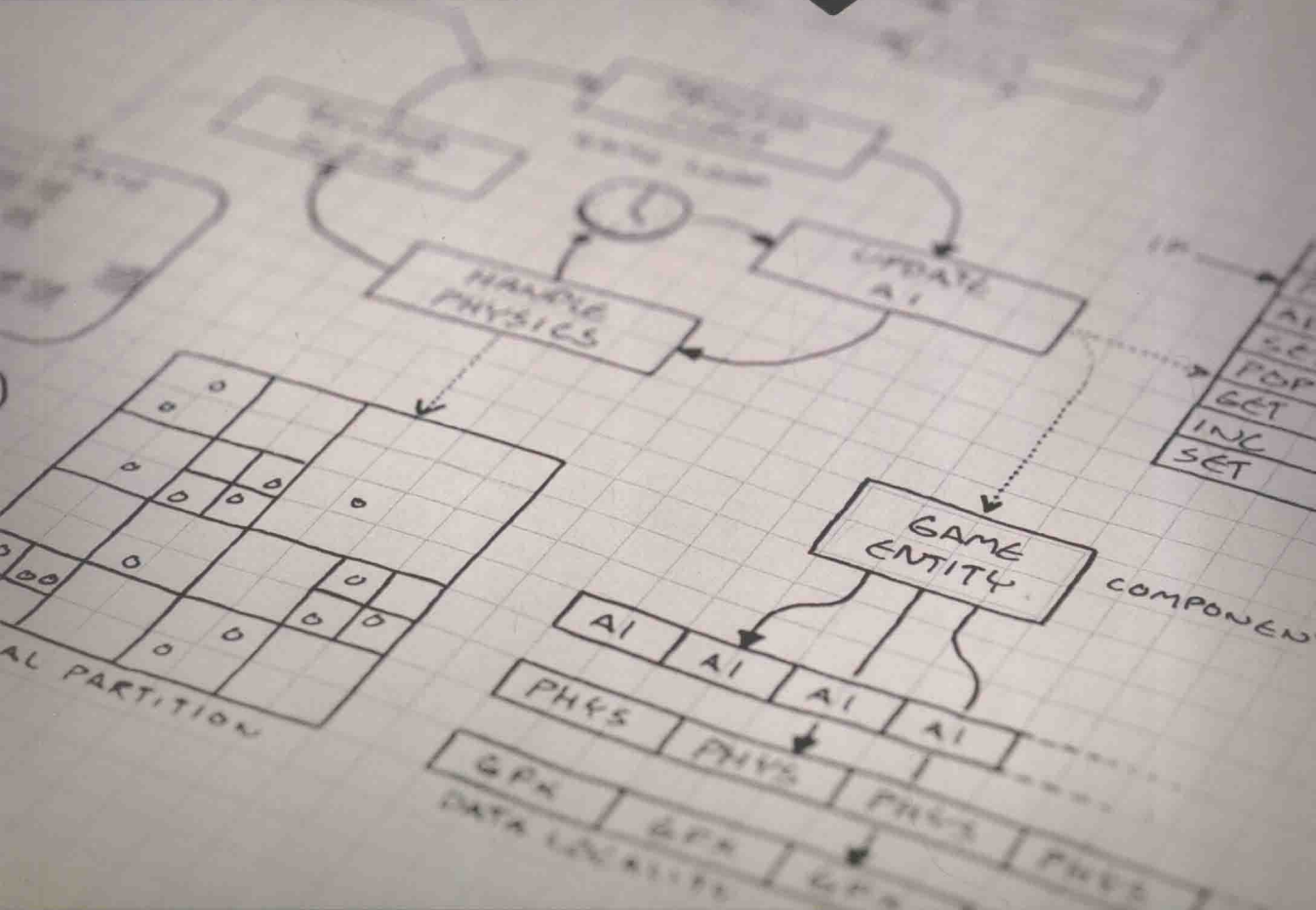


前EA著名游戏工程师经验凝结
4大类13种游戏编程模式精彩呈现



Game Design and Develop
游戏设计与开发



Game Programming Patterns

游戏编程模式

[美] Robert Nystrom 著
GPP翻译组 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

有关此电子书的说明

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融等等。质量都很清晰，为方便读者阅读观看，每本100%都带可跳转的书签索引和目录，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ1779903665。

PDF代找说明：

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF，大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系 QQ 1779903665.

备用:QQ 461573687

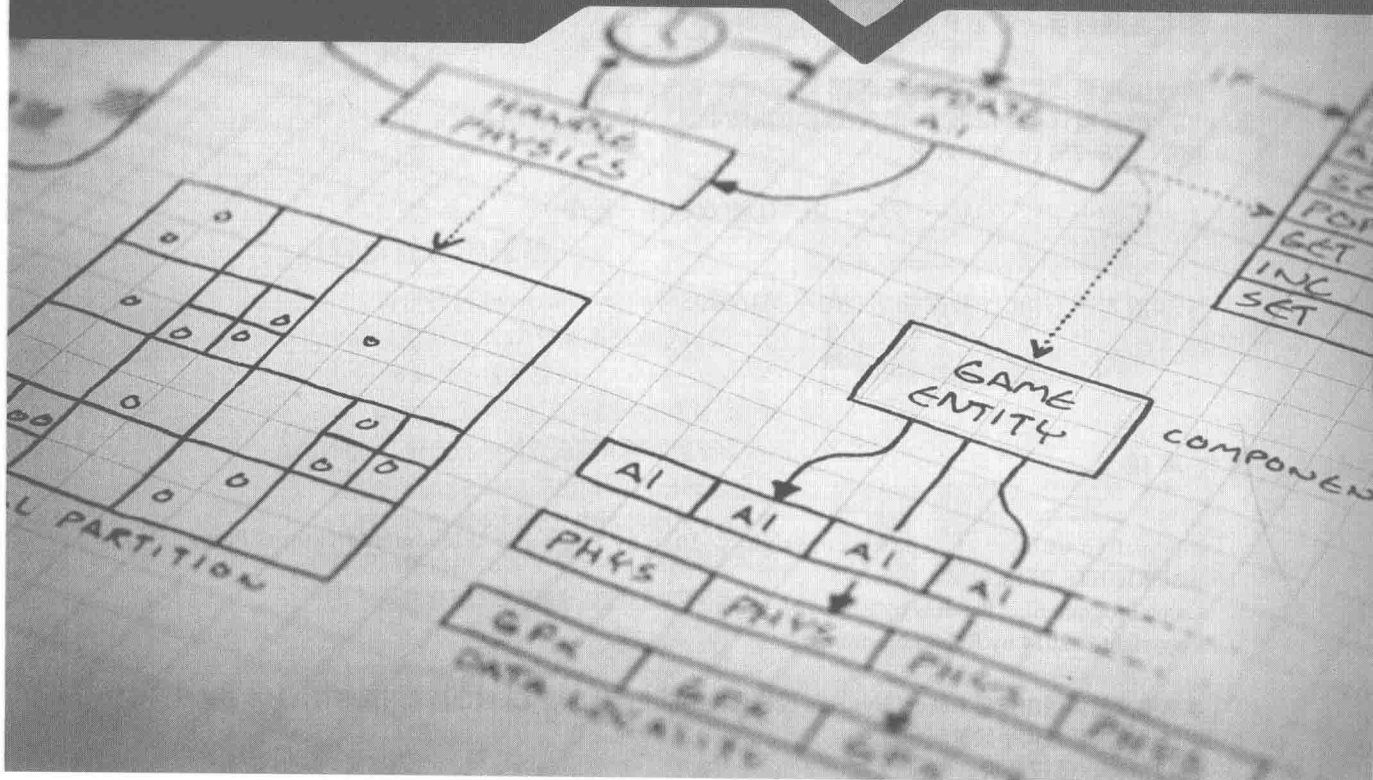
若以上联系方式失效，您可通过以下电子邮件获取有效联系方式。

E-mail : ebooksprite@163.com

E-mail : ebooksprite@foxmail.com

若您没有QQ通讯工具，请发送您的请求到 ebooksprite@gmail.com 与客服取得联系。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找和后期制作pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。



游戏编程模式

[美] Robert Nystrom 著
GPP翻译组 译

人民邮电出版社
北京

图书在版编目 (C I P) 数据

游戏编程模式 / (美) 尼斯卓姆 (Robert Nystrom)
著 ; GPP翻译组译. — 北京 : 人民邮电出版社, 2016.9
ISBN 978-7-115-42688-8

