Native Interface），Java 和 C/C++的相互调用并非难事，所以我们也可以在游戏中嵌入一个 Java 虚拟机，和游戏底层程序协同工作。据我所知，曾经有一款游戏 Vampire: the Masquerade 就是内嵌了一个 Java 虚拟机的。

我以前对 Java 兴趣不大，这点在近一年有所改变。Java 其实是一种设计不错的语言，并有不错的实现机制。它可以在中间代码级实现多线程，而不依赖操作系统。它的垃圾回收机制非常完善，以至于可以较完

美地解决主动内存管理所必然产生的内存碎片问题。我很希望能找到一个实现得比较好的 Java 虚拟机源码，做一些精简后嵌入到游戏中使用。可惜一直没寻觅到中意之作。

LCC

LCC 或许会是 C 语言爱好者的最佳选择，我曾经是一个纯粹的 C 语言爱好者，但近几年兴趣转移到 C++上更多。LCC 是一个开源的 ANSIC 编译器，它使得你可以把 C 语言编译成多种目标机器上的代码。如果你为游戏设计一个虚拟机，那么就可以改出一个 LCC 的版本生成虚拟机上的代码。有点像 Java 的运行机制，不是吗？

著名的 Quake3 就是这样做的，你可以追随 Carmark 也做一次这样的事情，很有趣的。

CH

如果你以 C 的语法为基础，又想拥有更多的功能，那么 CH 是一个成熟的解释性 C 语言，很适合做成嵌入式脚本。这是一个商业软件，不过可以在一定条件下免费使用，在 http://www.softintegration.com 可以找到下载。

另外还有很多有趣的脚本语言可供选择，比如：

RUBY(http://www.ruby-lang.org)

PERL(http://www.perl.org)

等等。

即使世界上已经存在这么多的脚本语言，依然会有游戏开发者自己开发专有的脚本。实现一个脚本语言并不困难，学过编译原理课程的人应该都可以独立完成。但这种做法不被推荐，因为设计一种通用的语言比实现困难得多，需要设计者本身有丰富的经验。而现有的开源脚本语言，已经可以满足几乎所有的需要了。

抛开嵌入式脚本，直接使用 C++编写 DLL 嵌入或许是一种另辟蹊径的做法。游戏底层程序搭建得好的话，我们可以做到扩展的游戏逻辑层做一些简单的描述就能驱动整个游戏了。

本质上说，这个逻辑层是使用解释性脚本还是使用编译为本地代码的C++并无太大区别。

11.6 实战嵌入式脚本Lua 5

我对Lua有一种特殊的喜好，因为第一次尝试在游戏程序中使用嵌入式脚本就是选择了Lua（虽然后来我所在公司的后继项目全部转向了Python）。Lua非常轻量，整个虚拟机编译出来只有不到100K字节的体积，没有什么多余的东西。它的精练设计使得其工作起来速度很快，例如Lua的操作全部面向操作栈，可以很快捷地复制移动数据；而且Lua的gc机制也并不依赖对象引用计数，这样，虚拟机里的对象复制仅仅做简单的指针拷贝就够了。

在GDC 2004上，有过两个关于Lua的圆桌会议，讨论Lua在游戏界的运用。不约而同的，大家都把话题引向了Lua vs Python。Python也是一个非常不错的嵌入式脚本的选择，Python比之Lua语言更为优雅，可以这么说，Python容易使人写出漂亮的脚本代码，而Lua会引导程序员把代码写得很难看。

但是，优雅是有代价的，我的个人观点，正是这种优雅导致了Python的运行效率大大不如Lua，也使Python的虚拟机过大，已经接近1M。从语言特性或者是易用层面上比较Python和Lua是不公平的。如果你追求高效和轻量，Lua是更好的选择，但是把本地代码封装起来供Lua调用却不容易做得美观，书写Lua程序也非常容易犯错误。好在Lua第5版已经改进了许多，下面我会介绍一些自己的经验。

11.6.1 嵌入Lua

嵌入Lua几乎没有什么难度，只需要把从http:// www.Lua.org下载到的最新版Lua源代码加入你的工程即可。当然，你可以为Lua创建一个静态链接库，然后把Lua.h，lualib.h，lauxlib.h所在的路径

加入工程路径中即可使用。

由于Lua是一套C代码，如果想在C++中使用，需要用extern"C"{}括起这几个Lua的头文件，保证Lua API可以得到正确的原型声明。至于如何在C代码中调用Lua函数，和怎样用C来扩展Lua，在手册中都有详尽的描述。

作为一个有经验的程序员，浏览一遍Lua的手册不会花费超过半个工作日的时间。

使用Lua应该注意一些常见的问题：

（1）Lua是基于垃圾回收机制来管理虚拟机中的对象的，临时构造一个table作为参数传入Lua函数的做法应该谨慎。因为反复地构造临时table会造成垃圾对象的堆积，而过频地触发垃圾回收。

同样，每次产生出一个新的字符串都会在Lua的虚拟机中产生一份新的字符串对象拷贝（而相同的字符串永远在Lua虚拟机中只有一份）。

Lua本身代码的小巧并不代表它运行时不会消耗大量的内存。当然另一方面，也不要因噎废食而过分恐惧Lua会导致无止境的吞噬内存。在实际使用中，Lua的垃圾回收模块工作得很不错。

Lua对底层内存分配API的调用是可以重载的，重定义Lua的内存分配调用而不是直接使用realloc也是我建议大家做的事情。这个工作做起来很简单，找到Lua的源码中叫做lmem.c的文件，打开稍微阅读一下就能明白。

（2）Lua不像C在使用变量前需要预先声明，而且如果不加声明，所有变量都是全局的。由于笔误，两个不相干的变量可能因为同名而出现错误却一时不被发现是很有可能的。

所以函数内的变量千万要注意加上local修饰一下，全局变量如非必要，不要扔在全局表中。应该仿造C++中使用namespace的习惯，放到特定的namespace下。

如果担心一个函数中namespace.value的写法过于频繁影响效率，可以用Localvalue=namespace.value暂时把值取出使用。

（3）从第 5 版开始，Lua_dofile()，Lua_dostring()这样的 API 已经不被推荐使用，而建议统一使用 Lua_load。我们可以写一个 Lua_Chunkreader 从需要的地方加载 Lua 的代码。作为严谨的游戏软件，这些脚本文本至少应该被打包放进数据文件中，而不要让它们散落在硬盘上的小文件中。

（4）Lua 是基于垃圾回收的，所以并没有删除对象的指令。不过，如果我们知道一个对象不会再使用，应该把它设置为 nil。即，从 Lua 虚拟机中的全局表遍历下来，不可能再访问到它了。这样当垃圾回收过程启动时，这个对象将被真正清理出内存。

11.6.2 使用 coroutine 驱动游戏逻辑

Lua 5 提供了 coroutine，可以利用它来简化游戏逻辑的设计。下面看一个简化过的例子：

```
/---------- example.Lua ------\

1. function main()
2.  local yield=coroutine.yield
3.  print("fragment 1")
4.  yield()
5.  print("fragment 2")
6.  yield()
7.  print("fragment 3")
8. end

9. return coroutine.create(main)

\-----------------------------/

/-------- example.c -------------\

1. #include "Lua.h"
2. #include "Lualib.h"
3. #include "lauxlib.h"
```

```
4. int resume(Lua_State *L)
5. {
6.  int ret;
7.  Lua_pushvalue(L,-2);
/* Lua stack: ( ... , coroutine.resume[function] ) */
8.  Lua_pushvalue(L,-2);
/* Lua stack: ( ... , coroutine.resume[function] , co[thread] ) */
9.  Lua_call(L,1,1);
            /* Lua stack: ( ... , result[boolean] ) */
10. ret=Lua_toboolean(L,-1);
11. Lua_pop(L,1);         /* Lua stack: ( ... ) */
12. return ret;
13.}


14.void main()
15.{
16. Lua_State *L=Lua_open();

17. Luaopen_base(L);      /* Lua stack: ( coroutine[table] ) */
18. Lua_pushliteral(L,"resume");
        /* Lua stack: ( coroutine[table] , resume[string] ) */
19. Lua_gettable(L,-2);
/* Lua stack: ( coroutine[table] , coroutine.resume[function] ) */

20. Lua_dofile(L,"example.Lua");
/* Lua stack: ( coroutine[table],coroutine.resume[function],co[thread]) */

21. while (resume(L)) {
22.     puts("now in C");
23. }

24. Lua_close(L);
25.}

\---------------------------------/

/-------运行结果---------------\
fragment 1
now in C
fragment 2
```

```
now in C
fragment 3
now in C
\---------------------------/
```

我们在 example.c 中使用嵌入脚本 Lua，首先在 Lua 的堆栈中压入 coroutine.resume 这个 function 供后面取用。在加载并执行了脚本 example.Lua 后，脚本返回了创建出来的 coroutine。

注意： Lua 和 C 的接口都是基于 Lua 的堆栈的，这种接口没有 Python 那么易于理解，但是巧妙应用 Lua 的堆栈可以得到高性能的 C 同 Lua 之间的数据交换能力。

这里的例子用注释写明了每个 Lua API 调用后堆栈的情况，方便初学 Lua 的程序员理解。虽然有人为 Lua 的接口书写了 C++的封装。但是我认为，想获得高效的 Lua 性能，还是熟悉 Lua API 更加重要。比如这里我们将 coroutine.resume 函数提前放到堆栈中，每次使用前都用 Lua_pushvalue 复制一份，比每次用字符串从虚拟机中检索出这个函数要快得多。

example.c 的第 21~23 行模拟了一个典型的游戏程序的运行框架，也就是一个循环结构。通常我们会希望游戏的逻辑（这里是放在脚本中的）运行完一帧，可以固定地循环地做一些事情。比如处理 Windows 消息。我们看到 example.Lua 这个脚本中，主函数 main()被用 coroutine.yield()切成了 3 个运行片段。这三段是循序执行的，但在 C 中是以一个循环结构来处理它们。

11.6.3 封装 C++中的对象

Lua 看起来太简单，更像是一个过程式的语言。所以大部分 Lua 脚本做的事情都很简单，且代码不够优雅，这也是被 Python 爱好者百般抨击的一点。实际上，我们是可以让 Lua 表现得更面向对象一些，并在 Lua 中对 C++对象封装得更漂亮一点。这需要 C++程序员的努力。下面来看

一下，我的方案，注意，这需要 Lua 5 中 metatable 的支持。

```
class object {
 int data_readonly;
 int data_readwrite;
public:
 object();
 virtual ~object();
 static object* creator() { return new object; }

 int get_data_readonly() const;

 int get_data_readwrite() const;
 void set_data_readwrite(int);

 void do_something();
};

class sub_object : public object {
 int sub_data;
public:
 sub_object();
 ~sub_object();
 static object* creator() { return new sub_object; }

 int get_sub_data() const;
 void set_sub_data(int);

 void do_sub();
};
```

这个叫做 object 的类有一个可读写的属性 data_readwrite，和一个只读的属性 data_readonly，还有一个叫做 do_something 的方法。

现在让我们把它封装为一个 Lua 对象，这个对象可以读写属性，运行对象的方法，我们还希望它看起来像从 C++类中继承而来的，可以在脚本中添加新的东西。

另外，还会有许多继承 object 而来的派生类，例如上面看到的 sub_object，它增加了新的属性 sub_data 和方法 do_sub。我们希望尽量简化 Lua 到 C 的接口，采用一个类工厂的模式在 Lua 中创建出这

些C++对象。

最后，可以设想一下封装后的脚本的样子：

```
obj=create_object("object")    -- 创建一个 object 对象
data=obj.data_readonly         -- 读取 object.data_readonly
obj.data_readwrite=100         -- 写入 object.data_ readwrite=100
obj.do_something()             -- 运行 object.do_something
obj.release()                  -- 删除 C++ 中的 object 对象

obj=create_object("sub_object")--创建一个 sub_object 对象
data=obj.data_readobly         -- 读取 object.data_ readonly
obj.sub_data=100               -- 写入 object.sub_data=100
obj.do_something()             -- 运行 object.do_something
obj.do_sub()                   -- 运行 object.do_sub
obj.new_data=200 -- 添加 new_data=200 到 obj 中 (脚本创建的成员变量)
obj.foo=function (a) print(a) end  -- 添加一个新方法 obj.foo
obj.foo(100)                       -- 运行 obj.foo(100)
```

由于要解决继承和类工厂的问题，我们先定义一些结构：

```
typedef object* (*creator_function)();

struct Lua_function_table {
 creator_function creator;
 Lua_CFunction get;
 Lua_CFunction set;
};

struct Lua_object {
 object *obj;
 const Lua_function_table *lft;
};
```

