快乐写游戏　轻松学编程

PC 游戏编程（网络游戏篇）

CG 实验室　王鑫　罗金海　赵千里　编著

清华大学出版社
重庆大学出版社

内容简介

本书的作者都是第一线的网络游戏开发人员，书中的所有内容都整理自完整的网络游戏项目，是实践经验的总结。第二章到第十四章的内容，基本是从一个多人在线冒险型网络游戏项目中整理而来，第十五章则整理于一个即时战略游戏项目。在内容的讲述中尽量避免了和具体游戏内容的关联，以便更具有通用性。

本书的目标是帮助读者掌握常见类型的网络游戏的开发环境、流程、关键制作技术、方法和技巧，读者通过学习和实践甚至可以成为专业的网络游戏程序设计师。本书面向的读者对象主要是有一定编程经验，并对制作网络游戏很有兴趣的爱好者或者是有单机游戏设计经验，正在向网络游戏转型的程序设计师。同时对于非程序开发，但对网络游戏项目很有兴趣的朋友也是很好的参考书。

图书在版编目（CIP）数据

PC 游戏编程. 网络游戏篇 / 王鑫，罗金海，赵千里编著. —重庆：重庆大学出版社，2003. 8
（快乐写游戏轻松学编程）
ISBN 7-5624-2766-6

Ⅰ. P… Ⅱ. ①王… ②罗… ③赵… 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（2003）第 060107 号

快乐写游戏 轻松学编程
PC 游戏编程（网络游戏篇）
CG 实验室 王鑫 罗金海 赵千里 编著
责任编辑：何 明 张 彬 版式设计：吴庆渝
责任校对：任卓惠 责任印制：秦 梅

*

清华大学出版社
重庆大学出版社 出版发行
出版人：张鸽盛
社址：重庆市沙坪坝正街 174 号重庆大学（A 区）内
邮编：400030
电话：（023）65102378 65105781
传真：（023）65103686 65105565
网址：http://www.cqup.com.cn
邮箱：fxk@cqup.com.cn（市场营销部）
全国新华书店经销
重庆科情印务有限公司印刷

*

开本：787×1092 1/16 印张：16.75 字数：418 千
2003 年 8 月第 1 版 2003 年 8 月第 1 次印刷
印数：1—5 000
ISBN 7-5624-2766-6/TP·403 定价：36.50 元（赠 1CD）

总 序 言

陈 其

《快乐写游戏 轻松学编程》丛书是重庆大学出版社为广大计算机编程爱好者和电脑游戏玩家送上的一份厚礼，是一套集学习、娱乐于一体的，全新教授模式的好书。全套书由陈其总策划，在多维图书策划中心以及各游戏工作室的鼎力协助下得以顺利出版。现就丛书的有关问题作出说明。

编程和游戏

程序是计算机的灵魂，掌握了编程技术就可以随心所欲地让计算机为你服务，让它实现你的梦想。但学习过程中大量的命令和语句又让人感到枯燥乏味，而每一个学编程的人都有过面对一大堆熟悉的命令却组织不起一个像样的程序的经历。于是我们联想到了一种让很多朋友都着迷的程序——电子游戏。

1）第 9 艺术

电子游戏如同戏剧、电影一样，是一种综合艺术，并且是更高层次的综合艺术，它的出现代表了一种全新的娱乐方式——交互式娱乐（Interactive Entertainment）的诞生，而且从它的诞生到现在一直以其独特的魅力吸引了许多玩家，同时也激发了更多的人想写游戏的愿望。

一种事物，当它具有丰富而独特的表现力时，当它能给人们带来由衷的欢愉时，当它表现为许许多多鲜明生动的形象时，它就是一种艺术。电子游戏已经成为一门艺术，继绘画、雕刻、建筑、音乐、诗歌（文学）、舞蹈、戏剧、电影（影视艺术）之后人类历史上的第 9 艺术。20 世纪 70 年代，出现了第一批简单的电子游戏；今天，它已经发展成为拥有亿万游戏迷的独立的新型艺术样式，向世人显示了其强大的艺术生命力。《文明》、《Doom》、《魔法门》……一个又一个奇迹在产生，进入这个行业成了很多人的梦想。娱乐界的大腕：卢卡斯、派拉蒙、华纳等都已致力于电子游戏产品的开发，并推出了一大批优秀的交互式电影（Interactive Movie）。在世界范围内，电子游戏业的利润已经超过了美国的电影工业和日本的汽车工业。相信不久的将来必然有一大批杰出的电影导演和真正的艺术家投身于电子游戏艺术作品的开发。而 VR 头盔与 3D 音效卡的诞生已使电子游戏远远跳出了一般电影所能达到的视听层次。可见，电子游戏已经将视听综合艺术推向了一个崭新的高度和崭新的领域。

在中国，电子游戏曾一度被称为是“电子海洛因”，一些教育界人士痛斥电子游戏是如何

毒害青少年,如何损害人的健康。其实与其千方百计扼杀它,还不如共同想办法来扬其长、避其短。因为绝大多数反对电子游戏的人,并不是反对电子游戏本身,而是反对电子游戏中存在的消极面。正如水能载舟亦能覆舟的道理一样,任何事物都有其两面性,关键是怎样利用好的那一面为人类造福。

本丛书正是要利用电子游戏的积极面,将枯燥的学习融入轻松的游戏之中,达到喻教于乐的目的。

2)培养全局观

许多刚学编程的朋友总是把大量的精力花在了命令和语句上,或是集中精力去学习那些复杂的函数。他们都忽略了怎样去实现一个完整的程序,所以有很多初学者到现在还没写过一个完整的程序。为了避免这种情况,在编写游戏实例时,各书都使用了简单而功能强大的游戏开发引擎,读者能非常轻松地学会如何显示图像文件、播放声音及控制输入设备等游戏中必备的功能,然后把注意力集中到如何实现一个完整游戏的过程及原理上来。

通过细致的讲解,读者朋友很快就能从实例中体会出程序全局观的作用和地位,并在一步步的学习后掌握它。

编程工具

作为一名程序员,要做的第一件事就是选择一把顺手的武器——编程工具。做程序的朋友都知道,比较流行的编程工具颇多,比如:VC,VB,DEPHI、汇编等等。由于本丛书是从编写游戏出发的,而为了能够完成一个完美的游戏,编程工具应具有贴近底层、代码运行速度快、便于优化等优点。于是 VC 成了不二之选。

初学 VC,会因为观念的改变而不知所措。其实,每个人时刻都面临着新知识的学习和旧知识的更新。这就好比,只有踏出新的一步才能前进。那么如何才能更快的学会程序(游戏)开发呢?很简单,那就是"边学边做"!所谓知识来源于实践,做做学学,学学做做,这样你很容易就能融汇贯通了。所以,首先了解一些 VC 使用常识,照着书中的一些简单的例子一步一步的实际操作,从中学会一些基本的游戏开发常识。然后学习一些 C++理论知识,选一些难一点的例子来学。之后再学一些游戏开发的高级技术,试着自己开发一个游戏出来。罗马不是一天建成的,饭不是一口吃得完的,游戏也不是一会就能做出来的。所以,每天砌一块砖,不久一座美丽的城堡就矗立在你面前了。

衷心祝愿每位读者能在本丛书中吸收到有用的知识。

前言

网络游戏无疑是近年来电子娱乐行业中最亮丽的风景，甚至整个信息产业也为之振奋。随之而来的是网络游戏的开发也变成了软件项目的大热门，大大小小的网络游戏开发公司如同雨后春笋一般涌现，而关心和投入到网络游戏开发的程序设计师也变得多了起来。但是从书刊到网络，关于网络游戏程序开发的资料，尤其是中文资料，还处于很稀缺的状态。希望本书的出现能够填补一块小小的空白。

本书的目标是帮助读者掌握常见类型的网络游戏的开发环境、流程、关键制作技术、方法和技巧，读者通过学习和实践甚至可以成为专业的网络游戏程序设计师。本书面向的读者对象主要是有一定编程经验，并对制作网络游戏很有兴趣的爱好者或者是有单机游戏设计经验，正在向网络游戏转型的程序设计师。当然对于非程序开发，但对网络游戏项目很有兴趣的朋友也可以来看看这本书，毕竟他山之石，可以攻玉嘛。

关于本书内容和章节安排的说明

本书的作者都是第一线的网络游戏开发人员，书中的所有内容都整理自完整的网络游戏项目，是实践经验的总结。这些经验代表的是开发上各种可行的方案方法，而不是教条准则，读者尽可以带着怀疑和改进的心态来阅读它。第二章到第十四章的内容，基本是从一个多人在线冒险型网络游戏项目中整理而来，第十五章则整理于一个即时战略游戏项目。在内容的讲述中尽量避免了和具体游戏内容的关联，以便更具有通用性。

关于本书出现的代码的说明

本书中出现的代码主要有三类：

1. 使用具有明确意义的英文变量和函数名，加上中文描述组成的伪代码，采用这类形式的代码主要是因为当前讲述的内容不适合使用大量的实际代码，这些代码会和游戏中的其他功能的代码混合在一起，而且篇幅巨大，既不容易看懂，也无助于理解。所以这部分内容主要靠结合伪代码来讲述思路、算法和流程等，例如服务器架构和即时战略游戏网络对战的部分内容就是以此类代码为主。

2. 对于一些很独立的功能模块的代码，我们用代码结构说明加上实际代码，再加上详细注释的方法来处理。这类代码通常只需略加改动甚至可以直接被读者使用，比如跨平台的 socket 类、Lobby 大厅的制作、自动更新系统、可自定义变量和函数的表达式脚本等。

3. 另外有些代码是很常见的算法或功能函数，为了不占用篇幅，就不列出详细内容，这类代码要么出现在附带光盘上，要么在网络上非常容易查到。

关于作者

在章节的编写安排上：

第一章、第六章、第十章、第十二章、第十三章和第十五章由王鑫编写。

第三章、第四章、第五章、第八章、第九章和第十四章由罗金海编写。

第二章、第八章和第十一章由赵千里编写。

感谢重庆大学出版社和重庆拓智文化发展有限公司（www. topwise. cn）！感谢所有给予我们关心和支持的朋友！

如果读者在阅读本书的过程中，产生疑问或者愿意帮助指正书中的错误和不足之处，可以发 Email 至 ryan _ www@21cn. com ，我们随时欢迎你们就有关问题进行交流、切磋！

王　鑫

2003 年 5 月于上海

目录

第1章 网络游戏开发概述

1.1 网络游戏的类型

Client/Server 结构是现今网络游戏最基本的框架。从开发的角度来看,常见 Client/Server 结构的网络游戏有如下几种类型:

(1)对等的 Client 和 Server

很多对战型的网络游戏都采用这种结构。这里所谓的对等并不是真正意义上的对等,而是指在有多个玩家参与的游戏中,其中一个玩家的机器既是 Client 又扮演 Server 的角色,通常由创建游戏局的玩家担任,称为主机。这种结构如图 1.1 所示。

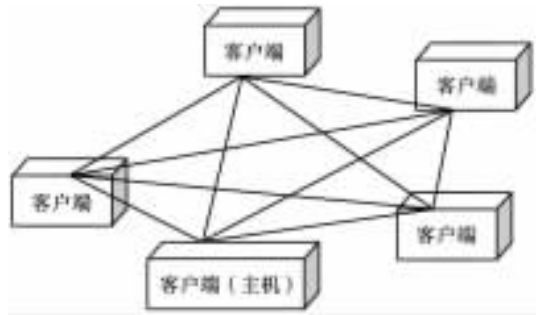


图 1.1

在这里,主机的作用包括同步的控制、消息的集中转发、关键计算的结果校验等等,取决于具体的游戏类型。具有网络对战功能的即时战略游戏、第一人称射击游戏以及动作 RPG 游戏多属于此类,它们通常既可以在局域网上玩,也可以在 Internet 上玩。

(2)会话的集散地:Lobby

首先明确会话和大厅的概念。英文 Lobby 是大厅的意思,以一个提供聊天服务的网站为例,登录之后可以看到网页上有可供选择的话题 A 和话题 B。选择话题 A 点击进入,便可以和所有位于话题 A 的人聊天。此时,我们把话题 A 叫作 session(会话),并称所有位于话题 A 的人正在进行一场会话。并可将这样一个聊天室网站称为大厅。可以根据自己的需要创建会话或者加入别人的会话,也可以取消自己所创建的会话。实际上,大厅就是一个专门的服务器,其作用是为处于不同位置的玩家牵线搭桥,让他们可以有机会进行同一场游戏。可以很快联想到世界上最著名的游戏大厅 BattleNet,其中有《暗黑破坏神》、《星际争霸》和《魔兽争霸》等广受欢迎的游戏,还有国内非常著名的游戏网站联众,上面运行着各种棋牌类游戏。下面来看看大厅和会话的架构,如图 1.2 所示。

当大厅作为中介把客户端撮合到一起之后,各客户端就可以自行开始游戏了,随后的处理要看具体的游戏设计。游戏在进行中,既可以继续保持和大厅的连接,并不断地汇报一些游戏的信息,也可以断开和大厅的连接,独自进行游戏,游戏完毕再重新连上大厅服务器,并由主机玩家向其汇报战果。

(3)Client:Server 的一个可以交互的窗口

并不是所有的网络游戏均用对等的 Client 和 Server,并借助大厅的力量运作起来。很多时

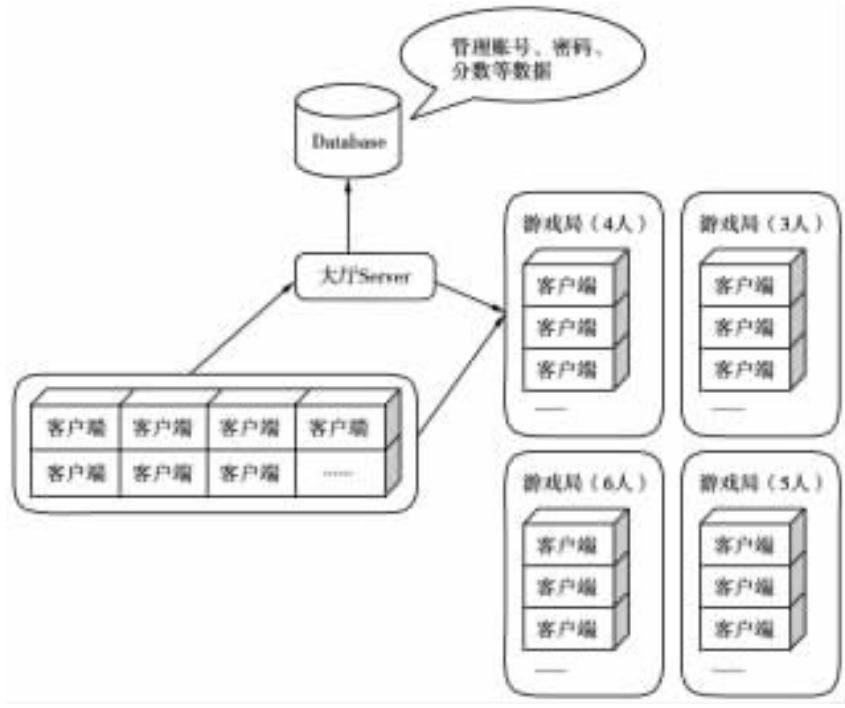


图 1.2

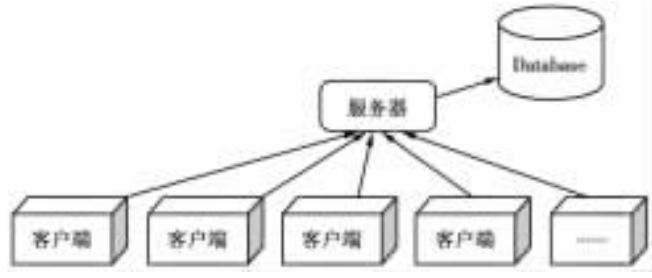


图 1.3

候，需要的是成千上万的人在进行同一场游戏，这些玩家在游戏世界中持续存在并且被记录下来，而这种游戏才是现在最大的热门，通常可以称之为图形化多人在线 RPG 游戏。这种游戏必须用一种新的架构来实现，图 1.3 所示为一简略的结构示意图，实际应用中为了满足海量人数以及在线互动的需求，结构会复杂得多。

在这种结构中，为了保证所有客户端都能拥有相同的游戏画面和游戏结果，关键的数值计算和逻辑判断必须放在服务器上，因此服务器上必须具备完整的游戏世界模型。此时客户端更像是在一扇窗口中，从玩家的角度去观察这个世界，并与之互动。服务器和客户端的连接方式和功能划分与前面提到的两种有很大的不同，以一个切景式回合制战斗的网络游戏为例，功能结构图如图 1.4 所示。

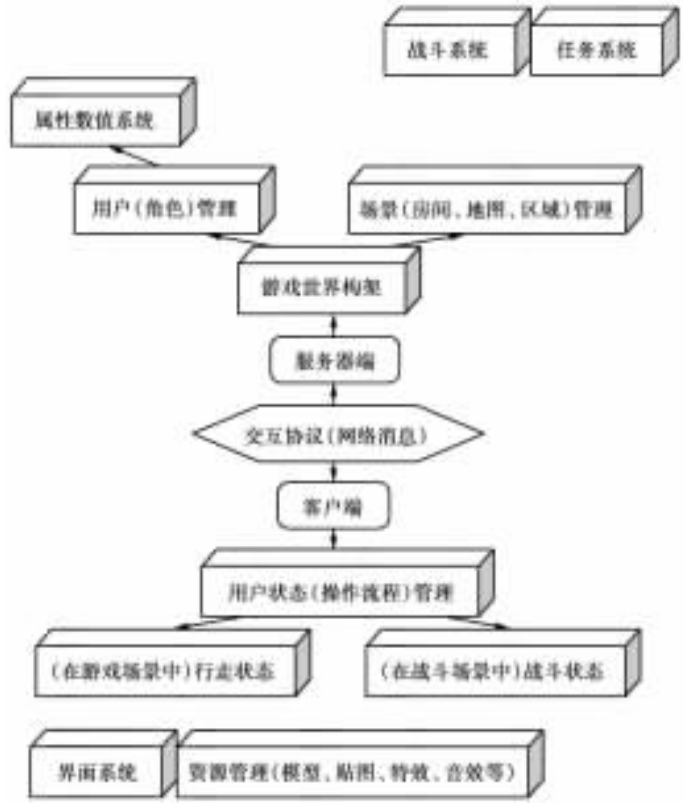


图 1.4

1.2 网络游戏的开发环境

当今,游戏制作已经成为一门分工极其细致的专业,其开发环境已不能简单地指一两种编程语言的编辑环境。网络游戏类型各式各样,实现起来针对性非常强,光是开发环境就是一项很复杂的课题。开发一个中等规模的多人在线游戏,可能会使用五六种开发工具。而且,服务器端和客户端的运行环境往往处于不同的操作系统。前者为了稳定性和经济性通常会选择 Linux 或 FreeBSD 之类的操作系统,其开发工具可能是 GNU C ++,也可能是某种类 C 的脚本语言,数据存储可能用 MySQL,乃至 Oracle;后者通常会在 Windows 操作系统下运行,使用主流的游戏开发工具 VC ++,协作开发控制工具 SourceSafe 或 CVS。这些还只是开发的基础工具,各家公司还会有自己的游戏开发引擎以及辅助制作的工具。开发环境的选择是影响游戏制作成败的关键因素。

一般来说,虽然服务器端的运行环境是在 Linux 下,实际开发基本上还是在 Windows 下进行,以便使用强大而方便的 Visual Studio 集成环境。调试通过后,再到 Linux 下用 C ++ 编译即可。Linux 下虽然没有 Debug 集成调试功能,但程序如果运行出错,可以产生 core 文档用来查看出错的代码位置。跨平台开发时,要特别注意平台相关的一些系统 API 调用。

为了方便跨平台操作,可以在 Linux 服务器上安装 SMB 文件共享服务和 CVS,但 CVS 并不用来维护源代码的版本,而是用来维护客户端以及服务器运行所需的资源文件。这是因为

开发人员大都工作在 VC ++ 环境下，源代码版本控制使用 SourceSafe 最方便不过了，而游戏运行环境中有大量的各种类型的文件，用 CVS 维护就比较方便。这样，策划、美术、程序的各类资源都有版本维护的工具，能提高开发的效率。

图 1.5 是一张开发环境示意图，适用于部分类型的网络游戏开发。

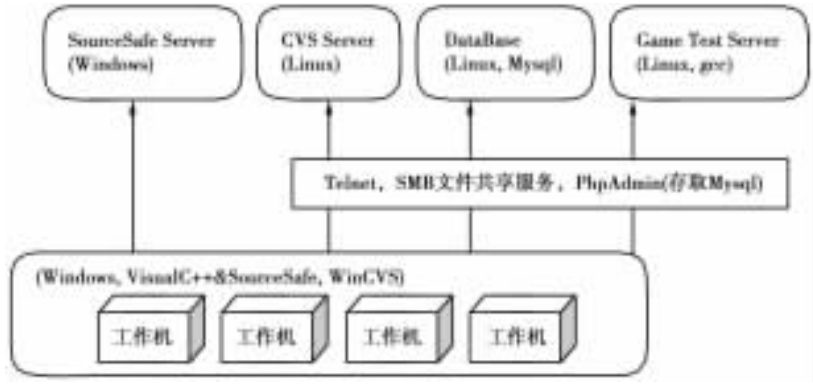


图 1.5

1.3 开发注意事项

从程序设计的角度出发，首先要明确一点，在进行网络游戏开发时，最重要的是设计思想、框架结构、数据组织等等，具体编程语言的技巧应该放到次要的位置上。所以，在面对一个网络游戏的设计时，首先要考虑的是如何建立框架结构，划分大的功能模块，确定模块之间的层次关系等。

在设计游戏模块的过程中，应该尽量将游戏内容模块和引擎模块分离开来。引擎是下一个游戏中可以重用的部分，因此这样做可以使游戏结构清晰、容易维护，也节省了开发时间和成本。但怎样划分内容模块和引擎模块却是一件非常讲究的事情。若偏向前者，则引擎会被游戏内容特例化，也就是说下一个基于该引擎的游戏只能做同类型的，甚至只能是一个资料片版本。事实上，很多游戏开发引擎被反复利用来做相同类型的游戏，开发商也没有觉得有什么不好。但作为开发人员，仍需要仔细设计这些模块，尽量提高代码的利用率和可维护性。

第2章 网络通讯底层设计与实现

本章主要讲述构架一个网络游戏所必须的底层功能模块。在设计与编写一个规模较大的软件时,需要把整个系统切分成数个小模块,以方便多人同时编写,而且也有利于将来维护与扩充。

传输层选用 Socket 接口进行开发。这样一来避免了开发人员直接面对复杂的网络协议,同时又有一定的灵活性,可以在 Socket 层上构架自己的传输协议。

2.1 Socket 简介

Socket(套接口)是一套用于网络数据传输的编程接口。它最早应用在 1983 年发行的 BSD 4.2 操作系统中,由 Berkeley 计算机系统研究组(CSRG)开发,所以又常被称为 Berkeley Socket。

许多 Unix 系统直接沿用源自 Berkeley 的网络支持代码。Linux 系统的网络代码是重新编写的,但接口规范还是与 Berkeley Socket 相同。Windows Sockets 规范(Winsock)也是以 Berkeley Socket 为范例,定义了一套 Micosoft Windows 下的网络编程接口,不仅包含了人们所熟悉的 Berkeley Socket 风格的库函数,也新增了一些针对 Windows 的扩展函数。

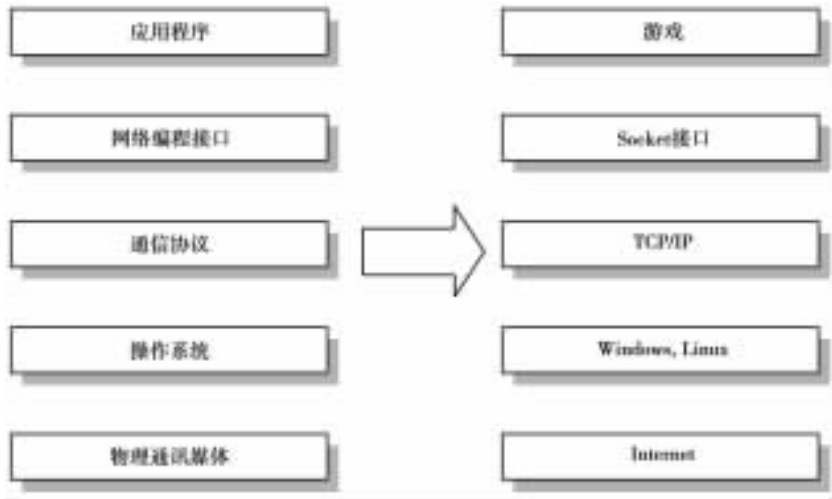


图 2.1

Socket 接口位于网络通信协议与应用程序之间,如图 2.1 所示。使用 Socket 开发网络通信程序,可以避免直接面对复杂的网络通信协议,加快开发速度,并且可以做到跨协议开发与跨平台开发。

使用 Socket 发送数据的流程(TCP)与打电话十分相似,如图 2.2 所示。

首先,都要建立一条连接,打电话是靠电话号码找到对方,而网络通信要靠 IP 地址标识主机,端口号标识进程,IP 加上端口号才能最终确定连接目标。连接建立后,就可以进行双向的数据接收与发送。最后结束时,需要关闭连接。

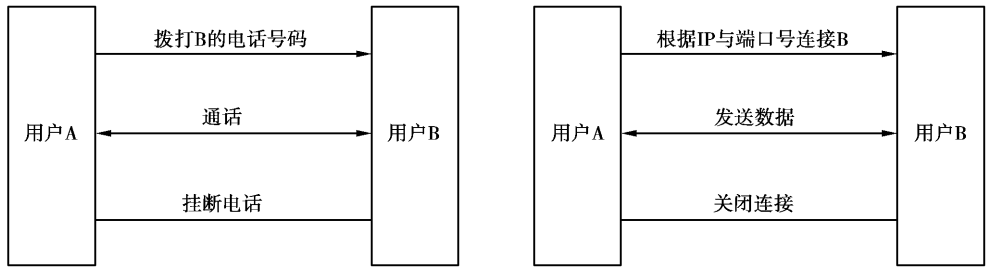


图 2.2

2.2 常用的 Socket 函数

下面以 WinSock 1.1 为例,介绍一些常用的 Socket 函数。

(1) Socket

功能:创建一个 Socket 套接口。

定义:SOCKET Socket(

```
int af,
int type,
int protocal
```

)

参数:af 表示使用的协议族;

type 表示 Socket 的类型,TCP 或 UDP;

protocal 表示使用的协议地址。

返回值:如果成功,则返回一个 Socket 描述字,否则返回 INVALID _ SOCKET。

(2) Connect

功能:尝试与远端建立一条 Socket 连接。

定义:int connect(

```
SOCKET s,
const struct sockaddr FAR * name,
int namelen
```

)

参数:s 表示 Socket 描述字;

name 表示远端的地址;

len 表示远端地址的长度。

返回值:连接的结果。0 表示连接成功,否则返回 SOCKET _ ERROR。对于非阻塞的 Socket,通常返回结果都为 SOCKET _ ERROR,并且错误代码为 WSAEWOULDBLOCK,表示连接正在进行,而不是一个真正的错误。

(3) Send

功能:尝试在某个 Socket 上发送数据。

定义:int send(
SOCKET s,
const char FAR * buf,
int len,
int flags
)
参数:s 表示 Socket 描述字;
buf 表示存放发送数据的缓冲区;
len 表示将要发送的数据长度;
flags 表示发送时使用的附加参数。
返回值:已成功发送的字节数,失败则返回 SOCKET_ERROR。
(4) Recv
功能:尝试在某个 Socket 上接收数据。
定义:int recv(
SOCKET s,
const char FAR * buf,
int len,
int flags
)
参数:s 表示 Socket 描述字;
buf 表示存放接收数据的缓冲区;
len 表示接收缓冲区的长度;
flags 表示接收时使用的附加参数。
返回值:已成功接收的字节数,失败则返回 SOCKET_ERROR。
(5) Close
功能:关闭某个 Socket 连接。
定义:int closeSocket(
SOCKET s
)
参数:s 表示 Socket 描述字。
返回值:如果关闭成功,则返回 0,否则返回 SOCKET_ERROR。
(6) Listen
功能:在某个 Socket 上进行监听。
定义:int listen(
SOCKET s,
int backlog
)
参数:s 表示 Socket 描述字;
backlog 表示缓存对列的长度。不管是 Windows 还是 Linux,对这个参数的解释都比较含

糊,一般理解为协议层已开始或已完成建立连接,但应用程序尚未 Accpet 的连接数量。一些常见的文章上经常将 backlog 设为 5,这是因为 BSD4.2 支持的最大值是 5,在当时(20 世纪 80 年代)看来这个数字已足够大。现在的系统内核都已经增大了这个值,普通应用程序不必太在意这个参数,直接设为系统最大值即可。

返回值:成功返回 0,否则返回 SOCKET_ERROR。

(7) Accept

功能:接受一条新的 Socket 连接。

定义:SOCKET accept(

```
SOCKET s,
struct sockaddr FAR * addr,
int FAR * addrlen
```

)

参数:s 表示监听中的 Socket 描述字;

addr 表示地址结构体的指针,用于存放新连接的地址;

namelen 表示地址结构体的长度。

返回值:若成功,则返回一个新的 Socket 描述字,否则返回 INVALID_SOCKET。

(8) Bind

功能:给套接口分配一个本地协议地址。

定义:int bind(

```
SOCKET s,
const struct sockaddr FAR * name,
int namelen
```

)

参数:s 表示 Socket 描述字;

name 表示地址结构体的指针;

namelen 表示地址结构体的长度。

返回值:若成功,则返回 0,否则返回 SOCKET_ERROR。

(9) Select

功能:检测 Socket 状态。

定义:int select(

```
int nfds,
fd_set FAR * readfds,
fd_set FAR * writefds,
fd_set FAR * exceptfds,
const struct timeval FAR * timeout
```

)

参数:nfds 表示 WinSock 中此参数无意义;

readfds 表示进行可读检测的 Socket 集合;

writefds 表示进行可写检测的 Socket 集合;

exceptfds 表示进行异常检测的 Socket 集合。

返回值 若成功,则返回 0,否则返回 SOCKET _ ERROR。

2.3 Server 与 Client 的常用模型

(1) Server 常用模型

Server 的处理流程如图 2.3 所示。

①初始化监听 Socket:

```
SOCKET m _ socket;
m _ socket = socket(AF _ INET,SOCK _ STREAM,IPPROTO _ TCP);
```

创建一个 Socket。其中 SOCK _ STREAM 与 IPPROTP _ TCP 表示新创建的 Socket 使用流式的 TCP 协议。

```
int port = 5555;
SOCKADDR _ IN addrLocal;
addrLocal. sin _ family = AF _ INET;
addrLocal. sin _ port = htons(port);
addrLocal. sin _ addr. s _ addr = htonl(INADDR _ ANY);
bind(m _ Socket,(SOCKADDR *)&addrLocal,sizeof(addrLocal));
```

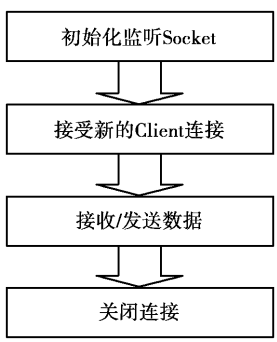


图 2.3

绑定 Socket 的本地 IP 与端口。这里,端口为 5555,地址为 INADDR _ ANY,表示绑定任意的本地地址。如果写入指定的 IP,则 bind 函数就会尝试去绑定该 IP;如果失败,则返回 SOCKET _ ERROR。

```
listen(m _ Socket,SOMAXCONN);
```

开始监听。

②接受新的 Client 连接:

首先检测是否有新的连接请求。对处于监听状态的 Socket,当有数据可读时,表示有新的连接请求。

```
fd _ set readfds;
timeval timeout;

timeout. tv _ sec = 0;
timeout. tv _ usec = 0;
FD _ ZERO(&readfds);
FD _ SET(m _ socket,&readfds);
```

```
int ret = select(FD_SETSIZE,&readfds,NULL,NULL,&timeout);
if(ret > 0 && FD_ISSET(m_socket,&readfds))
 {
    //有新的连接请求。
}
```

如果有新的连接请求,则尝试接受。

```
SOCKADDR_IN addr;
int len = sizeof(addr);
SOCKET tmp;
tmp = accept(m_Socket,(SOCKADDR *)&addr,(socklen_t *)&len);
if (tmp == INVALID_SOCKET)
 {
    //接受新连接失败
}
```

如果接受新连接成功,tmp 就是新的 Socket 描述字。

③读取数据:

首先检测是否有数据供读取。

```
fd_set readfds;
timeval timeout;

timeout.tv_sec = 0;
timeout.tv_usec = 0;
FD_ZERO(&readfds);
FD_SET(m_socket,&readfds);
int ret = select(FD_SETSIZE,&readfds,NULL,NULL,&timeout);
if(ret > 0 && FD_ISSET(m_socket,&readfds))
 {
    //有新的数据,可以读取。
}
```

然后调用 recv 函数接收数据。

```
char buf[1024];
int ret;
ret = recv(m_socket,buf,1024,0);
```

④发送数据：

首先检测当前 Socket 是否可以发送数据。

```
fd _ set writefds;
timeval timeout;

timeout. tv _ sec = 0;
timeout. tv _ usec = 0;
FD _ ZERO(&writefds);
FD _ SET(m _ Socket,&writefds);
int ret = select(FD _ SETSIZE,NULL,&writefds,NULL,&timeout);
if(ret > 0 && FD _ ISSET(m _ socket,&writefds))
 {
    //当前 Socket 可以写入数据。
}
```

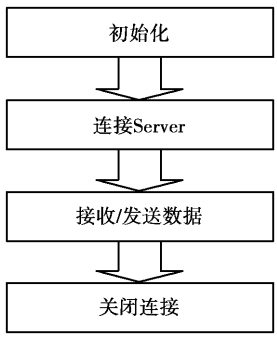


图 2.4

调用 send 函数发送数据。

```
char data[1024];
int ret;
ret = send(m _ socket,data,1024,0);
```

⑤关闭连接

```
closesocket(m _ socket);
```

(2) Client 常用模型

Client 的处理流程如图 2.4 所示。

①初始化：

```
SOCKET m _ socket;
m _ socket = socket(AF _ INET,SOCK _ STREAM,IPPROTO _ TCP);
```

创建一个 Socket,使用流式的 TCP 协议。

```
int port = 0;
SOCKADDR _ IN addrLocal;
addrLocal. sin _ family = AF _ INET;
addrLocal. sin _ port = htons(port);
addrLocal. sin _ addr. s _ addr = htonl(INADDR _ ANY);
bind(m _ Socket,(SOCKADDR *)&addrLocal,sizeof(addrLocal));
```

绑定端口为 0，表示由系统选择一个端口号。

②连接 Server：

```
SOCKADDR _ IN addrRemote;
memset(&addrRemote,0,sizeof(addrRemote));
addrRemote. sin _ family = AF _ INET;
addrRemote. sin _ port = htons(port);
addrRemote. sin _ addr. s _ addr = inet _ addr(szAddr);
connect(m _ socket,(SOCKADDR *)&addrRemote,sizeof(addrRemote)
```

其中，szAddr 和 port 分别是 Server 的地址与端口。

③读写数据：

与 Server 处理流程相同。

④关闭连接：

与 Server 处理流程相同。

2.4 编写一个 Socket 封装类

Socket 的 API 是以函数形式提供的，可以把这些 API 封装成自己的类库，这样在使用时会更方便。下面就是一个封装类的源代码，这个类可以在 Windows 与 Linux 下跨平台使用。

这里，主要封装了下列操作：

- 发送数据
- 接收数据
- 请求连接
- 接受连接
- 绑定地址
- 开始监听
- 可读判断
- 可写判断
- 关闭连接
- 设置非阻塞状态

Socket 封装类头文件 g _ netsocket. h 代码如下：

```
/ * MFNetSocket. h : interface for the CG _ NetSocket class. * /
#ifndef __ CG _ NET _ SOCKET _ H __
#define __ CG _ NET _ SOCKET _ H __
#include "g _ platform. h"
#ifdef WIN32
    / * for windows * /
    #include < winSock. h >
```

```
    #define GETERROR              WSAGetLastError()
    #define CLOSESOCKET(s)        closesocket(s)
    #define IOCTLSOCKET(s,c,a)    ioctlsocket(s,c,a)
    #define CONN_INPRROGRESS      WSAEWOULDBLOCK
    typedef int socklen_t;
#else
    /* for linux */
    #include <sys/time.h>
    #include <stddef.h>
    #include <unistd.h>
    #include <stdlib.h>
    #include <sys/wait.h>
    typedef int           BOOL;
    typedef unsigned char BYTE;
    typedef unsigned short WORD;
    typedef unsigned int  DWORD;
    #define TRUE  1
    #define FALSE 0
    /* for socket */
    #include <sys/Socket.h>
    #include <netinet/in.h>
    #include <unistd.h>
    #include <sys/ioctl.h>
    #include <netdb.h>
    #include <sys/errno.h>
    #include <arpa/inet.h>
    typedef int SOCKET;
    typedef sockaddr_in           SOCKADDR_IN;
    typedef sockaddr              SOCKADDR;
    #define INVALID_SOCKET        (-1)
    #define SOCKET_ERROR          (-1)
    #define GETERROR              errno
    #define WSAEWOULDBLOCK        EWOULDBLOCK
    #define CLOSESOCKET(s)        close(s)
    #define IOCTLSOCKET(s,c,a)    ioctl(s,c,a)
    #define CONN_INPRROGRESS      EINPROGRESS
#endif
const int PROTOCOL_UDP    =   1;
```

```
const int PROTOCOL_TCP    =   2;
class CG_NetSocket
 {
public:
    CG_NetSocket();
    virtual ~CG_NetSocket();
    bool Attach(SOCKET socket);
    bool Close();
    bool Connect(char *szAddr,int port,unsigned long ip = 0);
    bool Listen();
    bool Initialize(int protocol);
    int Recv(char *buf,int len);
    int Send(char *buf,int len);
    int RecvFrom(char *buf,int len,SOCKADDR_IN *addr,int *addrlen);
    int SendTo(char *buf,int len,SOCKADDR_IN *addr);
    bool CanWrite();
    bool CanRead();
    bool CanAccept();
    bool HasExcept();
    bool SetNonBlocking();
    bool BindAddr(char *ip,int port);
    void Reset();
    bool SetSendBufferSize(int len);
    bool SetRecvBufferSize(int len);
    bool SetReuseAddr(bool reuse);
    bool GetLocalAddr (char *addr,short *port,unsigned long *ip = NULL);
    bool GetRemoteAddr(char *addr,short *port,unsigned long *ip = NULL);
    SOCKET Accept();
private:
    bool _NetStartUp(int VersionHigh,int VersionLow);
    bool _NetCleanUp();
    SOCKET          m_socket;
    static int m_nCount;
};
#endif
```


下面写一个简单的程序,对封装类的各项功能进行测试。

测试封装类 main.cpp **代码如下：**

```
#include <stdio.h>
#include <Windows.h>
#include "g_netsocket.h"
int main()
{
    CG_NetSocket listen;
    /* initialize listen socket */
    if(!listen.Initialize(PROTOCOL_TCP))
        return 1;
    /* set to non-blocking mode */
    listen.SetNonBlocking();
    /* bind listen port */
    if(!listen.BindAddr(NULL,7788))
        return 1;
    /* begin listen */
    if(!listen.Listen())
        return 1;
    CG_NetSocket client,*server;
    if(!client.initialize(PROTOCOL_TCP))
        return 1;
    Client.Connect("localhost",7788);
    if(!listen.CanRead())
        return 1;
    SOCKET tmp = listen.Accept();
    if(!tmp)
    {
        Sys_Log("accept failed");
        return 1;
    }
    server = new CG_NetSocket;
    server->Attach(tmp);
    Sys_Log("accept ok");
    char addr[20];
    short port;
    server->GetLocalAddr(addr,&port);
    Sys_Log("local ip = %s,port = %d",addr,port);
    server->GetRemoteAddr(addr,&port);
    Sys_Log("remote ip = %s,port = %d",addr,port);
```

```
    char msg[128];
    strcpy(msg,"client send msg 1");
    /* client send msg */
    client.Send(msg,strlen(msg));
    char buf[128];
    /* server try recv msg */
    int ret = server->Recv(buf,128);
    if(ret > 0)
    {
        buf[ret] = '\0';
        Sys_Log("recv bytes = %d,msg = %s",ret,buf);
    }

    /* quit */
    getchar();
    return 0;
}
```

上述代码先初始化监听 Socket**,并在本地的** 7788 **端口进行监听。然后用另一个** Socket **进行连接。连接建立后,打印出** 2 **个** Socket **所使用的** IP **地址与端口号。最后** Client **向** Server **发送一条字符串,**Server **接收并显示。**

运行代码,屏幕显示如下:

```
WSAStartup OK                          //初始化 WinSock 成功
accept ok                              //Server 接受 Client 连接成功
local ip = 127.0.0.1,port = 7788       //Server 使用的 IP 地址与端口号
remote ip = 127.0.0.1,port = 2244      //通过 Server 取得 Client 的 IP 地址与端口号
recv bytes = 15,msg = client send msg  //Server 收到 Client 发送的数据,长度为 15B
```


2.5 Socket 高级话题

(1)非阻塞 Socket

Socket **默认工作在阻塞模式下,即某些** Socket **函数,如** recv,send **等,会一直等待,直到** I/O **操作完成。**

请看如下代码:

```
int ret;
char buf[128];
ret = recv(socket,buf,128);
printf("recv ok");
```

这段代码尝试在 Socket 上接收数据,不幸的是当没有数据可供接收时(即对方没有发送数据),recv 操作会一直等待下去。整个调用线程也会被阻塞,直到接收到数据。

由此可见,使用阻塞模式时,很难同时处理两个或两个以上的连接。执行线程经常会被阻塞,因此整个程序的执行效率非常低。解决方法有两种:一是用多线程模型,一条线程处理一个连接;二是使用 Socket 的非阻塞模式。在 Windows 下,可通过 ioctlsocket 函数将 Socket 设置为非阻塞模式。函数定义如下:

int ioctlsocket(SOCKET s,long cmd,u_long FAR * argp);

其中,s 表示设置哪一个 Socket;

cmd 表示设置哪一个命令,FIONBIO 代表设置 Socket 的工作模式;

argp 为命令的控制参数。

下面的代码将一个 Socket 设置为非阻塞模式:

```
u_long arg;
arg = 1;
ioctlsocket(socket,FIONBIO,&arg);
```

(2) Nagle 算法

当用户在网络上发送一段数据时,网络协议会对数据进行封包如下:

MAC 报头	IP 报头	TCP 报头	数据

其中,IP 报头 + TCP 报头约为 50B(长度可变)。因此,连续发送 1B 数据 10 次,在应用层统计的网络流量为 10B,而实际消耗的流量为(50 + 1) * 10B,等于 510B。可见效率是非常低的。

在[RFC896]中提出了一个简单有效的解决方法,在上一个包未被确认前,将后续的数据存入缓冲区,延迟发送。该文的作者为 John Nagle,他在 1984 年首次用这种方法来尝试解决福特汽车公司的网络拥塞问题,因此这种方法又被称为 Nagle 算法。

使用 Nagle 算法后,第一次 1B 数据被直接发送,后 9 次被累加(假设使用循环发送,后 9 次的发送执行时,第一次的确认包尚未到达)。则总计网络流量为(50 + 1)B + (50 + 9)B,即为 110B。

应用程序可通过设置 TCP_NODELAY 套接字选项,禁止使用其连接的 Nagle 算法。但一般应避免这样做,因为 Nagle 算法可提高网络的使用率,特别是当应用程序经常发送大量小数据包时。

(3) TCP 粘包问题

在 2.1.4 节中,曾有一段代码测试 Client 向 Server 发送一次数据,然后 Server 接收并

显示。

```
char msg[128];
strcpy(msg,"client send msg");
/* client send msg */
client.Send(msg,strlen(msg));
```

现在将代码修改一下,使 Client 向 Server 发送两次数据:

```
char msg[128];
strcpy(msg,"client send msg 1");
/* client send msg */
client.Send(msg,strlen(msg));

strcpy(msg,"client send msg 2");
/* client send msg again */
client.Send(msg,strlen(msg));
```

运行代码,结果如下:

```
WSAStartup OK
accept ok
local ip = 127.0.0.1,port = 7788
remote ip = 127.0.0.1,port = 2244
recv bytes = 34,msg = client send msg 1 client send msg 2
```

可以看到,Server 只调用了一次 recv 操作,就把 Client 分两次发送的数据接收到了。因为 TCP 是流式的协议,并不会去保留数据的边界,在调用 recv 时,系统总是把尽可能多的数据返回给用户。这样接收方就无法区分所收到的数据是分几次发出的。

TCP 粘包会对用户程序造成一定的麻烦。假设 Server 与 Client 的交互命令格式如下:

命令编号	命令内容

当发生粘包时,Server 接收到的数据内容如下:

命令 1 编号	命令 1 内容	命令 2 编号	命令 2 内容

如果 Server 没有做分包的处理,则第二个命令就会被舍弃掉。

引起 TCP 粘包的原因大致如下:

①Nagle 算法。当 Client 同时发送多个小数据包时,有可能被合并发送。

②接收方处理过慢,导致系统缓冲区数据多次积压。第一次的数据发送到时,接收方没有及时调用 recv,数据暂时留在系统缓冲区内。接着,第二次的发送数据到达,也被存入系统缓冲区。当接收方调用 recv 的时候,两次的发送数据就被一起取出。

在编写程序时,应当考虑 TCP 的粘包现象。如果会造成程序运行不正常,则必须在传输层将代码进行分包处理。一般可通过在命令包的头部添加命令长度来解决。

命令长度	命令编号	命令内容

(4)字节顺序

一个整型数字,在内存中存储时,一种是将低位字节存储在起始地址,称为小端字节序。另一种是将高位字节存储在起始地址,称为大端字节序。

小端字节序:

地址 x + 1	地址 x
高位字节	低位字节

大端字节序:

地址 x + 1	地址 x
低位字节	高位字节

字节顺序是由 CPU 决定的,因此除非保证 Client 与 Server 使用的是同类的 CPU,否则就有由程序处理字节顺序的问题。

对于新编写的程序,这个问题并不十分突出。因为现在不管是 PC 还是中低档的 Server,采用的都是 Intel 类 CPU,其字节顺序是一致的。

(5)应用层的缓冲区

请看如下的代码:

```
int ret;
ret = send(m_socket,buf,1024,0);
```

该段代码请求将存放在 buf 中的 1 024B 长度的数据通过 Socket 发送出去。send 函数的返回值是一个整型变量,表示实际发送成功的字节数量,但有可能比请求发送的字节数少。因此,请求发送 1 024B 的数据,函数可能返回实际发送 1 000B。应用程序必须对返回值进行校验,避免遗漏数据。代码可修改如下:

```
int ret,bytes;
bytes = 0;
while(bytes != 1024)
 {
ret = send(m_Socket,buf + bytes,1024 - bytes,0);
        if(ret == SOCKET_ERROR) break;
        bytes += ret;
}
```

上述代码只是举例说明应用程序必须对 send 函数的返回值进行校验与处理,实际编写代

码时应该有一个类来管理缓冲区,提供一些接口供调用,而不是在每个 send 的地方写上一段处理代码。

下面讨论为什么会出现上述情况。通信协议层的代码维护以下的变量:

①发送缓冲区的最大长度,可通过 setsockopt 的 SO _ SNDLOWAT 设置。

②发送缓冲区的低潮限度,可通过 setsockopt 的 SO _ SNDLOWAT 设置,缺省值一般为 2 048B。

当调用 select 检测 Socket 是否可写时,系统会检测当前系统缓冲区的剩余长度是否大于发送低潮限度。如果成立,则返回 Socket 可写。因此,当发送长度大于低潮限度的数据时,就有可能造成返回值小于发送值的情况。

例如,假设系统缓冲区还剩余 2 049B,低潮限度为 2 048B,这时用 select 检测返回 Socket 可写,于是发送一段 2 050B 的数据,实际返回值为 2 049B,表示成功发送了 2 049B,剩余 1B 无法发送。

因此,传输层代码需要维护一个自己的缓冲区。发送数据时,先将数据复制到缓冲区内,然后尝试 send,并且通过校验返回值保证数据的完整性,如图 2.5 所示。

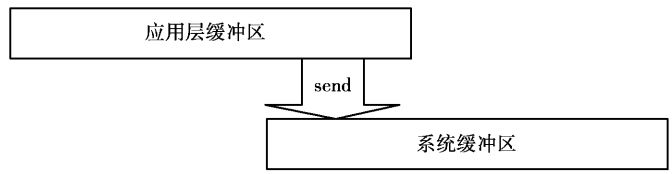
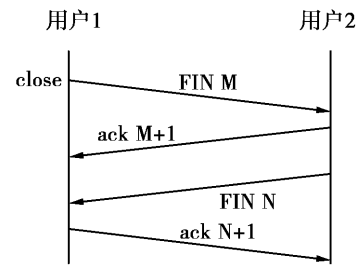


图 2.5

(6)连接进入僵死状态

TCP 的正常关闭流程如下:

当某一方调用 close 关闭 Socket 时,协议层代码会向对方 FIN,表示要求关闭网络连接。通过两次往返,4 个数据分节,连接被正常关闭。



在正常情况下,退出程序前都会编写清理代码,关闭 Socket。这样,对方就可以检测出连接已被关闭,再做相应处理。但在下列情况下,程序的清理代码并不会被执行。

①程序运行错误,导致进程异常中断。

②主机断电。

③网络设备损坏。

对于第一种情况,操作系统会关闭 Socket,因此不会造成异常。但对于第二种情况,操作系统无法执行关闭 Socket 的操作。对于第三种情况,网络的物理传输层被切断,导致 FIN 无法发送。

因此,第二与第三种情况都会造成 TCP 连接陷入僵死状态,即协议层认为连接仍然存在,但实际上已经无法发送或接收数据。TCP 一般需要 9 ~ 12min 才能检测出连接因为超时而中断。对于一般的应用程序来说,这个时间过长了。

解决方法是在传输代码中定时传输激活数据包。可定义如下的规则:

①每隔 N s 向对方发送激活数据包,表示连接正常。

②如果 3 * N s 未收到任何数据包,则认为连接已中断。

如果定义 N = 30,则僵死状态最长不会超过 90s。付出的代价是需要额外的带宽。

2.6 如何使 Socket 代码跨平台(Win32,Linux)

由于 Win32 与 Linux 的 Socket 规范都源自 Berkeley Socket,因此大部分的函数与参数都是相同的,这是代码跨平台运行的基础。

本节主要讲述如何使代码在 Win32 与 Linux 下跨平台运行。前提条件是按 WinSock 1.1 规范编写的网络代码,使用 select 模型。WinSock2.2 增加了太多平台性相关的内容,已经无法跨平台运行了。

以下列出了需要注意的几个方面:

(1)头文件

```
windows :#include  < winSock. h >
linux :#include  < sys/socket. h >
```

(2)修正一些函数的返回值与宏定义

①Win32 下关闭 Socket 使用函数 closesocket,Linux 下使用 close,可如下区分宏定义:

```
#ifdef WIN32
    #define CLOSESOCKET(s)          closeSocket(s)
#else
    #define CLOSESOCKET(s)          close(s)
#endif
```

这样,宏 CLOSESOCKET 就可以跨平台使用了。

②Win32 下对非阻塞的 Socket 调用 connect,大部分情况下返回 WSAEWOULD BLOCK,表示连接正在进行。Linux 的返回值是 EINPROGRESS。

```
#ifdef WIN32
    #define CONN _ INPRROGRESS      WSAEWOULDBLOCK
#else
    #define CONN _ INPRROGRESS      EINPROGRESS
#endif
```

其中,宏 CONN _ INPRROGRESS 用于检测连接是否正在进行。

③Linux 下用类型定义 socklen _ t 作为协议地址结构体 sockaddr 的长度单位,Win32 直接使用 int 类型。

```
#ifdef WIN32
    typedef int socklen _ t;
#endif
```

④Win32 下使用 WSAGetLastError()函数获得当前执行线程上发生的最后一个 Socket 错误,Linux 下直接使用错误变量 errno。

```
#ifdef WIN32
    #define GETERROR            WSAGetLastError()
#else
    #define GETERROR            errno
#endif
```