I. ①游… II. ①尼… ②G… III. ①游戏程序—程序设计 IV. ①TP311.5

中国版本图书馆CIP数据核字(2016)第160703号

版 权 声 明

Simplified Chinese translation copyright © 2016 by Posts and Telecommunications Press
ALL RIGHTS RESERVED
Game Programming Patterns by Robert Nystrom
Copyright © 2014 by Robert Nystrom

本书中文简体版由作者 **Robert Nystrom** 授权人民邮电出版社出版。未经出版者书面许可, 对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有, 侵权必究。

-
- ◆ 著 [美] Robert Nystrom
 - 译 GPP 翻译组
 - 责任编辑 陈冀康
 - 责任印制 焦志炜
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <http://www.ptpress.com.cn>
 - 三河市海波印务有限公司印刷
 - ◆ 开本: 800×1000 1/16
 - 印张: 20.75
 - 字数: 437 千字 2016 年 9 月第 1 版
 - 印数: 1—3 000 册 2016 年 9 月河北第 1 次印刷
 - 著作权合同登记号 图字: 01-2015-1268 号

定价: 69.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316
反盗版热线: (010)81055315

内容提要

游戏开发一直是热门的领域，掌握良好的游戏编程模式将是开发人员的必备技能。本书细致地讲解了游戏开发需要用到的各种编程模式，并提供了丰富的示例。

全书共 6 篇 20 章。第 1 篇概述了架构、性能和游戏的关系，第 2 篇回顾了 GoF 经典的 6 种模式。第 3 篇到第 6 篇，按照序列型模式、行为型模式、解耦型模式和优化型模式的分类，详细讲解了游戏编程中常用的 13 种有效的模式。

本书提供了丰富的代码示例，通过理论和代码示例相结合的方式帮助读者更好地学习。无论是游戏领域的设计人员、开发人员，还是想要进入游戏开发领域的学生和普通程序员，都可以阅读本书。

作者简介

Robert Nystrom 是一位具备超过 20 年职业编程经验的开发者，而其中大概一半时间用于从事游戏开发。在艺电（Electronic Arts）的 8 年时间里，他曾参与劲爆美式足球（Madden）系列这样庞大的项目，也曾投身于亨利·海茨沃斯大冒险（Henry Hatsworth in the Puzzling Adventure）这样稍小规模的游戏开发之中。他所开发的游戏遍及 PC、GameCube、PS2、XBox、X360 以及 DS 平台。但最傲人之处在于，他为开发者们提供了开发工具和共享库。他热衷于寻求易用的、漂亮的代码来延伸和增强开发者们的创造力。

Robert 与他的妻子和两个女儿定居于西雅图，在那里你很有可能会见到他正在为朋友们下厨，或者在为他们上啤酒。

译者简介

GPP 翻译组是一群游戏开发技术爱好者为了翻译本书简体中文版而成立的一个兴趣小组。GPP 小组的成员如下：

赵卫兵（ChildhoodAndy）

游戏开发爱好者，曾从事游戏开发，《Chipmunk2D Physics》官方文档译者，泰然网成员之一。崇尚开源、分享精神，目前就职于 58 同城。

屈光辉（子龙山人）

Cocos2d-x 核心开发者，Cocos Creator 核心开发者，《Cocos2D 权威指南》第二作者，泰然网早期创始成员之一，Cocos2d 社区知名博主，emc-china.org 创始人。专注于移动游戏开发和游戏 UI 框架开发及优化。

郑炯彬

“90 后”，香港科技大学研究生在读，视觉算法程序员，户外爱好者，自认为最离不开三样东西：书、音乐、NULL。

陈侃

游戏开发者、游戏爱好者。同样热爱文字和美术，致力于富有创造力和艺术性的工作。

姜召阳

从事移动游戏开发行业，一枚文艺帝都程序员，平时喜欢参与开源项目、读书和切磋篮球。

特别感谢其他的译者：许新星、唐宏洋、张植臻和洪孝强。

前言

在五年级的时候，我和我的小伙伴们获准使用一个放置着几台非常破旧的 TRS-80s¹ 的闲置教室。为了激励我们，一位老师找到了一份印有一些简单 BASIC 程序的打印文档给我们。

当时，计算机上的音频磁带驱动器是坏掉的，所以每次我们想要运行一些代码的时候，都不得不仔细地从头开始键入代码。这使得我们更喜欢那些只有几行代码的程序：

```
10 PRINT "BOBBY IS RADICAL!!!"  
20 GOTO 10
```

即便如此，整个过程还是充满了艰辛。我们不懂得如何编程，所以一个小的语法错误便让我们感到很费解。程序出毛病是家常便饭，而此时我们只能重头再来。

在这叠文档的最后部分，是一个真正的“怪物”：一个代码量占据几页篇幅的程序。我们思量良久，这才鼓起勇气去尝试它，不过它极为诱人——标题写着“巨魔洞穴”。我们不知道它是做什么的，不过听起来像是个游戏，还有什么能比亲手写一款计算机游戏更酷呢？

我们从没让这个程序真正运行起来过。一年后，我们搬出了那个教室（后来当我了解了一点 BASIC 时，才知道那只是一个供桌面游戏使用的角色生成器，而并非一款完整的游戏）。命中注定，从那之后，我立志要成为一个游戏程序员。

在我十几岁时，我的家人搞了一台装有 QuickBASIC 的 Macintosh，之后又装了 THINKC。我几乎整个暑假都在那上面倒腾游戏。自学是缓慢而痛苦的。我能轻松地让一些代码运行起来（也许是一张屏幕地图或者一

如果计算机打印足够多的次数，或许它会神奇的变成现实哦²。

¹ 见[维基百科 TRS-80s](<http://en.wikipedia.org/wiki/TRS-80>)。译者注：TRS-80s 于 1977 年诞生，是第一批问世的微型计算机之一。

² 这里指的是计算机反复打印第 10 行代码的语句“BOBBY IS RADICAL!!!”，作者开玩笑地说会变成现实。

我的许多夏天都是在路易斯安那州南部的沼泽中捕蛇和乌龟来度过的。如果户外不是那么酷热的话，这很可能是一本爬虫学的书，而不是讲游戏编程的书。

这是我和这位朋友第一次见面，在5分钟自我介绍之后，我坐在他的沙发上，在接下来的几个小时里，我聚精会神地阅读而完全忽视了他。我感觉从那以后自己的社交能力还是至少有那么一丁点儿提升的。

个小型猜谜游戏)，但随着程序增大，编码变得越来越难。

起初，我的挑战在于让程序运行起来。后来，我开始琢磨如何编写超出我大脑思考范围的更大些的程序。我开始试图寻找一些关于如何组织程序的书籍，而不只是读一些关于“如何用C++编程”之类的书籍。

几年很快过去，一位朋友给了我一本书：《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software）。终于来了！这就是我从青少年开始便一直寻找的那本书！我一口气将它一字不漏地读完了。虽然我仍纠结于自己的程序，但是看到别人也如此挣扎并提出了解决方案，也如释重负。原本赤手空拳的我终于有工具可使了。

在2001年，我得到了自己梦寐以求的工作：EA（Electronic Arts）的软件工程师。我迫不及待地想看一下真正的游戏，以及工程师们是如何组织它们的。像Madden Football这样的大型游戏到底是个什么样的架构？不同系统之间是怎么交互的？他们是怎么让一套代码库在不同平台上运行的？

分解阅读源码是一种震撼人心且令人惊奇的体验。图形、人工智能、动画和视觉效果方面，都有十分出众的代码。我们公司有人懂得如何榨取CPU的每一个周期并加以善用。一些我甚至不知道能否实现的东西，这些家伙一个早上就能搞定。

但是这些优秀代码所依托的架构往往是事后想出来的。他们太专注于功能以至于忽视了组织架构。模块之间的耦合现象很普遍，新功能往代码库里见缝插针，而不顾其是否契合。这些所见令我幻想破灭，看起来许多程序员，就算他们心血来潮地翻开过《设计模式》一书，恐怕能看完单例就很不错了。

当然，也不是真的那么糟糕。我曾设想游戏程序员们坐在放满白板的象牙塔中，连续几周冷静地讨论代码架构的细节。实际情况是，我眼前这份代码是别人在紧张的期限里赶工出来的。他们尽了自己最大的努力，同时，我逐渐认识到，他们竭尽全力的结果通常是编写出了十分优秀的代码。我写游戏代码的时间越长，就越能发现隐藏在这些代码之下的可贵之处。

