面向数据设计

Richard Fabian

zko (Translator)

2022 年 11 月 30 日

目录

第一章 面向数据设计 **7**
- 1.1 一切围绕数据 8
- 1.2 数据不是问题域 9
- 1.3 数据与统计 12
- 1.4 数据是可变的 13
- 1.5 数据的形态 15
- 1.6 框架 (framework) 17
- 1.7 结论和启示 19

第二章 关系型数据库 **21**
- 2.1 复杂状态 21
- 2.2 为复杂的数据寻找计算框架 22
- 2.3 标准化数据 23
- 2.4 标准化 28
  - 2.4.1 主键 (Primary key) 28
  - 2.4.2 第一范式 (1st Normal Form) 30
  - 2.4.3 第二范式 (2nd Normal Form) 33
  - 2.4.4 第三范式 (3rd Normal Form) 37
  - 2.4.5 Boyce-Codd 范式 (Boyce-Codd Normal Form) 37
  - 2.4.6 域键和领域知识 37
  - 2.4.7 反思 39
- 2.5 操作 40
- 2.6 总结 41
- 2.7 流处理 42

2.8 为什么数据库技术很重要? 43

第三章 存在性处理 45

3.1 复杂度 45
3.2 调试 46
3.3 为什么要用 if 47
3.4 处理的类型 50
3.5 避免使用 boolean 51
3.6 慎用枚举 55
3.7 初探多态 56
3.8 动态运行时多态 56
3.9 事件处理 59

第四章 基于组件的对象 61

4.1 野生组件 62
4.2 远离层级结构 64
4.3 面向管理器 66
4.4 没有什么实体 68

第五章 层级细节水平和隐式状态 71

5.1 从零到无限 71
5.2 快照 74
5.3 JIT 快照 75
5.4 替代维度 76
5.4.1 真正的指标 77
5.4.2 超越空间 77
5.4.3 总体 LoD (如何减少实例数量) 79

第六章 查找 81

6.1 索引 81
6.2 面向数据查询 82
6.3 寻找极值是排序问题 86
6.4 查找随机是哈希 (或树) 问题 87

第七章 排序 89
7.1 真的必要吗? 89
7.2 插入排序与并行归并排序 90
7.3 针对平台排序 91

第八章 优化和实现 95
8.1 什么时候应该做优化? 96
8.2 反馈 97
8.2.1 了解极限 98
8.3 优化策略 99
8.3.1 定义问题 99
8.3.2 测量 99
8.3.3 分析 99
8.3.4 实现 100
8.3.5 确认 100
8.3.6 总结 101
8.4 表格 101
8.5 变换 104
8.6 碰撞的空间集 105
8.7 针对大量数据惰性求值 106
8.8 必要性-只取所需 106
8.9 变化的长度集 107
8.10 间接连接 109
8.11 数据驱动的技术 109
8.11.1 SIMD 110
8.12 数组的结构 111

第九章 帮助编译器 113
9.1 减少顺序依赖 113
9.2 减少内存依赖 113
9.3 察觉缓冲区写入 114
9.4 别名 (Aliasing) 115
9.5 返回值优化 116
9.6 缓存行利用率 116

9.7 伪共享 (False Sharing) . . . 117
9.8 注意推测执行 . . . 118
9.9 分支预测 . . . 119
9.10 避免被驱逐 . . . 120
9.11 自动矢量化 . . . 120

第十章 维护与复用 125

10.1 宇宙层级 . . . 125
10.2 调试 . . . 126
10.2.1 生命周期 . . . 126
10.2.2 远离指针 . . . 126
10.2.3 不良状态 . . . 126
10.3 复用性 . . . 127
10.4 可复用的函数 . . . 129
10.5 单元测试 . . . 129
10.6 重构 . . . 130

第十一章 问题出在哪？ 131

11.1 危害 . . . 132
11.2 映射问题 . . . 135
11.3 内部化状态 . . . 138
11.4 面向实例开发 . . . 140
11.5 层设计与变革 . . . 141
11.6 分工 . . . 143
11.7 可复用的通用代码 . . . 144

第一章　面向数据设计

面向数据设计，已经以各种形式存在了几十年，但直到 2009 年 9 月，才在 Noel Llopis 的同名文章 [11] 中以这个名字首次出现。它算不算是种编程范式，一直以来争议不断。有些人认为，它可以与面向对象、过程式、函数式编程等其他范式混用。某种意义上来说，的确如此，面向数据设计确实能同其他范式共同发挥作用，但也不否认，它是种更广义的编程方式。Lisp 程序员会知道，函数式编程可以与面向对象共存，C 程序员很清楚面向对象能与过程式编程共存。这里我们将这些争议先搁置，直接在此断言：面向数据设计是新的重要工具；它能够与其他工具共存。[1]

2009 年是非常合适的时机。革命性的硬件业已成熟；潜力巨大的计算机被无视硬件的编程范式束缚；开发者的编码方式能让许多引擎程序员落泪。但时代变了。现在许多移动端和桌面端解决方案，似乎不太需要面向数据设计，并不是机器擅长处理低效实现，而是设计中的游戏要求不高，也不复杂。但现在手游的开发趋势似乎正向 AAA 级迈进，又一次，产生了对复杂情况管理的需求，以期最大程度发挥硬件。

现如今，我们被多核设备围绕着，口袋里这个也不例外。学习如何较少依赖串行开发软件就变得尤为重要。面向数据的程序员能获得诸多好处，包括但不限于：摆脱对象信息传递、获得即时响应等。编程时，坚实地依赖对数据流的认识，为将来进入 GPGPU 和其他计算方法做好准备。由此带来诸多落地游戏构想的工作。面向数据设计的需求只会增长。抽象和串行思维将制约你的竞争对手，而那些接受面向数据方法的人会茁壮成长。

[1] 不过仍有一些限制，但并不与任何范式相互排斥，可能除了像 Prolog 这种逻辑编程语言。完全陈述性的 *what, not how* 的方法，看起来先天放弃考虑数据本身，以及它如何与机器互动。

1.1 一切围绕数据

数据即一切。数据是为了创造用户体验而需要变换的；是打开文档时加载的；是屏幕上的图形；是手柄按钮的脉冲；是扬声器震动空气产生的波；是角色升级的路线；也是对手向玩家开枪的诱因。数据是炸药引信的时长；是撞到尖刺掉宝的数量；是游戏结束时绚丽场景中每个粒子即时的位置和速度：经由源码，编译至汇编指令，再由解码器变换为机器指令，操纵磁盘读取内容，再一步步最终呈现到眼前。

没有数据的应用并不存在。没有图像，Adobe Photoshop 无从谈起。没有画笔、图层、笔压，它什么也算不上。没有字符、字体、分页符，Microsoft Word 也没有意义。没有事件，FL Studio 毫无价值。没有源码，Visual Studio 也只是个花瓶。过去所有的程序，都是为了基于输入数据，输出数据。数据的形式有时极其复杂，有时简单到无需文档，但所有应用程序都接收、产出数据。如果它们不需要可识别的数据，就最多只算玩具，Demo。

指令也是数据。指令会占用内存，使用带宽，并且可以被变换，加载，保存，构建。对于开发者，自然不会认为指令是数据 [2]，但在旧的、保护性较差的硬件上，它们的区别很小。尽管大多数当代硬件，会保护为可执行文件预留的内存，避免其被损害、被修改，但这一相对较新的发明仍未成熟。改进的哈佛架构对内存中的数据和指令同等依赖 [3]。因此，指令仍是数据，它们也是要变换的对象。我们接受指令并将其转化为行动。指令的数量、大小、频率都很重要。我们控制、筛选、使用哪些指令来解决问题，即是优化。知道了数据是什么，便能决定如何处理数据。了解指令，便有了理论支撑，能决定哪些指令是必要的，哪些是冗余的，哪些可以用低成本方案替代。

现在，我们有了面向数据的开发方法的论证基础，但还遗漏了一个主要因素。所有这些数据和及其变换，从字符串，到图像，再到指令，都必须在某样东西上运行。这个东西有时相当抽象，如，未知硬件上运行的虚拟机；有时又很具体，比如自己的，已知 CPU 、GPU、内存容量、带宽。所有的情况下，数据又不仅仅是数据，而是存在于某个硬件上的数据，而且必须经由同一硬件变换。本质上讲，面向数据设计是变换结构良好的数据，设计软件的方法，其中**结构良好**的标准是由数据的目标硬件，对其执行变换的模式与类型共同决定。有时，数据并不是很明确，硬件可能也捉摸不透。但大多数情况下，良好的硬件评判能力几乎对每一个软件项目都有所帮助。

---

[2] 除非是 Lisp 程序员

[3] 译注：有一种改进中，提供了指令存储器和 CPU 间的通路，来自指令存储器的字被当作只读数据。

如果应用程序的最终结果是数据，且所有的输入都可以表示为数据，并且了解所有的数据变换都没有凭空发生，那就可以基于此原则建立一个软件开发的方法论：理解数据，了解机器对特定数量、频率、统计量的数据执行变换时都发生了什么。在此基础上，就可以草拟一个关于如何使方法论面向数据的声明。

1.2 数据不是问题域

原则一：数据不是问题域。

对于有些人，面向数据设计似乎处于大多数其他编程范式的对立面，因为它不太容易让问题域进到软件源码中。它不鼓励将对象概念映射到用户的语境 (Context)。因为数据刻意地，自始至终没有意义。重视抽象的范式会假装计算机和它的数据不存在，将字节、CPU 管道、其他硬件特征等概念抽象出去，取而代之的是：引入问题模型。他们经常把有观点的模型引入代码，或者把世界模型作为问题的语境。就是说，要么围绕预期解决方案的属性，要么围绕问题域的描述来构建代码。

赋予数据意义就能创造信息。意义并非数据固有。只有一个 4 时，几乎没什么意义，但如果说 4 英里，或 4 个鸡蛋，它就有了意义。假设有 3 个数字，作为一个三元组意义不大，但如果将它们命名为 x, y, z ，就可以赋予它们位置的意义。有一份游戏中的位置列表，在没有语境的情况下也没什么意义。面向对象设计可能会把位置作为对象的一部分，通过类的名称和相邻的数据（也已经命名），就可以推断出数据的含义。如果没有已命名的语境数据与之关联，位置可以被赋予其他意义。虽然某种程度上，把数字放在语境当中是好的，但同时也阻碍了把位置作为三个数字的集合来思考，然而这一点对程序员思考如何解决的真正问题时，至关重要。

举例来讲，当把数据放在对象的深层，到后来又忘了它的存在。想想诸多已发售或尚在制作中的游戏，本可以使用一个 2D 或 3D 网格 (grid) 系统处理数据布局。不知为何，开发人员将地图上的每一个引用都实例化了。这还不是个例，在已经发售的游戏中，这种以对象为中心的方法摧残硬件的案例并不少见：相较于由真正的网格驱动，有数百个对象直接放置在世界空间的网格坐标上。可能程序员看到一个网格，看到所需的元素数量，就会对是否要为它分配一块内存而犹豫不决。一个简单的 256x256 的 `tileMap` 需要 65,536 个 `tile` 。面向对象程序员可能会觉得 6 万多个对象相当耗费。对

他们来说，只在必要的时才为 `tile` 分配对象可能更有意义，甚至到了在编辑器中真的有 65000 个人工创建的 `tile` 的地步。但正由于它们是人工放置的，必要性就被确定了，于是就变成了不得不处理的确切问题。

缺乏对底层的认识，不仅会导致用糟糕的方式处理渲染、放置元素，同时在解释元素的位置时，也引入更高的复杂度。在无网格的形式上访问元素往往会有一些障碍，比如保有相邻元素的链接（需要保持更新）；或需要执行整个元素列表（开销很大）；或引用一个辅助的增强网格对象（或空间映射系统）管理那些被游戏设计限制移动的对象（原本可以自由移动的）。这种无网格设计带来的虚假的自由，流露出对数据的理解不足，并且已经给一些游戏造成了显著的性能损失。同样也是对程序员心智的极大浪费。

除了当用不用网格系统，很多游戏还将每个对象都实例化，而不是用一个变量保存物品数量。对于某些游戏，这算种优化，因为创建、销毁对象也会产生相当大的开销。但这种趋势实在令人担忧，这种存储方式将游戏的数据结构埋藏至深处。

许多游戏都试图把关于玩家的所有信息都保存在玩家类里。如果玩家在游戏中死亡，则须作为一个已死亡的对象继续存在，否则将无法访问成就数据。将数据是什么、存在哪里、与谁共享生命周期的联系到一起，带来单体类以及种种难以理清的关系。而这些关系被也随后被证明是最大的 bug 来源。在这里我不会提及任何游戏名称，不只是一个游戏，也不只是一个工作室，这是种不良技术设计的流行病。似乎那些使用现成的面向对象引擎的人比那些自己开发的人更易感，而且绝不局限于某个范式。

面向数据设计并不会把现实问题的模型引入到代码里。资深的面向对象开发者常将其看作是面向数据方法的缺陷，因为面向对象设计的成功范例来自于：把人类的概念带到机器上，然后在这个中间地带，可以写出一个人类和计算机都能理解的解决方案。面向数据方法则把问题域留在设计文档中，因而放弃了些人类的可读性，将约束和期望的因素带入到变换中。但也正是这一类操作，可以防止机器在数据层面上处理人类的概念。

现在考虑，在提倡无谓抽象的编程范式中，问题域如何成为软件的一部分。对于对象而言，我们把它包含的类及其相关的函数联系起来，将意义与数据联系起来。在高层次的抽象中，我们通过高层次的概念将操作与数据分离，而这一类概念可能并不适用于底层，从而使函数变得更难实现。

类包含数据，就赋予了这些数据语境，但有时也会限制数据的复用，影响对操作的理解。针对语境添加函数可以访问更多的数据，但很快就会导致

类中包含许多不同的数据。这些数据本身并不相关，但却得在同一个类里。因为某个操作需要语境，而这个语境由于某些原因需要更多数据，如，其他相关的操作。听起来很熟悉，引用 Joe Armstrong 的话说：”我认为缺乏复用性出现在面向对象的语言里，而不是函数式语言里。面向对象的语言的问题是它们随时携带着所有隐含的语境。你想要一根香蕉，但你得到的是一只拿着香蕉的大猩猩和整个丛林。”[4]显然这是被面向对象语言的语境引用问题困扰产生的吐槽。

使用接口（或依赖注入）消除语境间的联系倒也情有可原，但实际的联系不止如此。对象中的语境往往用于联接不同类型，不同级别的数据。比如一根香蕉，有多种不同用途，可以作为一种水果，也可以代表一种颜色，抑或是作为以字母 B 开头的单词。需要仔细考量香蕉作为实例带来的问题，香蕉同时也可以是“种类”的实例。如果从进口商品的法律角度去看，或者要获取它的营养价值的信息，显然，相对于香蕉的库存数量，是截然不同的呈现。好在还是从香蕉说起。如果谈论的是大猩猩，那么我们也会止步于：大猩猩个体的信息；动物园或丛林中的大猩猩；以及大猩猩的种类。上述示例是给同一个名字的东西的三个不同层次的抽象。至少对于香蕉，每个个体并没有多少重要的数据。现实世界中也经常能看到这种语境的联系，但在对话中我们能够很好地处理了这种复杂度。一旦开始强行规定这些语境，就使得不同语境之间产生了联系。那么原本赋予的意义就会变得脆弱不堪。

混合在一起的抽象层都很难解开，因为对每个语境进行操作的函数都会从各种类中拖入随机的数据块。也就意味着，为了保证正常访问，就不能随意删除数据。这足以阻止大多数程序员尝试大规模的软件项目。同时还有另一个问题，那就是隐藏对数据的操作，会引入不必要的复杂度。当看到链表、树、数组、map、表单、行，很容易就推测出其交互、变换方式。但如果你想对家庭、办公室、道路、上班族、咖啡馆、公园做同样的事情，往往会先陷入对问题域概念的思考中。反而因此错失了探明更好的数据表达和算法的这一类细节的机会。

很少有计算机科学的算法不能在原始数据类型上重复使用。但是当引入新类，有自己的内部数据布局，没有明确遵循现有数据结构的模式，那么就不能完全利用这些算法，甚至可能看不到它们会如何应用。把数据结构放在你的对象设计中，从它们的本质来看可能是有意义的，但从数据操作的角度来看，往往没有什么意义。

---

[4]摘自 Peter Seibel 的 *Coders at Work* [15]。

当从面向数据设计角度考虑数据时，数据只是一种存在，为了获取所需格式的输出，可以用任何必要的方式解释它。我们只关心我们做了什么变换，以及数据的最终去向。实践中，抛弃数据的含义，就减少了事实与其语境相互纠缠的几率，因此也降低了仅仅为了一两个操作而混合无关数据的可能性。

1.3 数据与统计

原则二：数据指类型、频率、数量、布局、概率。

这个原则是指，数据不仅仅是结构。对于面向数据设计，一个常见的误解是，以为只跟缓存命中 (cache miss) 有关。即便只是为了保证缓存命中率，也只是通过结构化数据，将冷、热数据分离开。这是种有效的编程技巧，但面向数据设计要考量的，是数据的所有方面。要写一本关于如何避免缓存未命中的书，需要的不仅仅是些关于如何组织结构的技巧，还需要了解当计算机在运行程序时，里面究竟发生了什么。在书里讲这些也不太现实，因为这只适用于一代的硬件和一代的编程语言。尽管最大的获益语言是 C++，而收效最大的硬件是任何存在不平衡的瓶颈的硬件，但面向数据设计并不只植根于一种语言和某些不寻常的硬件。数据的模式很重要，但是数值和数据的变换方式同样重要，甚至更重要。通过猎豹的照片来了解它能跑多快终究是纸上谈兵。要在野外环境里去看，去了解慢的真正代价。

面向数据设计模式以数据为中心。以实时的、真实的、同时也是信息的数据为支点。而面向对象的设计则以问题的定义为中心。对象不是真实的东西，而是要被解决的问题的语境的抽象表示。对象通过操作所需的数据以表示它们，不考虑硬件或现实世界的数据模式与数量。这就是为什么面向对象设计能够快速建立起应用程序的原型，允许把早期的设计文档或问题定义直接放进代码，从而快速尝试解决方案。

面向数据设计采取了另一套策略，相较于假设用户对硬件一无所知，这里选择假设用户对问题的真正性质知之甚少，并将数据模式贬为二等市民。任何一个写过大型软件的人都会意识到，一个项目的技术结构和设计经常会发生很大的变化，以至于在后来的实施过程中，几乎没有任何部分能维持初稿的设计。面向数据设计避免了资源浪费，它从不认为设计需要存在于文档之外的任何地方。通过一系列上层代码来控制事件序列，解决当前问题，并指定模式来赋予数据临时的意义，从而推进工作。

面向数据设计从已有或预期的数据中获取线索。相较于为所有可能性，或保证扩展性做规划，不如说它倾向于使用最可能的输入来决策算法。与其说计划需要支持扩展性，不如说计划要简单、可替换，并能够落实。扩展性能以后再添加，通过单元测试这张安全网，确保它简单，且仍能正常工作。好在已经有一种不需要过多考虑，就能够保证扩展性的技术了：就是利用经过多年实践开发的数据库技术。

引入关系模型后，数据库技术发生了巨大的转变。在 *Out of the Tar Pit*[13] 一文中提到了通过函数式方法变换关系模型数据结构，使得函数式关系编程[5]又向前迈进了一步。这份文献，正是一部教你如何调整数据结构匹配需求的秘籍。

1.4 数据是可变的

面向数据设计只适用于当下。它无法解决过去的问题，也不是什么新颖的方案，更不是解决潜在问题的通用方案。拘泥于过去会干扰灵活性，一味的着眼未来则又可能一场空，毕竟程序员也不是什么算命先生。以作者之浅见，很少有面向未来的系统。实际应用中，伴随着设计发生变化，面向对象设计的弱点才开始显现。

在面向对象设计的介绍中常提到：面向对象设计能很好地处理底层实现细节的变化。但实际上，仅限于那些显著的、可预期的。它无法很好地处理诸如用户需求、输入格式、数量、频率、信息传输通道等这一类更实际的变化。在 *On the Criteria To Be Used in Decomposing Systems into Modules*[14] 中提到，当时许多人会像管线一样利用模块化，在方案的各个阶段使用可执行单元。每个阶段都可以看作解决局部问题的方案。在早期的文档中，模块化通过隐藏数据得以实现。当时这还算是一种改进，但在后来的 *Software Pioneers: Contributions to Software Engineering* [12] 一书中，作者重新审视了这个问题，并提醒我们，虽然这样在开发初期根据业务状况做方案选择时更快，但同时也会增加维护和迭代成本。受到这种固有惯性的影响，面向对象的设计方法始终会有问题域与实现之间的耦合。如前所述，当问题域被引入到实现中，可以立即看到实现是否有效处理、解决了当下问题，因而可以快速做出决策。但面向对象设计的问题在于，在更高层次上的变化是不可避免的。