这里,宏 GETERROR 返回最后一个 Socket 错误。

⑤Win32 用于控制 I/O 模式的 ioctlsocket 函数,在 Linux 下为 ioctl。

```
#ifdef WIN32
    #define IOCTLSOCKET(s,c,a)    ioctlsocket(s,c,a)
#else
    #define IOCTLSOCKET(s,c,a)    ioctl(s,c,a)
#endif
```

这里,宏 IOCTLSOCKET 用于控制 Socket 的 I/O 模式。

⑥Linux 下并没有宏 INVALID _ SOCKET 与 SOCKET _ ERROR,可定义如下:

```
#ifdef WIN32


#else
    #define INVALID _ SOCKET          (- 1)
    #define SOCKET _ ERROR            (- 1)
#endif
```


(3)其他

①对于非阻塞 Socket,调用 connect 后,connect 成功的判断。

Win32:调用 select 函数时,该 Socket 处于 writefds(可写状态)中,并且不在 exceptfds 中,就表示连接成功。

Linux:Berkeley Socket 中相关的规则为:一是当连接成功时,套接字变成可写;二是当连接建立出错时,套接字变为既可读又可写。

解决方法:当发现 Socket 可写时,用 getsockopt 检测是否有错误发生。

```
#ifdef LINUX
        /* default result is true */
        bool ret = true;
        int err,len;
        len = sizeof(err);
        if(getsockopt(m_socket,SOL_SOCKET,SO_ERROR,
            (void *)&err,(socklen_t *)&len) != SOCKET_ERROR)
        {
            if(err != 0) ret = false;
        }
        else
        {
            ret = false;
        }
#endif
```

变量 ret 表示最终的连接结果。

②对已关闭的 Socket 调用 send。

Win32 :返回错误,无其他后果。

Linux :产生信号 SIGPIPE,操作系统对该信号的默认操作是关闭进程。

解决方法 :预先注册 SIGPIPE 信号。

先定义 SIGPIPE 的处理函数:

```
#ifdef LINUX
    #include <signal.h>
    void sig_pipe(int signal)
    {
        /* 处理 */
    }
#endif
```


在程序启动时,进行注册:

```
#ifdef LINUX
    signal(SIGPIPE,sig_pipe);
#endif
```

③在 Win32 下,如果监听 Socket 已被设置为非阻塞,则通过 accept 产生的新 Socket 都会继承原 Socket 的属性。Linux 下新产生的 Socket 不会继承监听 Socket 的属性,需重新设置为非阻塞模式。

④在 Linux 下,如果一个进程绑定了某个端口,那么当进程结束时,该端口仍会被继续占用几十秒,在这段时间内尝试绑定该端口都会返回失败。一个典型的例子是进程 A 用端口 B 监听用户的连接请求,由于某种原因,如用户关闭或程序错误,导致 A 被中断,接着 A 又被重新启动,这时 A 就无法继续绑定端口 B,启动失败。解决方法是利用 setsockopt 函数启用 SO_REUSEADDR 属性。

```
int opt = 1;
unsigned int len = sizeof(opt);
setsockopt(m_socket,SOL_SOCKET,SO_REUSEADDR,
        (const void *)&opt,len);
```

2.7 命令封包

编写网络程序大致有以下两种情况:

①根据已有的协议编写程序,例如编写 FTP 客户端或 HTTP 客户端等。由于协议已经制定好,因此只要 Server 与 Client 都按协议编写,即可互相通信。为了解决字节顺序等问题,这类协议一种是以纯字符形式进行命令解析,另一种是精确到位(bit)进行命令解析。

②自己制定通信协议,然后再编写代码,例如在线游戏。

本节主要讨论第二种情况。

在网络通信过程中,Server 与 Client 需要频繁地交换命令,应用程序负责写入与读取命令。命令的一般格式可定义如下:

命令 ID(long)	命令参数

用一个 long 表示命令的 ID,后面紧跟此命令的参数。参数的类型与含义是由命令 ID 决定的。

例如:

Client A 发送聊天信息

命令 ID(请求聊天)	命令参数(字符串,表示聊天内容)

Server 将聊天信息广播给所有人

命令 ID(广播信息)	命令参数(字符串,表示聊天内容)

这样,所有在线用户都可以收到 A 的聊天信息。

Server 与 Client 的交互命令有很多种,但这些命令都是由一些基本的数据类型组合而成的,大致可分为以下几种类型:

①字符型。

②短整型。

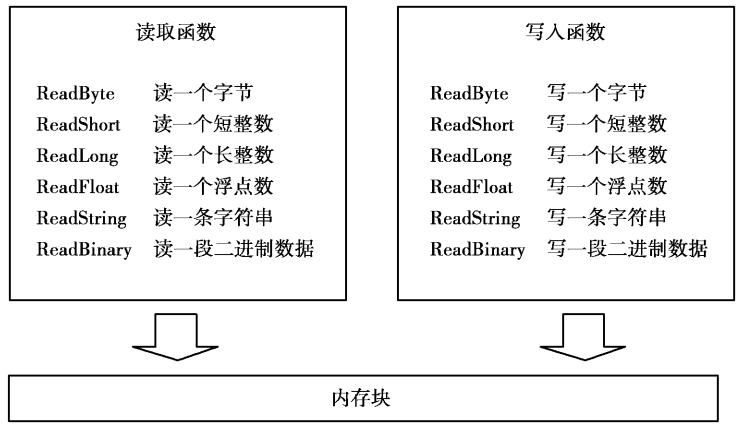
③长整型。

④字符串。

⑤浮点数。

⑥二进制数据块。

可以将写入与读取这些数据类型的代码封装成一个类,简化读取与写入流程。



命令封装类头文件 g_cmdpacket.h **代码如下:**

```
/*
g_CmdPacket.h :interface for the CG_CmdPacket class.
*/

#ifndef __CG_CMD_PACKET_H__
#define __CG_CMD_PACKET_H__
#include "g_platform.h"
/* define default packet buffer size */
const int DEFAULT_CMD_PACKET_SIZE = 1024;
/* define max packet buffer size */
const int MAX_CMD_PACKET_SIZE = 1024 * 16;
/* define some length used in packet class */
const int BYTE_SIZE = 1;
const int LONG_SIZE = 4;
const int SHORT_SIZE = 2;
const int FLOAT_SIZE = 4;
class CG_CmdPacket
{
public:
    CG_CmdPacket();
    virtual ~CG_CmdPacket();
```

```
    void BeginWrite();
    void BeginRead(char *p,int len);
    void BeginRead();
    bool ReadBinary(char **data,int *len);
    bool ReadString(char **str);
    bool ReadFloat(float *f);
    bool ReadLong(long *l);
    bool ReadShort(short *s);
    bool ReadByte(char *c);
    bool WriteBinary(char *data,int len);
    bool WriteString(char *str);
    bool WriteFloat(float f);
    bool WriteLong(long l);
    bool WriteShort(short s);
    bool WriteByte(char c);
    char *GetData();
    int  GetDataSize();
    int  GetMaxSize();
    bool SetSize(int len);
    bool CloneFrom(CG_CmdPacket *packet);
private:
    bool CopyData(char *buf,int len);
    bool WriteData(void *data,int len);
    bool ReadData(void *data,int len);
    char *m_pData;
    char *m_pReadData;
    int  m_nLen;
    int  m_nReadOffset;
    int  m_nWriteOffset;
    int  m_nMaxSize;
};

#endif
```

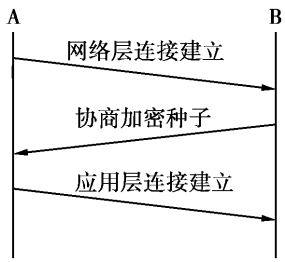
测试代码：

```
CG _ CmdPacket packet;
packet. BeginWrite();                    //准备写入数据
packet. WriteLong(1);                    //写入一个长整型
packet. WriteShort(2);                   //写入一个短整型
packet. WriteString("this is a test");   //写入一条字符串
long l;
short s;
char * str;
packet. BeginRead();
packet. ReadLong(&l);
packet. ReadShort(&s);
packet. ReadString(&str);
Sys _ Log("long = %d,short = %d,string = %s",l,s,str);
```

运行代码,屏幕显示 long = 1,short = 2,string = this is a test,表示读写成功。

2.8 数据压缩和加密

在网络数据的传输过程中,有时候不希望别人看到通信内容,因此就需要加密。同时,为了减少网络流量,可以对数据进行压缩。



一个简单的加密流程可定义如下:A 与 B 建立网络连接,B 与 A 协商一个加密种子,然后通信双方都用这个种子对数据进行加密。

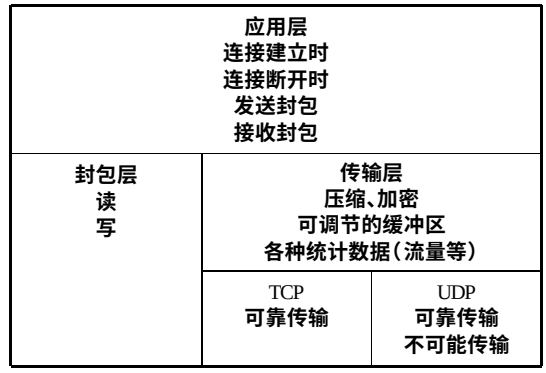
数据加密可以直接使用共享的加密类库,目前应用比较多的是 zlib,很多压缩程序都是使用它的压缩代码。需要注意的是,在数据量较小时,如只有几十个字节,压缩效果会很差,甚至根本无法压缩。

2.9 传输层整体结构

前面讨论了 Socket API 的封装、命令封包、加密与压缩等内容。这里,将把以上内容整合起来,构建一个完整的网络传输层。

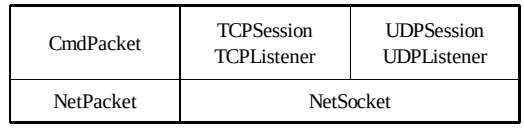
整体构想如下。

- 封包层:实现基本数据类型的读写操作。
- 传输层:提供数据压缩与加密功能,提供可调节的数据缓冲区以及各种统计数据,如网络流量,数据往返时间等。提供 TCP 与 UDP 两种传输方式,其中 TCP 是可靠传输,UDP 提供可靠与不可靠两种传输方式。
- 应用层:发送与接收封包,连接建立时与断开时的事件通知。



具体实现如下。

- NetSocket：封装 Socket API。用户不直接使用这个类。
- NetPacket：传输层使用的封包类，提供压缩与加密功能。用户不直接使用这个类。
- CmdPacket：命令封包类，提供整型、字符串等数据的写入与读取。用户可以直接使用。
- TCPSession：提供连接建立与断开的事件通知，有数据包到达的事件通知，发送与接收命令封包，各种统计数据。用户可以直接使用。
- TCPListener：用于 Server 端的 Socket 监听。用户可以直接使用。
- UDPSession：同 TCPSession。
- UDPListener：同 TCPListener。



第3章 客户端中文输入法的处理

在网络游戏中,人们希望建立一个比较完整的虚拟社会。在这里,用户之间的交流就成了关键的内容。由于种种限制,文字仍然是当前网络游戏的主要交流方式。然而,用户们使用的文字输入法各种各样。文字输入问题也是摆在开发者面前的迫切需要解决的问题。

在比较早期的游戏里,游戏制造商通常是做好一些用于输入文字的字母表,用户在字母表中选择字母后列出该字母对应的一些字,然后再选择需要的文字。后来,在一些游戏中,大多数用户使用了拼音输入法。

现在,前面的那些方法都不能满足用户的需求。

由于操作系统的发展,Windows 为我们提供了一系列功能强大的操作输入法的 Win32 API(IME API),使我们可以很方便地调用 Windows 的输入法。

DirectX 也提供一些方法可以在 DirectDraw 独占模式的窗口下显示 Windows GDI 的窗口。

直接显示 GDI 窗口的方法比较简单,这里不做讨论。下面着重介绍用 Win32 IME API 编程,在游戏中显示自己的输入法窗口。

3.1 输入法消息处理

要自己控制输入法就需要处理一些 Windows 消息,使应用程序掌握当前输入的状况。接受消息的方法属于 Windows 编程的范围,可以参考有关 Windows 编程的专著,这里不再赘述。

3.2 组字窗口消息

组字窗口是输入法句子输入的基本窗口,通常处理的消息有以下几种:

- WM _ IME _ STARTCOMPOSITION
- WM _ IME _ ENDCOMPOSITION
- WM _ IME _ COMPOSITION

3.2.1 开始组字 WM _ IME _ STARTCOMPOSITION

这个消息通知应用程序开始组字,应用程序会显示组字窗口,如图 3.1 所示。

图 3.1

如果程序需要自己显示组字窗口,就需要截下并处理这个消息。

通常,程序会需要在显示完组字窗口以后更新组字窗口的内容,应用程序可以采用下面的方法处理这个消息。

```
switch（message）
 {
……
case WM _ IME _ STARTCOMPOSITION：
    显示组字窗口；
    更新组字窗口的内容；
    break；
……
}
```

组字窗口的内容是采用 Windows 提供的 API ImmGetCompositionString 函数，取得的函数定义如下：

```
LONG ImmGetCompositionString(
    HIMC hIMC，
    DWORD dwIndex，
    LPVOID lpBuf，
    DWORD dwBufLen）；
```

其中，参数 hIMC 是输入上下文，可以用 ImmGetContext（hWnd）来取得；

参数 dwIndex 指定要取得的内容，这里用 GCS _ COMPSTR 表示需要的是组字字符串；

参数 lpBuf 就是用于保存取回来的信息的缓冲区；

参数 dwBufLen 指定了缓冲区的大小。如果这个值为 0，函数就返回取得组字字符串需要的缓冲区大小。

这样，使用下面的函数就可以取得组字字符串。

```
LONG GetComposition（HWND hWnd，char * lpszBuf，DWORD dwBufSize）
 {
    //取得输入上下文
    HIMC hIMC = :: ImmGetContext（hWnd）；
    //取得组字字符串的大小
    LONG lSize = :: ImmGetCompositionString（hIMC，GCS _ COMPSTR，0，0）；
    if（lSize == 0）return 0；
    //分配空间给用于取得组字字符串，对分配 1 个字节是为了 0 结束符
    char * pString = new char[lSize + 1]；
    //取得组字字符串
    :: ImmGetCompositionString（hIMC，GCS _ COMPSTR，pString，lSize）；
    //结束符
    pString[lSize] = '\0'；
```

```
        ……
        处理组字字符串;
        ……
        //释放字符串
        delete[] pString;
        //释放输入上下文
        ::ImmReleaseContext(hWnd,hIMC);
        return lSize;
    }
```

然后,就可以使用这个字符串了。

3.2.2 结束组字 WM_IME_ENDCOMPOSITION

这个消息通知应用程序结束组字,如果组字窗口显示,那么会关闭组字窗口。程序可以采用下面的方法处理这个消息:

```
switch(message)
 {
……
case WM_IME_STARTCOMPOSITION:
    如果组字窗口在显示状态就关闭组字窗口;
    break;
……
}
```

3.2.3 组字状态改变 WM_IME_COMPOSITION

如果有输入导致组字窗口的状态发生改变,那么应用程序会收到这个消息。程序可以采用下面的方法处理这个消息:


```
switch(message)
 {
……
case WM_IME_COMPOSITION:
    更新组字窗口的内容;
    break;
……
}
```

3.3 IME 通知消息

当 IME 窗口发生改变时，应用程序会收到 WM _ IME _ NOTIFY 消息。它的第一个参数 wParam 定义的一些命令决定了这个消息的意义。

下面列出一些经常使用的命令，并且随后会详细介绍它们：

- IMN _ OPENCANDIDATE 打开候选窗口
- IMN _ CLOSECANDIDATE 关闭候选窗口
- IMN _ CHANGECANDIDATE 候选内容发生改变
- IMN _ SETCONVERSIONMODE 变换模式（中英文模式、半角全角模式、中英文标点符号）
- IMN _ OPENSTATUSWINDOW 打开状态窗口

3.3.1 打开候选窗口 IMN _ OPENCANDIDATE

当 IME 打开候选窗口的时候会收到这个消息，如图 3.2 所示。

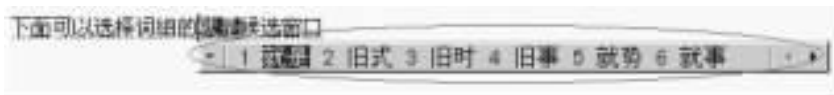


图 3.2

同样，如果程序需要自己显示候选窗口，那么就需要截下并处理这个消息。

通常，程序也会需要在显示完候选窗口以后更新候选窗口的内容。应用程序可以采用下面的方法处理这个消息。

```
switch（message）
 {
……
case WM _ IME _ NOTIFY：
      switch（message）
       {
      ……
      case IMN _ OPENCANDIDATE：
            显示候选窗口；
            更新候选窗口的内容；
            break；
      ……
      }
……
}
```

与组字窗口相同,Windows 提供了两个 API 给用户查询候选文字列表的内容。函数定义如下:

取得候选列表的大小和要取得的候选列表需要的缓冲区大小:

```
DWORD ImmGetCandidateListCount(
    HIMC hIMC,
    LPDWORD lpdwListCount);
```

函数返回要取得的候选列表需要的缓冲区大小。

取得候选列表:

```
DWORD ImmGetCandidateList(
    HIMC hIMC,
    DWORD deIndex,
    LPCANDIDATELIST lpCandList,
    DWORD dwBufLen);
```

其中,参数 deIndex 指出了需要取得的候选列表;

参数 lpCandList 是一个指向 CANDIDATELIST 结构的指针,是一个缓冲区,用于保存取回来的候选列表;

参数 dwBufLen 指出缓冲区大小。

这样,使用下面的函数就可以取出候选文字列表:

```
DWORD HandleCandidate(HWND hWnd)
 {
    unsigned long dwCount,dwSize,i;
    HIMC hIMC = ::ImmGetContext(hWnd);
    //取得候选列表的个数和要取得候选列表的缓冲区大小
    dwSize = ::ImmGetCandidateListCount(hIMC,&dwCount);
    //分配缓冲区空间
    char * pBuf = new char[dwSize];
    LPCANDIDATELIST pList = (LPCANDIDATELIST)pBuf;
    for (i=0;i<dwCount;i++)
     {
        //取得候选字符串列表
        ::ImmGetCandidateList(hIMC,i,pList,dwSize);
        ……
        处理 pList
        ……
    }
    delete[] pBuf;
    ::ImmReleaseContext(hWnd,hIMC);
    return dwCount;
}
```

其中，CANDIDATELIST 结构的定义如下：

```
typedef struct _ tagCANDIDATELIST {
DWORD dwSize；          //这个缓冲区的大小，包含这个结构本身
                        //和偏移数组以及所有的候选字符串
DWORD dwStyle；         //候选列表的类型
DWORD dwCount；         //候选字符串的个数
DWORD dwSelection；     //当前选择的候选字符串的编号
DWORD dwPageStart；     //当前页第一个候选字符串的编号
DWORD dwPageSize；      //当前页包含的候选字符串的数目
DWORD dwOffset[]；      //当前页中所有候选字符串的偏移量
} CANDIDATELIST；
```

从上面的结构可以看出，要访问当前这个候选列中的第二个字符串的值使用如下方式（假设缓冲区的指针为 pList）：

```
(char *) pList + pList -> dwOffset[pList -> dwPageStart + 1]；
```

结果如图 3.3 所示。

图 3.3

同样，也可以根据 dwSelection 的值知道当前选择的是哪个候选字符，所以要访问当前选择的那个字符串可使用如下方式：

```
(char *) pList + pList -> dwOffset[pList -> dwSelection]；
```

3.3.2 关闭候选窗口 IMN _ CLOSECANDIDATE

当 IME 关闭候选窗口时会收到这个消息。程序可以采用下面的方法处理这个消息。

```
switch (message)
 {
……
case IMN _ CLOSECANDIDATE：
     如果候选窗口有显示就关闭候选窗口；
     break；
……
}
```

3.3.3 候选窗口内容改变 IMN _ CHANGECANDIDATE

候选窗口的内容或者状态有发生改变时应用程序会收到这个消息。同样，程序需要更新

候选窗口的内容,可以采用下面的方法处理:

```
switch (message)
 {
……
case IMN _ CHANGECANDIDATE:
      更新候选窗口的内容;
      break;
……
}
```

3.3.4 输入模式改变 IMN _ SETCONVERSIONMODE

当输入上下文的输入模式发生改变时应用程序会收到这个消息。这时需要检查输入模式并且更新输入窗口的提示信息。

Windows 提供一个 API 来取得 IME 的输入模式,函数定义如下:

```
BOOL ImmGetConversionStatus(
      HIMC hIMC,
      LPDWORD lpfdwConversion,
      LPDWORD lpfdwSentence);
```

其中,参数 lpfdwConversion 可以取得输入模式,输入模式是按位定义信息的,常用的信息有以下几种,如表 3.1 所示:

表 3.1

定义	有定义值/没有定义值
0x00000001	中文输入/英文输入
0x00000008	全角字符/半角字符
0x00000080	打开软键盘/没有打开软键盘
0x00000400	中文标点/英文标点

取得这些信息后,就可以用于显示。

3.3.5 状态窗口消息

当应用程序创建状态窗口时会收到下面这些消息中的一种。

- IMN _ OPENSTATUSWINDOW
- IMN _ CLOSESTATUSWINDOW
- IMN _ SETOPENSTATUS

如果要自己显示状态窗口,那么就需要截断并处理这个消息。

可以通过 Windows 提供的一个 API 来获得当前的 IME 状态窗口是否打开。函数定义如下:

```
BOOL WINAPI ImmGetOpenStatus(HIMC hIMC);
```

3.4 获得输入的内容

至此,已经了解了如何处理组字窗口以及候选窗口。下面需要正确获取并处理输入的内容。

事实上,由 IME 输入的内容还是通过消息 WM_CHAR 传递给应用程序的,所以只需要处理 WM_CHAR 就可以取得输入的内容。但 WM_CHAR 只是如实反映了键盘输入的状况,并没有针对输入的内容进行任何处理。所以有时需要对一些特殊的情况进行处理。例如,按下退格键希望把前面一个字符删除,按下向左的方向键希望输入的光标向左移动一个字符,可以采用下面的方法处理 WM_CHAR 消息。

```
case WM_CHAR:
    if (wParam == 8) //退格键
     {
        对退格键的处理
    } else
     {
        对输入字符的处理
    }
    break;
```

由于方向键没有产生 WM_CHAR 消息,所以需对 WM_KEYDOWN 或 WM_KEYUP 消息进行处理。

```
case WM_CHAR:
    if (wParam == VK_LEFT) //向左方向键
     {
    }
    ……
```

3.5 获得当前输入法名字

IME 提供了一个 API 来取得当前输入法的描述,定义如下:

```
UINT ImmGetDescription(
    HKL hKL,
    LPTSTR lpszDescription,
    UINT uBufLen);
```

其中,参数 hKL 是指向 keyboard layout 的一个句柄,可以用如下的 API 来取得:

```
HKL GetKeyboardLayout( DWORD dwThreadId );
```

只要 dwThreadId 传入 0 就可以取得当前使用的 keyboard layout;

参数 lpszDescription 是用来保存输入法描述的一个缓冲区的指针,传回的是一个以 0 结尾的字符串;

参数 uBufLen 指出了 lpszDescription 允许存放的内容的大小,如果这个参数的值为 0,那么返回要取得描述需要的缓冲区的大小。

下面的一段代码可以取得输入法的内容:

```
HKL hKL = :: GetKeyboardLayout(0);
int iSize = :: ImmGetDescription(hKL,NULL,0);
if (iSize == 0) return;
char * pNameBuf = new char[iSize + 1];
:: ImmGetDescription(hKL,pNameBuf,iSize);
处理输入法的名字
……
```

通常,在以下 3 个地方会需要重新设置输入法,然后获取输入法的名字并显示给用户。

- 创建应用程序窗口的时候,即在收到 WM _ CREATE 消息的时候。
- 在重新设置窗口焦点的时候,即在收到 WM _ SETFOCUS 消息的时候。
- 在用户改变输入法的时候,即在收到 WM _ INPUTLANGCHANGE 消息的时候。

3.6 关闭当前的输入法窗口

至此,可以在游戏中自己定制输入法的显示了,但 Windows 默认的输入法窗口仍然存在。也就是说,能同时看到 2 个输入法窗口:一个是 Windows 默认的,另一个是自己定制的。当然,这不是我们希望看到的结果。那么,如何关闭 Windows 默认的输入法窗口呢?

Windows 提供一个 API 用于绑定一个指定的窗口到一个指定的输入上下文,函数定义如下:

```
HIMC ImmAssociateContext(
    HWND hWnd,
    HIMC hIMC);
```

其中,参数 hWnd 为指定的窗口句柄;hIMC 为指定的输入上下文。

下面的程序段可以绑定一个新的输入上下文,并且隐藏掉原来的输入法窗口。这段代码需要在窗口创建的时候就执行,即在 WM_CREATE 消息中执行。

```
HIMC        hIMC;
//创建新的输入上下文
hIMC = ImmCreateContext();
//绑定输入上下文给本窗口
hIMC = ImmAssociateContext(hWnd,hIMC);
//保存旧的输入上下文
SetWindowLong(hWnd,0,(LONG)hIMC);
//创建一个假的 IME 窗口
CreateWindow("Ime","",WS_POPUP | WS_DISABLED,0,0,0,0,hWnd,NULL,
g_hInstance,NULL);
```

程序结束后要恢复旧的输入上下文,即在 WM_DESTROY 消息中执行。

```
HIMC        hIMC;
//取出保存的旧的输入上下文
hIMC = (HIMC)GetWindowLong(hWnd,0);
//绑定默认的输入上下文给窗口
hIMC = ImmAssociateContext(hWnd,hIMC);
//销毁用户创建的输入上下文
ImmDestroyContext(hIMC);
```

通过以上的处理,就可以在游戏中使用自己的输入法了。

第4章 客户端界面系统之格式化文字显示

一个具有友好用户界面的软件能让用户更好地使用,要做到友好的用户界面,文字的表现是一个关键,特别体现在帮助系统上。在 Windows GUI 下实现起来非常简单,很多时候只需用户用一些软件进行排版即可,完全不需要了解显示的部分。但是在游戏中还不能这样,游戏制作者需要写一套完整的能显示指定格式的文字。下面来看看如何实现一套格式化的文字,在这个格式的基础上用户可以实现复杂的文字显示,甚至超级链接和图文混排。

4.1 自定义格式化文本 CFT

为了更好的可扩充和更好的用户自定义的功能,在底层就只能实现一种不依赖和不针对任何功能的格式。XML 文件的格式完全可以满足这种需求,所以这里选择一种类似于 XML 但功能简单的格式,我们把这种格式称为“自定义格式化文本”,可以用 CFT 来表示。

这里定义的 CFT 由下列 3 部分构成:

- 元素
- 标记
- 属性

(1)元素

元素是 CFT 的主要组成部分,是开始标记到结束标记之间的所有内容。例如:

```
<星期>1 星期有 7 天</星期>
```

就是一个元素。其中,“<星期>”为开始标记,“</星期>”为结束标记,“1 星期有 7 天”为元素内容。要注意的是所有的开始标记和结束标记必须是成对出现的,即使是一个没有内容的元素也必须包含完整的标记,如:

```
<星期></星期>
```

元素是嵌套定义的,也就是说元素可以包含子元素。例如:

```
<星期>
    <星期一>开始上班</星期一>
    <星期二>上班第二天</星期二>
</星期>
```

“<星期一>开始上班</星期一>”和“<星期二>上班第二天</星期二>”是元素“<星期>……</星期>”的子元素。

每个 CFT 文档都会自动有一个根元素,根元素不需要开始和结束标记,所有的元素都是根元素的子元素。

(2)标记

标记是用来表示元素的,可以认为标记是元素的名字。标记分开始标记和结束标记,上例中“<星期>”为开始标记,“</星期>”为结束标记。通常在开始标记的内容前面加符号“/”表示结束标记。符号“<”和“>”是标记的分隔符,事实上“<星期>”是表示一个名字叫“星期”的标记。

要注意的是,由于标记和属性是用空格分开的,所以标记名字中如果包含有空格、“<”或者“>”符号就必须用双引号括起来,例如:

```
<My Name = Janhail>
```

这个标记的名字是“ My ”;“ Name ”是属性,“ Janhail ”是“ Name ”的值。

< " My Name" = Janhail >

这个标记的名字则是“ My Name ”,这个表示是错误的,因为标记不能赋值,这里只是为了说明标记名字可以用双引号括起来。

上面所说的空格事实上即是分隔符号,跳格键(Tab)、空格和回车都可以作为分隔符。

(3)属性

属性是元素的性质,写在标记内。例如:

< 星期 时间 = 上半年 > …… < /星期 >

属性名和属性值是成对出现的。上面就定义了一个名字为“ 时间 ”的属性,其值是“ 上半年 ”。

一个元素可以同时有多个属性,排列在标记内,并用空格分开。例如:

< 星期 开始时间 = 2 月 结束时间 = 6 月 > …… < /星期 >

这里,“ 开始时间 ”和“ 结束时间 ”都是元素的属性,其值分别为“ 2 月 ”和“ 6 月 ”。

和标记的名字一样,对于包含有空格的属性的名字或者值都必须用双引号括起来。

由于 CFT 只是定义了一些语法规则,并没有对功能或者关键字进行任何限制,所以原则上说它是很灵活的,并且可以表示任何东西。

如下面的课程表:

周 1	周 2	周 3	周 4	周 5
语文	数学	英语	体育	政治
计算机	历史	物理	美术	休息

可以表示为:

```
< 每一周 >
    < 周 1  上午 = 语文  下午 = 计算机 > < /周 1 >
    < 周 2  上午 = 数学  下午 = 历史 > < /周 2 >
    < 周 3  上午 = 英语  下午 = 物理 > < /周 3 >
    < 周 4  上午 = 体育  下午 = 美术 > < /周 4 >
    < 周 5  上午 = 政治  下午 = 休息 > < /周 5 >
< /每一周 >
```

同样,下面的一段文字:

红色*和*蓝色的字

可以表示成 CFT 的格式:

```
< 字体 颜色 = 红色 >
    红色
    < 字体 样式 = 斜体 >
        和
    < /字体 >
    < 字体 颜色 = 蓝色 >
```

```
            蓝色
        </字体>
        <字体 样式=粗体>
            的字
        </字体>
    </字体>
```

由此可见,CFT 非常灵活,并且便于编辑。

由于 CFT 格式的灵活性和方便编辑修改,这里就选择它作为后面的文字格式描述语言。

4.2 自定义格式化文本 CFT 的读取

为了满足多方面需要,这里需要一个灵活的 CFT 分析程序。

下面定义了一个类,它足以满足大部分用户的需求。

```
//CFTAnalyser.h :interface for the CCFTAnalyser class.
//By. Janhail Luo
//////////////////////////////////////////////////////////////////////

#ifndef _CFTANALYSER_H_
#define _CFTANALYSER_H_
#include <map>
#include <list>
#include <stack>
#include <vector>
#include <string.h>
using namespace std;
class CCFTAnalyser
{
public:
    struct SItem
    {
        int                         iType;      //节点类型,0 元素,1 元素内容
        CString                     strName;    //标记名字
        map<CString,CString>        listParm;   //属性表
        vector<SItem*>              listNode;   //子元素列表
        void*                       pData;      //用户自定义数据
    };
public:
    //构造函数
```

```
    CCFTAnalyser();
    //析构函数
    virtual ~CCFTAnalyser();
public:
    //初始化
    //szString 是一个以 0 结尾的字符串,里面保存的是需要分析的字符串
    //szName 是一个以 0 结尾的字符串,里面保存这次分析的名字
    //返回 true 表示分析正确,返回 false 表示分析错误
    bool        Init(const char * szString,const char * szName);
    //清除
    void        Clear();
    //访问所有的字节点
    //iType 传入一个用户定义的参数
    //pParam 也是传入一个用户自定义的指针
    void        Visitor(int iType,void * pParam);
protected:
    //分析字符串 szStr
    //如果分析成功完成就返回 true
    //否则返回 false
    bool        Analyse(const char * szStr);
    //当分析程序碰到一个标记时调用这个函数
    //iState  当前分析状态
    //szStart  指向标记开始部分的指针
    //szEnd  指向标记结束部分的指针
    bool        DoTag(int iState,char * szStart,char * szEnd);
    //当分析程序碰到一个属性时调用这个函数
    //iState  当前分析状态
    //szStart  指向标记开始部分的指针
    //szEnd  指向标记结束部分的指针
    bool        DoItem(int iState,char * szStart,char * szEnd);
     //访问某一个节点
     //iType  是用户自定义的参数
     //iDepth  是这个节点在节点树中的层次,根节点的层次为 0
     //pItem  是当前访问的节点
     //pParam  是用户自定义的一个指针
     void        VisitItem(int iType,int iDepth,
                 CCFTAnalyser :: SItem * pItem,void * pParam);
protected:
    //当创建一个新节点时就会调用这个函数
```

```
        //pItem 当前创建的节点的指针
        virtual void OnCreateItem(CCFTAnalyser :: SItem * pItem);
        //当销毁一个节点的时候就会调用这个函数
        //pItem 当前销毁的节点的指针
        virtual void OnDestroyItem(CCFTAnalyser :: SItem * pItem);
        //访问一个节点前,这是一个虚函数,可以由用户自定义
        //iType  是用户自定义的参数
        //iDepth  是这个节点在节点树中的层次,根节点的层次为 0
        //pItem  是当前访问的节点
        //pParam  是用户自定义的一个指针
        virtual void OnVisitBefore(int iType,int iDepth,
                CCFTAnalyser :: SItem * pItem,void * pParam);
        //访问一个节点,这是一个虚函数,可以由用户自定义
        //iType  是用户自定义的参数
        //iDepth  是这个节点在节点树中的层次,根节点的层次为 0
        //pItem  是当前访问的节点
        //pParam  是用户自定义的一个指针
        virtual void OnVisit(int iType,int iDepth,
                CCFTAnalyser :: SItem * pItem,void * pParam);
        //访问一个节点后,这是一个虚函数,可以由用户自定义
        //iType  是用户自定义的参数
        //iDepth  是这个节点在节点树中的层次,根节点的层次为 0
        //pItem  是当前访问的节点
        //pParam  是用户自定义的一个指针
        virtual void OnVisitAfter(int iType,int iDepth,
                CCFTAnalyser :: SItem * pItem,void * pParam);
    protected:
        CString             m_Name;
        SItem               m_Root;      //根节点
        stack < SItem * >   m_Stack;     //m_Stack 是一个堆栈,用在分析和遍历节点树
                                           的代码中
    };
    #endif //_CFTANALYSER_H_
```

CET 读取类的实现请参考光盘中的内容。

4.3 游戏中 CFT 的应用

如果在游戏中需要能灵活地表示字体的颜色、大小、字体和样式等属性，那么可以在 CFT 中把它们都做成标记或者属性。我们做如下定义：

用标记 < font > 表示字体的属性包含字体（font）、颜色（color）和字体大小（size）；用标记 < i > 表示斜体；用 < b > 表示粗体。

例如：

< font font = "宋体" color = red size = 16 > 内容 < /font >

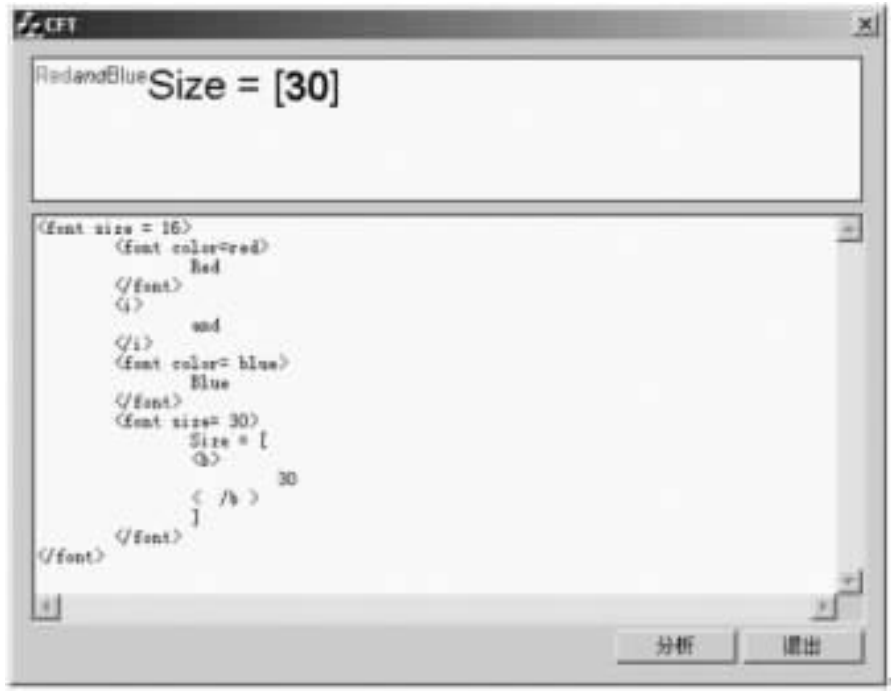


图 4.1

表示“ 内容 ”需要用宋体红色且大小为 16 的字来显示。

< i > 内容 < /i >

表示“ 内容 ”显示为斜体。

< b > 内容 < /b >

表示“ 内容 ”显示为粗体。

如图 4.1 所示。

要做到这样，只要在需要改变字体的时候创建一个新的字体，并设置成当前字体，同时把旧的字体保存到一个堆栈中。这样，以后的内容显示就按照新的字体显示。在字体结束标记的时候，只要删掉当前字体，同时把字体堆栈栈顶的字体出栈，并设置成当前字体即可。

这样即可实现要求的显示。下面列出这种实现的类的定义：

```
class CMyText ;public CCFTAnalyser
 {
    struct   SItemInfo
     {
        CFont *          pOldFont ;
        CFont *          pFont ;
        COLORREF         colorText ;
        int              nFontSize ;
        BOOL             bItalic ;
        BOOL             bUnderline ;
        int              nWeight ;
    } ;
public :
    CMyText() ;
    virtual  ~ CMyText() ;
private :
    stack < SItemInfo >  fontStack ;
private :
    virtual void OnVisitBefore(int iType,int iDepth,
            CCFTAnalyser :: SItem *  pItem,void *  pParam) ;
    virtual void OnVisit(int iType,int iDepth,
            CCFTAnalyser :: SItem *  pItem,void *  pParam) ;
    virtual void OnVisitAfter(int iType,int iDepth,
            CCFTAnalyser :: SItem *  pItem,void *  pParam) ;
} ;
```

在函数 OnVisitBefore 中如果碰到一个字体的开始标记就创建一个字体。

```
void CMyText :: OnVisitBefore(int iType,int iDepth,
             CCFTAnalyser :: SItem *  pItem,void *  pParam)
 {
    CPaintDC *  pDc  =  (CPaintDC * )pParam ;
    switch (iType)
     {
    case 1 :
        if (pItem - > strName == " font"
          || pItem - > strName == " i"
          || pItem - > strName == " b" )
         {
          SItemInfo item ;
```

```
item. pFont = new CFont;
if (fontStack. size() == 0)
 {
    item. colorText = 0;
    item. nFontSize = 12;
    item. nWeight = FW_NORMAL;
    item. bItalic = FALSE;
    item. bUnderline = FALSE;
} else
 {
    item. colorText = fontStack. top(). colorText;
    item. nFontSize = fontStack. top(). nFontSize;
    item. nWeight = fontStack. top(). nWeight;
    item. bItalic = fontStack. top(). bItalic;
    item. bUnderline = fontStack. top(). bUnderline;
}
if (pItem -> strName == "i")
 {
    item. bItalic = TRUE;
} else if (pItem -> strName == "b")
 {
    item. nWeight = FW_BOLD;
} else
 {
    long n;
    char * stop;
    if (pItem -> listParm. find("color")
          != pItem -> listParm. end())
     {
        //设置字体颜色
        n = strtol(color,&stop,16);
        item. colorText = RGB(
              (n&0x0FF0000)>>16,
              (n&0x0FF00)>>8,
              n&0x0FF);
    }
    if (pItem -> listParm. find("size")
          != pItem -> listParm. end())
     {
```

```
                    n = strtol(pItem->listParm["size"],&stop,10);
                    item.nFontSize = n;
                }
            }
            //创建新字体
            ……

            item.pOldFont = pDc->SelectObject(item.pFont);
            //保存旧字体到堆栈中
            fontStack.push(item);
        }
        break;
    }
    CCFTAnalyser::OnVisitBefore(iType,iDepth,pItem,pParam);
}
```

同样,在碰到字体的结束标记时,需要删除当前字体并恢复上一次的字体,即堆栈中的栈顶元素。

```
void CMyText::OnVisitAfter(int iType,int iDepth,
                CCFTAnalyser::SItem * pItem,void * pParam)
{
    CPaintDC * pDc = (CPaintDC *)pParam;
    switch (iType)
    {
    case 1:
        if (pItem->strName == "font"
            || pItem->strName == "i"
            || pItem->strName == "b" )
        {
            pDc->SelectObject(fontStack.top().pOldFont);
            fontStack.top().pFont->DeleteObject();
            delete fontStack.top().pFont;
            fontStack.pop();
        }
        break;
    }
    CCFTAnalyser::OnVisitAfter(iType,iDepth,pItem,pParam);
}
```

当然，在这个基础上可以做更多的事情，如定义带下画线的字符、定义字符的背景颜色等，甚至可以定义图片、表格。

光盘中包含有关于 CFT 的完整代码和例子。

第5章 客户端数据资源的管理

在一个游戏中,资源管理是一件非常重要的事情。良好的资源管理可以为游戏节省内存资源和载入资源时间,甚至是很多的计算量。

5.1 游戏中可重复使用的资源

在一个在线游戏中,同一个屏幕内经常会出现大量的单位。下面是一个单位的数据结构的定义。

```
//单位的数据
struct SUnit
 {
    //普通属性
    long lID;                  //单位的惟一 ID
    char * szName;             //单位的名字
    int iPosX;                 //单位当前的位置 X
    int iPosY;                 //单位当前的位置 Y
    …
    //图像属性
    int iCurDir;               //当前方向
    int iCurPose;              //当前动画
    int iImgWidth;             //当前图像宽度
    int iImgHeight;            //当前图像高度
    int iCurFrame;             //当前帧
    int iFrameCnt;             //当前动画总共多少帧
    unsigned char * pImg;      //当前帧的数据指针
    SDirAnis ani[8];           //动画数据(8 个方向)
 };
```

其中,SDirAnis 保存了单个方向的动画,由上面的结构可以看出这个单位支持 8 个方向。SDirAnis 的定义如下:

```
//单位的单个方向动画
struct SDirAnis
 {
    int iAniCnt;     //动画的个数
    SAni * pAnis;    //动画数组
 };
```

其中，SAni 保存了一个动画的信息，pAnis 是一个动画数组的指针，具体的大小由 iAniCnt 得到。SAni 的定义如下：

```
//动画的结构
struct SAni
 {
    int iFrameCnt;    //动画的帧数
    int iWidth;       //图像的宽度
    int iHeight;      //图像的高度
    unsigned char * pData;  //图像的数据
};
```

由上面的结构可以看出，每个单位都会独享一段数据。如果屏幕上同时有 50 个单位，就需要 50 个这样的数据结构。但是，有一些数据，特别是单位的图像数据，如果是相同的单位，那么其动画结构里的数据就相同，也就是成员变量 ani 里的数据是相同的。在这种情况下，可以把这些数据提取出来，放在一块独立的空间中，以后所有使用到相同数据的单位直接指向这块数据即可，而单位里只要保存自己单位相关的数据就可以了。上面的结构就改进成为下面这种形式：

```
typedef SDirAnis * LPSDirAnis;
//单位的数据
struct SUnit
 {
    //普通属性
    long lID;              //单位的惟一 ID
    char * szName;         //单位的名字
    int iPosX;             //单位当前的位置 X
    int iPosY;             //单位当前的位置 Y
    …
    //图像属性
    int iCurDir;           //当前方向
    int iCurPose;          //当前动画
    int iImgWidth;         //当前图像宽度
    int iImgHeight;        //当前图像高度
    int iCurFrame;         //当前帧
    int iFrameCnt;         //当前动画总共多少帧
    unsigned char * pImg;  //当前帧的数据指针
    LPSDirAnis ani[8];     //动画数据指针(8 个方向)
};
```

ani 中保存的不再是实质的数据,而是动画数据的指针。

原来单位数据的关系如图 5.1 所示。

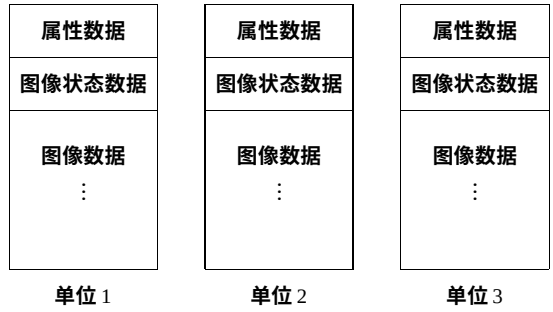


图 5.1

这里,3 个外形相同的单元使用了 3 块内存保存图像数据。

改进过后的单位数据与图像数据的关系如图 5.2 所示。

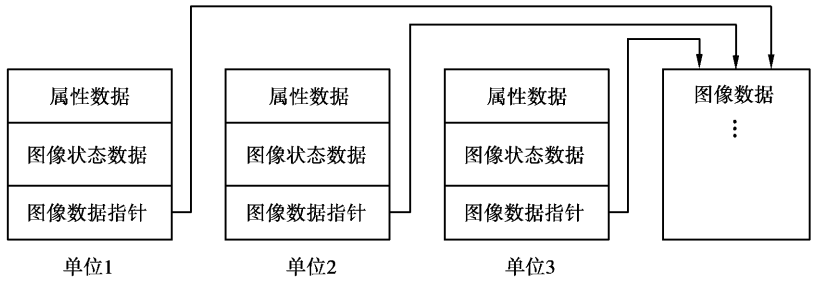


图 5.2

而在这里,3 个外形相同的单元使用了同一块内存保存图像数据。

很显然,改进后的方式可以节省更多的内存,同时可以减少程序读取文件和分配、释放内存的次数。

5.2 用统一的方法管理可多重使用的资源

大量重复的资源在游戏中可能会有很多,如 2D 游戏的人物的动画、攻击效果的动画和地表单位的图等等。类似的这种资源都有一个相同点,就是文件里的数据是可以独立出来存放的。

所以,可以定义出一种公共的接口来访问和管理这些资源。为了方便用户使用这个管理程序,对这个接口的定义,有几个基本的要求:

①简单的访问资源方式。

②快速的访问资源方式。

③安全的使用资源。

下面,针对这几个目标讨论实现的方法。

5.2.1 简单的访问资源方式

要使用户以最简单的方式访问资源的方法就是对用户隐藏实现的细节,也就是说资源管理程序对用户是透明的,用户只要按照一个简单的方法就可以访问到所需的资源,而不必知道这个资源是从文件中读进来还是从缓冲区中取得的。

下面定义一个资源对象类 CResObject,该类提供用户访问资源的方法:

```
class CResObject
 {
    friend class CResMgr;
public:
    //构造函数
    CResObject();
    //析构函数
    virtual ~CResObject();
    //创建一个资源对象
    //pResName 为需要创建的对象的名字
    void CreateObject(const char* pResName);
    //销毁当前的资源对象
    void DestroyObject();
    //取得数据指针
    void* GetData();
    //取得对象名字
    const char* GetName();
protected:
    //创建一个对象的时候,会调用这个函数
    //pResName 为需要创建的对象的名字
    //返回创建的对象的指针
    virtual void* OnCreateObject(const char* pResName) = 0;
    //当销毁当前对象的时候会调用这个函数
    virtual void OnDestroyObject();
protected:
    void* m_pData;                  //保存对象数据的指针
    const char* m_pName;            //当前对象的名字
    static CResMgr* m_pResMgr;      //对象管理实例的指针
};
```

可以看到成员函数 CreateObject(const char * pResName)用于创建一个资源,它用一个字符串表示一个资源。用户无须知道其他的信息,只要调用这个函数就会把当前对象的内容填充为所需的资源。用户只需用成员函数 void * GetData()就可以访问资源数据。以上就是用户需要用到的接口,资源的管理对用户是透明的。为了做到资源的管理,创建一个资源管理类,定义如下:

```
class CResMgr
 {
    friend class CResObject;
protected:
    //构造函数
    CResMgr();
    //析构函数
    ~CResMgr();
protected:
    //注册一个对象到使用资源的列表中
    //pResObj  是需要注册的对象的指针
    //pName  是需要注册的对象的名字
    //如果当前对象已经注册,就把对象加入到使用资源的列表中
    //同时修改对象的数据,使名字和数据指针指向当前资源
    //如果当前对象没有注册,就创建一个对象
    //同时把这个对象加入到使用资源的列表中
    void RegisterRes(CResObject * pResObj,const char * pName);
    //把已经注册的对象注销
    void UnregisterRes(CResObject * pResObj);
    //检查是否已经没有资源对象
    //如果没有资源对象就返回 true
    //否则返回 false
    bool Empty();
protected:
    struct SItem
     {
        void * pData;                          //数据指针
          list < CResObject * > objList;       //对象列表
      };
   protected:
      map < string,SItem >    m_Item;          //资源列表
  };
```

由上面的定义可以看出，构造函数和析构函数都是 protected 的成员函数，保证了这个类不能被用户实例化，只能被 friend 类 CResObject 实例化。

由于资源的创建并不是用户每次实例化都需要的（只有在缓冲区中没有该资源的时候才会创建），所以这里使用类中的一个虚函数 CResObject 来创建资源。需要创建资源的时候会调用这个虚函数。同样，销毁资源的方法也是如此。

由于成员函数 OnCreateObject 是一个纯虚函数，所有资源子类必须继承这个函数用来创建资源。

首先调用 CResObject 的成员函数 CreateObject 来创建一个资源对象。CreateObject 的实现如下：

```
void CResObject :: CreateObject(const char * pResName)
 {
    //如果资源管理类没有实例，就创建一个实例
    if (m _ pResMgr == NULL)
     {
        m _ pResMgr = new CResMgr();
    }
    //如果当前对象中包含有资源，就销毁这个对象的资源
    if (m _ pData != NULL)
     {
        this -> DestroyObject();
    }
    //注册这个资源
    m _ pResMgr -> RegisterRes(this,pResName);
}
```

可以看到，资源管理类的实例也是在这个函数中创建的，用户无需从其他地方创建管理类的实例。而资源的数据则是由管理类的成员函数 RegisterRes 填充的，其实现如下：

```
void CResMgr :: RegisterRes (CResObject * pResObj,
                             const char * pName)
 {
    //资源的名字
    string name = pName;
    //在列表中查找资源
    map < string,SItem > :: iterator it = m _ Item. find(name);
    //如果找不到资源
    if (it == m _ Item. end())
     {
        SItem item;
        //创建一个资源
        item. pData = pResObj -> OnCreateObject(pName);
```

```
        //把创建的资源加入到资源列表中
        m_Item[name] = item;
        it = m_Item.find(name);
    }
    //给对象的数据指针和名字指针赋值
    pResObj->m_pData = (*it).second.pData;
    pResObj->m_pName = (*it).first.c_str();
    //把对象加入到使用资源的列表中
    (*it).second.objList.push_back(pResObj);
}
```

其实现过程是,先去管理类的资源列表(m_Item)中查找当前需要的资源,如果找不到就调用 CResObject 的虚成员函数 OnCreateObject 创建资源,同时把资源加入到资源列表中。最后给 CResObject 对象的成员变量赋值,使它们指向资源。如果用户找到了已有的资源,就会把自己添加到成员列表中去,事实上就是使这个资源多了一个引用。那么我们为什么要把对象的指针加到列表中去,而不是单纯地添加引用计数的值呢?事实上,添加指针有利于资源自身发生变化时候的通知。如果这个资源发生变化,就可以通过列表中的指针通知所有的引用对象。

下面来看用户销毁一个资源对象的过程。首先调用 CResObject 的 DestroyObject 来销毁当前对象,DestroyObject 的实现如下:

```
void CResObject::DestroyObject()
{
    if (m_pResMgr != NULL)
    {
        if (m_pData)
        {
            //注销这个资源
            m_pResMgr->UnregisterRes(this);

            //如果资源管理类中没有保存任何资源
            if (m_pResMgr->Empty())
            {
                //删掉资源管理类
                delete m_pResMgr;
                m_pResMgr = 0;
            }
            m_pData = 0;
            m_pName = 0;
        }
    }
}
```