遗憾的是，“隐藏”一词往往说明了问题。宝藏埋在代码深处，而许多人正在它们之上路过（优秀的代码被许多人视而不见）。我看到过同事努力想改造出一个好的解决方案，那时，他们所需要的示例代码就隐藏在他们脚下的代码库之中。

这个问题正是本书力图解决的。我挖掘并打磨出自己在游戏代码中所发现的最好的设计模式，在此一一呈现给大家，以便我们将时间节省下来创造新事物，而不是重新造轮子。

市面上已有的书籍

目前市面已经有数十多本游戏编程的书籍。为什么还要再写一本？

我见过的大多数游戏编程书籍无非两类。

- **关于特定领域的书籍。**这些针对性较强的书籍带领你深入地探索游戏开发的一些特定方面。它们会教你 3D 图形、实时渲染、物理仿真、人工智能或音频处理。这些是众多游戏程序员在自己的职业生涯中所专注的领域。

- **关于整个游戏引擎的书籍。**相反，这些图书试图涵盖整个游戏引擎的各个部分。它们的目标是构建一整套适合某个特殊游戏类型的引擎系统，这类通常是 3D 第一人称射击游戏。

我喜欢这两类书，但我觉得它们仍留下了一些空白。讲特定领域的书很少会谈及你的代码块如何与游戏的其他部分交互。你可能擅长物理和渲染，但是你知道如何优雅地将它们拼合起来吗？

第二类书籍涵盖了这类问题，但我往往发现这类书通常都太过庞大、太过空泛。特别是随着移动和休闲游戏的兴起，我们正处在众多类型的游戏共同发展的时代。我们不再只是照搬 Quake¹了。当你的游戏不适合这个模型时，这类阐述单一引擎的书籍就不再合适了。

相反，这里我想要做的，更倾向于分门别类。本书的每个章节都是一个独立的思路，你可以将它应用到你的代码里。你也可以针对自己制作的游戏来决定以最恰当的方式将它们进行混搭。

这种分类讲解风格的另外一个例子，就是广受大家喜爱的《游戏编程精粹》系列。

本书和设计模式有什么联系

任何名字中带有“模式”的编程书籍都和经典图书《设计模式：可复用面向对象软件的基础》有所联系。这本书由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 编著（这 4 人也称为“Gang of Four”，即本书所提到的“GoF”四人组）。

¹ 《雷神之锤》，第一个真 3D 实时演算的 FPS 游戏。

设计模式一书本身也源自前人的灵感。创造一种模式语言来描述问题的开放性解决方案，该想法来自《A Pattern Language》，由 Christopher Alexander（和 Sarah Ishikawa、Murray Silverstein 一起）完成。

这是一本关于框架结构的书（就像真正的建筑结构中建筑与墙体和材料之间的关系），作者希望他人能够将其运用作其他领域问题的解决方案。设计模式（Design Patterns）正是 GoF 在软件领域的一个尝试。

本书的英文原名是 Game Programming Design Patterns，并不是说 GoF 的书不适用于游戏。恰恰相反，在本书第 2 篇中介绍了众多来自 GoF 著作的设计模式，同时强调了在它们游戏开发中的运用。

从另一面说，我觉得这本书也适用于非游戏软件。我也可以把这本书命名为《More Design Patterns》，但我认为游戏开发有更多迷人的例子。难道你真的想要阅读的另外一本关于员工记录和银行账户例子的设计模式图书吗？

也就是说，尽管这里介绍的模式在其他软件中也是有用的，但我觉得它们特别适合应对游戏工程中普遍会遇到的挑战，例如：

- 时间和顺序往往是一个游戏的架构的核心部分。事情必须依照正确的顺序和正确的时间发生。
- 开发周期被高度压缩。众多程序员必须在不牵涉他人代码、不污染代码库的前提下对一套庞大而错杂的行为体系进行快速的构建与迭代。
- 所有这些行为被定义后，游戏便开始互动。怪物撕咬英雄，药水混合在一起，炸弹炸到敌人和朋友……诸如此类。这些交互必须很好地进行下去，可不要把代码库给搅成一团毛线球。
- 最后，性能在游戏中至关重要。游戏开发者永远在榨取平台性能这件事上赛跑。多削掉一个 CPU 周期，你的游戏就有可能从掉帧和差评迈入 A 级游戏和百万销量的天堂。

如何阅读本书

本书大致分为三大部分。第一篇是介绍和框架。这包括前言和第 1 章。

第二篇，再探设计模式，回顾了 GoF 中的一些设计模式。在这个部分的每一章中，我都会试图给出自己对该模式的认识，以及对模式与游戏开发之间关联的看法。

最后部分是这本书的重头戏。这部分呈现了我认为十分有用的 13 种设计模式。它们分为 4 篇：序列型模式、行为型模式、解耦型模式和优化型模式。

这些模式使用一致的文本组织结构来讲述，以便你将该书作为参考并能快速找到你所需的内容。

- 目的部分简单介绍了该模式以及其力图解决的问题。以此作为开篇，以便你能够快速翻阅本书并根据自己眼下的问题对号入座。
- 动机部分描述了一个可引用该模式的示例问题。不同于具体的算法，模式只有运用到具体问题中时方能见其真章。教模式而不举具体例子，

就像教烤面包而不提面团一样。这个部分提供“面团”，之后的部分将会教你如何“烘培”。

- **模式**部分会提炼出前面示例中的模式本质。如果你想了解该模式枯燥的书面描述，就是这部分了。如果你已经熟悉了该模式，这部分也是一个很好的复习，确保你没有忘记该模式的要素。

- 到目前为止，该模式只是就一个单一的例子来解释的。但你怎么知道该模式是否适用于其他问题呢？**使用情境**对模式何时使用以及何时不该使用提供了一些指导。**使用须知**部分会指出使用该模式时带来的后果和风险。

- 如果你也像我一样，需要借助具体的实例才能真正的理解，那么**示例**部分正满足你的需要。它一步一步地展示这个模式的完整实现，以便你可以看到模式究竟是如何工作的。

- 模式和单一的算法不同，因为模式是开放式的。每次使用模式的时候，你实现的方式有可能会有不同。接下来**设计决策**部分，会探讨这个问题，并告诉你在应用模式时可供考虑的不同选项。

- 每章以一个短小的参考部分作为结束，它会告诉你该模式和其他模式的关联并指出使用该模式的一些真实的开源代码。

关于示例代码

这本书中的示例代码用 C++ 编写，但是这并不意味着这些模式仅能在 C++ 下发挥作用或者说 C++ 比其他语言要好。几乎所有的语言都适用，虽然有些模式确实倾向于有对象和类的语言。

我选择 C++ 有几个原因。首先，它是现行商业游戏中最流行的语言，是该行业的通用语言。另外，作为 C++ 基石的 C 语言的语法也是 Java、C#、JavaScript 和许多其他语言的基础。即使你不懂 C++，也没有关系，这里的示例代码基本上是你无需花太多力气就足以能够理解的。

这本书的目的不是教你学习 C++。示例会尽可能保持简单，但它可能并不符合优良的 C++ 编码风格或用法。阅读代码时要理解代码所传达的思想，而不是代码本身的表达。

特别一提的是，示例代码没有采用“现代”C++（C++11）或更高版本风格。它没使用标准库并很少使用模板。这是“糟糕”的 C++ 代码，但我仍希望保留这一特色，这样会对那些从 C、Objective-C、Java 和其他语言转来的读者更加的友好。

为了避免浪费篇幅，你已经看过的或者和模式不相关的代码，有时会

在例子中省略，通常用省略号来表示省去的代码。

例如有一个函数，它完成某项工作并返回一个值。同时讲解的模式只关心返回值，不关心其具体的工作内容。在这种情况下，示例代码看起来会像这样：

```
bool update()  
{  
    // Do work...  
    return isDone();  
}
```

何去何从

设计模式是软件开发中一个不断变化和扩展的部分。这本书延续了 GoF 的文献所开启的过程，并分享他们眼中的那些软件设计模式，而这一进程也不会因本书的完成而就此终止。

你是这个过程的核心之一。只要你开发了你自己的模式或提炼（或者反驳！）这本书中提到的模式，你就是在为软件社区贡献力量。如果你对书中的内容有任何建议、修正或者其他反馈，请与我联系。

