TURING 图灵程序设计丛书

高并发×低延迟 如何并存?

游戏开发者奥斯卡 CEDEC AWARDS 2011 最优秀著作奖

网络游戏核心技术与实战

MMORPG / 动作类游戏 / 多人游戏 / 非同步I/O / 套接字
RPC / 事件驱动 / 实时 / 在内存中 / 多核 / 并行处理 / 吞吐量 / 自动化测试
空间分配 / 积分管理 / 支付 / 基础设施 / 开发体制

【日】中嶋谦互 著
毛姝雯 田剑 译

人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：网络游戏核心技术与实战

作者：（ 日） 中嶋谦互

译者：毛姝雯，田剑

ISBN：978-7-115-34935-4

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

您购买的图灵电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

图灵社区会员 ptpress（libowen@ptpress.com.cn） 专享 尊重版权

版权声明
前言
关于本书
本书结构
案例游戏和视频录像
案例游戏的运行环境
关于通信中间件 VCE
使用须知 ※ 请仔细阅读 ※
本书附加信息
致谢
专业术语
第 0 章 [快速入门]网络游戏编程：网络和游戏编程的技术基础
0.1 网络游戏开发者所需了解的网络编程基础
0.1.1 网络编程是必需的
0.1.2 网络编程与互联网编程
0.1.3 互联网编程的历史和思想
0.1.4 OSI 参考模型——透明地处理标准和硬件的变化
0.1.5 网络游戏系统及其层次结构
0.1.6 套接字 API 的基础知识
0.1.7 网络游戏和套接字 API——使用第 4 层的套接字 API
专栏 网络编程的特性和游戏架构的关系——服务器、客户端所需具备的性能和功能
0.2 套接字编程入门——处理多个并发连接、追求性能
0.2.1 通信链路的确定（复习）
0.2.2 套接字 API 基础——一个简单的 ECHO 服务器、ECHO 客户端示例
0.2.3 TCP 通信链路的状态迁移和套接字 API

0.2.4　处理多个并发连接——通向异步套接字 API 之路
0.2.5　同步调用（阻塞）和线程
0.2.6　单线程、非阻塞、事件驱动——使用 select 函数进行轮询
0.2.7　网络游戏输入输出的特点——单线程、事件驱动、非阻塞
0.2.8　网络游戏和实现语言
0.2.9　充分发挥性能和提高开发效率——从实现语言到底层结构
0.2.10　发挥多核服务器的性能
专栏　输入输出的实现方针和未来提高性能的可能性
0.2.11　多核处理器与网络吞吐量——网络游戏与小数据包
0.2.12　简化服务器实现——libevent
0.3　RPC 指南——最简单的通信中间件
0.3.1　通信库的必要性
0.3.2　网络游戏中使用的 RPC 的整体结构
0.3.3　［补充］UDP 的使用
0.4　游戏编程基础
0.4.1　游戏编程的历史
0.4.2　采用“只要能画点就能做出游戏”的方针来开发入侵者游戏
0.4.3　游戏编程的基本剖析
0.4.4　游戏编程精粹——不使用线程的“任务系统”
0.4.5　两种编程方法的相似性——不使用线程
0.5　小结
专栏　确保开发效率和各平台之间的可移植性
第 1 章　网络游戏的历史和演化：游戏进入了网络世界
1.1　网络游戏的技术历史

1.1.1　网络游戏出现前的 50 年
1.1.2　20 世纪 50 年代前：计算机诞生
1.1.3　20 世纪 50 年代：早期的电子游戏
1.1.4　20 世纪 60 年代：各种颇具影响的机器登上历史舞台
1.1.5　20 世纪 70 年代：网络游戏的基本要素
1.1.6　20 世纪 80 年代：网络对战游戏登场
1.1.7　20 世纪 90 年代：游戏市场扩大
1.1.8　本世纪前 10 年的前期：网络游戏商业化
1.1.9　本世纪前 10 年的后半期：基于 Web 浏览器的 MMOG 在商业上获得成功
1.1.10　2010 年之后：究竟会出现怎么样的游戏呢？
1.2　从技术变迁看游戏文化和经济圈
1.2.1　解读技术发展图
1.2.2　3 个圈（三大范畴）
1.2.3　两个游戏经济 / 文化圈
1.2.4　文化、经济与技术的关系
1.3　小结
专栏　成为出色的网络游戏开发程序员的条件
第 2 章　何为网络游戏：网络游戏面面观
2.1　网络游戏术语的定义
网络游戏的 4 个层面
2.2　网络游戏的物理层面
2.2.1　物理构成要素
2.2.2　物理模式 / 物理上的网络构成
2.3　网络游戏的概念层面
2.3.1　网络游戏及其基本结构
2.3.2　游戏进行空间——进行游戏时所需的所有信息

2.3.3　游戏的进展——游戏进行空间的变化
2.3.4　共享相同的游戏进展
2.4　网络游戏的商业层面
2.4.1　商业层面的要求
2.4.2　有效地招募测试玩家——网络游戏与测试
2.4.3　不断更新——网络游戏的运营和更新
2.4.4　节约服务器数量和带宽——网络游戏开支的特殊性
2.4.5　从小规模开始，确保可扩展性——将风险降到最低，不要错过取胜的机会
2.4.6　提供多种收费方式——收费结算方式的变化
2.4.7　低价、快速地根除攻击者——攻击、非法行为及其对策
2.4.8　减少服务器停止的次数和时间——不要让玩家失望
2.4.9　反馈游戏结果——日志分析和结果的可视化
2.4.10　更容易地与其他玩家相遇——玩家匹配
2.5　网络游戏的人员和组织
2.5.1　与网络游戏服务的运营相关的人员
2.5.2　网络游戏服务运营的 3 项专门职责
2.5.3　开发团队
2.5.4　运维团队
2.6　网络游戏程序员所需的知识
2.6.1　网络游戏程序员所需的技术和经验
2.6.2　各种网络游戏开发知识
2.7　支持网络游戏的技术的大类
支持网络游戏的技术的 4 种形式
2.8　影响开发成本的技术要素
2.8.1　网络游戏与如今的开发技术
2.8.2　支持网络游戏主体的 3 大核心

2.9　小结
专栏　网络游戏编程的最大难点——解决冗余和异步的问题
第 3 章　网络游戏的架构：挑战游戏的可玩性和技术限制
3.1　游戏编程的特性——保持快速响应
3.1.1　响应速度的重要性——时间总是不够的
3.1.2　将数据存放在内存中的理由——游戏编程真的有三大痛苦吗
3.1.3　每 16 毫秒变化一次——处理的信息及其大小
3.1.4　大量对象的显示——CPU 的处理能力
3.1.5　无法预测玩家的操作——游戏状态千变万化
3.1.6　必须将游戏数据放在 CPU 所在的机器上
3.2　网络游戏特有的要素
3.2.1　通信延迟——延迟对游戏内容的限制
3.2.2　带宽——传输量的标准
3.2.3　服务器——成本、服务器数量的估算
3.2.4　安全性——网络游戏的弱点
3.2.5　辅助系统（相关系统）
3.3　物理架构详解——C/S 架构、P2P 架构
3.3.1　基本的网络拓扑结构
3.3.2　物理架构的种类
3.3.3　C/S 架构——纯服务器型、反射型
3.3.4　P2P 架构
3.3.5　C/S ＋ P2P 混合型架构
3.3.6　ad-hoc 模式
专栏　游戏客户端是什么
3.4　逻辑架构详解——MO 架构
3.4.1　MO、MMO 是什么？——同时在线数的区别
3.4.2　MO 架构、MOG

3.4.3　同步方式——获得全体玩家的信息后，游戏才能继续
3.4.4　同步方式 / 全网状结构的实现——所有终端都拥有主数据
3.4.5　同步方式 / 星型结构——暂时将输入信息集中到服务器上
3.4.6　异步方式——接受各终端上游戏状态的不一致
3.4.7　三大基本要素：自己、对手、环境——异步实现的指导方针
3.4.8　❶ 自己和对手——对战游戏和玩家之间往来数据的抽象程度
3.4.9　保持结果一致性的方法——两种覆盖方式
3.4.10　❷ 自己和环境——可使用物品的格斗游戏和互斥控制
3.4.11　互斥控制的实现——采用与同步方式类似的机制来实现异步方式
3.4.12　状态会自动变化的环境——静态环境和动态环境
3.4.13　❸ 对手和环境的关系
3.5　逻辑架构详解——MMO 架构
3.5.1　MMO 架构、MMOG——在大量玩家之间共享长期存在的游戏过程
3.5.2　MMOG 的结构
3.5.3　大型多人网络游戏（MMO）
3.6　小结
专栏　设法改善网页游戏的画面显示间隔
第 4 章　[实践]C/S MMO游戏开发：长期运行的游戏服务器
4.1　网络游戏开发的基本流程
4.1.1　项目文档 / 交付物
4.1.2　开发的进行和文档准备的流程

4.1.3 技术人员的文档 / 交付物
4.2 C/S MMO 游戏的发展趋势和对策
4.2.1 C/S MMO 游戏的特点
4.2.2 C/S MMO 架构（MMO 架构）特有的游戏内容
4.3 策划文档和 5 种设计文档——从虚构游戏 K Online 的开发中学习
4.3.1 考虑示例游戏的题材
4.3.2 详细设计文档
4.3.3 MMOG 庞大的游戏设定
4.3.4 5 种设计文档
4.3.5 设计上的重要判断
4.4 系统基本结构图的制定
4.4.1 系统基本结构图的基础
4.4.2 服务器必须具有可扩展性——商业模式的确认
4.4.3 各种瓶颈——扩展方式的选择
4.4.4 解决游戏服务器 / 数据库的瓶颈
4.4.5 什么都不做的情况（1 台服务器负责整个游戏世界）
4.4.6 空间分割法——解决游戏服务器的瓶颈
4.4.7 实例法——解决游戏服务器的瓶颈
4.4.8 平行世界方式——解决数据库瓶颈
4.4.9 同时采用多种方法——应对越来越多的玩家
4.4.10 各种方式的引入难度
4.4.11 各个世界中数据库（游戏数据库）服务器的绝对性能的提高
4.4.12 K Online 的设计估算——首先从同时在线数开始
4.4.13 根据游戏逻辑的处理成本来估算——敌人的行动算法需要消耗多少 CPU

4.4.14 根据游戏数据库的处理负荷进行估算——找到“角色数据的保存频率”与“数据库负荷”的关系
4.4.15 可扩展性的最低讨论结果，追求进一步的用户体验
4.4.16 服务器的基本结构，1制定系统基本结构图
4.5 2进程关系图的制定
4.5.1 2进程关系图的准备
4.5.2 服务器连接的结构——只用空间分割法
4.5.3 服务器连接的结构——使用平行世界方式和空间分割法
4.5.4 使用平行世界方式进行扩展的关键点
4.6 带宽 / 服务器资源估算文档的制定
4.6.1 以进程列表为基础估算服务器资源
4.6.2 以 CPU 为中心的服务器和以存储为中心的服务器
4.6.3 服务器资源的成本估算——首先从初期费用开始
4.6.4 带宽成本的估算
4.6.5 带宽减半的方针——首先是调整策划，然后在程序上下功夫
4.6.6 策划内容的分析对带宽的降低很有效
4.7 4协议定义文档的制定——协议的基本性质
4.7.1 4协议定义文档基础
4.7.2 “协议的基本性质”的要点
4.7.3 协议的种类、以及进程之间关系的种类
4.7.4 8 种类型的协议
4.7.5 C/S MMO 采用 TCP
4.7.6 与“协议的基本性质”的对应
4.8 4 协议定义文档——协议的 API 规范（概要）
4.8.1 协议的实现原则
4.8.2 8 种协议的功能 / 形式概述

4.9　4 协议定义文档——协议的 API 规范（详细）
4.9.1　协议 API 规范（详细）的制定
4.9.2　API 的函数定义
4.9.3　常量定义
4.9.4　API 的调用时序
4.9.5　时序图制定的要点
4.10　4协议定义文档——数据包的格式
4.10.1　C/S MMO 主要采用 TCP（复习）
4.10.2　C/S MMO 使用包含专用字节数组的二进制协议
4.10.3　二进制协议的实现——首先从术语的整理开始
专栏　C/S MMO 的压缩和加密
4.11　5数据库设计图
4.11.1　要在编程之前进行对重要的表进行设计
4.11.2　C/S MMO 中的数据库实现的历史变迁
4.11.3　整理 K Online 所需的表
专栏　百花缭乱的 KVS——未来 C/S MMO 中数据库的使用情况
4.11.4　数据库性能预测
4.12　服务器 / 客户端软件 + 中间件——实践中不可或缺的开发基础
4.12.1　网络游戏的中间件
4.12.2　开发的基础软件——可以立刻尝试的 C/S MMO 开发体验
4.13　程序开发中的基本原则
4.13.1　如何开始编程、如何继续编程
4.13.2　数据结构优先原则——基本原则1
4.13.3　实现数据结构之前的讨论——出现在画面上和不出现在画面上的元素

4.13.4 维持可玩状态的原则——基本原则2
4.13.5 后端服务器的延后原则——基本原则3
4.13.6 持续测定的原则——基本原则4
4.14 C/S MMO 游戏 K Online 的实现——编程开始!
4.14.1 开发的安排
4.14.2 K Online 中的分工计划
4.14.3 K Online 中“框架阶段”和“原型阶段”的区别
4.14.4 [步骤 1～2] 框架～原型阶段
专栏 每一步的进度管理形式
4.14.5 “不实际运行起来是不会理解的！”——游戏开发的特殊性
专栏 C/C++ 以外的语言
4.14.6 框架的整体结构
专栏 VCE 是什么
4.14.7 以怎样的顺序来编写代码
4.14.8 首先编写协议定义文件 k.xml——开发流程1
4.14.9 协议定义的要点
4.14.10 通信连通确认：ping 函数
4.14.11 账户登录和账户认证：signup 函数、authentication 函数
4.14.12 角色创建：createCharacter 函数
4.14.13 登录：login 函数
4.14.14 在地面上移动：move 函数、moveNotify
4.14.15 编写 gmsv/Makefile——开发流程 2
4.14.16 从示例中复制 gmsv/climain.cpp 和 gmsvmain.cpp——开发流程 3
专栏 dbsv 服务器代码的自动生成——后端服务器实现的简化

4.14.17 自动测试客户端 autocli 的实现——开发流程 4
专栏 gmsv 中线程的使用
4.14.18 图形客户端 cli 的创建和运行确认——开发流程
4.14.19 框架之后的开发——开发流程、后续事项
4.15 总结
第 5 章 [实践] P2P MO游戏开发：没有专用服务器的动作类游戏的实现
5.1 P2P MO 游戏的特点和开发策略
5.1.1 P2P MO 和动作类游戏——游戏的状态频繁发生改变
5.1.2 RPC 和共享内存
5.1.3 P2P MO 游戏的特点——和 C/S MMO 游戏的比较和难点
5.1.4 P2P MO 游戏的优点
5.1.5 从概要设计开始考虑[多人游戏模式]
5.2 J Multiplayer 游戏开发案例的学习——和 K Online 的不同
5.2.1 J Multiplayer ——和 K Online 的比较
5.2.2 P2P MO 游戏开发的基本流程
5.2.3 P2P MO 游戏开发的交付产品——开发各个阶段需要提交的资料
5.2.4 和 C/S MMO 的数据量/规模的比较
5.3 P2P MO 游戏的设计资料
5.3.1 系统基本结构图
5.3.2 进程关系图
5.3.3 带宽 / 服务器资源计算资料
5.3.4 通信协议定义资料和 API 规格
专栏 什么是“游戏逻辑”
5.3.5 带宽消耗量的估算

5.3.6 其他资料
5.4 客户端 / 服务器软件 + 中间件、基本原则
5.4.1 P2P MO 开发的最终交付产品
5.4.2 P2P MO 中使用的中间件
5.4.3 编程时应该注意的基本原则——针对 P2P MO 游戏
5.5 P2P MO 游戏 J Multiplayer 的实现——正式开始编程
5.5.1 J Multiplayer 的编程计划
5.5.2 开发流程——K Online 的回顾
5.5.3 J Multiplayer 开发阶段——开发顺序和内容
5.5.4 第 1 阶段的要点
5.5.5 客户端程序的开发案例
5.5.6 “共享内存方式”的实现——开始编码
5.5.7 P2P MO 游戏开发中该如何防止发生竞争状态
5.5.8 共享内存开发方式该如何编码——共享内存开发方式和 RPC 开发方式的比较
5.5.9 SyncValue 类
专栏 数据中心的地理位置分布
5.6 支持 C/S MO 游戏的技术 [补充]
5.6.1 C/S MO 和 NAT 问题
5.6.2 什么是 NAT 问题
5.6.3 NAT 遍历——解决 NAT 问题建立通信路径的技术
5.6.4 NAT 问题的实际解决方法
5.6.5 中继服务器
5.6.6 中继服务器的折衷方案
5.7 总结
第 6 章 网络游戏的辅助系统：完善游戏服务的必要机制
6.1 辅助系统需要的各种功能
6.1.1 现有服务提供的辅助系统功能

6.1.2 现有服务的功能一览
6.1.3 网页游戏开发方法和客户端游戏的开发方法
6.2 交流 / 通信功能
6.2.1 玩家匹配 P2P MO
6.2.2 游戏大厅 P2P MO
6.2.3 中继服务器 P2P MO
6.2.4 聊天 P2P MO C/S MMO
6.2.5 邮件 P2P MO C/S MMO
6.2.6 好友列表 P2P MO C/S MMO
6.2.7 玩家状态 P2P MO C/S MMO
6.2.8 加锁服务器 P2P MO C/S MMO
6.2.9 黑名单 P2P MO C/S MMO
6.2.10 语音聊天 P2P MO C/S MMO
6.3 游戏客户端实现相关的辅助系统
6.3.1 玩家成绩管理 P2P MO C/S MMO
6.3.2 存储功能 P2P MO
6.3.3 （游戏客户端）更新 P2P MO C/S MMO
6.3.4 排行榜 P2P MO C/S MMO
6.4 运营辅助系统
新闻发布 P2P MO C/S MMO
6.5 付费相关的辅助系统
6.5.1 付费认证 P2P MO C/S MMO
6.5.2 虚拟货币管理 P2P MO C/S MMO
专栏 C/S MMO 游戏的收入
6.6 其他辅助功能
6.6.1 游戏数据浏览 / 查询工具 P2P MO C/S MMO
6.6.2 敏感词过滤 P2P MO C/S MMO
6.7 本章小结

第 7 章　支持网络游戏运营的 基础设施：架构、负荷测试和运营
7.1　基础设施架构的基础知识
7.1.1　C/S MMO 和 P2P MO 的基础设施（概要）
7.1.2　基础设施架构需要进行的工作 ——从开发整体来看
7.1.3　基础设施的成本估算
7.1.4　成本的概念、单位
7.1.5　网络游戏服务器在一定程度上可以接收的条件
7.1.6　硬件、信息设备
7.1.7　软件
7.1.8　数据中心相关的成本
7.1.9　服务费（数据中心以外）
7.1.10　网络带宽费
7.1.11　电费
7.2　开发者需要知道的基础设施架构技巧
7.2.1　服务规模的扩大 / 缩小——用户数和需要的基础设施规模
7.2.2　典型的环境
7.2.3　负荷曲线
7.2.4　面向开发者的基础设施架构要点
7.2.5　服务器部署
7.2.6　登台环境
7.2.7　服务器的监控、生死监控
7.2.8　日志输出 / 管理
7.3　K Online 、J Multiplayer 游戏的基础设施架构
7.3.1　K Online 的基础设施
7.3.2　J Multiplayer 的基础设施
7.4　负荷测试
7.4.1　负荷测试的准备

7.4.2 K Online 在生产环境的负荷测试
7.4.3 负荷测试时使用的服务器监控命令
7.4.4 J Multiplayer 在生产环境下的负荷测试
7.5 游戏上线
7.5.1 游戏上线前——从确认安全设定开始
7.5.2 游戏上线后 ——系统监控
7.5.3 服务器的组群化
7.5.4 故障发生时的应对
7.6 本章小结
第 8 章 网络游戏的开发体制：团队管理的挑战
8.1 游戏的策划内容和开发团队 网络游戏特有的挑战
8.1.1 游戏的策划内容是团队管理的关键
8.1.2 游戏数据的持久化
8.1.3 游戏中玩家之间的关系
8.1.4 游戏结果的共享范围
8.1.5 聊天系统的内容
8.1.6 维护和升级的计划
8.1.7 代码规模——如果需要迭代的代码过多就会遇到问题
8.2 网络游戏开发团队的实际情况——和一般软件开发相同的地方
8.2.1 工作分配
8.2.2 持续提升网络游戏程序员技能的方法
8.2.3 项目管理术——游戏开发和 Scrum
8.2.4 开发环境的选择
8.2.5 项目的移交——理所当然的事情也需要仔细归纳总结
8.3 本章小结
专栏 网络游戏开发的成本

版权声明

ONLINE GAME O SASAERU GIJUTSU by Kengo Nakajima

Copyright © 2011 Kengo Nakajima

All rights reserved.

Original Japanese edition published by Gijyutsu-Hyoron Co., Ltd., Tokyo

This Simplified Chinese language edition published by arrangement with Gijyutsu-Hyoron Co., Ltd., Tokyo in care of Tuttle-Mori Agency, Inc., Tokyo

本书中文简体字版由 Gijyutsu-Hyoron Co., Ltd. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

敬告读者，在实际工作中，运用本书内容所产生的后果，原书作者、软件的开发者和提供者、技术评论社、人民邮电出版社以及译者都不承担任何责任。

“Battle.net” and World of Warcraft are registered trademarks of Blizzard Entertainment, Inc., and World of Warcraft is a copyrighted product of Blizzard Entertainment, Inc., and hereby used with permission.

书中出现的公司名称、产品名称一般为各个公司的注册商标，在书中并没有用™ 、©、® 等符号标记。

前言

关于本书

2011 年 1 月，苹果公司 App Store 的应用程序下载量超过了 100 亿次 [1] 。此外，谷歌公司的 Android Market、Chrome Web Store 以及 Amazon Appstore（亚马逊公司的 Android 应用商店，2011 年 3 月 22 日上线）、Facebook Credits 虚拟货币系统、游戏领域的网络游戏平台维尔福软件公司的 Steam 平台等，无论是面向通用操作系统的应用商店还是支付系统、游戏通信机制等都大量涌现，整个行业迎来了爆发式增长。

> [1] 2013 年的最新统计，App Store 应用累计下载次数超过 600 亿。——译者注

在每个应用商店，下载量最大的分类都是游戏，占到了总下载量的一半以上。Facebook 平台上甚至绝大多数销售额都由游戏贡献。现在，特别是智能手机和 PC 平台、画面精美的 3D 游戏，以及有玩法多样的对战游戏一般都在 10 美元以下，已经非常便宜了。再过几年，那些（开发成本）几百亿到上千亿日元的游戏应该也能很容易下载到。游戏是一种非常廉价的娱乐方式，可以满足人们放松心情这一基本需求，是一种主流的娱乐活动。

几乎所有的游戏都在增加联网的功能，让游戏向更实时、画面更加精美、利用更大的数据库、算法更加智能的方向发展。

随着市场规模的扩大，各种游戏的开发方式受到了更多的关注。虽说都是游戏，但将纸团扔进垃圾桶的 iPhone 游戏 *Paper Toss* 和即时通信大规模在线用户《魔兽世界》，在开发技术上是截然不同的。如果将它们比作交通工具，则一个是自行车，一个是新干线。它们在开发成本和销售额上有几万倍的差距，需要的技术也完全不同。只有拥有大量人才和资金的企业才有能力开发并且运营大型实时网络游戏。但如果换成 Web 服务的话，即使承载了数十万的用户，开发者一人也能够胜任开发和维护工作。两者之所以有这样的差别，我想其中一个原因就是开发网络游戏所需要的知识并没有得到充分的共享。

本书以实时通信、大数据量通信的多人网络游戏开发为中心，详细介绍了普通开发者如何在不使用昂贵的中间件或者特殊开发环境的基础上，独自从零开始实现有趣的多人网络游戏系统，并讲解了 C/S MMO 游戏和 P2P MO 游戏这两个典型的开发案例。此外还从游戏运营和基础设施架构等角度，向读者展现了支持网络游戏技术的全貌。本书的内容主要面向游戏开发技术人员，但无论是对游戏制作人还是运营者，本书都非常具有参考价值。

希望不仅仅是大企业，今后更多的独立开发者或者小企业也能够开发网络游戏，为玩家提供更多的充满创意的产品。

2011 年 2 月　中嶋谦互

网络游戏的开发技术每天都在不断进步。

如果对本书有什么疑问或者建议可以通过 Twitter 和 @ringo 联系，欢迎提问。

本书结构

本书的结构如下。要理解书中与编程相关的章节（第 0 章、第 4 章、第 5 章），需要具备一定的 C 或者 Java 等一般编程语言基础知识。其他章节（第 1～3 章、第 6～8 章）的内容不需要编程知识也可以理解。

第 0 章　[快速入门] 网络游戏编程

网络和游戏编程的技术基础

本章详细介绍专业的网络游戏开发者和其他行业的开发者（Web 开发者或者企业应用开发者等）在技术上的不同点。

第 1 章　网络游戏的历史和演化

游戏进入了网络世界

第 2 章　网络游戏到底是什么

网络游戏面面观

第 3 章　网络游戏的架构

挑战游戏的可玩性和技术限制

总结了网络游戏的历史、背景、商业模式、不同分类的区别以及架构等背景知识。

第 4 章　[实践]CS MMO 游戏开发

长期运行的游戏服务器

第 5 章　[实践]P2P MO 游戏开发

没有专用服务器的动作类游戏的实现

介绍了可以涵盖大部分网络游戏的两种实现方式，通过案例游戏和实际代码带领读者体验开发流程。

第 6 章　网络游戏的辅助系统

完善游戏服务的必要机制

第 7 章　支持网络游戏运营的基础设施

构筑、负荷测试、开始运营

第 8 章　网络游戏的开发体制

团队管理的挑战

介绍了对于网络游戏非常重要的运营相关的辅助技术和思维方式。

关于本书的结构：

同类书籍一般会首先针对网络游戏这个主题介绍相关的定义，说明背景，提出课题，简要地说明解决方法，然后通过案例程序具体介绍开发技术。本书也基本上沿用了这个方式，但是在书的开始部分增加了第 0 章，用来简要介绍开发网络游戏时的网络编程和游戏编程知识，方便读者快速学习。

第 0 章尽可能涵盖了全书的专业词汇，只需要快速浏览即可了解实际开发中需要用到的技术。如果在接下来的章节遇到了不太理解的内容，也可以回到第 0 章查阅以加深理解。

如果读者对第 0 章介绍的技术都非常熟悉，那一定可以在任何一家网络游戏开发公司成为核心开发人员。如果对这些知识都不太了解也不用担心，第 0 章之后会详细解释为什么需要用到这些技术，可以反复查阅相关内容进行学习，这样一定可以加深理解。

案例游戏和视频录像

由于篇幅所限，书中所附案例采用了一部分伪代码。同时我们为大家准备了可以实际运行的案例游戏（包括 C/S MMO 和 P2P MO 两种类型）作为参考。下载地址请参见稍后的本书附加信息。源代码仅仅是为了便于读者理解编程方法，请作为游戏运行情况和书中代码案例的参考。

除此之外，我们还为大家准备了以上两种案例游戏实际运行时的视频录像，请参见稍后的本书附加信息。

案例游戏的运行环境

为了正常运行本书的案例游戏，需要确保具备以下软件环境。使用时请确认拥有合适的软件授权。

- Mac OS X v10.6（Snow Leopard）
- Xcode http://www.apple.com/jp/macosx/developers/

- Boost 1.41.0 http://www.boost.org/LICENSE_1_0.txt
- SDL 1.2.14 http://www.libsdl.org/license-lgpl.php
- 通信中间件 VCE ※ 请参考第 4 章相关章节
- MySQL 等其他软件

※ 详细清单请参考压缩包中的 readme.txt 文件。

关于通信中间件 VCE

史克威尔艾尼克斯公司拥有通信中间件 VCE 的著作权和知识产权。因为本书旨在提高行业的技术水平，所以在本书规定的范围内（包含附属规定）可以授权使用该软件。

购买本书的读者，可以访问稍后的本书附加信息中的 URL，接受技术评论社网站的附属规定后可以得到授权使用 VCE（技术评论社授权版）。

另外，本书案例游戏的压缩包中所含的 VCE 程序为试用版，在启动最多一小时后会无法继续通信。史克威尔艾尼克斯公司、原书作者、技术评论社、人民邮电出版社以及译者不承担任何因为使用 VCE 所造成的损失。

使用须知 ※ 请仔细阅读 ※

- 案例游戏的压缩包中附带的数据和程序只是作为本书正文相关内容的参考和辅助材料。请不要用作其他目的。
- 史克威尔艾尼克斯公司、程序或者数据的作者、开发者 / 提供者、原书作者、技术评论社、人民邮电出版社以及译者不承担任何因为使用本书案例游戏程序或数据所造成的损失。请读者自行承担相应责任。
- 使用关于本书所收录的软件、程序前请注意，对于相关软件的信息和使用方法，我们不接受包括电话在内的任何方式的技术支持。

- **书中的案例游戏的程序代码是完全开源的，可以不经任何许可自由使用。**
- **其他的注意事项请参考案例游戏压缩包内的 readme.txt 文件。**

本书附加信息

本书的附加信息包含了案例游戏的下载地址和视频下载地址，如下所示。

- **案例游戏的下载地址和本书的主页：**

  →http://gihyo.jp/book/2011/978-4-7741-4580-8/support

- **游戏视频和特别内容下载地址：**

  →http://gihyo.jp/magazine/wdpress/plus/978-4-7741-4580-8

※ 请注意，上述下载地址可能会有突然中断访问的情况。

致谢

本书内容基于笔者到目前为止从参与的网络游戏开发项目中获得的经验。项目很多，在此就不一一列举了，我之所以获得如此多宝贵的经验，离不开各个项目组的同仁们的帮助。

此外还要感谢史克威尔艾尼克斯公司秉承提高业界技术水平的宗旨，为本书无偿提供通信中间件 VCE。

最后，在本书写作的近两年的时间中，我放弃了新年假期，减少了陪伴孩子的时间，给家里增添了许多负担。特别感谢妻子能够给与充分的理解和支持。

专业术语

本书涉及的内容比较广，使用了很多专业术语。下面整理了本书中出现的网络 / 游戏 / 编程相关的专业术语作为辅助资料。因使用场合和上下文的不同，每个术语的意思可能会发生变化，笔者基于本书内容对以下术语进行说明，供大家参考。书中的网络环境是指互联网，服务器操作系统为 Linux。

16 毫秒 / 帧速率 （Frame Rate）

电子游戏使用的光栅显示器是普通电视时，图像一般每秒更新 60 次。图像更新的时间叫做帧，1 秒 60 次即 1 次 16 毫秒（0.0167 秒 =16.7 毫秒）。16 毫秒是玩家可以识别的游戏画面改变的最短时间间隔。

ARPG （Action Role Playing Game）

角色扮演类游戏中动作性较强的实时游戏，也指包含冒险游戏特征的游戏。

bot

外挂。模拟游戏玩家自动访问游戏服务器、高效率地进行游戏、积累分数以及进行恶意的经济欺诈的程序。测试外挂是指开发者准备的用来自动化测试的客户端程序。

CPU 周期 （CPU Cycle）

CPU 处理操作的最小单位。1GHz 的 CPU 一秒有 10 亿个 CPU 周期，以执行的命令数而言，1 秒可以执行 10 亿次。

根据命令类型的不同，执行需要的 CPU 周期少则不到 1 个周期，多则有几百个。

FPS （First-Person Shooter）

第一人称射击游戏。

I/O （Input/Output）

输入 / 输出。包括网络 I/O、磁盘 I/O 等。服务器程序的 I/O 基本都是网络 I/O。

MMO （Massively Multiplayer Online）

大型多人网络游戏，本书是指有大规模用户、多人在线的网络游戏。也叫 MMOG。

MO （Multiplayer Online）

多人网络游戏，本书是指参与用户相对较少的网络游戏。也叫 MOG。

RPC （Remote Procedure Call）

远程过程调用，是指调用其他计算机的处理。例如，当客户端需要命令服务器做某个处理然后得到返回结果时会使用该技术。

RPG （Role Playing Game）

角色扮演游戏，根据游戏背景设定，由玩家扮演特定角色的游戏。

TCP （Transport Control Protocol）

传输控制协议，支撑整个互联网的可靠数据通信协议。可以根据需要续传 IP 数据包，确保大的数据可以正常传输。但是，在连接速度较慢时，为了提高传输效率需要占用大量的内存。

World of Warcraft （《魔兽世界》）

暴雪公司过去 5 年全世界市场占有率第一的 MMORPG。总注册用户 1200 万，仅在中国，同时在线用户就达 100 万。

并行 （Parallel）

包括物理上的多个处理同时进行，以及时间上的并发（Concurrent）处理。就像 CPU 中的命令和任务之间的区别。通过并行处理提高速度比较困难，所以基本方针是充分考虑处理器计算能力，通过在策划层次进行讨论，或者在算法上下功夫减少计算量。

部署（Deploy）

是指部署应用程序。服务器部署是指将最新版服务器程序安装到各个服务器上来更新版本的相关操作。

持久性（Persistent）

在数据库中，持久性是指需要持久化的时间长度，包括游戏玩法中必要的时间和游戏进行所需时间。竞速游戏的数据一般只需要保持几分钟，之后就可以丢弃，所以持久性较低，需要保存的数据量也比较小。但是 MMORPG 等不断进行的游戏需要较高持久性，数据量也比较大。根据持久性需求的不同，数据应该以什么形式、用什么物理介质来保存也会有所区别。

带宽（Bandwidth）

是指网络游戏开发中，通过网络传输数据的传输速率。也叫带宽幅度。

多进程编程（Multi-Process Programming）

灵活使用多个进程的编程方式。同时运行多个进程可以有效利用多核 CPU 的处理能力。

辅助系统（Additional System）

相对于游戏主体内容以外的辅助功能系统，例如玩家匹配、玩家成绩管理（积分管理）、排名以及通信功能等。多数情况下可以使用第三方的程序库或者服务。

负荷（Load）

是指 CPU 或者网络等承载的工作量。例如，处理复杂计算时 CPU 的负荷比较高。发送和接收大量数据时网络的负荷较高。许多场合都可

以使用，例如 CPU 负荷、I/O 负荷以及服务器负荷等。

负载均衡 （Load Balancing）

是指分散负荷。例如将一台数据库承担的负荷分散到多台数据库。

共享内存 （Shared Memory）

是指在多个进程间共享内存数据。例如共享运动物体的坐标、种类以及运动方向等信息。

缓存 （Cache）

为了高速读取数据而把数据暂时放在特殊区域。例如，磁盘访问比较慢时，可以把文件内容放在（缓存在）内存中，这样就可以高速读取数据。该机制被广泛应用在 CPU 缓存、缓存内存、浏览器缓存以及缓存服务器等地方。

进程 （Process）

进程是指操作系统上运行的程序的实体，和其他程序相分离，独立运行。进程与进程之间可以访问的资源（内 存、Socket 等）也是分离的。

进程间通信 （Inter-Process Communication）

在多个进程间通信。是指多个进程间传送数据或共享数据的技术。

竞态条件 （Race Condition）

是指同一个资源（内存地址等）被两个以上的使用者访问时发生的程序状态。会引起死锁（Dead Lock，互相等待对方处理结果的情况）等问题。

扩展性 （Scalable）

是指可以扩展系统性能。在网络游戏中需要应对用户的增长和饱和，所以希望性能和功能可以轻松扩展。

浏览器 （Browser）

浏览软件。网络游戏中的游戏浏览器范围较广，泛指将服务器网站管理的游戏进度信息展示给玩家的软件。例如使用 C++ 语言开发的面向 3D 游戏的专用程序，或者 Flash 游戏使用的 Google Chrome 等 Web 浏览器。和一般浏览 Web 服务器数据的 Web 浏览器有所区别。请参考“游戏客户端”的解析。

轮询 （Polling）

定期询问数据是否送达或者是否接收到的机制。太过频繁的轮询会无端增加 CPU 的负荷。

瓶颈 （Bottleneck）

系统中性能最弱的部分。系统的其他部分即使再快，如果有一个地方（瓶颈）处理比较慢，就会影响整体的性能。

冗余 （Redundancy）

是指作为预备而重复配置。游戏数据的冗余是指将数据在不同地方重复保存（主数据和备份数据的关系）。

事件驱动 （Event Driven）

在事件发生时进行处理的编程方式。事件的类型包括接收到数据、鼠标移动等。事件驱动的编程方式常用在网络开发和游戏开发中。

数据包 （Packet）

数据的传输单位。数据包通信是指将数据分割并添加控制信息后发送、接收后再合并的通信方式。TCP 协议的 数据通信单位是数据段（Segment），UDP/IP 协议为数据报（Datagram）。网络游戏开发会经常面临数据包延迟的挑战。

数据中心 （Data Center）

安置提供服务的服务器设施。配备了维持服务器所必需的电源、空调和防灾设施。

套接字 API （Socket API）

处理网络文件描述符中的套接字的 API。个别的函数 / 系统调用（socket、connect、accept）相关内容可以参考第 0 章。

同时连接数 （Number of Simultaneous Connections）

可以同时连接服务的用户数。游戏策划阶段一般指最大的同时在线数。运营阶段是指同时在线用户的瞬间值。和网络连接中（通信线路）的连接数有所区别。

图元 （Sprite）

是指电子游戏中使用的可以高速显示的小图像。事先准备好玩家角色移动等状态的小图片，可以通过指定图像位置，在画面的任意位置显示角色。

吞吐量 （Throughput）

是指系统在一定时间内处理的数量。例如，1 秒处理 1000 次的系统就比只能处理 100 次的系统的吞吐量高。

网络拓扑 （Network Topology）

是指网络中所含的各个计算机以什么结构相连接。计算机是节点，连接叫做边界。包括星状结构、总线结构和网状结构等，可以帮助分析和设计网络结构。

文件描述符 （File Descripter）

在 Unix 系的操作系统中，除了文件以外，网络、块设备等操作系统管理的输入输出资源也采用了文件形式。文件描述符是一个整数（C 语言中的 int 型），操作系统会通过程序分配文件描述符，并使用它来控制数据的输入输出。通过文件描述符，可以让网络的输入输出像文件的输入输出一样进行。

线程 （Thread）

比进程更细分的程序执行单位。共享资源比较多的情况下，线程间的协调比进程更加容易。只使用单一线程执行处理的方法叫做单线程（Single Threading），使用多线程同时进行处理的方法叫做多线程（MultiThreading）。

延迟 （Latency）

处理所需要的时间。例如，如果玩家按下按键到画面反应的时间间隔为 1 秒，那这一延迟就过长了。场合不同，单位也不一样，CPU 访问的时间间隔为纳秒、内存访问是几十到几百纳秒、SSD 访问是几百微秒、硬盘访问是几十毫秒、网络延迟是毫秒到秒。

游戏客户端 （Game Client）

是指在玩家的 PC 或者游戏机等机器上安装的，启动后可以显示游戏画面、接受用户输入输出的软件。例如，需要高性能绘图和输入输出功能的 ARPG 或者 3D 画面的 MMORPG 等网络游戏都需要开发专用的客户端。游戏客户端也可以称为客户端软件。请参考“浏览器”的说明。

游戏逻辑 （Game Logic）

相当于 Web 应用的业务逻辑，是指连接游戏进度信息和用户界面信息的算法。

云 （Cloud）

在云计算（Cloud Computing）中主要是指服务器端的计算机群。在单纯的主机托管中，包括存储、负载均衡、付费系统、日志解析等服务器架构中的计算机资源可以根据需要即时调整。尽管非常方便，但是需要将重要的保密信息和用户资料交给云服务提供商。

在内存中 （on Memory）

是指把数据放在内存中，可以在几个 CPU 时钟周期，也就是几纳秒到几百纳秒之间获取到数据的状态。

中间件 （Middleware）

将应用程序普遍使用的功能进行集成的专业化软件。比如将通信处理等对性能和适应性要求较高的处理做成程序库，并将复杂的处理隐藏起来，通过访问 API 即可完成复杂的工作。

纵向扩展 / 横向扩展 （Scale-up/Scale-out）

纵向扩展是指增加内存、升级 CPU 等，通过提升单台服务器的性能来改善系统性能的方法。横向扩展是指通过增加服务器台数来提供系统性能的方法。单台服务器有性能瓶颈，所以大型网络游戏需要具备可以通过增加服务器数量来横向扩展的性能。

阻塞 / 非阻塞 （Blocking / Non-Blocking）

阻塞是指处理完成之前持续等待。例如，收到数据前持续等待的程序（阻塞程序），在等待期间不能进行其他处理。采用非阻塞（不持续等待）处理可以解决这个问题。也可以叫做同步调用和非同步调用。

第 0 章 [快速入门]网络游戏编程：网络和游戏编程的技术基础

开发网络游戏必须掌握以下两大块技术。即使所处的团队实行分工制度，开发人员也需要对这两方面有所了解。

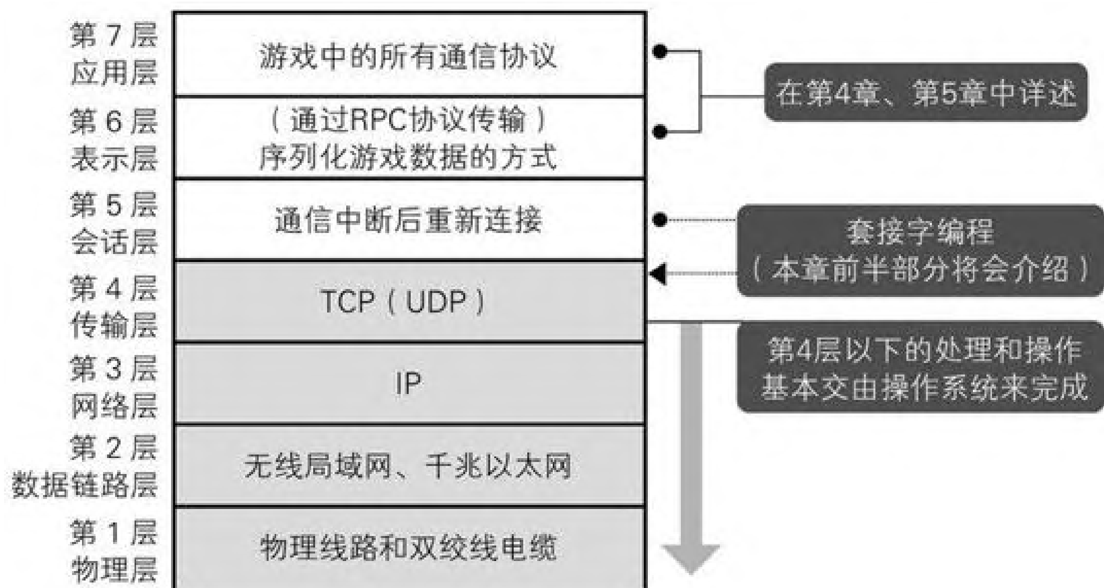
- 网络编程（Network Programming）
- 游戏编程（Game Programming）

为了便于读者更容易理解之后的内容，本章首先介绍一下网络编程和游戏编程的基本概念，对开发网络游戏来说，它们是非常重要的。本章将按照如下顺序进行讲解。

- 网络游戏开发者所需了解的网络编程基础
- 套接字编程
- RPC 指南
- 游戏编程基础

首先，0.1～ 0.3 节将介绍网络编程。这一部分的要点如图 0.A 所示，请先掌握以下重点，然后我们马上进入正题。

图 0.A 网络游戏的层次体系



0.1　网络游戏开发者所需了解的网络编程基础

首先，我们来学习网络游戏编程所需的网络编程基础知识。

0.1.1　网络编程是必需的

在实现网络游戏的过程中，网络上各个主机之间的数据传输（发送数据、接收数据）极其重要，因此，为了处理通信（也就是与远程终端进行数据的输入输出），必须掌握网络编程。

利用网络游戏的通信中间件（网络中间件）可以在很大程度上隐藏复杂的处理过程，但是如果不理解网络通信的内部机制，就很难有效运用中间件的功能，调试效率也会变低。因此，虽然不用亲自从底层开始编写所有的代码来实现网络通信，但是理解其中的机制还是很有必要的。此外，理解了本书中所介绍的这方面内容，也有助于理解网络通信中的基本技术。

网络游戏与网络游戏相关的网络编程基础知识并不多，本章将重点介绍那些网络游戏所需的网络编程基础知识。理解了这一块内容才能有效地加以运用，所以一起努力吧！

0.1.2 网络编程与互联网编程

为了使运行在两台以上机器的相关进程能够协调工作，必须在它们之间实现一些必要的通信功能，网络编程就是指实现进程间通信所需的编程技术。

网络编程的范畴非常广泛。比如，在使用 USB 接口将外置 HDD（Hard Disk Drive，硬盘驱动器）连接到 PC 的情况下需要网络编程，因为要使 PC 上运行的进程与 HDD 控制器上运行的进程进行通信。再有，SCSI（Small Computer System Interface）、红外线通信，以至空调的遥控器等都离不开网络编程。

除了任天堂 DS 之间使用有线连接的情况之外，网络游戏中的网络编程一般只使用与互联网有关的技术。这种互联网方面的通信编程称为“互联网编程”（Internet Programming），在国外有很多这方面的参考资料。

实现多人网络游戏的前提就是使用互联网，因此本书只讨论以互联网编程为主的网络编程技术。

0.1.3 互联网编程的历史和思想

在互联网通信的事实标准中，IP（Internet Protocol，网际协议）、TCP（Transmission Control Protocol，传输控制协议）、UDP（User Datagram Protocol，用户数据报协议）等网络通信协议自定义以来的三十几年里，基本的通信方式并没有发生什么变化。IP 协议等以安全、舒适地使用互联网服务为目的而产生的网络协议，被技术标准化组织 IETF（Internet Engineering Task Force，互联网工程任务组）作为基础资料收录在 RFC （Request For Comments[1] ）中。RFC 并不具有法律上的强制力，但是遵守这些标准可以带来经济上的利益，所以很多人都以此为准。

> [1] RFC 是一系列以编号排定的文件。文件收集了有关互联网相关信息，以及 UNIX 和互联网社区的软件文件。——译者注。

RFC 以编号排定，比如，TCP 是在 RFC 793 中定义的，DNS（Domain Name System，域名系统）的实现是在 RFC 1123 中，

HTTP（HyperText Transfer Protocol，超文本传输协议）的 HTTP 1.0 则是在 RFC 1945 中定义的。每次版本更新后，RFC 的编号也会随之更新。截至本书撰写时（2010 年 9 月），RFC 的文档总数已经超过了 6000 份 [2] 。

> [2] 顺带一提，RFC 1（http://www.rfc-editor.org/rfc/rfc1.txt ）描述了使用串行电缆将两台主机进行连接的过程，这实在是很简单的一件事情，令人不禁感慨互联网黎明期的艰辛。

IETF 的 RFC 是为了共享通信形式和协议而产生的，它里面并没有定义实际的编程接口，因此要开始网络编程，还必须进一步掌握一些基础知识。

0.1.4　OSI 参考模型——透明地处理标准和硬件的变化

如果每次出现新标准时，比如出现了更高速的 1Tbit 以太网，都要修改程序，那将非常麻烦，而且网络设备类型多种多样，所以由此产生了希望能尽可能不依赖于这些设备，来透明地处理网络通信问题的要求。

在过去，每次为了应对设备更新都不得不修改程序，而随着网络中主机数量的不断增加，为了降低由此产生的成本，20 世纪 80 年代，美国发起了制定标准化模型的运动。

其研究成果就是 OSI 参考模型（OSI Reference Model）。OSI 参考模型的优点为：如果基于该模型进行设计，那么即使标准和硬件发生了变更，位于其上方的层次也不需要做过多的改动。通信系统是一种“设备与设备互连”的系统，所以应该尽可能不要对相连的设备造成影响，而是对其本身进行修改，这才是我们期望的设计方式。

OSI 参考模型提出了 7 层结构，以下介绍各个层次的功能。

- **第 7 层：应用层**

  提供具体的通信服务，比如文件和邮件的传送，访问远程数据库等。这一层的协议包括 HTTP、FTP（File Transfer Protocol，文件传输协议）等。

- 第 6 层：表示层

  规定数据的表现形式，比如将以 EBCDIC[3] 表示的文本文件转换为以 ASCII 码表示的文件

- 第 5 层：会话层

  规定应用程序之间的通信从开始到结束之间的顺序（在连接中断的情况下，尝试恢复连接）

- 第 4 层：传输层

  实行网络中应用程序进程之间端到端的通信管理，如差错恢复、重发控制等

- 第 3 层：网络层

  对网络中的通信链路进行选择（路由选择）、中继

- 第 2 层：数据链路层

  控制直接相连（相邻）的通信设备之间的信号收发

- 第 1 层：物理层

  规定物理连接，包括连接器的引脚数、连接器形状等，以及铜缆与光纤之间电气信号的转换等

> [3] 广义二进制编码的十进制交换码（Extended Binary Coded Decimal Interchange Code）。在主机中使用的字符编码。

OSI 参考模型并不是一种非遵守不可的技术标准，它是一种有助于技术人员理解并记忆的概念性指导方针。对于各类技术人员和企业来说，遵循 OSI 参考模型来设计产品可以获得不少好处，所以这一模型自然而然地得到了普及。

只要在 OSI 参考模型的上层与下层之间提供相同的接口，即使任意一层中的组成要素发生了变更，系统仍能照常运作。

0.1.5 网络游戏系统及其层次结构

现在，请回看一下本章开头介绍的图 0.A。根据 OSI 参考模型的层次结构，对网络游戏的系统进行总结之后，我们划分出了如图 0.A 所示的分层。

第 4 层大多使用 TCP 协议、不需要直接操纵第 3 层以下的分层

在为了让应用程序能够在互联网上进行广泛通信的第 4 层（传输层）协议中，TCP 和 UDP 这两种协议是事实上的标准。如果要求收发信号按顺序、可靠地进行传输，就要使用 TCP 协议；如果对此不作要求，就可以使用 UDP。

另一方面，网络游戏要求只有在必要的情况下才使用 UDP，除此之外一概用 TCP。只要具有了 TCP 和 UDP 的功能，就基本不需要再进行更进一步的细微调整了，因此，没有必要直接操纵第 4 层以下的分层（第 3 层至第 1 层），将它们交给操作系统来处理就可以了。

但是，为了最大限度地提升性能，在第 3 层以下也有几个需要注意的地方，这几点我将在下一节的后半部分予以说明。

第 5 层以上的分层需要在游戏中予以实现

如图 0.A 所示，网络游戏中尚不存在能作为第 5 层、第 6 层事实标准的协议。因此，一般来说，第 5 层以上的功能都需要由网络游戏的开发人员来实现。这是因为根据游戏的类型和策划内容，在要求上有些微妙的差异，因而无法做到统一。但是在使用针对网络游戏的通信中间件的情况下，可以避免从零开始进行开发，从而能够大幅减少开发量。

接下来，我将介绍一下套接字 API 和套接字编程基础，以及使用套接字 API 的针对游戏的通信中间件的实现。

0.1.6 套接字 API 的基础知识

BSD 套接字 API[4] 是为了实现互联网连接而开发的 API，它是在所有操作系统（包括嵌入式系统）上进行网络开发的首选。使用 TCP /

IP（不是 BSD 套接字）开发的 API 不胜枚举，但是如今在广泛用于网络游戏的环境上（包括游戏机）全都可以使用套接字 API。

> [4] Berkley Socket，也叫做 Socket。

标准化的 C、Java 和 Ruby、Perl、C# 等几乎所有的编程语言都能使用这一 API[5] 。套接字 API 在 20 世纪 80 年代开始普及，此后基本没有进行过变更，因此当时开发的程序有很多至今仍在照常运行。

> [5] C 以外的编程语言（Java 和 ActionScript 等）不能直接调用 Socket API，但是可以使用在一定程度上仿效 Socket 行为的 API。

由于篇幅有限，本书对套接字 API 的介绍仅限于最基本的内容，网络上有大量学习资料可供参考，请读者务必进行查阅 [6] 。此外，前文提到过的 IETF 的互联网方法、OSI 参考模型以及套接字 API 等从互联网开创时期开始的历史和通信方面的基础理论，在《UNIX 网络编程》系列图书（该系列是网络编程领域的圣经）中也有介绍。该书中，套接字 API 的示例丰富且全面，其中的内容在所有网络游戏中都有用到。即使说网络游戏开发团队的书架上都有这本书也不为过。

> [6] 网络游戏也是同样，可以参考以下网站：
> http://x68000.q-e-d.net/~68user/net
> http://www.few.vu.nl/~jms/socket-info.html

0.1.7　网络游戏和套接字 API——使用第 4 层的套接字 API

使用套接字 API 可以控制位于第 1 层、第 2 层（物理层和数据链路层）之上的第 3 层的 IP 协议、第 4 层的 UDP、TCP、ICMP（Internet Control Message Protocol，Internet 控制报文协议）等协议。套接字 API 中包括直接控制 IP 协议的第 3 层的 API 和使用 TCP 等协议的第 4 层的 API。网络游戏只使用“第 4 层的套接字 API”。

面向连接（流式）和无连接（数据报式）

数据包会在网络的各条路径中传输，在这个过程中，数据包可能会因为路由器的处理能力不足或者通信链路拥堵等原因而丢失。在这种情

况下，作为互联网基础的第 3 层 IP 协议并不会重发数据包，所以它是不可靠的。此外，数据包的到达顺序也无法保证。但是一旦收取成功，其内容不会发生改变 [7] 。

> [7] 错误的修正由第 2 层以下的层次来保证。

如果使用第 4 层的套接字 API，就可以在不具可靠性的 IP 协议之上实现两种类型的通信：第一种是面向连接的通信（流式，STREAM），在建立了连接的两台主机之间维持通信线路畅通，保证通信持续进行；另一种是无连接的通信（数据报式，DGRAM），只进行一次数据包的交换，不维持各主机之间的通信线路。

在 IP 上层实现的面向连接的协议是 TCP 协议，使用 TCP 的典型代表是 HTTP 和 SSH（Secure Shell）。HTTP 和 SSH 对长时间传输、保证传输顺序一致、可靠性高的数据通信来说，是很有必要的。[8]

> [8] 如果 HTML（HyperText Markup Language）的顺序改变了，标记的前后关系就会完全混乱。

另一方面，无连接的代表协议是 UDP，该协议将 IP 数据包进行分割后发送出去。接收端只具有将其复原的功能，并不能保证数据包的到达顺序和可靠性，这一点与 IP 协议区别不大。使用 UDP 协议的典型代表是 DNS 查询。

＊　＊　＊

之前提到过，网络游戏中都会用到 TCP 和 UDP，但是为了让游戏中的主机能够持续进行通信，基本上都使用面向连接的 TCP 协议 [9] 。在下一节中，我们将介绍一个以 TCP 为前提的套接字编程示例。

> [9] 除了网络地址转换（NAT Traversal，将在 5.6 节介绍）等特殊情况。

专栏　网络编程的特性和游戏架构的关系——服务器、客户端所需具备的性能和功能

那么网络游戏到底需要哪些网络编程技术呢？为了回答这一问题，我们先来简单了解一下网络编程的特性与游戏架构之间的关系。网络编

程的特性根据游戏架构的不同而有所差异，作为铺垫，我们以第 2 章中会讲到的 C/S MMO、C/S MO、P2P MO 这几种游戏架构为例（有关上述分类请参见表 2.4）[10]，看一下它们之间的区别。

> [10] 后面将会详细讲述，C/S 架构游戏是指客户端 / 服务器（Client/Server）模式的游戏，P2P MO 指利用 P2P（Peer to Peer）通信的游戏。MMO（Massively Multiplayer Online）指容纳大量用户的多人网络游戏，而 MO（Multiplayer Online）则是指玩家数相对较少的多人游戏。

- **C/S 架构的游戏**（C/S MMO、C/S MO）

  →高性能、功能强大的服务器端编程 × 一般的客户端编程

  所有的处理都在服务器进行，每台服务器要容纳尽可能多的用户。另一方面，客户端的通信相对比较简单。

- **P2P 架构的游戏**（P2P MO）

  →一般程度的（Web）服务器端编程 × 高性能、功能强大的客户端编程

进行游戏处理的服务器只起辅助作用，由于客户端也要扮演服务器的角色，为此需要在客户端实现支持大量通信量的功能。

总而言之，C/S 架构的游戏要求编程结构满足“服务器端具有高性能”，而 P2P 架构的游戏则要求“客户端具有高性能”。

尽管如此，本书中涉及的一些实时游戏，不管在服务器端还是客户端，都要求高性能、高功能的网络编程。

高性能、高功能服务器的特性 ——网络游戏所需具备的要素

C/S MMO、C/S MO 游戏所要求的高性能、高功能服务器需要具备以下这些特性。

❶ 小带宽

每秒几次至 20 几次，达到几百位通信量的持续连接。

❷ **极高的连接数**

每台服务器需要维持数千至数万个连接。

❸ **低延迟**

处理、结果返回的延迟只能在几毫秒至 20 毫秒以内。

❹ **稳定**

服务器端保持游戏状态（Stateful），敌人等可以移动的物体实时地持续行动。

Web 服务器的特性与此截然不同。所以一般来说，Web 系统中使用的编程技术在其他的网络游戏中是不使用的。本书稍后也会谈及其中的差异。

高性能、高功能客户端的特性 ——网络游戏客户端所需具备的要素

另一方面，P2P 架构的游戏要求高性能、高功能客户端具备以下特性。

❶ **小带宽**

每秒几次至 20 几次，达到几百位通信量的持续连接。

❷ **连接数少**

每个客户端只连接几台机器。

❸ **低延迟**

处理、结果返回的延迟只能在几毫秒至 20 毫秒以内。

❹ **稳定**

服务器端保持游戏状态，敌人等可以移动的物体实时地持续行动，此外，画面渲染等非常重要的处理也要同时进行。

❺ 多样性

必须应对客户端的各种网络状况。

与服务器端相比，客户端的连接数较少，但是在进行渲染等重要处理的同时，必须在延迟很低的情况下进行通信，还要应对网络状况的多样性 [11] ，不管是性能上还是功能上，都需要具备一般的 Web 服务所不具有的要素。

> [11] 包括防火墙、各个 ISP（Internet Service Provider）的策略之间的差异等。

0.2 套接字编程入门——处理多个并发连接、追求性能

本节使用 C 语言风格的伪代码，对基于第 4 层的 TCP 的套接字 API 进行进一步的说明。首先，我们从作为通信起点的“通信链路的确定”（复习）开始，依次介绍“套接字 API 基础”、“处理多个并发连接的策略”（同步连接、异步连接），以及“与性能和开发效率相关的关键问题”。

0.2.1 通信链路的确定（复习）

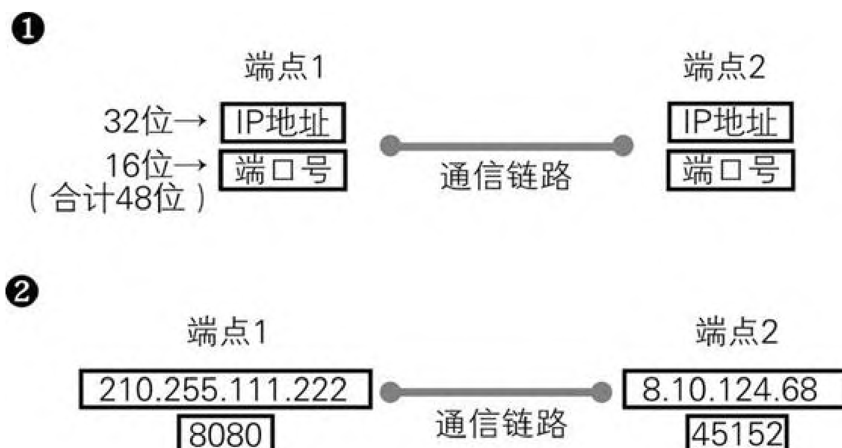
TCP 通信链路（面向连接的通信链路）并不是自然发生的。这是建立在一方提出“想要进行连接”，而一方接受这一连接请求的基础上的。提出“想要进行连接”的这一方称为客户端（client），接受方则称为服务器（server）。服务器在接受请求之前，还需要做一些准备工作。

IP 是由位于通信链路端点的一个 IP 地址（32 位）和一个端口号（16 位）来指定的 [12] 。IP 地址和端口号一共有 48 位，位于通信链路两端的 IP 信息作为一组，根据这总共 96 位的信息，可以指定互联网上任意一条通信链路（参见图 0.1 ❶）。

> [12] 本书以 32 位的 IPv4 为例进行说明。但是现在面临着 IP 地址即将消耗殆尽的问题，为此，采用 IPv6 将 IP 地址扩展至 128 位是非常必要的，IPv6 在技术上已经基本准备妥当了。

IP 地址是一组 32 位的数据，使用十进制值来表示就是 0.0.0.0～255.255.255.255（参见图 0.1 ❷），接入互联网的 Web 服务器需要一个固定的 IP 地址，这是在国际规定下获取的。

图 0.1　根据 IP 地址和端口号所指定的通信链路



另一方面，16 位的端口号可以由服务器的实现者酌情决定。但是有些端口号已经被使用了，比如 HTTP 所使用的 80 端口号这种公认端口号（WELL KNOWN PORT NUMBER，经常被使用的端口号）和注册端口号（REGISTERED PORT NUMBERS，已经登记过的端口号），这些端口号都可以在 IANA （Internet Assigned Numbers Authority） 的端口表中进行确认。此外，在 Linux 和 FreeBSD、Mac OS X 等基于 UNIX 的系统中，在 /etx/services 文件里包含这些端口号的子集。根据这一点，我们可以使用如下命令来指定端口号：

```
shell> telnet localhost http  ←与 telnet localhost 80 相同
```

用这一方式可以实现一些简单命令。

顺带一提，前文所述的 IANA 的端口表还记录了大型多人在线 RPG 游戏《魔兽世界》（*WoW*，后面会讲到）的端口号。

```
blizwow         3724/tcp   World of Warcraft
blizwow         3724/udp   World of Warcraft
```

0.2.2　套接字 API 基础——一个简单的 ECHO 服务器、ECHO 客户端示例

套接字 API 的行为在服务器和客户端是不同的。我们将假想的 ECHO 服务器 [13] 作为一个具体应用来加以说明。

> [13] ECHO 是一个“只要接收到了数据，就原样返回给发送端”的服务器，这是在讲解套接字 API 时广泛使用的一种服务器。

首先来看看代码清单 0.1 所示的 ECHO 服务器端的行为。这段代码很简单，各函数的含义都给出了注释。唯一比较难理解的是代码清单 0.1 中 ❶ 所示的内容，sock 传给 accept 函数以生成一个新的套接字 new_sock[14] 。本来，在代码一开始的地方使用 socket() 函数生成的套接字 sock，在严格意义上并没有建立通信链路，但这个函数却使用了与套接字（socket）相同的名字。为了让代码清单 0.1 中的服务器运行起来，使用 telnet 命令（客户端）以设置的端口号进行连接，同时发送一些合适的数据，这样可以确认数据是否从服务器端返回，请读者予以尝试。

> [14] 新的套接字（也叫连接套接字）new_sock 拥有客户端的连接信息，而原始套接字（也叫监听套接字）sock 只有协议簇等一些基本信息，用于监听。——译者注

代码清单 0.1　ECHO 服务器的行为

```
int sock = socket

(PF_INET, SOCK_STREAM); ←指定类型 17

                                        生成等待使用的套接字（文件描
述符）
bind

(sock, addr);     ←设置监听端口号 18


listen
```

```
(sock);          ←监听开始。待机中
while (1) {
  int new_sock = accept

(sock, &addr);  ← ① 在新的连接请求到来之前一直"等待"(阻塞)
                      当连接请求来到后，返回新的套接字，建立连接 19


  char buf[100];
  size_t size = read

(new_sock, buf, 100);    ←在读满最大的100 字节之前一直等待
                      ( 在数据到达前或是连接中断时阻塞) 20


  if(size == 0){       ←如果read 函数返回了0，意味着接收到了 EOF 21


    close

(new_sock);      ←收到EOF 后关闭连接
  } else {
    write

(new_sock, buf, size);      ←没有收到EOF 的话就写入 size 个字节的数据
  }
}
```

[15] 在这里指定网络接口 / 协议族的类型（本例中设为“PF_INET”，表示 IPv4，写成 AF_INET 在行为上也是相同的。IPv6 要写成 PF_INET6）和套接字类型（本例中设为“SOCK_STREAM”，表示面向连接的流式 TCP/IP。如果设为 SOCK_DGRAM 就表示指定的是 UDP 协议）。

[16] 将地址绑定到套接字中。设置监听用的端口号。

[17] 针对这个套接字进行输入输出，从而与客户端进行消息的收发。

[18] 这里为了方便起见，使用固定的 100 个字节。

[19] EOF 是 End OF File 的缩写，表示数据流的结束（数据不会再发送过来了）。

代码清单 0.2 所示的 ECHO 客户端比服务器端更简单。使用 socket() 函数生成套接字之后，针对指定的地址使用 connect() 函数进行连接，使用 write() 函数写入数据，等收到数据时使用 read() 函数来进行读取。最后的 read() 和 write() 函数会永远循环执行。

代码清单 0.2　ECHO 客户端

```
int sock = socket

(PF_INET, SOCK_STREAM);     ←生成新的套接字（文件描述符）
connect

(sock, addr);       ←使用生成的套接字，
                            向指定的地址和端口（持有这些信息的服务器）进行连
接。
                            在连接终止前等待
while(1) {
  write

(sock, "ping");  ←写入数据（这里是"ping"）
  char buf[100];
  read

(sock, buf, 100);   ←等待读取数据。期待数据（"ping"）的返回
}                                   ←永远往返通信
```

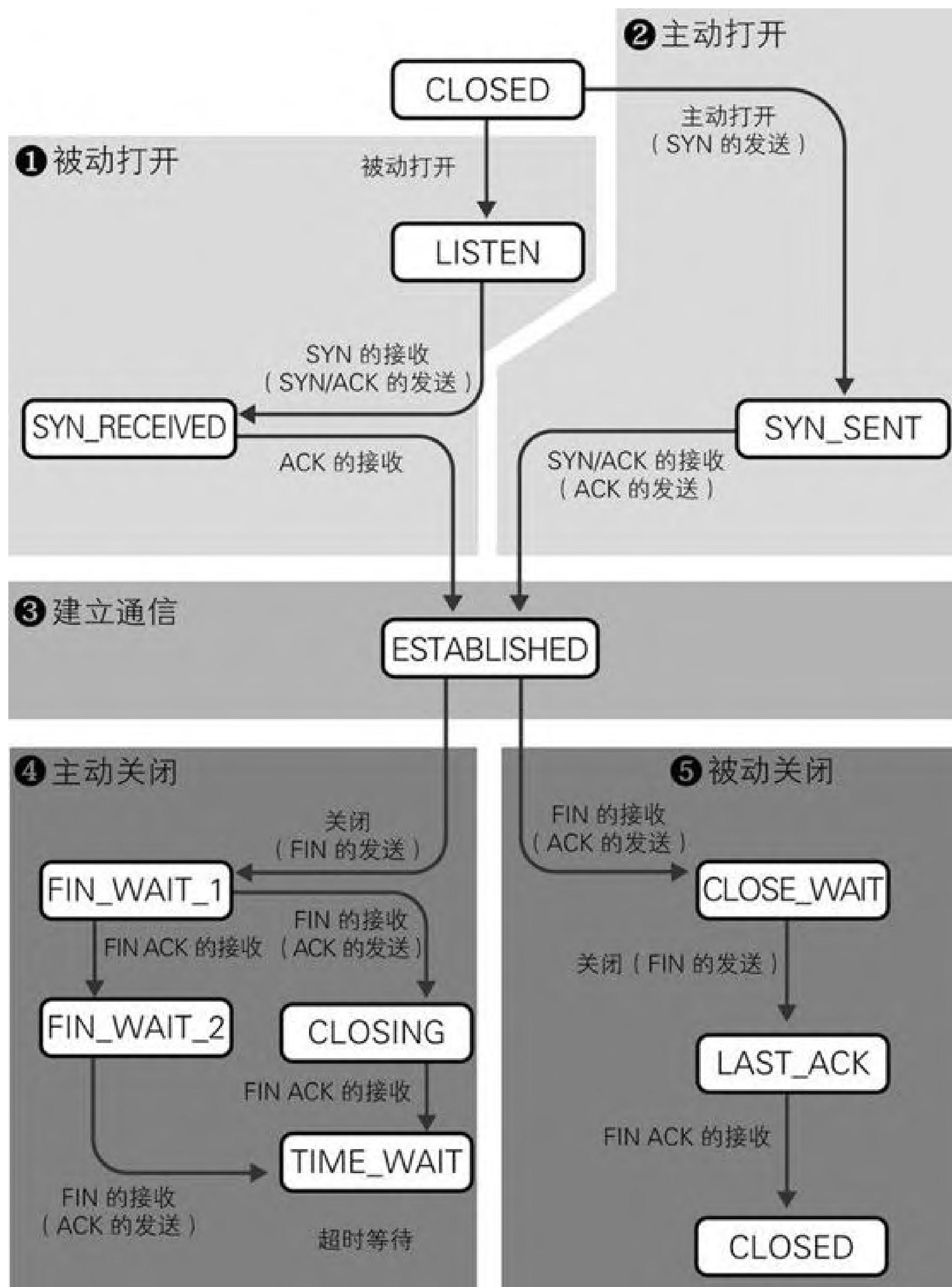
通过上面这个例子可以知道，为了生成作为通信链路的新套接字（确立连接）要用到 socket() 和 accept() 这两个函数。

0.2.3　TCP 通信链路的状态迁移和套接字 API

现在我们来仔细了解一下代码清单 0.1、0.2 与 TCP 通信链路的关系。TCP 的通信链路的状态变化如图 0.2 所示[20]。

[20] 每一条通信链路的状态都可以使用 netstat 命令来确认。

图 0.2　TCP 通信链路的状态迁移



出处：Digital Advantage 刊登的《连载：从基础开始学习 Windows 网络 -2. TCP 的状态迁移图》

http://www.atmarkit.co.jp/fwin2k/network/baswinlan016/baswinlan016_03.html

上述 TCP 状态迁移图中对各个状态进行了分组，非常容易理解。

“需要调用套接字 API 中的哪个函数、将会转变为哪个状态”是理解这一问题的关键。这里不可能对每一点都面面俱到地加以说明，因为这与 TCP 协议对此进行操作的 API 是完全独立的。本书将以套接字 API 为中心来加以解释，请好好掌握以下这些套接字 API 的函数与 TCP 通信链路的状态之间的对应关系。

- socket()

  →因为还不会生成新的 TCP 连接，所以还不存在 TCP 连接状态。

- connect()

  →connect() 函数开始进行如图 0.2 ❷ 所示的“主动打开”（active open）。由客户端调用 connect() 函数主动发起连接称为“主动打开”，而接收到这一请求的服务器被动建立连接则称为“被动打开”（passive open）。客户端在调用 connect() 的瞬间会发送 SYN 消息，此时客户端处于 SYN_SENT 状态，而服务器则处于 SYN_RECEIVED（SYN_RCVD）状态。服务器端的操作系统在收到了这个消息后立刻返回 SYN/ACK 消息，然后客户端在收到这个 SYN/ACK 消息后返回 ACK 消息，此时客户端处于 ESTABLISHED（连接建立）状态，服务器收到 ACK 消息后也将处于 ESTABLISHED 状态。由此可见，客户端会经过 SYN → SYN/ACK → ACK 三次消息收发，因此称为“三次握手”（Three-way handshake）。

- bind()

  →不会生成新的 TCP 连接，只是设置本地生成的套接字的监听端口号，所以没有 TCP 连接状态。

- listen()

  →开始进行如图 0.2 ❶ 所示的“被动打开”。被动打开就是从服务器端角度看到的主动打开，实际上这里的数据包流动顺序与主动打开的顺序是完全一致的。被动打开后，服务器进入 LISTEN（待机）状态。如果服务器处于这一状态，收到客户端发来的 SYN 数据包后就会开始生成新的套接字。

- accept()

  →在操作系统（UNIX 内核）建立了 TCP 连接（处于 ESTABLISHED 状态下的新通信链路）的情况下，我们在应用程序中将其作为新的套接字获取下来。虽然这里 TCP 状态没有变化，但是之后使用 read() 和 write() 函数通过连接套接字来进行数据的收发时需要一个文件描述符，accept() 函数就是获取这个文件描述符的必不可少的函数。

- read()/write()/send()/recv()/sendto()/recvfrom()

  →这些函数用来进行实际的数据收发，必须在处于 ESTABLISHED 状态时调用，否则会报错。

- shutdown()

  →通知操作系统不要再进行数据的写入和读取了。当在参数中指定了 SHUT_RD 停止数据读取时，本地的状态就发生了变化，不再是可读取的状态了，所以会话状态也就不会变化了；而在指定 SHUT_WR 通知操作系统不再向其发送数据（写入数据）的情况下，就会开始关闭套接字 [21]，这样，如图 0.2 ❹ 所示的“主动关闭”流程就开始了。主动关闭与主动连接不同，服务器端和客户端都可以发起。如果再次进入 ESTABLISHED 状态，客户端和服务器的处理都是相同的。

  在主动关闭的过程中，首先从 SHUT_WR 侧发送 FIN 数据包。接收方（被动关闭的一方）会立刻返回 FIN，然后进入 CLOSE_WAIT 状态。SHUT_WR 侧一旦接收到这个 FIN 消息就立刻发送 FIN/ACK 消息，然后释放通信链路。被动侧接收到 FIN/ACK 后也同样关闭通信链路。这个关闭过程也需要三次握手。

- close()

  →等同于调用 shutdown(SHUT_RD | SHUT_WR) 来同时关闭读写双方。

[21] 单单指定 SHUT_RD 或者 SHUT_WR 的话并不会关闭套接字，两个都指定了才会关闭。另外还有一个参数是 SHUT_RDWR，表示先 SHUT_RD 再 SHUT_WR。——译者注

影响 TCP 状态转变的套接字 API 函数就是如上所示的这些了。

0.2.4 处理多个并发连接——通向异步套接字 API 之路

从现在开始就要进入本章的核心内容了。代码清单 0.1 所示的 ECHO 服务器存在一个很大的缺陷。由于 accept() 函数在“新的连接请求到来前一直等待着”，所以 read() 函数在接收新的连接请求前不会再被第 2 次调用。这就导致为了调用 read() 函数，必须每次接收新的连接 [22] 。

> [22] 一般来说，向 ECHO 服务器进行了一次连接后，应该可以多次收发消息。比如，一般的 ECHO 服务器，在 accept 之后，会持续进行 read → write → read → write…的过程。但是在代码清单 0.1 所示的情况中，在 accept 之后，进行 read → write，之后因为已经调用了 accept，在客户端进行连接后，只能收发一次消息。这不能说是一般意义上的 ECHO 服务器。

此外还有一个问题。read() 函数在客户端发来数据之前也会“等待”，所以在开始读取数据前，accept() 函数也不会调用第 2 次。也就是说，在接收了客户端发来的一次新的连接请求后，在数据到达之前无法再接收其他连接请求。

为多个客户端同时提供服务的网络游戏在这种情况下是不可能实现的。为了解决这个问题，必须要处理多个并发连接，为此，需要同时控制多个套接字。方法大致有如下这些。

❶ 每次连接时启动一个进程

❷ 实行异步的多重输入输出（多重 I/O）

❸ 使用线程并行进行同步处理

虽然在 inetd（Internet 超级服务器）和从很早开始就在 Web 中使用的 CGI（Common Gateway Interface，通用网关接口）中都采用了方法 ❶，但是在网络游戏中，需要多个用户（连接）实时共享同一个游戏状态，所以不能使用这种方式。可以在方法 ❷ 和方法 ❸ 中选择。

0.2.5 同步调用（阻塞）和线程

套接字 API 的 connect()、accept()、read() 函数在处理成功之前会一直处于等待状态，而其他函数则不会等待，而是立刻返回（在几微秒时间内）。通常，像这种同步调用 connect()、accept() 和 read() 这类“永远等待着”的函数称为“阻塞”（Blocking）。

处理这种情况的方法一般是使用线程（Thread）。使用线程的示例如代码清单 0.3 所示。它与代码清单 0.1 的不同之处在于，read() 和 write() 函数的反复调用是在 create_thread 中并行执行的 [23] 。

> [23] C 语言与 Ruby 等有所不同，不能像代码清单 0.3 中那样以函数参数的方式来编写，这里所示的代码只是一种直观表示。请考虑将给出的代码段放在其他的线程中执行。在实际情况下，在 UNIX 内核的操作系统上，使用 fork 函数可以启动一个进程，也可以使用 pthread_create 等线程库，总之有很多种选择。

代码清单 0.3 ECHO 服务器（线程版，阻塞）

```
int sock = socket(PF_INET, SOCK_STREAM);
bind(sock, addr);
listen(sock);
while(1) {
  int new_sock = accept(sock, &addr);
  create_thread( {

        ← ① 启动并行处理
    char buf[100];


    size_t size = read(new_sock, buf, 100);


    if (size == 0) {


      close(new_sock);


    } else {


      write(new_sock, buf, size);


    }
```

```
  } );

}
```

根据代码清单 0.3 中的多线程，我们可以实现向多个客户端同时提供服务（这里是返回数据）。使用线程可以同时处理多个“等待”场景。从结构上来看，同步调用是在多个线程的内部并行执行的。

线程方式下的负载处理问题

上述的线程方式在网络游戏的服务器中存在“负载处理”的问题。在每次创建线程时都会启动 1 个线程和进程，如果同时连接数为 3000，就会同时启动 3000 个线程，对现在的机器来说，3000 个并行处理数实在太过庞大，服务器的性能会大幅下降。

活跃进程一般要控制在操作系统能够同时执行的进程数或线程数的 4 到 10 倍以内。如果超出了这一范围，操作系统内部进行线程切换的开销就会变得很大。比如，4 核处理器下的最佳线程数是十几个。3000 个线程实在太多了，但是考虑到服务器成本，每台机器又不得不处理 3000 个左右的并发连接。

0.2.6 单线程、非阻塞、事件驱动——使用 select 函数进行轮询

在实际调用 read 函数和 accept 函数之前，我们可以使用 select 函数 [24] 事先查询一下这些函数所等待的消息（数据以及连接请求）是否已经到达了。这种事先询问的方式称为轮询（Polling）。根据操作系统的版本，使用 poll 函数及更高速的 epoll 函数等多种接口都能实现同样的功能。

> [24] select 是等待输入输出完成（在读写完成之前等待着）的函数。由于是异步处理，可以实现输入输出的多重化。在前述的《UNIX 网络编程》一书中有详细介绍。

代码清单 0.4 展示了使用 select 函数的服务器端代码。在实际的服务器上运行的代码需要进一步设置标志和定义结构体，等等，但是基本的逻辑结构是相同的。

代码清单 0.4 在调用 accept() 和 read() 之前，为了确定该套接字所需处理的事件（数据）是否到达而调用了 select() 函数。这样，read() 和 accept() 函数就可以在几微秒内返回并且获取到数据。实际上，select() 函数不会像这样调用多次，调用一次就能一下子确认几千个套接字。

代码清单 0.4 中的代码与代码清单 0.3 的差别就是没有创建线程（单线程）就向多个客户端多重化地提供了服务。这种实现方式称为异步调用、非阻塞（Non-blocking）方式。另外，因为这种方式会事先查询数据到达这一事件是否发生，然后再调用相关函数，所以也叫做事件驱动（Event-driven）。

代码清单 0.4 ECHO 服务器（select 版本，非阻塞、事件驱动）

```
int sock = socket(PF_INET, SOCK_STREAM);
bind(sock, addr);
listen(sock);
allsock.add(sock);

 ←向allsock 队列注册sock
while (1) {
  result = select(sock);

  ←插入select 函数进行事先检测
  if(result > 0) {
    int new_sock = accept(sock, &addr);
    allsock.add(new_sock);

 ←向allsock 注册新的连接
  }
  foreach(sock = allsock) {
    result = select(sock);

 ←插入select 函数进行事先检测
    if(result > 0) {
      char buf[100];
      size_t size = read(new_sock, buf, 100);
      if(size == 0) {
        close(new_sock);
      } else {
        write(new_sock, buf, size);
      }
    }
  }
}
```

0.2.7　网络游戏输入输出的特点——单线程、事件驱动、非阻塞

在游戏编程中，同时处理数千个可移动物体是很平常的，这与“使用 1 个线程处理数千个套接字”类似。为此，在网络游戏中，客户端和服务器端通常都使用 select 函数（或者 poll/epoll 函数）在单线程中实现非常简单的事件驱动的非阻塞方式。

0.2.8　网络游戏和实现语言

网络游戏中最常用的编程语言是 C/C++，其次是 Java，此外也有一些其他语言。轻量级语言可以嵌入服务器的实现代码中与其他语言并存。轻量级语言以 Lua 和 Squirrel 为代表，最近 Ruby 和 JavaScript 也在崛起。由于篇幅有限，本书无法对此进行详细介绍，但是嵌入式语言 Lua 和 Squirrel 所用的内存很少，初期成本很低，所以在游戏开发中很受欢迎。

服务器端使用的语言以 C、C++、Java、Ruby、Python、Perl 为代表。在这些语言中都可以使用 select 函数来实现事件驱动、非阻塞方式的套接字。

客户端所使用的代表语言有 C、C++、Java、Objective-C、Flash/ActionScript3、JavaScript。在 Flash/ActionScript3 和 JavaScript 中使用事件驱动的套接字是基础，它们与 RPC（Remote Procedure Call，远程过程调用，后面将会介绍）服务也能很好地协作。

0.2.9　充分发挥性能和提高开发效率——从实现语言到底层结构

现在，我们来看一下如何充分发挥性能，提高开发效率。首先从实现语言开始，然后再看看第 3 层以下的底层部分的注意点。

如今的网络游戏大多使用 C/C++，而现状是大家迫切需要缩短游戏开发周期，由于开发成本很高，各公司都在讨论是否不能使用具有垃圾

回收特性的 Java 和 C#，或者轻量级语言。笔者自己也在想如果能用 Ruby 语言来编写具有 C++ 执行速度的服务器就好了。

就现状来看，如果牺牲服务器的最高性能，就能提高开发效率。这是种此消彼长的关系。表 0.1 以一些主要的编程语言为例进行了简单的总结。

从表 0.1 可以看到，C/C++ 与 Ruby 在吞吐量性能上相差了 1000 倍。服务器数量与处理性能是成反比的，所以 1000 倍的差异所带来的服务器数量的差异也是非常惊人的。

表 0.1　主要的编程语言与吞吐量

语言	吞吐量	特性
C/C++	100	静态语言、本地代码
Java/C#	1～10	静态语言、VM、字节码
Ruby/Python	0.1～1	动态语言

现在，拥有大量用户的游戏都是用 C/C++ 来编写的，内容相似的游戏纷纷涌入市场，如果不使用 C/C++ 的话就无法在价格战中获胜。

网络游戏的特殊性所造成的编程语言性能差异

表 0.1 中，Java 的吞吐量比 C/C++ 低了 10 倍，这是由于网络游戏的特殊性所造成的。一般在配备了 JIT（Just In Time）编译器 [25] 的虚拟机（Virtual Machine，VM）[26] 中，Java 的运行速度会因 JIT 编译的效果变得很快，某些情况甚至会比 C 语言更快。

[25] 在 Java 编程语言和环境中，即时编译器是一个把 Java 的字节码（包括需要被解释的指令的程序）转换成可以直接发送给处理器的指令的程序。——译者注

[26] 可以说这是全部的了。

但是这种效果只发生在以 CPU 为中心[27]的应用程序中，而在那些与操作系统频繁进行输入输出操作的应用程序中无效。比如，在一个对 100MB 的文件进行读取，每次读取 1KB 并对行数进行计数的程序中，C 语言要比 Java 快上 10 倍左右的情况也是常有的。这是因为 Java VM 在系统调用前后，每次都会进行缓存溢出和异常对象的处理。这是无法省去的处理过程，所以使用 VM 的处理系统存在一定的局限性。网络游戏的服务器每秒会进行数万次输入输出，这是 Java 和 C 语言产生速度差异的典型例子。Apache 和 MySQL 等服务器软件都用 C/C++ 编写也是基于同样的原因。

> [27] 指主要使用 CPU 来进行处理。另外以 I/O 为中心是指输入输出占了较大比重的处理方式。

其次，动态语言的吞吐量比起 Java 更是低了 10～100 倍，为什么会这样呢？这是因为每次进行一些处理时，对象调用的方法可能会发生变化，所以每次都必须进行检查确认。

顺带提一下，Google 的 Go 语言是一种静态的、本地执行的语言，它具有垃圾回收机制，程序员可以在代码的不同部分中选择类型化的强度，既不牺牲服务器的性能又可以提高开发效率，笔者对此十分期待。让人不禁感叹 Google 对服务器开发确实颇为了解。

0.2.10　发挥多核服务器的性能

通过单线程、事件驱动和非阻塞的实现，就可以充分发挥出多核服务器的性能。

举例来讲，在 CPU 只有一个处理核心的情况下，不可能同时执行多个线程或进程，而是为每个线程或者进程划分一小段时间片，轮流执行。例如，如下所示暂停 1 毫秒的程序：

```
while(1) {
  usleep(1000);
}
```

如果有两个这样的程序同时运行，那么 1 秒内要切换 1000 次。由于要从进程 A 切换到进程 B，又要从进程 B 切换到进程 A，每秒总共要切换 2000 次。这里的切换次数可以在 Linux 上用 vmstat 命令来确认[28]。

> [28] vmstat 的相关内容会在第 7 章进行说明。

专栏　输入输出的实现方针和未来提高性能的可能性

网络编程中输入输出的实现是一个非常深奥的问题，为了充分发挥设备的性能，开发人员提出了各种各样的方案。

比如，如果存储多个线程与事件驱动一起使用，是否会提高性能？围绕这一问题展开了诸多争论[29]。将来 Apple 的 Grand Central Dispatch[30] 这种并行执行机制、Google 的 Go 语言的 goroutine 特性等是否能减少工作量并且带来高性能，笔者对此也充满期待。

> [29] 可以参考以下网站：http://d.hatena.ne.jp/sdyuki/20090624/1245845216

> [30] Mac OS X 10.6（Snow Leopard）搭载的新功能之一，简化多核并行编程的一种机制。

上下文切换 ——保存 CPU 的设置状态

上文提到的“切换”称为“上下文切换”（Context Switch），在进行上下文切换时，CPU 核心内部将执行注册（Register）、保存虚拟内存（Virtual Storage）的管理表以及安全性设置的切换等。至笔者撰稿时，在 Linux 上，平均每个 CPU 内核一秒内可以执行 10 万～20 万次上下文切换。这也与 CPU 的高速缓存内容等应用程序的执行情况有关，这一点需要注意，但是，比如前文中提到过的那个暂停 1 毫秒的程序，如果运行了 200 个左右这样的程序，仅仅是上下文切换大约就会使当前时间 CPU 的系统占用率（由 vmstat 命令获得当前时间的 CPU 统计信息）达到 100%。

上下文切换要处理 CPU 设置状态的保存工作，CPU 内核多的话，整个系统每秒能执行的次数也会增加。比如，在拥有 10 个 CPU 内核的机器上执行同样的任务，基本上可以进行 100 万～200 万次上下文切换。[31]

> [31] 笔者没有接触过拥有 100 个内核、1000 个内核的这种庞大内核数的机器，也许这种机器的行为会有所不同。

因为一个内核的处理能力已经达到极限，而内核的处理也已达到了最优化，所以在今后的机器中，一个内核所能进行的上下文切换次数与上面给出的值相比，很有可能不会发生很大的变化了。

多核处理器上不要运行过多的服务器进程

在网络游戏的服务器上实现单线程、事件驱动、非阻塞的情况下，如果服务器的每个内核运行 1 个进程，就能充分利用多核处理器的性能。

上下文切换在针对网络的输入输出中也是必须的。假设 1 个内核可以执行 10 万次上下文切换，那就可以进行 10 万次网络输入输出。平均每个内核能有 1000 个同时连接数的话，每秒就可以进行 100 次输入输出，通常，每秒的输入输出只有几次的程度，可以说非常宽裕。因此，多核服务器中，服务器进程数只要不增长过多的话就不会有问题。

0.2.11　多核处理器与网络吞吐量——网络游戏与小数据包

服务器通常使用以太网 [或许是千兆以太网（1 Gbit Ethernet）] 连接至数据中心内的网络中。现在在 Linux 中，以太网在基础设施层次中可达到其名所示的速度。也就是说，千兆以太网的通信速度就是 1Gbit。交换集线器也可以应对这一速度。

但是，网络游戏中发送大量小数据包的情况下，有时也会无法达到预期的通信速度。

以太网帧

首先，以太网在发送 IP 数据包时，会向数据包中添加 IP 数据以外的信息一起发送。包括这些附加信息在内的总带宽是 1Gbit/s，实际上应用程序能够使用的带宽比这要小。

比如，在 Windows 中使用 telnet 命令连接 TCP 服务器，按下 A 键就会发送 ″a″ 这个 1 字节的数据。此时，以太网实际消耗的带宽是多少呢？在最典型的以太网中，发送数据包时会发送图 0.3 所示的信息。

在图 0.3 中，❶ 中的（7）表示 7 个 octet。octet 的意思是“8 位位组”。因为有一小部分机器“一个字节（byte）不等于 8 位（bit）”。这样就不能用 bit 来进行划分，所以在严格定义通信方法的情况下都使用 octet 这个单位。

图 0.3　最典型的以太网帧（IEEE 802.3）

❶	❷	❸	❹	❺	❻	❼
前导码信号 (7)	SFD (1)	目的地址 (6)	源地址 (6)	长度/类型 (2)	数据/LLC (46 ~ 1500)	FCS (4)

″a″ 这样的一个字符可以用 1octet 来表示，用 16 进制来表示就是 0x61，2 进制表示就是 8 位的 01100001。

图 0.3 中 ❶❷ 所 示的开头 7 octet 的“前导码信号”（Preamble，同步信号）和其后 1 octet 的 SFD（帧首定界符，Start Frame Delimiter）总是固定的（最后的 1 连续重复了 2 次）：

```
1010101010101010101010101010101010101010101010101010101010101011
```

这是一种为了从流入电缆的噪声中找出通信信号的固定信号。这会持续长达 64bit 的时间（虽说这是微秒以下的单位），所以以太网的转换器可以由此分辨出噪声和信号。

在图 0.3 中 ❸❹ 中，接收方和发送方的 MAC 地址各有 6 octet，总共 12 octet。MAC 地址是分配给与以太网连接在一起的所有设备的一个数值，由每个设备厂商进行分配。比如在 Windows 中，使用 ipconfig/all 命令[32] 可以看到如 04-A3-43-5F-43-23 这样的表示。

[32] 在 Linux 中，可以使用 ifconfig 命令。

接下来，❺ 用 2 octet 来指定数据的长度和类型，❻ 是数据本身，最后 ❼ 是 4 octet 的 FCS（Frame Check Sequence，帧检验序列，用于修正信号错误的总和检验码）。

各个网络层的头信息

使用 TCP 协议发送数据时，也会同时发送以太网帧头信息以外的一些头信息。在使用 TCP 发送 "a" 这样 1 octer 数据时，在 OSI 参考模型的层次结构中会用到以下 4 个下层系统。

- **第 4 层（传输层）：TCP**
- **第 3 层（网络层）：IP**
- **第 2 层（数据链路层）：以太网协议**
- **第 1 层（物理层）：双绞线电缆**

因为采用了层次结构这种方式而不是直接利用以太网，即使更上层的系统要使用不同的物理媒介（比如 Wi-Fi 和 3G 网络 [33] 等）来进行通信，也不需要修改程序。为了享受到这一优势，必须要向各个层添加必要的头信息。

[33] 3rd Feneration Network。第 3 代移动通信网络。

具体来说就是加上第 4 层 TCP 的头信息（20 octet）、第 3 层 IP 的头信息（20 octet）和以太网的帧头信息和帧尾信息（22 octet）。

TCP 头如图 0.4 所示，必须包括源端口号（16 位，2 octet）、目的端口号（16 位，2 octet）、序列号（32 位，4 octet）、ACK 序列号（32 位，4 octet）、标志（代码位）、窗口大小、（数据的）校验位、紧急标志等 20 octet。

图 0.4　TCP 头

IP 头如图 0.5 所示。图 0.5 的前 3 行（×4 = 12 octet）包括版本号（4 位）、头部长度（4 位）、服务类型（8 位）、数据包长度（16 位）等各种通信设置。紧随其后的是 32 位的源 IP 地址和目的 IP 地址。至此为止的 5 行 20 octet 是 IP 报文必须包含的内容。图中 IP 报文的“数据”部分放入了 TCP 头所包含的数据，而 TCP 的“数据”部分则放入了 ″a″ 这个应用想要发送的数据。

图 0.5　IP 头（IPv4）

由此可见，为了发送 "a" 这个 1 octet 的数据，需要用掉如下这些 octet。

- **TCP：总共 21 octet：**
  - 头：20 octet
  - 数据 "a"：1 octet
- **IP：总共 41 octet**
  - 头：20 octet
  - 数据（TCP 的部分）：21 octet
- **Ethernet：总共 67 octet**
  - 前同步信号：7 octet
  - SFD：1 octet
  - 接受方：6 octet

- 发送方：6 octet
- 长度：2 octet
- 数据：41 octet（IP 的部分）
- FCS：4 octet

可以看到，传输 "a" 总共需要耗费 67 octet。假设使用 1Gbit/s 以太网，应用程序每次发送 1 字节的数据，实际可能用到的带宽为 1Gbit/s 的 1/67，大约是 1.5Mbit/s。

多核处理器的数据传输能力

如上所述，使用 OSI 参考模型时，如果将数据分成非常小的数据块来发送，头信息就会占据很大一部分，对物理层来说负担非常大。相反，如果以 1400 字节的大数据块为单位来发送，各个头信息所占的比重就会降低，这样基本上可以达到理论上的通信速度。

网络游戏中，应用程序的数据单位有时不得不只有 20 个字节左右，所以很多情况下只能达到以太网理论速度的几分之一。一般来说，使用 10Mbit/s 以太网发送最小的数据时可以达到每秒 14881 次，100Mbit/s 以太网的话可以达到 148810 次，而 1Gbit/s 以太网则可以达到 1488100 次。

根据经验，将理论值的 1/10 作为基准，1Gibt/s 以太网每秒可以发送 100MB 的数据，能够发送的数据包数最好以每秒 10 万～15 万为上限（Linux 的情况下）。

如果在有 10 个内核的机器上使用 1Gbit/s 以太网，每个内核可以处理大约 1 万个数据包，如果同时连接数为每个内核 1000 个连接的话，或许每个连接必须设计为将发送频率限制在每秒 10 次以内。或者，如果服务器可以安装多个网络适配器（NIC，Network Interface Card），那么可以连接 4 根 LAN（Local Area Network）电缆，以实现 4 倍的吞吐量。

此外，10Gbit/s 以太网的交换集线器价格仍然很高，在必须降低成本的网络游戏中还不能使用。如果将来价格便宜的话，就可以选择它

了。

0.2.12 简化服务器实现——libevent

最后，我们来了解一下服务器实现的简化。在“单线程 + 事件驱动 + 非阻塞调用” 模式下，实现服务器的最佳程序库是 libevent 顺带一提，有个可以替代 libevent 的更为高速的程序库，名为 libev：http://software.schmorp.de/pkg/libev.html]}。libevent 是从文件共享软件 Tor 派生出来的库，在 memcached[34] 等系统中也有使用，在追求与网络游戏的服务器和客户端系统同等服务性能的网络服务器软件中，它被持续使用长达 5 年以上。

> [34] memcached（http://memcached.org ）常用于 Web 服务中，在 mixi 和 livedoor 等所有大规模服务中，为了提高服务器的处理性能而使用 memcached。2010 年夏天在 mixi 中使用的 memcached 虽然情况不佳，但是 mixi Engineers'Blog 立刻进行了公开、详细的解释。就是在特定硬件设置的状态下只有在发生大量连接时才会产生 bug。http://alpha.mixi.co.jp/blog/?cat=34

libevent 在全世界的网站中都有运用，不管是性能方面还是稳定性方面都很成熟。尽管如此，libevent 在实际用于商业服务时，在嵌入到游戏服务中后，应该进行并入单独的游戏内容中的负载测试。

libevent 的特点

使用 libevent 时需要注意以下关键点。

- **如果套接字处于某个指定状态时（可以 write、可以 read、可以 accept），调用事先指定的函数。**
- **libevent 库会自动选择各个 OS 中最高效的方法（比如，在 Linux 中当套接字数量很多时，选择 epoll 函数等）来轮询套接字的当前状态。**
- **应用程序用事先设置的函数调用（称为回调函数）来获取这一结果，在这回调函数中实际执行 read、accept 等本来应该在等待的函数。**

libevent 最大的特点就是：它的工作方式并非创建大量线程然后等待 read 等系统调用，而是“不创建线程，为每一个想要事先通知的事件

注册回调函数，当事件发生时，只进行一次函数调用”。

因为完全不创建线程，可以执行非常轻量且高速的行为，所以它面向的是实现以通信处理等数据的输入输出为主的并行处理。

libevent 非常简单且高速，所以在很多系统中都有使用。笔者也在实现网络游戏的服务器端时实际使用过，获得了相当出色的性能。在 Linux、Mac OS X、Solaris、Windows 等主要的操作系统都能使用，它的跨平台特性也非常出色。

使用 libevent 就可以简化采用单线程＋事件驱动＋非阻塞调用这种模式的系统的开发工作。

0.3 RPC 指南——最简单的通信中间件

本节将要介绍远程过程调用协议 RPC。RPC 将与通信有关的一些复杂细节封装起来，与一般的函数调用形式相同，它是确保与远程主机进行简单、安全的通信的一种方法。使用 RPC 就不用直接使用 BSD 套接字的 API 也可以进行通信程序的开发了。

0.3.1 通信库的必要性

上一节介绍了第 4 层（TCP）的套接字编程的基础知识。但是在实际的游戏中，应该避免直接调用（BSD）套接字 API 的函数，而使用更上层 API 封装了的通信库来开发应用。这是因为 BSD 套接字库会根据网络状况产生如下问题。

- 不一定能成功收发期望数据，所以之后需要再次调用。
- 可能会发生错误。
- 发送缓存满了的话，write() 函数会等待 [35] 。

> [35] 后面会讲到，在库的内部缓存，在可以写入数据的阶段再次发送，这样可以解决这个问题。

与向文件写数据相比，向网络写数据可能会碰到目标主机离线的情况，无法获知准确的状态，即使当前发送失败，之后也可能会成功，所以不能仅仅作为出错来处理。网络的基本性质就是“状态是不定的”。如果是文件，就算磁盘满了立刻写入失败也没什么关系。

现在来看看一段不甚理想的客户端代码（代码清单 0.5）。send 根据网络状况可能会产生如下问题。

❶ **可能等待。**

❷ **可能发送失败。**

❸ **在希望发送 4 个字节时，可能只发送了 1 个字节。**

在连续敲击键盘的情况下，可能会有 3 次没有发送。

代码清单 0.5　不甚理想的客户端代码示例

```
void keyDownHandler(KeyboardEvent e) {        ←客户端按下某个键盘按键
  send(sock, e.keycode);
}
void mouseDownHandler(MouseEvent e) {         ←客户端按下某个鼠标按键
  send(sock, e.buttonCode);
}
```

但是这些函数中的 send 在发送成功前不会阻塞，每次编写错误处理造成的代码重复也是引起很多错误的根源。

通常，需要有一个能独自负责这些工作的程序库。这个通信程序库应该首先将针对网络的输入输出要求装入缓存中，接着准确地加以执行。然后再准确地将数据发送出去，直到发送完成。如果在一段时间内无法发送出去则返回错误消息，像这样，用“时间”这个要素来区分成功和失败。

代码清单 0.6 中用调用 wrapperSend 函数来替换了 send() 函数。wrapperSend 函数不会阻塞，除了内存不足不能运行之外不会返回错

误，这样来确保成功。在通信库中有一个有名的库——boost::asio，能与这里实现同等内容。

代码清单 0.6　使用 wrapperSend 函数

```
void keyDownHandler(KeyboardEvent e) {        ←客户端按下某个键盘按键
  wrapperSend

(sock, e.keycode);      ← ①
}
void mouseDownHandler(MouseEvent e) {         ←客户端按下某个鼠标按键
  wrapperSend

 (sock, e.buttonCode);
}
```

确定数据格式来进行数据的收发

在这里，我们不对上述代码清单 0.6 作详细分析，但会仔细了解一下实际的数据发送方式。在其中的①处，将按下的按键以调用 wrapperSend(sock, e.keycode) 函数的方式发送出去，这些代码是很抽象的，那么具体发送的是什么样的数据位呢？

比如按下了 x 键，该键码值用十六进制来表示就是 0x78。TCP 是面向连接的流式协议，发送时不会间断。因此，①处代码会发送 0x78 这样一个字节，但是这里并不知道这个字节是否与前面发送的数据是连在一起的。

再比如，想要在鼠标移动时发送对应的坐标，这时调用 wrapperSend(sock, e.mouseX)函数来发送。如果 *x* 坐标用十六进制表示也是 0x78 的话，接收方就无法判断到底是按下了键盘按键还是移动了鼠标。

因此，很有必要指定如下这样的数据格式，然后再进行数据的收发。

[数据的类型代码 1 字节] [数据内容]

0.3.2　网络游戏中使用的 RPC 的整体结构

上述的数据格式经过一番发展之后，产生了一个称为 RPC 的概念。这是在本地模拟远程主机（其他的进程）中的函数调用，将数据流进行编码后发送出去，远程主机接收这些数据并将其解码，然后调用相应的函数。除了指针以外的数据都能顺利发送。

网络游戏中使用的 RPC 模式如图 0.6 所示。最下面的是物理层，向上依次对应七层模型。

图 0.6 左侧是调用 RPC 的一方，右侧则是接收方。不管是哪一侧都可以是客户端或者服务器端。

首先，图 0.6 ❶ 处调用侧的应用程序在程序内部调用了名为 attackAtEnemy 的函数。该函数的定义在源文件“RPC 存根代码”中。因为逐个手工编写函数非常麻烦，所以这个 RPC 存根代码是用工具自动生成的。在这里就是用套接字库调用 send() 函数发送两个字节。第 1 个字节表示想调用的函数是“attackAtEnemy 函数”，所以写入 123 这个固定值。下一个字节表示攻击对象的 ID 值，在这个例子中写入的是 99。这是通过 attackAtEnemy 函数的参数传入的。

虽然图中省略了错误处理，但其实也是同时进行的。

图 0.6　网络游戏使用的 RPC 模式图

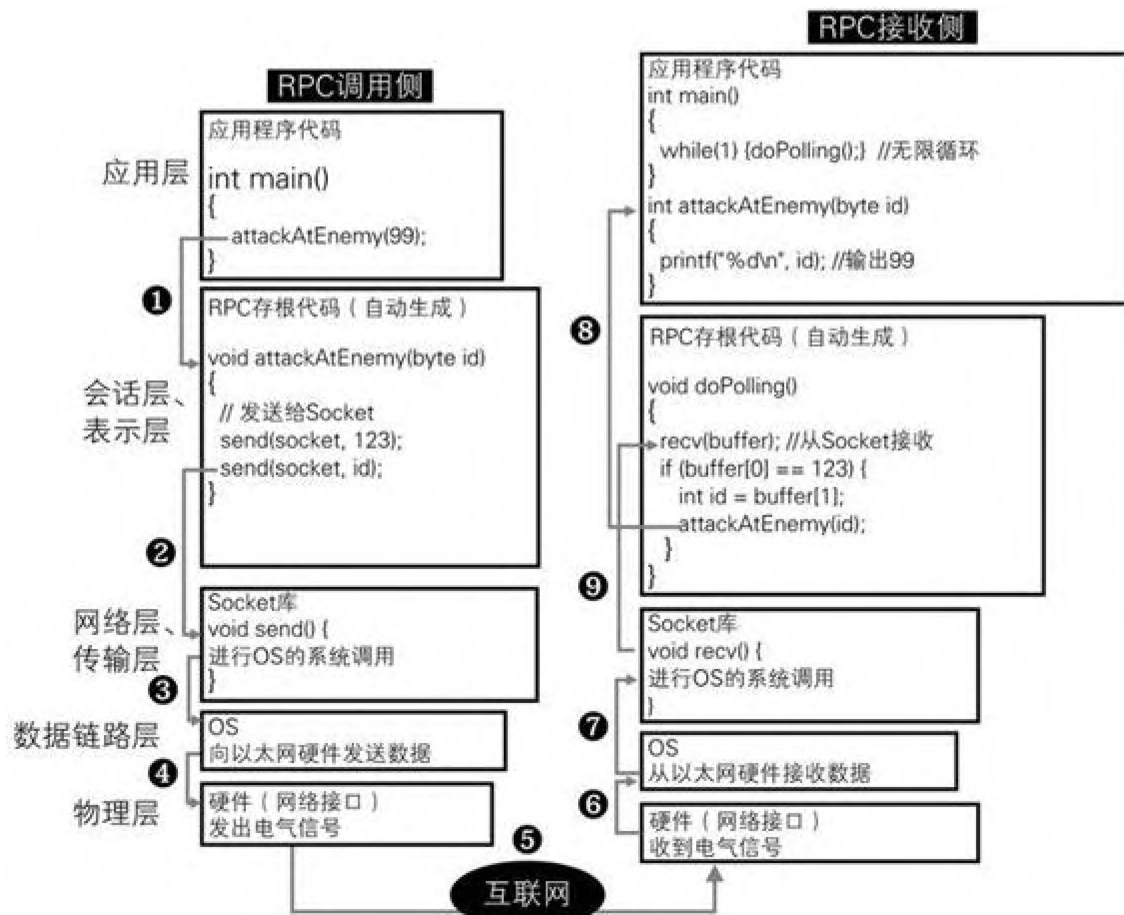


图 0.6 ❷ 处，在之前调用的套接字库的 send 函数内部进行了 OS 的系统调用（图 0.6 ❸），OS 让硬件开始运作，而硬件则发出电气信号（图 0.6 ❹），随后电气信号就在互联网中传输（图 0.6 ❺）。

在接收侧则将收到的电气信号从硬件传送给 OS（图 0.6 ❻），将其处理成程序可以使用的状态。OS 再将这些数据发送给套接字库（图 0.6 ❼）。

接收侧的应用程序在主循环内执行 doPolling 函数（图 0.6 ❽），在里面接收从套接字发来的数据。

由于接收到的第 1 个字节是表示函数类型的固定值 123，所以在条件分支中调用实际在接收侧应用程序中定义的 attackAtEnemy 函数。然

后从第 2 个字节中取出在调用侧作为参数传入的值 99，将其赋给 attackAtEnemy 函数的参数。

* * *

按照以上的方式来处理时，就不用关心与通信相关的任何细节内容，通过使用函数调用的方式，就能安全地编写通信程序。在有 2 个进程执行复杂通信的应用程序中，使用作为结构方法的 RPC 也是很普遍的。

自动生成 RPC 存根代码的 RPC 工具

RPC 存根代码文件中调用方的函数参数列表必须和被调用方的函数参数列表完全一致，但是如果有大量函数，全部手工编写这些函数的定义难保不会出错，还有可能产生极难修复的 bug。一般我们会使用一些用于省去手工编写函数定义这项工作的专用程序，来自动生成调用方和接收方的函数定义。在这里，我们将这种专用程序称为 RPC 工具。

通常，使用 Ruby 和 Python 等很容易进行 DSL（Domain Specific Language，领域特定语言）定义的语言来设计 IDL（Interface Description Language，接口描述语言），然后执行脚本，通过这种方式来自动生成发送侧函数和接收侧函数的存根函数的源代码和头文件，然后在项目中进行链接后加以使用。

为了让两个程序可以进行“对话”，必须定义一些必要的接口，这种定义接口的语言就称为 IDL。当两个以上的程序需要进行“对话”而又不能使用常规的函数调用时，IDL 是十分必要的。比如，这些程序各自使用不同的语言来编写，或者运行在不同的机器上，又或者在不同的时间内运行，等等。因为在这些特殊的情况下，必须具有因机器不同而不同的二进制格式的转换和持久化等功能。

IDL 一般与想要“对话”的应用程序的编程语言相同，或者使用可读性很高的第三方编程语言，或者配置文件等来描述。现在，如果想自己编写 IDL，可以利用 Ruby 的 ERB 和 Python 的 Django 等非常优秀的常用于 Web 的模板引擎，使用这些引擎可以很方便地生成源代码。

使用 RPC 工具自动生成源代码的话，那些与各类开发工作相关的费时任务都可以在 RPC 工具中自动完成，包括：直接发送整数、字符串、数组、列表、结构体和类等；在发送 2 个字节以上可能随机器的不同而不同的数据时，进行字节顺序的处理；针对两种以上的编程语言输出源代码，等等。

但是拥有复杂的功能后处理速度就会变慢，无法在游戏中使用。比如，以名为 XMLRPC 的 XML（Extensible Markup Language，可扩展标记语言）为基础的 RPC 结构，因为使用的是收发 XML 的形式，所以具有良好的可读性和调试效率，但是在网络游戏中，一个内核每秒要处理数万个 RPC，实在太过繁重。

网络游戏和二进制数据交换格式 / 库

在网络游戏中，不需要以二进制格式发送具有非常复杂的层次结构的数据，简单的数据结构就已经足够了。Google 公司由于其内部的需要，将名为 Protocol Buffers 的数据交换格式作为开源项目对外公布。Facebook 公司也同样发布了名为 Thrift 的库，现在该项目已经入驻 Apache Software Foundation （Apache 软件基金会）。笔者虽然没有用过日本开发的 MessagePack 程序库，但是知道它的处理速度非常快。

Protocol Buffers、Thrift、MessagePack 都能针对结构体和枚举进行处理。在定义文件与输出代码的对应关系时，请参考各自的说明文档。这些来自 Web 企业的开发工具都用 Python 等轻量级语言来实现，也同样适应易于使用的二进制格式，所以也可以用在网络游戏中。说不定全世界很多网络游戏都在使用这些工具。此外，如果觉得这些开发工具拥有太多不需要的功能以致进行了很多无用的处理，可以以此为参考自己来开发，想必不会太难。

* * *

如果将这一节所述的这类“RPC 功能”与之前介绍的 libevent 这样的“实现套接字 API 非阻塞化的 API”结合起来，就能具备作为网络游戏通信中间件的最基本的功能了。

0.3.3 ［补充］UDP 的使用

上一节和这一节以 TCP 为例进行了讲解，UDP 会在接下来的章节中介绍。目前在网络游戏中使用 UDP 主要有以下两个原因。

❶ 发送那些与可靠性相比到达速度更为重要的数据。

❷ 为了实现 NAT 遍历功能。

对于第 ❶ 点，比如在 FPS（First Person Shooter）中为了以最快的速度发送角色的移动信息而使用 UDP。在使用 TCP 的时候，数据包如果在发送过程中丢失了，就会再次发送。由于在该数据发送完毕之前下一个数据无法送达，所以会造成相当大的通信延迟。但是在使用 UDP 的情况下，不会对丢失的数据包进行重新发送，而是继续发送后面的数据包。在 Windows 和游戏机等非 UNIX 系统的主机上运行的软件，尤其是那些很占 CPU 的软件，由于经常会丢失数据包（在 Windows 内部），在这种情况下使用 UDP 可以将影响降到最低。

第 ❷ 点是在 P2P MO、C/S MO 架构的游戏中使用的技术，将会在第 5 章中详细介绍。

0.4　游戏编程基础

介绍完网络编程之后，这一节我们再来学习一下一般的游戏编程基础。

0.4.1　游戏编程的历史

自从 20 世纪 70 年代 Taito 公司的《太空入侵者》（*Space Invaders* ）开始，游戏编程方法基本上没有发生过变化。

使用的编程语言从当时的汇编语言，到之后的 C、C++、Objective-C，也没有发生根本性变化。与互联网编程的历史相同，几乎所有的游戏基本上都是以同样的方式开发的。

因为游戏编程是在一台机器上完成的，所以在处理流程方面并没有进行国际标准化，也没有像套接字 API 这样的事实标准库。现实情况就是，各种各样特定于游戏类型、特定于硬件平台的工具鱼龙混杂，各

个企业各自使用不同的工具[36]。为了理解网络游戏的实现方法，下面我们将以游戏编程的基本处理逻辑为中心加以说明。

> [36] 也有例外，比如 3D 多边形（在 3D 计算机图形中，为了表示立体形状而使用的多边形）的文件保存形式需要在各组织、各企业之间频繁交换，就此发起了 COLLADA（http://www.khronos.org/collada/ ）等标准。

0.4.2 采用“只要能画点就能做出游戏”的方针来开发入侵者游戏

对网络游戏来说，GPU（Graphics Processing Unit）和高速的渲染库等并非是必需的，如果将这些考虑在内的话就复杂了，所以在讲解游戏开发的基础知识时，我们以“只要能在屏幕上画点就能做出游戏”[37]的方针来说明。下面列出了一些需要用到的内容。

> [37] 在《ゲームプログラマになる前に覚えておきたい技術》（中文译名：成为程序员之前需要了解的技术）（平山尚著，秀和システム出版，2008 年）一书中有详细说明，请务必参阅。

- **调用 getKey() 函数可以知道按下了键盘上的某个键**

  返回值的第 1 个字节表示←（左方向键）、第 2 个字节表示→（右方向键），第 3 个字节表示空格键，由此判断按下了哪个键。

- **画面大小为 256 × 256 像素**

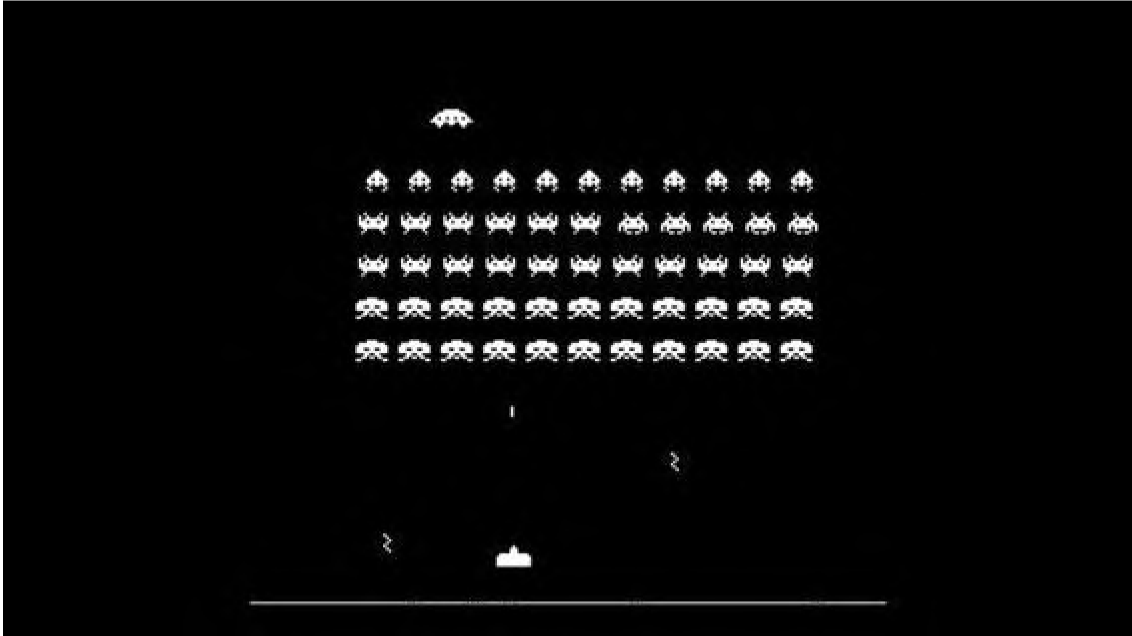
- **使用函数 point(10,10,3) 可以在画面的 (10,10) 处绘制“3”所代表的颜色**

  颜色值：0：黑色；1：白色；2：红色；3：绿色。

有了以上这些信息就可以制作出如图 0.7 这样的入侵者风格的游戏。看起来这与现在的游戏开发相距甚远，实际上并非如此。在能够实现像 *STAR STRIKE HD*[38] 这样的游戏的 PlayStation 3 上，也只是借助 GPU 和多核处理等手段来高速进行上述处理。首先我们忽略处理速度来加以考虑。

[38] 高速渲染大量物体的 360° 射击游戏的代表之一。

图 0.7　《太空入侵者》（入侵者游戏的例子）



图像由 Taito 公司提供。《太空入侵者》的第一代是在 1978 年发行的，它是入侵者类游戏的鼻祖。上图是截至本书撰稿时的最新版 *Space Invaders Infinity Gene*（http://infinitygene.net/ ）的截图。画面上方的是入侵者（敌对角 色），下面的是移动炮台（己方）。画面中央的短直线表示用子弹攻击入侵者，画面中锯齿状的格子是敌人的子弹。此外，本节之后给出的入侵者游戏示例的画面大小是 256×256 像素的。

0.4.3　游戏编程的基本剖析

前面所述的游戏整体结构，全部包含起来也就几个画面的程度。下面我们以 C 语言风格的伪代码来演示一下。话虽如此，这里的伪代码与实际运行的代码非常相近。

那么我们这就来看一下。MYSHIP 表示己方战机，INVADER 表示敌对的入侵者，MISSILE 表示己方导弹，将向画面上方飞去，BULLET 表示敌人攻击的子弹，将向画面下方飞来。

```
↓定义画面大小
#define WIDTH 256
#define HEIGHT 256
↓定义出场的图像编号（类别ID）
enum {
  MYSHIP = 0,          ←己方战机
  INVADER = 1,         ←敌对的入侵者
  MISSILE = 2,         ←己方子弹
  BULLET = 3,          ←敌方子弹
};    →（以下待续…）
```

接下来，我们再针对存在于画面上的物体（可移动物体）定义一个名为 Sprite 的类。Sprite 拥有图像编号（img)、坐标（x ，y ）、行进方向（dx ，dy ）这几个参数，move() 函数用于向行进方向前进一步。用 new 创建的对象初始状态下行进方向为（0，0），所以不会移动。在 hit() 函数中传入其他的 Sprite 对象，判断是否发生碰撞（在 8 像素矩形范围内是否有重叠）。

```
class Sprite
{
 public:
  int img;        ←图像编号（MYSHIP、INVADER、MISSILE、BULLET 中的哪一个）
  int x,y;        ←当前位置的坐标（单位为像素）
  int dx,dy;      ←行进方向，前进 1 帧（单位为像素）
  ↓构造函数
  Sprite(int x, int y, int img) {
    this->x = x;
    this->y = y;
    this->img = img;
    this->dx = this->dy = 0;
  }
  ↓移动 1 步
  void move

() {
    this->x += this->dy;
    this->y += this->dy;
  }
  ↓碰撞检测，范围为 8 像素
  bool hit
```

```
(Sprite *sp){
    if(!sp) return false;
    return (this->x + 8 > sp->x && this->y + 8 > sp->y
            && sp->x + 8 > this->x && sp->y + 8 > this->y);
  }
};    →（以下待续…）
```

上面这段代码只是 Sprite 类，下面将展示 main 例程。main 例程由以下两部分组成。

❶ 初始化

❷ 无限循环

在正式的游戏程序中也是如此。

初始化

首先来看一下初始化。在下面这段代码中，❶、❶’ 创建己方战机，将其置于画面下方。传入 MYSHIP 作为参数指定图像编号。游戏开始时，尚未发射导弹，所以不使用 new 创建（❷）。

接着 ❸ 处，因为游戏开始时敌方入侵者总共是 60 个（12 列 ×5 行），所以全部用 new 来创建。Sprite 类的实例数一下子增加了 60 个。创建好的 Sprite 指针全部存入名为 invaders 的数组中。

❹ 中，虽然入侵者也会发射子弹，但是在游戏开始时还没有发射，所以全部初始化为 0。敌方子弹也同样是同时存在多个，所以都存入了名为 bullets 的数组中。

```
int main()
{
  int i,j;
  ↓①己方战机出场
  Sprite *myship = new Sprite(WIDTH/2, HEIGHT*0.8, MYSHIP

); ← ①'位于画面下方正中央
  Sprite *missile = 0    ←②己方子弹
  ↓ ③ 入侵者出场
#define NUM_INVADERS(12 * 5)
  Sprite *invaders[NUM_INVADERS];
```

```
  for(i=0; i<5; i++)  {
    for(j=0; j<12; j++) {
      invaders[i*12+j] = new Sprite((WIDTH / 12) * j, (HEIGHT /
10) * i);
    }
  }
  ↓ ④ 入侵者同时发射的子弹最多为10 个
#define NUM_BULLETS 10
  Sprite *bullets[NUM_BULLETS];
  for(i=0; i<NUM_BULLETS; i++) {
    bullets[i] = 0;
  }
}     →（以下待续…）
```

以上就是全部的初始化内容。

无限循环

接着来看一下无限循环部分。❶ 处虽然用了 while(1) 这种没有循环条件的写法，但是在 PC 游戏中可以用 Esc 键等方式来终止游戏。

❷ 以下的部分在循环开始时获取按键输入。❷ 这一行调用 getKey 函数获取键盘状态。对于 Windows 应用程序来说，在消息循环中从 Windows 系统获取消息，然后进入条件分支。游戏机中经常调用这种简单的函数。

在这个入侵者风格的示例中，玩家操纵的只有己方战机，所以判断了按键后就能直接修改 myship 的 d*x* 值（行进方向）。在这个游戏中对行进方向的修改是不断累加的，所以如果一直按住相应的键，就会一直往那个方向移动。

按下空格键时，己方战机就会发射导弹，因此这时就用 new 来创建这个对象，将其图像编号设为 MISSILE，坐标与己方战机的当前坐标相同。将 d*y* 设为表示向画面上方前进的值：-1。

```
while (1)

 {    ← ①
  int key = getKey

();    ← ②
  if (key & 0x1) {        ←右方向键
```

```
    myship->dx = 1;
  } else if (key & 0x2) {       ←左方向键
    myship->dx = -1;
  } else if (key & 0x4) {       ←空格键
    if(missile == 0) {
      missile = new Sprite(myship->x, myship->y, MISSILE

);
      missile->dy = -1;
    } else {    ←没有按下任何键
      myship->dx = 0;
      myship->dy = 0;
    }
  }   →（以下待续…）
```

各个 Sprite 的行为 ——游戏逻辑主体

当主循环中按键输入完成之后，各个 Sprite 就要开始行动了。这部分是实现游戏策划内容的逻辑主体，所以相对比较长。

代码 ❶ 处，调用 move 函数将己方战机向行进方向移动 1 步，由于在之前的代码中指定了 1 或者 -1，因此每次循环移动 1 个像素。在每秒 60 帧的情况下就会移动 60 个像素。在商业游戏中，如果游戏速度过快，就会追加一些处理，比如使用空循环来加以调整，或者使用硬件的计时功能来等待一段时间等。

代码块 ❷ 中，如果己方战机发射了导弹，就调用 move 函数使其移动起来。与 myship 不同，子弹可能还不存在，所以必须使用 if(missile) 来进行判断。子弹飞出画面顶部后，就将其删除以释放内存。

继续，代码块 ❸ 中，调用 move() 函数使所有的敌方子弹都行动起来。这里最重要的就是碰撞检测。如果敌方的子弹击中了己方战机，那么游戏就结束了，所以这里以 myship 作为参数对所有的敌方子弹调用 hit() 函数，碰到了的话就调用 exit() 函数（很粗暴的方式呢！）。同样，当子弹飞出画面下方时也将其删除以释放内存，同时，将数组中的相应元素设为 0，重新初始化。

代码块 ❹ 是入侵者的行动逻辑。在商业化的入侵者游戏中，入侵者一般都按照波浪式等一些特殊方式移动，但是因为比较复杂，这里就

省略了。入侵者在被己方战机发射的导弹击中后会消失，所以在hit(missile) 的返回值为真时就将其删除。此外这里还使用了随机数以一定的概率使敌方发射子弹，发射子弹是通过创建（new）一个 Sprite 来实现的，在 for 循环中检测 bullets 数组中是否有空值（值为 0 的元素），如果元素为空则使用 new 创建。如果画面中已经存在了 NUM_BULLETS（NUM_BULLETS 是敌弹的最大个数）个子弹，那么这个循环就相当于什么也没有做。如果不这样事先设置上限值的话，就有可能发生内存不足。

```
↓移动己方战机和导弹、敌方子弹、敌人，碰撞检测
myship->move();          ← ①

if(missile) {            ← ②
  missile->move();
  if(missile->y < 0) { ←子弹飞出画面的话就将其释放
    delete missile;
    missile = 0;
  }
}

for(i=0; i<NUM_BULLETS; i++) { ← ③ 移动敌方子弹
  if(bullets[i]) {
    bullets[i]->move();
    if(bullets[i]->hit(myship)) exit(0);     ←被击中的话游戏结束
    if(bullets[i]->y > HEIGHT) {             ←飞离画面时将其释放
      delete bullets[i];
      bullets[i] = 0;
    }
  }
}

for(i=0; i<NUM_INVADERS; i++) { ← ④ 移动入侵者
  if(invaders[i]) {
    invaders[i]->move();
    if(invaders[i]->hit(missile)) {      ←打倒入侵者
      delete invaders[i];
      invaders[i] = 0;
    }
    if((random() % 10000) == 0) {        ←敌人偶尔会发射子弹
      for(int k=0; k<NUM_BULLETS; k++) {
        if(bullets[k] == 0) {
          bullets[k] = new Sprite(invaders[i]->x, invaders[i]->y,
BULLET);

          bullets[k]->dy = 1;
          break;
```

```
          }
        }
      }
    }
}      →（以下待续…）
```

绘制

主循环中各个可移动物体的行动完成了，就要进行最后的绘制工作了。在下面的代码段 ❶ 处首先清除画面上的所有物体。这是因为利用了眼睛的感知特点：“在描绘物体时，全部清除之后在稍微偏移的位置上进行绘制，就会感觉物体在动。”不过这与动画一样，重绘频率必须达到一定程度才能实现。

❷ 以下的绘制代码中，将 Sprite 类的实例变量作为参数传给 drawSprite 函数，分别绘制了 myship、missile、bullets 和 invaders。改变绘制顺序可以对将哪个物体绘制在最上方进行调整。击毁己方战机的只有敌弹，所以它们最重要，因此在最后绘制它们。

```
    ↓全部清除后进行绘制
    clearScreen();

        ← ①
    drawSprite

(myship);    ← ②
    drawSprite

(missile);
    for(i=0; i<NUM_INVADERS; i++) {
      drawSprite

(invaders[i]);
    }
    for(i=0; i<NUM_BULLETS; i++) {
      drawSprite

(bullets[i]);
    }
  }         ← while 无限循环结束
}           ← main 函数结束
```

至此，主循环（while 无限循环）和 main 函数都结束了。

子过程

下面来看一些子过程。首先要定义绘制的图像。每个 Sprite 由 64 个像素组成，比如子弹就是如下所示的形式。循环 8×8 次就能在画面上显示出类似子弹的图像。

```
00000000
00011000
00011000
00011000
00011000
00011000
00011000
00000000
```

首先在 imageData 中定义 4 个字符串（myship、invader、missile、bullet），实际中如何使其更具魅力是图形设计人员的工作。此外在 imageColor 中定义各个图像的绘制颜色。

```
↓ Sprite 的大小为 8×8。用 64 个字符来定义图像
char imaged

[BULLET + 1][] =
  {

“0001010010010101010101100001000010111110101110110101011010101111”
, ← myship

“1111010101110101111010010100010101010001010101010100101110101010”
, ← invader

“0000000000011000000110000001100000011000000110000001100000000000”
, ← missile

“0000000000011000000110000001100000011000000110000001100000000000”
, ← bullet
  };
int imageColor

[BULLET + 1] ＝
```

```
  {
   3,       ← myship 为绿色
   1,       ← invader 为白色
   1,       ← missile 为白色
   1,       ← bullet 为白色
  }
```

实现这一逻辑的是下面这段代码的代码块 ❶ 处的 drawSprite 函数。首先根据 Sprite 类的 img 成员变量来决定图像和颜色。

然后在 ❷ 的部分中循环 8×8 次，在循环体中调用 point() 函数。这里的要点就是根据各个 Sprite 的坐标以相对坐标来绘制。

❸ 处的 clearScreen 函数将整个画面全部涂成黑色。

```
void drawSprite

(Sprite *sp)    ← ①
{
  int i,j;

  if(!sp) return;

  char *toDraw = imageData[sp->img];    ←确定绘制的图像
  int col = imageColor[sp->img];        ←确定绘制的颜色

  for(i=0, i<8; i++) {         ← ②
    for(j=0; j<8; j++) {
      point(sp->x + j, sp->y + i, toDraw[i*8 + j] * col);   ←以相应
的颜色绘制每个点
    }
  }
}

void clearScreen

()              ← ③
{
  int i,j;
  for(i=0; i<WIDTH; i++) {
    for(j=0; j<HEIGHT; j++) {
      point(i,j,0);
    }
  }
}
```

至此，这个入侵者游戏的实现就完成了。

0.4.4 游戏编程精粹——不使用线程的“任务系统”

这里所要说明的实现方法从 30 年前的入侵者游戏到现在的 iPhone 游戏、MMORPG 游戏，几乎没有发生过变化。

从上面这个例子中我们可以看到，最重要的就是“许多物体在同一时刻都在各自运动，但实际上并没有用线程来实现”。在游戏逻辑中，所有的内容都是依次处理的。

在这个入侵者游戏的示例中，如果使用线程来实现多个物体又会如何呢？考虑一下每个 Sprite 使用 1 个线程的情况，如果就这样使用操作系统提供的线程的话就会陷入困境。

- **所有的敌方子弹可能不会完全以相同的速度移动，这样游戏的平衡性就会被破坏了。**
- **己方的子弹与敌人进行碰撞检测时，可能会发生两个以上的物体同时碰撞的情况，所以必须具有排他机制。**
- **比 CPU 内核数多得多的线程数会导致性能变差。**

综上，要严格实现游戏的策划内容，就会产生线程的运行计时和调度、正确控制排他机制等各种各样的问题。随着线程数的增加，处理负荷会变得相当高。

所以在网络游戏编程中一般不会使用多线程来控制多个敌方子弹的行为。这在业界术语中称为“任务系统”（Task system）。任务系统就是像 libevent 部分的“回调函数”所介绍的那样，以“1 个导弹每次只前进 1 帧”这种非常小的单位来进行处理，将此作为一个函数来定义。这样，对所有的导弹在 1 帧内依次调用该函数就能实现实质上的并行处理。由于不使用操作系统的本地线程来切换处理，运行速度就会很高。

但是为了充分发挥如今的 CPU 的能力，根据音效处理、AI、网络、主循环、渲染等方面，有时会用到 3～5 个线程。今后多核处理器得以

普及后，可以更有针对性地使用多个内核来处理物理仿真和图形绘制等方面。使用多个线程来实现游戏的处理逻辑是不太合乎常理的。

0.4.5　两种编程方法的相似性——不使用线程

对于上述不使用线程的这一点，读者是否注意到，这与之前网络编程中的服务器端编程方法是相同的。将这两者进行比较，可以看到它们有如下这些相似之处。

- **网络编程**

  对所有的套接字调用 select() 函数进行轮询，对于需要处理的内容执行 read 和 write 操作，调用回调函数来逐个处理。

- **游戏编程**

  对所有的可移动物体（这个例子中是 Sprite 类的实例）以帧为单位进行轮询，对于需要进行处理的物体调用回调函数来使其行动（进行 move 和 hit 处理）。

当并行运行的物体数达到处理器内核的数十倍以上，需要严格控制处理顺序时，使用多线程恐怕是不可能实现的。

0.5　小结

本章介绍了一般的游戏编程和网络编程的基础，读者对需要什么程度的基本技术已经有了一些认识了吧？在掌握了本章的基础知识之后，从下一章开始，我们将逐步进入网络游戏编程的核心。

专栏　确保开发效率和各平台之间的可移植性

网络游戏的开发中，也有以 Linux 服务器作为主要目标的云基础服务（IaaS，Infrastructure as a Service），正式服务器的运行环境平台集中在 Linux 的 Red Hat Linux 和 CentOS 系统 。另一方面，客户端的平台环境则非常多样，包括 iPhone、Web 浏览器、Windows、Android、移动电话、游戏机等。

提高开发效率

为了提高开发效率，这里在平台方面提出两点主要要求。

- **正式服务器采用 Linux 操作系统，但开发环境则是在 Windows 下使用 Visual Studio 以高效地进行开发。**

- **服务器端和客户端在碰撞检测等方面使用相同的游戏处理代码。**

这两点都与“同一个程序要在不同的操作系统上运行，即确保可移植性”这一要求联系在一起。

同时，这也要求在 C/S MMO 中，服务器端和客户端使用相同的编程语言。也就是说，客户端和服务器端双方都要使用 C、C++ 或者 Java 来编写。顺带一提，现在这种情况下能够选择的编程语言只有 C、C++ 或者 Java，但是将来，C#、Objective-C 和 Go 语言等也极有可能加入这一行列，轻量级语言和 node.js 等也是有力的候选语言。

P2P MO 游戏不包含服务器，所以一般不需要满足这些要求。

使用封装保持源代码级的兼容性

除了为了实现画面渲染、声音输出、键盘和鼠标输入、触屏、视频输出等客户端用户体验的功能，在使用 C/C++ 的情况下，现在通过简单的封装就能保证源代码级的兼容性。这种封装自然是在通信中间件的层次上实现的。

降低 OS 差异性的封装工作

为了降低 OS 的差异性，需要对以下这些方面进行封装。

- **内存管理**

  malloc 几乎在所有的系统中都会使用，所以很容易进行封装。

- **套接字 API**

Windows 与 UNIX 系统（包括 iOS）中，套接字 API 的方法有所不同，所以需要将函数全部封装。

- **线程**

  将 pthread 的基本 API 进行封装就足够了。线程的使用部分限定在客户端中。调度标志等细微部分在每个操作系统中都不兼容，就像本书建议的那样，不要使用多个线程，使用单线程来实现服务器端就不会有问题。在完全不使用线程的情况下，不需要与服务器端共享源代码，所以也需要进行封装。

- **信号**

  远程管理服务器的情况下需要使用信号，而这是一种可移植性很低的方法，所以并不推荐。实现工作在 TCP 之上的 HTTP 服务，再通过套接字来实现服务的停止具有更高的可移植性。

- **事件与计时**

  使用本书介绍的 libevent，可以有效地进行封装，性能非常高。

基本上以接近 POSIX（Portable Operating System Interface of Unix，可移植操作系统接口）标准来进行封装，可以降低整体的工作量。

习惯用 Qt 和 Boost 之类通用封装库的技术人员有很多。像 CAPCOM 公司这样使用公司内部独立开发的封装库的企业也有很多。如果可以在发售游戏的平台上使用自己用得惯的程序库，当然就选择这种库了。

第 1 章　网络游戏的历史和演化：游戏进入了网络世界

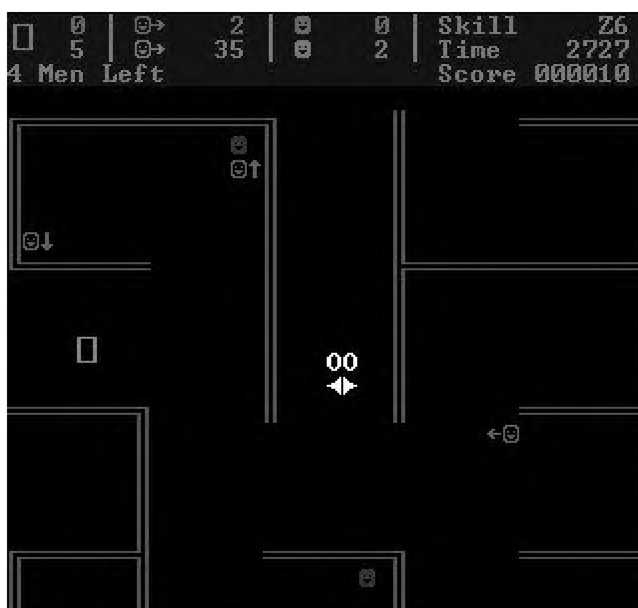
微软的 Kinect 、GPGPU（General Purpose GPU，通用计算图形处理器）、苹果公司的 iPad、立体显示和自动化设备、在 Social Graph 中记录生活点滴……这类例子举不胜举。新的技术在明确其实用性之前，都是首先运用在游戏中，然后才逐渐被了解并得到广泛普及。

其实电子游戏在最初加入网络功能的时候也是如此。

在 Web 浏览器诞生大约 10 年前的 20 世纪 80 年代前期，以太网和 TCP/IP 协议的出现终于将计算机用户连接了起来，实现了高速、实时（存在毫秒级的延迟）地互相传输数字数据。1983 年，网络操作系统 Novell Netware 发布了第一个版本，当时为了宣传该系统具有能够高速访问网络的能力，他们准备了一款网络对战射击游戏 *Snipes* （参见 图 1.A）[1] 。

> [1] 高速的文件共享具有很大的实用价值，但 Netware 经过了多年才得到普及。

图 1.A　*Snipes*



由 MyAbandonware 提供。http://www.myabandonware.com/

TCP/IP、UNIX、HTTP、Java、Flash……各种网络新技术层出不穷，并且纷纷运用在了游戏中。而现在运用了云计算、传感网络等技术的游戏，又能变得多有趣呢？开发者对这些话题的讨论永无休止。

随着网络技术的进步，网络游戏在不断成长。第 1 章就将为读者讲述那些网络游戏技术的历史背景以及相关模式的演化。

1.1　网络游戏的技术历史

本节以网络游戏的技术历史为题，总结如今与网络游戏紧密相关的技术的发展历程。

1.1.1　网络游戏出现前的 50 年

网络游戏并非诞生于一夜之间，它是在网络和电子游戏的基础上，经过日积月累的变迁之后产生的。虽说是日积月累，但其实这两个领域也顶多只有约 50 年的历史。开创时期的领军人物现在也还健在。如今，在网上也能获取很多相关资料，当你在网上查找这些信息时，可以很容易找到网络和游戏方面的内容。

但是按照时间顺序将这两方面结合在一起加以说明的资料却没有，所以我试着制作了一张图（参见图 1.1）。图 1.1 展示了每当开发了新技术并且加以运用时，将会产生怎样的新游戏。其中横轴代表年份（时间），纵轴代表相关技术领域。

图 1.1　游戏 / 网络游戏的发展

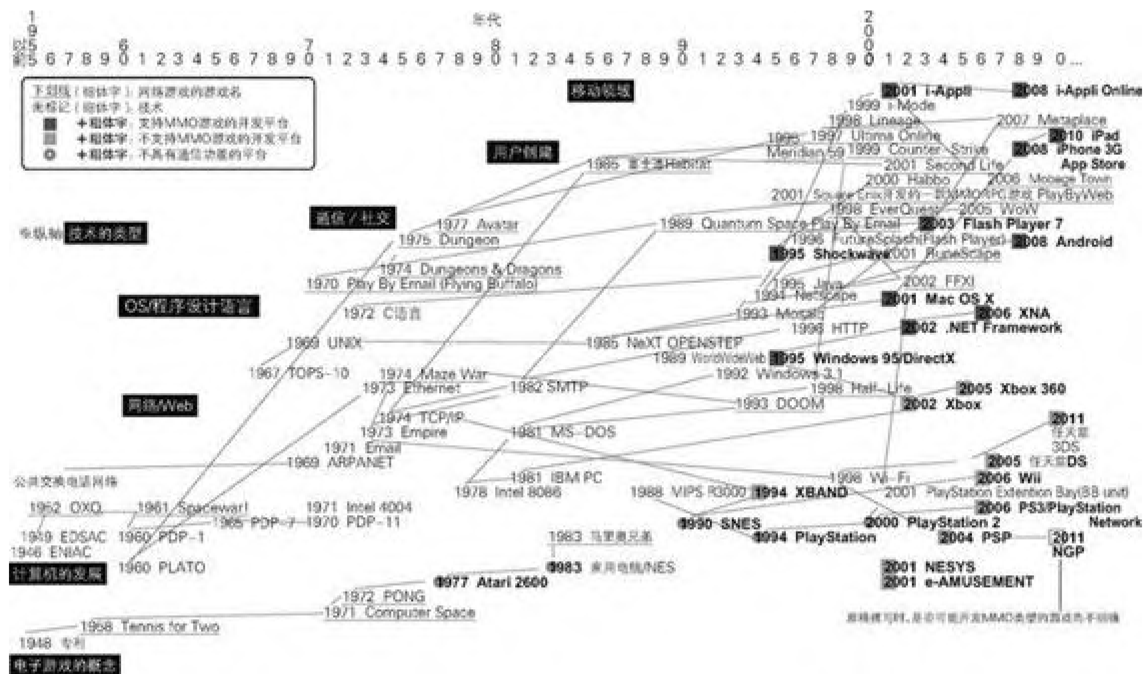


图 1.1 将会在后面加以介绍，首先，我们分年代了解一下游戏的发展过程。

1.1.2　20 世纪 50 年代前：计算机诞生

此时正值第二次世界大战，为了在战争中进行弹道计算，名为 ENIAC 和 EDSAC 的这类使用电子管的计算机开始投入使用。电话通信对军事活动来说非常重要，因此这一技术发展相当迅速，电话交换网开始成为主要的通信手段，这正是自动交换机的引入所带来的进步。

二战结束的时候，世界上的人口只有 25 亿 [2] ，虽然这是个只有计算机、研究所和军事设施的时代，但在 1936 年，美国就对当时的电子管显示设备在娱乐方面的使用申请了专利 [3] 。

> [2] 据美国人口普查局推断，2010 年 7 月全球人口已达到 68 亿。参考网址：http://www.census.gov/ipc/www/popclockworld.html

> [3] “Cathode Ray Tubes”(F.J.G.Van Den Bosch、US2062538(A))。

之后世界经济得以迅速发展，产生了大量剩余资源和剩余时间，开始向大量消费社会发展。这为电子游戏的发展提供了空间。

1.1.3 20 世纪 50 年代：早期的电子游戏

世界上最早的电子游戏 OXO[4] 是使用设立在英国剑桥大学的第一台计算机 EDSAC 开发的。该游戏使用 CRT（阴极射线管）显示器，这是个如其名的 OX 游戏。OXO 游戏是玩家与计算机进行 1 对 1 对局的单人“完全信息博弈”[5] 型游戏。

> [4] 在英语版 Wikipedia 和 Youtube 上可以找到相关参考内容：
> 参考网址：http://en.wikipedia.org/wiki/OXO ; http://youtube.com/watch?v=Xyi0_ViYmFY

> [5] 这里所说的“完全信息博弈”是指“二人零合有限确定完全信息博弈”。（零和：指一方的收益必然意味着另一方的损失，博弈各方的收益和损失相加总和永远为“零”。双方不存在合作的可能。有限：指双方玩家的手数总和是有限的。）游戏进行时的所有相关信息对所有参与者都是公开的，不像随机数等，完全信息博弈在理论上是可以进行预判的。

1958 年，布鲁克海文国家实验室（Brookhaven National Laboratory）为了让来访的客人消磨时间，制作了一款名为《双人网球》（*Tennis For Two*[6]）的双人游戏。这是最早的多人游戏。该游戏的特点是：它不是在计算机上开发的，而是仅仅使用模拟电路[7] 制作出来的。在游戏中可以看准时机使用专门的开关来打开、关闭电流，同时还能使用把手调节击球的角度。

> [6] 请参见“Retromodo:Tennis for Two, the World's First Graphical Videogame”(Dan Nosowltz、Gizmode)
> 网址：http://gizmodo.com/5080541/retromodo-tennis-for-two-the-worlds-first-graphical-videogame

> [7] 电容器（蓄电器）和继电器等。

《双人网球》的玩家数为两人，在游戏中，双方玩家需要把握时机将球打向对方，双方分别在球桌的两端击球，同时注意不要把球打到球网上。这一规则非常简单，人们理解起来毫不费力，所以在这个游戏中并没有加入胜负判定，因为人们可以自己来判定胜负。

当时，不管是否使用了计算机的电子游戏，只要是用了 CRT 示波器[8] 的设备，就常会使用多个发光二极管，通过控制其明暗来输出逻辑状态。

[8] 使用波形来表示电气信号（电压）变化的测量仪器。

此外，在 20 世纪 50 年代已经开始出现了一些重要术语，请注意本章中粗体所示的术语。

1.1.4　20 世纪 60 年代：各种颇具影响的机器登上历史舞台

1960 年，对后来的电子游戏产生重大影响的两款商用机器开始发售，那就是 DEC PDP-1 和教学用的 PLATO。

首先来看一下 PDP-1。这款机器拥有 9KB 以上的内存和 200KHz 的时钟速度，就当时来说，它的处理性能十分强大。PDP-1 的设计思想发挥了非常大的作用，在整个计算机历史上留下了重要的一笔。另一方面，当时其他很多计算机的价格都超过了 100 万美元，而整套 PDP-1 才 12 万美元，相对来说非常廉价，但是这款机器只制造了 50 多台就停止生产了，在商业上并不成功。

PDP-1 可以高速处理 sin()、cos() 这类算术函数。1961 年，麻省理工学院计算机科学专业的 3 名研究人员 Peter Samson、Dan Edwards 和 Martin Graetz 运用 PDP-1 的处理能力开发了一款单人射击游戏《空中大战》（*Spacewar!*，参见图 1.2），游戏中的玩家需要躲避漂浮在宇宙中的敌人，同时发射导弹击倒敌人。《空中大战》与如今使用操作杆等设备进行操作的游戏有着基本相同的外观和感觉，它成为了最早的可供多人进行的游戏。

图 1.2 《空中大战》（以及 PDP-1 的显示设备）

图片：Joi Ito

※*Spacewar!* 。参考网址：http://en.wikipedia.org/wiki/Spacewar ！（最后一次访问是在 2011 年 12 月）

接着是 PLATO（Programmed Logic for Automatic Teaching Operation），它是由伊利诺伊大学（University of Illinois）开发的用于教学的计算机。虽然在商业上并不成功，但它引入了分时系统[9] 等先进功能，以大学为中心进行销售，最终出售了 1000 台以上的终端。之后，世界各地的学生们使用 PLATO 开发了各种各样的游戏。

> [9] Time Sharing System(TSS)。交替运行多个用户的程序，使多个用户能够共同使用同一台计算机。

PLATO 系统在 1960 年到 1970 年，经过了多次版本更新后确定了最终的形式。最早的版本是安装在由伊利诺伊大学开发、名为 ILLIAC I 的具有高达 5 吨的电子管计算机上的，并配备了显示设备。最初的设计目的是为了实现“跳过课题布置进行计算练习”。1961 年，PLATO 的第二个版本是为了让两名学生能够同时进行计算练习。

之后，PLATO 仍在继续更新版本。1967 年，PLATO 3 在 CDC 1604 计算机（由日后创立了专门生产超级计算机的 Cray 公司的 Seymour Cray 研发）上，实现了能够让 20 个人同时进行计算练习的功能，此外还使用了称为 TUTOR 的语言处理系统，老师们不仅能提供计算练习，还能使用图像和文字自由编写学习教材。在学校让多名学生同时或者非同时进行计算练习，这一要求在当时的其他计算机系统上是无法实现的，这成为了引入分时系统的重要契机。

另一方面，在 1957 年，前苏联发射了人造卫星伴侣号（Sputnik1），成功围绕地球轨道航行。美国受此冲击，为重获优势地位，他们认为应该对能够高度应对攻击的信息通信技术进行大规模投资，为此将分组通信（Packet Communication）和分组交换技术联系在了一起。

由此研发了计算机网络 ARPANET，1969 年，美国国防部开始投入使用。当时只在非常重要的国防要地设置了 4 台计算机，以 50kbit/s 的速度将它们联结在一起，之间采用分组交换机制。而在之后的 10 年中，主机数量飞速增加（参见图 1.3）。1968 年左右提出了汇集各种网络技术的 RFC 文档（第 0 章也有提及），1969 年公开发布了 RFC 的第一份文档 RFC 1。

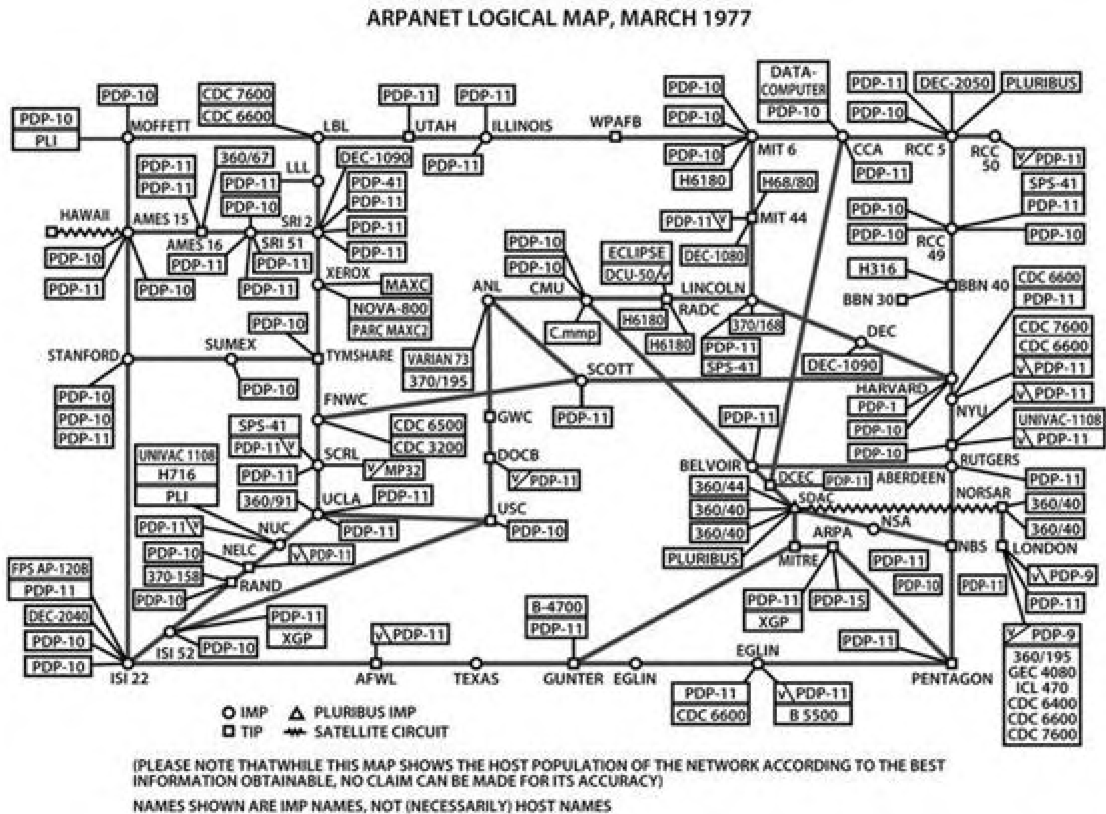
连接到 ARPANET 的计算机大多是 PDP 系列的，在 1960 年到 1969 年的 10 年内，PDP 系列从 PDP-1 到 PDP-10 进行了多次频繁的更新，微处理器也不断变更，所以就必须要在不同的机器之间对程序进行移植。为此，产生了 TOPS-10 这种吸取各机器之间差异的系统。TOPS-10 是操作系统的始祖。

TOPS-10 不仅具有可移植性，还引入了内存保护机制等为了充分发挥不断增强的机器性能而提出的概念。根据这些方面的功能，在一个机器上可以并行运行多个用户的多个进程。对共享内存（Share Memory）[10] 和套接字这类进程间通信（InterProcess Communication，IPC）[11] 概念的研究也是在这个时期开始发展的。

> [10] 有关共享内存的话题将在第 5 章中详细介绍。

> [11] 正如字面意思那样，这是一种在进程之间进行通信的机制，既包括同一台机器中不同进程之间的通信，也包括通过网络与另一台主机的进程进行通信。

图 1.3　ARPANET



※ARPANET Logical Map March 1977。参见 http://som.csudh.edu/cis/lpress/history/arpamaps

1969 年，供 PDP-7 使用的 UNIX 系统诞生了。Kenneth Thompson 针对 Multics 操作系统开发的 *Space Travel* 对 UNIX 的开发起到了重要作用。开发了 UNIX 的 Kenneth Thompson 团队起初是使用汇编语言将 *Space Travel* 移植到 PDP-7 上的，最后灵活运用了这些经验，开发了针对 PDP-7 的操作系统。Thompson 和 Dennis Ritchie 逐步实现了独立的文件系统、多任务机制和 shell 命令行等必需的要素。

即使在 21 世纪的今天，特别是任天堂的 Nintendo Dual Screen（NDS）和索尼电脑娱乐（SEC）的 PlayStation Portable（PSP）等便携式游戏机，都不使用庞大的操作系统，而倾向于直接利用硬件性能，程序的开发很多都是基于独立的文件系统和多任务机制。可以说，这些工作有一部分是和操作系统本身的工作重复的。在今后的便携式游戏机中，随着可用的计算机资源的增加，移植层的功能也应该更为丰富，那么这些工作的必要性就有可能降低了。

1.1.5 20 世纪 70 年代：网络游戏的基本要素

20 世纪 70 年代是非常重要的时期，在这一时期涌现的所有基本要素为如今的网络游戏形式奠定了基础。

首先在计算机方面，1971 年开始大规模生产世界上的首批商用处理器 Intel 4004（系列），计算机的价格也已降到普通家庭就能购买的水平。网络技术同样取得了巨大进步。1971 年，Email 作为杀手级应用程序，开始在 ARPANET 中普及。1973 年发明了以太网技术，接入网络的成本急剧下降。1972 年，C 语言诞生，它作为“高度可移植的汇编语言”在如今的游戏开发中仍然被广泛使用。计算机用户进行通信的机会越来越多，1974 年制定了 TCP 协议，这是如今的网络游戏的基础协议。

1971 年，创立雅达利（Atari）公司的 Nolan Bushnell 和 Ted Dabney 开发了名为 *Computer Space* 的最早街机游戏（商用），1977 年发售了名为 Atari 2600 的最早 ROM（Read Only Memory）卡带型电视游戏机（家用），获得了巨额利润。不过，将街机游戏和电视游戏机网络化则是之后的事情了。

1973 年，John Daleske 和 Silas Warner 利用单独的网络功能在之前提及的 PLATO 上开发了 *Empire* 游戏，接着就是 *Maze War* 。*Maze War* 的游戏规则是：几个人在立体迷宫中行走的同时互相射击，被击中就会失分，击中别人就能得分。该游戏是如今一种重要游戏类型 FPS（第一人称射击）游戏的先驱。

1974 年，名为《龙与地下城》（*Dungeons & Dragons* ）的桌上 RPG[12] 游戏首次大规模商业化，受其影响的开发者为数众多（笔者也是其中之一）。1975 年，还是在 PLATO 上重现了这款游戏，名为 *Dungeon* ，这是首款电脑 RPG 游戏，以文字来表示游戏内容 [13] 。没多久，*Dungeon* 针对网络确立了基于文字的 MUD（Muti User Dungeon）游戏形式 [14] 。如今基于文字的 MUD 游戏仍在各类网站上运营 [15] 。

> [12] 桌上角色扮演游戏。与电脑 RPG 游戏不同，这是一种玩家坐在一起，使用纸和铅笔一边对话一边进行的游戏。

[13] 可以参考包含 *Dungeon* 的服务器端程序的文档：
http://www.filewatcher.com/p/dungeon-1.0.tgz.722959/share/dungeon/history.html
FORTRAN 源代码：ftp://ftp.cse.buffalo.edu/mirror/BSD/FreeBSD-Archive/ports/i386/packages-4.6.2-release/All/dungeon-1.0.tgz

[14] 顺带提一下，PLATO 上最具人气的 MUD 游戏是 *Avatar* ，研究人员和学生们在这款游戏上花费了大量时间。

[15] 比如网站 mudcommect.com 上使用基于 Java 的 telnet 客户端将浏览器和 MUD 服务器连接在一起，使玩家可以随时享受 MUD 的乐趣。

到 70 年代后期，在活跃于 PDP 系列的 UNIX 系统上开发出了各种各样的 MUD 服务器端，其中的技术基础也用在了以后的游戏服务器端中。

1.1.6 20 世纪 80 年代：网络对战游戏登场

1981 年，IBM PC[16] 上市，计算机开始大规模普及。1982 年制定了发送 Email 的标准协议 SMTP（Simple Mail Transfer Protocol，RFC821），使用电话线和调制解调器将个人电脑和服务器连接、使用 ISP 服务并且在 Email 地址上加上 @ 标记来发送电子邮件，成为了很普通的事情。网络操作系统也在不断进步，1983 年 Netware 出现，与此同时在 Netware 平台上开发了一款名为 *Snipes* （参见图 1.A）的 P2P 网络对战游戏。

[16] IBM 最早发布的个人电脑被认为是 PC/AT 兼容机的基础。有时也指计算机结构。

1983 年，随着雅达利游戏机的日渐衰退，任天堂的家用电子游戏机（Nintendo Entertainment System，缩写为 NES，或称为 Famicom）快速普及，《马里奥兄弟》（1983 年）、《塞尔达传说》（1987 年）[17] 等游戏发售了几百万份。

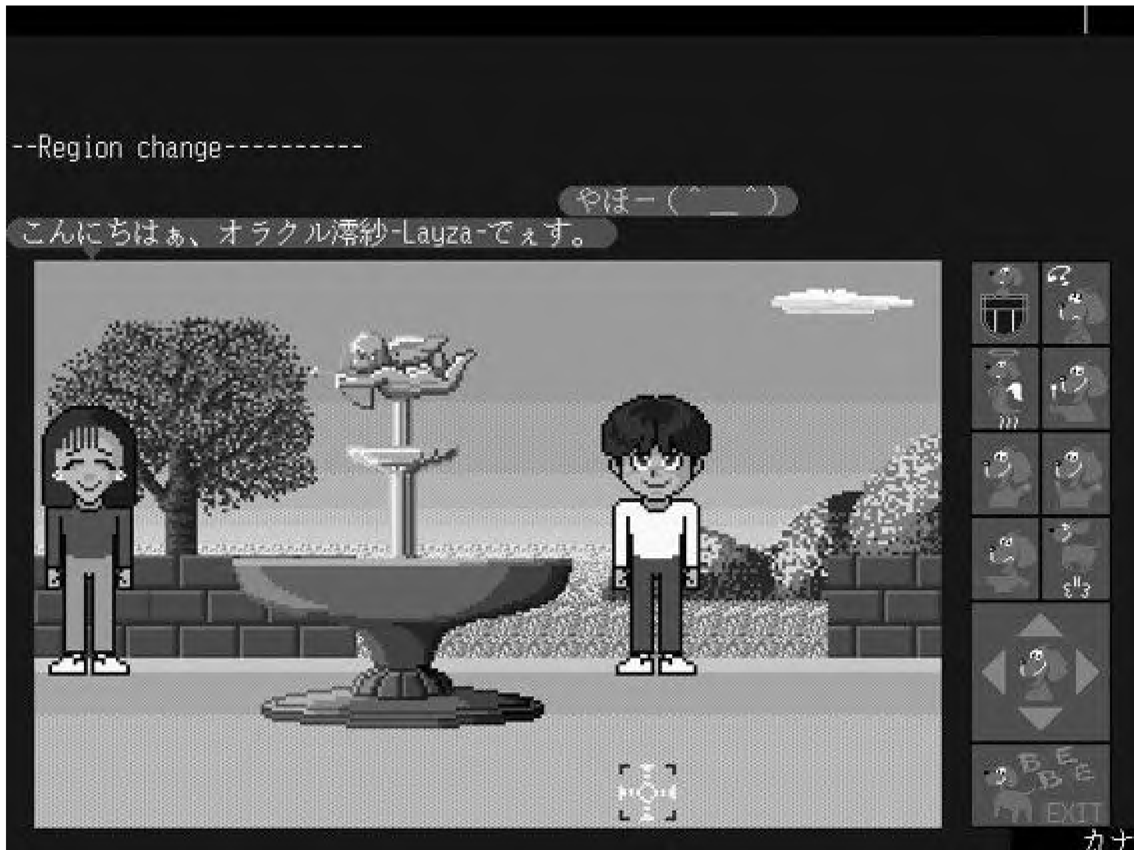
[17] 是 1986 年在 Famicom Disk System 上推出的，由宫本茂设计。

20 世纪 70 年代的文字 MUD 在进入 20 世纪 80 年代后，在富士通 Habitat（参见图 1.4）上开始以 2D 图像显示游戏内容，这与如今的大型多人网络游戏（MMO 游戏、MMOG）使用了基本相同的客户端 / 服

务器端（C/S）架构。使用 Email 进行 RPG 游戏的 Play by Email[18]（以邮件的方式进行游戏）的玩家也在增加，与网络上不认识的玩家一起游戏的机会也在增加。

[18] 也称作 Play by post 类型的游戏。

图 1.4　富士通 Habitat



※ 图像提供者：g-search。参考网址：http://www.g-search.or.jp/

2D 图像显示的先驱，提供可视化通信的聊天室服务。

参考“富士通可视化通信 15 年的历史进程”：http://j-chat.net/habitat/history.html

之后在 1989 年，在 CERN[19] 上开发了 HTML 和世界上首个 Web 浏览器 WorldWideWeb。1989 年后期，连接到互联网的主机数量达到了 30

万台。

> [19] 欧洲核子研究组织（European Organisation for Nuclear Research），http://cern.ch/

1.1.7 20 世纪 90 年代：游戏市场扩大

20 世纪 90 年代初期，电视游戏机市场急剧扩大，在任天堂的 Super Nintendo Entertainment System（SNES）和世嘉企业（当时叫做 SEGA enterprise）的 Mega Drive 上，开始了使用电话线网络对战游戏服务 XBAND[20]，但是最终在商业上失败了。当时的带宽速度和严重的延迟对持续高负载的游戏来说很成问题。

> [20] 在美国以 GENESIS 为名进行发售。

另一方面，MMOG 中的一款游戏《子午线 59》（*Meridian 59*）在商业上获得了极大的成功，而在 P2P 对战游戏中，1993 年发布的《毁灭战士》（*DOOM*，参见图 1.5）则引起了极大轰动。

图 1.5 DOOM 系列

DOOM II® © 2010 id Software LLC, a ZeniMax Media company. Bethesda Softworks, ZeniMax and related logos are registered trademarks or trademarks of ZeniMax Media Inc. in the U.S. and/ or other countries. DOOM, DOOM II, DOOM II (stylized), id and related logos are registered trademarks or trademarks of Id Software LLC in the U.S. and/or other countries. Developed in association with Nerve Software, LLC.