删除资源的过程是在 CResMgr 的 UnregisterRes 中实现的。如果删除资源后资源管理类中没有任何资源,那么就可以把资源管理类删除。

下面是 CResMgr 的 UnregisterRes 的实现:

```
void CResMgr :: UnregisterRes(CResObject * pResObj)
 {
    //资源的名字
    string name;
    if (pResObj -> m_pName == NULL) name = "";
    else name = pResObj -> m_pName;
    //取得使用资源的对象列表
    SItem * pItem = &(m_Item[name]);
    //查找
      list < CResObject * > :: iterator it;
      it = find(pItem -> objList. begin(),
                pItem -> objList. end(),pResObj);
      //把查找到的对象删除
      pItem -> objList. erase(it);
      //如果已经没有对象使用这个资源
      if (pItem -> objList. empty())
       {
          //销毁这个资源
          pResObj -> OnDestroyObject();
       }
  }
```

销毁对象的时候,管理类找出对象的指针,并且将其删除即可。如果没有任何对象引用这个资源,那么就可以把这个资源删除。

5.2.2 快速的访问资源方式

由于每个对象都保存了资源的指针,所以访问的速度非常快。函数 GetData 就是用于返回资源的指针,其实现如下:

```
void * CResObject :: GetData()
 {
    return m_pData;
 }
```

可以看到这个函数返回的是一个 void 的指针,没有任何的类型信息。所以通常都会在用

户继承的子类中定义一个取得数据的成员函数,返回具体的类型,如:

```
SImageStruct * CImageObject :: Data()
{
    return (SImageStruct *) GetData();
}
```

5.2.3 安全的使用资源

由于资源的管理对用户是透明的,用户不必对资源指针赋值,资源删除的时候,用户也不必去逐一清空指针,管理类自己会找到资源对象并将其清空。所以,对用户来说,它比使用指针更安全。

使用这种资源管理,可以很有效地降低程序在读取文件、分配和释放内存上花费的时间。用这种方法改进过的单位的结构就可能会成为如下形式:

```
//单位的数据
struct SUnit
{
    //普通属性
    long lID;                //单位的惟一 ID
    char * szName;           //单位的名字
    int iPosX;               //单位当前的位置 X
    int iPosY;               //单位当前的位置 Y
    …
    //图像属性
    int iCurDir;             //当前方向
    int iCurPose;            //当前动画
    int iImgWidth;           //当前图像宽度
    int iImgHeight;          //当前图像高度
    int iCurFrame;           //当前帧
    int iFrameCnt;           //当前动画帧的个数
    unsigned char * pImg;    //当前帧的数据指针
    CDirAnis ani[8];         //动画数据指针(8 个方向)
};
```

其中,CDirAnis 是用户从 CResObject 中继承下来的子类,用于保存游戏动画资源。

5.3 资源的缓冲池

(1)缓冲池的实现

上面讨论的资源管理方法虽然在很大程度上减少了程序读取文件、分配和释放内存的次数,但在一些特殊情况下,它就显得无能为力了。例如,在游戏中,有几个人不断地在玩家的周围行走,玩家看到的情况就是他们走进屏幕,然后走出去,接着又走进来。如果这几个绕圈走的玩家的图像都不同,那么还会出现不断地读取文件的现象,因为他们走出屏幕的时候会释放内存,而走进来的时候会从文件中读入数据到新分配的内存中。

要更好地解决这个问题,需要引入缓冲池的概念。在资源引用为 0 的时候并不是马上删除资源,而是缓冲一段时间,如果在这段时间内,又有对该资源的引用就不再需要重新读取资源文件和分配内存。

要实现缓冲池,只要简单地对 CResMgr 类做如下修改即可。

①添加一个缓冲区大小的成员变量。

②使它不立即删除引用为 0 的资源,可以隔一段时间去检测,也可以在创建或者删除其他资源的时候检测当前缓冲区状况以决定是否删除某些资源。

(2)缓冲的淘汰算法

缓冲的淘汰算法有很多种,选择用哪种淘汰算法取决于用户自己的选择。笔者自己选择的是淘汰最久没有使用的算法。

(3)更多的应用

有了缓冲池的资源管理就不再是为共享资源数据用了。事实上,如果把游戏中所有单位的内存也看成是一种资源,那么就可以为此建立一个单位的缓冲池。在一个游戏中,特别是网络游戏中,会频繁而且大量地创建和删除单位,所以这种缓冲池非常有助于提高管理单位的效率。

为了让资源管理类应用更加广泛,再为资源创建一种类型的属性很有必要。如单位的资源和图像资源就需要不同的属性,由于单位会被频繁地创建和删除,所以它需要一个比较大的缓冲区。而图像资源由于比较大,主要会需要共享数据而并不需要很大的缓冲区。所以不同的资源,可能有不同的需求。为资源添加类型的属性就可以为不同的资源分配不同大小的缓冲区,选择不同的淘汰算法。

游戏中有很多资源可以用上面描述的方法进行管理,如图像资源就非常适合用这种方法。很显然,这样可以大量节省内存,而且还避免了频繁地读取文件。

第6章 客户端基于3D多边形的寻路系统

Pathing-finding 的实现方法有很多种,如从起始点到目标点做连线,然后沿直线移动,碰到障碍就沿障碍边缘绕行;或者事先放置一些关键性的路点,然后在路点之间做遍历搜索;或者是基于格子,格子可以是四边形、六边形等等,利用格子之间的相连关系来遍历搜索。这里主要讲述第三种方法。

6.1 Pathing-finding 的基本原理

常见的 2D 游戏,如即时战略游戏或 RPG 游戏,地图系统通常都是由方块或菱形块拼接而成。整个地图是网格状的,每个格子自身都包含有"是否可以行走"的属性。用 0 表示可走,1 表示不可走,一张地图在计算机里就是一个由 0 和 1 组成的数组,如图 6.1 所示。

```
1 1 1 1 1 1 1 1 1 1
1 1 0 0 0 0 0 0 1 1
1 1 0 1 1 0 0 1 1 1
1 1 1 1 0 0 1 1 1 1
1 1 1 0 0 1 1 1 1 1
1 1 0 0 1 1 1 0 1 1
1 1 0 0 0 0 0 0 1 1
1 1 1 1 1 1 1 1 1 1
```

图 6.1

图中,代表地图宽度为 10,高度为 8,总共有 80(10 * 8)个格子;"1"都是不可走的地方,而"0"组成"Z"字形的一个通道。假设一个游戏中的主角站立在坐标为(2,1)的格子上,希望能通过 Pathing-finding 自己走到"Z"字形的尽头,也就是坐标(7,6)的位置上,这就需要计算机算法找到一条全部由"0"组成的通路。

从格子的相连关系来看,任意一个格子,与它相连的有 8 个其他格子,这些格子或可走或不可走,也就是主角有 8 个方向可供选择,而这 8 个方向对于主角来说,是完全平等的,不存在某个格子比另一个格子更优先考虑之类的加权描述。那么遍历这 8 个格子,碰到属性为 1,就是死路;碰到属性为 0,就形成一条通路。从(2,1)点出发第一次遍历形成了 2 条通路,6 条死路,把两条通路保存下来,并且把格子(3,2)和(2,2)的属性也设为 1,以防止下一个遍历中重复检查它们。在下一次的遍历中,需要分别检查(3,2)和(2,2)两个格子周围的 8 个格子,按这种方式遍历下去,很快通路的数量越来越多,就像洪水一样泛滥开来,把所有可走的地方全都塞满了。可以用如下的链表结构来表示这些通路:

```
struct NODE _ SHARED                    //寻路时公用的路径信息记录
  {
    UINT8 * buf _ ptr;                  //指向格子数组里对应位置的指针
    UINT8 dire;                         //下一个方向
    struct NODE _ SHARED * last _ node _ ptr;  //上一个节点
  };
  struct PATH _ LINK                    //最后生成的路径记录
  {
    UINT8 dire;                         //下一个方向
    PATH _ LINK * next _ path _ ptr;//路径中下一个节点
  };
  #define STEP _ LIMIT 65536
  static NODE _ SHARED node _ list[STEP _ LIMIT];
```

用一个很大的数组来存放整个寻路过程中产生的路径节点，而每一个节点都有一个指针指向上一个节点，并记录了从上一个节点到当前这个节点的方向。方向的定义很简单，就是围绕着某个格子周围其他格子共形成了 8 个方向，按顺时针方向分别是左上、上、右上、右、右下、下、左下、左。

随着对格子不断地遍历，新的路径节点不断产生，每个路径节点都有记录它的上一个路径节点，因此相当于记录了一条通路，直到碰到某个路径节点就是所要到达的目的地，遍历停止。从这个路径节点起通过链表指针回溯，就可以得到一个从出发点开始到目的地结束的链表，链表上每一个节点都记录了前往下一个节点的方向，也就得到了所需要的路径，而且这条路径必然是最短路径。这是因为是在不断遍历所有的通路，那么那条最先接触到目的地节点的通路就是最短的一条。其具体的代码如下：

```
//传入参数：
//block _ buf，由 0，1 信息组成的格子数组指针，类型是 char
//格子总宽度 width
//格子总高度 height
//搜索的起始坐标 sx，sy
//搜索的目的坐标 tx，ty
//返回值为路径链表的指针
PATH _ LINK * SearchPath(UINT8 * block _ buf, SINT16 width, SINT16 height,
SINT16 sx, SINT16 sy, SINT16 tx, SINT16 ty)
{
  if(block _ buf == NULL) return NULL;
  SINT16 off[8];                          //得到 8 个方向对应于格子数组的偏移量
  off[0] = width * (-1);
  off[1] = width;
  off[2] = -1;
  off[3] = 1;
  off[4] = (width - 1) * (-1);
  off[5] = width + 1;
  off[6] = width - 1;
  off[7] = (width + 1) * -1;
  UINT32 node _ count = 0;
  UINT32 current _ node = 0;
  UINT8 * target _ ptr = block _ buf + ty * width + tx;
  node _ list[0].buf _ ptr = block _ buf + sy * width + sx;
  node _ list[0].last _ node _ ptr = NULL;
```

```
UINT8 * last_buf_ptr, * new_buf_ptr;
PATH_LINK * path_link_ptr = NULL;
BOOL found_flag = FALSE;
long lMaxStep = (width - 1) * (height - 1);
if(lMaxStep > STEP_LIMIT) lMaxStep = STEP_LIMIT;//限制最大步数
while(!found_flag)
{
  if(node_count > STEP_LIMIT) break;
  if(current_node > node_count) break;//如果当前检查的节点计数已经到达目前的
                                      //节点总数,说明找不到路径了,退出
  last_buf_ptr = node_list[current_node].buf_ptr;
  for(register d = 0 ;d < 8 ;d ++)    //遍历 8 个方向
  {
    new_buf_ptr = last_buf_ptr + off[d];
    if( * new_buf_ptr == 0)           //如果该位置格子属性为 0,可走
    {
      * new_buf_ptr = 1;              //屏蔽掉此格子,避免重复处理
      node_count ++ ;                 //增加新节点
      node_list[node_count].buf_ptr = new_buf_ptr;
      node_list[node_count].dire = d;
      node_list[node_count].last_node_ptr = &node_list[current_node];
      if(new_buf_ptr == target_ptr)   //找到目的地
      {
        NODE_SHARED * node_ptr = &node_list[node_count];
        while(node_ptr->last_node_ptr)  //回溯链表,得到所需的路径
        {
          PATH_LINK * new_path = new PATH_LINK;
          new_path->dire = prior[node_ptr->dire];
          new_path->next_path_ptr = path_link_ptr;
          path_link_ptr = new_path;
          node_ptr = node_ptr->last_node_ptr;
        }
        found_flag = TRUE;
        break;
      }
    }
```

```
            }
            current _ node ++ ;                    //存放的节点依次逐个检查
        }
        if(found _ flag)
         {
            return path _ link _ ptr;
        }
        return NULL;
}
```

上面就是一个完整的寻路函数,可以直接投入使用,并且与具体的游戏类型和地图系统无关,返回值是一个包含每一步人物移动方向的链表。这种搜索算法与 A * 算法(有关 A * 算法的资料很多,读者可以另行查阅)不同,是不带任何加权值的,可以说是一种完全广度优先的算法,可能在有些场合比 A * 要花费更多时间,但算法本身过程简单,易于实现,速度也是够用的。其使用实例附在本书的光盘中。

6.2　3D 地形的概念:Terrain

在 3D 的世界里,所有的物件都是由多边形组成的。在一个游戏场景中,人物在地面上行走,地面同样也是多边形的集合,如图 6.2 所示。

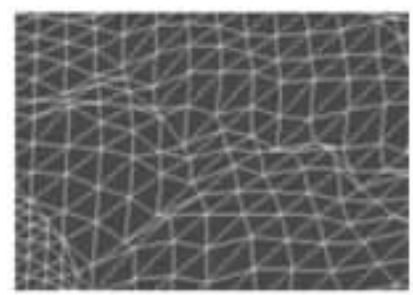


图 6.2　多边形组成的地形

通常,3D 人物在场景中行走要处理 2 个主要问题:一是 Terrain Follow,即沿着地形行走,同时根据脚下地形的高度和坡度来设置人物模型的所在高度和面对方向,这样人物在 Terrain 上行走看起来就像在真实世界里一样;二是 Collsion Detection,即常说的碰撞检测。有一种简化的方法可以把两者结合起来处理,就是在制作场景的时候,把可以行走的地面部分单独挖出来,生成其多边形模型,这样树、房子、石头之类人物不可通过的地方都不会包括在这个特制的地面部分之内,形成了一种独特的 Terrain,上面的每个多边形(三角形)都是人物可以抵达的,人物行走的范围会被限制在这块 Terrain 之上,就省掉了 Collision Detection 这个过程,可以针对这种 Terrain 来做 Pathing-finding。

在图 6.2 中,可以看到地面是由很多个相连的三角形构成的,我们可以把这些三角形想象为 2D 游戏中的格子,所不同的是这些格子是空间中的三角形,并不在同一个平面内。它们之间存在着固定的相连关系,每一个三角形有 3 条边,也就是最多可以与其他 3 个三角形邻边(不考虑顶点相连)。如果一个人物站立在某个三角形上,他可以选择 3 个方向来前进,这与 2D 游戏的四方格子类似,如图 6.3 所示。

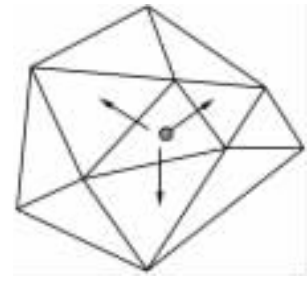


图 6.3　三角形的相连关系

需要建立一个数据结构来描述这种 Terrain。Terrain 是一个 3D 模型,和其他类型的 3D 模型一样,它包含了顶点信息和面信息。所谓面信息实际上就是构成面的每个顶点的编号,而顶点编号就是模型中顶点的存储位置的编号。

```
struct FACE _ INFO      //面信息
 {
    int m _ iFace[3] ;  //3 个顶点的编号
} ;
struct PATH _ NODE     //寻路用的节点信息
 {
    int iChildNo[3] ;   //相邻三角形的编号, - 1 表示无相邻
} ;
```

一个 Terrain 的数据就是 FACE _ INFO 的数组加上 PATH _ NODE 的数组。

下面来看看这些数据是怎样生成的。这里已经把与 Terrain 有关的数据和操作封装到一个叫做 CTerrain 的类里面,其生成地形数据的成员函数如下:

```
BOOL CTerrain :: GenerateTerrainPathInfo(CModel * pModel) ;
```

其中,pModel 是一个模型的指针。这里的模型格式并不是固定的,它取决于场景系统采用的模型格式,但不管是哪种格式,必然都会包含有顶点数据和面数据,所以实际上还是通用的。可以做一个 Interface 类 CModel,然后封装一些对实际模型数据的操作,把 3D 引擎的场景系统和用来做 Pathing-finding 的 Terrain 隔离开来。

```
BOOL CTerrain :: GenerateTerrainPathInfo(CModel * pModel)
 {
    int i, j ;
    if(m _ pFaces)      //清除以前的 Face 信息
     {
        delete m _ pFaces ;
        m _ pFaces = NULL ;
    }
    int nPolyCnt ;
    int * pnPoly = pModel - > GetModelTopology(&nPolyCnt) ;
    //请注意这个函数,它取得一个指针指向模型内部的多边形数据以及多边形
    //的数量。函数的内容取决于模型自身的结构定义

    m _ pFaces = new FACE _ INFO[nPolyCnt] ;//根据获得的多边形数量生成新的 Face 信息
    FACE _ INFO * pCurFace = m _ pFaces ;
    for( i = 0 ;i < nPolyCnt ;i ++ )
     {
        pCurFace - > m _ iFace[0] = pnPoly[0] ;     //取得当前三角形第一个顶点编号
```

```
    pCurFace->m_iFace[1] = pnPoly[1];      //取得当前三角形第二个顶点编号
    pCurFace->m_iFace[2] = pnPoly[2];      //取得当前三角形第三个顶点编号
    pCurFace++;
    pnPoly+=3; //这里的3指模型的数据结构中每个多边形的描述占用3个整数
               //位置,也是取决于具体的模型格式
}
FACE_INFO *pFaces = m_pFaces;
_nFaces = nPolyCnt;
PATH_NODE *pPathNode = new PATH_NODE[_nFaces]; //根据多边形的数量生成
                                               //相应的寻路节点数组
PATH_NODE *pCurNode = pPathNode;
FACE_INFO *p1, *p2;
p1 = pFaces;
for(i = 0;i < _nFaces;i++)
{
    for(j = 0;j < 3;j++)
    {
        pPathNode[i].iChildNo[j] = -1; //预设为-1,表示无任何三角形相连
    }
}
struct LINE_USED  //临时数据,通过记录对三角形顶点的公用来生成相邻关系
{
    BYTE m_iUsed[3][3];
};
LINE_USED *pLineUsed = new LINE_USED[_nFaces];
ZeroMemory(pLineUsed, _nFaces * sizeof(LINE_USED));
LINE_USED *pCurLineUsed = pLineUsed;
for(i = 0;i < _nFaces;i++)
{
    int nChild = 0;
    p2 = pFaces;
    for(j = 0;j < _nFaces;j++)
    {
        int iPublic = 0; //公用顶点预设为0
        int iUsed[2];
        if(j != i)
        {
```

```
            for(int v = 0 ;v < 3 ;v ++)
             {
                for(int u = 0 ;u < 3 ;u ++)
                 {
                    if(p1 -> m _ iFace[v] == p2 -> m _ iFace[u]) //发现有公用顶点
                     {
                        iUsed[iPublic] = v;
                        iPublic ++ ;
                    }
                    if(iPublic >= 2)
                     {
                        break;
                    }
                }
                if(iPublic >= 2) break;
            }
        }
        if(iPublic >= 2)
         {
            pCurLineUsed -> m _ iUsed[iUsed[0]][iUsed[1]] = 1;
            pCurLineUsed -> m _ iUsed[iUsed[1]][iUsed[0]] = 1;
            pCurNode -> iChildNo[nChild] = j; //如果有两个顶点公用,则说明
                                              //两个三角形相邻
            nChild ++ ;
        }
        p2 ++ ;
    } //end for j
    if(nChild < 3) //如果该三角形不是所有的边都与其他三角形公用,则说明
                   //它处于边缘的位置,这种三角形的边应记录下来,做边界检查
                   //的时候会用到
     {
        for(int m = 0 ;m < 3 ;m ++)
         {
            for(int n = 0 ;n < 3 ;n ++)
             {
                if(m == n) continue;
                if(pCurLineUsed -> m _ iUsed[m][n] == 0) //如果是出于边缘的三角形,
                //此三角形有一条边不与其他三角形公用,则
```

```
                    //记录其顶点编号
                    {
                        pCurNode->iChildNo[nChild] = -1 * (m * 10 + n); //这里利
                        //用 iChilNo 的位置来存储构成此边的两个顶点的编号,后面会解释
                        //是如何利用这项数据的
                        nChild ++;
                        pCurLineUsed->m_iUsed[n][m] = 1;
                    }
                }
            }
        }
        p1 ++;
        pCurNode ++;
        pCurLineUsed ++;
    }
    delete[] pLineUsed;
    m_pPathNode = pPathNode;
    //记录来自模型的顶点数据指针
    int nV, nVA, vLen;
    _pVertices = pModel->GetModelVertexPtr (&nV, &nVA, &vLen);
    _nVertices = nV * nVA; //顶点的数量
    _nVLen     = vLen; //单个顶点数据的长度
}
```

6.3 在 Terrain 上寻路

有了 Terrain 的三角形的相邻关系,便可以把它交给寻路函数来处理了,其过程和 2D 游戏相似。在 2D 寻路里提供给寻路函数的是格子坐标的起点和终点,而在这里提供一个起始多边形的编号和一个终点多边形的编号,然后寻路函数生成一个链表数据,记录途中经过的所有多边形的编号,也就是一条路径了。如何取得人物当前所在的多边形编号和它要去的地方的多边形编号呢?3D 系统里的坐标都是 float(x, y,z) 类型,给出一个 3D 坐标可以通过遍历检查取得它最接近的多边形,鼠标点击也可以通过类似于 ray-tracing 算法取得 2D 屏幕位置所对应的 3D 空间里的多边形,或者是对应的 3D 坐标,这些也都和具体的模型格式有关。

下面来看寻路函数:

```
struct PATH _ SHARED          //和 2D 一样,这里是一个寻路时公用的路径信息记录
 {
    int iNodeNo;              //这里实际上就是存放多边形的编号
    PATH _ SHARED * pLast;    //上一个节点
};
struct PATH _ LINK            //最后生成的路径信息节点
 {
    float fPos[3];            //和上一个多边形公用边的中点的 3D 坐标,后面会讲
                              述如何使用这个坐标
    int iNodeNo;              //多边形的编号
    PATH _ LINK * pNextLink;  //下一个节点
};
static BYTE RoadFlag[800][800];
```

其中,RoadFlag 是一个二维数组,初始值都是 0,用来表示 Terrain 上任意 2 个多边形之间的连接是否已经被检查过。如果是,则设为 1,即 RoadFlag[多边形编号 1][多边形编号 2] = 1,其作用是避免重复检查。需要注意的是,800 是假设一个 Terrain 最多包含的多边形数量,如果超过,则要加大这个数组的容量,但是消耗的内存也是非常巨大的。假设 Terrain 包含了 4 096个多边形,这个数组就要占用 16MB 内存。而且在每次执行寻路前需要把它置 0,时间消耗也是很大的。

在这种情况下,需要使用位来保存 0,1 信息,也就是每个字节可以对应于 8 个多边形编号,但是位操作势必比直接用 BYTE 花费更多的时间,在空间换时间的指导思想下,需要灵活处理。当多边形数量较小时,直接用 BYTE ;多边形数量很大时,则可以用位。

```
//传入参数
//pTer 是 Terrain 数据(指针)
//iNowPoly 是起始多边形的编号
//iDestPoly 是终点多边形的编号
//pfDestPos 是终点位置的 3D 坐标(float f[3]),之所以需要传入这个参数,是因为仅仅到
//达目的地所在的多边形还不够,要到达指定的 3D 坐标,所以这个坐标是路径的最后一
//个节点
PATH _ LINK * SeekPath(CTerrain * pTer, int iNowPoly, int iDestPoly, float * pfDestPos)
 {
    PATH _ NODE * pPathNodeList = pTer -> GetPathNode();

    ZeroMemory(RoadFlag, 640000); //把 RoadFlag 置 0
    PathList[0]. iNodeNo = iNowPoly;
    PathList[0]. pLast = NULL;
```

```
int iCurPathCnt = 0;
int iPathCnt = 0;
BOOL bFound = FALSE;
PATH_LINK *pPathLink = NULL;
while(1)
{
  if(iCurPathCnt > iPathCnt)
  {
    break;
  }
  int iNodeNo = PathList[iCurPathCnt].iNodeNo;     //取出当前节点的多边形编号
  for(int i = 0;i < MAX_NODE_CHILD;i++) //MAX_NODE_CHILD = 3,
                                        //因为1个三角形最多可以与其他3个三角形相邻
  {
    int iChildNo = pPathNodeList[iNodeNo].iChildNo[i]; //依次检查相邻的三角形
    if(iChildNo < 0) continue;
    if(RoadFlag[iNodeNo][iChildNo] == 0) //如果还不曾检查过
    {
      iPathCnt++; //增加一个新路径节点
      PathList[iPathCnt].iNodeNo = iChildNo; //记下对应的多边形编号
      PathList[iPathCnt].pLast = &PathList[iCurPathCnt];
      RoadFlag[iNodeNo][iChildNo] = 1;
      if(iChildNo == iDestPoly) //到达目的地多边形
      {
        PATH_SHARED *pPath = &PathList[iPathCnt];
        PATH_LINK *pNewLink = new PATH_LINK;
        memcpy(pNewLink->fPos, pfDestPos, 12);
        pNewLink->pNextLink = NULL;
        PATH_LINK *pPathRec = pNewLink;
        while(pPath->pLast) //回溯路径链表
        {
          pNewLink = new PATH_LINK;
          pTer->GetPolyLinkPos(pPath->pLast->iNodeNo,pPath->iNodeNo,
                                                    pNewLink->fPos);
          //此函数的作用是返回相邻的2个三角形共有边的中点坐标
          pNewLink->pNextLink = pPathRec;
```

```
                    pPathRec = pNewLink;
                    pPath = pPath->pLast;
                }
                pPathLink = pPathRec;
                bFound = TRUE;
                break;
            }
        }
    }
    if(bFound) break;
    iCurPathCnt ++;
  }
  if(!bFound)
   {
    Log("Search Over!not found Path, iPathCnt = <%d>", iPathCnt);
  }
  return pPathLink;
}
```

整个过程与 2D 的基本相同,只是把检查一个格子周围的 8 个格子变成了检查一个三角形周围的 3 个三角形,最后获得了一个记录路径信息的链表,如图 6.4 所示。

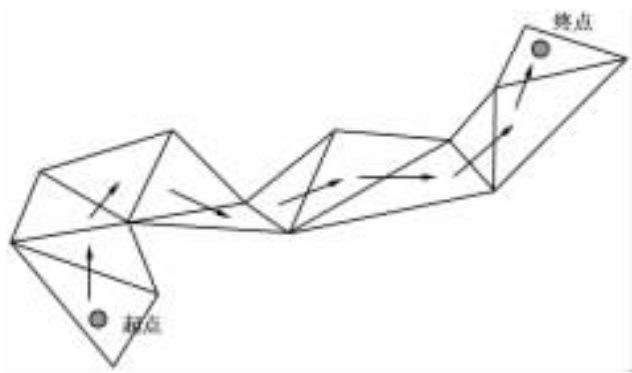


图 6.4 相邻的三角形序列组成的路径

由图 6.4 可以看出,人物行走的路线是三角形公有边中点的连线。在 CTerrain 类中有一个函数专门用来取得中点坐标。

```
//传入参数
//iPoly1,iPoly2 是 2 个多边形的编号
//pfPos 是用来返回中点坐标
VOID CTerrain::GetPolyLinkPos(int iPoly1, int iPoly2, float *pfPos)
 {
```

```
    FACE_INFO * pFace1 = &(m_pFaces[iPoly1]);
    FACE_INFO * pFace2 = &(m_pFaces[iPoly2]);
    //找到2个三角形共有的2个顶点
    int iFace = 0;
    int iFound = 0;
    int VexList[2];
    for(int i = 0;i < 3;i ++)
    {
        iFace = pFace1->m_iFace[i];
        for(int j = 0;j < 3;j ++)
        {
            if(iFace == pFace2->m_iFace[j])
            {
                VexList[iFound] = iFace;
                iFound ++;//共有顶点的数量计数
                break;
            }
        }
    }
    if(iFound < 2)
    {
        Log("Error GetPolyLinkPos(%d,%d)\n",iPoly1,iPoly2);
    }
    float *pV;//取得2个顶点的坐标数据
    pV = _pVertices + VexList[0] * _nVLen;
    float x1 = *(pV + 0);
    float y1 = *(pV + 1);
    float z1 = *(pV + 2);
    pV = _pVertices + VexList[1] * _nVLen;
    float x2 = *(pV + 0);
    float y2 = *(pV + 1);
    float z2 = *(pV + 2);
    if(pfPos != NULL) //取得中点的3D坐标
    {
        pfPos[0] = (x1 + x2) / 2.0f;
        pfPos[1] = (y1 + y2) / 2.0f;
        pfPos[2] = (z1 + z2) / 2.0f;
    }
}
```

如果人物在 Terrain 的多边形上行走,一直走的是三角形边的中点,就会引发一个问题:当三角形比较规则,也就是接近正三角形,并且三角形的尺寸都相差不大时,人物的行走看起来很正常。但是当三角形是钝角,并且尺寸相差很大时,沿着边的中点行走就会看起来很怪异。有两个办法来解决这个问题:一是在制作模型的时候就要求 Terrain 的三角形大小均匀,并且都接近正三角形,甚至可以用程序来专门生成这种 Terrain 模型;二是在人物行走的过程中对行走路线做优化,如预先检查下一个中点是否可以直接到达,或者把中线分成几段,按照距离较短的那个点来走其连线。在实际使用中,采用的是第二种方法。

对行走路线的优化的逻辑为:

①检查最终目的地是否可以直接到达,如果不能则转入②。

②从当前位置开始对即将前往的路径节点进行检查,如果可以直接到达,就检查下一个,直到碰到一个不能直接到达的节点为止,也就是说跳过了最后一个可以直接到达的节点的前面那些节点,转入③。

③人物开始移动,移动的过程中每经过一定距离就循环跳到①。

经过这一番处理之后,人物的行走就不会固定在三角形边的中点之上了,而是走一条看起来最近的路线。当前有些三角形的形状如果比较极端,仍然会产生一点绕路的迹象,但总体感觉已经很不错了,如图 6.5 所示。

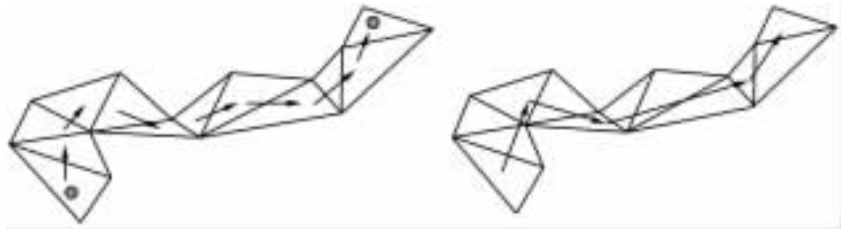


图 6.5　行走中,跳过一些三角形中点

判断是否可以直接到达是看当前位置与目标点的位置的连线,是否与 Terrain 的边界相交即可。Terrain 的边界是由三角形的边组成的,这些边的特征就是不为其他三角形所共有。我们事先在 Terrain 的数据就已经保存了这些信息,具体请参看前面生成 Terrain 数据的函数 BOOL CTerrain::GenerateTerrainPathInfo(CModel *pModel)。

下面是一个 CTerrain 类的函数,专门用来检查线段是否与 Terrain 边界相交:

```
//传入参数
//fline 为线段 2 个点的坐标
BOOL CTerrain::IsIntersectEdge(float *fLine)
{
    //依次检查 Terrain 边缘上所有的边是否与指定线段相交,但只限于 2D 范围内,Z
    //坐标忽略不计,当 Terrain 有比较大的坡度时,会引起一点误差,但也可以忽略不计
    for(int j = 0 ;j < _nFaces ;j ++)
    {
        PATH_NODE *pNode = &m_pPathNode[j] ; //取出三角形数据
```

```
for(int i = 0 ;i < 3 ;i ++)
 {
   int iFlag = pNode -> iChildNo[i] ;
   if(iFlag < 0) //如果小于零,就说明是 Terrain 的边缘,前面已约定了这个规
                 //则,请参看函数
                 //GenerateTerrainPathInfo(CModel * pModel)
    {
      iFlag = iFlag * - 1 ;
      int v1, v2 ;
      v1 = iFlag / 10 ;
      if(iFlag < 10) //取出 2 个顶点在数组中的位置
       {
         v2 = iFlag ;
       }
      else
       {
         v2 = iFlag - 10 ;
       }
      float p1[2], p2[2] ;
      FACE _ INFO * pFace = &m _ pFaces[j] ;
      int c1 = pFace -> m _ iFace[v1] ;              //取出 2 个顶点的编号
      int c2 = pFace -> m _ iFace[v2] ;
      float * pV = _ pVertices + c1 * _ nVLen ;      //取出第一个顶点坐标
      p1[0] = *(pV + 0) ;
      p1[1] = *(pV + 1) ;
      pV = _ pVertices + c2 * _ nVLen ;              //取出第二个顶点坐标
      p2[0] = *(pV + 0) ;
      p2[1] = *(pV + 1) ;
      float s1[2],s2[2] ;
      s1[0] = fLine[0] ; s1[1] = fLine[1] ;
      s2[0] = fLine[2] ; s2[1] = fLine[3] ;
      float fTmp[2] ;
      if(IsLineIntersect(s1, s2, p1, p2, fTmp, FALSE)) //这个函数检查 2
      //条 2D 的线段是否相交,网上有不少这种算法的资料,可以自行查阅
       {
         return TRUE ;
       }
    }
```

```
    }
    return FALSE;
}
```

Pathing-finding 中最关键的是搜索算法。无论在 2D 还是 3D 环境中,搜索算法都是基本不变的。在效率上,除了搜索算法本身的优化,如何应用好也是很重要的。例如在网络游戏中会有很多玩家同时操作,如果每个玩家控制的人物都不断地做 Pathing-finding,游戏速度就会受到影响,因此可以限定算法搜索的范围来减少花费。在最近的网络游戏项目中,只有当前玩家自己控制的人物会做 Pathing-finding,并把行走的路径切分成路点传到服务器上,而其他玩家是接受从服务器传来坐标而行动,因此 Pathing-finding 对游戏速度基本上没有影响。又如在著名的即时战略游戏《星际争霸》中,如果玩家同时指挥很多个 SCV 前往地图上某一位置,这些 SCV 就会走成一条串葫芦的路线,因为不是每个 SCV 都做寻路,而是公用了第一个 SCV 的路径,这样就节省了很多计算花费的时间。

第7章

简繁体文字转换的处理

在国际化的今天,游戏同样也要实现国际化。作为一个软件产品,国际化最主要的问题就是语言的问题。虽然现在有很多翻译软件,但是都不能完全正确地翻译语句。所以在需要做多国语言版本的时候通常都需要为新的语言专门做一套资源。这套资源包括所有和语言相关的东西,主要是文字、图像和语音。

如果图像上出现文字,那么需要修改为对应语言版本的文字。如果找不到这幅图片的源文件,可能就需要重新画一次了。特别是在一些游戏运行一段时间以后才需要做某种语言版本的时候这种情况比较常见。这样就会大量增加制作多语言版本的工作量。

对于语言,没有其他办法,如果需要对应语言的语音,只有重新录一次。对于文字,相对来说工作量最少,通常只需重新输入一次即可。但需要注意以下问题。

①如果程序中出现与用户交互的字符串,在制作新的语言版本的时候就需要修改它,这就意味着重新编译程序。

②如果程序中出现指定字符串长度的 char[],就有可能在制作新的语言版本的时候需要重新定义长度,这也意味着需要重新编译程序。例如,“钱”翻译成英文的时候变成“Money”,长度就比原来大了 3 个字符。所以在制作外文版的时候,发现翻译出来的语句长度大于定义的长度,这时就只有重新定义新的长度。

③游戏的按钮不能正好做到当前语言文字上的大小。同样是一些文字在特定语言版本里可以做到很小巧,但是在其他语言中可能同样意思的文字就变成了又长又扁,如果按钮不够大,就会放不下。

有一种特殊的语言文字——繁体中文,除个别的习惯用法外和简体中文基本上是一一对应的,所以可以用一些规则进行简体中文和繁体中文之间的转换。事实上,现在网络上已经有很多类似的软件,微软的 Word 还内置了简繁体转换的功能。下面详细介绍如何在游戏中实现简繁体转换的功能。

7.1 文字的编码

中文字不同于由字母组成的英文字。中文字都是采用 ASCII 编码的方式来表示,由于汉字的数目非常多,用 1 个 ASCII 字符显然是不够的,目前大都使用 2 个 ASCII 字符来表示 1 个汉字。从 ASCII 到汉字的转换是用查表的方式进行的,不同的码表决定了不同的汉字系统。目前比较常用的简体字码表是 GBK,而繁体字的码表是 BIG5。

(1) BIG5 的码区

目前繁体中文使用的编码规则是 BIG5 码,其分布如表 7.1 所示。

表 7.1

编　码	码　区	说　明
符　号	A140 ~ A3FE	实际止于 A3BF
常用字	A440 ~ C67E	
次常用字	C940 ~ F9FE	实际止于 F9D5,但倚天系统有 F9D6 ~ F9DC 制表符号有 F9DD ~ F9FE
空白区	A040 ~ A0FE,C6A1 ~ C8FE,FA40 ~ FEFE	

BIG5 码区分布图如图 7.1 所示。

黑色表示 BIG5 用到的码区。从这个分布图可以看出，BIG5 的编码并非 2 个字节值都在 128 以上，而是第 1 个字节总在 128 以上，而第 2 个字节有可能在 128 以下。

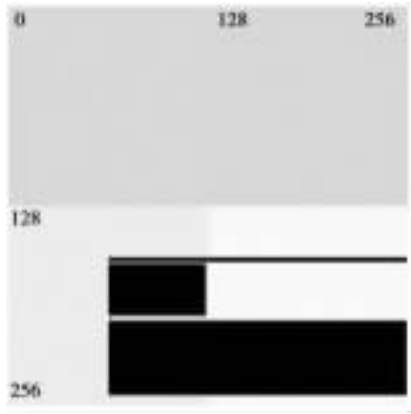


图 7.1　BIG5 码区分布图

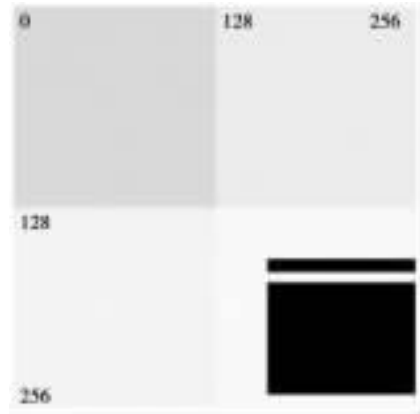


图 7.2　GB2312 码区分布图

(2) GB2312 的码区

目前简体中文使用的编码规则是 GBK。GBK 码的分布如表 7.2 所示。

编码的总体范围是 8140 ~ FEFE，首字节 81 ~ FE，尾字节 40 ~ FE，剔除 xx7F。

表 7.2

编　码	码　区
GBK/3	8140 ~ A0FE
GB2312 非汉字(GBK/2)	A1A1 ~ A9FE
GB13000 非汉字(GBK/5)	A840 ~ A9A0
GBK/4	AA40 ~ FEA0
GB2312 汉字(GBK/2)	B0A1 ~ F7FE
用户自定义区	AAA1 ~ AFFE
用户自定义区	A140 ~ A7A0
用户自定义区	F8A1 ~ FEFE

目前使用到的汉字都在 GB2312 的码区，所以，只对 GB2312(包括汉字和非汉字部分)进行处理就可以了。GB2312 的码区分布图如图 7.2 所示。由分布图可以看出，GB2312 编码的 2 个字节的值都大于 128。

7.2　码表的生成

因为 BIG5 和 GB2312 的编码并没有一个转换的公式，所以需要建立一个转换表来进行转换。要建立转换表，事实上，不需要从头做起，现在网络上有各种各样的转换程序，这里只需建

立一张需要转码的文字和符号的表格,然后用转换程序进行转换就可以得到对应的码表文件。下面的类就可以完成这个功能。

```
//////////////////// LanTranformTable.h ////////////////////
#ifndef _LAN_TRANFORM_TABLE_H_
#define _LAN_TRANFORM_TABLE_H_
//GB2312 与 BIG5 转换码表生成程序
class    CLanTransformTable
 {
public:
    //构造函数
    CLanTransformTable();
    //析构函数
    ~CLanTransformTable();
    //创建 BIG5 码表
    bool CreateBIG5Table(const char* lpszFileName);
    //创建 GB2312 码表
    bool CreateGB2312Table(const char* lpszFileName);
protected:
    //检测一个首字节为 bFirst,尾字节为 bSecond 的编
    //码是否是 BIG5 的字符
    bool IsBIG5Char(unsigned char bFirst,unsigned char bSecond);
    //检测一个首字节为 bFirst,尾字节为 bSecond 的编
    //码是否是 GB2312 的字符
    bool IsGB2312Char(unsigned char bFirst,unsigned char bSecond);
    //写文件头部的信息
    void WriteHeadInformation(void *fpFile);
    //写文件尾部的信息
    void WriteTailInformation(void *fpFile);
};
#endif  //_LAN_TRANFORM_TABLE_H_
////////////////////LanTranformTable.cpp ////////////////////
//转换码表生成程序
#include      <stdio.h>
#include      "LanTranformTable.h"
//构造函数
CLanTransformTable::CLanTransformTable()
 {
}
```

```
//析构函数
CLanTransformTable :: ~CLanTransformTable()
{
}
/* 检测一个首字节为 bFirst,尾字节为 bSecond 的编码是否是 BIG5 的字符
|-------------------------------------------------------------------------------|
| BIG5 的编码范围
| 符号       | A140 ~ A3FE
| 常用字     | A440 ~ C67E
| 次常用字   | C940 ~ F9FE
| 空白区     | A040 ~ A0FE,C6A1 ~ C8FE,FA40 ~ FEFE
|-------------------------------------------------------------------------------|
*/
bool CLanTransformTable :: IsBIG5Char(unsigned char bFirst,
        unsigned char bSecond)
{
    if( !((bSecond >= 0x40 && bSecond <= 0x7E)
        ||(bSecond >= 0xA1 && bSecond <= 0xFE)))
        return false;
    //符号 A140 ~ A3FE
    if(bFirst >= 0xA1 && bFirst <= 0xA3)
    {
        //实际只到 A3BF
        if(    (bFirst == 0xA1 && bSecond < 0x40)
            ||(bFirst == 0xA3 && bSecond > 0xBE))
            return false;
        return true;
    }
    //常用字 A440 ~ C67E
    if(bFirst >= 0xA4 && bFirst <= 0xC6)
    {
        if(    (bFirst == 0xA4 && bSecond < 0x40)
            ||(bFirst == 0xC6 && bSecond > 0x7E))
            return false;
        return true;
    }
    //次常用字 C940 ~ F9FE
```

```
    if(bFirst >= 0xC9 && bFirst <= 0xF9)
     {
        if(    (bFirst == 0xC9 && bSecond < 0x40)
            ||(bFirst == 0xF9 && bSecond > 0xFE))
            return false;
        return true;
    }
    return false;
}
/* 检测一个首字节为 bFirst,尾字节为 bSecond 的编码是否是 GB2312 的字符
|-------------------------------------------------------------------------------|
| GB2312 的编码范围
| 非汉字        A1A1 ~ A9FE
| 汉字          B0A1 ~ F7FE
|-------------------------------------------------------------------------------|
*/
bool CLanTransformTable :: IsGB2312Char(unsigned char bFirst,
        unsigned char bSecond)
 {
    //总体范围是 8140 ~ FEFE,剔除 xx7F
    if(bFirst  < 0x81 || bFirst > 0xFE
        || bSecond < 0x40 || bSecond > 0xFE || bSecond == 0x7F)
        return false;
    //GB2312 非汉字(GBK/2)   A1A1 ~ A9FE
    if(bFirst >= 0xA1 && bFirst <= 0xA9
        && bSecond >= 0xA1 && bSecond <= 0xFE)
        return true;
    //GB2312 汉字(GBK/2)   B0A1 ~ F7FE
    if(bFirst >= 0xB0 && bFirst <= 0xF7
        && bSecond >= 0xA1 && bSecond <= 0xFE)
        return true;
    //其他
    return false;
}
//创建文件名为 lpFileName 的 BIG5 的转换码表文件
bool CLanTransformTable :: CreateBIG5Table(const char *lpszFileName)
 {
    FILE     *fp;
```

```
//打开文件
fp = fopen(lpszFileName,"wb");
if(fp == 0)
 {
    //文件打开或者创建失败
    return false;
}
//写入文件头信息
this -> WriteHeadInformation(fp);
unsigned char    firstByte;                  //首字节
unsigned char    secondByte;                 //第二字节
unsigned char    space = 0x20;               //空格符号
unsigned char    r1 = 0x0D,r2 = 0x0A;        //回车换行符号
for(firstByte = 0xA1;firstByte <= 0xFE;firstByte ++)
 {
    for(secondByte = 0x40;secondByte <= 0x7E;secondByte ++)
     {
        if(IsBIG5Char(firstByte,secondByte))
         {
            //写入编码
            fwrite(&firstByte,1,1,fp);
            fwrite(&secondByte,1,1,fp);
        }else
         {    //不是 BIG5 字符,就写入空格填充
            fwrite(&space,1,1,fp);
            fwrite(&space,1,1,fp);
        }
    }
    for(secondByte = 0xA1;secondByte <= 0xFE;secondByte ++)
     {
        if(IsBIG5Char(firstByte,secondByte))
         {    //写入编码
            fwrite(&firstByte,1,1,fp);
            fwrite(&secondByte,1,1,fp);
        }else
         {    //不是 BIG5 字符,就写入空格填充
            fwrite(&space,1,1,fp);
            fwrite(&space,1,1,fp);
```

```
            }
        }
        //为了加快查表的速度,使每行凑足 160 * 2 个字节
        fwrite(&space,1,1,fp);
        fwrite(&space,1,1,fp);
        fwrite(&space,1,1,fp);
        fwrite(&space,1,1,fp);
        //回车换行
        fwrite(&r1,1,1,fp);
        fwrite(&r2,1,1,fp);
    }
    //写入文件尾信息
    this->WriteTailInformation(fp);
    //关闭文件
    fclose(fp);
    //写入信息完成
    return true;
}
//创建文件名为 lpFileName 的 GB2312 的转换码表文件
bool CLanTransformTable::CreateGB2312Table(
        const char *lpszFileName)
{
    FILE     *fp;
    //打开文件
    fp = fopen(lpszFileName,"wb");
    if(fp==0)
    {
        //文件打开或者创建失败
        return false;
    }
    //写入文件头信息
    this->WriteHeadInformation(fp);
    unsigned char    firstByte;              //首字节
    unsigned char    secondByte;             //第二字节
    unsigned char    space=0x20;             //空格符号
```

```
    unsigned char     r1 = 0x0D,r2 = 0x0A;    //回车换行符号
    for (firstByte = 0xA1 ;firstByte <= 0xF7  ;firstByte ++)
     {
        for (secondByte = 0xA1 ;secondByte <= 0xFE ;secondByte ++)
         {
            if (IsGB2312Char(firstByte,secondByte))
             {    //写入编码
                fwrite(&firstByte,1,1,fp);
                fwrite(&secondByte,1,1,fp);
            } else
             {    //不是 GB2312 字符,就写入空格填充
                fwrite(&space,1,1,fp);
                fwrite(&space,1,1,fp);
            }
        }
        //为了加快查表速度,使每行凑足 96 * 2 个字节
        fwrite(&space,1,1,fp);
        fwrite(&space,1,1,fp);
        //回车换行
        fwrite(&r1,1,1,fp);
        fwrite(&r2,1,1,fp);
    }
    //写入文件尾信息
    this -> WriteTailInformation(fp);
    //关闭文件
    fclose(fp);
    //写入信息完成
    return true;
}
//写文件头部的信息
void CLanTransformTable :: WriteHeadInformation(void * fpFile)
 {
    FILE * fp = (FILE *)fpFile;
    //写文件头部的信息
}
//写文件尾部的信息
void CLanTransformTable :: WriteTailInformation(void * fpFile)
 {
```

```
    FILE * fp = (FILE *)fpFile;
    //写文件尾部的信息
}
```