---

[5]译注：此处取**函数式**编程和**关系**模型的组合。

设计会因许多原因改变，偶尔也会包括实际上没有改变的时候。对设计的误解，或者是曲解，会直接改变设计，进而改变实现。面向数据代码的设计从数据层面理解其变化的意义，反过来指导设计。不同于 OOP 在封装内部操作状态，面向数据还允许在数据源发生变化时，修改代码。通常而言，对比对象的复用和突变，数据块及其变换的耦合和解耦更易实现，因而 DOD 能更好地处理变化。

数据，与其特征和用法产生了关联。把数据及其功能与对象混为一谈时，对象即为数据的画皮。数据的特征与对象关联，意味着很难从其他视角考虑数据。因而数据的用例和真实的设计，都与对象暗含的用法和特征产生联系。若数据的布局与用法相关联，而用法又与数据的特征相关联，就很难仅仅根据特征拆解数据。不同特征用到不同的数据子集时，因其 (特征) 相互交叠，便会化为重重阻碍。同时交叠的数据又会形成一个越来越大的值集，作为独立的单元在系统中到处传递。这种情况，常见的做法是将一个类重构为多个类，或将数据的所有权交给不同的类。这就是将数据与一种特征联系起来。强行赋予数据以目的。而对于静态对象，则是多个预定义的目的合集，甚至还会引入原本不存在的联系。有些目的可能不再是设计所需。然而，需求的关联总比不需求的更明显，看得见的、看不见的，随着时间的推移，关联只会越来越多。

倘若通过数据的关联性来决定其操作，如给一个类添加新方法：在数据改变或被拆分时，就很难再移除对数据的操作了；而当一个操作需要数据关联在一起，那不太方便再拆分数据了。但如果把数据与对其的操作分开，将数据的各个特征、用途，从操作与数据变换中提取出来，就不难发现：原本面向对象代码重构时会遇到的困难，变得微不足道。但也是有代价的，需要维护一份操作与其所需数据的标记（用于间接查找和访问），同时面临二者可能的不同步的风险。综上，代码保持面向对象的风格：其中对象负责保持内部一致性，效率和可变性的优先级也不是那么高。有些时候，面向对象的设计是要远优于面向数据。例如系统或硬件驱动层：Vulkan 和 OpenGL 是面向对象的，只不过对象的粒度很大，并在它自己的体系里与保持理念一致；又或者像 `FILE` 类型的面向对象方法：文件系统中的打开、关闭、读取、写入等操作。

许多刚接触面向数据设计范式的人，常会有一个误解：可以通过抽象，设计一个静态库（或一组模板）作为通用的面向数据的方案，就能够解决本书中提出的所有问题。同**领域驱动设计 (DDD)** 一样，面向数据设计是针

对产品和工作流的。这里学习的是如何做面向数据设计，而不是如何将其添加到项目中。这里遵循的基本原则是：尽管数据的类型可以是通用的，但在其使用层面却不是。数值千变万化，但常常隐含我们可以利用的模式。数据能够通用的想法，从根本上就是错的，面向数据设计则要去纠正它。应用于数据的变换，在某种意义上可以通用，但实际执行的操作及其次序，才是实质上的解决方案。源码是将数据从一种形式变换为另一种形式的秘方。不会有一个模板库去理解和利用数据中的模式，这应当是一个成功的面向数据设计的任务。诚然，我们可以建立算法来匹配数据中的模式（比如压缩），但提及面向数据设计时，这个模式是更高层次的，特定域 (domain-specific) 的，而非单纯的频率映射。

程序运行时，常会使用一些专业技巧优化性能。这样或许会降低代码可读性，但也常常因不是面向对象，或因为是硬编码的原因不被采纳。硬编码一个变换，可能要比把它包装进通用容器，再套一层算法来假装它不是硬编码来得好。如果熟悉现有模板库，直接用现成，可读性会更好；当然如果恰巧用到的是通用功能，潜在错误也会减少。但如果该功能没能很好地映射到现有的通用解决方案中，此时通过函数模板再对其扩展，无疑增加了理解代码的难度。取巧地将背后技术已被替换这一事实隐藏，会产生误导。这时候，硬编码一个实质上的新算法可能会更好，当然前提是做好充分测试。如果限定在具体数据上，只用简单真实的数据（而不是什么通用数据，通用类型）测试，测试也更好写。

1.5 数据的形态

现如今的游戏有大量不同格式的数据：针对不同平台的纹理；根据骨架、播放类型优化过的动画；音频、光照、脚本；还有由多个不同属性的 buffer 组合成的网格。只有很小一部分有固定用途，如顶点数据中的位置、UV 和法线。游戏开发中的数据很难框定，并且越来越难。许多以前无法实现的想法，现在逐渐流行。这也是为什么，需要在编辑器和工具链上花费大量时间，以便将设计师和美术们自由创作的产出，以某种形式放进引擎里。如果没有工具链、编辑器、查看器、调整工具，就不可能以同等时长产出游戏。面向对象是处理所有这些不同格式数据的方法之一。它能提供集中的视图，显示每种类型数据的归属，并根据可对其执行的操作归类。它还很容易快速添加、使用数据，但实现、封装这些不同的对象需要时间。有时对象

归类的方式，无法再添加新功能时，可能还需要大量重构。例如，在许多过去的引擎中，纹理总是每像素 1、2、4 字节。随着引入浮点纹理，这些代码就都需要做些重构了。以前，顶点着色器中无法访问纹理。所以当基于纹理的蒙皮出现时，许多程序员不得不重构引擎渲染模块，使其能够更新顶点着色器的纹理，因为更新 `transform` 用以渲染蒙皮的网格时，可能会用到。PlayStation2 面世时，或某个引擎首次使用到着色器时，“材质”这一概念，就发生了变化。从小型 3D 场景看向更开阔的世界的过程中，细节层次 (LoD) 不断变大。于是工程师们开始考虑，” 渲染” 到底意味着什么。新的硬件越来越注重对齐，因此实现不得不变得难以插入操作。许多引擎中的网格数据是为渲染优化过的，但是如果必须对网格投射射线，以确定子弹命中的位置，或用 IK，或用物理，一个实体就需要有多重呈现。从这点来看，面向对象的方法就像是拼凑起来的，只有较少的对象用以代表实物，更多的则只是容器，程序员就得从更大的构建块的角度思考。实际上，这些块只会阻碍思考，在脑海中就只剩了独立的块，而其中内在的联系很快就被抛诸脑后了。从 2D 精灵到 3D 网格，始终遵循硬件厂商的格式，自定数据流和运算单元被转化为渲染的三角形。音频波形，到 bank 文件，到包络控制的音频粒子和多层音频的回放。Tile-map，到传送门、房间，再到流式的有多级 Lod 的世界，最后到混合网格调色板 (hybrid mesh palette)、数据、特殊的混合资源。从翻书到欧拉角序列，到四元数和球形内插动画，到动画树和行为映射/树。不变的只有变化本身。

如果读者从事过游戏行业，可能已经有接触这些数据类型。许多引擎确实做了会抽象这些相对基本类型。新的数据类型被大量使用时，就会作为核心类型集成到引擎里。通常，该类型被推广之前，会当作特殊情况处理，算是在可用性和性能之间的权衡。谁都不想在游戏开发中，为尚未充分理解的元素敞开大门。那些不愿或不能投入时间了解新功能最佳实践的人，也可能会吃到闭门羹。面向对象开发中的对象，并不会呈现数据本身，转而向了解更高级的工具的用户提供各种功能。

除了代表数字资产的对象外，还有用于内部游戏逻辑的对象。每个游戏，都有一些对象仅仅为了推动游戏玩法而存在。可收集的卡牌游戏有很多纹理，但也有大量的规则、卡牌统计、玩家卡组、比赛记录，以及表示当前游戏状态的对象。这些都是为一个游戏完全定制的。游戏可能会有续作，但除非是换皮，不然游戏逻辑变化可能会相当大，因此需要不同的数据，意味着需要实现新的方法，原有的对象，实际上已经不再是前作中的那一个了。

游戏数据很复杂。第一次数据布局几乎都是受最初设计的启发。一旦开发启动，布局就需要跟上游戏开发的变化。面向对象技术能够快速实现任何给定设计，在分别实现每个单一设计时非常快，但难以胜任从一种干净或优雅的数据模式迁移到下一个。当然也有一些小窍门，比如使用基于版本的资产管理程序，或结合更新系统并变换脚本的框架，但通常情况下，游戏开发者会同时改变工具链和引擎，完全重新导出所有资产，然后一次性提交到下一个版本。如果必须同时更新多个网站；或者资产量巨大；又或者试图为同时用于多个项目的引擎提供支持，而只有其中一个项目想要更新；那这个过程或许会相当痛苦。Django 框架是面向对象方法的一个例子，它能比较优雅地处理设计的迁移。但原因是，这些对象呈现的是数据模型的视图，而非数据本身。

尝试建立出一个通用的游戏资产解决方案，到目前为止还没有一个成功案例。可能是因为所有的游戏都有很多微妙的不同，如果真的提供了一个通用的解决方案，那就不是游戏解决方案，只是一种新的语言。试图提供一个游戏可以使用的所有可能的对象类型，是不会找到解决方案的。但如果我们回到对游戏本身的思考，把它当作只是在一些数据上运行一组计算，那就有一个解决方案。截止到 2018 年，能得到的最接近的尝试是 FBX 格式，当然，它还一定程度上依赖当前的标准着色器语言。目前的解决方案似乎还有不太容易去除的包袱。由于需要通用，许多细节在以非对抗方式呈现数据的抽象过程中丢失了。

1.6 框架 (framework)

无论是从底层性能的角度，还是从上层的游戏性与交互角度，游戏开发者们对于开发的理解都声名狼藉。或许由于高性能代码与内容层代码之间的差距越来越大了。面向对象技术能很好地覆盖上层需求，生产内容的程序员们对此十分满意。而性能专家们则致力于利用硬件做更多的事情，以至于内容创作者们常常会觉得在优化过程中没他们的份。可在游戏开发中，并不存在什么"中间环节"，这可能也是为什么不采用大型计算机 [6] 的架构和性能技术。其次，游戏开发者通常不需要开发预期维护十几年的系统和应用 [7]，因而不太可能在代码封装和保护上费心，甚至不会劳神维护文档。20 世纪 90 年代末，游戏开发行业首次蓬勃发展，较大的工作室开始涌现。但那时

[6] 译注：原文为 Big-Iron companies, 代指 60 到 70 年代开发大型计算机的公司

[7] 暴雪娱乐公司的人可能有话要说。

学术界和企业的软件工程实践却备受质疑，哪里有他们的身影，哪里就出现性能骤降，来自这些行业的雇员，几乎都没能留下印记。随着游戏机变得更像标准的微机，而标准微机在设计上更接近以前的大型机，那些标准专业软件工程实践的用处开始逐渐显现。现在，游戏的规模已经发展到与硬件相匹配，但行业已经不再关注那些非游戏开发实践的方向。作为一个行业，我们应该关注前人走过的路，而最接近的学术和专业开发技术似乎是以模拟和海量数据分析为基础的。我们要面临行业特有的挑战，比如在足够多的 AI 环境中遇到的高频高异构转化需求的问题，以及网络环境中的用户距离问题，又比如 MMO 中有基于位置的事件时，面临的带宽 n^2 问题，因为每个人都在试图给其他人发消息。

随着游戏世代更迭，开发者创作游戏的时间也在增加，这就是为什么项目管理和软件工程实践在大型游戏公司里已经标准化。曾几何时，游戏开发者被视做顶尖程序员，根据需求开发新的技术；但随着不太冒险的硬件出现（最知名的是第八代 x86 架构处理器），重心从巧妙的编码实践转变成为标准化的过程。也即是说，为了确保发布日期与营销日期吻合，游戏开发进度可以调整。高调的游戏开发中，总会有随机因素存在。总会有新的原因，几乎可以保证无法准确预测项目（或某个阶段）的时长。即便不通过面向数据设计来提升游戏的运行效率，也可以靠它让游戏开发的时间表变得规律。

在游戏中引入新功能，困难之一在于数据布局。若要在现有框架内改变数据布局，就需要重新设计或扩展对象。即便没有新的数据，一个功能也可能从以前的独立系统变得突然需要密切交换信息。这种耦合往往会导致整个系统的混乱，进一步引发时间耦合和额外的边界情况。而这些情况或许只有百万分之一的复现几率。听起来好像问题也不大，但如果期望游戏能卖出几百万甚至几千万份，百万分之一的话，就是几个到几十个玩家。然后他们录下游戏的 bug 集锦传到网上，表示这游戏是垃圾，开发者都不好好干活：这么明显的 bug 都没有修复。这还不是最差的，如果这个问题是个规避内购的方法，而发现的人知道复现方式，随后这些步骤在网上大肆传播，或许足以在一个 MMO 游戏里产生一股破坏游戏内经济系统的资源流 [8]。现在怎么办？若是买断制的游戏，如果已经卖出了几百万几千万份，大可不必在意。但若是现如今的免费游戏，五百万玩家可能只算个好开局，而差评会遏制增长。绕过内购会直接扼杀收入，经济崩坏则直接断送前程。

[8] 在线漫画网站 The-Trenches 上有一则关于产品上线后发现问题并尝试修复它的漫画。`http://www.trenchescomic.com/comic/post/apocalypse`

早在 20 世纪 70 年代，大型计算机的开发人员们就有这样的担忧。由于他们的程序经常在与真实货币交易有关的数据上工作，因而软件必须以高标准构建。他们需要编写操作数据的业务逻辑，但必须确保数据是通过一套可证明的谨慎操作来更新的，进而保证其完整性，这一点非常重要。数据库技术的发展正是源于对处理和存储需求：对数据进行复杂分析，存储，更新，并保证其无论何时都有效。因此使用 ACID 测试来确保数据库的原子性 (atomicity)、一致性 (consistency)、隔离性 (isolation)、耐久性 (durability)。原子性测试确保所有事务只有“完成”或“不做”两种状态。如果一个数据库一次金融交易只更新一个账户，那可以说是非常差劲了。如果交易不是原子性的，就可能引发错误。一致性是为了确保在交易中应当发生的结果状态都会发生，也就是说，所有应触发的触发器都会被触发，即便是递归的也会，没有限制。若某个账户触发了欺诈检测需要被封禁，这一点就显得尤为重要。如果其中一个触发器失灵，数据库的使用者 (公司) 可能要因未能及时阻止账户而担责。隔离性是指确保所有发生的交易不会干扰任何其他交易行为。也就是说，如果两个交易出现要在相同的数据上工作，就必须排队，而非试图同时操作。尽管通常不会出什么问题，但它确实会引起并发问题。最后，耐久性。这是四要素中第二重要的，确保一个事务一旦完成，就要一直保持下去。在数据库术语中，耐久性意味着交易将确保以某种方式存储，即使服务器崩溃或停电时仍然存在。这一点对于联网的计算机来说是非常重要的，当服务器崩溃或连接中断时，需要知道哪些交易确实已经发生。

现代在线游戏也不得不担心类似的非常重要的数据。对于非免费的可下载内容，消费者关心的是一致性。对于付费的可下载内容，用户会关心每一笔交易。为了提供数据库 ACID 测试需求的大部分功能，游戏开发者们开始回过头来研究数据库如何应对严格的要求，找到大量关于分阶段提交、幂等函数、并发等参考，从文献中学习如何为数据库设计表。

1.7 结论和启示

前面已经谈到了面向数据设计是一种思考，布局数据，并决定架构的方式。面向数据设计时的许多决定由两个原则来驱动。在本章结束之前，让我们用一些可以直接应用的 tips 开始旅程吧。

考虑一下称谓如何影响数据。考虑临近数据对数据本身可能的影响，会把它困在灵活性受限的模型中。第一个原则：**数据不是问题域**，要考虑以下

问题：

- 是什么将数据联系起来，是概念还是隐藏的涵义
- 数据的布局是由单一角度的单一阐释定义的吗？
- 考虑如何重新解释这些数据，并按思路细分数据。
- 什么让数据独特而重要？

目标平台不是未知设备。了解数据，了解目标硬件。或者说，了解每个数据流的优先级，对每个使用者的重要程度。理解改进的成本和潜在收益。访问模式也很重要，如果在突发情况下访问数据，然后在整个应用周期内不再碰它们，就会影响到缓存命中。第二个原则：**数据指类型、频率、数量、布局、概率**，接下来考虑以下条目：

- 目标平台的最小内存单位是什么？[9]
- 读取数据后，又实际使用了多少？
- 数据访问的频率是？1 次/帧、或是 1000 次/帧？
- 如何访问这些数据的？是随机的，还是突发的？
- 数据一直在被修改，还是只读？需要修改所有数据吗？
- 数据对谁重要，它的哪些方面重要？
- 找出解决方案在带宽和延迟方面的质量限制。
- 有什么信息不在数据中？其中隐含的是什么？

[9]2018 年，大多数机器上，内存的最小单位是 64 字节对齐的块，称为缓存行 (Cache Line)。

第二章　关系型数据库

为了更好地布局数据，我们可以将现有的结构变换为线性的。将面向数据的方法应用于现有的代码和数据布局时，问题通常来自于，会隐藏，封装数据的编程范式中固有的状态复杂度。这些范式隐藏了内部状态，所以通常不会触及到，但当需要修改数据布局时，就会遇到阻碍。并不是因为它们不够抽象，无法做到在不影响用户代码正确性的前提下改变底层结构，而是因为数据结构被连接起来并赋予意义。这类耦合会很难消除。

在这一章中，我们将介绍关系模型、关系数据库技术、标准化的一些相关内容。它们是将高度复杂的数据结构和关系，变换为干净的、线性可存储数据条目集合的很好的例子。

当然做面向数据设计也不需要把数据变换到数据库风格，但很多时候，在简单的数组上工作会方便许多。本章将通过示例，告诉读者如何从有复杂连接的对象网络迁移到更简单的数组推理关系模型。

2.1　复杂状态

大多数软件中的数据，总有一些复杂或相互联系的特质。而在游戏开发中，为了保障玩家的沉浸感，需要在游戏内处理多种不同资源，在不同阶段实现听觉、视觉、甚至是触觉反馈。对于许多在面向对象设计中成长的程序员，把可用的结构类型减少到只有简单的数组似乎有些难以想象。从使用对象、类、模板、封装数据的方法，到一个只能访问线性容器的世界，是非常困难的。

Edgar F. Codd 在 *A Relational Model of Data for Large Shared Data Bank*[3] 中提出了关系模型，用于处理同数据交互的代理的，当下以及未来的需求。一个为插入、更新、删除和查询操作构建数据的解决方案。他认为这一建议在保证能良好利用数据的同时，不再那么依赖对数据布局的深刻

理解；同时还能降低引入内部不一致的概率。

关系模型能够提供一个框架。Edgar F.Codd 在 *Further Normalization of the Data Base Relational Model*[4] 中, 提出了我们至今仍在用的标准化基本术语，能系统地将最为复杂的、相互关联的状态信息，减少到只有唯一独立元组的线性列表中。

2.2 为复杂的数据寻找计算框架

数据库能以结构化的方式存储高度复杂的数据，并提供一种语言来变换数据和生成报表。SQL 语言由 IBM 的 Donald D. Chamberlin 和 Raymond F. Boyce 于 20 世纪 70 年代发明，它在能够存储可计算的数据的同时，也可以按照关系模型的形式维护数据。可游戏里没有简单的可计算数据，有的只是类和对象。都是些枪、剑、汽车、宝石、日常活动、纹理、声音、成就。看起来，数据库技术在使用面向对象时不会有什么帮助。

游戏中的数据关系可能非常复杂，乍一看似乎并不能整齐地放入数据库的行中。CD 收藏则很适合放进数据库，专辑可以整齐地排列在一个表中。但是，对于许多游戏对象来说却不适用。对于没有经验的人来说，很难用正确格式的表列来描述一个关卡文件。在正确的列项描述赛车游戏里的汽车可能也不简单。要为每个车轮设置一列吗？要为每个碰撞基元设置一列？还是只为碰撞网格设置一列？

好吧，整齐划一的数据库思维或许并不适用于游戏。但其实，只是因为数据还未标准化。这里我们会逐步执行这些标准化步骤，用以说明如何从网络模型或层级模型变换为需要的形式。我们会从一个关卡文件开始，随后读者会发现这些有几十年历史的技术，是如何拓展视野，帮助我们了解游戏数据到底在做什么的。