为每个C++类生成一个Lua_function_table结构，类工厂会利用这个结构创建出需要的对象来。每个C++对象再利用Lua_object结构映射到Lua中去。

```
int create_object(Lua_State *L)
{
const char *classname=Lua_tostring(L,-1);
Lua_settop(L,0);
```

```
const Lua_function_table *lft=find_class(classname);
if (lft==0) {
    return 0;
}

Lua_object *obj=new Lua_object;
obj->obj=lft->creator();
obj->lft=lft;

Lua_newtable(L);
Lua_pushlightuserdata(L, obj);
Lua_rawseti(L,-2,0);

Lua_pushliteral(L,"object_metatable");
Lua_gettable(L, Lua_GLOBALSINDEX);
Lua_setmetatable(L, -2);

return 1;
}

int release_object(Lua_State *L)
{
Lua_rawgeti(L,Lua_upvalueindex(1),0);
Lua_object *obj=static_cast<Lua_object *>(Lua_ touserdata(L,-1));
delete obj->obj;
delete obj;
Lua_pushnil(L);
Lua_rawseti(L,Lua_upvalueindex(1),0);
Lua_settop(L,0);
return 0;
}

int get_property(Lua_State *L)
{
Lua_rawgeti(L,1,0);
Lua_object *obj=static_cast<Lua_object *>(Lua_touserdata(L,-1));
if (obj==0) {
    Lua_pop(L,1);
    Lua_rawget(L,1);
    Lua_replace(L,1);
```

```
    Lua_settop(L,1);
}
else {
    obj->lft->get(L);
}
return 1;
}

int set_property(Lua_State *L)
{
Lua_rawgeti(L,1,0);
Lua_object *obj=static_cast<Lua_object *>(Lua_ touserdata(L,-1));
if (obj==0) {
    Lua_pop(L,1);
    Lua_rawset(L,1);
    Lua_replace(L,1);
    Lua_settop(L,0);
}
else {
    obj->lft->set(L);
}
return 0;
}

int object_do_something(Lua_State *L)
{
Lua_object *obj=static_cast<Lua_object *>
                (Lua_ touserdata(L,Lua_upvalueindex(1)));
obj->obj->do_something();
Lua_settop(L,0);
return 0;
}

int sub_object_do_sub(Lua_State *L)
{
Lua_object *obj=static_cast<Lua_object *>
                (Lua_touserdata(L,Lua_upvalueindex(1)));
sub_object *s=static_cast<sub_object *>(obj->obj);
s->do_sub();
Lua_settop(L,0);
return 0;
```

```
}

int object_get(Lua_State *L)
{
const char *key=Lua_tostring(L,2);
Lua_object *obj=static_cast<Lua_object *>(Lua_touserdata(L,-1));

if (strcmp(key,"release")==0) {
    Lua_settop(L,1);
    Lua_pushcclosure(L, release_object, 1);
}
else if (strcmp(key,"do_something")==0) {
    Lua_pushcclosure(L, object_do_something, 1);
    Lua_replace(L,1);
    Lua_settop(L,1);
}
else if (strcmp(key,"data_readonly")==0) {
    Lua_settop(L,0);
    Lua_pushnumber(L,obj->obj->get_data_readonly());
}
else if (strcmp(key,"data_readwrite")==0) {
    Lua_settop(L,0);
    Lua_pushnumber(L,obj->obj->get_data_readwrite());
}
else {
    Lua_pop(L,1);
    Lua_rawget(L,1);
    Lua_replace(L,1);
    Lua_settop(L,1);
}
    return 1;
}

int object_set(Lua_State *L)
{
 const char *key=Lua_tostring(L,2);
Lua_object *obj=static_cast<Lua_object *>(Lua_touser data(L,-1));

 if (strcmp(key,"data_readwrite")==0) {
obj->obj->set_data_readwrite(static_cast<int>(Lua_ tonumber(L,3)));
 }
```

```
    else {
        Lua_pop(L,1);
        Lua_rawset(L,1);
    }
    Lua_settop(L,0);
    return 0;
}

int sub_object_get(Lua_State *L)
{
    const char *key=Lua_tostring(L,2);
    Lua_object *obj=static_cast<Lua_object *>(Lua_touser data(L,-1));
    sub_object *s=static_cast<sub_object*>(obj->obj);

    if (strcmp(key,"sub_data")==0) {
        Lua_settop(L,0);
        Lua_pushnumber(L,s->get_sub_data());
    }
    else if (strcmp(key,"do_sub")==0) {
        Lua_pushcclosure(L, sub_object_do_sub, 1);
        Lua_replace(L,1);
        Lua_settop(L,1);
    }
    else {
        return object_get(L);
    }
    return 1;
}

int sub_object_set(Lua_State *L)
{
    const char *key=Lua_tostring(L,2);
    Lua_object *obj=static_cast<Lua_object *>(Lua_touserdata(L,-1));
    sub_object *s=static_cast<sub_object*>(obj->obj);

    if (strcmp(key,"sub_data")==0) {
        s->set_sub_data(static_cast<int>(Lua_tonumber (L,3)));
    }
    else {
        return object_set(L);
    }
```

```
 Lua_settop(L,0);
 return 0;
}

const Lua_function_table * find_class(const char *classname)
{
 if (strcmp(classname,"object")==0) {
     static Lua_function_table lft= {
         object::creator,
         object_get,
         object_set,
     };

     return &lft;
 }
 else if (strcmp(classname,"sub_object")==0) {
     static Lua_function_table lft= {
         sub_object::creator,
         sub_object_get,
         sub_object_set,
     };

     return &lft;
 }
 else {
     return 0;
 }
}

void init(Lua_State *L)
{
 Lua_pushliteral(L,"object_metatable");
 Lua_newtable(L);

 Lua_pushliteral(L,"__index");
 Lua_pushcfunction(L, get_property);
 Lua_settable(L,-3);

 Lua_pushliteral(L,"__newindex");
 Lua_pushcfunction(L, set_property);
 Lua_settable(L,-3);
```

```
    Lua_settable(L, Lua_GLOBALSINDEX);

    Lua_register(L,"create_object",create_object);
}
```

最后一个函数 init 可以把 create_object 注册到 Lua 的虚拟机中。调用完 init 后，在脚本中就可以自由地使用 C++中的 object 类和 sub_object 类了。

上面这段程序代码写得并不算好，细节上有许多可以改良的地方。比如在 create_object 和 object_get，object_set 等函数中都使用了很难看的 if/elseif/else 结构，并且使用了 strcmp 逐个比较字符串。这些是可以改进为字符串的 hashmap 的。而且，object_metatable 和 create_object 都被直接注册到 Lua 的全局表中，这是一个很糟糕的用法，当脚本变得很大时，容易引起变量名冲突。诸多改进留给读者研究完 Lua，真正想把 Lua 嵌入自己的游戏项目中时自行完成。

这里的代码，实际上揭示了活用 Lua 5 中的 metatable，可以把 C++对象很好地封装起来。为 Lua 写 C 扩展接口也有很多技巧需要掌握。

11.6.4 回调机制

上面我们已经看到了，在 Lua 中，可以通过 lightuserdata 或者 userdata 来映射到 C++对象上。但是 Lua 没有直接从 C++对 Lua 对象的引用的机制。这是由于 Lua 采用的是垃圾回收的策略，一个 Lua 对象甚至可能在垃圾回收发生后改变在内存空间中的物理位置，我们不可能直接保存 Lua 对象的指针。

所以，即使我们可以记录下 Lua 中对象的物理上的指针（使用 Lua_topointer），但 Lua 却不提供任何方法把这个指针还原成 Lua 对象。

大多数情况下，我们并不需要这种 C++对象对 Lua 对象的引用，可惜有一种情况例外。那就是我们使用 Lua 驱动 UI 模块时。

图形 UI 系统的底层通常会用 C++代码实现，那么在用户操作鼠标对界面上的元素做出某些操作时，UI 模块管理的 UI 对象首先获得了消息，这个 UI 对象肯定是用 C++实现的，并在 Lua 的虚拟机中有一个 Lua 对象对应着。

这个时候，就需要利用 LuaL_ref 等一系列的 Lua API 了。其原理就是在 Lua 的虚拟机中单独开列一块空间把 C++中需要引用的 Lua 对象记录在里面，这样在垃圾回收启动的时候也不至于把这些对象回收掉，从 Lua5 开始，这些对象被推荐记录在 Lua registry table 中，这块区域是 Lua 代码本身无法访问到的，这避免了 Lua 脚本代码的误操作破坏掉这个表。具体的使用方法是：

```
int r=LuaL_ref(L, Lua_REGISTRYINDEX);
```

这样为栈顶的对象在 registry table 中创建一份引用，并返回一个引用 id。如果以后 C++的代码中想找回这个对象，可以使用：

```
Lua_rawgeti(L,Lua_REGISTRYINDEX,r);
```

把对象重新压回栈顶。

当 C++不再需要使用这个 Lua 对象后，可以使用

```
LuaL_unref(L,Lua_REGISTRYINDEX,r);
```

取消引用。

在 Lua 4 以前的文档中可以找到三个更直接做这个事情的 API：

```
intLua_ref(L,lock);
intLua_unref(L,ref);
intLua_getref(L,ref);
```

分别对应于上面提到的写法。这或许是早期的 Lua 在设计时专门为引用 Lua 对象做了特殊的设计，而到了 Lua 5 以后，为了简化内核，把这部分删除了。

不过上面这三个 API 作为兼容的目的还以宏的形式存在于 Lua 的头文件中，只是从文档中消失了而已。

我见过许多高手，他们的编程能力，设计软件的能力都很强。但是在某些方面却比不上一个刚学编程几年的学生，那就是不会写有丰富界面的 Windows 软件，不懂得如何使用 MFC。同样，还有好多游戏程序员自己设计开发中间件，自己实现一套脚本语言，甚至自己做了一大套 STL 里已经实现了相同功能的库。仿佛除了语言和开发平台提供的 API 之外，所有的代码都应该自己完成。

其实，学习已有的库和成熟的代码和学习语言本身同样重要，或许已经成型的类库由于历史原因有这样那样的缺陷，但充分了解它们却可以规避这些不足之处。

了解别人的代码比自己编写代码更加困难，这不应该是真正的高手应该惧怕的。复用别人写出来的代码不等于自己写不出来，而是在了解了前人的工作后，可以继续做得更好。

时至今日，游戏程序已越来越多地把大部分代码放在了嵌入式脚本中完成，掌握一种通用的嵌入式脚本对于游戏程序员变得非常重要。

脚本编程的思维方式和 C/C++不大相同，这是 C/C++程序员必须转变的思维。想让嵌入式脚本发挥最大的效率，每个游戏程序员都有必要足够熟悉嵌入式脚本本身的实现。

开发一个普适的脚本语言成本已经越来越高，对于有能力自己实现一个的人来说，使用一个现成的完善的脚本语言仍旧是更好的选择。Lua，Ruby，Python 这些开源的脚本语言使我们拥有了很好的条件，无论使用哪一个，尝试一下，静下心从源码阅读开始。

11.7 反外挂技术

外挂泛滥是网络游戏现今面临的巨大考验。本来反外挂是一个不登大雅之堂的技术，我这几年由于参与几个网络游戏的运营，很无奈地花了许多的时间在这上面。我们公司自己的网络游戏经营了几年，一直都没有太过分的外挂，我们的努力也算是有所成效吧。

首先，网络游戏在设计的时候应该尽量杜绝把关键的计算放在客户端，也就是说，服务器应该对所有客户端数据采取不信任的态度，玩家机器上安装的程序只是反映玩家的操作的话，模仿客户端的正常动作却发出合法客户端所不能发出的信息用以作弊的方法就不攻自破了。剩下来的问题就在于怎样防止外挂程序帮助玩家做出超过人做到的指令频率，让玩家更快更简便地在游戏中完成任务升级这些破坏游戏平衡性。

11.7.1 防止加速器

Windows 系统是以一个内部时钟来调控各个应用程序的。用户完全可以写一个程序用来欺瞒操作系统，让程序假象中的时间变快。由于在 Windows 中，程序并不能直接物理上去访问时钟的硬件接口，而必须通过操作系统。那么接管 Windows 的时钟模块，并按自己的需要调整让程

序跑起来比实际上看起来的正常速度快或者慢并非难事。

当然，要让程序跑得比正常得快，需要程序设计本身留有余地，PC 平台的硬件性能参差不齐，一个游戏需要在大多数机器上都跑得流畅的话，那么相对好的硬件上，必然就允许运行得更快。

完全检测出操作系统中是否有其他程序修改了时钟不是件容易的事情，因为用户的机器的控制权归根到底在用户的手上，而不是在一个游戏客户端程序上。

但是我们却有一个最大的优势，就是可以在玩家自己的机器之外找到一个时间的参照物，那就是服务器的时钟。

最简单的方法就是从游戏登录之初就定期把游戏本地的时刻发到服务器，虽然有网络延迟的因素，以及用户机器的时钟不可能完全和服务器相等的可能性，但每次对时的时间差却是骗不了人的。如果游戏被加速，用户本地传递过去的时间差就会远远大于真实的时间(服务器时间)。

当然这个方法基于客户端程序以及通讯的数据包不被人恶意修改，只适合预防普通的加速器程序。

11.7.2　防止分析伪造网络包

网络游戏的架构中，必不可少的是让用户的机器上安装一份客户端程序，以此和服务器通讯。那么所有玩家做出的操作，最终都一定会以数据流的形式发给服务器。对于服务器/客户端的这种结构，显然最薄弱易被人监听篡改的部分就是网络传输中的数据包了。

在 Windows 环境下，游戏开发商为了客户端程序适应各种不同的硬件，只能利用合法的手段做网络通讯操作，即调用系统提供的网络通讯的 API。而机器毕竟在玩家手中，必然能使用许多的方式截获这些接受或发出的数据包。

固然一些简易的手法拦截游戏数据包能够被客户端程序检测得到，比如简单地对 socket API 做钩子的做法就可以通过再次由客户端程序注册更新的 API hook 来检测，但这些方法都是治标不治本。玩家的机器在玩家手上，客户端程序只能获得非常有限的机器控制权，不可能根绝玩

家安装外挂程序，所以唯一可行的方案是对网络包数据本身进行加密。

加密算法本身的选择是需要谨慎为之的，首先必须是对称算法，加密和解密速度都需要很快，然后不能消耗太多的内存，最后一点是不要产生过大的数据冗余。

服务器程序需要同时应付上千个连接，如果花在加解密上的时间太多，会大大增加服务器成本；数据冗余过大的话，无端增加了通讯带宽（网络游戏的数据包大多不大，只是几个操作指令，这样几个字节的冗余都会显得太大）。

符合要求的算法并不难找，如果研究过数据加密技术的话，就会发现，其实大多数加密算法本身并不复杂。很多数十年前设计的算法，比如DES，至今仍被使用并认为有效，DES在设计之初就要求使用经济的电子器件来实现。

实际上，DES就是基于数据分组后位交换的。或者在网络游戏中需要一个更为经济的算法，那么你可以做一个将数据包里数据利用密匙（一个握手时传入的数据）打乱的算法。和DES不同，我们完全可以舍弃防止人家黑盒分析的高强度数学方法，而采用一些更简易的算法，至少，不用对数据做位操作了，针对字节的操作速度要快得多。

加密算法的简化也不是可以任意为之的，如果仅仅是对数据包循环和一个密匙做异或操作就太弱不禁风了。那是把外挂制作者当傻瓜，这种太低强度的加密方式，多截取一些固定的数据包，加以推测就可以解开。如果我们换一种思路，先写一个伪随机数发生器，把种子作为密匙在握手时传入，初始化这个随机数数列，然后从数列中依次取出数据和网络数据包的数据逐个异或，黑盒破解的难度就大了许多。

数据加密方法有很多，游戏程序员尽可以多读一些相关专著，就算是耐不住性子做学问的人也可以去把好多年前就翻译出版的《应用密码学》当做工具书翻翻。

查查需要的加密算法并编程实现花不了几天时间。加密算法的难度适可而止就行，只要推测外挂制作者不可能在短时间，通过简单截取数据包对密文分析推导出加密算法，就够了。多半在这种黑盒分析尝试遇到挫折后，有反编译能力的外挂制作者就会尝试对游戏客户端做逆向工

程了。

如果还有人孜孜不倦分析，企图解开密文。我们的游戏开发维护人员尽可以每年更换一次加密算法，直到游戏生命结束。

11.7.3　防止逆向工程

以上提到的两点，都基于外挂制作人员不去分析并修改游戏厂商发布的客户端程序的基础上。事实上，这点无法保证。程序一旦到了用户手上，根本无法阻止用户的修改。即使有法律的制约，那些和用户签订的禁止对游戏客户端做逆向工程的协议在现阶段也是无法履行的。我们还是要靠技术手段来解决。

前些年，许多韩国进口的网络游戏普遍采用一种叫做 N-Protect 的程序对用户机器做监控，这种程序自己可以在线升级，并在游戏运行时检查用户的运行环境中是否有非法程序，有无对厂商发布的客户端程序做了修改等。

理论上，这种方案就是自欺欺人，N-Protect 本身也是跑在玩家机器上的应用程序，和游戏客户端本身可以被修改一样，它无法保证自己不被篡改和欺骗。事实也无数次地证明 N-Protect 这种软件，在“聪明智慧”的中国玩家面前，不堪一击。

网易公司曾经代理过一款韩国网络游戏——精灵。由于程序设计极不合理，导致了外挂泛滥成灾。最后无法从程序自身杜绝，公司被迫考虑购买使用 N-Protect 软件的使用执照。那个时候，N-Protect 出现的时间并不长，口碑还没有现在这么糟糕，公司就这个问题咨询过包括我在内的几个技术人员。我们一致表示，这笔钱花出去，不会有任何效果。最后的决定是和金山合作开发一款类似的软件，如果有效果，还可以进一步推广使用。