以上程序就可以创建出一个编码文件,然后用一个转码软件转换就可以生成需要的编码转换文件。

7.3 简繁转换程序

有了转换码表,下面就来讨论如何进行转换。

由码表的生成方法知道,要转换编码只要找出这个编码对应于码表的位置并读出该位置的值就完成了转换,如图 7.3 所示。

生成的码表　　转码后的码表

图 7.3

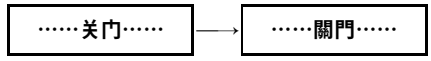


图 7.4

如果需要转换文字“关门”,就可以根据码表文件的规则找到对应文字的编号,再到码表中取出该位置的文字,就是已经转换过的“關門”,如图 7.4 所示。

下面的类就可以完成这个功能。

```
//////////////////// LanTranform.h ////////////////////////////
#ifndef _LAN_TRANSFORM_H_
#define _LAN_TRANSFORM_H_
//GB2312 与 BIG5 转码程序
class    CLanTransform
 {
public:
    CLanTransform();
    ~CLanTransform();
    //初始化转换程序,读入转换码表文件
    bool Init(const char *lpBigFile,const char *lpGBFile);
    //转换一个指定的 BIG5 编码的字符串到 GB2312 编码的字符串
    char *BIG2GB(char *lpszBuf);
    //转换一个指定的 GB2312 编码的字符串到 BIG5 编码的字符串
    char *GB2BIG(char *lpszBuf);
```

```
    //如果(isBig2Gb == true),就转换 BIG5 到 GB2312
    //如果(isBig2Gb == false),就转换 GB2312 到 BIG5
    inline char * Convert(char * lpBuf,const bool isBig2Gb)
     {
        if (isBig2Gb)     return BIG2GB(lpBuf);
        else              return GB2BIG(lpBuf);
    }
    protected:
        bool LoadTable(const char * lpFileName,const bool isBIG5);
        bool FreeTableBuf(const bool isBIG5);
    protected:
        unsigned char * pBIG5;        //BIG5 字符表
        unsigned char * pGB2312;      //GB2312 字符表
    };

    #endif   //_LAN_TRANSFORM_H_
////////////////////// LanTranform.cpp ////////////////////////
//BIG5 与 GB2312 编码转换程序
#include       <stdio.h>
#include       "LanTranform.h"
//构造函数
CLanTransform::CLanTransform()
 {
    pBIG5 = 0;
    pGB2312 = 0;
}
//析构函数
CLanTransform::~CLanTransform()
 {
    //释放码表
    FreeTableBuf(true);
    FreeTableBuf(false);
}
//初始化转换程序,读入转换码表文件
bool CLanTransform::Init(const char * lpBigFile,cosnt char * lpGBFile)
 {
    //读入 BIG5 转换 GB2312 的码表文件
    if (lpBigFile)
     {
```

```
            if (false == this->LoadTable(lpBigFile,true))
             {
                return false;    //读入失败
            }
        }
        //读入 GB2312 转换 BIG5 的码表文件
        if (lpGBFile)
         {
            if (false == this->LoadTable(lpGBFile,false))
             {
                return false;    //读入失败
            }
        }
    }
    //读入成功
    return true;
}
//转换一个指定的 BIG5 编码的字符串到 GB2312 编码的字符串
//   lpBuf 是指定字符串的指针
//   如果不能转换,就返回 0 否则,返回 lpBuf 指向的地址
char *CLanTransform::BIG2GB(char *lpBuf)
 {
    unsigned char *pCur,*pDest,bFirst,bSecond,bIndex ;
    //如果没有初始化数据或者没有需要转换的数据,就直接返回
    if (pBIG5 == 0 || lpBuf == 0)
     {
        return 0;
    }
    pCur = (unsigned char*)lpBuf;
    while(*pCur)     //处理到字符串结束
     {
        bFirst = *pCur;              //取得第一个字节的值
        bSecond = *(pCur+1);     //取得第二个字节的值
        if (bFirst<0xA1 || bFirst>0xFE)     //如果是英文字符
         {
            //不转换,直接跳到下一个字符
            pCur ++ ;
        }else if (bSecond<0x40 //如果是非 BIG5 的其他字符
            || bSecond>0xFE || (bSecond>0X7E&&bSecond<0XA1))
         {
```

```
        //不转换,直接跳到下一个字符
        pCur ++ ;
    } else      //是 BIG5 字符,需要转换
     {
        bIndex = bFirst - 0xA1 ;
        //由
        //if (bSecond < 0x7F)
        // {
        //      d = pBIG5 + ((temp * 0xA0 + (bSecond - 0x40)) << 1) ;
        //} else
        // {
        //      d = pBIG5 + ((temp * 0xA0 +
        //                   (bSecond + (0x7F - 0x40) - 0xA1)) << 1) ;
        //}
        //可以优化成下面的语句
        pDest = pBIG5 + (((bIndex << 7) + (bIndex << 5)) << 1) ;
        if (bSecond < 0x7F)
         {
            pDest += (bSecond - 0x40) << 1 ;
        } else
         {
            pDest += (bSecond + 0x3F - 0xA1) << 1 ;
        }
         * pCur ++ = * pDest ++ ;     //第一个字节
         * pCur ++ = * pDest ;        //第二个字节
    }
    }
    return lpBuf ;
}
//转换一个指定的 GB2312 编码的字符串到 BIG5 编码的字符串
//  lpBuf 是指定字符串的指针
//  如果不能转换,就返回 0 否则,返回 lpBuf 指向的地址
char * CLanTransform :: GB2BIG(char * lpBuf)
 {
    unsigned char * pCur, * pDest,bFirst,bSecond,bIndex ;
    //如果没有初始化数据或者没有需要转换的数据,就直接返回
    if (pGB2312 == 0 || lpBuf == 0)
     {
        return 0 ;
```

```
    }
    pCur = (unsigned char *)lpBuf;
    while(*pCur)    //如果字符串结束
    {
        bFirst = *pCur;            //取得第一个字节的值
        bSecond = *(pCur+1);       //取得第二个字节的值
        if (bFirst<0xA1 || bFirst>0xF7)    //如果是英文字符
        {
            pCur ++ ;     //不转换,直接跳到下一个字符
        }else if (bSecond<0xA1 || bSecond>0xFE)   //非 GB2312 字符
        {
            pCur ++ ;     //不转换,直接跳到下一个字符
        }else      //是 GB2312 字符,需要转换
        {
            bIndex = bFirst - 0xA1;
            //由 d = pGB2312 + ((temp*0x5E + (bSecond - 0xA1))<<1);
            //可以优化成下面的语句
            pDest = pGB2312 + (((bIndex<<6)+(bIndex<<5)
                + (bSecond - 0xA1))<<1);
            *pCur ++ = *pDest ++ ;     //第一个字节
            *pCur ++ = *pDest;         //第二个字节
        }
    }
    return lpBuf;
}
//从指定文件中读入字符转换表
//lpFileName 字符转换表的文件名
//bIsBIG5 是否是 BIG5 的转换表
//返回 true,读入成功 否则失败
bool CLanTransform::LoadTable(const char *lpFileName,const bool bIsBIG5)
{
    FILE *fp;
    fpos_t          pos;
    unsigned long   fileSize;
    unsigned char *pData;
    if (bIsBIG5)
    {
```

```
    //如果数据已经有读入
    if (pBIG5)
     {
        //释放原来读入的数据准备重新读入
        this->FreeTableBuf(bIsBIG5);
        pBIG5 = 0;
    }
} else
 {
    //如果数据已经有读入
    if (pGB2312)
     {
        //释放原来读入的数据准备重新读入
        this->FreeTableBuf(bIsBIG5);
        pGB2312 = 0;
    }
}
//打开文件
if ((fp = fopen(lpFileName,"rb")) == NULL)
 {
    return false;    //文件打开失败
}
//取得文件大小
fseek(fp,0,SEEK_END);
fgetpos( fp,&pos );
fileSize = (unsigned long)(pos&0x00000000FFFFFFFF);
//跳回到文件头部
fseek(fp,0,SEEK_SET);
//在这里跳过文件头部信息
//...
//分配内存
pData = new unsigned char[fileSize];
if (pData == NULL)       //如果内存分配失败
 {
    fclose(fp);          //关闭文件
    return false;        //内存分配失败
}
//读取文件数据到内存中
```

```
    fread(pData,fileSize,1,fp);
    //关闭文件
    fclose(fp);
    if (bIsBIG5)
    {
        pBIG5 = pData;
    }else
    {
        pGB2312 = pData;
    }
    return true;                //完成
}
//释放字符表
//bIsBIG5 是否是 BIG5 的字符表
bool CLanTransform::FreeTableBuf(const bool bIsBIG5)
{
    if (bIsBIG5)
    {
        if (pBIG5)              //如果有数据
        {
            delete[] pBIG5;     //删除数据
            pBIG5 = 0 ;         //清空指针
        }
    }else
    {
        if (pGB2312)            //如果有数据
        {
            delete[] pGB2312;   //删除数据
            pGB2312 = 0 ;       //清空指针
        }
    }
    return true;
}
```


7.4 如何更聪明地转换

至此,已经可以完成一般意义上的简繁体转换了,也就是说把相应的简体字的编码转换成相应的繁体字的编码。很多时候这样已经可以解决问题了,但是这种方法的天生缺点就是文

字只能一一对应，因此在碰到一些复杂情况的时候就没有办法解决问题。由于文化上的差异，使用繁体中文的区域与使用简体中文的区域对一些物体的名词是有不同的定义的。例如，简体叫光驱，繁体就叫光碟机；简体叫内存，繁体就叫记忆体；简体叫汇编语言，繁体就叫组合语言。对于这些词如果只是在文字编码上转换是不能解决问题的。还有一些特别的字，比如说繁体叫“乾淨”，简体叫“干净”；可是繁体叫“乾隆”，简体还是叫“乾隆”。那么在繁体转简体的时候就不能确定要把“乾”字的编码转成“干”的编码还是“乾”字的编码。这样的例子很多，那么如何做更聪明的转换呢？

事实上，要解决这个问题还是依赖于表格。可以建立一张对应表，这张对应表保存的是简体的词组以及转换成繁体后的词组。这样，当需要转换某一个字的时候就去查找这张表，如果表内存有有关这个字的词组就去比对有没有和当前需要转换的词一样的词组，如果有就把对应的词组都复制到目标地去，如果没有就只转换当前的字。

例如，需要转换“内存”二字，其步骤如下：

①当找到“内”字就去找第一个汉字为内字的词组。

②找到有关内字的词组有：“内部”、“内存”、“内地”、“内阁”、“内涵”等等。

③查找有没有和“内存”一词匹配的词组。

④发现有词组内存，就去找对应的表格保存的内存翻译的编码。

⑤找到内存对应的编码“記憶體”。

⑥复制词组“記憶體”到目标区域。

但是，如果找“内力”可能找不到，所以就只转换“内”字，下一个字就转换“力”字。

这种方法虽然可以解决很多转换问题，但还是会有部分出现错转的状况，如有一句话是“银行内存钱不多了”。如果用这种方法就会转换成“銀行記憶體錢不多了”，很显然转换错了。尽管如此，这种方法还是大大减少了开发另外一种版本的工作量。

第8章 游戏服务器的架构和设计要点

编写大型网络游戏的服务端程序是一个庞大的工程。不仅需要开发认证服务器、游戏服务器等多个服务端程序,而且其中最重要的游戏服务器又具有相当的开发难度,是对开发人员技术与毅力的一次考验。

8.1 大型网络游戏的起源与发展

早在 1979 年 Roy Trubshaw 和 Richard Bartle 在英国艾塞克丝大学开发了一个可以多人连线的游戏。当时的游戏没有图形,只有一行行文字,被称为 MUD(Multi-user Dungeon),国内一般称作泥巴游戏。

尽管 MUD 游戏没有漂亮的画面,只有冷冰冰的文字,但人与人之间交流与竞争的魅力是单机游戏所无法比拟的。因此,MUD 游戏还是拥有一大批爱好者。

在 MUD 游戏漫长的发展过程中,演化出许多种类型。国内流行的 MUD 游戏,如侠客行、风云、书剑等都属于 LPMud,其特点是整个游戏由解释程序(mudos) + 脚本(mudlib)组成。脚本由 mudos 解释执行,其中游戏层的内容都是由脚本编写的,因此修改游戏规则无须更改主程序。

最早的图形网络游戏是网络创世纪(UO),它实现了巨大地图即时拼接、人物换装染色系统、多内码识别等许多技术,直到今天仍被评价为有史以来技术成就最高的网络游戏。

8.2 整体结构

一般地,网络游戏的整体结构如图 8.1 所示。

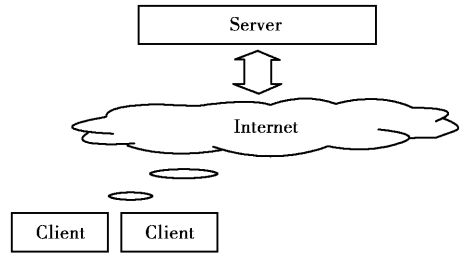


图 8.1

从大块的功能来看,有以下几点,如图 8.2 所示。

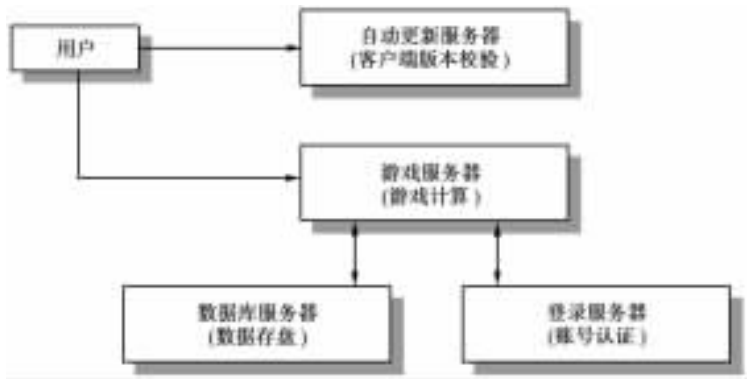


图 8.2

- 客户端版本的校验
- 账号认证
- 游戏的进行
- 玩家数据的存盘

8.3 游戏服务器

8.3.1 总体框架

游戏服务器需要处理的内容相当多,如玩家网络数据的传输、玩家命令的解析、游戏规则的实现等等。因此游戏器需要分层来设计,一方面可以使整体结构清晰,另一方面方便多人同时编码。

游戏服务器可以分为 3 层来设计,如图 8.3 所示。最底层是网络层,负责网络数据的接收与发送。中间层是命令处理层,负责命令的接收、发送与解析。上层是最复杂的虚拟世界层,在这一层上,游戏中的各种物体按既定的规则运行着,并发生交互关系。

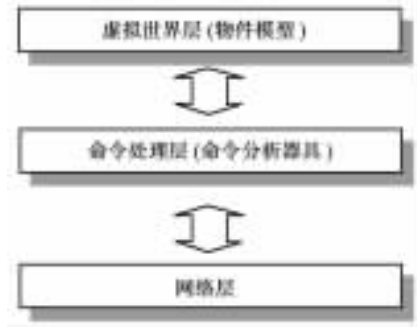


图 8.3

8.3.2 命令分析器

在游戏服务器与客户端的通信过程中,需要几十甚至上百条命令来传递各种消息,为了区分解析这些命令,可以采用以下几种方法:

- 使用 switch
- 使用 if. . . else if. . .
- 编写一个命令分析器

如果使用 switch 来区分命令,则会限制命令 ID 必须是整型变量,因为 C ++ 中的 switch 不能处理字符串。如果使用 if. . . else if 结构来处理,由于 VC ++ 的编译器对 if. . . else if 的长度有限制,在命令非常多时,必须分拆成几个函数。较好的方法是编写一个命令分析器,使用前向命令分析器注册一批命令,其中包括命令的索引与处理函数。在使用时,根据索引取出处理

函数并调用。

首先介绍一下函数指针。C ++ 中的指针一般在使用时都是指向变量的,实际上可以声明指向某类函数的指针。

```
typedef void( * TFunction)() ;
Tfunction func ;
```

上述代码声明了一个指向返回值为空,参数也为空的指针变量 func。可以使用如下代码通过 func 直接调用函数。

```
func() ;
```

将命令分析器的类命名为 CG_CmdPool,首先需要声明一个命令结构体,存放相关资料。

```
struct SCmd
 {
     void    * fun ;                    //函数指针
     int     num ;                      //数字索引
     string  str ;                      //字符串索引
     string  desc ;                     //描述字符串
} ;
```

声明命令数组。

```
vector < SCmd >  m_cmdList ;
```

向命令分析器注册一条命令。

```
bool CG_CmdPool ::AddCmd(SCmd  * cmd)
 {
     m_cmdList. push_back( * cmd) ;
     return true ;
}
```


根据字符串索引查找命令,若成功,则返回命令,否则返回零。

```
SCmd  * CG_CmdPool ::GetCmdByStr(char  * str)
 {
     for(int i = 0 ; i < m_cmdList. size() ; i ++)
      {
          if(m_cmdList[i]. str == str)  return  &m_cmdList[i] ;
     }
     return NULL ;
}
```

根据数字索引查找命令，若成功，则返回命令，否则返回零。

```
SCmd * CG_CmdPool::GetCmdByNum(int num)
{
    for(int i = 0;i < m_cmdList.size();i ++)
    {
        if (m_cmdList[i].num == num) return &m_cmdList[i];
    }
    return NULL;
}
```

取得已注册的命令总数量。

```
int CG_CmdPool::GetCmdCount()
{
    return m_cmdList.size();
}
```

以下的代码为一个使用实例。

```
//test cmd pool
#include "../gamelib/g_platform.h"
#include "../gamelib/g_cmdpool.h"
//声明函数原型
typedef void(*TFunction)();
void fun1()
{
    Sys_Log("in function 1");
}
void fun2()
{
    Sys_Log("in function 2");
}
void fun3()
{
    Sys_Log("in function 3");
}
void TestCmdPool()
{
    SCmd cmd,*pCmd;
```

```
    CG_CmdPool pool;
    cmd.str = "fun1";
    cmd.fun = fun1;
    pool.AddCmd(&cmd);
    cmd.str = "fun2";
    cmd.fun = fun2;
    pool.AddCmd(&cmd);
    cmd.str = "fun3";
    cmd.fun = fun3;
    pool.AddCmd(&cmd);
    //执行名为 xxx 的函数
    pCmd = pool.GetCmdByStr("xxx");
    if(pCmd) ((TFunction)pCmd->fun)();
    //执行名为 fun2 的函数
    pCmd = pool.GetCmdByStr("fun2");
    if(pCmd) ((TFunction)pCmd->fun)();
    //执行名为 fun3 的函数
    pCmd = pool.GetCmdByStr("fun3");
    if(pCmd) ((TFunction)pCmd->fun)();
}
int main()
 {
    TestCmdPool();
    getchar();
    return 1;
}
```

运行代码后,屏幕显示如下:

```
in function 2
in function 3
```

表示第二和第三个函数被正确执行了,命令分析器工作正常。

8.3.3 物件模型

在游戏世界中存在许多物体,例如道具、玩家、NPC 等等,正是由于这些物体之间的相互作用,构成了一个丰富多彩的虚拟世界。如何定义这些物体之间的相互关系,是首先需要考虑的问题。

游戏中的物体类主要实现以下功能:

- 物体的统一创建与删除

- **物体的定时器**
- **物体的查找**

整体结构图如图 8.4 **所示。**

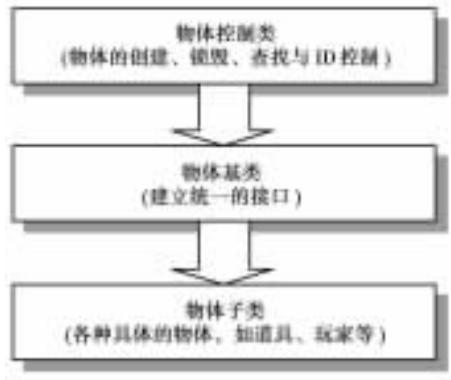


图 8.4

物体基类的实现代码如下：

```
//Object.h：interface for the CObject class.
#ifndef __COBJECT_H__
#define __COBJECT_H__
class CObject
 {
public：
      void SetUniqueId(long id)；
      long GetUniqueId()；
      virtual void HeartBeat()；
      CObject()；
      virtual ~CObject()；
private：
      long m_uid；                    //物体 ID
      long m_obType；                 //物体类型
}；
#endif
//Object.cpp：implementation of the CObject class.
#include "Object.h"
CObject：:CObject()
 {
}
CObject：:~CObject()
 {
```

```
}
void CObject::HeartBeat()
 {
}
//取得物体的 ID
long CObject::GetUniqueId()
 {
     return m_uid;
}
//设置物体的 ID
void CObject::SetUniqueId(long id)
 {
     m_uid = id;
}
//设置物体的类型
void CObject::SetObjectType(long type)
 {
     m_obType = type;
}
//取得物体的类型
long CObject::GetObjectType()
 {
     return m_obType;
}
```

有了物体的基类后,就可以考虑整个游戏世界的物体模型,一个简单的模型如图 8.5 所示。其中,玩家和 NPC 是从一个父类派生的。

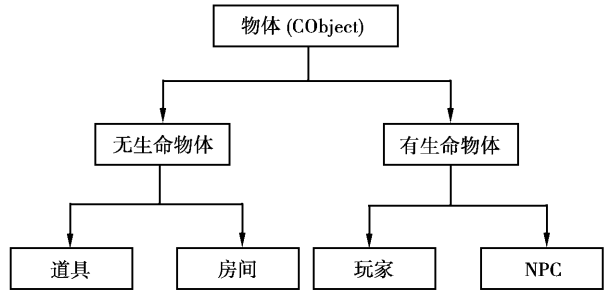


图 8.5

由于游戏中的所有物体都从一个基类派生,所以可以对物体进行统一管理。首先进行的是物体的创建,根据不同的物体类型,创建不同的实例,并返回统一的 CObject 指针。

```
//创建一个新物体
CObject * CMatrix::CreateObject(long type)
{
    CObject * ob = NULL;
    //根据物体类型创建实例
    switch(type)
    {
        //道具
        case OTYPE_ITEM:
            ob = new CItem;
            break;
        //玩家
        case OTYPE_USER:
            ob = new CUser;
            break;
    }
    if(ob)
    {
        //设置 ID
        ob->SetUniqueId(m_seq++);
        //添加到物体列表中
        m_obList.push_back(ob);
    }
    return ob;
}
```

物体的销毁也要使用统一的接口,避免内存泄漏等问题。

```
//销毁一个物体
void CMatrix::DestroyObject(CObject * ob)
{
    //从物体列表中清除
    m_obList.remove(ob);
    //释放内存
    delete ob;
}
```

提供查找物体的功能。

```
//根据 ID 查找物体
CObject * CMatrix::FindObject(long uid)
{
    CObject *ob = NULL;
    list<CObject *>::iterator it;
    for(it = m_obList.begin();it != m_obList.end();it ++)
    {
        if((*it)->GetUniqueId() == uid)
        {
            ob = (*it);
            break;
        }
    }
    return ob;
}
```

8.3.4 行走同步

行走是游戏服务器需要实现的重要功能之一。作为图形化的网络游戏,用户的绝大部分时间都处于行走状态。

最简单的实现方法是每次移动前向服务器提交移动请求,等服务器回应后再播放行走动画,这样客户端的状态始终是和服务器一致的,其实现过程如图 8.6 所示。

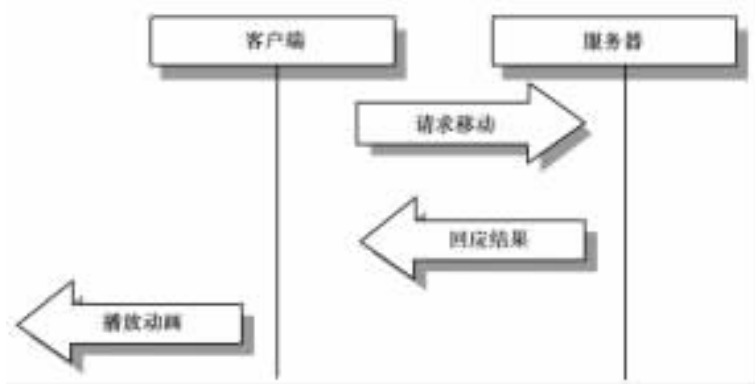
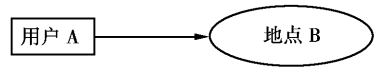


图 8.6

这样做的缺点是从用户按下鼠标到开始播放行走动画之间的时间间隔太长,其时间间隔等于网络往返时间加上服务器的计算时间。

改进的方法是客户端采用预测技术。当客户端提交移动请求后,并不是消极等待服务器的计算结果,而是根据客户端的本地数据进行预测,并马上播放行走动画。这样用户就不会感觉到停顿。

当用户 A 请求去地点 B 时，客户端根据本地数据发现 B 点没有人，因此直接播放 A 行走到 B 的动画。考虑这样一种情况，在同一时间有用户 C 也向 B 点移动，并且 C 的移动请求先被服务器收到。结果造成在用户 A 的客户端上看到自己和 C 同时处于 B 点，而在服务器上 A 被拒绝移动，因为 C 已经在 B 点。解决方法是客户端除了做预测外还做检验，当发现本地预测结果与服务器的计算结果不一致时，应该采用服务器的计算结果。



预测与校验的实现结构如图 8.7 所示。

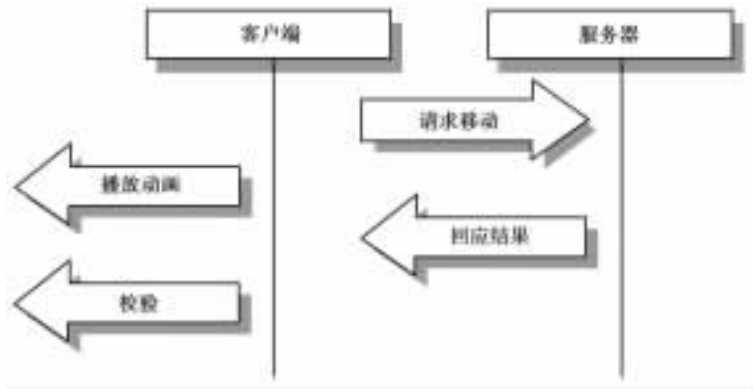


图 8.7

8.3.5 脚本

在网络游戏的开发过程中，有一些设定会被反复修改，例如属性计算公式、任务奖励等。如果直接采用 C ++ 编写，则经常需要重编译程序，增加了程序员的负担，同时也不利于企划对游戏进行调整。采用脚本则可以在不重新启动程序的情况下使设定直接生效。

脚本的执行速度比较慢，据一些资料显示，脚本的执行速度一般比直接用 C ++ 编写的代码慢 20 倍以上。

8.3.6 负载能力

对网络游戏来讲，服务器的负载能力是一个重要的指标。游戏服务器的负载能力受到硬件与软件两方面的限制。

硬件限制：

- CPU 的速度
- RAM 的容量
- 磁盘读取速度

软件限制：

- 操作系统的限制（如同时打开的文件数量等）
- 代码执行速度

- 是否支持多线程
- 是否支持分布式结构

玩家的指令从发出一直到服务器处理完指令并回应的这段时间称为命令响应时间。命令响应时间决定了游戏的流畅性,响应时间过长就会造成游戏不流畅,即玩家平时常说的服务器“卡”。因此,在确保游戏流畅的前提下,响应时间有一个最大值。在这个最大值下服务器所能承受的用户数量,就是服务器的最大负载能力。

命令响应时间 = 网络往返时间 + 服务器处理时间

其中,网络往返时间受到硬件的影响,无法从软件层解决。因此主要是想办法缩短服务器处理时间,服务器端的主函数一般如下:

```
while(①)
 {
    ①接收网络数据。
    ②处理玩家的网络命令。
    ③处理游戏世界中的所有物体的动作。
}
```

当玩家的数量增长时,每个处理循环所需的时间会相应增长,而且这种增长并不是线性的,一般会比线性增长高很多,因为人数增长时,带宽需求增长,内存使用增长,服务器需要处理的物体数量增长。以聊天系统为例,假设用户平均每秒发言 0.5 次,当 100 人在线时,平均每秒处理发言数量 = 100 * 0.5(发言数量) * 100(接收者数量) = 5 000 人次。当 200 人在线时,平均每秒处理发言数量 = 200 * 0.5(发言数量) * 200(接收者数量) = 20 000 人次。人数增长了 2 倍,但处理次数增长了 4 倍。

因此,通过优化代码可以使处理时间缩短,但只能使人数少量增长。早期的一些网络游戏都是用一个线程来运行一个游戏世界。即使服务器是多 CPU 的,单一游戏世界的用户数还是得不到提高,负载能力一般在 1 000 人以下。

单一线程结构如图 8.8 所示。

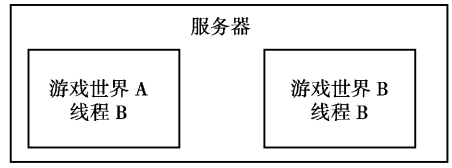


图 8.8

如果希望一个游戏世界能够容纳更多的用户,则需要采用分布式技术。既然单个线程的处理能力有限,那么就使用多个线程运行,这些线程可以分布在不同的服务器上。

分布式结构如图 8.9 所示。

将游戏中的地图切割成数块,每个服务器运行一张地图,这样就可以把玩家分散到各个子服务器中,用一台主控服务器协调地图切换等工作。理论上通过添加子服务器可以使用户数无限增长,但实际上最终的人数瓶颈在主控服务器上。

目前一些较新的网络游戏都使用了分布式结构,利用多台服务器运行一个游戏世界,负载能力一般在 3 000 人以上。

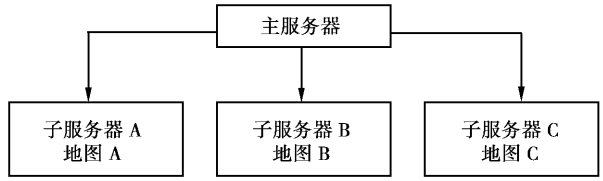


图 8.9

第9章 嵌入式表达式系统的设计与实现

对表达式的求解是建立在算符优先分析法的基础上的,有关于此分析法的详细描述请参考编译原理。下面将介绍表达式求解方法,它同样是建立在算符优先分析法的基础上。这个表达式是解释性的,可嵌入到任何应用程序中,不用修改和编译代码就可以替换或添加表达式,包含有算术运算、逻辑运算、变量、函数调用等功能,可以满足大部分人的要求,甚至可以在表达式的基础上建立一种简单的有条件和循环控制语句的程序。当然,由于笔者水平有限,可能还会有更多更巧妙的实现方法,这里仅供参考。

9.1 运算符的优先级别

任何表达式都可由操作数、运算符和界限符组成。通常操作数可以是常量、变量或者数字,这里的表达式中所有的数字都是操作数;运算符分为算术运算符、逻辑运算符和关系运算符;而界限符就是包括了左右括号和表达式结束标志。在后面的说明中,把运算符和界限符统称为算符。

我们都知道四则运算的基本规则为:先乘除,后加减;从左到右算;先算括号内,后算括号外。

由此可知,乘法和除法的优先级别比加法和减法都高,如果同时有相同优先级的乘法和除法,则先做前面的符号,这种规则称为左结合。所以,表达式

2 + 3 * 4 + 10 / (3 - 1) 的计算顺序应为:

2 + 3 * 4 + 10 / (3 - 1) = 2 + 12 + 10 / (3 - 1) = 14 + 10 / (3 - 1) = 14 + 10 / 2 = 14 + 5 = 19

假设有两个算符 a 和 b,可以对算符间的优先顺序定义如下:

a < b　表示 a 的优先级低于 b。

a = b　表示 a 的优先级与 b 相同。

a > b　表示 a 的优先级高于 b。

由"先乘除,后加减"的规则可以得出:* > +, * > -, / > +, / > -。

由"从左到右算"的规则可以得出:* > *, / > /, + > +, - > -。

由"先算括号内,后算括号外"的规则可以得出:括号内的优先级高于括号外的算符。

用"#"表示表达式的开始和结束标志,其优先级比任何算符都低。

表 9.1 定义了算符之间的优先关系。

表 9.1

	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	Error
)	>	>	>	>	Error	>	>
#	<	<	<	<	<	Error	=

如果出现‘）’连着‘（’、‘#’连着‘）’以及‘（’连着‘#’,就说明出现了语法错误。

9.2 简单表达式的求解

下面用算法优先分析法求解简单的四则运算表达式。

下面这段代码是简单的只包含加、减、乘、除和括号的表达式求解程序。程序中用了两个堆栈,一个是算符堆栈(stackOpr),另一个是操作数堆栈(stackNum)。程序计算的都是整型数字。

```
#include "ParseStr.h"
#include "SimpleExpr.h"
#include <stack>
using namespace std;
int SimpleExpression(char * lpExpr)
{
    //操作符堆栈,用于保存操作符
    stack<char> stackOpr;
    //操作数堆栈,用于保存操作数
    stack<int>  stackNum;
    char        *pCur, *pOpr, cOpr;
    int         iOpr, iNum1, iNum2;
    char        szSymbol[30];
    //操作符列表
    char *      szOpr = "+-*/()#";
    //操作符优先级别
    char        PRI[7][7] = {
        {'>','>','<','<','<','>','>'},
        {'>','>','<','<','<','>','>'},
        {'>','>','>','>','<','>','>'},
        {'>','>','>','>','<','>','>'},
        {'<','<','<','<','<','=','E'},
        {'>','>','>','>','E','>','>'},
        {'<','<','<','<','<','E','='}};
    //开始解析
    pCur = lpExpr;
    //把表达式开始标志压入操作符堆栈
    stackOpr.push('#');
    //表达式结束并且操作符堆栈里没有操作符的时候就结束分析
    while (*pCur!=0||stackOpr.top()!='#')
```

```
{
    cOpr = *pCur;
    //程序中把字符串结束标志表示成表达式结束标志
    if (cOpr == 0) cOpr = '#';
    //查找是否在操作符列表中
    pOpr = strchr(szOpr, cOpr);
    //如果不在列表中就表示是操作数
    if (pOpr == 0)
    {
        //取得一个 Symbol,在这个表达式中所有的操作数都默认为整数
        pCur = GetSymbol(pCur, szSymbol);
        //把操作数压入堆栈
        stackNum.push(atoi(szSymbol));
    } else
    {
        switch(PRI[strchr(szOpr,
            stackOpr.top()) - szOpr][pOpr - szOpr])
        {
        case '<': //小于
            //把操作符压入操作符堆栈
            stackOpr.push(cOpr);
            pCur ++;
            break;
        case '=': //等于
            //操作符出堆
            stackOpr.pop();
            pCur ++;
            break;
        case '>': //大于
            //从操作符堆栈中取出操作符
            iOpr = stackOpr.top();
            stackOpr.pop();
            //从操作数堆栈中取出第一个操作数
            iNum1 = stackNum.top();
            stackNum.pop();
            //从操作数堆栈中取出第二个操作数
            iNum2 = stackNum.top();
            stackNum.pop();
            //根据操作符计算结果
```

```
                switch（iOpr）
                 {
                case ' +' :iNum2 + = iNum1 ;break ;
                case ' - ' :iNum2 - = iNum1 ;break ;
                case ' * ' :iNum2 * = iNum1 ;break ;
                case ' /' :iNum2 / = iNum1 ;break ;
                }
                //把计算出来的结果压入操作数堆栈
                stackNum. push（iNum2）;
                break ;
            case 'E' :
                return - 1 ; // error
            }
        }
    }
    //返回结果
    return stackNum. top（）;
}
```

程序中，调用函数 GetSymbol 的定义如下：

```
char * GetSymbol（char * szExpr, char * szSymbol）;
```

其功能是在输入串 szExpr 中找出一个只包含字母、数字和下划线的符号并保存到 szSymbol 中，这个符号可能是数字也可能是常量、变量。函数返回除掉符号后的字符串位置，如执行如下语句：

```
char szSymbol[100] ;
char * p = GetSymbol（"970241 + 1", szSymbol）;
```

则 szSymbol 中保存的是取出来的符号“970241”，而 p 指向的内容为“+1”。

由上面的一段程序就可以求解简单表达式的值了。表 9.2 列出计算“9/（6 - 3）* 2”的计算过程，请注意里面的堆栈变化。

表 9.2

过程	操作符堆栈	操作数堆栈	剩余表达式
1	#		9/（6 - 3）* 2#
2	#	9	/（6 - 3）* 2#
3	# /	9	（6 - 3）* 2#
4	# / （	9	6 - 3）* 2#
5	# / （	9 6	- 3）* 2#

续表

过程	操作符堆栈	操作数堆栈	剩余表达式
6	# / (-	9 6	3) * 2#
7	# / (-	9 6 3	) * 2#
8	# / (	9 3	) * 2#
9	# /	9 3	* 2#
10	#	3	* 2#
11	# *	3	2#
12	# *	3 2	#
13	#	6	#

9.3 单目运算符

单目运算符是指一个算符只作用于一个操作数,如正号和负号。

上面提到的加号、减号、乘号和除号都是双目运算符,因为它们都作用于 2 个操作数。

单目运算符求解的过程和双目运算符一样,区别在于它们只作用于 1 个操作数。所以在求解过程中,双目运算符需要从操作数堆栈中取得 2 个操作数,而单目运算符只需要取出 1 个操作数。以下代码可以区分单目运算符和双目运算符:

```
opr = stackOpr. top();
if (IsSingleOpr(opr))
 {
     n1 = 0;
} else
 {
     n1 = stackNum. top();
     stackNum. pop();
}
n2 = stackNum. top();
stackNum. pop();
stackNum. push(Operate(n2, n1, opr));
stackOpr. pop();
```

但是,单目运算符正号正好和双目运算符加号相同,负号也有相同的情况,因此在程序中需正确识别是正号还是加号。

例如:

+1 +1	前一个为正号,后一个为加号
1 ++1	前一个为加号,后一个为正号
(1) +1	是加号

通过观察,可以整理出以下规则:如果“ + ”号前面是一个操作符并且不是“) ”就说明这个“ + ”号是正号,否则就是加号。下面的伪代码描述了如何识别正号和负号操作符:

```
设置当前运算符
if(上一个符号是算符)
 {
    if(当前算符是符号“ + ”或者“ - ”)
     {
    if(上一个算符是“ ) ”)
     {
    修改当前运算符为单目运算符正号或者负号
    }
  }
}
```

9.4 逻辑表达式

逻辑表达式用来判断表达式的内容是真还是假,只有 2 个返回值,真或者假。

这里需要增加一些逻辑运算符“ > ”、“ < ”、“ = ”、“ ! ”、“ & ”和“ | ”,分别表示大于、小于、等于、取反、逻辑与和逻辑或,如表 9.3 所示。

表 9.3

逻辑运算符	含义	“ 真 ”的例子	“ 假 ”的例子
>	大于	3 > 2	3 > 3
<	小于	2 < 3	5 < 1
=	等于	9 = 9	7 = 8
!	取反	! false	! true
&	逻辑与	true & true	true & false
|	逻辑或	true | false	false | false

在这些逻辑运算符中只有“ ! ”是单目运算符,其他都是双目运算符。因此只需把“ ! ”添加到单目运算符列表中,然后把所有运算符添加到运算符列表并根据运算优先级别扩大优先关系表即可,表 9.4 是扩大后的优先关系表。

表 9.4

	+	-	*	/	#	$	>	<	=	!	&	|	@
+	>	>	<	<	<	<	>	>	>	<	>	>	>
-	>	>	<	<	<	<	>	>	>	<	>	>	>
*	>	>	>	>	<	<	>	>	>	<	>	>	>
/	>	>	>	>	<	<	>	>	>	<	>	>	>
#	>	>	>	>	>	>	>	>	>	<	>	>	>
$	>	>	>	>	>	>	>	>	>	<	>	>	>
>	<	<	<	<	<	<	>	>	>	<	>	>	>
<	<	<	<	<	<	<	>	>	>	<	>	>	>
=	<	<	<	<	<	<	<	<	>	<	>	>	>
!	>	>	>	>	>	>	>	>	>	>	>	>	>
&	<	<	<	<	<	<	<	<	<	<	>	>	>
|	<	<	<	<	<	<	<	<	<	<	>	>	>
@	<	<	<	<	<	<	<	<	<	<	<	<	EE

表中,为了避免混淆,用‘ # ’代替单目运算符‘ 正号 ”,用‘ $ ’代替单目运算符‘ 负号 ”,用“ @ ’表示表达式结束符。

现在需要在程序中用某一种方式表示‘ 逻辑真 ’和‘ 逻辑假 ”。为了和算术表达式结合得更好以及更简单地实现,这里选择用数字 0 表示逻辑假,其他数字表示逻辑真。于是,!(2 +3)返回的数值是 0,因为!(2 +3) = !5,5 在逻辑表示中为‘ 真 ”,所以!5 为‘ 假 ”,返回数值 0。

9.5 变量

在使用表达式的过程中,需要控制一些表达式中项目的值以方便计算。如在游戏中计算某一个人物的攻击力:

攻击力 = 人物基本攻击力 + 道具附加攻击力

可以用下面的表达式表示:

ATK = BASE_ATK + ITEM_ATK

由于人物基本攻击力和道具附加攻击力对于不同的人物不同的道具都是不同的,惟独这个公式对所有的人物都是相同的。这种情况下可以把公式提取出来,在计算的时候根据表达式提供的项目(如:BASE_ATK)提供数值给表达式计算。

例如,现在有 2 个 NPC,分别为 NPC1,NPC2,其基本属性如表 9.5 所示:

表 9.5

	NPC1	NPC2
基本攻击力(BASE_ATK)	60	20
道具附加攻击力(ITEM_ATK)	30	- 4

那么，根据公式“ ATK = BASE_ATK + ITEM_ATK ”，可以计算出 NPC1 的攻击力为 90，而 NPC2 的攻击力为 16。

要实现在表达式计算的过程中实时地取得游戏里的数据，就需要一个表达式与游戏程序的接口。下面详细地介绍其实现方法。

一个表达式由操作数和操作符组成。在简单表达式的求解中操作数由数字组成，因此读入表达式时如果操作数不是数字，那么表达式错误。但是在有变量的表达式中，操作数由常量和变量组成，常量就是以前用的数字，而变量则是其他的操作符号。所以在解释到不认识的符号时需要检查这个数字是否是变量，如果是变量就需要返回变量所表示的值。

为了使程序能正确区分操作数、常量以及变量，这里定义变量是由字母、数字和下划线组成，并且首字母必须是字母或者下划线。

但是，当找出一个非常量的操作数时，还需要知道这个操作数符号是否就是所定义的变量。如在表达式

ATK = BASE_ATK + ITEMATK

中，“ ATK ”和“ BASE_ATK ”是需要的变量，而“ ITEMATK ”则是误写的变量。在表达式应用过程中，错写的情况是不可避免的。在这种情况下，需要正确识别变量，并能发现变量存在的错误。在实践中，可以任意使用以下 3 种方法：

①在表达式的初始化时就创建一张变量表，每次找到非常量的操作数就去查找变量表。如果该变量存在就通知主程序并返回变量的值，如果不存在就说明出现了错误。

②定义一个 Callback 函数，在找到非常量的操作数时就调用该函数处理，根据 Callback 函数的返回值来确定是否是变量值。

Callback 函数的实现如下：

```
bool ExprVariableCallback(char * lpszVarName, double *pNum)
{
    ……
    else if (strcmp(lpszVarName, "BASE_ATK") == 0)
    {
        ……
        *pNum = …;
        ……
        return true;
    } else if (strcmp(lpszVarName, “ ITEM_ATK ”) == 0)
    ……
    return false;
}
```

表达式解释的部分程序：

```
if (IsConst(lpszSymbol))   //如果是常量
{
    ……
```

```
        }else    //如果是“非常量的操作数”
         {
             if(!ExprVariableCallback(lpszSymbol, &fNum))
              {
                  //不认识的符号错误
             }
    }
```

③把表达式解释程序放在类中,并定义一个在找到“非常量的操作数”时调用的虚函数。在有不同的表达式求解需求时可以定义不同的子类以实现对不同的变量的解释。

这里,表达式解释程序选择使用第三种方法。与定义 Callback 函数类似,有一个返回 bool 的表示是否有对该符号的解释。在该类的定义中有如下定义:

```
    ……
    virtual bool ExprVariable(char *szCallName, double *var);
    ……
```

由虚函数的特性,只要在解释的时候碰到变量就调用这个函数,所有的子类就可以在这个函数中实现变量的解释。

9.6 函数调用

函数通常的表现形式都是:

符号(参数列表)

如:

Fun(a, b, c)

与变量相比,函数的表现上多了参数列表和包含参数的一对括号,但它们在解释的时候是类似的。同样,在找到“非常量的操作数”时,如果这个符号后面带有括号,就说明这是一个函数调用,否则就是变量。在找到函数后,表达式分析程序需要找到这个函数的所有参数并把函数名字和参数传给应用程序,应用程序根据表达式的名字和参数进行相应的处理后返回一个数值给表达式分析程序。

因为函数和变量的解释过程类似,所以其实现也可以采用以上 3 种方法,只是多了参数传递的过程。因此在定义的函数中也需要添加一些用来表示调用函数的参数。下面是添加好的 Callback 函数:

```
    bool ExprVariableCallback(char * lpszVarName, int iParmCnt, double *pParms, double *pNum)
```

其中,添加了 2 个参数,iParmCnt 表示参数的个数,pParms 表示保存参数数值的指针。在函数内部需要根据参数个数来取参数数值。

变量的解释过程和函数类似,但是变量比较简单。事实上,可以把变量看成一个没有参数的函数。如 BASE_ATK 可以认为是表示基本攻击力的变量,同样也可以认为它是一个取得基本攻击力的不需要参数的函数。这样就在没有增加任何东西的基础上去掉变量这个定义,使表达式更加单一。所以,函数在有参数的时候就是带有包含参数列表的一对括号,在没有参数的时候这对括号是可有可无的。也就是说 BASE_ATK 与 BASE_ATK() 的意义是相同的。

9.7 参数传递

函数的参数列表由逗号分隔的一些参数组成,参数可以是一个完整的表达式。在调用函数的时候,需要把所有参数的具体值求出来。也就是说参数列表中所有表达式都必须在函数调用之前计算。

由于在函数调用的时候还需要计算子表达式,这里引入现场的概念。每次在计算函数的时候把计算当前表达式的数据都保存下来,包括操作数堆栈、操作符堆栈以及函数名。下面再定义几个堆栈,如表 9.6 所示。

表 9.6

stack < int >	_SceneStack	保存现场用,记录操作数和操作符个数
stack < double >	_NumTotalStack	保存操作数
stack < char >	_OprTotalStack	保存操作符
stack < string >	_CallStack	保存函数名

定义一个保存现场的函数如下:

```
void  ExpressionSaveScene()
 {
     //保存操作数堆栈
     _SceneStack.push(_NumStack.size());//操作数个数
     //把当前操作数堆栈内容转移到存盘堆栈中
     while(!_NumStack.empty())
      {
          _NumTotalStack.push(_NumStack.top());
          _NumStack.pop();
     }
     //保存操作符堆栈
     _SceneStack.push(_OprStack.size());//操作符个数
     //把当前操作符堆栈内容转移到存盘堆栈中
     while(!_OprStack.empty())
      {
          _OprTotalStack.push(_OprStack.top());
```

```
            _OprStack. pop();
        }
        //保存当前的函数名
        _CallStack. push(_Symbol);
    }
```

定义一个取出现场的函数：

```
    void  CExpression::LoadScene()
    {
        int i;
        //把当前堆栈的内容清空
        while(!_NumStack. empty())    _NumStack. pop();
        while(!_OprStack. empty())    _OprStack. pop();
        //取出操作符堆栈
        i = _SceneStack. top(); //操作符个数
        while(i)
        {
            //把操作符转移到当前计算堆栈
            _OprStack. push(_OprTotalStack. top());
            _OprTotalStack. pop();
            i--;
        }
        _SceneStack. pop();
        //取出操作数堆栈
        i = _SceneStack. top(); //操作数个数
        while(i)
        {
            //把操作数转移到当前计算堆栈
            _NumStack. push(_NumTotalStack. top());
            _NumTotalStack. pop();
            i--;
        }
        _SceneStack. pop();
        //取出函数名
        strcpy(_Symbol, _CallStack. top(). c_str());
        _CallStack. pop();
    }
```

每次在监测到需要函数调用的时候就保存现场并清空当前计算堆栈,开始进行子表达式的计算。

由于一个函数中可能的参数个数是不定的,参数间用逗号隔开,为了方便计算多个参数的情况,这里引入“逗号表达式”。逗号表达式就是一系列用逗号隔开的表达式。由于其中子表达式的个数是不定的,再引入一个堆栈用来保存各个子表达式计算的结果。其定义如下:

```
stack < double > _ParmStack;
```

同样,由于函数和表达式可以嵌套,所以需要一个堆栈用来保存上一层的逗号表达式的值。这里还是保存在_ParmStack 中,同时引入一个堆栈用来记录上一层的表达式的个数。其定义如下:

```
stack < int > _CallParmCntStack;
```

这样,在一个逗号表达式计算出结果之前所有的子表达式的结果都保存在堆栈中。

对于一个普通的逗号表达式而言,最后返回给表达式的值是最后一个子表达式的值。如逗号表达式:

$$1*3,3+4,6/5,9-3$$

得到的值为 9 - 3 = 6。

对于一个函数的参数而言,只需要把计算得到的堆栈中的值作为函数的参数传给函数处理即可。例如函数:

$$fun(1*3,3+4,6/5,9-3)$$

逗号表达式计算到最后堆栈中的值为 3,7,1.2,6。调用 ExprVariableCallback 时,lpszVarName 的值为“fun”,iParmCnt 的值为 4,而数组 pParms 里面的值就是 3,7,1.2,6。

事实上,一个简单的没有包含逗号的表达式也可以认为是只有一个子项的表达式,所以对一个复杂的表达式可以从逗号表达式开始计算。

本书的配套光盘/补充与提高/第 9 章补充——一个完整的实现中,我们给出了一个完整的实现过程,这个程序包含一个头文件和一个源文件。

第10章 回合制战斗系统的设计与实现

10.1 战斗系统的功能结构

战斗系统的功能结构如图 10.1 所示。

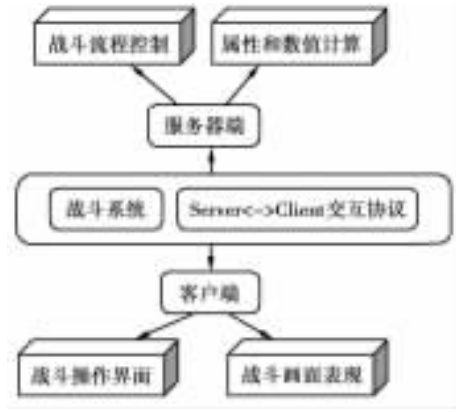


图 10.1

由功能结构图可知战斗系统主要分 Server 和 Client 两块,各自分管不同的功能。

(1) Server 的功能

①流程控制:Server 端通过踩地雷的方式触发战斗后,通知客户端进入战斗状态,然后控制战斗中每个回合的进行,直到战斗结束。

②属性数值计算:玩家角色的属性通常有生命点数、攻击力、防御力、速度等,在战斗中会通过这些属性来计算战斗伤害、命中率等数值。

(2) Client 的功能

①操作界面:玩家需要战斗菜单来选择角色行为,每回合操作一次,同时界面上还要显示一些角色状态信息如生命值、级别等等。

②画面表现:当玩家控制的角色攻击敌人,使用法术和道具时,画面上都会有相应的表现以及特效和数值显示等。

10.2 回合制战斗流程

在回合制战斗中,战斗流程是按战斗开始,回合开始,回合结束,回合开始,回合结束……战斗结束这样的步骤进行的,如图 10.2 所示。

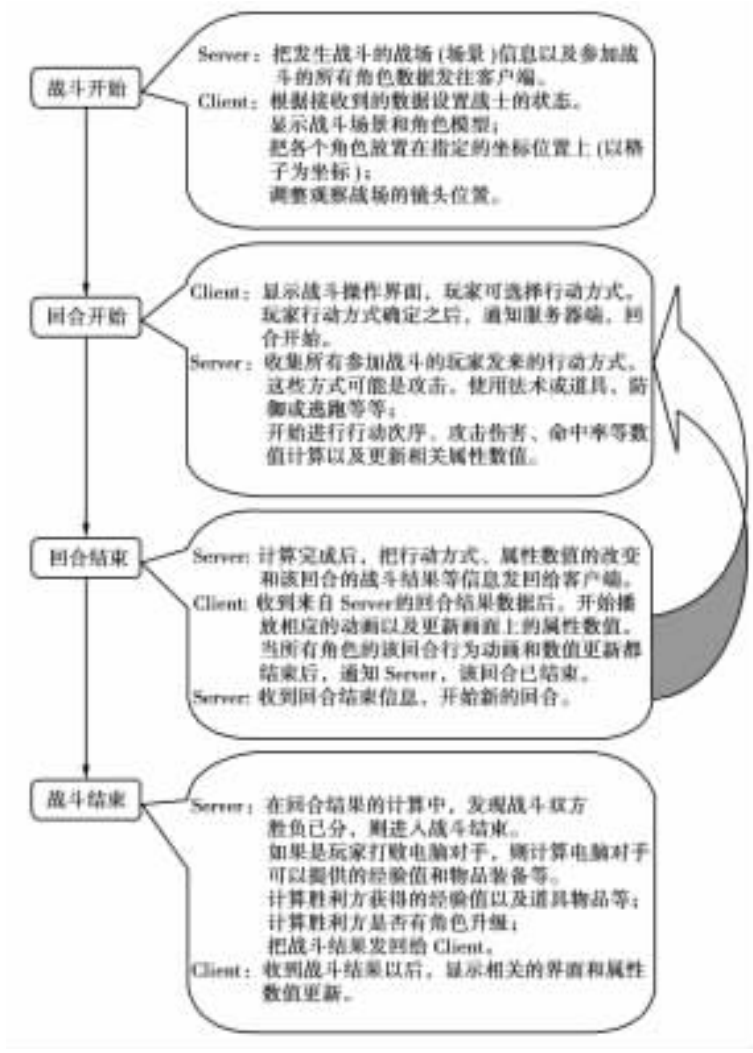


图 10.2

10.3 Server **与** Client **之间的交互协议**

战斗中主要的交互协议有：

(1) MSG_GWS2CLIENT_BATTLE_BEGIN **战斗开始**

消息格式：

byte 战斗类型
string 战斗场景 ID(名字)
byte 参加战斗的战士类型的数量
byte 当前类型的战士数量
long 当前战士的数据:ID
string 名字
long,long 坐标位置 X,Y(以格子为单位)
long 模型 ID
long 贴图 ID
short 级别 LEV
short 生命值 HP
short 最大生命值 MAX_HP
short 最大魔力值 MAX_MP
byte 身上所装备的数量
long 当前装备的数据 :ID
……(循环读取)
……(循环读取)
……(循环读取)

说明:战斗类型指玩家与电脑战斗或者玩家之间的 PK 等类型;战士类型是指战士在游戏中可以设置为战斗、支援、等待等状态,需要把这些状态发送给客户端,每个战士身上附有哪些道具也要发送给客户端,以便显示出佩戴好装备的战士模型。客户端收到此消息后,显示战场场景以及把战士放在指定的位置,并且显示战斗的操作界面。

(2) MSG_ GWS2CLIENT _ROUND_BEGIN 回合开始

消息格式:

short 回合计数
byte 角色属性数值更新的数量
long 角色 ID
byte 属性类型编号
byte 属性编号
long 属性数值
……(循环读取)
……(循环读取)

说明:在回合开始的时候,客户端会接收到一段角色属性数值更新的数据,因为在战斗中经常会有些技能(法术)可以造成持续多个回合的效果,比如某个角色中毒了之后,每个回合生命值都会减一定数值。那么像这样的角色属性更新就必须放在回合开始的消息数据里,以便客户端作相应的画面显示。属性数值的更新使用属性类型编号 + 属性编号 + 实际属性数值的格式来实现。通过约定这样的格式,就可以按照统一的方式处理一个角色有多种属性数值同时需要更新的情况。客户端在收到这个消息之后,就会弹出战斗操作选单,玩家此时可以进行新回合的行动选择。

(3) MSG_CLIENT2GWS_BATTLE_CMD 战斗行动指令

消息格式：

byte	行动类型
long	行动子类型
long	行动参数
byte	参加行动的战士(角色)数量
long	战士 ID ……
byte	行动针对的目标数量
long	目标 ID ……
byte	第一个目标所在的格子坐标 X
byte	第一个目标所在的格子坐标 Y