用不了多久，我们就会发现，数据库技术已经考虑过所有需要的操作了，只不过存储数据的方式将其掩藏了起来。任何数据结构都是在性能、可读性、可维护性、面向未来、可扩展性、可复用性之间做权衡。例如，常见的最灵活的数据库是文件系统。它有一个两列的表。一列是作为主键的文件路径，另一列是作为数据的字符串。这种简单的数据库简直就是完美的面向未来的系统。文件中可以存储任何东西。表越复杂，就越不具备未来性，也越不容易维护，但性能和可读性就越高。例如，文件没有自己的文档，但只需要数据库的模式 (Schema)，就足以理解一个设计得足够好的数据库。这

也是为什么，游戏甚至看起来都没有数据库。它们复杂到了，为了性能，已经忘了自己只是种数据变换。这种可变尺度的复杂度也影响了可扩展性，这就是为什么有人转向 NoSQL 数据库，以及文档存储这类数据归档。这些系统更像是一个文件系统，文件按名字访问，而且对结构的限制较少。同时也有利于横向扩展，如果不需要在不同机器上的多个表中保持数据一致，增加硬件就会更容易些。或许有一天，内存与最近的物理 CPU 紧密相连时，或内存芯片本身处理能力更强时，又或者桌面设备中运行 100 个 SoC 比一片 CPU 更高效时，从上层迁移到文档存储可能对应用程序会更好。但至少现在，对于本地硬件上的任务，还看不出这种处理模式究竟能带来什么好处。

这里我们不打算深入研究如何利用大型数据基元最底层的细节，如网格、纹理、声音等。现在，只把这些原始资产（声音、纹理、顶点缓冲区等）看作是如同整数、浮点数、字符串和布尔值一样基元。这样做是因为关系模型要求在处理数据时要有原子性。什么是原子性，什么不是，至今仍在争论，还没有一个绝对的答案。但是对于开发供人消费的软件，颗粒度可以植根于从人类感知的角度考虑数据。现有的 API 根据使用方式，会以不同的方式呈现字符串。例如，人类可读的字符串（通常是 UTF8）和用于调试的 ASCII 字符串之间的区别。如果能意识到所有这些东西都是资源，把它们拆分成小块，就会失去原本具有辨识度的特征。那么把声音、纹理、网格添加到这上面就顺理成章了。例如，半个句子比整个句子的作用要小得多。而且由于拆分破坏了完整性，一个句子的片断显然不能在保持其意义的前提下，连同另一个不同句子的随机片断重复使用。即使是字幕也是沿着有意义的边界做分割的。正是这种对有意义的边界的需求，给出了我们对于面向人类的软件开发中，原子性的定义。为此，在处理数据、进行标准化处理时，尽量停留在名词的层面，即可命名的片段。一整首歌可以是一个原子，时钟的一次滴答声也可以是一个原子，一整页的文字是一个原子，玩家的账号也是一个原子。

2.3 标准化数据

现在要处理一个游戏的关卡文件。这个游戏中，玩家要寻找钥匙开锁，然后进入出口所在的房间。关卡文件被一连串代码调用，用来创建和配置不同的对象集。这些对象一起组成了一个可玩的关卡，并且彼此之间也存在一些关联。首先，我们假设它包含了一些房间（有或没有陷阱），有门（可以上

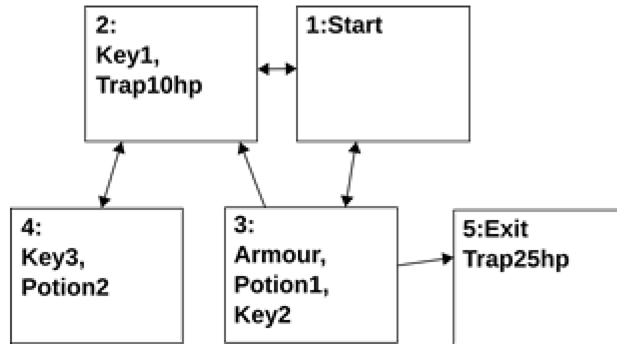


图 2.1: 图示初始化代码

锁）通向其他房间。还包含一些道具（可拾取物），有些可以用来开门，有些影响玩家的状态（如治疗药水和护甲）。所有房间和道具都有好看的的纹理网格。其中一个房间包含出口标记，还有一个房间是玩家的出生点。

初始化的代码中 (代码 2.1)，加载了一些资源，创建了一些拾取原型，建立了一些房间，在房间内添加了一些实例，然后将这些都联系起来。这里，可以看到一个处理相互引用的事物的标准方案。在把房间连接起来之前先创建房间，这是先决条件。在 C++ 中创建实体时，我们会假设它们被绑定在内存中，而唯一有效方式是通过指针引用。但在分配之前，还无法知道它们在内存中的位置。而且填充数据前，也无法分配它们，因为分配和初始化是通过 new 的机制绑定在一起的。所以很难在对象存在之前描述它们之间的关系，因而需要将内容创建这一步拆分为初始化、连接两个阶段。

```
// create rooms, pickups, and other things.
Mesh msh_room = LoadMesh( "roommesh" );
Mesh msh_roomstart = LoadMesh( " roommeshstart " );
Mesh msh_roomtrapped = LoadMesh( " roommeshtrapped " );
Mesh msh_key = LoadMesh( "keymesh" );
Mesh msh_pot = LoadMesh( "potionmesh" );
Mesh msh_arm = LoadMesh( "armourmesh" );
// ...
Texture tex_room = LoadTexture ( " roomtexture " );
Texture tex_roomstart = LoadTexture ( " roomtexturestart " );
Texture tex_roomtrapped = LoadTexture( " roomtexturetrapped " );
Texture tex_key = LoadTexture( "keytexture" );
Texture tex_pot = LoadTexture( " potiontexture " );
Texture tex_arm = LoadTexture( " armourtexture " );
Anim anim_keybob = LoadAnim( "keybobanim" );
// ...
PickupID k1 = CreatePickup ( TYPE_KEY , msh_key , tex_key ,
   TintColourCopper , anim_keybob );
PickupID k2 = CreatePickup ( TYPE_KEY , msh_key , tex_key ,
   TintColourSilver , anim_keybob );
PickupID k3 = CreatePickup ( TYPE_KEY , msh_key , tex_key ,
   TintColourGold , anim_keybob );
PickupID p1 = CreatePickup ( TYPE_POTION , msh_pot , tex_pot ,
   TintColourGreen );
PickupID p2 = CreatePickup ( TYPE_POTION , msh_pot , tex_pot ,
   TintColourPurple );
PickupID a1 = CreatePickup ( TYPE_ARMOUR , msh_arm , tex_arm );
// ...
Room r1 = CreateRoom( WorldPos (0 ,0), msh_roomstart , tex_roomstart );
Room r2 = CreateRoom( WorldPos (-20,0), msh_roomtrapped ,
   tex_roomtrapped , HPDamage (10) );
Room r3 = CreateRoom( WorldPos ( -10 ,20), msh_room , tex_room );
Room r4 = CreateRoom( WorldPos ( -30 ,20), msh_room , tex_room );
Room r5 = CreateRoom( WorldPos (20 ,10) , msh_roomtrapped ,
   tex_roomtrapped , HPDamage (25) );
// ...
AddDoor( r1 , r2 );
AddDoor( r1 , r3 , k1 );
SetRoomAsSpecial ( r1 , E_STARTINGROOM , WorldPos (1 ,1) );
//
AddPickup( r2 , k1 , WorldPos (-18,2));
AddDoor( r2 , r1 );
AddDoor( r2 , r4 , k2 );
// ...
AddPickup( r3 , k2 , WorldPos (-8,12));
AddPickup( r3 , p1 , WorldPos (-7,13));
AddPickup( r3 , a1 , WorldPos (-8,14));
AddDoor( r3 , r1 );
AddDoor( r3 , r2 );
AddDoor( r3 , r5 , k3 );
// ...
AddDoor( r4 , r2 );
AddPickup( r4 , k3 , WorldPos ( -28 ,14));
AddPickup( r4 , p2 , WorldPos ( -27 ,13));
// ...
SetRoomAsSpecial ( r5 , E_EXITROOM );
```

Listing 2.1: 初始化代码

要将这段初始化代码变成可用的数据库格式，或者说关系模型，我们要对其进行标准化处理。无论是何种类型的关系模型，引入新的元素都需要能放在表里。第一步，我们把所有的数据放到一个非常混乱的，但尽量完整的表。本例中，我们从创建对象的代码中获取数据的形式，然后将其放进表格。资源加载部分则可以直接转化为表格，如表 2.1 所示。

有了这些数据，就可以创建 `Pickups` 了。我们把对 `CreatePickup` 的调用变换为表格 (表 2.2)。注意，有一个 `pickup` 没有指定色调，这里我们用 `NULL` 表示该行没有这项内容。动画部分同理。如果只有钥匙（ key ）有动画，那所有非钥匙的行都用 `NULL`。

Meshes

MeshID	MeshName
msh_rm	"roommesh"
msh_rmstart	"roommeshstart"
msh_rmtrap	"roommeshtrapped"
msh_key	"keymesh"
msh_pot	"potionmesh"
msh_arm	"armourmesh"

Textures

TextureID	TextureName
tex_rm	"roomtexture"
tex_rmstart	"roomtexturestart"
tex_rmtrapped	"roomtexturetrapped"
tex_key	"keytexture"
tex_pot	"potiontexture"
tex_arm	"armourtexture"

Animations

AnimID	AnimName
anim_keybob	"keybobanim"

表 2.1: 根据资源名称创建的表格

资源加载完成，并且 `pickup` 的原型创建成功后，就可以为房间创建表了。要在实例没有的属性位置用 `NULL` ，以确保有值可用。我们把对 `CreateRoom`、`AddDoor`、`SetRoomAsSpecial` 和 `AddPickup` 的调用变换为

Pickups

PickupID	MeshID	TextureID	PickupType	ColourTint	Anim
k1	msh_key	tex_key	KEY	Copper	anim_keybob
k2	msh_key	tex_key	KEY	Silver	anim_keybob
k3	msh_key	tex_key	KEY	Gold	anim_keybob
p1	msh_pot	tex_pot	POTION	Green	NULL
p2	msh_pot	tex_pot	POTION	Purple	NULL
a1	msh_arm	tex_arm	ARMOUR	NULL	NULL

表 2.2: 根据 CreatePickup 的调用创建表格

Rooms 表中的列。参见表 2.3，了解如何建立一个表示所有设置函数调用的表格。

Rooms

RoomID	MeshID	TextureID	WorldPos	Pickups	...
r1	msh_rmstart	tex_rmstart	0, 0	NULL	...
r2	msh_rmtrap	tex_rmtrap	-20,10	k1	...
r3	msh_rm	tex_rm	-10,20	k2,p1,a1	...
r4	msh_rm	tex_rm	-30,20	k3,p2	...
r5	msh_rmtrap	tex_rmtrap	20,10	NULL	...

...	Trap	DoorsTo	Locked	IsStart	IsEnd
...	NULL	r2,r3	r3 with k1	true WorldPos(1,1)	false
...	10HP	r1,r4	r4 with k2	false	false
...	NULL	r1,r2,r5	r5 with k3	false	false
...	NULL	r2	NULL	false	false
...	25HP	NULL	NULL	false	true

表 2.3: 根据 CreateRoom 和其他调用创建的表格

根据 Setup 代码生成第一批表后，能看到表中出现了很多 NULL。如果实例没有某个元素，就都用 NULL 替换。还有有一些单元格包含多项数据。表中有多个门的房间就很难处理。如何知道它有哪些门呢？门是否锁住了？道具亦然。标准化的第一步，是将每个单元格中的元素数量减少到 1，并把为空的地方增加到 1。

2.4 标准化

SQL 最初应用时，明确定义的标准化阶段只有三个。而现在已经增长到了六个。如果要尽可能利用数据库技术，最好了每一个阶段，至少要知道为什么。这些范式能让增进对数据依赖的理解，若加以利用，可以帮助重排数据布局。对于游戏结构，BCNF (Boyce-Codd normal form，稍后会做解释) 应该已经足以应付大部分情况。此外，读者可能还希望针对热/冷访问模式标准化数据，不过这个主题并不属于常规的数据库标准化文献。如果读者除本书涉及的内容之外仍旧感兴趣，可以阅读 William Kent 写的 *A Simple Guide to Five Normal Forms in Relational Database Theory*[9] ，那句著名的 *The key, the whole key, and nothing but the key.* 就出自这里。

如果一个表格属于第一范式，那每个单元格都包含一个且只有一个原子值。也就是说，值中没有数组，也没有 `NULL`。第一范式还要求每一行都是独立的。不过在展开讨论之前，我们先来了解一下什么是主键。

2.4.1 主键 (Primary key)

所有的表都由行和列组成。数据库中，每一行都必须是唯一的。这个约束非常重要。在接下来了解数据标准化过程里，我们很快就知道，为什么行重复毫无意义。从编程的角度，我们先把表看作集合，那一整行就是集合的值。集合是无序的，数据库中的表也是无序的，很接近真实情况。数据库管理系统 (DBMS) 会依赖于隐式的行 ID：行与行之间总会有点区别。不过最好不要依赖这一点，只有在数据库的用途与设计相匹配时，才能更有效地工作。表都需要一个键。键常用于排序物理介质中的表，进而优化查询。因此，键得是唯一且小的。可以把键想象成 `map` 或字典中的 `key`。由于具备唯一性，每个表都会有一个隐式的键。表在同时用到所有列时，也能精确识别到每一行。因此，也可以使用整个行来作为主键的定义，或用于精确查询。如果该行是唯一的，那么主键也是唯一的。但一般而言，需要尽量避免使用整行作为主键。虽然有时也别无他选，之后我们会看到这样的例子。

例如，在 Mesh 表中，`meshID` 和文件名的组合肯定是唯一的。但它是人为确保的：我们已经假定 `meshID` 是唯一的。即便是同一个 Mesh，从同一个文件加载，仍可能会有不同的 `meshID`。纹理表中的 `textureID` 和文件名也是如此。从表 2.2 看出，我们可以根据类型、网格、纹理、色调、动画来唯一地确定每个可拾取物的原型。

再来看 Rooms 表。可以看出，只用房间表中的 `RoomID` 以外的所有列的组合，就已经能唯一地定义房间了。从另一个角度想，如果某行有相同的数值组合，那实际上就是在描述同一个房间。因此，可以认为 `RoomID` 是作为其余数据的别名。我们已经把 `RoomID` 加到表里了，但它是怎么来的？首先，它来自于初始化代码。代码里有一个 `RoomID`，但创建阶段还用不到。后面用它来确定门通向哪里。换个角度，如果房间跟任何东西都没有逻辑关系，就不需要 `RoomID` 了，毕竟也用不到。

主键必须是唯一的。以 `RoomID` 为例，它能唯一地描述一个房间，因此能作为主键。而且由于其本身不包含数据，只是作为句柄，因此也可以将它理解为一个别名。某些情况下，主键也是信息，我们以后也会遇到。

顺便提一下，数据库中的一行也是键这一点，值得花时间去理解。如果数据库表是一个集合，插入一条记录，实际上只需要将特定的数据组合记录下来。一个数据库表就好像是巨大值域中的稀疏集合。在某些情况下，可能值的集合范围没多大，就可以很方便地用位集表示。举例来说，有一个表格，其中列出了 MMO 中当前在线的玩家。如果是服务器分区的 MMO，每个服务器上唯一的玩家数量可能有几千人的限制。这种情况下，将当前在线玩家存储为一个位集会更方便。如果在线玩家只有 10,000 个，并且无论何时，同时在线的玩家都只有 1000 个，那么位集的表示方法将占用 1.25kb 内存。而如果将玩家存储为用 `short` 的 ID，需要至少 2kb 的数据。而如果是 `32bit`，要保证在多个服务器上都是唯一的，就需要 4kb。这种情况的另一个好处是数据查询的效率。快速访问表中的 ID 需要先对其进行排序, 其最佳情况是 $O(logn)$。而位集则是 $O(1)$。

回到资源表，有个细节需要提及：即使有两个不同的 `MeshID` 指向同一个文件，大多数程序员也会凭直觉理解，一个 `MeshID` 基本不会指向两个不同的 Mesh 文件。基于这种不对称性，可以推断，看起来更有可能是唯一的那一列，就是可以作为主键的那一列。在这里我们选择 `MeshID`，因为其更易操作，而且基本不会有多个含义和用途。但是，我们也完全可以用文件名替代它。

如果把 `TextureID`、`PickupID`、`RoomID`作为这些表的主键，就可以考虑使用第一范式了。我们用 `t1,m2,r3` 等来作为类型安全的 ID 值。实际上也可以直接使用整型数。这里主要是为了保证可读性，同时表明，每种类型都有该类型唯一的 ID，也与其他类型有相同 ID。例如，房间有整数 ID，值为 0 (译注：r0)，但纹理也有（译注：t0)。拥有跨类型唯一 ID 的好处是方便

调试，比如只用高位的几 bit。如果不是每个类型都拥有上百万个实体，那就有足够的位来处理上千个不同的类。

2.4.2 第一范式 (1st Normal Form)

第一范式可以理解为，消除元素的稀疏性。首先需要确保表中没有空指针，且元素中没有数组。可以通过将重复内容及可选项转移到其他表中来实现。任何有 NULL 的地方，表示该列为可选项。我们先来处理 Pickups 表，它有可选元素：ColorTint 和Animation。现在创建两个新表：PickupTint 和 PickupAnim，这里直接与 Pickups 使用相同的主键。表 2.4 是变换后的结果，可以看出，现在这里已经没有 NULL 了。

Pickups

PickupID	MeshID	TextureID	PickupType
k1	msh_key	tex_key	KEY
k2	msh_key	tex_key	KEY
k3	msh_key	tex_key	KEY
p1	msh_mpot	tex_pot	POTION
p2	msh_mpot	tex_pot	POTION
a1	msh_marm	tex_arm	ARMOUR

PickupTints

PickupID	ColourTint
k1	Copper
k2	Silver
k3	Gold
p1	Green
p2	Purple

PickupAnims

PickupID	Anim
k1	anim_keybob
k2	anim_keybob
k3	anim_keybob

表 2.4: 1NF 的 Pickups

可能你已经发现了两点不同：其一，标准化创建了更多的表，但每个表的列更少；其二，重要的事物才有行。前者令人担忧，它表示会占用更多内存。而后者就有意思了：在面向对象方法中，允许对象有可选属性；也就意味着在使用前需要判断其是否为空。但如果像现在这样存储数据，我们就知道所有的属性都不为空。不需要检查空值，代码可以更简洁。基于此，在尝试推理整个系统时，要考虑的状态也更少。

再来看 `Rooms` 表。这个表中有不少包含多个原子值的元素。由于不符合第一范式的规则，我们要先把它们从表里删去。首先删除对 `Pickups` 的引用，因为它的元素数量不定。然后是 `Trap`，尽管一个房间最多只有一个陷阱，但也可能没有。最后是 `Doors`，虽然每个房间都有门，但往往不止一个。这里重申一下规则，在每一个行与列的交汇处都有且只有一个条目。表 2.5 展示了如何只保留与 `RoomID` 有一对一关系的列。

Rooms

RoomID	MeshID	TextureID	WorldPos	IsStart	IsExit
r1	msh_rmstart	tex_rmstart	0,0	true	false
r2	msh_rmtrap	tex_rmtrap	-20,0	false	false
r3	msh_rm	tex_rm	-10,20	false	false
r4	msh_rm	tex_rm	-30,20	false	false
r5	msh_rmtrap	tex_rmtrap	20,10	false	true

表 2.5: 1NF 的 Rooms

现在我们为 `Pickups`、`Doors`、`Traps` 制作新的表格。在表 2.6 中，可以看到为了满足第一范式而做出的诸多选择。我们把类似数组的元素拆分为独立的行。注意，这里用了多行来指定同一房间里的多个 `Pickups`。还有，门现在需要两个表了。第一个表用于确定门的位置和通向。第二个表看着类似，但只包括了被锁住的那些。实际情况是：需要通过 `LockedDoors` 表中的主键识别哪些门是锁的。再来看 `Doors` 表，显然这两列都不能作为主键：两列值都有重复。但这里数值组合是唯一的，因而主键是由两列组成的。在 `LockedDoors` 表中，`FromRoom` 和 `ToRoom` 均可用于查询 `Doors` 表中的内容。这种被称为外键 (Foreign Key)，表示这些列能够直接映射到另一个表的主键。在这里，主键是由两列组成的，所以 `LockedDoors` 表有一个大外键，并且在外表 (Foreign Table) 中有关于该条目的额外细节。