致谢

我估计只有写过书的人才知道写书的过程中会遇到多少麻烦，但是还有另外一些人也知道写书的负担究竟有多重——那就是那些不幸和作者关系亲密的人。我是在妻子 Megan 煞费苦心地为节省的空余时间里写完这本书的。洗盘子和为孩子洗澡或许不能叫做“写作”，但是没有她的这些付出，这本书也不会出版。

当我还是 EA (Electronic Arts) 的一名程序员时，便开始写这本书了。我认为公司的同事们并不完全了解这本书的技术细节，但是我对 Michael Malone、Olivier Nallet 和 Richard Wifall 的支持表示感谢，感谢他们为前几章提供了详细、深刻的反馈。

写到大约一半的时候，我决定不做一名传统的出版者。我知道这意味着会失去编辑的指导，但是我收到了许多读者发送的电子邮件，他们告诉我希望这本书怎么写。我没有校对者，但是我收到了超过 250 份的 bug 报告，来帮助我改进写作。我也曾缺乏按计划写作的动力，但当我完成每一章并收到来自读者的鼓励的时候，我又有了充足的精神动力。

他们称这为“自出版”，但是“众包出版”更加贴切。写作是一份孤独的工作，但是我从未孤单过。即使整个写作过程持续了两年时间，但我总能不断得到鼓励。如果没有一堆人不断提醒我他们在期待着更多的章节，我绝不会想要继续写作并完成这本书。

对每一位发邮件的或者评论过的，点赞的或者收藏的，发微博的或者转发了的，任何帮助过我的，或者将本书告诉朋友的，或者给我提交一份 bug 报告的朋友们：我内心充满了对你们的感激。完成这本书是我人生中最大的目标之一，是你们帮我实现了它。

感谢你们！

我并不是没有文字编辑。Lauren Briesse 在我需要的时候帮助了我，并出色地完成了工作。

特别感谢 Colm Sloan，他仔细地把每个章节阅读了两遍，并给了我大量出色的反馈。这都出自他内心的善意。我欠他一份人情。

多数游戏程序员所面临的最大挑战就是完成他们的游戏。许多游戏止步于其高度复杂的代码库面前，而最终没能问世。游戏编程设计模式正是为解决此问题而生。带着多年上市 3A 级大作的经验，本书收集了许多已经实证的设计模式来帮助解构、重构以及优化你的游戏，书中将各大模式以菜单的形式分立以便开发者们各取所需。

你将学会如何编写一个健壮的游戏循环，如何应用组件来组织实体，并利用 CPU 缓存来提升游戏性能。本书将带你深入了解脚本引擎如何对行为进行编码，以及四叉树和其他空间划分等优化引擎的手段，并为你展示其他经典的设计模式是如何应用于游戏之中的。

目录

第 1 篇 概述

第 1 章 架构、性能和游戏	3
1.1 什么是软件架构	3
1.1.1 什么是好的软件架构	3
1.1.2 你如何做出改变	4
1.1.3 我们如何从解耦中受益	5
1.2 有什么代价	5
1.3 性能和速度	6
1.4 坏代码中的好代码	7
1.5 寻求平衡	8
1.6 简单性	9
1.7 准备出发	9

第 2 篇 再探设计模式

第 2 章 命令模式	13
2.1 配置输入	14
2.2 关于角色的说明	16
2.3 撤销和重做	18
2.4 类风格化还是函数风格化	21
2.5 参考	22
第 3 章 享元模式	23
3.1 森林之树	23
3.2 一千个实例	25
3.3 享元模式	26
3.4 扎根之地	26

3.5	性能表现如何	30
3.6	参考	31
第 4 章	观察者模式	33
4.1	解锁成就	33
4.2	这一切是怎么工作的	34
4.2.1	观察者	35
4.2.2	被观察者	35
4.2.3	可被观察的物理模块	37
4.3	它太慢了	38
4.4	太多的动态内存分配	39
4.4.1	链式观察者	39
4.4.2	链表节点池	42
4.5	余下的问题	43
4.5.1	销毁被观察者和观察者	43
4.5.2	不用担心, 我们有 GC	44
4.5.3	接下来呢	44
4.6	观察者模式的现状	45
4.7	观察者模式的未来	46
第 5 章	原型模式	47
5.1	原型设计模式	47
5.1.1	原型模式效果如何	50
5.1.2	生成器函数	51
5.1.3	模板	51
5.1.4	头等公民类型 (First-class types)	52
5.2	原型语言范式	52
5.2.1	Self 语言	53
5.2.2	结果如何	54
5.2.3	JavaScript 如何	55
5.3	原型数据建模	57
第 6 章	单例模式	61
6.1	单例模式	61
6.1.1	确保一个类只有一个实例	61
6.1.2	提供一个全局指针以访问唯一实例	62
6.2	使用情境	63
6.3	后悔使用单例的原因	65

6.3.1	它是一个全局变量	65
6.3.2	它是个画蛇添足的解决方案	66
6.3.3	延迟初始化剥离了你的控制	67
6.4	那么我们该怎么做	68
6.4.1	看你究竟是否需要类	68
6.4.2	将类限制为单一实例	70
6.4.3	为实例提供便捷的访问方式	71
6.5	剩下的问题	73
第 7 章	状态模式	75
7.1	我们曾经相遇过	75
7.2	救星：有限状态机	78
7.3	枚举和分支	79
7.4	状态模式	82
7.4.1	一个状态接口	82
7.4.2	为每一个状态定义一个类	83
7.4.3	状态委托	84
7.5	状态对象应该放在哪里呢	84
7.5.1	静态状态	84
7.5.2	实例化状态	85
7.6	进入状态和退出状态的行为	86
7.7	有什么收获吗	88
7.8	并发状态机	88
7.9	层次状态机	89
7.10	下推自动机	91
7.11	现在知道它们有多有用了吧	92

第 3 篇 序列型模式

第 8 章	双缓冲	95
8.1	动机	95
8.1.1	计算机图形系统是如何工作的（概述）	95
8.1.2	第一幕，第一场	96
8.1.3	回到图形上	97
8.2	模式	98
8.3	使用情境	98
8.4	注意事项	98

8.4.1	交换本身需要时间	98
8.4.2	我们必须有两份缓冲区	99
8.5	示例代码	99
8.5.1	并非只针对图形	102
8.5.2	人工非智能	102
8.5.3	缓存这些巴掌	106
8.6	设计决策	107
8.6.1	缓冲区如何交换	107
8.6.2	缓冲区的粒度如何	109
8.7	参考	110
第 9 章	游戏循环	111
9.1	动机	111
9.1.1	CPU 探秘	111
9.1.2	事件循环	112
9.1.3	时间之外的世界	113
9.1.4	秒的长短	113
9.2	模式	114
9.3	使用情境	114
9.4	使用须知	114
9.5	示例代码	115
9.5.1	跑, 能跑多快就跑多快	115
9.5.2	小睡一会儿	115
9.5.3	小改动, 大进步	116
9.5.4	把时间追回来	118
9.5.5	留在两帧之间	119
9.6	设计决策	120
9.6.1	谁来控制游戏循环, 你还是平台	121
9.6.2	你如何解决能量耗损	121
9.6.3	如何控制游戏速度	122
9.7	参考	123
第 10 章	更新方法	125
10.1	动机	125
10.2	模式	127
10.3	使用情境	128
10.4	使用须知	128

10.4.1	将代码划分至单帧之中使其变得更加复杂	128
10.4.2	你需要在每帧结束前存储游戏状态以便下一帧继续	128
10.4.3	所有对象都在每帧进行模拟, 但并非真正同步	129
10.4.4	在更新期间修改对象列表时必须谨慎	129
10.5	示例代码	130
10.5.1	子类化实体	132
10.5.2	定义实体	132
10.5.3	逝去的时间	135
10.6	设计决策	136
10.6.1	update 方法依存于何类中	136
10.6.2	那些未被利用的对象该如何处理	137
10.7	参考	137

第 4 篇 行为型模式

第 11 章	字节码	141
11.1	动机	141
11.1.1	魔法大战	141
11.1.2	先数据后编码	142
11.1.3	解释器模式	142
11.1.4	虚拟机器码	145
11.2	字节码模式	145
11.3	使用情境	145
11.4	使用须知	146
11.4.1	你需要个前端界面	146
11.4.2	你会想念调试器的	147
11.5	示例	147
11.5.1	法术 API	147
11.5.2	法术指令集	148
11.5.3	栈机	149
11.5.4	组合就能得到行为	153
11.5.5	一个虚拟机	155
11.5.6	语法转换工具	155
11.6	设计决策	157
11.6.1	指令如何访问堆栈	157
11.6.2	应该有哪些指令	158