说明：当玩家在战斗操作界面上选择了行动方式之后，就会发送该消息给 Server。行动类型有如下几种：

FIGHTACTION_USESKILL	//使用技能
FIGHTACTION_USEITEM	//使用道具
FIGHTACTION_FLEE	//逃跑
FIGHTACTION_DEFENCE	//防御

下面的结构体数据专门描述回合之间玩家所采取的行动：

```
struct SFightAction
 {
     bool    bValid；              //是否有效
     long    lOrder；              //行动次序的权值
     BYTE    btActionType；        //动作类型
     long    lModeNo；             //子类型（攻击方式、技能 ID、道具 ID 等）
     long    lParam；              //附加参数（如技能级别）
     BYTE    btFighterCnt；                          //行动者数量
     long    lFighterID[MAX_ACTION_FIGHTER]；//行动者 ID 列表
     BYTE    btTargetCnt；                           //目标数量
     long    lTargetID[MAX_ACTION_TARGET]；  //目标 ID 列表
     BYTE    btTargetGridX；                         //目标格子 X
     BYTE    btTargetGridY；                         //目标格子 Y
 }；
```

这个结构体在 Client 和 Server 上都会使用。当 Server 端收到此消息后，就会把玩家的行动方式数据存放在该结构体中，多个玩家包括电脑对手的行动就会形成一个此结构体的数组以供下一步计算时使用。

通常情况下都是玩家正在操作的角色选定一个目标来发起行动,但有时候会出现同时攻击多个目标的技能或者多人同时攻击一个目标,所以在消息的格式定义里预留位置,可以处理多个行动发起者和多个目标的情况。另外,有些技能是按格子范围攻击的,因此也把第一个目标所在的格子坐标发送给 Server。当客户端发出此消息后,便不再接受该玩家的其他操作,而是等待同一场战斗中的其他玩家操作完成。

(4) MSG_ GWS2CLIENT _ROUND_END 回合结束

消息格式:

```
byte        行动类型
long        行动子类型
long        行动参数
byte        参加行动的战士(角色)数量
long          战士 ID
              ……
byte        行动针对的目标数量
long          目标 ID
              ……
byte        角色属性数值更新的数量
long        角色 ID
byte          属性类型编号
byte          属性编号
long            属性数值
                ……(循环读取)
              ……(循环读取)
            ……(循环读取)
```

说明:此消息的格式实际上就是行动描述 + 行动结果 + 行动描述 + 行动结果,反复循环,直到该回合中所有的行动数据都读取完毕。其中,行动描述部分和 MSG_CLIENT2GWS_BATTLE_CMD 一样,只是少了第一个目标的坐标数据这一项。而行动结果则是针对行动中涉及的战士的属性更新列表,属性更新的数据格式与消息“回合开始”里的一样。客户端收到消息后,把其内容复制一份,然后按照步骤,每次解析出一个行动描述加上该行动的结果,依照读出的数据来做画面表现。

例如,战斗双方为:1 战士 A,2 战士 B 战士 C。在某回合中,A 使用技能 01 攻击目标 B,对 B 造成伤害;B 选择使用道具血瓶给自己补血;C 选择使用技能 02 攻击目标 A,对 A 造成伤害,并且使 A 处于混乱状态。

经过 Server 计算后,就会把如下形式的回合结果发给 Client。

```
行动 1 :    行动类型        ->    使用技能
            子类型          ->    1(技能编号)
            行动参数        ->    1(技能级别)
            行动者数量      ->    1
            行动者          ->    A
```

目标数量 -> 1
目标 -> B
行动 1 结果 :结果 1
战士 ID -> B
属性更新数量 -> 1
属性类型 -> 扩展属性(扩展属性指 HP、MP、攻击力等等)
属性 ID -> 6(属性 HP 的 ID)
属性数值 -> -30(B 的 HP 减少了 30)
行动 2: 行动类型 -> 使用道具
子类型 -> 103(血瓶的道具编号)
行动参数 -> 0
行动者数量 -> 1
行动者 -> B
目标数量 -> 1
目标 -> B(B 对自己使用血瓶,所以行动者 = 目标)
行动 2 结果 :结果 1
战士 ID -> B
属性更新数量 -> 1
属性类型 -> 扩展属性
属性 ID -> 6(属性 HP 的 ID)
属性数值 -> +100(B 的 HP 使用血瓶后增加了 100)
行动 3: 行动类型 -> 使用技能
子类型 -> 2(技能编号)
行动参数 -> 0
行动者数量 -> 1
行动者 -> C
目标数量 -> 1
目标 -> A
行动 3 结果 :结果 1
战士 ID -> A
属性更新数量 -> 2
属性类型 -> 扩展属性
属性 ID -> 6(属性 HP 的 ID)
属性数值 -> -20(A 的 HP 减少了 20)
属性类型 -> 战斗状态 (战斗状态指失明、混乱、迷惑等)
属性 ID -> 4(状态混乱的 ID)
属性数值 -> 1(表示设置为该状态, 0 表示清除该状态)

客户端接收到如上的回合结果,就会按照行动 1、行动 2、行动 3 的顺序来进行画面表现。

玩家会看到战士 A 跑到战士 B 的面前,做攻击动作,打了 B 一下, B 头上出现 HP-30 的漂浮文字;然后 A 跑回自己原位, B 做了一个使用道具的动作,头上出现血瓶的图样以及 HP +100 的漂浮文字;紧接着 C 使用的技能 02 是一个远距离攻击的技能,C 做出一个使用法术的动作,然后 A 身上出现被攻击的特效,显示 HP-20 的漂浮文字,同时头上还出现"混乱"状态的图标。所有行动的动画都播放完成后,该回合结束,客户端通知服务器,可以开始新的回合。

(5) MSG_CLIENT2GWS _ROUND_END 回合结束

消息格式:

无其他内容

说明:这条消息用来通知服务器端,当前回合所有行动的动画都已播放完成,可以进行新的回合了。服务器端只有接收到参加战斗的所有玩家的此消息,才会继续新的回合。

(6) MSG_GWS2CLIENT_ROUND_TIMER 回合操作计时

消息格式:

short TimeCount

说明:玩家在回合开始的时候,用来选择行动方式的时间是有限的,必须在指定的时间内完成操作,否则就算是本回合无任何动作。一般,时间可以限制在 30s,在此时间内, Server 就通过此消息,不断提示 Client 剩余的时间。

(7) MSG_GWS2CLIENT_BATTLE_END 战斗结束

消息格式:

byte 战斗结果(胜利、失败或逃跑)
byte 战士总数量
long 战士 ID
byte 获得奖励的数量(也可以是损失的数量)
byte 获得奖励的属性类型(如经验值属性的编号)
long 属性数值
……(循环读取)
byte 获得道具的数量
long 道具编号
……(循环读取)
……(循环读取)

说明:战斗胜负已分的情况下,客户端收到此消息,并且弹出一个战斗结果的界面,上面显示玩家在这次战斗中获得的经验值以及物品装备等信息。消息中包含了参加战斗的每个战士获得奖励的描述,可以根据需要只显示玩家自身的,或玩家所在一方所有战士的,甚至显示敌方的获得奖励的情况。在收到此消息后,客户端还会收到退出战斗而进入场景中自由行走的状态切换的消息。一场战斗就此结束了。

10.4 角色属性和数值

一个网络游戏的角色通常具有丰富的属性类别和变化,这些属性中多数又会为战斗而服务,毕竟多数网络游戏中,战斗系统都是重头戏,是游戏最主要的玩法之一。因此游戏策划在

设定属性数值系统的时候，通常也是从战斗系统的需要出发，下面看看在本实例中，属性数值是如何在战斗中发挥作用的。

角色有基本属性和扩展属性之分。

基本属性：

```
ATTR_STR        1    //力量
ATTR_CON        2    //体质
ATTR_DEX        3    //敏捷
ATTR_INTE       4    //智力
ATTR_KNOW       5    //知识
```

基本扩展属性：

```
ATTR_PATP       10   //攻击力
ATTR_PDFP       11   //防御力
ATTR_SPEED      12   //速度
ATTR_MATP       13   //魔法攻击力
ATTR_MDFP       14   //魔法防御力
ATTR_MAX_HP     7    //最大生命
ATTR_MAX_MP     9    //最大魔法点
```

这些属性的计算规则如下：

规则一　实际基本属性 = 基本属性 + 道具对属性的影响 + 技能对属性的影响

说明：角色当前的基本属性通常会受到身上所穿戴的装备或者学会的技能的影响，比如角色本来力量属性为 6 点，一件盔甲可以增加力量 10 点，那么穿上这件盔甲实际的力量点数就是 16 点；又比如有一种技能叫“力大无穷”，学会之后可以增加力量 20 点，那么实际力量点数就变成了 36 点。而装备是可以卸下的，技能也是可以被遗忘的，因此在保存角色属性数据的时候，必须把属性组成里的每个成分都保存下来。

规则二　基本扩展属性 = （实际基本属性）+ - / *（实际基本属性）

说明：所谓扩展属性就是攻击力、防御力之类，它们是用实际基本属性计算出来的，比如攻击力 = 力量 x3 + 体质 x2 + 智力 x1。当然运算法则各种各样。这些属性往往直接参加战斗中攻击伤害、命中率等数值的计算，比如伤害 = 攻击方攻击力 - 目标的防御力。当然实际的公式比这个要复杂得多。当角色的实际基本属性发生变化时，扩展属性就会根据公式重新计算，例如角色升级、穿戴装备、学会技能等情况下扩展属性就必须重新计算。但扩展属性也要分基本扩展属性和实际扩展属性，因为装备和技能也可能会直接影响扩展属性。

规则三　实际扩展属性 = 基本扩展属性 + 道具对属性影响 + 技能对属性影响

说明：与对基本属性的影响相类似，实际扩展属性也可以受到道具和技能的影响。比如一把黄金刀可以增加攻击力 10 点，但如果同一个道具既对基本属性有影响，又对扩展属性有影响，计算过程就会变得复杂。因为扩展属性本身就是从基本属性计算而来。

在客户端的角色属性界面里显示的是实际基本属性和实际扩展属性。

下面谈谈道具系统对属性的影响。

道具一般可以分两类 :装备类和使用类。

装备类道具对属性的影响有以下 3 种情况:

① 修改基本属性,引发扩展属性的重新计算。

② 直接修改扩展属性。

③ ①,②并存。

装备道具影响 = ① + ②,而顺序必须保证是先①后②。

使用类道具对属性的影响有:

①修改基本属性,引发扩展属性的重新计算。

为了防止此类道具在游戏中大量出现(原因可能是出现复制的 BUG 或安排数量不合理)造成基本属性很快提升,可以限制此类道具的影响总点数。

例如,某种白色药品,用一个加一点力量,那么基本属性实际构成为:

力量 = 初始基本点 + 点数分配 + 药品影响

药品影响是一个有上限的值 ,比如 10,当玩家使用道具时,如果超过 10,会提示力量点数受药品的影响已经到达上限,使用无效果。

②使用类道具不可以影响扩展属性。

因为当玩家升级后,扩展属性必须重新计算。和装备类道具不同,使用类道具用完就消失了,没有一个道具使用记录来支持实现扩展属性 = 基本扩展属性 + 道具影响。

回合制战斗中不允许使用类道具来改变基本属性,所以玩家在战斗中点击选择道具使用界面上的此类物品无效。客户端做一次检查,服务器端也会做检查。

技能对属性的影响:

技能也分两类 :被动类和使用类。被动类技能是学会了之后,就会一直发挥作用,而没有使用的概念,这类技能对属性的影响和装备类道具相似。技能可以学会,也可以放弃 ;使用类技能不可修改基本属性,但可以影响扩展属性,因为只限于战斗中的一定回合数,并不是永久性的改变,这种对属性的影响会记录在战斗的临时数据里。

10.5 运用表达式(公式)系统来计算

策划是游戏属性和数值系统的设计者。在设计和调整的过程中经常需要修改规则和计算公式,如果在设计游戏程序的时候把计算公式写死在代码里,就会出现频繁改动代码的情况,有时候甚至一天要改十多次。这样做效率非常低,所以游戏中需要的规则和公式计算应该尽可能提取到程序之外,用一种脚本形式保存起来,修改脚本并不需要修改任何代码,这样就可以方便策划来设计和调整属性数值系统。

关于可嵌入自定义函数的表达式系统,在第三章已经详细地介绍了它的实现方法,现在来看在战斗系统中是如何运用它的。

例如 :攻击伤害 HARM = ATP(1) x 3.0 - DFP(2) x 1.0

这个公式应用的场合是一个战士攻击另外一个战士。ATP 表示攻击力, ATP(1)表示攻击方的攻击力 ;DFP 表示防御力,DFP(1)表示被攻击方的防御力。

在表达式系统中，解析字符串 ATP 以及 DFP 的时候，会调用指定的回调函数，然后通过这个函数把字符串对应的数值传给表达式系统，最终便可以得到公式的计算结果。

在进入表达式计算之前，需要建立一个表达式计算的环境。这里用如下一些全局的变量来保存战斗中计算表达式所需的数据，这些数据就构成了计算环境。

CBattleField *	EX_BATTLEFIELD	//当前的战场
CCreature *	EX_FIGHTER[2]	//当前行动方和目标方
CFighterTeam *	EX_TEAM[2]	//当前战斗的双方
SCreSkill *	EX_SKILL	//当前使用的技能
SFStat *	EX_STAT	//当前处理的战斗状态
SItem *	EX_ITEM	//当前被使用的道具

解析自定义函数的回调函数如下：

```
szCallName  表达式中出现的自定义函数名
iParamCnt   函数的参数个数
pParam      参数数值列表
var           用来返回自定义函数调用结果的数值
bool CALLBACK_EXPR(CExpression * pExp , const char * szCallName,
                        const int iParmCnt, const double * pParms, double * var)
{
    if(strcmp(szCallName , "PATP") == 0)  //物理攻击力
     {
       if(iParmCnt != ) goto exp_err1 ;
       int p1 = pParms[0] - 1 ;
       // = 0 表示行动方, = 1 表示目标方
        * var = EX_FIGHTER[p1] -> GetCurAttr(ATTR_PATP) ; //返回指定方的攻击力属性
    }
    else if(strcmp(szCallName , "PDFP") == 0) //物理防御力
     {
       if(iParmCnt != ) goto exp_err1 ;
       int p1 = pParms[0] - 1 ;
        * var = EX_FIGHTER[p1] -> GetCurAttr(ATTR_PDFP) ; //返回指定方的物理防御力
    }
    else if(strcmp(szCallName , "MATP") == 0) //魔法攻击力
     {
       if(iParmCnt != ) goto exp_err1 ;
       int p1 = pParms[0] - 1 ;
        * var = EX_FIGHTER[p1] -> GetCurAttr(ATTR_MATP) ; //返回指定方的魔法攻击力
    }
    else if(strcmp(szCallName , "STR") == 0) //返回当前战士的力量点数
```

```
{
    *var = EX_FIGHTER[0]->GetAttr(ATTR_STR);
}
else if(strcmp(szCallName,"SPEED")==0) //返回当前战士的速度点数
{
    if(iParmCnt!=) goto exp_err1;
    int p1 = pParms[0] - 1;
    *var = EX_FIGHTER[p1]->GetCurAttr(ATTR_SPEED);
}
else if(strcmp(szCallName,"LEV")==0) //当前战士的级别
{
    if(iParmCnt!=) goto exp_err1;
    int p1 = pParms[0] - 1;
    *var = EX_FIGHTER[p1]->GetAttr(ATTR_LEV);
}
else if(strcmp(szCallName,"MAX_HP")==0)//当前战士的最大生命值
{
    if(iParmCnt!=) goto exp_err1;
    int p1 = pParms[0] - 1;
    *var = EX_FIGHTER[p1]->GetCurAttr(ATTR_MAX_HP);
}
else if(strcmp(szCallName,"CUR_HP")==0)//当前战士的当前生命值
{
    if(iParmCnt!=) goto exp_err1;
    int p1 = pParms[0] - 1;
    *var = EX_FIGHTER[p1]->GetCurAttr(ATTR_HP);
}
else if(strcmp(szCallName,"SLEV")==0)//当前使用的技能的级别
{
    *var = EX_SKILL->nCurLev;
}
else if(strcmp(szCallName,"TEAM_COUNT")==0)//两队人战斗时,指定方的战士人数
{
    if(iParmCnt!=) goto exp_err1;
    int p1 = pParms[0] - 1;
    int nCnt = EX_TEAM[p1]->GetMemberCnt();
    *var = nCnt;
}
```

```
        else if(strcmp(szCallName,"TEAM_EXP_OFFER") == 0) //指定队伍提供的经验值
        {
            if(iParmCnt != ) goto exp_err1 ;
            int p1 = pParms[0] - 1 ;
            * var = EX_TEAM[p1] -> GetExpOffer() ;
        }
          ...
        return false ;
    }
```

根据这样的实现方法,就可以让策划来设计所需的各种公式,填写在文本文件中,不用修改程序,就可以调整公式。在本实例中,表达式设计的规则如下:

<基本概念>

当前主动方|攻击方|自己 =1
当前被动方|被攻击方 =2
主动方队伍 =1
被动方队伍 =2

<基本函数>(大写)

物理攻击力 :PATP(n)
物理防御力 :PDFP(n)
魔法攻击力 :MATP(n)
魔法防御力 :MDFP(n)
属性相克率 :ELEM_EFFR(n ,n)
速度 :SPEED(n)
等级 :LEV(n)
队伍人数 :TEAM_COUNT(n)
队伍速度和 :TEAM_SUM_SPEED(n)
队伍等级和 :TEAM_SUM_LEV(n)
队伍经验值提供 :TEAM_EXP_OFFER(n)
……
最大技能级别 :MAX_SLEV
技能级别(用于填技能表) :SLEV
引起状态的技能级别(用于填状态表) :STLEV
……

<数学函数>(小写)

开方 :sqrt
随机 :rand(n1 ,n2) (n1 <= 取值 < n2)

公式格式为:

[名字] //中文注释

变量 1 = 表达式 1
变量 2 = 表达式 2
……
名字 = 表达式 n
[END]
最后一个变量名应该等于“名字”。
下面的程序代码提供了一些示例：

```
[PAT_HARM]  //物理攻击伤害
  PAT_HARM = PATP(1) * 3.5 - PDFP(2) * 1.5) * ELEM_EFFR(1,2)
[END]
[EXP_OFFER] //个体经验值提供
  EXP_OFFER = (LEV(1) + 1) * (LEV(1) + 1)/2 + 1
[END]
[EXP_GET] //个体经验值获得
  EXP_GET = TEAM_EXP_OFFER(2) * sqrt(LEV(1) / TEAM_SUM_LEV(1))
[END]
[ACT_ORDER] //战斗出手顺序
  ACT_ORDER = SPEED(1) * LEV(1) / (TEAM_SUM_SPEED(2) / TEAM_COUNT(2))
[END]
[FLEE_RATE] //逃跑成功率
  FLEE_RATE = SPEED(1) / (TEAM_SUM_SPEED(2) / TEAM_COUNT(2))
[END]
……
技能公式示例：火球
伤害：600 * SLEV
命中率：(90 + SLEV)/100
消耗：6 * SLEV
经验值：18 * sqrt(SLEV/MAX_SLEV)
```


10.6 技能的实现

(1)技能概述

技能是战斗系统中最大的一块，除了使用道具，最主要的战斗方式就是使用技能，有些技能也可称为魔法。技能数量庞大，形式多样，效果丰富，除了对角色属性的影响之外，还有各种特殊的战斗效果，例如使目标处于不能攻击的麻痹状态或者造成目标中毒，持续减生命值等等。最普通的技能就是物理攻击技能，画面表现为攻击方跑至目标面前，实行攻击，使目标生命值减少，然后返回原位。

因为技能数量庞大，效果各异，所以在设计和实现过程中必须找到技能之间的共有属性和

特征，进行提取和概括，尽量使各种技能的实现形式统一，而对策划来说又不失灵活性。在本实例中，技能是用如下的数据结构描述的：

short	wID；	ID
char	szName[16]；	技能名字
char	szIconFile[20]；	图标文件
char	cRange；	目标范围：1 单人，2 行，3 列，4 十字，5 全体
long	lTargetMask；	使用对象：敌、友、自身等
short	lMemory；	消耗的记忆量
char	cBattleSkill；	技能类型：0 非战斗技能，1 战斗技能
char	cContinue；	源目标已死，是否随机选择新的目标
char	cIsAuto；	是否被动技能
char	cCalFlag；	计算标志：1 物理型，2 魔法型，3 恢复型
char	cTransFlag；	伤害转化标志
char	szExprTRANS[200]；	伤害转化公式
char	szExprSELFHARM[200]；	自身伤害的公式
char	cEffCnt；	会产生多少种状态
short	lEffs[SKILL_MAX_EFFECT]；	状态编号数组
char	szExprRATIO[200]；	引起状态改变的几率公式
char	szExprROUND[200]；	状态持续回合数公式
short	sAutoEff；	被动技能对属性的影响
char	szExprAutoEff[200]；	被动技能对属性的影响公式
char	szExprSATP[200]；	技能伤害公式
char	szExprSHTR[200]；	技能命中率公式
char	szExprUSE[200]；	消耗魔法点数公式
char	szExprEXP[200]；	获得经验值公式
char	szEffFileName[30]；	特效文件名
char	szWave[64]；	音效
char	cBossFlag；	是否对 Boss 有效
char	szComment[80]；	文字描述

下面是关于技能的几个重要属性的说明。

①作用范围：例如，大火球魔法可以攻击对方队伍中整整一排的战士，而单火魔法只能攻击一个目标；单人回复魔法只能对一人使用，而全体回复魔法可以帮助全员加血。因此作用范围是不可缺少的一项技能属性。

②使用对象：例如，自身回复魔法只能对使用者自己起作用，单人回复则可以对使用者自己使用，也可以用在队友身上，但不可对敌使用。

③计算标志：物理伤害和魔法伤害在策划中是用不同的公式计算的，因此在计算技能的伤害效果时要有一个标志来区分。

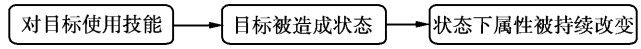
④是否被动技能：一般技能是在战斗中使用后产生效果，被动技能则是一旦学会就会持续发生作用。

⑤各类计算公式:技能伤害、命中率、消耗的魔力点数以及使用后获得的经验值等等都是用公式来描述的,也就是说需要用表达式系统来解析和计算。

⑥造成状态的改变:例如中毒、混乱、威力加强等都是目标被魔法命中后产生的可持续状态。

(2)技能和状态的关系

技能本身是没有持续时间这一概念的,它只是可以被使用一次而已。如果需要技能产生的效果持续一定的回合数,就必须引入战斗状态的概念。例如,有一种技能叫"下毒",希望被它命中的目标持续3个回合都要减血,每个回合减20点;另外有一种技能叫"毒镖",被命中的目标先受到伤害50点,然后每个回合减血20点,持续5个回合。从这样的文字描述中,可以提取出"中毒"这个战斗状态,它可以持续指定的回合数,回合数的大小由引发该状态的技能决定,每个回合对战士的某项属性有影响,这些也就是战斗状态的共性。因此,技能影响战士属性的逻辑就变成了:



在本实例中,战斗状态是用如下数据描述的:

```
short   wID;              ID
char    szName[16];       中文名称
char    cType;            类别(目前用于标识状态特效的表现)
char    cCanAction;       该状态下是否可以行动
char    cRandAtk;         随机攻击标志
char    cAtkClear;        被攻击则解除
char    cTransHarm;       对伤害的转化功效
char    cSpecEff;         特殊效果(0 无,1 透视,2 元素属性反转)
short   sAttribID[5];     最多可以影响 5 种属性
char    cCalFlag[5];      计算标志 0 无效,
                                  1 使用消耗型计算,
                                  2 使用比例累加型计算
char    szExpr[5][200];   影响属性的公式
char    szWave[64];       音效
char    szLinkEff[20];    特效文件名
```

下面是关于状态的几个重要属性的说明。

①状态下是否可以行动:例如,失明、迷惑等状态,玩家或电脑控制的角色在回合开始的时候不能选择行动方式,等价于失去一次操作的机会。

②随机攻击:例如混乱状态,角色也不能自行选择行动方式,而是部分敌友任选一个目标攻击。

③被影响的属性以及影响属性的公式:这里的属性是指角色的HP、MP、攻击力、防御力、速度等,每个状态下最多可以造成角色5种属性的改变,例如一种"重伤"状态,攻击力、防御力、速度等属性都下降到原来的50%。属性被影响的数值也是通过公式计算而来的。

④计算标志:对属性的影响通常会有2种方式:一种是在状态持续的回合中直接减少数

值，例如中毒，每回合减 HP20；另一种是在状态持续的回合中，属性数值变为原有的百分比，例如“减速”状态，角色的速度属性持续为原有的 30%。如果角色同时处于两种状态，而每种状态都会影响某项属性，那么属性的数值就是所有状态累加起来计算的。

10.7 客户端画面表现

图形网络游戏和文字 MUD 最大的区别就是在客户端的图像表现上，网络游戏的客户端要求具有完整的图形界面、动画、音乐音效等各方面功能，在这方面的要求和单机游戏是相同的。前面所讲述的主要围绕着服务器的数值计算、流程控制和技能等内容。下面来看如何在画面上表现回合制战斗。

与单机游戏的回合制战斗不同，网络游戏中每个回合的战斗结果都来自于服务器端。前面在介绍战斗系统交互协议的时候，已经讲述了回合战斗结果的数据格式（消息），这里再看看如何把这样的数据一步一步呈现在画面上。回合战斗数据中包含的信息为（行动方式 + 行动结果）* N，N 的取值最小是 0，最大为参加战斗的总人数，取决于该回合中有多少人行动。

在本实例中行动方式主要有普通物理攻击、使用道具、使用技能（魔法）、防御和逃跑 5 种。普通物理攻击实际上是技能的一种，也是最常用的。从画面看，需要进行动画表现也就是对应的 5 种情况。下面以普通物理攻击为例来看具体的实现方法。

普通物理攻击的画面表现描述：

攻击者跑动至目标面前，播放攻击动作和攻击特效，目标播放受击动作，显示属性改变（血值减少）。如果目标有状态改变，则头上出现对应的状态特效。攻击者攻击结束，转身跑回原位置，仍然保持面对敌人。

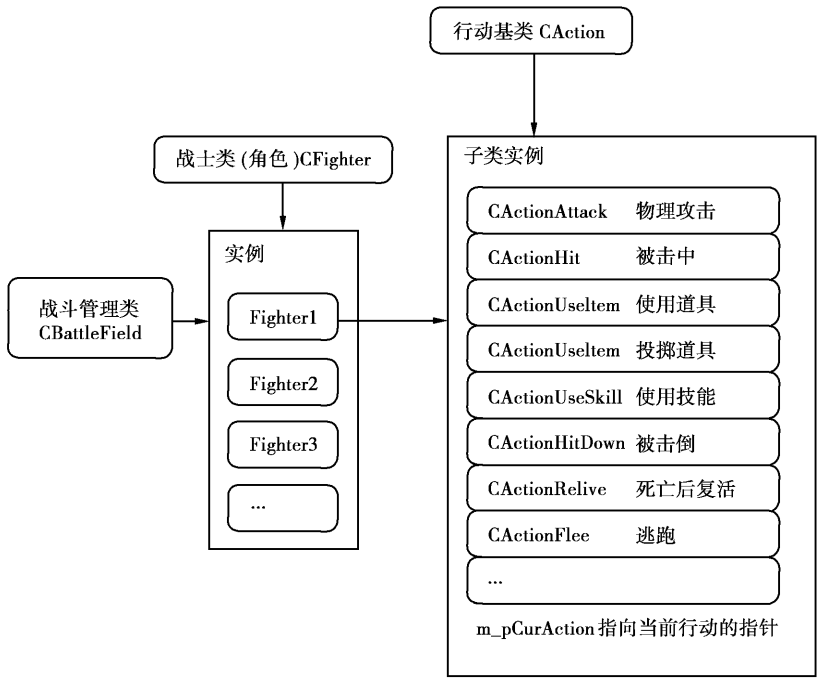


图 10.3

把上面的整个过程叫做一个 Action,其中出现的行动者和目标都叫做 Actor,就好比演员们在指定的场景拍一个情节一样。然后按照流程的进行来写处理代码,用一个基类 CAction 派生出所有的 Action,普通物理攻击就是继承于 CAction 的 CActionAttack,战场中的每个战士都是所有Action的一个实例,需要哪个就播放哪个。加上 Action 的概念之后,战斗系统在客户端的程序结构如图 10.3 所示。

CAction 类定义如下:

```
class CAction
 {
protected:
    CRNAction();
    int     _nStep;              //当前步骤
    int     _nStepTick;          //步骤所进行的时间计数
    int     _nRunTick;           //整个 action 进行的时间计数
public:
    char    m_szName[32];        //action 的名字
    BOOL    m_bRun;              //当前 action 是否在进行
    BOOL    m_bSlow;             //慢放状态标志
    float   m_fTurnAngle;        //所有 action 中,人物转身的速度
                                 //也就是每个 frame 所转的角度
float     m_fMoveSpeed;          //所有 action 中,人物移动的速度
VOID    Active(CBattleField * pStage, CFighter * pActor);     //行动开始
VOID    Render(CBattleField * pStage, CFighter * pActor);     //行动主体循环
void    StepBegin();                                          //步骤开始
void    StepTick();                                           //步骤计数
//虚函数 设置当前的步骤,对于不同的行动进入每个步骤时的处理都不同
virtual VOIDSet(CFightSys * pStage, CFighter * pActor , int nStep) = 0;
//虚函数 步骤的主体循环,对于不同的行动,每个步骤的循环处理都不同
virtual VOIDRun(CFightSys * pStage, CFighter * pActor) = 0;
};
```


说明 每个行动都是以步骤为单位进行的。例如普通物理攻击的行动,步骤分为:

步骤	步骤处理	进入下一步骤的条件
1	攻击者向目标转动方向	转向完成,攻击者面对目标
2	攻击者向目标前进	距离目标位置小于攻击距离
3	播放攻击动作	攻击动作到达关键桢
4	目标激活受击的 Action(CActionHit)	攻击者攻击动作播放完成

续表

步骤	步骤处理	进入下一步骤的条件
5	攻击者转动方向至面对原站立位置	转向完成
6	攻击者向原站立位置前进	到达原站立位置
7	攻击者转向至面对正前方	转向完成
8	行动完成	

行动之间是可以互相触发的，例如在上表中攻击动作可以触发目标的被击动作，而被击行动可以触发死亡的行动，当该回合中所有的行动都结束时，客户端就会通知服务器可以开始新的回合。

CAction 与 CActionAttack 的实现代码如下：

```
CAction::CAction()
{
    strcpy(m_szName, "NULL");
    m_bRun          = FALSE;
    m_bSlow         = FALSE;
    m_fTurnAngle    = F_TURNING_SPEED;
    m_fMoveSpeed    = F_MOVING_SPEED;
    _nRunTick       = 0;
    _nStepTick      = 0;
}
VOID CAction::Active(CBattleField * pStage, CFighter * pActor)
{
    if(m_bRun) //如果正在运行，则退出
    {
        return;
    }
    pActor->m_pCurAction = this; //设置战士的当前 Action
    m_bRun = TRUE;
    _nRunTick = 0;
    _nStepTick = 0;
    Set(pStage, pActor, 0); //进入子类的第一个步骤
}
void CAction::StepBegin()
{
```

```
        _nStepTick = 0 ; //步骤计时清 0
    }
    // Action 的主体循环
    VOID CAction ::Render(CBattleField * pStage , CFighter * pActor)
     {
        Run(pStage , pActor) ;//进入子类的循环处理
        _nStepTick ++ ;
        _nRunTick ++ ;
        if(_nRunTick > (30 * 60 * 2)) //行动超时,说明有步骤无法完成,以方便调试
         {
            Log("ERR :Action Flow Run too much time !") ;
        }
    }
    class CActionAttack :public CAction
     {
    private:
        float _fStandFace[3] ;      //行动前站立方向
        float _fStandPos[3] ;       //行动前站立位置
        float _fTargetPos[3] ;      //目标位置
        CFighter * _pCurTarget ;    //当前的攻击目标
        float _fLastX ;
        float _fLastY ;
    public:
        CActionAttack() ;
        BOOL m_bJoint ;             //其他战士合击
        float m_fAttackDistance ;   //攻击距离
        int    m_nRunPose ;         //跑步的动作编号
        int    m_nAttackPose ;      //攻击的动作编号
        CFighter * m_pTargetList[MAX_ACTION_TARGET] ;//目标列表,物理攻击有可能发动对
        //多个目标的攻击,依次进行
        int    m_nTargetCnt ;       //目标的数量
        int    m_nCurTarget ;       //当前目标
        VOID   Set(CBattleField * pStage , CFighter * pActor , int nStep) ;
        VOID   Run(CBattleField * pStage , CFighter * pActor) ;
    } ;
    CActionAttack ::CActionAttack()
     {
        m_bJoint      = FALSE ;
        m_nTargetCnt= 0 ;
```

```
    m_nCurTarget= 0 ;
    _pCurTarget  = NULL ;
    SetName(" ATTACK") ;
}
//步骤的进入处理
VOID CActionAttack ::Set(CBattleField * pStage , CFighter * pActor , int nStep)
 {
    StepBegin() ;
    if(nStep == 0) //进入步骤 0
     {
        //记录行动前站立的位置和方向
        pActor -> GetPos(_fStandPos) ;
        pActor -> GetFace(_fStandFace) ;
        m_nCurTarget = 0 ; //选择第一个目标
    }
    else if(nStep == 1) //进入步骤 1
     {
        _pCurTarget = m_pTargetList[m_nCurTarget] ; //设置当前目标指针
    }
    else  if(nStep == 2) //进入步骤 2
     {
        _pCurTarget -> GetPos(_fTargetPos) ; //取得目标的位置
        pActor -> FaceTo(_fTargetPos , m_fTurnAngle) ;  // 开始转向,以面对目标
    }
    else  if(nStep == 3) //进入步骤 3
     {
        if(m_nCurTarget == 0) //在前往第一个目标之前,播放起步动作
         {
            pActor -> PlayPose(F_POSE_MOVE_START , NOLOOP) ; //非循环动作
        }
    }
    else if(nStep == 4) //进入步骤 4
     {
         pActor -> PlayPose(F_POSE_MOVE_LOOP , LOOP) ;  //循环播放跑步动作
    }
    else if(nStep == 5) //进入步骤 5
     {
        pActor -> PlayFence(m_nAttackPose) ; //播放攻击动作
    }
```

```
        else if(nStep == 6) //进入步骤 6
        {
            pActor->FaceTo(_fStandPos, m_fTurnAngle); //转向行动前站立的位置
        }
        else if(nStep == 7)
        {
            pActor->PlayPose(F_POSE_MOVE_LOOP, LOOP); //循环播放跑步动作
        }
        else if(nStep == 8)
        {
            pActor->FaceTo(_fStandFace, m_fTurnAngle, 1); //转向原来的站立方向
        }
        _nStep = nStep;
    }
    //步骤的主体循环,将判断每个步骤结束后进入下一个步骤的条件是否满足
    VOID CActionAttack::Run(CBattleField *pStage, CFighter *pActor)
    {
        if(m_bSlow) //如果进入慢放状态
        {
            Sleep(F_DEAD_FRAME_DELAY / 2);
        }
        if(_nStep == 0) //步骤 0 的循环处理
        {
            //如果是合击并且该战士是合击的发起者,播放特定音效
            if(m_bJoint && pActor->GetStat(STAT_COMBINEHOST))
            {
                PLAY_WAVE("f_combineatk", ONCE);
            }
            Set(pStage, pActor, _nStep + 1);
        }
        else if(_nStep == 1) //步骤 1 的循环处理
        {
        //如果当前目标正在执行被击行动
        If(_pCurTarget->m_pCurAction == &(_pCurTarget->m_ActionHit))
        {
        //如果被击动作已经完成,则进入下一步骤,否则进入等待状态
        //之所以要判断这样的条件是因为有些时候,行动者对同一目标是攻击多次的
        //需要等待目标的被击动作播放完成,才能再攻击
        if(_pCurTarget->GetCurPose() == F_POSE_WAITING)
```

```
            {
                Set(pStage , pActor , _nStep + 1);
            }
        }
        else
        {
            Set(pStage , pActor , _nStep + 1);
        }
    }
    else if(_nStep == 2) //步骤 2 的循环处理
    {
        if(!pActor->Turning(m_fTurnAngle)) //按照一定速度转向,直到转到所要的方向
        {
            Set(pStage , pActor , _nStep + 1);
        }
    }
    else if(_nStep == 3) //步骤 3 的循环处理
    {
        if(m_nCurTarget == 0) //对于前往第一个目标的动作
        {
          pActor->Moving(F_MOVE_START_SPEED);//移动速度为起步的速度
          if(pActor->IsPoseEnd(F_POSE_MOVE_START)) //当起步动作完成,进入下一步骤
          {
              Set(pStage , pActor , _nStep +1);
          }
        }
        else  //对于对方的其他目标,无需再有起步的过程,所以直接进入下一步骤
        {
            Set(pStage , pActor , _nStep + 1);
        }
    }
    else if(_nStep == 4) //步骤 4 的循环处理
    {
        //检查当前距目标的位置是否已经到了攻击距离
        //攻击距离 = 攻击动作的有效范围 + 目标的身长
        if(pActor->Distance(_fTargetPos) <= (m_fAttackDistance +
                                _pCurTarget->GetModel()->pAttr->fFrontDist))
        {
                Set(pStage , pActor , _nStep + 1); //到达攻击距离,进入下一步骤
```

```
        }
        else
         {
            pActor->Moving(m_fMoveSpeed);//不断朝着目标跑动
        }
    }
    else if(_nStep==5)//步骤5的循环处理
     {
      //检查攻击动作是否播放到了特定帧数
      //把行动触发和关键帧或特定帧数对应起来,可以做出很准确的攻击和被击的动画感觉
      if(pActor->IsLessFenceHitFrame())  //到达攻击动作关键帧之前的某个位置
       {
        //下面这个函数内部即为对回合战斗结果数据的处理,比如目标是否躲过了攻击
        //就是来自服务器端的结果数据的一种,当攻击动作播放到的关键帧,就开始处理
        //以进行对应的画面表现
        //结果数据是以队列的形式存放在Fighter对象身上的,每处理一个,就弹出一个
        //下次调用此函数就会处理一个新的结果数据
        _pCurTarget->HandleCurActionUpdate(pStage,true);//目标的结果数据处理
        pActor->HandleCurActionUpdate(pStage,true);//行动者的结果数据处理
        if(_pCurTarget->GetStat(STAT_FY))  //如果目标处于防御状态
         {
          //目标进入被击行动,行动的动作是"防御",即目标属于防御被击
          _pCurTarget->m_ActionHit.m_nHitPose = F_POSE_DEFENCE;
          _pCurTarget->m_ActionHit.m_nAttackPose = m_nAttackPose;
          _pCurTarget->m_ActionHit.m_pAttacker = pActor;
          _pCurTarget->m_ActionHit.Active(pStage,_pCurTarget);
        }
        else if(_pCurTarget->GetStat(STAT_DODGE))  //目标躲过了此次攻击
         {
          _pCurTarget->m_ActionHit.m_nHitPose = F_POSE_DODGE;
          _pCurTarget->m_ActionHit.m_nAttackPose = m_nAttackPose;
          _pCurTarget->m_ActionHit.m_pAttacker = pActor;
          _pCurTarget->m_ActionHit.Active(pStage,_pCurTarget);
        }
    }
    if(pActor->IsFenceHitFrame())  //到达攻击动作的关键帧
     {
        float fTargetPos[3];_pCurTarget->GetPos(fTargetPos);
```

```
        if(_pCurTarget->IsDead())  //如果目标已经死亡
        {
            PLAY_WAVE("f_physicalhit",ONCE);
            if(_pCurTarget->GetStat(STAT_HITFLY)) //目标被打飞出战场
            {
                //进入“被打飞”的行动
                _pCurTarget->m_ActionHitFly.Active(pStage,_pCurTarget);
            }
            else
            {
                m_bSlow=TRUE;//用慢动作进入“死亡”的行动
                _pCurTarget->m_ActionDead.m_bSlow=TRUE;
                _pCurTarget->m_ActionDead.Active(pStage,_pCurTarget);
            }
            PlayEffect("eff_001",fTargetPos);
        }
        else
        {
            // 如果目标防御或者躲闪了此次攻击
            if(_pCurTarget->GetStat(STAT_FY)||_pCurTarget->GetStat(STAT_DODGE))
            {
                if(_pCurTarget->GetStat(STAT_FY)) //播放防御的音效和特效
                {
                    PLAY_WAVE("f_defence",ONCE);
                    PlayEffect("eff_fy",fTargetPos);
                }
                else // 目标被击中,播放对应的音效和特效,进入“被击”的行动
                {
                    PLAY_WAVE("f_physicalhit",ONCE);
                    PlayEffect("eff_001",fTargetPos);
                    _pCurTarget->m_ActionHit.m_bRun        = FALSE;
                    _pCurTarget->m_ActionHit.m_nAttackPose = m_nAttackPose;
                    _pCurTarget->m_ActionHit.m_nHitPose    = F_POSE_HIT;
                    _pCurTarget->m_ActionHit.m_pAttacker   = pActor;
                    _pCurTarget->m_ActionHit.Active(pStage,_pCurTarget);
                }
            }
        }
        if(pActor->IsFenceEnd()) // 攻击方攻击动作播放完毕
```

```
            {
                m_nCurTarget ++ ; //换下一个目标进入攻击
                if(m_nCurTarget < m_nTargetCnt ) //如果还有目标要攻击
                {
                    Set(pStage , pActor , 1); //跳回步骤 1
                }
                else //全部目标都已被攻击,进入下一步骤
                {
                    Set(pStage , pActor , _nStep + 1);
                }
                m_bSlow = FALSE; //如果有慢放,则结束
            }
        }
        else if(_nStep == 6) //步骤 6 的循环处理
        {
            if(!pActor->Turning(m_fTurnAngle)) // 转向直至面对原来站立的位置
            {
                Set(pStage , pActor , _nStep + 1);
            }
        }
        else if(_nStep == 7) //步骤 7 的循环处理
        {
            float fMove; //如果到达到原来站立的位置,则进入下一步骤
            if(!pActor->Moving(m_fMoveSpeed, _fStandPos, &fMove))
            {
                Set(pStage, pActor, _nStep + 1);
            }
        }
        else if(_nStep == 8) //步骤 8 的循环处理
        {
            if(!pActor->Turning(m_fTurnAngle)) //转向直到面对方向为最初站立时的方向
            {
                pActor->Stand(); //播放战斗等待动作
                m_bRun = FALSE; //行动结束
                m_bSlow = FALSE; //如果有慢放,慢放结束
            }
        }
    }
```

套用 CActionAttack 的结构，就可以实现 CActionUseItem，CActionFlee 等所有的行动处理类，而对于每个战士来说总是有一个当前的行动 m_pCurAction，战场类 CBattleField 来管理整个流程。

战场的主体循环为：

```
bool CBattleField::Run()
 {
     Fighter = FighterList[n];
     Fighter->Run()
 }
```

战士的主体循环为：

```
bool CFighter::Run()
 {
     if(m_pCurAction)
      {
          return m_pCurAction->Run()
      }
     return false;
 }
```

客户端收到回合战斗结果数据后，进入每个战士对应的 Action，当所有的 Action 进行完毕，就通知服务器端进入新的回合。战斗系统按照这个流程循环进行着，直到战斗结束。

至此，回合制战斗系统的制作完毕。从整体结构上来说还算不错，不过在设计中还有些不尽人意的地方，比如在制定客户端与服务器的交互时，没有考虑到非战斗状态下也需要一些同类的交互，因为有些道具和技能在非战斗状态下也是可以使用的，造成与非战斗状态有些数据结构和交互消息重复建设。另外，在客户端的动作控制和表现上，如果能采用脚本则会更好，而这里整个行动流程写死在代码里，不容易扩充和修改。

第11章 MySQL数据库的应用

这里,数据库选用 MySQL。一方面 MySQL 在性能上与其他数据库相比毫不逊色,另一方面 MySQL 在一定程度上是免费的。

11.1 MySQL 简介

MySQL 由瑞典的 T. C. X 公司开发的。最早只是计划制作一个快速的、结构简单的数据库,经过不断的发展,目前已成为一个优秀的数据库系统。

MySQL 有以下特点:

①真正的多用户、多线程 SQL 数据库服务器。MySQL 采用 Client/Server 的结构运作。Server 直接运行 MySQL 服务程序,Client 利用 MySQL 的开发包编写,一般使用 TCP/IP 与 Server 通讯。Server 支持多个 Client 同时访问,Server 使用了多线程技术,通过增加 CPU 数量可以提高负载,如图 11.1 所示。

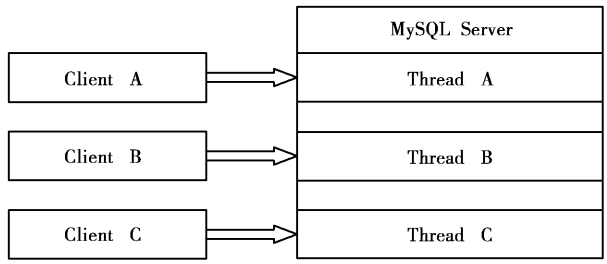


图 11.1

②跨平台的数据库服务器。MySQL 支持的操作系统多达十几种,其中包括 Windows 系列与 Linux 系列,给了用户更大的选择空间。

③快速。MySQL 最初的设计目的就是希望在速度上超越现有的数据库系统,并为此牺牲了一部分功能,因此在运行速度上还是有较大优势的。

④支持多种客户端接口。从最底层的 C API 到 Perl,PHP,Python 等等,都可以用来开发 MySQL 的客户端。

⑤完整的认证系统。MySQL 有自己的权限系统,可以给不同的用户划分数据库的读写权限。

11.2 MySQL C API

在 MySQL 提供的各种开发接口中,C API 是最底层的,同时也是最快速的。这里以 MySQL 3.23 版为例,介绍用 C API 读写 MySQL 的一般流程,如图 11.2 所示。

①建立与 MySQL Server 的连接。使用 MySQL 提供的连接函数,原型如下:

```
MYSQL * MySQL _ real _ connect (
    MYSQL * MySQL,
    const char * host,
    const char * user,
    const char * passwd,
    const char * db,
    unsigned int port,
    const char * unix _ socket,
    unsigned int client _ flag
)
```

其中,参数含义如下:

MySQL **为一个** MySQL **的句柄,用来代表一个** Client **到** Server**的连接;**

host **为** MySQL Server **的地址;**

user **为用户名;**

passwd **为密码;**

db **为默认使用的数据库;**

port **为** MySQL Server **的监听端口;**

unix _ socket **为字符串指定套接字或应该被使用的命名管道;**

client _ flag **为特殊标记,一般使用** 0。

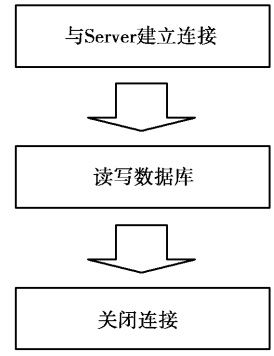


图 11.2

例如 MySQL Server IP **是** 10.10.3.1**,使用默认端口监听,用户名是** test**,密码是** 111111**,则可以使用如下代码:**

```
MYSQL       * m _ handle;
//初始化 MySQL 句柄
m _ handle = MySQL _ init(NULL);
//连接
MySQL _ real _ connect(m _ handle," 10.10.3.1 "," test "," 111111 ",NULL,0,NULL,0);
//检查连接结果
if(m _ handle)
 {
    //连接成功
}
else
 {
    //连接失败
}
```

②连接建立后,就可以直接执行 SQL **语句。使用查询函数,原型如下:**

```
int MySQL _ real _ query (
        MYSQL * MySQL,
        const char * query,
        unsigned int length
)
```

其中,参数含义如下:

MySQL 为一个 MySQL 的句柄;

query 为查询语句;

length 为查询语句的长度。

MySQL 还提供了另一个查询函数 MySQL _ query,函数内部使用 strlen 来确定查询语句的长度,但对于包含二进制数据的查询,应该使用 MySQL _ real _ query,因为二进制数据中可能包含代表字符串结束的'\0'。

对于有返回数据的查询,如 select,需要把查询结果存储在本地,供进一步分析。使用函数 MySQL _ store _ result:

```
MYSQL _ RES * MySQL _ store _ result(
        MYSQL * MySQL
)
```

返回一个结果集,然后再对结果集进行操作:

```
MYSQL _ ROW MySQL _ fetch _ row (
        MYSQL _ RES * result
)
```

整个流程大致如图 11.3 所示。

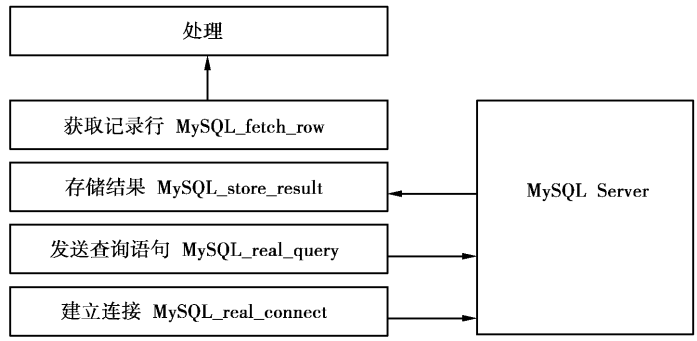


图 11.3

③查询结束后,应主动关闭连接,释放资源。代码如下:

```
void MySQL _ close (
        MYSQL * MySQL
)
```

传入连接句柄,关闭连接。如果在连接建立期间使用了 MySQL _ store _ result,**则应该释放本地的结果集。代码如下:**

```
void MySQL _ free _ result (
        MYSQL _ RES * result
)
```

11.3 设计一个简单的 MySQL 读写类库

由于 MySQL **使用的是** C API,**使用并不十分方便,这里做一个简单的** MySQL **读写类库,以方便使用,其结构如图** 11.4 **所示。**

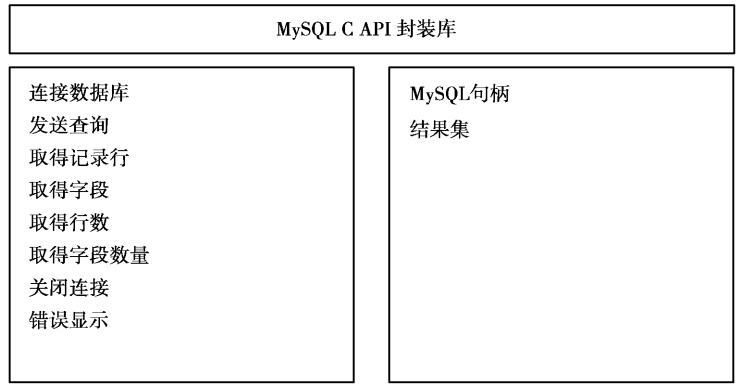


图 11.4

MySQL C API **封装类头文件** g _ dbMySQL. h:

```
/ *
g _ dbMySQL. h:interface for the CG _ DBMySQL class.
 * /
#ifndef __ CG _ DB _ MYSQL _ H __
#define __ CG _ DB _ MYSQL _ H __
#include "g _ platform. h"
#include  < MySQL. h >
class CG _ DBMySQL
 {
public:
```

```
    unsigned long GetInsertId();
    long   GetAffectedRows();
    bool   Query(const char * sql,int len = 0);
    bool   Connect(const char * host,int port,const char * name,const char * pwd,const
                   char * db);
    bool   GetFieldContent(int nIdx,int * pnType,char * * ppValue);
    bool   GetRow();
    void   SetLog(bool log);
    void   ShowError();
    void   Close();
    int    GetFieldInfo(int nIdx,char * pszName);
    int    GetRowCount();
    int    GetFieldCount();
    char * GetField(char * fname,int * len = NULL);
    CG_DBMySQL();
    virtual ~CG_DBMySQL();
private:
    bool    m_log;
    MYSQL   * m_handle;
    MYSQL_RES    * m_res;
    MYSQL_ROW    m_row;
    MYSQL_FIELD  * m_fields;
    int          m_fieldCnt;
};
#endif
```

