CHARLES RIVER MEDIA

源代码示例

大型多人在线游戏开发

MASSIVELY MULTIPLAYER GAME DEVELOPMENT

〔美〕Thor Alexander 编
史晓明 译

◆ 由入选美国互动艺术科学协会(AIAS)名人堂的游戏大师、卓越的电脑游戏先驱者、创世纪(Ultima)和网络创世纪(Ultima Online)之父，“不列颠之王”(Lord British)——理查德·加利奥特(Richard Garriott)挥毫作序。

◆ MMP开发精英倾情巨献，深入剖析如何在MMP游戏开发中避免冗余，应对常见的缺陷和玩家侵入等问题，从而节约宝贵的编程时间，实现高效开发。

◆ 内容覆盖MMP开发过程中各个主要领域，包括设计、架构、服务端和客户端开发、数据库技术以及游戏系统。

人民邮电出版社
POSTS & TELECOM PRESS

译者介绍

史晓明，澳大利亚 BigWorld Pty . Ltd（全球领先的大型多人在线游戏引擎开发商）的客户端工具程序员，主要从事游戏世界编辑器以及三维软件插件方面的工作。此前于上海育碧公司工作七年，先后参与过《雷曼 2》PS 版、《VIP》PS2/GC 版以及《分裂细胞（Splinter Cell）》I/II/IV 等游戏的开发工作，在此期间还汉化了《魔法门》、《辐射：钢铁兄弟会》、《家园》、《生化危机》等数十款游戏。

专家推荐

“本书阐述了 MMO（Massively Multiplayer Online，大型多人在线）网络游戏架构、设计、制作中的一些关键点，是有志于 MMO 网络游戏开发的朋友不可多得的一本好书。”

——李洪涛

第九城市主程序员 GA 游戏教育基地游戏程序高级讲师

“本书适合有一定编程基础的游戏开发人员进一步了解网络游戏体系以及从抽象角度来观察网络游戏开发。本书并未陷入繁长的代码细节，而是从宏观角度剖析了网络游戏设计，构架、设计、服务端开发，数据库技术以及网络游戏核心系统等网络游戏的知识。设计内容全面，讲述细致。是一本网络游戏开发人员不可多得的参考书。”

——田菲

GA 游戏教育基地游戏程序高级讲师（职业经历：EA、光通）

内容要点

- MMP 设计技术：提供了对设计过程的概括描述，包含《卡通城在线》开发团队在开发中所获得的重要经验，也提供了《创世纪在线》团队在在线客户支持中遇到的实际问题。
- MMP 架构：介绍了怎样应用面向对象技术和极限编程思想来创建稳健的 MMP 框架和架构。
- 服务端开发：详述在线游戏服务端的开发，包括无缝服务器的优缺点、开发和维护中常见的问题以及向无线设备移植的技巧。
- 客户端开发：阐释移动预测、角色定制以及在家用游戏机上开发 MMP 中所面临的问题。
- 数据库技术：介绍了数据库设计的基础以及如何在在线游戏中使用数据库。
- 游戏系统：介绍了怎样为 MMP 游戏设计游戏系统。

封面设计：胡平利

ISBN 7-115-15267-5

9 787115 152671 >

人民邮电出版社网址 www.ptpress.com.cn

CHARLES RIVER MEDIA

本书是一部系统介绍 MMP（大型多人，Massively Multiplayer）游戏开发知识的文集，汇集业内游戏开发精英们的智慧精华，提供了大量应用于 MMP 游戏开发工作中的独特技术。本书不仅从MMP游戏的角度对客户端技术进行了讨论，还深入剖析了MMP游戏设计、架构、服务端开发、数据库技术以及MMP游戏核心系统等知识。对于广大MMP游戏开发人员来说，本书是不可多得的参考资料。

和其他很多编程书籍不同，《大型多人在线游戏开发》是针对整个开发团队的：程序员可以获得大量的技术思想，游戏设计人员和制作人也可以从关于设计、架构和客户支持的详细信息中获益。

分类建议：计算机 / 程序设计 / 游戏开发

ISBN7-115-15267-5/TP • 5689

定价：59.00 元（附光盘）

大型多人在线游戏开发

[美] Thor Alexander 编

史晓明 译

人民邮电出版社

图书在版编目（CIP）数据

大型多人在线游戏开发 /（美）亚历山大（Alexander, T.）编；史晓明译.
—北京：人民邮电出版社，2006.12

ISBN 7-115-15267-5

Ⅰ. 大… Ⅱ. ①亚…②史… Ⅲ. 计算机网络—游戏—应用程序—程序设计—文集
Ⅳ. ①G899-53②TP311.5-53

中国版本图书馆 CIP 数据核字（2006）第 107533 号

981-XXX-XXX-X

Thomson Learning (A division of Thomson Asia Pte Ltd) , 5 Shenton Way, # 01-01 UIC Building Singapore 068808

大型多人在线游戏开发

◆ 编 [美] Thor Alexander
译 史晓明
责任编辑 王 琳

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京密云春雷印刷厂印刷
新华书店总店北京发行所经销

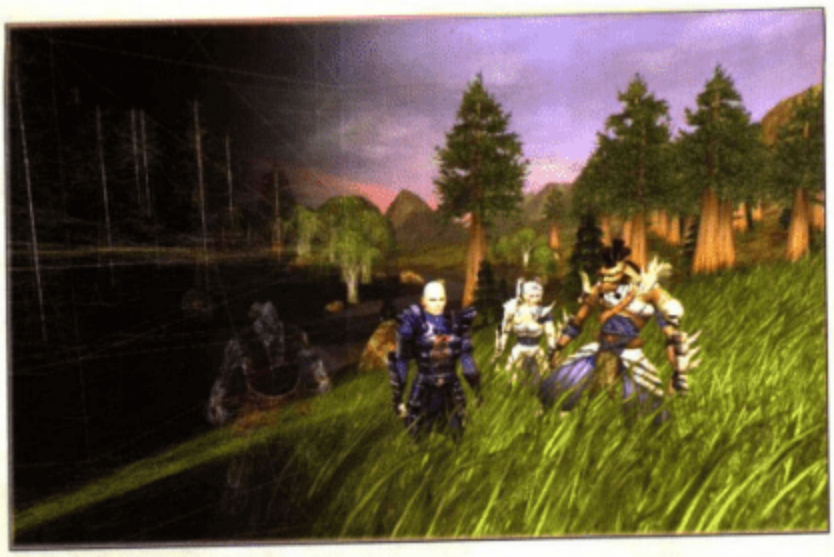
◆ 开本：787×1092 1/16
印张：26 彩插：2
字数：627 千字 2006 年 12 月第 1 版
印数：1－4 000 册 2006 年 12 月北京第 1 次印刷

著作权合同登记号 图字：01-2004-4426 号

ISBN 7-115-15267-5/TP・5689

定价：59.00 元（附光盘）

读者服务热线：(010)67132705 印装质量热线：(010)67129223



彩图1 《亚瑟龙的呼唤2：堕落之王》（*Asheron's Call 2: Fallen Kings*）中的场景图

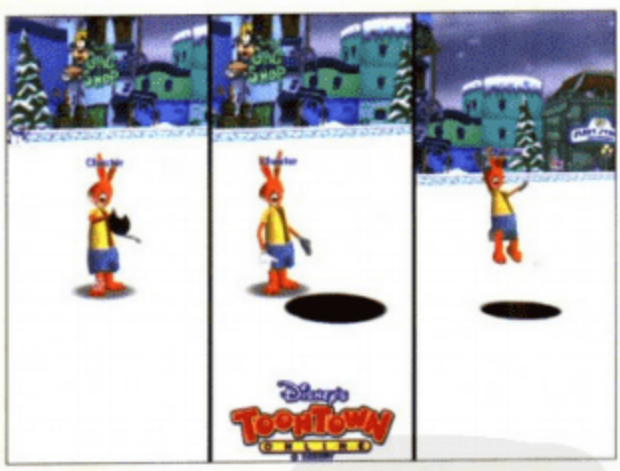
彩图2 在《亚瑟龙的呼唤2：堕落之王》中，很多游戏模式的特性都可以在运行时通过与图中类似的方式进行调整

彩图3 无缝服务器可以让玩家在不同区域间迅速平滑地切换

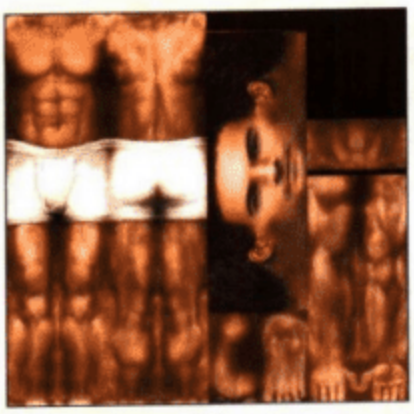
彩图4 《卡通城在线》中的中心游乐场
本彩图由《卡通城在线》提供

彩图5 《卡通城在线》中的角色创建和定制功能非常的直观，对用户很友好
本彩图由《卡通城在线》提供

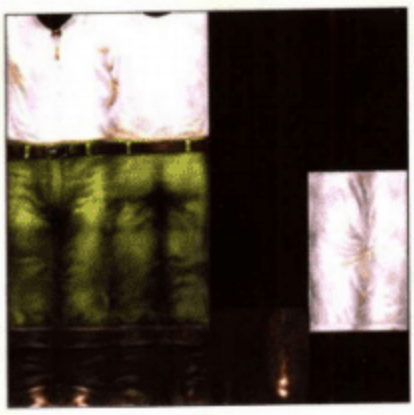
彩图6 《卡通城在线》中的迅速聊天（Speedchat）功能让游戏中的交流变得更加安全和快捷，并且不会含有不礼貌的信息
本彩图由《卡通城在线》提供



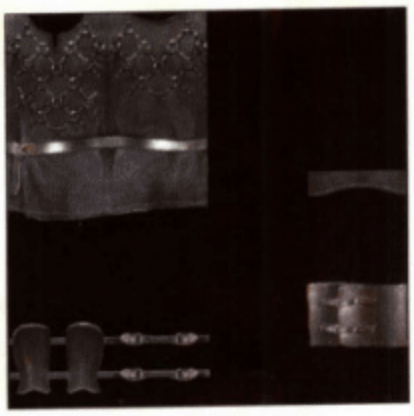
彩图7 《卡通城在线》中的瞬间移动功能使用过漫画风格的“便携洞”
本彩图由《卡通城在线》提供



(8)



(9)

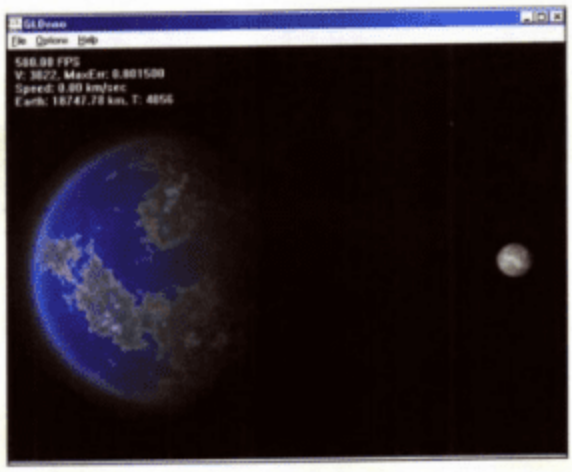


(10)

彩图8、9、10 基本贴图：（8）人的皮肤贴图（9）衣服贴图（10）盔甲贴图
这些彩图由Todd Hayer和Dale Homburg提供

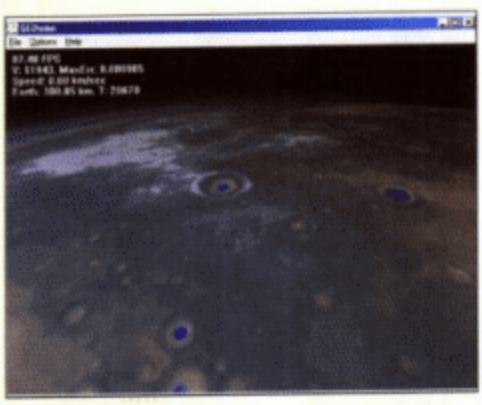
彩图11 可以把基本贴图按照不同的层次叠加起来以定制角色的外观
本彩图由Todd Hayer和Dale Homburg提供

彩图12 通过对贴图进行调色，少量的贴图层次组合起来可以生成大量不同的外观
本彩图由Todd Hayer和Dale Homburg提供

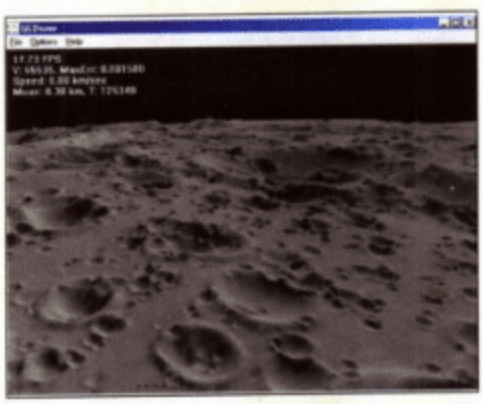


彩图13 逐步接近一个和地球类似的行星以及它的卫星。这个行星模型是在运行时使用球体实时优化自适应网格技术（spherical ROAM）动态生成的。当以图中距离观察这个行星时，我们使用替身技术（impostor）来渲染以提高性能
本彩图由Sean O' Neil提供并授权使用

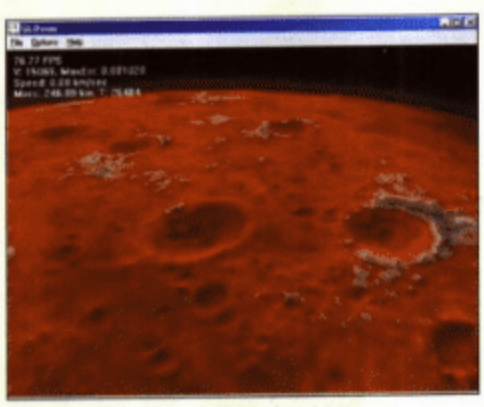
(14)



(15)



(16)



彩图14、15、16 （14）从轨道上观看一个具有不少陨石坑、类似于地球的行星。行星的高度图是使用基于fBm的分型算法在运行时生成的。行星的贴图可以为每个顶点提供高度值。陨石坑是在运行时使用虚拟四叉树添加的。（15）从很近的距离观看一个有着亿万个陨石坑的卫星。这些陨石坑使用虚拟四叉树在运行时生成，它不仅不需要使用任何内存，还可以实时生成。我们可以把陨石坑图形修改为任意我们想要添加到地形表面的图形。（16）从轨道上观看一个具有不少陨石坑的、类似于火星的行星。除了使用不同的贴图以及更多的陨石坑以外，它和前面类似于地球的行星相似。
这些彩图由Sean O' Neil提供并授权使用

内容提要

本书是一本系统介绍 MMP（大型多人，Massively Multiplayer）在线游戏开发知识的文集，汇集业内最优秀的游戏开发人员智慧的精华。本书不仅从 MMP 游戏的角度对客户端技术进行了讨论，还深入剖析了 MMP 游戏设计、架构、服务端开发、数据库技术以及 MMP 游戏核心系统等特定于 MMP 游戏的知识。对于广大 MMP 游戏开发人员来说，本书是不可多得的参考资料。

译 者 序

MMP 游戏的开发相对于单机游戏来说要复杂很多，开发人员不仅需要面对单机游戏开发中常见的图形、物理、碰撞、音效等方面的问题，还需要设计更为复杂的游戏系统，处理网络和服务端架构，搭建数据库平台，并且在此同时保持更高的稳定性。这意味着市场上大部分针对单机游戏的书籍中所介绍的知识对于 MMP 开发人员来说是远远不能满足需求的。

《大型多人在线游戏开发》是一本系统介绍 MMP 游戏开发知识的文集。本书中，那些业内优秀的游戏开发人员深入浅出地对 MMP 游戏开发的各个方面进行了详细介绍。它不仅从 MMP 游戏的角度对客户端技术进行了讨论，还深入剖析了 MMP 游戏设计、架构、服务端技术、数据库技术以及 MMP 游戏系统等 MMP 游戏的知识。因此，对于广大 MMP 开发人员来说，这都是一本不可多得的参考资料。

中国已成为最大的 MMP 市场之一，MMP 游戏的运营商收入和玩家人数等指标近几年一直以很高的速度增长着。然而，在 MMP 开发方面，我们还不是一个开发大国，大量的游戏主要通过引进，即使一些有能力进行本土开发的厂商也并不拥有所有的核心技术。在本书的翻译过程中，我自己也从一个单机游戏开发人员转变为一个 MMP 引擎开发人员。我也希望通过本书的翻译出版可以帮助更多的业界人士进入 MMP 开发队伍，从而为中国成为 MMP 游戏开发大国贡献力量。

在翻译本书时，我力求贴近原意。对于原著中难以理解或是有疑问的地方，和原书作者进行了沟通。在此，我不仅想感谢本书的作者们为我们提供了这样一本优秀的书籍，还希望能够感谢他们在本书的翻译过程中给我的支持。

在本书的翻译过程中，我得到了很多朋友的帮助，我想在此对他们致以最深切的谢意。他们是陈宇、董平、葛子昂、李劲松、龙春晖、康飞、马雅凡、毛震宇、史苏、王雅梦、魏翔、徐真、张尉、朱巍，以及 Christopher Zimmerman、Dherman、Igor Dopita、Paul McInnes、Simon Hayers、Steve Wang 等。

我还想感谢我的朋友周惟迪、徐翎以及人民邮电出版社的编辑李岚、李际、王琳让我有机会翻译本书。

最后，我想感谢我的父母，感谢他们给了我一切！

译 者

2006 年 6 月 23 日

前　言

Richard Garriott(“不列颠之王”)

有谁会想到计算机游戏业这样一个有趣的产业会成为主流产业并且在国际股票市场上上市？我们中那些早期加入这个产业的人对此感到十分惊讶。最初我们出于对游戏和计算机的热爱，把编写游戏作为一种爱好，然而它现在已经成为一个高度成熟、充满竞争、高技术、高收入的产业！不仅如此，关于游戏开发的书刊杂志层出不穷，基于游戏小说和背景的电影相继问世，在大学中更是有很多充满激情的程序员和美工，他们的梦想就是从事我们20年来所从事的工作——游戏开发。

我是多么的幸运！我亲眼目睹了游戏从亚文化逐渐融入到主流文化的过程。谁曾想到游戏业能够发展到现在这样？我必须承认计算机游戏产业的发展已经远远超越了我最初的预期。有时人们甚至忘记了我们曾经并没有那么多制作精良、技术先进的游戏，也没有那么多有经验的美工、设计人员和程序员，而行业内的竞争也不像现在那么激烈。当前计算机游戏尤其是大型多人在线游戏的热潮使得我们很容易忘却它们的起点。

一段简短的历史

随着上世纪70年代后期个人计算机的发展，人们开始进行计算机游戏乃至多人游戏的开发。然而，经过了相当长的时间之后，人们才认识到在线多人游戏在商业上是可行的并且值得投入最先进的技术。事实上，这历时近20年。

上世纪70年代初，我刚开始开发游戏的时候，是在一台Teletype电脑上进行的。很少有人还能记起那样的电脑，人们必须在纸带上打洞，并且利用纸带进行对计算机的输入。本质上说，当时还没有图形；人们通过把文本字符按照某些模式排列来模拟图形。只有小部分“骨灰级”玩家和计算机的狂热爱好者得以接触到这些早期游戏。不久，随着技术的不断进步，计算机游戏开始变得更加漂亮并且更具可玩性。电子游戏业和个人计算机产业是同步发展的，当个人计算机变得更好、更流行、更廉价时，电子游戏也呈现出同样的变化。

当廉价的调制解调器出现在计算机配件市场上时，这一切都开始改变。

调制解调器在个人电脑上的出现使得文本模式的多人游戏得以发展。计算机游戏业也将由此进入兴旺期。

随后，因特网时代到来了。

大型多人在线游戏的诞生

上世纪 90 年代，随着 AOL（美国在线）以及类似服务的流行，在线游戏变得更为成熟，但是以我们的标准来看，当时的在线游戏还非常原始。AOL 使得成千上万的玩家可以一起进行游戏；但是那些游戏通常是运行在文本界面或者是粗糙的图形界面上的。虽然用现在的标准来衡量，这些游戏不是很吸引人，并且这些游戏产品从技术上来说也不如当时其他商业软件产品，但它们还是获得了一些狂热的追随者。随着处理器速度、图形技术、声音技术以及存储设备不断变得更快、更好、更大、更便宜，到了上世纪 90 年代中期，我和同事们已经开始制作世界上第一个成功地采用主流技术的大型多人在线角色扮演游戏了。

自那时起发布了很多更新、更好、更成功的大型多人在线角色扮演游戏。这是为什么？是因为它们有趣，人们喜欢它们。越来越多的人投入到大型多人在线游戏的开发中去，因特网为玩家提供了其他类型的计算机游戏所不能提供的东西——社会化的能力。我们不仅可以和在同一城市里的朋友们一起进行游戏，还可以在大型多人在线游戏中遇到不同国家和地区的玩家。如果好好想象一下，你会觉得这非常的刺激。

作为游戏开发人员，我们有责任让玩家能够持续感到有趣并且保证游戏不断地发展。做到这点不仅意味着我们必须提供创意和内容，还意味着每个开发人员或者是开发团队每天都必须面对一些新的挑战。

游戏开发中的问题

虽然在大型多人在线游戏开发过程中会遇到很多问题和挑战，但最关键的一点就是有能力编写稳健的代码。这意味着我们必须进行良好的设计而不是随意地堆砌代码，这意味着我们必须在代码中加入足够的注释以方便支持和扩展，这意味着我们必须为每个游戏要素给出详细的文档以使得未来的开发人员可以理解这个游戏究竟是什么。并且，它最好尽可能“没有错误”。

虽然任意一段大型的代码都不可能完全没有错误，但是在为大型多人在线游戏定义编码和开发标准的时候，我们必须使用比传统单机游戏更高的标准。如果成千上万玩家中的某一个让服务器崩溃了，这个服务器上所有的玩家都不能继续进行游戏。并且，随着我们不断加入新的游戏功能，这些问题代码在内存中的位置会不断改变并且出现不同形式的崩溃，这会掩盖原先代码中的崩溃性错误。

创造性上的挑战

大型多人在线游戏开发中的另一个巨大挑战，是怎样才能让我们的游戏成为最好的。我相信在一个大型在线多人游戏中对单人游戏和多人游戏部分进行正确的组合可以制造出非常有趣的游戏。

我们应该尊重单机游戏，它们能有今天的成功并不是没有原因的。单机游戏最大的优点在于，游戏中每一个细节都被刻意设计得让玩家觉得他自己是游戏中的“救世主”；另一个优点是玩家并不会感觉到每一个拥有这个游戏的人都是“救世主”。单机游戏的缺点在于玩家不能和别人分享快乐，不能和朋友们一起进行游戏。

目前大型多人在线游戏的优缺点恰恰与单机游戏相反：玩家可以和朋友一起在游戏中历险，但不是每一个人都认为他是那惟一的“救世主”。因此，从统计学的角度出发，大型多人在线游戏中的每一个玩家都是均等的，没有一个人是特殊的，没有一个人是“救世主”。那些喜欢通过自己的奋斗解决问题或征服世界并以此获得荣誉的人可以从中感到快乐，然而那些希望自己与众不同或者是成为救世主的人可能并不会感到有趣。

所以在我们的设计中存在着挑战：怎样把单机游戏和大型多人在线游戏的优点结合起来，虽然这两者的本质并不相同。我们（NCsoft 公司的开发人员）已经摸索出一套自己的方案，我也期待着别人能够在这方面有所创新。

使大型多人在线游戏在外观和使用上达到单机游戏一样的效果

开发人员必须始终牢记目标市场对我们的技术需求。开始一个大型多人在线游戏比开始一个单机游戏要难得多。要在家用游戏机上开始游戏，我们只需要插入游戏盘，而单机游戏则需要我们放入光盘，安装游戏以后才能开始游戏。在我们开始享受大型多人在线游戏的乐趣之前需要更多的准备工作：放入光盘、安装游戏、建立账号、学习应该做什么，以及为什么要这样做，学习和别人交流，然后才能真正地开始游戏。

人们会觉得这些安装和进行大型多人在线游戏的步骤很麻烦而且非常无聊，甚至因此而放弃游戏。我们必须在用户界面，安装程序和游戏指南上花很多精力来保证玩家不会因为它们而感到麻烦或灰心。直到目前为止，这个问题还是为很多开发人员所忽视。

大型多人在线游戏开发人员所面对的另一个有趣的技术挑战是在游戏的图形表示方面的。通常，大型多人在线游戏的图像和界面不如单机游戏。造成这个现象的原因是，在开发大型多人在线游戏的过程中，我们需要面对很多其他方面的挑战，因此花费在图形和界面上的精力相对较小，因此历史上大型多人在线游戏在图像上的丰富程度上不如游戏机和

单机游戏。

从未有人成功地开发出优雅并且用户友好的界面，尤其是在最尖端的图形环境下。我希望可以在下一代大型多人在线游戏中看到这方面的突破。事实上，我在这点上向所有的开发人员发出挑战！

展望

在开发大型多人在线游戏的过程中，我们需要考虑多方面的问题。每一天我们都会面临新的挑战，国际化、排名、技术发展以及诸如此类的问题。随着这个产业的不断发展，挑战的数量将会成指数级增长。然而怎样在完成一个伟大而具有革命性的游戏的同时使得它稳定、可支持、可扩展并且易于使用永远是开发人员最根本的追求。

目前为止，在大型多人在线游戏的开发中我们只有一个共同的模式。无论是制作组、游戏公司还是开发人员都遵循同样的开发理念：创造大型的虚拟游戏世界。我希望能够看到这个产业在此基础上更进一步。过去 20 年来，我们已经达到了目前的成就；我相信除了传统的奇幻游戏和科学幻想游戏以外，还有其他类型的大型多人游戏可以获得成功。我希望这个产业中现在或将来的开发人员有朝一日可以给全球的游戏开发人员展示一个不同类型的革命性的虚拟世界！

关于本书

Thor Alexander，Hard Coded Games
thor@hardcodegames.com

欢迎进入《大型多人在线游戏开发》之旅！这是一本全面而深入的文集，其作者都是 MMP 游戏开发人员，正是他们开发了那些最成功或是最让人期待的 MMP 游戏，包括《创世纪在线》（*Ultima Online*）、《无冬城之夜》（*Neverwinter Nights*）、《模拟人生在线》（*The Sims Online*）、《卡通城在线》（*Toontown Online*）、《星球大战：帝国分裂》（*Star Wars Galaxies*）等。阅读本书时，读者可以获得大量独特而宝贵的知识，它们都是由在线游戏产业中最优秀、最聪明的开发人员在进行 MMP 和在线游戏开发时积累的。

目标读者——并不仅仅是程序员

虽然本书在形式上和《游戏编程精粹》（*Game Programming Gems*）这一成功的图书系列相类似，但是本书致力于面向更广泛的读者。不仅程序员可以在阅读本书时获得大量技术知识，设计人员和制作人也能够读懂很多章节并且获得有用的信息，因为这些章节中并不像大多数编程书籍那样使用了大量的技术术语。客户服务人员也会发现本书中不少文章是无价的资源，因为他们很难从其他地方获得类似的知识。读者可以参考下面的表格来获知哪些文章对他们来说是最有用的。

前言表格　　章次及其目标读者

章　名	设计人员	程序员	制作人	客户服务人员
MMP 设计技术	×		×	×
MMP 架构	×	×		
服务端开发		×	×	
客户端开发		×		
数据库技术	×	×	×	×
游戏系统	×	×		×

各部分概览

下面这些简短的概览有助于读者浏览本书的各个部分，发现最感兴趣的文章。

第一章：MMP 设计技术

第一章对总体设计过程进行深入介绍，适合设计人员和制作人员阅读。本章将介绍《卡通城在线》开发团队在为大众市场制作在线游戏时所获得的经验；同时，收录《星球大战：帝国分裂》设计人员 Ben Hanson 关于 MMP 游戏中游戏平衡技术的文章；在最后一篇文章中，前《创世纪在线》主设计师 Paul Sage 对在线客户支持中的常见问题进行了分析。

第二章：MMP 架构

这一章将详细讨论怎样运用面向对象技术以及极限编程方法来创建稳健的 MMP 框架和架构。虽然这一章主要面向程序员，但是它也同样适用于那些想要知道 MMP 中各个部分是怎样一起工作的设计人员。本章收录了由 Twisted 框架之父 Glyph Lefkowitz 撰写的关于 Twisted 开源框架的概览以及 Matt Walker 撰写的《大型多人游戏中的单元测试》。

第三章：服务端开发

这一章将对在线游戏开发中的服务端部分进行详尽讨论：服务端专家 Jason Beardsley 的文章将详细介绍无缝服务器的优缺点，《模拟人生在线》主程序员 Bill Dalton 的文章则会告诉我们应该怎样进行服务端的开发和维护，David Fox 将向我们介绍把 MMP 游戏引入无线设备相关的先进技术，而 Jay Lee 撰写的《在 MMP 游戏中实现移动和物理模块的注意事项》则是每个开发人员都必读的。

第四章：客户端编程

对于任何 MMP 开发人员来说，解决移动预测问题都是一个很大的挑战，Bioware 的 Mark Brockington 和 X-Box Live 的 Jay Patterson 通过两篇互补的文章对其进行了讨论。角色定制是很多在线游戏的一个核心功能，NCsoft 的 Todd Hayes 在文章《使用贴图定制三维角色》中对其进行了详细的讨论。章末是 John Olsen 的一篇优秀文章《游戏机平台上 MMP 游戏的独特挑战》。

第五章：数据库技术

在游戏产业中，很少有开发人员可以真正掌握数据库编程和管理这一“妖术（black art）”。为了改变这种状况，数据库专家 Jay Lee 为这一章撰写了两篇文章。第一篇是针对初学者的数据库基础简介，它适合于所有的开发人员，包括设计人员和制作人；第二篇文章对于在在线游戏中如何使用数据库以及怎样避免对数据库的误用进行了独到的剖析。

第六章：游戏系统

最后一章将讨论 MMP 游戏的核心游戏系统。最精彩的部分莫过于由 Artie Rogers 提供的

极具吸引力的一文《从原料到成品：社会经济中的职业生涯》以及 Mark Brockington 的一流文章《创建声望系统：无冬城之夜中的仇恨、宽恕和自首》。

怎样阅读本书

读者可参考下列方式阅读本书：

- 有选择地阅读，简要浏览本书并且详细阅读感兴趣的内容；
- 从头至尾地阅读，这可能需要很长时间，因此应该把书签放在伸手可及的地方；
- 在电脑前阅读本书。

有选择地阅读

这是我们推荐的方法。本书中的大多数文章都非常短小，读者可以一次读懂它们。由于这些文章都是独立的，因而可以打乱阅读的顺序而不必考虑它们之前的文章。

从头至尾地阅读

像本书这样的文集不是由一个作者写成的，如果我们分别阅读每篇文章的话，会发现它们都是完整的。然而，所有这些文章具有一些共同的主题，并且它们所讨论的重点也相互联系。我们特别安排本书的各章（以及每章中的文章）使得读者可以按照顺序阅读它们。

在电脑前阅读

很多文章中包含了代码示例，读者可以在键盘上输入并运行它们。本书尽可能地在配套光盘中提供相关代码，这样，读者就不需要亲手输入它们。在使用配套光盘前，读者可阅读本书最后的附录中“关于配套光盘”这一部分以获得一些重要细节。

本书的最终目的

信息应该是自由的！创意通常都是简单的。而正是对这些创意的实施成就了那些伟大的游戏。通过和读者们共享这些知识，希望本书能够在 MMP 和在线游戏开发人员之间建立一种合作的气氛。

作者简介

Thor Alexander，Hard Coded Games

thor@hardcodedgames.com

在过去十年中，Thor Alexander 一直致力于把可信的自主（autonomous）角色带入游戏产业中去。为了把最先进的 AI 和机器学习技术带入在线游戏，他于 2001 年在德克萨斯奥斯丁创建了 Hard Coded Games。此前，他曾经在 Electronic Arts、Microsoft 和 Xatrix Entertainment 担任过 AI 编程和游戏设计方面的高级职位；同时，他也是 Asgard Interactive 的创始人之一及 Harbinger Technologies Inc.的 CEO。Thor 还开发了 hyperSim 自主角色系统以及 GoGap（游戏观察捕捉，Game Observation Capture，是一种通过观察真实玩家怎样进行游戏来对 AI 角色进行训练的机器学习过程）。Thor 不仅是本书的编辑及幕后的驱动力，他还曾为《人工智能编程箴言》（*AI Programming Wisdom*）和《游戏编程精粹 3》（Charles River Media, Inc.）撰写过文章。

Jason Asbarh，Asbarh.com

jason@asbarh.com

Jason Asbarh 曾经在 Origin Systems 担任过《创世纪在线 2》的高级工程师。他最近在为 BigSky Interactive 提供咨询服务，为《超级海绵大冒险》（*SpongeBob Squarepants*）开发基于 Python 的脚本和 AI 系统，这是一款针对 Sony Playstation 2 和 Nintendo Gamecube 的游戏。在此之前，他曾经为 Compaq 开发基于 PC 的虚拟角色（这是最早的 PC 虚拟角色之一），也为建筑师制作过虚拟的大楼介绍，并且还曾经为休斯敦健康与医学博物馆（Houston Museum of Health and Medical Science）开发过用于教育的街机游戏。目前，Jason 正在开发基于 PC、家用游戏机和街机的开源模拟系统（open-source simulation system）。

Jason Beardsley，NCsoft Corporation

jbeardsley@ncaustin.com

从 1996 年起，Jason 就一直在为多人在线游戏编写网络和服务端代码。

他拥有麻省理工学院和宾汉顿大学的计算机科学学位。目前，他正受雇于 NCsoft 进行下一代在线游戏的开发工作。

Mark Brockington，Bioware Corp.

markb@bioware.com

Mark Brockington 是 BioWare 的首席研究专家，已供职五年。他曾经在《无冬城之夜》中担任 AI 和网络部分的主程序员。Mark 还曾为《伯德之门》系列游戏开发过多人游戏代码。他在 1997 年获得了阿尔伯塔大学的博士学位。

Mark 希望在此感谢 Yahn Bernier、Mark Darrah、Scott Greig、Sophia Smith 以及 Preston Watamaniuk 对其文章初稿的意见和帮助。

William Dalton，Maxis

bdalton@maxis.com

Bill Dalton 曾经是弗吉尼亚大学的天文学教师。Bill 在 1995 年结束了他短暂的学术工作，加入 Kesmai 进行大型多人游戏的开发。在开发了一系列开创性的在线游戏（包括大多数版本的《空中战士》（*Air Warrior*）后，他搬到得克萨斯奥斯丁开始在 Origin Systems 中担任《创世纪在线》的主程序员。随后，Bill 在加利福尼亚的 Walnut Creek 继续他的职业生涯，在那里担任《模拟人生在线》的主程序员。

Bill 希望在此感谢他美丽的妻子 Genny，感谢她长期的耐心和支持。

David Fox，Next Game

davidfox@ureach.com

David Fox 在 Next Game 工作，他在那里开发基于 Web 和无线的多人游戏。他撰写了多本关于互联网技术的畅销书，还常常在 Developers.com、Gamasutra.com 和 Salon.com 发表作品。过去几年里，David 还在 Sun Microsystem 的 JavaOne 开发大会上做过 Java 游戏方面的发言。

Tom Gambill，NCsoft Corporation

tgambill@ncaustin.com

1995 年，Tom Gambill 在 Kesmai 开始了 MMP 开发生涯。他参与了开发《空中战士》（最早的商业化 MMP 游戏之一）的几个续集。在完成了 Windows 版本的《空中战士》、《空中战士 2》和《空中战士 3》的开发后，他开始为《创世纪在线 2》开发图像引擎。此后，他为 NCsoft 在得克萨斯奥斯丁的分部开发了一个先进的图像引擎。目前他正在为几个原创 MMP 游戏的开发工作频繁地在汉城和美国的办公室之间奔波，那些游戏很快就会在亚洲和美国上市。

Mike Goslin，Walt Disney Imagineering VR Studio

mike.goslin@disney.com

Mike Goslin 在 VR Studio 工作了七年，几乎从事过各个方面的工作，包括编程、游戏设计、总体设计、纯研究等。Mike 曾经在各种各样的项目中工作过，包括主题公园、基于位置的娱乐（location-based entertainment）。最近他正忙于开发《卡通城在线》，一个为所有年龄段的孩子们开发的 MMP 游戏。

Ben Hanson，Sony Online Entertainment

benhanson@soe.sony.com

Ben Hanson 从 1992 年起就开始开发 MMP 游戏了。他参与过大量 MMP 游戏的开发工作，包括一些获得商业成功的早期作品，例如《宝石 3》（*Gem Stone III*）和《龙之世界》（*Dragon Realms*）。他目前正在奥斯丁的索尼线上娱乐全力开发《星球大战：帝国分裂》（*Star Wars Galaxies*）。

Todd Hayes，NCsoft Corporation

thayes@ncaustin.com

Todd Hayes 拥有 6 年的游戏三维图形编程经验，他曾参与《创世纪 9：升腾》（*Ultima IX, Ascension*）等游戏的开发。在进入游戏业之前，他还参与过 Video Toaster 和 Lightwave 3D 等项目的开发。目前他和妻子 Patricia 一起生活在得克萨斯奥斯丁，在那里他受雇于 NCsoft Corporation 为 Richard Garriott 的项目《面纱》（*Tabula Rasa*）工作。

Christian Lange，Origin System

clange@origin.ea.com

Christian Lange 拥有 12 年的编程经验，他曾经在各类项目（从政府项目到 MMP 游戏开发）中担任高级程序员和主程序员。他熟悉游戏开发的多个领域，尤其擅长数据库服务器编程。他目前正在开发他的第 4 个 MMP 游戏。在这个游戏中，他继续致力于服务端底层架构以支持数以十万计的玩家。这是一项非常具有挑战性的工作，而那些游戏迷们让这一切付出都是值得的。他希望在此感谢所有游戏迷，正是他们让他的工作变得如此有趣。

Jay Lee，NCsoft Corporation

gunner10@austin.rr.com

在进入游戏业之前，Jay Lee 曾经在 EDS 工作过十年，为 General Motors、Exxon 和 Sprint 等客户提供 IT 服务。随后他成为 Sierra 的程序员，参与开发了《狩魔猎人》（*Gabriel Knight*）

2、《安塔拉叛变》（*Betrayal at Antara*）、《科利尔百科全书》（*Colliers Encyclopedia*）和《霹雳小组》（*Swat*）2 等项目。在加入 Origin Systems 后，他开始参加大型多人在线游戏的开发工作。在那里，他是《创世纪在线 2》的数据库程序员和脚本主程序员。目前，Jay 供职于得克萨斯奥斯丁的 NCsoft 公司，他正在一个在线游戏中担任技术总监和数据库程序员。

Jay 非常感谢妻子 Jenni 和 3 个可爱的女孩 Shanney、Shannon 和 Shawna。

Glyph Lefkowitz，Twisted Matrix Labs

gunner10@austin.rr.com

Glyph 是 Twisted Matrix Labs 的项目主管，曾参与《创世纪在线 2》的开发工作。作为独立咨询师（independent consultant），Glyph 致力于帮助各种组织使用 Twisted 以及其他开源解决方案进行从部门级到全球级的开发和配置。开发 Twisted 项目时，他领导一支由 19 个开发人员组成的团队，并且管理整个开发过程中的各个部分：从高层设计、远程对象协议到底层的事件循环编程。

Paul McInnes，Micro Forte

paulmc@syd.microforte.com.au

Paul McInnes 不仅是社会人类学家，还是长期专注于在线社区的玩家。他从 1998 年起就进入游戏界。他曾经负责《辐射：钢铁兄弟会》游戏世界的创建工作，也担任过《四驱车竞技场》（*Hot Wheel Bash Arena*）的制作人和游戏设计。目前，他正在 Micro Forte 领导 *Citizen Zero* 的游戏设计工作。

他希望可以感谢 Brit，为了她长期的耐心和支持。

John M. Olsen，Microsoft Corporation

infix@xmission.com

在 1989 年从犹他大学毕业前，John M. Olsen 已参与过各种图形软件的开发工作，他目前正在 Microsoft 公司开发用于创建 Xbox 和 PC 游戏的内部工具。他曾经为《游戏编程精粹》系列撰写过文章，也曾在游戏开发者大会上发言。他感兴趣的领域包括自主 AI、立体图像创建（stereographic image production）、网络以及对数据的组织和分析。

John 特别想感谢和他一起对不计其数的游戏创意和功能进行讨论的 Jay Barnson 以及 Bryan Brown。

Sean O'Neil，Contract Developer for Maxis

s_p_oneil@hotmail.com

Sean O'Neil 于 1995 年在 Georgia Tech 获得了计算机学士学位。从那以后，他全职工作于一家位于乔治亚州亚特兰大的电信软件公司，目前和妻子以及两个孩子居住在公司所在地。

从 2000 年起，Sean 开始利用业余时间开发一个基于分形的行星引擎（planetary engine）。2001 年，他在 GamaSutra.com 发表了他的第一篇在线文章。一位 Maxis 的开发人员看到了这篇文章，于是 Sean 在其本职工作之外，还成了 Maxis 的业余合同制开发人员。

Sean 希望感谢 Maxis 的每个人，正是他们让他可以在这个领域有今天的成就，也正是因为他们的慷慨使得 Sean 可以发表他在 Maxis 所进行工作的大部分成果，和他们一起工作非常愉快。最后，Sean 还想感谢他的妻子 Terri。

Javier F. Otaegui，Sabarasa Entertainment

javier@sabarasa.com

Javier F. Otaegui 是 Sabarasa Entertainment 的项目主管。Sabarasa Entertainment 是布宜诺斯艾利斯的一个阿根廷游戏开发小组。他早在 1996 年就开发游戏了，那时开发《马岛战争》（*Malvinas 2032*），并且在当地市场上获得了成功。现在，他不仅负责美国客户的外包项目，并为当地玩家开发一款商业化的 MMP 游戏。

Jay Patterson，Microsoft Corporation

jaypat@centurytel.net

Jay Patternson 于 1988 年在贡萨格大学获得了计算机科学学士学位。1997 年他开始开发游戏，那时他接任了一款 MMP 游戏（*Castle Infinity*）的主程序员职务。在基于 Web 的虚拟竞技（fantasy sport）游戏领域短期工作之后，他跟随着他在 MMP 游戏方面的导师 Rick Lambright 到了 Cavedog。在那里他工作于 Cavedog 为《横扫千军》所提供的在线游戏《白骨之地》（*Boneyards*）服务，这是一个为了加强点对点的 Internet 游戏并且为之提供更多持久内容（persistent context）而设计的游戏服务。随后，由于不想被时代所淘汰，他进入了互联网领域。他不仅主持了 MMP 游戏 *4X space RTS* 的技术设计工作，还帮助建立了一个在线游戏下载服务。目前，他回到了在微软所担任的职位上，从事与 Xbox Live 有关的工作。Jay 迫不及待地希望基于游戏机的大型多人在线游戏可以发展出一个巨大的玩家群体。

Jay 希望在此感谢 Rick Lambright 为他指引了方向，感谢 Pete Isensee 和 Bruce Dawson 鼓励他坚持下来，他还要感谢 Judy Johnson 给了他做这一切的信心。最重要的是，他希望感谢他的家人（Paula、Jimmy、Amy、Kelly 和 Becky），是他们让他知道生活比工作更为重要。

Patricia Pizer，MMP Design Specialist

ppizer@earthlink.net

Patricia Pizer 于 1988 年开始游戏业生涯，那时她加入了 Infocom，从事游戏后端策划工作。最近，她在 Turbine 娱乐软件公司先后担任了《亚瑟龙的召唤 2》的主设计师和创意总监。目前，Patricia 以 MMP 设计专家的身份为几个大型发行商提供咨询服务，并且和其他游戏设计人员一起研究一些现今最 pertinent 的设计问题。她是波士顿地区游戏开发者网络（Boston Area Game Developers' Network）的发起人和顾问团成员，也是 Zform（针对视觉障碍人士的

游戏软件开发商）董事会成员。当然，大多数时间，她还是喜欢玩游戏。

Patricia 希望在此感谢拉拉队长、编辑和育碧北美在线游戏运行总监 Michael Steele 以及她周围那些了不起的家伙。

Sean Riley，NCsoft Corporation

srileyX@ncaustin.com

Sean Riley 1997 年进入游戏业。在那之前，他主要从事和 Web 服务有关的数据库服务器技术和多层（multitiered）服务器架构等工作。他参与开发的并已发行的多人在线游戏包括第一人称设计游戏《警戒》（*Vigilance*）和 Sega.com 的大型多人游戏 *10Six*。他也曾在 Origin Systems 短暂工作过。目前，他是得克萨斯奥斯丁 NCsoft 的开发人员，正在开发一款下一代的在线游戏，同时他也参与开源项目的开发。

Artie Rogers，Inevitable Entertainment

awrogers@texas.net

“野兽”Artie Rogers 自 1995 年起从事商业化 MMP 以及 PC 和家用游戏机游戏的设计和支持工作。他最近的 MMP 项目是《创世纪在线 2》的游戏设计。Artie 目前在 Inevitable 娱乐公司担任《哈比人历险记》（*The Hobbit*）的游戏设计，该游戏即将由 Sierra 发行。

Paul Sage，NCsoft Entertainment

psage@ncaustin.com

刚刚进入游戏产业时，Paul Sage 曾经做过 Origin Systems 和 Electronic Arts 的技术服务人员。随后他获得了一个 QA 职位，参与了《十字军战士》（*Crusader*）和《银河飞将》（*Wing Commander* ）IV 等项目，并且最后成为一个 QA 主管。之后从事《创世纪在线》测试工作。在游戏发行后，他成了一名客户服务人员，并最终成为游戏管理（Game Master）部门的主管。随后，他在《创世纪在线》中担任游戏设计，并且在玩家人数从 17.5 万人上升到 24 万人这一期间担任游戏主设计。他在 MMP 游戏领域有 6 年的专业经验，同时也是一位热切的玩家。2000 年，Paul 加入了 NCsoft Corporation 的《面纱》项目。

Derek Sanderson，Westwood Studios

gamedesigner@aol.com

自 1995 年至今，“风羽（Windfeather）”Derek Sanderson 一直在从事商业化 MMP 游戏的设计工作，在 5 个已经发行的项目中他从事设计和编程工作。他最近参与的一个 MMP 项目是《银河战将》（*Earth & Beyond*），这是由 Westwood Studios 为 Electronic Arts 开发的一个以太空为背景的 MMP 游戏。现在 Derek 是澳大利亚悉尼 MMP 开发商 Micro-Forte 的高级游戏设计员，大家可以通过电子邮件 gamedesigner@aol.com 和他联系。

Derek 希望感谢帮助他准备这篇文章的好朋友 Emily Jacobson，Emily 是一名无情的编辑，也是美国在线游戏社区的程序经理。

Jesse Schell，Carnegie Mellon University

jns@cs.cmu.edu

Jesse Schell 是卡内基梅隆大学的娱乐技术教授，他的研究方向是游戏设计。他曾经是 Walt Disney 假想虚拟现实工作室的创意总监，在那里，他的工作就是为 Walt Disney 公司构想互动娱乐的未来。Jesse 在那里工作了七年，曾经在多个 Disney 主题公园和 DisneyQuest（Disney 的虚拟现实娱乐连锁中心）项目中担任设计员、程序员以及项目经理。在离开前，他在 Disney 从事面向家庭的大型多人世界设计，譬如说 Disney 的《卡通城在线》。他加入 CMU 娱乐技术中心不仅仅是为了把实践经验带入中心，更为了在那里创造令人兴奋的事物。他在仁斯里尔获得计算机科学学士学位，在卡内基梅隆大学获得硕士学位。他还曾经是弗雷浩佛模仿杂技团（Freihofer Mime Circus）和杂耍工会（the Juggler's Guild）的编剧、导演、乐手、杂耍演员、喜剧演员和杂技演员。

Jesse 希望在此感谢每个为《卡通城在线》得以发行提供过帮助的人，感谢他们即使在眼看机械齿轮怪（Cogs，卡通城在线中的反派角色）将要获得胜利的时候仍然一如既往地给予支持。

Joe Shochet，Walt Disney Imagineering VR Studio

Joe.Shochet@disney.com

Joe Shochet 目前在 Walt Disney 假想虚拟现实工作室担任主设计员和主程序员。在从事 MMP 游戏开发工作前，他帮助 DisneyQuest 和 Disney 主题公园设计和创建了虚拟现实游乐设施。

Matthew Walker，NCsoft Corporation

mwalker@softhome.net

Matt Walker 从 1994 年起就开始编写客户-服务器应用程序。他为 NCR、BellSouth 和 Lexis-Nexis 开发过商业软件。1999 年，他在 Origin Systems 开始了游戏开发生涯。他曾经使用 Python 和 C++在游戏服务端实现各种系统，譬如说背囊管理、对象操作、制造、碰撞检测、行走表面（walkable surface）、脚本环境、可以用脚本控制的游戏构件、事件注册和分发以及各种工具。他对技术的看法更着重于工程方法，他会使用各种方法，包括用例、UML 以及单元测试。他在得克萨斯大学获得了金融学士学位，并且在代顿大学完成了计算机科学硕士的课程。

致　谢

首先，我想向所有参与本书编写出版工作的人员致以最深切的谢意，没有他们的辛勤工作，就没有本书的诞生。尤其要感谢为我们提供封面设计的 Dale Homburg 以及 John Olsen、Matt Walker、Mark Brackington、Artie Rogers 和 Jay Lee 等作者，他们做了很多额外的工作并且在本书出版前最后一刻提供了用于替换的文章。

其次，我想向那些在我追逐和实现梦想的过程中给予我鼓励和启发的人致以特别的谢意，他们是 Jimmy Monson、Marcia Prescott、Diane Pavlik、Steve Lindquist、Gary Moore、Andy Anderson、Jodie Norton、Scott Thomson、Tony Rocco Castellani、Phil Bailey、Lauren Winters、Debbie Schneider、John Rooney、Missy Henrickson、Kelly DeCook、Mark、Kelly Cussans、Andy、Sara Lynch、Buck、Dawn Andrews、Ryan Stoddard、Scott Wescott、Dan "Neller" Heller、Dmo Betts、Aiello 兄弟、Eddie Hamaty、Carl "SSCGBB" Jacobs、Anne Bowers、Scot Barnes、Jose Cayao Jr.、Reza Beha、Phil Dawdy、Melanie Rogers、Howard Jones 及 Aimer Noel Vladic。

此外，我还要感谢那些多年来我在游戏业这个"疯狂"的产业中有幸共事过的伟大的天才们，特别是 Rick Hall、Gordon Walton、Jeff Anderson、Richard Garriott、Drew Markham、Greg Goodrich、Mal Blackwell、Alex Mayberry、Corky Lehmkuhl、Max Yoshikawa、Starr Long 和《创世纪 2》开发团队。

最后，还要感谢 Steve Rabin 和 Dante Treglia 给我机会在他们的著作《AI 编程箴言》（*AI Game Programming Wisdom*）和《游戏编程精粹 3》（*Game Programming Gems 3*）中起笔开始我的写作生涯。同样感谢 vber-publisher 的 Jenifer Niles 在本书编写过程中给予的帮助和支持。

目　录

第1章　MMP设计技术

1.1　《卡通城在线》：面向大众的大型多人游戏 2
Mike Goslin
1.1.1　游戏设计问题 2
1.1.2　社会性问题 10
1.1.3　总结 13
1.2　每个人都需要某个人：怎样让在线游戏玩家进行合作 15
Derek Sanderson
1.2.1　玩家不会进行合作，除非他们必须合作 16
1.2.2　角色扮演是主流：玩家间的合作在玩家可以提供独特的功能时最为有效 17
1.2.3　提供功能：预先定义还是通过能力定义 17
1.2.4　为游戏角色提供挑战 18
1.2.5　保持功能的完整性 19
1.2.6　帮助玩家彼此找到对方 20
1.2.7　帮助玩家进行交流 21
1.2.8　总结 21
1.2.9　参考文献 22
1.3　MMP游戏中的游戏平衡 23
Ben Hanson
1.3.1　为游戏中的数值建立基线（baseline） 23
1.3.2　为数值编写模拟程序 24
1.3.3　建立游戏中的度量（Metric） 26
1.3.4　内部和外部测试 26
1.3.5　在发布后对游戏进行平衡 27
1.3.6　对新功能进行平衡 28
1.3.7　总结 28
1.4　使用支付矩阵来设计游戏平衡和AI 29
John M. Olsen
1.4.1　什么是支付矩阵？ 29
1.4.2　“War”纸牌游戏 30
1.4.3　囚徒困境（Prisoner's Dilemma） 31

1.4.4 简单的格斗游戏 31
1.4.5 不对称的能力 33
1.4.6 延迟和停止 34
1.4.7 自动化 34
1.4.8 总结 36
1.4.9 参考文献 37
1.5 使用用例来描述游戏行为 38
Matthew Walker
1.5.1 什么是用例？ 38
1.5.2 为什么要使用用例来开发 MMP 游戏？ 38
1.5.3 怎样编写用例？ 39
1.5.4 如何识别用例？ 39
1.5.5 用例中的元素：一个标准模版 40
1.5.6 漂亮的图表 45
1.5.7 开始实现 46
1.5.8 用例的指导方针 50
1.5.9 总结 54
1.5.10 参考文献 54
1.6 使用生命值来设计转换因子 55
John M. Olsen
1.6.1 武器的价值 55
1.6.2 治疗、防具和减轻伤害 58
1.6.3 从 NPC 获得的战利品 59
1.6.4 制造业 60
1.6.5 无关物品 61
1.6.6 检验 61
1.6.7 总结 61
1.7 在 MMP 游戏设计中加入故事情节 62
Paul McInnes
1.7.1 为了一个更有意义的 MMP 62
1.7.2 游戏与故事情节：尴尬的结合 63
1.7.3 故事情节在计算机游戏中的功能 63
1.7.4 挑战更多的认知能力 64
1.7.5 使用有意义行动的最佳场所 65
1.7.6 为 MMP 游戏模式加入故事情节 66
1.7.7 公共目标 68
1.7.8 总结 69
1.7.9 参考文献 69

1.8 客户服务和玩家声望：一切都和信任有关 70
Paul D. Sage
1.8.1 捣乱（grief） 70
1.8.2 规则只是工具 71
1.8.3 意图 72
1.8.4 当前的服务部门正在这样做，或许他们还没有意识到 72
1.8.5 声望 73
1.8.6 正当行为 73
1.8.7 恶意行为 74
1.8.8 多样性导致了所有的差异 75
1.8.9 当前使用的行为模式跟踪方法 75
1.8.10 更多的问题 76
1.8.11 使用声望 76
1.8.12 总结 76

第 2 章 MMP 体系结构

2.1 为 MMP 游戏制作仿真框架，第一部分：结构建模 80
Thor Alexander
2.1.1 体系结构纵览 80
2.1.2 支持类 81
2.1.3 核心类 83
2.1.4 管理器类和工厂类 87
2.1.5 仿真类 88
2.1.6 总结 89
2.1.7 参考文献 89
2.2 为 MMP 游戏制作仿真框架，第二部分：行为建模 90
Thor Alexander
2.2.1 把用户和行动者关联起来 90
2.2.2 动作请求 90
2.2.3 动作调度 92
2.2.4 事件广播和处理 92
2.2.5 服务端事件处理 93
2.2.6 客户端事件处理 94
2.2.7 客户端代理 94
2.2.8 仿真与表示分离 95
2.2.9 总结 96
2.3 为游戏脚本创建一个“安全沙盘” 97
Matthew Walker
2.3.1 脚本语言与 MMP 开发 97

2.3.2 使用沙盘的理由 98
2.3.3 安全沙盘的设计 99
2.3.4 在安全沙盘中编写游戏代码 106
2.3.5 总结 108
2.3.6 参考文献 108
2.4 大型多人游戏中的单元测试 110
Matthew Walker
2.4.1 为什么 MMP 游戏需要单元测试 110
2.4.2 单元测试的定义 111
2.4.3 单元测试框架 114
2.4.4 测试先行的设计 117
2.4.5 实用因素 118
2.4.6 总结 121
2.4.7 参考文献 121
2.5 使用 Twisted 框架进行 MMP 服务整合 123
Glyph Lefkowitz
2.5.1 DIY（Do It Yourself）所带来的问题 124
2.5.2 付费找别人来做带来的问题 124
2.5.3 问题小结 125
2.5.4 构造一个解决方案 125
2.5.5 简介：通用的延迟执行机制 125
2.5.6 高层网络服务：全景代理 127
2.5.7 基于 Web 的工具 135
2.5.8 整合独立对象 137
2.5.9 底层整合：协议与网络 139
2.5.10 开发社区 141
2.5.11 总结 141
2.5.12 参考文献 142
2.6 Beyond 2：构建虚拟世界的开源平台 143
Jason Asbahr
2.6.1 纵览 143
2.6.2 服务端架构 144
2.6.3 客户端架构 146
2.6.4 仿真模型 146
2.6.5 总结 148
2.6.6 参考文献 149
2.7 使用并行状态机来创建可信的角色 150
Thor Alexander
2.7.1 状态模式（State Pattern） 151

2.7.2 并行（Parallel）状态层 152
2.7.3 状态管理器 158
2.7.4 跨层阻止（Cross-Layer Blocking） 159
2.7.5 总结 160
2.7.6 参考文献 160
2.8 在 MMP 服务中使用观察者/可观察者设计模式 161
Javier F. Otaegui
2.8.1 观察者/可观察者设计模式 161
2.8.2 基本架构 162
2.8.3 服务端架构 162
2.8.4 客户端架构 166
2.8.5 增强 170
2.8.6 总结 171
2.8.7 参考文献 171

第 3 章 服务端开发

3.1 无缝服务器：优点和缺点 174
Jason Beardsley
3.1.1 杀死怪物不止一个方法 174
3.1.2 无缝世界模式的原型 176
3.1.3 无缝世界模式的优点 178
3.1.4 无缝世界的缺点 179
3.1.5 总结 184
3.2 服务端对象的更新频率 186
John M. Olsen
3.2.1 视觉连贯性与精确度 186
3.2.2 需要发送哪些数据 187
3.2.3 带宽限制 187
3.2.4 每个用户在服务端需要的数据 188
3.2.5 管理数据大小 188
3.2.6 更新队列 189
3.2.7 缺省的更新频率 190
3.2.8 计算范围 190
3.2.9 调整优先级 190
3.2.10 调整队列 191
3.2.11 总结 192
3.3 MMP 服务器开发和维护 193
William Dalton
3.3.1 基本问题 193

3.3.2 对复杂度进行管理 196
3.3.3 总结 199
3.4 小型入口：使用手持设备来接入 MMP 游戏世界 200
David Fox
3.4.1 无线设备和网络 200
3.4.2 J2ME 201
3.4.3 BREW（二进制的无线运行时环境） 202
3.4.4 无线界面和游戏设计总览 202
3.4.5 对象设计 203
3.4.6 网络设计：使用代理服务器 208
3.4.7 总结 208
3.4.8 参考文献 208
3.5 使用 Python 进行精确的游戏事件广播 209
Matthew Walker
3.5.1 事件驱动编程 209
3.5.2 延迟调用 211
3.5.3 事件广播 215
3.5.4 精确的事件广播 220
3.5.5 总结 226
3.5.6 参考文献 226
3.6 在 MMP 游戏中实现移动和物理模块的注意事项 228
Jay Lee
3.6.1 我们可以发布这个游戏了吗？ 228
3.6.2 这是一场战争 229
3.6.3 服务端永远是对的 229
3.6.4 移动的代价 230
3.6.5 移动速度 232
3.6.6 玩家可以从这里到那里吗？ 232
3.6.7 碰撞检测 234
3.6.8 物品放置 236
3.6.9 侵入检测（Hack Detection） 237
3.6.10 总结 238
3.6.11 参考文献 238

第 4 章 客户端开发

4.1 客户端移动预测 240
Mark Brockington
4.1.1 游戏的开发需要良好的移动预测 240
4.1.2 命令时间同步 241

4.1.3 合并路点 242
4.1.4 插值和推导 244
4.1.5 为瞄准延迟使用反向仿真 246
4.1.6 总结 247
4.1.7 参考文献 247
4.2 保持流畅：异步客户和时空穿梭 249
Jay Patterson
4.2.1 共享状态的基本问题 250
4.2.2 航位推测法：时空探索者会做得更好 251
4.2.3 仿真时间表示：为时空穿梭建立通道 252
4.2.4 直接操纵时间 253
4.2.5 总结 255
4.2.6 参考文献 256
4.3 使用程序生成游戏世界：避免数据激增 257
Sean O'Neil
4.3.1 运行时生成的优点 257
4.3.2 运行时生成的缺点 258
4.3.3 地形生成算法的分类 259
4.3.4 修改程序生成的地形 261
4.3.5 高效地渲染程序生成的地形 264
4.3.6 生成贴图 266
4.3.7 在程序生成的地形进行碰撞检测 267
4.3.8 大型游戏世界中的比例问题 267
4.3.9 总结 270
4.3.10 参考文献 270
4.4 为固定大小的对象编写一个高速有效的分配器 272
Tom Gambill
4.4.1 C++中的内存分配 272
4.4.2 一个简单的向量分配器 273
4.4.3 用户友好的分配器模板 275
4.4.4 降低分配器的内存开销 277
4.4.5 总结 279
4.5 使用贴图定制三维角色 280
Todd Hayes
4.5.1 角色定制的类型 280
4.5.2 贴图合成简介 282
4.5.3 分层 283
4.5.4 贴图定制模版和样本 284
4.5.5 样本集合 285

4.5.6 总体实现 286
4.5.7 对定制系统的进一步改进 288
4.5.8 系统的局限性 289
4.5.9 总结 289
4.6 游戏机平台上 MMP 游戏的独特挑战 290
John M. Olsen
4.6.1 环境 290
4.6.2 登录 291
4.6.3 分辨率 291
4.6.4 聊天频道 291
4.6.5 选择目标 293
4.6.6 菜单 293
4.6.7 背囊管理和交易 294
4.6.8 持久化存储空间的问题 295
4.6.9 补充界面 295
4.6.10 总结 296

第 5 章 数据库技术

5.1 关系数据库管理系统入门 298
Jay Lee
5.1.1 表 298
5.1.2 数据查询和关联 298
5.1.3 关系类型 300
5.1.4 属性 301
5.1.5 正规化 302
5.1.6 操纵数据 302
5.1.7 总结 303
5.1.8 参考文献 303
5.2 使用关系数据库管理系统来编写数据驱动的 MMP 游戏 304
Jay Lee
5.2.1 最明显的方法——为什么它不可行 304
5.2.2 可行的方法 305
5.2.3 获取数据 306
5.2.4 常量模块 306
5.2.5 查找表 308
5.2.6 字符串表 310
5.2.7 向客户发送数据 311
5.2.8 本地化 312
5.2.9 总结 313

5.2.10 参考文献 313
5.3 MMP 游戏中的数据驱动系统 314
Sean Riley
5.3.1 在 MMP 游戏中使用数据驱动系统的优点 314
5.3.2 在 MMP 游戏中使用数据驱动系统 316
5.3.3 不同类型的数据源 316
5.3.4 由数据驱动的游戏架构的类型 319
5.3.5 总结 323
5.3.6 参考文献 323
5.4 使用数据库来管理游戏状态数据 324
Christian Lange
5.4.1 模型（schema）设计 324
5.4.2 数据 326
5.4.3 注意事项 328
5.4.4 其他方法 329
5.4.5 总结 331
5.4.6 参考文献 331

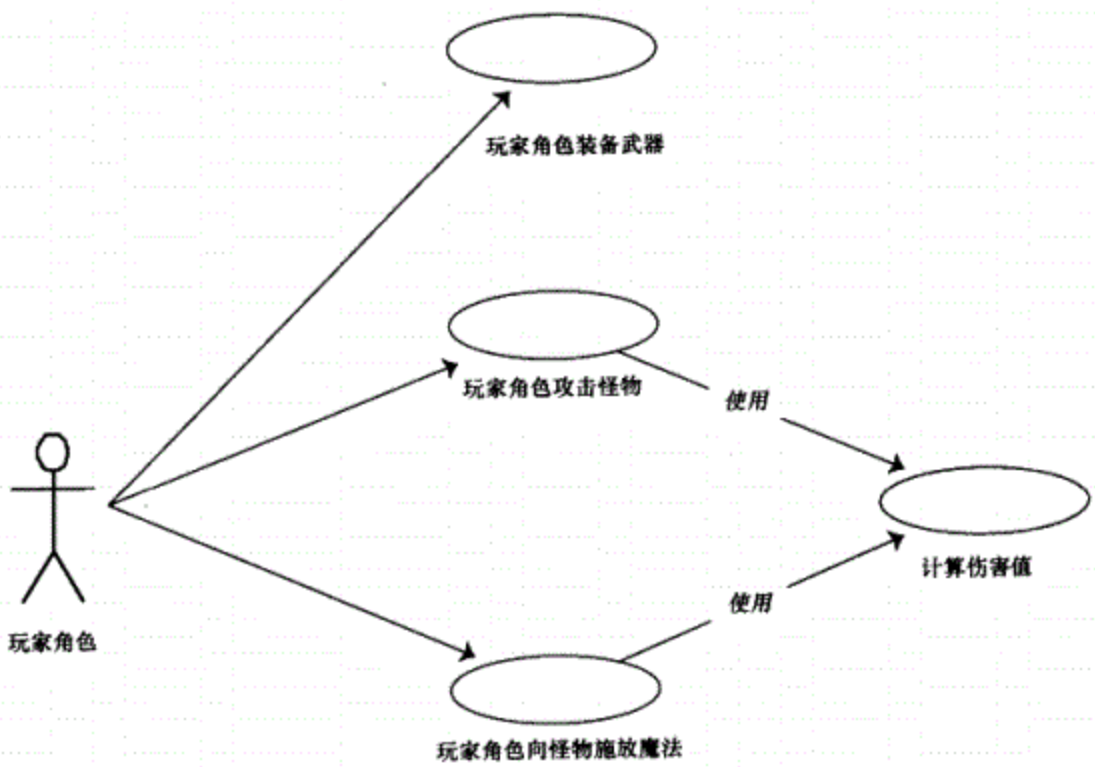
第 6 章 游戏系统

6.1 从原料到成品：社会经济中的职业生涯 334
Artie Rogers
6.1.1 原料获取和加工 335
6.1.2 社会经济中的合作制造 337
6.1.3 物品制造在社会经济中起到的作用 339
6.1.4 总结 342
6.2 玩家房屋供给：我的房屋就是你的房屋 343
Paul D. Sage
6.2.1 成长之路 344
6.2.2 商业方法 344
6.2.3 地段、地段、地段！ 345
6.2.4 应该把戟放在哪里？ 345
6.2.5 经验与教训 346
6.2.6 总结 347
6.3 社会游戏系统：促进玩家社会化及提供游戏回报的另一个途径 348
Patricia Pizer
6.3.1 什么是社会游戏系统？ 348
6.3.2 为什么要让玩家社会化？ 350
6.3.3 目前使用的社会系统 352
6.3.4 回报社会性游戏的创新方法 355

6.3.5 总结 358
6.3.6 参考文献 359
6.4 为创建和管理行会设计灵活的命令集 360
John M. Olsen
6.4.1 创建 361
6.4.2 领导 361
6.4.3 标识 362
6.4.4 行会维护 364
6.4.5 财产 365
6.4.6 专用区域 366
6.4.7 交流 366
6.4.8 总结 368
6.5 创建声望系统：《无冬城之夜》中的仇恨、宽恕和投降 369
Mark Brockington
6.5.1 友谊和仇恨的实现 369
6.5.2 宽恕 372
6.5.3 投降 375
6.5.4 玩家对战的设置 376
6.5.5 总结 377
6.5.6 参考文献 377
6.6 城邦政府在在线社区中的作用 378
Artie Rogers
6.6.1 公民生涯 379
6.6.2 参与城邦工作 381
6.6.3 定义政治过程 384
6.6.4 总结 386

第 1 章

MMP 设计技术



1.1 《卡通城在线》：面向大众的大型多人游戏

Mike Goslin Mike.Goslin@disney.com
Joe Shochet Joe.Shocket@disney.com
Jesse Schell Jns@cs.cmu.edu
迪士尼

《卡通城在线》（*Toontown Online*）是一款为孩子设计的大型多人（MassivelyMultiplayer，MMP）在线游戏。玩家可以创建自己的卡通角色，并加入到成千上万的玩家中去，共同保护卡通城免遭一群被称为机械齿轮怪（Cogs）的商业机器人侵略军的侵略。机械齿轮怪妄想把卡通城美丽的街道变成一排排灰暗的办公楼。卡通角色们通过向机械齿轮怪投掷奶油蛋糕、喷射苏打水或是往它们头上砸铁砧来和它们作战。

《卡通城在线》所面临的挑战在于目标要制作出一个面向大众的MMP。在试图开发一个广受喜爱的 MMP 的过程中，工作人员遇到了不少具体问题。这些问题分为两大类：游戏设计问题的和游戏社会性问题的。本文最后一节将描述轻量级角色扮演游戏（RPG lite）的概念，它是一种类似于角色扮演的游戏模式，但是进行了简化以吸引更多的玩家。轻量级角色扮演游戏的思想可以很好地解决很多游戏设计中遇到的问题。同样，前面提到的游戏社会性问题可以通过简化 MMP 设计中典型的社区要素来处理。

1.1.1 游戏设计问题

1. 问题：孩子的家长也必须是这个产品的销售对象

对于普通的儿童视频游戏来说，家长在选购游戏后就几乎和它没有任何关系了，而面向儿童的 MMP 则需要家长的持续参与。为孩子购买 MMP 的家长必须做到以下几点。

- 支付前期费用。在购买 MMP 时，这部分开支与购买传统游戏的开支相同。然而对于 MMP 来说，这只是一个开始。
- 向因特网服务提供商（Internet Service Provider，ISP）付费。没有 ISP，也就没有 MMP。
- 支付后续的 MMP 订阅费用。孩子们通常没有信用卡，如果 MMP 要不断付订阅费，家长们就必须持续地为此支付费用。

- 能够接受孩子对因特网的频繁使用。在很多只有一根电话线并且使用拨号上网的家庭中，通常上网的频率不高，而且每次上网的时间也不长。为孩子购买了 MMP 后，家长必须接受孩子在游戏时占用家中惟一的电话线。
- 认为 MMP 是一个安全的环境。很多家长对于他们的孩子上网非常担心（这也是理所当然的）。家长必须费神地对 MMP 足够了解，才能知道它是安全的。
- 认为 MMP 是一项有价值的投资。家长必须对 MMP 足够了解才能感到它具有很高的娱乐价值。

这意味着游戏必须让家长付出大量的时间和金钱。为了让他们愿意购买游戏，制作一个家长和孩子都喜欢的 MMP 是一个很好的策略。以下是游戏设计人员必须为此而采用的一些策略。

（1）制作一个足够简单的游戏使得儿童和成人都能喜欢

儿童需要简单的游戏，而很多家长需要更简单的游戏。把精力集中在简单上，就有可能创造出孩子和家长都喜欢的游戏。

（2）借助一个值得信赖的名字

使用类似于迪士尼这样的名字，可以让家长们觉得这是一个安全、优质的环境（更多关于安全感的讨论将在“社会性问题”这一小节中进行）。

（3）让家长和孩子可以共享一个角色

游戏设计人员有意将《卡通城在线》中的惩罚设计得尽可能“无痛”。玩家只需要付出一点点耐心就可以轻易地获得他所失去的任何东西。由此带来的一个意想不到的好处就是两个玩家可以共享一个角色而不必担心另一个玩家会不小心破坏这个角色。这意味着家长和孩子可以轮流玩游戏，共同完善同一个角色。

（4）使用不同层次的玩家都可以接受的故事和主题

《卡通城在线》理所当然地包含了不少喜剧成分，它的目标之一就是捕捉那种家长和孩子都很喜欢的 20 世纪 40 年代的卡通感觉。这需要不同层次的人都可以欣赏的笑话和情节。设计人员尝试着放入一些孩子们喜爱的闹剧幽默，以及一些成年人能够欣赏的办公室幽默来达到这个目标。最初他们比较担心加入孩子难以理解的笑话会带来不好的效果，但是后来发现加入一些成年人的幽默有不少好处。它们在让成年人感到有趣和快乐的同时，也促进了孩子和成年人之间的交流（孩子们想知道这些笑话的内容，所以他们会问他们的家长），它们还让孩子们感觉到自己被作为成年人来对待而不是一直处于被人照料的地位。

（5）制作一个可以与家人分享的游戏

那些有家庭的大型多人游戏骨灰级玩家通常都会有一个遗憾：他喜欢玩游戏，但是由于种种原因（游戏中有暴力内容、过于复杂或者必须投入太长时间），他不能和家中其他成员分享他的爱好。如果可以给他一个他喜欢并且能放心地和孩子以及配偶分享的游戏，他将成为最好的推销员。

2. 问题：冲突是必须的，而暴力是禁止的

创建一个针对大众玩家（尤其是家庭用户）的 MMP 有一些特殊的挑战。虽然开发一个充满欢乐的游戏世界，其中每一个玩家的工作仅仅是四处走动去美化环境或是做一些对别人有益的事情听起来很好，但事实上为了创建一个有趣的世界，游戏必须提供某些形式的冲突。游戏设计人员所面临的挑战在于，这个冲突对于玩家来说既需要有意义并且非常重要，又不

能过于激烈，否则就可能会失去目标客户中的一些重要细分，譬如说家长和女孩。以下几种解决这个问题的方法。

（1）设置玩家之间的对战

到目前为止，玩家之间的对战是 MMP 中让玩家感觉不好的主要原因。游戏设计人员的做法是完全不支持玩家之间对战，因为他们希望卡通城中的人们能够享受互相协作的乐趣，而玩家之间对战则与此背道而驰。如果游戏中必须包含这样的游戏模式，明智的做法是仅将其提供给高级玩家并可选择是否参加。

（2）工作和娱乐之间的冲突

既然决定《卡通城在线》中的卡通之间不会发生冲突，就需要某个会和它们发生冲突的人。由于卡通城被设计为一个适合娱乐的开心城市，很自然的敌人就是某个希望不停工作的人。游戏设计人员由此编出了关于机械齿轮怪的故事。机械齿轮怪是一支商用机器军队，它们想抢占卡通城并将其转变为办公楼。机械齿轮怪的主要目的，是消灭卡通们的建筑而不是消灭卡通本身，这两者之间的区别是非常重要的。这个故事创造了这样一个冲突使得玩家必须共同合作来保护他们的城市。

（3）卡通形式的战斗

游戏设计人员选择了一些传统而有趣的卡通形式来作为战斗模式，而不是很多游戏中所采用的标准攻击模式。卡通们可以向敌人投掷奶油蛋糕，喷射矿泉水，投掷香蕉皮或者把花瓶扔在敌人的头上。机械齿轮怪则使用一些和办公室有关的幽默方式来回击，譬如说官样话，退票或者是喷射水笔。这赋予游戏一个独特的战斗系统：它具有传统角色扮演游戏中所有的战斗策略，但是没有任何会使它失去所针对的大众市场的真实暴力。图 1-1 展示了卡通城中一个正在进行的卡通形式的战斗场景。

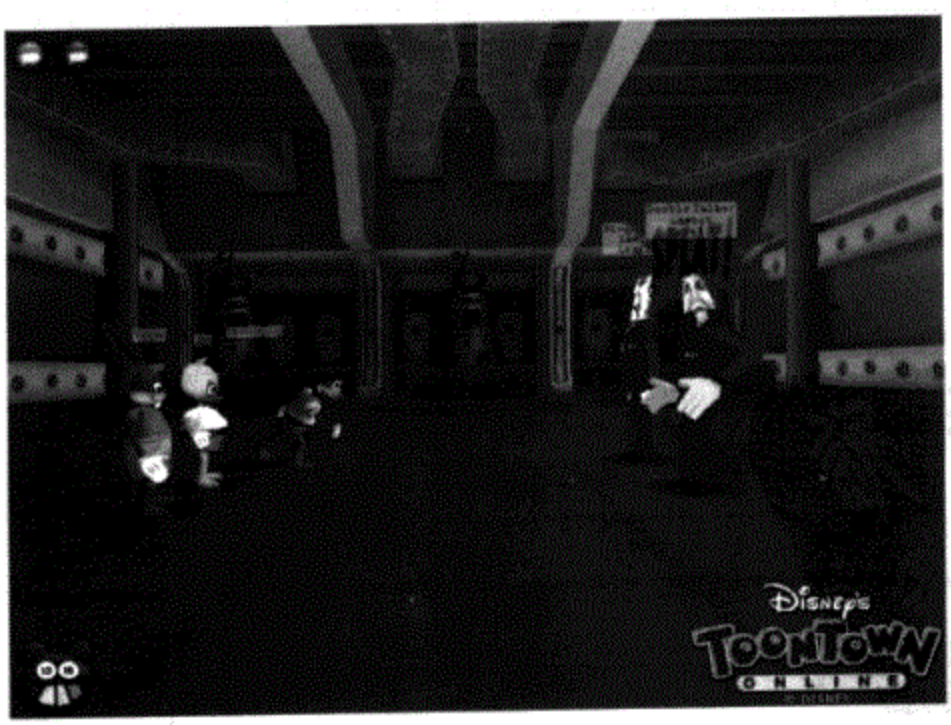


图 1-1 和机器敌人之间的一场具有卡通城特色的战斗

（© 2002. 华特·迪士尼公司授权复制）

（4）机器敌人

从某种意义上说，这是假想的，然而在游戏中它确实真实地存在着。摧毁虚构的机器人的暴力程度要比摧毁某些虚构的生命形式低很多。游戏设计人员通过在它被击败时使用的图像和声音效果来强调它是一个机器人而不是一个活着的生物。

（5）笑槽

传统的角色扮演游戏使用血槽或者是生命值来告诉玩家角色与死亡的接近程度，这对于《卡通城在线》来说不是很合适。所以游戏设计人员创造了“笑槽”（laff meter），它表示了卡通的快乐程度。如果某个卡通角色被机械齿轮怪击中的次数过多，它将会觉得悲伤。当笑槽达到零点时，这个卡通角色会由于过度悲伤而不能继续战斗，它会被送到附近的娱乐场所去恢复快乐。虽然这仅是对传统血槽系统的一个微小改变，但是它的确减少了战斗中的暴力因素。

（6）为了正确的动机和敌人战斗

很多游戏中，玩家可以从被打败的敌人身上获得财产，《卡通城在线》通常避免这样做。因为为了使敌人不能夺走玩家的城市而去击败它和为了获得敌人所拥有的财产而去击败它具有本质区别。游戏中也有一些例外，譬如说当玩家击败一个机械齿轮怪时，可以要求它归还一件从其他卡通那里偷走的东西。

（7）团队协作，而不是个人作战

相对于独立进行战斗来说，让这些玩家组成小组来和敌人战斗可以使他们获得更多的正义感。因为他们进行战斗不仅仅是为了保护自己，还为了保护团队中的其他玩家。

3. 问题：太多的选择会让玩家无所适从

在传统的 MMP 游戏中，玩家创建完角色后就直接投入到游戏世界中去了，他们需要自己去发现接下去该做什么。那些休闲玩家（Casual Gamer）通常不愿意去阅读很长的用户说明，因此他们会因为缺乏结构而沮丧，最好的方法就是在游戏内部建立一个提示机制。

（1）指南

逐渐带领玩家深入游戏是一个不错的方案。开始的时候非常简单，只有很少的选项，随着玩家对游戏的了解，可以提供更多的选项。卡通城使用一个集成的指南和任务系统来引导玩家完成以下步骤：

a) 单人、无互动的指南；
b) 必须完成一些简单的任务，作为单人指南的一部分；
c) 通过一个互动指南来介绍多人特性；
d) 在多人世界中执行一些简单任务；
e) 简单的多人任务（交朋友）；
f) 用户可以根据自己的喜好选择一系列难度递增的游戏任务并且完成它们。

对于大多数玩家来说，指南是他们接触游戏的第一环节，也是游戏设计人员获取良好第一印象的惟一机会。必须安排足够的时间设计这个指南，因为设计可能会经过几次失败。将指南集成到游戏中对设计来说是一个很大的挑战，但确实值得为此进行额外的设计、实现和测试。

（2）任务

《卡通城在线》的整个游戏给玩家提供一系列明确说明了的任务，因此他们对于所要做的

事情没有任何疑问。游戏还提供了一个“任务列表”来告诉玩家当前任务、任务要求以及任务的完成情况。为了使每个玩家的游戏经历更加个性化，我们创建了一个多等级的任务池系统，这样每个玩家都可以从随机产生的任务中自由选择。当一个玩家在一个等级中完成了足够的任务，就会进入下一个等级去接受难度更高的任务。玩家达到一定等级后，还可以同时进行多个任务。

虽然长期目标（譬如把机械齿轮怪从卡通城赶走）给玩家提供了一个很好的游戏背景，但是真正让玩家上瘾的是游戏中的短期任务（譬如把面包带给面包师，打败 3 个机器人）。不过如果总是使用同样的方法来升级，玩家会觉得单调。在游戏中使用不同的升级途径非常有效。在游戏中，通过任务系统获得的升级奖励所占比重最大，譬如提高笑槽的上限或是获得大甜豆罐。但是除此之外，游戏还提供了其他升级系统，譬如说，战斗中获得的经验就是一个独立的升级系统。战斗升级系统和任务升级系统没有任何联系并且获得的奖励也不属于同一类别（一些其他嘘头）。此外还有一个“捕鱼升级系统”用来帮助玩家恢复失去的笑槽点数。虽然玩家会发现他们需要同时在多个升级系统中升级，但是在不同的升级系统中切换焦点有助于使这个游戏更加有趣，多变且有深度。

4. 问题：玩家希望立即享受游戏的乐趣

骨灰级玩家会为进行游戏做很多事情，而普通玩家则希望能够立即享受游戏。前面描述的任务系统可以帮助用户找到游戏的乐趣所在，另外，《卡通城在线》还试图在那些 MMP 中必不可少的方面增添一些乐趣。

（1）下载过程

在《卡通城在线》里，玩家可以在等待下载的过程中享用简单的 Flash 电影和游戏，而不必盯着屏幕上的进度条一动不动。这也是介绍一些游戏要素和背景故事的好机会。

（2）创建卡通

设计自己的头像是一个有趣的混合匹配游戏。就连选择名字（在有些 MMP 游戏中，所选择的名字被多次拒绝让人非常灰心）也很有趣，因为游戏提供了一个可以生成各种疯狂的卡通名字的“趣味名字生成器”。

（3）瞬间传输

瞬间传输是快速移动所必须的。《卡通城在线》使用了一个异想天开的“便携洞”。卡通角色从口袋中拿出一个“便携洞”，跳进去，然后从目的地的另一个洞中跳出来，如图 1-2 所示。

5. 问题：普通玩家惧怕失败，尤其是在众人面前

休闲玩家在游戏早期遭遇失败并不会觉得有挑战性，反而会觉得气馁。在 MMP 游戏中有很多陌生人看着他们，因此失败会带来更多耻辱。游戏设计人员采用了很多策略帮助玩家克服惧怕心理并树立自信。

（1）确保成功

《卡通城在线》的指南在开始时会引导玩家进行一场不可能失败的“作弊了的”战斗。这可以让玩家对今后的游戏生活预先了解，并且有助于建立自信。玩家会发现在今后的游戏生活中他们可以很轻松地重复同样的胜利。

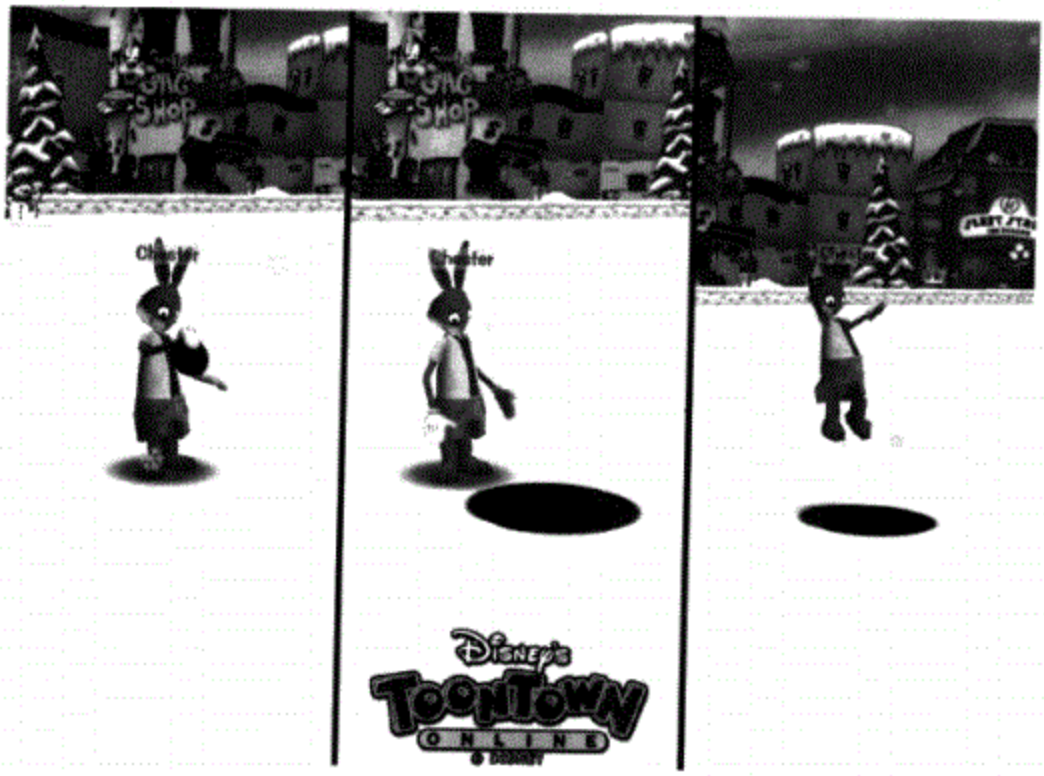


图 1-2　卡通使用“便携洞”在《卡通城在线》中瞬间传输

（2）观看战斗

回合制战斗系统的特点之一，就是玩家可以观看别人的战斗而不需要在战斗中亲自上阵。这可以帮助那些胆小的玩家了解更高级战斗中可能会发生的事情。年纪比较小的玩家在不知道怎样进行战斗之前特别担心被卷入战斗。在自己参与战斗之前观看别人的战斗可以帮助他们获得自信。这也是街机游戏玩家常用的策略，他们在自己进行游戏之前会先观察别人怎样玩。

（3）强调团队合作

对于大多数休闲玩家来说，肉搏战不仅意味着威胁和压力，而且毫无吸引力可言，他们更希望合作。游戏设计人员在《卡通城在线》的游戏系统中下了很多功夫使得玩家可以组成互利的团队，仅在少数情况下玩家之间会存在直接对抗。

（4）新手和老手一起游戏

《卡通城在线》中包括了能够让新手和老手一起游戏并且各取所需的有效方法。这有利于促进玩家间的社会化，也可以帮助新手树立信心并且给他们一些可望而可及的目标。很多老手也乐于帮助新手了解游戏的奥妙，所以使用这样的游戏模式可以获得多方面的好处。游戏设计人员把迷你游戏（minigame）系统设计为任何等级的玩家都可以共同参与并从中获益的模式。战斗系统中也有这样的要素：在团队战斗中，机械齿轮怪倾向于攻击团队中经验值较高的玩家，这可以给新手多一些保护。

6. 问题：过于复杂的游戏系统会让大众玩家不知所措

虽然传统 RPG 游戏的范式和技术都很有用，但大多数大型多人（MMP）游戏和角色扮演游戏（RPG）非常复杂，针对大众市场把它们修改得更简单是非常有效的。以下是一些比较好的方式。

（1）避免使用数字来描述状态

使用数字来描述状态可能会打断沉醉于游戏中的玩家，并且相对于使用图形表示的信息来说，它不容易被大多数玩家理解。当然，有时数字是惟一较好的方法，但是应该把它作为最后的选择。

（2）适当的冗余信息

适当的冗余信息可以让玩家获得更清晰的描述。譬如说，笑槽中的数字表示剩余点数，同时，笑槽中牙齿的数量则使用图形方式显示了同样的信息。玩家瞥一眼就可以很快地知道自己大概的健康值。如果他看得更仔细一点，就可以知道确切的健康状态。

（3）背囊应该是有限的

很多游戏把背囊中的物品作为一个无限列表来显示，这样，玩家就需要自己来进行分类管理。这通常意味着必需多次滚动屏幕才能找到想要的物品。《卡通城在线》中只有一定数量的战斗物品，它们排列在一个单屏幕的表格中，其中每一个格子是一类物品，如图 1-3 所示。玩家管理物品的工作量非常小，并且可以很方便地知道自己有些什么物品，因为同一种物品总是放在相同的位置上。

图 1-3 卡通城游戏中背囊里面的物品被排列得很容易管理

（4）合并和统一界面

背囊表格系统得到了进一步完善：它用来统一战斗、交易、背囊管理和经验接口——因为它们都是对同一物品集合进行操作的。这样一来，用户只需要了解一个界面的细微变化就行了，而不必学习 4 个不同的界面。

（5）避免不合理的按键

大多数大众玩家从来没有玩过电脑游戏《雷神之槌》（*Quake*），他们也不知道“扫射”是什么意思。如果一定要使用键盘来控制，确保让操作尽量简单。如果游戏中的键盘控制比俄罗斯方块还难，那就多花些时间去简化它。

（6）使用熟悉的界面

每当引入一个新界面时，尽量使它看上去和玩家曾经遇到过的某个界面相似。《卡通城在线》游戏的登录界面看上去像一个网页，因为人们习惯于从网页登录。该游戏对卡通的控制类似于其他三维游戏，因为两者有很多共同之处。《卡通城在线》的朋友列表和即时通信系统中的伙伴列表类似，因为很多人习惯于那样的界面。

（7）尽可能隐藏不必要的细节

《卡通城在线》游戏的玩家不需要知道他们在哪个服务器上进行游戏，他们甚至不需要知道什么是服务器。游戏会自动选择一个服务器，并且每次都会自动登录到同一个服务器上。玩家可以使用服务器切换界面来切换到新的服务器，但是那些不关心自己在哪个服务器上的玩家永远都不需要知道怎样切换。

（8）让探索充满乐趣

游戏应该设计得让玩家可以很方便地找到正确的路径。对大众玩家来说，在一个设计得不是很好的游戏中寻找道路并不是一件有趣的事情。《卡通城在线》中，去掉了街道中的回路（那些会使玩家回到原路上的十字路口），因为回路只会把玩家弄糊涂。此外，要尽量避免在游戏中出现死胡同，因为从原路返回对大多数人来说并不好玩。如果玩家需要画张地图才能在游戏世界中找到路径，那就说明游戏空间设计得太差了。

（9）不要过于简单

一个需要很多个小时才能完成的有深度的游戏必须具有一定的复杂度。复杂度应该逐步提高。游戏应该遵循奥塞罗所说的“一分钟就可学会，一辈子才能精通。”如果复杂场景出现得过早，玩家可能会无所适从，它们应该留到玩家能够应付的时候。《卡通城在线》中的战斗系统其实非常复杂，但是大多数复杂的部分或是被刻意隐藏以待玩家去发现，或是仅向达到一定经验水平的玩家开放。

7. 问题：通常玩家每次只玩一小段时间

现在的孩子都很忙，他们需要完成家庭作业，参加足球训练，看电视节目，还要早点上床睡觉。成年人则更忙！大多数玩家通常不能连续三四个小时玩游戏，通常他们每次只能玩 20～30 分钟。在大多数 MMP 游戏中，这只够一个玩家登录游戏系统找到另一个玩家，然后就要退出了。《卡通城在线》采用了一些技术，使得玩家只需要登录 20 分钟就可以获得一次有效的游戏经历。

（1）明确的目标可以节约时间

任务系统可以让玩家具有明确的目标，并且会通过“任务列表”提醒他们。这样玩家可

以节约很多时间，因为他们无需在游戏中徘徊就可以知道自己应该去做什么，也不需要回忆上一次游戏时做过什么。相反，他们可以专注于某个明确定义了的目标。

（2）可度量的进展使得玩家每次进行游戏的时间可以很短

通过游戏为玩家进度提供了一个度量标准，所以即使只是完成了一个很小的动作，他们也能获得成就感。游戏中的任务列表不仅仅显示一个任务是否完成，它还显示每一个任务的当前进度。譬如说，如果所执行的任务需要打败 5 个机器人，任务列表可能会显示玩家已经打败了其中的 3 个。如果玩家打败了第 4 个，那将会是一个重要的成就，因为它改变了任务列表中的进度显示。

（3）对行动进行良好的定义可以让玩家更好地管理时间

如果可以知道某个类型的行动需要多长时间，玩家就能更好地管理他们的游戏经历，这样他们就可以方便地在有限的时间里安排不同的行动。《卡通城在线》游戏力图使不同的行动需要不同长度的时间，这样玩家就可以根据自己的时间来选择不同的游戏方式。下面是一些例子。

- 捕鱼：1 分钟或更长时间。
- 迷你游戏：2 分钟或更长时间。
- 街头战斗：3～5 分钟。
- 物品递送：3～5 分钟。
- 建筑中的战斗：10 分钟。

（4）迅速地找到朋友

MMP 玩家会把很多时间浪费在寻找他们的在线朋友上。《卡通城在线》提供了一个朋友列表来显示当前在线的朋友。不仅如此，玩家还可以瞬间移动到朋友的身边，这样就可以立即和朋友分享游戏的乐趣了。

（5）随时退出

在某些 MMP 游戏中，如果在退出游戏时，玩家正处于一个危险区域中，下次登录时他还会出现在同一地点。更糟的是，在退出后，他的角色可能会在原地继续待 1 至 2 分钟，并且不能对任何攻击做出反应。因此，玩家必须在退出游戏前留出 5～10 分钟以返回安全的地方。在《卡通城在线》中，玩家随时退出游戏时，角色会立即从游戏世界中消失。下一次登录游戏时，角色会出现在最近的游乐场所以确保他的安全。

1.1.2 社会性问题

对于大型多人游戏来说，最具挑战性的一点就是，设计、开发和平衡游戏模式仅仅是游戏设计工作的一半。开发人员必须对整个游戏进行全盘考虑，包括聊天、玩家群组、网络安全、对《儿童在线隐私保护法案》（*Children's Online Privacy Protection Act，COPPA*）的遵守以及游戏背后的社会问题。因为开发人员把玩家邀请到游戏世界中来，那些玩家在游戏允许范围内制造的问题就是他们的问题。大众玩家对于这类事情的容忍度非常低。当玩家在网络游戏中陷入不愉快的环境时，一两个微小的社会问题就意味着他们可能会退出游戏。

1. 问题：受众对于“捣乱（Griefing）”的容忍度很低

捣乱是指以降低游戏的完整性为目的、在游戏世界中反复地骚扰其他玩家的行为，即使这并不会给捣乱者带来任何好处。《卡通城在线》的每个子系统都进行了特别的设计以把捣乱的影响降到最低。下面是一些例子。

（1）名字过滤器

在外部公测中，最多的捣乱形式就是有些人试图“愚弄”名字过滤器。虽然每一个卡通名字必须通过非常严格的自动检测，但是并不存在完美的自动检测系统。过滤器不能检测出外语中的不良词汇、倒着拼写的词汇、首尾互换的词汇、发音相似的词汇以及一些其他情况。一小部分玩家以欺骗名字过滤器为乐，直到被游戏管理员发现，并且将他们的名字更换掉。当系统发现某个玩家使用了不良的名字，就会把他的名字改为一个非常单调的，和机械齿轮怪类似的名字，譬如“卡通 45782”或者“卡通 20034”。最后得出了这样一个结论：没有一个自动名字过滤器可以符合游戏要求，所以现在使用人工来辅助名字过滤器的工作。

（2）在游戏中推动其他玩家的卡通

另一种捣乱形式是，玩家可以把其他玩家的卡通角色在游戏中推来推去。如果玩家把他的卡通角色留在游戏中，并且在一定时间内不去操纵它，它就会睡着。当卡通角色睡着时，游戏仍然会对其进行物理和碰撞计算，这样一来，其他玩家就可以推动它。实际情况下，其他玩家只需要 15 分钟时间，在一两个朋友的帮助下就可以把一个睡着的卡通角色从安全区域推到街上并且让它进入一场战斗。因为没有人操纵这个卡通角色，它必然会在这场战斗中失败。玩家只需要注意别把无人照料的卡通角色留在游戏中就可以轻易地避免这种形式的捣乱，所以游戏选择允许这种形式的捣乱，而不是把卡通角色设计为不能够被推动。因为如果一个卡通角色不能够被推动，玩家们就可以把几个卡通角色连在一起来挡在门口或是通向其他重要游戏场景的通路上。

2. 问题：玩家不能容忍那些在现实生活中也会发生的社会问题

《卡通城在线》没有实现任何形式的偷窃、储藏以及玩家间战斗，也没有实现真正的商业系统。另外，游戏中不同种族之间没有任何本质的区别。在设计早期设计人员就决定把设计重点放在娱乐性上，并且不断提醒自己：我们是在制作一个游戏，而不是要模拟真实生活。这样，玩家的卡通角色在游戏时不会遭受任何永久性的伤害。这让家长和孩子可以放心地共享游戏账号，而不用担心他们中的某个人在游戏时会被抢劫或是把贵重物品送给别人，也不用担心会失去游戏中的地位或是遇到其他对游戏账号不利的事情。

3. 问题：家长不放心孩子使用因特网

制作面向大众市场的在线游戏必须特别注意在线安全。能够和其他玩家进行交流是人们进行在线游戏的主要原因之一，因此必须寻找一个安全的交流形式，这样游戏才可能成功。游戏设计人员认为允许所有玩家之间进行开放式聊天是不安全的；而完全禁止任何形式的聊天也同样难以接受。因此游戏设计人员开发了一些系统来帮助玩家安全地聊天。

（1）基于菜单的聊天系统

游戏设计人员开发了一个基于菜单的聊天系统，玩家可以使用游戏预先设定的短语进行

交流。这在多数情况下都可以达到效果。基于菜单的聊天非常安全，因为玩家不可能泄漏私人信息。那些打字很慢的玩家可以快速交流而不必担心拼写、标点或是大小写。基于菜单的聊天系统中不会有缩写、俚语和行话，而那些都会让新手对 MMP 游戏敬而远之。随着玩家在任务系统中的进展，这个菜单还进行了动态更新。由于菜单上大部分聊天短语基本上都是积极友好的，《卡通城在线》中的陌生人之间往往也很友好。认知失调（cognitive dissonance）方面的研究支持这个现实，它表明人的态度实际上受到行为的影响。

（2）受密码保护的聊天

如果玩家想要在游戏中和现实生活里认识的朋友聊天，基于菜单的聊天系统就有很多局限。游戏提供了“秘密朋友”系统来为这种“封闭式聊天”提供便利。玩家可以生成一个密码并告知一个现实生活中的亲戚、邻居或是朋友。一旦对方输入这个密码，游戏就会建立一个只有这两个朋友可以看到的聊天通道。因为在游戏中无法交换密码，这样可以确保每个使用这个系统的人都在游戏之外见过面。游戏还采取了额外的防范措施：密码只能使用一次（也就是说，每个新的秘密朋友都有一个不同的密码），并且在密码产生后 48 小时内没有使用就会作废，游戏设计人员还为每个玩家可以拥有的秘密朋友数目设置了上限。

4. 问题：游戏特性和社会特性通常是不一致的

玩家长期进行游戏的主要原因是为了获得社会体验。这意味着社会特性具有与游戏特性相同甚至更高的优先级。认识到这点后，游戏设计人员就要尽全力避免游戏特性妨碍社会特性，除非是出于以下策略性的原因。

（1）永远不要阻止朋友

几乎在游戏中的每一个地方，玩家都可以和周围的朋友聊天或是发消息给远处的朋友。除了在一些特殊战斗中，玩家可以在任何时候瞬间移动到在游戏中的朋友身边以外。在那些战斗中不让玩家移动是为了增加一些风险因素。

（2）社交界面

设计人员试图让所有的游戏界面都“群组友好（group-friendly）”。无论是在战斗中选择攻击还是购买物品，玩家都可以看到群组中所有成员的交互、状态和聊天窗口。玩家角色的每个状态（譬如捕鱼、检查背囊、睡眠）都是分散的，因此所有其他玩家都可以准确地知道他在干什么。游戏世界中实现了卡通风格的“文字提示气球（word balloon）”。这样玩家不需要从游戏世界中转移注意力就可以阅读聊天记录。对大众玩家来说，在玩游戏的时候阅读聊天记录就和在看电影时阅读字幕一样令人讨厌。

（3）合作性迷你游戏

最初，设计人员想要在竞争性迷你游戏和合作性迷你游戏间达到一个平衡。他们认为玩家的贪婪本性可以确保一个适当的竞争气氛，通常情况下也确实如此。然而他们没有意识到当一个群组一起进行迷你游戏时，对于单个玩家来说他们的主要目标不再是获得尽可能多的糖豆，而是让整个群组都可以获得糖豆。于是竞争性游戏退化为消极的轮流游戏，这样每个玩家都可以轮流获胜从而获得尽可能多的糖豆。因此，在一个 4 人玩家群组中，每个玩家都可以确定在进行 4 次迷你游戏并且获得胜利后，自己会获得糖豆。合作性迷你游戏不会受到这一问题的影响，因为合作与获得糖豆这一共同目标是一致的。认识到这点就可以尽可能地

把合作因素整合到迷你游戏中去。

5. 问题：很多孩子和新手在刚开始游戏时会感到害羞

很多孩子生来就害羞；而一个成年人在刚刚开始进行 MMP 游戏时，既不知道游戏中会发生什么事情，也不知道应该怎样进行游戏。然而，社会归属感是 MMP 游戏中最让人上瘾的部分。为了帮助玩家跨越从新手到老手之间的这个缺口，下面是游戏中设计的一些重要的社会特性。

（1）轻量级群组（lightweight grouping）

如果能够很方便地和其他玩家组成群组，玩家就可以有效地降低建立关系的壁垒。设计人员决定在游戏中取消显式地（explicit）组成刚性的群组，而是让每个人都可以毫不费力地为特定任务建立一个临时的群组。为了做到这点，设计人员把组成群组作为游戏世界布局的隐含部分。譬如说，在迷你游戏电车（minigame trolley）和战斗系统上都有 4 个玩家位置。陌生的玩家们只要跳到同一辆电车上就可以一起玩迷你游戏，同样，他们只要加入同一场战斗就可以共同作战。这不需要口头交流或是预先计划。

（2）打破沉默

从《夏令营》（*summer camp*）人们都可以学到，打破沉默可以促进交流从而建立友谊。在《卡通城在线》中，像迷你游戏、捕鱼以及与机械齿轮怪作战之类的行动都被设计得有助于打破沉默。

（3）共同的目标

没有一样事物可以像共同的目标那样把玩家联系在一起。游戏中的任务系统倾向于给出和同一地区其他玩家的任务目标相重合的任务。因此，当玩家在街上徘徊时，很可能会发现其他玩家也具有相同或类似的任务目标。由于游戏中的战斗系统本质上鼓励多玩家合作，玩家可以很方便地和其他玩家一起与共同的敌人作战。

（4）朋友列表

为了可以对朋友进行长期追踪，游戏设计人员开发了一个集成在游戏中的朋友列表。玩家可以用这个朋友列表来看到哪些朋友在线以及他们的位置，玩家还可以瞬间移动到朋友身边或是向他们发送聊天消息。

1.1.3　总结

在设计面向大众市场的 MMP 游戏时，主要目标就是让玩家愿意在游戏中投入足够长的时间，这样他们才会在游戏角色中投入更多心血并且和在线社区建立联系。对于玩家对角色所作的投入，游戏必须有所回报。传统的 RPG 模式在这一点上做得很好，它可以被用来设计那些需要长时间进行的游戏。不幸的是，很多这类游戏被刻意地做得难以掌握并且非常复杂，以此来取悦那些骨灰级玩家。如果游戏在刚开始时向玩家隐藏复杂性并且让开始时的难度足够低，就可以有效地使用轻量级角色扮演游戏模式（RPG-lite model）来开发面向大众市场的 MMP 游戏。

轻量级角色扮演游戏的原理可以应用于游戏设计的所有方面。它的指导哲学是永远不要让 RPG 框架妨碍玩家从游戏中获得快乐。下面是一些关于应用轻量级角色扮演游戏原理的经

验方法。

- 让玩家可以立即享受游戏所带来的快乐。大众玩家没有骨灰级玩家那么好的耐心，因此在游戏最初的 15 分钟里，游戏不仅需要让他们感到快乐，还必须给予一定的回报。在《卡通城在线》中，最刺激的经历莫过于进行一场战斗。因此在游戏最初，游戏指南会立即把玩家带入一场战斗。玩家不会在这场战斗中失败，这样就可以使玩家在游戏指南结束后敢于参与其他的战斗。
- 避免强迫玩家对不了解的事物进行选择。典型的 RPG 设计会让玩家进行选择，譬如说，选择成为一个巫师、剑客或是小偷。这个选择会影响玩家在今后游戏中的能力和限制。通常骨灰级玩家知道成为一个巫师或是剑客意味着什么，但是大众玩家缺乏这方面的知识，并且，当他们在今后发现当初应该选择另一个职业时，他们会不高兴。《卡通城在线》通过把游戏中的能力和最初的角色选择相分离来避免这个问题。无论玩家选择成为一只狗、猫还是鸭子，他的能力都会由他在游戏中的行动来决定。
- 地理布局应该完全由游戏模式决定。大众玩家需要一些说明来告诉他们在不同的地方可以进行哪些行动。如果希望玩家们聚集在一起，就应该放一片空地或是一个可见的地标。在《卡通城在线》中，玩家在海岸线上看到一个码头就会知道这里有一个可以捕鱼的池塘。
- 每次进行游戏的时间可以很短，并且也应该很短。通常大众玩家不愿意花费 3 个小时来达到一个游戏目标，因此应该把任务分解成为子任务并且允许按照很小的增量来衡量进展。在《卡通城在线》中，玩家只需要 15 分钟就可以达到一个有意义的目标。
- 允许并且鼓励玩家成为观众。MMP 游戏的社会特性还可以进行修改以取悦更广泛的受众。MMP 玩家可以通过观看其他玩家来学习游戏技巧，这样就可以有效地对其自身进行训练从而更好地参与游戏。大众玩家在第一次遇到新的游戏模式时通常会害怕，因此应该让他们有机会去观察别人。在《卡通城在线》中，大多数战斗发生在路中央，发生在所有人面前。
- 相对于独立完成任务来说，团队合作更有意义。MMP 游戏的长期目标就是加强玩家间的联系，因此游戏模式应该尽可能地鼓励合作。《卡通城在线》中几乎每个行动都被设计为面向玩家群组，当玩家合作时，他们会获得点数奖励或是其他战略上的好处。

创建面向大众的 MMP 游戏非常有挑战性，通常也可以获得很多回报。重要的是永远不要忘记玩家是谁并且牢记轻量型角色扮演游戏的设计原理。最终你将获得一个持久的、具有广泛吸引力的游戏。

1.2　每个人都需要某个人：怎样让在线游戏玩家进行合作

Derek Sanderson，Westwood Studios
gamedesigner@aol.com

在过去的几年里，有一个观点已被广为接受，那就是构筑社区（Building Community）的思想是 MMP 游戏成功的关键。尽管支持这一观点的严谨的统计数据并不多，但是有 3 个合乎常识的理由让游戏鼓励玩家之间的合作。

第一个理由是，玩家间的合作可以降低系统的开销。让用户共享游戏资源可以增加单个游戏服务器能够处理的并发玩家数量，还可以降低为单个用户支出的硬件开销，从而提高净利润。假设每个服务器预计最多可以承受 3000 个并发玩家的负荷，并且每个玩家都可以安全地和一个怪物独立作战，这样一来，游戏服务器就必须能够同时对 3000 个怪物（当然，根据游戏设计的不同，实际的数字可能有所不同）进行 AI 调用、碰撞处理以及各种其他任务。如果 3 个玩家才能打败一个敌人，那只需提供 1000 个并发敌人就可以了。

第二个理由是，合作进行游戏不仅可以使游戏内容更加富于变化，还可以让玩家花费更多的时间来完成所有预先创建的游戏内容，并且可以降低他们对游戏产生厌倦的可能性。换句话说，如果可以制作一个游戏使玩家能够自行创造快乐，他们完成所有游戏内容的速度就会变慢。他们订阅游戏的时间越长，所产生的利润就越高，还会将更多的朋友和家庭成员带入游戏世界来与他们一起分享快乐。

第 3 个理由是，玩家之间的合作可以使他们建立起牢固的社会关系，从而不愿停止游戏。虽然有时候玩家在游戏中的社会关系会转移到游戏以外，但是如果他们的在线身份与他们在游戏中的社会关系紧密地结合在一起，他们将不愿意放弃这个身份。如果一个玩家停止参与某个 MMP 游戏，虽然他仍然可以保持和朋友之间的关系，但是他通常会担心失去在在线社会中的地位。

上面的理由都被极大地简化了，并且它们也不适用于所有类型的 MMP 游戏。但是，这篇文章假定它们是正确的。本文的目的是为成长中的 MMP 开发者提供一些实用的建议，而不是创建一个理论性的作品。本文中的大多数讨论都会以“传统的”MMP 模式为重点。在这种游戏模式中，玩家操纵他们的角色在一个假想的虚拟环境中进行游戏。然而，细心的读者会

发现本文所提供的设计策略可以应用于任何需要多个玩家协作的情况。

1.2.1 玩家不会进行合作，除非他们必须合作

游戏中应该包含多少合作游戏的成分？这一问题常常会激起 MMP 设计小组成员之间的争论，尤其是在讨论是否需要强制玩家进行合作时，争论会变得更为激烈。小组中一部分成员（通常是那些比较外向的设计人员）希望让合作游戏占据主导地位，而剩下的成员（通常是比较内向的）虽然也支持合作游戏，但是他们希望能够确保玩家也可以单独进行游戏。然而，没有一个游戏可以让玩家无论是独立游戏还是和别人合作都可以用完全相同的方式来进行。在所有其他条件都相同的情况下，相对于独立游戏而言，与其他玩家合作不仅需要付出更多的努力，还会面临更多的麻烦，因此玩家总会选择单独进行游戏。如果游戏设计人员希望玩家在任何行动中都和别人互相合作，就必须让合作游戏在大多数情况下可以比独立游戏获得更多的回报且更为有趣。

然而，千万不要错误地在游戏中去掉所有单独进行游戏的机会。网络游戏中有很多内向的人，他们在游戏中所占的比重甚至比在真实世界中还要高。内向的人需要可以独立进行的行动，除非他们做好了和别人交互的准备。如果一个 MMP 游戏要求玩家持续不断地与他人合作，内向的玩家就会很快感到疲倦，他们会觉得在游戏世界中历险不像是在娱乐，而更像是在工作。这会使他们转而选择竞争者的产品，如果他们能在那些产品中找到快乐并且不会被强迫地进行大量合作的话。在线游戏的固定成本很高，因此需要很大的订阅数来维持这一成本，如果一个游戏需要玩家不停地进行交互，这将是致命的。

可能读者会觉得上面的说法是自相矛盾的，怎样才能对一个游戏进行平衡使得它在支持合作游戏的同时也让独立游戏变得可行呢？答案很简单：与其试图让游戏模式中的每一方面都同样地适用于独立游戏和合作游戏，还不如创建一系列不同的任务，它们中一部分适合独立游戏，另一部分适合玩家间的合作，这些任务最终提供了一个合作性的社会环境。

《创世纪在线》（*Ultima Online*，*UO*）是上述思想在实践中一个很好的例子。这个游戏中的许多职业本质上就是个体户，尤其是在手工业模式中。譬如说，玩家可以从事矿工、樵夫、炼金术士、厨师以及其他职业，这些职业主要的任务就是搜集资源并提供用于历险的物品。这些任务不需要别人的合作就可以完成。一个希望独立地伐木或者制作药水的玩家在进行他们的工作时通常不必担心会受到伤害。*UO* 中的手工业系统是内向玩家的天堂。

这些职业通过游戏世界中的市场来与其他玩家进行合作。*UO* 中的玩家不断地需要药水、盔甲以及其他用于历险的物品。只有那些手工业大师才能制造出最高质量的物品，*UO* 中的手工业者向那些参与历险的玩家们源源不断地提供产品。这的确是合作游戏，即使不是在同一地点进行。然而，事实上，使用现金来交换物品的交易通常发生在这两个参与交易的玩家不同时在线的时候，出售物品的玩家把他的物品交给一个站在他房子外面的由计算机控制的 NPC（Non-Player Character，游戏中的非玩家角色）商人，这个商人对于手工业者的顾客来说，就像一个自动售货机。这样层次上的交互对于一个比较内向的、倾向于独立进行游戏的玩家来说最好不过了，他可以在游戏社会中起到独特而重要的作用，但不必为了销售他的物品而和其他玩家面对面地讨价还价。

1.2.2　角色扮演是主流：玩家间的合作在玩家可以提供独特的功能时最为有效

让玩家具有独特的功能对游戏的成功而言至关重要，因为 MMP 游戏的流行并不仅仅因为它是一个可以和其他玩家一起历险的社交场所。很少有人愿意为了和一堆多边形聊天而每月支付费用。相反，MMP 游戏成功的核心原因是，它们可以让玩家不必花费很多时间就获得成就感。人们参与 MMP 游戏是因为这些游戏可以使他们获得成功，而所花费的时间比在真实世界中所需要的短得多。玩家只需要付出几百个小时的努力就可以变得强壮、富有、闻名、博学、性感或是其他他所想要的结果。从心理学角度来说，这非常有吸引力。

这个基本概念非常容易把握，不过需要注意的是，不同的个性类型对于成功的定义差别很大，如果游戏设计人员假定玩家与他们有类似的喜好或厌恶，他们就必须为这个假定承担风险。类似于“玩家们不喜欢 X”（“玩家们不喜欢 PvP”是这些说法中较常见的一种）之类的说法对于成功的游戏设计来说是极为危险的。如果游戏设计人员常常做这样的假设，那应该改变这个习惯。要知道，不存在这样的“玩家们”。

某些游戏潜在用户会因为他们的角色变得更加富有而感到成功，有些则对通过提升技能和等级来完善他们的角色更有兴趣，有些渴望名望，还有一些仅仅期望那种被别人需要的感觉。和现实生活中一样，MMP 游戏中，成功的定义非常的多，为了吸引更多的潜在玩家，设计人员必须建立各种不同的方法使玩家可以在游戏中实现价值。要达到这个目的，最好的方法就是让玩家具有独特的功能，每一个功能都可以用来满足一类基本的个性化需求，并且必须有助于满足其他玩家的需求。

在 MMP 中，玩家作为角色来说他的作用是由游戏赋予他的能力决定的。在《无尽的任务》（*EverQuest*）中，一个具有“假死”技能的僧人是一个“招揽者（puller）”，他可以诱使怪物离开安全的群体从而被一起冒险的伙伴杀死。巫师则是“远程攻击者（nukers）”，这种角色在远处就可以造成大量的伤害。在很多虚幻 MMP 游戏中，牧师是“复活者（rezzers）”，这类玩家可以使无法继续游戏的玩家复活。从这些例子可以简单地看到玩家是怎样把角色和它们的功能联系在一起的：玩家用角色所具有的功能来定义它。

1.2.3　提供功能：预先定义还是通过能力定义

虽然很多方法都可以用来定义玩家在游戏中的功能，但近年来，有两种方法最为流行。第一种是传统的“角色分类（character class）”模式，它被用于很多不同时代的游戏中，从古老的《龙与地下城》（*Dungeons and Dragons*）到《暗黑破坏神》（*Diablo*）到目前最流行的 MMP 游戏，像《无尽的任务》（*EverQuest*）和《亚瑟暗世纪》（*Dark Age of Camelot*）。基于角色分类的游戏通过预先创建的模版来描绘一个角色在游戏世界中的功能，它可以让玩家知道他们的角色在今后的发展路线：例如，从一个侍僧开始，然后成为一个牧师，接着成为一个大祭司；或是从一个学徒开始，最终成为一个高级巫师。从护卫到骑士、从少尉到上将，“角色分类”模式在游戏初期就能把玩家和他们的在线角色联系在一起，这使得玩家可以立即

把自己和其他玩家区别开来并且为他在游戏中的任务建立一个早期目标。

在最近的项目《银河战将》（*Earth & Beyond*）中，游戏开发人员为角色发展使用了“角色分类”的模式，游戏中加入了很多特性来让玩家感到他们在游戏中所起的作用对于其他玩家来说是有价值的。譬如说，在角色创建过程中，一段话外音会详细地介绍角色分类，并且为每种分类指出 3～4 个关键技能。不仅如此，在游戏最初的两个小时中，每个角色会获得一系列任务，这些任务需要使用他们的关键技能，这样玩家就会知道这些技能在游戏环境中是多么的有用。通过同时使用这两个系统，玩家在游戏职业生涯的最初就会知道他们所选择的角色定义是《银河战将》社会中一个不可或缺的要素。

第二种用来提供功能定义的常见模式，通常被称为“基于技能（skill-based）”模式。在这种模式中，角色是由它所能做的事情定义的，而不是通过预先创建的模版定义的。在基于技能的模式中，玩家通过游戏提供的机制获得技能，他所掌握的技能决定了他在游戏社会中的功能。《创世纪在线》是“基于技能”系统的经典范例，游戏中的每一个角色都可以学习并且掌握任何技能，他最擅长的技能就是他练习最多的技能。譬如说，玩家通过砍树来掌握伐木技能；同样玩家也可以通过用剑和怪物作战来精通有刃兵器。

基于技能的模式在“核心”游戏市场中非常流行，因为它为玩家创建自己的角色提供了最灵活的方法。然而，使用这个模式也有一定的风险，因为相对于传统的“角色分类”模式来说，它更像是一个针对“专家”的系统，它可能会让从未玩过 MMP 游戏的玩家感到沮丧。告诉一个新玩家他的角色是一个治病术士，玩家也许就能猜到他在游戏世界中所起的作用。把他放入游戏世界中，他通常会尝试实现他的功能，即使在一开始会犯些错误。另一方面，如果不给一个新玩家任何类别，而是赋予他急救和白魔法这两种技能，他可能最终会知道如何成为一个治病术士，但是他的早期游戏经历很可能类似于鲁滨逊（Robinson Crusoe）——他会带着一些粗劣的工具在一个陌生的世界里迷失方向，想方设法地找些事情做。如果游戏使用了基于技能的模式，就应该考虑为玩家提供一些指导和建议来告诉他们那些技能可以做什么。

1.2.4 为游戏角色提供挑战

一旦决定了基本的角色发展机制以及所提供的功能类型，开发人员的下一个任务就是设计游戏内容使之能够不断地支持并加强那些功能。只有组合各玩家的游戏机能，他们才能发现和征服游戏中的每一个挑战。

创建这些挑战时，游戏设计人员要避免使某个特定角色只能通过惟一的方法去完成一项任务，而应该让不同的角色可以按照不同的策略进行组合来征服挑战，否则就会因为当前的玩家群组中没有具备必须技能的其他玩家而导致整个群组无法征服挑战。另外，不要把角色设计得只能在少数情况下使用，而应该让角色在不同情况下都有用，这样就可以避免某个玩家在游戏中的功能太窄以至于不是很有意义。这有助于避免游戏中的“死锁”（gridlock），在发生“死锁”时玩家是不会感到快乐的。

譬如说，如果游戏中有锁住的门，玩家必须打开它才能进入游戏中的某些重要区域，如果只有锁匠才能打开这样的门，那么，当一队玩家中没有锁匠时他们就不能继续游戏，最终受挫的玩家会拨打客户服务电话投诉。所以与其让一个单一的功能来限制玩家进入特殊区域，

还不如在指定一个功能作为创建某个角色主要方法的同时，赋予其他角色一些次要的功能，使他们可以用更大的代价打开这些门。譬如说，这样的次要功能可以是爆破专家，他可以打开这扇门，所付出的代价是可能会吸引附近的守卫过来察看爆炸的原因。

如果锁匠只能打开被锁住的门，他不是会变得很无聊吗？如果这是他在游戏中惟一的功能，那就犯了我们所说的第二种错误：给予功能的应用范围太窄。他会是其他玩家游戏过程中的一个重要部分，即使这个任务是他的玩家伙伴们急需的，但是反复地完成一个机械的任务会使他感到很无聊，这就会导致游戏中的锁匠越来越少。为了避免这样的情况，明智的方法是创建一个更广义的功能。例如给予锁匠更广泛的功能定义，他可以作为修补匠来打开被锁住的宝箱；制造陷阱来诱捕敌人；制作机械武器或是作为备用铁匠对盔甲和武器进行紧急修复。

1.2.5　保持功能的完整性

在拓宽功能的同时，游戏设计人员无论如何要小心地保持每一个角色在游戏世界中地位的完整性。如果所引入的游戏特性使玩家不再需要某个特定功能，提供这个功能的玩家就没有必要继续留在这个游戏中了。这类问题常常发生于 MMP 游戏交付后的“运行”环境中。开发小组为了向玩家提供新的游戏内容，往往会创建一些新的技能、游戏系统或是物品，它们可能与某个现有的功能重复。这样的重复是极其危险的，因为它削弱了那种职业的价值。

回到锁匠这个例子中。假设这是一个传统的“中世纪幻想”游戏，玩家进行冒险远征并且带回很多锁着的宝箱。通常这类游戏中会有某些类似于“浪人”或是“小偷”的角色来提供锁匠的功能。这些角色的核心技能之一就是为其他玩家打开锁住的东西。正如前面所说的，如果没有其他方式可以打开箱子，在没有浪人/小偷的时候，需要他们提供这一服务的玩家就会抱怨。

为了让玩家满意，开发小组可能会采取一个短视的方法来解决这个问题：提供一些对任意玩家来说都很容易获得和使用的开锁工具。虽然这让那些希望打开被锁物品的玩家感到满意，但这是一个危险的解决方案，因为它削弱了浪人/小偷的独特功能，这会使浪人角色失去他曾经拥有的那种为玩家伙伴们提供独一无二的服务的感觉。因此，与其为玩家提供一种打开箱子的简单方法，还不如为他们提供一些界面来找到某个愿意提供这项服务的浪人。

“不可削弱”原则也有例外。譬如说，如果一个角色提供了某个对于完成游戏来说至关重要的关键服务，就不应该让那个角色成为这项服务惟一的提供者。一个常见的例子是角色的“死亡”。MMP 游戏常常用死亡来控制玩家接受挑战和得到回报的节奏，它使得玩家试图接受远远超过其能力的挑战变得很有风险，从而防止他们这样做。尽管如此，玩家们仍然会不断地尝试安全的绝对极限来获得更大的回报，这会不可避免地带来很多死去的角色。

一个死去的角色在能恢复正常游戏前必须被复活。如果进行复活的惟一方法是寻找另一个具有这项技能并且愿意帮助死者的玩家，那么玩家有时会因为找不到具有这项技能的玩家来帮他复活而不能继续游戏。这会让人极为灰心。玩家可能会因此打电话给客户服务并且要求游戏职员来执行复活仪式，更坏的结果是，他们可能会因此取消账号。

这里，最好的选择是为游戏角色创建一个独立于其他玩家的复活机制，这样他们就可以继续他们的游戏而不必依赖于别人。然而，相对于找其他玩家来帮助复活，这个机制必须是

一个不太理想的方法。两相权衡，他们显然应该寻找其他玩家来帮助复活，但是那些不愿意等待的玩家至少还有一个缺省选择。

例如，在《创世纪在线》中，一个死去的玩家会变成一个鬼魂，他可以在一个死一般静寂的灰色地区中游荡以寻找一个由计算机控制的治病术士来复活他。这里的代价是要找到这个治病术士所耗费的时间，这在游戏中偏远的区域可能需要非常长的时间。因此，明智的 *UO* 冒险者团队在进行一次对地下城的远征时会带上一个熟练的治病术士。

1.2.6 帮助玩家彼此找到对方

在现实世界中，我们如果需要一辆出租车，通常就会去寻找一辆黄色的轿车并且拼命招手以吸引注意。如果因为散热器爆裂而被困在高速公路上并且没有移动电话，我们会寻找警车所特有的警灯。在餐馆里，我们会寻找穿着制服的服务员。这些职业以及很多其他职业在我们的社会里提供了独特的角色。因为他们穿着制服，我们可以很方便地找到并认出他们。

明智的 MMP 设计人员会想方设法在游戏中使用这些有效的方法。如果所有的角色都使用同样的原型、衣着和装备，并且游戏没有定义任何方法去区分它们，玩家可能需要花费很长时间来寻找可以为他们提供所需服务的人。

《亚瑟龙的召唤》（*Asheron's Call*）是一个基于技能的 MMP 游戏，它没有可以区分玩家职业的现成标识。游戏中有 3 个种族，但玩家的功能并不是由种族决定的，也没有什么方法可以对玩家所具有的特殊功能进行分类标识。因此，受伤的玩家会常常站在人群中，恳求某个会治疗魔法或是有医疗箱的玩家站出来帮助他们。从一个对《亚瑟龙的召唤》的设计人员比较公平的角度来说，要防止某些玩家为了引起注意而站在热闹的地方大声叫喊，惟一的办法是偷走他们的键盘，但是无法辨认那些具有关键技能的玩家使这个问题更为恶化。

《无尽的任务》则是另一个极端。角色的功能受到玩家种族的限制，每个功能都有一到两个“最佳选择”。不仅如此，在游戏模式的最高级别中，对每个功能来说，可供选择的“最佳装备”很少，这样一来，对那些具有给定功能的角色来说，他们中能力最强的玩家彼此在外观上都很接近。虽然通过外观上的相似来辨认角色的功能并不是最佳的方法，但这的确是一个可行的方法。

然而，要让玩家可以发现那些具有他们所需技能的其他玩家，通过视觉上的区别来进行辨认并不是惟一的方法。地理位置是促进玩家之间交互的另一个常用工具。这里的诀窍在于创建可以满足某种玩家共同需要的场所，这可以让玩家在这个特定的地方聚集。最常见的例子可能就是“银行”，它几乎存在于每个 MMP 游戏中，在银行里，玩家可以把物品保存在他们身体以外的地方。其他的常见场所包括可以购买物品的交易中心或是玩家必经的连接点（nexus），譬如说城门、桥梁或是十字路口。我最喜欢的例子来自于《宝石 3》（*GemStone III*）。

《宝石 3》中的节点（Node）网络是一个基于地理的社会系统，这些节点是一些可辨认的位置，在那里除了可以进行快速的治疗和魔力恢复以外，还提供了一些有利于游戏中所有角色类型的其他功能。这样做的好处很大，以至于很少可以在除了节点以外的其他地方发现玩家，这使得节点成为游戏中主要的社会中心。交易、销售、治疗、聊天，都发生在节点中。

然而除了视觉上的区别和地理上的接近来让玩家发现彼此，另一种做法是为玩家提供

一个界面使他们可以找到他们需要的其他玩家。这些界面中最常见也是最简单的是 who 这样的聊天命令，它可以列出在线的玩家，或许还会提供关于他们位置和能力的信息。更先进的方法是基于 GUI（Graphical User Interface，图形用户界面）的系统。譬如说，《亚瑟暗世纪》的界面可以让那些寻找冒险小组的玩家把自己标记为空闲，这样冒险小组可以搜索那些技能达到他们特定需要的空闲玩家。如果一个《亚瑟暗世纪》中的冒险小组正在寻找一个“萨满巫师”（shaman）类型的角色，他们可以在进行搜索时使用 GUI 以针对这个角色进行过滤。

1.2.7　帮助玩家进行交流

现在玩家已经具有不同的功能并且需要共同合作来面对挑战，他们也可以方便地找到彼此。下一个挑战就是提供一些工具使得玩家可以管理彼此之间的交互，这样玩家就可以把时间花在游戏上而不是和游戏机制（game mechanics）作斗争了。

首先应该简化交流方式。设计人员有必要为那些试图在游戏中高效合作的玩家提供有效的交流方法，虽然这应该是一个在项目初期就需要考虑的主要设计目标（它本身就值得用整本书来描述）。

某种意义上说，这应该由特定游戏的动态部分（dynamic）决定。譬如说，如果游戏需要玩家小组进行快速的战略决策，设计人员就应该考虑提供某种界面使得玩家可以通过发出简短的命令来协调行动。如果游戏更偏向于战略，那设计人员或许应该为玩家提供一个一致的交流方法，这样，在不同时刻进入游戏的玩家就可以协调他们的集体战略。

交流不仅仅是交谈。《创世纪在线》的贸易系统就是一种交流形式：玩家可以在家门口放上一个由游戏控制的商人角色来出售物品。放置了商人角色就意味着玩家在说：“我有物品出售——来看看吧！”另一种非口头的交流方式是在角色上显示一个其他玩家可以看到的标志，这样可以告诉别人当前的游戏状态或是期望做些什么。在《银河战将》中，丧失能力的玩家可以在他们的飞船损坏后开启一个“救援信号”，这使得他们的飞船在这个地区的任何一个雷达中可见。这里所传达的消息是：“我需要救援！”

其次要确保共同进行游戏的玩家能够方便地保持他们的游戏会话。玩家在 MMP 游戏中创建了一个冒险小组后，有些小组成员会因为互联网连接出现一些临时性问题而和其他成员失去联系。如果玩家需要花费一半的游戏时间去寻找这些失散的成员，他们会觉得很灰心。虽然带领-跟随（leader-follower）系统在基于文本的 MUD 中实现起来很简单，但它们还没有在图形界面的 MMP 游戏中完全实现过。

1.2.8　总结

事实上，很多现有的 MMP 游戏中常见的设计都是基于过去非图形状态 MUD 时期所做出的错误假设之上的，我非常希望在未来的几个月或几年里，随着设计人员对创建多人游戏环境的基本假设进行重新考虑，我们能够看到大量的创新。毋庸置疑，合作游戏仍然会是主流，特定游戏在多大程度上实现它将对这一游戏的寿命起到决定性的影响。无论如何设计人员应该注意——合作游戏实现起来很简单，但是要正确地实现却很难，因此应该把它作为一个早期的设计重点并且在整个开发周期中对最初的假设进行检验。

1.2.9 参考文献

[Bartle] Bartle，Richard，“Hearts，Clubs，Diamonds，Spades: Players who suit MUDs,” *http://www.mud.co.uk/richard/hcds.htm.*

[Koster01] Koster，Raph，“Online World Design Patterns,” *http://www.legendmud.org/raph/gaming/despat_files/frame.html.*

1.3　MMP 游戏中的游戏平衡

Ben Hanson，SONY Online
bhanson@soe.sony.com

MMP 环境对游戏平衡提出了一些独特的挑战。单机游戏通常只提供 40～50 小时的娱乐，而典型的 MMP 游戏把游戏时间延长到数百甚至数千小时。因此，在进行游戏平衡时必须考虑到这个超长的生命期。不仅如此，在线游戏的社会性使得玩家可以迅速地交流游戏平衡上的弱点并对其加以利用。如果存在一个最好的武器或是最好的技术，玩家群体很快就可以找到它。因为一个玩家的行为会影响到其他玩家，游戏中明显的不平衡会对整个游戏社区产生显著的影响，玩家会对这些弱点加以利用，这会导致那些没有利用这些弱点的玩家在与他们的对抗中处于下风。

本文所介绍的游戏平衡方法考虑了 MMP 环境所带来的额外的复杂情况。这些方法在开发的早期就开始对游戏进行平衡，并将其延伸到公开测试中，甚至在游戏发行后还持续进行。

1.3.1　为游戏中的数值建立基线（baseline）

虽然在开发早期建立的任何游戏数值都不可能保持不变，但还是可以在那时就开始游戏平衡过程。这时，开发人员可以为游戏中的数值建立一条基线。

要做到这点，最好从项目中的核心系统开始。大多数 MMP 游戏从战斗系统开始最为理想。为战斗系统建立数值基线可以达到很好的效果，因为在还没有设计和实现战斗系统之前，它的基本目的已经非常清楚了（攻击并且摧毁目标）。因此游戏开发人员可以很方便地对某些数值（譬如说攻击力和攻击速度）作出假定，在后续开发过程中也很可能可以对它们加以利用。表 1-1 中的例子展示了怎样为一个 MMP 幻想游戏建立基线。在这个例子中，我们用 3 个基本变量来定义武器：攻击力、命中率和攻击速度。当然，我们在实现战斗系统的过程中，可能会对这些变量进行修改，也可能引入其他变量。即便如此，这仍然是一个有效的平衡方法。

表 1-1 武器的数值基线

武器	攻击力	命中率	攻击速度
斧头	18	-20	10
棍棒	15	15	13
匕首	6	40	35
大槌	37	-40	6

要知道，这些数字本身并不是重点。相反，重点应该是它们彼此之间的关系。在表 1-1 中，武器的攻击力范围是从 6～37，匕首和大槌代表了两个极端。虽然这个范围很可能不是最终确定的，但是它提出了一个有用的规律：匕首具有最低的攻击力而大槌的攻击力最高。当实际的范围随着战斗系统的形成被改变时，这个规律可以保持。如果在游戏的后期测试中，游戏开发人员认为匕首的攻击力最好在 20，那么这个数值基线就可以让设计人员立刻知道必须提高棍棒和斧头的攻击力。

接着，这些初步的武器数值就可以用来建立其他对象的数值基线。譬如说，游戏开发人员觉得一个处于平均水平的怪物应该在被一个处于平均水平的武器攻击 10 下后死去，通过这样的粗略估计，可以推断这个怪物应该有大约 200 点的生命值（假设斧头是一个处于平均水平的武器）。这样就可以用处于平均水平的怪物的生命值来计算游戏中其他生物的生命值。强壮的怪物具有超过 200 的生命值，弱小的怪物则相对较少。只要保持武器之间的关系（斧头处于平均水平），那么处于平均水平的怪物的生命值也是可知的，因此每个怪物的生命值都会受到武器数值变化的影响。

这些武器数值的基线可以被延伸到战斗系统以外的其他系统中。作为攻击力最低但是最易于使用的武器，匕首是初学者最理想的武器。根据这一逻辑，设计人员可能会决定，玩家可以在游戏最开始的时候就获得使用匕首的能力。因为匕首易于使用，它们可能会是那些不适合搏斗的职业（譬如说男巫）最合适的武器。另外，因为大槌是游戏中攻击力最强的武器，可以把它保留给那些具有很高的近身战斗技能的角色使用。

1.3.2 为数值编写模拟程序

为游戏中的主要系统建立数值基线应该是一个相对较快的过程。因此，很可能在要使用这些数值的系统还没来得及实现之前就已经要建立数值基线了。这时设计人员就需要使用数值模拟程序了。

数值模拟程序是设计人员编写的用于测试数值基线的小程序。因为编写这样的程序所需的时间很短（几天），快速开发语言（譬如说 Python）可以用来编写它们。数值模拟程序简单地获取用户输入的数据；使用一些原型系统算法（prototypical system algorithm）对它们进行处理并且显示结果。

在战斗系统的例子中，一个数值模拟程序读入武器数据、装甲数据和作战者的属性。然后设计人员可以运行这个程序来查看输出的结果，譬如造成的伤害、攻击速度、攻击次数和未击中次数等。设计人员可以只改变一个变量，这样就可以清楚地看到调整个别值所带来的

效果。不仅如此，构成一个对象的属性集可以和构成其他对象的属性集进行比较。譬如说，棍棒和战斧相比有什么区别？一个穿着锁子甲、拿着大槌的作战者和一个穿着皮甲、拿着匕首的作战者交战时会不会占据上风？数值模拟程序使得设计人员可以在战斗系统实现前就给出这些问题的答案。

图 1-4 中是一个战斗系统的数值模拟程序的例子。在这个例子中，战斗仅限于在两个对手之间进行。游戏设计人员可以为每个作战者选择模版，每个模版对应于一组技能。对于“武士初学者”来说，这个模版可能包含了一些关于攻击力、攻击速度以及攻击能力的技能。还可以选择主要武器、次要武器和装甲。注意这些模版、武器和装甲中所用到的游戏数值是通过表 1-1 种所示的“建立数值基线”的方法获得的。虽然作战者的属性是由模版决定的，但是设计人员可以根据需要对它们进一步微调。因为像战斗系统这样的系统通常具有随机性，通常我们需要多次运行模拟程序才能得到可靠的结果。因此，这个战斗模拟程序必须能够对同一组数值进行多次快速运行，并且把数据记录下来以便于进行统计分析。运行模拟程序时，每个战斗事件都会显示在输出窗口内。

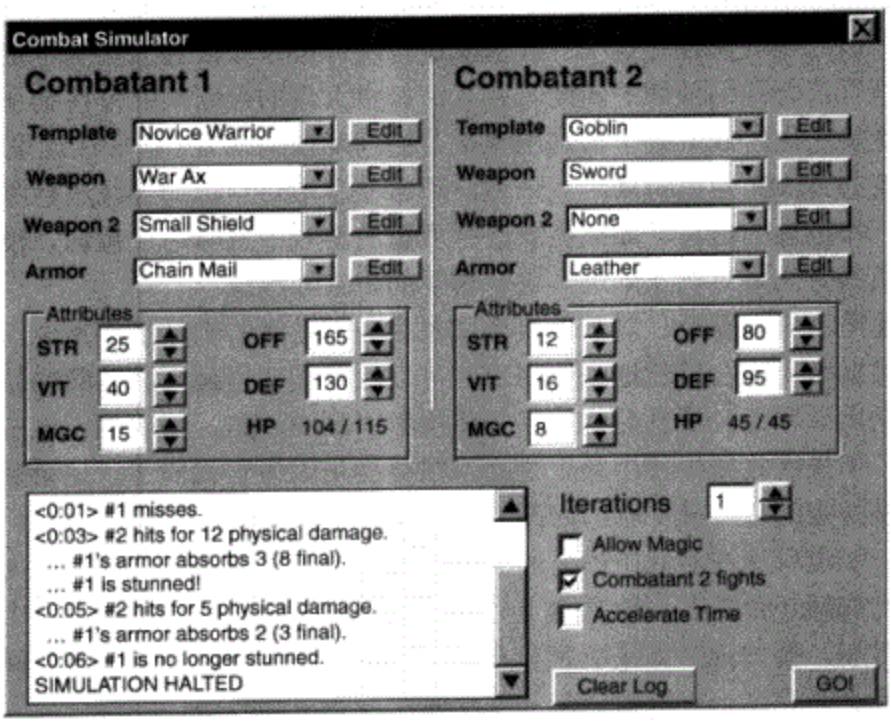


图 1-4　战斗数值模拟程序的例子

在得到模拟结果后，我们总是会不断地调整数值基线以更好地进行平衡。因为像技能、武器和装甲这样的数值会频繁改变，所以最好能够在程序内部修改它们。在图 1-4 所示的例子中，每一项的“编辑”（Edit）按键在按下后都会弹出另一个窗口，设计人员可以在此窗口内察看和调整每一项的数值。

这个战斗模拟程序不必考虑那些可能存在于最后实现中的每一个因素。由于它仅仅对基本的核心功能进行模拟，设计人员可以在进行平衡时获得更快的进展。

数值模拟不仅有助于进行游戏平衡，还具有一些其他的好处。首先，它可以让设计人员更清晰地向程序员演示模拟系统是如何运作的。虽然这并不能代替文档，但是看到一个可以

正常工作的简单实例对理解设计意图有很大的帮助。其次，程序员也许可以在最终实现中直接重用一部分模拟代码，或是对其进行改写后使用，这样可以减少编程所需要的时间。

1.3.3 建立游戏中的度量（Metric）

进入项目中期后，数字模拟程序会被实际的系统实现所代替。在这段时间里，能够对游戏变量进行简单的调整和跟踪是非常重要的。正如在模拟程序中一样，设计人员必须能够快速地查看/编辑游戏中任何对象的数值。然而，与模拟程序不同的是，这时设计人员不可能为了测试而对安全控制游戏环境。游戏受到其他玩家、由 AI 控制的生物以及各种游戏元素的持续影响。因此，建立游戏中的度量变得非常重要。

游戏度量是对所关心的游戏数值进行长期（按天、月、年或特定的时间段来进行维护）跟踪和记录的方法。设计人员可以用这些数据结果来确定，怎样对那些在数值模拟阶段设置的值进行进一步的平衡。下面这些例子说明了 MMP 游戏通常会对哪些因素进行跟踪以及为什么它们有助于进行平衡。

- 玩家升级：对玩家在游戏中升级情况进行统计可以让设计人员确定玩家在游戏中的进展是否符合他们预期的速度。并且，如果大量的玩家都停留在某个特定等级上，就可能反映出游戏中某个特别困难或是特别麻烦的地方（玩家在这里放弃）。对于一个传统的 MMP 角色扮演游戏来说，通常简单地对角色的等级进行跟踪就可以获得关于玩家进展的信息。如果需要进行度量的游戏中没有这个概念，设计人员有必要寻找某些方法来对玩家的进展进行客观的记录和跟踪。
- 物品使用：毋庸置疑，对于检查像武器和装甲之类的物品平衡来说，这是最有用的统计数据。玩家会自然地去使用那些可以让他们变得最为成功的物品。如果一个物品的使用频率比位居其后的物品的使用频率高 100 倍，很明显，这些物品失衡了。同样，如果玩家很少使用某个物品，很可能是因为它太弱了。记录武器的每一次挥动或是特定的装甲被穿着的时间长度可以用来跟踪物品的使用情况。
- 动作：知道玩家在游戏中的时间究竟花在哪里也非常重要。他们把大部分时间花在和怪物作战还是花在制造武器上？通常，缺乏平衡会导致玩家在某个动作上花费过多的时间。譬如说，如果玩家花在治疗上的时间是花在战斗上的时间的两倍，很可能游戏必须提高治疗速度或是让治疗魔法变得更为有效。

通常，游戏设计人员只能对诸多因素中的一小部分进行跟踪。不同的游戏，需要进行度量的因素也各不相同。然而，仔细地考虑怎样进行度量对于在内部和外部测试中的游戏平衡来说是至关重要的。

1.3.4 内部和外部测试

随着项目进入内部和外部测试阶段，游戏中会有更多的并发玩家，这也会影响游戏的平衡。设计人员必须注意在游戏正式运行时，对游戏进行平衡必须考虑所支持玩家的最大数量。譬如说，最初，攻入一个有着很多怪物的地下城看上去难度会很高，事实上当成百上千的玩家一起在里面历险时，它会变得非常容易。每小时只有一个玩家可以获得某个特定的物品在

开始时看上去很合理，但是当成千上万的玩家试图获得它时，却几乎没有人可以得到它。因此，仅当游戏测试者的数量接近于游戏正式运行时所期望的玩家数量时，才可能对 MMP 游戏进行正确的平衡。

在进行这些测试时，设计人员应该把大部分时间花在观察玩家的每一个动作上。早期建立的游戏度量在这个时期是非常有价值的。它们既可以提醒设计人员玩家对于物品、技能和动作的偏爱，也可以提供关于他们在游戏中升级速度的信息。在检查测试数据时，游戏测试人员应该把注意力放在那些异常数据上，譬如说过高的升级速度以及大量的财富。同时也必须用心观察那些玩家花费大量时间的地方。游戏数据和玩家行为中的异常通常可以指出游戏平衡中的错误。

与游戏设计人员的美好愿望相反，内部和外部测试是会发现游戏平衡系统中的缺陷。最常见的错误是总把游戏做得太简单了。永远不要低估玩家的能力，他们进行游戏的方式总比所预期的更为有效。只需要很少的时间，数以千计的玩家就可以对游戏的每个方面进行详细研究，并且找到获得成功的最佳方式。即使有些数据是不可见的，玩家也会通过精确的实验来发现它们，并且把他们的发现发布在网上，让所有人都可以看到。作者曾经遇到一个实例：设计人员在游戏中加入了一个魔杖恢复系统。它的成功率由很多因素决定，但是失败率绝不会低于 1/137。这个系统正式发布后，仅仅过了一个星期，作者发现一个玩家发表了一篇文章，详细地描述了这个魔杖恢复系统。最令人惊讶的是，这个玩家发现了最低失败率："看来有可能会失败。我猜测失败率在 1/150 左右，但是我获得的结果是 1/137。"要知道 MMP 玩家可以通过大量的试验来里确定任何数据。在观察游戏中的玩家行动后，游戏设计人员可以考虑把游戏变得更为困难。

测试期是在游戏中进行试验的最好时机。设计人员可以对游戏做一些修改，然后观察它们对游戏度量的影响。要记住，这是设计人员对他们在数值模拟阶段所建立的平衡数据进行无痛测试的最后机会。在游戏发布后进行修改将会极其复杂。

1.3.5　在发布后对游戏进行平衡

往往事与愿违，在游戏发布后平衡方面也会出现问题。问题在于，应该怎样解决它们。对一个运行中的 MMP 游戏进行平衡是非常危险的。因为每个人都必须使用相同的版本，玩家无法选择接受或是拒绝修改。因此，如果某个修改对某些玩家带来不利的影响，这就可能会导致他们离开游戏。在决定是否必须重新进行某项平衡之前，游戏设计人员应该先回答一个很简单的问题："相对于修复它来说，保留这个失衡的地方是否会导致更多的问题？"如果答案是"是"，那么就应该进行这个修改。

如前所述，最常见的失衡（也是危害最大的）是游戏中的某些系统对于玩家来说过于简单。譬如说，如果玩家发现，通过某种战术来使用特性的武器可以让他们杀死比他们更强大的怪物，战斗系统就会变得过于简单。这样，那些使用这种方式的玩家的升级速度会比其他玩家要快。因为这个技术如此有效，很多玩家会根据它来建立自己的角色，从而使这些角色精通所必须的武器和技能。

降低这种武器或是这种战术的威力，就可以解决这个问题。这样做很简单，游戏设计人员只需要改变一些游戏数值，但是要对这种做法给玩家带来的影响进行处理就是另外一回事

了。毫无疑问，在这些物品和技能上建立自己角色的玩家会反对这种做法。最坏情况下，这些不满的玩家在取消订阅前还会公开表达他们的不快，这不仅会进一步减少玩家数量，还可能导致公共关系上的问题。因此，游戏设计人员必须非常注意重新平衡的副作用。

虽然不太可能在没有任何反对意见的情况下进行重新平衡，但一些事情可以减轻它所带来的问题。首先，在进行修改前游戏开发人员把这些问题告诉玩家。向他们解释这个失衡会对游戏的长期娱乐性造成怎样的危害。虽然这不能彻底地消除抱怨，但是大多数人会愿意接受对这一事实的合理解释。其次，应该创造机会听取玩家的意见。提供一个论坛，使玩家可以表达他们对这些问题的看法并提供建议。虽然只有少量玩家会不厌其烦地以这种方式进行交流，但这样做至少可以在公共关系上带来非常大的好处。最后，尽可能地补偿那些受到影响的玩家。如果重新平衡会对某个贵重的武器带来不利影响，应该允许玩家用它来交换其他武器或是兑换游戏货币。如果重新平衡影响了某个玩家的技能，应该允许玩家使用别的技能来代替它。虽然我们不可能在任何情况下都让每一个人满意，但是主动表示设计人员对此的关心和想法有助于让玩家们保持愉悦的心情。

1.3.6 对新功能进行平衡

通常游戏开发人员会在游戏发布后加入新的功能。虽然这些新功能有助于保持甚至增加游戏的趣味性，但是和重新平衡一样，加入新功能也会带来风险。即使一个看上去非常小的功能，在和其他游戏系统互相作用后都可能带来很严重的后果。玩家会很快发现它所引起的任何新漏洞，这可能会让游戏中某些重要部分失衡。实际上，即使添加一个很小的功能也必须经过广泛的测试以避免失衡。

测试应该通过一个测试服务器来进行。简单地说，这是正在运行的游戏的一份拷贝，只不过它还包含新的功能。设计人员应该允许玩家访问这个测试服务器并且鼓励他们使用它。这样，设计人员就可以安全地调节游戏数值，从而降低这个新功能发布后对游戏带来不良影响的可能性。

1.3.7 总结

在 MMP 环境中进行游戏平衡，是一个在开发早期就应该开始的工作。首先我们要为游戏数值建立基线。这些数值通过数值模拟程序被进一步改进。接着，我们要在内部和外部测试中通过游戏度量对这些数值的作用进行观察和跟踪。对游戏进行平衡并不会随着游戏的发布而停止，因为新的问题会不断出现。在游戏发布后对其进行修改时，我们必须考虑到这些修改对现有玩家的影响。这对加入新功能来说同样适用。通过在游戏开发的每个阶段对其进行平衡，我们可以让游戏顺利地发布和运营。

1.4　使用支付矩阵来设计游戏平衡和 AI

John M. Olsen，Microsoft
infix@xmission.com

游戏平衡是一个非常棘手的问题，游戏设计人员必须投入大量时间才能获得正确的结果。在 MMP 游戏环境中，任何错误都会引起重大问题。玩家会通过合作来发现任何微小的优势并且加以利用，他们在这方面的能力超乎寻常。在这样的环境下，几乎每个游戏都有它独特的游戏平衡方法。本文介绍了怎样使用统计分析和一些简单的数学方法来简化游戏中特定部分的平衡工作，这些技术也可以用来确保 AI 程序利用了所有有利条件。

1.4.1　什么是支付矩阵?