金山的开发人员最终长驻网易广州的开发基地做这款软件的开发，最终的结果可想而知，和 N-Protect 一样以失败告终。遇到的问题无非几个：

（1）对程序加壳和反跟踪技术是无法完美做到的。

（2）检查游戏客户端的完整性的代码很容易被欺骗，因为作者可以在系统中另起一个进程，把合法的程序代码复制进去，然后修改监控程序，让它检测伪造的数据，这样无论用什么方法校验都会得到正确的结果。

（3）检测结果无法正确地通知给服务器，最终作弊者可以找到监控程序判断是否合法的条件分支处，并篡改检查结果，这样无论做出怎样的检查结果，都有正确的结果发向服务器。

或许以上一点当年我们遇到的问题，可以由更高明的编程手段和精巧的检测手法改进，但我个人始终不认为监控玩家机器的方法能有效防止外挂。

如果我们回过头来看，外挂制作者为什么要对客户端做逆向工程？反编译和动态跟踪只是一种手段而已，目的在于理解游戏制作人员的程序设计意图，或模仿之编写一个程序替代合法的客户端；或对客户端做一些小修改，完善一些“非法”的用途。既然我们无法通过对程序加壳的方法阻止人家阅读代码，那么就可以加大程序阅读难度的角度入手了。

我绝对反对，因为要反外挂而把程序的结构弄得很糟糕。

其实，我们尽可以采用一些更专业的设计模式。在熟悉模式的程序员眼里，源码变得清晰而不是混乱。对于编译成目标汇编码的执行文件，因为程序本身的模式和机器模式相差较远而难以阅读了。

我个人由一些个人的业余爱好所致，对于反编译一些有趣的软件这种事情也偶尔为之。干得多了，甚至可以做到从二进制代码中90%地还原出C++源代码。达到这个水平，需要对C++本身的编译模式非常熟悉，并且需要了解各种不同的编程手法，并非一个初窥门径的程序员能简单做到。在此同时，也深刻了解哪些代码易被反编译，哪些会让逆向工程者头痛。

这种经验不太好总结，不过可以给出一个有趣的事实：过度使用那些C++高级特性，尤其是过度利用template让编译器做编译时推导的代码，最让看汇编的人头痛☺。

希望这不至于成为游戏程序员滥用C++高级特性的理由。

从做逆向工程人的角度来看，他们面对的是汇编代码，是最底层的实现，所以一般都是接触到一些枝梢末节的东西，对于一些耦合度很低

（恰巧是软件设计良好的特征）的模块之间最容易动手脚。所以比较好的防反编译解读的方案是降低程序实现时在接近机器最底层地方的模块耦合度。最容易想到的方案是设计一个虚拟机，让程序逻辑运行在虚拟机上，这样，实现逻辑层的模块划分都是建立在虚拟机一层而不是实际机器。

这就好比，想弄清楚一部机器怎么运转，工程师可以把机器拆成零件去研究。但是进一步把零件细分到分子原子结构后，对研究工作就有害无益了。

可以想像，不懂 Java 虚拟机中的伪代码含义的破解人员，以跟踪 Java 虚拟机的方式，阅读那些虚拟机层次的汇编代码，想弄清楚程序具体的逻辑会是多么无助。

同样，我们可以用脚本语言来完成程序的逻辑，比如 Python，Lua 等等。需要提醒的是，诸如 Python 这样的开源脚本语言，甚至本身就提供了反编译用的模块，想把代码安全性建立在 Python 的虚拟机不被破解的前提上是很可笑的。作为对策之一，我们可以修改 Python 的源码，做出一个孤版的 Python。有研发实力的开发公司，甚至自行开发一种脚本语言，不对外公布，以保障自行开发的游戏的代码安全性。

如果我们用脚本本身来编写数据包加解密模块，解密难度是很大的。只是为了效率起见，算法本身最好能用高效的 C 代码甚至汇编代码实现一份放在服务器上使用。

另外需要注意的是，尽可能地不要在内存同时留下完整的明文数据包的痕迹。因为外挂制作者在毫无头绪的情况下，都是通过假设内存中有一段明文数据，比如用户输入的一串聊天字符串，动态暴力搜索内存得到这串数据的物理地址，然后反向找到处理这些数据的位置的。

如果没有这些可供参考的数据点，全部反编译跟踪整个的客户端程序的程序就过于巨大了。同样的道理，socket API 调用的位置也往往是跟踪的切入点，我们在设计通讯模块的时候，应该合理设计，让实际处理网络数据的地方远离这些敏感的跟踪点。

11.7.4 防止代码钩子

如果客户端程序被部分解读，外挂最简单的做法就是在已知含义的代码上做上一些钩子，让外挂的代码可以钩接在里面运行。或者自行调用一些合法的客户端代码发送数据。这种方式也是很容易预防的。

因为逆向工程者永远面对的是局部汇编代码，而不是全部。一些模块在逻辑策略上是顺序调用的，例如反映一个客户端操作的消息包就存在构成、加密、发送三个大步骤，里面还能细分成更多小步骤。或许加密这个步骤是允许单独调用成功的，外挂程序不需要了解加密模块具体怎么运作，找到模块进入点调用即可。

但作为游戏程序设计者，完全可以另用一个模块检查整个流程是否被完整地走过。这个检查过程可以隐藏在每个小模块里面，即使是一个简单的计数器，做逆向工程的人也不是轻易可以发现的。

当然一旦外挂开发人员读懂了所有的代码，这些小伎俩会看起来很可笑。但从汇编的级别，通篇理解全部的代码可能吗？看看一个网络游戏的开发成本就知道了，即使有 C++源代码，完全读懂整个工程都不是易事。

11.7.5 惩戒外挂作弊者

如果客户端检测到外挂，直接断线拒绝对玩家提供服务是一个下下策。因为这样，外挂制作者很轻易地知道了他所做的外挂程序被检测出来了，然后加以跟踪调试分析改进，直到能够通过客户端的检测。程序是死的，人是活的。这样下去，一个死的客户端程序，仅仅依靠那层并不厚实的铠甲是无法抵御外挂制作者灵活的进攻的。

更好的方法是，设法把消息通知给服务器做上记录。服务器了解到外挂使用后也应该酌情处理，比如对于一些处于免费期或者在免费服务器上的账号应当视而不见；对于检测到需要惩戒的账号，也不应该立即执行，这就好比程序调试 bug，如果 bug 立刻引起程序崩溃，我们是很容易修改过来的，一旦 bug 间接引起了数十分钟后的崩溃，则相当难找到。

对于一个公开有众多潜在用户的外挂，最好抑制其传播的方法就是让使用外挂的人没有安全感，在测试期没有注意到外挂制作不善可以被服务器检测到的话，只到刚开始传播时，第一批使用的人才被惩罚。这样，外挂使用者会对制作者失去信任。外挂制作者也会逐步失去耐心。

11.7.6　安全地将信息传递给服务器

客户端一旦检测到外挂，想安全地把这条信息传达给服务器是很困难的。这个时候，网络通讯一般已经收到外挂程序的接管，任何明显的通知服务器发现外挂的信息都可能被拦截修改，变成正常的数据包。其实，有无外挂这个信息仅仅是一个 bit 的数据，具体额外的诸如是什么原因判定有外挂的信息都不是至关重要的了。

想安全地把这一个 bit 的内容正确地发向服务器看似很简单，实则不然。因为客户端程序一旦发布，面对的是无数的破解人员。出于严谨的软件设计态度，我们又不太喜欢以丑陋的补丁形式在设计良好的通讯协议上硬加上一块，破坏掉设计上的美感。

可以利用的方法还是很多的。可以注意到，信息本身的传递不仅仅可以利用数据值的本身，有时候数据的排列和时序也可以携带信息。例如，如果我们协议要求每第一百个数据包必须发一个确认信息包，里面记录现在是否发现外挂程序。如果外挂拦截这个包不发，我们可以认为出现了问题。

但如果外挂篡改了这个包，把里面写的“发现外挂”内容变成“一切正常”怎么办？固然我们可以对数据加密，但还有一些巧妙的方案，例如在报警的同时，悄悄地把第 100 个数据包这个规则改为间隔 101 个数据包。让确认包的达到时机故意变更，这样服务器在不正确的时机收到了“一切正常”的信号，也是认为有问题的。类似的方案可以深入引申出很多技巧，这里就不一 一阐述了。

11.7.7 防止模拟输入

有另一种类型的外挂，用程序去模拟人手工的鼠标或键盘的输入，可以比人手以更高频率和准确性去操作网络游戏中的角色，这种程序一般被叫做“鼠标精灵”或者“点击器”等等。姑且不论此类程序是否真的有违网络游戏中公平竞争之道，作为技术人员，这里还是提出一些防止用户使用它们的解决方案。

直接用客户端程序自身来检测模拟输入的程序是否存在是很困难的，做不到百分百侦测出来。现在许多网络游戏也越来越强调升级过程中的机械操作（我甚至一度怀疑，许多玩家在脑力劳动疲惫了一天后，潜意识里欢迎这些机械操作，从体力上的重复性劳动来换取脑力的休息）。所以收集玩家操作的信息，分析操作规律来总结玩家是真人在操作还是雇佣了机器辅助，这只能作为一个研究课题而达不到实用的价值。

键盘输入在网络游戏中更多是用于聊天，我们暂时不考虑，实际上也没太好的办法阻止玩家模拟键盘输入，下面只好把目光专注于鼠标模拟。

考察人在操作鼠标和机器操作时的不同点：

人在操作鼠标时，和客户端的反应的是一个闭环系统，人眼观察到鼠标光标的位置和当时界面的情况，人手操作鼠标做出了相应的反应。此过程，人是不停地参与对鼠标控制的修正的。

而机器模拟呢，是一个开环系统，它只能发消息，并不能知道反馈，没有修正，我们知道开环系统都是很不稳定的。

只不过鉴于客户端程序这种非常稳定的系统上那些鼠标点击器才能得以正常地工作。如果写到这里大家还不明白，我做个小小的提示：

人是可以接受一段时间内，鼠标移动速度有轻微的不同，因为人眼会对这种情况做矫正，而机器却会因为误差累积而偏差过大。

人眼是以看到的鼠标光标位置为准，而外挂只能去查操作系统的鼠标位置。但是 client 可以自己去画鼠标光标，没有必要去按系统鼠标来。

如果做到鼠标显示的位置和操作系统内部认为的鼠标位置不同，并

且两者之间没有恒定的换算关系，外挂程序想模拟鼠标动作，恐怕要么依赖截图分析图像；要么设法找到程序内存记录鼠标坐标的变量。至于客户端内部鼠标逻辑所在的坐标怎么不让外挂制作人利用逆向工程查到，这就是上面已经探讨过的问题了。

制作网络游戏，如何有效地防止外挂，是游戏发展史上一个新的课题。仁者见仁，智者见智。我跟一些朋友聊起这个问题的时候，得到过各方面的答案。

有人说，根本不需要什么技术手段，只需要找到外挂制作者，无非是一些年轻气盛的小孩子，动用行政力量和法律武器，很轻易地就解决了。这是某大老板，据说跟政府关系不错。

有人说，自己开发的还不好办，每天出一个补丁，对客户端做一次升级，累死做外挂的。这是一资深玩家，偶尔给游戏媒体写写稿。

还有一种声音，给客户端加壳吧，我们开发的加壳工具很强的。而且破解圈子大家都是熟人，互相打个招呼，不帮做外挂的人脱壳，至少能保住程序几年不被分析。这是一位软件破解高手，自己也做过相当不错的加壳软件。

更有人主张，干脆把外挂的功能做进游戏，并做得更好。或者在游戏中就提供挂机功能，堵不如疏。

我的个人看法，为了防外挂，而降低程序的稳定性和破坏代码的优雅都是不能容忍的。在不影响程序结构的前提下，技术上杜绝外挂泛滥并非不可能。

好在现阶段制作外挂的人技术水平还不太专业，平均水平还在较低的层次，因为有经验的破解软件大多更注重的是技术枝节而忽略整体。

防治外挂的工作难度暂时还并不大。仅仅只是因为充斥中国市场的低劣韩国网络游戏对并无太大技术含量的外挂程序都无能为力而夸大了这点。

乘这个网络游戏刚发展起来的机会，游戏设计者应该多想想游戏的可玩性等更多问题，而不是一味地把心思都放在如何不让玩家作弊这一点上。

版本控制

我自己在进入网易做《大话西游》系列开发之前，是没有考虑过版本控制的问题的。甚至，我见到了许多游戏开发公司都没有做好这些。在四年多中经历数个网络游戏项目后，我深深感到版本控制在一个软件项目开发中的重要性 。

老实说，我不是这方面的专家，除了介绍一下自己熟悉的 CVS，SVN 这些版本控制软件的使用外，我实在做不到更多。

这里为版本控制单独列出一章，唯提醒程序员们，其实在为一个软件开发过程里重中之重。

许多喜爱单枪匹马做游戏的程序员几乎没有版本控制的概念，最多只是在项目达到一个发布版时把源码备份一下。这是非常危险的事情，一旦代码的重构错误，引入太多 bug 时，我们想回到从前的稳定版本的难度会随着间隔时间拉长而迅速变得困难。在我个人经历中，需要对前一阶段的工作大返工的机会很少碰到，但是一旦遇到这种难得一见的事情，版本控制系统就变得尤为重要。

在多人合作的项目中，如没有版本控制系统，则难以想像能顺利把工作做下去，靠手工的共享文件方式同步源代码只在上世纪的开发者中才难得一见。

由于每个开发人员都希望自己的机器上有一份完整的代码可以供运行测试，所以同一份代码在各自开发人员的机器上都有拷贝，这样不同的人员对不同部分的修改后的同步问题成了大麻烦。

先不说用文件共享复制的方式能否真的永远保持同步，有时候，部分人员希望临时修改一个部分以供测试，然后再改回，导致即使是本地机上也会存在多份复本，或者靠注释掉不要的代码增加源文件的体积，最终混乱不堪。随着时间的推移，开发人员可能对自己机器上的不同版本间具体差异的了解都变得模糊不清，甚至忘记了当时为什么区分这些版本的原因，只看见机器上不同版本的拷贝，或者是同一文件中众多 #ifdef。最后给多人的代码整合带来巨大的麻烦。

作为开发人员，有时即使知道自己机器上的某个版本可能不会再使用了，他也不敢去删除，害怕日后万一需要从里面找回点什么。久而久之，这种死掉不要的代码充斥于团队内的各台机器。

软件的开发维护的过程，不同人对项目的代码的修改有时会出现彼此交叉的情况。这时交流问题也会显露出来，源码的文档往往得不到及时跟进，更多的时候我们依据口头约定和查看他人对源码的修改来判断合作者的意图，远远比不上几乎所有源码控制工具中都嵌入的 diff 工具了。

版本控制系统对需要中途发布中间版本的需求更加有用，有时候我们需要在项目中间发布一个版权，并冻结住所有的新功能增加，而只是修改一些 bug，但后续版本的开发却不能停下来。这样的要求靠人工维