MySQL C API **封装类文件** g_dbMySQL.cpp:

```
/* g_DBMySQL.cpp:implementation of the CG_DBMySQL class. */
//头文件
#include "g_dbMySQL.h"
//构造函数
CG_DBMySQL::CG_DBMySQL()
{
    m_handle = NULL;
    m_res = NULL;
      SetLog(true);
}
```

```
//析构函数
CG_DBMySQL::~CG_DBMySQL()
{
    Close();
}
//尝试连接 MySQL Server
bool CG_DBMySQL::Connect(const char *host,int port,const char *name,const char
*pwd,const char *db)
{
    Close();
    m_handle = MySQL_init(NULL);
    if(m_handle==NULL) return false;
    if(port == 0) port = MYSQL_PORT;
    if(MySQL_real_connect(m_handle,host,name,pwd,NULL,port,NULL,0) == NULL)
    {
        ShowError();
        return false;
    }
    if(MySQL_select_db(m_handle,db) != 0)
    {
        ShowError();
        return false;
    }
    return true;
}
//关闭与 MySQL Server 的连接
void CG_DBMySQL::Close()
{
    if(m_handle)
    {
        MySQL_close(m_handle);
        m_handle = NULL;
    }
    //如果有本地结果集,那么就释放它,避免内存泄露
    if(m_res)
    {
        MySQL_free_result(m_res);
        m_res = NULL;
```

```
    }
}
//发送 SQL 语句
bool CG_DBMySQL::Query(const char *sql,int len)
{
    if(!m_handle) return false;
    if(len == 0) len = strlen(sql);
    if(MySQL_real_query(m_handle,sql,len) != 0)
    {
        //执行失败,显示错误
        ShowError();
        return false;
    }
    //释放上次的结果集
    if(m_res != NULL) MySQL_free_result(m_res);
    //保存本地结果集
    m_res = MySQL_store_result(m_handle);
    if(m_res != NULL)
    {
        //保存字段数量
        m_fieldCnt = MySQL_num_fields(m_res);
        m_fields = MySQL_fetch_fields(m_res);
    }
    return true;
}
//获取字段数量
int CG_DBMySQL::GetFieldCount()
{
    return m_fieldCnt;
}
//获取行数
int CG_DBMySQL::GetRowCount()
{
    return (int)MySQL_num_rows(m_res);
}
//显示错误
void CG_DBMySQL::ShowError()
{
```

```
    if(m_log)
        Sys_Log("SQL EER = [%s]",MySQL_error(m_handle));
}
//获得结果集中的下一条记录
bool CG_DBMySQL::GetRow()
{
    m_row = MySQL_fetch_row(m_res);
    if (m_row == NULL) return false;
    return true;
}
//根据字段名取得字段内容
char *CG_DBMySQL::GetField(char *fname,int *pnLen)
{
    int i;
    unsigned long *lengths;
    lengths = MySQL_fetch_lengths(m_res);
    for(i = 0;i < m_fieldCnt;i++)
    {
        if (strcmp(fname,m_fields[i].name) ==0)
        {
            if(pnLen) *pnLen = lengths[i];
            return m_row[i];
        }
    }
    return NULL;
}
//按索引获得字段名与字段类型
int CG_DBMySQL::GetFieldInfo(int nIdx,char *pszName)
{
    if(nIdx <0 || nIdx >= m_fieldCnt)
    {
        return 0;
    }
    if(pszName)
    {
        strcpy(pszName,m_fields[nIdx].name);
    }
    return m_fields[nIdx].type;
}
```

```
//按索引获得字段内容与类型
bool CG_DBMySQL::GetFieldContent(int nIdx,int *pnType,char **ppValue)
{
    if(nIdx<0 || nIdx>=m_fieldCnt)
    {
        return false;
    }
    *pnType  = m_fields[nIdx].type;
    *ppValue = m_row[nIdx];
    return true;
}
//获得上次插入新记录的自动 ID
unsigned long CG_DBMySQL::GetInsertId()
{
    return (unsigned long)MySQL_insert_id(m_handle);
}
//获得上次查询受影响的行数
long CG_DBMySQL::GetAffectedRows()
{
    return (unsigned long)MySQL_affected_rows(m_handle);
}
//设置是否显示错误
void CG_DBMySQL::SetLog(bool log)
{
    m_log = log;
}
```

11.4 一些常见的问题

(1) MySQL 函数所消耗的时间

由于 MySQL 是网络数据库,所以 Client 与 Server 可能不在同一台机器上,如图 11.5 所示。当 Client 的执行线程调用 MySQL_query 时,MySQL 库发送查询请求数据包,MySQL Server 接收到请求,进行计算,然后返回结果。因此 MySQL_query 的执行时间可计算如下:

花费时间 = 网络往返时间 + 计算时间

其中,计算时间和查询语句的复杂度、数据包的大小、MySQL Server 的当前负载有关;而网络往返时间则和网络环境有关,在目前的局域网环境下,一般只需 0.5ms,基本可以忽略不计。但在写程序时还是应该注意到网络往返时间。

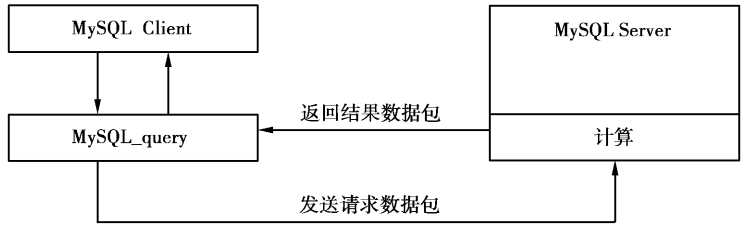


图 11.5

(2)错误显示

在使用 C API **编写程序时,需要通过返回值检查错误,然后再打印错误字符串。** MySQL **提供** MySQL _ error **函数,直接以字符串形式返回错误:**

```
char * MySQL _ error (
    MYSQL * MySQL
)
```

对于由 MySQL **句柄指定的连接,**MySQL _ errno **返回最近调用的** API **函数的错误代码。如果没有错误发生,则返回空字符串。**

第12章 数据库服务器的设计与实现

在前面的关于 Server 架构的章节中,已经讲过在 GWS(游戏世界服务器)对玩家数据库的存取过程中,有一个中介性质的服务器程序 DBS,多台 GWS 公用一个 DBS,而 DBS 才是直接与数据库系统如 MySQL 相连。下面就来看看如何设计这样一个服务器程序。

12.1 DBS 的设计目的和要点

(1) DBS 的作用

DBS 的作用如下:

①GWS 存取数据库的中间代理,避免直接的阻塞式存取。

②数据通讯的时候可以组合不同类型的数据成数据包,避免将数据全部转化为 SQL 语句字符串。

③针对跨越多台 GWS 的游戏内容进行特殊处理,如邮件系统、家族系统等。

④对玩家数据库表做定时的备份。

(2) DBS 的功能结构

DBS 的功能结构如图 12.1 所示。

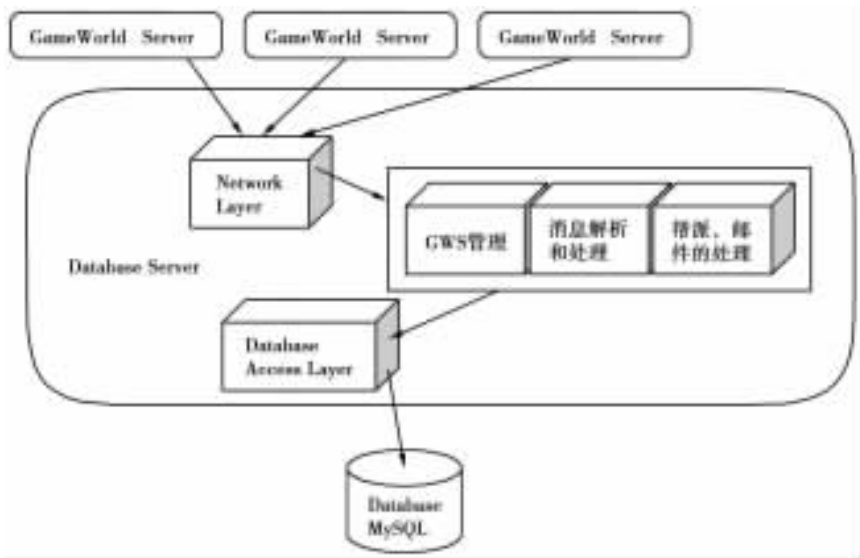


图 12.1

DBS 的功能结构如图 12.1 所示。DBS 具备一个基本的服务器程序的结构,其中 Network Layer 就是基于 Socket 的网络连接以及消息发送和接收,通过这一层,DBS 对多个 GWS 起到一个管理的作用,GWS 可以登录和退出 DBS,DBS 会对登录的 IP 做一个校验,以决定 GWS 的组和编号。通过编号,还会对游戏使用的数据库资源进行一些分配处理,如分配道具 ID 的数值范围等。然后 DBS 需要解析并处理来自 GWS 的消息,并且转换为数据库的存取操作,把需要的数据发回给提出请求的 GWS。对于帮派和邮件这类需要跨越多个 GWS 的游戏内容,除了对数据库表的操作之外,DBS 还需要对它们作特别的处理,例如帮派的解散,需要通知分布在各个 GWS 里的所有帮派成员等。Database Access Layer 则是封装了对 MySQL 数据库进行存取操作的一个类库,在前面的章节中已经详细介绍过。

举个例子来说,在存取玩家数据的时候,DBS 会处理的工作有:

①玩家通过账号登录以后,由账号查询所有角色信息。即 DBS 执行:

- 从“角色表”里取出并返回所有角色的数据;
- 从“道具表”里依次取出并返回各角色的道具数据。

②玩家退出游戏的时候,进行存盘操作,GWS 把角色数据、角色的所有道具数据发给 DBS,DBS 依次执行:

- 更新“角色表”角色数据;
- 更新“道具表”,把该角色上次拥有的所有道具的“所属”字段设空;
- 更新“道具表”,把角色现有的道具数据写入“道具表”。

③游戏定时存盘,操作过程同②。

了解了功能结构和需求之后,就可以进行具体的设计和编码了。这里用一个类 CGWS 来代表一个 Game World Server,用类 CGWSMgr 来进行管理操作和流程控制,它们的关系和作用如图 12.2 所示。

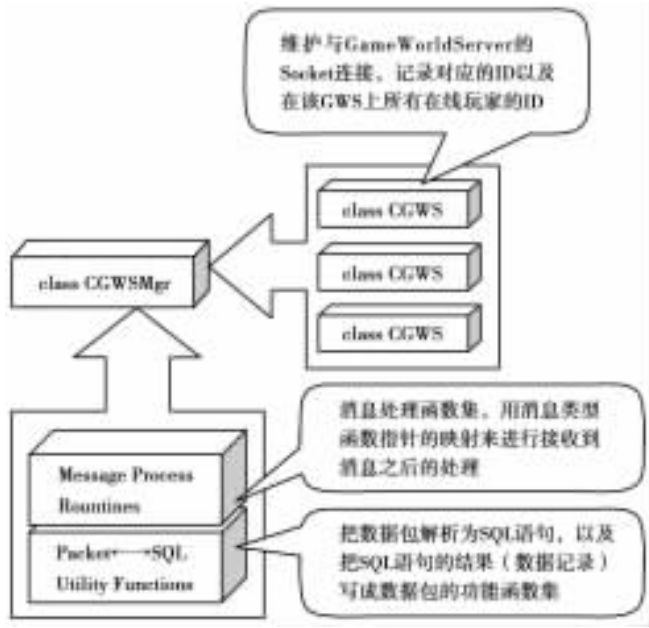


图 12.2

(3)设计要点

设计要点如下:

①DBS 本身只受数据表关键索引的影响,数据表中任何非关键索引的字段变动,如增加、删除或改变次序无需修改 DBS 程序。关键索引主要指 AccountID(账号 ID),Character ID(角色 ID),Pet ID(宠物),Faction ID(家族或帮派)等等。可以认为在任何游戏中,这些类型的数据索引都是存在的。

②GWS 属于直接游戏内容相关的服务器,需要解析数据表为游戏内部数据。DBS 则不同,它应该尽量保持与游戏内容较少的关联,这样可以提高可重用性。

③DBS 如何处理 GWS 的请求主要依赖于交互协议中消息的定义。

上面所说的要点都有一个共同的出发点,就是尽量把 DBS 从游戏层独立出来,使之更像一种通用的数据库存取服务器,可以略加修改就用于不同的网络游戏中。为了说明这种设计思想,先看一个实际应用的例子玩家角色数据存盘:

无论什么类型的网络游戏,玩家角色数据总是必不可缺的,在数据库里,相关数据表通常会有少则几十个,多则上百个数据库字段。例如,cha _ id(ID)、name(名字)、model _ id(模型 ID)等等。

如果 GWS 向 DBS 发送一个存盘的请求,那么可以用两种形式来实现这个过程,一是把请求存盘的消息定义为表 12.1 所示的格式。

表 12.1

消息中包含的数据项	数据类型	含　义
GWS2DBS _ SAVE _ CHARACTER	short	消息类型为角色存盘
character _ id	long	角色 ID
Name	string	名字
account _ id	long	所属账号的 ID
model _ id	long	所使用的模型的 ID
Money	long	金钱数
Atr _ str	short	角色的力量属性
Atr _ dex	short	角色的敏捷度属性
Atr _ magic	short	角色的魔力属性
…	…	…

按照这样的定义方式,GWS 和 DBS 之间约定好消息的格式,GWS 按照固定的次序和数据类型发送存盘消息,而 DBS 按照同样的次序和数据类型接收并解析该消息,然后根据这些数据的含义对应到数据库相应字段来执行操作。这种 DBS 来解析具体字段信息的方式的优点是简单明了、易于实现,缺点是不方便维护,尤其是在游戏开发阶段,数据库表中的字段经常会有添加、删除或改变次序的情况,一旦发生,就需要同时修改 GWS 和 DBS 的相应位置的代码,而且要求发送和解析顺序一定要完全一致,对应的数据内容和数据库字段也要做相应的修改。这样的做法就要求 DBS 自身了解每项游戏数据具体是什么数据类型,也就是要知道这些数据都是用来表示什么的,使得 DBS 和游戏内容本身紧密地结合起来。如果换一个新的游戏,有不同的游戏数据和类型,这些代码就要完全重写了,这与最初的设计要点不符合。

作为一个数据库存取的中介,DBS 不应该或者说最好不要去解析具体的游戏内容数据,因此这里采用另一种方式来制定 GWS 和 DBS 之间的交互协议。让我们来看看 GWS 对 DBS 连接的数据库到底有怎样的需求,并把它们概括出来。对数据库的操作也就是对游戏中用到的不同数据表的操作,如几乎每个游戏都具备的角色表、道具表等等(也有些游戏中道具数据会合并在角色表中)。对表里的数据进行的操作也就是插入、查询、修改、删除等有限的几种

而已，可以把这些操作定义成消息类型，把字段名和该字段对应的数据类型以及实际数据内容依次附在后面，DBS 接收到消息后，也就获得了字段名信息和对应的实际数据，可以自动地对数据表进行操作而不用在乎各个数据的含义和发送的次序。缺点是在数据包中增加了字段名和数据类型的额外信息，消息尺寸比使用第一种方式要大。但相比其方便之处，这个缺点可以忽略不计。在本实例的游戏开发过程中，角色数据表的字段增加了多次，但 DBS 本身完全不用修改程序，而且对数据表的操作可以做成通用的自动的函数，不易出错。游戏虽然在不断地增加内容和修改，但 DBS 基本上无需投入时间维护。

本实例中，游戏数据库的结构如图 12.3 所示。

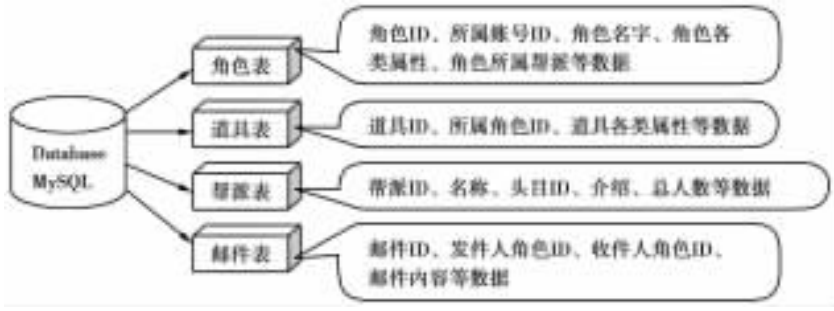


图 12.3

另外，还有交易记录表以及服务器关键操作的 Log 表等辅助性的数据表，图 12.3 中每个表都可以通过惟一的 ID 查询到相应数据。这些表之间的基本关系有：每个账号可以拥有 2 个角色，每个角色可以拥有数 10 种道具，通过帮派 ID 可以从角色表里查找到所有该帮派成员等等。

12.2 DBS 的实现

具体的类和函数的代码请参考本书配套光盘/补充与提高/第 12 章补充——具体类和程序代码。

光盘中的代码已经列出了 DBS 大部分关键的操作，可以看清作为游戏世界服务器 GWS 和数据库之间的一个中介程序 Database Server 是怎样发挥作用的。对于类似的功能，不同的网络游戏会有不同做法，在有些网络游戏的架构甚至完全没有 DBS 的存在，而是直接操作数据库，对于跨多个服务器的帮派或邮件等系统则由另外的游戏世界服务器管理程序专门处理。正所谓“仁者见仁，智者见智”，但需要特别强调的是，在保证功能和性能的前提下，结构清晰合理，重用性好，易于维护都是不可忽略的要点。

第13章 自动更新系统的设计与实现

经常玩网络游戏的人都知道,在很多网络游戏中,登陆游戏之前会出现一个提示更新的界面,告诉玩家有多少游戏客户端文件需要更新到新版本,然后就开始下载并更新文件,全部更新结束后才可以进入游戏。不光是网络游戏,还有不少应用软件也是如此,启动之前会有文件版本检查和自动更新过程。通过这种方式,用户的客户端可以时时保持在最新状态,也方便软件开发人员做更新和维护。下面介绍一种实现自动更新系统的方法。

一个自动更新系统可以分为 3 个部分 :文件版本管理、文件传输服务器和客户端用来接受更新内容的功能模块。

13.1 文件版本管理

自动更新的原理就是为客户端的每个文件建立版本,服务器上拥有客户端的最新版本,客户端连接上服务器以后,获取关于文件最新版本的数据,并和本地版本对比,发现不同后服务器就会把最新的文件传到客户端,并更新版本号。

首先来看如何建立文件的版本。标识一个文件的版本可以有很多种方法,如手工为文件建立版本号或者利用文件的最后修改日期等。但是,如果依靠手工去产生文件版本信息,那么当文件数量巨大,如上千个,将会很难人工维护,而且非常容易出错 如果靠文件的最后修改日期,也不能保证总是正确,假如在某次更新中,是用老文件替换掉有问题的新文件,那么最后修改日期就不起作用了,而且修改日期可能会受操作系统的影响,文件服务器很有可能运行于 Linux 下,而文件的产生和修改则多半是在 Windows 系统下,当在不同的操作系统之间建立、修改、复制文件,文件最后的修改日期就有可能变得不是需要的那样。到底怎样才能方便又不会出错地建立文件版本呢?下面介绍一种强大的标识方法,即 MD5 信息摘要算法。

MD5 的全称是 Message-DigestAlgorithm5(信息摘要算法),这个 5 实际上是指版本,早期还有 MD2,MD4 等,是由美国 MITLaboratory for Computer Science 和 RSA Data Security Inc 的 Ronald L. Rivest 开发的,主要作用是把一段数据‘ 摘要 ”为一个 128 位长的数字,而这个数字可以惟一标识出这段数据。就好像人的指纹可以惟一标识出一个人一样。MD5 可以把整个文件当作一个大文本信息,通过算法变换,产生惟一的 MD5 信息摘要。以后传播这个文件的过程中,无论文件的内容发生了任何形式的改变(包括人为修改或者下载过程中线路不稳定引起的传输错误等),只要重新计算这个文件的 MD5 值,就会发现是不相同的,由此可以确定文件发生了变化,平时常听说的所谓的数字签名也是类似的应用。MD5 还广泛用于加密和解密技术上,如在 UNIX 系统中用户的密码就是以 MD5(或其他类似的算法)经加密后存储在文件系统中。而且 MD5 算法的使用不需要支付任何版权费用,这里就利用它来建立文件的版本。

可以从网上获得 MD5 实现的 C 语言版本,一般有如下几个文件:

global. h	放置一些算法依赖的数据类型定义
md5. h	算法头文件
md5c. c	算法实现函数
mddriver. c	用法示例和一些简单的最外接口函数
	……

使用起来很简单,如调用函数 MDFile(char * pszFileName,char * pszDigest),传入要获得摘要的文件名,然后就可取得摘要的十六进制的字符串,长度为 32,是由 128 位常数值转换而成。当

文件发生变化时,这个字符串也必然发生变化,这个字符串就可以当作该文件的版本信息。

下面要对整个游戏客户端的文件进行版本管理。假设游戏客户端目录是 game_client,遍历其中所有文件,然后进行压缩并产生一个版本信息的记录文件:

目录 game_client -> 目录 game_client_ok
版本文件 ver. dat

新的目录 game_client_ok 里的目录结构和文件名与 game_client 完全一致,只不过所有的文件都被压缩过。当发生更新时,由位于玩家那边的客户端来接收并解压,这样就极大地节省了文件传输的数据量。压缩和解压采用的算法是 glib,也可以完全免费地使用。

这里封装了一个 CFileVersionMgr 类,专门用来对指定目录里的文件进行版本维护管理,它可以递归地遍历整个目录和其子目录中的所有文件,并产生一个文件信息列表,也可以压缩目录里的所有文件到指定目录,还可以跨平台使用,具体代码请参考本书的配套光盘。

13.2 文件传输服务器

准备好了文件版本,就可以通过一个文件传输服务器把文件发送到客户端,并且让客户端作比对工作。下面来看文件传输服务器的制作。文件传输服务器本身构架很简单,就是一个基于 Socket 的服务器,允许多个客户端连入,并进行文件传输。功能也很独立,只要有客户端提出文件请求,就开始传输,而且它自身并不知道有文件版本的存在,是个很单纯的文件传输工具。

这个服务器主要由下面 3 个类组成:

CServerApp:应用程序类,包含一些端口、绑定 IP、配置选项等网络应用程序的基本信息。

CClientSocket:保持客户端连接的 Socket 类,内部负责文件传输的控制。

CServerSocket:用来监听的服务器 Socket 类。

上面两个 Socket 类都是由 Socket 功能类继承而来,这些功能类在第 2 章有详细介绍。

ClientSocket. h:

```
#ifndef TAG_RNSOCKET
#define TAG_RNSOCKET
#include" MFTCPSession. h"
#include" NetDef. h"
#include" util. h"
#include < list >
using namespace std;
//用来监听的 Socket 类
class CServerSocket :public CMFNetSocket
 {
public:
      virtual void OnAccept();//虚函数,当有新的客户端 Socket 连入时被调用
};
```

```
//每一个 CClientSocket 代表一个连接中的客户端,控制连接、文件传输、断开
//连接等关键流程以及相关数据
class CClientSocket :public CMFTCPSession
 {
private ://下面是有关当前正在读取中的文件相关属性
    FILE   * _fp;                    //文件句炳
    char   _szCurFileName[255];      //文件名
    int    _nReadLoc;                //当前文件中的读取位置
    int    _nReadSize;               //读取长度
    BYTE   _btReadBuf[MAX_READ_BUF]; //读文件的缓冲
    int    _nReadBufSize;                    //缓冲大小
    int    _nWaitingState;           //等待状态
    int    _nRunTick;                //当前状态的循环次数
    void   _HandleMsg(CMFCmdPacket * packet); //消息处理函数
public:
    //连接中的 Socket 的状态定义
    stati cconst int STATE_WAIT_NOTHING;      //空闲
    stati cconst int STATE_WAIT_BEGIN_OK;     //等待客户端开始接收文件的回馈
    stati cconst int STATE_WAIT_CONTENT_OK;   //等待客户端接收了一段文件内容的回馈
    stati cconst int STATE_WAIT_END_OK;       //等待客户端结束文件接收的回馈
    void SetWaitingState(intnState);          //设置状态
    CClientSocket();
    ~CClientSocket();
    virtual void OnDisconnect();
    virtual void OnReceive(CMFCmdPacket * packet);
//文件传输控制
private:
    void_SendMsgFileBegin(char * pszName);//发送消息"文件开始"
    bool_SendMsgFileContent();            //发送消息"一段文件内容"
    void_SendMsgFileEnd();                //发送消息"文件结束"
public:
    bool BeginSendFile(char * pszName);   //开始发送指定文件
    bool SendingFile();                   //发送一段文件内容
    void EndSendFile();                   //结束文件传输
    //流程主循环
    void Run();
};
#endif
```

ClientSocket. cpp：

```
#include" ClientSocket. h"
#include" util. h"
#include" ServerApp. h"
//功能 处理新客户端连接
void CServerSocket::OnAccept()
 {
    CClientSocket * pSocket = new CClientSocket;//创建新的连接 Socket
    if(pSocket -> Accept(this))
     {
        theApp. m_ClientSocketList. push_back(pSocket);//添加到客户端连接列表中
        Log(" hnAccept hn");
    }
    else
     {
        delete pSocket;//如果 Accept 不成功,删除
    }
}
const int CClientSocket::STATE_WAIT_NOTHING    = 0;
const int CClientSocket::STATE_WAIT_BEGIN_OK    = 1;
const int CClientSocket::STATE_WAIT_CONTENT_OK = 2;
const int CClientSocket::STATE_WAIT_END_OK    = 3;
CClientSocket::CClientSocket()
 {
    _fp              = NULL;
    _nReadSize     = 0;
    _nWaitingState = STATE_WAIT_NOTHING;
    _nRunTick     = 0;
    _nReadBufSize = 3050;
}
//注 如果客户端在文件传输过程中断开,此处一定要注意关闭文件句柄
CClientSocket::~ CClientSocket()
 {
    if(_fp ! = NULL)//如果文件句柄仍处于打开状态
     {
        fclose(_fp);
        Log(" Close File Handle When Delete Socket ! hn");
    }
```

```
}
void CClientSocket::SetWaitingState(int nState)
{
    _nRunTick      =   0;
    _nWaitingState =   nState;
}
void CClientSocket::OnReceive(CMFCmdPacket * packet)
{
    _HandleMsg(packet);//进入消息处理
}
void CClientSocket::OnDisconnect()
{
    theApp.m_DisconnectList.push_back(this);//添加到列表,专门记录断开连接的 Socket
    Log("Disconnect!,current connection total = %d\n",theApp.m_ClientSocketList.size());
}
void CClientSocket::_HandleMsg(CMFCmdPacket * packet)
{
    BYTE btMsgType;
    if(packet->ReadByte((char *)&btMsgType) == false)//取得消息类型
    {
        return;
    }
    if(btMsgType == MSG_REQUEST_FILE)//消息:客户端要求传输文件
    {
        char * pszFileName;
        packet->ReadString(&pszFileName);
        Log("Request File Name = <%s>\n",pszFileName);
        if(strcmp(pszFileName,"ver") == 0)//新的客户端
        {
            _nReadBufSize = MAX_READ_BUF;
            Log("New Client,BufSize->%d!\n",MAX_READ_BUF);
        }
        if(!BeginSendFile(pszFileName))//打开文件失败,通知客户端文件没找到
        {
            CMFCmdPacket packet;
            packet.BeginWrite();
            packet.WriteByte(MSG_FILE_NOTFOUND);
            packet.EndWrite();
```

```
            SendPacket(&packet);
        }
    }
    else if(btMsgType == MSG_FILE_BEGIN_OK)//消息 客户端已经收到文件开始传输的通知
    {
        char * pszFileName;
        packet -> ReadString(&pszFileName);
        if(strcmp(pszFileName,_szCurFileName) == 0)//文件名校验
        {
            SendingFile();//开始发送文件内容
        }
        else//文件名不同,表示出错了
        {
            Log("MSG_FILE_BEGIN_OK Comfirm File Name Err? hn");
        }
    }
    else if(btMsgType == MSG_FILE_CONTENT_OK)//消息 客户端成功接收了一段文件内容
    {
        long lReadLoc;packet -> ReadLong(&lReadLoc);
        if(lReadLoc == _nReadLoc)//文件读取位置校验
        {
            SendingFile();//发送新的文件内容
        }
    }
    else if(btMsgType == MSG_FILE_END_OK)//消息 客户端收到文件接收已经结束
的通知
    {
        SetWaitingState(STATE_WAIT_NOTHING);//进入空闲状态
    }
}
//功能 检查文件是否存在,并且开始发送文件
bool CClientSocket::BeginSendFile(char * pszFile)
{
    if(_fp != NULL)//如果句柄没有关闭,则清掉
    {
        fclose(_fp);
        _nReadLoc = 0;
        _nReadSize = 0;
    }
```

```
    string strFileName = theApp. m_szWorkingDir;
    if(strFileName[strFileName. size() - 1] != '/')
    {
        strFileName + = "/";
    }
    //文件实际所在目录 = 服务器的当前工作目录 + 文件名(包含相对路径)
    strFileName + = pszFile;
    _fp = fopen(strFileName. c_str(),"rb");
    if(_fp == NULL)//文件打开失败,返回错误
    {
        Log("Open File < %s > Failed! \n",strFileName. c_str());
        return false;
    }
    strcpy(_szCurFileName,pszFile);//记录当前正在处理的文件名
    Log("Open File < %s > \n",pszFile);
    _SendMsgFileBegin(pszFile);//通知客户端,文件传输开始
    return true;
}
//功能:发送一段文件内容
bool CClientSocket::SendingFile()
{
    if(_fp != NULL)
    {
        _nReadSize = fread(_btReadBuf,1,_nReadBufSize,_fp);//读取文件内容到缓冲中
        _nReadLoc = ftell(_fp);
        if(_nReadSize < = 0)//文件已经结束
    {
            Log("File Send End! \n");
            EndSendFile();
    }
    else
    {
            return_SendMsgFileContent();//发送文件内容
    }
}
    return false;
}
//功能:文件已经传输完毕,通知客户端
```

```
void CClientSocket::EndSendFile()
{
    if(_fp != NULL)
    {
        fclose(_fp);
        _fp = NULL;
        _nReadSize = 0;
        _SendMsgFileEnd();//发送消息
    }
}
//功能:发送消息→文件传输开始
void CClientSocket::_SendMsgFileBegin(char * pszFile)
{
    int nFileSize = Util_GetFileSize(_fp);
    CMFCmdPacket packet;
    packet.BeginWrite();
    packet.WriteByte(MSG_FILE_BEGIN);
    packet.WriteString(pszFile);
    packet.WriteLong(nFileSize);
    packet.EndWrite();
    SendPacket(&packet);
    SetWaitingState(STATE_WAIT_BEGIN_OK);//进入等待回应状态
}
//功能:发送消息→文件内容
bool CClientSocket::_SendMsgFileContent()
{
    CMFCmdPacket packet;
    packet.Begin Write();
    packet.WriteByte(MSG_FILE_CONTENT);
    packet.WriteLong(_nReadLoc);
    packet.WriteBinary(_btReadBuf,_nReadSize);
    packet.EndWrite();
    int r = SendPacket(&packet);
    SetWaitingState(STATE_WAIT_CONTENT_OK);//进入等待回应状态
    return true;
}
//功能:发送消息→文件结束
void CClientSocket::_SendMsgFileEnd()
{
```

```
        CMFCmdPacket packet;
        packet. Begin Write();
        packet. WriteByte(MSG_FILE_END);
        packet. EndWrite();
        SendPacket(&packet);
        SetWaitingState(STATE_WAIT_END_OK);//进入等待回应状态
    }
    //功能:流程控制主循环
    void CClientSocket::Run()
    {
        Process();//socket 内部处理
        //等待状态下的计时和重发处理
        if(_nWaitingState == STATE_WAIT_BEGIN_OK)
        {
            if(_nRunTick == theApp. m_nResendTick)
            {
                Log("Try Send file begin again\n");
                _SendMsgFileBegin(_szCurFileName);
        }
    }
    else if(_nWaitingState == STATE_WAIT_CONTENT_OK)
    {
        if(_nRunTick == theApp. m_nResendTick)
        {
            Log("Try Send Content again\n");
            _SendMsgFileContent();
        }
    }
    else if(_nWaitingState == STATE_WAIT_END_OK)
    {
            if(_nRunTick == theApp. m_nResendTick)
            {
                Log("Try Send file end again\n");
                _SendMsgFileEnd();
            }
        }
        _nRunTick + +;
    }
```

下面是应用程序类 CServerApp **的代码。**

ServerApp. h：

```
#ifndef CLASS_SERVERAPP
#define CLASS_SERVERAPP
#include" ClientSocket. h"
using namespacestd；
class CServerApp
 {
public：
      CServerApp()；
      //下面是服务器启动所需要的一些配置信息
      int                    m_nPort；//端口
      char                   m_szWorkingDir[255]；//当前工作目录,文件读取都是以此
                                                  //目录为相对路径
      char                   m_szBindIP[64]；     //绑定 IP
      int                    m_nResendTick；      //重发消息计时
      int                    m_nSleep；           //主循环休息间隔
      CServerSocket          m_ListenSocket；     //用于监听的 Socket
      list < CClientSocket * > m_ClientSocketList；
      list < CClientSocket * > m_DisconnectList；
      bool Init()；
      int  Run()；
      void RemoveDisconnect()；
      bool LoadConfig(char * pszFileName)；
}；
extern CServerApp theApp；
#endif
```

ServerApp. cpp：

```
#include" ServerApp. h"
#include < fstream. h >
CServerApp theApp；
CServerApp：:CServerApp()
 {
      m_nPort = 0；
      strcpy(m_szWorkingDir," NULL")；
}
```

```
bool CServerApp::Init()
{
    Log("File Server Start\n");
    //用指定的 IP 和端口初始化
    m_ListenSocket.Initialize(PROTOCOL_TCP,m_szBindIP,m_nPort,true);
    m_ListenSocket.Listen();//开始监听
    Log("Begin Listening Port = [%d]...\n",m_nPort);
    return true;
}
//应用程序主体循环
int CServerApp::Run()
{
    m_ListenSocket.Process();//Socket 内部循环
    list<CClientSocket*>::iterator it = m_ClientSocketList.begin();
    while(it != m_ClientSocketList.end())
    {
        (*it)->Run();//依次处理每个连接
        it++;
    }
    RemoveDisconnect();//清除断开的连接
    return 1;
}
void CServerApp::RemoveDisconnect()
{
    list<CClientSocket*>::iterator it = m_DisconnectList.begin();
    while(it != m_DisconnectList.end())
    {
        CClientSocket *pSocket = (*it);
        m_ClientSocketList.remove(pSocket);
        delete pSocket;
        it++;
    }
    m_DisconnectList.clear();
}
//读取配置文件
bool CServerApp::LoadConfig(char *pszFileName)
{
    char szLine[1024];
```

```
    string strLine;
    ifstream in(pszFileName);
    if(in.is_open() == 0)
     {
        Log("Load Config Failed\n");
        return false;
    }
    string strPair[2];
    while(!in.eof())
     {
        in.getline(szLine,1024);
        strLine = szLine;
        Util_ResolveTextLine(strLine.c_str(),strPair,2,'=');
        if(strPair[0] == "directory")
         {
            strcpy(m_szWorkingDir,strPair[1].c_str());
        }
        else if(strPair[0] == "bind_ip")
         {
            strcpy(m_szBindIP,strPair[1].c_str());
        }
        else if(strPair[0] == "port")
         {
            m_nPort = Str2Int(strPair[1]);
        }
        ...
    }
    in.close();
    Log("\n");
    return true;
}
```


main.cpp：

```
#include"ServerApp.h"
int main(int argc,char * argv[])
 {
    if(!theApp.LoadConfig("ups.cfg"))//读取配置文件,格式为 key = value
     {
```

```
        Log("UpdateServer.cfg not found,Server exit\n");
        return 0;
    }
    theApp.Init();
    int r=1;
    while(r)
    {
        r=theApp.Run();
        Util_Sleep(theApp.m_nSleep);
    }
    return 0;
}
```

13.3 客户端功能

至此,已经实现文件版本管理和文件传输服务器,剩下最后一步也就是客户端功能,主要是版本比对、接受文件传输并更新。

更新流程为:提出文件发送请求,向服务器索取版本信息文件;接受版本信息文件之后,进行版本比对,生成更新列表;根据更新列表的内容,依次请求文件传输,并更新。

除此之外客户端还有重建客户端所有文件版本并生成版本信息文件的功能。另外,从界面上看,它还负责显示更新进度以及游戏网站上的关于游戏最近更新的信息。

在这里,把这些与自动更新有关的功能封装到一个类里面,这个类要既能满足上面的需求,而且在设计上力求与要更新的具体内容以及界面如何显示无关。因此,无论哪个网络游戏甚至其他的应用软件,都可以把这个类作为一个模块加入到自己的应用中。下面来看具体代码实现和说明。

UpdateMgr.h:

```
#include"ClientSocket.h"
//注意:这个文件包含的客户端用到的 Socket 连接类,虽然文件名字与上面列出的 Server 上用的相同,但实质是不同的
#include"FileVersion.h" //包含文件版本管理类
class CUpdateMgr
{
public:
        //客户端可以处于两种功能状态
        stati cconst int STATE_UPDATE;//更新状态
        static const int STATE_BUILD;  //重建版本状态这项功能是在客户端的
```

```
        //版本信息文件遭到损坏或不正确的修改时使用
protected:
        char                _szVer[64];              //版本信息文件自身的 MD5 值
        int                 _nState;                 //状态
        int                 _nConnectTick;           //连接服务器超时计数
        CClientSocket *     _pSocket;                //保持连接的 Socket 指针
        float               _fSpeed;                 //当前更新的速度
        bool                _ProcessFileUpdateList(); //处理比对后产生的更新列表
        bool                _CheckUpdate();          //检查是否需要更新
        //客户端重建版本时所用到的相关数据
        vector < string >       _FileList;           //文件列表
        list < SFILE_INFO >     _FileInfoList;       //文件信息列表
        int                     _nBuildCnt;          //当前已经处理的数量
        char                    _szBuildTagFile[255]; //引发重建版本操作的标志文件
        //更新或重建版本的时候,可以提供给该类的使用者的对外接口虚函数
        //具体用途请参看 UpdateMgr. cpp
        virtual void    _Connected();
        virtual void    _NothingUpdate();
        virtual bool    _GetUpdate();
        virtual void    _FinishUpdate();
        virtual bool    _InitOK();
        virtual void    _BeginQueryFile(SFILE_INFO * pFileInfo);
        virtual void    _ReceiveContent(bool bVersionFile);
        virtual void    _UpdateSpeed(float fSpeed);
        virtual void    _ConfigNewPair(string strKey,string strValue);
        virtual void    _RebuildVersionReady(vector < string > * pFileList);
        virtual void    _RebuildVersionEnd();
        virtual bool    _GenerateFileVersion(const char * pszFile,constchar * szMD5);
        virtual void    _BeginRcvFile(int nFileSize);
public:
        CUpdateMgr();
        ~CUpdateMgr();
        bool        Init();
        bool        m_bTryConnect;  //尝试连接的标志
        float       m_fRcvSizeCnt;  //接收文件内容的尺寸计数,用来计算传输速度
        bool        m_bCalSpeed;    //是否开始计算传输速度
        int         m_nLastTime;    //上次取速度值的时间,用来计算传输速度
        void        TryConnect(int nFlag);
```

```
        void        Connected();
        //开始接收文件传输
        void        BeginSendFile(char * pszFileName,int nFileSize);
        //正在接收文件传输
        void        OnRcvingFile(char * pszFileName,int nUpdateSize);
        //结束文件传输
        void        RcvFileEnd(char * pszFileName);
        int         m_nUpdateCnt;           //更新文件总数量
        int         m_nUpdateSize;          //更新文件总大小
        int         m_nCurUpdateSize;       //当前已更新大小
        int         m_nCurFileUpdateSize;   //当前文件已更新大小
        //重建版本的有关处理
        CFileVersionMgr  m_FileVerMgr;      //版本管理器
        vector < SFILE_INFO > m_UpdateList; //更新列表
        bool        CheckRebuildVersion();  //检查是否要重建版本
        void        SetState(int nState);   //设置状态
        //配置文件里的内容形式是关键字 = 值
        //这个结构体用来以字符串的形式保存读入的信息
        struct SKeyPair
        {
            string strKey;
            string strValue;
        };
        list < SKeyPair > m_ConfigPairList;//从配置文件读入的信息列表
        int         m_nOverTick;            //连接服务器超时时间
        vector < string > m_ServerList;     //服务器 IP 列表
        int         m_nCurServerNo;         //当前使用的服务器编号
        string      m_strRunApp;            //启动应用程序的路径和名称
        list < string > m_strParamList;     //启动参数
        string      m_strUrl;               //网站链接
        string      m_strInfoUrl;           //网站链接 Url:更新信息
        string      m_strBBSUrl;//网站链接 Url:BBS 论坛
        bool        LoadConfig(char * pszFileName);//读取配置文件
        bool        LoadVer(char * pszFileName,char * pszVer);//读取版本文件
        bool        RunAll();//流程主循环
};
inline void CUpdateMgr::SetState(int nState)
{
```

```
    _nState = nState;
}
```

UpdateMgr. cpp：

```
#include < direct. h >
#include < fstream >
#include" updatemgr. h"
#include" FileVersion. h"
using namespace std;
const int CUpdateMgr::STATE_UPDATE = 0;
const int CUpdateMgr::STATE_BUILD = 1;
CUpdateMgr::CUpdateMgr()
 {
    m_nCurServerNo = 0;
    m_bTryConnect = false;
    m_strRunApp = "" ;
    _fSpeed = 0. 0f;
    _nState = STATE_UPDATE;
    m_nOverTick = 400;
    m_nUpdateCnt = 0;
    m_nUpdateSize = 0;
    m_nCurUpdateSize = 0;
    m_nCurFileUpdateSize = 0;
    _nBuildCnt = 0;
    strcpy(_szBuildTagFile,"rebuild");
    strcpy(_szVer,"null");
    _pSocket = newCClientSocket(this);//创建与 Server 连接的 Socket
}
CUpdateMgr::~ CUpdateMgr()
 {
    _pSocket - > Reset();
    delete_pSocket;
}
bool CUpdateMgr::Init()
 {
    if(!LoadConfig("update. cfg"))//读取配置文件
     {
        Log("Config File Not Found!,Exit Update! hn","ERR",0);
```

```
            return false;
        }
        m_FileVerMgr.LoadFileVersionInfo("verinfo.rec");//读取版本信息文件
        LoadVer("ver.rec",_szVer);//读取版本信息文件自身的 MD5 值记录
        if(_InitOK())
        {
            return true;
        }
        return false;
}
//尝试连接当前选中的服务器,nFlag=0 表示首次连接
void CUpdateMgr::TryConnect(int nFlag)
{
        srand(GetTickCount());
        int nServerCnt = m_ServerList.size();
        if(nServerCnt == 0) return;
        _nConnectTick = 0;
        string strServer;
        _pSocket->Close();
        if(nFlag == 0)//随机选中服务器列表中的一个来连接
        {
            int nServerNo = rand()%nServerCnt;
            strServer = m_ServerList[nServerNo];
            _pSocket->Connect((char*)(strServer.c_str()),8088);
        }
        else
        {
            if(m_nCurServerNo < (m_ServerList.size() - 1))//换下一个 Server 来连接
            {
                m_nCurServerNo++;
            }
            else
            {
                m_nCurServerNo = 0;
            }
            strServer = m_ServerList[m_nCurServerNo];
            _pSocket->Connect((char*)(strServer.c_str()),8088);
        }
```

```
    Log("\nTry Connect IP = [%s]\n",strServer.c_str());
}
//已经连接上 Server 的处理
void CUpdateMgr::Connected()
{
    Log("\nConnected ip = [%s]\n",m_ServerList[m_nCurServerNo].c_str());
    _Connected();
    m_nUpdateCnt = 0;
    _pSocket->SendMsgRequestFile("ver");
}
void CUpdateMgr::BeginSendFile(char* pszFileName,int nSize)
{
    _BeginRcvFile(nSize);
    m_nCurFileUpdateSize = 0;
}
//开始接收一个文件的处理
void CUpdateMgr::OnRcvingFile(char* pszFileName,int nUpdateSize)
{
    m_nCurFileUpdateSize += nUpdateSize;
    if(strcmp(pszFileName,"VerComp.dat") == 0)//如果接收到版本信息文件
    {
        _ReceiveContent(true);
    }
    else if(strcmp(pszFileName,"ver") =0)//如果接收到版本信息文件自身 MD5 值的记录文件
    {
        _ReceiveContent(true);
    }
    else
    {
        m_nCurUpdateSize += nUpdateSize;
        _ReceiveContent(false);
    }
}
//接收文件结束
void CUpdateMgr::RcvFileEnd(char* pszFileName)
{
    if(strcmp(pszFileName,"ver") == 0)//版本文件的 MD5 值
    {
```

```
        if(_CheckUpdate()) //检查版本文件是否发生变化
        {
            Log("VerComp.dat is different,Need Query VerComp.dat\n");
            _pSocket->SendMsgRequestFile("VerComp.dat");
    }
    else //无需更新
    {
            _NothingUpdate();
        }
    }
    else if(strcmp(pszFileName,"VerComp.dat")==0) //版本文件
    {
        Log("\nGet Version Info OK!\n");
        if(_ProcessFileUpdateList())
        {
            if(_GetUpdate())
            {
                SFILE_INFO* pInfo=&(m_UpdateList[0]);
                _pSocket->SendMsgRequestFile(pInfo->szID);
                _BeginQueryFile(pInfo);
                m_nUpdateCnt++;
            }
        }
        else
        {
            CopyFile("ver","ver.rec",FALSE);
            DeleteFile("ver");
            _NothingUpdate();
        }
    }
    else //要更新的文件
    {
        //Update File Version
        SFILE_INFO* pInfo=&(m_UpdateList[m_nUpdateCnt-1]);
        m_FileVerMgr.UpdateFileVersion(pInfo,TRUE);
        if(m_nUpdateCnt>=m_UpdateList.size())
    {
            //更新完毕,版本文件的 MD5 记录也更新
```

```
                CopyFile("ver","ver.rec",FALSE);
                DeleteFile("ver");
                _FinishUpdate();
            }
            else
            {
                pInfo = &(m_UpdateList[m_nUpdateCnt]);
                _pSocket->SendMsgRequestFile(pInfo->szID);
                _BeginQueryFile(pInfo);
                m_nUpdateCnt++;
            }
        }
    }
    //检查版本信息文件的 MD5 值是否发生变化
    //说明:在前面介绍的更新流程中,实际上在最前面还加入一个过程,就是获得版本信息
    //文件自身的 MD5 值,而这个值也是以文件的形式存放在服务器的工作目录里,之所以要这么
    //做,是因为在客户端有上千个文件的情况下,版本信息文件自身的尺寸也达到了数百 KB 之
    //多,对于非宽带用户,每次启动游戏要去取一个几百 KB 的文件,就要等待很久。为了提高速
    //度和效率,首先通过检查 MD5 值来确定版本信息文件自身有没有发生变化,如果有,则取得
    //版本信息文件,否则直接进入游戏,。这样,在没有任何文件更新的情况下,用户就可以
    //很快进入游戏。
    bool CUpdateMgr::_CheckUpdate()
    {
        char szNewVer[64];
        if(!LoadVer("ver",szNewVer))
        {
            return true;
        }
        if(strcmp(szNewVer,_szVer)!=0)//版本文件并没有更新,无需取更新文件
        {
            return true;
        }
        return false;
    }
    //把取得的版本信息文件和现有文件版本做比对,找到版本不同的文件,生成一个列表
    bool CUpdateMgr::_ProcessFileUpdateList()
    {
        m_FileVerMgr.GenerateUpdateList("VerComp.dat",&m_UpdateList,&m_nUpdateSize);
```

```
    if(m_UpdateList.size() == 0)
    {
        return false;
    }
    return true;
}
//读取版本信息文件自身 MD5 值的记录文件
bool CUpdateMgr::LoadVer(char * pszFileName,char * pszVer)
{
    ifstream in(pszFileName);
    if(in.is_open() == 0)
    {
        Log("ver for vercomp.dat not found!\n");
        return false;
    }
    char szLine[64] = "null";
    in.getline(szLine,64);
    in.close();
    strcpy(pszVer,szLine);
    return true;
}
//读取配置文件,格式为 key = value,通过虚函数,继承者可以写自己的处理过程
bool CUpdateMgr::LoadConfig(char * pszFileName)
{
    char szLine[1024];
    string strLine;
    ifstream in(pszFileName);
    if(in.is_open() == 0)
    {
        Log("Load Config Failed\n");
        return false;
    }
    string strPair[2];
    while(!in.eof())
    {
        in.getline(szLine,1024);
        strLine = szLine;
        Util_ClearStrEndline(strLine);
```

```
            Util_TrimString(strLine);
            if(strLine.size() >0)
             {
                Util_ResolveTextLine(strLine.c_str(),strPair,2,'=');
                SKeyPair Pair;
                Pair.strKey = strPair[0];
                Pair.strValue = strPair[1];
                if(strLine.substr(0,2) == "//") //注释
                 {
                    continue;
                }
                _ConfigNewPair(strPair[0],strPair[1]);//每读到一对 key=value,就调用此函数
            }
        }
        in.close();
        Log("\n");
        return true;
}
//主体循环
bool CUpdateMgr::RunAll()
 {
        if(_nState == STATE_BUILD)//如果当前处于重建客户端所有文件版本的状态
         {
            if(_nBuildCnt == 0)        {
                _RebuildVersionReady(&_FileList);//重建开始
            }
            if(_nBuildCnt < _FileList.size())
             {
                string strFile = _FileList[_nBuildCnt];
                char szDigest[33];MD5String((char*)strFile.c_str(),szDigest);
                if(!_GenerateFileVersion(strFile.c_str(),szDigest))//生成 MD5 值
                 {
                    return false;
                }
                SFILE_INFO info;
                info.dwSize = 0;
                info.cFlag = CFileVersionMgr::OP_FLAG_VALID;//文件操作标志设为有效
                strcpy(info.szID,strFile.c_str());
```

```
            strcpy(info.szVersion,szDigest);
            _FileInfoList.push_back(info);
            _nBuildCnt++;
        }
        else//版本重建完成,写入文件
        {
            //write file version date to file
            FILE* fp = fopen("verinfo.rec","wt");
            fprintf(fp,"NULL\n");
            for(list<SFILE_INFO>::iterator it = _FileInfoList.begin();it != _FileInfoList.
            end();it++)
            {
                SFILE_INFO* pInfo = &(*it);
                fprintf(fp,"%s,%s,0\n",pInfo->szID,pInfo->szVersion);
            }
            fclose(fp);
            remove(_szBuildTagFile);
            m_FileVerMgr.LoadFileVersionInfo("verinfo.rec");
            _RebuildVersionEnd();
            SetState(STATE_UPDATE);
            strcpy(_szVer,"NULL");//需要再次获得版本文件
            TryConnect(0);
        }
    }
    if(_nState == STATE_UPDATE)//当前处于更新状态,按更新流程运作
    {
        _pSocket->Process();//调用 Socket 类的循环处理
        if(m_bTryConnect)//连接服务器
        {
            TryConnect(1);
            m_bTryConnect = false;
        }
        if(_pSocket->m_bConnect == false)
        {
            _nConnectTick++;
            if(_nConnectTick > m_nOverTick)
            {
                _nConnectTick = 0;
```

```
                TryConnect(1);
            }
            Log(".");
        }
        if(m_bCalSpeed)//计算更新速度
        {
            DWORD dwTime = GetTickCount();
            if((dwTime - m_nLastTime) >= 1000)//计算传输速度
            {
                _fSpeed = m_fRcvSizeCnt;
                m_fRcvSizeCnt = 0;
                m_nLastTime = dwTime;
                _UpdateSpeed(_fSpeed);
            }
        }
    }
    return true;
}
//检查是否需要进行所有客户端文件版本重建
bool CUpdateMgr::CheckRebuildVersion()
{
    if(!(_access(_szBuildTagFile,0) != -1))//检查 Rebuild 标志是否存在
    {
        return false;
    }
    _FileList.clear();
    ProcessDirectory("",&_FileList,DIRECTORY_OP_QUERY);
    _FileInfoList.clear();
    _nBuildCnt = 0;
    if(_FileList.size() == 0)
    {
        return false;
    }
    SetState(STATE_BUILD);
    return true;
}
```

下面都是可以被继承的虚函数,子类可以做自己的界面显示以及配置文件读取等操作。

```
//UpdateMgr 读取 config 文件,读入 Pair(Key = Value)
void CUpdateMgr::_ConfigNewPair(string strKey,string strValue)
{
    if(strKey == "server_ip") //增加新的 server_ip
    {
        m_ServerList.push_back(strValue);
    }
    else if(strKey == "run") //运行程序的名字
    {
        m_strRunApp = strValue;
    }
        else if(strKey == "web_addr") //网站连接
    {
        m_strUrl = strValue;
    }
    else if(strKey == "overtick") //连接超时
    {
        m_nOverTick = Str2Int(strValue);
    }
    …
    }
//UpdateMgr 已经初始化完毕,包括读取版本文件、连接服务器、发出取新版本请求等
//可以在此处启动界面等相关操作
bool CUpdateMgr::_InitOK()
{
    Log("InitOK!\n");
    //…
    return true;//继续运行
    //return false;//退出
}
//UpdateMgr 连接上 Server
void CUpdateMgr::_Connected()
{
    Log("ServerConnected!\n");
    //…
}
```

```
//UpdateMgr 接收了版本信息发现无需更新
void CUpdateMgr::_NothingUpdate()
{
    Log("No File need Update!\n");
    //…
}
//UpdateMgr 接收了版本信息发现需要更新,更新列表在 m_UpdateList
bool CUpdateMgr::_GetUpdate()
{
    Log("Get File Update!\n");
    //…
    return true;//continue update
    //return false;//don t update
}
//UpdateMgr 开始请求一个新文件
void CUpdateMgr::_BeginQueryFile(SFILE_INFO * pInfo)
{
    Log("\nUpdateFile[%s]:\n",pInfo->szID);
    //…
}
//UpdateMgr 开始接收一个新文件
void CUpdateMgr::_BeginRcvFile(int nnFileSize)
{
    Log("Begin Rcv File Info\n");
}
//UpdateMgr 文件更新中,接收到文件内容
//当前已经更新的总长度为 m_nCurUpdateSize
//当前文件已经更新的尺寸为 m_nCurFileUpdateSize
void CUpdateMgr::_ReceiveContent(bool bVersionFile)
{
    Log(".");
    //…
}
//UpdateMgr 文件更新速度
void CUpdateMgr::_UpdateSpeed(float fSpeed)
{
    Log("Speed Now = %.2f\n",fSpeed);
    //…
```

```
}
//UpdateMgr 更新结束
void CUpdateMgr::_FinishUpdate()
{
    Log("File Update Finish!\n");
    //…
}
//UpdateMgr 重建版本过程,已取得文件列表
void CUpdateMgr::_RebuildVersionReady(vector<string> *pFileList)
{
    Log("Rebuild Ready,File Cnt = %d\n",pFileList->size());
    //…
}
//UpdateMgr 重建版本过程,处理一个新文件
bool CUpdateMgr::_GenerateFileVersion(const char *pszFile,const char *szMD5)
{
    //Log("MD5(%s) = %s\n",pszFile,szMD5);
    //…
    return true;//continue
//return false;//stop
}
//UpdateMgr 重建版本结束
void CUpdateMgr::_RebuildVersionEnd()
{
    Log("Rebuild All File Version OK!\n");
    //…
}
```

应用程序只需要生成一个此类的实例,并调用它的初试化和主体循环,就可以变成具有自动更新功能的程序,并通过继承它的虚函数接口来做自己的更新相关的界面显示。至此,自动更新系统的三大组成部分都已讲述完毕,对于上文列出的代码,只需略加修改,就可运用于自己的应用程序。

第14章 巫师管理工具的设计与实现

14.1 GM 概述

游戏中经常会需要一些特殊的人,在举办一些特殊的活动时这些人会出来带领那些参加活动的人。同样,游戏中也会需要一些维持游戏秩序又能为玩家提供服务的人,如惩罚恶意破坏游戏的人,帮助被困的玩家等。

GM 就是在某些游戏中扮演这种角色的人。GM 为游戏带来更多的活动,使游戏内容更加丰富,同时 GM 也在一定程度上弥补了游戏设计上的缺陷。

通常,GM 在游戏中通过如下 2 种方式提供服务:

①GM 进入游戏和玩家面对面的交流和服务。

这种方式就像现实中的巡警,它能为玩家提供直接的服务。

②用文字的方式为游戏中的玩家提供服务。

这更像现在的热线电话,如你不知道某个公司的电话可以打 114 进行查询。同样,有问题的时候也可以通过游戏提供的请求 GM 帮助的方法来请求帮助。

14.2 如何快速响应用户的服务请求

在现在的游戏中,为玩家提供类似热线电话的服务更加重要。那么如何做到快速反应用户的请求和高效处理请求的服务呢?