图片由 Bethesda Softworks/Zenimax Asia 提供。这是一款引起了强烈轰动的 P2P 对战游戏，FPS 的代表作之一。该图片是 1994 年发布在 Xbox LIVE 街机上的《毁灭战士 2》（*DOOM 2* ）的截图。Http://www.bethsoft.com/jpn/game/doom2.html

在 Web 方面，1993 年 NCSA（National Center for Supercomputing Applications，美国国家超级计算机应用中心）的 Mosaic 浏览器以及 1994 年 Netscape Navigator 浏览器的问世，不仅可以使用文字

来表示内容，还可以显示图片，为基于浏览器的网页游戏提供了契机。

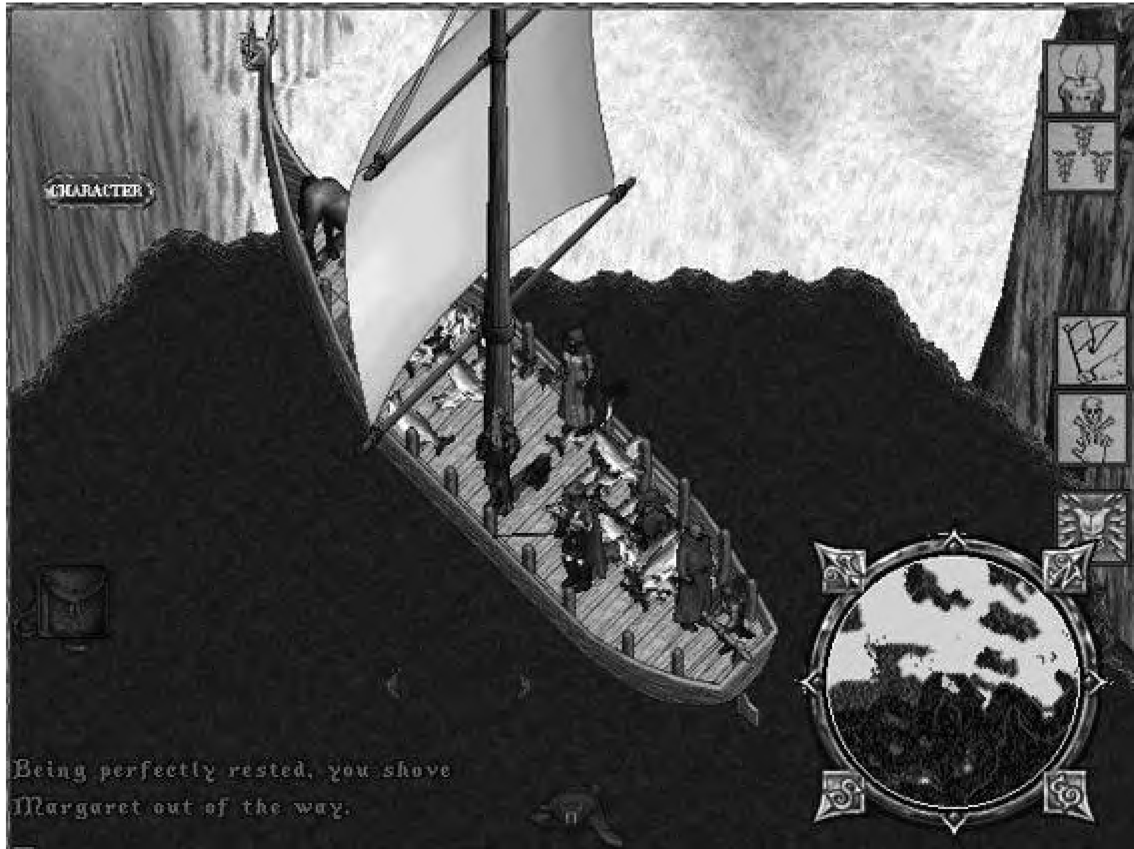
到了 20 世纪 90 年代后期，针对 Netscape Navigator 浏览器开发了 Shockwave（现在是 Adobe Sockwave）、Flash（现在是 Adobe Flash）、Java 等扩展功能，同时也开始运用在游戏中。笔者使用 Java Applet 开发 MMORPG 游戏 *Lifestorm* （参见图 1.6）也是在那个时候。

图 1.6　Lifestorm

此外，1997 年《网络创世纪》（*Ultima Online* ，UO，参见图 1.7）进一步推动了 MMOG 的发展。1999 年，美国索尼在线娱乐（Sony Online Entertainment，SOE）发行的《无尽的任务》（*EverQuest* ）将真正的 3D 显示带入了 MMOG 中。这些软件几千日元的包装费加上每月固定收取的 1000 日元，一年可以获得数亿日元的利润，但是与现有的游戏软件相比，还需要庞大的服务器运营成本。顺带一提，UO 的制作者 Richard Garriott 称“我最大的失败就是月收费一律 9 美元”[21] 。

[21] *Developing Online Games: An Insider's Guide* （p.xxuil、Jessica Mulligan、Bridgette Patrovsky 著，New Riders Publishing，2003）

图 1.7 《网络创世纪》



©Electronic Arts Inc. Electronic Arts, EA, EA GAMES, the EA GAMES logo, Ultima, the UO logo and Britannia are trademarks or registered trademarks of Electronic Arts Inc. in the U.S. and/or other countries. All rights reserved.

图片由 Electronic Arts（株）提供。
http://www.eajapan.co.jp/

网络 RPG 游戏的先驱。该图片是 1998 年在日本进行发售时的截图。http://ultimaonline.jp/

P2P 游戏的设定数据文件可以被替换，于是，玩家开始使用这项功能制作出各种各样的衍生游戏。这些衍生游戏被称为 MOD（全称 *Modification* ，游戏增强程序）[22] ，为了交换 MOD 而产生的网络社区随之急速扩大。

> [22] 虽然大多指的是用户制作的 MOD，但也包括厂商制作的扩展包和作为附加物发售的内容。

由 Valve Software 公司开发、Sierra 公司在 1998 年发行的 FPS 游戏《半条命》（*Half-Life* ）的 MOD《反恐精英》（*Counter-Strike* ）反响巨大，特别是在亚洲，影响力之大，竟然达到了在网络上出现了数万个网络游戏咖啡馆的程度。在游戏玩家中也存在着游戏制作者。

尽管如此，其实直到 20 世纪 70 年代前期，电子游戏的玩家和游戏开发者还基本都是研究人员、大学生和计算机迷，他们大多都在之前提到的网络社区中。20 世纪 90 年代，家用游戏和商业电子游戏纷纷产业化，因为考虑到 MOD 变得日益重要，制作商开始和消费者分离。

今后会怎么样我们无从断言，但是就在本书出版之时（原稿写作时间：2008～2010 年），读者眼中的游戏业界景象恐怕是大企业面向数万消费者提供统一化的商品。这可能是由于 20 世纪 80 年代至 90 年代的电子游戏形式给人们留下了深刻印象。

1.1.8　本世纪前 10 年的前期：网络游戏商业化

本世纪初，游戏市场上出现了利润高达数百亿日元的韩国《天堂》（*Lineage* ）[23] 系列游戏，因为有了成功实现网络游戏商业化的例子，其他的大企业也纷纷加入这一行列。

> [23] 韩国开发的 MMORPG。根据韩国漫画 *Lineage* 改编。

2000 年，SCE 发售了 PlayStation 2 游戏机，2001 年针对这款游戏机发售了 PlayStation Extension Bay（用于连接网络的辅助设备。BB Unit），对此，Square（现在是 Square Enix）的《最终幻想》（*Final Fantasy* ）系列、《信长之野望》系列（光荣株式会社出

品，当时公司名是 KOEY，现在是 Tecmo Koei Games）等日本知名的系列游戏纷纷发布网络版，如今收益依然可观。

在街机中，世嘉、TAITO、KONAMI（科乐美）等大型企业增强了网络对战功能，由此提升了重玩率，大获成功。

大型企业正致力于商业化，另一方面，一些中小企业提出了“由玩家自己创建游戏内容”（User Created）的概念，比如美国 Linden Lab 开发的《第二人生》（*Second Life* ）和芬兰 Sulake Corporation 开发的 *Habbo Hotel* （参见图 1.8），以此开始了新的服务，并从中产生了大量的内容和收益。

图 1.8　*Habbo Hotel*

©Sulake Corporation Ltd.

图片由 Sulake Corporation 提供：http://www.sulake.com

“用户创建”（User Created）类型的在线交流服务的典型代表之一。http://www.habbo.com/

在移动电话方面，自 2001 年 NTT DoCoMo 公司推出的 i-appli 之后，面向移动领域开发了网络游戏，虽然得到了广泛普及并且获得了一定的利润，但是随之产生的高额流量费用却也成了一个社会问题。

1.1.9 本世纪前 10 年的后半期：基于 Web 浏览器的 MMOG 在商业上获得成功

本世纪前 10 年的后期，Flash 和 Java 大量普及，加之服务器和 TCP 的使用，人们开始通过计算机自由通信，这些技术很快就应用到了游戏中，基于 Web 浏览器的 MMOG 大规模增加，数百个网页游戏在商业上获得了成功 [24] 。

> [24] 排行榜可以参考如下网页：http://mmorpg.toparena.com/

在游戏设计方面的研究也在进步，2004 年美国暴雪娱乐公司发布《魔兽世界》（*World of Warcraft* ，*WoW* ，参见图 1.9），该游戏的设计将重点放在“一个人也能愉快地进行游戏”上，由此获得了 1200 万以上的惊人玩家数 [25] 。

> [25] http://us.blizzard.com/en-us/company/press/pressreleases.html?101007

图 1.9 《魔兽世界》

Image used by permission. © 2011 Blizzard Entertainment, Inc.

获得了令人惊叹的分享率。网络游戏以及 MMORPG 的代表作之一。

该图片是 2004 年发布时的截图。
http://www.worldofwarcragt.com/

收费模式和小额的计费认证系统也有所进步，收费金额是固定的，玩家越多收到的费用也就越高，这种模式进一步提高了收益。在这一背景之下，DB 产品的竞争加剧所引起的价格降低、机器性能提升，使得能够高速处理事务的高性能 DB 能以相对低廉的价格来构建。

另外，以任天堂的 Wii、Microsoft 的 Xbox 360、SCE 的 PlayStation 3（PS3） 为代表，2000 年代后期的游戏机搭载了通信性能达到兆级传输速度的 LAN（局域网）连接功能，在线连接率达到了 20%～40%。

由于要兼顾到游戏机商业收益模式（发行大量磁盘来出售），适应网络游戏的只有 P2P 类型的游戏。据说下一代的游戏机计划进一步加强在线功能，但在原稿撰写之际尚未明确。

在移动电话中，自从 NTT DoCoMo 的 i-appli 在线和 iPhone、Android 问世后，也很快出现了和 PC 游戏一样，具有高度自由的通信功能和 3D 显示功能的游戏，同样也在商业上获得了成功。

1.1.10　2010 年之后：究竟会出现怎么样的游戏呢？

2010 年之后，玩家互相交换制作好的游戏数据、在游戏机上也能体验到 MMOG、针对移动设备的网络游戏得以普及等都是目前的潮流，但是究竟会发展到何种地步，究竟又有怎样的技术会运用到游戏中，笔者自己也非常期待。

1.2　从技术变迁看游戏文化和经济圈

上一节我们粗略温故了网络游戏和相关技术的发展历程，这一节来看一下它们的发展趋势。

1.2.1　解读技术发展图

现在我们再来仔细看一下前文中的图 1.1。图 1.1 的纵轴自下而上划分为电子游戏的概念、计算机的发展、网络 /Web、操作系统 / 程序设计语言、通信 / 社交、用户创建和移动领域这几个技术分类，横轴则根据年代线性划分。该图展现了 1945 年至 2010 年之间的技术发展。让我们来思考一下能够从该图中看到些什么。

1.2.2　3 个圈（三大范畴）

在绘制这张图时，我们很自然地划分出了上下两大块，但是我们注意到在它们之间，从 20 世纪 90 年代后期开始增加了一些小块，它们代表了什么呢？

从图 1.10 中我们可以看到，这些块依次为：黑客文化圈、电视游戏 / 街机游戏商业圈和微软圈。

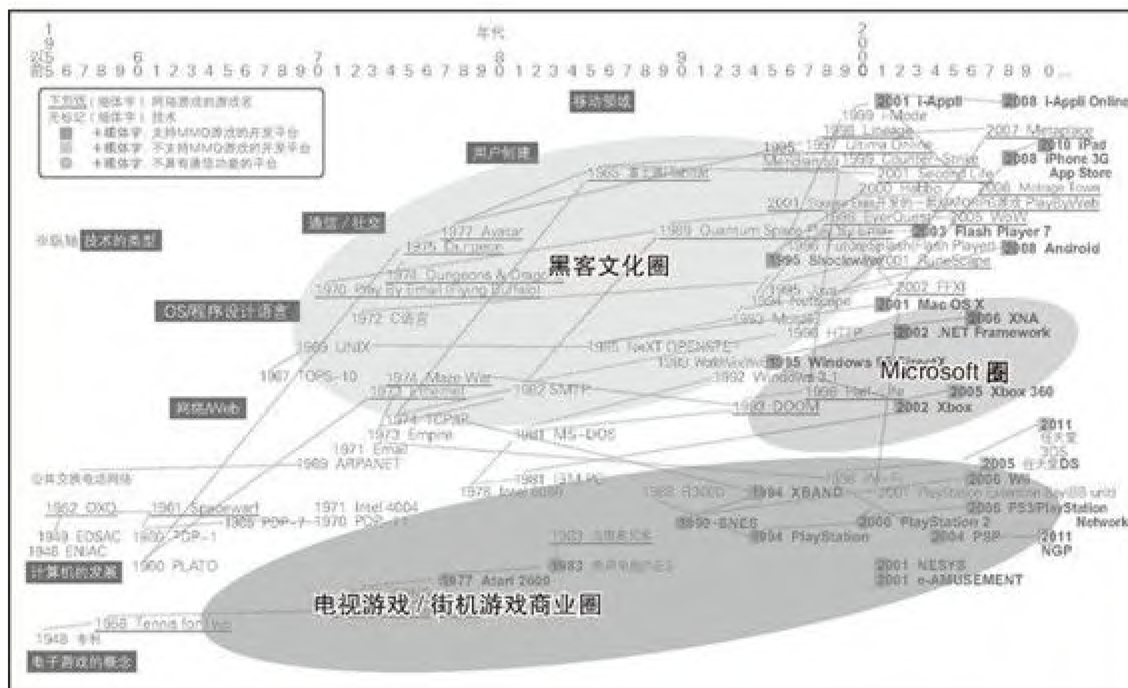
黑客文化圈

黑客文化圈指的是以美国西海岸为中心，由全美国的大学研究人员和学生们，以及利用互联网集中在一起的早期计算机怪才们组成的信息共享团体。要想了解黑客文化圈，可以阅读 Eric S. Raymond 所著的《大教堂与市集》[26] 一书。如果非要用一句话来形容他们的文化，那或许就是“知识公开，自由共享”。

> [26] 原版为 *The Cathedral and the Bazaar* 。参见 http://www.catb.org/~esr/writings/cathedral-bazaar/

他们中的很多人都在研究所和硅谷的风险投资企业中工作，处于技术革新的中心，一心致力于开发新产品。他们开发游戏的目的并不是为了赚钱，更多的是为了演示新的技术和吸引同行的注意。

图 1.10　3 个圈



电视游戏 / 街机游戏商业圈

电视游戏 / 街机游戏商业圈是美国雅达利公司、任天堂、索尼、微软等企业所形成的商业电子游戏经济圈。开发商业游戏的首要目的就是从游戏中获得利益。

在 Web 和数字传输还没有诞生的 20 世纪 70 年代，虽说是为了高效地将数 Mib[27] 的数据传送给几十万以上的终端而选择了使用 ROM 卡带，但这也强制规定了商业模式。也就是说，制作数万个 ROM 存放在仓库中，然后用卡车运送至各零售店销售。在这过程中所产生的平均每个 ROM 数千日元的流通成本必须由某一方来承担，负担流通成本的企业需要制定流通方案，进而还要决定制作什么样的游戏 [28] 。

> [27] 2 的 20 次方，Mega binary byte 的缩写。常用来表示 ROM 和 RAM（随机存储器）的容量。

> [28] 换言之，企业不能制作那些超过了成本而收益又无法预见的游戏。

微软圈

1995 年以后出现的是微软圈。由于 Windows 95 以及同时期出现的 DirectX 库的普及，IBM PC 搭配 Windows 操作系统作为一种非常适

合游戏开发的平台开始急速发展，电视游戏所存在的内容限制问题在这一平台上不复存在，由此诞生了以 Megademon[29] 和 Bio_100% 为代表的黑客大集团。

> [29] 指运用实时计算，并且将细致的 CG 和音乐结合起来进行画面描绘的视频作品和文化。

从 2000 年左右开始，微软积极着手研发能通过网络共享数据的开发环境（.NET），又将采用了 DirectX 的电视游戏机 Xbox 投入市场，在黑客文化圈与电视游戏商业圈之间构建了自己的文化圈。

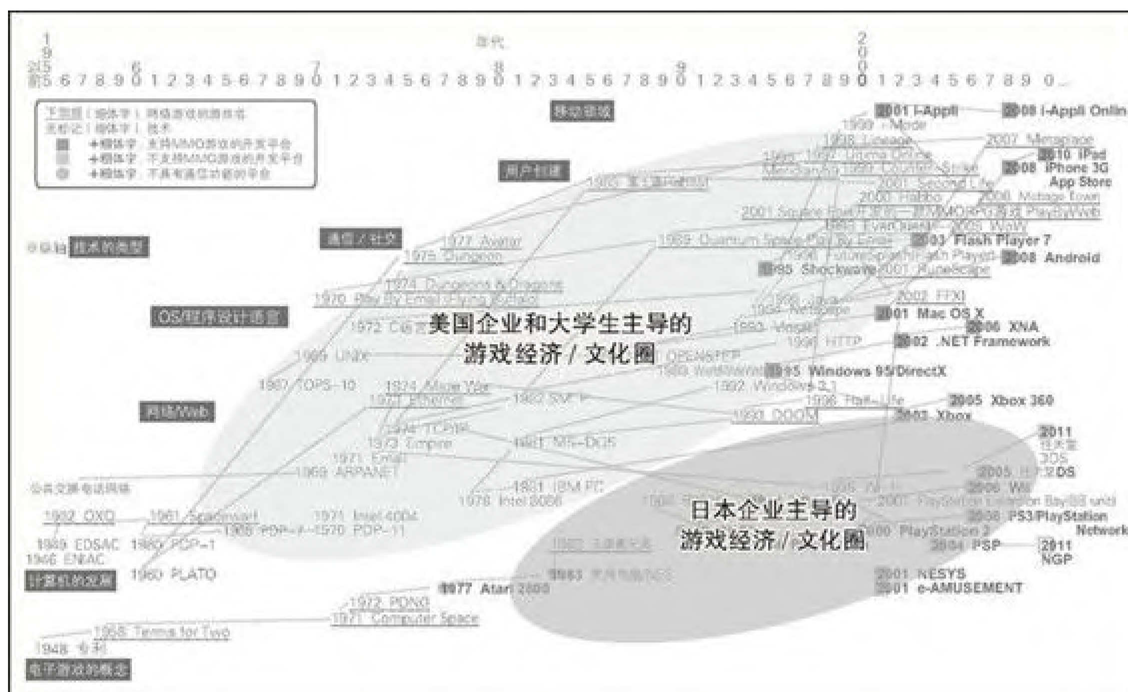
在微软有很多出生在美国、深受黑客文化影响的中老年工程师，据说他们至今仍在编写代码[30]。正是因为认可“自由共享知识”这种黑客文化，微软才会将使用 XNA（一种开发工具）开发的游戏在 Xbox 上发售。这虽然是采取了电视游戏商业圈的做法，但也在相当大程度上融合了黑客文化圈的思想。

> [30] 实际上，笔者也曾访问过美国的软件公司，年过 50 的工程师大有人在。与日本软件公司的氛围很不相同。

1.2.3　两个游戏经济 / 文化圈

对图 1.1 和图 1.10 进行更进一步的分析，还能得到什么样的结果呢？如图 1.11 所示，笔者进行了总结。为什么要这样划分呢？

图 1.11　两个游戏经济 / 文化圈



首先，20 世纪 80 年代前期，雅达利公司在电视游戏上的失败致使日本企业在这一领域独占鳌头。日企中主要从事游戏开发的 40 岁至 50 岁的开发者，虽然大多都在从事面向任天堂和索尼的软件开发，但是他们中的很多人都对美国的黑客文化圈所知甚少。

比如，在日本的游戏公司里，程序员很难积极地向其他程序员、开发团队和公司公开自己的知识。笔者有一位名叫清水亮的朋友，他担心日本的电子游戏行业会因为知识不公开而导致技术水平低下，于是策划了游戏开发者会议 CEDEC 。他也许是在美国微软工作时感受到黑客文化的。

虽然现在各个企业内部也开始盛行信息共享了，但是仍然还没有达到在不同企业之间共享信息的程度。笔者认为，虽说这是因为黑客文化没有在日本的游戏业界中扎根，但是能够体会到知识共享给自己带来的直接、短期好处的人还很少，这也是一个很大的原因。此外，还有因不擅长英语 [31] ，从而缺乏与参与开源活动的外国人一起工作的机会，很少能接触黑客文化等因素。通过比较英语 和日语 的维基条目就能知道，仅仅依靠日语知识并不足以孕育黑客文化。

[31] 在日本程序员中，理解黑客文化的人似乎很多都是擅长英语的。

另一方面，面向任天堂和索尼的游戏在美国也在积极开发。但是为了向朋友和网络上认识的人展示成果，就需要制作 ROM 卡带和磁盘，然而其成本非常高。而且，使用任天堂和索尼的开发工具又要严格遵守保密义务，开发工具完全是封闭的，与黑客社区的交流也非常少，黑客们被告知“禁止将研发成果在网络上公开”，因此他们不再留在那里。

此外，特别是 20 世纪 90 年代前期，在轻量级编程语言（LL）的处理体系、Web 浏览器、Linux 开发等方面，黑客集团的兴趣转向了开放源码上。与此同时，在游戏机方面，越发倾向在构成游戏的数据量和游戏质量上，而非在 3D 和图像上一决胜负，或许就因此与黑客团体产生了距离。从笔者个人的活动范围而言，到 20 世纪 90 年代前期为止的游戏机与当时流行的基于 MC68000[32] 的 WorkStation 相比，因为处理性能和操作系统的能力过于低下，吸引力不免有所不足。

[32] 美国 Motorola 开发的 MPU。

* * *

从以上这些文化和经济背景中可以看到，电子游戏的文化和经济始于美国，但是随着雅达利公司的落败以及日本和美国在游戏产业上的分离，网络游戏及相关技术经过 20 世纪 80 年代和 90 年代的各自发展，形成了现在的局面。

1.2.4　文化、经济与技术的关系

文化和经济会对支持网络游戏的技术造成什么影响？恐怕对此抱有疑问的读者不在少数。但事实上，它们极大地影响了网络游戏的技术。

此前的图 1.1 对支持网络的各个平台进行了归纳，详细内容会在本书后面的章节中加以介绍，这里只是简单说明一下。能够支持 MMO（C/S 通信的游戏）游戏开发的平台以粗体加上深灰色正方形表示，不支持 MMO 的平台（平台不允许）则以粗体加上浅灰色正方形来表示。另外，不具有通信功能的平台则以粗体加上圆形图标表示。

MMO 类型的游戏需要专用的服务器。游戏内容安装在服务器上，运行在客户端上的程序只是用来显示游戏内容。此外，服务器由游戏的开发者们而非平台支撑人员进行管理。服务器上设有密码，就连平台的所有者（NES 的任天堂、Windows 的微软、Flash 的 Adobe 等）也无法访问。在控制台游戏的商业模式中，为了降低 ROM 的流通风险，必须严格控制游戏内容，这一点非常难办。这些因素都集中在深灰色正方形的上方和浅灰色正方形的下方。

从图 1.1 中可以看到，日本最早具有通信功能的游戏是 2001 年上市的 [33] 。在 PC 领域，富士通 Habitat 在 1985 年就已支持网络了，1993 年的 *DOOM* 则实现了实时的局域网对战，但是在电视游戏机上网络功能的实现却晚了好几年。

> [33] 由于 XBAND 很遗憾未能得到普及而被排除在外。

技术是根据其使用情况而发展的。为了实现网络游戏而产生的技术首先是在美国发展起来的。截至 MMO 类型的游戏出现时，日本和美国相差了将近 30 年。图 1.1 中出现的具有革命性的网络游戏几乎都是美国制作的，本书所介绍的技术也几乎都是来自美国，个中缘由大家从以上介绍就能理解了。

1.3　小结

本章追溯了网络游戏相关技术从开始到现在的发展历史。虽然只是简单地进行了一番说明，但是请整理一下技术背景中的历史要素，试着以此把握网络游戏相关技术的发展趋势吧。

专栏　成为出色的网络游戏开发程序员的条件

网络游戏行业中，为了成为出色的游戏开发工程师（深受信赖、加薪快），需要具备怎样的条件呢？笔者咨询了相识的企业管理人员和技术人员，总结了如下几条建议。

喜欢游戏

对于自己非常喜欢的游戏类型，要非常投入地去玩其中的某款游戏，并且对一款网络游戏要有非常深入的了解。人的精力有限，不可能精通各种类型的游戏。但是有自己非常擅长的游戏类型和游戏并对其非常了解，这一点是非常重要的。

喜欢编程和实际工作

网络游戏的开发永远不会有终点。为了向用户提供持续服务，开发人员需要不断改进游戏，修复 bug，修正各类问题。如果不能享受这一过程，那就很难在项目中体会到乐趣。要出于兴趣而开始游戏开发。

＊ ＊ ＊

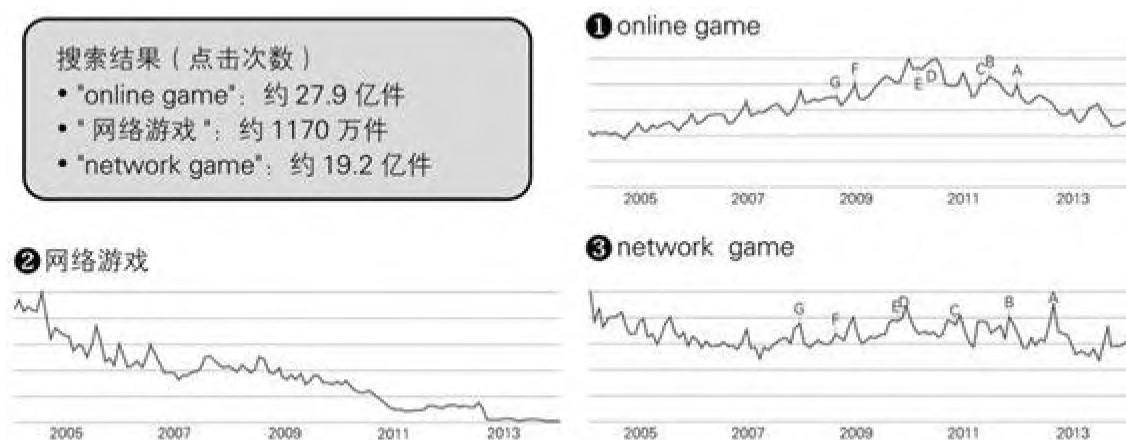
当再次回顾这些条件时，你是不是也觉得真的很简单呢？

第 2 章　何为网络游戏：网络游戏面面观

正如上一章所讲，网络游戏一直在不断地发展。“网络游戏”这个说法并非是一种专业用语，而是一种商业用语；更进一步说，这是一种通称，所以无法严格定义。但是网络游戏在发展过程中形成了现在的商业模式，对技术人员来说，这毫无疑问是一个很重要的技术领域。

首先，我们来看一下网络游戏的各种叫法。笔者根据从 Google 搜索的结果中整理出了 2.A 这张图以作参考。该图左上部分显示了各个关键字的点击数，❶～❸ 则是 Google 趋势的图表。

图 2.A　网络游戏与各种叫法



从图 2.A 中可以看到，全世界几乎都以“网络游戏”（Online Game）这种说法为准。此外，日本市场的网络游戏热潮正趋向于统一化，而全球市场上的则仍在发展。上一章已经讲过，日本在技术上处于特殊的状态中，但其实不仅仅在技术上是如此，在市场上同样也处于特殊地位。技术与市场的发展是不可分割的。本书的主题是“支持网络游戏的技术”，所以接着我们就主要从技术的角度来讨论网络游戏的相关内容。

本章就“何为网络游戏”为题，从一些基本术语的定义开始加以探讨。

2.1　网络游戏术语的定义

首先，我们思考“网络游戏”这一术语的定义。

网络游戏的 4 个层面

在查阅了各种定义之后，笔者认为日语版维基百科中オンラインゲーム（网络游戏）这一词条的定义最为恰当，下面引用了这一定义。顺带一提，该定义与英语版 Wikipedia 中的 Online game 基本相同。

> 网络游戏是指通过计算机网络，与专用服务器和用户的客户端设备（PC、游戏机等）相连，让多名玩家同时联机进行游戏的娱乐形式。
>
> ——引用自 http://ja.wikipedia.org/wiki/ オンラインゲーム（2010 年 9 月 24 日（星期五）09：05，日语版 wikipedia）

更为准确的表达方式如下。

> 网络游戏是指通过计算机网络，与专用服务器和用户的客户端设备（PC、游戏机等）相连，包含让多名玩家共同进行游戏的软件的服务。

维基百科的定义中提到了“同时进行游戏”，但是在网络中是不可能“同时”的（理由将在后面说明），所以这里去掉了。而“联机”与前面的“通过计算机网络”重复了，所以也省略了。另外，“娱乐形式”就是指“包含软件在内的服务”。

接下来，本章试着拆分一下该定义的要素，来说明本章的主题“何为网络游戏”。我们将以上定义分成 3 个部分。

❶ 通过计算机网络，与专用服务器和用户的客户端设备（PC、游戏机等）相连

❷ 能够共同进行游戏

❸ 包含软件在内的服务

❶ 定义了网络游戏的物理层面，❷ 定义了其概念层面。从技术观点来看，❶ 和 ❷ 是网络游戏的定义中不可或缺的要素。对于 ❸，本书的对象是支持商业网络游戏的技术，因此，这里 ❸ 所涉及的是网络游戏的商业层面。从商业观点来看，网络游戏包含成本和市场等要素。

除了上述 3 个要素，实现商业游戏还需要包含技术人员在内的各种人员和组织的参与，所以为了说明支持网络游戏的技术，人员和组织的这一方面也是不可或缺的。

根据以上这些定义和要素，为了让读者对支持网络游戏的技术有个整体把握，本章前半部分 2.2～2.5 节将分别从以下 4 个层面进行说明[1]。

> [1] 本章后半部分（2.6 节之后）将会讨论一些为第 3 章的内容做准备的面向开发人员的基础知识。

- **物理层面（2.2 节）**
- **概念层面（2.3 节）**
- **商业层面（2.4 节）**
- **人员和组织层面（2.5 节）**

2.2　网络游戏的物理层面

网络游戏的物理构成有哪些方面呢？现在，我们以网络、服务器和客户端设备为主，看一看网络游戏的物理层面。

2.2.1　物理构成要素

要实现网络游戏，计算机和网络是必不可少的。首先我们将这些物理构成要素拆分如下。

计算机网络

- 计算机
    - 客户端设备

      →各种 PC、台式游戏机、掌上游戏机、移动电话、PDA（Personal Digital Assistant，个人数字助理）等（就是前述定义中的客户端设备）

    - 服务器设备

      →放置在数据中心的服务器（就是前述定义中的服务器）

    - 负载均衡设备（Load Balancer）等
    - 网络设备（集线器、路由器、Wi-Fi 适配器）等
- 网络
    - 使用互联网协议进行通信的网络

      → WAN（Wide Area Network，广域网）（基本所有的网络游戏都是通过 WAN 实现的）

      → LAN 内部网 [一部分街机游戏和多人网络游戏（MOG）]

      → LAN 与专用网络连接的内部网络（大多数街机游戏）

  ◆ 使用局域网的物理网络进行通信的网络

  →使用红外线和 ad-hoc（点对点）模式[2] 的网络

  →使用 RS-232[3] 和 MIDI（Musical Instrument Digital Interface）、USB 电缆等的网络

> [2] 无线 LAN 通信的一种，终端机器直接进行通信的运行模式。

> [3] Recommended Standard 232。串行通信接口的一种。

以上这些基本上就是与网络游戏相关的计算机网络的物理构成要素。

2.2.2 物理模式 / 物理上的网络构成

上述的物理构成要素如图 2.1 所示。

图 2.1 中展示了以下 5 种物理模式、物理网络构成。

❶ 互联网直连模式

为 PC 直接分配全球 IP 地址，与其他终端直接进行数据包交换。与服务器和数据中心进行必要的通信。

❷ 移动网络模式

移动电话通过 NTT（日本电信电话株式会社）等移动电话网络公司运营的企业网关连接到网络中。通过 au（日本移动电话网络品牌）等服务，移动电话之间可以直接进行数据包的收发，但是有些情况下无法接收到除此之外的数据包。本模式在必要时，会与服务器和数据中心进行必要的通信。

❸ 路由模式

一般的 PC 和游戏机通过宽带路由连接至互联网。因为要通过 NAT（Network Address Translation，网络地址转换）访问互联网，所以通常无法直接从路由外部的终端上接收数据包 [4] 。本模式在必要时，会与服务器进行必要的通信。

> [4] 也有例外，比如 NAT 转换技术之一的 UDP Hole Punching 等。详细内容将在 5.6 节介绍。

❹ 街机模式

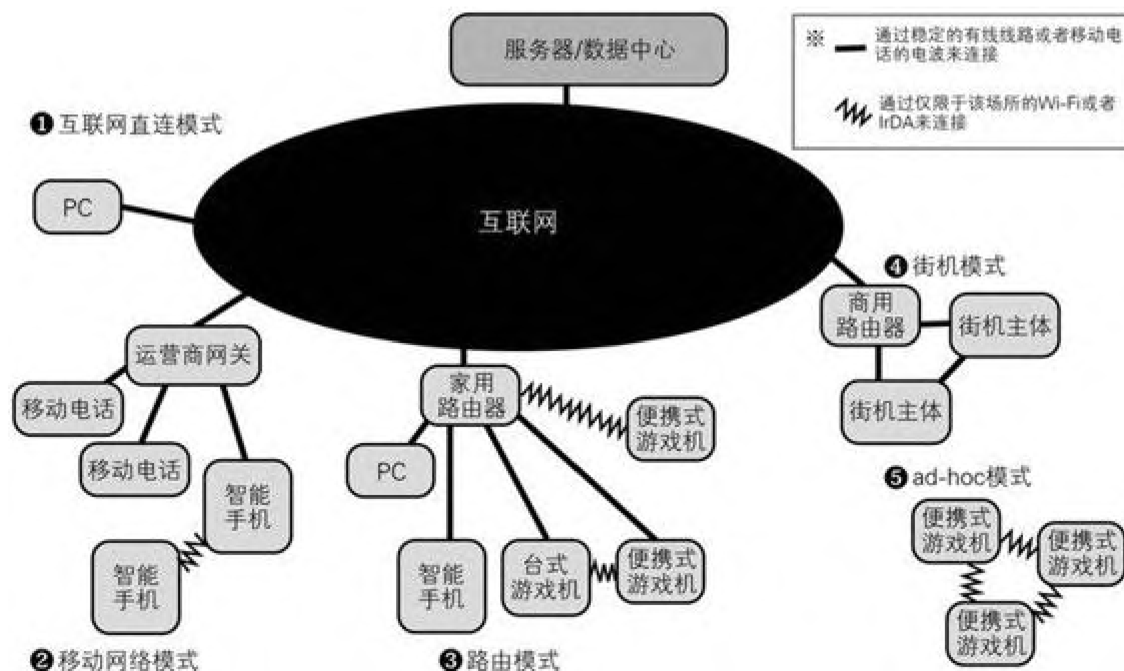
与路由模式相近，使用专用线路直接连接到数据中心。本模式从互联网与隔离的服务器和数据中心进行通信。

❺ ad-hoc 模式

PSP 与任天堂 DS、iPhone 与 Android 终端等智能手机的无线 LAN 功能使用 ad-hoc 模式，即使没有接入互联网，也能构建独立的小规

模网络，几个人共同进行对战游戏。由于没有连接到互联网，所以不能与服务器和数据中心连接。

图 2.1　网络游戏的物理构成要素和物理模式



以上 5 种模式网罗了与网络游戏相关的所有物理模式。因为互联网上存在很多怀有恶意的第三方攻击者，所以通常为了确保安全性，开发工作量也会增加，这 5 种模式的开发工作量按如下顺序递增：❺ad-hoc 模式→❹街机模式→❷移动电话模式→❸路由模式→❶互联网直接连接模式。

本书中的物理线路是基于互联网的

在此，我们从使用者的角度出发，将网络环境分成了 5 种模式。如果从技术角度来看，可以按照网络速度的快慢来进一步分类。如果使用慢线路，游戏策划和程序员需要做很多工作。

上述 5 种模式中，❶❷❸使用互联网这种较慢的线路。❹因为使用了专用线路，所以延迟较小[5]。❺是本地通信，所以通信延迟比互联网低得多。

[5] 有一点需要注意：街机游戏中，有些店家也会使用互联网，这种情况下相当于❸这种模式。

本书针对的是使用互联网这种较慢的网络来进行游戏开发，所以将以❶❷❸这 3 种模式为主，详细介绍典型的 C/S 架构和 P2P 架构的相关内容。由于比起前 3 种模式的网络环境，❹❺这两种模式的网络延迟更低，所以用于前 3 种模式的实现技术也同样适用于❹❺模式。

本书所介绍的 C/S 架构和 P2P 架构的技术可以使用❶❷❸以及❹中的任何一种模式，但是❺的 ad-hoc 模式主要用在 P2P 架构中。

2.3　网络游戏的概念层面

从 2.1 节的定义中所知的概念层面，也就是网络游戏中“能够共同进行游戏”，这一概念并不普通。我们首先从游戏的基本概念开始说明。

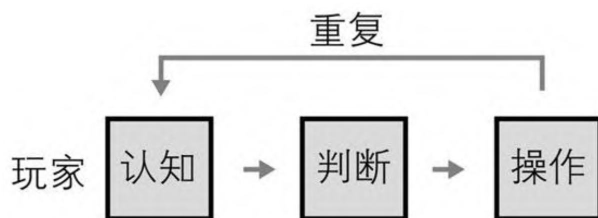
2.3.1　网络游戏及其基本结构

为了了解网络游戏的概念层面，首先需要理解游戏的基本结构。

游戏的基础 ——认知、判断、操作的重复

在进行游戏时，玩家就是在重复着认知→判断→操作这一过程。这并不仅限于电子游戏，也包括离线游戏和体育游戏等（参见图 2.2）。

图 2.2　游戏的基础



这里的“认知”对象就如同棋盘上象棋子的当前位置，对其有所“认知”（看到）后，使用自己的原有知识加以“判断”，进而基于这一

判断“操作”棋盘上的棋子。然后不断重复这一过程。

在象棋中平均每一步要花几分钟时间，所以每次的重复过程将会持续几分钟到十几分钟。而足球和保龄球这种体育游戏，可能需要在几秒或者 1 秒内重复多次。

电子游戏的结构

在进行电子游戏时，软件的运行结果以画面显示和声音的方式输出，这就是人们的“认知”过程。此外，人们使用鼠标、键盘、游戏手柄、触摸板等输入接口进行操作（图 2.3）。

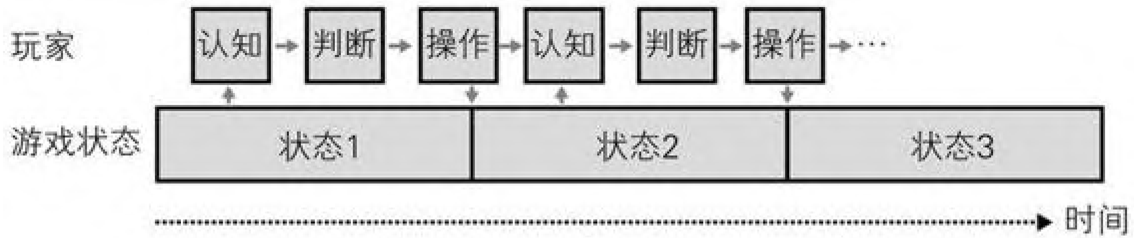
图 2.3　电子游戏的结构

比如《超级马里奥兄弟》和《俄罗斯方块》，它们与足球、保龄球一样，1 秒内需要重复多次认知→判断→操作的过程。这类游戏称为“动作游戏”。与此相对，像扫雷和接龙这种在玩家进行操作之前始终保持原本状态的游戏也有很多（一般的游戏类型不是以此来分类的）。

2.3.2　游戏进行空间——进行游戏时所需的所有信息

根据玩家的“操作”，游戏软件的状态（内存状态）会发生变化（参见图 2.4）。

图 2.4　游戏空间（游戏状态）



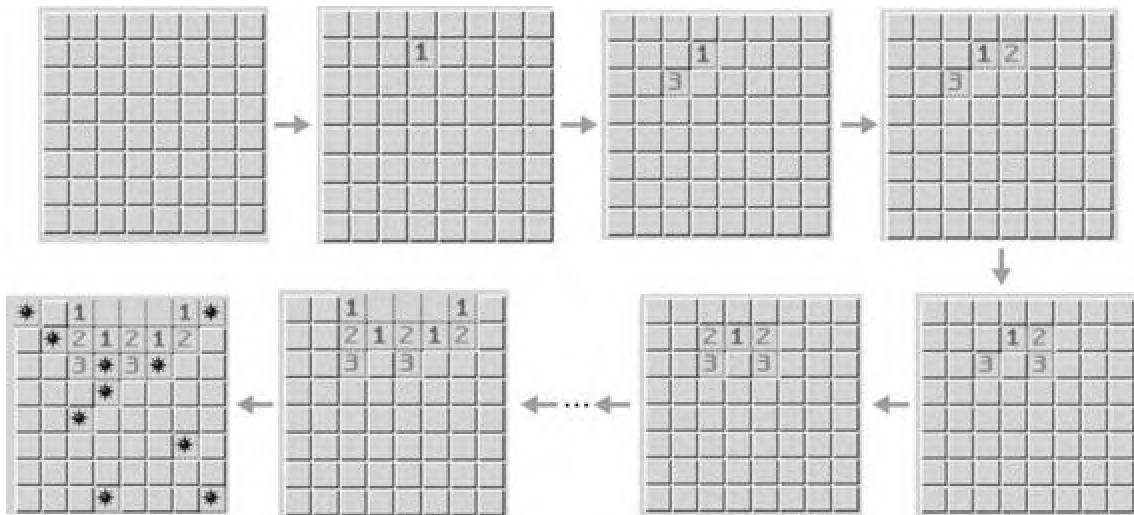
游戏程序员将游戏软件的内存状态称为“游戏进行空间”，或者“游戏状态”，或者就叫“状态”、“游戏空间”、“空间”，等等。

游戏程序员在谈到游戏时，通常会提及“游戏所需的所有信息”，所以各个开发团队常会为其附加一个名称。笔者多使用“游戏状态”或“游戏进行空间”这种叫法。“游戏状态”是一般的术语组合，为了便于区分，本书采用“游戏进行空间”这种说法。

2.3.3　游戏的进展——游戏进行空间的变化

在此，我们以众所周知的扫雷游戏为例进行说明。扫雷游戏根据玩家的“操作”情况，游戏进行空间的变化如图 2.5 所示。

图 2.5　游戏进行空间的变化（以扫雷游戏为例）



对于扫雷游戏的具体情况就不作详细说明了，它的最终目的就是将所有的地雷从雷区中拆除。没有放置地雷的区域都可以点击并且清除，

如果点到了放有地雷的格子就算输了。

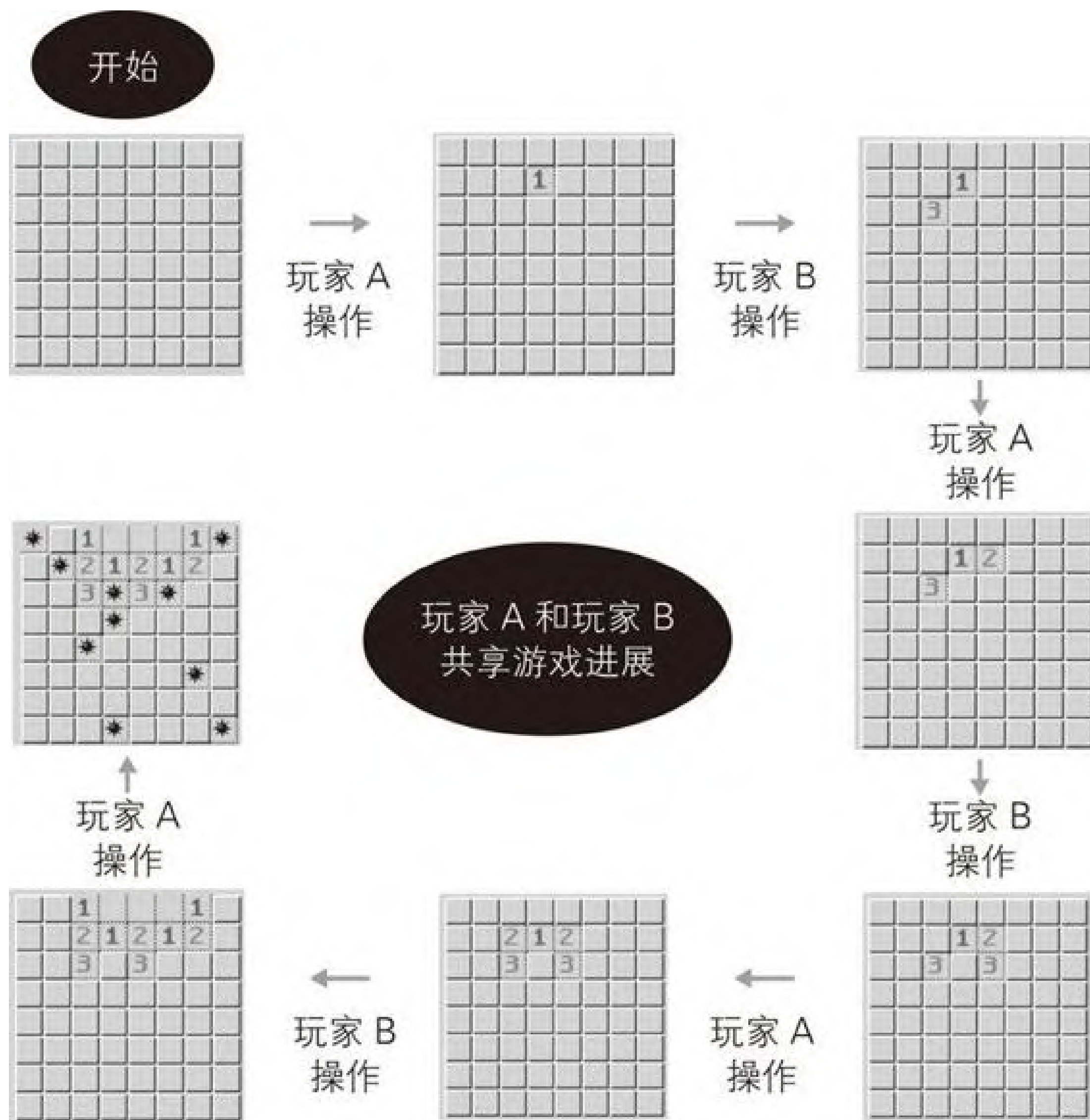
图 2.5 的例子中，扫雷区域以 8×8 的格子来表示。开始时，游戏进行空间全部隐藏着，随着不断地点击其中的格子，游戏空间的内容则会逐渐显示出来。图 2.5 中，到了第 10 手时点到了地雷，于是游戏结束。

在这个例子中，游戏进行空间一共变化了 9 次。这 9 次的游戏过程称为“游戏的进展”。网络游戏中这些（相同的游戏进展）是在线共享的。那么接下来，我们就来思考一下“共享”。

2.3.4　共享相同的游戏进展

图 2.5 所示的扫雷游戏由一个人进行，要由两个人交替进行游戏也很容易。如图 2.6 所示，一个游戏进行空间从产生到结束的游戏进展由两个人共享。

图 2.6　游戏进展的共享（2 人）



这在网络游戏中很容易进行概念上的定义。我们不难想象，一系列的游戏进展很可能是由使用移动电话的美国玩家 A 和使用 PC 的日本玩家 B 来进行的。

可以进行共享

有一点要注意，“可以共享”并不意味着“必须共享”。很多网络游戏都有仅供 1 人游戏的选项，所以“可以共享”的游戏就是网络游戏。

* * *

接下来，本书以国际通用的名称“网络游戏”来称呼基于物理层面、概念层面定义的软件。

2.4 网络游戏的商业层面

商业游戏中，基于市场和成本的商业观点备受重视，接下来我们就来看一下网络游戏的商业层面。

2.4.1 商业层面的要求

向网络游戏工程师提出要求的一定是游戏公司的经营者和策划人员。他们充分掌握了网络游戏的目标用户及其市场的动向，他们是出于让游戏获得商业成功而提出要求的。他们绝对会提出以下这样的要求。

- 有趣
  - 尽早开始开发迭代
  - 有效地招募测试玩家 *
- 尽早完成
  - 尽早开发出原型 *
- 降低开发成本
  - 尽早测试
- 不断更新 *
- 降低运营成本
  - 减少服务器数量 *
  - 节约带宽 *
  - 从小规模开始 *

- **运营时间久**
    - 具有可扩展性 *
- **评价高**
    - 提供多种收费选择 *
- **希望有大量玩家参与**
    - 适配多个平台
    - 为多个国家提供服务
- **不希望给玩家添麻烦**
    - 低成本、尽早地、可靠地消除攻击者 *
    - 尽量减少服务器停止的情况 *
    - bug 要尽可能少
- **更佳的用户体验**
    - 频繁进行各类活动 #
    - 反馈游戏结果 *
    - 促进玩家之间的交流 #
    - 能够更容易地与其他玩家相遇 *
    - 能够长时间游戏（不会觉得厌烦）

以上都涉及权衡成本和开发时间，如何权衡就是网络游戏开发人员大显身手的地方了。

上述条目中标有 *、# 的，表示这是“网络游戏所特有的”。与《超级马里奥兄弟》这种非网络游戏相比，所需的知识大有不同。在游戏

业界，对于这种差异，有种说法是“网络游戏是服务业，单机游戏是制造业”。在笔者看来，实际上，“网络游戏兼顾了制造业和服务业”。

我们从上述要求中挑出标有＊符号的条目说明一下它们的背景。此外，第 2 个标有＊的条目“尽早开发原型”将在第 4 章中详细讨论，请参考该章的相关内容。而标有＃的“频繁进行各类活动”和“促进玩家之间的交流”脱离了技术范畴，所以本书不作讨论。

2.4.2　有效地招募测试玩家——网络游戏与测试

与内容固定的电影不同，游戏的一大特点就是根据玩家的操作会产生不同的结果。在很多游戏中，使用者和开发者都会碰到没有预想到的情况和游戏结果。

对于单机游戏，在即将发售之前会招募数十至数百名测试人员进行为期 1～6 个月的短期集中测试，尽可能地以各种方式玩游戏，对有问题的地方进行修改。在大规模的 RPG 中，测试人数通常达到 200 名，修正的问题数在 1000～5000 以上。

而对于网络游戏，由于玩家全部共享游戏状态，所以与单机游戏相比，一些小 bug 和策划上的缺陷等无法事先预测的重大问题会逐渐浮现出来。这里所说的缺陷大体可以分为：

❶ 策划上的意图本来就无法再现

❷ 策划上的意图可以再现，但是缺乏可玩性和价值

在进行了 1 到 2 次公开测试后，问题❶基本都能发现（发现是能发现，但是能不能解决就是另一回事了）。对于问题❷，在进行公测时可以发现如下这些问题（同样地，能否解决也是另外一回事）：

- 明显不佳的游戏平衡性
- 谁都觉得没意思的游戏内容
- 明显引起不满的内容

如果问题非常严重，在测试之后会设立“改进阶段”，在正式版发行时可以修改。因为可以通过这样的公测来确保游戏的可玩性，所以希望尽可能进行多阶段的公开测试。

封闭测试（CBT）

网络游戏除了上述问题，网络游戏还有一些问题只有在很多人经互联网同时且长时间进行游戏时才会出现，这种情况是常有的。比如，有些问题只有在使用某些特定 ISP 的用户在某个时间段进行游戏时才会发生。于是，企业就招募 500～3000 名测试人员在 2 个月～半年以上的时间内，进行总共几万分钟的游戏试玩。这种封闭测试的形式就是在一段期限内招募一定数量的测试人员。业界将其简称为 CBT。

可供数千以上玩家同时游戏的网络游戏，在收取费用之前，必须对此进行充分测试。对于网络游戏，历史上有种说法：“一经发行就遭遇失败的游戏是无法复兴的。”（很严酷呢。）从过去 10 年的短暂历史来看，确实如此。在进行 CBT 的过程中，如何解决严重的 bug 对游戏的正式运营是非常重要的。

公开测试（OBT）

在封闭测试解决了游戏平衡性和服务器负载等基本问题后，就要开始实施不限定人数的公开测试了。

出于宣传目的，公开测试一般都会免费试玩，营销色彩浓重。由于公开测试是不限制参与人数的，所以系统负载实际上与正式服务是同等的。工程师们的所有工作在此需要告一段落。因此，若开发计划中设有 OBT 阶段，开发工程师就会有一种意识：开发工作至此完成。

＊ ＊ ＊

有关网络游戏的开发计划的制定方法，请参阅 4.1 节“网络游戏开发的基本流程”。

2.4.3　不断更新——网络游戏的运营和更新

如前所述，网络游戏兼有“制造业”和“服务业”两者的性质。我们常说长盛不衰的 Web 服务的法则是“不断更新”，网络游戏的运营也

是如此。

网络游戏的运营中，大致有 3 种更新模式。以实施机会的多寡顺序排列如下：

❶ 定期的补丁

❷ 大型补丁（扩展包、追加包）

❸ 紧急维护

❶ 定期的补丁

首先我们来看一下定期的补丁（常规补丁）。很多网络游戏会进行几周一次甚至几百次以上的 bug 修复、细微的平衡性调整、系统改进等各种修正。可以在以下网页查看被誉为拥有世界顶级运营机制的 MMORPG 游戏《魔兽世界》的各补丁内容。

http://www.worldofwarcraft.com/patchnotes/

《魔兽世界》发布补丁的间隔一般在 2～4 个月。其他游戏也会同样时常追加更新内容。大型游戏一般会将补丁先行发布在专用的“测试服务器”上，让广大用户登录试玩，以此发现问题并修正。

❷ 大型补丁（扩展包、追加包）

接着，我们来看一下半年至两年一次的大型补丁。比如，日本的 MMORPG 代表游戏《最终幻想 9》（*FFXI* ），过去平均两年发布一次扩展包，总共发售了 4 个扩展包。

扩展包并不仅仅追加游戏内容，还包括服务器和数据库的游戏软件更新，通常，在这些开发工作中可以继续对系统的整体内容进行修正。扩展包和追加包这类更新称为大型补丁，追加与之前有极大差别的游戏系统，同时引入新玩法。有时开发大型补丁与开发游戏续作规模同样大。大型补丁也同样需要准备测试服务器，但是因为新的游戏内容不能向广大玩家公开（为了避免泄露游戏材料），所以大多都会严格限制人数。

❸ 紧急维护

最后，紧急维护是在发布了定期补丁和大型补丁后，普通玩家开始玩游戏时，发现了事前没能测出的 bug，对此发布的紧急修复版本。就是《最终幻想 9》和《魔兽世界 这种位于游戏界顶峰的游戏也无法避免紧急维护。

* * *

对程序员来说，如何应对网络游戏的“不断更新”非常重要。首先，必须同时进行“大型补丁”的开发和每日的小规模 bug 修复。可以采取很多种措施，比如，分别设置问题修复团队和新内容开发团队[6]、尽可能分离源代码和数据内容、对软件各功能进行模块化，从而保证对软件某一模块进行的修改不会影响到其他模块，等等。在软件的开发过程中，应该尽量落实这类基本的解决措施。

> [6] 但是在实际开发中，这并不怎么有效，现实中没有做到这一点的很多。

有关网络游戏更新的方法将在 6.3.3 节的介绍。

2.4.4　节约服务器数量和带宽——网络游戏开支的特殊性

网络游戏的商业利润是以“销售额－开支”来计算的。销售额（收入）暂且不提，开支（成本）中，❶人力成本、❷设备成本、❸宣传费用占据了很大一部分。

网络游戏与单机游戏的差异主要是❶和❷这两点，所以这里将对这两点进行说明。

❶人力成本和❷设备成本 ——运营、发售后花费的成本很大

首先，由于游戏发售之后必须进行“运营”，所以经常需要等同开发时甚至更多的❶人力成本，这一点是网络游戏特有的。而且，这一费用随着游戏发售后用户数的增加会变得越发高昂。

接着，在游戏发售后也要继续支出❷服务器成本和线路设备成本，这一点与单机游戏也不同。开始运营或者说游戏发售之后所需支付的这些成本因素是游戏公司远离网络游戏开发的原因之一。

设备成本中占据最多的就是服务器成本和线路成本。这两者在设计游戏阶段就需要加以考虑。游戏设计以及游戏类型方面的判断都与这些成本相关。这些也都是网络游戏的策划人员必备的知识。

服务器成本的预估 ——估算服务器损坏的概率

服务器的损坏概率曲线具有先平缓、两年以后急剧增加的性质。两年之内基本不会损坏，而之后损坏的概率为 5%，3 年之后就将达到 10% 以上。

主要易损的是发热风扇和 HDD[7] 等高速旋转的部分。网络游戏的运营通常会持续 5 年以上，因此对服务器损坏概率进行预算是很有必要的。大致上说，每增加 1 台服务器，每月就要花费数万日元（约 2 万至 3 万元）。这里“每月”是关键所在。

> [7] 有关 HDD 故障的内容可以参考论文“Failure Trends in a Large Disk Drive Population”（Eduardo Pinheiro/Wolf-Dietrich Weber/Luiz Andre Barroso 著）。http://labs.google.com/papers/disk_failures.pdf （PDF 文件）

线路成本的预估 ——尽可能节省带宽

服务器带宽的大致费用为 1Mbit/s ＝ 1 万日元 / 月，一般不会相差很多。这里，结合大量交易所带来的折扣和对等互连 [8] 等，尽可能降低成本，就要靠基础设施负责人了。

> [8] Peering，指互相连接到互联网，交换流量。

网络游戏开发工程师与基础设施负责人及策划负责人商讨时，常会受到“这个游戏内容会使带宽过高，成本就……不，如果这么来处理数据包……”这类问题的困扰。这里成本的关键也是“每月”。

＊ ＊ ＊

有关网络游戏基础设施的成本预估以及降低该成本的方法将在第 7 章中详细讨论。

2.4.5　从小规模开始，确保可扩展性——将风险降到最低，不要错过取胜的机会

无论是怎样的游戏策划者，都会认为当前正在开发的游戏一定会大获成功。但是到底能否获得成功，在实际运营之前，很多方面都是不确定的。即使在募集了 1000 人的封闭测试中大获好评，但是开始正式服务后，玩家人数却只在 1 万左右徘徊，虽然很可惜，但这种情况也不在少数。

虽然服务器设备和线路的成本会根据游戏内容的不同而有所差异，但一般来说与用户数成正比。比如，1000 个用户可能只需要 2 台服务器，而 10 万个用户就需要 100 倍，也就是 200 台服务器了。但是如果一开始就假设有 10 万个用户并以此来架设服务器，就太过浪费了。

理想情况是从 2 台服务器开始，用户增加后逐渐增加，但是根据服务器软件的设计情况，现实中不会这么简单。服务器软件的扩展性目前也仍在发展中。

* * *

这里的可扩展性问题根据网络游戏类型的不同，处理方法有所差异，对策划、开发也很重要，这些内容将在后续章节中进一步探讨。

2.4.6　提供多种收费方式——收费结算方式的变化

商业网络游戏服务刚刚问世时，能够使用的收费方式有：游戏包售卖、信用卡、专门的预付卡、银行汇款等。此外，收费单位以月计，向所有的用户收取固定的费用。

之后，又出现了 WebMoney[9] 等“电子货币支付”[10]，以及“便利店支付”[11] 和“移动电话支付”等支付方式。

[9] 预付费类型的电子钱包。http://www.webmoney.jp/

[10] WebMoney（简称 WM）是由成立于 1998 年的 WebMoney Transfer Techology 公司开发的一种在线电子商务支付系统。——译者注

[11] 在日本，便利店可以代收各种费用，网上购物后如果选择这种方式，会提示你便利店付款的支付码，到指定便利店支付。——译者注

收费单位也逐渐向细分化和实时化发展。细分化指的是可以以更小的单位来购买游戏服务，而实时化则是指不妨碍游戏的进行，在判断了用户购买游戏服务（比如点击了“购买”按钮）的瞬间就完成结算。

比如，在中国的网络游戏中，并非按照每月 1000 日元的方式来付费的，而是 1 分钟支付 1 日元。此外，还有一次游戏支付 10 日元，或者是花费 50 日元购买游戏内的一件道具等称为“道具收费”的方式，这在日本和亚洲其他国家、欧美很普及。

使用信用卡和预付卡进行交易存在❶手续费、❷响应时间两个问题。

首先来看一下❶手续费问题，由于每次付费都要支付一定的手续费，所以细分程度具有一定限制。如果是 100 日元、200 日元这种小额付费，手续费反而占了一大部分，这就不划算了。

接着看❷响应时间的问题，使用信用卡来进行结算需要花费几秒至十几秒。这是由于信用卡支付的系统负载过高所造成的。过于频繁的交易会使信用卡系统不堪重负，因此设定了一定的等待时间。这段时间无疑会中断良好的游戏节奏。

游戏点数的出现 ——实现了收费的细分化、实时化

为了解决这些问题，2003 年开始盛行将货币一次性转换为游戏点数的方式，使用针对这些点数单独开发的系统来管理、使用和购买游戏服务。

为每个游戏准备特别的专用系统，可以使收费单位更细化、响应时间更高速。由此，2006 年实现了 10 日元和 50 日元这种小额即时结算方式，此外还出现了“游戏内部的道具购买商城”，也就是能够在不影响连贯的游戏体验的情况下进行收费。比如，在游戏《勇者斗恶龙》的道具商城中，终于可以花费实际的 10 日元（与此相等的价值）来购买药草了。为了实现这一需求，必须以 0.1～0.5 秒的极高

速度完成结算，使用游戏点数系统可以比使用信用卡快得多，交易频率也更高，另外还能并行执行。

这里的“并行”并非是“点击确认购买的按钮→结算界面→调用外部系统的结算 API→接受结果→交易完成界面”这种以前的购买流程，而是“点击确认购买的按钮→交易完成界面→收费 API 的处理将在之后进行”。通过单独管理游戏点数，余额信息可以保存在游戏服务器中，这样就能够实现并行了。

游戏点数与现实中的货币具有同等的价值，所以对其进行管理的服务器系统必须具备极高的安全性、可用性、容错性，以及确保数据不会丢失的可靠性，这是最重要的。

游戏点数系统目前仍未普及，各大公司都在努力实现。今后，随着点数管理系统性能的提高，估计可以向 0.1 日元这种更细化的收费单位和更加实时化的方向发展，如果可以实现新的收费系统，或许就能为此开发出全新的游戏。

* * *

有关收费系统的内容将在 6.5 节中详细讨论。

2.4.7 低价、快速地根除攻击者——攻击、非法行为及其对策

针对网络游戏服务的攻击，更确切地说是滥用（abuse，非法的使用方式、非法行为），可以分为商业目的的攻击和除此之外的攻击。事实上，它们是互相关联的，但是我们先分开来看。

商业目的的攻击

首先，商业目的的攻击中有一种行为称为 RMT（Real Money Trade，真实货币交易）。RMT 指的是，用真实的货币来交易需要花时间培养的道具和角色。虽然人们强烈认为这种做法是很不公平的，但毕竟这还算是一种合法的行为，至于摒除 RMT 到底是好是坏还存在诸多争议，包括政治上的争议。另外，对企业来说，这是否与利益紧密相关还有待探讨。由于这与技术无关，所以在此不讨论 RMT 的是非。

关系到网络游戏的技术人员的问题是：以 RMT 为目的、称为 Gold Farmer（打金者[12]）的人实现了一种称为 bot（外挂，也叫做宏）的特殊工具，可以自动进行游戏并以此敛财。

> [12] 意指那些利用赚取虚拟货币然后通过各种手段把其兑换为现实货币的人。——译者注

bot 是一种以人类无法达到的速度反复进行游戏的软件，这种速度是人类可达速度的几倍甚至几百倍以上。使用 bot 会在服务器和数据库中产生大量流量，因此很有必要对此进行检测并且进行适当的处理。执行自动测试的测试 bot 也是其中一种。

Gold Farmer 是使用这种 bot 来赚钱的职业人员。就算不使用 bot，也有利用系统中的缺陷（道具的卖出价比买入价更高等），来赚钱的行为。

被评为 AAA[13] 的具有大规模运营体制的游戏也设立了专门的对策小组，为了杜绝这些攻击，由专人在游戏中进行巡视。比如日本 FFXI 的运营团队为了管制来自中国的GoldFarmer，成立了专门的小组，成功地大幅消减了 Gold Farmer 的人数。

> [13] 三 A。慎重起见，这里补充说一下。这并非正式机构所认定的级别，而是在玩家社区中广为使用的术语，指那些高品质、投入充足资金的大规模游戏系列。请参考：http://www.gameproducer.net/2006/05/26/what-are-aaa-titles/

对于那些占据了游戏市场大半江山的中小型游戏，要经常人为打击 Gold Farmer 是很难做到的，因此需要寻求一些自动的处理方法。目前在业界中还没有防止这一问题的常规技术，各个公司都是使用脚本语言编写一些简单的工具来个别处理。日志的格式化与处理方式等也尚未确定，事实上，即使是大企业也还无法做到有效地利用日志。要求开发更加易于使用的日志分析工具的呼声很高，大家都希望技术获得进一步发展。

非商业目的的攻击 ——各种攻击方式，3D 网络游戏专用的客户端

接着来看一下非商业目的的攻击，这是出于个人兴趣和某些目的而进行的攻击。大致可以分为以下这些攻击手段：

❶非法侵入服务器 / 篡改数据

❷根据程序和对数据包的逆向工程盗取数据

❸拒绝服务攻击等

为了应对这些攻击，必须采取一些 Web 服务中常用的安全措施。

- 专用的游戏客户端

3D 网络游戏并不使用 Web 浏览器，而是通常开发专用的客户端 [14] 来访问服务器，在这种情况下，以 Web 为例，通过实现类似于具有特定目的的 Web 浏览器的客户端，在 Web 下的浏览器处理部分 [15] 和用协议格式之类的 Web 标准来定义的部分必须要由自己来实现。需要自己实现的部分很多，安全性方面的工作也随之增加了。

> [14] 有关“游戏客户端”的内容详见 3.3 节的专栏。

> [15] SSL（Secure Socket Layer）和网络访问之上的域名限制等。

目前，使用用于网络游戏开发的中间件可以避免以上方面的重新开发，这种方式日渐发展起来，但是攻击方的技术水平也在不断提高。

* * *

第 7 章将会对一些常用的管理工具的功能进行介绍，此外在 3.2 节的“安全性”（有关作弊的说明）中，将详细介绍如何才能满足必要的安全性要求。

2.4.8　减少服务器停止的次数和时间——不要让玩家失望

网络游戏中有一种说法：“服务器停止一次，就会流失 1% 的玩家！”可见服务器停止给商业运营造成的影响颇大。服务器的停止往往会造成“虽然很想玩游戏，但偏偏不能玩”的现象。

现实中服务器的停止有两种情况：❶计划中的例行维护造成的停止，❷故障或攻击所造成的停止。不管是哪种情况，都会让玩家产生“想玩不能玩”的感觉，随之而来的就是流失 1% 的玩家。这里我们将其

称为“玩家感到失望”。对于大多数玩家来说，游戏并非生活必需品，所以一旦感到失望就可能不再玩下去了。

❶ 计划中的例行维护所造成的服务器停止

计划中的例行维护所造成的服务器停止出于数据库的备份和碎片清理、bug 修复等目的，是不可缺少的一个环节。在运营级别很高的游戏中，为了防止玩家出现失望心理，通常提出“一定会在星期二下午停止服务”的明确规定，然后确实在这个时间段内关闭服务器，利用 Web 等系统，通俗易懂地传达服务器系统的运行状态。这时就需要技术人员一展拳脚了。为了在这段明确规定的时间内完成数据库的操作等工作，必须装备高质量的工作工具。

“每周停止一次，一次停止几个小时”作为业界标准已经形成了一种常识，所以在这个范围内停止服务器就不会令玩家太过失望了。

❷ 故障或攻击所造成的服务器停止

另一方面，故障或攻击所造成的服务器停止就确确实实会令玩家失望。这里所说的故障和攻击可以细分为：a. 游戏平衡性和游戏内经济方面的缺陷；b. 服务器崩溃；c. 超负荷。

- **a. 游戏平衡性和游戏内经济方面的缺陷**

  首先，游戏平衡性和游戏内经济方面的问题是由于游戏制作方的设置失误、计算失误以及 bug 所引起的。举个典型的例子，“从游戏系统中能以 10 个游戏点数无限购买的物品，还可以无限制地以 15 个游戏点数出售”，这样每次可以赚取 5 点的差额，而且还可以无限重复，这就造成了经济崩溃（通货膨胀）。

  如果这种做法在运行中的服务器上加以利用，那么几小时后就会有人开发出能够自动进行这一重复操作的工具（初级的是按键重复器（keyboard repeater），还有高性能的专门的程序），经过一段时间就会在网上共享，1 天之后或许就会被尝试数千万次至数亿次，以致造成几十亿游戏点数的损失。

  对于现有玩家来说，这种类型的通货膨胀会让人觉得“好不容易积攒下来的资产贬值了”，从而会非常失望。运营方一旦发现了

这种情况，首先要做的就是立刻修复 bug。比如将金额设置中 15 点的出售价与 10 点的买入价进行交换[16]。然后确认此次 bug 修复会不会导致其他 bug 发生，如果确认没有问题，就宣布紧急停止服务器，停止时刻一到就关闭服务器，然后将数据库恢复到利用 bug 之前的状态，并且对非法利用该 bug 的玩家进行封号，完成了以上这些操作后再重启服务器。

精湛的运营团队可以在提供服务的同时并行处理这些问题，但这当然需要事先准备好必要的工具才能做到。

- b. **服务器崩溃**

  接下来，我们来看一下服务器崩溃的情况。服务器是人制作出来的软件，当然包含了一些 bug，所以不可避免的会发生一些异常。

  即使发生了异常，也不会导致所有服务器同时停止运行，应该根据游戏的类型做好充分准备，尽可能降低用户的失望度，并设法实现服务器自动重启的机制。

- c. **超负荷**

> [16] 为了避免这样的失误，必须准备一些测试工具。

超负荷所引起的服务器停止，并非所有的设备都一下子陷入了超负荷状态，而是由于前端服务器的负荷、后台服务器的负荷、设备负荷、带宽负荷、网络聊天室和消息等各辅助系统的负荷等导致超负荷，从而使得一部分服务器突然停止运行，响应变得异常缓慢。

针对这种情况，需要采取如下措施：设计时要做到“一部分设备的超负荷不会影响整体的运行”；从一开始就要设法减轻负荷；在发生超负荷时通知系统管理员和用户，尽快解决。

* * *

根据游戏类型的不同，以上这些防止服务器停止的方法差别很大，详细内容将会从第 3 章网络游戏开始讨论[17]。

> [17] 有关负载的内容在 7.1.3 节和第 0 章有关性能方面的讨论中介绍，想进一步了解的读者请参阅这两部分。

2.4.9　反馈游戏结果——日志分析和结果的可视化

本章开头讲过，游戏时会不断重复认知→判断→操作这一过程（参见图 2.2）。“认知”是指，通过画面和声音向玩家提示某些信息，这些信息通过玩家的眼睛和耳朵而被认识。

具有这些信息后就能进行最基本的游戏了，如今，开发人员还会设法将用户的行动记录在这一不断重复的过程中，从而提供更好的游戏体验。以 Web 浏览器中的操作为例，什么时候点击了页面上的哪一个链接？是从哪里进入当前页面的？用的是什么浏览器？等代表信息本身的 HTML 文件以外的信息，作为日志记录在了日志文件中，然后使用程序对这些日志文件的内容进行分析（日志解析），由此得到诸如“本周最高排名”这样的信息。

图 2.5 的扫雷示例是一个使用画面来告知玩家游戏状态的例子，显然要进行游戏，这个扫雷的 8×8 网格画面是不可欠缺的，但是为了让游戏体验能更加出色，还需要向玩家提供进一步的提示信息。具体示例将在稍后介绍。

游戏的元信息

这些追加信息，也就是游戏进行时的外部信息称为“游戏的元（meta）信息”[18] 。元信息有如下几种类型：❶高分排行榜、❷游戏成就、❸其他统计。每一种都是网络游戏特有的，而且都需要各自的技术解决方案。这里举一些例子来看一下。

> [18] 慎重起见补充一下，这并非业界的专业术语，笔者希望加个名称有助于读者理解，所以在此予以尝试。

❶高分排行榜

高分排行榜就是在世界范围内或者本国范围内，或者其他一些范围内根据游戏中所获得的成绩由高到低进行排序。在网络游戏中，不管是

什么类型的游戏，游戏结果产生后通常都会将这一结果发送至服务器，然后分享至全世界，所有人都可以浏览。

来看一个典型的例子，在游戏时，其他玩家的游戏结果通常会反馈给当前玩家，比如“在所有玩过困难模式的 38 万 9500 人中，您位于 6 万 4010 位”。这样，玩家就能很容易地制定出自己的下一个目标。

❷游戏成就

游戏成就（得分管理）是指在游戏过程中，玩家满足了一定条件后，画面上显示“〇〇已达成”的消息。

Xbox 360 中不管什么游戏都内置了这项功能，比如在高尔夫游戏中，通常会设定诸如“一杆进洞”、“连续 5 次小鸟球[19]”、“总杆数低于标准杆 20 杆”等成就。此外也会设置一些颇具幽默感的成就，“只用推杆就打完了 18 洞”、“击球打中飞行中的乌鸦”等，玩家可以结合自己的兴趣制定下一个目标。

> [19] 高尔夫术语，指击球杆数低于标准杆数 1 杆。——译者注

成就的评定内置在游戏程序（客户端或者服务器）中，在游戏进行时会自动评定，然后发送到服务器并存储在数据库中。现在不仅仅可以将发送到数据库的结果显示在游戏画面上，还可以通知登录游戏的好友、发送到 Facebook 等 SNS（Social Networking Service，社交网络服务）上、或者发送邮件，等等。

❸其他统计

对于其他一些统计，特别是在虚拟世界的游戏中，为了加深游戏体验，通常会将玩家的行为进行统计处理，然后将其结果以可视化的方式显示出来。

比如，《第二人生》统计玩家之间的财产交易，以 GNP（游戏国民生产总值）的方式来表示，这样就能很容易了解到游戏世界中的整体经济状况了。

游戏成就的实现需要更高的技术

从技术上来说，游戏成就这一项的数据非常庞大，而且始终在增加，因此必须具备能够高效存储、读取这些数据的系统。此外，当要与 Web 同步游戏结果时，要求采用不会造成游戏服务器负担的 Web API 的设计方法。如果将游戏成就分得更细更多、运用得更加灵活，就有可能进一步提升游戏体验，因此，相关方面的新提案备受期待。

＊ ＊ ＊

更详细的内容将在第 6 章网络游戏中加以介绍。

2.4.10　更容易地与其他玩家相遇——玩家匹配

网络游戏最大的特点之一就是：其他玩家也是游戏内容（用户要求的信息）的一大部分。在单机游戏中，游戏内容只包含作为软件发布的程序和数据（世界地形、人物角色和敌人的模型、音乐、美术效果、游戏剧情、策划等），而在网络游戏中，其他玩家的行为结果也是游戏内容的一部分。

其他玩家的行为不可能按照事先设定好的剧情来发展，具有不可预料性。但正因为不可预料，才能实现充满意外性的游戏体验。这一点是网络游戏与单机游戏的一个显著差异。正因如此，人们在描述网络游戏时，常会说“网络游戏是一种交流工具”。

正因为网络游戏具备这样一种特性，所以无论是哪种游戏类型，“如何找到其他玩家”都是一项很重要的任务。进一步说，也就是如何满足“只想和志同道合的玩家一起游戏”。为此，网络游戏中通常采用一些“玩家匹配”机制，以适当的方法对玩家进行匹配[20]。

> [20] match，原意是“比赛”，或许有人会认为这是在比赛中一决胜负的意思，但是在游戏行业中，这个词表示“匹配”，“共同协作进行游戏”和“互相对抗进行游戏”统称为玩家匹配。

如今，人们想出了很多实现玩家匹配的方法，大致分为❶自动选择、❷专用游戏大厅和❸虚拟世界这 3 种。

❶自动选择

自动选择这种方式与 Web 很接近。比如，现在想要进行奥赛罗（黑白棋）对战，系统就将按照各种条件对玩家排序。这些条件首先包括当前在线的玩家，这个是必须满足的条件，其次是尽可能与当前玩家的等级相近、比赛时中途退出次数少、曾经与其对战过的玩家，等等，系统从中挑选出最合适的玩家推荐给该玩家。有些实现方法可以在从推荐结果中选择玩家时做到完全自动匹配。这与 Web 系统比较相似。

任天堂公司针对 NDS 游戏机提供了玩家匹配功能，在对应任天堂 Wi-Fi 连接的游戏（比如《马里奥赛车 Wii》）中采用了这种方式。

❷专用游戏大厅

专用游戏大厅是一种“等待系统”，是指为了让多名玩家（比如 3 人）共同游戏而专门构建的系统，在集结了一定数量的玩家之后才正式开始游戏。通常，这是通过很早就开始使用的中继聊天（IRC，Internet Relay Chat，互联网中继聊天）机制来实现的。

❸虚拟世界（可视化游戏大厅）

虚拟世界又叫做可视化游戏大厅，将 3D 虚拟世界在计算机上重现。在这个世界中，玩家化身为各种“虚拟人物”（avatar），操纵自己的虚拟人物与其他玩家的虚拟人物进行交流。通过交流找到想要共同游戏的玩家，然后一起开始游戏。使用虚拟人物可以进行深度交流，因此不可能仅仅根据文字和关键字、玩家分数等定量化的信息来寻找其他玩家，而是要参考对方的想法和性格。

这种方式通常用在需要长时间共同游戏的 MMORPG、MORPG 等类型的游戏中。比如，日本世嘉公司的《梦幻之星 OL》（*Phantasy Star Online* ，PSO）等网络游戏系列就具备了可视化游戏大厅的功能。再比如，面向浏览器的游戏《企鹅俱乐部》（*Club Penguin* ）也采用了这种方式。

专用游戏大厅和虚拟世界的区别

虚拟世界与专用游戏大厅的区别在于：在虚拟世界中，玩家可以实时地进行交流。

将虚拟世界作为 Web 服务来实现是很困难的，这需要专门用于交流的实时服务器，所以专用游戏大厅与虚拟世界在技术上存在极大的差异。

未来的玩家匹配

虽然可以使用以上 3 种匹配方式，但是在匹配的精度和使用的容易程度上，现在都还不够完善，期待将来能有飞跃性的提升。此外，我们希望在降低系统负荷优化匹配时的响应、改进搜索算法使匹配结果更符合要求等方面的技术，将来也能取得进一步发展。

有关玩家匹配的详细内容将在网络游戏 6.2 节中介绍。

＊ ＊ ＊

这一节简要地介绍了商业对网络游戏的技术所提出的要求。网络游戏的技术人员必须在可能的成本范围内实现这些要求。这一点将在第 3 章以后加以探讨。

2.5　网络游戏的人员和组织

本书的目标读者主要是软件工程师和系统管理员，因此本书内容基本上都围绕着软件系统相关的技术，但是本节我们以“人员和组织”为题，简要介绍一下网络游戏的开发工程师们的现状。

2.5.1　与网络游戏服务的运营相关的人员

哪些人与维持一个网络游戏的服务相关呢？我们首先从整体上来看一下，然后从中找到开发工程师所处的位置。

与网络游戏服务的运营相关的人员基本上可以分为开发人员、运维人员和销售人员（参见图 2.7）。

图 2.7　与网络游戏服务的运营相关的人员

全体

开发人员	运维人员	销售人员

有一点要先说明一下，“运营”和“运维”这两个词很容易混淆，事实上它们是不同的。“运营”是指网络游戏服务中所有商业方面的管理工作。确切地说，“运营”＝“管理”。而“运维”则是指设置专门的团队，从事网络游戏的系统维护方面的技术性工作。

3 种职责以及分配方式

一个网络游戏在运营时，通常会将上述的 3 项基本职责分给 3 个公司或部门。大型企业往往能同时承担这 3 项职责，由公司内部的 3 个部门来分担。这 3 项职责的人数分配比例完全由游戏内容决定。公司或部门的职责分配方式大致有如下这些。

- **一个公司负责“开发”，一个公司负责“运维”，还有一个公司或者一个部门负责“销售”（典型情况）**
- **一个公司或者一个部门全权负责“开发＋运维＋销售”**
- **一个公司负责“开发”，另一个公司或者一个部门负责“运维＋销售”**

如果要将一个网络游戏推广到海外或者其他平台，由于地区和平台的不同，用户群体会有很大的差异，因此开发商会与各个地区或各平台中拥有较强实力的销售服务公司签订许可协议，让他们代理自己的网络游戏，以横向发展。这种情况下，职责分配为。

- **一个公司“开发”，一个公司“运维”，一个公司“销售”，多个公司开展“销售服务”**

本书的重点在于技术，并不讨论网络游戏的商业模式，所以在此不再详述。

2.5.2　网络游戏服务运营的 3 项专门职责

如上所述，网络游戏服务运营的相关工作分成了“开发、运维、销售”三项专门职责，每一项所需具备的知识都大为不同，不同场合下的工作类型也不同。这一点在表 2.1 中进行了总结。

表 2.1　3 项专门职能

职能	必要知识	工作类型
开发（人员）	游戏设计、编程、美工、测试、工具制作	集中于一个游戏开发项目，2～4 年内完成项目，然后开始下一个项目。与一般的程序员的工作相同。在大型公司中，就算服务器出问题，这些人员也不会收到大量手机消息
运维（人员）	网络架构和数据库架构、安全性、负载均衡、操作系统管理、数据备份的知识、紧急情况的处理等	基本上就是一般的系统管理员的工作，在运营多款游戏的公司中，也有进一步分为设计架构部门和运维管理部门的。运营管理人员有时需要在晚上工作或者联系他人。如果服务器出问题了，就会收到很多手机消息。在小规模游戏中，通常由开发人员担任
销售（人员）	市场调研、商业调研、资金筹备、QA（Quality Assurance，质量保证）、营业活动、活动举办、商业数据分析等	与实际的开发工作没有直接关系，进行游戏后作评价。为了与各个公司进行谈判交流，常常需要在外奔波。需要同时兼顾多个游戏

由于本书的重点是技术讲解，因此我们首先讨论一下开发人员和运维人员。另外，在网络游戏的运维方面，虽然访问模式、系统更新方式与 Web 服务的系统运维不同，但是因为从 Web 和 SI（System Integration，系统集成）等其他行业转行过来的人相对较多，所以一些必备知识也都非常相似。而由于系统运维方面的书籍非常多，所以本书重点针对网络游戏的特点进行说明。

基于以上因素，本书以“开发人员：80%”、“运维人员：20%”的比例来加以讨论。销售人员方面的内容超出了本书范畴，所以不做介

绍。

2.5.3　开发团队

我们首先来看一下开发人员。当然，在大型团队中会划分多个职务。网络游戏的开发团队有如下这些规模：小型开发团队通常有 2～4 名开发人员，20 人左右的开发团队是最多的，而大型团队中有超过 150 人的。

不可或缺的 4 种职业

就算是兼任，至少也要具有以下这些职位。我们使用常用的行业术语来说明一下：

- **项目总监：统筹管理整个游戏开发过程**
- **程序员：编写游戏程序**
- **美术工程师：制作游戏图像**
- **设计师：根据策划主旨，进行详细设计** [21]

> [21] 制作翔实的数据，为具体工作进行设计。

美术工程师在日本通常称为グラフィッカー（graphic+er），并没有正式的英文单词可以表示。国际上通常称为美术工程师。笔者在本书中也沿用这一称呼。

小型团队

一个团队中，如果上述 4 种职位各有一名经验丰富的成员，就能开发一款小型的网络游戏了。

规模非常小的团队（基本在 4 人以下）也可以开发 iPhone、Flash、移动平台的游戏等，这种情况下通常一人兼任多职。虽说一个人也能开发极小型的网络游戏，但是同时兼任程序员和美术工程师的情况比较少见。

大型团队

如果一个团队的规模很大，那么通常会进一步细分多种职务，然后加以组织。

20 人左右的团队中，通常会细分为如下结构。超过 150 人的团队会根据 20 人的团队分工进一步细分，调整各项分工的人数。

- 项目总监（2 人）
    - 技术总监：负责游戏技术方面的事项、制定开发计划。
    - 美术总监：负责游戏中的世界观设定以及保证美术制作的质量。
- 程序员（4 人）
    - 主程序员：负责程序的整体架构。
    - 系统程序员：负责游戏程序的底层部分。
    - 游戏逻辑程序员：在搭建好的底层架构上实现游戏逻辑。
    - 工具程序员：负责开发包括美工和设计在内的各项工作所需的工具。
- 美术工程师（6 人）
    - 角色设计：负责人物和其他生物的造型设计。
    - 动画设计：负责人物和其他生物的动作设计。
    - 地图设计：负责建筑物和地形的构思和各种组件的设计。
- 设计师（8 人）
    - 关卡编辑：使用地图设计人员的构思和组件制作大量的游戏关卡。

- 游戏剧本编写：负责设计出场人物的台词以及游戏中事件的发生顺序、逻辑关系等。
- 脚本编写：将实际的游戏数据集成到程序员开发的系统中。

20 人中有两名管理人员在游戏行业中是很普遍的。此外，在有 8 名设计师的情况下，为了降低沟通成本，需要由 1 人担当主设计师。

从职责平衡来看游戏开发的特点 ——数据制作人员的比例

这里的游戏开发的特点就是制作数据的人员（美术工程师＋设计师）占了开发团队的一大部分。以 20 人的团队为例，如上所述，项目总监两名、程序员 4 名、美术工程师 6 名、设计师 8 名，共有 14 人担任数据的制作工作。是程序员的 3 倍以上。在 Web 服务的开发项目中就不会出现这种情况。

游戏中各项职责的人员比例，不管是网络游戏还是单机游戏都没有根本性变化。但是根据游戏的类型和内容却有很大变化。在更接近于“工具”的游戏中（比如《第二人生》等），程序员占主要部分，而在更接近于“电影”的游戏中（比如像《勇者斗恶龙》这样的日式 RPG 等），数据制作人员则占大多数。根据游戏类型和游戏内容的不同，所需的职责平衡也完全不同。

如今，在游戏市场上极具竞争力的企业都有自己的一套流程，它们井然有序地、高速、高质量地制作大量数据。游戏开发不是靠几个天才来完成的，而是由整个团队来完成的，这主要是受到由数据构成的系统的影响。

因此，当新的小型企业进入游戏市场时，都会避开电影式的游戏，而是会选择开发工具类游戏、社交类游戏和小型游戏等。

2.5.4　运维团队

接着，我们来看一下运维团队。同样，在大型团队中，职责会进一步细分，但即使是兼任，也要基本分成以下两种。

- **系统工程师**

把握整体计划和游戏的商业模式等，与开发团队协调。

- **网络 / 基础设施工程师**

  负责设备和线路的搭建。

这些人员的工作成果对网络游戏的运维是很重要的，而他们所构建的系统的组成要素分为：❶服务器设备、❷网络设备。

在服务器设备方面，基本工作是根据层次结构来分工。在路由器和交换集线器等网络设备方面，如今已经能够很容易地安装和配置高性能的设备了。在使用 HTTP 以外的协议进行传输时，在传输量（持续会话时产生的流量）方面的性能也很不错，所以这里省略❷这一点，只讨论有关❶服务器设备的内容。

服务器设备

游戏行业所使用的与服务器设备相关的典型层次结构如下所示。

- **应用程序本身**
- **应用程序的基本设施**
- **数据库、文件系统、备份设备、网络设置**
- **程序库和中间件、语言处理系统、虚拟机等**
- **操作系统**
- **硬件**

上述结构中，硬件设备位于底部，应用程序位于顶部。以每月 10 万活跃玩家为例，即使服务器数量和通信流量与支持玩家游戏所需量相同，但是根据游戏的内容、类型、以及通信机制的不同，服务器负载和潜在的瓶颈所在也会大幅变化。比如，在某些游戏中，数据库很容易发生瓶颈，而有一些游戏则是语言处理系统的性能容易成为瓶颈。

因此，担任网络游戏系统设计的系统工程师需要从游戏的策划和设计阶段开始就积极地与开发人员沟通。系统工程师在游戏开发的最初阶段与开发人员进行交流后，再在开发进行了 80% 左右时与开发人员进行最终商讨，游戏发布后，以这个系统工程师团队为中心进行运维管理。

2.6　网络游戏程序员所需的知识

本节，我们将围绕程序员必须掌握的知识以及所从事的工作来进行更深入的探讨。

2.6.1　网络游戏程序员所需的技术和经验

网络游戏程序员所需的技术和经验实际上涉及很多方面。编程基本技术、游戏编程、游戏客户端开发、数据库、系统运维等领域中，都需要网络游戏特有的知识。下面在列举的同时予以说明。

编程的基本技术

作为大前提，成为网络游戏的程序员之前，首先要掌握“程序员”必备的知识，然后再掌握“游戏程序员”必备的知识。如果不掌握编程的基础知识，是无法开发网络游戏的。以此为前提，我们一起来看一下网络游戏需要哪些知识吧。

- **设计**
    - 架构：处理拥有几十万行代码的大规模软件的技术

      →网络游戏是一种复杂的系统，多个进程包含分散在多台服务器上的数据库，它们通过跨越互联网和企业内部网的网络连接，实时协作运行。每个进程的程序大约几千到几万行，从整个系统来看，中等规模的游戏在 20 万到 30 万行左右，大型游戏甚至能达到 100 万行。因为必须要高效处理各种规模的软件，所以需要一些设计方面的技术。这是一项与游戏内容的规模相关，而与通信形式无关的技术。

    - 设计技术：结构化、模块化、面向对象编程、设计模式等

- 设计任务：并发性、事件驱动、错误处理、容错性、可用性

  →需要上面这些基本技术的原因有很多，比如，在互联网中直接暴露服务器和客户端的进程，所提供的服务是实时并且双向的，游戏服务并非是广告模式，以及如果在进行收费时服务器停掉了，就会给用户带来很坏的影响，等等。

- 质量：质量定义、基准、服务水平定义、检测

  →很多情况下，网络游戏，特别是服务端的程序问题只有在进入公测，大量玩家登录游戏后才会显露出来。这是因为有些程序部分（管理计算机资源的机制）只有在接收了来自各种环境的多个同时连接的情况下才有多个。所以必须将这些系统细分为多个子系统分别进行负载测试、然后再同时使用多个子系统进行组合测试，在公测之前验证是否能够实现必须达到的服务水平。

- 记法：UML（Unified Modeling Language）和各种图示、结构，以及行为图的表示

- **架构**

  - 管理：大规模协调工作的计划、预算

    →商业游戏的开发项目中，中小型项目一般有 3 ～ 4 名程序员、2 ～ 3 倍的数据制作人员，大型项目将达到相当于中小型项目的 10 倍左右的人数（100 ～ 200 人）。另一方面，游戏开发的过程中常有变数。在这种环境下要遵守预算的制约来进行开发是很不容易的，所以管理非常重要。

  - 编码：多种编程语言、安全模型、排他机制、内存管理、文档化、优化、松耦合模式

  - 复用

    →在笔者参与过的项目中，尽管没有同时使用到 C/C++、Ruby、PHP、Java、Python、C#、Perl、Lua 等所有的编程语言，但是有同时使用多种语言来进行开发的情况。此外，即

使是同一种编程语言，也混合了代码规范、设计规则等各种组成要素。因此，为了在有限的预算中控制成本，必须尽可能利用商业渲染库、过去开发的工具资产、自己公司开发的程序库等各种可用的资源。为此，必须具备复用以及降低耦合度方面的知识。

  - 质量：单元测试、组合测试

- **测试**
  - 各种测试：功能、性能、负载、安装、可用性测试、各种检测

    →测试时需要大量人员参加，这一点是增加网络游戏的测试难度的一个因素。功能测试、性能测试、可用性测试等，不管是哪一种都是如此。无论如何，网络游戏都必须要有大量人员参与测试，而且在不能进行公测的情况下，需要准备能够模拟玩家的程序。

  - 管理：测试计划

- **维护：现有代码的修复、移植、交接、文档化、说明书**

  →网络游戏的运营将会持续多年。1 年内仍在盈利的游戏在之后的 5 年里并行准备各国语言版后，又会继续运营 5 年。网络游戏的相关工程师在最初的版本发布后进行首次大规模的 bug 修复后，就会交接给后继的工程师（维护人员）。此外，在外国企业中对游戏进行授权的情况下，通常会将游戏软件的一部分作为 SDK（Software Development Kit）来授权。将针对 PC 开发的游戏移植到游戏机上时则相反。像这样，将网络游戏软件的维护进行长期交接需要做很多方面的工作。

游戏编程的基础知识（对网络游戏的开发是必需的或者有用的知识）

在网络游戏的开发（服务端开发）方面，一般的游戏开发技术是必需的。如今，单单是游戏开发所需的技术就涉及很多方面，同时掌握一

般的游戏开发技术以及网络游戏特有的知识的开发人员几乎没有。大多开发人员只了解特定类型的游戏或者特定的部分。

网络游戏的开发人员大致可以分为两类：擅长网络游戏必备技术（渲染、物理模拟和操作等）的开发人员，以及擅长网络游戏特有技术的开发人员。下面列出了一般的游戏开发所需的技术，并且简单的加以说明。由于这些内容与本书的主题相关性不大，所以在此只是简单地提一下[22]。

> [22] 对于那些想了解一般的游戏开发技术的读者，笔者推荐之前也提过的一本参考书籍《ゲームプログラマになる前に覚えておきたい技術》（中文译名：成为游戏程序员之前需要掌握的技术）。讲解量很大（总共近 900 页）。

- **版本管理、自动化构建 / 测试**

  →如今的游戏程序包含了程序代码和各类数据，由多人一起开发。特别是在游戏中为了在版本管理的状态下处理大量二进制数据，必须要有高性能的版本控制系统。Perforce[23] 是游戏行业中广泛使用的版本控制系统之一。此外，自动化构建和测试乍一看似乎不会在游戏测试中用到，但是如今的游戏程序很多都是以层次化方式开发的，在更低的层次上常会实行自动化测试。

- **对排序等基本算法的理解**

  →不通过手工编写来组合基本的算法。

- **对 CPU 结构、命令集等的理解**

- **特定于硬件的知识、内存系统等**

  →游戏画面的显示必须非常流畅。特别是在使用游戏机、移动电话、NDS 等极限性能较低的设备进行开发时，为了持续流畅、高速的处理，必须要理解硬件的工作方式。

- **对 OS、SDK 的理解**

  →在开发比较大型的游戏时，通常会将游戏分为多个模块分别进行开发，如果对现代 OS 和 SDK 的设计的典型实例有所了解，对开发是很有帮助的。

- **制作简单的编译程序**

  在各种游戏中，通常会定义专门的脚本和描述语言以加快开发速度。此时，了解语法解析的技巧是很有用的。

- **对加密方式的理解**

  →在对数据文件进行加密时必须要用到（网络游戏也是同样）。

- **特定于游戏类型的知识**

  →动作游戏、RPG 游戏、射击游戏、卡片游戏、冒险游戏、竞速游戏等不同的游戏类型中，程序的基本架构差别很大（网络游戏也同样如此）。

- **碰撞检测、物理运算**

  →利用物理模拟进行游戏开发，不需要事先制作游戏中的出场者运动时的所有数据，这样就可以用更低的成本开发数据量很大（数据量大就不容易让玩家感到厌烦）的游戏。

- AI

  →根据玩家的操作和行为自动让对手（游戏中的敌人和对战对象）行动起来。与物理模拟一样，使用这种技术能够以更低的成本实现内容更丰富的游戏。

- **对象、任务系统**

  →如今的游戏以每秒 60 次的频率来并行控制几千、几万个游戏中的对象是很普通的。用一般的 OS 本地线程是无法做到并行处理的，所以需要专门的并行处理方法。

- **嵌入式脚本**

  →使用 Lua、Squirrel、Python 等适合嵌入的脚本语言可以加快游戏的开发。

- **转换工具、设置工具、GUI 工具**

→游戏开发中有编程、数据制作、美术制作和策划几大类。其中占比例最高的是前述的数据制作这一块。数据制作指地图制作、事件制作、物品制作等，最近普遍使用 C# 和 Web 应用等适合制作工具的开发环境来为各种游戏类型制作新的工具。大家常说“有熟练的工具程序员的游戏开发团队必会成功”。不过现实是满足这个条件是很困难的。

[23] Google 也使用过该系统。http://www.perforce.com/

游戏客户端开发的知识

网络游戏的开发中，必须掌握以下这些客户端开发的知识：

- **二进制操作、数据文件操作**

  → SoftImage （XSI）和 3ds Max 等 3D 创作工具、图像制作工具、音频控制程序等与 数据制作相关的各种数据格式的文件以及插件等各种专属的工具非常多。在这种情况下，为了获得理想的处理性能，一般会采用二进制格式的文件。为了提高开发效率，通常需要制作一些访问这些文件的工具，但是很多情况下，数据文件并不会被文档化，也并非是对用户友好的。因此，操作二进制格式文件方面的知识对程序员，特别是对工具程序员是非常重要的。

- **文件的打包**

  →有很多方法可以提高从 DVD、Blu-ray Disc、HDD 等设备中读取数据的速度，其中也有一些方法是特定于游戏的。通常游戏中会大量使用非常零碎的图像，大约会用到几万乃至几百万种图像。在一般的文件系统中解压如此大规模的文件时，随机存储的时间将大幅增长，游戏体验就会变差，而且还要压缩磁盘容量，因此根据游戏的类型、内容以及场合，我们可以采取以下这些措施：将部分文件并成一个文件、使文件冗余、从试玩日志中找到最合适的模式。

- **操作菜单和特效**

→在开发中，操作菜单和特效会被频繁地改进、变更。为了禁得起频繁的改动，有必要采用 Flash、准备专门的开发工具，进行工具化、抽象化。

- **渲染（2D、3D）**

  →在游戏中，渲染是最为基本的，因此这是与游戏的性能直接相关的一部分。有些游戏中，处理器能力的 90% 以上都用在渲染上。如今，一些大型游戏通常使用第三方开发的渲染库[24]，但是根据环境和游戏的内容，有时也需要自己开发。

> [24] Epic Games 公司的 Unreal Engine、Emergent Game Technologies 公司的 Gamebtyo、Unity Technologies 公司的 Unity 等库都广为所知。

限于篇幅，本书对这些知识的说明只能到此为止了，但是除了上述这些，音频和视频方面的知识也需要有所了解。

数据库知识

网络游戏几乎不可能不用到数据库。在游戏中使用数据库的方式也是相当不同的，与商务系统的要求大不相同。详细内容将在后面的章节中探讨，这里只列举一些要点。

- **SQL、查询优化、高速缓存、扩展性**
- **根据各种 DBMS 和 DB 库的用途分开使用**

系统运维知识

虽然系统运维与 Web 应用开发有很多共同点，但是在访问模式和服务器构成，发生故障的地方及其发生原因、修复方式等方面就有所不同了。所以需要掌握一些必要的系统运维知识。

网络游戏的特定知识如下所示。印有 † 标记的内容将在第 7 章中详细讨论。

- **各种业务流程**

  通告、数据入库、用户管理、维护等

- 服务器部署 †
- 负载均衡
- 数据备份、数据恢复[25]
- 服务器监控（Monitoring，状态监控）、服务器心跳监控（监测服务器是否在运行）、监控工具 †

> [25] 在第 7 章中有简单介绍。详细内容请参考 MySQL 相关书籍。

2.6.2　各种网络游戏开发知识

以上介绍了网络游戏程序员必须了解的相关技术和前提技术。本书后面将要具体介绍的技术与其他领域的程序员所需掌握的技术有所不同。

就算是熟练的网络游戏程序员，也几乎不可能在所有领域中都能立刻编写程序，但是如果掌握了上述知识的绝大部分概要，就能立刻理解说明文档中的内容，并且可以立即判断自己是否能够实现。

如今游戏开发方面的技术有很多，以后肯定还会进一步扩展，单单是掌握概要这一点，用普通的方法也难以做到。当然，以上技术中有很多部分在多数企业中都已经采用了中间件，没有必要从零开始开发，但是不理解其内部实现原理就不能有效地加以使用，所以应该尽可能了解这些内容。

2.7　支持网络游戏的技术的大类

至此（2.1～2.6 节），我们已经介绍了支持网络游戏的一些基本要素：包括网络游戏的相关技术、物理层面、概念层面、市场和商业层面、人员和组织层面，以及网络游戏程序员所需了解的各种知识。

下一章终于要开始正式探讨详细的技术内容了，在此之前，本节和下面的 2.8 节首先简要地介绍一下支持网络游戏的技术的大类，以及影响开发成本的技术要素，让读者们先热热身。

支持网络游戏的技术的 4 种形式

支持网络游戏的技术分为物理结构和逻辑结构两大部分，总共分为 4 种形式（类型）。第 0 章中已简单介绍过，这里进一步加以说明。

C/S 架构和 P2P 架构 ——物理结构的两种典型模式

支持网络游戏的技术的物理结构就是指“实际进行通信的设备之间存在怎样的关系”。

物理结构中有 C/S 和 P2P 这两种典型的模式。表 2.2 简单总结了这两种分类以及它们之间性质方面的差异。表中的各项：延迟（Latency）、服务器设备成本、带宽成本、非法行为都是要点。这些我们将从第 3 章开始详细介绍。

表 2.2　物理结构上的分类及其性质上的差异

	C/S	P2P
延迟（所需时间、通信延迟）	大	小
服务器设备成本	大	小
带宽成本	（根据游戏内容决定）	
非法行为	困难	容易

MMO 架构和 MO 架构 ——逻辑结构的两种典型模式

逻辑结构则是指“实际进行游戏的玩家之间存在怎样的关系”。逻辑结构中有 MMO 和 MO 这两种典型模式，它们拥有完全不同的游戏内容。

前面讲过，MMO 是 Massively Multiplayer Online 的缩写，这类游戏的目的是让大量玩家长期游戏。因为要接收大量玩家，所以必然会牺牲响应时间。又因为要供玩家长期游戏，所以必须处理大量数据。

MO 是 Multi-player Online 的缩写，这类游戏意在让少数玩家享受实时对战的乐趣。这类游戏需要尽可能追求高速的响应时间。由于要在短时间内实时进行游戏，所要处理的数据微不足道。

MMO、MO 这两种类型的游戏在内容上有很大的差异，所需的技术截然不同。根据逻辑结构，表 2.3 对其分类和特征进行了比较。这些差异是理解的重点，请务必掌握。

表 2.3　逻辑结构上的分类及特征

	MO	MMO
游戏核心	少量玩家聚集在一起竞赛	大量玩家进行社交活动
同时玩家数	20 左右	200～1000000
延迟	50 毫秒	300 毫秒
游戏时间（累积）	几分钟	几年
RMT（真实货币交易，Real Money Trading）	很少	活跃
平台	游戏机	PC、移动设备

网络游戏的 4 种形式 ——物理结构 × 逻辑结构

物理结构有 C/S 和 P2P 两种，逻辑结构有 MO 和 MMO 两种，它们各自独立，所以总共有 4 种形式（参见表 2.4）。

在实际的游戏中，除了 P2P 加 MMO（P2P MMO）这种模式是不存在的，其他的所有模式在游戏中都有广泛运用。

表 2.4　网络游戏的 4 种形式及游戏类型

	MO	MMO

	MO	MMO
C/S	休闲游戏	MMORPG 虚拟世界、大战
P2P	ARPG（动作类 RPG） 对战格斗游戏、FPS 竞速游戏、RTS（即时战略游戏） 射击游戏	×

2.8 影响开发成本的技术要素

本节面向程序员总结了“网络游戏的开发成本”和实现上的难易程度方面的要素。

2.8.1 网络游戏与如今的开发技术

开发网络游戏时必须对各种方面进行权衡，但现在的开发人员已经不需要完全从头开始考虑所有的折衷方案了。典型的模式已有不少，可以在此基础上考虑如何进行定制。

在笔者印象中，20 世纪 90 年代后期为了高效地开发 Web 服务而产生了一种 3 层结构，可以说这与之后的情形很相似。在那之后扩展 Web 服务的典型方式得到了整理。而在如今的网络游戏开发中，各种技术也得到了整理。

2.8.2 支持网络游戏主体的 3 大核心

网络游戏的整体技术可以分为“游戏主体”的软件技术和“辅助要素”（辅助系统）两大类。其中，辅助要素已经得到了相当程度的封装[26]。相关内容将在第 6 章中详述，我们先来看一下形成有关“游戏主体”的典型模式的 3 大核心（技术要素）。

[26] 举个例子，比如 nonoba.com，将本地化以外的功能全部封装起来，可以用很低的成本使用来自世界各地的封装包。

这些核心有如下方面，对这些方面都要有所权衡。

- 游戏的数据形式
- 游戏的通信方式
- 游戏的反应速度（延迟）

上面这 3 个技术要素会影响到程序在实现上的难易程度。这里的“实现上的难易程度”指的是如果需要更复杂的算法，或者要处理很多异常情况，又或者因为估计到攻击种类会有很多而必须采取多种防御手段等，由于种种理由而导致的编程工作增加、软件复杂性增加。这与“开发成本”直接相关。总成本的大小可以通过将上面这些技术要素相乘来加以想象。

游戏的数据形式

本书所讲的“游戏的数据形式”指，包含了一系列与游戏进行相关的所有信息的数据是以怎样的形式存储在物理媒介上并加以使用的。

物理媒介有 RAM 和磁盘。内存等 RAM 的特点是易失（当电源关闭时 RAM 不能保留数据），容量小但速度快。而 HDD 等磁盘的特点是持久、容量大但是读写速度较慢。RAM 访问速度以纳秒（10^{-9} 秒，ns）为单位，而磁盘的访问速度的单位则是微秒（10^{-6} 秒，ms）。 它们之间相差了 3 个数量级，它们的使用方式将会影响软件的设计 [27] 。

> [27] 最近，SSD（Solid State Drive，固态硬盘）开始介于这两者之间，网络游戏在数据库服务器中用到 SSD 的情况特别多。另外，SSD 的访问速度介于 RAM（几微秒）和磁盘（几毫秒至几十毫秒）之间，大约为几十微秒至几百微秒。

是使用易失的 RAM，还是使用持久的磁盘，要根据游戏数据的持久性，也就是游戏进行的持久性来决定。

从将一系列的游戏信息全部存储在 RAM 中，到将 RAM 的使用控制在最低限度而将大部分信息都存储在磁盘中，根据游戏的类型和内容，游戏的数据形式也是各种各样的。

顺带提一下，在使用 Ruby on Rails 等框架开发的、不使用高速缓存、低负载的 Web 服务这样的存储中心的应用程序中，数据在内存中

存放的时间相当短（几毫秒的程度），绝大部分时间都存放在磁盘上，但是在像地球模拟器（Earth simulator，一个基于高性能集群计算的模拟地球的自然地理过程的集成计算环境。）这种计算中心的应用程序中，决定哪些数据要长时间存放在内存中，这对服务器的性能好坏是起关键作用的。网络游戏的技术中这两方面都包含在内。

- **游戏数据形式的分类**

从本书中游戏内容的持久性的观点来看，游戏的数据形式分为以下两大类[28]。

> [28] 除此之外，最近还有一种数据形式，游戏内容将会持久地存在于服务器中，而且还能实时变化，这种形式称为动态持久化（dynamic persistent）。动态持久化实现起来比一般的持久化形式更加复杂。有关这两种形式的内容请参照 3.5 节。

- **一次性的（disposable）：每次初始化游戏内容后就丢弃**
- **持久化的（persistent）：游戏内容持续存在于服务器端**

首先，游戏内容的持久性指的就是游戏进行时相关数据的存续时间。一般来讲，在格斗类游戏和竞速类游戏中，游戏内容的持久性相对较短，象棋和游戏人生类的就稍微长一些，而 RPG、MMORPG 和虚拟世界这种类型的网络游戏中的数据则存续时间相当长。

“一次性的”（disposable）意味着游戏的数据“在使用后就丢弃”，一系列的游戏进展将在几分钟内结束。游戏从开始到结束所需的几分钟时间内将游戏内容存储在内存中，只要能在这段期间保证数据不被破坏就不会有问题。

“持久化的”（persistent）意味着游戏数据“永远存在”。在将要持续数十小时的游戏中，游戏中断后，必须在必要的时刻恢复数据。而网络游戏更是在长时间内有多人共享数据，所以必须确保在服务器的内存和磁盘中正确、完整地保存游戏状态，只要玩家一有需要，就瞬间取出相应的数据。比起“一次性”的数据形式，实现“持久化”的数据形式困难得多。

保存信息的地方根据游戏的数据形式也有所不同。可以保存在客户端或者服务器端。一般来说，保存在服务器端的数据更值得信赖，客户

端上的数据有可能会被全部破坏。一台客户端上的数据复制到另一台客户端上，而在服务器上是被持久化的这种情况需要特别注意。

＊ ＊ ＊

3.5 节以逻辑结构、MMO 架构、持久化为中心，详细介绍了游戏的数据形式和数据的持久化方面的内容。

游戏的通信方式

网络游戏的通信方式是指，游戏的数据包通过怎样的路由在玩家（客户端）之间进行交换 [29] 。

> [29] 如前所述，游戏客户端就是“用户为了进行游戏而在自己的计算机上运行的软件”。并非客户端 / 服务器架构（C/S 架构）中的客户端，而是“用户终端上运行的游戏程序”。

本书以经由互联网连接的 P2P 和 C/S 为中心进行说明 [30] 。

> [30] 与 P2P 同等的 ad-hoc 模式的通信，以及 LAN 局域网内对战这种局限于场地的本地通信在这里不予讨论。

- **P2P：没有中央服务器，各个终端之间直接相连的通信方式**
- **C/S：只在客户端 / 服务器之间进行通信的星型结构（将在以后介绍）的通信方式**

有关 P2P、C/S 的内容将作为物理结构在 3.3 节中详细讨论。

游戏的反应速度

游戏的反应速度（响应速度）就是“认知→判断→操作”的速度。比如，这 3 个步 骤在象棋中，短的话需要 2～3 秒，长时间思考的话需要几个小时。在电子游戏中，指的就并非象棋的棋子了，而是以游戏的控制器和键盘、鼠标为操作对象，通过显示器和扬声器输出。

人类的动态视力和听力在适应了移动速度的情况下，可以分辨出物体在 0.1～0.2 秒内的动作变化，在可以预想到物体接下来会如何运动的情况下，能够分辨出 0.05 秒内的动作变化，而如果是出于自己意图进行动作的话，则可以分辨 0.01 秒内的动作变化。计算机显示器

上的显示内容的更新速度在 0.01～0.02 秒（每秒更新 100～50次）。

在充分发挥人体能力的游戏类型（比如动作游戏）中，在每次显示器对画面进行更新的这段间隔[31] 内，游戏情况可能发生变化。比如，在对战格斗游戏的代表作“街头霸王”等系列中，玩家能否看穿这 1 个渲染帧内的差异并且释放技能就会直接影响到游戏的胜负。

> [31] Interval。这里的“间隔”称为 1 个渲染帧。

如今的电子游戏开发中基本上全部都将帧缓冲区（Frame Buffer）中的内容输出在光栅扫描式图形显示器（光栅显示器）上。光栅显示器每秒进行几十次的画面更新。在使用一般的电视机的情况下，典型的是以每秒 60 次的频率来更新画面。除了军事模拟和工业用途等特殊目的，这种画面更新时间（0.0167 秒 =16.7 毫秒，大约 16 毫秒）对玩家来说，是游戏状态变化最短的时间间隔。

因此，在认知→判断→操作这一循环中，认知与认知之间的时间间隔在 0.0167 秒以上。在要求高速判断的网络游戏中，必须在这段时间内进行数据包的往返传输。因此在多人游戏中，必须要在这段时间内将操作信息以某种方式（通常是以数据包方式）从某个玩家的机器上传送到其他玩家的机器上。

- **游戏处理的冗余化和异步化** ——网络延迟问题

  网络游戏自然离不开网络。网络也是各种各样的，我们首先来考虑一下本书主要讲述的使用互联网的游戏，因为通过互联网进行数据交换是存在一定的物理距离的，所以通信延迟是避免不了的。

  另一方面，网络游戏的开发也不是这样的。游戏软件在运行它的设备中处理完成后，只要程序不出错，就可能在 16 毫秒内完成传输。

  在网络游戏的开发中，一旦网络延迟成为一大问题，就会对游戏造成不好的影响，所以必须尽量避免这个问题。因此，游戏处理的冗余化和异步化是很有必要的。但随之而来的程序复杂性也会急剧增加。

冗余化指的是游戏数据在较远处和较近处的两个地方进行复制，保持数据的冗余（主数据和副本数据之间的关系）。

异步化指的是保持冗余的游戏数据在不同的时刻发生变化。

在互联网上，分散的、冗余的、异步的游戏数据要保持整体的完整性在理论上是不可能的。但是保持从玩家角度看到的完整性就并非不可能了，为了这个目的，特定于各种游戏类型的方法在不断发展。

- **网络延迟的 3 种形式** ——同步方式、异步方式、浏览器方式

为了对应网络延迟的 3 种典型形式，表 2.5 总结了作为实现方法的 3 种类型：同步方式、异步方式和浏览器方式。这 3 种类型要从想要实现的游戏内容和类型、平台、逻辑结构（MO 还是 MMO）等方面进行综合判断。

下面列出了表 2.5 中的各种延迟及其对应的游戏类型。

  - **25 毫秒以下**

    实时对战格斗类游戏和正式比赛中的延迟需要控制在这个范围内

  - **100 毫秒以下**

    FPS 和战略游戏等一般的游戏中的延迟必须控制在这个范围内

  - **300 毫秒以下**

    MMORPG 等动作性少的 C/S 游戏中的延迟必须控制在这个范围内

表 2.5　网络延迟与实现方法

名称	延迟	同步/异步	冗长	补充
同步方式	25 毫秒以下	同步	非冗余	在高速网络的前提下，同步方式的实现方法比较好
异步方式	100 毫秒以下	非同步	冗余	为了在低速的网络上实现高速的游戏，冗余性是必需的
浏览器方式	300 毫秒以下	非同步	非冗余	延迟的程度超出了用冗余性可以解决的范围，所以将游戏数据和游戏客户端（游戏浏览器）完全分离

同步方式指的是，运行在所有进行游戏的玩家的设备上的程序是同时运行的。也就是说，只要有一个人的网络发生延迟，程序运行变慢，所有人都会受到影响。

异步方式指的是终端程序异步运行，各个终端上的游戏进行状态也是异步的。就算某一台终端上的程序停止运行，其他终端的程序也能照常运行，但是会发生“游戏数据不匹配”的情况。

最后，浏览器方式是指所有的游戏内容都运行在某个中心程序（通常称为主机、服务器等）上，其他程序只起到浏览游戏内容的作用。这种情况下，即使服务器以外的程序都停止运行，游戏也能继续运行，也不会发生数据的不匹配。因此，游戏停止的概率可以大幅下降。但是由于所有的信息都经过服务器，延迟就会相应增加。

* * *

冗余化（游戏进行的主数据和复制而来的副本数据）、延迟方面的技术非常重要，我们将从第 3 章开始详细介绍。此外，本章最后设置了面向程序员的专栏供大家参考。

2.9　小结

本章就“何为网络游戏”，简要地说明了网络游戏的物理层面、概念层面、商业层面、人员与组织层面，以及技术大类、影响开发成本的要素等。读者对于网络游戏已经有了一些具体认识了吧。从第 3 章开始，我们将以 2.7 节的技术分类中提到的 4 种模式为基础，详细讨论一下如何进行选择。

专栏　网络游戏编程的最大难点——解决冗余和异步的问题

本专栏将介绍使网络游戏的编程变得极为复杂的两个要素：游戏处理的“冗余”和“异步”问题。这两个要素将在第 3 章以后的章节作深入讲解，在此，笔者先进行一些补充说明，对这两个问题有初步了解的程序员们可以参考。

游戏进行时的状态保存在某一侧（服务端还是客户端）的计算机内存上，每当玩家做了某些操作，就会有一部分指定的数据被改变。网络游戏中，“某一侧的计算机”就是“实际在什么地方”的问题。这在 SNS 等 Web 服务中完全不是问题，但却是网络游戏开发中最大的问题。

为了深入讨论这个问题，首先必须对一些非常基本的概念有所了解。

- **游戏处理的冗余化→解决主数据和副本数据之间的关系的问题**
- **游戏处理的异步化→解决在处理数据变化时同步和异步之间的关系的问题**

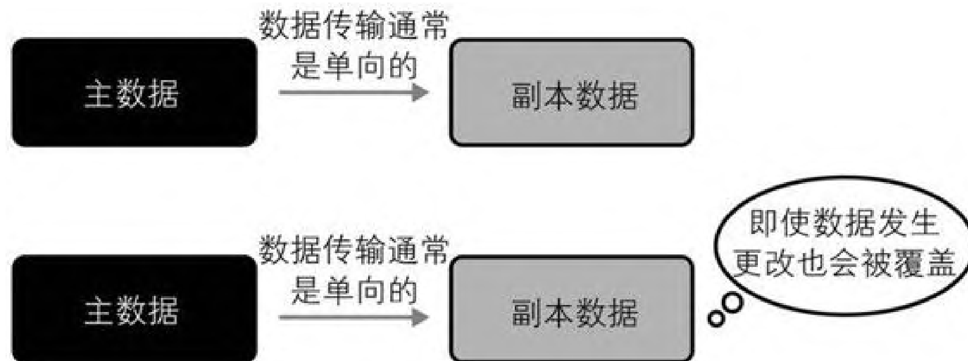
主数据和副本数据 ——游戏处理、游戏的冗余化

我们首先复习一下有关主数据和副本数据的概念。这些术语在我们的日常会话中也会经常出现，但在这里我们尝试对其进行更为严格的定义。

主数据指的是原本的数据，副本数据是复制之后派生出来的数据。如果是数据库中的数据发生了改变，那么首先会对主数据执行修改，然

后将主数据发送给副本。即使副本中的数据也发生了改变，但是之后发送过来的主数据会将这些改变覆盖掉，对副本所作的更改就会丢失。图 C2.A 展示了这一情况。该图中，即使在副本一侧进行修改，但是因为数据传输总是单向的，这些修改全部都会丢失。

图 C2.A　主数据、副本数据单向传输的关系

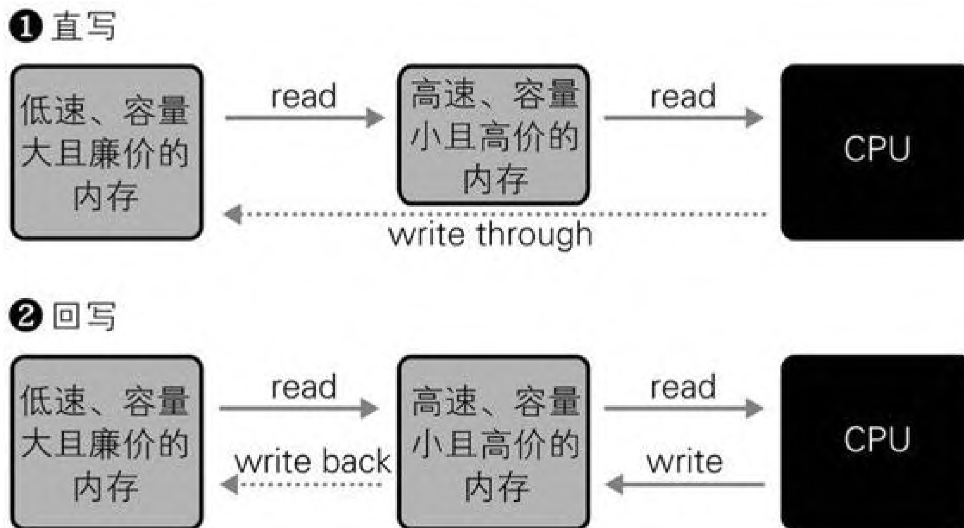


数据是一种“被频繁使用”的东西。本来进行复制的目的就是将数据放置在离自己较近的地方来节约访问时间。比如，会议资料只有一份，在没有副本的情况下，参加会议的所有人员都不得不伸长脖子等着看那份资料，信息共享的速度非常慢。因此，“复制”的好处就是，主数据和副本数据的距离想多大就能多大。

高速缓存的例子

通过复制数据提高访问性能的典型例子就是“高速缓存”。存在于 CPU 和主存之间的 SRAM（Static RAM）是一种比主存的速度更快的高速存储器，将主存中的数据复制到 SRAM 中可以提高处理性能（图 C2.B）。

图 C2.B　高速缓存的直写式和回写式



在最初提出高速缓存时，最原始的方式就是只有在从内存中读取数据时才使用高速缓存，在写入数据时并不缓存，而是直接写入主存中。这种方式称为“直写”（Write through，图 C2.B ❶）。在这种情况下，向主存写入数据后，缓存中对应的内容就不是最新的了，无法再使用。因为几乎所有的应用程序都会频繁进行读写操作，所以除了一些极为特殊的应用，这么做并没有带来显著的性能提升。

随后，一种称为“回写（Wtire back，图 C2.B ❷）”的缓存方式被提了出来，数据被高速写入缓存后，在具备了某些条件的阶段再将数据写入主存。在“具备某些条件的阶段”这一方面，各大芯片制造商实现了几十种不同的方法，比如“采用先进先出的算法写入”、“采用 LFU 算法（least frequently used，最不经常使用页置换算法）写入”、“随机写入”、“顺序写入”这种简单的方式，到使用更为复杂的算法、使用某些统计信息等，有各种各样的方式。

为什么回写方式中必须要提出这么多种写入方式呢？这与❶处理器的用途、❷内存的传输延迟和容量、❸处理器数量这 3 个方面有很大的关系。在❶处理器的用途方面，比如在信号处理器的情况下，通常持续对内存区域的 2～3 个地方进行一元扫描，而另一方面，在互联网路由的情况下，就变成了片断地、零散地随机访问大片内存区域。对于❷，在主存延迟方面，有在 1 个 CPU 周期内完成对数据的访问的，也有必须要花费 100 个 CPU 周期的。而内存容量也在 8KB 至几百 GB 之间不等。对于❸，处理器的数量少的只有 1 个，多的可以达

到 1024 个以上，因为涉及的方面非常多，所以必须要有多种方式来处理不同的情况。

以上内容偏离了本书主题，所以这里不对高速缓存系统进行详细讨论。根据主数据 / 副本数据的思想、应用程序的用途以及处理器、内存等物理组件的性能 / 性质，缓存系统的设计涉及很多方面，必须有各种技术上的解决方案，我们重点讨论这些。

网络游戏和高速缓存的比较参见表 C2.A，要求不同，网络游戏的系统设计变化很大。

表 C2.A　高速缓存和网络游戏

高速缓存	网络游戏
处理器用途	游戏内容
内存延迟和容量	数据包延迟和游戏的数据量
处理器数量	玩家数

异步方式、同步方式 ——游戏处理的异步化

接着我们再来举一个运用复制提高工作效率的例子：软件的版本控制系统。本书的读者应该每天都用得到，比起高速缓存，读者应该对版本控制系统更有切身体会。在这里，为了对其功能进行比较，我们以功能相对简单的 2 个产品为例进行说明，一个是开源的 Subversion，另一个是微软在 Team Foundation 之前发售的 Visual Source Safe。