护几乎是不可完成的任务。

如果你的项目还没有使用版本控制工具，请不要犹豫，现在还不迟，马上加上，而不要等到下个项目。

12.1　常用版本控制软件

1．VSS (Microsoft Visual Source Safe)

VSS 应该是 Windows 下的游戏开发者用得最多的版本控制工具了，只要安装了微软的开发套件，就嵌入到了开发工具中，使用起来也非常方便。它是基于 Windows 文件共享的机制来实现多人协作的。在局域网上用一个机器创建一个源码的数据库，用于保留所有源文件每个版本的数据。如果是单人工作，同样可以把数据库创建在自己的本地机上。

每次对项目中文件的完整修改后，都把修改提交到服务器上，服务器作为一个新版本保留下来。如果是多人维护一个项目，为了防止两人同时修改同一份文件而造成结果不可预知，采用先需要修改的人锁定住文件，修改完成后提交再解锁的方式（如图 12-1 所示）。在 VSS 中，协作者通过共享文件的方式从源码服务器中获得每个文件的最新版本，并对自己并未提交加锁准备修改的文件设置为本地只读的。这样，任何时间都只有一个人可以修改某个特定的文件。

我几乎没有用过 VSS 和这种基于加锁的协作式开发版本控制工具，所以其中的细节这里不再多表述，以免贻笑大方，只是多提一下 VSS 不太方便的地方。

如果使用 VSS 做多人项目开发，一个必要的协定就是所有参与开发的人员在下班之前都必须让所做的工作搞一段落，并把自己今天的工作提交给版本控制服务器，解锁所有文件，以免第二天由于每个人上班时间不太相同，某些需要修改的文件被锁定影响他人的工作。

我自己几乎没有用过 VSS，不方便对具体功能做评价。但是我有许多朋友都喜欢用它，微软的软件品质一贯还是很不错的。据朋友说，VSS 同样可以支持 CVS 那样的提交融合机制，在 VSS 中被叫做 multicheck，不过我很少见人那样用过；另外，也有人告诉我，通过某些方法可以不局限在局域网内使用 VSS，只是我个人所见，使用 VSS 的项目都是被限制在办公室内，如果有程序员想把工作带回家中，都是简单地把源码复制一份了事。

对于非程序员，比如喜欢使用 Word 和 Excel 的游戏策划人员恐怕更青睐 VSS。毕竟同为微软的产品，VSS 的 diff 功能对 Word，Excel 支持得相当好，而不像其他版本控制软件默认只能把这些非文本格式的文档当做二进制数据比较。

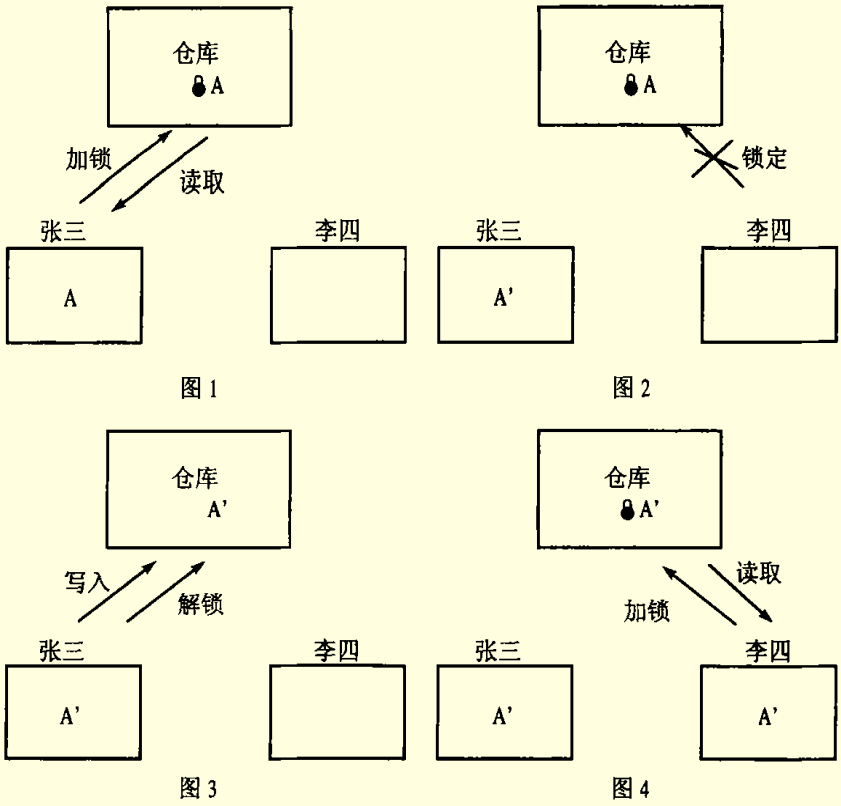


图 1：张三锁定文件 A，然后复制一份出来编辑。

图 2：当张三编辑的时候，李四企图锁定文件 A 是会失败的。

图 3：张三写回他修改过的版本，然后解锁。

图 4：现在李四可以加锁了，然后对最后一个版本做编辑。

图 12-1　加锁-修改-提交解锁的图示

多人修改同一文件的机会不算太多，但也非很少发生，那么发现项目中某个源文件中的 bug 而不能及时修改是件很郁闷的事情。如果想减少这种情况的发生，必须减少每次修改的时间，这样就使提交的频率增加，不稳定的版本也变多了。

VSS 基于局域网文件共享来解决多人协作问题，这样对异地开发很不利，即使开发人员都在一个办公室办公，那么对想把工作带回家处理的程序员来说，也是一件麻烦事。

2．CVS (Concurrent Versions System)

https://www.cvshome.org/

这恐怕是互联网上使用人数最多的版本控制系统了，几乎所有的基于

互联网的开源项目都用它管理代码。各个平台都有 CVS 的实现，理所当然，Windows 下也有 CVS 的带图形界面实现，不过基于习惯，我个人还是偏好使用 CVS 的命令行版本。和 CVS 不同，CVS 天生是用于多人异地同时维护源码用的。所以 CVS 虽然提供了锁定功能，但并不提倡使用。

通常，参与项目的人都是可以通过加密的 ssh 连接从 CVS 服务器上取下源码的副本，其中有修改提交权限的人是可以同时在本地机上修改这些代码并分别提交到服务器上的（如图 12-2 所示）。

虽然大部分情况下，同时修改同一文件，由于修改没发生在同一位置，CVS 服务器会协助用户把不同的修改点融合起来，但也有少数情况，修改会发生冲突，这个时候要由后提交的人负责检查自己修改的部分和上一个提交人的修改部分的冲突部分解决后再提交（如图 12-3 所示）。

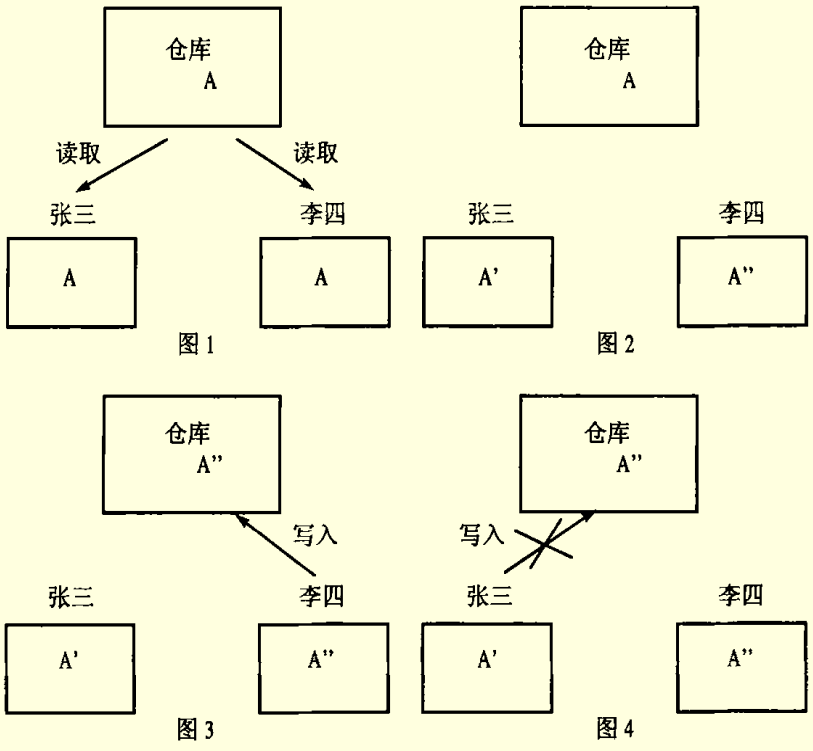


图 1：两个用户分别取到同一个版本

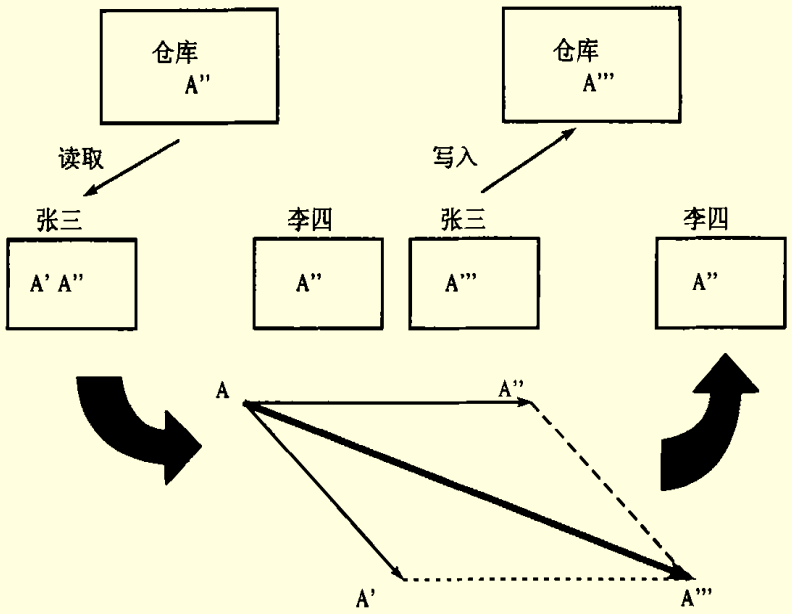
图 2：张三和李四分别对这个版本进行修改

图 3：李四将他修改后的版本写入仓库

图 4：张三在企图写入他修改的版本时失败

图 12-2　工作示意图

我是从 CVS 开始接触版本控制软件的，最初使用 CVS 的一个项目的主要程序员分布在广州和北京。北京的同事一个月才来广州几天做当面的交流，剩下的时间都是通过即时通讯软件，和 CVS 服务器共同维护同一份代码。这个项目后来完成得很成功，代码质量也非常高。合作的过程也很顺利，多个人维护同一个源文件的情况也时有发生，但几乎没导致过严重的问题，反而多次相互检查出 bug。http://sourceforge.net 这样的开源项目网站使用 CVS 不无道理，如果想跟踪某个开源项目的进程，甚至参加进去，安装并学会使用 CVS 是非常必要的。



张三将 A"这个版本读回，然后对 A'和 A"做了一个融合(merge)，得到 A'"版本。然后将 A'"写入仓库。

图 12-3　工作示意图

3．SVN (SubVersion)

http://subversion.tigris.org/

近一年多，我已经把所有的工程从 CVS 过渡到 SVN 中。这是一个很不错的版本控制软件，大有取代 CVS 之势。从功能上说，SVN 拥有所有的 CVS 功能。就一年多的使用感受，SVN 最方便的莫过于在它上面的所有操作都是原子的。不像 CVS，CVS 对每个文件的版本控制是独立的，也就是说，如果 A 文件和 B 文件是有关联的，但更新的版本次数不同，导致了两个文件的版本号不同，这样，我们在某一时刻想找回相关的一对版本会比较困难（大多数时候，我们只能利用提交时间来人为判断）。

而在 SVN 中，相关的文件要求同时提交，这些文件的提交请求的相关性被记录下来，而且提交操作保证了要么所有的文件都被提交，要么都提交失败。这样更好地保证了工程的每个版本的完整性。

另外，有为 SVN 开发的 Windows 带图形界面的版本 TortoiseSVN（http://tortoisesvn.tigris.org/），使用起来非常方便。和 Windows 文件管理器结合得非常完美，并且可以方便地在本地机上创建数据仓库，不仅适合多人合作，个人使用也是非常合适。

12.2　版本控制工具在项目中的使用

最近两年，对于新招聘进来的程序员，我们公司都会首先对他们做版本控制工具的培训，甚至正确地使用版本控制工具也成了最重要的试用期考核点之一。但是，还是有许多对版本控制工具的误解和误用。

每个版本控制工具都会要求先在某个地方创建一个数据仓库（Repository），每个作者都将仓库的全部或一个分支 checkout 到本地硬盘上。在增加新的文件或对老文件做出修改后，把更改提交（commit）到仓库。这个提交过程每次都生成一个新的修订版（Revision）。由于每个修订版都是可以被单独取回的，所以使用了版本控制工具后，就不用再用操作系统的文件复制功能单独备份某个特别的 Revision 了。

版本控制系统中，每个文件的 Revision 都保存了很多信息，包括作者、日期，还有提交时的日志，这些都是日后回忆和找回历史版本的线索。每次对版本控制系统做的操作都被忠实地记录下来，甚至包括了对文件的删除，整个数据仓库都是出于一种不断叠加的状态，真正地从仓库中删除一个历史上的文件是不可能的。对于使用者，就应该谨慎对待每一次版本控制的操作。

版本控制系统绝对不是数据备份的工具，也不是数据同步工具，对于软件开发，应该合理地提交应该提交的文件，在保证任何一个参与者 checkout 出一份完整的源码后，可以顺利地编译整个工程而不缺失什么文件；同时也不用提交用程序自动生成的文件，例如编译器生成的目标文件。

刚刚使用版本控制系统做协作开发的人总是拿不准何时是应该提交修改的时机，在我们公司的新人中，有怕犯错误迟迟不肯提交代码让版本控制系统失去它的价值的人；有不厌其烦改了几处代码就提交一次的人。这个的确没有什么绝对的准则，在不同的项目，不同性质的版本中，提交时机的策略并不一样。

大体上我们应该遵循一个大原则：要让仓库中最新的版本是可以编译，能保证正常工作的。并且提交新版本的频率上，应该在全部团队的

人员中达成共识，每个人在提交的时候还应该认真对待日志，一定不能空缺不写，这些信息在团队合作中尤为重要。

多人合作总是不能避免同时修改一个文件，迫使融合（Merge）的发生。大多数情况下，几个人修改同一个文件也不会在同一个地方，版本控制程序会帮助你做自动融合操作。一般都是可以信任的，不过一旦发生了融合，我们应该立刻查看 diff 信息，以确认机器到底帮你干了些什么。如果机器不能帮你解决冲突，那么还是需要人工解决了。