随关卡文件变得复杂，空条目和数组的量也会随之增加。因而像这样布

PickupInstances

RoomID	PickupID
r2	k1
r3	k2
r3	a1
r3	p1
r4	k3
r4	p2

Doors

FromRoom	ToRoom
r1	r2
r1	r3
r2	r1
r2	r4
r3	r1
r3	r2
r3	r5
r4	r2

LockedDoors

FromRoom	**ToRoom**	LockedWith
r1	r3	k1
r2	r4	k2
r3	r5	k3

Traps

RoomID	Trapped
r2	10hp
r5	25hp

表 2.6: 应用 1NF 的 Rooms 补充表格

局数据，在大型项目中的空间占用反而会比较少。同时还能够在避免重新评估原始对象的前提下，添加新的功能。比如要添加怪物，通常情况下，不仅要为怪物添加一个新对象，还要把它们添加到房间对象中。而在新的格式下，要做的就只是添加一个新的表格，如表 2.7 所示。

于是，在没有触及任何关卡原始数据的情况下，我们就已经知道怪物的信息及其刷新点了。

Monsters

MonsterID	Attack	HitPoints	StartRoom
M1	2	5	r3
M2	2	5	r4

表 2.7: 添加怪物

2.4.3 第二范式 (2nd Normal Form)

第二范式用于分离那些只依赖部分主键的列。或许会有需要复合主键的表，但其行中的某些属性，只依赖于该复合主键的一部分。例如，表 2.8 中由品质和类型定义的武器，可以看到主键必须是复合的，该表中没有元素唯一的列。

Weapons

WeaponType	**WeaponQuality**	WeaponDamage	WeaponDamageType
Sword	Rusty	2d4	Slashing
Sword	Average	2d6	Slashing
Sword	Masterwork	2d8	Slashing
Lance	Average	2d6	Piercing
Lance	Masterwork	3d6	Piercing
Hammer	Rusty	2d4	Crushing
Hammer	Average	2d4+4	Crushing

表 2.8: 1NF 的武器

不难看出，该表的主键应该是`WeaponType` 与 `WeaponQuality` 的复合键。根据当前武器查询伤害和伤害类型，非常正常的操作。再仔细看，伤害类型并不依赖于 `WeaponQuality`，而只依赖于 `WeaponType`。这就是上文中，取决于部分键的意思。尽管每个武器的定义都符合第一范式，但其伤害类型对主键的依赖性太小，因而其不满足第二范式。我们在表 2.9 中将该表分离出来，删除了只依赖 `WeaponType` 的那一列。如果发现有一种武器的伤害类型还会依赖于其品质，那就把这个表再复原回去。比如武器：*严重损坏的晨*

星[1]，不再造成穿刺伤害，现在造成打击伤害。

Weapons

WeaponType	**WeaponQuality**	WeaponDamage
Sword	Rusty	2d4
Sword	Average	2d6
Sword	Masterwork	2d8
Lance	Average	2d6
Lance	Masterwork	3d6
Hammer	Rusty	2d4
Hammer	Average	2d4+4

WeaponDamageTypes

WeaponType	WeaponDamageType
Sword	Slashing
Lance	Piercing
Hammer	Crushing

表 2.9: 2NF 的武器

考虑第二范式的关卡数据的时，可以留心一些转移回第一范式的捷径。首先，不一定要使用 PickupID，可以通过 PickupType和 TintColour 来引用可拾取物。不过实际操作时会比较麻烦，而且因为护甲没有色调，反而会引入空项。表 2.10 就是这种情况，其中因为引入了 PickupID, 使得其与房间之间的关系变得尤为复杂。如果没有 PickupID，要把可拾取物放进房间，就需要有两个表。一张是有色调的可拾取物，另一张是没有的。虽然没那么蠢，但这种特殊条件下，好像也没那么纯粹。但是这种情况迟早会有，也算是正确选择了。

再回来看原先的 Pickup 表。已知 PickupID 是 PickupType 和 ColourTint 组合的别名，就可以应用之前变换到 1NF 时的方法。即，将 MeshID 和TextureID 移到他们自己的表中，用对 PickupType 的依赖替换掉原先 PickupType 与 ColourTint 的复合键。

表 2.11 中，资源现在依赖于其完整的复合键了。

现在还无法对房间表做同样的标准化处理。表中的 RoomID 可能是整个行的别名，也可能只是 WorldPos 的别名。但两种情况下，MeshID、TextureID、

[1]译注：晨星即钉头锤，或因其外观而得名。

Weapons

MeshID	**TextureID**	**PickupType**	**ColourTint**
mkey	tkey	KEY	Copper
mkey	tkey	KEY	Silver
mkey	tkey	KEY	Gold
mpot	tpot	POTION	Green
mpot	tpot	POTION	Purple
marm	tarm	ARMOUR	NULL

使用 1NF 标准化：

1NF 的 Pickups

PickupType	MeshID	TextureID
KEY	mkey	tkey
POTION	mpot	tpot
ARMOUR	marm	tarm

1NF 的 TintedPickups

PickupType	ColourTint
KEY	Copper
KEY	Silver
KEY	Gold
POTION	Green
POTION	Purple

表 2.10: 0NF 和 1NF 的 Pickups

`IsStart` 的值之间都有关联。关键是，它还依赖外表中的条目。这样看来，`MeshID` 和 `TextureID` 都直接依赖于表中的 `RoomID`。

Pickups

PickupID	PickupType
k1	KEY
k2	KEY
k3	KEY
p1	POTION
p2	POTION
a1	ARMOUR

PickupTints

PickupID	ColourTint
k1	Copper
k2	Silver
k3	Gold
p1	Green
p2	Purple

PickupAssets

PickupType	MeshID	TextureID
KEY	msh_key	tex_key
POTION	msh_pot	tex_pot
ARMOUR	msh_arm	tex_arm

PickupAnims

PickupType	AnimID
KEY	key_bob

表 2.11: 2NF 的 Pickups

2.4.4 第三范式 (3rd Normal Form)

在进一步标准化之前，得先清除传递式的依赖。这里指的是表中的任何一列都只跟主键有依赖。这里快速过一下当前表格，可以发现，所有资源引用都依赖于 `MeshID` 和 `TextureID`。每个有 `MeshID` 的东西都有相应的 `TextureID`。所以，可以从所有表中剥离其中一个，生成一个新表用作查询。这里随机用了`TextureID`作为主键，并将网格和纹理信息填进表里 (表 2.12)。

WeaponDamageTypes

TextureID	TextureName	MeshName
tex_room	"roomtexture"	"roommesh"
tex_roomstart	"roomtexturestart"	"roommeshstart"
tex_roomtrap	"roomtexturetrapped"	"roommeshtrapped"
tex_key	"keytexture"	"keymesh"
tex_pot	"potiontexture"	"potionmesh"
tex_arm	"armourtexture"	"armourmesh"

表 2.12: 3NF 的资源

2.4.5 Boyce-Codd 范式 (Boyce-Codd Normal Form)

一个房间用到的资源跟是否有陷阱，或是否是起始点有关。这种属于功能上的依赖，而非直接依赖。所以我们引入一个新列来描述这些内容，同时需要有中间数据用于间接查询，并促成房间和资源间的解耦。房间里可以有陷阱，也能作为起始房间，而与房间产生关联的资源取决于这两个属性，而非房间本身。这就是为什么 Boyce-Codd 范式 (或 BCNF)，能作为功能依赖的标准化阶段。

2.4.6 域键和领域知识

域键范式 (Domain Key Normal Form) 一般作为最后的标准化步骤。但如果想开发高效的数据结构，最好尽早准备并研习这部分。领域知识 (Domain Knowledge) 这个术语可能对程序员而言更熟悉些，它更直接，能够应用于键和表之外。领域知识是指数据依赖于其他数据，但只是给定其所在领域的信息。它可以很简单，就好比对某个事物的通俗认知，比如知道某个摄

Rooms

RoomID	WorldPos	IsStart	IsExit
r1	0,0	true	false
r2	-20,10	false	false
r3	-10,20	false	false
r4	-30,20	false	false
r5	20,10	false	true

Rooms

IsStart	HasTrap	TextureID
true	false	tex_rmstart
false	false	tex_rm
false	true	tex_rmtrap

表 2.13: BCNF 后的 Rooms 表

氏或华氏度是热的；或某个国际单位 (SI) 是否与人造概念有关，比如 100 (米/秒) 很快。

领域知识能够帮助发现问题：比如将人类的价值判断放入断言。试想一个捕捉物理系统爆炸[2]的断言。加速度的有效范围是什么？将其乘以 10，在事情失控之前，就有了一个检查。

一些应用会使用模糊的倒计时，替代传统的易误判的单位。如几分钟后或喝杯咖啡的时间。但领域知识不仅仅能够呈现人对数据的解释。例如，声速、光速、特定道路网上的限速和平均速度、心理声学特性、水的沸点，以及人对特定视觉输入的反应时间。这些事实在某种意义上有其用处。但只在程序员将其转化为程序性的，或作为特定实例的属性专门添加进来时，才可用。

再来看关卡数据，可以根据基本名来推断完整的文件名。纹理和网格名称使用相同格式。所以避免存储完整的文件名，便是一个领域知识的范式。

领域知识能让我们剔除一些不必要的数据。编译器的工作是分析代码输出（抽象语法树），为自己提供数据，在此基础上推断并使用其领域知识，了解哪些操作可以被省略、重排、变换，以产生更快或更低占用的汇编。而我们人的工作，是为编译器不知道的信息做同样的处理，例如，战斗中的人能够听到另一个房间里的硬币掉落的几率是多少？

[2]译注：blow-up，但不确定此处是否有专门的术语。

caption = BCNF 后的资源表 [!ht]
AssetLookupTable

AssetID	StubbedName
ast_room	"room%s"
ast_roomstart	"room%sstart"
ast_roomtrap	"room%strapped"
ast_key	"key%s"
ast_pot	"potion%s"
ast_arm	"armour%s"

领域知识催生了 JPEG 和 MP3 等格式。思考哪些是可能的，哪些能被感知，哪些会被用户行为影响，都能减少应用程序的工作量，并降低其复杂度。当玩家在有物理的游戏中跳跃时，也不必因为反作用力把世界向下移动几分之一纳米吧。

2.4.7 反思

对数据进行标准化处理时，我们看到的是一种按依赖关系分割数据的趋势。许多第三方引擎和 API 里，都能看到与这些标准化的影子。参与这些引擎设计和迭代的人，不可能拿着数据去应用数据库标准化技术。但有时对象和及其组成之间，可能分离的很明显，不需要标准的技术就能实现一些积极的结构变化。

一些游戏中，实体对象不单单是可以是任何东西的对象，而是游戏中实体类型的特定子集。例如，游戏中，可能有一个玩家角色的类，有不同类型的敌人角色的类，还有车辆类。玩家可能拥有不同于其他实体的属性，例如，无 AI 控制，玩家可控，可回复生命，有弹药等。这种面向对象的方法，在对象的类和实例之间划了一条线。虽然对用户来说不可见，但它会干扰开发者。同时也具备侵入性，类之间接触时，需要适应彼此。而如果几个类在不同的层次结构中，还必须通过抽象层传递信息。弥合这些差距所需的代码量或许不多，但终归只会让系统变得更加复杂。

实践中，常常是：实现对多个类执行操作的模板花费的时间，远多于将类向着离散化方向重构。就如同要考虑，是否有针对所有大于零的对象执行操作的可能，基本上属于浪费时间。而重构组件需要花费努力，通常与创建有效的模板操作不相上下。

如果没有类来定义边界，基于表的方法，将对数据的操作放到了同一水平线上。通过标准化关卡数据，我们已经知道，数据需要随同设计一起改变，并且尽量不要使状态变得不一致。我们常常在毫无必要的时候，把事情变得复杂，而能带我们走得更远的，只有实践和经验。

2.5 操作

在面向对象的情况下，可以直接通过调用方法对其执行操作。那在基于表的方法中，我们要如何打开门锁呢？总归是要有插入、删除、更新。在 Edgar F. Codd 的文章中明确规定了这些行为。也即是操作关系模型的全部。

真正的数据库中，要找到需加载的网格，或检查某扇门是否上锁，通常需要表与表之间相互连接。数据库也会通过改变操作的顺序来优化连接，使预期开销尽可能小。但我们能做得更好。因为查看和请求表内数据的方式完全控制在我们手中。要检查一扇门是否上锁，这里不需要连接表，可以直接查到上锁的门的表。另外，虽然数据有跟数据库一样的布局，但不代表就必须得用查询语言访问。

涉及到改变状态的操作时，最好尽量模仿 DBMS 中常见的那种操作，意外的操作会引入复杂度。例如，假设分别有一份打开的门的表，和关闭的门的表。在表之间相互移动门显然是种浪费。所以可以考虑将其改为单表，所有关闭的门在一端，打开的门在另一端。将两个表合并成一个，并给数组设置截止点，隐式地定义 `isClosed` 属性。比如在代码2.2 中，该表在某种程度上是有序的。这种内存优化也有其代价。在表中引入顺序，使得难以并行操作整个表。所以，这些改变就要多加留意，警惕其引入的复杂度，并整理好文档。

```
typedef std::pair <int ,int> Door;
typedef std::vector <Door> DoorVector
DoorVector gDoors;
int gDoors_firstClosedDoor = 0;
AddClosedDoor ( Door d ) {
   gDoors.push_back();
}
AddOpenDoor ( Door d ) {
   gDoors.insert(gDoors.begin() + gDoors_firstClosedDoor, d);
   gDoors_firstClosedDoor += 1;
}
```

Listing 2.2: setup 代码

开一扇门锁可以被当做是一次删除操作。一扇门被锁住，是因为在 LockedDoors 表中有一个条目，对应玩家可能需要交互的门。如果门和玩家持有的钥匙匹配，那开锁就是一次“删除”。

玩家的背包则是一个只有 PickupIDs 的表。这就是之前提到的”主键也是数据”。如果玩家进入一个房间，并拾取一个道具，那么将删除与该房间相对应的条目，而背包里会更新刚才的 PickupID。

数据库里有一个概念，触发器：对一个表的操作会引发一系列额外操作。捡起一把钥匙时，我们希望在放入背包时有一个触发器，将新的PickupID 与 LockedDoors 表连接起来。找到匹配的行，并删除它，门就解锁了。

2.6 总结

可以看出，数据库能胜任存储任何高度复杂的数据结构；即便是高度相关、设计快速变化的游戏数据，也不在话下。

游戏中有很多状态，而关系模型能提供一个强大的结构，可以保存静态、动态的信息。在这种结构下，实践中，相似的问题就有相似的方案，而相似的方案就会有相似的处理。使用时，数据布局更易推断，因而算法和技术也更易复用。

如果想找一种方法，能将相关联的复杂对象，变换为更简单扁平的内存布局。那可能很难比按照标准化形式来变换做得更好。

以数据库的形式存储数据还有一些比较有用的“福利”。它允许旧的可执行文件在新数据上运行，新的可执行文件也更容易在旧数据上运行。试想，如团队中有人分别使用新旧版本一起工作时，就会很有用。可以看到，有时添加新功能只需添加一个新表，或者在现有表中新加一列。此时，如果是用的数据库式的存储，这次修改就是非侵入性的；但如果要在类中添加一

个新成员，恐怕就会是一次重大改变。

2.7 流处理

现在我们知道，游戏数据和运行时，都能用类似数据库的方式来实现。并且显然，游戏数据可以实现为流。如果长期存储是数据库，运行时的数据格式与磁盘上的一样，那么，我们能从中得到什么好处？数据库可以看作行或列的集合，也可以看作表的集合。此处的集合，指的是属性所有可能的排列组合。

对于大多数应用，用比特集来表示一个表会很浪费，大小很快就会超出任何硬件所能承受的范围。但从处理角度，这一点还是值得注意的。处理一个集合，将其转化为另一个，可以看作是遍历该集合并输出新集合。但集合有意思的点就在于，它是无序的。无序的表很容易并行处理。任何时候，只要有机会利用这种明显的并行性，就都能获得巨大好处；而由于面向对象方法的数据布局原因，我们通常无法接近这一点。

从另一个角度看，多年来，显卡供应商也一直朝着这个方向努力。我们现在也要以这种方式思考游戏逻辑。只要尽可能利用流处理或集合处理，并尽量减少随机存取，就能快速处理大量数据。这种情况下，流处理意味着，在处理数据时不写入进程外部的变量。意味着避免使用类似全局累加器这些，同时避免访问未被设置为进程输入的全局内存。这样就能够确保进程、变换能够并行。

想象为显卡准备图元渲染的场景：首先设置了一些常量，如变换矩阵、纹理绑定、光照值、着色器。运行着色器时，每个顶点和像素的可能都有自己的便签式存储 (scratchpad) 应对局部变量，但它们绝不会写入全局变量，也不会引用全局的便签式存储。通用 GPU 代码中的共享内存概念，如 CUDA 和 OpenCL 中使用的管理型缓存。没有哪一种 GPGPU 技术提供对全局内存的访问。因而它们能保持明确的领域分离，并持续保证，任意内核都能在其沙盒共享内存之外运行时，不会产生副作用。强制使用这种没有副作用的方法，就能确保这明显的并行性，因为操作顺序已经确定是不相关的。如果允许着色器写入全局，就会有锁，或者它会变成一个固有的串行操作。两种情况对于显卡这种的核心数目巨大的设备，都不是好事。所以一直以来，这都是从设计层面考量的，主动的限制。如果让共享内存参与进来，就会在整个过程中引入潜在的锁，因此需要明确地只在写计算着色器时使用。

经过一些列调整，没有了全局数据 [3]，我们可以清晰地看到一条高度并行化处理的路线。现在也更容易思考、检查、调试、扩展、乃至中断以适应新设计。只要能保证无序，就可以自由执行那些易出问题的测试。

2.8 为什么数据库技术很重要?