Subversion ——异步方式

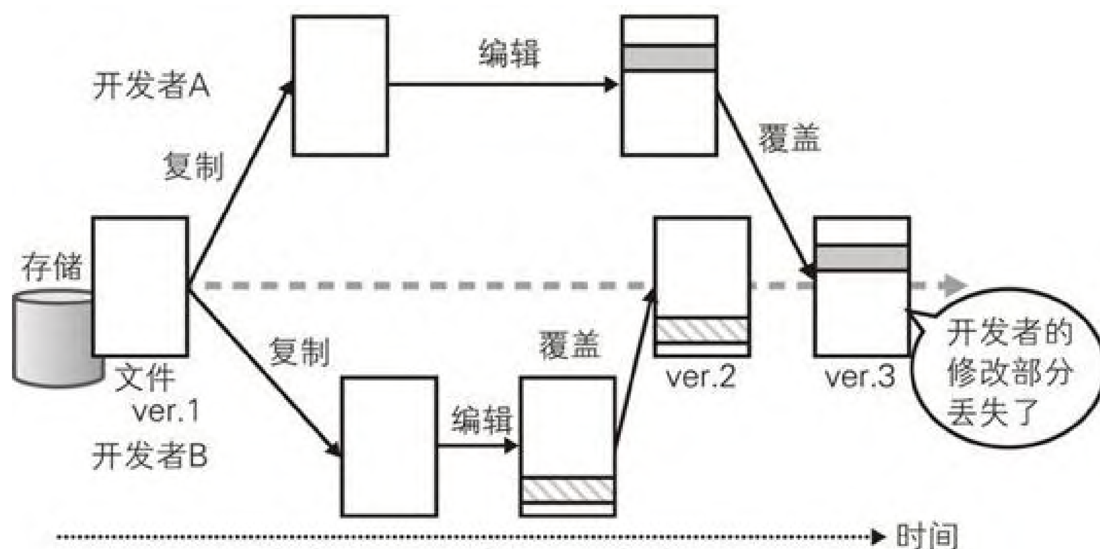
Subversion 的基本工作流程是：将源代码从版本库（repository）复制到本地→“在本地对复制下来的源代码进行修改、添加、删除”→向版本库提交更改。

这里说的“版本库”就是指主数据，通常存放在事先准备好的可以经常访问的服务器上。“提交”就是指将改动过的部分发送至版本库，替换其原本的内容。

通过 Subversion，可以将开发所需的源代码全部复制到本地，然后对这份副本进行编辑。这样就不会受到与服务器进行通信所带来的延迟的影响了，还可以和世界各地的开发人员同时工作。

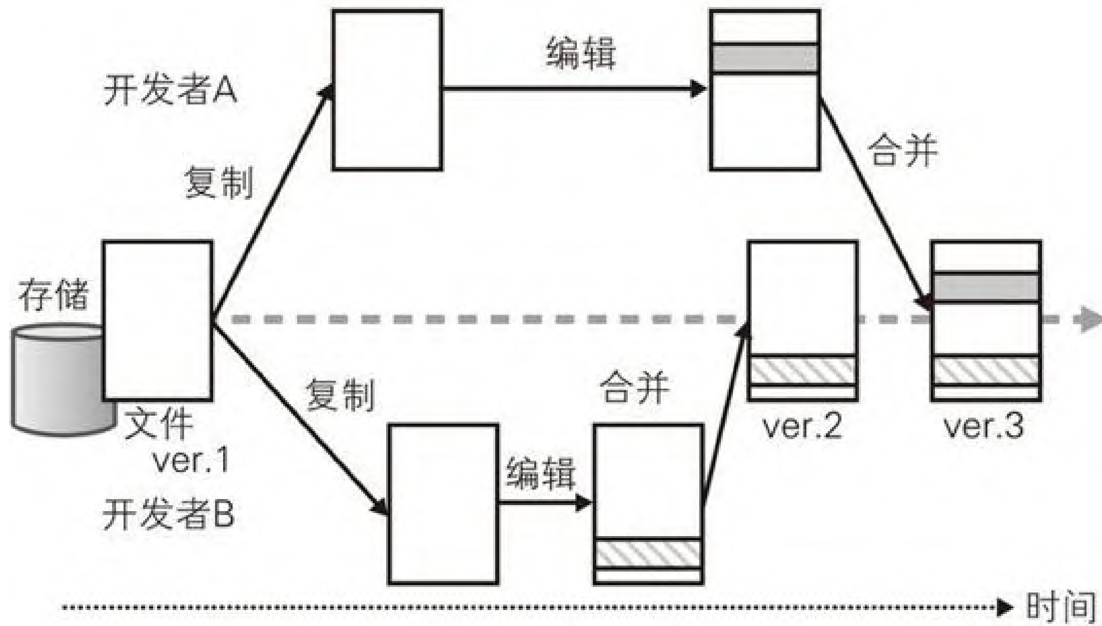
但是，如果同一份源代码文件同时被两名开发人员修改了，那么在提交时就会出现问题（参见图 C2.C）。图 C2.C 中，开发人员 B 的修改内容被开发人员 A 的内容覆盖了，那么 B 所作的修改（图中条状部分）就丢失了。

图 C2.C　覆盖问题



为了解决这个问题，Subversion 具有一项称为“合并”的功能，以源代码的每一行为单位自动混合源文件的内容（参见图 C2.D）。

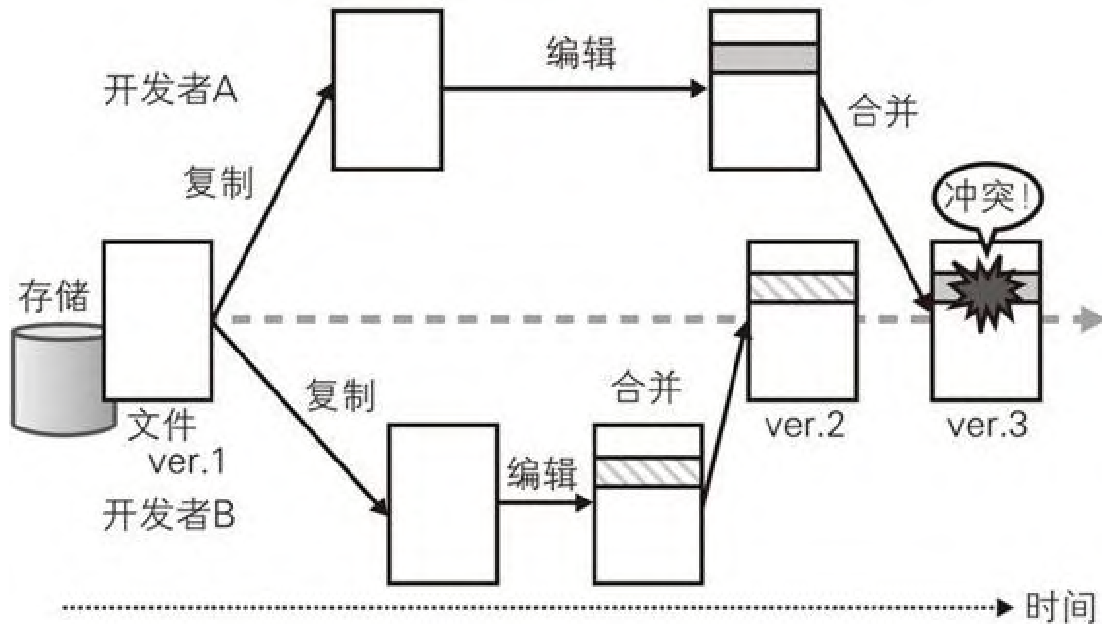
图 C2.D　合并功能



不能覆盖的文件部分在服务器端自动合并，这样辛辛苦苦所做的工作就不会丢失了。一点错误都不能有的源代码文件可以就这么交给服务器来合并吗？当然不行，过分依赖工具势必出现问题，所以必须再将合并后的文件取回本地进行检查。

即便如此，还是有个问题。如果两个开发人员同时对同一个源文件的同一个地方进行了不同的修改，情况又会如何？（参见图 C2.E）此时，Subversion 检测到冲突，然后发出合并失败的警告。在这种情况下，必须要由开发人员自己手工修正。如果采用模块化方式编写程序，可以有效降低发生冲突的概率，从而提高开发效率。

图 C2.E　冲突问题



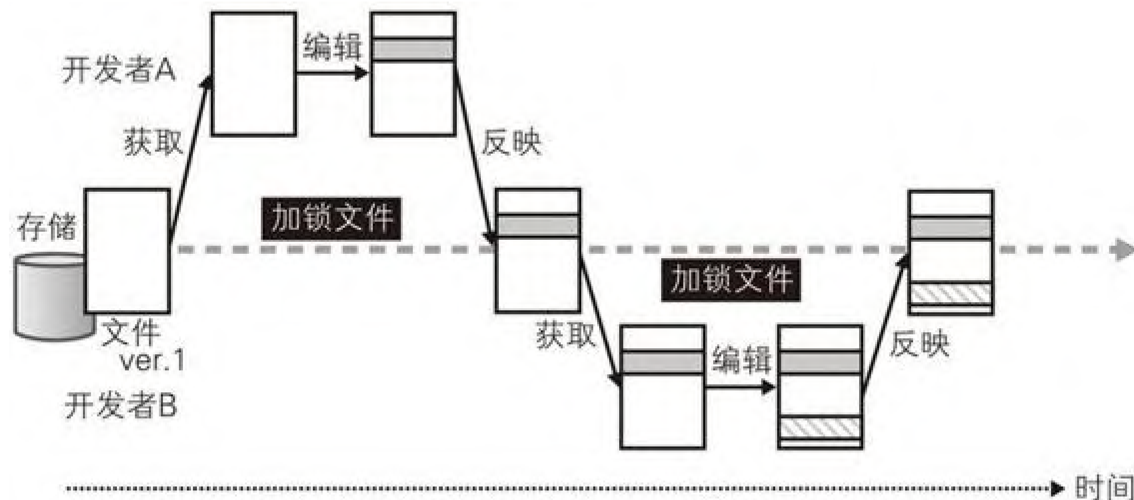
在使用 Subversion 的情况下，开发人员需要面对“如果发生了冲突就必须手工解决”、“合并之后必须进行充分测试，看看是否运行正确”这样的麻烦问题。当然，这种问题很少发生。发现一些麻烦问题是非常重要的，网络游戏的情况将在后面详述，这些问题将直接关系到玩家的游戏体验。

Subversion 的处理方式相当于网络游戏中的“异步方式”。

Visual Source Safe ——同步方式

微软的 Visual Source Safe 采用了与 Subversion 完全不同的方式来寻求问题的解决之道。也就是“以源文件为单位对其进行加锁（Lock）”（图 C2.F）。

图 C2.F　文件加锁



给文件加锁也就意味着其他人不能访问该文件。在图 C2.F 中，一开始时，开发人员 A 获取文件并对该文件加了锁，在此期间，即使开发人员 B 想要访问该文件也无法访问。当 A 完成工作并将修改部分提交到版本库后就解开锁，然后开发者 B 才可以开始对该文件进行修改。

在这种机制下，文件的修改不会发生冲突，总是能保持完整的状态。但是另一方面，开发人员却会陷入“想要访问却不能访问”的困境。在一个房间里工作的 4～5 个开发者要使用某个文件的情况下，只要说一声“现在我要对这个文件加锁了”，那整个房间的人就都知道了，在这种情况下，这一问题并不严重。

但是在开发人员分布在多个地点、互联网的开源活动很活跃、想要在不同的时间段内工作等情况下，“想要访问却不能访问”比起“如果发生了冲突就必须手工解决”更加糟糕。所以微软在 Visual Source Safe 之后的团队支撑软件 Team Foundation 组件中，增加了对合并方式的支持。

使用锁定机制的 Visual Source Safe 所采用的方式在网络游戏中称为“同步方式”。

表 C2.B 中总结了以上两种源码控制工具与网络游戏的对比。

表 C2.B Subversion、Visual Source Safe 和网络游戏

项目	Subversion	Visual Source Safe
用途	供分散的开发团队使用	供聚集在一间屋子里的小团队使用
避免覆盖和冲突的策略	复制 + 合并	文件加锁
在网络游戏中的对应说法	异步方式	同步方式

在判断网络游戏的架构时，必须决定“用哪里的 CPU 处理数据”。为此，必须对有关高速缓存的写入方式的策略、源代码控制工具的策略等关于“对于从主数据复制而来的副本数据，应如何处理对其进行的处理”有所了解。接下来，我们一起进入下一章吧。

第 3 章　网络游戏的架构：挑战游戏的可玩性和技术限制

游戏软件的最大价值就是可玩性。为了设计出有趣的游戏，游戏开发人员可谓使尽浑身解数。网络游戏的开发人员也不例外。网络游戏当然也属于游戏，为了保持游戏的可玩性，同样必须具备“游戏编程特性”这一大前提。一般来讲，当人们进行某项操作之后，如果立刻就能引起变化（得到响应），那么人们对这件事的兴趣就会持续下去。因此，为了保持游戏的可玩性，游戏编程必须具备一种特性，即流畅地持续重复“认知→判断→操作”这一过程。具体来说，就是游戏必须在为时 16 毫秒的 1 次循环中顺利完成 以下工作。

- 通过画面和声音适当地对游戏进程给予提示，帮助玩家迅速了解当前情况并作出判断。
- 玩家可根据自己的判断进行输入操作。
- 输入操作即刻反映在游戏进程中。

以上这些工作都必须顺畅地进行处理，说是“响应主宰一切”也不为过。如果处理时间忽长忽短，玩家就会觉得游戏不稳定，如果反应迟缓，又会觉得响应太差。这些都是剥夺游戏乐趣的重要因素。尤其在多人游戏中，随着游戏的进展，会包含越来越多的信息，需要显示在画面上的对象也会不断增加，而响应速度也容易因此变慢。

另外，作为网络游戏特有的要素：通信延迟、带宽、客户端、服务器、安全性、辅助系统等也是必须了解的。

一般来说，比起单机游戏，开发网络游戏所需要的技术知识更多。第 3 章前半部分将对“维持游戏可玩性的要素”和“维持网络游戏特有的可玩性的要素”进行一些总结和说明。然后在后半部分将详细探讨网络游戏的物理架构和逻辑架构。

第 3 章旨在引导读者自主思考游戏内容和商业要素等，从而能够选择最合适的游戏架构。下面我们就来学习一些必要的背景知识。

3.1　游戏编程的特性——保持快速响应

作为对网络游戏架构的引入，本节首先来讨论一下游戏编程的一个重要特性：响应。在游戏编程中，对程序员来说，最重要的就是对响应速度的追求。

3.1.1　响应速度的重要性——时间总是不够的

视频游戏软件的最大特点就是：为了最大限度地发挥其可玩性，必须流畅地持续进行实时的高速处理。而且网络游戏的程序还必须始终保持高速响应。

为了始终保持实时的高速处理和稳定的高速响应，通常游戏程序会将所有必要数据都存放在内存中（on Memory）进行处理。“存放在内存中”指的是，在花费几个 CPU 时钟周期（几纳秒至几百纳秒）就能取得信息的距离内所配备的内存中存放数据。

除此之外，在开发网络游戏时还必须与网络数据包的“通信延迟”这一强敌作战。这里的延迟是纳秒的 100 万倍，也就是以毫秒为单位。为了不让这么高的延迟破坏游戏的乐趣，开发人员必须想尽各种方法来解决这个问题。

首先我们来考虑一下“为什么要将数据存放在内存中”。

3.1.2　将数据存放在内存中的理由——游戏编程真的有三大痛苦吗

之所以将数据存放在内存中，是因为游戏编程中存在被称为“三大痛苦”的三种特性，即：

❶ 游戏数据要在“16 毫秒”这一短暂的时间内持续变化。

❷ 大量对象的显示直接关系到游戏的可玩性。

❸ 不知道玩家会在什么时候进行操作，所以无法事先进行计算。

这些特性并非只出现在当今的游戏中，而是之前就已经存在了。正是因为这些特性，数据才需要存放在内存中，下面我们来依次了解一下这些特性。

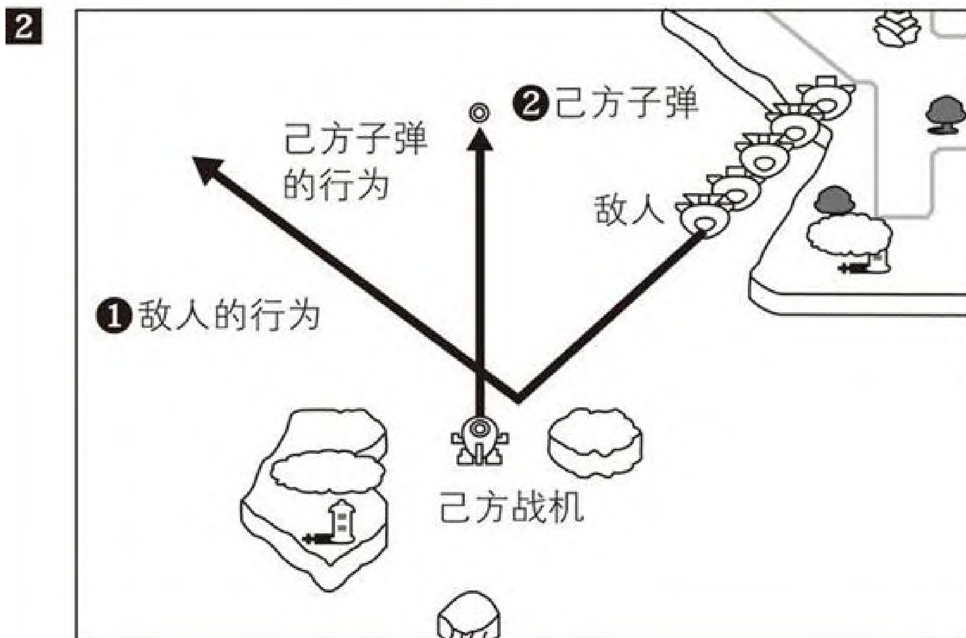
3.1.3 每 16 毫秒变化一次——处理的信息及其大小

这里以画面简洁明了的一款射击游戏为例进行说明（参见图 3.1），图 3.1 1是 KONAMI 公司（现更名为 KONAMI 数字娱乐）在 20 世纪 80 年代发售的一款家用机游戏《兵蜂》（*TwinBee* ）的游戏开始画面。

敌人（画面显示为红色）飞出来后如图 3.1 2- ❶ 所示的那样行动。玩家按下家用游戏机的控制键后，自己的子弹（图 3.1 2- ❷）就会在 1 秒内发射 30 次。显示器画面的帧速率（更新速度）是前一章所提及的每秒 60 次，程序的处理也是每秒 60 次，因为当时的家用机每一帧都要判断按钮的 On/Off 状态，1 次判定需要花费 2 帧，因此 1 秒内最多只能判断 30 次。

己方子弹以每帧 8 像素的速度在画面上方飞行，最多显示 2 个。敌方的移动速度为每帧 2 像素。击中敌人后加 100 分，而被敌人击中就算失败。当时的游戏机画面显示能 力有限，所以无法在画面上显示剩余的己方战机数。

图 3.1　《兵蜂》与游戏进程的相关要素



©1986 Konami Digital Entertainment

※ 1 图像提供：KONAMI 数字娱乐公司

http://www.konami-digital-entertainment.co.jp/

该游戏作为街机视频游戏发售之后移植到了家用机上，是一款颇具人气的射击游戏

表现游戏进程所需的信息及其大小

就图 3.1 所示的画面而言，为了表现游戏进程，所需的信息有以下这些。

- **己方战机（1 架）**
- **己方子弹（2 发）**
- **敌人（5 个）**
- **背景**
- **得分**

那么在以上这些方面，从程序的角度来看各自需要哪些信息呢？敌人的配置、打倒单个敌人时的得分表等，这些直接写在源代码中的固定数据，以及与背景相关的数据是不会发生变化的，所以在此不作说明。此外，得分也不一定每帧都会改变，所以也排除在外。那么，以下这些信息就是必不可少的。

- **己方战机（1 架）**
    - 坐标：因为是二维坐标 [1] ，所以需要 2 个 16 位 [2] 数据 → 4 个字节。
- **己方子弹（2 发）**
    - 坐标：二维坐标，需要 2 个 16 位数据 → 4 个字节 ×2 个子弹＝ 8 个字节。
    - 速度：二维坐标，需要 2 个 16 位数据 → 4 个字节 ×2 个子弹＝ 8 个字节。
- **敌人（5 个）**
    - 坐标：二维坐标，需要 2 个 16 位数据 → 4 个字节 ×5 个敌人＝ 20 个字节。

- 速度：二维坐标，需要 2 个 16 位数据 → 4 个字节 ×2 个敌人＝ 20 个字节。
- 计数：因为在某个时刻敌人会返回（路线折回），所以在敌人个数不固定的情况下需要进行计算。需要 8 位数据 → 1 个字节。

> [1] 坐标值在 0～255 的范围内，分为 *x* 坐标和 *y* 坐标，所以需要 2 个 16 位数据。除了己方战机，己方子弹、敌人也都是如此。

> [2] 因为子弹、敌人、己方战机的坐标和速度变化及其细微，所以 16 位（2 个字节）的数据会精确到小数点。

由此可知，共计 4+8+8+20+20+1=61 个字节就可以将该图中的游戏内容表现出来。看到这个数字，读者或许会觉得“还挺少的”吧。

可以用 RDBMS 实现吗 ？——与在内存中存放数据进行比较

这里稍微岔开一下话题，谈谈在Web服务和业务系统中广泛使用的 RDBMS（relational database management system，关系型数据库管理系统），将其与在内存中存放数据这种方式进行一下比较。下面是一种作为网络游戏开发框架来实现（继而以失败告终）的方法，假设这里用 RDBMS 来实现，我们暂且定义一个表结构，如表 3.1 所示。简单起见，我们只定义了 1 个表。由于出现在画面上的物体都是在空中飞的，所以我们将表命名为“FlyingObjects”。

表 3.1　FlyingObjects 表结构

列名	类型	内容
OBJECTTYPE	byte not null	飞行物体的类型
X	byte not null	X 坐标
Y	byte not null	Y 坐标

列名	类型	内容
DX	byte	X 方向上的速度
DY	byte	Y 方向上的速度
COUNTER	byte	计数

表 3.2　实际的表内容

对象类型	X	Y	DX	DY	COUNTER
己方战机	120	120	null	null	null
敌人	192	48	-4	4	20
敌人	184	56	-4	4	22
敌人	176	64	-4	4	24
敌人	168	72	-4	4	26
敌人	160	80	-4	4	28
己方子弹	120	80	0	-12	null
己方子弹	120	10	0	-12	null

针对图 3.1 的画面状态，实际的表内容如表 3.2 所示。其中有两点很重要。

- **所有的物体在每一帧中都是持续运动的。**
- **下一帧会发生什么是无法预测的。**

FlyingObjects 中的所有对象在每一帧中都是持续运动着的。因此，X、Y 的值会不断变化。另外，因为无法预测玩家会在哪一帧中作出怎样的操作，所以无法将事先计算好的数据存入表中。因此，每一帧都要取出表中的所有行，进行判断，然后再全部存入表中，如此反复。这些操作必须在 1 秒内重复 60 次，也就是每 16 毫秒重复一次。

RDBMS 本来并不是刻意设计成这种使用方式的。尽管如此，比如现在的 MySQL，对于每行几个字节的数据，每秒能更新 1 万～10 万次以上。也许有人会认为能做到这样的话就没问题了，然而果真如此吗？

关于这一点将在下一节进行深入介绍。这里首先需要注意“游戏进程的数据每 16 毫秒变化一次”，请牢记这一点再往下看。

3.1.4　大量对象的显示——CPU 的处理能力

接下来我们看一下 CPU 的处理能力。根据摩尔定律，CPU 的晶体管数量大约每隔 18～24 个月就会翻一倍。晶体管数量基本上与计算机的性能直接相关。虽然也有人说摩尔定律很快就要失效了，但毕竟这条定律迄今为止已成立了将近 40 年。

家用游戏机与 CPU 周期

1984 年左右，任天堂家用游戏机（HVC-001）采用的是 6502（MOS 6502）的兼容芯片（处理器），主频 1.79MHz。这款 25 年前的 CPU 与现在的芯片比起来，足足慢了 2 的 12 次方，也就是 4096 倍。

单单说 4096 倍也很难想象，我们来具体地分析一下。如果使用这种芯片，数据在 1 秒内变化 60 次的情况下，CPU 处理能力大约是 1 次（1 帧）179 万 /60 ＝ 29666，也就是大约 3 万周期。

为了让飞行物移动起来，必须进行以下处理：读取坐标、读取速度、进行计算、保存结果。使用 6502 处理器的命令集处理以上操作需要 2～8 个周期，假设在效率较高的程序中需要 4 个周期，如果飞行物有 8 个，就要对二维坐标中 X、Y 两方数据各处理 1 次，共计两次，由此可知，总共需要（4+4+4+4）×2×8 ＝256 个周期。对于 3 万周期来说可谓十分充裕。

接着需要判断己方子弹是否击中了敌人。假设要对正在飞行的 8 个物体全部进行碰撞检测，那就必须进行 8×8 ＝ 64 次检测。碰撞检测不能根据两个物体的坐标是否一致来进行判断，而是必须通过矩形来判断（这一点将在后面详述），所以需要进行二次比较。因此必须进行如下处理：读取坐标、二次比较，以及根据结果进行不同的处理。这次每项处理需要 10 个周期。又因为要处理 X、Y 坐标两方数据，所以还要翻一倍。最终需要（10+10+10+10）×2×（8×8）＝ 5120 个周期。再加上移动处理的 256 个周期，也就 5400 个周期不到，看上去还挺充裕吧。

家用游戏机上的经典游戏通常有 20 个左右的角色登场。若对 20 个对象进行碰撞检测时，根据上面的计算方式，需要（10+10+10+10）×2×（20×20）= 32000 个周期，这就超过了 3 万周期的上限。另外还有其他操作以及声音等的处理，所以必须采取一些措施来降低所消耗的周期。比如，把“己方子弹之间不会发生碰撞”、“敌方子弹与己方子弹不会发生碰撞”等游戏内容考虑在内，从而对内存进行优化。

实际上，为了增强游戏的可玩性，一般会根据情况使物体运动更为复杂，而不是简单的匀速直线运动，这样一来，CPU 周期就更显不足了。一旦 CPU 周期不足，游戏中物体的运动就会显得迟缓，导致游戏体验急剧下降。而另一方面，如果 CPU 周期过于空闲，游戏的进行速度就会显得过快，需要追加一些额外的“为了配合显示设备的渲染更新速度稍作等待”的处理。

- 在家用游戏机中使用 RDBMS 时

假如要在搭载了 6502 芯片的家用游戏机上使用 RDBMS 会怎么样呢？当然首先必须通过 SQL 语句，但是像 SELECT * from FlyingObjects 这样的语句，单单判断语法是否正确就要消耗几百个 CPU 周期，显然不现实。

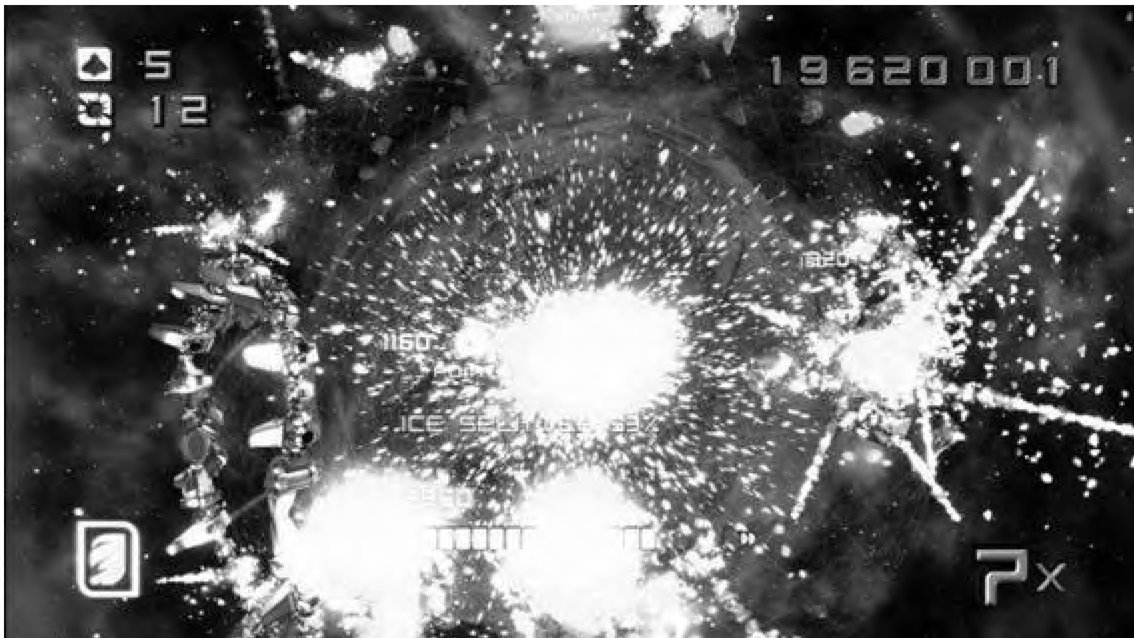
游戏编程必须在 1 帧内完成坐标的判断和保存。为此，必须只通过组合 CPU 所具有的一些最原始的命令来实现这些处理，只是读取数据就要花费几百个周期是相当不合理的。因此，在家用游戏机中，基本不考虑使用 RDBMS 这种方式。

PlayStation 3（PS3）与 CPU 周期 ——可以用 RDBMS 来开发吗

那么如今的游戏机又如何呢？PlayStation 3（PS3）自诩具有卓越的性能。主频 3.2GHz 的 8 核处理器，128bit 访问总线，确实是优势明显的。有了这样的性能，或许可以使用 RDBMS 的方式来开发呢……那么，真是这样吗？

图 3.2 是 PlayStation 3 的一款游戏《星际出击 HD》（*STAR STRIKE HD* ）的画面。在画面中我们可以看到许多细小的粒子，这些粒子实际上使用了绿色、粉色、橘黄色等颜色，虽然《星际出击 HD》与《兵蜂》一样都是射击游戏，但是闪烁的粒子过多，几乎都看不到己方的战机了。

图 3.2 《星际出击 HD》



©2007 Sony Computer Entertainment Europe.

Published by Sony Computer Entertainment Inc.All Rights Reserved.

Developed by Housemarque.

每一个粒子的运动轨迹都比匀速直线运动复杂，而且还是在三维空间内运动的。在这个游戏中，单单需要进行碰撞检测的对象就高达数千个，除此之外还有数万个以上的粒子要以每秒 60 次的速度四处移动，为了进行这样的移动，必须通过三阶行列式来进行计算。

在主频 3.2GHz 的设备中，1/60 秒内可以利用 5300 万的 CPU 周期。假设每次有 2000 个物体互相碰撞，总共就是 2000×2000 = 400 万、5300 万 / 400 万 = 13.25 个周期。在三维空间内运动的对象进行 1 次碰撞检测就要 13 个周期，这对于现在的设备是不可能的，所以需要运用空间分割等各种优化手段。所以对于在 PlayStation 3 中是否可以使用 RDBMS，还有很多方面是必须要考虑的。接着我们继续往下看。

3.1.5 无法预测玩家的操作——游戏状态千变万化

在对象会发生碰撞的游戏中，一般来说，“物体的行为比较混乱”，所以无法预测数据何时会发生变化。而且玩家什么时候会进行操作也是完全无法预测的。在《星际出击 HD》这个例子中，无法预测几千个对象在下一帧会移动到何处。

用 RDBMS 方式无法实现的信息量和处理速度

假设玩家可以作出的选择比较少，而可能的模式又很有限，虽说只要事先保存所有的状态，只在需要时读取出来就可以了，但是因为数据的变化每次都不可预测，结果还是必须在每 1/60 秒（16 毫秒）内从数据库中读取几万行的数据进行重新计算。无论 MySQL 的速度有多快，都无法做到以这样的速度读写。

如果不要显示这么多对象就好了……话虽如此，但是显示大量的对象可以提升游戏的实时性和画面的精美度，所以为了在市场上保持竞争力无论如何也要做到这一点。虽说市场上可能也有一小部分使用少量对象的游戏……不仅是单机游戏，网络游戏也同样如此。

3.1.6　必须将游戏数据放在 CPU 所在的机器上

至此，我们已经了解了游戏编程的 3 大根本特性：❶ 游戏数据每 16 毫秒变化一次，❷ 大量对象的显示很多情况下都直接关系到游戏的可玩性，❸ 不知道玩家会在什么时候进行操作，所以无法事先进行计算。由此，读者应该能够理解为什么要将数据存放在内存中进行处理了吧。这些特性并不依赖于如今的时代，所以游戏程序员需要具备“通过在内存中处理数据，最大限度地节约 CPU 周期，使大量的对象能够持续运动”的技术能力。

这里再次对上述内容进行一次总结：游戏过程中的数据需要以非常快的速度不断变化，所以这些数据必须在内存中进行管理。从支持网络游戏的技术这一层面来说，从这一点可以引出更为重要的问题。在内存中进行管理的关键是注意 CPU 频率，也就是要在几纳秒至几百纳秒的延迟内访问数据。在光速下就是 1 纳秒 30 厘米的距离。

也就是游戏数据需要存放在距离 CPU 数十厘米以内的地方，也就是存放在 CPU 所在的机器中。但是在实际中，玩家通常分散在各处，甚至相距几千公里以上。由于玩家与数据中心相距几公里，乃至几千公里，所以必然要在较远处和较近处（数 10cm）之间冗余地保存数据，以此保证数据的一致性。这一点非常困难！那么下面我们就来探讨一下这一问题的解决方案。

3.2　网络游戏特有的要素

本章开头提到了一些网络游戏特有的要素：通信延迟、带宽、客户端、服务器、安全性以及网络游戏的辅助系统。本节就将对此逐一加以说明。

3.2.1　通信延迟——延迟对游戏内容的限制

比如，为了使用同步方式来实现动作性很高的游戏（详细内容将在后面介绍），必须采用 LAN 或者 ad-hoc 等数据包通信延迟非常小的物理网络。“延迟非常小”是指要小到什么程度才够呢？

互联网是通过几百万、几千万的路由互相连接而成的网络集合体。我们来利用软件测量一下通过互联网收发数据包的往返时间吧。这里用了 ping 命令。笔者是在东京撰稿的，我们来看两个与大学服务器连接的例子，大学服务器都是位于大学校内的 [3] 。图 3.3 是 ping 京都大学的例子，最快用了 22.646 毫秒。图 3.4 是 ping 大阪大学的例子，最快 19.097 毫秒。

> [3] 慎重起见使用了 traceroute 命令确认服务器位于关西地区。

图 3.3　测量数据包的往返时间（东京 - 京都大学）

```
$ ping www.kyoto-u.ac.jp
PING www.kyoto-u.ac.jp (130.54.120.215): 56 data bytes
64 bytes from 130.54.120.215: icmp_seq=0 ttl=44 time=22.646

 ms
64 bytes from 130.54.120.215: icmp_seq=1 ttl=44 time=23.102 ms
^C
--- www.kyoto-u.ac.jp ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 22.646

/22.874/23.102/0.228 ms
```

图 3.4　测量数据包的往返时间（东京 - 大阪大学）

```
$ ping www.osaka-u.ac.jp
PING www.osaka-u.ac.jp (133.1.8.18): 56 data bytes
64 bytes from 133.1.8.18: icmp_seq=0 ttl=47 time=21.673 ms
64 bytes from 133.1.8.18: icmp_seq=1 ttl=47 time=19.097

 ms
^C
--- www.osaka-u.ac.jp ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 19.097

/20.385/21.673/1.288 ms
```

传输时间的具体内容

我们以图 3.4 为例看一下 19 毫秒内的传输情况。

光的速度是 30 万千米 / 秒，单模光纤的折射率为 1.5，30 万 /1.5 = 20 万千米 / 秒。东京与大阪之间的距离是 500 千米，往返一次就是 1000 千米。光速往返一次需要花费 1000/200 000 = 0.005 秒，也就是用了 5 毫秒。

笔者家中与大阪大学之间配置有 20 台以上的路由软件和硬件，剩下的 14 毫秒就用在了这些路由的处理上。但是其中有 1 毫秒左右是由笔者所用笔记本电脑的无线 LAN 所消耗的。

无法避免的延迟 ——延迟与游戏类型

东京与大阪之间铺设了日本最快的光纤。笔者更是用了非常高速的 OCN 光纤，而大学的网络也是相当快的，所以由此测量结果可知，在日本，东京与大阪的玩家进行游戏对战时，游戏数据经由数据包进行传输时，至少会有 19 毫秒的延迟。

19 毫秒比显示设备 16.7 毫秒（16 毫秒）的显示速度慢，即使面向日本互联网市场的网络游戏可以做到最大限度地利用显示设备的显示速度，但是东京和大阪的玩家进行对战时还是无法达到这一速度。由此可见，网络游戏并不适合那种挑战反应神经的游戏。

笔者又试着从自己的家中 ping 位于东京的某一台服务器，得到了如图 3.5 所示的结果，这次只用了 7 毫秒，可谓极其高速。

图 3.5　测量数据包的往返时间（东京 /OCN - 东京）

```
$ ping www.ce-lab.net
PING www.ce-lab.net (61.194.89.196): 56 data bytes
64 bytes from 61.194.89.196: icmp_seq=0 ttl=51 time=7.702 ms
64 bytes from 61.194.89.196: icmp_seq=1 ttl=51 time=7.962 ms
64 bytes from 61.194.89.196: icmp_seq=2 ttl=51 time=7.233 ms
^C
--- www.ce-lab.net ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 7.233/7.632

/7.962/0.302 ms
```

随后又尝试了通过 EMOBILE[4] 进行连接，如图 3.6 所示，即使是最快的一次也用了 68 毫秒。相当于 16.7 毫秒的 4 倍左右，也就是花费了 4 次更新显示内容的时间。

[4] EMOBILE 是日本的一个移动宽带品牌，提供 3G/HSPA+/DC-HSDPA/LTE 服务。——译者注

图 3.6　测量数据包的往返时间（东京 /EMOBILE - 东京）

```
$ ping www.ce-lab.net
PING www.ce-lab.net (61.194.89.196): 56 data bytes
64 bytes from 61.194.89.196: icmp_seq=0 ttl=48 time=89.294 ms
64 bytes from 61.194.89.196: icmp_seq=1 ttl=48 time=69.001 ms
64 bytes from 61.194.89.196: icmp_seq=2 ttl=48 time=68.864

 ms
64 bytes from 61.194.89.196: icmp_seq=4 ttl=48 time=78.422 ms
64 bytes from 61.194.89.196: icmp_seq=5 ttl=48 time=78.342 ms
64 bytes from 61.194.89.196: icmp_seq=6 ttl=48 time=88.047 ms
^C
--- www.ce-lab.net ping statistics ---
7 packets transmitted, 6 packets received, 14% packet loss
round-trip min/avg/max/stddev = 68.864

/78.662/89.294/8.069 ms
```

最后又 ping 了美国的服务器，到东海岸的大学最快用了 163 毫秒。往返大约 2 万 5000 千米，光速传输就需要花费 125 毫秒，再加上路由处理所需的时间，基本不可能更快了。

图 3.7　测量数据包的往返时间（东京 - 美国）

```
$ ping www.mit.edu
PING www.mit.edu (18.7.22.83): 56 data bytes
64 bytes from 18.7.22.83: icmp_seq=0 ttl=233 time=163.316

 ms
64 bytes from 18.7.22.83: icmp_seq=1 ttl=233 time=173.139 ms
64 bytes from 18.7.22.83: icmp_seq=2 ttl=233 time=184.068 ms
^C
--- www.mit.edu ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 163.316

/173.508/184.068/8.476 ms
```

物理规律不会变化，只要地球的大小保持现状，日本玩家和美国玩家就不能进行挑战反应神经的视频游戏。

在实际进行游戏开发时，需要根据目标平台和地区，像上面这样在多个时间段内进行 ping 测定，确认其结果是否处于 2.8 节介绍的各种游戏类型所允许的延迟范围之内，由此考虑游戏的具体内容之后再着手开发。

3.2.2　带宽——传输量的标准

接下来，我们来看一下网络游戏的另一个特有要素：带宽（传输带宽）。如前所述，在游戏中，每次进行数据处理时，都会有大量的状态发生变化。如果可以，最好尽可能毫无遗漏地传输这些状态变化。但是如果传输量过多，从商业角度来看就很不合算了，不仅如此，即使采用 P2P 通信，也可能中断与 ISP 的通信，引起玩家不满。

为此，一般需要将传输量控制在一定的范围内。

- **C/S MMO 的情况下：每人 10kbit/s～100kbit/s。**

- **P2P MO 的情况下：30kbit/s～300kbit/s。**

特别是在 MMOG 的情况下，花点儿心思就有可能将传输量控制在 10kbit/s 以下。带宽是非常重要的，所以在后面的章节中我们还会详细说明如何通过各种方式来进行评估以及具体要采用怎样的方法。

3.2.3　服务器——成本、服务器数量的估算

用于 C/S MMO 游戏服务和 P2P MO 游戏辅助系统的服务器越是充裕，开发人员就越轻松。但是实际上并不能这么做。虽说可以利用各种云服务，但是每个月的成本还是相当高的。对网络游戏所需的服务器数量的估算大致如下。

- **C/S MMO：每台服务器有 1000～3000 个同时连接，那么服务器数量预计等于设想的同时连接数 /1000～3000 个同时连接。**

- **P2P MO：每台服务器的同时连接数相当于上面的 3～5 倍，计算方式同上。**

如果设计目标大幅低于上述水平，从运营成本来看很难获利。

为了减少服务器数量，技术人员首先应该做的就是进行估算。通过这样的估算，可以对成本进行评估，讨论在哪些方面降低成本。具体的估算方法将在第 4 章之后详细介绍。

3.2.4 安全性——网络游戏的弱点

网络游戏的结构在面对非法行为时显得非常薄弱。用户可以通过传输线路收发“破坏了的数据”和“恶意数据”，而这些有问题的数据会经由网络渐渐扩散开来。

作弊 ——最大的安全隐患

在各种非法行为中，最大的安全隐患就是称为“作弊”（Cheat）的行为。作弊就是指通过某种方式非法利用构成网络游戏的系统。绝大多数的作弊行为都会降低游戏内容的吸引力，所以游戏运营商会尽可能防止作弊行为。这在网络游戏中是一个很典型问题，所以这里详细地讨论一下。

作弊行为很多情况下都是玩家出于个人利益的目的而进行的，主要有以下这些动机。

- **纯粹的个人利益**
    - 想要加快游戏进度。
    - 作为加快游戏进度的结果，想要通过 RMT 赚钱。
    - 出于好奇，单纯地想要尝试一下作弊行为。
    - 通过开发作弊工具加深自己的技术知识。
- **与其他玩家相关的个人利益**
    - 想要比其他玩家以更有利的方式进行游戏，从而处于有利地位。
    - 想要偷看其他玩家的行为。

    - 觉得妨碍其他玩家进行游戏是件很愉快的事情。
    - 想要向其他玩家炫耀自己的作弊技术。
    - 对某个玩家心存反感，想要报复对方。
- **与运营公司相关的个人利益**
    - 对运营公司心存反感，想要报复。
    - 想要盗取互相竞争的游戏产品的技术。
- **公司利益**
    - 成立公司，开发高级的作弊工具，想要通过 RMT 获取利益。
- **非有意地造成损失**
    - 误用了作弊工具。
    - 感染病毒，安装了会自动作弊的程序。
    - 不知道是程序 bug 而多次利用了该 bug。

由于以上的这些动机，有些玩家获得了利益，有些玩家遭受了损失。作为网络游戏的技术人员，首先应该致力于研发不易进行作弊的环境，如果能做到这一点，就自然能减少那些并非出于玩家本意而造成的损失。因此这里只讨论故意进行的作弊行为。

为什么要作弊

如前所述，作弊行为是玩家为了获得某些利益而进行的。可以说，作弊是一种经济上的行为。玩家为了获得想要的东西，使用并非出于游戏提供方本意的或者使用游戏提供方禁止的方法，通过大多数玩家都无法使用的方式获得游戏上的利益。绝大多数情况下，游戏上的利益指的都是通过 RMT 进行售卖，由此获得与实际货币等值的利益。通过作弊行为所能获得的价值越大、同时所需为此付出的成本越小，作弊行为的诱惑就越大。作弊所获利益越大，进行作弊行为的玩家就会越多。

因为作弊是一种经济上的行为，所以可以用以下这种简单的方式来表示作弊所得利益。

作弊所得利益 = 通过作弊所获得的利益－作弊所需花费的成本

因此，减少作弊行为的措施大致可以分为“降低作弊所得的利益”和“增加作弊所需花费的成本”这两种。

作弊行为的手段

首先列举一下作弊行为的手段。

- 内存破解

  直接篡改终端的内存上所存储的游戏过程数据。

- 数据包破解

  使用某些工具篡改游戏程序所收发的数据包的内容。

- 数据文件破解

  篡改游戏程序所读取的文件的内容。

- DDL（Dynamic Link Library）破解

  对游戏程序启动时所读取的动态链接二进制文件进行篡改。

- 时钟破解

  将操作系统的时钟设置为与当前不同的时间，使程序作出错误的行为。

- UI 工具破解

  鼠标点击和键盘操作等游戏必不可少的操作通过自动化工具反复高速地进行，而无需人工操作。

- 服务器攻击

非法侵入服务器，偷看、篡改服务器端数据。

- **伪客户端**

  制作假的游戏程序，生成、收发正常的游戏程序不可能具有的数据。

此外，还有一些虽然不是作弊，但也存在问题的行为。

- **违反规则**

  正常使用游戏程序，但是违反游戏规则。

- **非法利用 bug**

  不管什么游戏都有一些广为人知的 bug。利用这种 bug 获取大量利益。

- **给服务器造成极大负担［DoS、Denial of Service attack（拒绝服务攻击）］**

  通过向服务器和其他玩家的终端发送极其大量的数据包、反复登录，造成超负荷。处于超负荷状态下的服务器运行状态变得很不稳定，这些入侵者就利用这一状况复制（Dupe，将在后面介绍）游戏物品。

- **滥用隐藏命令**

  滥用调试或者开发用的命令。与非法利用 bug 类似。

作弊的操作对象

使用上述手段作弊，不管是哪一种，操作对象无外乎以下几种。

- **本地的内存和文件**

  试图作弊的玩家自己终端上所保存着的资源。

- **其他玩家的内存和文件**

没有进行作弊的其他玩家所使用的终端上保存着的资源。

- **数据中心服务器上的内存和文件**

  运营公司所有的服务器上的资源。

- **存在于本地与其他玩家之间的数据包**

- **存在于本地与数据中心的服务器之间的数据包**

游戏运营方认为安全的 CPU 只有“数据中心服务器上的 CPU”，除此之外的 CPU 对进行作弊行为的玩家都无能为力。

比如，即使采取了将文件加密后再发布的措施，能够有效防止的作弊行为仍然非常有限。根据经验来看，1000～2000 个玩家中只要有 1 个程序员，就能制作出作弊用的工具。该玩家可以将加密文件进行解密后从内存中取出。这样的工具一旦通过网络传播开，大量玩家都能很容易地看到加密文件中的内容。

这种恶性循环虽然依赖于通过作弊所得经济价值的大小，但是在拥有数万玩家的游戏中，2 天～1 周内就会造成很大的损失。

冯诺依曼型计算机的宿命 ——防止作弊行为的服务

现在也有一些企业提供专门的服务来尽可能降低这些作弊行为所造成的影响 [5] ，在他们的产品中囊括了嵌入在游戏程序中使用的库、通过传输交换密钥的服务，以及制作新补丁的服务。也就是说，并不仅仅是给数据加密，还可以将一些重复工作交给外部企业去做。这些企业还提供一项服务：在发生新的作弊行为的 24 小时以内开发补丁，提供开发库。

> [5] GameGuard（http://gameguard.nprotect.com/ ）等。

与这类企业签订合同的话，每月的成本大约在数十万日元以上。反之，即使花费很大的成本，游戏运营公司也有防止作弊行为的动机。实际上，用于使防卫服务本身无效的工具仍然一直在开发中。

只要还在使用冯诺依曼型的计算机，这种循环往复的工作就会永远持续下去。

恐怖……作弊行为的影响

如上所述的在玩家的终端上实行的作弊行为原则上无法防止，那么实际进行的作弊行为对游戏玩家群体来说会有怎样的影响呢？下面根据影响的程度列举了 3 个例子。

❶ 流通非法物品

→在进行游戏时，如果玩家所持有的物品（比如能给对手造成伤害的弹药）在游戏中起到的作用很大，那么通过非法篡改所持有的弹药，或者强度异常的物品来获得非法物品的情况下，再将这些非法物品交给其他玩家，非法的数据就会在玩家和玩家之间进行传播，就像流行性病毒那样急速扩散。

在这种情况下，玩家团体就会以这类物品的存在为前提制定游戏策略，也就是说，会对其他物品的使用方式和游戏方式造成影响。因此，即使之后对这些非法物品进行了处理，但是因为此时游戏整体已经受到了影响，这样反而会引起玩家的排斥。

❷ 流通非法的身份角色

→如果能交换物品，作弊行为的影响就有变大的倾向，所以有很多游戏不支持物品交换。但是即使在这种情况下，还是可以通过使用非法角色开启多个客户端进行游戏，从而对其他玩家造成不好的影响。比如，不正当地使用强大的角色帮助自己的其他角色练级，从而更快地让那个角色强大起来。这种情况会给实际的游戏玩家团体的经济造成很大的影响。

❸ 积累非法的游戏结果

→不支持多角色的游戏也有很多，但是在这种情况下，也可能使用非法的角色获取不正当的胜利以及游戏成绩。如果游戏成绩是作为等级来保存的，就会给对战游戏的玩家匹配造成恶劣影响，最终导致游戏体验变差。

这些影响的关键之处在于，为了减小作弊行为的影响，必须减少多人游戏中有趣的物品交换和角色之间相互作用的要素，以此获得平衡。

3.2.5 辅助系统（相关系统）

为了让网络游戏的“游戏程序主体”正常地运行、让玩家始终保持良好的心情专心进行游戏，还需要玩家匹配机制、等级机制、收费认证等各种支持游戏程序的“辅助系统”。

有关辅助系统的内容将在第 6 章进行讨论，详细内容请参照该章所述。

3.3 物理架构详解——C/S 架构、P2P 架构

下面我们来看一下网络游戏的物理架构。第 0 章和第 2 章已经提到过，物理架构可以大致分为 C/S 架构和 P2P 架构两种。

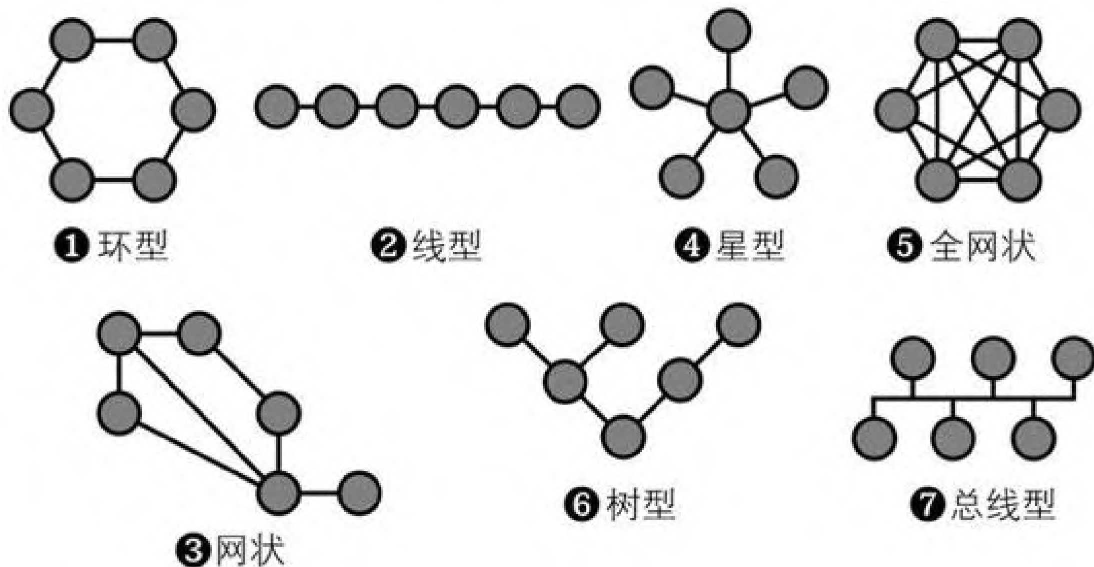
3.3.1 基本的网络拓扑结构

在开始讨论具体的架构之前，必须对网络拓扑结构（Network Topology）有所理解，网络拓扑结构是在考虑网络游戏的架构时，用来具体思考网络相关问题的一种工具。所以我们首先来了解一下网络拓扑结构。

网络拓扑结构是指构成网络的要素之间的连接形状，有助于研究网络的连接要素是以怎样的结构加以构建的。

网络拓扑结构有图 3.8 所示的这些。图中，圆形部分称为节点（Node），实线称为边（Edge），或者线路、链路。在网络游戏中，节点就是 PC 和服务器等各种计算机设备，线路就是指网络连接。如果从某个节点到另一个节点经过了两条线路，就称为“拓扑数为 2”。图 3.8 中 ❶～❼ 这几种结构的特点如下所述。

图 3.8 网络拓扑结构



❶ 环型（Ring）

形成一个环状，即使一条线路中断，还是能使用反方向的线路将信息传输到所有的节点中。从一个节点到另一个节点的平均拓扑数为所有节点数的一半。

❷ 线型（Line）

从一端的节点开始以管线方式依次传输信息，包括缓存方式在内，单纯地维持各节点的运行方式。从任意一个节点到另一个节点的拓扑数可能最长（节点数－1）。

❸ 网状（Mesh）

多个节点由不规则的线路连接在一起。从一个节点到另一个节点的线路有多种。需要注意很多方面，比如不要将负荷集中在某个特定的节点上。

❹ 星型（Star）

多个节点连接到一个特殊的中央节点上。从一个节点到另一个节点必然经过两条线路。另一方面，中央节点的处理负荷很高。

❺ 全网状（Full Mesh、Fully connected）

所有的节点全部互相连接。因此一个节点到另一个节点都只需要经过 1 条线路。这种连接方式并不适合那些线路维护成本很高的网络。

❻ 树型（Tree）

将信息传输给所有节点时，根据作为信息发送方的节点所处位置的不同，拓扑数也会相应变化。

❼ 总线型（Bus）

多个节点由一条共同的总线（Bus、传输线路）连接。也被认为是对星型结构的应用，但是不存在中央节点，因为只是简单地复制信息，所以中央节点就被整个省略了。

实际使用的有星型（和总线型）、全网状结构 ——把通信延迟降至最低

如上所示，网络系统的实现可以采用各种各样的拓扑结构，但是本书所介绍的网络游戏的实现中，所用到的基本拓扑结构只有星型及其应用型总线型，还有全网状结构。

P2P 架构常用全网状结构（如同步方式的实现）、星型结构（如存在主机的游戏的实现）、总线型结构（如反射型的实现）。C/S MMO 架构则常用星型结构。

采用这些结构主要还是因为，相对于游戏中的容错性和整体的吞吐量，优先级最高的还是“尽可能降低通信延迟”。

比如，在星型结构中，如果中央节点被破坏了，那么整个网络就会全部中断。全网状结构中，虽然整体的传输量最大，但是确实只要经过一条线路就能将信息送至目标节点，所以速度最快。环型、线型、网状、树型结构在节点与节点之间都存在两条以上的线路，通信延迟过大，所以不予使用。

因此，目前的情况就是只使用“星型结构”（及作为其应用的总线型结构）和“全网状结构”。

3.3.2　物理架构的种类

本书所讨论的网络游戏的物理架构以 C/S 架构、P2P 架构为主。这里将实际在网络游戏中使用的架构分为以下几种进行说明。

- C/S 架构（客户端 / 服务器架构）
- P2P 架构
- C/S ＋ P2P 混合型架构
- ad-hoc 模式

3.3.3　C/S 架构——纯服务器型、反射型

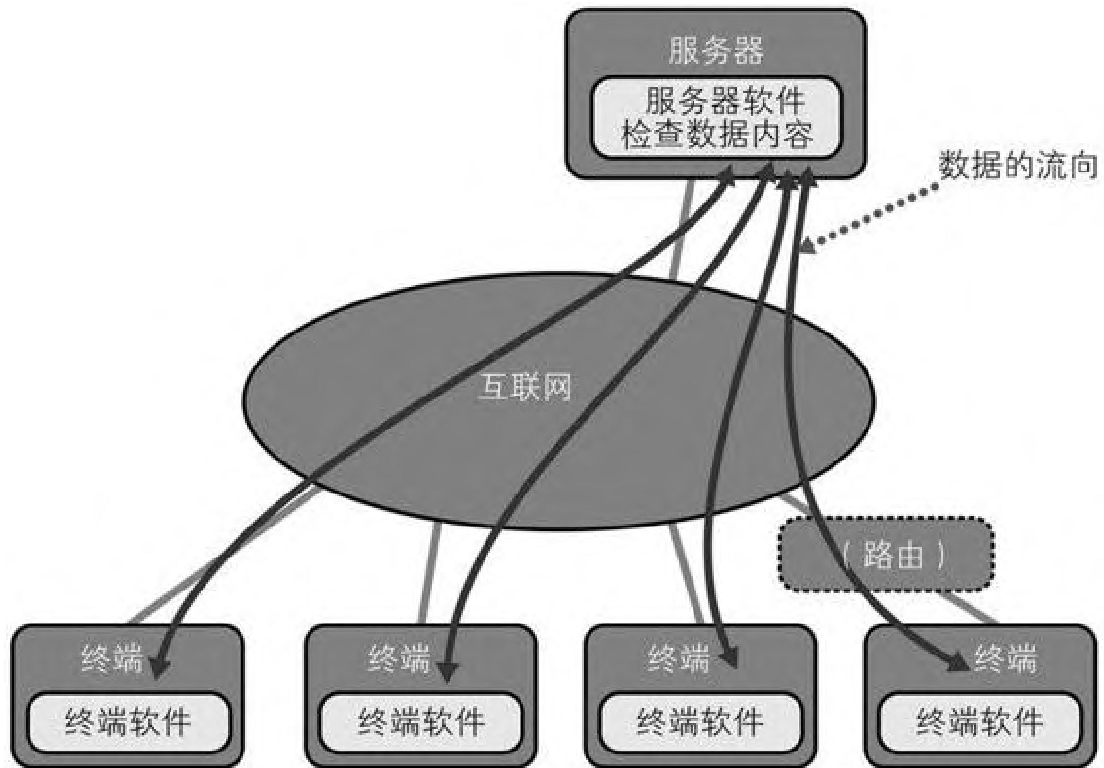
在 C/S 架构中，数据中心中的专用服务器（Dedicated Server）由运营方所有，通过这个专用服务器来传输游戏程序之间的数据（参见图 3.9）。这种结构与一般的 Web 服务完全相同。

C/S 架构根据服务器功能的不同，分为 ❶ 纯服务器型、❷ 反射型两种类型。图 3.9 所示的是纯服务器型。

在 C/S 架构中，因为要从终端软件，也就是游戏客户端向服务器提出连接要求，即使终端配置在带有防火墙的路由器内部，也可以顺利地确立连接。

图 3.9 所示的服务器上所运行的服务器软件将会对各个客户端发送而来的数据进行检查，判断是否包含非法数据，然后拒收那些不符合规则的数据。

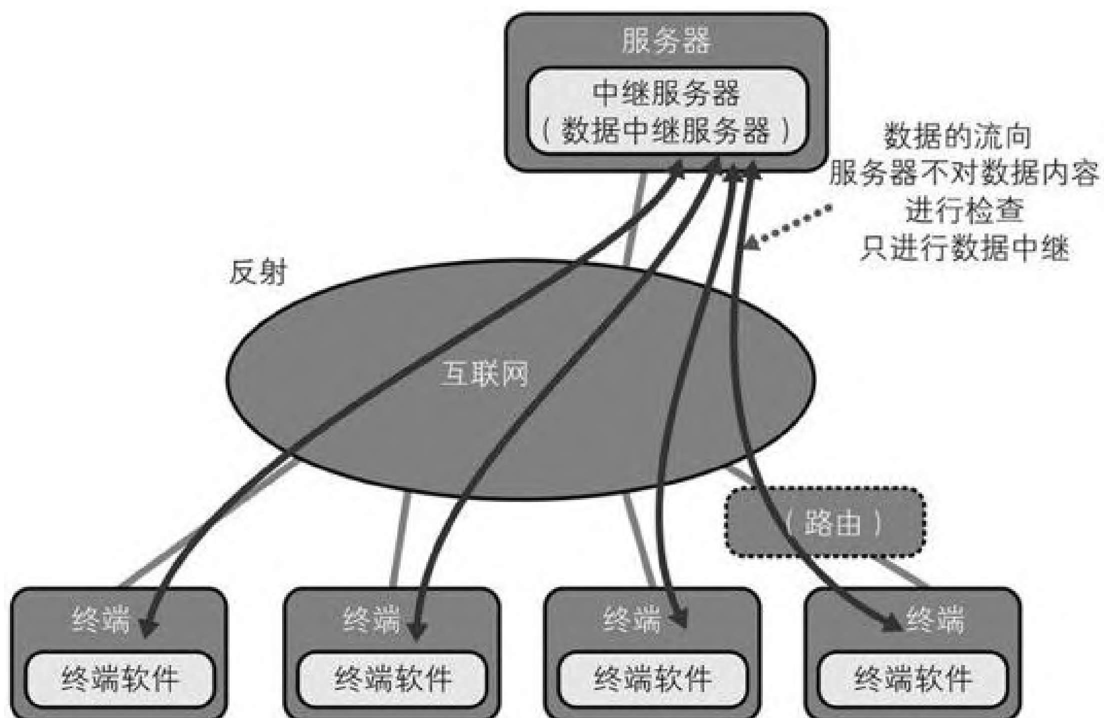
图 3.9　C/S 架构（纯服务器型）



反射型

图 3.10 所示的是反射型。单从数据包的流向来看，这种结构与纯服务器型相同。但是，在这种结构中，服务器上运行的服务软件只进行数据包的交换，并不检查其中的内容。使用这种结构的话，就不必为每个游戏都准备一个独立的服务器了。这种单功能的中继服务器（将在后面讨论）也被称为“无需编程的服务器”、“超节点”（Super Node）。

图 3.10　C/S 架构（反射型）



由于服务器不会对数据包内容的合法性进行确认，所以怀有恶意的玩家可以很容易地将非法数据包发送给其他客户端。

3.3.4 P2P 架构

P2P 架构有两种方式，“同步方式”和“异步方式”。同步方式有全网状结构和星型结构两种，它们的数据流向有所不同，如图 3.11、图 3.12 所示。在终端配置在带有防火墙功能的路由器内部（图 3.11、图 3.12 右侧所示）的情况下，必须进行特殊处理，这一点将在后面介绍。

在 P2P 架构中，为了控制数据的流向，在数据中心不必设置特殊的设备。

图 3.11　P2P 架构（全网状结构）

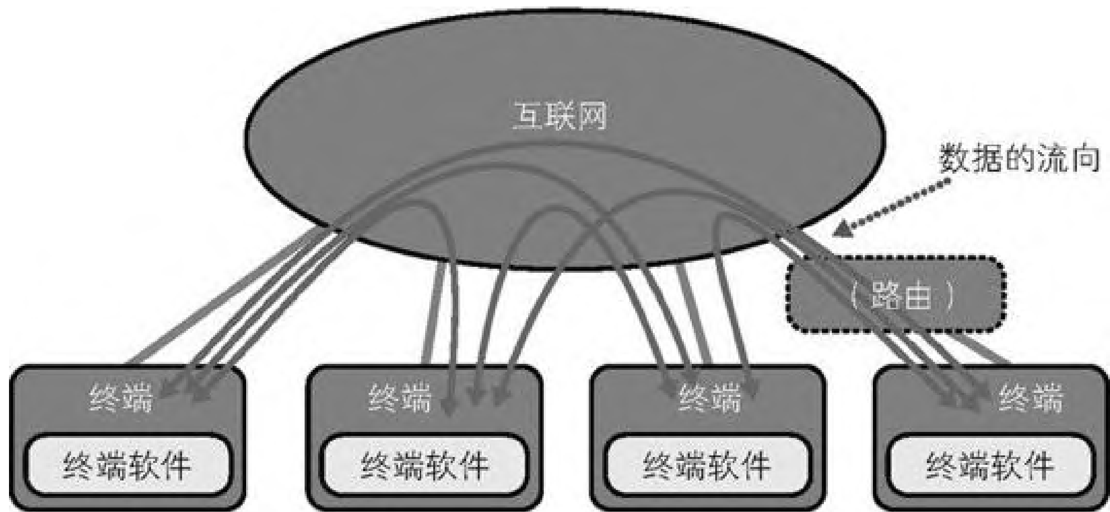
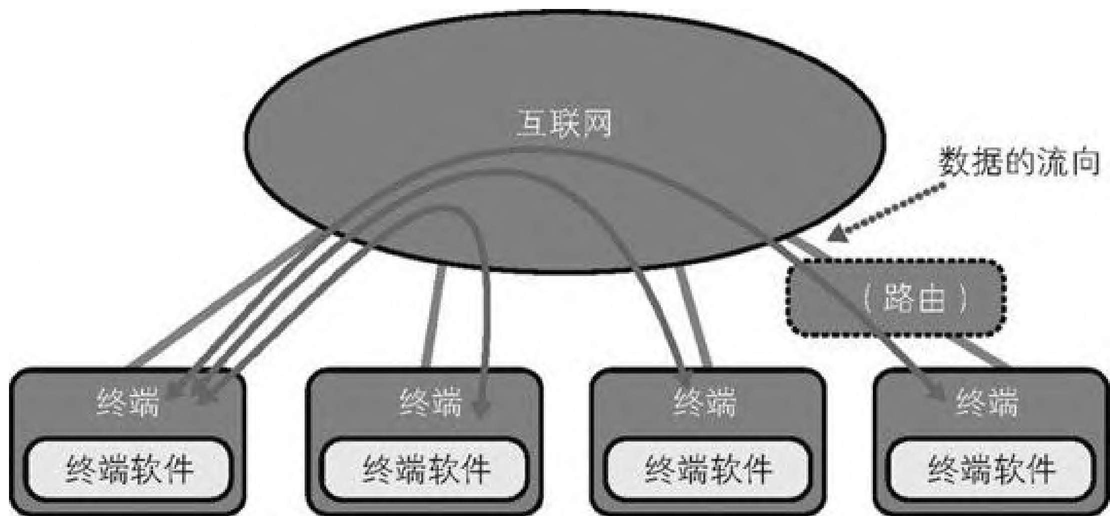


图 3.12 P2P 架构（星型结构）



NAT 穿透

在 P2P 架构的游戏中，连接至互联网的 2 台 PC 机或者游戏机直接相连，然后互相发送数据。P2P 架构有一个问题，最近大多数玩家的 PC 机或者游戏机都是通过具有 NAT 功能的路由器连接到受保护的住宅网络，所以不能通过指定全局 IP 地址来建立直接通信。这一比例逐年上升。为了避免这种情况，需要采用一种业界称为“NAT 穿透”（NAT 越え、NAT トラバーサル）的技术。这方面的内容将在 5.6 节中详细说明。

3.3.5 C/S ＋ P2P 混合型架构

本书并不详细介绍这种将 C/S 和 P2P 混合在一起的架构，但是这种架构还是存在的。图 3.13 是 C/S 和 P2P 全网状结构的混合型架构，而图 3.14 则是 C/S 和 P2P 星型结构的混合型架构。粗箭头代表 C/S 架构的数据流向，细箭头代表 P2P 架构的数据流向。星型结构和全网状结构不管在哪种情况下都可以使用。

图 3.13　C/S 架构与 P2P 全网状结构的混合型架构

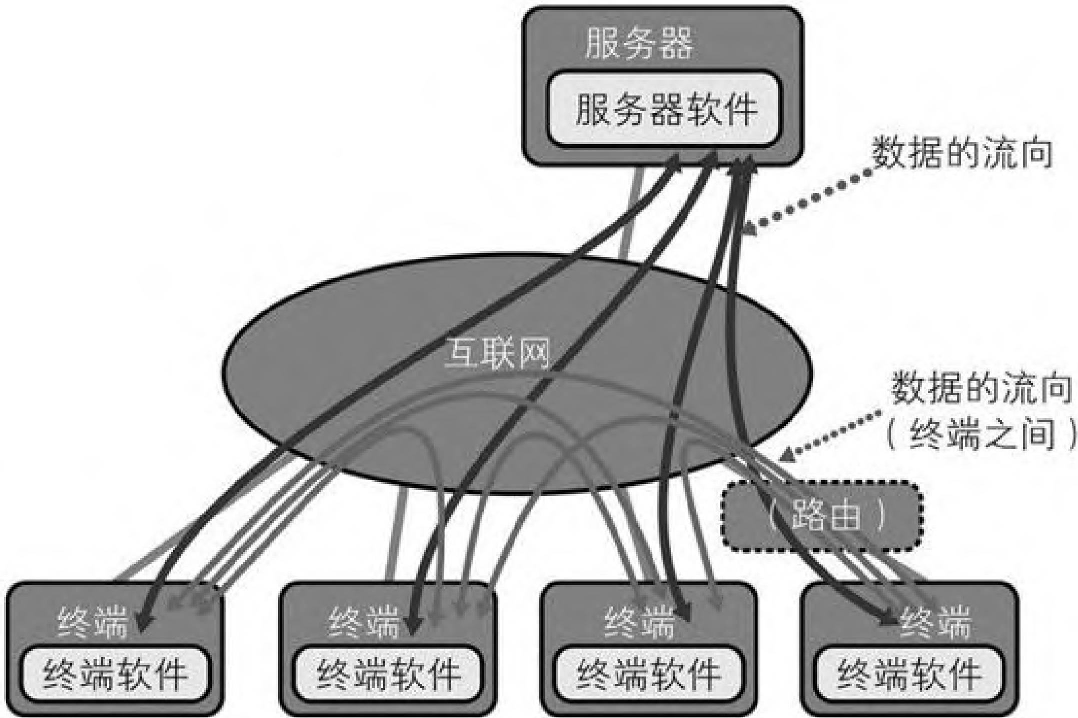
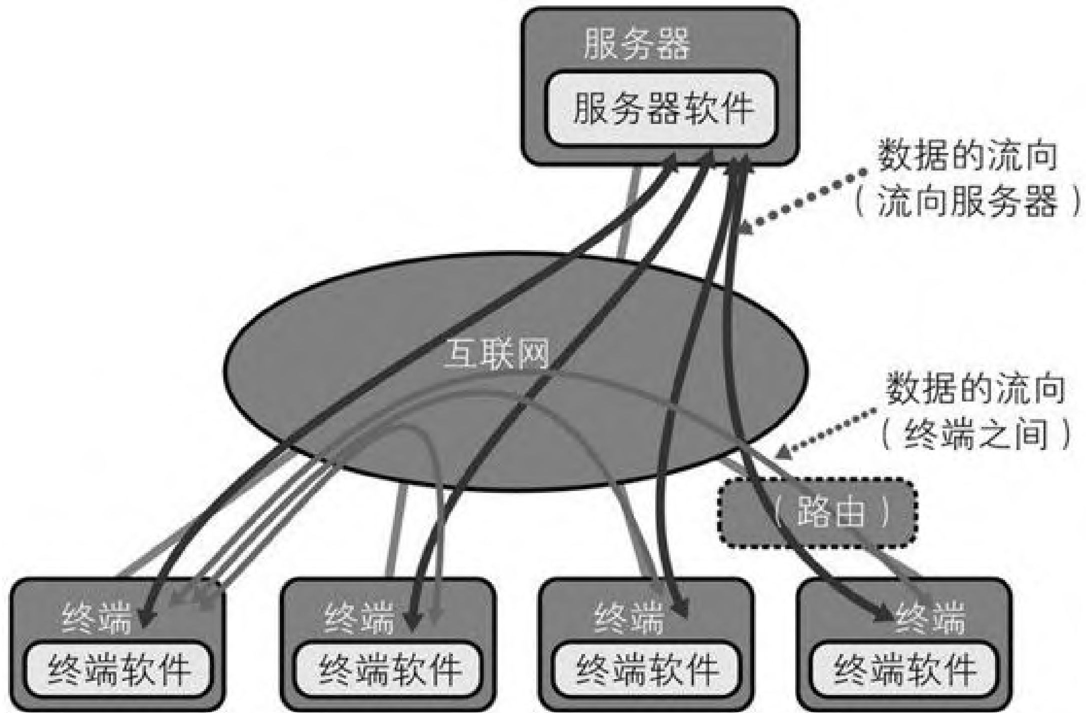


图 3.14　C/S 架构与 P2P 星型结构和混合型架构

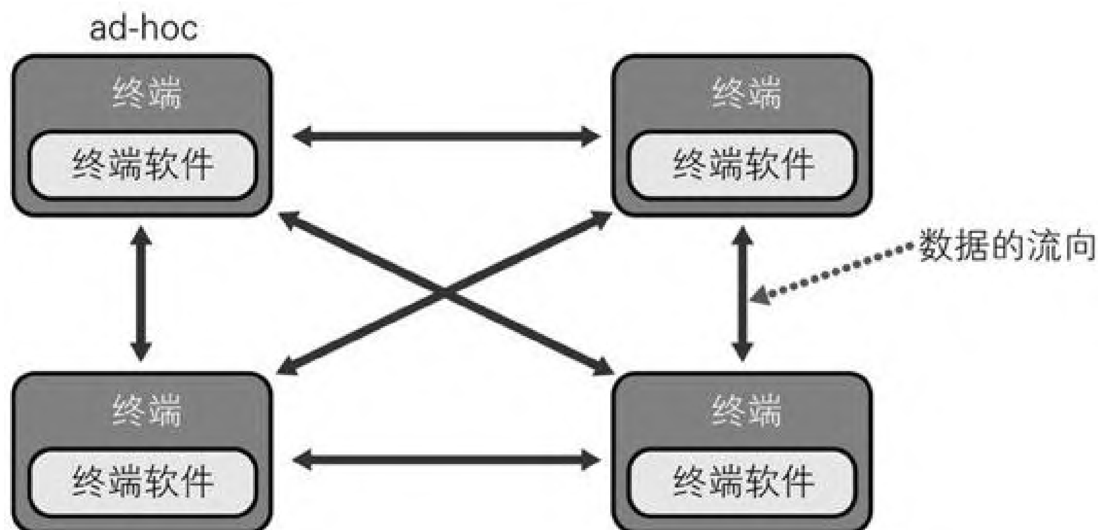


3.3.6 ad-hoc 模式

ad-hoc 模式也不作详细介绍，这里只是简单提一下。在 P2P 架构中还有一些其他的结构，比如不使用互联网的 ad-hoc 模式的无线 LAN、通过串行电缆直接通信、通过以太网等 LAN 连接直接通信，等等。

ad-hoc 模式中的各终端都是不通过线路来连接的移动终端，所以某个终端突然离线的概率很高，因此很少会将某个终端作为主机。不管哪个终端在何时离线，都可以在逻辑上使用异步方式，在物理上选择 P2P 架构。

图 3.15 ad-hoc 模式



专栏　游戏客户端是什么

在网络游戏的开发中，在玩家所用设备上（PC 机或游戏机）由玩家启动的为了进行游戏而运行着的、用来进行渲染处理和输入输出处理的专用游戏软件称为“游戏客户端”（Game Client）。事实上，这并不是那些与网络上的服务器进程进行连接的软件，但是通常它们都统称为客户端，所以在严格说明网络架构时，有时会引起混乱。

比如，在 P2P 架构的网络游戏中，在网络架构上，一个游戏软件接收来自另一个游戏软件的连接，而服务器也是存在的，在这种情况下，如果单单使用“客户端”这样的术语，就会说“这个客户端，既有服务器也有客户端……”，或者说“连接玩家 A 的客户端”，等等，这就造成了混淆。

因此，本书不使用作为通称的“客户端”，而是使用“游戏客户端”、“游戏终端软件”或者“终端软件”这样的名称。此外，在网络结构的说明中，运行终端软件的计算机就称为“终端”。而在表示 telnet 和 ssh 命令等用于远程操作计算机的通信软件时，也常会使用“终端软件”这样的术语。

3.4　逻辑架构详解——MO 架构

第 2 章提到过，逻辑架构包括“MO 架构”和“MMO 架构”两种。它们实现着完全不同的游戏类型。根据游戏的策划内容，也有一些处于这两者之间的游戏。本节从术语的整理开始，逐一介绍各个实现方式。

3.4.1 MO、MMO 是什么？——同时在线数的区别

对游戏玩家来说，“同时能跟多少其他玩家一起进行游戏”这一点对游戏体验产生了决定性的影响。在表示同时在线数的区别时，最重要的术语就是 MO 和 MMO。

- MO（Multiplayer Online）

  同时在线数为 2～100 人的游戏称为 MO（MOG）。这类游戏的游戏时间相对较短，一般在几个小时之内就能结束，每次开始游戏时，游戏的状态都会被重置（这与在将棋中每次开始对战时都会将棋盘重新排列为初始状态是一样的）。游戏数据的形式是一次性的（disposable）。

- MMO（Massively Multiplayer Online）

  同时在线数达到数百、数千以上的游戏称为 MMO（MMOG）。因为游戏参与人数众多，所以游戏时间通常长达几十个小时，而且也不能重置游戏数据。游戏数据的形式是永久性的（persistent）。

现在，同时在线数与网络的物理结构已经划不上等号了，但是基本上会采取以下方针。

- **同时在线数少（MO），采用“同步方式”和“异步方式”。**
- **同时在线数多（MMO），采用“浏览器方式”。**

MMO 和 MO 的混合

此外，作为现在的市场上的另一种趋势，又出现了第三种形式，也就是同时使用 MO 和 MMO 的混合架构，这种架构的游戏也为数不少。根据游戏的策划内容，想要同时兼具 MMOG 的优点和 MOG 的优点时就会

采用这种混合架构，它成为了从 MMO 游戏启动 MO 游戏的形式。在 MMO 游戏中，游戏玩法涉及很多方面，可以在特定的场所、与特定的敌人作战时采用 MO 架构，也可以在每次进行游戏时对一部分游戏状态进行重置，还可以节约带宽。比如，在 *WoW* 中，地面上的地图和大部分的地下城（Dungeon）都是以 MMO 架构来实现的，而那些供少数人重复在短时间内进行游戏的暂时性的地图则是以 MO 架构来实现的，通过这种方式将两种架构结合了起来。

在这种情况下，因为要用到差异非常大的实现方法，所以开发成本和测试所要花费的精力也大幅增加。因此，基本上只考虑在主要类型中使用。

* * *

在第 4 章、第 5 章我们将对 MMO 架构和 MO 架构的技术一一进行详细探讨。在此之前，首先对其做一些简要的介绍。本节就先来讨论一下 MO 架构。

3.4.2 MO 架构、MOG

MO 架构经常在 FPS 和 RTS（Real-time Strategy，即时战略）等类型的游戏中使用。这种架构适合那些在线人数较少、实时性很高的游戏。

如前所述，MOG 的实现方式有“同步方式”和“异步方式”。这两种方式都有全网状结构、星型结构两种实现形式。我们就按如下顺序来看一下这些结构的具体情况。

- MO 架构
  - 同步方式 / 全网状结构
  - 同步方式 / 星型结构
  - 异步方式 / 全网状结构
  - 异步方式 / 星型结构

3.4.3　同步方式——获得全体玩家的信息后，游戏才能继续

同步方式是一种只有在获得了全体玩家的信息之后，游戏才能继续进行下去的方式。

同步方式 / 全网状结构就是，参与游戏的所有终端都拥有主数据，这些终端互相传输所有的控制设备输入信息，在获得所有终端的输入数据之前，游戏始终处于等待状态。

同步方式 / 星型结构是，配置一个综合管理游戏数据的根服务器，所有参与游戏的终端（客户端）将玩家的所有输入信息发送至服务器，游戏状态一旦有所进展，服务器就将那些改变了的状态数据返回给所有客户端。在服务器返回信息之前，所有的客户端都不进行任何渲染，只是单纯地等着。

3.4.4　同步方式 / 全网状结构的实现——所有终端都拥有主数据

首先，我们来看一下同步方式 / 全网状结构。图 3.16 展示了拥有 3 个终端的全网状结构的情况。

各个终端所持有的数据全都是“主数据”，所以不会复制任何信息。而且各个终端通过网络互相传输的都只是控制设备的信息（比如按下方向键、 A 键、B 键等），所以在图 3.16 中标为“输入”。所有的终端都会将这些输入信息发送给除了自己以外的其他各个终端，所以图中总共有 6 个箭头。

图 3.16　同步方式（全网状结构）

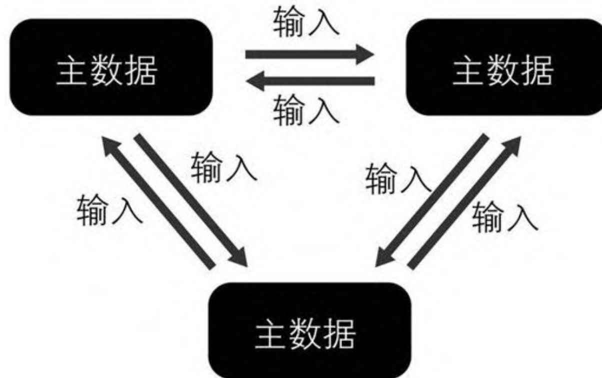
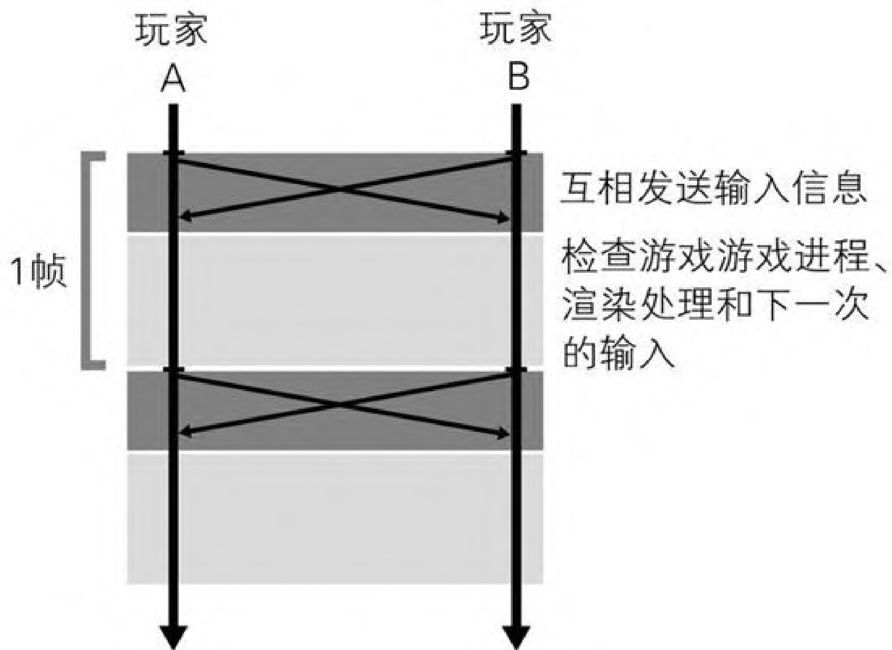


图 3.17 表示的是传输过程的时序，图中，一块深灰色带状部分和一块浅灰色带状部分组成了 1 帧。这会在 1 秒内重复 60 次。在从自己以外的所有终端上获得输入信息之前，只会单纯地等着。

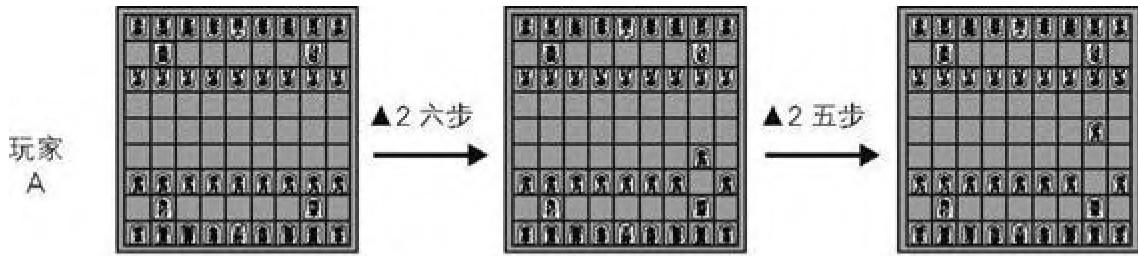
图 3.17　时序图



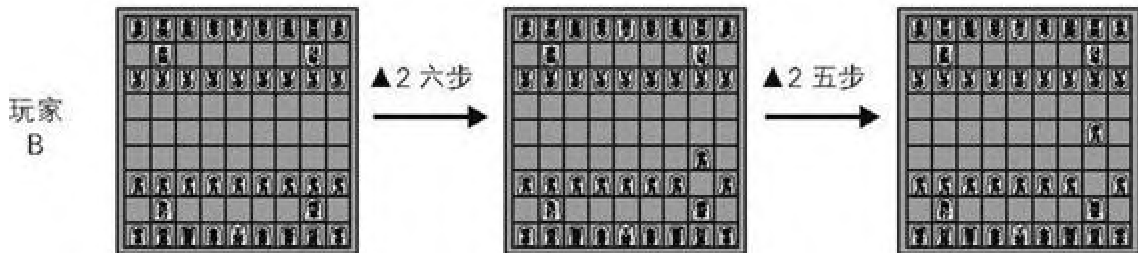
各终端（玩家）收发的信息内容

这里有一点很重要，那就是各终端之间只发送“控制设备的输入信息”。游戏过程数据都是数字数据，所以如果能毫无遗漏地发送初始状态及其对应的变更部分，所有玩家的状态就能始终保持一致了。这里我们以将棋为例（因为将棋易于理解）来说明（参见图 3.18）。

图 3.18　将棋示例



所传输的差异部分（▲2 六步、▲2 五步）相同的话，
即使不发送当前的状态，2 个玩家的棋盘状态也是相同的。

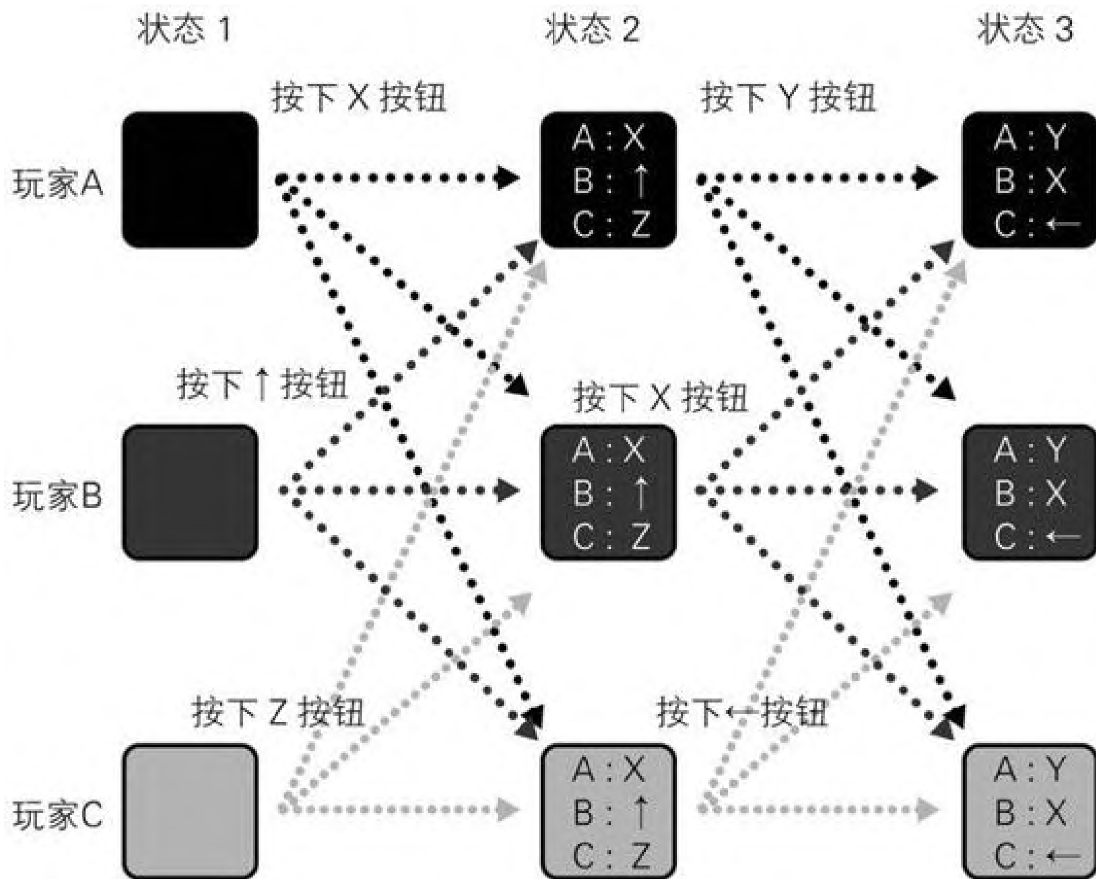


在将棋中，玩家 A 和玩家 B 在初始状态下是相同的。从初始状态开始，如果只发送了“2 六步”[6] 这样的输入信息，那么，即使不发送当前的棋盘状态和其他棋子的状态，两个玩家的棋盘状态也是相同的。

> [6] 将棋棋谱记法：每一手棋先记录棋子移动后的位置，先写筋（横坐标，相当于中国象棋的路），后写段（纵坐标，相当于中国象棋的线），再记录该棋子的名称。先手着法以▲表示，后手着法以▽表示。如“▲ 2 六步”就表示先手第一手推进飞车前的步兵。——译者注

有 3 个玩家的情况下，各个玩家之间收发的信息内容如图 3.19 所示。从图中可知，游戏状态每次有所进展时，所有终端都将把各自的信息发送给其他终端，从而玩家 A、B、C 都接收到了相同的输入信息。

图 3.19　3 个玩家的情况



当从状态 1 进入状态 2 时，各个终端都向其他终端发送输入信息，当所有终端都收取完成时就进入状态 2。此时，输入的内容在所有的终端上都是相同的，所以作为处理结果的状态 2 的内容也是相同的。然后到状态 3、状态 4……一直这么持续下去。

同步方式 / 全网状结构的必要条件和优势

到此为止所说明的同步方式 / 全网状结构，有没有觉得它有点“危险”？事实上确实如此。

首先我们来看一下这种结构的前提条件和它的优势。

为了使之前的假设成立，必须满足以下几个条件。

❶ 初始状态完全相同。

❷ 所有的输入信息数据包都确确实实地、毫无遗漏地发送至其他所有终端。

❸ 游戏过程数据不会随机变化（如果是结果完全相同的伪随机数也没有问题）。

❹ 游戏过程数据的变化不会发生波动。具体来讲，比如输入有限资源的数据包不会互相竞争到达的顺序，不会因为微妙的时机问题产生不同的结果，等等。

以上 4 个条件并不难满足，只要游戏程序做到以下几点，就能简单地满足这 4 个条件。

❶ 伪随机数的种子在所有终端上都保持一致。

❷ 所有终端都以完全相同的数据来初始化游戏。

❸ 循环开始。

a. 所有终端开始进行输入信息的传输，在全部接收完成前始终处于等待状态。

b. 按照玩家 A～Z 的顺序进行处理，依次改变游戏状态。

c. 渲染。

d. 受理下一个操作。

e. 发送自己这部分的输入信息。

以前的家用机游戏基本都是按照这几点来实现的。循环中的 a 处并不检查本地的控制输入，只要检查来自网络中的输入，本地的双人游戏就可以支持网络功能了。这种方式的优点就是：使用一般的方法开发的游戏能够很容易地支持网络功能，而且可以使程序非常简单。

同步方式 / 全网状结构的 3 个问题 ——通信网和收发信息在完整性上较为脆弱、中途加入游戏

此前说过同步方式 / 全网状结构有点“危险”，这种结构存在着以下 3 个问题。

❶ 人数增加后，收发信息的完整性极易崩溃。

❷ 最慢的终端会拖长整体的传输时间 [7] **。**

> [7] ❷ 这个问题是同步方式普遍存在的。

❸ 不能中途加入游戏 [8] **。**

> [8] 在同步方式下，❸ 这个问题在星型结构下也存在。对于中途加入游戏的相关问题，将在后面讲到同步方式 / 星型结构时进行讨论

- **混沌理论（Chaos theory）**

输入信息哪怕只缺少了一个，游戏数据就会不一致，这种不一致性就如混沌理论 [9] 所描述的那样会不断扩大，导致游戏结果变得不可收拾。

> [9] 混沌理论是一种兼具质性思考与量化分析的方法，用以探讨动态系统中无法用单一的数据关系，而必须用整体，连续的数据关系才能加以解释及预测之行为。混沌理论认为在混沌系统中，初始条件十分微小的变化，经过不断放大，对其未来状态会造成极其巨大的差别。——译者注

正如混沌理论所描述的那样，这种不一致性指的是“虽然只有些微的差异，但是通过不断重复单纯的规则，这些差异就会不断被放大，最终产生完全不同的结果”。比如，将“水从高处流向低处”这条简单的规则运用在水滴上，如果这个水滴落在乘鞍岳（日本北阿尔卑斯山脉的一部分）的顶上向北 1 米处的地方，最终将流入日本海，如果落在向南 1 米处的地方，最终就将流入太平洋。虽然只有 1 米的差异，但是最后的结果却相差了几百千米。这方面的理论就称为“混沌理论”，在游戏开发中频繁使用。

在游戏中，在某一个瞬间“按下了 A 按钮”，这个信息即使只漏发了一次，也可能造成在玩家 A 的终端上能够打倒敌方 boss，而在玩家 B 的终端上却没能打倒，10 秒之后玩家 A 和玩家 B 就会得到完全不同的结果，玩家 B 所得到的结果将是游戏结束。

传输线路的可靠性

不仅仅互联网不稳定，通信网也同样是不稳定的。假设每条传输线路在 10 分钟内不发生信息丢失的概率为 0.99，两名玩家需要 1 条传输线路，3 名玩家需要 3 条，n 名玩家就需要 $n \times (n - 1)/2$ 条。只要有一条传输线路出现问题，就会对所有玩家造成影响，有 4 名玩家时，传输线路为 (4×(4－1)/2) = 6 条，不发生信息丢失的概率则为 0.99×6 = 0.94，线路的可靠性和游戏出现问题的概率可以通过以上方式来计算。如果传输线路的故障和传输线路之间的关系无关，那么发生故障的概率如表 3.3 所示。

表 3.3　传输线路发生故障的估算

玩家人数	传输线路的数量	90% 可靠性※	99% 可靠性※	99.9% 可靠性※
1	0	1	1	1
2	1	0.9	0.99	0.99
3	3	0.729	0.97	0.99
4	6	0.53	0.94	0.99
5	10	0.34	0.90	0.99
6	15	0.2	0.86	0.98
7	21	0.1	0.80	0.978
8	28	0.05	0.75	0.972

玩家人数	传输线路的数量	90% 可靠性※	99% 可靠性※	99.9% 可靠性※
10	45	0.008	0.63	0.95
15	105	几乎为 0	0.34	0.9
20	190	几乎为 0	0.14	0.82
30	435	几乎为 0	0.012	0.64
40	780	几乎为 0	几乎为 0	0.45

※ 表示 10 分钟内不出问题的概率

在传输线路的可靠性方面并没有很准确的数据可供参考，就笔者经验来讲，如果使用无线和国际互联网，可靠性程度一般在 0.9 左右，如果使用国内有线宽带则在 0.99 左右，只有在局域网内使用有线以太网时才达到 0.999 的程度。实际发售的商业游戏很多都将同时在线数限制在 3～5 人，这正是出于传输线路的可靠性问题来考虑的。

商业游戏对可靠性的要求根据其商业模式的不同而有所不同。比如在游戏中心，每次进行游戏需要支付 200 日元，在这种情况下，是不允许 10 次里面出 1 次故障的。即使是免费游戏，10 次里出故障 1 次也会使玩家失去继续玩下去的兴致。在那些游戏行业与其他娱乐行业竞争较少的发展中国家或许情况有所不同，但是对于像 NDS 这种以软件包形式发售的游戏，可靠性起码要达到几十次里只有 1 次故障的程度。

即使在传输线路不可靠时，也要能够通过重新发送数据来恢复原先的状态。具体来讲，如果数据到达的顺序发生了错乱，就要考虑要求重发，或者使用针对游戏特别调整过的 TCP 协议等方法。这样就能在 1 个数据包没有送达的情况下自动重发，从而避免游戏数据的损坏。但是重发数据还是避免不了网络延迟这个问题。

虽然可以通过这种方式确保数据包的可靠性，但是传输线路本身发生中断的概率也是相当高的。这种情况下没有任何恢复手段。根据笔者的经验，如果使用的线路容易造成数据包不能正确到达，那么线路本身发生中断的概率就会非常高，印象中，引入和不引入拓扑结构之间的差异不超过两倍。加上这一点，即使引入了保证数据包可靠性的结构，根据经验来预测可靠性的具体数值的话，还是如前述的表 3.3 所示。

最慢的终端会拖长整体的传输时间

在同步方式中，除了传输线路的可靠性之外还存在另外一个问题，那就是“最慢的那个终端会拖长整体的传输时间”。同步方式中最基本的行为模式就是“在获取到所有终端的输入信息之前，系统将一直处于等待状态”，所以，如果有一个终端发送数据包非常慢，其他的所有终端都不得不一直等着该终端的信息的到来。

数据包发送迟缓一般有两种情况：❶ 一时之间传输突然变得非常缓慢；❷ 长期传输较为缓慢。

对于第一种情况，玩家会感觉到暂时性的中断。这在行业术语中称为“跳帧”，与 Web 服务中的 5 秒规则一样极为令人厌恶。“认知→判断→操作”的过程会因此而中断，导致游戏体验显著受损。

使用 eMobile 等无线终端很快就能确认多个数据包中某个数据包的传输异常（参见图 3.20）。在图 3.20 中，最快 128 毫秒，最慢 570 毫秒，平均下来是 191 毫秒的 4 倍以上。在使用有线 ADSL 等网络的情况下，虽然数据包发生传输异常缓慢的情况减少了很多，但有时还是会发生。

造成长期传输缓慢的原因有很多，比如玩家在地理上相距甚远、玩家一边下载一边进行游戏、PC 或路由器性能低下、安全软件正在自检，等等。

在采用同步方式时，必须关注各种各样的问题。

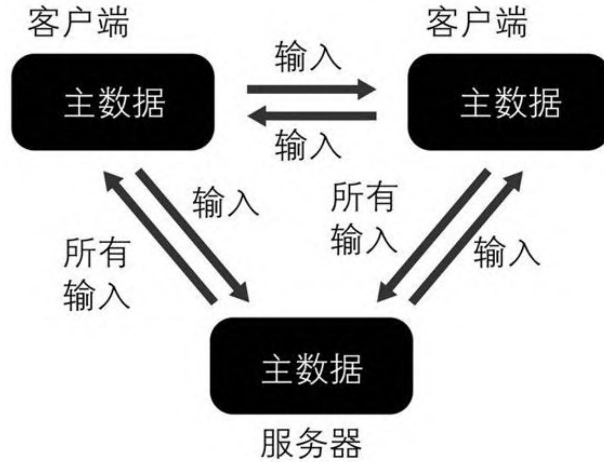
图 3.20　数据包传输异常缓慢的现象

```
$ ping www.kyoto-u.ac.jp
PING www.kyoto-u.ac.jp (130.54.120.215): 56 data bytes
64 bytes from 130.54.120.215: icmp_seq=0 ttl=50 time=136.247 ms
64 bytes from 130.54.120.215: icmp_seq=1 ttl=50 time=151.256 ms
64 bytes from 130.54.120.215: icmp_seq=2 ttl=50 time=210.943 ms
64 bytes from 130.54.120.215: icmp_seq=3 ttl=50 time=145.197 ms
64 bytes from 130.54.120.215: icmp_seq=4 ttl=50 time=130.420 ms
64 bytes from 130.54.120.215: icmp_seq=5 ttl=50 time=140.207 ms
64 bytes from 130.54.120.215: icmp_seq=6 ttl=50 time=160.243 ms
64 bytes from 130.54.120.215: icmp_seq=7 ttl=50 time=150.718 ms
64 bytes from 130.54.120.215: icmp_seq=8 ttl=50 time=129.447 ms
64 bytes from 130.54.120.215: icmp_seq=9 ttl=50 time=267.831 ms
64 bytes from 130.54.120.215: icmp_seq=10 ttl=50 time=570.406

 ms
64 bytes from 130.54.120.215: icmp_seq=11 ttl=50 time=214.361 ms
64 bytes from 130.54.120.215: icmp_seq=12 ttl=50 time=219.075 ms
64 bytes from 130.54.120.215: icmp_seq=13 ttl=50 time=128.852

 ms
64 bytes from 130.54.120.215: icmp_seq=14 ttl=50 time=148.724 ms
64 bytes from 130.54.120.215: icmp_seq=15 ttl=50 time=158.347 ms
^C
--- www.kyoto-u.ac.jp ping statistics ---
16 packets transmitted, 16 packets received, 0% packet loss
round-trip min/avg/max/stddev = 128.852/191.392/570.406

/105.350 ms
```

3.4.5　同步方式 / 星型结构——暂时将输入信息集中到服务器上

接下来，我们来看一下同步方式 / 星型结构（参见图 3.21）。

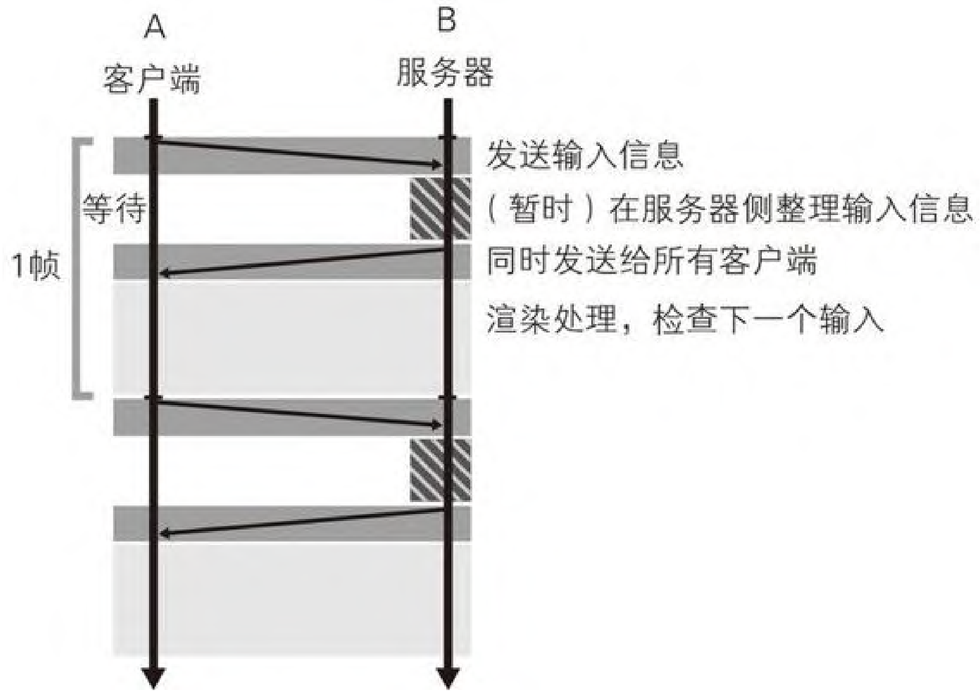
图 3.21　同步方式 / 星型结构



在这种结构下，网络中的所有成员并不是完全平等的，星型结构的中央终端称为“服务器”，其他终端则称为“客户端”。客户端将控制设备上的方向键等输入信息发送至服务器，服务器在接收完所有客户端发来的输入数据前一直处于等待状态，接收完成后则将接收到的输入信息同时发送给所有客户端。

这种情况下的时序图如图 3.22 所示。在图 3.22 中，玩家 A 这一端是客户端，玩家 B 这一端则是服务器。在最初的 1 帧中，首先 A 向 B 发送输入信息，B 侧接收该信息，等到接收完所有客户端的输入信息后就将其发送给 A。对玩家 B 来说，这些输入信息就这么原封不动地反映在自己管理的游戏数据中再加以显示就可以了，所以不需要传输。星型结构的特点就是“暂时将输入信息集中到服务器上”，这一点与全网状结构不同。

图 3.22　时序图



玩家数为 3 时的情况如图 3.23 所示。在从状态 1 进入状态 2 之前，首先将所有终端的输入信息全部集中到玩家 A 的终端上，它作为所有终端的代表来接收输入信息。

图 3.23　3 名玩家时的情况

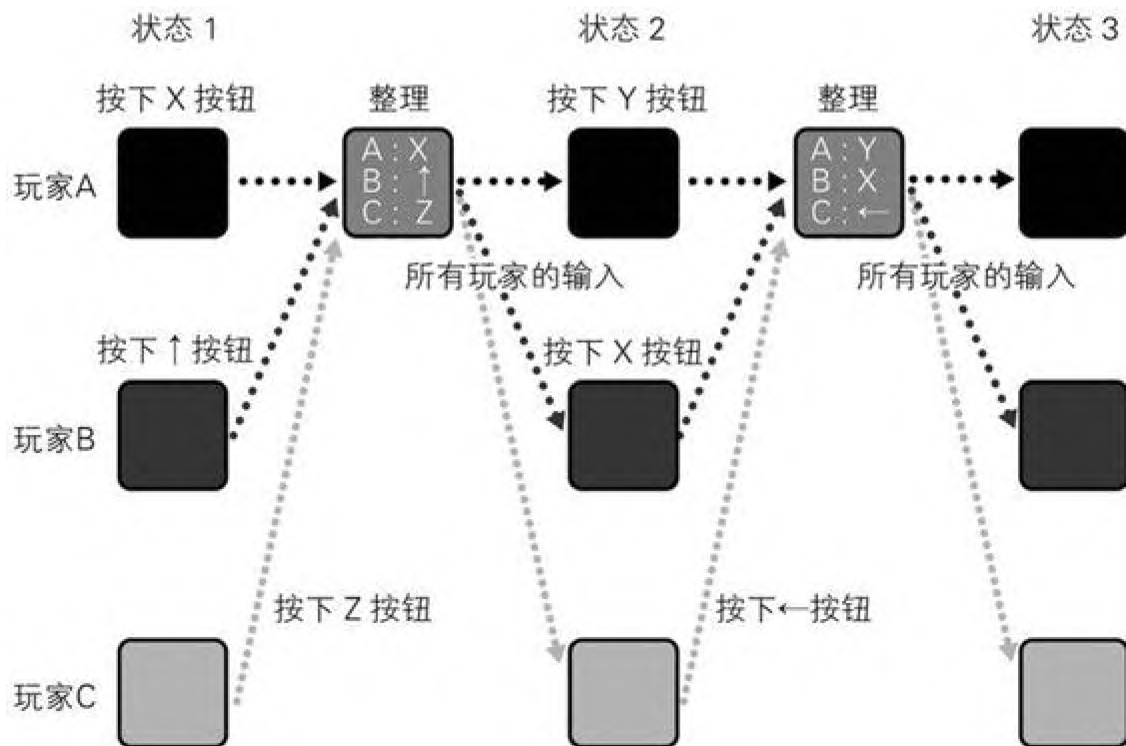


图 3.23 这种方式的最大优点就是，所需增加的传输线路与所增加的玩家数的一次方成正比。传输线路不会爆发性地增长，所以发生游戏数据不一致的概率将大幅下降。从这个意义上来说，“危险的感觉”确实比全网状结构来得薄弱。

星型结构的 4 个问题

但另一方面，星型结构也产生了 4 个问题。总结如下。

❶ 响应较慢。

❷ 如果玩家 A 中途离线，游戏无法恢复，只能强行中止。

❸ 信息整理方面的逻辑增加时，程序的结构比全网状结构稍稍复杂一些。

❹ 玩家 A 的终端上的传输负荷比其他终端高出许多，不甚公平。

在全网状结构中，传输线路的增加与玩家数的 2 次方成正比，而星型结构则是 1 次方。这样，同时在线数就能增加。但是也因此需要在响

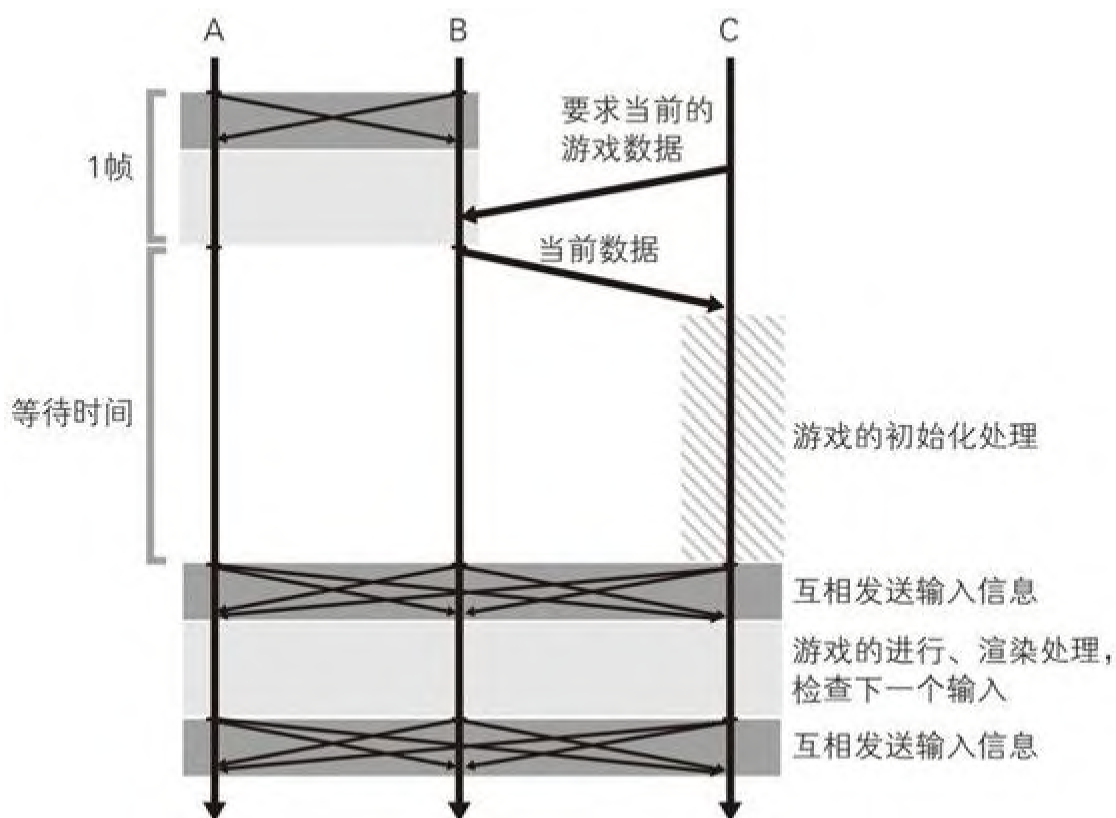
应时间和编程成本上付出一些代价。正所谓有得必有失。

同步方式不可避免的重大问题 ——不能中途加入游戏

最后，我们来看一下同步方式下一个不可避免的重大问题，那就是“不能中途加入游戏”。在同步方式中，只会从游戏的初始状态开始发送控制设备的输入信息，然后进行游戏，所以如果在中途增加玩家，必须要将游戏的所有状态发送给该玩家。像将棋和扫雷这种游戏的数据，全部加起来也不过几十个字节，即使要向中途加入的玩家发送所有的游戏数据，也可以在 1 帧内完成处理，但是如果是那种需要处理大量数据的游戏，不暂时中断游戏就无法完成。

这种情况下的时序图如图 3.24 所示。

图 3.24　时序图



将棋和奥赛罗这类游戏暂且不谈，在更大型的游戏中，游戏数据的初始化处理通常无法在 1 帧内完成。数据量也是相当大的，有时还需要

从 HDD 或 DVD 等设备中获取必要的数据，仅仅是实际的处理就常常需要花费 1 秒甚至几秒以上（图 3.24 中条纹状部分）。

在图 3.24 中，新加入的玩家 C 可以向任何一个玩家要求当前的游戏状态（这里是玩家 B），这个例子中由玩家 B 返回游戏数据。如果这里的数据量达到 1 兆字节，仅仅是传输就需要花费好几秒。在这段时间内，游戏的状态不能改变，所以不仅仅是玩家 C，所有的玩家都必须暂时中断游戏。这里所说的“不能中途加入游戏”就是因为“在中途加入游戏的这一刻，为了传输游戏数据，所有玩家都必须长时间停止游戏”。

选择异步方式可以避免中途加入游戏的问题。这一点将在后面介绍。

同步方式的优势和问题解决的方法

不管是星型结构还是全网状结构，只要使用的是同步方式，就能简单地保存程序内容，这是一个很大的优势，所以有时会为了使用同步方式而采取各种各样的方法。

为了使用同步方式，可以采用以下方法。

- **像竞速游戏和对战格斗游戏这样，在几分钟之内结束一回合的游戏（竞速和比赛），这样就没必要中途加入游戏了。**
- **考虑玩家匹配系统，优先匹配地理位置相近的玩家。**

＊ ＊ ＊

以上我们讨论了同步方式的两种结构：“全网状结构”和“星型结构”。不管使用哪种结构，每个终端都要在“获得全体玩家的信息后，才能继续游戏”，这一点是最基本的，所以根据传输线路的可靠性和延迟的长短，玩家人数在达到一定程度后就不能再继续增加了。

3.4.6　异步方式——接受各终端上游戏状态的不一致

与同步方式相同，异步方式也有全网状结构和星型结构这两种实现方式。

异步方式的最大特点就是：各个终端上的游戏状态是不同的。也就是说，在游戏数据的一致性方面作出妥协，不要求数据完全一致。

由于这种妥协，比起同步方式，在异步方式下可以使用更加不稳定的传输线路和延迟更大的线路，也可以支持更多的同时在线数。但是另一方面，程序相较于同步方式就略显复杂了些，而且在有些情况下游戏体验也更差一些。

异步方式下实现方针的制定方法 ——对游戏内容的详细分析是不可缺少的

在异步方式的实现方面，应该对什么样的游戏数据作出何种妥协完全依赖于游戏内容。在同步方式下，选择了全网状结构或者星型结构后就自动决定了相应的实现方法，但是在异步方式中，对游戏内容的详细理解和分析是不可缺少的。

因此，为了进一步深入游戏内容，下面我们对游戏中的各类要素进行分解，考虑一下充分利用游戏特有的条件的方法。

3.4.7　三大基本要素：自己、对手、环境——异步实现的指导方针

首先将构成游戏世界的基本要素分为三大类：“自己的状态”、“对手（们）的状态”、“环境状态”。可以说这些就涵盖了游戏中的所有基本要素。

“自己”指的是，在可以直接操纵虚拟人物（avatar）的游戏中，玩家自己所操纵的角色。“自己的状态”就是指该角色的坐标、剩余体力、装备、剩余战机数，等等。如果操纵的不是单个人物，而是一个群体或者一个军队，那么“自己的状态”就是指这个群体的状态。而在那些像俄罗斯方块这样的完全没有人物登场的益智游戏中，指的就是正在往下掉落的屏幕最中间的那个可以直接操纵的物体的状态。总而言之，就是处于自己控制下的、自己必须注意的一系列数据。

“对手”则是指必须注意的其他玩家的一系列数据。玩家之间基本上是对等的，所以自己必须注意的角色所具有的信息量对于对手来说也是相同的。

“环境”就是指掉落在地上的物体、天气情况、敌人的状态等所有不属于任何玩家的东西。在俄罗斯方块这样的游戏中，已经堆叠在下方区域中的方块就是“环境”。

三大要素之间的关系

因为有 3 个要素，所以它们之间的关系也有 3 种。

❶ 自己和对手

❷ 自己和环境

❸ 环境和对手

这 3 种关系中，哪些比较重要呢？ ❶、❷ 两点不管在什么游戏中都非常重要，而 ❸ 就不怎么重要了。

❶ 和 ❷ 到底哪个更重要是相对于游戏内容而言的。比如，在 CAPCOM 的格斗游戏《街头霸王》系列中，环境的影响非常小，玩家可以心无杂念地将注意力集中在自己和对手的状态上。男性之间的格斗就是这么一回事，灵活利用地理上的优势巧妙获胜并非格斗的王道。如果在《街头霸王》中强化了环境要素，游戏的核心就有所偏离了。毕竟还是挥击自己的拳头打中对方这一易于理解的方式才是格斗游戏的设计中所需要的。可以说这就是以自己和对手之间的关系为主的游戏。

另一方面，在需要持续打倒逐渐逼近的敌人的 ARPG 游戏中，自己和包含敌人在内的环境之间的相互作用（Interaction）是最主要的，通过环境建立起与其他玩家的关系。这种就是以自己和环境之间的关系为主的游戏。

这里我们按照上面 ❶❷❸ 的顺序来加以说明。

3.4.8 ❶ 自己和对手——对战游戏和玩家之间往来数据的抽象程度

在网络游戏中，玩家与玩家直接对战以一决胜负的游戏称为PvP（Player versus Player），这类游戏拥有着以男性玩家为主的庞大市场。玩家之间进行对战以决出胜负是充满乐趣的。以玩家之间进行激烈交锋为中心的 PvP 游戏主要有对战格斗游戏、FPS 这样的射击游戏、竞速游戏、以战争为主的 MMORPG 等。

格斗游戏的例子

格斗游戏的例子这里我们先来看一下格斗游戏的传输时序图，了解一下需要怎样的技术判断。

首先假设有两名玩家：玩家 A、玩家 B 各自进行操作（参见图3.25）。画面的显示每 16 毫秒更新一次。在日本国内使用互联网提供的对战游戏服务，数据包延迟几帧是很有可能的，所以必须要考虑到通信延迟的问题。

图 3.25 《街头霸王》系列

©CAPCOM U.S.A.,INC

1992 ALL RIGHTS RESERVED

※ 图像提供：（株）Capcom http://www.capcom.co.jp/

对战格斗游戏中的经典之作。上图是在全世界售卖了 630 万份的杰作《街头霸王 2》的截图。该图显示了两人对战的情况，伤害值始终显示在画面上。

攻击、防御、碰撞检测

玩家的动作分为攻击、防御、被打倒三种。受到攻击后会被打倒在地，每次被打倒就会计算伤害值，一旦达到最大值就判为负。当前的伤害值总是会在画面上方的横条中以易于理解的方式显示。游戏中最重要的就是通过组合击打、踢腿等基本动作，首先对对方造成一定量的伤害。

图 3.26 1中的 ❶～❸ 是基本动作中的踢腿攻击。在图 3.26 1中将这个动作分解成 了 ❶～❸这 3 种形式进行了说明，但是一般来讲，从按下按钮开始，就会立刻流畅地开始踢腿动作的动画，通常使用 200～500 毫秒，也就是 10～30 帧来表现整个动作。在 3D 游戏中，这样的动画称为“Motion”。

如果在攻击动作进行时命中了对手，那么就如图 3.26 2所示，对手被击倒了。

在显示攻击动画的状态下，每一帧都要对不同的坐标进行碰撞检测。“碰撞检测”就是判断攻击是否命中了对方角色，这是游戏开发中的一个基本术语。

图 3.26　踢腿攻击

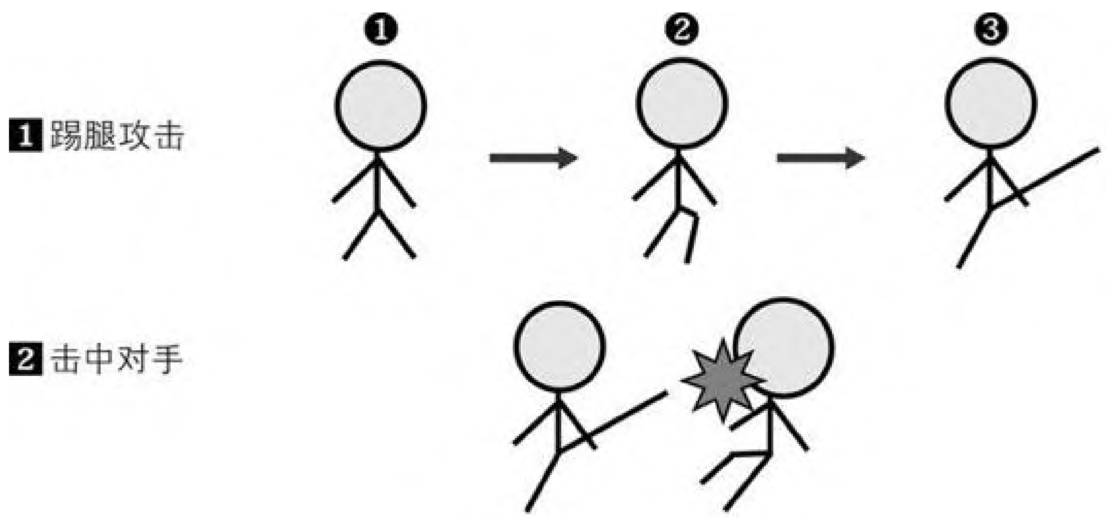


图 3.27 对全部 8 帧的攻击动画中每一帧在哪个位置发生了碰撞检测进行了图示。碰撞检测的方法各种各样，但是通过将用于碰撞检测的形状设置为与实际图像的外观不同的形状，可以提高游戏的真实感。这里假设使用矩形来进行碰撞检测。

图 3.27　碰撞检测（矩形）

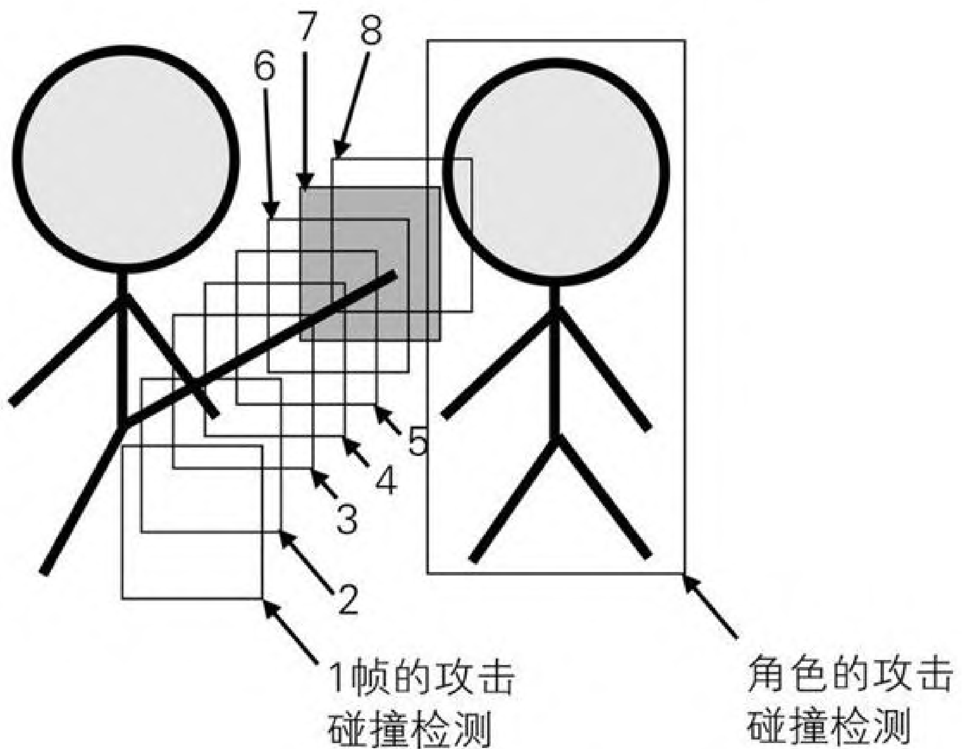
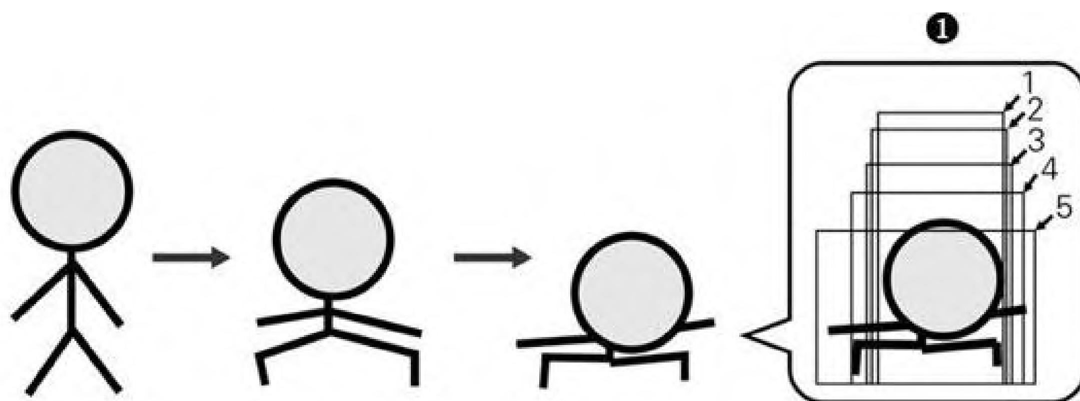


图 3.27 中，在总共 8 帧的踢腿动作中，与对战对手的角色进行碰撞检测时，在第 7 帧时两者的矩形首次发生了重叠。如果对手完全不动，玩家开始攻击后，就会在第 7 帧，也就是大约 116 毫秒（16.6×7）后命中对手。攻击命中后，就会中断第 8 帧的动画显示，如果游戏规则允许后续动作的话，玩家可能在那一瞬间进行操作。

另一方面，被攻击的一方可以采取躲闪行为而不是站在原地不动。在图 3.28 中，对手采取了下蹲的方式来躲避攻击。

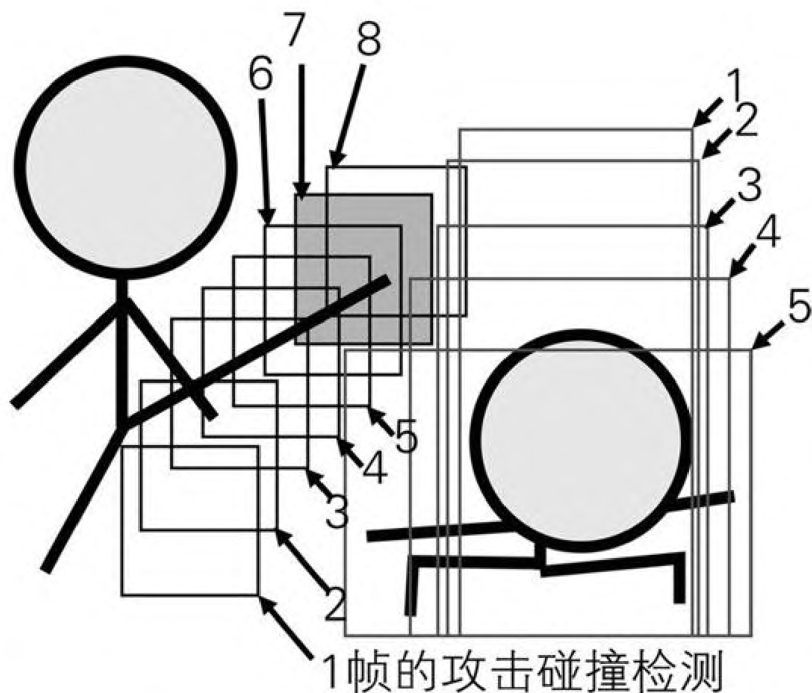
图 3.28　采取下蹲的方式躲避攻击



与攻击方一样，采取下蹲行为的角色的碰撞检测也是随着时间变化的。图 3.28 ❶ 显示了 5 帧内的变化。因为下蹲时角色的体型发生了变化，所以用于碰撞检测的矩形也有所改变。

图 3.29 显示了攻击方和被攻击方两者的碰撞检测的变化情况。在这个例子中，碰撞检测的矩形发生重叠的是在5-5、6-5、7-4、8-4、8-3、8-2、8-1 的时候（以攻击方—躲闪方的形式来表示）。

图 3.29　攻击方与被攻击方的碰撞检测的变化情况



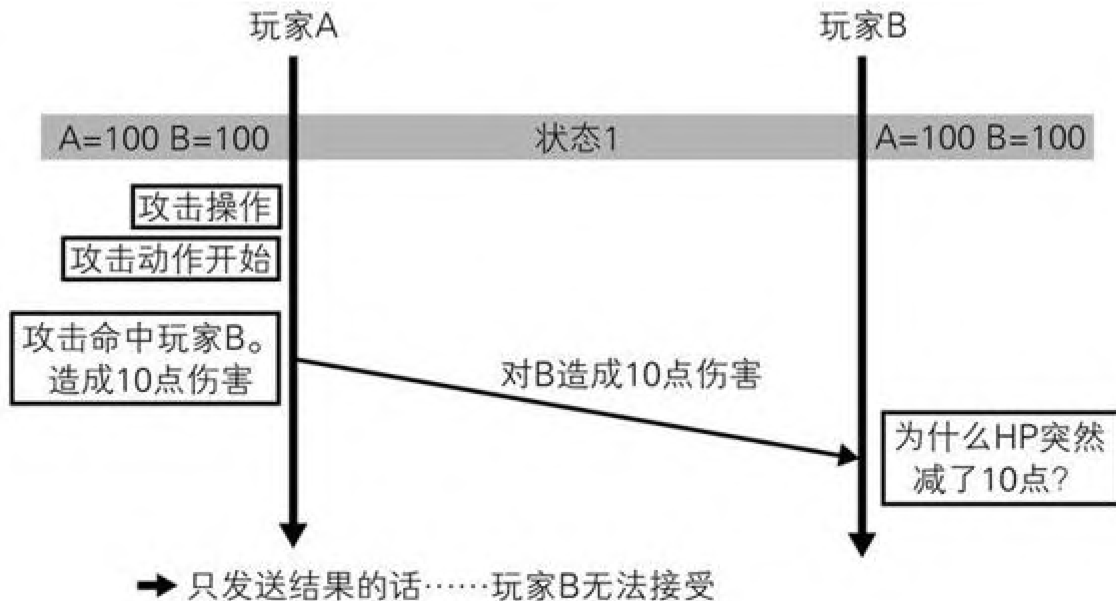
如果躲闪方在看到对方发动攻击后能够在 3 帧之后开始采取躲闪行为，那么两者之间动画帧的关系就是：3-1、4-2、5-3、6-4、7-5，这样就不会发生碰撞，也就意味着躲闪方能够闪避掉对方的攻击。但是如果在 4 帧之后才开始躲闪行为，那么就是 4-1、5-2、6-3、7-4，可见，在 7-4 的时候被踢中了。

由此可知，1 帧，也就是 16 毫秒之差直接关系到游戏中的所有动作，从而影响了对战的结果。

格斗游戏的时序图

牢牢记住格斗游戏中这 1 帧之差的重要性，然后试着画一下网络对战时的时序图（参见图 3.30）。

图 3.30　时序图（格斗游戏的网络对战）❶※



※HP（Hit Point）：生命值（角色的体力）

图 3.30 以时序图的方式显示了在对战格斗游戏中进行 1 次攻击时，玩家 A 和玩家 B 之间的关系。横向的箭头表示传输的内容和方向，为一个数据包。为了表现通信延迟，将这个箭头向下倾斜了一些。时间顺序以向下的方式表示。围在长方形中的内容是各个玩家的行动和体验。

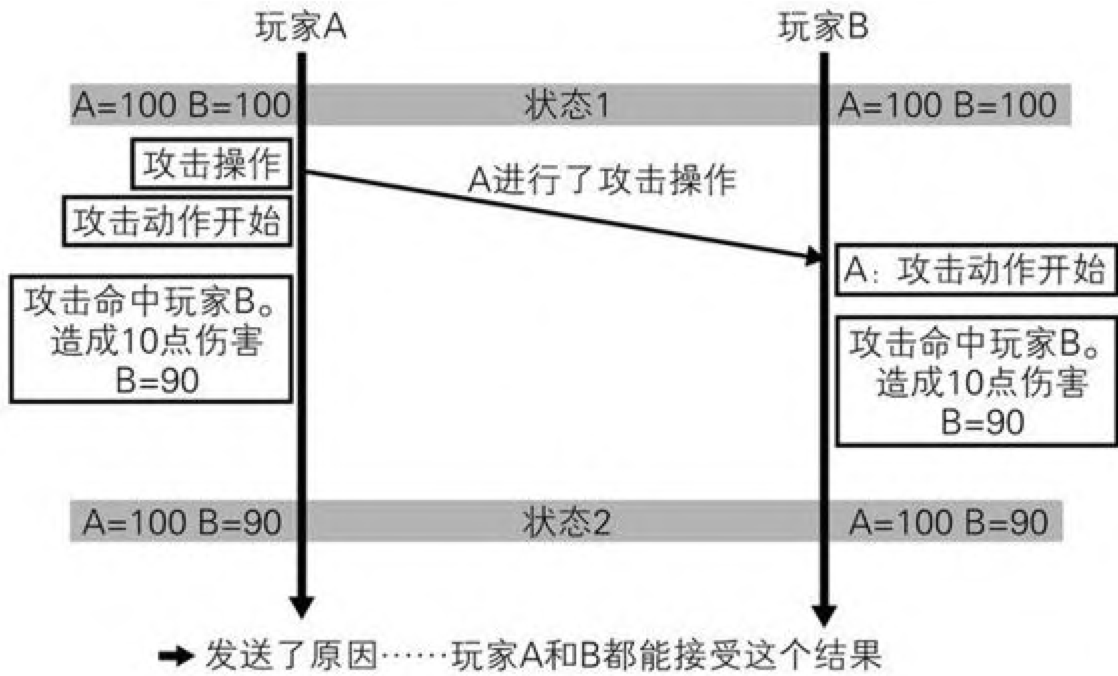
每个玩家的角色在初始状态下拥有的伤害量都是 100 点，受到伤害后该数值就会减少，首先减至 0 的玩家算败北。以 A = 100、B = 100 这种记法来表示该时刻下的剩余伤害量。

在初始状态（状态 1）下，玩家 A 和玩家 B 看到的状态是完全相同的。在图 3.30 所示的最初的时序中，发送了游戏结果（玩家 B 受到了伤害），但是只是突然发送一个结果的话，玩家 B 无法获得产生这一结果的信息，所以也就无法理解和接受这样的游戏结果。

必须发送抽象度较低、表示原因的数据 ——对结果的接纳感

格斗游戏中的格斗过程是非常重要的，所以必须发送造成伤害这一“结果”的“原因”。如果将其以“抽象度”这样的概念来表示，就是“因为在格斗游戏中，具体的格斗过程对游戏来说非常重要，所以必须发送抽象度较低的数据”。将其用时序图来表示的话就如图 3.31 所示。

图 3.31　时序图（格斗游戏的网络对战）❷



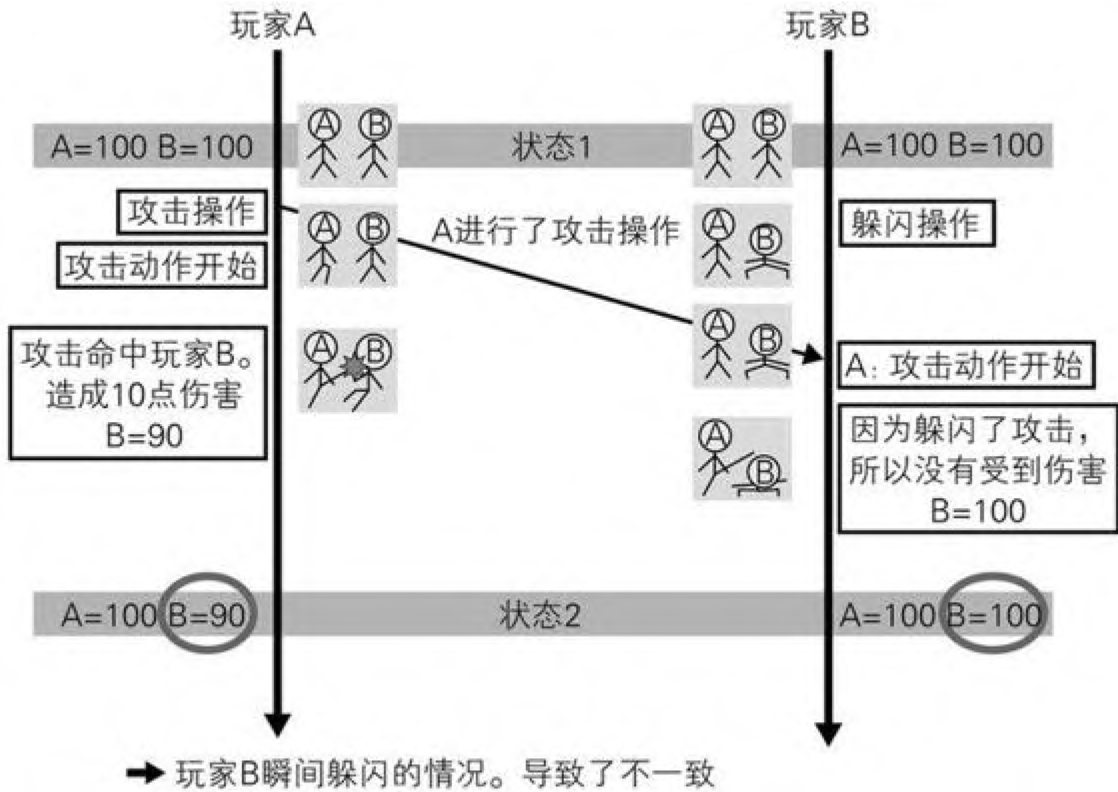
玩家 A 按下了攻击按钮开始进行攻击的瞬间，立刻向 B 发送了这一信息。与图 3.30 不同的是，这次发送的不是这一行为的结果（造成的伤害），而是行为的内容。

在玩家 B 这一侧，收到这一信息后就立刻开始显示玩家 A 的攻击动画。同时在玩家 B 这一侧进行碰撞检测，攻击命中的话就对玩家 B 造成伤害。这样，玩家 B 就能接受这 个结果了。

可能产生不一致的结果！

如果只是单单踢了一下什么也没做的玩家 B，那么这次的攻击就到此为止了。但在实际的游戏中，玩家 B 会一边揣摩比赛的发展，一边调动反应神经，在最恰当的时机躲避攻击。如图 3.32 所示。图中的灰色插图表示每个玩家所看到的画面。

图 3.32　时序图（格斗游戏的网络对战）❸



在这个例子中，玩家 B 基本上在玩家 A 发动攻击的同时就进行了躲闪，由于从玩家 A 这里接收到代表攻击的数据包时已经开始了躲避操作，所以在玩家 B 这一侧成功避开了这次的攻击。但是在玩家 A 这一侧看到的结果却是攻击命中了。从而，玩家 A 和玩家 B 看到的状态 2 产生了完全不同的结果（玩家 B 的体力值不同了，图下方圆圈内的数值：B = 90 和 B = 100）。

3.4.9　保持结果一致性的方法——两种覆盖方式

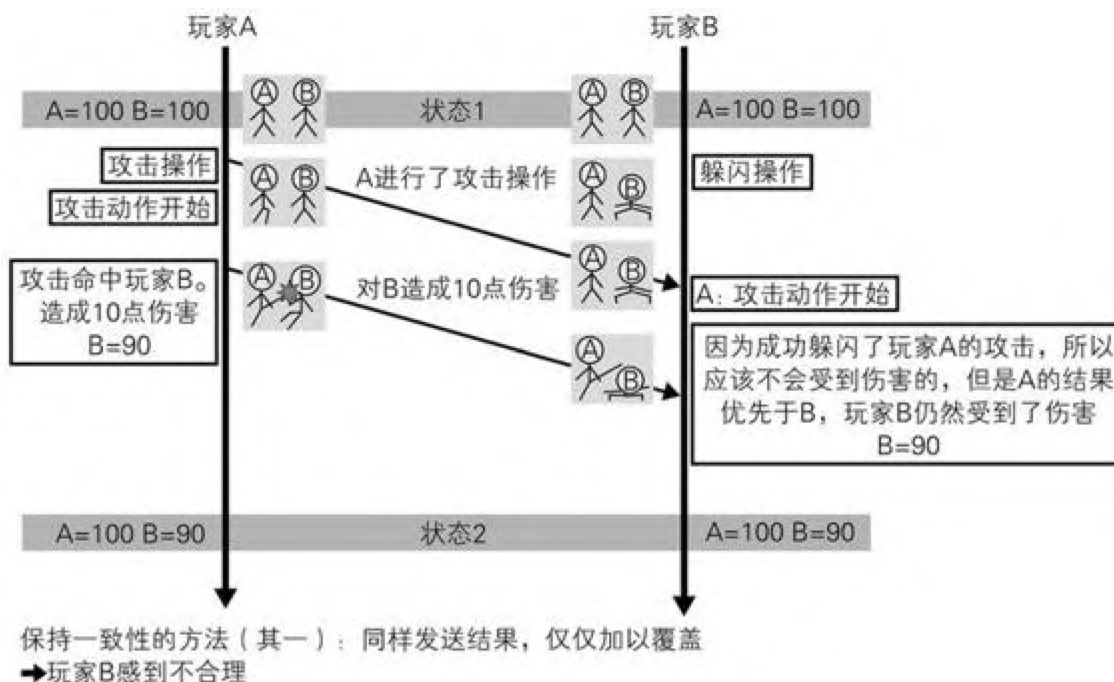
为了避免这种情况，保持结果的一致性，需要采用一些方法。以下是两种典型的覆盖方式。

- 采用造成伤害的那一方的结果。
- 采用受到攻击的那一方的结果。

采用造成伤害的那一方的结果

图 3.33 所示的方法是将造成伤害的那一方的结果发送给另一方，并且强制采用这一结果。虽然从画面上看玩家 B 成功躲避了这次攻击，但是如果在玩家 A 这一侧对玩家 B 造成了伤害的话，就采用这个结果。这样在状态 2 下就不会产生不一致的结果了。但是对 B 来说，从画面上看，自己明明完美地避开了攻击，为什么却受到了伤害？这样的结果实在无法接受。而另一方面对玩家 A 来说，在自己的画面上发生的情况毫无意外地反映在了最终的状态上，所以不会感觉到有任何不妥。