11.6.3	值应当如何表示	158
11.6.4	如何生成字节码	161
11.7	参考	162
第 12 章	子类沙盒	163
12.1	动机	163
12.2	沙盒模式	165
12.3	使用情境	165
12.4	使用须知	165
12.5	示例	166
12.6	设计决策	168
12.6.1	需要提供什么操作	168
12.6.2	是直接提供函数, 还是由包含它们的对象提供	169
12.6.3	基类如何获取其所需的状态	170
12.7	参考	173
第 13 章	类型对象	175
13.1	动机	175
13.1.1	经典的面向对象方案	175
13.1.2	一种类型一个类	177
13.2	类型对象模式	178
13.3	使用情境	179
13.4	使用须知	179
13.4.1	类型对象必须手动跟踪	179
13.4.2	为每个类型定义行为更困难	179
13.5	示例	180
13.5.1	构造函数: 让类型对象更加像类型	181
13.5.2	通过继承共享数据	183
13.6	设计决策	185
13.6.1	类型对象应该封装还是暴露	186
13.6.2	持有类型对象如何创建	187
13.6.3	类型能否改变	187
13.6.4	支持何种类型的派生	188
13.7	参考	189

第 5 篇 解耦型模式

第 14 章	组件模式	193
--------	------	-----

14.1	动机	193
14.1.1	难题	194
14.1.2	解决难题	194
14.1.3	宽松的末端	194
14.1.4	捆绑在一起	195
14.2	模式	196
14.3	使用情境	196
14.4	注意事项	196
14.5	示例代码	197
14.5.1	一个庞大的类	197
14.5.2	分割域	198
14.5.3	分割其余部分	200
14.5.4	重构 Bjorn	202
14.5.5	删掉 Bjorn	204
14.6	设计决策	206
14.6.1	对象如何获得组件	206
14.6.2	组件之间如何传递信息	207
14.7	参考	210
第 15 章	事件队列	211
15.1	动机	211
15.1.1	用户图形界面的事件循环	211
15.1.2	中心事件总线	212
15.1.3	说些什么好呢	213
15.2	事件队列模式	215
15.3	使用情境	215
15.4	使用须知	215
15.4.1	中心事件队列是个全局变量	216
15.4.2	游戏世界的状态任你掌控	216
15.4.3	你会在反馈系统循环中绕圈子	216
15.5	示例代码	217
15.5.1	环状缓冲区	219
15.5.2	汇总请求	222
15.5.3	跨越线程	223
15.6	设计决策	224
15.6.1	入队的是什麼	224

15.6.2	谁能从队列中读取.....	224
15.6.3	谁可以写入队列.....	225
15.6.4	队列中对象的生命周期是什么.....	226
15.7	参考.....	227
第 16 章	服务定位器.....	229
16.1	动机.....	229
16.2	服务定位器模式.....	230
16.3	使用情境.....	230
16.4	使用须知.....	231
16.4.1	服务必须被定位.....	231
16.4.2	服务不知道被谁定位.....	231
16.5	示例代码.....	231
16.5.1	服务.....	231
16.5.2	服务提供者.....	232
16.5.3	简单的定位器.....	232
16.5.4	空服务.....	233
16.5.5	日志装饰器.....	235
16.6	设计决策.....	236
16.6.1	服务是如何被定位的.....	236
16.6.2	当服务不能被定位时发生了什么.....	239
16.6.3	服务的作用域多大.....	240
16.7	其他参考.....	241

第 6 篇 优化型模式

第 17 章	数据局部性.....	245
17.1	动机.....	245
17.1.1	数据仓库.....	246
17.1.2	CPU 的托盘.....	247
17.1.3	等下，数据即性能.....	248
17.2	数据局部性模式.....	249
17.3	使用情境.....	249
17.4	使用须知.....	250
17.5	示例代码.....	250
17.5.1	连续的数组.....	251
17.5.2	包装数据.....	255

17.5.3 热/冷分解	258
17.6 设计决策	260
17.6.1 你如何处理多态	260
17.6.2 游戏实体是如何定义的	261
17.7 参考	263
第 18 章 脏标记模式	265
18.1 动机	265
18.1.1 局部变换和世界变换	266
18.1.2 缓存世界变换	267
18.1.3 延时重算	268
18.2 脏标记模式	269
18.3 使用情境	269
18.4 使用须知	270
18.4.1 延时太长会有代价	270
18.4.2 必须保证每次状态改动时都设置脏标记	271
18.4.3 必须在内存中保存上次的衍生数据	271
18.5 示例代码	271
18.5.1 未优化的遍历	272
18.5.2 让我们“脏”起来	273
18.6 设计抉择	275
18.6.1 何时清除脏标记	275
18.6.2 脏标记追踪的粒度多大	276
18.7 参考	276
第 19 章 对象池	277
19.1 动机	277
19.1.1 碎片化的害处	277
19.1.2 二者兼顾	278
19.2 对象池模式	278
19.3 使用情境	279
19.4 使用须知	279
19.4.1 对象池可能在闲置的对象上浪费内存	279
19.4.2 任意时刻处于存活状态的对象数目恒定	279
19.4.3 每个对象的内存大小是固定的	280
19.4.4 重用对象不会被自动清理	281
19.4.5 未使用的对象将占用内存	281

19.5	示例代码	281
19.6	设计决策	287
19.6.1	对象是否被加入对象池	287
19.6.2	谁来初始化那些被重用的对象	288
19.7	参考	290
第 20 章	空间分区	291
20.1	动机	291
20.1.1	战场上的部队	291
20.1.2	绘制战线	292
20.2	空间分区模式	293
20.3	使用情境	293
20.4	使用须知	293
20.5	示例代码	293
20.5.1	一张方格纸	294
20.5.2	相连单位的网格	294
20.5.3	进入战场	296
20.5.4	刀光剑影的战斗	297
20.5.5	冲锋陷阵	298
20.5.6	近在咫尺, 短兵相接	299
20.6	设计决策	302
20.6.1	分区是层级的还是扁平的	302
20.6.2	分区依赖于对象集合吗	303
20.6.3	对象只存储在分区中吗	305
20.7	参考	305

第 1 篇 概述

第 1 章 架构、性能和游戏

1

在我们一头扎进一堆模式之前，我想为你介绍一些关于我如何看待软件架构以及它是如何应用到游戏的一些背景，这可能会帮助你更好地理解这本书的其余部分。至少，当你陷入关于设计模式和软件架构是多么糟糕（或者很棒）的一场争论中时，它会给你一些论据来使用。

请注意，我没有假设你站在争论中的哪一方。就像任何军火商一样，我为所有战斗方提供武器。

1.1 什么是软件架构

如果你从头到尾阅读了这本书，那么你并不会了解到 3D 图形背后的线性代数或者游戏物理背后的演算。这本书也不会告诉你如何一步步改进你的 AI 搜索树或者模拟音频播放中的房间混响。

哇，此段简直为这本书打了一个糟糕的广告。

相反，这本书是关于上面这一切要使用的代码的组织方式。这里少谈代码，多谈代码组织。每个程序都具有一定的组织性，即使它只是“把所有东西扔到 `main()` 函数里然后看看会发生什么”，所以我认为讨论如何形成好的组织性会更有趣些。我们如何分辨一个架构的好坏呢？

我大概有 5 年时间一直在思索这个问题。当然，像你一样，我对好的设计有着一种直觉。我们都遇见过非常糟糕的代码库，最希望做的就是剔除它们，结束自己的痛苦。

不得不承认，我们大多数人只接触到一部分这样的工作。

少数幸运儿有相反的经验，他们有机会与设计精美的代码共事。那种代码库，感觉就像在一个完美的豪华酒店里站了很多礼宾在翘首等待你的光临。两者之间有什么区别呢？

1.1.1 什么是好的软件架构

对于我来说，好的设计意味着当我做出一个改动时，就好像整个程序都在期待它一样。我可以调用少量可选的函数来完美地解决一个问题，而不会为软件带来副作用。

这听起来不错，但还不够切实。“只管写你的代码，架构会为你收拾

一切。”没错!。

让我解释下。第一个关键部分是，架构意味着变化。人们不得不修改代码库。如果没人接触代码（不管是因为代码非常完美，又或者糟糕到人人都懒得打开文本编辑器来编辑它），那么它的设计就是无法体现其意义的。衡量一个设计好坏的方法就是看它应对变化的灵活性。如果没有变化，那么这就像一个跑步者从来没有离开过起跑线一样。