从一开始就避免融合其实更为有效，这需要我们代码有明确的分工，把每个源文件的体积保持在一个合理的尺寸（就我个人而言，一个 cpp 文件不应超过一个类的实现）。

每天的工作开始之前，第一件事情就是应该 update 代码，减少和合作者的冲突，尤其是在开发期，总有人可能把工作带回家去做，深夜里提交代码并非不可能的事情。

做出阶段性修改后一定提交一次代码，如果代码并没有经过充分测试，则应该在日志中写明。当需要融合时，无论是自动融合还是需要手动融合，SVN 都不会允许提交，强迫最后提交的人做一次代码审核；但是，CVS 系统做出自动融合后，是允许提交的，只是做出一点提示，这种情况下，程序员也应该自己审查一遍自动融合的结果，确认正确无误。

互相修正 bug 也是造成代码融合的原因之一。在实际开发中，我们采用了通知作者的方式，也就是说，一旦发现某段代码的 bug，都先从版本控制系统中找到代码的最初作者，沟通后确认 bug 以及修正方案，通常由原作者自己修改；或者在得到认可并双方确认了修正方案后才代为修正。

利用版本控制工具做协作开发，文件甚至目录的命名变得更加重要了。创建好一个项目的目录结构是需要项目领导者一定的经验的，其他项目组成员可能日后就只维护某个分支目录了。一开始，项目领导者一定要规划好，文档、源码、测试代码、工具代码，以及主版本，分支版本如何组织到目录树下。这些东西一旦确定，日后再改变就很麻烦了。

在特别大的项目中，合理的目录结构才方便设置分支代码的控制权限。

作为游戏这样的不太大的项目，参与的程序员并不算多，我们实际操作中并没有严格控制每个人的操作权限。实际上，一些团队中的约定比物理上控制每个人的权限更为灵活有效。

如非必要，请给予团队中所有参与人员查看和修改所有代码的权利。

我个人并不太主张维护分离太远的多个分支，那样事后融合的工作会非常困难。但是像网络游戏这样的需要不断地更新维护，并随时根据用户的需求修改的软件，定期发布稳定的版本是很重要的，这也是使用版本控制工具的一个重要原因。当项目进行到一个阶段时，应该利用版本控制系统，从正在开发的版本中分出一个分支，这个时候，对分支版本应该冻结所有的新功能增加，着力寻找潜在的 bug 并修正。如果有可能，找到 bug 后应该及时同时修正最新版本，如果期待日后的融合阶段，那简直是一个噩梦。

对于这个分支版本最终的结局，我们通常的做法是放在那里不需要继续更新下去，而非融合到最新的版本中。

在 2001 年的时候，我所在的公司（网易）代理了韩国某游戏公司的一款游戏《精灵》。当时为了汉化游戏，公司派人去了韩国，在他们的办公室里，我们惊讶地看到了他们开发项目没有使用版本控制工具，所有的协作都是通过共享文件夹互相拷贝来完成的，这给人带来了极其糟糕的印象。后来这个游戏在中国运营失败，抛开别的许多问题不说，没有使用版本控制工具一定是造成开发混乱的原因之一。

最明显的例子是，我们要求韩方公司为游戏开发中秋节活动，等到中秋节前夕接到新版本后，bug 不断，公司这边客服人员叫苦连天。当我们的投诉信件辗转送到韩方开发人员那里后（之所以说辗转，是指语言不通的障碍，无论用中文、英文、韩文直接交流都很困难），收到的反馈却是，开发人员正在为国庆活动添加代码。游戏提供新版本，而中秋版本由于无法退回，也不能针对老版本修复 bug。等挨到国庆，新的版本发来，中秋节版本的 bug 已经补上，可国庆节活动又出现新问题。如此反复，最终的玩家永远玩不到真正稳定的游戏了。

当我们求爷爷告奶奶的，几尽哀求地写 E-mail 让韩方开发人员赶紧

使用一种版本控制工具辅助开发时，他们对此仍旧不屑一故。看来，让没有版本控制概念的开发人员建立新观念也是一件不太容易的事情。正所以如此，我才在本书中特地单独列出一章提出这个问题，希望得到广大游戏开发者的重视。

事隔三年，公司又代理了韩国另一家游戏公司新的一款网络游戏。这次我们花重金购买了大部分代码。实际上，购买源代码的唯一意义在于出现问题后，可以自己解决。而维护这些代码，对于我们这边的程序员来说，痛苦程度远远胜过自己开发，那么糟糕的代码除了增加一些午餐时聊侃的谈资，甚至没有什么借鉴价值。

不过这次终于在代码中发现了对方使用了版本控制系统的痕迹，这就足以让人欣慰了。可惜，代码的作者似乎还是不太习惯使用版本控制，大段注释掉的废弃代码遍迹于工程的各处，以至于有一个 cpp 文件居然长达九千行。用这种方式保留住历史痕迹，置版本控制系统于何地?

12.3 不仅仅是程序才需要

版本控制系统不仅仅是程序员的专利，对于策划和美术人员同样重要。策划人员大多独立行动，不大可能出现多人同时编写一个文件的情况。在不使用版本控制时，许多策划喜欢手工备份关键阶段的文档。事实上，使用版本控制工具可以取代这种落后的手段，并且很容易做到文档共享。

像 SVN 这样的工具，支持直接从 Web 页上访问仓库中的内容，同一小组的技术人员就可以方便地查阅策划文档了。如果案子并不是那么机密，甚至可以尝试直接用 wiki 系统纂写，我们现在的一个小项目就做了这样的尝试，效果不错。

由于 wiki 在公司内部公开，任何人都可以看到最新版的文档并提出建议，甚至看到错别字都可以随手修正过来。可惜出于程序员不可理解的某些原因，游戏策划往往不愿意公开自己的思想和灵感（反之，程序员大多喜欢和同行分享自己的知识和新写出的代码），直接利用 wiki 系

统充当文档的版本控制系统还需要一段路要走。

美术人员的思维方式和技术人员不太一样，版本控制观念的灌输是需要费一些力气的。而且美术的图档不太容易比较出二进制的差异，一些专业的工具可能更合适一些。

可惜这些工具售价都非常高，如果不追求图形界面的易用性（比如和作图软件的整合），数据仓库服务器有个高速海量的硬盘的话，直接使用 SVN 也不坏。

在游戏开发中，会有一些自行开发的专用工具，比如地图编辑器。在游戏规模不大的时候，可以切分地图给不同的美术人员编辑。但有时候会有一些特殊的要求，比如游戏的地图面积巨大并且空间连续，那么势必要求多人同时编辑。这种情况下，最好的方法是把版本控制工具内嵌到编辑工具中。

好在 CVN，SVN 这样的系统都是开源项目。编辑器开发人员倒不用自己重头设计了。不过需要额外注意的是，一般的版本控制系统是基于文件的，即使是一整张大地图，需要协同工作时，在开发期我们还是需要把数据切分成很多小文件分别记录。

这里应该留意到，Windows 系统在同一个文件夹内不太适合放置过多的小文件，这会使目录中检索文件的效率变得很低，用浏览器打开这个文件夹的时候也会像死机一样停在那里。在打算切分小文件时，一定要把它们分到多层不同的子文件夹中去。

开发方法

这是个人对软件开发的经验体会。我既参与过失败的游戏项目，如《大话西游》的最早版本，对于失败深有感触；也在几个游戏项目中体会到了成功的喜悦，如后来的《大话西游II》和《梦幻西游》，我都从头到尾地目睹了它们从无到有成长到百万用户在线。

我在它们上面付出了许多的时间和精力，或多或少都悟到点什么。尽管文字的传达必然会损失掉许多色彩，但我已尽力来和读者分享。成功很难复制，失败也难回避。过去的历史却总能给我们一些借鉴。

游戏虽然同样是一种计算机软件，但似乎一直都没有被当做正统的软件来制作，也没有按各种严格的软件制作方法来制作游戏。在上个世纪，游戏开发往往被玩家们想像成一堆电脑奇才窝在几间破旧的小房间里，没日没夜吃方便面打地铺做出来的作品。事实离这想像也差不了太远，国内早期的游戏制作者无一不是从玩家转过来的，他们实践着自己的想像。

最早的人们是一批从学校里毕业不久甚至没有毕业的学生，多半是痴迷于编程和游戏的，在老师的眼里不能算得上是好学生，但绝对是聪明的。这批人开始用自己学会的编程语言开始编写心目中的游戏。然后是一些骨灰级玩家，他们不一定懂编程，但是他们自信能设计出一流的游戏，这批人如果机缘巧合碰上了前一批人，就成了中国最早的游戏策划。这应该就是九十年代初国内游戏产业刚刚发展起来的样子吧。先有一些狂热分子自我组成的开发小组，然后或者自己凑钱办个小公司继续开发，或者被别的公司买入。

在游戏产业刚刚迅速发展起来的几年里，光是北京就大大小小遍迹了十多家游戏公司。诸多的梦想着制作自己游戏的玩家投靠过去，形成了最初的数百人的群体。

可以看到，游戏开发从一开始就是一种玩的性质，靠的是开发者个人的力量，尤其是编写程序的这个程序员。如果这个人很强，就能把游戏顺利做出来，如果做不下去，那么整个团队就会崩溃。可惜，这些带有浓郁个人性质的制作组，根本谈不上质量监控，谈不上进度的把握。长期的开发也很容易使主要开发人员精神状态受到极大的影响，开发了一半放弃离开的不在少数。没有开发经验的人们本来就在一边摸索一边制作，我们真的不好过于责怪他们的失败。

其实，最初的国产软件业的情况也好不到哪去，可是最早的国产软件规模都不大，只需要一两个程序员就能完成，坚持不下来的一定有很多但是我们看不见了；坚持下来的最终获得了市场的认可，慢慢地发展了起来。

游戏，到90年代初的时候，市面上的成熟产品已经很复杂了，做一个相当的游戏已经不是程序员个人所能完成的。至少需要包括美术策划

在内的 4，5 人才能完成，

每个人的思考方式都有所不同，协调起来更加困难，导致了我们几乎找不到一款成功的国产游戏。那个年代大部分国产游戏都有严重的质量问题，没有制作者敢拍着胸脯保证我的作品绝对不会出错跳出来；更不用说那些几乎不能顺利从头玩到尾的《江湖》，《血狮》之流了。

我们不用抱怨盗版太严重，游戏制作的周期太长，成本无法回收，其实最大的问题还是游戏的质量。频频出错的游戏，玩家玩得不顺心，怎样也不可能大卖。

关于质量问题，其实一直延续到了现在。我本人也经历了一次《大话西游 I》的失败，并目睹过好几款不能顺利开发下去的网络游戏，归根到底还是开发方式的问题。

过于懒散，并没有质量保证的开发流程，最终注定了结局。

13.1　失败的经验

（1）过多的工作压到了同一个人身上

在很多的小公司里，总有那么一个核心程序员，负担了几乎所有的编程任务。这个人即使是个天才，也被繁重的工作压的没有学习和自我提高的时间。

比如设计模式这样的思考方式也是近几年才进入中国普及开来，却并非那么容易学习。整天迫于项目压力，就无法领悟新的东西，使用更好的方法解决问题。

独自一个人总揽大局，在项目后期很容易因为想早点结束，而把整个代码补丁打得满目沧夷，只求能解决眼前碰到的 bug 而不为长远的结构稳定去考虑了。

结果整个工程无法分出明显的模块，各部分耦合度太高，改一处牵动全身，一直到新的 bug 出来后无可救药。

（2）过分的弹性工作制

游戏程序员好似天生追求自由的一群人，白天办公室里经常看不到

几个人，晚上通宵达旦地在电脑前奋战。不否认好的程序员在精神兴奋的时候可以连续工作写出许多质量不错的代码，但是停下来时几天没有进展也是时有发生。尤其到了项目后期，一旦 bug 重重，程序员备受挫折，仿佛眼前的活永远都干不完，而且调试程序也失去了编写程序时的新鲜感。最终假借弹性工作制的名义，消极怠工拖垮项目的先例已经有很多了。

弹性工作制不等于没有计划，不等于可以随意为之。在项目没有进展，而程序员却整天不干正经事，只知道聊天，浏览网页的时候，程序员并非没有心理压力，他可能只是不知道下一步该怎么做，或是问题太多已无从入手，这说明项目已经失控了。

（3）没完没了的变化，没完没了的返工

大家都想把自己的作品做得更好，往往正在做的还是自己的处女作。自己的游戏做了一半，看见新上市的游戏有了什么新玩法、新东西，都想抄到自己的东西里来。

新发现什么技术可以让画面效果更绚，就迫不及待地实现出来看看。又或者，新做出来的东西老是不满意，三天两头地放弃重做，力求完美。最终，游戏老是做不完，看不到尽头。如果从投资方来了时间压力，最后一咬牙把个四处有缺漏的版本交出去，等待着的是玩家没完没了的投诉电话。

（4）没有及时的测试

谁都知道软件做完了是需要做测试的，测试出问题就应该修正。这似乎是天经地义的事情，但是落到实处，早期的国产游戏却很少有严格的测试。因为很多人忽略了一点，无论是哪种软件开发方法，测试都不是在软件全部开发完毕后才进行的。等到大家看到项目好像已经完工，这时候对游戏做整体的测试已经完了。必定发现一大堆的 bug，已经到了无处着手的地方，如果只是头痛医头脚痛医脚那还真的不如借这次的经验教训，把程序推倒重来做更好的设计呢。

（5）项目的主导严重地偏向了某一个职位上

早期的游戏大多是程序员主导整个游戏的设计的，程序员说应该怎么做就怎么做，说今天要怎么改就怎么改，甚至没有单独的游戏策划一

职，全部由程序员包办。

这对小型游戏倒没什么，对于规模较大的游戏，最终即使游戏程序本身没有大的问题，但是在游戏性方面却让玩家难以接受。作为技术人员，很容易把目光注视到技术的枝节上，而忽略了玩家的感受，大多数情况玩家并不把技术人员扬扬得意的地方放在眼里，游戏毕竟是给人玩的。

后来又出现了策划为主导的游戏，策划拍脑袋一想，我们应该加点什么，那么程序员就日日夜夜地加班完成。如果实在是技术解决不了的还就算了，最怕的是，本来技术解决有问题，可能新加的设计会影响到整体的框架结构、对硬件的过分要求等，暂时像狗皮膏药一样把功能补上去看起来工作得很好，可是早早地就给日后的持续开发留下了隐患。程序员在实现的时候盲目听从策划的安排，没有仔细地全面思考，造成了项目无法完成。

我也听说过美术甚至市场人员为主导的项目，但并没有接触过这样的团队，就不太清楚会出现什么问题了。

总而言之，游戏开发是一个需要大家共同努力的事情。我们应该把自己的思考都带入到开发中，盲目地跟从别人的想法或固执地坚持自己都有可能影响整个项目。

13.2　成功的经验

我这两年都在从事网络游戏的开发，虽然自己不算太满意，好在产品得到了市场的认可，算是小小的成功吧。出于程序员的角度，在程序开发方面可以给出一些建议。

虽然一款成功的游戏不光是程序员的努力，成功也有一些偶然性，但不能不说下面这些还是值得后来者借鉴的。

（1）引擎和实现的分离

许多游戏程序员是从研究图形显示技术开始的，总是想让游戏表达更丰富绚丽的图像，结果游戏程序就变成了一个图像显示的集合。代码

中充斥着对图像 API 的直接调用，图片或者模型本身居然成了游戏中的逻辑对象的一部分。这种方式势必造成图像处理部分的模块和游戏逻辑处理模块的耦合度过高，我们知道，模块间耦合度过高对于软件整体的稳定性是非常不利的。