图 3.33　使用造成伤害的那一方的结果（时序图）



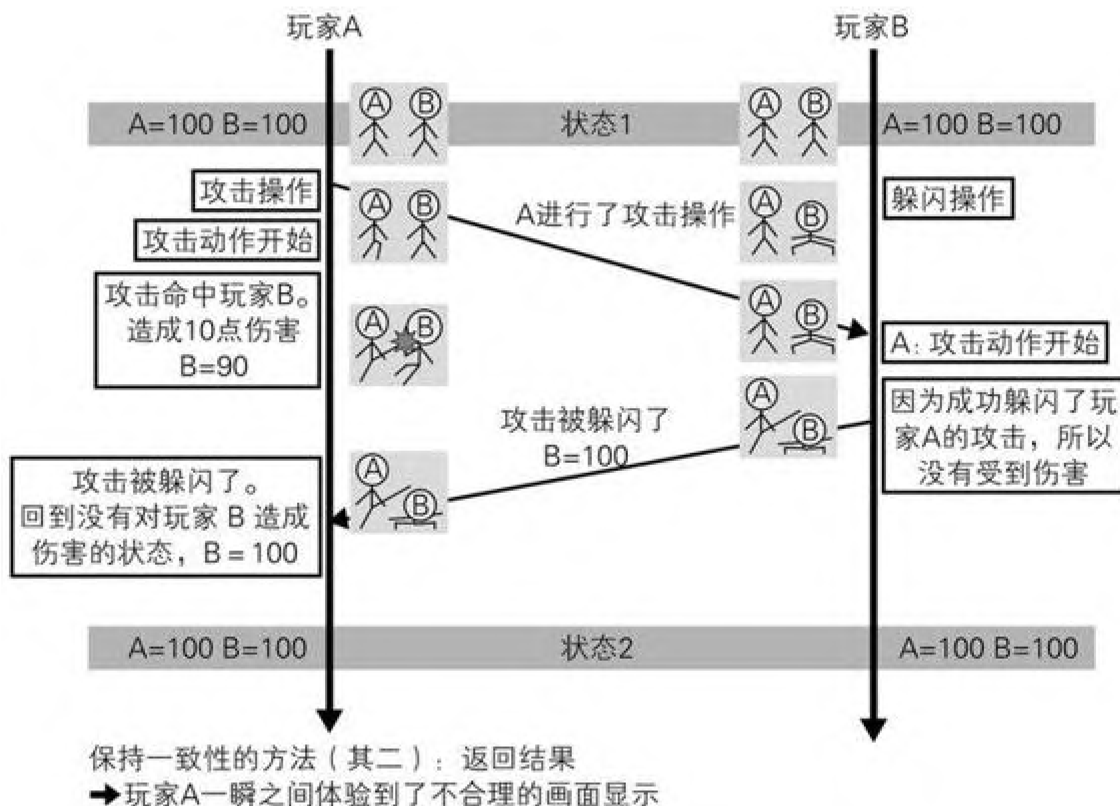
采用受到攻击的那一方的结果

图 3.34 则正好相反，这种方法重视受到攻击的那一方，发动攻击的那一方的结果将在之后被覆盖。

在图 3.34 的情况下，玩家 A 因为延迟的缘故，只发送了某些攻击操作的信息。如果玩家 B 没有受到伤害，就会把这个结果发送给玩家 A（B 没有受到伤害）。此时，对受到攻击的玩家 B 来说，这一结果毫无疑问，但是对发动攻击的玩家 A 来说，明明看到攻击成功了，但是一瞬间的延迟导致产生了没有造成伤害的结果，因而感觉在游戏进

行过程中发生了不合理的情况。使用这种方法也不会在状态 2 时造成数据不一致。

图 3.34　使用受到攻击的那一方的结果（时序图）



对这两种方式进行选择的原则 ——增加玩家的总体满意度

以上介绍了“使用造成伤害的那一方的结果”和“使用受到攻击的那一方的结果”这两种方式。就其结果而言，这两者都不完美，有无法接受的地方，也有不合理的地方，它们各自都有缺点。

那么到底应该如何进行选择呢？在大多数情况下，可以根据以下原则进行选择。

如果是不利的结果，就采用承受方的结果。如果是有利的结果，那就采用施与方的结果。

网络游戏的原本的价值就是通过游戏体验给予玩家一种满足感，所以在进行选择时就要从这个角度去考虑，采用能给玩家带来更多满足感

的方法。

比如在之前所举的格斗游戏中，进行攻击以对对方造成伤害的这一行为，对于受到伤害的这一方来说是绝对的不利情况，所以使用被攻击方的结果，而不是攻击方的结果。受到损害的玩家为什么受到了损害，如果不能对此做到正确把握，就无法使玩家获得满足感。

与此相对，如果是在多人动作游戏中给同伴回复体力的这种使对方获利的行为，即使在对方的画面上碰撞检测并不成立，但是仍然使其得以回复，那么该玩家就会感到非常满足。

能够通过上面这个方针来判断的只有那些“自己与对方之间存在直接的得失关系，以及游戏响应非常重要”的内容 [10] 。以直接的得失关系为主、而且响应又很重要的游戏包括对战格斗 /FPS 这类互相攻击的游戏，以及竞速游戏等。这类游戏需要更为具体的、与游戏过程相关的信息，所以需要尽可能频繁地发送抽象度较低的数据。

> [10] 此外，有关自己和对方之间的间接关系将在“自己和环境的关系”一节中介绍。在响应不怎么很重要的情况下，实现方法根据作弊行为是否会成为一大问题来分类。不会成为问题的话就使用之前已经讲过的同步方式，如果会有问题的就使用浏览器方式。

3.4.10 ❷ 自己和环境——可使用物品的格斗游戏和互斥控制

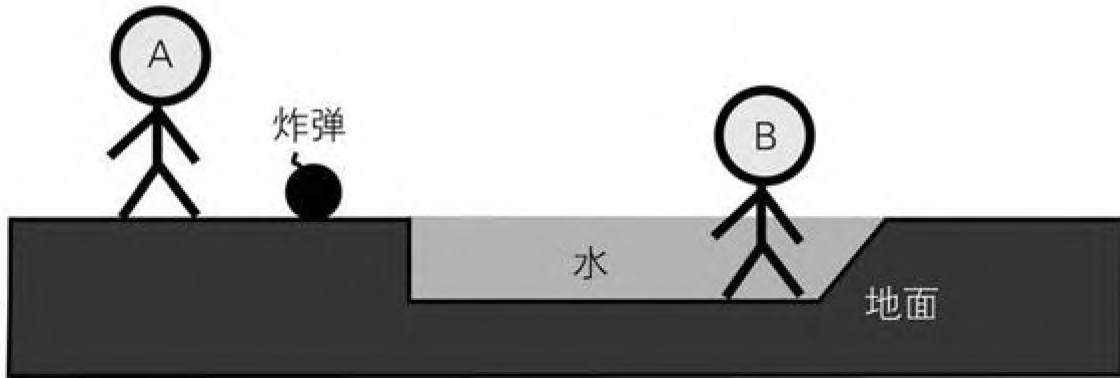
对于异步方式 3 个要素（自己、对手、环境）之间的 3 个关系：自己和对手、自己和环境、对手和环境，我们已经对自己和对手的关系进行了说明。接着我们来看一下自己和环境的关系。

“环境”就是既非自己也非对手的东西，包括地面等背景、掉落的物品、共同的敌人、天气情况，等等。“环境”要素根据“是否需要互斥控制”分为两大类。

需要互斥控制的环境要素 ——互相竞争的资源“炸弹”

比如在之前那个格斗游戏中，既有需要互斥控制的情况，也有不需要的情况（参见图 3.35）。

图 3.35　具有“环境要素”的炸弹游戏



在图 3.35 所示的游戏中有一条规则：在比赛场上会落下一颗炸弹[11]，而且只有这一颗，首先拾得这颗炸弹的玩家可以用它给对方造成极大的伤害。在这种情况下，一旦某个玩家首先得到了这颗炸弹，就必须确确实实地将其消除，不能再次获得它。这种类型的游戏资源称为“互相竞争的资源”。对战双方中只有一名玩家能够获得炸弹，这条规则是游戏可玩性的关键所在，所以绝对不能二度获取。这就是“需要互斥控制的环境”。

> [11] 用于在地面上炸开一个洞并且对对手造成一定伤害。

不需要互斥控制的环境要素 ——不会减少的资源“水”

与此相对，图 3.35 所示的游戏还有一条规则：在水中的玩家，移动速度和攻击速度减半。在这种情况下，虽然用到了“水”这个元素，但是它并不会减少，所以这是种不需要互斥控制的环境。一般来讲，在不需要限制事项发生次数的，或者事项的发生不会造成环境变化的、不需要向全体玩家告知这种变化的情况下，就不需要互斥控制。

游戏中的环境要素极难处理 ——必须详细理解游戏内容

环境的变化经常会导致晚家很难明确判断自己和对手之间的得失关系。

比如，由于炸弹能给对手造成伤害，所以起初会觉得如果自己得到了炸弹，对手就明显处于不利状态了。但是如果这个游戏还规定，炸弹使用不当也会给自己造成伤害，那情况又如何呢？或者，在有第 3 个玩家——玩家 C 的情况下，玩家 C 被炸弹击倒后对玩家 B 来说或许

更有利了。又或者，使用炸弹可以在地面上炸个洞出来，如何利用这样的地形，这之间的得失关系还要在之后才能确定。

在游戏中，自己和环境要素之间的关系并不像自己和对手这样这么直接，而是一种间接的关系。但是因为也有像炸弹这样会造成强大威力的元素，所以为了避免出现问题，必须进行一些技术处理。

3.4.11　互斥控制的实现——采用与同步方式类似的机制来实现异步方式

上面简要介绍了一下互斥控制，但是异步方式下的互斥控制实现起来并不容易。虽然已经采用异步通信发明了一致性算法（Consensus Algorithm），在 Google 的服务器内部也有使用 [12] ，但是这种算法需要多次往返传输消息，采用异步方式的网络游戏需要在几十毫秒的这段极短的时间内保证数据一致，这种情况下不能使用这种算法。

> [12] 比如分布式锁服务 Chubby 中使用了 Paxos 算法。
> http://labs.google.com/papers/chubby.html 、
> http://labs.google.com/papers/paxos_made_live.html

因此，在采用异步方式的游戏中，在实现自己与环境的关系时，通常使用之前所说的“与同步方式类似的机制”来解决。

物品复制问题

那么我们使用炸弹的例子来看一下应该要解决的一些问题（参见图 3.36）。

图 3.36　时序图（炸弹游戏）

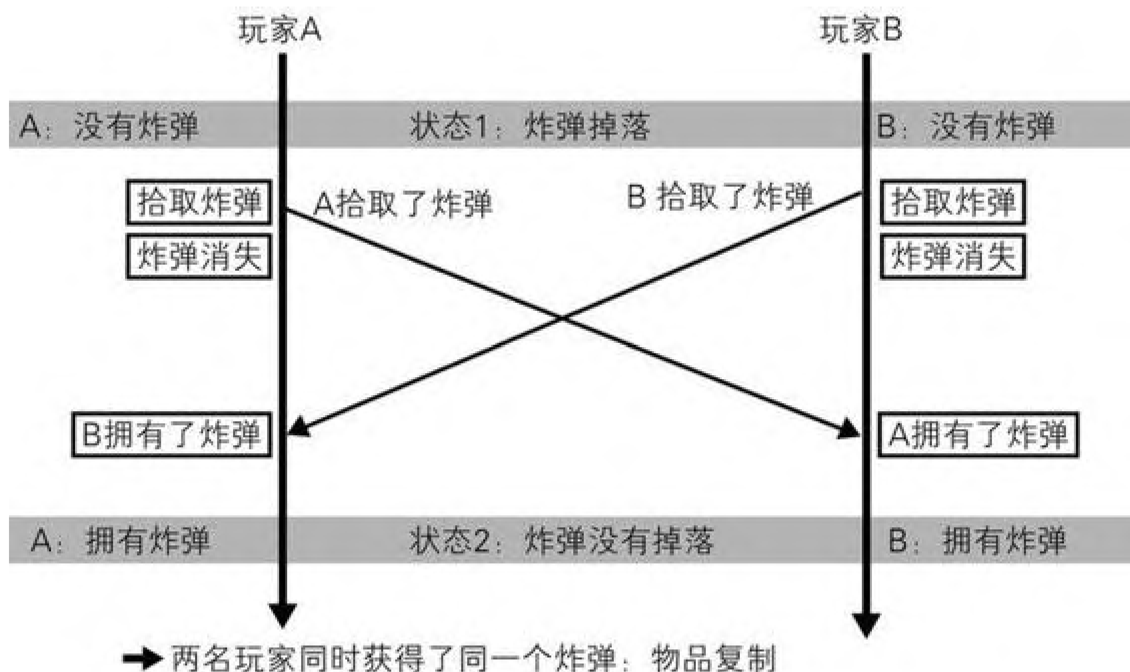


图 3.36 显示了从“状态 1：掉落 1 个炸弹”到“状态 2：不掉落炸弹，两名玩家各持有一颗炸弹”的时序。问题是，明明只有 1 个炸弹，最后却增加到了两个。这在网络游戏的术语中称为“物品复制”（Item Dupe），也就是说游戏内的物品被复制了。

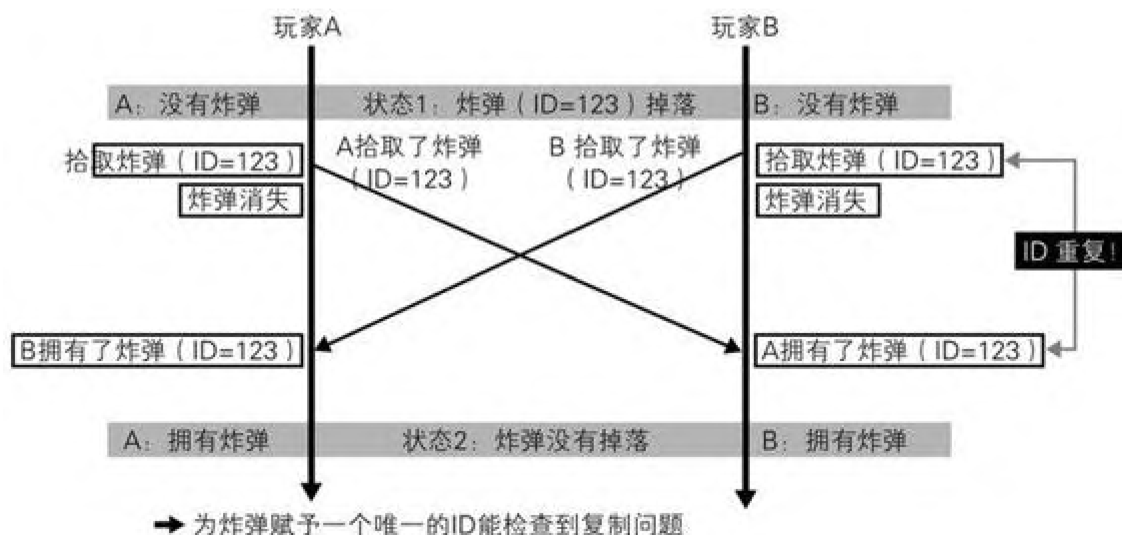
在网络游戏中，由于没有解决这一问题（可能在经过考虑之后决定置之不理）而导致物品复制现象猖獗，最终缩短游戏寿命的例子数不胜数。无论如何，这个问题都必须解决。

一开始，只有 1 个炸弹掉落地面，玩家 A 和玩家 B 可以在短于传输所需的时间内取得炸弹，于是就发生了物品复制。这种类型的复制可以有意而为，比如，在可以“放置炸弹”的情况下，反复进行炸弹的放置、拾取操作，几次里有 1 次是在微妙的时刻进行了操作，就能够进行复制。

给物品赋予唯一的 ID ——判断物品是否被复制，发生的问题

通过为炸弹赋予一个唯一的 ID，可以判断物品是否被复制了（参见图 3.37）。如果能够判断是否发生了物品复制，接下来就可以采取以下这些手段。

图 3.37　赋予唯一的 ID 后进行判断



❶ 被复制了的物品毫无疑问要消除

获取物品是构成玩家满足感的基本要素，所以要消除相当有害的物品（比如《超级马里奥兄弟》中碰到后就会受到伤害的毒蘑菇等）时不能使用这种方法。

❷ 允许复制

如果存在有意进行反复操作的玩家，物品“以稀为贵”的价值就降低了，从而导致玩家满足感降低，游戏寿命缩短。但是，如果该物品对游戏过程的影响很小，只是用于显示则可以进行复制。

❸ 通过某些规则确定优先级最高的玩家

将物品给予优先级最高的玩家，而从优先级较低的玩家处剥夺该物品的所有权。比如，玩家 ID 号小的玩家优先、偶数号码的玩家优先；或者等级较低的玩家、刚开始游戏的玩家优先等。考虑到应尽可能不让玩家的满足感降低，或许让刚开始游戏的玩家优先获得物品更为自然。但是这也可能产生一个弊端：有经验的玩家申请多个账号，假装自己刚开始游戏。

❹ 使用专用的交易接口来解决

发生物品复制时，会有一种“发生物品复制了，请猜拳决定”的感觉，为此，可以采用专门用于解决复制问题的接口。但是在这种情况

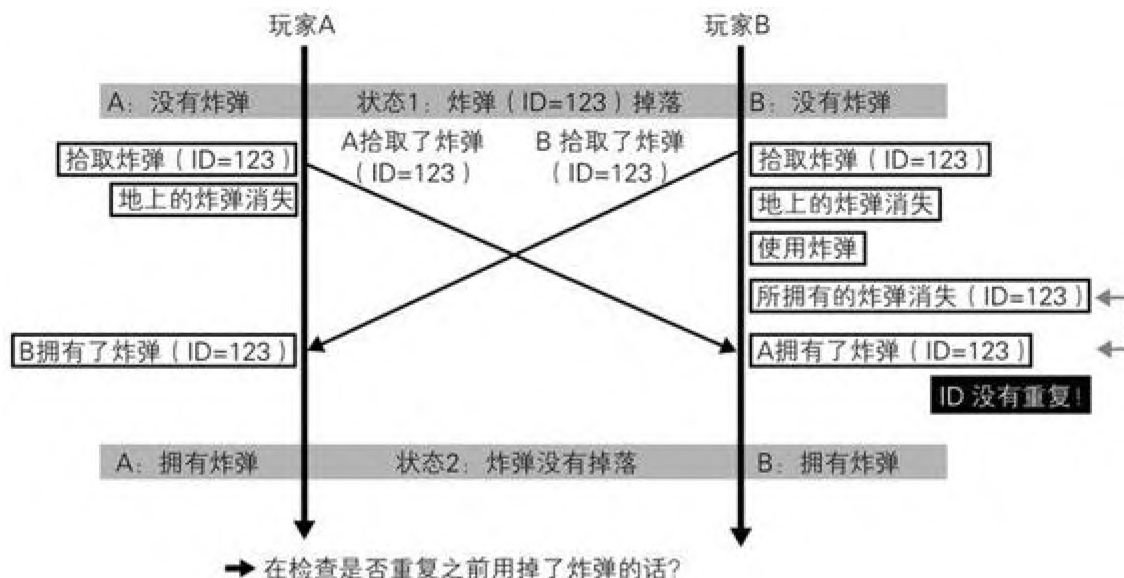
下，直到从所有相关的玩家处获得猜拳结果之后才会决定由谁获得该物品，这一点可以部分引入同步方式来解决。

❺ 允许复制，但是通过其他方法减小影响

比如在游戏结束时检查物品总数。如果游戏结果保存在服务器中，可以将最大值设为每天 10 个，在保存时进行检查，超过最大值的话就无法保存或者弹出警告或者进行封号判断。此外还有种方式，事先共享“物品最多可能出现 5 个”的信息，之后进行核对。这样多少可以防止将超出部分作为游戏结果保留下来。

通过给物品赋予一个唯一的 ID 值来解决物品复制问题还是很简单的，但是从拾取到物品开始至进行复制检查为止的这一段时间却是一个问题。图 3.38 显示了拾取到炸弹后，在进行复制检查之前使用了该炸弹的情况。玩家 B 拾取到炸弹后，在接收到玩家 A 拾得炸弹的消息之前，已经使用了该炸弹。使用之后所持有的炸弹就消耗掉了，所以在复制检查时也就不包括在内了。明明只掉落了 1 个炸弹，但是结果却被拾取了两次。

图 3.38　在进行复制检查时已经把炸弹用掉了

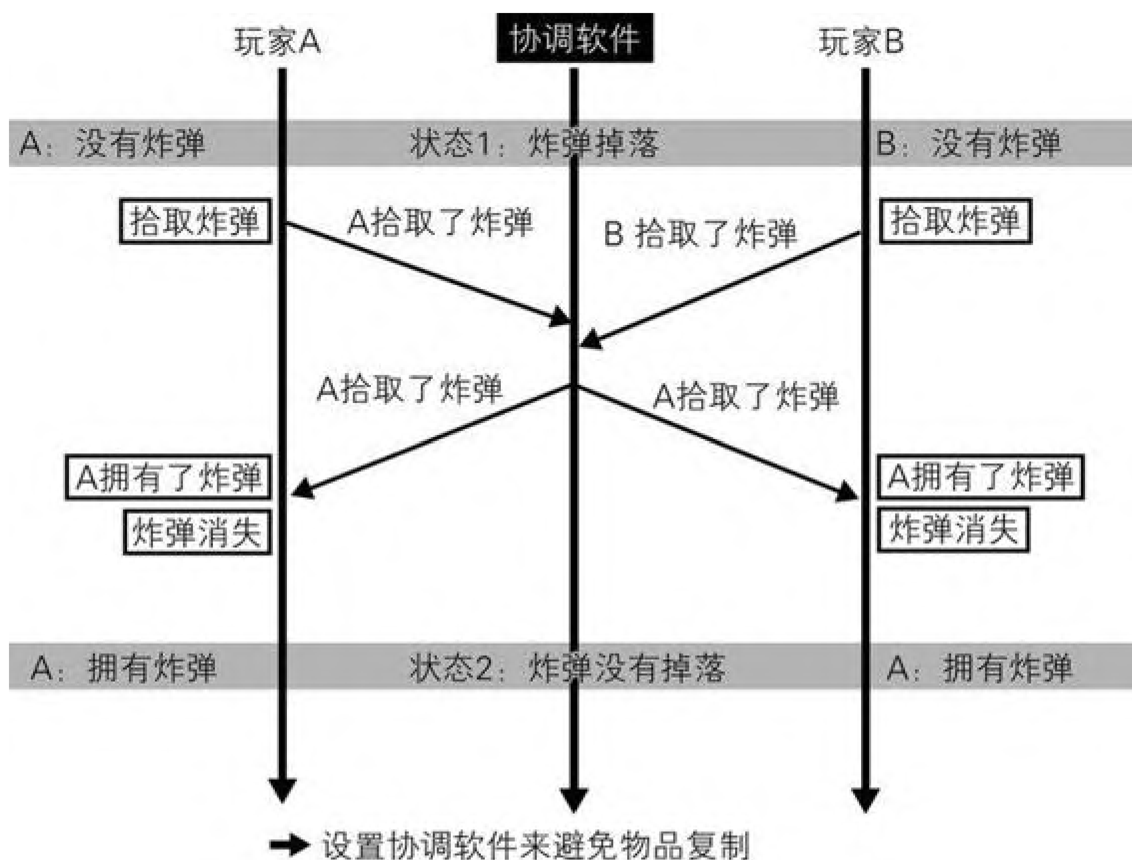


要解决这个问题的就必须进行一些非常复杂的处理：保留过去所有的游戏经过，一一检索……

物品复制的解决对策 ——由专门的软件负责协调

为了解决上面这个问题，可以采用第三方的负责协调的软件（而非玩家 A 或玩家 B）来控制炸弹拾取操作的顺序（参见图 3.39）。

图 3.39　设置用于协调的软件



在图 3.39 中，在玩家 A 和玩家 B 之间设置了协调软件，将获得炸弹的相关操作首先发送给协调软件。协调软件通常只有 1 个，安装在（数控）计算机上，即使多个玩家同时进行操作，所发出的消息也一定会在不同的时刻到达。在图 3.39 中，来自玩家 A 的数据包早到了一点点，因为协调软件知道炸弹只有 1 个，拾取后就会消失，所以就根据这一情况进行判断，最后将“玩家 A 拾取了炸弹”这条消息同时发送给玩家 A 和玩家 B。

通过图 3.39 可以知道，在这种方式下不会发生物品复制。而且也不会发生在进行复制检查之前就使用了该炸弹的情况。这种方式有一个

缺点，在与协调软件完成传输之前，拾取炸弹的操作不会结束，这样就会感觉有点延迟 [13] 。

> [13] 在行业术语里也称为“反应迟缓”。

协调功能的实现可以采用如下两种方法：❶ 转移到某个玩家的终端上、❷ 在服务器端实现。在实际的游戏中，这两种方法都有使用。在采用同步方式的游戏中，虽说使用方法 ❸ 会在服务器带宽上花费很大的成本，实现起来不大现实，但是在这种情况下，在实际产生的传输量中，拾取炸弹的操作占 1%，除此之外，击打、踢、移动等基本行为占 99%，所以通过“部分并用”，服务器成本并不大。

协调软件的基本功能和使用方法

不管使用上述 ❶、❷ 的哪种实现方法，协调软件的基本功能都是相同的。下面以伪代码的形式（类似于 Java）对这些功能进行说明。

- **为值赋予一个名字来保存：set(key, value)**

  key 是赋予值的一个名字，value 是要保存的值。

- **读取值：get(key)**

  key 是值的名字，将 value 作为结果返回。

- **通过附加条件改变值：setIf(key, currentValue, newValue)**

  key 是名字。现在，保存在协调软件内的值 currentValue 如果一致，就保存为 newValue，不一致的话该函数就返回失败。这个功能是“完全的互斥控制”的基本实现方式。

- **值发生变更后，通知所有参与者：changed(who, key, value)**

  如果 set 和 setIf 成功后值发生变化了的话，协调软件就将变化后的值发送给所有的玩家终端。key 是变化了的值的名字，value 是变化后的值。另外也会发送是哪个终端更改的。

协调软件所必须具备的功能就是这些了，很简单吧。下面我们以炸弹的例子来了解一下协调软件的使用。

- **比赛开始时**

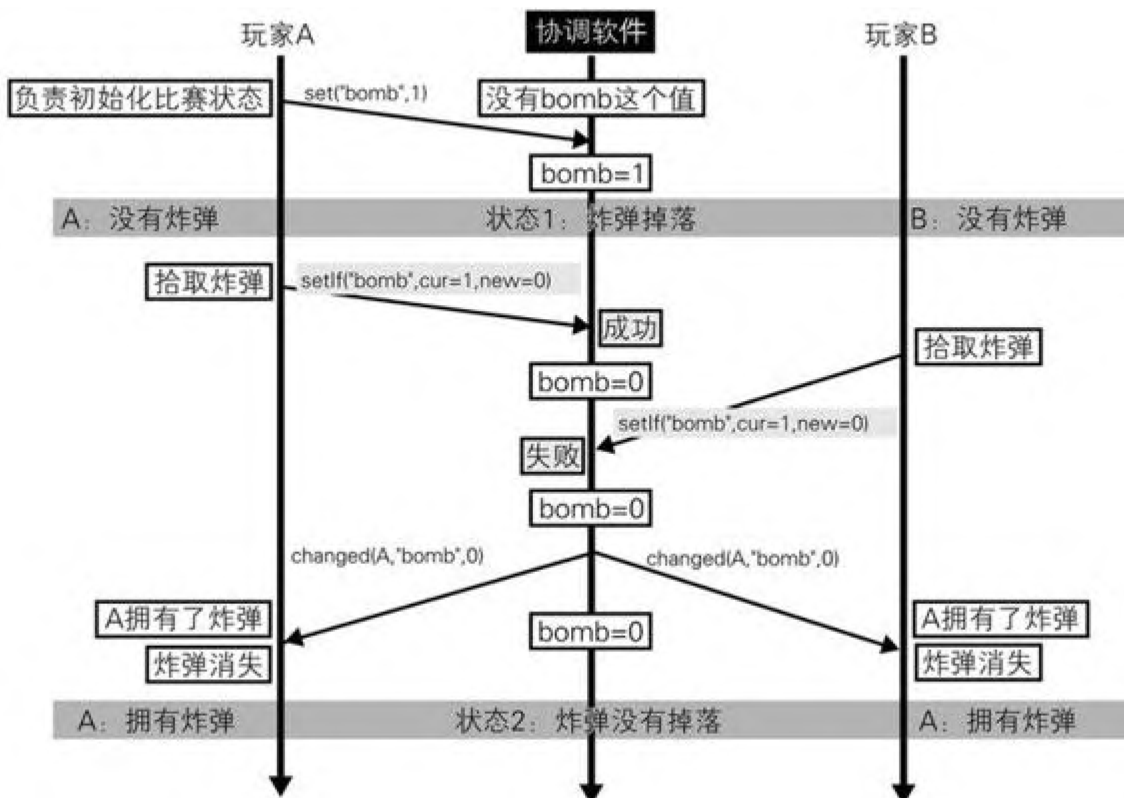
  通过 set("bomb", 1) 设置炸弹。开始比赛时是同步开始的，不管由哪个玩家来向协调软件发送必要的信息都没有问题。“1”这个值以 "bomb" 为名进行保存。“1”就是炸弹的个数。

- **比赛中**

  拾取炸弹时调用 setIf("bomb", currentValue=1, newValue=0)，只有当炸弹在协调软件内部为 1 个时，这条语句才会成功地将值更改为 0，然后发送 changed("bomb", 0)。如果玩家 A 和玩家 B 同时拾取了炸弹，协调软件就会收到两次拾取炸弹的消息。但是 changed("bomb"，0) 只会群发一次。

以上过程的时序图如图 3.40 所示。

图 3.40　协调软件的作用（伪代码）



在图 3.40 中，首先玩家 A 负责比赛状态的初始化（由玩家 B 负责也可以）。然后炸弹掉落，进入状态 1。

接着在游戏的进行过程中，玩家 A 和玩家 B 几乎同时拾取了炸弹，然后向协调软件发送了同一个函数。在协调软件中，调用 setIf 函数进行处理。因为 cur=1、new=0，只有在当前值为 1 时才会成功写入 0。在从玩家 B 处接收到拾取炸弹的消息时，协调软件已经完成了对玩家 A 的操作的处理，所以当前值就改为 0 了，此时 setIf 返回失败。因此，协调软件向所有玩家发送表示“数据被玩家 A 改变了”的消息：changed(A, ″bomb″, 0)。接收到这条消息后，各个终端都能意识到是玩家 A 获得了炸弹，然后消除显示在画面上的炸弹。

炸弹以外的环境要素

在实际的游戏中，除了炸弹之外，还有其他各种各样的环境要素，但是基本上都可以通过组合以上所述的协调软件的功能来实现。

- **放置的物品只能获取 1 次（炸弹的例子就与此对应）**
- **门和开关、控制杆**
    - 起初是关上的，之后只能打开一次

      →这种情况下不需要互斥控制，只要发送 set(″door″, OPEN) 就可以了。
    - 每一次操作都可以关上打开着的物体、打开关着的物体

      →这种情况下实际上也不需要互斥控制。如果在玩家 A 这一侧，门是打开着的，而玩家 A 又没有进行操作，不需要发送 set(″door″, CLOSED) 的话，就只要发送 set(″door″, OPEN) 就可以了。
- **布阵**

  →如果只能获取一次的物品以日本地图的形状来配置多个，也只是归结为有很多炸弹的问题。地形的变更也是同样。
- **争夺排名**

→在竞速游戏中，必须确定作为比赛结果的排名。在胜负难分的情况下，由于是在非常接近的时刻到达终点的，所以必须进行互斥控制。设法只使用 setIf 来实现也是可能的。比如，在参与者为 3 人的情况下，3 名玩家各自到达终点（在各终端进行判定）的消息：setIf("1st", cur=0, new=1)；setIf("2nd", cur=0; new=1)；setIf("3rd", cur=0, new=1) 将会发送给协调软件。如果协调软件发送了 changed(A, "1st", 1); changed(C, "2nd", 1); changed(B, "3rd", 1)，那么就可以知道排名就是 A、C、B。也有不使用这样的方法，在协调软件中实现专门进行这种处理的功能。

3.4.12　状态会自动变化的环境——静态环境和动态环境

在自己和环境的关系方面，至此所述的炸弹示例中，炸弹是放置在地面上的，是一种不会自己移动的环境。但是在游戏环境中也有很多状态会自动变化的环境。这也能用同样的方式来处理吗？当然没有这么简单。

游戏环境分为“静态环境”和“动态环境”。静态环境指的是只有在玩家进行某些操作之后才会发生变化的环境。而动态环境则是指，即使玩家没有进行任何操作，也会持续发生变化的环境。

至此为止所介绍的掉落在地面上的物品、门、开关、布阵、排名争夺等环境要素都是在玩家进行操作之后才会发生变化的静态环境。

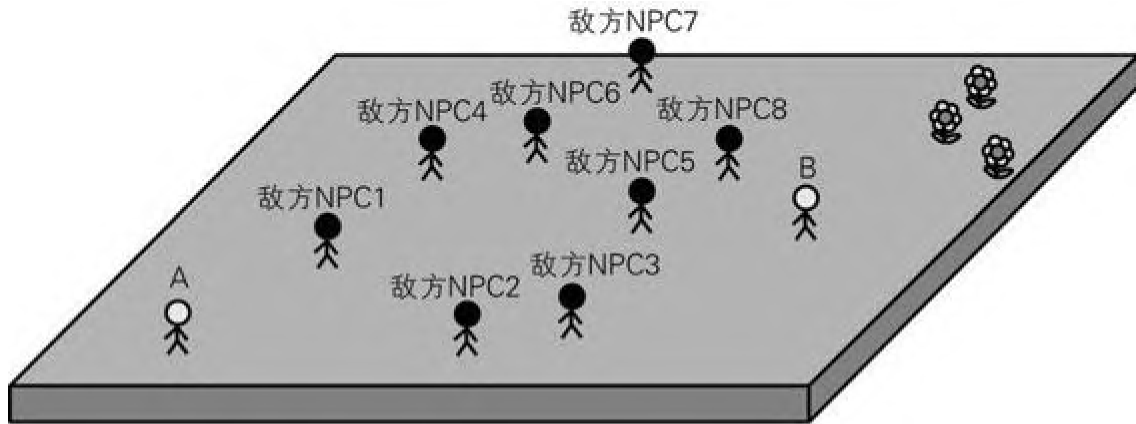
与此相对的动态环境的典型代表则包括一直在四处移动的敌人、以物理方式运动的物体，等等。为了不让玩家感到不和谐，在实现动态环境时需要花一些功夫。

动态环境引起的问题 ——很难完全并行管理

首先来考虑一下动态环境所引起的问题吧。

在图 3.41 中，平坦的地面上有玩家 A、玩家 B，以及 8 名敌方角色。敌方角色由程序控制。这种由程序控制的角色在技术术语中称为 NPC（Non-Player Character）。NPC 编为 1～8 号。

图 3.41　动态环境的例子



即使玩家不进行操作，NPC 也会自己行动，也就是说，移动、踢、拾取炸弹等行为都是自动进行的。NPC 通常具有与玩家同等以上的能力。NPC 始终运动着，并非特定玩家的操作对象，所以可以说是动态环境。

游戏中敌方 NPC 的典型动作如下。

- **逼近距离最近的玩家（移动）。**
- **如果玩家进入了攻击范围，则对其发动攻击（踢）。**

NPC 每 16 毫秒进行移动等行为，坐标和速度等状态一直在改变。那么这些 NPC 的信息存放在哪里的内存中，由哪里的 CPU 来进行处理呢？

图 3.41 的游戏所涉及的只有玩家 A 的终端和玩家 B 的终端。NPC 的处理就要在其中一方进行。

作为最原始的模式，首先考虑在玩家 A、B 两方的终端上各自并行进行管理。游戏刚开始时，玩家 A 和玩家 B 处于完全相同的初始状态下，如图 3.42 所示。该图比图 3.41 更为模块化，图中的大四边形代表游戏的整个区域。白色圆圈代表玩家 A 和 B，黑色圆圈代表敌方 NPC，敌方 NPC 的数量减少到了 3 个。NPC1（图中 1）距玩家 A 较近，NPC2 正好位于玩家 A、B 正中位置，NPC3 距玩家 B 较近。

图 3.42　游戏开始时的初始状态

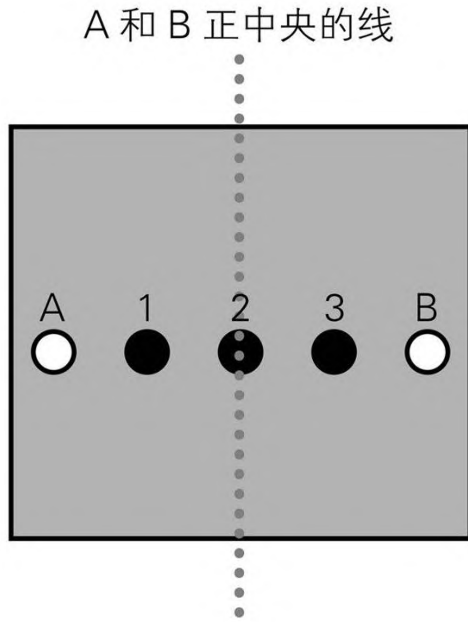


图 3.43、图 3.44 对玩家 A 和玩家 B 的状态进行了对比。当然初始状态是相同的（图 3.43❶）。

图 3.43　状态变化 ❶

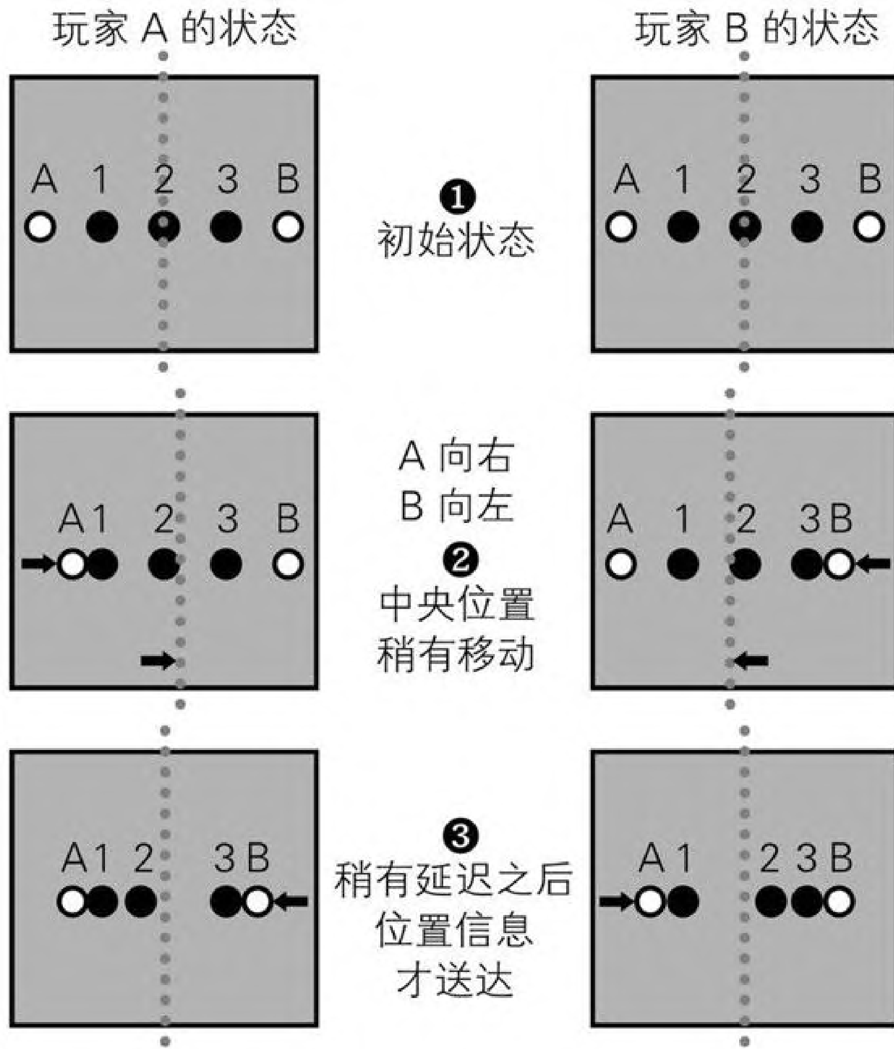
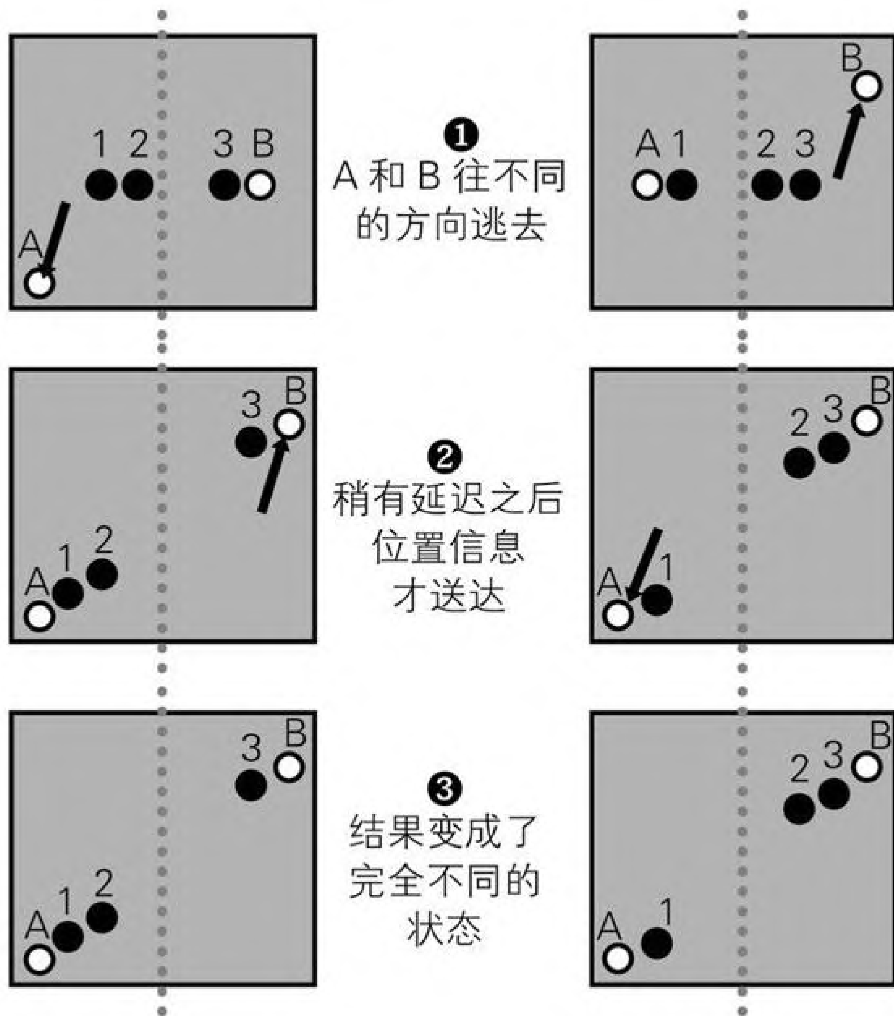


图 3.44 状态变化 ❷



在下一刻，玩家 A 向右移动，玩家 B 向左移动（图 3.43 ❷）。在这一行为的影响下，玩家 A 和玩家 B 之间的中心位置在玩家 A 的终端上稍稍向右偏移，而在玩家 B 的终端上则稍稍向左偏移。因此，对于 NPC2 来说，在玩家 A 的终端上，最近的玩家是玩家 A，而在玩家 B 的终端上则是玩家 B。在这个阶段中，由于网络延迟，各个玩家的移动信息尚未送达。

结果，在下一个瞬间，在玩家 A 的终端上，NPC2 为了追赶玩家 A 开始向左移动，而在玩家 B 的终端上则向玩家 B 的方向移动。此时，各个玩家的移动信息终于送达，反映在了画面上（图 3.43 ❸）。

在图 3.44 ❶ 中，玩家 A 与玩家 B 往完全相反的方向逃去，但是因为是朝着中点对称的方向移动，所以中央线的位置没有变化。

接着，图 3.44 ❷，敌方 NPC 继续向最靠近自己的玩家追去，在玩家 A 的终端上，玩家 A 被 NPC1 和 NPC2 两名敌人追击，另一方面，在玩家 B 的终端上，玩家 B 被 NPC2 和 NPC3 追击。相互之间的位置更新仍然有所延迟。

最终，产生了完全不同的结果（图 3.44 ❸）。在图 3.43～图 3.44 的例子中，因为 NPC“持续追赶距离最近的目标”，所以“一点小小的差异累积起来就会产生极大的差异”。在游戏中这种情况是很普遍的。

由此可知，完全并行的管理方式是有很大问题的。

解决动态环境所引起的问题的几种选择方案

实际使用的解决方案有如下几种。

❶ 所有 NPC 的相关信息都在玩家 A（或者玩家 B）的终端上进行处理

→在这种情况下可以很简单地解决之前的问题。总是按照单个玩家所管理的坐标信息进行处理。这里的问题是，不负责处理的玩家对 NPC 采取踢等行为所产生的结果（对 NPC 造成伤害）总是通过另一方的终端来发送，所以很容易感觉到通信延迟。而且因为所有 NPC 的所有行为都必须通过网络来传输，所以传输量大增。尤其是在延迟方面，在游戏内容较为严苛的情况下会让玩家感到不公平，可能降低玩家的满足感。此外传输量问题也导致这种方法无法在移动终端上使用。

❷ 定时修正 NPC 的位置

→继续进行基本的并行处理，但是定时发送所有 NPC 的坐标，在差异较大的情况下，强制采用某一个终端上的坐标。这种方法的问题就是，偶尔会发生正在追赶自己的敌人突然消失了，或者从未存在过的敌人突然出现了的状况。但是只要能尽可能频繁地进行修正处理，至少能避免突然大规模发生这种现象。实际上不仅仅是坐标，伤害值等信息也必须加以修正。与方法 ❶ 相比，这种方式用几十分之一的传输量就足够了。对 1 秒内发送多次相关信息所需的传输量，与几秒发

送一次的情况下所需的传输量进行比较，由此找到最合适的修正频率。

❸ 根据某些规则将 NPC 进行分组

a. 根据 ID 编号的奇偶等机械地进行分组

→看上去很简单也很不错，但是通常比起这个方法，方法 ❶ 更好一些。比如，有两名玩家正在进行游戏，其中一半的 NPC 有延迟，有延迟的 NPC 与没有延迟的 NPC 混在一起，这种情况下的游戏体验还不如所有的 NPC 都有延迟。

b. 根据某些条件转移管理权限

→这种方法相对用得较多。比如，根据 NPC 出现的时间，在距离较近的玩家的终端上进行管理，每隔一段时间计算与各个玩家之间的距离，然后将管理权限转移给最近的那个玩家的终端。在之前所举的 NPC“向较近的玩家靠近”的例子中，这种方法尤为有效。游戏基本上是以与逐渐逼近的 NPC 交互为中心的，使用这种方法可以降低距离自己较近的 NPC 的延迟，从而在很大程度上防止游戏体验的恶化。这种方法也有个问题，在转移管理权限时，必须暂时（大约几百毫秒至 1 秒左右）停止 NPC 的行动。传输量比方法 ❶ 少，比方法 ❷ 多。

c. 使用 AI 管理的信息来进行分组 [14]

> [14] 第 5 章中的示例代码（P2P MO）所使用的就是这种方式。

→这也是较为常用的一种方法。这里所说的 AI 就是指人工智能（Artificial Intelligence），这里所指的并非一般意义上的人工智能，而只是 NPC 的行为算法。在游戏行业中，即使是“冲向最近的玩家”这样简单的动作也叫做 AI。

在实际的游戏中，NPC 的行为并没有这么简单。一种典型的方法是，NPC 具有一个称为“仇恨值”的数值。仇恨值可以在敌人受到玩家攻击时增加，一段时间之后降低。比如，某个敌人最初的仇恨值为 0，在受到了玩家 A 的攻击之后，该敌人对玩家 A 的仇恨达到 100，然后该敌人不再逼近距其最近的玩家，而是冲向仇恨值最高的玩家，也就是玩家 A。这个仇恨值每过 1 秒就减少 1 点，所以 100 秒之后该

敌人就会忘记玩家 A，不再继续追赶该玩家。而在攻击过后 50 秒，该敌人对玩家 A 的仇恨值降至 50，此时另一个玩家，比如玩家 B 向该敌人发起攻击，该敌人对玩家 B 的仇恨值就达到了 100，于是转而开始追赶玩家 B。但是如果之后玩家 B 被打倒了，那么该敌人对玩家 B 的仇恨值就会恢复为 0，然后再次开始追赶玩家 A。采用仇恨值作为基本 AI 的游戏很多，在这种游戏中，基本的游戏玩法就是与跟自己有仇的敌人作战。也就是说，由于作战对手主要就是这个对自己仇恨值最高的敌人，所以如果在自己的终端上管理这个敌人，玩家所感觉到的延迟就会降低，从而提高玩家的满足感。

＊ ＊ ＊

环境会怎样变化呢？想给玩家带来怎样的游戏体验呢？数据的不一致会造成多大程度的影响？通过以上的介绍，我们知道必须在考虑游戏内容的同时选择相应的解决方案。

静态环境和动态环境的相关技术就介绍到这里。以上我们以“自己与对手的关系”、“自己与环境的关系”为中心进行了说明，最后剩下的就是“对手与环境的关系”。

3.4.13 ❸ 对手和环境的关系

就结果而言，大部分游戏并不重视自己以外的玩家与环境之间的关系。比如，不需要接收“敌方 NPC 攻击了自己以外的玩家”这样的信息。

在很多游戏中，自己以外的玩家的情况只是出于“游戏显示”方面的考虑，只在游戏画面上反映那些必不可少的部分。这么做的原因有如下几点。

- **在游戏画面上表示其他玩家会加重渲染处理的负担。**
- **增加传输量。**
- **画面上表示的物体过多的话会引起混乱，导致游戏困难。**

但是其他玩家的信息也不应该完全没有。以下这些内容就需要传达给玩家。

- 如果是以聚会为中心的游戏[15]，必须向参加聚会的成员通知所发生的一些重要变化。假设自己与对手的关系所需的信息量为 10，那么那些变化所需的就要在 5 左右。

- 在 MMORPG 等想要让玩家感觉到是一个世界的情况下，远处所发生的事件即使只能粗略表现出来，也应该传达给玩家。假设自己与对手的关系所需的信息量为 10，那么那些远处的事件大约在 0.5 左右。

[15] 多名玩家聚集在一起共同进行的游戏。

在网络游戏中，为了传达对手和环境之间的关系，大多会采用大幅降低传输内容、仅传达一种氛围而非精确的信息等方法。在上面的例子中，对手和环境的关系所需的信息量是自己和对手的关系所需信息的一半至 1/20，在实际的游戏中，为了判断具体的信息量，必须对游戏的细节内容进行详细分析。

比如，在两人对战格斗游戏中，只有自己和对手，因此并不存在对手和环境这样的概念。与此相对地，在 10 对 10 的集体对战游戏中，远处的玩家和环境的关系只需发送极少的一部分信息。通常，在发送 1 秒内移动 10 次这样的数据包时，1 秒内只发送 1 次。这样传输量就很简单地降到 1/10 了。此外，更远一些的情况则通常不在画面上显示，而只是播放一些音效。这样可以大幅降低渲染负担。

综上所述，在对手和环境的关系方面的开发方针在很多情况下并不重要，所以只要“分析游戏的策划内容，讨论要减少多少信息”。

* * *

以上我们介绍了作为异步方式 3 大要素之间的基本关系“自己与对手”、“自己与环境”以及“对手与环境”，并且分别讲解了制定这种实现方针的要点。在异步方式下，需要通过对游戏的详细理解和分析，根据各自的关系确定方针并且加以实现。

3.5　逻辑架构详解——MMO 架构

逻辑架构中的另一种基本架构就是 MMO 架构。

3.5.1 MMO 架构、MMOG——在大量玩家之间共享长期存在的游戏过程

MMO 架构就是“在大量玩家之间共享长期存在的游戏过程”。为此应该尽可能防止游戏的过程信息被破坏。在发生 bug 等异常问题时，需要对游戏数据进行回退。

MMOG 也叫做“持久的游戏”，英语中称为 Persistent game、Persistent world 等。

什么是持久 ？——游戏所需的时间和积累性

这一点我们在 2.8 节中简单地接触到一些，这里所说的“持久”究竟指的是什么呢？其实就是一系列的游戏过程持续多长时间。我们以持续时间的长短来列举一下。

- **2～3 分钟**

  在街头霸王系列这类典型的对战格斗游戏中，一个回合大约几秒至几十秒，一场比赛有 3 个回合，所以 2 名玩家的比赛大约持续 2～3 分钟。胜负情况作为比赛（游戏）结果将被记录下来。赛车类游戏也是一回合 2、3 分钟～5 分钟左右，跑完全程的时间和排名也会被记录下来。

- **20 分钟～1 小时**

  FPS 和 RTS 的对战通常持续 20 分钟～1 小时。

- **1 小时～几小时**

  将棋和《奥赛罗》、《大富翁》（*Monopoly* ）和《人生游戏》（*The Game of Life* ）等棋盘游戏持续的时间稍长一些，大约 1 小时至几小时。

- **10～20 小时**

  以模拟城市为代表的模拟类游戏大约持续 10～20 小时。

- **30～200 小时**

  RPG 游戏可以达到 30～200 小时。

- **1000～5000 小时以上**

  在 MMORPG 游戏中，大多具有游戏时间长达 1000～5000 小时以上的大规模的游戏内容。

以上这些游戏所需时间的长短，根据游戏内容从 2 分钟到 5000 小时不等，它们之间的差异达到了 15 万倍！

网络游戏中说得上“持久”（Persistent，对人们来说相当于一生）的游戏包括 MMORPG 和诸如《网页三国志》这样的大规模对战网页游戏等。

在游戏行业中，花费时间很长的游戏称为“高累积性的游戏”。也就是说，玩家的游戏时间等重要财产在游戏数据中累积的程度很高。玩家投入了大量时间的游戏数据，其相对价值也因此而得以提高，所以为了防止游戏数据遭到破坏，游戏系统必须具有很高的可靠性。

保证持久性数据、不断累积的大量数据的一致性的难度

那么，人们可以连续进行游戏的时间上限一般是 45 分钟～1 小时，在利用连休长时间进行游戏的情况下，也就 8 小时，最多十几个小时。

在一系列的游戏过程在几分钟内就结束的游戏中，游戏内容只会在游戏从开始到结束的几分钟之内保存在内存中，只要在这段期间保证数据正确就不会发生任何问题。

但是在那些游戏时间长达几十小时、几百小时以上的游戏中，中途中断游戏后，不仅需要某种机制能支持之后继续开始游戏，而且因为众多玩家共享这耗时颇长的游戏，所以还必须在服务器的内存和磁盘上准确无误地、完整地保存游戏中的各种信息，当玩家需要时瞬间取出来展现给玩家。因为有永久存在的含义，所以称为 persistent。在 RAM 上互不干扰地、保证一致性地维护大量游戏数据是需要非常注意的。

在持久性的游戏中，MMORPG 和虚拟世界这类网络游戏出现了一种动态持久化类型（dynamic persistent）的游戏，在服务器内部实现物理模拟和经济模拟结构，游戏的设置数据会持续变化。在这种情况下，当玩家再次登录游戏时，即使玩家什么也没做也能感觉到跟以前不一样了。这类游戏中的设置数据包括游戏地图的形状，生物、物品的出现场所及其出现频率的设置，数据量高达几十吉字节至几十钛字节。因为这种规模的数据始终在动态变化着，所以对这些数据的保存、以及在其遭到破坏时进行修复的技术是非常必要的。

客户端和服务器的完全分离

因为数据一致性方面的要求非常严苛，所以在构建系统时要将“游戏客户端或者说游戏浏览器”和“游戏服务器”完全分离。

严格来讲，物理架构和逻辑架构之间没有什么关系，不具备服务器的 MMO 游戏在理论上也是可以实现的，但是以现在的技术还无法实现。所以目前所有 MMOG 都是 C/S 架构的。下面我们将对此进行详细说明。

3.5.2　MMOG 的结构

在 MMO 架构中，对游戏过程的管理和保存全部都由数据中心负责，由数据中心持续地将游戏结果发送至玩家终端。玩家使用专门开发的游戏浏览器来观察这些结果，使用鼠标、键盘和控制设备等将操作信息发送至服务器（参见图 3.45）。

图 3.45　MMO 架构

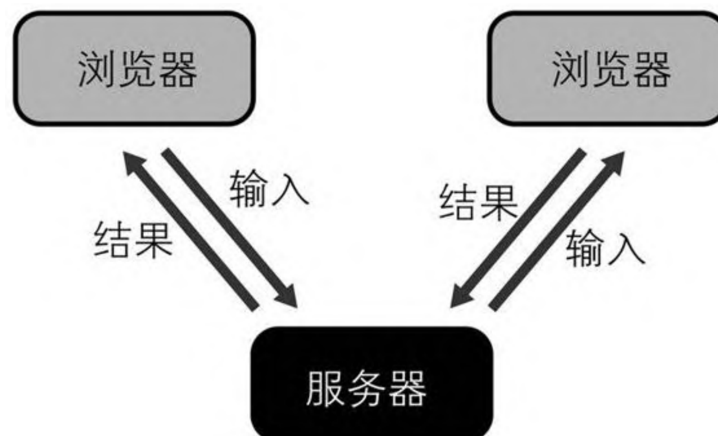


图 3.45 中的“浏览器”指的是游戏客户端（游戏浏览器、游戏浏览软件），比如 3D 游戏中，指的就是使用 C++ 等语言开发的本地应用程序，而在 Flash 游戏中，指的就是运行在 Google Chrome 等 Web 浏览器上的 Flash 应用程序。慎重起见我们说明一下，这里说的“浏览器”并不是 Web 浏览器的意思。

MMO 架构的实现方针 ——浏览器方式、纯粹的 C/S 模式

不论是 MO 架构还是 MMO 架构，都可以通过“同步方式”、“异步方式”和“浏览器方式”来实现，是 MMO 架构时，因为游戏内容有如下这些特点。

- 持久性
- 累积性
- 大规模（大量玩家同时在线）

所以能够使用的方法有所限制。

就其结果而言，能用的只有“浏览器方式”，其他几种实现方法基本上都不合适。慎重起见，我们还是对各种方式作一下简要说明。

- 可以使用同步方式吗？

  →只要 1 个人处理延迟，所有玩家都只能等待，所以同步进行几百个玩家的程序处理是很不现实的。

- 可以使用异步方式吗？

  →因为游戏内容是持久累积的，如果能在客户端侧篡改游戏数据则会给其他玩家造成很大的影响，所以不能使用异步方式。

- 浏览器方式呢？

  →可以使用。但是需要设法最小化服务器与客户端之间的传输量，减少操作频率，考虑能够负担较大通信延迟的游戏内容。

浏览器方式、同步方式和异步方式的差异

首先我们比较一下浏览器方式和同步方式，这两种方式的差异在于“传输内容”。同步方式下收发的只有玩家输入的信息（也就是游戏的原因），而在浏览器方式下，浏览器向服务器只发送玩家的操作信息（也就是原因），服务器只向浏览器发送游戏过程中的结果。

此外，在同步方式和异步方式下，共享游戏过程的所有终端都共享游戏过程中的所有主数据。而在浏览器方式下，管理游戏数据的只有服务器，各个终端（浏览器）只是将当前的游戏情况可视化展现给玩家，这一点与同步方式和异步方式不同。在图 3.45 的例子中，即使增加几个浏览器，管理主数据的还是只有 1 台服务器。

MMO 架构中服务器、客户端的功能

MMO 架构的实现采用浏览器方式，而物理架构则采用纯粹的 C/S 架构。顺带一提，如前所述，用户直接使用的相当于游戏浏览软件（浏览器）的程序称为“游戏客户端”。

在 MMO 架构中，游戏逻辑全部都在服务器上实现，自己与对手的关系、自己与环境的关系，以及环境与对手的关系都是如此。客户端中并不包含用于使游戏发展下去的程序，而只包含与渲染、音效以及操作有关的处理，所以服务器和客户端的功能组织完全不同。而且对应的操作系统也不同。通常，客户端使用 Windows，服务器使用 Linux 等。

与此相对，同步方式和异步方式下，每个终端都是同等的，所以所实现的功能也是同等的。浏览器方式下，客户端和服务器在功能上的差异就相当于 Web 中客户端和服务器功能的差异。服务器数据难以篡改的程度也与 Web 相同。比如，通过限制远程 shell 的登录、防止利用 SQL 注入（SQL injection）和缓存溢出的程序非法运行，基本上可以防止直接的作弊行为。

服务器端处理 ——游戏在服务器上持续运行

在网页游戏中，就算没有客户端（指浏览器本身），游戏也会在服务器上持续运行。

网页游戏中所有的操作都是按照如下流程来进行的：所有的操作都在客户端上进行→服务器端进行处理→将处理结果显示在客户端上。在

这整个过程中，信息需要来回发送，于是在客户端和服务器端之间会进行两次通信。因此，玩家在完成一次操作到得到反馈之间会存在一段时间的间隔。

由于所有的步骤都是在服务器端上依次进行处理的，所以画面的显示不会出现不一致。但是在通信延迟过长的情况下，画面就显得不连贯，玩家的游戏体验就会变得很糟糕。

在网页游戏中，服务器端不进行画面的渲染，几乎所有的渲染都能在游戏进行时由客户端加以处理。因此，可以以同步方式和异步方式实现大量玩家同时在线。

比如，在使用 CPU 主频为 2GHz 的机器来实现游戏内容的情况下，对于以同步方式来处理的游戏，1 秒钟的 CPU 时间片会以如下方式进行分配（粗略估计）。顺带提一下，以下的分配方式不仅只针对 CPU，内存的分配也是如此。

- 50%：用于处理 3D 渲染。
- 30%：用于处理物理计算。
- 20%：用于处理游戏逻辑。
- 误差度：通信处理。

与此相对，在 2GHz 的服务器上实现网页游戏的内容时，则是按以下方式分配 CPU 时间。

- 将近 100%：用于处理游戏逻辑。
- 误差度：通信处理。

单纯地对这两者进行比较，对于网页游戏中的游戏内容，由于每个 CPU 可以处理 5 倍的游戏逻辑、使用 5 倍的内存，所以它能比同步方式处理更大的地图、更多样的敌对角色等。

3.5.3　大型多人网络游戏（MMO）

此外，还有一类话题需要讨论，就是大型多人网络游戏（MMO），有关这方面的内容及其解决方案将会在第 4 章中详细介绍，这里我们先来简要地介绍一下 MMO。

首先，不考虑任何实现上的问题，如果要将游戏进行时的状态变化传输给所有的玩家，通信次数将会达到玩家数的平方次。要尽量避免这样的情况出现，就需要系统的优化。比如，设法只发送给附近的玩家，或者只发送给登录游戏的朋友和同伴等。

另外，考虑到 MMO 服务器所花费的成本，我们需要尽可能提升传输内容的抽象程度，从而减少传输量。

以上都是些有关通信成本的问题。因此，可见，一旦今后网络通信延迟缩短，延迟所造成的混乱减小，以及服务器的维护成本和带宽成本减少，会有越来越多的游戏选择使用 MMO 服务器。在这类游戏中，因为不能作弊，所以它们在商业上有很大的优势。比如，在实行道具收费时，如果不采用 MMO 类型，玩家就可以更改数据、复制高价的游戏物品，导致物品卖不出去。

此外，MMO 型的游戏还可以防止对游戏数据的篡改，所以类似花费大量时间培养角色的游戏就可以实现，这样就可以开发出有持续性收益的游戏。

3.6　小结

本章就构成网络游戏的整体体系结构，介绍了在处理大量信息的同时又必须维持毫秒级的高速响应等只有网络游戏才必须达到的要求，此外还介绍了应对这一要求的典型物理结构（P2P、C/S）和逻辑结构（MO、MMO），从下一章开始，我们将深入探讨最常用的 C/S MMO 和 P2P MO 这两种类型的游戏。

专栏　设法改善网页游戏的画面显示间隔

在只能以网页方式开发的大型多人网络游戏中，如果设法在客户端进行开发，就能稍稍降低一些画面显示的时间间隔，本专栏将对此作一下简要介绍。

假设网络延迟为 200 毫秒，实际移动需要 500 毫秒，在网页游戏中事件的发生顺序为。

- 0 毫秒：玩家开始操作（移动到坐标 XY 处）。
- 0 毫秒：操作消息开始向服务器发送。
- 200 毫秒：消息到达服务器。
- 205 毫秒：服务器处理结束，将结果（移动开始）发送给客户端。
- 405 毫秒：“移动开始”的消息到达客户端，开始进行渲染。
- 905 毫米：移动画面渲染结束。

继续后面的处理…

在这种方式下，玩家在操作后大约 400 毫秒之后，角色才会开始移动，从我们的感觉来看，这种效果实在太差了。

只要将事件发生顺序更改为如下形式就能改善操作时的体验。

- 0 毫秒：玩家开始操作（移动到坐标 XY 处）。
- 0 毫秒：操作消息开始向服务器发送。
- 0 毫秒：客户端并不等待服务器送来的结果消息，立即开始渲染移动画面。
- 200 毫秒：消息到达服务器。
- 205 毫秒：服务器处理结束，将结果（移动开始）发送给客户端。
- 405 毫秒：“移动开始”的消息到达客户端，如果移动成功则忽略该消息 *。500 毫秒：移动画面渲染结束。

继续后面的处理…

这样就将完成移动所需的 905 毫秒缩短到了 500 毫秒。由此，玩家就能感觉到自己的角色在很流畅地移动。

这里的问题是，在上文的 * 处，如果从服务器端发回了“移动失败”的消息，就需要立即回到原本的位置，否则游戏状态会不一致，导致后续操作（比如遇敌进入战斗）出现问题。

因此，根据游戏内容，如果是在静态地形以外不会出现移动失败的情况下，这种方法是可行的，但是如果遇到敌人或者其他角色等正在移动的物体，而需要进行碰撞检测时，这种方法就未必有效了。

那么，对于“自己以外”的其他角色呢？人们的感觉是很奇怪的，其他角色就算移动得很慢、或者很不流畅，又或者突然返回到原位，玩家也不会感觉到不协调。

第 4 章 [实践]C/S MMO游戏开发：长期运行的游戏服务器

本章将详细讨论作为网络游戏典型类型之一的 C/S MMO 游戏的开发技术，C/S MMO 游戏在物理上采用 C/S 架构（客户端 / 服务器架构），逻辑上采用 MMO 架构。本章将按照以下顺序进行解说。

- 网络游戏开发的基本流程
- C/S MMO 游戏的发展趋势和策略
- 策划文档和 5 种设计文档——从虚构的游戏 *K Online* 的开发中学习
- 1系统基本结构图
- 2.进程关系图
- 3.带宽 / 设备资源估算文档
- 4.协议定义文档
- 5.数据库设计图
- 服务器 / 客户端软件和中间件
- 程序编写的基本原则
- C/S MMO 游戏 *K Online* 的实现——开始编程！

在接下来的章节中，我们将对 C/S MMO 游戏和 P2P MO 游戏所涉及的技术进行详细讨论，本章针对 C/S MMO 游戏，第 5 章针对 P2P MO 游戏。这两类游戏在开发技术上有很大的不同，但是从策划文档的制订开始，一直到正式进入商业运营，整个开发流程还是有很多共通之处的。因此，本章将以 C/S MMO 游戏开发为例，介绍一下开发人员在开发任何类型的网络游戏时都需要遵循的工作流程。第 5 章则对有别于 C/S MMO 的 P2P MO（+C/S MO）架构进行说明。

此外，要阅读本章和第 5 章所讨论的游戏编程和网络编程，读者需要先了解一些基础知识。本章中所出现的一些基础知识和一部分专业术语都在第 0 章、基本术语专栏（前言）中进行了总结，如有需要请参阅。

4.1 网络游戏开发的基本流程

C/S MMO、P2P MO、C/S MO 以及其他形式的网络游戏都是以团队为单位进行开发的。最近小规模团队投入网络游戏开发的现象也不断增加，尽管如此，通常必须有 2～3 名以上的程序员参与开发。

既然要进行团队工作，那就必须建立一些用于保证工作内容一致的文档。如果团队中有缺乏经验的成员，那这一点就更为重要了。根据开发进度依次查阅所需的各种文档，可以在某种程度上把握整个开发流程。

4.1.1 项目文档 / 交付物

我们首先来看一下团队所要提交的项目文档 / 交付物（与项目有关的各种文档），参见表 4.1。表 4.1 中所示的文档是根据项目进度的先后顺序列出的。

其中，策划文档主要由项目总监和策划人员负责制定，而素材则由美术工程师负责。在检验渲染能力等必要情况下，程序员也要一起参与进来。设计文档主要由程序员负责制定。商业相关的文档则主要由项目总监和客户公司的负责人和经营者（小公司的话就是公司负责人）一起制定。这是因为运维也对公司内部的人事体制等有所影响。

如果在实际开始编写代码之前就能准备好表 4.1 中所列的文档，就能大幅降低开发的失败概率。当然，如果一个团队都是由经验丰富的人员组成的，那或许可以不受文档的限制，凭借感觉快速推进项目的初期开发。但是在游戏完成之前的几个月里，必须要向实际进行服务器管理的公司和发行公司（进行游戏发售、宣传的公司）进行说明，所以最后还是要制定这些文档。尤其是设计文档，之后补写是很困难的，即使早期的文档不够完善，但有了一定程度的原型之后，就能在进行开发的同时，逐步推进运营体制的制定。

表 4.1 项目文档 / 交付物

类别		

名称	目的	内容
策划文档		
概要设计文档	用于让出资开发费用的一方判断可行性以及用于原型开发	包括游戏内容、游戏的基本规则、游戏中的主要元素以及它们之间的关系、平台和商业模式方面的概述。在这个阶段中，数据总量尚不明确，所以用于设计工作方面的信息还有所欠缺。很多情况下都采用PowerPoint（.ppt）和Word（.doc）等可以直接进行设计的形式，页数较少
详细设计文档（也叫规格说明书）	用于让开发人员进行设计工作	此文档所包含的信息与游戏攻略、游戏说明中的内容相同。以此为基础可以决定以怎样的方式编写代码、如何开始进行设计工作。由于信息量庞大，需要跨多个Excel（.xls）文件，或者利用Wiki，直接编写HTML，制定大量的 doc 文档，准备各种专用的模板等，采用各种方式来处理大量的信息。在小型游戏中，xls 表单通常在数十张左右，而大型游戏甚至有达到数百～数万程度的情况
开发要素/工作量列表	用于估算开发成本、制定准确度较高的计划	罗列出所有必须实现的功能；确定没有遗漏的状态；为了让实际从事开发工作的程序员实现各项功能而将所有必要的工作量进行数值化。基本上归纳在一个 100～1000 行左右的 xls 文件中。与 Scrum 中的 Product Backlog [1] 类似，是进度管理的基础文档
设计文档		
系统基本结构图	说明服务器设计的概要	为了进行进程关系图和资源的设计，将所需的设计方针可视化。根据这个设计方针，就能对策划层面上的一致性加以确认。比如，是优先考虑实时性，还是优先考虑数据的正确性
进程关系图	用于提高设计的正确性，以及用于筹备服务器、构建系统	网罗了构成游戏服务器系统的所有进程（包括客户端）的任务，以及数量、关联性、网络连接应有的状态、连接顺序、访问模式、扩展方式等。使用Jude 、Microsoft Visio、ppt 和xls、Apple Keynote 等具有作图功能的工具来制作。因为通览所有进程之间的关系是非常重要的，所以需要将其容纳在一张 A3 纸内。而在网络游戏的情况下，由于 1 个稍大的白板还不能完全容纳下整个关系图，所以也有人采用画在墙壁上进行适当更新的方式
服务器物理结构概要图	用于将如何配置服务器告知系统运维公司，以及用于详细设计	以图形方式来说明具有某种特性的服务器必须以怎么样的物理关系来配置。并不需要立刻就能开始精确地进行采购和配置，只要让承担这一工作的系统运维团队或者专门公司的团队能够开始着手详细设计就可以了。在使用云服务的情况下，为了进行安全性和 VM 设置等方面的设计，也需要用到此文档
带宽/设备资源估算文档	用于估算所需筹备的服务器、传输线路、存储设备等的成本，以及用于验证商业	从技术层面上估算需要具备那些类型的服务器、各自需要多少台、可用性需要达到何种程度等。信息量并不是很庞大，所以 1 个 100 行左右的 xls 表单就足够了。这也称为服务器资源估算文档

	策划的正确性	
协议定义文档	用于定义传输内容、以及提高服务器和客户端并行开发的效率	记述了需要在客户端和服务器之间进行传输的数据包的内容。包括数据包的格式及其所包含的值等，对执行远程调用的 API 进行定义。这样，就能让多名开发人员同时进行服务器和客户端的开发
数据库设计图	用于实现数据库	为了避免在运营开始之后突然出现数据库性能问题，需要事先对数据库的查询语句进行规划（以怎样的方式执行怎样的查询语句）并将其文档化，从而防患于未然。最理想的方式就是在开发之前为一些重要的表设计 ER 图。为此需要一些设计工具
系统运维设计文档	用于实际构建物理的服务器系统	服务器需要具备怎样的性能，应在何时购买，购买几台，如何进行配置、连接、设置，何时进行交换，等等。为了进行这些物理环境的搭建，必须提供一些必要的确认文档
商业相关文档		
合同	用于防范投资方、开发方、运营方等在交易方面出现纠纷	包括报酬、初期开发费用 / 最低保障金额 / 成功报酬的条件 / 产品的所有权和专利的处理 / 机密的保守 / 问题担保等一些重要的交易方面的条件。通常使用 doc 文件
商业计划书	用于判断投资计划的可行性	根据过去的经验来预估初期开发，乃至运营开始后 2～3 年之内的动向。以其他类似的或者互相竞争的产品所获得的成绩为依据。通常使用 1 个表单的 xls 文件
开发日程	用于把握整体的开发进度	根据开发要素列表和工作量列表，制定开发日程、设置适当的里程碑，以此来防止开发周期的延长。至少每月更新一次。通常使用 100～1000 行左右的 xls 文件，应将其合并在一个表单中，以便于投资者进行判断。还需要将发售之后 2～3 次的更新计划也包含在内
开发体制图	用于确认开发流程是否正常	包含了项目总监、程序员、设计人员的开发体制图。在网络游戏的开发中，运营公司、系统管理公司等多个公司之间必须进行协调，尤其是对公司外部的人员说明己方的开发体制是非常重要的。最好绘制在 1 页 xls 文件内

[1] Product Backlog 是整个项目中所需要素的概述。Scrum 是敏捷软件开发方法的一种。

同时进行开发之前的准备和初期实现

聚集起所有必要的程序员需要几个月的时间。所以，绝大多数的项目都会在准备以上这些文档的同时，实际进行一些程序开发工作。如果团队中存在具有一定经验的程序员，那么可以在制定概要设计文档的阶段

中，探讨一下需要进行怎样的验证工作、制作怎样的原型更为有效，等等。

在项目的初始阶段，需要像这样同时进行开发之前的准备工作和初期实现。

4.1.2　开发的进行和文档准备的流程

开发的整体进度以及在何时需要文档的大致流程如下。为各项工作设置里程碑，在无法判断是否应该继续进行下去的情况下，应该中止开发或者改变方针。

❶ 准备概要设计文档、商业计划书。

❷ 对概要设计文档、商业计划书进行评估，如果没有问题则继续下一步。

❸ 准备详细设计文档、各种设计文档、开发要素列表、工作量列表、开发日程等所需的文档，开发原型。

❹ 以原型为基础，使游戏始终保持可玩状态，同时把握服务器的性能指标，然后进一步进行详细开发，编写程序、制作数据等。不断更新工作量列表、任务列表、开发体制图、开发要素列表。

❺ 程序和数据的形式大致确立后，开发用于管理 / 运营的工具。

❻ 在完成前的半年左右决定服务器的筹备方式，更新估算好了的资源计划书，将这些信息交给系统管理 / 游戏运营公司，开始进行服务器的筹备和运营体制的构建。

❼ 实现收费系统。

❽ 进行内部 α 测试（不包含玩家的多人测试）。

❾ 进行封测 β 测试（限制玩家人数的多人测试）。需要提交面向运营团队的测试说明书。

❿ 进行公测 β 测试（不限制玩家人数的多人测试）。

⓫ 开始收费。

⓬ 在半年内进行第一次更新（需要不断重复此过程）。

4.1.3　技术人员的文档 / 交付物

接着，我们来归纳一下技术人员的文档 / 交付物。为了实现网络游戏的服务，技术人员必须准备的文档 / 交付物如表 4.2 所示。

表 4.2　技术人员准备的文档 / 交付物

类别	
名称	内容
程序	
服务器端程序	C/S MMO、P2P MO、C/S MO 架构需要一些服务器。虽然在 P2P MO 中，也有一些完全不需要开发服务器的情况，但这很少。服务器端程序具有各种作用，接下来将会介绍
客户端程序	渲染、音频、操作输入等玩家直接接触到的部分
开发工具	网络游戏需要制作大量的数据，而且因为在客户端上使用的部分与在服务器上使用的部分存在一定的差异，所以需要用一些管理工具来保证这些数据的一致性
单元测试工具	以容易成为瓶颈的部分（数据库访问、消息服务器、验证服务器等）为中心执行自动测试
测试 bot	为了进行给服务器和客户端带来高负荷的测试，使用专门制作的程序。基准 bot
编程相关文档	使用 UML 来绘制类图、时序图等，用于交接、发现改进之处
数据	
游戏设定数据	这一部分的数据量非常庞大。除了用 xls 文件、文本文件来保存，有时还要作为数据库来进行动态管理。
图像 / 视频 / 音频数据	数据量庞大。为了避免在之后进行版本更新时重新发布所有的文件，需要设法将其打包。此外，数据量变得非常庞大的情况下，还需要用到数据结构管理工具
测试	
测试	网络游戏的测试不能仅由开发团队来进行，还需要运营公司等其他团队来进

操作手册	行。该文档不仅要包括游戏的操作顺序，还应包括测试服务器的启动顺序、安全性的设置、调试命令的说明等各种必要的信息
性能测试结果	在什么样的服务器环境 / 网络设置下，对哪个进程造成了怎样的负荷？对这样的结果以及这种情况下的问题和解决方案进行总结。作为构建服务器 / 基础设施时的基本文档，在团队外部也会被用到
管理 / 运营	
服务器管理工具	为了扩展、启动、停止和修复多种类型的服务器，必须准备一些专用的软件。如果仅仅使用针对Nagios（稍后介绍）等通用工具的插件还不够，那就需要开发专用的客户端软件
日志管理工具	特别是在 MMO 游戏的运营中，常常需要动态地、大量地搜索游戏服务器的运行日志。为此有时需要开发专用的日志解析工具。在 MO 游戏中，经常需要根据日志解析来改变等级排名和匹配设定
运维操作手册	用于帮助运营公司的工作人员正确地进行服务器的维护。最好是开发用于自动进行维护的工具，但是事实上要实现完全的自动化是非常困难的，所以必须准备一些易于阅读的文档。同样，要让之后加入团队的技术人员准备这些文档是很困难的，所以最好是由在开发初期就参与设计的开发人员来负责编写

＊ ＊ ＊

本节总结了网络游戏开发所需的文档和交付物。此外，根据开发规模及其内容以及开发团队的差异，有时还需要更多的文档。

4.2 C/S MMO 游戏的发展趋势和对策

那么现在我们来了解一下 C/S MMO 架构的游戏开发的实际内容了。

4.2.1 C/S MMO 游戏的特点

在 C/S MMO 架构中，游戏运营公司所管理的服务器总是在数据中心内持续运行着，这些服务器保存着游戏过程中的所有信息。运行在玩家 PC 机上的游戏客户端程序与 Web 浏览器一样，只是用来显示保存在服务器上的游戏信息，作为一种浏览程序而运行着。

“在数据中心被安全管理着的服务器中，存在着持续运行着的游戏服务器”，这一点是 C/S MMO 技术上的最显著的一个特点。由于这个特点，商业模式也受到了很大的影响。在第 6 章中将会详细介绍应该采用怎样的形式来进行收费以筹措运营费用，本章则从技术角度出发，着重介绍如何实现“游戏本身”的系统。

4.2.2 C/S MMO 架构（MMO 架构）特有的游戏内容

我们复习一下只有具备了持续运行着的服务器（以下称为游戏服务器），才能实现的游戏内容。

- **处理大量的数据**

  当游戏设定数据达到几十吉字节时，就无法在用户 PC 上安装。在使用如此庞大的数据量来构建游戏世界时，当然只能将这些数据存放在游戏服务器上，随时取出需要的部分来使用。像《第二人生》这种由用户来创造世界的 MMOG，保存着几十 TB 的数据。

- **向玩家严格保密设定信息**

  由于游戏的设定信息并不安装在玩家的电脑上，所以除非玩家突破了游戏服务器的安全机制进行非法访问，否则绝对无法访问这些数据。

- **严格维护游戏数据的更改内容**

  将用于游戏进行的所有操作反映在游戏过程中之前，通过游戏服务器对操作内容进行确认，屏蔽非法操作，以此确保长期存储在游戏服务器内的游戏数据始终处于正常状态。这样就可以实现一种“持续投入时间、精力”的玩法，比如让玩家花费几百小时来培养一个角色，或者花费极大的工夫创建世界。另一方面，如果没有专用服务器，怀有恶意的玩家随时都能访问到游戏数据，这样就很难实现这种类型的玩法了，所以不得不采用以瞬间分出胜负为中心的玩法。

- **简单地进行设定信息的更改**

  数据中心的游戏服务器上的硬盘和内存中所存放着的游戏设定数据可供游戏运营公司自由地访问、更改。与此相对地，在采用以软件方式向玩家发布的游戏中，运营方没有权限访问运行在玩家设备上的程序和硬盘，所以无法立刻进行更改。

- **易于结合 SNS 等其他服务系统**

  由于游戏服务器始终在数据中心内运行着，所以在与 SNS 等服务系统协作时，随时可以从 SNS 的 Web 服务进行 API 调用。

C/S MMO 架构的限制

另一方面，在 C/S MMO 架构中，由于要维持游戏服务器，不得不作出一些牺牲，例如如下这些方面。

- 延迟较大

  所有的操作都要发送至游戏服务器，由服务器进行验证后再发送给其他玩家，所以仅仅是这一部分操作，就会使响应降低。

- 游戏服务器的带宽负荷很高

  所有的操作都要发送至游戏服务器，所以服务器的带宽负荷自然就变得很高，成为运营成本上升的主要原因。

- 游戏服务器的维护费用很高

  因为要验证所有的操作，又要存放在服务器的内存中，所以在游戏的运营过程中，服务器的维护费用不断增加。

- 服务器停止期间，无法进行游戏

  由于在玩家 PC 上没有安装软件和游戏的设定数据，所以一旦停止了游戏服务器，玩家就无法进行游戏了。

* * *

总而言之，C/S MMO 架构的游戏的特点就是能够“大规模、持续地进行游戏”，但是另一方面，也因此而“必须付出不少代价”。

4.3 策划文档和 5 种设计文档——从虚构游戏 *K Online* 的开发中学习

从本节开始，我们将以一个虚构的示例游戏为题材，着手进行 C/S MMO 游戏的开发。首先我们从游戏题材和设计准备开始。

4.3.1 考虑示例游戏的题材

几乎所有的游戏编程入门书籍都不会采用概论性的或者实际的商业类型的示例，而只是提炼出其中本质性的部分，以此来设计示例游戏并进行开发。本书为了探讨典型的模式，同样采取这种方式，采用虚构的游戏策划。最理想的做法是根据游戏的各种类型来设计多个示例游戏，但在本书中，本章的 C/S MMO 游戏和下一章的 P2P MO 游戏都分别列举一种类型加以说明。

但是游戏的策划工作超出了本书的范畴，所以这里尽可能参考已经存在的游戏来设计这个示例游戏的内容，然后再来说明制作该游戏需要哪些技术。

我们通过以下这些条件来寻找作为参照的游戏。

- **本书的读者可以免费试玩的游戏**
- **本书的读者可以免费查阅到 Wiki 等设定文档的游戏**
- **在商业上获得成功的游戏**
- **在今后几年里，服务能够维持下去的游戏**
- **笔者玩过的游戏**
- **使用典型方法实现的、较少使用特殊技术的游戏**
- **规模适中的游戏**
- **在版权方面没什么问题的游戏（笔者不涉及开发的游戏）**

通过上述条件来寻找参照游戏，不需要完全相同，只要取出本质性的部分进行技术说明，那基本上就能略过策划工作方面的说明了。

查找的结果显示，以英国 Jagex 公司运营的网页游戏 *Runescape* 作为基础最为合适。*Runescape* 的游戏内容在 Wiki 上的介绍非常详细，策划文档中的详细设计文档基本上都可以在 Wiki 页面上浏览到。

Runescape 是一款使用 Java Applet 开发的面向 Web 浏览器的 MMOG 游戏。很多其他面向 PC 的 MMOG （比如 *WoW* 、*LEGO Universe* 等）都是作为本地应用安装的，但是最近也有很多与浏览器相关的平台（比如

Flash、Adobe AIR、Unity[2] 等）用在了 MMO 游戏中，所以这里作为示例，采用运行在 Web 浏览器上的 MMO 游戏也没有什么问题。

> [2] 在 Web 和智能手机、各种游戏控制台等平台上制作具有交互性的 3D 内容的开发环境。http://unity3d.com/unity

- **参考信息**
  - 实际试玩 *Runescape* → http://www.runescape.com/
  - Web 上提供的规范文档汇总→ http://runescape.wikia.com/wiki/RuneScape_Wiki

首先，我们将本书中的 C/S MMO 示例游戏暂时称为 *K Online* 。*K Online* 与 *Runescape* 一样，可以让玩家探索极具多样性的古代世界，几千名玩家可以协同作战，游戏具有丰富的任务、魔法、技能、boss 战等富有趣味的元素。基本上可以免费进行游戏，通过每月支付 5 欧元，可以体验到两倍以上的游戏内容。

此外，*Runescape* 是一款使用 Java Applet 开发的网页游戏，而 *K Online* 则主要针对机器性能相对低于欧洲市场的亚洲城市，所以采用 Windows 本地应用的方式来实现。

以上所述的各要素就是应该写在概要设计文档中的内容。

4.3.2　详细设计文档

决定题材的概要设计文档完成后就要准备详细设计文档了。*K Online* 计划参考 *Runescape* 的文档。为了想象一下详细设计文档中的必需信息，我们来看一下 *Runescape* 的 Wikia 分类页面 [3] 。*Runescape* 的游戏内容由以下要素组成。

> [3] Wikia 页面可以通过以下网址访问：http://runescape.wikia.com/wiki/Category:Content

成绩的记录、敌人、聊天、公会 [4] 、战斗、日期、与 NPC 的对话、消遣、地理、拍卖行、高分排行、互动场景、持有物、物品、经济、原料、结构、会员（付费玩家）、小游戏、小任务、音乐、玩家之间的交战、玩家所有的家园、任务、随机事件、规则、限制、技能、装备

> [4] Clan。角色团体。

只有这些的话还是不能开始着手编程。因为这些要素列表还称不上是详细文档。那么我们来挖掘一下作为游戏要素中最重要的技能吧 [5] 。技能分为以下这些种类。

> [5] 可以在以下的页面上查阅：http://runescape.wikia.com/wiki/Skills

攻击、力量、防御、飞行工具、祈祷、魔法、魔符石制造、伤害抗性、手工艺、采矿、锻造、垂钓、烹饪、生火、伐木、敏捷性、草药学、药物、偷窃、弓箭制造、屠杀、农业、建筑、手工、狩猎、召唤

虽然列出了详细列表，但还是无法进行编程。我们进一步将“伐木技能”进行细化。Wikia 说明中提到的关键字如下所示。

树、斧头、原木、世界、持有物列表、火、火盆、燃烧、小刀、弓箭制造技术、弓、箭、青铜斧、铁斧、钢斧、黑铁斧、秘银斧、合金斧、魔符石斧、粘土斧、龙斧、地狱（Inferno）扁斧 [6] 、付费会员、魔法树、普通树、75 等级、Normal（普通木材）、Achey、Oak（橡木）、Willow（柳木）、Teak（柚木）、Maple（枫木）、Hollow（空心木材）、Mahogany（红木）、Arctic pine（北极松）、Eucalyptus（桉木）、Yew（紫衫木）、Ivy（常春藤）、Magic（魔法木材）、Cursed Magic（受魔法诅咒的木材）[7] 、独木舟制造技术、Log、Dugout、Stable dugout、Waka[8] 、造船、河流、水上移动

> [6] 青铜斧 ~ 地狱扁斧都是斧头的种类。

> [7] Normal~Cursed Magic 都是木材的种类。

> [8] Log、Dugout、Stable dugout 和 Waka 都是独木舟的种类。

“树木”看起来很重要，所以我们进一步从 Wikia 的链接 中选出“普通树木的原木”这一关键字。于是，终于得到了以下数字结果。

shortbow（短弓）、crossbow（弩）、longbow（长弓）[9] 、等级 5、9、10、热气球、经验值、每次 25、每次 40、每次 5、15 支箭、每次 10、每次 6

> [9] shortbow~longbow 是弓的种类。

至此已经无法再进一步挖掘了。以上我们逐步挖掘了游戏整体的要素→技能→伐木→普通树木的原木。只是挖掘这极小的一部分，就能得到超过 100 个关键字。

详细设计文档的必要性

对于上述所有的要素，“全部的设定要素有多少？”、进而“各种要素之间存在怎样的关系？”、乃至“某个数值与另一个数值全部被定义”，拥有这种程度的信息量的文档就是详细设计文档。

了解总共有多少元素就意味着需要与攻略书具有同等程度的信息量。当然，根据开发人员的熟练程度，抱着“代码就是说明书！”这样的想法或许也能进行下去，但是在商业 C/S MMO 游戏中，采用这种方法最多只能开发出原型而非最终产品 [10] 。

> [10] 所有成员都拥有丰富的经验的团队是很少见的，因为通常游戏的规模总是超过人的记忆。

4.3.3 MMOG 庞大的游戏设定

由于游戏程序中的很多行为都是无规则的，所以在实际的开发项目中，无法在开始编程之前就完全将数值之间的关系设想清楚。所以要在一开始进行原型的开发，集中处理游戏逻辑（请参考专栏部分）中看起来最难的那些部分，然后编写验证程序等。

在 *Runescape* 中，光物品就高达数千以上，生物达到了数百以上，大量的设定数据遍布于网络。*Runescape* 的 Web 网站声称该游戏“能玩上几万小时”，其依据就是具有“庞大的游戏设定”。物品和技能并非只是存在在那里而已，不仅是通过任务和事件等各种各样的方法使用它们，安排着这些任务的世界也非常庞大，光是探索这个世界就要几千个小时。在 *Runescape* 的世界中，玩家可以在长达几万小时的时间里参与各种活动，同时自由地生活其中。

在 MMOG 的世界中，数量如此庞大的上千种物品和技能遍布整个网络，从而可以应对玩家所抱有的“试试看怎么样。”这一期待，这正是虚拟空间。

看一下详细设计文档的概要，估计大家就能感觉到 MMOG“庞大的游戏设定”所具有的规模了。当然，在技术上，要在保证一致性的前提下实现上述这样具有庞大功能的系统，并且在几年内持续进行修改，必须付出

很大的努力，准备相关的文档、设法降低程序和数据之间的耦合度、开发用于测试数据一致性的工具以及用于高效制作大量数据的工具，等等。

4.3.4　5 种设计文档

至此，我们已经了解了 *Runescape* 的详细设计资料。那么，如何来设计整体的系统呢？作为设计文档的交付物，我们试着制定如下 5 种文档。

1. **系统的基本结构图**

2. **进程关系图**

3. **带宽 / 设备资源估算文档**

4. **协议定义文档**

5. **数据库设计图**

4.3.5　设计上的重要判断

首先，在制定文档之前，必须要考虑一下设计的大方针。

在 *Runescape* 游戏中，设计上的重要判断有如下两个。

- **一概不采用实时性高的策划内容。**

- **并行启动多个具有相同内容的游戏世界，不与朋友在同一个世界游戏也没有关系（平行世界方式[11]）。**

> [11] “平行世界方式”是用来扩展 C/S MMO 服务器几大基本方式中的一种。我们将在后面的“系统基本结构图”一节中介绍。此外，平行世界方式将玩家的交流切断了，所以为了保证游戏的可玩性，在策划上应尽量避免这种情况，相关内容也将在后面介绍。

通过这样的决策，服务器系统就能够得以简化。另一方面，从如何实现游戏可玩性的观点来看，我们不得不放弃那些动作性强的游戏内容，而且之后参与游戏的玩家不能与以前就已经参与游戏的玩家在同一个服务器上注册，这是策划上的不足之处。但是接受这些不足可以一下子提高技术上的实现程度。

在 4.2 节中介绍了 MMOG 的限制，在 *Runescape* 中，如何来判断这些限制呢？

- **延迟较大**

  Runescape 以游戏内容的探索、经济活动、与敌方怪物战斗为中心，不需要在 16 毫秒内进行操作，即使是 200 毫秒～500 毫秒的延迟也完全没有问题，其游戏内容完全可以在这样的延迟下体验。

- **游戏服务器的带宽负荷很高**

  Runescape 对游戏内容进行了限制，不需要任何激烈的操作。基本上来讲，几秒内只需要进行 1 次鼠标点击，操作频率较低。此外，也不需要频繁发送其他玩家的动作方面的数据，平均下来不到 10kbit/s。

- **游戏服务器的维护费用很高**

  不仅操作频率较低，敌人的行为也调整得较慢。因此，服务器的逻辑负荷非常小。

- **服务器停止期间，无法进行游戏**

  因为同时启动了多个具有相同内容的服务器（平行世界），因此可以避免所有的玩家都无法进行游戏的情况。在 *Runescape* 中，这并非是刻意实现的。

不需要高速响应、采用平行世界，通过这两项基本决策，基本上就可以避开之前所述的 MMO 架构的主要问题了。如果能像这样来制定设计的基本方针，接下来就可以进入设计文档的制定阶段了。

这里的关键是，将游戏的策划内容特定化为 MMO 结构，以此来避免技术上的问题。可以说这是游戏策划人员和技术人员有效协商沟通，并且予以实现的结果。一般来讲，如果策划人员和技术人员协商失败，就会在游戏开发的后期阶段出现无法修改的架构上的问题，这种情况很多。

下面我们开始制定各个文档，展开设计工作。

4.4　系统基本结构图的制定

首先，我们从系统基本结构图出发来制定设计文档。准备妥当后就可以对一些必要的数字进行估算。

4.4.1　系统基本结构图的基础

系统基本结构图的目的就是明确设计方针。只要能确认程序内容符合商业游戏*K Online* 的策划内容和商业模式就可以了。其顺序如下所示。

❶ 确认期望的同时连接数，以及能否免费进行游戏等商业模式。

❷ 确认预想的瓶颈内容，并且选择用来避免瓶颈的扩展方式。

4.4.2　服务器必须具有可扩展性——商业模式的确认

首先要确认商业模式。在 *K Online* 中，假设有几万名玩家，其中有一小部分是付费会员，每月支付 5 欧元，这样的假设是很有必要的。在这种情况下无法估算最大玩家数。

玩家数多到无法估算的主要原因有。

- **可以免费试玩**
- **游戏大受欢迎**

游戏能否大受欢迎事前是无法得知的，特别是无法预知游戏“什么时候”会火起来。哪怕服务器设备只多出来一天，这些多出来的部分也会产生成本。所以在最初发布时应将服务器设备限制在最低程度，而之后，即使访问量增加，也必须能够在不改变程序和设计的情况下，通过简单地增加服务器来加强游戏对大量玩家的支持。实现这一目标的工作被称为“使服务器具有可扩展性（可以扩大规模）”。

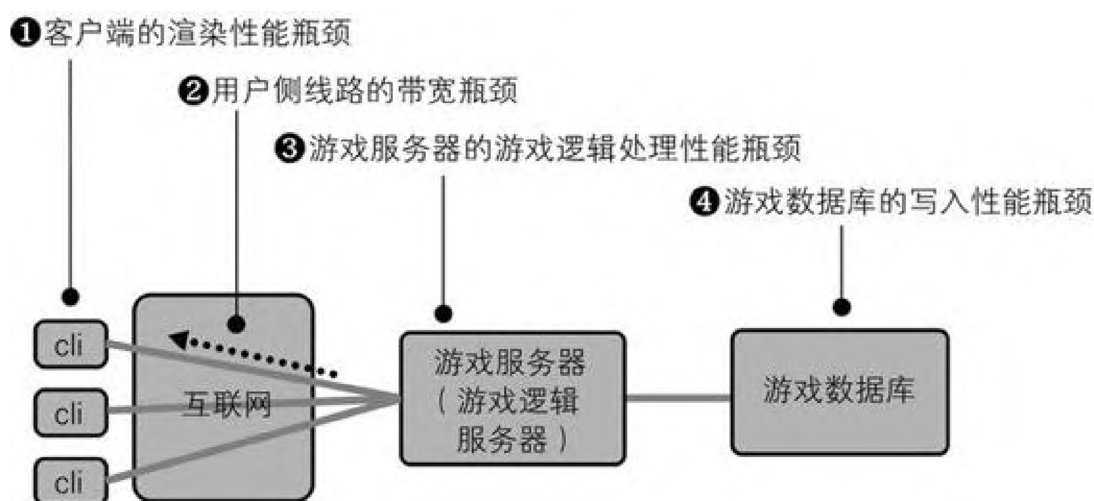
启动少量的服务器，根据需要予以增加，这并不是一个很难的问题。在没有特别限制的情况下，服务器的配备少则数日，多则两周左右就能准备妥当了，而现在有些地区也能使用云服务。但是，程序必须事先在可扩展性方面做好充分的准备。

4.4.3　各种瓶颈——扩展方式的选择

接下来，我们来确认一下瓶颈，了解一下用于消除 C/S MMO 架构的服务器系统的瓶颈而采用的扩展方式。

虽说要扩展服务器，但是有必要进行加强的部分是有限的，在整个系统中可扩展性并不一定会成为问题。在某些经常产生瓶颈的地方，如何采取扩展方式则是个问题。存在会产生瓶颈的地方，也就意味着构成服务器系统的某些进程的处理性能有所不足。进程设计将在后面详细介绍，首先我们来讨论一下 C/S MMO 中经常产生瓶颈的地方。图 4.1 对此进行了总结。

图 4.1　C/S MMO 系统和产生瓶颈的地方



该图中出现的一些术语的意思如下所示。

- **cli：客户端上的终端程序。**
- **游戏服务器（游戏逻辑服务器）：在内存中实际存放游戏设定数据、并在其中进行高速处理的主服务器。**
- **游戏数据库（DB）：搭载了用于保存游戏结果的存储器的服务器。**

这里最重要的是，客户端（cli）、游戏服务器和游戏数据库这 3 个部分可以构成整个系统。

专栏总结了图 4.1 ❶ 客户端的渲染性能瓶颈，请参考该部分的内容。在图 4.1 ❷ 用户侧线路的带宽瓶颈方面，除了那些要持续加载图像数据的

游戏或者要传输几吉字节的庞大的差分数据，在其他情况下都不会有问题，所以在此不作讨论。

这里要考虑的就是应该采取什么方法来解决剩下的两种瓶颈：图 4.1 ❸ 游戏服务器和 ❹ 游戏数据库的瓶颈。通常，这几种方法都是利用数据访问的局部性（来自于游戏特性），通过某种方式将服务器进行划分。

专栏　MMO 客户端特有的渲染性能瓶颈——游戏客户端的瓶颈

MMO 架构的游戏客户端有一种特有的瓶颈。首先，游戏客户端整体的处理性能瓶颈产生在“游戏处理”和“渲染处理”这两个地方。在 MMO 架构中，游戏的处理全部在服务器端进行，所以在 MMOG 的客户端中，只会在“渲染处理”上产生瓶颈。

通常，渲染就是从磁盘中读取模型数据和纹理数据，然后将其交给 GPU 加以处理。渲染的处理速度在不同阶段中有所不同。

- **从磁盘读取数据至主存中：非常缓慢。所需的时间以毫秒为单位。**
- **从主存读取数据至 GPU 的内存中：较慢。所需的时间以微秒为单位。**
- **GPU 内存内部的传输：非常快。完成时间以纳秒为单位。**

因此，在 GPU 内部进行处理才能达到高速渲染。

一般游戏中的渲染

在一般的游戏中，启动时所需的所有图像至少要在主存中加载，尽可能全都在 GPU 中加载，之后在游戏过程中，所有内容的读取都尽量不要访问主存和磁盘。这一点是最基本的。

比如，在《马里奥赛车》等 MO 架构的游戏中，比赛开始时，在渲染参加比赛的 8 名玩家时，所有必要的图像都已确定，也不会有中途参加游戏的情况，此外赛道的地形在比赛中也不会改变。因此，赛道和角色都能在比赛开始前就读取完毕，并存储在 GPU 的内存中。

MMO 中的渲染 ——玩家角色

但是在 MMO 游戏中，游戏过程中所需的图像是无法完全确定的。其中最典型的就是玩家角色。在本章的示例游戏 *K Online* 中，玩家具有各种不同的能力，能够同时加入游戏。此外，各种元素的组合方式几乎接近无限。如果其他玩家的形象都跟自己很相似，那就辨别不出玩家角色了，也很难判断游戏的进展，游戏体验也就因此大打折扣。为此，在 *K Online* 中，我们想要使玩家角色的形象多样化，各个玩家的个性和能力都能尽可能地通过角色反映出来。

其结果就造成了无法“在内存中存储所有可能出现的模型”这种情况。这样就必须动态地从主存或者从磁盘中加载。通常，从一个人烟稀少的地方移动到一个聚集着大量玩家的地方时，必须一口气读取大量的纹理数据。为了应对这种情况，我们可以采取以下这样的方法，尽可能避免游戏体验的恶化。

- **不要“一口气读取所有必需的数据”，而是在保持帧速率的情况下花时间一点点读取。**
- **即使是在读取纹理数据的过程中，也尽量只先进行轮廓的渲染，只对所处位置有所了解。**

MMO 的渲染瓶颈 ——敌方、友方，总共要显示多少名呢？

不管是 3D 渲染还是 2D 渲染，瓶颈都是发生在“要在 1 个画面中总共显示多少名玩家角色和敌方角色”的地方，所以在进行策划工作时，首先必须对该数量进行明确和验证。

比如图 C4.A 所示的 MMORPG *Lineage* 的战斗画面，*Lineage* 的卖点就在于庞大的玩家群体之间所进行的战斗（称为攻城战）。

图 C4.A *Lineage* 中人数庞大的战斗（攻城战）

Lineage and Lineage Eternal Life are registered trademarks of NCsoft Corporation. 1998-2011 ©NCsoft Corporation. NC Japan K.K. was granted by NCsoft Corporation the right to publish, distribute, and transmit Lineage Eternal Life in Japan. All Rights Reserved.

※ 图像提供：NC Japan（株）。http://www.ncjapan.co.jp/

官方网站：http://lineage.plaync.jp/

在 MMORPG 中，“哪种形态的玩家角色在哪个时刻初次出现在画面上”是无法进行定义的。像 *Lineage* 这样，200 名以上的敌友在混战之中，每秒都会有多人交替。如果不尽可能在画面上显示出玩家角色的职业和能力的差别，画面就会变得枯燥无味。

为此，如何减少图像数据、如何重新使用这些数据以及高效地在 VRAM（Video RAM）中存储这些数据都必须好好考虑。

4.4.4　解决游戏服务器 / 数据库的瓶颈

首先，解决游戏服务器的瓶颈主要有以下两种方法。

❶ 空间分割法（空间地理上的分割）

根据地理结构将游戏世界进行分割，分配给其他的服务器进程或者服务器设备进行处理。

❷ 实例法（游戏副本实例）

将负荷特别高的、用户集中在一起的部分独立出来，将这些部分分配给专用的服务器来处理。这些部分未必都是以游戏副本的形式来呈现，但典型情况下都是使用这种形式的，所以也称为游戏副本实例（instance dungeon）。

接着，解决数据库瓶颈的方法是。

❸ 平行世界方式使容易成为瓶颈的数据库本身并行化，以此将瓶颈分为多个。但是保存了玩家信息的数据库分成多个后，角色就无法在不同的世界之间移动，所以玩家之间的交流也被切断了。

上述 3 种方法也可以同时使用。比如在 *WoW* 中就同时使用了上述 3 种方法，从而得以支持数百万玩家参与游戏。

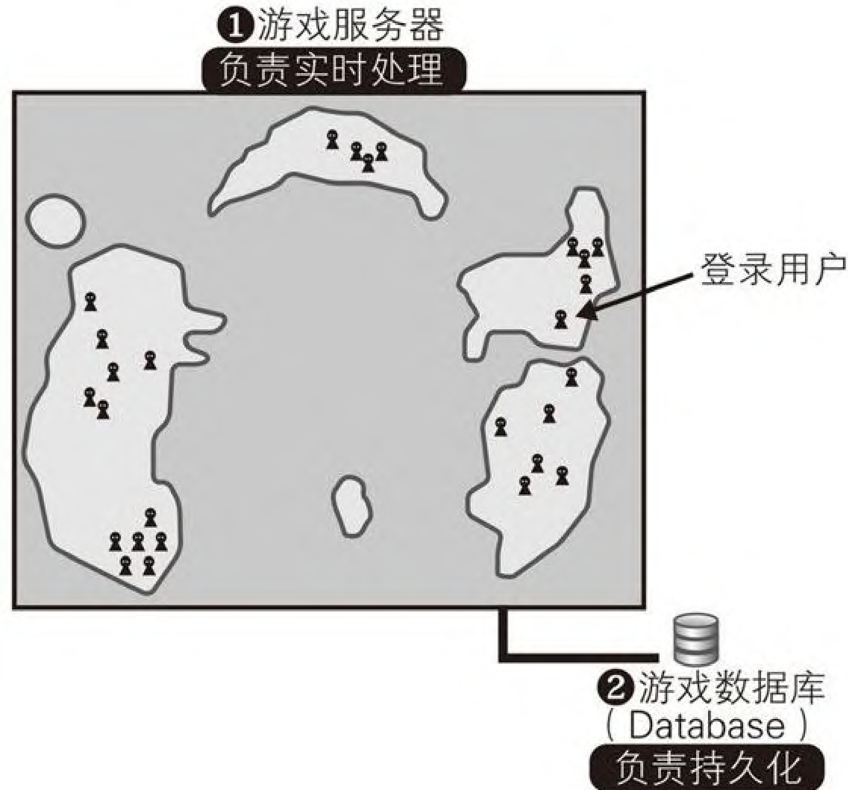
决定 C/S MMO 服务器的基本结构，也就是要判断使用上述 3 种方法中的哪种。下面我们就来详细探讨一下。

4.4.5　什么都不做的情况（1 台服务器负责整个游戏世界）

首先，什么都不做的情况是指，在 Linux 等服务器上只运行 1 个进程，数据库也只有 1 个。这里的数据库指的是 MySQL 等 DBMS 服务器实例，该实例只有 1 个。

请看一下图 4.2，登录游戏的玩家分散在游戏世界的各个地方。根据游戏中实时处理的复杂性，每 1 台服务器大约可以处理 200～2000 左右的登录数。这里存在着 10 倍的差异，这是因为根据游戏世界中四处徘徊的敌人的数量以及敌人动作算法的复杂性，服务器能处理的登录数会有很大的变化。在图 4.2 的示例中，每个玩家的游戏结果都存储在用于持久化游戏数据的数据库中。

图 4.2　1 台服务器负责整个游戏世界的情况

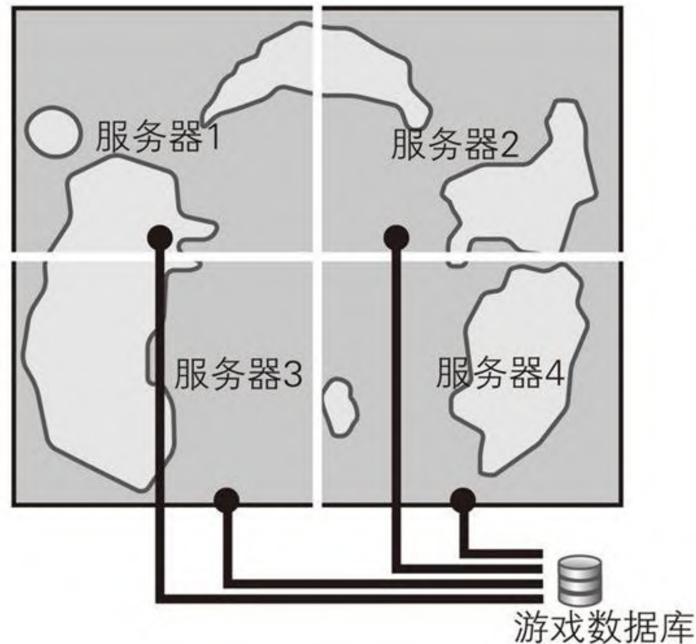


在开发游戏原型时，最多也只有 10 个左右的开发人员登录游戏，所以即使是在这种单进程的形式下也不会出现问题。但是如果就这么开始商业服务的话，游戏服务器（负责游戏实时处理的服务器）和游戏数据库肯定会成为瓶颈。

4.4.6　空间分割法——解决游戏服务器的瓶颈

图 4.3 是使用空间分割法的示例。通过使用空间分割法，实时处理这一部分的负荷就能够得以减轻。由于玩家角色在游戏世界中的移动受到了很大的空间限制，所以 MMOG 的实时处理负荷在空间上具有局部性。因此，利用这一局部性，通过将游戏世界在空间上进行分割，基本上就能线性地提高性能。图 4.3 将世界等分为上下左右 4 部分，但是为 了充分利用空间的局部性，根据各个大陆板块、各个城市来进行分割更为有效。

图 4.3　空间分割法



空间分割法对实时处理的部分进行了分割，但是它并没有分割游戏数据库。因此，在图 4.3 中，如果服务器 1～4 的总访问量增加，还是无法避免游戏数据库成为瓶颈。

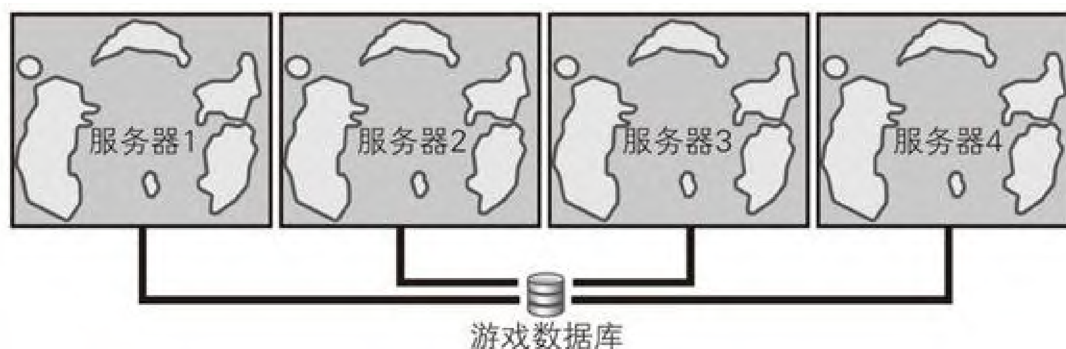
此外，即使事先对空间进行了分割，也还遗留了一个问题：由于游戏中会有一些吸引玩家的活动，所以无法排除玩家一时之间大量集中在某台服务器上的可能性。在这种情况下，玩家会感觉到无法前往某些区域，这与“无法登录游戏”一样都是最坏的情况。

空间复制

作为空间分割法的一种简化形式，有种方法是将看上去完全相同的世界复制 4 次（参见图 4.4）。这称为“空间复制法”（复制法）。决定玩家登录哪一台服务器可以有以下几种选择：❶ 明确指定，❷ 随机选择，❸ 自动选择空闲的服务器。在 C/S MMO 的情况下，比如 *K Online* ，各个服务器上有怎样的玩家正在进行游戏？物品拍卖行和用于集结成员的“广场”上有着怎样的玩家？情况各不相同，所以为了满足玩家的需求，一般需要采用方法 ❶。

在图 4.4 中，并没有将游戏世界进行分割，而是准备了 4 个完全相同的世界，这一点请加以注意。严格来讲，这不能叫做空间分割法，但是所需的技术更为简单，实现也较为简单。

图 4.4　空间复制



空间复制法有个缺点，简单来讲，就是有时会产生“玩家过少的感觉”。比如，在整个的游戏世界地图中，对某个特定的岛屿感兴趣的玩家总共占了 1 万名玩家中的 300 人。此时，如果像游戏世界的空间分割图（前面的图 4.3）那样将游戏世界进行 4 等分，在该岛上就会聚集前述的 300 人。而如果是像空间复制图（图 4.4）那样进行划分的话，每个世界就只有 1/4 的玩家聚集在一起了（也就是 75 人），大家零零散散，在互相看不到对方的状态下进行游戏。也就是说，对同一个地方感兴趣的玩家明明有 300 人，但是玩家只能实际感觉到 75 人的热闹程度。在 C/S MMO 中，这被称为“玩家过少感”。

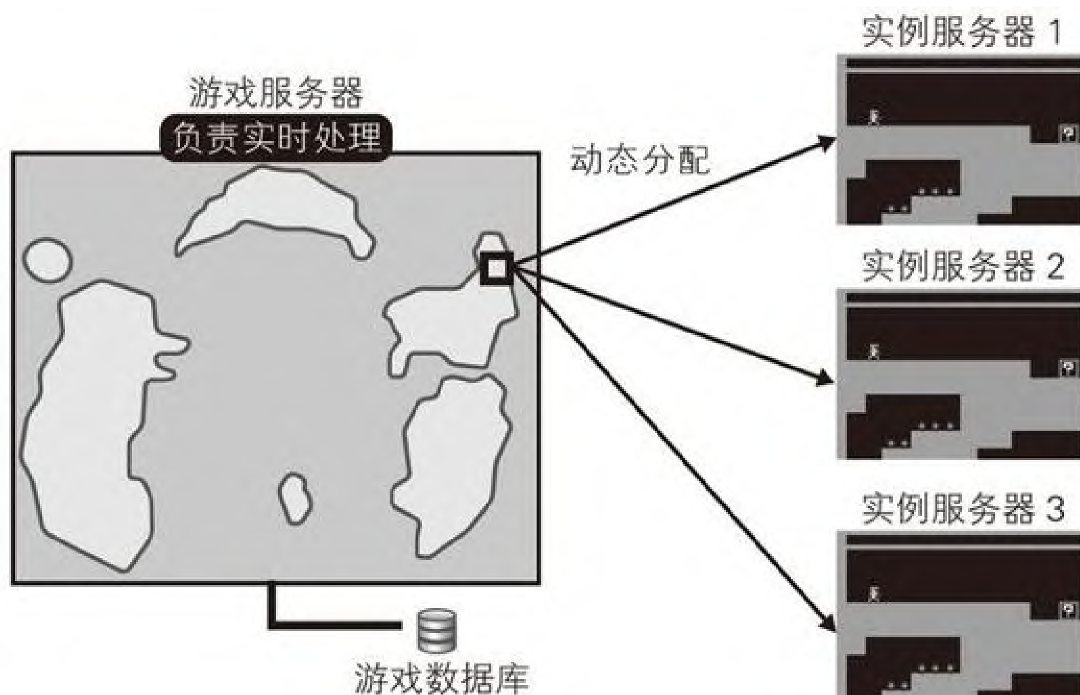
虽然存在着以上这个问题，但空间复制法仍然是一种能够通过较少的工作量来扩展游戏服务器的有效方法。

4.4.7　实例法——解决游戏服务器的瓶颈

在讲解空间分割法时，我们提到过这样一个问题：由于受到潮流等方面的影响，无法排除大量玩家聚集在某个场所的可能性。想要完美解决这个问题就要用到实例法（游戏副本实例）。

在大多数 MMOG 中，玩家只集中在整个游戏空间中极其有限的地区和场所中。在图 4.5 中，就是世界地图上正方形围起来的地方。在典型的 MMORPG 中，就是被称为“游戏副本”的地方。玩家集中在那里的理由是，在那里能高效赚取经验的怪物比其他地方更为密集，还特别设有一些珍贵的宝物，以及一些功能便利又常用的道具。

图 4.5　实例法



在游戏的设计阶段我们就已经能够设想到玩家会聚集在这样的地方了，所以可以启动多个只负责处理这一小部分内容的专用服务器（实例服务器），当玩家进入该场所时，动态选择实例服务器，然后就让玩家登录那个被选择了的实例服务器。

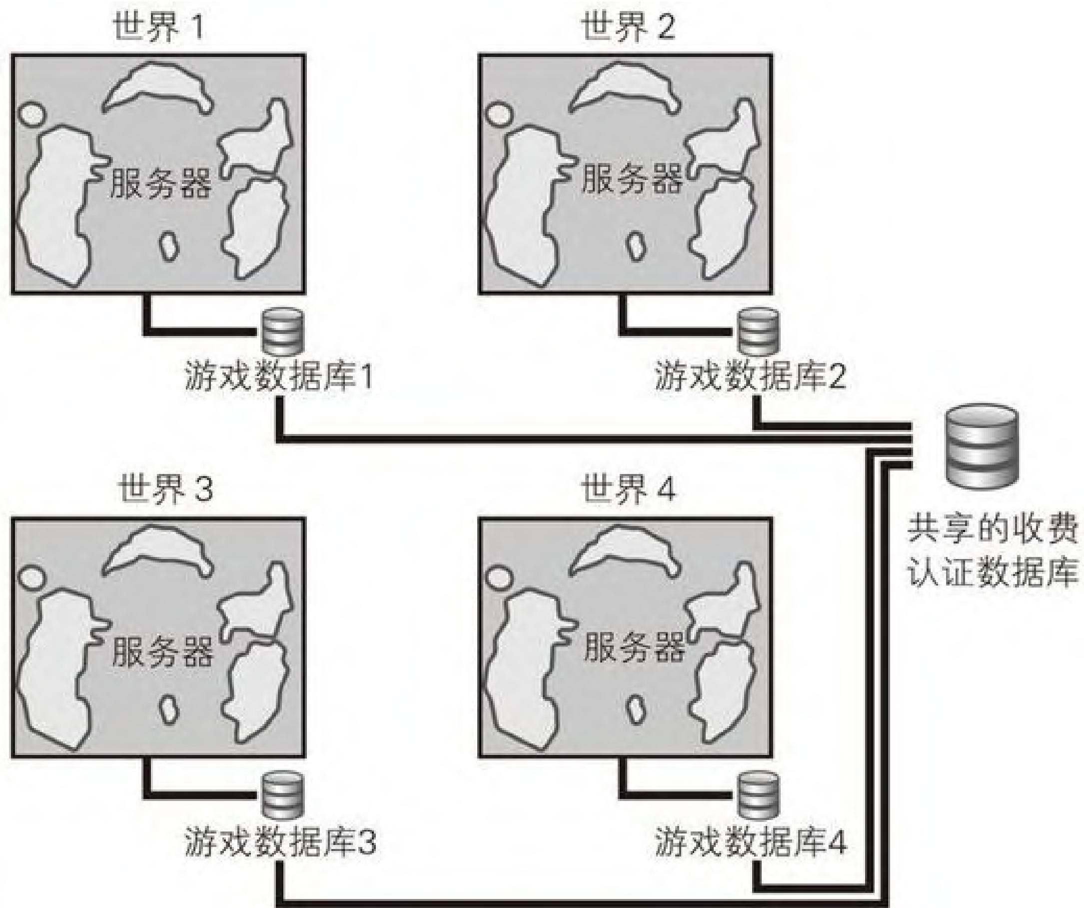
可以采用以下几种方法来确定登录哪个实例服务器：❶ 玩家自己进行选择；❷ 自动分配空闲的服务器；❸ 根据朋友的登录信息等社交信息来进行计算。

在图 4.5 中，右侧的 3 个黑色地图代表聚集了大量玩家的地下城副本，在负责进行处理的游戏处理服务器中并行地运行着这 3 个副本。当玩家角色从这一狭小的区域退出时，就会自动回到原来所在的地方（由负责整个世界的服务器处理）。

4.4.8　平行世界方式——解决数据库瓶颈

空间分割方式和实例法都是用于解决服务器瓶颈的方法。所以用来存储游戏数据的数据库只有 1 个。如前所述，平行世界方式就是用来解决数据库瓶颈的主要方法。其基本结构如图 4.6 所示。

图 4.6　平行世界方式



最容易产生瓶颈的是数据库写入操作

在 C/S MMO 中，如何高效地向数据库中写入数据是一个很重要的问题。在普通的 Web 服务访问模式中，数据库写入操作的频率还不到读取频率的 10%。但是在网络游戏中，这一频率正好相反，写入操作占了 90%。

这是因为数据的读取是由负责实时处理的游戏服务器在内存中进行管理的，可以随时从内存中读取这些数据。可以说 C/S MMO 系统是一个具有内置了处理功能强大的高速缓存服务器的 Web 系统。

C/S MMO 中的游戏信息主要是玩家角色的状态变化。在 *K Online* 中，玩家每次对物体进行操作或者使用技能时，玩家角色的状态都会一点一点地发生变化。这一频率大约是 10 秒～1 分钟一次。在理想情况下，每次发生变化时都写入数据库。同时连接数为 1000 时，如果平均每 10 秒需要存储 1 次，1 秒内就要发生 100 次写入操作。如果玩家信息很多，就需要同时、同步写入多个表中，所以 1 秒内要持续进行 100 次

的写入事务，其处理量是相当大的。因此，数据库写入操作很容易成为瓶颈。

此外，即使是在平行世界方式中，也还需要一个供整个系统使用的数据库来实现收费认证功能。对于这个数据库，当天初次登录游戏时只进行 1 次认证，实际支付时进行结算事务等操作的写入频率并没有 10 秒 1 次这么频繁，而是几十分钟还不到 1 次的程度。

平行世界方式下的数据库分割

平行世界方式是将 1 个游戏数据库分割为多个，以此来处理数据库存储性能上的瓶颈。

将数据库分为多个实际上就是增加 MySQL 的服务器实例。将数据库分为多个后，角色 ID 和名字等重要的编号体系就会发生重复，所以存储在某个数据库中的角色无法迁移到其他数据库中。为此，出于以下几个原因，这种方式被称为“平行世界方式”。

在游戏中使用的角色（等重要信息）的转移受到限制

- 角色所在的世界并不相同→世界不同
- 但是世界的呈现和游戏的内容完全相同

在图 4.6 中，1～4 这 4 个游戏世界具有完全相同的地形设定，所以世界看上去是并行存在的。在平行世界方式中，玩家的角色信息分别存储在各自的游戏数据库中。所以，世界 1 中的玩家和世界 2 中的玩家无法进行共同抗敌等多人游戏元素。正因如此，这种方式也存在着与“空间复制”（空间分割法中实际上并不进行空间分割的一种方式）相同的问题，而且可以说更为严重。

平行世界方式引起的问题

在平行世界方式下，玩家不能与其他世界中的玩家共同进行游戏，对于玩家来说，这种不便是强加在他们身上的。不仅如此，在经过了一段时间的运营之后，当玩家数有所减少时，还会产生更为强烈的“玩家过少感”。这比空间复制法所引起的问题更为深刻。

比如，游戏开始运营后的第一年里有 1 万名玩家，世界增加到 4 个之后，过了 3 年，玩家总数减少到了 3000 人，此时，如果还是 4 个世

界，每个世界中的人数就比一开始少了很多。

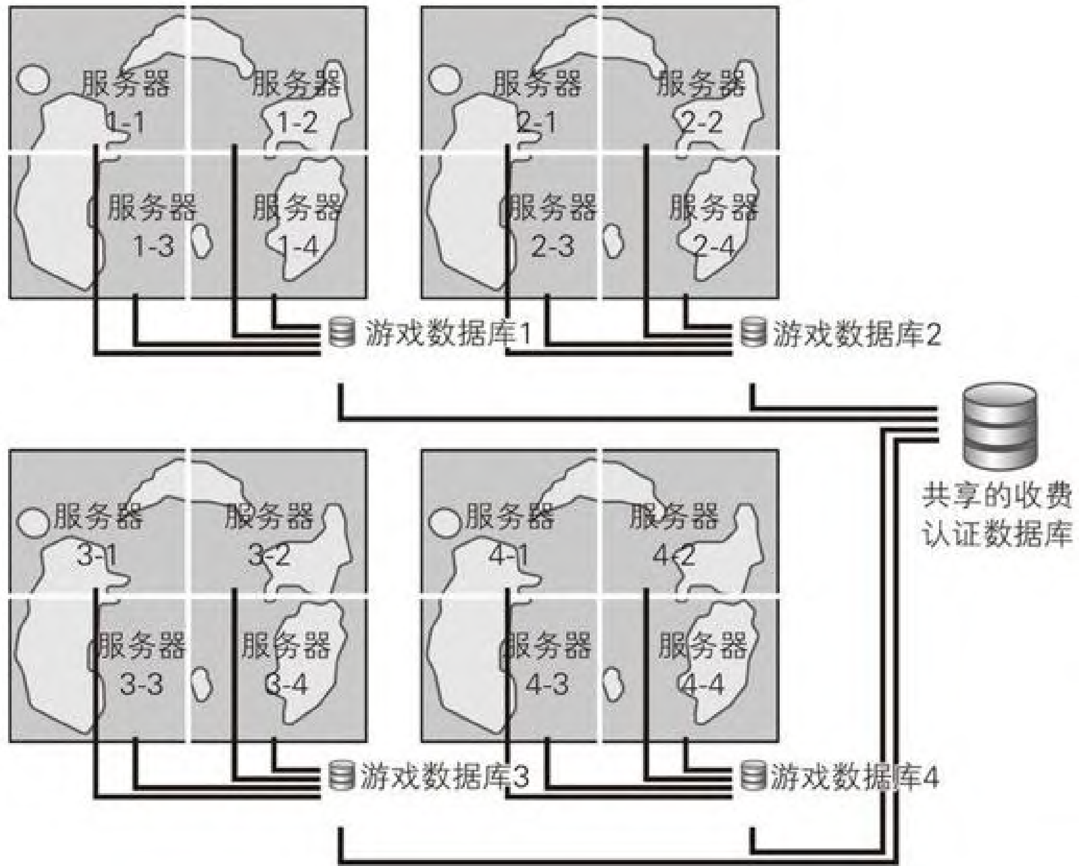
为了解决这个问题，就需要采取“世界合并”等处理方法。但是实际上并没有这么简单。很多玩家在经过了这么长的一段游戏时间之后，在各自的世界中都已经形成了自己的人脉，各自的团体和用户组都有着各自的作风和文化，所以单单因为人数减少而合并世界会给玩家团体带来更大的影响。这样一来，可能会导致大量的玩家离开游戏。

为了尽量避免这个问题，必须对玩家团体（公会和帮会等）的名字和昵称等与命名相关的信息进行检查以防止冲突。这样就要事先在数据库中加入这样的检查功能，但这么一来，越是使各个世界的数据相关，数据库的瓶颈就越容易发生，原本是为了解决瓶颈而引入的平行世界体系反而造成了相反的结果。

4.4.9　同时采用多种方法——应对越来越多的玩家

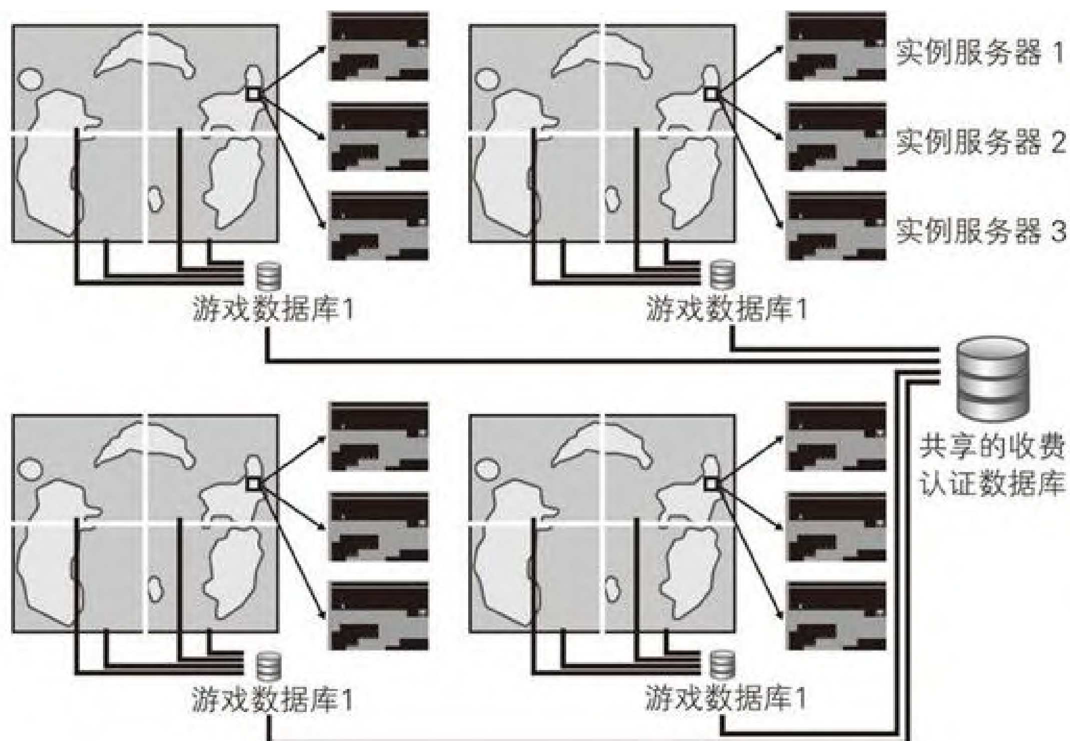
玩家进一步增加后，游戏服务器和数据库两方面都可能发生瓶颈。为了应对这一情况，可以同时采用空间分割法和平行世界方式这两种方法。这里就不作详细讨论了，这种情况下的系统结构如图 4.7 所示。

图 4.7　同时采用平行世界方式和空间分割法



至此，如果还需要进一步扩展，可以同时采用空间分割、平行世界、实例法。在此，我们也不作具体说明了，其结构请参见图 4.8。

图 4.8　使用所有方法的情况



4.4.10　各种方式的引入难度

空间分割法、实例法、平行世界方式这 3 种类型的方法都用于在玩家数增加后，确保系统可扩展。

但是，参与游戏的玩家什么时候会超过最大限制是很难事先预料的。在游戏还没全部完成，并且尚未正式开始运营时，很难对大量的用户资源进行分割来为将来的扩展做准备。但是如果所做的设计会导致之后很难进行修改，服务器负荷加重而导致玩家无法登录游戏的情况持续 1 周的话，玩家对游戏的评价或许就会一落千丈。一般来讲，一旦对网络游戏的评价下降之后，就不会再有起色了，为了避免这种情况的发生，应该怎么做才好呢？

通常，后期引入平行世界方式还是比较简单的，可以在一定程度上实现实时追加。但是在采用空间分割和实例法时，游戏数据的存储和制定、客户端的处理逻辑以及用户界面等很多地方都需要更改，所以后期引入需要很长的时间。尤其是在已经运行着的游戏中，要向玩家说明这一情况也很困难。就笔者所知，在运营正式开始之后，不可能有效引入后面两种方法来实现可扩展性。

4.4.11　各个世界中数据库（游戏数据库）服务器的绝对性能的提高

一般来讲，要提高游戏数据库的综合性能只能采用平行世界方式，但是为了让每个世界能够支持数千以上的同时在线玩家，还必须考虑一些方法来提高各个世界中游戏数据库的绝对性能。要想使用实例法来妥善处理更多同时在线的玩家，这些方法是必不可少的。

随着 DBMS 处理性能的提高和服务器性能的提高，游戏数据库的绝对性能也在持续提高。但是每秒处理事务的性能，3 年也提高不了 4 倍。当然，今后，随着 SSD、重视高速性的 KVS（Key-Value Store，键值存储）技术，以及 DBMS 的表压缩等技术的发展，事务处理性能可能也会受到影响，但在现在，这些是否符合 C/S MMO 游戏的要素还不明确。

因此，目前的现状就是，为了设法达到数倍、数十倍的高速，在应用程序层面上追加某些方法是不可或缺的。这里，我们来看一下不依赖平行世界方式来提高性能的一些技巧。

应用层面的技巧

提高各个世界中游戏数据库的绝对性能的代表性方法是：在保存玩家角色的状态变化（该部分占了访问过程中的绝大部分）时，设置执行写入缓存的中间高速缓存服务器。

这种方式最大程度地发挥了游戏过程中的一个特点：“玩家角色的状态变化是局部于用户 ID 的”。虽然对游戏数据库进行写入操作的频率很高，但是同一个 ID 的玩家角色的信息是反复保存多次的。

比如，假设同时连接数为 1 万名玩家，每个玩家角色的信息为 2 兆字节，如果具有 2 兆字节×10 000 = 20 吉字节的内存，那么几乎所有的写入操作都能在内存中缓存。这样，如果两次中有 1 次是保存到数据库表，负荷就基本可以减少一半。

如果想要使所开发的游戏大获成功，必须准备这种中间缓存层。在实现数据库（后面的进程关系图中所介绍的 dbsv 数据库）访问网关的功能的过程中，自然要实现这个功能。

4.4.12 *K Online* 的设计估算——首先从同时在线数开始

接着，我们就对示例游戏 *K Online* 的系统基本结构图中所涉及的各个部分进行估算。

这里，我们考虑与玩家数众多的中国网吧签订特殊的宣传合同。这样我们必须假定同时在线数将会达到 3 万。截至原稿撰写时（2010 年 9 月），中国最具人气的游戏达到了 60 万的同时在线数，所以在中国市场上，如果进行大规模宣传，必须预计会有 3 万的同时在线数。“3 万的同时在线数”这个数字是之后进行各种估算的根据。

瓶颈的确认

首先我们来确认一下前面说到的 4 种瓶颈。

❶ 客户端渲染性能的瓶颈

预计画面上需要同时渲染的敌人和 NPC 的数量为 10～20 左右，渲染性能验证的结果是，即使运行在性能较差的 Windows PC 上，渲染性能也没有问题（详略）。

❷ 用户侧线路带宽的瓶颈

平均下来不到 10kbit/s，完全没有问题。

❸ 游戏服务器的游戏逻辑处理性能的瓶颈

这里的瓶颈在于 CPU 的处理性能，所以需要估算实际处理的游戏逻辑的利用量。

❹ 游戏数据库写入性能的瓶颈

瓶颈在于数据库的存储性能，所以需要估算实际要存储的数据内容、数据量以及存储频率。

问题在于 ❸、❹ 两点。下面我们针对 ❸ 游戏服务器的游戏逻辑处理性能的瓶颈和 ❹ 游戏数据库的写入性能的瓶颈进行估计。

设计估算的思考原则

思考原则有以下两点。

❶ 对特别耗费处理成本的部分进行估计，求出“绝对的服务器数量”。

❷ 对特别难以扩展的部分进行估计，求出“每个平行世界可能扩充的最大服务器数”[12]。

> [12] 这里“每个平行世界可能扩充的最大服务器数”指的是，不对后端用来存储游戏数据的游戏数据库进行分割时，所能支持的最大服务器数（依赖于后端的所有服务器 + 后端服务器自身）。如果游戏数据库的性能无限高，平行世界的数量只要 1 个就可以了（不需要平行世界）。但是实际上，前端服务器达到一定数量后，数据库的写入瓶颈就会变得严重起来，所以就要考虑 1 个游戏数据库可以扩充多少台服务器，求出“每个平行世界可能扩充的最大服务器数”。

❶ 中耗费成本的部分在于敌人的行动算法，以及与同时在线数成正比增加的“游戏逻辑”主体部分的处理。❷ 中难以扩展的部分是“游戏数据库”的处理。明确了 ❶、❷ 这两点，就能根据“平行世界数 = 绝对的服务器数 / 每个平行世界可能扩充的最大服务器数”这个等式来计算平行世界的数量。

平行世界越少越好，这是个前提，最好是 1 个，所以在技术上需要讨论能够减少多少台绝对的服务器、每个平行世界可以扩充的最大服务器数能够增加多少这些问题。我们的目标是精度“相差不到 2 倍”，下面我们就以此为目标来进行估计。

4.4.13　根据游戏逻辑的处理成本来估算——敌人的行动算法需要消耗多少 CPU

在游戏逻辑处理成本的估算方面，有一种观点认为“很多情况下最内侧的循环占了执行时间的 8 成”。在 *K Online* 中，从策划内容来看，CPU 执行的游戏逻辑很大一部分都把时间花在“敌人的行动算法”中（除此之外就是误差程度）。这里要计算出“1 个服务器内核能够处理多少同时在线数”。其结果是“1 个内核处理 500 个同时在线玩家”。下面我们将介绍这个结论是如何得到的。为了得出结果，只要对等式“同时在线数 × 每个玩家所面对的敌人数 × 每秒敌人的行动次数 × 每次行动所需的 CPU 时间 × 安全系数 = 1 秒” 进行求解就可以了。

在上面这个等式中，“同时在线数”是需要求解的值，每个玩家所面对的敌人数、每秒敌人的行动次数、每次行动所需的 CPU 时间都能根据游戏的策划内容推断出来，这样最后就能求出 1 个 CPU（1 个内核）时间内所能处理的同时在线数了。

- 每个玩家所面对的敌人数

  →首先，根据 *K Online* 的策划内容，我们要实现“整个画面被敌人包围，玩家要在其中杀出重围”的场景。“整个画面”具体指 20 个左右的敌人。在最坏的情况下，*K Online* 的大多数玩家都没有组队作战，而是一个个分开行动，不断攻击敌人、拾取地上的物品。所以，最坏的情况就是每个玩家可能同时面对 10～20 个左右的敌人。

- 每秒敌人的行动次数

  →在 *K Online* 中，玩家角色的移动速度更快一些（参考 *Runescape* ），策划要求每秒处理 5 次。

- 每次行动所需的 CPU 时间

  →大致需要以下这些处理。

  ❶ 通过简单的循环来检查敌人周围的地形数据

  ❷ 查找附近的玩家角色

  ❸ 通过非常简单的排序方法来决定下一次的行动

  ❹ 执行 1 次预定的行动

  上面 ❶～❹ 这几个方面可以通过编写小段程序来进行验证，所以可以编写游戏逻辑来尝试。由此得到，每次行动平均需要 10 微秒。娴熟的开发人员在平时，总会经常无意间进行这样的计算，但在 C/S MMO 中，在实现服务器时就需要运用这类计算。

- 安全系数

→目前，大约增加两倍就可以了吧。看上去很合适，但是根据经验，增加两倍是之后稍微下点工夫就能做到的，而增加 10 倍就需要一定的技术支持。

那么，将上面这些数值代入等式中，< 所要计算的值 >×20×5×10 微秒×2 = 1 秒，就可以求得“同时在线数”为 500。

网络、安全性、用于提高开发效率的间接成本也是需要考虑的，但是这些就像加在安全系数中这么小。但是如果使用了尚未取得实效的通信中间件，或许最好还是对其进行测定。

在 *K Online* 中，总共需要 3 万同时在线数的处理能力，所以可以得出，需要 3 万 /500 = 60 个内核所具有的处理能力。至此我们估算出了 *K Online* 整体需要的绝对的服务器数。

4.4.14　根据游戏数据库的处理负荷进行估算——找到“角色数据的保存频率”与“数据库负荷”的关系

接着我们来估算一下 1 个平行世界可能扩充的服务器数。如果可以扩充到 60 个以上，那怎么样呢？结果是需要 5 个平行世界，下面我们就介绍一下得到这个结论的计算过程。

我们转到另一个问题上：执行游戏处理的 60 个内核实际上如何配置呢？原则上，应该尽可能不影响玩家的游戏。如果只使用平行世界方式，从玩家角度来看，60 个服务器林立着，不仅引起混乱，而且还非常不方便。至少平行世界要设置 3 个左右，最多不超过 10 个（因此同时还需要使用空间分割法和实例法，这一点将在后面讲述）。

那么，1 个平行世界可以投入多少个处理游戏逻辑的内核呢？对此起决定性作用的是数据库（游戏数据库）的处理负荷。而数据库的处理负荷一般由写入性能来决定。

将写入成本以公式来表示的话就是：同时在线数 × 每个连接平均的数据存储频率 ×1 次存储所需的查询数 × 安全系数 = 数据库服务器总共可以查询的频率。这里，游戏数据库相关的写入成本一般是指对“角色数据的存储频率”和“数据库负荷”的权衡。也就是说，如果存储频率低，玩家好不容易在游戏过程中积累起来的进展就容易丢失，但是游戏数据库的负荷就会下降。这就需要根据策划内容决定两者之间的关系。接着我们来求解刚才的公式。

- **每个连接平均的数据存储频率**

  →首先我们来考虑一下每个连接平均的数据存储频率。*K Online* 与一般的 MMOG 相同，执行游戏逻辑处理的游戏服务器（gmsv）总是要执行一些复杂的、无法完全测试的游戏逻辑，所以无法完全避免程序崩溃问题。在发生崩溃的情况下，内存中的游戏过程信息就会丢失，于是就会发生游戏状态的回退（总的来说就是，角色明明已经升级了，但是现在又降下来了）。可以通过定期在数据库中存储游戏信息来防止这种情况。平均 1 分钟存储 1 次就基本上不会有什么问题了。

  K Online 可以以这个“1 分钟存储 1 次”的频率为基准吗？首先我们从策划内容开始判断。在 *K Online* 中，1 次战斗和拾取操作大约要花费 30 秒～5 分钟，与敌对角色的战斗一般需要 1 分钟～几分钟。通过这些游戏内部的基本活动，玩家角色可以获得随机定义的、或许有利于推动游戏进程的一些重要物品以及经验值。出于这些原因，如果能平均 1 分钟存储 1 次，那这些游戏相关的更新就能被存储了。因此，我们可以判断，在 *K Online* 中 1 分钟存储 1 次是很合适的。整体的同时在线数是 3 万，除以 60 秒，也就是 1 秒存储 500 次。

- **1 次存储所需的查询数**

  →接着，我们来考虑 1 次存储所需的查询数。角色数据可以有两种方式来存储：一种是以 BLOB 等形式，每 1 行存储 1 个角色的信息；第二种是将角色数据的各个参数 [13] 作为数据库的表字段来存储。后一种方式对数据的复用性更高、检索性更好，也更容易对用户提供支持，但是数据库的性能就有所下降。这就需要加以权衡。在 *K Online* 中，首先要考虑的就是最大限度地高效使用数据。为了存储 1 名玩家角色的游戏数据，假设进行 10 次 SQL UPDATE 操作。

- **安全系数**

  →最后，安全系数与之前计算内核数时一样，也是 2。

[13] HP、MP（Magic Point）等。

将上面这些数值代入之前的公式中，可以得到：3 万 ×（1/60）×10×2 = 10 000。

具备与正式服务中使用的服务器相同的设备，并且配置用于备份的副本，测定实际的 MySQL 速度，在行数非常多的情况下，如果是 2000 次查询 / 秒的程度，那么处理起来就很宽裕。因此，绝对要达到的 10 000 次查询 / 秒除以 2000，就可以得到平行世界的数量为“5”。

通过以上计算，我们估算出“平行世界数 = 5”，“每个平行世界的同时在线数 = 6000”。每个平行世界的游戏服务器平均有 12 个内核（60 内核 /5）。

4.4.15　可扩展性的最低讨论结果，追求进一步的用户体验

通过以上讨论，有关服务器的可扩展性问题已经告一段落了，但是为了追求更好的用户体验，我们还要进一步进行探讨。

首先，1 个平行世界中有 12 台游戏服务器的情况下，要将哪个玩家分配到哪个服务器中呢？如果要让两名玩家共享相同的游戏画面，一起进行游戏，他们必须登录到同一台服务器。通常，玩家通过聊天等方式商量决定“今天上 3 号服务器进行游戏”，然后在该服务器上碰头。最简单的就是将 12 个服务器以列表形式显示在界面上，让玩家从中手动选择，很多情况下这样是可以的，但是非常麻烦。而 *K Online* 的负责人认为进行这样的选择很麻烦，所以希望有更为简单的方式。