1.1.2 你如何做出改变

在你打开编辑器添加新功能，修复 bug 或者由于其他原因要修改代码之前，你必须明白现有的代码在做什么。当然，你不必知道整个程序，但是你需要将所有相关的代码加载到你的大脑中。

我们倾向于略过这一步，但它往往是编程中最耗时的部分。如果你认为从磁盘加载一些数据到 RAM 很慢的话，试着通过视觉神经将这些数据加载到你的大脑里。

一旦你的大脑有了一个全面正确的认识，则只需稍微思考一下就能提出解决方案。这观点值得反复斟酌，但通常这是比较明确的。一旦你理解了这个问题和它涉及的代码，则实际的编码有时是微不足道的。

你的手指游走于键盘间，直到右侧的彩色灯光在屏幕上闪烁时，你就大功告成了，是吗？还没有！在你编写测试，并将它发送给代码审查之前，你通常有一些清理工作要做。

你在游戏中加入了一些代码，但是你不希望后面处理代码的人花大量时间理解或修改你的代码。除非变动很小，通常都会做些重新组织工作来让你新加的代码无缝集成到程序中。如果你做得很好，那么下一个人在添加代码的时候就不会察觉到你的代码变动。

简而言之，编程的流程图如图 1-1 所示。

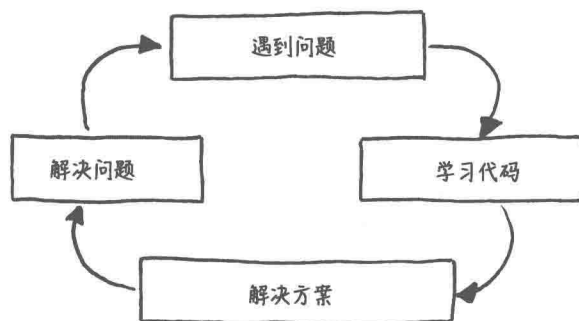


图 1-1 编程的流程图

这在字面上是一个 OCR1 过程，不过这个想法有些奇怪。

我说“测试”了吗？哦，是的，我说了。为一些游戏代码编写单元测试比较难，但是大部分代码是可以完全测试的。我这里不是要慷慨陈词，不过，如果你之前没有考虑过多做自动化测试的话，我希望你多做一些。难道没有比一遍一遍手动验证东西更好的事情要做吗？

现在想想，流程图的环路中没有出口有点小惊悚。

1.1.3 我们如何从解耦中受益

虽然不是很明显，但我认为很多软件架构师还处于学习阶段。将代码加载到脑中如此痛苦缓慢，得自己寻找策略来减少装载代码的体积。这本书有一整章（第 5 章）是关于解耦模式的，许多的设计模式也有相同的思想。

你可以用一堆方式来定义“解耦”，但我认为如果两块代码耦合，意味着你必须同时了解这两块代码。如果你让它们解耦，那么你只需了解其一。这很棒，因为如果只有一块代码和你的问题相关，则你只需要将这块代码装载到你的脑袋中，而不用把另外一块也装载进去。

对我来说，这是软件架构的一个关键目标：在你前进前，最小化你脑海中的知识储存量。

当然，对解耦的另一个定义就是当改变了一块代码时不必更改另外一块代码。很明显，我们需要更改一些东西，但是耦合得越低，更改所波及的范围就会越小。

1.2 有什么代价

这听起来很不错，不是吗？对一切进行解耦，你就可以迅速编写代码。每一次变化意味着只会涉及某一个或两个方法，然后你就可以在代码库上行云流水地编写代码。

这种感觉正是为什么人们会为抽象、模块化、设计模式和软件架构感到兴奋的原因。一个架构良好的程序工作起来真的会令人愉悦，每个人都会更加高效。良好的架构在生产力上会产生巨大的差异。怎么夸大它带来的效果是如何深远都不为过。

但是，天下没有免费的午餐。良好的架构需要很大的努力及一系列准则。每当你做出一个改变或者实现一个功能时，你必须很优雅地将它们融入到程序的其余部分。你必须非常谨慎地组织代码并保证其在开发周期中经过数以千计的小变化之后仍然具有良好的组织性。

你必须要考虑程序的哪一部分应该要解耦然后在这些地方引入抽象。同样地，你要确定在哪里做一些扩展以便将来很容易应对变化。

人们对此非常兴奋。他们设想着，未来的开发者（或者是他们自己）进入代码库，发现代码库开放、强大，只等着被加些扩展。他们想象一个游戏引擎便可统治一切。

这小节的下半部分（维护你的设计）需要特别注意。我曾见过许多程序在开始时写得很漂亮，但死于一个又一个“一个小补丁而已”。

就像园艺一样，只种植是不够的。你必须去除草、修剪。

有人杜撰了“YAGNI”一词（You aren't gonna need it 你不需要它）作为口头禅，用它来与猜测未来的自己会想要什么这种冲动进行斗争。

但是，事情就在这里开始变得棘手。当你添加了一个抽象层或者支持可扩展的地方，你猜想到你以后会需要这种灵活性，于是你便为你的游戏增加了代码和复杂性，这需要时间来开发、调试和维护。

如果你猜对了，那么你之前的辛苦就没白费，而且也无须再对代码进行任何修改。但是猜测未来是很难的，并且当模块最终没起到作用时，很快它就变得有害。毕竟，你必须处理这些多出来的代码。

当你过度关注这点时，便会得到一个架构已经失控的代码库。你会看到接口和抽象无处不在。插件系统、抽象基类、虚方法众多，还有各种的扩展点。

你将花费大量时间去找到有实际功能的代码。当你需要做出改变时，当然有可能有接口能帮上忙，但你会很难找到它。从理论上讲，解耦意味着在你进行扩展时仅需理解少量代码，然而抽象却增加了理解代码的难度。

像这样的代码库正是让人们反对软件架构尤其是设计模式的原因。对代码进行包装很容易，以至于让你忽视了你推出了一款游戏的事实。一味地追求可扩展性让无数开发者在一个“引擎”上花费数年却没有搞清楚引擎究竟是用来做什么的。

1.3 性能和速度

一个有趣的范例是 C++ 模板。模板元编程有时可以让你获得抽象接口而没有任何运行时开销。

对灵活的定义，不同人有不同的看法，当你在某些类中调用一个具体方法时，你相当于将这个类固定（很难做出改变）。当你使用一个虚方法或者接口时，被调用的类将直到真正运行起来才能被追踪到，这样的程序更具灵活性但是会增加额外的运行成本。

模板元编程介于两者之间。在模板元编程中，在编译期间你就能决定在模板实例化时调用哪个类。

你有时候会听到关于软件架构和相关概念的批评声，尤其在游戏开发中：它会影响到游戏的性能。许多模式让你的代码更加灵活，但是它依赖于虚函数派发、接口、指针、消息以及其他至少有一些运行成本的机制。

还有一个原因。很多软件架构的目标是使你的程序更加灵活，这样只需较少的代价便可对代码进行改变，这也意味着在程序中更少的编码。你使用接口，以便代码可以与任何实现这些接口的类进行工作，而不是使用具体类。你使用观察者模式（第 4 章）和通信模式（第 15 章）使得游戏的两部分互相沟通，而将来它们自身就会成为另外两个需要沟通的部分。

但是性能优化总是在某些假设下进行的。优化的方法在特定的条件下进行更好。我们能肯定地假设永远不会有超过 256 个敌人吗？好极了，我们可以将 ID 打包成一个单字节。在这里我们只会在一个具体类型上调方法吗？好，我们就静态调度或者对它内联。所有的实体都是同一个类吗？太好了，我们可以将它们做成一个很棒的连续排列（第 17 章）。

这并不意味着它的灵活性很差！它可以让我们快速地进行游戏更新，

开发速度是让游戏变得有趣的关键性因素。没有人，哪怕是 Will Wright¹，可以在纸上设计出一个平衡的游戏。这需要迭代和实验。

你越快地对想法付诸实践并观察效果，你就能越多地尝试并越有可能找到一些很棒的东西。即便在你已经找到合适的技术之后，你也要用充足的时间来进行调整。一个细小的不平衡就会破坏掉游戏的乐趣。