所以，在整个游戏项目中，最好能分出一个人独立编写图像相关的模块，他不需要去管任何的游戏逻辑；而其他的人则完全不需要了解什么 DirectX，OpenGL 等底层的东西。如果是购买现成的图像引擎，也最好有这么一个人能专门熟悉其的使用，并教授给团队中其他程序员，或者负责加上对图像引擎的改造。一言以蔽之，和图形图像打交道的人，整个开发团队有一人足可。如果太多人参与做这件事情，容易引起混乱。

不光是图像处理模块，声音、网络、时钟、文件读写、Windows 窗口控制，这些都是需要和操作系统直接打交道的东西，不管多简单，都应该封装一层和游戏实现层隔离开，在游戏实现的代码中还出现对 Windows API 或者 COM 组件的接口直接调用都将是非常糟糕的事情。表面上说是不利于跨平台（或者很多人压根就没想过要把自己的作品发布到其他平台上），实际上增加了许多维护和排错的困难。

（2）结对编程

我第一次尝试结对编程是很早的事情了，那个时候还不知道 XP（极限编程）。到接触 XP 时，才知道这已经是一套发展了很多年的开发方法中非常重要的一部分。

其实结对编程做起来很简单也很有趣，找个水平差的不太远的程序员和自己配成一对。只用一台计算机，大家选一个人坐在键盘前面负责输入，另一个人坐在后面口述。两人要不断地交流，频率不应该低于一分钟一次。整个的设计思想由后面只动口不动手的人主导，而由操键盘的人做实现。由于人的思维速度是快于输入代码的速度的。那么观看的人可以有空闲的时间做额外的思考，观察代码写的有没有问题，结构有没有问题。

如果程序员的经验积累足够，是很容易看出存在潜在问题的代码的，即表面上实现了功能，但实际上是一种糟糕的做法。这在 XP 中被称为代码的坏味道，在 Martin Fowler 的《重构》一书中有详细的介绍。两个

有经验的程序员同时在一起工作，看起来好像浪费了一个人的时间；但实际上的效果却是完成了更高质量的代码。程序变得不那么容易出 bug，而且代码也写得更为优雅和紧凑。

我最近一年在大项目完成后的闲暇阅读了一些关于 XP 的书籍，深受启发。发现以前无意中尝试过的一些类似 XP 的开发方法其实是可以做得更好的，并且就书中所做的介绍，在公司内部用一个小项目的形式做了些实验。

关于结对编程，发现了一些新的受益之处。首先，它可以促进参与项目的程序员自身的提高，一对程序员工作的时候，水平较低的一方会潜移默化地受水平略高的程序员影响，学到一些新的东西。而水平高的一方同样因为不断地把自己的想法说出来而整理了自己的思路。

其次，一定时间周期地打乱配对，让参与项目的人员相互转换位置，使得维护繁杂的文档变得不那么重要（其实我所见的游戏开发过程中，几乎没有人把源码的文档和代码同步书写的。要么不写文档，要么在项目完成后才补上）。大家分组打乱后，口头的交流很容易让所有人都熟悉每个模块，这样对于公司也很有好处，项目中万一有人离开，也不至于影响到整个项目。最后，开发过程变得更为有趣，人和人的交流变得很多，大家关系更为融洽。

另外想补充一点的是，讲解 XP 的书籍上都没有提到，但是实际上却存在的一点：结对编程使得程序员被迫提高了工作效率。如果单独工作，在遇到困难的时候，并不是所有人都立刻积极地去解决问题，这时或许会上网和网友聊聊天，看看无关的网站等等。有可能因为工作的打断，大半天的时间都浪费了。看起来，程序员每天都在加班，实际有效工作时间往往还达不到 6 个小时。而结对编程有一种相互督促的作用，在一边工作疲惫状态不好时，另一边会起一个鼓励和激发斗志的作用。

而且两人共用一台电脑，略带私人性质的聊天等活动都会很自觉地不去进行了。结果一天下来，新实验结对编程的程序员都会喊累，神经紧绷 8 个小时的工作不累才怪。

从这个角度看，严格限制结对编程的程序员不准加班是合理的，实际上，开始每天甚至不必限制 8 小时工作，每天这样工作 6 小时对项目

同样是非常高效的。

当两人不断地交换角色，以至于最后谁也记不清哪行代码是谁敲的；团队内循环的分组以至于分不清到底哪个模块是该谁负责；反而大家的感觉会不错。整个项目的代码是团队共有，而不再是个人作品了。

（3）随时方便地测试

测试是如此重要，以至于从项目一开始就应该进行。不要吝惜为实现代码写测试部分。测试代码的行数达到实现部分的行数甚至超过它都不稀奇。或许在写测试代码时程序员会感觉痛苦，不过一旦需要改写某个模块后，新的代码通过了所有的测试那种安全感是非常美妙的。发现新的 bug 后，增加对应的新的测试案例，保证犯过的错误不再犯。

一旦程序通不过测试，就应该放下手头所有工作解决问题，不让 bug 泛滥，这样最终的软件质量才能够得到保证。

测试应该是自动化的，是方便快捷的。很多游戏程序员叫嚣着游戏程序图形显示太多，和用户的交流过多，做不到自动测试，其实完全是一个误解。图形模块的测试，可以写出测试程序把所有可能的显示方式都进行一遍，虽然很难做到程序自动校验结果的正确性，但只要做到所有的过程都是自动化，方便自己坐在电脑前无须操作地观看就已经够了。

至于用户操作的交互性，完全可以在用户正常操作的时候把一些过程录制在文件中，利用这些用户的数据做一些不需要再干预的操作回放。非网络游戏只需要记录下玩家键盘鼠标的操作状态；网络游戏额外记录下数据包就够了。

一件多么简单的事情，居然没有多少游戏开发人员去实行，在我看来是一件奇怪的事。尤其是网络游戏的服务器部分程序，唯一的数据就是外界传入的网络数据包，记录并重现是很方便的。不做这个工作的开发人员，一面责怪着服务器程序不像 Windows 客户端部分可以使用强大的 IDE 调试，一面只会用写 log 的原始方法猜测 bug 的位置。对内存越界，内存泄露这些故障一筹莫展。实际上，一旦做好了网络数据录制及回放操作，理论上可以无限次重复 bug 的发生，找到问题是多么简单。

把游戏的测试工作全部推给测试人员做黑盒测试是一个愚蠢的行为，测试应该从开发者这里做起，从项目启动的日子就不断地进行。只

有开发者自己最熟悉代码的结构，只有更早地发现 bug，才能够更容易地解决。

（4）尽早地发现结构上的问题并及早重构

一旦发现了游戏程序在框架结构上出现了问题，就不能急于添加新特性了。结构的问题不及时解决，后果是严重的，早一天重构就不至于把重构的消耗变得无法控制。

一般来说，发现结构问题都是游戏特性实现得不太多的时候，这时，有经验的程序员应该嗅到代码中糟糕的味道了。明知道大厦将倾还修修补补，真不如赶快推倒重来。

如果发现得太晚，就应该及时冻结新东西的增加，裁减游戏内容，保证现有的东西不至于出错了。

（5）其他

使用嵌入脚本和版本控制系统这两点在前面的章节已经花掉很大的篇幅讲解其重要性，这里就不在复述了。

程序员不要过于抱着我做的是一个游戏的看法，首先它是一个软件，把游戏真正当做严格的软件来开发才能保证其质量。游戏软件中，代码更多的部分将是和画面上看到东西关系不大的，也并非只因为这是一个玩具就可以随意地编写程序。游戏程序员同样需要不断地提高自己在软件开发方面的素养。

13.3　对游戏开发方法的一些思考

我从头至尾参与完成的大的游戏项目至今有三个，分别是《大话西游 I》，《大话西游 II》和《梦幻西游》。这三个游戏在玩家的眼里可能有诸多类似，但是作为开发人员我很清楚，至少在我参与的客户端模块，三个项目几乎每一行都重新写过的，可以说是截然不同的三套代码。

《大话西游 I》是一个失败的作品，bug 迭出，但是后两个产品却都取得了成功。反观三个游戏的开发过程，我不由自主地去总结一些成功的经验和失败的教训。

一个共同点是，成功的产品都在开发期借用了上一个产品的资源。《大话西游 II 》开发之初，虽然《大话西游 I 》的程序错误百出，但是制作好的图片都是可以再次使用的。

整个开发过程不需要受美术人员进度的约束，想要什么样的图片，立刻就可以拿来加到游戏中使用。虽然理论上，对于这款游戏的开发，客户端的程序员只需要有一张地图，一个角色的动画就能够进行下去工作了，满世界的 NPC 长得一个样，满世界的场景都似曾相识并不影响开发工作，但实际上，人的心理因素是不能忽略的，毕竟，我们喜爱游戏，才有了做下去的兴趣。千篇一律的画面，让一切都仅存于想像里是很打击开发人员的士气的。

而《梦幻西游》，最早的内部版本根本就是用的一个和《大话西游》II 二进制相同的执行文件。我们的游戏逻辑被放在脚本中，编写不同的客户端脚本就能够满足新版本服务器和新版本玩法的需要了。这时候，新的服务器程序可以顺利地写下去，不需要担心客户端的不稳定，可以集中开发人员的精力。

同时，新版的客户端在开发中，可以为新玩法增添新的表现效果，为日后可能增加的新特性做准备。程序的框架也可以搭得更完善稳定，没有人催促新的客户端赶快增加某个功能以适应新的需要。项目的进度可以有条不紊地进行。

游戏毕竟是有许多不同工作的人合作做出来的，粗分的话有策划、程序和美术；从程序角度再细分，还可以分做引擎、工具和实现。每个部分之间都存在一定的依赖关系，同时开展这些工作有时候欲速而不达。

那么，作为一些需要延续性开发的团队，实际上是可以把职能不同的工作分开来做的。比如，策划可以先写案子，然后让美术根据策划案来完成一些场景和人物甚至界面。程序部分可以在这些都已经完成后再开始，然后根据已有资源开始逐步完善。

在制作程序部分的时候，实际也是一个整合的过程，项目经理应该能看到游戏逐步成型，但是如果在策划案或者美术表现上有所不满，不应该再临时提出改动。我们可以把所有改动的方案整理记录下来放到下一版本。

即使美术人员希望重新制作图片，也可以全部为下一版本设计。这样，所有的进度控制和质量保证就放在程序一个环节，而不至于出现多方面相互掣肘的情况。等第一版本推出后，即使效果不理想，但是时间并没有拉得太长，制作成本被严格控制了，并且吸取了大量的经验。许多修改工作都是经过多番思考过的，这样制作出的新版本一定能好很多。

这上面的流程即使颠倒过来也能行得通，定了游戏大体类别后，让程序先开始制作一些和具体设定无关的东西，然后在让美术向里面填资源，策划按程序可以实现的功能做案子；当功能和美术表现上已经实现的程序不尽如人意的时候，程序员们已经脱身可以为下个版本的引擎做准备了。历史上成功的游戏大多延续开发了好几个版本，第一版很难有出色的表现，我们也不能苛求第一次制作的作品就能成为经典。

在程序制作中，很多项目开展之初就希望能提供给非技术人员一套顺手的开发工具，往往地图编辑器、效果编辑器开发的时间比主程序时间周期还长。而且在开发过程中不断地改进，还需要不断地更新生成资源的格式，一不小心甚至会影响到美术人员的工作。

其实，许多工具都是可以在开发初期省略的，尽量利用 Photoshop，3ds max，Maya 等的固有功能，让美术人员使用自己熟悉的工具工作，生成一些通用的好解析的目标文件。或使用自行写的控件把数据导出，或编写一些简单易用易实现自动化的命令行工具做文件转换。

这样可以节省程序员大量的先期工作，如果有制作开发工具的计划，一开始就着手准备是没错的，只不过要安排大量的时间让其稳定，而不要过早地投入实际工作。在《大话西游》的开发期，我们甚至没有编写一个地图编辑器，全部在 Photoshop 里用固有的图层功能标识出来，然后用命令行工具转换整合。直到项目进入维护器，真正意义上的地图查看及编辑软件才完成，游戏中所用到的自定义格式的精灵图片的查看转换工具才取代最初原始的无界面命令行工具。

从策划角度看，许多项目失败的原因是把游戏设计得过为复杂，超出了程序员能稳定实现的范围。程序勉强做出来，造成了许多的 bug。即使程序上能做得稳定，没有太多经验的游戏策划，过多的设定让游戏玩法失衡，游戏性大大减弱。在游戏的平衡性这点上，对于网络游戏尤

为重要。

虽然我是从事游戏程序开发的，并非专职策划，出于共同的道理，越是简洁的设计，越不容易出错。所以我个人提倡在完成了策划案后，先不要考虑娱乐性，从平衡性考虑，把可以删除的东西从案子中删掉，直到无可再删。以此为基点让程序把之实现，然后再考虑扩展，这样似乎更能做出一个成功的游戏。

编程和游戏

这章是本书的一个总结。本书以编程技术内容为主干，但却以非编程内容而结束。我因为爱游戏，才去制作游戏，因为制作游戏，才去学习和钻研编程技术。而最终，我是一个游戏玩家，好的游戏，而非好的游戏程序，才是我的目标。

与游戏程序员们共勉。

程序员到底应该在游戏制作的工程中占据一个怎样的位置？最早的游戏几乎都是由程序员单枪匹马做出来的，除了程序，还是程序。

而稍晚一点，出现了程序员和策划人员的分化，部分的程序员转向偏重游戏的设计，而另一部分着重去解决技术问题，主要是让低性能的计算机可以用一些取巧的方法做一些高难度的工作，而后策划人员从技术人员中彻底独立了出来，并且美术工作也分离为独立的职务，计算机游戏也不再是几个亮点几条线段的小玩具了。

可以说，游戏产业从程序员中诞生，但面对现代电子游戏，程序员又经常迷茫于自身的定位，到底是实现策划意图的工程师呢，还是把游戏当做展现自己编程技巧的舞台？我的回答，是也不是。作为程序员，首先是要实现策划所构想的游戏，同样也要充分展现自己的编程能力。

但几乎在成功的游戏开始过程中，程序员几乎都起着第二策划的作用。程序员是开始过程中最接近游戏的人，是最熟悉游戏的一切的人，他不仅在实现其他策划的想法，也在不断实现自己的想法，让指尖流出的代码变成互动的画面。

所以程序员同样应该去了解怎样让一个游戏变得更好玩，游戏性更多的是程序员不断地去修改代码中的细节慢慢改出来的，我们在玩别人的游戏时，也应该细心留意其中的细节，对他人的设计方法做一些揣摩。在沉浸于游戏带来的乐趣的同时，心里多一份冷静，以程序实现的角度去捕捉普通玩者容易忽略掉的细节。永远不要间断对怎样才能使游戏好玩这一问题的思考。

究竟什么才是衡量一个游戏好坏的标准？对此未能深入研究并付诸实践，是国内尚无令人非常满意作品的原因之一。如果让我给好游戏下一个定义，那就是能够让玩者沉浸其中，不能自拔。让我们先把电子游戏的范畴收得狭窄一点，把目光从现在流行的大型网络游戏上移开，暂且不考虑那种社会性为主的游戏方式。曾见到，有人昼夜不眠地扫雷，打砖块，或者面对屏幕上旁人看不明白的文字或者一些莫名其妙的图案、数据呵呵傻笑，这就是好游戏的魅力吧。