那么，使用空间分割法和实例法的话能否改善这种情况呢？理想的用户体验是“服务器完全是自动选择的，玩家完全不需要自己考虑”。首先，根据对参考游戏 *Runescape* 的研究，可以知道 *K Online* 中玩家的分布情况。

- **50%～70% 的玩家（同时在线 3000～4000 人）集中在游戏副本、塔、广场等 20 个左右的小场所中。**
- **剩余的玩家（同时在线 2000～3000 人）在广阔的地上世界中冒险。**

这里同时在线总数为 6000。

游戏世界大致分为 8 个岛和大陆，此外，游戏副本和塔等有 20 个左右。所以合计为 28 个，比游戏服务器的数量（12 台）大得多，所以根据地区来划分服务器，当玩家角色从一个地区移动到另一个地区时，如果能自动切换服务器，玩家就可以省去对服务器进行选择的操作了。

这样就能大幅改善用户体验了，但是我们不得不承认，这种方式有一个问题，在发生一些特殊事件（比如政治家 [14] 的演说突然开始、或者发生游戏平衡性出现问题）时，玩家就会集中在某个地区内，负责该地区的逻辑处理的服务器就无法应对这种情况了。想要移动到该地区时，如果发生了这种情况，就只能给出“服务器已满，无法进入”的提示。但这毕竟是特殊状况，所以最好还是便利性高一些。

> [14] 游戏的参与方法 / 立场的一种。执行民主主义战略决定的角色。

但是，负责人又提出了更高的要求。塔和游戏副本需要尽可能做成狭窄的地形和细长的道路，否则游戏性就会下降，但是那里集中了 100 人以上的话人口密度又会过高，游戏体验就会因此而恶化，负责人希望能设法解决这个问题。

这里所说的游戏体验恶化，具体是指，当大量玩家集中在一个狭小的空间中时，“对敌方怪物和物品的争抢”就会频繁发生。所以，为了降低人口密度，还是应该将游戏副本实例化比较好。至于实例化的单位，根据副本地图的大小和长度来考虑，平均 10～20 个玩家 1 个实例。

这里假设总共有 3600 名玩家同时访问塔和副本等的实例。在 *K Online* 中，平均必须要有 10～20 名玩家同时进入副本展开与敌人的战斗（特别是在与 boss 战斗时），所以假设平均每个副本 10 人，就需要 3600/10 = 360 个实例。之前已经估算出，服务器的每个内核可以处理 500 个同时在线。3600/500 = 7.2 个内核的话就可以了，考虑到比起一般的处理，实例处理多少增加了地图数据初始化等额外工作，再加上安全系数，估计需要 16 个核心。

4.4.16 服务器的基本结构，1制定系统基本结构图

K Online 的服务器基本结构总结如下，这反映了以上的讨论内容。

❶ 收费认证服务器是通用的。

❷ 分割为 5 个平行世界，1 个世界允许同时在线 6000 人，总共 3 万人同时访问。

❸ 1 个世界分为 8 个地区（8 核）。

❹ 1 个世界准备 360 个实例。

❺ 玩家继续增加的情况下，追加平行世界。

这样，*K Online* 就同时使用了 3 种方法，玩家的体验也不会受到太大影响了，同时还预见了游戏的大规模化，确保了服务器的扩展性。根据服务质量，实际需要多少内核数会有所变化，所以本书不再进一步讨论。

根据以上所述，图 4.9 对“系统基本结构图”进行了总结。

图 4.9　*K Online* 的系统基本结构图

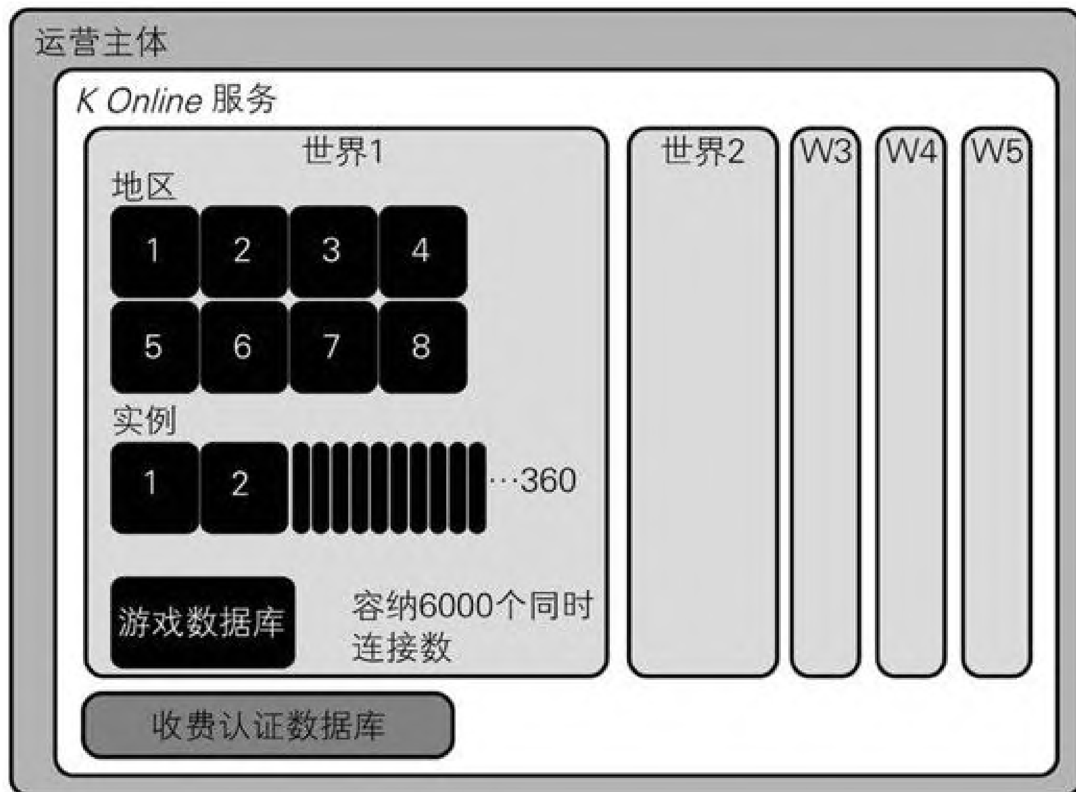


图 4.9 中，运营主体是指，比如将 *K Online* 授权给越南的游戏运营公司时，图 4.9 所示的系统结构应该运行在越南本土所设置的数据中心

中，在这种情况下运营主体会增加。在通过游戏进行交流的方面，即使运营主体增加了，收费验证数据库还是要求通用的，这部分在之后不易更改，所以需要在这里予以明确。

4.5 2进程关系图的制定

系统的基本结构定义之后，就能进一步讨论需要构建怎样的进程结构了。下面我们就来着手制定进程关系图。

4.5.1 2进程关系图的准备

首先，为了实现 *K Online* ，表 4.3 列举了作为其构成要素的必要的进程。请牢记各个服务器、进程的作用。我们直接先讲结论，之后将会说明原因。

4.5.2 服务器连接的结构——只用空间分割法

我们通常按照图 4.10 所示的方式来连接表 4.3 中列出的服务器。图 4.10 只用了空间分割法。图 4.10 中的一些关键点总结如下。

- gmsv、loginsv、msgsv、proxy 这些靠近服务前方的进程称为“前端”（front-end），除此之外都称为“后端”（back-end）。
- 实线是用来表示服务器的连接必不可少的线，虚线表示即使切断也无所谓的线。
- 黑色四边形所示的部分表示在不使用中间件时，必须自己制作。灰色四边形表示使用 MySQL 等现成的程序。
- 所有的 gmsv 都连接到 dbsv、authsv、worldsv 上。
- backup 指通过单纯的复制来备份数据库。
- gmsv 能增加到 3 个以上，目标是 20～30 个。
- 1 个 gmsv 对应 1 个 proxy，两个以上也可以。

- msgsv 和 loginsv 各有 1 个基本上就够了（因为并不总会造成负荷）。
- authsv 通常也不需要多个。

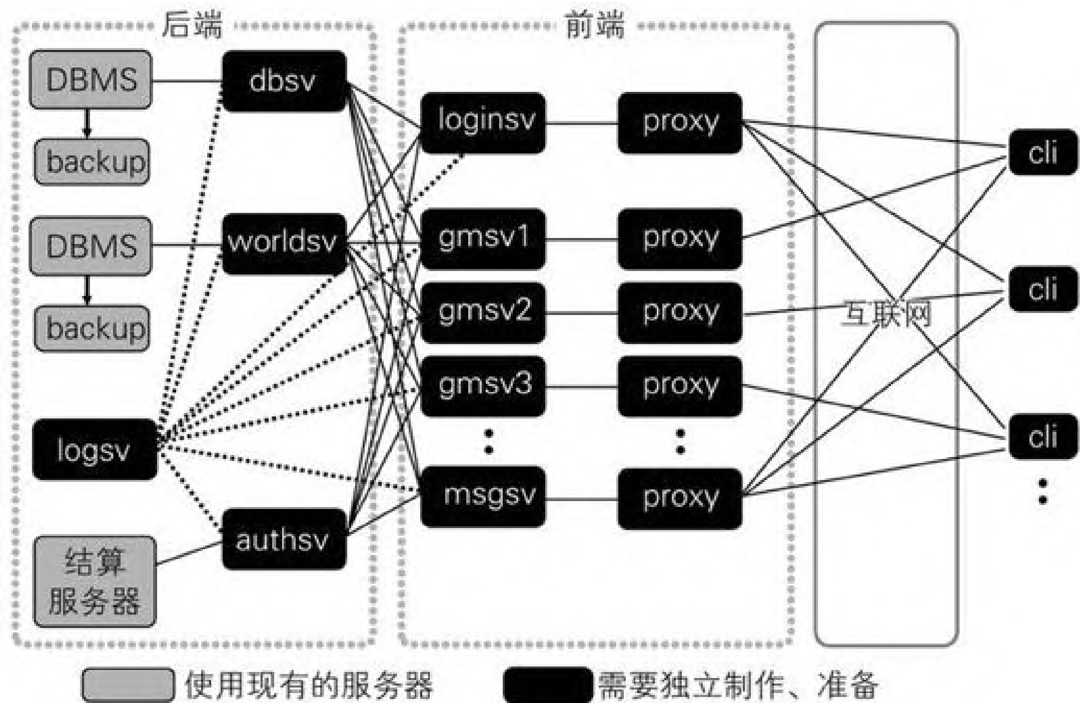
表 4.3　构成 *K Online* 的服务器、进程

名称	本书中的缩略语	用途
客户端	cli	用户直接使用的、用于访问游戏服务器的专用程序。运行在用户终端上
游戏服务器	gmsv	用来执行敌人的行动、地形判断、事件、角色升级、作弊检测等用于管理游戏数据的逻辑
登录服务器	loginsv	游戏客户端（cli）登录游戏时，最早要连接的服务器。负责服务整体使用情况的管理、负荷的控制以及会话密钥的发放等
数据库服务器	dbsv	对 MySQL 和 Oracle 等 DBMS 进行统一的连接，实现游戏服务器异步访问数据库时所需进行的负荷减轻处理
反向代理服务器	proxy	接受来自多个客户端的同时连接（数千左右的规模），负责进行 TCP 会话本身的管理、解压缩以及解密等处理，减轻游戏服务器的负荷
消息服务器	msgsv	用于使网上聊天、即时消息、公会等社交活动能够跨越平行世界和空间分割来进行消息交换。也有将这个功能整合在 gmsv 中的，但是由于进行修改的时机有所差异，所以为了尽可能使 gmsv 保持功能的单一，大多将这些消息处理交由专用服务器来实现。此外，还要在该服务器上实现用于防止玩家二次登录的“锁机制”
世界服务器	worldsv	使用空间分割的情况下，属于该世界的所有 gmsv 连接至这个服务器进程，负责各个世界共同需要的处理

名称	本书中的缩略语	用途
通用数据库服务器（整体通用的数据库服务器）	commondbsv	在使用平行世界方式的情况下，用于实现所有世界都需要的内存处理。除了作为 msgsv 的后端，还多用来进行实时统计处理和等级排名等处理
收费认证服务器	authsv	调用结算公司（结算代理公司）所提供的 API（大多是用 Perl 和 C 语言编写的程序库），相当于与结算公司之间的网关的服务器。结算公司提供的 API 大多都是同步 API，所以游戏服务器必须做到异步访问这些服务器
日志服务器	logsv	通过 TCP 收集整个游戏服务中的日志，根据时间顺序全部保存在文件中，用于进行循环、检索等处理
DBMS	DBMS	MySQL、Oracle、PostgreSQL 等数据库管理服务器的进程。在商业 C/S MMO 中，为了达到需要的性能，这是必不可少的
结算公司服务器	结算 sv	结算公司所管理的服务器，在服务的外部系统中管理，从收费认证服务器经由互联网的 VPN（Virtual Private Network，虚拟专用网络）和专用线路进行连接。用于连接的协议为 TCP/IP 和结算公司提供的程序库所隐藏的专用协议

根据 *K Online* 的游戏内容，如果估算同时在线数为 3000～5000 左右，图 4.10 的结构就绰绰有余了。

图 4.10　使用空间分割法（包括空间复制）的进程关系图



4.5.3 服务器连接的结构——使用平行世界方式和空间分割法

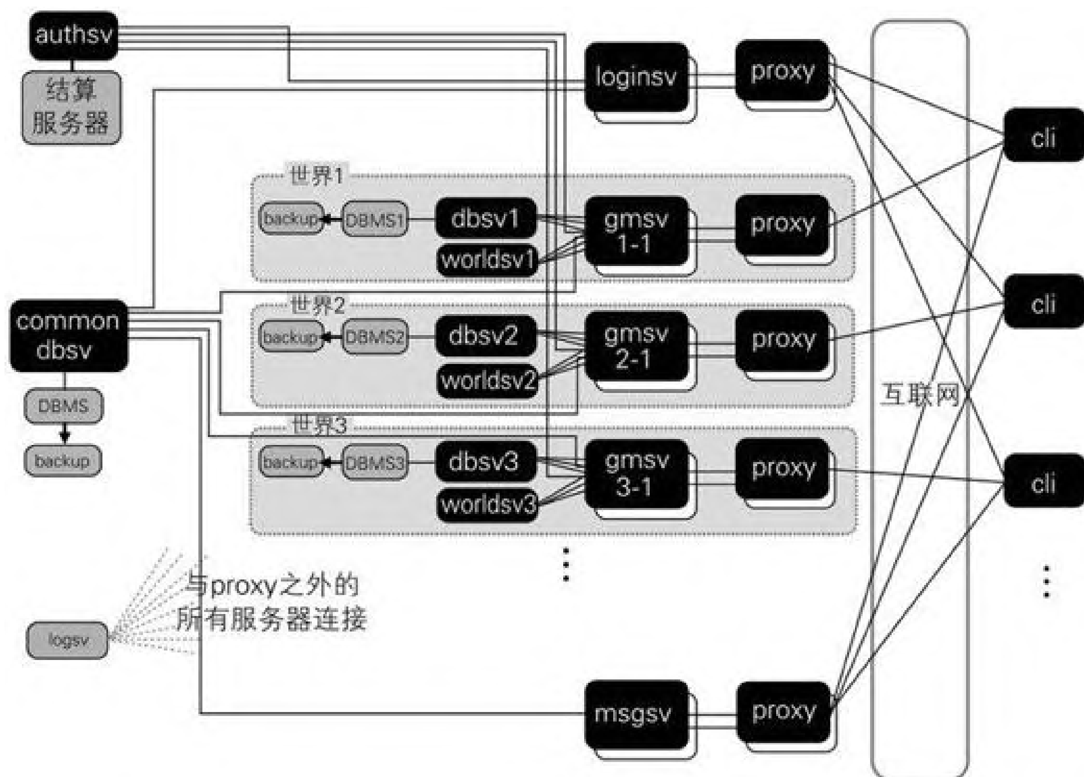
使用平行世界方式和使用空间分割法时，它们的关系图如图 4.11[15] 所示。

> [15] 图中的方框叠加部分表示多个服务器。

在仅使用空间分割法还无法达到足够的可扩展性的情况下，需要增加平行世界的个数。在图 4.11 中，世界 1、世界 2、世界 3 这 3 个灰色方框代表了 3 个世界。实际上，按照同样的这种结构，可以将世界的个数增加到 10 个。由于 1 个世界可以处理 1 万～2 万个同时连接，所以如果在这种结构下将世界增至 10 个，可以将同时连接数扩展到数十万的规模[16]。

> [16] 顺带一提，据说现在世界上同时在线数的最高纪录是 100 万～200 万人。

图 4.11 同时使用空间分割法和平行世界方式的情况下的进程关系图



4.5.4 使用平行世界方式进行扩展的关键点

使用平行世界方式进行扩展的关键是将 dbsv 分为 dbsv1、dbsv2、dbsv3 这样的多个数据库服务器，从而线性地提高存储游戏数据时的写入性能。将平行世界的单位作为 gmsv 的个数是因为改变游戏数据的进程只有 gmsv。loginsv 和 msgsv 与游戏进行没有直接关系，所以与 gmsv 的个数无关。

在 *K Online* 中，我们已经进行了估算。

- 收费认证服务器是共通的。
- 分为 5 个平行世界，1 个平行世界允许同时连接 6000 名玩家，总共允许 3 万名玩家同时访问游戏。
- 1 个平行世界分为 8 个地区（8 核）。
- 1 个平行世界准备 360 个实例（16 核）。
- 玩家继续增加的情况下，追加平行世界。

图 4.11 反映了这样的设计。

❶ 尽量使 authsv 进程并行化，实际的进程数由结算公司的程序库所约定的结算执行速度来决定。

❷ 各个世界中，worldsv 为 1 个进程，gmsv 中地区用到 8 个进程，实例用到 16 个进程，不能动态增减。

❸ proxy 与 gmsv 的个数相同（8 + 16）。

❹ dbsv、MySQL、备份用的 MySql 每个世界 1 套。

❺ msgsv 是所有世界共用的，尽量并行化。实际的进程数要根据之后的基准测试来决定。

❻ logsv 是所有世界共用的，只要 1 个进程，生成多少日志要在开发中决定。

详细的示意图这里就省略了，它反映了这些设定的进程关系图对之前的设计假设了进程的实际数量。图 4.11 的结构部分完全没有变化。

现阶段知道进程数的大致比例就可以了，实际的运营究竟要用到多少，则要在之后开发、构建服务运营体制时决定。在很多项目中，需要在发布前 3～4 个月确定服务器数量。

4.6　带宽 / 服务器资源估算文档的制定

接着我们来看一下带宽 / 服务器资源估算文档。在上一节中，我们完成了进程结构图的制定，随后我们就要开始讨论各个进程需要消耗多少服务器资源，并将其文档化。

4.6.1　以进程列表为基础估算服务器资源

表 4.3 的进程列表是带宽 / 服务器资源估算文档的基础。在表 4.4 的估算中，世界数以“W”表示。

在表 4.4 中，有与 W 参数有关的地方，也有无关的地方。这是因为世界数增加时，有些部分会随之增加，也有些不会增加。

容易成为瓶颈的地方以灰色表示，并各个备注栏里进行了简要的说明。以下对表 4.4 中的术语进行了补充。

- CPU（内核）

  表示所需的 CPU 内核数。1 表示常数，整个服务只要有 1 个就可以了。W×2 表示需要的个数为世界数的 2 倍。

- RAM

  服务器所需的物理内存。内核数×2 吉字节表示：如果内核数预计为 W×2，RAM 就必须达到 W×2×2 吉字节。

- 存储

  所需的存储量。事实上，对数据内容进行持久化的只有 MySQL，所以在 MySQL 的地方，存储量与 W 成正比，除此之外并不特别需要（网络引导等也可以）。

- TCP 会话数

  TCP 会话数以及通信吞吐量会给操作系统和路由等造成负荷，需要作为必要的资源来进行估计。因为只有 proxy 大量接收来自玩家的 TCP，所以除了 proxy，TCP 会话数成为瓶颈的可能性很低。

表 4.4　进程所需的服务器资源

进程	CPU（内核） RAM 存储器 TCP 会话数	备注
gmsv	W×（8 + 16） 上述×1 吉字节 	*K Online* 中的世界数据都是二维的，所以数据量并没有那么大，内存也只要每个内核具有 1 吉字节就足够了。游戏服务器的处理通常是以 CPU 为中心的，所以 CPU 成为瓶颈的可能性很高

	不需要	
	3～	
loginsv	W×1	通信缓存部分的内存。实际上 loginsv 很少会成为瓶颈，位于其后侧的 dbsv 和 authsv 大多会成为瓶颈
	上述×500兆字节	
	不需要	
	2～	
dbsv	W×1	通信缓存部分的内存。dbsv 中的MySQL 大多会成为瓶颈
	上述×500兆字节	
	不需要	
	8 + 16～	
proxy	W×（ 8 + 16）	通信缓存部分是十分必要的。其关键是 TCP/IP 的吞吐量性能能达到多少。但是如果 1 个 gmsv 分配 1 个 proxy，就不会成为瓶颈
	上述×500兆字节	
	不需要	
	500～	
msgsv	W×2	为了对玩家的在线状况进行管理，以及防止二次登录，内存和 CPU 都是必须的。内存和 CPU 都可能意外地成为瓶颈，所以在 beta测试之前必须先进行基准测试
	上述×2吉字节	
	不需要	
	3～	
worldsv	W×1	由于必须在一定程度上把握各个 gmsv 中的玩家的状态，所以内存是必需的。由于经常要增加排序和搜索处理，所以CPU 和内存竟然都很容易成为瓶颈
	上述×2吉字节	
	不需要	
	8 + 16～	
commondbsv	1	在平行世界方式下，进行与世界无关的持久化处理。存储设备更多地使用后端的 MySQL。成为瓶颈的是 MySQL
	上述×500兆字节	
	不需要	
	W	

authsv	W×1	只具有作为结算公司网关的功能。具备通信缓存部分的内存就可以了。authsv 中所存在的与结算公司之间的传输线路如果延迟，有时就会成为瓶颈
	上述×500 兆字节	
	不需要	
	2～	
logsv	1	当然，存储设备的容量会成为瓶颈。但是，如果日志数量太多，也无法有效利用。如果有 10TB，每个玩家可以保存 10 兆字节左右的日志，所以大部分的行为都可以记录下来。具有 10TB 就可以确保不会产生很大的成本。为了高效地对文件进行读写，最好保证一定程度的内存容量。此外，如果 gmsv 等前端服务器突然发送大量日志，网络吞吐量就会出现问题
	1 吉字节	
	10 钛字节	
	10 + W×30～	
MySQL	W×2	需要尽量避免对 HDD 的访问。有 100 万名（= 1M 人）活跃玩家时，如果内存为 8 吉字节，每个人8K，大部分的处理都可以在高速缓存中执行。通过在预算允许的范围内配置内存来避免 CPU 和存数设备的瓶颈
	内核数×8 吉字节	
	W×100 吉字节	
	1～	

对于所需的资源，如果可以做到表 4.4 这种程度的估算，那就能确认是否能根据假设的用户数来估算金额，并加入到预算中。估算金额时，粗略的估算就足够了。

4.6.2　以 CPU 为中心的服务器和以存储为中心的服务器

在进行估算时，服务器可以大致分为“以 CPU 为中心的服务器”和“以存储为中心的服务器”两大类。它们各自的特点如下所示。

- **以 CPU 为中心的服务器：CPU 较快，内核较多，内存一般，存储量少，容错性低。一次性的。**

- **以存储为中心的服务器：CPU 一般，内核一般，内存高，存储量大，容错性高。使用期长。**

4.6.3　服务器资源的成本估算——首先从初期费用开始

假设，以 CPU 为中心的服务器，每台配备 8 个内核，其中 6 个用于服务，而以存储为中心的服务器配备两个内核，其中 1 个用于服务，我们以此来进行估算。

在上一节2进程结构图中，我们知道，为了处理 3 万个同时连接，必须准备 5 个能够分别处理 6000 个同时连接的平行世界。因此，在 *K Online* 的成本估算中，W 为 5。

各服务器的估算如下。绝对数为 1 台的服务器，出于备用的目的再增加 1 台，这样无论什么时候都能使用。

- **以 CPU 为中心的服务器**
    - gmsv：5×（8 + 16）= 120 内核 = 20 台（1 台 6 内核）
    - proxy：5 ×（8 + 16）= 120 内核 = 20 台（1 台 6 内核）
    - msgsv：5 ×2 = 10 内核 = 2 台
    - loginsv：5×1 = 5 内核 = 1 台 + 备用 合计 2 台
    - commondbsv：1 = 1 台 + 备用 合计 2 台
    - authsv：1 = 1 台 + 备用 合计 2 台
    - dbsv：5×1 = 1 台，但是要分配给每个世界，所以要 5 台
    - worldsv：5×1 = 1 台，但是要分配给每个世界，所以要 5 台
- **以存储为中心的服务器**
    - logsv：1 = 1 核心 = 1 台
    - MySQL：5×2 = 10 核心 = 10 台

以 CPU 为中心的服务器合计 58 台，以存储为中心的服务器有 11 台。除去监控费用和配置费用，仅考虑单纯的初期费用的话，以 CPU 为中心

的服务器需要 20 万日元，以存储为中心的服务器需要 80 万日元，合计就是 20×58 + 11×80 = 1160 + 880 万日元 = 2040 万日元。再加上备用费，大概是 2300 万日元左右。至此，用于支持 3 万个同时连接的服务器成本就大致估算出来了。

服务器资源的维护成本

服务器维护成本分为两部分，第一部分是最初购入服务器设备的成本，第二部分就是恒定的维护成本。事实上，最初所需的成本与所有的费用比起来并不算多。恒定的维护成本包括：❶ 监控成本，❷ 更换故障设备的成本，其中服务器的更换费用并不高。故障服务器的更换，每 3 年全部更换一次就可以。

最大的问题是为了确保服务器处于正常状态而需要的监控成本。在 24 小时监控的情况下，聘用专业人员的话，1 台服务器每月需要花费 3 万～10 万日元。假设按照最低费用，*K Online* 中监控对象为 69 台，所以每个月为 69×3 = 207 万日元，每年需要花费 2500 万日元。3 年的话就比初期服务器费用高了两倍。所以，很有必要尽量减少服务器的数量。

4.6.4 带宽成本的估算

除了服务器资源，带宽方面也要花费很大的成本。带宽是所需资源中非常重要的一部分，所以必须对其消费量进行估算。在 C/S MMO 的情况下，对带宽消费量的估算非常简单。

98% 的传输量用于玩家和 NPC 的移动通知

在 C/S MMO 中，通信传输量的 98% 是用在玩家和 NPC 角色的移动通知上的。只要对显示在画面上的角色每秒移动几次进行估算就可以了。

在 *K Online* 中，1 名玩家大致与 20 名敌人作战。游戏画面是由二维平面构成的，所以 20 个角色的每个坐标可以用一个 4 字节的整数（int）来表示。

- 移动物体的 ID：4 字节
- X 坐标：4 字节

- Y 坐标：4 字节
- 移动方向等附加信息：8 字节

总计 20 字节，20 个角色就是 400 字节，再加上 TCP 报文头信息总共 440 字节，由于这些信息每秒发送 5 次，所以就是 2.2 千字节 / 秒，也就是 17.6kbit/s。这是 1 个人进行战斗时所需的平均传输量。这些都是从服务器发送给客户端的数据包。反之，从客户端给服务器发送数据包的时，由于只有在玩家进行某些操作时才会发送 1 次，所以基本上可以忽略[17]。

> [17] Web 服务中，发送至服务器的传输量也是小到可以忽略，这一点两者是相同的。

在 *K Online* 中，游戏内容并非全部都是战斗，如果有一半左右的玩家在战斗，那么每个人平均需要 17.6/2 = 8.8kbit/s 的带宽。3 万人同时在线时，就需要 264Mbit/s。通常，1Mbit/s 每月需要 1 万日元，所以所需的带宽成本大约为每月 200 万～300 万日元。1 年算下来比服务器的成本还高。

付费玩家与同时在线数是相同的，这是 C/S MMO 的定律。也就是说，将每月 200 万～300 万日元的成本除以 3 万，每个玩家需要支付 70 日元以上的带宽费用。这笔费用是高还是低，这是商业层面的判断，超出了本书的范围，但是玩家还是会觉得偏高吧。如果可以，最好还是控制在 20～30 日元之间。

4.6.5　带宽减半的方针——首先是调整策划，然后在程序上下功夫

出于商业角度判断，如果需要将带宽成本控制在每人每月 20～30 日元之间，那么带宽的使用量就要控制在一半以下。毫无疑问，如果放任不管当然是无法降低成本的。

通常，首先要考虑是否可以调整策划内容，接着再来考虑在程序上下功夫。虽然实际上这两者基本上都是需要的，但是通过调整策划，可以大幅度降低通信传输量。在程序上下功夫不仅会延长开发时间，也会增加代码的复杂度，导致可维护性下降。而策划的调整，则可以在保持程序简单的情况下降低传输量。

策划的调整 ——带宽降低方案1

那么，要如何调整策划呢？我们首先来考虑一下，之前说过，战斗中的移动数据每秒发送 5 次，那能不能只发送两次呢？如果可以，那么通过牺牲一部分游戏动作性，就能将传输量减少一半。

事实上，在 *Ultima Online* 等游戏中也是每秒发送两次的，所以我们以此为参考，考虑一下发送次数降低是否会影响到战斗的趣味性。

那么我们就来具体看一下。比如，对于移动速度很快的敌人，考虑使其不是一步一步行走，而是以一半的频率、一次移动两步以上。这就需要从游戏策划的层面上进行深入探讨，比如，这么做会对敌人和地形之间的关系造成怎样的影响呢？如果策划内容需要利用细微的地形差异，那么就可能出现问题。篇幅所限，这里不作详细讨论，但是在 *K Online* 的情况下，每秒两次也是可以的吧。如果处理得当也许可以将传输量降低一半左右。

如果对策划内容进行更为详尽的审查，或许可以提高传输量估算的精度。传输量的估算是通过对一些恶劣情况的假设计算出的。但是仔细分析一下策划内容就可以知道其实并不需要这么大的传输量。

这里，我们对“每个玩家与 20 名敌人作战”这一点进行分析。20 名这个数量其实是相当多的，事实上，很难想象所有的玩家都在到处与大量的敌人作战，平均下来很可能在 10 名以下。因此，策划人员要对制作中的地图以及敌人的分布进行确认，明确一下那些敌人密度很高的地区所占的比例。经过确认之后，就能知道其实在一半以上的地区中，敌人的密度都很低，只有 2～3 名同时出现。因此，将敌人的数量估算为原先的一半也没有问题。

在程序上下功夫 ——带宽降低方案2

在程序上下功夫又如何呢？有一个非常简单的方法。虽然有 10～20 名敌人在画面上移动，但并非所有的敌人都在玩家角色的旁边。很多都在稍微离开一些的地方。因此可以这么考虑，在玩家角色 3 步以内的敌人每次移动时，都发送 1 次相关的信息，而更远一些的敌人，则是每两次移动发送 1 次数据。当然还必须确认会不会对想要运用这种方法的策划内容造成影响。确认之后我们可以知道，在使用魔法进行攻击的情况下，虽然仍然必须正确地攻击远处的敌人，但即使将两次发送降低到 1 次，也没有任何问题。因此可以判断，远处的敌人可以以 50% 的一半的发送频率进行发送。这样一来，50% 的 50%。也就是可以降低 25%。

4.6.6　策划内容的分析对带宽的降低很有效

策划上的调整可以两次将传输量降低 1 半，而程序上的处理可以降低 1 次，所以之前估算出来的频率一下子就能降低到 1/4 以下了，这样就有希望实现收费玩家每人 20 日元的水平了。在这两种方法中，重新分析策划内容更为有效。从技术上进行估算时，仔细分析策划内容、频繁交换意见是项目成功的重要因素。

那么，对于成本和所需资源的估算，越是花时间提高分析的精度，就越能找到一些降低成本的方案。但是在实际推进开发工作之前，也要考虑到对所需资源的估算也是很花时间的。这次我们已经达到了每人 20 日元的程度，可谓非常低廉了，所以我们就到此为止，开始进入下一部分。

＊ ＊ ＊

对服务器资源和带宽的大致估算差不多完成了。接下来，我们就要将在网络上以及制成列表的各个进程之间以怎么样的顺序来传输怎么样的信息进行文档化。

4.7　4协议定义文档的制定——协议的基本性质

一般的协议是指，为了使计算机通过网络进行通信而互相决定的各种约定事项的集合。另一方面，在 C/S MMO 中，我们将“进程与进程之间以怎样的顺序交换怎样的内容”这一块称为“协议”。为了正确定义这些协议所需的文档则称为“协议定义文档”。请注意不要混淆。

4.7.1　4协议定义文档基础

协议定义文档中需要记录以下信息。

❶ 协议的基本性质（→本节）

❷ 协议的 API 规范。包括函数定义、参数定义、调用时序（→ 4.8 节、4.9 节）

❸ 数据包格式。包括分隔符（Delimiter）、大小、字节顺序等（→ 4.10 节）

C/S MMO 并不使用 HTTP，而是使用采用 TCP 的专用二进制协议，所以需要单独定义数据包的格式（在使用中间件的情况下就遵循中间件的规定）。首先，本节来讨论一下“❶ 协议的基本性质”，后续章节将会依次讨论 ❷、❸ 两点。

4.7.2 “协议的基本性质”的要点

在 C/S MMO 系统中，要在 TCP 的基础上构建专用的协议。那么这些协议的基本性质有哪些呢？主要有以下这些。

- 哪些作为服务器，哪些作为客户端。
- 永久连接，还是一次性连接。
- 是否需要在服务器端管理各个会话的状态（是否是有状态的）。
- 是否有必要管理认证状态。
- 1 对 1 ？ 1 对多？还是多对多？
- 是否要推送（Push）信息。
- 连接中断时是否需要立即结束服务。

接着我们针对各个协议回答以上问题。首先将进程关系图中出现的要素列出来，慎重起见，我们对各个简称加以补充，如下所示。

- 客户端→ cli
- 游戏服务器→ gmsv
- 登录服务器→ loginsv
- 消息服务器→ msgsv
- 数据库服务器→ dbsv

- （逆向代理服务器→ proxy）
- 世界服务器→ worldsv
- 全体公用服务器→ commondbsv
- 收费认证服务器→ authsv
- 日志服务器→ logsv
- DBMS → DBMS
- 结算公司服务器→结算 sv

4.7.3　协议的种类、以及进程之间关系的种类

典型的进程之间的关系如表 4.5 所示，各个服务器与其他进程使用专用的协议进行连接。这里，proxy 通常夹在 cli 和 gmsv 之间，不需要专用的协议，所以这里省略了。

表 4.5 中标有圆圈的地方表示进程之间要进行通信。表 4.5 中圆圈共有 22 个，根据游戏内容和详细设计，这个表是会有所变化的，这一点需要注意。*K Online* 中进程之间的关系就是该表所示的这种配置，但是在实际中，进程的种类可能会有所增减，或者需要与其他进程连接等。近来，也有为了进一步提高 gmsv 的性能，根据处理目的而将服务器进程分成战斗服务器、NPC 服务器的例子。

表 4.5　进程之间的关系

	cli	gmsv	loginsv	msgsv	dbsv	worldsv	commondbsv	authsv	logsv	DBMS	结算服务器
cli		○	○	○							
gmsv					○	○	○		○		

	cli	gmsv	loginsv	msgsv	dbsv	worldsv	commondbsv	authsv	logsv	DBMS	结算服务器
loginsv					○	○	○	○	○		
msgsv					○	○	○		○		
dbsv									○	○	
worldsv									○		
commondbsv									○		
authsv									○		○
logsv											
DBMS											
结算服务器											

4.7.4 8 种类型的协议

进程与进程之间需要互相通信时，必须要有一些通信协议。协议定义文档就是用于定义这些通信协议的。也就是说在 *K Online* 中需要定义 22 种协议（进程间的关系）。由于这 22 中协议各自对不同的内容进行传输，所以必须给它们取一些不同的名字以便区分。

但是这所有的 22 种协议都是构建在使用 TCP 从客户端连接至服务器的结构之上的，所以只要根据“连接到哪台服务器”来决定协议的名字就可以了。比如说，cli 要连接到 gmsv、loginsv、msgsv 这 3 种服务器上，所以在连接 gmsv 时就将对应的协议称为“gmsv 协议”。而 dbsv 被 gmsv、loginsv、msgsv 这 3 种服务器连接，不管是被哪个服务器连接，所对应的协议都称为“dbsv 协议”。

也就是说，只根据服务器的种类来决定协议的种类。这样就能简化协议的定义文档，也更易理解了。在 *K Online* 中，除去 DBMS，只要定义以下“8 种协议”就可以了。

- gmsv 协议
- loginsv 协议
- msgsv 协议
- dbsv 协议
- worldsv 协议
- commondbsv 协议
- authsv 协议
- logsv 协议

“sv”这个接尾词看着很啰嗦，习惯上也有将其简称为 game 协议、login 协议、msg 协议等，但是本书要对应各个服务器，所以还是使用 gmsv 协议这种叫法。

4.7.5 C/S MMO 采用 TCP

在网络游戏中，为了在进程之间进行通信，传输层采用 TCP 或 UDP 协议，异步传输则采用 RPC，这一点是基础（请参见第 0 章）。而在 C/S MMO 中通常“全部采用 TCP”。

这是因为一小部分（百分之几）的用户端路由设置是无法让 UDP 数据包通过的，为了避免这种情况，所以就不采用 UDP。虽然采用 UDP 有望在一定程度上提高吞吐量，但是要接受这个缺点就得不偿失了。另外，数

据中心内所有服务器进程之间的数据传输都必须非常可靠，所以还是采用 TCP 更为妥当。

所有的进程间通信基本上都是采用 TCP 之上的 RPC 来建立的，只要能对“进程之间的通信以怎样的顺序调用怎样的函数”进行定义，就能完成协议的定义了。

4.7.6 与“协议的基本性质”的对应

8 种类型的协议用在 22 种进程关系中，我们将之前所述的协议的基本性质归纳在了一个表里（表 4.6）。

表 4.6 “协议基本性质”一览

服务器 客户端	cli	gmsv ❶	loginsv	msgsv	dbsv	worldsv	commondbsv	authsv	logsv	DBMS	结算服务器
cli		stateful n:1 push	每次连接 n:1	stateful n:1 push harmful	❷						
gmsv					n:1	n:1	n:1		n:1 harmful		
loginsv					n:1	n:1	n:1	n:1	n:1 harmful		
msgsv					n:1	n:1	n:1		n:1 harmful		
dbsv								❸	n:1 harmful	1:1	
worldsv								❸'	n:1 harmful		
commondbsv									n:1 harmful	1:1	
authsv									n:1 harmful		n:1
logsv											
DBMS											
结算服务器											

- **哪些作为服务器，哪些作为客户端**

  通过表的列和行来表示。左侧的列表示客户端，上面的行表示服务器。

- **永久连接，还是一次性连接**

基本上所有都是始终连接的，不是的话就写明“随时连接”。

- **是否需要在服务器端管理各个会话的状态**

  需要的情况下记为“stateful”。

- **是否有必要管理认证状态**

  需要的情况下记为“auth”。

- **1 对 1 ？ 1 对多？还是多对多？**

  - 记为“1：1”、“1：n”、“n：n”。

- **是否要推送信息**

  - 要推送的记为“push”。

- **连接中断时是否需要立即结束服务**

  把这种情况作为 critical，但是基本上都是 critical 的，所以不需要的情况下就记为“harmful”。

在表 4.6 中，所有服务器在纵向上的值都是一种类型的。也就是说，一种服务器不会实现基本性质不同的多种协议。顺带一提，本书中称为 proxy 的服务器出于负载均衡的目的而配置在 cli 和 gmsv 之间（参照前述图 4.11 中的进程关系图），它一概不参与协议相关的处理，只是在 cli 和 gmsv 之间进行数据的交接，所以这里就省略了。

表 4.6 分为了 3 大组。下面对各组中为什么需要这些基本性质进行了总结。

- **表 4.6 中的组 ❶：从 cli 到各个前端服务器的连接**

  →表 4.6 ❶ 是从 cli 到各个前端服务器的连接。cli 数量高达数万，而 gmsv 的数量只有数十台。即使是 n:1，在数量上也是有相当大的差异。另外，cli 和 gmsv 是通过互联网来进行通信的，为了访问 gmsv 和 msgsv 的功能，必须进行严格的验证。

gmsv 实现了游戏过程中所需要的所有逻辑，所以当 gmsv 和 cli 之间的连接中断时，需要立刻停止游戏，但是 msgsv 是用于聊天的，由于游戏还是可以继续进行，有时不需要立刻关闭游戏客户端（*K Online* 就是如此）。

- **表 4.6 中的组 ❷：各个前端服务器与后端服务器之间的协议**

  →表 4.6 ❷ 是各个前端服务器与后端服务器之间的协议。不管是前端还是后端，这些所有的服务器都是配置在安全的数据中心内的，所以除了用于防止人为操作失误的验证机制，其他的验证机制一概不需要。所有与后端服务器的连接中断或超时的情况下，前端服务器则停止运行。特别是在 dbsv 停止的情况下，明明已经无法保存数据了，但如果还是保持这种状态继续进行游戏，存储在数据库中的状态和角色等的状态就会发生明显的不一致，这种情况必须避免，所以没有等待数秒尝试重连的空闲。

- **表 4.6 中的组 ❸❸'：具有某些低速输入输出处理的服务器**

  →表 4.6 ❸❸' 是具有某些低速输入输出处理的服务器。使用 DBMS 的唯一的服务器dbsv，以及使用作为外部服务的结算服务器 authsv，会进行低速输入输出处理。在无法使用 DBMS 和外部结算服务器的情况下，整个服务无法正常继续，所以 dbsv 和 authsv 都要停止运行。而由此，所有的服务都会发生连锁反应停止运行。

协议设计的基本策略

总体而言，出于对编程复杂度、调试容易度、测试简单化等方面的考虑，在与后端服务器通信时应尽可能无状态（Stateless）。*K Online* 的游戏内容可以对所有的后端服务器做到无状态。

此外，查询频率在每秒数十次到数百次以上的情况下，每次开始 TCP 会话所产生的开销会变得很高，所以需要选择永久连接而非一次性连接。如果一次性连接没有问题，采用使用 HTTP 的 Web API 形式来实现也是可以的。

4.8　4 协议定义文档——协议的 API 规范（概要）

至此，我们已经明确了每个服务器所实现的“协议的基本性质”。接下来，我们就来看一下有关协议 API、协议顺序方面的内容。本节首先简要介绍一下协议的 API 规范。

4.8.1 协议的实现原则

在 *K Online* 中，8 种服务器实现各自的协议。各协议的基本性质如前所述，但是各个服务器通过这些协议具体需要提供哪些功能，如何决定调用顺序呢？这里有一系列的原则。下面我们来依次看一下前端服务器和后端服务器的关系、协议的制定方法、异常以及 API 的调用顺序方面的一些原则。

后端实现基本的、通用的功能，前端实现专用功能

首先，一般来讲，如果要修改某些被依赖的基本要素，依赖这些部分的内容就要全部进行修改，在 C/S MMO 中也是如此。

因此，通过在后端实现基本的、通用的功能，而前端实现更为专用的功能，就可以降低系统的修改成本，提高开发效率。如果在各自的进程中实现各项功能，即使发生了内存访问冲突，也可以防止包含了相关功能的部分同时崩溃 [18] 。

> [18] Google Chrome 浏览器采用多进程也是出于同样的原因。

前端依赖于后端的结构

尽可能采用前端依赖于后端的结构。

实际使游戏进行下去的主服务器当然就是 gmsv 了。如前所述，在 gmsv 中，大量的游戏逻辑和游戏数据在内存中处理。从程序的复杂度和代码量（行数）来看，特别大的就是 gmsv 了。gmsv 经常需要修改，每周服务器维护时需要修改，每月要修改数十到数百次。

但是每次在修改游戏逻辑时，基本上不需要修改后端的功能。随着对游戏逻辑的修改，有时也要在数据库表中添加新列，但这种频率很低，在 100 次的 gmsv 逻辑修改中最多只会发生 1 次。1 年也只有几次。

在商业 C/S MMO 游戏中，必须每年进行两次“大规模更新”以挖掘潜在用户，开发过程中的列增加姑且不谈，在正式运营之后能进行增加就可

以了。事实上，“对于表 4.6 中的 ❷ 后端服务器，两年以上都不需要修改”的情况也很多。

综上所述，位于后端部分的服务器先行启动，前端部分的之后启动，然后连接至后端服务器，建立这样的依赖关系比较好。

协议是无状态和简单操作的集合

另外，应该尽可能使协议无状态。这与 Web 中的 REST[19] 的思想相同。不要保持协议的状态，这是因为为了实现应用程序，在客户端自由地进行必要的状态管理，可以简化难以修改的服务器端的实现。在 C/S MMO 中，通过将复杂的处理集中在 gmsv 中，可以将需要有状态的协议限制在 cli 和 gmsv 之间，这是最低限度的，大家尽量接近这一状态吧。*K Online* 的游戏内容也是可以实现这一点的。如果除了 gmsv 协议，还有其他协议需要是有状态的，那还应该进一步讨论是否真的需要。

> [19] Representational State Transfer。分布式超媒体结构类型的思想中的一种。其中作为代表性成功示例的 Web 结构是由资源及其标识符（URI），还有对其进行的 GET、POST、PUT、DELETE 这一系列简单的操作方法组成的。

再进一步看，还应该尽量使协议只具有简单的操作，这也是 REST 的概念。在 C/S MMO 的协议中，各个后端服务器的协议基本上可以只集中于 CRUD[20] 这种非常基本的操作。这样可以将后端服务器的修改频率控制在最低程度，从而提高系统整体的可维护性。在 gmsv 的协议方面，也应该尽量简化操作。

> [20] 即 Create、Retrieve（Read）、Update、Delete。这些是软件持久性的基本功能。在 RDBMS 中，就是数据的创建、查询、更新、删除这几个需要实现的主要功能。

在一个地方接受外部的异常状况

还有一个原则就是要尽量在一个地方接受和处理外部异常。比如，在访问 1 个文件时，如果在两个地方对文件读写进行系统调用（比如 write() 函数），那就在这两个地方都要检查返回值，如果又增加了好几个地方，检查起来就会很复杂，而且容易出错和遗漏，为了覆盖到这些情况，测试案例也会相应增加。所以，为了防止这种情况，实际调用 write() 的地方通常应该限制在 1 个地方，这样代码就能得以简化，程序也可以更为健壮。

C/S MMO 的服务器实现中使用到的输入输出方式只有“网络和文件”。这些都是通过调用套接字 API、数据库的 API 和 C 语言运行库等程序库和中间件来进行输入输出的，调用失败即为异常。

C/S MMO 服务器系统中最致命的，由外部原因引起的异常有以下两处。

- DBMS 查询失败

- 网络发送失败

这是因为在对文件进行读写时就不会执行了。

网络发送失败指的是，由于要在各个进程之间进行网络通信，所以每个进程都有发生异常的可能。

但是执行 DBMS 查询的地方是可以锁定的。在 *K Online* 的服务器设计中，之所以要在 1 个地方进行数据的持久化，就是因为对于游戏服务的进行，“DBMS 调用失败”这个最致命的异常只会在 1 个进程，也就是 dbsv 中发生。除了 dbsv，其他地方都不会发生这种情况，所以可以确确实实地抓住问题所在，发生数据库连接异常时的问题也很容易定位。

进行某些数据持久化操作的 API 应该尽可能集中在 dbsv 协议的一个地方。也许有人会因为担心瓶颈问题而想在系统中配置多个 DBMS，但是将其集中在 1 个地方反而更容易对瓶颈和 bug 进行检查和修改，这一点很关键。

优秀的 API 的调用时序 ——不调用才好吗

最后，API 的调用时序以下面的顺序为优。

- 不调用 API。也就是说不需要（笑。但这是认真的）

- 只调用而没有返回值的 API 的单向时序图（图 4.12）

- 只调用一次然后获取返回值的呈三角状的时序图（图 4.13）

- 呈锯齿状的时序图（图 4.14）

图 4.12　单向时序图

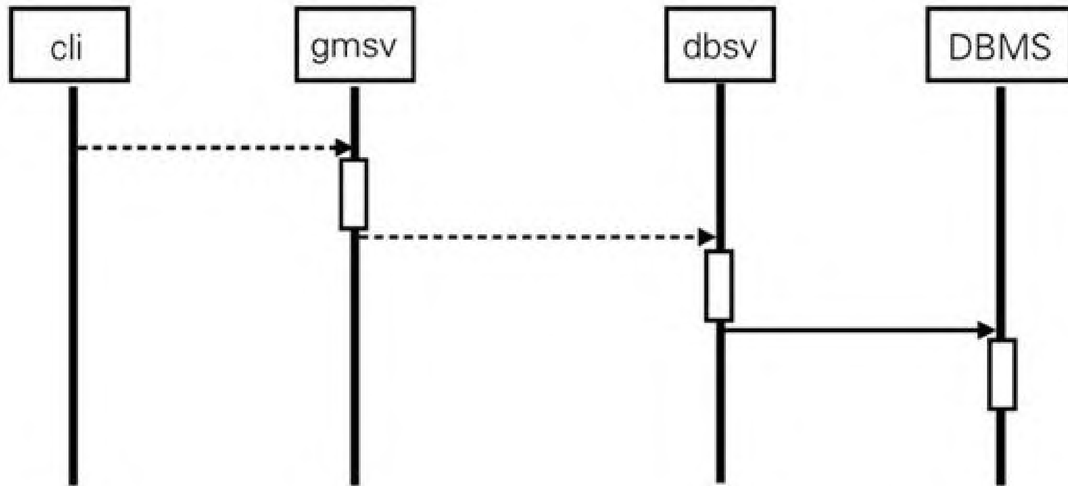


图 4.13　三角状时序图

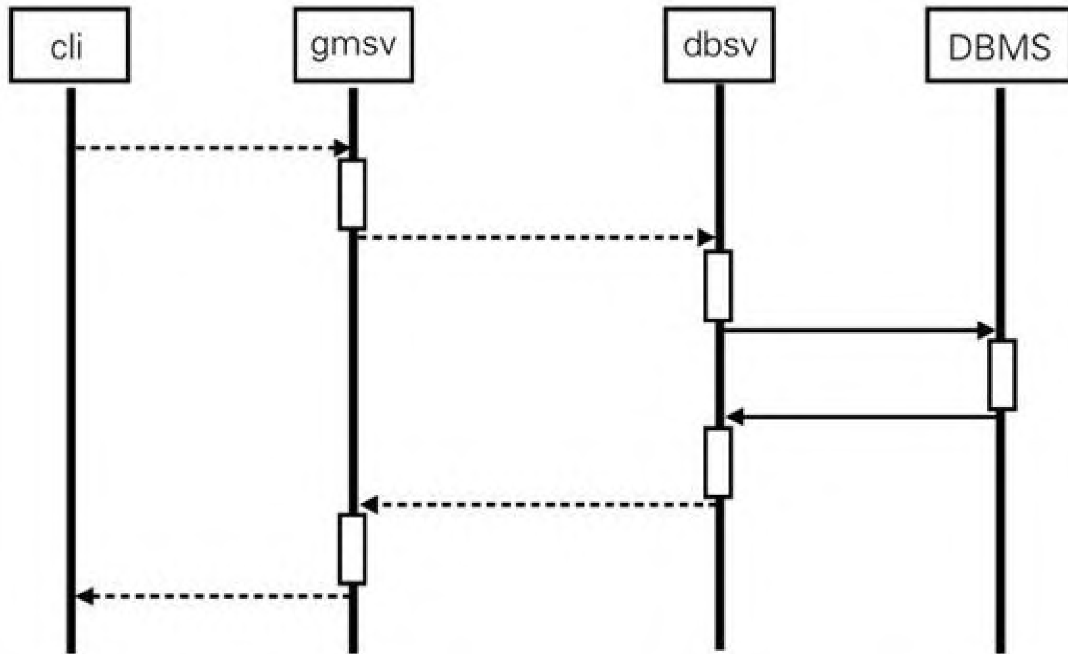
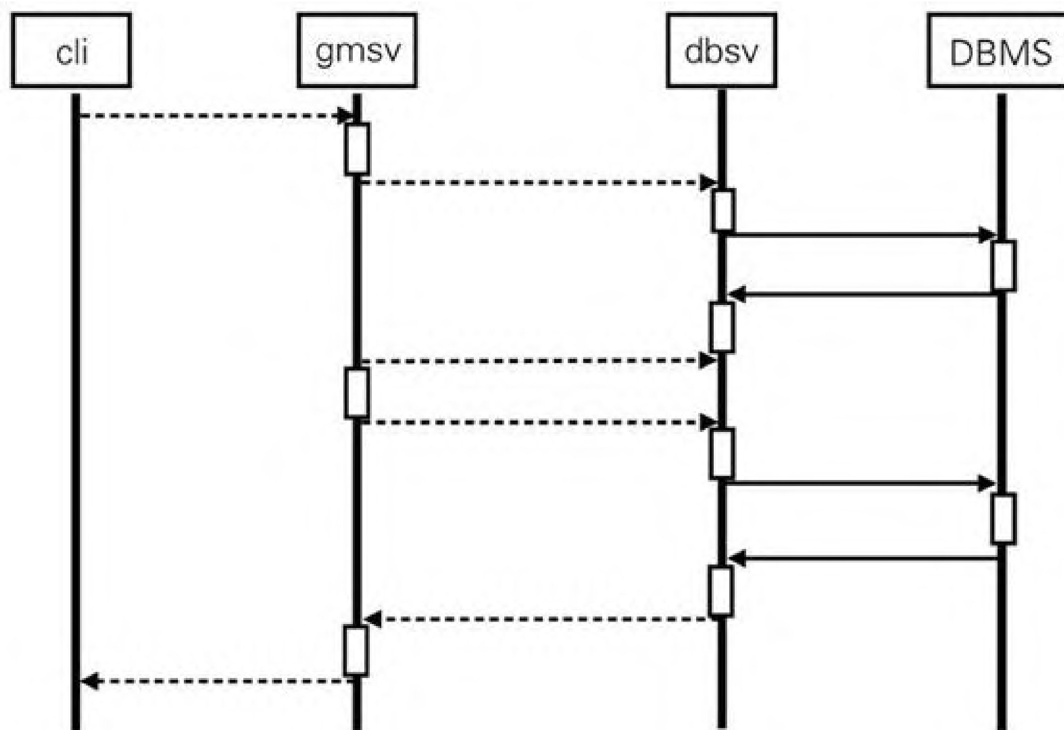


图 4.14　锯齿状时序图



gmsv 中采用的是使用比线程更轻量级的“回调”和“任务系统”来实现的异步编程[21]，所以为了实现图 4.14 所示的这种锯齿状的时序图，回调函数就要增加，导致程序过于复杂，更容易产生 bug 和时序异常。在 *K Online* 中，在处理角色制定和物品交易等需要互斥机制的部分时，可以采用图 4.13 的“三角状时序图”来实现，除此之外还可以采用图 4.12 的“线状时序图”来实现。当 C/S MMO 中存在“锯齿状时序图”时，就需要对其必要性重新加以讨论。

> [21] 这方面的内容在 0.2 节的 libevent 部分以及 0.4 节中讲过。请好好掌握。

- **有必要推送（Push）吗**

还有一点……将某些信息从服务器推送至客户端，或者说从后端推送至前端时，需要好好考虑一下。多用信息推送的问题就是，实际上调用该功能的是其他的进程或其他的程序，所以在服务器端程序中查看代码时，也弄不清是在哪里调用的。

比如，要从 dbsv 向 gmsv 推送某些信息，就要在 gmsv 中编写回调函数。但是实际调用该函数的是 dbsv，如果不看 dbsv 的代码，就无法把握整个程序。换句话说，应该在调用处的附近管理函数的返回。此外，

对接受异步推送的程序进行自动测试的程序也很难编写。如果这样来使用推送，程序的可维护性就会下降。

在 gmsv 和 msgsv 的协议中，无论如何都需要推送数据，这时推送就成为核心部分了，但是除此之外基本上不需要推送。如果随着设计的深入，需要推送时，就应该好好考虑一下是否可以排除这种可能性 [22] 。

> [22] 排除推送的最简单的方法就是轮询（Polling），多少比推送有所改善，但是“是否需要轮询”这本身也是个问题，所以在考虑时也请多加注意。

4.8.2　8 种协议的功能 / 形式概述

以上述原则为依据，我们来简单看一下各个协议以怎样的形式提供了怎样的功能（各服务器的作用前面已经介绍过，这里再次对其中的一部分进行说明）。

gmsv 协议

gmsv 协议负责执行管理敌人的行动、事件、角色升级、作弊行为检测等的游戏逻辑。

gmsv 针对 cli 实现的 API 大致可以分为以下 3 种。“通知”反映了只调用，没有返回值的单向时序图。“请求”反映了调用 1 次，期望返回 1 次的“三角状时序图”。

❶ 来自 cli 的通知

典型示例：使用鼠标移动角色时立即发送的移动通知 API。不需要返回值，之后消息从 gmsv 送达。全部通过写操作来改变 gmsv 内存中的数据。如果有必要，gmsv 需要通过 dbsv 对这些数据进行持久化。

❷ 来自 cli 的请求

典型示例：打开物品栏时，获取当前所持物品列表的 API。这是获取信息的请求。gmsv 原样返回内存中游戏状态的信息。

❸ 来自 gmsv 的通知

典型示例：敌人的行为。即使 cli 没有发送任何指示，gmsv 仍然要每秒发送数次通知。内存中的游戏状态发生变化的瞬间，gmsv 对必要的 cli 发送信息。

在 gmsv 的内存中，以对象数组或列表（List）来存储玩家角色、NPC、物体、效果、物品等，对此可以通过 ❶～❸ 的方式来访问。

loginsv 协议

loginsv 是游戏客户端（cli）登录游戏时最先连接的服务器。负责服务整体使用情况的管理、负荷的控制以及会话密钥的分配等。

loginsv 对 cli 提供的 API 只有从 cli 对 loginsv 发出请求，包括密码验证，以及查询能够使用的 gmsv 列表。

msgsv 协议

msgsv 是用于使聊天、即时消息、公会等社交活动能够跨越平行世界和空间分割来进行消息交换的服务器。

msgsv 对 cli 提供的 API 与 gmsv 一样，也大致分为 3 类。

❶ 来自 cli 的通知

典型示例：聊天输入，只是调用不需要返回值，所以一旦用户按下 Enter 键就立刻发送。

msgsv 接收到该消息后，就向其他玩家发出通知。好友的添加也是如此。

❷ 来自 cli 的请求

典型示例：获取好友列表。

❸ 来自 msgsv 的通知

典型示例：好友登录时发送的上线通知。作为 ❶ 的结果而发送的来自其他玩家的聊天消息也对应这一点。

dbsv 协议

对 MySQL 和 Oracle 等 DBMS 进行统一的连接，在异步访问数据库时，实现必要的负荷降低处理。

将 DBMS 中各个表的 CRUD 操作作为 API 来提供。虽然从 gmsv、loginsv、msgsv 等前端服务器进行访问，但只有以下两点不从 dbsv 推送数据。

❶ 来自前端服务器的通知

典型示例：玩家角色的保存。保存是绝对必要的，一旦保存失败就意味着 gmsv 需要强制终止，所以不需要返回值。

❷ 来自前端服务器的查询

典型示例：玩家角色的加载。

worldsv 协议

使用空间分割时，属于世界的所有 gmsv 都连接到这个服务器进程上，负责为各个世界提供通用处理。在 *K Online* 中，用来实现“显示世界地图（用来指示玩家处于游戏世界的哪个位置）”的功能。具体有以下两点，不从 worldsv 推送。

❶ 来自 gmsv 的通知

典型示例：保存所有在线玩家的坐标。

❷ 来自 gmsv 的请求

典型示例：获取包括其他 gmsv 在内的所有玩家的坐标。

worldsv 不需要对坐标信息进行持久化，所以不使用 DBMS。

commondbsv 协议

在使用平行方式的情况下，对所有的世界共同需要的信息进行持久化。在 *K Online* 中需要保存用户 ID、密码信息，以及好友列表信息。这些都是与世界无关的而又必要的信息。所有的 gmsv、loginsv、msgsv 连接至该服务器。

提供的 API 有以下两种，不推送。

❶ 来自 gmsv 的通知

典型示例：通知指定的 gmsv 中当前有多少人在线。

❷ 来自 loginsv、msgsv、gmsv 的请求

典型示例：获取各个 gmsv 中当前在线的总人数。

不使用平行世界方式时，不需要 commondbsv，这种情况下 commondbs 的所有功能都由 dbsv 来实现。

authsv 协议

authsv 调用结算公司所提供的 API（大多是 Perl 和 C 语言编写的程序库），相当于与结算公司之间的网关。

K Online 采用“每月定额计费”的方式，提供的 API 只有“来自 loginsv 的请求”，其内容只有“获取是否成功收取了某个玩家上个月的费用”，不推送。

logsv 协议

通过 TCP 收集整个游戏服务中的日志，根据时间顺序来罗列，保存在文件中，用于进行循环检索等处理。提供的 API 是“各服务器的日志写入”，只有单方向的通知，所以 没有请求、推送。

* * *

以上对各个协议进行了简要介绍，接下来我们进一步具体展开讨论。

4.9　4 协议定义文档——协议的 API 规范（详细）

至此，我们已经对构成 *K Online* 的服务器系统的 8 种服务器协议进行了简单说明。现在我们要对这 8 种服务器 API 的调用进行详细介绍。

4.9.1 协议 API 规范（详细）的制定

上一节简要介绍了各协议中所需具备的功能 / 形式。在详细的 API 规范方面，我们来了解一下针对具体实现的内容，具体包括“API 的函数定义”、“常量定义”，以及在制定时序图时，决定“API 的调用时序”。

4.9.2 API 的函数定义

各个服务器所提供的 API，也就是函数，在 *K Online* 这种规模的游戏中大致有 200～500 个。比如，进行信息持久化的数据库表有 30 种，各自包含 CRUD 操作和应用方面的 API，实际对各种数据库表的操作进行定义时，差不多就是这种程度的量。因为需要定义如此多的 API，所以通常使用中间件等支持工具，以某种机器可读的形式来定义函数、常量和类型，否则很难进行控制。

这里不涉及特定的中间件，而是使用类似于 C 语言的原始写法来定义函数。对所有的协议进行完全定义需要大量篇幅来讲解，所以这里我们只看一下代表性的 API 定义。

gmsv 协议

首先从 gmsv 协议的代表性 API 开始。

- **使用鼠标移动角色时，立即发送的移动通知**

```
void move(int x, int y);  ← ①one-way message（单向消息）
```

- **打开物品栏时，获取当前所持物品的列表**

```
void get_inventry_list(); ← ②query（查询）
void inventory_list(int item_id[]); ← ②' query result（查询结果）
```

- **从 gmsv 向 cli 通知敌人的行动**

```
void notify_move(int id, int x, int y);  ←③ notification（通知）
```

我们首先来看一下上面 ❶ 的部分。在 *K Online* 中，使用鼠标点击画面上的地面时，角色就会自动向那个坐标移动。这里只是通知，所以不需要返回值。上面 ❶ 中的注释“one-way message”就是这个意思。此外，请注意一下 API 的函数名。用 move、walk、say 等表示玩家实际操作的直接动词来命名。

接着，❷ 中的“inventry”是一个游戏开发术语，代表所持物品的列表。向服务器发 送 ❷ 的函数 get_inventry_list，就能向服务器请求自己角色所持有的物品的列表。服务器将该列表作为返回值返回给 cli。❷ 的注释“query”表示要求 1 个返回值的函数。通过将 API 的函数名命名为“get_XXX”这种形式，可以明确表示存在一个返回值。

针对 ❷ 的请求，从 gmsv 返回给 cli 是通过异步调用其他 API 来实现的。也就是上❷’这个 API。

因为在 *K Online* 中，所持物品的状态可以用整型数组来表示，所以可以通过 ❷’ 的 API，从服务器返回所持物品的所有信息。因为它所表示的是查询结果，所以注释标为“query result”。

在 gmsv 中，内存中管理的敌人是定期移动的。敌人移动指的是敌方目标的 *x* 坐标和 *y* 坐标持续发生变化。坐标变化时调用上述 ❸ 的 API。这里有一点很重要，❸ 是从 gmsv 对 cli 调用的通知。为了便于查看，在注释中记为“notification”。此外，将 API 命名为“nofity_XXX”，这样就能很容易知道这是从服务器发来的通知。

API 的类型和消息的特性

这里需要注意的是，one-way message（上述 ❶ 处）、query（❷ 处）和 notification（❸ 处）的 API 函数的返回值都是 void。这是因为所有的 API 调用都是异步的，其结果在之后通过回调函数来返回。

表 4.7 总结了各种类型的消息的特性。

表 4.7　API 的类型和消息的特性

类型	方向	返回值
one-way message	cli → gmsv	无
query	cli → gmsv → cli	有。一定有一个返回值，通过回调返回
notification	gmsv → cli	无

表 4.7 的关键之处在于 query 的逆向操作，也就是说，gmsv 用来获取来自 cli 的信息而进行的查询没有必要作为 API 来提供。这是因为在 C/S MMO 中，游戏相关的所有数据都集中在 gmsv 中，cli 中完全不存在需要 gmsv 知道的信息。

表 4.8 对以上内容进行了总结。

表 4.8　返回值的有无

	无返回值	有返回值
cli 调用	one-way message	query
gmsv 调用	notification	不存在

* * *

这里采用了原始的 C 语言风格的写法，通过 API 函数名和注释来避免错误的发生。进一步来讲，还有一种使用 IDL（接口描述语言）的代码生成程序的写法来进行实现的方法[23]。

> [23] 这方面的内容在 0.3 节中有所介绍，请参阅该节。

下面我们以同样的方式来对 gmsv 协议以外的协议进行定义。

loginsv 协议

游戏客户端（cli）在登录游戏时最先连接到 loginsv，由该服务器负责进行验证、管理服务整体的使用情况、控制负荷以及分配会话密钥等。这里以 loginsv 协议中最重要的验证处理为例进行说明。

```
void get_session_key(char email[], char password[]);   ← query
void session_key(int result, char key[]);    ← query result
```

K Online 中的验证是指，玩家从 cli 的终端程序输入自己的邮箱地址和密码，程序将其与存储在数据库中的信息进行核对，如果一致，就向该玩家分配一个直到登出之前都能用来访问服务器各项功能的会话密钥。顺利获得 session 密钥就能登录成功，不能获取到就会登录失败。email 和 password 都以 char 类型的数组来传递。

包含返回值的 result 存储着一些另外定义的常数。比如，使用 enum 定义的 SUCCSS、PASSWORD_ERROR 等常数。常数的定义将在后面介绍。

msgsv 协议

通过 msgsv 协议，聊天、即时消息、公会等社交活动就能够跨越平行世界和空间分割来进行消息的交换。

- **聊天消息的输入**

```
void say(char text[]);          ← ①one-way message
void notify_say(int said_by, char text[]);    ← ①'notification
```

- **好友列表的获取**

```
void get_friend_list();         ← ②query
void friend_list(friend_t contacts[]);        ← query result
```

- **上线通知**

```
void notify_presence(int);      ← ③notification
```

在上面的 ❶ 处，我们将聊天消息的输入作为单向消息的示例，msgsv 接收到这一消息后，就向其他玩家通报这一消息。也就是 ❶’ 的 notification。said_by 表示说话者的 ID，用于在画面上显示。

❷ 是 query 的示例：好友列表的获取。在该例中，msgsv 将 friend_t 类型的数组作为返回值返回。friend_t 含有玩家的 ID 编号和名字，其定义如下面的伪代码所示。像这样定义一个单独的类型，可以降低 API 定义的冗余性。

```
friend_t: {
    int player_id;
    char player_name[];
}
```

最后，❸ 处列举了一个 notification 的示例，好友登录时发送至相关 cli 的上线通知。

dbsv 协议

dbsv 协议是用于将游戏数据在数据库中进行持久化的协议。dbsv 是后端服务器，所以调用这个 API 的是 gmsv 和 msgsv 等前端服务器。

这里列举一个作为 one-way message 的示例：玩家角色数据的保存。

```
void save_character(int player_id, character_t data);   ← ①one-way
message
```

通过指定 player_id 来保存 data。数据的保存是绝对不可或缺的，如果保存失败，就意味着 gmsv 要强制终止，所以要自动终止 gmsv。因此，该函数不需要返回值。在 *K Online* 中，character_t 可以通过以下这样的伪代码来定义。

```
character_t : {
  unsigned int hp;  ←角色的当前体力值
  unsigned int maxhp;    ←角色的最大体力值
  unsigned int money;    ←所持有的金钱
       ...
}
```

query 的示例为玩家角色的加载：

```
void get_character(int player_id); ← ②query
void character(int player_id, character_t data);  ← ②'query result
```

此外，dbsv 协议中不存在 notification 类型的 API。

worldsv 协议

worldsv 协议是世界中所有 gmsv 都要连接的服务器协议，负责各个世界都需要的处理。*K Online* 中使用 worldsv 来实现“显示世界地图（用来指示玩家处于游戏世界的哪个位置）”的功能。

作为 one-way message 的示例，下面列举了一个保存所有在线玩家的坐标的 API。从 gmsv 向 worldsv 通知某个玩家当前所处的位置。worldsv 则不断在内存中存储这些信息。

```
void update_player_position(int player_id, int x, int y);   ← ①one-way message
```

接着我们来看一下 query 的例子，获取包括其他 gmsv 在内的所有玩家的坐标。

```
void get_all_player_position();    ← ②query
void all_player_position(player_position_t list); ← ②'query result
```

worldsv 接收到 get_all_player_position 后，就将内存中所有玩家的位置信息以 player_position_t 数组来返回。player_position_t 定义如下。

```
player_position_t {
  unsigned int player_id;
  unsigned int x, y;
}
```

worldsv 协议也没有 notification 类型的 API。

commondbsv 协议

commondbsv 协议是实现所有世界共同需要的内存中处理的进程。所有的 gmsv、loginsv、msgsv 都连接到该服务器上。

首先，我们来看一下 one-way message 的示例：通知指定服务器上当前有多少人在线的 API。

```
void update_concurrent_player_num(int world_id, int gmsv_id, int
num); ← ①one-way message
```

K Online 中存在着多个世界，所以要为各个世界赋予一个 ID。gmsv 中也设有该值，gmsv 将该值作为上述 API 的参数来指定。此外，每个 gmsv 都被分配了不同的 ID，所以在 commondbsv 中可以对各个 world_id 计算总人数。

接着我们来看一下 query 的示例，以下 API 用于获取各个 gmsv 中当前在线的总人数。

```
void get_concurrent_player_num();  ← ②query
void concurrent_player_num(concurrent_player_t list);  ← ②'query
result
```

在 commondbsv 中总计有多少人登录了哪一个世界，并将其作为 concurrent_player_t 类型的数组返回。concurrent_player_t 类型定义如下。

```
concurrent_player_t {
  int world_id;
  int num;
}
```

commondbsv 协议没有 notification 类型的 API。

authsv 协议

authsv 服务器进程调用结算公司提供的 API[24]，相当于与结算公司的网关。

> [24] 大多是 Perl 和 C 语言编写的程序库。

query 的示例：获取某一玩家是否成功支付了上个月的费用。

```
void get_payment(char email[]);     ← ②query
void payment(int result, char email[]); ← ②' query result
```

结算公司的数据库中管理着与玩家的 Email 地址等绑定的收费信息，所以在函数中指定 email[]。这里就不作详细介绍了，但是在别的地方定义了一些常数，比如 PAID（已支付）、UNPAID（未支付）等，这些常数就作为 result 的值返回。没有 notification 和 one-way message 类型的 API。

logsv 协议

通过 TCP 收集游戏服务中的所有日志，按时间顺序排列，保存在文件中，用于进行循环检索等处理。

提供的 API 只有 one-way message 类型的日志写入，定义如下。

```
void print_log(char message[]);    ← one-way message
```

logsv 调用上述 API 后，在日志文件中添加指定字符串。没有 query 和 notification 类型的 API。

4.9.3 常量定义

在定义了 API 函数之后，我们接着来看下一步工作：常量的定义。API 函数定义中 query 类型的 API 会在返回值中包含 result 这样的参数来返回一个代表处理结果的代码。比如，在 authsv 中命名为 PAID、UNPAID。如果在各个进程的源文件中定义它们的实际取值，每次需要修改一个值时就要追溯到所有相关的进程中进行修改，这为 bug 的产生提供了良机。

因此，我们将协议中使用到的常量定义在一个通用的头文件中（采用 C 语言时是类文件），该头文件被所有的进程共享。

比如，采用 C 语言的话定义如下。

```
typedef enum {
  PAID,         ←完成支付
  UNPAID,       ←未支付
  SUSPENDED,    ←
  ....
} auth_result_t;     ←表示authsv 协议所使用的值

typedef enum {
  SUCCESS,           ←成功
  NO_CHARACTER,      ←角色不存在
  ...
} db_result_t;
```

如果采用相同的编程语言来实现各服务器进程，比如说 C/C++，那么在所有的程序中 include 上述定义文件就可以了，但是实际上 cli 有时会用 Flash 来实现，对性能没什么要求的 commondbsv 会用 Python 来实现，各种情况都可能发生，对此，可以采取以下两种方法。

❶ **将实现 gmsv 的编程语言作为主要语言，对其进行语法解析，生成面向其他语言的源文件。**

❷ **使用某些 IDL，生成面向各进程实现语言的源文件。**

这种方法不依赖于其他特定语言，所以更为通用、灵活。

4.9.4　API 的调用时序

终于到了协议定义（详细）的最后一步——API 的调用时序了。C/S MMO 的开发和一般的游戏开发相同，在开始编程之后，游戏规范的更改是不可避免的，所以即使一开始就制定了完整的数据包时序图，维护成本也会很高，并不划算。但是通过对最低限度的模式进行确认，可以事先发现一些重大的问题。时序上的重大问题就是在着手开发之后，需要同时更改多个协议。我们的主要着眼点就是要避免这种情况。

在上一节“协议的实现原则”中，我们在协议的设计方面指出“尽可能做到无状态”。在 *K Online* 的协议规范中，后端服务器所实现的协议都是无状态的，记为“stateful”的只有 gmsv 协议和 msgsv 协议。

表 4.9 总结了后端服务器的协议规范。

表 4.9　后端服务器的协议规范

协议	one-way message	query	notification
gmsv protocol	○	○	○
loginsv protocol		○	
msgsv protocol	○	○	○
dbsv protoco	○	○	
worldsv protocol	○	○	

协议	one-way message	query	notification
commondbsv			
protocol	○	○	
authsv protocol		○	
logsv protocol	○		

后端服务器的协议都是仅由 one-way message 或者 query 构成的。通过像这样简化协议的基本性质，就可以简化之后绘制的时序图。

必要的时序图 ——关系到多个进程的典型处理是什么

时序图的绘制要达到何种程度呢？在 *K Online* 中，关系到多个进程的典型的事务处理有以下几处。

❶ 验证

❷ gmsv 中的角色创建*

❸ 登录 gmsv、msgsv

❹ 从 gmsv 登出

❺ gmsv 中的角色移动

❻ gmsv 中角色的商业操作（购物、交易）

❼ msgsv 中向好友列表添加、删除好友

❽ 向在线好友发送消息

如果编写程序之前对这些事务的时序图进行绘制并加以确认，可以避免以后发生一些难以更改的时序上的问题。

复杂性尽可能集中在网络的终端，也就是 cli 侧

终于要开始绘制各协议的时序图了，这里有一点很重要：复杂性要尽可能集中在网络的终端，也就是 cli 侧。

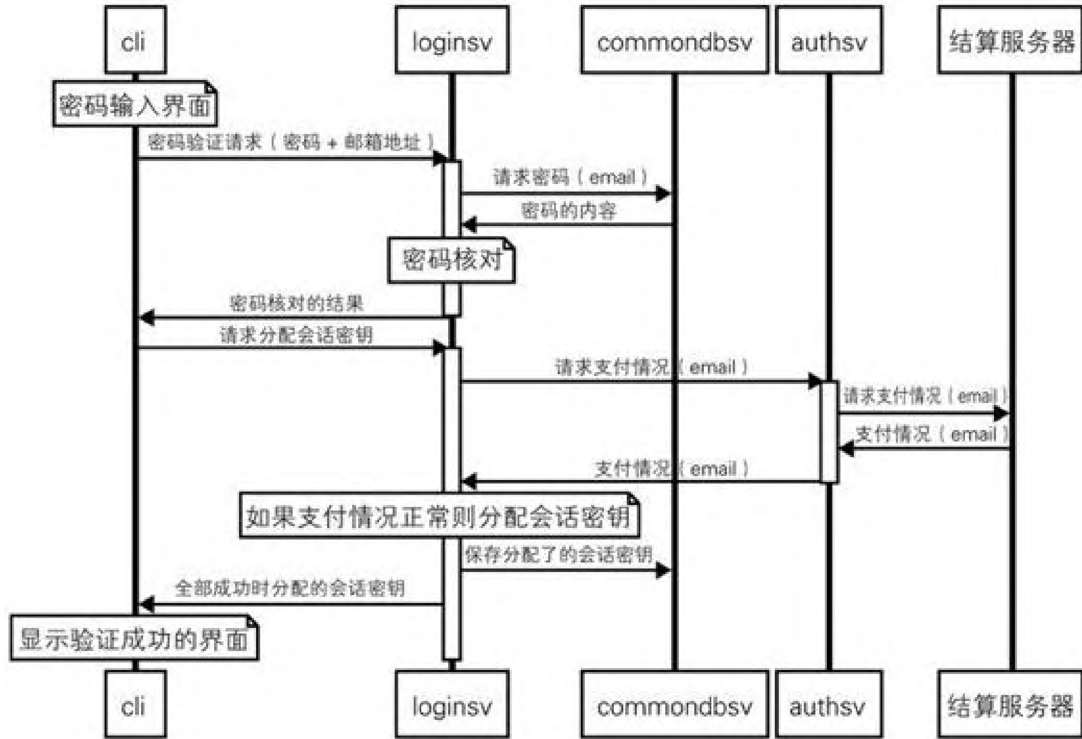
复杂性指的是条件分支和异常处理等根据情况的不同而有所差异的程序所特有的处理。后端服务器侧应该尽可能不依赖于上下文，只实现简单的功能。这样一来，之后需要修改处理内容时，就可以减少要修改的地方了。

❶ 验证

我们首先来制定 ❶ 验证的时序图。验证指的是所有前端服务器参与的游戏登录处理。为了登录游戏以获得各项游戏服务，玩家需要接受一系列必要的验证，在 *K Online* 中这就意味着登录 gmsv 和 msgsv 这两种服务器。

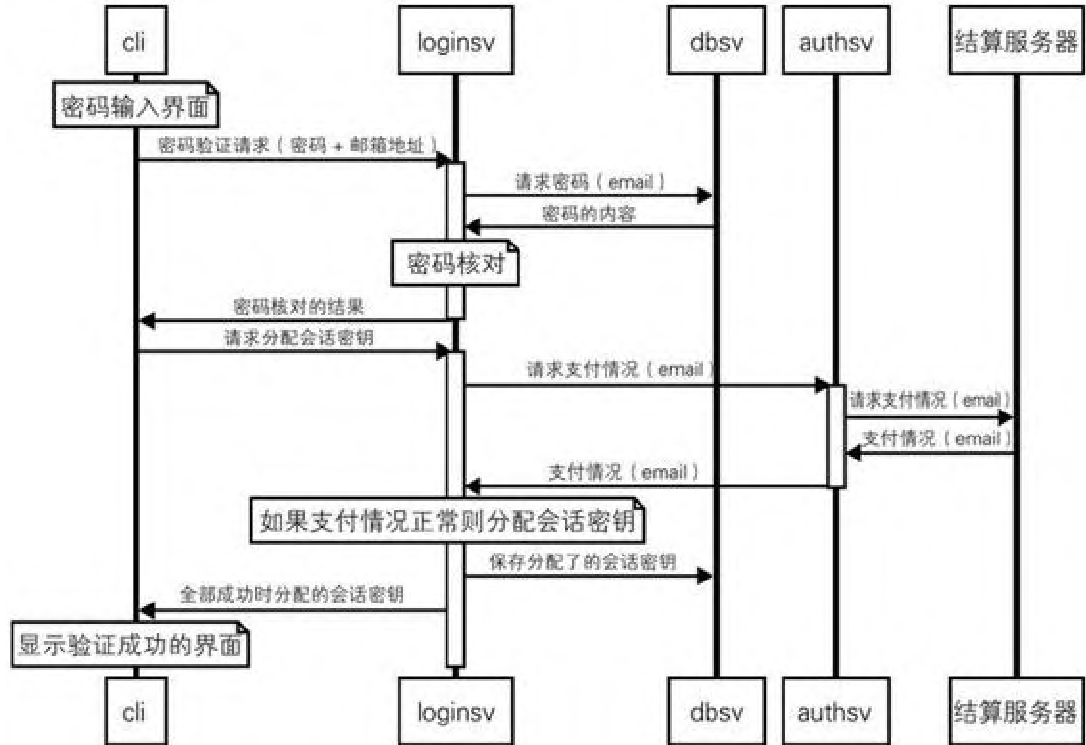
验证是通过 loginsv 来进行密码验证以及付费情况的确认。在图 4.15 所示的时序图中，两种验证都成功后，loginsv 就分配会话密钥，在之后对 gmsv 和 msgsv 进行访问时，使用这个会话密钥就可以访问了。*K Online* 采用了平行世界方式，所以有多个世界。为此，用户的 ID 和密码信息都保存在 commondbsv 中。

图 4.15　认证的时序图（采用 commondbsv 的设计）



在 *K Online* 中，为了在开始正式运营时就实现 3 万的同时在线数，我们以使用平行世界方式为前提设计了 commondbsv。在图 4.15 中也有 commondbsv，但是在服务规模并不大的游戏中，可以考虑由 dbsv 来实现 commondbsv 的功能。这种情况下的时序图如图 4.16 所示。

图 4.16　认证的时序图（由 dbsv 实现 commondbsv 功能的设计）

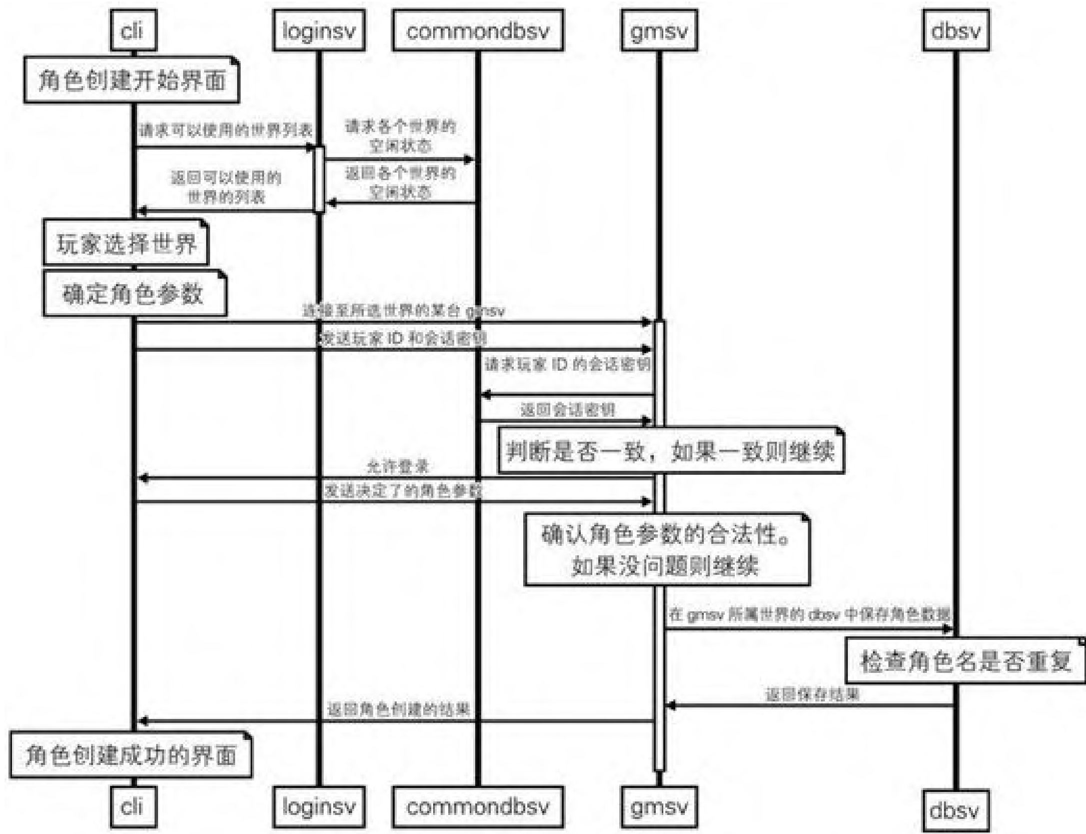


之后的说明都假设使用了图 4.15commondbsv 的模式。

❷ gmsv 中的角色创建

接着，我们来看一下 ❷ 角色创建的时序图。在 *K Online* 中，通过 ❶ 的验证而获得了会话密钥之后，就要开始创建角色了，成功之后登录 gmsv。其时序图如图 4.17 所示。

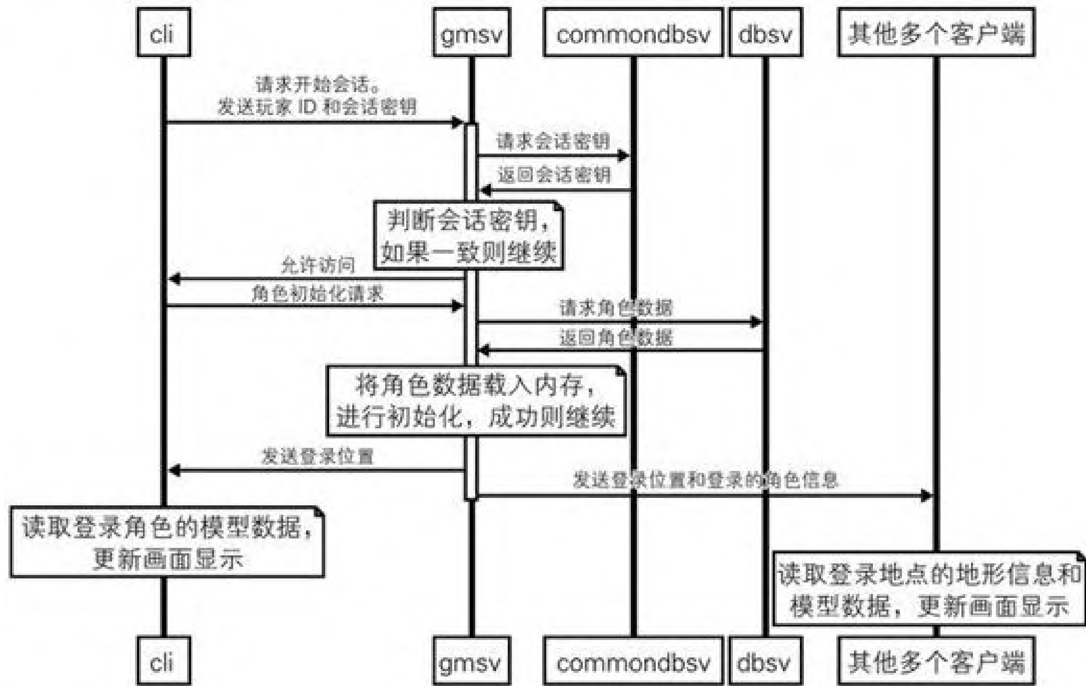
图 4.17　角色创建的时序图



❸ 登录 gmsv、msgsv

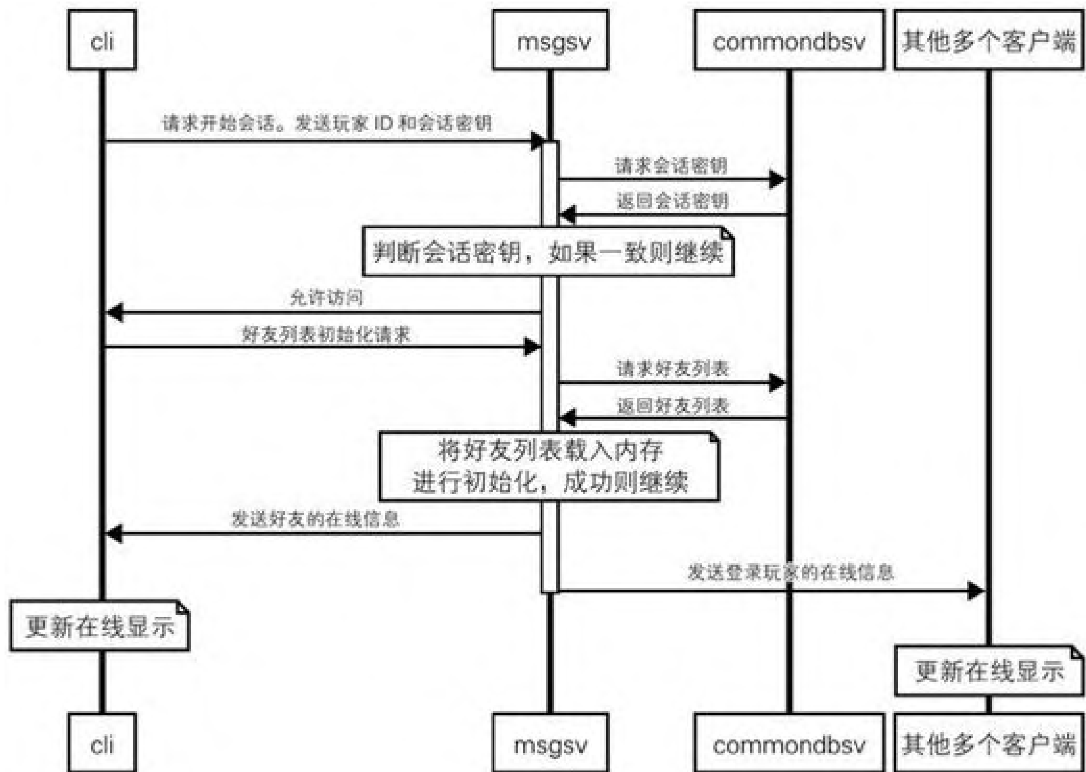
接着要登录 gmsv 的时序图（图 4.18）。

图 4.18　登录 gmsv 的时序图



成功登录 gmsv 后，也要登录 msgsv（图 4.19）。gmsv 和 msgsv 都登录之后，就可以访问所有的游戏功能了。

图 4.19　登录 msgsv 的时序图

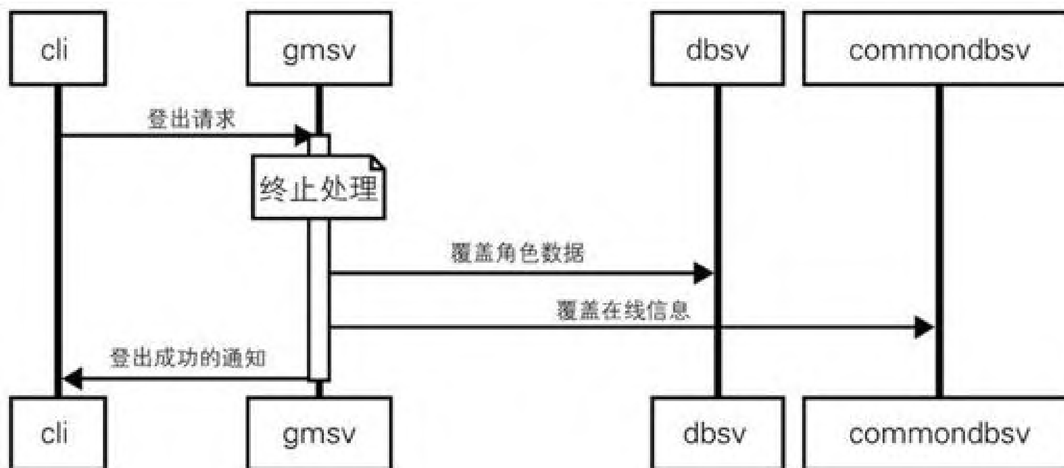


由于 msgsv 还需要好友列表的信息，所以要从 commondbsv 获取。

❹ 从 gmsv 登出

从 gmsv 登出的时序图如图 4.20 所示。

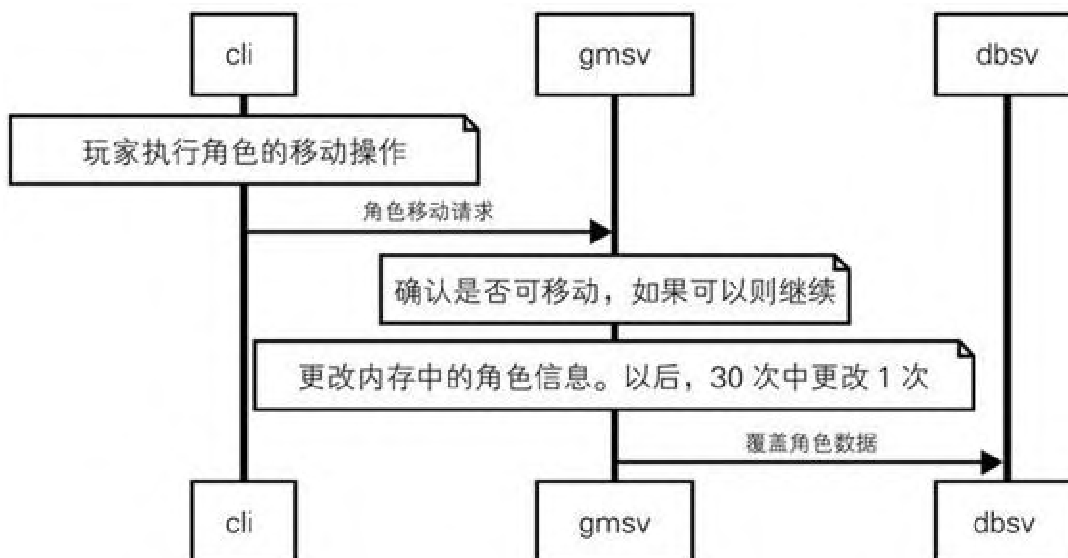
图 4.20　从 gmsv 登出的时序图



❺ gmsv 中的角色移动

角色在地图上移动时的 gmsv 的处理时序如图 4.21 所示。

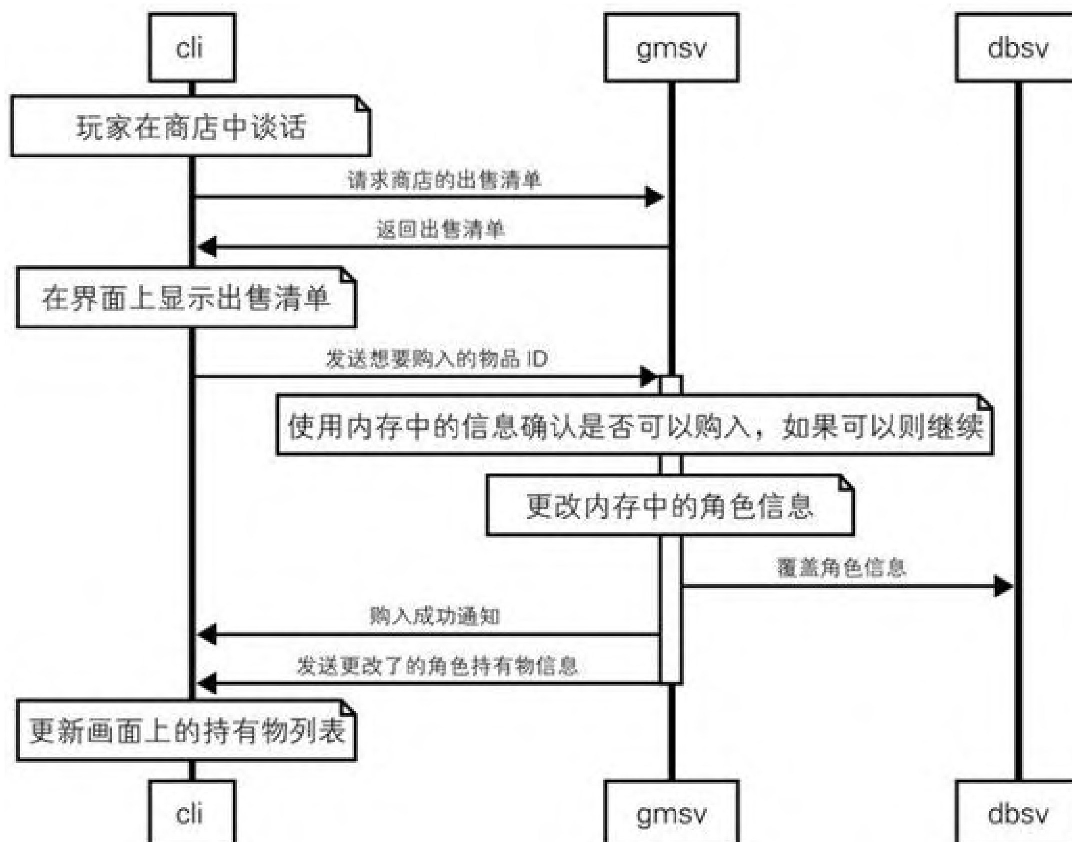
图 4.21　gmsv 中的角色移动



❻ gmsv 中角色的商业操作（购物、交易）

游戏内的角色在商店中购入物品时的时序如图 4.22 所示。

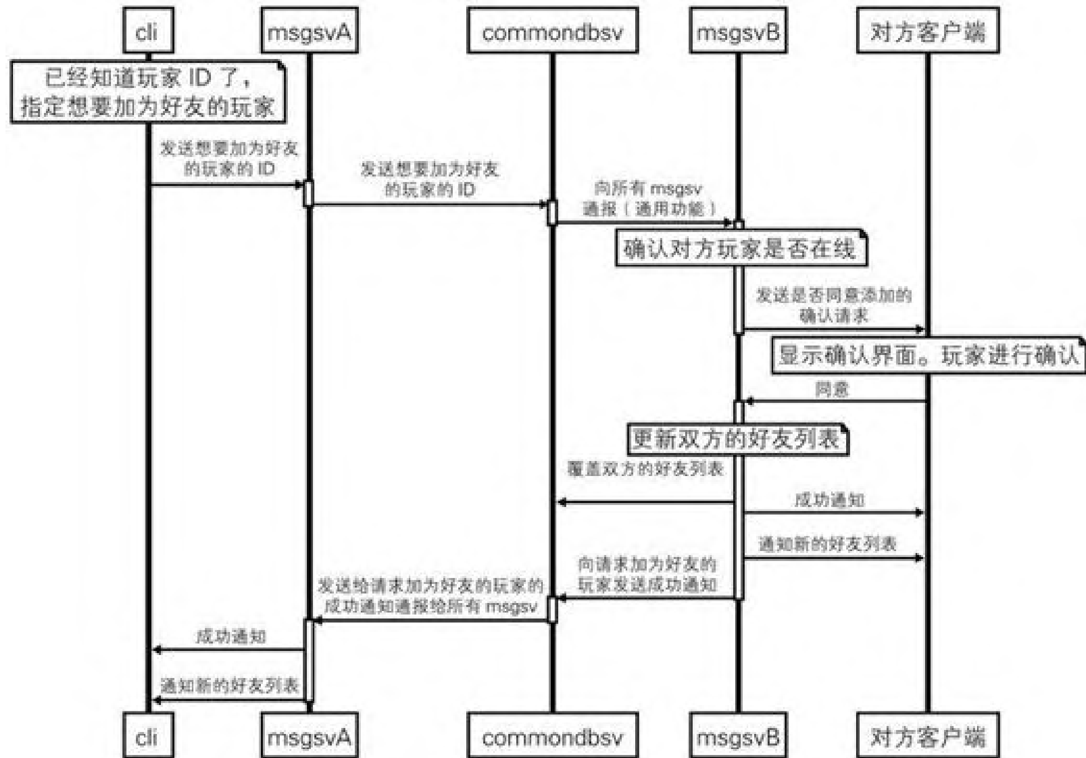
图 4.22　gmsv 中角色商业操作的时序图



❼ msgsv 中向好友列表添加、删除好友

接着是向好友列表中添加好友的时序图。在 *K Online* 中，无法添加不在线的好友（图 4.23）。

图 4.23　msgsv 中向好友列表添加、删除好友的时序图

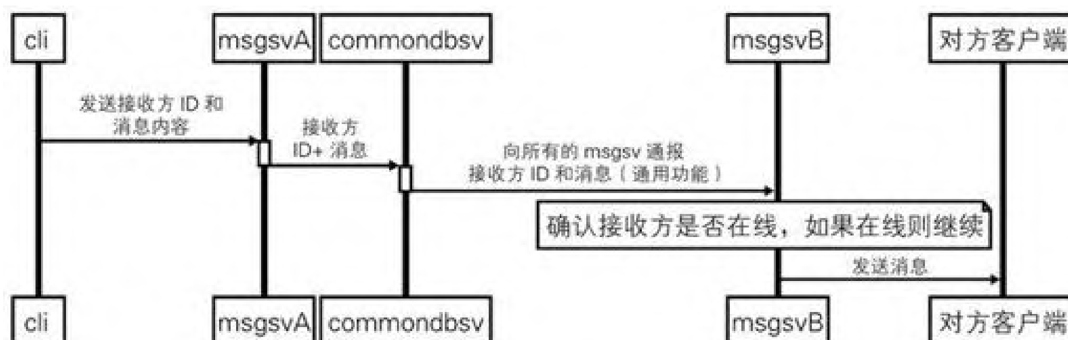


❼ 的时序有一个特征，那就是 commondbsv 实现了向所有 msgsv 进行通报的通用处理。msgsv 中 1 个进程可以处理 2000～3000 个同时连接，*K Online* 总共要处理 3 万个同时连接。也就是说，msgsv 需要使 10 个进程并行处理。在图 4.23 中，玩家 A（提出添加好友的要求的一方）连接至 msgsvA，玩家 B（被添加的一方）连接至 msgsvB。为了处理这种将 msgsv 分成了多个以提高并行度的情况，都要通过 commondbsv 来使所有的 msgsv 共享与添加玩家所相关的所有消息。

向在线好友发送消息

最后，向在线好友发送消息的时序图如图 4.24 所示。消息的发送也全部暂由 commondbsv 处理。这样可以保持系统整体的简单性，易于维护。

图 4.24　向在线好友发送消息的时序图



- **注意点**

建立如上所示的结构时，有两个方面令人担忧。

❶ commondbsv 是 SPOF（Single Point Of Failure[25] ）的

> [25] 指一个系统的一个部件如果失效或停止运转，将会导致整个系统不能工作。——译者注

❷ commondbsv 会成为性能上的瓶颈

上面这两个问题确实存在，但事实上并不会成为问题。首先关于 ❶ 这个方面，commondbsv 只实现一些通用的功能，并不实现特定于游戏的复杂逻辑，所以 commondbsv 比整个服务的其他部分更易维护。每周的例行维护之间，如果 gmsv 异常终止的概率为 1，那么将 commondbsv 的异常终止下降到 0.01 并不怎么难，所以将开发和设备的资源用在其他的地方更为明智。

与此相比，我们更担心 ❷ 性能上的瓶颈。如果每个玩家平均 60 秒向好友发送 1 次消息，在 *K Online* 中假设有 3 万的同时在线数，60 秒就有 3 万条消息，也就是每秒发送 500 条消息。commonsv 和 msgsv 之间采用 TCP/IP，通过专用的二进制协议进行永久连接，在这样的通信方

式下，大致每秒可以处理数千次查询，与此相比，每秒 500 次查询实在不算高负荷，所以不用担心性能上的问题。

如果 *K Online* 要支持更多的访问，比如，想要实现 30 万的同时连接就很接近这个上限了。虽然笔者至今尚未参与处理过如此庞大的访问量的服务器的开发，但是应该可以采用将 commondbsv 分割为多个、使用想要发送消息的玩家 ID 的 hash 值来使其分散等方法。

4.9.5 时序图制定的要点

上面我们制定了 8 种时序图，对一些典型的通信时序进行了简要介绍。

其要点就是，只有 gmsv、msgsv、loginsv 等前端服务器保持状态，并向作为主体的后端服务器和客户端收发各种各样的消息，而后端服务器则是被动应答。

这样，系统整体的复杂性就尽可能地集中在了前端服务器（终端）中。基本上只有前端服务器会使用内存中的信息并对其进行判断。

4.10 4协议定义文档——数据包的格式

至此我们已经讨论了协议定义文档所需的“协议的基本性质”和“协议的 API 规范（概要、详细）”。现在我们就来看一下 4.7 节开头所介绍的文档的 3 大要点以及“数据包的格式”。

4.10.1 C/S MMO 主要采用 TCP（复习）

我们首先来复习一下，C/S MMO 采用的是 IPv4（请参照第 0 章）。IPv4 上所实现的传输层协议中，在网络游戏中可以使用的有 UDP 和 TCP，C/S MMO 中主要使用 TCP[26] 。

> [26] UDP 更能降低 CPU 负荷，所以在 Linux 内核 2.2 的时候使用了 UDP，但是现在这也是优化过的，基本不会通过 UDP 获得多大好处，所以在 C/S MMO 中基本不用（其他形式的游戏会用到）。

4.10.2 C/S MMO 使用包含专用字节数组的二进制协议

TCP 是一种流式协议，理论上数据包是连续的，不存在间隔。因此当调用了两次 RPC 时，为了在数据接收侧判断到底调用了 1 次还是两次，需要有一些规则来分隔各个数据包。

在 Web 和电子邮件等通用系统中，通常使用以 HTTP 为首的文本协议，这类协议使用换行记号来分隔记录，但是在 C/S MMO 中，需要实现“处理负荷较低”、“耐得住更改和攻击”、“要有良好的开发效率”。为了满足这些条件，除非特殊的情况，否则都使用“包含专用字节数组的二进制协议”。这是因为，由于 C/S MMO 的服务器中每秒几千次的 RPC 调用都是从 cli 接收的，为了进行庞大的文本处理，需要避免过多使用 CPU。

4.10.3　二进制协议的实现——首先从术语的整理开始

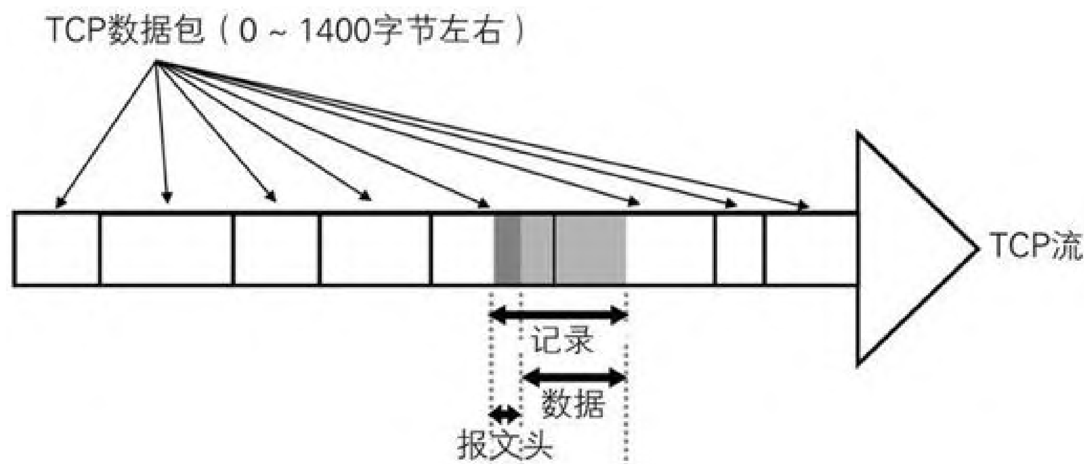
那么，我们就来看一下如何实现二进制协议。通常，与 IP 和 TCP 等通用协议相同，采用交替发送固定长度的报文头和可变长度的数据的形式。报文头中保存了数据部分的长度等元信息（参见图 4.25）。下文中，我们将 1 组的报文头和数据部分称为“记录”。我们首先以图示的方式来整理一下定义数据包格式时用到的一些必要的术语之间的关系（参见图 4.26）。

图 4.25　记录的内容

“TCP 流”由一系列 0 字节～1400 字节左右的不定长的“TCP 数据包”组成。根据途中经过的路由、操作系统的设置等，流的长度会发生变化。通常最大长度为 1460 字节。之后所说的“数据包”指的就是 TCP 的数据包。

我们接着来看一下 TCP 数据包的内部，TCP 数据包的边界指的是先前所述的由报文头和数据组成的记录。图 4.26 中只标示了 1 个记录，但事实上，流中各记录之间是没有间隔的，全都紧密相连。

图 4.26　记录、TCP 数据包、TCP 流的关系



记录的大小

在 C/S MMO 中，1 个 RPC 参数并不需要这么庞大的数据。这是因为虽然用户每次进行某些操作时都会调用某些 RPC，但是用户的操作基本上就是从鼠标和键盘进行输入，虽然操作频率很高，但是字节数并不多，充其量也就数十字节的程度 [27] 。

> [27] 但是，语音聊天和截图发送则例外。有关语音聊天的内容将在 6.2 节介绍，请参照该节。截图的情况下，瞬间就会产生数百千字节的数据。目前，这并不会在 gmsv 协议或 msgsv 协议中实现，而是完全独立地实现专用的 Web API（比如 Flickr 的 Web API），使用该 API 来发送。这并不是网络游戏特有的技术，所以本书就不作介绍了。

C/S MMO 中各个记录的长度也就数十至数百字节，虽然很短，但是发送频率很高，综合下来，传输量也不小。实际的传输量就如“必要的带宽估算”（4.6 节）中所述的那样。

报文头

下面来考虑以下实际的数据内容。毫无疑问，报文头还是短一点比较好。为了保存数十至数百字节的数据部分的信息，最少需要 2 个字节（16 位）。如果对 *K Online* 中的 传输内容进行严格检查，或许用不了 16 位，只要 13 位就可以了，这样可以节约 3 位，但是考虑到误差范围，还是以代码的简单性为优而采用 16 位。

数据部分的压缩和加密

对于数据部分的内容，为了控制带宽成本，必须对其进行压缩，而为了防御攻击，必须对其进行加密。

如果使用了合适的压缩算法，可以将从 gmsv 到 cli 的传输量降低 50%。虽然数据包内容的压缩会对 CPU 造成一定的负荷，但是比起带宽成本，服务器的 CPU 成本还是很低的，所以在商业上的优势非常大。然而，从 cli 到 gmsv 的传输不会重复发送同样的数据，基本上无法对压缩效果进行估算，所以就算不压缩也没关系。在数据包中设置两个字节的报文头来存储压缩后的大小，适时地将数据传递给解压缩程序。

接着，为了提高安全性，需要对数据部分加密。其目的是在数据包的传输过程中，对来自怀有恶意的第三方的入侵（称为 Man in the middle 攻击，中间人攻击）进行基本的防御。这里说“基本”是因为，在知道了在 cli 和 gmsv 中实现的加密算法后，从开始 TCP 会话起，除非拦截所有的数据包，否则无法得知数据包中的内容。

一般来讲，肯定是通信链路越安全越好。但是，加密程序会给 CPU 带来很高的处理负荷，而且对用户来说，每次输入密码也很不方便，所以也有缺点。作为娱乐性质的服务，不仅支付不了这么巨大的成本，还给玩家带来了很大的麻烦。因此，公认为最可靠的电子签名的机制并不符合 C/S MMO 的要求。在很多情况下，使用 RSA 和 Diffie-Hellman 等密钥交换方式来共享密钥，然后在会话存在期间持续使用该密钥，这样可以大幅降低 Man-in-the-middle 攻击的风险（完全消除是不可能的），而不管在 CPU 成本方面，还是在给玩家带来的体验方面，在 C/S MMO 中都还不错。

实现上的要领

综上，通过 RPC 发送的数据被序列化之后，对其所生成的字节序列，需要一个用来对其进行压缩的层和一个用来加密的层。

至此所考虑的报文头的最小长度为两个字节。但是在从 cli 到 gmsv 的通信中不需要压缩，而前端服务器和后端服务器等数据中心内部的传输则不需要加密，即使是在一个 C/S MMO 系统内部，对通信链路的要求也各不相同，为了充分实现用来生成各个进程之间的记录内容的程序，建议对加密层和压缩层进行划分，使其可以交换。

比如，划分成一个具有两个字节报文头的压缩层和一个具有两个字节报文头的加密层，虽然这样一来原本两个字节的报文头就变成了 4 个字节，但这么做就可以用很简洁的方式来实现加密功能的去除、压缩功能

的去除、对单个功能进行测试以及进行性能测试等。典型的针对 C/S MMO 的通信中间件就是这么实现的。

图 4.27 中所示的记录中具有两个报文头，其最内侧保存了序列化之后的数据。此外，为了提高压缩率，加密一定要在压缩完成之后进行，这一点请务必注意。

图 4.27　加密 / 压缩之后的记录

对数据包格式的讨论就到此为止了。

专栏　C/S MMO 的压缩和加密

包括 OSI 参考模型中的物理层在内，本文所提到的 RPC 序列化层的层次结构如下。

- RPC 序列化（序列化通过 RPC 发送的数据）层：将 RPC 转换为二进制数据。
- 压缩层：对序列化之后的数据列进行压缩。
- 加密层：对压缩之后的数据进行加密。
- TCP：将加密后的数据放在数据流中。
- IP（IPv4）：将数据流放在数据报中，在 cli 和 gmsv 之间传递。
- 以太网（等）：在链路中的各设备之间传递数据。
- 物理层，在链路中，传递电信号和光信号。

从上到下依次对应着层次的从高到低。在 C/S MMO 中，由于压缩和加密功能的增加，TCP 上面增加了两个处理层。

4.11 5数据库设计图

本章提出的 5 种设计文档的最后一种就是“数据库设计图”了。在同时在线数众多、累积性很高的某些 C/S MMO 中，用于数据持久化的数据库设计非常重要。我们首先要牢牢掌握其基础。

4.11.1 要在编程之前进行对重要的表进行设计

在 C/S MMO 中，良好的数据库设计就意味着被持久化的信息的结构非常清晰，能够高效检索所需的信息。这样运营和管理工作也就更容易进行了。比起持久化信息较少的其他形式的网络游戏，数据库设计在 C/S MMO 中格外重要。

在开始编程之前，不管是用 ER 图这样的表设计文档，还是用 Excel 制作的表格，都可以用来设计表结构，理想情况下，保存着重要信息的数据库表要在编程开始之前进行设计。因为通过在设计时定义表结构，可以找到更好的实现方法。

K Online 中所需的表应该如何设计呢？在此之前，我们先来追溯一下 C/S MMO 游戏中数据库实现的历史变迁。

4.11.2 C/S MMO 中的数据库实现的历史变迁

C/S MMO 中游戏数据的持久化方法随着时代不断发生改变。作为整体的趋势，由于计算机性能的提高，以及软件技术的发展，更具结构化的语义信息得以丰富，其形式也更为易用了。这种变化与 Web 电子公告牌服务（BBS）的实现方式的变化很相似。

20 世纪 70～80 年代：没有数据持久化，复活[28]咒语

> [28] 在密码保存时使用

20 世纪 70～80 年代，MUD（Multiple User Dimension）被开发出来，当时是所谓的C/S MMO 游戏的黎明时期，那时还没有数据持久化。数据只存储在运行在服务器上的进程中，如果服务器进程因为某些原因而终止，游戏记录就会丢失。如果游戏时间短，这也没什么问题，但是如果游戏时间长，就要设法对数据进行持久化。80 年代的家用机游戏中设计了一种叫做“复活咒语”的持久化方法，但是在网络游戏中，这种方法面对作弊行为就显得比较薄弱了，恐怕无法使用。

20 世纪 90 年代：保存在文件中

到了 20 世纪 90 年代，所有想要持久化的信息都以文件形式存放在操作系统的文件系统中，使用标准文件 I/O 的系统调用来直接保存。同时期的 Web 公告牌服务也是同样，1 个文件通常以 1 个线程的形式来保存。

那个时期的文件内容都是二进制的数据，这些二进制数据是通过某些方法将角色数据的内存转储[29]，或者转换成文本数据的内容进行序列化而得到的。不管什么游戏都会单独设计一种形式来加以使用。这种方式有一定的优点，现在也有使用，所以这里简单介绍一下。

> [29] 内存转储是用于系统崩溃时，将内存中的数据转储保存在转储文件中，供有关人员进行排错分析用途。而它所保存生成的文件就叫做内存转储文件。——译者注

- 保存到文本文件中的方法

首先，当时的游戏信息中只包含玩家角色信息。C 语言风格的定义如下所示。

```
struct Item {
  unsigned char typeID;
  unsigned char num;
};   ← 2 个字节

struct Location {
  unsigned int x, y;
  unsigned int fieldID;
};   ← 12 个字节

struct Character {
  char password[20];
  unsigned int hitPoint, maxHitPoint;
  unsigned int magicPoint, maxMagicPoint;
  Location currentLocation;
  Item inventory[10];
};   ← 20+8+8+12+2*10=48 个字节
```

Item 是玩家所持有的物品，由物品类型 ID（typeID）和持有数量 num 组成。比如，假设“药草”的 typeID 为 2，持有 4 个，就是 [2,

4]。2 个 unsigned char 就是 2 个字节。Location 是玩家在世界中的位置信息，3 个 unsigned int 类型的变量，一共 12 个字节。

Character 类型中包含 HP、MP、登录密码等信息，总共 48 个字节。

将这些信息像下面这样保存在文本文件中。

```
item 2 1
item 11 1
location 110 240 91
maxHitPoint 12
item 3 20
item 6 2
password hogehoge
magicPoint 20
item 9 3
item 7 1
hitPoint 10
maxMagicPoint 20
```

这里以易于人们阅读的形式来保存是因为，能够使用通用的工具来提供用户支持。不管是通过远程文件复制，还是其他方式，只要有编辑器，就能很简单地进行损坏数据的修正和检索等基本事务。

此外，文本文件不依赖于机器的字节顺序，也不依赖于编译器在内存中排列结构体成员变量的方式，所以在对服务器操作系统进行更新以及之后修改程序时，系统可以很健壮。

将上面的内容保存在文件系统中。比如 /var/gamedata/character/ringo。ringo 是角色名，如果玩家以 ringo 名登录就会访问该文件，如果文件不存在，则创建一个新的角色，如果存在，就继续游戏。此外，如果一个目录下保存过多文件就会出现问题，所以一般使用玩家名等信息来进行横向分散。

这个系统非常简单，也不容易发生 bug，但是如果数据变大、超过几十千字节，就会出现读写速度方面的问题。

本世纪初前期～：RDBMS

好了，我们继续回到历史中。到了 21 世纪初前期，MySQL 和 PostgreSQL 等可供廉价使用的 RDBMS 的稳定性得到了加强，原本保存在文件中的数据可以原封不动存入 RDBMS 的 BLOB 类型的记录中。存入 BLOB 时，在数据库表的列中复制角色名等相当一小部分的信息，冗余地进行写入，通过尽可能降低检索而使游戏的运营工作更为高效。这种方法下，多个角色的表的数量和关系的数量也有多个。

在 2005 年左右，角色数据和所持物品列表等核心信息继续使用 BLOB 的形式来保存，好友列表和消息等其他信息通过利用 RDBMS 的 SQL 功能来实现。此时表的数量达到了 10～20 个。

如今，核心信息也通过使用 RDMS 的 SQL 功能来实现了。表的数量超过了 50～100 个，字段数则超过了几千个。由于数据库的利用度提高了，从而需要使用对象关系映射（Object-Relational Mapper）。另外，服务器的内存也便宜了，可以安装数吉字节的物理内存，大部分游戏信息可以在内存中缓存，性能上的瓶颈大幅扩大，实现难度降低了。

以上，通过 C/S MMO 中数据库实现的历史变迁我们可以知道，从数据库层面来看，游戏发生了很大变化。

4.11.3 整理 *K Online* 所需的表

现在我们开始着手整理 *K Online* 所需要的一些表。考虑到运营和可扩展性，尽可能充分利用 RDBMS，通过结构化的状态在 RDB 中保存数据。

这里所说的“结构化的状态”指的是数据本身与其内容相关，以及大量持有结构信息。具体来说就是，数据是基于规则来保存的，要尽量使规则更为严密以便以后易于使用。规则越是适用，用来保存信息所需的的内存和 HDD 的容量以及 CPU 使用量就越会增加，所以需要通过设计工作来找到其平衡点。

用于 C/S MMO 的数据库，需要怎样的表结构呢？这里首先列举一些典型的模式。

根据 mmogchart.com[30] ，C/S MMO 游戏中的类型 95% 以上都是 MMORPG，剩下的 5% 中有一小部分是 MMOFPS，还有的就是模拟类和解谜类游戏。模拟类和解谜类游戏基本可以用基于 Web 的系统来实现，这类游戏的内容缺乏实时性。因此，在 C/S MMO 中，需要用到本章所述技术的基本上都是 MMORPG 和 MMOFPS。MMOFPS 的基本游戏内容是角色培

养，多名玩家共同对敌，在这一点上沿袭了 MMORPG 的内容，同时 MMOFPS 的实时性、动作性更高，以第一人称视角展开，操作多样化，所以这两者在数据库的设计方面基本相通。

> [30] http://www.mmogchart.com/Chart8.html

因此，如果掌握了 MMORPG 中的模式，那么可以说就基本覆盖了 C/S MMO 中典型的数据库设计模式。本章的示例 *K Online* 就是一个典型的 MMORPG，所以接下来我们就通过这个示例游戏来看一下典型的表结构。

专栏　百花缭乱的 KVS——未来 C/S MMO 中数据库的使用情况

现在，距离 RDBMS 的使用已经过了好几年，在今后的 MMO 开发中，作为 RDBMS 之外的另一种选择，KVS（Key-Value Store，键值存储）开始崭露头角。

KVS 并不是通过 Scheme 在结构化定义的表中存储数据的，而是通过“键”（Key）、“值”（Value）对来保存信息的数据库。“键”指的是 128 位的整数或者短字符串等，“值”指的是角色数据或 Wiki 文档等数据量很大的数据主体。

KVS 的广泛使用原本是为了实现以 Web 服务的 memcached[31] 为代表的“内存缓存”。KVS 以特定的数据库行（比如玩家 ID）作为键，或者将 SQL 查询本身作为键来缓存，这些 Web 服务所需的数据缓存处理位于应用服务器和数据库服务器之间，为了灵活实现而被使用。

> [31] Memcached 是一个高性能的分布式内存对象缓存系统，用于动态 Web 应用以减轻数据库负载。——译者注

用来实现 KVS 的服务器软件中，笔者用过 MongoDB ，此外 Tokyo Tyrant 、Membase 等各种功能各异的开源产品也不断被开发出来，在商业服务方面不断创下佳绩，可谓百花缭乱。

这里以笔者使用过的 MongoDB 为例，介绍一下 KVS 的典型特征，以及它和 RDBMS 之间的区别。

- **无模式**（Schema-less）

不需要定义表结构。在 RDBMS 中，在执行 create table 时指定表中应具备的字段，如果之后指定了没有包含在其中的字段，就无法进行 Insert 等操作，而在 MongoDB 中，这是很自由的。此外，哪个字段作为索引也可以自由指定。也就是说，可以自由选择多个作为键的字段。

如果使用无模式的表，就不需要在改变表结构时重新对其进行定义了，从而提高了初期开发效率。而在 RDMS 中，必须对表中残留的不需要的字段，或者应用程序侧不存在的字段进行处理。就笔者来看，RDBMS 中表结构的改变非常麻烦，所以如果字段很多，最后就会留下很多垃圾字段，如果实现某些 O/R 映射层，最后可以实现对 NULL 值的处理，所以这一点不会成为特别大的问题。

- **只有简单的查询**

  只能进行 Find（相当于 select）、Insert、Update、Remove（相当于 delete）操作。没有联合查询功能，这一点是 KVS 的特征。正是由于这一点，MongoDB 可以自动横向分割数据库，从而很简单地就能实现横向扩展的“数据拆分”（Sharding）。在 C/S MMO 中能实现简单的查询就已经足够了，所以这个限制完全不成问题。

- **高速**

  在执行简单查询的情况下，如果 MySQL 在某台机器上可以发挥出每秒 1 万次查询的性能，那么 MongoDB 通常可以更快，达到每秒 2 万～3 万以上。TokyoTyrant 等似乎也能实现这样的速度。这是因为 MongoDB 和应用程序之间使用的是简化了的二进制协议，另外，也因为经常使用 MySQL 中的 HandlerSocket plugin[32] ，所以在服务器侧不需要编译 SQL。

- **不能使用事务功能**

  之后本章所要说明的 C/S MMO 不使用事务功能，所以也不需要。但是在收费和验证等辅助系统中，也有需要用到的时候。比起将所有的服务都转移到 KVS，根据用途来分开使用更好。存储过程等其他的一些复杂功能也没有，同样也是根据用途来分开使用。

[32] 实现非 SQL 的处理的 MySQL 的接口的一种。以下文档对其进行了详细说明。 请参考：http://www.slideshare.net/akirahiguchi/handlersocket-plugin-for-mysql-4664154

如上所述，KVS 作为一种“简单、高速的数据存储引擎”是非常优秀的，出于保存游戏数据的目的，在 C/S MMO 开发中，今后会频繁用到。

但是就目前来说，角色数据和根本的数据保存方式还是继续使用 MySQL 和 PostgreSQL 等成熟的系统，图片和游戏日志等附加信息则保存在 KVS 中，这种根据用途同时使用两种存储方式的情况还会继续下去。

此外，笔者现在参与的项目中，系统运营公司提出了“KVS 的优势尚不明显，最好还是先不要使用”的要求。在出现 2～3 种社会中广泛使用的事实标准之前，这种情况看来还会持续下去。

需要持久化的信息以及数据的包含关系

K Online 中需要持久化的信息如表 4.10 所示。在对策划内容进行确认的同时，记录下需要持久化的要素的特性和内容。在将一些项列在表 4.10 中时，还需要将它们对应到数据库的表中。

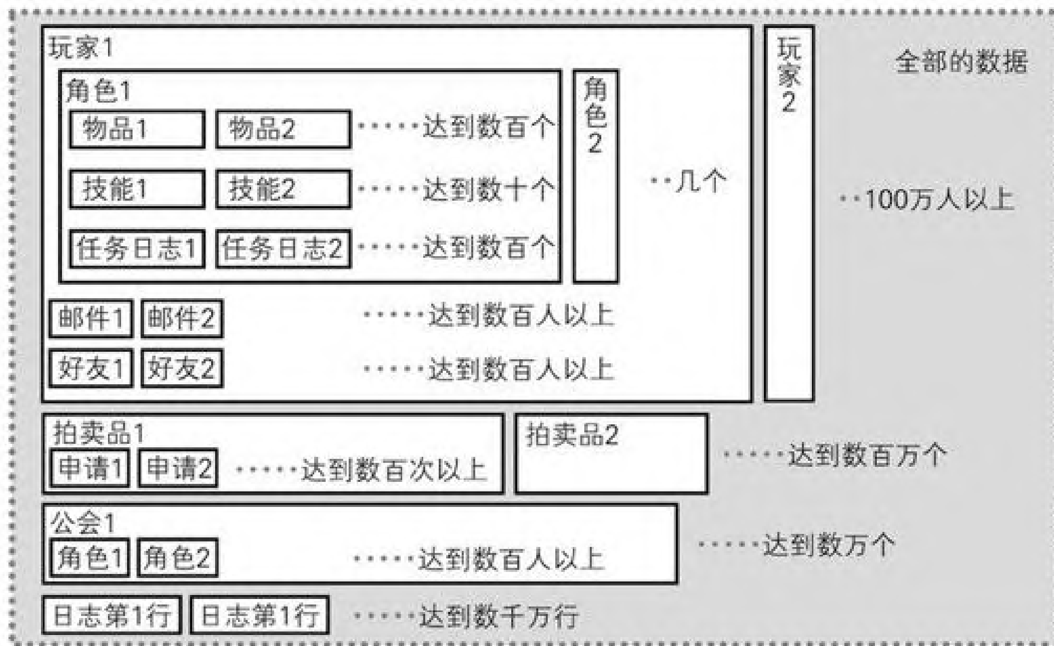
表 4.10 中所示的这些持久化数据的包含关系如图 4.28 所示。

表 4.10　*K Online* 中需要持久化的信息

信息	内容	持久化的必要性
用户	ID 和密码、计费方式等对应于 1 名用户（玩家）的信息集	对于同一个用户，需要持续进行对相同内容的访问控制
角色	游戏进行空间中 1 个玩家角色的体力、经验值、金钱、当前位置、名字等当前状态的集合。*K Online* 中每个用户可以有多个角色	登出之后，下次登录时，需要在同一个位置，以相同的状态开始游戏
物品	角色所拥有的物品列表。*K Online* 对持有物品的数量没有限制，但对其总共的重量有所限制。一部分物品可以装备在角色身上，对角色的状态产生各种影响。物品的持有数根据玩家的情况会有很大的差异，而且数量的变化也很厉害，此外又想从多个玩家的所持物品中进行查询，所以将这些内容保存在另一个表中	登出之后，下次登录时，需要持有同样的物品进行游戏。另外，在查询由多名玩家组成的队伍中所持有的物品，以及在实现物品交易功能时，需要高效利用 DBMS 的查询功能

信息	内容	持久化的必要性
技能	管理角色能力中的制作弓箭等技能。每个玩家所具有的技能数和内容都有所不同，所以保存在另一个表中	登出之后，下次登录时，需要以同样的技能进行游戏。此外，技能等级要在 Web 网站上显示出来
任务日志	保存分配给玩家的任务。将当前玩家角色正在进行的任务和已完成的任务信息作为日志保存	任务需要在登录之后重新开启。此外，任务的种类达到数百以上，而且各个任务还需要中间过程的状态，每个玩家的任务记录量大不相同，所以创建另外的表来保存
拍卖	谁都可以竞拍，出价最高者得到物品。将拍卖物品和竞拍价格保存在表中	一次交易会持续几天，需要在各个玩家登录、登出以及游戏服务器重启时调出
好友列表	玩家和玩家之间是否存在好友关系	再次登录之后需要调出相同的好友列表。此外，因为每个玩家的好友数差异很大，所以需要单独的表
邮件	即使某个玩家处于登出状态，也要能向该玩家发送必须送达的消息	只要玩家没有明确删除，就需要一直保留着。此外，每个玩家保存的邮件数差异很大
公会	玩家可以属于多个团队，这种团队称为公会。保存哪个玩家属于哪个团队的信息	再次登录后，需要保持所参加的团队的信息
日志	某个玩家采取了什么行动的详细的文本记录。数据库的每 1 行中写入 1 行文本日志	用户支持、异常的验证、调试、统计数据的获得等日志以后都要用到

图 4.28　需要持久化的数据的包含关系



整个游戏中有玩家、拍卖物品、公会、日志，100 万人以上的玩家，每人都拥有多个角色，每个角色拥有多项物品、技能、任务日志。每个玩家都有自己的邮件和好友。出售到拍卖行的物品包含一个竞价列表，公会包含多个角色。

如果游戏规则为“角色持有的物品数最多为 10 个”、“每个玩家只能创建 1 个角色”、“每个玩家同时只能拍卖 1 件物品”、“好友列表最多可以添加 20 名好友”、“邮件中最多能保存 50 条消息，自动删除先前的消息”，那么这些就不需要作为单独的表来保存了。

- **保存可变长度的数据的必要性的提高**

看一下图 4.28 就会了解保存可变长度的数据的必要性。保存可变长度的数据越来越有必要，笔者认为这有两个原因，这里简单地介绍一下。

首先，现在的游戏中，每个玩家的游戏风格都非常多样，想要尽可能适应极端情况就会有所影响吧。

第二，MMORPG 总是希望能尽可能运营多年，开始游戏 1 天的玩家和玩了 1 年的玩家在游戏过程中积累的信息量肯定会有很大的差异。

数据的特性和各个表的准备

以上整理了 *K Online* 中看起来需要的表。上面列出的总共有 10 个，但是对于拍卖功能，拍卖的物品和竞拍价格需要在不同的表中保存，或许最后会超过 10 个表。

当然，表的数量越少越易于管理。就像 20 世纪 90 年代所实行的那样，对于每个玩家角色，在 1 个文件中以二进制数据的方式保存角色数据的结构体，这种方法是最简单的。

但是在 *K Online* 中，与角色相关的信息的条目非常多 [33] ，如果保存在二进制文件中，1 个角色多达数兆字节的信息每次都需要在内存中加以组织，这样服务器的处理时间就会过长。比如，1000 个 100 字节的任务数据、1000 个 300 字节的邮件、200 个 24 字节的好友信息、200 个 100 字节的物品信息、50 个 32 字节的技能信息、20 个拍卖中的竞拍信息……等大量的信息种类，每次集中在文件中进行读写，这项处理太过耗时。为了避免这种情况，考虑分类保存，这样最终就产生了保存在 RDBMS 表中的解决方法。

> [33] 随着条目的增加，数据的字节数也会多起来。

一般来讲，具有以下特性的数据，通过另外创建 DBMS 表来保存是很有效的。

- **相关信息的数量对于每个玩家来说大不相同的情况下**

  比如，平均持有的物品数为 10，持有量多的玩家拥有 1000 个，而持有量少的玩家只有 1 个，其差异达到了几百倍。

- **条目数不断增加的情况**

  比如，所持物品平均有 10 个，使用后就会减少，获得后就会增加，而任务的完成不会减少，只会不断增加。

对于不具备这些特性的信息，在保存角色数据的表中进行保存和管理就能达到要求了。

4.11.4　数据库性能预测

至此，我们已经大致对需要什么样的表进行了设计，接着我们就对其性能进行预测。对于各个表，需要从以下几个方面进行考虑：一开始就要

对实际执行的查询作出一定程度的估计。一开始就有需要负载平衡的要素吗？或者，将来需要追加负载平衡的措施吗？如果一开始就考虑进行负载平衡，那就需要在最初的设计中考虑具体要采取的方法。

运营开始之后再更改数据库的表结构基本上是不可能的，所以至少也应该做到这样的设计：即使不改变表结构和索引、ID 的处理方式，也能在将来获得良好的性能。

当然，在开发的过程中，可以通过使用测试数据和实际开发的服务器来进行负载测试，以此来判断是否能够承受实际的负荷，但是在项目刚开始时，无法进行这样的实测。最初的估算是非常重要的。

说到负荷，登录用户数 / 同时连接数是首要的线索。*K Online* 的销售战略是在最初的 1 年里分阶段展开宣传，计划用 1 年时间实现 3 万个同时连接，这是很典型的一个计划。

在能够免费试玩的游戏中，同时连接数为 3 万的话，估计需要应付 30 万～100 万的登录用户。

数据库的处理性能及对其的预测

数据库的处理性能大致与表中存储的数据量（行数）的对数成正比，与访问频率的 1 倍成正比。此外，一般来讲，数据库写入操作方面的性能是很难提高的，而读取操作的性能则相对比较容易提升。因此，一开始只要在数据量和访问频率方面，将读取操作和写入操作分开预测就足够了。

对数据库的查询大致分为以下几种。

- **类型 1：指定主键，只取出 1 行的查询。**
- **类型 2：使用包含主键在内的 1 个以上的索引，以一定的条件（比如两个值之间的范围等）来检索、排序的查询。**
- **类型 3：不使用索引的查询。**

从数据库处理性能的观点来看，基本上不要采用类型 3 这种方式，类型 2 也应该控制在最小程度，大部分由类型 1 构成是最理想的。

表 4.11 中针对各个表，以运营开始 1 年后达到 100 万登录用户、3 万同时连接为前提，进行了预测。

表 4.11　各个表的特性及预测

表	100 万登录用户时的行数	read 访问模式	write 访问模式	1 行的大小	合计大小
用户	1M	登录时只读取 1 次。10分钟 1 次（类型1）。3 万/ 60 / 10 = 50 次/ 秒	密码变更时进行 write。大多数情况下不需要	100 字节	1 × 100 字节 = 100 兆字节
角色	1M	登录时只读取 1 次。10 分钟1 次（ 类型1）。3 万/ 60 / 10 = 50 次/ 秒	1 分钟 1 次 update（500 次/ 秒）	10 千字节	1 M × 10 千字节 = 10 吉字节
物品	1M × 100（最大持有数）	登录时只读取 1 次。10 分钟1 次（类型1）。3 万/ 60 / 10 = 50 次/ 秒	1 分钟 1 次对 1 行进行 insert/update（500 次/秒）	100 字节	1M × 100 × 100 字节 = 10 吉字节
技能	1M × 100（最大持有数）	登录时只读取 1 次。10 分钟1 次（ 类型1）。3 万/ 60 / 10 = 50 次/ 秒	10 分钟 1 次 insert/update（50 次/ 秒）	100 字节	1M × 100 × 100 字节 = 10 吉字节
任务日志	1M × 100（ 平均任务数）	登录时只读取 1 次。10 分钟1 次（ 类型1）。3 万/ 60 / 10 = 50 次/ 秒	1 分钟 1 次 insert/update（500 次/ 秒）	10 字节	1M × 100 × 10 字节 = 1 吉字节
拍卖	1M × 10（平均竞拍数）	每次打开拍卖行窗口时。（ 类型2）10 分钟1 次。3 万/ 60 / 10 = 50 次/ 秒	每次竞拍时。10 小时 1 次（ 可以无视的频率）	100 字节	1M × 10 × 100 字节 = 10 吉字节

表	100 万登录用户时的行数	read 访问模式	write 访问模式	1 行的大小	合计大小
好友列表	1M × 100（ 最大好友登录数）	登录时只读取1 次。10 分钟1 次（ 类型1）。3 万/ 60 / 10 = 50 次/ 秒	每次添加好友时。1 小时 1 次（可以无视的频率）	100 字节	1M × 100 × 100字节 = 10 吉字节
邮件	1M × 100（平均邮件保存数）	每次打开邮件画面时。（类型2）10 分钟1 次。3 万/ 60 / 10 = 50 次/ 秒	每次发送邮件时。1 小时1 次（可以无视的频率）	1 千字节（最大值）	1M × 100 × 1 千字节 = 100 吉字节
公会	1M × 1（平均参加数）	登录时只读取1 次。10 分钟1 次（ 类型1）。3 万/ 60 / 10 = 50 次/ 秒	每次新加入种族和退出时。1 天1 次（可以无视的频率）	10 字节	1M × 1 × 10 字节 = 10 兆字节
日志	1M × 10 万	不会发生read（在备用服务器上运行）	10 秒1 次 insert（3000 次/ 秒）	100 字节	1M × 10 万 × 100 字节 = 10 钛字节

表的特性、必须注意的表

表 4.11 总结了各个表的特性，下面我们对这些信息加以整理。

首先是关于 read 的，除了日志以外，其他表的访问模式都是 10 分钟 1 次。另外，只有在拍卖和邮件等历史信息很重要、需要进行检索的表中采用类型 2，其他都是类型 1。对于类型 1，具备 KVS 的功能就足够了。

接着，关于 write，对于用户表，基本上不进行写入。对于角色、物品、任务日志，1 分钟写入 1 次，是读取频率的 10 倍。C/S MMO 与其他 Web 服务的一大区别就是“写入占了绝大部分”。

对于日志表，10 秒进行 1 次 write，整体就是每秒 3000 次，频率相当高，所以必须采取一些措施。其他的表都是 1 小时～1 天 1 次，频

率很低，不会成为负荷。

综上所述，对于 read，只要注意“拍卖”、“邮件”这两个表。对于 write，注意“角色”、“物品”、“任务日志”这几个表就可以了。

查询的内容 ——read 篇

知道了需要注意的表后，我们就可以回到游戏的策划内容上，简单地来看一下实际要进行怎样的查询。首先来看一下 read 中的拍卖表。

游戏世界中设有拍卖行，如果玩家角色与那里的 NPC 进行对话后，就会打开专门的交易窗口，使用该窗口进行所有的操作。*K Online* 中的竞拍规则与 Yahoo! 相同，出价最高者可以得到竞拍物品。这与股票市场不同，各个被拍卖的物品具有很多信息，完全相同的拍卖品基本上一个也没有。比如，在 *K Online* 中，拍卖物品具有“等级 23 的铁剑，攻击魔法 +3%，嵌有宝石”等各种信息。

拍卖的典型实现方法是分为两个表：“拍卖物品表”（每一行对应一个拍卖物品）和“竞价表”（每一行对应一名买家的竞拍价格）。只有在拍卖和出价时才会进行 write，查询频率很低，所以完全没有问题。

“在买家查询拍卖物品时显示列表”和“获取某个拍卖物品的竞价列表”这两种处理的负荷较高。

列表的显示需要同时用到多个设置在拍卖物品表中的索引，将其进行排序，然后每次显示 20 项左右的结果。每秒 50 次，也就是说要在 20 毫秒内完成 1 次处理。这相当令人担忧。估计拍卖表的大小为 10 吉字节左右，所以通过装配更大的服务器内存，可以全部在内存中进行处理，这样处理速度可能比 20 毫秒更快，但是在测定处理时间之前，最好还是不要做出这么乐观的估计。必须想好将来处理性能不足时所能采取的措施。那时，还是要回到策划内容上来加以考虑。

首先，查询是只读处理。通常，如果允许 read 操作在时间上有所延迟，就可以获得很大的扩展性。对 *K Online* 的策划内容加以确认，由于是在新物品加入拍卖之后才开始更新拍卖物品列表的，就算延迟 10～20 秒以上也没关系，所以在 DBMS 的处理性能不足的情况下，可以考虑使用 DBMS 的复制功能和 memcached 等在之后改变表的数据项结构。获取竞价历史也可以考虑同样的方法。

与拍卖一样具有很多 read 操作的还有邮件，在提高邮件读取的性能时，可以采取与拍卖相同的方式。

查询的内容 ——write 篇

接着考虑一下对角色表、物品表、任务日志表的 write 操作。同时连接数为 3 万的情况下，每秒对这些表写入 500 次。

- 角色表
    - 每个角色 10 千字节
    - 每个角色占 1 行
    - 10 千字节
    - 大约 100 列
    - 通过 update 写入
    - 不执行 delete
- 物品表
    - 每个物品 100 字节
    - 每个物品占 1 行
    - 每个角色设定 100 行（合计 10 千字节）
    - 对表的操作每次进行 1 行
        - 获得物品时 insert1 行
        - 使用物品时 delete1 行
        - 物品状态发生变更时 update1 行
- 任务日志
    - 每个任务 100 字节

- 每个任务占 1 行
- 每个角色设定 100 行（合计 10 千字节）
- 开始一个任务时 insert1 行
- 任务状态发生变更以及完成任务时 update1 行
- 不执行 delete

保存的数据最多的显然是角色表。对于 100 个列，如果每秒 500 次、每次写入 10 千字节，就相当于 5 兆字节 / 秒。频率相当高，单从数字来看，就相当于在 1 台 Linux 机器上运行 1 个 DBMS（比如 MySQL）实例所处理的量。但是万一用户进一步增加的话，还可以做些什么呢？

在 *K Online* 中，我们使用索引来查询角色的状态，所以对数据库进行横向分割是很容易的。横向分割是指什么呢？比如说，将表示用户 ID 的字符串按照 A～M、N～Z 这样根据取值的范围来分割。通常使用的分割方式有取值范围、列表、hash 值等。这是一个不用更改表结构，之后还能采取一些对策的好方法。