支付矩阵（Payoff Matrix）是一个博弈论概念。博弈论主要是在约翰·冯诺依曼和奥斯卡·摩根斯坦[Neumann44]的研究基础上发展而来的。这种矩阵用来表示博弈中两个或更多的参与者在做出不同决定时所获得的收益或惩罚。在博弈论中，一次“博弈”是指这样一个竞赛：在一次或多次对抗后，一个参与者获胜而另一个失败。可以根据参与者所做的选择从这个矩阵中得到他们的分数。

首先，看一下在图 1-5 中玩家 1 身上发生了什么。可以通过在左边的列中查找他的选择并且在表格上方查找对手的选择来得到他的分数。

再来看看玩家 2 的情况。图 1-6 是另一个版本，它包含了玩家 2 的分数。

现在对它们做一些简化，把这些矩阵组合在一起，使用一些颜色编码来同时显示两个参与者的分数。

要使用图 1-7 中的矩阵，需要获得两个玩家的选择并且在矩阵中查找相应的单元。第一个数字是玩家 1 的分数，第二个数字是玩家 2 的分数。玩家标志中的阴影和他们在每个单元中所获得的分数的阴影一致，这使得表格的可读性更强。

	玩家2选择1	玩家2选择2
玩家1选择1	玩家1获得0分	玩家1获得-1分
玩家1选择2	玩家1获得1分	玩家1获得0分

图 1-5 玩家 1 的回报矩阵

	玩家2选择1	玩家2选择2
玩家1选择1	玩家2获得0分	玩家2获得1分
玩家1选择2	玩家2获得-1分	玩家2获得0分

图 1-6 玩家 2 的回报矩阵

	玩家2选择1		玩家2选择2	
玩家1选择1	0	0	-1	1
玩家1选择2	1	-1	0	0

图 1-7 玩家 1 和玩家 2 回报矩阵的组合

图 1-7 所示的矩阵还有一个额外的特征。如果把每个单元中的两个分数加在一起，和都为 0。这是一个零和博弈，它确保在每个回合中，如果有一个胜利者，就必然有一个失败者。同时也确保在经过一系列的竞赛之后，玩家 1 的分数是玩家 2 分数的相反数。

1.4.2 “War”纸牌游戏

War 是一种非常简单的古老游戏，它使用一副纸牌来进行。每个参与者取出他们手中纸牌最上面的一张。牌大的玩家可以得到这两张纸牌，如果是平局，那么它们由下一轮的胜利者获得。图 1-7 可以用来表示该游戏的一个简化版本。每个参与者可以出一张小牌或是一张大牌。如果玩家 1 出大牌（选择 2），玩家 2 出小牌（选择 1），那么玩家 1 获得一分，玩家 2 失去一分。如果他们都出小牌或者都出大牌，那就形成平局，没有人得分。

这是一个经过大量简化的例子，只是为了让读者知道这种矩阵是怎样应用于一个熟悉的游戏中的。下面本节会讨论一些更复杂的问题和用法。

1.4.3　囚徒困境（Prisoner's Dilemma）

囚徒困境是两个罪犯被警察逮捕后所进行的博弈。如果他们都不招供，他们面对的指控是无力的，因此他们每人会被判入狱一年。如果其中一人招供并且作出不利于另一人的证供，他所得到的奖励是只需入狱服刑 3 个月，而另一个罪犯将会入狱 10 年。如果两人都招供，那么他们每人都需要入狱 8 年。每个人都必须在不知道对方决定的情况下，决定是招供还是保持沉默。

这个困境的矩阵可以帮助游戏设计人员理解所发生的事情，如图 1-8 所示。

	参与者2保持沉默		参与者2招供	
参与者1保持沉默	-1	-1	-10	-0.25
参与者1招供	-0.25	-10	-8	-8

图 1-8　囚徒困境的支付矩阵

很容易看到，对于参与者 1 来说，最好的情况是他招供而参与者 2 保持沉默。问题在于参与者 1 并不知道参与者 2 会怎样决定。参与者 1 必须根据参与者 2 可能作出的对他最不利的决定来作出他的决定（译者注：这里参与者 1 这样做选择的原因其实是因为他和参与者 2 都知道他们在各种选择组合下的结果，因此他知道无论参与者 2 做出什么选择，他选择招供总比选择保持沉默获得的好处要大，所以选择招供对他来说是明智的，请参考相关博弈论书籍以获得更详细的解释）。如果参与者 1 保持沉默，可能发生的最坏情况是入狱 10 年。如果他招供了，最坏的可能是入狱 8 年。

两个参与者都招供的状态具有一个有趣的特性，那就是在另一个参与者没有改变策略的前提下，任何一个参与者都不会选择离开这个状态。这被称为纳什均衡[Nash50]。如果一个参与者在其他参与者没有改变主意的情况下离开这个状态，也就是说他选择保持沉默，那么他会获得两年额外的徒刑，因此没有一个参与者会选择独自离开这个状态。

支付矩阵是表示这些状态和行为之间关系的一种紧凑形式，这里每个参与者在作出自己决定时都需要考虑其他参与者的行为。

1.4.4　简单的格斗游戏

现在本节开始讨论更加“动作化”的问题。图 1-9 是为一个简单的格斗游戏设计的支付矩阵，其中每个玩家可以选择攻击、格挡或是休息。这个矩阵并没有表示出游戏模式中的某些机制，譬如说每个状态的持续时间，本节将在后面做更详细的讨论。

这个矩阵被刻意地设计为每个状态都会对分数进行一些改变，这样可以使游戏变得更富于变化。矩阵显示了一次成功攻击的结果、格挡所付出的代价以及休息带来的好处。

这个矩阵中还可以提示一些有趣的游戏方式。如果一个玩家能够和对手保持一段距离，他就可以从休息中获益。但是一旦他的对手再次接近，这种好处将不复存在，因为在遭受攻击时，处于休息状态会带来更大的伤害。总是保持格挡状态的玩家将会受到惩罚，因为格挡者会不停地失去点数而对手可以在不失去任何点数的情况下持续攻击。所有这些特性都倾向于让游戏可以结束而不是陷入僵局。

	玩家2进攻		玩家2格挡		玩家2休息	
玩家1进攻	-10	-10	0	-1	0	-20
玩家1格挡	-1	0	-1	-1	-1	1
玩家1休息	-20	0	1	-1	1	1

图 1-9 一个简单格斗游戏的支付矩阵

对这个矩阵进行进一步的分析后，通过在每行每列寻找最高分数，设计人员可以发现一个纳什均衡，并且消除那行或那列所有其他的结果。从被消除的单元画一个到这行或这列最优值的转换箭头就可以说明这个现象。如果对手在攻击，最好的选择是保持格挡。纳什均衡存在于任何没有外向的转换箭头的单元中，最后只剩下两个玩家都休息这个均衡状态。这些转换如图 1-10 所示。

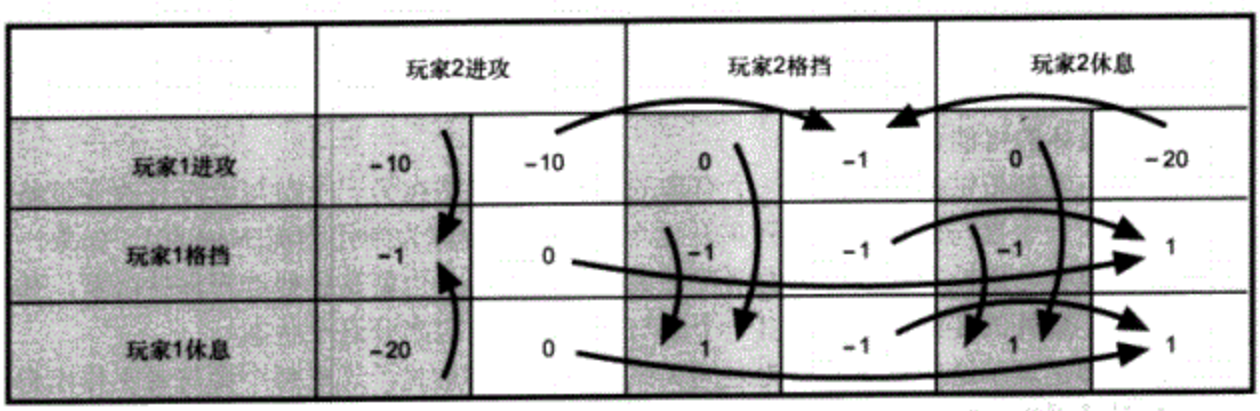


图 1-10 表示怎样达到纳什均衡状态的有向支付矩阵

这个结果比囚徒困境中自相矛盾的方案要好得多。“要想大家都活命，就应该一起休息”，这一结论很有意义。

一个任意对称矩阵（按照对角线对称的矩阵）如果含有一个不在主对角线上的均衡状态，这个状态及其包含的数据都会被映射到主对角线的另一边。这意味着如果有一个不在主对角线上的均衡状态，那么在主对角线的另一边必然有一个一致的对称状态。

现在可以知道最佳选择是什么：如果两个玩家都想活命，那么就一起坐下休息。这个结果告诉设计人员必须对这个状态进行特别观察以防止被滥用。幸运的是，游戏中的玩家并不

是以活命为目的的，他们还需要杀死对手。那些没有包含在矩阵中的游戏特性（譬如移动、位置和其他的游戏目标）可以用来打破这个均衡状态，这样可以避免游戏的结果被轻易地预测到。

必须注意的是纳什均衡并不考虑一个序列中的多个转换。即使有两个或者多个均衡状态，也不能在同一个回合中因为某一个玩家的决定而在它们之间变化。这意味着矩阵中没有一行或者一列可以有多于一个的均衡状态，所以均衡状态数的最大可能值是行数和列数这两者中较小的那个。然而，如果两个玩家同时改变状态，还是有可能在均衡状态间移动的。

玩家可能会故意地承担从一个均衡状态移出所受到的惩罚，从而通过在一系列可预测的状态中移动最终达到一个战略上更有利的状态。这类有条件的均衡也可以使进行游戏的方式变得更为复杂。

只要矩阵是对称的，也就是说所有的玩家具有相同的能力并且受到同样的惩罚，就不会产生有关于玩家能力平衡上的问题。因为所有的玩家具有相同的能力，一个玩家无法获得比另一个玩家更多的内在优势。当然，一些和整体可玩性有关的平衡问题，譬如说有些玩家会认为某个设置过于愚蠢，这或多或少地困扰着每一个游戏设计。

1.4.5　不对称的能力

一旦玩家的能力不同，游戏设计人员就需要对玩家进行平衡，这是最具挑战性的任务之一，因为这个问题涉及的范围可能很大。如果玩家的能力、等级和类型差异很大，并且还有一些具有不同能力和特征的 NPC（计算机控制的非玩家角色），所需测试的组合很快就会成为天文数字。

MMP 游戏，可以把测试范围划分为特定的领域。首先，保持玩家之间的平衡，这样某个特定的职业或类型就不会在游戏中处于支配地位。其次，在进行完玩家之间平衡后，要对玩家和 NPC 之间进行平衡也就更容易了。

图 1-11 是一个玩家与玩家间的测试，图中两个玩家的能力不一致。玩家 1 进攻可以造成更大的伤害，同时也需要更多的体力来格挡。这里仍然模仿图 1-9 中所描述的格斗游戏进行建模。

因为他们的能力不同，很可能某个玩家会具有一些优势。在试图为有区别的玩家类别建立平衡时，游戏设计人员需要密切注意失衡的情况。

	玩家2进攻		玩家2格挡		玩家2休息	
玩家1进攻	-10	-15	0	-1	0	-30
玩家1格挡	-2	0	-2	-1	-2	1
玩家1休息	-20	0	1	-1	1	1

图 1-11　为一个玩家与玩家间的测试建立的不对称支付矩阵

1.4.6 延迟和停止

现在是时候加入更多的复杂度了。从一个单纯的回合制系统演变到一个实时游戏会带来一些新的要求。第一步是定义每个状态所需最少和最多的时间，然后为每个状态添加一个转换表，这个转换表会显示在这个状态完成后可以进入哪些状态。现在试图对这个不对称矩阵进行平衡，加入一些延迟可以来抵消玩家 1 过强的攻击能力。图 1-12 显示了和前面一样的数据，但是它在每个状态中加入了时间范围以及可以进行合法转换的状态列表。这个矩阵加入了一些限制：不能重复同一个状态，并且攻击后必须休息。

	玩家2进攻 (0.3秒后进入休息状态)		玩家2格挡 (0.1到2.0秒后进入休息或进攻状态)		玩家2休息 (0.3到无穷大秒后进入进攻或格挡状态)	
玩家1进攻 (0.5秒后进入休息状态)	-10	-15	0	-1	0	-30
玩家1格挡 (0.2到2.0秒后进入休息或进攻状态)	-2	0	-2	-1	-2	1
玩家1休息 (0.3到无穷大秒后进入进攻或格挡状态)	-20	0	1	-1	1	1

图 1-12 加入时间延迟和允许的转换

这个矩阵说明了怎样用速度来抵消玩家 1 的力量优势。如果他能够造成更大的伤害，那就应该让他更慢一点。考虑每秒的最大伤害，现在玩家 1 可以以 0.5 秒来进攻，然后休息 0.3 秒或更长时间，接着再重复进攻。这使他最多可以以 30/0.8，也就是 37.5 点每秒的速度进行进攻。玩家二可以以 0.3 秒来进行更迅速的攻击，然后休息 0.3 秒或更长，这使得他可以以 20/0.6，也就是 33.3 点每秒的速度进行攻击，这意味着他们之间的差距只不过略大于 10%。虽然攻击力较强的玩家仍然具有一定的优势，但是时间上的要求把它大大地削弱了。

对合法的转换进行限制也增加了游戏的策略成分。如果玩家 1 正在进攻而玩家 2 正在格挡，那么玩家 2 就可以知道如果玩家 1 没有及时地从接下去的休息状态转移到格挡或是攻击状态，玩家 2 就有机会击中他。

前面本文简要地提到过一个状态间无法转换的情况：由于玩家不能在有效范围以外进行攻击，这限制了可以转换的状态。必须注意的是任意具有最大时间限制的状态都需要有一个允许的转换。如果有一些状态仅仅持续 1 秒并且要求紧接着一个进攻，那么在需要进攻时他在攻击范围之外就会遇到问题。要解决这个问题，玩家可以在任何时候都允许转换到一个在所有情况下都有效的状态，或是为攻击状态创建副本，譬如说“挥动武器但是未击中”，这样就可以在不能进行任何攻击时使用这个状态。

1.4.7 自动化

有两种方法可以把这些关于矩阵和均衡的理论应用到实际游戏中去：自动化的矩阵测试、

记录模拟战斗或游戏中的战斗日志。设计人员只需要一点创造力就可以把任何战斗系统映射到一个支付矩阵中。矩阵的左边是一个战斗参与者可用的选择，矩阵的上边是另一个战斗参与者可用的选择。因为战斗系统可能允许可变伤害或是未击中的情况，矩阵中的支付值不一定是一个固定值。本文所介绍的方法也适用于可变支付的情况。

支付矩阵可以看成两个相交的状态机。在前面介绍的支付矩阵中，玩家 1 的状态机沿着左边显示，玩家 2 的状态机沿着上方显示，这两个状态机的交集就是支付信息。改变动作（也就是从一行移动到另一行或是从一列移动到另一列）就会导致状态转换。一个玩家控制列状态的变化，另一个玩家控制行状态的变化。

现在，只要选定两个敌对者然后识别他们可以选择的动作列表，就可以为他们建立一个矩阵。对当前合法的目标状态进行估价，然后从中选择一个合法的新状态，这样就可以构造出一个复杂的 AI 系统。这个系统最复杂的部分是估价函数，它必须考虑 AI 的当前状态以及每个合法行为的支付。

在一个幻想游戏中，支付可以定义为在使用当前盔甲和武器的情况下，攻击可以造成 X 点的伤害；休息可以治愈 Y 点的伤害；施展某个魔法可以使对手暂时麻痹从而在一定程度上降低它的攻击力。一旦从估价函数获得这 3 个选项的不同结果，就可以从中作出选择。

要在这些选项中作出选择，加权随机数可以用来创建一个简单的模糊逻辑系统。关于模糊逻辑系统更详细的信息可以参考[McCuskey00]和[Dybsand01]。如果估价结果显示不同选项所带来的好处是 7、5 和 8（数字越大表示这个选项越好），就可以使用一个在 1 到 20（7+5+8）之间的随机数在它们之间做出选择。这样做可以使得所做的选择一直在变化，并且选择有些行为的可能性会变小，这可以让游戏变得更有趣。任意好处为 0 的选项在使用这种方法时都会被自动忽略。一旦生成了随机数，在 1 和 7 之间的值对应于第一个选项，在 8 到 12 之间的值（包含 5 个数的区间）对应于第二个选项，在 13 到 20 之间的值（包含 8 个数的区间）对应于最后一个选项。

很多另外的方法都可以在多个有效单元的支付值中进行选择，但是它们的效果并不都很好。通常，如果对手因为某个动作比其他选项略微有利一些就一直选择这个动作的话，玩家会觉得很无趣。

正如前面提到过的，设计人员应该对矩阵中每个从当前单元中合法转换的单元进行估价。这就回到了图 1-12 中所示的时间限制和转换要求。如果仔细地选择操作的顺序，就可以为 AI 节约一些 CPU 时间。

设计人员可以先对当前是否存在可行的转换进行检查。如果没有，就可以在这次估价中忽略整个 AI 系统。接着可以过滤掉那些在当前情况下不允许的行为。

估价函数本身高度依赖于游戏，但是设计人员通常希望可以基于下面的条件对当前情况进行估价：耗费或是获得的资源、当前状态（包括生命值）的增加或减少以及这些动作对朋友和敌人的能力所带来的影响。

一旦建立了估价函数，它就可以自动化地进行平衡和测试过程。通常，游戏中存在大量由群体和个体组成的独特组合，这对于测试来说简直是噩梦。这使它成为实施自动化的理想对象。

如果我们只在相当少的 AI 类别间进行测试，即使使用一个自动化的轮流对抗使每个类别都和所有其他类别进行对抗也很简单。在重复几次这样的对抗后，记录将呈现出一定的趋势，它可以告诉我们这些不同的对手之间是否平衡。通过对比不同类别的胜利次数，游戏设计人员可以知

道相对其他类别来说，与这个类别作战的难度是多少。图 1-13 中每个单元都保存了矩阵左边的玩家相对于矩阵上方的玩家的胜利数。玩家不会和自己进行比赛。在 20 轮对抗后，通过比较胜利总数，可以知道玩家 1 最好而玩家 3 最差。如果希望这些玩家处于大致相同的等级，那么矩阵中的数字说明了还需要作一些调整，以提高玩家 3 的能力并且略微降低玩家 1 的能力。

千万不要以为这样的轮流测试就足够了，因为此时游戏完全依赖于尚未完善的 AI 代码。玩家可能会发现一些漏洞，这会使得游戏必须更新所有的日志数据。

	玩家1	玩家2	玩家3	总胜利数
玩家1	0	8	15	23
玩家2	12	0	9	21
玩家3	5	11	0	16

图 1-13 20 次对抗后的胜利结果示例

设计人员一旦可以对 AI 控制的对手之间的回合制战斗进行记录，就应该开始记录 AI 对手和玩家控制的对手之间的战斗结果。坦白地说，这部分工作应该在第一次使用战斗代码时就开始，并且直到游戏最终被关闭前一直进行。随着玩家的不断成熟以及游戏的不断更新，在玩家和 AI 间进行调整必须是一个持续的过程。这不仅非常复杂，还很棘手。每当设计人员根据玩家和 AI 间的交互性测试对支付矩阵做出任何修改后，他们必须对这些修改重新进行前面所描写的 AI 对 AI 的自动化测试，这样 AI 就可以通过学习来弥补玩家所发现的漏洞从而变得更为精确。

一对一的记录是最容易进行的，但是 MMP 游戏的主要目的之一就是倡导团队合作。通过计算团队间（无论这些团队是否是 AI 控制的）的数据，我们可以获得全新的数据。数据的复杂度会很快增长，并且具有大量的可能组合。对可能组合进行过滤，从而获得有用组合的方法完全依赖于具体的游戏设计。这通常会从游戏中可以预测的或是已观察到的情况开始，当注意到记录文件中有奇怪的事情发生时，再把它们也加入。

当数据中某个特定的 AI 类别或是玩家类别/群组在战斗中胜利或失败的次数过多时，很多方法都可以对它加以解决。查找历史记录的趋势或是进行一些自动化测试可以决定矩阵中的哪个支付值需要被调整，简单地通过观察用户的行动也可以做出这个决定。当发现某些地方效果不是很好时，只需要改变攻击方式的使用频率或是调整行为之间的延迟就可以获得完全不同的结果。

1.4.8 总结

令人惊讶的是，纳什均衡以及相关的博弈论知识通常应用在与商业和经济相关的模拟中，而不是用在互动娱乐产业中。Paul Walker[Walker01]整理了一份博弈论的简要历史。通过对本文讨论的问题的理论基础加以利用，设计人员可以对系统的行为进行更严格的定义，而采用

统一的方法进行设计也会更加方便。在使用非正式方法的系统中，随着测试在最初的设计中发现更多的漏洞，必须对它进行不断地扩展和改变，与之相比，使用上述方法进行的设计更容易被实现。

从某种观点来看，所有这一切只不过是在做那些总是要做的事情。检查自己有哪些选项，接着在那些有意义的选项中做出选择；然后通过对玩家或 AI 有利的情况进行查找来发现那些相持状态（static state）或是可能存在麻烦的地方。

本文所描述的方法最主要的优点，在于它使用易于理解的格式来对非常复杂并且不可靠的设计进行组织。这有助于使用一致的方法对 AI 和战斗系统进行创建、分析和调整，并且在获得稳定性的同时避免过多的限制和开销。

1.4.9 参考文献

[Dybsand01] Dybsand，Eric，“A Generic Fuzzy State Machine in C++,” *Game Programming Gems 2*，Charles River Media，2001.

[McCuskey00] McCuskey，Mason，“Fuzzy Logic for Video Games,” *Game Programming Gems*，Charles River Media，2000.

[Nash50] Nash，John F.，“Equilibrium Points in N-Person Games,” *Proceedings of the National Academy of Sciences of the United States of America*，1950: pp. 36，48–49.

[Neumann44] von Neumann，John，and Oskar Morgenstern，“*Theory of Games and Economic Behavior,”* Princeton University Press，1944.

[Tucker50] Tucker，A. W.，“On Jargon: The Prisoner's Dilemma,” *UMAP Journal*，Vol. 1，No. 101，1980.

[Walker01] Walker，Paul，“History of Game Theory,” http://www.econ.canterbury.ac.nz/hist.htm，May 2001.

1.5 使用用例来描述游戏行为

Matthew Walker，NCsoft Corporation
mwalker@softhome.net

现代的 MMP 游戏项目都非常庞大，涉及到很多具备不同技能的参与者。让一个人同时进行游戏设计和程序编写是非常少见的。这就对专业化的需求提出了新的挑战：怎样在设计人员和程序员之间进行关于游戏需求的交流。游戏包含了实体之间复杂的交互，需要复杂的状态管理和事件处理机制。设计人员和程序员必须把游戏设计中的创造性思想系统地映射到一个技术性的设计模型上，才能使最终实现表达出最初的意图。用例（Use Case）为此提供了一个有效的方法。

1.5.1 什么是用例?

用例使用自然语言以一个简单的结构为系统行为定义了模型，设计人员和程序员可以把它作为共同的参考。用例的主要功能之一就是把一个难以处理的大型系统规范地分解成可管理的较小的部分。这些较小的部分更容易描述，因此也更容易实现。

每个用例都代表了一个离散的行为单元，它具有定义清晰的作用域、清楚的步骤以及明确的前提条件和后置条件。具有清晰作用域的用例明确了特定责任应该在哪里实现，这有助于游戏设计人员开发一个稳健的模块化系统设计。通过使用清楚的步骤来描述行为可以表明实现用例目标所需的细节，这些步骤最终可以映射为程序代码。对每个用例的前提条件和后置条件进行约定，不仅可以确保系统状态在进入和退出这个用例时都处于已知的状态，还可以指出用例之间的依赖关系。

1.5.2 为什么要使用用例来开发 MMP 游戏?

目前 MMP 项目的开发团队通常都非常庞大，设计人员、美工和程序员必须定期进行明确的交流来协调产品的创作过程。这样的交流大都是在技术水平参差不齐的人员间进行的。通过在用例中使用简单而结构化的自然语言，这样设计人员和程序员无需对各方的技术理解能力作太多要求就能有效地降低二义性。设计人员和程序员可以合作为游戏创建一个相当稳健的描述，而不必受行话或技术细节的困扰。随后，程序员可以把用例的描述转换成代码，这比使用文本形式的游戏设计文档更简单，也更容易使

程序员理解设计人员的需求。

1.5.3　怎样编写用例?

自从 Ivar Jacobson 在 20 世纪 90 年代早期[Jacobson92]提出用例这一概念后，它就一直被使用在游戏产业以外的软件开发中。从那以后，很多作者撰写了关于用例的著作，也提出了很多不同的方法来编写用例，这其中有非常正式的、具有交叉索引的层次性文档，也有简短而一致的模块化功能描述。

本文的目的是要介绍一个经过实践证明、一致而清晰的方法来描述怎样在 MMP 游戏开发中使用用例的思想。本文所介绍的技术将使用简单的可重复结构，基于日常词汇的简明术语以及无二义性的语句。

1.5.4　如何识别用例?

组成用例的行为有时很难识别。用例从哪里开始？在何时结束？这是在一开始就必须研究的问题。对角色和事件的识别可以帮助游戏开发人员做到这点。

1. 角色

角色是展示行为的实体，它们既可以在所开发的系统内部进行交互，也可以和系统进行交互。角色导致用例的产生。识别角色有助于识别用例是从哪里开始的。表 1-2 是一份具有角色描述的简单列表，应该足以保证完整性。

角色通常处于系统的外部。用户或其他软件程序是常见的角色例子。这种观点在开发用户界面和游戏中的非游戏方面时非常有用，它也是传统软件工程项目中最常见的观点。

然而，游戏开发中还有一个非常重要的概念，那就是角色本身也是游戏环境中的一个实体。很多游戏都是对现实中某些方面的开放式模拟，在这些模拟中，实体往往以复杂的方式进行互动。在这种情况下，与其把玩家本身作为一个角色来建模，还不如把玩家所控制的人物作为一个角色来建模更为有用。很明显，在这一层次上，把独立的由计算机控制的实体，像怪物、NPC 以及车辆作为角色处理是有益的。使用这个方法可以把涉及多个实体的复杂行动链打破为成对角色之间的离散交互。

表 1-2　　常见 MMP 角色扮演游戏中的角色

角色	描述
玩家角色（PC）	玩家在游戏中的代表
非玩家角色（NPC）	计算机控制的实体，它和 PC 有很多共同的特征，包括能够聊天、交易、战斗以及和 PC 一起加入群体
怪物	计算机控制的实体，它只能与 PC 或 NPC 战斗

2. 事件

从总体上看，用例是由角色发起的事件继承而来的[Fowler99]。对这些事件进行列表和描述是一个开始识别用例的有效方法。将游戏中可能发生的事件简略地一一列出来可以对用例集合进行整体思考。

为了做到这点，游戏开发人员需要识别出产生事件的对象，并且为每个事件提供一句话的描述，如表 1-3 所示。使用一句话来进行描述的原因很多。首先，一句话比较简短，不仅适合放在表格中，还有效地保证了可读性。这就有助于在和团队成员讨论开发时通过名字来对事件加以记忆和引用。第二个原因是，使用一句话来描述引入了一个容易遵守的规则：一个事件是一件单独的事情；一个事件可以用一句话简单地描述。如果所发生的事情需要两句以上的语句来描述，很可能是在对多个事件进行描述，以这种方式把多个事件组合起来很难以让人理解真正发生的事情是什么。

表 1-3 角色扮演游戏中一个带有描述的事件列表的例子

时间	描述
PC 攻击怪物	PC 瞄准怪物并使用当前装备的武器进行战斗
PC 对怪物使用魔法	PC 瞄准怪物并对它施放一个伤害性的魔法
PC 和 NPC 进行交易	PC 从 NPC 那里购买物品
PC 治疗 PC	PC 对另一个 PC 施放治疗魔法
怪物攻击 PC	怪物瞄准 PC 并使用当前装备着的武器进行战斗
PC 装备武器	PC 从背囊中取出一个武器并把它放入武器槽内

1.5.5 用例中的元素：一个标准模版

既然已经发现了可以发起用例的事件，就必须建立一种方式来描述它们。游戏开发人员可以使用一个标准的模版来捕捉用例中的关键元素。

用例是对系统行为单元的结构化文本描述。这样的结构有助于保持清晰性和一致性，因此最好使用某种形式的标准模版来实施。这里所需要的形式必须既简单又完整，同时还要能方便地完成从白板到纸张到字处理软件之间的转录。在图 1-14 中所示的模版中，每一个字段都表示一个关键的用例元素，同时包含了一个简短的提示来表明其目的。这一节的余下部分会对每个元素进行详细描述。

1. 标题

用例之所以能够毫不费力地被识别出来是因为它们都有各自的标题。一开始，游戏设计人员可以简单地使用用例所继承的事件的名字作为标题。随着用例逐渐成形，它可以演变成某些更合适的描述。譬如说，从上述事件列表中派生的用例的标题可能是：

玩家角色攻击怪物

标题：*[用来标识这个用例的简短描述。]*

描述：*[一到 3 个句子，用来对用例进行总结并且定义它的作用域。]*

基本过程：*[列出执行用例所需的关键步骤。最小化条件逻辑，并且需要时引用其他用例。]*

扩展：*[列出采取其他步骤的条件。]*

前提条件：*[列出这个用例能够正确执行所需要的条件。]*

后置条件：*[列出这个用例正确执行后的结果条件。]*

注释：*[额外的信息和/或脚注，用来添加那些如果放在其他部分会导致混乱的细节。]*

图 1-14　推荐的用例模版

2. 描述

用例的总体描述有两个作用。首先，它对用例进行概括，因而可以作为一个摘要使用在任何像状态报告、计划文档或备忘录这样的概括性交流中。其次，也是最重要的一点，它定义了用例的作用域。描述应该只包含这个用例所实现的行动，而必须把那些发生在其他地方的行动显式地排除在外。这并不是说描述必须详细地描写用例中所有的行动。而是说，它必须为所要发生的行动设置一个边界。基本上，描述必须从发起这一用例的事件开始，并且结束于对这一事件的某个响应。我们用例的描述可以是：

玩家角色使用当前武器攻击怪物；怪物受到伤害。

注意这两句简单的描述说明了几件事情。玩家是一个角色；攻击是一个行动；怪物是目标；结果是怪物受到伤害。这个用例小心地避免了对怪物怎样报复或是怪物是否因为受伤而死去进行讨论。那些概念最好还是在独立的用例中进行讨论。

3. 基本过程

用例的核心是它的步骤列表。这些步骤描述了游戏开发人员想要实现的行为细节。它可以用来表示动作发生的次序、是否有重复以及是否存在条件逻辑。步骤列表的第一步应该和用例描述的开始部分一致，最后一步应该能够反映出描述中提到的最后概念。这样就可以确保这些步骤集合一直处于描述建立的作用域中。如果某一步骤处于用例的作用域之外，就需要删除这一步（把它放到其他用例中），或是重新定义用例的作用域。符合本文先前描述的作

用域的步骤列表可能是这样，如下。

基本过程：
（1）玩家角色瞄准怪物；
（2）玩家角色使用当前武器攻击怪物；
（3）计算攻击的成功率；
（4）计算伤害值；
（5）将伤害作用在怪物身上。

现在用例开始成形了。它仍然非常简单，只有5个步骤。按照Jacobson的说法，最初的步骤集合称为基本过程[Jacobson92]。现在，更多关于用例的细节已经被发现了。为了攻击一个怪物，游戏设计人员必须先把怪物指定为一个目标。为了获得更有趣的游戏模式，他们或许希望加入一些攻击失败的可能性。最后，怪物受到的总伤害可能依赖于很多变量，因此可以引入一个步骤来计算这个值。在此，本文也向读者暗示了可能存在其他的行为，它们会带来额外的复杂度。

4. 从一个用例中引用另一个用例

为了维持完整性和可读性之间的平衡，在编写用例步骤时，程序员使用了一定程度的抽象。用例的可读性较高是因为它只有较少的步骤，并且从当前的抽象层次来看，它是完整的。当然，在编写游戏代码时，程序员需要知道一些额外的细节，譬如说计算伤害值究竟是什么意思。

简单地在用例中加入额外的步骤就可以描述这些细节，这些步骤可以被放在目前第4步的位置上。但是，这有可能会降低用例的可读性。另外，游戏开发人员可能会在这个用例所描述的情形以外的地方计算伤害值，玩家可能会掉进一个装满强酸的大桶，被火烧伤或是掉入一个陷阱。

要描述如何计算伤害大小的细节，一个更好的方式是为它写一个独立的用例，并且在当前的用例中对新用例加以引用。新用例可能会像下面这样。

标题：计算伤害值
描述：计算目标受到的伤害，包含任何与基本攻击以及防御值有关的修正。
基本过程：
（1）获得攻击武器的基本攻击值（Attach Value）A；
（2）获得攻击者装备的攻击修正值（Attack Modifier）AM；
（3）获得攻击者对于这个目标类型的特殊攻击修正值（Special Attack Modifier）SAM；
（4）获得防御者的基本防御值（Defense Value）D；
（5）获得防御者装备的防御修正值（Defense Modifier）DM；
（6）获得目标对于攻击者的特殊防御修正值（Special Defense Modifier）SDM；
（7）使用等式1-1中所示的公式计算伤害值DAMAGE。

$$DAMAGE = ((A + AM) * SAM) - ((D + DM) * SDM) \qquad \text{（等式 1-1）}$$

现在，修改玩家角色攻击怪物这一用例的步骤，在第 4 步中加入计算伤害值用例的标题

基本过程：
（1）玩家角色瞄准怪物；
（2）玩家角色使用当前武器进行攻击；
（3）计算攻击的成功率；
（4）**插入：计算伤害值；**
（5）将伤害作用于怪物。

这样第一个用例保持了和之前一样的可读性而且以另一个用例的形式捕捉到了额外的信息。如果这个新用例描述了一个在游戏的其他方面也用得到的公共行为，那还可以在其他的用例中引用它，正如在这个用例中所做的。新用例和引用它的用例之间的关系被称为使用（uses）关系[Jacobson92]。

5. 扩展

术语“扩展（extension）”用来描述用例中的一些步骤，这些步骤的行为会随着某些条件的改变而改变。扩展包括了错误处理和非异常状态下的分支逻辑。为了清楚起见，游戏的程序员应该把基本过程中的步骤以不带有任何扩展的直接序列方式列出。这样，阅读用例的人就可以很方便地知道在一般情况下会发生什么。当然，编程中有很多工作与怎样处理条件分支有关，那应该怎样表示这些情况呢？这完全由个人习惯决定，主要的方针就是在保持用例可读性的同时，最小化管理这些用例所需的整体代价。

传统的做法是为每个扩展做一个用例并且在主用例中引用它。引用它的主用例和这个新用例之间的关系是扩展（extends）关系[Jacobson92]。这个方法有些矫枉过正，因为很多扩展只需要一到两步就可以对条件行为进行描述。然而，如果一个扩展需要使用很多步骤来描述一个比较详细的执行过程，并且具有良好定义的作用域，那么创建一个新的用例则更为合适。

对于简单的扩展，譬如说检查错误并停止执行或是结束一个循环，最佳的方案是把这个扩展作为主用例的一部分。怎样完成这点只是一个风格问题。另一个方法是简单地把这个条件作为正常序列的一个步骤。

下面是玩家角色攻击怪物的基本过程。已经加入了内嵌的扩展。

基本过程：
（1）玩家角色瞄准怪物；
（2）玩家角色使用当前武器进行攻击；
（3）计算攻击的成功率；
（4）如果攻击失败则停止；
（5）插入：计算伤害值；
（6）将伤害作用于怪物。

另一个方法是在用例模版中为扩展创建一个独立的部分[Cockburn01]。这里列出了每个

扩展并且使用了步骤编号来引用与之对应的主过程中的步骤。如果使用一个独立的部分来表示扩展，用例会变成下面这样。

基本过程：
（1）玩家角色瞄准怪物；
（2）玩家角色使用当前武器进行攻击；
（3）计算攻击的成功率；
（4）插入：计算伤害值；
（5）将伤害作用于怪物。

扩展：
(3a) **如果攻击失败则停止。**

无论采取什么方法，最重要的是始终如一地使用它。

6. 前提条件和后置条件

用例需要特定的前提条件才能成功运行。前提条件是由用例运行前的系统状态或用例本身的某些输入组成的。用例成功运行后就会产生后置条件。后置条件是对系统状态的改变或是用例的某些输出。这两个概念对于确保系统模型的完整性，以及捕捉到所有重要用例来说都非常重要。如果某个用例的一个前提条件既不能被其他用例的后置条件所满足，也不能被系统的输入满足，那么这里面一定遗漏了什么。同样，如果一个用例产生的后置条件既不是其他用例的前提条件需要的，也不是系统输出需要的，那么这个后置条件很可能是不必要的。

在用例模版中，具有编号的简单语句用来表示前提条件和后置条件。

前提条件：
（1）怪物是一个有效的攻击目标；
（2）玩家角色装备了一个武器。

后置条件：
怪物受到伤害。

就游戏开发而言，这些概念有助于避免项目早期游戏模式中的错误。在本节的例子中，如果因为某种原因，怪物无法成为一个有效的攻击目标，前提条件就得不到满足，因此不能开始执行“玩家角色攻击怪物”这个用例。同样，如果玩家角色没有装备武器，也不能执行这个用例。这些明显的信号可以让游戏开发人员在游戏代码中，通过检查这些前提条件来预防奇怪的行为或潜在的侵入。不仅如此，这些信号还可以告诉游戏开发人员应该去识别那些相关的用例，如描述被攻击目标怎样才能变得有效的用例和玩家角色应该怎样装备武器来进行攻击的用例。

“怪物受到伤害”这一后置条件也用到了。刷新显示可以用来表示怪物正在衰减的生命值，一个用例来也有可能用来描述当怪物生命值到达 0 时所发生的事情。关键在于，游戏开发人员是在用例的引导下才注意到这个问题的。

前提条件和后置条件是配对的。用一致的语言来描写可以帮助游戏开发人员识别相关的

配对——并通过它们来识别用例之间的依赖关系。要避免把两个或更多的前提条件或是后置条件组合在一行上，否则可能会推断它们之间存在着某些联系，而这很可能只是巧合。有时候，前提条件列表的一个子集就可以让用例成功运行，那就把可选的前提条件用“或”关联起来表示。本书不推荐使用可选的后置条件，因为它们的出现意味着存在某些被忽视的扩展。如果发生这种情况，游戏开发人员就需要检查用例并且试图对其进行分解，从而确定所有的后置条件，这也许意味着需要创建额外的用例来捕捉所发现的扩展。

7. 注释

模版的最后一项是为了捕捉那些过于冗长，或者不适合放在模版其他项目结构中的细节。这里游戏开人员要遵守两个原则。首先，我们必须列出每个注释并对其编号，然后在用例中使用一个数字上标来引用它们。其次不要把那些更适合在模版的主要项目中描写的内容作为注释列出。

1.5.6　漂亮的图表

组织、讨论和编写用例时，画一些图表来表示用例及它们之间的关系是非常有用的。这些图表，就像图 1-15 中的一样，被称为用例模型图[UML01]。它们有 3 个主要目的：

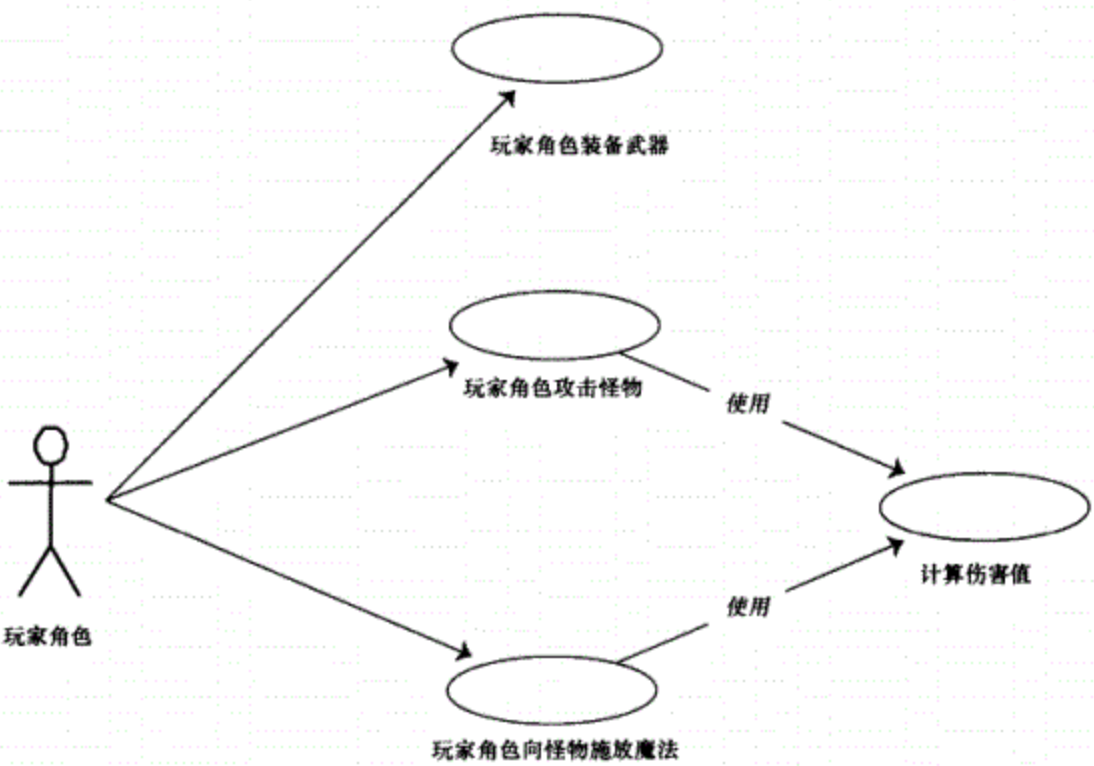


图 1-15　一个用例模型图

1. 为系统的作用域提供一个可视化的表示；
2. 识别用例之间的关系；

3. 有助于重构和组织相关的用例来进行子系统设计。

注意图 1-15 是怎样表示“玩家角色攻击怪物”和“玩家角色向怪物施放魔法”对“计算伤害值”的共享的。这对于在开发早期编写模块化设计来说非常有用。只要一个图表工具既能用线条连接图形，又支持文本注释，就可以用它来绘制用例图表了。这些工具不是必须的，但是通常很有用。

1.5.7 开始实现

一旦编写了用例，应该怎样用它们来指导游戏的实现呢？此时游戏开发人员所有的不过是关于他们在代码中想实现功能的精细而有组织的结构化描述。他们需要再一次地使用规则、判断和经验。

正规的方法学家通常建议不要从用例直接进行编码[Evans02]。很多有根据的理由支持这一观点。虽然用例的确使游戏开发人员更好地理解了游戏结构和特定行为，但它表示的仅仅是一个比较抽象的概念。简单地把这些用例直接转化为代码必定会使他们忽视某些关键的架构因素，还会掩盖某些重要的实现细节。

一个广为接受的做法是通过运用各种面向对象设计技术在用例的基础上建立一个更充实的设计模型——特别是序列图[McNeish02]。序列图描述的是为实现给定用例所必须的对象实例之间的方法调用序列。它提供了一个中间粒度的层次，使游戏开发人员有机会能够更详细地考察类接口和对象互动。随着在开发序列图的过程中对所需要的对象和方法进行详细说明，他们可以逐渐开发出一个可以在静态类图中建模的类结构。这个过程不仅让他们可以对很多架构性问题进行考虑，还让他们可以更有效地处理这些问题。

虽然上面所说的是首选的方法，但是很多游戏开发人员可能没有时间和资源也不愿意轻易采取这么正规的方法。并且他们对这些技术的讨论已经超出了本文的范围。相反，本文假设游戏开发人员在技术设计和实现上具备足够的经验，而且总是使用合理的工程和技术设计原则。接下来的讨论将集中在如何从用例中获益，即使并没有使用正规的方法。

1. 候选抽象概念

既然知道了游戏是怎样运行的，游戏开发人员就需要确定这些行为的宿主结构。这一点非常重要。因为一个定义良好的程序结构会更加稳健有效，并且会比没有良好定义的程序更容易维护。良好的定义意味着代码是有组织的：相关的功能和数据集合在一起，实现细节隐藏在定义功能的接口之后，程序元素之间的依赖是显式的而不是隐式的。

为了获得良好定义的结构，游戏开发人员需要找出那些负责执行用例中所描述的操作的抽象概念。用例的每个步骤都是一个具有主语和谓语的句子，通常还有一个宾语：某个事物执行某个行动，通常作用于另一个事物。做出这个行动的事物负责执行这一行动。这是一个符合程序结构的抽象概念，也有助于游戏开发人员定义程序结构。因此，发现抽象概念的第一步是识别用例步骤中的名词，无论它是这个步骤的主语还是宾语。最终，这些候选的抽象概念会被具体化为面向对象设计中的类、数据结构或是重要属性。

现在，回到玩家角色攻击怪物和计算伤害值这两个用例上。游戏开发人员可以从这些用例的步骤，以及它们的前提条件和后置条件中识别出一些候选抽象概念。把它们在表 1-4 中

列出并且尝试按照潜在的类、数据结构、属性和变量来对它们进行分类，并且为它们在游戏中的含义做出注释。随着这些工作的进行，游戏结构的本质开始显现出来。

表 1-4　候选抽象概念、类型和注释

抽象	类型	描述
玩家角色	类	表示游戏世界中的玩家，执行玩家发起的行动
怪物	类	表示游戏世界中 AI 控制的角色。和玩家角色的相似程度有多大
武器	类/数据结构	一个 PC（也可能是怪物吗？）用来挥舞作战的对象。包含对攻击结果产生影响的信息。问题：挥舞和装备是否类似
伤害值	变量	攻击过程中根据输入计算出来的瞬间值
攻击值	属性	攻击计算的输入。武器的属性。每个武器的攻击值不同。问题：是什么决定了武器的攻击值
攻击修正值	属性	攻击计算的输入。所装备物品（武器、带有魔法的盔甲？）的属性
特殊攻击修正值	属性	攻击计算的输入。根据攻击目标类型的不同而不同。玩家角色的属性。问题：是否每个目标类型都具有不同的SAM？如果某个目标类型没有相对应的SAM该怎么办？这应该是谁的属性
装备了的物品	类	PC（也可能是怪物吗？）可以装备（穿着）的对象。可能具有可以修改攻击计算的属性
目标	无	其他类充当的角色（Role）。怪物（可能是 PC 吗？）可以是目标。问题：还有什么可以作为目标？是什么使它成为目标
目标类型	属性	把这个目标和其他目标区别开来的分类
攻击者	无	其他类充当的角色（Role）。PC（可能是怪物吗？）可以是攻击者。问题：进入攻击状态是否会导致其他事情的发生（除了攻击以外），譬如说向敌人 AI 报警、影响 PC 的声望等
防御值	属性	攻击计算的输入。目标怪物（或 PC？）的属性。每个怪物或 PC 可以具有不同的防御值。问题：特定怪物的防御值由什么决定
防御修正值	属性	攻击计算的输入。所装备物品的属性
特殊防御修正值	属性	攻击计算的输入。根据攻击者类型的不同而不同。怪物的属性。问题：每个攻击者类型是否都有不同的 SDM？如果某个攻击者类型没有相对应的 SDM 该怎么办

2. 对抽象概念的分析

上述研究揭示了一些与游戏相关的有趣事实，同时提出了不少需要进一步研究的重要问题。虽然在用例中没有提到怪物能否攻击玩家，但可以合理地作出肯定的结论以使游戏更加有趣。也就是说，怪物做出的这类行为可能和玩家发出攻击的行为非常相似。那么玩家和怪物之间有多少共同点呢？这个问题需要进一步探讨。回顾玩家角色攻击怪物这一用例，并且

颠倒这两个角色来看看会发生什么情况。也许可以从玩家角色和怪物这两个抽象概念中特化出一个尚未定义的更高层次的抽象概念。

武器抽象概念包含了一些有趣的属性，它们对计算攻击结果来说至关重要。任何特定于武器的行为都还尚未被发现，因此这可能只是一个数据结构。在此还有一个问题：挥动一件武器是否和装备一件物品类似？无论如何，玩家或怪物与一个或多个物品之间的联系显现出来。还有额外的问题。

（1）武器以及其他装备了的物品和游戏中其他类型的物品有何不同？

（2）是否不能装备某些其他物品？

（3）是否每个装备了的物品都可以作为武器使用？

有些答案是显而易见的，还有一些则可能需要额外的用例来发现。

目标和攻击者抽象概念对于玩家和怪物来说是没有区别的。更确切地说，它描写的是其他抽象所充当的角色。为了让模型更加简明，排除那些不会成为类、数据结构或是属性的候选抽象。把抽象识别为角色提示了，攻击的参与者在与攻击有关的状态上可能会发生变化。这同时也指出了攻击者和目标之间的关系。因此游戏设计人员必须确定成为一个攻击者或是目标是否会在攻击的上下文之外对一个 PC 或怪物产生影响，以及这种关系是显式而长久的还是隐式而短暂的。

3. 开发一个类结构

对候选抽象的研究可以帮助游戏开发人员建立了游戏的结构视图。其中可能会有一个包含玩家和怪物的类层次以及另一个包含武器和其他物品的类层次。此外，玩家角色、怪物和武器及其他装备了的物品之间存在关联。不仅如此，玩家和怪物之间也存在着关联，它们一方是攻击者，另一方是目标。图 1-16 用一个简单的类图来说明这些关联。简单地阅读用例中的步骤可以说明这些抽象执行了哪些行动。这最终揭示了哪些类需要实现的方法。

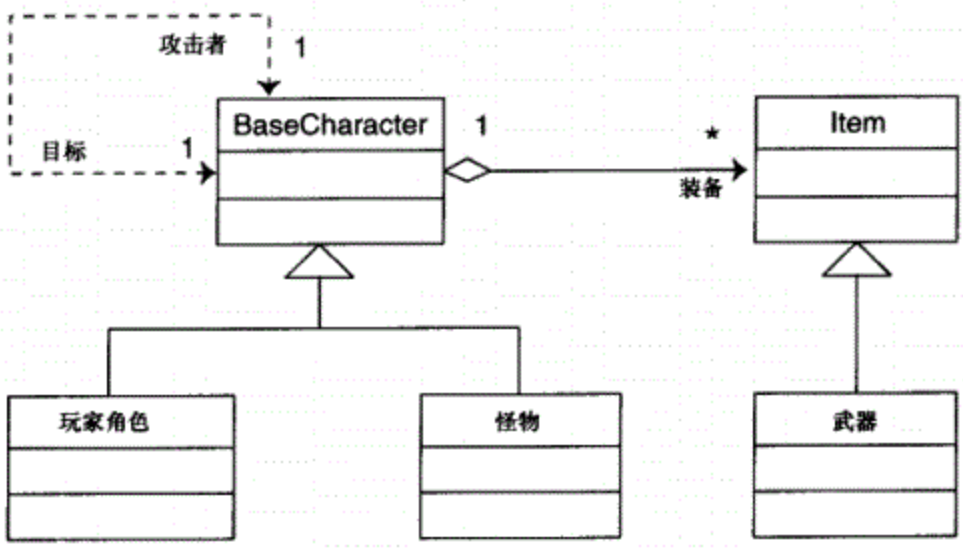


图 1-16 最初的用例类图

如果游戏开发人员打算用 C++类来编程实现目前所知的功能，代码可能会和下面的差

不多。

```
class BaseCharacter
{
public:
     // ...
       bool SetTarget(BaseCharacter* pTarget);
       BaseCharacter* GetTarget();
       bool AttackCurrentTarget();
       void ApplyDamage(int iDamageAmt);
       int GetDefenseValue();
       const string& GetCharacterType();
       bool EquipItem(Item* pItem);

protected:
       bool AttackIsSuccessful();
int ComputeDamage();
Item* GetEquippedItem(const string& name);

   // dictionary of equipped items, keyed by name.
   hash_map<string, Item*> m_EquippedItems;
   int m_MaxHitPoints;    // our amount of "life"
   int m_CurrentHitPoints;
   int m_DefenseValue;
   string m_CharacterType;
};

class PlayerCharacter : public BaseCharacter
{
public:
   int GetSpclAttackMod(Monster* pTarget);
   // other player specific stuff to be determined
private:
   // mapping of SAM by target type
   hash_map<string, int> m_SpclAttackMods;
};

class Monster : public BaseCharacter
{
public:
   int GetSpclDefenseMod(PlayerCharacter* pTarget);
   // other monster specific stuff to be determined
private:
   // mapping of SDM by target type
   hash_map<string, int> m_SpclDefenseMods;
};

class Item
{
public:
   bool CanBeEquipped();
```

```
    bool Equip(BaseCharacter* pOwner);
    int GetAttackMod();
    int GetDefenseMod();
protected:
    int m_AttackModifier;
    int m_DefenseModifier;
    BaseCharacter* m_pOwner;
};

class Weapon : public Item
{
public:
    GetAttackValue();

Protected:
    int m_AttackValue;
};
```

这些类只不过是游戏实现的框架，而仅从两个用例就可以派生出这些类。随着游戏程序员编写更多的用例来描述其他重要的游戏行为，游戏的行为和结构建模将更为充实。的确，很多结构元素是由架构目标驱动的，譬如说网络、数据持久化、3D 图形渲染以及第三方库的使用。不仅如此，任何受架构因素影响的结构必须支持开发人员在用例中阐述的游戏行为，并将其进一步扩展。

1.5.8 用例的指导方针

初次接触这项技术的设计人员通常能够很快掌握编写用例的方法。然而，和其他技能一样，良好的判断和有效的应用需要经验的积累。文章接下来将介绍一些松散的指导方针，来帮助游戏设计人员避免一些在以 MMP 开发为背景的用例中经常会遇到的错误。

1. 描写问题，而不是解决方案

用例是对创建软件系统所要实现的功能进行分析的技术。用例描述了有待解决的问题，而不是怎样解决问题。用例不能用于实现细节，譬如说对数据结构的描写、用户界面元素、网络协议、数据库查询甚至算法。

这样做的理由很简单：在编写用例时，游戏设计人员还处于对系统的了解过程中，没有足够的信息去判断某项技术决定是否正确，而这些决定很可能会在今后的研究中作废。这样的决定很可能是错误的，需要对它进行重新研究。更糟的是，游戏开发人员有可能为了迎合某些技术假设而在编写用例时有所偏颇，这使我们错误地表示了游戏设计所要表达的需求，最终导致游戏偏离设计人员的目标或期望。

2. 迭代

用例是一种探索工具，它可以帮助游戏设计人员了解所要创建的系统的细节。同样地，随后的发现常常会使游戏设计人员对那些自认为已经理解了的早期观点进行推敲。游戏开发人员应该准备好在开发用例时，根据需要对其进行多次调整。他们可以在发现所有用例之前

就开始实现关键用例的代码。因为无论他们在捕捉游戏需求时多么尽力，一旦开始实现游戏，很多被忽视的细节还是会被发现。

一个良好的迭代方法是从一个与其他未实现的子系统依赖很少的子系统开始编写用例。然后，在合理的前提下使用在这些用例中所获得的信息尽可能地实现这个子系统。接着，编写另一个相关子系统的用例，它必须和前一个子系统有某种形式的交互。因为对第一个子系统中用例的实际实现已经有所了解，在开发第二个子系统时我们也许可以用到这些信息。当在第二个子系统中有进一步的发现时，很可能需要对第一组用例做些改善。在第二组用例的基础上编写代码时，也有可能需要游戏开发人员对第一组用例的某些代码进行重构。这个循环会一直持续到系统发展成为一个完整的游戏。

3. 面对面的合作

用例是一种交流工具。它的主要目的是把关于系统需求的想法——游戏的设计——传递给代码的实现者。这并不是一条单行道。那些尝试独自编写一堆用例然后把它们交给程序员的游戏设计人员，或是那些希望从游戏设计文档中搜集用例内容的程序员都是在自找麻烦。

设计人员使用游戏设计文档书面表达他们的创造力。然而，无数细节和假设并没有记录在案，它们只存在于游戏设计人员的思维过程中。程序员和设计人员之间的合作至关重要，它使程序员可以洞悉设计人员的思路。同样重要的是，它提供了一个程序员向设计人员提出反馈的途径，这有助于确保设计被及时理解并且在逻辑上保持一致。这是任何一方都无法独立做到的。

4. 不要指望用例可以捕获所有需求

对于识别系统需求来说，用例是个很有价值的工具，但它既不是惟一的工具，也不可能捕获所有的需求[Evans02]。想当然地认为它能够做到这点，会给之后的开发过程带来难以解决的意外问题。从根本上说，用例可以对和游戏功能相关的需求进行建模，这包含了游戏模式的规则、游戏世界中对象的交互以及像聊天、背囊管理、探测地图、团队管理这样的游戏支持系统的行为。

用例不能对安全性、性能、网络延迟、侵入预防、过程控制等对 MMP 的成功非常重要的架构因素进行建模。用例也不能对运行期支持需求进行建模，譬如说行动报告或是客户支持调停等。然而，在这些架构性和支持性框架中对用例进行建模，可以确保功能需求不会违反架构目标。譬如说，游戏的设计目标之一可能是允许玩家上传他们自己的内容，但是客户服务的目标之一可能是最小化对玩家行为进行管辖的必要。在这种情况下，游戏开发人员可以使用用例来定义这样一个系统：玩家需要作出某些承诺才能看到其他玩家上传的内容。

5. 交流

在向团队领导、制作人、计划经理和其他团队的成员描述正在开发的游戏时，用例的作用是巨大的。简单地列举一组相关用例的标题，就可以传递出惊人的游戏信息。仅仅通过列举用例标题，过去不了解情况的团队成员就可以得知游戏将要实现的特性。譬如说，下面就是一些列出的用例标题：

- 玩家角色装备武器；

- 玩家角色瞄准怪物；
- 玩家角色攻击目标；
- 计算伤害值；
- 目标死亡。

相对于“战斗系统”这样含糊的标题来说，用例标题可以告诉临时成员更多关于游戏的信息。相对于用例标题来说，用例描述具备更多的细节，它可以作为对游戏设计的一个彻底的重新迭代，对那些设计人员和程序员达成一致的部分进行描述。这可以让管理层清楚地知道这个游戏所要实现的真实功能。

6. 使用用例进行计划

用例有助于项目计划。因为每个用例都代表了开发工作中一个相当小的可管理单元，游戏开发人员可以对实现它们所需的时间进行估计。用例揭示了游戏实现中不同部分间的依赖关系，因此有助于对任务列表进行排序以使效率最大化，同时还可以避免为了等待解决没有预料到的问题而导致停工。然而，必须谨记，由于用例并没有对所有需求进行建模，所以并不能识别实现游戏所需要的所有任务。要避免简单地认为“因为所有用例都得到了解决，所以系统被完整定义了”。

7. 避免线性思考

用例是用来发现的工具。游戏开发人员不能寄希望于以一种确定性的方式来定义游戏行为。有序的步骤可以用来描述用例，但这并不意味着必须按照其执行的顺序去认识到每一步。作为迭代开发方法的一部分，游戏开发人员必须考虑在一个已经“完成”的用例中插入额外的步骤或是删除某些步骤。

游戏开发人员可能会认为某一步骤可以扩大为一个完整的用例，但是如果在当时还没有足够的信息去充实它，那就不要勉强。也许对这个重要步骤进一步研究后会发现使用一行文字就可以更好地描述这一步了。急于为新的发现创建并不需要的用例会增加管理用例的负担。

注意，随着对用例的步骤按照更能体现需求的方式进行重构，游戏开发人员可能需要修正最初的描述。不要仅仅因为旧的用例描述了最初的意图就拒绝对它进行修改。刻板地坚持第一印象会让我们遗漏重要的游戏需求或是识别出无效的需求。

8. 不要过于强调工具

目前有很多软件设计工具都声称支持用例开发。其中的一些提供了很有价值的特性，但是通常都需要一大笔开销，并且，使用这些工具的项目要采取某些形式的严格过程，才能从工具的特性集中获得最大好处。超大型项目可能会受益于这些工具，但是本文认为大多数游戏开发团队将把它们视为负担而不是从中获益。因此，开发人员应该按照本文所描写的模版或是某些更适合于项目的变种，使用一个简单的字处理程序来捕捉用例。如果希望使用图表，任何可以把图形用线条连接起来，并且对它们进行文本注释的绘图工具就已经足够了。其他的功能反而可能让开发人员把注意力从游戏开发的主要目标上移开，或是带来不必要的负担直到他们完全放弃用例。

9. 把重点放在清晰的交流上而不是格式上

不要为遵守本文或其他文章所提出的用例编写“规则”而感到不安。只有认识到这些方法无非是对游戏行为进行描写的一般化手段，才能有效地使用它们。对游戏需求的清晰交流比遵守任何约定都重要。如果把每个步骤单独地写一行妨碍了游戏开发人员思维过程和合作流程，那就考虑一个更好的叙述方式。如果在步骤中写入内嵌的条件逻辑伪码让叙述方式更加清晰，那就这样去做。本文所推荐的方法是为了帮助交流，如果这个方法阻碍了交流，那就应该对它进行改善或是尝试其他方法。

10. 避免把注意力集中在客户服务器的细节上

在为 MMP 游戏编写用例时，游戏开发人员经常会遇到哪个操作应该在客户端执行、哪个操作应该在服务端执行的问题。本文认为这是一个和用例无关的实现细节，通常不应该在用例中表达。

在一个拥有无限带宽和零延迟的完美世界，一个 MMP 游戏程序可以完全在服务端运行。每个客户端等价于一个图形化的“哑终端”（dumb terminal），只用来显示服务端所发生的事情。因为玩家和游戏间的所有交互都存在于单一的逻辑空间中，而这必然是在服务端。

当然，无限带宽和零延迟并不存在，因此设计必须加入游戏客户端功能来对这些情况进行模拟。游戏设计人员需要一个可以作出及时响应的游戏，因此他们对某些行动——譬如说游戏世界中的移动，执行与动画有关的行动等——在客户端进行预测，并且让服务端对客户端的错误猜测进行修正。这个功能很明显是一个实现细节。试图在用例中插入这样的细节会轻易地把描述从“发生了什么”转移为“怎么发生的”。

11. 把游戏和用户界面分离开

很多用例编写者常犯的错误是，在一个想要描述游戏行为的用例中描述了用户界面交互。为游戏行为建模时，用例编写者不应该去关心按下哪个按键或是选择什么菜单会导致行为的发生。从游戏的观点来看，这些都是实现细节。玩家在装备武器时，是把武器放入一个纸人的口袋里还是把它拖曳到角色的 3D 模型上并不重要。无论在什么情况下，装备武器所使用的游戏代码都是相同的。

这并不是说不应该使用用例对用户界面行为进行建模，事实上，在这方面用例同样有效。编写这类用例时，工作人员应该把注意力集中在用户界面元素本身，而不是其背后的游戏内容。这类用例将对玩家按下某个按键、选择某个菜单项或显示一个窗口后所发生的行为进行描述。在这里，用例编写者不应该去描写游戏怎样对这些输入作出反应，而应该对如何更新显示、可能发生的任意输入状态的改变以及所期望的用户输入类型进行描写。

把游戏用例和用户界面用例分离开可以让游戏开发人员对这两个子系统进行更好的建模，这样对一个子系统的建模就不会受到另一个子系统所作的隐含假设的影响。

12. 不要强求完全覆盖

出于很多原因，用例可能无法应用于整个项目。有些技术领域（特别是数据库、网络、进程管理这样的底层架构领域）可能早已被实现者良好地理解了。有些程序员或游戏设计人

员可能会对采用那些还没有证明对他们直接有效的新技术进行抵制。某些管理层成员可能不理解用例能够带来的好处，并且觉得在进度表上为编写用例留出专门的时间不太合适。

不要认为如果不能使用用例来表达游戏的所有方面，那使用用例就毫无意义。通常，游戏可以通过在某些领域对用例进行战略性的应用来获得好处，即使也许只有一个设计者或程序员在使用。既然开发人员可以亲自证实它们的效果，又何必费神地试图说服抵制它们的人呢？如果希望获得团队中其他成员的支持，游戏开发人员应该以易读的方式编写用例并与团队分享。当对所负责的游戏部分进行讨论时，游戏开发人员应该以用例的方式进行讨论，并且在其他团队成员有问题时让他们参考用例。

如果用例确实对项目的其他部分有价值，团队中的其他成员就会逐渐对用例产生兴趣。即使他们不感兴趣，至少游戏的开发工作可以从中获益。

1.5.9 总结

用例是一种把游戏的创造性设计转换成有助于驱动技术实现的结构化方式的有效方法。在软件产业中，对于编写用例的有效方法有着不同的意见。本文的定位是使用任何适合当前项目的方法，同时要考虑到所开发的游戏的特性、开发团队的动态性以及游戏开发人员愿意在多大程度上接受规则和正式的方法。本文所介绍的方法不仅很实用，理解和执行起来也非常方便。无论采用什么方法，按照用例的思想进行思考可以帮助游戏开发人员更好地理解并实现所要解决的问题。

1.5.10 参考文献

[Cockburn01] Cockburn，Alistair，*Writing Effective Use Cases*，Addison-Wesley，2001.

[Evans02] Evans，Gary K.，“Why Are Use Cases So Painful?”，*http://www.evanetics.com/articles /Modeling/UCPainful.htm*，2002.

[Fowler99] Fowler，Martin，*UML Distilled Second Edition: A Brief Guide To The Standard Object Modeling Language*，Addison-Wesley，1999.

[Jacobson92] Jacobson，Ivar，*Object-Oriented Software Engineering: A Use Case Driven Approach*，Addison-Wesley，1992.

[McNeish02] McNeish，Kevin，“UML Sequence Diagrams，” *CoDe Magazine*，March/April 2002，*http://www.visualuml.com/whitepapers/UML%20Diagrams.pdf.*

[UML01] Unified Modeling Language，v1.4，*http://www.omg.org/cgi-bin/doc?formal/01-09-67*，Object Management Group，2001.

1.6　使用生命值来设计转换因子

John M. Olsen，Microsoft Corporation
infix@xmission.com

本文介绍了怎样根据物品造成伤害或是治疗伤害的能力为其赋予适当的价值，并以此为基础展开讨论，从而让读者知道应该怎样为游戏经济体系中的大多数物品确定价值。

在生活中，几乎每件东西都有相应的价格。只要把这个简单的真理延伸到游戏中对生命值的应用中去，就可以极大地简化那些通常难以平衡的机制。在 MMP 游戏中，游戏开发人员必须知道金钱是怎样进出游戏世界的并对其加以控制，否则游戏中的经济体系会瞬间崩溃。

这篇文章假设某个游戏开发团队在设计一个虚幻游戏，但这里的概念同样适用于任何使用标准方式的游戏：在这些游戏中，玩家和非玩家角色（Non-Player computer-controlled Characters，NPC）的等级越高，他们的生命值和防御力也越高。本文讨论的所有物品都基于一个衡量货币的通用单位：硬币。本文的目标是寻找一个可编程的方式来确定在游戏设置中每个物品的价格，并应用于大量物品上。一旦计算出游戏中物品的相对价值，就可以任意决定一枚“硬币”所表示的实际单位。

为了使用这一系统，游戏设计人员必须作一些假设以确保这个转换关系在角色升级后仍然保持平衡。第一个假设是随着玩家在游戏中不断地升级，某个特定武器的攻击力不会有显著提升。降低初学者使用某种高级武器的能力是可以接受的，但是任何武器在被符合要求的玩家使用时，基本攻击力应该是一样的。

1.6.1　武器的价值

通常，在一个游戏中有很多方式可以造成伤害。有些武器只能使用一次，譬如说箭或魔法；有些物品可以有效地进行一定次数的攻击，随后就会消失或是需要重新充值；还有一些物品可以持续使用很长时间，它们会逐渐损坏。假设随着时间的推移，所有武器都会损坏。做此假设是因为游戏设计人员希望根据物品损坏前所能造成的所有伤害来决定其价格，而不只是把价格建立在其造成伤害的速度上。

现在，从一个简单的假设开始，假设一枚硬币等价于一点生命值。这个假设本身并不重要，在完成定价后，设计人员还可以根据需要对基础值进行调整。

如果某个物品可以使用一定的次数，那就简单地按比例增加其价格。假设一根魔杖可以使用 5 次，而不是像魔法卷轴那样只能使用一次，那么它的价格就是总的攻击力，也就是每次的攻击力乘以使用次数。根据表 1-5 中的数值，总的价格是 750*5，也就是 3750 枚硬币。

为那些可以使用很长时间的物品（譬如说剑）定价则更为复杂。假设玩家购买了一把崭新的剑并持续使用，直到完全损坏。这时，游戏设计人员可以让这把剑的攻击力降至最低，也可以让它折断从而不能继续使用，这完全取决于。他们需要的数字是这把剑在它的有效期内究竟可以造成多少伤害。

是时候重申假设了。首先，这把剑在崭新的情况下可以造成 1 到 15 点的伤害，在完全损坏后仅可以造成 1 点的伤害。这样就可以得出它在有效期内的平均攻击力范围是从 8 到 1。

表 1-5 造成一次性伤害或是固定次数伤害的物品

描述	使用次数	总攻击力	硬币数
劣箭	1	1-5	3
标准箭	1	2-10	6
改良的箭	1	6-12	9
超强箭	1	50-100	75
地域烈焰卷轴	1	750	750
地域烈焰魔杖	5	3750	3750

假设这个武器的有效期是 18 000 秒（5 个小时）的战斗时间。这意味着玩家在武器需要修复前可以使用很长的时间。假设玩家把大约 20%的时间花在战斗上，那么 5 小时的战斗时间相当于 25 小时的游戏时间。这段时间足够长，因此玩家不必频繁地往返于武器维修店。

如果这个武器每次攻击需要 5 秒的时间，那么它可以有效地进行 3600 次攻击。如果这个武器的攻击力随着时间线性地降低，那么可以简单地得出等式 1-2。

价值 = 攻击次数 *（损坏后的平均攻击力 +（全新时的平均攻击力 — 损坏后的平均攻击力 /2） （等式 1-2）

计算出的结果就是图 1-17 中斜线以下的阴影区域。图中纵轴表示攻击力，横轴表示挥动次数。

在等式 1-2 中填入具体数字就得到等式 1-3。由此可以得出这把剑的价值：

$3{,}600\times(1+(8-1)/2)=16{,}200$ 个硬币 （等式 1-3）

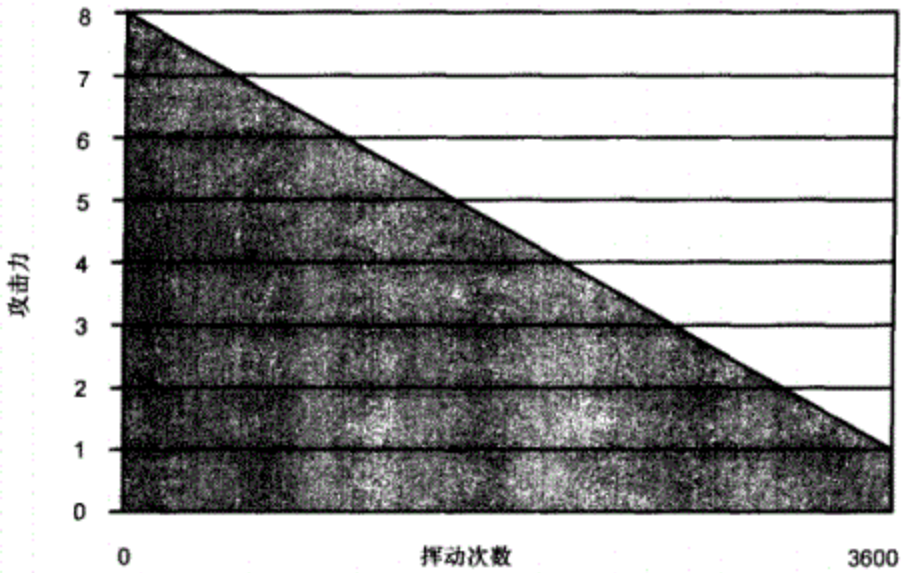


图 1-17　表示期望伤害的区域

再举一个例子，假设一种简单武器，譬如说青铜匕首。它不仅攻击力较低，有效期也很短，因此可以预计它的价格也会便宜很多。如果它在有效期中可以进行 1000 次攻击，攻击力在全新的时候是 1 到 4，损坏后是 1，这把匕首的价格如等式 1-4 所示。2.5 表示全新时候的平均攻击力，不要和最大攻击力混为一谈。

$$1000\times(1+(2.5-1)/2)=1750 \text{ 个硬币} \qquad \text{（等式 1-4）}$$

参照前面表 1-5 中箭的价格，可以知道一把匕首可以交换超过 290 支标准箭，或是大约 23 支超强箭。

这个等式还给出了这些物品损坏后对其进行修复的价格。因为修复价格直接与攻击力相对应，所以游戏设计人员就可以直接在武器损坏程度的基础上确定修复价格。如果武器损坏了 50%，那么修复它的价格就是新武器价格的 50%。如果游戏设计人员希望鼓励玩家尽早修复他们的武器，就应该使用非线性的比例，使得修复费用一开始增长较慢，在物品有效期的最后阶段加速。等式 1-5 中的函数使用了这种方法来计算物品修复价格：

$$\text{修复成本} = \text{基本成本}\times\text{损坏百分比的平方} \qquad \text{（等式 1-5）}$$

再一次用损坏了 50%的武器作为例子，使用这个非线性比例计算出来的费用将是购买一个新武器所需费用的 25%。等式中的平方项可以被任意定义域和值域都是 0%到 100%的单调递增函数取代。图 1-18 对两种可行的函数进行了对比。

高质量的武器以及那些有效期很长的武器价格应该更高，上述公式与这一想法非常一致。正如游戏设计人员所预期的，如果某种武器可以造成更大的伤害，其价格也会成比例增加。这样玩家就能够决定他们想要什么武器以及他们买得起什么武器。如果他们想要一个高攻击力的武器，又不愿意多花钱，就必须以牺牲持久度为代价，去使用那些容易损坏因而需要频繁修

复的武器。

游戏的设计还需要让玩家觉得这一切都很有趣。如果修复过于频繁，玩家就会觉得很无趣；因此必须对其进行大量的游戏测试来找到合适的平衡，从而正确地实现物品的使用和修复。

表 1-6 中所列出的武器都是使用上述方法计算的。注意最后两项，武器可以使用的次数和价格均匀地成比例增长。

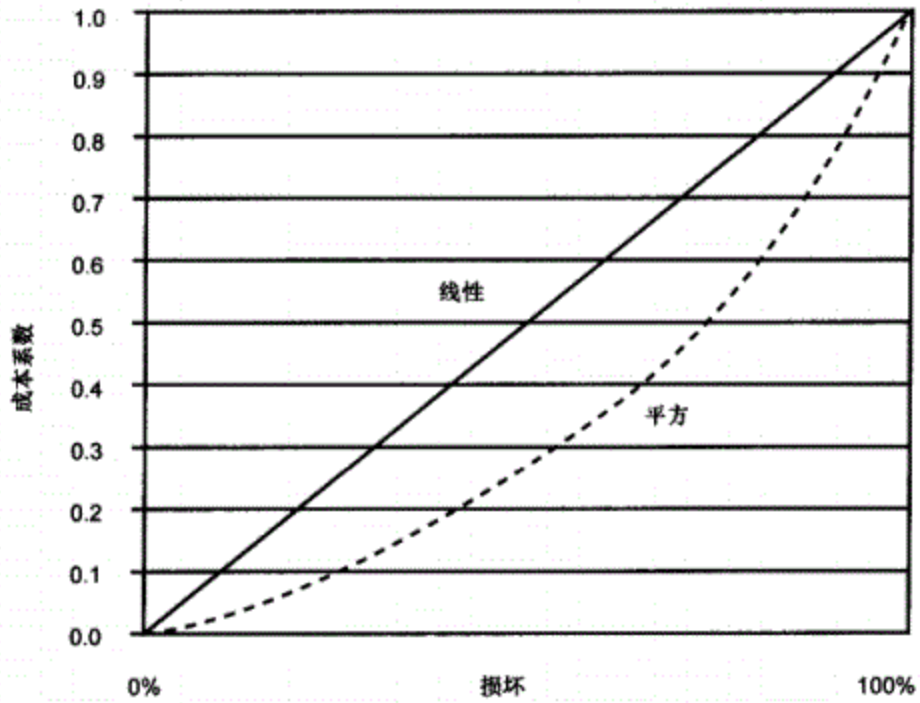


图 1-18 使用线形函数和平方函数的修复乘数

表 1-6 随着时间损坏的物品

描述	使用次数	全新时的攻击力	损坏时的攻击力	硬币数
剑	3600	1-15	1	16200
青铜匕首	1000	1-4	1	1750
棍子	200	1-10	2	750
精制小剑	5000	1-5	1-2	11250
极好的双刃剑	7000	8-30	2-4	77000
青铜双刃剑	1000	8-30	2-4	11000

1.6.2 治疗、防具和减轻伤害

上述数值方法同样可以用于修复伤害的物品上，譬如说治疗药水或医药箱。治疗效果的大小决定了物品的价格。如果治疗效果的大小是可变的，那么就需要基于平均值来计算价格，

就像上面所讨论的具有可变攻击力的武器一样。

通过跟踪一些额外的信息，游戏的设计也可以在防具上使用这些方法。游戏设计人员需要知道的就是防具在需要修复前所能为玩家吸收或是转移的攻击力。直接把生命值的大小映射到硬币数就可以确定防具的价格。游戏的设计可以简单地把攻击力平均分配到每一件防具上去，也可以把受到的伤害计算在作为攻击目标的那件防具上。

防具修复也可以使用和武器修复相同的方式。修复可以恢复防具吸收伤害的能力，正如对武器的修复可以恢复它造成伤害的能力一样。

1.6.3　从 NPC 获得的战利品

既然怎样为物品定价的问题已经解决了，那么这些物品从哪里来呢？当然是作为战利品获得。那怎样决定玩家击败某个特定的 NPC 后应该获得多少战利品呢？

这里的基本规则是，战利品的平均价值应该略大于杀死这个 NPC 所需造成伤害的数量，加上这个怪物会对玩家造成的伤害点数，再加上玩家在防具和武器上的消耗。如果使用图 1-18 所示的非线性修复系数，任何战利品比例乘数都可以减少到接近于 1。

战利品的基本数量取决于玩家杀死这个怪物的速度。既然考虑了概率的作用，那就可以作一个一般化的假设：玩家造成的伤害约等于怪物造成的伤害。防具和魔法保护也会对数据产生影响，但是它们的效果有可能会被其他因素所抵消。基本的计算如等式 1-6 所示。为了让这些等式的结果对玩家有利，游戏设计不对战利品的参考价值进行调整。

战利品参考价值 =（玩家武器的损坏 + 玩家防具的损坏）　　（等式 1-6）

假设从统计角度来看玩家武器的损坏约等于玩家防具的损坏，等式 1-7 就可以用来对其进行简化。

战利品价值 =2×玩家武器的损坏　　（等式 1-7）

总损坏量决定了战利品参考价值，也就是每个怪物平均会带多少战利品。从另一个角度来看，如果知道了它可以给出什么样的战利品以及它所使用的防具，游戏设计人员就可以确定其生命值是多少。等式 1-8 可以用来求解生命值。

玩家武器的损坏 = 战利品价值 / 2　　（等式 1-8）

玩家武器的损坏源于对 NPC 造成的伤害以及对 NPC 防具造成的损坏。因此，武器的损坏应该是这两个值的和，如等式 1-9 所示。

玩家武器的损坏 =NPC 受到的伤害 +NPC 防具的损坏　　（等式 1-9）

再一次对战利品参考价值进行求解，得到等式 1-10。

战利品价值 =2×（NPC 受到的伤害 +NPC 防具的损坏） （等式 1-10）

譬如说，假设没有防具的兽人被击败后，有 50%的可能性会掉出一根棍子，其战利品参考价值是 750 个硬币，而还有 50%的可能性不会掉出任何战利品。如果这根棍子每次都已经损坏了 50%（使用简单的线性比例），等式 1-11 中的战利品价值就是新棍子价格的 4 分之 1。在对战利品参考价值进行调整时，必须同时考虑概率以及损坏量，因为对平均情况也要进行处理。

战利品价值 =750×0.5×0.5=187.5 （等式 1-11）

假设兽人没有防具，那么所有的伤害直接作用于生命值之上。在等式 1-8 中填入这些数字得到了等式 1-12，这个结果说明了这场战斗会给玩家的武器带来多大的损坏。因为这些损坏都是为了对 NPC 造成伤害，它可以用来计算 NPC 的最大生命值。

玩家武器的损坏 =187.5 / 2=93.75 （等式 1-12）

以这个为基础，增加战利品或是减少怪物的生命值可以改变玩家在游戏中致富的速度。最初时可以把 1 作为基本比例因子，然后根据游戏测试的需要进行调整。

如果一个兽人所穿的防具可以吸收 20%的伤害（15 点生命值），那么其战利品的参考价值就应该增加 30 个硬币，这是因为玩家在攻击它时不仅需要对它造成额外的伤害，并且由于它的生命期延长了，它在死前也会对玩家造成更多的伤害。

有时游戏设计人员可能会希望需要付出很大的代价才能杀死某些特定的 NPC，它们的战利品参考价值将远远低于根据它们的生命值和防具等级所计算出的值，当然，这个特定的怪物会偶尔掉下一个罕见的物品来补偿玩家。有时游戏设计人员会希望一个 NPC 可以掉出比其他 NPC 多得多的战利品从而使玩家群起而攻之，这样做可以鼓励玩家去一些比较偏僻的地方。

整体理解平均情况和统计理论在这里很有用。如果要让一个容易杀死的怪物掉下很昂贵的战利品，就必须调节掉下战利品的概率使得平均数接近这个 NPC 的战利品参考价值。譬如说，如果游戏设计为一只等级很低的老鼠会掉下一颗昂贵的宝石，就必须让这种情况非常罕见。

1.6.4 制造业

如果游戏中有贸易功能并且允许玩家制造武器，就应该鼓励玩家使用制造技能。要做到这点，只需要让这些从事手工业的玩家自行制造物品比直接从 NPC 商人那里购买合算。如果玩家可以通过自行制造而获得折扣，一定会有玩家愿意利用贸易技能，提供低于 NPC 商人定价的物品和服务。

制造一件武器时，部件的价格之和绝不能超过武器的价值，否则玩家就不会去制造它们。但是也不要让部件的价格远远低于武器的价值，否则就好像在游戏中制造了一部印钞机，玩

家会不停地制造武器并且把它卖给 NPC 商人获利直到游戏经济彻底崩溃。

必须确定玩家通过制造物品并且用低于 NPC 商人的价格出售可以获得的最大利润。一旦知道这个值，就可以确定部件的组合价值。游戏的设计可能需要进行一些尝试甚至犯一些错误才能获得用来制造物品的所有部件的合理成本。因此，有可能要对有关的配方进行仔细地调整。如果发现部件成本太低，可以通过增加制造过程中的人力成本来进行调整。

1.6.5　无关物品

游戏中有些物品和战斗、防具以及治疗伤害完全没有关系。在这种情况下，只能靠游戏设计人员自己来定价了。既然已经有了一个装备和部件列表以及它们的价值，游戏设计人员可以借助这些已知物品来得到与其他数据一致的价格。

譬如说，从森林中获得的木棍的价值是确定的，因为它可以用来制造箭；布的价值也可以得到确定，因为它也出现在一些防具配方中。如果需要确定一个帆布床的价格，就可以使用特定数量的木棍的成本，再加上所需要的布的成本以及一小部分人力成本来得到帆布床的价值。

很多关于食物的信息很可能无法从制造武器的部件上获得，因此需要根据已知的价格为其估计一个比较合理的价格。一旦对某个新物品类型中的某些物品作出初步定价，游戏设计人员就可以通过配方把这些物品关联起来以保持它们彼此之间的一致性。

1.6.6　检验

一旦建立了一个系统并且开始运行，游戏开发人员就应该对所有的贸易进行记录，并且把玩家之间的交易价格和预设的价格进行比较。如果某个物品的价格和玩家之间的实际交易价格不同，或是该物品的交易量突然上下起伏，游戏开发人员就可以明确地知道在在线测试中为它制定的价格是错误的。

通常玩家的定价和游戏预设的价格差别不会很大，因此当区别显著时，就需要去调查原因了。这很可能只是因为一个定价错误，但是也可能是因为玩家对游戏机制的滥用。进行价格检查有助于捕捉到这些问题。

1.6.7　总结

最后，切忌一味地遵循这些转换因子和定价方式。它们只是用来帮助对 MMP 游戏中一些容易出问题的部分进行平衡的方法。有时游戏开发人员希望鼓励玩家精通修复技能，有时他们则希望通过调整修复价格和部件成本来鼓励制造。

也有可能游戏中根本就不需要贸易技能，那就可以忽略所有关于制造成本的问题。一旦掌握了基本思想，就可以对物品的价格进行各种各样的变化。它可以作为游戏中的定价系统的基础。

1.7 在 MMP 游戏设计中加入故事情节

Paul McInnes，Micro Forte
paulmc@syd.microforte.com.au

近年来，一个崭新的商业化计算机游戏类型开始兴起：大型多人（Massively Multiplayer，MMP）游戏的诞生。由于可以从订阅者那里获得持续的收入，并且目前大多数商业化 MMP 游戏都获得了成功，这使大家对这个市场细分都非常感兴趣。对于设计人员来说，一个最紧迫的现实挑战就是怎样才能在这个竞争日益激烈的市场中获得足以维持商业化运营的订阅者群体并把他们留在游戏中。

怎样才能吸引玩家并且留住他们呢？这个问题很复杂，一方面，游戏要具有容易理解的设计，这对于那些投入的在线游戏玩家来说具有长期的吸引力。这些设计不仅具有复杂的抽象游戏系统，而且包含了角色提升机制（character advancement）来引导游戏的进行并成为玩家的最终目的。同时，在角色升级系统（character progression system）中，玩家需要成千上万个小时才能达到最高级别。对于骨灰级玩家来说，一个需要大量时间和毅力才能完成的游戏是极具吸引力的。

与此同时，能够在这个市场上存活下来的骨灰级游戏并不多。随着受众的增长，越来越多对在线游戏好奇的休闲玩家也加入其中，但他们并不愿意也不可能在游戏中花费那么长的时间。事实上，那些让现有游戏得以长期存在的因素也正是休闲玩家进入游戏的主要壁垒。游戏的设计应该怎样吸引更广泛的受众并且让玩家在最初的新鲜感消退后仍然对游戏感兴趣呢？

这涉及到很多复杂的问题，与游戏体验的本质、故事情节在计算机游戏中的作用以及 MMP 媒体自身的基本性质有关。要在 MMP 游戏中创造有意义的故事情节牵涉到很多问题，本文第一部分对它们进行了理论性分析，第二部分则会介绍一些用来实现这些目的的特定策略。

1.7.1 为了一个更有意义的 MMP

当前这代 MMP 游戏最突出的特征之一就是它们是纯粹游戏（pure game）。纯粹游戏就是像国际象棋和俄罗斯方块这样的游戏。它们可以完全地用游戏自身的内在逻辑来理解，并且游戏的结果就在游戏之中。

虽然目前的 MMP 游戏可以让玩家在一个引人入胜的 3D 世界中冒险，但这些行为不仅具有重复性而且除了完成游戏的当前目的以外毫无意义。这与同类型的单机游戏截然不同，在单机游戏中设计人员需要更多地考虑

故事情节、拟人化的角色交互以及为玩家的行动创建一个更广阔更有意义的环境。

如果可以在 MMP 游戏中使用这些特性，就可以拓宽受众、减少壁垒从而照顾到更广泛的玩家。然而，简单地在 MMP 游戏中加入故事情节并不是一个很好的方法。为一个典型的角色扮演游戏创建 40 到 80 小时拟人化的详细游戏经历需要数百个月。如果一个 MMP 玩家在 3 年中每周上线 5 小时，就需要为他准备 780 小时的故事情节。因此，在游戏中加入故事情节的“传统”方法不适用于 MMP 游戏。

为了达到与使用故事情节相同的效果，游戏设计人员必须寻找其他的更具伸缩性的方法，而不是去尝试创建线性的故事。

1.7.2　游戏与故事情节：尴尬的结合

故事情节和计算机游戏是一个相当尴尬的结合。游戏意味着在规则许可的范围内获得开放性的结果。从这方面来说，游戏类似于运动。计算机游戏中的虚拟“世界”使得新形式的交互成为可能，它和开放性一起成为计算机游戏的快乐源泉。开放性并不需要完全的自由，但是它要求玩家能够以有意义的形式确定游戏的结果。

故事需要一个线形或是半线形的结构。书本和电影中的故事之所以有影响力，是因为它们讲述的是一个个独立的故事，并且由讲故事的人负责步调和结果。在书本、电影或戏剧中，我们知道故事有一个明确不变的结果。如果结果改变了，那就变成另一个故事了。故事中的分支结构最多只能为其线形思想提供一些简单的扩展。它们就像带着一两个岔道的铁轨，无论如何，它们始终是紧贴在铁轨上运行的。

然而，计算机游戏作为一种交互媒体，它的开放性与玩家对故事结构的期望是冲突的。因此，在玩一个具有故事情节的游戏时，玩家处于一个预定的故事结构之中，但又想要真正地拥有对结果的决定权，他们往往就处于这两者的折衷中。即使这个折衷富有建设性，它也只是一个折衷。既然故事情节在计算机游戏中的地位如此尴尬，为什么我们还需要它呢？

1.7.3　故事情节在计算机游戏中的功能

关于计算机游戏（包括 MMP 游戏）是否需要更多更好的故事情节的争论是很常见的。参见[Klugg02]和[Luban01]可以获得一些典型的例子，参见[Koster01]可以获得对另一种方法的讨论。实际上，游戏设计人员在计算机游戏中使用故事情节，是因为它有助于创建一个动人的交互式体验。虽然人们通常认为讲故事可能是书本或者电影的目的，但对于计算机游戏来说它们也是达到目的的有效手段。故事情节在单机游戏中有两个主要功能。

1. 故事情节为游戏提供了结构

故事情节为普通玩家创造了一个熟悉而有结构的切入点。日常生活中人们接触到的大多数娱乐形式是基于讲故事的。故事情节可以带给那些对纯粹游戏不感兴趣的玩家一个熟悉舒适的结构，从而使他们融入游戏体验。通过这种方式，它拓宽了潜在受众。一部分人可以把这个产品当作游戏来享用，而另一部分人可以把它作为一个穿插着游戏片段的故事来享用。

故事情节创建了一个隐含的线形发展过程。很多游戏中的故事情节是通过战胜游戏中的

挑战（谜题，战斗）逐步发展的。一旦玩家进入下一个场景，就意味着游戏已经承认他们有所实现。这是一种对玩家进行奖励的自然而令人满意的方式，它充分利用了玩家对于故事情节发展的基本期望。

类似地，故事情节的存在也保证了结局的存在。并不是每个玩家都想要一个可以一直玩下去的游戏。游戏情节的存在保证了游戏具有一个明确且令人满意的结局。故事的完成就意味着游戏的完成，即使玩家并没有完成所有的挑战或是体验完所有的游戏内容。

游戏情节的这一功能揭示了一个事实：人们都希望在看故事的时候能够看到结局——这个策略被用于像漫画书这样的连载小说中，它们往往在结束旧剧情的同时开始新剧情。

2. 故事情节使玩家的行动更有意义

故事情节结合游戏设置向玩家展示了一个全新的冒险幻想。计算机游戏让他们有机会去做那些在真实世界中不可能、不合适或是过于危险的事情。它让玩家可以在新的幻想中冒险——从运动健儿到残忍的犯罪。幻想需要背景和故事情节，这和游戏的整体设置组合在一起，为游戏中的行动创造了背景。

故事情节使玩家可以对游戏经历进行预测，可以很快地知道游戏的背景以及他们要做的事情。玩家都非常熟悉故事的标准结构、角色、剧情以及发展过程。他们知道在故事中情节是怎样发生的，因此他们可以用这些知识来预测游戏中可能发生的事情。如果游戏中一个邪恶的巫师绑架了主角的恋人，他们就可以作出一大堆假设和期望来预测将会发生的事情。

通过把行动和“戏剧性场景（dramatic situation）”组合起来，故事情节可以让游戏中的行动变得有意义。为了营救恋人而杀死6个怪物和为了战斗的乐趣而杀死6个怪物是截然不同的。游戏《毁灭战士》（*Doom*）为戏剧性场景提供了一个很好的例子，这个游戏使用了极小的篇幅来讲述一个故事，但是却表达了一个强烈的目的。这对于计算机游戏来说非常重要，因为玩家要通过他们的行动在游戏中积极地前进，而不是被动地看到故事的展开。

在这个过程中，故事情节可以让玩家更容易地在游戏体验中投入情感。它使玩家有机会去关心游戏中的场景；去鄙视坏人；去和英雄一起庆祝。游戏之所以是游戏，是因为它有很多机械属性（mechanical propertie），但是在此之上还有很多有趣的东西。换言之，在这里故事情节还起到了一个综合职能：它可以把游戏中有意义的方面联系在一起。

总的来说，故事情节对于单机游戏来说非常重要，因为它们不仅为玩家提供了一个熟悉的结构，还为游戏中有意义的行动创造了背景。那么为什么有意义的行动对于游戏来说那么重要呢？

1.7.4 挑战更多的认知能力

为了对此进行分析，本文对纯粹游戏和剧情游戏（theatrical game）加以区分。纯粹游戏的结果在游戏之中，只需要最小限度的上下文就可以完成。俄罗斯方块是纯粹游戏的一个很好的例子。剧情游戏通过把主角放在一个交互世界中来展现可能的剧情，它们通常都有一个目的。

如果把战胜游戏作为惟一目标的话，带有剧情的游戏也可以作为一个纯粹游戏来进行。如果玩家把对游戏真实程度的怀疑减少到一定程度，他们也可以用一种更加沉醉于故事的

方式来玩这些游戏。对于那些喜欢游戏剧情方面的玩家来说，游戏必须向他们提供有意义的行动。

玩家可以分为纯粹玩家（pure gamer）和剧情玩家（theatrical gamer），但是还有更好的方法来考虑这个问题。有一些观点认为，游戏之所以能够带来满足是因为玩家可以用交互方式来主动地使用各种认知能力（Cognitive Capability）。把一个纯粹游戏当作一个纯粹游戏来玩涉及一个特定的认知能力集合（反应、观察能力以及空间和逻辑智能）。而把一个剧情游戏当作一个剧情游戏来玩则涉及一个不同的认知能力集合（社交、叙事和情感智能）。

在进行任何相当复杂的游戏时，玩家都会在不同时候牵涉到对不同能力的运用。在类似于《雷神之槌》（*Quake*）或是《半条命》（*Halflife*）这样的第一人称射击游戏（First-Person Shooter，FPS）中，玩家可以在白热化的战斗里测试自己的反应（reflex）和空间能力（spatial ability）。在战斗之后，当玩家试图找到一条路以进入一个锁住的房间时，他们在测试自己的感知能力（perception ability）和逻辑能力（logical ability）。当玩家和那些提供故事说明的 NPC 打交道时，他们在使用叙事技巧（narrative skill）。

为什么有意义的行动如此重要？在计算机游戏世界中，人们更倾向于那些测试反应和空间能力的游戏。然而《模拟人生》（*The Sims*）的成功展示了这样一种可能：计算机游戏也可以使用和挑战玩家的社交和情感能力。当游戏中行动具有特定意义时，玩家会主动使用范围更广的认知能力。

1.7.5　使用有意义行动的最佳场所

虽然一些良好的实用性理由使游戏设计人员试图把单机游戏中故事情节的一些特性引入到 MMP 游戏中，但是可行的方法远不止这些。MMP 游戏非常适合于创建有意义的、具有故事性的、引人入胜的交互体验。通过为玩家同时提供一个社交场所和一个持久的虚拟世界，它们为具有丰富故事情节的游戏体验提供了理想环境。

MMP 游戏为玩家的冒险和居住提供了一个持久的世界。这是 MMP 游戏和单机游戏在定义上的区别之一。在这里，没有保存和重启，每一个行动都会产生结果并且不能被“取消”。在单机游戏中，当玩家开始游戏时，游戏世界即刻存在，当玩家关闭游戏时，游戏就立即消失了。但是 MMP 游戏世界始终存在并且持续发展，无论玩家存在与否。这使得 MMP 游戏可以提供给消费者独一无二的“真实”感觉。

玩家在 MMP 游戏中投入的时间相对于他们在大多数单机游戏中所花的时间来说要长得多，这进一步强化了 MMP 游戏世界是一个“客观世界”的幻象。这意味着玩家在对游戏的新鲜感消退后仍然会继续游戏。这种持续投入使玩家把游戏当作一个真实世界而不仅仅是游戏，从而与之建立强有力的联系。仅凭这点 MMP 游戏就可以促使玩家在游戏体验中进行更大程度的投入。

MMP 游戏创建了共享的社交世界，游戏中的玩家为彼此提供了共享的观众，玩家会去欣赏他们在游戏中感兴趣的内容并作出反应。玩家还可以用一种在单机游戏中不可能的方式来共享经验，进行游戏的时间越长，这种效果就越明显。而在游戏中通过朋友和社会网络所形成的有共同语言的观众可以进一步加强这种经验共享。MMP 游戏的特别之处就是它让玩家在（以身体、情感、社交等方式）做出行动时能被别人看见。[Baron99]对这个问题的负面

影响进行了讨论。

这使得 MMP 媒体不可避免地具有剧情。在真实世界中，人们也不断地在进行“同样的游戏”：阅读、表现以及与那些我们究竟是谁的暗示进行交互。在线空间中保留并且夸大了这些寻常的行为。因特网的匿名性以及创建可见替身的能力使这个世界变得更具戏剧性。玩家可以反映他们自己的真实人格、其他人格或是两者的组合。无论他愿不愿意接受，这个替身所做出的每个行动都向其他玩家暗示了这个替身背后的真实人物。

如果在游戏世界中惟一有意义的行动是机械的角色升级，那么玩家就没有机会去成为英雄或是坏人，就没有机会去归属于这个世界，他们最终只会成为一个对游戏内容毫无感情的消费者。完全基于机械性角色升级的游戏会让他们丧失人类喜欢戏剧性和故事的本能。如果目标受众是“纯粹玩家”（pure gamer），那这没有问题。如果游戏开发者想要获得更广泛的受众，这就不太合适了。

那些需要玩家使用叙事或情感能力的单机游戏需要某种类型的结构，因为它们的游戏世界非常孤立，这个世界中惟一的居民就是玩家。在 MMP 游戏中，游戏世界中充满了对社交感兴趣并且具有丰富感情的居民，他们就是其他的玩家。

1.7.6 为 MMP 游戏模式加入故事情节

游戏中的叙述性部分是一个可以使玩家共同分享快乐的自然结构。作为一种通用的共享“语言”，它需要玩家使用独特的认知能力。只要游戏的背景丰富并且具有故事情节，游戏设计人员就可以以游戏的原始素材为基础创造出像故事一样的体验而不需要任何显式的手工编写的情节。

怎样才能在 MMP 中加入故事情节呢？只需要少量的努力，就可以在以下 3 个地方加入丰富的故事内容。

- 可以在那些具有进展的游戏元素中加入故事情节。角色升级就是一个非常明显的例子。
- 可以通过创建背景来建立一些游戏世界中的戏剧性场景。戏剧性场景作为背景故事，可以为游戏行动建立上下文从而使行动自身成为一个逐渐展开的故事。
- 可以创造机会让玩家卷入那些具有故事形式的公众事件或场景中去。使他们具有某个共同的目标就是一个很好的例子。

1. 把角色升级作为故事

每当存在一个可以让玩家行动更有意义的结构时，游戏就可以在此之上开展一个新的故事。通过让这些结构在形式上更具故事性并且在游戏中更加有意义，设计人员可以为游戏加入故事资源。

大多数游戏中都存在着某种形式的玩家升级系统。它们不需要太多的故事成分就可以为玩家行动提供一个必需的结构，但是游戏设计人员也可以很自然地在角色升级中加入故事成分。毕竟，角色成长是一个很基本的故事线。如果角色成长的过程变得更有趣，玩家就会更愿意在游戏中持续投入。这在游戏只有单一的线性角色升级机制时尤为关键。

当然，这种方式需要仔细地设计。虽然很多玩家会赞成把升级系统变得更为有趣，但是

大多数玩家最终还是为了在游戏系统中获得更大的利益而进行游戏的。特别是他们会寻找风险最小回报最多的升级方式，即使这种升级方式需要在游戏世界中重复地进行毫无意义的单调行动。

游戏设计人员应该对这一点加以利用而不是简单地忽视它。这意味着，奖励系统可以用来确保并加强游戏行动的故事性。

经典的纸上 RPG 游戏《武士道》（*Bushido*）对“荣誉分”的使用就是一个优秀的例子。在这个游戏中，玩家需要一定数量的经验值和荣誉分来达到一个新等级。失去荣誉将会导致在荣誉恢复前等级被降低。这个方法中最有趣的一点是它通过奖励玩家来让他们注意到那些复杂的故事背景。“荣誉分”的存在隐含地保证了一个有趣的“游戏之上的游戏（metagame）”。

在这个系统中，使用会影响正确性的华丽剑术以在胜利中获得更多的荣誉是明智的。这为在战斗或是任务中展示武士道精神提供了一个完美的动机。它之所以可以奏效是因为玩家都很熟悉的，关于荣誉和面子的想法，即使他们不知道日本中世纪时期真实体系的所有细节。通过把荣誉作为一个目标和一个基本机制，这个系统在为玩家行动带来切实回报的同时给能力至上倾向（power-playing imperative）带来了一层新的意义。

MMP 游戏中对于行动的回报不一定是物质的或是与角色威力直接相关的，关键在于这些回报必须在某种意义上是可衡量的，并且在游戏中具有客观的因果关系。荣誉、声望、社会地位以及名誉都可以作为玩家的中期目标，因此它们都是加入故事情节的理想选择。

那些只有单一的线性升级路径的游戏都过于简单，游戏的设计可以通过使用上述方法来弥补这一点。当玩家能够真正地选择他们的角色向哪个方面发展时，他们就会根据自己的人生目标来选择发展方向，并且为角色在游戏中的地位奠定基础。这些也会成为可供其他玩家使用的资源。变得臭名昭著或者非常可敬（或是可耻）本身就是一个结果。社会名誉本身就是具有故事性的。

2. 发掘戏剧性场景

戏剧性场景是游戏行动的背景故事，它为玩家行动赋予目标和意义。不同于大多数单机游戏中的更具有结构性的线性剧情，在 MMP 游戏中戏剧性场景为行动赋予目标和意义后就进入后台，并且由游戏模式来继续展开故事。它可以使行动具有目标而不需要把线性结构强加于游戏之上。

某些优秀的计算机游戏使用戏剧性场景，而不是线性故事来创建背景和目的。早期的《毁灭战士》是这种方式的一个很好的例子。玩家作为一个顽强的太空战士被送往月球基地，当伙伴们上战场时他被留在后方，最后所有的队友都不幸战死。

这个背景故事刚好足以建立场景。它对玩家角色进行了非常简单的定义并且建立了目标（试图生存并且为队友们复仇）。从那时起，游戏使用偶尔放置的战士尸体以及游戏本身的整体线性结构来引导剧情前进。只有在不同级别中转变时，游戏中才会有真正的剧情发展，它会对原先的设定做一些变化：现在玩家在地狱中并且生存的挑战变得更加困难。

这种方法的优点是不需要故事或是逐渐展开的情节，也没有会妨碍游戏行动的场景。通过在游戏中射击和继续生存，它隐式地发展了“故事”，此时行动处于主要地位而故事被刻意地设计为次要的。当戏剧性场景只是一个非常简单的原型时，这种方法表现得尤为出色。

背景故事和游戏的真正目的必须协调才能正确工作。如果游戏实际上只是关于怎样通过

杀戮一切来变得强壮，那么创建一个关于高贵的英雄与邪恶势力之间斗争的复杂背景故事是毫无意义的。如果游戏是关于杀戮的，那么背景故事就需要解释为什么这一切都必须被杀死，这样才能最大化角色的威力。

戏剧性场景可以用于不同的层次中。它可以用在单个角色、某个区域、整个游戏世界或是一个社会角色中。这个方法在真的有可能去解决问题或是改变世界时效果最好：《毁灭战士》中的战士在击败地狱中所有怪物后就能回到地球。如果这个场景是一个“无穷无尽”的场景，那么玩家角色升级的空间必须包含在这个场景中。

为了更令人信服，戏剧性场景需要“写”入游戏世界中。游戏中还需要有对于这个场景的一致引用或暗示，并且要让游戏模式成为这一场景所创建的冲突或挑战的自然结果。这个场景是否过于简单或是毫无新意并不重要，因为最终重要的是行动。

譬如说，如果恐惧之塔（Towers of Fear）是光明力量和黑暗力量之间的永久争端所在，那么与这个地区有关的角色名誉提升或是派系对立就会充满故事性。如果恐惧之塔可以被占领并且保持一段时期，那么这个地区本身就会充满故事性。如果达成这一胜利的英雄会在游戏中被其他玩家承认，那么故事性将会更为强烈。

1.7.7 公共目标

为游戏加入故事情节的第3个方法是创造具有独特生命力的、动态而一致的游戏元素。它们可以是像殖民地这样的具体实体，也可以是目标或危机这样的抽象元素。关键在于，随着时间的推移，它们要能够根据玩家的行动而变化，而且它们未来的轨迹并不是预定的。这可以使玩家真正地感觉到他们被卷入了游戏的发展之中。

《亚瑟暗世纪》中王国间的战斗以及《星球大战：帝国分裂》（*Star War Galaxies*）中帝国和叛军之间的竞赛是这种方法的一些简单例子。这个方法的与众不同之处在于它把一个充满故事的游戏空间和交互体验的开放性结合在了一起。

另一个方法是在游戏世界中，加入一些需要玩家长时间持续行动才能解决的动态危机（dynamic crise）。譬如说，某个特定区域可能会产生很多敌人从而给一条重要的贸易路径带来危险。只有通过持续努力才能消灭这些敌人。这样的系统可以构建在游戏中的基本产生（spawning）系统之上。

这些方法的好处在于它们为共享的公共世界提供了故事场景，而这些场景需要多个玩家共同努力才能解决。它同时利用了MMP的内在优势（譬如说一致性和共享的社会世界）和计算机游戏的内在优势（譬如说结果的开放性）。与通常的故事不同，这里玩家可以真正地塑造结果，并且如果具有正确的结构，游戏行动的展开也可以具有独特的故事形式。虽然游戏会在剧情方面失去一些（譬如说缺乏剧情交替和主角），但可以从那种被卷入游戏世界的真实感觉中获得更多。

每个场景或是危机都会成为需要玩家努力的目标。某些情况下这是一个玩家与环境间的目标；另一些情况下这是玩家间的目标。如果可以把这些目标同时整合进游戏世界的戏剧场景和角色升级系统中，就可以获得一些强大的协同作用（synergies）。那些为了故事情节而进入MMP游戏的玩家会发现他们不仅能发现故事，而且所发现的故事是由他们自己塑造出来的。

1.7.8　总结

商业化 MMP 游戏的产生为一种新的媒体奠定了基础。它与单机游戏的区别就和电视与戏剧的区别一样。和所有的媒体一样，它也具有其内在的优势和弱点。它最大的优势在于它不仅可以创建一个共享的世界，还可以让玩家自由地选择想要进行的行动并且获得长期的成就。

很多现存的游戏都是“纯粹游戏”。它们挑战和使用了玩家的逻辑和分析能力。本质上它们更适合于那些能力较强的玩家。要让 MMP 媒体能够达到更广泛的受众，就必须扩展 MMP 游戏中要使用到的认知能力。尤其是在 MMP 游戏中加入更多富有故事性的、有意义的行动，可以使玩家发挥他们的社交、情感和叙事技巧。

有一些简单而实用的方法可以在 MMP 游戏体验中加入故事情节。这篇文章对其中 3 个方面进行了讨论：角色升级、使用戏剧性场景以及创建持久的动态场景。但是还存在着其他的方法。这些故事情节不仅增强了 MMP 游戏的吸引力，还可以防止玩家在对游戏设置的新鲜感以及对游戏模式的兴趣消退后离开游戏。

1.7.9　参考文献

[Baron99] Baron，Jonathan，“Glory and Shame: Powerful Psychology in Multiplayer Online Games,” *http://www.gamasutra.com/features/19991110/Baron_01.htm*，November 10，1999.

[Klugg02] Klugg，Chris，“Implementing Stories in Massively Multiplayer Games,” *http://www.gamasutra.com/resource_guide/20020916/klug_01.htm*，September 16，2002.

[Koster01] Koster，Raph，“Two Models for Narrative Worlds,” *http://www.legendmud.org/raph/gaming/narrative_files/frame.htm*，January 29，2001.

[Luban01] Luban，Pascal，and Joël Meziane，“Turning a Linear Story into a Game: The Missing Link between Fiction and Interactive Entertainment,” *http://www.gamasutra.com/features/20010615/luban_01.htm*，June 15，2001.