这里没有简单的答案。将你的程序做得更具有灵活性，以便能够更快速地进行原型编写，但这会带来一些性能损失。同样地，对你的代码进行优化会降低它的灵活性。

根据我的经验，将一款有趣的游戏做得高效要比将一款高性能的游戏做的有趣更简单些。一种折中的办法是保持代码的灵活性，直到设计稳定下来，然后去除一些抽象，以提高游戏的性能。

1.4 坏代码中的好代码

这使我想到的下一个点是，编码风格讲求天时地利。本书的很多部分是关于编写可维护的、干净的代码，所以我的意图很明确，就是用“正确”的方式做事情，但是也存在一些草率的代码。

编写架构良好的代码需要仔细的思考，这是需要时间的。更多的是，在项目的生命周期内维护一个良好的架构需要很大的努力。你必须把你的代码库看作一个好的露营者在寻找营地一样：总是试着寻找比眼下更好的扎营点。

当你准备要长期和那份代码打交道时，这样是好的。但是，就像我前提到的，游戏设计需要大量的试验和探索，特别是在早期，编写一些你知道迟早要扔掉的代码是很稀松平常的。

如果你只是想验证一些游戏想法是否能够正确工作，那么对其精心设计架构就意味着在想法真正显示到屏幕并得到反馈之前需要花费更多时间。如果它最终没有工作，那么当你删除代码时，花费在编写优雅代码上的时间其实都浪费掉了。

原型（把那些仅仅在功能上满足一个设计问题的代码融合在一起）是一个完全正确的编程实践。然而，特别提醒下，如果你编写一次性的代码，那么你必须确保能将之扔掉。我不止一次看到一些糟糕的经理重演以下场景。

老板：“嘿，我们已经有想法了，准备尝试下。只是一个原型，所以

¹ 译者注：威尔·赖特，著名游戏制作工程师。

有一个小技巧确保你的原型代码不会变成真正的代码，就是使用不同于你游戏使用的语言来编写。这样的话，你就必须用游戏使用的语言重写一遍了。

不必感觉必须要做得正确。大概多久能实现？”

开发：“嗯，如果我简化很多，不测试，不写文档，不管 bug，我几天内就可以给你一些临时的代码。”

老板：“太好了！”

几天后……

老板：“嘿，原型写得很不错。你能花几个小时清理下代码然后开始真枪实弹的干么？”

你需要确保这些使用一次性代码的人们明白这种一次性代码看起来能够运行，但是它却不可维护，必须被重写。如果可能，最终你也许会保留它们，但需要后续修改得特别好。

1.5 寻求平衡

开发中我们有几个因素需要考虑。

1. 我们想获得一个良好的架构，这样在项目的生命周期中便会更容易理解代码。
2. 我们希望获得快速的运行时性能。
3. 我们希望快速完成今天的功能。

这些目标至少部分是相冲突的。好的架构从长远来看，改进了生产力，但维护一个良好的架构就意味着每一个变化都需要更多的努力来保持代码的干净。

最快编写的代码实现却很少是运行最快的。相反，优化需要消耗工程时间。一旦完成，也会使代码库僵化：高度优化过的代码缺乏灵活性，很难改变。

完成今日的工作并担心明天的一切总伴随着压力。但是，如果我们尽可能快的完成功能，我们的代码库就会充满了补丁、bug 和不一致的混乱，会一点点地消磨掉我们未来的生产力。

这里没有简单的答案，只有权衡。从我收到的电子邮件中，看得出来，这让很多人头疼。特别是对于想做一个游戏的新手们来说，听到这样说挺吓人的，“没有正确答案，只是错误口味不同”。

但是，对于我而言，这令人兴奋！看看人们从事致力的领域，在这中心，你总能找到一组相互交织的约束。毕竟，如果有一个简单的答案，每个人都会这么做。在一周内便可掌握的领域最终是无聊的。你不会接触到在别人的杰出职业生涯中所挖掘出的东西。

对于我而言，这和游戏本身有很多共同点。就像国际象棋永远无法掌

我认为一个有趣的地方是这些都是关于某种速度：我们的长期开发速度，游戏的执行速度，以及我们短期内的开发速度。

你绝对没听到过某人在挖掘水沟上的卓越事迹。也许你有，但我却没有研究过这个领域。据我所知，那里也许有热衷于水沟挖掘的爱好者，水沟挖掘准则，并且有一个自己的文化圈子。我们凭什么去评判呢？

握，因为它是如此完美的平衡。这意味着你可以穷尽一生来探索可行的战略空间。设计不当的游戏如果用一个稳赢的战术一遍遍玩，会让你厌倦并退出。

1.6 简单性

最近，我觉得如果有任何方法来缓解这些限制，那便是简单性了。在今天我所写的代码中，我非常努力地尝试着编写最干净、最直接的函数来解决问题。这种代码在你阅读之后，就会明白它究竟做了什么，并且不敢想象还有其他可能的解决方案。

我致力于保持数据结构和算法的正确性（在这个顺序下），然后继续往下做。我觉得如果我能保持简单性，代码量就会变少。这意味着更改代码时，我的脑袋里只需装载更少的代码。

它通常运行速度快，因为根本就没有那么多的开销，也没有太多的代码要执行（这当然并非总是如此，你可以在小部分代码中进行很多的循环和递归）。

但是，请注意，我并不是说简单的代码会花费较少的时间来编写。你会觉得最终的总代码量更少了，但是一个好的解决方案并不是更少的实际代码量，而是对代码的升华。

我们很少会遇到一个非常复杂的问题，用例反而有一大堆，例如，你想让 X 在 Z 的情况下执行 Y 而在 A 的情况下执行 W，以此类推。换句话说，是一个不同实例行为的长列表。

最省脑力的方法就是只编写一次测试用例。看一下新手程序员，这是他们经常做的：为每个需要记住的用例构建大量的条件逻辑。

在那里面毫无优雅性，当程序有输入或者编码者稍微考虑得跟用例有些不一样时，这种风格的代码就最终会沦陷。当我们考虑优雅的解决方案时，浮现脑海中的就有一个：一小块逻辑就能正确地处理一大片用例。

你会发现这有点像模式匹配或解谜。它需要努力识破测试用例的分散点，以找到它们背后隐藏的秩序。当你把它解决时，会感觉很棒。

1.7 准备出发

几乎每个人都会跳过介绍章节，所以在这里我祝贺你能够阅读到这里。我没有太多的东西来回报你的这份耐心，但是这里我能给你提供一些建议，希望对你有用。

Blaise Pascal 用了一句名言作为了一封信的结尾：“我会写一封更简短的信，但我没有足够的时间。”

另一种引用来自 Antoine de Saint-Exupery：“极臻完美，并非无以复加，而是简无可减。”

言归正传，我注意到，每次我修改这本书的章节时，它都会变得更短。一些章节在完成时要比原来缩短 20%。

相信我，在游戏发布前的两个月并不是你开始担心“游戏FPS只有1帧”问题的时候。

- 抽象和解耦能够使得你的程序开发变得更快和更简单。但不要浪费时间来做事，除非你确信存在问题的代码需要这种灵活性。
- 在你的开发周期中要对性能进行思考和设计，但是要推迟那些降低灵活性的、底层的、详尽的优化，能晚则晚。
- 尽快地探索你的游戏的设计空间，但是不要走得太快留下一个烂摊子给自己。毕竟你将不得不面对它。
- 如果你将要删除代码，那么不要浪费时间将它整理得很整洁。摇滚明星把酒店房间弄得很乱是因为他们知道第二天就要结账走人。
- 但是，最重要的是，若要做一些有趣的玩意，那就乐在其中地做吧。

第 2 篇

再探设计模式

《设计模式：可复用面向对象软件的基础》(Design Patterns: Elements of Reusable Object-Oriented Software)一书已经出版了将近 20 年。如果你并不认为自己比我更为高瞻远瞩，那么现在正是阅读《设计模式：可复用面向对象软件的基础》这本经典的好时机。对于软件这个发展迅速的行业来说，这本书确实有些古老了。但是，这本书的经久不衰说明比起许多框架和方法论而言，设计模式更加永恒。

尽管我认为《设计模式：可复用面向对象软件的基础》一书到今天仍然适用，但是我们从过去几十年中学习到了许多新的知识。在本章节中，我们将回顾一遍 GoF 记载的几个最初的设计模式。对每一种模式，我希望都能说出一些实用或者有趣的东西来。