正如本章开头提到，关系模型目前非常适用于非稀疏数据布局的开发，一旦设计好了表，就可以操作，不太需要复杂的状态管理。然而，变化才是常态。现在还常用的，忽然可能就老办法，对于大型系统，关系模型也已不再提供所有需要的功能。

随着处理更大工作量的 NoSQL 方案出现，以及大公司在分布式计算方向的投入，处理巨大的数据集在技术方面已经取得了进展。在保持数据库的实时、分布式、一致性（在容许范围内）方面，也有了进展。现在的数据库经常包括 `NULL` 条目，甚至于 `NULL` 条目远远多于数值，这些高度稀疏的数据库，需要一个不同的解决方案。许多大型计算和进程，现在都通过一种叫 map-reduce 的技术运行。分布式工作负载已经变得足够普遍，以至于人们不得不提醒，做些加法运算不一定需要用到集群。

过去十年，已经很清楚的是，大多数证明有用的高级数据处理技术，都是类似于这种组合：函数式的高级算法应用于硬件感知的数据操作层。随着 PC 中的硬件变得越来越像互联网，这些技术将开始在个人硬件上占据主导地位，无论是个人电脑、手机、还是下一代的什么。面向数据设计的灵感来自于这样一种认知，即硬件已经发展到这种程度：我们过去用来抵御从 CPU 与硬盘的延迟的技术，现在也适用于内存。将来，如果能利用大量孤立的不可靠的计算单元来提高处理能力，那么我们在这个时代开发的服务器上的分布计算技术，可能会适用于下一个时代的桌面系统。

[3]译注：globals/global scratchpads

第三章　存在性处理

如果发现苹果已经售罄，你还会砍价吗？

存在性处理旨在消除："是否要处理数据"这一类冗余查询。大多数软件中，为了确保对象在工作开始前有效，会先检查其是否为空。那如果能始终保证指针不为空呢？如果可以确保其始终有效，并且一定会处理呢？

本章中，我们会展示一种可以用于面向数据的，运行时多态技术。当然，它不是唯一的面向数据设计友好型的运行时多态。但却是作者找到的第一个解决方案，并且很适合其他游戏开发技术，如：组件和计算着色器。

3.1　复杂度

学习软件工程时，我们可能会看到，其中有提到过循环复杂度 (或条件复杂度)。这是一个用数字表示，用于分析大型软件项目程序复杂度的指标。循环复杂度只涉及到流程控制。在这里，该公式表示为为 **(1 + 被分析系统中存在的条件数)**。因此对于任何系统，它都从 1 开始。对于每个 `if`, `while`, `for`, `do-while` 都加 1。另外 `switch` 语句中除 `default` 外的每个 `case` 也要加 1。

现在，仔细考虑虚函数调用的原理，即在函数指针表中查找，并进入类方法的分支。显然，虚函数调用实际同 `switch` 语句一样复杂。虚函数调用中，想要统计流程控制的数目会比较困难。要知道复杂度度，就必须知道满足的方法数目。因此必须计算对父类虚函数的 `override` 数目。如果方法是纯虚函数，那复杂度可以 -1。然而，有时无法看到所有运行时代码，如动态加载库，那潜在的代码分支数目就会增加一个未知量。对于允许接入第三方库的系统，有必要接受这种不可见或模糊的复杂度，但需要一定程度的信任，因为这表示任意环节都没法被彻底测试。

这种复杂度通常称为控制流复杂度。软件中还有一种固有的复杂度，就

是状态复杂度。在 *Out of the Tar Pit*[13] 一文中有结论：最能提升软件复杂度的是状态。这篇论文提出了一个方案，以图最大限度减少所谓的**意外状态**，即：不直接解决问题，但在软件完成工作需要的状态。该方案还尝试废弃掉那些仅仅为了支持某种编程风格而引入的状态。

必要的控制是：实现设计时，一个功能必须在某些条件满足时才发生。例如：按下跳跃键时跳跃；存档数据变脏或计时器结束时，在检查点自动保存。

意外的控制，从使用者的角度看，对于程序是非必要的，但可能也是关乎程序能否工作的基础功能。这种控制的复杂度一般分为两种形式。第一种是结构性的，如支持某种编程范式、提供性能改进、驱动一种算法等。第二种，则是防御性编程或用于辅助开发者的，如引用计数、垃圾回收。这些技术在使用时会去确定数据是否存在，也会检查边界，因而也会增加复杂度。实践中，可以在使用容器和其他结构时看到它们，控制流会以边界检查确保数据没有以超出范围。垃圾回收也会增加复杂度。许多语言中，都难以确保回收会何时，怎样触发。也就意味很难推断对象的生命周期。使用这些语言时，人们倾向于在开发初期忽略内存分配，因此在临近交付时，可能很难修复内存泄漏。非托管语言的垃圾回收处理起来要容易些，因为引用计数更易查询，但也是因为非托管语言通常预先分配的频率较低。

3.2　调试

在高复杂度的程序中，会遇到哪些问题？分析系统的复杂度有助于了解其测试难度，反过来也有助于了解其调试难度。有些问题可以归类为处于意外状态，但也无法更进一步了。其他的可以归类为进入坏的状态：由于对无效数据做出反应而表现出意外的行为。不过，还有一些问题可以归类为性能问题，而非正确性问题。某种程度上，这些问题虽然被大量的学术文献忽视，但在实践中代价很高，而且通常来自于复杂的状态依赖关系。

例如，由缓存等优化技术引入的复杂度，是状态复杂度问题。CPU 的缓存处于一种不知道的状态，且在工作中没能预期到，就会导致性能不佳或不稳定。

许多时候，调试中的困难来自于：没有完全了解所有的流程控制点，假设已经采取了一条措施，而实际上并没有。程序按我们的要求去做，而非遵照我们的意思，它们就会进入一个预料外状态。

使用虚拟调用的运行时多态，会大大增加这种情况发生的可能性。因为不确定我们是否已经完全知道代码所有不同的分支，除非使用日志记录代码，或者用调试器来查看它在运行时的走向。

3.3 为什么要用 if

真实的游戏开发案例中，显式的流程控制语句常常属于非必要集。实行防御性编程的地方，许多流程控制语句只是为了防止崩溃。阻止越界访问，保护为 `NULL` 的指针，防御其他会使程序终止的特殊情况。好在，GitHub 上有很多高质量的 C++ 源码，与这种趋势背道而驰。它们更倾向于使用引用类型，或尽可能使用值类型。游戏开发中，另一种常见的流控制是循环。虽然这种情况很多，但大多数编译器都能识别它们，并做出很好的优化，而且在去除不必要的条件检查方面做得很好。最后一种不重要但常见的流程控制来自多态调用，它对于实现一些游戏逻辑很有用，但主要是为了满足面向对象编写游戏的方法中，部分执行的“更少代码，更多用途”开发模式。

本质上，游戏设计中的流控制，不太会在性能分析文件中以分支控制出现，因为所有支持性代码会运行得更频繁。因此可能会忽视每个条件对软件性能的影响。用条件实现 AI，处理角色移动，决定何时加载关卡的代码，会在充满循环和树状遍历的系统中调用；或对访问中的数组边界检查，返回数据，产生布尔值，最终驱动 `if` 落入其中一个分支。也就是说，当代码库其他部分都很慢时，就很难验证为其中一项任务编写快速的代码。很难讲又增加了哪些额外成本。

如果觉得值得考虑一下清除控制流，就得先了解哪些控制流操作是可以清除的。如果从防御性编程开始着手，可以用数组的集合来表示工作数据集。这样就可以保证数据中没有 `NULL`。仅这一步，就可以清除许多流程控制语句。这样并不会摆脱循环，但只要是运行纯函数式变换的数据的循环，就不必担心副作用，并且反而会更容易推理。

虚拟调用中固有的流程控制也可以避免。事实上，有许多程序是以非面向对象风格编写的。没有虚拟，还可以依赖 `switch` 语句。没有那些，还可以依赖函数指针表。再不济，还能用一连串 `if`。有许多方式能实现运行时多态。可以认为，如果没有显式类型，就不需要对其进行切换。所以如果能根除面向对象的方法来解决问题，那些流程控制语句也会完全消失。

当涉及到游戏逻辑中的控制流时，就会发现，想要根除他没那么容易。

这倒也没那么可怕，游戏开发中，游戏逻辑是我们能看到的最接近本源的复杂度。

减少条件语句，从而减少这种规模的循环复杂度，能带来不容忽视的好处，但也是有代价的。之所以能够避免检查 NULL，是因为数据格式根本不允许出现 NULL。我们很快会证明，这种不灵活性其实是种优势，但需要一种新的方法来处理实体。

以前，游戏中会有一片区域的对象实例，我们会查询它是否有去其他区域的出口；而现在，只需要查看一个只包含区域间链接的结构，并通过我们所在的区域做过滤。这种所有权的颠倒在调试中很有优势，但是当有时只想找出哪些出口可以离开一个区域时，就显得很落后。

如果读者用过购物清单或待办清单，肯定能理解，如果有个明确的要完成的清单时，效率会高很多。制定清单很容易，只要把东西添加上去即可。如果要去购物，很难用排除法得出自己需要什么。如果要规划三餐，一张清单就必不可少，除了要弄清楚要哪些原料，还要计算出所需的数量，才能保证膳食计划。如果有待办清单和日程，就可以知道谁会来，需要做什么准备。知道有多少张嘴要吃饭，要买多少食物和饮料，以及要为访客准备多少套床褥。

待办清单好就好在，可以设定一个最终目标，然后加入子任务，使一个庞大而遥远的目标看起来更加可行。加入估算可以提供一些紧迫感，而这种紧迫感，在最后期限如此遥远的情况下，通常是缺失的。许多公司使用软件来跟踪任务，这些软件通常提供一些功能，允许生产者确定关键路径、预估所需的开发人员时间，甚至是维持项目所需的技术平衡。不使用这种软件常常是公司没有不怎么关注效率，甚至是浪费的标志。如果关注项目中的效率和浪费，任务清单就不失为一个分析成本来源的好方法。通过记录并追踪这些清单，可以通过观察数据，了解软件执行中的操作的大致形态。不这样做的话，就很难定位真正的瓶颈，问题可能不是处理，而是处理数据的请求本身就已经失控了。

程序运行时，若不让它处理同质化的列表，而是有什么做什么，那效率就会很低，还会导致帧时长不规则、不稳定。低效利用硬件往往都是因为处理无法被预测。在指向异质类的大数组都被 update() 函数调用时，会遭遇大量的数据依赖，导致数据和指令缓存均出现未命中。原因详见第 14 章。

慢，还因看不到有多少工作待办，因而无法确定工作的优先级和规模，进而不能确定在给定的一帧时间内完成多少任务。若没有待办清单，也没能

力估计每项任务的耗时，就难以在保证用户反馈的同时，决定最佳行动方案来减少开销。

面向对象编程工作能在程序运行时，模式较少的情况下，工作地很好。程序只处理少量的数据，或数据有难以置信的异质性，以至于有多少种事物就有多少类。

不规则的帧时长，往往是因为没有提前对远期目标采取行动。如果读者，作为一个开发者，知道必须为一个新的岛屿加载资产，当玩家冒险进入周围的海域时，流加载系统能收到通知，载入必要的数据。可能是一个房间和远处的其他房间。也可能是玩家视线范围内的地下城或洞穴的数据。我们把这种先发制人的数据流当作一种特殊情况，并创造提供这一级别规划的系统。依靠人类（哪怕是关卡设计师）来把这些联系在一起很容易出错。许多情况下，如果没有自动检查，就会漏掉一些依赖关系链。没有一种常规语言可以描述时间上的依赖关系，因而也无法让系统有足够的自我意识去执行自我加载。

许多游戏中，我们会用显式的触发器将事情串联起来，但对于许多其他游戏元素，往往没有这样的系统。几乎从未听过过 AI 向着某个目标执行寻路，是因为马上可能会去那边。最接近的做法是，开发者预填充一个导航图 (navigation map)，这样就可以迅速粗略地完成路径选择。

还有一个先期工作的深度问题。考虑一个小房间，创建为独立资产，一个等候室有两个相邻的门，都通向很大但不同的两张地图。当玩家在地图 A 中靠近等候室的门时，这个小房间就可以被抢先流加载进来。然而，在许多引擎中，地图 B 不会流加载，因为地图 B 到地图 A 的位置特性隐藏在等候室的逻辑层后面。

物理系统做预判也不太常见，比如为了执行下一步工作，检查未来是否会发生碰撞。如果它能感知更多，或许能实现一个更复杂的破碎模拟。

如果让游戏生成待办清单、购物清单、远期目标，并允许通过前瞻性思考来采取预防措施。那就可以把程序员的任务，简化为对目标和效果做优先级排序，或编写运行时生成优先级的代码。读者可以考虑如何将这些依赖关系连锁起来，以解决等候室的问题。也就可以开始抢先处理所有类型了。

3.4 处理的类型

存在性处理与待办清单有关。处理同质的数据集时，我们已经知道，要以相同方式处理每个元素。集合中的每个元素上会执行相同的指令。这里对输出没有明确的要求。但通常归结为三种操作：**过滤** (filter)，**突变** (mutation)，**散发** (emission)。突变是对数据执行一对一操作，接收数据输入和一些在变换前设置的常数，并为每个输入生成一个唯一元素。滤波同样接收传入的数据，在变换前设置一些常数，并为每个输入元素要么生成 0 或 1 个元素。散发是对传入数据的操作，能生成多个输出元素。和其他两种变换一样，散发可以使用常数，但输出表的大小没有限制，它能生成零到无穷个元素。

第四种，也是最后一种形式，叫做**生成** (generation)，并不能算是真正的数据操作，但通常是变换管线的一部分。生成不需要数据输入，而只根据设置的常数产生输出。使用计算着色器时，就可能会遇到这样的函数，它只是对数组执行置 0 、置 1 、升序操作。

变换

突变	$in == out$	处理输入数据。每一个输入项目产生一个输出项目。
过滤	$in >= out$	处理输入数据。每一个输入项目最多产生一个输出项目。
散发	$out = \begin{cases} 0, & in = 0 \\ >= 0, & otherwise \end{cases}$	处理输入数据。每项输入产生未知数量的项目。如果没有输入，输出也是空的。
生成	$in = 0 \land out >= 0$	不读取数据。仅仅通过运行就产生了未知数量的项目。

表 3.1: 常见的变换类型

这些类别可以帮助决定：要使用什么数据结构来存储数组；是否需要一个结构；或是否应该用管线将数据从一个阶段输送到另一个阶段，而非在中间缓冲区上操作。

每个 CPU 都能有效地在核心上处理同质数据集，也就是在连续的数据上反复做相同的操作。没有全局状态，没有累加器，就证明可以并行。可以看一下 map-reduce 和简单的计算着色器，用现有的技术举例，来说明如何在这些限制中，建立真正的工作应用。无状态变换在运用分布式处理技术时

也是无害的。Erlang 依赖于没有副作用这一点，实现了线程间、进程间、乃至分布式计算的安全处理。对有状态的数据执行无状态的变换高度稳健，可以深度并行。

处理每个元素时，对于变换核心操作的每个数据，使用控制流非常合理。几乎所有的编译器都应该能将简单的局部分支指令，简化为平台首选的无分支表示。如 CMOV，或 SIMD 操作的 `select` 函数。在考虑变换内部的分支时，最好是比对着现有的流处理实现，如显卡着色器或计算核心。

在分支低阶断言 (predication)[1]中，不会忽略流控制语句，而是被用作如何合并两个结果的指标。若流控制不基于常量时，一个低阶断言 `if` 会生成代码，同时运行分支两边，并根据条件的值放弃其中一个结果，选择另一个。如前所述，许多 CPU 本身就有这个功能，不过所有 CPU 都可以使用位掩码去实现它。

SIMD (single-instruction-multiple-data, 单指令-多数据) 能够在指令相同时并行处理数据。数据可以不同，但都是局部的。没有条件语句时，SIMD 操作在我们的变换上很容易实现。在 MIMD (multiple-instruction-multiple-data，多指令-多数据) 中，每块数据都可以由一组不同的指令操作。每一块数据都可以用不同路径。它是最简单，也最容易出错的编码，目前大多数并行编程都是如此。每增加一个线程，就要用一个单独的执行线程处理更多数据。MIMD 包括多核通用 CPU。通常，它允许共享内存访问，也会有伴随而来的同步问题。目前为止，它最容易启动和运行，但也最容易出现那种，由状态复杂度引发的罕见的致命错误。因为操作顺序不确定，通过代码产生的的不同的可能路线的数量，向着超指数级爆炸。

3.5 避免使用 boolean

研究压缩技术时，我们必须了解的最重要的一点：数据和信息之间的区别。系统中存储信息的方式有很多，从表明某物存在的可解析的明文字符串，到简单到用来描述某物具有某属性的单比特标记。例如，代码中声明的局部变量，或一个物理网格中用于查询会响应哪些碰撞类型的比特集。有时可以通过先进的算法（如算术编码）或领域知识让存储的信息少于比特集。领域知识标准化适用于大部分游戏开发。但它的应用越来越少，因为很多开发者都过度热衷于引用“过早优化”，反而身陷囫囵。信息被编码进数据，而

[1]译注：尚不确定这个词的术语，用法与一阶谓词逻辑中 (或低阶断言逻辑) 相同

编码的信息量可以被领域知识放大。重点是，我们可以看到，压缩技术提供的建议是：真正编码的是概率。

举个例子，一个游戏中的实体有生命条 (一段时间不受伤害后就可以回复)，会死亡，能互相射击。我们来看看怎样利用领域知识减少处理。

```
struct Entity {
    // information about the entity position
    // ...
    // now health data in the middle of the entity
    float timeoflastdamage;
    float health;
    // ...
    // other entity information
};
list<Entity> entities;
```

Listing 3.1: 基本实体方法

```
void UpdateHealth ( Entity *e ) {
    TimeType timeSinceLastShot = e->timeOfLastDamage - currentTime;
    bool isHurt = e->health < MAX_HEALTH;
    bool isDead = e->health <= 0;
    bool regenCanStart = timeSinceLastShot > TIME_BEFORE_REGENERATING;
    // if alive, and hurt, and it's been long enough
    if( !isDead && isHurt && regenCanStart ) {
        e->health = min(MAX_HEALTH , e->health + tickTime * regenRate);
    }
}
```

Listing 3.2: 简单的生命回复

假设有以下领域知识:

- 若生命值已满，就不会继续回复。
- 一旦被击中，需要一些时间才开始回复。
- 一旦死亡，生命就不会再回复了。
- 死亡时，生命值会为零。

现在来看代码 3.1 中的实体，可以看到这里的数据会引发常见的缓存行问题。此外，代码中会如何调用 `update` 相关的函数呢？如代码 3.2 示，每次 `update` 都会针对每个实体调用一次相关函数。

这里我们可以从流程控制语句入手，做些改进。如果生命值满，函数就不执行。如果实体已死亡，也不执行。回复函数只在距离上次受伤过了足够久才执行。考虑所有这些情况，其中血量回复并非常态。所以，这里应该尝试为常见的情况组织数据布局。

现在把结构体改为代码 3.3 中所示。更新函数不再针对实体执行，而是针对生命表。因此我们知道，只要这个函数在执行，实体就没死，它就会受伤。

```
struct Entity {
    // information about the entity position
    // ...
    // other entity information
};
struct Entitydamage {
    float timeoflastdamage;
    float health;
}
list<Entity> entities;
map<EntityRef ,Entitydamage> entitydamages;
```

Listing 3.3: 存在性风格处理的生命

只在实体受到伤害时才需要添加一个新的 `Entitydamage` 元素。如果实体在已有 `Entitydamage` 的情况下受伤，它就只需更新 `health` 状态和 `timeoflastdamage`，无须再创建新的。如果想知道某人的生命情况，只需检查他是否有 `Entitydamage`，或者查看 `deadEntities` 表中是否有他。之所以能这么做，是因为每个实体都有一个隐式的布尔值，藏在已有的行里。对于 `entitydamages` 表，这个布尔值就相当于第一个函数中的 `isHurt` 变量。同样的，`deadEntities` 表中的 `isDead` 也是隐式的，表示生命值为 0。这就可以省下资源用于其它系统。不必加载一个浮点数并判断其值是否小于 0，省去了浮点比较、转换为布尔值的过程。

```
void UpdateHealth () {
    for( edIter : entityDamages ) {
        EntityDamage &ed = edIter ->second;
        if( ed.health <= 0 ) {
            // if dead , insert the fact that this entity is dead
            EntityRef entity = edIter ->first;
            deadEntities.insert( entity );
            // if dead, discard being damaged
            discard(ed);
        } else {
            TimeType timeSinceLastShot = ed.timeOfLastShot - currentTime;
            bool regenCanStart = timeSinceLastShot > TIME_BEFORE_REGENERATING;
            if( regenCanStart )
                ed ->health =ed ->health + tickTime * regenRate;
            // if at max health or beyond, discard being damaged
            if( ed ->health >= MAX_HEALTH )
                discard(ed);
        }
    }
}
```

Listing 3.4: 每个实体的生命回复

消除布尔值也不是什么新鲜事，因为每有一个指向某物的指针时，都会

引入一个非空的布尔值。正因为不想检查 NULL，才促使我们为处理“对象是否存在”寻找不同的表示方法。

其他类似的情况包括：武器换弹、游泳时的氧气余量、任意会耗尽的有数值的事物、有极值的数据等。甚至汽车的行驶速度：如果参与交通，大部分时间都会在限速区间内行驶，而非需要算出某个速度。如果有群人都朝同一个方向走，那么进入这个群体的人会一直受阻，直到与群体不再相斥。此时他可以放弃独立的想法，在群体中随波逐流。这一点，会在第五章详细介绍。

转换为保存属性状态的列表，能实现更好的性能优化。与时间相关的属性，第一要务是将其放进有序的表中，按它们应被执行的时间排序。我们可以把回复时间放进有序表，然后不断 pop 出 entityDamage，直到遇到无法被移到活动表中的元素，然后一次性跑完所有活动列表。现在就知道哪些对象受伤了，没有死，可以再生，并且可以开始回复生命了。

再来看不同时间间隔内更新的属性。动植物响应环境的机制有所不同。有的非常快，如远离危险的反应：把手抽离热锅。也有较慢的，如负责推理的脑区。也有快到近乎即时的，像是反射，是大脑在没时间详细思考时的反应，如接球、在自行车上保持平衡。大脑还有更慢的区域，比如现在，与其说你是在读这本书，不如说是在整理出一个模型，以便理解文字的含义，并最终吸收他们。还有更慢的：压力响应，如荷尔蒙弥散在体内的化学物质，当前体内能调动的糖分，当前的水合水平，所有这些组成一套相应的系统。能够在多个时间尺度上思考和反应的 AI 或许更节约资源，也不太可能出现奇怪的行为，或在决定间犹豫不定。保证每个系统每帧都更新，可能会陷入不可能的境地。将工作分成不同的更新率仍有规律可循，同时带来了能够平衡多帧工作的机会。