1.8 客户服务和玩家声望：一切都和信任有关

Paul D. Sage，NCsoft Corporation

psage@ncaustin.com

在不到一小时的时间里，一个玩家就可以让游戏开发人员损失 40 个玩家。如果他们不相信这句话，或是没有全力以赴地消除这类问题，那就在游戏包装盒或是网页上画了一个靶子，并且在靶心写上“捣蛋鬼，过来！”对 MMP 游戏玩家来说，最让他们难受的莫过于被别人激怒、骚扰或是无法玩得尽兴了。设计人员怎样才能防止“杰克”（假想的捣蛋鬼）破坏“雷蒙”（假想的普通玩家）的游戏体验呢？让杰克在游戏中的声望来说明问题吧。

1.8.1 捣乱（grief）

首先，定义一下“捣乱”这个术语。基本上，捣乱就是指在游戏规则允许的范围内，某个玩家故意通过一些方法来激怒、威胁或是骚扰其他玩家。那些人为什么这样做并不值得去探究，重要的是这样的事的确发生了——并且经常在没有明显真实动机的情况下发生。

在《创世纪在线》（*Ultima Online*，*UO*）中，一个时常出入太平洋缺口（Pacific Shard）的玩家可以在数小时内杀死超过 40 个人，一周以后，他身后留下的尸体可能比汉尼拔·雷科特（Hannibal Lector，杀人博士）杀的人还要多。每天晚上他的受害者都会向游戏管理员（Game Master，GM）的等候队列（也就是我们在游戏内部的服务请求排队系统）发出 50 次以上的呼叫。每次呼叫内容各异。

- “霍布斯是个骗子”。
- “霍布斯以杀人为乐”。
- “霍布斯拿走了我的东西”。
- “这人疯了，而且还说粗话”。

霍布斯是不是真的在欺骗别人？答案不得而知。霍布斯是不是以杀人为乐？可以做一个肯定的假设，但没有证据。霍布斯是不是拿了别人的东西？（或许那时设备还无法对此进行跟踪。）他是不是说粗话了？也许，他的确称呼某人为“连路都走不稳的剑客”。

也许有人会指出，正是因为《创世纪在线》在运营初期允许玩家大量屠杀才会导致这样的捣乱行为。这也许有些道理，既然这就是游戏的意图，那么事实上霍布斯只是在游戏规则允许的范围内进行游戏。是否应该允许

霍布斯继续留在游戏里面，尽管他使其他玩家很痛苦？这个问题的答案乍看上去可能不是很清楚，但是真正的答案是“不”。不过他确实留下来了，这是因为很多理想主义的念头让游戏开发人员去选择看上去正确的答案，而不是真正正确的答案。正是这些“美好”的意愿铺就了通向失去所有订阅者的道路。

1.8.2　规则只是工具

看到上面的情况，游戏开发人员的第一反应很可能就是试图通过改变游戏的规则来禁止这些行为模式。要理解需要实现哪些规则，关键就是看看下面的规则以及怎样才能打破它们。以下就是一些建议的规则。

1. 禁止玩家杀害其他玩家或许是一个可行的方法，但这依赖于不同的游戏。譬如说，如果《亚瑟暗世纪》（*Dark Age of Camelot*，*DAoC*）真的实现了“不许杀害其他玩家”这一规则，那么从此以后惟一被“杀害”的就是这个游戏的大部分设计目的。显然，这个规则并不适合所有游戏的意图。
2. UO 曾经尝试过限制在一个特定时间段中可以杀害的玩家总数。随着时间的推移，这个规则以及对于违反这个规则的玩家的惩罚变得越来越复杂。很多被这个规则惩罚的人是无辜的；而很多应该受到惩罚的人却成功地避开了它。
3. 也可以让玩家在战胜后得不到战利品，这样的规则或许可以奏效，但是必须考虑到有些游戏的设计目标之一就是通过给予一定程度的自由来奖励那些愿意冒着生命危险与别人作战的玩家，在这种情况下，玩家得不到战利品这一规则排除了一个允许玩家杀害他人的合理理由，这使得只有那些反社会的杀手才会杀害别人。当然，通常玩家杀害别人的时候并不需要理由。
4. 不能和你刚刚杀害的人说话。《亚瑟暗世纪》中运用这个规则的方式是非常有意义的。很明显这是其游戏设计的一部分，它非常有效地避免了那些杀手或是捣蛋鬼把一次简单的玩家对战变成一次粗鲁而带有侮辱性的事件。然而，在 UO 这种没有聊天频道的游戏中，这个解决方法并不明智。取消一个人对另一个人说话的能力不过是让杰克的同伙取代杰克来嘲弄受害者。很少有人愿意看到间接的捣乱行为。
5. 禁止粗话——这是一个很棘手的问题。人们不愿意承认和朋友在一起时他们会像水手一样咒骂——但事实上有时候他们会这样做。限制玩家的言论有点可怕，而且也不是游戏设计人员的本意。如果想要为玩家创建一个与朋友欢聚的普通场所，或许应该允许他们说些粗话；或是让玩家可以根据自己的意愿来打开或者关闭粗话过滤器。尽管如此，本文认为，服务提供商仍然有权对玩家群体的言论在他们认为有必要的程度上进行控制。问题的关键在于怎样捕获粗话。如果他们认为玩家除了脏话就想不出可以侮辱别人的方法，那很可能是低估了玩家的创造性。

上文提出了 5 个规则，随后又把它们都否定了。读者可以提出或尝试更多的规则。这里的关键不是通过这些规则来发现解决的方法，而是要提出一个令人难以接受的思想：如果不适应游戏设计的意图，再好的规则都可能会被破坏或滥用。不要为了把所有的杰克从游戏中赶走就去破坏这个游戏的设计，因为几乎可以肯定总有一小撮玩家会竭尽全力地为其他玩家制造麻烦。一个更可靠的方法是为游戏定义一个清晰的边界，并且牢记对于不同玩家来说，

社会行为是否适当的边界是不同的。

1.8.3 意图

很多设计人员会为他们的游戏服务以及他们防止捣乱的能力担心，对于他们来说，最关键的就是要知道一个重要的事实：不能单凭玩家自身的行动去猜测他的意图。UO 游戏允许玩家（甚至是鼓励玩家）回击那些试图招惹他们的人。当屠杀变成一个显著的问题后，游戏设计人员决定限制玩家在特定时间内的杀人数量。这里的前提是：玩家在一个特定时间段内被人骚扰 5 次以上的可能性很小。因此，在这一时间段中玩家能够进行至少 4 次回击而不会被认为是杀人犯。谋杀计数，如同它们的名字一样，会随着时间的过去而衰减，这意味着任何用完他们所规定的杀人限制的玩家在过了一定时间后就可以杀死更多的玩家而不必受到任何惩罚。因此，如果杰克称呼雷蒙为还没进化好的蠢货，雷蒙就能够进攻杰克以把他那可恶的舌头割掉。问题是如果杰克知道那天雷蒙已经杀了 4 个人，他就可以用任意可以想到的方式侮辱雷蒙而不必担心会受到惩罚，因为如果雷蒙想要杀死杰克，系统会认为他是一个杀人犯。因此，杰克可以不断地刺激雷蒙来进攻他。这里的问题在于杰克的意图是惹恼其他玩家，而雷蒙的意图只是想保护自己的名誉。尽管雷蒙以遵守游戏系统的规则为目的，最终他还是被系统惩罚了。

举另一个不同系统的例子。在《无尽的任务》中，有个被称为棋盘（chessboard）的位置。棋盘上有大约 3 到 4 个再生点，彼此间的距离有几个屏幕的大小；玩家可以杀死由再生点产生的僵尸怪物。处于某一等级和类别的玩家可以去棋盘上独自杀死一些简单的怪物而不必担心会有危险。可是如果一个玩家和另一个玩家为了谁先到达那里以及谁应该获得这些战利品而争吵，该怎么办呢？第一个玩家并不想窃取别人的战果，并且提出他们可以轮流获得，但是另一个玩家可能真的认为自己有权独享任何在这个棋盘上出现的怪物，因此他们向客户支持代表（Customer Support Representative，CSR）投诉。CSR 到了那里就会简单地告诉这两个玩家他们应该好好游戏，然后结束这次客户支持。为什么这个 CSR 什么都不做？因为这个 CSR 不能根据给定的情况来猜测这两个玩家的意图，如果他随意作出决定的话，没有人会对此感到满意。

CSR 或者任何真人观察员理解处于某个特定情况下各方意图的能力并不比开发人员通过游戏系统来对意图进行猜测的能力要强。因此让一个 CSR 来察看这个情况未必能让它变得更好。惟一可以持有某些可以信赖的精确观点的是那些身临其境的人。

1.8.4 当前的服务部门正在这样做，或许他们还没有意识到

服务部门的经验揭示了有两种投诉和玩家间的交互有关。一种类型的投诉与某个玩家或一组玩家有关，他们尽其所能地使其他玩家的游戏经历变得痛苦。而另一种更常见的投诉则与游戏中对是非观念的不同见解有关，双方都确信自己是对的。然而在大多数情况下，除非服务人员了解问题的双方，否则几乎不可能区分这两种情况，而了解问题的双方才是清晰地识别和解决这些真正的问题的关键。

在后一种情况中，有两个个体或群体，他们可能只是对某个特定问题的见解不同。这个

问题也许有实际的解决方法，但也可能没有。如果真的有，那么很可能已经在游戏规则中处理了。如果双方都是合理的，他们可能只是想解决他们的问题或是用一种有意义的形式来表达他们的不快。

在前一种情况中，服务人员遇到了一个真正的问题。有人确实想要破坏这个游戏，发现这种情况的惟一方法就是寄希望于客户服务部门对于某个捣蛋鬼有关的所有情况都保留记录。这个过程需要花费不少时间和金钱。这些坏蛋不仅增加了客户服务部门的压力，还直接提高了客户支持方面的开销。99%的时间里，发现这类客户的惟一方法就是通过客户服务部门的努力工作——把和某个帐户有关的大量投诉连接起来以尝试追捕问题玩家。

也许有人会问："这不正是我们设立客户服务部门的原因吗？"。不！设立客户服务部门是为了解决那些开发团队无法通过游戏机制来发现或解决的问题、观察有问题的玩家、以及贯彻执行服务条款协定。在这里所讨论的情况下，客户服务部门必须就玩家对杰克的投诉进行一段时期的跟踪后才能确认杰克的确是个坏蛋。一旦最终确认杰克是一个问题玩家，这就意味着服务部门花费了大量的时间和金钱发现了一个很多玩家早就知道的事实：杰克非常讨厌。

1.8.5　声望

韦氏词典在他们的网站上对声望（reputation）所作的定义如下所示。

主条目：声望。

1a：一般人所能看到和判断的总体质量或特性。b：别人对某些特性或者能力的承认。

2：受到公众尊重或尊敬的地方；好名声。

无论在哪个定义中，都是由别人来对那个被讨论的人作出评价。在游戏这类以社会交互来吸引别人的娱乐形式中，游戏开发人员必须时刻牢记这点。玩家每天都在对他和其他玩家之间的交互作出评价。大多数时候，这些评价是有利的；但有时却不是。当多个玩家在一段时间内都认为某个特定玩家是一个坏人时，他们很可能是对的。

作为一个开放性的大型社区，eBay 就是一个有趣的例子。它是一个提供在线拍卖服务的网站，在那里卖家可以为某个物品做广告，然后潜在的买家可以对这个物品出价。在拍卖中获胜的人必须从卖家那里购买这个物品。可以想象，这里发生欺诈的可能性非常大。eBay 通过允许买家和卖家对这次交易留下好评或差评来巧妙地防止欺诈。虽然这一声望系统并不是很完美，但它至少让潜在的买家和卖家在和某人进行交易前对他有所了解。它使得一个良好的销售人员可以证明他是可靠的，并且可以阻止不诚实的交易行为。对于所有这些信息究竟有多精确以及是否会被人捣乱可能还存在一些疑问，但是没有这个系统，eBay 可能不会像今天那样流行和成功。

大多数浏览 eBay 反馈系统的人都在寻找一样东西：一致性。如果他们看到具有一致性的反馈，无论是好还是坏，对这个人的看法通常会更为有效。

1.8.6　正当行为

人们提及声望系统时，通常讨论的是预防或阻止不良行为的方法。事实上，声望系统不

仅可以用来阻止尽可能多的不良行为，还可以用来鼓励或奖赏游戏社区中的良好行为。仔细观察那些在社区中受到尊敬的人们就会发现每个人所表现出的让他们受到尊重的特性都存在巨大的差异。不是每个人都喜欢同一类型的交互，但是如果一个人在和他人的交互中感到很开心，无论让他开心的究竟是什么，都应该让他能够表达他的感谢。

下面进行一些哲学探讨，譬如说人们帮助别人是因为他们想帮助别人，而不是因为他们想从中获得什么。给予做好事的人一些回报，哪怕只是给他们一个声望标记，也会改变人们帮助别人的动机并且会促使那些坏人尝试滥用这个系统。然而，与其进行那样的讨论，不如这样想："我当然也很痛恨人们为了错误的理由而对别人好。"严肃地说，这里的关键在于怎样让人们对别人好。如果在某个情况下善意的行为相对于粗鲁的行为来说更有利，那么事后对行为动机进行指责并没有什么意义。

概要：

声望系统的目标不是为了让人们彼此盲从或是奉承对方，而是对那些一心向善的行为进行奖励。当然，仔细地平衡给予良好声望的奖励是很重要的。

1. 当人们对于和别人的交互感到高兴时，应该让他们能够表达出来。
2. 以一种平衡的方式来奖励良好行为。

1.8.7 恶意行为

如果声望只是用来鼓励正确行为，那它就不是声望了，它还需要能阻止恶意行为。当然，如果游戏社区中的人们可以清楚地知道社区中谁是坏人，估计他们就不会和那些人打交道了。虽然声望可能会阻止某些人变成坏人，但是如果做坏事并没有什么实际惩罚的话，捣蛋鬼杰克可能并不在意他的声望如何。

这里本文并不建议应该或不应该使用某种惩罚，因为这完全是由特定的游戏服务所决定的。然而，不应该鼓励简单地使用声望系统来轻易地把其他玩家标记为"恶意"的。玩家需要知道哪些冒犯行为应该给予一次不良评论而哪些不需要。GM(G...M...)们遇到很多麻烦的投诉者——那些人觉得游戏中的每个人都在给他们惹事，并且认为 GM 有责任赶走这些问题制造者。声望系统看上去可能是对他们来说最好的解决方案，但是事实上，这可能是被错误的人掌握的危险工具。

让人们随随便便地认为别人是"坏的"并不能给予社会一个精确的描述。即使是轻微的争吵也会导致坏声望变得常见。一个良好的声望系统应该鼓励人们在认为另一个人"坏"之前小心地斟酌他的决定。如果在某个社区中有人被指为一个捣蛋鬼，事实上提出评价的人自己的可信度也处于模棱两可之间。譬如说，如果 eBay 中有一个人惯于给别人留下差评，那么他所留下的差评的作用非常小，除非受到差评的人还有很多其他的差评，让人清楚他的确有不足之处。通常人们会认为一个总是倾向于留给别人差评的人，是以毁坏别人的声望为目的的，而最终受到损害的是他自己的声望。

概要：

声望不仅要鼓励正当行为，还应当阻止恶意行为。

1. 必须给那些公认的"坏"人一些惩罚以促使他们改变或者把他们从游戏社会中赶走。
2. 在玩家指控其他玩家的不良行为时，他必须小心地斟酌他的决定。

1.8.8　多样性导致了所有的差异

无论一个人怎么评价另一个人，这并不代表他所说的就是事实。只有在对同一个人有很多好的或是坏的评价后，才能以某种方式对其进行公正的判断。

如果玩家杰克进入这个游戏并且有 5 个人说他很伟大，这并不会让杰克成为一个伟大的人。他的伟大程度只不过等同于 5 个人在当时游戏社区总人口中所占的比例。

有一天，雷蒙可能比较倒霉并且最后被某人评价为“坏蛋”。但是在确定这一点之前，应该查看一下雷蒙是不是具有一个不良的行为模式。大多数好玩家都有过崩溃的时候，每个人都可能走厄运。然而，大多数捣蛋鬼不能随着时间的推移而对他们的行为做出足够的改变以建立一个可信的声望。虽然邪恶的科克船长对善良的斯伯克先生进行长时间愚弄并不是不可能，但是这种可能性并不大。（《星球大战前传》（*Star Trek Episode*）中的“镜像（Mirror），镜像”就可以说明这个道理）

概要

不能以封面来判断一本书的好坏，同样，对人也不能以偏概全。

1. 要判断一个玩家的好坏，不能只根据几个人的说法，而要根据他们对社会的实际影响。
2. 要根据玩家的行为模式来判断他们是不是在长期地捣乱。

1.8.9　当前使用的行为模式跟踪方法

有很多方法可以对行为模式进行跟踪从而实现一个系统来达到本文所描述的目的。阿佛加多信任度量（Advogato's Trust Metric）是一个用来处理大型社会交互的系统。贝叶斯分析也可以给出解答，尤其是在大型社区中。无论开发者选用什么方法，像图 1-19 那样的小型图表都有助于决定如何使用所收集的信息。

	Time Low	Time High	Low # of Players	High # of Players	Results
Positive	X		X		GF
Negative	X		X		YF
Positive		X	X		GY
Negative		X	X		RF
Positive	X			X	YF
Negative	X			X	RF
Positive		X		X	GF
Negative		X		X	RF

图 1-19　跟踪声望模式

GF = 绿旗：这个玩家很可能是一个好玩家，至少他所做的尚不足以证明我们的猜疑。

YF = 黄旗：这个玩家有些可疑，值得对他进一步调查。

RF = 红旗：这个玩家值得引起客户服务部门的注意，要么有人试图让他看上去像坏人，要么他真的是一个捣蛋鬼。

1.8.10 更多的问题

游戏开发人员需要处理很多重要的问题，匿名就是其中之一。虽然很多合理的理由可以让玩家具有不同的角色和个性，但是允许一个玩家在游戏服务中保留两个独立的声望所带来的好处并不比坏处多。当然，有人会想要“角色扮演”一个坏角色和一个好角色，这绝不是一件坏事。然而，如果游戏不允许玩家把自己描绘成另一个人，并且因此而对社会造成损害，这个想法必须被抛弃。关键不在于某种做法是正确的还是错误的，而在于哪种做法更加正确，或者更加错误。

游戏不允许匿名，并不是要建议在游戏中泄露某人的地址或是真名（虽然这很可能会摆脱 99%的在线捣乱者），而是服务提供者应该努力地识别每一个帐户持有者。如果某个帐户持有者具有多个帐户，他们的声望应该跟着他。更进一步的想法是：有一天玩家的声望不仅是在给定的游戏服务中跟随着他，而是在所有的游戏服务中都跟随着他，就像信用记录或是房屋租赁企业的租客跟踪系统一样。玩家和虚拟生活结合得过于紧密以至于认为应该继续保持匿名。如果游戏开发人员坚持认为匿名是正确的，那么就没有什么商业能够在线维持了。

这样的系统并不能够制止捣乱了。尽管在努力使它最小化，但世界上最好的系统也不能完全制止捣乱。一个致力于打破规则的人可以击破任何系统，因为他们可以分析系统以寻找弱点。当这样的系统面对 100000 到 1000000 甚至更多人时，很可能就会在几场战役中失败。玩家可能会全心全意地获得一个非常好的声望然后改变方向，用任何他们可以的方式来毁坏这个游戏。这就是客户服务部门的价值所在。

1.8.11 使用声望

最后游戏开发人员必须决定怎样使用所开发的声望系统。这里不对这个系统的特定用户界面进行讨论，只是建议要避免把对其他玩家进行评价变得很麻烦。和其他所有功能一样，玩家需要能够很快确定他们在和谁交互以及对他们的感觉。不仅如此，或许评价时双方并不需要都在线。很多人会在交易发生以后捣乱，而这时另一方很可能不在线。

同时，要让客户服务部门把声望系统作为一个让游戏社区处于正轨的主动工具来使用。迅速地识别捣蛋鬼可以为游戏社区以及客户服务部门节约大量的时间和金钱——这些钱更应该被花在游戏娱乐系统的工作上。

1.8.12 总结

让玩家可以表达他们对其他玩家的看法并且对这些看法加以记录有助于下述目的：

1. 当玩家和另一个玩家进行了一次良好的交互后，这个玩家想给他的不仅是一句“谢

谢你”；

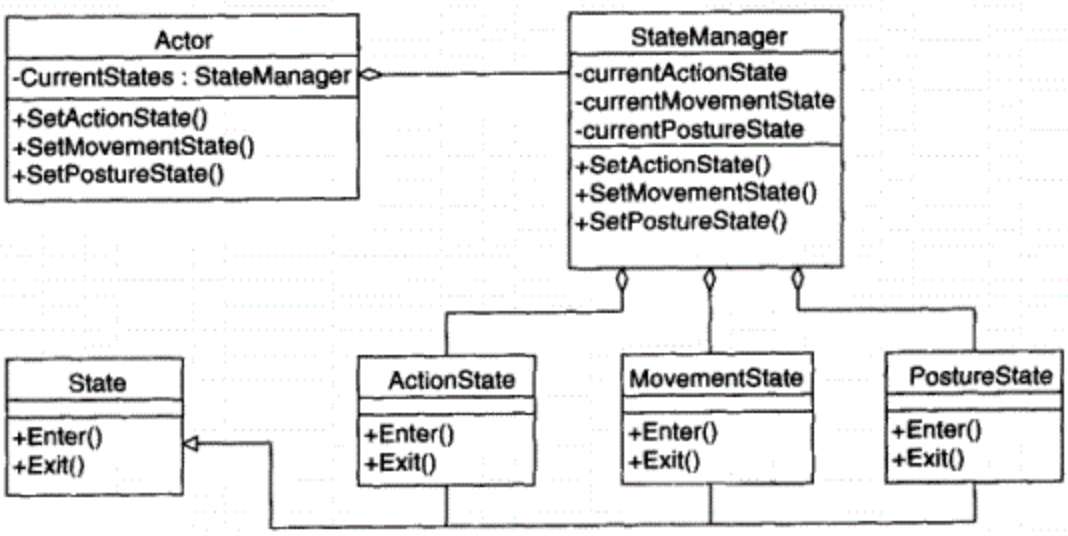
2. 当玩家和另一个玩家进行了一次不好的交互后，他需要某种立即的方式来让对方及其他玩家知道他的感觉；
3. 它允许玩家用很长时间来树立一个他们是怎样对待其他玩家或与其他玩家进行交互的真实形象；
4. 有良好行为的玩家应该受到奖励；
5. 玩家应该对他们的社区有真正的控制权。

换句话说，要允许玩家建立在游戏社区中有意义的声望。

这需要信任。开发人员必须对游戏社区能够一如既往地作出正确的判断有足够的信任。游戏社区必须信任彼此的看法以及开发人员发现任何问题玩家的能力。能力是信任的基础，而不是信任的替代。

第 2 章

MMP 体系结构


Actor
-CurrentStates : StateManager
+SetActionState()
+SetMovementState()
+SetPostureState()
StateManager
-currentActionState
-currentMovementState
-currentPostureState
+SetActionState()
+SetMovementState()
+SetPostureState()
State
+Enter()
+Exit()
ActionState
+Enter()
+Exit()
MovementState
+Enter()
+Exit()
PostureState
+Enter()
+Exit()

2.1 为 MMP 游戏制作仿真框架，第一部分：结构建模

Thor Alexander，Hard Coded Games
thor@hardcodedgames.com

与其把商业性的 MMP 游戏看作一个产品，还不如把它看作一项服务。一个成功的在线游戏服务至少会有 5 年的生命期。在此期间，游戏开发人员必须在开发环境或运行环境中对代码库进行及时的维护和改进，而每次修改都会受到很多极端玩家的批评。为了在这种环境下生存，游戏的代码库必须建立在一个可靠的工程基础上。本文介绍的就是这样一个设计精良的 MMP 游戏仿真框架。

本文将对设计模式及 UML 等技术加以讲解。设计模式作为一种有用而可靠的软件工程工具，近年来获得了广泛的支持。模式是指对标准问题的可重复解决方案（参见[Gamma 94]可获得更多关于模式的信息）。UML 是一个对软件系统进行可视化定义的设计过程。本文用到了 UML 的类图和顺序图（参见[Booch98]可获得关于 UML 更深入的讨论）。

这篇文章包含了不少最初在作者的另一篇文章[Alexander02]中介绍的概念，在此又做了一些补充。本文分为两个部分，第一部分讨论如何通过结构建模来生成类图。第二部分讨论如何使用这些类图建立一个仿真框架。

2.1.1 体系结构纵览

本文所介绍的体系结构构筑于已知的面向对象原则之上。本文把它设计得既可以灵活地在不同平台上实现，又具有高度的可伸缩性，这样用户就可以根据特定项目的需要对它进行定制。

1. 客户/服务器组件

就内核而言，MMP 是一个客户/服务器系统。它提供了一个网络抽象层，用以在因特网上分发客户和服务器进程之间的消息数据包。游戏仿真层接收这些消息数据包并对其进行处理，它还负责保持客户端和服务端游戏状态空间（state-space）的一致性。物理层（physical layer）对这个状态空间中对象的物理表示（physical representation）进行详细仿真。因为客户端的渲染层（client-render layer）只需要向玩家显示它能感知到的物体，在任何给定时间，客户端只需对服务端仿真对象的一个子集进行仿真。这使

得客户端可以在比服务端物理层更高的细节层次上进行物理仿真。图 2-1 所示就是这个体系结构的纵览。

图 2-1　客户/服务器游戏架构

2. 通过代理（Proxy）仿真

服务端的游戏仿真层保留了对游戏状态空间的惟一精确表示。如果客户端之间的状态不一致，服务端必须对它们作出仲裁。服务端把状态空间中行动者（Actor）和其他对象的变化以仿真事件的形式广播给客户端。客户端使用这些事件来更新代理对象（proxy object），这些代理对象构建了一个与本地近似的游戏状态空间。用户通过向服务端仿真层发送动作请求（action request）来与之交互，服务端仿真层在执行动作之前必须验证这些请求。图 2-2 描述了请求/事件流。出于安全原因，在设计 MMP 游戏时永远都不能信任客户端对于状态空间的表示。

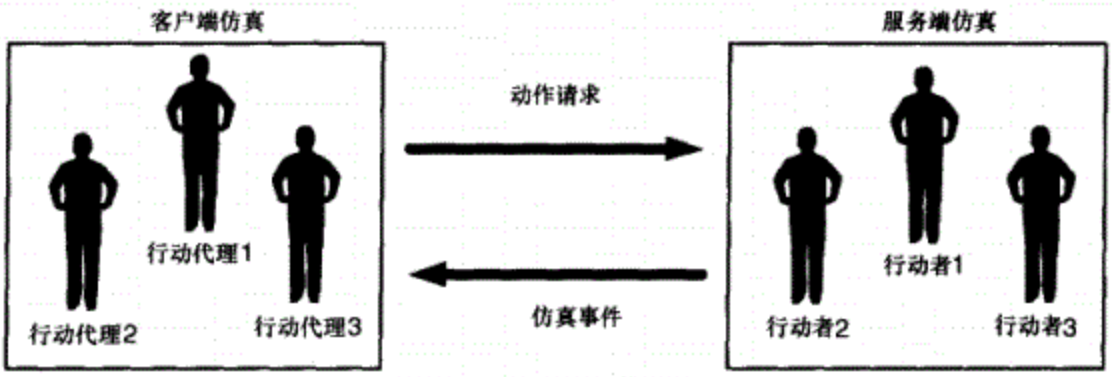


图 2-2　通过行动者和代理对象来进行客户/服务器仿真

2.1.2　支持类

在对体系结构中的核心类进行讨论之前，必须先定义一些支持类。这些支持类将用于构建系统中的核心类。

1. 词典类/哈希表

词典类（Dictionary）是一种抽象数据类型，它可以用来存储按值访问的元素。它的基本操作为 AddEntry（添加条目），LookupEntry（查询条目）和 RemoveEntry（删除条目）。哈希表（Hash Table）可以用来实现词典类。哈希表是一种关联数组，它通过一个哈希函数来把键

映射到数组中的某个位置。图 2-3 中的类图就是使用哈希表实现的词典类。这个词典类将会成为体系结构中最常用的数据结构。

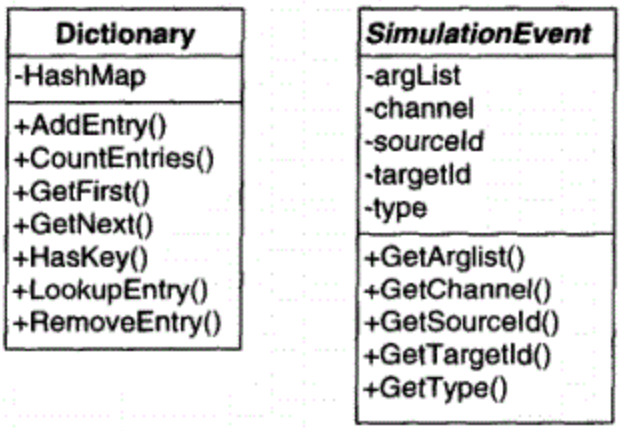


图 2-3 支持类的类图

2. 仿真事件

仿真事件（SimulationEvent）是系统中的事务对象。当某个行动者通过执行动作，来和环境进行交互的时候，这些动作执行的结果会被传达给其他可以感知这些事件的行动者。事件对象包含了事件类型（type）、源行动者标识（sourceId）、目标行动者标识（targetId）以及一系列特定于这个事件类型的参数（argList）。它还有一个频道属性（channel），它使得事件的接收者可以对事件按照不同类别进行过滤，从而获得感兴趣的事件。

3. 仿真状态

仿真状态（SimulationState）类是系统对状态模式（state pattern）实现的一种描述。为了清晰起见，这里介绍的只是这个模式的简化版本，不包含状态机和状态管理器（读者可参考[Boer00]和[Dybsand00]以获得一个更为详细、可靠的实现）。仿真状态类是游戏体系结构中所有状态的基类，它的基本操作是 CanTransition（能否进行转换）和 Transition（进行转换）。这个基类中的 CanTransition 是一个用于预先测试的方法，它可以验证在特定上下文中对象是否可以进行一个有效的转换以进入这个状态。派生的状态类可以实现额外的检查，以满足它们各自的需要。所有实际操作都在 Transition 方法中进行。每一个子类都需要实现各自的 Transition 方法，并在对象进入这个状态时提供一些特定的行为。

4. 动作状态

动作状态（ActionState）类表示那些典型的游戏仿真操作，譬如说击打对手或是打开一扇门。每一个动作状态对象都有各自的持续时间（durationTime）属性，它可以用来详细描述执行这个动作所需要的时间；通常这个属性还可以用来同步服务端仿真与客户端的动画回放时间。动作状态还维护了一个开始时间（startTime）属性。如果一个旁观者（spectator）在行动者转换到某个动作状态一段时间后才看到这个行动者，它可以用开始时间属性计算出动画中的时间坐标（参见图 2-4）。

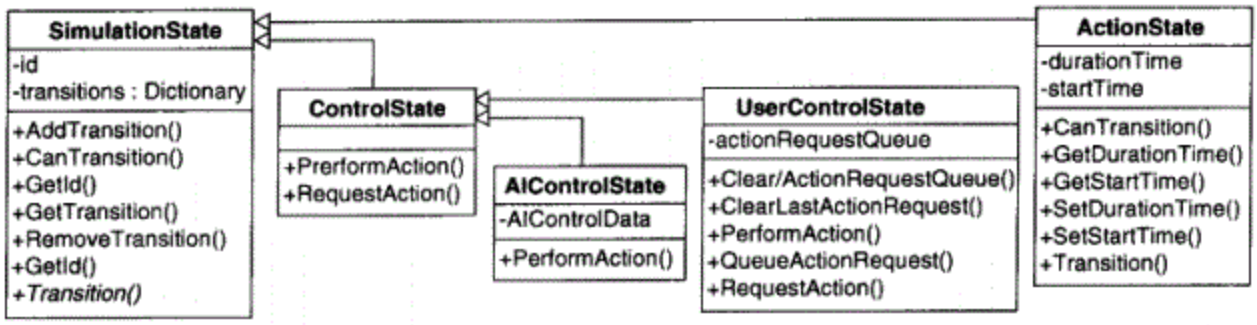


图 2-4　仿真状态类图

5. 控制状态

控制状态（ControlState）定义了与它关联的行动者可以从哪里接受命令。这使得行动者既可以由玩家控制，也可以由 AI 控制。在某些模式中（譬如说在游戏中的过场动画中），行动者还可以由某个脚本驱动的状态（scripted state）控制。在运行时，改变控制状态可以让行动者在这些状态中进行转换。控制状态还可以用在基于观察的训练（training by observation）中。在这种训练状态中，玩家可以控制他的行动者，就好像他在用户控制状态下所做的一样，而计算机会对这个行动者进行观察并且从玩家的行动中学习（参见[Alexander02a]可获得更多相关细节）。

6. 用户控制状态

用户控制状态（UserControlState）维护了一个等待执行的动作请求队列，这些动作由用户所在的客户端发出并且为服务端所接收。这个控制状态的 PerformAction（执行动作）方法会在调用时从这个队列中取出请求。

7. AI 控制状态

当一个处于某个 AI 控制状态下的行动者调用它的 PerformAction 方法时，AI 控制状态类（AIControlState）要负责确定用于执行的合适动作。实现这一决策过程所用的 AI 技术必须符合特定仿真情况下的游戏机制，譬如说有限状态机、神经网络或者是模糊逻辑（fuzzy logic）（参见[Alexande02b]可获得一个决策子系统的例子）。

2.1.3　核心类

现在上文已经定义了辅助的支持类，接下来本文将继续建立那些构成游戏仿真中枢的核心类。

1. 仿真对象

仿真对象类（SimulationObject，SOB）是所有由仿真层处理的核心类的基类，这些类包括了行动者（Actor）、区域（Area）、物品（Item）和障碍物（Obstacle）。图 2-5 是核心类的层次图。游戏开发人员可以为每一个 SOB 分配一个独立的标识（Id）以指定其为惟一的对象。

仿真对象通过发送仿真事件（SimulationEvent）来和其他对象通信。它们可以向其他仿真对象订阅（subscribe）所关心的事件。每个 SOB 根据各自维护的订阅者（subscriber）词典来发布所产生的事件。

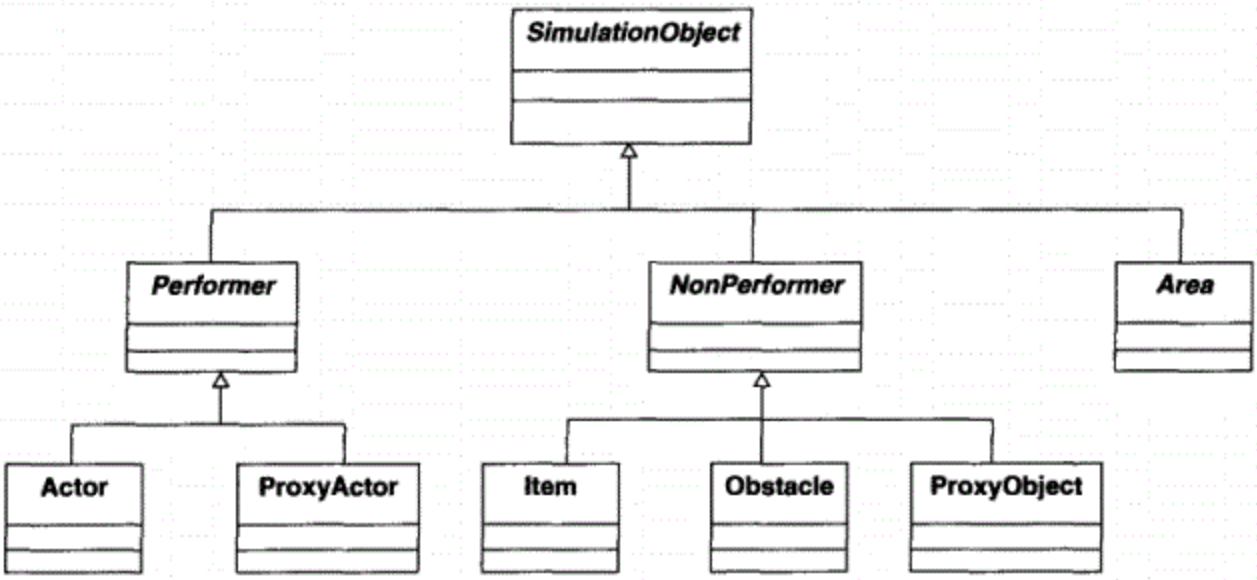


图 2-5 核心类层次图

SOB 还可以包含其他仿真对象。每个 SOB 会为这些对象维护一个内容（contents）词典以及一个指向包含它的 SOB 的所有者标识（ownerId）。仿真对象之间的包含关系作为一个抽象的概念使子类中的很多功能可以很容易地实现。行动者的子类可以用它来实现背囊物品管理系统（inventory item management system）。类似箱子和包这样的物品也可以作为游戏中的容器来装入其他物品。用于表示抽象空间的区域对象则使用包含来跟踪那些进入或者退出其边界的其他仿真对象。

与包含类似，仿真对象还使用一个联系（links）词典来维护与其他对象之间的联系。联系可以在仿真对象之间提供聚合或“整体/局部”的关系。这为对象临时从它的组成部分获得事件提供了一个简单而强大的方法。联系可以被用来实现连接不同区域的、可以打开和关闭的入口或门户。游戏还可以把作为部件的物品“联系”起来创建出复合物品。

仿真对象还提供了一个属性（property）词典，它以数据驱动的方式存储每个对象特定于游戏系统的属性。每个仿真对象的子类可以根据需要向词典中添加自己的属性。属性词典的内容可以以一种智能的、即时的方式复制到客户端对应的代理对象中去，从而降低网络传输的开销。对象在世界坐标中的位置、对象的移动速度、生命值和魔法值等都可以被当作属性。

最后，仿真对象提供了一套持久化机制，这样仿真层就可以用一个通用的方式和存储系统合作了。每个仿真对象维护了一个脏（dirty）标志，每当有数据或者属性发生变化，这个标志就会被设置。仿真层可以在需要时调用 SOB 的 Store（保存）方法。Store 方法检查脏标志，如果被设置了就存档。Restore（恢复）方法则进行相反的操作来把对象载入仿真层。在最终应用中，这些方法的实现既可以把对象存储到可扩展标记语言（XML）这样的纯文本格式中去，也可以把对象保存到 Oracle 和 MS-SQL 这样的关系数据库系统中去（如图 2-6 所示）。

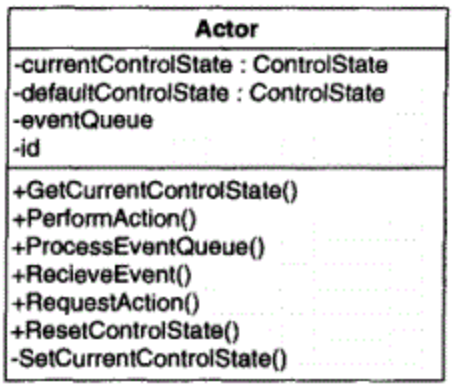


图 2-6　模拟对象类图

2. 执行者

执行者（Performer）是一个抽象核心类，它为行动者（Actor）和行动代理（ActorProxy）类提供了在客户和服务端共享的功能以及共同的接口。图 2-7 是执行者的类图。执行者类维护了一个调度优先级（schedulePriority）属性来实现对调度的仿真。同时还维护了对象的当前动作状态（currentActionState）属性，这个属性表示了这个对象当前所执行的动作，它由 PerformAction（执行动作）方法确定并进行设置。执行者类可以对这个对象进行扩展使之包含属于不同动作层次的多个并行的动作状态以实现更高级的仿真。这些动作层次可以包含移动状态、姿势状态（posture state）和对话状态等（本书第 2.7 节将会对并行状态机进行详细讨论）。

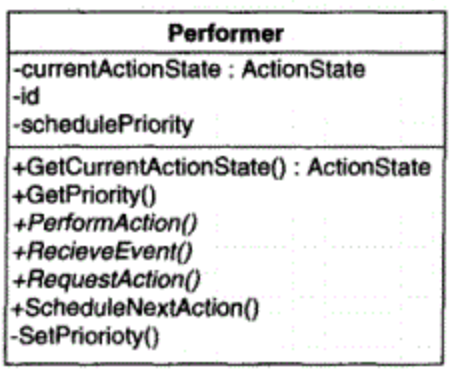


图 2-7　执行者的类图

3. 行动者

行动者（Actor）被定义为能够与仿真环境进行交互的服务端仿真对象。行动者具有一个控制状态（ControlState）属性，这个属性使得行动者可以被不同的控制方（agent）控制，包括玩家、AI 或是脚本驱动的过场动画。它把 PerformAction（执行动作）和 RequestAction（请求动作）方法委派（delegate）给当前的控制状态，而控制方必须提供相应的实现。行动者类还维护了一个由 ReceiveEvent（接收事件）方法产生的事件队列（eventQueue）。这样的事件队列使得行动者可以把事件批量地存储起来，并且延迟到下一次 PerformAction 方法被调度执

行时一并处理。这种被动而即时的事件处理方法使得系统不必在行动者每次收到消息时都要去判断应该做什么。在 MMP 这种具有大量事件的仿真系统中，每个行动者都需要知道它周围所发生的每件事情，因此，相对于在每个事件产生的时候立即进行处理的方式来说，采用延迟处理事件的方式是一种很大的改善。

ReceiveEvent（接收事件）方法还可以把需要立即处理的事件类型过滤出来，而不是放到事件队列中去，并在接收到这类事件时立即执行相应的处理。图 2-8 是行动者的类图。

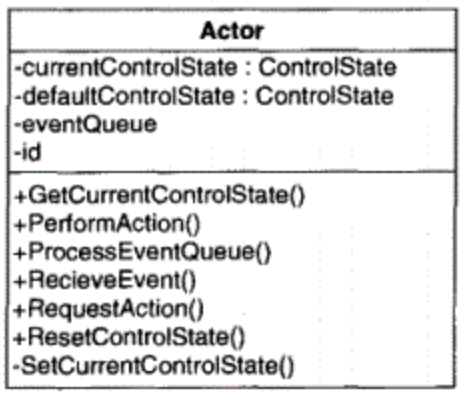


图 2-8 行动者的类图

4. 行动代理

行动代理（ActorProxy）是行动者在客户端的副本。它和对应的行动者对象共享相同的标识，并且会复制（replicate）相应的数据。客户端的仿真模块会把所有从外界接收到的事件发送给相应的行动代理对象，由它的 ReceiveEvent（接收事件）方法来处理这些事件。这个代理还提供了一个 RequestAction（请求动作）方法，它可以把外向（outbound）的动作请求发送给服务端与之对应的行动者。此外，行动代理还有一个 PerformAction（执行动作）方法，它用于处理任何不需要复制到服务端仿真层的、只需在客户端就可以进行处理的行为，譬如说一个动态的音轨选择或是用户界面（UI）元素的触发。图 2-9 是行动代理的类图。

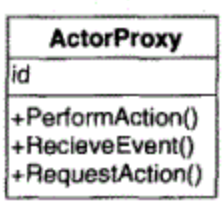


图 2-9 行动代理的类图

5. 非执行者类（Nonperformer）

那些不需要和仿真环境直接交互的核心类要比执行者类简单得多。这些对象不直接执行动作，仿真模块也不会为它们调度处理时间。它们必须在接收到事件时立即作出主动的处理。

- 物品（Item）：那些可以被行动者捡起、移动或是扔下的小型游戏对象。
- 障碍物（Obstacle）：游戏中那些既不能被捡起也不可移动的对象。

- 代理对象（ProxyObject）：物品和障碍物在客户端的副本。
- 区域（Area）：这是一个抽象的空间表示，它可以把仿真空间分隔成可管理的部分。区域定义了局部可见集（local potential visible set），这可以用来对仿真事件是否可以被看见（或被听见）进行过滤。

2.1.4　管理器类和工厂类

游戏中把管理器（Manager）和工厂（Factory）实现为单件（Singleton）[Gamma94]。在单个全局对象需要被多个不同的类和对象访问时，单件使用十分方便。单件会在初始化仿真层的时候创建，并且持续提供服务直到仿真层终止。这样客户端和服务端相对独立地维护了彼此相似的管理器。

1. 仿真对象工厂

仿真对象工厂（SobFactory）负责创建仿真对象并确保它们具有独一无二的标识。为此，游戏开发人员可以把仿真对象工厂作为“下一个仿真对象标识”（nextSobId）属性的惟一管理者。这个工厂提供了一个创建方法，它接受一个 SOB 类型参数以指定究竟要创建仿真对象的哪一个子类（行动者、区域、物品……）。如果客户端的仿真模块需要创建本地的仿真对象，它可以维护自己的仿真对象工厂，但是必须实现相应的机制以保证客户端 SOB 的标识不会同服务端所生成的发生冲突。图 2-10 是仿真对象工厂的类图。

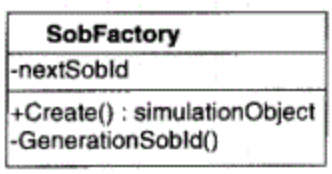


图 2-10　仿真对象工厂的类图

2. 仿真对象管理器

仿真对象管理器（SobManager）负责维护一个仿真对象的词典。这个管理器最主要的功能是通过 LoopUpById 方法把仿真对象的标识转换为对象的引用。它还提供了方法来保存（Save）和载入（Restore）其管理的所有对象。这两个方法把真正的对象持久化委派给特定的仿真对象进行。这样游戏开发人员就可以使用对仿真层而言透明的单一函数调用来保存或载入管理器内的所有对象。图 2-11 是仿真对象管理器的类图。

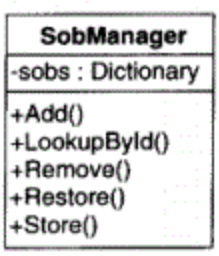


图 2-11　仿真对象管理器的类图

3. 调度管理器

调度管理器（ScheduleManager）负责为仿真中每一个活动的执行者的仿真对象进行调度以安排时间调用它们的 PerformAction 回调方法。它还提供了 ProcessTasks（处理任务）方法，这个方法以一个时间片为参数，并按照执行者对象的调度优先级属性对它们进行排序，然后调用在这个时间片内所能处理的所有已被调度但尚未执行的回调函数。图 2-12 是它所对应的类图。虽然有效的调度管理器可以和图中一样使用词典来实现，但是一个更理想的解决方法是使用优先队列。[Nelson96]是一篇关于优先队列实现的优秀文章。

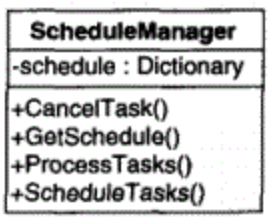


图 2-12 调度管理器的类图

4. 查询管理器

查询管理器（LookupManager）为在运行时访问静态游戏数据提供了一个快速高效的机制。这样的数据通常保存在关系数据库或是对象数据库中，并且在仿真层启动时载入。这些数据被映射到一个嵌套的词典中，这个词典把表（Table）作为主键，把条目（Entry）作为副键。这些静态游戏数据包括动作状态（ActionState）的初始数据、模拟事件（SimulationEvent）的类型以及模拟对象的属性等。（本书第 5.2 节详细讨论了一个查询管理器的实现。）图 2-13 是查询管理器的类图。

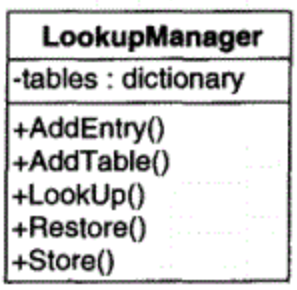


图 2-13 查询管理器的类图

2.1.5 仿真类

现在本文已经定义了支持类、核心类和管理类，还需要把它们包装成一个良好的顶层接口来管理客户端和服务端的仿真层。基本仿真（BaseSimulation）类就提供了这样一个接口。它不仅拥有对所有管理单件的引用，还拥有对根仿真对象的引用。这个对象代表了整个仿真系统，并且为所有顶层的区域仿真对象提供了一个可以挂接的点。仿真层维护了仿真对象管理器的两个实例：一个区域管理器（areaManager）和一个行动者管理器（actorManager）。对仿真对象的隔离有助于对其进行调试、维护和存储。图 2-14 就是仿真类图。

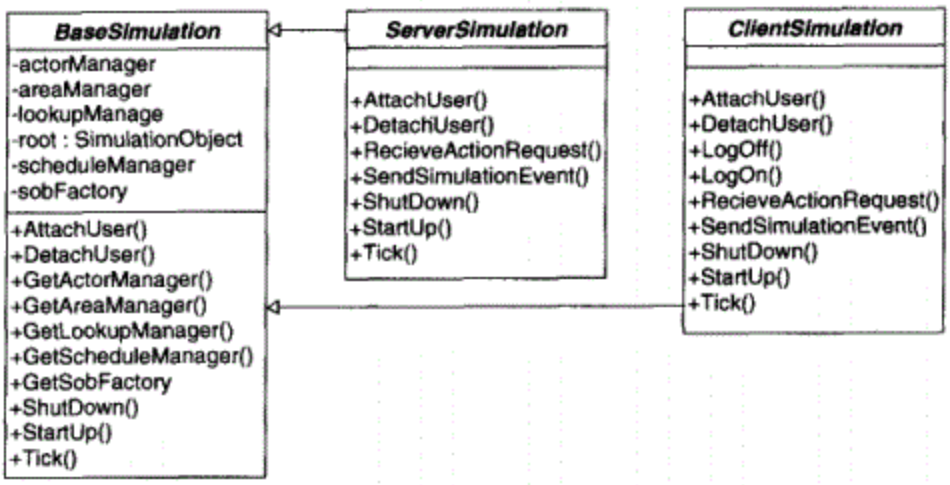


图 2-14　仿真类图

2.1.6　总结

这部分文章定义了创建一个 MMP 仿真框架所需要的类结构。第二部分（第 2.2 节）将介绍怎样在这个类框架的基础上把这些类扩展为一个可运行的在线仿真。

2.1.7　参考文献

[Alexander02] Alexander，Thor，“A Flexible Simulation Architecture for Massively Multiplayer Games,” *Game Programming Gems 3*，Charles River Media，2002.

[Alexander02a] Alexander，Thor，“GoCap: Game Observation Capture,” *AI Game Programming Wisdom*，Charles River Media，2002.

[Alexander02b] Alexander，Thor，“An Optimized Fuzzy Logic Architecture for Decision-Making,” *AI Programming Wisdom*，Charles River Media，2002.

[Boer00] Boer，James，“Object-Oriented Programming and Design Techniques,” *Game Programming Gems*，Charles River Media，2000.

[Booch98] Booch，Grady，*The Unified Modeling Language User Guide*，Addison-Wesley，1998.

[Dybsand00] Dybsand，Eric，“A Finite-State Machine Class,” *Game Programming Gems*，Charles River Media，2000.

[Gamma94] Gamma，et al.，*Design Patterns*，Addison-Wesley Longman，Inc.，1994.

[Nelson96] Nelson，Mark，“Priority Queues and the STL,” *Dr. Dobb's Journal*，*www.dogma.net/markn/articles/pq_stl/priority.htm*，January 1996.

2.2 为 MMP 游戏制作仿真框架，第二部分：行为建模

Thor Alexander， Hard Coded Games
thor@hardcodedgames.com

本文将以第 2.1 节中描述的类框架为基础，介绍怎样把这些类扩展为一个可以运行的在线仿真环境。

2.2.1 把用户和行动者关联起来

服务端仿真（ServerSimulation）和客户端仿真（ClientSimulation）都提供了对 AttachUser（连接用户）和 DetachUser（断开用户）方法的实现。这使调用这些方法的用户（User）可以被映射到服务端某个特定的行动者实例上以及与之对应的客户端行动代理对象上。设计者把用户与行动者连接起来后，就可以发送动作请求并且接收仿真事件了。目标行动者对象（target Actor）是接受还是拒绝这个连接请求完全取决于它当前所处的控制状态。这样的连接机制还有另一个好处：它可以支持一些高级的功能，譬如说可以让多个用户与同一个行动者对象连接。通过这个功能，客户支持人员不仅可以在某个玩家惹麻烦时接管他的角色，还可以在遇到麻烦的新玩家寻求帮助时，站在和玩家相同的视角来观察这个游戏。

2.2.2 动作请求

一旦与服务端的某个用户相连接，动作请求将成为客户端与外界通信的主要形式。客户端仿真类（ClientSimulation）的 SendActionRequest（发送动作请求）方法接受一个动作状态标识（ActionStateId）以及一个代表用户想要执行的动作的参数列表，并把它们传递给服务端的仿真层。服务端把这些请求依次委派给由所连接的行动者对象及其控制状态组成的一个职责链（chain of responsibility），由它来决定到底是处理还是拒绝这个请求。图 2-15 是用来阐述动作请求处理过程的一个 UML 顺序图。

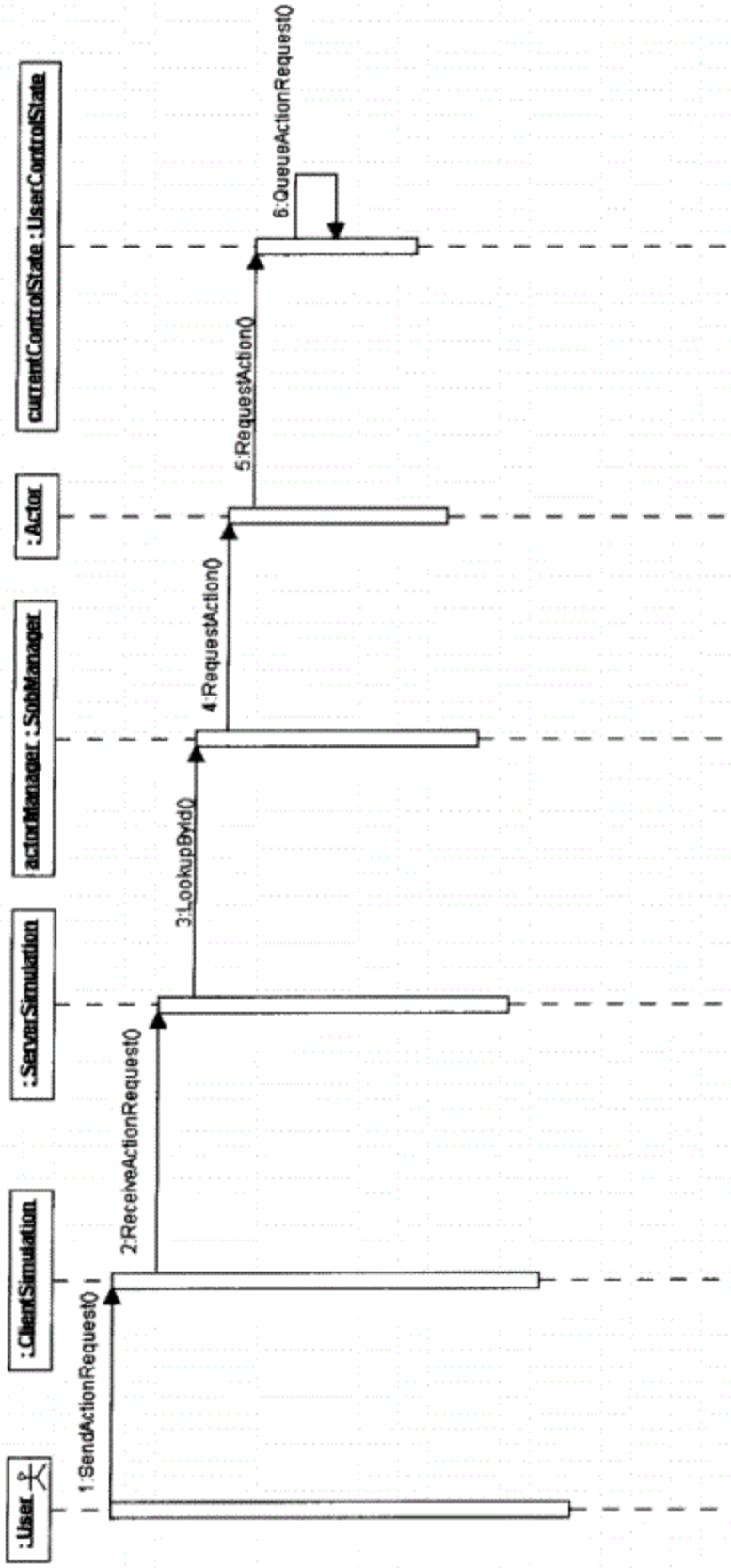


图 2-15　动作请求处理过程的 UML 顺序图

2.2.3 动作调度

服务端仿真类提供了一个 Tick 方法，游戏开发人员可以从仿真层的外部调用这个方法来对所有还未执行的动作进行处理。这个方法负责计算仿真处理可用的时间片，并且把这个时间片通过 ProcessTasks 方法传送给调度管理器。因为通常服务端物理层的处理频率必须比仿真层更高，所以 Tick 方法为维护游戏主循环的物理层提供了一个很好的回调方法，图 2-16 是动作调度的顺序图。

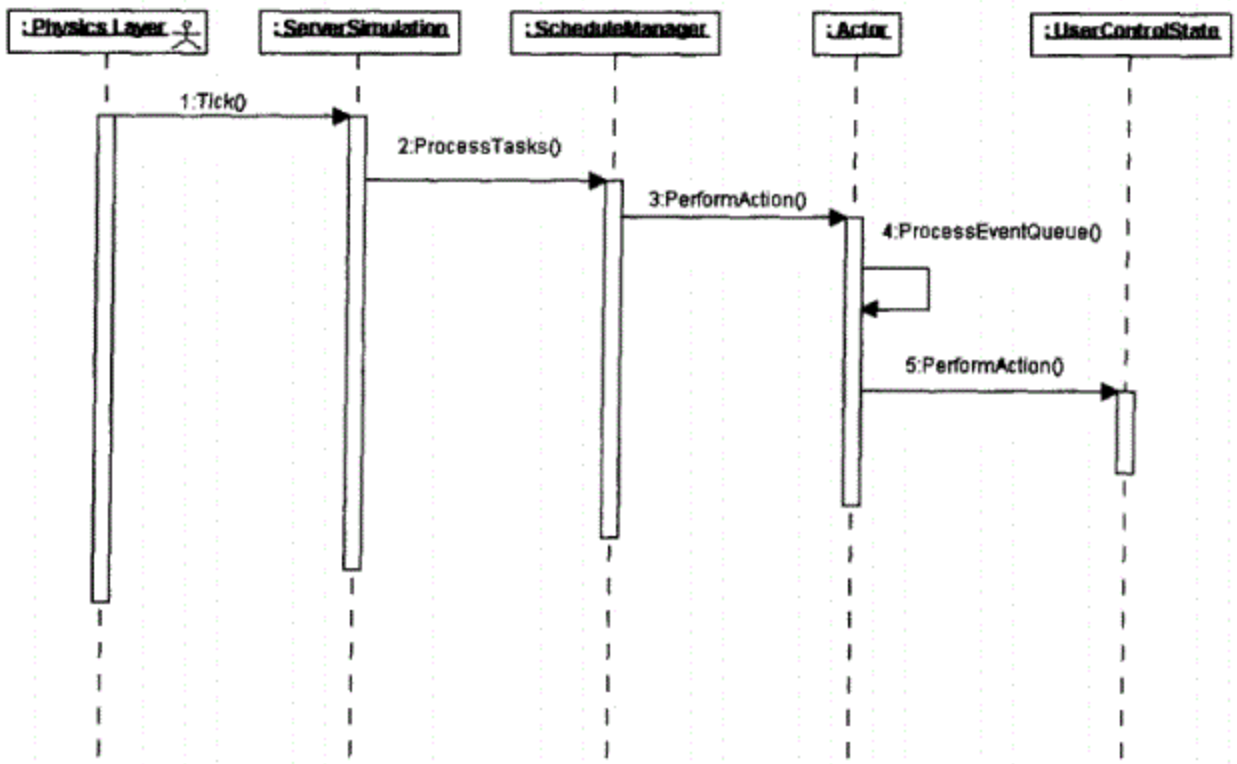


图 2-16 动作调度的顺序图

2.2.4 事件广播和处理

当行动者试图进行转换时，它会调用控制状态对象（ControlState）的 PerformAction 方法，这个方法会返回一个目标动作状态（desiredActionState）。随后行动者调用这个目标动作状态的 CanTransition 方法来进行预检，如果通过了，就可以调用目标动作状态的 Transition 方法来进行真正的转换操作。每个动作状态都需要提供其各自的特定实现。通常这个实现需要把状态转换通知给其他仿真对象，这可以通过使用仿真事件来完成。活动的仿真对象还维护了一个订阅者（subscribers）词典，对其动作感兴趣的其他对象可以进行登记。仿真对象会调用这些订阅者的 ReceiveEvent 方法来通知他们，并由消息的接收者决定是立即还是消极处理这个消息。在前一种情况下，消息会在接收后立刻得到处理；后一种情况下，消息会被放入队列中并且在接受者的 PerformAction 方法下一次被 Tick 调用时进行处理。图 2-17 显示了

相应的顺序图。

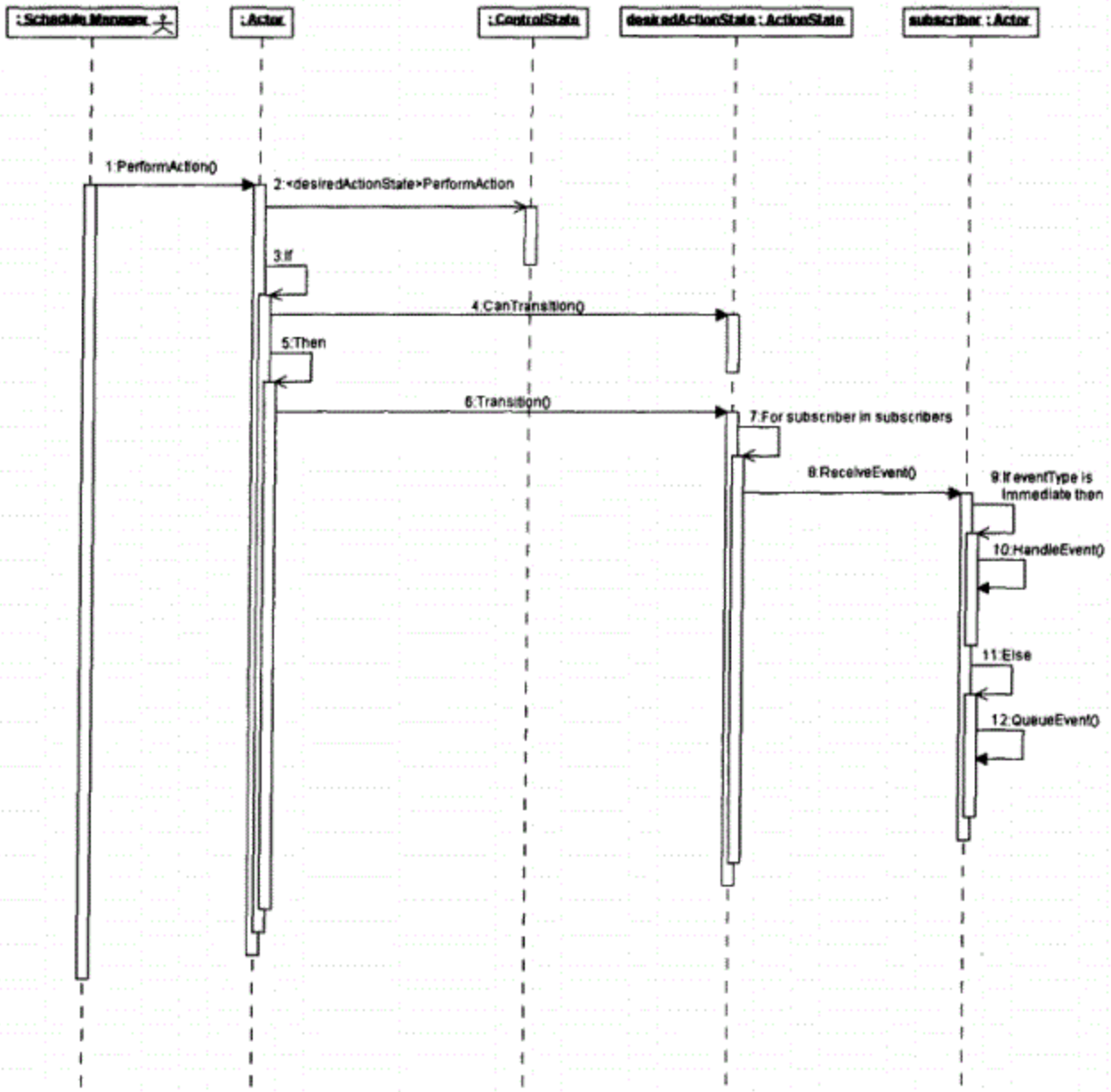


图 2-17　服务端行动者把动作改变的结果作为事件进行传播

2.2.5　服务端事件处理

当事件订阅者接收到一个事件时，它会把这个事件传递给这个类型的事件所对应的事件处理程序。这个事件处理程序会负责进行所有的服务端处理。另外，它还需要检查这个行动者对象是否有一个相连接的用户。如果有的话，它必须从服务端仿真中把事件发送给客户端仿真。这要求把这个事件对象分解并且放入一个可以通过网络传递给客户端的事件消息。图 2-18 展示了这个过程。

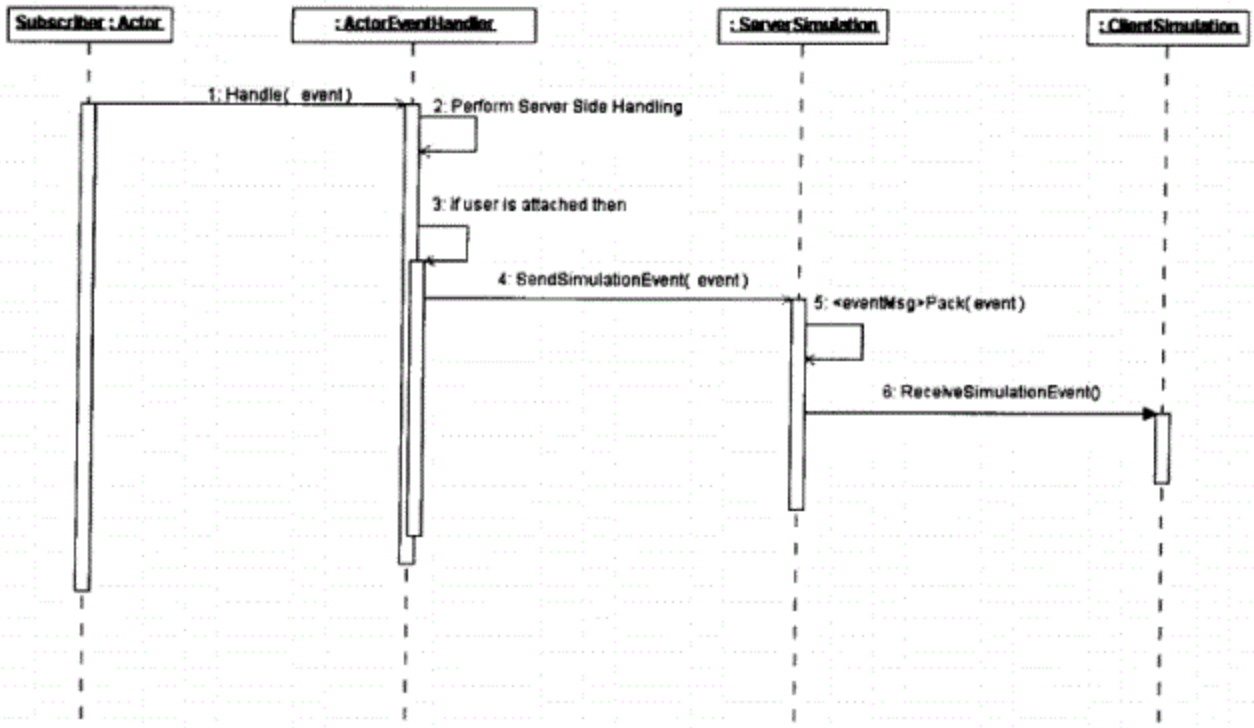


图 2-18 服务端行动者向客户端的用户发送消息

2.2.6 客户端事件处理

一旦客户端接收到事件消息，就会把它解开并且恢复为一个事件对象；接着会把事件的源行动者标识（sourceId）从事件对象中取出，并把它传递给客户端仿真的行动者管理器（actorManager），以查找相应的行动代理（ActorProxy）对象。这个代理对象是服务端对应的行动者对象在客户端的代表。如果行动者管理器没有找到这个代理对象，就意味着这是客户端第一次和这个行动者打交道，或是这个代理对象已经从代理对象缓存（这里面保存了客户端所关心的有限数量的代理对象）中清除了。在这种情况下，客户端需要使用它的仿真对象工厂（SobFactory）来创建一个新的行动代理，并且把它加入到行动者管理器所维护的缓存中。现在已经有了一个代理对象，可以把服务端传来的消息传递给它，处理过程如图 2-19 所示。

2.2.7 客户端代理

当客户端代理对象接收到一个事件时，“它”处理该事件的方式与之前在服务端处理该事件的方式非常相似。行动代理对象为事件类型确定合适的处理程序，并且把事件传给它。这和服务端找到的那个事件处理程序不同，它是一个负责进行客户端处理的代理事件处理（ProxyEventHandler）对象。为了复制服务端相应对象的状态，通常这些处理需要把行动代理对象转换到相应的代理行动状态（ProxyActionState），处理过程如图 2-20 所示。

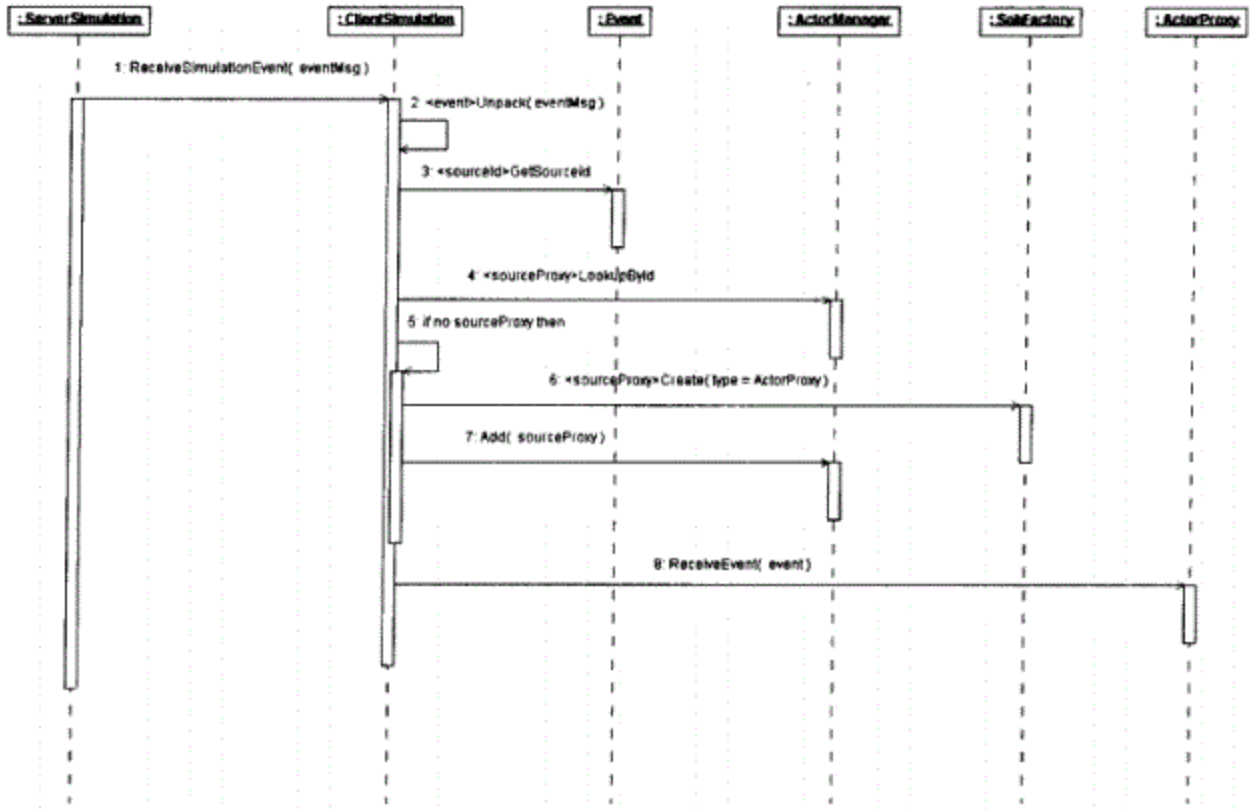


图 2-19　客户端仿真把事件发送给行动代理

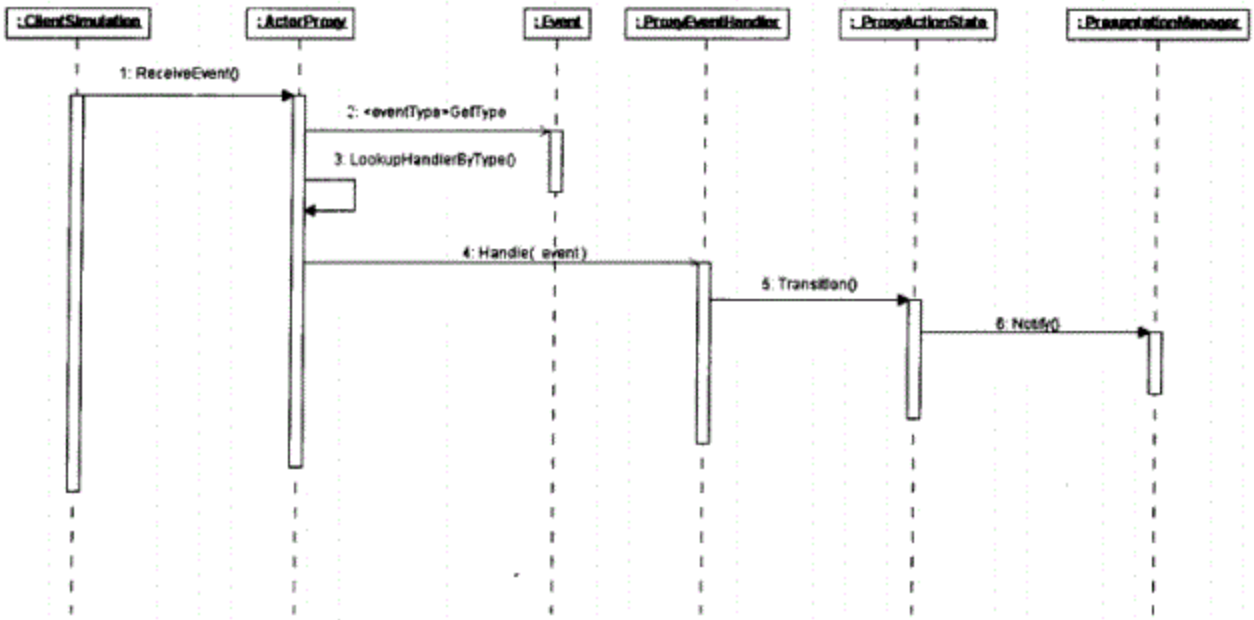


图 2-20　事件被客户端行动代理所处理

2.2.8　仿真与表示分离

通常，这样的状态转换需要在客户端有一个可视化的表示，为此仿真框架提供了一个钩

子，以通知任何对状态改变或其他相关事件感兴趣的表示管理器（presentation manager）。这些管理器可以是用户界面要素、音频播放器或是一个二维/三维渲染引擎。如图 2-21 所示，这个通知系统是基于观察者（Observer）设计模式的。游戏的代理动作状态（ProxyActionState）类有一个 Attach（连接）方法，任何对这个状态感兴趣的观察者都能通过调用这个方法进行注册；它还包含了一个 Notify（通知）方法，在状态发生改变时，这个方法会调用所有已注册观察者的 Update（更新）方法来通知它们。本书第 2.8 节会对观察者设计模式进行更详细的讨论。

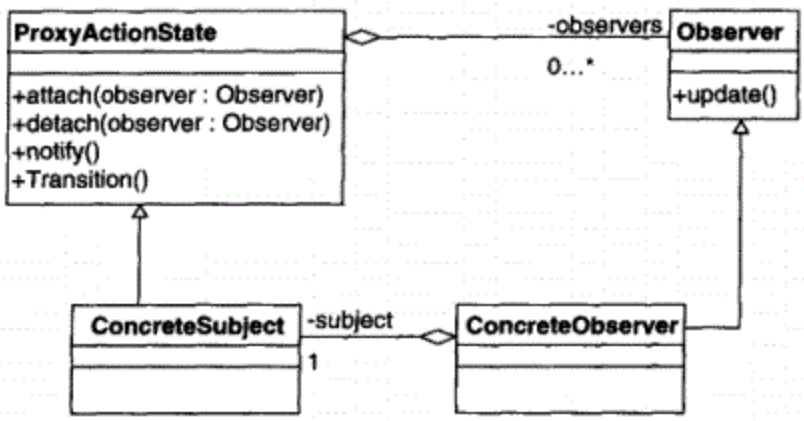


图 2-21 观察者模式使得仿真和表示相分离

这样的系统使游戏可以在仿真层和任意的表示层之间维护一个强有力的分隔。通过将表示和仿真分离开来，游戏可以方便地移植到不同的平台，而不需要重新编写客户/服务器代码。随着图像技术不断地向前发展，还可以更换不同的渲染引擎来使 MMP 游戏外观在市场上保持领先。这种分离的体系结构还可以实现在新兴的平台上开展游戏服务，譬如在游戏机和无线网络上运行 MMP 游戏。

2.2.9 总结

MMP 游戏开发所要面临的挑战常常被低估了。它通常由很多大型的部分组成。从一个类似于这篇文章中所介绍的基于可靠软件工程原则的框架开始开发将是一个良好的开端，它可以使游戏开发人员把时间和精力放在创建具有新意的、引人入胜的游戏模式上。

2.3　为游戏脚本创建一个“安全沙盘”

Matthew Walker，NCSoft Corporation
mwalker@softhome.net

本文的源代码包含在所附的光盘中。

游戏的开发需要一个稳健的服务器架构来满足 MMP 游戏在技术和支持方面的需求。然而，根据一般规律，持续地开发动态内容会加大引入错误的风险，这些错误可能会使入侵者有机可乘，破坏游戏的平衡。降低这种风险的方法之一就是建立一个“安全沙盘”（safe sandbox），这样游戏设计人员就可以编写代码实现他们的创造性想法，而不必担心会触及那些敏感的子系统。本文探讨了怎样使用高级程序设计语言 Python 来创建一个“安全”沙盘以实现一个封装的消息驱动环境，从而使设计人员“可以”把精力集中在游戏模式的开发上。

2.3.1　脚本语言与 MMP 开发

像 Python、JavaScript、Perl 和 UnrealScript[Sweeney98]这样的脚本语言都可运行于虚拟机中。本质上来说，虚拟机是一个 CPU 模拟程序。它可以用来执行那些与平台无关的字节码（byte code）表示的程序，而不是那些由 C 或 C++之类的语言生成的针对特定处理器的机器码[Kohlbrenner99]。在程序设计语言中使用这样的架构可以隐藏很多与内存管理、资源管理以及其他与操作系统功能相关的复杂细节。隔离这些细节还可以把由编程错误引起的问题限制在一定范围内。

脚本语言使用弱数据类型，这意味着变量可以在不同时刻包含不同类型的数据；脚本语言使用自动类型转换，这使得某种类型的数据可以方便地转换为另一种类型，譬如说从数字转换为字符串。这给开发函数/方法接口以及数据结构带来很大的灵活性。最后，这些语言通常还提供了像字符串、线形表、词典之类的高级对象，这可以使基于它们的开发更为有效[Ousterhour98]。

MMP 游戏是基于服务的产品，它不仅需要持续地保持可靠性，还需要经常更新动态内容。这意味着在 MMP 游戏项目的整个生命期内，游戏开发人员不仅要对其进行持续的改进和修正，还要保持一份可维护的稳定代码。这两个目标常常是对立的，因为通常对软件的频繁修改意味着会稳定性的丧失。脚本语言的特性使它们在那些需要同时满足这两个目标的应用中非常有价值。

使用 Python 还是实现自己的脚本语言

开发小组度身定制他们自己的脚本语言在游戏行业已经成为一种传统。UnrealScript 和 Quake C 是两个比较知名的例子。然而，设计、实现、调试并扩展一个程序设计语言是否能够更有效地使用开发小组的时间呢？从一个定制的语言中，游戏开发人员就可以精确地获得他们所需要的特性。但是有几个缺点抵消了这一优势：

- 用定制语言开发的游戏总是落后于对这个语言本身的开发，这无疑会给开发进度带来风险；
- 整个项目中，至少必须有一个全职程序员专职对这个语言进行开发和维护；
- 定制的专用语言会对它的用途作一些假设，最终这些假设可能会被证实为不够灵活因而不能满足项目发展的需要；
- 定制的通用语言需要精心的设计并且很可能会超出项目的需求范围，从长期来看，它可能会由于规模过大、涉及面过广而变得难以维护；
- 任何定制语言对于大多数程序员来说都是陌生的，并且它不像主流语言那样具有很多培训资源，譬如说参考书和网站，这会限制熟练的程序员在项目中的产出。

因此程序员应该使用那些现存的、已经使用了一段时间并且被很多应用验证过的通用脚本语言。由于下述原因[Python01]，本文推荐使用 Python。

- 它的语法非常简单易学。
- 大部分使用 C/C++来实现，这使得其执行速度相对来说比较快。
- 可以很好地和 C、C++及其他编译语言整合。
- 它是面向对象的，不仅支持类、单一和多重继承，还具有定义良好的作用域和名字空间，这有助于养成良好的设计习惯。
- 用它来完成一个给定功能所需要的代码行数比其他大多数语言少。
- 它有一套丰富的内建库，提供了包括文字处理、数据库交互、进程控制、文件输入输出、网络、加密等功能。

2.3.2 使用沙盘的理由

在对任何一个生命期较长的软件项目进行维护和改进的过程中，开发人员常常会因为贪图方便或是某些不成熟的“创造”行为而为系统引入风险。就代码完整性而言，最大的危险往往存在于那些需要经常修改的代码中。在整个游戏服务的生命期内，这些代码往往和实现游戏规则以及定义游戏中有趣部分的上层系统有关。开发人员需要对游戏进行频繁的修改（进行游戏平衡、加入特殊事件、加入新的物品和技能等）来保持游戏的趣味性。

与之相反，关键的底层系统（例如：网络代码、数据库接口、物理、移动、骇客检测等）应该很少需要修改并且每次修改都必须非常小心。避免在更新游戏系统时意外地影响到这些关键系统尤为重要。上述命题的逆命题几乎同样重要：仅仅因为害怕引入这些不稳定因素而不对游戏系统进行修改也是不应该的。

1. 创造性天赋并不等同于技术才能

MMP 项目常常寄希望于那些技术经验有限的游戏设计人员通过编码来实现游戏模式中的规则和关卡。虽然作为具有高度创造力的、非常有价值的设计人员，他们知道怎样使游戏变得更有趣，但是他们通常并不知道应该怎样设计游戏才能使游戏更稳健并且更容易维护。指望设计人员意识到诸如最小化依赖、定义一致的接口、抽象等概念的重要性是不现实的。

2. 一个用于实验的安全空间

下面开始讨论安全沙盘的概念。这是一个“人造”术语，它表示了一个受保护的空间。在这个空间里，游戏设计人员和内容开发人员可以任意地进行修改，而不必担心他们的工作会破坏这个系统。这个安全沙盘的目的就是要把用于实验的代码同关键的底层部分完全隔离开，这样使得只有在需要的时候才能通过良好定义的接口对特定的子系统进行访问。下面是在游戏编码和内容维护的实践中应该避免的一些做法：

- 在正常的框架以外任意地创建和销毁游戏对象；
- 直接控制游戏中对象的移动、位置和方向；
- 为了适合特殊的关卡而对 AI 或寻路（path-finding）算法进行有条件的调整；
- 绕开游戏数据库并且把游戏数据写到本地文件系统中；
- 直接打开一个连接到其他服务端/客户端的套接字（socket），而不是使用现有的消息系统；
- 直接使用嵌入式结构化查询语言（Embeded SQL）来查询或更新数据库，而不是通过现有的数据更新和读取接口；
- 向游戏客户端发送任意的文本消息；
- 对某个现存子系统的使用与其设计目的不一致。

通常，最初的游戏开发过程不应该允许这些行为的发生。不过这并不会限制后续开发人员这样使用系统，他们可能是想通过一些捷径来获得一个可以立即运行的服务。

2.3.3　安全沙盘的设计

在最初设计中就把安全沙盘作为整个架构的一部分有助于避免很多在开发后期会遇到的问题。

1. Python 的受限运行（Restricted Execution）特性

安全沙盘使用了一项 Python 独有的特性：受限运行。它由 rexec 和 Bastion 模块[Python02]组成。rexec 模块可以在游戏服务端的上下文中创建一个独立的具有自己名字空间的 Python 运行环境。游戏开发人员可以使用 rexec 来限制在沙盘中可以使用哪些 Python 提供的服务，也可以控制底层系统中有哪些在沙盘中是可用的。通过使用 Bastion 模块，游戏开发人员可以用一个保护性的代理（protective proxy）来包装那些他们想提供给沙盘的类，使得沙盘只能访问其正常接口的一个子集。

2. 安全沙盘的关键元素

每个游戏服务都会定义自己的架构规则和需求。因此本文只能为这个概念提供一个纯理论化的典型实现。安全沙盘具有下面这些主要设计元素。

1. SandboxRExec 类

这个类从 Python 的 rexec 模块中的 RExec 类继承而来，它定制了缺省的受限执行环境以满足游戏的需要。

Python 具有一个内建的 sys.path 属性，它使得解释器可以在收到载入模块的请求时找到模块。SandboxRExec 模块定义了_GetSandboxPath 函数，它会返回沙盘中所支持的模块搜索路径。这个路径用来替代内建的 sys.path 属性。

```
def _GetSandboxPath():
    return ['/usr/lib/python2.2',
        '/usr/lib/python2.2/plat-linux-i386',
        '/usr/lib/python2.2/site-packages',
        '/usr/lib/python2.2/site-packages/Numeric',
        '/usr/local/lib/gameserver/']    # our server's area
```

RExec 基类维护了一些元组（tuple）属性，用来表示允许访问或拒绝访问的模块。元组是 Python 的一种数据结构，本质上它是一个不变的线性表。在 SandboxRExec 类中，程序员简单地使用自己的模块列表来覆盖（override）这些元组。以_names 为后缀的属性表示相应的关键字和函数能够使用（用 ok_为前缀）还是不能使用（用 nok_为前缀）。那些具有_module 后缀的属性表示整个模块都受到这个环境的控制。最后，用_GetSandboxPath()函数来初始化 ok_path 属性。

```
class SandboxRExec(rexec.RExec):
    nok_builtin_names = rexec.RExec.nok_builtin_names + \
        ('compile', 'delattr', 'execfile', 'globals',
        'input', 'locals', 'raw_input', 'vars')

    ok_builtin_modules = ('math', 'operator', 'time')
    ok_path = _GetSandboxPath()    # load the valid path
    ok_posix_names = ()    # no os module access
    ok_sys_names = ()    # no sys module access

    ok_library_modules = \
        ('types', 'operator', 'copy', 'string',
         'math', 'cmath', 'random', 'time')
```

程序员添加了他们自己的属性 ok_server_modules，它列出了允许访问的游戏服务端模块。另一个定制属性 ok_packages 包含了系统中那些从任意模块都可以引入（import）的包（package）。包是 Python 中把相关模块封装在一起的机制。

```
    ok_server_modules = \
```

```
        ('safesandbox', 'events', 'errorhandler', 'log')
    ok_packages = ('sandboxmodules',)
```

最后，把所有可用的模块一起放入可以被基类识别的 ok_modules 元组中。

```
    ok_modules = ok_builtin_modules + \
        ok_library_modules + ok_server_modules
```

为了让某些 Python 核心函数以某种特定的方式运行，基类 RExec 为这些 Python 核心函数提供了一系列钩子，包括 r_exec()、r_eval()、r_execfile()、r_import()、r_reload()和 r_unload()。这些钩子与 Python 中那些去掉了前缀 r_的函数相对应，它们将取代对应函数而被调用。游戏开发人员希望对文件系统的任何访问都被禁止，所以他们覆盖了 r_open()函数，这样一旦在受限环境中调用 open()函数就会抛出一个异常。

```
    def r_open(self, filename, mode=None, bufsize=None):
        raise IOError, 'No access to filesystem is allowed.'
```

为了让特定的模块能够在受限环境中运行，游戏开发人员创建 SandboxExec 的一个实例并且通过它的 r_import()方法来引入模块，而不是象通常那样使用 Python 的 import 关键字。这个方法通过检查本节前面定义的元组来决定那些能够被引入模块。

```
def r_import(self, modulename):
    okToImport = 0

    if modulename in self.ok_modules:
        okToImport = 1
    else:
        # special test to see if module is a
        # submodule of an allowed package
        for pkg in self.ok_packages:
            if len(modulename) >= len(pkg):
                if modulename[:len(pkg)] == pkg:
                    okToImport = 1

    if okToImport:
        # call base class implementation
        mod = rexec.RExec.r_import(self, modulename)

        # if module is in dotted-path notation,
        # must get the left-most name
        components = string.split(modulename, '.')
        for comp in components[1:]:
            mod = getattr(mod, comp)
        return mod
    raise ImportError('Restricted: %s' % (modulename,))
```

在受限环境中运行的代码只能引入和执行那些被 r_import 允许的模块和函数。

2．Bastion()和 BastionClass

Bastion()函数是一个工厂（factory），它可以创建 BastionClass 的实例，而 BastionClass 可以通过包装对象来避免未经授权的访问。这些功能由 Python 的 Bastion 模块直接提供，在使用时不需要对其进行任何修改。Bastion 化（Bastion-ized）的类的缺省行为是阻止对任何数据成员以及以下划线（_）为前缀的方法的访问。

使用这个缺省的实现，给定一个实例 x，就可以合法地进行调用。

```
loc = x.GetLocation()
```

然而，如果试图调用：

```
x._SetLocation(1.5, 3.7, 0.0)
```

就会得到一个 Python 异常：

```
AttributeError: _SetLocation
```

通过提供一个可选的过滤函数作为 Bastion()函数的第二个参数。就可以定制这个行为。这个函数必须接受一个包含了属性名称的字符串为参数，并且在允许访问这个属性的情况下返回 true，在不允许访问的情况下返回 false。这种方式可以对那些受保护的类应该怎样向受限环境导出方法作出规定。可以使用下面的形式来实现这样的过滤函数。

```
def LegalMethodFilter(name):
  if name[:5] == 'game_':
    return 1
  return 0
```

然后，当用这个过滤函数创建一个受保护对象时，可以这样调用 Bastion 函数。

```
ob = SomeObject()
safeob = Bastion.Bastion(ob, filter=LegalMethodFilter)
```

现在，游戏开发人员可以在受限环境里调用 safeob 中任何以字符串“game_”为前缀的方法，其他函数将被禁止调用。

如果需要明确指定哪些方法是可访问的，就可以覆盖缺省的 BastionClass 以显式地支持列出每一个导出方法，而不是像以上的例子中那样使用一个名字约定。程序员甚至可以编写自己的 Bastion 函数实现来支持对某些特定属性的直接访问同时禁止访问其他属性。读者可参考所附光盘中的 Python Bastion.py 模块（在 Python 发布的 lib 目录中）以获得更多的信息。

3．SafeSandbox 基类

SafeSandbox 是一个抽象基类，游戏脚本的作者可以由此派生出他们自己的实现类。这些派生类的作用域可以与游戏世界中的某个环境一致，也可以和需要遵循某些规则或是某些预期行为的上下文（context）保持一致。这些环境可以是人们进行商业交流或是雇佣的公共场所、战斗场所、魔法森林，也可是一个废弃的太空站。

SafeSandbox 基类管理环境的活动范围，并且在受限环境内部和服务端框架的其他部分之间提供了一个坚实的壁垒。它还把服务端框架提供的服务聚合起来，其表现类似于 Façade 设计模式[Gamma95]，但区别在于是从 Façade 派生而不是把功能委派给它。这些服务包括在沙盘中分发游戏事件、在游戏世界中访问对象以及其他的实用功能号。

这个类的标准构造函数接受一个对游戏世界（world）对象的引用，通过这个引用，玩家可以访问游戏世界中的游戏对象。它还会初始化 EventManager 类的一个实例，服务端框架通过这个类来分发游戏事件。

```
class SafeSandbox:
    def __init__(self, world):
        self.__world = world # access to the game world
        self.__eventManager = eventmanager.EventManager()
```

这个类还声明了_FrameworkInit()方法作为创建后的初始化钩子。如果派生类实现了一个 Init()方法，_FrameworkInit 会调用它。这个技术可以确保基类的初始化函数总是被框架所调用，并且不需要派生类调用基类中 Init()函数的实现，因为派生类的实现者很容易会忘记这一步。

```
    def _FrameworkInit(self):
        # do post-creation SandBox init stuff here
        # ...
        # let derived class init if it likes
        if hasattr(self, 'Init'):
            self.Init()
```

游戏开发人员希望在 SafeSandbox 内部只能处理游戏事件而不能发出事件。把一个 EventManager 实例作为属性保存在 SafeSandbox 内部，并且覆盖内建方法_getattr_()就可以控制对它的访问。在此，下面的例子通过把 EventManager 所实现方法的引用赋给类中的同名属性来模拟对 RegisterHandler()和 UnregisterHandler()的实现。

```
    def __getattr__(self, name):
        # Posting of game events is not allowed within the
        # sandbox, so we only expose registration and
        # unregistration methods.
        if name == 'RegisterHandler':
            return self.__eventManager.RegisterHandler
        elif name == 'UnRegisterHandler':
            return self.__eventManager.UnRegisterHandler
        else:
```

```
            raise AttributeError(name)
```

某些游戏机制需要在 SafeSandbox 中访问游戏对象，譬如说游戏世界中的玩家角色和物品。然而，受限环境并不需要使用这些对象的所有特性。把它们包装在一个 BastionClass 代理中返回来，就可以对其进行保护。

```
    def GetGameObject(self, obId):
        # Return a protected game object from the world
        # region, based on its object id.
        gameobject = self.__world.GetGameObject(obId)
        return Bastion.Bastion(gameobject)
```

因为 SafeSandbox 基类引入了那些游戏开发人员希望在受限环境中不能使用的服务端模块，所以派生类不能像下面的代码那样直接引入 SafeSandbox 模块。

```
import safesandbox # exposes modules imported by safesandbox

class MySandbox(safesandbox.SafeSandbox):
    # implementation
    # ...
```

相反地，必须在通过受限环境引入派生类所在的模块之后，在运行时建立继承关系。这在 Python 中很简单，因为一个特定类的基类集合是由类对象（class object）的元组表示的，任何时间都可以对这个元组进行修改。在服务端启动时，_InitSandboxClasses()函数创建了一个由 SafeSandbox 派生类类对象组成的词典，CreateSafeSandbox 工厂函数会使用这个词典在运行时创建实例。

```
# a global instance of our restricted environment
g_re = sandboxrexec.SandboxRExec()

# a global dictionary of protected sandbox classes
g_classDict = {}

def _InitSandboxClasses(sandboxModules):
    for module in sandboxModules:
        # expand the row tuple into its elements
        moduleName, className = module

        # import the module via the restricted environment
        try:
            module = g_re.r_import(moduleName)
        except ImportError:
            print 'Cannot import %s.' % (moduleName,)
            continue
```

```
            # get the class object from the sandbox module
            safeSandboxClass = getattr(module, className)

            # disallow implementing __init__() in derived classes
            assert not hasattr(safeSandboxClass, '__init__'), \
               '%s must not define __init__()' % (safeSandboxClass,)

            # Add the SafeSandbox class into the list of bases,
            # creating inheritance without the derived class
            # needing to import this module.
            bases = safeSandboxClass.__bases__
            safeSandboxClass.__bases__ = bases + (SafeSandbox,)

            # store the sandbox class object indexed by its name
            g_classDict[className] = safeSandboxClass

        # end for
```

注意，派生类不可以实现自己的__init__()构造方法。这是因为游戏开发人员并不想为派生类是否实现了具有正确参数的构造方法或者是否调用了基类的实现而操心。正如前面所描述的，所有派生类的初始化由 init()方法完成。

最后，工厂函数在创建 SafeSandbox 的派生类时，会在_InitSandboxClasses()创建的词典中查找它的类名。在本节的例子中，工厂函数还调用了_FrameworkInit()钩子。

```
def CreateSafeSandbox(world, className):
    # Factory function that returns an instance of the derived
    # class of SafeSandbox, as indicated by the className. The
    # world parameter is an object containing information
    # about the physical game world and the objects it contains.
    try:
        s = g_classDict[className](world)
        s._FrameworkInit() # let derived classes init
        return s
    except KeyError:
        print 'No sandbox class for id [ %s ]' % (className,)
        return None
```

在一个或多个玩家进入这样的环境时，游戏服务端底层架构（infrastructure）会调用 CreateSandbox()工厂函数来创建一个相应的 SafeSandbox 派生类实例。每个派生类都依赖于它所处的环境，它所实现的方法必须按照这个区域的规则对环境中所产生的事件进行处理。可能某个公共场所是禁止战斗的，因此处理程序要能够检测攻击他人的企图并且给予其严厉的惩罚。例如在一个废弃的空间站中，一个发生泄漏的密封舱可能会爆炸也可能会被修复，这完全取决于玩家或其他游戏对象所产生的事件。

2.3.4 在安全沙盘中编写游戏代码

在受限环境中编写游戏代码很简单，程序员只需要为每一个沙盘编写一个 Python 类就可以了。这个框架不需要特殊的初始化，甚至不必从 SafeSandbox 基类直接继承。

以下是一个沙盘的例子，它实现的环境中包含了一个会对玩家造成伤害的陷阱。假设在地图的某个地方有一个触发装置，用来检测是否有玩家进入某个特定区域。当玩家进入时，警报器会被激活并且鸣叫 5 秒，玩家有 30 秒时间找到地图中某处的复位按钮。如果没有找到，就会有炸弹在他所处的位置爆炸。

首先，从游戏中引入需要的模块。这时，只能引入合法的模块。

```
# all our imported modules are legal
import sandboxmodules.alarm    # an alarm that alerts others
import sandboxmodules.bomb    # a bomb class (deadly!)
import events    # event ids to register for
```

接着，声明 SafeSandbox 的派生类 DangerousArea（危险区域）。然后，实现 init()方法来创建场景中的对象，并且注册由派生类实现的事件处理程序。这个方法是由_FrameworkInit()调用的。

```
class DangerousArea:
   def Init(self):
      # Do our own initialization and register event handlers
      # upon start-up here. Called by SafeSandbox.
      # alarm goes off every 5 seconds when triggered
      self.alarm = alarm.Alarm(interval=5)
      self.alarm.Arm()

      # bomb explodes with a radius of 10 meters
      self.bomb = bomb.Bomb(radius=30, damage=100)

      # register handlers with the SafeSandbox base class
      self.RegisterHandler(events.TRIGGER_ACTIVATED, \
         self.OnTrigger)
      self.RegisterHandler(events.ALARM_ALERT, \
         self.OnAlert)
      self.RegisterHandler(events.BUTTON_RESET, \
         self.OnDisarm)

   # maximum number of pulses the alarm can fire
   # before the bomb goes off (5 sec * 6 pulses = 30 sec)
   self.maxAlarmPulses = 6
```

大部分功能是在事件处理程序中完成的。发挥一下想象力，当玩家走过一个压感盘（plate）或是被一个传感器检测到的时候，会发出一个 TRIGGER_ACTIVATED（激活触发器）事件，

这时服务端框架会调用 OnTrigger()方法。

```
def OnTrigger(self, player):
    # Handle the trigger event. The activator is
    # the player who tripped the trigger.
    self.alarm.Alert(player) # announce who set off the alarm
```

当发出 ALARM_ALERT（警报）事件时，会调用 OnAlert()处理程序。

```
def OnAlert(self, player, pulsecount):
    # Called once for each pulse of the alarm.
    # The pulsecount is how many times the
    # alert pulse has occurred.
    if pulsecount < self.maxAlarmPulses:
        # boom... right at players location
        self.bomb.explode(player.GetLocation())
```

如果玩家找到警报器并且关闭了它，就会发出 BUTTON_RESET（复位按钮）事件，这会调用 OnDisarm()函数。

```
def OnDisarm(self):
    # We stop the alarms pulsing when
    # the reset button is pressed.
    self.alarm.Disarm() # whew!
```

1. 违反限制

游戏代码中没有对受限环境的直接引用。这是因为所有这些都由沙盘框架自动处理了。然而，如果可以按照下面的处理方法试图引入一个受限的模块。

```
import database    # let's query the DB!
```

当模块被引入时，可以看到这个异常。

```
ImportError: Restricted module: [ database ]
```

这是因为 database（数据库）模块没有包括在 SandboxRExec 类的 ok_server_module 列表所列出的允许引入的模块中。

同样，如果工作人员想通过下面的代码把被陷阱抓住的玩家的名字保存到一个文件中。

```
f = open('victims.dat', 'a')    # append to existing data
f.write(player.GetName())
f.close()
```

下面的异常会提醒工作人员有人已经违反了规则。

```
IOError: No access to filesystem is allowed.
```

最后，如果在这个例子中试图用下面的代码访问 player（玩家）对象的一个受限成员，譬如说直接减少他的生命值而不是用炸弹来造成这个伤害。

```
player.health = player.health - 100
```

游戏程序员就可以立即从响应中得知他们犯下的错误。

```
AttributeError: health
```

作为游戏脚本编写者，他们不必担心什么能做什么不能做，他们将在跨过边界时知道这些。

2. 当安全沙盘不安全的时候

本文所描写的技术并不能阻止不良程序员的恶意代码。沙盘中所有的代码是想帮助那些乐于为游戏作出贡献的人实现预期的行为，而不必受服务端框架的结构性细节的干扰。这个系统的目标用户是那些富有创造性的人，他们具有一定的程序或脚本编写技能，并且致力于为玩家创建有趣而引人入胜的游戏体验。

任何刻意地试图突破这个系统强加控制的人很可能找到另一个方法，只要他具有足够的时间、技能和动力。因此，本文不推荐在客户端代码中使用这种方法来防止黑客。

2.3.5 总结

一个 MMP 开发小组中最好的游戏设计员往往不是最好的程序员。开发小组雇佣他们是因为他们具有想象力、创造力和幽默感，而不是因为他们具有编写结构化代码或是设计关系数据库模型的能力。不过，在实现游戏规则和关卡的代码时，程序设计和游戏设计的规则会交织在一起。这时最好的选择就是由设计人员亲自实现自己的想法。

然而，在这种情况下也不能忘记软件开发是一项工程。MMP 开发团队往往希望可以按照某种特定的编程惯例来编写游戏代码。使用安全沙盘方法就可以通过创建一个环境来强制这些惯例的实施。这不仅可以让设计人员把工作重点放在设计上，还可以帮助程序员确保服务端代码的完整性，并且有助于减少因为使用没有预料到的方法而引起大量错误。

2.3.6 参考文献

[Gamma95] Gamma，Erich，*Design Patterns: Elements of Reusable Object-Oriented Software*，Addison-Wesley，January 1995.

[Kohlbrenner99] Kohlbrenner，Eric，et. al.，"Introduction to Virtual Machines," *http://cne.gmu.edu/itcore/virtualmachine/index.htm*，1999.

[Ousterhout98] Ousterhout，John K.，"Scripting: Higher Level Programming for the 21st Century," *http://home.pacbell.net/ouster/scripting.html.*

[Python01] van Rossum，Guido，"Comparing Python to Other Languages," *http://www.python.org/doc/essays/comparisons.html.*

[Python02] van Rossum，Guido，"rexec — Restricted execution framework," Python Library Reference，Fred L. Drake，Jr.，ed.，*http://www.python.org/doc/current/lib/ module-rexec.html*，April 2002.

[Python03] van Rossum，Guido，"sys — System-specific parameters and functions," Python Library Reference，Fred L. Drake，Jr.，ed.，*http://www.python.org/doc/ current/lib module-sys.html*，April 2002.

[Sweeney98] Sweeney，Tim，"UnrealScript Language Reference," *http://unreal. epicgames. com/ UnrealScript.htm*，December 1998.

2.4 大型多人游戏中的单元测试

Matthew Walker，NCSoft Corporation
mwalker@softhome.net

单元测试（Unit Testing）就是通过编写程序，在一致的条件集下使用另一个程序的功能，并且把获得的结果和期望的结果进行比较。单元是指一个具有内聚性（cohesive）的软件部分，它具有良好定义的接口并且对其他部分的依赖较少。单元测试由一个或多个测试用例（Test Case）组成，这些测试用例会以特定的方式使用单元的功能，并且对结果进行检查。把多个测试用例组合成一个测试集（Test Suite），这样游戏开发人员就能以批处理的方式运行它们。单元测试需要一个或多个特定的数据集来运行被测软件，这些数据集被称为测试数据集（Fixture）。在每次测试运行中使用相同的测试数据集可以保证输入的一致性，这样被测软件的任何行为变化都可以归结为软件的变化，而不是数据的变化。

单元测试已经在软件工程领域应用了很多年，最近更是随着极限编程（Extreme Programming，XP）的发展而流行起来。XP 是一种强调“简单（simplicity）、沟通（communication）、反馈（feedback）和勇气（courage）”的软件开发方法[XP99]。单元测试是 XP 的核心方法之一，它与持续代码整合（continuous code integration）、小型发布（small releases）、结对编程（pair programming）、集体拥有代码（collective code ownership）以及其他实用方法一起，致力于为迅速高效地开发高质量的软件提供服务[Jeffries01]。

2.4.1 为什么 MMP 游戏需要单元测试

通常单机游戏和那些小型在线游戏的绝大部分利润是在发布后的短时间内获得的。在这类游戏发布后，开发人员很少会对它们再作修改，除非是为了改正那些影响销量的严重错误。与此相反，大型多人游戏往往是一项会在数年时间里持续不断地创造收入的服务。这样的服务在它的整个生命期中需要大量的开发和支持人员。在 MMP 游戏的生命周期内，为了向订阅者提供新鲜有趣的游戏体验，游戏开发人员需要对代码和游戏内容进行多次修改。在 MMP 中，一个软件错误就可以让在线服务崩溃或是给游戏平衡带来相当大的麻烦。这些问题威胁着收入的稳定性以及项目的整体成功。对于 MMP 开发人员来说，单元测试具有两个显而易见的好处：

1．它可以在开发过程中保证代码的完整性，尤其是在整合过程中：

2. 在游戏发展过程中，它可以最小化在成品代码中引入错误的风险。

相对传统的计算机游戏来说，MMP 产品的长期性以及它们对服务提出的要求使游戏开发人员必须更加重视管理软件开发的风险。单元测试可以让错误更早、更频繁地暴露出来，从而使风险最小化。

2.4.2 单元测试的定义

要设计一个有效的单元测试，必须明确地定义3个主要概念。它们分别是单元（unit）、测试数据集（fixture）和测试本身。

1. 单元的定义

所要测试的软件单元必须具有良好定义的接口并且对其他单元的依赖较少。在把所有的依赖条件抽象到单元接口之后，进行测试的代码就没有必要知道它们了。通常，从面向对象的角度来看，一个单元就是一个类。

设计良好的软件通常由互相关联的对象集合和层次组成。定义单元最有效的方法是从最内部、最底层或是最基本的类开始，然后向系统上方进行直到那些高层类被定义。这种方法和软件发展的正常顺序相一致，这可以确保在对那些复杂的代码进行测试之前，它们所依赖的代码已经可以正常工作了。

常见的底层单元有用于创建网络包的数据缓冲（DataBuffer）类、用于并发处理的线程（Thread）类和用于加密的加密（Crypt）类等。高层类则包括那些管理游戏中运行对象的对象管理器（ObjectManager）类、用于分发异步消息的事件管理器（EventManager）类以及用于检测对象是否相交的碰撞管理器（CollisionManager）类等。

2. 数据集的定义

每一个单元测试都包含一个或多个数据集，用来为将要测试的单元建立数据集合。一个数据集可以简单到仅仅是一个用来测试数据缓冲类的硬编码字符串，也可以由一组提供给碰撞管理器的具有碰撞几何信息的对象组成。如下就是这两类的数据集：

```
// test string for the DataBuffer
const char* data = "the quick brown fox";

// collection of Collider objects
vector<Collider*> colliders;
colliders.push_back(new Collider(0.0, 1.0, 0.0));
colliders.push_back(new Collider(1.0, 0.0, 0.0));
colliders.push_back(new Collider(0.0, 0.0, 0.0));
```

针对同一个单元的不同测试可能需要不同的数据集。一个对象管理器的正向测试（被设计为应该获得正确结果的测试）可能需要一个对象集合，里面的对象都具有独一无二的标识。而它的逆向测试（被设计为会引起错误的测试）也许需要其测试对象集合中包含指向同一个

对象的重复引用以强制错误情况的产生。我们应该定义足够多的数据集以彻底地测试这个单元的各个方面。

通常，数据集可以直接硬编码在单元测试内。这当然是最方便的方法，因为测试人员对于提供给单元的数据具有完全的控制。然而，在测试高层单元时往往会需要具有外部数据的数据集。外部数据是指从文件、数据库、网络套接字或是这个单元测试代码以外的其他资源中读取的数据。之所以使用这种类型的数据集，是为了使被测单元在测试中能够以和游戏实际运行时相同的方式接收数据。

在测试数据集中使用外部数据的难点在于，必须确保这些数据在测试时总是存在，并且在多次测试之后不会改变。这就需要小心地管理测试环境并且把产品数据和测试数据分离开。通常被测单元的开发目标不同于项目目标，这会导致一些不可预测的变化。在测试数据集中使用产品数据会使单元测试受到这些变化的影响。被测单元必须“拥有”它的测试数据以保证其完整和一致。

3. 测试的定义

被测单元所有的主要功能都必须使用单元测试。这就意味着要调用这个单元公共接口中声明的方法或函数。游戏开发人员应该对所有衍生数据（根据单元状态计算出的数据）或是改变单元状态的方法进行测试。通常没有必要去测试那些简单地获取或者改变数据的方法，除非它们会返回衍生数据或者导致单元的状态发生改变。

通常是无法对单元的保护或私有接口进行测试的。大多数语言（如 C++和 Java）所具有的强制访问保护使得测试代码不能访问私有或保护方法。然而这并不会造成什么问题，因为单元测试就是要模拟那些使用被测单元的代码，而这些代码也不能访问那些受限的接口。这里不推荐为了测试的方便去修改那些本来需要限制访问的代码接口。

在对一个单元进行测试时，游戏开发人员必须把调用函数或方法所获得的实际结果和预期结果进行比较。如果实际结果和预期结果一致，测试通过；否则，测试失败。当测试通过时，单元测试代码什么都不用做；当它失败时，单元测试代码应该抛出一个异常，测试框架中的错误报告代码会处理这个异常。预期结果会作为单元测试代码的一部分来进行预先定义，它完全依赖于所使用的数据集和被测代码。

游戏开发人员可以使用多种方法来比较实际结果和预期结果。对于计算所得的数值，我们把它和预期结果进行简单比较就可以了。

```
// test derived damage value
BattleDrone d;
d.SetWeaponType(PULSE_RIFLE);  // damage=energy*0.4
d.SetEnergyLevel(150);
d.AddDamageBonus(15);
int damage = d.GetAttackDamage();
assert(damage == 75); // (150 * .4) + 15
```

对单元内部的状态变化进行测试则更为复杂，最主要的问题在于有时无法访问被改变的状态。当这个状态能通过公有接口访问时这不成问题，正如下面所示。

```
// test string concatenation
MyString* s = new MyString("hello");
s->Append(" world");
assert(s→GetContents() == "hello world");
```

然而，内部状态的改变有时候是通过单元行为的改变表现出来的。在这种情况下，需要编写代码以检测这样的行为改变。下面是一个例子。

```
// test collision manager
Collider* pC1, pC2, pC3;
pC1 = new Collider(0.0, 0.0, 0.0); // start at origin
pC2 = new Collider(1.0, 0.0, 0.0); // 1 away from c1
pC3 = new Collider(-1.0, 0.0, 0.0); // 1 away from c1

CollisionManager manager;
manager.AddCollider(pC1);
manager.AddCollider(pC2);
manager.AddCollider(pC3);

// first, ensure not colliding
bool bCollided = false;
bCollided = manager.TestCollision(pC1, pC2);
assert(!bCollided);

bCollided = manager.TestCollision(pC2, pC3);
assert(!bCollided);

bCollided = manager.TestCollision(pC1, pC3);
assert(!bCollided);

// move pC2 closer to pC3
pC2.SetPosition(-1.0, 0.0, 0.0);

// test collisions again
bCollided = manager.TestCollision(pC1, pC2);
assert(!bCollided); // still not colliding

bCollided = manager.TestCollision(pC2, pC3);
assert(bCollided);  // this pair is now colliding

bCollided = manager.TestCollision(pC1, pC3);
assert(!bCollided); // still not colliding
```

在上述例子中，改变一个碰撞（Collider）对象的位置会使它和另一个对象相交，这将导致碰撞管理器（CollisionManager）检测到一个本来不存在的碰撞。

2.4.3 单元测试框架

通过编写程序来使用其他代码的功能并不困难。单元测试通常不过是一个批处理：初始化数据、调用组成测试用例的方法或函数、检测错误。为了便于使用，这些基于批处理的程序通常必须提供一些工具来为每个测试用例重新初始化数据集、把测试用例彼此隔离开、处理预料之中和预料之外的错误、搜集统计信息并且报告结果。

游戏开发人员可以自己编写这些工具，但这没有必要。这些功能虽然是必须的，但其中的大多数都既简单又重复，毫无乐趣可言。目前就有很多设计良好的单元测试框架可以为我们所用。大多数这类框架的实现思想都基于 Kent Beck 在 1994 年首次提出的 Smalltalk 测试框架[Beck94]。Beck 把这个框架命名为“Sunit”，这很快就成为实现单元测试常见功能的标准方法。

1. xUnit

人们使用很多语言重写了 Beck 的框架，包括 Java（JUnit）、Python（PyUnit）、C++（CppUnit）和一些其他语言。这些框架被统称为 xUnit 框架[xUnit02]。它们所提供的工具可以使编写单元测试变得更为简单并且高效。这些框架的实现细节各不相同，但是都实现了和下面类似的接口。

测试用例（TestCase）——这个类是实际测试代码的宿主。单元测试的编写者通常为每个单元测试编写一个测试用例的派生类。每个测试用例的派生类都会实现一个或多个方法来表示一个具体的测试用例。它还实现了一个 setUp()方法来初始化数据集以及一个 tearDown()方法来做必要的清除工作。这个类的每个实例只会执行它所拥有的测试方法中的一个，这个方法由构造函数的参数指定。需要测试的方法通过标准的 run()方法间接调用。run()方法会先调用 setUp()，接着调用在构造函数中指定的测试方法，最后调用 tearDown()。这可以保证每一个测试用例都能使用一个未经修改的测试数据集。

测试集（TestSuite）——这个类负责把多个测试用例聚合起来，从而把它们当作一个整体来运行。测试人员要为每一个需要运行的测试创建一个测试用例派生类的实例并且把它加入到测试集中去。当单元测试被执行时，测试集会枚举测试用例集合中的每一个实例并且调用它们的 run()方法。虽然可以从测试集派生以获得定制的功能，但是通常测试人员会直接使用这个类。

测试结果（TestResult）——这个类负责捕捉测试的运行结果并且记录失败（与预期不一致的测试结果）和错误（预料之外的异常）。测试结果还会为特定的失败和错误记录详细信息，从而帮助测试人员诊断所遇到的问题。

测试运行（TestRunner）——这个类管理所有测试集的批处理运行，它聚合了相应的测试结果，并且会把结果写到指定的输出设备。测试人员可以使用这个类的不同变种来生成不同格式的输出，如文本、XML 或是 HTML。

2. 使用 xUnit

Python 是一门简单的语言，即使对其一无所知的程序员也能够轻易读懂它。因此，本节

使用 Python 的测试框架 PyUnit[PyUnit02]来介绍 xUnit 的一些特性。本节试图说明那些本质上和语言无关的通用概念。这里所使用的例子可以很方便地转化到 CppUnit、JUnit 或者其他基于 xUnit 的框架。

3. 实例：对易耗属性（Consumable Attribute）进行测试

下面的例子介绍了对属性（Attribute）类的测试，这个类用于管理角色扮演游戏中的易耗属性，例如健康值和魔法值。这个类包含两个值：当前值（current）和最大值（max）。当前值可以增加或减少，但是永远不能超过最大值。通常，当前值和最大值相等。在当前值减少时，一个时钟会被启动，这个时钟每隔一段时间为当前值增加一定的点数，直到它和最大值相等。最大值也可以增加或减少。如果它减少到一个比当前值还要小的值，当前值也必须作相应的减少。如果最大值增加了，时钟会被启动来逐渐增加当前值。

这个测试是一个独立的 Python 模块，它的实现在源代码文件 attribute_t.py 中。这个模块不仅定义了测试类，还提供了初始化代码，这样就可以把这个测试整合到总体框架中去。

```
# attribute_t.py
# Unit test of the Attribute class.

from unittest import TestCase      # always need this
import time                        # needed for our test
from attribute import Attribute    # unit being tested

#
# Test class for testing Attributes
#
class AttrTest(TestCase):  # derive from TestCase
   # class-level test suite member
   suite = unittest.TestSuite()

   #
   # Fixture definition
   #
   def setUp(self):
      # init with 100 pts current and max value
      self.attr = Attribute(100,100)
      self.attr.SetRefresh(1,5) # 1 pt every 5 sec

   def tearDown(self):
      pass # nothing to do, cleanup is automatic

   #
   # Test methods
   #
   def testChangeCurrent(self):
      # Ensure changes are reflected correctly
      a = self.attr
```

```
        a.ChangeCurrent(-50)    # decrease 50 pts
        self.assert_(a.current == 50, "Decrease failed.")

        attr.ChangeCurrent(25) # increase halfway to full
        self.assert_(a.current == 75, "Increase failed.")


    def testChangeMax(self):
        # Ensure changes to max are reflected correctly
        a = self.attr

        a.ChangeMax(-50) # max reduction reduces current
        self.assert_(a.max == 50, "Max decrease failed.")
        self.assert_(a.current == 50, "Current > max.")

        a.ChangeMax(25) # regain max, current stays
        self.assert_(a.max == 75, "Max increase failed.")
        self.assert_(a.current == 50, "Current grew.")


    def testRegeneration(self):
        # Ensure depleted current value increases over time
        a = self.attr

        a.ChangeCurrent(-50)
        time.sleep(5)  # regen period of 5 seconds
        self.assert_(a.current == 51, "5 sec failed.")

    time.sleep(10) # should get 2 more pts
    self.assert_(a.current == 53, "10 sec failed.")

    a.ChangeCurrent(47) # take us back to max
    time.sleep(10) # should have no more refresh
    self.assert_(a.current == 100, "Excess refresh.")


#
# Module-level test suite initialization.
# PyUnit framework will call this function.
#
def suite():
    # Add reference to each test method to the
    # TestSuite, so the framework can call it.
    AttrTest.suite.AddTest(AttrTest.testChangeCurrent)
    AttrTest.suite.AddTest(AttrTest.testChangeMax)
    AttrTest.suite.AddTest(AttrTest.testRegeneration)

    # The returned suite will be combined with others
    # by the framework.
    return AttrTest.suite
```

模块一级的 suite()方法使用我们的测试类来初始化一个测试集对象并且将其返回。这个对象是 AttrTest 类的属性，而不是每个 AttrTest 实例的属性。这意味着所有的 AttrTest 实例只有一个测试集。这个测试集包含了一个列表，里面保存了对测试方法的引用。当测试运行时，测试框架会为列表中每一个测试方法创建一个 AttrTest 实例。每一个实例都会依次调用 setUp()、测试方法以及 tearDown()。

4. 尽可能使用简单的测试方法

注意，本节所用的测试方法都是非常简单的。每个方法只负责某个特定的部分。这样做有两个原因。第一个原因是测试方法越小就越便于阅读和理解。另一个原因是测试人员通过在测试用例的 assert_()方法中抛出异常来进行错误报告。（也存在其他错误报告方式，读者可查阅 PyUnit 的文档以获得进一步的信息[PyUnit02]。）它会停止执行当前正在进行的测试方法，因此这个方法中任何后续的代码都不会被执行。测试方法的数量越多并且规模越小，在测试失败时可以生成的信息也越多。

2.4.4　测试先行的设计

单元测试最有用的应用之一就是测试先行的设计（test-first design）。这个方法强调测试是软件设计的驱动力而不仅仅是一个副产品[Langr01]。

1. 在编写代码前编写测试代码

这个方法的关键在于程序员必须在编写任何被测代码前先编写单元测试。这种方法是一个易于采用并且可以快速生成结果的方法。如果测试在第一次运行时就没有任何错误，那么有两种可能：a)代码没有缺陷并且老板给的工资太低了；b)没有真正地使用到他们想要测试的代码。一般，b)是最有可能发生的情况。

测试人员必须确保单元测试确实执行了被测代码并且被测代码能够正确工作。要做到这点，最简单的方法就是为那些他们明知有问题的代码编写测试代码，并且修正那些代码使它们能够正确通过测试。测试人员能够获得有问题代码的最早时间就是在还没有编写代码前。

2. 编码的步骤

测试先行的设计按照有规律的步骤进行，如下。

（1）为一个还不存在的函数或者方法编写测试代码。
（2）必要的话，编译测试代码；编译将会失败。
（3）通过编写被测试的函数或方法的存根（stub）版本（译者注：存根版本是指仅包含必要的函数定义和返回值的版本，没有实际功能，仅用来通过编译。）来修正编译错误。
（4）运行测试；因为还未实现功能，测试将会失败。
（5）通过编写被测函数或方法的函数体来修正测试错误。
（6）运行测试；如果失败，修正代码；如果成功，编写另一个测试。
（7）重复上述步骤。

如果按照这个步骤进行开发，就能够以更快、更自然的方式让代码运行起来。更重要的是，在实现设计时游戏开发人员会非常自信，因为他们对每个主要功能都进行了彻底的测试，并且证明它们是可以正确运行的。

3. 让自己成为最初的受害者

测试先行的设计强调从使用者的角度编写代码。也就是说，程序员编写他们想要使用的代码，而不是编写他觉得别人可能会使用的代码。他们是功能的作者，同时也是设计决策的第一个受害者。

一个很常见的情况是在亲自使用自己所写的代码之前，他们并没有真正地理解它。有些程序员甚至从来没有使用过自己的代码，而是扔给一墙之隔的质量保证人员或是其他程序员去运行。测试先行的设计，特别是单元测试，通常可以让他们更快地理解自己的设计决策所带来的影响。

4. 单元测试和重构（refactor）

随着在代码中加入新功能或是改正错误，游戏开发人员会改变早期的设计决策以适应新的需求。在此，类、模块、方法、函数和数据结构将被分割、合并以及重写，从而使新的设计可以通过一个慎重的有机过程融入到现有的代码中去。

测试先行的设计非常适合在这个过程中使用。这个情况下所使用的模式如下。

（1）辨认出那些需要重构的代码。

（2）编写单元测试来证明现存的代码能够工作。

（3）开始重构，对代码进行微小的、慎重的修改。

（4）重新编译并且对每个修改运行单元测试；单元测试将失败。

（5）重写测试使得它符合新的代码。

（6）重新运行测试，如果代码能够工作，继续重构；如果测试失败，修正新代码。

（7）重复上述过程。

2.4.5 实用因素

把单元测试合并到开发过程中并不只是简单地编写测试。要实现总体的成功，还必须注意测试逻辑，并且应该开发一个可以用于各种不同环境的测试方法。这里，本节对下面这些因素进行讨论。

1. 测试过程自动化

单元测试除了可以证明特定的代码单元在最初开发期间能够正确工作，还可以知道对共享代码的改变是否破坏了依赖于它的其他代码。游戏开发人员最好通过建立一个自动化的测试过程来实现单元测试。这个过程通常包含如下几步。

1. 从代码库中取出最新的代码。
2. 对代码进行编译和链接。
3. 运行所有的单元测试。

4. 向编程团队报告结果。

这个过程的目标是让代码可以随时通过所有的测试。测试人员应该每天运行几次测试，从而迅速地识别那些会影响到多个团队成员的错误。当某个测试失败时，修复这个失败将是团队中优先级最高的任务。游戏开发人员可以采用这个过程来确保在代码库中的所有代码都能正确工作。

2. 独立的测试程序

要进行单元测试，最简单的方法就是使用一个独立的可执行程序以批处理的方式来调用所有的测试。程序员只需编写一个主模块来载入测试框架，然后确定需要运行哪些测试并且执行它们。为了增强灵活性，可以开发一个工具，每当代码中有新的测试加入，这个工具会自动把它们加到测试程序中。还可以加入生成报告的功能，譬如说把结果用电子邮件发出或是生成一个网页。

由于这个方法实现起来很方便，游戏开发人员应该尽可能地使用它。这个方法最适合用来测试那些对外部系统和资源依赖很小，并且对初始化要求不高的代码单元。这类代码单元包括类库和中间件工具、底层 API、输入输出程序、数据管理工具以及游戏中其他自包含的基本元素。

3. 集成的运行时测试程序

有时候，游戏测试人员想要测试的代码必须在特定的运行环境下才能执行。这往往出现在那些需要进行复杂初始化的系统中，譬如说需要从数据库载入数据并且通过网络和其他进程交互的游戏服务器。理论上说，可以把这个初始化看作单元测试数据集的一部分，每次运行测试时都可以调用这个运行时系统的一个新实例。不过，这种方法对资源和运行时间的要求使得它并不是很实用。

可以使用一个相反的模式：在运行环境中执行测试。这样，单元测试就可以对宿主进程提供的资源进行访问。使用这个方法时，游戏测试人员必须注意不要让对测试框架的调用破坏了产品代码。测试必须从宿主进程中某个位置开始，以批处理的方式执行。这个位置通常紧跟在所有必要的初始化完成之后。理想情况下，这个调用点应该只有一行代码，用来调用一个函数以启动测试过程。所有像测试识别、初始化、执行和报告之类的实际工作都应该在这个函数的代码中完成。这样就可以通过条件编译选项在开发人员编译发行版本时把这个函数调用去掉。

这种类型的测试环境最适合用于那些由很多小型系统组成的大型系统。这类代码单元包括主要的游戏系统如 AI、高层的移动控制、入侵检测系统、地图或关卡初始化系统、对象管理器、消息分发系统、图形和碰撞管理系统以及其他类似子系统。

4. 异步测试

测试异步代码是为 MMP 游戏编写单元测试的一个主要挑战。游戏开发人员所调用的代码会把控制权立即交回给调用程序（即单元测试），而在另一个线程、另一个进程或是当前线程主循环的下一次运行时再执行主要功能。大多数基于批处理的测试方式都是同步的，它们期望被测代码阻塞执行直到完成。在异步环境中按照这个假设来进行测试很可能会导致测试

代码在被测代码还没开始运行时就认为它已经完成了。

测试这种代码时，工作人员需要更加小心地设计单元测试。关键就是要在被测代码执行的同时强制单元测试等待它完成。根据异步行为实现的方式，测试完成的方式也有很多种。如果这个异步行为是由一个像网络套接字这样的非阻塞 I/O 调用产生的，就可以使用标准的操作系统调用，像 Linux 上的 select()或是 Windows 操作系统上的 I/O 完成端口来等待一个确认或回复。如果功能被分发到一个独立的线程中去，就可以使用操作系统的线程管理机制来等待特定的事件，然后再检查测试结果。

最后，如果异步行为是通过调度一个方法调用，使它在当前线程下一次循环时被执行所产生的，程序员就可以编写一个回调函数并且让它在被测代码完成后设置一个可以检测到的条件。这种情况主要发生于游戏开发人员是在游戏运行环境中而不是在一个独立的程序中运行测试时。此时，游戏开发人员还必须确保当前线程可以在等待结果时持续运行。为实现这点，可以把主循环封装在一个函数中，并且在测试代码内部调用它。测试自身也是运行在主循环中的，这意味着每次对这个函数的调用会把主循环引入更深一层的递归。当主循环中调用的所有代码都可以被安全地重入时，这么做通常不会产生什么问题。下面的 Python 例子展示了这个概念的意义。

```
import unittest
import gamethread # main game thread module
import dbclient   # database interface
import player, weapon   # persistent objects to test

# object persistence test
class PersistenceTest(unittest.TestCase):
    suite = unittest.TestSuite()

    def setUp(self):
        db = dbclient.InitDB()
        self.conn = db.OpenConn() # open a DB connection
        self.dbFinished = 0

    def dbCallback(self):
        # called when db processing is done
        self.dbFinished = 1

    def testSaveAndLoad(self):
        p = player.Player()
        p.name = 'Test Player'
        p.health = 100
        p.strength = 50
        p.weapon = weapon.LongSword()

        # test save; provide callback
        self.conn.Save(p, self.dbCallback)

        # allow game loop to continue
        while not self.dbFinished:
            gamethread.Iterate()
```

```
# done, now load
p2 = player.Player()
self.conn.Load(p2, dbCallback)

self.dbFinished = 0 # reset

# allow game loop to continue
while not self.dbFinished:
   gamethread.Iterate()

# now check result
self.assert_(p2.name == 'Test Player')
self.assert_(p2.health == 100)
self.assert_(p2.strength = 50)
self.assert_(isinstance(p2.weapon, weapon.Sword))
```

这个方法的主要功能都是在游戏线程（gamethread）模块中实现的，它暴露了 Iterate()（枚举）函数。通过在一个 while 循环中调用这个函数，游戏的其他部分都可以正常运作，直到回调函数被调用并且设置标志来打断这个循环。这是一个强有力的概念，它可以测试一些本来无法测试的地方。

2.4.6　总结

单元测试是一个很有价值的工具，它可以确保那些不断发展的代码库（譬如说 MMP 项目）的完整性。游戏开发人员可以利用现有的框架和工具来提供标准的测试功能，从而把注意力集中在提高测试质量上。xUnit 是最常见的框架，它受到了广泛的支持。工作人员也可以根据测试需要对它进行修改。从极限编程的原理出发，可以使用测试先行的设计、重构和自动化测试过程来确保代码的持续完整。游戏开发人员必须修改测试环境和工具以使它们适合所要测试的代码。这样的修改使得他们不仅可以在独立测试程序中进行测试，还可以在运行环境内部进行测试，并且在必要的时候进行异步执行。

2.4.7　参考文献

[Beck94] Beck，Kent，“Simple Smalltalk Testing: With Patterns,” *http://www.xprogramming.com /testfram.htm*，1994.

[Fowler99] Fowler，Martin，*Refactoring: Improving the Design of Existing Code*，Addison-Wesley，1999.

[Jeffries01] Jeffries，Ron，“What is Extreme Programming?”，*http://www.xprogramming.com/xpmag/whatisxp.htm*，November 2001.

[Langr01] Langr，Jeff，“Evolution of Test and Code Via Test-First Design,” *http://www.objectmentor.com/resources/articles/tfd.pdf*，March 2001.

[PyUnit02] van Rossum，Guido，"unittest --Unit Testing Framework," *Python Library Reference*， *http://www.python.org/doc/current/lib/module-unittest.html*，April 2002.

[XP99] Wells，Don，"The Rules and Practices of Extreme Programming," *http://www.extremeprogramming.org/rules.html*，1999.