＊ ＊ ＊

以上就是制定数据库设计图时所要注意的一些方面。由于篇幅所限，有关实际文档的内容在此就不作详细介绍了，如此前所述，不管是 ER 图这样的表设计文档，还是 Excel 等，都可以用来进行表结构的设计，只要将本节所述的内容在文档中进行总结就可以了。

只要在实际开始编程之前，对这些数据库方面的问题进行充分的讨论就可以了。

4.12　服务器 / 客户端软件 + 中间件——实践中不可或缺的开发基础

下面我们来看一下为了获得作为最终运行的交付物的软件套件，我们需要做些什么工作。读者或许想要开始编写程序了，但是实际上还没有进入这一环节。在进行编程之际，需要具备一定程度的技术基础。本节就

来介绍一下网络游戏开发中所需的技术基础：中间件和服务器 / 客户端开发的基础软件。

4.12.1　网络游戏的中间件

K Online 是商业项目，编程工作应该尽可能简化。如果能降低工作量就能提高利润，这是因为收回投资所需的时间变短了。此外，在要编写的程序中，对于那些与游戏差异化元素无关的部分，应该积极使用已有的资源，把时间花在开发差异化内容的工作上。

如果从头开始进行 C/S MMO 游戏编程，就会将大量的时间花在所有的游戏都需要的部分上。这样，在体现游戏可玩性（这是 *K Online* 的价值所在）的工作上所花费的时间就减少了。为此，我们要灵活使用各种支持 C/S MMO 游戏的开发工具来进行开发工作，这类开发工具称为“中间件”。

本书的主旨并非介绍高效开发网络游戏的方法，而是“支持”实际的网络游戏的技术。所以还是要对中间件的内容加以说明。所以本书以“尽可能不使用 C/S MMO 专用的中间件，而是自己独立实现”为前提来讲解。但是在实际的商业项目中，从开发成本的观点来看，选择这种方法是很少的。但从实践的角度来看，对中间件的类型有所了解也是很有用的，所以本节简单介绍一下。

C/S MMO 中间件

C/S MMO 中间件有好几种形式。一般来讲，中间件中存在着一种基本的权衡关系：“越是对应用程序的用途加以限制，就越能减少编程量”。

举一个不是 C/S MMO 的例子，第 1 章中提到过的《半条命》，基于该游戏，只对其设定数据进行更改而没有直接修改程序，就开发出了另一款游戏《反恐精英》。在《反恐精英》中，其游戏类型（实时 FPS）、最大同时在线数、反应速度、物理行为、地图大小、对应的平台、画面渲染算法、基本的操作体系等游戏的基本部分完全沿用了《半条命》中原有的代码。不同的只是地形、角色模型、敌人的行动和用户界面。

笔者曾参与过大规模的幻想 MMORPG，也参与过一些休闲游戏，中间件的使用形式可以大致分为“全功能型”、“小规模型”、“通信中间件”3种。

全功能型中间件

在 C/S MMO 中也有售卖以现有 C/S MMO 游戏为基础的工具。当前最著名的有美国 Bigworld 公司的产品 BigWorld MMO Technology Suite 。该产品以市场规模最大的幻想 MMORPG 为中心。其特点就是完整包括了以下这些内容：在模拟了广阔的 3D 地球环境的自然环境中高效部署地下洞窟、神殿和高塔等在 RPG 中经常出现的地形的工具、定义敌人行为的工具、可以使用 Python 来自定义游戏服务器的系统、与 3ds Max 等建模工具进行协作的功能，以及客户端程序的框架，等等。仅仅是备齐这些功能恐怕就要数千万日元以上。这种中间件可以称为“全功能型中间件”，使用这种中间件基本上可以省掉编程工作。

关注于幻想 MMORPG 的全功能型中间件还有 Hero Engine 和 Monumental 公司的产品等，这类中间件经常在拥有几亿日元预算的大规模游戏中使用。

C/S MMO 并不只有预算规模相当大的幻想 MMORPG。这几年，休闲 MMO 和虚拟世界类游戏的用户急剧扩大。这些项目的预算额也有在 5000 万日元以下的，非 3D 的也很多。

事实上，*K Online* 所参照的 *Runescape* 就是其代表，所以不需要像 BigWorld 产品或 Hero Engine 这样处理真正的 3D 空间，也不需要那些基于物理规则的实时行为以及服务器地形的无缝连接。比起这些功能，更需要关注物品的拍卖功能和任务脚本的实现等独特的功能，但这些功能大多不包含在全功能型的中间件中。

小规模型 MMOG 中间件

在预算规模较小、游戏内容不是幻想 MMORPG 的情况下，全功能型中间件就不怎么适用了。但即使是这种情况，也应该尽量避免完全自己开发。减少自己开发的内容，参考已有的内容，在这一点上，与《网络创世纪》兼容的协议所对应的服务器充斥着市场，所以也可以参考[34]。此外也有一些商业 MMOG（比如 *Ryzom*[35] 等）公开源代码的情况。过去一段时期中，也有一些实现了 MMORPG 中部分功能的开源中间件可供获取，但是使用这种中间件的引擎基本都没有被广泛使用。在使用开源 MMO 中间件的情况下，需要自己对源代码的使用负责。特别是，与 Web 服务等不同，MMO 并不是游戏通用的工具，所以没有像针对 Web 的框架那样的用户社区很发达的工具。因此，参考源代码跟自己来开发基本上没什么差别。

[34] 比如可以参考 SunUO 中用 C# 编写的程序。http://www.sunuo.org/download.html

[35] http://www.ryzom.com/en/

开源中间件可以称为“小规模型 MMOG 中间件”。比起全功能型中间件，使用这种小规模型的中间件，可以制作更多类型的游戏，但是需要自己开发的部分多了很多。

通信中间件

还有种更加不涉及游戏内容的解决方案。那就是对游戏内容一概不作规定，纯粹只用通信中间件，除此之外全部自己制作。这里所说的通信中间件指的是 libevent 和 Twisted 等用于处理网络事件的中间件。

在称为虚拟空间和虚拟人物聊天的服务中，虽然服务器的基本结构以及数据库设计都与 MMORPG 的相似，但是它不存在游戏世界的概念，没有活动在服务器中的敌人，玩家之间的交互除了聊天就再没有别的了，所以只要具备极其简化的服务器就足够了。为此，反倒是只有通信中间件更为简洁，不需要进行某些清理工作（这里的清理工作是指去掉不必要的处理），开发效率更好。

* * *

以上对使用中间件的服务器开发进行了简要说明。此外，在讨论实际项目中所使用的中间件时要基于实际的验证。

4.12.2　开发的基础软件——可以立刻尝试的 C/S MMO 开发体验

如前所述，本书并不使用正式的中间件，尽可能从基本部分开始介绍支持网络游戏的技术，这才是本书的目的。话虽如此，如果所有的程序都用汇编语言来编写，所有的数据都用二进制数据编辑器来制定，也实在是与现实项目中的做法偏离太大了。甚至连操作系统也必须自己编写了。因此，必须选择一些作为说明基础的系统。

为了更好地进行说明，我们选择网络游戏业界内外都广泛使用的并且可以用来实际开展商业服务，能随时随地入手的开源通用软件，以及能够免费使用的软件来构建我们的游戏[36]。

> 36 笔者很想使用工作中所开发的中间件，但是遗憾的是，该中间件尚未公开源码（还没有决定能否公开）。

服务器相关的软件

大致具备了以下的库、操作系统和中间件后，就可以构建 *K Online* 的整体结构了，我们首先来看看服务器相关的软件。

- Linux

  作为操作系统的一个选择，Linux 操作系统是很具代表性的。2～3 年以内发布的任何一种主套件都可以。其他的选择还有 FreeBSD、Solaris、Windows Server。从安全性、维护性，以及高负荷下的稳定性等观点来考虑，基本上都使用 UNIX 内核的操作系统。本书假设使用 Ubuntu Linux ，要求具备标准的 socket/inet 体系的系统调用和标准库函数。

- MySQL

  DBMS 中大多使用 MySQL，多数情况下都能实现足够的性能。同样，2～3 年内发布的任何一种稳定版本都可以。其他也有使用 PostgreSQL、Microsoft SQL Server、Oracle 的，但是网络游戏还是使用 MySQL 居多。

- OpenSSL

  游戏客户端和服务器之间会通过 TCP 通信来传输数据流，为了对数据流进行加密，需要对使用 RSA 和 Diffie-Hellman 等方法的密钥交换、块密钥加密算法以及能够高效应对攻击的 hash 函数加以实现，所以要使用 2～3 年以内的稳定版本。

- GCC（Gnu Compiler Collection）

  C/S 服务器开发中最常用的编程语言是 C 语言或者 C++，其次是 Java，轻量级语言在游戏服务器中基本不使用。本书使用的是最为常用的 C++，所以使用 GCC 进行来编译。为了节约代码量，部分内容使用 STL（标准模板库）来实现。实际上，为了节约编程时间，也有适当使用 JSON（JavaScript Object Notation）库和 XML 读取程序等小型程序库的，但基本上只要是开源的，Google 能搜索到的工具就够用了。

- 轻量级语言

  Python、Perl、Ruby 等在通用的系统编程中使用的语言，本书假设使用 Ruby。为了开发用来实现辅助服务器和用来生成代码的工具，以及开发管理工具，轻量级语言是必不可少的。

客户端相关的软件

接着我们来看一下面向客户端的一些必要的软件。*K Online* 是面向 Windows PC 的本地应用。与 *Runescape* 相同，相对比较简单，虽然其显示效果是最低程度的，但是也需要让用户体验到 3D 世界，为此，需要以下这些模块。

- DirextX

  渲染画面必须具备的库。比 DirectX 移植性更高的还有 OpenGL、SDL（Simple DirectMedia Layer）和 Ogre 等各种库。渲染方面的内容超出了本书的范围，所以不作详细介绍。使用 DirectX 时，可以通过少量的程序来实现动画，而 3ds Max 和 maya 等很多 3D 建模工具都可以输出建模数据，非常便利，在游戏行业中使用最为广泛。DirectX 有一点非常好，那就是有很多易于使用的示例程序，可以以这些示例为基础进行简单的扩展来制作游戏。

- Winsock

  Windows 下的标准套接字库。可以免费使用。

- Visual Studio .NET

  集成的编程环境。使用 Cygwin/GCC 和免费编辑器这种组合也没有问题，但是通常，客户端程序的调试与服务器的不同，3D 和 GUI 的实现等方面需要大量用到复杂的数据结构，如果不能运用功能强大的调试器，就会严重降低开发效率，所以通常必须使用 Visual Studio（服务器中则没有这个限制）。

- OpenSSL

  不仅在服务器端需要加密程序，在客户端上也同样需要。因为要与 Windows 兼容，所以要使用相同的包下的相同的源代码。

* * *

综上所述，作为 C/S MMO 游戏开发技术的基础，全部只使用开源工具和免费工具也能完成。但是为了提高开发效率，还是要以 Visual Studio 和各种通信中间件为主，根据项目需要以及项目预算来结合使用各种工具。

4.13　程序开发中的基本原则

接下来马上要进入编程环节了，但是还需要少许前提知识。在开始编程时以及在开发过程中，需要先了解一些基本原则。

4.13.1　如何开始编程、如何继续编程

在不使用全功能型中间件的情况下，开始编程和继续编程时具有一定的基本原则。下面列举了 4 项。

1. **数据结构优先原则：构成游戏世界的数据结构要在编写代码之前大致决定下来。**

2. **维持可玩状态原则：经常试玩游戏，在对游戏的可玩状态进行确认后进行开发。**

3. **后端服务器延后原则：整个系统并非一下子能构建出来，而是以客户端→前端游戏服务器→后端服务器的顺序，从最靠近终端用户的一端开始开发。特别是后端服务器的开发要尽可能靠后。**

4. **持续测定原则：经常对延迟、带宽、CPU 时间进行测定，将其表示出来，以此为参照进行开发。**

下面我们就针对 *K Online* 来看一下以上各个原则。

4.13.2　数据结构优先原则——基本原则1

首先来学习“数据结构优先原则”。

构成游戏世界的“数据结构”是游戏策划负责人和程序开发负责人两方都能理解的共同的部分。通过审视这些数据结构并且事前进行商谈，策

划人员可以对能否实现策划的游戏内容进行确认，而程序开发人员可以对在游戏程序中使用怎么样的算法来处理这些已经决定了的数据结构进行考虑。

在游戏开发中，后续更改数据结构会对整体的游戏规则产生很大的影响，所以基本上是不可能的。因此，在实际编程之前，使用两方都能理解的方式来说明是很重要的。

视频游戏中的数据分类

不仅是 C/S 游戏，几乎所有的视频游戏都可以将数据分为两大类来加以实现，第一种类型就是以地形数据和棋盘信息为代表的“基本不会变化的部分”，第二种就是部署在其上的“频繁变化的部分”。不会变化的部分称为地图、BG（Background）、棋盘、地形、背景等。经常变化的部分称为对象、角色、NPC、行动者、可移动的、可动物体等。它们的动作频率的差异在 100～1000 倍。动作指的是对保存在内存中的值进行写操作的频率很高，会对实现算法产生影响。

在这些术语中，本书将不会动的物体称为“BG”，会动的物体称为“可动物体”[37] 。

> [37] 游戏行业中经常使用的说法是“对象”（Object），但是这与“面向对象”这种术语冲突了，所以本书采用“可动物体”这种说法。

C/S MMO 游戏中的游戏世界由 BG 和可动物体构成。以 *K Online* 为例，游戏内出现的物体可以分为“明显作为 BG 存在”、“明显作为可动物体存在”、“介于两者之间的物体”，下面我们就来看一下每种类型中各有些什么样的物体。

- **明显作为 BG 存在的物体**

    - 地面

      →地面是完全固定的，不会发生变化。对地面没有什么可做的操作，但是地面的类型分为可以通行的和无法通行的。地面由称为“高度图”（Heightmap）的具有高度信息的顶点数据的集合组成，用格状形式来表示（参见图 4.29）。

      图 4.29　高度图

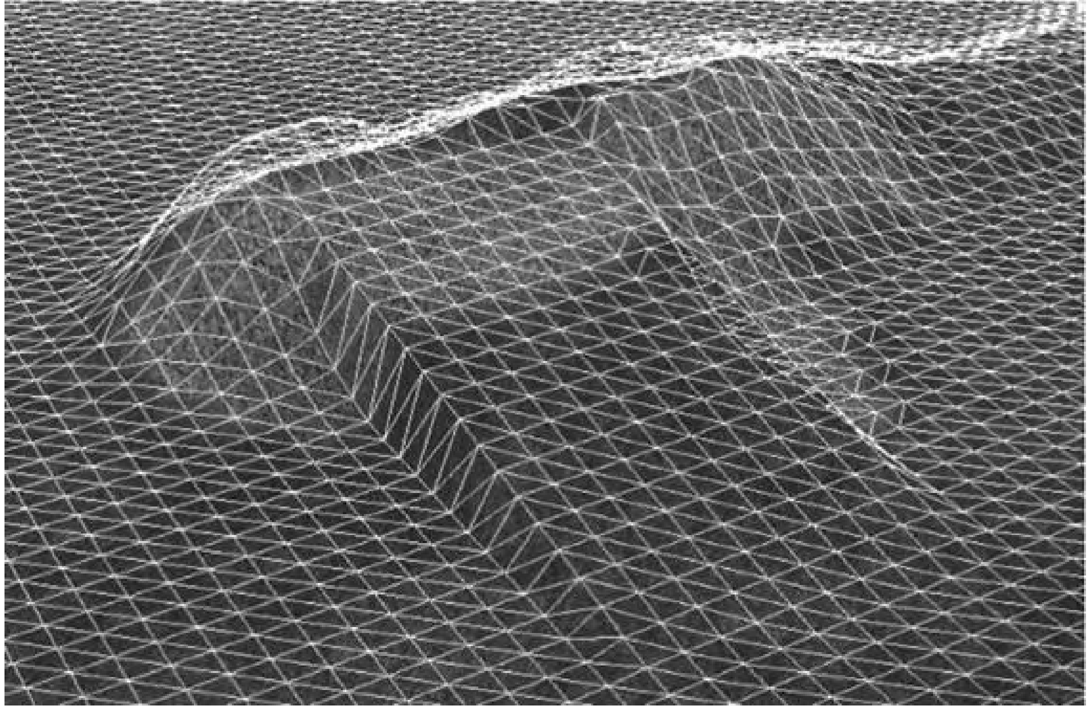


图 4.29 所示的顶点数据的集合一次大致能在画面上显示 15×15 个顶点的范围。因此，3D 多边形需要用 15×15×2 = 450 个顶点来表示。此外，策划要求在服务器中表现出一个具有 4000×4000 = 1600 万个顶点的广阔世界，如果预计每个顶点包括高度和种类信息在内，需要 20 个字节的话，那么 1600 万 × 20 = 320 兆字节的数据量全部都要在内存中处理。从现在的服务器性能来考虑，该数据量并不是很大，大概没什么问题。用一个二维数组（数组中的元素就是保存着高度和种类信息的结构体）来管理就基本上可以了。

- 天空

  →也称为“天体”。用来进行云彩和太阳等的渲染。根据时间段而变化。策划上需要白天、黑夜、早晨、傍晚 4 种类型。通常画面上只显示一种。天空的信息不会给服务器的内存造成负担，所以服务器方面不会有什么问题。不需要特别的数据结构。

- 建筑物

  → *K Online* 中，建筑物都是背景，不会发生变化。在画面上，建筑物一次最多显示 10 个。从策划的层面出发，确认全

部需要多少建筑物，整个游戏世界 1000 个左右的建筑物应该就可以了。相对于之前的 1600 万个顶点数，这里的 1000 足足小了 5 位数，位置也不会发生变化，所以具有一个保存所有建筑物的列表，从坐标开始用 R-tree[38] 来检索就可以了（另外，或许也有引用顶点数据的方法）。

- **明显作为可动物体存在的物体**
  - 玩家角色

    →玩家角色每秒行动 2～5 次。可见范围内 1 次最多显示 20 人。虽然希望有尽可能多的玩家参与游戏，但是如前所述，每个服务器内核可以实现 500 人同时在线。这也可以采用列表 + R-Tree，或者在顶点中维护指针的方法。一个单元中可以存在多名玩家角色时可以使用列表吗？如果不可以，能使用引用吗？顺便提一下，这里所说的“单元”指的是由 4 个顶点围成的四边形区域，在 *K Online* 中，可动物体可以在以这些单元为单位的区域中行动。

  - 敌方角色

    →敌方角色每秒移动 0～2 次。画面中 1 次最多出现 20 个。如前所述，整个世界中，每个登录玩家对应 10～20 名左右的敌人，所以从服务器的每个内核来考虑的话，就是要使 5000～1 万个敌人行动。这是相当多的，所以服务器的内存和 CPU 都消耗相当大。这比地形的顶点数小了 3 位数，所以可以采用指针，或者 列表 + R-Tree 的方式。

  - NPC（Non-PlayerCharacter，非玩家控制角色）

    → NPC 有数秒内来回走动 1 次的，也有站在原地不动的，即使是不动的 NPC，也可以点击它，与它进行对话等操作。同时显示在画面上的 NPC 最多有 5 个。其数量与建筑物大致相同的话就足够了，所以设定为 1000 个。因为数量较少，所以基本不会给服务器内存造成负担。也可以采用列表 +R-Tree。

  - 飞行中的炮弹和魔法效果

    →飞行中的炮弹和魔法效果每秒移动 1～5 次。同时出现在画面上的数量相当于玩家角色与敌方角色数量的总和，也就是最

多 40 个。考虑到如果每名玩家 1 个，每个服务器内核处理 500～1000 个，如果进一步考虑敌人发射炮弹，可能会出现 2000 个左右。该要素最可能消耗 CPU，数量上还是比顶点少了 3 位数，所以不用数组保存，用列表 + R-Tree 就可以。

- **介于两者之间的物体**
  - 地面上生长的植物

    →包括作为背景生长的花草，以及可以砍倒的大树等。草是背景，树木是可动物体。被砍倒的树木过几分钟之后会在同一个地方再生。通常，“树木”在草原上为数百单元 1 棵，在较为稀疏的树林中为数十单元 1 棵，在密林中则是几个单元 1 棵。最好的情况是处于画面中所显示的单元的一半，最坏的情况是配置为所有单元的一半，所以估计大约要在画面上显示 15×15/2 = 110 个，这由制定地形数据的人员来决定。在服务器中，整个游戏世界的面积为 4000×4000，如果以世界中所有场所平均一半的密度来配置，大约 800 万。与顶点数具有相同的位数，所以如果每个消耗 20 字节的内存，就需要 160 兆字节。因为树木不会移动（位置不会发生变化），所以应该没关系。

  - 地面上的岩石

    →小沙砾为背景，击碎后可以出矿石的大岩石为可动物体。同样，击碎后过了数分钟会自动再生。这与之前所述的树木相同，在画面上最多表示 110 个，这个数字是与植物加起来的总和。岩石也不会移动，所以没关系。与定点数量具有相同的位数，使用数组存储。

  - 其他

    →空中飞行的昆虫、喷泉、流淌的水和雨滴等出现在游戏画面上的所有要素，它们是背景还是可动物体，要与策划人员进行确认后加以分类。这些总共要在画面上显示 100 个。数量很少，所以没有什么问题。

[38] R-Tree 是一种空间索引（Spacial Indexes）数据结构。可以在 2 维空间内进行高速查询。

4.13.3 实现数据结构之前的讨论——出现在画面上和不出现在画面上的元素

450 个作为背景地面的多边形、天空、10 栋建筑、20 个玩家角色、25 个 NPC（包括敌人）、40 个效果、合计 110 个的植物和岩石，所有这些同时显示在画面上时，客户端程序的渲染性能会不会出现问题呢？这需要程序员结合玩家的运行环境的条件来确认。对整体的数量与单个物体的渲染品质进行权衡，以此来进行判断。此外，在服务器方面，还要考虑总共需要多少内存（在 *K Online* 中，每个内核需要将近 1 吉字节）和 CPU（*K Online* 中，同时在线数为 500 左右），以此来讨论其可行性。

以上，我们对实际出现在画面上的要素进行了分类，也考虑了各类物品的最大数量。同时，不出现在画面上的要素，也就是维持可动物体密度的处理也是需要考虑的。下面，我们结合示例来看一下。

敌方角色和 POP 设定

在 *K Online* 中，作为代表性可动物体的“敌方角色”在被玩家攻击，受到一定的伤害之后就会被打倒，然后消失。根据 *K Online* 的游戏策划，在沙漠中会有一种名为“蝎子”的敌人以一定的密度出现，由于在 C/S MMO 中，所有的玩家都是共享游戏信息的，所以玩家打倒蝎子后，其数量就会减少，很快就会灭绝了。如果敌方角色灭绝，之后的玩家就无法获得经验值了，所以必须采用某种方法来恢复蝎子的数量。

一般在 C/S MMO 中，敌方角色出现被称为“POP”。为了使敌方角色以一定的密度存在于某个范围之内，需要进行如下处理：（1）对该范围内的敌人进行计数；（2）数量没有达到要求的数量时，在随机位置上 POP。这种处理在单纯使用随机数的情况下负荷并不高。

K Online 的整个世界中的 POP 处理如图 4.30 所示。

图 4.30　*K Online* 整个世界中的 Pop 处理

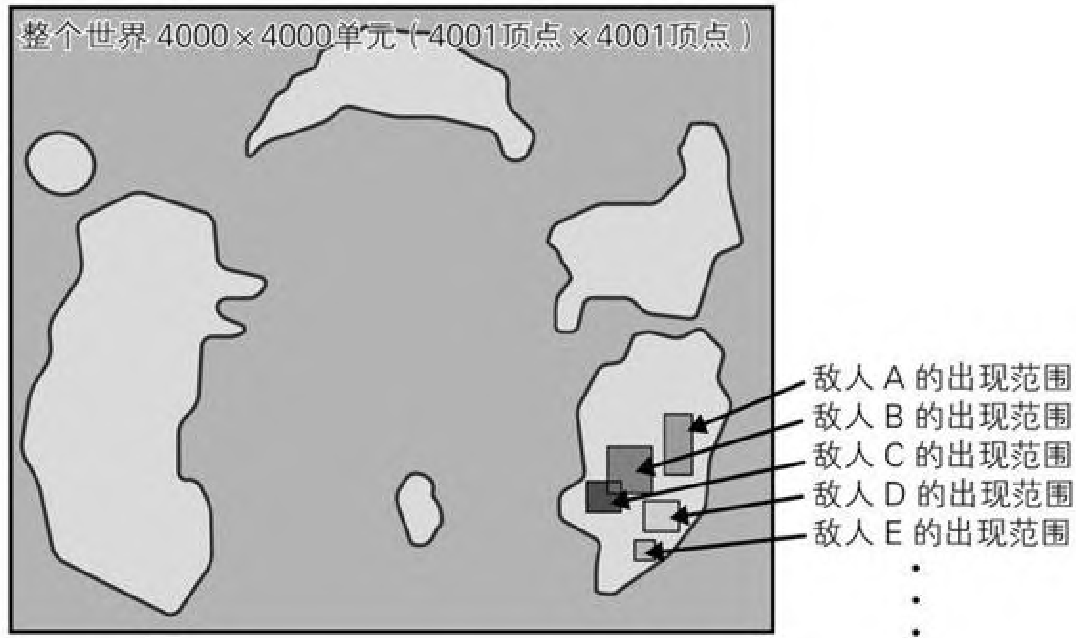


图 4.30 中，首先，整个世界的面积为 4000×4000 个单位。在位于地图右下角的大陆中用矩形（也有使用多边形和密度方程式的）表示了 5 种类型的敌人。实际的游戏中敌人的设定会多很多。

在服务器中对存在于矩形中的敌人进行计数，不到规定数额时，就创建新的敌人来加以补充，这称为“POP 设定”。

对策划内容进行确认后可以知道，在 *K Online* 中，该范围内的信息（POP 设定）大致需要 5000～1. 5 万左右。对于所有单元（4000×4000 = 1600 万）的顶点，一个设定信息（矩形所示的范围）的平均面积大约 1000～10 000 左右。假设需要一直对 10 000 个设定（矩形）进行检测，如果每秒检测 100 个，那么每 100 秒可以对 1 个地方的敌方角色进行补充。策划人员判断等待 100 秒没有问题，所以就按照这样的计算来进行实现。时间过长就会导致所有的敌人都被玩家打倒，然后全部消失，数十秒后再一下子出现大量敌人。在实现上，通常只对玩家所在的地方进行处理。

根据策划内容，为了提高游戏的可玩性，或者为了保证游戏具有更好的平衡性，根据服务器的“游戏内部时间段”、气候、附近存在的玩家人数，也有动态改变 Pop 设定的。在对这类处理加以组合的情况下，CPU 的处理量就会有所增加，所以要更加注意服务器的超负荷问题。

＊ ＊ ＊

以上我们基于构成游戏世界的地形、可动物体的数量及其密度、性质等方面的策划内容，介绍了使用什么样的数据结构，并且对使用怎样的算法进行了说明。

4.13.4　维持可玩状态的原则——基本原则2

接着我们来讨论一下“维持可玩状态的原则”。游戏开发中难度最高的就是“使游戏具有可玩性”，不做做看的话是不知道游戏是否具有可玩性的。一边开发一边反复进行调整，反复调整的次数越多，游戏就会越有趣。这与“制订计划→设计→实现→单元测试→集成测试”这种瀑布模型不符。

在游戏开发中，如何增加“一边开发一边调整”的迭代次数是开发的一大课题，C/S MMO 也不例外。首先要快速制作原型，对其进行修改，并且在此过程中确认对游戏的可玩性。在原型开发阶段，如果达不到预期的可玩性，或者无法发现预料之外的可玩性，那么即使进一步推进开发，也很难使游戏具有可玩性。

C/S MMO 中也可以采取以下这种理想的开发方法：在开始编程之前，在一定程度上进行本书至此所介绍的设计，尽早制作原型，试玩之后在 1～3 天内尽量进行小幅修改，然后不断地重复试玩。刚开始时以 1 天为单位进行修改，如果能够对游戏可玩性进行确认，就能逐渐延长修改的间隔。间隔的延长能使更多的人员更方便地来处理。

4.13.5　后端服务器的延后原则——基本原则3

接下来，我们再来看一下“后端服务器的延后原则”。

优先考虑游戏的可玩状态，在确认游戏可玩性的同时对其进行修改，从而推进开发。在这种情况下，后端服务器与游戏可玩性没有任何关系，所以不应该在项目一开始就着手开发。准确说来，应该延后开发的与其说是后端服务器，不如说是与游戏可玩性无关的所有系统（虽然它们对于实现扩展性和安全性是必不可少的）。首先开发这些系统不仅浪费时间，还会影响反复修改的速度。但是为了使这种推进方式可行，需要像本章所述的那样，对后端服务器相关的部分进行一定程度的设计，并对设备进行估算，然后将其以一定的程度文档化。

如果延后后端服务器的开发，就无法“从中途开始游戏”。这无疑会给试玩造成障碍，所以必须加入最低限度的持久化处理。所以，为了尽早

完成游戏原型，需要按照以下的顺序进行。

整个大流程就是：一口气实现在客户端中显示游戏信息的程序，随后定义通信协议的 API，在 gmsv 中实现那些 API。还是从靠近终端用户的一侧开始逐步进行开发。*K Online* 的原型就是一口气完成的，这是大部分 C/S MMO 游戏通用的模式。详细的实现将在之后介绍。

4.13.6　持续测定的原则——基本原则4

最后，我们来看一下“持续测定原则”。

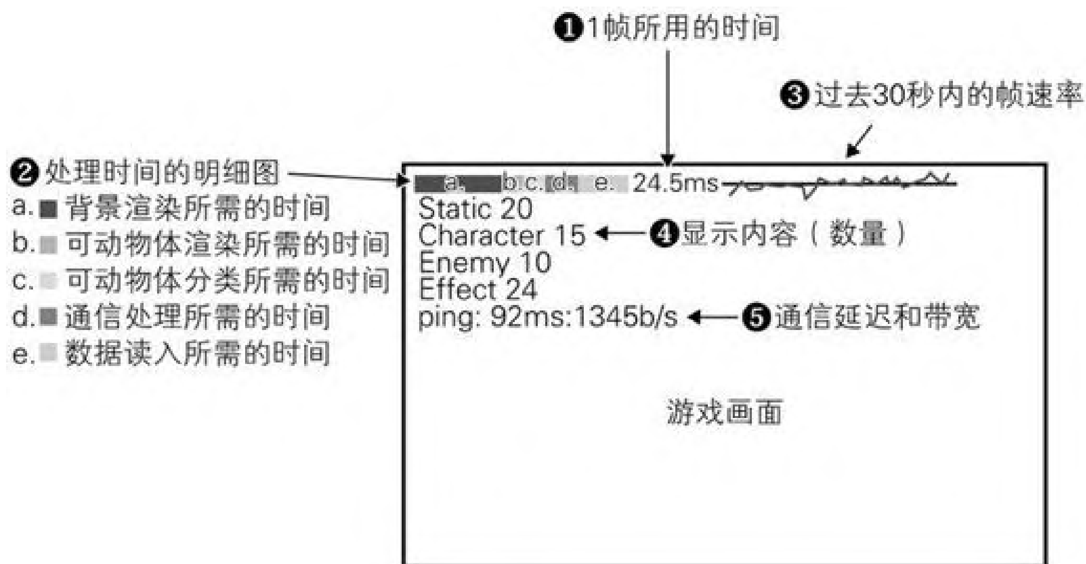
在网络游戏中，如果通信和渲染方面存在延迟，就会有损游戏体验。游戏处理的简单程度在游戏价值中占了很大一部分。之后集中进行性能试验后再进行调整是很难的，所以在游戏的开发过程中，需要时常对处理速度进行监控，从而尽早发现问题，保持程序稳定。

在 C/S MMO 中，从原型开发阶段开始，主要程序要素包括“客户端”和“服务器”，这两者都可能发生性能上的问题。

客户端开发中持续测定的例子

图 4.31 是客户端开发中的画面。图中除了显示每 1 帧渲染所花的时间（图 4.31 ❶），还对这段时间内的各种处理进行了大致的分类，并以不同的颜色加以区分，持续地实时显示在画面上（❷ a.～e.）。此外，❸ 以图表方式显示了过去 30 秒内的帧渲染时间，❹ 显示了画面中当前所示内容的数量，❺ 显示了通信延迟（毫秒）和每秒传输了多少字节。

图 4.31　客户端开发中的画面（原型）※



※ 在实际的开发环境中，❷ 的明细图等简单地使用颜色来区分，以易于阅读的方式显示

在开发原型的过程中，基本上要一直显示这些信息。这样一来，当自己进行了某些修改导致数字急剧发生变化时，就能立刻发现问题了。通过使用有颜色的图表来对不同的处理进行可视化的显示（而不仅仅是用数字），可以更快地发现问题所在。

K Online 的客户端中要显示的基本内容就是以上这些了，但在实际的商业游戏开发中，应该尽可能通过输入客户端调试命令来详细追踪 VRAM 和内存的使用情况等。

服务器开发中的持续测定

接着是服务器。游戏服务器是长时间持续运行的进程，要持续地定期向日志文件输出状态。可以使用 tail -f 命令来查看日志文件，如图 4.32 所示。

图 4.32　游戏服务器的日志文件（tail-f）

```
.....S.....I.....I......S....SS..S.....S..0....0.....    ← ❶
Loopcount 660/sec char:27 item:7 plant:0 magic:0 tot:34 diff:0 PC:1
Session:1
send:41.91Kbps recv:1.81Kbps 97API   ← ❷（“Loopcount”～“API”为 1 行，❷'
也是同样））
S....S..SS.I..I.....S..S.........S..0.0....0I..I....+...I....S..   ←
❶'
```

```
Loopcount 1444/sec char:22 item:7 plant:0 magic:0 tot:29 diff:0 PC:0
Session:0
send:30.02Kbps recv:1.05Kbps 61API   ← ❷'
........S....I..   ← ❶''
```

服务器的消息循环每循环 100 次，就对该文件输出 1 个点号（.）。在服务器的开发中，设置一个子显示器，在画面中显示 tail -f 的输出结果，由此来检查循环是否中断，是否不均匀。此外，当发生了对服务器来说很重要的事件时，就输出点号以外的字符。在图 4.32 ❶❶'❶″的例子中，“S”表示保存玩家数据的瞬间，“I”表示玩家登录的瞬间，“O”表示登出的瞬间，这样，对服务器进行访问的集中度就能一目了然。

图 4.32 ❷❷'中，10 秒 1 次，进一步地将过去 10 秒内的统计数据以字符串的方式输出。其内容包括：过去 10 秒内有多少次消息循环、内存中处理的角色数量、物品数量、植物数量、效果数、整体的合计、玩家数、TCP 会话数、过去 10 秒内的平均传输速度以及通过 RPC 调用 API 的次数等。

比如，比起前一次的统计 ❷，❷'循环数多了 1 倍，由此可知由于某些原因导致访问模式发生了变化。如果呈数倍或者数十倍的规模发生变化，一定是发生了预料之外的事件。除了定期输出文字信息，查看使用其他工具输出的内容（比如，使用 top 命令后显示出来的内容，以及正在使用的带宽的情况等）也是很有帮助的。其关键就是，不仅要在游戏的服务运营中显示这些信息，在开发过程中、甚至是在原型中也要显示。

4.14 C/S MMO 游戏 *K Online* 的实现——编程开始！

牢记之前所说的原则，现在我们要开始编写程序了。在制定设计文档的阶段中已经大致定义了登录、玩家角色的移动、敌方角色的移动、聊天以及对敌人的攻击等通信协议方面所需传输的信息，但是在程序中连 1 个字符都还没有。下面我们从开发的安排开始。

4.14.1　开发的安排

如前所述，为了说明支持网络游戏的技术，本书不使用全功能型的中间件。在这种情况下，推荐按照以下阶段来进行开发。

❶ 框架阶段

对于游戏客户端（cli）、游戏服务器（gmsv）和数据库（dbsv），编写可以对最低程度的动作进行测试的 1 组程序。这个阶段编写的程序还谈不上是“游戏”。

❷ 原型阶段

保持运行状态，持续进行扩展和调整，将可以作为游戏来玩的版本作为原型来开发。最好是开发 1 个自动对服务器进行测试的测试客户端和 1 个用于让人们进行游戏的客户端。

❸ 整体框架的实现阶段

试玩游戏的原型版，如果能确信游戏具有可玩性，就可以开始制作开发工具、dbsv 以外的后端服务器，以及登录服务器等为了将游戏商业化所必须具备的相关要素。

❹ 量产阶段

进行游戏数据的量产。

❺ 收尾阶段

为了正式运营游戏，这个阶段以 Bug 修正、平衡性调整、试玩等为主要的开发活动。

❻ 运营开始后的阶段

并行实施用户支持和补丁的追加。

在上述阶段中，阶段 ❶ 由 1 名主程序员开发 1 周左右是最适当的。在阶段 ❷ 中，根据策划内容，通常由两人以上分担能完成得更快。比如，由两名程序员进行开发，测试客户端由阶段 ❶ 中的主程序员负责，包含

渲染在内的玩家客户端则由另一名程序员负责。进入阶段 ❸ 后，可以由 2～4 人负责。❹ 之后，由于所有的基本工作都已经完成了，所以由 10～20 人分担也是可能的。

4.14.2　*K Online* 中的分工计划

为了在 PC 性能低下的发展中国家进行普及，在策划上决定不将 *K Online* 作为 Java 应用来实现，而是作为 Windows 本地应用。此外，也不采用非常漂亮的 3D 空间的表现形式，而是用最低程度的表现形式来呈现游戏画面。*K Online* 选择以 2D 画面来显示游戏内容，即使是 2D 画面，也有很多像 Tibia 这种极具人气的游戏。

作出了以上决定后，

- **客户端程序的实现就变得很简单了。**
- **可以采用与服务器相同的语言来开发（服务器运行在 Linux 上，采用 C++ 开发，客户端运行在 Windows 上，也采用 C++ 开发）。**

这种情况下充分具备了由 1 个人来开发原型的条件。

与此相对地，如果服务器采用 C++，客户端采用 Flash 来开发，或者游戏内容采用第一人称视角，以 3D 形式呈现。在这种情况下，如果服务器和客户端原型都由一个人来开发，工作量就很大了，需要花费的时间也更长了，从商业进度的角度来看，这可能是无法接受的。

在 *K Online* 中，如果完成了框架阶段或者原型阶段，客户端和服务端就都可以由 1 名程序员在短时间内予以实现。

4.14.3　*K Online* 中“框架阶段”和“原型阶段”的区别

K Online 是一款在世界中冒险，打倒敌人以赚取经验值，同时培养角色的 MMORPG，在原型中必须要能确认这一部分的可玩性，然后加以实现。最初制作的框架只包括敌人出现在画面上并且来回走动，玩家看到后发起攻击，获得经验值然后升级，拾取物品并加以使用，这只是游戏的大

体结构，并不具有任何可玩性。但是，到了原型阶段，就要确认“是否是一个有趣的游戏”了。

可以说框架和原型的区别就在于是否实现了游戏的可玩性要素。

表 4.12 列出了 *K Online* 中框架和原型之间的差别。从表 4.12 中我们可以知道，在框架阶段，没有“不存在”的要素。所有的要素在框架阶段都已经存在一些了，在原型中就对这些要素进行扩展和改善、增加数量以及加以调整等，其结果就是使框架成为游戏。

表 4.12　*K Online* 中框架和原型之间的差别

项目	框架	原型
敌人的种类	1 种	5 种左右
敌人的思考例程	随机走动	实现简单的动作
物品	临时实现两种	基于策划方案实现 5 种左右
地图	随机生成	基于策划方案手工制作数据然后读入
角色成长	每个敌人的经验值是固定的，经验值每增加 100 升 1 级	手工制定 20 级以下的设定表，并加入原型中。加入敌人的经验值表

项目	框架	原型
渲染	由圆形和三角形、文字、网状组成的图标，沿用过去的游戏中所使用的 3D 模型、纹理、UI 图像等	结合游戏策划制定数据，然后替换框架中的部分
操作	多用使用键盘的特殊操作（比如，按下 A 进行攻击等）。稍微有点闪烁或延迟也没关系	结合游戏策划实现专用的操作体系。要求具有高度的平滑度、速度以及优秀操作设计
处理性能	不予考虑。即使知道处理速度慢，也还是使用线性搜索。但是如果实际的游戏体验有所迟缓就不行了。数据量很少，所以只需线性搜索	进行一定的优化，尽可能对实际的思考例程和渲染例程等处理负荷进行测定（为了确认可行性）。所使用的算法的计算量要与商业版游戏相同（比如，不再使用线性搜索，而是采用二叉树等）
数据存储	在退出游戏后再次登录后，还能以之前的状态继续进行游戏	沿用

因此，在 *K Online* 的框架开发和原型开发期间，只是对各工作事项进行大量的细节增加和调整，开发难度并不高。

在 *K Online* 中，框架和原型之间有种“平滑连接”的感觉，但事实上，这种不能明确划分阶段的状态可以说是通过对 MMOG 开发的良好设计，很好地进行了框架内容的规划。为什么这么说呢？

暂时忘掉表 4.12 中的内容，我们可以将框架的交付物定义为“敌人的思考例程、物品、角色的成长等内容都没有的”状态。实际上，由于在框架中并不改善游戏的可玩性，即使将这种“与游戏内容的本质完全无关的代码”定义为框架也没有问题。

但是对于框架代码，笔者也计划针对与游戏本质无关的内容大致实现“虽然不具可玩性，但也适当予以策划了的游戏规范”。比如经验值每攒满 100 升一次级，或者使用后 HP 能够恢复 10 点的物品，等等。实现这样的游戏规范事实上只需要 10 分钟或者 30 分钟左右（当然，这

是在熟练的情况下）就够了，基本上花不了多少时间。而实现具有良好平衡性的角色成长逻辑则需要花费数十小时，但敷衍了事的话只需要 10 分钟左右，其时间成本相差了数百倍。稍稍花一些工夫，就能事先发现一些实际的游戏开发阶段中出现的“重大设计问题”（比如保存角色时的系统异常、状态发生变化时的通信方式、在内存中保存物品的方法等）。

因此，这里所说的“框架和原型没有区别，两者联系在一起”是理想策划的结果。虽然两者平滑地联系在一起，但在框架阶段中，还是不能确认游戏的可玩性，而在原型阶段中则可以对其进行确认，这一点依然是它们之间最大的区别。

4.14.4 ［步骤 1～2］框架～原型阶段

如前所述，在开发 *K Online* 的框架代码时，并不考虑可玩性方面的内容，但是需要开发游戏内容的基本部分。

接下来，我们从步骤 1 开始，平稳地过渡到步骤 2。

框架代码的准备

首先准备框架代码。为此我们来考虑一下一些必要的元素。首先准备一个文本文件，列出各项实现事项，基本上要按顺序来实现。将该文件命名为 todo.txt，将其放置在项目目录的顶部以便随时可以在编辑时参照，这种做法是笔者非常赞同。

此外，在这个阶段中，不仅需要实现的内容增减很厉害、实现顺序经常发生变化，而且由于是一个人进行开发的，因此比起任务管理系统的跟踪管理[39]，使用易于灵活操作的文本文件更为适合。下面依次列出了各项内容。

[39] 这种类型的跟踪管理可以使用 Trac、Mantis、Basecamp 等。

•todo.txt（的内容）

•cli

←2
•autocli（用于自动测试的客户端）

←1 •系统

- 对各协议进行整体测试的bot
  - 任务（Task）、Sprite等的基本系统
  - BMP读入
  - BG
  - 滚动（因为是demo，所以画面超出范围）
  - 可动物体

←4

- 角色
- HUDA[54]

- 性能表现
- ping
- ESC菜单

←5

- 上下光标移动
- quit
- login ID选择
- 操作
  - 点击背景进行移动
  - 点击可动物进行攻击
- 通信
- **协议**

←3

- signup
- login
- landscape
- move
- movablestatus
- disappear
- attack
- quest
- item
- useItem
- equip
- chat

- 渲染（`SDL`）
- useskill
  - shop（推迟）
  - buy（推迟）
- sell（推迟）
- **ID**
  - 背景ID
- 可动物体ID
  - 物品ID
  - 结果ID
- **gmsv**
  - 可动物体的分类
  - verify()
  - 调试命令
  - 敌人出现
  - popper（密度检测）
  - 敌人聚在一起
  - 读入地形数据
- 移动
  - 不能进入水中
  - 攻击
    - 敌人damage
      - HP = 0（打倒）
      - 经验值上升
      - 等级上升
    - 掉落物品
      - 拾取物品
    - 被敌人攻击
      - 如果死了则受到处罚
    - 在数据库中保存变化了的状态
- **dbsv**

40.Head-Up Display（平视显示器）。在画面上重叠显示体力值等信息。

制定以上的列表时，可以通过几大要点来确保没有遗漏，那就是1autocli、2 cli、3 协议、4 ID、5 gmsv、6 dbsv（后端，数据库保存）这几个大类的顺序以及各自的内容。如果与各项具体的开发内容结

合起来来制定各条目，可以每完成一项就划掉 1 行，这样对开发进度进行确认的成本就降低了。下面我们来简单地看一下1～6这几个大类。

1 autocli

autocli 是自动测试程序。也叫做 bot、测试 bot。网络游戏中的 bot 指的是像自动赚取经验值这样的违反使用规则的专用程序，但是在开发初期，由开发人员自己制作的 bot 程序指的是对协议的功能进行整体的测试，判断服务器是否有问题等的命令行程序。

采用自动测试可以大幅提高开发的迭代速度。比如，一次一次地点击地图，或者一次一次地点击敌人来进行攻击，以此来赚取经验值进行升级，等等，实际操作起来相当花时间，但是有了 bot 就可以将这些操作自动化了，程序员在这期间可以去做其他的工作。

2 cli

cli 包括系统、背景、可动物、HUD。画面渲染和鼠标操作等，实际的终端用户所接触到的客户端。

如前所述，即使是最终数据量很大的游戏，到原型开发为止的阶段（以下称为原型阶段），很多情况下也是由 1 个人进行开发的。但是也有由擅长处理渲染和用户界面的程序员，以及擅长实现游戏逻辑的程序员两个人来分担客户端和服务器的开发。在这种情况下，autocli 和 cli 的划分最能发挥效力。

首先，客户端开发的主要工作包括画面的操作和渲染、效果等与通信和游戏逻辑无关的内容。而服务器的实现则是在框架开发完成之后，将几乎所有的时间用在实现各协议的函数以及实现游戏逻辑上。对于相同的部分（比如，服务端的在战斗中受到伤害的处理，和客户端的表现该伤害的处理），服务端和客户端并不同时进行开发，如果可以，最好在进行其他工作时，在其他时刻进行开发。

在这种情况下，如果能确保在 autocli 中对各协议在服务器中实现的行为都进行了测试，那么客户端的负责人员在通信部分的开发阶段中，就可以将 autocli 的协议调用部分的代码作为“应用实现”来参考，同时加入到自己的代码中。

这样一来，就不需要同时进行这两者的开发，工作的分配可以更为灵活。

- autocli 和 cli 合并在一个程序中的例子

  开发原型时，由一个人负责客户端开发和服务器开发的情况下，也有考虑将 autocli 和 cli 合并在一个程序中进行实现的。autocli 和 cli 在通信部分的实现上基本是共通的，所以如果将它们合并在一个程序中，可以大幅减少重复。

  在将 autocli 和 cli 综合在一个程序中时，将通信部分作为共同使用的部分，完全不进行画面渲染就可以了。如果不进行画面渲染，1 台机器中可以同时运行数十至数百个客户端，这样可以模拟大量的访问来进行负载测试。

- 用于负载测试的客户端开发方法

  开发用于进行负载测试的客户端的方法有如下这些。

  ❶ 使用多个游戏会话，从 1 个客户端进程访问服务器。1 台机器也可以运行多个客户端进程

  因为 1 台设备造成 1000～5000 个连接这样的高负荷，所以可以对服务器的最大性能进行测算。但是测试结果只限于将某些统计信息（差错率等）显示在标准输出中。有时，游戏服务开始后，可能会发生“由登录认证的高负荷所引起的系统宕机”，为了降低其发生的可能性，需要对后端服务器进行负载测试，在进行包含后端服务器在内的系统整体的负载测试时使用这种方式。

  ❷ 使用 1 个会话，从进行最低程度的窗口显示的 GUI 执行程序访问服务器，1 台机器启动多个客户端进程

  1 台机器造成 50～100 个连接的负荷。还可以在每个窗口中，对最低程度的信息（角色状态和周围的地形等）进行图示。因此，可以比统计信息更详细地把握服务器实现的正确性。这种类型的自动测试用于在游戏服务的运营中确认服务器是否碰到了预料之外的事件，以及用于检测进行非法活动的用户的活动。

  ❸ 在 1 台服务器上启动多个客户端程序（包括面向终端用户的画面渲染），在客户端上加入某些进行自动操作的结构（比如，使用 Lua 的脚本结构），给服务器造成负荷

1 台机器最多建立 10 个连接。在这种情况下，主要从用户角度进行自动化测试。

对于这里的 autocli，原型开发时所需的自动测试就是 ❶ 或者 ❷。❸ 在实际的客户 端开发中需要 [41] 。

> [41] 这个测试客户端最终能将渲染功能分离，在原型阶段之后的阶段中也可以用来进行负载测试。

专栏　每一步的进度管理形式

我们对框架、原型、整体框架实现这几个步骤中所需的要素进行了总结，下面简单介绍一下推荐的形式。

- **文件文本：在步骤 1～2（框架阶段和原型阶段）中最适合。**
- **xls 文件：在步骤 2～3（原型阶段和整体框架的实现阶段）中最适合。**
- **项目管理工具：最适合步骤 3（整体框架的实现阶段）之后。**
- **跟踪管理工具：最适合步骤 3（整体框架的实现阶段）之后。**

3 协议

在协议项中，根据列出的顺序依次实现 RPC 的各函数。通过依次实现其中的函数，可以把握整体的进度。在一般的 MMORPG 中，最初需要实现 20～30 个左右的协议。

4 ID

在 ID 这一项中，对背景、可动物体、物品等游戏内容所需的 ID 进行定义。在 *K Online* 中，预计在框架阶段定义 10 个左右，原型阶段扩展到 100 个左右，而产品版中将增至数千个。

5 gmsv

在 gmsv 中，为了实现游戏的策划内容，需要编写必要的逻辑，但是，基本上，终端用户也能看到游戏内容，所以应该在协议中列出，这里只编写与协议不重复的部分。

- gmsv 的源代码的规模 / 行数

在 *K Online* 中，预计框架阶段的代码在 2000 行以内，原型阶段的在 4000～5000 行以内，商业版的则在 3 万～5 万行以内（错误处理占了一大部分）。

事实上，对于这里代码的行数，说是预计，其实更有一种“如果不控制在这个程度内就糟了”的含义，这是一种设计上的指导方针。在 *K Online* 中，gmsv 的代码是使用 C++ 来编写的。为了编写健壮的代码，使用 C++ 也有很多必须注意的地方。而且 gmsv 包含了所有的游戏逻辑，最为复杂，加上玩家终端还可能会通过互联网直接、大量地发送包含恶意数据的信息，因此，如何健壮地编写这个进程可以说是 *K Online* 开发项目的关键之处。在 *K Online* 的项目中，从头编写的 C++ 代码的量必须尽量控制在最小程度。

在 *K Online* 中，物品和技能数都达到了“数千以上”，敌人的种类则达到了“数百以上”，可见具有庞大的游戏策划量。如果单从这个数量来进行简单估算，每种物品必须在 1 万～3 万行左右，但是笔者并不认为通过这么几行就能把各个功能实现完整。这里并不这么做，为了尽量将 C++ 代码量控制在最小程度，需要尽可能采用一些方法来避免直接编写 C++ 代码，比如从数据文件中读取，或者使用 Lua 等语言来定义 gmsv 的开发专用语言等。

出于这些考虑，应该牢记“即使是最终版，也应该将 gmsv 的代码控制在 3 万～5 万行以内，否则就糟了”这点，以此来估算代码量。

实际上，在笔者曾经开发过的游戏中，基本上每次都通过引入 DSL 来控制 C++ 的代码量。与其说 DSL 是代码，它其实更像数据，所以，通常使用用 Microsoft 的 Excel 等制定的易于从 C++ 读入的 CSV（Comma-Separated Values）、XML、YAML（YAML Ain't Markup Language、Yet Another Markup Language）、JSON 等，或者使用 Ruby 和 Python 用的模板引擎来自动生成 C++ 代码。从扩展性的角度来看，笔者喜欢“使用代码来生成代码”，但是在除了程序员，还有很多其他人员参与制作游戏数据的情况下，使数据（而非代码）成为主要部分，通常这样更为安全。适当地结合在一起使用会更好。

6 dbsv

在原型阶段，虽然后端服务器全部都不需要予以实现，但是应该加入 dbsv。持久化的结构会影响到游戏逻辑以及协议的调用顺序，所以要尽

早发现问题。

4.14.5 “不实际运行起来是不会理解的！”——游戏开发的特殊性

本书中，我们在原型阶段之前放置了框架阶段，但在实际的游戏开发中，制作人和项目经理会提出更为细化的要求“希望将可玩版本细化发布”，要尽可能频繁。对于策划人员来说，也有坚持“玩过之后提出意见”的人。在至此为止的游戏开发过程中，要时常一边游戏一边加以评价，不断更改规范，思考“这样真的没关系吗？”，在确认这个问题的同时进行开发。

这项修正是很频繁的，只修改程序代码的情况也很多，对于程序的注释和规范文档，在更改代码时不用每次都进行修改，偶尔要对用以整合已实现内容的“追加工作”进行总结，如果是小规模的游戏，那么这种更改需要每天进行几次。100 人规模的游戏，则每周对运行的部分进行确认，并推进开发。

因此，如前所述，在游戏中，无法先定义要素，设计，然后再开发框架。即使是那些使用了 Unreal Engine 等全功能型中间件的游戏，肯定也是这样，必须在最短的时间内提交可供试玩的版本。必须设置公司内部的试玩版、alpha1、alpha2 这种细化的里程碑。游戏开发或许是不适合采用瀑布模型的开发项目的极端例子。

典型的失败 ——企业中的网络游戏开发

企业中的网络游戏开发，典型的失败之处有如下这些。

❶ 不是可玩状态

a. 1 个人可以进行游戏，但两人以上就不行了

→网络实现方面的欠缺和故障

b. 可以多人进行游戏，但是动作非常迟缓，明显有损游戏体验

→性能和通信算法不充分

❷ 可玩，但是不像游戏

a. 只是某些模拟的物体在行动

→开发要素不充分

b. 不是预期中的类型

→比如应该是动作类的游戏，但是开发完成后不像个动作游戏。对策划的理解不充分

c. 由于通信的问题导致游戏逻辑变得不完整，以致游戏无法正常进行

→讨论不充分，技术不足

❸ 可玩，也确实像个游戏，但是质量（可玩性）欠缺

a. 数据和程序的数量不足

→讨论不充分，技术不足

b. 数据和程序的质量不足

→讨论不充分，技术不足

对于网络游戏的开发项目，❶❷ 中，“网络游戏中是否具备特有的技术”是瓶颈所在。在 ❸ 这个阶段中，其技术要点是用来制作大量数据的编辑工具和开发支持工具、策划、为了达到游戏平衡而采用的测试体制等，这些相当于一般的大规模游戏开发中的技术。

通过本书所说的“框架”的开发，可以解决上面 ❶ 中的问题，达到 ❷ 的阶段。如果进行“原型”开发，则可以解决 ❷ 中的问题。

因此，接下来我们就打算着手开发 *K Online* 的框架，试着把握其流程。笔者将依次介绍实现的每一个步骤。通过以下介绍，就能对“客户端→服务器→协议→服务端→客户端”这整个的流程有所了解。那么下面我们就来一起看一下吧。

专栏　C/C++ 以外的语言

MMOG 的服务器开发中有哪些合适的编程语言呢？老实说，笔者也认为用 C/C++ 来编写健壮的代码很够呛，也该想个办法了吧……但是决定性的编程语言还没有出现。MMOG 服务器特有的条件具有如下这些特征。

- 确保要在数百 MB 至数 GB 的内存下运行。
- 玩家数千，可动物体数百万以上，这么多的对象并行运行。
- 网络的输入输出每秒发生数千～数万次。
- 要求稳定而高速的响应。

由于这些特性，笔者所钟爱的 Python 和 Ruby 等轻量级语言基本都无法使用。能用 Rails 这样的框架来开发 MMOG 的服务器就好了，但是……

轻量级语言不行，那 Java、或者 C#、Scala、Erlang 等语言呢？它们比 C++ 更易写出健壮的代码。但是使用 VM 的处理体系只有在 CPU 的处理上可以高速执行，但是在对操作系统的网络和磁盘等进行输入输出的频率（不是吞吐量）很高的情况下，由于在输入输出前后需要进行严格的安全性检查，所以执行速度大幅下降，现阶段还很难投入使用。

但是，在进行大量并行处理以及管理大量对象的情况下，其性能是根据语言处理系统而稳步上升的，希望能早作评估，找到能够轻松开发的编程语言。

话虽如此，考虑到要尽可能采用与客户端相同的语言来开发，可能也只剩下 C# 了吧……由于客户端可能是 Flash 和 iPhone 等，根据市场情况会不断变化，所以或许最好不要作为判断依据更好。

此外 Google 的 Go 语言等不使用 VM 的新的处理体系也非常有意思，但是实效尚少，还不好说。

以上，我们对编程语言进行了总结，使用通信中间件和 DSL 花很大的工夫来彻底控制重新编写的 C++ 代码量，但是笔者的本意在于将来能使用其他的语言更轻松地进行编程。

4.14.6　框架的整体结构

我们来看一下编写完成之后的框架，框架目录下有如下这些内容。

- SDL：在 cli 中使用的开源渲染库
- boost：在 cli/gmsv 中使用的开源 C++ 通用库
- cli：cli 的所有源码
- dbsv：dbsv 的所有源码
- gmsv：gmsv 的所有源码
- proto：协议的定义文档、以及生成的源文件
- vce：通信中间件

上面这些内容中，笔者编写的是 cli、dbsv、gmsv 和 proto。至于 SDL、Boost 和 VCE（参考本节专栏）相关的目录都是现有的库（以及头文件）。

cli 中的文件

cli 目录下的文件如下所示。

- **用于构建**
  - Makefile
  - Makefile.bak
- **源文件**
  - app.cpp：游戏程序所需的所有过程
  - app.h：上述文件的声明部分
  - font.cpp：用来渲染使用 SDL 的英文数字字体的类
  - font.h：上述文件的声明部分
  - kcli.cpp：定义用来接收服务器的协议调用的函数
  - kcli.h：上述文件的声明部分

- sdlmain.cpp：SDL 的 main 函数的封装
- sprite.cpp：用于渲染画面上的 Sprite 的库
- sprite.h：上述文件的声明部分
- util.cpp：算术运算等必要的实用函数的定义
- util.h：上述文件的声明部分

- 数据文件
  - fonts/：渲染使用 SDL 的英文、数字所需要的位图字体文件
  - images/：背景和角色等图像文件

游戏画面如图 4.33 所示。图 4.33 对背景（草地和岩石地），以及作为可动物体的玩家角色（白色）和敌人（黑色）进行了渲染。背景的渲染特意弄得比画面尺寸小。这样就能看到是否可以在正确的时刻从服务器获取到地形数据（在正式版本中要扩大）。

图 4.33　框架的画面

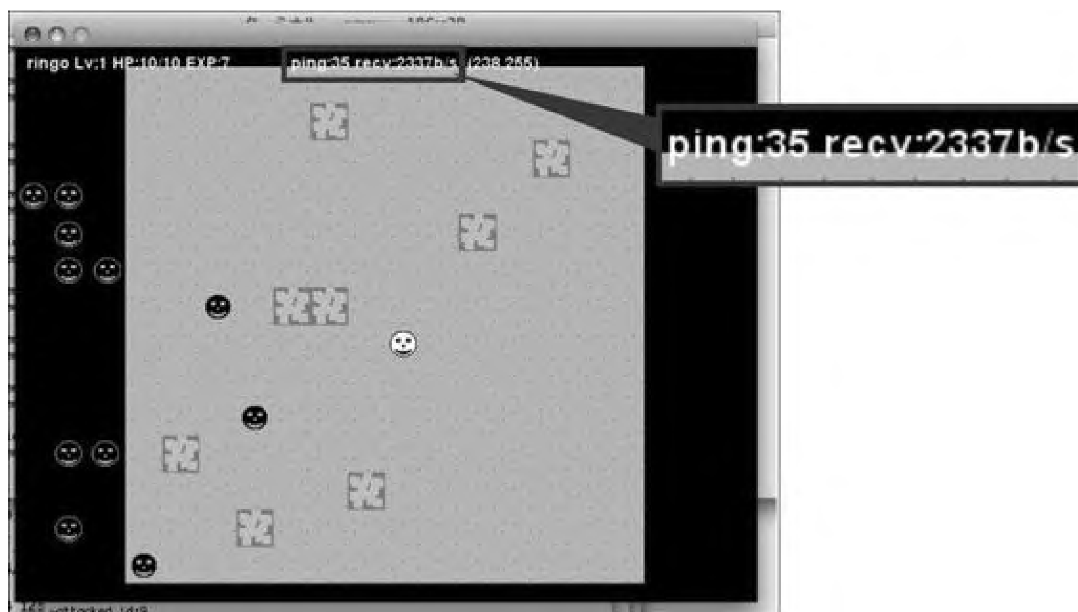


图 4.33 所示的画面上部显示了当前角色的状态。右上部分显示了 ping 值，也就是从客户端发送 ping 消息之后直到服务器返回为止的时间。

在局域网中运行时，ping 值在 30～50 左右就没有问题。recv 值“recv ：2337b/s”表示 1 秒内收到了 2337 字节。动态读取地形数据时，该值会瞬间上升，什么也不做的情况下该值就很小，2337 是读取瞬间的值。继续下去的话，在这个游戏的实现中需要 2337×8 = 20kbit/s 左右的带宽。

专栏　VCE 是什么

VCE 是笔者曾经经营的公司 Community Engine 面向 MMO 类型的游戏而开发的通信中间件。包括相当于 libevent 的部分和用于 RPC 存根代码生成的 IDL，不仅是 PC，还能生成对应于 Flash 和各种游戏机的代码。

现在，Square Enix 拥有 VCE 的著作权等知识产权和所有权，如今也有在一部分游戏中使用。本书为了提高和振兴行业的技术水平，在本书（包含附加条款）所指定的范围内，特别提供许可。购买了本书的读者可以访问本书序文中的 URL，根据技术评论社网站的协议界面，接受其中的附加条款，然后在该网站上获取 VCE（由技术评论社提供）（详情请参照 p.vii）。

gmsv、dbsv、proto 中的文件

gmsv、dbsv、proto 目录下的文件如下所示。

- **gmsv 中的文件**
- **用于构建**
  - Makefile
- **源代码**
  - climain.cpp：测试 bot 客户端。编译后就是 autocli
  - climain.h：上述文件的声明部分
  - common.h：gmsv 和 cli/sutocli 共享的定义。定义地图的最大尺寸和瓷砖等可以共享的数据
  - floor.cpp：定义用于管理背景地形的 Floor 类

- floor.h：上述文件的声明部分
- gmsvmain.cpp：gmsv 的主程序。在这里编写初始化函数和主循环（无限循环）。此外，编写用来接收 dbsv 远程函数调用的函数
- gmsvmain.h：上述文件的声明部分
- id.h：实现用于在世界中分配唯一 ID 的“ID 池”机制
- movable.cpp：用于定义可动物体的行为的 Movable 类
- movable.h：上述文件的声明部分
- sv.cpp：定义最先接收来自 cli 和 autocli 的函数调用的函数
- sv.h：上述文件的声明部分
- util.cpp：定义计算 hash 值的函数等整个服务器将会经常用到的实用函数
- util.h：上述文件的声明部分
- zone.cpp：定义对服务器进行区域分割时，用来管理 ID 和服务器地址的类
- zone.h：上述文件的声明部分

* 文档
  - spec.txt：在编写代码之前准备的规范文档
  - todo.txt：编程时使用的工作列表和记录文件
* dbsv
* 用于构建
  - Makefile

- 表定义文件
  - k_table_def.py：以该文件为出发点，自动生成多个源代码（除此之外的所有源代码）
- 用于自动生成源代码的脚本
  - dbgen.py：实际进行自动生成的脚本
  - settings.py：内容为空，但 Django 要求具备该文件，所以放在这里
- 用于自动生成源代码的模板
  - cltemplate.cpp：用于生成自动进行 dbsv 单元测试的 bot（与 gmsv 的 bot 不同）源代码的模板。Django 读取该文件和 k_table_def 文件来生成 dbclmain.cpp
  - cltemplate.h：上述文件的声明部分
  - svtemplate.cpp：用于生成 dbsv 服务器本身实现的源代码的模板。生成 dbsvmain.cpp
  - svtemplate.h：上述文件的声明部分
  - template.sql：以这个模板文件为基础生成 k_table_create.sql
  - template.xml：以这个模板文件为基础，生成用来定义 gmsv 和 dbsv 之间的传输内容的 dbproto.xml 文件
- 链接使用的源代码
  - qm.cpp：安全生成 SQL 查询、用来防范注入攻击的查询生成工具的实现文件
  - qm.h：上述文件的声明部分
  - util.cpp：编写计算数组元素个数的宏等的实用工具
  - util.h：上述文件的声明部分

- **自动生成的源代码**
  - k_table_create.sql：用来初始化 *K Online* 数据库所需的 SQL。对于 mysql 命令，可以从标准输入读入该文本文件来初始化数据库
  - dbproto.xml：定义 dbsv 和 gmsv 之间的通信内容。通过 VCE 的 gen 进一步生成 dbproto.cpp、dbproto.h
  - dbproto.h：被 dbsv 和 gmsv 使用，定义 gmsv 和 dbsv 之间的传输内容的类的声明文件
  - dbproto.cpp：上述文件的实现部分
  - dbsvmain.cpp：dbsv 服务器的实现主体，包含 main() 函数
  - dbsvmain.h：上述文件的声明部分。gmsv 也使用该文件
  - dbclmain.cpp：用来自动测试 dbsv 的测试程序的主程序
  - dbclmain.h：上述文件的定义文件。gmsv 也使用该文件（实现在其他地方）
- proto
  - auth.xml：定义 authosv 和 gmsv 之间的传输内容
  - k.xml：定义 gmsv 和 cli 之间的传输内容
  - log.xml：定义 gmsv 和 logsv 之间的传输内容

4.14.7　以怎样的顺序来编写代码

在实际的编程中，不管怎么样都要牢记“尽快使游戏达到可以运行、可以游戏的状态”，根据这一点来进行开发。

在开始编写代码时，想象一下代码的整体结构，其中包括 3 个阶段。

❶ **代码结构尚未确定。**

❷ 代码结构确定后只要对增加常量种类等进行数量上的增加。

❸ 进行少量的错误检测、异常事件的处理等的收尾阶段。

只有在阶段 ❶ 中还不存在可运行的游戏。在 ❷ 和 ❸ 这两个阶段中，可以在试玩可运行的游戏时，不断对其进行修改和改进。因此，为了尽快达到可运行状态，必须尽早完成阶段 ❶。

在 MMOG 的开发中，要想尽早完成阶段 ❶ 的最好的方法就是：首先编写决定了客户端和服务器之间如何进行通信的“协议定义”。因为实现了这些协议后，游戏服务器和客户端实现也就基本决定下来了。

如果熟悉网络游戏开发，只要看一下协议文件，就能大致了解服务器和客户端是如何实现的了，还能判断出是否会发生什么重大问题。

4.14.8　首先编写协议定义文件 k.xml——开发流程1

那么，我们就先来粗略地编写一下用来定义 cli 和 gmsv 之间的传输内容的协议定义文件 k.xml。为了使 *K Online* 可玩，必须编写所有所需的协议定义，但是这里还要记住一点，不仅要开发可供人们进行游戏的程序，还要在此之前开发用来自动进行游戏的测试 bot。这里，测试 bot 可以进行整体的游戏测试，所以这里编写玩家用的客户端。

这是因为这里要尽可能更新要频繁改动的内容。

玩家用的客户端需要在接收到协议的函数调用时，更新渲染在画面上的内容。因此，比起不需要这么做的测试 bot，玩家用的客户端具有更大的代码量。比如，测试 bot 中接收函数的代码只需要两行就可以了，而玩家客户端则需要 100 行左右，这种情况是很多的。因此，修改传输顺序和参数内容时的代价非常高。实现“登录之后在地面上来回走动”的测试 bot 要在一开始就进行开发，至少要将协议精炼到在通信顺序和参数方面没什么大问题的状态，之后再开发玩家用的客户端，这样在实现客户端时，对协议的修改频率就会下降，从而客户端程序的开发效率就会大幅提高。

此外，进行自动测试的测试 bot 还有一个优点：在修改协议时所进行的测试，所需的时间比起玩家用的客户端短了很多。比如，“玩家创建一个角色，然后登录游戏，打倒周围的敌人，使用 1 个物品”这样的一系

列操作，如果使用玩家用的客户端，人工操作时，需要花费好几分钟，操作失误的话还会花费更多的时间。

但是使用测试 bot 的话，从命令行启动后，只要 1～2 秒就能完成对这一系列操作的测试了。可以在编译的时间内完成测试。由于除了编程，还有其他的一些工作，所以使用测试 bot 还可以防止忘记掉短期记忆中的宝贵的代码信息。

4.14.9　协议定义的要点

这里我们总结一下协议定义的要点，为了尽量减少开发客户端时修改协议的次数，应该先开发测试 bot 来对协议进行测试。达到“让游戏运行起来”的状态所需的协议必须具有以下这几项。

- **通信连通确认：ping 函数。在密码验证之前确认是否连通。**
- **账号登录：singup 函数。**
- **账号验证：authentication 函数。**
- **角色创建：createCharacter 函数。**
- **登录：login 函数。需要返回值，所以还定义了 ResultCode。**
- **在地面上移动：move 函数和 moveNotify 函数。**
- **攻击：attack 函数和 attackNotify 函数。**

下面我们就依次进行说明。

4.14.10　通信连通确认：ping 函数

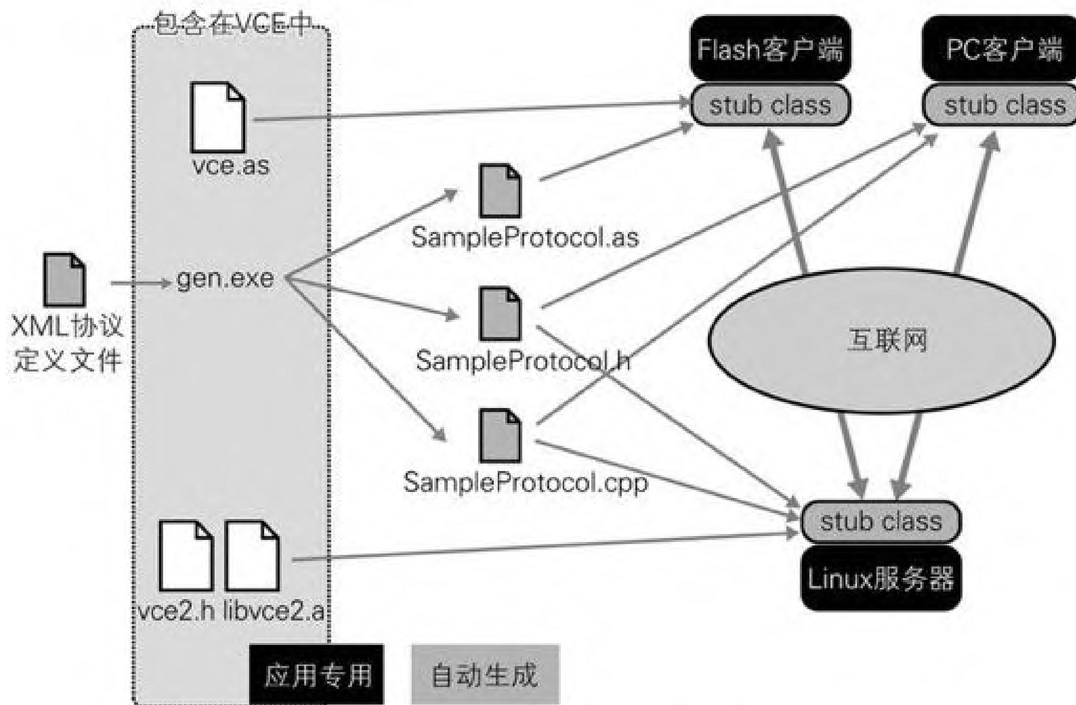
首先是 ping 函数。严格来说并不需要这个函数，但是实现起来也最简单，而且能够立刻确认服务器和客户端是否正在运行，所以还是来实现一下。

```
↓ ping
<method methname="ping" prflow="p2p">        ←加入timer 值的惯用写法
  <param prtype="qword" prname="timestamp" />
</method>
```

在这个定义文件中，methname 表示函数名，这里就是“ping”，prflow 表示函数调用的类型，它的值是“p2p”，它表示进行的是双向调用，可以从客户端向服务器发起调用，也可以从服务器向客户端发起调用。通过 <param> 标签可以指定参数的类型和名字。这里将 qword 这个 64 位整数以 timestamp 为名进行发送。

通过像这样在 k.xml 中编写 ping 函数的定义来输出图 4.34 的流程中所需的存根源代码。该图是 VCE 的例子。

图 4.34　应用开发模型（VCE 的示例）



不仅是 VCE，支持多平台之间通信的通信中间件大多都是从 XML 和某些 DSL 等“基础文件”来输出各种源文件的。

修改 k.xml，在 cli 和 gmsv 所使用的存根文件（kproto.cpp/kproto.h）中，分别定义 bool send_ping(vce::VUint64 timestamp); 函数。

在 cli 和 gmsv 中链接包含该函数的对象文件，如果调用该函数，就会在另一侧（从 gmsv 发送的话就是 cli，从 cli 发送的话就是 gmsv）调用 virtual void recv_ping (vce::VUint64 timestamp) = 0;。

在 cli 中调用 send_ping(vce::GetTime()); 函数，其中指定了当前时刻。vce::GetTime() 是一个将当前时刻以毫秒为单位返回一个 64 位值的函数，该函数将返回的时刻原封不动地发送给服务器。

在服务器侧，接收函数的实现如下所示。

```
void KServer::recv_ping(vce::VUint64 timestamp) {
  send_ping(timestamp);
};
```

也就是说，仅仅是将 cli 发来的时刻再原封不动地送回给 cli。ping 函数对游戏的进行不会造成任何影响，所以不需要进行验证。

cli 接收到 gmsv 返回的消息后，就通过以下的函数计算时间差，将其保存在 lastRoundTripTime 变量中。

```
void KClient::recv_ping(vce::VUint64 t)
{
  if (g_app->state == App::TEST_INGAME) {
    g_app->lastRoundTripTime
      = vce::GetTime() - g_app->lastPingSentAt;
      std::cerr << " now:" << vce::GetTime() << std::endl;
  } ...
}
```

在前面的 cli 的画面（图 4.33）中显示的 ping:35，就是这么显示了该变量的值。

cli 在主循环中定期发送 ping，从而这个值就会不断更新。目前在 cli 中还没有实现，但如果这个值超过了 500，就说明游戏正处于网络延迟相当大的环境中，可能促使玩家退出游戏。

4.14.11 账户登录和账户认证：signup 函数、authentication 函数

接着是账户登录。协议的编写如下所示。这里分成了 c2s 和 s2c。signup() 函数负责处理账户登录，authentication() 函数负责进行账户验证。对于账户登录这一点，或许有人会认为，为了“最低程度地让游戏运行起来”，是不是不需要账户登录？但是，如果一开始就在测试客户端中加入这一功能，除了客户端程序，其他的测试程序都不需要，所以在日常的开发过程中，为了确保在整合整个系统的状态下持续进行测试，这么做是很有效的。

```
<method methname="signup" prflow="c2s" >
  <param prtype="string" prname="accountname" prlength="100" />
  <param prtype="string" prname="password" prlength="100" />
</method>

<method methname="signupResult" prflow="s2c" >
  <param prtype="ResultCode" prname="result" />
</method>
```

signup() 函数只会在之后异步返回成功或失败 [42] 。

> 42 慎重起见补充一下，这里的函数名和下面的 authenticationResult() 等的命名规则与 4.9 节之后出现的 *_result 有所不同。本书中，只有生成 dbsv 的工具使用 _result，除此之外都用 Result。

不仅仅是 signup 函数，所有的协议都是作为异步的函数调用来实现的，对于确实需要某些结果的函数，比如对于 authentication() 的调用，会在“之后”调用诸如 authenticationResult() 这样的返回结果的函数，signup() 也是如此。为此，客户端代码的编写有些麻烦。在 *K Online* 中，验证和购物等 1 对 1 调用的协议总共有 10 种左右，注意尽量避免“锯齿状的后端调用”（锯齿状的时序请参照第 4.8 节），通过 1 次调用来实现。

K Online 的验证很简单，只对账户名和密码进行验证。该玩家是免费用户还是付费用户等信息都在服务器中管理，不需要与 cli 进行通信。

这里通过指定账户名（accountname）和密码（password）来调用认证函数。

其协议编写如下所示。同样分为 c2s、s2c。

```
↓认证
<method methname="authentication" prflow="c2s" >
  <param prtype="string" prname="accountname" prlength="100" />
  <param prtype="string" prname="password" prlength="100" />
</method>
<method methname="authenticationResult" prflow="s2c" >
  <param prtype="ResultCode" prname="result" />
</method>
```

账户名和密码的验证成功之后，就会通过 ResultCode 返回结果。如果结果是 FAIL 则需要通知用户。

访问数据库后，如果确认密码一致，则设置表示认证成功的标识 m_authenticationSuccess。

4.14.12 角色创建：createCharacter 函数

接着是角色创建，角色创建是更新游戏进行信息的一项处理。创建至今所不存在的角色，这会消耗服务器资源，所以这是一项重大的处理。因此需要进行验证。

```
↓角色创建
  游戏逻辑：除了角色名，其他都在服务器中自动生成
<method methname="createCharacter" prflow="c2s" >
  <param prtype="string" prname="characterName" prlength="100" />
</method>
<method methname="createCharacterResult" prflow="s2c" >
  <param prtype="ResultCode" prname="result" />
</method>
```

需要指定的信息只有角色名。今后在能够选择角色职业（战士、魔法师等）时，还要增加参数。但是目前，达到可玩状态才是先决条件，之后不管增加多少参数都可以，所以现在不予实现 [43] 。

> [43] 就目前来说，signup 函数已经很充分了，但是当然在之后的开发阶段中还有很多要增加的内容。它有无限的可能性，比如加入 Facebook 版、iPhone 版的标志，等等。

gmsv 接收到 createCharacter 函数时，像下面这样检查 m_authenticationSuccess 的值，验证失败的话就直接忽略。这里不返回错误信息是因为，如果验证出错，应该在调用 authentication 函数时就会发生，现在验证通不过，却调用 createCharacter 函数，不改造 cli 的话，这是不可能的，是非法访问。对于非法访问，在之后的收尾阶段中的“增加对运营有帮助的日志输出”这一阶段中实现就可以了。在最初的阶段中不需要。

```
void KServer::recv_createCharacter(const char *characterName)
{
  if (!m_authenticationSuccess) return;
...
```

4.14.13 登录：login 函数

接着是登录。

```
↓角色登录
<method methname="login" prflow="c2s" >
  <param prtype="string" prname="characterName" prlength="100" />
</method>
<method methname="loginResult" prflow="s2c" >
  <param prtype="ResultCode" prname="result" />
  <param prtype="dword" prname="movableID" /> ←登录角色的可动物体ID
</method>
```

到了这一步，角色创建应该已经成功了，通过 characterName 指定角色名，然后登录到游戏世界。登录结果通过 result 来返回，此时返回 movableID 这个作为可动物体的玩家角色 ID。

gmsv 接收角色的登录消息，生成玩家角色，使其存在于背景之上。如果角色不存在，或者内存不足，又或者坐标异常，就登录失败。

```
m_pc = World::allocPlayerCharacter(World::getFloor(dbpc->floorID),
Coord(dbpc->x, dbpc->y ), this);
m_pc->stat = CharStat(dbpc->hp, dbpc->maxhp, dbpc->level, dbpc->exp);
```

gmsv 中的代码如上所示，基于从数据库读取出来的信息来生成角色。

4.14.14 在地面上移动：move 函数、moveNotify

使角色移动的 move 函数如下所示。

```
↓使角色移动
<method methname="move" prflow="c2s" >
  <param prtype="int" prname="toX" />
  <param prtype="int" prname="toY" />
</method>
```

首先，move 函数是在角色移动时调用的。通过 toX、toY 来指定想要前往的位置（到达地点）。在 *K Online* 的参照物 *Runescape* 中，用鼠标点击远处的地方，角色就会自动移动到该处。但是，移动时并非是平滑地移过去的，而是以方格为单位来移动。在 *K Online* 中，背景采用格子的方式来管理，只能像将棋那样以格子为单位来移动。此外，格子中可以容纳多个角色（1 个格子只能有 1 名角色时，就会经常发生“走不过去”的这种使人感觉不快的现象和一些异常情况）。

比如，如果 1 个格子为 1，向右移动记为 +X，向下记为 Y+，那么（1, 0）就表示向右前进 1 个格子。如果是（3, 2）的话，在允许斜着移动的情况下，最短的路径就是“右下、右下、右”。

以方格为单位

像这样以方格为单位是为了使服务器的处理高速化。以方格为单位的碰撞检测是一种速度最快的方法，经常被使用。比如，墙壁、敌人、植物、门等不能通过的物体在策划上是必需的，但是如果以任意尺寸的矩形集合来管理，需要循环的次数就和周围存在的个数成正比，但是如果以方格为单位来记录“是否存在不能通过的物体”的标志，就与个数无关，不需要循环，可以在一定的处理时间内完成检测。配置了多扇门等物体之后，为了实现“不能通过的处理”，可以说这是最轻量的处理方法。

在 *K Online* 中，预计每秒向服务器发送数千次的移动请求，所以需要使用最轻量级的方法。

路径搜索和实际的移动处理

在 *K Online* 中，我们已经决定以方格为单位进行碰撞检测，接着必须判断移动路径的搜索和实际的移动处理是在服务端进行，还是在客户端进行。

从结论来讲，*K Online* 采用的方法是：路径搜索在客户端自动进行，实际的移动处理则在服务端进行。

在达到这个设计之前，从最简单直接的方法开始，到最充分的方法为止，一共经过了 4 个阶段的研究。为了对实际开发进程进行充分把握，这里将研究过程也一并包含进来加以说明。

首先，*K Online* 中的“移动路径”如图 4.35 所示，是由上下左右以及 45 度斜向的 8 个方向所组合而成的。图 4.35 所示的是从 (5, 5) 移动到 (8, 7)，为此需要移动 3 步。

图 4.35 *K Online* 中的移动路径

其次，“实际移动”指的是是否更改角色的坐标。比如，如果在服务端进行实际的移动处理，那就意味着要在服务端判断是否可以通过该方格。

- **阶段 1：玩家自己进行移动路径的计算，实际的移动也在客户端进行，服务器无限制的接受**

  使用这种方法时，由于只能根据点击次数来移动，如果点击速度很快，就无法限制角色的移动速度，这样就会破坏游戏平衡性。在 *K Online* 的游戏策划上，需要将角色的移动速度限制在一定范围内。虽然主要的游戏内容是与敌人作战，但是既然敌人的移动是一步一步地以某一速度在格子中行走的，那么如果玩家的移动速度没能控制在一个良好的范围之内，游戏的平衡性就会被破坏。*K Online* 中的理想速度是 1 秒内移动两步。

- **阶段 2：与阶段 1 相同，但是在服务端加入一项限制：限制移动请求的接收频率**

  既然客户端每 0.5 秒移动 1 次是最理想的，那么如果发送速度比这个速度快就移动失败，这样就可以了吧。比如，以图 4.35 为例，对于 move(6, 6); move(7, 7); move(8, 7);，每 0.5 秒发送 1 条。但是在这种方法下，由于客户端与服务器是存在于互联网之上的，由于数据包的发送延迟，从服务器角度来看，这 3 个消息可能会跳过 0.5 秒的间隔而同时到达。因此，这种方法不能限制速度。

- **阶段 3：与阶段 2 相同，但是在 move 函数中加入客户端的发送时间，一起发送给服务器**

  比如 move(6,6, 时刻 0); move (7,7, 时刻 1); move(8, 7, 时刻 2); 这样在各个移动请求中增加一个表示时间的参数，然后在服务端对其时间差进行计算，如果过快，服务端就拒绝接收。但是，*K Online* 是向玩家分发程序（Windows 的 .exe）的，如果该程序被破解了，玩家就可以发送虚假的时间。这样一来就会放任作弊行为，所以是个很大的问题。

综上，看来简单直接的方法不能处理实际的移动问题。那么，我们来进一步讨论一下。

- **阶段 4：服务端进行路径搜索和移动处理**

在这种情况下，客户端只发送最终想要到达的目的地（比如：move(8, 7)）。服务器接收到这一信息后，就搜索到达目的地的路径，得到 (6, 6)、(7, 7)、(8, 7) 这 3 步移动路径。然后，服务器基于这个路径，每 0.5 秒使角色移动 1 步。每次移动时，向客户端发送新的坐标。由于所有的处理都在服务器中进行，所以服务器的处理就会相当多，可以说这是一大缺点。如果地形复杂，路径搜索的处理负荷也会上升，如果角色增加，每秒都要对所有的角色确认是否到了设定的时刻，这样的处理量实在太大了。没有什么办法了吗？

- **阶段 5：在客户端中只进行路径搜索中的搜索处理**

  K Online 的地下城中有很多道路狭窄的迷宫，所以想在这里下点工夫。路径搜索处理本身就算碰到作弊，也不会有实际的损害。因为最短路径的计算本来对玩家来说就是有利的，除了想要攻击服务器，否则用户不会想要特地在客户端程序中加入对自身不利的因素。在使用这种方法的情况下，客户端就会以服务器发送过来的地形数据为基础来计算移动路径，然后像 move([6, 6], [7, 7], [8, 7]); 这样，将多个表示移动方向的数组合并起来发送给服务器。这样，虽然通信量增加了一些，但是在最短路径的计算方面（这方面的处理量很大），服务器的处理负担轻了不少。

经过这些阶段，就得出了之前所述的结论，采用“路径搜索在客户端侧自动进行，实际的移动处理则在服务端进行”的方法。

移动路径的发送方式 ——优先发送最终结果

最后还有个细节需要注意，移动路径的发送不使用相对坐标，而是采用绝对坐标。之前说过，*K Online* 的世界的面积为 4000×4000。发送这个坐标需要一个 12 位的数值。C 语言的话，就相当于一个 2 字节的 short 类型的数据。但是以相对坐标来发送时，1 个字节就够了。

比如，在刚才那个例子中，自己的角色正位于 (5, 5) 处，移动到 (8, 7) 时，可以发送 move(3, 2) 这样的相对坐标。自己的位置为 (2000, 2000) 时，不是发送 (2003, 2002)，而是发送(3, 2)，这样似乎可以节约数据量。

但是，发送相对坐标时，如果角色在移动途中又点击了别的地方，那就会移动到意料之外的地方。在以下这样的情况下就会发生这种意外之事。

- Step0：自己所处位置的绝对坐标为（5，5）。
- Step1：玩家点击处的绝对坐标为（8，5）。
- Step2：客户端发送相对坐标（3，0）。
- Step3：送达服务器，角色开始移动。
- Step4：移动 1 步后，服务器向客户端发送新的坐标（6，5）。
- Step5：（在接收到新坐标之前）正在移动的玩家再次点击同一个位置（8，5）。
- Step6：客户端发送相对坐标（3，0）。
- Step7：虽然服务器已经移动了 1 步，但又收到了（3，0）。
- Step8：之后根据这进一步的指示移动 3 步，一共就前进了 4 步。

一共前进 4 步，这并非出于玩家的意图。

Step4 和 Step5 之间由于互联网的通信存在延迟。虽然只延迟 100 毫秒左右，但是在这期间，玩家还是会经常再次点击。

在 Step4 和 Step5 之间追加一步处理“在角色的移动停止之前，客户端忽略玩家的再次点击”，可能有人会认为这样一来就不会有问题了。但是，人们在 1 秒内一般只能点击 2～3 次，如果必须在停止之后才能点击，就会导致“走几步停一下”，实在令人沮丧。

那么，或许可以这么想：在客户端首先输入 1 次点击，将“已输入”这个信息保存着，角色一停下就立刻发送出去。但是在这种情况下，在点击新的位置之后，从 gmsv 到 cli 的 1 次网络往返传输的时间内，角色还是会停下来。

像这样在存在网络延迟的情况下采用相对坐标的方式来进行操作是很难调整的，很多情况下都无法很好地解决。角色的移动就是真实反映了这个问题的例子。因此，应该尽量围绕“优先发送最终结果”这一点来进行设计。

移动结果的通知范围

那么，作为角色移动的结果，每 0.5 秒从服务器向所有看到该角色的客户端（一定范围内的所有客户端）发送以下数据包。

“范围”指的是 *K Online* 中 20 个左右的方格所涵盖的范围。在 *K Online* 中，角色的身高大约 1.5 米，方格的长宽为 1 米，画面的显示范围为长宽 20 米左右的四方形。所以方格为 20 个。

20 米的显示范围是根据敌人的移动速度等策划内容以及画面的渲染性能等来决定的。这里的 1.5 米也好、20 米也好，突然用真实世界的单位或许会让人困惑，但这是为了便于参与开发的开发人员进行交流而经常使用的方式。在 3D 建模软件中，人物只是一系列数值的集合，比如，如果一般人听到“100 米”，也可以感觉到大致的距离，尤其是在有人物角色出场的游戏中，人物角色最基本的高度大致为 1.5～2 米，像这样不单单使用数值，还会加上“米”等单位来称呼。1.5 米的话，数值部分也跟浮点数 1.5 一致。比起说“此人身高 9 万 2000”，还是说“92 厘米”更易理解。

此外，是将浮点数 100 表示为 1m，还是将 1.0 表示为 1m，有时会根据所使用的渲染库和开发工具的类型而改变。

moveNotify 函数

基本上，move 函数并不是“调用 1 次必须返回 1 次结果”的 1 对 1 函数，调用 1 次可能 1 次也不返回，也可能返回 10 次，特别是对于自己以外的角色，即使自己什么也不做，其他角色的移动结果也会不断地发送过来。这种具有推送性质的函数在作为 s2c 的基础上，再在函数名后面加上接尾词 Notify。

```
↓ moveNotify：可动物体移动时发出的通知
<method methname="moveNotify" prflow="s2c" >
  <param prtype="int" prname="movableID" />
  <param prtype="MovableType" prname="typeID" />    ← MOVABLE_*
  <param prtype="string" prname="name" prlength="100" />
  <param prtype="int" prname="x" />
  <param prtype="int" prname="y" />
  <param prtype="int" prname="floorID" />
</method>
```

movableID 表示“什么移动了”。typeID 用来指定移动物体的图像编号。比如，如果是“龙”这种敌人，就传入常量值 ENEMY_DRAGON。name 是名字，当移动物体是 NPC 时使用。x、y 表示位置，floorID 表示可动物体位于哪个层面上。这是在坐标系有多个的情况下（比如地面上为 0，地下城 1 层为 1），表示使用哪个坐标系的 ID。在往返于地下城和地面上的情况下，gmsv 和 cli 之间会产生时间差，使用这个参数的话，可以避免将应该存在于地下城中的敌人显示在地面上。

- characterStatus 协议

前面我们已经说过，在 moveNotify 中发送名字、位置和图像编号。如果是 NPC，这样也许就足够了，但是敌人还具有等级、HP 等其他很多信息。

在 *K Online* 中，在使用 moveNotify 来更新可动物体的状态时，我们采用一种经常使用的方法：使用其他函数异步来获取经常变化的、信息量大的部分。首先，如下所示，定义 characterStatus 协议。

```
↓ characterStatus：取出指定ID 的可动物体的状态
<method methname="characterStatus" prflow="c2s" >
  <param prtype="int" prname="movableID"/>
</method>

↓ characterStatusResult：返回结果。ID 不存在的话什么也不返回
<method methname="characterStatusResult" prflow="s2c" >
  <param prtype="int" prname="movableID"/>
  <param prtype="CharacterStatus" prname="charstat" />
</method>
```

指定 movableID，使用 CharacterStatus 结构体获取其状态。CharacterStatus 结构体定义如下：

```
↓ struct PlayerProfile：传送各玩家的详细信息
<struct strctname="CharacterStatus">
  <member mbtype="string" mbname="name" mblength="50" />
  <member mbtype="dword" mbname="hp" />
  <member mbtype="dword" mbname="maxhp" />
  <member mbtype="dword" mbname="level" />
  <member mbtype="dword" mbname="exp" />
  <member mbtype="dword" mbname="mapID" />
```

```
  <member mbtype="dword" mbname="x" />
  <member mbtype="dword" mbname="y" />
</struct>
```

mbtype 是结构体的成员变量的类型。hp 和 maxhp 等是成员变量的名字。完整表示敌人状态所需的值全部定义在这个结构体中。

cli 接收到 moveNotify 时，首先在内存中进行搜索，如果判断出这是还没有接收消息的角色，那就新生成一个角色，然后在画面上表现出来。此时，在最初的 moveNotify 中，只要接收到了消息，就能最低限度地在画面上的适当位置显示出某个图像，使玩家早一点看到当前的状态。

- send_characterStatus(movableID)

与此同时，调用 send_characterStatus(movableID)，进一步向服务器请求 HP 等更为详细的信息。HP 等信息稍微晚点发送过来也不要紧。这样一来，可以将玩家第一次看到敌人的时间缩短 100 毫秒。

```
void KViewClient::recv_moveNotify(vce::VSint32 movableID,
                    k_proto::MovableType typeID, const char *name,
                    vce::VSint32 posx, vce::VSint32 posy,
vce::VSint32 f)
{
  std::cerr << "kv move:" << Coord(posx,posy).to_s() << " id:"
            << movableID << std::endl;
if (movableID == g_app->myMovableID) return;
if (g_app->getMovable(movableID) == NULL) {
  CliMovable *m = new CliMovable(movableID, typeID, Coord(posx,
posy)); assert(m);
  g_app->movmap[movableID] = m;
  send_characterStatus(movableID);
} else {
  g_app->movmap[movableID]->setCoord(Coord( posx, posy));
  g_app->movmap[movableID]->typeID = typeID;
 }
}
```

- characterStatus 结构体的用途

随着今后开发的深入，characterStatus 结构体会渐渐扩大。其中会加入敌人的装备信息、状态异常等各种各样的信息。那时，在发送频率最高的 moveNotify 中自然要避免每次发送这些信息。

顺带一提，在与敌人的战斗中，HP 和 MAXHP 等信息会时刻发生变化。如果是频率非常高的情况下，可以在别的地方定义“只改变 HP”这样的小型函数，在造成伤害时只调用该函数。在开发过程中随时测定所用的带宽量，如果出现什么问题，只要将 CharacterStatus 细分就可以了。

这里，我们将 characterStatus 函数从 moveNotify 函数中分离了出来，但如果只是“开发最低程度的可运行状态”，也许没有必要进行分离。但因为之后确实还是需要这么做的，而且实现起来也简单，所以就顺便实现一下。

attack 函数、attackNotify 函数

接着，我们来看一下攻击敌人的函数。

```
↓ attack：攻击敌人
<method methname="attack" prflow="c2s" >
<param prtype="int" prname="movableID" />
</method>

↓ attackNotify：向其他玩家通知攻击了敌人
<method methname="attackNotify" prflow="s2c" >
<param prtype="int" prname="attackerMovableID" />
<param prtype="int" prname="attackedMovableID" />
<param prtype="int" prname="damage" />
</method>
```

指定 movableID 进行攻击。在 *K Online* 的策划内容中，敌人与玩家之间的位置关系不会对战斗结果造成什么影响。在有些游戏中，会以某种形式将敌人与玩家之间的位置关系作为信息来使用（在这种情况下，通常敌人不会剧烈移动）。比如，从后面攻击造成的伤害更大，或者在攻击体型庞大的敌人时，如果没有击中正中位置，伤害就会降低，等等。在 *K Online* 中并不使用这些信息，所以在 attack 函数中，只通过 movableID 指定敌人（可动物体）的 ID 来进行攻击。

攻击的结果通过 attackNotify 发送给所有的客户端。在该函数的参数中指定谁（attackerMovableID）攻击了谁（attackedMovableID），伤害量（damage）为多少。

- **将结果处理为抽象度很高的信息，以尽量少的位数来发送**

  在 cli 中，通过指定攻击方（玩家角色）和被攻击方（敌人），就可以表示“攻击者朝着敌方逼近”。对于敌人经常在移动中的 *K Online* 来说，这是个很重要的要素。不管是敌人还是玩家，都是混杂在一起，一边移动，一边攻击。在这种情况下，由于通信会有所延迟，所以玩家 A 所看到的画面中与敌人的位置关系，与玩家 B 所看到的经常会不一致。如果在服务器发送的 moveNotify 消息到达客户端时，可动物体“瞬间”飞到该位置的话，这种不一致的情况就不会发生了，但是为了使动作更为自然，通常进行“平滑移动”处理。

  这样的移动过程通常花费 0.2 秒至几秒的时间来表现，但此时，在这段时间内，各个可动物体所在的位置都还“赶不上”服务器上的最新状态。如果每个玩家的网络延迟各不 相同，攻击数据包和移动数据包的接收时刻有的近、有的远的话，最后看到的位置就会不一样，这样可能就会造成一种不协调的感觉，“为什么从那个位置攻击可以命中敌人呢？”。

  比起“尽量反映出服务器状态”，更应该关注于最终结果是什么，即最终的攻击结果是什么，以此来考虑协议内容。

  在 MMOG 中，“必须将结果处理为抽象度很高的信息，以尽量少的位数来发送”，这一点在 3.5 节中也有所涉及，在这里也能体会到吧。

4.14.15 编写 gmsv/Makefile——开发流程 2

那么，最低要求的协议已经编写完成了，接着开始编写 gmsv/Makefile。在 Windows 下使用 Visual Studio 等集成环境的情况下是不需要编写 Makefile 的，而且在技术上也不怎么重要，所以在此省略。

其关键是，autocli 和 gmsv 要同时构建。这样一来，对协议进行了修改后，作为测试程序的 autocli 和被测试的 gmsv 可以同时报出构建错

误，从而可以通过依次解决这些错误来进行开发。依赖编译来开发可以有助于减轻程序员的记忆负担。

4.14.16　从示例中复制 gmsv/climain.cpp 和 gmsvmain.cpp——开发流程 3

接着，我们来编写 gmsv 和 cli 的基本代码。

我们从最上面开始按顺序来实现协议的定义。首先是 ping 函数，ping 放在 VCE 的 sample 文件夹的 genping 文件夹中。其他中间件也一定有与 ping 相关的示例文件。

将该文件复制下来，用 ping.xml 这个作为示例的协议定义文件替换 k.xml 这个游戏用的文件，然后进行构建。由于缺少了 move 函数的接收函数等一系列的函数，构建后会提示链接错误，目前暂时用空函数来实现，使其不报错。解决了构建错误后，第一版的gmsv、autocli 就完成了。autocli 只要实现启动后进行连接，然后定期发送 ping 就可以了。

像上面这样，从“示例的复制”开始积累是一种常规方法。通过实现 ping，gmsv/autocli 的组合最低限度地运行起来后，之后只要在协议定义文件中不断增加所需的协议就可以了。

在 sv.cpp 中实现 signup 函数

ping 之后是 signup 函数，将其添加进来进行构建，报错提示不存在 recv_signup()，下面我们来对其进行定义。signup 函数的最初定义如下所示。

```
void KServer::recv_signup(const char *accountname,
                          const char *password) { }
```

我们知道 cli（autocli）会向服务器发送要登录的账户名和密码，除此之外就没别的了。recv_signup 的最终的实现如下所示。

```
void KServer::recv_signup(const char *accountname,const char
*password)
{
```

```
  std::string pw = makeHashString(std::string(password));
  db_proto::Player pdata(g_idpool->get(), accountname, pw.c_str());
  g_dbcli->send_put_Player(uID, pdata);

m_lastFunction = FUNCTION_SIGNUP;
}
```

这里有一些地方需要解释一下。在后端服务器的访问方面，使用源代码的自动生成（参照后面的专栏）有点复杂，之后将会单独提出来加以说明，这里先简单提一下。

首先，为了避免在数据库中以明文的方式保存密码，我们使用 makeHashString 函数将密码不可逆地散列。比如，有一个密码是“hogehoge”，通过 makeHashString 函数将其变换成“c7b47e41d31b6878 1c2c14e7d0dd965e32718f01”这种值。

接着，通过构造器对 db_proto::Player 结构体所需的数据进行初始化。使用构造器就不会遗漏掉某个数据项的初始化了。构造器的第 1 个参数是整个世界中唯一的玩家 ID 编号，g_idpool 是用于分配该 ID 编号的全局对象。第 2、第 3 个参数则传入账号名和变换成 hash 值的密码。

- send_put_Player 函数

  为了访问名为 g_dbcli 的 dbsv，使用初始化了的 pdata 的引用，通过 send_put_Player 函数发送。这个函数是用后面要讲到的工具自动生成的，用于异步访问 dbsv。dbsv 与 gmsv 需要异步通信。这是因为，在与 dbsv 进行通信期间，gmsv 需要不间断地处理各种来自 cli 的请求（比如使 NPC 移动等）。因此，发给 dbsv 的发送函数在其内部并不实际进行通信，而是在发送给 dbsv 的通讯缓存中存放所要发送的内容，然后立刻返回。该函数的原型如下所示。

```
bool send_put_Player(vce::VUint32 sessionID,Player data);
```

将 uID 的值传递给第 1 个参数 sessionID。uID 是与 cli 进行连接（Session）时被分配的固有 ID，之后当 dbsv 的函数调用异步返回时，用这个 ID 来找出是哪个 cli 的请求。dbsv 原样返回这个 uID。

之后，存放的数据通过主循环的 vce::Poll 函数实际传递给操作系统。

cli 调用的最后一个函数是 signup 函数。

- recv_put_Player_result 函数

从 dbsv 返回的值由以下函数接收。这也是自动生成的函数。

```
void DBClient::recv_put_Player_result(vce::VUint32 sessionID,
                                      ResultCode result,Player
data)
{
  KServer *ks = findKServer(sessionID);
  if(ks) ks->db_recv_put_Player_result(result, data);
}
```

sessionID 中传入的是刚才的 uID 的值，使用该 ID 查询与客户端的连接。KServer 指的是作为服务器运行的会话，用 findKServer 函数来查询。

- db_recv_put_Player_result 函数

这里，如果 cli 与 dbsv 在通信过程中连接中断了的话就会报错，所以要确认 ks 是否存在。如果存在，就通知 session : put_Player 操作成功。用来完成这项工作的函数就是 db_recv_put_Player_result 函数。在这个函数中，当最后调用的函数是 signup，并且 dbsv 的返回值为 SUCCESS 时，就向 cli 发送 send_signupResult(k_proto::SUCCESS)。

```
void KServer::db_recv_put_Player_result(db_proto::ResultCode
result,
                                        db_proto::Player data)
{
  if (m_lastFunction == FUNCTION_SIGNUP) {
```

```
    if (result == db_proto::SUCCESS) {
      send_signupResult(k_proto::SUCCESS);
    } else {
      send_signupResult(k_proto::FAIL);
    }
  } else {
    assert(0);
  }
}
```

- **“dbsv1 次往返”的请求**

  这里与 dbsv 的通信，让人感觉是一种非常迂回的实现方法（事实上，写起来很麻烦）。正是由于与 dbsv 进行的异步通信，导致了这种迂回。

  在 gmsv 中，为了处理来自 cli 的请求，需要访问 1 次 dbsv，然后再向 cli 返回 1 次，这种“dbsv1 次往返”的请求每次都必须这么实现。

  但是，从用户登录、角色创建、角色列表获取、验证、拍卖等游戏内容来看，辅助要素（周边部分）的功能受到了限制，这种辅助要素采用基于 Web 的实现，或者实现只针对这些要素的专用程序，以此来回避这个麻烦。但是周边元素不多的情况下，需要进行结构体的版本管理等，程序分离反而增加了复杂度。特别是在不使用 C++，而是使用其他语言，作为 Web 服务来实现的情况下，所要依赖的外部库和框架会急剧增加，可维护性反而变差了。

“dbsv1 次往返”的请求和线程

即使判断将“1 次往返”的功能内置在 gmsv 中，看上去使用线程好像可以用更简单的编写方法，但实际上，因为必须很仔细地处理互斥控制的实现，所以相当麻烦。比如，如果 signup 函数的接收函数使用线程来编写的话，应该如下所示。

```
↓以下是伪代码
void KServer::recv_signup(const char *accountname,const char
*password)
{
  ↓创建线程来进行处理。startThread 函数本身是立刻返回的
  startThread(signupRun, accountname, password);
}
```

```
void signupRun(const char *accountname,const char *password)
{
  std::string pw = makeHashString(std::string(password));
  db_proto::Player pdata(g_idpool->get(), accountname, pw.c_str());
  int result = g_dbcli->put_Player(uID, pdata);   ←实际进行传输、等待、返
回值

  if (result == k_proto::SUCCESS) {
    send_signupResult(k_proto::SUCCESS);
  } else {
    send_signupResult(k_proto::FAIL);
  }
}
```

这么一写，就又简单又明了吧。但是在采用这种写法的情况下，在需要访问所有的全局变量以及线程之间共享的变量（KServer 的成员变量等）时，必须考虑线程之间的互斥控制。比如，访问 g_idpool 中的全局变量。此外，对 g_dbcli 这个 dbsv 的连接，是所有线程共享的 1 个全局变量。dbsv 发送过来的结果是从哪个线程分配的？为了对此进行查询，必须实现使用队列等的互斥控制。此外，除了这里所说的 signup，其他的请求（login 等）中，新分配可动物体，使其出现在地图上等，由于对全局变量的访问很频繁，所以也需要周密地进行互斥控制。

- **单线程 / 多进程结构更好吗**

  对在 signup、login 等“dbsv1 次往返”型的请求中使用线程这一点，笔者就现状作出了如下评价。

  - **写法简单易读。**
  - **1 次往返的请求频率较少，对提高服务器的基本性能没什么帮助。**
  - **由于请求的种类不多，所以未必很花工夫。**
  - **需要实现互斥控制。在不使用线程，依次进行处理的情况下，互斥机制是自动、周密的。**

  因此，可以得出这么一个结论：单线程就足够了。但是，如果策划内容具有以下这些条件的话，那就不要迷惑，果断使用线程吧。

- 大部分处理在线程之间都是独立的，没有共享的变量。
- 请求的种类很多，大部分都是 1 次往返，或者多次往返的。

但是，在这种情况下，或许使用 Django 这样的 Web 应用框架，作为 Web 服务来实现更好。

那么，虽然不使用线程来实现“1 次往返型的请求”，但是线程仍然是一种发挥多核 CPU 性能的方法。MMOG 中的服务器处理性能是非常重要的，所以这个问题总是被提出，但是笔者根据现状所作出的判断是“单线程 / 多进程的结构更好”。理由请参照专栏“gmsv 中的线程使用”。

专栏　dbsv 服务器代码的自动生成——后端服务器实现的简化

本书的 C/S MMO 框架代码中，自动生成了所有的 dbsv 源代码。进行自动生成的工具命名为 dbgen.py。保存在 dbsv/dbgen.py 中。

dbsv 的主要目的是异步访问 MySQL，除此之外，只具有一些进行写入缓存以降低数据库负荷的基本功能。因此，只要像 dbsv/k_table_def.py 文件这样，定义了“访问哪个表的哪个列”这样的信息后，就能自动生成源代码了。这样一来，即使是频繁执行的对数据库表列进行修改，也变得容易了。

接下来，我们简单介绍一下 k_table_def.py 的内容。以下的列表中定义了玩家角色表的内容。

```
tbl = Table(name="PlayerCharacter") // 初始化Table 类，定义表名
tbl.add( Field( name="id", type="qword", primary = True,
auto_increment=True))
// ↑向Table 类的实例添加字段。各个字段中，传入name、type、index、primary、
//auto_increment、size 等组成SQL 各列的定义的信息
tbl.add( Field( name="playerID", type="qword", index=True ))
// ↑玩家角色所附带的玩家ID。1 个玩家可以拥有多个角色，这样可以具有1 对多的关系
tbl.add( Field( name="name", type="string", size=50, index=True))
// ↑（给其他玩家看的）角色名。不唯一（不排他）
tbl.add( Field( name="level", type="word", index=True )) // 角色等级
tbl.add(
Field( name="exp", type="dword" )) // 角色的经验值
tbl.add( Field( name="hp", type="dword" )) // 当前的HP
tbl.add( Field( name="maxhp", type="dword" )) // 当前的MAXHP
```

```
tbl.add( Field( name="floorID", type="dword" )) // 在哪一层
tbl.add( Field( name="x", type="dword" )) // 在地图上所处位置的X 坐标
tbl.add( Field( name="y", type="dword" )) // 在地图上所处位置的Y 坐标
tbl.add( Field( name="equippedItemTypeID", type="dword" )) // 装备着的
物品的类型
db.add(tbl) // 最后，向数据库添加表定义
```

dbgen.py 基于以上信息生成 XML 文件（作为生成 VCE 的 RPC 存根时使用的 IDL）。VCE 基于这个 XML 文件生成协议存根 CPP 文件及其头文件。以下摘录了该 XML 文件一部分。

```
<struct strctname="PlayerCharacter">
  <member mbtype="qword" mbname="id" />
  <member mbtype="qword" mbname="playerID" />
  <member mbtype="string" mbname="name" mblength="50"
mbvariable="true" />
  <member mbtype="word" mbname="level" />
  <member mbtype="dword" mbname="exp" />
  <member mbtype="dword" mbname="hp" />
  <member mbtype="dword" mbname="maxhp" />
  <member mbtype="dword" mbname="floorID" />
  <member mbtype="dword" mbname="x" />
  <member mbtype="dword" mbname="y" />
  <member mbtype="dword" mbname="equippedItemTypeID" />
</struct>
```

首先，上面定义了 PlayerCharacter 结构体。VCE 的生成工具 gen.exe 根据这个定义生成 C++ 头文件。

接着像下面这样定义各个函数。

```
<method methname="put_PlayerCharacter" prflow="c2s" >
  <param prtype="dword" prname="sessionID" />
  <param prtype="PlayerCharacter" prname="data" />
</method>
<method methname="put_PlayerCharacter_result" prflow="s2c" >
  <param prtype="dword" prname="sessionID" />
  <param prtype="ResultCode" prname="result" />
  <param prtype="PlayerCharacter" prname="data" />
</method>
```

以上是用于保存角色信息的 RPC 定义。

RPC 从 gmsv 调用，在 dbsv 接收。

```
↓从各个字段获取
<method methname="get_PlayerCharacter_by_id" prflow="c2s" >
  <param prtype="dword" prname="sessionID" />
  <param prtype="qword" prname="id" />
</method>
<method methname="get_PlayerCharacter_by_id_result" prflow="s2c" >
  <param prtype="dword" prname="sessionID" />
  <param prtype="ResultCode" prname="result" />
  <param prtype="PlayerCharacter" prname="data" prlength="100"
prvariable="true" />
</method>
```

上面的函数在 gmsv 调用，指定 id 后进行查询。

根据字段的类型定义所有这样的查询。因此，RPC 定义的函数的总数将大幅上升。但由于全部都是自动生成的，所以实际上不用花多少工夫。此外，除了实际用到的函数，其他函数都不会被链接，所以服务器的执行文件不会无谓地增大。只是构建时间有所延长……

有所限制的是，使用多列的复杂查询和范围查询等各种 SQL 所具备的能力在此都无法使用。在 *K Online* 的实现中，为了尽量避免数据库成为瓶颈，采取了不依赖于数据库功能的设计，所以这个限制不成问题。

在实际的 dbsv 服务器进程中，接收到上面的 RPC 调用时执行的对 MySQL 服务器进行 SQL 查询的处理代码也同样由 dbgen.py 自动生成。

4.14.17　自动测试客户端 autocli 的实现——开发流程 4

至此，signup 函数的服务端实现就完成了。现在想要启动自动测试客户端（autocli），自动调用 signup 函数，测试一下其结果。

下面就来概要地说明一下测试客户端的实现。测试客户端的运行方式如下所示。

- **启动。**
- **向 gmsv 发起连接（只有 1 个会话）。**
- **基于当前的执行状态（TestState state）开始逐个对函数进行测试。**
- **测试的状态进行到最后，测试结束。**
- **只要有 1 个条件没有满足，就调用 assert。**

测试的状态迁移

autocli 内部的测试进行状态定义如下。示例代码中使用较为原始的方法（switch 结构）来管理状态迁移，但在进行更为复杂的测试的情况下，使用状态机（State Machine）[44] 来实现更好。

> [44] 面向对象中常称为状态模式（State Pattern），而游戏行业中常称为状态机（State Machine）。

```
↓测试的状态迁移
typedef enum {
  TEST_INIT = 0,     ←连接后的初始状态
  TEST_PING_SENT,    ←发送了ping()
  TEST_PING_RECEIVED,          ←接收到了ping() 的返回
  TEST_SIGNUP_SENT,            ←发送了signup()
  TEST_SIGNUP_RECEIVED,        ←接收到了signupResult()
  TEST_AUTHENTICATION_SENT,    ←发送了authentication()
  TEST_AUTHENTICATION_RECEIVED, ←接收到了authenticationResult()
  TEST_CREATECHARACTER_SENT,   ←发送了createCharacter()
  TEST_CREATECHARACTER_RECEIVED,    ←接收到了createCharacterResult()
  TEST_LISTCHARACTER_SENT,          ←发送了listCharacter()
  TEST_LISTCHARACTER_RECEIVED,      ←接收到了listCharacterResult()
  TEST_LOGIN_SENT, ←发送了login()
  TEST_LOGIN_RECEIVED,    ←接收到了loginResult()
  TEST_INGAME, ←正在进行游戏的状态
  TEST_LOGOUT_SENT,       ←发送了logout()
  TEST_SESSION_CLOSED,    ←接收到了logoutResult()
  TEST_FINISHED ←测试结束，不管什么时候结束都是良好的状态
} TestState;
```

autocli 的 main() 函数

极端简化之后的 autocli 的 main() 函数如下所示：

```
int main(int argc, char *argv[])
{
  KClient *kcli = new KClient();   ←创建客户端实例
  g_vceobj->Connect(kcli,"localhost",9000);  ←连接gmsv

while (true) {       ←进入无限循环
  kcli->Poll();      ←每次对客户端实例进行状态确认
  if (vce::GetTime() > (testStarted+10*1000)) break; ←测试开始10 秒后退
出循环
  }
  assert(kcli->evaluate());   ←评估测试结果
  return 0;
}
```

连接 gmsv，进入无限循环后在 10 秒内通过 Poll() 函数不断对状态进行确认，收集这 10 秒内的信息，过了 10 秒后退出循环，通过 evaluate() 函数对结果进行评估。评估结果主要对协议的 RPC 函数调用的次数进行计数，检查是否缺少。如果圆满通过了到此为止的大约 30 个以上的 assert() 断言并退出 main() 函数的话，就可以说测试成功了。

因此，autocli 通过 assert() 函数得到的结果就是，是 abort（中断），还是正常结束。过了原型阶段之后，最后可能会实现数百个以上的 assert。根据笔者经验，每次实现新的功能都要向 autocli 添加测试，强烈建议不要中断对 assert 的添加。

- **KClient::Poll() 函数**

```
bool KClient::Poll()
{
  switch(state){
  case TEST_INIT:
          send_ping(vce::GetTime());
          state = TEST_PING_SENT;
          break;
```

```
  case TEST_PING_RECEIVED:
          send_signup(localAccountName.c_str(),
localPassword.c_str());
          state = TEST_SIGNUP_SENT;
          break;
  case TEST_SIGNUP_RECEIVED:

send_authentication(localAccountName.c_str(),localPassword.c_str(
));
          state = TEST_AUTHENTICATION_SENT;
          break;
  case TEST_AUTHENTICATION_RECEIVED:
< 继续类似的组合…>
  case TEST_LOGIN_RECEIVED:
          state = TEST_INGAME;
          break;
  case TEST_INGAME:
          ingameSender();
          break;
  case TEST_FINISHED:
          break;
  default:
          assert(0);
  }
  return true;
}
```

首先，autocli 开始执行后，KClient::state 的值从 TEST_INIT 开始。

在上面的代码中，首先发送 ping()，然后状态变为 TEST_PING_SENT。

- **ping()**

gmsv 接收 ping() 的代码如下所示：

```
void KClient::recv_ping(vce::VUint64 t)
{
assert(state == TEST_PING_SENT);
state = TEST_PING_RECEIVED;
}
```

如果能成功接收，状态就变为 TEST_PING_RECEIVED，然后进入下一个状态。这样，在 KClient::Poll() 函数中就开始调用 signup() 函数，然后就这样一直继续下去。

- TestState

enum 类型的 TestState 中，大致分为“三角状的 API 调用”（三角状的时序）和“游戏客户端的运行状态”这两种类型的定义。ping、signup、authentication、createCharacter、listCharacter、login、logout 是三角状的 API 调用，其结构是：调用 xx() 后，从 gmsv 返回 1 次 xxResult()。TEST_INGAME、TEST_SESSION_CLOSED、TEST_FINISHED 是客户端的运行状态。

首先，在 KClient::Poll() 函数中，只有在 TEST_INGAME 状态下时，才会在每次循环时调用 KClient::ingameSender() 函数，实现移动、攻击自己周围的敌人这种“登录游戏后，持续进行的处理”。autocli 中会持续检测周围是否存在敌人，存在的话就进行攻击。通过这样的处理，在 gmsv 中就会执行实际的游戏处理，对敌人发起的攻击命中敌人并将其打倒后，就能获得经验值。如果 autocli 打倒敌人获得经验值之后中断连接，数据库中就会保存经验值已经增加了的角色。

处于 TEST_INGAME 状态下时，过一段时间之后就会发送 logout。gmsv 一旦接收到 logout，就会中断连接。autocli 接收到该中断消息后，就会变成 TEST_SESSION_CLOSED 状态，最后再变成测试结束的状态 TEST_FINISHED。这两个状态是表示 autocli 测试结束的状态。

* * *

autocli 测试的范围是“在 autocli 启动 1 次的范围内进行测试”。所以并不进行同一个角色下次继续登录游戏的测试。这项测试之后是需要的，但是在开发的初期阶段并不需要。

以上是 autocli 的整体实现，下面我们来介绍一下各个协议函数的调用。

signup() 函数

ping() 函数已经讲过了，所以下面我们来看一下 signup()。

autocli 中，发送 signup() 时的代码如下所示。

```
std::ostringstream idss;
idss << "ringo" << time(NULL) << "." << g_id_generator;
localAccountName = idss.str();
localPassword = std::string("testpass");

g_id_generator ++;
send_signup(localAccountName.c_str(), localPassword.c_str());
std::cerr << "signup" << std::endl;
state = TEST_SIGNUP_SENT;
```

通过时刻的秒数来自动生成账户名，密码固定为 testpass。调用 send_signup() 后，向 gmsv 进行传输。

gmsv 对其进行处理，将结果返回给 autocli。接收函数如下所示。

```
void KClient::recv_signupResult(ResultCode result)
{
  assert(state == TEST_SIGNUP_SENT);
  assert(result == SUCCESS || result == ALREADY);
  state = TEST_SIGNUP_RECEIVED;
}
```

这里加入了两个 assert。按照期望顺序下的状态迁移，通过测试 state 的值来检查是否从 gmsv 返回。如果调用了 1 次 signup，但是却从 gmsv 返回了两次以上，那么可以认为这是个 bug。

接下来，当然是对账号创建的处理结果进行测试。如果不是 SUCCESS 或者 ALREADY 的话，就 assert。

* * *

就这样对接收函数所需的测试项依次进行测试。其要点就是，依序进行 signup、authentication、createCharacter、listCharacter、login，然后达到游戏进行状态。

- **KClient::ingameSender() 函数**

  游戏进行状态就是 TEST_INGAME，此时每次循环都调用 KClient::IngameSecder() 函数。对该函数的内容进行简化后如下

所示：

```
void KClient::ingameSender()
{
  ↓事情1：移动的测试
  int dx = -1 + ( random() % 3 );
  int dy = -1 + ( random() % 3 );
  send_move(myCoord.x + dx, myCoord.y + dy); ←移动到附近的1 个随机位
置处
  sendCounter[FUNCTION_MOVE]++; ←对移动次数进行计数

  ↓事情2: 由于记录了自己以外的可动物（TestMovable），所以对其进行攻击
  std::map<vce::VUint32,TestMovable*>::iterator it;
  int cnt=0;
  for (it=movmap.begin(); it != movmap.end(); ++it) {
    TestMovable *m = (*it).second;
    if (m->typeID == k_proto::MOVABLE_GOBLIN) {
      send_attack(m->id); ←可动物是GOBLIN 的话就发起攻击
      sendCounter[ FUNCTION_ATTACK]++; ←记录攻击次数
      cnt++;
      if (cnt == 15) break; ←为了1 次不要攻击太多次而进行调整
    }
  }

  ↓事情3：还没有请求持有物列表的话，请求1 次
  if (recvCounter[FUNCTION_ITEMNOTIFY] == 0) {
    send_item();
    sendCounter[FUNCTION_ITEM]++;
  }
}
```

上面这段代码中做了 3 件事。

- **事情 1：随机移动**
- **事情 2：随机攻击周围的敌人**
- **事情 3：获取持有物列表**

“事情 1”很简单，使玩家角色的坐标 x、y 在 -1～+1 的范围内变化，将其传递给 move 函数，定期发送。

“事情 2”也很简单。接收到 moveNotify 函数时，由于敌人的移动通知对 std::map 记录了坐标和种类，所以对其进行循环，发送

send_attack()。因为从 gmsv 发送过来的应该只是距离较近的敌人，所以对所有这些敌人发起攻击。

“事情 3”其实是三角状的 API 调用（三角状的时序）。请求玩家角色的持有物列表，服务器只是简单地响应这个请求。但是这个 API 调用在游戏过程中随时都会异步发起，而且在 gmsv 侧，持有物列表发生变化时，也会从 gmsv 突然向客户端发送。因此，在 ingameSender 中进行测试。

* * *

通过这样来实现 autocli，可以对自动登录 gmsv、四处走动、攻击敌人（自动赚取经验值）进行测试。熟练了后，上面这样的实现用不了一两天就能完成了。在这个阶段中，还完全看不到游戏画面，但如果只是持续几天，其他的开发成员并不会担心。

专栏　gmsv 中线程的使用

在 MMOG 行业中，关于如何使用线程这一点议论颇多。大致来看，作为 ❶ 实现对 dbsv 的异步通信，❷ 发挥多核 CPU 的性能这两方面的手段，线程是相当有用的。❶ 在 之前已有论述，❷ 是作为一个问题遗留下来的。

这里，线程是指“共享进程的内存空间而运行着的多个操作系统本地线程”。在 1 个进程中，虚拟并行运行的线程无法发挥多核 CPU 的性能，所以不包含在内。

如何在各部分中使用线程呢？

- cli：游戏客户端的实现中，在音效、网络、渲染、AI 等功能中使用线程，使这些功能 各自独立，从而发挥多核 CPU 的性能，并且使代码更为易读。出于这种目的而使用的线程数通常在 2～10 个之间。

- gmsv 以外的服务器：基本上都是作为对 DBMS 的游戏网关的功能，所以并没有必须发挥多核性能的负荷，所以不需要使用线程。

- gmsv：大量用到 gmsv 的 CPU 的处理有以下两种。

→使 NPC 等可动物体移动的处理。

→处理几千个以上的 cli 通过网络发送过来的大量请求（1 秒内几万次）。

如果可以使用线程来进行这些处理，分散到各个内核来执行，就能提高性能。那么要怎么实现呢？如今，可以采取的选择有：为了有效利用像线程和 Mac OS X 的 Grand Central Dispatch 这样的本机搭载的 CPU 的多核性能，而采取“设备内部的分割处理”，以及将处理分割为多个进程，由远程机器上运行着的进程进行协调的“跨设备的分割处理”。

可选方法有如下 3 种。

❶ 只使用设备内部的分割处理。

❷ 只使用跨设备的分割处理。

❸ 既使用设备内部的分割处理，也使用跨设备的分割处理。

我们比较一下这 3 种方法（表 C4.A）。

图 C4.A　分割处理的选择

	❶ 只使用设备内分割	❷ 只使用跨设备分割	❸ 同时使用两种方法来分割
编程所花的工夫	小	小～中	大
最大扩展	取决于设备的内核	取决于设备的台数	取决于设备的台数
大量的同时崩溃	有	无	有

首先，在“编程所花的工夫”方面，通常是表 4.A 中 ❶ → ❷ → ❸ 的顺序。❸ 由于同时使用了 ❶ 和 ❷，所以所要花的工夫比 ❶ 和 ❷ 加起来的程度还要大。这是因为软件的复杂度并不是简单的相加而得，这

种情况下，由于同时使用了两种并行处理的实现方法，所以其复杂度远远大于 1+1=2。

接着，我们来看一下“最大扩展”。选用方法 ❶ 时，为了进行扩展，除了增加设备的内核数，再无其他方法。因此，只能在 1 台物理设备之内进行扩展。在 2010 年，一般能用（成本适中）的服务器设备和 laaS 云服务大致也只能使用 2～8 个内核，但是如果将来能发展到 32 个内核，甚至是 128 个内核，也许这个方法也很不错。

再来看一下“大量的同时崩溃”，在使用线程和 Grand Central Dispatch 的设备内分割处理中，多个线程共享同一块内存区域，由于它们之间互相不受保护，所以 1 个线程甚至可以破坏另一个线程的内存区域。因此，程序的 1 个异常就会影响其他线程的执行。

比如，在一个 32 核的设备中，使用方法 ❶ 来构建游戏服务器时，理论上性能最大能提高 32 倍。如果 1 个线程能处理 100 名的同时连接玩家，32 个线程就能处理 3200 个同时连接。但是此时，如果某 1 个线程发生了内存访问冲突，或者对内存内容造成了破坏，或者发生了什么异常，其他的 31 个线程全都会停止，对玩家造成的损害相当严重。选用方法 ❷，因为进程具有单独的虚拟内存空间，所以不会发生前面这种情况（受到损害的玩家限制在 100 人以内）。

由此可见，方法 ❶ 不仅没有充分的可扩展性，在健壮性方面也很薄弱。因此，必须选择方法 ❷。

此外，如果使用方法 ❷ 来实现，某一台设备搭载了多个内核的情况下，还具有能够有效利用这一性能的优点。这种情况下，在操作系统方面，比起使用线程，进程之间进行切换的成本很高，所以会导致处理性能有所下降。

综上，在如今的经济条件下，使用方法 ❷ 来实现 *K Online* 可以说是最合适的，但是如果今后的经济条件发生变化的话，还需要重新考虑。

4.14.18　图形客户端 cli 的创建和运行确认——开发流程 5

至此，autocli 已经完成了，那么接着我们开始进入图形客户端（cli）的开发。按顺序展开客户端程序的开发，反复进行功能扩展，然后完成

作为在终端用户的 PC 上实例化的应用程序。通过在实现了 autocli 之后着手开发实际的 cli，我们可以顺利地进入到实际的 cli 的开发阶段。

SDL

那么，cli 的最大功能就是为终端用户渲染画面，接收终端用户的操作。这是它与 autocli 的最大区别。

cli 使用 SDL（Simple DirectMedia Layer[45] ）。SDL 中附带了 2D Sprite 渲染以及键盘输入的示例程序。这些程序在开发设备上运行，所以在基本的差异（渲染和输入）上并没有问题。

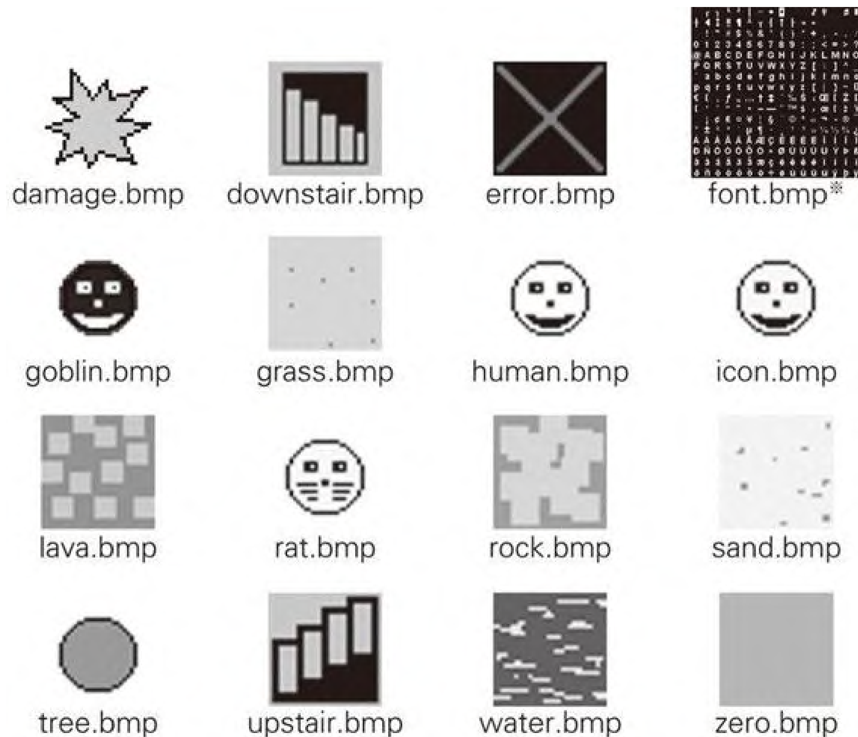
> [45] 一套开放源代码的跨平台多媒体开发库，使用 C 语言写成。SDL 提供了数种控制图像、声音、输出入的函数，让开发者只要用相同或是相似的代码就可以开发出跨多个平台（Linux、Windows、Mac OS X 等）的应用软件。——译者注

绘制图形

接着，我们来绘制必要的图形（参见图 4.36）。在游戏行业中，我们常会为原型命名，这次，叫做“四边形和圆形”这种版本就足够了。可以自己绘制，也可以在网络上找一些免费图像，我们参考从网上下载下来的笑脸符号，使用 Photoshop[46] 绘制临时的带有纹路的四边形和圆形。笔者本人是一名程序员，没有绘画方面的才能，所以就这么将就下就可以了。

> [46] 在自己的程序中加入设计人员完成的图像文件，不是一次就能完全到位的，所以必须一点点尝试并修改。但一般来说，大多程序员在绘画方面没什么天赋（虽然对图像本身还是有不错的审美能力）。

图 4.36　准备的图像（图像文件一览）



※ 图像提供（font.bmp）:Marius Andra。

如图 4.36 所示，文件名和内容大致上是一致的。

只有 font.bmp 这个文件最大，网络上公开了用来渲染使用了 SDL 的英文和数字字体的类，为了使用这些类，需要相应的字体图像，这个文件就是同样公开在网络上的字体图像[47]。使用像 SDL 这样广为普及的工具可以从网络上获取大量好用的程序，这一点非常有帮助。渲染日语字体时需要其他的程序，这不在本书讨论的范围，所以在此省略。

> [47] 参考 http://cone3d.gamedev.net/cgi-bin/index.pl?page=tutorials/gfxsdl/tut4

不仅是对示例所附带的图像，也对“自己制作的 bmp”能否顺利渲染进行了确认。

运行确认

之后，在“整合整个系统，确保系统处于运行状态”的概念下，赶紧连接至 gmsv。autocli 作为实际运行的客户端而存在，所以就从其中开始复制粘贴。

具体来说就是完整复制下连接服务器后发送 ping 的部分、signup、authentication、createCharacter 和 login。加以编译并运行，这就算是运行了一次吧。与 autocli 的差异是，cli 将画面渲染限制在每秒 60 次，而 autocli 有时每秒循环 60 次以上。*K Online* 的策划内容允许 100 毫秒以上的延迟，所以可以忽略这个差异所造成的影响。

进入 TEST_INGAME 状态。在具备了 autocli 的基础上开始着手开发 cli 后，只需 1～2 个小时就可以进入 TEST_INGAME 状态的开发了。如果没有开发 autocli，这里就要加上开发 autocli 所需的时间了。这样一来，就要同时考虑通信问题和渲染问题，恐怕有点伤脑筋了。

进入游戏进行状态后，需要对原本自动发送随机“移动、攻击”的部分进行修改，实现让玩家通过键盘输入来操作。移动的发送方法虽然是设计为在客户端进行路径搜索，但是首先要实现按下上下左右键后，角色朝相应的方向移动一步。这是之后迁移到使用鼠标的正式操作体系的前提。

由于可以用 autocli 来进行测试，实际实现后如果进行了测试，那服务器端就不会有问题，只要对客户端侧进行调试就可以了。由于可以对服务器和客户端分开进行调试，所以负担就减轻了。

实现字体的显示

接着要来实现总会需要的字体的显示。字体的渲染已经完成了，这里需要将自己的角色的坐标等内容通过文字显示在画面中（上部）。此外，测量从向服务器发送 ping()，到它返回所花费的时间，用“ping ：毫秒数”这种方式来显示。最后还要通过“recv ：字节数”来显示过去 1 秒内接收到的数据量。图 4.37 所示的画面的顶部所显示的就是这些信息 [48] 。

> [48] 图 4.37 是完成后的截图的一部分。

图 4.37　字体的渲染实验

总是在画面上显示 ping 值和 recv 值等测定数据的话，就可以立刻发现意料之外的通信延迟和通信量的增加等程序实现方面的任何错误。建议大家在开始编程后，从最初期开始就尽可能显示这些信息。

出现敌人、追赶敌人

在 gmsv 的内部，敌人已经在四处晃荡了。由于敌人每次移动时都会发送 moveNotify，所以从 autocli 将接收部分的代码复制过来，根据通信内容来更改在画面上渲染的敌人的位置。首先做到每次接收后就瞬间移动到该处，实现了之后再改为平滑地移动。

画面上出现敌人后，自己可以进行操作以追赶敌人，所以接着要实现“在行进方向上有敌人的话就发送 attack()”的功能。所要移动的方向上存在敌人的话发送 send_attack()，不存在的话则发送 send_move。在正式版的商用客户端中，点击敌人时就发送 send_attack()。

此时，gmsv 中还没有实现攻击、被攻击的游戏处理，所以按如下顺序来实现。

❶ 定义 attack() 协议

❷ 在 gmsv 中添加攻击处理

❸ 也在 autocli 中对攻击部分的逻辑进行修改，通过测试

❹ 在 cli 中实现“按下 a 键后调用 send_attack()”

❺ 在 cli 中进行测试

协议的定义最先进行。基本上，每次添加功能都是重复这个流程。

打倒敌人，获得经验值

在 gmsv 中实现攻击处理时，同时还要针对角色定义 MAXHP 和当前的 HP、攻击力等基本属性。此外，对于打倒敌人后获得经验值的处理，全部采用固定值来实现，比如，对敌人造成的伤害总是为 3，敌人的 MAXHP 为 10，敌人的经验为 10，升级所需的经验为 100 等。

因为敌人会被打倒，所以要定义 disappear() 协议，使敌人消失，这个协议在客户端也要予以实现。

此外由于可以获得经验值，所以还要定义 characterStatus() 协议，使客户端可以接收变更之后的状态。这也要在画面上部用文字显示出来。比如，“ringo: Lv:1 HP: 10/10 EXP: 7”这样。Lv 表示等级，EXP 表示经验值。

确保以后可以继续进行游戏 ——游戏状态的保存

逐渐推进实现。由于打倒了敌人，获得了经验值而得以升级，所以要将自己角色的游戏状态保存在数据库中，确保以后可以继续玩下去。*K Online* 是 MMORPG，为了从原型开始就能进行试玩，并且不断改善游戏内容，必须保证以后可以继续进行游戏。

由于之前已经开发了一种工具，用来自动生成在数据库中保存数据的代码，所以这里就对 dbsv/k_table_def.py 进行修改，添加用来保存 HP 和 EXP 等信息的列。

4.14.19　框架之后的开发——开发流程、后续事项

至此，说 *K Online* 的“框架”开发阶段告一段落了。从现在开始，就要逐步对协议进行定义，反复在 gmsv、autocli、cli 中添加必要的实现部分。这就是原型阶段。原型阶段的目的是对游戏可玩性的本质部分进行确认。为此，需要进行以下工作。

- **充实游戏设定数据。**
- **增加图像、影像、音效数据，改善游戏界面。**
- **用于多人高效进行游戏试玩的服务器启动和版本管理等的工具。**
- **用于分析游戏结果、查找问题的日志解析工具。**

只是从这个阶段开始，一般的游戏开发流程成为了中心，所以。本章中不涉及这些内容。服务器管理工具、日志解析工具等不管哪种网络游戏都需要的辅助系统，由于并不在原型阶段进行开发，所以将留到第 6 章进行介绍。

4.15 总结

本章以 C/S MMO 游戏开发为基础，介绍了各种类型的网络游戏的开发所共通的开发流程。下一章我们将介绍 P2P MMO 类型的游戏所需的技术。

第 5 章 ［实践］P2P MO游戏开发：没有专用服务器的动作类游戏的实现

没有专用服务器的动作类游戏的实现

本章主要介绍 P2PMO 游戏的开发。详细内容如下。

- P2P MO 游戏的特点和开发策略
- *J Multiplayer* 游戏开发案例的学习——与 K Online 的不同
- P2P MO 游戏的设计资料
- 客户端 / 服务器软件 + 中间件、基本原则
- P2P 游戏 *J Multiplayer* 的实现——正式开始编程
- 支持 C/S MO 游戏的技术 [补充]

前面章节介绍的 C/S MMO 游戏开发中使用了专用服务器，所以从最初的概要设计、流程设计到物理设计就需要考虑服务器的相关问题。

但是对于 P2P MO 游戏来说，因为参与游戏的玩家少、不需要为（游戏使用的）专用服务器付费，所以在概要设计、流程设计、物理设计中需要考虑的情况就大为减少，这样可以加快开发游戏的速度。

本章针对 P2P MO 游戏和 C/S MMO 游戏在设计上的不同，以及那些不需要花费很多精力解决的问题，通过实际的开发流程来向读者进行说明。

5.1 P2P MO 游戏的特点和开发策略

第 3 章已经介绍了 P2P MO 游戏的概要。本章主要整理开发中需要的知识点。

5.1.1　P2P MO 和动作类游戏——游戏的状态频繁发生改变

P2P MO 这种典型的多人游戏的设计主要有以下几种分类。

- ARPG

  例：暴雪娱乐公司的《暗黑破坏神》、卡普空的《怪物猎人》系列

- FPS

  例：id Software 的《雷神之锤》系列

- RTS

  例：微软的《帝国时代》、暴雪娱乐公司的《星际争霸》系列

- **竞速游戏**

  例：任天堂的《马里奥赛车》系列

- **格斗对战游戏**

  例：任天堂的《全明星大乱斗》系列

这类游戏都具有动作游戏的特点。动作类游戏的游戏状态频繁发生变化，会产生大量数据交互，如果使用 C/S 方式实现，对服务器、带宽等成本要求过高，所以从商业角度考虑是不可行的。

因此，即便有使用游戏外挂作弊的风险，P2P 的实现方式从经济角度来说也是合理的。但是，目前服务器硬件以及带宽成本不断降低，这种经济角度的考虑已经变得不那么重要了。所以游戏的数据通信全部通过服务器来传输的方式也在逐渐流行起来[1]。

[1] 5.6 节会介绍这种实现方式。

上述动作游戏都可以使用“同步共享内存”（数据对象同步和事件通信）的方式来开发。这种方式与 C/S 类型的 MMO 游戏中经常使用的 RPC 方式相比，通信量会增加，但是程序开发相对简单，而且可以在游戏的单机部分开发完成后再扩展其网络功能。

共享内存这种方式是 P2P MO 游戏开发中经常使用的特有方法，本章会详细说明。

5.1.2 RPC 和共享内存

在 C/S MMO 游戏中，服务器通过 RPC 方式同步客户端数据进行游戏。P2P MO 类型的游戏使用共享内存方式，它们的区别请参考表 5.1。

表 5.1 RPC 开发方式和共享内存开发方式的比较

	RPC 开发方式	共享内存开发方式
通信量	最小限度	会有多余的通信
CPU 负荷	最小限度	少量多余的运算量
最大同时连接	数千	数十到 100
开发难易度	复杂	简单但是代码量大
通信延迟	没有区别	
游戏类型	即时性低	即时性高
游戏数据规模	数据量大（对象数百万以上）	数据量小（对象数几百到几千）
通信方式	一对多	1 对全部
数据同步的开发时机	在最开始需要开发	可以在单机版之后扩展

通过表 5.1 可以看到，RPC 开发方式面向的是 MMO 游戏，而共享内存的实现方式更适合 MO 游戏。共享内存其实在 C/S 和 P2P 游戏中都可以使用。相对于通信方式来说，我们更应该根据游戏的内容来选择开发技术。

例如，如果要求单服务器同时连接数在数千级别，带宽负荷和 CPU 负荷都应该尽量减少，就只能选择 RPC 开发方式。对于不使用服务器的 P2P MO 游戏，即使单个玩家需要的带宽多一点，也可以使用共享内存的方式来开发。

笔者认为，应该尽量采用共享内存的方式开发游戏。这是因为共享内存相对于 RPC，不需要针对每个操作分别定义函数，开发起来比较容易。但是对于大型 MMO 游戏来说，由于它们使用了大量数据对象，而且 CPU 负荷和服务器的带宽要求都很高，只能选择 RPC 开发方式。随着云计算的使用费逐渐降低，未来也许可以都采用共享内存的方式来开发，但目前趋势还不明显。

RPC 开发方式的相关内容，在介绍 C/S MMO 的章节已经详细描述过了，本章主要对共享内存开发方式进行详细说明。

5.1.3 P2P MO 游戏的特点——和 C/S MMO 游戏的比较和难点

P2P MO 游戏的需求和 C/S MMO 游戏正好相反，这里列举一下它们的不同特点。

- **不需要大量数据交互**

  游戏的相关资源，除了贴图和动画之外一般在几百兆以内，可以全部安装在普通玩家的 PC 中 [2] 。因为不需要联网也可以进行游戏，所以需要安装全部的数据和资源。

- **配置信息对玩家是可见的**

  因为游戏的配置信息是安装在玩家的电脑中，所以可以在本地文件中看到配置内容，游戏运行时也可以跟踪内存信息，而且还可以通过逆向工程破解全部的游戏内容，这些都是无法预防的。

- **不能严格保证游戏数据变更的安全性（可以作弊）**

  正如前文所述，玩家可以通过修改内存数据来作弊，要保护游戏数据的安全性是很困难的。这样造成的结果就是：原本需要玩家花费数百小时，长时间培养游戏角色的游戏方式，以及具有精心

创造的宏大游戏世界，能让玩家在不知不觉中消耗大量时间和精力的游戏方式就难以实现了。对玩家来说，这种经过长时间游戏积累的数据一旦损坏或者丢失将是难以承受的损失，游戏体验的品质也就不能得到保证。导致的结果就是 P2P MO 游戏的游戏时长较短并且以游戏内容为中心。