要达到快速反应的要求,最快的方法就是为每一个区设置一个在线的 GM。但是这种方式不受欢迎,因为现在的游戏都由很多的区域组成,这样会导致需要一个很大的 GM 团队,如图 14.1 所示。

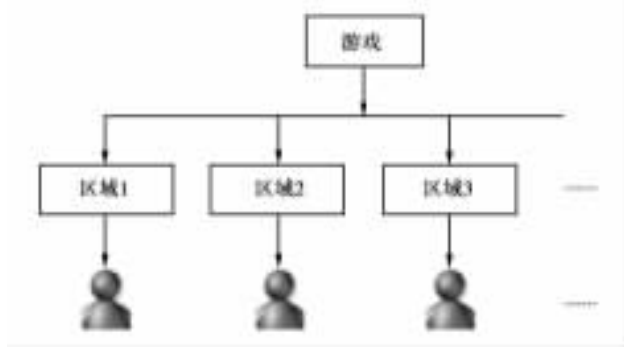


图 14.1

如果 1 个区域里的问题很多,1 个人就会应付不过来。但,如果问题少,那么 GM 的利用率就太低。因此在这个基础上可以做一个改进,使一个 GM 可以同时接收和处理多个区域传来的信息。假设原来每个 GM 都几乎有 2/3 的时间是空闲的,那么就可以每个 GM 划分 3 个区域,这样就能比较充分地利用 GM 的时间。

同时,这种做法还可以比较自由地分配区域以平衡 GM 的工作量。比如说区域 1 是一个村庄,玩家可以在里面买卖道具,补生命或魔法,或者干其他的事情。可能每个玩家都要在这里花 1/4 的时间,这里的玩家可能会有很多问题,而且这里是新玩家出现的地方,会有很多新

的玩家在这里学习如何玩这个游戏，所以这个区域的 GM 可能很忙；而区域 2 ~ 10 都是冒险区域，而且都比较大，这里的玩家的问题也相对少，那么这些区域的 GM 就比较空闲。在这种情况下，可以把区域 1 分配给 GM1，而区域 2、区域 3、区域 4 都分配给 GM2，这样就可以在一定程度上平衡 GM 的工作量，如图 14.2 所示。

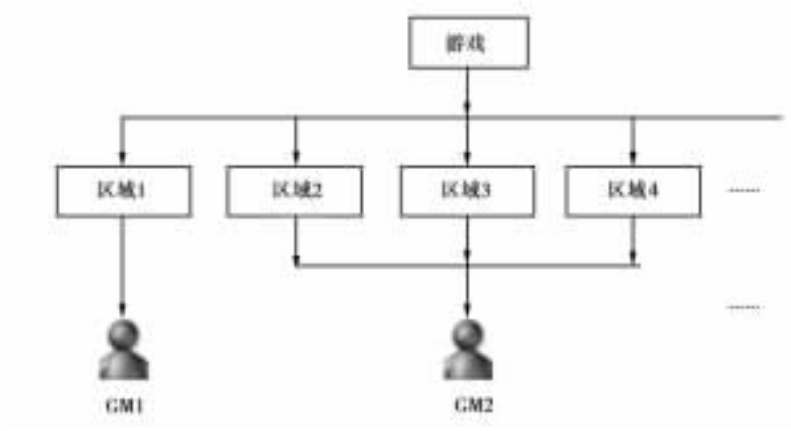


图 14.2

但是，当区域 1 的问题太多 GM1 处理不完的时候，这种方法没有办法把区域 1 的问题分一部分出去，因为区域与 GM 的关系是多对一的关系，在个别的时候就会导致对玩家的响应不够及时。

还有一个问题就是，由于区域与 GM 的关系是多对一的关系，就会导致当 GM 有事临时离开的时候就没有办法响应相应玩家的请求。

要改变这种状况就要改变区域与用户之间的关系。也就是说由原来的多对一关系变成多对多关系，如图 14.3 所示。

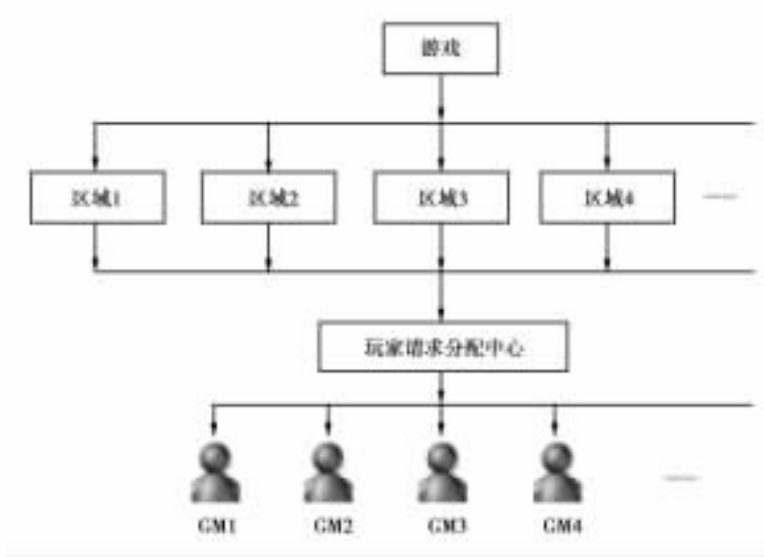


图 14.3

可以看到,玩家请求分配中心起到至关重要的作用。下面就来讨论这个分配中心,如何让它更好地工作。

14.3 玩家请求分配中心

分配中心的工作就是要合理分配玩家的请求给 GM,因此它需要有识别 GM 当前任务状况的能力。当一个玩家请求到分配中心的时候,就会去检查哪个 GM 处于空闲或者比较空闲的状态,然后把玩家的请求发送给比较空闲的 GM。

当没有 GM 在线的时候,为了及时地响应用户,这个分配中心就要有自动回复的功能,如没有 GM 在线时回复玩家说目前没有 GM 在线。

如果所有的 GM 都是忙的时候,分配中心接收到玩家的服务请求,如图 14.4 所示。

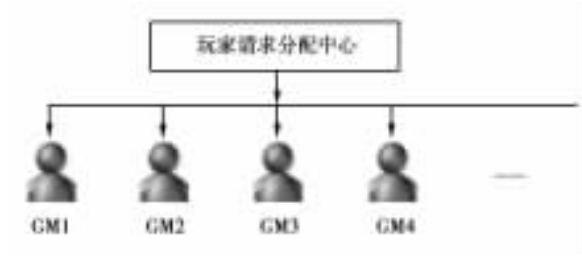


图 14.4

假设目前所有的 GM 都处于忙状态,同时分配中心又接收到玩家的请求。如果这时分配中心把这个请求发送给 GM1,而 GM1 可能正在处理一件棘手的事情,但 GM2 在一小段时间后就处理掉一些工作处于空闲状态了。很显然,这次的分配工作是不恰当的。

要解决这个问题,只需要引入玩家请求缓冲区即可。当有玩家请求进来的时候,如果此时 GM 都处于忙状态就回复玩家说请求进入排队系统,请耐心等待。同时把玩家的请求加入分配中心的请求缓冲区,等有 GM 处于空闲状态的时候就把请求发送过去,如图 14.5 所示。

14.3.1 判断 GM 处于空闲状态

在上面的请求响应中,分配中心分配请求的依据就是 GM 是否处于空闲状态。那么如何判断 GM 处于空闲状态呢?不同的游戏可能会需要不同的处理。当然,最简单的方法就是判断当前 GM 处理多少个玩家的请求,比如说可以设定当 GM 处理玩家请求的个数小于等于 10 就处于空闲状态,大于 10 就处于忙状态。

14.3.2 GM 空闲状态的玩家请求分配

如果分配中心对每个玩家的请求都是按照 GM 的排列顺序分配,那么对排列在前面的 GM 就不公平,因为它总是接收工作,除非处于忙状态。所以说分配中心也要按照一定的分配原则来分配玩家的请求,使工作尽量平均。这里举一个简单的方法。

建立一个接受玩家请求的指针,然后把所有的 GM 数据指针组织成循环链表。如果从来

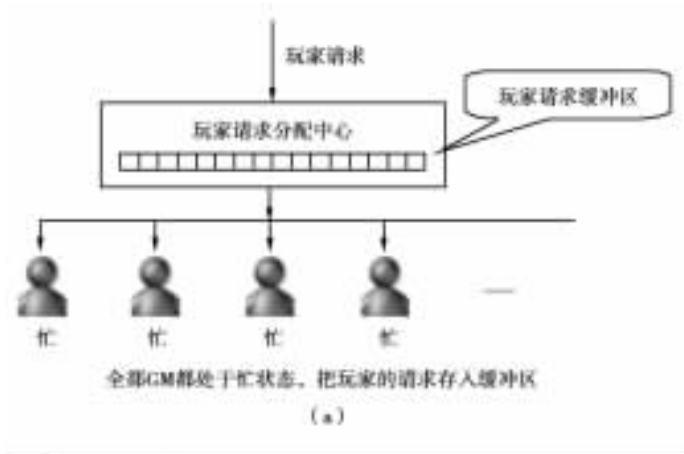


图 14.5

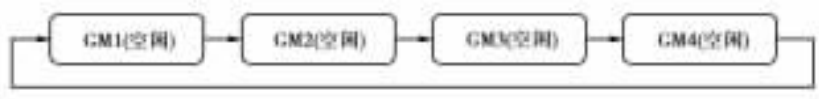


图 14.6

没有玩家请求过服务就设这个指针为空，如图 14.6 所示（假设有 4 个 GM 在处理玩家请求）。

如果这时候有玩家请求过来，分配中心就检查接受玩家请求指针。如果为空就发送请求给位于链表的头部的那个 GM，同时设置接受玩家请求指针指向下一个 GM。

```
if(接受玩家请求指针 == NULL)
 {
     发送请求给链表头部的 GM
     接受玩家请求指针 = 链表头部 GM 的下一个 GM
 }
```

如果有玩家请求并且接受玩家请求指针不为空，就检查这个指针指向的 GM 是否处于忙状态。如果忙就检查下一个 GM，直到发现有空闲的 GM 就把请求发送过去，并且设置接受玩家请求指针指向当前接受请求的 GM 的下一个 GM。如果没有 GM 处于空闲状态，就把这个玩家请求放到分配中心的请求队列中。

```
if(接受玩家请求指针 == NULL)
```

```
    {
        ……
    }
    else
    {
        bool 是否发送;
        pGM = 接受玩家请求指针;
        是否发送 = false;
        do
        {
        if(pGM 处于空闲状态)
        {
            发送请求给 pGM
            接受玩家请求指针 = pGM 的下一个 GM
            是否发送 = true;
            break;
        }
        pGM = pGM 的下一个 GM
    } while(pGM ! = 接受玩家请求指针)
    if(是否发送 == false)
    {
        把请求送到请求队列中。
    }
}
```

图 14.7

如图 14.7 所示的状况下,分配中心就会移动指针,直到找到 GM4 处于空闲状态就把请求发送给 GM4,同时置指针指向 GM4 的下一个 GM,也就是 GM1,状态如图 14.8 所示。

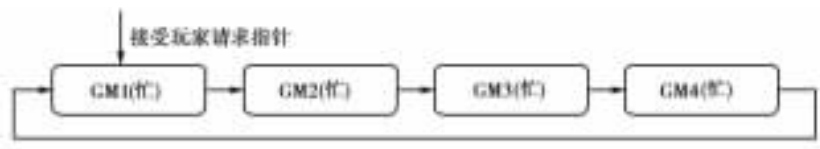


图 14.8

如果上面的那次请求正好使 GM4 处于忙状态,如图 14.8 所示。这时候又来一个玩家请求。分配中心移动指针把所有的 GM 都检查了一次,发现没有 GM 处于空闲状态,就会把这个玩家请求存入请求队列,并且不移动接受玩家请求指针。

14.4 同时处理多个玩家的请求

我们在用 ICQ、OICQ 或者聊天室这种文字聊天方式和别人聊天的时候经常是和多个人一起聊的，GM 与玩家之间的互动也是建立在文字聊天的基础上。和聊天一样，GM 也可以同时处理多个玩家的请求。

如果所有玩家的对话信息都显示在一个屏幕中，如下面的一段文字：

用户 1 我怎么买东西啊

用户 2 我怎么卖东西啊

用户 3 我怎么装备东西啊

用户 4 我怎么卖东西啊

用户 5 我怎么买东西啊

这样的问题多了，GM 就容易混淆不同用户的问题，会回错问题，把用户 1 的答案告诉用户 2。而且当 GM 需要查看前面和玩家的对话的时候很不方便，必须从列表中找出该玩家说过的话。因此这种方法在 GM 同时为多个玩家服务的时候效率很低，出错率也比较高。

所以 GM 使用的软件最好是能像 OICQ 一样与每个玩家的对话都能独立存放和显示。最好一个窗口显示的是同一个玩家的这次请求的所有聊天记录。

下面将举一个界面的例子供读者参考。图 14.9 展示了一个 GM 工具的界面。

图 14.9

这个界面主要分为 2 个部分：一是左边的服务器和用户列表区，二是右边的用户信息区。下面将逐一介绍它们的功能。

(1) 服务器和用户列表区

这里可以列举出服务器和有请求服务的用户，如图 14.10 所示。

图 14.10

服务器分为 3 层:第一层是区域(北京、广州等);第二层是服务器组,一个区域可以有多个服务器组;第三层是服务器,一个服务器组有多个服务器,用户就包含在每个服务器中。

注意:这里显示的玩家是有请求服务并且服务还没有完成的玩家。如果有新的玩家请求服务,就会被添加到这个列表中,GM 完成对玩家的处理该玩家就会从列表中删除。被选中的玩家的信息就会在右边的玩家信息显示区中显示出来。

(2)用户信息区

用户信息区显示了在用户列表区中被选中玩家的信息,由聊天信息区、用户资料信息区、文字输入区组成。

①聊天信息区:聊天信息区显示了玩家这次服务的所有聊天信息。这里显示的信息就只有这个玩家和 GM 的聊天信息,每次选中不同的玩家就会显示对应的玩家的聊天信息。所以 GM 要察看聊天的记录是一件很方便的事情。

②用户资料信息区:在聊天信息显示窗口下面有 2 个标签,如果选中资料标签,上面的窗口就会显示用户的资料;如果选中的是聊天记录标签,上面窗口显示的就是聊天记录。具体的用户资料随着游戏的不同而不同,这里不再赘述。

③文字输入区:文字输入区就是 GM 和当前玩家对话文字输入的地方,有一个文字输入窗口和一个发送按钮。

由于玩家提出的问题重复率会非常高,为此准备了一个快速回复的方法,如右图所示。在发送按钮旁边添加一个快速文字按钮,点击这个按钮会弹出一个菜单,上面写了一些常用的文字给 GM 选择,GM 可以选择其中的文字快速地回复玩家。

14.5 GM 的管理

GM 的服务质量直接关系到这个游戏的服务质量,所以要保证游戏的服务质量,就必须保证 GM 的服务质量。必须找到一种方法来管理和监督 GM 的工作,以保证 GM 的服务质量。

14.5.1 日志文件

建立一种 GM 服务日志来记录 GM 和玩家交互的整个过程。记录的内容包括:

①玩家的请求服务时间和内容。

②GM 开始响应请求的时间。

③GM 和玩家的所有谈话内容和时间。

④GM 对玩家的操作指令和操作时间。

⑤GM 完成玩家请求时间。

另外,建立一种 GM 登陆日志来记录。记录的内容包括:

①GM 登陆的时间。

②GM 登出的时间。

③GM 暂停服务的时间。

④GM 恢复服务的时间。

用以上 2 个日志就可以追踪 GM 的服务过程,并且可以做很多统计,如统计每小时平均的处理量、每天玩家请求的个数等。甚至可以用统计的数据来作为评定 GM 绩效的一项标准。

由于玩家和 GM 的互动都要通过玩家请求分配中心,而且 GM 的所有操作也要通过这个分配中心,所以这些日志可以放到分配中心进行记录,如图 14.11 所示。

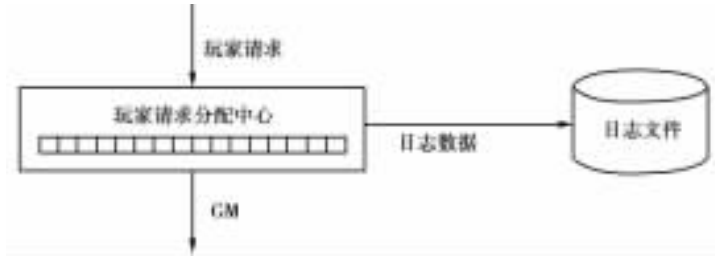


图 14.11

在玩家请求分配中心生成日志后就把它送到一个专门的日志服务器中,由日志服务器负责存入日志文件中。

14.5.2 玩家的反馈和申诉

在游戏中,可以提供给玩家反馈游戏 bug 或者其他信息的功能,这样有助于玩家帮助改善游戏的功能。同样,也可以提供申诉 GM 服务质量的功能。这样,当一个玩家不满意 GM 的服务时可以申诉,这个申诉信息可能会发给 GM 的主管,这样能帮助 GM 主管了解 GM 的工作。

14.5.3 GM 的操作安全

GM 在游戏中通常有一些超出普通玩家的权限,比如说 GM 可以察看并修改一个指定玩家的属性信息。由于不同部门的人可能需要不同的功能,即使都是 GM,也有可能因为职责不同而需要不同的功能。为了安全和管理方便,可以为每个提供的功能都设定一个权限标志位,然后在每个 GM 的个人资料中添加一个表示 GM 权限的标志。这样就可以为不同的 GM 设定不同的权限。

例如,设定功能的权限标志位如表 14.1 所示:

表 14.1

功　　能	权限标志位
察看玩家属性信息	0x00000001
察看玩家道具信息	0x00000002
修改玩家属性信息	0x00000004
修改玩家道具信息	0x00000008
重新启动服务器	0x10000000
关闭服务器	0x20000000

注意 标志位是按位表示的。

那么,可以设置权限如表 14.2 所示:

表 14.2

GM	权 限 描 述	权 限 标 志
一般 GM	只要察看属性和道具	0x00000003
高级 GM	需要能察看及修改属性和道具	0x0000000F
负责活动的 GM	需要察看和修改道具	0x0000000A
GM 主管	需要能重新启动和关闭服务器	0xF0000000

由于游戏的功能非常多,其权限标志位也非常多,而需要定义的权限并不多,所以可以做一种默认权限的方式来简化配置权限。

例如,设置 10 个默认权限,每个默认权限都有一个固定的权限配置方案。当默认权限为 0 的时候就表示自定义权限,其他的权限标志位才起到实际用户的作用。这样在配置权限的时候通常只要从 10 个默认权限中选择一个即可。这样,在人员流动比较大的时候能简化配置的操作,减少操作错误。

第15章 即时战略网络对战功能的设计与实现

15.1 功能需求

(1)即时战略网络游戏概述

游戏类型:即时战略游戏(RTS:Real-Time Strategic Game)。

游戏背景:古代士兵和冷兵器为主。

游戏方式:玩家扮演指挥者,通过采集、建造、生产等方式壮大军力,与电脑 AI 或其他玩家扮演的指挥者进行战争。

游戏画面:2D,斜 45°视角。

(2)功能需求

下面是需求描述:

①有限数量的玩家,通常为小于等于 8 个,通过局域网、Internet、串口直连等方式可以进行连线对战,要求每个玩家进行游戏的时候游戏状态完全同步、画面表现一致。

②基本连线方式为由一个玩家建立游戏会话,其他玩家加入游戏。在局域网环境中,会自动列出当前创建的游戏名称或玩家名称,其他玩家可以直接选择加入;在 Internet 上,建立游戏的玩家提供自己的 IP 地址给其他玩家,那么其他人可以通过输入 IP 地址来建立连接;如果使用串口相连,方式同局域网环境,但只可以两人进行游戏。

③玩家在创建游戏的时候,可以决定允许加入游戏的最大人数。游戏被创建后,处于等待其他玩家加入的状态,称为预备状态。建立游戏者,称为主机,在预备状态下主机可以踢掉其他加入的玩家,并可以选择地图或场景、设置游戏模式等相关选项。主机所做的这类操作,其他玩家应该马上得到反应。在预备状态下,玩家之间也可以聊天。

④当加入该游戏的玩家都已确认准备好时,主机便可以正式开始游戏。主机如果中途退出或者战败,其他玩家应该可以将游戏继续进行下去,直到剩下最后一个人。

⑤游戏中玩家相互之间可以结盟或设为敌对状态,并以此作为战斗结果的判断依据。战斗进行中,任何玩家改变这类状态,其他人会得到提示。

⑥游戏进行中,玩家之间可以聊天,并设有盟友频道,玩家也可以自行设置其他玩家中哪些人可以接收自己的消息。

⑦游戏的对战方式上,会有正常、遭遇战、夺旗等模式。每种模式的游戏初始状态不同,如拥有基地可以建设发展或只拥有一定数量的军队,胜利条件也不同,可以是全灭敌方,也可以是摧毁敌方特定建筑,将敌方的旗帜带到己方大本营等等。

15.2 程序结构和环境

首先,来了解一下在设计网络对战功能之前,从程序的角度看,这个游戏已经具备哪些元素或者说处于一个怎样的‘环境’中。

15.2.1 游戏循环的概念

一个常见的游戏主文件可能具有如下的形式:

```
main()//主函数
 {
     Game->Init()//初始化游戏
     while(bRun)
      {
          bRun = Game->Run();//游戏循环
      }
     Game->End()//结束游戏
}
```

在 Windows 环境下,它更有可能是如下的形式:

```
win_main()//主函数
 {
     Game->Init()//初始化游戏
     while(bRun)
      {
          MSG msg;
          if(PeekMessage(&msg,NULL,0,0,PM_REMOVE))//检查 Windows 消息循环
           {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
          }
          else//如果消息循环中没有任何消息等待处理
           {
               bRun = Game->Run();//运行游戏循环
          }
     }
     Game->End()//结束游戏
}
```


由此可见,总有一个 Game->Run()来表示游戏的主运行函数,它被放到一个 while 的循环里反复调用,主要有两方面的作用:

①按照一定周期计算和更新游戏内部数据。

②根据游戏数据来刷新画面。

如果这两方面的作用同时被执行,那么可以称这个游戏的画面是同步刷新的,否则就是异步刷新的。为了方便说明网络对战功能的实现,假定游戏画面是同步刷新的。

每调用一次 Game->Run()便执行了一次游戏的循环,而这个循环中会执行很多操作,如

人物向前移动一步、建筑物建造的计时加 1、电脑 AI 做一次计算判断、检查士兵是否升级、某个音效是否播放完毕等等。游戏循环内部的这些操作也都有各自的执行周期,类似于如下的简单形式:

```
Game::Run()
{
    if(nRunCnt%2 == 0) //每 2 次游戏循环,人物往前移动一步
    {
        Man->Move(1);
    }
    if(nRunCnt%3 == 0) //每 3 次游戏循环,更新建筑的建造
    {
        Construction->Building();
    }
    if(nRunCnt%5 == 0) //每 5 次游戏循环,计算一次电脑 AI
    {
        ComputerAI->Run();
    }
    …
    Render();      //更新游戏画面
    nRunCnt++; //游戏循环计数
}
```

显然,游戏循环运行的次数就直接代表了游戏数据所处的状态,仿佛连续照相的快照一般。

15.2.2 玩家输入:键盘和鼠标

在 Windows 环境下,键盘和鼠标消息可以通过检查消息队列获得,如果使用 DirectInput API,也可以查询到键盘和鼠标的状态。玩家对鼠标或键盘的操作会直接引发游戏内部数据更新,可以理解成如下的形式:

```
Game->HandleInput(Key,Mouse);
```

玩家指挥士兵移动、建造建筑或者聊天的命令等都可以通过这项处理完成。

这里需要特别说明的是,通常鼠标和键盘消息的接收与处理与游戏的主循环是放在同一个线程中,当游戏因为计算或画面刷新的原因占用过多的 CPU 时间时,对消息的接收和处理也会很缓慢。经常可以看到游戏速度一旦减慢,连同鼠标箭头的移动速度也减慢了,即时战略游戏常用鼠标来卷动屏幕,这时卷动屏幕的速度也一同减慢。有一种做法是把鼠标消息的接

收与处理放到另外一个线程中，游戏主线程虽然已经变得很慢，鼠标消息线程仍然可以获得一定的 CPU 时间，鼠标移动和卷屏保持正常的速度。这种做法也有缺陷，画面刷新的速度已经变得很慢，鼠标却可以正常移动和卷屏，会造成画面有一定幅度的跳跃。如果认为鼠标保持正常的反应比避免画面刷新的跳跃更重要，就可以选择使用这种方法处理。

15.2.3 地图与坐标系统

整个地图是由逻辑上的方块组成，也可以是棱型块，如图 15.1 所示。地图最大宽度为 256 块，最大高度也是 256 块。第一个地图块位于左上角的坐标为（0，0），最后一个地图块位于右下角的坐标为（255，255），游戏中的遮挡和障碍处理、人物的地图位置记录以及寻路算法都依赖于格子实现。

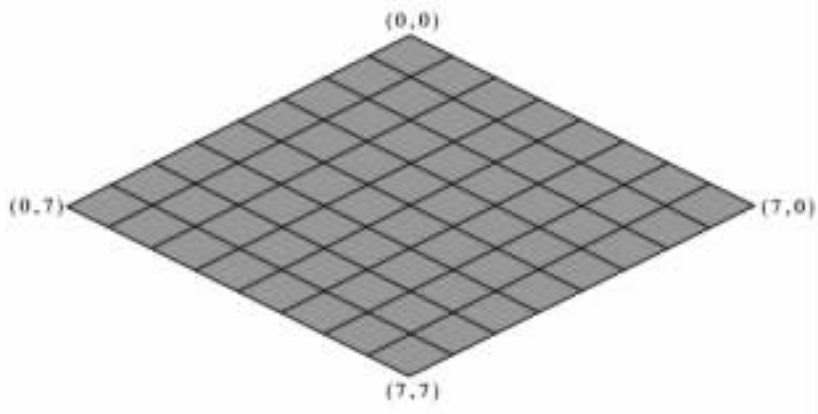


图 15.1　棱形格子组成的地图坐标系统

15.2.4 精灵 Sprite 的概念

人物、建筑、树木、特效（刀光、爆炸）等统称为精灵。它们有共同的属性，如编号、尺寸、当前位置等，也有各自不同的其他属性，例如人物有生命值、攻击速度、所属国家等属性，而特效有作用范围、生存时间等属性。在程序中它们可以拥有的结构如图 15.2 所示。

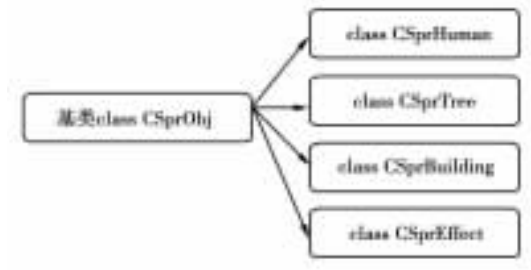


图 15.2

它们存放在各自的数组或者同一个数组的不同段，数组的下标就是它们的编号，例如，SPRITES[4096] 表示游戏中最多有 4 096 个精灵，编号从 0 ~ 4 095，其具体的编号划分如下：

士兵编号:0~1023
建筑编号:1024~2047
树木编号:2048~3071
其他(特效之类):3072~4095

这样的划分可能不是那么具有技巧性,但为了能够方便地说明问题,这里暂且认为游戏中的所有精灵就是这样被存放的。

所以,在游戏循环中,关于精灵的处理可以写成如下的形式:

```
Game::Run()
{
    for(int i=0;i<4096;i++)
    {
        if(Spirte[i])
        {
            Sprites[i]->Run(nRunCnt);//把游戏循环的计数作为参数传给 Sprite
        }
        …
}
```

Sprites[i]可能是人物、建筑、数据、特效等其中任意一种,因为 Run 是虚函数,所以它们调用各自内部的处理方法。

15.2.5 玩家对象的概念

在连线对战的环境中,创建一个代表玩家的对象叫做 GamePlayer,并分配编号。GamePlayer 的属性包括编号、名字、国家、,颜色、连接状况等,例如:

```
class CGamePlayer   //游戏者的定义
{
    strcut SPlayerData
    {
        int     nPlayerID;              //ID
        char    szName[16];             //名字
        char    nOrder;                 //加入游戏的顺序,Order=0 是游
                                        //戏创建者
        bool    bExist;                 //进入游戏设为 true,掉线或退出设为 false
        bool    bReady;                 //是否已准备好
        short   sControlManIDStart;     //可控制的人物起始编号
        short   sGenerateManCnt;        //生产出的人物总数
```

```
        char cColorNo;              //颜色编号
        char cCountryNo;            //国家编号
        char cGroupNo;              //同盟组号
        short sMapViewStartX;       //地图可视起始位置 X 坐标
        short sMapViewStartY;       //地图可视起始位置 Y 坐标
        short sBaseLocX;            //基地初始位置 X 坐标
        short sBaseLocY;            //基地初始位置 Y 坐标
        bool bAccpetChannelMsg;     //是否可以接收当前玩家的聊天信息
        unsigned long dwWaitingTick; //超时等待计数
        unsigned long dwSendCnt;    //消息重发计数
    };
};
```

每个玩家运行的各自的程序中都会包括代表其他玩家的 GamePlayer 对象。如果参加对战的有 8 人,那么程序中会有数组 GamePlayer 来表示参加的所有人,GamePlayer[1]代表创建游戏的玩家然后依次是其他加入的玩家。1 ~ 8 就是 Gameplayer 的编号。例如,当编号为 3 的玩家改变了其颜色标志,从蓝色变为红色,那么其他玩家都会通过网络游戏获得这一更新,程序里的 Gameplayer[3]. cColorNo 都会变为红色。

15.2.6 网络通讯底层

无论使用 WinSock 还是 DirectPlay 或者任何其他网络开发包,需要的基本功能都不会改变,可以认为网络通讯底层被封装好并提供了最方便的接口,例如:

```
SendMessage(TargetID,MsgContent);  //发消息给指定目标
TargetID = 0        表示目标是全体玩家
TargetID = 1 ~ n    表示目标是编号为 n 的玩家
ReceiveMessage(&MsgContent);       //接收消息并把内容存放到 MsgContent
```


15.3 确定实现的基本方案

看完了上述的功能需求,也了解了目前程序现有的“环境”,假设在没有任何设计经验的情况下,从游戏中最基本的画面表现上来做分析:

玩家 A 和玩家 B 分别坐在相邻的两台电脑上,正在进行游戏,地图是一片沙地。玩家 A 处于沙地的左上角,玩家 B 处于右下角。从 A 的游戏画面上可以看到 B 在做什么,B 也可以从自己的游戏画面上看见 A 的任何行动。A 和 B 从游戏开始后还没有做任何操作,只是卷动屏幕来观察地图。此时 A,B 的各自机器上游戏中各项数据还处于相同的状态。如果他们继

续这样下去,什么都不干,一直到有人不小心踢到了电源线导致两台电脑熄火,那么可以说网络对战功能已经大功告成。因为从游戏开始到结束,A 和 B 的状态都是一样的,真是一种理想的状态啊。可是不凑巧,A 终于忍耐不住了,决定选中自己的一个士兵,并点击了沙地的中心,命令士兵向那里移动,而 B 也正好在从自己的画面上观察这个属于 A 的士兵,结果可想而知,A 的画面上这个士兵愉快地接收了 A 的指令,并在寻路算法的指引下,顺利地移动到沙地的中心。而 B 的画面上同样位置这个士兵仍然若无其事地站在那里,还时不时做个俯卧撑什么的。B 看看 A 的屏幕,再看看自己的屏幕,终于忍不住要拍桌子站起来骂人了。这都是因为我们什么都还没有做,不是吗?于是我们首先要做的是让 A,B 机器上士兵的状态保持一致。

很容易便有了如下思路的代码。

对于 A:

```
if(A 指挥士兵移动)
 {
     设置该士兵为移动状态
}
取得所有正处于移动状态的士兵的编号和坐标位置发送给 B
if(士兵停止移动)
 {
     取消移动状态
}
```

对于 B:

```
if(接收到士兵的编号和位置)
 {
     命令该士兵向该位置移动
}
```

于是,B 欣喜地看到,随着 A 指挥那个士兵前进,在自己的画面上的那个士兵也跟着移动起来。而自己选中士兵移动,A 的画面上也能看到。似乎这种思路是正确的,可是 B 并不满足于此,他开始指挥自己的士兵攻击 A 的士兵,士兵被攻击后生命值也应该减少,还没有处理这个新情况,看来士兵只有移动状态是远远不够的,下面为士兵建立一个专用的刷新数据结构:

```
SManUpdate
 {
     int nState;  //基本行为状态 :行走、攻击、死亡等等
     int nFace;  //方向
     int nLocX;  //位置
     int nLocY;
```

```
int nHP；          //生命值
    …              //其他需要更新的属性
}
```

每当士兵的状态或属性发生变化时，便计算更新并发送给其他玩家。

在这种方式中，A 负责自己控制的士兵的有关数值计算，并把结果发送给 B，B 根据结果来做处理和显示，反之亦然。

但源源不断地出现了如下问题：

①B 的士兵攻击 A 的士兵，A 的士兵生命值减少，谁来负责计算，B 还是 A，如果 A 的士兵因此而反击，又是谁来计算。

②B 的士兵攻击 A 的建筑，建筑被毁坏至不同程度，并且建筑也做反击，那又有多少种状态需要刷新，谁来计算建筑的伤害。

③有数百多种士兵、建筑、树木等游戏元素同时处在各种不同的状态，各自触发着不同的功能，累计起来需要有多少数据不断在 A，B 之间来回传递刷新。

④负责计算的对象是不确定的，一旦状态不一致，应该以谁为标准做纠错处理。

思考上述问题，我们意识到，在即时战略游戏中，游戏元素的类别和数量都很庞大，依靠状态刷新来达到各玩家数据同步是不现实的。

现在，换种新的思路来考虑，还是 A 选中士兵并指挥其移动到沙地中心的例子。下面来看看这个过程中 A 的机器上都发生了些什么。

步骤 1：A 用鼠标选择士兵甲，士兵甲被设置为 A 的当前控制对象。

步骤 2：A 在沙地中心点击鼠标左键，点击的位置被记录下来，并设置为士兵甲移动的目标位置。

步骤 3：士兵甲得到目标位置，根据寻路算法，算出自己从目前所在位置到目标位置的所有步数信息。

步骤 4：士兵甲开始按照步数移动，一直到达目标位置后停止。

提取其中的关键信息：士兵甲、目标位置、寻路算法。在玩家 A 的机器上，这 3 样东西实现了 A 的指挥操作；在玩家 B 的机器上，士兵甲、寻路算法同样存在，只是不知道目标位置而已。如果在步骤 2 之后，把士兵甲的编号以及目标位置发送给 B，B 马上开始执行步骤 3 和步骤 4，那么在 B 的机器上便会看到士兵甲做了和 A 机器上同样的事情，走到了沙地中心。A，B 两台机器上最终的位置和方向都相同。原因是士兵甲的移动过程在 A，B 的机器上经过了完全相同的算法的处理。如果中途不受外来操作的干扰，必然产生完全相同的结果。而且在其移动过程中不再需要传递任何消息来刷新状态。但这样的方式是否适用于更复杂的情况呢？下面来看看在玩家 A 的机器上，指挥士兵甲攻击属于 B 的士兵乙的整个过程：

步骤 1：A 用鼠标选择士兵甲，士兵甲被设置为 A 的当前控制对象。

步骤 2：A 对着属于 B 的士兵乙点击鼠标左键，点击的位置被记录下来，并通对坐标计算得知点中的目标是士兵乙。

步骤 3：设置士兵乙为士兵甲的攻击目标，根据游戏的 AI 设定，士兵甲应该按顺序做两件事情：一是移动到士兵乙所在的位置；二是对士兵乙展开攻击。

步骤 4 :士兵甲把士兵乙的位置作为目标位置,根据寻路算法,算出自己从目前所在位置到目标位置的所有步数信息。

步骤 5 :士兵甲开始按照步数移动,一直到达目标位置后停止。

步骤 6 :士兵甲开始对士兵乙攻击。

在经过步骤 2 之后,A 把士兵甲和士兵乙的编号以及要求甲攻击乙的消息发送给 B,B 收到消息之后,马上开始执行步骤 3、步骤 4、步骤 5。那么可以看到 B 的机器上发生了与 A 完全相同的事情。即使当士兵甲对士兵乙造成伤害,士兵乙生命值减少,而且士兵乙开始反击的情况下,A,B 上也做了同样的判断和计算,看到同样的结果,一直到士兵乙不敌士兵甲,倒地身亡,屏幕上的表现都是一致的。这里一个重要的事实就是,A 只对 B 提交了自己的操作,而具体这项操作产生的结果是 A,B 各自计算的,因为算法完全相同的缘故产生了相同的结果。

综上所述,游戏中有地图和精灵,精灵包括游戏中的人物、建筑等。这些人物和建筑有着自己的状态属性,如坐标、方向、生命值等。建立多人游戏,就是要在游戏过程中,保证每个玩家机器上的每个精灵状态一致。

这样就有两种方法可以使用:

①把精灵的状态作为网络消息,在玩家之间传递更新。

②把玩家可控制角色的行为,例如移动、攻击、建造这类直接影响精灵状态的行为作为网络消息,发送给其他人,然后通过玩家机器各自的计算来更新。

在即时战略游戏中,②才是合理的选择。

15.4 定义操作命令

表 15.1

名 称	定 义	数据格式(以 Bit 为最小单位)
行走	MSG_ORDER_WALK	0 ~ 8 消息类型
		9 ~ 32 坐标 X,Y
		33 ~ 40 人物的 ID 号
攻击人	MSG_ORDER_ATTACKMAN	0 ~ 8 消息类型
		9 ~ 24 攻击对象的 ID 号
		25 ~ 32 被指挥的人的 ID 号
攻击建筑	MSG_ORDER_ATTACKBUILD	0 ~ 8 消息类型
		9 ~ 24 攻击对象的 ID 号
		25 ~ 32 被指挥的人的 ID 号
使用魔法	MSG_ORDER_USEMAGIC	0 ~ 8 消息类型
		9 ~ 32 坐标 X,Y
		33 ~ 40 人物的 ID 号
		41 ~ 48 魔法 ID 号
		49 ~ 80 鼠标位置(X,Y)
停止魔法	MSG_ORDER_STOPMAGIC	0 ~ 8 消息类型
		9 ~ 16 人物 ID 号
巡逻	MSG_ORDER_PATROL	0 ~ 8 消息类型
		9 ~ 32 坐标 X,Y
		33 ~ 40 人物 ID 号
……	……	……

已经确定使用传递命令操作的方式来实现网络对战功能,首先要做的就是将玩家的各类操作都转化为游戏内部的命令。这些命令可以被储存、被执行,也可以通过网络消息在不同玩家之间传递。在本实例中,有如下的命令定义:

(1)人物控制命令

人物控制命令如表 15.1 所示。

(2)建筑控制命令

建筑控制命令如表 15.2 所示。

表 15.2

名　　称	定　　义	数据格式(以 Bit 为最小单位)
修建一般建筑	MSG_ORDER_BUILD	0 ~ 8 消息类型
		9 ~ 32 坐标 X,Y
		33 ~ 40 国家 ID + 城市 ID
		41 ~ 48 建筑类型
加派农民	MSG_ORDER_ADDFARMER	0 ~ 8 消息类型
		9 ~ 16 建筑 ID 高 8 位
		17 ~ 24 建筑 ID 低 8 位
加派工人	MSG_ORDER_ADDWORKER	0 ~ 8 消息类型
		9 ~ 16 建筑 ID　高 8 位
		17 ~ 24 建筑 ID　低 8 位
减少工作的人	MSG_ORDER_DECREASE	0 ~ 8 消息类型
		9 ~ 16 建筑 ID　高 8 位
		17 ~ 24 建筑 ID　低 8 位
工作状态切换	MSG _ ORDER _ SWITCHWORK-STATE	0 ~ 8 消息类型
		9 ~ 16 建筑 ID　高 8 位
		17 ~ 24 建筑 ID　低 8 位
拆除建筑	MSG_ORDER_DISMANTLE	0 ~ 8 消息类型
		9 ~ 16 建筑 ID　高 8 位
		17 ~ 24 建筑 ID　低 8 位
修建矿洞	MSG_ORDER_BUILDMINE	0 ~ 8 消息类型
		9 ~ 32 矿洞坐标 X,Y
		33 ~ 56 矿房坐标 Cx,Cy
		57 ~ 64 国家 + 城市
		650 铁矿 1 铜矿
		66 ~ 72 建筑类型
训练士兵	MSG_ORDER_TRAINSOLDIER	0 ~ 8 消息类型
		9 ~ 32 建筑(兵营) ID + (将军 ID + 士兵类型)

续表

名　　称	定　　义	数据格式(以 Bit 为最小单位)
资源交易	MSG_ORDER_TRADE	0 ~ 8 消息类型
		9　　标志位 0 表示金钱换取资源 1 表示资源换取金钱
		10 ~ 24 建筑 ID
		25 ~ 32 木头 + 食物(各 4 位)
		33 ~ 40 铁 + 铜(各 4 位)
士兵治伤	MSG_ORDER_CURE	0 ~ 8 消息类型
		9 ~ 24 医馆 ID
		25 ~ 32 士兵 ID
……	……	……

(3)其他类别

其他类别如表 15.3 所示。

表 15.3

名　　称	定　　义	数据格式(以 Bit 为最小单位)
升级	MSG_ORDER_UPGRADE	0 ~ 8　消息类型
		9　级别 0 中等 1 高等
		10 ~ 24　城市 ID
结盟或宣战	MSG_ORDER_CHANGESTRATEGY	0 ~ 8　消息类型
		9　0 和平 1 战争
		10　保留
		11 ~ 16　国家 ID + 国家 ID 各 3 位
……	……	……

这里列出游戏中所用到的部分命令以及格式,为了节省网络带宽,使用位(bit)作基本单位来保存数据。如指挥人物移动的命令 MSG_ORDER_WALK,总长度为 5B,其中 0 ~ 8 位也就是第一个字节表示命令类型,最多有 255 种类型,9 ~ 32 位表示格子坐标的 X,Y,各占 12 位,数值范围是 0 ~ 4 095,也就意味着游戏中地图的格子尺寸最大为(4 096 × 4 096),这应该是够用了。然后是人物的 ID 号,只使用 8 位来表示,数值范围是 0 ~ 255,游戏中实际的人物数量并不是限制在 255 之内,而是这里考虑游戏中每个玩家可以控制的人物数量是有限的,每一方最多可以训练和使用大约 200 个士兵,如果把每个玩家可以控制的人物 ID 范围区分开,就可以只用 1B 来表示人物 ID。如第一个玩家控制的 ID 范围是 0 ~ 255,第二个玩家控制的 ID 范围是 256 ~ 511,依次往后:

人物实际 ID = 玩家 ID * 255 + 人物 ID

这种编号方式主要是为了节省一切不必要的数据传输,但并不是越省越好。例如,如果游戏设定中有一种对战模式需要多个玩家可以同时操作一支部队,那么这样的编号方式就不可行了。总之,要根据需求来灵活处理。

15.5 操作命令在游戏中的使用

下面以鼠标点击指挥士兵行动为例，来看一下这些操作命令在游戏中是如何使用的。

```
Game：:MouseButtonUp(int nButtonID,int nMouseX,int nMouseY)
//处理玩家鼠标点击操作
 {
    if(nButtonID == 鼠标左键)
     {
        //确定鼠标点击在地图上的格子坐标
        ScrPointToGrid(nMouseX,nMouseY,&nGridX,&nGridY);//坐标转换
        if(存在当前控制对象)
         {
            //指挥人物向目标点移动(添加人物移动的命令)
            AddCommand_ManMove(当前控制对象 ID,nGridX,nGridY);
        }
        else//通过点击选中要控制的对象
         {
            当前控制对象 ID = GetManAtGrid(nGridX,nGridY);
        }
}
```

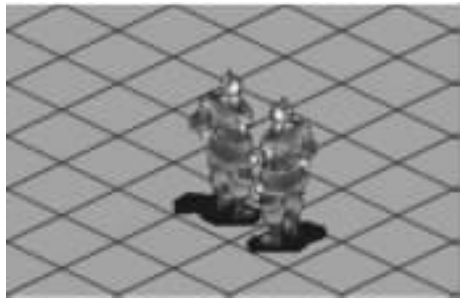
玩家点击鼠标后，函数 ScrPointToGrid() 将鼠标位置的屏幕坐标转化为游戏中的格子坐标。如果玩家在此之前没有选择过操作对象，那么通过函数 GerManAtGrid() 会返回位于该格子坐标里的人物 ID，并设置为玩家当前操作对象 如果玩家在此次点击鼠标之前已经选择好了要控制的士兵，那么玩家就是要指挥被控制的士兵向目标点移动，函数 AddCommand_ManMove() 就会被调用，其作用就是生成一条人物移动的命令，类型为 MSG_ORDER_WALK，并存放起来。

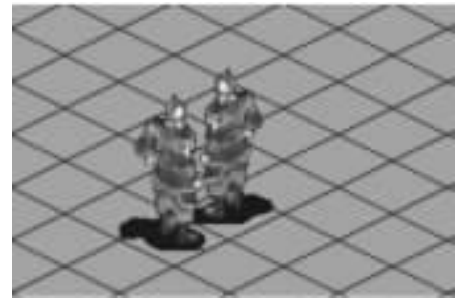

```
AddCommand_ManMove(BYTE btID,short sX,short sY)
 {
    LPBYTE * pbCmd = new BYTE[20];
    pbCmd[0] = MSG_ORDER_WALK;  //第一个字节是命令(消息)类型
    Bits32To24(pbCmd + 1,X,Y);  //将坐标 x,y 共 32bits 转化为 24bits 3 个字节
    pbCmd[3] = btID;            //人物 ID
    SendMessage(其他玩家,pbCmd);  //将命令发送给其他玩家
}
```

同理,玩家的其他各类操作也都通过类似的函数转化为命令,并作为网络消息发送给其他人。

15.6 同步周期的概念

经过一番分析,找到了前进的方向,但实际上长征才刚刚开始,更多的问题还在后面。仍然是士兵甲向沙地中心移动的例子,假设在士兵甲前往目的地的过程中不受其他操作的干扰。这意味着什么呢?先来看一个新情况,A 在指挥士兵甲出发之后,决定再增派一个士兵丙也前往沙地中心同一位置。也就是说士兵甲和士兵丙的目的地都是沙地中心,坐标为(x0,y0)。士兵甲先行出发,出发时的坐标是(x1,y1);士兵丙后出发,出发时的坐标是(x2,y2),两者行进速度相同。如果坐标(x2,y2)~(x0,y0)的距离大于(x1,y1)~(x0,y0)的距离,很显然,士兵甲会先到达沙地中心(x0,y0),根据设定,士兵丙到达之后,发现目的地的坐标位置上已经有人,于是停留在旁边。但是如果士兵丙的出发位置距离沙地中心比士兵甲更近,尽管后出发,也可能比士兵甲先行到达,然后占据了沙地中心(x0,y0)的位置,等士兵甲到达之后,发现有人,于是停留在旁边的位置上。玩家 A 把上述过程反复进行了 10 遍,其中有时候是士兵甲先到,有时候是丙先到,对于玩家 A 来说,谁先到达都无所谓,可以发现玩家 B 的机器上,出现了和 A 不一致的情况。在玩家 A 的屏幕上,士兵甲在沙地中心坐标(x0,y0)的位置上,士兵丙站在一旁,而在玩家 B 的机器上,士兵丙却站在沙地中心坐标(x0,y0),士兵甲站在一旁。