这里没有用具有华丽图形的Diablo，Starcraft，或者是给人带来如临

其境的 Quake，DOOM 做例子，虽然实际上它们可能有更广泛的玩家群。

这是因为基于我对游戏精髓的理解，使用更新的技术达到更高的视觉和听觉享受固然是其中重要的组成部分，但当我们忽略了本质的东西，去一味追求华丽的外表，将是最大的错误。

我个人认为，声光效果是为了利用人的好奇心把人带入游戏中来，但一旦沉浸于游戏中后，声光效果就不那么重要了。例外是有的，绚烂的声光效果可以麻醉人们平时生活的压力，让他们在游戏中不断追求更逼真真实的感觉。这迫使游戏制作商不断地提高游戏的拟真度，吸引这些焦虑的人们，但有一天，玩家们角逐五光十色的游戏的动力忽然消逝后，很可能就失去了玩游戏的兴趣。

实际上，游戏使人上瘾的因素很多，根本因素不是技术就可以决定的，技术只是一种手段，往往我们只是想看看游戏的结局而去玩游戏，这样，将通关动画的 avi 直接放在光盘上就极大损害了玩者的积极性；而有时，玩一个游戏仅仅因为想完成它，这个游戏肯定有一个很变态的难度设定，它被看做是一个挑战。去持续地玩一个明显很难的游戏并直到完成为止，这可能是一种满足自信心的行为。这样的情况都导致游戏被通关一次后，再不会去碰它，甚至因为开发者公布了一个修改游戏难度的密码，或者难度很方便地被修改掉，都可能让玩家中途终止玩这个游戏。

另一方面，一些游戏被人反复去玩，它可以被用来与人竞争（比如 Starcraft），这就需要一个合理完善且自由的规则，谁都希望能在某方面表现得比别人强，当然要是在公平竞争的条件下；或者玩家想提高游戏的操作技巧（比如 Virtual Fighter，侍魂，极品飞车，实况足球），然后老练的玩家能够向新手卖弄自己的技术，如果玩家在做一些动作，比如赛车的弯道技术，格斗游戏中灵活的拆招，新手和老鸟间没有什么差别，必然就没有人去反复研究它们了。

另外有的玩家想从游戏中得到探险的乐趣，解开一些谜题，找到一些隐藏的场景和情节；有些简单的游戏仅仅以看谁能获得更高的分数来吸引人。这些简单的设计虽然不对游戏的完成有太大的影响，但是很多玩家却会反复玩一个游戏，一些可以衡量玩家水平高低的数据可以使人

得到满足，甚至得到同一玩家群里其他人的尊敬。

很可惜的是，这些问题还没有得到国内游戏设计者的足够重视。技术派在挖空心思模拟出更真实，更绚丽的画面；创意派在为自己构思的游戏中一个绝妙的主意沾沾自喜。

这些都能让新的游戏更具震撼力，但我们是不是忽略了一些基础而重要的方面？

下面简单挑两个方面来论述。

14.1 操作

首先想谈谈目前我们的游戏制作者最需要重视的东西：操作。这并不是说，它比别的方面更重要，只是在我们看到了和国外成熟作品在美术方面的差距，或者是游戏系统的不成熟，亦或程序的不完善，往往忽略掉这个方面。

任何一个游戏都是让玩者和程序之间通过操作设备进行交互来产生乐趣。这是游戏存在的前提。建立一个有效的交互系统是困难的，通常，简洁的界面和简单的控制方法是受欢迎的。如果你在一个 RPG 游戏里设计一个进食的按键，要求游戏者定时按一下，保证控制的小人不至于饿死，这似乎增加了模拟的真实性，但肯定会受到大多数人的指责，因为它增加了控制的复杂度，却无甚意义。

能让人很快地上手，是促使玩者尽快投入游戏的重要条件，所以需要提供一个令人熟悉的控制方式，当我们熟悉了 StarCraft 里用鼠标右键命令单位运动，必然对用左键控制单位的 RTS 游戏生出一些不快，甚至一丝反感；当我们在玩某些中文 RPG 的时候，发现鼠标对主角的控制并不像 Diablo 里那样自如，而仅仅是起了个模拟键盘方向键的作用时，难免会有一些烦躁。

这些控制因素都影响着对游戏的投入，所以程序设计者们就需要反复考虑，当鼠标点在空地上要做什么操作，点在敌人身上有什么反应，点在不可达区域怎样，点在物品上怎样，点在 NPC 上怎样，左键点和右

键点的区别，普通攻击怎么发出，魔法怎样发出，游戏对玩者的操作的反应会不会和玩者所想产生差异……需要考虑的方方面面实在太多，很多细节并不是在游戏设计之初能够考虑全面的，需要反复测试手感，而很多国产游戏在这些方面做得还很不够。

Windows 操作系统使用得过于广泛，固然是人们习惯了一种通用的软件操作方法，但也桎梏了 PC 游戏开发人员的想像力。举个简单的例子，许多游戏都把操作局限在用鼠标去点击屏幕上的按钮，用键盘输入热键。但为什么很少有人两者结合起来？单独按下 A 键和一手按住鼠标的左键一手按下 A 键在程序逻辑上是可以被区分开为两种操作的。

这样双手配合的操作方式，比单手按 Alt-A 来赋予 A 键不同的功能要舒适得多。

早些年，很多人沉迷于吃豆、小蜜蜂、太空大战等，操作着飞机躲避子弹，打击敌人，或者在迷宫中穿梭，都是很有趣的，但是重新拿出这些游戏，仅仅能满足我们的怀旧情结，不可能让我们永远保持着对它们的投入感。这是因为，游戏的操作方式也会过时的。有一定游戏龄的玩家，心中总有几个经典，但是当再次将这些游戏拿出来玩的时候，又会觉得索然无味，很大程度上并不是因为画面的相对简陋，而是不再习惯那种操作。那些过时的操作方式影响了投入感，或者是那些操作不再能使人投入到游戏中。

对这一点，我重温几年前的RPG，三国系列甚至用 FC 模拟器玩儿时的动作游戏时深有体会。当我们审视并不长的电子游戏史各个阶段的经典作时，就会发现新的游戏总是有更新的人性化的交互系统使之比老游戏有更深刻的表现力度，当然这些是伴随精良的（软件/硬件）技术发展。

为了使玩者更加投入到游戏中，除了快截的控制方式外，迅速且合理的反馈也是同样重要的，这也是操作性的一个组成部分。如果在游戏中做了某一动作，而没有马上得到相应的回馈，就会让人失望。游戏中，玩者总是会将自己所见所感结合自己的想像，尽力投入游戏。

所以，当我们设计一个RPG，按动攻击键，被操作的小人就应该立刻挥剑，但是并没有碰到敌人（还有一小段距离），程序却判断攻击到了，

屏幕上的敌人显示出被攻击的画面。在程序设计师里眼里，可能无所谓，但是这破坏了玩者的想像（我并没有碰到敌人呀？）。虽然聪明的玩家马上理解了这一设定，但是却在某一程度上影响了投入感，就是平常我们常说的，这个游戏怎么玩起来这么别扭。同样的反面例子还有很多，比如某些设计不成熟的游戏中，经常出现的小人走路打滑，使人感觉人物和场景分离的现象等。

怎样加强人和游戏交互的临场感，一个例子可以说明问题。如果我们真正攻击到敌人，下一步，若敌人非常强壮，或者他对你的攻击做了隔挡，我们的手中武器将受到震荡，使短时间不能进行下一次攻击，甚至脚步倒退两步，这些反应都可能符合玩者对游戏发出攻击操作时的联想。游戏机设备上大多配备了震动装置来模拟这种感觉，在 PC 上即使我们不用什么力反馈设备作出物理上的反馈感，若弄在视觉上做到上面这些，也会极大地增加玩者的投入感，因为那些视觉和听觉之外应该得到的反馈信息已经由玩者的想像力去弥补了。

当你理解了这些，就能明白市场上充斥着模仿 Diablo 的游戏，能与之抗衡的却不多的原因。

14.2 角色设定

另一重要的是我们的游戏需要一个生动的角色。在RPG 里的重要性不言而喻。但是我们的设计者有时却只注意了游戏中需要表现的部分，而实际上，如果能尽可能地对游戏中出现的角色做详尽的设定是很有益的，比如人物的身高、体重、年龄、身世背景、喜好、武器的外形、名字、口头禅、习惯的动作等等等等，即使它们不一定会通过游戏给玩家讲述，但是在游戏中不经意地表达出来，在说明书里写进去，都会增进玩者对人物的认可，对发生在角色身上的事更在意，玩者也会被更深地带入到游戏世界中去。

在游戏中用更多的反映角色性格的对话和一些人物小动画，更是增加了游戏的气氛。像《最终幻想》系列就在角色刻画上做得很好，我的

很多朋友，都对这个系列游戏中的某些角色产生浓厚的兴趣，他们从哪来，喜欢什么，是什么类型的人，有些什么性格上的弱点，不同的人会中意于游戏中不同的某个角色。

如果想感受一下一个好的 RPG 如何通过画面中人物的语言动作塑造出生动的角色，有条件的话，应该看一下 NGC 上的塞尔达传说：风之韵。主角 Link 不断把自己的情绪表达出来，Link 的恐惧、痛苦、忧伤、迷惑、惊异都可以被玩家感受到，这是一种异样的真实，比之绚丽的光效更容易捕获玩家的心。当然作为一个被 FAMI 通评为满分的游戏来说，这些只是冰山一角，自己去体会能发现每个角落都被灌注了制作者的苦心刻画的细节，对于同为游戏开发人员的我们是需要学习的。

遗憾的是，现在我们的游戏在这方面还远远不够，角色形象都还比较单薄。欧美游戏中通常采取的自设计角色，和日式游戏先设计出一个有血有肉的主角，都是想让玩者能够认同所控制的那个屏幕上乱转的小家伙。而让玩者能全心投入的，往往只是那些穿插的无关情节发展的对话；对话时大头像眨巴一下眼睛，改变一下表情；场景中的小人偶尔做的一些小动作。随着机器内存/外存的增加，CPU 处理速度的加快，这样的细节描绘将越来越多。当你在感叹某个经典游戏给人的感觉如此之好，却又说不出好在哪里的时候，不妨多注意一下这些对角色刻画的细节。

14.3　再谈技术

技术问题，本才是程序员的专职，而上面说了那么多和专职无关的话题，除了想点出一些游戏设计中容易忽略的重要因素外，另一方面是想表明进行游戏设计中程序编写工作的人，并不是只去钻技术的牛角尖，且即使是上文中偶尔提到的细节描绘，有的看似简单，但在技术实现上却可能遇到不少麻烦。

很多时候，美术的表现力和程序员的实现是分不开的，同样外观精美的 Starcraft，玩家首先会感叹，暴雪公司的美术这么精美，而游戏程序员如我会首先惊异于 Blizzard 公司的程序师，居然在 256 色调色板模

式下，实现了如此逼真效果的黑雾、黑暗圣堂的半透明身躯和每个单位的半透明的影子。

当大多数玩家在惊叹《二战特种兵》那美妙绝伦的 2D 图形效果，并指责国内游戏的美术差距的时候，游戏程序员们会发现更多的东西：Commandos 里的小人都是带有 Alpha Channel 绘制在屏幕上，所以人物的轮廓上并没有锯齿或者黑边。3D 游戏到了 NGC 版的 Zelda 中，没有逼真的场景人物、五光十色的效果，只有卡通渲染出来的画面和任务丰富的表情，这些都不是简单地会用几个 3D API 谁都能实现出来的。

说起这里，很多人都会自然联想起 3D，2D 之争。有人指责国内的游戏技术不行的时候，总要说一句，人家老外什么都 3D 实现了，我们却老是做 2D 的；或者说 2D 的做不好，还做什么 3D 游戏。其实，图像最后的表现给玩家的感觉并不直接来处于 3D 技术或 2D 技术，而是看你对某种技术的挖掘程度，以及是否有好的解决方案利用现有的技术表现足够多的细节。

显示张图片大多数程序员都会，但是 2D 游戏做出来画面依然良莠不齐；渲染出一组模型有过一个月左右的 API 学习，大部分程序员都能办得到，可不堪入目的 3D 画面依然存在于太多市面上的游戏中。对于一个游戏来说，技术是非常重要的因素，它决定了游戏的表现力度，但是 3D 还是 2D 绝不成为技术高低的衡量标准。

使用哪种视角应该由游戏本身的需要决定，而且在短时间内全 3D 的游戏的视觉效果，还不能达到动画电影中的那种画面。也就是说，完全用 3D 实时渲染出所有画面还远远不能取代现在游戏开发中画面预渲染的工作。

基于 3D 技术搭建的游戏需要解决的技术问题往往是如何通过一些近似的算法而不是数学上严格的方案来快速地得到令人满意的画面；而 2D 引擎中，最需要人研究的是怎样控制和管理不断膨胀的图片数量。在未来的很长一段日子里，我相信会不断有人尝试综合实时渲染和预渲染的方案，找到一个最利于游戏表现的平衡点。

前面反复强调游戏让玩者的投入性，而做到这点，并不仅仅是精美的画面、动人的故事就够了的。画面固然精美，为什么用在游戏里感觉

差很多？为什么景物像纸片一样贴在屏幕上？为什么色彩凌乱？为什么人物动作干瘪？为什么就是看着别扭……这些很多属于程序在游戏中加工的工作；为什么游戏剧本那么吸引人，但玩起来却索然无味？

为什么游戏不能表达好剧本创作者的思想？为什么策划的意图不能得以展现……程序没有设计出强大的脚本语言支持，没有支持更复杂的场景表现，没有做到的事情太多了。

所以，程序必须为美术的表达、策划对游戏的演绎，做好坚实的后盾。怎样使游戏的画面表现得和宣传画里一样精美？怎样支持更多的场景中的小动画的数量，并使其错落有致？怎样使游戏更加流畅（这并不仅仅是优化代码，增加 FPS 数所能办到的，试问 VCD 的画面不到 30fps，为何自然流畅？）？当我们各方面细节都做得完美时，游戏能够在不经意的地方打动玩家，当玩家全心投入游戏中去，情不自禁地感叹，这个游戏我喜欢，那就是一个经典之作了。

我们的游戏程序员们，任重而道远。

14.4　浅谈网络游戏

这里所说的网络游戏更多的是 MMORPG，现在充斥着游戏市场的网络游戏已经和传统的游戏意义有所不同，传统的游戏强调的是玩家和游戏设计者的一种间接交流；而网络游戏则是由设计者提供一个公平的舞台，让玩家之间相互交流。传统游戏吸引玩家在于游戏它自身的娱乐性，而网络游戏更多的是来自于游戏营造的一种社会性，所以在设计和制作网络游戏时，应该有一些不同的思考角度。

鉴于网络游戏的社会性，我们更加需要在游戏中设计完备的经济系统，追踪货币的流通、实物的交换，弄清游戏社会中的经济是如何运作的。这表面上是不需要程序考虑的问题，由策划人员去设计。