- 不能进行 P2P 连接的 NAT 网络环境有很多

玩家之间不能通过端对端方式进行网络连接（TCP 或者 UDP 的连接方式）的情况有很多。比如在公司局域网游戏时，或者使用公寓楼、大学、当地的有线电视网络等通过路由器构建的 NAT 公共网络。在日本，这种情况大概占到 10%～30%。另外还有因为 P2P 通信量的异常增大而禁止直接发送数据的情况。这种案例最近有所增加。利用这些网络环境的玩家根据所在地区的差别又会出现各种各样的情况。游戏内容针对的目标玩家不同、上市时间、游戏主机等各种因素的复杂组合导致了 NAT 问题的发生。为了解决诸如 UPnP[3] 、SOCKS[4] 、UPD Hole Punching（后面章节会介绍）等问题考虑了各种方法，但是还是不能完全解决。IPv6 网络普及之后也许情况会有所好转，但前景还不是很明朗。

- 不能简单地更新游戏

P2P 游戏的更新比较麻烦，一般要向玩家发布软件更新包并安装到 PC 来升级硬盘中的游戏程序。所以可能会有玩家在玩老版本的游戏。Steam 这类新的下载平台虽然可以解决这个问题，但是因为在不能访问网络的情况下一个人也可以玩，所以玩家往往选择不升级继续在离线的情况下玩老的版本。

- 不方便结合社交网络等网络服务

因为游戏程序不联网也可以玩，所以游戏结果就不能发布到社交网络上。

- 玩家掉线 [5] 的情况比较多

因为没有一直在线的服务器，游戏的通信依赖于玩家的机器，如果玩家突然关掉电源或者突然结束运行中的游戏就会导致通信中

断。这种情况，需要同步玩家之间的游戏数据，同步显然不能花很长时间，所以如果数据量过大是不可行的。

> [2] 可以通过 DVD 或者 Steam（第 6 章会说明）之类的下载服务进行完整安装。

> [3] Universal Plug and Play。PnP(Plug and Play) 的网络版协议，http://www.upnp.org/ 。

> [4] 在传输层进行访问控制的安全协议。参考 RFC 1928。

> [5] 玩家掉线也叫 churn。

5.1.4 P2P MO 游戏的优点

上面列举了 P2P MO 游戏复杂的地方，不过这类游戏也有很多优点。

- **延迟较少**

  所有的操作都是直接在玩家的机器之间通信，因为不通过服务器，数据不需要在互联网上通过各种路由器来传输，所以网络延迟较少。这样就比较适合动作类游戏的实现。

- **游戏服务器的带宽负荷低（甚至没有负荷）**

  P2P 游戏中的数据都不需要通过服务器传送，所以服务器带宽的负荷很低，甚至没有负荷。只有 MMO 游戏带宽负荷的几十分之一到几百分之一，几乎到了可以忽略不计的程度。

- **游戏服务器硬件成本低（甚至零成本）**

  正如前文所述，因为不需要管理游戏数据的服务器程序，所以也就不需要配置专门的游戏服务器，也就不存在服务器运营的成本。

- **服务器维护期间也可以正常进行游戏**

  服务器（辅助系统的服务器）在维护期间，玩家也可以正常进行游戏。不过积分榜或者玩家匹配等功能还是需要正常的网络连

接，这些辅助系统相关的内容会在第 6 章介绍。

5.1.5 从概要设计开始考虑［多人游戏模式］

考虑到之前所描述的那些和 C/S MMO 游戏的不同，在概要设计一开始就要有多人游戏的意识，必须针对这些特点进行设计。游戏设计师如果只是专注于自己的设计而忽略 P2P 技术开发这个前提，在之后实际开发时就会遇到各种问题，导致项目失败，甚至重新开发也不行，遇到这样严重问题的案例有很多。

开始编程时，P2P 游戏可以先制作单机版本。之后再使用共享内存技术开发多人游戏部分。但是，如果游戏设计开始就没有考虑到多人游戏的情况是不行的。

5.2 *J Multiplayer* 游戏开发案例的学习——和 *K Online* 的不同

本章以 *J Multiplayer* 这个假想游戏为案例，学习如何设计以及采用共享内存方式的实现，与第 4 章的 *K Online* MMORPG 游戏的不同点也会进行对比说明。

5.2.1 *J Multiplayer* ——和 *K Online* 的比较

J Multiplayer 是一个共享内存开发方式游戏的学习案例，让我们选择适合的游戏方案来进行设计。这里假定以《暗黑破坏神》这种 ARPG 为基础，因为笔者的相关经验较丰富，而且也容易和 MMORPG 游戏进行比较，具体的设计内容之后说明。为了便于与第 4 章的 C/S MMO *K Online* 的游戏设计内容进行比较，这里也使用同样的设计流程。

5.2.2 P2P MO 游戏开发的基本流程

P2P MO 游戏开发的基本流程和 C/S MMO 相同。不过和 MMO 相比，服务器端代码相对较少，开发的迭代速度也比较快。此外还可以采用单机游戏的开发方式，一边开发一边试玩。从这个角度来说，P2P MO 游戏开发的风险相对较低。

5.2.3 P2P MO 游戏开发的交付产品——开发各个阶段需要提交的资料

P2P MO 游戏开发项目的资料 / 交付产品中不包括服务器相关（游戏程序）的部分，DB 设计图等除了辅助系统几乎不需要。另外服务器端的单元测试，运营工具的准备等与 C/S MMO 游戏相比规模都很小，所以在交付产品中所占比重较小。

另外，在第 4 章 C/S MMO 游戏中，大概分为以下几个阶段来介绍提交的资料。

- **框架开发阶段**
- **原型阶段**
- **商用版本阶段**

通常，P2P MO 游戏会先开发单人游戏。不用像 MMO 游戏开发那样详细设定游戏框架开发阶段，因为这种开发方式在游戏业界已经很少见了。当然如果游戏的题材新颖、有创新性的话，对这些创新部分的详细技术验证还是有必要的。

和 MMO 不同，MO 游戏的开发顺序可以先制作单人游戏（原型），商业游戏也会先从单人游戏开始，然后追加网络功能，以及商业化相关的辅助系统。不过根据游戏题材和类型的不同，比如遇到对战格斗游戏等对网络延迟要求较高的情况时，开发完单人游戏后，会进行商业化版本前的网络版本的验证，然后再进行商业化版本的开发。

之后是辅助系统的开发，比如玩家匹配系统、积分榜、玩家之间的交流功能等，这样就更接近最后发布的商业版本了。具体内容会在第 6 章详细说明。

和 C/S MMO 游戏一样 MO 游戏也需要概要设计书。

概要设计的详细资料

在第 4 章中，我们参考了 *Runescape* 这款游戏，这里我们以著名的 P2P MO 游戏《暗黑破坏神》为例，开发一款同类型的游戏。由于篇幅

所限，这里就不再对《暗黑破坏神》进行详细描述。在模仿某款游戏之前，应该花大量时间去试玩，然后再参考相关英语维基百科或者其他玩家的游戏视频，这样初期的开发就没有什么问题了。

从这些公开的资料中我们可以整理出以下设计要点。本书的案例游戏 *J Multiplayer* 的目标是实现《暗黑破坏神》类型游戏的核心玩法部分，不会完全照搬。核心玩法部分主要是指在单元格类型的地图上，可以控制玩家角色攻击不断出现的敌人获得经验值。

以下要点之后会作为 *J Multiplayer* 游戏框架开发的功能列表。

- 一张游戏地图
- 敌人在本地机器上移动
    - 暂定一种类型的敌人
    - 不同步全部的敌人
    - 离玩家角色最近的优先移动＋攻击
    - 同一层有数十到数百只怪物
    - 路径搜索暂时不做
- 玩家角色
    - 暂定一种类型
    - 直线移动，数据同步
    - 只攻击指定的目标
    - 生命值（HP）和经验值（EXP）随等级上升
- 有房屋和门，在房屋和走廊里随机动态生成敌人
- 只能重新启动游戏来重置游戏状态

- 原型阶段不添加玩家匹配和聊天功能
- 同一台电脑启动多个游戏程序时使用不同的端口号
- 游戏启动时，可以指定 IP 地址和端口号

完成上述功能后，最基本的游戏玩法就可以实现了。

- 敌人会联合起来一起攻击，如果是单个敌人，会自杀式攻击
- 有 2 个敌人时，同伴可以分工合作各个击破
- 门被打开后，同伴也可以通过

为了实现以上概要设计，需要在详细设计阶段将这些要点反映在设计中。

5.2.4　和 C/S MMO 的数据量 / 规模的比较

在 *J Multiplayer* 游戏中，一层关卡的地图和敌人的分布情况可以参考图 5.1。

图 5.1　*J Multiplayer* 的地图和敌人分布

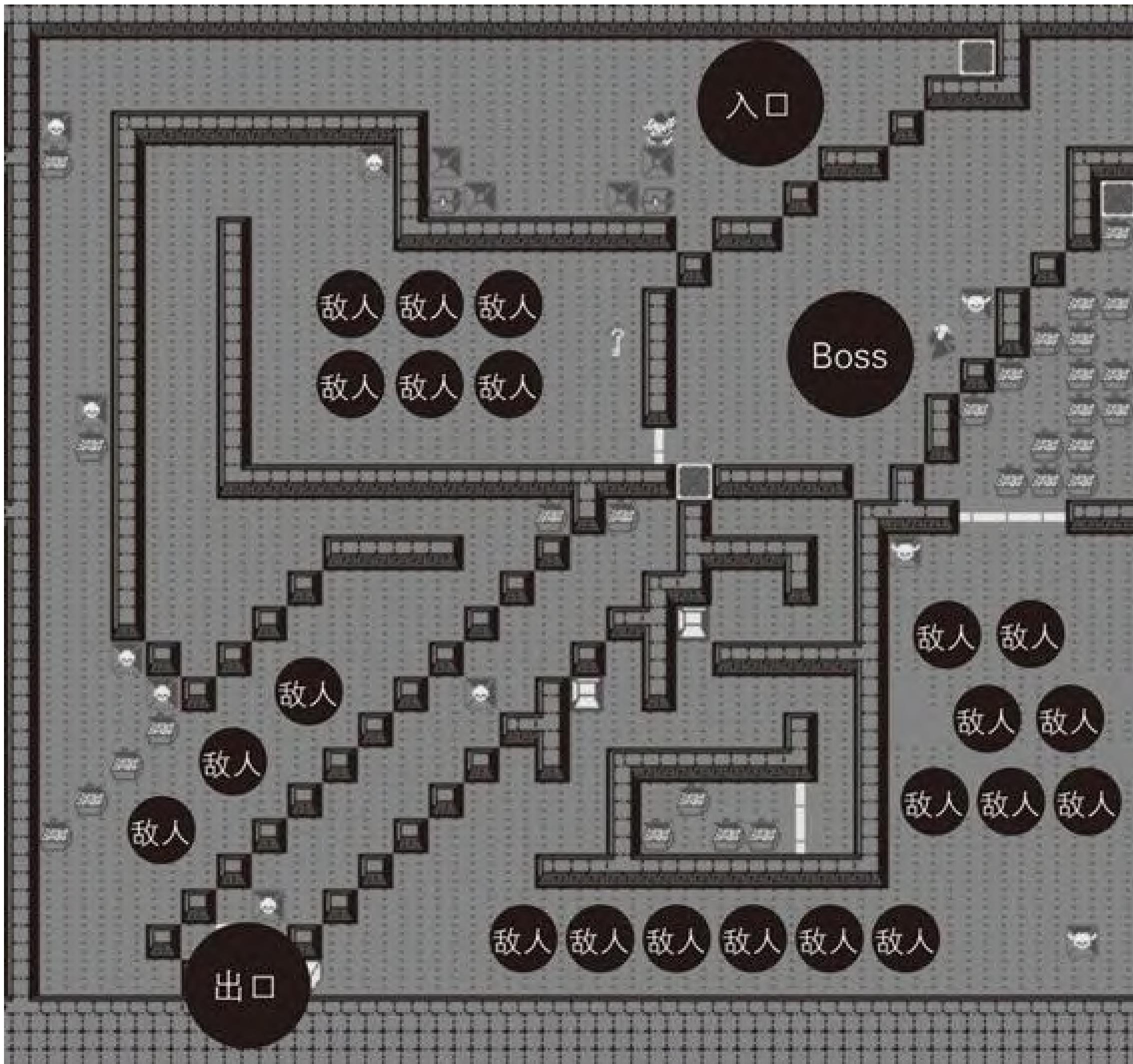


图 5.1 是 *J Multiplayer* 中一层关卡的整体分布图，包括了敌人、入口、出口、Boss 等的分布情况。地图由几百 × 几百的单元格组成（图 5.1 可以看到部分）。这些地图单元格的信息大概几十字节。客户端程序 的内存中只有地图信息。

K Online 有非常大的游戏世界，各个地方都分散着不同的玩家，所以游戏世界的信息有几百兆，没有必要都存在内存中。在 *J Multiplayer* 中，游戏玩家最多也就 6～8 人，每个人必要的关卡数最多只有几个，这些关卡信息乘以玩家数量得到的总内存消耗也就几百千字节 × 8，最多不过几兆字节而已 [6] 。

[6] 因为有好几兆字节，所以可能需要传输所有玩家相关的信息和全部关卡的信息。

当然，如果游戏的设计在不同的关卡频繁移动，每个关卡逗留的时间很短的话，必要的内存使用量有增加的可能性，这种情况需要特殊考虑。

如上所述，*J Multiplayer*（P2P MO）和 *K Online*（C/S MMO）相比，使用的数据量有很大不同，所以程序的设计必然也不一样。当然，数据量 / 规模的大小根据游戏题材和设计的不同也会不一样，所以程序员需要正确理解游戏概要设计的内容。

5.3　P2P MO 游戏的设计资料

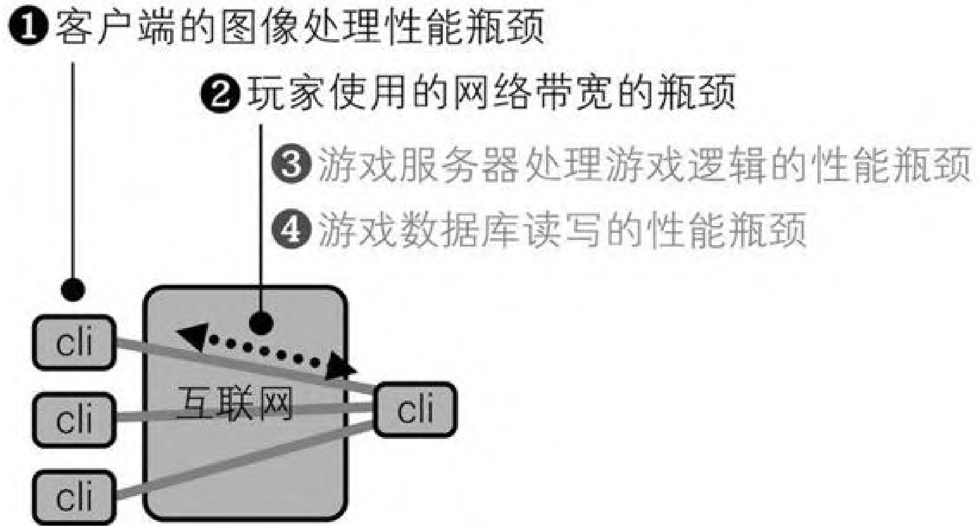
P2P MO 类型的游戏和 MMO 类型游戏不同，不需要使用服务器来处理游戏数据，也没有必要估算运营成本，所以需要准备的设计资料比较少。这里简单说明应该准备的最少限度的资料。

5.3.1　系统基本结构图

首先作为比较，我们来回顾下 C/S MMO 类型游戏的系统瓶颈。请参考第 4 章图 4.1。第 4 章对 C/S MMO 系统容易产生性能瓶颈的 ❶～❹ 点做了详细说明。

但是，P2P MO 类型游戏没有专用的服务器，所有玩家只在共同游戏的客户端之间进行通信，图 5.2 右侧 ❸、❹ 点那样的性能瓶颈理论上是不会发生的，所以只需要考虑客户端的图像处理性能和玩家所使用的网络带宽。

图 5.2　P2P MO 系统和可能产生性能瓶颈的地方



在设计 *K Online* （C/S MMO）时格外担心，所以设计师们会在全部服务的系统基本构成图上，标注随着用户的增加会如何产生什么影响以及哪些部分的压力会增长（参考图 4.9）。*J Multiplayer* 没有服务器端的处理，需要准备服务器的只是和游戏内容没有直接关系的辅助系统。参考 *K Online* 制作的系统基本构成图如图 5.3 所示。

图 5.3　*J Multiplayer* 的系统基本构成图

详细内容会在第6章说明，图 5.3 中玩家匹配系统、中继服务和存储服务等各种服务会在不同的服务器上运行。实际的开发中，这些辅助系统都不需要自己开发，索尼、微软和任天堂等企业都提供了这些服务，可以直接使用。

5.3.2 进程关系图

C/S MMO 类型的游戏，包含的进程有客户端程序、游戏服务器、登录服务器、验证服务器和数据库服务器等，按不同的功能划分有十几种。它们之间的关系需要详细定义。但是对于 P2P MO 游戏来说，只有游戏客户端一种，所以可以不用制作系统关系图的资料。

开始开发后，后期比较难更改的部分如下：

- **使用星状拓扑结构还是网状托普结构（请参考第 3 章图 3.12、图 3.11）**
- **同步、非同步还是网页方式实现**

这两点从一开始就需要考虑清楚。

关于第二点，因为 *J Multiplayer* 是在互联网上通信所以不能使用同步的方式，而网页方式又没有那么多玩家参与，所以采用非同步的方式实现是较为合适的。

相对麻烦的是网络拓扑结构的选择，到底是用星状拓扑还是网状拓扑。

星状拓扑结构还是网状拓扑结构

星状拓扑结构和网状拓扑结构的区别在于，共享全部客户端程序（玩家机器上运行的游戏程序）的数据所必须的通信链路连接数。星状拓扑结构是 2，网状拓扑结构是 1。星状拓扑结构的客户端之间通信需要经过主机，而网状拓扑结构不需要。所以，经过互联网的通信延迟除主机之外，玩家机器之间理论上都有两倍的延迟。但是，网状拓扑结构除了网络的 NAT 问题之外，互相之间不能通信的玩家也比较多，会有无法进行游戏的情况。

因此，就网状拓扑结构来说，只有对反应速度要求较高的对战格斗游戏、街机游戏或者任天堂 DS 等使用 ad-hoc 通信的掌机等，没有 NAT 问题，可以保证通信延迟在几微秒之内的情况下才使用。

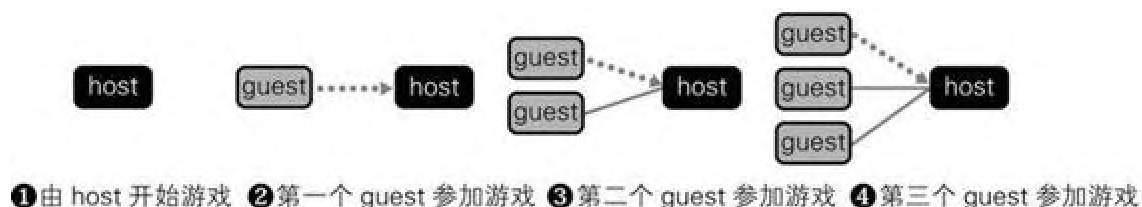
另外，在星状拓扑结构中，根据游戏内容的不同，如果有排他限制的功能时，可以在主机的处理程序中实现相应功能，程序也比较简单。

首先来验证星状拓扑结构

通过以上分析，我们可以知道在开发 P2P MO 类型游戏时，基本的流程是首先验证能否采用星状拓扑结构，在追求较快响应速度的特殊情况时再验证网状拓扑结构。面向互联网用户的游戏基本上都是采用星状拓扑结构。

本章的案例游戏 *J Multiplayer* 也是采用星状拓扑结构实现。也就是说，在玩家中由一人作为主机，剩下的玩家作为客户端。如图 5.4，作为主机的玩家首先初始化并开始游戏。之后其他玩家再加入这个游戏，直到人数达到上限。

图 5.4　游戏开始和玩家（进程）的加入



中途加入游戏的实现

不论是星状拓扑结构还是网状拓扑结构，都可以实现途中加入游戏的功能。不过星状拓扑结构相对容易实现。

首先，我们来看一下星状拓扑结构的做法。在主机的游戏程序中已经保存了排他限制的相关信息以及所有游戏运行数据，当新的玩家加入后，只需要下载所有的数据就可以保持最新的游戏状态。采用网状拓扑结构时，需要先确认游戏内容都没有排他限制后再直接和其他游戏客户端建立连接。比如典型的赛车游戏，流程如下：❶ 首先和所有的游戏客户端建立连接。❷ 在其他游戏客户端初始化并显示新加入玩家

的赛车。❸ 开始传输新加入游戏玩家的操作数据。❹ 更新所有游戏客户端中新加入玩家的赛车位置信息。

实现中途加入游戏功能时，需要传输初始化游戏所需要的数据。在游戏进行的同时传输数据，对客户端程序的处理和带宽有一定要求。如果数据量特别大，则无法实现中途加入游戏的功能。另一种做法是让之后参加游戏的玩家处于等待状态，直到下一个合适的时间点再进入游戏。比如可以在一局游戏尚未结束时禁止新的玩家加入。一般中途参加游戏对程序的负担比较大，增加这个功能时需要慎重考虑。

无论是星状拓扑结构还是网状拓扑结构，根据游戏内容的不同，中途参加游戏功能的开发还有很多难点。就拿赛车游戏来说，信号灯变绿表示游戏开始，实现比较简单，也容易理解。其他的桌游，比如麻将，从游戏内容上来说并不适合中途参加游戏。即时战略类（RTS）游戏也是如此，游戏初期状态和开局非常重要，中途参加游戏很难实现。但是第一人称射击类（FPS）游戏就比较容易实现途中参加游戏的功能。所以根据游戏的题材和设计，途中参加游戏功能的开发难度和工期也有很大不同。

采用星状拓扑结构时，如果游戏进行时主机连接中断，所有的玩家就会失去连接，变成单人游戏状态（图 5.5）。比较典型的情况是由AI（电脑）代替其他玩家的操作。有些游戏在主机掉线时，其他玩家可以变成主机，并允许新的玩家加入。

图 5.5　正常状态和主机终端时的状态

与此对应，使用网状拓扑结构时，各游戏程序之间没有依存关系，任何人失去连接也不会有影响。

另外，在实际的商业化游戏中，我们还需要连接玩家匹配服务器、认证服务器等辅助系统，相关介绍请参考后面的章节。

5.3.3　带宽 / 服务器资源计算资料

不需要考虑服务器数量的预算。在服务器端也没有使用专用的网络，除了后面章节会介绍的辅助系统之外也不用考虑带宽的情况。

5.3.4　通信协议定义资料和 API 规格

C/S MMO 类型的游戏需要定义不同进程之间的通信协议，对于 P2P MO 游戏来说，只有游戏客户端一种，所以通信协议也比较简单。

就拿本章的案例游戏 *J Multiplayer* 来说，除了辅助系统以外游戏只需要处理 ping、getid、guestid、sync 和 delete 这 5 种函数，这其中比较重要的 sync 和 delete 在客户端和服务器端都需要使用。

通信协议的序列图

在 C/S MMO 游戏中，从客户端→游戏服务器→数据库服务器，数据会经过多层服务器，需要保持其一致性。但是在 P2P MO 游戏中，如果是星状拓扑结构，基本上只有主机和客户端之间的通信，不需要画序列图进行详细确认。

函数和常量的定义

J Multiplayer 的通信协议中定义了一下 5 种操作类型。

- ping、pong
- getid
- guestinfo
- sync
- delete

- **ping 函数和 pong 函数**

ping 和 pong 函数用来验证主机和客户端直接能否通信，以及网络延迟情况。

```
void ping(U64 guestclock);    ←客户端调用的函数，参数为 64 位无符号整数
void pong(U64 hostclock, U64 guestclock);    ←主机调用的函数
```

如上述例子所示，客户端和主机之间交换时间。客户端和主机都分别管理着自己的时间。在调用 ping 函数时，将程序启动后经过的时间（单位是毫秒）传递给主机，然后主机通过 pong 函数返回主机的时间。如果网络延迟大于一定数值会显示警告信息，并断开与主机的网络连接进入单人游戏模式。

P2P MO 游戏没有服务器也可以进行游戏，所以大部分游戏会在遇到网络问题时转入单人游戏模式，这也是在游戏逻辑上与 C/S MMO 游戏不同的地方。

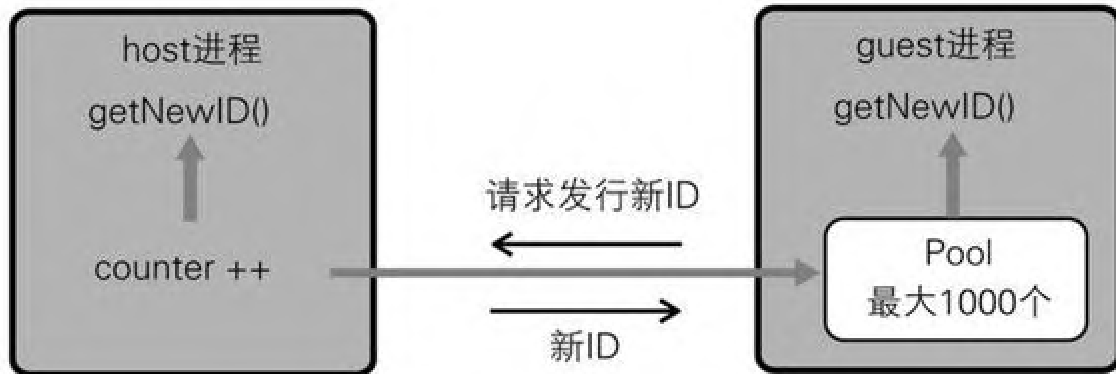
- **getid——ID 管理池**

getid 函数用来分配在游戏房间内的唯一 ID，使用了 ID 管理池的方式实现，占用位数少。

```
void getid(U16 num);
void getid_result(U64 ids[1000]);  ← getid_result 函数返回新的ID
```

ID 管理池用来管理游戏房间内可动物体的唯一 ID，是一种常用的方法。ID 管理池的示意图如图 5.6 所示。

图 5.6 ID 管理池的示意图



获取特定进程内唯一 ID 的方法很简单，只需要计数器逐渐累加即可。但是如图 5.5 那样有主机和客户端两个进程同时运行时，因为两者的计数器之间并无联系，所以可能会分配到相同的 ID。为了避免这种问题，有两种解决方式。

专栏　什么是“游戏逻辑”

游戏逻辑（Game Logic）一词在本书中经常出现，目前还没有准确的定义 [7] 。

[7] 本书撰写时（2010 年 10 月），在维基百科（Wikipedia）的英语版和日语版中都没有该词的解释。翻译时也没有（2013 年 3 月）。——译者注

本书中该词的意思和 Web 程序中经常使用的业务逻辑（Business Logic）类似。在 Web 程序开发中，业务逻辑是指“控制数据库和用户接口之间的数据操作的逻辑”，在游戏开发中，包含以下两点。

- **游戏运行时的数据：以象棋为例，指棋子的分布情况，和数据库相关联。**
- **用户界面信息：还是以象棋为例，在游戏界面中什么地方应该绘制什么颜色，鼠标可以在什么地方点击等信息。**

游戏逻辑就是这些数据背后相应的控制逻辑。

比如下面这一系列的游戏规则。

- **游戏开始时棋子的位置应该怎么摆放。**

- **不同棋子可以移动的位置。**
- **“步”（也称步兵）只能纵向走一格。**
- **不能将己方棋子放在对方棋子之上。**

只有遵守一定的游戏规则才能保证游戏的乐趣。

和 Web 程序开发一样，游戏逻辑部分和其他模块之间的界限并不十分明确。所以开发的时候，需要尽可能的让逻辑部分和数据库以及用户界面分离开来，这样才能称之为游戏逻辑部分。

- **方案 1：ID 由 [进程 ID, 程序内部 ID] 两部分组成。例如，host 的进程 ID 是 0，客户端是 1 的话，[0,1] 和 [1,1] 就构成了不同的 ID。这种方法可以叫做进程 ID 法。**
- **方案 2：客户端的 ID 都通过主机分配，由主机的 ID 管理池分配，并自动累加。这种方式叫做 ID 管理池。**

ID 管理池是第二种方式 [8] 。可以在最开始采用方法 2 获取了起始 ID 后，再用方法 1 获取之后的 ID。

> [8] 理论上兼容第一种方式（大的兼容小的，ID 管理池是大的）。

第一种进程 ID 法在获取最初的进程 ID 时也需要访问主机。例如，使用两个变量 [U32 型的进程 ID，U32 型的内部 ID] 的情况和使用 ID 管理池时，[在最开始的时候一次性分配 32 位，也就是 42 亿个 ID] 是差不多的。所以 ID 管理池和进程 ID 法在实际的程序开发中的工作量几乎没有区别。

笔者比较偏向于使用 ID 管理池的方式，因为 ID 可以使用一个原生数据类型（U64 或者 U32）保存，代码简单高效，节省内存性能优异，而且 ID 的位数比较小。ID 的读取、搜索会使程序的循环处理比较多，所以应该尽可能让 ID 的构成简单化，只有一个变量会比较方便使用。

不管是进程 ID 法还是 ID 管理池的方法，如果客户端长时间不能访问主机，会造成不能生成新的 ID 的情况。所以需要根据游戏的内容

来决定采用什么管理策略。假设游戏中每秒新产生 3 个敌人，游戏时间在一个小时左右，如果能够保证生成 10 万个 ID，理论上就基本没有问题了。

前面的例程所示生成新 ID 的 getid_result 函数可以返回 ID 列表，例如，一次申请 6 个 ID 时，就会返回连续的 ID[100, 101, 102, 103, 104, 105]。不过这样会造成带宽的浪费，可以使用 start=100、end=105 的方式，让返回结果更经济高效。

- guestinfo

  guestinfo 函数是主机用来分配所在游戏空间内与主机相连接的客户端的唯一 ID。

```
void guestid(U32 id);
```

  这个函数是主机告诉所有客户端他们各自的唯一 ID。

  对于一个 ID 来说，理论上主机知道它是客户端申请的还是主机自己使用的，不过逐个搜索会比较慢，所以在物体移动时，会在所有的同步数据包内包含控制该物体移动的 guestid，这样就可以省略搜索了。主机可以使用 guestid 函数来告诉客户端该唯一 ID。在 *J Multiplayer* 游戏中，主机的 guestid 默认为 0，客户端是 1 以上的值。

- sync

  sync 函数用来同步可动物体在移动时的数据。可动物体出现时，根据之前不存在的 ID 所对应可动物体的移动来判断。

```
void sync(U32 guestid, U32 id, U8 data[200]); ←一个对象的变化在
200 字节以内表达
```

guestid 是客户端的 ID，id 是可动物体的 ID，在整个游戏房间内是唯一的。data 是 2 进制的数组，格式如下所示，大小限制在 200 字节以内。

```
data: X 列
X: [1 字节，表示变量类型 ][1 ～ 4 字节，数据 ]
```

J Multiplayer 游戏包含 1 字节的 U8、4 字节的 I32 和 4 字节的浮点数（Float）。很多游戏没有使用 8 字节的双精度浮点数（double）类型，这是因为和精度相比更重视节省带宽。大部分游戏也不需要使用这种双精度浮点数的数据。

另外，在 *J Multiplayer* 中，“表示变量类型的 ID”（类型 ID）使用了以下枚举变量定义。

```
typedef enum {
  SVT_TYPEID,
  SVT_COORD_X,
  SVT_COORD_Y,
  SVT_DELTAPERSEC_X,
  SVT_DELTAPERSEC_Y,
  SVT_GOAL_X, SVT_GOAL_Y,
  SVT_TOSTOP,
  SVT_TODELETE,
  SVT_TARGETID,
  VT_HP,
  SVT_MAXHP,
  SVT_MP,
  SVT_MAXMP,
  SVT_EXP,
  SVT_LEVEL,
  SVT_SHOOTER_ID,
  SVT_HITTYPE,
} SyncValueType;
```

例如，在主机上新生成了可动物体，分配的 ID 是 45，调用 sync 函数同步客户端数据，参数为 guestid=0、id=45。客户端首先在内存中搜索，如果 id=45 的物体不存在，就新建该物体；

如果存在该物体，就使用服务器端发送来的数据替换本地内存的数据。

- delete

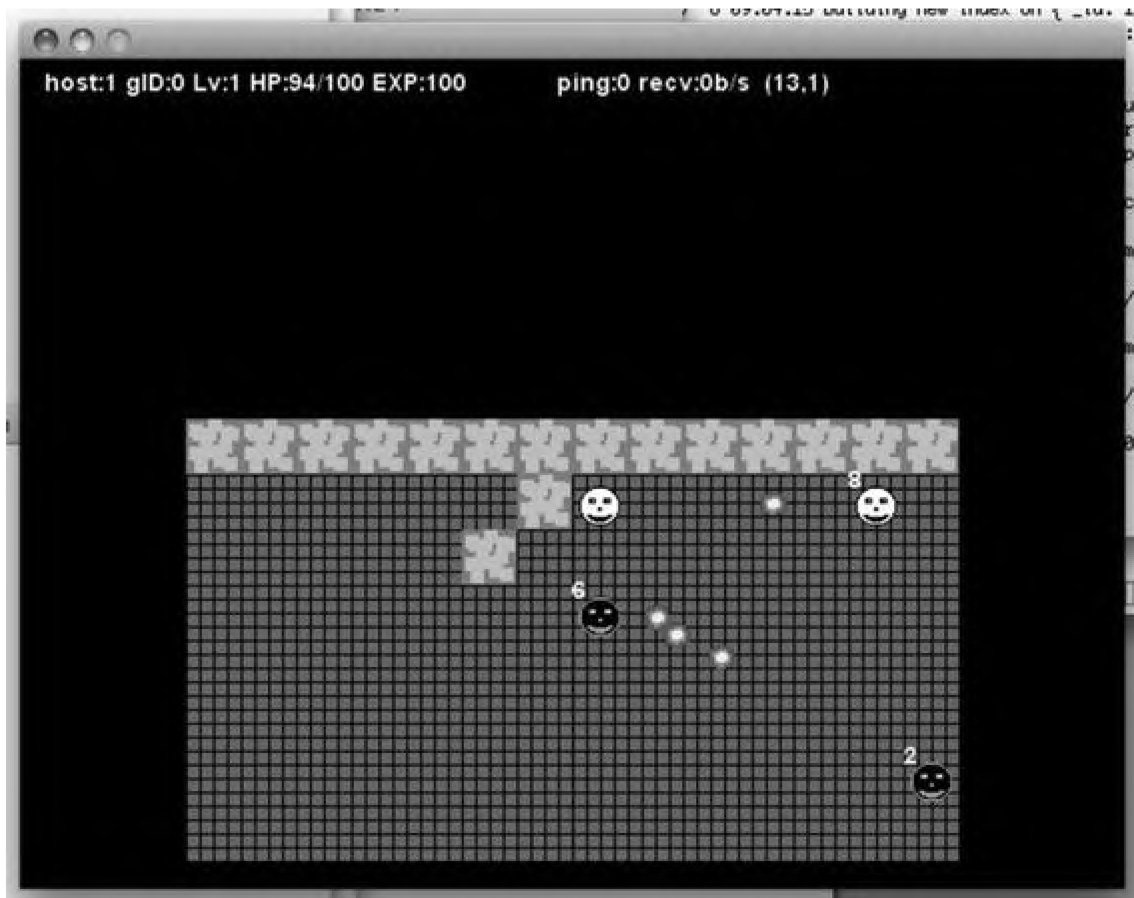
delete 函数用来删除可动物体的信息。例如主机要删除45号可动物体则调用 delete(0,45)，向各个客户端发送该消息。

```
void delete(U32 guestid, U32 id);
```

5.3.5　带宽消耗量的估算

下面我们来看一下带宽消耗量的估算。*J Multiplayer* 运行时的画面如图 5.7 所示。和 C/S MMO 的感觉差不多，图像的质量这里暂且忽略。

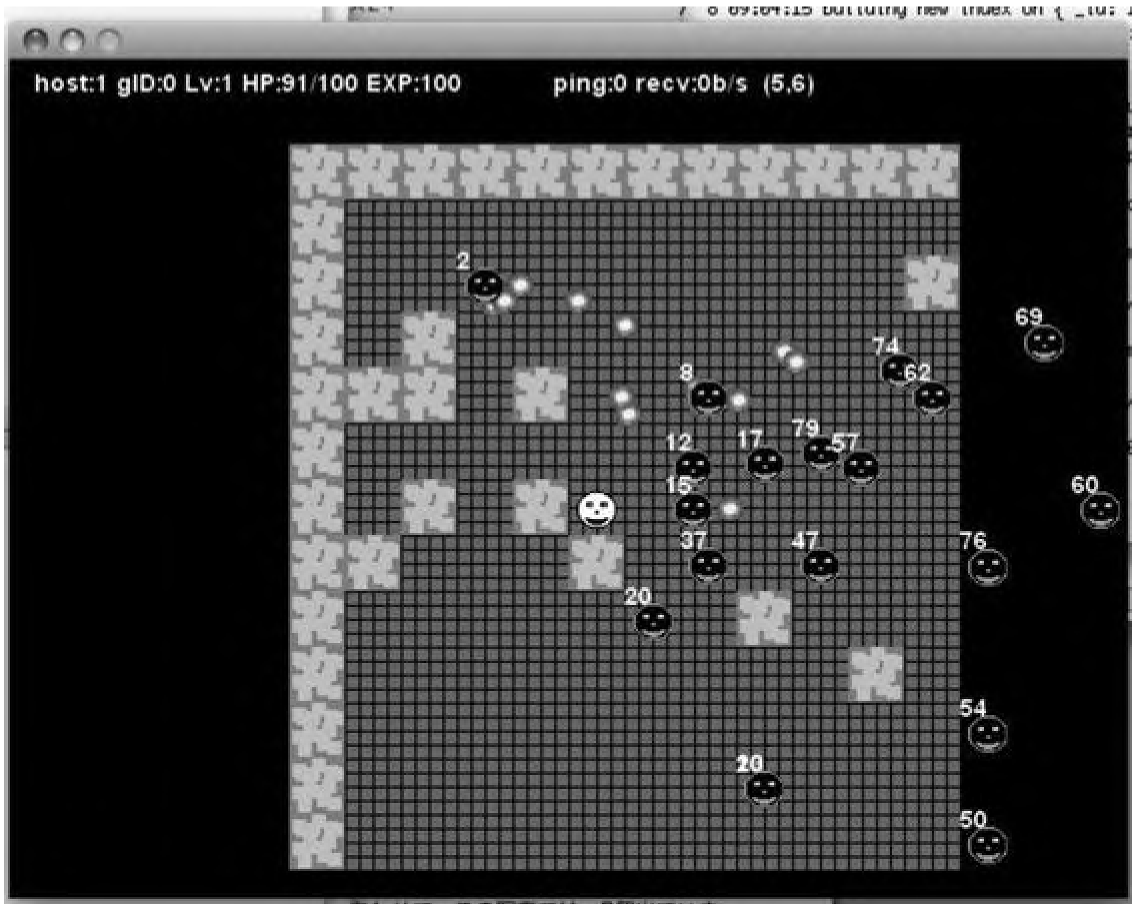
图 5.7　*J Multiplayer* 运行时的画面



在图 5.7 中，白色的为玩家控制的角色，黑色的是敌方角色，小圆点表示子弹。敌人左上方的数字表示调试时可动物体的 ID。从图 5.7 中可以看到可动物体共计 8 个（玩家角色、敌人角色、子弹）。

在游戏设定中，实际运行时可动物体的峰值大概是 40～50 个（如图 5.8），如果包括屏幕外的可动物体，最多有 1000 个左右。这个游戏是一个会与大量的敌人作战的游戏。在实际的即时战略类游戏（RTS）中，可动物体一般没有 1000 个。

图 5.8　峰值时的 *J Multiplayer* 的画面



该游戏客户端运行时每秒帧数为 60，也就是说，这 1000 个可动物体每秒可以动 60 次。

根据前面的协议说明大部分是调用 sync 函数更新坐标，即计算坐标后调用 sync 函数，如下所示。

```
[UPD/IP 标头 20 字节 ] [ 函数标头 4 字节 ] [U32 guestid] [U32 id] [U8 变量类型 ID]
[Float4 X 坐标 ] [U8 变量类型 ID] [Float4 Y 坐标 ][9]
```

[9] Float4 表示 32 位浮点小数。

函数标头包含函数 ID 和数据长度等信息，总共 4 字节。

60×1000×（20 + 4 + 4 + 4 + 1 + 4 + 1 + 4）× 8 位 = 20.0Mbit/s[10] 。4 人同时游戏时，主机需要和 3 个客户端通信，通

信负荷为 20.0Mbit/s × 3 = 60 Mbit/s（！）。即便是光纤用 户也会比较紧张。[11]

> [10] 60 表示帧数，1000 是运动物体数。

> [11] 开发中常见的失误是在公司内部局域网测试，一直到网络公测时才发现在实际的网络中会有带宽的问题，所以要注意避免这样的问题。

一般的游戏机要求通信量在 50～150kbit/s 左右。大部分用户使用的是普通宽带，实际的带宽达不到几 Mbit/s 这种程度。

“1/600”的实现方法 ——仔细检查游戏设计寻找解决方案

假如想让主机通信量在 100kbit/s，需要压缩“1/600”左右。这个目标可以实现吗？遇到问题时，我们可以采用相同的办法。即“仔细检查游戏设计寻找解决方案”。这里先不考虑 *J Multiplayer* 的游戏内容是否有趣，从开发角度可以采用下面的分析步骤。

- **方案 1：是否可以不同步和自己无关的敌人信息？**

  →这个是 P2P MO 游戏必须要考虑的问题。在 *J Multiplayer* 游戏中，敌人按照下面的顺序移动。

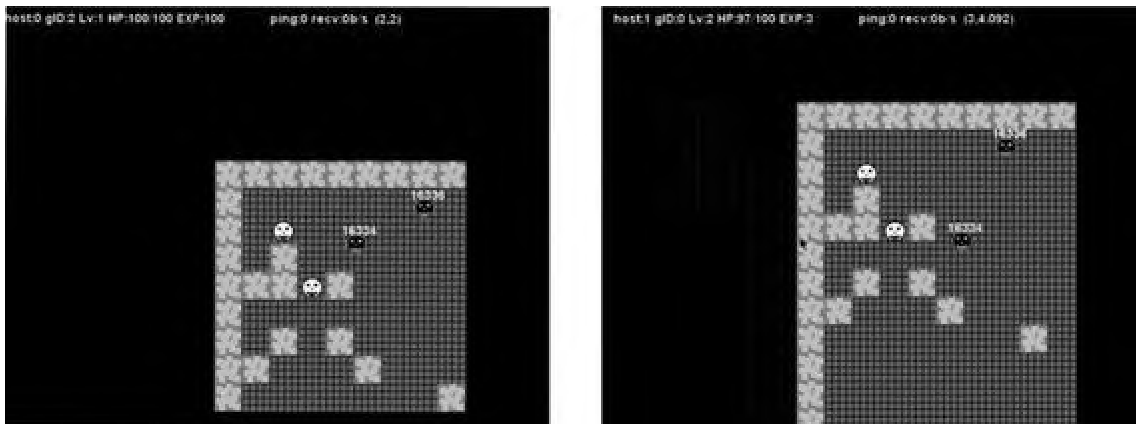
  - 受到玩家角色攻击时，锁定该角色并追击，在一定时间后恢复原状态。
  - 就算没有受到攻击，只要玩家靠近就追击玩家。

  J Multiplayer 的基本游戏玩法是同追击自己的敌人战斗。所以没有追击玩家的敌人就可以不用同步相关信息。这不是“极端情况的处理”，而是在实际开发中应该选择的，也是技术上经常采用的方式。

  分析的结果就是需要同步一部分敌人的信息，但是没有追击玩家的敌人可以不用同步。此外，根据第 3 章的结论，追击玩家的敌人在玩家机器的进程内移动会有更好的游戏体验。

在实际开发时，两个客户端在同一时间的游戏截图请参考图 5.9，玩家的位置相同，但是敌人的位置不一样。两个画面放在一起比较我们会觉得游戏体验不一样，但实际上是两个不同玩家通过网络进行游戏，不存在体验不同的问题。*J Multiplayer* 游戏的基准是“敌人数量一致”，只需要确保敌人出现和死亡的时间一致。这样就可以只在出现和删除时进行通信，从而减少大量因为移动而产生的数据通信。

图 5.9　玩家位置一样但是敌人的位置不同



根据这个结论，假设 1000 个敌人的构成如下所示。

- 200 个追击玩家 A（主机）
- 200 个追击玩家 B（客户端）
- 200 个追击玩家 C（客户端）
- 200 个追击玩家 D（客户端）
- 200 个空闲敌人，随机分布

上述情况，对于主机玩家 A 来说，应该向客户端玩家 B 发送什么信息呢？首先追击玩家 A 的敌人信息是不需要向 B～D 玩家发送的。追击 B 的敌人是在 B 客户端上运算，所以也不需要发送。这样就只需要同步剩下那 200 个空闲敌人的信息。

这样一来就成功的减少了“1/5”的通信量。在带宽上可以削减12Mbit/s，而且游戏体验也更好了。另一方面，正如第 3 章所讲，游戏的策划内容中说明了除 Boss 角色之外的敌人在出现后会很快死掉，所以也不需要同步每个角色。如果是那种角色之间相互影响，角色寿命也比较长的游戏就不能使用这个方法了。

现在距离 100kbit/s 的目标还需要减少 1/120……

- **方案 2：距离远的敌人信息可以不同步吗？**

  J Multiplayer 游戏的地图比较大，没有显示在屏幕上的地方有很多，所以可以每次只同步离玩家近的敌人信息，距离远的偶尔同步一次。通过观察图 5.10 的游戏画面，对于屏幕外一定距离的可动物体，每一帧都同步，其他位置的物体每 100 帧同步一次，这样又可以减少大约一半的通信量。现在是 6Mbit/s，距离 100kbit/s 还有 1/60……

图 5.10　画面显示区域 / 非显示区域和敌人的移动

- **方案 3：1/60 的话……那可以每 60 帧同步一次吗？**

  这样就可以一下减少到 1/60 了……不过需要仔细考虑才行。同步频率降低可能会大幅度影响游戏体验。

  首先，*J Multiplayer* 中的角色移动很快，每秒移动 4 格。60 帧同步 1 次的话就是 1 秒通信一次。因为移动 1 格是 0.25 秒，1 秒最少可以移动 3 次。如果“向右移动一步再向左移动 1 步”，可能会显示成“没有移动”。这是游戏无法接受的情况，应该“每秒至少同步 5 次”。1 秒 5 次就是 12 帧同步一次，这样可以一下减少到 6Mbit/s/12 = 500kbit/s。

  但是，如果 12 帧才更新一次位置信息，会造成“移动时的画面跳动”，看上去会不舒服，为了有更流畅的移动效果，需要加入“角色移动时的坐标补充”处理[12]。

  再“削减 1/5”就到 100kbit/s 了。

- **方案 4：可以只同步变化的数据吗？**

  J Multiplayer 中的敌人角色不是一直在移动。处于待机状态的情况也很多，特别是敌人刚出现时大部分是待机的。而且斜着移动的情况也比较少，多数是上下左右十字方向的移动。如果“只在 X 坐标改变时”或者“只在 Y 坐标改变时”同步，也可以减少通信量。在屏幕显示的部分，游戏运行画面中移动的物体（需要同步的）只有原来的一半。这个方法效果也很好，“500kbit/s 的一半”是 250kbit/s，还差最后一点。

- **方案 5 子弹的处理可以优化吗？**

  J Multiplayer 中的子弹是直线飞行。所以知道了发射的时间、方向和速度，只需要一次同步，之后的运行是可以计算的。调用 sync 函数同步子弹信息时，只要在数据包内添加发射时间就可以了。时间差可以通过 ping 函数调整。这个方法有一个问题，就是在子弹发射后进入游戏的玩家会看不到之前的子弹。不过这个问题不大，可以忽略不计。根据游戏画面观察和计算，游戏运行时子弹的数量大概占可动物体的 20% 左右，所以效果应该很明显。

  现在的通信量是 250Mbit/s ×0.8 = 200kbit/s，还差一半。

- **方案 6：数据包可以优化后再发送吗？**

  现在 sync 函数的格式如下。

  ```
  [UPD/IP 标头 20 字节 ] [ 函数标头 4 字节 ] [U32 guestid] [U32 id]
  [U8 变量类型 ID]
  [Float4 X 坐标 ] [U8 变量类型 ID] [Float4 Y 坐标 ]
  ```

  上述格式中 [UPD/IP 标头 20 字节][函数标头 4 字节] [U32 guestid] 是冗余部分，不包含坐标值，每次都重复发送同样的数据。UDP 数据包最大可以传输 1500 字节的数据，所以

```
[UPD/IP 标头  20 字节 ] [ 函数标头 4 字节 ] [U32 guestid] [ 剩余的数据 ]
```

数据包的头 28 个字节只需发送一次，剩下的 1472 字节用来传输移动坐标，我们假设一下最坏的情况，所有的可动物体 X 和 Y 都发生变化。

```
[U32 id] [U8 变量类型 ID] [Float4 X 坐标 ] [U8 变量类型 ID]
[Float4 Y 坐标 ]
```

（4 + 1 + 4 +1 + 4）= 14 字节，应该可以削减不少。

全部按低压缩率、X 坐标和 Y 坐标都变化的情况下，1 个可动物体的数据是（20 + 4 + 4 + 4 + 1 + 4 + 1 + 4）= 42 字节，用新的方法只需要（4 + 1 + 4 +1 + 4）= 14 字节，差不多可以压缩“1/3”。标头的大小在 1500 字节中比重很小，可以不用计算。

我们将按这个协议通信的函数名定义为 sync_multi，通过调用

```
void sync_multi(U32 guestid, U32 id_array[100], data_array[]);
```

可以实现数组方式的通信。

[12] Web 上也叫做 Tween 移动。

* * *

这样 200kbit/s / 3 = 70kbit/s 左右，还能有一些缓冲空间，真是可喜可贺……

- **方案 7：敌人少时多发送一些数据**

  游戏开发者的特点就是不会仅仅满足于达到通信量的需求。已经确保在敌人最多时也不会发生数据传输的问题，那么在敌人少时就能多同步一些信息，给玩家提供更准确的游戏状态。可以按照以下方式划分：敌人数量在 50 个以上时，每 12 帧同步一次；20 个以上时，每 6 帧同步一次；10 个以上时每 3 帧同步一次；10 个以下时每 2 帧同步一次。

* * *

采用了上述 6 种方案终于让 *J Multiplayer* 游戏达到了 100kbit/s 以内通信量的要求，不过还有很多可以探讨的地方。最极端的一种考虑就是“所有装饰性的可动物体都不同步”，*J Multiplayer* 游戏中的通信基本都是用来同步“四处分布的敌人”的信息，其他的物体可以忽略不计。但是如果在游戏中连四处分布的敌人信息也不同步，就没有多人游戏的意义了。TPS（Third Persion Shooting）——第三人称射击类的动作类角色扮演（ARPG）游戏的基本原理也是类似的。

尽管我们反复叙述了原理，但是如果不仔细理解游戏内容，在实际开发中还是会碰到很多问题。

5.3.6　其他资料

在此前章节，我们已经详细描述过通信协议定义资料中关于数据包格式、数据大小以及必要的标头等的相关信息，此处不再赘述。

另外，DB 设计图只在排行榜和玩家匹配系统中使用，属于辅助系统的范畴，在后面的章节中，我们会详细说明。

5.4　客户端 / 服务器软件 + 中间件、基本原则

此前章节已经说明了如何准备 P2P MO 游戏开发中最基本的设计资料。本节我们开始学习如何开发实际交付的程序。还是像第 4 章 C/S

MMO 的开发案例一样，我们先来确认一下在开发客户端 / 服务器软件和中间件时，程序中应该注意的基本原则。

5.4.1 P2P MO 开发的最终交付产品

P2P MO 游戏和 C/S MMO 游戏的最终交付产品有很大不同。首先就是服务器端程序和客户端程序并没有分开。

笔者开发 *J Multiplayer* 的原型时使用的文件结构如下所示。客户端程序和服务器端程序最终被编译成了一个可执行文件，所以只有一个文件夹。通信协议的定义文件（j.xml）也只有一个，和可执行文件放在同一位置，没有测试用的机器人程序。

- **编译用**
  - Makefile：笔者使用的是 UNIX 的操作系统，通过 Emacs 调用 GNU Make 编译程序。也可以使用 IDE（集成开发环境）。
- **源代码**
  - app.cpp、app.h：源程序文件，包含每帧的实际处理程序。
  - floor.cpp、floor.h：管理内存中地形数据的类。定义了地图单元格的种类和大小，实现了碰撞检测等。
  - font.cpp、font.h：在游戏画面上控制字符显示。
  - game.h：明确了游戏中使用的地形、敌人种类、坐标值等游戏设定信息。
  - id.h：ID 管理池的实现。
  - movable.cpp、movable.h：可动物体以及其子类等的实现。
  - net.cpp、net.h：从网络接收数据后相关处理的实现，还有之后会提到的 SyncValue 类的实现。

  - sdlmain.cpp：在使用 SDL 开发程序时 main 函数的实现。
  - sprite.cpp、sprite.h：使用 SDL 开发的绘制小精灵的程序。
  - util.cpp、util.h：通用工具类代码。
- **通信协议定义相关**
  - j.xml：*J Multiplayer* 游戏中 5 种必要 RPC 的定义。
  - jproto.cpp、jproto.h：通过 j.xml 自动生成的 RPC 存根文件（stub file）。
- **数据**
  - fonts：绘制文字所需要的一些必要的英文字母、数字和符号的图像文件
  - images：存放玩家角色、敌人、子弹和地形相关图片资源的文件夹。

以上就是案例代码的文件结构。扩展名为 .cpp/.h 的代码文件一共 3600 行，其中包含 600 行自动生成的代码。商业游戏中的游戏处理会更多，3D 游戏的话代码量大概在 10 万～20 万行，2D 游戏在 5 万～10 万行左右。除此之外，图片等多媒体资源也可能会有几吉字节，这些不在本书的讨论范围内。

5.4.2 P2P MO 中使用的中间件

4.12 节介绍了 C/S MMO 游戏使用的中间件：全功能型、小规模型和通信中间件。但是 P2P MO 类型的游戏还可以使用 Quazal 提供的 Net-Z，一款可以在 PC 或者游戏主机等各种平台使用的产品。此外，免费软件 GNE （Game Networking Engine）、Unity 等价格低廉的工具也具备必要的网络功能 [13] 。

[13] 其他相关信息请参考如下网页：
http://www.thefreecountry.com/sourcecode/games.shtml

另外，游戏主机平台也可以使用 SCE 或者 Microsoft 提供的强大的程序库，出于保密需要，这里不再详述。

5.4.3 编程时应该注意的基本原则——针对 P2P MO 游戏

在 4.13 节针对 C/S MMO 游戏列举了以下 4 个基本原则。

- **数据结构优先原则**
- **保持游戏状态原则**
- **后台处理延迟原则**
- **连续测量原则**

P2P MO 游戏因为没有后台服务器，所以除了第 3 条以外，其他原则都同样适用。

5.5 P2P MO 游戏 *J Multiplayer* 的实现——正式开始编程

接下来，我们就开始编写 P2P MO 游戏的程序代码。我们先来一起看一下如何分阶段编写程序。

5.5.1 *J Multiplayer* 的编程计划

和 MMO 类型游戏相比，P2P MO 类型的游戏只有一个程序，所以可以 1 个人开始原型的开发。

这里采用游戏开发中常用的方法，即将游戏逻辑、图像处理和数据处理等模块分开处理。如果有两个以上的程序员，不管在哪一开发阶段，负责开发游戏逻辑的程序员首先要理解网络部分的实现方式，然后再实现各自负责的模块。

特别是在开发初期阶段，如果开发的程序没有考虑网络功能的影响，之后再修改可能会比较困难。所以负责游戏逻辑部分的程序员在最开始一定要充分理解网络通信的设定和相关游戏设计，以防出现网络方面的问题。

另外 C/S MMO 游戏在开发客户端程序之前，需要先开发一个用来自动测试服务器的测试程序 autocli，而 P2P MO 游戏则是首先开发单人游戏部分，不用开发 gmsv，所以也不需要类似的测试程序。

如果是使用简单的矩形或圆形的 P2P MO 游戏原型，一般 1 个人用时一个月，熟悉之后，一周就能开发出可以玩的程序。

此外在描述 C/S MMO 开发的 4.14 节中提到的“不实际运行起来是不行的！——游戏开发的特殊性”这点，在 P2P MO 中也同样适用。

5.5.2　开发流程——*K Online* 的回顾

本章我们来开发 *J Multiplayer* 的原型。在此之前我们来回顾一下在开发 C/S MMO 的案例游戏 *K Online* 时过的流程。

❶ 定义通信协议（k.xml）。

❷ 初始化并通过 bot 测试 autocli 和 gmsv。

❸ 通过 bot 使游戏能够大致完整运行。

❹ 从 cli 着手，先实现图像绘制功能，再实现网络功能。

❺ 一边运行游戏一边添加和调整游戏功能。

5.5.3　*J Multiplayer* 开发阶段——开发顺序和内容

接下来，我们以开发备忘为基础，详细解释一下案例游戏 *J Multiplayer* 的原型开发工作。通过以下描述，我们可以了解和 C/S MMO 相比二者开发内容的一些不同。

首先确定通信方式及原型的开发目标，然后开始开发单人版游戏。接着验证小精灵和字体的绘制以及键盘的输入等。前文介绍的 C/S MMO 游戏的 cli 开发过程，在这里也可以沿用：

❶ 开发可以单人进行游戏的版本。

❷ 在单机版基础上实现具有 cli 连接和共享内存功能的多人版本。

❸ 最后添加玩家匹配、排行榜和交流功能，使其更接近商业化版本（参考第 6 章）。

5.5.4 第 1 阶段的要点

这里整理一下上述第 1 开发阶段的要点。作为参考，下面是以玩家视角记录的一些游戏内容。到了这一步必要的程序算法已经基本确定，之后就可以开始编写代码了。

- **开发可以单人进行游戏的版本**
    - 绘图 UI（从 cli 程序复制）
    - 单屏幕地图
        - TILE =32×32
        - CELL = 128 × 128
        - 玩家角色在屏幕中间
        - 点击鼠标的位置

→如果是地面则移动到该位置

→如果是敌人则发射子弹攻击

→使用弓连续射击时会有冷却时间（冷却时间有上限）

-
    - 大的房间

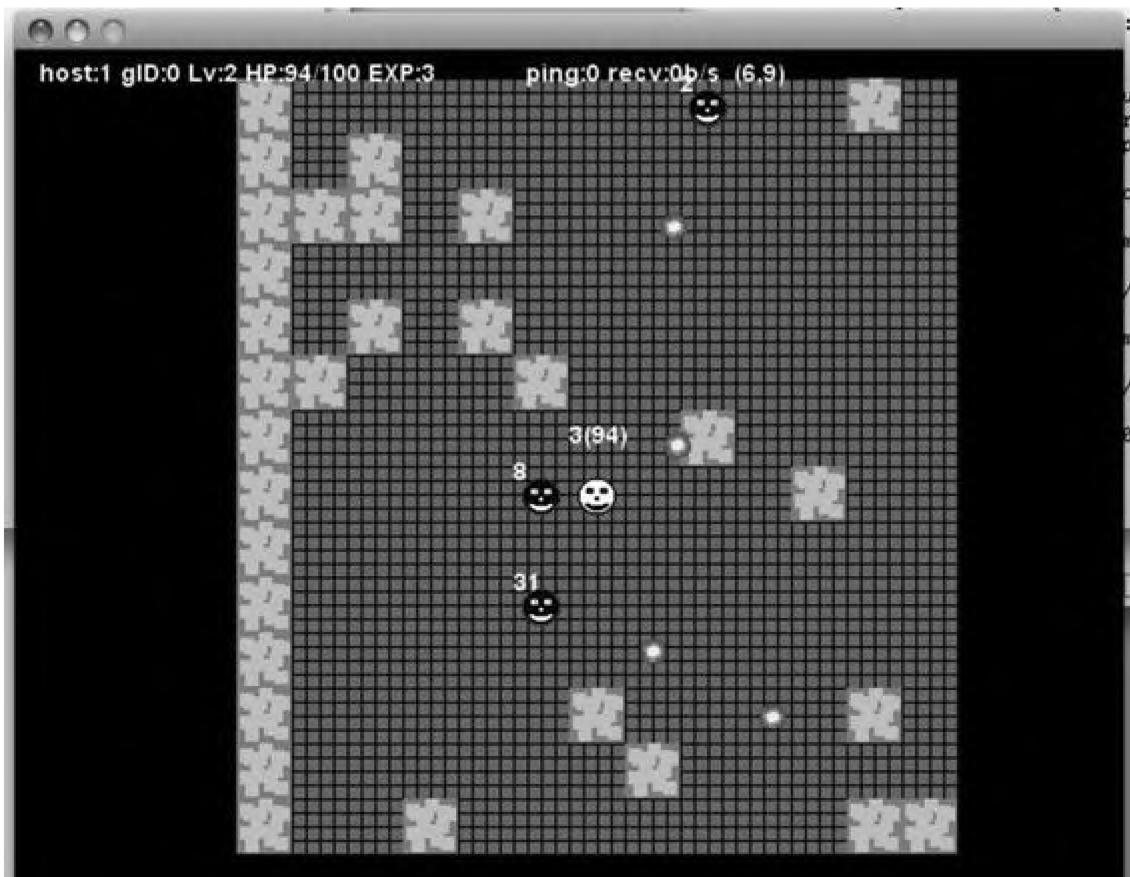
    - 敌人 =100 个哥布林左右，1 种类型，用子弹攻击
    - 随机移动 + 玩家角色进入一定范围后开始追击 + 攻击
    - 敌人在本地机器移动
    - 不用同步所有敌人
    - 死亡时同步
    - 新生成时同步
    - 用子弹攻击时同步
    - 每 30 秒同步一次位置（大致 30 秒左右就行，不用太精准）
    - 敌人之间检测碰撞（子弹之间不用检测）
    - ID 唯一（毫秒单位的启动时间 + 计数器）
    - 不考虑路径探索，直线移动
  - 玩家角色（PC）
    - HP
    - EXP
    - LEVEL
  - 网络（Net）
    - 同一台机器可以启动多个客户端，可以通过参数直接指定 IP

5.5.5　客户端程序的开发案例

这个游戏中角色使用的 BMP 图像和前文提过的 cli（*K Online* 的 cli 程序、图 4.33）相同，所以游戏画面也类似。

如图 5.11 所示，画面中已经实现了显示背景和可动物体，1 个玩家角色来回走动，消灭追击的敌人并获得经验值。当然，如果是在同一台机器上启动两个客户端程序，相互之间也不会有任何影响。

图 5.11　开发中的客户端程序的图像



5.5.6　“共享内存方式”的实现——开始编码

单机版开发好以后，就可以开始实现“共享内存方式”的功能了。顾名思义，共享内存就是将相同的内存数据共享到不同进程之间。需要共享的具体数据包括可动物体的坐标、种类以及移动方向等信息。

竞争状态 ——共享内存方式的注意点

使用共享内存时需要注意的是内存容易遇到资源竞争状态（Race Condition）。竞争状态是指处理结果和预期结果不一致。比如“整数变量加 1”，这个最简单的处理也有可能发生竞争状态。

如果用 C 语言来说明，变量 i 初始值为 0，进行加 1 的运算如下所示。

```
i = i + 1;      ←这里i 的结果应该是1
```

这行代码在实际编译后可分解为以下处理。

❶ 从变量 i 的内存区域中将值拷贝到变量 a 的内存区域

❷ 给变量 a 加 1

❸ 将变量 a 的值从内存区域中拷贝到变量 i 的内存区域中

该例子执行了两次拷贝和 1 次数据计算。

只有 1 个进程时，按照上述处理顺序执行一定会得到预期的结果。但是有两个进程的时候，❶～❸ 处理的执行顺序就不一定了。

表 5.2 按时间顺序展示了内存中数值的变化情况。表格从上到下表示时间顺序，进程 A 和进程 B 交替执行上述 3 个分解后的处理。两个进程分别进行了“加 1”的处理，所以最终结果 i 在初始值 0 的基础上增加了 2，没有出现问题（和预期结果相同）。

表 5.2　内存中数值的变化 1

进程 A	进程 B	i	a
从变量 i 的内存区域中将值拷 贝到变量 a 的内存区域		0	0

进程 A	进程 B	i	a
	从变量 i 的内存区域中将值拷贝到变量 a 的内存区域	0	0
给变量 a 加 1		0	1
	将变量 a 的值从内存区域中拷贝到变量 i 的内存区域中	0	2
给变量 a 加 1		2	2
	将变量 a 的值从内存区域中拷贝到变量 i 的内存区域中	2	2

但是参考表 5.3，执行时机稍微不同时会得到什么结果呢？

表 5.3　内存中数值的变化2

进程 A	进程 B	i	a
从变量 i 的内存区域中将值拷贝 到变量 a 的内存区域		0	0
给变量 a 加 1		0	1
	从变量 i 的内存区域中将值拷贝到变量 a 的内存区域	0	0
	给变量 a 加 1	0	1

进程 A	进程 B	i	a
将变量 a 的值从内存区域中拷贝 到变量 i 的内存区域中		1	1
	将变量 a 的值从内存区域中拷贝到变量 i 的内存区域中	1	1

最终结果变成了 1，和之前的执行结果不同。两个进程都执行了两次“i 的值 +1”的处理，最终结果应该是 2，所以这个结果是不对的。

这种情况我们称为竞争状态。

加锁处理

竞争状态发生的根本原因是同一个处理（i 的值 +1）被分解成了多个操作。所以要避免发生竞争状态，一般可以在处理实际运行前后给资源加锁，让分解的操作集中执行。这个也叫做原子处理。代码如下所示，表 5.4 按时间顺序展示了内存中数值的变化。

表 5.4　内存中数值的变化（加锁）

进程 A	进程 B	i	a
加锁处理		0	不定
从变量 i 的内存区域中将值拷贝 到变量 a 的内存区域		0	0
给变量 a 加 1		0	1

进程 A	进程 B	i	a
将变量 a 的值从内存区域中拷贝 到变量 i 的内存区域中		1	1
解锁处理		1	1
	加锁处理	1	1
	从变量 i 的内存区域中将值拷贝到变量 a 的内存区域	1	1
	给变量 a 加 1	1	2
	将变量 a 的值从内存区域中拷贝到变量 i 的内存区域中	2	2
	解锁处理	2	2

```
lock();    ←加锁处理
i = i + 1;
unlock(); ←解锁处理
```

加锁处理同时只能调用一次，在某一进程调用加锁处理时其他进程会处于等待状态。

P2P MO 和竞争状态

在 C/S MMO 游戏中，管理着全部游戏状态的游戏服务器一般会进行加锁处理。玩家的任何操作都需要请求服务器进行实际的数据操作，游戏状态只能通过服务器进行变更，所以不会发生竞争状态。

P2P MO 游戏也可以采用相同的方式，只是将玩家的机器当做游戏服务器，以“分布式 MMO”的方式实现，从而避免竞争状态的发生。最近这种实现方式的使用逐渐增多，但是其存在的问题是“一旦主机在游戏中途退出”，其他玩家就不能切换到单人模式继续游戏。另外，MMO 游戏的进行完全依赖其他的进程这点会造成网络延迟，所以对于激烈的动作类游戏这种方式也不适合。

P2P MO 游戏其实还有与前面描述的“加锁处理”类似的实现办法。不过这个方法和一般程序中使用的 mutex 或者 FIFO（先进先出）等方式不同，为了避免不同机器上运行的进程之间发生竞争状态，需要实现带有通信功能的加锁处理。在不同机器上运行的进程为了进行加锁处理需要和主机之间进行通信，所以理论上和前面描述的分布式 MMO 有同样的网络延迟问题。但是，好处是中途退出时可以切换到单人模式继续游戏。

5.5.7 P2P MO 游戏开发中该如何防止发生竞争状态

综上所述，在 P2P MO 游戏的开发中，判断该如何防止共享内存中数据发生竞争状态十分重要。对于游戏内容的充分理解可以帮助开发者做出正确的选择。

5.3 节中“带宽消耗的预估”部分为了节省带宽决定“不同步没有追击玩家的敌人”，这个节省带宽的实现方式也是最重要的判断依据。

同步处理的实现例子

下面是关于同步处理正式编码前的记录。这里展示的是未经修改的实际文档，阅读可能稍微有点困难。我们以此为基础来说明在实际编码中应该注意的地方。

```
* Enemy
  * Create: host -> guest (sync, ALL)
  * Move
    * target（有）: guest 处理
    * target（无）: host -> guest (sync, coord)
  * Delete: guestAttackKill -> hostKill -> guestKill (delete ALL)
* PlayerCharacter
  * Create, Move, Delete: guest -> host -> guest (sync ALL)
* Bullet
  * Create, Move, Delete: local
```

```

```

上面这些记录是 *J Multiplayer* 游戏中关于可动物体同步的设计资料，但具体该怎么设计呢？

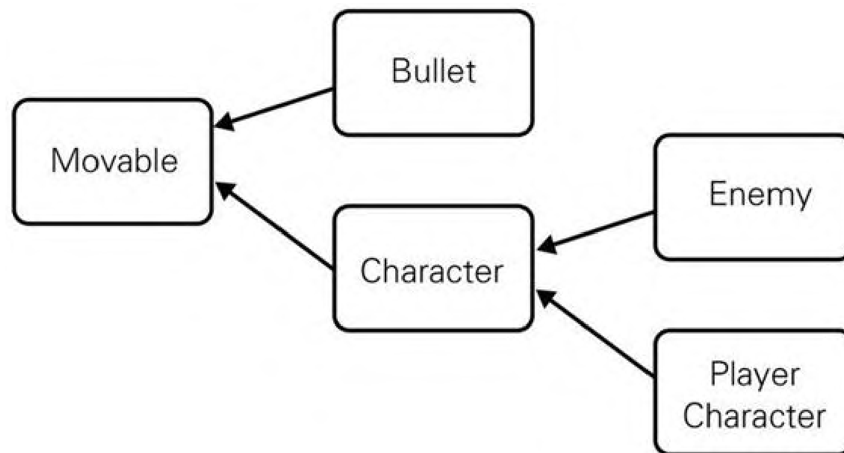
可动物体的枚举类型和类的设计

首先一起来看看在游戏中出现的所有可动物体。

```
* Enemy
* PlayerCharacter
* Bullet
```

J Multiplayer 客户端程序中可动物体的类设计如图 5.12 所示。各个类的要点如下所示。

图 5.12　*J Multiplayer* 可动物体的类的设计



- class Movable

  可动物体的基类。包含坐标、移动方向、图像编号等信息。

- class Character

游戏角色类，继承 Movable 类。可以互相攻击，包含 HP 等状态信息。

- class Enemy

  敌人角色类，继承 Character 类。具有管理攻击值的人工智能（AI）。

- class PlayerCharacter

  玩家角色，继承 Character 类。能通过网络来控制。

- class Bullet

  子弹类，继承 Movable 类。可以击中 Enemy 或者 PlayerCharacter。

基本操作的矩阵

接下来，像在 DB 中进行 CRUD 那样，将基本操作 Creat、Move、Delete 矩阵化，这些都是机械化的工作。

```
* Enemy
  * Create
  * Move
  * Delete          ↗
* PlayerCharacter
  * Create
  * Move
  * Delete          ↗
* Bullet
  * Create
  * Move
  * Delete
```

修改游戏进行状态相关处理的规范

创建敌人、移动玩家角色的操作都属于修改游戏进行状态的处理，这些处理有两点需要考虑。

- **最初在哪里发生？**

- **应该通知哪里？**

下面，我们来针对这两点分别说明各个类的相关处理规范。

“哪里”主要只是主机和客户端。如果是在主机发生，并且需要发送信息到所有客户端的话，记为：host -> guest

玩家角色的处理规范

首先从简单的地方开始规范。

玩家角色最多同时出现 4 个，对带宽的影响很小。这其中，其他玩家的位置和他们正在做什么是最重要的信息，所以这些信息需要一直保持同步。另外，其他进程无法修改玩家角色自身的状态，所以玩家 A 操作的时候，A 的角色信息会得到更新，而其他玩家则不能修改这个角色的信息。

因此，Create、Move、Delete 这些操作只在各自的客户端执行，然后向主机发送信息，再由主机通知全部客户端。可以简化记为 guest -> host -> guest 。下面的代码（sync ALL）表示使用 sync 通信协议，向所有的客户端发送信息。

```
* Create, Move, Delete: guest -> host -> guest (sync ALL)
```

敌人的处理规范 ——Enemy/Create

下面是最重要的 Enemy 类，敌人的处理是比较复杂的。

首先 Create 操作是指生成新的敌人。根据笔者的经验，在 P2P MO 游戏中生成新的可动物体有两种方法。

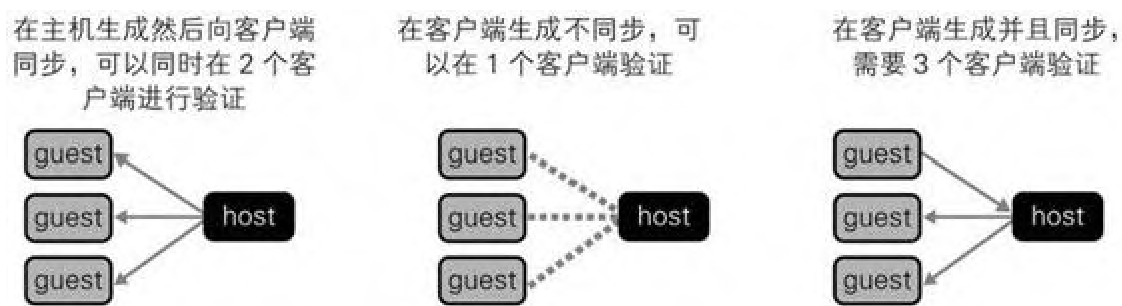
- **在主机生成。**

- **在客户端生成，但不同步。**

这两种方法在设计上不容易出现问题。

这是因为在客户端生成的物体如果要同步到其他机器需要经过两次跳转，系统整体可能出现的状态会增加（图 5.13）。

图 5.13　P2P MO 类型游戏可动物体的新生成



如图 5.13 所示，如果采用客户端生成并同步的方式，为了调试需要使用 guest - host - guest 三个进程，重现 Bug 比较麻烦。另一方面，系统可能出现的状态增加为以下三种。

- **在客户端 A 生成后还没有向其他机器同步。**
- **客户端 A 通知了主机，但主机还没有通知其他机器。**
- **客户端 A 通知了所有的客户端。**

敌人状态的同步是这个游戏最重要的部分，如果因为对应的状态管理变得复杂而增加了验证的难度，那就有可能导致不必要的技术上的问题。

为了实现 *J Multiplayer* 的设计内容，应该慎重考虑在客户端生成的可动物体是否需要同步到其他客户端。在示例代码中没有进行同步，所有生成敌人的处理也都由主机来控制，这样可以将产生问题的风险降到最低。不仅游戏内容的开发会变得更加容易，调试会比较方便，而且能够缩短开发周期。

简而言之，就是只在主机生成敌人，然后同步到所有客户端。可以调用 sync 协议（sync, ALL）来进行同步，如下面的代码所示。

```
* Create: host -> guest (sync, ALL)
```

以上就是敌人的创建。

- Enemy/Move

  接下来是 Move 操作，需要规范的是当敌人移动时，应该同步的信息和同步的方式。

  如果 Create 不同步的话，Move 或者 Delete 等后续的操作当然也不需要同步。Move 操作也是 host -> guest 。

  在 5.3 节中提到过，*J Multiplayer* 游戏的设定是“追击玩家的敌人在客户端本地机器上进行处理，分散在地图上处于待机状态的敌人由主机控制移动并向各个客户端同步数据”。

  在这里可以将该设定加入到程序的规范中。在 Enemy 类里添加成员变量 targetID，用来记录追击目标的 ID，即被追击的玩家 ID。没有追击目标时设为 0，追击 ID3 的玩家时设为 3。

  ```
  * Move
    * target（有）: guest 处理
    * target（无）: host -> guest (sync, coord)
  ```

  上述“target（有）”表示 targetID 不为 0。这种情况时，“guest 处理”表示在 target 所在的客户端处理；“target（无）”表示没有追击目标，通过 host -> guest 进行同步，使用 sync 通信协议来传输坐标（Coordinate），而且只同步坐标，不用传输 HP 等信息。当然大 Boss 这样需要很长时间才能击倒的敌人可能需要同步 HP，但是虾兵蟹将的话同步下位置信息就可以了。通过这些判断，在实现游戏内容的过程中，也许能够找到如何减少带宽消耗、处理内容、编码量等问题的方法。

- Enemy/Delete

最后是 Delete 操作，在 *J Multiplayer* 游戏中敌人死亡的唯一条件是受到攻击。在 5.3 节中提到过的设定是“敌人的数量需要同步”，这也可以直接作为程序的规范。

敌人的死亡有以下两种情况。

❶ 在客户端被玩家击倒。

❷ 在主机被主机的玩家击倒。

```
* Delete: guestAttackKill -> hostKill ->guestKill(delete ALL)
```

第 1 种情况时，如果在客户端 A 被击倒，调用 delete 通信协议以 guestA->host-> 其他客户端的顺序同步。而在主机被击倒时，顺序则是 host-> 其他客户端。用总结成一行的方式来描述可能更为明确。

```
* Delete: [ guestAttackKill-> ] hostKill -> guestKill(delete ALL)
```

[] 内表示可以省略

以上大体明确了 Enemy 同步处理的规范。

子弹的处理规范

还剩下子弹相关的处理。

```
* Bullet
* Create, Move, Delete: local
```

只有正在追击目标的敌人才发射子弹，四处分布的待机敌人不会发射子弹，因此可以在各个客户端分别处理而不需要同步。

因为设定太过简单反而有些犹豫，但是经过实际测试并没有发现问题，所以还是把它作为规范保留了下来。

在实际开发中随着测试的不断进行，会发现“这里还需要更多同步”、“这个不需要吧？”、“这里的同步频率不提高的话可能会有问题”、“装备这个武器的时候需要提高同步频率”、“只同步这个敌人”等问题，所以需要根据游戏内容，在深入调查的同时不断调整同步的处理。

5.5.8 共享内存开发方式该如何编码——共享内存开发方式和 RPC 开发方式的比较

如果提到共享内存一般是指 POSIX 规格的共享内存（shm_ 函数）。

但是本章的“共享内存”只是为了与 RPC 开发方式进行比较，和 POSIX 的 shm 函数并没有关系。这里稍微有点复杂，所以需要明确说明“共享内存开发方式”和“RPC 开发方式”的区别。

二者在实现上的根本区别是：在游戏逻辑中，游戏进度数据的保存处理（覆盖或者不覆盖数据）所发生的时间和地点不同。

RPC 开发方式 ——C/S MMO 游戏

C/S MMO 游戏采用的 RPC 开发方式是按照下列顺序更新游戏进度数据的。

- 玩家在客户端（程序）进行角色移动的操作。
- 客户端向服务器端发送操作的 RPC 请求。

  例：向服务器发送 move(5, 5) 函数。

- 服务器接收到 move(5, 5) 函数的请求后，首先检测参数 (5, 5) 是否正确，如果正确就在游戏进度数据中将角色的位置坐标更新为 (5, 5)。

- 服务器更新数据成功后，将新的值同步到其他客户端。

采用 RPC 开发方式时，网络上传输的是“变更的请求”，而不是“变更的结果”。所以根据处理结果的不同，有可能会发生值并没改变的情况。

共享内存开发方式 ——P2P MO 游戏

P2P MO 游戏采用的共享内存开发方式要求客户端和服务器端共享相同的游戏数据。在这个前提下按照下列顺序处理游戏数据。

- 玩家在客户端（程序）进行角色移动的操作。
- 客户端检测移动操作是否正确，如果没有问题则更新本机管理的游戏进度数据。
- 客户端将更新后的数值传输到其他客户端。
- 其他客户端无条件接收更新后的结果。

采用共享内存方式时，网络上传输的是“变更的结果”，而不是“变更的请求”。所以只需要无条件替换对应的数值。

如果灵活运用共享内存开发方式的特性就能够大幅度减少代码量，这也是采用这种方式的优点。

为了体现这个优点，我们以 move(5, 5) 为例来比较 RPC 开发方式和共享内存开发方式在代码量上的区别。

RPC 开发方式的代码量 ——move(5, 5)

RPC 开发方式的代码参考以下示例。

```
void player_touched(mouseX, mouseY)
{
  send_move(mouseX,mouseY);

 ←†
}
void recv_move( int x, int y )
```

```
     ←†
{
if( check(x,y) == OK ){
data->x = x;
data->y = y;
data->broadcast_move_notify(x,y);

        ←†
}
}

void recv_move_notify(int x, int y)

 ←†
{
character->x = x;
character->y = y;
}
```

要增加 z 参数的时候，需要在标记了“†”符号的 4 个函数，即客户端的 send_move 函数和服务器端的 recv_move 函数、broadcast_move_notify 函数、客户端的 recv_move_notify 函数中都添加 int z 参数。而包含 int x, int y 两组参数的地方也都必须添加相应参数。

为了避免这样的麻烦，RPC 所有的参数都可以使用同一个参数构造体，但是这样难免会降低代码的可读性。

共享内存开发方式的代码量 ——move(5, 5)

共享内存开发方式的代码参考以下示例。

```
void player_touched(mouseX, mouseY)    ←*
{
  character->x = mouseX;
  character->y = mouseY;
  character->changed(VAL_X);
  character->changed(VAL_Y);
  character->broadcastChanged();
}
void recv_changed(int type, int value)
{
  switch(type) {
    case VAL_X : character->x = value; break;
    case VAL_Y : character->y = value; break;
    default:break;
  }
}
```

要增加 z 参数时，代码如下。

```
void player_touched(mouseX, mouseY, mouseZ)
{
  character->x = mouseX;
  character->y = mouseY;
  character->z = mouseZ;

  ←†
  character->changed(VAL_X);
  character->changed(VAL_Y);
  character->changed(VAL_Z);

  ←†
  character->broadcastChanged();
}

void recv_changed(int type, int value)
{
  switch(type) {
    case VAL_X : character->x = value; break;
    case VAL_Y : character->y = value; break;
    case VAL_Z : character->z = value; break;
```

```
    ←†
    default:break;
  }
}
```

把上面标记了“†”符号的 3 个地方修改后就可以告诉别的进程 Z 坐标发生了变化。因为是无条件接收改变后的值，并没有返回值，所以代码量也会有很大不同。而且如果想要“只修改 X 和 Z”，采用 RPC 方式不仅需要定义新的函数，还需要重新编译相关的文件。而采用共享内存方式则不需要增加新的函数，只要修改并编译相关文件即可。

共享内存开发方式的优缺点

虽然增加参数这样的调整不能使用 IDE 的重构工具，但是代码的修改还是比较容易的。在游戏开发中，需要一边试玩一边进行无数次的细微调整，因此迭代的速度直接决定了游戏的品质。通过比较可以发现“采用共享内存开发方式更容易进行细微的修改，有利于追求更高的品质”。

共享内存方式的缺点是即使只有少数几个数值发生了变化也需要进行通信，这样可能会产生一些多余的通信量。

[补充] 你会进行代码清洁工作吗？

众所周知，代码应该尽可能保持精炼。上节“共享内存开发方式的代码量”列出的代码清单中有一处标记“*”号的地方，这个地方可以使用下例中的访问器（Accessor）来实现自动化同步。

```
void player_touched(mouseX, mouseY, mouseZ)
{
character->setX(mouseX); character->setY(mouseY);
character->broadcastChanged();
}
void Character::setX(float value)
{
this->x = value;
this->changed(VAL_X, value);
}
```

当然在这种程度下用访问器来实现自动化是可行的，不过当可动物体有成百上千个时，这些处理每次访问成员变量时都要调用 changed 函数，这点会产生性能上的问题。

所以应该直接访问原生类型的变量，处理完结果后再调用 changed 函数。因此实际的 *J Multiplayer* 案例代码并不像上面例子中的那样简洁，具体情况在后面的章节中会详细说明。不过对于可动物体少的游戏——比如将棋这类游戏——就可以采用上面那样简洁的做法。

5.5.9 SyncValue 类

在 *J Multiplayer* 案例代码中，SyncValue 类的实现采用了前面提到的共享内存方式。SyncValue 是用来记录在共享内存中变化部分的类，而实际用来处理可动物体的内存区域则是通过别的方式管理的。

首先来看可动物体类 Movable，为了方便说明做了以下变化。

```
↓可动物体类
class Movable {
 public:
  U32 id;  ←游戏房间内的唯一ID
  float x,y;         ←现在的位置
  float dx, dy;      ←方向
  U32 typeID;        ←种类ID

  SyncValue *m_pSyncValue;    ←同步用内存指针

  enum {
    SVT_TYPEID,
    SVT_X,
    SVT_Y
  };

  ↓构造函数
  Movable(U32 _id, float _x, float _y,  U32 _typeID)
     : id(_id), x(_x), y(_y), typeID(_typeID)
  {
    m_pSyncValue = new SyncValue();
    m_pSyncValue->registerIntType(SVT_TYPEID);
    m_pSyncValue->registerFloatType(SVT_X);
    m_pSyncValue->registerFloatType(SVT_Y);

    m_pSyncValue->setInt(SVT_TYPEID, typeID);
```

```
    m_pSyncValue->setFloat(SVT_X,  x);
    m_pSyncValue->setFloat(SVT_Y,  y);
  }
};
```

正如前文所述 [14] ，为了保证执行效率，需要直接访问原生 float 型坐标值 x，y。构造函数通过指针在缓存中记录 SyncValue 内的数值变化，使用 registerIntType 函数记录变量类型，调用 setInt 函数修改实际的值。

> [14] 是指上一节“[补充] 你会进行代码清洁工作吗？”的末尾部分提到的“直接访问原生类型的变量”。

在游戏的主循环中，同步可动物体移动后变更值的代码可以简化如下。

```
x += dx;
y += dy;
this->m_pSyncValue->setFloat(SVT_X, x);
this->m_pSyncValue->setFloat(SVT_Y, y);
```

通过上述代码在 SyncValue 类的实例中记录了变更结果，所以如果在主循环之外的地方有修改，就可以通过网络发送修改信息。

```
std::vector<Movable*>::iterator itm;
for (itm = vm.begin(); itm != vm.end(); ++ itm) { ←遍历所有的Movable
  Movable *m = (*itm);
  vce::VUint8 buf[256];
  size_t buflen;
  m->m_pSync->getDiffBuff(buf, &buflen);      ←只取得缓存中变化了的部分

  if (buflen > 0) {    ←如果有需要同步的就发送信息
    broadcast_sync(m->guestid, m->id, buf, buflen);
    }
    m->m_pSync->clearChanges();      ←信息发送完成后删除变更记录
}
```

```
```

broadcast_sync() 函数只取出缓存中变化的部分并发送信息。

收到消息的一方处理如下所示。

```
void receive_sync(U32 id, const U8 *data, U32 data_qt)
{
  SyncValue sval;   ←声明SyncValue 变量

  ↓注册数据格式
  sval.registerIntType(SVT_TYPEID);
  sval.registerFloatType(SVT_X);
  sval.registerFloatType(SVT_Y);
  ↓从网络中读取缓存数据
  sval.readBuffer(data, data_qt);
  Movable *m = g_app->getMovable(id);   ←通过ID 获取 Movable 实例的指
针

  ↓如果有变更就同步（直接覆盖写入本地内存）
  if (sval.isChanged(SVT_TYPEID)) {
     m->typeID = sval.getFloat(SVT_TYPEID);
  }
  if (sval.isChanged(SVT_X)) {
      m->x = sval.getFloat(SVT_X);
  }
  if (sval.isChanged(SVT_Y)) {
      m->y = sval.getFloat(SVT_Y);
  }
}
```

为了实现上述操作，SyncValue 类需要有以下成员变量。

```
class SyncValue
{
 private:
  static const int SYNCVALUES_MAX = 32; ←以同步的变量个数上限
  static const int BUFLEN = 128;
  vce::VUint8 m_buf

[BUFLEN];   ←不保存大量数据所以设为固定值
  size_t m_offsets
```

```
[SYNCVALUES_MAX];      ← ID，从哪一位开始；如果是 0 则为 m_buf[0]；类型种类决定大小。这个在add 操作时确定
  size_t m_sizes

[SYNCVALUES_MAX];        ←不同变量的大小
  bool   m_changes

[SYNCVALUES_MAX];      ←记录变化了的ID
  bool m_uses[SYNCVALUES_MAX];          ←使用了哪些域
  size_t m_currentOffset;
```

在上述代码中，m_buf 变量用来记录使用 setFloat 函数变更后的数据，m_offsets、m_sizes 保存变量和变量所在的位置。m_changes 保存变更的变量 ID，m_uses 表示使用的变量。这些数组都使用了 SVT_ 枚举型中定义的值，当最大值大于 SYNCVALUES_MAX 时就会报错。

SyncValue 类的构造函数仅使用默认值 0 初始化所有的变量。

```
public:
  ↓ SyncValue 类的构造函数。必要的变量都初始化为0
 SyncValue() : m_currentOffset(0), m_offsets(), m_sizes(),
m_changes(), m_uses() {
}
```

SyncValue 类现在支持的 3 种类型都是比较实用的类型，可以在构造函数内进行无限扩展。reg 是内部使用的函数，计算并保存缓存的使用方式。

```
void registerCharType(int index) {
  reg(index, 1);
}
void registerIntType(int index) {
  reg(index, 4);
}
void registerFloatType(int index) {
  reg(index, 4);
}
```

setChar、setInt、setFloat 函数通知发生实际变化的数据。

index 是 SVT_ 枚举型。更新 m_buf 中的数值后将 m_changes 设为 true，之后就可以发送消息同步数据。

```
void setChar(int index, char val) {
  char prev = ((char*)(m_buf+ofs(index)))[0];
  if (prev != val) {
    ((char*)(m_buf+ofs(index)))[0] = val;
    m_changes[index]=true;
  }
}
void setInt(int index, int val) {
  int prev = ((int*)(m_buf+ofs(index)))[0];
  if (prev != val) {
    ((int*)(m_buf+ofs(index)))[0] = val;
    m_changes[index]=true;
  }
}
void setFloat(int index, float val) {
  float prev = ((float*)(m_buf+ofs(index)))[0];
  if (prev != val) {
    ((float*)(m_buf+ofs(index)))[0] = val;
    m_changes[index]=true;
  }
}
```

上面是 set 相关函数，接下来是 get 相关函数。

```
char getChar(int index) {
  return ((char*)(m_buf+ofs(index)))[0];
}
int getInt(int index) {
  return ((int*)(m_buf+ofs(index)))[0];
}
float getFloat(int index) {
  return ((float*)(m_buf+ofs(index)))[0];
}
```

使用 get 函数取得变量值是很简单的处理，只需从 m_buf 中获取数值。下列 clearChanges、fillChanges、isChanged 函数用来清除变更标记、设定变更标记和取得变更标记。

```
void clearChanges

() {
  fillChanges(false);
}
void fillChanges

( bool flag ) {
  for (int i=0;i<SYNCVALUES_MAX;i++) {
    if (m_uses[i]) {
      m_changes[i]=flag;
    }
  }
}

bool isChanged

(int index ) {
  assert(index < SYNCVALUES_MAX);
  return m_changes[index];
}
```

下面的 ❶ 号函数 getDiffBuff 将 m_buf 中变更的部分全部输出到缓存 outbuf 后，返回该变量。

❷ 号函数 readBuffer，从缓存参数中读取数据并且判断哪些变量发生了改变、变更值为多少，从而为 get 操作准备必要的信息。

```
void getDiffBuff(vce::VUint8 *outbuf, size_t *outbuflen);   ← ①

void readBuffer(const vce::VUint8 *inbuf, size_t inbuflen); ← ②
```

＊ ＊ ＊

以上就是 SyncValue 类的相关内容。

在实际编码中，j.xml 文件中定义的通信协议 IDL 和 C/S MMO 游戏 *K Online* 中的有所不同，只有 5 种类型，而且无论什么游戏都差不多。这在开发实际通信处理阶段已经做了说明。

经过了这些准备，剩下的就是在实际开发中一边试玩，一边讨论各种细节，在实现游戏内容的过程中不断调试。至此，关于 P2P MO 游戏实现的说明就告一段落。

专栏　数据中心的地理位置分布

K Online 或者 *J Multiplayer* 这样的网络游戏和 Web 服务相比需要更高的响应速度，为了保证游戏体验应该尽可能在离玩家近的地方，甚至是玩家所在国家或地区设置服务器。在日本和北美两个区域使用同一服务器提供游戏服务已经是能做到的最大限度。

不过也并不是每个游戏公司都有能力在多个国家设置自己的分支机构，一般的做法是将游戏的服务器代码、编译后的程序或者设定文件等授权给不同国家的运营商，由这些运营商来提供服务而己方仅收取授权费（License Fee）。当然利用的是当地的数据中心等资源。

这种情况下应该尽量避免不同地区的数据中心互相连接，让数据库之间没有相互依赖的关系，保持完全独立的运行状态。

现在网络环境在不断改善，有通过使用统一的 Web 站点来降低成本的趋势，但是距离“单一位置”服务器提供游戏服务还需要很长时间的发展。

5.6　支持 C/S MO 游戏的技术 [补充]

本章最后部分说明一下支持 C/S MO 游戏的技术。需要注意这里是 C/S MO 而不是 C/S MMO。

5.6.1　C/S MO 和 NAT 问题

C/S MO 是 Client/Server 型 MO（少数用户在线的）游戏的简称。之前提到过很多次，P2P MO 游戏最大的问题是“不是任何玩家都可以直接连接”。例如，当 *J Multiplayer* 使用星状拓扑结构时，直接与网络连接但是没有公网 IP 地址的玩家机器不能做主机，这就是“NAT 问题”。

5.6.2 什么是 NAT 问题

据笔者所知，关于 NAT 问题科乐美公司（Konami Digital Entertainment, Inc.）的佐藤良先生在网上发布的资料最为详细。其中的重点部分请参考图 5.14。

http://homepage3.nifty.com/toremoro/study/voip2008/NATTraversal.pdf

注意图 5.14 中“公开网络（Open Internet）”部分，在韩国可以作为主机的用户比例很高，而在日本几乎为 0，而且不同国家的比率都不相同。

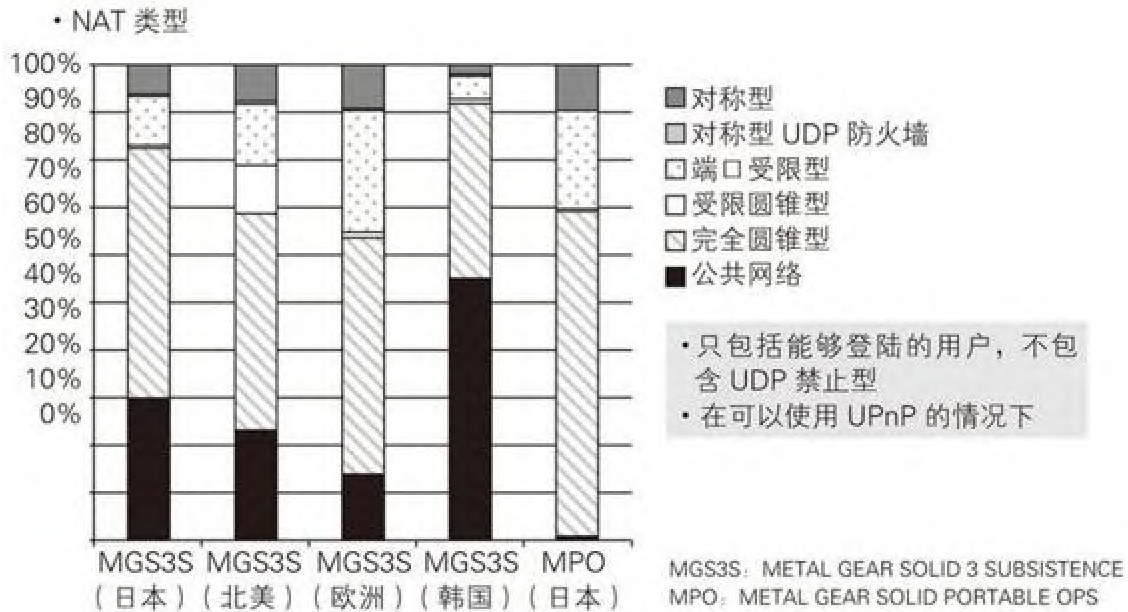
一般说来，如果家庭使用 Softbank 等运营商提供的“宽带路由器”连接网络，PC 的 IP 地址和路由器的外网 IP 地址是不同的，这时 NAT 是有效状态。因为 PC 的 IP 地址和路由器的外网 IP 地址通过某种规则可以转换，所以叫做网络地址转换（Network Address Translation）。使用 NAT 不仅可以解决公网 IP 地址不足问题，还具有提高安全性和简化网络连接等优点。

不过网络游戏需要和网络上的其他 PC 联网，而 NAT 可能会造成网络无法连接的问题。

5.6.3 NAT 遍历——解决 NAT 问题建立通信路径的技术

图 5.14 说明了 NAT 遍历（traversal），即解决 NAT 问题建立通信路径的技术。该技术不仅在网络游戏中使用，在 VoIP（Voice over IP）、网络会议系统、VPN 等领域也有广泛应用，因此市场上除了各种中间件产品，还有很多开源产品。

图 5.14　NAT 问题（NAT 问题的统计数据）



※ 出处：科乐美公司（Konami Digital Entertainment, Inc.）佐藤良

“P2P 通信技术：NAT 穿越～ STUN、UPnP 和 TURN”（p36“实际标题的统计数据（2）”）

http://homepage3.nifty.com/toremoro/study/voip2008/NATTraversal.pdf

首先按限制程度将互联网用户的 NAT 状态划分为 6 个阶段，如下所示，强度逐渐增加。

❶ 公共网络

拥有公网 IP 的终端。

❷ 无限制 NAT →图 5.14 中的“完全圆锥型”（Full cone）

访问路由器的指定端口时，固定转换为 LAN 中 PC 的特定端口。可以传输所有的收到的数据包。

❸ 限制地址的 NAT →图 5.14 中的“受限圆锥型”（Address——Restricted cone）

只接收来自于 NAT 内部，并且之前有发送过数据包的终端的数据包。通过设定 IP 地址提高安全性，不接受之前没有产生过通信的 IP 地址所发送的数据，这也叫做“受限圆锥形”。

❹ 限制端口的 NAT →图 5.14 中的“端口受限型”（Port——Restricted cone）

和限制地址的 NAT 相同，不过又增加了端口号的限制，也叫做“地址 / 端口受限型”

❺ 对称 NAT（Symmetric NAT）→图 5.14 中的“对称型 UDP 防火墙”

每次建立通信路径时都使用不同的端口号，不同的地址。这个方式有很多分支类型。

❻ UDP 禁止型 →图 5.14 中的“对称型”

禁止使用 UDP 通信

NAT 遍历技术的限制

以现有的技术基本可以解决 ❶ 到 ❹ 中的 NAT 问题。针对 ❺ 的情况可以使用“UDP Hole Punching”[15] 、“STUN”[16] 等技术，不过不能保证 100% 解决问题。❺ 的一部分和 ❻ 的 NAT 问题目前还无法解决。

> [15] 不同 NAT 路由器下面的局域网内，主机之间经过互联网使用 UPD 协议建立连接的方式。

> [16] Simple Traversal of UDP through NATs 的简称，使用标准化协议的一种 NAT 遍历方法。

以网络游戏玩家作为对象调查了“无法解决的 NAT 问题比率”，其结果请参考之前介绍的资料 [17] 。

[17] 十分感谢将这些宝贵数据公开的公司。

使用 NAT 遍历技术的其他缺点

使用 NAT 遍历技术的另一个缺点是，在玩家和其他 PC 实际建立连接交换数据之前需要各种初始化处理和验证处理，这不但需要花费数秒至数十秒的时间，还需要专门的服务器，所以产生了成本问题。此外，E❺ 的一部分玩家和 ❻ 的玩家的 NAT 问题也暂时无法解决。

开发网络游戏的时候，实现 NAT 遍历技术是十分重要的。在 2000 年左右，各个公司都采用 UDP 的方式实现 NAT 遍历功能。但是现在互联网基干网络的性能有了大幅度提高，东京都内企业间的通信基本都在 1 毫秒之内，此外 Amazon 等提供云计算服务的企业在世界各地都建立了数据中心[18]，这使全世界范围内构建游戏服务器也变得越来越容易，所以今后为了避免网络游戏中 NAT 遍历技术的缺点，使用 C/S 型架构的游戏会越来越多。本书对于 NAT 遍历相关的探索就不再深入。

[18] 也称作边缘定位（Edge Location）。

5.6.4 NAT 问题的实际解决方法

NAT 问题的一个实际解决案例如下。

- **在 Microsoft 的 Xbox 360 标准程序库中，为了与别的玩家建立连接，采用了多个阶段的通信确保数据包能够到达。**
- **如果还是有问题，则使用中继服务器建立连接。**

因为大部分的开发者并不是为 Xbox360 平台开发游戏，这里主要介绍使用中继服务器的方法。使用中继服务器可以让用户的连接失败率降到几乎为 0[19]。

[19] 不能保证完全为 0 是因为有的 ISP 禁止 HTTP 以外的通信。

5.6.5 中继服务器

中继服务器是指第 3 章介绍的“反射型架构”的数据包中继服务器。这种服务器既不检测数据包中的数据，也不需要认证。具体请参考第 3 章图 3.10。

调用 SyncValue 的逻辑在 *J Multiplayer* 中直接保留，只需要修改物理网络的使用方法。

J Multiplayer 最初的原型没有使用反射型架构，而是采用了直接连接的 P2P/ 星状拓扑结构。所以当 4 个人游戏时，主机接收 3 个客户端的连接。如图 3.12 所示。

使用中继服务器的时候，主机之外的客户端连接的 IP 地址仅从主机地址变成中继服务器的地址，所以主机的操作只需要稍微调整一下即可。

具体说来主机需要添加以下功能。

- **启动时连接中继服务器，创建新的通信频道（channel）。**
- **广播时，不向客户端直接发送数据，而是向中继服务器发送。**

此外在客户端，为了获取中继服务器的通信频道信息，需要使用匹配服务器等辅助系统（参考第 6 章），还要从主机获取通信频道的 ID。

使用中继服务器进行交换的数据和使用直接连接方式时的一样，都是 SyncValue 的差分数据。

5.6.6　中继服务器的折衷方案

实际上中继服务器的实现十分简单，不过在程序开发上有两个需要折衷的地方。

❶ **因为通信需要经过服务器，所以会有游戏延迟。**

❷ **需要消耗服务器带宽。**

将影响降低到最小正是需要开发者大显身手的地方，虽然 Xbox 360 平台有提供解决方案，不过还可以进一步得到改善。

❶ 游戏延迟增大的问题

首先关于游戏延迟增大的问题，并不是每个用户都需要使用中继服务器，让那些怎么都无法建立主机的玩家使用就可以了。这样两种方式共存，在一定时间内不能建立通信的情况下切换成使用中继服务器的模式。

❷ 消耗服务器带宽的问题

关于服务器带宽的消耗问题有几个阶段的折衷方案。虽然可以尽可能的压缩带宽消耗，不过这样会增加开发时间。

这里介绍几种方法。

- **什么都不调整的时候**

首先来看看什么都不调整的情况。假设 4 个人同时进行 *J Multiplayer* 游戏，主机需要 100kbit/s 的带宽，那么 1 个人做主机有 100kbit/s 带宽，向其他 3 个客户端发送数据，每个客户端需要 33kbit/s。

- **主机：[上行]33kbit/s×3=100kbit/s、[下行] 非常少。**
- **客户端：[上行] 非常少、[下行]33kbit/s。**

通信量 [非常少] 是因为游戏开发规范中要求同步敌人数据的时候只同步 Create/Delete 操作。

综上所述 4 个人同时连接时需要中继服务器 100kbit/s 的上行下行带宽，如果以同时连接数 4000 为基准，需要 100kbit/s×(4000/4)=100Mbit/s 带宽。目前这个带宽的每月花费大概在几万到几十万日元之间。这不是一笔小数目，所以需要考虑该如何解决这个问题。

- **解决方案1**

4 个人有中如果有 1 个人可以使用 NAT 遍历，那么就让这个人作为主机开始游戏，同时作为中继服务器进行通信。

假设按照刚才图 5.14 的数据，在日本可以使用 NAT 遍历的用户大概是 7 成。4 个玩家在线时，不能使用 NAT 遍历的概率是 0.3 除以 4 等于 0.008，不到 1%。所以服务器成本马上缩减为 1/100。不过这样可能会遇到通信测试时间很长的问题，全部玩家都测试的话估计要花费 1～2 分钟。

在服务器保存通信测试结果也许可以缩短等待时间（不过笔者目前还没遇到这样的做法）。

- 解决方案2

  将同步数据拷贝到中继服务器。

  主机向其他 3 个客户端发送的同步数据都是一样的，如果主机将发给 3 个客户端的相同数据包改为只发送一次同步数据，由中继服务器将该数据发送给其他 3 个人，这样也可以减少上传数据消耗的带宽。4 个玩家时所需带宽如下。

  - 主机：[上行]33kbit/s×1=33kbit/s
  - 客户端：[下行] 33kbit/s

  上行带宽一下减少到 1/3，整体来看可以减少 2 成左右。不过这种方案需要修改中继服务器的程序。

* * *

使用中继服务器开发 MO 游戏时，需要注意上面两点。如果不实现这两点的优化，服务器成本可能会很高，难以实现游戏的商业化运营。

最后中继服务器相关辅助系统的话题会在第 6 章继续说明，读者可以参考其中的内容。至此，MO 游戏开发相关的话题就告一段落。

5.7　总结

本章以 P2P MO 游戏为中心进行了开发相关的说明，同时也对比了其与 C/S MMO 系统的区别。下一章我们将会和读者一起探讨支持网络游戏的辅助系统。

第 6 章　网络游戏的辅助系统：完善游戏服务的必要机制

第 4 章、第 5 章主要说明了如何实现玩家的游戏体验，也就是网络游戏的“游戏内容”（游戏策划内容）部分。为了追求游戏的独创性，我们提到了很多网络游戏特有的技术。

本章不再讲述网络游戏设计 / 游戏程序的相关内容，主要为大家介绍支持网络游戏的“辅助系统”的实现方法。之前我们曾提到一些相关的辅助系统，本章会介绍具体的内容，包括目录中列举的玩家匹配、聊天功能、排行榜、计费认证等。

大部分项目都会尽可能削减辅助系统开发的预算，如果可以就使用其他公司提供的现有服务。这部分的开发时间能省即省，将主要精力集中在游戏内容本身的开发上。

但另一方面，如果认证或者玩家匹配等辅助系统的开发进展不顺利也会影响到游戏主体部分的体验，导致游戏的价值大幅下降。很多游戏在刚发行或者举行活动时会发生“因为超过系统负荷而无法登录游戏”的情况，有过这种经验的玩家应该不在少数。笔者认为一个成功的辅助系统对于游戏的整体价值来说，不是游戏内容的价值加上辅助系统的价值，而是二者相乘的结果。不重视辅助系统就会让游戏的整体价值大为降低。

虽然即使辅助系统进展不顺利也不会中止游戏的发售，但是也应该尽可能利用现有的成熟服务来降低开发风险。

自己开发辅助系统时很少会用游戏开发的特殊方法，一般都是采用系统开发或者 Web 开发的技术，其他公司提供的现有系统基本也是如此。一般这些系统都有性能上的瓶颈，如果理解了本章介绍的这些辅助系统的实现原理，那么不论是自己开发还是利用现有系统都能在技术上降低风险。

接下来，我们一起来看一下辅助系统相关的详细内容。

6.1　辅助系统需要的各种功能

这里我们会参考多个辅助系统，整理出辅助系统所需要的功能。

6.1.1 现有服务提供的辅助系统功能

下面我们将服务分为通用、游戏主机用、网页游戏用、现有中间件几类，一起来看一下其中包含的辅助系统。

通用

通用服务可以用于多个平台，并且没有游戏类型的限制。例如，下面的 Uplay 服务可以为游戏主机、iPhone 等移动平台以及 PC 等提供服务。Plus+ 可以用于 C/S MMO 游戏和 P2P MO 游戏。

- Steam

  官方网站：http://store.steampowered.com/

  开发者网站：http://www.steampowered.com/steamworks/

  这是 2010 年在全世界范围拥有 3000 万以上用户的大型服务。最大的特点是具有功能强大的销售渠道，提供游戏的介绍、下载和更新。还具有玩家匹配、玩家成绩管理等完善的功能。2010 年继 Windows 版后发布了 Mac 版，可以玩 *Portal* 等主流游戏。 此外，将来还会发布 Linux 版，在发展中国家市场也会开始提供服务。

- Plus+

  官方网站：http://plusplus.com/

  开发者网站：http://plusplus.com/developers

  提供玩家匹配、点数管理、玩家成绩管理、连接社交网络等功能。没有中继服务，相对来说是比较简单的功能。

- OpenFeint

  官方网站：http://www.openfeint.com/

  开发者网站：http://www.openfeint.com/developers

  提供玩家匹配、点数管理、玩家成绩管理、连接社交网络并集成 Apple Game Center 等功能，早已支持 Android 平台。对于开发者来说这是个开放的环境，可以自由尝试 SDK，没有什么政治上的约束。

- GameSpy Arcade[1]

  官方网站：http://www.gamespyarcade.com/

  开发者网站：非公开（咨询网站：http://www.poweredbygamespy.com/）

  历史最久的服务。2000 年左右就提供了玩家匹配、玩家成绩管理等功能。SDK 没有公开，需要联系管理 GameSpy 服务的 IGN 公司并说明商业计划。

- Uplay

  官方网站：http://uplay.uk.ubi.com/

  开发者网站：非公开（需联系育碧公司 Ubisoft）

  没有向第三方公开，是育碧公司内部游戏的专用服务，但是功能十分强大。独特的功能是 Uplay 管理的虚拟货币可以在游戏内购买道具。

> [1] 2013 年 1 月 1 日已停止授权。

面向游戏主机的服务

针对游戏主机的服务，一般只需要联系游戏主机提供商（微软、任天堂、索尼电脑娱乐、苹果等），提出服务申请，说明商业计划并签署保密条款后即可。一般在开发工具包内都会提供标准的平台开发 SDK。

- Microsoft Xbox LIVE

  官方网站：http://www.xbox.com/ja-jp/live/

  开发者网站：非公开

  包括了玩家匹配、玩家成绩管理等各种业界标准的游戏辅助系统。SDK 的文档十分详细。

- 任天堂—Wi-Fi Connection

  官方网站：http://Wi-Fi.nintendo.co.jp/

  开发者网站：非公开

提供玩家匹配、玩家记录等基本服务。功能比 Xbox LIVE 少，因为任天堂的游戏主机主要面向儿童和青少年，一般都不是很复杂的多人游戏。

- PlayStation Network

  玩家匹配、玩家成绩管理、中继服务等，提供和 Xbox LIVE 相当的最完备的辅助系统服务。SDK 的文档虽然可以更为详细，但和 PSP 交互的普通游戏的基本功能已经相当完备。

- App Game Center

  官方网站：http://www.apple.com/game-center/

  开发者网站：http://developer.apple.com/

  提供了许多针对 iPhone/iPad 的功能，包括 P2P MO 游戏使用的通信、玩家匹配、玩家成绩管理、语音聊天等。特点是可以使用 iTunes 并提供了强大的应用内付费功能，具有非常方便的游戏发售渠道。

面向网页游戏的服务

针对网页游戏，比较有名的现有服务如下 [2] 。

> [2] 笔者执笔时，服务的状况还在不断变化，其他比较有名的服务还包括 InstantAction 公司的 Torque：利用 InstantAction 提供的浏览器插件可以在浏览器上实现和原生程序同等品质的游戏，并且包含玩家匹配、程序介绍和发布、点数管理等功能。

- Kongregate

  官方网站：http://www.kongregate.com/

  开发者网站：http://www.kongregate.com/labs

  基于浏览器的 Flash 游戏使用的平台。提供玩家匹配、玩家成绩管理等基本功能。

现有中间件

现有中间件提供了程序库，但需要自己准备服务器并搭建环境。

- Rendez-Vous

官方网站：http://www.quazal.com/

开发者网站：没有公开的开发者社区网站

是世界上市场占有率最高的网络程序库。根据 Quazal 公司官方网站的描述，他们提供的功能包括：好友列表、工会、比赛系统、Ping 统计、聊天室、用户管理、即时聊天系统、邀请、聊天、敏感词过滤、联赛、排行榜、统计、事件日历、账号管理、加密、认证、文件上传、下载，等等。这是功能最全的程序库，堪称网络游戏辅助系统的功能大全。软件授权价格是非公开的，基本以公司为单位，根据不同的商业需求定制了不同的价格。如果是针对游戏主机的开发，因为游戏主机平台已经提供了很多相同的功能，所以还可以选择使用其中的部分功能。

6.1.2 现有服务的功能一览

我们将以上介绍的现有服务的功能整理成了表格，请参考表 6.1。各个公司提供的服务功能在不断提升，所以表 6.1 的内容会不断变化，这里只列举目前的功能。在开发实际的商业化游戏时请做好相关技术调查。

从表 6.1 可以看到不同公司所提供服务的差异，以及网络游戏辅助系统需要什么样的功能。另外现有服务一般不包括表 6.1 中“其他辅助功能”的内容，有些不是针对游戏的特有功能，比如用于节省开发团队或者运营成本的辅助系统，我们会在本章的最后介绍。

表 6.1 现有服务的功能比较

分类												
名称	对象	Steam	Plus+	Open Feint	Game Spy Arcade	Uplay	Xbox LIVE	Wi-Fi 连接	Play Station Network	Apple Game Center	Kong regate	RendezVous
交流/ 通信功能												
玩家匹配	P2P MO	○	○	○	○	○	○	○	○	○	○	○
游戏大厅	P2P MO	○			○		○	○	○			○
中继服务器	P2P MO					○		○			○	
聊天	P2P MO	○	○	○							○	○

	C/S MMO											
邮件	P2P MO C/S MMO	○	○	○	○	○	○		○	○		○
玩家状态	P2P MO C/S MMO	○	○	○	○	○	○	○	○	○	○	○
加锁服务器	P2P MO C/S MMO	○	○	○	○	○	○	○	○			○
好友列表	P2P MO C/S MMO	○	○	○	○	○	○	○	○	○	○	○
黑名单	P2P MO C/S MMO	○	○	○	○	○	○		○	○		○
语音聊天	P2P MO C/S MMO	○					○	○	○	○		○
游戏客户端辅助功能												
玩家成绩管理	P2P MO C/S MMO	○	○	○	○	○	○	○	○	○	○	○
存储功能	P2P MO C/S MMO	○	○	○	○	○	○	○	○	○	○	○
客户端更新	P2P MO C/S MMO	○					○	○	○	○	○	○
排行榜	P2P MO C/S MMO	○	○	○	○	○	○	○	○	○	○	○
运营												
新闻发布	P2P MO C/S MMO	○	○	○	○	○	○		○		○	○

付费相关的辅助系统												
付费验证、账号管理	P2P MO C/S MMO	○	○	○	○	○	○	○	○	○	○	○
虚拟货币管理	P2P MO C/S MMO	○	○	○	○	○	○			○	○	
其他辅助功能（现有服务基本不包含的功能）												
敏感词过滤	P2P MO C/S MMO											
数据浏览、查询工具	P2P MO C/S MMO									○		

6.1.3　网页游戏开发方法和客户端游戏的开发方法

网络游戏使用的辅助系统基本都需要网络通信功能。因为项目预算有限，所以一般都尽可能使用现有的服务，但是游戏主机以外的平台并没有太多可供选择的服务，所以很多时候还是需要自己开发。

自己开发时，需要考虑开发成本、运营成本、必要性能、程序员技能水平等因素，一般采用 Web 开发技术开发，除非某些特定的场合需要用 C/S 架构。

使用 Web 技术开发比较困难，更适合使用 C/S 架构开发的情况如下。

- **需要快速发送消息的。**
- **想沿用游戏使用的 RPC 通信或者共享内存方式的代码。**
- **实现方式是在 C/S MMO 的 gmsv 服务中追加 RPC 函数。**

在接下来的章节中，我们会列举实例，针对 *K Online* 和 *J Multiplayer* 游戏，说明一些比较有代表性的辅助系统。我们按照表 6.1 中的几大类，分为“交流 / 通信功能”、“游戏客户端辅助功能”、“运营”、“付费相关的辅助系统”、“其他辅助功能”五部分进行说明。

6.2 交流 / 通信功能

交流 / 通信功能相关的辅助功能如表 6.1 所示，包括玩家匹配、游戏大厅、中继服务、聊天、邮件、玩家状态、锁服务、好友列表、黑名单以及语音聊天等功能，下面我们按照这个顺序来介绍。

6.2.1 玩家匹配 P2P MO

用途：P2P MO 游戏中查找对手玩家。

问题：无法解决 NAT 问题、很难避免高级玩家遇到低级玩家、需要一些处理时间、不能很好地取消匹配、无法顺利实现中途参加游戏的功能。

实现：三种方式：在现有服务的基础上定制开发、基于 Web 技术简单实现或者 C/S 架构实现。可以采用横向分区 [3] 、常驻内存方式实现。

> [3] Horizontal Partitioning。第 4 章曾提到过，这是一种 DB 的数据分区方式。将一张表里的数据分散存储在不同的表中，每个表保存不同类别的数据。假设有 100 万以上玩家数据的表，可以按照账号开头字母 A 到 K 的数据存在一张表中，剩下的存在其他的表中，或者由不同的 MySQL 服务器管理。

玩家匹配系统是指，在 P2P MO 游戏中，按照一定规则，选择网络上的两名玩家，让他们可以进行通信，或者查询出满足游戏条件的玩家，让他们可以进行多人游戏的系统。第 2 章介绍了这个系统的必要性。

这里所说的需要玩家匹配系统的游戏是指两人以上可以同时游戏的即时多人游戏，不包括“Travian ”、网页三国志等 Web 游戏。

玩家匹配系统遇到的典型问题是“无法解决 NAT 问题”、“很难避免高级玩家遇到低级玩家”、“需要一些处理时间”。一般实现方式包括“在现有服务的基础上定制开发”、“基于 Web 技术实现”、“采用 C/S 方式实现”。此外还有“是否采用水平分区方式”和是否采用常驻内存”的争论。下面我们详细介绍一下这些技术的判断标准。

P2P MO 游戏中多人游戏需要满足的条件

P2P MO 多人游戏需要满足以下条件。

1（必要条件）参加游戏的玩家机器之间可以相互通信。

2（尽可能）可以选择一起游戏的玩家。

在客户端和服务器端都可以实现满足上述条件的功能，不过为了避免重复开发，一般会在“玩家匹配服务器”中实现全部功能。

玩家匹配服务器

为了满足上述两个条件，玩家匹配功能会按照以下方式实现。

- **发送“多人游戏请求”到玩家匹配服务器。**
- **匹配服务器收到请求后，查询“匹配候补列表”，如果找到满足条件的玩家，则向各个客户端返回匹配信息。**
- **如果“匹配候补列表”中没有符合条件的玩家，则将相关信息加入候补列表，并保存变化后的信息。**
- **所有的客户端在取消玩家匹配请求后（关闭客户端程序时）、游戏开始后和游戏结束后需要将相关状态的变化通知匹配服务器。**
- **状态更新后，匹配服务器需要从候补列表中查询出满足条件的玩家。**
- **客户端的通信中断后，匹配服务器需要进行垃圾清理（必要时）。**

采用上述实现方式，匹配服务器的负荷计算方法：想进行多人游戏的玩家数 × 状态变化频率。

满足多人游戏的两个条件的具体做法 ——*J Multiplayer* 游戏

要在 *J Multiplayer* 游戏中实现上述两个条件的具体做法如下。

- **（必要条件）参加游戏的玩家机器之间可以相互通信**

  J Multiplayer 是包含主机的 P2P MO 游戏。P2P MO 游戏不需要配置专用的游戏服务器，而是需要玩家机器之间通过某种方式建立连接。我们按照以下步骤构建通信线路。

  游戏客户端启动时，需要调查客户端使用的是什么网络。具体情况包括：❶ 具有公网 IP，外部主机可以直接连接（Global）。❷ 可以使用 NAT 遍历技术（NAT-OK）。❸ 不能使用 NAT 遍历技术（NAT-NG）。

  如果是 Global 或者 NAT-OK 时，则自动满足条件 1。如果是 NAT-NG 的情况，需要在候补列表中确认是否有 Global 或者 NAT-OK 的玩家，并且没有处在游戏中状态，有则满足条件 1。

- （尽可能）可以选择一起游戏的玩家

  J Multiplayer 游戏的设计是“尽可能让相同等级的玩家，并且是不同职业的 4 个角色一起游戏”。也就是说，如果不满足这个条件就不能获得最好的游戏体验。

  使用各游戏主机平台公司提供的现有匹配服务时，也需要像这样针对游戏策划进行相应的定制化开发，有很多方法可以调整匹配算法，也可以采用专用的模块。

满足多人游戏的两个条件的实现 ——*J Multiplayer* 游戏

为了实现 *J Multiplayer* 的“尽可能让相同等级的玩家，并且是不同职业的 4 个角色一起游戏”的游戏策划，具体该怎么做呢？这里给出的方法，虽然不能算是最好，但还是想和读者一起分享一下实际开发中程序员解决这个问题的做法。

- 在匹配服务器采用计算量不大的算法

  为了不让匹配服务器负载过量，需要采用计算量不大的算法。

  匹配服务器的负荷是由“想进行多人游戏的玩家数 × 状态变化频率”来决定的。不管玩家数量如何增加，对于一个人来说，即使玩家增多，他的状态变化频率也是一定的，所以简单的做法是以玩家数量的一次方作为负荷。

  如果是每次变化后需要查询全部的匹配候补列表的方法，则负荷为玩家数量的二次方。为了避免因为涌入玩家超过预期，导致游戏负荷过高无法进行而失去商机的情况，一般会要求比实际需要更高的性能，但具体应该高到什么程度这也是一个问题。如果是玩家数量的二次方或者三次方，即便玩家稍微多一点，性能也会达不到要求。

  采用简单的做法，计算量是 N 的二次方或者三次方时 [4] ，稍微优化就可以得到 logN 或者 NlogN 左右的结果。

  游戏主机等提供的现有玩家匹配服务一般登录了玩家的 2～3 个整数类型的属性，制定了匹配条件后，就可以得到满足条件的结果，非常方便。而且微软等公司还有 TrueSkill 这样的统计分析程序库，可以“根据游戏的胜负次数或者趋势查找有趣的对手”，正如他们宣传的那样。而这里的理论基础是不能仅仅通过胜负次数和胜率来判断玩家的强弱。

自己研发时，也需要实现同样的算法，到时再根据游戏的策划内容开发必要功能即可。

- **开发时需要注意的两点**

  开发时需要注意的两点是“无需持久化”和“不易扩展性能”。

  - **无需持久化**

    查找匹配的玩家只是暂时的请求，所以没有必要持久化相关数据。虽然不需要使用 MySQL 这样的关系型数据库（RDBMS），但是 SQL 比较方便，可以考虑采用常驻内存型的表来实现。也可以使用一些轻量级语言，或者 Java、C++ 在内存中进行处理。这两种方法都没有问题。

  - **不易扩展性能**

    玩家多的时候处理也会变慢，所以需要考虑匹配服务器的分区。如果想得到理想的结果，就得尽可能在更多的匹配请求基础上进行查询。但是简单的分区会造成匹配请求被分散，让结果变得不理想。为了避免这样的问题，需要谨慎考虑分区的方式。例如，4 个人玩的游戏，性能上需要单服处理 1 万人左右。如果是每 1000 人分区的设计，可能对匹配结果会有非常大的影响。不过对于将棋这样 1 对 1 的游戏来说，并不需要那么多人。查询条件每增加一个，人数每增加一人，查询需要的请求数也会成几何数字式增长。所以，匹配服务器的性能测试要尽早开始。

> [4] 严格来讲，玩家匹配这个例子因为使用的内存比较大，缓存命中率比较低，所以会比二次方稍微多一点，大概是 2.2 次方左右。

匹配结果画面

“匹配成功”后的结果必须包括以下信息。

- **主机的 IP 地址和端口号，以及 NAT 类型。**
- **玩家的信息（名字、级别等）。**

有了这些信息就可以显示“匹配结果画面”，在玩家之间建立网络连接并且开始通信。NAT 相关的问题在 5.6 节已有说明，在此不再赘述。

6.2.2 游戏大厅 P2P MO

用途：在 P2P MO 游戏中查找对手玩家。

问题：没有特别难的问题，不过想要开发容易变更、成本低的服务器需要更多时间。

实现：利用现有服务或者 C/S 方式实现两种选择。容易进行横向分区。

游戏大厅是匹配服务器的一种形式。匹配服务器是自动为玩家查询游戏对手，而游戏大厅是由玩家自己从列表中选择游戏对手。

游戏大厅和匹配

有些游戏可以同时使用游戏大厅和匹配服务两种方式。这时通常会把匹配服务称作“快速匹配”。从名称上来区别，自动帮玩家找对手的叫做“快速匹配”，玩家自己选对手的叫做“游戏大厅”。

使用游戏大厅得到的结果包括以下信息。

- **主机的 IP 地址和端口号，以及 NAT 类型。**
- **玩家的信息（名字、级别等）。**

和匹配服务的结果包含的信息相同。

“游戏大厅”是从酒店的“大厅”（Lobby）一词引申而来，意为“等待的地方”。游戏开始前需要等待，查询游戏对手，找到合适的对手后才开始游戏。和匹配服务不同，除了 P2P MO 游戏，C/S MMO 游戏在选择服务器的时候也经常使用游戏大厅。

《星际争霸 2》（*StarCraft II* ）的例子

以《星际争霸 2》为例，游戏中的“战网”（Battle.net）就相当于游戏大厅和匹配服务（图 6.1）。战网是由暴雪娱乐公司提供的专门用于暴雪娱乐公司出品游戏的对战平台。这个服务发布的时间很早，功能十分完善，是游戏大厅的行业标准，影响力很大。*J Multiplayer* 的游戏大厅基本上是参考战网的功能开发的。

图 6.1　《星际争霸 2》的战网游戏大厅画面

Image used by permission. ©2011 Blizzard Entertainment, Inc.