另一个用途位于状态管理中。若一个 AI 听到了枪声，那他可以在表格中添加一行，记录最后一次听到枪声的时间，可以用来确定他们是否处于高度警觉状态。若 AI 与玩家进行了交易，只要玩家有可能想起，那 AI 也就必记住。若玩家刚把 +5 的长剑卖给 AI，且只是离开商店一会儿，那这把剑就很有必要保存在店主 AI 的库存里。有些游戏甚至在交易过程中也不保留库存，如果玩家不小心卖掉了需要的东西，然后存档，大概会相当痛苦。

从游戏角度看，这些额外信息都是玩家与世界的互动。一些游戏中，玩家可以把自己的东西永远留在周围，它们会永远保持留下时的样子。一些开放世界 RPG 里，玩家丢在山洞里的所有东西，仍然准确出现在几个小时前

丢下的位置，这已经是是相当大的成就了。

增补数据 (tacking on data)，或者说，用动态的附加属性补充加载的数据的基本概念，已经存在了相当长时间。保存游戏通常是将动态世界与基础状态比较后的差值编码，其中一个早期用途，是在完全动态环境中，加载世界，但其后可以被摧毁、改变。一些世界生成器使用程序化地形，允许其内容创作者添加额外的补丁信息：村庄、堡垒、前哨，甚至催生出大量地形工具，用于调整生成的数据。

3.6 慎用枚举

枚举用于定义状态集。原本可以为回复中的实体设一个变量，包含 `infullhealth`、`ishurt`、`isdead` 三种状态。也可以给无效的实体设索引，枚举可用的组别。但这里，我们用表格表示所需的信息，毕竟只有两组。任何枚举都可以用各种表模拟。只需要为每个枚举值建一个表。设置枚举值即是插入，或是从一个表迁移到另一个。

使用表替代枚举时，可能会带来更多困难：找出一个实体中的枚举值会变难，因为需要检查所有能代表该实体状态的表。然而，需要获取该值的主要原因，也许是为了根据外部状态执行操作；又或是为了找出满足状态的实体以判断是否需要进一步操作。大多数情况下，这些都是不允许，也没必要。首先，访问外部状态在纯函数中是无效的。其次，任何依赖数据都应该已经是表元素的一部分了。

如果这个枚举是以前由 `switch` 或虚拟调用处理的状态或类型，就不需要再查询了。其实，要改变的是思考方式。解决方法是通过转换，将每个 `switch case`、虚方法的内容，作为操作应用在相应的表中，即对应原始枚举值的表。

如果枚举是用来确定是否可以对一个实体进行操作，比如考量到兼容性，那可以考虑用一个辅助表，来表示处于兼容状态。如果情况是，查询结果需要返回一个实体，并且需要在确定提交修改前，知道它是否处于某种状态。那可以考虑，将兼容的数据，作为输出表标准的一部分先生成；也可以在提交一个过滤操作，创建正确形式的表。

总之，之所以把枚举转换为表的形式，是为了减少控制流的影响。鉴于此，如果不使用枚举来控制指令流，就不用管了。还有一种情况，枚举的值频繁变化时，因为表到表迁移对象也是有成本的。

合理的枚举的例子如：按键绑定、颜色枚举、命名合理的小的有限值集合。返回枚举的函数，如碰撞响应（无、穿透、通过）。任何实际表现为对另一种形式的数据的查询的枚举，都是好的。这些枚举用在那些较大、难以记忆的数据表中，将数据访问合理化。有些枚举还有一个好处，就是可以帮助你在 `switch` 中捕获未处理的情况，并且某种程度上，它们也是大多数语言中的自解释 (self-documenting) 功能。

3.7 初探多态

现在来考虑如何实现多态。没必要使用虚表指针；可以使用枚举变量来指明类型。这个变量可以用于在运行时定义该结构应具备什么能力，以及要如何响应。也可以在对象上调用方法时，用来判断和选择函数。

类型的定义基于成员变量的类型时，虚函数通常会实现为基于它的 `swtich` 或函数数组。若要允许运行时加载库，就得有个系统去更新被调用的函数。简陋的 `swtich` 无法胜任，但函数数组可以在运行时修改。

现在有了一个既不优雅，也不高效的方案。数据仍然由指令负责，并且每当有意料外的虚函数出现，我们仍会在指令缓存未命中和分支预测错误的问题上煎熬。但真正避免使用枚举，并且用表来表示枚举的每个可能的值时，我们仍旧有可能，同基于指针的多态一样，兼容动态库加载。同时也能保证处理异质类型的数据流时的效率。

对于每个类，都有一个工厂类来替代类的声明，它能选择生成正确的表插入调用。同时利用存在性处理，替代了多态方法调用。表中的元素允许类的特征以隐式存在。用工厂创建的类可以很容易通过运行时加载的库扩展。只要有数据驱动的工厂方法，注册新的工厂也应该很简单。表的处理和它们的 `update()` 函数也会被添加到主循环中。

3.8 动态运行时多态

如果通过组合创建类，并允许通过对表做插入、删除来改变状态，那也就解锁了动态运行时多态。这是通常只在通过 `switch` 进行动态响应时才有的功能。

多态是指程序中的一个实例能够以不同的方式对一个共同的入口点作出反应，具体由该实例的性质决定。C++ 中，编译时的多态可以通过模板

和重载实现。运行时多态是指一个类能够为一个共同的基础操作提供不同实现，而该类的类型在编译时未知。C++ 通过虚表处理这个问题，在运行时根据隐藏在虚表指针中的类型，从该指针所指向的内存调用正确的函数。动态运行时多态是指一个类可以根据其类型以不同的方式对一个共同的调用签名做出响应，并且其类型在运行时可以改变。C++ 没有明确实现这一点，但是如果类允许使用一个或多个内部状态变量，那它就可以根据状态，以及查询核心语言的运行时虚表提供不同的响应。其他能更流畅地定义其类的语言，如 Python，允许每个实例更新它响应消息的方式。但这些语言大多数总体性能非常差，因为调度机制已经建立在动态查找之上。

现在来看代码 3.5，我们希望通过运行时方法查找，来解决不知道类型但想知道大小的问题。允许对象在其生命周期内改变形状需要做出妥协。一种方法是在类中保存一个类型变量，如代码 3.6，对象作为类型变量的容器，而非特定形状的实例。

```
class shape {
public:
   shape() {}
   virtual ~shape() {}
   virtual float getarea() const = 0;
};
class circle : public shape {
public:
   circle(float diameter) : d(diameter) {}
   ~circle() {}
   float getarea() const { return d*d*pi/4; }
   float d;
};
class
square : public shape {
public:
   square(float across) : width(across) {}
   ~square() {}
   float getarea() const { return width*width; }
   float width;
};
void test() {
circle circle( 2.5f );
square square( 5.0f );
shape *shape1 = &circle , *shape2 = &square;
printf( "areas are %f and %f\n", shape1->getarea(), shape2->getarea() );
}
```

Listing 3.5: 简单的面向对象形式的代码

```
enum shapetype { circletype, squaretype };
class mutableshape {
public:
   mutableshape ( shapetype type, float argument )
      : m_type( type ), distanceacross( argument )
   {}
   ~mutableshape() {}
```

```
    float getarea() const {
        switch( m_type ) {
            case circletype: return distanceacross*distanceacross*pi/4;
            case squaretype: return distanceacross*distanceacross;
        }
    }
    void setnewtype( shapetype type ) {
        m_type = type;
    }
    shapetype m_type;
    float distanceacross;
};
void testinternaltype() {
    mutableshape shape1( circletype, 5.0f ); mutableshape shape2( circletype, 5.0f );
    shape2.setnewtype( squaretype );
    printf( "areas are %f and %f\n", shape1.getarea(), shape2.getarea());
}
```

Listing 3.6: 丑陋的内部类型代码

另一个更好的方法是通过变换函数来处理每种情况。实现详见代码 3.7 。

虽然这样有用，但所有指向旧类的指针现在都无效了。使用句柄可以减轻这些忧虑，但大部分情况下，又增加了一层间接访问，反而会进一步拖累性能。

```
square thecircle( const circle &circle ) {
    return square( circle.d );
}
void testconvertintype() {
    circle circle( 5.0f );
    square square = squarethecircle( circle );
}
```

Listing 3.7: 将现有类变换为新类

如果使用存在性处理技术，类由它们所属的表定义，就可以运行时在表之间切换。因此可以在没有任何技巧的情况下改变行为，而不需要为需要的所有状态管理 union 来保存所有不同的数据。如果用不同的属性和能力来组合类，又要在创建后改变它们，也是可以的。如果正在更新表，实体的指针地址发生变化也影响甚微了。在基于表的处理过程中，实体在内存中移动是很正常的，所以意外反而较少。从硬件的角度来看，为了实现这种形式的多态，需要为每个类属性、能力中的实体引用提供一点额外空间，但不需要虚表指针来寻找要调用的函数。还可以优先遍历相同类型的实体来提升缓存利用率，尽管它已经提供了一种安全的方式来在运行时改变类型。

由于类是由它们所属的表隐式定义的，所以有机会将一个实体注册到多个表中。因此表明，一个类不仅可以在动态运行时具备多态性，还能具备多

面性，即它可以在同一时间内成为多个类。单一的实体可能会对同一个触发器调用做出两种不同的反应，因为它适合该类的当前状态。

这种多维分类在传统的游戏代码中并不多见，但在渲染中，通常会有几个不同的维度，如材质、混合模式、某种蒙皮、其他顶点调整，会发生在某个特定的实例上。或许在代码中看不到这种灵活性，因为它不能通过语言的自然工具获得。可能我们确实看到了，但它其实就是一些人提到的 ECS (entity component system)。

3.9 事件处理

过去，如果想监听系统中的事件，需要绑定到一个中断上。有时候可能还得琢磨一下这类代码，通常它们是给旧的、微控制器规模的硬件用。出发点也很简单，当时的处理器速度，还不足以快到轮询所有可能的信息来源并处理，但又足以接收事件并即时处理。游戏中通常这样处理事件，先注册某个感兴趣的事件，然后在事件发生时被告知。发布订阅模型已经存在了几十年，但一些语言中没有为它建立标准接口，另一些中则有太多的标准。它往往需要一些来自问题域的知识，以选择最有效的实现。

有些系统希望能获取系统中的每一个事件，并自行决定，如 Windows 事件处理。有些只订阅非常特殊的事件，但期望立即对事件做出响应，如 BIOS 事件处理程序、键盘中断。有些事件可能非常重要，并直接由发布事件的这一行为来调度，如回调。有些也可能是懒惰的，停留在某个队列中，等待之后的某个时刻被分发。最佳的方法由要解决的问题来确定。

通过利用在表中的存在性去注册的技术，让事情变得更简单，并且也能极大提升注册和取消的速度。订阅变成了插入，取消订阅变成了删除。可以用全局表来订阅全局事件。也可以有命名的表。通过命名的表，就可以使订阅者在发布者存在之前订阅事件。

发布事件时，我们会有一个选择。可以选择立即启动转换；或者排队等待新事件，直到整个转换完成，然后一次性全部派发。随着模型变得更简单、更可用，带着更通用的可能性，我们能够以新的方式，实现传统中通过轮询完成的代码。

例如：除非玩家角色在激活门的距离内，不然玩家的操作按钮的事件处理程序，不需要与门关联。当角色进入范围内时，就会在动作事件表中注册 *has_pressed_action*，并以 *open_door_(X)* 返回。这样就能避免 CPU 在弄

清玩家到底试图激活什么东西上浪费时间，同时也有助于提供状态信息，如，在屏幕上显示按绿色按钮开门。

如果让所有的表都拥有类似 DBMS 中的触发器，就可以注册输入映射的变化及其响应。钩住低级别的表，如，插入 *has_pressed_action* 表可以让用户界面知道：该更新提示信息了。

这种编码风格有点像面向方面编程，代码中很容易实现横切关注点。面向方面编程中，任何活动的核心代码都很干净，而任何副作用或禁止的活动行为，都借由其他关注点从外部钩住活动来处理。核心代码因此得以保持干净，但代价是，写代码时不知道哪些真正会调用。而使用注册机制的不同之处在于：响应从哪来？如何确定它？位于面向方面编程中，通常隐式存在的因果关系大大减少甚至移除，因而调试难度也大大降低。同时也能弱化面向对象的决策难以调整的性质，代码变得更加动态，并且免去了通常与数据驱动控制流有关的成本。

第四章　基于组件的对象

面向组件设计算是面向数据的高层设计的好开始。用组件开发，就是正确的思考第一步，避免不必要地将概念联系起来。这样构建对象，更容易按类型，而非实例去处理，也更好做性能分析。游戏开发中经常看到围绕它们建立的实体系统，实体有了数据驱动的功能集，设计人员因此可以触及程序员领域的东西。基于组件的实体不仅能更快速设计变化，且更易摆脱单体对象。大多数游戏设计师会要求更多新功能组件，而非扩展现有的。大多数新设计都需要迭代，而通过代码扩展现有组件来适应设计上的变化，游戏设计师无法来回切换，尝试不同的东西。通常情况下，添加另一个组件作为扩展或替代，会更灵活。

讨论面向组件开发时，一个问题是，有多少种不同类型的实体组件系统？为避免歧义，我们会描述一些面向组件设计不同的工作方式。

大多数人用到的第一种面向组件的方法是，复合对象。有些引擎是这样实现，其中的大多数利用其脚本语言的力量，从而以实现灵活、设计者友好的，编辑、创建组件中的对象。例如，Unity 的 `GameObject` 是基础实体类型，它会将组件添加到特定实例的组件列表中，这样就引入了组件。它们都建立在核心实体对象上，并通过它相互引用。这样，就表示每个实体仍然倾向于通过迭代根实例 (root instance) 来更新，而不是通过系统迭代。

通常讨论到创建复合对象时，常提到使用组件直接组成对象，将它们作为对象的成员。尽管优于单体类，但它仍不是完全基于组件。这种技术用组件让对象更易读，复用性，应对变化也更稳健。这些系统有足够的扩展性，足以支撑在项目间共享大型组件的生态。*Unity 资源商店*就从快速开发角度证明了组件的价值。

引入基于组件的实体时，就有机会颠覆定义对象的方式。在面向对象设计中，通常定义对象就是是命名它，然后在必要时填写细节。例如，汽车对象被定义为汽车，如果不扩展载具，那至少包括一些关于物理和网格的数

据、车轮、车壳模型资源等的构造参数，可能还会根据是否属于 AI 或玩家而改变类别。面向组件设计中，定义对象没那么死板，也不是在命名后才变成定义，而是选择或编译定义，然后在必要时标记名称。例如，用四轮物理实例化物理组件，为每部分（车轮、外壳、悬架）实例化可渲染物，添加 AI 或玩家组件来控制物理组件的输入。增加的所有这些一起，我们将其标记为汽车；或者就不管它，让它成为隐式定义，而非显式和不变的定义。

真正基于组件的对象只不过是其各部分的总和。这意味着基于组件的对象的定义，也不过是带一些构造参数的列表。于是就与对象或定义无关，重构和重新设计也更容易。Unity 的 ECS 就是这样的解决方案。在 ECS 中，实体是无形的、隐式的，组件才是头等市民 (first class citizen)。

4.1 野生组件

基于组件的开发方式经过了验证。许多著名工作室都通过使用组件驱动的实体系统，获得了巨大成功 [1]，因为他们的开发人员很清楚，对象并不是存储所有数据和特征的好地方。对一部分人来说，这是个机会，借由展示用简单的组件，组成相当复杂的实体。设计师和 Mod 作者由此能够推理出，如何才让变化符合游戏框架。还有一部分人，意味着权力移交给了性能，因为组件更容易整合为由数组构成的结构 (structure-of-arrays) 来处理。

Gas Powered Games 发表的 Dungeon Siege Architecture 可能是最早有关游戏公司使用基于组件的方法的文档了。如果有机会，请读者务必读读这篇文章 [1]，看看梦开始的地方。文章中解释道，使用组件意味着实体类型 [2] 什么能力都不需要。相反，所有属性和功能，都来自于实体的组件。

迁移到由管理器驱动，基于组件的方法有很多理由。接下来我们的尝试，至少涵盖其中几个。稍后，我们会讨论清晰的 `update()` 序列的好处；会提及组件如何让调试更容易；会谈到对象中，将意义应用于数据带来的问题、耦合，以及随着以对象为中心的实体瓦解，实例的专横如何得以缓解。

本节会展示如何将一个现有类，以基于组件的方式重写。这里要处理的是一个相当典型的复杂对象，玩家 (player) 类。通常，这种类很快会混乱、失控。本例中，假设玩家类是为常规的第三人称动作游戏设计的，并以典型的混乱作为起点。见代码 4.1。

[1] Gas Powered Games，Looking Glass Studios, Insomniac, Neversoft 都使用基于组件的对象。

[2] GPG:DG 使用 GO 或 Game-Objects，但这里我们坚持使用实体一词，因为它已经成为标准术语。

```
class Player {
public:
   Player();
   ~Player();
   Vec GetPos(); // the root node position
   void SetPos( Vec ); // for spawning
   Vec GetSpeed(); // current velocity
   float GetHealth();
   bool IsDead();
   int GetPadIndex(); // the player pad controlling me
   float GetAngle(); // the direction the player is pointing
   void SetAnimGoal( ... ); // push state to anim -tree
   void Shoot( Vec target ); // fire the player's weapon
   void TakeDamage( ... ); // take some health off , maybe animate for the damage reaction
   void Speak( ... ); // cause the player to start audio/anim
   void SetControllable( bool ); // no control in cut -scene
   void SetVisible( bool ); // hide when loading / streaming
   void SetModel( ... ); // init streaming the meshes etc
   bool IsReadyForRender();
   void Render(); // put this in the render queue
   bool IsControllable (); // player can move about?
   bool IsAiming(); // in normal move -mode , or aim -mode
   bool IsClimbing();
   bool InWater(); // if the root bone is underwater
   bool IsFalling();
   void SetBulletCount ( int ); // reload is -1
   void AddItem( ... ); // inventory items
   void UseItem( ... );
   bool HaveItem( ... );
   void AddXP( int ); // not really XP, but used to indicate when we let the player power -up
   int GetLevel(); // not really level, power -up count
   int GetNumPowerups(); // how many we've used
   float GetPlayerSpeed(); // how fast the player can go
   float GetJumpHeight();
   float GetStrength(); // for melee attacks and climb speed
   float GetDodge(); // avoiding bullets
   bool IsInBounds( Bound ); // in trigger zone?
   void SetGodMode( bool ); // cheater
private:
   Vec pos;
   Vec up, forward, right;
   Vec velocity;
   Array<ItemType> inventory;
   float health;
   int controller;
   AnimID idleAnim;
   AnimID shootAnim;
   AnimID reloadAnim;
   AnimID movementAnim;
   AnimID currentAnimGoal;
   AnimID currentAnim;
   int bulletCount;
   float shotsPerSecond;
   float timeSinceLastShot;
   SoundHandle playingSoundHandle; // null most of the time
   bool controllable;
   bool visible;
   AssetID playerModel;
   LocomotionType currentLocomotiveModel;
   int xp;
   int usedPowerups;
   int SPEED, JUMP, STRENGTH, DODGE;
   bool cheating;
```

```
};
```

Listing 4.1: 巨大的玩家类

这个例子包含许多能在游戏中找到的类型，其中代码库已经有序开发了许久。玩家类通常有很多辅助函数，方便其他实现。从保存数据到渲染至屏幕，辅助函数通常将玩家类本身视为一个实例。玩家类常几乎涉及到游戏的方方面面，因为人类玩家是代码首要目标，玩家类也会引用几乎所有东西。

若 AI 角色更泛化，而不是更专门，那它们也会有类似奇怪的类。较小机器中的游戏里，专门的 AI 就比较普遍；但现在，因为玩家类在游戏过程中要与许多 AI 互动，所以它们往往会像玩家类一样，被统一为一种类型，如果与玩家不一样，则有助于简化互动的代码。截至目前，区分 AI 的方式主要是通过数据，而行为树则是驱动 AI 思考的主舞台。行为树属于另一个有多种解释的概念，所以有些形式是面向数据设计，有些则不是。

4.2 远离层级结构

人们从面向对象的游戏类层级结构，转向基于组件的方法时，常在文章和总结中提到这样一个主题：把类变成较小对象的容器的过渡状态，通常称为组合。这种过渡形式会用一个现有类，找到它内部概念之间的界限，尝试重构为新类，原来的类会拥有或指向这些新类。从前面巨大的玩家类中，可以看到很多东西不是直接相关的，但不意味着它们没有联系。

面向对象的层级结构是一个**是什么 (is-a)** 的关系，而组件和面向组合的设计则是**有什么 (has-a)** 的关系。前者变换到后者可以当作是下放责任，或从原本绑定在”是什么”上的角色，变得更松散，但在整个树中都维持着专业。组合能清除大多数常见的钻石继承问题，因为类的能力是通过积聚不同的基类增加的，就跟以前通过 `override` 增加一样。