[xUnit02] xUnit Testing Frameworks, *http://www.xprogramming.com/ software.htm.*

2.5　使用 Twisted 框架进行 MMP 服务整合

Glyph Lefkowitz，Twisted Matrix Labs
glyph@twistedmatrix.com

本文的源代码包含在所附的光盘中

大型多人在线游戏是世界上最复杂的软件系统之一。让成千上万人在同一个空间奔跑、与同样的敌人作战并且消耗同样的游戏资源是一个相当艰巨的任务。这些游戏还必须面对一个重要挑战：在 MMP 虚拟世界中对外部服务进行整合。开发团队往往会忽视这部分工作并且把它们推迟到游戏运营后由运营团队来进行专门的（ad-hoc）实现。然而，由于 MMP 游戏逐渐面向越来越多的主流玩家，而留住订阅者所需要的资金投入也越来越高，对外部服务进行整合也就变得越来越重要了。

游戏中有一些功能明显需要由外部工具来帮助实现。游戏需要一个聊天系统，就算没有其他用途，至少可以用它来和客户进行交流从而在游戏内部提供支持。这个功能可以用第三方的即时消息系统实现。游戏开发人员还可以使用外部的报表工具（reporting tool）从关系数据库中获取数据来为营销人员和管理人员提供报表。

一旦完成了游戏核心功能的设计，开发人员往往不愿意整合这些外部功能。因为通常要为第三方整合提供机制，游戏开发人员必须在行政方面和技术方面都作出很大的努力，而出于进度方面的考虑，他们常常会放弃这个想法。

整合会带来一些好处，但是这些好处可能不值得为此付出努力。然而，如果对整个游戏生命期进行全盘考虑，就会发现整合显然是无法避免的，游戏开发人员所能采取的最佳行动就是提前为它作好准备。

譬如说，假设在某些版本的客户端软件中有一个错误（这是不可避免的），玩家可以使用某种方法来对游戏进行攻击从而使其他玩家退出游戏。如果用于发行的游戏客户端（或许对于某些授权用户来说还支持游戏管理员扩展）是访问游戏的惟一机制，那么支持人员就无法访问游戏世界，也就无法在开发人员之前改正这个错误。由于不同问题的严重程度和复杂度各不相同，开发人员可能要花费几天甚至几个星期的时间来改正它们。

游戏开发人员往往低估了加入聊天之类的简单功能所带来的复杂度。譬如说，如果游戏玩家觉得游戏中的聊天系统约束过多或功能有限，他们可能会去使用那些能更好地满足他们需要的其他外部程序。有些玩家可能

会在 IRC（Internet Relay Chat）之类的聊天服务上受到其他玩家的骚扰，而服务人员会发现在处理这类抱怨时，他们也许都无法登录那些服务器，更别提获得日志文件或管理员权限了。

相对来说，整合一个关系数据库而不是开发自己特有的数据存储机制所带来的好处已经得到了广泛的认同：编写自己的数据存储层意味着需要编写所有的报表和数据获取工具，更不用说正确地实现存储管理本身就非常困难。

以上这些看上去很像一个对陈腐的面向对象软件重用思想的冗长呼吁。当我们所遇到的问题与业务领域的核心优势不一致时，与其自行编写，还不如整合那些现存的能够运行的可靠方法。即使一个团体能够良好地重用内部开发的代码，也不如直接使用外部代码来得省时省力。

2.5.1 DIY（Do It Yourself）所带来的问题

由整合和扩展带来的问题会在系统的每一层发生。创建一个稳健并且可扩展的底层架构是一个极具挑战性的问题。这和编写一个可以发布运行的简单游戏完全不同。事实上，正如极限编程（Extreme Programming）方法所指出的，过早地为可扩展性作计划是在浪费资源。要决定一个特定的扩展机制是否必须，惟一的方法就是为它找一个用例（参见第 1.5 节，“使用用例来描述游戏行为”）。

这并不是说游戏的开发永远都不需要可扩展性。而是说，创建一个可扩展系统的惟一方法就是让它持续运行一段很长的时间，并且让一个具有不同兴趣的庞大用户群体来使用它，从而暴露出那些真正需要扩展、改进和稳定的地方。

为特定目标创建的框架很难和不同的服务进行良好的整合，这是因为它们面向的用户有限而且开发的时间往往很短。游戏开发人员很难通过对今后开发中所需要的功能进行预期来提供意料之外的功能；错误的猜测会把本来可以用于开发重要的核心特性的时间浪费在实现那些可能毫无用处的扩展上。如果在建立底层框架时考虑不周全，往往会顾此失彼。有时，游戏开发人员所创建的底层架构可以很好地为特定的游戏模式服务，但却有可能最终成为客户服务人员的沉重负担。

最后，如果游戏开发人员从上到下地创建了一个底层架构，即使它卓然超群并且易于扩展，有哪些第三方开发人员（无论是免费的还是商业性的）能够向其中加入有用的特性呢？如果只有一小部分人知道怎样为它设计模块，那么即使是设计得最好的模块化整合软件包也是毫无用处的。

2.5.2 付费找别人来做带来的问题

现有的可以立即使用的 MMP 解决方案都具有相同的问题。虽然它们附带了很多工具，但是其价格都很高昂以至于第三方开发人员无法创建任何其他工具。游戏开发人员最多只能从同一个厂商那里分别购买底层架构的不同组件，但是这种模式并不常见。随着更多的游戏转向 BSD 或是 Linux 服务器并在内部使用像 Python 和 Lua 之类的开源组件，这些稳健的后端工具变得日益重要。

MMP 解决方案包有时会在错误的领域提供灵活性。它们会提供一个游戏世界底层架构的完整实现，这使非常规的游戏模式不能使用；并且，它们也不会提供那些游戏开发人员没有兴趣去尝试的外部工具。

在游戏产业以外，还存在着很多其他类型的底层架构，它们被应用在通用网络、分布式计算以及数据存储中。这些架构中包含了类似于 J2EE API 这样的架构。从产品描述来看，它们似乎可以作为开发游戏底层架构的起点。

然而，绝大多数这类产品都是非常昂贵的中间件，它们往往定位于批量交易处理中。而游戏包含了事件驱动的动态交互。无论一个底层架构是否适合其他整合任务，如果它不能对玩家的行动做出实时反应，游戏开发人员就无法用它来实现游戏世界中的大多数基本操作。

2.5.3　问题小结

要获得一个 MMP 游戏能够高效运行所必需的灵活性，最重要的是使用一个满足下述条件的底层开发架构：支持 MMP 游戏的事件驱动本质；提供了足够的服务因而值得为此付出额外的培训费用；提供了公开的编程接口从而可以利用多个专业群体的技能来解决创建一个完整的 MMP 解决方案所面临的各种领域的问题。

2.5.4　构造一个解决方案

Twisted 网络框架（*Twisted Network Framework*）就是这样一个解决方案。虽然作为一个完整的网络应用程序通用平台，它已经在几个不同的应用领域使用过，但在其设计初期，支持大型多人游戏就已经作为其设计目标之一了。作为一个开源软件，Twisted 网络框架拥有一个大规模的开发人员社区。作为一个消息驱动的网络平台，它同时支持高级编程（使用 Python）以及为满足更高的速度要求（speed-critical）而使用 C 和 C++进行扩展。

Twisted 可能并不适用于所有游戏。如果出于某些原因本文所描述的特定实现不适合某些特殊需要，请记住上述问题仍然需要解决，而使用和本文所描述的方案相似的方法也许是一个不错的主意。

在本文所提供的代码例子中，假定读者熟悉 Python 及基本的网络编程。本节将以一个假设的游戏《变形记在线》（*Metamorphosis Online*）[Kafka]为背景来讨论这些工具，这个游戏中的玩家是患了忧郁症的昆虫。

2.5.5　简介：通用的延迟执行机制

Twisted 使用一个通用的机制来对可能阻塞（block）的行动进行包装，这是它整合网络应用程序最基本的方法之一。在很多异步网络框架中，阻塞行动主要使用回调函数来表示，也就是说，必须把符合特定接口的对象作为一个参数传入任何需要对阻塞操作进行等待的函数。对不同的阻塞操作类型来说，这些接口通常也不同，这会导致两个问题：首先，向回调函数传递额外的参数会很困难；其次，由于接口不同，如果任意两个阻塞系统需要

相互通讯，就必须在它们之间编写一段特定的胶合代码，这常常会引入第 3 个不同的公有接口。

Twisted 通过 twisted.internet.defer 模块来解决这些问题。所有可能等待阻塞操作的函数都会返回一个延迟操作（Deferred）类的实例。一个延迟操作对象起到了两个逻辑函数链的作用，一个用来进行错误处理，另一个用来传递结果。当一个延迟操作对象的结果变为可用时，这个结果会被传递给函数链中的第一个回调函数，然后这个回调函数的结果会被传递给第二个回调函数。如果有错误产生，错误处理回调函数就会被调用。这种方法部分是由 E 语言[Steigler02]中的结果（Eventually）运算符启发而来的，并且具有这个运算符所拥有的大部分预防死锁的特性。同样，因为延迟操作对象可以接受任何具有一个参数并且返回一个结果的函数，很多 Python 标准库函数都可以使用于 Twisted 网络框架中。

下面是一些使用延迟操作的例子。

```
# defer-examples.py

def printResult(x):
    print 'result:',x

# remote method call
remoteObject.callRemote("test").addCallback(printResult)

# database interaction
dbInterface.runQuery("select * from \  random_table").addCallback(printResult)

# threadpool execution
def myMethod(a, b, c):
    return a + b + c

# adding a callback
deferToThread(myMethod, 1, 2, 3).addCallback(printResult)

# Remote method calls that invoke a database connection and wait
# until the database is done returning its query before turning
# the query into a string and then sending the string as the
# result of the remote method call.

class MyClass(Referenceable):
    def remote_doIt(self):
        return self.databaseConnector.runQuery( \
            "select * from foo").addCallback(str)
```

本文已经对此作了简单的介绍，接下去深入了解一下 callRemote（远程调用）是怎样工作的。

2.5.6　高层网络服务：全景代理

Twisted 框架为网络应用程序开发人员提供的最强大的工具之一就是它内建的多用途远程方法调用协议（multipurpose remote method-invocation protocol）和视角代理（Perspective Broker，PB），通常被称为 PB。

PB 是从头开始设计的，它的不少设计目标都有助于解决整合问题，比如以下所列的几点。

- PB 运行在独立的传输层连接上，它同时允许客户/服务器（未授信的、广泛分布的）以及服务器/服务器（授信的、本地集群的）通信。这可以减少系统所需要的通讯代码的总量。PB 具有非常高的可定制性，并且在解码的 3 个层次：(反)列集（marshal）、(反)串行化（serializing）和消息路由上都提供了钩子（Hook）。这使得 PB 在保持很高的通用性和兼容性的同时，仍然可以对特定应用程序的需要做出迅速响应。
- PB 提供了对象间的通信，而不是进程间的通信。这使得多个服务可以在一个连接上进行透明的多路复用，也使得独立的功能可以被整合到现有的服务器和客户中。
- PB 使用动态的方法查询（method lookup）和方法路由（method routing）。这使客户/服务器的版本可以平滑地升级，这样在决定对遗留客户端的支持要维持多久的时候，不会再受到技术上的限制。
- PB 在协议层上提供了安全稳健的认证机制。

这些特性中有很多不仅有趣，而且非常有用，最关键的是它们能够在同一网络连接上对独立开发的服务进行整合，而不必在完全不同的代码上进行耗时的整合工作。这无论是在外部（譬如说，使用从第三方厂商或是志愿者那里获得的代码）还是在内部都很有用。通过使用 PB 或类似的技术，如果两个开发团队工作在彼此无关的抽象功能（譬如说聊天和背囊管理）上，它们之间就可以完全独立，这使得服务端的开发过程可以并行地进行。

认证同样是这个问题的重要部分。无论所访问的服务是什么类型的，用户需要通过这个服务的认证来证明他们有权使用。

Twisted 使用它的认证模式 twisted.cred 来统一 PB 中的服务。

1．twisted.cred 入门

twisted.cred 把认证分解为 4 个主要的抽象概念：

1．twisted.cred.service.Service；
2．twisted.cred.perspective.Perspective；
3．twisted.cred.authorizer.Authorizer；
4．twisted.cred.identity.Identity。

前两个特定于应用程序，后两个则特定于认证方式。

服务（Service）作为一个支配性（over-arching）抽象概念代表了所要提供的整个服务。对于任意需要在一个特定的子系统中进行全局跟踪的对象来说，服务对象所起的作用类似于管理者（Manager）设计模式[EventHelix]。一个服务对象应该包含与一个逻辑服务相关的所

有状态。通常的应用程序，在最后都需要编写至少一个服务类的子类，有时候很可能会需要 2 个到 3 个这样的子类。

服务主要是对视角（Perspective）类的实例进行管理。每个服务的子类通常都会有一个与之相应的视角子类，并且服务只会对它自己实例化的视角实例进行管理。视角代表了与给定服务相关的用户状态。它把用户角色和他们在服务内部所使用的功能联系起来。每个视角都有一个名字。

用来管理认证的对象被称为授权者（Authorizer）。在通常的 Twisted 应用程序中并不需要开发者实现一个授权者，因为它可以使用现有的实现，也可以通过文件、数据库或是内存中的表来进行认证。根据保存认证信息的场所不同，可能需要自己实现授权者来载入或保存账户信息。

一个授权者对象本质上就是标识（Identity）实例的持久化集合。每个标识通常代表一个有权限访问游戏系统的真实用户。只有当需要实现一个新的认证方式时，系统才需要实现一个新的标识类型。Twisted 提供了可以使用明文密码和口令-应答（challenge-response）认证的标识实现。它还实现了一个支持 SSH 的公共密码加密认证机制以支持 SSH 客户。一个标识不仅维护了用户必须提供的证书，还包含了这个用户通过认证后有权访问的视角实例。twisted.cred 和 PB 之间的接口非常紧密。

这里所提供的例子将包括对内建认证机制的使用以及新服务的创建。

2. 一个简单的例子：弗立茨和弗朗茨去看心理医生

在《变形记在线》游戏生涯的第一天，玩家乔和鲍勃登录到游戏中并使用他们的角色：弗立茨和弗朗茨。弗立茨和弗朗茨需要在心理医生的办公室见面以进行小组治疗。

在这个例子中，弗立茨和弗朗茨首先会通过移动电话讨论他们的约会，然后商量怎样在游戏世界中进行历险，最后准时到达办公室。

虽然游戏仿真很难分解为独立的服务，但是对玩家进行全局处理的功能（像日志、审计、消息流量控制、脏话过滤等功能）是肯定可以分开的。

这个例子将实现一个基于移动电话的简单聊天系统。在实际应用中，使用整合的 twisted.words 聊天包及其所附带的工具可能是一个更明智的方法。

关于代码片段

下面的每段代码示例都应该被放在单独的源文件中。这些代码片段，加上一个 Twisted 的安装，应该可以生成一个完整的（虽然很小）应用程序。

```
# cellphone.py - cell phone service
from twisted.spread.pb import Service, Perspective

class Cellphone(Perspective):
    def attached(self, remoteEar, identity):
        self.remoteEar = remoteEar
        self.caller = None
        self.talkingTo = None
        return Perspective.attached(self, remoteEar, identity)
```

```
    def detached(self, remoteEar, identity):
        del self.remoteEar
        self.caller = None
        self.talkingTo = None
        return Perspective.detached(self, remoteEar, identity)

    def hear(self, text):
        self.remoteEar.callRemote('hear', text)

    def perspective_dial(self, phoneNumber):
        otherPhone = \
        self.service.getPerspectiveNamed(phoneNumber)
        otherPhone.ring(self)

    callerID = True

    def ring(self, otherPhone):
        self.caller = otherPhone
        if self.callerID:
            displayNumber = otherPhone.perspectiveName
        else:
            displayNumber = "000-555-1212"
        self.remoteEar.callRemote('ring', displayNumber)

    def perspective_pickup(self):
        if self.caller:
            self.caller.phoneConnected(self)
            self.phoneConnected(self.caller)
            self.caller = None

    def phoneConnected(self, otherPhone):
        self.talkingTo = otherPhone
        self.remoteEar.callRemote('connected')

    def perspective_talk(self, message):
        if self.talkingTo:
            self.talkingTo.hear(message)

class PhoneCompany(Service):
    perspectiveClass = Cellphone
```

第一段代码提供了一个可以独立使用的移动电话服务，它不依赖于任何游戏。在这个服务中，每个玩家都被表示为一个移动电话（Cellphone）实例，用一个电话号码来作为独一无二的标识，这个号码被当作这个视角的名字。注意，那些用 perspective_作为名字前缀的方法，它们可以被这个服务的网络客户直接调用。

```
# metamorph.py
from twisted.spread.pb import Perspective, Service
```

```
class Bug(Perspective):
    angst = 0
    def attached(self, remoteBugWatcher, identity):
        self.remoteBugWatcher = remoteBugWatcher
        self.psychologist = None
        self.angst += 5 # It's hard to get up in the morning.
        return Perspective.attached(self, \
        remoteBugWatcher, identity)

    def detached(self, remoteBugWatcher, identity):
        self.remoteBugWatcher = None
        if self.psychologist:
            self.psychologist.leaveTherapy(self)
            self.psychologist = None
        return Perspective.detached(self, \
        remoteBugWatcher, identity)

    # Remote methods.
    def perspective_goToPsychologist(self, name):
        psychologist = self.service.psychologists.get(name)
        if psychologist:
            psychologist.joinTherapy(self)
            self.psychologist = psychologist
            return "You made it to group therapy."
        else:
            return "There's no such psychologist."

    def perspective_psychoanalyze(self):
        if self.psychologist:
            self.psychologist.requestTherapy(self)
            return "Therapy requested!"
        else:
            return "You're not near a therapist."

    # Local methods.
    def heal(self, points):
        self.angst -= points
        if points > 0:
            feeling = "better"
        else:
            feeling = "worse"
        self.remoteBugWatcher.callRemote(
            "healed",
            "You feel %s points %s. Your angst is now: %s." %
            (abs(points), feeling, self.angst))

class Psychologist:
    def __init__(self, name, skill, world):
        self.name = name
```

```
        self.skill = skill
        self.group = []
        world.psychologists[name] = self

    def joinTherapy(self, bug):
        self.group.append(bug)

    def leaveTherapy(self, bug):
        self.group.remove(bug)

    def requestTherapy(self, bug):
        for bug in self.group:
            bug.heal(self.skill + len(self.group))

class BuggyWorld(Service):
    perspectiveClass = Bug
    def __init__(self, *args, **kw):
        Service.__init__(self, *args, **kw)
        self.psychologists = {}
        Psychologist("Freud", -5, self)
        Psychologist("Pavlov", 1, self)
        Psychologist("The Tick", 10, self)
```

第二段代码定义了一个非常简单的游戏。为了展示视角之间的间接通信，游戏开发人员在视角和服务（心理医生）之上提供了另一个类。同样，那些用perspective_为前缀的方法组成了这个游戏中客户和服务器间的远程接口。

现在系统已经实现了两个完全独立的完整模块。然而，最重要的是实现它们的方式。PB可以利用它们的并行结构，这样只需要一段简短的代码就能把它们整合到一个独立的服务器中。

```
# fritz-franz-setup.py
from twisted.spread.pb import AuthRoot, BrokerFactory, portno
from twisted.internet.app import Application
from twisted.cred.authorizer import DefaultAuthorizer

# Import our own library
import cellphone
import metamorph

# Create root-level object and authorizer
app = Application("Metamorphosis")
auth = DefaultAuthorizer(app)

# Create our services (inside the App directory)
phoneCompany = cellphone.PhoneCompany("cellphone", app, auth)
buggyWorld = metamorph.BuggyWorld("metamorph", app, auth)

# Create Identities for Joe and Bob.
```

```
def makeAccount(userName, phoneNumber, bugName, pw):
    # Create a cell phone for the player, so they can chat.
    phone = phoneCompany.createPerspective(phoneNumber)
    # Create a bug for the player, so they can play the game.
    bug = buggyWorld.createPerspective(bugName)
    # Create an identity for the player, so they can log in.
    i = auth.createIdentity(userName)
    i.setPassword(pw)
    # add perspectives to identity we created
    i.addKeyForPerspective(phone)
    i.addKeyForPerspective(bug)
    # Finally, commit the identity back to its authorizer.
    i.save()

# Create both Bob's and Joe's accounts.
makeAccount("joe", "222-303-8484", "fritz", "joepass")
makeAccount("bob", "222-303-8485", "franz", "bobpass")

app.listenTCP(portno, BrokerFactory(AuthRoot(auth)))
app.run()
```

最后，游戏系统通过为鲍勃和乔创建账户来把这些服务联系起来。这个例子的关键在于makeAccount（创建账户）方法，它创建了与正确的视角实例相关联的标识实例。

通过修改这个文件，可以任意添加更多的服务。这个过程分为 3 步：

（1）实例化新的服务——在“# Create our services（创建我们的服务）”小节中有讲解；

（2）调用服务的 createPerspective（创建视角）方法来创建新的视角；

（3）在相应的标识对象中加入视角的键（key），这样用户一旦通过了这个标识的认证就可以对相应的视角进行访问。

乔和鲍勃怎样才能真正地进入游戏呢？一个完整的交互式客户端足以作为另一篇文章的主题来介绍了，这里本文只提供一个简单的例子，它会登录到《变形记在线》中并且进行 5 秒钟的游戏。

```
# metaclient.py
from twisted.spread.pb import authIdentity, getObjectAt, portno
from twisted.spread.flavors import Referenceable
from twisted.internet import reactor
from twisted.internet.defer import DeferredList

class BugClient(Referenceable):
    def gotPerspective(self, pref):
        print 'got bug'
        pref.callRemote("goToPsychologist", \
        "The Tick").addCallback(self.notify)
        pref.callRemote("psychoanalyze").addCallback(self.notify)

    def notify(self, text):
```

```
            print 'bug:', text

        def remote_healed(self, healedMessage):
            print 'Healed:', healedMessage

    class CellClient(Referenceable):
        def gotPerspective(self, pref):
            print 'got cell'
            # pref.callRemote()

        def remote_connected(self):
            print 'Cell Connected'

        def remote_hear(self, text):
            print 'Phone:', text

        def remote_ring(self, callerID):
            print "Your phone is ringing. It's a call from", \
            callerID

    bugClient = BugClient()
    phoneClient = CellClient()

    # Log-In Information
    username = "bob"
    password = "bobpass"
    bugName = "franz"
    phoneNumber = "222-303-8485"

    # A little magic to get us connected...
    getObjectAt("localhost", portno).addCallback( \
        # challenge-response authentication
        lambda r: authIdentity(r, username, password)).addCallback( \
        # connecting to each perspective with 'attach' method
        # of remote identity
        lambda i: DeferredList([ \
        i.callRemote("attach", "metamorph", bugName, bugClient), \
        i.callRemote("attach", "cellphone", phoneNumber, \
        phoneClient)]) \
        # connecting perspectives to client-side objects
        ).addCallback(lambda l: (bugClient.gotPerspective(l[0][1]), \
        phoneClient.gotPerspective(l[1][1])))

    reactor.callLater(5, reactor.stop) # In five seconds, log out.
    reactor.run() # Start the main loop.
```

客户端代码的主要责任就是为服务器上存在的每个视角对象提供客户对象，以及接收服务器发送的通知。用几行胶合代码就可以把客户端实例连接到服务器端的视角上，其中大部分与服务端胶合代码相同。

图 2-22 所示是对 PB 应用程序所需要的登录过程进行了可视化的说明。客户和服务器需要通过 8 个基本交互步骤来建立一个工作会话，图中用带有数字的箭头来表示。

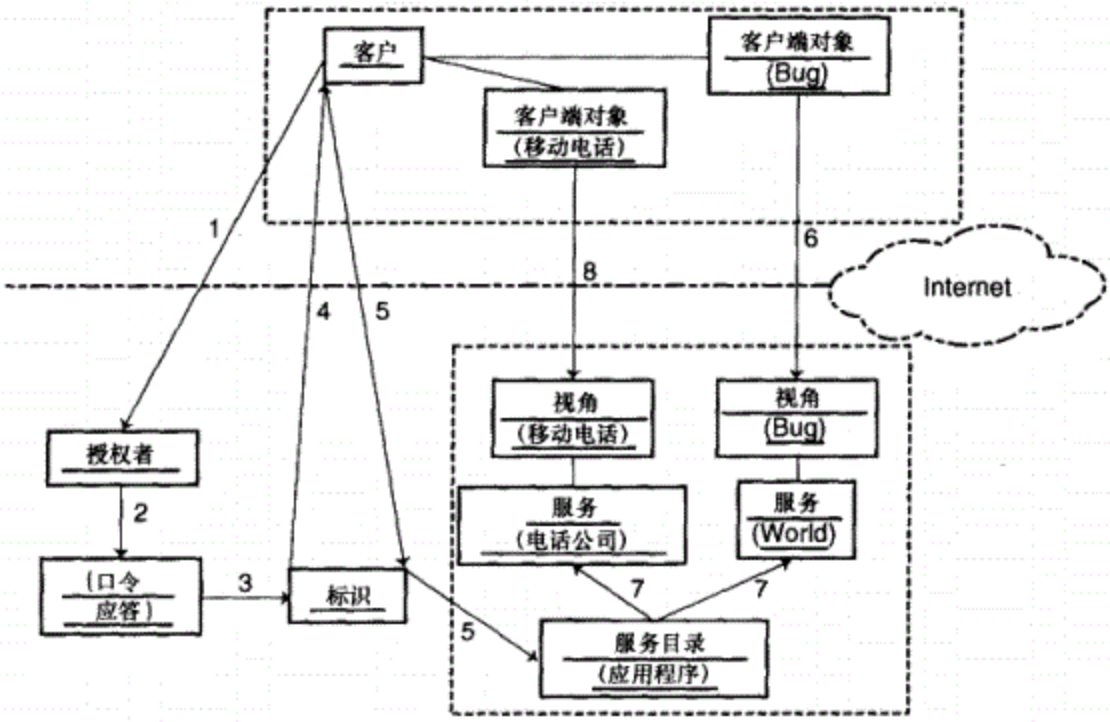


图 2-22 登录过程

(1) 客户联系授权者请求登录。

(2) 客户通过口令-应答会话提供一个用户名和一个密码，确保密码即使在未加密的连接上也是安全的。

(3) 口令-应答过程找到一个标识对象并且进行认证。

(4) 给客户一个指向标识的远程对象引用。

(5) 客户使用服务名/视角名来向某个视角发出请求，并且在找到这个视角后使用标识接口中的 attach（连接）方法把一个指定的“客户”对象和视角连接在一起。

(6) 如果这个标识包含了一个指向服务/视角名的键，它就会去寻找相应的服务。

(7) 这个服务会从它的持久化存储中载入视角并且返回它，然后调用这个被装载的视角的 attached 方法。

(8) 这个视角被返回给客户并且与客户端对象相关联，然后客户对象就可以发起通信了。

3. 其他功能

虽然《变形记在线》是一个非常简单的 Twisted 应用程序实例，但仍然用到了很多有用的技术。这种方法最直接最明显的好处就是，这类极其简单的客户对进行服务器负载测试来说是非常有用的。设计人员必须能够使用简化了的客户来访问游戏服务器，既可以用它来防御正式客户代码中某些不稳定的部分，也可以把它作为一种自动化机制来完成一些简单的常用任务。

Twisted 的底层网络架构对游戏的开发也起到了有益的作用。通过产生一个加密的密钥，游戏开发人员可以用基于行业标准的安全套接字（Secure Socket Layer，SSL）来保护 PB 连接。只需要简单地把服务器对 listenTCP 的调用替换成 listenSSL 就可以了。

这个方法的另一个好处就是它具有良好的安全性。通常，游戏开发人员都想要在公开发布的客户中为游戏管理员提供某些便捷的功能，但是并不想让普通用户访问。在 twisted.cred 中，游戏系统可以把这些危险的功能封装到一个独立的视角中，并且永远不让玩家的标识对象访问这种类型视角的键。这么做可以减少在方法中编写访问检查的需要。由于不需要进行这些检查，也就不会忘记写上这样一个检查——缺省情况下，不安全的功能将不可用。

2.5.7　基于 Web 的工具

在这个日益由互联网驱动的时代，任何关于网络技术整合的讨论都不可避免地包含了对 HTTP 或是传统协议（Legacy Protocol）支持的讨论。Twisted 包含了一个稳健灵活的 Web 服务器，它既可以作为一个独立的服务器使用，也可以用来在 Web 上发布其他 Twisted 对象。

Web 浏览器是一种无处不在的稳健客户端。Web 既可以作为方便的统计报表机制，也可以用作技术支持人员的管理前端，它还可以显示持久化的阶梯（laddering）信息以满足某些玩家自吹自擂的需要，它甚至可以用来为用户提供一个小型游戏客户。

除了 Web 服务器，Twisted 还为基于浏览器的应用程序提供了一个模版（template）和应用程序框架（application framework）。这些工具统称为 *WebMVC*。*WebMVC* 使用了一种较为罕见的方式来处理在创建 Web 应用程序时面对的问题。很多 Web 专用的语言和服务器 API 把关系数据库作为 Web 和应用程序的整合点，并且假设整个应用程序是专门为 Web 编写的。与此相反，Twisted 假设游戏开发人员想要把一些现有的应用程序组件放在网页上，因此它直接把这些对象挂接到 Web 上。

简单地说，*WebMVC* 把 Web 看作一个符合 MVC 模式[Helman]的 GUI 工具箱而不是一个批量输出格式，它是为了与运行中的系统进行基于 Web 的实时交互而设计的。

为了介绍这种把现有的对象发布到 Web 上的方法，本文将为前面的例子整合一些简单的 Web 功能：显示一个所有玩家的列表。

首先，创建一个 HTML 模版 metaweb.html 来说明本文的意图。

```
<html>
<head>
<title>Metamorphosis Player Rankings</title>
</head> <body>
<h1>Metamorphosis Player Rankings</h1>
<table view="bugDisplay" border="1">
   <tr> <th>Rank</th><th>Name</th> <th>Angst</th> </tr>
   <tr rowOf="bugDisplay">
      <td columnOf="bugDisplay" columnName="rank" />
      <td columnOf="bugDisplay" columnName="name" />
      <td columnOf="bugDisplay" columnName="angst" />
   </tr>
</table>
```

```
</body>
</html>
```

WebMVC 要求所有的模版都是格式正确（Well-formed）的 XML（eXtensible Markup Language）。如果游戏开发人员已经很熟悉 HTML，但还不熟悉 XML，那么这不过意味着要在所有不需要关闭的标签的“>”之前加一个“/”，譬如说<HR/>和
。

这个文件只不过是一个模版。它并不包含任何显示逻辑或数据，它只是指定了输出应该是什么样子。注意，加入 view、rowOf 以及 columnOf 这样的属性，就可以把文档中的某些部分标记为被代码所用。

```
# metaweb.py
from twisted.web import wmvc, microdom
from twisted.python import domhelpers

class MBuggyWorld(wmvc.WModel):
    def __init__(self, world, *args, **kw):
        wmvc.WModel.__init__(self, world, *args, **kw)
        self.world = world

class CBuggyWorld(wmvc.WController):
    pass

class VBuggyWorld(wmvc.WView):
    templateFile = "metaweb.html"
    def factory_bugDisplay(self, request, node):
        rowNode = domhelpers.locateNodes([node], "rowOf", \
        "bugDisplay")[0]
        node.removeChild(rowNode)
        bugList = self.model.world.perspectives.values()
        bugList.sort(lambda a, b: a.angst < b.angst)
        rank = 0
        for bug in bugList:
            rank += 1
            rnode = rowNode.cloneNode(1)
            node.appendChild(rnode)
            colNodes = domhelpers.locateNodes( \
            [rnode],"columnOf","bugDisplay")
            bugDict = {"name": bug.perspectiveName,
                "angst": bug.angst,
                "rank": rank}
            for cn in colNodes:
                cn.appendChild( microdom.Text( str( bugDict[ \
                cn.getAttribute( "columnName") ] )))
        return node

wmvc.registerViewForModel(VBuggyWorld, MBuggyWorld)
wmvc.registerControllerForModel(CBuggyWorld, MBuggyWorld)
```

创建了模版以后，还要创建一个表示逻辑（Presentation Logic）模块，游戏开发人员在这个模块中获取数据并使用 W3C（WWW Consortium）标准的文档对象模式（Document Object Model，DOM）对其进行格式化。*WebMVC* 中的显示逻辑大致有 4 个执行步骤。

1．把模版读入一棵 DOM 树。

2．通过深度优先搜索寻找特定的节点（那些具有 view 属性的）。

3．对这些节点调用以 factory_[view 属性]命名的方法。这些方法会修改由模版生成并经过分析的 XML 树中的节点。

4．处理完毕后，把这棵树以 XHTML 流的形式传给浏览器。

现在需要用一种方法来测试这段代码。回到前面的例子（fritz-franz-setup.py），并且修改服务端代码使得它在运行 PB 服务器的同时还运行一个 HTTP 服务器。把下面这段代码加在 app.run()这一行之前。

```
# add-web.py
from twisted.web.server import Site
from twisted.web.static import File
from twisted.web.script import ResourceScript
f = File('.')
f.processors = { '.rpy': ResourceScript }
app.listenTCP(8080, Site(f))
```

这段代码创建了一个 Web 服务器，用来向外界提供执行路径下的文件。这段代码还指定了被作为动态内容来解释的文件类型：任何以.rpy 为扩展名的文件都是一个资源脚本（ResourceScript），这意味着它是一个用来产生 Web 资源实例的 Python 文件。

最后，必须创建一个.rpy 文件（metaweb.rpy），由它来指定这个 Web 应用程序的模式。

```
import metaweb, __main__
resource = metaweb.MBuggyWorld(__main__.buggyWorld)
```

因为 fritz-franz-setup.py 是这个程序的__main__模块，初始化的 BuggyWorld 服务可以在那里找到。

现在这个游戏已经运行了自己的 Web 服务器，它既是一个安全的基于文件系统的全功能 Web 服务器，也是游戏世界的动态窗口。

当然，游戏的企业网站也是一个问题。Twisted Web 提供对 Zope 应用服务器[O'Brien01]的直接支持，而 Zope 正迅速地成为内容管理方面的行业标准。虽然企业网站的开发已经超出了本文的讨论范围，但是与 Zope 的整合使得游戏开发人员可以非常简单地把上面所描写的组件放到一个具有新闻发布、留言板以及一个门户网站所有功能的网站中——要做到这些只需要在一个管理界面上单击几下鼠标即可。

2.5.8　整合独立对象

面向对象编程（Object-Oriented Programming，OOP）长期被吹捧为可以解决整合所带来的问题。每个有经验的软件开发人员都应该知道，这种说法虽然具有一定的正确性，但是被

那些提供软件开发工具的公司无限地夸大了。OOP 那时灵时不灵的特性可以归根于两个原因：a)对于什么是面向对象以及什么不是面向对象的定义过于广泛；b)人们不愿意从制度上去区分 OOP 中所使用技术的好坏。

到目前为止，本文对怎样使用 Twisted 所提供的服务来推动与外部工具的整合进行了讨论。下面将介绍 Twisted 怎样提供这一系列多样的内部服务而没有受到依赖性问题的困扰。

Twisted 设计中与此相关的主导思想是可分离性（Separability）。两个不同的类如果能够简单地独立实例化并且执行各自的功能，那么改变其中的一个会导致另一个发生改变的可能性就会很小。此外，新用户在能够使用一个类之前需要知道的信息应该是越少越好。

通常情况下，可分离的实例是一件好事，因此 Twisted 在大部分情况下避免继承，除非两个类必须被紧密地联系在一起或者基类非常简单、只提供一些公共的功能。

此外，这个方法具有很好的安全属性。当某些对象可以通过网络访问时，很重要的一点就是要仔细地分离那些敏感的操作，并且确保它们远离可能的访问源。如果我们能把大多数功能分散在不同的类中，就可以避免访问控制中的错误。

这是一个古老的概念。事实上，从某种意义上来说这正是 OOP 的起源，它源于仿真中的行动者（Actor）模型。由于网络和安全方面的原因，很多人仍然对行动者模型很感兴趣。这也是一个崭新的概念。各种被狂热追捧的 “组件框架” 本质上可以归结于一个简单的特性：这些架构强迫对象把它们的功能分离到可以独立使用的软件包中。

twisted.python.components 就有这样一个具体而微的组件框架。它主要基于 Zope3 组件框架。虽然提供了一些高级特性，但是其本质还是一个非常简单的系统。它几乎完全基于一个特性：适配器（Adapter）。

下面的代码片段介绍了适配器的基本使用方法。严格地说，适配器只不过是一个具有特殊要求的类：它的构造函数只带有一个参数。适配器被注册后就可以在需要某个特定接口的情况下代表另一个类，因此它至少会实现一个接口。

```
# component-example.py
from twisted.python.components import getAdapter, registerAdapter, Interface

# Define an interface that implements a sample method, "a".
class IA(Interface):
    def a(self):
        "A sample method."

# define an adapter class that implements our IA interface
class A:
    __implements__ = IA
    def __init__(self, original):
        # keep track of the object being wrapped
        self.original = original

    def a(self):
        # define the method required by our interface, and have it
        # print 'a' then call back to the object we're adapting
```

```
        print 'a',
        self.original.b()

# the hapless B class doesn't know anything
# about its adapter and it defines one method which displays 'b'
class B:
    def b(self):
        print 'b'

# register A to adapt B for interface IA
registerAdapter(A, B, IA)

# adapt a new B instance with an A instance
adapter = getAdapter(B(), IA, None)

# call the method defined by interface IA
adapter.a()
```

本文已经介绍了 Twisted 中的组件系统是怎样对外部游戏服务进行整合的。这些例子中所涉及到的大多数“幕后”工作——尤其是关于 *WebMVC* 的例子——都是由适配器实现的。那些使它有助于开发一个独立于所发布对象的 Web 发布系统的特性也使它有助于其他形式的分离——譬如说，独立于游戏世界几何表示的魔法系统。

2.5.9　底层整合：协议与网络

处于 Twisted 底层架构最底层的是 twisted.internet。Twisted 中其他的部分都是在这个部分的基础上构建的。twisted.internet 主要是一个基于 Reactor 事件-响应模式[Schmidt]的网络核心。

Twisted 的设计思想之一就是它在任何层次上（从最高层的消息抽象到网络上传输的位和字节）都是可插拔（pluggable）的。这符合 Twisted 框架早期对 MMP 的想法：我们的 Twisted 应用程序能够根据实际情况满足对效率和灵活性的不同需求。

前面的例子已经使用了 twisted.internet API 来把 Web 和 PB 服务器连上因特网。这里我们将介绍怎样编写自己的服务器。要在这个层次上对 Twisted 进行扩展，最简单例子就是编写一个 echo 服务器。

```
# echoserver.py
from twisted.internet.protocol import Protocol, Factory

# Class designed to manage a connection.
class Echo(Protocol):
    # Method called when data is received.
    def dataReceived(self, data):
        # When we receive data, write it back out.
        self.transport.write(data)
```

```
class EchoFactory(Factory):
    # Build Protocols on incoming connections
    def buildProtocol(self, address):
        return Echo()

from twisted.internet import reactor

# Bind TCP Port 1234 to an EchoFactory
reactor.listenTCP(1234, EchoFactory())

# run the main loop until the user interrupts it
reactor.run()
```

这段简单的 Python 脚本是一个基于 Twisted 框架的全功能异步多路复用服务器。这是一个不错的基本实例，可以在此基础上创建一个能够与现存 MMP 底层框架中的非标准传统协议进行交互的服务端处理系统。

为这些服务器实现其他形式的客户也非常有用，它们可以用来自动执行常见任务或是进行负载测试。Twisted 提供了完整的客户端支持，它被刻意设计得与服务端支持尽可能地接近。

```
# shoutclient.py
from twisted.internet.protocol import Protocol, ClientFactory

# Class designed to manage a connection.
class Shout(Protocol):
    # a string to send the echo server
    stringToShout = "Twisted is great!"

    # Method called when connection established.
    def connectionMade(self):
        # create an empty buffer
        self.buf = ""
        print "Shouting:", repr(self.stringToShout)
        self.transport.write(self.stringToShout)

    # Method called when data is received.
    def dataReceived(self, data):
        # buffer any received data
        self.buf += data

        # if we've received the full message
        # then close the connection.
        if self.buf == self.stringToShout:
            self.transport.loseConnection()
            print "Echoed:", repr(self.stringToShout)
            reactor.stop()

class ShoutClientFactory(ClientFactory):
```

```
    # Build Protocols on incoming connections
    def buildProtocol(self, address):
        return Shout()

from twisted.internet import reactor

# connect to local port 1234 and expect an echo.
reactor.connectTCP("localhost", 1234, ShoutClientFactory())

# run the main loop until the user interrupts it
reactor.run()
```

注意，客户端和服务端的这两个例子都以reactor.run()结束。相同的reactor对象被同时用于客户端和服务端。事实上，任何数量的客户和/或服务器可以在同一个进程中运行。

2.5.10 开发社区

Twisted框架提供了很多其他有用的协议。它目前支持超过10个RFC（Request for Comments），包括因特网邮件协议（Internet mail protocol）、异步数据库访问、域名服务器和客户端以及IRC（Internet relay chat）。

比现存的工具更能让人感兴趣的是，它未来的开发潜力。对于游戏开发人员所选择的平台而言，一个有着持续的兴趣和投入的开发者社区是很重要的，它确保了对这一平台的支持不会在一夜之间消失。同样，能够与一个现有社区合作或是从中获得支持也可以大大地减少开发一个新功能所需要的时间。

其他非开源的项目也可能涌现出像Twisted这样大型而活跃的支持社区，譬如说[Java]。而这样的社区对于一个可靠的框架来说是必不可少的。

开源开发社区通常对那些难以正确解决的、晦涩难懂的问题感兴趣，但是这并不能为任何人带来真正的竞争优势。Twisted框架还具有一个好处：Twisted框架的开发者把它作为自己要使用的工具来开发，因此他们非常注重稳定性。并且，当很多不同的组织致力于同一个项目的开发时，使用Twisted框架可以大大地加强它的稳健性。

2.5.11 总结

对于MMP游戏开发人员来说，它不仅仅是一个游戏，更是一个完整的小型底层架构。他们需要工具来进行持续的开发、营销和支持。如果开发人员所关注的只是游戏，那么提供这些工具会带来非常复杂的问题，它们往往很深奥并且不是很容易解决。

如果存在一个合适的解决方案，那就应该使用它。然而，要找到一个合适的解决方案很困难，特别是它们中的大多数缺少一些有用（如果不是必需的话）的功能。

Twisted框架对于很多游戏来说都是一个有效的解决方案。它提供了多种不同的网络访问机制，包括一个灵活的通用客户服务器协议以及Web发布。另外，在这些协议上又衍生出了像认证之类有用的抽象概念。

Twisted框架在设计时就考虑了安全性。对于其开发团队来说，保持它的安全性和稳定性

至关重要，因为它要用来运行 twistedmatrix.com。

Twisted 框架在很多层次上都是可扩展的。如果所需要的功能还不存在，那么很可能存在创建它的钩子。一个大型多样的开发社区可以确保这一点持续存在。

很少有人会用到 Twisted 框架提供的所有功能，但是通过在一个 Twisted 服务器中提供一到两个辅助游戏服务可以很方便地对某些功能进行运用。对一个底层架构方案进行评估是非常重要的。

2.5.12 参考文献

[Caromel] Caromel，Denis，"Towards a Method of Object-Oriented Concurrent Programming," *http://citeseer.nj.nec.com/300829.html.*

[Cunningham] Cunningham，Ward，et al.，"Proto Patterns," *http://www.c2.com/cgi/wiki?ProtoPatterns.*

[EventHelix] Event Helix，Inc.，"Manager Design Pattern," *http://www.eventhelix.com/Realtime Mantra/ManagerDesignPattern.htm.*

[Helman] Helman，Dean，"Model-View-Controller," *http://ootips.org /mvc-pattern.html.*

[Java] "The Java™ White Paper," Sun Microsystems，*http://java.sun.com/ docs/white/index.html.*

[Kafka] Kafka，Franz，"The Metamorphosis," *http://www.kafka.org/transl/ english/metamorphosis.htm.*

[O'Brien01] O'Brien，Larry，"And Then Came Zope...," *http://www.sdtimes.com/cols/webwatch _ 023.htm*，February 1，2001.

[Schmidt] Schmidt，Douglas C.，"Reactor -- An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching," *http://www.cs.wustl.edu/ ~schmidt/PDF/reactor-siemens.pdf.*

[Steigler02] Steigler，Marc，"E in a Walnut," *http://www.skyhunter.com/ marcs/ewalnut.html*，2000.

[W3C] World Wide Web Consortium Document Object Model Working Group,"W3C Document Object Model," *http://www.w3.org/DOM/.*

[Zadka] Moshe Zadka，"Writing Servers in Twisted"，*http://twistedmatrix.com /documents/howto/servers.*

2.6　Beyond 2：构建虚拟世界的开源平台

Jason Asbahr，ASBAHR.COM，Inc.
jason@asbahr.com

任何试图构建 MMP 虚拟世界的开发团队都会面临很多设计上的挑战。要创建一个持久的三维人造现实世界，并且让处于不同地理位置的用户在其中历险，至少需要以下 3 个架构组件：一个共享状态模型（通常是一个或多个服务器）、一个面向用户的视图（通常是一些图形客户端）和一套控制机制（通常是一个面向行动的请求协议）。无论是什么类型的应用程序，它们的底层技术都具有一些共同的模式。这些公共模式适用于各种类型的应用程序——无论是聊天社区、教育仿真还是娱乐软件。本文建议在可行的情况下要尽量重用现有的技术，而不是在每个项目中都进行重复开发。那些能够获得源代码的技术重用起来是最方便的。这篇文章将以一个名为 Beyond 2 的开源虚拟世界构建系统为背景来描述这些架构设计问题。

Beyond 2 是向网络虚拟世界的统一平台迈出的重要一步。作为一个开源系统，它可以加快那些虚拟世界项目的开发工作。Beyond 建立在另一个开源项目——Twisted 网络服务器[Leftkowitz02]之上并对其加以扩展，从而使任意数量的三维客户端都可以连接到一个共同的仿真框架上。Beyond 还提供了一个客户端的参考实现，它使用了开源三维引擎星云设备（*Nebula Device*）[Nebula02]。服务端、客户端和仿真层共同为创建新游戏提供了一个基石。

这篇文章描述了一些与常用仿真概念、高级架构组件以及通信协议有关的模式。

2.6.1　纵览

设计良好的仿真代码既可以和图形界面一起执行，也可以脱离图形界面独立执行。游戏开发人员通过使用良好的面向对象设计原理把模型与视图和控制器分离开[Burbeck92]，可以使仿真与场景图形及其他子系统保持相对的独立。因为客户端和服务端的区别非常大，所以仿真进程只能在客户端或者服务端运行。“客户”和“服务器”是两个很有用的术语，但是不妨更直观地把它们看作“观察者（observer）”和“主宰者（arbitrator）”。

Beyond 利用适用于仿真域的模型而不是表示方法来建立行动，它为行动系统提供了可以被各种用户界面调用的接口。仿真代码可以支持多个图

形化客户端、文本客户端和仿真实体（simulated entity）的抽象调试视图之间的交互。

因为可以通过共同的接口插入不同的子系统，使用这种方法的项目就可以独立于特定子系统的实现。转移到新的子系统就可以对新技术加以利用，或是在不同的软硬件环境中进行配置。

Beyond 中，在仿真域中的一个活动对象被称为仿真对象（SimObject）[Asbahr98]。仿真元素（simulation element）被建模为仿真对象的属性：实例变量代表了它的状态，方法则实现了被仿真元素的行为。

仿真对象通过维护行动（Action）和行为（Activity）的集合来支持由数据驱动的状态和行为。行动和行为体现了 Beyond 的仿真核心原则——“仿真就是调度（simulation as scheduling）”。在最高层次上，行动是一个瞬时事件，行为则是一段时间中执行的行动集合。仿真接收到的事件会驱动行为（及其包含的行动）的发生，从而对仿真进行更新。

行动和行为都是抽象概念。行动包含了一个仿真对象方法以及相应的参数。行为则包含了一个具有时序的行动集合。行动在行为中的相对时间用一个实数表示，而给定行为中行动间的延迟则是一个变量。在相同时间执行的行动是“同时的”，随着仿真中时间的流逝，系统会计算出一个时间变化，如果行动处于这段时间变化的范围中，它们就会被执行。仿真对象通过调用和执行自己或其他仿真对象的方法来完成行动。仿真对象、行为和行动之间的关系如图 2-23 所示。

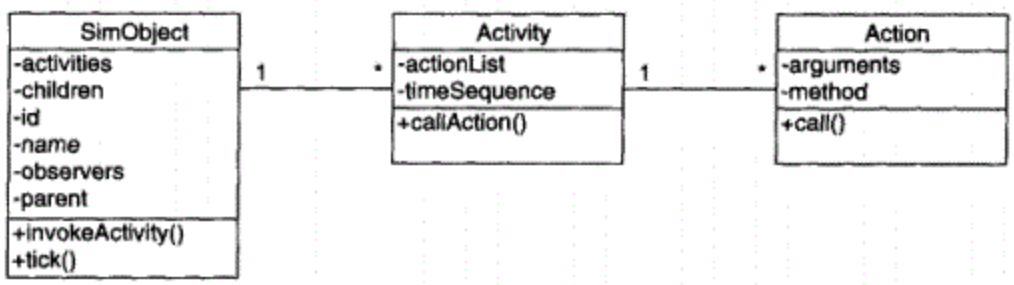


图 2-23 仿真对象、行为和行动类之间的关系

2.6.2 服务端架构

服务器就是维护仿真标准状态的程序（或是程序集合）[Singhal99]。除了要维护组成仿真的模拟对象集合以外，服务器还必须提供一些机制来处理客户请求、进行认证或是在系统中表示用户。

Beyond 2 建立在 Twisted 网络服务器之上。Twisted 是一个开源项目，它由 Glyph Lefkowitz 发起，并得到了全世界很多开发人员的支持。它支持各种通用网络协议、数据库整合、动态 Web 内容的生成，并且还提供了一个被称为视角代理（Perspective Broker）的轻量级高效远程对象协议。

Beyond 2 使用和扩展了 Twisted 框架中的一些类，用于支持虚拟世界的仿真。仿真世界的顶层接口被表示为服务（Service）的一个子类，称为仿真（Simulation）类。它通过授权者（Authrizer）的子类对用户进行管理和认证，每个用户账户被实现为标识（Identity）

类的子类。仿真使用视角类（Perspective）的子类存在（Presence）来表示用户本身。一个存在的对象（Presence）对虚拟世界中一个或多个角色进行管理，每个角色都是角色（Character）类的实例。

Beyond 仿真中的通信——无论是客户-服务器之间的还是服务器-服务器之间的——都通过视角代理来处理。游戏开发人员可以把服务端或客户端对象的方法注册为远程可调用的。这样连接的任意一端就都可以调用它们了。作为参数传递的值和远程方法的返回值会被自动处理，用以在网络上传输。视角代理提供了一个串行化机制，可以在进程间传输、拷贝和缓存结构化数据。它直接支持数字、字符串、列表和词典等基本类型，在登记了串行化和反串行化方法后，用户定义类型也可以被传递。

根据对象在远程可见的形式，仿真对象可以从 4 个视角代理类中的某一个合适类型多重继承而来。这 4 个类包括可引用对象（Referenceable）、可视对象（Viewable）、可拷贝对象（Copyable）和可缓存对象（Cacheable）。可引用对象和可视对象提供了一个远程调用接口，但是不传递状态；而可拷贝对象和可缓存对象传递了状态，但是没有可调用接口。

可引用对象提供的方法可以被远程直接调用。可视对象为对象提供了一个远程代理，它提供了一组方法，每个方法都接受一个用户的视角对象作为第一个参数，并且随着视角参数的不同其执行方式也不同。这有助于实现那些只针对某个特定角色类型执行的逻辑，譬如说具有夜视能力的角色，或是那些能够对仿真进行任意改变的具有管理权限的“游戏管理员”角色。

可拷贝对象的状态在每次对拷贝进行串行化处理时都会被传递到远端。而可缓存对象的所有状态仅在其第一次串行化处理时被传递到远端，在后续的串行化处理中，只传递一个对原始对象的引用。为了支持对象状态的增量更新，可缓存对象为那些存在疑问的状态定义了显式的方法。

服务端仿真循环首先通过调用视角代理的方法，来获得与系统连接的各个客户端或其他服务端的输入。接着，服务端仿真对从远程仿真中获得的仿真更新进行处理，并设置服务端仿真中不同对象的状态。然后，服务端仿真根据一个计算出的时间变化向前一个嘀嗒（Tick）从而使仿真进入下一个状态。最后，用可缓存数据块向客户端广播这个新状态，以更新本地仿真并渲染当前的共享状态。图 2-24 所示是这里所讨论的类层次的一个子集模式图。

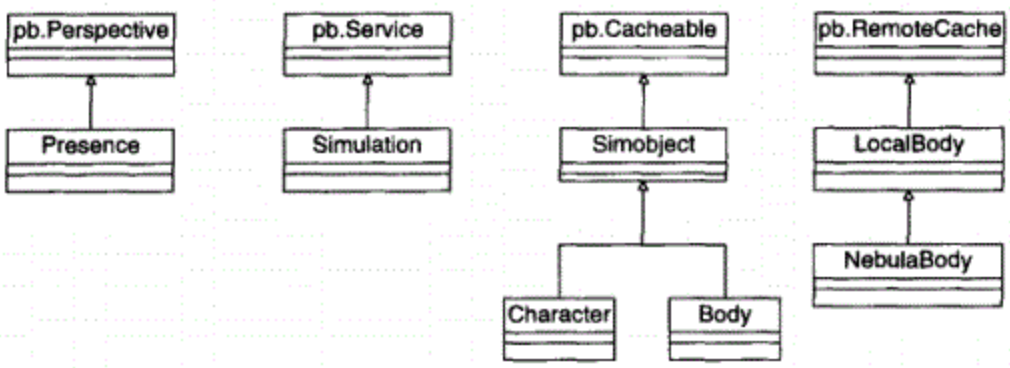


图 2-24　Beyond 2 类层次的一个子集

2.6.3 客户端架构

用于让用户体验虚拟世界的软件通常被创建为一个独立的客户程序。一个或多个客户与服务端相连以获得某个仿真对象子集的镜像并且为用户渲染它们，然后把用户输入转化为行动（Action）请求。

Beyond 系统的客户端参考实现建立在三维引擎星云设备之上。这是氡实验室（Radon Labs）为他们的商业游戏《游牧部落》（*The Nomads*）开发的。它是一个可移植的免费渲染引擎，支持 Direct3D 和 OpenGL，并且可以在 Windows、Linux 和 Mac OS X 上运行。目前 Beyond 已经被整合入星云设备，它能够在进行对象载入、动画和渲染的同时与远程仿真服务器进行通信。

一旦与服务端建立了连接，客户端就会对服务端维护的仿真子集进行镜像并且在本地保存仿真对象的拷贝。这些仿真对象通过视角代理传输，并且在客户端仿真中被表示为本地形体（LocalBody）或是其他类的实例。

客户端从用户处接受输入并将其转换为向仿真提出改变的请求。在 Beyond 中，游戏系统通过向服务端仿真对象发起远程方法调用来发送请求，这些请求都是由 PB 传送的。

客户端与服务端处于一个持续的会话中，它可以接收仿真的更新，并且向特定的仿真对象发送行动请求。当且仅当客户发出的请求在服务端通过所有检查后，它才会被服务端执行。作为状态镜像的仿真对象要以服务端状态为标准进行纠正和调整。

服务端不仅做出了所有的仿真决策，还维护了表示虚拟现实“惟一正确”版本的仿真对象集合[Asbahr01]。访问仿真的客户只不过展示了这一虚拟现实的一个视图。试图对映射到客户端的仿真对象的状态直接进行修改，会改变客户端对仿真的本地表示，但是不会影响到服务端所维护的统一的虚拟现实。

最后，客户端对图像和声音进行实际的渲染，从而以一种用户可以感知的方式向他们展示仿真。在 Beyond 2 中，形体（Body）的缓存版本和它们在场景图（scene graph）中的节点使用星云设备中类似的场景图节点来实现，譬如说三维节点（n3Dnode）、网格节点（nMeshNode）、光照节点（nLightNode）和声音节点（nSoundNode）。

2.6.4 仿真模型

在 Beyond 中，开发人员使用类似于 *Swarm* 系统中所描写的方法[Minar96]把按照层次结构组合的仿真对象放入到包容它们的图（graph）中来构造仿真。

Beyond 把由仿真对象继承而来的合适的抽象概念，连接到对象树中来对不同的问题域进行仿真。譬如说，一个关于牛顿力学的物理仿真可以通过表示质量、体积和速度的仿真对象来构造；而一个关于神秘谋杀的仿真可能会包含表示嫌疑、线索以及杀人武器的仿真对象。

多个仿真可以并行运行，并且某个给定仿真中的仿真对象可能与另一个仿真中的仿真对象存在代理/通知（delegation/notification）关系。在前面的例子中，在一个仿真中对指纹和子弹进行跟踪的武器仿真对象可能与另一个仿真中跟踪位置、方向和质量的物理仿真对象相关

联。用这种方式可以组合出复杂的虚拟世界。Beyond 给出了对虚拟世界进行仿真的 4 个层次：

- 物理层；
- 逻辑层；
- 认知层；
- 叙事层。

1. 物理层

虚拟世界系统的物理仿真框架需要支持对空间中场景的描述、度量和操纵。Beyond 进行物理仿真主要包括了对 4 个方面的考虑：图像、体积、力和碰撞。

物理仿真层中的仿真对象代表了场景中的“模型（model）”（如 *WavesWorld*[Johnson95] 中所描述的），它们对形状、投影和状态信息进行了封装。这些抽象被称为形体（Body），它们所包含的行动对象和行为对象可以对场景中特定的物理元素进行构造和操纵。因为形体是仿真对象，所以它可以包含其他形体，形体通过这种方式层次化地组织起来形成了场景图。形体中的行动对象和行为对象可以对场景图和动画集进行参数化控制。

虚拟环境的可视部分是通过创建场景图的子集来构造的。形体把相互连接的可塑三维几何片断组合形成场景图子集。它们可以从磁盘上载入几何信息或是调用场景图 API 来动态地构造几何信息。形体可以被放置、旋转或是作为动画播放。它们可以通过场景图 API 来激活、关闭所控制的场景图子集的不同特性，也可以为这些特性的浮点参数赋值。

2. 逻辑层

Beyond 的逻辑仿真框架主要用于创建可以表示现实世界中实体的仿真对象，并管理它们之间的交互。逻辑仿真对象对于人类来说有象征意义——譬如说，物理仿真中的一个橡皮球体在逻辑仿真中可能会被表示为一个“篮球”仿真对象。

Beyond 对象框架为逻辑仿真提供了一些类，如对象通信碰撞反应、抽象逻辑交互、行为初始化以及角色控制。这些仿真对象表示了一些人类可以理解的抽象概念和操作规则，譬如说角色、房子以及武器。

逻辑仿真对象包括开着、关着或是锁着的门，可以造成伤害的武器，可以吸收伤害的防具，可以用来制造物品的工具以及环境中静止的元素，譬如说具有碰撞属性的树、灌木和岩石。

逻辑仿真对象和物理仿真对象的相互联系，使得像压碎一个箱子或是开弓射箭这样的逻辑行动可以触发那些会让几何物体作出相应的消失、显示或移动的行动或行为。

3. 认知层

Beyond 的认知仿真框架主要用来处理虚拟世界中那些会“自我思考”的元素。对于它们所控制的逻辑仿真对象来说，认知仿真对象起到了“虚拟大脑”的作用。

这一层次的抽象包括传感器（Sensor）、受动器（Effector）以及行为（Behaviour）。认知仿真对象通过把作为输入的传感器和作为输出的受动器与内部的状态机（包含了动作（Behaviour）的行为集合）联系起来，对对象的自治（autonomy）进行仿真。传感器可以检测距离和方向，还可以使用同心包围体（concentric bounding volume）来检测与其他仿真对象

间的碰撞。传感器把对象组合成碰撞列表来进行处理[Terzopoulos94]。受动器是可扩展的行动对象，它与移动、通信以及对其他逻辑对象的操作有关。特定领域的仿真可以为角色（Character）增加额外的行动（Action）功能，譬如说战斗、制作、询问等。

认知仿真还引入了角色（Character）这一抽象概念，用来表示在虚拟世界中有感觉的事物。角色可以由玩家控制或是完全自治（autonomous）。

由于角色在仿真中的目的或功能不尽相同，一个自治的认知仿真对象所具有的动作模型可能很简单，也可能很复杂。很多游戏中都有一些在受到足够大的伤害后会“死去”的有意识生物（sentient being），在这种情况下，仿真对象需要对健康和知觉进行建模。类似于害怕、恼怒、期望、精力这样的情感状态也可以被仿真，这些状态可能会发出特定的角色行为请求，譬如说逃跑、攻击或是以不同的速度和频率进行追逐[Bate94]。

4. 故事层

故事仿真是最高层次的仿真，它捕捉了虚拟世界中具有游戏性或是故事性的方面。这个层次的仿真对象主要对人物、故事分支、点数、胜利条件以及失败条件进行处理。它们通过以下方式和逻辑层及认知层的仿真对象进行交互：初始化仿真对象中的值，注册可以调用的触发器，在虚拟世界中创建或放置仿真对象。故事层仿真对象是处于最顶端的仿真对象，它为下面的层次提供了控制机制，并且把事件和动作紧密地组合在一起。

最简单的故事层仿真对象莫过于那些记录游戏或关卡统计数据的仿真对象，它们可以用来记录获得物品或是给出物品的数量、交谈过的角色、已完成的任务之类的信息。

典型的故事层仿真对象集合由一个包含了几个关卡（level）仿真对象的游戏（game）仿真对象构成，每个关卡仿真对象都包含了一些任务（Quest）仿真对象和目标（Goal）仿真对象。

游戏仿真对象会封装最外层的故事情节；关卡仿真对象会封装故事片断和对相关地形、逻辑及角色元素的引用；任务和目标仿真对象则封装了这些故事片断中可以独立衡量的步骤。此外，日志仿真对象还可以对每个用户的状态和分数进行记录。

故事层仿真是最具有研究潜力的。对其进行扩展就可以用来包括更复杂、更容易引起共鸣的故事，也可以用来包括动态生成的故事或是教学系统[Bruckman98]。

2.6.5 总结

使用 Beyond 就意味着，游戏开发人员通过使用编程平台来进行虚拟世界的开发。使用 Beyond 的游戏通过对仿真模型进行扩展来获得适合游戏仿真领域的类和属性。物理模型会被扩展为新的场景图节点（譬如 BSP 节点、粒子发射器）；逻辑模型会被扩展为新的虚构交互类（譬如刀、叉）；认知模型会被扩展为新的人工智能（或是玩家）类（譬如龙、毒蛇或是毛茸茸的小兔子）；而故事层模型则会被扩展为新的故事成分或是游戏流程类（譬如关卡、打杂任务、独白以及闹剧笑话）。

为进入这一市场，很多公司和开发团队开始创建他们自己的 MMP 世界。然而，由于从头开发一个虚拟世界系统所需的努力和代价非常巨大，很少有个人或是公司能够成功地完成这个任务。通过在一个开源系统中实现不同的 MMP 功能，Beyond 2 有助于减少新 MMP 游

戏的开发成本和营销时间。Beyond 2 是一个正在进行中的项目，欢迎开发人员访问项目站点：http://www.beyond2.org。

2.6.6　参考文献

[Asbahr98] Asbahr，Jason，"Beyond: A Portable 3D Simulation Framework," Proceedings of the Seventh International Python Conference: pp. 103–109, *http://www.asbahr.com/papers.html.*

[Asbahr01] Asbahr，Jason，"Python for Massively Multiplayer Worlds," Proceedings of the Ninth International Python Conference: pp. 39–42，*http://www.asbahr. com/papers.html.*

[Bates94] Bates，Joseph，"The Nature of Characters in Interactive Worlds and the Oz Project," *The Virtual Reality Casebook*，Van Nostrand Reinhold Publishing Co.，1994: pp. 96–102.

[Bruckman98] Bruckman，Amy，"Community Support for Constructionist Learning," ACM Conference on Computer Supported Cooperative Work，Seattle，Washington，November 14–18，1998.

[Burbeck92] Burbeck，Steve，"Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)," 1992，*http://st-www.cs.uiuc.edu/users/smarch/ st-docs/mvc.html.*

[Johnson95] Johnson，Michael B.，"WavesWorld: A Testbed for Three-Dimensional Semi-Autonomous Characters," MIT Media Lab Ph.D. dissertation，1995，*http://xenia.media.mit.edu /~wave/PhDThesis/outline.html.*

[Leftkowitz02] Leftkowitz，Glyph，et al.，"The Twisted Network Framework," Proceedings of the Tenth International Python Conference: pp. 83–101，*http://www.python10.com /p10-papers /09/index.htm.*

[Minar96] Minar，Nelson，et al.，"The Swarm Simulation System，A Toolkit for Building Multi-Agent Simulations," June 1996，*http://www.santafe.edu/projects/swarm/overview /overview.html.*

[Nebula02] RadonLabs.de，"The Nebula Device 3D Engine," *http://www.radonlabs.de*，September 2002.

[Singhal99] Singhal，Sandeep，et al.，*Networked Virtual Environments*，Addison-Wesley，1999.

[Terzopoulos94] Terzopoulos，Dimitri，et al.，"Artificial Fishes with Autonomous Locomotion，Perception，and Learning in a Simulated Physical World," *Artificial Life IV*，Brooks & Maes，ed.，MIT Press，1994: pp. 16–27.

2.7 使用并行状态机来创建可信的角色

Thor Alexander, Hard Coded Games
thor@hardcodedgames.com

本文的示例代码包含在所附的光盘中。

当游戏开发人员可以获得更多的资源来创建更有深度、更引人入胜的游戏模式时，他们就会开始试图让游戏角色具有更多强大且富于变化的能力。由玩家和 AI 控制的角色可以执行更多种类的行动，这些行动的对象可以是其他角色以及游戏世界中的物品。随着可供执行的行动越来越多，游戏设计人员所创建的角色就能以更可信、更动人的方式来执行行动。

本文讨论了怎样使用并行（parallel）状态机来控制这些可信的角色。状态机（State Machine）是进行游戏开发的常见工具。

状态机提供了下面这些属性，这让它成为实现游戏设计目标的理想工具。

- 状态为针对行为改变进行建模提供了良好的机制。
- 状态以及它们之间的转换可以用易读的状态图来说明。
- 状态可以在客户端和服务端之间复制以使用于在线游戏中。
- 状态可以在多个游戏子系统间重用，如图 2-25 所示。

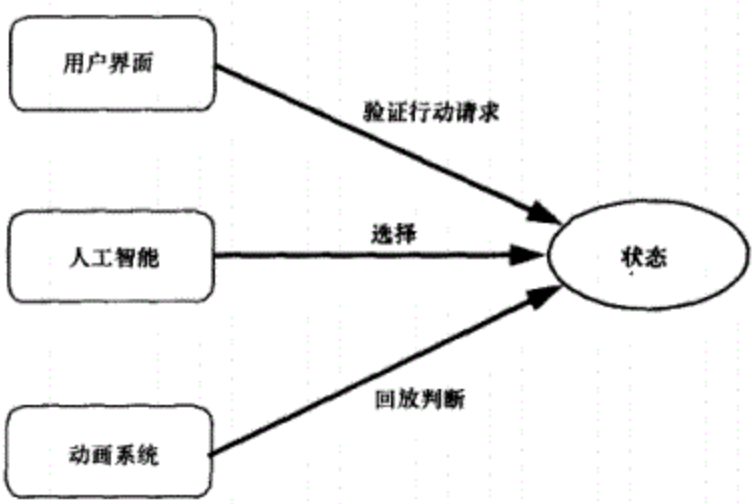


图 2-25 角色状态可以用在很多地方

本文所包含的某些图表是使用 UML 标记来表示的，读者可参考 UML[Booch98]以获得对 UML 的深入讨论。

2.7.1　状态模式（State Pattern）

如果分离（separation）可以从多态（polymorphism）这一面向对象特性中获益，那么把行为分离到不同的对象中是一个有用的方法。多态使得我们可以用相同的方式使用两个对象，调用同样的方法（method），即使这些对象实现这些方法的方式完全不同。多态使得超类（super class）可以定义一个通用的接口，由子类在必要的时候实现具体的方法。

状态模式是一个非常有用的软件设计模式[Gamma94]，它利用多态来为同一个对象的不同状态定义不同的行为。图 2-26 所示是一个状态模式的 UML 类图。这里，行动者（也就是 Actor 对象）有一个名为当前状态（currentState）的对象成员，它的类型是具体状态（ConcreteState）。这个状态方法的接口由它的超类抽象状态（AbstractState）定义。这个子类实现了超类指定的 Enter（进入状态）和 Exit（退出状态）方法。这些方法在状态机实现中很常见，如果在运行时进入或退出一个状态，它们将被分别调用。

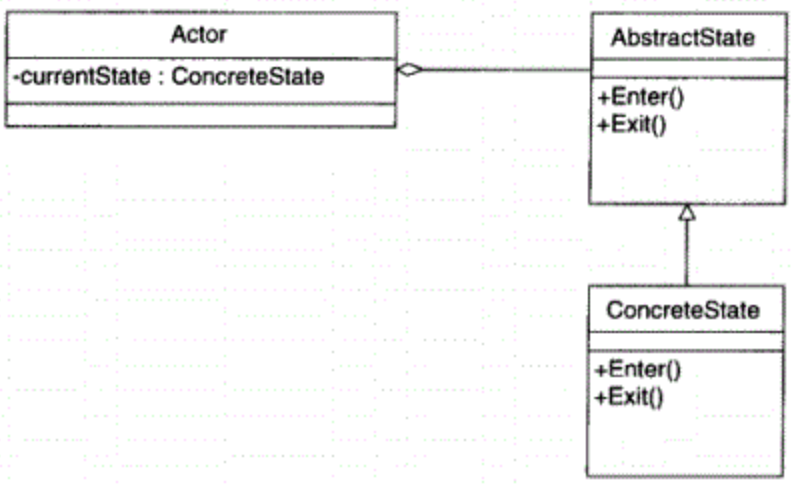


图 2-26　状态模式的 UML 类图

图 2-27 是对这个例子进行了扩展并且描述了加入多态后的状况示意图。这里使用了一个简单的例子，它具有 3 种具体状态，分别对应于行动者能够进入的 3 种可行的战斗状态。这些战斗状态中的每一都继承自抽象类状态（State）。这个角色具有一个 SetCurrentState（设置当前状态）方法，这个方法可以让当前状态属性迁移到一个新的战斗状态中。

下面这段代码是一个用[Python]实现的例子。

```
class Actor:
    def __init__(self):
        self.fightingStatePunch = State.Punch()
        self.fightingStateKick = State.Kick()
        self.fightingStateBlock = State.Block()
        self.currentState = self.fightingStateBlock
```

```
    def SetCurrentState(self,targetState):
        self.currentState.Exit(self)
        self.currentState = targetState
        self.currentState.Enter(self)

    def Punch(self):
        self.SetCurrentState(self.fightingStatePunch)

    def Kick(self):
        self.SetCurrentState(self.fightingStateKick)

    def Block(self):
        self.SetCurrentState(self.fightingStateBlock)
...
```

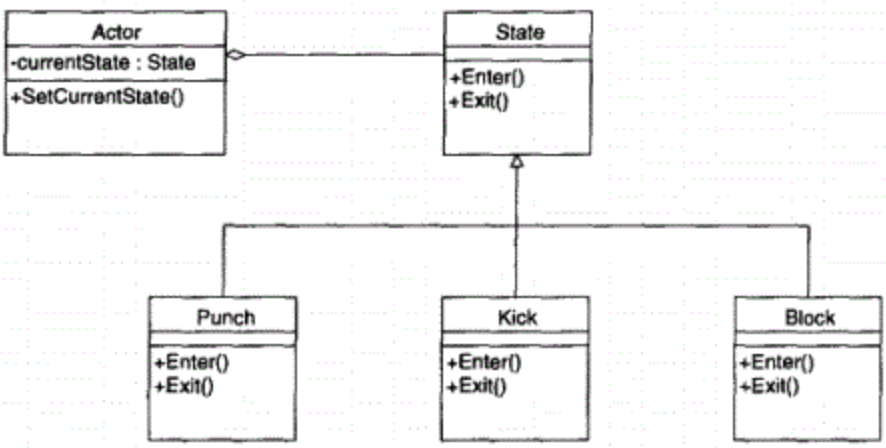


图 2-27 使用多态的状态模式

注意，上面的例子是怎样把这个行动者对象传入 SetCurrentState 方法中，并且调用 Enter 和 Exit 方法的。

2.7.2 并行（Parallel）状态层

随着游戏设计人员为角色加入越来越多的功能，它们的状态图会变得日益复杂以至于难以处理。为了控制复杂度，这些功能应该被分解到独立的概念层次中去。如果某个功能与某一层中其他功能只能以互斥的方式执行，就把这个功能放入这一层中。而可以独立或是并发执行的功能则被放入不同的层次。图 2-28 展示了 MMP 游戏角色常见的行为，它们被分为 3 个层次。

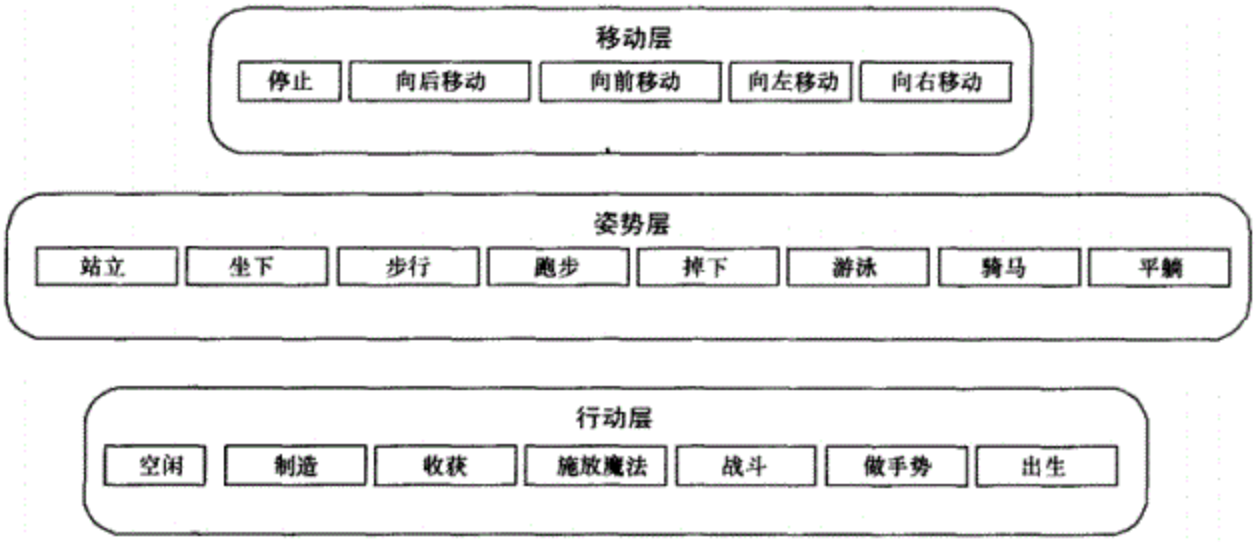


图 2-28　把互斥行为分解到并行的层次中去

1. 移动层

让游戏中角色可以在游戏世界中四处移动是最基本的行为。在大多数现代游戏的实现中，移动（movement）独立于正在播放的实际动画（譬如说，步行、跑步、游泳）。移动仅对表示角色碰撞模型的二维或三维对象在游戏世界中的平移（translation）进行处理。这是游戏的移动层所负责的范围。为了清楚起见，本文仅对平移进行讨论，并且忽略任何可以控制角色旋转（譬如说，左转或右转）的额外层次。设计并实现这样一个转向（steer）层将作为留给读者的练习。

表 2-1 中的状态对应于角色需要执行的移动行为。通过对这些状态两两之间的合法转换进行定义，可以把它们组合为一个移动状态机。这样就可以从一个状态转换到另一个状态，也可以在同一个状态中循环，如图 2-29 所示。自身循环的状态表示一个可以被重复的行为，譬如说坠落或是步行。

表 2-1　　移动状态

移动状态	描述
Stop（停止，缺省状态）	休息，不移动
MoveBackward（前移）	向后移动
MoveForward（后移）	向前移动
MoveLeft（左移）	保持面向前方，向左移动
MoveRight（右移）	保持面向前方，向右移动

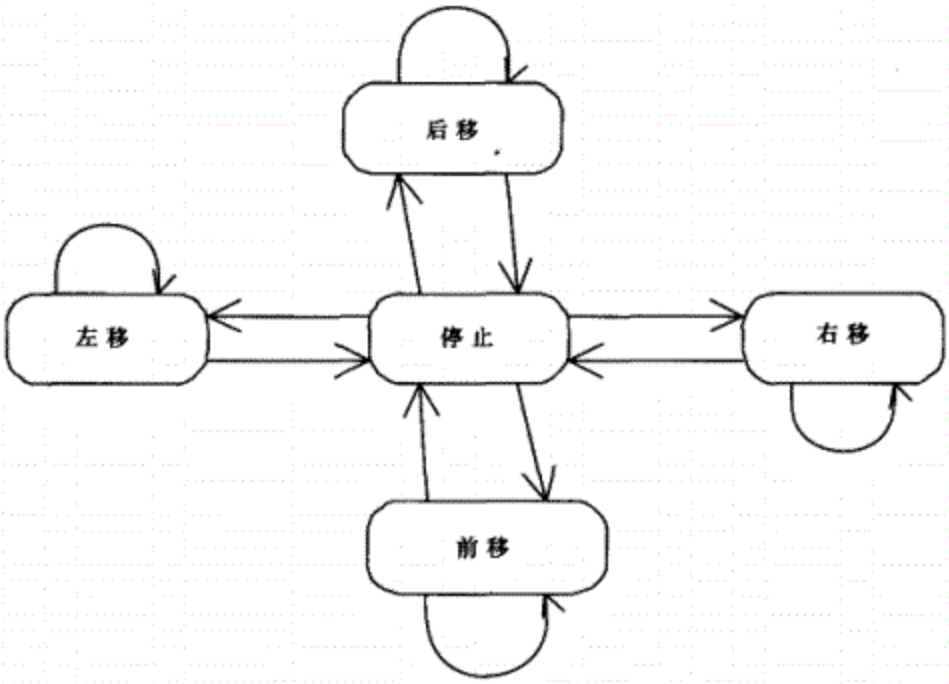


图 2-29 移动层的状态图

本文已经定义了移动状态和它们之间的转换，接下来可以把它们转化为一个类图，如图 2-30 所示。每种状态都从公共超类移动状态（MovementState）中继承，如图 2-29 中所示的状态模式那样。这些状态子类将从抽象超类移动状态中继承它们共享的方法接口。

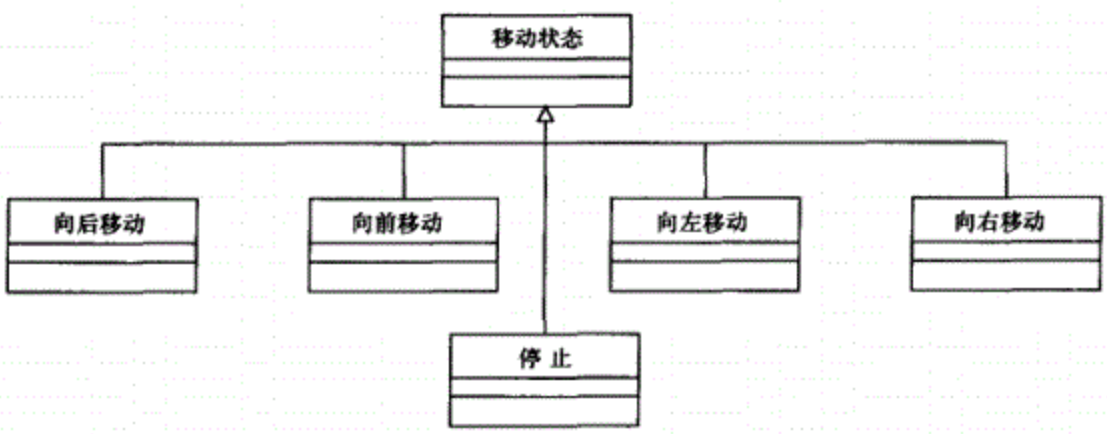


图 2-30 移动层的类图

下面的 Python 代码给出了移动状态层的部分实现。

```
class MovementState():
   def __init__(self):
      self.id = 0
```

```
        self.transitionList = []

    def CanTransition(targetState):
        if targetState.GetId() in self.transitionList:
            return 1 ### Valid transition.
        else:
            return 0 ### Invalid transition.

    def GetId(self):
        return self.id

    def Enter(self,actor):
        return

    def Exit(self,actor):
        return

class MoveForward(MovementState):
    def __init__(self):
        self.id = 1
        self.transitionList = [
            MovementState.MoveForward.GetId(),
            MovementState.Stop.GetId() ]

    def Enter(self,actor):
        ### perform move-forward tasks on the actor here.
        …
return

    def Exit(self,actor):
        ### perform state-exit cleanup tasks here.
        …
return
…
```

注意每个状态对象都维护了各自的惟一标识（id）和转换列表（transitionList）属性。每种状态可以进行的合法转换保存在它的转换列表中。CanTransition（能否进行转换）方法会使用这个列表来检查是否可以从当前状态合法地转换到目标状态（targetState）。

2. 姿势层

这个层次表示了角色可以呈现的各种姿势（posture）。移动层处理了与角色在游戏世界中实际移动相关的问题，而姿势层则处理了角色在移动时的外观。这一层控制的是类似于从坐下到站立或从站立到步行这样的转换。它还可以帮助我们管理更高级的行进方式，譬如说骑马和游泳。图 2-31 和表 2-2 列出了一些基本的姿势状态以及它们之间的合法转换。

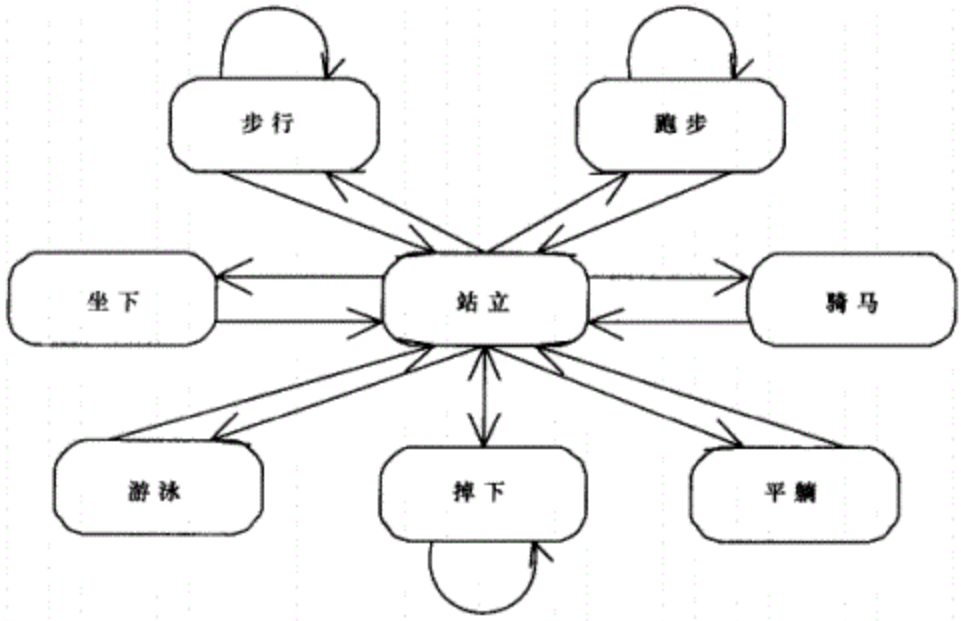


图 2-31 姿势层的状态图

表 2-2 姿势状态

姿势状态	描述
Standing（站立，缺省状态）	空闲的站立状态
Sitting（坐下）	在地上坐着
Walking（步行）	在地上步行
Running（跑步）	在地上跑步
Falling（掉下）	正在空中往下掉
Swimming（游泳）	在水中游泳
Mounted（骑马）	骑马
Sprawled（平躺）	躺在地上

姿势层有两个显而易见的用途：a)通知动画子系统什么时候应该改变这个角色正在播放的动画；b)接受或拒绝用户界面提出的状态改变请求。它还有一个非常有用但是不太明显的用途：他可以把这些状态传给看得到玩家的敌人的 AI，这样这个敌人就可以根据玩家姿势的改变作出反应。譬如说，AI 可能会忽视一个处于中立姿势的玩家，但是如果玩家拔出武器进入战斗姿势，AI 可能就会对其进行攻击。

图 2-32 所示的姿势层的类图也遵循状态模式，它从抽象超类姿势状态（PostureState）中派生出所有的具体姿势状态。姿势层的实现和前面所示的移动层代码非常接近，因此这里不再重复。

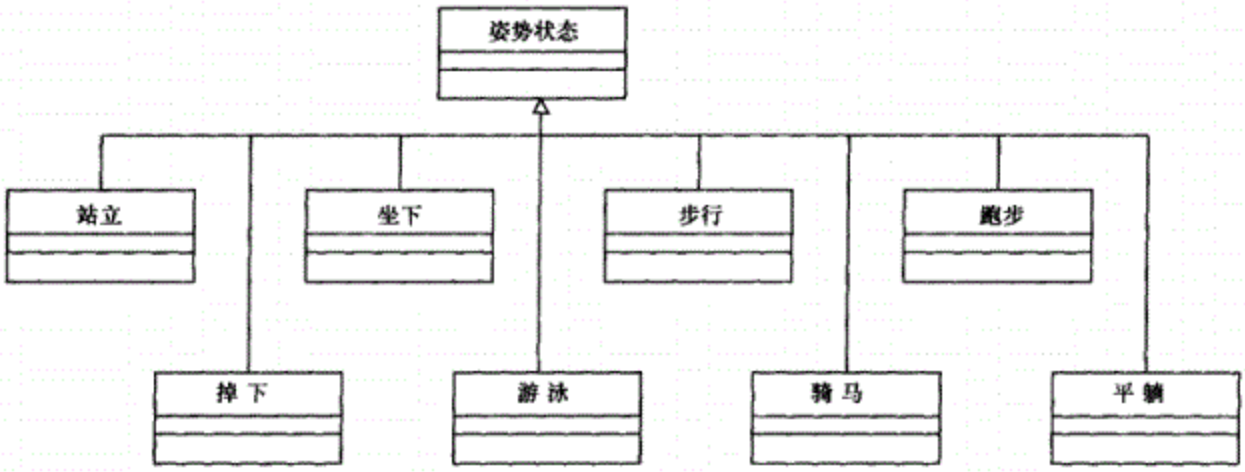


图 2-32　姿势层的类图

3. 行动层

游戏系统的最后一层处理角色可以进行的各种行动。这里，本文把行动（action）定义为那些不会随着姿势和移动的变化而变化的行为。譬如说，游戏中角色应该能够挥动他的剑，无论他是在跑步、站立不动还是在骑马。游戏设计人员可以把行动层理解为对上身（upper-body）行为进行处理，而姿势层是对下身（lower-body）行为进行处理。不过也会有例外，某些情况下需要把上身的状态也归为姿势层并且阻止行动层，譬如说在游泳或从高处掉下时。本文会在随后讨论的跨层阻止（cross-layer blocking）中处理这种例外。图 2-33 和表 2-3 的行动层和行动状态包含了 MMP 游戏中常见的行为。这些部分组合在一起形成了行动层的类图，如图 2-34 所示。

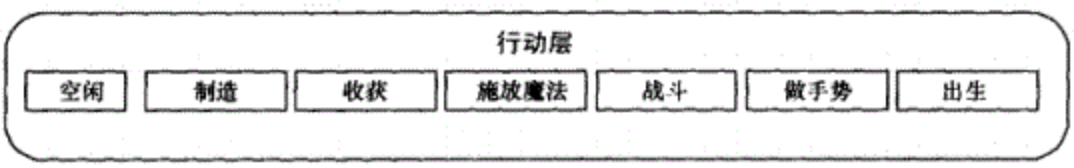


图 2-33　典型 MMP 游戏中的行动层

表 2-3　　行动状态

行动状态	描述
Idle（空闲，缺省状态）	空闲状态
Crafting（制造）	建造或是修复物品
Casting（施放魔法）	使用一个魔法符咒
Fighting（战斗）	攻击一个角色或对象
Gesturing（做手势）	挥手，跳舞，等等
Spawning（出生）	在游戏世界中诞生

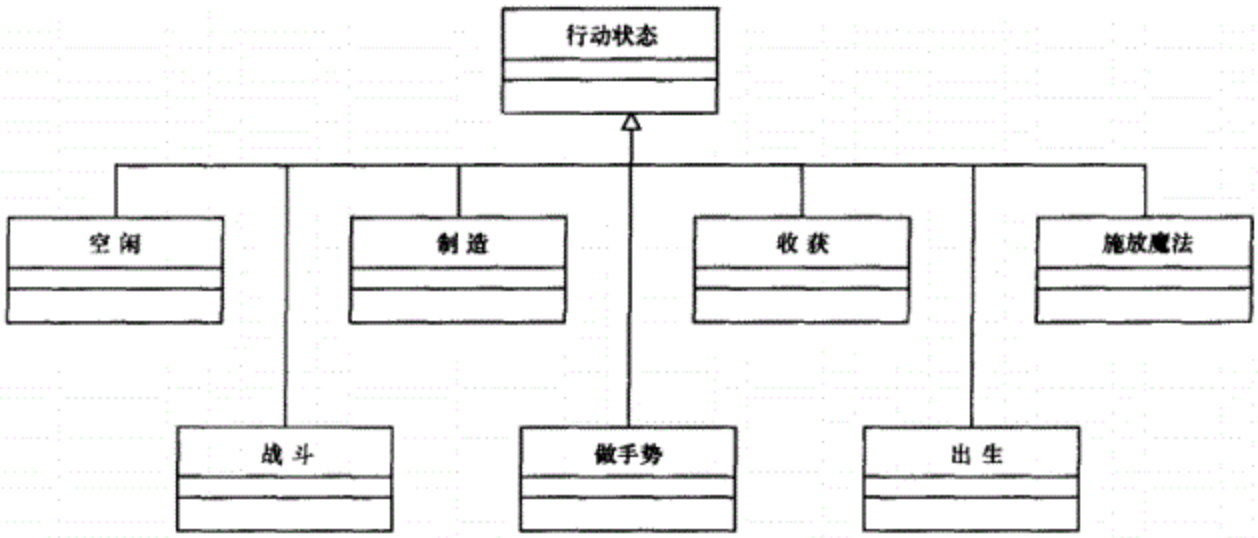


图 2-34 行动层的类图

2.7.3 状态管理器

角色所执行的行为已经分离到了并行的层次中，这时需要另一个类来对它们进行管理。这里要介绍的是状态管理器（StateManager）类。行动者类的每个实例都具有对这个管理器的引用功能。状态管理器维护了每个状态层当前状态的引用：当前行动状态（currentActionState）、当前移动状态（currentMovementState）以及当前姿势状态（currentPostureState），如图 2-35 所示。

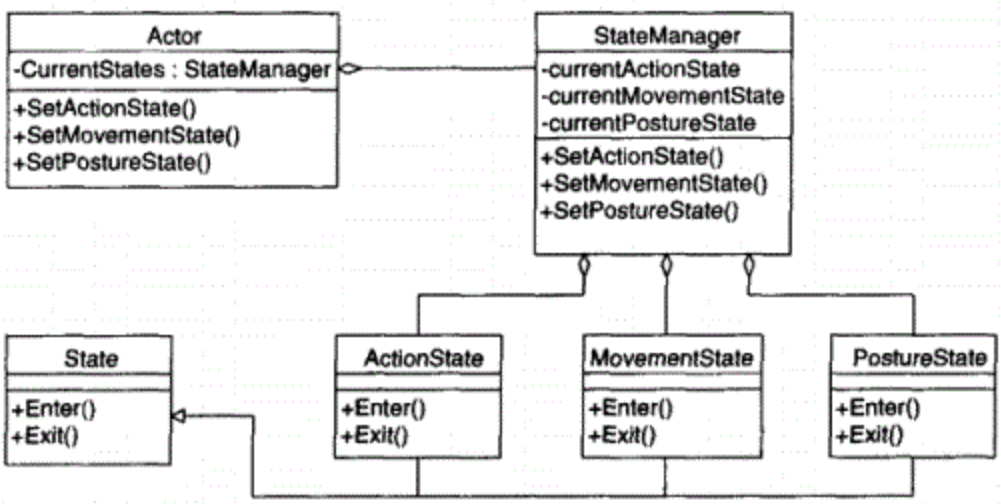


图 2-35 我们状态层的状态管理器的类图

下面这段 Python 代码是一个状态管理器的实现。

```
class StateManager:
    def __init__(self):
        self.currentActionState = ActionState.Idle()
```

```
            self.currentMovmentState = MovementState.Stop()
            self.currentPostureState = PostureState.Stand()
            …

        def SetActionState(self,actor,targetState):
            if self.currentActionState.CanTransition(targetState):
                self.currentActionState.Exit(actor)
                self.currentActionState = targetState
                self.currentActionState.Enter(actor)
                return 1 ### Successful transition.
        else:
            return 0 ### Cannot transition.

        def SetMovementState(self,actor,targetState):
            if self.currentMovementState.CanTransition(targetState):
                self.currentMovementState.Exit(actor)
                self.currentMovementState = targetState
                self.currentMovementState.Enter(actor)
                return 1 ### Successful transition.
        else:
            return 0 ### Cannot transition.
    …
```

2.7.4 跨层阻止（Cross-Layer Blocking）

在大多数情况下，每层的状态可以独立于其他层次执行，但是总会存在一些例外。譬如说，如果游戏的角色从很高的电梯上掉下来，处于空中时，游戏设计人员想要让他不能进行移动或是做出任何行动。为实现这点就需要一个能够进行跨层阻止的机制。

如果在移动状态超类中增加一个布尔变量 blocked（是否被阻止），就能通过 Block（阻止）和 Unblock（取消阻止）方法设置或清除它，代码如下。

```
class MovementState():
    def __init__(self):
        self.id = 0
    self.transitionList = []
    self.blocked = 0 ### false

def Block(self):
        self.blocked = 1

    def Unblock(self):
        self.blocked = 0
```

接着可以在 CanTransition 方法中加入一个新的检查，并在状态被设为阻止时返回一个表示转换失败的代码。

```
def CanTransition(targetState):
```

```
        if self.blocked == 1:
            return 0 ### Transitions are blocked

        if targetState.GetId() in self.transitionList:
            return 1 ### Valid transition.
        else:
            return 0 ### Invalid transition.
```

现在就可以阻止移动状态的转换了。

```
class Falling(PostureState):
    ...
    def Enter(self,actor):
        actor.currentStates.SetMovementState(actor,StopState)
        actor.currentStates.SetActionState(actor,IdleState)
        actor.currentStates.currentMovementState.Block()
        actor.currentStates.currentActionState.Block()
        ...
        return

    def Exit(self, actor):
        actor.currentStates.currentMovementState.Unblock()
        actor.currentStates.currentActionState.Unblock()
        ...
       return
```

这里，姿势状态 Falling 的 Enter 方法会强制对象进入移动状态 Stop 以及行动状态 Idle，然后阻止这些状态进行转换。一旦角色退出 Falling 状态，它会取消对这些状态的阻止。

2.7.5 总结

本文介绍了一个并行状态机制的设计和实现方法，它基于久经考验的状态模式。要在下一代游戏中加入更为可信的角色需要创建更高级的状态机制，而本文中并行状态机制的设计和实现则是一个良好的开端。开发人员还可以把状态模式中的每个状态都定义为一个单件（singleton），这样一来，系统中每个给定状态在内存中就只有一个实例。这么做可以把对内存的使用保持在一个较低的水平，本文推荐使用这个方法来对上面所介绍的方案进行优化。

2.7.6 参考文献

[Booch98] Booch, Grady, *The Unified Modeling Language User Guide,* Addison-Wesley, Inc., 1998.

[Gamma94] Gamma, et al., *Design Patterns,* Addison-Wesley Longman, Inc., 1994.

[Python] Python Language Web site, *http://www.python.org.*

2.8　在 MMP 服务中使用观察者/可观察者设计模式

Javier F. Otaegui，Sabarasa Entertainment
javier@sabarasa.com

在[Gamma95]中介绍的观察者/可观察者（Observer/Observable，O/O）设计模式可以成为创建 MMP 游戏架构的强大工具。本文介绍 O/O 设计模式在 MMP 引擎架构设计中的应用方法。

这里所要介绍的结构是一个多功能 MMP 游戏仿真复制引擎（all-purpose simulation replication engine）的基础。这个结构可以把游戏程序员从类似于对象复制和消息管理这样的乏味工作中解放出来。它可以在客户端为游戏仿真建立一种良好的近似状态，这正是多人游戏引擎最重要的职责[Sweeney99]。

2.8.1　观察者/可观察者设计模式

O/O 设计模式包含两个抽象类：观察者和可观察者。这两者之间彼此交互。一个可观察对象具有其观察者的引用集合。当可观察对象被修改时，它会调用所有观察者的 Touch()（修改）或是 Update()（更新）方法来通知它们（见图 2-36）。当观察者需要获得所观察对象的状态时，它首先检查这个对象是不是脏的（即是否被修改了），然后查询其内部状态并且更新自己的内部状态。

观察者可以充当各种角色。譬如说，游戏系统可以实现一个 3D 观察者（3DObserver），它知道怎样在 3 维引擎中表示一个特定对象；或是一个等容观察者（IsometricObserver），它会把自身显示在一个等容世界（isometric world）中。游戏系统还可以提供用来备份的观察者，它可以把对象状态持久化到数据库或是文件中。

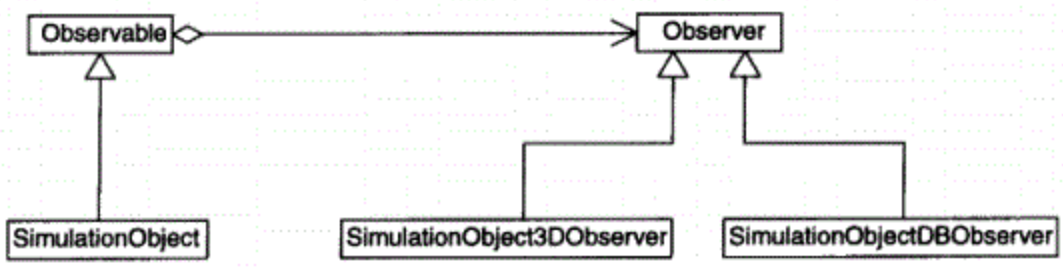


图 2-36　观察者/可观察设计模式的 UML 图

2.8.2 基本架构

O/O 设计模式对于 MMP 游戏的设计来说非常有用，它使得开发人员可以在服务端和客户端使用完全相同的类，从而只需要对游戏世界实现一次仿真。在服务端，有一些类会对仿真进行观察并且把检测到的所有改变持续地传输到客户端；然后在客户端使用同样的仿真对象并使用一个图形观察者来观察它们，这个图形观察者知道怎样在设计人员为游戏选择的图形引擎中生成并显示这些对象。

在服务端和客户端运行相同的仿真的好处是，可以为两个仿真输入同样的时钟信号，从而在服务端和所有客户端（也就是有效的仿真代理）得到同样的结果。服务端会在累积一段时间内的改变后再发送给客户端来进行修正和更新。由那些不可预知的事件（譬如说用户进行了某个行动）触发的改变也可能会被累积起来。

不仅如此，在服务端还可以建立其他的观察者对象。使用观察者对象不仅可以把整个仿真备份到文件或是数据库系统中，还可以让系统管理员使用游戏世界管理工具来查询底层仿真的细节（见图 2-37）。

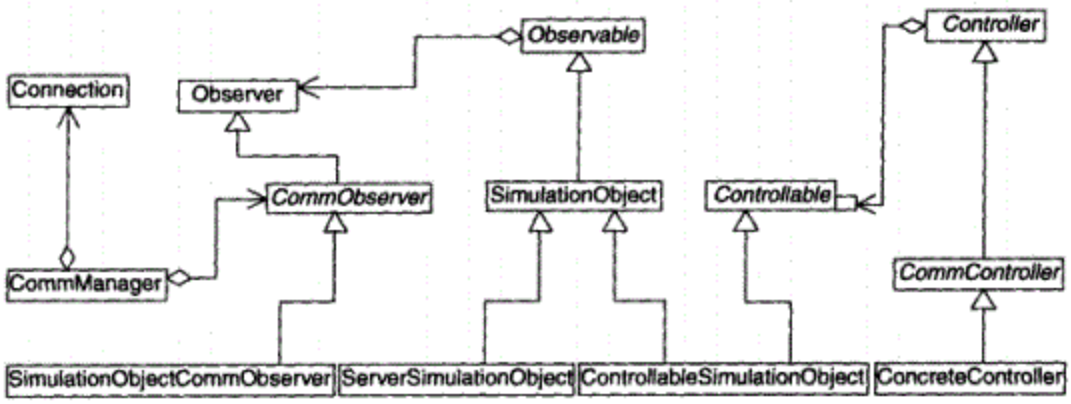


图 2-37 基本系统架构的 UML 图

2.8.3 服务端架构

服务端是由仿真对象、相应的通讯观察者以及通讯管理器（CommManager）组成的。仿真对象在服务器启动时创建。每个仿真对象都会创建与之相应的观察者并将其注册到通讯管理器中。服务端会处理仿真对象的创建、修改和删除。每当有新的客户与服务端连接并且试图接收完整的仿真状态时，服务端会对其进行处理。

每个仿真对象都实现了可观察接口。不仅如此，游戏系统管理员还需要在服务端程序中定义一些特定的类来把服务端行为加到仿真对象中去。通讯观察者（CommObserver）类实现了观察者接口，并且还加入了与通讯相关的方法。每个仿真对象都有一个对应的仿真对象通讯观察者（SimulationObjectCommObserver），它从通讯观察者类继承产生，这要求它必须定义 Pack()（打包）方法（见图 2-38）。

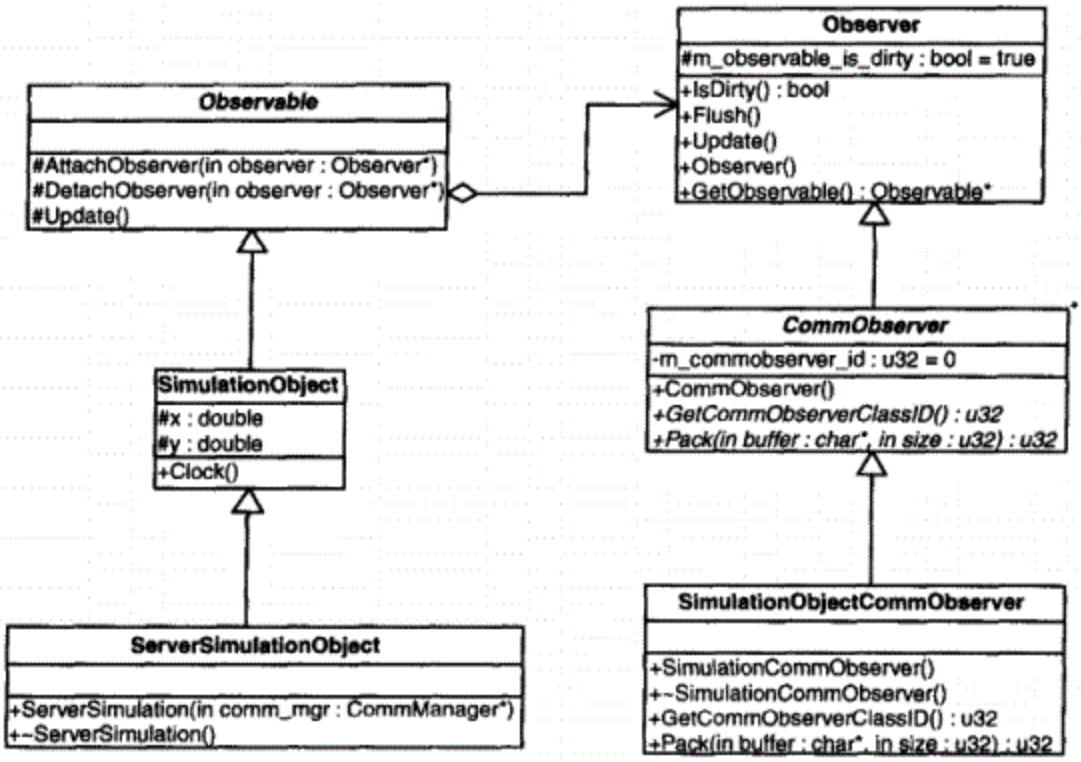


图 2-38　服务端仿真架构的 UML 类图

通讯管理器负责跟踪所有的通讯观察者实例。它还可以发送不同类型的网络消息。它除了提供客户端订阅和取消订阅（也就是建立/断开连接）的方法以外，还提供 SendChanges()（发送改变）方法。后者会通过循环调用所有的通讯观察者对象来把更新消息发送给客户（见图 2-39）。

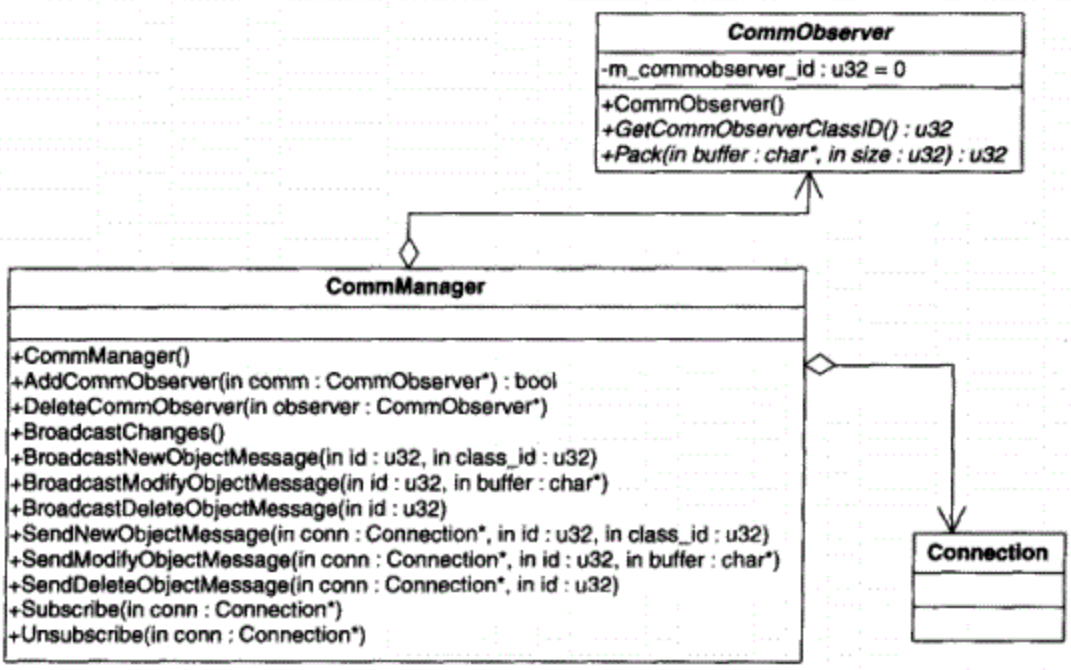


图 2-39　服务端通讯架构的 UML 类图

1. 更新消息（Update Message）

通讯管理器拥有对所有通讯观察者的引用。当服务端进入一个特定状态（超时，或是发送队列为空）时，服务端会让通讯管理器发送自上次对所有连接的客户更新以来，发生的全部改变。

接着服务端会循环处理所有的仿真观察者。如果某个观察者被标为脏，就查询其内部状态以获得一个将要发送给客户的更新缓存（update buffer）。每个观察者有必要了解它所观察的对象：怎样读取对象的数据；怎样把数据打包进一个字节流。服务端会保留这个缓存，并且生成一个包含了观察者标识的修改消息，这样客户就能知道应该修改对象池中的哪个对象了。客户在收到这些更新后，会用它们来和服务端的仿真状态进行同步处理。

当一个仿真对象包含了其他对象的指针（譬如说一个包含了物品的背囊）时，这个对象的引用也会被发送到客户端。缺乏经验的实现者可能会动态地为每个观察者分配一个惟一标识。但是这会导致 Pack()方法出现问题。虽然观察者了解它们所观察的仿真对象，但是它们无法在不违反 O/O 设计模式的情况下把不同的观察者区分开来（见图 2-40）。解决这个问题

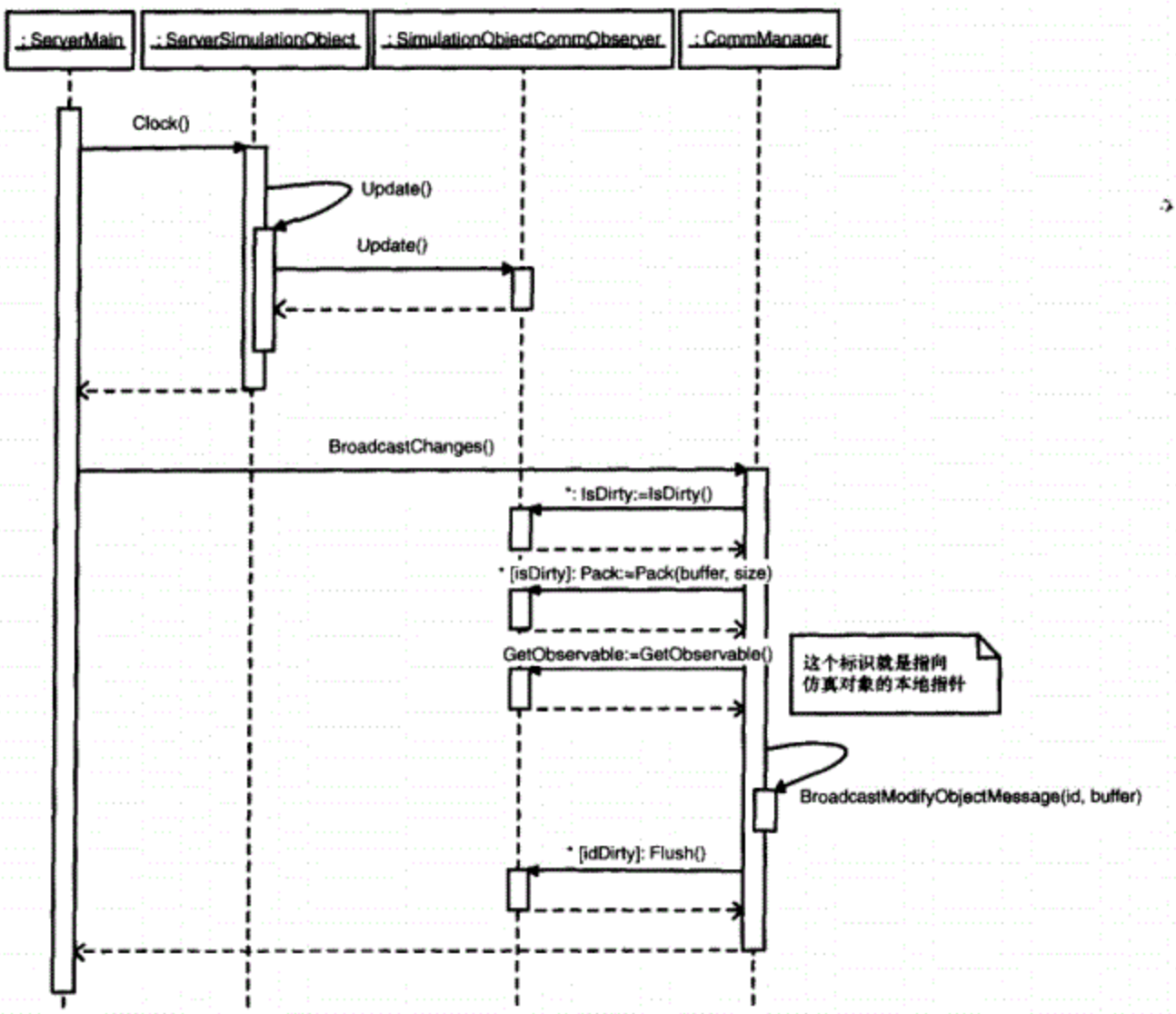


图 2-40 服务端更新对象的 UML 顺序图

的方法是把指向被观察对象的地址值作为惟一标识。这在游戏服务器集群上可能会有问题，因为这时的内存空间可能会重叠。通过创建由机器标识和指针地址值构成的复合标识就可以简单地解决这个问题。下文会对解包方法进行分析。

2. 创建消息（Create Message）

当服务端仿真要创建一个新的仿真对象时，它必须向所有客户端通知与这个新建对象有关的信息。它必须发送一个包含了类标识和对象惟一标识的对象创建消息。

当新的仿真对象被创建后，它必须创建与之对应的通讯观察者。这是游戏开发人员需要把仿真对象类特化为服务端仿真对象类的原因之一。相应地，客户端仿真对象也会创建它们的图形用户接口观察者（GUI observer）。

当一个通讯观察者被创建后，它必须把自己挂接到通讯管理器上。类标识必须通过调用观察者的 GetClassID()（获取类标识）方法来查询。对象标识则由被观察仿真对象的指针值来确定。接着通讯管理器向所有客户发送包含了类标识和对象标识的 NewObjectMessage（新对象消息）（见图 2-41）。

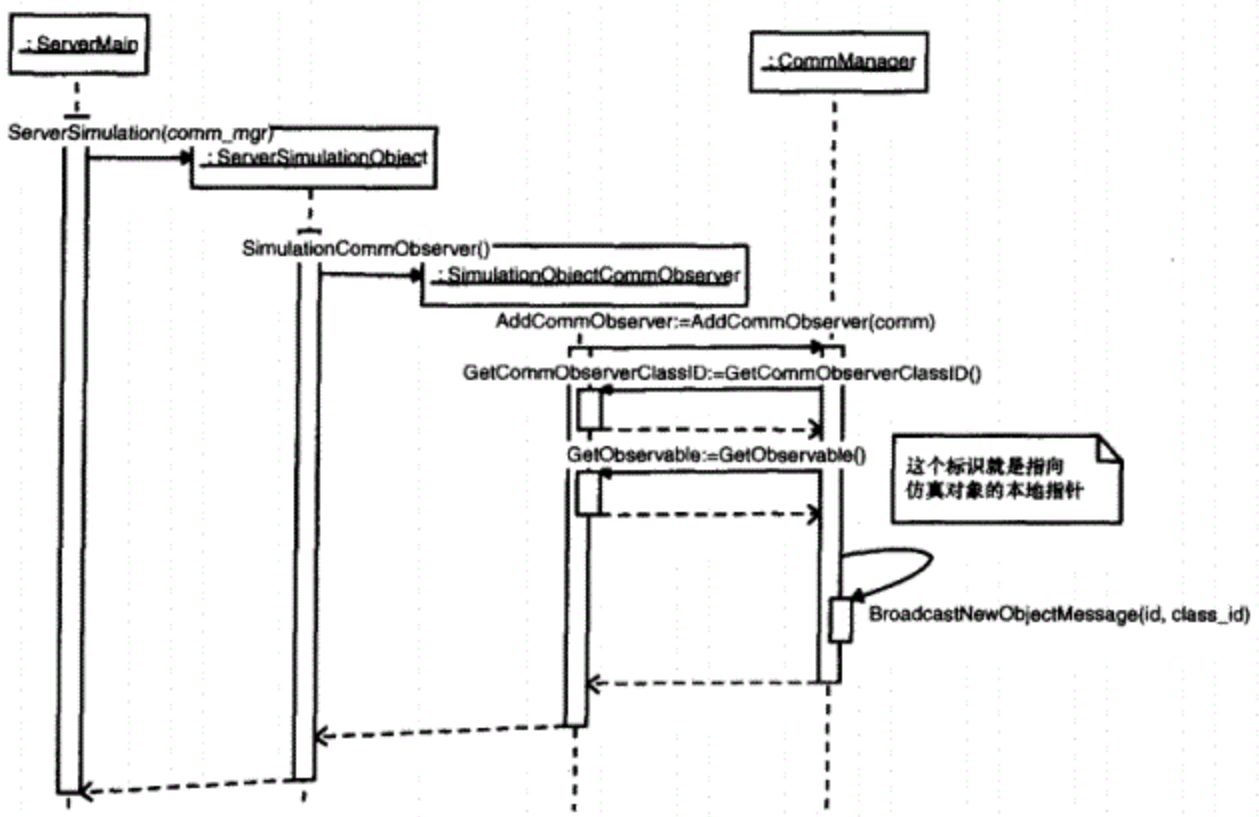


图 2-41　服务端发送 NewObjectMessage 时的 UML 顺序图

3. 删除消息（Deletion Message）

删除对象类似于创建对象，但在发送 DeleteObjectMessage（删除对象消息）时只需要观察者的标识就可以了。当一个仿真对象被销毁时，它对应的所有观察者也必须被销毁（游戏系统的管理可能不仅使用了通讯观察者）。因为在每对观察者/可观察之间有着一对一的映射

关系，每个可观察对象还必须负责删除它的观察者对象。

当一个通讯观察者被销毁时，它必须把自己从通讯管理器中删除。删除的方法负责向所有正在监听的客户发送删除消息。客户收到消息时会把观察者删除，并且通知所有其他仿真对象以使那些对象可以删除任何相关的引用，如图 2-42 所示。

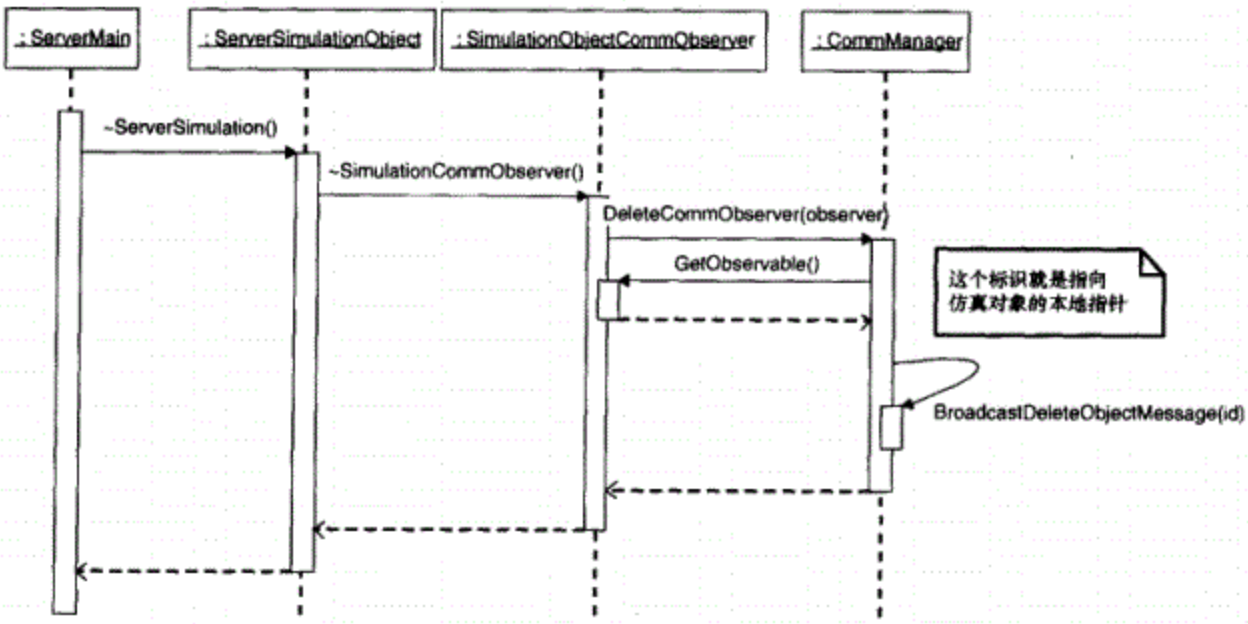


图 2-42 服务端删除对象时的 UML 顺序图

4. 新客户消息（New Client Message）

服务端还需要对新客户订阅仿真这种情况进行处理。在接收更新消息前，客户端必须获得仿真的完整状态。

当一个新客户订阅仿真时，必须先接收仿真中所有对象的创建消息。在它创建了这些对象后，还必须通过一系列更新消息来与所有对象的内部状态达到同步。

在一个大型仿真中，这个方法会占用大量的带宽从而引发性能问题。有几个办法可以对其进行处理——最简单的方法是在对象传输时使用一个基于优先级的方案，优先级最高的（可能也是最相关的）对象最先被更新；优先级较低的对象在后续仿真循环中被更新，如图 2-43 所示。

2.8.4 客户端架构

客户端由多个仿真对象及与它们相对应的图形观察者和一个通讯控制器（communications controller）组成。每个仿真对象都实现了可控制（Controllable）接口，用来解包服务端发来的消息并且更新它们的内部状态。通讯控制器对仿真对象的创建、修改以及删除活动进行处理。它不需要对创建新客户的情况进行处理，因为这些事件是被当作独立的仿真对象创建和修改的，如图 2-44 所示。

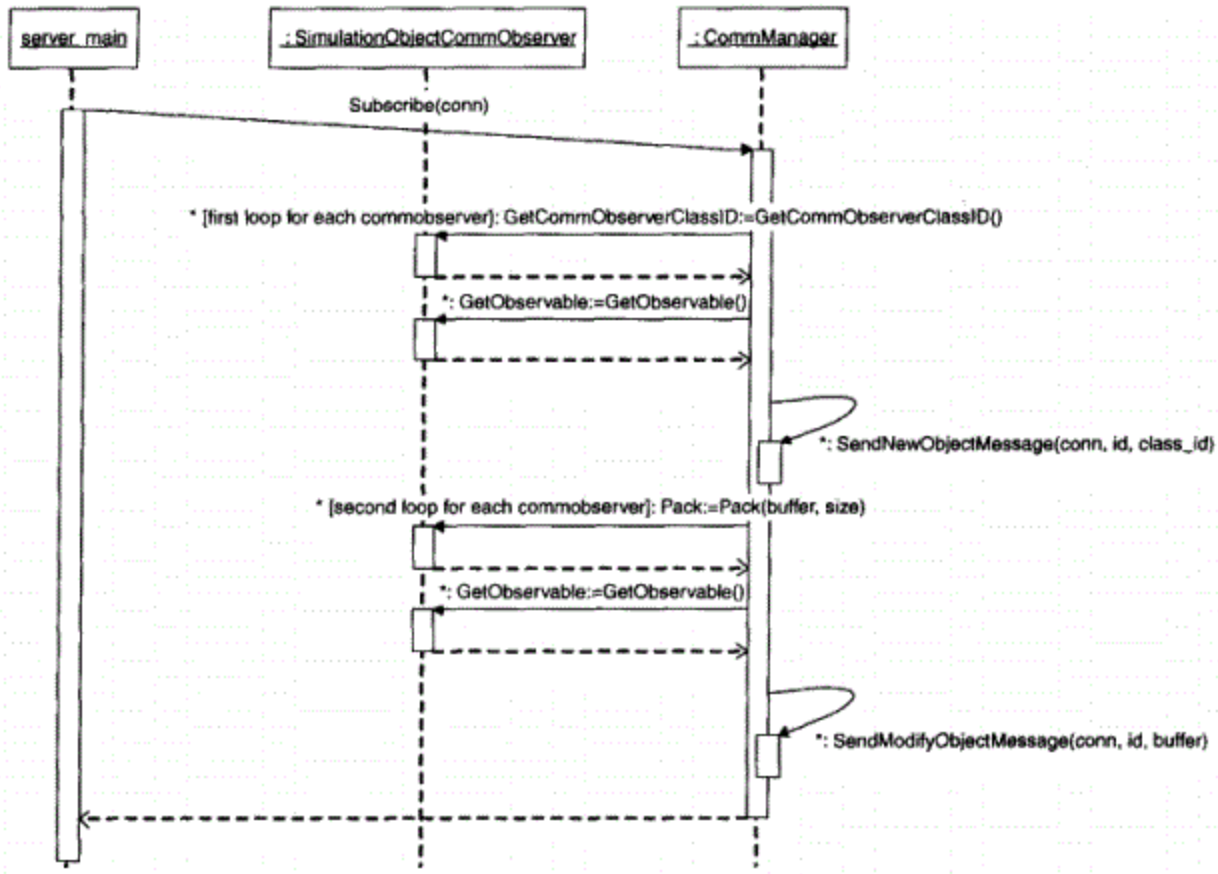


图 2-43　服务端创建新客户时的 UML 顺序图

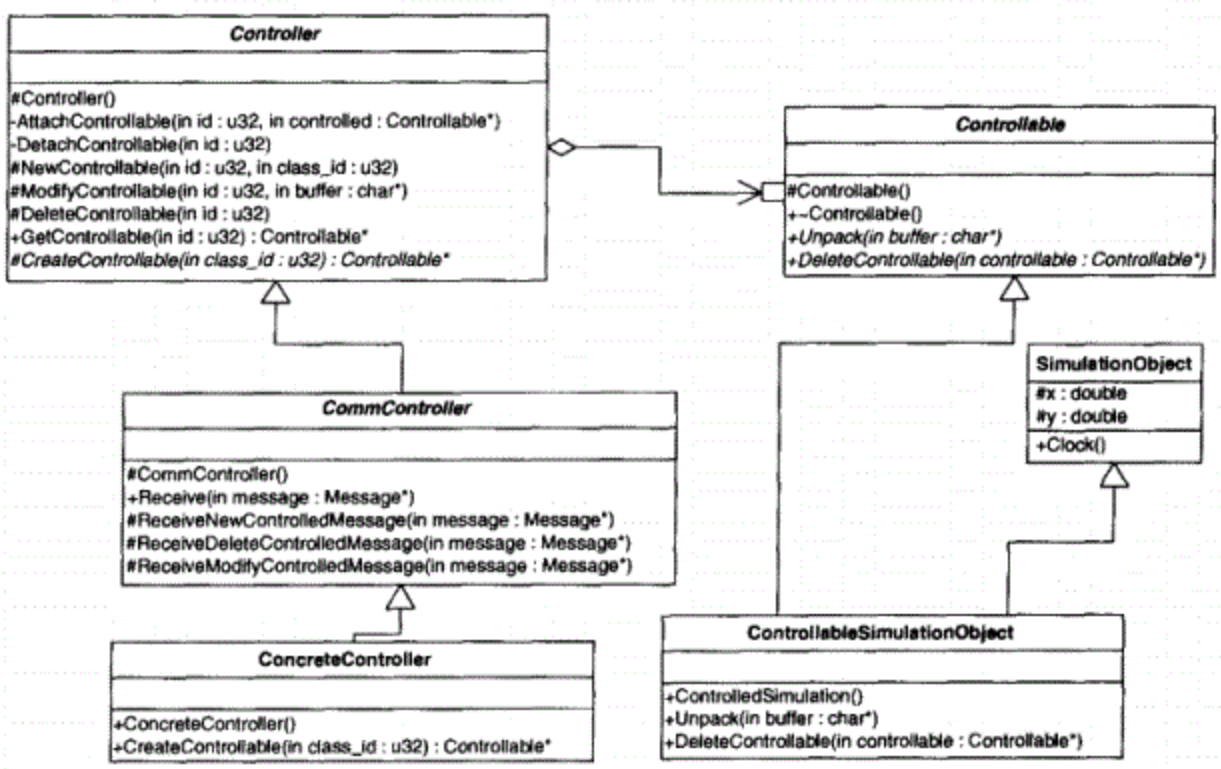


图 2-44　客户架构的 UML 类图

1. 创建消息（Creation Message）

当通讯控制器收到一个创建消息时，它必须创建正确的对象并且建立从接收到的标识到这个客户端新实例指针间的映射。控制器（Controller）类定义了控制器机制的基本行为，可以用来控制所有的仿真对象。通讯控制器（CommController）类定义了解释不同消息的特定方法。

当一个创建新对象消息到达客户端时，控制器会收到待创建对象的类标识，它必须能够根据这个给定的类标识创建一个相应的类。随后客户端程序应该创建一个具体控制器（ConcreteController）类，其行为继承自通讯控制器，因而可以定义抽象方法CreateControllable()。这个方法接受一个类标识并且返回一个新创建的仿真对象（见图 2-45）。

创建了这个对象后，相应的查找项就会被加入到映射中，这样后续的消息就可以被发送给正确的仿真对象实例了。

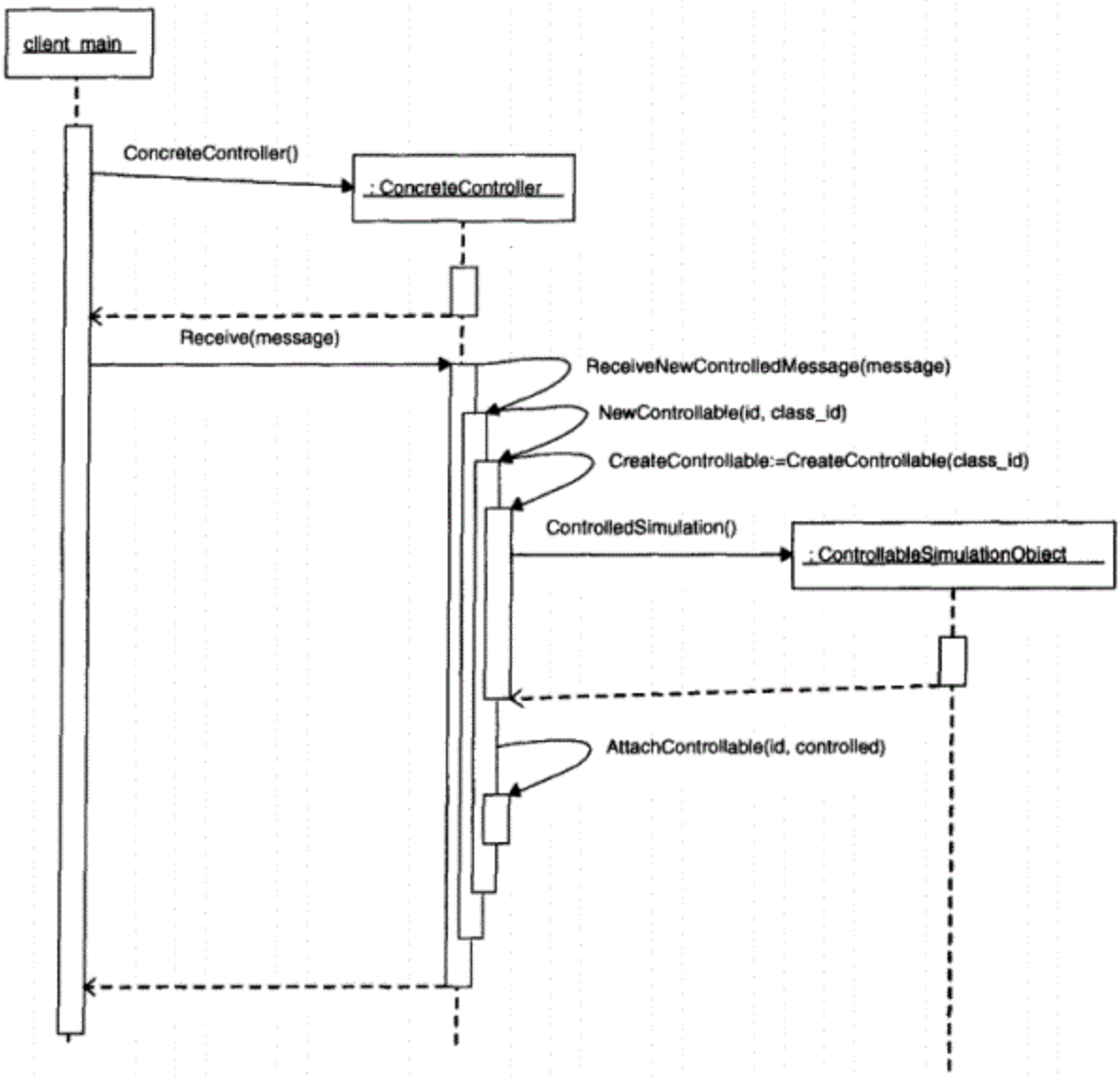


图 2-45 客户端创建对象时的 UML 顺序图

2. 更新消息（Update Message）

当通讯控制器收到一个更新消息时，它必须先确定应该由哪个仿真对象来接收这个更新。通讯控制器只需要通过调用 GetControllable()方法就能获得这个标识所映射到的对象指针。一旦控制器得到这个对象的引用，它就会把相应的更新缓存发送给这个对象。每个可控制仿真对象（ControllableSimulationObject）都知道应该怎样用自己的 Unpack()（解包）方法来处理这个缓存。

如果 Unpack()方法需要解出一个指向其他对象的指针，那么服务端应该已经把与这个对象对应的标识发送给了 Unpack()方法。可控制仿真对象可以简单地通过调用 GetControllable()来获得这个对象的引用，并把它赋予相应的内部状态。如果标识不存在，这个方法返回空（null）指针（见图 2-46）。

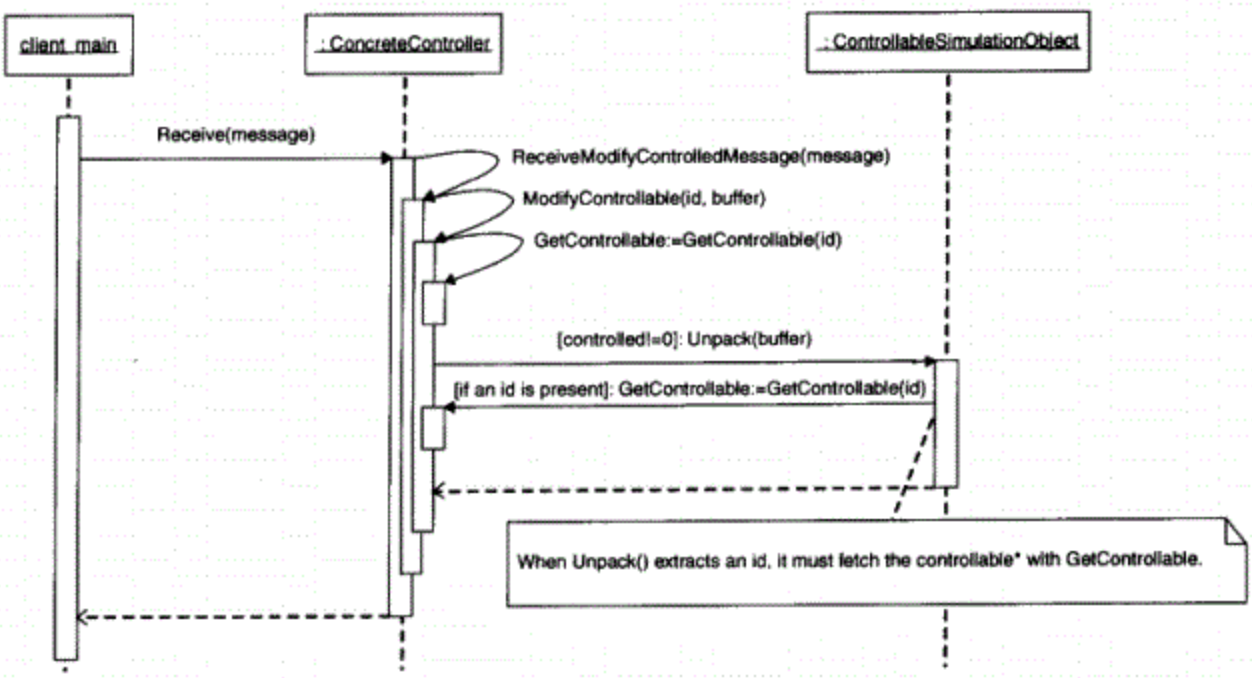


图 2-46　客户端更新对象时的 UML 序列图

3. 删除消息（Deletion Message）

当通讯控制器接收到一个删除对象消息时，它必须销毁相应的可控制仿真对象，并且删除所有对它的引用。对于那些持有已删除对象指针的对象的更新消息仍然会到达客户端，但显然游戏设计人员不希望在仿真中存在无效指针。最佳的处理方法是调用可控制所有仿真对象的 DeleteControllable()方法，这样就可以正确地删除任何对它的引用了。在客户端，可控制仿真对象的析构函数还应该销毁所有的图形观察者，如图 2-47 所示。

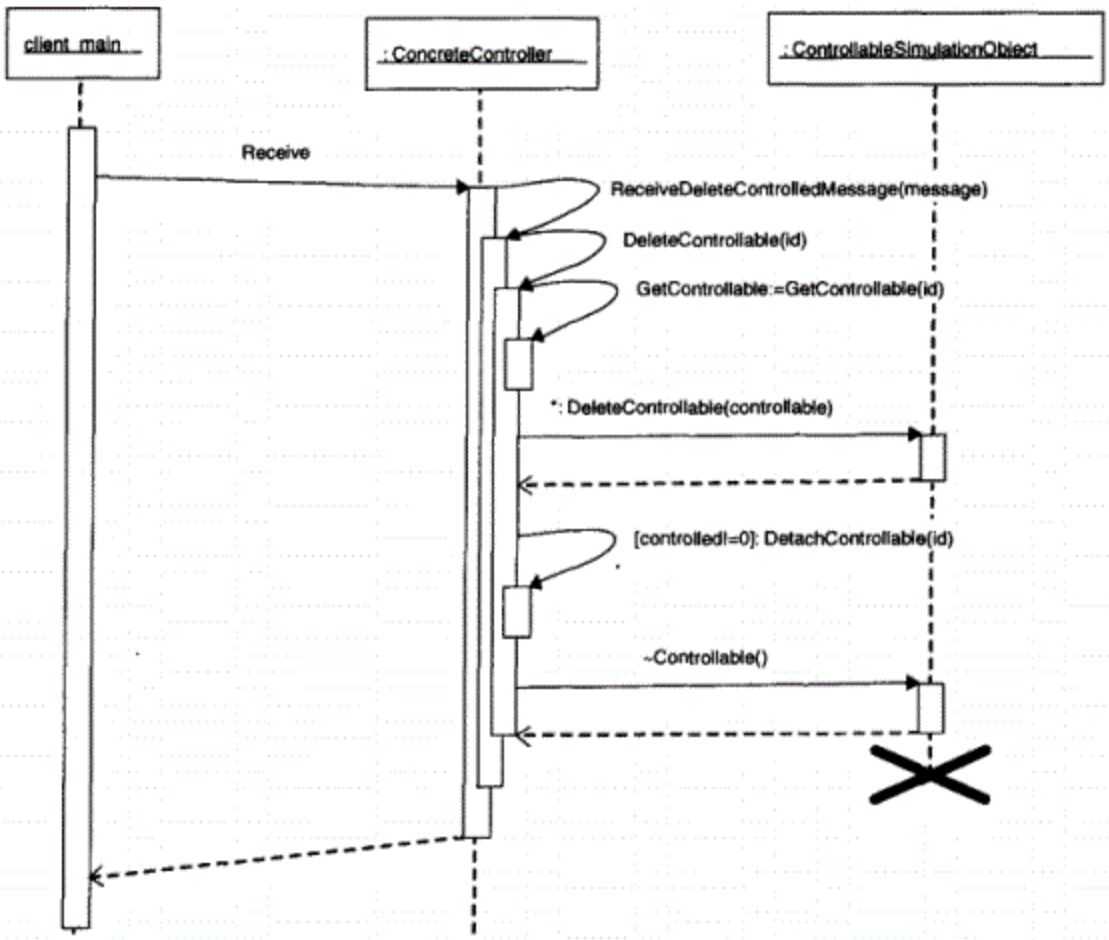


图 2-47 客户端删除对象时的 UML 顺序图

2.8.5 增强

下面的增强可以为上面描述的基本系统加入一些有用的功能。

1. 差别观察者（Difference Observer）

对基本的多人 O/O 模式进行的第一个增强是，加入一个功能使其只发送那些在上次更新后修改过的属性。因为系统只发送相关的数据，所以可以优化带宽的使用。未被改变的数据将不会通过因特网发送出去，因此带宽不会浪费。这个修改可以通过使用一个位图来实现，每个位代表仿真对象的一个属性。

本文必须提供一个新类：差别可观察（DifferenceObservable），它具有一个全新的 Update() 方法，这个方法接受属性的标识作为参数。当某个属性发生变化时，它必须用相应的标识来调用 Update()方法。从观察者的角度来看，差别可观察对象必须检查位图中的 1（这表示发生了修改），然后打包这些修改了的属性。游戏系统需要建立一种机制来告诉 Unpack()方法哪些属性被发送了——譬如通过发送位图（如果它不是很大的话）或是发送属性的索引。这些位图发出后，会被重置为 0。

通过引入一个通用的差别者（DifferenceObserver）来对其进行进一步的改进可以让游戏开发人员不必为每个仿真对象类型编写相应的仿真对象通讯差别观察者（SimulationObjectCommDifferenceObserver）类。它提供了一种属性登记机制（譬如说使用指针和数据长度）。这个改进也许可以为实现过程节省很多工作量。

2. 每个连接一个观察者

在实际模型中，每个可观察的仿真对象都只有一个通讯观察者。这确保了所有客户都能按照相同的顺序收到相同的更新消息，从而在他们的电脑上都运行着相同的仿真。

一个可行的优化是为每个连接都指定一个观察者。这可以更好地管理具有不同连接流量的客户的带宽使用。低速接收者可以让修改累积一段很长时间后再调用SendChanges()方法。高速接收者可以在每帧都接收改变。这个架构与前面提议的模型有所区别因此需要对它进行重新构思。另一个可行的优化是对类似于[Lopez02]中所描述的解耦（decoupled）O/O模式进行分析，这可以通过为所有观察者建立一个在独立线程中运行的公共队列来避免通知延迟。

3. 优先级和视野

正如[Sweeney99]中所讨论的，当不同的对象被赋予不同的优先级时才能更好地使用带宽。这样做的结果是一个对象与游戏的相关程度越高，分配给它的带宽也就越大。当前的UpdateChanges()会把每个更新了的对象发送给所有的客户。根据当前的带宽使用情况，我们可以把它改写为只发送那些相关度最高的改变。

另一个优化带宽的方法是仅发送客户视野范围内对象的更新。这需要设计额外的管理代码来处理对象进入视野（类似于在仿真中加入新客户）的情况。游戏系统可以通过实现一个Relevance()方法来检查某个特定对象是否在给定客户的视野中。这里的问题是如果视野改变非常迅速，那么重新发送和重新创建的开销会导致延迟。

2.8.6 总结

正如上文所介绍的，观察者/可观察者设计模式非常稳健，它适用于MMP游戏的基础架构。这个架构可能会变得非常复杂，但是这个引擎能够自动地处理所有的通讯和复制问题，因此有助于缩短游戏的设计和开发周期。这样游戏程序员就可以把注意力集中在游戏仿真上，而不必为通讯问题或是为客户和服务端开发独立的仿真模型操心。

观察者/可观察者设计模式为服务端的仿真模型提供了一个良好的客户端近似，它不断地发送对象更新，同时也允许客户端仿真对时钟信号进行处理。这样可以获得中心仿真真实的仿真代理。

2.8.7 参考文献

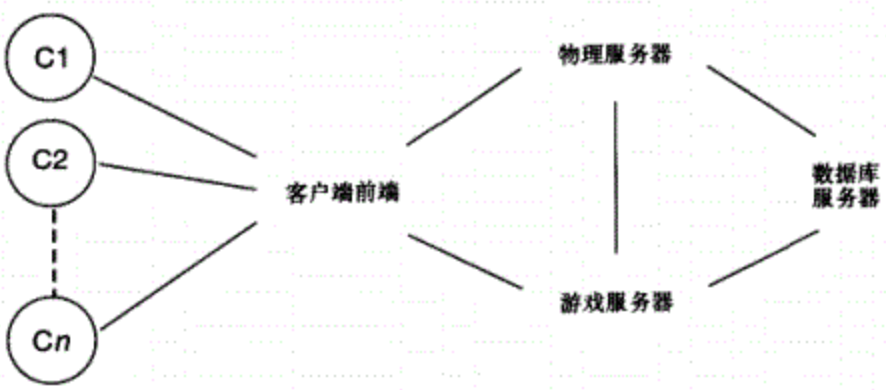
[Gamma95] Gamma, Eric, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Lopez02] Lopez，Albert，"How to decouple the Observer/Observable object model," *http://www.javaworld.com/javaworld/javatips/jw-javatip29.html*，August 2002.

[Sweeney99] Sweeney，Tim，"Unreal Networking Architecture," technical paper from Epic MegaGames，Inc.，July 21，1999.