※ 官方网站：http://us.battle.net/sc2/

《星际争霸 2》的游戏大厅画面可以显示由玩家创建的“自定义游戏”列表，列表中还可以列出没有 NAT 问题的玩家。

自定义游戏是指玩家创建的，自己设定游戏玩法和地图的游戏。可以在游戏大厅选择其他玩家创建好的自定义游戏，建立连接后开始游戏。

《星际争霸 2》的自定义游戏可以设定地图种类、胜负方式、奖励方式、参加条件等，因为条件非常多，自动匹配很难得到理想结果，所以采用了手动选择的方式。在游戏大厅画面的列表中选择自定义游戏后，可以显示更详细的信息。

开发要点

游戏大厅很像实时变化的拍卖场一样，开发时需要注意以下要点。

- 玩家创建（自定义）游戏后，将相关信息发送到游戏大厅服务器（只有能做主机的玩家需要发送 NAT 状态）。
- 游戏大厅服务器收到“现在游戏的列表”请求后返回相关信息。
- 当玩家加入游戏或者其他操作导致游戏状态发生变化时，将该状态发送到游戏大厅服务器。

- **游戏客户端定期向服务器发送请求更新游戏列表。**

对玩家来说，即使游戏大厅按游戏条件在 DB 上进行横向分区，也不会影响其查询结果，所以不需要在这上面考虑过多。负荷过高时分区即可。具体的实现可以利用现有服务、Web 技术或者 C/S 架构。

6.2.3 中继服务器 P2P MO

用途：解决有 NAT 问题的 P2P MO 游戏玩家通信问题。

问题：吞吐量（Throughput）小、带宽要求高、响应时间长。

实现：利用 TCP C/S（TCP 客户端服务器）方式实现。服务器操作系统的选择，分组方式，保障安全性的方法。

对于 P2P MO 游戏来说，就算采用了 NAT 遍历技术也还是会有一部分玩家无法通信。而利用中继服务器就可以解决玩家的通信问题 [5] 。当 NAT 连接无法使用时，中继服务器可以作为玩家之间通信的媒介。

> [5] 中继服务器在第 3 章和第 5 章都介绍过。

中继服务器的作用就是让尽可能多的玩家享受到多人对战游戏的乐趣。中继服务器使用 TCP 的大端口号，尽可能避免通信过程中路由器、防火墙和 NAT 设备等的数据包限制。如果服务器资源和带宽允许，还可以使用占用 80 号端口的 HTTP 通道来建立连接。

中继服务器需要的性能

中继服务器需要尽量避免网络延迟，让通信可以快速进行。由中继服务器造成的网络延迟大致估算如下。

首先假设互联网的平均跳转数是 20，20 次跳转的通信延迟大概是 30 毫秒，玩家之间直接连接时，发送数据包的延迟是 30 毫秒。

客户端→网络（20 次跳转）→主机

经过中继服务器时的过程如下。

客户端→互联网（20 次跳转）→中继服务器→互联网（20 次跳转）→主机

总计 40 次跳转，延迟 60 毫秒，其中可以看到中继服务器增加的延迟。P2P MO 游戏要求游戏延迟在 50 毫秒以内（参考第 2 章），这里由于中继服务器的原因让整体延迟超过了 50 毫秒，无法实现预期的游戏内容。因为有游戏内容上的限制，所以中继服务器自身的延迟需要在 10 毫秒以下，甚至要尽可能缩短到 1 毫秒左右。

网络编程基本部分在第 0 章已经说明，中继服务器的 CPU 单核需要处理几百至几千的 TCP 连接，如果不降低计算成本就无法商业化。这种情况下，1 个 TCP 连接是 1 个线程，总共几百至几千个线程的性能开销太大，不符合要求，所以决定采用非同步套接字轮询的策略 [6] 。

> [6] 非同步 I/O 策略在 0.2 节中有详细说明，请参考相关章节。

例如，1000 个连接每秒发送 10 次消息时，CPU 单核会收到 1 万个以上的 TCP 数据包。使用 C/C++ 语言和操作系统的事件轮询功能可以实现这点，延迟在 1～ 2 毫秒左右。但是如果使用云计算（Amazon EC2 或者 Nifty Cloud 等）服务，由于硬件或者网络的原因可能会产生很大的延迟，所以应该提前验证再决定是否使用这些服务。

另外，如果预算或者运营体制允许，应该尽可能把中继服务器配置在距离提供游戏服务区域比较近的地区。索尼电脑娱乐等公司在世界 5 个以上的地区配置了中继服务器，从这个角度来说，这些大公司提供的服务是非常便利的。

6.2.4 聊天 P2P MO C/S MMO

用途：通用。

问题：吞吐量（Throughput）小。

实现：利用 Web 技术或者 TCP C/S 技术。性能扩展方法，不同国家访问方式的不同。Spike 问题的对应。

聊天是一项基本功能，P2P MO 游戏和 C/S MMO 游戏都可以使用该功能。不管哪种游戏，聊天需要的通信功能对于服务器来说都是一样的，所以会在游戏通信开发时顺便实现聊天功能。P2P MO 游戏中的实现方式是由客户端向主机发送消息，主机再向其他客户端发送消息。C/S MMO 游戏是客户端向游戏服务器发送消息，然后由游戏服务器向全体客户端发送消息。

简单的聊天功能可以按这种方式开发，没有特别复杂的地方。

聊天功能的实现

如果 P2P MO 游戏玩家之间的通信超过了网络限制范围，或者 C/S MMO 游戏中的通信跨越了游戏服务器就会出现一些问题。

出现这种情况时请参考 IRC，设置专用的聊天服务器，并实现服务器之间的通信处理。笔者也开发过这样的聊天系统，仅仅使用开源的 IRC 实现并不能满足服务器端的功能需求，所以基本上项目中都是自己开发。

虽然是自己开发，但是只要注意一些要点，处理逻辑比起实现游戏服务器来说还是简单很多。

聊天功能的基本处理

聊天功能的基本处理整理如下。

- **cli（游戏客户端）向聊天服务器发送消息，并同时向需要的客户端发送。**
- **需要在 1～2 秒内将消息送达需要的客户端（Push）。**
- **和邮件不同，不需要给不在线的玩家发消息，所以不用保存信息。**
- **服务器端不需要保存用户的日志。**

一般客户端和服务器之间的 TCP 会话（session）会一直保持，Push 时会利用这个会话发送消息。也可以使用 HTTP 轮询或者 Comet[7] 的方式，不过会增加服务器的负荷。

> [7] 在 Web 应用中，服务器将后台发生的变化实时发送到客户端而不需要客户端不停的刷新、发送请求。

因为在单台服务器上无法为几万用户同时提供服务，所以当游戏用户量超过一定规模时需要多服务器处理。

同步发送消息的规模

接下来，针对“应该对哪些用户同步发送消息”这一点，我们分成几类来讨论。对于网络游戏，根据游戏策划内容的不同，有如下一些典型的形式。

- **1 对 1：也叫做 Tell、Whisper、Private 或者 Direct 等。某个玩家和某个玩家之间的通信。**

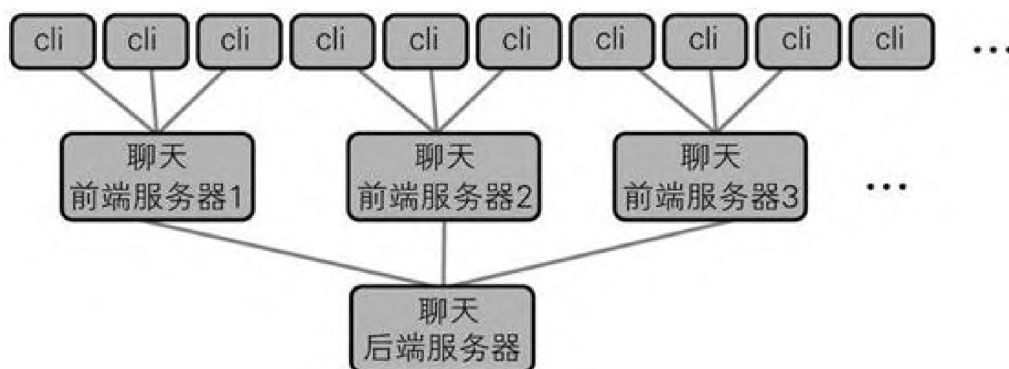
- 小规模：也叫做 Party，在 5～6 人的临时小组内发送消息。
- 中等规模：也叫做 Clan、Guild 或者 Group 等。对 SNS 的组群级别的全部用户发送消息。
- 大规模：也叫做服务器消息，对某个服务器的全体用户发送消息。最多可以同时发送给 1000 人以上，根据传送距离又分为 Shout、Scream 等。
- 最大规模：也叫做世界消息，针对游戏的全体玩家发送。一般是游戏管理员发出的通知。比如在游戏策划中，某个城池被攻陷时发出的通知，为了系统的特殊目的而使用的消息。同时对几万玩家发送，这是负荷最高的聊天消息。

为了满足大量客户端的响应需求，需要多台服务器分别处理。其中最重要的问题是“哪些客户端连接哪些服务器比较好”。

比如游戏策划中需要 Tell、Party 和 Guild 三种消息类型，其中大部分是 Party 类，对于这种情况，尽可能让小团队内部的玩家都连接相同的服务器，这样可以减少服务器之间的通信。

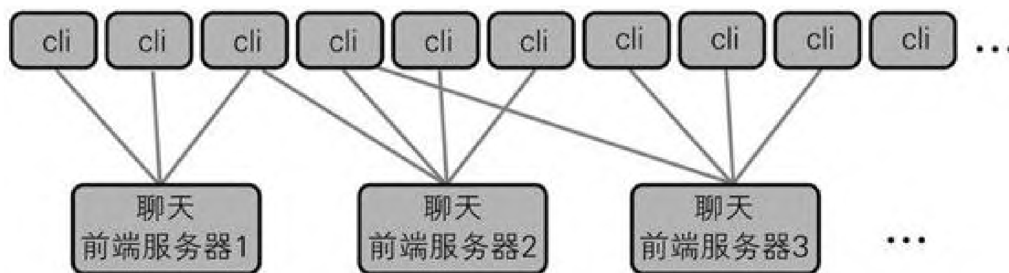
比如图 6.2 中的结构，前端服务器之间的通信经过后端服务器，连接聊天服务器时，会根据团队成员连接的是哪台服务器来做相应的选择，只有必要的时候才会连接后端服务器。

图 6.2　满足 Tell、Party、Guild3 种消息的服务器结构



还有一种情况，游戏策划中只有 Tell 和 Party 消息时，服务器之间就没有通信的需要。如图 6.3 所示，这种结构的服务器之间没有通信，如果想和不在团队内的玩家发送私信，可以在自身团队所在的服务器和对方所在的服务器中建立两个会话。当会话数上限最低为 2 时，就可以实现这种需求。

图 6.3　只有 Tell、Party 消息时的服务器结构



这种情况不需要后端服务器，管理起来也比较容易。以上两种是比较典型的服务器结构，基本可以满足各种游戏的需求。

6.2.5　邮件 P2P MO　C/S MMO

用途：通用。

问题：没有特别难的问题，但要注意控制成本。已读邮件管理、逻辑删除、服务提供商责任限制法 [8] **。**

> [8] 正式名称是《特定电信服务提供者损害赔偿责任限制及发信者信息揭示法》。在互联网上，隐私或者著作权等权益受到侵害时，规定了服务提供者（网络提供商或者网站管理者）应该承担的损害赔偿责任范围，以及受害者可以要求公开信息的权利。

实现：利用现有服务、Web 技术或者 C/S 技术（和 Web 应用开发相同，横向分区比较容易）实现。

邮件服务和聊天服务有以下不同点。

- **消息比较长。**
- **对于不在线的用户也需要发送邮件（需要持久化）。**
- **需要记录日志。**
- **允许邮件延迟，几秒至 20 秒左右都可以接受。**
- **需要管理已读邮件。**
- **可能需要提供查询服务。**

因为可以有一定延迟，所以一般通过采用 HTTP 协议的 Web 服务（Web Service）来实现。此外，利用 Web 服务实现的邮件系统和 C/S MMO 的游

戏服务器之间也可以进行通信。编码、测试和发布一体化非常方便，所以经常使用这种方式。

邮件系统可以很容易地按照用户 ID 进行横向分区，没有特别需要注意的地方[9]，因此本书不再深入讨论。

> [9] 和 E-mail 相互关联时，情况会比较复杂。

6.2.6　好友列表 P2P MO　C/S MMO

用途：通用。

问题：有单向和双向两种选择，双向列表会有不一致的情况。

实现：列表不会很大，没有特别困难的地方。利用 Web 技术、C/S 技术都可以实现，也可以使用现有服务。

即朋友列表。自己在 Twitter 等平台上关注的人称为好友。

和邮件相同，好友列表也可以按照用户 ID 进行横向分区，不会出现负荷过高的情况。访问频率的变化也不大，可以使用 HTTP 协议实现。现有的服务也都比较成熟，可以很容易集成到游戏中。

6.2.7　玩家状态 P2P MO　C/S MMO

用途：共享 P2P MO 或者 C/S MMO 游戏中好友的状态。

问题：吞吐量不大。

实现：可以使用现有服务，自己开发的话因为需要频繁通信所以建议采用 C/S 结构；不需要性能扩展方面的考虑。

玩家状态是指实时通知好友是否在线的服务。以 Skype 的联系列表为例，好友状态分为在线、离线和自动回复等。网络游戏中，为了查询同时游戏的玩家，需要知道好友列表中的玩家目前是在线还是离线状态，对于很多游戏来说这个信息十分重要。

游戏中的玩家状态服务的特点

和 Skype 等软件不同，游戏的玩家状态功能需要显示的信息很多。比如某个 C/S MMO 游戏，玩家被敌人攻击时 HP 的变化也需要通知好友。所以，

根据游戏策划的不同，玩家状态服务器的负荷有可能比较高。

玩家状态服务器和聊天服务器中“工会”（Guild）消息的通信量差不多，不过有以下一些区别。

- **特征 1：工会成员列表对于所有工会内的成员都是一样的，但是好友列表不同，每人的好友是不一样的，所以不能像工会那样可以指定专用服务器。**
- **特征 2：和聊天不一样，比如 HP 的变化这样的消息，中间丢失几条也不要紧。**
- **特征 3：消息传输不用像聊天消息那么快，允许延迟 10～30 秒左右。**

玩家状态服务的实现

参考上面提到的这些特征，基本采用和聊天服务器的实现相同的结构（参考图 6.2），在实现上需要花一些功夫。

首先需要估算通信量，消息可以分为以下三种。

❶ 绝对不能省略的消息

登录、退出的通知。

❷ 尽量不要省略的消息

现在所在服务器、所在位置。

❸ 可以省略的消息

HP 的变化、击倒的敌人的名字等。

不管什么游戏，登录和退出都不会很频繁地发生。就拿笔者参与过的 MMO 游戏项目来说，时间短的平均是每 40 分钟一次。假设 100 万玩家同时在线（非常高的设定，很少见），平均 2000 秒 1 次的话，1 秒也只有 500 次访问。这种规模的用户，前端服务器应该有 100 台左右。

对于特征 3“可以允许一些延迟”，需要做以下处理。

- **后端服务器收到状态变化通知后记录所有信息**

- **前端服务器每 10 秒从后端服务器更新一次状态变化信息**
- **前端服务器将更新后的状态通知各个客户端**

关键点是不用全部实时通知。后端服务器需要进行以下处理。

- **保存状态变化：每秒接收 500 条消息，向前端服务器分配**
- **读取状态变化并通知：每秒响应 10 台前端服务器的请求**

以使用 MySQL 的后端服务器为例，在单个服务器实例（instance）中，每秒 500 次简单的 update 操作，需要进行 10 次 1 万行数据的 select 查询。表不需要持久化，所以可以在内存上操作。

这种情况下不仅可以满足上述“❶ 绝对不能省略的消息”的性能要求，还有一些富余。

“❷ 尽量不要省略的消息”可以使用剩余的性能部分。剩余的性能部分具体是多少呢？不如我们来测试一下 MySQL 的性能，看看吞吐量是多少，这样效率更高一些。假设 update 操作可以达到每秒 3000 次左右，因为已经使用了 500 次，所以还剩 2500 次可以使用。通过这种方式可以确定设计的细节。

6.2.8　加锁服务器 P2P MO　C/S MMO

用途：防止 C/S MMO 游戏中双重登录、数据损坏或者无限增长的情况。

问题：锁定状态残留、解锁攻击的漏洞。

实现：C/S、处理序列原子化（atomic）。

C/S MMO 游戏有多个游戏服务器，如果同时登录多个服务器，最后退出的玩家数据会覆盖之前的数据，导致玩家状态回档。如果被恶意利用还会发生无限道具的问题。所以需要防止 C/S MMO 游戏的双重登录，加锁服务器可以实现这个功能。

加锁服务器的实现

加锁服务器的功能一般会在开发玩家状态服务器时一起实现，这里将针对功能本身进行说明。

首先，我们来看一下采用 MySQL 实现加锁服务器的方式。

- 玩家登录的时候，首先在加锁信息表中使用 select 操作查询目前的状态。
- 如果已经登录则拒绝再次登录。
- 如果是退出状态则允许登录，并使用 update 操作更新加锁信息表，设为登录状态。

这里有个问题，如果没有实现“游戏服务器发生异常退出时，解除该服务器的所有锁定状态”的功能，就会发生锁定状态残留的问题。

所以使用 MySQL 实现是比较麻烦的，一般会“实现专用的常驻内存处理方式的服务器”。例如在第 4 章介绍 *K Online* 的章节中，防止双重登录的功能是在管理玩家状态的消息服务器（msgsv）中实现的。锁定状态不需要持久化，所以不会有性能问题（负荷比玩家状态管理要小很多）。

6.2.9 黑名单 P2P MO C/S MMO

用途：通用。

问题：降低处理负荷、在客户端还是服务器端进行限制、黑名单被其他玩家看见会很麻烦。

实现：没有可以使用的现有服务，可以采用 Web、C/S 架构。名单不大所以比较简单。

黑名单是指在聊天或者邮件等与其他玩家通信的功能中，用于屏蔽特定玩家的通信的功能，也叫做“限制名单”。

最近，游戏中发送垃圾消息的玩家多了起来，所以很多游戏需要黑名单功能。这个功能可以拒绝黑名单里的用户发出的邮件，不显示他们发出的聊天信息。

黑名单功能的实现

游戏中的黑名单功能需要考虑下面两个问题从而决定实现方式。

- 黑名单的信息是在客户端还是在服务器端保存。
- 黑名单的判定是在客户端还是在服务器端进行。

Web 应用的所有处理都是在服务器端进行。而对于游戏来说，主数据（master data）可以在客户端保存，登录时再发送给服务器。

“黑名单的判定在客户端进行”是指服务器将全部消息发给客户端，由客户端来决定是否显示。

黑名单的信息也包含隐私信息，所以需要结合游戏策划考虑采用什么方式。如果黑名单功能的实现是在服务器端，比如与玩家状态、聊天、邮件等功能结合时，这种情况下黑名单的判定处理可能会占大部分处理负荷，在设计时需要十分小心。

6.2.10　语音聊天 P2P MO　C/S MMO

用途：通用。

问题：占用带宽太多、NAT 问题。

实现：利用现有服务器或者使用中继服务器自己开发，使用编码解码器。

语音聊天技术非常复杂，详细说明的话可以写一本书。对于主机游戏或者 PC 游戏来说，利用现有服务会比较方便快捷，比如 BOSE 公司的商业程序库或者各个平台提供的工具。

如果要自己开发，需要注意以下几点（笔者也没有实际的开发经验，也不好多说……）。

最大的问题是占用带宽太多。一个用户发言时需要 10kbit/s 左右，以此为平均标准，向 5 个人发消息时，服务器就需要占用 50kbit/s。服务器带宽不足时，虽然可以像 P2P MO 游戏那样转换成玩家之间直接连接的方式，但这样又会遇到 NAT 的问题。

不过不发言时是不占用带宽的，实际上游戏中同时发言的玩家人数并不是很多（10 个人一起发言的情况应该没有，一般都是一个人说话其他人听），所以可以利用这个特点进行相关设计。

＊＊＊

以上就是交流 / 通信辅助功能的介绍。

6.3　游戏客户端实现相关的辅助系统

接下来，我们来介绍一下游戏客户端实现相关的辅助系统。主要内容如表 6.1 所示，包括玩家成绩管理、存储功能、更新功能和排行榜。

6.3.1　玩家成绩管理 P2P MO　C/S MMO

用途：通用。

问题：如何防止作弊、怎样在之后追加游戏成绩。

实现：可以使用现有服务。如果自己开发，需要在客户端处理还是上传全部游戏数据在服务器端处理，自己开发也不是很麻烦。

玩家成绩管理是指保存 / 浏览 / 查询 / 共享游戏中获得的成就。一般的游戏策划中玩家可以获得的成就能达到 10～100 种，例如“超过了 100 万分”、“击败了最终 Boss”和“清除了全部敌人”等，如果在游戏过程中获得了这些成就则显示在列表中，其他玩家也可以看到。

玩家成绩管理的实现

玩家成绩管理一般在游戏服务器或者游戏程序内部实现，因为是通用功能，所以 Xbox LIVE 平台集成了该功能。现在还可以使用其他服务。利用这些现有功能，可以在 玩家成绩管理的实现上节省很多时间。

在游戏机或者 iPhone 等手机上开发游戏时，可以在客户端程序中，利用一些公司提供的玩家成绩管理服务的开发程序库，当游戏内满足条件时调用相应的函数，显示成绩画面，并将信息发送到玩家成绩管理服务器。

使用这种方式记录玩家成绩的 P2P MO 游戏，很容易通过对游戏客户端程序的破解来作弊。所以并不适合那种根据游戏成绩来获取稀有物品的游戏策划方案。

C/S MMO 游戏可以在服务器端实现玩家成绩管理，目前大部分的游戏都是自己开发该功能。只要对程序库稍微修改一下就可以调用服务器记录玩家的成绩，所以那些轻量级的 C/S MMO 游戏可以直接利用这些程序库。

自己开发的话，因为记录成绩的频率并不是很高，可以很容易地使用 HTTP 协议实现。分区方式和邮件系统的实现基本相同，可以根据用户 ID 进行横向分区。

6.3.2　存储功能 P2P MO

用途：在服务器而不是存储卡内保存 P2P MO 游戏的游戏进度。

问题：作弊、DB 碎片化。

实现：利用现有服务，可以采用 Web、C/S 架构。不是很困难，横向分区简单。

在服务器端保存 P2P MO 游戏的游戏进度的功能。为了中途继续游戏，需要在某个地方保存游戏的进度。可以保存在游戏机的主机或者手机的存储卡中。

- 在基于网页的 Flash 游戏中，想从多台电脑启动并获取相同进度。
- 在朋友家的机器上读取自己的进度继续游戏。
- 为了防止进度丢失。
- 获得更多存储空间。
- 想让游戏数据和网站关联。

为了满足上面这些需求，需要在服务器端保存游戏数据。很多公司提供了这种功能的服务。如果是自己开发，也可以像邮件系统那样通过用户 ID 进行横向分区。

6.3.3 （游戏客户端）更新 P2P MO C/S MMO

用途：更新需要安装的游戏。

问题：带宽 [10] 、文件一致性、更新补丁。

> [10] Spike 问题（Spike，流量急剧上升或者下降）。

实现：现有服务、利用 BitTorrent 等工具实现、需要策划。CDN（Contents Delivery Network）。

为了延长游戏的寿命，需要发布更新补丁、升级服务器。所以要求定期进行低成本地维护游戏。

不管是 P2P MO 游戏还是 C/S MMO 游戏都需要更新游戏客户端程序。目前主流的做法是直接利用现有服务（苹果、微软、Steam 等公司提供的平台）。按照不同平台的打包方式压缩并上传，通过审查后就可以发布。

当然，该功能只是针对可以在客户端安装程序的平台。

更新的基本功能

如果自己开发更新模块，需要哪些功能呢？

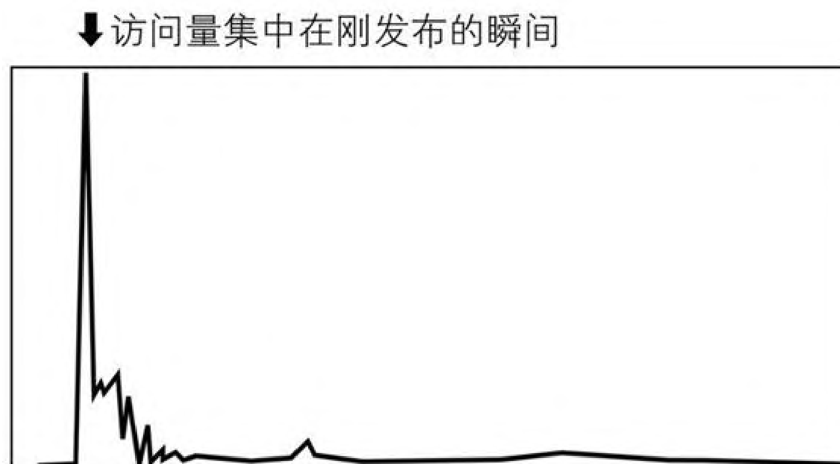
- 在游戏客户端启动时，访问更新服务器，确认是否有新的安装包需要更新。
- 如果有新的安装包则开始下载。
- 下载安装包。
- 下载完成后进行内容的校验。
- 如果没有问题就更新已有游戏（一般不备份）。
- 重新启动游戏。

客户端的实现不是很难，仅有的问题是服务器的带宽。游戏中使用了大量的图片和声音，所以一般更新包会比较大。假设需要给 1 万用户发布 10 兆字节的更新包，则需要 100 吉字节的通信量，即使是使用全速 100Mbit/s 的专线也需要 8000 秒（两个小时以上）。现在很多高画质的游戏需要几百兆字节，甚至几吉字节大小的更新包。

更新和访问模式

程序的更新如图 6.4 所示，有两种模式。

图 6.4　访问模式 1

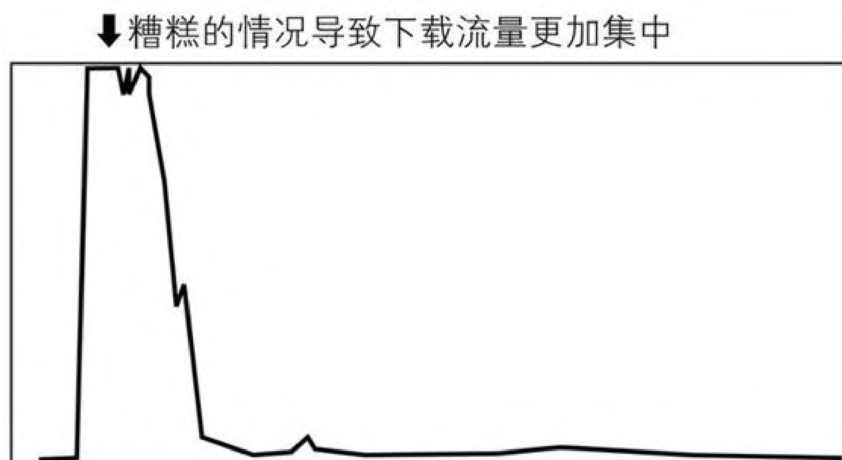


访问集中在发布的最开始，这时如果出现下面的情况会更加麻烦。

- 旧版本的客户端无法连接最新版的服务器。
- 等待更新的玩家很多。

这些情况都出现时，下载流量如图 6.5 所示。

图 6.5　访问模式 2



图的左边表示“几乎没有可以使用的带宽了”，这种状况下玩家只能处于等待状态，所以必须避免这种情况。特别是对于“旧版本无法登录”的设计，“无法正常游戏”是最严重的 Bug。

自己开发更新功能

现有服务都有高性能的基础设施，所以基本不需要担心上述问题。但是对于下列情况，还是需要通过某种方式自己开发该功能。

- 使用 Flash 的游戏。
- 在平台的审查前或者规避审查的游戏。

笔者参与过的项目采用了下面的方法。

- 在游戏的官方网站只发布文件尺寸比较小的安装程序，图像等数据可以和 Akamai 等 CDN 公司合作，分散下载负荷。
- 利用 BitTorrent 程序库分散下载负荷。

- **利用 Nifty Cloud 等公开云计算服务，临时增加 Web 服务器，发布静态文件。**

BitTorrent 方法是上面 3 种方法中成本最低的，但是在实际分散负荷之前，文件的传播时间比较长，需要花费数小时甚至数日，短时间内的负荷分散率比较低，所以只有在那种推迟更新也不影响正常游戏（没有登录限制）的游戏策划中才能考虑这个方法。

6.3.4 排行榜 P2P MO C/S MMO

用途：通用。

问题：服务器负荷非常高、使用 SQL 实现“显示距玩家排名前后 5 位的玩家”比较麻烦。

实现：现有服务比较方便，可以采用 Web 或者 C/S 架构。批处理、性能扩容比较困难。

很多游戏为了增加趣味性，会根据玩家的得分进行排名。这里说的排名是指第 1 名、第 2 名这样的数字排名，而不是 A 级、B 级这种分级。根据某种条件对玩家分级的功能在“玩家成绩管理”功能部分实现。

排名功能的实现 ——网络游戏特有的需求

使用现有平台的排名服务比较简单。

自己开发时，可以使用 HTTP 方式访问，需要注意的是 DB 的设计方法。一般采用 Web 服务的方式实现排名功能，对于游戏来说有以下特征。

- **排行榜的名单很长，有的射击游戏的排行榜中的玩家在 10 万人以上。**

- **玩家想知道自己的排名。**

总的说来就是玩家想知道“我在 12 万 3410 人中排名第 896 位”这样的信息。在一般的 Web 应用中没有这个功能，Web 应用主要是按时间顺序，或者某一列值的大小排序，还有利用索引查询靠前的数据，或者排序后分页，等等。可以组合使用 select 查询、offset、limit、count 等实现这些功能。

但是仅靠这些语句还不能得到“我在 12 万 3410 人中排名第 896 位”的信息。

最简单的排行榜的表（SCORES）应该包含以下信息。

- playerd（整数）
- point（整数）

为这两列都添加索引。

```
> select * from scores order by point limit 10;
```

上面的语句可以查询出排名前 10 的玩家。

```
> select * from scores order by point limit 10 offset 10;
```

上面的语句可以查询出排名第 11～20 位的玩家。

这些查询使用了索引所以比较快，“显示排在自己前后 5 位的玩家”这样的查询一般会复杂一些。因为要知道自己的排名，需要对全部的数据做排序。

增加“rank”列，在每次更新表的时候对所有数据进行排序，然后更新 rank 列的值，这一操作的负荷太高了。

另一方面，如果每次查询排行情况时都对所有数据排序，负荷同样会比较高。

为了解决这个问题，常用方法是。

- 使用 DBMS 时：增加“rank”列，排名前 100 的数据每次都重新排序并更新实时的结果，其他数据可以暂时允许有不正确的结果，所有数据的排序和更新每天做 1 次，一般的查询就返回不正确的排名结果，这个做法叫做“临时排名法”。
- 使用 C++ 或者 Java 这样的高级语言在内存中处理，每次都进行排序，但是不能处理大于 100 万行的数据。

对于大规模的用户，推荐使用第一种方法。

临时排名法

临时排名法是指，找出“比自己分数低的玩家中，分数最高的玩家的排名”，然后加 1。

```
> select * from scores where point <= 新分数order by point desc limit 1
```

尽管不正确，但是玩家也应该不会有意见。

* * *

以上就是与游戏客户端实现相关的辅助系统的介绍。

6.4　运营辅助系统

接下来介绍帮助网络游戏运营的辅助系统。主要内容是“新闻发布”系统。

新闻发布 P2P MO　C/S MMO

用途：消息通知。

问题：限定时间、自动更新、已读管理。

实现：如果有邮件功能的话可以附加在其中，如果没有就使用 Web 技术实现。比较简单。

新闻发布是指游戏运营团队发送通知的方式，尽可能向所有玩家发布同样消息的系统。

新闻的内容可以是服务器的故障报告或者游戏更新通知、活动通知，等等。虽然这些信息可以在游戏的官方网站的首页、开发博客、Twitter 等发布，但是这些途径只能将消息传递给不到 10% 的玩家，而且时效性也不高。

新闻发布的机制

这里需要特殊的机制，采用下面的方法可以在短时间内提高到达率。

- 客户端程序更新时。

如果设计要求每次程序启动时访问 Web 服务器确认是否有最新版，可以在这个过程中同时发送最新的消息，然后在更新时的画面或者更新处理画面中显示该消息。

- **客户端程序启动时。**

  在程序启动时访问 Web 服务器获取最新的消息，并将游戏开始按钮设为不可用，直到读完新的消息，以此确保消息确实被玩家阅读过。

- **游戏中。**

  游戏中玩家操作时，肯定会注视着游戏画面。在适当的时机显示消息，例如，打开邮件时，或者在商店买东西时。

对玩家来说，重复看同样的消息是非常令人讨厌的，所以需要实现已读管理功能，避免同样的消息对同一个玩家多次显示。这样一来，即使消息比较多时也可以大大减少玩家的压力。如果想实现这个功能，Web 服务器就不能只返回静态的信息，需要采用邮件系统一样的设计。[11]

> [11] 不过因为预算的原因，没有实现已读管理的游戏也很多。

6.5 付费相关的辅助系统

本节介绍付费相关的辅助系统，主要内容包括“付费认证”和“虚拟货币管理”。

6.5.1 付费认证 P2P MO C/S MMO

用途：通用。

问题：不保存用户个人信息、外部系统的延迟、多重化、库封装。

实现：除了现有的外部系统的 ID，其他信息都不保存。

本书是针对商业化的网络游戏，所以付费认证的机制是必要的。如果这个系统经常延迟或者不稳定，需要处理包括退款在内的大量客服工作，在商业上这是十分棘手的问题。所以必须采用结构简单、高性能的方式实现。

什么是“付费认证”呢？简单来讲就是回答“玩家 A 可以玩这个游戏吗？Yes/No”这个问题。正确实现回答这个问题的函数后，当玩家开始游戏时，

调用该函数，如果返回值是 No，则显示“请付费”这样的消息。

这个系统貌似比较简单，但是现在网络游戏的商业模式越来越多样化，为了满足更多玩家的需求，需要考虑的细节非常多。

网络游戏的付费

具体来讲，网络游戏的付费方式包括下列一些组合（参考 6.5 节末专栏）。

- 包月费。
- 小额付费（付费道具）。
- 虚拟货币付费。

不管哪种方式都需要使用其他公司提供的充值服务。在 Google 上搜索“网络游戏充值”可以查到很多公司。

付费的机制

提供充值服务的公司需要正确记录玩家购买的这些注册好的“商品”。这些虚拟的“商品”包含以下信息。

- 商品名（例：100 字以内的文字描述）。
- 商品 ID（例：16 字固定文字描述）。
- 价格（例：日元或者美元）。
- 图像等其他信息。

首先，网络游戏为了收费，需要告诉提供充值服务的公司这些“商品”信息。以包月费、小额付费、虚拟货币付费为例，参考下面的详细信息。

- 包月费
    - 商品名“：2010 年 7 月的游戏包月卡”。
    - 价格：480 日元。
    - 商品 ID：KONLINE00012345。

- **小额付费**
  - 商品名：“传说中超强的剑”。
  - 价格：150 日元。
  - 商品 ID：KONLINE00011223。
- **虚拟货币付费**
  - 商品名：“K Online 100 分”。
  - 价格：1000 日元。
  - 商品 ID：KONLINE00000100。

用户使用实际日元购买时，有很多支付方式可以选择，如提供充值服务的公司网站、便利店、信用卡、电信运营商，等等。实际的支付方式根据不同的充值服务会有所不同，调用统一的 API 即可。

提供充值服务的公司网站为了和游戏网站的风格尽量相同，也会使用很多游戏主题的 HTML 网页和图像素材。

支付的序列图

提供充值服务的公司在玩家充值后需要正确记录交易信息。可以通过以下两种方法获取记录。

- **购买时，通过 HTTP 的 API 推送（push）记录给游戏服务器。**
- **游戏服务器使用 HTTP 的 API 从充值服务器拉取（pull）记录。**

交易记录包含下列信息。

- **用户在游戏中的 ID 或者充值账号。**
- **购买的商品 ID。**
- **购买时间。**

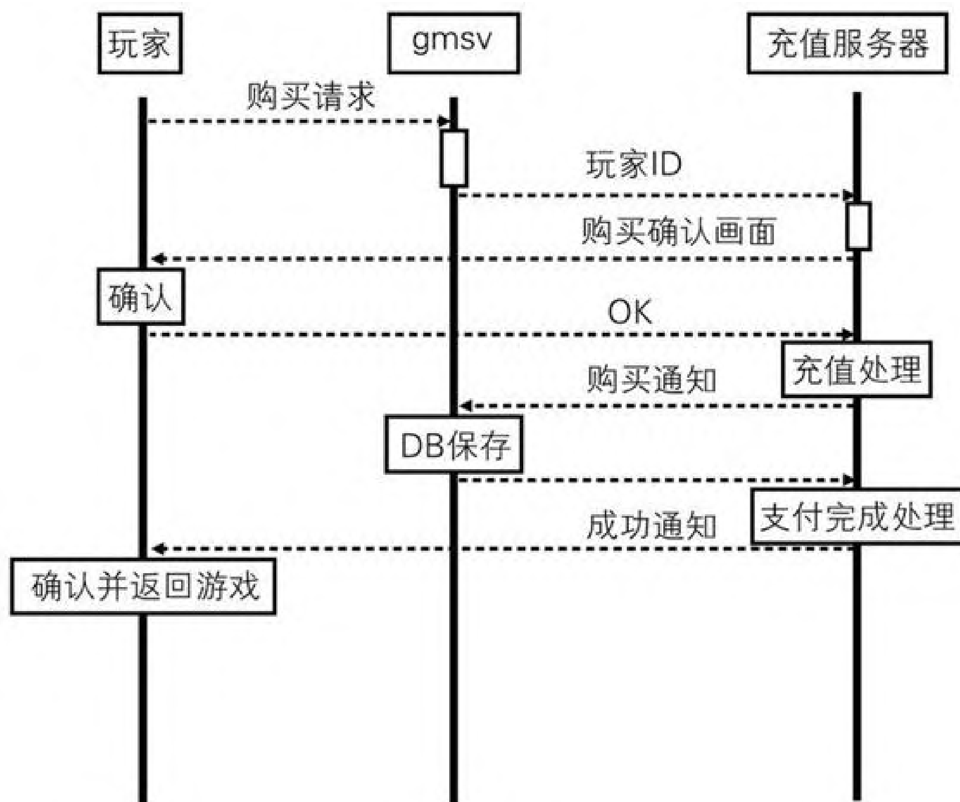
可以使用 JSON 等方式发送。

```
{
"USERID" : "hogehoge@gmail.com",
"PRODUCTID" : "KONLINE00000100",
"DATE" : "Thu Jul 15 06:02:25 JST 2010"
}
```

游戏服务器接收信息后更新服务器端的交易管理数据库内容，增加 100 个虚拟货币。这样玩家就可以使用这 100 个虚拟货币购买游戏中的道具了。之后就是游戏逻辑部分如何处理，也就是在游戏中判断“该玩家剩余的 100 个虚拟货币可以购买这个道具吗？”。

典型的序列图如图 6.6 所示。

图 6.6　购买商品和支付的典型序列图



在实际的系统中其实有很多种处理序列，基本上需要和充值服务器进行数次 HTTP 通信。结算公司也会提供 Perl 或者 C 语言开发的访问程序库。还有一种典型的方式是为多个游戏服务器设置统一的“网关服务器”，将游戏服

务器和结算公司的通信集中，并将所有充值相关的通信按时间顺序记录日志。

使用充值服务的好处

使用这些公司提供的充值服务的好处不仅仅是减少开发时间，从安全角度来说，因为信用卡号等用户的隐私信息不需要在游戏中保存，所以可以大大降低用户隐私保护方面的风险。

Facebook 等社交网络也开始提供类似的充值服务，实现基本上和上述的支付序列相同。

6.5.2 虚拟货币管理 P2P MO C/S MMO

用途：通用。

问题：丢失的话会比较麻烦。

实现：尽量使用现有服务，自己开发的话要做好日志记录（Journaling），从最开始就决定好“定期结算”的方针。

游戏中的虚拟道具种类很多，如果每个道具都分配一个充值服务公司的商品ID，数量会非常多。特别是使用多个充值服务时，每件商品都更新会非常麻烦，不能灵活的修改。所以大部分游戏都是只要用日元或者美元等实际货币购买了虚拟货币后，就可以使用它来购买游戏中的道具。

P2P MO 游戏和 C/S MMO 游戏

P2P MO 游戏和 C/S MMO 游戏都可以使用虚拟货币购买物品，其中 P2P MO 游戏一般使用现有的服务。例如 Uplay 或者 OpenFeint 等，在 Uplay 的网站购买虚拟点数后，使用该点数可以下载扩展内容或者解锁游戏内的功能（解锁道具），这些功能只需要调用简单的 API 就可以实现。

因为 P2P MO 游戏的程序是在玩家的机器上，所以玩家可以自己修改游戏的程序或者存档数据来获取虚拟点数。这种作弊的行为理论上是无法防止的，不过根据经验，99% 以上的玩家是不会这样做的。

在 C/S MMO 游戏中，游戏时间越长获得的道具越稀有，为了防止玩家作弊一般会自己设置管理点数的服务器。点数管理服务器的实现有一些需要注意的地方。

虚拟货币管理服务器的实现

虚拟货币管理需要满足下列功能需求。

- **可以显示总额。**
- **用户可以查询充值（兑换虚拟货币）记录。**
- **即使服务器出故障也不会轻易丢失虚拟货币。**

通过这些需求，可以看出表中不是存储“哪个用户有多少虚拟货币”，而是“用户进行过何种交易的历史列表”，因此也可以叫做（交易）日志方式。

表包含下列信息。

- **处理（交易）ID。**
- **玩家 ID。**
- **增加的点数（可以是负数）。**
- **充值服务的商品 ID，即增加点数的原因。**
- **日期。**

例如，*K Online* 游戏中 100 日元可以兑换 1000 虚拟货币，如果用户分 10 次每次 100 日元购入虚拟货币，这张表就会增加 10 行记录。

虚拟货币管理的注意点

就整个系统来讲，每次购入点数时都会增加一行记录，这导致该表越来越大，会影响系统的性能。大概 10 万以上用户、1000 万行记录时就会出现问题。所以，在服务器维护时应该运行批处理程序，像银行账号一样定期进行结算处理。例如，可以开发一个工具，把一年前的记录归结成一行，然后删除数据库中的旧数据。

虚拟货币的历史记录丢失是很严重的问题，可以使用 DBMS 的复制功能备份数据库，也可以备份硬盘或者使用 SQL 语句将数据导出到别的硬盘中。总之，采取必要的备份措施是非常重要的。

游戏中的虚拟货币在法律（指日本法律）上是不能兑换现金的，金额也不会很大（绝大多数玩家账号中的虚拟货币大概等值几千到几万日元），所以只要不是 AAA 游戏那样总额数百亿日元规模的游戏，就不用像银行系统一样建立完善的预算方案（而且这样的预算也做不了……）。

从安全角度来讲，应该严格限制可以远程访问虚拟货币服务器的主机。如果可以的 话应该限制从 Internet 直接连接，最好设置专门的网络线路，从后台网络访问。

* * *

本章主要介绍了从外部支持游戏策划、与充值付费相关的辅助系统，其中着重说明了“付费认证”和“虚拟货币管理”。

专栏　C/S MMO 游戏的收入

这里简单介绍一下 C/S MMO 游戏的收入，大致包含以下 6 种来源。

- **游戏软件销售**
- **小额付费（付费道具）**
- **网吧授权费**
- **包月费**
- **按时间计费**
- **广告**

游戏软件销售

业界的一些 AAA 顶级游戏（《最终幻想》、《魔兽世界》等），游戏安装包的售价大概在 4000～2 万日元左右。这些安装包包含以下物品。

- **客户端程序**
- **音乐和动画**
- **游戏中的 3D 图像数据**
- **1 张或几张游戏月卡**

包月费

需要玩家每月花费 300～3000 日元，大部分是 1000～1500 日元左右。多付几百日元可以让玩家增加一个游戏角色，或者在游戏中拥有虚拟的

“家”，以及其他各种形式的扩展选项。支付方式可以采用信用卡、WebMoney 这类预付费卡、便利店支付等多种形式。

从技术上来说，需要在游戏服务器和管理玩家付费状态的服务器之间建立付费验证机制，每次玩家登录游戏服务器时都需要进行付费验证。

小额付费（付费道具）

游戏客户端无偿提供下载，游戏本身免费，如果想获得两倍经验，或者想拥有个性化的游戏角色时可以通过购买特殊功能或者道具来实现，每次花费在 50 至几百日元左右。这种机制因为都是小额的消费，所以也叫小额付费。

任何人都可以很容易地加入游戏，所以可以使数量广大的玩家来试玩游戏，这是这类游戏的特点。另一方面，一部分玩家不知不觉中投入了大量的金钱，（对游戏开发者来说）这也是一个期待出现的“问题”。

在技术上，为了满足玩家的各种需求，需要推出各种各样不同价格的虚拟道具，所以实现“虚拟货币服务器”，让现金可以兑换虚拟货币是非常必要的。2009 年游戏业界通过这种方式带来的收入急速增长，游戏中 5%～15% 的玩家每个月平均花费 300～2000 日元左右。假如 10 万用户的 5%（5000 人）付费 500 日元，那么每个月销售额就是 250 万日元。

按时间计费

登录游戏服务器后，按照分钟计费的模式。在东南亚地区经常采用这种计费方式。汇率稍微有些偏差，一般换算过来大概 1 分钟 0.5～5 日元左右。

在技术上，需要实现的功能是登录游戏服务器进行付费验证之后，确认该玩家剩余的虚拟货币或者预付费点卡余额，并计算还能玩多长时间，到时间点后强制中断游戏。大部分游戏同时还提供小额付费的功能，所以当通过游戏服务器购买虚拟货币后，需要重新计算剩余的可玩时间。

网吧授权费

这也是东南亚地区常用的方式。东南亚地区很多网络游戏的玩家为了使用高端 PC 和宽带网络会在网吧玩游戏。

网吧中预先安装好最新的游戏客户端，并准备专用的启动程序，玩家开机后马上可以开始游戏。通过这个启动程序可以测定游戏时间，玩家支付给网吧的一部分费用会付给游戏运营公司。网吧的 PC 由网吧运营者管理，所以可以防止玩家通过作弊绕开付费环节。

广告

这种模式在很多免费的网页 MMOG 游戏中很常见，在游戏过程中，在游戏画面外部显示横幅广告，通过玩家的点击获取收入。

MMOG 游戏一般不用这种方式，市场规模比较小，一般单人网页游戏用的比较多。从技术角度来说不需要为游戏服务器开发额外功能。

＊ ＊ ＊

市面上的游戏一般会同时采用多种形式获取收入。在本书原稿截止时，它们被称作 F2P（Free to Play）的模式，因为可以免费游戏，通过小额付费比较容易获得高额收入，所以大约 90% 的 MMOG 都采用了这种方式。

6.6　其他辅助功能

最后介绍一下“游戏数据浏览、查询工具”和“敏感词过滤”两个辅助功能。

6.6.1　游戏数据浏览 / 查询工具 P2P MO　C/S MMO

利用存储功能在服务器端保存游戏数据以及 C/S MMO 类型的游戏，都可以访问所有玩家的游戏数据。如果可以对这部分数据进行高效的查询，在运营中发生问题时可以缩短故障应对的时间。

但是，游戏运营开始以后，为了应对每天产生的各种问题，可能会没有时间开发新的工具。因此，在运营前应该达到的最低要求是：能够以便于阅读的形式浏览玩家的游戏数据，并且具备关键词、时间、项目或者角色种族等数值查询的功能。实际上在游戏开发过程中，稍微有这样的意识就可以很容易地实现这些功能。提前准备是非常重要的。

游戏数据的保存状态

首先，游戏数据一定是以某种形式保存在 MySQL 的数据表中，基本形式包含以下 3 种。

❶ 清晰地划分为不同的数据列保存。

❷ BLOB 形式。

❸ 两种形式混用。

第 ❶ 种是比较理想的情况，比较容易查询。使用 Navicat 或者 phpMyAdmin 这样的查看表内数据的通用工具可以进行很细致的数据管理。不过实际游戏开发时，第 ❷、❸ 种情况应该比较多。

不是很清晰的保存状态时

出现第 B❷、❸ 种情况一般是因为游戏数据中的数据项太多，数据结构中还包括很多子类型。例如，玩家可以持有多个道具，每个道具又有几十个数据设定，甚至某些道具还包含角色持有宠物的信息，可以对宠物的成长有特殊效果。

对于这种复杂的数据可以严格按照数据表定义分成几十个表格管理，也可以使用 BLOB 的形式保存在一张“游戏信息”表中。这两种方式很难说哪个更好，但对于程序员来说 BLOB 的方式应该更容易一些。

笔者的观点是第 ❷、❸ 种形式可以有效缩短开发周期。总的来说就是：需要索引的数据应该划分出数据列，按照关系型数据库的原生类型保存，其他的数据则以 BLOB 对象形式保存。

选择可以阅读的形式

直接保存成 BLOB 对象会影响查询的性能，所以应该采用 JSON 或者 XML 这种不管机器还是人工都可以阅读的形式保存。特别要推荐 JSON，它支持数组、Map 等必要的数据结构，解析速度快，有各种语言版本的解析程序库，也便于人工阅读。不过需要注意文字编码、Float 数的四舍五入方法、程序库的习惯用法等问题。如果管理工具不仅要查询还会覆盖 / 更新数据，最好添加相应的自动测试功能，以确保玩家角色数据的读取和更新。

关键词查询

如果保存在 DBMS 中的数据是可以人工阅读的格式，那么就可以使用关键词查询游戏数据。不过还需要进一步的处理。

首先在理想的情况下，假设使用 MySQL，like 语句会查询所有行，所以当数据量有几百兆字节时是无法使用的。所以需要在 MySQL 中集成 Senna 等全文检索程序库。

对于数据量不太大的游戏，没有关键词查询需求时也可以使用 like 语句。不过为了避免服务器负荷过大，应该在有相同数据的从服务器上运行。

游戏的设定数据和数据库

游戏的设定数据比较复杂，相对于 RDBMS，ODBMS（对象数据库）更适合。根据笔者的经验，KVS 这样的对象型数据库虽然在商业游戏中还没有使用，但是今后也许会成为与 MySQL 一起使用的一种选择。

如果以上述方式将数据保存在 DBMS 中，之后用少量代码开发管理工具也是比较容易的。

6.6.2　敏感词过滤 P2P MO　C/S MMO

敏感词过滤是指在聊天、邮件或者角色名中出现不合适的词语或文字时，显示警告并删除的功能。比较典型的用途有防止电话号码泄露等。实现方式的判断条件如下所示。

- 主数据保存在客户端还是服务器。
- 关键词判断处理在客户端执行还是服务器端执行。

其他和黑名单基本相同，有所区别的地方是，

- 不需要用户动态增加关键词。
- 不想让用户看到敏感词列表（里面包含了开发者的名字或者公司名字等，有可能会被用于毁谤中伤等）。
- 不需要实时更新数据（也不需要停止服务器）。

这些都涉及到游戏发行公司的相关政策，应该按照相关规定决定实现的方式。如果是在服务器实现这个功能，应该按照与黑名单同样的负荷来考虑。

6.7　本章小结

本章简单介绍了交流 / 通信、客户端实现、运营、充值等网络游戏的必要辅助功能的实现方针。随着时间的积累，辅助系统会慢慢成为通信平台的基础功能，实际的程序开发会逐渐减少，但是如果理解了它们的内部实现方法，可以帮助大家更加灵活地应用相关服务。

第 7 章　支持网络游戏运营的 基础设施：架构、负荷测试和运营

发行网络游戏，必须做好运营的准备工作，仅仅完成了网络游戏的游戏程序部分是远远不够的。为了运营网络游戏，需要准备好运行程序的硬件基础设施。

运营网络游戏的基础设施和一般的网络服务基本相同。网络游戏的技术人员需要做到以下两点。

- 决定硬件基础设施的规格，搭建运营环境。
- 控制基础设施的成本。

到本章为止，系统的设计已经大致完成。具体来讲，服务器端的设计可以通过增加服务器数量来提升系统的整体性能（可以扩展性能）。在此基础上，关于使用什么样的服务器、多少台、是什么样的架构、可以达到什么样的性能这几点，通过负荷测试应该可以得到满足性能要求的最终设计。

不过，在向服务器销售商或者服务提供商询问报价和下订单时，只有这些信息是不够的，即便满足了信息上的要求，也还需要大量部署服务器相关的基础设计构建的工作。

本章在设计的基础上，详细介绍如何向销售商和服务提供商询问报价、下订单到实际的部署工作该如何进行，从基础设施架构（7.1～7.3 节）、负荷测试（7.4 节），运营开始前后的注意点（7.5 节）几方面详细说明程序员应该注意的事项。

7.1　基础设施架构的基础知识

本节主要介绍基础设施架构的基础知识，主要内容包括基础设施架构工作的基本流程、构成要素、基本条件和成本估算、用户数和负荷需

要的基础设施。

7.1.1 C/S MMO 和 P2P MO 的基础设施（概要）

C/S MMO 游戏因为游戏逻辑处理基本都在服务器端进行，所以需要的服务器资源较多，而 P2P MO 游戏只是玩家匹配、排行榜这些辅助系统需要在服务器端运行，所以需要的服务器资源较少。只要 P2P MO 游戏不是完全不用服务器，就需要了解一些和服务器运营维护相关的知识和技能。否则一旦遇到匹配服务器无法正常运行的时候，P2P MO 游戏就不能提供多人游戏体验了。

本章的前半部分首先介绍在 C/S MMO 和 P2P MO 游戏中通用的基础设施相关的知识点。然后以 *K Online* 为例，介绍 C/S MMO 游戏相关的知识点，最后以 *J Multiplayer* 为例，介绍 P2P MO 游戏相关的知识点。

我们首先从 C/S MMO 和 P2P MO 游戏中通用的知识点开始。

7.1.2 基础设施架构需要进行的工作 ——从开发整体来看

架构基础设施时需要进行的工作实际上从项目最初就开始了。策划内容的变化会导致基础设施的变化，所以开发者在工作进行过程中应该具有基础设施相关的意识。

- 策划概要的决定（→之前章节已说明）。
- 程序设计（→之前章节已说明）。
- 基础设施的大体估算（与实际成本可能有 2～5 倍左右的误差）

  →游戏开发者（程序员）根据程序设计书列出系统需求清单。
- 程序实现（→之前章节已说明）。
- 基础设施估算（与实际成本可能有两倍左右的误差）

→游戏开发者（程序员）参考实际的程序测试结果，进一步细化系统需求清单。

- **向销售商询问报价（→基础设施主管负责）。**
- **报价评估（决定支付金额）、下订单（→基础设施主管负责）。**
- **决定日程（→基础设施主管负责）。**
- **架构**

  →一般由数据中心的系统工程师提出系统架构的方案。游戏开发者确认该方案，特别是 DB 的架构要确保没有理解错误的地方。如果没有问题，由数据中心的系统工程师负责系统的架构工作，游戏开发者负责（游戏服务器端）系统的部署。
- **负荷测试，调试**

  →必要时将结果反馈给基础设施架构方。
- **上线。**
- **开始运营。**
- **设备和服务更新。**

以上流程中最需要评估的部分是实际付款之前的流程。一旦付款后，运营相关的合同就算生效，所以需要仔细评估之前的内容。接下来，我们看一下针对这些流程，应该怎样做评估。

7.1.3　基础设施的成本估算

支持网络游戏的基础设施全部是由商品化的设备构成，这是成本估算的基本条件。也就是 Dell 或者秋叶原电器街销售的硬件设备，或者很多公司提供的设备配置服务。这些都可以用金钱计算，并且可以比较不同公司提供的配置组合报价，计算出“成本大概多少钱”。

基础设施的成本如下。

- **初期费用**
    - 信息设备：服务器、交换器、路由器等
    - 软件：Red Hat 操作系统等
    - 其他：机柜、电缆、电源等
    - 服务：服务器监控费用、数据中心使用费、域名费、电子签名等
- **后续费用**
    - 网络带宽费
    - 电费
    - 软件：杀毒软件使用费等
    - 服务：Red Hat 的服务费、域名费等
- **保修费用**
    - 信息设备：设备故障更换、保修等

以上每个项目的成本都可以从下列两个方面来考虑。

- **金钱（单位：日元）**
- **时间（单位：小时）** [1]

> [1] 大规模宽带合同的谈判需要比较多时间，所以时间成本也包含在预算中会比较好。稍后会详细介绍。

基本流程是先自己估算，预期精度是 1 位有效数字，然后让销售商提出报价。实际合同相关的细节，应该根据市场状况做出更改，个人的客户关系也会影响价格，所以本书只估算一位有效数字。而且游戏公

司对技术人员的估算精度的期待也就是这个程度，更准确的估算应该由采购和合同相关部门负责。

7.1.4 成本的概念、单位

基础设施的成本由设备的“数量”和“单价”决定。以一台 10 万日元的服务器为例，单位是“台”，单价是 10 万日元，按照这种方式估算成本即可。例如，技术人员可以为采购人员准备以下清单。

- **市面上比较容易买到的常用产品，例如 Dell 的最低性能的服务器 OptiPlex 9xx 以上型号，价格大概 25 万日元左右，生产环境（Production）需要 25 台，测试环境（Staging）需要 5 台[2]，备用 2 台，总共购买 32 台**

  → 32×25 = 初期大概 800 万日元。

  → 由采购负责人筛选供货商。

- **Red Hat Linux 授权费每年 8 万日元，购买 10 个许可证**

  → 一年预计 80 万日元。

- **100Mbit/s 带宽 1 条**

  → 最便宜的套餐 1Mbit/s = 5000～1 万日元。

- **应用程序监控，需要两人全职**

  → 外包的话 50 万日元 / 月 ×2 = 每月 100 万日元左右。

- **操作系统和数据库的基本监控，使用 A 公司的“轻松监控服务”**

  → 大概 1 台服务器每个月 1～3 万日元，32 台服务器每个月大概 100 万日元。可以有一定折扣，应该会便宜一些。

[2] 测试环境相关内容稍后介绍。

有了技术人员提供的这个清单，采购负责人就可以和销售商进行谈判和采购的工作了。

7.1.5　网络游戏服务器在一定程度上可以接收的条件

网络游戏的服务器是典型的 OLTP（On-Line Transaction Processing，联机事物处理）系统。不过和一般商业用的 OLTP 系统不同，因为数据损坏对每个用户造成的损失相对较小，也不会造成人身伤害事故。发生系统故障只会对公司的商业价值造成损失。

每周几个小时或者每个月几个小时的停机时间不会对信誉造成很大损失。公司的基干系统可能要求一年的停机时间不大于 1 个小时（99.99%），对于游戏来说，1 周停机 1～2 小时（99%）是可以接受的。

所以，对于基础硬件设施来说，并不需要（昂贵的）高品质的服务器。应该优先选择性价比高的机器，服务更多玩家。按照这个标准，接下来，我们分别介绍各类基础硬件设施的成本计算。

7.1.6　硬件、信息设备

硬件主要包括服务器、存储设备、网络交换机、路由器、防火墙。如果需要架设 Web 服务还需要负载均衡器，P2P MO 或者 C/S MMO 游戏并不需要，这里就不再介绍。

服务器

服务器采用市面上的普通产品即可[3]。

> [3] 网站上就可以查询价格，笔者一般使用 Dell 的网站估算价格。其他厂商的产品价格相差也不会太多，只要不是特别的促销活动价格都不会有较大的差距。此外，购买二手设备的情况很少。

采购价格一般一台 10 万～30 万日元左右。“DB 服务器和其他服务器”应该区分开，其他的服务器可以适当压缩预算，因为这些服务器就算坏了，更换也比较容易。

服务器使用几年后，会不断升级，比起 CPU 运算速度，内存更容易出现不足的情况，所以一开始应该配置更大的内存，这样可以延长服务器使用的周期。特别是数据库服务器，需要配置更大的内存。这样即使用户大量增加，也可以保证服务器的性能。因为 MMO 游戏的用户会不断增长，所以必须配置大容量内存。

使用公有云服务 [4] 时，考虑到网络延迟问题需要在不同的地区选择不同的服务，日本可以在 Nifty 或 SAKURA Internet 的网站上估算价格，北美或欧洲可以使用 Amazon 等服务。公有云服务的价格每年都在变化，1 个月的费用大概相当于服务器费用的 1/2～1/5 左右。和服务器购买费用相比，使用 2～5 个月以上时，云服务会更贵一些。当然，云服务的成本中包含了服务成本。从整体来讲，在实际使用中，云服务的成本可能会相对低一些。

> [4] Amazon EC2 或者 Nifty Cloud 等云服务。为了区别于本章后面提到的私有云服务，这里称作公有云服务。

存储设备

可以选择 RAID1 或者 RAID5 的 Linux 服务器。没有必要使用高端机器，可以使用 MySQL 关系型数据库在多台机器上备份，从软件上确保数据冗余性。

使用公有云服务时，应该调整内存容量和服务器实例的数量，尽量减少磁盘访问。因为磁盘阵列会有多个节点共享数据，如果每次执行 SQL 查询都必须访问磁盘，会产生性能问题。

网络交换机

不需要高性能的机型，只要可以在所有端口间提供线速传输速度即可。不用使用光纤，也不需要第 3 层以上的交换机。48 端口大概是 10 万日元左右。不需要具备 99.999% 的可靠性。

使用公有云服务时，已经包含了网络交换机的费用，不需要另外计算。

路由器 / 防火墙

服务器直接暴露在互联网中是比较危险的。应该给服务器分配私有 IP 地址，通过路由器的 NAT 功能转换 IP 地址传送数据包。如果使用具有 ❶ 可以限定开放端口号和协议的功能、❷ IP 地址列表限制功能、❸ 数据包标头限制功能、❹ 保留日志功能、❺ 数据包流量限制功能的路由器，可以有效的防止网络攻击。

为了实现这些功能，需要能够检测 IP 数据包的第 3 层 [5] 功能的路由器。

> [5] 第 3 层和 OSI 网络模型相关的内容请参考第 0 章。

例如，“24 端口 Gbit 第 3 层智能交换机”这样的机型大概 20 万日元左右。使用 1 台，备用 1 台即可。在笔者有限的经历中遇到过两次路由器损坏的情况。路由器损坏造成的影响远远大于服务器，所以最好有备用设备。有些机型还具有自动替换成备用设备的功能。

公有云服务一般包含了路由器的费用。2010 年时大部分云服务还不能设定第 3 层网络协议的相关的参数（Nifty 没有提供这样的功能）。

7.1.7　软件

软件主要包括服务器操作系统、数据库管理系统、杀毒软件和虚拟化软件等。

服务器操作系统

笔者使用过的比较稳定的操作系统有 Solaris、FreeBSD 和 Linux。Linux 系统推荐 Red Hat 各版本和 CentOS。Red Hat 的企业级用户很多，优点是熟悉其监控的服务公司、工程师比较多。目前，网络游戏市场选择 Linux（Red Hat）的公司在逐渐增多。

如果所有服务器都购买 Red Hat Linux 的许可证（支持服务），成本会很高，可以只为一部分需要得到技术支持的后台服务器购买许可，前端或者验证用的服务器则使用 CentOS 以节约成本。

一个许可证一台机器一年需要 8 万日元左右。购买了许可证后，一般会自动安装安全补丁程序，可能的话应该每台服务器都安装这些补丁

程序，但是现实情况是，游戏服务器的预算有限，只能针对有数据保护需要的服务器安装。

公有云服务一般都可以使用 Red Hat 或者 CentOS。

数据库管理系统

数据库管理系统和 Linux 一样，像 MySQL 或者 PostgreSQL 这样的产品都提供了有偿服务和免费服务两种方式。笔者参与过的项目使用的都是 MySQL，有的也使用 PostgreSQL、Microsoft SQL Server 或者 Oracle。网络游戏业界很少保存图片等大尺寸数据，所以使用 KVS 数据库的还很少。

第 4 章中已经介绍过，在网络游戏的开发中，主要的逻辑是在游戏服务器端实现，通过访问数据库保存数据。所以很少使用复杂的数据主键、存储过程、复杂的事物处理以及集群（Cluster）等功能。

因为逻辑并不复杂，一般不需要使用付费的性能优化服务，实际的游戏项目中也很少使用 MySQL 的付费服务。如果是 Oracle 数据库，公司内部又没有熟悉系统的人，也很难申请付费支持服务需要的预算。所以 DBMS 相关的成本几乎可以不用考虑。

杀毒软件

需要在 Linux 服务器中安装杀毒软件，用来检测是否运行了恶意程序或者安装了不正当的程序库。

例如 HDE 面向服务器的产品，一台服务器一年 5 万日元左右。和 Red Hat 的许可证一样，如果为所有服务器都购买，预算会很高，所以只针对需要数据保护的服务器购买即可。在公有云服务上也可以安装杀毒软件。

虚拟化软件

购买物理服务器并投入使用后，可以使用虚拟化软件 VMware 提高服务器的资源使用效率，而且服务器资源的分配也比较容易调整可以方便管理。非常适合对于延迟要求较高的游戏服务器和中继服务器以外的服务器。这也叫做“私有云服务”。

如果是 Beta 版的游戏服务，可以选择免费的 VMware ESXi 搭建虚拟服务器环境，之后的生产环境可以替换为商业版。VMware 的许可证价格体系比较复杂，在开发网站上很难了解到具体价格，需要通过商务谈判确定最终价格。不过一般来讲，10 万～20 万日元的 Dell 服务器一台一年需要 3 万日元左右的 VMware 支持费用。因为成本非常高，所以对于单用户付费额较小的游戏来说，在预算上并不适合。

目前还没有和 VMware 的 vSphere 产品功能相当、管理方便的开源虚拟化软件，不过市场需求很大。对于开发环境和测试环境，使用免费的 ESXi 可以有效的减少物理服务器台数。

公有云服务包含了虚拟化软件的使用费。Nifty Cloud 中就使用了某个版本的 VMware。

7.1.8　数据中心相关的成本

和数据中心相关的成本包括数据中心使用费、数据中心建设费等。机架、线路、电源等设备的费用也包含在使用费中 [6] 。

> [6] 公有云服务的使用费中也包含了这些设备的费用。

数据中心使用费

使用数据中心时，实际上是占用了最珍贵的土地资源。成本的计算单位是“机柜”（Rack）。图 7.1 是并排放置的两个长机柜，像这样的机柜在一般的数据中心里有几百个。

图 7.1　数据中心的机柜

※ 图像提供：GPG-Solutions http://www.gpg-solutions.com/

用来表示服务器数量的“机柜”（19 英寸机柜）有全球通用的工业标准（TIA/EIA-310-D）。如果购买 Dell 等厂商的服务器，一般是如图 7.2 那样的高 1.75 英寸，宽 19 英寸，像批萨外卖盒一样的机箱。这种服务器的高度称作“1U”，1 机柜是 42U。考虑到电源、电缆线的空间，42U 大概可以放 15～30 台服务器。如果电源容量允许，可以使用“刀片服务器”[7] 进一步提高服务器密度。

> [7] 刀片服务器（Blade Server）。一个主板上即可搭载的、外形薄而细长的服务器。

图 7.2　1U 服务器

※ 图像提供：日本电器株式会社 http://www.nec.co.jp/

图片中的产品是 NEC Express5800 系列 R110c-1

数据中心的空调、冗长电源、停电对策、安全、抗震、服务器检测服务等各种服务是成套配备的，所以不仅仅是土地的成本。42U 的机柜每个月大概 30 万～50 万日元。在数据中心的价格组成中，土地租金所占比重较大，特别是城市中心的数据中心价格非常昂贵，垃圾填埋场或者郊外相对便宜一些。通过各种讨价还价，42U（也就是 1 个机柜）如果能谈到 20 万日元就算成功了。数据中心一般不是以 1 个机柜为单位出售，通常以 1/2 个机柜为单位。

Alpha 测试等初期也经常会采用图 7.3 那样简单的设备。

图 7.3 简单的设备

数据中心建设费

※ 图像提供：Neda Communications http://www.by-star.net/techspeak/datacenter/

数据中心建设费

虽然也可以自己在数据中心架构环境，但一般会让数据中心来负责。这样需要计算人工费 + 材料费。因为游戏服务器的设定并不复杂，人工架构费时不多，设置 1 个机柜用不了 1 个月，花费在 50 万～100 万日元左右。因为都是普通的产品，所以 1 个机柜的材料费大概花费 10 万～20 万日元。这个建设费只是初装费，如果需要增加服务器，需另行计算。

7.1.9　服务费（数据中心以外）

服务费相关的成本包括服务器监控服务、域名使用费和电子签名服务费等。

服务器监控服务

服务器监控包括 4 个部分：❶ Linux 等操作系统是否正常运行、❷ DBMS 是否正常运行、❸ HTTP 或者 SSH 等协议是否可以正常访问、❹ 游戏服务器程序是否正常运行。

❶～❸ 可以交给专门的服务公司做，比如常用的监控软件 Nagios ，通过设定功能列表中的选项可以实现 ❶～❸ 部分的监控。使用这些工具的监控公司有很多，可以在网上搜索“服务器 监控 服务”。

1 台服务器每个月的监控成本大概 2 万日元左右。初期一般需要支付一至几个月的费用。如果根据服务器台数按照一定比例进行收费的话，成本会比较高，可以考虑开发能够确认所有服务器运行状态的工具让监控公司使用，或者仅仅监控比较重要的需要保护数据的后台服务器。Alpha 测试、Beta 测试阶段不监控的情况也很多。

域名使用费、电子签名服务费

1 个域名 1 年的使用费大概几千日元。根据服务器的不同价格略有差异，这里就不详细介绍了。

使用 AppStore 等现有的发售平台时，电子签名服务费一般包含在使用费中。不同平台的费用也不一样，本书不再详细介绍。

7.1.10　网络带宽费

日本最贵的网络带宽费大概是 1Mbit/s=1 万日元 / 月，其他国家可能有比较便宜的。这个费用在不同地区差异很大，所以在国际市场运营时，需要进行许多商务谈判。一般合同时间越长，带宽越大，价格也会越便宜[8]。

> [8] 例如 10Mbit/s 的月租金是 17 万日元左右，如果是 100Mbit/s，只需要 30 万日元。请参考 http://www.kddi.com/business/ethernet_senyo/charge.html

由于目标客户的不同，不同的游戏的峰值时间和峰值比有很大区别，所以根据游戏策划内容提前估算通信量，并以此作为商务谈判的基础，可以进一步增加议价空间。因为需要私下进行协商，所以商务谈判的负责人也会很大程度影响最后的价格。需要带宽特别大时，可以直接接入 ISP 的专线网络。此外，还经常有以很低价格调配空闲的光纤或者路由器容量的情况。通过各种方式最后有可能把价格谈到 1Mbit/s = 1000 日元左右。

使用公有云服务时，可以使用的带宽是整体能提供的最好状态。

7.1.11　电费

1 台多核 / 多处理器服务器每个月电费 5000 日元左右。不与电力公司进行特别的协商的话，大概相当于 100 瓦电灯的电费。工业用电或者夜间用电也许有一些优惠，不过笔者没有相关经验就不再详细介绍。一般电费都包含在数据中心的使用费中，作为游戏开发公司或者运营公司来说也没有必要和电力公司协商。

当然，公有云服务的使用费也包含了电费。

7.2　开发者需要知道的基础设施架构技巧

本节整理了一些面向开发者的基础设施架构技巧。主要为大家介绍基础设施架构中最重要的用户数的估算、开发者需要了解的基础设施架

构技巧和相关工具。

7.2.1 服务规模的扩大 / 缩小——用户数和需要的基础设施规模

网络游戏的基础设施架构中最重要的一点是正确估算决定基础设施规模的“用户数”。与此对应，“为了应对用户数的增减，能否相应调整基础设施规模”是一个关键。

这里所说的“用户数”包括“注册用户总数”和“同时在线用户数”两层含义。注册用户的增加会逐渐影响数据库的处理负荷、如果同时在线用户数较大，会影响前端服务器的性能。游戏上线后，同时在线用户数所占注册用户总数的比例会逐渐降低。

理想的情况是可以根据用户数灵活地调整服务器资源，不过实际情况中不管是使用物理服务器还是云服务都有最少服务器数量的限制。特别是对于 MySQL 等数据库服务器，之后调整的风险和成本都比较高，尽可能在一开始就确定适合的架构。反之，对于 gmsv 等前端服务器来说，增加或者减少 1 台服务器就比较容易。如果使用云服务，增加或者减少几台至几十台服务器可以在 1 天之内完成。

7.2.2 典型的环境

典型的环境有 3 种形式。

- **初始环境**

  公司内部搭建的小规模服务器。服务器在 20 台以内。网络采用 KDDI 等公司的 10Mbit/s 宽带。配置公司内部的电源系统。

- **数据中心**

  和数据中心签约，在机柜中设置自己购买的服务器。服务器数量没有限制。

- **公有云服务**

使用 Nifty Cloud 等云服务。服务器数量没有限制。

网络游戏一般不使用 Web/ 邮件服务经常用到的 VPS（Virtual Private Server）服务，所以就不作为典型的环境介绍。

不容易估算的条件

运营开始后，如果有下列市场活动会增加用户数估算的难度。

- **可以免费试玩**
- **没有用户数量限制的试玩**

虽然想尽可能避免上面的情况发生，但是经过 Alpha 测试、封闭 Beta 测试、公开 Beta 测试到正式上线几个阶段后服务器会变得饱和，从销售的观点来说，需要做这些市场推广活动。

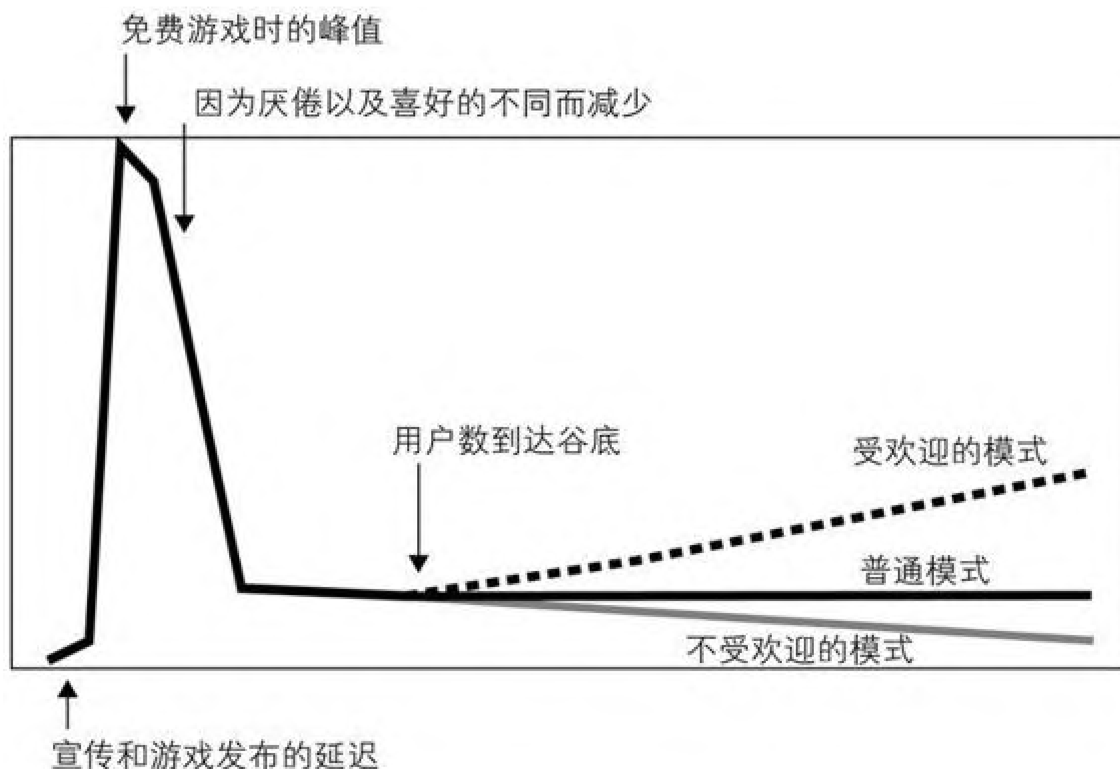
根据笔者经验，如果是对某个类型的游戏没有经验的团队，进行“没有限制免费试玩直接发布”，一定会失败。所以要尽可避免这种情况。

7.2.3　负荷曲线

游戏运营开始后 [9] ，通常的用户数曲线如图 7.4 所示。

> [9] 补充一点，实际运营前必须进行用户数的预估。

图 7.4　运营开始后的用户数（通常情况）



最初峰值的玩家数量根据游戏的发售平台的特征、广告宣传方式（预算）、目标用户层的不同会有很大区别。

能带来最大用户流量的一般是 Mobage、Gree、mixi 以及 Facebook 这样的有数以千万活跃用户的社交网络，嵌入了这些社交网络平台的免费应用，第一天就带来 10 万以上注册用户的案例有很多。

相反，有些游戏机平台的面向硬核玩家的游戏，没有免费试玩，在几乎没有怎么宣传的情况下，用户数量也会一点一点的增长。被称作“欧洲系 MMO”的 *Runescape* ，CCP Games 的 *Eve Online* 等游戏就是这种经过几年用户逐渐增长的模式。这类每天新增几百注册用户的游戏有很多。不过现在 Twitter 等社交网络的信息传递速度非常快，“欧洲系”这种模式可能会逐渐减少。

最初的设计需要考虑峰值的应对

网络游戏开发者在最初设计时，就需要考虑峰值到来时如何应对。

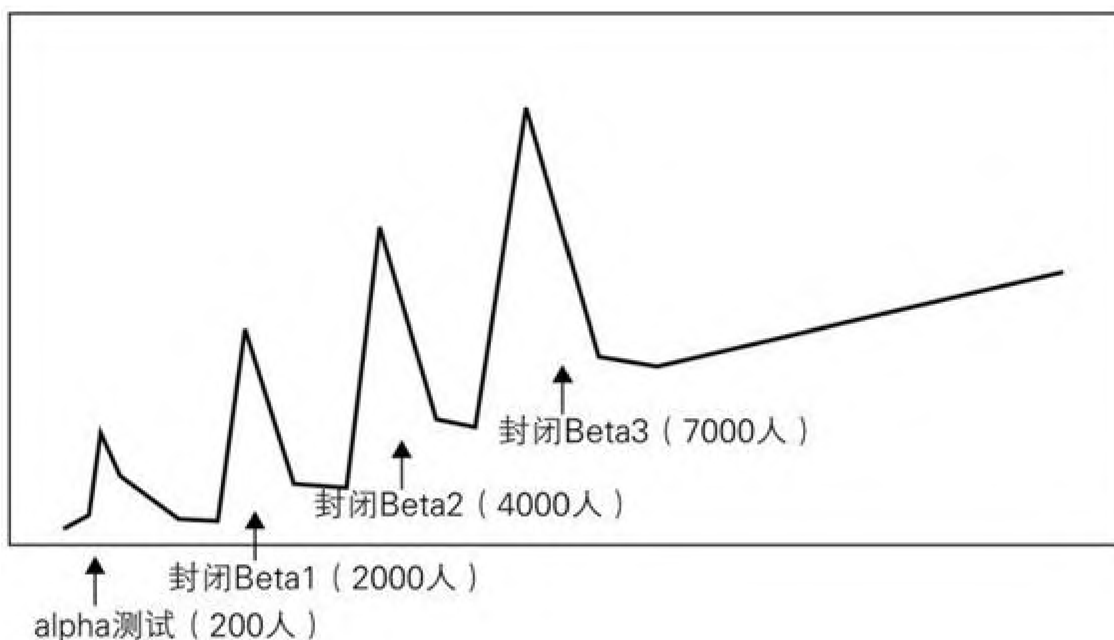
花费几千万日元甚至几亿日元开发的游戏成功的几率不到 1/3。大多数情况是如图 7.4 的“常见的模式”或者“不受欢迎的模式”所示。

所以在上线最开始时不可能准备很多的服务器。应该尽可能分为多个阶段，根据用户的反馈，一边修改游戏，一边逐渐增加服务器数量。另外，在程序设计上应该尽量减少游戏正常运行时需要的服务器数量，并考虑到后期性能的扩展性。

K Online

以 *K Online* 游戏为例，4 次测试后的用户数据如图 7.5 所示。考虑这个测试计划时，Beta 测试次数过多的话玩家可能会感到厌倦，所以需要考虑运营上的影响，尽可能减少测试次数。

图 7.5　4 次测试后的用户数量变化



K Online 采用了典型的模式，首先在“20 台服务器以下的初始环境”进行“alpha 测试”，在数据中心进行剩余的 3 次“封闭 Beta 测试”，并假设之后用户会逐渐增多。

K Online 不是那种动作性很高的游戏，可以接受比较大的延迟，所以使用 Nifty Cloud 或者 Amazon EC2 等公有云服务和使用数据中心（在性能表现上）是差不多的。

J Multiplayer

J Multiplayer 是 P2P MO 游戏，不需要游戏服务器，除了中继服务器都可以利用公有云服务。

P2P MO 游戏的游戏逻辑部分不是在服务器端实现，所以一般不需要更新服务器程序，Beta 测试不用像 *K Online* 一样进行多次。

7.2.4 面向开发者的基础设施架构要点

架构基础设施时，开发者的实际工作内容包括下列几点，我们依次说明。

- **服务器部署**
- **登台环境**
- **服务器的监视**
- **日志输出**

7.2.5 服务器部署

服务器部署是指将新版本发布到每台服务器，更新版本的相关工作。大致分为以下几个阶段。

- **服务器程序的测试**

  拿到服务器程序后需要进行相关测试（请参考接下来将要介绍的“登台环境”）。

- **将测试完成的程序打包（使用 RPM 或者 tar.gz 等工具）**

  打包后可以更加明确需要安装的文件，便于版本管理。在服务器保留旧的版本可以在需要时回复到以前的版本。

- **上传打包后的程序到各个服务器并解压缩**

  C/S MMO 游戏的服务器设定数据非常大，服务器数量也多，文件拷贝需要花很多时间。如果在停止服务器后才解压缩，会延长停

机时间。

- **暂停用户登录和注册等功能**

  事前使用管理工具通知用户，然后更改防火墙设定或者服务器运行模式。

- **中断已经登录的所有玩家的游戏（强制退出）**

  一般会在 C/S MMO 游戏的服务器程序中实现强制退出的功能，通过命令运行。P2P MO 游戏可能只需要通过防火墙隔断新建游戏的访问。中断玩家的游戏前需要保存所有必要的数据。这里需要注意，如果同时中断所有服务器，会产生过量的数据库保存请求，所以应该错开时间强制退出游戏。

- **停止旧版本**

  实现停止服务器程序的功能，通过命令行执行。在各个服务器设置通过 HTTP 访问的内部端口，限定只能内网访问，然后使用 curl 命令[10]或者其他管理工具会比较方便。不能正常退出时，等待超时后发送信号强制 kill。

- **备份数据库**

  如果是 MySQL，可以使用 mysqldump 等命令进行本地备份。这样可以缩短维护时间，之后再将备份数据拷贝到其他地方。

- **需要修正数据库时，运行相应的 SQL 批处理**

  对表的结构进行修改时，先准备好批处理文件。批处理文件的运行时间很多时候无法获取，所以可以通过事前复制的数据来测量执行时间。

- **确认数据库**

  准备简单的 SQL 查询脚本。

- **启动新版本的服务器程序**

使用管理工具在各个服务器通过 SSH 执行启动脚本（可以配置在 /etc/rc.d/init.d 中，或者 /var/k-online-sv/bin 中）。

- **新版本服务器程序的测试**

  这个测试包括使用测试工具的自动化测试和手工测试。最好准备可以进行自动化测试所有服务器是否正常运行的简单工具。C/S MMO 游戏可以在所有服务器测试用户登录和角色的正常移动，P2P MO 游戏可以使用测试 ID 验证排行榜和匹配功能。通过这个脚本可以发现大部分基本的问题。

- **开放用户登录**

  还原防火墙设定。停机维护时间到此为止。

- **从服务器转移备份好的数据和日志文件**

  维护时间结束后，使用 SCP（Secure CoPy）等工具转移数据库的备份文件。可以使用管理工具自动进行该操作。

> [10] 指定 URL 获取 Web 页面的命令。

维护时间和自动化发布

维护时间是指限制用户访问到解除限制之间的时间。对于 C/S MMO 和 P2P MO 游戏都是一样的。

为了防止出现误操作并尽快完成维护工作，应该尽可能采用自动化处理的方式。如果有几十台至几百台服务器，上面提到的这些操作可以使用 SSH 和 SCP，根据情况还可以使用 expect[11] 命令或者 Shell 脚本实现全自动化处理。有几千台服务器时，Twitter 或者 Facebook 那种使用 BitTorrent 的方法很有名。

> [11] tcl 的扩展语言。可以自动执行 telnet 或者 SSH 等对话通信。

7.2.6 登台环境

登台环境（Staging）不是生产环境，但和生产环境相近，是在生产环境运行时用来测试新版本的环境。本书最开始就提到对于 C/S MMO 和 P2P MO 都需要登台环境。

例如，*K Online* 的生产环境有 69 台服务器，登台环境和封闭 Beta 测试时的 4 台服务器构成相同。

使用登台环境测试时需要注意，要在不同的版本中选择合适的组合进行测试，并防止版本发布的操作失误（把测试版发布到生产环境），和普通的 Web 服务或者办公系统中的注意点一样。

网络游戏的特殊注意点

网络游戏的特别之处在于需要面向不同的用户群体进行测试。所以登台环境应该准备多个，❶ 面向开发者，❷ 面向经过特殊设定的一般用户。有时还要准备多个面向一般用户的环境。

准备面向一般用户的登台环境时，可以在用户信息中添加用来判断是否可以访问测试环境的标记，在用户登录时判断是否有权限，如果没有权限就禁止访问。

7.2.7 服务器的监控、生死监控

接下来，我们整理一下和服务器监视相关的用语。生死监控是指确认“服务器是否处于可以使用的状态”。状态监控是指除了生死监控之外的详细状态监控。

服务器的监控和普通的 Web 服务或办公系统的机制相同（一般用途）。因为服务器监控中会使用非 HTTP 协议，所以需要实现“专用协议和 HTTP 协议的转换工具”。这里只是简单说明，网络服务相关的详细技术请参考相关专业书籍。

服务器监控从低层到高层分为“服务器操作系统监控”、“MySQL 等数据库管理系统监控”、“应用程序监控”和“用户行为的正常性监控”。

- **服务器操作系统监控**

监控服务器上运行的操作系统是否正常。设置 MRTG（Multi Router Traffic Grapher），监控 CPU 使用率、内存使用率、硬盘使用率、进程数、通信量等信息。

- **MySQL 等数据库操作系统监控**

监控运行在服务器的 Linux 操作系统上的 MySQL（数据库管理系统）的运行状态。❶ 进程是否正在运行、❷ CPU 使用率是否在正常范围、❸ 是否能在规定时间内执行完特定 SQL 查询、❹ 是否发生了速度比较慢的查询、❺ 数据表大小的监控、❻ 单位时间处理的查询数、❼ 线程数的监控，是否发生异常。可能的话这些监控记录都应该保存在日志文件中（之后会说明），未来发生故障时可以作为解决问题的参考。

- **应用程序监控**

监控游戏服务器程序的运行情况。游戏服务器只接收数据中心内部的监控服务器的连接请求，并通过 HTTP 端口返回结果，返回信息包括（1）现在的同时连接用户数、（2）内存中对象的使用量（敌人数量等）、（3）单位时间内主循环的处理次数、（4）活跃的 TCP 连接数等信息，使用 cacti[12] 等监控工具处理这些信息。如果没有收到应答消息，就表示服务器程序可能出现异常，可以向相关人员发送警告邮件，如果返回数据中有超过正常范围的情况也自动发送邮件报告问题。

- **用户行为的正常性监控**

即使服务器程序运行正常，只要玩家的游戏体验受到影响，就应该停止服务器进行改善工作。在网络游戏中，特别需要监控影响游戏平衡性的恶意行为、不良商家或者玩家的行动。可以统计单位时间内游戏中的操作次数，并将结果整理为方便查看的表格。比如显示持有金额增加最多的 10 个玩家，或者聊天记录中出现禁止词汇的玩家等。

[12] 和 RRDTool 一起使用的图形化工具。http://www.cacti.net/

7.2.8 日志输出 / 管理

游戏服务器的运行日志是快速解决问题的重要参考信息。输出的日志是通用的，特别是对于 C/S MMO 游戏，仅仅在很多玩家在线时才发生的问题往往只出现一次，很难再现，所以从运营角度来说，应该尽可能的保留日志。一般使用日志服务器来实现，syslog 工具有比较慢、输出文件较大、时间戳不准确等问题。下面我们来进行详细说明。

日志保留的策略 ——“尽可能保留所有日志”并且“原因和结果都保留”

日志保留的策略有两点：“尽可能保留所有日志”并且“原因和结果都保留”。“原因”是指用户的操作请求，“结果”是指操作的结果[13]。

> [13] 以 *K Online* 为例，“攻击敌人”是原因，“敌人被击倒”是结果。

当然，如果所有日志都保留，数量会非常多。例如，*K Online* 同时在线数 3 万，平均每秒接收两次客户端的操作请求，如果所有的操作都记录的话，每秒有 6 万次操作，每个操作 1 行数据的话就有 6 万行，每行 100 个字，日志总共 6 兆字节。1 天有 10 万（100k）秒，1 天记录的日志有 6 兆字节×10 万 = 600 吉字节，1 个月 600 吉字节×30 = 18 钛字节。不过日志文件的压缩率比较高，一般可以压缩到 1/10，所以 1 个月有 1.8 钛字节。记录 *K Online* 游戏的所有操作的日志大致是这个数量级。

K Online 需要记录的“结果”信息包括 ❶ 道具交换 / 取得 / 使用、❷ 玩家技能的使用 / 获得、❸ 经验值、HP 等角色状态的变化、❹ 对敌人造成的伤害、技能效果 / 击倒的结果、❺ 和商店 NPC 的所有对话、❻ 登录退出记录、❼ 地图跳转记录。“结果”信息一般比“原因”信息少，但更重要。

像 *K Online* 这样的 C/S MMO 游戏，如果数据丢失或者损坏，因为玩家那里也没有备份数据，所以运营方需要承担所有责任。因为 bug 等原因丢失道具的情况下，补偿的依据是日志文件，还有就是查询数据库备份或者 SQL 查询日志。因此作为运营工作的证据，应该尽可能的收集日志。

日志输出方法 ——推荐的组合、syslog 的问题

日志的输出经常采用下列方法的组合。

❶ 通过标准输出中的管道，传输给其他进程或者在文件中保存。

❷ 写入文件并传送给其他工具。

❸ 通过网络发给服务器。

❹ 写入 syslog。

和 syslog 相关的工具有很多，比较方便，但是设计上没有考虑到海量数据的情况，游戏服务器的日志输出对于这些工具来说有些负荷过大。所以方法 ❹ 一般只限制输出服 务器启动 / 关闭日志或者不能恢复的错误日志，主要的游戏日志采用方法 ❶～❸。

笔者推荐方法 ❷ 和 ❸ 的组合。首先在文件中记录日志，然后将同样的信息通过网络再发送给日志服务器。这样即使出现网络故障时，文件中也有日志信息。如果觉得删除文件或者部署时比较麻烦，也可以只保存在日志服务器上。

日志服务器

使用日志服务器时，接收到的所有日志都会按时间顺序直接保存到文件中，每 100 兆字节左右就保存到新的文件中。

使用日志服务器的优点是，可以把所有日志按照时间顺序排列，从中可以追踪多台服务器的玩家活动（根据 ID 使用 grep 搜索）。

上面使用各种方式记录的日志可以灵活运用，业界并没有标准的做法。也没有统一的标准格式。相对来说，普遍使用的方法有以下几种。

❶ 使用 grep/sed/awk/perl/ruby/python 等处理日志文件。

❷ 按照 JSON 格式输出日志便于程序处理。

❸ 按照 CSV 格式输出日志便于在 Excel 中处理并存入 DB。

❹ 不保存文件直接 insert 到 MySQL 的表格中。

方法 ❹ 在实际运用中还有很大的改善余地。确定了日志各行中的时间、玩家 ID、服务器 ID 等相关数据列后，可以使用 LIKE 语句进行查询，这样比起 grep，在查询上会方便很多。

但是对于几百吉字节这样规模的日志文件，使用 like 查询不是可行的做法，应该对各列添加索引，然后使用 Senna 等全文检索引擎。不过这种情况下，因为输出日志时会更新索引，所以可能会影响输出速度。

期待游戏开发者未来能够开发出可以高效利用游戏服务器的大量日志信息的工具。

7.3 *K Online* 、*J Multiplayer* 游戏的基础设施架构

第 4 章和第 5 章分别介绍了 *K Online* 和 *J Multiplayer* 的游戏设计，并估算了和基础设施相关的服务器台数以及成本。本节主要介绍架构工作的要点。

7.3.1 *K Online* 的基础设施

第 4 章中，*K Online* 游戏有 3 万同时在线玩家时，预计需要 58 台运算服务器和 11 台存储服务器。初期费用大概是 2300 万日元。

3 万同时在线是游戏大卖后的事情了，实际上 *K Online* 的运营分为 4 个阶段 Alpha 测试、封闭 Beta 测试、公开 Beta 测试、正式上线（当然只是作为例子）。最开始架构基础设施时并不需要那么多服务器。

各个阶段测试需要的服务器数量可以根据“各个阶段的测试目的”、“允许参与测试的玩家数”来决定。下面分阶段进行说明。

Alpha 测试

Alpha 测试的目的是“测试系统的基本功能”。

假定用户数为 100，Alpha 测试要尽可能控制成本，并确认所有功能都能正常使用，所以测试用户设定为 100 个。

测试在初始环境进行。所有的服务器进程运行在一台服务器上，只需要把 dbsv 配置在别的网络的服务器上。此外还需要防火墙、前端服务器 1 台、后端服务器 1 台。

封闭 Beta 测试

封闭 Beta 测试主要目的是验证游戏服务器的性能。

用户数设定为 3000 人。游戏服务器主要使用的资源是 CPU 和内存，所以最开始需要测定这个部分的性能。另外，其他服务器的性能也需要测定，获得的结果可以作为公开 Beta 测试时估算服务器台数的依据。因为需要知道游戏服务器的性能极限，所以 1 台就够了，让尽可能多的用户登录服务器测出性能极限。但是为了避免用户过多造成无法登录的情况，所以设定为 3000 人。Beta 测试中典型的活跃用户比例在 20%～50%，3000 用户的话，期待的服务器单核同时在线人数是 1000 左右。

测试环境和 Alpha 测试一样在初始环境进行。gmsv 放在另外的服务器，其他的分别配置在 4 台服务器中，服务器数量比进程数量少，总共 5 台。

公开 Beta 测试

公开 Beta 测试的目的是验证包括后端服务器在内的系统的整体性能。并且还要测试能否使用多台服务器进行性能扩容。

不限制用户数量。

基础设施的规模和生产环境一样，需要 69 台服务器。这个阶段的测试使用“数据中心”。公开 Beta 测试的基础设施架构需要采用和生产环境相同的体制。这里一口气增加了大量服务器，在购买服务器（签订云服务合同）之前，需要测试基本的性能。所以 *K Online* 游戏最少需要 3 个阶段的测试工作。

另外，使用公有云服务时，*K Online* 的专用私有云服务获得了服务方提供的特殊优惠价格，所以选择他们的服务（比较常见的情况）。

7.3.2 *J Multiplayer* 的基础设施

J Multiplayer 是 P2P MO 类型的游戏，没有游戏服务器，只有运行辅助系统的服务器。第 5 章没有假设同时在线玩家数，这里也同样假设 3 万同时在线玩家来进行基础设施的估算。

J Multiplayer 的辅助系统包括“排行榜”和“玩家匹配”。如果使用现有服务的话就不需要服务器了。所以也不需要其他的基础设施。使用 Uplay 或者 OpenFeint 服务即可，非常方便。

自己开发部分辅助系统

这里说明一下自己开发辅助系统的情况。

J Multiplayer 游戏内容比较简单，未来计划移植到 Flash、游戏主机和 PC 等各个平台，另外细节的处理也希望进行一些定制化开发，所以排行榜和玩家匹配系统决定自行开发（比较常见的情况）。其他的辅助系统，如新闻发布、付费系统等都使用现有的服务。

因为想开发定制化的游戏内容，所以考虑到开发效率，排行榜和匹配系统的后台使用 MySQL，使用 Python 开发 Web 应用，游戏客户端使用 HTTP 协议访问服务器，采用无连接的方式实现。HTTP 协议的话可以在所有平台使用。

首先进行负荷测试、估算基础设施

排行榜和匹配系统实现的注意点在第 6 章已经做了介绍。不管哪个系统，出错一步都会浪费大量计算资源，需要特别注意系统的设计。细微的逻辑错误都会极大地影响性能，所以这里先进行负荷测试，再根据结果来进行基础设施的估算。测试步骤如下。

- **根据游戏策划内容决定需要测试的处理。**
- **准备测试环境（Web 服务器 1 台、DB/MySQL 服务器 1 台）。**

- 进行简单的负荷测试。
- 根据测试结果进行基础设施估算。确认是否在允许范围内。
- 搭建生产环境。
- 基础设施准备。
- 在生产环境进行负荷测试。

最初进行的简单的负荷测试是关键点。

排行榜和玩家匹配系统的负荷测试都比较简单，所以先通过测试得到 CPU 等硬件资源的使用量，以此作为基础设施估算的基础。

同时在线数和注册用户数 ——排行榜的典型负荷测试步骤

这里介绍一下排行榜的典型估算步骤。

如果 P2P MO 游戏同时在线数是 3 万，注册用户数大致在 50 万～100 万左右。同时在线玩家是指某个瞬间正在游戏中的玩家。以手机游戏为例，意思是指一般在公司或者学校的休息时间，同时在线数会达到峰值，这个时候人数为 3 万。

所有玩家不可能同时进行游戏，根据游戏类型不同，“游戏的注册用户数 = 注册用户数”和“同时在线数”的比例也不同。C/S MMO 游戏一般同时在线比例是 10%～30% 左右。P2P MO 会低许多，在 1%～10% 这个范围，不管哪种游戏，上线之后同时在线比例都会逐渐下降。在编写网络游戏的商业计划书时，注册用户数是一个基本指标，和推广成本息息相关，架构基础设施时，“注册用户数”和“同时在线数”都会影响系统负荷，所以都需要考虑。

用户的访问是随机的，此外排行榜和玩家匹配并不是实时的通信，所以可以使用公有云服务，负荷测试也在公有云服务上进行。

- 定义数据表
- 排行榜逻辑的实现（包含批处理部分）

- **使用 JMeter 的脚本，在云服务器上进行负荷测试，随机登录 500 万用户的分数**

  商业计划书中希望的注册用户数为 100 万，数据规模需要取一个几倍的安全幅度，这里按 5 倍进行计算。这个倍率可以根据测试预算的上限进行估算。比如可以大幅增加服务器台数，直到不能增加为止。

- **Web 服务器和 MySQL 服务器的负荷测试分别进行**

  测试结果的例子：排行榜每秒录入数据 100 次，浏览 200 次时 MySQL 服务器的 CPU 达到峰值，出现性能瓶颈，Web 服务器的负荷稳定在 20%。

- **500 万用户的批处理 1 次处理需要 1 分钟。250 万人需要 30 秒。人数和处理时间是等比例的。**

- **在访问的同时可以进行批处理，批处理进行时，最大的吞吐从 100 次录入数据降低到 60 次**

可以在短时间内验证上述结果。

预测玩家的游戏方式 ——排行榜的典型负荷测试步骤

之后是根据策划内容预测玩家的游戏方式。*J Multiplayer* 是 ARPG 类型的游戏，1 局游戏大概要花 10～30 分钟，一个在线玩家大约 10 分钟（600 秒）向排行榜发送数据一次或查看排行榜一次。假设每次游戏发送一次查看两次排行数据。

3 万同时在线玩家，除以 600 秒是 50 次 / 秒，即每秒发生 50 次访问。不过在午休或者傍晚休息时间访问比较集中时设定 6 倍的安全幅度，目标是每秒 300 次。

根据测试结果修改设计

根据测试结果，MySQL 服务器的 CPU 承受不了每秒 300 次的访问，这里有三个方案可供选择。

- **纵向扩展（scale up）**

- **横向扩展（scale out）**
- **排除扩展的必要性**

考虑到机器和磁盘的性能，仅仅通过纵向扩展让性能提升 6 倍是比较困难的。因为已经使用了比较快的服务器了。MySQL 的横向扩展有两种方式，❶ 统一数据读取和写入，并做备份，将操作分散到多个 MySQL 数据库。❷ 将数据水平分割。排行榜要求对“所有数据进行排序”，所以方法 ❷ 并不合适。方法 ❶ 的话，在写入分数时有处理负荷比较大的问题，高速读取数据也不能解决这个问题，所以应该采用方法 ❸ 更改处理内容，优化算法，从根本上提高处理性能，排除性能扩展的必要。这就是根据测试结果修改设计的要点。

具体的改进方式如前一章所述，“放弃完全正确的排名方法”。500 万用户中排名前 100 的玩家实时更新，其余的用户排名保留在 Web 服务器的内存中，偶尔进行一次数据更新。

例如，可以将所有用户的排名信息缓存在 Web 服务器，浏览排名时访问该缓存数据即可，写入数据（更新分数）也只用在 Web 服务器保留相关日志，每 24 小时进行一次批处理，在访问量少的时间段更新数据库。“不准确的排名和批处理”是排名系统经常采用的方法。

负荷测试每秒 100 次写入 200 次浏览时，Web 服务器的负荷是 20%，所以 1 台 Web 服务器就可以满足处理需求。游戏制作人要求尽可能节省预算，所以服务器配置如下。

- **Web 服务器 1 台（运算服务器）**
- **MySQL 服务器 1 台（存储服务器）**
- **MySQL 复制 / 备份服务器 1 台（存储服务器）**

都使用 Nifty Cloud 的服务器，所以没有什么问题。运算服务器每月 5 万日元，存储服务器每月 7 万日元，3 台服务器每个月大致预算为 19 万日元。

匹配系统也采用同样的步骤进行估算，应该也是 3 台服务器的构成。

在搭建 *J Multiplayer* 游戏的排名和匹配辅助系统时，服务器为 3+3 总共 6 台的结构。实际上，P2P MO 游戏的服务器构成大部分情况下要比这个少。

7.4 负荷测试

服务器搭建完成后，接下来在实际环境中进行负荷测试。本节分别说明 C/S MMO 和 P2P MO 游戏法负荷测试。

7.4.1 负荷测试的准备

基础设施架构已经完成，下面开始负荷测试，并同时进行调试工作。生产环境和开发环境有很大区别，一般会出现许多 bug。所以需要预留一定时间在生产环境进行测试。另外，由于服务器设定的变化会进行多次负荷测试，所以尽可能准备好相关工具，实现自动化测试。

接下来，我们来看一下实际的 *K Online* 和 *J Multiplayer* 的负荷测试。

7.4.2 *K Online* 在生产环境的负荷测试

理想的情况当然是模拟和实际生产环境一样的负荷测试。不过需要开发比较复杂的自动化测试程序。例如，模拟用户登录、击倒敌人、和 NPC 对话获得道具、存钱，仅仅是实现这些功能，优秀的程序员也需要花费数周时间。

对于网络游戏，特别是 C/S MMO 游戏，用户的活动往往不会像预测的那样按部就班地进行。对所有无法预测的地方都进行负荷测试性价比不高，程序员一直都有各种开发工作，所以应该尽量筛选重要的地方进行测试。筛选的方法如下。

- **如果因为超负荷导致当机，会对服务造成致命影响的部分。**
- **不容易横向扩展（scale out），发布后不容易修改的靠近后端的部分。**

- **从游戏策划角度看，和其他游戏区别度比较高、创新性强的部分。**

负荷测试的优先顺序可以参考上述方法。

K Online 的测试场景（Test Scenario）

K Online 的策划参考了 *Runescape* ，实际上没有创新的部分。所以负荷测试主要针对登录 / 退出 / 角色信息保存、单服最大同时在线数这几个方面。自动测试程序使用 C++ 实现，具体测试场景如下。

- **创建大量用户，在服务器上保存用户退出后的信息。**
- **用户登录。**
- **随机移动、发送消息 5 秒钟。**
- **用户退出。**

测试该场景时，在服务器的 10 个地方每处 100 人，总共进行 1000 次连接，测定 CPU 的使用率、各进程的负荷以及 MySQL 的负荷。

测试的分解

进行上述测试时，每次大概需要花费 10 秒，同时连接数为 1000 时平均每秒发生 100 次登录 / 退出操作。这种频率的操作对于 1 台游戏服务器来说负荷过高，可能会影响 MySQL 服务器的负荷测试。所以测试游戏服务器的最大同时在线数和后台服务器的负荷测试应该分开进行，具体应该分为以下两个阶段。

- **测试后台服务器负荷时使用多台游戏服务器。**
- **在 1 台游戏服务器上测试最大同时在线数。**

负荷测试需要的环境

进行负荷测试时，需要准备客户端机器。经过测定，预计需要 10 台 Windows 机器。

在负荷测试进行的同时，服务器一直运行着 vmstat、ps、top、netstat 命令和 /proc/interrupts，用来记录测试结果。此外还需要调整 MySQL 的“slow query”值，以便发现比较慢的查询。关于 MySQL 的性能优化，有很多专业书籍可以参考。

7.4.3 负荷测试时使用的服务器监控命令

下面介绍一些在负荷测试中常用的 Linux 命令，和他们各自的功能。

vmstat、/proc/interrupts

vmstat 是用来调查系统瓶颈的标准工具，在负荷测试中是最经常使用的。图 7.6 是正在进行负荷测试的一个实际例子。

图 7.6 在负荷测试中使用 vmstat

```
$ vmstat 2
 ↓ ①           ↓ ②               ↓ ③       ↓ ④   ↓ ⑤         ↓ ⑥
procs --------memory---------- -swap- ---io-- -system- ----cpu----
-
 r  b swpd  free  buff  cache  si  so  bi  bo   in  cs  us sy id wa
 0  b    0 25632 34640 819200  0  0   5  21  114  43   0  0 99  0
 0  0    0 25632 34648 819192  0  0   0  30  111 411   0  1 98  0
 1  0    0 25560 34668 819432  0  0   2 212  183 588  17  3 79  0
 1  0    0 25536 34720 819380  0  0   0 330  234 707  28  5 67  0
 2  0    0 25472 34768 819332  0  0   2 326  229 693  27  4 70  0
 1  0    0 25344 34828 819272  0  0   0 304  220 679  29  2 69  0
 1  0    0 25280 34876 819484  0  0   2 344  235 695  27  3 69  0
 1  0    0 25216 34968 819392  0  0   2 280  225 679  25  7 68  0
 0  0    0 25080 35064 819296  0  0   0 336  236 701  26  4 70  0
 0  0    0 24952 35152 819468  0  0   2 328  223 686  27  2 71  0
 0  0    0 24824 35244 819376  0  0   0 320  232 679  26  4 71  0
```

下面，我们来看一下图 7.6 的内容。图中第 ❶ 部分的 procs 表示进程状态。该列中“r”表示等待执行的进程，如果数值很大则表示 CPU 资源不足。如果处于等待的进程比处理器个数多了很多时，需要特别注意。图 7.6 的例子中 r 只有 1 个，处理器有两个内核，所以应该没有问题。

第 ❷ 部分的 memory 中，如果分配给服务器程序的内存是正常的话就没有问题。如果担心服务器程序的内存泄露问题，可以使用 vmstat -a 动态观察内存使用量的变化。

第 ❸ 部分是 swap，生产环境的服务器基本上不需要设定 Swap，当 si（swap in）、so（swap out）的值大于 0 时，表示每秒几兆字节至几十兆字节的数据交换导致系统内存不足，不能正常工作。

第 ❹ 部分是硬盘访问。bi（blocks in）表示读取，bo（blocks out）表示写入。该例子中每秒写入 300～500 block（300～500 千字节）。比硬盘最大速度（20 兆字节 / 秒）小很多，所以没有问题。

第 ❺ 部分 system 包含系统中断次数（in）和上下文切换次数（cs）。系统的时钟中断频率是 10ms 时，在没有负荷的情况下，系统中断次数应该接近 100 左右。通过网络发送大量数据包或者有大量磁盘访问时，这个值会增加到几千至几万。这个例子中通过网络加载的负荷并不大，所以中断次数在 200～300 之间。虽然前端服务器有大量的网络访问，但是后台服务器的中断值应该比较小。

vmstat 只显示了中断总次数，想知道具体内容的话可以查看 /proc/interrupts 的信息（图 7.7）。

图 7.7　确认 /proc/interrupts 的内容

```
$ cat /proc/interrupts
           CPU0
  0:    5146206   IO_APIC_edge  timer
  1:         11   IO_APIC_edge  i8042
  8:          1   IO_APIC_edge  rtc
  9:          0  IO_APIC_level  acpi
 12:         67   IO_APIC_edge  i8042
 15:     153452   IO_APIC_edge  ide1
177:      32568  IO_APIC_level  ioc0
185:     598735  IO_APIC_level  eth0
NMI:          0
LOC:    5138769
ERR:          0
MIS:          0
```

从图 7.7 可以看到 ide1 的磁盘访问和 eth0 的网络中断次数。根据服务器设置的不同，实际结果也不一样。

回到图 7.6，第 ❺ 部分的 cs 是上下文切换次数（context switch），每当进程分配到 CPU 资源后增加 1 次。切换上下文时，需要保存和切换 CPU 与操作系统管理的数据，每次需要花费几微秒的时间。如果这样每秒切换几千次，系统的整体开销会比较大。线程和进程过多也会引起频繁的休眠和启动，造成上下文切换次数增多。

比如，下面这个程序每隔 1 微秒休眠一次。

```
#include <unistd.h>
int main(){
  while(1){
    usleep(1);
  }
}
```

系统的时钟中断频率是每秒 100 次，所以启动一个该程序的话 cs 大概会增加 100 左右。100 个这样的程序同时运行时，cs 会超过 1 万。这种状态下可以从 CPU 统计（后面会介绍）中看到，us 基本为 0，但是 sys 的值已经接近 10%，内核中的处理开销明显增多。切换上述程序需要的资源比较少，所以处理比较快，如果是实际运行中的占用较多内存的程序，sys 会进一步增长。cs 超过 2000 就可能会引起一些问题。如果到达 1 万或 者 5 万，系统就可能会无法正常工作。不过这个和 CPU 的内核数相关，如果是 16 核的服务器，10 万也是可以承受的，所以需要根据服务器的实际情况调整。

最后是图 7.6 的第 ❻ 部分 CPU。这里的显示的是 CPU 整体的使用率。us 是用户时间，是服务器程序的进程内部使用的时间。sy 是系统时间，内核中使用的时间。cs 比较大时，内核中的处理会增加，这个比例会上升。另外，在磁盘访问相关的缓存处理、内存交换比较多时，网络访问等设备驱动的处理较多时，或者内核中数据拷贝量比较多时，sy 也会增加。id 是指空闲时间，表示 CPU 处于休息状态。Wa 是 io 等待时间。磁盘访问较多时，该值较大。

第 ❻ 部分 cpu 的理想状态是，id 值较大，us 较大的状态次之。sys 如果超过 20%～30% 这个范围，wa 值一直大于 10%～20%，就说明系统出现了问题。对于网络游戏的服务器来说，wa 应该尽可能为 0。特别是前端服务器，如果不是 0 就表示不正常。

ps

在 ps 命令后追加参数 ps auxww -L 执行的话，可以知道 MySQL、Apache 等多线程服务中每个线程的负荷。关键是“线程是否过多”。如果线程过多，vmstat 中 cs 值会较大。线程数量最大应该在可以利用的处理器内核的 2～4 倍左右。如果超过了这个范围，对系统性能就很难有贡献了。当然，这个也和应用的实现方式和架构设计有关。

ps 可以按照线程显示 CPU 的使用率。

top

top 显示的信息基本是其他命令可以获取信息的集合，可以让 vmstat 或者 ps 的结果更便于理解。用 top 查看系统运行状态非常方便，所以可以使用该命令监控负荷测试的对象服务器。

netstat

进行负荷测试时，可以使用 netstat 确认各个服务器之间的通信状态。图 7.8 是实际负荷测试时的例子。

图 7.8　负荷测试中使用 netstat 确认状态（片段）

```
         ↓ A    ↓ B
Proto Recv-Q Send-Q Local Address          Foreign Address
(state)
tcp        0   2309 10.16.113.103:12001  172.16.0.100:55361
ESTABLISHED
tcp        0      0 10.16.113.103:12001  172.16.0.100:55617
ESTABLISHED
tcp        0      0 10.16.113.103:12001  172.16.0.100:55105
ESTABLISHED
tcp        0      0 10.16.113.103:12001  172.16.0.100:55618
ESTABLISHED
tcp        0  10220 10.16.113.103:12001  172.16.0.100:55362
ESTABLISHED
```

```
tcp        0   8760 10.16.113.103:12001  172.16.0.100:56130
ESTABLISHED
tcp        0  10895 10.16.113.103:12001  172.16.0.100:55874
ESTABLISHED
tcp     4026  10618 10.16.113.103:12001  172.16.0.100:56386
ESTABLISHED
tcp        0   2817 10.16.113.103:12001  172.16.0.100:55106
ESTABLISHED
tcp        0      0 10.16.113.103:12001  172.16.0.100:55363
ESTABLISHED
tcp        0   7454 10.16.113.103:12001  172.16.0.100:55619
ESTABLISHED
tcp        0   3170 10.16.113.103:12001  172.16.0.100:55875
ESTABLISHED
tcp        0   6807 10.16.113.103:12001  172.16.0.100:56131
ESTABLISHED
```

实际的负荷测试有几百至几千行，图 7.8 只选取了 10 个会话。图 7.8 有许多 TCP 的会话，负荷测试时请注意 ❶ Recv-Q 和 ❷ Send-Q 两列。图 7.8 的例子中，Send-Q 大概是 10 千字节左右，预备发送的数据有所滞留。这是因为应用程序（游戏服务器的进程）调用了操作系统（Linux）接口，发出了传送数据的请求，但是通过 NIC 网络尚未发出。如果 Send-Q 到 100 千字节左右，说明服务器的瓶颈不是 CPU 而是网络，很可能是接收数据方的数据处理尚未完成。

负荷测试经常出现的问题是加载负荷的客户端机器的处理性能不足，当服务器端的 Send-Q 值较高，则很可能是客户端出现了性能问题。

客户端处理性能不足时，在客户端查看 netstat 的输出结果可以发现 Recv-Q 数值较大。

如果负荷测试中，Recv-Q 和 Send-Q 基本都没有数据阻塞，说明运行状况良好。

此外还可以使用 netstat -s，方便地确认 TCP 发送失败后再次发送的数据包是否有增加。比如，可以知道负荷测试中每秒发送 3 万个数据包，其中有 1 万的数据包需要再次发送。Windows 系统的网络数据包在高负荷时会有丢失的现象，所以在 Windows 客户端进行负荷测试时，或者服务器端程序是在 Windows 平台实现时，需要经常监测数据

包是否有大量数据包需要再次发送。TCP 再次发送数据包会造成通信性能的急剧下降，所以数据包的重发率最差也要控制在整体通信量的 1% 以下，如果不能保证接近 0 的重发率，就无法正常地进行负荷测试。如果达到 10%，那负荷测试的结果就不能成立了。

7.4.4 *J Multiplayer* 在生产环境下的负荷测试

与 *K Online* 相比，*J Multiplayer* 的负荷测试可以在更准确的估算基础上得到更接近生产环境的预测值。原因是，玩家实际的游戏内容本身并不需要在服务器实现，在服务器端只是实现了玩家匹配和排行榜等在其他游戏中也常见的基本功能。

所以，运行调用所有 API 的测试场景并测算时间，可以进行准确度很高的性能测试。

J Multiplayer 的测试场景

J Multiplayer 的测试场景可以根据玩家的典型行为定义下列操作。

- 登录验证
- 玩家匹配（设定可以确保通信成功的数据）
- 1 次取消然后再次进行玩家匹配
- 连接中继服务器
- 发送游戏数据包 1000 次（50 秒）
- 中止游戏通信
- 保存游戏结果
- 退出登录

上述操作在 1 分钟内进行，并模拟 3 万用户运行实现上述操作的测试程序。

负荷测试必要的测试环境

要进行测试，包括通信缓存量需要大概 500 千字节的内存，3 万用户同时运行时则需要 15 吉字节的内存。所以需要确保 3 台 8G 内存 8 核的 Linux 机器。每个内核同时处理 1500 个连接，每个内核每秒最多发送 3 万个数据包，如果性能不能满足需要，再增加机器。

负荷测试时服务器的监控和 *K Online* 完全一样。

* * *

通过以上描述我们可以看到，负荷测试需要大量的机器，并且有各种各样的测试前的准备工作。

7.5 游戏上线

游戏马上就要上线了！网络游戏上线后需要做的事情有很多。到底应该做哪些工作呢？本节会分别说明游戏上线前、上线后以及发生故障时的应对处理。

7.5.1 游戏上线前——从确认安全设定开始

游戏上线前和上线后的区别在于是否有玩家。有了玩家后可能也会混入恶意玩家，所以需要确认相关的安全设定。

安全设定包括两种，防止系统外部攻击的安全设定和系统内部和管理方法相关的安全设定。

防止系统外部攻击的安全设定

在 Alpha 测试或者封闭 Beta 测试阶段，一般是通过限定服务器端口号来确保安全。公开测试和正式上线后，因为可能受到各种预想之外的黑客（有的是职业黑客）的广范围的攻击，所以需要设定以下两点

- **数据包内容检测（固定字段）。**

- **每个 IP 地址的连接 / 带宽限制（使用路由器的防 DoS 攻击功能或者设定游戏服务器）。**

其他和网络系统相关的安全设置也都可以利用，请参考 SELinux ，iptables[14] 等专业书籍。当然，游戏服务器程序本身采用安全的方式实现是最重要的，具体方法请参考信息处理推进机构（IPA） 的相关资料。

> 14 使用数据包过滤。

和系统内部管理相关的安全设定

接下来是和系统内部管理相关的安全设定。例如，使用公有云服务时，远程连接通常需要使用 SSH 登录服务器系统，使用数据中心或者公司自己建立的服务器时，为了降低管理成本通常也采用同样的措施。

在 Alpha 测试和封闭 Beta 测试时，因为测试时间短、游戏没有收费、用户数量不多、只有少量服务器、开发人员应该优先改善程序等原因，所以基本上没有进行安全设定，但是游戏正式上线后，应该只对部分开发人员开放访问权限。这个时候最好准备一些辅助工具，一是为了方便管理者更容易获取运行日志和保存在 MySQL 中的数据，另一方面也可以让开发者从繁琐的工作中得到解脱。（当然，如果直接开放远程访问 MySQL 服务器的权限，就本末倒置了。）

和系统内部管理相关的安全设定也可以参考服务器系统管理的一些技巧。具体可以参考和服务器安全相关的专业书籍。

7.5.2　游戏上线后 ——系统监控

游戏正式上线后，和负荷测试一样需要监控系统的运行状态。C/S MMO 游戏很难进行充分的负荷测试，所以几乎一定会遇到下列问题。

1. 游戏服务器进程因为 bug 的原因当机

2. 游戏服务器进程在高负荷情况下无法访问

3. 运行游戏服务器端进程的服务器不能访问

4.游戏服务器因为 bug 的原因保存了异常数据

游戏上线后会不断涌入玩家，能否尽快解决上述问题，将在很大程度上决定游戏的成败。

应该做好准备，万一出现上述问题时可以及时处理。

1.游戏服务器进程因为 bug 的原因当机

→准备好自动启动服务器的脚本。后面会详细描述。

2.游戏服务器进程在高负荷情况下无法访问

→通过管理网络的 HTTP 远程连接监控工具检查游戏服务器的运行状态，如果没有响应或者发现服务器出现超负荷的情况，通过 SSH 登录服务器并 kill 相关进程。将这些操作做成自动化的脚本。

3.运行游戏服务器端进程的服务器不能访问

→Linux 服务器在进程数过多、使用的内存过多、发生 swap 时会出现这样的情况。游戏服务器程序如果不是实现方式有问题，一般不会因为进程数过多而不能访问，在程序实现中尽量避免动态内存分配来防止内存使用过多的问题。服务器一旦不能访问就要马上重新启动。使用 VM 时，可以使用远程访问重启服务器。其他情况下，尽可能准备好预备服务器，可以随时通过更改路由器的 NAT 设定进行服务器的快速切换。还可以更进一步只通过 NAT 的切换让运行中的服务器实现切换。很多路由器的 Web 管理界面可以仅仅通过点击就实现 NAT 切换功能。

4.游戏服务器因为 bug 的原因保存了异常数据

→应该迅速判断采用哪种修复方式：❶ 简单回滚（rollback）数据；❷ 运行问题解析脚本修复；❸ 手动执行 SQL 修复。为了防止保存异常数据，强烈推荐在向数据库发送数据前进行数据的验证（Validation）。数据验证包括：a. 枚举值是否在正常范围；b. 文字长度；c. 和之前数据的差分是否在正常范围。根据游戏的策划内容，很多数据可以通过简单的条件进行验证，所以应该仔细确认。在 Alpha 测试和封闭 Beta 测试中，应该采取严格的措施，如果出现数

据异常就马上终止游戏服务器（Assert）。这样就可以避免数据库中保留脏数据。一旦保存了异常数据可能会影响到很多玩家。在游戏服务器的 bug 改掉之前，就算回滚数据也还是会继续写入异常数据。一般游戏服务器的代码在前端更容易修改，产生异常数据时，通过 assert 终止程序，然后查看日志进行快速调试。如果使用 MySQL 的约束条件进行数据验证会让程序运行缓慢，所以一般在游戏服务器的程序中进行验证处理。

实际情况中需要进行 ❶ 的数据回滚处理，所以 Alpha 测试和封闭 Beta 测试中会设定时间，每 1～2 天就停止服务进行维护。

如上所述，游戏运营开始后会发生大量程序的问题，能否迅速判断优先级别，让程序员尽快进行修复是非常关键的。

7.5.3　服务器的组群化

正如本书前文所述，对于 P2P MO 类型的网络游戏，系统多则需要几十台，而 C/S MMO 类型则需要多组服务器供玩家选择，每组服务器由几十台服务器组成。如果可以对几十台服务器进行组群化管理，就可以大大提升性能的扩展性。超大型游戏往往架构多组服务器，50 台（总共 200～400 个内核）服务器为一组，同组内的机器不依赖于共同的后端服务器。

使用几万台服务器的大型 Web 企业也通常将几十台服务器划分为一组，并以这种方式进行性能扩展 [15] 。

> [15] 例如《Google クラウドの核心》（中文译名：Google 云平台的核心，Luiz André Barroso/Urs Hölzle 著，丸山不二夫、首藤一幸、浦本直彦、高岛优子、德弘太郎译，日经 BP，2010 年出版）书中所述，其中一个原因是 48 端口的以太网交换器比更多端口数的交换器价格低很多。基于机器故障率等经济角度的判断，除搜索引擎之外也能使用，是很值得参考的信息。

这里介绍一下笔者在几十台服务器规模的网络游戏中使用的简单的脚本。实际运用中会根据项目的需要进行调整，下面只介绍重要的部分。

首先来看一下服务器启动脚本。这里使用 SSH 从外部启动服务器。ssh 命令可以指定参数，不用远程登录服务器即可执行命令，数据中

心内部的服务器之间使用 ssh 时，没有必要重复输入密码，所以可以使用 ssh-add 命令设定 RSA 认证省略密码输入，或者使用 expect 命令自动输入密码。

- **服务器启动脚本（片段）**

```
#! /bin/csh  ← ①

while 1  ← ②
  echo rename logfile....
  mv gdb.log gdb.log.`date '+%Y_%m_%d_%H%M%S'`     ← ③
  echo cp server binary...
  cp gmsv gmsv_endless    ← ④
  echo start gdb...
  gdb -x autogdb.scr ./gmsv_endless  >>& gdb.log   ← ⑤
  gprof gmsv > log/prof.`date '+%Y_%m_%d'`    ← ⑥
  echo mailing..
  ruby gdbmailer.rb gdb.log   ← ⑦
  sleep 15      ← ⑧
  echo ENDLESS LOOPING...
end
```

❶ 使用的是 csh，csh 的特点是取巧的用法比较少。

❷ 使用 while 无限循环。不过执行脚本时可以给 csh 发送指令停止运行。

❸ 游戏服务器的运行日志输出到 gdb.log 文件中，并做好备份。

❹ 复制游戏服务器程序 gmsv，启动复制后的程序。这样可以在服务器程序运行时更新二进制文件。

❺ gdb（GDB）是 GCC 附带的命令行调试器，使用 autogdb 批处理命令文件后，当服务器异常终止时，可以自动生成错误报告（crash report）。标准输出信息会保存到 gdb.log 文件中。

❻ gmsv 的运行结果记录在日志中，文件保存在 log/ 目录。gmsv 正常结束时写入有效值。

❼ 将错误报告和日志文件发送到紧急邮件列表。

❽ 等待 15 秒。这个等待很重要。如果服务器因为某些原因没有启动成功时，如果没有休眠时间的话会发出大量的紧急邮件。有了这 15 秒的等待时间，1 小时之内就算一直是这种情况，最多也只会发送 240 封邮件。

道具的使用情况、登录、退出等普通的游戏操作日志，一般会通过网络发送到日志服务器。上述 ❼ gdb.log 仅仅是用来判断错误和调试。

autogdb.src 的代码如下。

- autogdb.scr

```
handle SIGPIPE nostop noprint  ← ①
run
where
list
quit
```

上述代码是 autogdb.scr 脚本的基本形式，根据调试的 bug 的内容，可以 print 全局变量值或者进行其他处理。

❶ 在使用了 socket 实现的服务器中接收到 SIGPIPE 也不能停止调试，所以不能通过信号终止。

发送邮件脚本的处理如下。

- 摘取 gdb.log 末尾 1000 行。
- 附加操作系统的运行状况（vmstat 的值、磁盘、netstat 的值、/var/log/messages 的末尾等信息）。

- 生成上述信息的文本文件并通过邮件服务器发送。

服务器进程相关的操作也需要准备自动化脚本，功能可以参考上述脚本，形式可以有所不同。

7.5.4　故障发生时的应对

开始运营后会同时发生多个故障。

应对同时发生的故障时，最重要的是怎么决定优先顺序。

几乎所有情况下，解决故障的速度瓶颈是程序员的时间，所以需要通过设定优先顺序、确定联系方式，从而最小化程序员的工作量或者需要处理的工作量。

优先顺序按照下列分类，并进一步细化。具体情况可以针对游戏策划内容进行调整，但应该尽可能简单、容易理解、层级少。

- 全部玩家不能进行游戏。
- 一部分玩家或者某些时间段无法游戏。
- 全部玩家可以进行游戏，但是全部玩家都有同样问题。
- 全部玩家可以进行游戏，但是一部分玩家有这样的问题。
- 性能上存在隐患。

另外，典型的应对方法如下。

- 等待。
- 重新启动游戏服务器、数据库管理系统、操作系统、服务器等。
- 更改设定文件（重新启动）。
- 向用户发布更新文件。

- 修改数据库内容。
- 修改程序。

在考虑优先级别和应对方法时，程序员之外的人如果可以使用工具完成下列工作，

- 搜索日志。
- 查看和修改数据库内容。
- 更改设定文件。
- 重新启动。

不但可以节约程序员大量时间，还可以形成同时应对多个故障的体制。

一开始就准备好所有的工具是不现实的，所以为了弄清除需要什么样的工具，在 Alpha 测试和封闭 Beta 测试阶段应该尽可能多考虑相关需求。

7.6　本章小结

本章针对网络游戏在实际运营中的重要的几点，介绍了基础设施相关的实际情况。希望大家充分认识到“进行阶段性的测试，并相应提升基础设施”的重要性。

第 8 章　网络游戏的开发体制：团队管理的挑战

网络游戏的开发团队管理并没有万能的方法。网络游戏的开发和运营属于脑力劳动的范围，作为管理的基础理论，可以参考彼得·德鲁克所著《管理：使命、责任、实务》一书[1]。

> [1] 《管理：使命、责任、实务》一书分为使命篇、责任篇、实务篇三册，由机械工业出版社分别于 2009~2012 年出版，译者王永贵。——译者注

笔者自己除了经营公司，还作为项目经理参与了几个游戏项目，但是从来没有感觉到自己管理团队很成功，每次项目中总会出现许多问题，老实说，我也不敢断言自己的方法一定有效。

但是，不论哪个项目，容易出现问题的地方总是相似的。本章的前半部分重点介绍网络游戏团队管理中特有的挑战。游戏策划是战斗、竞争还是协调，游戏结果是否公开等具体的策划内容，这些都会在很大程度上影响开发和运营的体制，所以需要格外注意。

本章的后半部分在探讨一般软件开发项目中经常遇到的挑战的同时，还为读者介绍了网络游戏开发团队的实际情况。

8.1　游戏的策划内容和开发团队 网络游戏特有的挑战

下面我们来看一下网络游戏的开发 / 管理体制中特有的挑战。这些挑战与游戏策划内容、游戏数据的持久化、玩家之间的关系等特殊因素有着紧密的联系。

8.1.1　游戏的策划内容是团队管理的关键

在网络游戏的开发 / 管理体制中，对于开发团队最重要的也是必须要解决的挑战是“游戏的策划”。策划也会影响开发团队的管理吗？这可能有些令人惊讶。当然，开发者的熟练程度、做事的方法、公司的经营方针等也会影响团队的管理，但最重要的还是“游戏的策划内容”。

网络游戏的开发团队面对的挑战大致可以归纳为以下几点。

- **游戏数据的持久化**
- **游戏结果的共享范围**
- **维护和升级的计划**
- **游戏中玩家之间的关系**
- **聊天系统的内容**
- **代码的规模**

8.1.2 游戏数据的持久化

C/S MMO 游戏的数据需要长期保存。大部分 FPS 类的 P2P MO 游戏，只在客户端保存数据，几个小时的游戏结束后就被重置。有些 P2P MO，比如卡牌游戏的卡片收集系统也需要长期保存数据。

说得极端一点，C/S MMO 这样的需要永久保存所有数据的游戏，从最初用户登录系统并开始保存数据（通常是封闭 Beta 测试开始时）起，才是项目真正意义上的开始，在此之前都属于“开店”前的准备阶段。准备期间就会面临的挑战是游戏系统能否承受压力。

运营刚开始的 3 个月最困难，运营可以持续 5～10 年

运营最初的 3 个月是工作最辛苦的时期。运营开始后 1 年内新增代码量往往远远超出运营前 1 年的代码量。而且，1 年的开发结束后，很多游戏的运营时间在 5～10 年左右，时间很长。

游戏的运营和技术人员

对工程师来说，5～10 年的时间可以拥有很强的专业技能。在游戏运营开始后，如果项目相关的核心技术人员不是真心喜欢游戏，不是那种在 5 年的长时间内参与同样工作也不会厌倦的人，运营就很难顺利进行。

但是在实际工作中，游戏运营后的主要开发内容是反复进行细微的修改，来满足玩家的需求。优秀的工程师一般都希望参与新的项目，而且对公司来说，新的项目也更需要这些技术人员。

MMO 游戏因为设计庞大而且相互之间的联系十分复杂，所以需要精通游戏策划和设计的人参与运营。如果经常更换熟悉系统的运营人员，就很难持续高效地运营游戏。所以，如何合理安排工程师的工作计划一直是一个比较难解决的问题。

实际项目管理中的挑战

实际的项目中面临的不得不解决的挑战有如下几点。

- 希望长期参与游戏项目的人是否也持有相关的技能。
- 如何合理使用在初期开发结束后想马上参与新项目的技术人员。
- 游戏开始运营后团队的休假计划如何调整。
- 项目奖金该如何设置。

8.1.3　游戏中玩家之间的关系

在游戏中，“玩家之间是什么关系”非常重要。游戏策划一般按照以下几种方式划分玩家之间的关系。

- VS（对抗）：玩家之间可以互相攻击，破坏。还可以互相 PK（Player Kill）。例如对战类、即时战略类（RTS）游戏，将对手全部消灭或者使其投降才算获得胜利。
- Compete（竞争）：以玩家之间互相竞争为基础的游戏，比如看谁先登上山顶、看谁速度更快的竞速游戏。

- CO-OP（合作）：以玩家合作为基础的游戏。如果不合作就无法取得进展，不存在胜负。
- Share（共享）：玩家之间没有直接的联系，但是可以共享少量数据。比如给好友送礼或者在玩家的游戏空间搞一些小破坏，FarmVille[2] 就属于这类游戏。这种游戏基本上单人也可以玩。
- Share nothing（无关）：仅仅作为一个分类，但是多人游戏的玩家之间应该不会没有任何关系吧。

> [2] Zynga 的农场经营类游戏。

在游戏运营和游戏规则需要谨慎考虑，此外游戏的平衡性也需要反复调整。

例如在对抗型游戏（VS）中，如果某种攻击方式有绝对性优势，或者玩家可以选择的种族的角色属性和实力比较强，或者看起来的不错的角色实际却很弱，客服系统肯定会受到大量投诉。不过对于同样的设定，如果游戏机制是合作（CO-OP）的方式而不是玩家之间竞争，平衡性也不会成为很严重的问题。一旦可以分出胜负，就会让一部分玩家不愉快，所以玩家还是期望不要出现不平衡的情况。在 bug 和系统故障的影响下，这种游戏的口碑在玩家社区之间会逐渐变差，再加上 2CH 论坛、Twitter 上各种对于开发团队的批评，会让运营团队的工作压力增大并产生大量加班的问题。

另一方面，数据持久程度越高，难度也会成几何倍数增长。如果胜败的结果并不是暂时性的，通过胜利获得的道具等也会影响之后的游戏的话，那么道具的出现方法和分配机制就必须反复推敲。所以，C/S MMO 类型的游戏中对抗型（VS）是最难开发和运营的，这种游戏对于开发团队的要求非常高。

在社交网络上提供的大部分社交游戏为了降低开发和运营成本，都尽量避免做“决定胜负”的对抗类型游戏，更多的是选择合作（CO-OP）和共享型（Share）的游戏策划，或者是在合作和共享的基础上增加一部分对抗（VS）或者竞争（Compete）的元素。实际上这种方式也更容易被玩家接受。但是存在喜欢对抗和竞争的硬核玩家可以提高一般用户的留存率，所以一般游戏策划都选择“基本关系是合作或者共享，对抗和竞争作为可选项”。

玩家之间的关系也受国民特点的影响，在不同市场会有很大的不同。

8.1.4　游戏结果的共享范围

游戏中获得的道具、创建的地图或者胜负的结果等玩家的游戏数据，应该展示给多少玩家？这个共享范围十分重要，共享范围的分类有以下几种。

❶ 只展示给自己。

❷ 可以逐个发送给别的玩家。

❸ 互相关注的好友。

❹ 展示给粉丝（关注自己的玩家）。

❺ 工会、粉丝团中的成员。

❻ 游戏服务器的范围。

❼ 不限定范围的全体玩家。

阐述上述共享范围对系统造成影响的必要性逐渐递增。对于游戏规则的公平性要求越高，就越需要谨慎对待游戏的升级和修改。比如 ❶ 的情况，因为信息只有自己知道，所以就算一部分玩家获得了有问题的道具，影响范围也不会很大。在 ❺ 的情况下，一旦道具给了工会其他人，就有可能影响工会全部成员。

❼ 是玩家的总排行榜的例子，如果某个玩家因为 bug 的原因获得了极高的分数，就会引起其他所有玩家的不满。

对于希望在更多玩家之间共享数据的游戏，更容易发生预想之外的问题，而且因为问题产生的影响会迅速扩大，所以必须尽快解决。如何建立迅速解决问题的机制并控制好预算是必须要考虑的问题。

具体的解决方案应该根据游戏的具体策划内容来决定。在不影响游戏性的基础上尽可能缩小游戏结果的共享范围的做法也是需要反复讨论的。

8.1.5　聊天系统的内容

虽然是细节性部分，但是聊天系统的形式还是十分重要的。

❶ 没有聊天

❷ 有固定句子

❸ 自由输入

这 3 种类型的开发和运营成本逐渐递增。

一般常见的问题是，获胜或者得利的一方遭到输家或者受损失玩家的谩骂，这也是玩家流失和玩家社区产生矛盾的原因。如果是系统允许❸ 自由输入，运营人员或者开发人员需要直接和游戏内的玩家通过聊天系统进行沟通，这个过程非常花时间。

但是另一方面，玩家之间可以交流一些有趣的事情，或者分享游戏攻略，对于游戏来说也是非常重要的（可以大幅降低运营成本），所以聊天系统还是十分必要的。聊天记录应该怎样保存、在什么范围进行传播、是否支持搜索、玩家禁言的规则等，在实现聊天系统时遇到的这些需求也会影响运营工作，也需要根据游戏的策划内容进行调整。

8.1.6　维护和升级的计划

系统维护和游戏内容的更新是每天都进行还是每周、隔周、每月或者是不定期进行，都会影响开发团队和运营团队的架构。

一般更新越频繁，玩家的留存率或者回头率越高，但是也会增加开发人员的多线工作强度，影响工作效率。

此外，因为更新游戏内容的开发和系统维护工作是不同的，理想情况是由不同的人负责，但是因为有工资预算有限、共享必要的技能和游

戏设定相关的知识比较困难等问题，一般很难做到。系统维护和游戏内容的更新都由相同的技术人员担任会增加多线工作的强度，从而影响工作效率，目前还没有比较好的解决办法。

8.1.7　代码规模——如果需要迭代的代码过多就会遇到问题

4.14 节提到 C/S MMO 的 gmsv（游戏服务器）的代码量比较大。*K Online* 游戏的商业化版本的游戏服务器端大概有 3 万～5 万行左右（不包括自动生成的代码），代码的规模对于开发和开发效率的影响很大。

不过也不是单纯的相关代码越多开发效率越低。比如，举个普通软件的例子，如果开发中使用了 GCC、MySQL 或者 Linux 等开源软件，这些软件加起来有几千万行的代码，但是这些软件十分成熟稳定，所以并不会对开发效率产生不良影响。

程序的开发效率一般是根据“策划→修改代码→编译→运行→测试→策划……”的迭代速度决定的。GCC 或者 MySQL 等子系统的迭代并不需要开发者参与所以不会影响开发效率（使用了不稳定的最新版产生问题的除外……）。

开发中的游戏代码也是一样，和迭代开发无关的部分，即使代码再多也不会影响开发效率。所以问题的关键不是减少代码的总量，而是缩短开发的迭代周期。

网络游戏开发中影响开发迭代周期的原因如下 [3] 。

> [3] 既有一般软件开发中的共同问题，也有不一样的。

编译时间

C++ 语言因为规范复杂，所以编译时间比较长。因为在实际开发中不得不使用 C++ 模版。

所以编译时必须采用 IncrediBuild 这样的分布式编译系统或者 make-j 等工具。除此之外更重要的是将程序的一部分分离出来，做成

程序库，并准备相应的单元测试，根据实际需要在程序中链接相应程序库。

分离出程序库后可以最大程度的降低需要迭代开发的部分。如果修改头文件后编译还需要花费 1 分钟以上的时间，说明还有代码能够分离到程序库。

程序启动时间

一般开发中的游戏的启动时间都会比较长。这是因为数据的读取比较花时间。使用开发中的 debug 版，需要读取几十兆字节至几百兆字节的数据，速度更慢。

如果启动时间在 30 秒以上，可能是开发专用模式的准备不够充分，可以关闭部分绘图功能或者设为最低画质。

测试的步骤数

网络游戏除了要启动游戏客户端，还需要启动服务器和数据库。如果启动步骤比较复杂，也会影响迭代开发的速度。

第 4 章提到过，服务器可以使用测试程序进行自动化测试，测试完成后再着手客户端程序的开发，如果服务器和客户端的开发可以分别进行就可以提高迭代的速度。服务器和客户端在同时开发时，如果采用自动再连接的方式则可以做到半自动化测试。

服务器程序的启动步骤

C/S MMO 游戏或者辅助系统一般都是多个服务器程序，如果服务器可以按照任意顺序启动也可以提高迭代的速度。其他一些比较常用的方法还有服务器自动连接机制、依赖的服务器当机后本机也自动重启的机制等。

数据验证

游戏中使用了大量数据。根据数据生成代码，并将信息保存在其中，再作为数据文件读取，在编译阶段是没有办法发现问题的，只有程序启动后在游戏的过程中才能碰到。

使用从 CSV 数据文件等生成大量代码的工具时，编译后的代码作为程序库，并准备好单元测试，通过调用函数来验证数据完整性，这样可以在早期就发现数据的输入错误，提升迭代开发的效率。

此外，还可以通过自动验证机制对游戏服务器程序保存在内存中的游戏数据状态进行验证，同样也可以加快迭代速度。

对于 DB 中保存的数据也可以采用同样的方法。将代码或者数据的测试加入到每天的自动编译过程中，随时都能为数据创建者提供可以运行的程序，这样就可以大大降低程序员手中的数据出错的几率，提升迭代开发的速度。

8.2　网络游戏开发团队的实际情况——和一般软件开发相同的地方

本节主要介绍和一般软件开发大体相同的开发团队结构和挑战，并同时说明网络游戏开发团队的实际情况。

8.2.1　工作分配

两个开发者以上时，并没有固定的工作分配方法。现在，各公司内，除了开发者们擅长和不擅长的技术和经验之外，由于

- 全职还是兼职。
- 在同样的办公地点还是远程办公。
- 专门负责还是同时负责多个工作。
- 掌握的语言（日语、中文等）。
- 在项目开始时加入，还是中途加入。

等不同，非技术部分也开始变得非常多样化。兼职、远程办公、同时负责多个项目、语言不同、中途加入项目会让协同工作的难度加大，所以不管在什么项目中，都迫切地需要找到解决这个问题的方法。

为了解决这个问题，需要慎重考虑如何在实际的开发中合理分配工作。从方法论上来说有以下两种方式。

- **将客户端和服务器端的开发工作分开　　　　→横向分配**
- **根据游戏策划内容分配（如战斗、赌博、钓鱼、BOSS 战等）→垂直分配**

并没有标准的做法，这里仅仅作为一种参考。游戏业界一般把开发中必要的基础部分，即“系统部分”的工作分配称为“横向分配”，在基础系统之上的“游戏策划部分”的工作分配称为“垂直分配”（如图 8.1）。

图 8.1　一般的工作分配

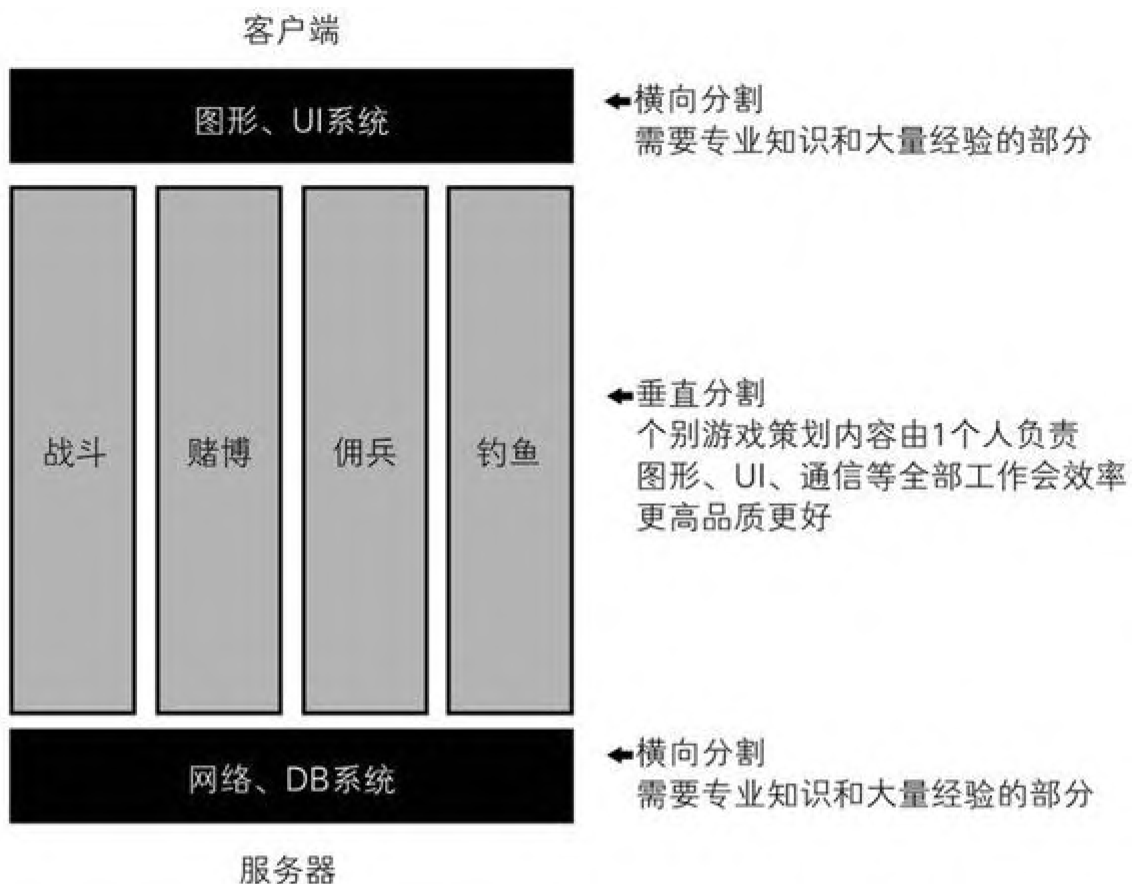


图 8.1 中，基础系统部分（网络、数据库系统）直接影响到游戏整体的性能和开发效率，所以需要由经验丰富和拥有相关专业技能的工程

师负责。虽然现在看来应该这样做，但以前红白机的时代，所有工作都按照游戏策划进行垂直分配的情况并不少见。

比如，到了射击游戏的 Boss 战时，绘图系统也切换到了负责 Boss 战的工程师制作的版本。

目前，小型的 Flash 应用中也还会采用同样的方式分配工作（在垂直分配后的 swf 文件之间进行切换）。

不过程序达到一定规模后，从经济角度考虑，网络游戏并不适合采用这样的垂直分配方式。

8.2.2　持续提升网络游戏程序员技能的方法

对开发团队来说，提升工程师的技能水平非常重要。网络游戏的开发周期一般在半年至 1 年左右，大型的 C/S MMO 游戏甚至会花费 3 年以上的时间。而且运营开始后，往往还需要 3～5 年的版本升级工作。在这个期间，如果更换对系统比较了解的工程师，会对游戏更新造成影响，所以从项目角度还是希望尽量由相同的工程师继续负责维护和升级工作。但是从另一方面来说，新项目的初期更需要设计和系统架构的技术，作为技术人员从提升技能的角度也更希望参与新的项目，这也是一个问题。

另外，还经常出现因为初学者被突然分配到商业项目中，在长期工作中写了大量低质量的代码，拖了项目的后腿。通常项目都没有时间从头培养初学者，这也是一个经常遇到的问题。

从武术的“守、破、离”中学习提升技能的方法

每个阶段都有不同的提升技能的方法，现实中并没有万能的做法。这里以笔者的经验介绍一些有效果的方法。

技能的提升分为 3 个阶段。3 个阶段的划分方式有很多种，这里按照武术中常用的“守、破、离”划分。“守”是说初学者按照标准做法去做，“破”是说在标准做法基础上进行改进，“离”是指自己进行创新。

“守”的阶段 ——从模仿开始

首先是“守”的阶段，从模仿开始，这是最快的学习途径。可以先参考别人的游戏。

就本书来说，*K Online* 和 *J Multiplayer* 分别模仿了 *Runescape* 和 *Diablo* ，可以先模仿现有的游戏，并进一步根据本书介绍的示例代码进行开发。按照这种方式学习时需要注意两点：❶ 要模仿经典作品而不是非主流的游戏。❷ 要使用标准的、争议较少的程序库。如果使用最新的技术，因为要同时学习很多东西会让学习难度加大，影响效果。举个例子，如果直接用英语教材学习数学肯定会比先用日语教材然后再用英语教材学习的难度大。

初学者采用这种“模仿现有游戏”的方式时，如果有可以随时请教的资深工程师的帮助，一般 1～2 个月就能入门。而且现在开发环境都很方便，大大缩短了学习周期。

服务器端的环境可以租用 Linux 的 VPS 服务器 [4] ，安装 Ruby 和 MySQL 等工具后就可以搭建自学的环境。此外，在 Twitter 上关注那些 iPhone 开发者或者 Web 开发者列表，就能够通过网络随时向资深的游戏开发者提问。有了这些环境就可以快速的学习相关技术。现在开发游戏时缺乏美术素材的问题也有办法可以解决了，在 Google 上搜索“nes sprite”可以找到很多可以使用的美术素材 [5] 。

> [4] 本书截稿时（2010 年 12 月），Sakura Internet 的 Sakura VPS 试用期免费，之后每个月收费 980 日元。（http://vps.sakura.ad.jp/ ）

> [5] 不过使用时需要注意版权问题。

另外，开发者还可以利用 Facebook 等社交网络的服务，收集玩家的反馈意见，甚至可以进行商业化尝试。这对开发者也是比较有利的一点。

“破”的阶段 ——研讨会、技术会议

初学者阶段结束后进入“破”的阶段，可以参加一些技术沙龙、会议，把听到的讲座转化为自己的理解。同时也是很好的机会，可以了解自己和业界的资深技术人员之间的技术差距。

不过，日本国内游戏开发的高水平会议只有 CEDEC，网络游戏的技术会议非常少。不过最近以网页游戏为主的技术会议逐渐增多，大家可以在网络上搜索相关信息。

国外有大量诸如 GDC（Game Developers Conference） 这样的高水平技术会议，仅仅是网络游戏相关的会议就多到一个人无法消化的数量。笔者基本两年参加一次。

除了技术会议，笔者也经常被问到在中级“破”的阶段应该怎样提升技术水平的问题。在“破”的阶段，我推荐大家多参与不同技术领域的游戏开发，参与到网络游戏和 Web 业界的开发中去，比如可以开发网页游戏、手机游戏，还可以参与教育游戏的项目、加入硬件开发团队、参与数据挖掘的项目、参与和游戏开发并不完全相关但是有关系（很可能会用到游戏开发中）的其他项目。

和游戏关系密切并且十分热门的相关技术有 Web、WebGL 、NoSQL、P2P、Android 和 LL 社区等。在 Twitter 的列表或者标签中搜索并关注那些活跃的同行可以很快扩大自己的视野。然后可以在短时间内通过“模仿”的方式，快速尝试自己专业之外的技术。

在“破”的阶段提升技术，需要将一半的时间用在边缘领域，和之前没有共事过的人一起学习提高会取得更好的效果。

“离”的阶段 ——应该怎样进入工程师的最高级阶段

进入到“离”的阶段的话……笔者自己也还没达到这样的水平，所以没有办法说明，还没有到带领团队开发全新游戏的阶段，不过希望能早日进入这样的状态。

8.2.3　项目管理术——游戏开发和 Scrum

最近 3～4 年，GDC、CEDEC 等技术会议中将 Scrum 引入游戏开发的议题越来越多。从参加研讨会的人那里经常听到“游戏开发一开始就采用敏捷方式，正心想学习 Scrum 还有必要吗，但直到听了实际运用中的做法后，才发现和游戏开发中所做的完全不同”这样的话。这到底是怎么回事？

大部分游戏开发者认为“游戏开发不运行起来是不知道具体情况的”，所以习惯了不断修改的过程。最初听到“Scrum 要求频繁的发布”后就觉得“现在做的不就是多次修改每周都发布”嘛。

不过笔者认为 Scrum 的重点不在于频繁发布可运行的程序。更重要的是测试发布的程序是否符合要求，然后根据结果调整工作的优先顺序，以此循环。

截至目前笔者所接触过的网络游戏开发团队中，有个负责的项目最多的制作人因为不同项目的工作很多，无法按计划测试发布版本，工作的优先顺序也很久没有更新，导致在不重要的工作上浪费了大量精力，还很长时间没有和客户确认工作的优先顺序表，所以开发出来的产品和客户的期待有很大差距。结果，在规定的时间内无法交付客户期待的成果。

在游戏开发中，开发团队外围的环境相对分散和多样化，比如发行商无法每天和开发团队会面、制作人和程序员在不同场所办公等，如果可以引入 Scrum 那样的发布、测试和反馈机制，就可以大大提高项目成功的概率。特别是现在很多网页游戏的开发团队一般只有 3～5 个程序员，非常适合采用 Scrum 的敏捷方式，这也是一个趋势。

8.2.4　开发环境的选择

兼职、远程办公、同时负责多个项目、外语交流、中途参与项目……开发团队的构成越来越多样化，在开发环境中应该使用什么团队协作工具呢？

- **代码和数据可以带到公司外吗**
- **可以从公司外部访问内部的 Wiki 或者 Trac 吗**
- **可以从家里访问服务器环境吗**

一般会使用 VPN 从外部网络远程访问开发环境，这样可以避免拷贝所有的代码和数据，是一种基本的解决方案。

如果可以通过 VPN 远程访问就能直接拷贝源代码，所以相互信任是非常重要的。特别是和其他公司员工协同工作时，因为相互利益不同，

这是一个比较严重的问题。

笔者的经验是，不管是公司还是个人，要选择值得信赖的并有过相关经历的工程师，签署合同后再开放 VPN 的访问权限。不过在两家以上公司的情况下，如果项目开始后代码和 Wiki 无法完全统一管理，项目失败的可能性极大。因为很难做到“保证随时可以发布产品”。

特别是项目开发初期，“完整的规格文档还没写好，程序员需要根据自己的判断来开发”等因素的影响，再加上如果程序员不在一起办公，问题会更严重。因为不同的情况有不同的解决办法，所以对于这类问题，需要在项目开始前就做好充分的应对准备，这一点十分重要。

8.2.5　项目的移交——理所当然的事情也需要仔细归纳总结

将网络游戏开发项目移交给其他公司或者其他团队一直是一个比较令人头疼的问题。仅仅在日本，每年游戏业界就有几十件项目移交的工作，不管对于什么游戏，在开发中都需要对未来可能发生的移交工作做好准备。

开发项目的移交是指从目前负责开发的团队或个人手中将项目移交给其他团队或个人，而且目前负责的人将不再参与项目。通过授权而将技术转移给国外运营公司，因为负责项目的人还继续参与项目所以不属于移交的范畴。

项目的移交往往伴随着公司的改制、重组、组织的改变和设置等，所以没有充裕的时间允许工程师进行几周的结对编程。在没有充分的计划、没有足够的时间的前提下，又必须尽快完成。那么在移交工作中需要注意那些环节呢？

测试的准备

首先是测试的准备，这个并不局限于游戏。正如本书之前所述，在实际移交之前，首先更新游戏的自动化测试工具。如果可能的话，在自动测试客户端工具中要添加尽可能多的操作。这个准备工作可以由目前的团队独立完成。

缩短搭建开发环境的时间

接下来，一个重要的工作就是尽可能缩短搭建开发环境的时间。操作系统的安装时间、必要程序库和开发工具的安装时间都要尽可能缩短。

如果是 Windows 系统，最好可以做到只需要从 DVD 光盘拷贝文件。如果 Linux 系统，从 VPS 租借服务器后使用 yum 安装标准程序包，然后解压缩 tar 文件、make、接着 make test 就可以完成全部工作。

尽量使用 CenOS 中自带的工具版本，比如 Ruby 或者其他商业程序库。使用 Web 浏览器时，要保证默认设定可以运行。最好可以在一台机器上完成所有设置。如果步骤中需要先设定 3 台机器然后再设定 2 台服务器的话，移交工作就会变得更加困难。接受移交一方的工程师大多在最开始还同时负责其他的项目，所以如果开发环境的架构比较花时间的话会极大的增加移交的难度。

设定合适的访问权限

最后为了迅速的完成交接工作，在项目交接过程中应当“逐渐”开放最大访问权限。为了访问数据文件和代码，在交接时需要打破组织关系造成的障碍。也不是说让每个人都有权限，只要在系统空闲时开放权限让参与项目交接人员可以访问即可。

8.3　本章小结

本章为大家深入介绍了网络游戏的开发团队，并列举了一些开发中棘手的问题。每个问题都没有标准的解决方案，都需要开发团队内部进行探讨，本书仅仅介绍了笔者的一些亲身经验。

至此，本书的内容就告一段落。本书从网络编程和游戏编程的快速入门开始，介绍了网络游戏的历史、需求的理解、架构、C/S MMO、P2P MO、辅助系统、基础设施和开发体制等许多话题，并通过案例游戏的开发，说明了网络游戏整体的开发思路。

贯穿本书的最重要的一点就是“要根据游戏策划内容来决定应该采用的技术”。换句话说，就是在网络游戏的开发中，想实现的游戏内容会影响到开发需要的技术、硬件设备、团队体制、预算规模等方面。

如果更多的开发者能够理解这一点，就一定能够开发出更多的受欢迎的有趣的游戏。非常期待这一天的到来。

专栏　网络游戏开发的成本

开发项目有常识性的日程安排和成本。不过日程安排和成本是不能划等号的，当然，同时参与项目的人越多，即使工期时间短，成本也会非常高。

程序开发中增加人数并不会让工作效率呈线性增加，不过美术方面的工作可以通过增加人数来提高开发速度。因为美术工作可以按照“100个道具的图像”来量化规模，所以可以按照美工人数来分配。

正如本章所述，开发成本和运营成本可以分别计算。在这里我们分别介绍项目中程序部分和美术部分的开发成本。根据游戏内容的不同，美术成本上下浮动很大，基本没有什么意义，这里只作为参考。调试的成本占总成本的 1～2 成，虽然在实际开发中也是不可忽略的部分，但限于篇幅在此不再介绍。下面介绍的成本可以作为一般性的指标，在具体开发中，根据项目的不同，可能会有很大不同，需多加注意。

根据游戏类型来划分，开发成本大致如下。

- 程序的开发成本
- C/S MMO
    - PC 原生游戏：40～100 人月
    - Flash：20～40 人月
    - 手机游戏：20～40 人月
- P2P MO

    - PC 原生游戏：15～50 人月
    - Flash：3～30 人月
    - 手机游戏：2～20 人月
    - Web 社交游戏：2～20 人月
- 美术成本
- 大型 MMORPG
    - 3D：20～400 人月
    - 2D 像素：30～300 人月
- 简单的 MMOG
    - 3D：10～100 人月
    - 2D 像素：20～100 人月
- P2P MO
    - 3D：10～200 人月
    - 2D 像素：20～200 人月
    - Web 社交游戏：1～10 人月
- 运营成本
- C/S MMO：每月成本为初期项目每月开发成本的 50%～100%
- P2P MO：每月成本为初期项目每月开发成本的 5%～20%
- Web 社交游戏：每月成本为初期项目每月开发成本的 100%～200%

如果是 AAA 游戏，大概需要在上述成本上再加一位数。比如跨平台游戏、在多个国家或者全球，同时发售多语言版本。这种情况可以使用 Unreal Engine 级别的昂贵的中间件，开发人数可能会超过 100 人。

开发工期大致如下。

- C/S MMO
  - PC 原生游戏：1～3 年
  - Flash：半年～1 年
  - 手机游戏：3 个月～1 年
- P2P MO
  - PC 原生游戏：半年～1 年
  - Flash：3 个月～1 年
  - 手机游戏：3 个月～1 年
  - Web 社交游戏：2～6 个月

上文列举了使用 HTML 技术的 Web 社交游戏的开发成本作为参考。相对于初期开发成本，社交游戏的运营成本一般会比较高。

如果开发团队给出的估算和上述数据有较大出入，则需要调查具体的原因。如果游戏中有“想要展现不同效果”的地方，可能也会得出应该积极投资的判断。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks

图灵社区会员 ptpress（libowen@ptpress.com.cn） 专享 尊重版权