但实际上，不光大规模的数字采集和统计的工作，非程序的策划很难去做，而且由于大规模的经济控制并不是简单的服务器程序架构就可以实现出来的，所以为了减少数据库的负担，减轻服务器之间通讯的带

宽压力，我们必须在设计上针对实际硬件水平做出精简，这只能是熟悉软件架构的人员才适合去做的事情。

网络游戏的生命期也远较之传统游戏长，一个玩家沉浸其中后往往需要游戏数月甚至几年。同时间交互的玩家也少则几百多则上千，这样玩家间既需要产生差异性又需要达到实力的平衡，这个矛盾就一直存在于一个网络游戏的生命期内。平衡度的调整是网络游戏中最复杂的课题之一，从传统游戏中过渡来的方法，大多是靠经验而不是数学推算来解决，慢慢的会变得不那么有效。

好在程序员大多是理工科出身，普遍有不错的数学基础，又有熟练使用计算机做运算工具的能力，希望随着网络游戏的飞速发展，为游戏平衡性而发明的更理论化的数学工具可以尽快出现并完善。

14.5 小结

如果游戏程序员，尤其是一个项目的主程，只把心思放在编程上，那么很难促成游戏的成功。游戏中的细节往往在制作过程中被发掘出来，和策划人员不同，程序员是最快接触到开发中的游戏的，对于游戏的表现有切身的体会，也是最方便对游戏做修改的人。一个好的游戏程序员和游戏策划之间的界限会变得很模糊，所以游戏程序员除了编程方面的修养，各个领域的知识都应该有所涉猎，以提高自身的修养。

后 记

几年前，我在北京认识了孟岩，他刚翻译完《C++标准程序库》。那天，我第一次听到朋友的建议，应该把数年游戏开发中的感触和理念整理成册。

去年（2004）四月的时候，《程序员》杂志的闫辉介绍我认识了现在这本书的编辑朱沐红。我已经不记得当时是怎么决定真的动手写这么一本书的。虽然我更早的时候，喜欢在网络上随便写点技术文章。但现在看来，当年许多所谓的技术连载文章，都是略带幼稚，不登大雅之堂的。如果真是白纸黑字地印出来，真的会贻笑大方。

五一回家，读了新出的《DOOM 启世录》，感触颇多。原来，地球那一边，名满全球的游戏程序员，本质上还是一个普通人嘛，就和你我一样。从一个游戏玩家、技术狂热者，成长为优秀的游戏制作者。如果我能把自己的成长历程介绍给国内热爱游戏，想成长为优秀的游戏程序员的人们，对于他们说不定有更多的启发呢？所以过完五一长假，便回信答允写这样一本书。这本书最早理出的提纲，就是以自己的成长为线索的。

但是我并不想把书写成自传，无论是自我肯定还是否定，那样都显得过于自私。叙事型的文字要好写一些，毕竟只需要把自己的记忆、自己的知识，自然地表达出来就可以了。可是，这样就逃避了技术书最重要的一点，帮助读者归纳总结技术点的精华。毕竟并不是写小说，我更希望读者能从书里多领略一些我对技术本身的看法，技术问题的要点，而非单纯的文字阅读的乐趣。读技术书不会是太轻松的事情，而写作更应该花掉数倍的精力。基于这点想法，我后来的章节改变了写法，最后又将叙事味道较浓的前五章，全部打乱重新写了一遍。

大凡第一次写技术书的人，都想穷尽自己所知。翻翻这本书的目录，也能看出此种迹象。其实，最初的时候，我想写的更多。后来，我把书里最初完成的章节，例如汇编、Windows操作系统等，做成 PPT 放到公司内部的培训课程中，给新员工讲解。这才发现，几乎每个细节都有很大的发挥余地，越是基础的知识越难讲得面面俱到。课堂中如有错误，可以即时跟听众道歉，更正；有争议的地方，大家能够聚在一起讨论。但是，写进书中，就必须小心谨慎了。所以，越写越怕，甚至有些畏缩，那些不是特别擅长的领域的内容，干脆回避不写了。即使这样，我依旧不敢保证书中那些自以为精通的领域中的论点，或是对问题的阐述不犯错误。而以自己的个性，又不屑于从前人成熟的书籍中摘抄结论。只好等着

读者的批判了。

技术问题，固然本源的方法，理论变化不大。但计算机这种偏重工程的领域，一旦实际化到具体上，却有着日新月异的变化。有着钻研精神的程序员，想把任何知识搞清来龙去脉，都不是朝夕之功，只好借着自己对旧知识的理解去参悟新的知识，触类旁通。但写书不能这样，我想像中的读者，绝对不愿意看到对新知识的错误理解的。所以，我只好写一些更久以前，被自己反复理解过的内容。确实在最近一年中，我总是生出一些冲动，想把正在研究的东西写进书里，跟大家分享一下学习创新的快乐。最后，都还是按捺下来，老老实实地修改旧稿。

智者不擒二兔。我必须承认，我这过去的零四年里犯了很大的错误。算上写书这只兔子，我在追逐的绝对不只两只了。网络游戏的繁荣，让我的工作也特别忙。即使公司一贯地来帮我保持一个自由宽松的研究环境，而我心中依然有好几件事在同时进行。这导致了写这本书的实际时间非常零散，一直静不下心把握书的整体。只能是隔上几周，什么事情都放下来专心写上一章的文字。书的内容本身远远没有达到我希望的质量，对此我甚是遗憾。当然另一方面，我也感到欣慰，很多一直想说的话，终于有个窗口直接表达出去了。毕竟我还是花了许多的心血来完成其全部。

致　谢

首先，我最想感谢的是我的父母。

我的父亲，一个平凡的工程师，默默无闻地在他的工作岗位上为社会做着自己的贡献，但在我眼中，他是一个不平凡的人，一个我所崇敬的人。

父亲在他能够接触到的领域内，几乎无所不精。他是一个博学多艺的人，这深深影响了我的童年。当我骑着出自父亲之手的木马，玩着他削出的玩偶，拿着精美的风筝出去春游的时候，也明白了，自己想要的一切原来都是可以用自己的双手来创造的。当然，包括我后来接触到的电脑游戏。

是父亲引导我进入计算机的世界，虽然他也曾经教过我画画，或是启发我对数学、物理等的兴趣。但是在我选择了计算机游戏这条路后，父亲却一直坚持维护着我自己的爱好，在最大限度上，让我的童年没有受到应试教育的干扰。记得在初中毕业前夕，我曾经萌发过放弃上重点高中乃至上大学的想法。但是，面对盛怒之后老泪纵横的父亲，第一次看到父亲的眼泪时，我理解了，理解了一个求知欲如此之强的人却由于时代的错误，甚至连高中都没有机会去读，在繁忙的工作中，熬夜自学考上大学的人，怎能不对自己的儿子在学业上有更高的期望呢。后来，我考上重点高中，并且升入重点大学，我知道，父亲是高兴的。而我，也发现，这些被人抨击的中国式教育制度，其实对于自己的人生，也是收益良多。

父亲淡泊名利，一心求知。几十年勤勤恳恳为单位工作，无留有半点积蓄。九十年代初，曾经有几十万的工程项目找到过他，但是却被一个月只拿几百元工资的他推掉。理由仅仅是为单位工作应该尽心尽力，如果心有旁骛，情理不合。这些都深刻影响了我的人生观。当一个人可以心无所求，执意做一件事情时，总有一天，做过的所有就会反馈丰厚的回报。这样的人生是快乐的。

而我的母亲，一个很多方面和我的父亲截然不同的人。她爱着父亲爱着我，为我们付出了许多。即使我独立生活多年来，她依然在生活上给予了我太多的照顾。但母亲给予我的绝对不仅仅是这些，更多的是为人处世的方法。

我继承了父亲的傲气和特立独行，做着自己认为对自己应该做的事情。大四那年，我与学校老师闹了一点矛盾，差点自己退学。最后，这件事情仿佛化于无形之中，我的记忆中只留下，母亲匆匆赶来学校对我的心情做过一些安慰。事隔多年，我才知道，母亲在这件事情中活动良多。她不愿意告诉我这些，因为她认可我的处事方式，却又有自己对人与

事的理解，直到她认为我更成熟了一些，才讲给我听这些旧事。类似的事情，她也为父亲甚至她的朋友默默做过更多而不说出来。母亲是一个绝对正直的人，这点不亚于我的父亲。她用无私的心去帮助她身边的很多人，帮助家族中的亲戚，资助毫无关系的大学生读书，几十年如一日地照顾没有血缘关系的一对老人。对于社会上诸多的不合理，她却有着不平凡的手法去面对这些现实。入世且脱俗，我能从她那继承点滴已经是获益匪浅了。

正是我的父母，让我感受到这个世界的美好，我才能最终做我想做的事情，才会在这个时间有了这么一本书。

其次，我想感谢在写书期间对本书有过直接帮助的人们。

我的编辑朱沐红，她是除了我自己之外唯一的一个从始至终用心在这本书上的人。没有她的鼓励和监督，我真的无法坚持完成。来来去去几十封 E-mail，至今保存在我的硬盘上。而且我知道，编辑和修改书稿，并不会比创作本身轻松很多。

只见过一面，却在网络上交往颇多的好友孟岩。他耐心地阅读了我最初完成的许多章节，并给予了诸多意见。最后，应诺帮我的这本书作序。对于那篇序中对我的个人评价，我觉得有些不敢当。但是的确反映出我想表达却没有用文字述说的内容。

我的同事，王远明，庞伟，关子敬，等等许多人（请原谅我没有一一记录下你们的名字）。书中的许多技术细节，都来自于和他们的讨论。正是在这些讨论的过程中，让我更正了原来自己的一些理解错误。

还有我的同事兼领导，詹钟辉，丁磊。他们给了我一个非常宽松的工作和研究环境。我甚至可以在工作的空挡记录下思绪点滴，放进书中。如果没有这种环境，很难想像书要到什么时候才可以出版。

当然还有我们所尊敬的游戏圈中的前辈，姚壮宪。他在大约我现在这个年纪时，开发了一代巨作——《仙剑奇侠传》，深深影响了一代游戏现象。游戏程序、策划、美术均由其一人把握，着实让人倾慕。有这么一个作品也是我的愿望，可惜一直未能如愿。2000 年的时候我们曾经有几次 E-mail 交流，但无缘相见。这次我请姚仙为本书作序，他竟欣然答应，让我欣喜不已。

最后，我感谢网络上跟我交流过的所有人。包括和我通过电子邮件讨论技术问题的人；我的个人主页留言本上，水木清华 BBS 的游戏制作版，新浪游戏制作论坛上的诸多网友。无论你们的观点是认同、反对、赞美还是对我口诛笔伐，我都同样地感激。正是这些交流，促进了我自身知识的成长。这些知识构成了这本书。

云风 2005 年 5 月 23 日晚 广州

参考书目

1．**图形程序开发人员指南**

Michael Abrash's Graphics Programming Black Book Special Edition

ISBN 7-111-06396-1/TP.799

Michael Abrash 著　前导工作室 译

机械工业出版社

2．Intel **体系结构** MMX（tm）**技术开发指南**

Intel Archirecture MMX(tm) Technology Developer's Manual and Programmer's Reference Manual

ISBN 7-5053-4183-9/TP.1866

Intel 公司著 李晖 译

电子工业出版社

3．Windows 2000 **编程技术内幕**

Programming Microsoft Windows 2000 Unleashed

ISBN 7-111-07632-x/TP.1271

Mikey Williams 著 前导工作室 译

机械工业出版社

4．COM **本质论**

Essential COM

ISBN 7-5083-0611-2/TP.191

Don Box 著 潘爱民 译

中国电力出版社

5．**程序员修炼之道**

The Pragmatic Programmer

ISBN 7-5053-9719-2

Andrew Hunt/David Thomas 著

马维达 译

电子工业出版社

6. C++编程思想 第一卷：标准C++导引

Thinking in C++ Second Edition Volume One: Introduction to Standard C++

ISBN 7-111-10807-8/TP.2560

Bruce Eckel 著

刘宗田 袁兆山 潘秋菱 等译

机械工业出版社

7. More Effective C++中文版

ISBN 7-5083-1486-7

Scott Meyers 著

侯捷 译

中国电力出版社

8. Modern C++ Design - Generic Programming and Design Patterns Applied（影印版）

ISBN 7-5083-1496-4

Andrei Alexandrescu 著

中国电力出版社

参考网站

http://www.agner.org Agner Fog Research

http://www.codingnow.com 云风工作室

http://msdn.microsoft.com MSDN

http://www.talula.demon.co.uk/allegro Allegro: A game programming library

http://www.intel.com/design/pentium4/manuals/index_new.htm IA-32 Intel(R) Architecture Software Developer's Manuals

http://www.research.att.com/~bs Bjarne Stroustrup's homepage

http://osl.iu.edu/~tveldhui Todd Veldhuizen's homepage

http://www.lua.org The Programming Language Lua

http://bdn.borland.com/museum/ Borland Developer Network Museum

Broadview®
www.broadview.com.cn

《游戏之旅——我的编程感悟》读者调查表

尊敬的读者：

感谢您对我们的支持与爱护。为了今后为您提供更优秀的图书，请您抽出宝贵的时间将您的意见以下表的方式及时告知我们（可另附页）。我们将从中评选出热心读者若干名，免费赠阅我们以后出版的图书。

姓名：	性别：☐ 男 ☐ 女	年龄：	职业：
通信地址：		E-mail：	
电话：	传真：	邮编：	

1. 影响您购买本书的因素（可多选）：

☐ 封面封底　☐ 价格　☐ 内容提要、前言和目录　☐ 书评广告　☐ 出版物名声
☐ 作者名声　☐ 正文内容　☐ 其他_____

2. 您对本书的满意度：

从技术角度　☐ 很满意　☐ 比较满意　☐ 一般　☐ 较不满意　☐ 不满意
☐ 改进意见_____

从文字角度　☐ 很满意　☐ 比较满意　☐ 一般　☐ 较不满意　☐ 不满意
☐ 改进意见_____

从版面、封面设计角度　☐ 很满意　☐ 比较满意　☐ 一般　☐ 较不满意
☐ 不满意　☐ 改进意见_____

3. 您最喜欢书中的哪篇（或章、节）？请说明理由。

_____

_____

4. 您最不喜欢书中的哪篇（或章、节）？请说明理由。

_____

_____

5. 您希望本书在哪些方面进行改进？

_____

_____

6. 您感兴趣或希望增加的图书选题有：

_____

_____

通信地址：北京市万寿路 173 信箱博文视点　收（100036）
电话：010-51922832　　E-mail:jsj@phei.com.cn；editor@broadview.com.cn

Broadview®
www.broadview.com.cn
技术凝聚实力·专业创新出版



我现在是中国并不成熟的游戏制作行业中的一员，游戏给了我太多，我告诉自己需要做一点事情。分享知识和经验是我的义务，别无它。

——云风

图书分类：程序设计〉游戏

ISBN 7-121-01609-5

9 787121 016097 >

网上订购：www.dearbook.com.cn
第二书店·第一服务

博文视点

责任编辑：朱沐红
责任美编：张子建

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 7-121-01609-5　定价：46.00元