第 3 章

服务端开发



3.1 无缝服务器：优点和缺点

Jason Beardsley, NCSoft Corporation
jbeardsley@ncaustin.com

目前的 MMP 游戏通常都支持成千上万的并发玩家。为了可以把所有与游戏相关的计算分散到多个进程和多个主机中去，游戏系统需要使用分布式服务器环境。要实现这样的分布式环境，一个常见的做法是把游戏世界分成由不同服务器进程管理的部分（piece）或区域（region）。

在这样的环境中，游戏设计人员可以根据在游戏中能否显式地观察到服务器进程的边界来把游戏世界本身分为无缝的（seamless）或是分区的（zoned）。在无缝世界中，玩家也许无法察觉到与他交互的对象其实是由多个游戏进程或服务器控制的，因为从客户的角度来说没有可察觉的区别。分区世界则由没有（或是很少）关联的独立地理区域组成，玩家只能和同一服务器上的其他对象进行直接的交互。

究竟要实现一个无缝世界还是一个分区世界是一个至关重要的决定，而且这个决定必须在游戏设计的早期就做出。这个选择对游戏的每个方面（从设计和程序开始，然后是美工制作和游戏世界构筑，最后贯穿游戏的实际运营和维护）会产生什么样的影响并不是显而易见的。本文的目的就是要让读者了解到这个决定的重要性。

3.1.1 杀死怪物不止一个方法

本文的一个基本假设是，为了达到预期的并发玩家数量，游戏设计人员必须按照地理分界线对游戏仿真进行分割并把它们分配给多个服务器。这的确是最普遍的解决方案，但显然不是分配服务器负载的惟一方法。问题在于，对于一个特定的游戏设计来说，是否存在其他方法可以在多个服务器间对处理进行划分呢？这个问题的答案依赖于游戏设计本身以及游戏世界中玩家的数量。这些因素决定了计算的类型和数量。在此本文介绍了几种可行的方法，一方面为了使分配方法更为完备，另一方面也便于进行对比。

1. 分割物理和游戏计算

现代 MMP 游戏通常建立在三维世界中，因此游戏开发人员会把大量时间花费在检验角色移动、进行碰撞检测以及三维空间中其他和物理相关的计算上。把三维移动/碰撞处理与游戏的其他部分分离开，并使用一个独立的服务器就可以对此进行处理。这样一来，“物理”服务器会处理移动和

碰撞，然后向“游戏”服务器发送更新。

图 3-1 所示的服务器架构就是基于这种设计的。客户连接到一个前端服务器，服务器会根据数据包的类型把它们重定向到游戏服务器或是物理服务器上。

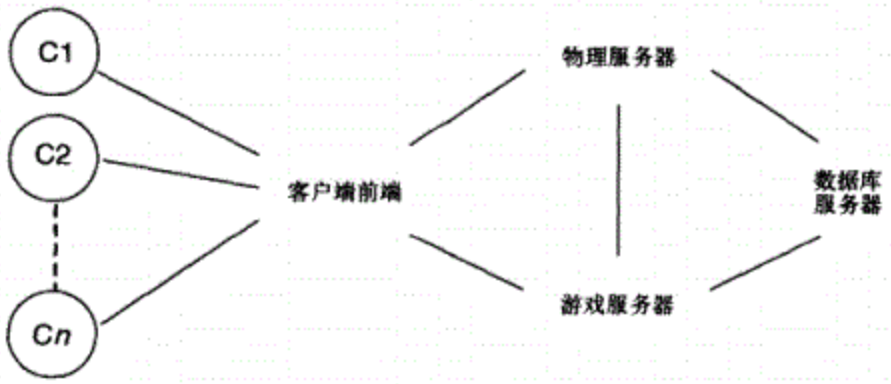


图 3-1　在服务端架构中使用分离的物理服务器和游戏服务器

最后，值得注意的是基于图块（tile-based）的二维游戏对物理计算能力的要求较低，因此在使用这种方式时伸缩性会更好。

2. 让 AI 独立运行

分配服务器负载的另一种方法是把人工智能（AI）处理分离到一个独立的服务器中，它本质上就像是所有 AI 进程的超级服务器。这个概念看上去很容易实现，但是存在一些缺陷。因为 AI 决策系统通常依赖于玩家角色及其他游戏对象的属性，因此需要对数据进行复制，这不仅会带来明显的同步问题，还会增加服务器间的通讯需求。考虑到现代局域网的可用带宽非常大，后者可能只是一个很小的不便之处。

代码复制则是一个更为重要的问题。通常 AI 和玩家会使用同样的游戏系统（例如战斗、魔法等），因此这部分代码可能需要同时存在于游戏主服务器和 AI 服务器中，但是这两部分很可能在形式上有一些细微的差别。如果不对这些代码进行谨慎的管理，它们很可能会不同步，这将导致难以发现的错误或反常的行为。

3. 花更多的钱

最后，游戏开发人员总是可以通过使用更强大的（并且更贵的）服务器这样的“暴力”方法来解决问题。这意味着使用多处理器的电脑以及编写多线程的游戏代码。编写多线程的游戏代码当然要比编写单线程的代码更为复杂，但是绝不会比在多个服务器间进行同样的操作更困难。

到了一定程度，那台用来支持大量（并且不断增长）玩家的电脑会贵得难以承受。譬如，一台需要 8 个处理器的服务器要比一台需要 4 个处理器的服务器贵一倍以上。最终，同时在线玩家的数量总会有一个上限，对不同的游戏来说，这个上限也不同。随着设计人员加入更多的游戏内容，这个上限只会变得更低。

4. 小结

毫无疑问，一些与地理无关的方法可以用来解决服务器负载分配问题，某些方法实现起来还非常方便。至于它们的伸缩性是否和对游戏世界进行空间划分的方案一样，就完全是另一个问题了。

本文认为把游戏世界分割为更小的部分可以使伸缩性更好，因为它把不同类型的游戏处理平均地分布到了不同的服务器上。同时，这还便于在不影响当前服务器负载的情况下添加更多的游戏内容（譬如说一块用于历险的新大陆）。

从另一个角度来看待这个问题也是成立的：这里所提到的各种方案和用于加速现代 CPU 的各种方法在某种程度上是一致的。第一个方案：把“物理”服务器和“游戏”服务器分离，这种方法早已为大部分 MMP 服务器架构所采用，它类似于处理器的流水线。第二个方案：独立的 AI 服务器，这大致和一个具有独立整数和浮点单元的超标量体系结构 CPU 相似。第 3 个方案：“暴力”方案，类似于提高处理器的时钟主频。对游戏世界进行地理上的分割类似于使用多个 CPU。如果可以对要处理的任务进行合理的分配（MMP 游戏世界正是这样），并行处理的可伸缩性远比单个处理器（无论多快）要高。

3.1.2 无缝世界模式的原型

在对无缝游戏的优缺点进行讨论之前，应该先为这样的系统提供一个“蓝图”，并且列举在无缝设计中有待解决的问题。这也是后续小节的基础。

通常的做法是先把游戏世界分成小块，然后在相邻的小块间加入一些服务器间的通讯以使每个服务器都可以自主地作出反应。要实现这点，可以让服务器互相通知那些与边界接近的对象，这样一来，每个服务器都可以通过它们的本地代理来对远程对象（在某种程度上）进行控制。从概念上来说，这很简单，然而在实践中却未必如此。

1. 分割游戏世界

第一个问题是怎样才能最好地划分游戏世界。这个问题往往依赖于特定的游戏。在一个由玩家控制游戏人物的基于地形（terrain-based）的游戏（也就是典型的在线 RPG）中，应该假设游戏世界处于一个二维平面上，并按照二维网格对其进行划分，这是最可行（也是最简单）的方法，并且也适用于三维游戏世界。对于那些真正的“6 度空间（six degrees of freedom）”仿真游戏来说，这个方法也许并不好，使用八叉树或其他方式来进行空间细分可能会更合适。

2. 平均分配

下一个需要考虑的问题是相邻区域的共享边界应该有多大。最小情况下，它们应该与客户端的游戏世界和/或玩家的可感知范围一样大。如果再小一点的话，当玩家在服务器边界附近时，他视野中的对象可能和边界另一边的其他玩家不同，这可能会引起很多对这一游戏特性的滥用。

图 3-2 描述了这种情况。服务器 A 和 B 共享同一个边界，边界区域中的对象会被映射到

另一端。玩家 P2 在共享边界中，而玩家 P1 在共享边界外。P1 的可见对象集合由 P1 周围的虚线表示，它包含了 P2。但是由于 P1 并没有映射到服务器 B 中，他对于 P2 来说是不可见的，即使 P2 和 P1 具有相同的可视范围。由于服务器 B 完全不知道 P1 的存在，因此无法把关于 P1 的信息告知 P2。所以 P1 也许可以攻击 P2，但 P2 却不能攻击 P1。

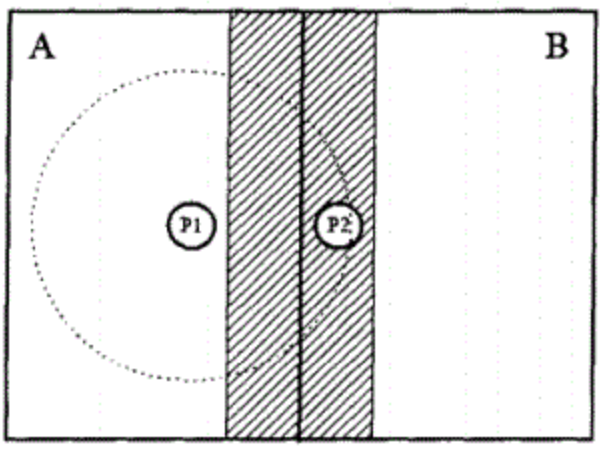


图 3-2 一个过小的共享边界区域

3. 详细的代理对象还是精确的代理对象（两者不可兼得）

边界区域中的对象可以跨越服务器边界进行交互。当一个对象进入边界区域时，它所在的服务器会通知相邻的服务器，后者会创建一个代理对象来表示远程对象。同样地，当这个对象从边界区域移动到服务器“内部”时，这个代理对象会被销毁。

另一个必须解答的问题是代理对象应该为游戏提供多少信息。如果信息太少，就很难在没有异步消息传递的情况下编写任何游戏代码；如果信息太多，就需要对更多的数据保持同步，这必然也会导致更多的数据得不到同步。一个实用的代理对象至少必须包含位置、方向、碰撞信息及对象类型等基本属性。此外，它还可以包含任何其他依赖于游戏且不会发生变化的属性（譬如说：比例、颜色等）。

为了减少服务器间为了更新代理对象而进行的通讯，可以为对象属性赋予优先级。也就是说，确定每一个需要映射的属性，然后确定应该以多快的速度对它进行更新。当一个低优先级的属性发生改变时，代理对象可以延迟更新（视环境而定，甚至可以永不更新）。这么做通常会使同步问题更加严重，但却可以在效率和精确度之间进行平衡。

4. 定义边界线

另一个需要考虑的技术细节是服务器的边界是柔性的还是刚性的。当一个对象越过刚性服务器边界时，它的所有权会立即转移给另一个服务器，没有任何延迟。而一个柔性边界在转移前只允许对象“略微跨越”边界。转移对象的代价可能很大，而柔性边界可以减少对象转移的数量，但是代码的编写会变得更为复杂。譬如说，如果有些代码（无论游戏开发人员尽多大努力去避免，总会有一些）需要对代理对象使用不同的方式进行处理，那就不能简单地用“是否在本服务器边界内？”这样的判断来区分本地对象和代理对象。

5. 静态的边界还是动态的边界

最后需要考虑的是在游戏运行过程中是否可以改变服务器的边界——也就是说，它们是静态的（不变的）还是动态的（可移动的）。要正确地设计和实现可以动态调整的服务器边界是非常复杂的。譬如说，游戏系统必须通过一个原子事务（原子事务的各个步骤不可分割，要么全部成功，要么全部取消）处理把一组对象整体转移到另一个服务器上，并且不能产生任何玩家可以察觉的延迟。如果边界和两个以上的服务器相关联（在实践中这必然会发生），那么边界的调整将涉及到 3 个或更多的服务器。

即使实现了这个机制，要正确判断应该在什么时候对服务器边界进行怎样的调整也是非常困难的。过于频繁的调整会带来很大的资源消耗。而过少移动边界则会导致服务器负担过重，尤其是出于某些原因大量的玩家同时在一个地点聚集时（譬如说开设了一个新的地下城）。在确定新的边界时，必须确保修改后的服务器负载是均衡的，从而降低在今后对此进行重复修改的可能性。服务器负载依赖于很多参数：玩家数量、AI 数量、游戏世界中有多少可碰撞对象等等。这些都需要进行大量的调整，才能得到正确的结果。

6. 小结

一张图片胜过千言万语，所以本文就用图 3-3 来结束这一小节。这张图片中有 3 个相交的服务器，在边界周围有很多对象，它们处于不同的位置。

阴影区域代表共享边界，其中的对象会映射到其他服务器中。交叉线阴影标出的区域必须被映射到两个相邻服务器中。对象 1、7、9 对于它们各自的服务器来说是严格本地的。对象 2、4、8 在相应的相邻服务器中有一个代理。对象 3、5、6 有两个远程代理。

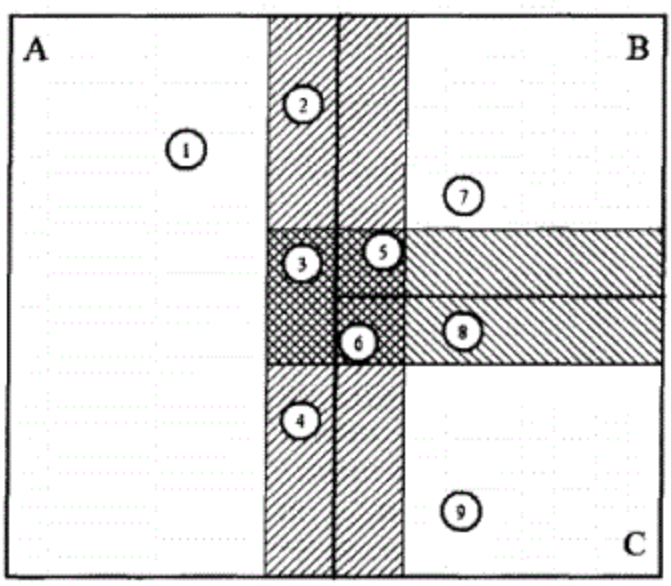


图 3-3 3 个服务器共享同一边界

3.1.3 无缝世界模式的优点

阅读了前面几个小节以后，读者可能会觉得设计并实现一个无缝的游戏世界从技术角度

来说显然是非常困难的。然而，的确有不少理由值得我们付出这样的努力。

1. 拥有更大的游戏空间

无缝世界在大小上具有明显优势。无缝游戏世界可以有更大的连续区域来进行游戏，而一个分区游戏世界所能支持的区域大小，受限于单个服务器所能处理的玩家数量。对玩家来说，无缝游戏世界可以营造出一个更加引人入胜的环境，并且游戏仿真也可以变得更为真实。对于某些游戏设计来说，把游戏世界分解为独立的区域甚至是不可行的，这时必须使用无缝游戏世界。

作为这种观点的对立面，一个更大的游戏世界与更好更有趣的游戏体验并没有必然的因果关系。一个游戏的娱乐价值与游戏世界的大小无关。事实上，很可能一个巨大的游戏世界中有很多地方是空的，这些区域可以用来分隔那些玩家时常出现的地方。

2. 可伸缩性和可靠性

无缝游戏世界可以给游戏带来一个更为实际的好处，那就是可以增加可伸缩性（scalability）。随着游戏世界中的玩家越来越多，服务器需要进行调整以处理不断增加的负载，更不用说那些能在运行时自行平衡的动态服务器边界了。即使服务器边界是静态的，也可以根据玩家的数量和分布，在停机时间对它们进行调整来达到同样的效果。

无缝世界还具有更高的可靠性。如果某个服务器中止服务或是某个服务器进程崩溃了，就可以调整服务器边界来把负载分散到其余的服务器中。一个更强大的动态服务器边界实现甚至可以在检测到某个服务器崩溃时自动地做到这点。

3. 不确定的边界线

游戏中有很多错误与在服务器间来回移动数据相关，动态服务器边界的另一个优点是它们可以降低玩家对这些错误进行滥用（exploit）的可能。理由很简单：玩家对于究竟哪里是服务器边界并没有一个确切的概念，当玩家发现这些边界后，只需要对服务器边界进行调整就可以避免他们对此滥用。

然而，有些玩家偶尔会变得出乎意料的幸运，因此上述方法并不是一个安全的解决方案。显而易见，潜在的错误仍然必须被找到并修正。

4. 不需要载入地图

无缝游戏世界甚至对玩家也是有好处的。“载入地图”所需要的时间会大大减少（除了初始化以外），这些载入操作会被平摊到游戏世界中每一个分块上，这些分块仅在需要时才会被载入。

3.1.4 无缝世界的缺点

现在上文已经列出了无缝游戏世界的优点，是时候看看它的缺点了。通常，最主要的问题是复杂度。和分区服务器设计相比，无缝服务器会使游戏开发和维护上的所有工作都变得更为困难。

1. 不确定法则：不仅仅是客户端

分区服务器和无缝服务器在复杂度上的区别类似于单人游戏和多人游戏之间的区别。在单人游戏中，只有一个游戏仿真系统，只有一个进程对它进行管理，也只有一个外部实体（玩家）去面对游戏状态的精确表示。在一个基于客户服务器的多人游戏中，对游戏世界来说，每个客户都具有它自己的视野，它们不仅不精确，所有客户的视野之间还会有些区别。游戏世界的规范状态保存在服务器上。换句话说，多人游戏必然会引入不确定性，而客户端必须对其进行处理。幸运的是，有很多众所周知的技术可以处理这样的不确定性（譬如说，航位推测法（dead reckoning））。

在一个无缝服务器环境中，由于表示远程对象的代理是不精确的，因此在服务端存在着相同类型的不确定性。游戏设计必须考虑到，代理对象不可能和它们所对应的真实对象完全同步，除非付出巨大的代价。这意味着游戏系统必须对很多很简单的玩家交互进行异步处理（譬如说，使用消息传递）实现以避免使用可能失效的数据。异步处理的本质导致我们需要解决大量的失败情况。不仅如此，代理对象缺乏同步是很多错误的根源，而这些错误可能会被玩家滥用。

2. 对设计的影响

设计一个在跨越服务器边界的情况下也能正常工作的游戏系统并不简单，会出现大量的竞争和失败的问题，要诊断并解决这些问题需要很高的专业技术。如果不能修正这些问题，游戏就无法正常运行。

譬如说，在分区游戏服务器中，玩家间的物品交易可以通过一个原子事务来实现（一旦这两个玩家认可这个交易）。但在无缝游戏世界中，两个玩家有可能位于不同的服务器中，因此需要一个多阶段的方法，还可能需要一个第三方的仲裁对象（或者直接由数据库进行）。不仅如此，某个服务器随时有可能在进行交易操作时崩溃，因此每个阶段都必须能够从各种可能的故障中恢复。

在设计可以跨越服务器工作的游戏系统时，还必须对代理对象的不完整性（incompleteness）进行处理。当对对象的某些处理依赖于未被映射的属性时，就必须重新考虑这个系统的设计。假设某种武器在某个特定的方向上可以对角色造成更大的伤害，但如果代理对象并不能反映角色的方向，那么这个武器在攻击一个远程角色时就会发生错误。

实践中，几乎每个游戏系统（移动、战斗、角色升级以及制造）在无缝世界中都会变得更为复杂并且更容易发生错误。这会直接导致开发周期延长或是使游戏变得更不稳定，更可能是两者都有。

3. 例子：给一个物品

既然对于设计复杂这一点怎么强调都不为过，下面本文将在这里详细介绍一个例子。这里要实现的游戏机制很简单：把物品交给另一个玩家，不是双向的交易，仅仅是转移一个对象。而且，为了减少一些客户服务器之间的信息往返，这里假设接受物品的玩家不能拒绝这个物品。这应该是很简单的。

在一个分区游戏世界中，实现这一特性的所有代码可能会象下面这样。

```
def Give(srcPlayer, trgPlayer, itemId):
    if not srcPlayer.HasItem(itemId):
        WARN("illegal give: src=%s trg=%s item=%s",
            srcPlayer, trgPlayer, itemId)
        srcPlayer.SendMessage(GIVE_FAILED, itemId)
    elif not srcPlayer.CanRemoveItem(itemId):
        # perhaps it is cursed, or non-tradeable
        srcPlayer.SendMessage(ITEM_NOT_GIVEABLE, itemId)
    elif not trgPlayer.CanAddItem(itemId):
        # inventory full? already got one?
        srcPlayer.SendMessage(GIVE_FAILED, itemId)
    else:
        # do the transfer (and update the clients)
        srcPlayer.RemoveItem(itemId)
        trgPlayer.AddItem(itemId)
        # make it persistent
        srcPlayer.SaveToDatabase()
        trgPlayer.SaveToDatabase()
```

这段代码可读性不错，理解起来也很方便。它分为3个大致的阶段。

1. 给予物品的玩家调用 HasItem()（是否有物品）和 CanRemoveItem()（是否可以删除物品）进行判断。

2. 接受物品的玩家调用 CanAddItem()（是否可以添加物品）进行判断。

3. 提交操作。

在无缝服务器中，不能使用等价的代码来实现同样的功能，因为给予和接受物品的玩家可能在不同的服务器进程中。为了简化这个任务，游戏系统要求，对象无论在哪个服务器上，工作人员都可以向它发送消息，即使接受物品的对象正处于在服务器间转移的过程中，消息最终也会被送达。并且，如果一个消息因为任何原因（包括服务器崩溃或是对象不在游戏中）无法送达，就会返回给发送方一个错误提示。

因此，这个交易不是在两个服务器之间发生的，而是在两个玩家之间发生的，无论他们当时处于哪个服务器上（即使是同一个服务器）都一样。为了不过多地陷入细节（也就是代码），下面给出了大致的事件序列，假定给予物品的玩家是P1，接受物品的玩家是P2。

1. 服务器收到一个从P1客户端发出的Give（给予物品）请求。

   a) 由于P1是给予物品的玩家，对他进行HasItem()和CanRemoveItem()检查。如果检查失败，进行与前面相同的错误处理并且立即返回。

   b) 否则，对物品加锁以防止P1对它进行任何其他操作，譬如把它给另一个人或是扔在地上。

      i. 这个加锁状态是非持久的，因此不会被保存在数据库中，也不会在P1移动到其他服务器后继续保持，如果P1移动到另一个服务器或是退出游戏，必须发送一个GiveCancel（取消给予）消息给P2（参见第8步）。

   c) 发送一个GiveRequest（请求给予）消息给P2。

   d) 如果GiveRequest消息发送失败，就对物品解锁，并且发送GIVE_FAILED（给予失败）消息给P1的客户端，然后停止。

2. P2 收到 P1 发出的 GiveRequest 消息（在步骤 1c 中发送）。
   a) 由于 P2 是接受物品的玩家，对他进行 CanAddItem()检查。
      i. 因为这个物品本身不一定是一个本地对象，GiveRequest 消息必须包含足够的信息以符合检查的需要。在最极端的情况下，必须在 P2 所在的服务器上实例化一个该物品的临时拷贝。
   b) 把 CanAddItem()的结果在一个 GiveCanAddItem 消息中发送给 P1。
      i. 游戏服务端希望把交易中状态的数量最小化并且希望到最后一步才提交。所以，它并不在 P2 的背囊中为这个物品“保留”一个位置。因此，它可以不去处理 GiveCanAddItem 消息的发送失败。
3. P1 收到一个由 P2 所在的服务器发出的 GiveCanAddItem 消息（在步骤 2b 中发出）。
   a) 如果结果是 false（也就是说不能把这个物品给 P2），解锁这个物品，向 P1 的客户端发送 GIVE_FAILED 消息，然后停止。
   b) 否则，把这个物品从 P1 的背囊中暂时删除并把它放在一边。此时并不对 P1 的客户端和数据库进行更新。
   c) 向 P2 发送一个 GiveAddItem 消息。这个消息表示从 P1 到 P2 的所有权转移。
   d) 如果 GiveAddItem 消息发送失败，把物品放回 P1 的背囊，向 P1 的客户端发送 GIVE_FAILED 消息，然后停止。
4. P2 收到 P1 发出的 GiveAddItem 消息（在步骤 3c 中发出）。
   a) 对 P2 检查 CanAddItem()是否仍然成功。毕竟在此期间 P2 可能改变了状态。如果检查失败，向 P1 发送 GiveFailed 消息。
      i. 这次仍然不必担心这个消息会发送失败，因为 P2 的状态并没有改变。
   b) 否则，把这个物品暂时加入 P2 的背囊。不要更新 P2 的客户端和数据库。
   c) 向 P1 发送 GiveAddedItem 消息。
   d) 如果这个 GiveAddedItem 消息发送失败，把物品从 P2 的背囊中删除，然后停止。
      i. 在这种情况下，P1 没有确认这个转让，因此必须撤销对 P2 背囊的暂时修改。
5. P1 收到 P2 发出的 GiveFailed 消息（在步骤 4a 中发出）。
   a) 把这个物品放回 P1 的背囊，向 P1 的客户端发送 GIVE_FAILED 消息，然后停止。
6. P1 收到 P2 发出的 GiveAddedItem 消息（在步骤 4c 中发出）。
   a) 最终，转让成功。到目前为止，P2 只不过暂时获得这个物品。现在把这个物品从 P1 的背囊中“永久”删除，并且更新 P1 的客户端和数据库。
   b) 然后，向 P2 发送 GiveAddedItemAck 消息。
   c) 如果最后一个消息发送失败，必须撤销在 6a 中所作的修改，也就是说，把物品放回 P1 的背囊（在必要情况下可以违反任何游戏规则），通知 P1 的客户端，然后更新数据库。
7. P2 收到 P1 发出的 GiveAddedItemAck 消息（在步骤 6b 中发出）。
   a) 直到现在，这次物品转让才真正完成了。更新 P2 的背囊，把这个改变保存到数据库中，通知 P2 的客户端。
8. P2 收到 P1 发出的 GiveCancel 消息（在任何时候）。
   a) 这意味着 P1 退出游戏或是在服务器间移动了。根据在这个过程中所处的阶段，

需要进行不同的恢复步骤。然而，服务端可以保证在第 7 步以后不会再收到这个消息，因此对于 P2 的任何状态改变都可以持久化。

正如我们所看到的，在分区世界中只需要十几行代码就可以实现的一个简单机制在无缝世界中就扩展为一个多段事务（multiphase transaction）。（上面的过程很可能不能正确处理某些竞争情况。毕竟，它还需要在真实情况中进行测试）。想象一下，如果采用这种类型的环境，游戏系统将会变得多么复杂。

4. 构筑游戏世界

“构筑游戏世界（world building）”就是创造游戏发生的物理世界的过程。构筑游戏世界的任务包括地形建模、贴图绘制和对象放置等。显而易见，建造一个无缝世界比建造一个分区世界要困难很多。刚开始的时候，整个世界可能太大了，以至于无法把它作为一个整体进行编辑，因此，游戏编辑人员需要能够在更小的分块上工作，再加上多个游戏世界构筑人员需要同时进行工作，显然他们需要使用一个基于分块的版本控制系统来管理对游戏世界的并发访问。

游戏编辑人员需要花费额外的时间来“整理”那些由不同人员编辑的区域边界。在放置大型对象时需要格外小心，因为它们可能会跨越服务器边界而导致问题的产生。如果假设在同一建筑物中的对象应该在同一服务器上（可能是为了有效地进行裁减），那么和潜在服务器边界相交的建筑物必须改变位置。

与之相反，在分区游戏世界中，编辑人员可以在一个既没有边界区域也没有放置限制的连续区域中工作。虽然这仍然可能需要某种形式的锁定方式，但是它可以在一个更高的粒度（譬如说地图文件）上进行，而不是在整个游戏世界的任意子集上进行。

5. 美工

令人难以置信的是，在无缝世界中就连美工制作也会变得更为复杂。创建像建筑物那样的大型对象会受到服务器边界区域大小的影响；任何对象的最大尺寸必须小于共享边界的大小；否则，这样的对象可能会由于跨越边界区域而导致问题。

图 3-4 中，一个大型对象位于服务器 A 中，并且在共享边界中。服务器 B 上的玩家 P 可能看不到这个对象（因为 P 在共享边界之外，而这里共享边界的宽度恰巧和玩家的感知半径相同），然而很明显它应该是可见的。更糟糕的是，P 可能和它发生碰撞，这不仅会给用户带来不必要的麻烦，还会给运营团队带来很多莫名其妙的错误报告。

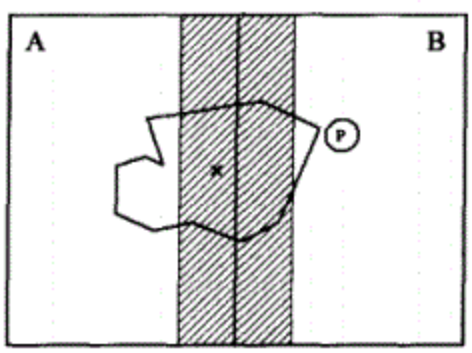


图 3-4　服务器边界区域中过大的对象

6. 运营和扩展

一个 MMP 游戏是永远不会结束的，因此这种情况会继续下去。玩家在支付了每月的订阅费用后，他们就想得到具有更多特性的扩展包、更多可以探索的地点以及更多可做的事情。因此，上述所有和开发相关的问题即使在产品发售以后也会出现。即使已经有了一个良好的基础，要在一个无缝游戏世界中创造更多的内容仍然需要游戏开发人员更多的努力。此外，如果游戏运营团队中的人员不是出自原始的开发团队，他们就可能会忘记无缝服务器带来的大量问题和限制，而这会给运营团队带来很大的麻烦。

不仅如此，大量对游戏的侵入行为会在服务器边界上发生。即使在分区世界中，物品复制也是一个比较常见的问题。然而，在服务器边界上更容易进行物品复制的根本原因是竞争条件。代理和相应的对象不能保持完全同步会导致大量的“失效状态（stale state）”的错误。发现并且修复这样的错误将是运营团队的主要职责之一。

继续前面提到的“给予物品”例子，假设在未来某个时刻，游戏开发人员需要对某些额外的条件进行检查。这些检查会封装在 CanGiveItemTo 和 CanTakeItemFrom 函数中，它们会使用游戏中的其他系统来决定这个转让是否合法（譬如说，如果玩家所在行会正向另一个行会宣战，那么把物品给那个行会的成员是禁止的。）。对分区世界中的代码进行修改非常方便，只需要加入几行很容易测试的代码就可以了。然而，对无缝服务器的实现进行更新不仅更加费时，而且很可能不正确。不妨想想，如果这些新函数不能工作于代理对象上，那游戏的运行会陷入多么困难的局面。

7. 对额外的复杂度处理失当

为了减少设计一个可以跨越服务器工作的游戏系统所带来的额外工作，有人可能会试图使用一些简单的解决方法，譬如说禁止某些特定的动作跨越服务器边界。譬如说，只支持在同一服务器上的玩家之间进行物品交易。这个主意听起来好像不错。

然而，从长期来看，这种方法不可能奏效。这种方法不仅以一种明显的方式把服务器边界告诉了玩家，还会为更多的游戏侵入行为打开大门。假设战斗是这样一个“迟钝的（lobotomized）”游戏系统。现在，如果某个玩家正被一个非常凶恶的怪物追赶，他很可能会反复穿越服务器边界（这就可以避免受到怪物的攻击），治愈自己，然后重新投入战斗。

逻辑上的重复是另一个需要避免的陷阱。很明显，当两个参与者位于相同的服务器时，进行“给予物品”的行动会更为快捷。因此，游戏代码可以检查这个条件并调用原先的基于分区的代码，而不是开始一个消息传递过程。这个建议应该在读者的脑海中敲响一个警钟：这导致两段代码（理论上）实现了同样的功能。随着时间的推移，这两段代码不一致的可能性会很大，如果最终它们不一致，那麻烦就大了。

简而言之，在处理服务器边界的复杂度时并没有什么神奇的方法。如果服务器边界存在于游戏设计之中，那么每个游戏系统都必须对它们进行处理。

3.1.5 总结

本文介绍了在 MMP 游戏中实现无缝服务器的动机以及这个方法的优缺点。它还解释了

为什么“无缝还是分区”并不是一个简单的决定，因为这个决定从开始设计时就会影响游戏的每个部分，并且持续到游戏发售之后。

设计人员可能会有这样的疑问：既然无缝服务器会带来那么多的麻烦，为什么还有人想使用它呢？即使不考虑那些难以处理的游戏设计需求，设计人员也应该选择分区世界。事实上，作者的看法是并不值得为无缝服务器付出那么大的努力。它不仅会导致开发时间变长，还会增加每一行代码的复杂度，并且限制了游戏中可以做的事情。花费在为无缝服务器边界制定规则上的时间可以用来把游戏系统变得更为有趣和复杂，创建一个细节更加丰富的游戏世界、制作更多的贴图/模型/动画或是更早地发行游戏。简而言之，那些时间可以被用来在更短的时间里制作出一个更好的游戏。

然而，就像塞壬（Siren，希腊神话中的女海妖，她们用歌声诱惑海员，从而使船只在岛屿周围触礁沉没）的歌声一样，无论从游戏模式角度还是从技术角度出发，人们都很难抵抗一个无缝世界的诱惑。对那些对此感兴趣的服务端程序员来说，这还是一个非常有趣的研究项目。程序员们可以去实现一个无缝世界，但惟一要小心的就是那些陷阱。

最后，是否使用无缝服务器是在设计一个 MMP 游戏时必须考虑的关键问题之一（如果不是最关键的话）。谨慎地做这个决定，并且尽早做出决定。

3.2 服务端对象的更新频率

John M. Olsen，Microsoft
infix@xmission.com

在考虑这篇文章是否有用时，可能有人会对自己说："从服务端发送大量的数据有什么大不了的？现在不是都有宽带（broadband）了吗？"在任何 MMP 游戏的预算中，带宽都是一项重要支出，它可以花去订阅费用中的很大一部分。在本文写作时，带宽的批发价格通常在 5 美元/GB 左右。只需要把数以万计（如果幸运的话，可能是数以十万计）并发玩家的数据需求累计起来，就可以知道为什么对发送数据量进行管理是那么重要了。

本文会从一些基本原理出发来指出那些在决策过程中需要用到的关键信息，随后介绍一个裁减和组织数据包的方法以对网络进行有效的运用。

3.2.1 视觉连贯性与精确度

所有的实时在线游戏都会在某种程度上和延迟问题作斗争，它们努力地寻求各种方法来对游戏状态进行精确而可信的描述。游戏开发人员的目标是要交付一个视觉上和感觉上都很好的游戏，即使以牺牲一小部分视觉精确度为代价。

如果玩家可以看到一个在视觉上连贯一致的游戏状态，并且其他玩家角色（Player Character，PC）和由计算机控制的非玩家角色（Non-Player Character，NPC）都以一个可信的方式移动，那么他们就不会察觉到存在的任何微小的延迟错误。

如果另一个玩家或是非玩家角色从一个位置跳跃到另一个位置，玩家会很快察觉到。这通常是由网络包丢失（网络以及硬件限制）或是数据发送频率过低（实现问题）造成的延迟引起的。此外，开发人员应该知道一致的延迟通常比变化的延迟更加容易补偿。

有些游戏会以自适应的方式根据延迟自行调整，它们会对位置进行平滑的调整以把对象带到预期的地方。根据需要修正的误差量的不同，这可能会引起一个难以察觉的调整，也可能会导致高速的移动或是怪异的滑行。

在对那些用于寻找合适数据更新频率的方法进行评价时，本文的指导方针是"如果玩家发现不了，那就不要紧"。

3.2.2 需要发送哪些数据

服务端需要在玩家和服务器之间发送多种类型的数据，每一种都需要根据它们是否需要及时传递来采取不同的处理方法。

1. 环境变化

环境变化具有非常高的优先级。如果玩家眼中的游戏世界和服务端的游戏世界不一致，就会造成大量的碰撞和同步问题。譬如当玩家本地信息和服务端信息不同时，他会惊讶为什么他总会错过一个移动的平台。由于环境变化非常重要，游戏开发人员需要花费大量的精力来减少对这类数据进行的传输。

环境中的活动地形元素（active terrain element）只是数据流中一个非常小的部分。这些数据应该包含正被打开或是关闭的门以及像旋转的风车之类的循环移动，游戏的设计人员希望所有玩家和它们的碰撞都可以与它们的视觉表示一致。如果一个玩家从风车的翼板上笔直穿过而另一个玩家却撞上了两翼之间的空档，那看上去一定很傻。

2. 玩家之间以及玩家和 NPC 之间的交互

由玩家或 NPC 触发的交互包括战斗、交易以及各种形式的聊天。在大多数情况下，这在数据流中占据的比重并不大。交易和聊天并不需要非常及时，服务端可以很方便地在玩家不会觉察到的情况下把它们延迟一小段时间。另一方面，战斗信息至少需要和任何环境变化一样及时。如果游戏支持通过服务器中转的语音传输，那就必须给语音数据设置一个很高的优先级，而且它还会占据带宽的很大一部分。

3. PC 和 NPC 的移动

最重要的是，服务端必须以一个足够高的频率来发送 PC 和 NPC 的移动信息，这样才能使玩家在视觉上感觉是连续的，但是也不能过于频繁，否则我们的网络就会陷入困境。

3.2.3 带宽限制

目前市场上的游戏对带宽的用量都差不多，大约在每个玩家 1KB/s 左右。有两个原因使得这个数据传输率成为一个有用的基准。首先，它是在调制解调器允许的范围之内的（modem-friendly），因为即使是 28.8KB 的低速调制解调器也可以维持这个数据传输率。其次，这对于预算来说也是一个很好的数字。即使某些玩家在一个月里全天挂在游戏上游戏开发人员仍然可以从他们的月费中获得一些剩余资金来支付硬件、办公室以及雇员等开销。

1. 更新频率

必须注意更新频率会影响总体数据的大小。在网络上发送的每个数据包都有一个不同大小的附加头部，像 UDP（没有确认的数据包）的头部大小是 28 个字节，而 TCP（具有确认

的）数据包在满载时头部大小是 72 字节。

更新频率越高就意味着这个额外头部数据的发送频率也越高。理想情况下，服务端应该把需要发送给某个特定玩家的所有信息打包在一起作为一个网络数据包，一下子发送出去，即使它包含的是玩家所需要的各种不同类型的信息。如果游戏服务端在服务器和玩家之间同时使用 TCP 和 UDP 数据包，就需要把这些信息分成两部分独立发送，但是它应该可以把所有的 TCP 数据合并成一个数据包，把所有的 UDP 数据合并为另一个数据包。

如果对数据进行打包发送的频率过高，数据包头部会消耗不必要的额外带宽。这样一来，当游戏中的行为发生变化时，玩家更容易发现延迟。网络传输层可能会自动把小的数据包合并起来，但是服务端无法控制它们进行这种合并的时机和方式。

如果发送更新的频率过低，随着服务端降低向玩家更新位置的频率，PC 和 NPC 在视觉上的跳跃和偏差也会加重。发送数据的频率过低还会增大数据包，这是因为网络会把它们打碎并在另一端重新组合起来。虽然这一切都是由网络层透明处理的，这些大的数据包也更容易受到丢包的影响，因为消息任何部分的丢失都意味着必须丢弃整个消息并且重新发送。

2. 分区（Zone）的地形和连续的地形（Continuous Terrain）

从网络的角度来看，这两种地形模式具有相同的需求。无论使用哪种模式，游戏系统都需要以某个方式来向玩家更新游戏世界的状态，并且把重点放在那些在玩家附近发生的事情上。分区地形系统会自动为最远范围赋予一个上限，而在连续地形系统中，游戏设计人员则需要使用额外的代码来决定界限。

3.2.4 每个用户在服务端需要的数据

在为整理数据以及调整数据优先级定义一个良好的方案之前，必须先确认在服务端要为每个活动玩家保存哪些信息。虽然每个游戏在具体实现时可能会有很大的差别，但是基本上每个玩家在服务器上都应该有一个发送队列。

这个发送队列包含了那些需要从服务端发送给玩家的消息。服务端表示每个玩家的数据结构中还应该包含一个列表，其中包含了最近几次发送更新时玩家的位置。这个位置列表的长度应该和发送队列相同，稍后本文将对此进行详细讨论。

3.2.5 管理数据大小

当角色数量增加时，保存角色数据所需要的内存可能会成指数级增长，游戏管理人员必须避免这种情况的发生。在 n 个玩家之间可能的关系数量是 n^2，因此游戏管理人员需要尽可能地在可接受的范围内把 n 减到最小。

理想情况下，游戏开发人员应该建立某种机制来把游戏环境划分成区域，在这些区域中管理人员可以很方便地知道在某个特定玩家的视野中有哪些玩家。在一个基于门户（portal）的系统中，管理人员应该确认对于每个门户来说，哪些门户是重要的，如果另一个门户的距离它很近，那就需要知道那个区域中的 PC 和 NPC 信息。

如果不使用这种机制，游戏管理人员就需要进行周期性的范围判断并且把每个玩家能够看到的其他 PC、NPC 和环境对象保存下来，不过这种方案会占用大量的内存和 CPU 时间。

3.2.6　更新队列

每个玩家的发送队列都是由一系列时间槽（time slot）组成的，如表 3-1 所示，这个列表的头表示的是马上要被发送出去的数据。在对某个玩家进行更新时，服务端会把第一个槽中的所有数据更新拼接为尽可能少的网络包后发送给他。从服务端的角度来看，这意味着每当槽中包含了任何网络包，服务端就会马上对玩家进行更新。这与客户端的屏幕刷新率无关，事实上它比屏幕刷新率低多了。

本文使用这个数据队列来向玩家发送所有的更新，因此它会包含移动数据、聊天消息、商业信息以及所有其他从服务端到客户端的数据传输。

表 3-1　　发送数据前的数据队列

槽 1	槽 2	槽 3	槽 4	…	槽 n
更新 A	更新 C	更新 D	更新 F		更新 G
更新 B		更新 E			

在发送槽 1 中的每个更新时，服务端会判断何时需要再次发送这个更新。如果这是一个循环更新（譬如说玩家位置），服务端会把它移动到其他槽中，这样它很快就会被再次发送。如果这是一个一次性更新（譬如说聊天消息），服务端在发送以后就会丢弃它。一旦完成对这个槽的处理，服务端就把这个空闲槽移动到列表尾部，并且把其他槽向前移动一个位置，如表 3-2 所示：

表 3-2　　发送数据后的数据队列

槽 2	槽 3	槽 4	…	槽 n	槽 1
更新 C	更新 D	更新 F		更新 G	
更新 A	更新 E				

注意在表 3-2 中，更新 A 被移动到槽 2 中去了，它会被立即重新发送（这可能是对对手位置的更新），而更新 B 被删除了（这可能是一个聊天消息）。槽 2 现在是队列的头。有时服务端会故意推迟发送数据，因此游戏管理人员需要特别注意收集这些数据的时机和方式。服务端通过特定的更新消息来指定需要发送的数据（譬如说某个玩家的位置），但是直到发送时它才会去获取实际的信息（譬如说这个位置是 X、Y、Z），这么做可以使它当前发送的信息总是最新的。

游戏管理人员应该对每个槽中所包含的更新数量作柔性的限制。如果有一个槽中装满了更新，就意味着已经接近预期的最大带宽限制，因此应该有选择地推迟几个更新。如果管理人员想要放置更新的那个槽已经满了，就应该向后查找第一个还有空间的槽。这种方法只适用于那些次序并不重要的更新（譬如说位置更新）。文本信息的次序可不能被颠倒。

有些数据在发送时不能有任何延迟，还有一些则需要确保正确的发送次序。这时，服务

端要么在这个槽中放入更多的更新，要么把这个槽尾部的另一个更新移动到下一个槽的头部。如果下一个槽也满了，就会引起连锁反应。如果这种溢出情况经常发生，就意味着网络的槽太小了，或者是发送数据的频率太高了。游戏开发人员需要测量整个网络的吞吐量，从而确定问题出在哪里。

3.2.7 缺省的更新频率

应该怎样决定某个信息的发送频率呢？应该怎样决定对于特定的玩家来说，在每次决定某个 NPC 的位置更新时应该向后移动几个槽呢？这些问题不仅决定了玩家可以用多高的频率获得 NPC 的更新，同时也决定了对于玩家来说，这个 NPC 的行动有多流畅。

通常，服务端希望对近距离的物体进行频繁的更新，也就是在每次发送更新数据时都会包含这些物体的信息，这很可能是每秒一次甚至是两次。远处的物体可以更新得慢一点。当然，总是存在某些情况使这个一般规律需要被改变，本文会在稍后对此进行讨论。

3.2.8 计算范围

精确的范围计算需要大量的 CPU 时间。为得到两点间的距离通常需要对它们的正交分量间距离的平方和计算平方根，如等式 3-1 所示：

$$dist = \sqrt{dx^2 + dy^2 + dz^2} \qquad \text{（等式 3-1）}$$

如果不想花那么多的时间，就必须寻找一条捷径，回到这篇文章最早的主题之一：如果玩家不能发现，那就不要紧。实际上玩家永远都不会直接看到计算出的范围，因此可以使用曼哈顿距离（Manhattan Distance），如等式 3-2 所示。它之所以这样命名是因为用它可以计算出从一点走到另一点时所需经过的正方形街区数。在允许的精度范围内也可以使用其他方法进行估计。

$$\text{estimate} = \text{abs}(dx)+\text{abs}(dy) + \text{abs}(dz) \qquad \text{（等式 3-2）}$$

一旦知道了近似距离，就可以根据 NPC 和玩家间的距离来把这个更新放到某个槽中。最简单的方式是使用一个线性函数。如果游戏管理人员可以知道最大的可视范围并且估计出某个特定物体的距离，就可以通过简单的计算知道应该把更新放入第几个槽中，如等式 3-3 所示：

$$\text{slot} = \text{floor}(\text{slotcount} \times \text{estimate}/\text{range}) \qquad \text{（等式 3-3）}$$

由于使用了近似值，管理人员应该总是把槽的下标限制（clamp）在有效范围内，因为它有可能会超出列表的长度。

3.2.9 调整优先级

上文已经讲解了应该怎样根据距离来确定一般更新的频率了。还有不少方法可以对数据流做进一步的优化。首先，某些 PC 和 NPC 会在很长时间内保持静止。服务端可以一开始就

把这些位置更新发送给玩家以使玩家得到起始位置，之后就只需偶尔更新一下以防数据包的丢失。

服务端还希望从某些 PC 获取更频繁的更新。对于那些在游戏中与服务端处于同一群组中的其他玩家来说，游戏管理人员应该使用一个较小的槽偏移或是特殊的槽范围限制（clamp）来提高他们向我们发送更新的频率。如果玩家把任意 PC 或者 NPC 选为目标，服务端也应该同样地增加其更新频率。因为这表示玩家出于某种原因想要了解更多关于这个 PC 或 NPC 的信息，因此我们有必要对它们进行更频繁的更新。

还有一种方法可以调整优先级，但是需要付出更多的努力。服务端可以使用玩家过去位置的列表来判断他们的位置是否可预测，并以此对玩家的优先级作出调整。如果某个玩家在一个位置停留超过几秒，服务端就应该开始调整优先级以使这个玩家的更新频率越来越低。同样地，如果玩家以直线奔跑，服务端可以根据已知数据精确地推断出他们的位置。如果玩家时而奔跑，时而停止，或是不规则地进行转向和移动，就需要对他们使用更高的更新频率。

对那些很少移动并且已经静止了很长时间的物体，可以使用最极端的调整方法。当这种物体第一次加入队列时，使用通常的方式进行发送就可以了。但是当服务端判断它今后应该如何更新时，可以为它加上一个“不要发送”标记。一旦设置了这个标记，以后就可以跳过这个更新的发送，除非在之后的判断中，它的状态有所改变。

为了减少判断时间，我们在做出判断后，应该把具有这个标记的物体放到最后的那个槽中，从而尽可能地降低对它们进行判断的频率。

在服务端更新玩家状态同时，应该周期性地调整 PC 和 NPC 的优先级。由于这个队列系统具有内建的延迟，通常服务端不必立即把这些改变发送出去。

3.2.10 调整队列

在大多数情况下，服务端可以让队列把数据打散，并且按照它们进入队列的时间发送更新。有时候，这可能会导致一些不连续性（在前面已经讨论过）的发生，譬如说当玩家在一个地方坐了很长时间后突然四处奔跑。

如果玩家的优先级调整突然发生很大的改变，服务端就应该在队列中放入一个新的更新。仅当这个更新可以在没有副作用的情况下进行多次发送时才可以这样做，因为原来的更新也在队列中等待处理。由于另一个更新也在队列中，所以这个新加的更新必须被标识为一次性的，以免在队列中一直出现对同一个 PC 或 NPC 的两个更新。

当 PC 和 NPC 进入玩家范围时，服务端也需要加入一个更新，并且在它们离开范围时删除这个更新。这不仅包括通常的移动，还包括进入和退出游戏。当它们第一次进入范围时，服务端需要在队列中加入一个更新，这个更新应该使用前面所讨论的标准的队列槽计算方法。

这里有一条捷径可以把那些超出范围之外的更新删除。因为在每次发送更新时服务端都必须组织数据，所以可以把这些超出范围的物体在将要发送时裁减掉。这样就不必为了在每个玩家的队列中搜索那些失效更新而浪费时间了。这时服务端应该发送一个代替消息，告诉客户端要更新的这个对象已经在视野之外了。

3.2.11 总结

带宽管理是一个非常关键的问题，它可以为游戏的开发节约很多开支，可以很方便地使用所支付或是所节省的钱来衡量管理数据流的工作质量。这里的难点在于怎样在带宽开销和愉快的玩家体验之间找到平衡，并且在有限的网络资源情况下尽可能地保证游戏环境的平滑。玩家能够获得的游戏体验有多流畅，一部分取决于服务端向他们发送数据的频率，这也决定了玩家对游戏设计的满意程度。

使用这里所描述的技术就可以提供给玩家他们期待的游戏体验，并且也不会浪费不必要的网络带宽，从而在这两者之间找到一个完美的平衡。

3.3　MMP 服务器开发和维护

William Dalton, Maxis
bdalton@maxis.com

对于交付一个稳定的在线游戏来说，最主要的困难在于缺乏一个对终点的良好定义。虽然游戏开发人员已经对 MMP 游戏至少应该具备的功能达成共识，但是要使一个游戏获得成功，除了需要进行日常维护以外，还需要在整个生命周期中进行持续的开发。本文介绍了一些管理开发的技术，适用于那些要对运营中的产品进行持续维护和功能开发的情况。开发人员应该在游戏的生命期中尽早采用这些方法。一旦第一台服务器开始运营并且接受客户连接就是在提供一个运营中的服务了。要让开发人员、制作人员、美工、游戏构筑人员、设计人员和行政人员每天都能正常使用服务器已经很不容易，要是在此基础上再增加 5000 甚至 500 000 个付费用户，那麻烦就更多了。

在线游戏开发中的大部分工作都是在服务端完成的。这样做主要是因为以下几个原因。

- 安全性：服务器是开发人员能够绝对控制的组件。本文不会对安全性进行任何深入的讨论，但是要让游戏变得更安全，最重要的一步就是把所有敏感数据都从客户端移到服务端中。
- 用户偏好：用户讨厌客户端补丁，因为它们会延长进入游戏的时间，而服务端补丁则不会。
- 开发简便：如果游戏客户端在多个操作系统/硬件配置上运行，为每个可能的组合打补丁并且进行测试，不仅会耗费大量的人力而且非常容易出错。通常，在线游戏的服务器架构在整个服务中都是一致的，因此更容易进行可靠地测试和更新。

因此，本文将着重讨论那些能够让开发团队、测试人员、内容编写人员、设计人员、制作人员以及行政人员在整个游戏的开发过程中，甚至在游戏实际运营后一直有效地进行工作的方法。

3.3.1　基本问题

先回答一些基本问题将会使开发过程更为顺利。

1. 正在使用什么服务器？

这是一个在在线游戏开发初期非常常见的问题。大多数游戏程序员习惯于在 PC 上进行开发，他们倾向于相信对网络库的调用会“正确地工作”

并且让他们可以和服务器进行通讯。如果他们不能确定服务器的特征，那么即使是与游戏服务器进行连接也会很麻烦。譬如说，服务器和客户端是否兼容？用来构建（build）它的代码是否最新？这个服务器是否与正确的数据库相连接？

客户端程序员和其他用户应该能够自己回答这些问题。这不仅为游戏的开发减轻了负担，还可以使任何针对游戏的错误报告更为有用。一个简单的方法是，让系统的每个组件都向与它连接的其他组件报告版本。因为每个组件都维护了一个与它连接的组件列表，当它向其他组件发起连接时，它会把整个列表转发给那个组件。与服务器连接的客户必须能够向所连接的组件查询这个列表，并把它显示给用户。这可以让用户拥有一个他们正在使用的组件的精确列表（包括版本）。

图 3-5 的例子中有一个错误的服务器组件。可以看到其中的一个数据库服务器已经过时了。假设游戏使用了上面所说的版本报告机制，用户甚至不需要直接登录到任何服务器就可以立即检测到这个问题，这也避免了用户偶然发现这种情况所带来的错误。

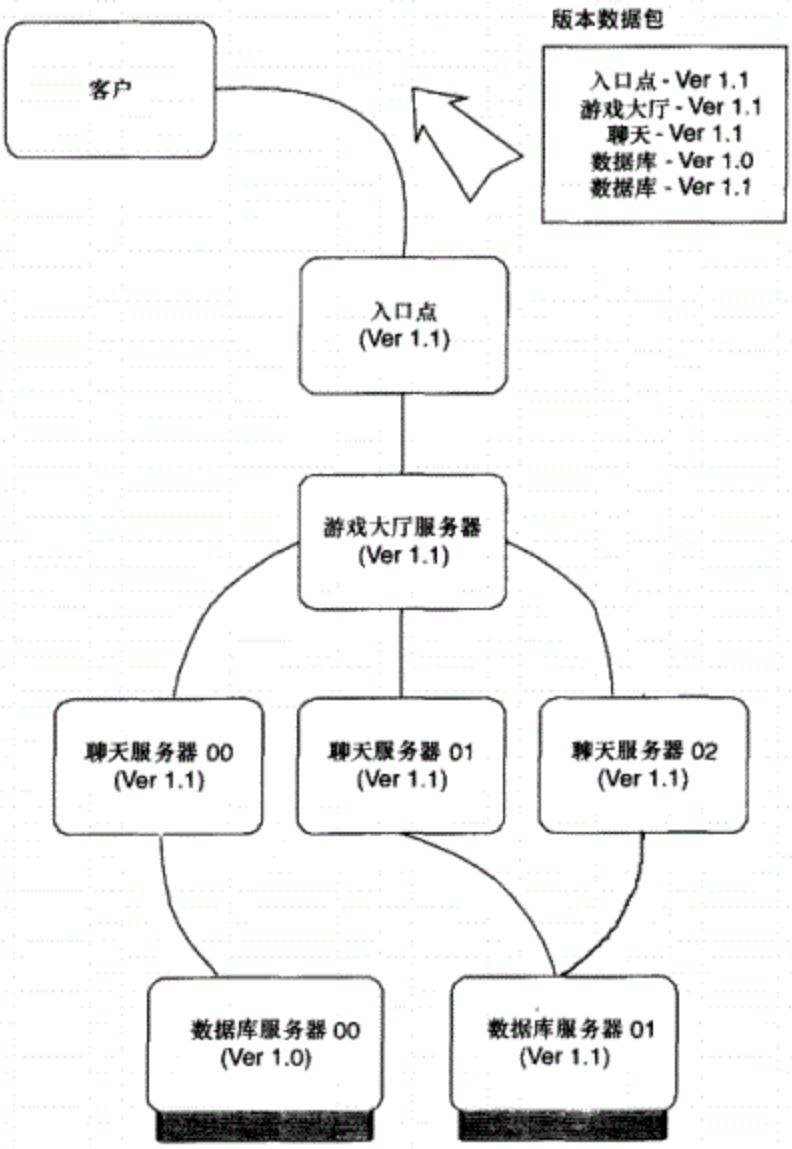


图 3-5 服务端版本报告

除了版本以外，游戏还可以使用完全相同的逻辑在系统间传递其他信息。譬如说，让某个客户知道他所连接的每个服务器组件的进程标识和硬件标识会很有用。如果一个集群（cluster）中的服务器使用了多种硬件，这类功能会为调试工作提供非常大的帮助。

2. 应该和谁建立连接?

客户端程序员及其他用户怎样才能在任意给定时刻确定哪台服务器最适合他们呢？如果上面的第一个基本问题已经有了答案，那么他们可以对所有已知的服务器进行尝试直到获得他们心目中服务器组件和客户端代码的最佳组合。他们也可以通过口口相传或是电子邮件来获得所需的信息。但是更好的方法是把他们进行选择所需要的信息集中放在一个随处可以访问的地方，譬如说一个网页上。这个资源不仅应该提供所有已知服务器的列表，还应该告诉用户他们的版本以及当前是否正在运行中，此外，这个资源还应该提供做出一次可靠的连接选择所需要的任何信息。

最后的警告：把服务器地址硬编码在客户端代码中实际上是对大多数用户隐藏了这一信息，因此这是一个馊主意。

3. 怎样才能知道角色出了什么问题?

当用户与服务器的连接/游戏会话发生简单的问题时，他们是否可以进行调试？这里的可能性几乎是无穷的，每个团队都必须面对一个选择：在开发这个功能上应该投入多少努力，并且他们必须承受由这个选择所带来的后果。

如果开发团队中的某些成员并不进行服务端开发，甚至都不能接触到服务器硬件，下面这些建议或许有用。

- 服务端应该尽可能地向客户端提供有用的出错反馈。这种做法可以防止很多关于“服务器崩溃”的抱怨。如果管理人员可以确定客户端正在使用一个过时的协议，或是发送了一个有问题的数据包，都应该向客户端返回一个有意义的出错信息。如果服务端可以告诉客户端它的版本过时了或是服务器正在关机（故意的），就不应该只发送一个通用的“服务器连接断开”的错误消息。尽可能向客户端提供精确的出错反馈，这可以使开发团队节约大量的调试时间。
- 让用户可以看到服务器日志，或是日志的某些合理部分。更好的方法是，允许用户对服务器日志进行搜索（这也是游戏开发人员应该把游戏日志写得更为简洁有用的原因之一）。游戏管理人员可以在需要时把某些信息从服务器发回给客户端，而使用一个能够访问到所有服务器进程日志的网页就是一个很好的方法。
- 不断监视所有服务器。服务器工程师应该在大多数团队成员之前发现存在于服务端的问题。有些系统可以周期性地检查所有已知的服务器以发现错误，这非常有用。服务器工程师还可以对日志文件进行分析来检查服务器是否发生错误，并且确保在发现了任何问题后都能够及时通知开发团队中的相关人员。服务器工程师应该对任何可能发生的硬件问题进行周期性的检查，譬如说网络问题、进程失控（由于 CPU 或内存问题）、硬盘空间不足等。可以编写一些脚本来周期性地搜集这些数据并且转发给开发团队。此外，把这类信息放在网页上也是一个不错的主意。

下面这段简单的 bash 脚本可以用来进行监视和报告。在 Unix 系统中，程序员可以把它放到 cron 表中来对它进行周期性的调用。类似的脚本可以用来检查游戏日志中是否有重要错误发生，甚至可以用来检查每个服务器的硬件情况（譬如说，可用磁盘空间、内存等）。

```
#! /bin/bash

### Intention: Find all of the core files in ~/production directory on ### a set of
server clusters.
HOME_DIR=/home
   HOSTNAME=`hostname`

SERVER_LIST= devdaily devtest livedaily livetest liveregress

OUTPUT_FILE="coreReport.txt"
MAIL_LIST="bdalton@maxis.com"
SUBJECT_LINE="Core Sweeper Report: ${HOSTNAME}"

### Init file nulling out results from previous run
cp /dev/null ${OUTPUT_FILE}

for SERVER in ${SERVER_LIST}; do
   echo "—— -cores on ${SERVER}:" > ${OUTPUT_FILE} 2>&1
   FILELIST=`ssh ${SERVER}"find ${HOME_DIR}/prod96 -name core*"`

   if [ "${FILELIST}" != "" ]; then
      for FILE in ${FILELIST}; do
         REPORT=`ssh ${SERVER} "ls -l ${FILE}; file ${FILE}"`

        ### Format output in whatever way is useful to you

         FIELDS1=`seq -s"," 1 9`
         FIELDS2=`seq -s"," 16 19`
         echo $REPORT | cut -delim=" " -
         fields=$FIELDS1,$FIELDS2 | column -t > \
         ${OUTPUT_FILE} 2>&1
      done

   fi
   echo "————————————————————————————————————"   > ${OUTPUT_FILE} 2>&1
   echo ""                                        > ${OUTPUT_FILE} 2>&1
done

mutt -s "$SUBJECT_LINE" "$MAIL_LIST" < "$OUTPUT_FILE"
```

3.3.2 对复杂度进行管理

下面这些技术有助于让复杂度最小化。

1. 分支

在游戏发布前的某个时刻，程序员需要对代码进行分支（branch）。要在维护一个运营游戏的同时还进行持续的开发，这可能是最好的方法了。通常，开发团队应该建立一个运营分支和一个开发分支，这里本文会对它们进行详细讨论。

对于大多数组织来说，对一个客户服务器游戏进行分支开发并不简单。通常，开发团队对游戏服务器内部机制的了解并不比他们对其他游戏组件的了解多。这不仅会增加他们的挫折感，还会产生很多错误的缺陷报告（defect report），这最终会使游戏的开发失去大量的时间。所以，应该尽量简化这个过程。

- 坚持使用命名规范。用于开发的服务器必须在它们的名字里包含“Dev”或是类似的字符串，这有助于向潜在用户提供信息。所有运行中的分支服务器/测试中心都必须进行相应的命名（如果游戏系统已经实现了在“基本问题”这一节中所讨论的第一个方案，那么任何连接上的客户都可以很方便地知道这个命名规范）。
- 为开发服务器和运营服务器建立独立的环境。不要把运营分支中的代码放在开发服务器上运行，也不要把开发分支中的代码放在运营服务器上运行。
- 应该使用一个自动构建系统来进行每日构建和用于检查的构建。应该用两台独立的电脑来构建运营和开发版本，并且这些电脑应该是它们所对应的分支中所有构建的惟一来源，最好让它们都不能从其他分支获得代码。

图 3-6 是一个比较简单的开发环境布局示意图。由于开发团队所使用的开发技术和规范不同，有些可能需要使用更多的资源，但是应该不能再少了。

需要注意的是这仅仅是服务端的开发环境。客户端也应该使用专门的电脑来进行构建，并且它们应该符合服务端对开发团队中客户端安装程序进行更新的机制。简单地把客户端软件的镜像放在一个共享的网络服务器上，或使用更复杂的机制都可以达到这个目的，譬如说一系列补丁频道（patch channel）。

在图 3-6 中，本文假设游戏的入口点会显示不同分支和不同环境中的所有服务器。如果这个入口只是一个非常简单的进程，那么这样做也够了。如果入口比较复杂，需要把它和其他代码一起进行分支。譬如说，如果入口点需要决定客户端是否应该安装补丁或是它还包含了其他有趣的游戏特性，可能就需要这样做。

图中还假设每个分支只需要一个数据库实例。这并不一定适用于所有的游戏，但是无论采用什么方法，都必须准备好为所有的服务器提供持久化支持。记住这还包括那些最先进的开发服务器，它们可能会对数据库提出新的要求。

2. 共享代码

游戏开发人员应该怎样构建客户端软件和服务端软件呢？它们是不是在相同的操作系统上进行构建？它们是否会在同样的操作系统和硬件环境中运行？在决定是否让客户端和服务端进行代码共享时，这些重要问题必须首先得到回答。

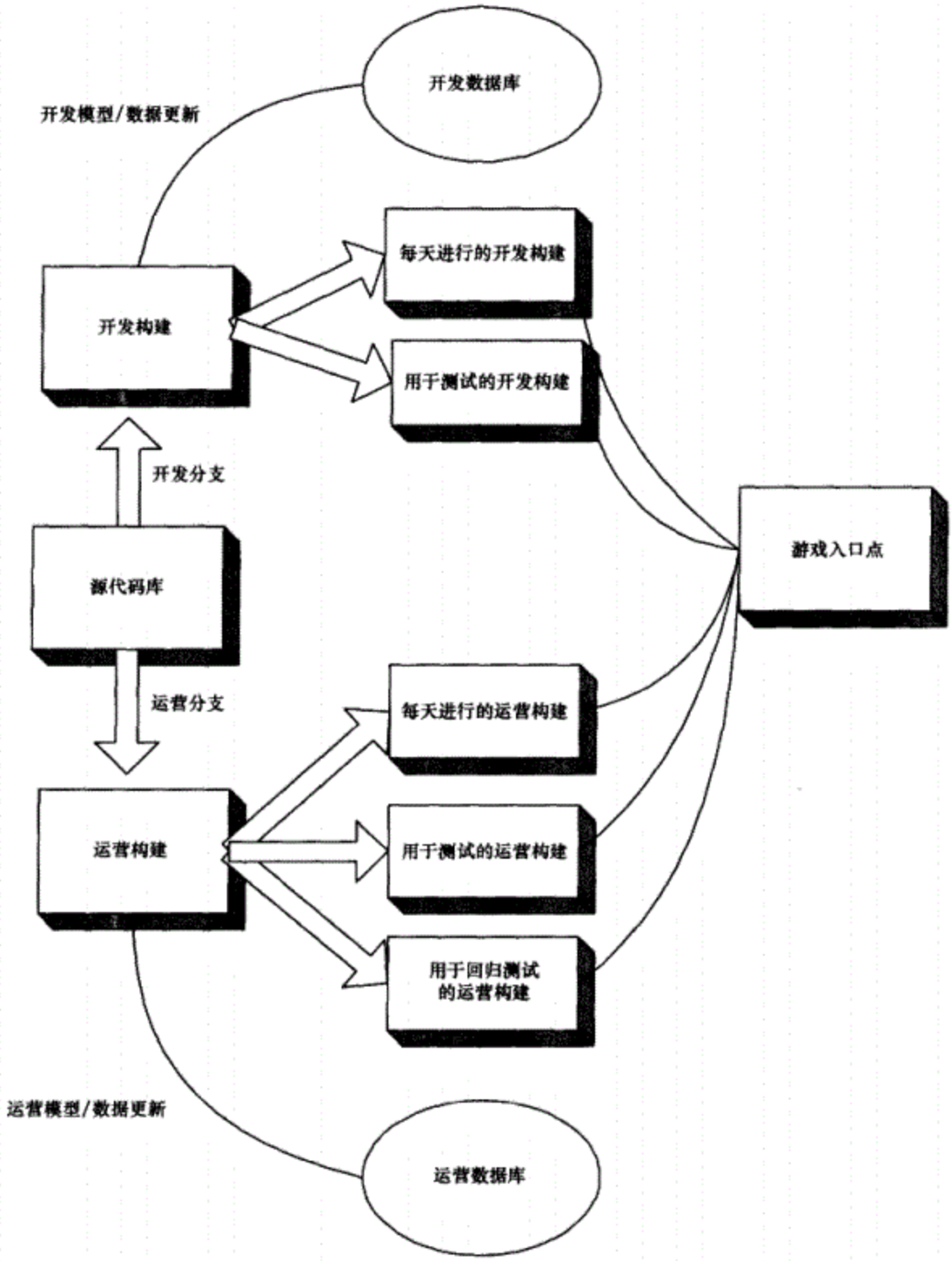


图 3-6 分支开发环境的简单示意图

“代码共享”听起来是个好主意，因为它意味着“代码重用”。那些被重用的代码只需要在一个地方进行修改，但可以在多个地方进行测试，因此它的质量很高，这可能是因为一开

始设计得很好，也可能是因为后期不断改善的缘故。平台无关性同样也意味着高质量的代码。然而，在客户端和服务端之间进行代码共享也会带来一些负担，开发人员应该在这些负担和代码质量上的好处之间进行权衡。

- 流行的 PC 编译器常常鼓励用户使用非 ANSI 的 C++代码。如果客户端程序员使用这样的产品，而服务端开发人员使用一个符合 ANSI 标准的产品，这些差异会影响服务端构建。
- 在客户端和服务端进行代码共享是不是还意味着必须对其他数据进行共享？这些数据是否可以在客户端和服务端的操作系统间移植？在客户端和服务端的目标硬件中是否会有字节顺序问题？这会对服务端和客户端的补丁带来什么影响？
- 最后，还要考虑一下代码共享会为游戏带来哪些安全上的漏洞。对于一个专家来说，客户端程序就好像是一本打开的书。应不应该让别人了解服务端的内部实现，即使只是一部分？

3.3.3 总结

本文最关键的主题就是可见性（visibility）。要尽量让用户可以接触到服务器。游戏开发人员应该让用户在任何时候都可以知道他们使用的是哪些服务器进程。当有问题发生时，游戏开发人员应该为用户提供一些工具来帮助他们解决这些问题。在开发过程中，系统应该向用户提供足够多的提示和线索来帮助他们理解服务端正在做些什么。游戏开发人员应该使用一切可用的方式来告诉用户与系统有关的信息，无论是使用命名规范、使用其他技术还是使用游戏本身。

3.4 小型入口：使用手持设备来接入 MMP 游戏世界

David Fox，Next Game
davidfox@ureach.com

从玩家的角度来看，一流的 MMP 游戏通常都具有流畅的角色动画、大量的城镇、结构细致的建筑、富有层次的环境音效以及一些其他的虚拟现实特征。而最新的游戏机和 PC 硬件更是不断地把这一现实向前推进。

然而，正如任何一个文本界面 MMP 游戏的爱好者可以告诉你的，令玩家沉迷于 MMP 游戏世界的真正原因并不是它的视觉效果，而是它可以不受限制地访问另一个空间的能力。在 21 世纪初人们所拥有的技术中，最注重于使访问不受限制的莫过于移动电话和其他无线设备了。毕竟大多数人都会带着移动电话随处走动。支持无线设备的 MMP 游戏可以把玩家带入一个前所未有的游戏体验中。

在大多数 MMP 游戏中，即使在玩家退出游戏后，游戏世界仍然会持续运行下去。在玩家睡觉时，他们的角色和建筑可能会遭到攻击，游戏计划也可能会被破坏。不仅如此，当玩家离开游戏时，他们还可能会失去很多重要的社交机会。如果一个游戏可以在无线设备上运行，那么玩家几乎永远都不会退出游戏。至少在有重要事件发生时，玩家可以收到无线电子邮件或是短消息（Short Message Service，SMS）。有些游戏甚至允许玩家发出一些简单的高级指令，譬如说“让所有部队都进入防御状态”。

理想情况下，一个真正的移动游戏不仅应该允许玩家使用手持设备来接入游戏世界，还应该让他们可以做到任何在台式机上能够做到的事情。如果某个玩家的部队在他参加业务会议时遭到偷袭，他应该可以立即收到通知并且暗自登录游戏和敌人作战。

3.4.1 无线设备和网络

虽然无线设备的功能和速度正在飞速地提高，它们仍然具有很多缺陷。大多数无线设备的屏幕是黑白的，并且长和宽都只有大约 100 个像素；处理器的速度比台式机慢 100 倍；而内存更是小得可怜（以 K 而不是 M 来计量）。

不仅如此，大多数第二代（2G）无线网络的传输速度只有 9.6kbit/s。

大多数欧洲移动电话都是基于 GSM（Global System for Mobile Communication）的，它们发送数据的速度是 9.6kbit/s。有些网络使用时分多址（Time Division Multiple Access，TDMA），和 GSM 一样，它也受到 9.6kbit/s 的限制。北美和南美、前苏联、以色列、东亚和中非的网络基于码分多址（Code Division Multiple Access，CDMA）来传输数据，它的速度略微高一些，可以达到 14.4kbit/s。虽然 2.5G 和 3G 网络逐渐开始产业化，但是要让大多数用户使用上它们还需要很多年的时间。

更糟糕的是，由于无线电通讯固有的干扰和噪声，广域无线网络通常都具有很大的延迟。大多数无线网络需要让数据包通过很多路由器。通常基于卫星的无线网络会导致更大的延迟。在日常使用中，1 到 2 秒的网络延迟并不少见。

虽然这些设备和网络受到诸多限制，但它们依旧非常流行。某组织预测到 2005 年底，全球使用移动无线设备访问因特网的用户将达到 7.29 亿人，而在 2000 年底，这个数字还只有 3900 万[Intermarket01]。

MMP 游戏开发人员所面临的问题是，怎样才能用这些高延迟、低带宽并且 CPU 也不强的设备来让玩家忘我地投入到游戏世界中去呢？首先，开发人员有必要知道编写无线游戏所使用的语言以及它们的优缺点。

3.4.2　J2ME

Java 是由太阳微电子公司（Sun Microsystems）创造的，它已经成为无线设备开发的标准语言。这是一个开放的语言，所有必须的开发工具都是免费提供的。

Java 运行在虚拟机上，这意味着只要开发人员使用正确的方法，理论上相同的字节码就可以在任何支持 Java 的平台上运行。Java 被设计得非常易于使用。作为一种面向对象语言，它不支持显式的指针，也没有复杂的内存操作，通过自动垃圾收集来管理内存。系统事件（譬如说，在进行游戏时接到一个语音呼叫）会被自动处理。最重要的是，Java 小应用程序（applet）不能访问安全运行环境以外的任何函数和内存，这意味着几乎不可能用它编写出恶意的代码或病毒。

J2ME（Java 2 Micro Edition）[J2ME01]在试图保持标准 Java 语言的优点的同时，对其进行了精简以适应小型设备上的开发，譬如说行动电话、寻呼机和掌上电脑。几乎所有主流移动电话生产商都参加了 Sun 制订 CLDC（Connected，Limited Device Configuration）[CLDC01]和 MIDP（Mobile Information Device Profile）[MIDP01]的过程。一个为移动电话编写的 Java 小应用程序被称为“移动应用程序（MIDlet）”。

很多设备生产商发布了 J2ME 的扩展 API。譬如说，西门子（*Siemens*）以 MIDP 为基础开发了一套游戏 API。NTT DoCoMo 并没有使用 MIDP，它使用的是一个独立的 Java 实现，被称为 I-Appli。

目前，MIDP 1.0 还有一些限制：它不支持透明图形和浮点运算；它不能通过本地代码（native code）直接访问设备硬件、文件 I/O 或是内存。HTTP 是 MIDP 设备必须支持的惟一协议。通常所有 MIDP 游戏都是根据 HTTP 来设计的，因为这是所有 MIDP 设备都必须支持的最低标准。

MIDP 2.0 是为更强大的设备设计的。它对上述缺点中的大多数进行了修正和改进。

3.4.3 BREW（二进制的无线运行时环境）

Qualcomm 开发了一套虚拟机和语言，称为 BREW（Binary Run-Time Environment for Wireless）[BREW01]。这是一个以 C++为基础的开放性语言。Qualcomm 已经把 BREW 植入到了很多 CDMA 手机的芯片中。BREW 也可以支持其他语言，目前 Quallcomm 正和 IBM 合作在 BREW 上创建一个 Java 虚拟机。

BREW 没有安全运行环境，这意味着应用程序可以直接访问文件系统、网络套接字、内存和屏幕。它还支持透明的精灵（sprite），并且有一个内建的资源文件系统以方便开发人员载入图片和声音。

不仅如此，BREW 发行系统（BREW Distribution System，BDS）还为在线购买应用程序并直接下载到移动设备提供了一个标准方法。这个集中支付系统对开发人员和最终用户来说都很方便。

BREW 的缺点在于，它的当前版本只支持 500 字节的动态内存，并且不支持任何静态数据。要开发面向最终用户的 BREW 应用程序，开发人员必须购买非常昂贵的 ARM 编译器。目前支持 BREW 的移动电话也很贵，并且内存非常有限。此外，对一个运行在真实手机上的 BREW 程序进行调试也是非常困难的。

因为今后大多数设备会支持 J2ME，并且在 BREW 上也可以进行 Java 开发，本文中的例子将使用 Java。然而，这里的概念可以用任何语言实现。

3.4.4 无线界面和游戏设计总览

把一个精彩纷呈的虚拟现实移植到小型设备中，需要开发人员对现实做一些本质上的改变。每个游戏环境都是由人物、地点和物品等组成的。玩家和这个环境进行交互的方式称为“游戏模式（gameplay）”。移植时，设计人员必须同时改变游戏环境和游戏模式。

大多数游戏的核心是人物或角色。游戏通常围绕一个主角，并且主角通常会和某个敌人进行战斗。为了使游戏中丰富的角色即使在移动设备上也能栩栩如生，设计人员必须把它们浓缩为精华。很明显，游戏设计人员必须去掉所有漂亮的三维模型以及那些复杂的统计信息。必须使用精灵（sprite）来表示人物，动画的帧之间只能改变很少的像素，就好像处于黄金时代的家用游戏那样。就算是劳拉·克劳伏特（《古墓丽影》的女主角）看上去也只能和大嘴巴（PacMan）一样。

游戏中的地点是所有游戏行动发生的地方。通常二维地图就可以对三维空间进行充分地表示。游戏设计人员可能需要实现战场雾化（fog-of-war）或是运用其他技巧来避免让某些玩家获得战术上的优势。不仅如此，他们还必须对复杂的物理运算以及其他环境交互进行大量的简化。游戏设计人员最好把游戏环境中最有趣的特性提取出来，然后在移动设备上对这些部分进行模拟。

设计人员必须对很多东西进行修改以使得它们适合二维环境，这包括武器、财宝、钥匙、以及角色在游戏中可以穿过的门。那些在显示器和电视上看上去很真实的东西，在小屏幕上只能是抽象的。我们必须把所有物品都缩减为图示、图表或其他简化的表达。

最后，也是最难实现的，就是对游戏模式自身的改变。移动电话通常只能通过几个方向

键来进行输入。并且，因为延迟很大，任何依赖于迅速反应的游戏模式都必须改变。本质上，必须把游戏的一部分做成回合制的。这种模式转变类似于教练和四分卫之间的关系。这里，玩家就是教练，他通过发送指令来指挥球的运行，而不是像四分卫那样直接发出命令来让球移动到（x，y）位置。

表 3-3 中所包含的例子说明了在移动设备上可以怎样表示游戏机游戏或是 PC 游戏中的主要特性。

表 3-3　　游戏特性：从大到小

	游戏机/PC	手持设备
户外	在旷野中有很多城镇、自然资源和废墟	好像旅游地图，使用图标来表示特殊位置或特殊特性
城市街道	3 维或等容（isometric）的迷宫	垂直卷轴迷宫
户内	使用走廊分隔的房间，有锁住的门、自动扶梯和垂直电梯等	好像建筑蓝图，使用图标来表示锁、楼梯等
过场	动画	静态的图片和一些文本
其他角色	使用动画表示	地图上的图标
角色状态	带有图形的数据表	记分牌
背囊	支持拖曳的图形界面	带有复选框（checkbox）的列表
角色互动	非常精确，使用特殊的用户界面以及特殊的动画	使用文本或是图标
大型战斗	动画，鸟瞰	文本描写
战略	即时	使用分支图（branching diagram）预先计划，可以进行更新
战术	不停的输入，迅速切换武器、魔法和模式	使用具有教练功能的 AI 代理
购物	在商场中游荡，从货架上购物	层次菜单
听觉	音箱、耳机或是像漫画书一样的气球提示窗口	滚动信息
交谈	话筒或是打字	预先定义的消息

要记住的关键是，虽然在移动设备上具有完全不同的游戏经历，但真实的游戏世界并不需要被改变。虽然游戏设计人员不能使用第一人称视角来设计游戏，就好像一个正在法国北部村庄中战火纷飞的街道上奔跑的士兵那样，但可以使用一个将军的视野，使用一个基于计算机的卫星视角来控制特定士兵的行进。

3.4.5　对象设计

由于同一个对象在不同的无线设备和桌面版本中表现方式会完全不同，游戏必须使用一个可高度扩展的架构。设计人员必须对每个人物、地点和物品进行重新设计以支持各种不同的输入设备和不同的细节层次。虽然每个对象的关键属性（譬如说它在游戏世界中的位置和

它的生命值）将保持一致，但是输入/输出方法和对象的显示必须非常灵活。

游戏的设计需要克服两个主要问题：向每个平台发送合适的对象和事件；对用户输入进行处理。要做到这两点，关键在于让游戏中的每一个数据类型都可以根据具体情况进行定制和伸缩。这一设计方法不仅可以帮助开发人员把游戏延伸到无线设备上，还可以提高游戏在不同的平台和网络类型上的总体性能。

1. 发送对象

假设在一个虚拟的客栈中，有一个移动用户的角色和一个桌面用户的角色站在一起。每个玩家都需要接收同样类型的信息：周围的建筑、角色和物体。但是发送给每个客户的实际内容可能有着巨大的差别。

现存的标准可以用来参考以构思怎样创建这样一个系统。对于 Web 应用程序来说，使用 XSLT（Extensible Stylesheet Language Transformation）[XSLT01]模版语言就可以对同样的 XML 数据为不同类型的客户进行转换。譬如说，同样的内容可以被格式化为 Web 浏览器可以理解的 HTML，也可以被格式化为无线设备可以理解的 WML（Wireless Markup Language），或是被格式化为电话设备可以理解的 VoiceXML。如图 3-7 所示。

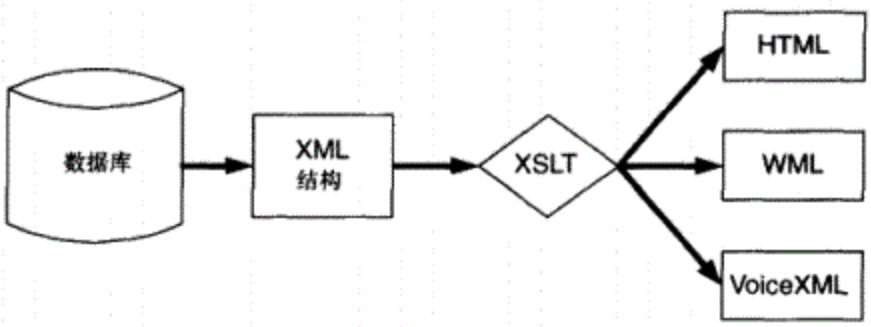


图 3-7 XSLT 是怎样工作的

XSLT 可以帮助游戏开发人员把数据的逻辑结构和表示分离开。设计人员可以在游戏中使用类似的设计方法。譬如说，角色的数据可以使用图 3-8 中的基类表示。

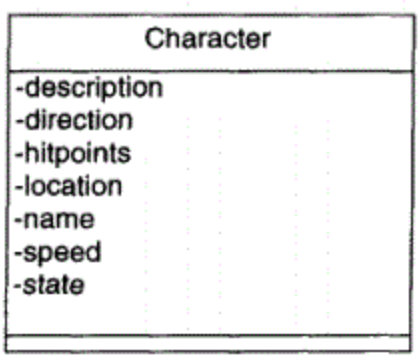


图 3-8 基本的角色（Character）类

假设这个类中的每一个属性都有它自己独立的类。这使得每个属性都可以执行一些特殊的功能，并且可以有父属性和子属性。这个角色（Character）类可以使用下面的 XML 表示。

```
<CHARACTER>
   <HITPOINTS>100</HITPOINTS>
   <NAME>Bartender</NAME>
   <DESCRIPTION>This swarthy character can serve you a fine mug of ale!</DESCRIPTION>
   <LOCATION><X>100</X><Y>320</Y><Z>99</Z></LOCATION>
   <SPEED><X>23</X><Y>0</Y><Z>0</Z></SPEED>
   <DIRECTION><X>1</X><Y>0</Y><Z>0</Z></DIRECTION>
   <STATE>pouring</STATE>
</CHARACTER>
```

作为一个被简化了的例子，服务端可能希望把上面所有的数据发送给一个游戏机玩家，但是只发送如下某些数据给无线玩家。

- 只发送战斗（FIGHTING）或是死亡（DEAD）状态，并且把状态简化为“F”和“D”。
- 只发送 X 坐标和 Y 坐标的更新，不发送速度和方向。
- 不发送描述（description）。

也就是说，服务端只会发送下面的 XML 数据。

```
<CHARACTER>
   <HITPOINTS>100</HITPOINTS>
   <NAME>Bartender</NAME>
   <LOCATION><X>100</X><Y>320</Y></LOCATION>
</CHARACTER>
```

通过 DeviceSender 接口，游戏管理人员可以为每个属性设定规则，并且在运行时使用一种类似于 XSLT 的方式对它们进行转换。基本上，每种数据类型都需要实现 DeviceSender 接口。图 3-9 中是这个接口的类图。

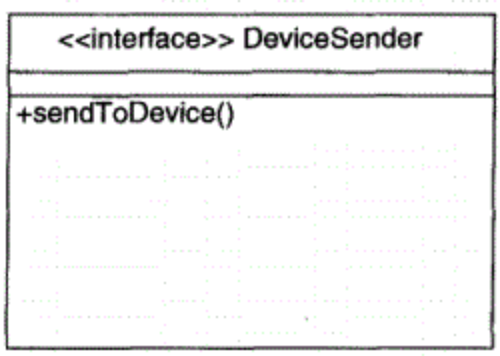


图 3-9　DeviceSender 接口

sendToDevice（向设备发送）方法要返回一个字符串、字节数组、字符指针或是任何可以用来创建网络消息的数据类型。用来保存每个角色状态信息的状态（State）类可能会有如下这样一个方法。

```
public String sendToDevice(Device d)
{
   if (d instanceof Wireless)
```

```
        {
            if (thestate = State.FIGHTING)
                return "F";
            if (thestate = State.DEAD)
                return "D";
            else
                return null;
        }
        return thestate.toString();
    }
```

每当服务端向一个客户发送数据或事件时，只需要调用每个数据的 sendToDevice 方法就可以了。

2. 接收输入

现在，假设这个移动用户决定和桌面用户进行一场战斗。服务端不仅需要向移动用户更新战斗情况（可能是文本方式的），还需要让他可以发送高级的“指导”命令，譬如说通过在嵌套菜单中进行选择来发出这些命令。

与此同时，这个桌面用户可能正一边看着两个 3D 角色在进行搏斗，一边疯狂地摇动着他的游戏手柄。在所有条件相同的情况下，由于桌面玩家能够更快地对意外情况作出反应，因此相对而言他更占优势。但是，如果 AI 设计得很好，客观端就可以给移动用户一些战斗机会。一个由移动用户控制的高级角色应该能够战胜一个由桌面用户控制的低级角色。

大多数 MMP 游戏都有一个仿真对象（Simulation Object，SOB）基类，所有的角色、地点和对象都是从它继承而来的[GPG01]。这些 SOB 之间的关系使得服务端可以为游戏世界维护一个巨细无遗的仿真。根据具体实现的不同，SOB 可以被保存在数据库、内存或是 XML 中。

在客户端，这些 SOB 将会是特殊的代理对象。譬如说，行动代理（ActorProxy）类可以实现两个主要的方法：ReceiveEvent（接收事件）会处理来自服务端的事件，RequestAction（请求行动）会告诉服务端试图进行一个给定的行动。

从服务端发出的（由客户端的 ReceiveEvent 方法接收的）事件也应该实现 DeviceSender 接口。这样服务端就可以根据不同的目标平台以不同的方式传输同一个事件。

对于游戏机平台和无线设备来说，客户端的 RequestAction 方法将有很大的差别。服务端必须能够对接收到的事件进行合适的转换。每当服务端接收到一个事件，它必须把发送这个请求的设备作为一个参数传递给一个从事件处理器（EventHandler）抽象类派生出来的类，如图 3-10 所示。

游戏开发人员应当为游戏支持的每个设备创建一个独立的事件处理器类。譬如说，一个无线事件处理器（WirelessEventHandler）可以把无线设备发送的事件转换为较大的游戏事件。事件处理器还可以和主要的游戏仿真对象相关联以确定如何处理事件。譬如说，一个由无线设备发出的事件可能过了很长时间才到达服务端以至于该事件已经失效了，这时事件处理器可以把它忽略掉。

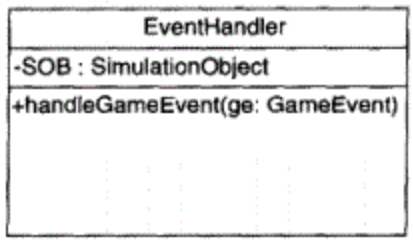


图 3-10　事件处理器抽象类

3. 加入智能

大多数游戏只把人工智能应用在非玩家角色上。为了创建一个即使在用户输入很少的情况下都可以持续运行的游戏世界，本文建议让每个角色都能与一个很强的 AI 控制状态相关联。这样游戏开发人员就可以使用游戏中最复杂的 AI 机制去控制每个角色以及游戏中的其他对象。

图 3-11 中是 AI 控制管理器（AIControlManager）类的类图，它保存了指向所有控制状态（ControlState）类的连接。当需要进行一个行动时，它会根据当时的情况选取一个合适的控制状态类。每个角色都应该有它自己的 AI 控制管理器。这样，同一个角色就可以在多种状态（譬如说完全由系统控制；偶尔受到提示；完全由用户控制）之间进行切换。因为 AI 的运算量很大，所以游戏控制人员甚至可以根据服务器的当前性能来调整 AI 控制的级别。

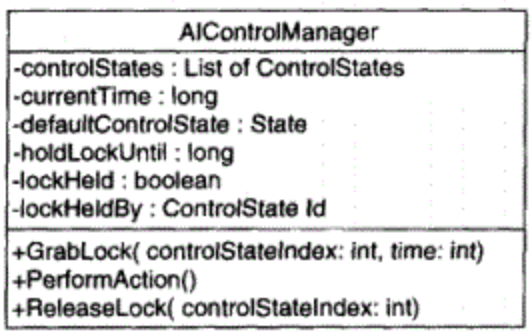


图 3-11　AI 控制管理器类

控制状态（controlStates）数组具有给定角色可以使用的每一个控制状态的指针。AI 控制管理器中的 PerformAction（执行动作）方法会检查是否锁定了某个控制状态，如果锁定了，它会调用那个类的 PerformAction 方法。如果当前没有锁定任何控制状态，缺省控制状态的 PerformAction 方法将会被调用。

譬如说，如果一个无线用户觉得他可能会遭到一次偷袭，他会要求一条特定的巡逻犬在接下去的 10 分钟内负责警戒。服务器应该把用户控制状态（UserControlState）设为当前活动的控制器，它会执行用户的警戒请求。用户控制状态会持续锁定 10 分钟，除非用户发出另一个请求。一旦时间过去，用户控制状态就会解除锁定，于是 AI（缺省控制状态）就会重新获得控制。

服务器不仅可以根据时间来进行用户控制状态锁定，还可以根据优先级来进行锁定。服务器还需要确保对 ReleaseLock（解除锁定）方法进行适时的调用。譬如说，如果巡逻犬将会持续警戒 10 分钟，除非它快要死亡，那么当它接近死亡时，用户控制状态就会解除锁定，这样 AI 控制状态就会接管并尝试挽救它。

3.4.6 网络设计：使用代理服务器

每当有相关事件发生，服务端都会向客户端发送简单的更新。问题在于，服务端怎样才能确定对于不同客户而言，哪些事件是相关的呢？通常，客户端会持续运行一个本地的游戏仿真并且可以告诉服务端它所需要的信息。然而，由于无线设备的内存极其有限，即使要在客户端保持一个游戏仿真的近似也是不可行的，更不用说在客户端保持整个游戏仿真了。

服务端可以通过一个代理服务来模拟客户的视角并且只向小型设备发送那些最重要的信息。不仅如此，对很多移动电话（特别是那些使用 MIDP 的）来说，HTTP 是它们惟一可以使用的通讯协议。游戏开发人员当然不希望把所有的游戏数据包都通过 HTTP 传输。相反，他们希望可以用任何协议来编写游戏服务端。代理服务器会负责转换并且把相关信息用 HTTP 协议来打包。

当然，代理服务器会带来额外的延迟。但是它可以为那些支持更好的网络协议的设备创建更好的游戏体验。譬如说，代理服务器可以通过调整所要发送的数据的内容和频率，来对不同设备进行智能化的处理。

3.4.7 总结

要把一个内容丰富的图形化多人世界装入一个小型设备，工作人员必须作出大量的牺牲和妥协。游戏开发人员不仅需要支持两套客户代码，还需要进行主动的设计、测试和调整来确保游戏既富有趣味又能够在所有平台上都保持平衡。只需要在设计时多加考虑，就可以让每个最新的游戏特性都可以被不同的设备访问。这可以让玩家在旅途中、在工作时、在上课时或是在住所附近闲逛时都可以投入到游戏世界中。让玩家更方便地进入游戏世界有助于创建一个更加投入的玩家社区，最终可以消除游戏和现实之间的界限。

3.4.8 参考文献

[BREW01] Qualcomm.com，BREW home page，*http://www.qualcomm.com/brew/.*

[CLDC01] Sun.com，“CLDC Information Page,” *http://java.sun.com/ products/cldc/.*

[ALEX02] Alexander，Thor，“A Flexible Simulation Architecture for Massively Multiplayer Games,” *Game Programming Gems 3*，Charles River Media，2002.

[Intermarket01] The Intermarket Group，“Mobile Wireless Internet Briefing,” 2001.

[J2ME01] Sun.com，J2ME information page，*http://java.sun.com/j2me/.*

[MIDP01] Sun.com，“MIDP Information Page,” *http://java.sun.com/ products/midp/.*

[XSL01] W3.org，“The Extensible Stylesheet Language (XSL),” *http://www.w3.org/Style/XSL/.*

3.5　使用 Python 进行精确的游戏事件广播

Matthew Walker，NCsoft 公司
mwalker@softhome.net

多年来，事件驱动系统一直被用来实现现代软件所要求的高度交互。游戏设计人员不仅可以使用这些系统来模拟并发，还可以用它们来提高软件对非预期输入迅速作出反应的能力。正如大多数商业软件中所提到的，事件驱动系统通常用于图形用户界面（Graphical User Interfaces，GUI）的开发。随着计算机游戏从简单的 DOS 程序发展到复杂的三维虚拟现实引擎，事件驱动系统对于游戏开发来说越来越重要。在 MMP 游戏中，事件将会空前重要，因为大量的用户会导致在任意时间都有很多事情发生。

本文介绍了 3 种不同的模式，它们使用 Python 程序设计语言，以在 MMP 服务器中实现高级游戏事件系统，每一种都比前一种复杂。本文的源代码可以在所附的 CD-ROM 中找到。每种模式都建立在前一种模式的基础上并且使系统变得更为强大和灵活。与此同时，本文会对每种实现的优缺点进行讨论。第 3 种（也就是最终的）模型中所使用的技术能够精确地控制事件的分发和处理，它可以用来创建一个非常灵活有效的框架并使用在不同的游戏系统中。

3.5.1　事件驱动编程

事件驱动编程（event-based programming）这一术语是指不需要使用线程就可以在一个进程内模拟并发的技术。线程是一种管理并发的通用系统，它在操作系统这一级实现。要正确地使用线程非常困难，因为代码执行权在任何时刻都可能会被抢占。使用线程不仅需要对共享数据进行专门的管理，调试起来也很麻烦，更不用说它们通常要求类和模块能够正确地处理对时间的依赖，而这往往会违反抽象规则。与此相反，事件是非抢占式的，它们只有在被应用程序调用时才会执行，并且会持续执行直到任务完成。这使得事件驱动编程更容易理解和调试，并且可以在代码中合适的地方对时间依赖进行分离[Ousterhout96]。

1. 同步和异步调用

事件驱动编程依赖于对两个互补概念的理解：同步（synchronou）和异步调用（asynchronous call）。同步调用（也被称为阻塞（blocking）调用）在执行完毕之前不会把控制权返回给调用者。同步操作的调用者可以认为

当调用返回时，它所请求的操作已经执行完毕。与此相反，从调用者的角度来看，异步调用（也称为非阻塞（non-blocking）调用）会立即返回，但是它们会独立于调用者的执行而继续执行。异步操作的调用者必须作一些特殊安排从而在异步调用完成后能够收到通知。通常，游戏开发人员可以使用回调函数（callback function）的方式，异步操作会在完成任务后调用这个函数。

2. 并发模拟

在 MMP 游戏服务端架构中提供一个并发的假象是至关重要的，因为游戏服务端必须随时为游戏中成千上万的玩家提供支持。事件驱动系统通过把每个玩家的请求当作一个小型原子操作来实现这点。请求会随着接收的顺序被依次执行。如果一个复杂操作可以被安全地分解为两个或多个异步操作，它就会被分发给事件系统并在游戏循环下一次执行时被处理。因为完成每个请求所需要的时间都很短，这样服务端就可以很快地对新的请求进行处理，这就提供了一个假象：多个请求被同时并发地处理了。

3. 高级游戏事件

本文着重讨论了怎样使用基于事件的方法来处理高级（high-level）游戏操作。这包括大多数由玩家请求直接调用的功能，譬如说与游戏世界中的物品、其他玩家或是关键的游戏系统进行交互。这还包括了由 AI 控制的角色发出的类似行为。但是它不包括那些像底层网络代码、物理系统、碰撞检测、移动控制之类的功能。然而，这些底层系统可以为事件驱动的游戏代码提供钩子（hook）。譬如说，碰撞系统可以在一个对象接触到一个碰撞体（collsion volume）的时候调用游戏代码注册的回调函数。这样的钩子可以被简单地看作事件。

4. 游戏服务端主循环

游戏服务端的主循环是事件驱动系统的根（root）。它非常简单，正如下面的伪码所示。

```
mainloop()
{
    while(game_is_running)
    {
        // wait for incoming requests
        WaitForRequests();

        // handle requests
        ProcessRequests();
    }
}
```

WaitForRequests（等待请求）函数可以让服务端进入一个有效的等待状态，直到收到某个请求。要实现这个功能，游戏服务端可以在 I/O 资源（譬如说网络套接字和内存文件、I/O 完成端口或是异步函数调用（APC）[MSDN01]）上使用 select()，也可以循环地调用 sleep() 函数并检查输入队列中有没有新的数据。

ProcessRequests（处理请求）函数接收任何等待请求并且把它们分发给合适的代码来进

行处理。请求可以包含从游戏客户端收到的玩家请求，也可以包含游戏循环某次执行中所做的异步调用。分发可以是对回调函数的直接调用，也可以是根据请求中的某些输入进行由数据驱动的委派。这一功能的关键在于所需执行的代码可以预先知道，并且只需要很少的判断就可以决定应该执行哪些代码。

5. 事件和线程并不是互斥的

在游戏服务端使用事件驱动的方式与在系统中正确地使用线程并不冲突。上面所介绍的游戏循环可以运行在一个多线程服务器中，它可以在一个独立的游戏线程中运行，而由其他线程对网络消息、初始化系统、关闭系统等进行处理。网络线程可以把接收到的请求放入一个队列，游戏线程从这个队列中取出请求并进行处理。游戏循环中发出的异步调用也可以插入这个请求队列，无论是否还有其他线程存在。

6. Python 和事件驱动编程

几乎所有程序设计语言都可以实现事件驱动编程。由于以下理由，本文中的示例都使用 Python 实现。

1. 它的语法和语义很容易理解。
2. 它是动态类型的语言，可以创建复杂的数据结构而不会受到由此带来的类型限制。
3. 在 Python 中，类、模块、函数和方法都是头等对象（first-class object），这样可以很方便地实现回调和其他事件驱动方案。
4. 在 Python 中，类、模块和实例的属性都可以通过一个字符串名字来访问，因此游戏设计人员可以更灵活地定义延迟绑定（late-bound）的函数调用来实现事件驱动。

3.5.2 延迟调用

本文所介绍的第一个事件驱动系统是基于延迟调用（deferred call）模式的。这是一个非常常见的基本用法，所有后续的实现都以它为基础。延迟调用就是指使用请求的方式来对函数或者方法进行调用，从而使它们可以在未来的某个时间被处理。在事件驱动系统中，这意味着这个请求将会在游戏循环的某次后续执行中得到处理。延迟调用具有 6 个主要元素：

1. 调用的目标对象（target object）；
2. 调用本身；
3. 调用的参数，它必须符合调用的形式参数列表；
4. 调用的执行时间，这必须是将来的某个时间；
5. 当延迟调用完成后所调用的回调函数；
6. 回调函数的参数。

1. 延迟调用接口

延迟调用实现使用了下面的定义。

```
def Call(target, call, args, delay, cb, cbArgs):
```

```
# deferred call implementation
```

目标对象（target）参数是一个整型的对象标识。使用一个对象管理器类就可以把它转化为对象引用。因为这个调用是异步的，在执行到它的时候，这个目标对象可能已经被游戏销毁了。如果使用目标对象的引用而不是标识来作为参数，可能会影响这个对象的清除工作，因为 Python 是基于引用计数的[Python01]。

调用（call）参数是一个字符串，它记录了目标对象中服务端想要调用的方法的名字。它使用 Python 的 getattr()函数来通过名字查找要调用的方法。因为 Python 的方法是头等对象，它们也会被引用计数，这个方法避免了上面所说的相同的问题。

调用参数（args）参数是一个元组（tuple），它包含了这个调用的形式参数，这些形参必须符合这个调用的参数列表。元组是 Python 的基本数据类型，它本质上是一个不可改变的列表（immutable list）。

延迟时间（delay）参数表示进行这个延迟调用之前所需等候的最小时间，它的单位是毫秒。这个值也可以是 0；这意味着这个延迟调用会立刻进入请求队列。

回调函数（cb）参数是对某个回调函数或方法的引用，它会在延迟调用完成后被调用。如果不需要通知，可以把它设为 None（Python 中的 NULL 类型）。这里接受一个引用是可行的，因为发出调用的代码应该知道它自己的生命期，可以假设如果它知道自己不能接收到这个通知，它就不会提出要求。

回调函数参数（cbArgs）参数也是一个元组，它包含了传递给回调函数的参数。如果不需要任何回调函数，也可以把它设为 None。如果一个回调函数不需要任何参数，就可以使用一个空元组“()”，这样可以避免对这个值进行额外的检查。

2. 延迟调用的实现

在 deferred.py 模块中可以实现这些功能。这些功能的实现需要满足两个关键要求。

1. 延迟调用必须能被缓存在某种类型的容器里，并且可以在以后取出。
2. 延迟调用不能在它们所期望的执行时间之前执行，如果多个延迟调用被安排在同一时间执行，它们将按照先进先出的方式被处理。

因此，这些功能的实现不仅需要某种形式的优先队列，还需要某种方式来根据一个调用的调度时间来决定其优先级。第二个问题很有趣，它需要我们定义一个可以按照时间来排序的“可调用对象（callable object）”。下面是一个可行的实现。

```
import objMgr # object manager: maps ids to objects

class DeferredCall:
    def __init__(self, id, call, args, t, cb, cbArgs):
        self.targetId       = id
        self.call           = call
        self.args           = args
        self.time           = t
        self.callback       = cb
        self.cbArgs         = cbArgs

    def __cmp__(self, other):
```

```
        return cmp(self.time, other.time)

    def __call__(self, objMgr):
        target = objMgr.GetObject(self.targetId)
        if target is not None:
            try:
                method = getattr(target, self.call)
            except AttributeError:
                print "No %s on %s" % (self.call, target)
                return

            apply(method, self.args) # make the call

            if self.callback is not None:
                # notify that call is complete
                apply(self.callback, self.cbArgs)
```

这个类对延迟请求的重要属性进行了封装。导入对象管理器（objMgr）模块就可以根据对象标识获得目标对象。__init__()构造函数使用请求数据来构造对象。注意在self.time属性中包含了一个从Call()函数的延迟时间参数计算而来的绝对时间值。这个转换是为了支持在优先级队列中对延迟调用（DeferredCall）对象进行排序。要进行这样的排序，程序员必须使用Python保留的__cmp__()钩子[Python02]，它使得对对象进行比较时可以以它们的执行时间（time）属性为依据。通过另一个Python保留的钩子__call__()方法就可以使得延迟调用对象可以像函数/方法那样被调用。这个方法先获得目标对象和以它为对象进行调用的方法，然后再进行调用，如果存在回调函数，它还会接着执行回调函数。无论是调用目标对象的函数还是调用回调函数，都要用到Python的apply()函数[Python03]，它使得我们可以把参数作为元组来传递给函数从而避免受到目标函数参数数量的限制。

Call()函数的实现使用了一个延迟调用对象，如下所示。

```
import time
import bisect    # efficient list operations

# module-scoped list
deferredCalls = []

def Call(target, call, args, delay, cb, cbArgs):
    # schedule a call for later execution
    callTime = time.time() + float(delay) / 1000.0
    dCall = DeferredCall(target, call, args, callTime, cb, cbArgs)
    bisect.insort_right(deferredCalls, dCall)
```

这里的实现是基于模块的，而不是基于类的。这纯粹是出于服务端架构的要求，它并不会影响游戏原理。这个模块定义了一个延迟调用（deferredCalls）列表，所有的延迟调用都会被插入这个列表并且按照时间排序。Call()函数会把延迟时间（delay）参数转换为一个绝对时间，随后创建一个延迟调用对象并把它插入到延迟调用列表中。Python的bisect模块中的insort_right()函数可以使用游戏开发人员为延迟调用类所定义的__cmp__()方法来进行一个高

效的插入排序。

现在来实现执行延迟调用的代码。这里使用的基本算法是枚举延迟调用列表中的元素，并且执行每个执行时间属性小于或等于当前时间的延迟调用对象。在完成相应的操作后，延迟调用对象会调用回调函数。因为这个列表是经过排序的，所以可以在遇到第一个执行时间属性晚于当前时间的延迟调用对象时跳出循环。

```
def ExecuteDeferredCalls():
    # run deferred method calls
    dCall = None
    now = time.time()
    while 1:
        dCall = deferredCalls.pop(0) # front of the list
        if dCall.time > now:
            # not time yet, put it back at front of the list
            deferredCalls.insert(0, dCall)
            break

        dCall() # execute the call, and any callbacks

    # now return how long in ms until next call
    next = None    # forever
    if dCall is not None:
        next = int((dCall.time - time.time()) * 1000.0)
        if next < 0:
            next = 0

    return next
```

ExecuteDeferredCalls()（执行延迟调用）函数会返回在执行下一个延迟调用前还需要等待的毫秒数。可以把这个作为像 select()一样的异步等待函数的参数。

3. 进行一次延迟调用

进行一次延迟调用几乎和进行一次普通的方法调用一样方便。主要的区别在于是怎样指定所调用的对象和方法的。此外，游戏的程序编写人员还必须决定是否提供一个回调函数来接收通知。

```
import deferred

def DoSomething():
    objId = 42 # a parrot, Norwegian Blue
    intensity = 9
    sincerity = 0
    deferred.Call(objId, 'WakeTheParrot',
        (intensity, sincerity), 100, Callback, (objId, ))

def Callback(objId):
    print 'Parrot %d is pining for the fjords.' % objId
```

上面的例子说明了如何向 42 号对象（这是一只挪威蓝鹦鹉）发出一次延迟调用。WakeTheParrot（唤醒鹦鹉）方法会在 100 毫秒以后被调用，强度（intensity）为 9，真实度（sincerity）为 0。这个操作完成后，鹦鹉的对象标识会被作为参数来调用回调函数，“Parrot 42 is pining for the fjords.（第 42 号鹦鹉非常向往海湾）”被打印出来。

4. 在主循环中调用

只需要参考最初的游戏循环伪码，就可以理解这些函数在事件驱动系统中的作用。WaitForRequests()函数会保持阻塞，直到它从客户端接收到一个请求或者下一个延迟调用的延迟时间过去了。ProcessRequest()函数调用 ExecuteDeferredCalls()，它会执行所有可以执行的延迟调用，并且返回下一次延迟调用前所需等待的时间。如果要让 ProcessRequests()把这个时间值返回给主循环，可以对主循环的伪码作如下修改：

```
mainloop()
{
    waitTime = 0;
    while(game_is_running)
    {
        // wait for incoming requests
        WaitForRequests(waitTime);

        // handle requests
        waitTime = ProcessRequests();
    }
}
```

5. 延迟调用带来的影响

延迟调用是为游戏服务器实现事件驱动系统的一个简单而灵活的方法。它的优势在于它容易理解和实现，并且适用于任何形式的异步调用。然而，它也有一些特有的限制。首先，调用方必须知道被调用的目标对象的标识，这在游戏这样的动态环境中会带来一些限制。其次，游戏中的事件往往会导致一些动作的执行，这些由给定事件发起的动作通常会随着游戏状态的改变而改变。要使用延迟调用模式，游戏的开发人员必须使用某些状态机从而在任何时刻都可以发出特定的调用。这可能需要加入复杂的条件代码，甚至连调试和维护都会变得更为困难。不仅如此，有些新的功能需要对给定的事件作出反应，加入这些新功能后，调用代码也必须被修改。因为延迟调用可以在游戏代码中的任何一点进行，无法找到一个独立的点来进行修改或调试错误。

3.5.3 事件广播

本文在这里对延迟调用模式所做的第一个改进是在它之上加入一个层次来为游戏事件实现广播（broadcast）机制。这里，服务端并不直接把事件发送给每个可以接收事件的对象，而是让这些对象注册它们感兴趣的事件，并且在事件发生时调用它们。这个模式增加了一些

高层的抽象概念：游戏事件（Game Event）、事件广播器（Event Broadcaster）、事件参数（Event Parameter）和事件处理器（Event Handler）。

1. 游戏事件

一个游戏事件只不过是一个标识，它可以表示游戏中发生的某件事情。它可以实现为一个整形常量标识，任何想要使用它的代码都可以从特定模块中导入它。下面的例子介绍了一个可行的方法。

```
# gameevents.py

ACTION_ATTACK = 1
ACTION_PLACE_OBJECT = 2
ACTION_TAKE_OBJECT = 3
```

这些常量的值除了独一无二地标识这个事件以外并没有其他的意思。使用这些事件的代码只需要导入这个模块并且使用名字来引用它们就可以了，如下所示：

```
import gameevents

myEvent = gameevents.ACTION_ATTACK
```

2. 事件处理器

事件处理器是任何可以调用的对象、函数或是方法，它们在某个特定的事件发生后被调用。任何类和模块都可以定义和实现事件处理器。

3. 事件广播器

事件广播器（EventBroadcaster）这个抽象概念的任务是把事件发送给对其感兴趣的对象。这些对象会为某个特定事件向事件广播器注册（register）一个事件处理器。当其他代码触发这个事件时，它们通过事件广播器来发送（post）这个事件。通过对某个函数进行延迟调用就可以实现这一功能，这个函数会把这些事件分发（dispatch）给所有注册过的处理器。下面的代码对一些重点进行了描述。

```
import deferred

class EventBroadcaster:
    def __init__(self, objId):
        self.id = objId
        self.handlers = {}

    def RegisterHandler(self, event, handlerRef):
        hList = self.handlers.get(event, [])
        if not hList:
            # handlers for new event
```

```
            self.handlers[event] = hList
        if handlerRef not in hList:
            # ensure only register once for a given event
            hList.append(handlerRef)

    def UnregisterHandler(self, event, handlerRef):
        hList = self.handlers[event]
        hList.remove(handlerRef)
        if len(hList) == 0:
            # remove unneeded entries
            del self.handlers[event]

    def PostGameEvent(self, event, delay, *args):
        deferred.Call(self.id, 'Dispatch', delay, args, None, None)

    def Dispatch(self, event, *args):
        handlerList = self.handlers.get(event, []):
        for handler in handlerList:
            apply(handler, args)
```

事件广播器的构造函数会赋予它一个对象标识，这样它就可以作为延迟调用模块的目标对象。如果需要的话，游戏中可以有多个事件广播器，但是每个事件广播器的作用域都必须小心地管理。self.handlers 属性是一个 Python 词典，它把事件映射到事件处理器的引用列表。

为了把一个事件处理器和一个事件关联起来，RegisterHandler()（注册处理器）方法会在 self.handlers 词典中寻找把这个事件作为索引（key）的事件处理器列表，然后把这个事件处理器加入到这个列表中。在这里使用列表是为了允许对同一个事件注册多个处理器。UnregisterHandler()（注销处理器）方法则通过把处理器从那个事件列表中删除来打破这个关联。

PostGameEvents()（发送游戏事件）方法会为这个事件进行一个延迟调用。这只不过是对 deferred.Call()方法的一个简单包装。这个延迟调用会调用事件广播器的 Dispatch()（分发）方法并把 self.id 作为第一个参数，这样就会把事件广播器自身作为目标对象。通过这种机制，程序员就可以使用延迟时间参数来延迟这个事件，就好像直接使用延迟调用模式一样。

4. 事件参数

为了保留最初的延迟调用模式的强大功能，事件必须是函数形式的。换句话说，它们应该可以接受参数以使事件包含更多意义。请参见上述 PostGameEvent()方法，注意最后一个参数：*args。这个星号（*）表示这个参数是一个变长参数列表，这使得程序员在调用这个方法时可以在那些显式定义的参数后面添加任意数量的参数。Python 把变长参数实现为一个元组，这样程序员就可以在由 deferred.Call()发起的调用链中把 args 参数透明地传递给事件处理器。

为每个事件定义一个逻辑签名，这样，编写事件处理器的程序员就可以知道应该为这个

处理器定义哪些参数。这对于避免那些由于把错误数量的参数传递给某个异步调用而导致的运行时错误来说非常重要。

```
# gameevents.py

ACTION_ATTACK = 1          # (attacker, defender, damage)
ACTION_PLACE_OBJECT = 2    # (actor, object, destination)
ACTION_TAKE_OBJECT = 3     # (actor, object, source)
```

使用字符串元组来声明这个签名是一种更为稳健的方法，它可以指定与事件相关联的参数。当注册一个事件处理器时，游戏开发人员可以使用 Python 的 inspect 模块[Python04]来比较事件处理器的形式参数和这个元组的元素。

5. 使用事件广播

现在本文已经说明了应该怎样实现这个事件广播机制了，下面用一个简单的例子来演示一下它的使用方法。首先应该为所关心的事件声明一个或多个处理器。

```
def HandleAttack(attacker, defender, damage):
   # a function-based handler
   print '%s attacked %s for %d damage.' % \
         (attacker, defender, damage)

class ObjectPlacementWatcher:
   # a class with method-based handlers

   def HandleTakeObject(actor, object, source):
      print 'Help! %s is taking my %s!' % \
                (actor, object)

   def HandlePlaceObject(actor, object, destination):
      print 'Thanks, %s, for the %s' % \
                (actor, object)
```

接着把这些事件处理器注册到事件广播器中，并传入实现这些处理器的函数或方法的引用。

```
import objMgr
import gameevents

# retrieve event broadcaster by id
eb = objMgr.GetObject(EVENT_BROADCASTER_ID)

# registration of function-based handler
eb.RegisterHandler(gameevents.ACTION_ATTACK,
                   HandleAttack)

# registration of method-based handlers
```

```
watcher = ObjectPlacementWatcher()

eb.RegisterHandler(gameevents.ACTION_TAKE_OBJECT,
                   watcher.HandleTakeObject)
eb.RegisterHandler(gameevents.ACTION_PLACE_OBJECT,
                   watcher.HandlePlaceObject)
```

现在，每当有游戏开发人员感兴趣的事件发出，处理器就会被调用。

```
# somewhere else in the game
import objMgr
import gameevents

# retrieve event broadcaster by id
eb = objMgr.GetObject(EVENT_BROADCASTER_ID)

eb.PostGameEvent(gameevents.ACTION_ATTACK, 0,
                 ATTACKER_ID, DEFENDER_ID, damage)
...

# at the very same time, elsewhere still...
eb.PostGameEvent(gameevents.ACTION_TAKE_OBJECT, 0,
                 ACTOR_ID, SOMEOBJECT_ID, sourceLoc)
```

对 PostGameEvent()函数的每次调用都会通过 deferred.Call()发出一个独立的异步调用。如果把 0 作为延迟时间参数传入，这个调用就会尽快执行。这些事件处理器会被独立地调用，就好像它们是同时执行的一样。

6. 事件广播带来的影响

广播模式使事件驱动系统更为强大。尤其是，游戏程序员成功地解耦（decouple）了事件发送者和接收者，这使得他们可以根据运行时的具体情况动态地创建和打破事件与处理器之间的关联，而不需要实现复杂的状态机或是笨拙的条件代码。不仅如此，在程序中还有一段惟一的代码，所有的事件都从那里分发，这样程序员就可以根据需要方便地添加日志和调试功能。

然而，这种方法也有其局限性。最明显的就是系统的广播器在每次接收到一个给定事件后，会调用所有注册了的事件处理器。在一个游戏里，仅当事件发生在特定的上下文中时，才会有人关心它们。考虑这样一个例子：有一个秘密的入口（portal），只有戴着一种特殊项圈的玩家才能通过。使用游戏当前的实现可以为 PORTAL_ENTRY（进入入口）事件注册一个处理器。每当有入口发送这个事件时，游戏处理器都会被调用，并判断这个事件是不是它所关心的入口发出的。事实上，大多数情况下这些事件和设计者的目的并不相关。如果游戏里有很多入口，但是只有一个是秘密的，那么这种做法效率非常低，它导致了很多无效的调用，而仅仅是为了支持偶然使用的秘密入口。

3.5.4 精确的事件广播

对事件驱动系统的最后一个改进可以达到更精确地注册事件处理器。使用这个模式不仅可以指定所关心的事件，还可以指定在什么情况下才会调用处理器。这可以把对处理器进行不必要调用的开销最小化。这个模式所引入的主要抽象概念是游戏事件索引（game event key）。

1. 游戏事件索引

前面的实现根据发生的事件来调用相应的处理器，也就是说，这种实现把事件本身作为选择事件处理器的判断标准（criteria）。游戏事件索引封装了与一个事件相关的信息，游戏开发人员可以把它作为选择事件处理器的判断标准。这些信息通常是那些在游戏系统的基本广播模式中作为事件参数来传递的值。为了简单起见，最好找出那些具有相同语义的公共参数类型，然后就可以使用这些公共参数类型来开发游戏系统的索引了。

譬如说，所有的事件都会由某个事件源（source）发出，它通常是一个对象。另外，大多数事件都与某个特定的目标（subject）相关，也就是事件中最让人关注的关键点。再加上事件本身，现在可以用 3 个值来判断我们是否对某个事件感兴趣。这样就有了下面这些可能的组合：

- 仅仅考虑事件；
- 事件和事件源；
- 事件和目标；
- 事件、事件源和目标。

虽然它也可以包含那些不包括事件本身的组合，但是这对游戏开发人员的目的来说并不是很有用。

词典数据类型是 Python 的一个强大功能，它使开发人员可以把任意一个不变的对象作为索引来映射到任何其他对象。这些索引可以是整数和字符串，如果用户定义类的实例可以被散列（hash）为某个特定的值，就可以把它作为索引。使用这个功能就可以创建一个映射，它把封装了上面这些标准组合的对象映射到一个事件处理器集合。下面用一个简单的例子来演示一下。

首先，在 gameeventkeys.py 模块中声明一个通用的游戏事件索引（GameEventKey）基类。

```
# gameeventkeys.py

class GameEventKey:
    # event key base class
    def __init__(self, event, src=None, subj=None):
        self.event = event
        self.src   = src
        self.subj  = subj

    def __eq__(self, other):
```

```
        # equality test
        return (self.event == other.event) and \
                  (self.src    == other.src)    and \
                  (self.subj   == other.subj)

    def __hash__(self):
        # hash function
        return hash((self.event, self.src, self.subj))

    def RegisterHandler(self, handlers, handlerRef):
        # this key maps to a dictionary of handlers
        hDict = handlers.get(self, {})
        if not hDict:
            handlers[self] = hDict
        hDict[handlerRef] = 1    # dictionary as a set

    def UnregisterHandler(self, handlers, handlerRef):
        # this key maps to a dictionary of handlers
        hDict = handlers[self]
        del hDict[handlerRef]
        if len(hDict) == 0:
            # remove unneeded key entries
            del handlers[self]

    def Lookup(self, handlers):
        # return the dict for this key, or an empty one
        return handlers.get(self, {})
```

这个类可以作为一个事件处理器判断标准的散列容器。__eq__()和__hash__()方法是Python钩子，可以说明应该怎样把游戏事件索引类的实例用作词典类的索引[Python02]。通常，Python通过比较对象的标识来比较两个实例；换句话说，如果两个引用指向同一个对象，那它们就相等。然而，游戏的设计希望能把包含了相同数据的游戏事件索引也认为是相同的，因此系统实现__eq__()来达到这个目标。同样，基于对象的内容，游戏实现了__hash__()，它返回了由类属性组成的元组的散列值。现在，游戏系统可以创建一个游戏事件索引类的实例并且把它作为索引来向一个词典插入值，然后使用另一个具有相同内容的实例来获取这个值。

目前这个类中已经实现了RegisterHandle()（注册处理器）和UnregisterHandler()（注销处理器）。游戏事件索引的不同实现可能会使用不同的方法来注册处理器。这里，游戏的系统在基类中提供了一个缺省实现，但是也允许派生类在需要时重新实现它们；同样，它们也可以重新实现处理器查找机制（目前由Lookup()（查找）完成，它由事件广播器的Dispatch()（分发）方法直接调用）。在这里，例子中的模式并没有像在前面的模式中那样把某个特定事件索引相对应的处理器引用保存在一个列表中，而是把一个词典作为集合来使用。这样做可以有效地确保与每个索引相对应的事件处理器不会重复。

2. 事件索引派生类

如果程序员想要使用很多判断标准来注册事件处理器，为了实现这些不同的组合，可以从基类中派生出以下这些类。

```
class ByEvent(GameEventKey):
    def __init__(self, event):
        GameEventKey.__init__(self, event)

class ByEventAndSource(GameEventKey):
    def __init__(self, event, src):
        GameEventKey.__init__(self, event, src)

class ByEventAndSubject(GameEventKey):
    def __init__(self, event, subj):
        GameEventKey.__init__(self, event, None, subj)

class ByEventSourceAndSubject(GameEventKey):
    def __init__(self, event, src, subj):
        GameEventKey.__init__(self, event, src, subj)
```

这些类让游戏的程序编写人员可以以一种方便而易于理解的方式来指定他们希望使用的那种标准来注册事件处理器。注意 ByEventSourceAndSubject（通过事件源和目标）类本质上与基类相同，但这纯属巧合。为了确保一致性，永远都不要直接使用游戏事件索引这个基类来注册事件。

3. 基于索引的事件广播

对事件广播器类加以改进以满足新设计的需要。前面只使用事件作为 self.handlers 词典的索引，现在的模式要使用游戏事件索引类的实例来作为索引。下面是相应的修改。

```
import deferred
import gameeventkeys

class EventBroadcaster:
    def __init__(self, objId):
        self.id = objId
        self.handlers = {}

    def RegisterHandler(self, key, handlerRef):
        key.RegisterHandler(self.handlers, handlerRef)

    def UnregisterHandler(self, key, handlerRef):
        key.UnregisterHandler(self.handlers, handlerRef)

    def PostGameEvent(self, key, delay, *args):
        kArgs = (key, ) + args
        deferred.Call(self.id, 'Dispatch',
```

```
                                        delay, kArgs, None, None)

    def Dispatch(self, key, *args):
        handlerList = \
            gameeventkeys.GetHandlers(key, self.handlers)
        for handler in handlerList:
            apply(handler, (key, ) + args)
```

注意现在的 RegisterHandler()和 UnregisterHandler()直接委派给作为索引（key）参数传入的游戏事件索引实例。同样地，Dispatch()方法也把获取处理器列表的功能委派给 GetHandlers()（获取处理器列表）函数，它使用索引来作为查找标准。对 PostGameEvent()方法惟一的修改是它现在接受一个事件索引参数，这个参数会被传给对 Dispatch()方法的延迟调用。

4. 使用游戏事件索引来注册事件处理器

现在，注册和注销事件处理器变得更为精确了。游戏开发人员不仅可以指定当某个特定事件发生时我们的处理器会被调用，还可以规定发出这个事件的对象（事件源）和这个事件涉及的对象（目标）。譬如说可以按照下面的形式修改事件处理器。

```
def HandleAttack(key, damage):
    # a function-based handler
    attacker = key.src
    defender = key.subj
    print '%s attacked %s for %d damage.' % \
                (attacker, defender, damage)

class ObjectPlacementWatcher:
    # a class with method-based handlers

    def HandleTakeObject(key, source):
        actor = key.src
        object = key.subj
        print 'Help! %s is taking my %s!' % \
                (actor, object)

    def HandlePlaceObject(key, destination):
        actor = key.src
        object = key.subj
        print 'Thanks, %s, for the %s' % \
                (actor, object)
```

注意事件索引（key）是怎样取代我们早期处理器实现中的某些参数的。那些数据仍然都在，但是它们现在包含在事件处理索引对象中，这个对象由事件发起者通过调用 PostGameEvent()函数提供。

现在可以使用更精确的标准来注册这些处理器。这里的模式希望 HandleAttack()处理器只在以 ATTACKER_ID 为标识的进攻者（事件源）发起进攻后才会被执行。同样，在放置物品时，下面的模式只关心物品（目标）标识是 SOMEOBJECT_ID 的情况。

```
import objMgr
from gameevents import *      # all event ids
from gameeventkeys import *   # all event keys

# retrieve event broadcaster by id
eb = objMgr.GetObject(EVENT_BROADCASTER_ID)

# register a function-based handler to fire when a
# specific attacker attacks
aKey = ByEventAndSource(ACTION_ATTACK, ATTACKER_ID)
eb.RegisterHandler(aKey, HandleAttack)

# register method-based handlers to fire when a
# specific object is manipulated
watcher = ObjectPlacementWatcher()

tKey = ByEventAndSubject(ACTION_TAKE_OBJECT,
                         SOMEOBJECT_ID)
eb.RegisterHandler(tKey, watcher.HandleTakeObject)

pKey = ByEventAndSubject(ACTION_PLACE_OBJECT,
                         SOMEOBJECT_ID)
eb.RegisterHandler(pKey, watcher.HandlePlaceObject)
```

5. 使用游戏事件索引来发送游戏事件

现在，在发送游戏事件时服务端必须创建一个游戏事件索引类的实例。之所以要这样做是因为它并不知道游戏中使用了哪些索引来注册事件处理器，因此它必须提供所有的判断标准。

```
# somewhere else in the game
import objMgr
from gameevents import *      # all event ids
from gameeventkeys import *   # all event keys

# retrieve event broadcaster by id
eb = objMgr.GetObject(EVENT_BROADCASTER_ID)

aKey = GameEventKey(ACTION_ATTACK,
                    ATTACKER_ID, DEFENDER_ID)
eb.PostGameEvent(aKey, 0, damage)

...

# at the very same time, elsewhere still...
tKey = GameEventKey(ACTION_TAKE_OBJECT,
                    ACTOR_ID, SOMEOBJECT_ID)
eb.PostGameEvent(tKey, 0, sourceLoc)
```

6. 获取事件处理器

这里还有一个重要的细节需要处理：怎样使用游戏事件索引来获得与给定事件相关联的所有事件处理器。回忆一下，在发送事件时，游戏系统使用了一个完整的游戏事件索引，它包含了关于特定事件调用的所有信息：事件、源对象和目标。现在需要获得按照不同的判断标准组合注册的所有事件处理器。

然而，调用 PostGameEvent()函数时所传入的游戏事件索引对象产生的散列值是根据所有判断标准计算出来的。要获得那些根据判断标准的子集注册的事件处理器，必须把接收到的游戏事件索引对象强制转换为某个表示系统所支持的有效组合的派生类，并且使用这些派生类来获取事件处理器。

这个操作是由 GameEventKey 模块的 GetHandlers()函数完成的，正如前面已经介绍过的，这个函数由事件广播器的 Dispatch()方法调用。

```
def GetHandlers(key, handlers):
    # retrieve a list of handlers for all possible
    # combinations of key criteria
    hDict = {}
    hDict.update(ByEvent (key.event).Lookup(handlers))
    hDict.update(ByEventAndSource(key.event,
                        key.src).Lookup(handlers))
    hDict.update(ByEventAndSubject(key.event,
                        key.subj).Lookup(handlers))
    hDict.update(ByEventSourceAndSubject(key.event,
                        key.src, key.subj).Lookup(handlers))

    # since hDict is used as a set, the data we need
    # is in the keys, not the values
    return hDict.keys()    # list of handler references
```

这个函数为游戏事件索引类的每一个派生类创建一个实例，并且使用每个实例的 Lookup()方法在传入的 handlers（事件处理器词典）参数中查找事件处理器。Loopup()方法返回一个事件处理器引用的词典，这个词典可以作为集合来使用。这意味着有意义的数据是这个词典中的键而不是它的值。这个函数会把这些事件处理器引用添加到临时词典对象 hDict 中，由它使用 update()方法来进行数据合并。最后，如果想要返回一个包含了事件处理器引用的列表，可以通过调用 hDict.keys()来获得。

这个函数说明了另一个基本问题。如果使用不同的游戏事件索引来注册某个事件处理器，当使用特定的游戏事件索引触发事件时，可能会多次出现同一个事件处理器。譬如说，系统可以使用 ByEventAndSource（使用事件和事件源）来注册一个事件处理器，同时还使用 ByEventAndSubject（使用事件和目标）来注册它。如果一个特定事件的事件源和目标与用于注册的两个游戏事件索引相匹配，这个处理器就会被找到两次。很明显，为同一个事件两次调用这个事件处理器是不恰当的，因此这里使用临时词典对象 hDict 来确保这个事件处理器在 GetHandlers()函数返回的列表中只会包含一次。

7. 使用游戏事件索引带来的影响

事件索引模式使游戏开发人员可以对事件处理器的注册进行微调。与前两种模式相比，它可以让游戏管理人员进行更多的控制。不仅如此，当游戏管理人员需要使用大量的事件处理器时，事件索引模型在算法上也比通常的事件广播机制更为高效。假设要为某个特定的事件注册 N 个处理器，如果不使用事件索引，就必须进行 O(1)+O(N)次查询和 N 次方法调用。与此相反，如果为某个特定的事件索引注册了 N'个处理器，就必须进行 O(1)*M 次查询和 N'次方法调用，这里 M 表示的是游戏事件索引类所具有的派生类总数。几乎可以肯定，在使用精确的判断标准进行注册时，N'会比较小。在运行时，对于特定事件注册的事件处理器的总数 N 可能会很大，而事件索引派生类型的数量 M 仅当我们需要加入新功能时才会增长，因此这样做是值得的。

不过这个模式还是有一些限制的。如果一个游戏具有多种不同的事件，并且它们的参数类型也不尽相同，那么就很难为它们提供一个共有的抽象概念（就好像事件源和目标之类的判断标准一样），这会导致开发人员必须为游戏事件索引类创建大量的派生类。这个问题可以通过仔细地定义事件的作用域来避免，正如上文在设计类和方法时所做的那样。

还有一个要考虑的问题是在事件索引的属性中保存对象引用时必须非常小心。还记得 Python 使用的引用计数方式吗？一旦事件索引被注册到处理器词典中，它们就会保留在这个词典中，这些事件索引会导致某些对象不能被如愿地销毁。这个问题同样适用于那些实例方法的事件处理器，因为拥有指向这种方法的引用也会增加这个实例的引用数。这些问题可以靠精心的设计来解决，譬如说使用 Python 内建类型而不是对象来作为事件索引属性，并且注意游戏中对象的生命期。

3.5.5 总结

基于事件的程序设计对于在游戏服务端模拟并发来说，是一种非常有效的机制，它没有使用多线程所带来的复杂度。Python 的动态本质，以及它的弱数据类型和高级结构，使得它非常适合为 MMP 游戏服务器实现基于事件的系统。延迟调用是一种在底层实现事件的有效方式，而事件广播方法为游戏提供了更大的灵活度。在广播模式中加入事件索引的概念使游戏开发人员可以对事件处理进行更细致的控制，从而获得一个可以满足他们对稳健性、动态游戏模式的需求的游戏系统架构。

3.5.6 参考文献

[MSDN01] "Asynchronous Procedure Calls," Microsoft Platform SDK Documentation, http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/asynchronous_procedure_calls.asp.

[Ousterhout96] Ousterhout, John, "Why Threads Are A Bad Idea (for most purposes)," an Invited Talk at the 1996 USENIX Technical Conference, January 25, 1996, *http://home.pacbell.net/ouster/threads.ppt*, 1996.

[Python01] van Rossum，Guido，"Objects，values and types，" Python Reference Manual，*http://www.python.org/doc/2.2.1/ref/objects.html#l2h-18*，April 10，2002.

[Python02] van Rossum，Guido，"Basic customization，" Python Reference Manual，*http://www.python.org/doc/2.2.1/ref/customization.html*，April 10，2002.

[Python03] van Rossum，Guido，"Built-in Functions，" Python Library Reference，*http://www.python.org/doc/2.2.1/lib/built-in-funcs.html*，April 10，2002.

[Python04] van Rossum，Guido，"inspect — Inspect live objects，" Python Library Reference，*http://www.python.org/doc/2.2.1/lib/module-inspect.html*，April 10，2002.

3.6 在 MMP 游戏中实现移动和物理模块的注意事项

Jay Lee, NCSoft
jlee@ncaustin.com

开发 MMP 游戏是一项极具挑战性的任务。成千上万的并发玩家期待着可以在同一个虚拟世界中流畅地移动，并且与其他玩家、生物或是物品以一种可信的方式进行交互。个人电脑和游戏机上的单机游戏或是小型多人游戏不断地在“夸张的”玩家移动、真实的物理以及对地形进行逼真的改变（通常通过破坏地形和建筑物来实现）等方面追求极限，这些都可以让玩家体验到很高的自由度，从而使他们对 MMP 游戏具有更高的期望。

MMP 游戏开发者实在无法抗拒日益逼真的游戏环境和交互所带来的诱惑。每个新发布的 MMP 游戏通常都会在至少一个具有重要意义的方面挑战极限从而与竞争者有所区别。然而，真正地在发布时成功做到这点的 MMP 游戏很少，这就说明了追求极限所带来的压力可能会导致游戏的最终失败。

本文的目的是要介绍与 MMP 游戏中的移动和物理系统相关的各种问题。理解这些问题才能更好地为发布引人入胜的游戏做好准备，并且可以避免由于以一个幼稚的，过于冒进的方式来为这些系统进行建模所引起的无法预料的后果。

3.6.1 我们可以发布这个游戏了吗?