我们要做的第一件事是，把单体类的有关系的部分移到各自的类中。按照组合的思路，把类从拥有数据以及能修改它们的操作的状态，变成包含数据的实例，并尽量把操作下放到这些专门的结构中。把数据移出到独立的结构中，这样以后就更容易组合成新类。一开始只要按理解中的系统边界去分类。如，将渲染与控制器输入、游戏细节（如背包）等分离，动画也独立出来。

现在看一下拆分玩家类的结果，如代码 4.2 中，可以做点初步评估。能看到，通过从较小类构建一个大类，第一遍可以帮助将数据组织成不同的、

面向目的的集合，也可以看到类最终变得纠结混乱的原因。当考虑到每部分的需求，他们需要什么样的数据，耦合就越来越明显。渲染函数要访问玩家的位置和模型，而游戏功能，如 `Shoot(Vec target)` 需要访问背包、设置动画、造成伤害。受到伤害需要访问动画和生命。现在看起来已经比预期的更难处理了，但其本质，却已经很清楚：代码需要跨越不同的数据块。仅是这第一遍，就可以看出，功能和数据不适合在一起。

```
struct PlayerPhysical {
   Vec pos;
   Vec up, forward, right;
   Vec velocity;
};
struct PlayerGameplay {
   float health;
   int xp;
   int usedPowerups;
   int SPEED, JUMP, STRENGTH, DODGE;
   bool cheating;
   float shotsPerSecond;
   float timeSinceLastShot;
};
struct EntityAnim {
   AnimID idleAnim;
   AnimID shootAnim;
   AnimID reloadAnim;
   AnimID movementAnim;
   AnimID currentAnimGoal;
   AnimID currentAnim;
   SoundHandle playingSoundHandle; // null most of the time
};
struct PlayerControl {
   int controller;
   bool controllable;
};
struct EntityRender {
   bool visible;
   AssetID playerModel;
};
struct EntityInWorld {
   LocomotionType currentLocomotiveModel;
};
struct Inventory {
   Array<ItemType> inventory;
   int bulletCount;
};
class Player {
public:
   Player ();
   ~Player ();
   // ...
   // ... the member functions
   // ...
private:
   PlayerPhysical phsyical;
   PlayerGameplay gameplay;
   EntityAnim anim;
   PlayerControl control;
   EntityRender render;
```

```
    EntityInWorld inWorld;
    Inventory inventory;
};
```

Listing 4.2: 组合的玩家类

第一步，我们把玩家类变成组件的容器。目前，玩家拥有组件，而玩家必须在玩家类实例化之后才会存在。为了尽可能干净地分离为例为组件，并尽力保证复用性，就不能依赖由其实体处理或更新，可以试试交由管理器。这样做的话，在迭代多个相关任务的实体时，从拥有者那里移除实体，仍能获得缓存局部性 (cache locality) 的优势。

现在就变得有些哲学了。每个系统都有它需要的数据，才能正常运作，但即便它们会有所重叠，也不会全面共享数据。考虑一下，序列化系统需要了解字符的什么信息。它不可能关心动画系统的当前状态，但它会关心背包库存。渲染系统会关心位置和动画，但不关心当前的弹药量。UI 渲染代码甚至不关心玩家的位置，但关心背包库存以及玩家的生命值和伤害。为什么把所有的数据放在一个类中，不是长久之计？问题的核心，便是这种利益上的差异。

一个类或对象的功能，来自于如何解释内部状态，以及如何解释随时间变化的状态。事实间的关系属于问题域，称之为意义，但事实只是原始数据。这种事实与意义的分离，在面向对象方法中是不可能的。这就是为什么每次事实获得一个新意义时，该意义必须作为包含该事实的类的一部分来实现。将类拆解，提取事实作为独立的组件；才得以摆脱那种，为了给类灌输永久意义，时常不得不通过间接方法去查找事实。与其按意义存储可能相关的数据，我们只在必要时赋予它。只有当意义是解决直接问题的一部分时，才需要赋予。

4.3 面向管理器

把类拆成组件后，我们的类现在看起来好像更笨拙了，它们在访问隐藏在新结构中的变量。但不是类应该去寻找变量，而是像渲染这样的普通操作，需要位置和模型信息，也需要访问渲染器。游戏开发中，这种对象越界访问通常是种妥协，在这里，则是从以类为中心的方法，转向面向数据的方法。我们的目标是，将数据转化为影响图形管道的渲染请求，且不触及渲染器与无关数据。

请看代码 4.3，这里不再有玩家的更新，而是对组成玩家的每个组件执行更新。这样一来，每个实体的物理都会在渲染前，或在另一个线程渲染时更新。所有对实体的控制（无论是玩家或 AI）都可以在执行动画前更新。管理器控制代码的执行时间是实现代码完全并行化的重要部分。这时可以更有信心地提升性能，而不必担心会对其他领域产生负面影响。分析哪些组件需要逐帧更新，哪些可以不那么频繁，优化由此产生，组件之间也相互解锁。

```
class Renderable {
   void RenderUpdate () {
      auto pos = gPositionArray [index ];
      gRenderer.AddModel( playerModel , pos );
   }
};
class RenderManager {
   void Update () {
      gRenderer.BeginFrame ();
      for( auto &renderable : renderArray ) {
         renderable.RenderUpdate ();
      }
      gRenderer.SubmitFrame ();
   }
};
class PhysicsManager {
   void Update () {
      for( auto & physicsRequest : physicsRequestArray ) {
         physicalArray [ physicsRequest.index].UpdateValues(
         physicsRequest.updateData );
      }
      // Run physics simulation
      for( auto &physical : physicalArray ) {
         positionArray[physical.index].pos = physical.pos;
      }
   }
};
class Controller {
   void Update () {
      Pad pad = GetPad( controller );
      if( pad.IsPressed( SHOOT ) ) {
         if( inventoryArray [index ]. bulletCount > 0 ) {
            animRequest.Add( SHOOT_ONCE );
         }
      }
   }
};
class PlayerInventory {
   void Update () {
      if( inv. bulletCount == 0 ) {
         if( animArray.contains( inv.index ) {
            anim = animArray[ index ];
            anim. currentAnim = RELOAD;
            inventoryArray [index ]. bulletCount = 6;
            anim. playingSoundHandle = PlaySound( GUNFIRE );
         }
      }
   }
};
class PlayerControl {
   void Update () {
```

```
        for( auto &control : controlArray ) {
            control.Update();
        }
        for( auto &inv : inventoryArray ) {
            inv.Update();
        }
    }
};
```

Listing 4.3: 由管理器 tick 的组件

在许多脚本语言的组件系统中，可以定义其组件或实体的行为。面向对象程序设计存在的低效同样会影响性能。要注意，调用 `Tick` 或 `Update` 函数的依赖反转 (dependency inversion) 的做法，通常必须以某种形式沙盒处理，这就会导致错误检查和其他安全措施包装内部调用。有一个很好的例子，旧版本的 Unity 中，他们基于组件的方法允许每个实例有自己的脚本，每帧会被 Unity 核心调用。主要的开销看似是进出脚本语言，反复跨越于核心 C++ 和描述组件行为的脚本间。Valentin Simonov 在他的文章 *10,000 Update() calls*[16] 中提供的信息，证明了迁移到管理器有重大意义。文章中详细介绍了在利用依赖反转来驱动一般代码更新策略时，什么开销最大。主要的成本是在不同代码区域间移动，但即便不需要跨越语言，管理器也有其意义：它能确保组件同步更新。

如果我们让玩家之外的其他类使用这些数组呢？通常会有独立逻辑处理玩家射击。假设将武器类重构为通用武器后，如通过可以被玩家或 NPC 指向的新武器类，NPC 也可以使用相同的武器代码了。但是，还有个方法可以分离武器射击的代码来实现共享，而非创造一个新类去包含射击行为。实际上，这里要做的，就是将射击拆分成它实际包含的不同的任务 (task)。

任务有利于并行处理。有了基于组件的对象，就能把以前面向类的大部分处理，变成更通用的任务转移出去，交给任何能胜任的 CPU 或协处理器。

4.4 没有什么实体

如果彻底删除玩家类的话会怎么样？如果一个实体可以由它的组件集合来表示，那除了这些同质的组件，它还需要任额外的身份吗？如同表格各行的数值，组件一起描述了单一的实例；同样像表格中的各行一样，表格也是一个集合。在组件组合的多元宇宙中，构成实体的组件不是有关实体的信息，而是实体本身，即实体需要的唯一身份。既然实体就是其当前配置的组件，那就可以彻底删除核心玩家类。这就意味着我们不再认为玩家是游戏的

中心。同时由于这个类不复存在，代码也不再与特定的单一实体有联系。代码 4.4 粗略展示了如何开发这种方案。

```
struct Orientation { Vec pos, up, forward, right; };
SparseArray <Orientation> orientationArray;
SparseArray <Vec> velocityArray;
SparseArray <float> healthArray;
SparseArray <int> xpArray, usedPowerupsArray, controllerID, bulletCount;
struct Attributes { int SPEED, JUMP, STRENGTH, DODGE; };
SparseArray <Attributes> attributeArray;
SparseArray <bool> godmodeArray, controllable, isVisible;
SparseArray <AnimID> currentAnim, animGoal;
SparseArray <SoundHandle> playingSound;
SparseArray <AssetID> modelArray;
SparseArray <LocomotionType> locoModelArray;
SparseArray <Array<ItemType>> inventoryArray;
int NewPlayer(int padID , Vec startPoint) {
   int ID = newID ();
   controllerID[ID] padID;
   GetAsset( "PlayerModel", ID ); // adds a request to put the player model into modelArray[ID]
   orientationArray [ ID ] = Orientation(startPoint);
   velocityArray [ ID ] = VecZero();
   return ID;
}
```

Listing 4.4: 组件的稀疏数组

摆脱编译期定义的类，就可以创造更多其他的类，且不用增加太多代码。脚本能够通过组合或原型生成新的实体类，力量大大增强；干净利落地使游戏呈现的内容更复杂，但实际复杂度却并没有增加太多。终于，游戏中所有不同的实体，都能同时运行相同的代码了，处理也集中并简化了，于是有更多的机会共享优化，隐藏的 bug 也会减少。

第五章　层级细节水平和隐式状态

游戏主机和显卡通常不会在管线的多边形渲染阶段遇到瓶颈。通常会是带宽有限。如果有大量 alpha 混合，还可能是填充率问题。多数情况下，图形芯片会在读取纹理上花大量时间，纹理带宽往往就成为瓶颈。正因如此，用多个多边形数量递减的网格来做 LoD[1] 的老办法，怎么也比不上考虑到每个渲染对象 LoD 的实际数据的新技术。渲染时，驱动端的处理能涵盖绝大部分停顿，或者说是，相对实际渲染的内容，处理得太多。HLoD [2] 可以解决图元数量过大，远超必要的驱动调用问题。

有一个基本优化技巧，通过将许多低 LoD 的网格，组织、合并成一个低 LoD 网格。这样就能减少设置渲染调用的时间，比较适用于驱动调用成本较高的情况。典型的超大规模的环境中，使用 HLoD 能够将游戏引擎的工作量降一个量级，场景中需要处理、渲染的实体数量显著下降了。

即便对比之下，渲染的多边形数量可能完全相同，甚至更高。实际引擎也只同时处理了数量相当的实体，稳定性增加了，也可以对美术和代码层面做更精确的针对性优化了。

5.1　从零到无限

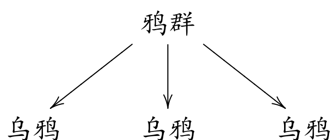
既然实体可以根据其属性隐式存在，我们倒可以利用 HLoD 技术做些代码优化。传统 LoD 中，相机离物体（或实体）越远，细节和准确度就越差。比如减少多边形数量、纹理大小，甚至是驱动蒙皮网格的骨架数量。游戏逻辑也会退化。远离实体时，它可能会切到更粗粒度的时间步长。AI 行为从 $50Hz$ 的更新频率变到 $1Hz$ 也不是没有过。在 HLoD 实现中，实体越近，又或是对玩家越明显，才是它开始存在的时候。

[1] 译注：level of detail, 细节水平

[2] 译注：hierarchical level of detail, 层级细节水平

假设有个射击游戏，玩家正在抵御敌袭，保卫基地。玩家守在一座防空炮塔里，敌方声势浩大，包含上万架敌机联队涌来，每个中队里有上百架。玩家必须击落所有敌机，否则就会被炮火淹没，整个基地也夷为平地。

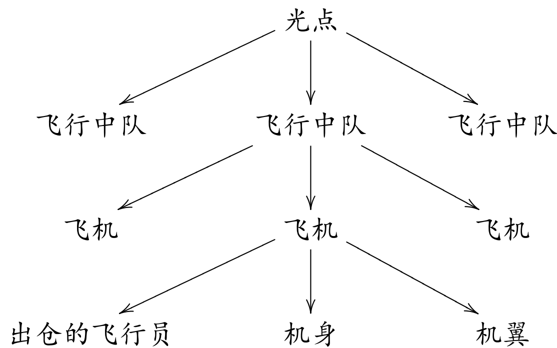
如果每次每架敌机上都执行完整 AI，上万架敌机成群移动，躲避慢吞吞的火炮，开销大概会很大。但实际上不需要这样。大多数 AI 程序员都会做一个基本假设：AI 只在其攻击范围内才会执行。的确如此，而且与相比笨办法，直截了当就提升了速度。HLoD 则打开了思路，根据玩家对实体的感知方式，改变实体的数量。用更合适的术语，**总体 LoD** [3]就不错，它能更好地描述背后发生的事情。就算有时没有层级，也仍旧可以变动不同 LoD 之间元素的引用规则。总体 LoD 一词的灵感来自于总体这一术语。一个鸦群[4]为一个计算单元，每只乌鸦都是总体中低 LoD 的子元素。



这个游戏的总体 LoD 版本中，雷达上几支飞行中队投射出的光点。但他们接近前不会有自己的实体。一旦有中队进入射程，代表当前中队地光点被移除，并生成新的中队实体。新实体会在雷达上显示出每架飞机的光点。这些飞机还不存在，而是隐含在中队中，就如同中队隐含在更大的联队中一样。只要有中队进入射程，就移除它，直到联队内部计数降为零，就可以删掉自己，因为它已经不代表任何实体了。如果有中队进入更近的范围，它就会把其中的飞机移除，创建独立的飞机实体，最终删除自身。随着飞机越来越近，传统的 LoD 技术开始发挥作用，可渲染对象也能切到更高分辨率，AI 也能以更高规格运行。

[3]译注：*collective lodding*，为本书新引入的术语，此处为直译。

[4]译注：a murder of crows, 原文中一群乌鸦的量词为 murder，为英语中固定搭配。参见 [5]



飞机被击中时，会切到受损的类型。未受损前，是满血敌机。如果 AI 对伤害的反应是惧怕，可能就会弹射驾驶员，在世界里增加一个实体。如果机翼被打掉，也会变成一个新实体。一旦坠毁，就可以删除其实体，用硝烟残骸的实体替代，无论是不是模拟的，都要比完整的空气动力学模拟容易处理得多。

如果事态失控，玩家没能阻止进攻，飞机数量剧增，以至于常规的 LoD 系统都无法减轻其影响。此时总体 LoD 仍能带来一点帮助：将飞机收回中队，用集群取代单体进攻。桌游《战锤幻想战役》中，经常有很多小队互相射箭，玩家也常会以小队为进攻单位，不会真的为每个独立的士兵、老鼠、兽人等掷骰子。实际上会根据有多少小队，然后掷出等量骰子，看看有多少攻击通过判定。这就是作为中队攻击的意思。飞机不再攻击，而是计算攻击的成功概率，掷下骰子，攻击命中。LoD 的启发规则 (heuristic) 可以调整，所以最近、最靠前的中队一定是最高的 LoD，他们可以有效地独立掷骰子，而在后方的中队则保持非常简单的表示。

这就是游戏开发的魔术[5]，是基本的游戏引擎元素。过去，开发者减少了同时攻击的 AI 数量 [6]；通过交错排列赛道，减少同屏汽车数量 [7]；与其说许多人同屏，但其实所有人都合并成了一个 [8]。这种减少处理的方法很常见。接下来要考虑把它用在所有合适的地方，不不单单是玩家看不到的时候。

[5]译注：原文为 smoke and mirrors，是魔术中常用的障眼法。

[6]没错的话，应该是《半条命》。

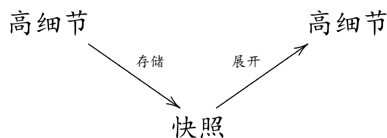
[7]《山脊赛车》以此著称。

[8]《上帝也疯狂》就是这么实现的。

5.2 快照

减少细节，带来一个老问题。改变游戏逻辑、AI 等方面的 LoD，会丢失掉高细节的历史状态。这样，就需要能够来存储所需，保证玩家有紧密流畅的体验。如果玩家面前的一个高细节中队离开了视线，另一个中队进入，那在前一个中队回到视野时，应当仍旧保留之前的损伤。试想，如果之前把飞机所有玻璃都打碎了，但再次看到时，玻璃却又完好如新。虽然只是外观，但显得特别扎眼，让人出戏。

高细节的实体降到低 LoD 时，应存储一个快照[9]，一块小且压缩良好，包含所有必要信息的数据块，能够从低细节实体重建高细节实体。中队离开视野时，存储压缩好的快照，包含损伤的数量、位置，飞机的大致位置等信息。当中队回到视野，读取这些数据，将高细节实体恢复到之前的状态。大多数情况下，有损压缩也没什么问题。哪些窗户？怎么碎的？都不重要，或许只用知道它坏了 2/3 就够了。



另一个例子，城市漫游类游戏。如果 AI 可以进出车辆，就可以在 AI 进入车辆时从世界中移除它，减少处理时间。如果只是作为乘客，只需保存少量信息就足以重建。如果是司机，可能就需要在他下车前，基于行人的属性创建司机类型，然后再创建快照 [10]。

如果车辆远离玩家到一定距离，就可以删除。为了性能，可以改变有快照的车辆的优先级，使其尽量离开玩家视野，这样就能提前删除他们。这类优化在面向对象系统中会很难协调，因为其不鼓励做类型的内部检查。有些游戏，会通过从设计层面重置快照数据，并将其作为游戏元素的一部分，解决掉这个问题。《塞尔达：荒野之息》会在血月期间重置怪物。因此，当玩家回到营地，即便发现所有怪物都跟没打过一样，也不会诧异了。

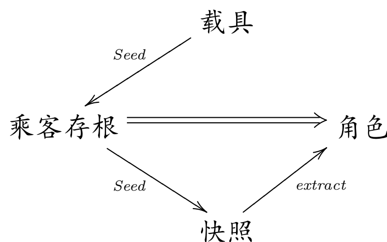
[9] 译注：mementos, 此处取设计模式中的译名之一：快照模式。

[10] 译注：与乘客相反，司机会在下车时创建快照，返回车辆时恢复其原本类型，驾驶席才是他的家。

5.3 JIT 快照

如果有辆车，本来只是作为环境元素放进场景，然后忽然得承担更重要的角色，比如卷入一场枪战。那它就得有更多细节，并且得真实可信。因此考虑到玩家对游戏的了解，需要特别注意，生成的新实体不能过于普通或出戏。生成数据的行为可以当作是记录一张快照，以便即时读取。即时快照[11]能够创建假快照，利用伪随机发生器 (或哈希函数)，按需创建合适的信息来保证一致性。并且它不需要依赖任何存储的数据，只依赖需求方实体的隐含信息。

除了用全局随机发生器生成新角色，还可以用生成目标的细节作为种子。比如，玩家正靠近一辆车，现在要生成司机和乘客，渲染车内的人。通过查表创建随机角色固然可以。但如果车辆远离了渲染范围，再折返，车内的人看起来可能就不同了，因为他们必须重新生成。但如果使用其他独特属性 (如车牌) 做种子生成司机与乘客，既不影响生成的快照，也无需内存开销，同时也不用担心对象的生命周期。总是能从无到有，重新生成。



这种技术常用于地形景观生成器，其中，地形通常用坐标 (x, y) 做种子。那用来生成游戏中第 107 天的天气是否可行呢？生成 Perlin 噪声时，许多算法都会基于某个噪声函数。为了确保地形能够复现，噪声函数必须可重复，所以它需要每次都能产生同样的结果。如果要生成地形，噪声函数最好也具备连贯性。即，输入函数的微小差异，对应输出中的微小变化。不过生成 JIT 快照时，不需要这种特性：微小的输入差异都能让输出大相径庭的哈希函数足矣。