我认为有些模式被滥用了（单例模式），而另一些又被冷落了（命令模式）。同时我想要阐述另一对设计模式（享元模式和观察者模式）在游戏开发中的联系。最后，我认为发掘那些在更为广泛的编程领域背后所潜藏的设计模式（原型模式和状态模式）是件很有趣的事。

本篇模式

- 命令模式
- 享元模式
- 观察者模式
- 原型模式
- 单例模式
- 状态模式

第 2 章 命令模式

“将一个请求 (request) 封装成一个对象，从而允许你使用不同的请求、队列或日志将客户端参数化，同时支持请求操作的撤销与恢复。”

命令模式是我最喜爱的模式之一。在我开发的绝大多数大型游戏或其他程序中，最终都用到了它。正确地使用它，你的代码会变得更加优雅。关于这个重要的模式，GoF 做了上述具有预见性的深奥描述。

我想你也和我一样觉得这句话晦涩难懂。首先，它的比喻不够形象。在软件界之外，一词往往多义。“客户 (client)” 指代同你有着某种业务往来的一类人。据我查证，人类 (human beings) 是不可“参数化”的。

其次，句子的剩余部分只是列举了这个模式可能的使用场景。而万一你遇到的用例不在其中，那么上面的阐述就不太明朗了。我对命令模式的精练 (pithy) 概括如下：

命令就是一个对象化 (实例化) 的方法调用 (A command is a reified method call)。

当然，“精练”通常意味着“简洁到令人费解”，所以这里我的定义可能显得不够好。让我解释一下：你可能没听过“Reify”一词，意即“具象化” (make real)。另一个术语 reifying 的意思是使一些事物成为“第一类” (first-class)。¹

这两个术语都意味着，将某个概念 (concept) 转化为一块数据 (data)、一个对象，或者你可以认为是传入函数的变量等。所以说命令模式是一个“对象化的方法调用”，我的意思就是封装在一个对象中的一个方法调用。

你可能对“回调 (callback)”、“头等函数 (first-class function)”、“函数指针 (function pointer)”、“闭包 (closure)”和“局部函数 (partially applied function)”更熟悉，至于熟悉哪个取决于你所使用的语言，而它们本质上具有共性。GoF 后面这样补充到：

“Reify” 出自拉丁文 “res”，意思是“thing”，加上英语后缀 “-fy”，所以就成为了 “thingify”，坦白说，我认为直接使用这个词会更有趣。

¹ 译者注：你可能在其他书籍中也见到过“第一类值”、“头等”、“一等”等类似说法。

一些语言的反射系统 (Reflection system)¹ 可以让你在运行时命令式地处理系统中的类型。你可以获取到一个对象，它代表着某些其他对象的类，你可以通过它试试看这个类型能做什么。换句话说，反射是一个对象化的类型系统。

命令就是面向对象化的回调 (Commands are an object-oriented replacement for callbacks)。

这个说法比他们上面那句概括要好得多。

但是这些听起来都比较抽象和模糊。正如我所推崇的那样，我喜欢用一些具体点的东西来作为开篇讲解。为弥补这点，现在开始我将举例说明命令模式的使用场景。

2.1 配置输入

每个游戏都有一处代码块用来读取用户原始输入：按钮点击、键盘事件、鼠标点击，或者其他输入等。它记录每次的输入，并将之转换为游戏中一个有意义的动作 (action)，如图 2-1 所示。

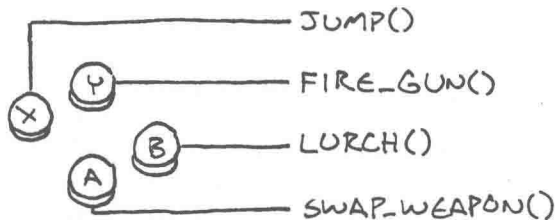


图 2-1 按钮与游戏行为的映射

下面是一个简单的实现：

```
void InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) jump();
    else if (isPressed(BUTTON_Y)) fireGun();
    else if (isPressed(BUTTON_A)) swapWeapon();
    else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

这个函数通常会在每一帧中通过游戏循环 (第 9 章) 被调用，我想你能理解这段代码的作用。如果我们将用户的输入硬编码到游戏的行为 (game actions) 中去，上面的代码是有效的，但是许多游戏允许用户配置他们的按钮与游戏行为之间的映射关系。

为了支持自定义配置，我们需要把那些对 jump() 和 fireGun() 方法的直接调用转换为我们可以更换 (swap out) 的东西。“可更换的 (swapping out)” 听起来会让人联想到分配变量，所以我们需要个对象来代表一个游

¹ 译者注：如.NET。

专业级提示，请勿常按 B 键。

戏动作。这就用到了命令模式。

我们定义了一个基类用来代表一个可触发的游戏命令：

```
class Command
{
public:
    virtual ~Command() {}
    virtual void execute() = 0;
};
```

当某个接口中仅剩一个返回值为空的方法时，命令模式便很可能适用。

然后，我们为每个不同的游戏动作创建一个子类：

```
class JumpCommand : public Command
{
public:
    virtual void execute() { jump(); }
};

class FireCommand : public Command
{
public:
    virtual void execute() { fireGun(); }
};
```

// You get the idea...

在我们的输入处理中，我们为每个按钮存储一个指向它的指针。

```
class InputHandler
{
public:
    void handleInput();

    // Methods to bind commands...

private:
    Command* buttonX_;
    Command* buttonY_;
    Command* buttonA_;
    Command* buttonB_;
};
```

现在输入处理便通过这些指针进行代理：

```
void InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) buttonX_->execute();
```


注意，我们这里没有检查命令是否为 NULL。因为这里假设了每个按钮都有某个命令对象与之对应关联。

如果你想要支持不处理任何事情的按钮，而不用明确检查按钮对象是否为 NULL，我们可以定义一个命令类，这个命令类中的 execute() 方法不做任何事情。然后，我们将按钮处理器（button handler）指向一个空值对象（null object），就好像它指向了 NULL 一样。这便是应用了空值对象模式。

```
else if (isPressed(BUTTON_Y)) buttonY->execute();
else if (isPressed(BUTTON_A)) buttonA->execute();
else if (isPressed(BUTTON_B)) buttonB->execute();
}
```

以前每个输入都会直接调用一个函数，现在则增加了一个间接调用层，如图 2-2 所示。

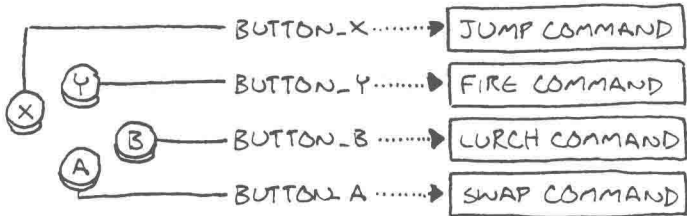


图 2-2 按钮与可分配命令的映射

简而言之，这就是命令模式。如果你已经看到了它的优点，不妨看完本章的剩余部分。

2.2 关于角色的说明

我们刚才定义的命令类在上个例子中是有效的，但它们却有局限性。问题在于它们做了这样的假定：存在 jump()、fireGun() 等这样的顶级函数，这些函数能够隐式地获知玩家游戏实体并对其进行木偶般的操控。

这种对耦合性的假设限制了这些命令的使用范围。JumpCommand 类的跳跃命令只能作用于玩家对象。让我们放宽限制，传进去一个我们想要控制的对象而不是让命令自身来确定所控制的对象：

```
class Command
{
public:
    virtual ~Command() {}
    virtual void execute(GameActor& actor) = 0;
};
```

这里，GameActor 是我们用来表示游戏世界中的角色的“游戏对象”类。我们将它传入 execute() 中，以便命令的子类可以针对我们选择的角色进行调用，如下所示：

```
class JumpCommand : public Command
{
public:
```

有关此电子书的说明

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融等等。质量都很清晰，为方便读者阅读观看，每本100%都带可跳转的书签索引和目录，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ1779903665。

PDF代找说明：

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF，大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系 QQ 1779903665.

备用:QQ 461573687

若以上联系方式失效，您可通过以下电子邮件获取有效联系方式。

E-mail：ebooksprite@163.com

E-mail：ebooksprite@foxmail.com

若您没有QQ通讯工具，请发送您的请求到 ebooksprite@gmail.com 与客服取得联系。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找和后期制作pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。