这个问题的关键在于，开发人员总是面临着发布游戏所带来的压力，他们必须以一种及时而经济的方式来实现这点。毫无疑问，这必然导致他们在实现用于支持游戏的底层系统时必须作出很多折衷。开发人员必须对这些系统能够做什么加以限制。否则，这个游戏永远都不会完成。

另一方面，在游戏模式中加入过多的限制会降低游戏的真实性。如果一个玩家觉得自己和游戏的关系非常松散，他就很可能会取消订阅。开发人员当然希望能够避免这种情况的发生。

同样，开发团队不能就所作的折衷达成一致时也会非常危险。设计人员可能会觉得这些限制使游戏世界变得死气沉沉，并且无法让玩家在交互中获得一种融入其中的真实感受。游戏世界构建人员也许能理解这些限

制，但是他们可能会不停地寻找那些可以突破限制的方法来创建一个更具交互性或是在视觉上更为有趣的游戏环境。虽然这些想法都是正确的，但是它们会影响到整个团队最后发布产品的能力。

必须以一个平衡的方式来看待这个问题：在移动和物理模块中实现足够的真实性从而让玩家能够持续地参与游戏，并且加以足够的限制使得这个游戏可以以一种及时而经济的方式被实现。让所有团队成员对此达成共识非常重要。这不仅会减轻总体压力，还会减少返工或是长期维护带来的麻烦，程序员的创造力也可能会因此而增加。

3.6.2 这是一场战争

本质上，在 MMP 游戏中进行移动和物理方面的开发会受到一类特定玩家的影响，本文把他们称为“恶意（rogue）玩家”。这类玩家乐于使用任何必要的方式以在游戏世界中获得优势。他们会试图以比游戏规则所允许的方式更快地进行移动，或是去那些他们不应该到的地方，甚至去发掘那些不曾预料到的物体-环境交互。游戏管理人员必须解决这些恶意玩家，否则就不得不去面对游戏中可能出现的无政府状态。

游戏做得越成功就越有可能成为恶意的玩家的目标。这些人的数量远大于游戏开发员工数量，他们通常具有与开发游戏技术人员相同的才能，有时可能会更高。更可怕的是，恶意玩家们有时能够花费大量的时间来试图攻击游戏。一旦某个恶意玩家攻击成功，所有诚实的玩家都会受到伤害。这种行为反过来会诱使那些原本诚实的玩家也采用某些攻击手段来重新获得公平的地位。当支持人员发现有人欺骗，就会封锁这个玩家的账户，从而导致游戏失去一个付费用户。最终，这会成为一个恶性循环。如果开发人员在实现游戏时对所做的决定进行仔细考虑，就可以显著地减少这类问题。

因此，在对移动和物理的设计和实现的各个方面进行评估时，都必须考虑到恶意玩家的存在。游戏开发人员必须考虑那些有时间和能力去反汇编并且修改客户端的代码、游戏资源、网络数据包的人会怎样使用这些信息在游戏中进行欺骗。

相对于玩家都是诚实的情况来说，这毫无疑问会让本身就极具挑战性的工作变得更为困难。根据市场上已经发行的游戏的经验，如果开发人员认为他们的游戏不会受到这样的攻击或是只会吸引那些诚实玩家的话，那是非常愚蠢的。必须准备好建立一条防线，这就顺理成章地引入下一个话题：防御恶意玩家的第一条防线，也是最主要的防线。

3.6.3 服务端永远是对的

在移动和物理中必须考虑的第一个问题是客户端自治。在目前的 MMP 游戏环境中，每个玩家的计算机都具有大量的 CPU 资源和硬件加速能力，在游戏中尽可能地使用这些运算能力是非常有用的。然而，当服务端需要确定玩家应该移动到什么位置以及他们应该怎样和环境交互时，就必须建立一条防线。游戏系统应该通过客户端代码来让玩家感觉到他们在进行控制，而对这些事物的最终裁决则应该由服务端来进行。

为了让玩家可以在游戏世界中流畅地移动，游戏客户端必须立即响应玩家的移动请求并且根据客户端的地形和物体表示对这些请求做出判断。如果玩家要向前移动，并且客户端代

码确认没有障碍物阻止玩家进行移动（譬如说不能穿越的地形或是会发生碰撞的物体），他才可以移动。每一帧都会进行常规的判断，一旦玩家与某个障碍物发生碰撞，他会立即停止。如果某个玩家故意地修改客户端游戏代码或是游戏世界中的几何信息以访问游戏世界中他不应该进入的区域，服务端就必须检测到这些违例行为并且立即在服务端对游戏世界的表示中进行修正，然后把这些修正广播给每一个需要知道这些信息的客户端。

譬如说，在服务端和客户端都有对游戏世界的详细表示，其中有一个由很多玩家的房屋组成的村庄。当一个玩家在村庄中转悠时，如果没有某间房屋的钥匙，他就不能进入这个房屋，而只有房屋的主人才可以给他钥匙。如果服务端代码假设玩家进入房屋的惟一方法是使用钥匙，那么就很容易受到攻击。一些恶意玩家会通过修改客户端代码来绕过这样的判断从而把他们的角色放在房屋内部，这样就可以模拟一次合法进入的结束情况。服务端必须判断一个成功地把他的角色放在另一个玩家房屋中的恶意玩家会不会顺手牵羊拿走房屋内的某件物品。

一旦服务端检测到一次非法进入，它至少必须把那个玩家的位置修改为一个在房屋以外的已知合法位置。这个位置可以是这个玩家在房屋以外时最后的合法位置，也可以是一个已知的位置，譬如说房屋的门阶。无论采用哪种方式，一旦服务器发现了这一情况，那么当玩家试图请求获取房屋中的某件物品时，这个请求应该被拒绝，因为玩家实际上是在房屋外面。不仅如此，周围的其他玩家都应该看到这个玩家在房屋外面。说实话，游戏的服务端并不关心在那个恶意玩家的（被修改过的）客户端上这一切看起来会是什么样的。以下安全措施也可以用来预防这类可能的侵入。

- 先确认某个玩家的进入是否合法，然后才把房屋的内容发送给这个客户端。
- 要求把房屋中的物品都保存在一个固定的保险箱里，并有一把单独的钥匙。
- 把进入房屋的行为实现为把玩家传输到另一个区域中，这样玩家就不能仅仅通过移动来进入房屋。

总之，服务端永远是对的。服务端必须具有内建的修正机制来防范那些任性的玩家。此外，服务端的仿真必须具有确定性，它必须对每一个玩家行动的处理都应该产生一个清楚明白的状态。譬如说，一旦服务端得知某个玩家掉下一个致命的悬崖，它会立即把这个玩家移动到这个悬崖的底部并且认为他已经死亡。而客户端会根据服务端确定的最终状态提供相应的视觉表示。

3.6.4 移动的代价

70%的 MMP 游戏带宽与移动有关。因为带宽是 MMP 游戏中一项重要的持续成本，努力地减少这方面的需求可以节约大量的金钱。不仅如此，如果可以提高一个游戏的数据传输效率，就可以让更多的硬件有能力运行这个游戏，从而拓宽潜在的用户群。

在最快情况下，客户端能够以和帧频一样的速度向服务端发送移动消息，这个消息描述了玩家的新位置。保守估计的话，这意味着每秒钟有 15 个移动数据包，因为 15 帧/秒已经接近一个游戏能够做出正常反应的极限了。假设每个移动数据包的大小是 20 个字节，每个玩家每秒会产生 300 字节的移动数据。如果游戏中每个客户端可以同时看到 20 个玩家，简单的移动所导致的数据传输率就会提高到 6kbytes/s。这已经超过了一个 56k 调制解调器的理论极限，而我们所实现的功能只不过是让玩家像雕像一样四处移动。

很明显，开发人员必须要寻求一个更好的方法。这个方法可以从降低移动数据包的发送频率开始。15 次每秒的更新频率对于服务端验证这个玩家的移动是否合法来说的确是高了点。客户端仍然可以产生这些数据包，但是它可以使用一个名为“暂时频道（ephemeral channel）”的方法，只有那些在一个更新周期中最新的移动数据包才会被真正地发送给服务器。这对于瞬时信息（譬如说移动数据）来说非常有用。玩家位置最精确的表示就是这个客户端最新生成的数据包。

每当服务端收到移动数据包时，它会根据来自这个客户端的前一个移动数据包做出判断。随后服务端确认从前一个移动数据包的位置到当前移动数据包的位置之间的移动。为了节约 CPU 时间，服务端可能只对这两个位置之间的几个特定点进行采样。如果这些采样点以及最终位置都是合法的，那服务端就会确认这个玩家进行了一次有效的移动。一旦检测到一个无效的位置，服务端会向这个客户端发送一个修正数据包。同时也会把这个修正数据包作为这个玩家的实际移动情况发送给附近的客户。

使用这样的技术最重要的就是要确保在两个位置之间的采样点距离应该小于游戏中最小的碰撞/触发体（collision/trigger volume）。譬如说，如果两个采样位置之间最大不超过 1.2 米，那么所有的墙壁、陷阱、瞬间移动门户等玩家可能会被阻挡或是触发的物体的尺寸都必须超过 1.2 米，这样才可以确保玩家不会穿过它。图 3-12 中的玩家在不经意间就穿过了一个一米宽的障碍物，因为两次移动更新之间的采样点以及这两次更新的位置本身都是合法的。

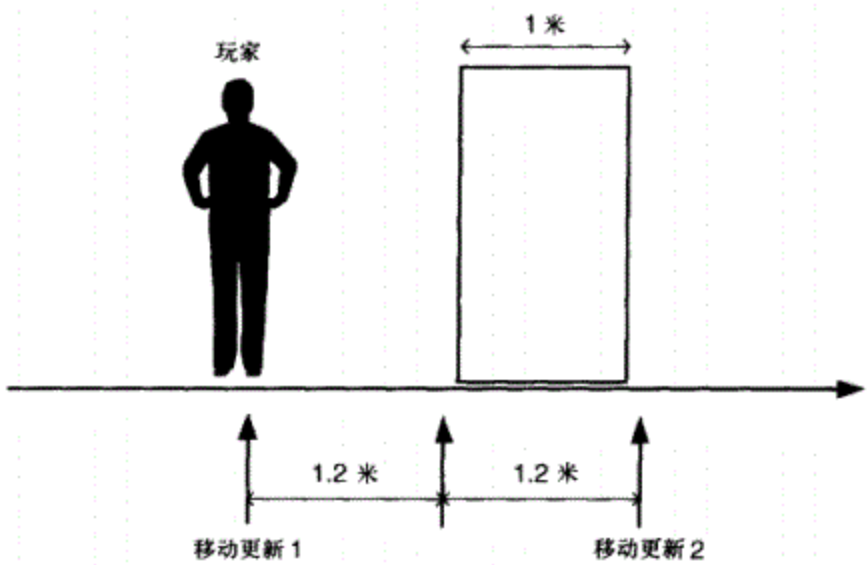


图 3-12　玩家穿过了一个小于采样距离的障碍物

此外，还应该为一个玩家角色在两次移动更新之间可以移动的距离设置一个上限。考虑到因特网的不一致性和延迟性，游戏的设计允许在移动的外观上有一些变化，这也可以减少修正带来的影响。然而，设置一个上限可以确保玩家不能滥用这一特性来获益。譬如说，如果玩家角色的最大移动速度（后面将会对此进行更详细的讨论）是 6 米/秒，而工作人员每秒对服务器更新 4 次，这样设计人员可能会把两个数据包之间的移动上限设为 2 米。任何超过这个范围的移动都会被修正后发回给客户端。这不但确保考虑了足够的与因特网相关的变化，也可以确保恶意玩家无法使用手工制作的数据包在游戏世界中四处乱窜。并且，为了使其更

为有效，必须采用某种欺骗检测机制来记录玩家移动超过最大角色移动速度的历史。这有助于查明哪些玩家的非法移动在统计上是不可接受的，从而使游戏管理人员可以对他们进行更细致的监视（参见侵入监测）。

3.6.5 移动速度

如果一个游戏设计允许玩家使用不同的移动速度（譬如说跑步和行走），并且能使用魔法或其他方式来提高移动的速度，游戏世界就会变得更为有趣。游戏设计还可以让玩家角色能够通过加速来加快步伐或者减速来放慢脚步甚至停下来，从而使玩家对游戏更加着迷。

无论如何，一个游戏设计也不能让客户端自主地决定一个玩家角色的移动速度。各种在线游戏中都曾有玩家成功地改变了他们的移动速度，他们会使用修改过的程序来加速客户端游戏中的动画播放速率或是帧频。

玩家们总是想要减少令他们觉得无聊的时间，并希望能够以允许的最快速度进行移动。在大型游戏世界中更是如此，因为在不同位置间进行移动需要花费大量的时间。

此外，设计人员应该知道在游戏中允许提高移动速度会导致系统在对玩家角色移动进行验证时为错误留下更大的余地。譬如说，在《天堂》（*Lineage*，[LIN]）中，玩家可以通过各种很容易获得的药水来加快移动速度。不仅如此，角色还可以变形为另一种生物来加快移动速度。这些选项的组合再加上网络延迟所带来的影响会导致移动速度具有很大的变化。在这种情况下，玩家常常会试图修改客户端来获取速度上的优势，由于允许的变化范围很大，系统很难检测出这种尝试。

这里有一种保守的方法，那就是在游戏世界中只允许以两种速度移动：快速移动和慢速移动。当玩家想要在某些环境中对他们的移动距离进行精确控制，或是想要准确地指出他们的目的地而不想超过时，他们会使用较慢的移动速度。在其他情况下，他们会使用较快的移动速度，这个速度必须足够快以使玩家在游戏世界中快速移动时能够感觉到明显的进展。这样就可以把玩家的移动速度变化保持在某个预定的基本值附近。验证一个玩家角色是否以允许的速度进行移动的代码将会更容易实现，因而不容易受到侵入的影响。

游戏中的非玩家角色（NPC）也可以使用变化的移动速度，因为对他们的控制完全是由服务端进行的。首选方法是把NPC的移动速度调节到和玩家角色的标准移动速度一致，而不是让玩家能够自主地改变他的标准移动速度。

另一种方法是把游戏世界的尺寸按比例缩小。设计人员没有理由把游戏中的地形和空间做得很大并且要求玩家花费相当长的时间作来回移动。游戏的设计应该把玩家在有趣事件之间的无聊时间降到最低限度。很多现存的MMP游戏都会给人一种空荡荡的感觉，开发人员通常会实现特定的机制来使玩家在游戏世界中的移动不会过于沉闷。而按比例缩小游戏世界的尺寸并且增加有趣场所的相对密度可以让这种感觉大大降低，开发人员也不必再为此开发特定的机制，这可以节约不少开发时间。

3.6.6 玩家可以从这里到那里吗?

如果假定玩家具有完全的自由，那么要对玩家在游戏世界中的移动加以限制会变得非常

困难，除非在游戏中强加一些限制来阻止移动。

1. 游戏世界中的移动

这个方法需要游戏世界构筑人员坚持一系列的规则，譬如说“如果不希望玩家到达那里，就让这个斜坡大于 60° 。”或是在游戏世界中放置障碍物来隔开某个区域。随着测试人员和玩家发现各种方法来进入那些禁止入内的区域，他们必须持续作战来填补现存的漏洞，然而他们并不能立即确定这个问题是否真的被修复了。

如果在开始设计游戏世界时先假定玩家哪里都不能去，问题就会变得相对简单。随后游戏世界构筑人员就可以通过开辟一条玩家角色可以行走的道路来指定那些可以合法移动到的位置。这么做可以使检验非常容易，因为当玩家不能到达某个位置时情况会很明显：他们在去那里的路上停下来了。

2. 瞬间移动

瞬间移动是一种非常方便的游戏机制，几乎每个 MMP 游戏都会使用它。对于玩家角色来说，这是一个逃避危险或迅速移动到另一个位置的好方法。还有一种类似于瞬间移动的机制可以帮助那些被卡住一定时间的 NPC 摆脱困境。这有助于处理那些游戏世界中的故障点。譬如说，玩家可以把游戏中的怪物引诱到某个地方以使它们无法动弹，这样他就可以随意攻击而不必担心会受到伤害。

然而，每当一个角色开始瞬间移动，游戏开发人员就面临一个问题：怎样确定这次瞬间移动的有效目的地。如果游戏实现了角色之间的碰撞（也就是说，角色不能在同一位置重叠），这个问题就会变得更难解决。要实现一个能够在具有可移动和不可移动物体的任意三维世界中可靠地发现一个有效位置的算法不仅很有挑战性，而且还很容易出错。

即使游戏设计把瞬间移动限制在一些预定的位置，譬如说两个瞬间移动装置之间，仍然必须确定角色到达时应该放在什么位置。游戏设计人员可以通过为每个瞬间传输装备标记到达点来简化这个问题，每个到达点只能被来自某个方向的瞬间传输角色访问。这个方法意味着需要面对的问题只剩下一个了：当一个玩家试图瞬间移动时，所有有效的到达位置可能都已经被占用了。

然而，从游戏特性的角度来看，以这种方法来实现瞬间移动会受到很多限制，并且不能解决玩家角色在登入游戏后怎样在游戏世界中出现的问题。如果游戏世界中的玩家之间没有碰撞，网络就可以更为稳健地实现瞬间传输。可能发生的最坏情况就是角色最终会和另一个角色重叠。通常玩家会很快地进行调整并且移走。

3. 平移动画（translating animation）

瞬间传输问题的一个变形是有些玩家行动可以让角色的最终位置和他的起始位置不同。这些移动通常不是任意的：这时电脑播放器会播放一个特定的动画并且把角色在 3 个轴（也就是 x，y，z）中的一个或多个上进行平移。这些移动可能包括跳跃、翻筋斗或是一些夸张的武术动作。服务端进程通常会认为这些是特殊的移动类型而把它们和通常的移动区分开。通过从动画中提取移动信息，服务端能够处理这些移动请求并且以服务端对游戏世界的表示为背景对这个平移进行正确的处理。随后服务端就能够确定移动的新位置，并判断这个平移

是否会被障碍物打断，譬如说墙或者门。

游戏开发人员可以根据游戏的需要决定是否要等待服务器确定后才播放动画，从而提供更精确的表示。当然，有时给予玩家及时的响应也许更为重要，这样做的代价是有可能要进行很大的修正。这里还有一种方法，那就是让客户端判断这个动作是否能够成功完成，如果不能，就让玩家的动作请求失败，相对来说，前两个方法看上去更为合适。譬如说，如果有一个“夸张的”武术动作可以让一个角色的位置向前移动 5 米，系统可能会选择在获得服务器的确认后才开始显示相应的动画，并且最终玩家会撞到 3 米外的门上而被迫停止。系统也可以选择立即显示这个动画，这样玩家角色就有可能会在服务器告知他正确的终止位置前略微移动到门里面。对于玩家来说，这两个选择都比因为这个角色无法按照动画定义的那样向前移动 5 米而不能进行这次移动要好。

3.6.7 碰撞检测

通常，确定一个游戏中的物体是否挡住其他物体的移动路径的过程被称为碰撞检测。碰撞检测并不是在 MMP 游戏中才有的问题，它在 MMP 游戏中的重要性也并不比在其他游戏中的重要性低。这里介绍的大部分方法原先都是在单机游戏或是小型多人游戏中实现的。

碰撞检测的计算开销非常大，这是因为所需进行的测试数量会随着所需测试的对象数量的增长呈几何级增长。碰撞测试的基本方针是尽量把在任何时刻必须进行碰撞检测的物体数量最小化。如果游戏设计让开发人员可以使用一个根本不需要任何碰撞的游戏世界而不会影响玩家对游戏的投入，那么他们应该为此高兴并且利用这一优势。然而，通常并不是这样，因此本文将对这个问题进行更加细致的研究。

1. 碰撞体（Collision Volumes）

在实现碰撞检测时，游戏的设计通常使用简单的几何体（Geometric Volume）来表示游戏世界中的各种物体和角色。对简单几何体（譬如说球体、长方体、胶囊体（Capsule，圆柱体的两端各接一个半球）和圆柱体）之间的交叉进行计算要比确定两个复杂物体之间的交叉简单得多。由于典型的 MMP 游戏在运行时需要大量的碰撞检测，为了性能而进行这样的简化是很有意义的。

毫无疑问，简单几何体很可能只是游戏中所出现物体的粗略近似。然而，这个问题是否真的像乍看上去那么大？拿一棵树做例子。树在游戏中的作用是什么？当然是用来构造游戏世界并使其看上去更加生动。不仅如此，它可能还会被用作障碍物来限制角色的移动，正如在真实世界中的树一样。如果树和角色所使用的碰撞模型和它们在真实世界中的外观非常接近，玩家在移动时将会遇到不少麻烦。当玩家在森林里四处走动时，他经常会被树枝和树根的突出部分挡住，甚至连叶子都会在玩家经过时缠住他的肩膀。虽然这要比只使用简单的几何体来进行碰撞的游戏世界更加真实，但它事实上会使这个游戏更为无趣。使用和树干一致的圆柱体来表示一棵树是一个更好的方法，它可以让玩家移动起来更为方便。或许玩家的脑袋偶尔会和树叶或树枝交叉，但是相比在移动时被意外的中断来说，玩家不太会注意这些视觉上的问题。

在游戏世界允许的情况下，开发人员可以使用另一种常见的技术来提高碰撞的效率和精

确度。游戏开发人员可以用一个简单的几何形状来构建碰撞体并把它用于粗略的判断。一旦这个粗略的判断确定这两个物体发生了碰撞，开发人员就会使用更加细致的几何体来进行精确的判断。譬如说，如果有两艘宇宙飞船在太空中飞行，游戏的设计可以简单地使用外接球体来表示它们。一旦这两个球体相交了，这两艘飞船就可能发生碰撞，这样就可以继续对那些表示飞船各个部分的更小的几何体进行判断。这样不仅可以确定碰撞是否真的发生了，还可以精确地知道碰撞发生在什么地方，从而利用这一功能来指出损坏的位置。

2. 角色碰撞体

那应该怎样在碰撞中表示一个玩家角色呢？显然可以从引擎所支持的基本几何体（譬如说球体、圆柱体等）中选择一种。游戏的设计不应该试图把整个角色都用所选择的形体来包住。正如前面在树的碰撞中所提到的，这样做同样会导致类似的问题。通常的游戏可以使用一个和角色的躯干类似的圆柱体，最低不要低于膝盖的上方，最高不要超过角色的肩膀，使用这种做法，角色不太会被那些简单的物体（譬如说楼梯）挡住，但是他们会被那些从逻辑上会阻止他们的物体挡住。图 3-13 中是本文建议为角色使用的碰撞体，下图把它和游戏世界中可能出现的其他物品的碰撞体作了对比。

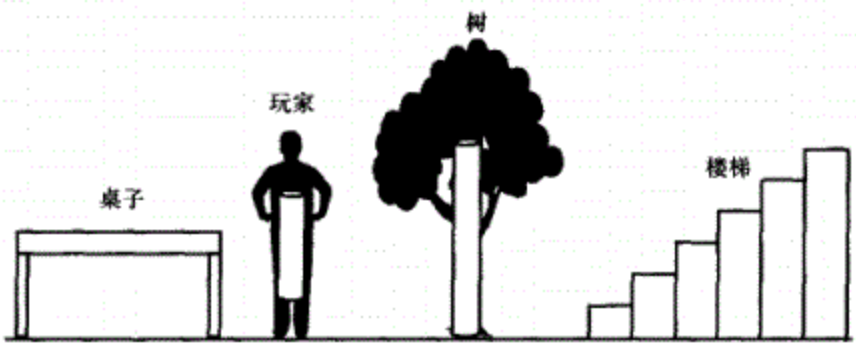


图 3-13 角色的碰撞体与其他物品的碰撞体之间的对比

碰撞系统不仅要能够满足游戏的要求并正常工作，还应该保持游戏的趣味性。为了让玩家能够持续地投入到游戏中，游戏的设计通常宁可引入一些视觉问题。在为游戏世界创建障碍物时，必须确保其碰撞体的长度和高度比角色在游戏中能够移动的最大合法距离要长，并且碰撞体的高度要高于角色碰撞体的最低点。

3. 角色与角色之间的碰撞

对于角色是否应该被设计人员在游戏世界中故意放置的静止障碍物阻挡住这一问题，实际上不会有什么争议。而对于角色是否应该被其他角色（无论是玩家角色还是 NPC（譬如说由 AI 控制的怪物））阻挡来说，则没有一个确定的答案。下面列出了一些实现角色之间碰撞的优点：

- 可以使游戏世界变得更为真实；玩家“拥有”他们占据的空间并确立了自己的地盘；
- 如果在服务端和客户端都实现了避免碰撞的代码，就不会因为玩家在游戏世界中占据同一块空间而导致视觉问题；

- 它的实现很简单：因为可以使用同样的方式来处理所有的碰撞表示；
- 它使游戏的开发可以实现更有趣的机制，譬如阻挡将会成为战略中一个不可分割的部分（譬如《天堂》中的围城战（castle siege））。

下面这些则是不实现角色之间碰撞的优点：

- 可以减少与碰撞有关的 CPU 负荷：因为根本不需要进行任何角色之间的碰撞计算；
- 向任意位置进行瞬间移动将会更容易实现，因为在这种情况下，只要目标位置对于角色来说是一个有效位置就可以了；
- 玩家不能使用碰撞来骚扰其他玩家（譬如说阻挡在某个商店的门口）；
- 不必为了处理那些同时到达同一位置的角色而编写相当复杂的代码（譬如说，当确定某个玩家最终可以占据这个位置时，必须采用一些方式来进行修正，这往往还会影响玩家对游戏的专注程度）；
- 即使在玩家角色们聚集成很大的群体时，与碰撞有关的 CPU 负荷也不会一下子大量增长；
- 不再需要为那些由 AI 控制的 NPC 编写复杂的代码来避免碰撞。

现在不少游戏在实现角色碰撞时采用了一种“推挤（shove）”模型。如果一个玩家角色的去路被另一个玩家角色挡住，他可以把那个玩家角色推开。然而，那个被推开的玩家角色肯定会感到不高兴，因此有必要事先在游戏背景中对这个模式的合理性加以评估。

4. 缩放

最后，游戏的设计对角色碰撞体进行处理时还有一个问题需要考虑。如果游戏要求支持改变角色几何建模的大小来为角色增加多样性和独特性，那么设计人员必须决定这一变化应该对角色的碰撞表示产生什么样的影响。如果仍然使用同样的碰撞表示，现存的碰撞代码就很可能可以正确工作于具有不同尺寸变化的角色上。另一方面，设计人员也可能因此丧失了针对不同的尺寸变化实施独特的游戏机制的机会。

3.6.8 物品放置

为了让玩家能够更加投入地参与 MMP 游戏，通常游戏设计会允许他们在游戏世界中放置各种物品，这样他们就能够在游戏世界中留下自己的“标志”。譬如说，有些游戏中玩家可以在游戏世界中获得自己的房屋，如果他们能够使用各种物品来装饰房屋的话，那会是非常诱人的。然而，允许玩家在游戏世界中任意放置物品通常会导致大量的问题。不仅服务端会因为需要对大量的物品进行记录而增加很多开销，还会为那些恶意玩家带来很多机会。考虑下面这些可能。

- 如果一个游戏支持玩家之间的对战，玩家就可以在某个很小的区域内放置大量的物品；一旦其他玩家进入这个区域就会暂时失去控制，因为描写这些物品的数据包会占据大量的带宽。
- 可以通过富有创造性的方式把物品堆叠起来以形成一些通路，这样就可以进入那些通常无法进入的区域。
- 玩家可以把某个物品放置在游戏世界中并且站到这个物品上面；随后他可以把这个

物品抽出来并且把它放到游戏世界中的另一个地方，这可能会让玩家能够移动到一个通常不能进入的区域。

- 同意交易的玩家会通过把物品放在地上来进行交易，但是不诚实的玩家可以在交易完成前把这个物品拿走并且不归还这个物品。

这样的场景举不胜举。游戏的设计应该尽可能地避免让玩家可以在游戏世界中的公共区域内任意地放置物品。下面是一些可行的方案。

- 在一定时间内，把放置的物品标记为属于它原先的主人。这段时间过后，其他玩家也可以拿走它。
- 实现一个非常积极的衰变机制：如果某个物品在一段时间内没有人认领，它就会被自动销毁。
- 把放置的物品抽象为某种通用的容器，看上去就好像一个袋子一样；并且它不能被堆叠，也不能放置在其他容器中。
- 在游戏世界中指定一些预先定义的、可以接受的位置来让玩家放置物品。譬如说，可以在游戏世界中设置一个公共的基座（pedestal），掌管它的玩家团体可以在上面放置一个物品来进行装饰。
- 不允许在公共区域放置任何物品；与此相反，只允许把物品放置在安全的区域，譬如说玩家的房屋中。
- 实现一些会四处走动并且会把放置在地上的物品捡起来的NPC。
- 物品只能在预先定义的放置点上进行有限的堆叠，譬如说可以把画挂在墙上，可以把银器放在桌子上。
- 在每次向客户端发送更新数据时，为与物品相关的信息设置一个上限，赋予那些会影响游戏的对象更高的优先级，譬如说有威胁的怪物。

3.6.9 侵入检测（Hack Detection）

几乎可以肯定，虽然游戏管理人员竭尽全力地预防，那些在客户端和服务端之间传输的数据包的详细内容最终还是会被破解的。这并不是说不应该对这些内容进行加密或者以其他方式来使其更难理解[Randall02]。而是应该把它当作一道能够延缓破解的额外防线，这样它就可以帮我们赢得很长的时间。游戏开发人员可以用这段时间来对游戏系统进行持续的加固以检测和防止那些恶意玩家的侵入。

毫无疑问，开发人员必须对每一个游戏系统的设计和实现进行评估来看看那些恶意玩家会怎样侵入这些系统。在对不同的设计进行评估时，应该优先选择那些很难或是不可能被侵入的游戏机制。譬如说，如果系统架构要求游戏世界仿真必须分散到多个服务器中，千万不要允许在服务器间的过渡区域上进行任何能够使游戏状态产生重大改变的行为。这意味着如果玩家可以把物品放置在地面上，系统就必须对游戏世界加以处理以把对任何物品的放置行为约束在同一台服务器中。

对服务端发生的所有玩家行为进行记录也是非常有价值的。必须注意不要把自己淹没在海量的数据中，也不能影响游戏的性能。对于游戏系统侵入检测机制来说，构建于商业性数据库管理系统之上的异步日志机制将会是一个强有力的工具。游戏开发人员可以对这些数据

进行挖掘（mining），这不仅有助于理解游戏的趋势，还可以检测到那些试图违反游戏规则的行为，譬如说复制有价值的游戏物品；闯入通常无法进入的区域等。

如果可以用更多的侵入检测代码来对那些有恶意嫌疑的玩家进行更细致的评估，那将会极为有用。譬如说，如果某个玩家不停地试图达到允许的移动距离上限，他就很有嫌疑，就应该对他进行更多的侵入检测。缩短对这个角色的移动进行检验的采样距离就可以进行更精确的采样。游戏管理人员应该对这个玩家记录更多的数据（相对于普通玩家来说），把本来可能忽略掉的细节也捕捉下来，譬如说这个角色进行物品交易时所处的精确位置。当这个可疑玩家登录时，支持人员会收到一个通知消息从而对他的行动进行实时监控。支持人员还可以查询这个玩家的所有聊天消息，寻找是否有证据证明他在进行非法的行为。可以采取的行动几乎是无穷无尽的，并且都不会被这个玩家察觉到。不管怎样，恶意玩家所带来的潜在影响值得对他们进行如此细致的审查。

3.6.10 总结

本文介绍了在设计和实现 MMP 游戏中的移动和物理系统时需要考虑的各种基本问题。这些系统的发布是 MMP 游戏开发中一个非常具有挑战性的部分。可行的方案有很多，然而每一个决策都会对游戏产生贯穿于整个生命期的深远影响。

如果把这些系统做得过于简单，就可能会失去潜在玩家，而且在与竞争对手进行对比时也会落入下风。同时，在这些领域过于冒进也会影响开发人员发布游戏的能力。更坏的是，如果不顾这些错误决定而强行发布游戏，之后的经历会非常痛苦，因为游戏里加入了一大堆无法预料的问题，而这些问题会随着游戏的发展突然出现。

知己知彼才能百战百胜。本文帮助游戏开发人员了解了在 MMP 游戏开发过程中会遇到的敌人之一。

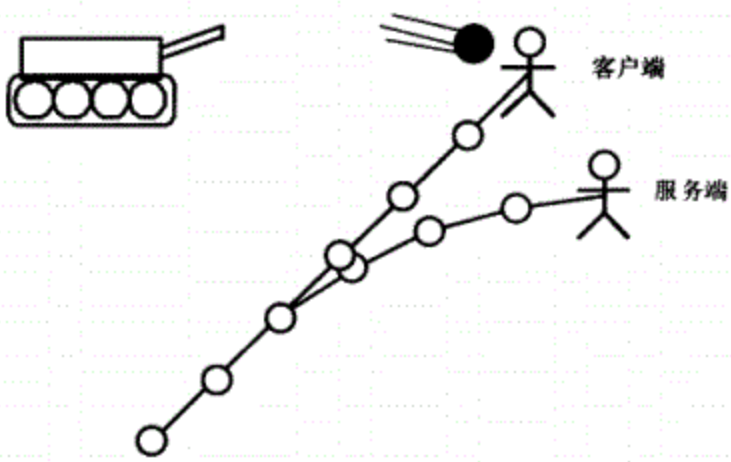
3.6.11 参考文献

[LIN] Lineage Product Web site, *http://www.lineage.com*.

[Randall02] Randall, Justin, "Scaling Multiplayer Servers," *Game Programming Gems 3*, Charles River Media, 2002

第 4 章

客户端开发



4.1 客户端移动预测

Mark Brockington, BioWare Corp.
markb@bioware.com

MMP 游戏的玩家希望可以从我们为游戏设计的数百万个对象中获得乐趣，在这些对象中，大部分都会自发地移动。玩家对游戏世界中生物移动的感官体验会影响到他们对游戏的入迷程度。即使是在偶然情况下，一个生物直接跳到新的位置也会造成玩家视觉上的不和谐。如果玩家看到一个怪物穿越关闭着的门或箱子，他们就会觉得自己只是在玩一个游戏，而不是真的在和那些穿着紫色夹克、戴着棒球帽的暴徒进行一场生死攸关的战斗。

这个问题并不是 MMP 游戏所独有的，在所有联网的多人游戏中，这个问题都很常见。在这篇文章将讨论一个算法以防止最终用户遇到这样的问题。

4.1.1 游戏的开发需要良好的移动预测

两个原因让游戏开发人员必须进行客户端移动预测：用户对获得即时反馈的需求以及网络数据包延迟的不确定性。人们希望看到事件可以在他们的屏幕上立即发生。当他们在地上点击、加大油门或是按下向前移动的按键时，他们希望看到角色的移动。玩家不能忍受他们的输入和所获得的结果之间存在较大的延迟，如果必须等待服务端来计算结果然后重新广播给客户端，他们很快就会感到厌倦。

很多方法都可以用来补偿用户对于即时反馈的需求。游戏系统可以在玩家发出指令后立即播放一小段声音提示来告诉玩家，他所控制的角色已经收到了命令，从而掩盖把这个命令传递给服务端所带来的时间延迟[Svarovsky02]。还可以在游戏中的单人部分引入命令延迟，来降低玩家在多人游戏中对立即响应的期望。然而，精明的玩家仍然可能注意到延迟。

即使服务端可以使用适当的方法来满足用户对于即时反馈的期望，还是会有不少超出它控制范围的力量与它作对。大多数多人游戏试图以一个恒定的频率来为每个相连接的用户进行更新。关于多人游戏的文献通常认为，这个更新频率应该是每秒钟 4 到 20 次。然而，对于游戏通信来说，因特网是一个不稳定的网络。有些信息会丢失，有些数据包会通过非最佳的路径路由。这就导致对处于不同位置的客户端进行更新所需要的时间会有很大的差异。在比较差的网络环境下，用户会看到角色在不同位置间跳跃。

这通常被称为“错位”（warping）。

鸵鸟策略并不适用于这个领域。也就是说，不能把脑袋埋在沙子里并且寄希望于什么都没有发生。像《半条命》（*Half-life*）、《战地 1942》（*Battle Field 1942*）以及《虚幻竞技场 2003》（*Unreal Tournament 2003*）这样的动作类游戏需要对所有用户的位置进行精确表示，以提供一个令人满意的有趣界面。即使像《无冬城之夜》（*Neverwinter Night*）、《横扫千军》（*Total Annihilation*）和《可汗：不朽的君主》（*Kohan:Immortal Sovereigns*）这样的游戏也需要不断的更新，以避免生物和坦克突然跳动到新的位置，这样使玩家难以在激烈的战斗情景中选中它们。

这篇文章将对三种独立的技术进行讨论，它们中的任意一种都可以帮助我们解决这个问题。

- 命令时间同步；
- 插值和推导；
- 可逆仿真。

4.1.2　命令时间同步

类似于《网络风暴》（*Netstorm*）这样的即时战略游戏（Real-Time Strategy，RTS）必须基于寻路请求（pathfinding request）来把对象移动到特定的位置。然而，如果每个客户都用相同的速度来移动他们的对象，那么，每个客户端的对象到达最终位置的时间将由服务端发出的移动命令的延迟决定。一般来说，离服务端较远的电脑更容易收到过时的信息。

因此，客户端与其仅发送“移动单元 X 使它到达位置 P”，不如同时发送预计的完成时间：“移动单元 X 使它在未来的某个时间 T 到达位置 P”。如果客户端和服务端之间的时间是同步的[Greer02]，服务端就可以调节移动单元 X 的速度以使它在正确的时间到达终点。

这个办法可以解决本文开始时所提出的那两个问题。例如：一辆坦克以较低的速度向着终点方向开始移动，并且，它在服务端也会做出同样的移动。如果某个客户端收到命令的时间比其他客户端要晚，那它可以以一个较快的速度移动坦克，使其可以在正确的时间达到终点。

图 4-1 到图 4-3 所示的例子中，第一个客户控制着坦克并试图将其移动 10 米（图下方标尺中的每一格刻度表示一米）。一辆坦克通常可以以 10m/s 的速度移动。在图 4-1 中所描绘的是坦克刚刚收到移动命令时的情景，客户正决定以一个稳定的速度来移动这个坦克。

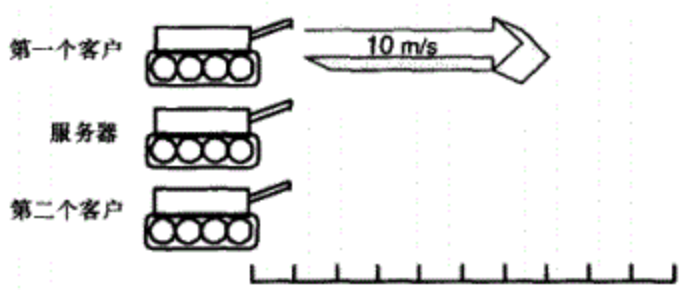


图 4-1　客户 1 在 0ms 时向坦克发出移动命令

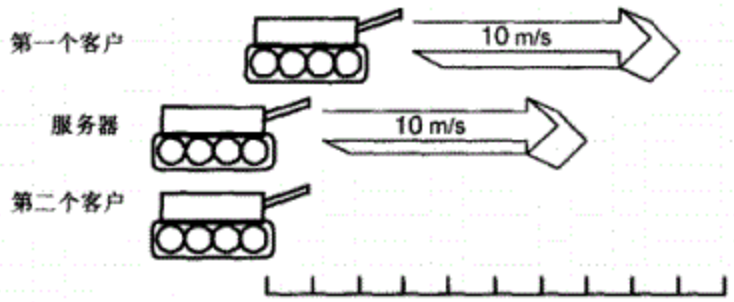


图 4-2 200ms 后，服务端接收到一个“坦克移动”命令并且开始移动坦克

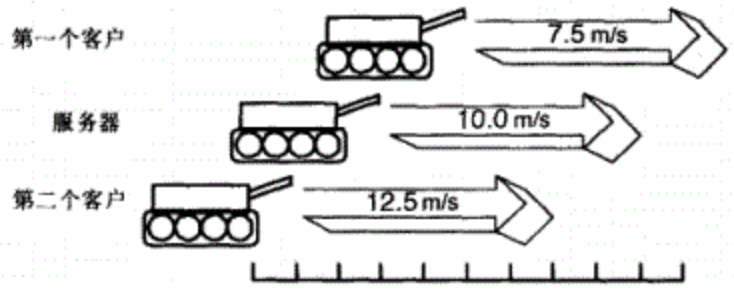


图 4-3 又过了 200ms，每个客户端都收到“坦克必须在 1200ms 时移动 10m”的命令

200ms 后，客户端的坦克向目标靠近了 2m，这时服务端收到了移动命令。服务端通过验证得知，这个移动命令是正确的并且开始让这个坦克以 10m/s 的速度向最终目标移动。因为坦克以 10m/s 的速度移动，服务端通知所有的客户端这个坦克必须在 1200ms 时移动 10m。

又过了 200ms，每个客户端都收到了“坦克必须在 1200ms 时移动 10m”的命令。因为第一个客户端的坦克已经向目标移动了 4m，它必须在 800ms 中移动剩下的 6m，因此它将把速度降低到每秒 7.5m。第二个客户端并不知道第一个客户端的移动命令，它必须在 800ms 内把坦克移动 10m。为了到达最终目标，第二个客户端必须以平均 12.5m/s 的速度移动这辆坦克。

在调整速度时，请注意使用平均速度。为了平缓移动速度的骤然变化（例如，从 10m/s 到 7m/s），可以在对象到达最终位置前等量减慢它的最终速度。进一步说，如果第一个客户端精确地知道它与服务端之间的延迟，它就可以预测到从发出移动请求到从服务端获得响应之间会有 400ms 的延迟，因此它可以以大约 8.3m/s 的速度在 1200ms 中把坦克移动 10m。

4.1.3 合并路点

有些游戏通过使用路点（译者注：路点（waypoint）是物体移动路径上的一系列关键点）来控制移动的方向，譬如《无冬城之夜》（*NWN*）。为了便于寻路，*NWN* 被做成一个二维游戏。怪物（可以通过寻路来移动的对象）通过一系列通往终点的二维路点移动。如果一个怪物没有路点，它就不能移动。怪物总是会向着它的第一个路点移动，并且在到达后删除这个

路点。使用这种基于路点的移动方法，只需要在移动开始的时候发送一个初始的路点列表，在这个怪物停止移动前就不用再去更新它了。

在 *NWN* 中，所有的移动请求都必须得到服务端的认可，但是如果客户端发现它到终点的直线上没有障碍物，客户端也可以自行移动。当一个对象撞上另一个对象的时候，服务端会发出停止移动的命令。这些特殊的情况使得服务端的对象位置和当前路径与客户端的对象位置和当前路径会存在差别。服务端和客户端之间的这种矛盾必须要在客户端得到解决。

为了减少跳跃的出现，必须把服务端更新过的位置和路径与客户端的位置和路径合并起来。如果客户端路径上有 c 个路点，服务端路径上有 s 个路点，那就有（c+1）×（s+1）种方法，通过在服务端路点和客户端路点间单一的连接，把这两条路径组合在一起，这里服务端和客户端的初始位置也必须作为路点包含进来。

图 4-4 展示了一条具有 6 个路点的服务端路径和一条具有 3 个路点的客户端路径。8 条灰线表示了在 *NWN* 中，可以尝试的 8 种用来连接当前客户端和服务端路径的可能。如果在这 8 条路径中的任意一条上都没有静止不动的对象，就可以把这些路点合并起来建立一个新的路点序列。这使客户端的对象可以在最短时间内回到服务端提供的路点上。路点列表被合并后，当前服务端路点列表的长度将会被计算出来，并且与客户端路点列表的长度进行比较。*NWN* 中怪物的移动速度会通过命令时间同步的方式得到调整。在 8 条线都不能产生一条无障碍路径的情况下，可以使用插值（将会在下一节中描述）来把对象移动到服务端的路径上去。

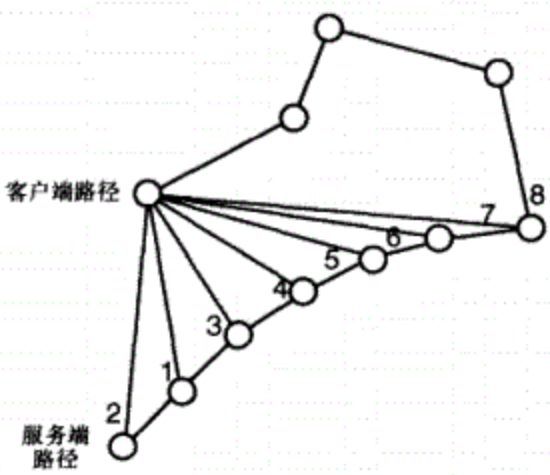


图 4-4　《无冬城之夜》中路点列表的合并

为什么不使用客户端和服务端路点之间的所有配对来进行上述计算呢？如果这两条路径中没有可行的方法来进行合并，上述的例子中，游戏支持人员将被迫检查 28 条路径上有没有固定的对象。在绝大多数情况下，如果这个算法不能在上面所显示的 8 条路径中找到路径，那么对余下的 20 条路径的搜寻，未必能得出一条路径，反而会大大地降低渲染循环的速度。

4.1.4 插值和推导

在前面所说的坦克的例子中，假设第一个客户端已经在沿着正确的方向进行移动。在某些情况下，这条路径可能与服务端所选取的路径不同。譬如说，这 200 毫秒的延迟可能导致起点到终点间的直线路径被永久地挡住了，从而使得服务端为坦克选择了另一条路径。在基于物理（physics-based）仿真的游戏中（例如大多数第一人称射击游戏），把客户端的怪物放置在正确的位置上是非常重要的，它使得瞄准不再是一项无用的训练。

《星球大战之帝国生死斗》（*X-Wing vs. Tie Fighter*）会让客户端的飞船立即跳到它们正确的位置上[Lincroft99]。这在延迟很小的时候不算是一个很坏的选择，但是如果延迟的时间过长就会导致“错位”。在游戏发行前，设计人员加入了一个“平滑”算法来对过去预测的玩家位置和他们当前的新位置进行插值。《部落》（*Tribes*）以 100 毫秒的周期实现了类似的插值技术[Frohnmayer00]。

下面的例子将说明插值和推导是怎么工作的。图 4-5 中，有两条线。黑色的直线代表了客户端已知的对象运行方式，而灰线代表了对象在服务端实际所做的转向。线上的每一个小圆圈代表一个很小的时间片（在这个例子中，每一个圈表示 100 毫秒）。如果在 400 毫秒中没有从服务端收到任何输入，客户端将通过推导来使这个对象沿着直线路径移动。推导使用最后得到的位置（或者位置序列）和对象的速度来预测对象在未来的位置和速度。服务端对客户端进行更新所需等待的时间越长，两者之间对象位置的差别就越大。

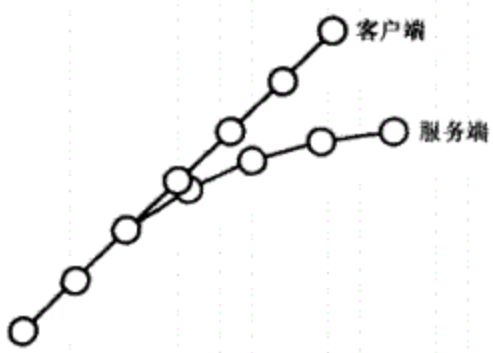


图 4-5 服务端的对象位置和客户端推导出的对象位置

图 4-6 展示了当客户端接收到这个对象在 200 毫秒后转向的输入时，将会怎样进行处理。客户端必须使用插值来与服务端的速度和位置进行匹配。客户端对象必须在一个时间段内达到与服务端相匹配的位置和速度，服务端才能使用这个处于未来某个时间上的位置和速度来进行插值。在这个例子中，客户端做了一个很大的转向以使客户端对象的位置和速度在 200 毫秒后与服务端的相同。

如果客户端位置和服务端位置之间没有可见的障碍，插值就可以很好地工作。如果这条路径不是畅通无阻的（譬如说一堵墙截断了客户端的插值路径），就可以采用一个备选方案：把客户端的对象直接放到正确的路径上以统一客户端和服务端的位置。

图 4-6　客户端使用插值来和服务端的位置相匹配

前文所描述的是一种线性插值技术[Olsen00]。这种技术只用了一个参考点来把客户端的对象移回它正确的轨迹。对大多数游戏来说，这足够了。然而，在类似于《半条命》之类的三维游戏中，死亡对战（deathmatch）玩家更喜欢像球一样在地面上跳动，这样他们就会变得更难击中[Bernier01]。在任何给定的时刻，玩家不是在空中就是在地上准备下一次跳起。如果使用采样方法，玩家可能看上去像是浮动在空中。在《半条命》中所使用的方法是通过最近更新的几个点来找到一条通过这些点的曲线，并由此预测玩家是怎样在游戏世界中运动的。通过插值，对象将被正确地拉回地面并且像皮球一样重新弹起，而不是在地面上方滑翔到下一个位置。

图 4-7 的例子展示了一条路径以及一个跳动的怪物，其中垂直方向是 z 轴正方向。这个怪物（用灰线表示）在到达地面的同时再次跳起。圆圈代表了客户端所知道的这个物体的最后 3 个位置，而三角形代表了服务端的当前位置以及未来 100 和 200 毫秒时的位置预测。这个例子会像图 4-5 和 4-6 中所示的线性插值例子那样，仍然用 200 毫秒使怪物到达正确的服务端位置。线性插值技术将强制怪物沿着图 4-7 中的黑线移动。

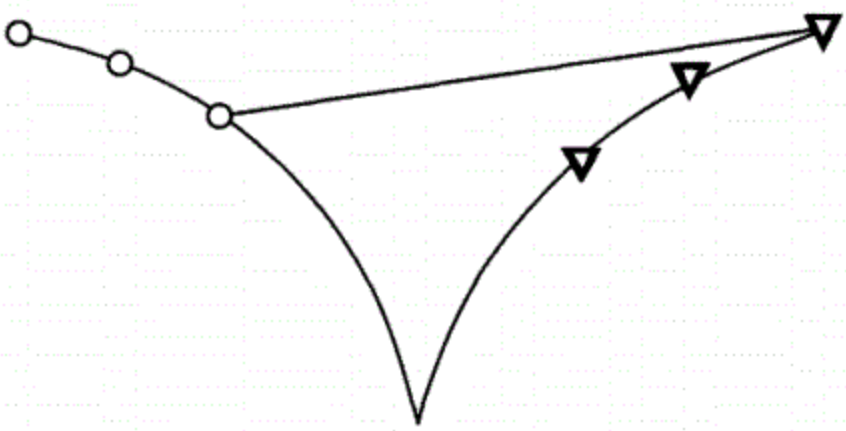


图 4-7　为一个跳动的怪物所做的线性插值

图 4-8 展示了一条在服务端新路径上的前两个预测点和客户端所知道的最后两个点之间作出的 Catmull-Rom 样条[Rabin00]。高次的样条可以使插值路径更接近于影响点。

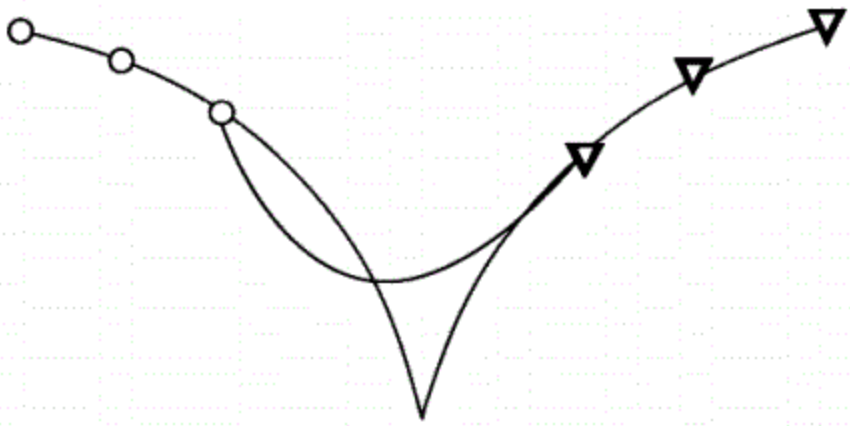


图 4-8 为一个跳动中的怪物所做的 Catmull-Rom 样条插值

4.1.5 为瞄准延迟使用反向仿真

仅仅使用上面的方法并不足以解决在第一人称射击游戏中所出现的问题。那些游戏不依赖于路点工作，所以无论是命令时间同步还是路点合并技术都是无效的。虽然插值好像很有用，但是在客户端延迟较大的情况下，当客户端的“射击”指令被发送到服务端执行时，玩家会发现他总是无法击中目标。玩家可能已经在他的视野中瞄准了这个对象并且在合适的时刻扣动了扳机，但实际上这个目标对象在服务端却处在另一个不同的位置上。最终这可能会使用户遭受巨大的挫折，他们会认为射击代码编写得实在是太“粗心”了。

图 4-9 展示了这个问题。玩家处于坦克的位置，在他的视野中有一个士兵，于是他扣动了扳机。然而，服务端知道这个士兵已经转向到另一条不同的路径上去了。当服务端收到这个射击命令时，士兵不会受到任何伤害。

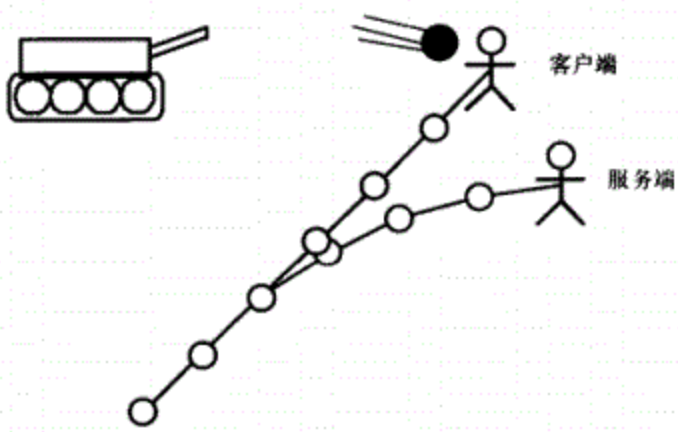


图 4-9 服务端和客户端所见的不同导致没有击中物体

那么，怎样才能解决这个问题呢？《半条命》中的另一个例子[Bernier01]可以帮助找出解决问题的办法。如果服务端可以知道客户端进行射击时的情景，它就能正确地判断这个人是否被击中了。最简单的方法是由客户端来告诉服务端它是否击中对象。但事实上，游戏开

发人员永远都不希望出现这种由客户端来告诉服务端一个人是否被击中的情况。因为这等于邀请黑客来摧毁游戏服务[Randall02]。

在《半条命》中，游戏仿真的核心不仅能够作正向的仿真，还可以作反向的仿真。服务端的游戏代码可以通过它发给客户端，从而得到确认的消息序列来沿着历史记录回溯，并且计算出发出命令时，客户端对象的位置。完成对击中/未击中的计算后，这个有疑问的对象会被放到它当前的时间和位置中去。这需要客户端和服务端用一个同步时钟来精确地计算延迟。

在图 4-9 的例子中，客户端在 400 毫秒前收到最后一个移动数据包，但还没有确认任何后续的命令。因此，服务端知道它必须回到客户端确认过的最后一个命令的时间上去。图 4-10 中的带箭头曲线就说明了这点。然后，服务端必须推断如果客户端没有收到任何后续指令将会发生什么（图 4-10 中的倾斜箭头）。现在这个士兵所处的位置和客户端所看到的一样，因此服务端可以证实坦克的确击中了士兵。

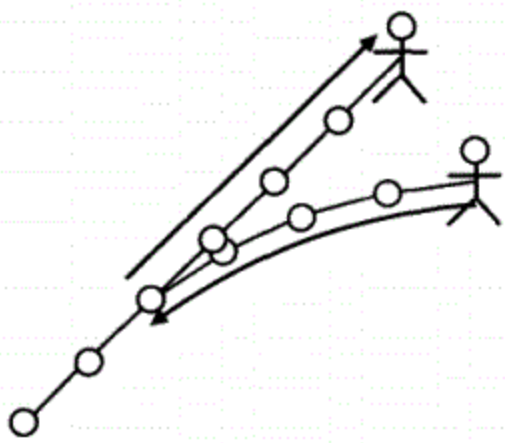


图 4-10　使用反向仿真可以使服务端从客户端的角度进行判断

4.1.6　总结

对于不同类型的游戏来说，客户端移动预测可能是“一个可以拥有的好特性”，也可能“对于游戏的成功至关重要”。本文希望以上所讨论到的技术以及附加的参考文献可以帮助读者理解，在完成一个 MMP 游戏时，开发人员需要做些什么以及在这个方面可以达到怎样的程度。

4.1.7　参考文献

[Bernier01] Bernier, Yahn, “Latency Compensation Methods in Client/Server In-game Protocol Design and Optimization,” Game Developers Conference 2001 Proceedings, 2001: pp. 73–85.

[Bettner01] Bettner, Paul and Mark Terrano, “1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond,” Game Developers Conference 2001 Proceedings, 2001: pp. 789–803.

[Frohnmayer00] Frohnmayer, Mark and Tim Gift, “The TRIBES Networking Model or How

to Make The Internet Rock for Multi-player Games," Game Developers Conference 2000 Proceedings, 2000: pp. 191–207.

[Greer02] Greer, Jim and Zachary Simpson, "Minimizing Latency in Real-Time Strategy Games," *Game Programming Gems 3*, Charles River Media, 2002.

[Lincroft99] Lincroft, Peter, "The Internet Sucks: What I Learned Coding X-Wing vs. TIE Fighter," Game Developers Conference 1999 Proceedings, 1999: pp. 621–629.

[Olsen00] Olsen, John, "Interpolation Methods," *Game Programming Gems*, Charles River Media, 2000.

[Rabin00] Rabin, Steve, "A* Aesthetic Optimizations," *Game Programming Gems*, Charles River Media, 2000.

[Randall02] Randall, Justin, "Scaling Multiplayer Servers," *Game Programming Gems 3*, Charles River Media, 2002.

[Svarovsky02] Svarovsky, Jan, "Real-Time Strategy Network Protocol," *Game Programming Gems 3*, Charles River Media, 2002.

4.2　保持流畅：异步客户和时空穿梭

Jay Patterson, Xbox Live
jaypat@xbox.com

当人们在《星际迷航》（*Star Trek*）的预告片中看到企业号被一场巨大的爆炸摧毁时，大家都会猜测，这是否又是一个关于时空穿梭（time travel）和异度空间（alternate reality）的故事。真实生活中的舰艇是不会这样爆炸的，所以这必然发生在另一个宇宙中。不仅如此，还可以确定的是，这个舰艇是由于时空错乱（temporal anomaly）而毁灭的。星际迷航的作者知道时空穿梭和异度空间之间的关系符合人们的常识。

本文的目的就是通过使用时空穿梭和异度空间之间的这一简单关系，来帮助游戏开发人员处理在发生延迟或是客户端不愿意等待时经常会出现的那些复杂问题。在发生延迟时，这一关系还可以帮助服务端对玩家的游戏体验进行处理。“异步客户（asynchronous client）”本质上就是一个拒绝等待数据的客户端。虽然实际的网络模式和协议可能不同，但是只要某个客户端在网络出现问题时仍然可以继续运行，本文就认为它是异步的。在这种类型的游戏中，每个玩家在某种程度上都处于他自己的世界中，因此，开发人员不能再想当然地认为他们的游戏仅仅创建了单一的虚拟现实。

接下来要提出一个基本假设：必须从用户的角度去考虑问题才能做出一款好游戏。也就是说，游戏开发人员应该要把处理问题的角度定位在游戏真正发生的地方（也就是玩家的电脑屏幕前），而不是将自己置身于服务端中。传统的做法强调只有一个游戏世界，提倡通过巧妙地共享游戏世界的状态来实现多人功能。共享的游戏体验需要所有客户端能够对所感知的游戏世界达成一致，大多数情况下，这种做法都是有效的。本文所表述的方法并不是要取代这个观点，而是要通过对各个独立的私有世界的可能性和限制作更深入的描述来扩展这个观点。要真正地站在玩家的角度设身处地考虑这个问题可能很困难，甚至会很别扭，但如果把这一切想象成时空穿梭就会简单得多。

如果游戏设计人员想为所有的玩家创建一个流畅、一致的游戏视图（game view），延迟和带宽（数据传输的时间和一定时间内可以传输的数据总量）是两个最大的障碍。游戏必须不停地与这些限制作斗争，而它们具有一个共同的因素，那就是时间。要在这些斗争中获胜，开发人员必须以一种对游戏有利的方式去处理时间。随着游戏事件在网络上传输，本质上游戏也在随着时间向前推进。要想成为一个时空穿梭者就意味着要对这个流程有所了解。

开发团队的所有成员都必须能够接触到关于时间的流逝和使用方面的实践性或是理论性的建议。每个 MMP 游戏制作人员都知道应该怎样做才能使游戏获得最大的成功。同时，每个团队成员也都应该意识到他们所做的工作会对玩家的游戏体验造成什么样的影响。即使是美工，如果他们能够对网络的基本限制有所理解，游戏的质量也可以得到很大的提高。

通常，AI 和网络程序员负责驱动游戏状态的传输和使用，因此他们很清楚会在网络上传输些什么。然而，并不可能完美地保护其余的每一个人。如果要对互联网进行最优化的使用，必须针对游戏中特定玩家在特定时刻发生的事情制定特定的方法。设计人员必须确定哪些状态需要在不同玩家的游戏世界中进行同步，并且为它们设定优先级。他们在设计的时候必须非常小心以使程序员能够完成设计目标。此外，在整个开发过程中，工作人员必须不断地使用工具来评估游戏在真实的因特网环境下会发生什么。当游戏系统允许客户端可以异步地决定它们自己的状态时这一点尤为重要，因为游戏现实中所存在的固有差异会导致它以不可预测的方式运行。

4.2.1 共享状态的基本问题

共享状态是多人游戏中“多人”的由来。如果一个游戏中，没有在共同的环境中基于共同目标的交互，它就不是一个多人游戏了。这个交互可以简单到只有一张虚拟的桌子和一副牌，也可以是一个能让玩家身临其境的完整的仿真世界。在任何情况下，这个游戏里都会有一些对所有玩家都必须相同的共享数据。并且，由于进行游戏并不是一个被动的行为，所以有些数据会以不可预见的方式发生改变。怎样发送这些共享状态的改变是所有多人游戏都必须解决的基本问题。

长期以来人们对这个问题的探讨一直很活跃，大家使用了各种不同的方法来研究这个问题[Singhal99]。美国军方对创建分布式虚拟环境（*Distributed Virtual Environment, DVE*）进行了广泛的研究，其中的一些重要成果有分布式交互仿真（*Distributed Interactive Simulation, DIS*）、仿真网络（*Simulator Networking, SIMNET*）和高级架构（*High-Level Architecture, HLA*）等。这些都为解决共享状态问题贡献了一些新的思想，不过要把它们直接应用于因特网多人游戏中还非常困难。这种情况的出现有两方面的原因。一方面很多有关的文章都是以一个完善的高速局域网为前提的，因此完全忽略了延迟和带宽问题。另一方面，为了实现互用性（interoperability），军事仿真方面的研究都非常注重于标准化，而游戏开发人员并不需要这些。然而，通过在因特网上搜索任何关于上述研究的信息，就找到很多有趣的文章。

很多关于状态传播的讨论都强调吞吐量，也就是被移动数据的总量。[Singhal99]是一篇关于这方面的优秀文章，它从“吞吐量和一致性的折衷”这一角度探讨了这个问题。然而，很少有人从“延迟和一致性的折衷”角度来思考这个问题，而这正是本文的目的。造成这种局面的原因很可能是因为吞吐量方面的问题更容易通过技术手段解决。程序员可以在网络拓扑和消息设计等领域多做一些工作来减少带宽的限制。但这种解决方法对于处理延迟来说并不适用——也正是因为这个原因，开发人员才要让整个团队都参与解决这个问题。关于吞吐量和一致性方面的技术诀窍就留给读者自己去发现了。

4.2.2　航位推测法：时空探索者会做得更好

正如前面提到的，游戏开发人员可以从军事仿真研究中得到一个与因特网相关的实用方法：航位推测法（*dead reckoning*）。这是一种客户端预测的方式，它被广泛地用在那些需要发布位置数据的游戏中，譬如说第一人称射击游戏。大多数关于这一方法的讨论都集中在怎样用它来有效地减少所需发送的数据上，但是这里本文想讨论怎样用它来补偿网络延迟。关于航位推测法的完整讨论本身就是一个独立的话题。简单地说，它就是根据一个物体过去的行为来判断它“现在”应该在做些什么。譬如说，如果系统必须要移动某个物体，但是还没有接收到它的新位置，于是就要对此进行猜测。这种猜测往往基于这个物体过去的位置和移动的速度，（有时）也会考虑它在已知的最后位置上的加速度。在使用航位推测法时，开发人员可以保留大量的历史信息从而能够使用各种不同的预测方法。游戏开发人员还可以根据不同类型的实体对航位推测法作出特定的调整。

使用带有时间戳（time-stamp）的信息来进行预测，可以使物体的移动更为流畅。通常，一个预测算法需要使用速度和加速度信息，这意味着游戏的开发也必须对时间加以利用，但是它需要的可能只是从上次更新以来本地时间的变化。用一个共享的仿真时间来标记状态信息可以让游戏开发人员更好地对真实位置和预测位置之间的差异进行调整。图 4-11 中使用的是基本的航位推测法。

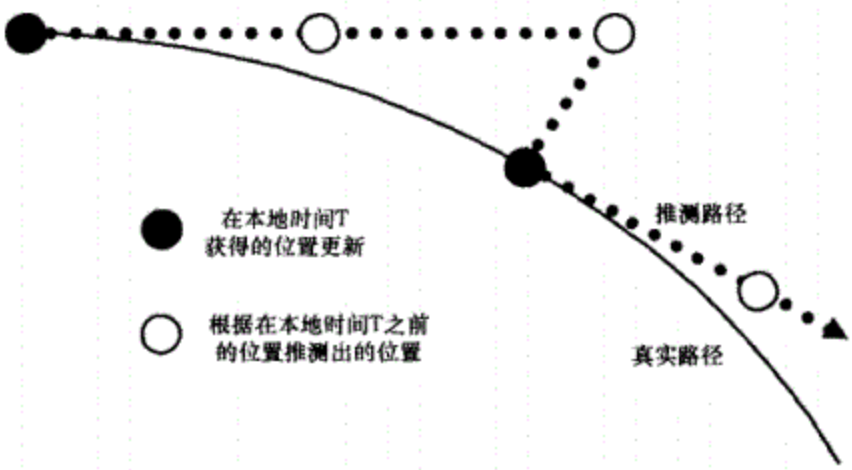


图 4-11　基本的航位推测

图 4-11 显示了，一个物体要移动到正确的位置会导致大量突发的不连续。但如果使用了仿真时间，就会得到不一样的结果。因为有延迟，服务端接收到的可能只是对过去某个时刻真实状态的更新。但它已经对那个时刻的位置进行了猜测，因此直接跳跃到实际位置效果并不好。于是服务端可以根据更新时间和当前仿真时间的区别再一次对新位置进行预测，如图 4-12 所示。

服务端还可以通过对原先预测的路径和重新预测的路径进行插值，来获得更为流畅的效果[Fujimoto00]。需要注意的是，插值可能会导致预测结果与真实状态之间出现无法接受的偏

差。正如图 4-12 中所示，即使不进行插值，显示路径也不会回到真实路径上去。并且，除非这个物体进行直线运动或是完全停止，否则它永远也不可能回到真实路径上去。如果保持在实际路径上非常重要，那就有必要让这个物体跳转到那里，游戏系统甚至可以通过让本地时间倒退来实现这点。

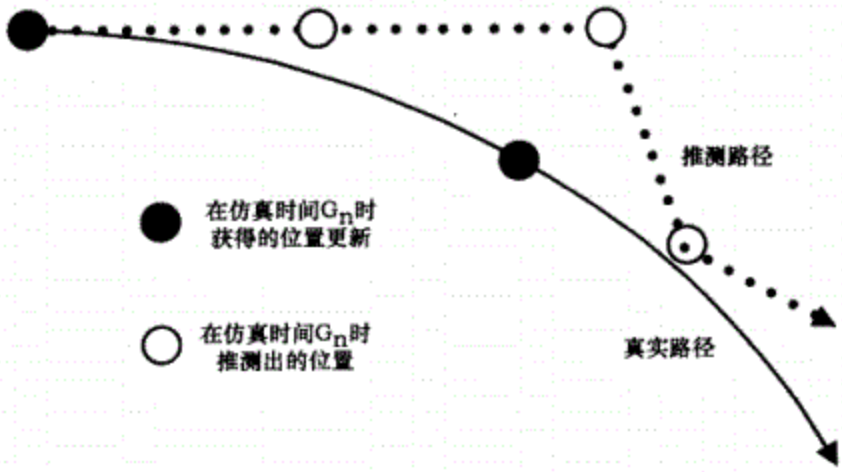


图 4-12 使用时间补偿来进行航位推测

在使用航位预测法时，要在流畅和精确之间找到一个最优的折衷，就必须从游戏模式的角度来进行考虑。游戏中的每一个物体不一定需要遵循同样的规则。譬如说，服务端往往很难对会突然改变方向的物体进行预测，因此对其进行流畅的显示往往意味着不是很精确。然而，如果这个物体并不需要在每个玩家视野中都具有相同的表现，服务端也可以这么做。要使航位推测法发挥最大的功效，设计人员和程序员必须共同合作为每一个物体作出最佳的选择。

4.2.3 仿真时间表示：为时空穿梭建立通道

当提及游戏中的时间（仿真时间）时，通常人们说的并不是真实时间（有时候人们也把真实时间称为挂钟时间（wall-clock time））。在游戏中会同时存在很多时间比例。正如地图上的比例一样，时间比例是玩家钟表上过去的时间和游戏世界中过去的虚拟时间之间的比例。把事件和行动按照所使用的时间比例分类有助于游戏开发人员对它们进行更为有效的处理，因为这打破了“时间是绝对的”这一概念。譬如说，手和脚的动画可以按照实际时间来操作，子弹也可以按照实际时间来飞行，但是游戏的设计或许应该让一些环境效果（譬如说白天、黑夜和资源再生）加快速度，游戏中经济和文化的发展也应该变得更快。并且，考虑这些时间比例可以使用在哪些情况下对游戏的设计也有概念上的帮助。譬如说，某些情况下一个时间比例可以持续不断地向前推进，而另一些情况下时间可以被提前。对后者来说，游戏服务端可以以一种离散的方式对事件进行批量处理。

无论采用什么样的比例，也无论客户端怎样根据这个比例进行移动，要达到目的，我们需要一个大家一致公认的共享仿真时间来让所有参与者能够同步。这方面已经有不少完善的协议了。它们之间的主要区别在于：有些使用一个中央控制，而另一些则使用分布式的控制；有些周期性地进行广播，而另一些则对请求作出反应[Fujimoto00]。这里最基本的问题是服务

端需要对分发时间值所花费的时间进行补偿。一个简单可行的方法是使用一个集中的时间源，向它发出一大堆访问，然后把平均回复时间的一半加在接收到的值上。也可以使用更复杂的方法，但是通常没有必要。

虽然因特网的延迟在传输的两个方向上并不一定相同，但是这个技术仍然适用。只要所有的事件使用同一个地点产生的时间戳来进行排序，就不会在仿真逻辑中由于事件之间互相依赖而导致错误。

一旦所有的时间都同步了，工作人员就可以在一个全局的环境中对事件和消息进行定位，这可以带来很多好处。首先它可以改进前面介绍的航位推测法的效果，并且能够迅速地确定当前的延迟。全局时间环境使得游戏开发人员可以为事件维护一个适当的次序。有一个早期军事仿真中的例子可以说明这一点的重要性。如果事件没有时间戳，服务端就只能按照事件接收的次序来进行处理。这意味着当消息的延迟不同时，服务人员很可能会看到一把枪在目标被击中后才开火。需要注意的是，使用一个全局的时间来对事件进行定位和排序并不意味着所有的客户端必须使用相同的实际时间。与此相反，参与者甚至可以在仿真时间中按照他们自己的意图任意地移动来解决由于延迟而导致的问题。这可以是加速、减速、故意滞后，甚至可以是根据接收到的新信息回溯到过去的某个特定时间点来对事件进行重新计算。对于按序执行事件来说，全局仿真时间只是一个可以参考利用的依据，而不是一个强制使用的限制。

仿真时间必须完全独立于帧频（frame rate）。我们对客户端时间的操作应该独立于画面更新的节奏。这并不意味着不能按照帧之间的固定间隔来接受用户输入，也不意味着不能在每一帧都对仿真进行更新。事实上，设置仿真时间的目的是要让物体移动得更为流畅，因此游戏设计人员希望每一帧都有一些新的变化。这里要说的是，每秒 50 帧并不意味着强制每帧都必须经过 20 毫秒的仿真时间。游戏设计人员希望可以自由地决定每一帧所经过的仿真时间。因此，像动画之类的对象特征必须独立于仿真时间，这样在仿真时间变得不规则时，玩家也不会有所觉察。并不是任何形式的延迟补偿都需要支持不规则的时间片。严格地说，本文前面所使用的航位推测法例子就不依赖于这样的支持，通常的做法（简单地让每个参与者在略有滞后的时间里运行）也不需要这一支持。然而，让仿真时间能够独立于帧频的确可以让游戏设计人员选择更多的方法。此外，为了获得最大的灵活性，对象行为的计算必须基于时间的改变（time delta）而不是基于固定的离散仿真更新（tick）。

4.2.4　直接操纵时间

游戏中，处理时间的一种常见方法是让每个客户端的仿真时间略微落后于实际时间。这样在服务端需要使用某个网络消息时，可以向它提供一个到达客户端的时间，当然这种方法会给玩家的交互带来一个固定的延迟。但即使在共享的游戏世界，在能够对玩家行动进行处理之前存在这样一个固定的延迟，要想掩盖这一延迟，可以使用一些声音和动画效果来立即对玩家的请求进行确认，并且马上开始玩家的行动。要想知道这个方法有多方便，可以看看前面提及的，进行流畅预测的例子的一个退化情况。如果客户端仿真时钟指向的是过去的某一个时间，服务端通常可以在一个远程对象真正地到达某个位置之前预先知道那个位置。这意味着服务端是要与这个对象未来的某个位置进行插值，而不是从某个已知的过去位置进行

推导。这样一来，就不会产生偏差，因为服务端可以有规律地获得真实的路径。这个方法的另一个改进是使用可变的延迟。在网络延迟增加或者数据包丢失的时候，服务端可以放慢客户端的仿真时间从而增长消息窗口；相反，当情况有所改善后，服务端可以加速客户端的仿真时间从而减少延迟。这可以让游戏的运行在大多数时间里都使用一个最小化的延迟，并且在因特网出现阶段性的拥堵时也可以正确处理。

类似于让客户端运行在过去的某个时间，游戏管理人员还可以使用一个中央服务器来观察过去发生的事情并做出决定。《半条命》（*Half-Life*）使用了这种方法来进行延迟补偿，这是一个非常好的应用实例[Brenier01]。本质上，服务端返回到过去的某个时间点，并且把玩家的行为放在当时他们所能看到的环境中。管理人员需要为命令加上时间戳以使这个方法可行。服务端可以知道当时的周围环境，因为它与客户端使用同样的代码，并且知道客户端从它这里接收到了什么。这使得服务端可以重新创建玩家在射击时所看见的场景，从而来决定他是否击中目标。

无论怎样进行预测或是对时间进行特殊的操纵，游戏中总会存在某些情况使得客户端缺乏足够的信息来以某种有意义的方式向前推进游戏的进行。一旦发生这种情况，游戏就会停止。玩家对游戏的沉醉感会立即被打破，而且很可能会错过一些游戏共享体验中发生的有趣事情，譬如说，他被杀害了。设计人员在选择游戏模式时，必须进行风险或回报分析。游戏设计人员必须知道一个游戏发生停滞或是跳跃的可能性有多大，哪些方法可以解决这些问题，而哪些方法则不起作用。譬如说，如果在一个移动缓慢的游戏中一种做法的风险不大，尽管这所带来的好处也不大；而在需要不停跳跃的游戏中这么做就带有很大的风险，因为这里面有很多难以预测的边缘情况，更不用说这种游戏需要实时的控制。如果使用很多延迟来补偿网络延迟，就会使场景变得很不真实。游戏设计人员必须对此有所了解，并且最终必须能够提出一个可行的游戏模式。

要使设计人员的工作更加容易，他们可以对问题进行分解。这样他们就可以为不同的对象设定不同的时间容忍度（temporal tolerance）并且在运行时动态改变。只有根据问题域（游戏）来做决定，我们才能作出对风险管理和优化的最佳抉择。

下面的技术正是要实现这一点。这项技术出自于军用 HLA 仿真标准[DMSO02]的运行时底层架构（Runtime Infrastructure, RTI）中的时间管理系统。在某种程度上，这种方法类似于前面介绍的让客户端在一个特定的延迟上运行的方法，只是它的应用面更广。

RTI 时间管理主要着重于处理那些由时间控制的物体和那些可以控制时间的物体。它还规定了每个参与者必须通过请求来让本地时间向前推进，这样它就可以控制时间的消逝。在这里本文将着重讨论控制时间以及被时间控制究竟意味着什么。要想做到这点，必须理解下面这两个重要概念。

- 最小更新时间（Lookahead）：能够控制时间的参与者产生事件的最小粒度。换句话说，最小更新时间就是一个对象的状态发生改变到它的状态可以再次发生改变所要经过的最短时间。
- 时间戳下限（Lower-Bound Time Stamp, LBTS）：一个受到时间限制的参与者从可以调整时间的其他参与者那里获得状态改变的最早的时间点。也就是说，这是一个参与者可能获得下一个更新的最早时间。

图 4-13 说明了这些概念的意思。图中有 4 个实体。第一个是控制实体，第二个是受控实

体，第 3 个在受控的同时也可以进行控制，第 4 个则什么都不是。它们中的每一个都沿着一个全局的仿真时间线向前移动，它们的当前时间由一个有阴影的圆点表示。每一个调整实体前面都有一个矩形阴影，表示它们的最小更新时间。一根垂直的虚线代表 LBTS。

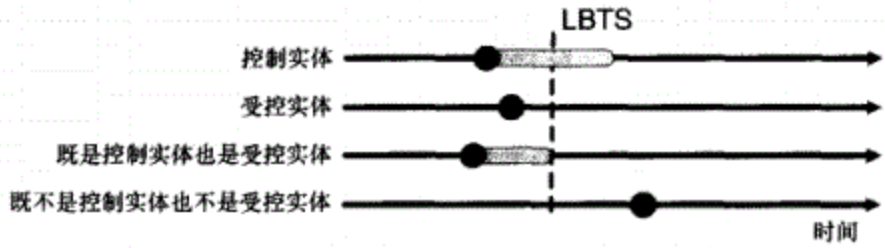


图 4-13　使用 RTI 时间管理的实体状态图

首先假设游戏支持人员可以方便地为处于不同状态下的不同对象指定最小更新时间并进行试验。然后把 LBTS 想象成一个壁垒，如果不想让本地仿真错过可能的状态更新的话，就不能让本地仿真时间跨越这个壁垒。读者可以由此总结出因为最小更新时间在大多数情况下都很小，所以 LBTS 取值的范围不大。当然，这只是众多网络工具中的一个，而它们可以组合在一起使用。譬如说，通过适当的预测，系统可以对最小更新时间进行拉升来把它用在那些本来没什么用处的场合。游戏系统可以使用 LBTS 来决定一个客户端仿真的时间延迟，也可以把它当作一个目标以对客户端时间延迟像橡皮筋似地进行拉长或缩短。无论怎样使用这个方法，其关键的好处都在于，它让时间成为离散的从而可以被更方便地使用。

4.2.5　总结

要在游戏中完全支持不同的客户视角是一项极具挑战性的工作。无法想象如果客户端可以自由地向未来流畅地推进时，会发生些什么。这不仅是一个压在每个团队成员肩上的重担，而且非常难以管理。要对此进行管理，必须高度重视以下这些目标。

- 要让游戏在客户端的运行保持流畅，这比精确更重要。游戏中可能会有很多玩家，但要让每个玩家都觉得他才是游戏中惟一的玩家。必须不惜一切代价来避免停滞和跳跃。
- 要让玩家能够知道游戏中发生的事情，并且能够参与其中。让玩家可以掌控自己的游戏世界要比让他们为其他玩家的游戏世界做贡献更为重要。

或许有人会想当然地认为第二点有可能略微超出了本文的讨论范围，就好像本文在这里对其他关于安全和竞争有限资源的问题进行讨论一样。然而要注意的是，这并不是说相对于使用服务端来决定全局状态，游戏开发人员更倾向于让客户端来决定。本文的目的就是要倡导这种观点：无论底层是怎样实现的，都必须让玩家能够对游戏有一个更好的感知。虽然说如果信任某个客户端，并且让它来决定游戏世界的状态，那么系统非常容易让它看上去就能够做到。然而，必须注意，互联网游戏的历史说明了让客户端来做决定并不是一个好主意。不过安全和信赖只是程度上的问题，并不是绝对的。还是可以针对特定的游戏情况和一般的玩家/平台进行有选择的优化。

所有这些都与用户的感知有关。如果打算让客户端可以异步执行，就需要让它们能够以一种合理的方式进行异步处理，无论这些客户端是否真的能够决定游戏状态。要确定应该让客户端能够对哪些情况作出决定，牵涉到不少游戏设计人员在支持异步客户时所需作出的一些最基本的平衡和决策。游戏的设计允许玩家其自身所做的事情，与它允许其他玩家对他们做的事情有着潜在的关联。比方说本文前面讨论过的用于《半条命》中的延迟补偿方法。为了把射击其他玩家的过程做得更加一致，半条命的设计人员不得不接受的事实是：受到射击的过程可能会不一致。像《半条命》这样会很快再生的游戏，蕴含的真正乐趣在于对他人进行攻击，因此对死亡的惩罚非常小，所以这样做是一个明智的决定；而在一个玩家死亡所付出的代价很大的 MMP 游戏中，这就不一定了。

构思和创建一个可以使时间按照自身需求进行改变的游戏，是一个不断探索和发现的过程。只有通过测试才能知道游戏的设计、代码以及美术资源在恶劣的互联网延迟情况下表现如何。游戏测试人员使用工具来仿真因特网的环境。要确保从项目一开始就让尽可能多的人定时使用这些工具进行测试，最好就是让每个运行游戏进行测试的人都能够同时使用两个客户端，并且在这两个客户端之间有一个互联网模拟器。能够亲眼看到游戏并从中看到他们决策结果的人越多，游戏就会做得越好。让游戏引擎能够（以当前延迟和带宽的方式）显示当前网络的状况也是非常重要的，因为设计人员可以使用这些信息来作出有价值的决策。可以掩盖所有网络问题的完美抽象是并不存在的，但是有不少好方法可以把这些问题包装起来以便于处理。

通过事件展开（event unfold）来让时间向前推进这种方法，对于所有网络多人游戏的开发人员来说都很有帮助。即使进入了宽带时代，游戏的开发仍然无法摆脱延迟和带宽的限制。要想成为一个时空穿梭者重点在于一个角度问题。所有这一切都意味着游戏开发人员必须在同一时间依次从每个玩家的视角去观察这个游戏世界；所有这一切都意味着游戏开发人员必须理解为什么程序员并没有魔弹（magic bullet）去对付这些与因特网相关的延迟所带来的限制。

4.2.6 参考文献

[Bernier01] Bernier, Yahn W., "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization," 2001 Game Developers Conference Proceedings, 2001.

[DMSO02] Defense Modeling and Simulation Office, *RTI 1.3-Next Generation Programmer's Guide Version 6*, September 4, 2002.

[Fujimoto00] Fujimoto, Richard M., *Parallel and Distributed Simulation Systems*, John Wiley & Sons, Inc., 2000.

[Singhal99] Singhal, Sandeep, et al., *Networked Virtual Environments*, Addison-Wesley, 1999.

4.3　使用程序生成游戏世界：避免数据激增

Sean O'Neil, Maxis, Inc.的合同制开发人员
s_p_oneil@hotmail.com

正如很多业界人士所知道的，游戏中的细节层次（detail level）正以惊人的速度增长。随着这种增长，高质量的专业游戏对数据量的需求也越来越大。为了对这种类型的数据激增（data explosion）进行管理，游戏开发团队开始大量地使用程序来生成游戏世界。很多开发团队以第三方产品的方式购买这些生成程序，这些产品包括 3D Studio、Bryce、Lightwave、TerraGen 和 MojoWorld。还有一些开发团队则编写他们自己的生成函数，并把它作为独立工具或是第三方产品的插件来使用。有些甚至可以在运行时动态地生成游戏世界。

本文将对在游戏引擎中的运行时数据生成——即在运行时动态地生成大型游戏世界进行介绍。本文不仅会对运行时生成的一些优缺点进行讨论，还会讨论一些在处理大型游戏世界时会遇到的伸缩性问题。本书附带的光盘中有一个范例，它包括一组可用的源代码和可执行程序。

4.3.1　运行时生成的优点

在运行时使用程序生成游戏世界有不少优缺点。对于那些对游戏世界的细节层次和大小要求都很高的在线 MMP 游戏来说，使用这种方法的优点远大于所带来的缺点。即使某些项目并不适合在运行时生成游戏世界，本文所介绍的技术仍然有助于预先生成（pregenerate）大型游戏世界，而所附带的代码则可以用来在开发过程中迅速地检查生成函数的输出。

运行时生成的最大优点是可以有效地减少数据尺寸。使用一个随机数种子和一个良好的随机数发生器，就可以生成整个游戏世界。只要每个玩家使用同样的种子和生成函数，他们就会看到相同的游戏世界。1984 年发行的游戏 *Elite*[Bell84]是一个对运行时生成游戏世界进行有效运用的经典例子。这是一个图形界面的太空战斗/交易游戏，它的游戏世界接近于无限，而它最初运行的平台仅仅具有 32KB 内存。有不少书籍和文章对这些技术进行了介绍[Lecky01]。

虽然计算机硬件自 1984 年以来有了很大的进展，但是由于细节层次的不断提高，同样的问题仍然存在。如果需要一个细致入微的大型游戏世

界，游戏开发人员必须采取某种方式来生成这些细节。可以选择在运行时生成，也可以预先生成这些数据并且保存在某个地方。运行时生成的另一个好处与它相对较小的数据量有关：它不仅使得游戏的在线发布变得更为快捷和廉价，还减少了游戏的存储量，这样就可以使玩家感觉到载入游戏世界所需要的时间减少了，不仅如此，如果使用某些特定的实现方式，还可能有效地减少系统内存的用量。

对于 MMP 游戏来说，数据量方面的改善非常重要。游戏的开发不再需要使用大型的数据文件，而是通过发布新的生成程序和生成 DLL（后者的更新并不频繁）来进行更新。地形的改变、游戏世界中静态对象的改变、新的贴图以及其他的一些相关数据都可以在各个客户端自动生成。当然，有些贴图必须手工进行绘制，但是目前大多数贴图都是完全靠 3D Studio Max 中的生成函数生成的，正是这些贴图可以被生成程序所取代。

为了让读者可以对程序生成所能达到的细节层次，以及所节约的空间有所了解，本文使用一个完整的行星——地球作为例子。地球的平均半径大约是 6378km。为了对它进行建模，可以从一个 20 面体开始，并且递归地把每个三角形分割为 4 个三角形（这么做可以为球体建立更好的近似）。如果递归地分割 20 次（20×4^{20}），就会生成大约 22 万亿个三角形，每个三角形的边长大约是 0.3km。存储这些高度贴图、纹理贴图、凹凸贴图等数据所需要的空间巨大无比，但是能够获得的细节层次仍然很低。在人们的视野中，任意 $90000m^2$ 大小的地块没有任何地形细节。想象一下，在这种情况下，山顶甚至整座小山都会被砍掉。但如果使用运行时生成，只需要用一个随机数种子就可以生成想要的任何细节。

4.3.2 运行时生成的缺点

运行时生成最明显的缺点就是对性能的影响。生成函数越复杂，生成游戏世界中每一个高度值的代价就越大。虽然不少技术和算法可以用来减少这方面的开支，但是永远都不可能把它完全地消除掉。因此必须谨慎地决定哪些数据应该在运行时生成；哪些数据应该在载入游戏关卡时预先生成；哪些数据应该在游戏更新时预先生成以及哪些数据应该在游戏发行时就预先生成。

这个方法的另一个缺点是它会增加引擎的复杂度并且延长开发时间。通常，这些生成函数本身并不复杂，但是运行时生成大型游戏世界需要用到动态的细节层次（Level Of Detail, LOD）函数，而它们可能会非常复杂。本文的示例中使用了一个基于实时优化自适应网格技术（Real-time Optimally Adapting Meshes, ROAM）[Duch97]的球体算法（spherical algorithm），它既简单又高效。下文将会对这个算法进行详细介绍。只需要稍事修改就可以让示例实现中的算法支持二维高度贴图。还可以通过简单地修改来处理预先生成的数据、动态生成的数据或者是两者的组合。

运行时数据生成的第 3 个缺点是大多数生成函数（譬如说随机分形函数）都很难控制。如果设计人员和美工不能对一个随机生成的游戏世界进行修改，那么很多游戏设计人员就不愿意把它发布出去。本文会介绍一些可以弥补这一缺陷的方法，但是这个缺陷总是会或多或少地存在着。最简单的方法是预先生成到某个特定的细节层次，然后动态地生成数据来补充其余的细节。更复杂的方法是要编写更容易配置的生成程序，从而可以在动态生成的数据上进行一些修改（譬如说在某处加一个弹坑、火山或是城市）。

最后，有必要注意到，在多人游戏中使用动态的 LOD 算法会导致一些独特的问题。无

论地形数据是怎样生成的，动态 LOD 算法总是根据当前的视角来显示细节。因为在多人游戏中每个玩家会从不同的视角观察同一个游戏世界，因此同一片地形在每个玩家的视角中是由不同的三角形集合而成的。这不仅会导致一些显示上的问题，譬如说一个玩家的脚会漂浮在地表之上或是被地面所掩盖，还会为对象渲染和碰撞检测带来问题。

LOD 算法在多人游戏中的这些问题已经在最近的游戏中得到了处理，关于这方面的讨论已经超出了本文的范围。*Tread Marks*[McNally]和 *Starseige: Tribes*[Sierra]的两个版本都是使用动态 LOD 算法的多人游戏。

4.3.3　地形生成算法的分类

现存的地形生成算法不计其数。然而，某些类型的算法更适合运行时生成，有些则在更高的细节层次上具有更好的伸缩性，另一些则善于创建更富于变化、更真实的地形。当然，所有的算法都必须是可重复的，也就是说，给予同样的参数，它们必须始终生成同样的地形。那些缺乏伸缩性的算法往往需要预先生成一个固定尺寸的网格（mesh）。这些算法称为“静态算法”，因为它们只能用于静态网格（static mesh）。

只需要对前面关于地球的例子中按比例提高细节层次的问题进行考虑，就可以很容易地理解，为什么静态算法不适用于动态生成了。最好的运行时生成算法可以通过一个函数（在这里本文称它为 GetHeight）来生成地图上任意点的正确高度。实现一个作用于预先生成的，高度贴图上的 GetHeight 函数非常方便，因此即使地形不完全是动态生成的，也可以使用这个机制。这些算法被称为“动态算法”，因为它们可以动态地为任意位置生成高度值，这使得地形中的任意部分都可以生成任意层次的细节。

1. 静态算法

有不少算法是完全静态的，它们先确定网格的大小，然后对整个网格进行操作。值得一提的是，因为它们控制起来较为方便，所以可以用它们来把地形预先生成到一定的细节层次，随后在其上继续使用动态算法补充细节。

断层线（fault line）算法[Elias]是一种很好的静态算法。这个算法使用一个固定尺寸的网格，并且把它随机地分为两片。对于二维地形来说，它生成一条随机线来切分一个正方形。对于三维地形来说，它生成一个随机面来切分一个球体。然后它提高断层一边的高度值并且降低另一边的高度值。经过几千次的迭代，就可以生成不错的地形。迭代的次数越多，生成的地形就越美观。

值得注意的是，只需要创建一个 GetHeight 函数（用于在运行时对具有成千上万个元素的断层线列表进行检查）就可以把这个算法变为动态的。但是这会导致两个问题。首先，这个操作非常慢。其次，在非常接近时，这些地形中的断层线会在垂直方向出现不连续。

2. 细分算法

很多人都曾经阅读过关于等离子算法（plasma algorithm）的资料，这是最基本的细分算法之一。等离子算法对一个二维的正方形或是三角形进行递归细分，并且把抬高网格中每个新顶点或是降低一个随机值。在每个层次，新顶点的最大变化会减少一个特定的系数。这个

系数被用来控制地形的粗糙程度。大多数算法的应用使用一个固定的系数，这会导致地形变得非常单调：所有的山脉和峡谷看上去都很相似。在不同的位置使用不同的减少系数可以改进这一算法，从而使地形的某些部分比其他部分更为平坦。

细分算法本质上是静态算法。每次为一个细分层次生成点时必须使用相同的顺序，这样才能从同样的随机数种子获得同样的网格。因此，必须在每个层次都生成完整的网格，这使得这个算法是静态的。值得注意的是，细分算法也可以变成伪动态（pseudo-dynamic）算法。由于对三角形网格进行细分时，我们通常会在正中间分割三角形的边，然而，编写一个可以迅速返回地形中任意点精确高度的 GetHeight 函数需要在细分树中进行无限递归，因此是不可行的。

然而，可以建立一个树状结构来模拟自适应网格算法（adaptive mesh algorithm）。目前只需要知道这些算法通常使用和地形细分算法非常类似的方式来对三角形进行细分就行了。惟一的区别是，自适应网格算法中对三角形进行细分的顺序依赖于摄像机路径（camera path），而这会弄乱随机数发生器的种子。通过为每个三角形生成一个新种子，可以来解决这个问题。怎么生成这个新种子并不重要，重要的是要保证同一个三角形总是能得到同样的种子，并且整个生成过程都由根节点上的种子决定。必须记住的是，只有使用不同的方式来应用这个方法，才能发挥其最大功效。

3. 动态算法

目前使用的大多数动态生成算法都是基于 Perlin 噪声[Perlin99]的。Perlin 噪声算法非常受欢迎，以至于三维芯片制造商都计划将它应用在视频硬件中。简单地说，这个算法可以通过一个随机生成的数字表（table of numbers）迅速地生成平滑的随机噪声。它使用一个特殊的点阵函数（lattice function），根据该函数的数字表，Perlin 噪声算法对接收到的参数进行改变以减少重复，其重复的几率基于数字表的大小。表中的每一项都表示了地图中整数位置的结果值，并且使用了线形插值，在这些整数位置之间进行平滑地插值。这种算法可以处理任意的维数，但是每增加一维就会让线性插值的代价呈指数增长。

根据 Perlin Noise 算法来生成地形的生成算法中，最著名的莫过于分形布朗运动了（factal Brownian motion，fBm）。简单地说，fBm 就是使用不同的波长（点阵中的位置）来调用 Perlin 噪声函数并且对结果进行加权求和。请参见等式 4-1 来获得一个更清楚的认识。变量 p 是地图上的点，它作为参数传递给噪声函数。变量 n 在 fBm 函数中被称为“度数（number of octaves）”，在实际应用中，它会在到达某个预设的值后停止，而不像等式中那样趋于无限。

$$\sum_{n=0}^{\infty}\frac{1}{2^n}noise\left(2^n p\right) \qquad \text{（等式 4-1）}$$

Ken Musgrave 在基本 fBm 方法的基础上发明了一种全新的概念：“多重分形（multifractal）”[Musg98]。多重分形本质上就是一种更为复杂的 fBm 变形。有些多重分形算法使用加权积而不是加权和，有些则使用了额外的偏移和参数。Kenton Musgrave 是 MojoWorld 背后的主要驱动力，MojoWorld 是一个优秀的虚拟世界创建程序，它结合了 Kenton 的研究成果和分形地形生成领域中其他学者的研究成果。

只需要一点数学知识，就可以任意地对 fBm 进行改变。然而，程序员可能花费了大量时

间却没有得到任何感兴趣的结果。这些分形算法的部分难点在于，即使一个很小的改变也会导致极大的（在很多情况下难以接受的）效果。需要一些时间才能知道某个修改会对输出产生什么样的影响，即便如此还是经常会得到令人吃惊的结果。

如果这成为一个比较大的问题，那么游戏设计人员可以使用预先生成的地图对生成程序的任何部分进行替换或补充，从而更好地控制生成的高度贴图。此外还需要记住的是，并不一定要使用基于 fBm 的程序，还可以使用任何基于噪声的函数。只需简单地生成一组断层线，然后使用基于噪声的函数，以这些断层线为参数来生成地形。

4.3.4　修改程序生成的地形

从某种角度上讲，大多数设计人员都想对程序生成的地形进行手工修改。当他们试图使用数学方法来生成不同类型的地形，并且试图把它们真正地整合在一起时，在创造性和可行性方面受到的限制将会使人大失所望。这时，很多人会放弃使用数学方法转而使用手工处理。幸运的是，手工修改的结果偶然可以应用于运行时由程序生成的游戏世界中。使用静态或是伪动态算法还可以来改变一个在运行时生成的游戏世界，从而迅速地加入一些定制，这些定制无法使用基于噪声的函数来实现，但如果使用手工实现的话又需要太多的时间。

本文将使用月球表面的陨石坑来举例。任何地形特征都可以使用同样的方式进行处理，但是陨石坑是一个很好的例子，因为它们是一种几乎在任何行星体表面都可以找到的常见地形特征，并且任何地形生成器都可以使用它们。当然，月球表面遍布着大小不一的陨石坑，那些大的在形成时可能毁灭了附近所有的生物，而那些小的则可能是由灰尘粒子形成的。要为月球进行精致的建模需要很高的细节层次和数以亿计的陨石坑，这一节将介绍怎样对此进行实现。实例代码实现了动态算法，但是并没有实现静态算法。

1. 手工修改

设计人员可能想要在地图上的某个特定位置放置一个陨石坑。设计人员应该有一个界面来定义碰撞点和半径，只需要按一下鼠标，就会有一个陨石坑呈现在他眼前。只需要为这个陨石坑对象创建一个 GetHeight 函数，他就可以把它和生成程序联系起来。这个方法可以从高度贴图中获得高度值，也可以通过某个数学函数来获得高度值。每当创建了一个新顶点，应该首先把它在地图上的位置传给全局的 GetHeight 函数，然后传递给陨石坑对象的 GetHeight 方法，来对全局函数返回的高度作一个偏移。

2. 使用静态算法修改

游戏设计人员可以把陨石坑链表作为整个游戏世界的一部分来进行创建和存储。虽然这增加了游戏世界的数据量，但是相对来说，需要进行手工修改的地方会比较少。如果设计人员需要花费大量的时间来手工添加成千上万个陨石坑，那这里面肯定有问题了。写一个分布函数来为游戏世界生成几千个陨石坑并不困难，因此这种工作应该由生成算法来完成，并在载入游戏时进行。

当然，如果运行时，生成引擎要为每个新顶点都去检查成千上万个陨石坑的位置，那肯

定会出现很大的停顿。要解决这个问题，设计人员只需要用一个经过优化的碰撞检测来确定需要为地图上的指定位置检查哪些陨石坑就行了。本文所提供的例子把一个行星球体映射到一个正方体的 6 个面上，每个面都是一棵四叉树的根。这样不仅可以使用链表来存储这些陨石坑，还可以根据每个陨石坑的位置和半径来把它们的指针保存在四叉树中。要获得某个新建顶点的高度，需要先调用全局的 GetHeight 方法，然后遍历这些四叉树来确定需要对哪些陨石坑进行检查。

3. 使用伪动态算法修改

现在开始介绍最有趣的部分：一个可以在程序生成的游戏世界中加入上亿个陨石坑的实时算法，并且不用使用任何内存，也不会重复（如果不考虑随机数产生器的限制以及随机数种子大小的话）。回想一下前面介绍的把细分算法变为伪动态的方法，这个解决方案很容易理解。

把包含了陨石坑信息的四叉树转化为一棵虚拟树（也就是一颗并不真实存在于内存中的树），然后为树中每个节点生成一个新的随机数种子。在树中的每个节点上，使用这个种子随机地生成一些陨石坑，然后继续下一个层次。不同于在一颗真实的四叉树中所使用的指针和递归，这棵虚拟四叉树的 GetChild 方法只返回下一个随机数种子，而不是树中的下一个指针。在本文所提供的实现示例中，虚拟陨石坑树有 3 个参数：开始生成陨石坑的层次（或是最顶层），停止生成陨石坑的层次（或是最底层），每一层的陨石坑数目。

使用非常简单的代码来进行一次简单的循环就可以实现这棵虚拟树。除了参数以外，它所需要的内存仅仅是一个树节点对象的空间，而这块空间作为 GetHeight 函数中的临时变量分配在栈上。如果把最低层次设为 20，游戏世界将会包含大约 1.3 万亿个陨石坑。虽然使用了陨石坑以后，引擎会运行得慢一点，但它仍然是实时的。不过值得注意的是，太多的陨石坑会让地形变得非常粗糙，并且需要更多的三角形来实现同样层次的细节。

```
float GetHeight(CVector vPosition)
{
   // First get the base height for vPosition
   float fHeight = GetBaseHeight(vPosition);

   // Then determine the face vPosition is in,
   // along with the x and y face coordinates
   float x, y;
   int nFace = GetCoordinates(vPosition, x, y);

   // Skip over the crater levels we don't want,
   // maintaining the proper child seed
   int nSeed = m_nCraterSeed + nFace;
   for(int i=0; i<m_nCraterTopLevel; i++)
      nSeed = GetChildSeed(nSeed, x, y);

   // Finally, loop to the bottom of the tree,
   // adding the height of the craters in each node
   CCraterMap node;
```

```
        for(; i <= m_nCraterBottomLevel; i++)
        {
            node.Init(nSeed, m_nCratersPerLevel);
            fHeight += node.GetOffset(x, y);
            nSeed = GetChildSeed(nSeed, x, y);
        }
    }
```

4. TANSTAAFL 和 NEWYDIFR

是时候提个醒了："世上没有免费的午餐（There ain't no such thing as a free lunch，TANSTAAFL）"，"说起来容易做起来难（Nothin's easy when you're doing it for real，NEWYDIFR）"。前面描写的伪动态算法有一些显而易见的缺点。虽然这些缺点在静态算法中很容易解决，但是和之前谈到的分形算法一样，它们在动态算法中却非常难以控制。因此，所有这些问题都可以通过把静态和动态算法组合在一起来得到缓解。设计人员可以为固定数量的层次使用一个真实的四叉树，然后在更低的层次上切换到虚拟树。对于可以由动态算法解决的问题，这种解决方案的代价太大了。

第一个问题是陨石坑不能跨越它们所在的四叉树节点的边界。这意味着在较高层次的节点边界上，很长的区域明显地缺少陨石坑。虽然一般的玩家不会很快意识到这点，但是大多数在月球上历险足够长时间的玩家会注意到这个问题。设计人员可以在静态算法中简单地修正这个问题，如果一个陨石坑跨越多个四叉树节点的话，只需要把它的指针放到多个四叉树节点中去就行了。

第二个问题是四叉树上的节点并不是完美的正方形。作为开始的那个立方体的每个面都会投影到球体上，这会导致节点发生变形。这种变形在每个面的角上尤为严重。为了保证没有陨石坑跨越节点边界，示例代码会在节点坐标系中，对陨石坑的半径和陨石坑到边界的距离进行计算，这意味着陨石坑的形状也会和节点一样发生变形。静态算法中并不存在不能跨越节点边界这一限制，因此这又是一个只会在动态算法中出现的问题。可以把这些计算投影到世界坐标系中，从而对动态算法进行修正，但是这会导致更严重的性能下降。

另一个问题也是与陨石坑和现存地形之间的交互有关的。目前的示例实现只是把所有的高度偏移加起来，因此重叠的陨石坑彼此之间会有相加效应（additive effect）。这是非常不真实的。由于陨石坑的自然特性，它们会把现存的地形完全地消除。因此，新产生的大陨石坑会把它们半径之内已经存在的小陨石坑完全消除，而已经存在的大陨石坑的部分边界也会被新产生的小陨石坑消除，而不是相加。

可以非常简捷地为每个陨石坑随机产生一个年龄，然后把所有对一个新顶点有影响的陨石坑根据年龄进行排序。这样就可以使用一些简单的逻辑来处理它们之间的交互。处理这个问题的关键在于要找到在创建某一个陨石坑时，列表中每一个陨石坑中心点上地形的高度。每个陨石坑的这一"基本高度"计算起来很复杂，因为它需要一个 GetHeight 函数来把所有在当前陨石坑之前创建的陨石坑的高度加起来。由于一个老陨石坑的基本高度会影响新的陨石坑的高度，因此必须按照正确的次序来计算这些基本高度。可以在静态算法中预先计算这些值，但是由于它的代价过大，不能使用在动态算法中。

4.3.5 高效地渲染程序生成的地形

球体 ROAM 算法只是一个关于怎样对程序生成的地形进行高效渲染的例子。目前，不少其他的动态 LOD 算法被广为使用。[Duch97]、[Hoppe91]、[Lind96]、[deBoer00]、[Hakl]和[Hill01]都是一些很好的例子。无论选用什么算法，最重要的是要确保它是针对运行时程序生成进行过优化的。

1. ROAM 算法

本文只会对 ROAM 算法的示例代码进行一些基本的介绍从而使读者对本节中介绍的概念有所理解。请参见[ONeil2]以获得更详细的解释。

ROAM 算法通常从一个包含了两个直角三角形的正方形开始。新顶点必须添加在三角形最长边的中点上，这样就把三角形分割成了两个更小的直角三角形，并且每一个的大小正好是分割前的一半。除非用于分割的边在地图的边缘上，否则它会被另一个三角形共享，因此必须对另一个三角形也进行分割以避免网格出现裂纹。把小的三角形合并起来也需要实现同样的功能来避免裂纹。ROAM 算法的球体版本从一个立方体的 6 个正方形面开始，这些面都投影到一个球体上，并且所有三角形的边都是共享的（也就是说，这个地图没有边界）。

通常程序使用一棵维护了父子关系的三角形二叉树来实现 ROAM。在渲染每一帧时，都需要进行优先级检查来判断哪些三角形需要被分割或合并，然后对获得的所有叶节点进行渲染。这个优先级检查会先根据当前的视角来建立一个误差标准（error metric），然后对三角形进行分割和合并，以达到一个特定的帧频或是三角形总数。也可以选择一个最大误差阈值，并且对超出这个阈值的三角形进行分割或合并。一个简单的依赖于视角的误差标准是用三角形中的误差总量（或是一个新顶点被创建时所需要移动的数量）除以摄像机和三角形之间的距离（因此距离较远的误差相对于距离较近的误差来说，权值比较低）。

需要注意的是，大多数其他的动态 LOD 算法也使用同样的，基于视角的误差标准和优先级计算，因此这里所介绍的大多数优化同样适用于那些算法。

2. 重要的优化

对于运行时程序生成来说，最重要的优化就是最小化对 GetHeight 的调用次数。因为 ROAM 中的优先级判断需要每个三角形的下一个可能顶点的高度值，因此第一个优化就是生成这些高度值，并把它们缓存在三角形对象内。在每一帧中，都可以使用这些缓存了的值来判断优先级。如果我们要对某个三角形进行分割，就可以使用缓存值来计算新顶点，然后再调用 GetHeight 函数来为当前三角形的子三角形计算下一个高度值。

即便如此，程序对 GetHeight 函数的调用次数仍然是它在分割时应该被调用的次数的两倍。这是因为在对每个子三角形进行分割时，都会有一个相邻的三角形与它共享这个新的高度值。因此在每个三角形被创建时，程序都可以轻松地判断与它相邻的三角形是否已经计算了这个高度值。

接下来可以考虑怎样对合并函数进行优化。GetHeight 函数永远都不应该在合并时被调用。大多数 ROAM 实现维护了一棵三角形二叉树，每一个父节点都具有一个预先计算出的高

度值，即使它需要被再度分割。在示例实现中，父节点并不保存在树中，但是算法知道正在删除的是哪个顶点，因此可以很方便地在删除这个顶点前获得它的高度值。

此时，程序很难再对分割和合并算法作进一步的优化，因此这时候，需要优化一下分割和合并的优先级检查算法了。这对于任何动态 LOD 算法来说都是一个重要的优化，而对于运行时地形生成来说，则尤为重要。首先可以对基于视角的优先级计算进行优化。把位于视平截体或视图角度外部的三角形的优先级降为 0，从而使它们可以尽可能地合并。对于二维地形来说，位于远截面（far clipping plane）之外的三角形应该被丢弃；对于三维球体地形来说，位于地平线以外的三角形应该被丢弃。程序还必须确保用于分割和合并的优先级计算是相同的，否则三角形可能会在各帧中被反复分割和合并，从而导致对 GetHeight 不必要的额外调用。

如果所有其他的优化都达不到预期的效果，可以为任意给定帧中的分割和合并次数做一个限制，这也会限制对 GetHeight 方法的调用次数。仅当摄像机移动或旋转过快时，这个限制才会产生问题，因为摄象机移动或旋转过快会导致每一帧都需要生成大量的地形来保持一定的细节层次。不幸的是，对分割/合并操作的次数进行限制，或是按照某个帧频目标/三角形数量来进行分割/合并都需要对三角形按优先级进行排序，这样才能确保优先级高的三角形可以在优先级低的三角形之前被分割。

如果必须使用上述的某种限制操作，桶排序（bucket sort）可以用来对三角形进行排序，它的复杂度是 O(n)。何况程序本来就需要通过遍历三角形树来计算优先级。因此从性能角度来说，这个算法很快。因为误差标准等同于误差/距离，并且某个三角形的投影大小随着它到摄像机的距离呈线性递减，所示误差标准可以比较精确地表示出与这个误差等同的，屏幕上的像素数量。所以，只需要创建一个桶数组，并且用近似的误差像素数作为索引来把三角形放入这个数组就可以了。不需要对同一个桶中的三角形进行排序，因为对于玩家来说，它们之间没有视觉上的区别。

```
void SortedROAMUpdate(CVector vCamera)
{
    float fPriority;
    int nPixels;
    for(every diamond)
    {
        fPriority = diamond.GetPriority(vCamera);
        nPixels = GetPixelOffset(fPriority);
        diamondbucket[nPixels].add(diamond);
    }
    for(every triangle)
    {
        fPriority = triangle.GetPriority(vCamera);
        nPixels = GetPixelOffset(fPriority);
        trianglebucket[nPixels].add(triangle);
    }

    // Merge all diamonds with less error than
    // max triangle error
```

```
    while(diamondbuckets.min < trianglebuckets.max )
        merge(diamondbuckets.min);

    // Split triangles up to desired count
    while(trianglecount < desiredcount)
    {
        if(exceeded max splits per frame)
            break;
        split(trianglebuckets.max)
    }
}
```

4.3.6 生成贴图

虽然让程序来生成游戏世界是一个非常好的方法，但是如果没有任何贴图来美化它的话，这个方法根本行不通。然而，如果游戏世界很大以至于无法使用单一的贴图，或是一组固定大小的贴图来达到预期的效果，这个问题就会变得非常棘手。

1. 简单的解决方案

示例实现中生成贴图的方法非常高效。本质上，程序员可以创建一个很小的一维贴图并把其元素映射到地形中的某个特定高度。每个顶点的贴图坐标是基于顶点的高度计算的。在过去的实现中使用的是二维贴图，它把第二维作为纬度来使行星被冰雪覆盖。为了避免在行星表面出现明显的水平颜色带，它使用了噪声函数来对纬度进行改变，从而使得行星表面看上去具有合理的气候区域。在那些支持多重贴图（multitexture）的显卡上，设计人员还添加了一层基于噪声的细节贴图，来避免让地形在接近时显得过于单调。

示例实现乍看上去可能还不错，但是它实际上有一些很大的缺陷。最难解决的问题是行星的颜色本质上是顶点色。这样一来，只要改变一些细节层次就会产生一个完全不同的外观。在对三角形进行分割和合并时，地形会在垂直方向发生细小的变化，而颜色则会发生很大的变化。如果摄像机离表面很近的话并不会有什么问题；但是如果是站在太空轨道的角度，那么相邻两帧里的整个岛屿可能会时隐时现。即使这算不上什么问题，基于顶点的海岸线还是会非常的难看。除非玩家的电脑非常高级，可以计算到很高的细节层，次从而使得顶点细节接近于像素细节这个问题才可以得到解决。但是即使在 CPU 和 GPU 技术高度发展的今天，也仍然达不到这样的效果。

示例实现中确实使用了一个临时的方案来减轻颜色跳跃的问题。这个方案使用和计算贴图坐标相同的函数来预先生成一张具有固定分辨率的全局贴图。从太空轨道中看起来，它的效果很好，但是当接近表面时会很难看，因此示例中的代码要在全局贴图和细节贴图之间进行平滑地切换。这种方案的效果相对较好，但是仍然不能满足高质量游戏的需求。

2. 更复杂的解决方案

不少实现方法可以解决这个问题，譬如说[Ulrich00]，它为动态网格生成了一棵贴图四叉树（或是类似的细分结构）。当某个特定节点的三角形网格被过度分割后，它就把这个节点分

割成两个，并且为每一个新节点动态生成贴图。出于性能上的考虑，这些贴图必须相当小。为了使 CPU 和 GPU 的并行处理能力最大化，需要用一个独立线程来生成这些贴图。

为了可以正确地计算顶点的贴图坐标，并且使用正确的贴图来渲染三角形，必须把这棵贴图树与自适应网格算法紧密地结合在一起。一个比较可行的做法是，根据树在最顶层的正方形中的位置来设定贴图坐标，然后为树中每个节点计算出一个贴图转换矩阵。当然，本文假设不存在浮点数的精度问题（请参见后面“大型游戏世界中的比例问题”这一小节）。

4.3.7　在程序生成的地形进行碰撞检测

现在，可以对运行时动态生成的游戏世界进行有效的渲染了，那么怎样有效地处理碰撞检测呢？网格是不断变化的，因此 BSP 树以及任何预先进行分割的方法在这里毫无用武之地。更糟糕的是，位于摄像机的视平截体（view frustum）之外的三角形的优先级是 0，因而它们会被不断地合并直至消失。万一有个玩家倒退着进入陨石坑，或是在下落时抬头或环顾四周，那又会怎样呢？很多进行精确的碰撞检测所需要用到的三角形此时根本就不存在。在 MMP 游戏中存在类似的问题：每个玩家看到的三角形集合是不同的。

这里的答案是：避免在碰撞检测时进行三角形判断。只需要在一个对象的位置上调用 GetHeight 函数，就可以迅速地判断是否发生了碰撞。虽然 GetHeight 函数很大，但它是 100% 精确的，并且，在处理复杂网格时，它实际上还是要比多边形相交测试小很多。对于那些位于表面上的对象来说，可以预先进行包围半径（bounding radius）测试。只需要使用包围盒（bounding box）下面的 4 个角就可以了。这是因为当对象向一堵陡峭的墙移动时，除非位于中心下方的那个点与包围半径相交，否则它不会被检测到。如果初步测试指出一个可能的碰撞，就可以使用 GetHeight 函数来对这个对象网格顶点的一个子集进行更彻底的测试。

对于那些在表面上移动的物体来说，游戏检测人员可以对包围盒底部的 4 个角加以利用，还可以借助包围盒中每一点所处地形的高度来对它们进行位置和斜率判断。斜率上的急剧变化可能意味着遇到了一堵陡峭的墙，这时就需要进行碰撞检测。在多数情况下，并不需要在每一帧都进行 4 次计算。通常每一帧的包围盒与上一帧的包围盒是重叠的，如果新包围盒的一个角落在旧包围盒中，就可以用前一个包围盒的斜率进行插值。甚至可以在每一帧只生成一个新的高度值，譬如说轮流计算选定角的值，或者选择离移动方向最近的那个角来计算。

如果以这种形式调用 GetHeight 函数仍然太复杂，可以在特定的对象和区域周围生成固定尺寸的网格片（mesh tile）。这种方法需要更多的内存而且并不是很精确，但是适合像《亚瑟暗世纪》（*Dark Age of Camelot*）这样的游戏，这种游戏的游戏世界中可能有很多城市，每个城市中都有成百上千个四处移动的物体。游戏世界中还可能有很多巨大的野外区域，这些区域中只有少量的物体。在这种情况下，比较有意义的做法是为城市生成网格并且为物体和地面之间的交互编写特殊的代码。事实上，甚至可以假设城市中地形都处于同样的高度，这样在整个城市中只需要对这个固定的高度值进行检查就可以了。

4.3.8　大型游戏世界中的比例问题

为一个大型游戏世界进行建模和渲染需要处理的另一个关键问题是比例和精度。一个 32

位的 float 数据类型最多只有 6 个精确的有效数字，而一个 64 位的 double 数据类型最多也只有 15 个有效数字。如果可接受的精确度损失是一毫米，那么 float 在 1000 公里左右就开始不够精确了，而 double 在一万亿公里时就不够精确了。

1. 视频硬件/驱动程序的问题

硬件加速的 Z 缓冲通常使用 16、24 或是 32 位数据类型。为了便于理解，假设要显示的是整个地球。地球的直径是 12756km，如果使用 32 位 Z 缓冲，从低空轨道来看，一厘米左右的距离就将失去精度。而一个更小的 Z 缓冲则基本没有任何用处。

如果试图从太空中观察地球的话，则会导致更大的精度损失。如果从距离地球约 384400 公里的月球轨道上进行观察，任何位的 Z 缓冲都将变得毫无用处。更坏的是，即使游戏开发人员愿意牺牲 Z 缓冲的精度，大多数显卡驱动程序也没有能力渲染 30000 到 50000 个像素以外的三角形。这里的根本原因是因为在驱动程序（或是 GPU）的矩阵转换中，浮点运算操作本身就具有精度损失。

2. 游戏世界建模的问题

这些精度上的损失并不仅限于显卡和驱动程序。物体位置和速度也存在精度损失。如果要使用 float 来为整个太阳系建模（冥王星轨道的半径接近于 50 亿 km），并试图改变一个接近冥王星轨道的物体的位置，游戏开发人员就需要在每个坐标轴上，按照大约 5000km 的距离进行四舍五入。因此有些浮点运算单元（Floating Point Unit, FPU）可能被舍去。这意味着一艘绕着冥王星航行的飞船，必须在每一帧（不是每秒）中在每个坐标轴上的移动都超过 5000 公里才能进行移动。飞船里的玩家会飞快地经过冥王星以至于看不到它。不仅如此，由于这 5000 公里的跳跃是与各个轴平行的，因此一个物体即使移动了，它也很可能没有向正确的方向移动。

3. 简单的解决方案

所有这些问题其实都可以轻易地解决，但是如果没有预先考虑并且认真地处理，它们将会变成非常严重的问题。双精度运算的代价很大，因此必须尽可能地避免。然而，如果游戏世界实在太大，那就可以在某些关键位置，使用双精度浮点数来解决上述的一些问题，而这个关键位置就是对象的位置。相对于使用 float 来表示地球半径大小的量来说，使用具有 15 个有效数字的 double 可以更精确地表示比冥王星轨道长 1000 倍的长度。

开发人员可以继续使用单精度的 float 来表示所有其他值，而双精度操作只用在要改变物体的位置，或者要与其他物体进行位置比较的时候，而所有这些操作都是加减操作。假设有一艘飞船绕着冥王星航行，一旦把飞船的位置向量减去冥王星的位置向量，就可以把这些 double 表示的数据转换为 float 类型，并且在余下的运算中使用单精度浮点运算。其实，一旦正确计算出了物体之间的距离和方向，余下的运算中有一些小范围的精度损失也是无关紧要的。

要解决矩阵转换上的问题，可以在计算观察矩阵（view matrix）的时候假设摄像机总是位于原点，然后在计算物体矩阵（model matrix）的时候，把每个物体的位置按照摄像机的位置进行偏移。这使得物体在摄像机附近的精度损失总是接近于 0。对于远距离的星球来说，虽然精度损失的绝对值可能很大，但是由于视觉上的损失总是基于物体与摄像机之间的距离，

因此这一损失在屏幕上引起的误差都不到一个像素。

一些简单的措施可以用来减轻与远距离渲染和 Z 缓冲相关的问题。可以先把远截面设置为一个合理的距离，比如说 1000km。忽略那些超出这个距离的小物体，因为它们是根本看不到的。可以把远处的大型物体按照与摄像机的相对距离来进行排序，然后使 Z 缓冲无效，从后往前进行渲染。为了避免驱动程序渲染过程中的问题，远处物体的尺寸和距离可以按比例缩减，这样就可以使它们的距离和远截面一样了。尺寸的缩减和距离是呈线性关系的，因此使用同样的比例对行星的大小和距离进行缩减后，它在屏幕上的大小不会改变。

然而，这个方法存在两个问题。第一个问题是，因为没有使用 Z 缓冲，它只适用于凸物体，而凹物体需要 Z 缓冲才能得到正确的渲染。因此，这对于渲染远距离的球形行星来说还不会构成什么问题。第二个问题是，假设远截面在 1000km 处，那么一个半径为 6378km 的行星总是会按照远截面的距离进行渲染，并且不使用 Z 缓冲。如果根据行星表面（而不是球心）来计算距离的话，一切正常；但是反之，如果按照到球心的距离来计算的话，一个 1000 公里的远截面对于从低轨道观察这个行星来说，可能短得都看不到地平线。

一个可行的解决方案是选取与摄像机最接近的行星（摄像机永远都不应该同时和两个行星都相距那么近），如果它位于一个特定的距离之内，就把它按比例地缩减到使它的水平距离与远截面相匹配。也许还有更简单的好办法，但是这个方法目前看起来也足够了。

4. 替身（Impostor）渲染

游戏开发人员还可以使用一项更为复杂的技术来提高渲染性能和 Z 缓冲精度：替身渲染。替身就是一个动态生成的公告栏（billboard）。可以先把复杂物体渲染到后备缓存（back buffer），或是在视频内存中分配的像素缓存中，然后把这个缓存拷贝到一个贴图中，在接下去的几帧里就可以把这个贴图作为一个公告栏来渲染这个物体。和自适应网格算法（adaptive mesh algoritm）一样，替身使用一个误差标准来确定什么时候需要对这个贴图进行更新。为了计算这个误差标准，必须把每次替身更新时摄像机和物体之间的向量保存下来。每一帧都会把摄像机和物体之间的当前向量和这个保存的向量进行比较。如果它的改变程度超过一个特定的量，就需要更新这个替身了。

替身渲染可以提高 Z 缓冲的精度，因为每个替身都有它自己的视平截体，也就是说每个替身都具有它自己的远截面和近截面。为了优化对贴图分辨率以及 Z 缓冲精度的使用，视平截体的 6 个面应该尽可能地贴近这个物体。

其他因素（譬如说光照的改变）也会影响替身的更新频率。替身本身也可以包含一组物体。由远处的星星组成的整个天空可以被渲染到 6 个替身中去，只要摄像机没有在行星间移动，就可以在很长时间里复用这些替身。如果在这些替身中的物体会彼此独立的移动，那么这种移动也会引入误差，最终导致必须对这些替身进行更新。

当然，替身技术并不仅限于渲染星球。只要有足够的视频内存，替身技术可以通过减少每一帧所需渲染的三角形数量来极大地提高性能。替身技术还可以用来渲染细节丰富的树木，甚至是整片森林[Remolar02]。适用于对云、雾进行渲染的体渲染（volumetric rendering）技术通常也是作为替身来处理的[Harris01]。那些具有大量建筑物的城市场景通常也是用替身进行渲染[Franc97]。游戏设计人员可以很方便地确定一个替身在屏幕上的大小，并且把那些较小的替身放到较小的贴图中，这样就可以创建一个替身层次（hierarchy），用较小的替身来构

造较大的替身。

然而，使用替身还需要解决很多技术问题。替身都是部分透明的，因此显卡必须支持在后备缓存，或是像素缓存，中使用目标 alpha（destination alpha），这样最终的 alpha 值会被正确地拷贝到贴图中。不仅如此，如果过于频繁地使用替身，视频内存会很快耗尽。幸好可以对远处的替身使用较小的贴图，而贴图缓冲（texture cache）技术也可以有效地控制替身所使用的视频内存总量。

除此之外，替身技术还具有透明公告栏（transparent billboard）技术的所有问题。在绘制透明公告栏时，必须先根据距离进行排序，然后在 alpha 混合打开的情况下，按照从后到前的顺序进行渲染，这可能会带来一些性能问题。并且，本质上公告栏都是平面的，如果有两个公告栏彼此相交，或是一个普通物体和一个公告栏相交，深度测试就会给出错误的结果，从而导致视觉上的瑕疵。当摄像机离得很近时，公告栏会变得不太好看。对于替身技术来说，可以在摄像机离得很近时切换到普通的渲染方式。

怎样解决公告栏相交问题已经超出了本文的范围。毕竟行星不仅数量很少、尺寸很大，而且它们之间相距很远，因此通常不会遇到这个问题。[Harris01]中介绍了一个解决这个问题的好例子。

4.3.9 总结

目前最尖端的游戏所需要的数据量非常大，而运行时程序生成游戏世界对于管理这样的数据量来说，是一个非常有效的方法。它有很多显著的好处，尤其适用于那些需要在线分发游戏数据的 MMP 游戏。因为 CPU 运算能力的发展速度相对于调制解调器技术来说要快得多（很多玩家还没有 DSL 或者是有线通），无论对于玩家还是游戏服务器来说，尽可能地在客户端生成游戏数据都是既迅速又方便的。游戏系统既可以在运行时、载入时生成数据，也可以在更新时一次性生成。这种方法即使对于非在线发布的游戏来说也同样有效，因为没有人喜欢在安装或是运行游戏时更换光盘。

本文中提及的一些问题可能会让开发人员无法在开发高质量的游戏时使用运行时程序生成。幸运的是，不仅有很多人正在研究怎样解决这些棘手的问题，硬件的性能也在飞速攀升。即使本文中的一些想法在今天还不可行，然而或许它们在一个很长的开发周期完成后就会变得可行。正如数据激增的问题不会得到减轻而只可能会更为严重一样，程序化生成技术的进步也只会越来越快。

4.3.10 参考文献

[Bell84] Bell, Ian, "The Elite Home Page," *http://www.iancg- bell.clara.net/elite/*, 1984.

[deBoer00] de Boer, William H., "Fast Terrain Rendering Using Geometrical MipMapping," *http://www.flipcode.com/tutorials/geomipmaps.pdf*, 2000.

[Duch97] Duchaineau, Mark, "ROAMing Terrain: Real-time Optimally Adapting Meshes," *http://www.llnl.gov/graphics/ROAM/*, 1997.

[Elias] Elias, Hugo, "Spherical Landscapes," *http://freespace.virgin.net/ hugo.elias/models/m_*

landsp.htm.

[Franc97] Sillion, Francois, "Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery," *http://www.cs.unc.edu/%7Eibr/other_pubs/Sillion_Impostor_ Eurographics_97.pdf*, 1997.

[Hakl] Hakl, Henri, "Diamond Basics," *http://www.cs.sun.ac.za/~henri/ diamondbasic.html*.

[Harris01] Harris, Mark, "SkyWorks Cloud Rendering Engine," *http://www.cs.unc.edu/%7Eharrism/ Sky Works/index.html*, 2001.

[Hill01] Hill, Dave, "Rendering Planets In Real Time," *http://www.dgp.toronto.edu/~dh/research .html*, 2001.

[Hoppe91] Hoppe, Hugues, "Hugues Hoppe's Home Page," *http://www.research.microsoft.com/~hoppe/*, 1991.

[Lecky01] Lecky-Thompson, Guy W., *Infinite Game Universe: Mathematical Techniques*, Charles River Media, 2001.

[Lind96] Lindstrom, Peter, "Real-Time, Continuous Level of Detail Rendering of Height Fields," *http://www.cc.gatech.edu/gvu/people/peter.lindstrom/papers/ siggraph96/*, 1996.

[McNally] McNally, Seumas, "Tread Marks—Battle Tank Combat and Racing," *http://www.treadmarks.com/*.

[Musg98] Musgrave, F. Kenton, Texturing & Modeling: A Procedural Approach, AP Professional, 1998.

[ONeil00] O'Neil, Sean, "A Real-Time Procedural Universe," *http://home.attbi.com/~s-p-oneil/* , 2000.

[ONeil1] O'Neil, Sean, "A Real-Time Procedural Universe, Part One: Generating Planetary Bodies," *http://www.gamasutra.com/features/20010302/oneil_01.htm*, 2001.

[ONeil2] O'Neil, Sean, "A Real-Time Procedural Universe, Part Two: Rendering Planetary Bodies," *http://www.gamasutra.com/features/20010810/oneil_01.htm*, 2001.

[ONeil3] O'Neil, Sean, "A Real-Time Procedural Universe, Part Three: Matters of Scale," *http://www.gamasutra.com/features/20020712/oneil_02.htm*, 2002.

[Perlin99] Perlin, Ken, "Making Noise," *http://www.noisemachine.com/ talk1/*, 1999.

[Remolar02] Remolar, I., "Real-Time Tree Rendering," *http://nuvol.uji.es/~ribelles/Investigacion/Papers/dlsi_01032002.pdf*, 2002.

[Sierra] Sierra, "Sierra: Tribes 2 - Team Combat on an Epic Scale," *http://www.sierra.com/*.

[Ulrich00] Ulrich, Thatcher, "Quadtree tiling / unique full-surface texturing," *http://www.tulrich.com/geekstuff/tiling/*, 2000.

[VTP97] The VTP Group, "Virtual Terrain Project," *http://www.vterrain.org/*, 1997.