为什么会出现这种情况呢,难道不是 A 和 B 经历了完全相同的计算过程,产生的结果也理应一致么?没错,可是这里忽略了时间的因素,再回头看看游戏循环的概念。A,B 的机器是同时进入游戏的,游戏循环从 0 开始计数,当 A 的机器上的游戏循环跑了 100 次的时候,点击鼠标指挥士兵甲开始前进,假设每次游戏循环,士兵甲往前走一步,如从坐标(x,y)移动到坐标(x+1,y+1),而目标点沙地中心距离出发点距离为 10,那么经历 10 次游戏循环后,也就是在计数为 110 次的时候士兵甲便可以到达目的地了。当 A 的机器上游戏循环跑到 105 的时候,A 再次点击鼠标指挥士兵丙前往沙地中心,如果士兵丙距离沙地中心为 6,那么他会在计数为 111 的时候到达目的地,并且发现士兵甲已经到了,于是停在一旁。整个过程中,A 向 B 发送了自己的两次操作,B 接收执行。可是,关键的问题是 A,B 机器的速度是不一样的,而且消息在网络上传递也要花费时间,那么在 B 机器上执行这两次操作的时间(也就是游戏循环的计数)是不一样的。然后来看看 B 机器上的运行过程。首先假设 B 也在游戏循环跑到 100 次的时候收到了士兵甲出发的消息并执行,那么在 B 的机器上,士兵甲也会在计数到 110 次的时候抵达目标点,可是 B 的机器运行速度比较慢,当 A 机器上的游戏循环跑到 105 的时候,B

才跑到了 102，此时士兵甲在 A 的机器上距离目标点 5，假设网络上消息的延时接近一次游戏循环的时间，B 接收并执行士兵丙出发的消息是在 103，也就是士兵甲在 B 的机器上，此时的位置距离目的地还有 7，而士兵丙开始出发了，经过 6 次游戏循环，也就是计数为 109 的时候，士兵丙到达了沙地中心，而士兵甲在 110 的时候才会到达，于是在 B 的机器上，士兵丙站立在沙地中心坐标（x0，y0），而士兵甲在旁边停下了，A，B 不再一致。

上面的问题使我们意识到，通过网络消息在玩家之间传递操作是不够的，还要保证这些操作的执行时间在玩家的机器上是一致的。可是玩家机器的运行速度是无法控制的，诸如 CPU、总线、磁盘和操作系统等多种因素都会影响游戏运行速度，但是可以想办法统一玩家机器上游戏循环的速度。由于在游戏循环内部各项游戏元素是以一定周期运行的，游戏循环本身也是有周期的，因此在不做任何限制的情况下，游戏循环的运行速度就取决于机器的速度，但实际上可以限制游戏循环的速度以每秒固定次数来运行，如 30 次/s，计数函数可以采用 timeGetTime（）或者 GetTickCount（），形式如下：

```
unsigned long dwUpdateInterval = 33 ;//周期 = 1000（ms）/30
unsigned long dwCurTick = timeGetTime（）；
if（dwCurTick - dwLastTick）> = dwUpdateInterval）//每隔 33ms，游戏循环被调用一次
 {
     Game -> Run（）；
     dwLastTick = dwCurTick ；
}
```

但这只是一项最基本的约束，如果机器速度不够，或者某次游戏循环本身耗费的时间超过了 33ms，或者突发一些很耗费 CPU 时间的操作，如系统清除临时文件、回收内存等，都可能导致游戏循环并没有严格按照 30 次/s 的速度运行，这里需要真正保证的是发生操作的时刻完全相同。玩家 A 执行点击鼠标操作的时间是不能控制的，玩家 B 收到此消息的时间也是不能控制的，但是要让士兵开始移动的时间在 A 和 B 上完全相同。惟一的办法就是约定 A，B 机器上执行操作的时间，当玩家 A 点击鼠标后，并不是马上就执行士兵甲向目标移动的命令，而是等待一个约定的时间；玩家 B 收到消息之后，也不是马上执行，同样需要等待一个约定的时间。这个所谓约定时间是以游戏循环为计数单位的，这样就能让 A 和 B 上执行操作的时间完全相同。以游戏循环为计数单位是因为游戏循环的次数直接反映了游戏数据所处的状态，而任何的操作命令都会转换成对游戏数据的操作，这样 A，B 双方执行同一命令以同样的游戏数据状态为基础，游戏数据就保持一致了。

这里设定游戏循环每隔 5 次计数，便到达执行指令的时间，也就是位于 5，10，15…这样的时间点上。在达到时间点之前，所有的操作命令都被存储到一个列表中，到达时间点之后，一并执行。或许有人担心约定时间点来执行指令，是否会让玩家感觉操作不流畅，实际上这取决于时间点的周期到底有多长。当游戏循环以 30 次/s 的速度运行，如果以 5 来作执行指令的时间周期，每隔 1/6s 便发生了一次，也就是说玩家鼠标点击之后最慢 1/6s 内指令便得以执行，不会产生滞后的感觉。但是 5 是不是一个合适的周期时间和机器的速度差异以及网络上的延时都有很大关系，后面将会对这个问题进行专门讨论。

为了方便起见,把这个时间点的周期叫做‘同步周期”,有了它的帮助,网络对战的功能便有了实现的基础。

15.7 命令的收集与缓冲

约定相同的时间点来执行操作命令,首先需要对玩家的操作命令进行收集,而且必须保证在到达同步周期的时间点时,参加游戏的各玩家都有一份完全相同的命令集合。但是这一点并不是那么容易保证的,举个很简单的例子:

玩家 A 和玩家 B 各自进行了如下操作:

在 A 的机器上,

游戏循环计数 = 101,产生命令 01,点击鼠标指挥士兵甲移动到目标点(x1,y1);

发送命令 01 给玩家 B;

游戏循环计数 = 105,执行目前收集到的命令集;

游戏循环计数 = 106,接收命令 02,士兵乙移动到目标点(x2,y2)。

在 B 的机器上,

游戏循环计数 = 103,接收到命令 01,士兵甲移动到目标点(x1,y1);

游戏循环计数 = 104,产生命令 02,点击鼠标指挥士兵乙移动到目标点(x2,y2);

游戏循环计数 = 105,执行目前收集到的命令集;

发送命令 02 给玩家 A。

这里约定的执行命令的时间是游戏循环计数为 105。由于网络延时和玩家机器速度的差异,玩家 A 在自己的游戏循环 106 的时候才收到来自玩家 B 的命令 02,而玩家 B 在 103 的时候收到了玩家 A 的命令 01。显然,玩家 A 此同步周期的命令集合为一条命令 01,而玩家 B 此同步周期的命令集合为 2 条,分别是命令 01 和命令 02,双方出现了不一致的情况。产生这个问题的根源就是玩家各自产生命令的时刻都是以本地的游戏循环计数为主的,缺少一个统一的标准。为了解决这个问题,选用创建主机的玩家作为游戏循环计数的标准。当其他玩家产生操作命令的时候,本地并不做任何保存工作,而是直接发送给主机,由主机统一收集后,再以命令集合的形式发回给各个玩家。这样可以保证每个玩家执行的命令集完全相同,顺序也是一致的。这里体现了一个 Client/Server 结构的重要思想:Client 总是提交操作,判断和处理都在 Server 上进行,才能保证所有 Client 获得同样的结果,只不过这里的 Server 也是参加游戏的一个玩家,是一个兼职的 Server 而已。

另外,游戏循环虽然以 30 次/s 的速度运行,但它是不稳定的。当有玩家操作发生的时候,还没有一种方法来保证 A 和 B 机器上都已经过了相同次数的同步周期。事实上,需要让 A,B 机器上游戏循环的运行次数总是保持相同,即便是有不同的时候,相差也不会超过一个同步周期。这样当玩家有操作命令需要执行的时候,就可以发生在同一个时间点上。这里需要从 A,B 之中确定一个运行次数作为统一的标准,总是由标准的一方把自己的运行次数通知另一方,让另一方跟上自己的节奏,特别是当玩家个数超过 2 的时候,这样做是必需的。同样,用创建游戏的主机来作为标准的一方。定义一种同步消息来专门处理,主机需要把自己收集来的命令集合作为消息发送给其他玩家,这个命令集合的消息和需要的同步消息可以合并为一个。

同步消息(MSG_PERIOD_ORDER) = 主机已运行的同步周期次数 + 当前同步周期的命令

集合

命令存放的结构可以制定为：

```
SOrder
 {
     short       sSize；          //命令长度
     LPBYTE      pbContent；      //命令的内容
     DWORD       dwTime；         //命令的生成时间,以游戏循环计数为单位
     BYTE        btPlayerID；     //产生命令的玩家编号
}；
vector < SOrder * > OrderSet；//命令集合
```

假设玩家 A 是创建游戏的主机,那么游戏循环中会有如下代码：

```
Game：:Run()
 {
     if(m_bHostPlayer) //主机标志,玩家 A 此项被设为 true
      {
          if(nRunCnt == 发送时刻) //比同步周期的时间点大致提前一点的时间
           {
               SendMessage(其他所有玩家,同步消息)；
          }
     }
     if(nRunCnt == 同步时刻)
      {
          HandleOrderSet()；//执行命令集合
     }
     …
     nRunCnt + + ；
     }
```


考虑网络延时的因素,发送同步消息的时间并不是正好在同步周期的时间点上,而是适当提前一点,比如同步时刻是在 5,10,15 这样的时间点上,那么发送同步消息的时间就是 3,8,13,提前了 2 个游戏循环时间。这样,等非主机的玩家接收到同步消息,也就正好到达或已经接近所有玩家统一执行命令的时间,取得的效果最好。按照这样的处理方式,再来看看上面的例子发生了怎样的变化。

假设玩家 A 是创建游戏的主机,那么 A 的机器负责收集所有的命令操作。

在 A 的机器上,

游戏循环计数 = 101,产生命令 01,点击鼠标指挥士兵甲移动到目标点(x1,y1),保存命令

01 到此同步周期的命令集合里;

游戏循环计数 = 103,把目前收集到的命令集合作为同步消息,发送给玩家 B;

游戏循环计数 = 105,执行目前收集到的命令集合;

游戏循环计数 = 106,接收到命令 02,士兵乙移动到目标点(x2,y2),加入到新的命令集合中;

游戏循环计数 = 108,把目前收集到的命令集合作为同步消息,发送给玩家 B;

游戏循环计数 = 110,执行目前收集到的命令集合。

在 B 的机器上,

游戏循环计数 = 103,接收到命令 01,士兵甲移动到目标点(x1,y1);

游戏循环计数 = 104,产生命令 02,点击鼠标指挥士兵乙移动到目标点(x2,y2);发送给主机玩家 A;

游戏循环计数 = 105,接收到来自玩家 A 的同步消息;

游戏循环计数 = 105,执行同步消息中的命令集合;

游戏循环计数 = 110,接收到来自玩家 A 的同步消息;

游戏循环计数 = 110,执行同步消息中的命令集合。

可以看到,玩家 A,B 都在同步时刻 105 执行了命令 01,命令 02 在同步时刻 110 的时候得以执行,双方始终保持一致。

还有一个需要提到的问题是:操作命令的缓冲。这里必须保证主机玩家 A 在发送同步消息之后,不再往当前命令集中添加新的命令,那么在此之后产生或接收到的新命令会存放到一个后备命令集中。到达同步时刻后,当前命令集的内容被执行并清空,然后把后备命令集的内容转移到当前命令集。但是,对于非主机玩家,总是有一个接收到的命令集合就够用了。这里的做法可以比较灵活,如仅仅使用一个命令集,当主机到达同步时刻的时候,根据命令结构里记录的时间来决定是否应该执行,大于发送时刻的命令就会等到下一个同步周期才执行。本实例中,为了概念上比较清晰,使用的方法是当前命令集 + 后备命令集。

15.8 同步的相关处理

非主机玩家到达同步时刻后,便要执行来自主机的命令集,假如主机的同步消息还没到达,便处于等待状态。一旦收到了命令集,就需要通知主机自己的此同步周期已经可以顺利完成,否则主机永远不知道其他人现在处于什么状态。这里设计一个消息来表示这种回应信息 MSG_PERIOD_OK,把主机发过来的同步周期次数再发回主机,表示自己已经收到此周期的命令集,经过同步时刻,执行了命令集之后便可以进行下一个周期了。而主机在发送了当前周期的命令集之后,便开始检查其他所有玩家的 MSG_PERIOD_OK 消息是否抵达,如果全部抵达,主机开始继续下一个同步周期,否则便进入等待状态。

例如,现有玩家 A,B,C,D,其中玩家 A 是主机,同步周期是 5,当游戏循环次数进行到 103 的时候,A 把同步周期次数 21 以及本周期的命令集发送给玩家 B,C,D,如果没有出任何问题,玩家 B,C,D 都会在自己的游戏循环进行到 101 ~ 105 之间收到来自 A 的同步消息,一旦收到马上给 A 回应 MSG_PERIOD_OK,回应之后 B,C,D 的游戏循环会继续,到达 105 的时候,执行命令集,然后继续。主机 A 会在自己的游戏循环进行到 103 ~ 105 之间收到来自 B,C,D 的回

应消息。确定它们的回应消息都抵达之后,主机的游戏循环从 105 继续。如果在此期间 B 不幸掉线或死机了,没有能够发给 A 回应消息,那么当 A 的游戏循环进行了 105 的时候,由于缺少 B 的回应,进入等待状态,而玩家 C,D 的游戏循环虽然可以从 105 继续到 110,但没有收到新的同步消息,也进入了等待状态。A 可以把等待 B 的时间累计下来,并产生一个等待消息 MSG_WAIT 发送给 C,D。消息内容包含玩家 B 的 ID 和目前经过的等待时间。这样,便可以看到一个很多即时战略游戏里都会出现的等待界面。如果主机 A 等待了约定的超时时间或者决定不再等待,而是从游戏中剔除玩家 B,那么 A 的游戏循环会继续,而游戏循环位于 110 的 C,D 也会收到同步周期为 22 的命令集,游戏继续进行。玩家 B 不会再产生控制命令,他控制的一方不再动作或改为电脑 AI 接管。如果玩家 B 并没有掉线只是暂时无法回应,那么还有时间可以允许它发出回应消息,主机收到后,游戏也可以继续进行。

这个过程的代码形式如下:

```
Game::Run()
{
    if(m_bHostPlayer)//主机处理
    {
        if(nRunCnt == 发送时刻)//比同步周期的时间点大致提前一点
        {
            SendMessage(其他所有玩家,同步消息);
        }
        if(nRunCnt == 同步时刻)
        {
            if(CheckPeriodOK(所有其他玩家))//检查其他玩家是否都已回应
            {
                HandleOrderSet();//执行命令集
                m_bPause = false;//取消等待状态
            }
            else m_bPause = true;//进入等待状态
        }
    }
    else//非主机处理
    {
        if(nRunCnt == 同步时刻)
        {
            if(已经接收到同步消息)
            {
                HandleOrderSet();//执行命令集
                m_bPause = false;
            }
```

```
            else    m_bPause = true ;//进入等待状态
           }
      }
      if(!m_bPause)nRunCnt++;
}
```

其中,函数 CheckPeroidOK 用来检查是否收到其他玩家的回应消息。

```
boolGame::CheckPeriodOK()
{
      bool bWaiting = false;
      for(int i=0;i<m_nPlayerCount;i++)//依次检查每个玩家
      {
           SPlayerData * pPlayerData = &(m_pGamePlayer[i].m_Data);
           if(!pPlayerData->m_bPeriodOK)//收到玩家 i 的回应,就返回 true
           {
                pPlayerData->dwWaitingTick++;//等待时间累加计数
                bWaiting = true;
           }
      }
      if(!bWaiting)//已经无需等待任何人
      {
           for(int i=0;i<m_nPlayerCount;i++)
           {
                SPlayerData * pPlayerData = &(m_pGamePlayer[i].m_Data);
                pPlayerData->m_bPeriodOK = false;//设为 false,为下一个周期做好准备
                pPlayerData->dwWaitingTick = 0;//等待时间计数清零
           }
           }
           return bWaiting;
}
```

另外,主机发给其他玩家的同步消息中包含有当前同步周期的次数,这个次数还隐含了游戏循环次数的信息,也就是当其他玩家接收到同步消息之后,马上知道主机的游戏循环至少已经到达了发送时刻(比同步时刻提前的一个固定的时间),如果某玩家的机器很慢,如主机发送时刻是 103,收到主机同步消息的时侯还是 101,那么这时需要加快游戏循环的速度来赶上主机的节奏。

15.9 DirectPlay 在游戏中的使用

在前面的 7 个小节中，已经讲述了制作网络对战功能的思路和实现方法，它们都是和具体使用哪一个网络底层（API）无关的。在本实例中，使用的网络 API 是 DirectPlay，从本节开始，会陆续接触到 DirectPlay 的具体使用细节和方法。

DirectPlay 实际上就是将各种网络连接方式和协议封装起来，并提供简单易用的一套界面（函数）来进行网络通讯的操作。这些函数可以大致分为如下几类：

（1）初始化

负责创建 DirectPlay 对象、列举可用的连接等等。

CoInitialize()　初始化 COM 接口。

CoCreateInstance()　通过 COM 接口来创建 DirectPlay 对象。

EnumConnections()　列举可用的连接方式。

InitializeConnection()　初始化连接。

（2）会话操作

建立玩家相互连接的一个形式，会话（Session）是 DirectPlay 里的一个重要概念。在玩联网游戏时，某人建立了游戏服务，其他人加入，我们就可以说他们正在进行一个会话。

EnumSessions()　列举当前网络上所有的会话。

Open()　打开（创建）一个新会话或加入一个已有的会话。

> 注意：这个函数是用参数（DPOPEN_CREATE 或者 DPOPEN_JOIN）来区分是打开（创建）一个新会话还是加入一个已有的会话。

Close()　关闭会话

（3）Player 的管理操作

创建 Player、从会话中添加或删除 Player 等等。

CreatePlayer()　用参数（DPPLAYER_SERVERPLAYER 或者 0）的不同来区分该 Player 是否为会话的创建者。

DestroyPlayer()　清除 Player

GetPlayerAccount()

GetPlayerAddress()

GetPlayerName()

GetPlayerData()

SetPlayerData()…一套存取有关 Player 信息的函数，从名字上就可以很清晰地看到它们的功用

（4）消息处理

负责 Player 之间网络消息的发送、接受等操作。

Send() 发送

Receive() 接受

这里只列出了常用到的重要函数,如果想全面了解 DirectPlay,请查阅 DirectX SDK 文档。下面结合游戏,列举了一些运用 DirectPlay 的场合。

①最基本的发送和接收网络消息。

②查询当前机器支持的网络通讯协议。

③主机创建游戏,并选择所用的协议类型,如 IPX,TCP/IP,串口连接等。

④其他玩家查询现有的游戏列表,选择后加入。

⑤在游戏预备状态下,各玩家更新自己的国家、颜色等选项以及游戏正式开始后,主机发送同步消息以及收集操作命令等网络消息的接收和处理。

⑥玩家离开游戏,收到 DirectPlay 系统消息。

⑦主机离开游戏,其他玩家中有一人收到特别的 DirectPlay 系统消息,定义为 DPSYS_HOST,即主机转移消息。

要使用 DirectPlay,首先要创建相关的对象,然后通过这些对象提供的接口函数来实现需求。DirectPlay 提供了两种对象,DirectPlay 和 DirectLobby。

(1)定义

LPDIRECTPLAY4A m_pDP;

LPDIRECTPLAYLOBBY2 m_pDPLobby;

其中,LPDIRECTPLAYLOBBY2 是用来处理 Lobby(大厅)的,在后面会具体介绍;会话操作、Player 管理、消息处理等接口函数都是属于对象 DIRECTPLAY4A 的。如 m_pDP->Open(),m_pDP->Receive()等。

(2)初始化

CoInitialize(NULL);

CoCreateInstance(CLSID_DirectPlay,NULL,CLSCTX_ALL,IID_IDirectPlay4A,(VOID**)&m_pDP);

DirectX 的对象都是基于 Com 组件的,DirectPlay 也不例外,所以创建对象前要先初始化 COM 接口。主要的好处是当新版本的 DirectX 出现的时候,开发者不用修改接口的调用,也可以获得新版本的特性。IID_IDirectPlay4A 指定要创建一个版本是 4 的 DirectPlay 对象,大写字母 A 表示使用 ANSI 字符集,如果不写 A,表示使用 Unicode,如果要制作的游戏会有各个语言的版本,那么就应该选择后者。

(3)查询通讯协议并列表

下面是一个列举当前机器支持的所有协议(连接类型)的函数。这个函数是基于对话框界面的,其作用是把支持的协议列表放到一个 ListBox 控件中。在 DirectPlay SDK 的例程中,有相同作用的函数,这里的对话框针对游戏内容修饰过,并增加了一些游戏选项。

```
HRESULT GetConnectionList(HWND hDlg)//传入对话框的句柄
{
    HRESULT hr;
    LPDIRECTPLAY4pDP = NULL;
```

```
    //创建一个临时的 DirectPlay 对象
    if(FAILED(hr = CoCreateInstance(CLSID_DirectPlay,NULL,CLSCTX_ALL,
    IID_IDirectPlay4A,(VOID * *)&pDP)))
     {
        if(hr == E_NOINTERFACE) //如果不存在 IID_IDirectPlay4A 的接口
         {
            MessageBox(NULL,"缺少接口功能,请检查 DirectX 版本是否过旧","初始
化错误",MB_OK|MB_ICONERROR);
        }
        return hr;//出错,返回
    }
    //取得 ListBox 的句柄
    HWND hWndListBox = GetDlgItem(hDlg,IDC_CONNECTION_LIST);
    //开始列举连接类型,每发现一个新的类型,都会触发指定的回调函数
    if(FAILED(hr = pDP -> EnumConnections(&g_AppGUID,
    CALLBACK_EnumConnections,hWndListBox,0)))
     {
        SAFE_RELEASE(pDP);
        return hr;
    }
    SAFE_RELEASE(pDP);
    SetFocus(hWndListBox);
    //发送消息给 ListBox  查找列出的协议中是否有名字与 g_strPreferredProvider
    //相同,如果有则把该协议名称所在的那一行设为选中的行,否则选中第一行,
    //这里 g_strPreferredProvider 中实现存放一个默认协议名称,如 TCP/IP 等
    int nIndex = SendMessage(hWndListBox,LB_FINDSTRINGEXACT,WPARAM) - 1,
    (LPARAM)g_strPreferredProvider);
    if(nIndex != LB_ERR)
     {
        SendMessage(hWndListBox,LB_SETCURSEL,nIndex,0);
    }
    else SendMessage(hWndListBox,LB_SETCURSEL,0,0);
    return S_OK;
}
```

下面是列举连接类型的回调函数,每当查找到新的连接类型会向此函数传递该类型的名字(pName)和相关的数据内容(pConnection)。

```
BOOL FAR PASCAL CALLBACK_EnumConnections(LPCGUID pguidSP,VOID *
pConnection,DWORD dwConnectionSize,LPCDPNAME pName,DWORD dwFlags,
VOID * pvContext)
{
    HRESULT   hr;
    HWND   hWndListBox = (HWND)pvContext;
    SetFocus(hWndListBox);
    //创建临时的 DirectPlay 对象
    LPDIRECTPLAY4pDP = NULL;
    if(FAILED(hr = CoCreateInstance(CLSID_DirectPlay,NULL,CLSCTX_ALL,
    IID_IDirectPlay4A,(VOID * *)&pDP)))
    {
        return FALSE;//如果创建失败,则返回错误,列举过程就会停止
    }
    //对查找到的连接进行初始化
    if(FAILED(hr = pDP -> InitializeConnection(pConnection,0)))
    {
        SAFE_RELEASE(pDP);
        return TRUE;//如果不能正常初始化,说明该连接不可用,返回 TURE 以后
        //列举过程继续
    }
    SAFE_RELEASE(pDP);
    //上面找到的连接类型可用,把它的名字加入到 ListBox 中
    int nIndex = SendMessage(hWndListBox,LB_ADDSTRING,0,(LPARAM)pName -> 
lpszShortNameA);
    //初始化该连接的数据内容
    VOID * pConnectionBuffer = newBYTE[dwConnectionSize];
    memcpy(pConnectionBuffer,pConnection,dwConnectionSize);
    //存放到 ListBox 的 Item 中,和名字一一对应
    SendMessage(hWndListBox,LB_SETITEMDATA,nIndex,(LPARAM)pConnectionBuf-
fer);
    return TRUE;//返回 TURE,继续列举下一个连接类型
}
```

清除通讯协议列表的函数如下,在对话框关闭的时候被调用:

```
VOID ClearConnections(HWND hDlg)
{
    HWND hWndListBox = GetDlgItem(hDlg,IDC_CONNECTION_LIST);
    DWORD dwCount = SendMessage(hWndListBox,LB_GETCOUNT,0,0);
    for(int nIndex = 0;nIndex < dwCount;nIndex++)
    {
    //逐个删除 ListBox 的 Item 所附属的 Connection 数据内容
    VOID * pConnectionBuffer = (VOID *) SendMessage(hWndListBox,
    LB_GETITEMDATA,nIndex,0);
    SAFE_DELETE_ARRAY(pConnectionBuffer);
    }
}
```

在游戏中,为了方便玩家的操作,这里并不打算列出所有支持的通讯协议来让玩家选择,而是在界面上给出如下几个选项:

①局域网对战,使用 TCP/IP 协议,自动在局域网内查找游戏。

②Internet 对战,使用 TCP/IP 协议。

③战网 Server 对战,默认转而连接 Lobby Server(仍然使用 TCP/IP 协议)。

事实上,在本实例中完全没有使用 IPX 协议,这样玩家只要在机器上安装了 TCP/IP 协议,就可以进行游戏了,这是完全可行的,著名的 A-RPG 游戏《DiabloII》也是使用同样的处理方法。但是使用 TCP/IP 协议,需要玩家输入 IP 地址才能进行连接。在局域网对战的时候,利用 DirectPlay 可以不用输入 IP 地址进行连接。因此,玩家并不会看到列表框的存在,而是看到一个对话框界面,在界面中可以输入名字和选择简化的网络连接类型。

如果玩家选择①或②,那么就会进入创建游戏的相关界面;如果选择③,则会进入 Lobby 服务器的选择界面。

该对话框的消息处理函数如下:

```
BOOL CALLBACK ConnectionsDlgProc(HWND hDlg,UINT msg,WPARAM wParam,
LPARAM lParam)
{
    HRESULT hr;
    switch(msg)
    {
        case WM_PAINT:
        {
            //此处可以放置一些有关绘制对话框背景图片的代码
```

```
        break;
    }
    case WM_INITDIALOG://对话框初始化
    {
        //限制用来输入玩家名字的 Edit 控件可以接收的最大长度
        HWND hEdit = GetDlgItem(hDlg,IDC_PLAYER_NAME_EDIT);
        SendMessage(hEdit,EM_SETLIMITTEXT,8,0);
        //设置对话框图标和标题等
        ……
        //开始列举连接类型
        if(FAILED(hr = GetConnectionslList(hDlg)))
        {
            //如果失败,则退出
            EndDialog(hDlg,EXITCODE_ERROR);
        }
        …
        break;
    }
    case WM_COMMAND:
    {
        switch(LOWORD(wParam))
        {
            case IDC_BUTTONLOCAL://点中按钮选择局域网对战
            nGameConnectType = LOCALNET_GAME;
            break;
            case IDC_BUTTONNET://点中按钮选择 Internet 对战
            nGameLinkType = INTERNET_GAME;
            break;
            case IDC_BUTTONLOBBY://点中按钮选择转入大厅对战
            nGameLinkType = SERVER_GAME;
            break;
            case IDC_CANCEL://选择退出
            nGameLinkType = 0;
            EndDialog(hDlg,EXITCODE_QUIT);
            break;
            default:
            return FALSE;//没有定义的消息,出错退出
        }
```

```
        if(FAILED(hr = ConnectionsSelectOK(hDlg)))
                {
                    EndDialog(hDlg,EXITCODE_ERROR);
                }
            }
            case WM_DESTROY:
            ClearConnections(hDlg);//对话框关闭时清除连接
            break;
        }
        return TRUE;
}
HRESULT ConnectionsSelectOK(HWND hDlg)
        {
        //检查玩家输入的名字
        GetDlgItemText(hDlg,IDC_PLAYER_NAME_EDIT,g_strLocalPlayerName,
        MAX_PLAYER_NAME);
        if(strlen(g_strLocalPlayerName) == 0)
        {
            MessageBox(hDlg,TEXT("请输入一个名字")),TEXT(" ?? 名字"),MB_OK);
            return S_OK;
        }
        HWND hWndListBox = GetDlgItem(hDlg,IDC_CONNECTION_LIST);
        int nIndex = SendMessage(hWndListBox,LB_GETCURSEL,0,0);
        SendMessage(hWndListBox,LB_GETTEXT,nIndex,(LPARAM)g_strPreferredProvider);
        VOID * pConnection = (VOID *)SendMessage(hWndListBox,LB_GETITEMDATA,
        nIndex,0);
        if(NULL == pConnection)
        {
            EndDialog(hDlg,EXITCODE_LOBBYCONNECT);
            return S_OK;
        }
        ResetDPlayServerIP("0x00");
        EndDialog(hDlg,EXITCODE_FORWARD);
            return S_OK;
}
```

15.10 网络消息的接收与处理

常用的接收网络消息的方式有两种：

①阻塞式的接收。

②异步的非阻塞式的接收。

如果使用方式①,那么接收网络消息的函数就会一直等待消息,除非到达指定的时间,与此同时,程序的运行也停在此处,不能做其他任何动作。如果使用方式②,接收消息的函数只是去消息缓冲里检查一下是否有新的消息,然后马上返回,中间不存在等待的过程,而在程序中需要设置循环来反复检查消息。

这里的游戏程序主线程中有一个大大的游戏循环,显然不能用阻塞的方式来处理网络消息,否则就会中断游戏过程。但如果把这个阻塞的过程放到另一个单独的线程中,就不会影响游戏主线程的运行,实际上这是一种很常见的做法。

在使用 DirectPlay 的情况下,与线程相关的代码如下所示。

首先在游戏初始化过程中创建线程：

```
Game::Init()
{
    m_dwMsgThreadID = 0 ;//线程 ID 初始化为 0
    //创建线程
    m_hMsgThread = CreateThread(NULL,0,CheckNetMsg,NULL,0,&dwMsgThreadID);
    //定义 DirectPlay 所需要的事件,当接收到网络消息时触发该事件
    g_hDPMessageEvent = CreateEvent(NULL,FALSE,FALSE,NULL);
    //定义线程退出事件,当游戏结束的时候,使用 SerEvent()退出线程
    g_hPlayerExitEvent = CreateEvent(NULL,FALSE,FALSE,NULL);
}
//线程函数
DWORD WINAPI CheckNetMsg(LPVOID)
{
    HANDLE EventHandles[2];//定义两种被检查的事件
    EventHandles[0] = g_hDPMessageEvent;
    EventHandles[1] = g_hPlayerExitEvent;
    //进入检查消息的循环,等待时间是 INFINITE,即阻塞式的检查
    //当发生消息事件 WAIT_OBJECT_0 就调用 ReceiveMessage 来处理
    //当发生退出事件 WAIT_OBJECT_1,即 g_hPlayerExitEvent,就结束
    //while 循环
      while(WaitForMultipleObjects(2,EventHandles,FALSE,
            INFINITE) == WAIT_OBJECT_0)
    {
```

```
        ReceiveMessages();//接收消息的函数
        }
        ExitThread(0);//退出线程
        return 0;
}
```

其中,函数 ReceiveMessage()负责接收网络消息。

使用 DirectPlay,首先要定义一个 DirectPlay 的对象,例如:

LPDIRECTPLAY4A m_pDP;

接收消息和发送消息的调用方式分别为 m_pDP->Receive()和 m_pDP->Send()。这两个函数的定义如下:

HRESULT Receive(LPDPID lpidFrom,LPDPID lpidTo,DWORD dwFlags,LPVOID lpData,LPDWORD lpdwDataSize);

lpidFrom 指针用来获得消息发送者的 PlayerID。

lpidTo 指针用来获得消息接收者的 PlayerID。

dwFlags 可以用来联合使用的标志,有如下取值:

DPRECEIVE_ALL 缺省值,直接取出一条消息;

DPRECEIVE_PEEK 取出一条消息,但消息不会从消息队列中被删掉;

DPRECEIVE_TOPLAYER 取得传给 lpidTo 指向的接收者 Player 的第一条消息;

DPRECEIVE_FROMPLAYER 取得传给 lpidFrom 指向的发送者 Player 的第一条消息;

lpdwDataSize 传入时表示用来接收的缓冲的最大尺寸,返回时表示实际接收到的字节数。

HRESULT Send(DPID idFrom,DPID idTo,DWORD dwFlags,LPVOID lpData,DWORD dwDataSize);

其中:

idFrom 为发送消息的本地 PlayerID。

idTo 为接收者的 PlayerID。

dwFlags 为可以用来联合使用的标志,有如下取值:

DPSEND_ENCRYPTED 对消息加密(当前会话是安全类型时起作用);

DPSEND_GUARANTEED 消息按照有保证的方式发送;

DPSEND_SIGNED 对消息用数字签名(当前会话是安全类型起作用);

lpData 指针指向要发送的数据;

dwDataSize 为要发送的数据长度。

下面是消息接收过程,这里将其封装到函数中。

```
Game::ReceiveMessage()
 {
    //定义 DPlay 所需要的 ID,用来标识消息的发送者和接收者
```

```
    DPID idFrom,idTo;
    //DPlay 的接收消息函数,消息内容存入 pbMsgBuf 中
    //nSize 返回消息的大小
    HRESULT hr;
    hr = m_pDPlay->Receive(&idFrom,&idTo,DPRECEIVE_ALL,
                                        pbMsgBuf,&nSize);
    //DPlay 建议接收消息采用不固定大小的缓冲区,用试探的方式来接收,如果将要接
收的
    //内容尺寸大于已经分配好的缓冲区,则重新分配缓冲区大小,再次接收
    if(hr == DPERR_BUFFERTOOSMALL)//返回信息表示缓冲区太小
    {
        if(pbMsgBuf)
        {
            delete pbMsgBuf;//删除老的缓冲区
    }
    pbMsgBuf = new BYTE[nSize];//创建新的缓冲区
    }
    else if(SUCCEEDED(hr))//成功接收了消息
    {
        //对消息进行处理
        HandleNetMessage(pbMsgBuf,nSize,idFrom);
    }
}
```

函数 HandleNetMessage()负责解析网络消息,并转换为对游戏数据的操作。

```
Game::HandleNetMessage(LPBYTE pbMsgBuf,int nSize,DPID idFrom)
{
    BYTE btMsgType = *(pbMsgBuf+0);//取得消息类型
    switch(btMsgType)//根据不同消息类型做相应处理
    {
        case …
    }
}
```

这里要注意的是参数 idFrom,它除了标识不同的玩家之外,还可以表示和 DirectPlay 相关的消息属性。当它标识玩家的时候,就是 struct SPlayerData 中的 nPlayerID,通过比较 nPlayerID 和 idFrom 的值,就可以知道消息来自哪个玩家;idFrom 还可以等于 DPID_SYSMSG,表示消息来源是 DirectPlay 内部,即系统消息。这是 DirectPlay 特有的功能之一,可以定义函数来专门处理这种系统消息。其算法和函数意义如下:

```
if(idFrom == DPID_SYSMSG)
 {
     HandleDPlaySysMessage(pbMsgBuf,nSize);//对 DPlay 系统消息的处理
}
else
 {
     HandleNetMessage(pbMsgBuf,nSize);//自定义消息的处理
}
//函数定义
Game::HandleDPlaySysMessage(LPBYTE pbMsgBuf,int nSize)
 {
     switch(btMsgType)
      {
          case DPSYS_CREATEPLAYERORGROUP:
          //DPlay 消息:创建玩家组
          case DPSYS_HOST: …
          //DPlay 消息:主机玩家更换,这是一个对设计者来说非常体贴的消息
     }
}
```

15.11 消息接收的线程同步

我们知道主机会按一定周期不断发给其他玩家同步消息,同步消息包括一个命令集以及主机的当前同步周期次数,在同步周期之间,主机会收到来自其他玩家的操作命令,并添加到命令集中。这里会产生一个很值得注意的问题就是线程的同步。主机玩家自己也会不断产生操作命令,那么在主机玩家的主线程中,会对命令集进行添加操作。例如,鼠标点击指挥士兵移动,其处理过程如下。

主机玩家的主线程:

```
->HandleInputMessage()
        ->MouseButtonUp()
            ->AddCommand_ManMove()
                ->AddOrder()
                    ->AddToOrderSet()
```

其中,函数 AddCommand_ManMove()会产生一条人物移动的指令,前面已经列举过它的代码,最后一行是把消息发送给其他玩家,即:

SendMessage(其他玩家,pbCmd);

这里采用主机收集和转发的做法,因此这一行应该被修改为:

AddOrder(pbCmd);

函数 AddOrder 内部会作如下处理:

```
Game::AddOrder(LPBYTE pbCmd)
{
    if(m_bHostPlayer)//如果是主机
    {
        AddToOrderSet(pbCmd);//直接添加到命令集中
    }
    else
    {
        SendMessage(主机,pbCmd);//如果不是主机,则将命令发送到主机
    }
}
```

下面再来看看当主机收到其他玩家指挥士兵移动的消息的处理流程。

主机玩家接收消息的线程:

```
->CheckNetMessage()
    ->ReceiveMessage()
        ->HandleNetMessage()
Game::HandleNetMessage(LPBYTE pbMsgBuf,int nSize)
{
    …
    switch(btMsgType)
    {
        case MSG_ORDER_WALK:
        AddToOrderSet(pbMsgBuf);
        break;
    }
}
```

显然,游戏的主线程和接收消息的线程都会调用 AddToOrderSet(),也就是两个线程很有可能会同时操作用来保存命令集合的数组。除此之外,主线程在同步时刻执行完所有命令之后还会做清除命令集的动作,也就是说会出现一边在添加,一边在删除的情况。这些都会导致命令集混乱甚至程序出错。多线程最需要注意的就是同时对数据操作引发的问题,解决的办法也很简单,WIN32 API 和 MFC 中有很多用于线程互斥的信号量处理函数。如 CSemaphore,

只要定义一个 m_Sem,在操作数据之前 Lock()和操作结束之后 Unlock()即可。例如,

主线程:

```
if(m_Sem. Lock())
 {
     HandleInputMessage()
     m_Sem. Unlock();
}
```

消息线程:

```
if(m_Sem. Lock())
 {
     HandleNetMessage();
     m_Sem. Unlock();
}
```

这样,多线程对数据的操作就是互斥进行的,不会引发问题。

15.12 建立容错机制

已经知道玩家之间保持同步的最主要的两条消息是 MSG_PERIOD_ORDER 和 MSG_PERIOD_OK,它们不停地来往于主机玩家(Server)和非主机玩家(Client)之间,一旦有一方没有收到消息,游戏便停止运行并且进入等待状态。那么消息万一发生了丢失,是否意味着游戏就此不能进行下去呢,因此应该建立一个容错机制。

首先,由于使用的网络底层是 DirectPlay,它包含对 TCP/IP,UDP,IPX 等各种通信协议的支持。以 TCP/IP 协议为例,它本身是保证数据可靠传输的,对网络通信中的硬件故障、网络堵塞、延迟、数据丢失和损坏、数据重复和顺序错误等问题都有相应的处理办法,很好地保证了数据包的完整和可靠收发。但它不是万能的,我们知道数据通信还会经过操作系统这一关,操作系统内部有网络消息的缓冲,当发生网络故障,消息就会堆积于缓冲之中,一旦塞满,就会有舍弃的处理,所以如果希望消息的传输 100% 可靠,最好是游戏本身在这方面具有容错机制,而且这种机制不仅仅可以用来保证消息传输,还可以使开发者在通信协议的选择上更灵活。我们都知道 UDP 协议的特点是不对数据包进行分组和组装,也不排序,不保证消息一定正确到达,应用程序使用它时必须自己保证消息是正确完好地到达目的地。随之换来的是处理速度的提高,既然用 TCP 协议也要靠自己建立容错机制,为什么不选用速度快、效率高的 UDP 协议呢,而且 UDP 协议很适合一些类型的游戏开发,比如 QUAKE 之类的 3D 射击游戏,需要以很快的速度不断传输和更新玩家控制人物的状态,包括位置、方向、动作等等,新的状态会源源不断地被接受然后转变为屏幕上的显示,即使中间出现了消息丢失的情况,新的状态消息马上就会补上来,玩家仍然可以得到正确的屏幕显示,在这种情况下,可以高速传输的 UDP 消息自然成了不二之选。

DirectPlay 的消息发送函数提供了一个参数,可以指定当前要发送的消息是需要有保证的

还是无需保证的,如果是前者,DirectPlay 内部会对这个消息有一个返回确认,发送消息的速度也会明显变慢。出于对同步速度的需要,对同步消息必须采用无保证的发送方式,然后为同步消息加一个附加数据可以标识它的次序。在上面关于同步消息的定义中已经提到,同步消息 MSG_PERIOD_ORDER 包含了主机已运行的同步周期次数,正好可以用来实现对消息的确认,流程如下:

①主机按一定周期不断向其他玩家发送同步消息,如果超过一定时间,仍然得不到回应,则进入超时处理。

②其他玩家接收到同步消息,解析后,进入③。

③其他玩家记录来自主机的同步周期次数,并且标志为已接收,进入④。如果该同步周期次数已标志为接收,表示已经是重复接收,不做处理,进入④。

④其他玩家向主机发回确认消息,消息中附带刚收到的同步消息次数。

⑤主机收到所有玩家的确认消息,才可以继续进入下一轮同步,否则循环至①。

经过这样的流程处理之后,用来保持同步的最主要的两条消息 MSG_PERIOD_ORDER 和 MSG_PERIOD_OK 就完全不怕丢失了,测试的时候甚至可以故意扔掉一些消息,游戏依然运作正常,而且 DirectPlay 也专门提供了这样的工具来帮助进行测试。

15.13 用 Log 文件帮助调试

在调试对战的网络游戏程序的时候,经常会出现 2 台机器上出现不一样的画面,说明游戏已经不再同步了,这时候想要知道到底什么地方出错了是非常困难的,没有什么调试工具能够帮助,因此非常有必要建立一个良好的 Log 机制。所谓 Log 就是把游戏进展的过程或者一些程序运行状态有关的数据记录到文件里,用于跟踪查看。

Log 的实现方法有很多种。在 Windows 平台下调试程序时,最好还是能开一个额外的窗口来显示 Log 信息,很方便的方法是另外创建一个 Console 窗口,任何 Log 的内容既写入文件又输出到 Console,那么就可以即时查看。

下面是一个常用的 Log 函数。

```
BOOL      g_bFirstLog = TRUE;//全局变量表示是否第一次 Log
FILE *    g_fpLog = NULL;//Log 文件的指针
HANDLE    __hStdOut = NULL;
void Log(char * szFormat,...)//参数是不定长的,使用方法与 printf 函数一样
{
#ifndef_LOG
    return;
#endif
char szLog[512];
    if(g_bFirstLog)//如果是第一次 Log
    {
        g_fpLog = fopen("log.txt","wt");//打开文件
```

```
        g_bFirstLog = FALSE;
#ifdef_LOG_CONSOLE
            CreateConsole(80,24);//创建一个 Console 窗口,宽为 80 字符,高 24 字符
#endif
    }
    BOOL bMsg = FALSE;//是否显示 Windows message box
    if(strlen(szFormat) > 3)//if MsgBox
     {
        //格式符的前 3 个字母如果是‘ msg ’就弹出一个 MessageBox 来显示 Log 的内容
        if(szFormat[0] == 'm'&&szFormat[1] == 's'&&szFormat[2] == 'g')
         {
            bMsg = TRUE;
        }
    }
    va_list list;
    va_start(list,szFormat);//开始解析格式符和对应的参数数据
    if(bMsg)
     {
        vsprintf(szLog,szFormat + 3,list);
    }
    else
     {
        vsprintf(szLog,szFormat,list);
    }
    va_end(list);
    fprintf(g_fpLog,"%s",szLog);
    fflush(g_fpLog);//这一句很重要,保证每次 Log 的内容一定写入文件,以防止程序
漏写
#ifdefWIN32//如果是在 Windows 操作系统中,并且 Console 窗口存在
    #ifndef_CONSOLE
        if(__hStdOut)
         {
            unsigned long cCharsWritten;
            WriteConsole(__hStdOut,szLog,strlen(szLog),&cCharsWritten,NULL);
        }
    #else
        printf("%s",szLog);//直接用 printf 输出
```

```
        #endif
#else
            printf("%s",szLog);//直接用 printf 输出
#endif
#ifdef WIN32
        if(bMsg)
        {
            MSGSTR(szLog);
        }
#endif
}
//在 Windows 中,创建 Console 窗口的函数
//参数为窗口的宽和高,以字符为单位
void CreateConsole(int width,int height)
{
        AllocConsole();
        SetConsoleTitle("Debug Window");
        __hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);//standard output HANDLE
        COORD co = {width,height};
        SetConsoleScreenBufferSize(__hStdOut,co);//set buf size
        freopen("CONOUT $","w+t",stdout);
        freopen("CONIN $","r+t",stdin);
#endif
}
```

这样,要记录玩家接收到了消息就可以写成:

```
Log("玩家%d 接收到消息%d\n",nPlayerID,nMsgType);//记录下玩家 ID 和消息类型
```

在本实例中,Log 文件记录了玩家执行过的所有命令和命令执行的时间,每个参加游戏的玩家机器上都有一份,而且必须完全一样。如果出现不同,则表示游戏发生了不同步,即出错了。这时,就可以对比不同机器上的 Log 文件来查找错误。下面是一个经过简化的 Log 文件内容示例:

```
Init DirectDraw...OK!
Init AStar...OK!
Init Map...OK!
Init Game Successful!
当前帧:165
```

```
此轮命令数量:1
命令类型:150 <行走>
目的地坐标:65,91ManCount=3
ID=3 出发坐标:59,93
ID=4 出发坐标:60,92
ID=5 出发坐标:59,91
……结束……
当前帧:180
此轮命令数量:1
命令类型:150 <行走>
目的地坐标:66,90ManCount=3
ID=3 出发坐标:61,93
ID=4 出发坐标:60,92
ID=5 出发坐标:61,91
……结束……
当前帧:3480
此轮命令数量:1
命令类型:150 <行走>
目的地坐标:194,180ManCount=3
ID=3 出发坐标:65,87
ID=4 出发坐标:66,92
ID=5 出发坐标:66,90
……结束……
当前帧:3510
此轮命令数量:1
命令类型:215
……结束……
当前帧:3915
此轮命令数量:1
命令类型:216
……结束……
当前帧:4305
此轮命令数量:1
命令类型:150 <行走>
目的地坐标:189,173ManCount=5
ID= 200 出发坐标:183,165
ID= 201 出发坐标:185,163
```

```
 ID =  202 出发坐标 :184,164
ID =  203 出发坐标 :182,164
ID =  205 出发坐标 :183,163
……结束……
当前帧 :6830
此轮命令数量 :1
命令类型 :160
当前帧 :7715
此轮命令数量 :1
命令类型 :150 < 行走 >
目的地坐标 :202,178ManCount = 2
ID = 5 出发坐标 :201,171
ID = 12 出发坐标 :196,172
……结束……
当前帧 :7720
此轮命令数量 :1
命令类型 :160
……结束……
……
Game Over
```

把每一个同步周期里发生的所有玩家操作和操作发生时间都记录下来,一旦出现不同步的情况,就把参加游戏的玩家的 Log 文件拿来比对,看是什么操作上出现了问题,还是整个同步机制出了问题。因为 Log 文件行数巨大,还可以使用 DOS 命令 FC 来帮助找到这些文件中第一个发生不同的地方。

15.14 实现录像功能

如果把上面提到的 Log 文件定义成一种易于读入和解释的格式,并且把所有的信息都完整地记录下来,那么就可以重新获得每个玩家的每个操作和发生时间,并可以在游戏中重现它们。因此按照 Log 文件的记录来重新运行游戏而不是玩家直接操作,就可以完全重现一场战斗,就好像看录像一般。游戏循环运行的次数直接代表了游戏内部数据的状态,而玩家的操作是促发游戏数据改变的惟一原因,而且操作发生的时间是以游戏循环次数为单位的,因此得以重现游戏过程,还可以用不同的速度来“播放”。有一些即时战略游戏,如《星际争霸》和《帝国时代》,都先后加入了重现战役的功能,其原理相同。

15.15 Lobby 大厅的制作

在第一章中提到过大厅的概念和网络结构，这里将结合本实例讲述的即时战略游戏来具体实现一个大厅服务器。

服务器运行的基本流程如下：

①服务器启动，等待客户端连入。

②维护一个当前所有连入的客户端的数据列表，数据内容包括 IP 地址、名字等。

③如果客户端创建了一个游戏服务（房间），则把该游戏服务加入到一个列表中，并向其他客户端刷新列表数据，触发该客户端成为游戏的主机。

④当有客户端选择加入某个游戏服务时，帮助它连接到主机。

⑤维护当前所有游戏服务的参加游戏人数、游戏是否已经正式开始等信息，这些信息由服务器向建立服务的主机玩家获取。

⑥当一场游戏结束后，服务器从主机玩家那里取得游戏结果和分数等信息，并更新排行榜。

服务器本身其实是一个比较简单的基于 Socket 的网络服务程序。在前面的章节中已经很详细地介绍了有关 Socket 的知识，这里直接进入程序代码的编写。本书配套光盘的服务器程序（/补充与提高/第 15 章补充——服务器程序）是用 MFC 和 WinSock 写的一个 Lobby 大厅的示例程序，几个关键类如下：

CServerApp 为应用程序类，主要功能是控制服务器的运行流程，包括初始化、建立 Socket 监听、维护连入的 Player 数据等；

CServerSocket 为用来监听服务器的 Socket；

CReceiveSocket 为用来保持和客户端连接的 Socket；

CLog 为用来保存和输出到对话栏等各种信息。

这里一些不相关的代码和 MFC 框架自动生成的无关代码均已去掉。

除了补充内容中的这些代码之外，还有一些关于界面显示的处理，目的是把客户端的连接情况以及当前的状态呈现在窗口界面中，以方便查看和管理。这里就不列出具体代码，读者可以参阅配套光盘。有了这些功能，一个简单的 Lobby 大厅服务器程序就完成了。在本实例中客户端相应连接的部分也是使用 WinSock 编写的，而启动客户端游戏使用 DirectPlay 的 Lobby 部分功能，可以帮助玩家直接进入他人建立的游戏局，而不是仅仅获得一个游戏局的 IP 地址然后靠玩家手动地启动游戏进入。

将来的游戏大厅设计的趋势应该是同一个大厅程序支持各种网络对战游戏，而这些游戏的客户端启动游戏部分按照某种规范统一编写，如 DirectPlay 的 Lobby 就试图建立一种这样的规范。这样就可以做到玩家更方便地进入不同游戏的游戏局。

附录一 Internet上的开发资源

http ://www. gamedev. net 综合性游戏开发网站

http ://www. libsdl. org 跨平台的游戏开发库 SDL

http ://www. swissquake. ch/chumbalum-soft 3D 模型(人物)系统 milkshape3D

http ://cal3d. sourceforge. net 3D 人物系统

www. genesis3d. com 完整而强大的 3D 引擎

http ://www. flipcode. com 综合性游戏开发网站

http ://www. gamasutra. com

综合性游戏制作网站,上面有非常多的质量很好的技术文章

http ://www. peroxide. dk/download/tutorials/tut10/pxdtut10. html

关于 3D 多边形碰撞检测不错的文章,还有一个应用实例,不过缺少源代码

http ://nehe. gamedev. net

上面有很强的 OpenGL 全系列教程,还有一些其他关于游戏制作的教程

http ://www. codeproject. com

编程网站,程序员必去,上面的技术文章类别丰富,而且通常会有源代码示例

http ://www. fanqiang. com(中)

Linux 编程网站,有不少技术文档

http ://www. xoreax. com/main. htm

局域网内联合编译工具,和 VC + + 完好集成,机器越多速度越快,快得让你不敢相信,再也不惧怕数万行的代码编译。不过付费前只能试用一段时间。