4.4 为固定大小的对象编写一个高速有效的分配器

Tom Gambill, NCsoft
tgambill@ncaustin.com

在任何复杂到一定程度的 C++程序中，尤其是在目前那些大型 MMP 游戏中，内存分配可能是主要的性能瓶颈之一。随着角色在游戏世界中走动以及物体进出视野，需要不停地把游戏数据载入内存，或是把它们从内存中清除，这必然会导致大量的内存碎片。由于程序会按照不同的次序向公共堆（common heap）分配和释放不同大小的对象，那些已分配的内存块之间的空余内存会形成大量的空闲块。随着这些碎片和空闲块的产生，维护所有内存块信息的数据结构将会迅速地增长，从而进一步增加系统的内存负担，这样，内存分配就会变得越来越慢。

对于像这样支持任意长度内存块的分配器来说，内存碎片是最显著的问题。因为这些分配器是通用的，它们通常会比针对某个特定的内存分配模式（allocation pattern）设计的定制分配器更慢，并且更为低效。尽管 C++程序中存在着很多不同的分配模式，但本文仅对一种在 C++中非常常见的模式进行讨论。它通常被称为固定大小分配器（fixed-size block allocator），也就是说，从一个特定的内存池（memory pool）中分配的所有内存块都具有同样的大小。由于所有的内存块的大小都相同，游戏系统就不会再因为碎片而浪费内存。任何新的分配都可以使用内存池中任意的空闲槽（slot），而要找到一个空闲槽，只需要返回指向下一段空闲内存的指针。

4.4.1 C++中的内存分配

如果观察一下 C++中通常的内存分配操作（分配某个类的新对象）是怎样进行的，就可以知道一个具有固定大小内存块的分配器为什么会那么有用了。游戏的程序编写员已经很多次地读到过下面这样的代码：

```
T* objectPtr = new T;
```

如果能够让程序中不同类型的对象都从不同的内存池中分配内存，那么就可以彻底消除碎片。而且事实上并不需要为每个类实现这一功能。游戏系统会频繁地分配很多对象，像矩阵、向量、浮点数、整数等，这些对象更适合于分配在栈上。这不仅可以减少内存/高速缓存不一致带来的问

题，也可以避免对指针的使用，还可以避免为那些只使用一次的对象类型分配独立内存池而带来的额外负担。游戏系统的运行只需要为那些在程序的运行周期内反复分配和释放的常用对象实现固定长度分块的分配器就行了。譬如说一个用来在大型游戏世界中，表示独立地形块的类。

游戏开发人员必须记住一些事情。不要为特定的类分配过多的内存。如果不使用那些多分配的内存，那就是在浪费内存。同样，也不应该为某一类分配得过少，否则就要频繁地为额外的对象进行重新分配，这会重新陷入本文想要解决的问题。此外，使用这种技术的内存池，在它所有的分块被释放前它本身不能释放。上述理由让程序员必须仔细地考虑在最初时应该为不同类型的对象分配多少分块。

4.4.2　一个简单的向量分配器

一个内存分配器通常包含了一块用来分配对象的空间，以及来跟踪这块空间的使用的某种形式的数据结构。游戏设计希望这种数据结构的尺寸很小，并且能够高效地进行分配和释放操作。假设用一块连续内存来保存对象，它的长度是类 T 大小的整数倍；还用一个 T 的指针数组/向量来保存空余的内存块，最初，内存池中每一个分块的指针都在这个数组/向量中；还使用一个索引来指示下一个空余内存块，最初它为 0。每当有一个内存块被分配时，它会返回在索引位置的元素，并且把索引增加 1，使它指向下一片可用的空闲内存。释放一个分块时，把索引减 1，并且把指向这个空闲内存块的指针放回这个指针数组/向量中去。

下面是一个非常简单的分块分配器的类模板。它和本文中其他的代码一样，可以在附带的 CD-ROM 中找到。

```
template <class T, size_t numBlocks>
class SimpleAllocator
{
public:
   SimpleAllocator() : nextAllocation(0)
   {
      for (size_t i=0; i< numBlocks; _++i)
         pPointers[i] = &(pool[i]);
  }

   T* AllocateBlock()
   {
      return pPointers[nextAllocation ++];
  }

   void ReleaseBlock(T* pBlock)
   {
      if (pBlock)
         pPointers[--nextAllocation] = pBlock;
   }
```

```
private:
    T     pool[numBlocks];
    T*    pPointers[numBlocks];
    size_t nextAllocation;
};
```

如果需要更多的对象，就必须为 T 类的对象分配另一块内存池。这相对来说很直接：程序员只需要管理一个内存池向量的数组。

首先需要把初始化工作从构造函数中移出来，并且放入 AllocateBlock()（分配分块）方法中，这使系统可以在需要时分配新的内存池向量。使用 STL 的 vector 类来保存这些指向内存分块的指针，这可以把管理指针数组的细节封装起来，以使程序更为清晰。需要注意的是，这些隐藏的内存分配是在堆上进行的，并且不受程序所实现的类的管理。当然，程序员也可以为 vector<>写一个分配器，或是为了管理这些数组实现一个自己的 vector 类，但是这已经超出了本文的范围。现在，每进行一次分配，都必须先判断是否有空闲分块，如果没有，必须分配另一个内存池。

这个例子只是在应用程序的堆中分配这些内存池；下文会讨论怎样对这些分配进行优化。程序员还必须扩大这个指针数组，使它与刚刚分配的内存池中分块的数量相匹配。一旦完成了这些，程序员就要像前面一样，通过一个循环来为这些指针赋值。惟一不同的是，此时他们把这些新指针添加在向量的尾端，并且，当分配器析构时，必须释放这些分配的内存池。下面是更新了的分配器模板类。

```
template <class T, size_t blocksPerBatch>
class SimpleAllocator
{
public:
    SimpleAllocator() : nextAllocation(0) {}
    ~SimpleAllocator()
    {
        // Clean up the batches that we allocated
        size_t iNum = batches.size();
        for (size_t i=0; i<iNum; ++i)
        {
            byte* p = (byte*)batches[i];
            delete [] p;
        }
    }
    T* AllocateBlock()
    {
        if (nextAllocation >= pointers.size() )
        {
            // Allocate a new batch of blocks and pointers
            byte* pBatch =
                new byte[sizeof(T)* blocksPerBatch];
            batches.push_back(pBatch);

            pointers.resize(pointers.size()
```

```
                +iBlocksPerBatch);
            // fill or add the pointers for the new blocks
            size_t iNew = nextAllocation;
            for (size_t i=0; i< blocksPerBatch; ++i)
                pointers[iNew++] = &(pBatch[i]);
        }

        return pointers[nextAllocation++];
    }
    void ReleaseBlock(T* pBlock)
    {
        if (pBlock)
            pPointerArray[--nextAllocation] = pBlock;
    }

    typedef vector<byte*> BatchPtrVector;
    typedef vector<T*>    ObjectPtrVector;

private:
    BatchPtrVector  batches;
    ObjectPtrVector pointers;
    size_t          nextAllocation;
};
```

现在程序实现了一个分配器，在有空闲分块的情况下，它进行分配操作的复杂度是一个常数：只需要判断一次大小并且把索引递增 1。无论分配了多少内存池或是使用了多少内存分块，也无论次序如何，这个命题永远成立。这比应用程序堆好多了，应用程序堆会随着时间的推移逐渐变慢。同样的道理，释放一个分块也非常迅速，并且与分配了多少分块或是有多少空闲分块无关。

4.4.3　用户友好的分配器模板

要真正地把分配器整合到 C++中，应该使用标准的 new 和 delete 运算符，并且确保对象的构造函数和析构函数被调用了。这样一来，调用者不需要做任何特殊的工作就可以使用分配器的功能。要实现这点，游戏的程序编写人员需要为每个要使用这个分配模板的类重载 new 和 delete 运算符。以下为原型。

```
class T
{
public:
    // single allocation
    void* operator new(size_t s);
    void  operator delete(void* p);

    // array allocation
    void* operator new[](size_t s);
    void  operator delete[](void* p);
```

```
};
```

为了使用这个分配器，程序员只使用这些运算符的单实例形式。在这样一个简单的实现里，使用 new[]和 delete[]会引入一系列棘手的问题。因此本文在这里，并不准备讨论怎样追踪所分配分块的大小，以及在这些分块被释放时怎样获得它们的大小等问题。

为了让使用这个模板的类的所有对象都可以使用同一个内存池，把这个分配器作为一个静态成员。这需要一个外部的实例化，可以把它放在模板头文件中模板代码的下面。正如程序员想要得到的，每个使用这个模板的类都会实例化一个静态的分配器对象。

```
template <class T, size_t blocksPerBatch>
class BlockAllocator
{
   // new and delete operators here...

   struct BlockStore
   {
      // Our allocator implementation goes here...
   }
   static BlockStore s_Store;
};

// Instantiate the static allocator member for each type
template <class T, size_t blockSize>
   BlockAllocator<T, blocksPerBatch>::BlockStore
   BlockAllocator<T, blocksPerBatch>::s_Store;
```

现在，使用这个模板时，只需要从这个模板派生一个类，并且把这个类作为模板的第一个参数。然后，把最初要分配的分块数量作为第二个参数。通常，最好使用一个比整个程序生命期中所需要的这个对象的最大数量的一半或是 1/3 多一点的数字。当然，实际的数字需要通过常识、程序的行为和要分配的对象来决定。如果知道，某个特定类型总是有 100 个对象，那就可以在最初时，让这个分配器分配 100 个对象所需要的空间；但是如果这个数字在 50 到 100 之间，系统可能需要在最初时，分配 60 或 120 个对象所需要的空间。

对这个简单的分配器进行扩展

这里还有一个内存分配方面的问题上文没有涉及：字节对齐。在缺省情况下，大多数堆分配会和 DWORD（也就是 4 字节边界）对齐。这对那些需要同时工作在 8 个甚至 16 个字节之上的，32 位机器的性能影响很大。在目前的电脑上，通常所有的分配都应该至少和 8 个字节边界对齐，以获得最佳性能。而在某些情况下，为了针对缓存线（cache line）和处理器中的向量单元（对于 PC 上的 SIMD 和 Mac 上的 AltiVec 这样的处理器来说，数据在 16 位或更高位对齐时工作可以获得最佳性能）进行优化，分配与 16 位或 32 位对齐可能会更好，即使这么做会浪费很多字节。可以使用下面的代码来使任意指针向下一个边界值对齐。

```
size_t alignment = 16; // 1, 2, 4, 8, 16, etc.
T* pAligned =
   (T*)(pOriginal + (alignment-1))&(~(alignment-1));
```

首先，它在最初的指针上加上所需要对齐的字节减 1，以此确保这个值不小于最终值。接着和对齐值减 1 的反使用一个逻辑与操作。这样做的效果是把低位都屏蔽了，使得指针对齐。

同样的技术可以用来把类的大小对齐到某个特定的值。

```
size_t aligned =
       (sizeof(T)+(alignment-1))&(~(alignment-1));
```

上面的代码在分配对齐的对象时都非常有用。新的 AllocateBlock()方法会使用上述两段代码来把所有内存池对齐到 16 字节，并且把所有分配了的分块按照 blockAlignment（分块对齐）的值进行对齐。

```
T* AllocateBlock()
{
   if (nextAllocation >= pointers.size() )
   {
      // Align the block size to blockAlignment
      static const size_t blockSize =
         (sizeof(T)+alignment-1)&(~alignment-1));

      // Allocate a new batch of blocks and pointers
      byte* pBatch =
         new byte[blockSize*blocksPerBatch+15];
      batches.push_back(pBatch);

      pointers.resize(pointers.size()+blocksPerBatch);

      // Align the new batch on a 16-byte boundary
      byte* pAligned = (byte*)(pBatch+(16-1))&(~(16-1));

      // fill or add the pointers for the new blocks
      size_t iNew = nextAllocation;
      for (size_t i=0; i< blocksPerBatch; ++i)
         pointers[iNew++] = (T*)&(pBatch[i]);
   }

   return pointers[nextAllocation++];
}
```

把 blockSize（分块大小）作为一个静态常量，就可以让预处理器对它进行计算，并且把结果直接使用在代码里。这么做可以让代码更快。此外，可以把 alignment 的值作为模板参数来传递。这可以让每个实例化的类都可以指定自己的对齐方式。

4.4.4　降低分配器的内存开销

操作系统或是文件系统这样的页面分配器（page allocator）通常会使用一个技术：把管

理分配的数据结构放在未分配的分块中。借用这个技术，游戏的程序编写员不再需要分块内存指针数组，而只需要在每一个空闲块开始的几个字节里，保存指向下一个空闲块的指针。

每当一个新的内存池被分配时，不再像上面那样，创建一个指向每个内存块的指针数组，而是把这块新的存储区域当作一个链表来进行循环。每一个分块的前 4 个字节现在包含了指向下一个分块起始地址的指针。当想要分配一个分块时，返回链表中的第一个分块，并且保存指向下一个分块的指针。每当一个分块被释放时，就把当前的下一个指针放在被释放分块的前面，并且设置下一个指针指向被释放的分块。以下是新的分配方式。

```
T* AllocateBlock()
{
    //* Is there any room?
    if (!ppNextBlock || !*ppNextBlock)
    {
        // determine the aligned size of the blocks
        static const size_t blockSize =
        (sizeof(T)+blockAlignment-1)&(~(blockAlignment-1));
        // make a new batch
        byte *pBatch = new byte[blocksPerBatch*blockSize+15];
        batches.push_back(pBatch);
        //* Align the block on a 16-byte boundary
        byte* pAligned = (byte*)((uint)(pBatch+15)&(~15));

        // fill the pointers with the new blocks
        ppNextBlock = (byte**)pAligned;
        for (int i=0; i<blocksPerBatch-1; ++i)
        {
            *((uint*)(pAligned + i*blockSize)) =
                        (uint)(pAligned + (i+1)*blockSize);
        }
        *((uint*)(pAligned+(blocksPerBatch-1)*blockSize)) =
                (uint)0;
    }
    byte* pBlock = (byte*)ppNextBlock;
    ppNextBlock = (byte**)*ppNextBlock;
    return (T*)pBlock;
}

void ReleaseBlock(T* pBlock)
{
    if (pBlock)
    {
        *((uint*)pBlock) = (uint)ppNextBlock;
        ppNextBlock = (byte**)((byte*)pBlock);
    }
}
```

4.4.5　总结

很多方法可以对这个想法进行扩展。如前所述，使用一个低层分配器就可以一次性地从堆中分配一大块内存。然后，当一个分块分配器需要一个新的分块池时，它可以使用已分配的这一大块内存的一部分。这可以大大地减少当分配器需要增长时，分配新的内存块所需的时间，但是它也可能会带来一些副作用：它会导致大块的系统内存空闲。

还可以加入针对数组的 new 和 delete 实现，以提高这些分配操作的速度和效率。此时最耗时的操作是要找到一段连续的空闲分块。程序员会多分配一个分块来保存所分配分块的数量，以及用来描述这次数组分配的其他相关数据。当释放这个数组分块时，就可以使用保存在这个额外分块中的信息来把分块放回空闲列表。

要勇于进行实验。在分配分块时，使用前面所讨论的基于 vector 的分配器有可能比使用缺省的堆分配器要快几百倍。使用一个性能分析器可以发现代码中的问题区域。在关键代码路径上进行大量的分配总不是很理想，但是对于那些无法避免的情况来说，使用一个专门的分配器或许会对游戏的开发和运行有所帮助。

4.5 使用贴图定制三维角色

Todd Hayes, NCsoft
thayes@ncaustin.com

由于 MMP 游戏本质上具有社会性，玩家试图在这些游戏中模仿真实生活中的社会行为也就不足为奇。这方面最有力的例子莫过于玩家对角色个性化的渴望，他们希望自己的角色可以通过某些方式与其他人的区别开来。要满足他们的这个需求，最好的办法就是为玩家提供一个可以定制角色外观的机制。从玩家的角度来看，可供定制的选项和变化越多越好；但从开发人员的角度来看，更多的选择意味着需要更多的美术资源、更多的测试以及更长的整体开发时间。不仅如此，所有这些额外的资源都可能导致性能问题，因为游戏中可能随时需要处理更多的网格和贴图。这里最大的挑战就是要寻找这样一种方法：它不仅具有玩家可以接受的定制程度，而且对开发和性能的影响也很小。

本文所提供的角色定制机制可以在对性能的影响和对美工资源的需求进行有效控制的同时，为玩家提供较高层次的定制。

4.5.1 角色定制的类型

角色的外观由两个部分组成：贴图和几何信息（后者也被称为网格（mesh））。程序员可以通过对角色的几何信息和/或贴图进行修改来定制角色。这些方法都有各自的优缺点，因此程序员必须根据游戏引擎来考虑所有的问题，从而为项目选择最合适的方法。

1. 使用网格进行定制

对角色网格进行修改有一个明显的好处：它可以改变角色的轮廓。通常可以通过缩放网格，或是对身体部分的几何信息进行混合和匹配来创建最后的角色网格，但是在对网格进行缩放时必须非常小心。虽然可以简单地使用缩放矩阵来改变角色的大小，但是仅当对网格的缩放在各个坐标轴上一致时，这种做法才会有效。譬如说，如果为了获得一个高大的角色，把一个不一致的缩放矩阵应用在一个符合比例的标准网格上，这时副作用就会很明显。这个角色移动时手臂的长度会改变，当它们在角色的侧面垂下时会最长，而在水平向外伸出时会（譬如说出拳）最短。要想正确使用不一致的缩放矩阵，则需要对网格的不同部分在合适的方向上进行缩放，并且把修改后的网格作为角色的网格使用。即使这个问题得到了解决，游

戏中还可能会有很多其他问题影响网格的外观，而且，动画系统也需要能够处理网格的新尺寸。不仅如此，无论怎样进行缩放，都需要对网格的法向量进行正确的转换，以避免任何由于长度变化，或是不一致的扭曲引起的光照或碰撞问题。

分段网格交换（Piecewise mesh-swapping）技术使用较小的部分或是多个网格片来为角色创建网格。譬如说，在一个幻想游戏中，美工人员可能想要创建一个完整的网格来表示一个穿着皮甲的角色，并且创建另一个网格来表示穿着金属盔甲的角色。他们可以利用分段网格交换技术来选用这些网格的组成部分，以生成一个角色网格，这个网格中盔甲的某些部分是皮的而另一些部分是金属的。此外，这些片断可以作为完全分离的网格而不是被当作一个大网格的一部分进行保存，他们还可以使用同样的技术对它们进行处理。这个方法避免了很多网格缩放带来的问题，并且带来了更加生动的轮廓变化。然而，这些额外的好处需要付出沉重的代价。虽然只需要对单一的几何片段简单地施加一组转换就可以获得最终的网格，但是一个网格交换系统需要美工为每一个可以交换的片断都生成新的美术资源。这样就不仅需要创建更多的美工资源，还要在游戏运行时对更多独立的网格和贴图进行处理和渲染。更为重要的是，网格交换方法非常依赖于底层的游戏引擎。如果游戏引擎中的角色是由分块的网格组成的，那么很可能需要一种机制，以使在游戏中交换这些分块变得更为方便。然而，如果引擎使用皮肤和变形网格（sknined or deforming mesh），那么网格交换技术的使用将会变得非常复杂。它需要把基于皮肤的几何信息和角色骨架中的骨骼正确地联系起来。如果在这样的系统中创建美工资源时没有遵守严格的规范，可能需要把这些网格片断缝合起来才能避免当角色网格变形时出现裂缝。这个缝合过程无论对于开发过程还是运行过程来说，都是一个不小的代价。为了能够正确地进行缝合，必须让这些美工资源遵守一定的规范和限制。对于美工人员来说，这些规则通常难以理解，更不用说让他们在使用美工工具时还要遵守这些规则。质量保证（QA）也变得更为困难，因为在缝合过程中，任意网格的组合都可能导致视觉上的异常，从而必须对它们进行测试。

2. 使用贴图进行定制

虽然不能像网格交换那样改变轮廓，使用贴图进行定制可以在付出一定的几何信息和贴图开销后，为角色提供更广泛的外观选择。美工在创建角色网格时必须同时为其进行贴图映射，因此在那时，美工就已经知道了该网格所要用到的贴图的数量和类型。无论游戏设计人员在运行时对贴图进行多大的修改，贴图的真实大小和内存用量都不会改变。因为可以非常直观地对贴图的大小作出限制，所以无论开发人员在什么层次上进行定制，都可以把角色的开销控制为一个固定值。这个方法并不会对网格进行任何修改，它对渲染引擎的惟一要求就是能够修改内存中的贴图。它还使得同样的网格可以同时被多个角色共享，这进一步减少了内存的占用。通过使用 Alpha 混合和其他技术，美工人员可以把多个贴图或是贴图上的区域合成（composite）在一起，从而为角色生成几乎是独一无二的贴图。譬如说，美工人员可以把表示行会标志或联盟标志的小贴图放在角色贴图的任意位置上来提供代表角色们组织的纹身。还可以使用小贴图来创建眼睛颜色、头发颜色以及面部特征，从而提供各种外观和风格的变化。

就上文所讨论的两种定制技术而言，使用贴图进行定制的优势在于它可以提供大量的变化；无论将来会采用什么定制选项，所需资源都比较少；并且对引擎功能的依赖也较低。

因此，本文其余的部分将着重对基于贴图进行定制的技术和实现方法进行更加细致的讨论。这些讨论将避免一些特定的实现细节，因为那些细节依赖于底层系统所采用的引擎技术。下文会着重对相关的高层概念进行讨论，并指出那些底层引擎必须满足的基本技术要求。

4.5.2 贴图合成简介

如果要使用贴图进行定制，系统必须支持一些基本的功能。贴图定制系统所基于的底层引擎必须能够创建和修改，用于在渲染时映射到角色上的贴图。它还必须提供某些方法，来把源贴图全部或部分地渲染或拷贝到合成的角色贴图中去。至少，它要使美工人员能够使用在合成中用到的任何贴图的像素数据。这样才可以编写代码，用源贴图中的任何区域来完全替代目标贴图的任何区域。怎样定义和构造一个可以用于贴图合成系统中的贴图区域呢？这取决于所使用的贴图定制系统本身（这个定义所包含的问题将在后面讨论）。虽然系统只要满足上述要求就可以工作了，但是为了获得这一系统全部的优点和灵活性，还需要在进行合成操作时支持一些额外的操作，尤其是对调色（hueing）和 Alpha 混合的支持。

调色功能使得玩家可以为合成操作指定一个颜色。每一个像素在画到合成贴图中之前都会按照这个颜色进行调节（modulate）。这时玩家可以使用任意颜色来为同一个美工资源创建不同的贴图。譬如说，美工人员可以创建一个灰度模式（grayscale）的衬衫贴图。通过在合成时，而不是在创建时为衬衫指定颜色，这个贴图可以创建出超过 1 千 6 百万种不同颜色的衬衫。

在源贴图的某个区域和目标贴图的某个区域进行混合时，如果可以使用源贴图中的 Alpha 通道，就可以得到很多更高级的效果。如果系统支持 Alpha 混合，就可以把源贴图中的任意形状渲染到合成贴图中去，这只需要把那些不打算画到合成贴图中去的像素的 alpha 值设为 0（而如果不支持 alpha 混合，就只能拷贝实心的矩形区域了）。此外，它们不必完全替代目标贴图中的相应区域。如果源贴图中的某个像素具有一个透明的非零 alpha 值，它将不会取代合成贴图中的像素，而是与它合成。这种效果在进行纹身和人体彩绘时尤为突出。在进行纹身和人体彩绘时，首先把一张皮肤贴图渲染到合成贴图中去。假设把源贴图中所有表示纹身的像素的 alpha 值设为 128。当把纹身贴图混合到合成贴图中去时，它会为已经存在的皮肤贴图着色，而不会影响到肌肉和其他的身体组织。

图 4-14 一次合成过程中所使用的贴图

图 4-14 介绍了把纹身画到角色手臂上的合成过程中所使用的贴图，它分为以下 4 个部分。

1．所使用的纹身贴图。

2．纹身贴图的 alpha 通道（注意这是纹身贴图的一部分，这里分开显示只是为了清晰。这个例子中使用的 alpha 值为 195，这使得在它下面的贴图可以部分地显示出来）。

3．皮肤贴图中表示角色肩部的部分。

4．纹身和皮肤混合之后生成的合成贴图。

把这两个功能组合起来使用我们就可以获得各种令人称奇的效果，譬如说透视装、有色的纹身、污垢以及伤痕等。另一个使用 alpha 混合的功能（前面的例子中已经隐约提到了）是分层（layering）。

4.5.3　分层

在只支持从源贴图到目标贴图进行不透明拷贝的系统中，合成步骤通常非常简单。对于目标贴图的指定区域来说，只有一个源贴图会与之对应。只需要对每个区域进行循环，并且用相应的源贴图进行渲染就可以完成合成过程。加入了 alpha 混合以后，合成过程的管理会变得更为复杂，这是因为合成贴图的某个区域可能会受到多个源贴图的影响。这时就需要使用分层技术。层次是把源贴图与它在组合贴图中要占据的一组区域关联起来的一种途径。层次也可以包含在合成过程中用到的额外参数，譬如说颜色是否透明。

为不同的衣着概念层次赋予不同的优先级，可以使层次间的关系更为清晰，在大多数情况下这都非常有用。举一个简单的例子，可以把“皮肤”的优先级设为 0，“衣服”的优先级设为 1，“盔甲”的优先级设为 2。在合成时，需要先合成皮肤，接着是衣服，最后才是盔甲。它们所包含的 alpha 通道使得每一层都可以在重叠时从它们上面的层中露出来。值得注意的是可以为使用不同贴图的多个层次赋予相同的优先级，这在某些情况下是非常有用的。譬如说，游戏设计人员希望把衬衫和短裤放在不同的贴图中，但是让它们都属于“衣服”这一类。只要这两个层次没有重叠的区域，一切都会正确地合成。但是如果它们重叠了，最终的结果将是未定义的。表 4-1 中是一些层次和它们对应的优先级。

表 4-1　　层次及其对应的优先级

优先级	层次	常见内容
0	皮肤	角色基本皮肤
1	身体艺术	纹身、人体彩绘、伤痕
2	毛发	肌肉、胡子、鬓角
3	衣服	衬衫、短裤、鞋子、手套
4	外衣	盔甲、背心、外套、首饰
5	徽章	武器盾徽、行会徽章

游戏设计人员必须能够通过层次来确定它们所使用的贴图区域，因此需要有一种机制来对这些区域的定义进行有效的管理。为了在对这些区域进行处理的同时避免给层次本身带来

不必要的复杂度，本文引入了贴图模版和样本的概念。

4.5.4 贴图定制模版和样本

要实现一个贴图定制系统，最关键的步骤就是要确定最终贴图是怎样映射到角色网格上去的。贴图到网格的正确映射是贴图定制成败的关键。虽然大多数美工工具都能够做到让美工人员把贴图的任意部分映射到角色网格的任意点上，定制系统仍然需要以一个可靠且合乎逻辑的方式把贴图的某个区域拷贝到最终的合成贴图中。要做到这点，游戏设计人员需要为贴图定义一个与角色相应部位对应的区域。譬如说，为了可以在合成贴图中使用一张贴图中表示脸部的部分，源贴图和目标贴图必须都有一个表示脸部贴图的区域，这样系统就知道应该使用源贴图中的哪个部分，以及应该把它放在合成贴图的什么地方。定义这些区域的结构被称为贴图模版（texture template），这个结构定义的区域被称为贴图样本（texture swatches）。一旦为一个角色定义了贴图模版，那么美工人员就可以使用任何遵循这个布局的贴图，来创建合成贴图。图 4-15 中是一个贴图模版示例。

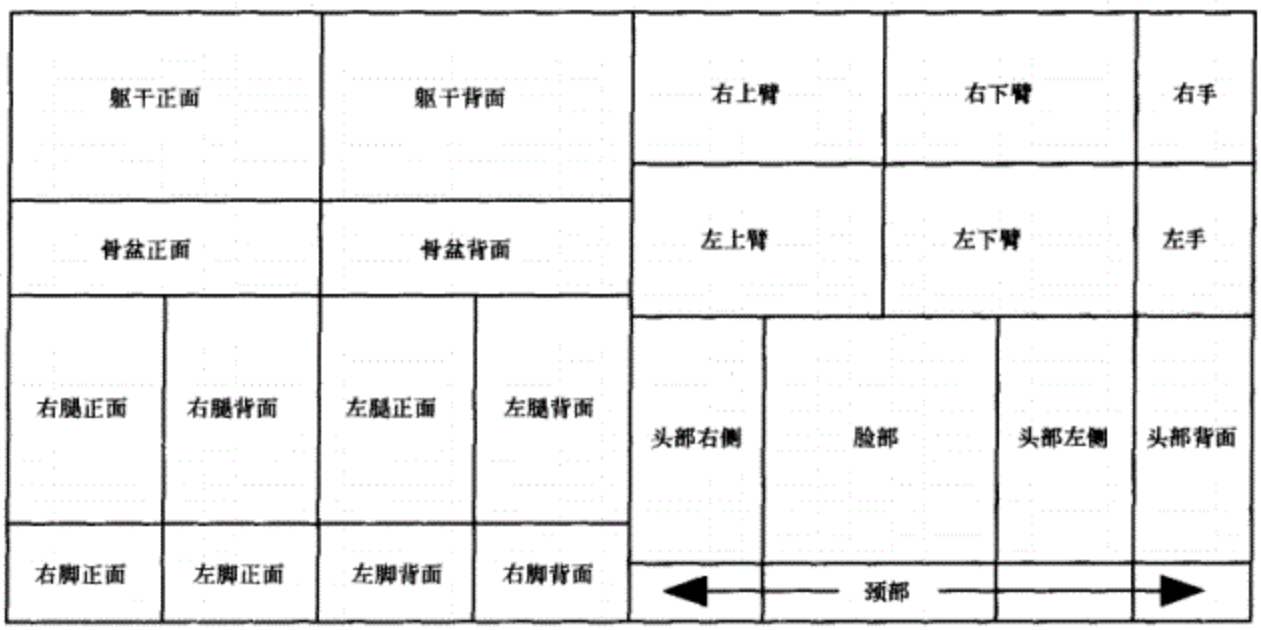


图 4-15 使用常见贴图样本的贴图模版布局示例

贴图模版的惟一用途就是定义贴图样本。很多方式都可以定义这样一个模版，但是本文所介绍的方法可以让美工人员创建某种具有特殊结构的贴图，通过载入并分析贴图，从而找到其内部的样本。因为这个模版贴图除了定义样本以外没有任何其他用处，可以在里面使用一些特定的保留颜色来定义样本区域。要做到这点，最好的方法莫过于使贴图既可以被人类读懂又可以被代码解析。首先可以定义一对易于辨识的颜色作为“边界颜色（border colors）”。譬如说，可以使用紫色（R:255，B:255，G:0）作为边界上的“边缘”颜色，使用绿色（R:0，B:0，G:255）作为边界的“起点”（也就是左上角的颜色）。这样，每个区域边界左上角的颜色是绿色，并且区域的边界是紫色。这样一来，贴图分析器就可以在贴图中搜索，每当遇到

一个绿色的像素时，它就开始对整个边缘进行处理。接着它会移动到右面的像素。通过搜索贴图中在这个像素右面的紫色像素，分析器可以找到这个区域右面的边缘。同样，通过向下搜索，它可以找到下面的边缘。一旦找到了所有的边缘，这些边界及其内部的像素就是这个样本所覆盖的区域。

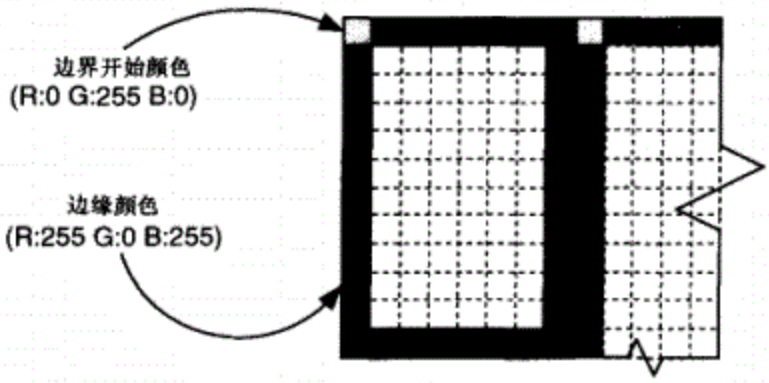


图 4-16　边缘定义示例：8×12 像素大小的样本

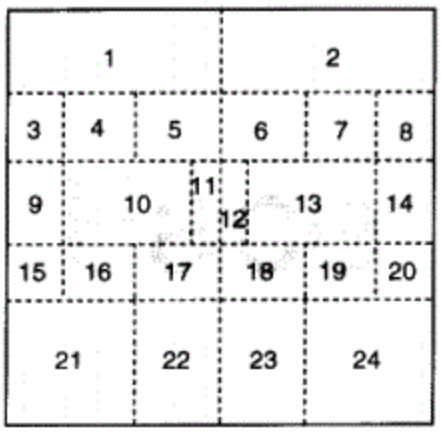
这些样本本身是由它们在贴图模版中包围的矩形区域定义的。游戏程序通过读取这些样本来获得它们所标示的矩形区域的信息。贴图模版中的每一个样本都会被赋予一个在模版中独一无二的标识。一旦定义了样本，层次只需要包含一个样本标识的列表就可以了。游戏的美工人员可以用这些标识来把样本合成到目标贴图中去。

因为模版定义了合成贴图的粒度，所以必须根据要使用它的游戏的特定需要，对其进行定制。如果一个游戏只需要一定层次的定制，譬如说为角色选择面部、衬衫和短裤贴图，那么只需要定义几个大的样本就行了。如果游戏像图 4-14 中那样，允许玩家在指定的区域放置纹身，就需要在模版中提供大量小尺寸的区域，以使纹身可以在任何区域中使用。后面这种方法的问题在于，为了达到更大的灵活性，游戏设计人员会把比较大的抽象区域分解为很多难以管理的小型样本。可能要把角色的胸部分为左上、中上和右上等区域以便放置纹身，这样一来，当美工人员要合成一个衬衫贴图时，必须指定所有这些小型样本，而不是指定一个胸部样本。下面所要介绍的样本集合可以帮助游戏开发人员对这些样本的逻辑集合进行更有效的管理。

4.5.5　样本集合

虽然样本集合（swatch groups）并不会影响底层的贴图定制系统，但是它们可以提供一个抽象层来简化用户与贴图定制系统之间的交互。通过使用样本集合，角色的贴图模版可以被细分为大量的样本。这使得工作人员可以根据合成过程的需要使用任意大小的粒度，并且不会因为要在合成中指定会使用哪些样本而增加额外的工作量。在处理面部区域模版的例子中，可以把整个区域进一步细分为眼睛、鼻子、嘴巴、纹身、伤痕以及其他面部特性，这样就可以达到更细的粒度。可以创建一个名为“面部集合”的样本集合，它包含了这些小型样本的标识。使用样本集合不仅可以把用户和模版的细节隔离开来，还可以确保在对模版进行修改或细分时不会影响用户已经作出的关联。在面部集合的例子中，用户可以指定单一的参

数来把一个层次应用于整个面部；如果他们只想把这个层次应用到一部分的面部上，那他们也可以任意指定多个更为细致的样本。图 4-17 所示的面部样本集合对原始面部区域进行了细分，从而使游戏设计人员能够在不影响使用的情况下进行更加细致的定制。



1. 右额
2. 左额
3. 右太阳穴外侧
4. 右太阳穴内侧
5. 右眼
6. 左眼
7. 左太阳穴内侧
8. 左太阳穴外侧
9. 鬓角右部
10. 脸颊右部
11. 鼻子右部
12. 鼻子左部
13. 脸颊左部
14. 鬓角左部
15. 鬓角右下部
16. 脸颊右下部
17. 嘴的右部
18. 嘴的左部
19. 脸颊左下部
20. 鬓角左下部
21. 右颌
22. 右颊
23. 左颊
24. 左颌

图 4-17 面部样本集合

4.5.6 总体实现

本文对贴图定制系统的各个部分都已经进行了讨论，下面本文将介绍怎样把这些部分整合在一起。下面的伪码是这个系统的一个实现示例。需要定义的第一个类是样本。

```
class Swatch
{
    float left;
    float top;
    float right;
    float bottom;
}
```

保存在样本中的值可以是像素坐标或者贴图坐标，它们可以按照最适合于游戏的方式进行存储，譬如说存储左上角/右下角或是原点/宽/高。对于样本来说，最关键的就是它为源贴图和目标贴图同时定义了一个矩形区域。

给定模版的所有样本都可以保存在这个模版类中并由其进行管理。这个例子是一个经过简化的实现，它只保存了样本的 vector，这样它就可以通过下标来获得样本。在一个真实实现中，模版类或许还会为生成它的贴图保留一些额外的信息，并且还会提供函数，这样，就可以通过样本标识来获得指定的贴图区域。这些功能与合成引擎对贴图坐标的具体要求相关，因此本文的例子中没有包含它们。

```
class Template
{
    vector<Swatch> swatches;
}
```

接下来要对层次进行定义，这是一个比较大的系统类，它将会使用到样本和模版。

```
class Layer
{
    int priority;
    int opacity;
    color hue;
    Texture *pSourceTexture;
    list<int> swatchIDs;
}
```

优先级的分配和特定的应用程序有关。这里假设优先级为 0 表示的是最底层，优先级高的层次总是画在优先级低的层次之上。透明度通常是一个从 0 到 255 之间的数字，0 表示完全透明，255 表示完全不透明。在某些底层引擎中，可以把透明度嵌入到颜色成员的 alpha 通道中去。每一层还必须包含一个引用，它指向在合成过程中会用到的贴图；同时，每一层还必须包含一个样本列表，这个列表保存了那些需要合成进去的样本标识。

最后要为这个系统定义一个用来进行实际工作的类。角色贴图（CharacterTexture）对象需要保存最终的贴图，以及合成这张贴图所要用到的所有层次。它还需要能访问定义贴图布局的模版。角色贴图类实际上执行了贴图定制系统中所有的工作，因此它不仅包括了数据成员，还包含了一些方法。

```
class CharacterTexture
{
    list<Layer> layers;
    Texture *pCompositeTexture;
    Template *pTemplate

    AddLayer(Layer *);
    RenderTexture();
}
```

AddLayer()（添加层次）方法可以向系统添加一个新的层次。它必须确保列表中的层次是按照优先级排序的，这样它就可以在合成时，对层次列表进行枚举，并且把每一层依次合成到最终贴图中去。在实际系统中，还必须实现它的对等方法 RemoveLayer()（删除层次）以及一些其他的辅助函数，但是为了表述更清楚，本文在这里把它们省略了。

一旦加入了所有的层次，就可以调用 RenderTexture()（渲染贴图）方法来进行合成工作。

```
void RenderTexture()
{
    // Clear out the destination texture to
    // prepare it for a new compositing
    // operation.
    ClearComposite();

    // Iterate through each layer and composite it
```

```
        // into the final texture
        for each layer in layers
        {
            // To composite a layer, each swatch must
            // be used to render its region into the
            // final texture
            for each id in layer.swatchIDs
            {
                CompositeRegion( pTemplate->swatches[id],
                                 layer.pSourceTexture,
                                 pCompositeTexture,
                                 layer.opacity,
                                 layer.hue);
            }
        }
    }
```

RenderTexture()会按照从后到前的顺序根据所有的 alpha 和颜色属性把每一层都合成到最终的角色贴图上去。

4.5.7 对定制系统的进一步改进

很多方法可以对本文所介绍的基本系统进行改进和优化。大多数优化或是改进方法都非常依赖于底层的渲染引擎。如果使用软件来实现角色贴图类的 CompositeRegion()（合成区域）方法，就可以为不同类型的区域（譬如说不透明的或是没有调色的）度身定制不同的优化版本。此外，该方法还可以利用硬件渲染来进行合成，这样做可以大大地减少使用 alpha 混合和色彩所要消耗的资源。示例系统中在整个合成过程中只使用了一个模版。在一个更为灵活的系统中，美工人员可以为每个层次和最终贴图提供不同的模版。这样的系统使得他们可以把较小的贴图合成到较大的贴图中去。譬如说，如果一个游戏只支持几种皮肤色调，但是却支持很多种面部特征。那么，在只支持单一模版的系统中，美工人员就必须让面部特征贴图和整个人体贴图的大小保持一致，这么做非常浪费贴图空间。但如果合成系统能够通过一个只包含面部特征样本的较小模版，把样本合成到基于大型模版的最终贴图上，就可以节约不少贴图空间。

一个支持角色多重贴图的系统可以为上面的示例加入更多的功能。要实现这样一个系统，就不能只为 CharacterTexture 类提供一张贴图，而必须提供一个表示复合贴图的贴图列表。复合贴图中的每张贴图都可以用来实现一个特定的多重贴图效果，譬如发光（glow）贴图或者凹凸（bump）贴图。同样地，每个层次也可以把多个源贴图合成到最终贴图中去。

为了避免对所有样本进行渲染，设计人员可以为样本或层次提供一个标志，来表示它是否透明。在合成过程中，如果一个层次中的某个样本被更高层次中的样本覆盖，而后者又不是透明的，就不必渲染它了。使用这样的标志可以减少合成过程中的资源消耗。

4.5.8　系统的局限性

贴图定制系统与现代渲染引擎中一个主要功能——mip-mapping 不能很好地并存。整个过程使用的最终贴图实际上是由不同贴图中的矩形区域自下而上合成而来的。而贴图中，彼此相邻的区域在映射到角色网格上以后却不一定是相邻的。当工作人员为贴图创建 mip-map 时，需要对相邻的像素取平均值。对位于贴图区域边缘的像素这么做，就会导致与之对应的网格区域所使用的贴图数据来自于不相邻的网格区域。在个别情况下，这看上去就像是把贴图放在搅拌机中搅拌后再贴到角色身上一样。游戏设计人员应该尽量避免为这些合成贴图生成 mip-map。如果这么做导致贴图“闪烁”或是产生了其他的副作用，就应该去研究一下那些用于生成 mip-map 的非传统技术。在生成每一个后继的 mip-map 时，设计人员可以把当前的贴图和某个颜色（譬如说灰色）进行混合，这样做可以更为自然地表示远处淡出的细节。甚至可以为 mip-map 的每一层引入一个 alpha 通道，从而使得角色可以在远处淡出。这些方法的结果会随着游戏需求的不同而不同，所以只有进行一些实验才能够获得可行的方案。

如果试图在系统中使用具有层次细节（Level-of-Detail ，LOD）的网格，也会遇到同样的问题。因为贴图映射是不连续的，开发人员可能必须对网格上的某些顶点进行复制，以使贴图上不相邻的区域能够映射到网格中相邻的三角形上。如果在降低网格 LOD 时，删除了这些成对顶点中的某一个，那么整个网格上就会有一些区域的贴图由于受到拉伸而不正确。但如果这个 LOD 网格是通过算法生成的，那么通过不删除任何成对的顶点就可以避免这个问题。这样做可以避免删除接缝处的顶点，从而保证这些不连续区域具有正确的贴图映射。反之，如果这些具有层次细节的网格是由手工修改的，那么通常就可以为新生成的网格找到一个更合适的贴图映射。

游戏设计人员可以通过合理地定义贴图模版与网格之间的关系来降低这两个副作用。如果尽可能地让贴图上的连续区域也映射到网格上的连续区域，那么在渲染时出现的瑕疵就会少很多。想象一下把一块正方形的橡胶布经过拉伸以后裹在网格上，然后在这张橡胶布上画上我们的贴图，最后再让这张橡胶布恢复原先的形状。但如果贴图模版的布局就像这张橡胶布一样，那么最后渲染时的瑕疵将会非常少。反之，如果这个贴图模板被分为很多小块并且任意地贴在网格上互不连续的位置，那么在渲染时将会看到明显的瑕疵。

4.5.9　总结

使用贴图合成机制来进行角色定制具有非常广泛的用途，游戏开发人员可以方便地把它整合到大多数引擎中。它不仅让游戏开发人员能够精确地判断角色定制所带来的资源消耗，而且可以在只需要少量美工资源的情况下，为模型生成大量不同的外观，这对于那些资源有限的平台来说是一个非常有吸引力的选择。由于系统只依赖于底层渲染引擎中的一小部分功能，因此在开发初期就可以对其进行实现，而不需要等到引擎中那些更为先进的功能被开发出来以后。不仅如此，在开发初期只需要简单地实现它就可以了，在随后的开发过程中可以不停地对其进行改进，更不用说它还可以和其他定制方式共同使用。因此，在大型多人在线游戏的整个生命周期中都可以使用贴图合成机制，无论是在早期的原型化阶段，还是最终将其作为一个完整而长期的角色定制方案。

4.6 游戏机平台上 MMP 游戏的独特挑战

John M. Olsen, Microsoft Corporation
infix@xmission.com

为什么会有人想要在游戏机平台上创建 MMP 游戏呢？大多数原因就和游戏开发人员选择在游戏机平台上而不是在其他平台（譬如 PC）上进行某个游戏的开发工作一样。游戏机平台最大的好处在于一致的硬件，这极大地简化了针对不同配置所要做的测试工作。

此外，因特网宽带接入越来越普及了，在可预见的将来，这一趋势还会继续。最新的游戏机平台都内置了因特网连接，或是把它作为可选附件提供，这就创建了一个特定的用户群。与一般的游戏用户相比，对他们进行描述、跟踪和营销在很多方面都会变得更为简单。

基于游戏机平台的 MMP 市场还处于发展初期，如果一个厂商能够在第一时间进入市场，并且能够对游戏机平台的特有问题进行正确处理，从而生产出高质量的游戏，它就有机会去影响这个市场的形成和成长过程。这其中最棘手的问题在于，怎样成功地处理基于游戏机平台的游戏所独有的问题。

4.6.1 环境

游戏机平台和 PC 的第一个区别在于放置硬件的地点。一个简单的事实就是游戏机通常放在起居室（或者说客厅）里，而 PC 放在书房里，这使得游戏机更多地处于一个共享的环境中——人们聚集和社交的地方。这一点可以通过很多方式在设计中体现出来。

分屏模式是一项很多基于游戏机平台的游戏经常要用到的技术。正如在更为典型的单机游戏中一样，在 MMP 游戏中使用分屏模式同样是有效的。让 2 到 4 个人在同一台游戏机上共同游戏可以降低很多问题的复杂度，包括交流、分组和交易。

游戏机环境的另一个要求是，必须减少单次游戏的持续时间。很多 MMP 游戏可能需要花费半个小时来进行连接；寻找朋友；决定要做什么，然后到达目的地。在游戏机上，游戏设计人员必须加快这一过程以使玩家在半小时内足以进行一次有效的游戏。

4.6.2　登录

商业性的 MMP 游戏要求用户进行登录，从而确保只有付费用户才能进入游戏。由于任何现有的游戏机平台都不把键盘作为一个标准配置，因此必须在保证安全的前提下尽可能地简化登录过程。

最简单的方法之一是，记住用户名，从而使玩家不必在每次连接时都要输入用户名。一旦通过屏幕上的虚拟键盘（或是别的文本输入方法）输入一个用户名，它就应该被保存到游戏机的某种持久化存储器中去，譬如说存储卡或者硬盘。

玩家可以通过一系列的按键动作来输入密码，或是采用和输入用户名相同的方式，通常是使用虚拟键盘。游戏机用户对这两种方式都很熟悉，因此开发人员可以根据游戏设计来决定使用哪种方式。

4.6.3　分辨率

可用分辨率的范围是游戏机和 PC 的主要区别之一。PC 游戏可以利用一些很高的分辨率，并且通常能够在游戏运行时切换分辨率。对于大多数 PC 游戏来说，屏幕分辨率通常是由用户来控制的。

在游戏机上，帧缓冲通常被设定在 NTSC 或是 PAL 分辨率。有些游戏机平台已经开始支持高分辨率的 HDTV 模式来打破分辨率造成的障碍，但是在设计时必须考虑最低标准。对于当前这代游戏机来说，在 NTSC 模式下，分辨率是 640×480，在 PAL 模式下，分辨率为 640×512。而手持设备的分辨率则会受到更大的限制。

在这么低的分辨率下，在决定怎样向用户显示文本时必须非常小心。游戏设计人员需要精心设计字体和图标以确保它们即使在模糊的电视机上也能正常显示。另外，屏幕一次可以显示的文本量也是有限的。测试人员要确保能够在办公室里面使用性能最差的电视机来进行测试，虽然这可能会让测试人员吃惊。但如果他们看不清某些内容，我们就要对游戏进行修改。

使用分屏模式只会使分辨率问题更加严重。使用两路或者 4 路的分屏模式会对在屏幕上显示文本甚至是图标信息的能力带来更为苛刻的限制。通过对一些共享元素（譬如说雷达地图和小组名单）进行组合，这个问题可以在一定程度上得到减轻，因为它们仅仅显示一次而不必为每个玩家都显示一次。

4.6.4　聊天频道

PC 机上 MMP 游戏的一个主要功能是玩家间的聊天。聊天不仅是帮助玩家形成社会感的主要手段之一，还是一个用来对各种小组和事件进行组织的简单方法。聊天可以分成几类，每一类满足一个特定的需要。

- 和朋友一对一的个别聊天。
- 和少量其他玩家一起组织分组和游戏。

- 和大量的朋友或是在行会中聊天。
- 大型事件中关于战略或是战术的聊天。

那么应该怎样处理游戏机上的这些需求呢？现在市场上任何一款游戏机的基本配置中都不包含键盘，而大多数基于 PC 的 MMP 游戏正是用键盘来进行聊天的。

语音是一个可行的解决方案，但是这需要进行大量细致的思考和计划。要想能够对上述所有需求进行处理，就需要对语音输入进行切换，以使它仅被发送到想要的通道，这样玩家就可以只和他所期望的其他玩家进行通话。这意味着游戏系统需要一个简单而迅速的切换机制来把语音映射到合适的通道。

随着更多的人加入一个聊天通道，语音通讯会变得很成问题。这就好像有很多人在一个房间里，如果大家都同时说话，就很难听清楚。虽然文本消息具有相同的问题，但是相对来说要好得多，因为在使用文本信息时我们还能够向前翻页。在使用语音时，任何丢失的消息就永远丢失了。

使用语音时需要考虑的另一个问题是系统要能够独立地对每个接收通道进行过滤，从而使玩家只听到对此刻有用的聊天内容。多个语音通道还会导致更大的带宽需求。降低服务器端带宽需求的最简单的方法是把所有通道合并成为一个语音流，或是以点对点的形式进行分布式聊天，从而使聊天完全不通过服务器进行。但即使合并为一个通道，聊天也很可能会使预计的带宽成本加倍。

语音还会带来一些独特的难题：我们通常不能对它的内容进行过滤；只能将它打开或者关闭。我们可以使用的最细致的方法是对特定的玩家或是频道进行过滤，而使用文本过滤就可以很方便地对个别单词进行过滤。

通过发送表情也可以进行玩家间的交流，或是把形象的手势和简短而固定的文本消息组合在一起来进行交流。市场上很多游戏还可以使用热键来触发简单的声音事件：发出命令；在玩家间交换信息或是对受害人进行嘲弄。

自定义的表情需要一些额外的传输时间，这可能会导致某些玩家不愿意使用它们。因此提供一组不需要任何额外传输时间就可以使用的缺省表情是非常重要的。

在 PC 上，玩家可能只需要简单地敲一下功能键就可以触发表情，但是游戏机手柄上并没有这样一排功能键。玩家需要能够使用手柄来输入表情，这也就意味着与 PC 相比，玩家需要在移动和选择菜单上花费额外的时间和精力。

如果某种游戏机可以附加文本输入设备，那么使用这类设备将会给游戏开发带来很多便利。因为开发人员随时可以自行决定，是否实现所有（或是部分）由于在基本配置中不包含键盘而无法实现的功能。通过显示在屏幕上的键盘来进行聊天的方法并不可行，因为即使是键入一个简短的消息也需要花费很长时间。而在游戏机上附加某种文本输入设备，可以使得通常的文本消息变得真正可行，因此如果游戏机上提供了这种设备，就应该加以利用。

由于在游戏机上进行开发受到不少限制，并且用户输入需要使用特定的库，因此仅当游戏被设计为可以使用这些附加的文本输入设备时，才可以用到这些设备。这不仅意味着那些在某个新的控制设备问世之前就已经发布的游戏，并不会因为这个新设备而变得更具有可用性，还意味着是否要使用某种输入设备必须在设计过程的早期就加以计划。

为了减少在游戏中进行聊天带来的麻烦，设计人员也可以刻意地在游戏设计中把对聊天的需要最小化。虽然《卡通城在线》（参见 1.1：《卡通城在线》：面向大众的大型多人游戏）

不是一个基于游戏机的游戏，但却是一个很好的例子。与典型的 MMP 游戏相比，它使用的聊天机制被大大地简化了。

4.6.5　选择目标

出于很多原因，玩家需要选择目标。战斗、通讯、交易、创建小组以及很多随着游戏设计而变化的其他行动都需要选择目标。

由于目标选择有很多不同的用处，玩家在游戏中对另一个玩家，或是 NPC 进行选择时必须尽可能的简单。PC 游戏的玩家习惯于使用鼠标或是键盘热键来方便地选择目标，因此游戏设计有必要保证游戏机上也可以进行同样简单的操作以减少玩家的麻烦。

和其他领域的应用一样，这里的困难来自于游戏机手柄的限制，必须用它来取代键盘和鼠标的功能。因此，究竟应该把哪些功能移植到手柄上呢？ 有一种方法可以让玩家像使用键盘一样用手柄来选择目标，那就是菜单：

- 在可能的目标中选择下一个对手；
- 在可能的目标中选择最近的对手；
- 选择在攻击范围内最弱的对手；
- 选择同一个对手作为当前目标；
- 选择下一个小组成员。

这个列表肯定是不全面的，但是在创建一个简单的目标选择界面时，它可以提供一些有用的信息。想要创建的功能可能太多了，以至于不适合都放在一个手柄上，因此需要根据游戏的设计来决定哪些功能才是最有用的。

另一个在很多游戏中用到的方法是，在玩家进攻时，系统会自动地把处于攻击范围内的对手作为目标。这是第一人称射击游戏中典型的模式。玩家对准什么就攻击什么。这使游戏可以完全避免选择目标这样的问题，从而大大地简化游戏设计。这种隐式的目标选取（implied targeting）会在很大程度上改变战斗、交易和其他交互形式的工作方式，因此尽早地决定选取目标的方式对游戏设计人员来说是非常重要的。

隐式目标选取被应用于很多基于游戏机的游戏中，并且获得了巨大的成功，因此这是一种玩家非常熟悉的方式。它让游戏具有更多的行动性，从而使玩家可以在瞄准和跟随目标时变得更为主动。

4.6.6　菜单

毫无疑问游戏的设计会需要某种形式的菜单，它们可以是简单的游戏保存菜单，也可以是选项和配置菜单、交易、背囊或者玩家需要的其他菜单。菜单只不过是一个用来把命令树中的某些选项隐藏起来的简单方法，使用菜单可以使系统不必一直显示所有选项。而游戏机平台总是没有足够的按键来让玩家可以同时对所有选项进行直接访问。

游戏机游戏中菜单所提供的功能和PC游戏中菜单所提供的功能在很多方面都非常相似，甚至前者会比后者更为复杂。这是因为那些通常会由鼠标来完成的功能在游戏机上很可能会被加入菜单系统。在游戏机游戏中，游戏暂停菜单是非常普遍的，因此可以把它作为起点来

设计一个 MMP 游戏所需要的，更为复杂的菜单系统。游戏机的技术要求（technical requirement）通常会给出一些菜单操作的指导方针，可以遵循这些指导方针，从而使那些层次很深的复杂菜单系统具有一致的感觉。

即使采用了游戏机制造商技术要求中的方式，设计人员仍然可以有很多灵活的选择。有些情况下，譬如说在玩家的多个背囊中进行选择时，如果能让玩家在多个菜单或者菜单项之间进行迅速切换可能会更为方便。此时，除了像通常情况下那样使用方向键来进行移动以外，还可以使用游戏机顶部边缘的按键在子菜单中进行移动，这并不会影响玩家使用方向键在单个菜单或者列表中进行移动。手柄上之所以会有那些多余的按键，就是为了让玩家使用的，因此游戏设计开发人员必须确保在任何有意义的情况下都有效地对它们加以利用。

如果在游戏机平台上的 MMP 最终使用了一个层次很深的复杂菜单系统，这很可能是因为游戏设计人员把那些在 PC 游戏中，通常使用命令输入来实现的功能加入到了菜单选项中。在游戏机中，系统不能弹出一个命令窗口，输入一个配置命令，然后继续游戏，因此所有这些在 PC 上由命令窗口实现的功能都必须移动到菜单中。

设计菜单系统有一个非常重要的注意事项：设计人员很可能会倾向于通过对功能进行分组来把它们放到多个顶层菜单中，从而把菜单系统平铺开，这样用户就可以使用手柄上的不同按键来触发不同的顶层菜单。虽然这对于那些可以很方便地独立开来的小部分功能（譬如说武器选择）来说非常合适，但在通常情况下，最好把大部分菜单功能组合起来，放到一个独立的顶层菜单下。这样就可以把手柄上其余的按键用于其他的游戏功能。

4.6.7 背囊管理和交易

一些 PC 上的 MMP 游戏具有非常复杂的背囊管理系统，以及玩家之间或玩家与 NPC 之间的交易系统。游戏机上的 MMP 游戏需要对所有这些功能进行简化，以使它们易于使用。这意味着，背囊管理系统是另一个在设计的最初阶段就必须注意的方面，只有这样，设计人员才可以确保所创建的功能不仅足以满足游戏的设计需要，还足够简单。

有不少情况需要使用不同的设计，譬如说基于拖曳的背囊管理。因为游戏机没有鼠标，而使用游戏手柄来控制一个虚拟指针非常麻烦，所以必须找到一个替代方法。这个功能最简单的替代方法是使用一个菜单来列出想要的物品，并做出选择，然后再使用菜单选取要放置物品的目的地。

选取并且放置的功能和拖曳是一样的，而且非常类似于使用一个虚拟指针，但这对于玩家来说则更简单。因为通过使用菜单，玩家就可以用一些离散的步骤来把物品从背囊的一个槽（slot）移动到另一个槽中，从而很方便地完成物体的放置。

背囊管理中还有一个需要注意的方面是装备、保存、交易和使用物品的能力。以上每一项都可以通过上面所描述的方法来实现，只需要为物品创建一个特定的目的地就行了。譬如说，玩家可以通过把物品放到角色的肖像上来装备或使用它，也可以把物品放在背囊里面或是背囊的某个槽中来保存它。

交易相对来说则更为复杂，并且根据游戏设计，变化更大，但是它主要也就是通过在菜单中选择，从而在某个玩家的背囊和其他玩家或 NPC 的背囊中移动物品。

4.6.8　持久化存储空间的问题

有些游戏机使用存储卡来存储数据，而另一些具有内置或是外加的硬盘。虽然还有其他的存储方法，但是这两种类型囊括了目前所有游戏机系统。每一种数据存储方式都有其优缺点。

存储卡在移动性方面占有优势。玩家可以轻易地把一块存储卡放在口袋中，带到朋友家里去。存储卡的大小足够保存一些简单的数据，例如朋友列表、首选的服务器列表以及其他小型的数据集。服务器端的数据库可以借助这种少量的本地存储来允许玩家保留一部分他们自己的数据，而不是都放在游戏主服务器上。

存储卡的缺点在于用户不能用它来存储大量的数据。这意味着在一个只有存储卡的系统中，我们只能进行很小的客户端更新，譬如说少量的地图更新或是加入新的角色。即使这样的更新也必须限制在一定范围内，否则玩家必须根据他们想要使用的扩展，保存一堆不同的存储卡。任何对客户端软件的更新都有可能需要通过更新光盘来进行。

相对于只有存储卡的游戏机来说，那些有硬盘的游戏机可以发挥更多的功能。如果有一个硬盘，客户端软件更新就可以加入大量的新游戏内容，甚至可以定期发行扩展包。这样一来，对附加内容的大小进行管理和限制会变得更为方便。开发人员甚至可以为了发布安全补丁或是增加新功能而更新全部的可执行程序。

把所有的数据保存在硬盘上的缺点在于，这些新数据再也不具有移动性了，玩家无法把它们搬到朋友的游戏机上去。这并不是一个很大的问题，因为具有硬盘的游戏机都可以使用存储卡来对那些需要移动的游戏数据进行处理。

在游戏机上使用硬盘可能会带来的另一个问题，游戏开发人员必须强制所有玩家具有相同的更新。他们必须设计一个方案，它不仅需要让所有玩家都能及时更新，还需要让那些新玩家或是把游戏盘带到朋友家去的玩家不必花费几个小时才能把当前版本的游戏世界下载到游戏机中。

如果在目标平台上，硬盘是一个可选件，设计人员就必须事先决定游戏是否要求硬盘或者根本就不使用硬盘。如果试图采用一个折衷方案，那将会很不好。因为如果试图采用某些方法来对硬盘加以利用，只要这些方法不利于只使用存储卡的玩家进行游戏，游戏开发人员就不能使用它们。

4.6.9　补充界面

在完成了对所有问题的讨论后，设计中可能还会有少量功能无法在游戏机上实现。但是不要仅仅因为游戏机硬件不能支持这些功能就抛弃它们。游戏开发的计划中很可能已经包含了为玩家们提供的其他界面（譬如说 Web 站点）。在很多情况下，可以对项目中这一部分进行扩充，以支持那些在游戏机上不能直接实现的功能。

开发人员可以把有些功能（譬如说语音聊天或是文本消息）作为外部工具，提供给那些可以同时使用计算机和游戏机平台的用户群体。但是最好还是由用户自己来决定他们需要什么。

如果玩家可以通过行会网站、玩家组织、关于游戏诀窍的站点或是各种其他相关的 Web 拓展来建立他们在游戏以外的存在，他们之间的集体感就会加强。对玩家在这些领域进行鼓励可以强化他们的归属感，使得玩家集体更加稳定，并且使他们觉得自己是集体的成员，而不仅仅是一个游戏玩家。

鼓励外部集体的一个方法是给予他们访问任何游戏数据的权限，只要不会带来安全/隐私方面的风险；不会引起带宽上的问题；不涉及专有知识。这些信息可以包括哪个服务器正在运行；哪些玩家正与游戏相连（在他们的许可下）；集体广播消息；用户和行会统计信息（同样需要玩家的许可）以及很多基于特定设计可以提供的其他信息。

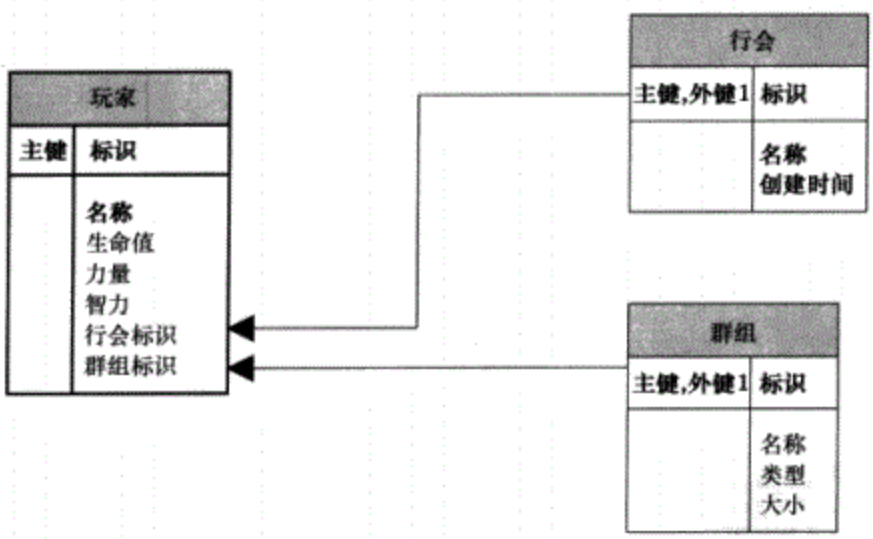
4.6.10 总结

MMP 游戏已经在游戏机上出现了，游戏机为 MMP 市场带来重要而持久的影响只是时间问题。但是考虑到 MMP 游戏在游戏机上生命周期的特殊性，游戏的设计人员必须在设计阶段的初期就采取不同的方法。

网络、语音、大规模存储以及其他设备对于基于游戏机的 MMP 游戏的成功来说，具有重要意义。要让 MMP 游戏能够在游戏机上得到成功，必要的前提就是要把这些先进的硬件内置在游戏机中或是作为附件提供给用户并且被他们所接受。即使在游戏机上开发 MMP 游戏存在诸多困难，这个市场细分被一些重要作品所开辟也只是迟早的问题。

第 5 章

数据库技术


玩家
主键
标识
名称
生命值
力量
智力
行会标识
群组标识
行会
主键,外键1
标识
名称
创建时间
群组
主键,外键1
标识
名称
类型
大小

5.1 关系数据库管理系统入门

Jay Lee, NCSoft Corporation
jlee@ncaustin.com

关系数据库管理系统（Relational Database Management System, RDBMS）就是，通过一种被称为结构化查询语言（Structured Query Language，SQL）的标准语言，来提供数据存储、获取（retrieval）和操纵服务的软件。它也是企业环境中，各种商业应用程序进行数据持久化的事实标准。很多商业性公司在市场上销售 RDBMS 软件，譬如说[MSSQL]、[ORACLE]、[SYBASE]和[BORLAND]，同时也有不少开源的选择，譬如说[MYSQL]和[PGSQL]。

本文的目的是对 RDBMS 系统的基础知识进行介绍，从而为那些对这类软件了解甚少，或是没有这方面背景的用户提供一些帮助。希望本文可以激发他们在 MMP 游戏中使用 RDBMS 的兴趣，并对此进行进一步的研究。不仅如此，本文中关于数据库管理系统的讨论也是第 5.2 节“使用关系数据库管理系统来编写数据驱动的 MMP 游戏”的基础。

5.1.1 表

RDBMS 中的数据是以表（table）的形式组织的。表包含了不同数据类型的列（column），表中的每一个条目称为行（row），例如表 5-1。

表 5-1 雇员表

雇员编号（PK）	名	姓	社会保险号
1	Jay	Lee	999-99-9999
2	John	Doe	888-88-8888

每个表通常都有一个主键，它是一个或多个列的组合，用来在表中独一无二地标识一行。这个主键创建的独一无二的索引通常是对表中某一行进行访问的最佳方法。在雇员表中，雇员编号这一列是主键（用 PK 表示）。

5.1.2 数据查询和关联

用下面的 SQL 语句可以获取 Jay Lee 对应的那一行（SQL 语句的格式

很灵活，可以加入空格和换行来保持清晰）。这个查询将返回表 5-2 中所示的结果。

```
SELECT *
FROM Employee
WHERE EmployeeId = 1
```

表 5-2　　一次简单查询的结果

雇员编号（PK）	名	姓	社会保险号
1	Jay	Lee	999-99-9999

在对系统进行建模时，通常要为感兴趣的领域中，每一类实体创建一个表。譬如说，MMP 游戏中，可能可以找到下列表：玩家、角色、魔法、怪物、财宝、技能、背囊以及银行。关系系统的强大之处在于，它对不同的表和实体之间的关系进行描写的能力。有了表 5-3 中的部门表，就可以通过把部门表中的主键作为图 5-4 中雇员表的外键（foreign key），来为每个雇员指定一个相应的部门。

表 5-3　　部门表

部门编号（PK）	部门名称
1	Jay
2	John

表 5-4　　雇员表

雇员编号（PK）	名	姓	社会保险号	部门编号（FK）
1	Jay	Lee	999-99-9999	1
2	John	Doe	888-88-8888	2

部门编号的使用使得下面这些规则被形式化：

- 每个雇员只对应于一个部门；
- 一个雇员可以对应的有效部门必须在部门表中。

不仅如此，这些表中的引用完整性（referential integrity）也得到了维护。只有当雇员表不再引用某个部门时，才能把它从部门表中删除。从数据完整性的角度来看，这非常有用：在删除一个雇员对应的部门前，必须先把他分配给其他部门。

由于在游戏世界中，玩家必须对很多复杂的关系进行捕捉和管理，这个功能可以给 MMP 游戏带来很大的好处。由于在游戏代码中，对引用完整性进行手工的维护非常困难，MMP 游戏中往往出现很多错误。

下面这个 SQL 查询可以回答这个问题：“程序部门中所有雇员的名字是什么？”。

```
SELECT FirstName, LastName
FROM Employee
WHERE DepartmentId = 1
```

一次 SQL 查询可以包含多个表。通过使用两个具有相同值域的列，可以对多个表进行连接（join）操作。如果在一次连接中，程序要引用多个名字相同的列，SQL 语法就会要求程序员必须用表的名字（或是一个相关的别名）来限定某个列以避免二义性。

譬如说，要得到不同部门的员工列表，可以使用下面的代码。

```
SELECT D.DepartmentName, E.FirstName, E.LastName
FROM Department D, Employee E
WHERE E.DepartmentId = D.DepartmentId
ORDER BY D.DepartmentName
```

SQL 使工作人员可以很方便地对数据进行查询。关系数据库的另一个强项是它可以把数据以一种和查询无关的方式进行存储。

5.1.3 关系类型

关系主要分为三种类型：一对一、一对多和多对多。这些关系往往可以通过与一个领域中的专家（对于 MMP 游戏来说，这就是设计人员）面谈来发现。他们需要回答，对于两个需要建立联系的特定实体 A 和 B 来说，“每一个 A 会与多少 B 对应？”以及“每一个 B 会与多少 A 对应？”譬如说，给定一个学生，那么他可以同时修多少课程？给定一个课程，同时会有多少学生登记这个课程？

表 5-5 和表 5-6 分别是雇员表和公司车辆表，它们之间是一对一的关系。实践中，这种类型的关系很少见。这个例子表明一个雇员只能对应于一辆公司车辆，并且一辆公司车辆也只能对应于一个雇员。有趣的是，在一对一的关系中，无论是哪个主键都可以被用来作为另一个表的外键，并且都可以精确地描绘这种关系。因此，这个例子也可以进行修改使得雇员编号成为公司车辆表的一个外键并且仍然保持正确。

表 5-5 雇员表，它和公司车辆表形成一对一关系

雇员编号（PK）	车辆编号（FK）
1	1
2	2

表 5-6 公司车辆表，它和雇员表形成一对一关系

车辆编号（PK）	型号	年份
1	BMW M1	2001
2	Chevrolet Malibu	2002

表 5-7 和表 5-8 中的部门和雇员表是一对多的关系。在这种情况下，一个雇员属于一个部门。而一个部门可以有多个雇员。在一个一对多的关系中，外键必须出现在那个实体是关系中“一”这边的表中。因为一个雇员只能属于一个部门，部门编号必须出现在雇员表中。把雇员编号放在部门表中将限制每个部门只能有一个雇员，很明显这不是游戏设计想要的结果。

它们也可以是其他表的外键。

在前面的表中，雇员表中的部门编号列就是一个属性，它的有效值都属于部门表的主键，因此它是一个外键。雇员表中还可以有名为“性别”的属性，它的值域可以是“男”和“女”。这个值域并不依赖于另一个表，因此它不是一个外键。MMP 游戏中可能会出现下列角色属性：类别标识、生命值、当前等级、力量以及敏捷度。

5.1.5 正规化

把数据库中所有的模型（schema）整合在一起的过程（发现所有想要的实体；为它们建立合适的关系；在每个表中加入想要的属性来捕捉它的本质）已经被广为讨论了[Date] [SKS]。

要做到这点，只需要知道，有很多规则可以用来确定数据库对某些期望特征遵守的程度就行了。这些特征被设计为可以让数据库能够不受使用这些数据的应用程序的改变的影响；它们不仅可以使得存储冗余最小化，还可以更为优雅地处理这些变化。遵守这些规则的模型被称为是“正规的（Normalized）”[Codd]。

由于一些性能上的原因，现实生活中的应用程序很少会把数据库中的所有模型都正规化。有经验的数据库管理员会先获得模型所要表达的“逻辑”视图，并且把它转换为一个，可以为使用最频繁的查询路径提供最高性能的“物理”实现。在进行这些决策时，他还会考虑维护反向正规化（denormalized）的数据带来的额外工作量。

5.1.6 操纵数据

SQL 不仅可以用来查询数据库，还可以用来操纵表中的数据。它提供了从表中增加、修改和删除行的语法。下面的语句在部门表中加入一行。

```
INSERT
INTO DEPARTMENT
VALUES (3, 'Accounting')
```

下面的 SQL 语句会删除雇员表中所有雇员编号大于 12 的行。

```
DELETE
FROM EMPLOYEE
WHERE EMPLOYEEID > 12
```

下面的语句会为每个雇员加薪 10%。

```
UPDATE
EMPLOYEE
SET SALARY = SALARY * 1.1
```

5.1.7　总结

关系数据库管理系统的功能非常强大，它可以为工业强度的应用程序提供良好的数据持久化和操纵服务。很多 MMP 游戏已经通过使用这些功能来为用户提供更好的服务。

虽然我们还可以找到很多关于 SQL 和 RDBMS 的知识，但是这个入门足以作为进一步学习的基础了。请阅读本文的参考文献以获得更多关于有效使用 RDBMS 的材料。此外，本书第 5.2 节讨论了 RDBMS 在 MMP 游戏中的一个实际应用。

5.1.8　参考文献

[BORLAND] Interbase product Web site, *http://www.borland.com/interbase/*.

[Codd] Codd, E. F., "Relational Completeness of Data Base Sublanguages," R. Rustin, ed., in "Database Systems," Prentice Hall and IBM Research Report RJ 987, 1972: pp. 65-98.

[DB2] IBM DB2 product Web site, *http://www-3.ibm.com/software/data/*.

[Date] Date, C. J., *An Introduction to Database Systems, 7th Ed.*, Addison-Wesley, 1999.

[MSSQL] Microsoft SQL Server product Web site, *http://www.microsoft.com/sql/default.asp*.

[MYSQL] MySQL Open Source Database, *http://www.mysql.com/*.

[ORACLE] Oracle product Web site, *http://www.oracle.com/*.

[PGSQL] PostgreSQL Open Source Database, *http://www.pgsql.com/*.

[SKS] Silberschatz, Korth and Sudarshan, *Database System Concepts, 4th Ed.*, McGraw-Hill, 2001, *http://www.bell-labs.com/topic/books/db-book/*.

[SYBASE] Sybase product Web site, *http://www.sybase.com/*.

5.2 使用关系数据库管理系统来编写数据驱动的 MMP 游戏

Jay Lee, NCSoft Corporation
jlee@ncaustin.com

近20 年来，关系数据库管理系统（Relational Database Management System, RDBMS）一直是美国企业所使用的商业应用程序中，进行数据存储和获取的主要软件，因此也不难理解为什么越来越多的 MMP 游戏开发人员试图使用这项技术。

MMP 游戏具有很多和其他商业领域一样的需求：收费、客户支持以及账户管理是一些显而易见的例子。不仅如此，使用关系数据库系统来维护玩家和游戏世界的状态也是一个很自然的选择。这样一来，无论玩家正在游戏之中还是退出游戏了，或者是服务器重启了，它们的状态总是能够被相应地持久化。

本文的目的是以上面描述的这些方式为基础，对关系数据库系统的使用进行进一步的研究，并且把它应用到游戏中，以发布一个数据驱动的游戏系统。而玩家在虚拟世界中行走时，就是在对这个系统进行体验。

在附带的光盘中，本文的例子和源代码将使用解释语言 Python 来编写，并且使用特定于 *Microsoft* 的 RDBMS（MSSQL Server）的 SQL 语法。然而，这些都可以被很方便地应用在 SQL 语言的其他变种以及其他 RDBMS 中。

5.2.1 最明显的方法——为什么它不可行

多年来，随着 RDBMS 在商业领域的应用，它的总体性能得到了显著的增长。由于硬件性的提升以及优化技术的重大进展，关系数据库厂商开始宣称，其产品的性能基准可以达到每秒钟进行成千上万次事务处理（Transaction Per Second, TPS）。

这听上去很不错，因为典型 MMP 游戏的持久化需求并不比这个数字大。但至少有 3 个重要的原因让游戏开发人员不会去购买一套 RDBMS 系统（软硬件），并且开始以实时事务处理的形式实现他们的持久化需求。

第一个原因是成本。MMP 游戏的开发成本已经很高了。为了要达到

厂商所宣称的性能，开发人员必须在多处理器硬件上进行巨大的投入，再加上通常商业 RDBMS 的许可费用是按照处理器个数来计算的，他们很快就会发现，对这种方案加以实施和管理所需要的费用是惊人的。

假设游戏开发人员为游戏产业中少数几个能够负担这么高的费用的公司之一工作，他们仍然必须面对一个问题：只有销售团队才会频繁地使用性能基准，当他们对游戏在实际环境中，与数据库进行实时 I/O 的能力进行预测时，性能基准并不会有太大帮助。因此，游戏开发人员使用 RDBMS 来作为 MMP 游戏解决方案的第二个原因是因为它对可以达到的性能进行预测非常困难，并且只有在开发后期才能进行测量。而到了那时，对架构进行任意的重大修改都需要付出大量的费用，并且还可能导致发布延期（这和上面的第一点理由相同，也说明了为什么“使用更强大的硬件”在一开始就是一个险坡（slippery slope））。

还有一个理由可以让游戏开发人员知道，对数据库进行实时 I/O 并不是一个好想法：一个运行中的 MMP 游戏产生停顿是不可接受的，而在等待数据库完成事务时，不可避免地会产生停顿。这些停顿不仅会发生在任何时候，也可能会发生在一些很难重现的场景中，更不用说调试和修正它们有多困难了。当玩家正在进行激烈的游戏时，服务端可不能为了进行数据持久化而显示一个“等待”光标。

5.2.2　可行的方法

接下来本文继续对一些技术（以及一些实现方法）进行讨论，游戏开发人员可以使用这些技术从使用数据库中获得好处，而不会给游戏带来负面影响。

访问数据最快的方式就是，让它们在工作人员需要时已经在内存里了。这意味着，虽然在运行时需要的所有数据都保存在数据库里，游戏的开发人员仍然应该尽可能地预先载入（preload）数据，以避免直接向数据库获取数据带来的影响。譬如说，如果进行每一种作战武器描述的所有数据都保存在一系列表中，可以在游戏服务器启动时载入这些数据。这样一来，每当游戏中发生战斗时，就可以以最快的速度查询数据，并且把它们使用在与战斗相关的计算中（本文后面中将会使用几个小节来介绍我们应该怎样组织这些数据）。

对于要查找的数据按需进行缓存是上述方法的一个变种，它对于那些只有在某些特定事件发生后才会被用到的数据来说非常有用。在 MMP 游戏中，典型的情况就是那些与玩家相关的数据。玩家的物品信息、任务日志等在他们离开游戏时并不需要保存在内存中。如果游戏系统可以使用某些异步方法来访问数据库中的数据，那就可以利用玩家登入游戏的时间来缓存与他们相关的数据。

对于任何形式的数据延迟载入（deferred loading）来说，对数据库进行异步访问的机制都是必不可少的。异步访问不是 RDBMS 的内建操作，因此，必须把这个功能构建在服务端架构中。譬如说使用一个服务端线程来进行数据访问。当数据访问线程获得所需要的数据时，游戏主线程会得到通知。主线程也可以在数据访问线程进行处理时，向其发出请求，这不仅需要一个比较稳健的线程间通讯机制，还需要一个支持多线程的数据库访问库。

游戏开发人员还可以使用一种更为简单的方法来实现这个异步机制：创建一个独立的数

据库服务程序，这样游戏主服务器就可以通过网络消息与之通讯。每当游戏需要与数据库进行交互时，它会生成一个请求并把这个请求作为网络包，发送给数据库服务程序，然后继续运行。当游戏服务器接受到这个请求的响应消息时，就对这个数据进行处理，然后再继续游戏主循环中的其他任务。

客户端也应该使用 RDBMS 来访问数据。然而，客户端永远都不能通过直接访问数据库来获得数据。为了提高访问速度并且尽可能地不影响带宽，凡是在客户端能够静态访问的数据都应该静态访问，也就是说，它不仅应该在客户端启动时载入，还必须以一种容易被访问的方式保存。并且，它必须足够“静态”，从而可以延迟对它的更新。通常对这些数据的更新可以使用游戏中的补丁机制来实现。后面所介绍的技术同样也适用于客户端数据，虽然游戏开发人员必须仔细地选择究竟应该把它们使用在哪些数据上。

5.2.3 获取数据

本文并不会对怎样访问数据库中的数据进行详细介绍。这方面已经有很多解决方案了，其中包括数据库厂商提供的，带有特定扩展的库、ODBC（Open Database Connectivity，平台无关）库、微软的 ADO（ActiveX Data Objects），以及其他第三方方案。游戏开发人员应该根据服务器整体架构来选择一个最适合的方案，并且优先选择那些可以简单地把获取到的数据映射到游戏所使用的程序设计语言中的数据结构的方法。

此外，游戏的设计还应该优先选择那些支持数据库存储过程（stored procedure）的方案。这些存储过程包含了使用 SQL 语句编写的查询，它们可以被数据库的任何用户重用。它们还有一个额外的优势：数据库会对它们进行预编译，这可以带来很大的性能提高。不仅如此，它们还具有通用的接口，这样就可以方便地在服务端创建一个通用的机制，从而来对数据进行任意的查询和操纵。此外，还应该尽可能地避免在代码中嵌入 SQL 表达式。

5.2.4 常量模块

本文所介绍的第一个技术致力于，弥补数据库中的数据与程序，对这些数据使用之间的缺口。它还使游戏程序员可以避免使用硬编码的数据，从而保持良好的编码习惯。假设程序员想要针对接受到的不同类型的聊天消息进行不同的处理。通常我们会编写如下的代码。

```
# chatprocessor.py

GLOBAL = 1    # defines of constant values to
RADIAL = 2    # maintain readable code
WHISPER = 3
⋮
if chat_type == GLOBAL:
   # do global chat stuff
elif chat_type == RADIAL:
```

```
    # do radial chat stuff
elif chat_type == WHISPER:
    # do whisper stuff
```

如果在数据库中有一个像表 5-12 那样的聊天类型表。

表 5-12　　聊天类型表

聊天类型编号	聊天类型描述
1	Global
2	Radial
3	Whisper

我们可以编写下面的代码。

```
# chatprocessor.py

import chattype  # a generated constant module

if chat_type == chattype.GLOBAL:
    # do global chat stuff
elif chat_type == chattype.RADIAL:
    # do radial chat stuff
elif chat_type == chattype.WHISPER:
    # do whisper stuff


# chattype.py - this module is autogenerated from the database

GLOBAL = 1
RADIAL = 2
WHISPER = 3
```

在 Python 中，可以通过两种方式来创建常量模块 chattype.py。第一种方式是通过某种方法来把它作为一个源文件生成。这个文件就像其他源文件一样，可以通过版本控制进行管理，并且被需要它的其他代码引入。由于 Python 是一种解释语言，也可以在运行时动态地创建这个模块。在程序启动时，程序员获取这些数据，并且用它们来创建这个模块。使用这种方法需要注意，在创建这个模块之前，代码中不能对它进行引用。

使用常量模块的另一个好处是，创建了一些符号（symbol）。游戏代码中的其他部分可以使用这些符号来建立查找表（本文将会在后面介绍这一技术）。请参照本书配套光盘上的 constantmodule.py，以获得关于使用数据表来生成常量模块的细节。

5.2.5 查找表

查找表是一种在内存中的数据结构，它根据数据库表中的内容来创建。在运行时，这些数据结构将用来对游戏的各方面进行数据驱动。譬如说，在战斗处理中，服务端可以根据攻击者手中武器的编号（WeaponId）来查询命中率（PercentageToHit）、攻击力（DamageAmount）和对不死族的攻击奖励（BonusDamageAgainstUndead）。在以后对战斗处理进行调整时，只需要更新武器表中的数据就可以了。表 5-13 是数据库中的武器表。

表 5-13 **武器表**

武器编号	命中率	攻击力	对不死族的攻击奖励
1	90	8	3
2	85	10	0

使用程序对这些数据进行处理后可以生成下面这个模块，它创建了一个名为“lookup”的变量，这个变量是一个以武器编号为键的词典（dictionary），它的值是武器表中其余列的元组（tuple）。

```
# weapon.py - a generated lookup structure

lookup =
{
   1 : (90, 8, 3),
   2 : (85, 10, 0)
}
```

游戏代码中，也使用这个查找表，如下所示。

```
# combat logic example

import weapon

def Attack(attacker, defender):
   attackingWeapon = attacker.GetWieldedWeapon()
   stats = weapon.lookup[attackingWeapon.GetId()]
   percentToHit = stats [0]
   damageAmount = stats [1]
   undeadBonus  = stats [2]

   if _AttackSuccessful(percentToHit):
      totalDamage = damageAmount
      if defender.IsUndead():
         totalDamage += undeadBonus
      defender.TakeDamage(totalDamage)
```

这个技术是本文的重点。它说明了怎样才可以从数据库中获取数据，并且让游戏代码对

其进行最有效的访问。

这个方法具有一个很大的好处，数据库模型可以保持正规化。一旦进行了正规化，关系数据会变得更为灵活，并且不会由于内容增加，或是应用程序实现方法的改变而受到影响。当速度并不是至关重要时，可以把这些数据查询操作延迟到游戏运行的某些特定时刻中进行。在这些时刻，游戏速度必须不是很关键（也就是说，这些特定时刻在游戏主循环之外），最明显的莫过于游戏服务器启动或是玩家登录。

每当使用这些数据来生成查找表时，游戏程序员就把数据从正规化的形式转换为便于访问的数据结构。这时，不必考虑数据源是怎样组织的，只需要考虑怎样才能在运行时，对数据进行最有效的访问。

譬如说，如果可以更方便地查找排序了的数据，就可以使用内建的 SQL 排序机制来进行排序。可能有些关系数据分散在 3 个不同的表中，但是游戏开以人员希望使用一个查找表来访问它们。把这 3 个表连接到一个单一的结果集中，并且用这个结果集来生成查找表就可以达到这个目的。如果要以两种不同的方式访问某个特定的数据子集，就可以根据原始数据生成两个不同的查找表，相对于其他因素来说，查询速度总是处于支配地位。

Python 具有一些有用的内建数据结构，这使得程序员可以方便地在运行时创建查找表。正如上面的 weapon.py 所示，可以使用一个词典，来创建一个从武器编号到武器属性的查找表。因为 Python 可以灵活地对数据进行处理，所以程序员可以对更复杂的（具有复合键以及混合数据类型的）表进行同样的操作。

下面这些数据表也会对游戏中，设计人员对武器的处理有所影响。表 5-14 是效果类型表，它描述了客户端可以把哪些声音/图象效果应用在战斗武器上。表 5-15 是武器效果类型表，它描述了玩家使用某种武器时可以应用的各种效果，还包括了一个以毫秒表示的延迟时间（DelayTime）以及一个资源名称（ResourceName）；前者表示在攻击开始后多久才会播放效果，后者表示产生效果所需的资源。

表 5-14　　效果类型表

效果类型编号	效果类型描述
1	粒子效果
2	武器动画
3	音效

表 5-15　　武器效果类型表

武器编号（PK）	效果类型编号（PK）	延迟时间	资源名称
1	1	200	gamedata/particles/part01.ptl
1	1	2000	gamedata/particles/part02.ptl
1	2	200	gamedata/anim/weapon1.ani
1	3	100	gamedata/audio/swoosh.wav

Python 可以生成下面这样的查找表。

```
# weapon.py - generated lookups

weapon =
{
   1 : (90, 8, 3),
   2 : (85, 10, 0)
}

weaponVisualEffect =
{
   (1, 1) : [(200, 'gamedata/particles/part01.ptl'),
             (2000, 'gamedata/particles/part02.ptl')
            ],
   (1, 2) : [(200, 'gamedata/anims/weapon1.ani')]
   (1, 3) : [(100, 'gamedata/audio/swoosh.wav')]
}
```

武器视觉效果（weaponVisualEffect）这个变量也是一个词典，但是它的键是一个元组，这是因为必须同时提供武器编号和效果类型编号，才能获得所需要的数据。此外，值得注意的是，使用这个查找表的代码必须知道这个函数的返回值是一个列表，因为同一个武器可能可以使用不止一个类型的效果。还要注意的是，程序员可以在同一个模块中放入两个查找表。这对于关系数据集来说是有意义的，只需要给它们起不同的名字就可以了。

总之，给定一个游戏的数据库模型就可以创建出一系列的查找表，并且可以用它们在运行时进行有效的查询，从而实现数据驱动的游戏模式。光盘上的 lookup.py 中，有一些用来创建各种不同查找表的工具函数。

5.2.6 字符串表

下面要介绍的技术可以让游戏开发人员通过使用关系数据库，来解决 MMP 开发中最大的问题：怎样让网络上传输的数据量最小化，从而使得带宽开销尽可能的低。举一个例子，假设游戏开发人员想要发送一个消息，让客户端播放一段声音/动画或是载入一个特定的网格，他们必须在数据包中指定一个游戏资源。当这些资源被创建后，它们通常会被保存在某个处于版本控制系统管理下的位置，通常这被映射为游戏可以访问的某个文件路径，譬如说 thegame/gamedata/audio/footsteps.wav。

有必要在游戏开发过程的某些时刻，生成一个可以通过数字编号来找到文件路径的查找表，这样，在网络包中，只需要传输一个数字（譬如说，65）而不是字符串“/gamedata/audio/footsteps.wav”。

在客户端，程序可以使用下面的代码。

```
def ParseAudioPacket(packet):
```

```
audioFileId = packet.GetId()
audioFileName = stringtable.lookup[audioFileId]
Play(audioFileName)
```

程序员想要在系统中支持多个逻辑字符串表。因为在 MMP 游戏开发中程序员会对 ASCII 字符串按照不同方式进行分组。因此，他们使用表 5-16 中所示的表。这样他们就可以按照使用环境，把相关的字符串存放在一起。譬如说，当一个设计人员在处理数据库中，与施放魔法有关的数据时，他需要知道当前可以使用的动画列表。

表 5-16　　字符串表类型表

字符串表类型编号	字符串表类型描述
1	动画
2	网格
3	声音

把要使用的所有 ASCII 字符串放在一个表中，可以确保不存在重复的编号和字符串。设计人员并不需要关心每个字符串所对应的编号是什么，只要求它是独一无二的。让字符串编号这一列自动增长（auto increment），这样就可以让数据库来为游戏系统维护这个数字，并且每当加入一个新字符串时，它都会自动加一。字符串表的结构如表 5-17 所示。

表 5-17　　字符串表

字符串编号	字符串值	字符串表类型编号
35	/gamedata/audio/footsteps.wav	3
36	/gamedata/animation/fly.ani	1

使用光盘中的 stringtable.py 文件中的代码，可以把字符串表作为一个模块来创建以及怎样创建前面提到的字符串查找表。文件 storedproc.sql 所包含的 MSSQL Server 代码是一个向字符串表中添加新表项的存储过程。

于是，字符串表成为游戏中所有 ASCII 字符串的中央存储机制（注意任何需要显示给用户的数据并不包括在内，因为那些数据需要进行本地化。下面本书会介绍对本地化进行处理的方法）。在产品开发过程中，开发人员可以把底层架构构建在字符串表上。一旦创建了一个新表项，它就可以被整个项目访问，其他数据表也可以对它们进行引用而不必担心数据是否重复，开发人员也不需要在所有资源到位后再手工生成这些查找表。

5.2.7　向客户发送数据

所有在这里描述的技术，都可以用来为游戏客户端应用程序提供数据访问。游戏开发人员希望尽可能把所有数据都放在客户端，以避免在网络上进行传输。当然，本文这里假定这不包括任何不能让玩家获得的敏感数据。那些可以采用上述技术的数据包括物品描述文本、帮助文本、字符串数据、玩家状态信息、特效参数以及 NPC 对话的文本。

这里的关键在于程序员在对数据库中的数据进行更新时，必须保持这些数据也得到更

新。最简单的方法莫过于使用一个批处理过程来重新生成所有的客户端数据，并且用包含这些数据的最新文件来更新版本控制。当有新的更新时，开发人员可以运行这个批处理过程。如果对客户端进行自动构建（automated build）的话，也可以把这个批处理过程整合进来。

5.2.8 本地化

本文所介绍的最后一个技术可以被用来处理 MMP 游戏中的另一个重要问题：游戏本地化。通常游戏文本的本地化只面向客户端，游戏服务端很少会使用这些数据。另一方面，把本地化数据保存在 RDBMS 中非常有用。相对于把这些文本嵌入到资源文件，或是（更坏的情况下）分散在源代码中来说，把它们放在数据库中可以使本地化工作人员能够更好地对它们进行管理。参见表 5-18、表 5-19 和表 5-20。

表 5-18 语言表

语言编号	语言名称
1	英语
2	法语
3	韩语

表 5-19 本地化条目表

本地化条目编号	本地化条目描述
1	欢迎消息
2	确认按钮
3	取消按钮

表 5-20 本地化文本表

语言编号	本地化条目编号	翻译
1	1	N'Welcome to the game, we are glad...'
1	2	N'OK'
1	3	N'Cancel'
2	2	N'Oui'
2	3	N'Non'

语言表中包含了游戏支持的所有语言，要加入新的语言也很方便。本地化条目表包含了游戏中每一个必须被翻译的文本。本地化文本表把语言和本地化条目关联起来，这样就可以为游戏中每一个文本的不同语言版本，在这个表中创建一行相应的数据。注意在“翻译”列中包含了多字节编码（multibyte）的字符串数据，这意味着它是一个多字节编码的字符数据类型。因为大多数 RDBMS 都具有内建的多字节支持，并且 MMP 游戏在韩国和台湾等地非常流行，那些地区的本地语言都需要多字节支持才能正确显示，因此很明显，游戏的开发应

该使用这个类型。

通过为每个语言生成一系列查找表，程序员可以编写出下面这样的本地化代码。请参考 Python 源文件 localization.py 以获得关于创建实现这个功能所需要的查找表和常量的细节。

```
import localization
import localizeditem
import dialog          # some dialog class

def DisplayLoginDialog(locationX, locationY):
    dlg = dialog.Dialog(locationX, locationY)
    dlg.AddButton(localization.lookup[localizeditem.
        BUTTON_OK], _OnOK)
    dlg.AddButton(localization.lookup[localizeditem.
        BUTTON_CANCEL], _OnCancel)

def _OnOK():
    # handle OK button pressed

def _OnCancel():
    # handle Cancel button pressed
```

5.2.9　总结

本文说明了只需要使用一些简单明了的技术就可以在 MMP 游戏系统中利用 RDBMS 的强大功能。这使得整个游戏可以真正的由数据来驱动，这对于一个需要在很多年的生命期中不停地进行支持和发展的服务来说是非常有用的。

不仅如此，这些技术使得工作人员可以把过去保存在配置文件或是二进制数据文件中的数据合并到一个单一的、易于访问的地方，这并不会对游戏的运行速度带来任何影响。本文所介绍的技术还可以确保，游戏开发人员可以在运行时以一种比较高效的方式获得这些数据。

一旦游戏以这种方式基于一个数据库运行，就可以实现很多新功能。通过与其他第三方工具一起使用，RDBMS 可以很方便地实现报表和数据挖掘等功能。数据库会在其内部保持引用完整性，这可以大大减少那些由无效引用引起的问题，而这些问题在像 MMP 游戏这样的复杂软件中非常常见。

因为可以通过改变数据来对游戏进行扩展和调整，那么进行维护和错误修正的开发负担将大大降低，这样开发人员就可以把更多的时间用在改进游戏以及发展客户上。

5.2.10　参考文献

[Python] Python language Web site, *http://www.python.org*.

[MSSQL] Microsoft SQL Server product Web site, *http://www.microsoft.com/sql/default.asp*.

[TPC] Transaction Processing Performance Council Web site, *http://www.tpc.org*.

5.3 MMP 游戏中的数据驱动系统

Sean Riley, Ninjaneering
sean@ninjaneering.com

数据驱动系统早已成为游戏开发人员使用的标准工具之一。所有类型的游戏，无论是角色扮演游戏、实时策略游戏还是第一人称射击游戏，都在开发过程中使用了数据驱动系统[Rabin]来提供必要的灵活性和扩展性，这样做也使得玩家可以在游戏发行后为它创建“扩展包（mod）”。对于 MMP 游戏来说，“发行”只是一个开端而已。开发人员和运营团队必须在游戏的整个生命周期中和他们自己的数据驱动系统打交道。这使得相对于单机游戏和小型多人游戏来说，MMP 游戏更加需要一个强大而稳定的数据驱动系统。

本文从不同角度介绍了 MMP 游戏中的数据驱动系统。本文会先介绍它的优点和功能，然后会对数据源（data source）和数据驱动的游戏架构（data-driven game architecture）中的一些实现细节进行描述。

5.3.1 在 MMP 游戏中使用数据驱动系统的优点

从技术角度看，MMP 游戏具有下面这些根本特征：它们需要大量的代码，这些代码的生命周期很长（包括整个开发过程以及发行后的时间），并且它们还需要大量的内容。这些特征会给开发人员带来很多问题，而这些问题都可以通过使用数据驱动系统来解决。

1. 降低开发成本

数据驱动系统可以从很多方面降低 MMP 游戏的开发成本。

- 可以使更多的项目成员在项目早期就参与到开发过程中。
- 使游戏设计人员可以立即获得对他们想法的反馈。
- 减少所需的代码量。

数据驱动系统使得非程序员不必依赖于程序员就可以对数据进行操纵。这样游戏设计人员和开发团队中的其他成员在项目早期就能亲自进行工作了，这也会让他们感到离实现更为接近。这是一个很大的优点，它使得在开发初期，更多的团队成员可以切实地投入到工作中。

游戏设计是一项实践性很强的工作，如果设计人员可以在开发环境中立即获得他们的想法的反馈，其开发效率就会大大提高。同样，数据驱动系统使游戏设计的迭代（iteration）更为迅捷。它可以让设计人员和程序员

获得对他们想法的反馈，从而在开发进程的早期就发现技术和游戏设计中的优缺点。同时，它还能使设计人员和程序员对不同的设计和实现进行实验时，所需改变的代码量更小。

使用数据驱动系统所需要的代码量通常比较小。它可以把代码中的重复部分重构[Fowler99]为由数据驱动的单一代码片断。这项技术可以应用在游戏的很多方面，包括类定义、输入验证、由玩家调用的命令以及游戏对象可以进入的状态。更少的代码意味着所需的编程时间更短，这最终导致开发成本的降低。

2. 降低维护成本

数据驱动系统有助于降低 MMP 游戏的维护成本。这是因为它不仅可以降低维护人员在对游戏进行修改时的风险，还可以使这些修改更为可靠。并且，使用数据驱动系统还可以减少需要维护的代码量。对游戏的开发而言，能够完全通过修改数据来对游戏进行修改是非常有用的，因为修改数据比修改代码更为可靠。虽然修改数据并不是完全没有风险的，但是它引起意外问题的可能性较低，需要进行的测试也较少。进行数据修改不需要重新编译，（有时）甚至不必关闭服务器。使用数据驱动系统对游戏服务器进行实时修改非常有用。

那些时间充足的无聊玩家会试图去发现 MMP 游戏中每一行代码的错误，然而，更多的代码就意味着更多的错误。MMP 游戏的生命周期很长，不可避免地会有新的程序员加入开发团队，他们必须读懂现有的代码。新程序员需要理解的代码越少，就意味着他们成为多产的团队成员所需要的时间也越短。

以上事实说明，代码越多，维护成本就越大。数据驱动系统可以减少代码量，从而降低维护成本。

3. 更快地创建游戏内容

数据驱动系统可以加速游戏的开发，这是因为它可以使内容创建独立于游戏代码，有时甚至可以在没有编写任何游戏代码前就开始内容创建。对于一个需要创建大量内容的游戏来说，尽早消除这些内容创建流水线（pipeline）上的障碍是必须的，而数据驱动系统就是实现这个目标的方法之一。

通过在内容创建过程和源代码之间建立一个缓冲，游戏设计人员在任意时候都能够迅速地创建内容。不仅如此，这还让他们可以创建一些工具来更好地创建游戏内容——这些工具与游戏技术相对独立，通过对它们的使用，非程序员可以在项目早期就成为多产的团队成员。

程序员可以为这些数据驱动的内容创建一个具有标准格式的内容库，它独立于游戏代码。这样一来，在游戏代码不能正确编译或是某个程序员休假时，仍然可以继续进行游戏开发。甚至可以在还没有编写任何支持代码前就加入游戏内容。通过使用数据驱动系统，把内容创建过程和开发过程分离开，就可以更加灵活地安排这两项工作。对于那些需要大量游戏内容和长时间开发的 MMP 游戏来说，让内容创建和编码工作同时进行可以为内容创建赢得更多的时间。

开发人员可以创建、购买或是下载那些能够独立于游戏，并且能够对内容库进行操作的工具。它们在开发早期尤为有用，因为那时游戏技术还不够成熟稳定，使用这些工具可以提高游戏设计人员和游戏世界创建人员的产出。即使是在 MMP 游戏运营过程中或是在为游戏创建扩展包时，这些工具也非常有价值。

4. 总结

从开发人员的角度看，数据驱动系统可以降低 MMP 游戏的开发、维护和扩展成本——这些都可以让发行人更加高兴。从玩家的角度看，数据驱动系统可以减少 MMP 游戏中的错误、增添更多的内容。这使得玩家更愿意参与游戏并为此支付订阅费用。

5.3.2 在 MMP 游戏中使用数据驱动系统

数据驱动系统在 MMP 游戏中最主要的应用领域就是，定义各种游戏规则。MMP 游戏中的大多数游戏规则最终都可以被简化为，对游戏中不同类型的对象和行为进行定义。通过代码可以实现这些规则，但是大多数规则（如果不是所有的话）都可以由那些定义不同类型游戏对象的数据来驱动。下面这些例子就是可以由数据来驱动的游戏对象：

- 生物；
- 武器；
- 装备；
- 飞船；
- 行星；
- 地形类型；
- 角色技能；
- 角色类别；
- 角色魔法；
- 游戏世界中没有生命的对象（譬如说，树、石头、建筑物）。

举一个例子，思考一下怎样在一个基于幻想的 MMP 游戏中创建成千上万种不同类型的怪物。如果不使用数据驱动架构，程序员必须使用某种方法，把所有这些不同类型的怪物，以及它们所有的特征都嵌入到代码里，每次对这些数据进行微小的改变都需要重新编译代码。把这些数据完全嵌入代码的坏由此可见。尤其是当他们还需要用同样的方法去处理很多其他类型的游戏对象（譬如说武器、防具、魔法和种族）时，这种方法就更不实用了。

数据驱动系统使得游戏规则（对于不同类型游戏对象的定义）可以独立于游戏的源代码。程序员可以把这些规则保存在数据源中，从而使对它们的管理更为方便。

5.3.3 不同类型的数据源

使用很多不同类型的介质，可以保存数据驱动系统中用到的数据。每种类型的数据源都具有不同的特征和复杂度。那些驱动 MMP 游戏系统的数据可以保存在下面介绍的一些方式中。

1. 文本文件

从某些方面来说，文本文件是最简单的数据源。虽然开发人员也需要编写代码来分析文本文件并把它转换为数据，但是文本文件的开销很小并且易于处理。对大量的文本文件进行

管理会很麻烦，但是可以借助于像制表软件（spreadsheet）这样的工具。

文本文件没有强制的内部完整性和结构，任何对语法、结构以及规则间关系的验证都必须由程序员编写代码来处理，而其他的数据源则可以更好地处理这些问题。表 5-21 是一个以制表符分割的文本，它可以从制表软件中导出。这个文件里保存了不同类型的剑的数据。

表 5-21　　从制表软件导出的关于剑的数据

类型	名称	最小攻击	最大攻击	重量	需要的手	图标
CLongSword	长剑	1	8	20	1	sword4.bmp
CTwoHandedSword	双手剑	2	12	30	2	sword5.bmp

2. XML

XML（eXtensible Markup Language，可扩展标记语言）是用来定义数据的标准语言。可以使用现有的软件库来对其进行分析，这样就不必为读入数据文件编写一个分析程序了。但是，相对于其他方法来说，XML 会生成很大的数据文件，因此非常冗长。相对于其他数据源来说，XML 数据也更不容易读懂。

但是 XML 非常灵活，它可以很好地表示那些没有结构或是没有关系的数据。XML 数据的语法和结构验证是由进行导入和分析工作的库来完成的。

对于把 XML 用作数据格式的方法来说，最大的缺陷就是没有良好的支持工具。开发人员在使用 XML 来驱动游戏系统时，通常必须创建定制的工具来对游戏数据进行操纵并且把它们导出为 XML 格式。

下面的 XML 文件定义了不同类型的剑。各种类型的剑的属性并不完全一致。XML 允许不同的实例具有不同的属性，这提供了更大的灵活性。

```
<object name="LongSword">
   <property name="minDamage" value=1>
   <property name="maxDamage" value=8>
   <property name="weight"    value=20>
   <property name="hands"     value=1>
   <property name="icon"      value="sword4.bmp">
   <property name="damageMod" value=1>
</object>

<object name="2HandedSword">
   <property name="minDamage" value=2>
   <property name="maxDamage" value=12>
   <property name="weight"    value=30>
   <property name="hands"     value=2>
   <property name="icon"      value="sword5.bmp">
   <property name="hitMod"    value=2>
</object>
```

3. 脚本语言

像 Python 和 TCL 这样的脚本语言也可以作为数据源使用。当游戏引擎也使用这些脚本语言来保存数据时，这种方式更为有用。因为程序员不再需要编写一个数据分析器，并且可以很方便地把它们集成到游戏系统中去。与 XML 类似，脚本语言可以很好地表示复杂的数据。为 MMP 游戏选择脚本语言是一个非常复杂的问题，并且已经超出了本文的范围，可以选择 Python、TCL、Lua、Java 或是开发人员编写的定制语言。

开发人员编写的工具可以很方便地访问保存在脚本语言中的数据，甚至可以使用脚本编写一些非常简单的工具，或是使用命令行界面来对它们进行处理。本文在下面的例子中使用 Python 来定义不同类型的剑。数据中的变长列表表示了可以装备武器的位置集合。因为需要使用子表（subtable）或是表之间的关系，所以把这种类型的数据保存在其他类型的数据源中可能会非常复杂。

```
InitialSwordTypes = [
   ["longsword", 1,  8, 20, 1, "sword4.bmp", \
      [LEFTHAND, RIGHTHAND, 2HAND]],
   ["twohandedsword", 2, 12, 30, 2, "sword5.bmp", [2HAND]]
]
```

4. 关系数据库

关系数据库也可以被用作数据源，与那些更为简单的技术相比，它具有独特的优缺点。关系数据库通常具有标准接口，因此可以使用标准的组件来创建与之交互的工具。它们可以有效地处理对不同数据源的并发访问，并且易于备份。它们还可以对所包含的数据进行查询或生成报表，不仅如此，它们还具有很高的可伸缩性，通过扩展可以处理海量的数据。

然而，关系数据库的安装和维护非常复杂。而且，为关系数据库进行开发会带来大量的问题，譬如说需要编写特定于厂商的代码、需要使用额外的库并且（有时）需要昂贵的使用许可。关系数据库会为所存储的数据带来一些限制，有时，可以用于一个厂商的数据库的数据和代码可能不能使用在另一个厂商的产品上，这会导致厂商依赖（vendor lock-in）。尤其是在 MMP 游戏的开发过程中，开发人员需要对数据库中的数据模型进行改进来处理游戏规则。想想在整个开发过程中会对数据进行多少改变，就可以知道这样做不仅代价很大，还非常耗时。

可以使用很多非常成熟和稳定的工具对数据库中的数据进行查看和操纵，甚至可以使用工具来定义和管理数据模型以及进行日常管理。然而，这些工具不是非常复杂就是非常昂贵，有的还依赖于某些特定的数据库厂商，这进一步加重了厂商依赖问题。

下面的 SQL 代码可以创建一个保存不同类型的剑的表，并且在创建的表中插入数据。

```
CREATE TABLE swordtypes
(
   sword_id   int         PRIMARY KEY,
   name       varchar(32) NOT NULL,
   mindamage  int         NOT NULL,
   maxdamage  int         NOT NULL,
```

```
    weight      int          NOT NULL,
    hands       int          NOT NULL,
    icon        varchar(32) NOT NULL,
);

INSERT INTO swordtypes VALUES (1001, "longsword",1, 8, 20, 1, "sword4.bmp");
INSERT INTO swordtypes VALUES (1002, "twohandedsword", 2, 12, 30, 2, "sword5.bmp");
```

从数据库中取出这些数据，并且把它放到游戏里的代码相当的复杂，并且还会与特定的平台/厂商有关。然而，还是有很多程序库和工具集可以帮助我们与关系数据库进行交互。

5.3.4 由数据驱动的游戏架构的类型

除了 MMP 游戏以外，还有很多其他类型的游戏架构也可以使用数据驱动。这一节介绍了 3 种不同的游戏架构，它们可以被用在 MMP 游戏中的数据驱动系统中，它们是以下三种战游戏架构：

- 生成类（Generated classes）；
- 动态属性（Dynamic properties）；
- 类别对象（Category objects）。

1. 生成类

在这个架构中，程序员使用数据来为每种类型的游戏对象生成源代码，这样就可以为每种类型的游戏对象生成一个相应程序设计语言中的类。

这个系统的核心被实现为一个模块，它可以从一组数据中生成类定义，这些类定义使用的是游戏本身所使用的编程语言。编译过程将分为两个阶段，首先从数据生成类文件，然后把这些类文件和通常的源代码放在一起编译。我们必须在对其余代码进行编译之前生成这些类，这样在编译那些非生成的代码时它们所需要的头文件都已经存在了。

用数据来驱动这个系统需要把数据从相应的数据源载入，然后生成系统所需要的类代码。这个方法的好处如下。

- 高效，所有的数据都在类中，而不是在实例中。
- 高效，不需要在运行时查找游戏规则数据。
- 高效，不需要载入，因为所有的数据都在类属性中。
- 可以使用程序设计语言中的类来标识游戏对象的类型。
- 一致的代码，因为它们从相同的数据源生成。
- 可以改变每种类型的对象的每个实例的属性。

缺点如下。

- 代码量很大（可执行文件需要的内存很大）。
- 因为所有的游戏规则都被编译在可执行文件中，要仅仅载入一部分将会很复杂。
- 编译过程被分成了两个阶段。
- 任何微小的修改都需要重新编译代码。

- 很难在运行时进行改变。
- 游戏对象的类型只能通过与之对应的程序设计语言中的类来标识。
- 生成的代码可能很难读懂。

下面的例子中是一个生成的 C++头文件，它定义了不同类型的剑。这些类的不同实例共享同样的静态数据成员（译者注：并不能在类的声明中直接初始化 static const string 成员变量）。

```
class CLongSword : public CItem
{
    static const string   mLabel = "long sword";
    static const int      mMinDamage = 1;
    static const int      mMaxDamage = 6;
    static const int      mWeight = 20;
    static const int      mHands = 1;
    static const string   mIcon = "sword4.bmp"

    CLongSword(void);
    virtual ~CLongSword();
};

class CTwoHandedSword : public CItem
{
    static const string   mLabel = "2-handed sword";
    static const int      mMinDamage = 2;
    static const int      mMaxDamage = 12;
    static const int      mWeight = 30;
    static const int      mHands = 2;
    static const string   mIcon = "sword5.bmp"

    CTwoHandedSword(void);
    virtual ~CTwoHandedSword();
};
```

2. 动态属性

在这个架构中，游戏对象具有一组完全由数据驱动的动态属性。游戏对象的类型完全独立于程序设计语言中的对象模型。

这个系统作为一个模块实现。游戏对象具有一组动态的属性，每个属性都有一个值。在这个系统中，只有一个属于程序设计语言层次的类。不同类型的游戏对象都是这个类的实例，只不过它们具有不同的属性。通常，一个名为“类”或“类型”的属性会用来标识这个游戏对象的类型。通常同样“类型”的实例具有同样的属性，但是在具体实现时可能并不强制这点。

通过拷贝已知的“原型”对象，可以创建不同类型的对象实例。这些拷贝的属性与模版对象的属性相同。使用这个机制也可以实现比较简单的继承。

使用数据来驱动这个属性系统意味着我们需要从相应的数据源载入数据，然后创建游戏对象的实例并为它们生成属性。这个方法的优点如下。

- 类的数量很少（只有一个!）。

- 非常灵活，每个实例都可以有所不同。
- 允许动态地创建对象类型。
- 允许在运行时动态修改对象属性。

缺点如下。

- 内存消耗很大：所有的数据都保存在实例中，而不是类中。
- 运行时开销很大，因为每个属性都保存在数据成员中。
- 很难对一个类型的所有实例进行改变，因为并不存在真正的类型。
- 同一个类型的对象并不总是相同，并且可能不具有一致的行为。
- 很难确认某个游戏对象的类型。
- 对不同类型的属性（int、string、float）进行处理可能很复杂。
- 需要在载入时花费一定时间来创建所有的对象。

下面的代码创建了剑的原型对象，它们具有不同的属性值。在游戏中创建这些类型的真实实例时，可以从这些原型对象拷贝。

```
GameObject longswordTemplate = new GameObject();
longswordTemplate.addProperty("minDamage", 1);
longswordTemplate.addProperty("maxDamage", 8);
longswordTemplate.addProperty("weight", 20);
longswordTemplate.addProperty("hands", 1);
longswordTemplate.addProperty("icon", "sword4.bmp")

GameObject twohandedswordTemplate = new GameObject();
twohandedswordTemplate.addProperty("minDamage", 2);
twohandedswordTemplate.addProperty("maxDamage", 12);
twohandedswordTemplate.addProperty("weight", 30);
twohandedswordTemplate.addProperty("hands", 2);
twohandedswordTemplate.addProperty("icon", "sword5.bmp")
```

注意这些属性的值可以是整数或是字符串，这一特性会导致实现变得非常复杂。

3. 类别对象

在这种架构中，游戏对象的类型进一步被分为不同的类别，并且每个广义类别的类都被定义，这些类别对象的变化则由数据驱动。每种类型的游戏对象都有一个“类别对象”的实例与之对应，一整套“类别类”定义了不同类型游戏对象的类别。这是上述两种实现的折中。

这个系统作为一个模块实现，它使用一个表示类别的类体系，其中每个类表示了不同类型游戏对象的广义类别，这些游戏对象具有不同的行为或是数据需求。这些类由“手工”编写而成而不是由代码生成。与此平行的是另一个由类别实例类（category instance classe）组成的类体系，每一个类别实例类都表示了特定类别游戏对象的实例。每个类别类都有一个与之对应的类别实例类。每个类别实例类都有一个类别类属性，在这个属性中保存了该类别特有的数据。这个系统的优点如下。

- 类的数量较少：每个类别一个，而不是每种类型的游戏对象一个。
- 可以在运行时创建类别类（category class）。
- 内存需求比较小，数据仍然保存在类中，而不是实例中。

- 可以使用类别对象来标识游戏对象的类型。
- 可以通过修改类别对象的实例来修改一个类别的所有实例。

缺点如下。

- 它需要两个平行的类体系，一个是类别类型，另一个是实例类型。
- 有些数据需要通过类别实例来访问，而不是对象实例。
- 需要在载入时创建类别对象。

下面的例子中使用 C++为不同类型的剑定义了一个类别对象，并且定义了与这些剑的实例相对应的游戏对象类。

```
class CSwordCategory : CItemType
{
   CSwordCategory(string label,
      int minDamage,
      int maxDamage,
      int weight,
      int hands,
      string icon ) :
         mLabel(label),
         mMinDamage(minDamage),
         mMaxDamage(maxDamage),
         mWeight(weight),
         mHands(hands),
         mIcon(icon) {}

      virtual CSwordCategory();

      const string mLabel;
      const int    mMinDamage;
      const int    mMaxDamage;
      const int    mWeight;
      const int    mHands;
      const string mIcon;
};

class CSword : CItem
{
      CSword(category) : mCategory(category) {};
      Virtual ~CSword();

      const CSwordCategory &mCategory;
};

//Creating Category instance objects
CSwordCategory longsword = CSwordCategory(
"long sword",1,8,20,1,"sword4.bmp");
CSwordCategory twohandedsword = CSwordCategory(
"two handed sword",2,12,30,1,"sword5.bmp");
```

下面的代码在 Python 中进行同样的定义。

```
class SwordCategory(ItemCategory):
   def __init__(self, label, minDamage, maxDamage, weight, /&/
      hands, icon):
         self.label = label
         self.minDamage = minDamage
         self.maxDamage = maxDamage
         self.weight = weight
         self.hands = hands
         self.icon = icon

class Sword(Item):
   def __init__(self, category):
      self.category = category

longsword = SwordCategory("long sword",1,8,20,1, "sword4.bmp")
twohandedsword = SwordCategory("two handed sword",2,12,30,2, "sword5.bmp")
```

使用 Python 例子中的代码，就可以用数据来创建这些类别类的实例。下面的例子中使用了 Python 的 apply 关键字来为每一行数据创建一个剑类别（SwordCategory）实例。

```
InitialSwordTypes = [
   ["longsword", 1,  8, 20, 1, "sword4.bmp"],
   ["twohandedsword", 2, 12, 30, 2, "sword5.bmp"]
   ]
swordCategories = {}
for swordData in InitialSwordTypes:
   newSwordCategory = apply(SwordCategory, swordData)
   swordCategories[newSwordCategory.label] = newSwordCategory
```

5.3.5　总结

上面这些一般化的架构并不一定需要独立使用。可以把这些方法或是它们的一部分组合在一起来创建可以满足特定需求的系统。更为高级的组合包括：具有动态属性的生成类，在运行时把多个类别对象组合成一个更复杂的对象，以及使用数据来生成类别类。

通过使用不同类型的数据源、架构以及它们的各种形式，开发人员在为 MMP 游戏实现游戏系统时具有非常广泛的选择。游戏开发人员必须好好地对它们进行选择，因为在整个游戏的生命周期中，他们都要和这一选择打交道。

5.3.6　参考文献

[Fowler99] Fowler, Martin, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Rabin00] Rabin, Steve, "The Magic of Data-Driven Design," *Game Programming Gems*, Charles River Media, 2000.

5.4 使用数据库来管理游戏状态数据

Christian Lange, Origin Systems, Inc.
clange@origin.ea.com

使用数据库来管理游戏状态会给我们带来一些独特的挑战。对于那些致力于对 MMP 游戏在持久状态世界（Persistent State World, PSW）这一方面进行研究的人们来说，一些有用的实践技巧可以帮助他们进入这一广泛领域。

随着 MMP 游戏对越来越多的玩家具有吸引力，也随着大型动态游戏世界变得越来越大，游戏开发人员必须为游戏生成的海量数据做好准备。游戏很可能必须支持数以千计的玩家和数以百万计的游戏对象。这些对象中的哪一部分需要保存，以及它们应该怎样持久化只是游戏开发人员需要在最开始就解决的问题之一。要对这些海量的数据进行良好的管理，就必须在 MMP 游戏开发过程中进行持续地计划。本文将对一些构建 MMP 游戏数据库的常见方法进行介绍和比较。

5.4.1 模型（schema）设计

数据库模型是对数据表和这些数据表之间关系的概括性表示。从概念上说，数据模型就是创建数据库时游戏设计人员作为依据的蓝图。图 5-1 是一个小型数据库的模型。

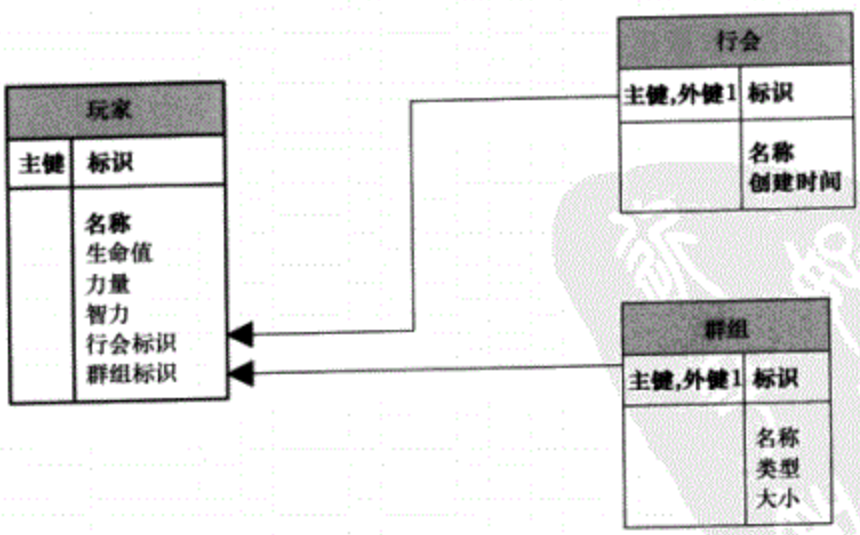


图 5-1 数据模型实例

群组标识）存在于群组表中，就需要使用引用约束了。在这里，本文会同时对这两个概念进行介绍，因为它们之间有着密切的联系。在前面的例子中，玩家数据表包含了群组标识和行会标识这两列。为了确保这两列具有正确的数据，必须对这两列加以约束，并要求玩家数据表中的任何群组或是行会标识都存在于相应的数据表中。也就是说，所有用到的群组标识必须存在于群组数据表中；行会标识必须存在于行会数据表中。要保持数据的完整性，必须使用引用约束，而正规化可以大大地降低所需引用约束的数量和复杂度。正规化的目标就是让数据表中的数据只和这个表的键相关。在玩家数据表中，除了那些受到引用约束限制的列以外，没有任何数据属于其他任何数据表。因此，玩家数据表中不会包含群组名称或是行会的创建日期，这对于简化那些维护数据所必须的更新、删除和插入操作来说是非常重要的。

一个没有良好正规化的数据库会表现出下面这些症状。数据表的大小（列数）也许会变得很大。如果数据被冗余地保存在一个或者多个数据表中，这些表的大小就会增加。如果不对行会名称和群组大小进行正规化，系统就可能会把所有的列都放在玩家数据表中，这会使玩家数据表变大。把行会和群组数据表中的数据放在玩家数据表中会增加对这些信息进行维护所需的工作量。每次群组大小改变了，工作人员都还必须对玩家数据表进行更新以确保这些数据保持一致。引用依赖可以确保群组标识的有效性，但是应用程序必须确保两个表中的群组大小是一致的。这很容易引发问题，并且需要进行更多的数据库事务（transaction）才能保证数据的完整性。这样一来，应用程序必须对数据完整性负责，而不是让数据库系统来负责。

然而，过多的正规化有时也会导致数据库查询性能下降。当数据被正规化后，有些查询需要从多个表中获取数据，这样的查询会比较慢，因为这些数据并不是保存在同一个地方。数据库系统会根据标识列对这些数据表进行一次连接操作，以把所有相关信息组合在一起。在玩家、群组和行会的例子中，数据库系统会根据相应的标识，把行会和群组数据表与玩家数据表连接起来。相对于从单个表中获取数据来说，多表连接操作要慢一点。在这种情况下，可能需要小心地选择数据模型中的一部分进行反向正规化（denormalize）。由于通常情况下正规化的好处更多，因此仅当工作人员必须在一个特定的查询操作上获得高性能时，才可以使用反向正规化。

5.4.2 数据

当数据被放入数据库中时，数据库系统提供了很高的灵活性。不同厂商的数据库系统提供的数据类型区别很大，不过它们还是会具有很多共性。下面是那些在 MMP 游戏数据库上开始工作时常用（或是常常被误用）的技术。

1. 大型二进制对象（Binary Large Object, BLOB）

大型二进制对象中可以保存任何数据。它们可以是一张简单的位图，也可以是玩家记录的二进制表示。这类似于在 C++中把玩家数据用二进制方式存储在一个文件中。这些 BLOB 使用起来非常方便：通常在一个数据表中只需要一个 BLOB 列就可以把所有数据都保存在里面。BLOB 还可以使用专有格式来保存数据。在一些数据库系统中，BLOB 的尺寸常常可以

增长到很大。随着玩家数据的不断增长或是加入更多的属性，开发人员不需要对数据库进行任何变化，因为可以对 BLOB 的尺寸进行任意增长来容纳这些改变。

BLOB 也有不少缺点。在 BLOB 上不能使用引用依赖，这意味着必须由应用程序来确保数据库的完整性。维护数据完整性应该是数据库系统的工作，而 BLOB 妨碍了数据库系统执行这一功能。开发人员不能为 BLOB 创建索引，因此很难对 BLOB 数据进行查询。如果查询需要使用索引，就必须使用额外的列来建立索引。在 MMP 游戏中使用 BLOB 最大的缺点在于无法从游戏外部对数据进行挖掘，所以不能在游戏外对它们进行检查。一方面，数据的正确性无法保证；另一方面，这些数据表示的是什么都不清楚。

2. 数据库中的数据库（Database in Database）

对于 MMP 游戏数据库来说，这是一项非常诱人的技术。它和 BLOB 很像，惟一的区别在于它使用一个字符串来表示数据。这项技术的关键在于，它把包含了多个信息的数据放在一列中。这种做法很常见，譬如说，可以使用一个 4 字节字符串的第一个字节来表示玩家头发的颜色，第二个字节表示他们的佩剑，第 3 个字节则是它们背囊中物品的数量，依此类推。这个方法很灵活，这样，设计人员可以把多个信息压缩在较小的空间里，并且不需要在数据表中添加列。

它的缺点在于数据库不再能够维护数据的完整性。同样，使用了这个方法后也不能再使用引用约束了。要对这样的数据列进行索引会很难（几乎不可能）。数据库也无法通过每个字节所表示的独特含义来对字符串的内部结构进行验证。使用了这个技术就好像每一列中都有一个数据库一样，这可能是一个诱人的方法，但是它把维护数据库完整性的任务从数据库中转移到了应用程序中。

3. 多功能列（multipurpose column）

定义多功能列是指这样一种技术：通过创建一个通用列，在某个记录中保存某种类型的数据，并且在另一个记录中保存另一种类型的数据。譬如说，设计人员可以为某个列取一个通用的名字，例如“数据槽 1”，然后在一个记录中用它来保存生命值，在另一个记录中用它来保存对象的位置。这种方法非常灵活，它可以减少数据表中列的数量。为不同类型的对象添加新记录也非常方便，因为它们只不过是在重用现存的数据表结构。这项技术也可以简化数据访问，因为每一列都具有同样的数据类型，并且列的总数也不会改变。

这种做法的缺点是数据库系统无法使用引用约束。譬如说，如果一个记录使用某个列来保存群组标识，数据库就不能验证这个标识的有效性。无法在这样的列（在这里，就是包含了群组标识的列）上使用引用约束，因为其他记录会用这一列来保存其他类型的数据。也很难为多功能列建立正确的索引。这些列中的数据可能表示了各种无关的数据。即使可以为它建立索引，也很难对这些索引的性能进行预测。这个技术也把维护数据完整性的任务从数据库系统移到对数据进行访问的应用程序中。因为不同记录中数据槽 1 的数据类型也不同，因此数据库系统不知道这些数据的类型，只有应用程序才知道这些信息。同样，当使用客户支持程序或是报表生成器来从外部查看这些数据时，也会遇到类似的问题。如果数据槽 1 中的数据是 89，那么它表示的究竟是生命值、魔法防御还是体力并不清楚。

5.4.3 注意事项

直到现在，本文所讨论的问题都着重于一些特定的数据库系统细节。把数据库系统作为一个整体来考虑也非常重要。除了数据库系统中的数据模型以外，游戏开发人员还必须对一些其他的因素有所了解。有不少内部或是外部因素会影响数据库系统的性能。

1. 性能

前面的一些小节已经对性能有所提及，但都不是以整个数据库系统为背景的。对于数据模型的整体设计来说，数据表过大；大量地使用连接或者不使用索引等都是没有对数据库进行良好计划的常见结果。通常数据库系统并不能对那些具有成百上千个列的大型数据表进行良好的管理。对这些大型数据表进行维护不仅非常耗时，而且在发生严重错误时也很难对数据库进行备份和恢复。大规模的连接操作（譬如说，必须对 10 个表进行连接才能获取所需信息）会对性能带来很大的影响。

对于像 MMP 游戏这样性能至关重要的系统来说，应该避免对两个或 3 个以上的表进行连接。要知道即使是那些小规模的连接操作如果过于频繁的话也会影响系统的性能。通常我们会忽视对数据进行的索引是否适当。索引可以极大地改善数据库系统的查询性能。然而，它们会降低其他操作的速度，譬如说插入和更新操作。开发人员必须为此做出平衡。索引太少，查询就会变慢；索引太多，插入和更新就会变慢。因此，需要根据特定的应用程序来决定哪个更为重要。通常，MMP 游戏主要进行查询和更新操作，而删除操作通常只需要在某些特定的时刻进行，因此最好在维护时而不是在系统繁忙时进行。要对一个数据库进行调整，使得它可以以最优化的方式进行查询、更新和删除操作不仅代价很大，而且非常困难。一个常见的实现方法是，在删除时只对记录做一个标记，然后在游戏停止运行或是进行维护时对这些记录进行检查并且真正地删除它们。删除操作代价很大。避免或是延迟执行这些删除操作对于数据库系统的整体性能会有非常大的好处。通常，一个典型 MMP 游戏的使用强度会在某天或某周内以一个可以预测的周期升高或降低。那些非繁忙时段是进行数据库维护以及删除那些需要被清除的旧记录的最佳时间。

2. 网络

在开发大型数据库应用程序时，对网络带宽的利用也是必须考虑的一个关键问题。对更新和查询请求中应该包含哪些数据作出定义是非常重要的。

数据库系统通常具有很高的灵活性，它允许应用程序发送或接收大量的数据。数据库在这方面的开放性可能会带来问题，因为游戏的运行并不需要每次都对一个对象的所有属性进行更新。如果一个对象（譬如说一个玩家角色）向前移动了几步，应用程序不需要发送这个角色的生命值、背囊信息或是方向，只要这些属性的值在上次更新之后没有改变过就行了。试图在应用程序中把那些独立的更新打破为更小的请求是明智的。这种做法同样适用于非常大的查询。与其直接使用一个很大的请求，还不如看能否把应用程序修改得只请求那些必须的数据，并且延迟对其他数据的请求。这不仅可以降低对带宽的需求，也可以从整体上提高数据库系统的性能。

3. 事务负载

事务（譬如说查询）会为数据库系统带来很大的负担，因为它必须对典型的 MMP 游戏所创建的几百万个记录进行检索。如果不使用索引的话，这些查询对数据库系统的影响会非常大。如果没有合适的索引，数据库系统可能需要把数据表中的每个记录都检查一遍。有一个简单的方法可以对查询进行检查：看看作用于这个查询上的条件。在前面例子中，查询试图根据特定的情况来获得玩家的信息，获取生命值大于 80，属于某个行会但是不属于任何群组的所有玩家。当检查这个查询的条件时，就应该看看这些条件是否被索引了。系统需要在生命值、行会和群组上建立索引，也可能是使用一个包含这 3 列的索引。如果不存在这样的索引，就应该仔细检查一下这个查询是否存在性能问题。应该用这种方式对所有向数据库系统发出的请求进行检查。减少每个请求所需的时间就可以减少整个数据库系统整体的事务负载。

4. 动态 SQL（Dynamic SQL）

还有一种请求也会大大地增加数据库系统的事务负载并且降低其性能，这就是动态 SQL[Date95]语句。这是一种功能非常强大的技术，有了它，就可以以字符串的形式向数据库系统发送一个即时生成的请求。通常情况下，数据库系统并不能预知这个请求。与那些经过编译的或是保存在数据库系统内部的请求不同，这种请求是运行时动态生成的。这意味着这个请求无法获得数据库系统对数据库请求进行准备（prepare）、预先优化（preoptimize）和预先缓存（precache）所带来的好处。当一个数据库请求在编译时，被创建或是保存在数据库系统中时，数据库系统会对其进行预先分析。这样一来，数据库系统就知道应该怎样做才能以最快的速度对这个请求进行处理。而对于动态请求来说，数据库系统必须对其进行分析并且确定完成这个请求的最佳方式，这是向数据库发送请求最慢的方式了。游戏开发人员必须对任何用到这类功能的需求仔细斟酌，这里面很可能存在问题，并且会严重地影响性能。

5.4.4　其他方法

MMP 游戏的数据库可以变得很大。即使我们遵照所有推荐方法对数据进行处理和管理，系统仍然可能过载甚至崩溃。这一小节不仅向开发人员介绍了一些可以使用的其他方法，还提出了一些在开发 MMP 游戏数据库时需要考虑的额外问题。

1. 数据缓存

这里所说的数据缓存系统是一个外部系统，它和存在于数据库系统内部的缓存不同。数据缓存是在应用程序和数据库之间创建的一个额外进程（如图 5-2）。所有数据库请求都会先经过这个数据缓存，它可能会把这些请求继续传递给数据库系统。通过使用这个方案，所有请求数据都被缓存在这个进程的内存里。如果将来某个请求需要同样的信息，这些信息会直接从这个进程的内存中返回，而不是从数据库系统中。这可以显著地降低数据库系统的负载。对更新请求的缓存则与上述方式相反：游戏开发人员会修改缓存中的数据，而延迟对数据库的更新。使用缓存会带来一些独特的挑战；然而，和建立数据缓存进程所需要的额外工作相比，在性能上获得的好处要大得多。

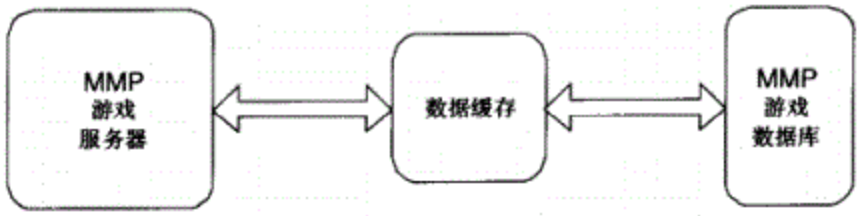


图 5-2 数据缓存进程

2. 架构

在设计 MMP 游戏时，必须把数据库请求架构设计得能够对任何请求的发生进行控制，这样做是为了防止 MMP 用户能够直接影响数据库的性能。换句话说，如果用户每次按下一个按键都可以产生一个对数据库的请求，那就可能会导致非常严重的问题。这实际上是让用户能够对数据库进行“拒绝服务（denial-of-service）”攻击。想象一下，成千上万的用户，每个人都能够不停地按下那个按键来向数据库请求数据。这样大规模的请求轻易就可以使数据库系统持续满负荷运行。在 MMP 游戏系统中，设计人员不应该让用户和数据库有直接联系。必须在它们中间使用某些进程或是服务器来过滤或是延迟这些请求，以使得请求按照一定的频率进行。前面提到的数据缓存系统也有助于缓解这种特定的情况。

3. 维护

随着数据库的持续增长，游戏开发人员必须对它进行周期性的维护。日志文件和备份会变得越来越大，而索引和数据则会有很多碎片。应该定期关闭数据库并且对其进行优化，从而保证它始终以最佳性能运行。如果没有对数据库进行正确地维护，时间一长其性能会大大地降低。如果需要生成大量的报表或是进行很多数据挖掘，可以考虑把数据库复制到另一台电脑上去。开发人员可以在数据库的拷贝上进行这些额外的报表生成工作。这样就不会因为报表、数据挖掘、统计信息搜集等操作降低系统的响应速度了。

4. 连接

与数据库系统的连接是一个很容易被忽视的问题。数据库系统通常都可以同时处理多个连接。要利用这个特性，就可以为数据库创建一个连接池。游戏服务器上可以建立多个数据库连接，而不是只有一个连接。多个独立的线程或是一个线程池可以用来对它们进行管理，每个线程都有其自己的数据库连接。这样做的好处在于，当服务器接收到一个新的请求时，可以把它分派给一个空闲的线程以立即对它进行处理。只要对连接和线程进行正确的平衡，所有请求都不必等待就可以与数据库进行通信。很明显，如果只是用一个连接和一个线程来处理与数据库系统间的通信，结果就会完全相反。要注意的是，这里所指的是静态连接，这些连接在游戏服务器启动时就会被建立，直到游戏服务器关闭时才会断开。由于 MMP 游戏数据库对于事务的要求很高，应该避免使用动态连接。动态连接是指那些在发送请求时才建立的连接，当请求结束后，它们就会断开。打开和关闭数据库连接的开销很大，通常是一个耗时的操作。因为数据库系统会直接影响到 MMP 游戏的性能（譬如说等待数据），系统架构师可以考虑把数据库服务器和那些需要发送数据库请求的游戏服务器直接连接起来。这样数

据库服务器就不会受到外界网络的影响。这样系统架构师就能够对游戏服务器和数据库之间的带宽配置进行优化。

5.4.5　总结

MMP 游戏数据库管理，并不存在一个完美的方案，只有一些经过验证的策略和方法可以遵循。每个应用程序的要求都是不同的，但是这里所提到的指导方针适用于所有应用程序。对此进行学习的最佳方法就是尝试这里所介绍的策略，并且使用所选择的数据库系统进行实验。从而彻底精通这个数据库系统。每个数据库系统都具有不同的优势和弱点，应该学会怎样利用它们。在编程时，程序员通常会避免编写任何针对特定系统的代码，而尽量让代码一般化、接口化。而对于数据库编程来说，尤其是那些在 MMP 游戏中所使用的对事务处理要求很高的系统来说，这恰恰相反。为了使系统的性能达到极致，游戏开发人员必须对其进行深入的了解并且利用这一数据库系统所提供的任何优势。

读者应该使用本文所介绍的技术、策略和教训。无论在开发 MMP 游戏数据库时面临怎样的重任，这些都会是一个良好的开端。本文介绍了很多有吸引力的技术，希望读者有机会用到它们时，可以重读这篇文章来理解这项技术的优缺点。虽然本文并没有把所有的优缺点都列出来，但是那些在不同 MMP 数据库中经常出现的问题都已经在本文中介绍了。

对数据库中的游戏状态数据进行管理实际上可以被归结为：理解需求是什么；需要存储哪些数据；数据量多大；怎样进行存储以及在以上这些步骤中都使用合适的策略和技术。

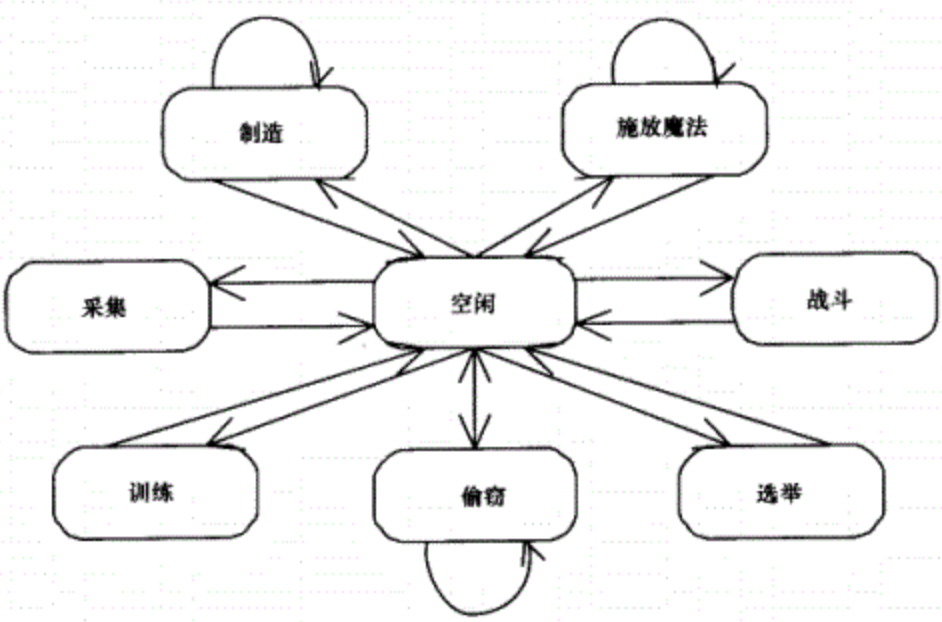
5.4.6　参考文献

[Date95] Date, C. J., *An Introduction to Database Systems*, Addison-Wesley, 1995.

[Ensor97] Ensor, Dave and Ian Stevenson, *Oracle Design*, O-Reilly, 1997.

第 6 章

游戏系统



6.1 从原料到成品：社会经济中的职业生涯

Artie Rogers, Inevitable Entertainment
awrogers@texas.net

在任何社会中，手工制造物品都是表达个性的重要方式。制造物品使人们投入到创造性过程中，并获得切实的回报。在 MMP 游戏中，一个成熟的手工业制造系统可以给玩家带来回报，像在现实生活中一样。游戏中的手工业制造系统不仅可以让玩家进行创造，表达自己，从而体现个性以获得满足；还让玩家不必通过战斗来升级，只需安分地进行制造就可以获得成功。成熟的手工业制造系统提供了另一种成功的途径，这使 MMP 游戏的吸引力更为强大，拓展了现象群体。

除了战斗以外，制造系统提供了另一种可行的游戏形式，这可以延长玩家在游戏中停留的时间。它使得一个全新的致力于创造和销售物品的玩家社团得以形成。不仅如此，那些在设计时就考虑到社会发展的制造系统有助于培养一个丰富的在线社会经济。我们把在线社会经济定义为一个具有紧密联系的制造者社团，他们必须依靠彼此才能获得更高的成就。在一个以彼此合作为前提的制造系统中，玩家需要向其他玩家学习怎样制造某个物品，而只有那些专家才能制造出最难制造的物品，同时，他们都需要到市场上进行交易，这样的制造系统可以为构建一个稳固的在线社会做出显著的贡献。

本文强调了一些在设计制造系统时必须考虑的主要目标。首先，我们会讨论在制造系统中引入概率的重要性，以及怎样用它来为制造过程引入一种像玩“老虎机”一样的兴奋感。在考虑加入概率时，必须注意最好是根据结果给予玩家奖励，而不是去惩罚他们。让玩家可以获得超过期望的回报，使得概率的引入成为一种带给玩家积极愉快的经历的元素。

接着将对技能多样性在促进社会经济扩展及扩大社会经济的影响中起到的作用进行讨论。实现技能多样性的一个重要环节是设计一些方法使玩家在试图完成一些高级任务时需要依靠其他玩家的帮助。在制造过程中加入一些随机性，可以创建一个有趣的游戏系统。通过提高游戏中制造专家的多样性以及鼓励他们在制造最好的物品时与其他玩家合作来让游戏经济的影响最大化。

本文将对物品制造过程（即从获取原料到对新制造的物品进行改进）进行全面介绍，并对在构建制造系统的过程中需要考虑的原料和技能方面的一些参数进行了分析，最后就制造在其他社会系统中的作用进行简单的讨论，并且还对制造系统的一些常见问题提出了一些想法。

6.1.1 原料获取和加工

获取原料是制造物品的起点。为了让尽可能多的玩家参与到社会经济中来，我们应该设计一个以技能多样化和社会合作为目的的原料采集系统。采集原料的方式有很多，本文仅对其中的一个模式进行简要讨论。

1. 原料采集系统

每当玩家需要从游戏世界中采集原料时，就必须使用原料采集系统。这可能包括玩家砍树来采集木头、从庄稼中挑选食物或是在石头中挖掘矿石。我们的讨论主要针对“采集原料”这一动作，也就是对原料核心（material core，含有各种游戏资源的对象，例如木头、食物或矿石）进行砍伐、挑选或是挖掘的动作，同时对原料核心的不同质量进行讨论。

采集是指这样一种动作，角色为了获得一定数量原料在原料核心上进行的动作。一些和能力或技能有关的玩家属性会影响采集的结果。

- 采集原料精度：这会影响玩家每次试图采集原料时所获得原料的数量。
- 采集原料能力：这会影响玩家每次试图采集原料时所获得原料的质量。
- 采集原料速度：这会影响玩家每次试图采集原料时所需的时间。

把采集原料的技能按照所需要的时间、质量和数量分为不同的类别，让玩家自由地定制他们的角色，这样就会出现不同类型的原料采集专家。

原料核心是游戏中的对象，它们是原料的来源，并且只有在具有合适技能的玩家对它们做出合适的动作后才会产出原料。根据不同玩家的技能，原料核心可以是可见的，也可以是不可见的。系统会维护一定数量的原料核心对象，它们中的每一个都与特定类型的原料相关。每当一个原料核心对象销毁了，系统会为它创建一个替代品。这样一来，设计人员不仅可以控制在哪里可以找到特定的原料，还可以控制它们的总量。以下三个变量和原料核心相关。

- 核心强度：表示原料核心在销毁前可以经受的原料采集行动的次数。
- 核心质量：表示原料核心所产生原料的质量范围。
- 核心深度：表示原料核心在每次采集原料操作时所产生原料的数量范围。

引入原料采集过程后，除了那些和获取原料有关的基本动作，我们还可以加入其他的技能。在使用采集到的原料制造物品之前，玩家可以使用某些技能对它们进行加工。这些加工会改变原料的质量从而改变产品的最终属性。玩家还可以使用某些高级技能来找到最好的原料核心对象，也就是说找到可以产出高质量或高数量原料的核心。如果在创建制造系统时就把社会合作考虑在内，我们就有很多方法可以使制造系统中进行原料采集的玩家在游戏中的作用并不仅仅局限在上述的这些。

2. 超越期望的乐趣

概率是一个重要的因素，我们应该把它引入制造过程中包括采集在内的每一个阶段。不要让玩家因为概率结果受到惩罚，譬如说采集到一些质量非常低的资源，而是应该用一些超出他们期望的资源来奖励他们。超出某个行动所期望的结果可以为玩家带来惊喜。任何与概率有关的结果总是会在某种程度上与失望相关。为了对这种失望加以限制，需要为

特定的采集或制造行动建立一个底线，一旦玩家完成某个行动，就总是可以获得这个结果，然后在这个标准结果上加入一些奖励。原料采集中，我们可以通过让玩家采集到超出他们技能范围的高质量原料来作为一种概率奖励：有时玩家会高兴地发现采集到了原以为不可能获得的原料。另一种可能就是让采集行动偶尔会得到一些和所采集的原料完全无关的物品，当然这些物品在游戏中也必需有一定的价值。在原料采集中引入概率为玩家带来惊喜，这会让游戏更为有趣。

3. 时间投入和原料产出以及风险和回报

在原料采集上投入的时间和原料的产出存在一个内在的平衡，以使资源的价值和采集它们所耗费的时间相一致。玩家想要采集的资源以及采集的方式会影响采集所需要的时间。譬如说，玩家可能会想采集一种比较廉价的资源，因为采集时间很短，即在保持原料类型和采集时间不变的前提下，以降低原料质量为代价来获得更多的原料。玩家也可以通过降低原料质量来提高采集速度，而不是以更慢的速度获得质量更高的原料。因此，玩家可以在采集某种特定原料时，以原料的质量为代价来获取更快的采集速度。这意味着，投入的时间与所采用的采集方法相关。玩家可以采用灵活多样的方式来采集资源，这样就需要有很多与资源采集相关的技能。

我们将在后续小节中讨论风险和回报之间的平衡关系对资源采集的影响。在设计 MMP 游戏的探险部分时，通常必须考虑风险和回报之间的关系。这一关系涉及到玩家所承担的风险以及他们在克服这些风险后能够得到的回报。这意味着杀死强大的敌人后，往往可以获得更多的战利品。这一关系在原料采集中的重要性常被忽视，然而它却可以用来扩展社会经济的范围和影响力。譬如说，原料采集者可能想要经过一个游戏世界中非常危险的地方去采集最有价值的资源，因此他需要其他玩家的帮助。高回报的诱惑使得他会向那些经验丰富的冒险者寻求保护，从而在这种高风险的行动中能够存活。在设计资源采集技能时游戏设计者考虑到风险和回报之间的关系可以加强原料采集者和其他冒险玩家之间的联系，从而扩展社会经济的影响力。

4. 原料采集实例

假设一个按照上述资源采集系统规则进行资源采集的情景：一个玩家在某个制造专家的委托下去采集那些只有在非常危险的山区才能发现的高质量矿石。这个玩家具有采集矿石的技能，但是缺乏在这个地区生存所需要的战斗技能，也无法对可以产出所需矿石的原料核心进行迅速可靠的定位。虽然他可以自己找到这个原料核心，但是由于这非常危险，因而最好能够迅速地找到它，在采集完所需的矿石后立即离开。玩家可能使用简单技能而不是复杂技能，因为时间非常关键，他宁可为了速度在质量上做出一点牺牲。

于是玩家去城镇寻找那些可以帮助他的其他玩家。通过使用“任务公告栏”（一种简单的电子公告系统，玩家可以用它在任何城镇的中心寻找或发布任务）或是询问一些朋友，他不仅可以找到一群有能力的玩家来护送他，还可以找到一个采集助手来帮他找到原料核心。鉴于群组的等级非常高，时间对于玩家不再那么重要，因此他决定使用复杂技能来获得最高质量的矿石，而那些负责护卫的玩家想要得到的回报则是群组在执行任务时所发现的除了矿石以外的物品。因为他们要采集的矿石质量很高，同时玩家用以进行资源采集的工具质量也

很高，因此在采集过程中发现矿石以外的其他物品的概率很大。几乎可以肯定，玩家在进行采集工作时会发现一些宝石或是其他珍贵矿藏。因为这个玩家并不需要这些额外的物品，他同意让他的帮手获得除了矿石以外的任何附产品。

最后，这个玩家和他的伙伴们见面，一同前往充满怪物的山区去最危险的悬崖上获取这个大陆上最有价值的矿石。如果他们能够活着回来，就都可以获得丰厚的回报。

5. 原料采集小结

对原料的采集和加工是制造过程的起点。通过对采集过程进行分析设计，我们不仅可使玩家群体中有不同种类的技能专家，还可为采集过程加入一些有趣的因素，并可以促使游戏中具有不同背景的玩家共同合作来实现更高的成就。一旦采集到了原料，就可以用它来和制造人员交换物品和服务，从而把原料转换为有用的物品。

6.1.2　社会经济中的合作制造

设计完善的制造系统可以拓展玩家群体，延长玩家的游戏时间，加强玩家之间的社会联系。通过对原料采集、物品的需求度、合理的技能多样化以及丰富的物品制造配方等方面进行仔细考虑，游戏设计人员可以设计增加游戏体验的深度，为游戏建立一个社会经济体系，玩家在该体系下可对装备、货物、原料以及制造服务进行交易。

1. 制造物品

制造是指玩家使用各种工具对一组物品或原料进行某些行动来创建一个物品的过程。为了制造一个物品，玩家必须先获得制造这个物品所需的技能。下一步是找到制造这个物品的配方。物品配方会详细描述玩家制造这个物品所需要的工具、原料或是其他物品；它还可以列出在制造过程中出现问题时玩家所需要的原料。配方也是打开制造某种物品能力的钥匙，因此，即使玩家知道制造某个物品需要些什么，他们仍然需要拥有配方才能制造出该物品。让配方成为必要前提并且不允许在玩家间转让配方，可以让设计人员控制市场中配方的饱和度，即配方是一种由系统控制的物品。

在制造物品时，玩家必须仔细考虑所使用的原料、物品和工具的质量，因为最终产品的属性会受到它们的影响。一旦拥有了制造物品的配方以及所需的资源，玩家就可以进行制造了。制造需要一定的时间，这完全取决于玩家的技能以及所制造的物品。一旦完成了某个物品，玩家可以出售它，也可以想办法改进它。他可以在这个新制造的物品上加入额外的资源或是其他物品，也可以向其他具有改进技能的玩家寻求帮助。这样一来，玩家就可以制造各种不同的实用物品或是装饰品。

2. 时间投入与产品价值

在资源采集中，投入的时间和原料的质量与数量之间存在一个平衡关系，而在制造中，这个平衡则存在于投入的时间与产品的价值之间。对于一个价值较高的物品来说，它不仅需要更长的制造时间，获得物品配方中所需要的原料和工具也需要更多的时间。

正如在真实的市场中一样，物品的稀有程度决定了它的价值。这在在线游戏中尤为明显，

因为在线游戏中的市场较小并且这些因素的影响更为显著。有些物品制造所耗时间很长，其价值直接由它们的功能以及对冒险玩家的用处所决定。然而，有些价值很高的物品仅仅是因为它们很稀有，即使它们在游戏中毫无用处。

我们可以使用很多方式来调节制造某个物品所需的时间。其中之一，就是对创建某种物品所需要的原料或是工具进行限制。系统会控制游戏世界中特定原料在特定时间的总量，这样就可以限制在那一时刻所能制造的物品数量。我们还可以通过技能系统来调整制造时间，把原料采集技能设计为玩家必须掌握一定数量的技能后才能进行所需的采集行动。譬如说，某个特定的物品需要高质量的木头才能制造，并且在制造前还必须对这些木头进行加工。如果所需的技能都是高级技能，那么玩家就需要在一定时间后才能获得采集和加工木头所必需的技能。这样一来，我们可以通过要求玩家必须掌握高级技能才能制造某个物品来推迟这一物品在游戏世界中的出现。有些物品需要一定数量的高级玩家共同合作才能制造。通过为物品制造加入一定的社会复杂度可以延长制造所需要的时间，这同样也会影响它的价值。

3. 原料、工具和物品的作用

物品部件主要有三大类：

- 工具；
- 原料；
- 其他物品。

每种类型的部件都会对制造过程和最终产品产生不同的影响。不同类型的部件具有不同的属性，这些属性决定了它们的质量。用它们制造出来的物品也会继承这些属性，因此这些属性也会影响物品的最终质量。

首先，对工具进行讨论。工具是一种有趣的部件，它是由其他玩家制造出来的，这可以把另一种技能引入社会经济，不仅如此，它还可以为制造过程带来独特的变化。通常，工具质量会影响最终产品的属性。有时它们也能改变与正常制造过程有关的属性。譬如说，某些工具可以缩短制造时间。有一些高质量的工具在使用时可以提高玩家的基本技能，还有一些工具可以为最终产品带来一些新的属性，这些属性在使用其他工具时不会出现。

工具还可以带来一些独特的制造限制，譬如说，对制造物品的时间和地点进行限制。一方面，如果一个工具很大以至于不能从一个地方移动到另一个地方，那么制造人员就必须在特定场所才能制造这个物品。另一方面，如果配方中需要使用特定的原料来制造这个工具，并且由于这个原料很重（或是一些游戏中的其他限制），玩家不能移动它。那么，这些工具只能存在于某个地理区域中，这也会把相应的物品制造限制在这个区域中。在制造系统中引入工具不仅可以对制造过程产生各种常见或罕见的影响，还可以在玩家社团中引入新的制造者类型。

接下来，对原料进行讨论。原料具有很多核心属性，它们会影响最终产品的属性。这些核心属性都有一个基准值，因原料质量的不同而不同。原料质量会对最终产品产生影响，这可以表现在对最终产品属性的影响上，也可以为最终产品增加新特性。具体的结果是由特定的原料和物品决定的。原料是制造物品的基本，制造人员可以通过选择原料来改变最终产品的质量。

最后，对物品进行讨论。高级物品需要其他物品作为部件。和原料一样，作为部件的物品也有一些核心属性，其基准值也受到物品质量的影响，并且这些值也会影响最终产品的质量和属性。通常制造作为部件的物品所需要的技能与制造最终产品所需要的技能不同，这有助于增强对于其他制造玩家的依赖。譬如说，如果一个特定玩家具有从零件创建引擎的技能。另一个玩家精于制造某个引擎的部件，但是他没有把这些部件装配起来的技能。如果他们各自独立地工作，那么所创建的引擎差不多都是普通质量的：一种引擎的装配技术很高，但是却使用普通的零件；另一种引擎使用很好的零件，但是装配技术一般。当这些玩家共同工作时，就可以制造出更好的引擎。这些物品部件的主要目标就是加强不同技能群组之间的依赖，并由此加强整个社会经济。

在加入概率的作用时，要注意，应该在系统中加入一些会随着玩家行动的成果而改变的奖励，而不是让概率事件的结果给玩家带来惩罚。应该试图超越玩家的期望并且为某个有利的概率事件给予玩家奖励。在制造中，概率事件的结果可以是制造出质量更高的物品，也可以是制造出的物品具有一些出乎意料之外的新属性。

4. 对新制造的工具进行改进和修理

对所制造的物品进行后期改进或是修理可以扩展制造过程。玩家可以使用和制造物品相同的方式对它们进行修理或改进，只需要简单地把部件和所需修理或改进的物品组合起来。如果成功的话，这个物品就会被修好或是改进了。物品制造完毕后，它们仍能被处理用以扩大社会经济的范围。

5. 制造小结

通过分析制造系统对游戏可能起到的影响，设计人员可以使系统支持那些围绕着物品的采集和创建而产生的社会组织。我们必须对各种和制造相关的行为（从原料采集到对现有物品的修理）进行仔细考虑以拓展制造系统的范围，从而增加系统所能影响的玩家数量。通过保持物品需求、建立合理的技能分工以及提供大量的物品配方，一个成熟的制造系统不仅可以增加游戏体验的深度，还可以促进社会经济的发展。

6.1.3　物品制造在社会经济中起到的作用

物品制造有助于提高城镇在社会中的重要性并加强 MMP 游戏的吸引力。本文的目的之一，就是把制造作为扩展和加强社会经济的一种方式来进行研究。通过观察玩家在游戏内外所交易的物品数量，验证于 MMP 游戏中存在的社会经济。这适用于各种各样的游戏，无论是那些着重于制造和社交的游戏，还是那些着重于冒险的游戏。对于一个繁荣的在线游戏经济来说，制造业是它的生命源泉，各种具有不同游戏风格的玩家会在一起对物品、原料和制造进行交易和讨论。

制造系统可以从很多方面提高社会经济中团队合作的重要性，这一小节将对其中的几个方面进行探讨。我们先研究怎样可以把一个成熟的物品制造系统和声望系统联系起来，然后研究声望系统对一个制造者的声望所起到的作用。最后，我们还会探讨在一个运行了多年的游戏中保持需求对保持社会经济的稳定所起到的作用。

1. 制造业的社会作用

我们可以把制造系统设计为玩家们需要彼此合作才能制造出最高质量物品的系统。这种对玩家合作的依赖是通过两种不同的方法实现的。一种方法是把制造系统设计为需要很多不同的技能集合；另一种方法前面已经提到了，就是通过调整风险和回报之间的关系以使从事制造业的玩家依赖于那些冒险玩家。

我们必须考虑如果提供多种多样的玩家技能会对社会经济造成哪些影响。除此之外，我们还必须对这一系统进行设计以使一个玩家不能在过多的领域成为专家。同时，在制造一个物品的过程中必须要使用到多种互斥（mutually exclusive）的技能。

在制造系统中，有多个不同的技能组合，包括：

- 采集；
- 工具制造；
- 物品制造；
- 改进；
- 修理。

如果一个专家级玩家希望制造出具有最高质量的最强物品，他必须和其他的专家级玩家合作来制造这个物品。这一合作的回报可能是物品或者服务，这为游戏社会中的专家级玩家创造了交易的可能。

我们必须仔细地考虑这种做法，因为过多地强迫玩家进行合作反而会使他们与社会疏远。那些希望独自进行游戏的玩家可以通过通常的升级系统和游戏模式来使用绝大部分游戏系统。他们能够独立制造出大多数物品而不需要其他玩家的帮助，仅当想要制造高质量的物品以及少数质量极高的物品时合作才是必需的。

2. 制造业、物品和声望

在制造系统中，玩家需要找到所需的专家级玩家以便制造想要的物品。而制造者也需要为其技能做广告以找到那些可以成为他们客户的玩家。对于制造者来说，与其他玩家的交流非常重要，因为他们必须为其技能和产品做广告。如果制造者可以访问一个列表，里面列出了与所制造的物品有关的信息（如使用情况、游戏中存在的总量等），这对他们来说会非常有用。在构建那些对物品的制造过程没有直接影响，但是会对制造业的社会作用起到支持或是扶植作用的游戏系统时，我们也需要对上面这几点加以考虑。

对于一个制造者来说，最需要的莫过于可以通过某种方法来和其他玩家进行交流从而把他们的物品和服务信息提供给感兴趣的玩家。有几种不同的方法可以在游戏中或从游戏外部解决这个问题。例如，创建一个公告栏，使那些达到特定等级的制造者可以在这里发布信息，为他们的技能和产品做广告；也可以使用两个独立的公告栏：一个为需要出售的产品或服务做广告，另一个可以让玩家张贴寻求帮助的请求（这就是前面例子中提到的“任务公告栏”）。在游戏中，玩家不仅可以对这些公告栏应用某些简单的过滤选项（譬如说物品的等级或是发布广告的制造者的经验等级），还可以自动根据当前的位置来对信息进行筛选。用于发布请求的公告栏可根据玩家的技能等级来筛选请求，这样的过滤机制下玩家只能看到在某个特定技能范围内的请求。在一个外部的网页上建立公告栏的一个镜像后，玩家就可以在游戏之外寻

找潜在的商业伙伴。这为玩家寻求和获得帮助提供了一个中心位置，玩家也可以在这里寻找完成特定任务所需要的专家级玩家。

如果玩家可以获得关于所制造的物品的特定信息，这会非常有用。如果玩家能够追踪某个物品的历史并且用它来证明所制造物品的质量，这对这个制造者的声望也有正面影响。譬如说，如果某个玩家所制造的武器在一场大型战役中作出致命一击，制造这把剑的玩家应该可以获得这个信息并且把它传递给其他玩家，这可以提高他在游戏中店铺的声望。制造者最终可以为他们所制造的物品建立一系列相关信息。下面是一些例子。

- 所杀死的怪物总数（可以按照等级、区域和稀有程度进行排序）；
- 在游戏中他所制造的物品的总数；
- 使用这些物品的角色的平均技能等级；
- 使用这些物品的角色的平均声望；
- 当这些物品被再次出售时它们的平均售价；
- 所制造物品的总数；
- 所制造物品的质量；
- 使用某个物品采集到的资源总量。

声望不仅和所制造出物品的质量有关，它还和在声望系统中这个角色的声望有关。声望系统会对角色的行为进行评价，从而得出一个总体评价来告诉我们这个玩家是好是坏。如果制造者希望为某个重视声望的市场所接受，他们可以使用其制造的物品的玩家的平均声望。设计人员也可以让某些物品具有固有的邪恶属性而另一些物品具有固有的正义属性。制造者可以通过制造不同的物品来调整他们的声望。譬如说，制造偷窃工具或是可以伤害其他玩家的陷阱会对制造者的个人声望有负面影响。通过这些方式，制造系统和游戏的声望系统联系在一起，从而进一步提高它在塑造在线社区中的作用。

交流和信息是经济繁荣两个主要基础。给予制造者他们所需要的工具对于建立一个稳健而持久的社会经济来说是非常重要的一步。

3. 保持需求的重要性

保持需求是一个常常被忽视的重要细节，因为这样做会给那些不从事制造的玩家带来麻烦。然而，保持对物品的总体需求是非常关键的。我们必须知道制造者会根据市场需求来制造有人需要的物品，直到没有充足的需要从而没有理由继续制造这个物品为止。在游戏世界的整个生命周期中我们必须一直保持对所有物品都有良好的需求，否则需求会逐渐偏向那些难以制造的物品。如果我们没有注意到这一现象并且任之继续下去的话，那么那些新入行的制造者就没有足够的市场需求来维持他们的职业，因为这个市场已经被前人的劳动成果占据了。为那些新入行的制造者保留一定的市场是非常重要的，否则随着游戏的成熟，这一部分游戏模式将不再可行。

保持物品需求主要有两种方式。第一种就是让物品会损坏，第二种是让玩家的物品保持一个较高的周转率。损耗的作用是不言而喻的，每个物品最终会由于正常的损耗而消失。某个物品使用 X 次后，它就会坏掉。如果我们可以通过某些方式对它们进行修理，就可以降低损耗带来的痛苦，也可以延长物品的使用时间；但是即便可以修理，每个物品的使用时间也应该有个限度。这样一来，通过引入损耗就可以为那些常用物品保持一定的需求。

另一种保持需求的方法是创建特定的系统来确保玩家物品的周转率保持在某个特定水平。特定 MMP 游戏的具体需求决定了哪类系统更为合适。在一个基于冒险的 MMP 游戏中，让玩家在死去后会失去背囊中部分或是全部物品是一个保持需求的有效方法。这两种方法是为制造者维持必要市场的有用工具。如果游戏设计者希望在 MMP 游戏的整个生命周期中玩家都可以把手工业作为一个可行的游戏选择，那么建立这样的系统来确保物品需求总是存在是至关重要的。

6.1.4 总结

制造业是持久在线社会的一个重要组成部分。如果我们可以意识到技能多样性还能够扩大社会经济的影响范围，就会发现它更为重要。通过创建一个稳健的采集系统，我们可以加强冒险玩家和制造玩家之间的联系。通过设计各种通信系统，制造者可以向玩家群体寻求物品或服务，或是向他们提供物品和服务，整个玩家群体就可以认识到并且认可制造者的作用。通过保持对低级物品的需求，即使是一个新的制造者进入这个存在已久的游戏社会时，他们的市场仍然很大。通过创建一个覆盖面非常广泛的制造系统来促进合作，玩家通过 MMP 游戏可以从活跃的社会经济中获得很多好处。

6.2　玩家房屋供给：我的房屋就是你的房屋

Paul D. Sage, NCsoft
psage@ncaustin.com

在2000 年春天，当《创世纪在线》（*Ultima Online*，*UO*）为提供用于建设房屋的土地时，玩家们反应非常强烈；玩家们连续在线等待超过 24 小时只是为了有机会能在游戏世界中拥有自己的房屋。这并不出乎游戏设计者的意料，它进一步让游戏设计者知道玩家在不列颠尼亚（Britania）拥有一份地产所产生的激情是多么强烈。在一些 MMP 游戏中，房屋供给仍然不是一个主要功能。要想理解特定的房屋供给系统为什么成功，我们必须先理解与此相关的游戏设计。为了能够更好地理解房屋供给系统是怎样运转的，我们必须先澄清一些对玩家购买房屋动机的误解。

我们要讨论的第一个误解也是这些误解中副作用最大的一个：有人认为玩家希望把房屋作为私人空间使用。这种说法听上去很正确，但是它并不是在游戏中加入房屋供给的合理理由。既然说玩家需要私人空间，假设这个说法具有下面两种意思之一：玩家希望可以独处，或是玩家想要一个地方来与其他玩家进行讨论而不必担心会被打扰。

如果第一个假设是正确的，也就是说玩家希望独处，那么玩家只需要独自探险或是根本就不要登入游戏就可以更好地满足这个需要。也许会有一些玩家想要在一个大型多人环境中独处，但是我敢肯定这样的人绝对是少数。

第二个假设（希望和朋友们独处）或许是正确的，但如果游戏中有一个强大的聊天系统并且具有足够的地标（landmark）可以让玩家群体在那里集结，那么与朋友们独处这一需要并不会激发对房屋的需要。在一个私人聊天系统中，大多数玩家群体可以在任何地方自由地讨论他们想要讨论的任何事情而不必担心会被打扰。如果在安全区域有一些明显的地标可以被玩家用来作为开会地点的标志，玩家群体就可以在那些地标附近聚集而不必担心会被打扰，这让进行会议比进行一次简单的聊天更为亲切。

在《创世纪在线》中玩家获取房屋的目的多种多样，而隐私实际上是最不常见的一个。事实上，当玩家可以选择把一间房屋声明为公共的或是私人的时，大多数玩家选择了前者。私人房屋意味着只有拥有钥匙的玩家才能进入，而公共房屋意味着每个人都可以进入。当我们加入一个系统使得玩家可以把房屋中的某些物品“锁住”（这意味着只有这间房屋的主人才能拿走它们）后，私有住房变得越来越少。在那时，《创世纪在线》中还没有一个强大的聊天系统，要进行一次真正的私人谈话，惟一的方法就是拥

有一间只有玩家和他的朋友才能进入的房屋。那为什么公共住房会更为流行，即使它以牺牲私人交谈为代价呢？这个问题的答案在下面这些原因之中：

- 成就感；
- 展示成就；
- 通过商业进一步积累财富；
- 传统的收藏行为。

6.2.1 成长之路

很少有另一个 MMP 游戏拥有像在《创世纪在线》中建造一间房屋那样令人愉快的游戏经历。原因之一是在 *UO* 中很难买到房屋，因为房屋不仅价格很高，而且供应量也很少。玩家必须不断存钱（通常是和其他玩家一起存钱）才能购买他们的第一套房屋。一旦玩家有能力购买房屋，在游戏中为房屋选址则是另一个挑战。在 2000 年春天，不列颠尼亚中的地产已经非常有限了，玩家很难找到建造一间房屋所需的空间。这个困难常常会引起争论。有些人觉得应该有一些难以实现的挑战，而另一些觉得房屋对于游戏中其他关键特性来说至关重要，而缺少房屋空间会使很多玩家不能接触到游戏中一块很大部分的功能。无论哪种意见是正确的，有一点是肯定的：在《创世纪在线》中为房屋选址的回报是不确定的；与房屋选址相关的不确定性使得一旦选址成功后玩家会格外高兴。

一旦玩家幸运地拥有了一间房屋，他们的注意力通常就会转向获得一间更好的房屋。《创世纪在线》的一个重要特性就是它为房屋升级提供了明确的路径，这为那些想要获得一间更大房屋的玩家不断地制造挑战。当然，获取更大房子的难度成指数增长。它们不仅需要花更多的钱来购买，还需要找到更大的土地，而后者对于大多数普通玩家来说几乎是一个不可能的任务。这使得玩家花费大量游戏中的或是真实世界中的金钱来从屋主那里购买在昂贵土地上的小屋。玩家们（通常是整个工会）会购买三间或更多彼此相邻的小屋并且把它们从地上删除从而为塔楼、要塞或是城堡腾出空间。虽然这种挑战会带来很强的荣誉感和成就感，但是要让它成为一个重要的成就，我们就必须让玩家在房屋中能进行一些独特的行动。

6.2.2 商业方法

当《创世纪在线》中第一次引入房屋时，房屋除了作为集合场所或是社会地位的象征以外，没什么别的用处。因为玩家可以从任何房屋内拿走任何物品，除了一小部分团体会邀请玩家去他们的家或商店中聊聊以外，大多数房屋并不向公众开放。大湖碎片（Great Lakes shard）的 Kozola 以及 Baja 碎片的 Golden 最终发展为人口众多的城镇，就连 *UO* 中最初的城镇都很少能有那么多的人口。这些成功来之不易，也促使玩家要求更好的工具来帮助他们发展社团。开发团队看到了这些由玩家创建的城镇和聚居地的前景，他们开始使用更多的时间来开发玩家所需的社团工具。从此 *UO* 中的房屋开始具有一些独特的功能，这使得它们对于玩家来说更具吸引力。

商人是附加功能之一，它使得玩家可以在下线时共享和交易货物。随着玩家通过历险积累的物品越来越多，他们需要一个地方来进行销售。要在 *UO* 中的银行销售货物非常困难，

因为那里不仅非常拥挤，还会发生很多偷窃行为。即使在游戏中彻底禁止偷窃后，很多玩家仍然不喜欢在城镇中销售货物，因为他们觉得站在人群中吆喝上几个小时只为了销售一些货物并不有趣。由于没有一个全局的聊天或是拍卖频道，与其他玩家进行交易非常困难。那时，有些玩家甚至会对《创世纪在线》中的交易功能进行滥用来欺骗其他玩家。

商人在此时应运而生，它被用来解决这些问题并且为玩家的房屋带来更多的人流。《创世纪在线》中的商人是销售货物的 NPC，玩家可以雇用它们。玩家可以让商人帮他们销售货物，而商人会收取少量费用以持续工作。只要支付了费用，商人会一直开心地站在房屋门口，而其他玩家则可以浏览商人手中的货物。商人要求玩家必须有一座房屋，这进一步增加了对房屋（不是说是幢房屋就可以了，而是那些在高人流区域的房屋）的需求。

6.2.3　地段、地段、地段！

随着商人的出现，*UO* 中的地产变得更为昂贵，这也让 *UO* 中的某些地方变得非常拥挤。当然，之后游戏设计者意识到应该在游戏早期就通过为房屋选址制定规则来对此加以限制。这些限制后来都加入了，但是 *UO* 中很多地方已经被房屋弄乱了，这都让部分玩家有幽闭恐怖症的感觉了。玩家在野外历险时被房屋完全挡住去路的情况也并不少见。因此可得出这样一个结论：房屋应该位于一个独立的、远离繁忙人流的区域。

然而，这个结论可能并不正确。建立一个这样的独立区域会为由 NPC 商人形成的商业带来负面影响。那些高人流的区域，譬如说进出城镇或是通往特定场所的道路以及常用的狩猎地点等，通常可以为玩家雇用的商人带来更多的顾客。让这些区域彼此独立会让人流变少，虽然我们已经支持可以让玩家移动到任何位置的瞬间移动魔法，这在某些程度上与我们的理论不符。然而，相对于那些看不到的人流来说，所有直接在房屋边通过的人流会为房屋带来更多的顾客。

此外，在 *UO* 中建造房屋带来的回报还类似于那些可以让别人看到的战利品或是可以炫耀的事情。战利品最大的价值在于展示出来让别人看到，这意味着拥有住宅区中最大的房屋这一事实本身的意义相对于这间房屋对于其他玩家的视觉影响来说要小很多。对于 MMP 游戏中的大多数玩家来说，没有人羡慕的成就是没有意义的。

房屋供给系统得以成功的真正原因直接依赖于特定的游戏设计。如果一个游戏具有强大的拍卖系统，购买货物的人会被立即传输到存放货物的地方，那么存放货物的具体地点并不重要，我们也不再需要售货窗口了。《无政府在线》（*Anarchy Online*）中的拍卖系统就是一个很好的例子（虽然它需要玩家在线才能使用），它可以在内部显示所提供物品的属性。如果更多的游戏使用这种类型的系统或是像 eBay 那样基于浏览器的拍卖，那么在房屋中交易物品将成为过去。如果这种交易需要在房屋中进行，那么房屋仍然会起到重要作用，但是房屋内部所起的作用可能会比外部的更大。

6.2.4　应该把载放在哪里？

如果玩家把住房的内部向玩家社团开放，他们就需要室内设计工具来表达自己，正如玩家社团中其他表达玩家个性的扩展一样。在《创世纪在线》中，玩家可以在家中以任意方式

在任意位置放置物品来展示他们的创造力和设计能力。

对物品的展示最后产生了一个副作用，那就是《创世纪在线》中的珍品市场。珍品市场得以发展是因为玩家可以在房屋中展示一些非常难以获得，甚至需要通过类似黑客的手段才能获得的物品。对这些不同寻常的物品感兴趣的其他玩家会去那些具有珍稀物品的玩家家中询问。最终玩家会使用大量的金钱来购买那些看上去没什么用处的物品，而这完全是为了让一间房屋变得真正的独一无二。某些玩家的爱好就是去看看《创世纪在线》中的每一种物品，但是如果一个玩家想要拥有任意可以拥有的东西，他就会变成谢尔•西尔弗斯坦笔下的哈克特（Hector，西尔弗斯坦名为 Hector the Collector 的诗中喜欢收藏各种废品的人物）那样。

因为玩家可以自由地放置物品，他们可以为他们的家创建令人吃惊的内部装饰，甚至创建新的物品。玩家可以用棋盘和染色的斗篷制作大钢琴，或是用图纸和其他各式各样的东西来制作水族箱，甚至可以用棉花做出装饰性的喷泉。进入一个玩家的屋子有点像进入废物拼装比赛（Junk Yard Wars）的现场一样。玩家很少把原料按照它们本来的用途进行使用，但是它们都莫名其妙地工作着。

然而，有些开放性的规则最终对游戏造成了负面影响。因为没有对房屋中可以容纳的物品数量进行限制，很快玩家就开始用这些房屋来储藏那些不需要的垃圾了（虽然公平的说，这个游戏鼓励储藏，因为玩家在死亡后会丢失物品）。我们必须对房屋中所能放置的物品作出限制，这促使开发团队设计出了一种巧妙的方法：给予玩家奖券（就好像在本地的皮萨店里面可以看到的那些用来玩史基球（SkeeBall，一种手动球类游戏）的那种）来奖励他们删除那些不需要的物品。从玩家社团的角度看，这个富有创造力的计划把一个本来会非常难以推进的过程简化了，但是我们仍然建议在开始设计房屋供给系统时就进行限制，这样就不必在以后为添加限制头痛了。要知道，人们不喜欢放弃东西。

6.2.5 经验与教训

《创世纪在线》的经验让我们知道加入房屋供给的好处：它为 MMP 游戏的设计方面带来了很多有益的功能：

- 它可以作为一个社交集合场所；
- 它代表了玩家的发展；
- 它可以用来自我吹嘘或炫耀；
- 它是一种对游戏世界进一步定制的方法，相对于玩家形象来说，它可以给游戏世界带来更大的影响。

UO 中的房屋供给系统还包含了一些上面没有提及的功能，但是通过阅读本文已经可以对最重要的部分有所了解了。上面列出的 4 个功能已经获得了成功，与其尝试加入更多功能，可能对怎样把它们应用于将来的游戏中进行讨论会更为重要。

当试图把房屋作为社交集合场所时，促成这种集合的情形必须是房屋所独有的。如果这个特性在其他地方有所重复，那么它在游戏设计中对促成集合的影响就会打点折扣。不妨想象一下玩家会因为什么原因集合起来，如果这些特性可以被用在房屋供给上，就这样做。虽然每个游戏都是用独特的方式来促进社会交往，但是它们的目标都是一样的：建立社会联系。社会联系会把玩家和游戏绑定起来；而快乐则是最好的催化剂。进行商业活动是对造访其他

玩家房屋最好的理由之一。这是一个基于需要的系统，在任何具有交易功能的游戏中都会自然地出现。让玩家可以方便地浏览货物也非常重要，但过于方便有时候会损坏食物的口味。

如果游戏中的成长路径并不能使每个玩家都能成功，那是很危险的。况且，那些已经实现了终极目标的玩家将会发现很少会有更为高兴的事情了。我们并不推荐在游戏中设置一个具有很多特性的终极目标并且这些特性只有实现这个目标的玩家才能使用。到撰写这篇文章的时候为止，我们一直确保 *UO* 中所有的玩家都能赚到购买一间房屋所需要的钱。不能确定的只是玩家房屋的大小，这才是给予玩家挑战和回报的地方。房屋供给系统必须具有一条有挑战性的成长路径并且给予玩家相应的回报。

对于大多数玩家来说，能够对他们的成就进行展示本身就是一种回报。房屋使得玩家获得的回报和成就可以被大家看见并且长期存在。MMP 游戏最关键的特性之一就是给玩家一个交谈的理由。一个玩家可能会问“当你获得房屋之后，你做了些什么？”或是“哇塞！那么长的一支魔法剑。你在哪里得到的？”因此很多设计人员花费很多时间来创建那些可以用来吹嘘的故事。玩家对小说和背景故事的关心程度就好像普通公民对政治系统的关心程度一样，而可以吹嘘的事情则让玩家有理由去创建他们自己的故事。它远比任何开发人员创建的故事更为重要，因为它与玩家直接相关。

最后，MMP 游戏玩家的终极目标之一就是以有意义的方式对游戏世界产生影响。让玩家可以管理一家商店、开一家酒馆或是拥有一座壮丽的城堡可以让他们以一种难以置信的方式对身边的世界产生影响。

6.2.6　总结

设计人员可以对 MMP 游戏中房屋的未来进行大量的探索。不是每个 MMP 游戏都需要房屋，房屋也不适用于每个游戏。然而，还有很多方面有待探索。玩家可以创建一个独特的冒险并且使用房屋作为进入这个冒险的门户，甚至可以直接把这个冒险创建在房屋里面。在未来的游戏中，我们应该提供更多的工具来让房屋真正地成为由玩家管理的城镇中的一部分，这些城镇可以有自己的政治系统和规则。玩家应该可以像搭积木一样搭建房屋，这样就可以按照他们想要的方式来建造房子了。我们可以做的事情就像房屋供给系统设计人员的想象力那样无穷无尽。

6.3 社会游戏系统：促进玩家社会化及提供游戏回报的另一个途径

Patricia Pizer, MMP Design Specialist
ppizer@earthlink.net

毫无疑问，人都是社会动物。在历史上，人们根据他们的共同点形成了不同类型的社团，而接近和共存则是促成这一事实最关键的因素。我们形成了村落，它们都有一些特定的规则来让人们可以更好地生存。从那以后，我们还在各种共同目标的基础上建立了社团。我们有地域集团（从家族到国家）、宗教组织、学术群体、俱乐部等。

自从信息社会进入了网络时代，人们与虚拟世界建立起更多的联系。一家主流 ISP 现在正在做的广告是一个已经成为祖母的老人在那里讨论使用电子邮件是多么方便。随着人口流动的日益加剧，人们开始在全国甚至是全球范围内进行迁移，这会在传统社团内部造成隔阂，而虚拟世界中的互动可以弥补这一隔阂。

在目前公众可以访问的虚拟世界中，大型多人在线游戏所采用的科技最为先进，它正逐步取代全世界所有人进行交流的传统方式。这些“游戏”代表了人类社会的最新进展。作为游戏开发人员，我们应该怎样去影响、培养和塑造电脑世界中的社会经历呢？

6.3.1 什么是社会游戏系统?

游戏系统定义了一个游戏运转的基本方式，是一个与游戏内容相对的概念。譬如说，“咒语”是一种游戏内容（也称为游戏特性），而“魔法”则是一个游戏系统。虽然这个概念听上去很简单，但是很多人不能理解两者之间的差别，甚至包括不少游戏开发人员。一旦我们创建了一个游戏系统，我们就可以很方便地改变和操纵它的细节（也就是游戏内容）。然而，很难改变底层游戏系统，因为它是游戏基础的一个有机部分。

简而言之，社会游戏系统（social game system）就是那些给予不同的社会行为支持、鼓励、回报和惩罚的游戏系统。然而真实生活中的社会系统比这个远为复杂。在传统的人类社会（无论这是一个国家还是童子军）中，有很多方面需要社会化（socialization）。牛津英语辞典把社会化定义为“建立组织或是使自己适应这个组织的过程；特指已经或即将成为某个社会群体成员的个人为了这个社会群体的稳定而改变自己的行为和价值观

的游戏时必须合作。而创建那些可以独自进行游戏类别事实上会影响目标。《亚瑟龙的呼唤》中的角色升级机制不仅让玩家可以独自进行游戏，还对此加以促进，甚至在很高级别的游戏中，玩家都不必和别人合作。在进行 MMP 游戏时不和游戏世界中的其他玩家交流是对网络带宽的浪费。他们最好去玩单机游戏。

MMP 游戏是建立在玩家交互的基础上的，正如玩家在其他合作性或政治性的社会组织中一样。对所加入的社会缺乏归属感和投入感会导致负面行为甚至是反社会（sociopathic）行为；反社会的人仇视社会和朋友，他反对社会赖以存在的原则，他缺乏甚至反对通常的社会本能和行为。如果玩家和游戏世界没有联系并且也没有任何东西可以失去，为什么他还要遵守社会规范？

社会系统的主要目标之一就是积极地回报正确的社会行为并且阻止反社会行为，然而在 MMP 游戏中这个问题并没有被有效地解决。虽然有些系统对此有所触及，譬如《创世纪在线》中的声望系统，但是这个领域仍然有待研究，它对于为新生的虚拟社会中的行为建立社会规范和指导方针来说非常关键。

4. 培养归属感

良好的社会系统有助于在游戏世界中建立归属感，这是一种人类的基本需要。请参考马斯洛需要层次（Maslow's hierarchy of needs）来获得对其他基本需要的说明[Maslow68]。成为群体的一部分是马斯洛需要层次中的第三步，这包括归属、联系、被他人接受、付出或获得爱和友谊。用来实现这些目标的社会系统可以简单导致是一个具有有效聊天工具的工会系统。然而，复杂的系统可以更进一步，正如将要介绍的。

5. 小型世界网络

最后，当对社会游戏系统进行考虑时，应该对“小型世界（small world）”网络和玩家组织的效果加以研究。小型世界网络可以告诉我们两个相互连接的顶点之间的关系（在本文中，也就是玩家之间的关系）是什么。小型网络世界的理论之一就是“六度分离（six degrees of separation）”现象，也就是说在这个星球上的任何人同任意他人之间的“度数（hop）”小于等于 6[Milgram67].

譬如说，如果你最好的朋友的表弟和罗伯特·里德福德是邻居，那么你和罗伯特·里德福德之间的度数是 3。有一个名为“凯文·培根的 6 度空间”[UVA99]的 Web 游戏可以说明这个现象。这个游戏是由 3 个宾夕法尼亚大学的学生（Craig Fass、Brian Turtle 和 Mike Ginelli）发明的。使用这一现象，我们可以创建玩家组织系统来促进虚拟世界中的“联系”。通过提供更强的社会系统，可以降低所需的度数，进一步增强玩家间的联系。

6.3.2 为什么要让玩家社会化?

目前几乎所有获得商业成功的 MMP 游戏都使用按月付费的商业模式。要在这个模式中获得成功，就必须让玩家有理由继续回到游戏中。那种迫使玩家回到游戏中来的想法称为“粘性（stickiness）”[Gladwell]。通过增加粘性，可以保证有更多的玩家，从而使玩家群体保持稳定并且有利可图。

1. 创建自治社会

没有人会觉得要维护一个虚拟世界是非常方便和廉价的。在客户支持、持续的游戏内容创建、社区管理、带宽、服务器等上的持续支出会很快增长。作为开发人员，如果我们可以创建一个不需要开发人员的参与就可以独立运转和发展的自治世界，就可以大大地减少游戏的长期开支。就算不考虑这个方便的“商业”原因，也需要让虚拟世界持续运行来让玩家社区保持高兴。虚拟世界中的居民会与这个世界建立很强的联系，如果“他们的”世界消失了，他们会觉得受到伤害、愤怒、困惑、悲哀，甚至还会觉得他们的权利被剥夺了。如果游戏世界以一个让玩家不高兴的方式进行改变，玩家会转向其他游戏。

2. 减少“反社会”玩家

那些不同程度地进行反社会行为的玩家，通常被称为“捣蛋者（griefer）”。这个绰号是因为这些人会为游戏世界中的其他玩家制造麻烦（grief）。这种行为可以是用一种令人讨厌的方式和其他玩家交谈，或是使用不受欢迎的语言，也可以是重复地挡住其他玩家的道路来阻挡他们前进，甚至重复地杀死某个特定玩家。

这类行为会给社会中的其他玩家带来负面影响，这可能小到只是让别人觉得厌烦，也可能达到促使其他玩家在真实生活中对捣蛋者做出某些举动。虽然只有大约 3%的玩家会属于这一类，但是他们仍然可以破坏整个游戏世界。这个统计结果是引用 2002 年游戏开发者大会（Game Developers Conference）圆桌会议的结果[GDC02]。然而数据表明一个捣蛋者再次进行捣蛋的概率是 99.7%。花些时间考虑一下这个数据。一旦一个玩家为其他玩家带来麻烦和烦恼，几乎肯定他会再一次使用这种（或是其他的）反社会的方式，无论怎样警告或惩罚他。

这种行为对游戏社区的效果是显而易见的。客户服务代表（Customer Support Representative, CSR）花费大量的时间来处理关于玩家捣蛋的投诉。从 Asheron's Call 的数据我们可以估计大于 40%的 CSR 时间花在与捣蛋相关的问题上，20%的时间花在程序错误上，其余的时间则用来回答各种问题。如果可以把这些时间花费在帮助那些具有技术困难或是遇到游戏中的程序错误的玩家身上，会更有建设性。无论从哪方面说，这都不是一个简单的问题。然而，如果可以解决它，我们就可以获得一个更好的游戏世界以及一个更稳固的游戏社区，这使得它成为一个非常重要的社会问题。

3. 增加玩家的股份

玩家“股份”是指任何给定玩家在游戏世界中所有的投入。假设一个刚开始进行某个游戏的玩家（在创建了他的第一个角色并且升了几级之后）发现他认识的所有人都在另一个服务器上。这个角色的“股份”仍然很低，因此玩家很可能会放弃这个角色并且在他的朋友进行游戏的服务器上创建一个新角色。另一方面，某个玩家在进行了一定的投入（譬如说让一个角色升到 10 级）后，发现由于他早期的选择错误，这个角色具有致命的缺陷。但是由于他和这个角色有一定的感情（通过时间、共享经历等形成的），即使有缺陷，他也很难放弃这个角色。这个角色开始具有“粘性”了。

同样，一个积极地参与到某个虚拟世界社会活动中的玩家对于那个世界投入很多，因而忠诚度也很高。当《亚瑟暗实际》开始发行时，《无尽的任务》中的活跃玩家明显地少了。这

两个游戏有很多共同点，有些玩家觉得更喜欢《亚瑟暗世纪》并说服朋友跟着他们一起去新的游戏世界。过了几个月，那些玩家中的一部分回到了《无尽的任务》中；虽然有些玩家是因为发现那个新游戏在某些方面有所欠缺，而大多数玩家是因为他们不能放弃他们在《无尽的任务》中的“股份”。事实上，他们想家了。

4. 创建更丰富的游戏世界

当然，要增加玩家在某个游戏世界中的“股份”，一个明显的方法就是在深度和广度上拓展这个游戏世界。这里的广度是指实现更多的游戏系统，尤其是社会系统。深度是指为游戏世界中的居民提供更丰富的游戏体验。更好的 AI、更复杂的任务以及更广阔的地理区域等条件都可以增加玩家游戏体验的深度。在游戏中加入复杂的社交系统可以同时增加游戏世界的深度和广度。

5. 它不仅仅是一个游戏

我们需要创建社会系统来给玩家正确的社会行为以一定的回报，这样做最重要的理由是那些并不仅仅是“游戏”。本节讨论的都是玩家和社会系统，然而，真实情况是这些虚拟世界已经超出了传统游戏的范畴。这些服务几乎支持所有传统的人际交往。它们是一种人类通讯的新媒体；它们在很多地方就和电话一样，具有自身独特的重要性。没有人会认为电话只不过是一种新奇的玩具。很多人下意识地否认这个结论，理由是“但是这不是真实的，它是虚拟的。”任何成功发布了 MMP 游戏的厂商以及那些游戏的热爱者都知道这句话是错的。

譬如说，有些玩家在游戏世界中发现了他所未开发的技能或是天赋。成为领袖的玩家在生活中并不一定有机会以一种安全的方式也成为领袖，而游戏世界使得这成为可能，这增加了玩家的技能和信心，使得他在真实世界中也会使用这些技能。举一个令人吃惊的例子，《无尽的任务》中一个玩家在真实生活中的技能受到了游戏的影响。那个玩家在真实生活中曾经害怕游戏。在游戏中，玩家掉入水中并且沉了下去。玩家最后发现这并不会发生他所害怕的结果，于是他开始在游戏中游泳。最后他克服了恐惧并且在真实生活中也学会了游泳。

玩家会爱上游戏世界。虚拟爱情会摧毁真实世界中的婚姻或是导致真实世界中的婚姻。有些玩家过于投入他们的虚拟生活以至于影响到了他们的工作和人际关系。这些游戏社会可以和与之互补的真实社会竞争，因此，必须严肃地对待它们。

6.3.3 目前使用的社会系统

几乎每一个在销售的 MMP 游戏都具有某种形式的社会系统。至少都具有聊天功能，这可以让玩家彼此进行交流。然而，还有一些通信系统有待研究。让我们从最基本的开始。

1. 聊天

目前，在市场上每个成功的游戏中都有某种形式的聊天系统。它们的形式多种多样，但基本上都具有类似的功能。有些游戏使用语音聊天，这既有优点也有缺点。根据所使用的语音客户端的不同，这可以使得玩家更为投入游戏，也可以完全打断玩家的游戏经历。在一个

超现实游戏中，玩家自己的声音不仅可以让他们的角色更容易辨认，还可以让玩家更为投入，并且可以改善玩家间沟通的体验。另一方面，对于那些在游戏中改变性别的玩家来说，这会完全破坏对游戏的投入。

在现有的游戏系统中已经建立起不少常见的功能。如说，大多数游戏使用文本在玩家之间传递消息。玩家在一个文本框内输入他们想说的话，然后这些话会在所处的世界中广播。通常，游戏提供“频道”来区分哪些人可以接受某个信息。这通常使用类似于标签的界面实现，玩家单击与他们想要发送消息的频道相对应的标签，在文本框中输入，然后按一下回车键送出消息。这样一来，玩家可以向某个特性玩家、某个特定群体或是消息可以到达的所有玩家发送消息。

不仅如此，在大多数游戏中还可以向其他玩家做“表情”，这是一种文本或是图形化的符号。譬如说，在 Asheron's Call 中，玩家可以向其他玩家“挥手”。这种功能为玩家进行彼此交流提供了另一种方法，即使不是口头的。下一代的游戏都计划支持语音聊天，这使得玩家可以真实地进行交流。

2. 层次结构

《亚瑟暗世纪》中的行会系统是一种当前正在使用的层次结构。工会必需有一定数量的玩家创建，他们可以为工会注册一个名字。成员可以通过购买顶饰（crest）继续为工会投资，这些顶饰可以用在那些作为防具使用的斗篷上。除了具有一个私人聊天频道以及可以迅速地找到一起历险的其他玩家以外，行会系统并没有提供多少其他功能。根据记录，大多数优秀的 MMP 游戏具有某种形式的行会系统。

《亚瑟龙的呼唤》中的效忠君主系统是层次社会系统的另一个例子。通过使用效忠系统，一个玩家可以成为另一个玩家的主人，而另一个玩家则成为附庸。在金字塔顶端的人就是君主。在这个特定的系统中，与游戏中行为相应的经验值会向上延伸，某个主人的附庸获得了经验值后，他也会获得经验值，这样一直向上直到君主。因此，具有大量附庸的君主可以持续不断地得到经验，即使他并没有积极地进行游戏。

有人把这称为金字塔效应。虽然系统鼓励玩家去寻找附庸，但是它并不强制或者鼓励主人去帮助他的附庸，尤其是那些菜鸟玩家。譬如说，如果当主人从附庸那里获得东西时，会给附庸一定的回报，那他们之间的关系就不再是单方面的了。虽然大多数主人都会在某种程度上帮助他们的附庸，但很多主人和君主只是坐在那里看着经验值上升。正如任何历史上的君主体系一样，在金字塔底端的附庸需要承担在他们之上的所有重量，并且不会获得任何有意义的帮助。

3. 声望

《牛津英语字典》（*Oxford English Dictionary*）对于声望（reputation）作出如下定义：

“根据某人的人格或其他品质对其作出地一般评价；某个人或是某种事物所受到的相对评价或尊重。”

声望是对某个人各方面映像的集合，它是这个人在和其他人交往过程中言行的反映。它可以是负有盛名，也可以是臭名昭著。

声望系统系统让玩家可以评价其他玩家。正如一个人在真实生活中累积声望一样，他在

虚拟世界中也可以这样做。譬如说，如果一个人在一次交易中欺骗了别人，受害者可能就会花些时间来对犯错者的声望作出相应的评价。

《创世纪在线》是一个典型的图形 MMP 游戏，它通过一个声望系统来让玩家可以把其他玩家标为正的或者负的。同样，eBay 所使用的声望系统让参与者对彼此进行评价，这样其他参与者就可以对于某个特定个人做出有根据的决定。如果一个卖家声望不是很好，潜在买家可能不会与他们交易。

然而，这类系统需要进行非常仔细的构建，否则它们会被破坏；玩家可以轻松操纵这些系统来让别人获得不良声望。虽然这类系统比较容易被滥用，但是一个经过仔细构建和测试的系统不容易受到操纵，因而可以帮助玩家选择是否于某个特定的人一起进行游戏。虽然永远都有玩家乐于给别人留下坏印像，但是大多数玩家会寻找和他们类似的玩家来获得一次更快乐的游戏体验。

4. Bartle 类型

1990 年，理查德·巴特（Richard Bartle）发表了一篇关于玩家怎样进行多人地下城（Multiuser Dungeon，MUD）游戏的文章，MUD 是目前 MMP 游戏的基于文本的前身[Bartle90]。他对玩家动机进行分类，这个分类对于了解和创建 MMP 游戏来说是非常有用的。这一开创性的研究介绍了 4 种不同类型的玩家，他们进行游戏的目的分别是探险、获得成就、杀人和社交。

让我们看看这些定义。探险者希望了解这个世界，他们会为游戏世界绘制地图，测试游戏机制，在新地区被引入游戏世界时立即发现它们；希望获得成就的人为自己创建与游戏相关的目标并且努力去完成它们。这些玩家主要积攒战利品或是寻求地位。杀手则会缠着其他玩家，寻求展示比其他玩家更为优越的机会。通常，这适用于支持杀害玩家（Player Killing，PK）的游戏系统。当游戏系统不支持杀害其他玩家时，他们会寻找各种方法来和其他玩家捣蛋。最后，喜欢社交的玩家与其他玩家进行交互；或者说，他们在进行社会化。

虽然每个玩家都会在不同程度上表现出 Bartle 所说的 4 种玩家类型，但是我们可以确定所有玩家在社交这个类型上的程度都是大于 0 得。即是那些纯粹的杀手通常也会与其他玩家交互，吹嘘他们对游戏系统的滥用、嘲弄其他玩家，或是使用游戏中提供的通信系统和其他玩家联系。换句话说，任何让玩家可以交互的游戏系统可以被认为是一个社会系统。

5. 匹配服务

《亚瑟暗世纪》中的聊天系统是一个非常成熟的系统，它和行会系统结合在一起以促进玩家组成小组一起游戏。这个游戏中还使用了一个匹配系统（这可以在所有玩家中选出一小部分以满足某个特定需要）来帮助玩家寻找或是加入其他玩家。譬如说，如果一个玩家小组缺少一个治疗术士，他们可能会寻找一个牧师或是男修道士。一个骑士在独自登入游戏后，如果他不认识游戏中任何当前在线的玩家，他可能会去看看是否有玩家小组在寻找战士；他们可以通过玩家类别和位置来寻找附近的玩家。这个系统的一个特殊优势在于那些需要复活的玩家可以在附近寻找一个治疗术士来帮助他们。

这个系统是目前 MMP 匹配系统中最好的。当然，这个设计良好的社会系统鼓励玩家组

成小组来进行游戏以获得一个共享的游戏体验。事实上，它帮助玩家找到那些可以在在线游戏和他们共同渡过一段时间的其他玩家。在下一节中我们会对此进一步分析。

6.3.4　回报社会性游戏的创新方法

著名 MMP 设计人员拉夫·科斯特（Raph Koster）[GDC02]说过："我们给予那些社会创造者（social builder）的回报太少了"。让我们一起把现存的社交系统升级到新的层次。

1．对目前的社会游戏系统进行改进

几乎目前正在使用的所有社会游戏系统都可以有所改进。某些情况下，这只不过是改进用户界面以使系统更为易用。事实上，个人配置界面的开发会大大改善当前系统中的问题。公正地说，目前大多数游戏都通过补丁、每月定期下载以及扩展包等来进行改进。通过采纳玩家反馈的建议，开发人员可以更好地完善游戏系统。

《亚瑟暗世纪》中的匹配系统具有一个很大的弱点。在目前的系统中无法区分其他玩家。譬如说，如果一个玩家小组的成员都非常讲究战略战术。每当卷入一场战斗之前，他们都会仔细考虑其他成员的情况以及当时的情形是否有危险。然而，这个小组发现他们需要一个战士。通过使用匹配系统，他们在附近找到了一个具有合适等级骑士，而他也正在寻找一个可以加入的小组。

一旦这个小组进入战斗，他们可能发现新玩家喜欢任意行动，与这个小组通常的风险管理方式相比，这会带来很大的风险。然而，直到这个新玩家把整个小组带入麻烦时他们才会发现这点。同样，有的玩家坚持在进行新战斗之前一定要进行治疗和准备，那些喜欢快节奏的玩家会因为等待而感到失望。不仅如此，总是会有一些捣蛋鬼，他们加入其他玩家的小组只是为了给他人制造麻烦。匹配系统的这个弱点导致了很多玩家小组都是"一次性"的，而不会建立起稳定的玩家小组。

这里所谓的"一次性"，是指那些需要投入一定时间和努力的一次性情况。小组必须了解这个新玩家，甚至他的缺点。如果建立了良好的关系，这些玩家可能会长期合作进行游戏，在社会网络中加入新的链接从而加强社会网络。但是那些一次性的行为需要投入同样的努力和时间，并且不会在小组玩家间建立长期联系。

为了显著改善这一服务，我们可以借鉴一度流行的在线约会系统。在那些系统中，参与者会填一张问卷，这可以帮助系统选择合适的人选。通过对小组中的玩家进行优化组合，建立一个强大而稳固的社会网络，正如前面"6 度空间"所介绍的那样。

如何对聊天系统进行改进呢？目前的聊天系统中最需要的功能是可以同时进行多个对话，而交流过多常常会打断游戏体验。不妨在当前的系统中加入一些新奇的功能。譬如说，email 可以用来弥补传统的聊天功能；消息发送也是 MMP 社会中必须的功能，它也可以被挂接到聊天系统中；使用 WAP（Wireless Application Protocol）消息系统，用移动电话来为玩家提供一种新的交流方式。或许我们还可以让电报重生！

这里的关键在于，我们只需要在设计和实现上做出很小努力就可以使用很多真实世界中现存的交流方式。但是必须注意一些问题：对于骚扰的控制将会更为困难。在每一种聊天扩展中玩家必须可以使用某种形式的"忽略列表"，这样他们就可以阻止一些讨厌的联系人。参

照真实生活中使用的交流方式，考虑怎样把它们应用在虚拟世界中。

对其他基本游戏系统的进一步开发为开发人员提供了大量的实践机会。应该怎样对游戏中的交易技能进行扩战使得它对于游戏世界中的社会工作更为重要？应该怎样利用声望来改进社交系统？应该对行会和效忠系统进行怎样的改进才能促使玩家社会化并且加强他们和游戏世界以及游戏社会的联系？

对现存社交系统进行一些修改可以让这些系统更为有效，这可以把它变成更为有用的工具而不是一个简单的安全网。简而言之，应该对现存的游戏系统进行分析。玩一下这个游戏，看看什么可以很好地工作、什么有待改进。类似系统的区别在哪里？一个系统比其他系统更为成功的原因是什么？要改进游戏制作技能，最好的方法就是学习已有的系统。应该使用那些好的系统并且对它们进行改进。

2. 增加社会联系

在马尔科姆·格拉德威尔（Malcolm Gladwell）的著作《引爆流行》（*The Tipping Point*）中，他对“少数法则（The Law of the Few）”的法则进行了定义[Gladwell]。这个概念描述了社会中的一小部分成员是怎样通过他们独特的社交技巧来帮助人们进行联系并且传递消息的。“连接者（Connector）”是指这样一种人：他们能够认识大量其他人。在任何在线游戏中，都有很多连接者积极地把社会中的其他成员连接起来。如果开发人员可以找到某些方法来识别这些连接者并且给予他们回报，就可以让他们帮助创建和加强社交网络。毕竟，大部分游戏爱好者网站都是由他们运行的！譬如说，数据挖掘就是开发人员可以使用的一种资源，它可以帮助我们分析、操纵和改进游戏世界中的社会。应该寻找一些方法用所收集的玩家数据来识别那些特定的玩家类型。

3. 开发与真实世界平行的社交系统

在前面“为什么要让玩家社会化？”一文中，简要地介绍了虚拟世界和真实世界的关系。在虚拟世界中，玩家聚集点往往都是那些建立起真正友谊的玩家。早在 1995 年就建立的一些行会目前仍然团结在一起，这些成员会一起进行游戏，利用行会成员资格带来的好处以及成员所提供的社交联系。很多次，玩家在真实生活中去世了，没想到在虚拟社会中的朋友会参加各种形式的悼念活动，并且也在游戏中对他进行悼念。如果某个行会成员遇到了严重事故并且无法支付相应的医疗开销，他们也会在真实世界中发起捐款来进行帮助。

从这些自发行为中，我们可以得到一个简单的结论：应该使用人们在真实生活中使用的社交系统并且把它们运用在游戏世界中。游戏设计者应该让玩家可以通过在游戏中提供服务或支持来升级。在《亚瑟龙的呼唤》中，一群玩家通过使用烹饪系统来经营饮食服务。这些服务可以为那些像行会会议或是婚礼之类的在线功能提供食物。

且不说这样使用烹饪系统是开发人员从来未曾预料到的，这是一个非常好的例子，它让我们知道玩家会怎样自己动手来创建社交系统。毕竟，那些虚拟客人是否真的需要一杯虚拟的啤酒？不，但这的确让玩家对某个游戏功能更具有投入感。它也让游戏体验更为丰富。同样，我们也可以看到玩家花费大量时间来创建色彩绚丽的全套服装（从上到下），这样他们可以在参加游戏中的婚礼时有合适的穿着。如果玩家可以有这样的想法，为什么我们不能？任何游戏世界中的自发行为告诉我们在游戏中有一种需要未被满足（这通常总是发生在那些喜

欢社交的玩家中，大多数 MMP 游戏不能满足他们的需要）。

4. 为新的社会行为定义规范和礼节

一旦创建了一个游戏世界，在这个世界中就应定义玩家应该遵守哪些社会规范和礼节。目前最常见的规定就是命名规范。设计人员和程序员尝试了所有方法来防止玩家为角色起一个令人不快的、偏执的或是淫秽的名字。起先，只使用一个简单的列表来检验名字中是否包含淫秽词语。然而，玩家很快就以欺骗这个系统为乐，想出各种独特的拼写或是文字组合来绕过这些规则。因此，需要加入更多的规则。

上述的情况是关于游戏规则的，但这也与社会规范有关。另一些情况包括不让玩家使用重复的信息填满公共聊天频道。“寻找神圣之剑。请发消息到 L33THAX0R”的信息在屏幕上跳动 20 次让人觉得非常讨厌。虽然有些系统允许玩家通过用户界面进行定制来拒绝这类消息，但这仍然是一个问题。

此外，对那些社会行为规范的玩家提供回报更具有激励作用。在游戏世界中，有很多符合游戏文化的社会规范，有些玩家会鼓励其他玩家遵守它们，还有一些玩家则通过实际行动来强制执行这些社会规范，我们应该跟踪这些玩家并且回报他们。我们应该怎样跟踪这些事情？应该怎样回报他们？这的确是一个很棘手的问题，但是这也是我们应该做的。

5. 通过仪式

《亚瑟暗世纪》中使用了由《模拟人生在线》（*The Sims Online*）执行制作人戈登·沃尔顿（Gordon Walton）提出的通过仪式（rites of passage）[GDC02]。游戏最初，玩家必须为他的角色选择一个范围较广的类别，但是直到玩家升到第 5 级时他才必须选择一个专业领域。同样，一旦玩家升到第 10 级，他们可以为角色选择一个姓。对于玩家的成就来说，这是一个简单、清晰且可见的记号；我们看一下这个角色就可以知道他们已经至少升到了 10 级。所有其他玩家都可以通过这个记号来知道这个玩家的状态。虽然选择一个专业领域是一个非常简单的通过仪式，它可能对外界是可见的，也可能是不可见的，但是获得一个姓可以被所有人方便地看到。通过设计类似的通过仪式（玩家成就的里程碑），不仅可以加强玩家在游戏世界中的参与感，还可以增加他们对于角色或是整个游戏社会的“股份”。

6. 新的经济体系

在目前大多数游戏（以及大多数即将发行的游戏）中，经济总是以财富为中心的。财富可能由货币、战利品或是商业组成。货币是最明显的经济基础；然而，要在这些经济体系中对货币进行维护和平衡一向是非常困难的。在大多数流行的 MMP 游戏中，货币受到恶性通货膨胀的影响，正如一手推车的二战前德国马克只能购买一个面包。我们都知道怎样获得战利品，不是吗？去杀死怪物、完成任务或是向别人索要。商业机能系统让玩家可以制作货物并且与其他玩家交易。

我们已经有了三个既定的经济体系。还有什么可以被用作为货币？经济学家认为可以使用各种不同类型的等价物（commodity）。等价物本质上就是有用的东西。一张美元钞票就是最常见的等价物。时间和空间也是。任何等价物，无论是物质的还是概念的，都可以被用来作为经济基础。参考前面所说的声望系统，玩家的声望可以是一种有效的等价物。我们可以

继续寻找新的等价物以在游戏中驱动多个经济体系。

7. 导师

我们也可以使用一个学徒或导师系统来回报那些社交玩家。譬如说，在 *DAoC* 中有一个“学徒”系统，它由游戏世界中的 NPC 管理，而不是由人工管理的。通过让更有经验的玩家对新玩家进行指导，学徒系统可以帮助新玩家社会化。我们也可以在学徒过程中建立一些通过仪式。

8. 社会化

通过观察玩家进行游戏的方式，我们发现可以对 Bartle 的玩家动机定义中喜欢社会化的玩家进行进一步细分。有不同类型的喜欢社会化的玩家。最简单的就是那些使用虚拟世界中的通讯工具的玩家。他们聊天，他们整天挂在公共场合与路过的人交流。他们会在酒吧或是吟游诗人圈子中和别人分享故事。

然而，也有一些玩家使用非传统的方式进行社会化。前面提到的《亚瑟龙的呼唤》中的饮食业者就通过提供服务来发明了一种与其他玩家进行交互的全新方式。还有一些玩家会帮助那些有困难的人、向在危难中的玩家捐献财物或是向那些不熟悉游戏的玩家介绍游戏世界中某些特定的地方。当游戏世界回报这些玩家时，他们通常更看重交流，而不是对经验值的积极追求。

对于开发人员来说，这里的挑战在于怎样回报那些主要以社会事务为动机的玩家。毕竟，是他们把游戏社会连接在一起。他们积极地介绍游戏中的系统来让其他玩家学习。他们说服其他人进入游戏世界。我们需要想方设法报答这些社交玩家，因为他们让游戏世界相对于刚发行时来说，成为一个更丰富、更稳固、更有趣的地方。

6.3.5 总结

强大的社交系统可以提高玩家在进行游戏时的社交成分。从过往经验来看，显然玩家们会使用所有对他们有帮助的游戏系统。因此，通过向玩家提供一个交易系统，某些玩家就会致力于成为一个交易高手（并且获得由此而来的财富）。如果在游戏中可以捕鱼，那么玩家们就会在游戏世界中捕鱼。在 MMP 游戏中设置的任何游戏系统都会被某些玩家所喜爱。通过的游戏世界中提供强大的游戏系统来支持社交规范和文化，玩家就会进行更多社交活动。如果让玩家可以对游戏世界中的社会进行改变，他们就会这样去做。如果我们创建了一个系统，玩家就会使用它，就这么简单。

随着玩家们积极地参与这些社交活动，我们必须认可他们所做的努力并且给予感谢和回报。如果社交系统可以带来具有建设性的新行为，它就会为游戏世界带来更多创造力，对于那些建立了积极的、具有创造性的游戏方式的玩家，我们应该给予回报。如果只认可玩家在游戏中杀死其他怪物和玩家所付出的努力，那游戏设计人员在向玩家传递一种什么样的信息呢？很清楚，如果给予这些行为回报，就是在鼓励这些行为。我们希望确保在玩家中那些积极的行为更受欢迎。这会让玩家们更希望造访我们的游戏世界并参加游戏。

最后，我们都在为创建一个伟大的游戏而奋斗。伟大的游戏提供了更为丰富和深远的

游戏体验。大型游戏社区在互动和发展、在相互学习以及在创造使用工具的新方式上的优势可以让游戏世界更为有趣。这些人与人之间的联系对于任何游戏设计人员所设计的让玩家回到游戏中的其他类型的联系系统来说都非常重要。共享的游戏经历才是引人入胜的，因此我们提供的任何可以帮助游戏世界中的成员们共享他们游戏经历的方法都会让这个世界更为团结。

6.3.6　参考文献

[Bartle90] Bartle, R. A., "Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDs," *Comms Plus!*, October/November 1990, *http://www.mud.co.uk/richard/ hcds.htm*.

[Koster02] Koster, Raph, Patricia Pizer, Gordon Walton, et al., "Are Massively-Multiplayer Games Blazing a New Trail for Humanity?" Game Developers Conference, 2002.

[Gladwell] Gladwell, Malcolm, *The Tipping Point*, Little, Brown and Company, 2000.

[Maslow68] Maslow, Abraham H., *Toward a Psychology of Being*, D. Van Nostrand Company, 1968.

[Milgram67] Milgram, Stanley, "The Small World Problem," *Psychology Today*, 1967.

[OED89] *Oxford English Dictionary, Second Edition*, Oxford University Press, 1989.

[Reynolds99] Reynolds, Patrick, *The Oracle of Bacon at Virginia*, *http://www.cs.virginia.edu/ oracle /*.

6.4 为创建和管理行会设计灵活的命令集

John M. Olsen, Microsoft Corporation
infix@xmission.com

玩家组织（本文中简称为行会）在目前大多数 MMP 游戏中都扮演了重要的角色。对于那些渴望和别人一起进行游戏的新玩家来说，游戏中的社会交往是把他们带入游戏的主要驱动力。让玩家能够自行组织起来成为行会可以增强游戏中的社会交往。同时，很多玩家已经习惯了行会或是别的游戏中以其他形式存在的长期组织，设计一个没有行会的新 MMP 游戏意味着我们需要面对他们所发出的大量抱怨。

本文的目的有两个，一是详细地介绍一个自上而下的设计过程，它可以帮助我们发现用户所需要的功能；二是建立一个框架，使得我们在以后加入新功能时不会对游戏的稳定性带来很大的风险。由于交流对于建立玩家间的社会感来说非常重要，本文中使用了很多篇幅对其进行描述。

这里所描述的行会命令集是完全基于文本的，因为几乎所有的 MMP 游戏都是使用键盘进行的。只要在设计上做一个小小的改进就可以把基于文本的命令集修改为多级菜单的形式，这样我们就可以在家用游戏机上使用或是用鼠标来进行操作。要做到这点，只需要把每个关键字替换成一个菜单选项，并且使用菜单列表（而不是通过键盘输入）来选择玩家。

为了对所有的行会命令进行组织，它们将会以“guild（行会）”为前缀以便于进行语法分析。由于篇幅所限，本文并不对命令语法分析器本身进行讨论。

本文将使用下面的格式来描述文本命令：为了避免给玩家带来不必要的麻烦，命令都是大小写不敏感的。这并不包括那些用来表示新分配的名称的参数，它们的大小写应该和玩家提供的一致（譬如说，在指定一个行会的名称时，我们就需要保持大小写）。必须的参数会显示在小于和大于号（<>）之间。可选的参数会在方括号（[]）之间指定。当一个参数可以重复时，会在它后面添加一个“+”。

```
[可选参数]
<必须参数>
[重复 0 次或多次的参数]+
<重复 1 次或多次的参数>+
```

6.4.1　创建

首先，玩家应该能够创建行会。创建一个行会时所需要的所有信息就是初始成员列表和行会名称。假设发出这个命令的玩家也想成为行会的一员，即使他没有列出自己的名字。

```
guild create <行会名称> [用户名]+
```

玩家输入的创建命令可能像下面这样：

```
guild create "Hamster Crew" Bill Bob Fred
```

一个新创建的行会可以仅仅包含发出创建命令的玩家，但是也可以对初始成员的最小数量做出要求以避免对这个功能的滥用。如果我们考虑得更周到一点，还可以要求玩家只能以一个临时的名称来创建行会并且把这一请求放到一个队列中由游戏管理员来处理，而不是立即就能成功创建。只有通过管理员的批准，行会名称才会在游戏中显示出来。这可以在相当程度上减少行会创建中的恶作剧。

随着时间的推移，玩家可能会改变主意，而且，有时一度符合游戏要求的行会名称可能会不再适用。当然，这也可能是因为一个行会名称被游戏管理员否决了，因此创建这个行会的玩家需要选择一个更好的名称。相对于创建一个新的行会并且设法把成员迁移过去来说，让行会中的某些成员可以改变行会的名称是一个更简单的选择。下一节中将会讨论怎样防止那些心存不满的行会成员对改名命令进行滥用。

```
guild rename <新的行会名称>
```

游戏设计者应该对行会改名命令使用相同的名称核准机制以防止玩家使用一个不符合游戏要求的行会名称。由于一个玩家同一时刻只能属于一个行会，并且只有经过授权的行会成员才能为其改名，因此在这里不必指定旧行会名。与此相反，这个命令的游戏管理员版本不但需要指定新行会名，还需要指定旧行会名。

6.4.2　领导

现在已经能够创建行会了，我们还需要某种方式来决定谁是行会领袖。缺省情况下，可以让创建行会的成员成为行会领袖，其他的人则是普通成员。为了便于排序以及指定等级间的相互关系，可以使用一个数字来表示等级，其中行会领袖为 0，普通成员为 100。

大多数情况下，这个仅有两个等级的系统并不具有足够的表达能力，因此需要能够在行会中创建新的等级。这不仅需要为新等级起一个名称，还需要能够指定它与其他等级之间的关系。为了对等级进行维护，还需要一些其他的功能，譬如说，加入为某个特定玩家指定等级的功能以实现升降级。**rank delete**（删除等级）命令将不能作用于行会领袖这一等级上，因为对于一个行会来说，在任何时候都需要有这个等级以及某个处于这个等级的成员。

```
guild rank new <等级名称> <等级数字>
guild rank delete <等级名称>
guild rank list [等级名称]
guild rank rename <旧等级名称> <新等级名称>
```

```
guild rank assign <玩家> <等级名称>
```

现在我们已经可以创建一个行会并且对每个等级进行定义和命名，接着需要为这些等级指定一些职责和能力。行会领袖（或是任何由它改名而来的等级）将拥有所有的权限和能力。不同的游戏设计可以包括不同的职责分类。为了保持语法的一致性，这将会是一个与等级相关的行会命令，我们可以用它来调整某个等级的职责，因此这个命令将使用以下形式：

```
guild rank duties add <职责> <等级名称>
guild rank duties remove <职责> <等级名称>
```

通常，在职责和能力列表中可能会包含是否能够邀请新成员、是否能够删除现有成员以及是否允许这个等级显示行会标识（ShowTag）和行会等级（ShowRank）。如果想要创建一个见习等级使得新成员在成为行会的正式成员之前有一个见习期，最后一项将会非常有用。新成员能够参与像行会聊天频道之类的活动，但是需要升到一个新的等级才能成为正式成员。

我们可以在上面所示的命令中使用下列职责。在这些职责中，有些是某个等级的属性；而另一些仅仅是命令，仅当一个成员被指定为某个可以使用这个命令的等级时，他才能使用这些命令。后面的小节中将对那些还没有描述过的命令进行讨论。

```
Area
Bank
CreateChat
Crest
Invite
EditRank
HideRank
HideTag
Merge
Motd
Remove
Rename
ShowRank
ShowTag
SeeMembers
```

6.4.3 标识

有时，最好能够为行会提供除了名称以外的其他标识。纹章（heraldry）和饰章（family crests）的概念已经存在好几个世纪了，可以借助它们为游戏增加一些感觉。有些游戏已经开始加入行会纹章符号的功能，譬如说《亚瑟暗世纪》（*Dark Age of Camelot*）。可以从一个背景，也称为普通级纹章（*ordinary*，图形最为简单且最为普遍的纹章，例如中斜线纹和十字形）来创建一个纹章符号，如图 6-1 中所示，它由两种不同的颜色或图案组成。

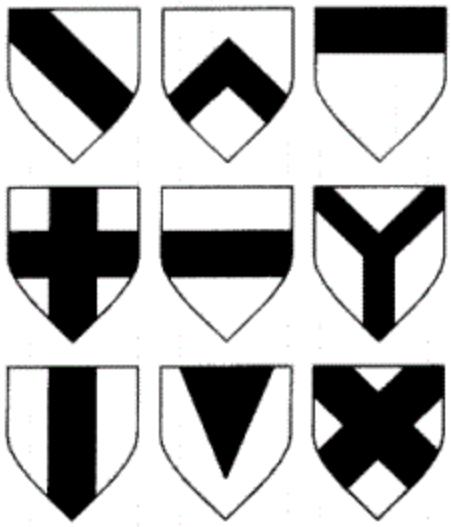


图 6-1　一些历史上使用过的普通级纹章

如图 6-2 所示，我们可以在背景上放置一个被称为图记（*charge*）的前景符号。有很多种普通级纹章和图记可以组合在一起，并且仍然保持一个比较正统的感觉。

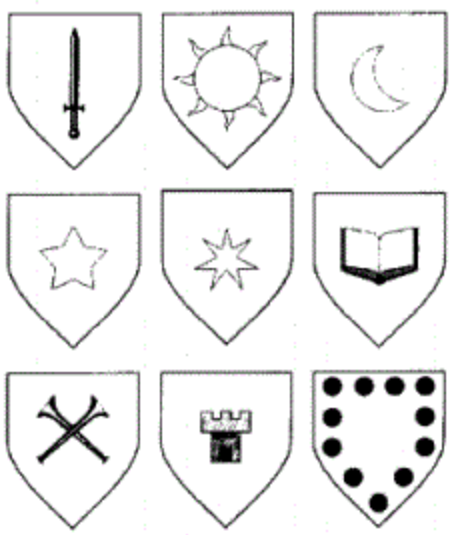


图 6-2　一些可能的前景符号（图记）

典型的欧洲纹章中，只采用很少的几种颜色，它们是银色或白色（argent）、金色或黄色、黑色（sable）、红色（gules）、天蓝色（azure）、绿色（vert）、紫色（purpure）、橙褐色（tawny organe）和血红色（reddish purple）。我们还可以在此之上加入各种皮毛（fur，其实就是银色与其他颜色组成的图案，而不是前面这些纯色），譬如貂皮（ermine）或是松鼠皮（vair 以及 potent）。我们可以在这些颜色的基础上为玩家提供可以在普通级纹章上使用的颜色列表。

为了避免与真实的国旗或是种族象征相同所带来的问题，应该考虑使用那些源自游戏的完全虚构的图记，并且仔细地检查各种可能以排除那些容易遭到反对的组合。

一旦确定了能够选择的饰章，就需要一个命令来为行会指定一个饰章。因为玩家很可能想先试验一下，我们会提供一个包含所有玩家都可以使用的预览模式。

```
guild crest preview <普通级纹章> <颜色 1> <颜色 2> <图记>
guild crest create <普通级纹章> <颜色 1> <颜色 2> <图记>
guild crest remove
```

如图 6-3 所示，这些组合相当有趣并且易于区分。

图 6-3 将图 6-1 和图 6-2 所示的元素组合生成的例子

这样一个基于组件的系统的优点之一是可以具有大量可能的组合。一旦把一个普通级纹章、两种颜色和一个图记组合在一起，就可以生成成千上万种组合，哪怕仅仅使用这里所列出的少数几种组件和颜色。即使这个范围仍然太小，我们也可以使用很多方法来解决。可以在普通级纹章的边缘添加图案（之字线、曲线和开口线等），也可以使用更多的颜色以及更多的普通级纹章和图记。还可以通过一些历史上有据可查的方法把多个图记或是多个完整的纹章符号组合成一个复合纹章符号。只需要简单地在网上搜索“纹章学（heraldry）”就可以获得足够的信息对这个简单的例子进行扩展。

6.4.4 行会维护

每个行会都需要成员。这意味着玩家应该能够为行会添加新人。玩家还应该可以删除那些不受欢迎的成员，虽然这多少有些让人遗憾；此外，玩家还应该能把自己从行会中删除。不带任何参数的 remove（删除）命令意味着删除发出这个命令的玩家。用于添加或者删除其他玩家的命令由那些被授予这个权力的行会领袖使用。能够邀请其他玩家的成员在把自己从行会名单中删除之前，应该给他们一个弹出式的警告。

```
guild invite <玩家>
guild remove [玩家]+
```

当玩家受邀加入行会时，他们不应该自动成为成员。首先，他们必需没有加入其他行会；其次，他们需要显式地加入这个行会。如果他们仅有一个未决的行会邀请，行会的名称是可选的。

```
guild join [行会名称]
```

一旦他们成为成员，他们会被自动地指定为最低的等级。除了让他们输入“join”命令以外，还有另一种方式可以实现这一功能：显示一个对话框询问玩家是否想要加入。应该让玩家的操作尽可能简单，这样他们就不会觉得这个命令集使用起来很麻烦。

玩家常常想要查看行会成员名单以及在线成员列表。所有玩家都应该能够使用下面这两个命令，因此缺省情况下应该允许新建的等级使用这些命令。可以通过指定可选的行会名称来列出玩家所在行会以外的其他行会的成员。通过改变 SeeMembers（查看成员）等级设置，可以改变这些命令的行为，这个设置可以对谁可以查看行会成员列表进行限制。对于非行会成员来说，可以使用行会中最低等级的 SeeMembers 设置，这样就可以简单地对那些行会以外的玩家隐藏行会成员信息。

```
guild members [行会名称]
guild online [行会名称]
```

有时，两个行会会发现它们具有很多相同的利益并且希望能够合并为一个行会。当一个 guild merge（行会合并）命令发出时，它必须获得被合并行会中所有最高等级成员的同意。如果发生一些意外事件，玩家还可以取消合并命令。行会合并并不是一个常见的事件，但是提供这个功能可以大大方便那些有此需要的玩家。

```
guild merge create <新行会名称> <行会名称>
guild merge accept
guild merge cancel
```

这个命令将会保留发出合并命令的行会的饰章和等级。由于不同行会的等级系统很可能完全不同，发出合并命令的行会的领袖必须把玩家升级到新行会中的适当等级。

6.4.5　财产

对于那些具有共同利益的不同玩家之间的交流来说，行会非常重要，但是它们的功能远不止是辅助交流。以游戏中的资产为例。在那些没有行会财产概念的游戏中，在行会中进行装备共享需要采用一系列容易出问题的策略，譬如说“杂交”角色或是密码共享等，这些策略往往会违反游戏的最终用户许可协议（EULA）。

玩家往往采用这一策略的事实指出了一个需要解决的问题。解决这个问题的第一步是允许行会建立银行来保存货币和物品。它不一定需要具有一个实际的位置；玩家可以使用下面这组命令将资金和物品存入或者取出虚拟的行会账户：

```
guild bank donate <数量> <物品>
guild bank disburse <玩家> <数量> <物品>
```

这样就可以把金钱或物品存入行会银行的账户，并且能够把它们从行会银行中取出并给予一个玩家。查看行会基金的运作情况也是玩家需要的一个重要功能，因此需要一个 report（报告）命令来显示所有的近期交易。只要有一个简单的方法可以跟踪物品和资金的转移，就可以降低滥用的可能。玩家们可以决定是否仅仅把分配行会物品的权力赋予一些可信赖的玩家。donate（捐赠）命令向所有成员开放，而所有其他的银行命令受到权限设置的限制。

```
guild bank report
```

已有一个可以存放资金的场所，那怎样才能获得资金？玩家很可能不会记得经常以一个合理而定期的方式向银行捐赠，因此一些行会或许想征收入会费用和每月的会员费用。

```
guild bank fee join <coin count>
guild bank fee membership <coin count>
```

当玩家需要交纳每月的会员费用时，系统会自动提醒他们，而行会的领导者可能希望能够看到一个显示了每个成员交费情况的列表。下面的命令会显示出所有的行会成员以及他们最近的交费情况。

```
guild bank fee report
```

6.4.6 专用区域

如果游戏设计得允许行会拥有自己的会所，我们可以增加功能以根据是否是行会成员以及成员的等级来对能否进入特定区域进行限制。譬如说，行会可以在城市中拥有一个会所，并且仅允许会员进入会所。在会所中可能会有一个委员会办公室，只有达到一定等级的成员才可以进入。

```
guild area add <等级>
guild area remove <等级>
guild area list
```

要使用这些命令，玩家需要到达行会拥有的某个特定区域。一旦玩家在合适的区域，所发出的命令将会被应用于这个区域。list 命令描述当前区域的属性，譬如说允许哪个等级进入这一区域。

6.4.7 交流

行会存在的主要原因之一是玩家们可以更加方便地交流。需要有一个能向整个行会发送信息的行会命令，下面这个命令符合现有的命令方案，虽然格式有点长：

```
guild say <消息>
```

这不仅很麻烦，而且与“交流就应该迅速而简便”这一原则不符，因此应该考虑对这个命令进行简化，正如几乎每个具有这个功能的 MMP 游戏所做的那样。游戏中会有其他长期或短期的聊天频道，譬如说玩家当前的组、任何朋友列表或是和处于同一地理区域的玩家进行聊天。无论采取什么方法来简化上述命令，都应该在把它放入游戏时让它能够与这些长期或短期的频道有效地组合在一起。

对于行会来说，我们还需要考虑“搜捕（raid）”或是其他需要大量玩家聚集在一起进行的大规模行动。通常，有些参与者不是行会成员，并且不是所有的行会成员都会参与行动。这样一来，通过普通的行会频道进行交流就会变得非常麻烦，因此对于大范围的交流来说，创建新的频道变得非常重要。这个命令不是行会命令的一部分，但是在这里把它包含进来，是因为它对于行会交流非常有用，并且为其他的行会命令建立了部分基础。

只需要名字和密码就可以创建一个频道。我们也可以限定能够加入频道的行会。

```
chat create <聊天列表名称> [密码]
chat create <聊天列表名称> <行会名称>+
chat create <聊天列表名称>
```

于是玩家可以随意加入或退出列表。只要列表存在，列表的成员资格就应该保留，即使在成员离开游戏的情况下。玩家可以使用 join 和 leave 命令来控制自己的成员资格，invite 命令可以让其他玩家加入列表。

```
chat join <聊天列表名称> [密码]
chat leave <聊天列表名称>
chat invite <聊天列表名称> <玩家>+
```

在某些游戏设计中，可能不应该允许大规模的邀请，而是让玩家自己来决定是否加入一个列表，这样就可以避免玩家不断地使用 join 命令来攻击别人的恶意行为。

我们还需要处理聊天频道的所有权问题。创建频道的玩家拥有这个频道，但是我们必须对这个玩家离开游戏的情况进行处理。缺省情况下，如果所有者不在游戏中，我们可以简单地把所有权给予在这个频道中时间最长的玩家。但是这可能并不是所期望的行为，因此需要有一个命令来指定所有权。我们没有任何理由不让一个列表具有多个所有者，因此在命令集中加入：

```
chat owner add <聊天列表名称> <玩家>+
chat owner remove <聊天列表名称> <玩家>+
```

删除讨厌的成员也是一个问题。这应该由聊天列表的所有者执行。任何所有者可以删除任何人，包括别的所有者。可以采用的一个更有力的措施是通过阻止玩家来防止他们重新加入聊天频道。

```
chat remove <玩家>+
chat ban <玩家>+
```

通常在频道的最后一个成员从游戏中退出时把这些聊天频道从系统中删除。聊天频道的所有者可以覆盖这个规则以使聊天频道在没有任何人的情况下仍然保持一定时限（譬如说 1 天）。

```
chat persist <聊天列表名称>
```

玩家可能还想要能够查看他们所在聊天组的设置。当使用一个参数来指定某个特定的列表时，我们只对相应列表的设置进行描述。当这个命令不带任何参数时，应该对这个玩家所在的所有列表进行描述。这些信息应该包括成员和所有者的列表，以及他们当前是否在游戏中。

```
chat list [聊天列表名称]+
```

有了聊天列表，玩家们可能希望把列表向公众开放或是把列表限制为专用的。这可以通过增加三个新的聊天命令来实现。前两个改变聊天频道的状态，第 3 个列出所有的公开聊天频道。这可能会是一个很长的列表，因此我们需要允许带有通配符的查找。第 4 个命令可以用来描述聊天频道的目的，这会在玩家发出 chat lists 命令的时候显示出来。

```
chat public <聊天列表名称>
chat private <聊天列表名称>
chat lists [搜索模式]
chat describe<聊天列表名称> <描述>
```

由于一个玩家可以同时加入很多频道，因而应该提供一个命令来控制接收到的文本数量。玩家可以在不退出频道的情况下让它“静音”，这样在重新加入时就会更为方便，因为不需要通过密码验证和所有权的变更。可以通过允许玩家把聊天频道重定向到特定的窗口来降低对这个命令的需要，正如《无尽的任务》所做的那样。

```
chat mute <聊天列表名称>
chat unmute <聊天列表名称>
```

非行会聊天组的基础打好后，我们可以把它合并到行会命令的设计中。譬如说，行会的管理人员可能需要一个他们自己的永久聊天组。这些聊天组应该总是处于活动状态，即使当时没有任何在游戏中的玩家使用这个频道。我们可以简单地通过等级来指定成员资格，而不是通过玩家的名字来指定成员资格和所有权，因此不需要用于加入或离开聊天频道的命令。

```
guild chat create <聊天列表名称> <等级>+
guild chat delete <聊天列表名称>
guild chat owner add <聊天列表名称> <等级>
guild chat owner delete <聊天列表名称> <等级>
```

```
guild chat list [聊天列表名称]+
guild chat mute <聊天列表名称>
guild chat unmute <聊天列表名称>
```

用户界面必须具有一些功能以便于向某个特定的聊天列表发送消息以及对玩家所加入的每个聊天列表中的消息进行监视。这非常关键，我们需要在这上面花费大量的时间以确信玩家可以很方便地进行配置和使用。

本节最初所描述的 guild say 命令会使用一个缺省的行会聊天频道，这会为每个行会自动创建，并且它包含所有的行会成员。行会列表不需要有所有者也不能被修改，因为其成员资格是基于行会等级的。

行会交流中的最后一个问题是应该有一个消息区域使得行会领袖可以发送所有成员都能看见的消息。这是基于 UNIX 的 motd 命令的，motd 表示“今日消息（message of the day）”。这个消息将会在用户每次登录系统的时候显示。不仅如此，我们还必须在游戏服务器停止运行的时候保存这个消息。

```
guild motd [消息]
```

如果没有指定任何消息或是这个用户没有改变消息的权限，就会显示当前的消息。

6.4.8 总结

本文所示的行会命令集无论如何都不是最终的，但是它描述了一系列方法，我们可以使用这些方法来确定一个需求，然后创建一个可以满足玩家的这个需要并且符合现有命令方案的命令集。它非常灵活，足以满足大多数用户的需要，即使他们的游戏风格和对结构的要求各不相同。

在我们觉得某个新命令很有用或是玩家提出新的要求时，可以在现有的基础上加入新的功能。使用行会命令的一个主要目标是给予玩家组织和管理他们自己所需要的所有工具，从而减少对客户支持的需要。如果玩家能够自己处理某个问题，就能减少游戏设计的开支。

一个灵活并且易于扩展的行会命令集可以使玩家之间更方便地进行组织和交流。交流是人们进行任意 MMP 游戏的主要原因之一，因此应该尽可能地鼓励交流。顺畅交流将会使玩家变得更加快乐，这也使得玩家更加稳定并且失去更少的客户。保持客户并且让他们开心将会使设计的游戏更为成功。

6.5　创建声望系统：《无冬城之夜》中的仇恨、宽恕和投降

Mark Brockington, BioWare
markb@bioware.com

《无冬城之夜》（*Neverwinter Nights，NWN*）是一个运行在 IBM PC 兼容机上的基于龙与地下城的多人角色扮演游戏，其游戏设计人员和程序员为 AI 引擎定下了非常宏伟的目标。在第一次对 NPC 会怎样和其他生物进行交互作出假设时，我们试图建立一个容易理解的小型系统，这样就可以在项目早期实现它并且永远也不用再进行修改。

本文将详细介绍《无冬城之夜》中的声望系统是怎样进化为目前深入《无冬城之夜》内部的各种子系统的。本文会按照对它们进行处理的时间先后对 4 个方面进行讨论：友谊和仇恨、宽恕、投降以及玩家对战的设置。

6.5.1　友谊和仇恨的实现

1999 年第一次对声望系统进行讨论时，设计人员说："我们需要一个声望系统来告诉 NPC 我们喜欢哪些生物以及痛恨哪些生物"。这看上去并不是一个很困难的任务。

在那时，《创世纪在线》已经对其原本的声望系统进行了较大的修正，并且加入了因果、名望、罪犯和杀人犯[Grond98]。BioWare 中的所有成员几乎都曾着迷于《无尽的任务》，我们可以看到每当杀死一个怪物时，游戏是怎样改变声望值的。譬如说，通过杀死掠夺者可以改善与守卫者之间的关系，但是这些掠夺者（marauder）并不情愿被玩家角色（Player Character，PC）杀死。最终，他们会意识到玩家角色对于他们的部落来说是一个威胁，他们将会变得非常具有侵略性，每次看到玩家角色都会进行攻击。对于那些并不会向他们发起战争的玩家角色来说，这些掠夺者还是很和平的。事实上，如果玩家角色通过完成任务或者杀死敌人来帮助这些掠夺者的朋友，他们可能会变得非常友好并且给玩家更多的任务。

对于我们来说，很难了解《无尽的任务》中的系统是怎样实现的。聊天窗口中会有类似于"你与掠夺者之间的关系恶化了"这样的语句，但是它并不会说明降低了多少。我们并没有因为缺乏信息而放弃复制这个系统和按照需要对其进行修改的努力。

正如在《无尽的任务》中那样，我们要求游戏世界中的每个 NPC 都属于

某个派系。这可以让我们去描述一个派系中的成员应该怎样对另一个派系中的成员做出反应。譬如说，“怪物”派系应该怎样对“守卫”派系做出反应？我们可能会希望他们彼此见到时就进行攻击。如果一群“平民”正在反抗不平等或暴政来捍卫他们的生命，一个“平民”可能会帮助他的伙伴。在怪物攻击“强盗”时应该发生什么？我们可能会预料其他强盗并不关心他们派系中的某个成员受到攻击，那些看热闹的强盗会保持中立地观看整场战斗直到受到攻击。

这产生了一系列可能的反应，从友好到中立到仇恨。我们用 100（真正的朋友）到 0（仇视的敌人）的范围来表示这一系列的反应。在 *NWN* 中，设计人员决定在大于等于 90 时，这是一个友好的反应；在小于等于 10 时，这是一个仇恨的反应。

在对所有派系之间可能的反应都进行考虑以后，生成了一张像表 6-1 那样的声望表。使用行来表示源派系，使用列来表示目标派系。首先需要注意的是，这张表并不一定要对称。怪物痛恨任何不是怪物的派系，而强盗则把怪物看作是中立的。我们还必须注意派系中的成员并不一定会喜欢同一派系中的其他成员。譬如说，强盗与强盗是中立的。

表 6-1　　全局声望数据表

	怪物	强盗	守卫	平民
怪物	100	0	0	0
强盗	50	50	0	0
守卫	0	0	100	100
平民	0	0	100	80

《无尽的任务》中，这些基本反应会随着时间的推移而改变，因此我们也想要实现同样的功能。公然的恶意事件（譬如说杀害同一个派系中的朋友）会降低玩家在这个范围中的水平。还有一些事件可以改善玩家与某个群体之间的关系，譬如说，向教堂捐钱或是完成一项任务。在最初的设计中包含了三种不同的恶意行为（攻击、杀害和偷窃）。这些恶意行为发生后，系统会自动修改派别的声望，把与此相关的信息放在数据库中名为“HostileEvents（恶意事件）”的数据表中，设计人员可以对其进行调整而不需要依赖程序员。

当系统建立后，我们预料一个小偷会很快在社会中获得坏名声。设计人员并不希望一个在繁忙的街心进行谋杀的玩家和一个晚上在无人小巷里进行谋杀的玩家受到相同的对待。这个问题的核心在于，是否有人看见这个行动。

在 HostileEvents 数据表中加入一列，它表示根据这个时间是否被其他人所目击而进行的不同调整。根据这个目击者是朋友、中立者或是对受害人有恶意，它会同时影响玩家与受害者派别和目击者派别之间的关系。表 6-2 所示为恶意事件数据表。

举一个更详细的例子，假设一个强盗在守卫面前杀死了一个怪物。怪物派系就是受害者，而守卫派系则是目击者。怪物和守卫并不喜欢对方，因此查看表 6-2 中杀害和敌对受害以及杀害和敌对目击者这两个交点。按照这个表，怪物集团对强盗集团的容忍度降低 5 点。守卫集团对于强盗集团的容忍度根本不会改变，因为守卫本来就痛恨怪物因而他们认为强盗在做好事。如果守卫事实上喜欢怪物，我们就必须使用杀害和友好受害者和杀害和友好目击者这两个交点。于是怪物对于强盗的容忍度会降低 45 点，守卫对于强盗的容忍度会降低 25 点，因为我们假设目击这一行为的守卫会告诉他所有的朋友（以及他所有的怪物朋友）这个海盗犯下了多么卑鄙的过错。不

仅如此，对容忍度如此大的修改会导致守卫立即对强盗发起进攻。

表 6-2　恶意事件数据表

目击者容忍度	无	友好受害者	友好目击者	中立受害者	中立目击者	敌对受害者	敌对目击者
攻击	-2	-12	-6	-4	-2	-2	0
杀害	-5	-45	-25	-15	-5	-5	0
偷窃	-1	-5	-3	-2	0	-1	0

注意这些声望调整是立即进行的。这使得目击者可能会立即变为敌对。我们并不使用一个基于事件知识的严格意义上的目击者交流模式[Alt2002]。我们曾经计划在根据受害者的朋友目击了整个事件而进行声望调整之前加入一段延迟。这段延迟可以模拟把信息传递给集团中其他成员时的通讯延迟，这也使得一个小偷角色可以让某个敌对事件中所有目击者都永远保持沉默。然而，这导致了很多问题，最终它被删除了。

还需要注意的是，在表 6-2 中我们无法自动地通过敌对行动来改善和一个敌对目击者之间的关系。如果想要按照“敌人的敌人就是朋友”这一格言来实现，我们可以在杀害和敌对目击者相交处放入一个正数。

玩家角色声望

介绍了 NPC 怎样对彼此做出反应后，接下来介绍怎样对 PC 进行同样的操作。和上面所描写的系统不一样，NPC 对于 PC 的反应变化很大。PC 具有自己的意愿，他们可以通过很多方式来改变其他集团对他们的反应。守卫可能在冒险行动的最初对 PC 很友好。如果某个 PC 开始杀害守卫，我们可以预计这个信息会在其余守卫间传播开，他们最终会决定消灭所有被怀疑的 PC。绝不能让某个杀害了守卫的玩家影响到所有玩家对于守卫的声望。这样做会导致某个犯错误的玩家毁坏其他玩家的游戏经历。

为了把 PC 引入这个系统，我们在表中加入一列来指明当某个 PC 过去在游戏世界中没有任何声望时，其他集团对他的态度。表 6-3 所示为一个修改过的全局声望数据表，它包含了关于 PC 的信息。注意，如果再重建这个系统，我们很可能会为每种 PC 种族加入一列而不是用一列来表示所有的 PC。这可以让设计人员实现种族间的关系，譬如说精灵族和矮人族 PC 之间的仇恨。

表 6-3　《无冬城之夜》中完整全局声望数据表

	PC	怪物	强盗	守卫	平民
怪物	0	100	0	0	0
强盗	0	50	50	0	0
守卫	95	0	0	100	100
平民	75	0	0	100	80

当一个 PC 加入游戏时，在他们角色记录的 PC 容忍度数据表（PCFractionTable）数组中会有这些列的一个拷贝。随着 PC 与游戏中的不同集团进行交互，这个数据表会发生改变来指明

NPC 是怎样看待这个 PC 的。

那么，一个 PC 会怎样看待其他 NPC 和 PC？通常，玩家并不喜欢被告知他们的 PC 应该对某人做出什么反应。然而，为了获得更多的信息，有些玩家希望知道谁是朋友及谁目前敌视他们。因此，每当一个 PC 试图确定他们是否喜欢一个特定的 NPC，我们把这个问题反过来考虑，并且看看这个 NPC 是否喜欢这个 PC。很明显，当这是两个 PC 时这种方法并不能奏效，在我们对玩家之间的设置进行讨论前，我们通过假设所有的 PC 都把其他 PC 作为中立的来避免这个问题。

在下面的代码中，Object::GetReputation()（获得声望）函数返回一个对象对于另一个对象（用 oTarget 来表示）的态度。

```
// GlobalReputationTable stores Table 3.
int GlobalReputationTable[NUM_FACTIONS][NUM_FACTIONS];

int Object::GetReputation(object oTarget)
{
   if (GetIsPC() == TRUE)
   {
      if (oTarget->GetIsPC() == TRUE)
      {
         // All PCs are neutral to all other PCs.
         return AI_REPUTATION_NEUTRAL;
      }
      else
      {
         // How a PC feels about an NPC is
         // the same as how the NPC feels
         // about the PC.
         return oTarget->GetReputation(self);
      }
   }
   else
   {
      if (oTarget->GetIsPC() == TRUE)
      {
         return oTarget->PCFactionTable[self->faction];
      }
      else
      {
         return GlobalReputationTable[self->faction][o->faction];
      }
   }
}
```

6.5.2 宽恕

设计人员很快就意识到这个系统充满危险。如果在游戏中放入几个守卫，并且调整某个守卫的容忍度使他攻击玩家，这导致所有的守卫都会攻击玩家。我们想要的只不过是能够让一个守

卫攻击这个玩家，而不是让所有的守卫都与这个 PC 对立。最后，这个敌对的守卫会忘记所发生的事情并且回复到通常（略微降低一点）的容忍度。

这样个人声望的想法便油然而生。在《无冬城之夜》中，游戏引擎会在遭到 PC 攻击的守卫中保存一个关于攻击他的 PC 的声望调整。图 6-4 所示为《无冬城之夜》中一个被攻击后的典型反应。虚线表示在攻击前守卫对 PC 的态度。

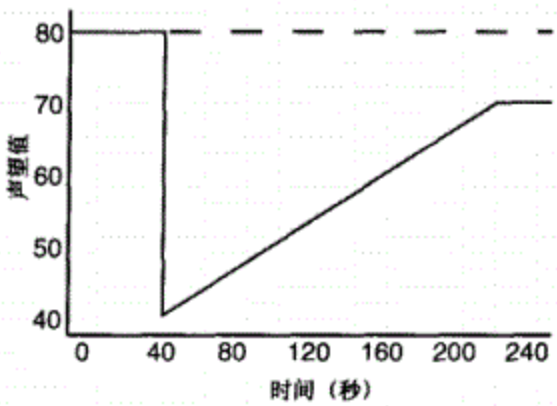


图 6-4　个人声望调整是怎样影响今后声望的

当第一次实现这个系统时，我们坚持使个人声望调整最终必须衰减直到消失（它们不能无限地持续下去）。这样做是有必要的，因为我们担心在保存游戏和游戏引擎中保存成千上万的对象关系所带来的资源开销。在《无冬城之夜》中，这些个人声望改变会不断衰减并在 180 秒内删除。

注意，守卫集团对攻击守卫的那个 PC 的容忍度会永久降低一个很小的量。这可以防止某个 PC 杀死一个守卫，逃走，等待 3 分钟，然后再这样一次又一次地杀死游戏中所有守卫。对最终玩家来说，我们把这个解释为对某个集团进行大量的“偶然”攻击会导致一个明显的敌对模式，因而这些守卫一看到这个 PC 就会试图杀死他。

该个人声望系统是以前面小节中的伪码为基础的，它对 GetReputation()函数的返回值进行调整。下面是 GetFinalReputation()函数的实现。

```
float DECAY_TIME = 180.0f;

int Object::EvaluatePersonalReputation(object oTarget)
{
   int nPersonalReputation = 0;
   CPersonalReputationEntry *pEntry;
   PEntry = FirstPersonalReputationEntry(oTarget);
   while (pEntry != NULL)
   {
      int nTimeRemaining = nTimeExpiry - GetCurrentTime();
      if (nTimeRemaining > 0)
      {
         float fRemainRatio = nTimeRemaining / DECAY_TIME;
```

```
            nPersonalReputation += pEntry->Adjust * fRemainRatio;
        }
        pEntry = NextPersonalReputationEntry(oTarget);
    }
    Return nPersonalReputation;
    }

    int Object::GetFinalReputation(object oTarget)
    {
        int nReputation = GetReputation(oTarget);
        if (HasPersonalReputation(oTarget) == TRUE)
        {
            nReputation += EvaluatePersonalReputation(oTarget);
        }
        return nReputation;
    }
```

最终完成《无冬城之夜》第一关的设计并在引擎中运行时，很多人开始玩这个游戏。在 15 分钟内，有些人就决定模仿一个捣蛋者并且开始杀害游戏中所有的人。最终，他会被守卫追捕并杀害。然而，当最后被复活时，他发现一大群愤怒的 NPC 在寺庙中等着他，因为玩家角色刚刚进行的血腥屠杀让这些 NPC 集团非常不高兴。当玩家角色被这些平民无情地杀害了 5 次以后，测试人员开始到设计人员的办公室提意见："我认为这个声望系统有问题"。

设计人员希望在 PC 死亡后重置所有集团对 PC 的态度。这可以实现游戏机制中最高级别的宽恕。幸运的是，这是一个非常简单的任务，因为游戏机制并不会改变全局全局声望数据表中的 PC 列。当 PC 死亡后，脚本命令会用全局声望数据表的 PC 列重置它的声望数据表，并且删除 PC 死亡时在附近的所有 NPC 所保存的个人声望值。

全局声望数据表中的 PC 列并不会被改变，但是其他所有列都会被改变。这会导致问题。我们的 QA 人员看到友善的鹿群与游戏中所有怪物进行战争。这可能是由于在游戏中一个完全无关的区域中，一个兽人杀死了一头鹿，这引起的声望改变导致所有的鹿进入攻击状态并且开始攻击游戏中所有敌对集团的成员。遭遇战系统（让玩家可以触发遭遇战的系统）中的一个错误导致这些鹿会不断地产生，并且冲向兽人群组，最终被这些挥舞着大斧的兽人杀死。根据不同的遭遇战设置，有些情况下游戏中的鹿的数量是无限的，这样鹿群就会因为人多而最终获得胜利。这产生了一个有趣的 AI 生命场景，但是它毁坏了设计人员的计划，设计人员只是想让兽人和 PC 作战。当 PC 进入这个区域时，他只能看到遍地都是鹿和兽人的尸体。

对于大多数集团来说，设计人员并不希望 NPC 和 NPC 之间的全局声望会被自动改变。所使用的解决方案是把每个种族标为"全局"或是"非全局"的。如果某个种族是全局范围的，就可以对它在全局声望表中的行进行修改。如果这个种族不是全局的，只会改变这个种族中怪物的个人声望。这使得受到攻击的鹿和目击这次攻击的鹿会报复兽人，它可以防止集团中的其他鹿进行疯狂的战争。

6.5.3　投降

投降是角色扮演游戏中常见的情节，我们期望玩家角色在释放或是杀死敌人前听取投降的信息。要实现投降，我们必须把在前面的战斗中对个人声望进行的自动调整抵消掉。最初，我们在脚本中进行了一些特别的调整并且获得了成功。我们重写了一些场景以在很多地方支持投降。

在游戏即将发布时，QA 报告说大多数投降场景都出问题了。第一个问题，是那些敌对的怪物在投降 180 秒以后会重新变成敌对的，因为那时候我们对个人声望进行的调整已经失效了；第二个问题，是游戏设计人员不知道应该对个人声望进行多大的调整才能确保在一定时间内保持友善。基本上，如果在声望系统中没有程序员的帮助，我们很难实现投降情节。

初始，程序员提供了一个脚本函数来清除所有对个人声望进行的调整，但是这个系统仍然不能正常工作。他们所做的个人声望调整需要很长时间才会消失，有时甚至长得连 PC 都觉得这个人无论如何得死了，因此投降的 NPC 就呆呆地等着 PC 杀死他。为了解决这个问题，我们在游戏引擎中通过 SetIsTemporaryFriend/Enemy()这两个脚本命令提供了不会消逝的永久个人声望调整。下面是一个新版本的 EvaluatePersonalReputation()，它实现了不会消逝的声望：

```
float DECAY_TIME = 180.0f;

int Object::EvaluatePersonalReputation(object oTarget)
{
    int nPersonalReputation = 0;
    CPersonalReputationEntry *pEntry;
    pEntry = FirstPersonalReputationEntry(oTarget);
    while (pEntry != NULL)
    {
        if (pEntry->Decays() == FALSE)
        {
            nPersonalReputation += pEntry->Adjust;
        }
        else
        {
            int nTimeRemaining = nTimeExpiry - GetCurrentTime();
            if (nTimeRemaining > 0)
            {
                float fRemainRatio = nTimeRemaining / DECAY_TIME;
                nPersonalReputation += pEntry->Adjust * fRemainRatio;
            }
        }
        pEntry = NextPersonalReputationEntry(oTarget);
    }
}
```

6.5.4 玩家对战的设置

拥有追随者（henchman）、召唤兽等来帮助玩家进行战斗是《龙与地下城》的一个重要特色。在声望系统中，追随者和召唤兽始终使用他们主人的声望，并且当一个主人和他的追随者结成同盟以后他们之间的关系永远是友善的。

在《无冬城之夜》中有三种不同类型的区域：没有玩家对战、基于群组的玩家对战和完全的玩家对战。在没有玩家对战的区域中，所有玩家角色和他们的群组成员之间禁止发生任何敌对行为（譬如说使用剑进行攻击、使用地对魔法或者偷窃）。如果一个区域允许基于群组的玩家对战，不在同一群组之中的玩家之间可以发生敌对行为。在完全支持玩家对战的区域中，即使是出于同一群组中的成员（譬如说玩家和他的追随者或是其他玩家）也会被玩家的魔法所伤害。

在像《暗黑破坏神 II》或是《无冬城之夜》这样的游戏中，可以通过“玩家关系”面板来控制是否仇视其他玩家。虽然从 PC 的角度来说，喜欢或者仇视另一个玩家对游戏的影响不大，但是对于 PC 的追随者来说这具有很大的影响，因为它们并不是由 PC 直接控制的。如果一个 PC 仇视另一个 PC 并且他们同处于一个允许敌对行动的玩家对战区域中，他们的追随者会在看到另一个 PC 或是其追随者时立即发起攻击。在 GetFullReputation 方法中包含了对 PC 以及他们的追随者之间的喜欢或仇视关系进行的处理。

```
int Object::GetFullReputation(object oTarget)
{
    // Henchmen do not have reputations of their own.
    if (GetMaster() != NULL)
    {
        return GetMaster()->GetFullReputation(oTarget);
    }
    // To see if we like the henchman, examine the master.
    if (oTarget->GetMaster() != NULL)
    {
        return GetFullReputation(oTarget->GetMaster());
    }

    // We are always friends of ourselves.
    if (oTarget == self) { return 100; }

    // Do we need to use the player-versus-player tables?
    // If not, use the NPC calculations.
    if (GetIsPC() == FALSE || oTarget->GetIsPC() == FALSE)
    {
        return GetFullReputation(oTarget);
    }

    // At this point, both self and oTarget are player chars.
    //
    int nPVPSetting = GetAreaPVPSetting();
    bool bInSameParty = (GetInParty(oTarget) == TRUE);
```

```
    bool bPCLiked = GetPCLiked();

    // GetPCtoPCReputation lookup, see which of the 12 possible
    // values applies, and sets the reputation based on the table.
    nReputation = GetPCtoPCReputation(nPVPSetting, bInSameParty, bPCLiked);

    return nReputation;
}
```

6.5.5 总结

前面介绍了在为《无冬城之夜》创建一个令人满意地声望系统时所进行的各种不同的尝试。相信这个系统相对来说还是易于理解的，无论游戏设计人员还是最终用户都可以方便地编写代码对 NPC 的行为进行自动控制；在向最终用户解释这个系统时也很简单。声望系统的绝大部分（除了玩家关系面板）外，都可以不经修改地直接用于其他 MMP 游戏的实现。

创建声望系统是非常复杂的任务，《模拟人生在线》使用了一支由三个程序员组成的团队来为实现游戏的声望系统。《创世纪在线》经过了很多次修改才形成了目前的声望系统。对在《无冬城之夜》中所使用的系统的介绍可能会让读者觉得创建一个声望系统很简单，但是千万不要低估所需要付出的努力。如果你需要在下一个项目中实现一个声望系统，那么不妨把这本书放在制作人的鼻子下面让他（她）将估计的在声望系统上的用时加倍。

6.5.6 参考文献

[Alt02] Alt, Greg and Kristin King, "A Dynamic Reputation System Based on Event Knowledge," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Grond98] Grond, G. M. and Bob Hanson, "Ultima Online Reputation System FAQ," Origin Systems, 1998, http://www.uo.com/.

6.6 城邦政府在在线社区中的作用

Artie Rogers, Inevitable Entertainment
awrogers@texas.net

当我们为了能够向玩家群体提供一个长久稳定的游戏社会而对 MMP 游戏中的某些系统进行设计时，我们有必要参考真实世界中的社会规则来了解究竟是什么原因使得人们聚集在一起并成为社会的一分子。通过观察真实社会和政府的作用，我们可以选取一些能够在设计在线政府基本框架时用得到的特征。我们可以选用真实世界中民主制度的一些重要思想来为在线政府创建一个稳固的基础。这些思想包括：创建一系列法律的职能、向公民征税的职能、使用通过税收收集到的资源来为公民提供保护以及其他利益的职能、公民选举领袖的职能。那些稳定的民主制度所拥有的规则也是一个稳定的在线政府应该具有的规则。

创建一个强大的在线政府系统是解决一些 MMP 游戏常见问题的重要一步。通过让那些休闲玩家感到他们归属于游戏中的大型社会，通过给予资深玩家对所处游戏环境施加影响的途径，在线政府可以稳定玩家群体。这种在线形式的政府不仅仅是基于种族或者区域的松散结合的联盟，它也不仅仅是一个超级行会（metaguild，可以使多个行会互相结盟并且为了共同的目标奋斗）。它是一个真正的政治系统，它可以制定法律、征服土地、经历政治斗争并且有助于建立起强烈的社会感。

在开放式的持久状态（open-ended, persistent state）世界中，玩家杀手（player killer）是一个常见的问题。之所以玩家杀手会成为问题，其主要原因之一就是很少有游戏能够让玩家群体保护自身不受到玩家杀手的伤害。如果一个社会能够在面临外部威胁时有效地保护自己，那么在其社会成员之间就能发展起更为密切的关系，因为他们曾经共同努力去克服困难。在与玩家杀手进行斗争的过程中存在两个主要问题：缺乏手段去回报那些抗击玩家杀手的玩家，缺乏工具对玩家杀手进行有效的控制。我们在设计在线政府时就考虑到了这点。所采用的解决方法是在那些玩家杀手头上自动生成一些赏金，并且给予相关玩家所需的工具来限制玩家杀手的影响。通过提供合适的赚钱机会和工具，控制玩家杀手成为赏金猎人们的一个可行的游戏职业。我们将在后面详细介绍赏金猎人这个职业。

本文通过追踪玩家在政府系统中的三种基本参与方式来详细介绍一个在线政府的设计实例。从那些刚刚进入游戏的休闲玩家开始，简要介绍政府怎样立即给他们带来群体的认同感。随后将介绍玩家在政

府中所起的功能（包括选举权），并对政府中的职位示例（譬如说赏金猎人）进行描述。最后，本文将详细介绍统治者的义务、权力、责任以及一个玩家必须怎么做才能成为统治者。

6.6.1　公民生涯

在线政府系统中，民主城邦政府是最强大的形式。城邦不仅是指那些玩家们聚集在一起进行社交和贸易的城市区域，它还包含附近的区域，在那里玩家们可以采集原料或是进行冒险。作为一个社会政治系统，城邦不仅可以让玩家们对他们周围的游戏环境施加影响，还可以为其公民在游戏时提供导引。

在那些具有很强的在线政府形式的 MMP 中，一个新玩家的游戏生涯始于选择他们的国籍。那些给新玩家的游戏经历带来困扰的问题通常都是由于他们感到没有目标或是无法参与周围的游戏活动。玩家们（尤其是那些第一次进行 MMP 游戏的玩家）通常会因为游戏的开放性而感到困惑。与通常在进行单人游戏时不同，他们在多人游戏中缺乏导引。他们常常会感到与身边的游戏行动脱节。通过赋予他们城邦政府的国籍，玩家从登录到游戏中的那一刻起就成为这个大型在线社会的一部分，这可以让他们获得一些归属感，让他们觉得自己属于游戏中的群体。城邦政府还可以为玩家提供一些导引，因为在那里总是有任务需要完成。无论参与度如何，城邦的公民（不管是那些休闲玩家还是最活跃的在线政客）都可以享受到归属和导引所带来的社会优越性。

正如在设计任何在线游戏系统时都需要考虑的一样，我们有必要使用某些限制和规则来保持城邦公民社会的流动性不会过大，并且确保它不会被玩家滥用。那些恶意玩家会试图滥用游戏开发人员给予玩家群体的自由来达到他们自己的目的。恶意玩家也可能试图通过使用这些自由来破坏各种政府系统，譬如说选举或是赏金猎人的运作，他们这样做只是为了证明他们在游戏中的存在，而不是为了任何个人利益。我们会对可以解决这个问题的各种方法进行讨论。暂不考虑这些问题，一个新玩家的游戏生涯始于选择他的国籍。

1. 国籍

国籍是任何政府的基础。在创建一个在线政府时，我们必须对国籍的一些重要方面进行考虑，这包括获得国籍的过程、一个现有的玩家修改国籍所需的步骤、对国籍进行滥用带来的问题以及对策。

新玩家在第一次登录游戏时就会被授予国籍。我们根据他们在哪里开始游戏来赋予他们特定的国籍。在创建角色时，玩家可以在做出选择前查看每个城邦的一些简单数据。下面是这类数据的一些例子：

- 城邦总资产；
- 总人口；
- 所占据的地区；
- 完成的任务；
- 公民的平均声望。

不同类型的 MMP 游戏所提供的数据类别也不同。这些信息可以为玩家提供正确选择国

籍所必要的信息。如果玩家对国籍无所谓并且希望游戏为他选择，那么游戏就会自动地把玩家在现有的城邦政府之间平均分配。玩家还可以选择以流浪者的身份进入游戏并且在游戏中进行选择。不过我们还是鼓励玩家在创建角色时就选择国籍。

对于那些要成为其他城邦公民的玩家来说，他们必须先放弃目前的国籍进入流浪状态。我们会给玩家一些时间来对他们所做的选择进行仔细考虑。在此期间，他们仍然可以获得目前国籍所带来的权益。一旦过了这段时间，他们将成为流浪者。给予玩家一些考虑的时间非常重要，因为重新获得国籍是一个既困难又花费巨大的过程，因此玩家必须确定这是他们想要做的。一旦流浪玩家想要成为另一个城邦的一部分，他们必须先找到一个担保人。这个担保人必须是他们想要加入的城邦的公民。一旦获得担保，玩家就可以请求入籍。这一过程不仅需要一定的时间，还需要在游戏中支付一定的费用，这可以由统治者决定。这一费用可能只是一定量的游戏货币，也可以是一定数量的原料。统治者可以根据城邦的需要来设定入籍费用。一旦等待期过去了并且玩家已经支付了这一费用，他就可以获得作为一个公民所能获得的所有权益。

我们可以通过让改变国籍的过程变得更加困难并且限制每个游戏账号只能拥有一个国籍来防止游戏社会中的流动性过大，有很多原因使我们要这样做。通过巧妙地阻止玩家改变国籍，我们可以给玩家一定时间来建立他们对城邦的忠诚度，这种忠诚度是随着玩家对周围的事物越来越熟悉而慢慢形成的。除了前面所介绍的对改变国籍所做的限制以外，我们还需限制每个游戏账号只能拥有一个国籍。通过对游戏账号的国籍以及国籍的改变进行限制，可以避免一些由于滥用而引起的问题，譬如说干扰政府选举和城邦的运作。为了破坏选举系统，恶意玩家可能会创建“傀儡”角色来操纵选举向有利于他们自己的候选人的方向进行，甚至可以进行纯粹的破坏活动。仅仅为了选举的目的而创建傀儡角色会稀释那些诚实玩家的选票，诚实玩家不会因为每个角色都可以进行选举并且拥有国籍就滥用这一自由。如果玩家可以在一个游戏账户上拥有多个国籍，恶意玩家可能会试图破坏赏金猎人们对玩家杀手的控制。要实现这点，他们只需要通过创建傀儡赏金猎人并且假借对城邦的忠诚来和那些诚实的赏金猎人进行对抗，甚至还可以帮助玩家杀手们。有一个方法可以解决基于角色的选举和基于角色的国籍所带来的问题，那就是只有账号才能获得国籍，并且自动延伸到属于这个账号的角色。选举权应该与玩家的付费账号相关联，而不是玩家的角色。由于这个要求，如果一个玩家想要给自己更多的选票，那么就必须在真实世界中付出与此相关的费用：他们必须为每张选票建立一个新的付费账号。通过让国籍变得更有价值并且更难改变，我们可以解决在线政府系统中可能出现的一些滥用问题。

2. 国籍可能带来的权益

在游戏中城邦国籍可以带来一些切实的权益。这些权益包括可以从玩家以及周围环境获得保护，可以根据角色的能力获得奖励，并且可以参与与城邦相关的行动和任务。不仅是那些积极参与游戏的玩家能够获得这些权益，即使是对城邦政治没有兴趣的玩家也会对它们感兴趣。

国籍带来的某些权益同时和城邦的核心特征以及城邦所占据的区域的核心特征有关。有些城邦更倾向于特定的技能，因此他们的成员可以在这些技能上获得奖励。譬如说，如果一个城邦倾向于魔法，所有现行的公民都能够在魔法技能上获得奖励；如果一个城邦倾向于战

斗，它的公民可以在战斗技能上获得奖励。这些角色所能获得的权益还与城邦的扩张有关。一旦一个城邦占据了地图上的特定区域，它的公民就可以获得与这个区域相连的技能奖励。公民还可以以进入那些只有占据了这一区域的城邦的公民才能进入的采集、狩猎或是冒险区域等形式享受基于区域的权益。通过把区域和某些权益联系起来，我们可以鼓励城邦的玩家在这一区域聚集，从而进一步加强公民之间的联系。

还有一些公民权益与参与政府工作相关。城邦会发布符合其利益的任务，玩家在完成这些任务后可以获得经验。这些随机任务可能是杀死特定的 NPC、获得资源或是探索特定区域。所有这些任务都是为了达成城邦的某些目标而建立的，譬如说消灭某个原料采集区域所有惹麻烦的 NPC，获取建造建筑物所需要的资源或是探索附近某个可能值得占领的区域。通过完成这些由城邦发布的任务，玩家为他们社会的权益而工作，这样他们冒险行动的目的将更有意义。

3. 国籍小结

城邦系统可以给玩家群体带来很多好处。它在把新玩家引入游戏中起到了重要的作用，它使新玩家会立即成为社会群体的一部分。新玩家获得任务后，就可以有机会通过完成这些任务来帮助政府向游戏中的目标迈进。能够选举并且参与城邦工作使得新玩家可以获得认同感，同时城邦在游戏中向玩家提供可预见的、会对游戏产生广泛影响的工作目标。由城邦带来的包容和接受的社会利益总是显而易见的。在基于城邦的特性带来利益（会随着城邦的扩张和缩减而改变）的同时，玩家始终要牢记城邦政府的存在，因为他们的财富将受到所属城邦成败的影响。城邦的设计采用了向玩家提供保护和利益的思想，并且赋予其公民选举的权力，从而向玩家传递游戏中的好处。

6.6.2　参与城邦工作

一个强健的在线政府系统可以为玩家们提供大量的参与机会。让公民参与政府工作最基本的方式之一就是让他们可以选举他们的领袖。即使是那些休闲玩家或是对游戏不感兴趣的玩家也有时间在他们登录游戏的时候投票。有大量不同层次的参与方式可供玩家选择：他们可以协助向新玩家介绍游戏，也可以更积极地参与，譬如说担任城邦中的某个职位。下面是一些可能会在中世纪游戏设置中出现的职位。

- 摄政者：他们待在议会里面，代表玩家创建的城市。
- 贤人：他们会被赋予与城邦的核心特征相一致的独特能力，对于城邦的扩张非常重要。
- 赏金猎人：他们可以控制玩家杀手或是帮助公民消灭 NPC 敌人。

在线政府的存在还使得开发人员有机会根据玩家在政府中的参与程度建立另一条升级路径。参与程度可以包括选举或者成为统治者。

1. 选举

选举是个人参与民主政府最基本的形式。这一事实同样也存在于在线城邦政府。每个玩家在成为城邦公民后都可以参与选举他们的领袖。玩家会在登入游戏时获得选票，他们可以

选择投票或是放弃选票并继续进入游戏。玩家可以访问关于每个候选人的简单数据以及这些候选人提交给系统的选举宣言。因为玩家需要克服不少困难才能成为候选人，因此对选举宣言系统进行滥用的可能性很小。要进一步降低滥用的可能，可以通过限制候选人人数，使用简单的脏话过滤器以及提供候选人账户信息，在可能的情况下让游戏管理员检查这些宣言等方式。一旦玩家对此系统进行滥用，我们就会对其进行惩罚。候选人数据包括候选人属性的详细列表，如游戏中的成就、声望以及拥有的土地。在选举中获得多数选票的人将成为胜利者，选举的结果不仅会在游戏中进行广播，还会显示在网页上。选举行动可以让公民们有一些参与感并且能够对他们政府的发展方向产生一些影响，而这一切并不需要他们付出很大的努力。

为了防止由于恶意玩家对选举过程进行干扰而导致的问题，我们必须采取一定的手段来稳定选民。正如前面提到的，对国籍的限制可以减缓人口的改变，从而减少那些想要破坏选举结果的玩家所造成的影响。如果不想阻止玩家改变国籍，我们还可以采用其他的方法：我们可以根据成为公民的时间来限定是否拥有选举权；或者建立一个基于公民身份的方案，而不是把限制建立在选举行为本身。这样我们就解决了投票中会产生的问题并且使所有的公民都能够参与投票，这一点是至关重要的。

2. 政府职业的例子：赏金猎人

城邦政府最重要的好处，就是它可以保护玩家不受玩家杀手或是其他环境危害的影响。要做到这点，我们需要创建一个由系统支持的职位，并向玩家提供工具和回报，从而使控制玩家杀手的行为成为一个可行的职业。赏金猎人就是这样一个职位。在这一小节中，我们将详细介绍玩家成为赏金猎人所要经历的步骤，同时对他们的职责以及怎样才能履行这些职责加以说明。最后，我们还将讨论赏金猎人达成目标所需要的回报和工具。

赏金猎人通常由玩家担任，如果愿意成为赏金猎人的玩家不多，NPC 也可以从事这一职务。为了便于讨论，本文将着重对由玩家担任的赏金猎人进行讨论。

赏金猎人是由统治者任命的。玩家的角色必须事先达到一定的级别或者完成某些特性的任务。玩家在成为赏金猎人前还必须已经保持这个城邦的公民身份达到一定时间。在申请成为赏金猎人时，玩家还必须支付一定的费用。这一申请过程可以通过游戏中的公告栏进行，公告栏还应该能强制这一职位的基本要求。统治者还可以有选择性地通过时间或者信息的总量来进一步限制对公告栏的访问。一旦玩家支付了相应的费用，他们要发出赏金猎人职位的申请。如果这一玩家符合我们对公民权、等级以及费用的要求，这一申请就会被转发给统治者。统治者可以访问角色以及帐户信息（譬如说账户存在时间、各种游戏中的成就，甚至可以通过玩家/管理员对这个玩家的评价来得知这一玩家是否有反社会行为）来更好地了解申请人。一旦对所有的信息审查完毕，统治者就可以授予玩家这个职位。

正如前面所说的，赏金猎人的主要任务就是保护公民。这种保护中有很大一部分是对游戏环境的保护。要让赏金猎人可以在野外帮助玩家，就要把玩家和赏金猎人关联起来。这种关联非常重要，因为只有这样才能确保玩家可以及时地获得帮助。我们可以通过很多方式来解决这个问题。例如，让玩家在有需要时呼叫赏金猎人。在玩家向赏金猎人发出求助信息后，他们可以瞬间移动到玩家身边。为了避免对这一功能的滥用，玩家在发出呼叫时需支付一定的费用。

为了鼓励赏金猎人在野外扩大巡逻的范围，可以在狩猎或资源采集区域设置一些动态路点。每当赏金猎人移动到某个路点上时，他们可以获得一定的报酬。这样一来，赏金猎人就会到那些他们平时不会去的区域巡逻。

城邦政府的公民遇到的另一个威胁就是玩家杀手。公民依靠赏金猎人来保护他们免遭玩家杀手的侵害。游戏系统会向任命的赏金猎人提供一系列工具来定位、跟踪并最终消灭城邦的敌人；这些敌人可以是玩家杀手，也可以是敌对城邦的成员。这些工具还可以让赏金猎人能够方便地与公民以及其他赏金猎人互相交流并且移动到他们身边。赏金猎人还能够借助工具来识别玩家杀手并且对其进行跟踪。赏金猎人还能够将某个非公民的玩家杀手角色驱逐出城邦并且在一定时期内不能返回。只有满足特定条件的玩家才能被驱逐。这些条件由统治者设定，它可以包括那些基于声望、行会关系以及所杀死的玩家数量的角色信息。

保护玩家在游戏中并不是一个很好的职位。本质上说，大多数玩家都希望可以做些有利于自身角色的事情。他们愿意帮助公民，但是这不能以牺牲自己的将来为代价。另一方面，玩家杀手是一个利润非常丰厚的职位，因为他们可以杀死弱小的玩家，而玩家携带的物品通常比游戏中的怪物多。为了解决这个问题，城邦统治者可以向赏金猎人提供金钱来奖励他们为玩家提供的服务、杀死玩家杀手（奖励由被杀死的玩家的声望或是杀害玩家的总数决定）以及巡逻（奖励由巡逻这个区域的难度决定）。这样，赏金猎人所付出的时间和努力就会获得大量的回报。

在城邦为了扩张势力而要占领其他区域时，我们也需要赏金猎人的服务。扩张领土是一个非常重要的事件，作为城邦的领袖，统治者也可以参与这一行动。统治者可以召集赏金猎人来帮助占领土地。并且，必须要有相当数量的赏金猎人才能占领一个区域。

当然，我们还需要某些措施来确保好的赏金猎人可以获得回报而那些较差的则会被从赏金猎人队伍中清除。通过系统追踪赏金猎人，很容易就能识别出那些滥用这一职位的人。有些玩家成为赏金猎人只是因为他们希望获得这个职位带来的特权和能力，而不是为了履行相应的职责。因此我们必须对赏金猎人的所有行为进行记录从而使统治者可以发现那些滥用这一系统的玩家。方法就是让呼叫赏金猎人的玩家能够对他们做出评价，还可以让统治者查看与赏金猎人相关的数据，譬如说：

- 获得的呼叫总数；
- 忽略的呼叫数量；
- 杀死敌人的数量；
- 到达过的路点数量；
- 回复过的帮助请求总数。

每当一个职位具有特权并且这一特权没有内在的回报时，就必须采取一些防范措施。由于可以获得一系列的工具和奖励，对那些愿意帮助热爱和平的公民远离危险或是希望帮助城邦扩张的玩家来说，赏金猎人是一个极具吸引力的职位。

3. 参与城邦工作的小结

让玩家能够自行参与城邦政府的工作可以让他们感到自己是社会的一份子。选举是参与政府工作的最基本、最重要的步骤。它让每个玩家都能对政府的发展方向起到一定影响。玩家可以在城邦中获得某个职位（譬如说赏金猎人）来进一步参与政府的工作。让玩家可以通

过某种形式去向社会系统提供帮助有助于让玩家感到自己被社会包容和接纳，并进一步为他们提供在游戏中的导引。通过向那些愿意为社会作出贡献的玩家提供必要的工具和报酬，我们可以确立一些可行的游戏职业。社会联系是每个 MMP 游戏必不可少的组成部分，让玩家能够帮助改善他们的社会环境对于建立和加强社会联系来说非常关键。

6.6.3 定义政治过程

城邦政府系统中最重要的职位就是统治者。接下来将讨论玩家怎样才能成为统治者的候选人以及公民怎样通过选举来决定他们的统治者。本文还将介绍玩家群体在罢免统治者时需要的过程，以及介绍统治者的权利和义务。

1. 成为统治者

玩家必须满足一定的条件才能成为统治者的候选人，譬如说其游戏账户必须已经注册了一定时间或是他成为特定城邦的公民超过了一定时间。一旦他满足了最基本的要求，接下来还需得到公民的支持。要做到这点，这个玩家必须让一定数量的玩家宣誓支持他成为候选人。宣誓支持他的玩家数量必须达到城邦总人口的某个特定百分比，这个数字随着城邦人口的改变而上下浮动。不仅如此，在宣誓支持时，还必须付出一定的代价（这可能是一定数量的金钱或资源），这可以让那些公民仔细考虑他们的决定。我们还可以对特定玩家在一定时间内进行宣誓支持的次数加以限制，也可以给每一份支持一个特定的价值，这个价值随着给予支持的玩家在特定时间段内放弃的支持而减少。这样做是很有必要的，否则玩家在进行支持时就不会仔细考虑。在特定的 MMP 中，还可以加入其他要求，譬如说要求候选人曾经担任过一段时间的工会领袖，并且这个工会要具有一定的规模。我们也可以要求候选人具有一定的声望或是具有一定的技能和等级。一旦玩家满足所有的要求，他们就可以以候选人的身份参加选举。

一旦玩家成为候选人，他们必须决定是否接受相应的条款。这些条款不仅要求玩家支付一定的费用，还要求他们向选民公开游戏账户的历史。譬如说，要求玩家根据城邦的总人口或是总资产捐献一定量的金钱。这一费用必须足够高，这样才能确保玩家是认真地对待选举的，并且这笔费用还要随着游戏中的通货膨胀而增加。我们并不要求玩家公开任何真实世界中的个人信息，但是我们会公开这个账户所有角色的行为以及这个账户所有者的总体趋势以提供与该候选人有关的信息。玩家接受这些条款以后，他们可以决定是否提交选举宣言。这一宣言的目的是向选民介绍他们成为统治者以后的治理计划，从而让选民们投票给他。当他们提交了选举宣言以后，就可以参加选举了。

选举会持续一定的时间，譬如说一个月。在一个月中进行城邦选举可以避免很多玩家在同一时间投票。让选举持续很多天是非常重要的，如果选举只持续一天，游戏服务器的压力就会很大。不仅如此，权力的转移会让城邦的成员感到焦虑，因此我们必须根据玩家群体的焦虑程度对权力转移中发生的改变和混乱进行控制。一旦一个玩家被选举为统治者，他将立即获得统治者应有的权利和义务。玩家担任统治者的任期是有限制的。有很多方法可以对此加以限制，例如限制连任的次数，譬如说连任不能超过两次；也可以限制一定时期内玩家担任统治者的次数，譬如说每年不超过三次。