举例来说，在给定地形上生成一栋建筑。先用任意常规的随机生成器，用建筑物坐标作种子。根据建筑所在及其约束，从建筑模板中选择、生成随机的建筑，如同通过从磁盘加载配置文件生成一样。建筑多大？随机选择大、中、小。根据大小，有多少房间？小的 2、3 个，大的 `(int)(7+rand*10)`

[11]译注：Just In Time Memetos, 或称 JIT 快照

个。重点是，只要随即发生器接受输入的种子，每次相同的过程都有相同的结果。每次 223.17, −100.5 的建筑，或许都是一样的四面墙，一样的油漆，破损的窗户，后花园里还有完美的田园小青蛙池塘。

基于 JIT 快照，生成有质感的环境。通过风格表（或风格指南）指导在虚拟空间中生成快照的感觉偏向。假设，有一份城市的风格指南有规定汽车乘员。其中描述道，商人可以与人共乘，但倾向于独乘；家庭会有孩子在后座，由年长的成人驾驶；年轻人通常结伴驾车出行。风格指南能帮助生成的数据变得更可信。加入局部变化，如，将汽车类型与司机类型联系起来。敞篷车由穿着考究的人 (或年轻人) 驾驶；低趴 [12] 在刻板印象里几乎和他们车主形象绑定；进口车、改装车则由年轻人驾驶。就像太空游戏中，货船里蓬头垢面的船员，指挥豪华游艇的舰长，无畏舰里粗野或精干的佣兵。于是，有了氛围，进一步增加点惊喜，整个游戏便栩栩如生。

JIT 快照还能用来保持多样性。依赖风格指南，每个人都没能给玩家留下印象，所以每个人都一样。显现这些偏向，又不严格遵守，就能建立更富有质感的环境。如果环境中一直有大量完全不同的人，就没什么好坚持的，无法识别出什么模式。大脑在没有模式时，倾向于看到噪声，千篇一律。即便最多样的虚拟世界，如果有太多的内容扎堆在一起，也会显得很平淡。沿着街道走，是否能发现有完全一样的地砖？或许可以，但也能看到的些许损坏、腐烂、灰尘、出错、瑕疵。为了让环境真实可信，必须让它看起来像是有人花了大力气，力图让全部细节符合预期。

5.4 替代维度

同所有事物一样，去掉一些假设，就能发现工具的其他用途。熟悉 LoD 系统的话，就不难发现，其显式的约束条件，通常是一些空间中的距离函数。是时候抛弃这个假设，并分析其背后时如何运作的了。

首先，去掉距离的假设，条件就可以推断为某种线性度量。这个值通常由函数生成，该函数能获取目标物体与相机的距离。舍弃距离假设，还要对接下来要做的事情有个更基本的认识。现在，我们正在用单一的运行时变量，控制游戏中实体的呈现。虽然现在已经有很多游戏状态由运行时变量控制，但当前，监控变量 (或轴) 反映出的表现是被动的。这种表现通常是一些图形、逻辑的 LoD，但也可能是对实体很重要的东西，例如它本身是否存

[12]译注：悬挂刻意调到很低的一种改装车风格，https://en.wikipedia.org/wiki/Lowrider

在。

5.4.1 真正的指标

虽然通常会用距离来确定某个东西的 LoD。但他其实并不是真正的指标，只算是密切相关。并且其实是反过来的。真正衡量 LoD 的标准，应该是实体在我们的感知中的占比。一个大而远的实体占据的感知，应当和小而近的一样多。我们一直讨论的 HLoD，用房间里的大象描述这种情况再贴切不过。雷达上有远方敌机投射的光点。它们占用的感知与一个中队相当，而中队在射击范围内占用的感知空间和一架敌机相当。

理解这个概念：LoD 应该由玩家在它所处的范围内如何感知事物来定义。如果能够内化这一点，就能选对 LoD 间的界限。

5.4.2 超越空间

再来考虑，还可以计算哪些变量？哪些变量，能够去除冗余的细节？任何能省去不必要处理的机会都不能放过。如果游戏中某些元素不是玩家当前关心的，或者很容易忘掉的，就可以让其消解。把玩家关心事物的概率当作尺度，玩家的记忆和注意力就能被量化，就可以用来决定最终表现。

已知有个玩家关注的实体，但是实际看不到，却又玩家的感知中占很大比重。这种利害关系，使其在细节方面有着更高优先级，远高于本来应有的水平。例如，玩家在暗杀游戏中追踪的人物，或许在任务开始时只被发现了一次。但其在整个任务中必须保持高度一致，因为它是玩家最关心的对象，仅次于生存等原始需求。即使角色躲进人群，很久之后才被发现，也必须和玩家上次看到的样子一致。

试想，玩家多久会忘记相对重要的事情？这些信息有助于尽可能减少内存占用。如果读者玩过《侠盗猎车手 4》，可能有注意到：只要不看车，车就会消失。尝试转身几次之后，也许会发现，每次看向车的时候，它们似乎都不一样。这就是 TLoD [13] 的绝佳案例。玩家撞过、开过、停过的汽车仍停在原地，本质上，是玩家把它们放在那里。玩家与它们互动过，因而玩家很可能记得它们的位置。然而周围的车辆，无论警车还是民用车，都没那么重要。而且通常没有任何特殊状态，所以在玩家移开视线时就会消失。

另一种极端，有些游戏会记住玩家所做的一切。玩家在游戏的前几分钟

[13] 译注：temporal level of detail.

杀敌，掠夺，物品散落一地；一百个小时后回来，物品仍在离开时的地方。这种的游戏存储了大量微小的细节，它们需要仔细存储，否则会引发持续的让人崩溃的性能问题。其中一个方法是使用空间映射的快照，能将这种级别的，对玩家与游戏互动的关注度合理化。

除了前面提到的时间，有些元素的细节则由其他变量决定：如玩家的进度，拥有某物的数量，或某些行为的次数。比方说，典型的交易动画可能会被剪得越来越短，因为游戏会以玩家*最近交易的次数*为轴，缩减由该事件导致的非交互部分的时长。这类功能很容易实现，且从玩家那边收效巨大。例如只在一定量的单项交易后才可以开启多项交易。实际上，可以通过这些抽象 LoD 风格的维度去设置游戏元素，响应状态，触发教程，提示玩家，扩展游戏选项等。还能以玩家的熟练度为轴，去处理游玩法深度和复杂度的LoD。

这样操作游戏当前状态更安全，能够避免过渡错误。从一个状态到另一状态过渡时，某些属性可能已经设置为 `true`，但反向过渡时，没能将其设置回 `false`，就可能引发这类错误。但其实可以将状态视作维度上的隐含信息。状态，很容易在错误的时间，错误的方式被修改。如果状态与其他变量相关联，即，状态是其他状态的函数，就更难出现不一致问题了。

举个例子，菜单系统中，所有变换都是可逆的。有时我们会发现向下两级菜单，再返回一级，会返回到最开始的界面。比方说，选项-选中音量滑块-返回，此时选项菜单完全关闭了。这类错误并不少见，UI 系统中有大量不同的交互层。对比玩法，UI 中获取输入的方式更为隐晦。菜单的常见问题之一，就是关于某帧输入的所有权。例如，如果玩家同时按下前进和后退，状态机实现的 UI 可能会进入先捕获的按键响应。另一种情况，先收到进入事件，然后在下一级菜单收到返回事件。还有概率极小但更差的情况，但也不是没发生过：菜单同时过渡到两个不同菜单。有时，菜单还可能会因为外部力量进入过渡动画，如在不同的执行线程中捕获到玩家输入，游戏状态就会不连续，没有响应。比如在网游大厅，所有玩家都准备好了，但在有人比赛开始前打开了选项界面，忽然房主连接断开。如果以传统状态机处理菜单，一旦退出选项界面，玩家应该回到哪里？通常，大厅会让玩家返回到服务器搜索界面，但现在大厅已经没了，取而代之的是一片虚无。这便是使用简单维度替代状态机的优势。更简单，进而错误更少，响应更快。

5.4.3 总体 LoD (如何减少实例数量)

这个术语不太好，希望哪天有人能想到更好的。但恐怕直到没人用了，它也都不需要命名。本书写作期间，已经有很多应用它的案例了。这里指的不是那类有成百上千飞机交战游戏，交战双方发射漫天的导弹，带来大量的 GPU 粒子特效。这里要讨论的游戏，实际看起来很简单。比方说有个园艺模拟器，不知为何，植物的每片叶子都建模为单个实例；四处授粉的每一只昆虫也都是单个实例；植物扎根的每块土壤也都是单个实例；埋下的每颗种子也都是单个实例。每个实例都有自己的生命周期、组件、动画、内部状态，整个系统的复杂度不断增加。

假设有个虚构的种田游戏，可以在里面种小麦。游戏中有片由 100×100 的地块组成的麦田。有些游戏中，这些种植小麦的地块是实例，上面生长的小麦也是实例。这样做其实没什么道理，可以把田地的数据量缩减到非常小。实际需要了解田地和小麦的哪些信息？需要知道小麦的位置吗？并不，它就在平铺的网格中。需要知道地块上是否有小麦吗？是的，但不需要用对象实例表示。需要用对象渲染小麦吗？小麦会随风舞动，所以在它需要被吹动时，跟踪位置并维持动量吗？也不用，几乎所有情况，都能通过作弊，以低成本实现足够的可信度。没有每片草的实例，也能正常渲染草地。一片麦田正确的数据格式，可以简化到 10,000 个 `unsigned char`，0 表示没有小麦，[1, 100] 表示小麦的生长情况。小麦没有坐标，坐标上方有小麦。

如果在游戏《我的世界》里有一组块，但背包的实例不足 64，那就只有一个类型，一个倍数。即是一组（或一叠）。如果在餐厅模拟游戏里有一摞盘子，且实例数目小于 10，就可以换成一摞盘子的对象，用一个 `int` 表示当前盘子数目。

这方面的基本原则，是确保这个世界中有“插槽”。无论是手动放置，还是用模式生成，持续跟踪其中的内容，取代掉直接将物品放置在世界这种方式。学习用路人称呼事物的方式。如果问起路人房间里会有什么时，大概不会说是沙发、书架、两个扶手椅、茶几、电视柜、书架等等。不，他大概会说，家具。跳出框框，从外部审视游戏。用玩家的方式描述屏幕中的内容。看他们如何描述背包。从玩家的视角，了解玩家的心理模型，并匹配它，就会找到占用玩家感知空间的内容及其关联。

标准化数据时，可以注意观察，行如何与容器对齐。若有任何形式的网格，从一维到四维，都值得去研究和利用。也不要忽视其他网格，如三角网格、六边形网格。尤其是六边形网格，名字绕口，但它可以通过不同的遍历

函数，表示为一个正方形网格。也不要因网格在字面上不规则就绕开，一些基于网格的游戏中，单元格的中心会被扰动，以使其看起来更自然。但游戏代码仍可以严格基于网格，从而将问题放进更好的解决域中，也更容易让玩家推导出能做什么，不能做什么。

第六章　查找

知道为什么需要查找特定的数据，非常重要。如果没必要，可能就直接省下最大的一环。天真地去查找表中某一行是否存在，会很慢。可以手动添加一些查找助手，如二叉树、哈希表，或者只是在插入表时选择有序插入。如果想做后者，也有可能让其变慢：有序插入通常不是并发的；而且添加额外的辅助工具，通常是一项手动任务。本章中，我们会尝试解决所有这些问题。

6.1　索引

长期以来，数据库管理系统种一直都有索引的概念。当 DBMS 注意到特定查询执行了很多次，就会自动添加索引。可以根据这个思路，在游戏中实现即时 (just-in-time) 索引系统，引入同类的性能优化。

SQL 中，每次要确认一个元素是否存在，或是生成一个子集 (如找到某范围内的所有实体), 就得创建一次查询操作。查询作为实体存在，能帮助建立在 DBMS 中的直觉。

可以把用于创建行、生成表的查询，当作是对象。它能挂起，以备不时之需，并且可以根据其在一段时间内的使用情况，适时变换。开始时只是简单的线性查询请求 (若数据还未被排序)。进程经由内部监测，可以发现它是否经常使用，并且能发现它是是否通常返回简单的可调整的结果集 (如有序链表的前 N 项)。超过某些预定义阈值（如：操作次数、生命周期等）之后，如果能让查询对象将自己与其所引用的表挂钩，就会非常有价值。通过钩住插入、修改、删除等操作，查询本身可以更新结果，就不用在每次要访问结果时，重新完整运行。

这种智能对象，是面向数据能从 OOP 那边借鉴的。某些情况下，它能大大节省开支，并且因为是可选的，也可以很安全。

如果建立通用的后端，处理对这些表的查询，能带来多种好处。不仅能预期不再使用的索引被垃圾会收，还能让程序在某种程度上自我记录、归档。如果再研究一下日志，发现有哪些表要建立查询，就可以看到数据热点，找到改进的空间。甚至说，让代码自我记录并决定采取哪些优化步骤，也是可能的。

6.2 面向数据查询

面向数据查找的第一步，是理解查找条件，并理解其依赖的数据间的不同。面向对象查找的方案，经常问询对象是否满足某些条件。对象被问到时，可能就需要大量代码，间接内存访问，缓存行被填满，但几乎未能充分利用。即使在面向对象代码库之外，仍然有很多无法充分利用内存带宽的情况。代码 6.1 是一个简单的二分查找的例子：在简单的动画容器实现中，查找一个键。这类数据访问模式常见于动画库中，而且在许多人工编写的结构中也很常见，它们的条目都沿着某个维度排序。

```
struct FullAnimKey {
    float time;
    Vec3 translation;
    Vec3 scale;
    Vec4 rotation; // sijk quaternion
};
struct FullAnim {
        int numKeys;
        FullAnimKey *keys;
        FullAnimKey GetKeyAtTimeBinary ( float t ) {
            int l = 0, h = numKeys -1;
            int m = (l+h) / 2;
            while( l < h ) {
                if( t < keys[m]. time ) {
                    h = m -1;
                } else {
                    l = m;
                }
                m = (l+h+1) / 2;
            }
            return keys[m];
        }
};
```

Listing 6.1: 基于对象的二分查找

了解进程中的生产者和消费者的依赖关系，我们就能快速改进它。代码 6.2 即是快速重写后的例子，通过移出到部分数组结构，节约了大量内存请求。数据布局，源自于认识到满足程序所需的数据。

首先，要考虑什么能做输入，然后需要提供什么做输出。当前唯一的输

入，是一个浮点数形式的时间值，而这个例子中要返回的唯一数值，是个动画键。要返回的动画键取决于系统内部的数据，而且我们也可以任意重排数据。已知，输入将与键中的 `time` 做比较，且与键中的其他数据无关。因而可以将其提取到单独的数组中。找到要返回的动画键时，并不是只要访问其中一部分，而是要返回整个键。鉴于此，将动画键数据保持为结构数组是有其意义的，这样在返回最终值时，访问的缓存行就会减少。

```
struct DataOnlyAnimKey {
   Vec3 translation;
   Vec3 scale;
   Vec4 rotation; // sijk quaternion
};
struct DataOnlyAnim {
   int numKeys;
   float *keyTime;
   DataOnlyAnimKey *keys;
   DataOnlyAnimKey GetKeyAtTimeBinary( float t ) {
      int l = 0, h = numKeys -1;
      int m = (l+h) / 2;
      while( l < h ) {
         if( t < keyTime[m] ) {
            h = m -1;
         } else {
            l = m;
         }
         m = (l+h+1) / 2;
      }
      return keys[m];
   }
};
```

Listing 6.2: 基于值的二分查找

这样在大多数硬件上都更快，但是为什么？大多数人的第一反应是，键被从返回的数据附近移开了，并确保在返回前只取一次数据。有时候，比起第一眼，最好想得更远一点。来看一下 `AnimKeys` 的数据布局。

t	tx	ty	tz	sx	sy	sz	rs	cacheline
ri	rj	rk	t	tx	ty	tz	sx	
sy	sz	rs	ri	rj	rk	t	tx	cacheline
ty	tz	sx	sy	sz	rs	ri	rj	
rk	t	tx	ty	tz	sx	sy	sz	cacheline
rs	ri	rj	rk	t	.	.	.	

主要是因为这里的处理都是根据 `time` 的值来定位键的索引。在提取 `time` 的代码中，不再通过结构数组中的某个成员来寻找整个结构。在查找阶段，缓存会被大量相关的数据填满，因此这样会更快。原来的布局中，每

条缓存行只有一到两个 key.time。更新后，则有 16 次之多。

t0	t1	t2	t3	t4	t5	t6	t7	cacheline
t8	t9	t10	t11	t12	t13	t14	t15	

有些方法能更好地组织数据，但任何额外优化，都得在复杂度、空间、时间上做取舍。基本的二分查找能很快找到结果，但每一步都会读入新的缓存行。如果知道目标硬件的缓存行大小，就能在加载下一段缓存行之前，检查所有已加载的数值。达成这样的结果，大部分需求的数据都在缓存里，此时唯一要做的就是确保找到正确的结果。在会关注缓存行的引擎里，游戏代码可以调用这些优化过的查找算法，悄然完成工作。值得一提，每次进入更大的数据结构时，都会错失复用成熟代码的机会。

二分查找是最好的查找算法之一，它能用最少的指令寻找关键值。但如果想找到最快的算法，就必须看看究竟什么在花费时间，而通常不是指令。加载一整行缓存信息，并尽可能多地利用它们，会比使用最少指令数更有帮助。试想，一个算法分别用两种不同的数据布局，会不会比算法本身的影响都来得要大？

times		keys		n	s0	s1	s2	cacheline
s3	s4	s5	s6	s7	s8	s9	s10	

为了对比前面查找动画键的代码，再来看第三种方案：尝试利用结构中剩余的缓存行空间。原有的结构体包含 numKeys，以及 keyTime 和 keys 两个指针，缓存行还有很多剩余。PS3 和 Xbox360 最大开销之一就是缓存行利用率[1]低。现代 CPU 中，情况没那么糟，部分原因是缓存行比较小。但每次请求时，可以免费读取些什么，仍旧值得考虑。本例中，缓存行足以存储额外 11 个浮点值，用来存储类似于跳跃链表 (skip-list) 的内容。

```
struct ClumpedAnim {
   float *keyTime;
   DataOnlyAnimKey *keys;
   int numKeys;
   static const int numPrefetchedKeyTimes = (64- sizeof(int)-sizeof(float*)-sizeof(DataOnlyAnimKey *))/sizeof(float);
   static const int keysPerLump = 64 / sizeof(float);
   float firstStage[ numPrefetchedKeyTimes ];
   DataOnlyAnimKey GetKeyAtTimeLinear(float t){
      for(int start = 0; start < numPrefetchedKeyTimes; ++ start ) {
         if( firstStage[start] > t ) {
            int l = start* keysPerLump;
            int h = l + keysPerLump;
```

[1]译注：cache line utilization, CLU

```
            h = h > numKeys ? numKeys : h;
            return GetKeyAtTimeLinear( t, l );
         }
      }
   return GetKeyAtTimeLinear(t, numPrefetchedKeyTimes * keysPerLump);
   }
   DataOnlyAnimKey GetKeyAtTimeLinear ( float t, int startIndex ) {
      int i = startIndex;
      while(i < numKeys) {
         if( keyTime[i] > t ) {
            --i;
            break;
         }
      ++i;
      }
      if(i < 0)
         return keys[0];
      return keys[i];
   }
};
```

Listing 6.3: 优化利用缓存行

既然这些键必然会被加载到内存中，我们就能趁机免费查询一些数据。代码 6.3 中，可以看到它使用了线性查找，而非二分查找，但根据测试结果，仍要快于二分查找。在此我们假设，像现代机器上的大多数情况一样，是因为代码所采取的路径更好地利用了资源，而不是因为理论更优，或用了更少指令。

```
i5-4430   @ 3.00GHz
Average   13.71ms   [Full anim key - 线性查找]
Average   11.13ms   [Full anim key - 二分查找]
Average    8.23ms   [Data only key - 线性查找]
Average    7.79ms   [Data only key - 二分查找]
Average    1.63ms   [Pre-indexed - 二分查找]
Average    1.45ms   [Pre-indexed - 线性查找]
```

如果查询需求比较简单，如检查某项是否存在，那还有更快的方案。布隆过滤器 (bloom filter) 的查询时间为常数。尽管它有一定的误识别率，但对于超大集合，可以做些调整，直至达到可接受的正确率。尤其是，如果要检查某行在哪张表中，那布隆过滤器的效果就非常好。它能返回需要的结果，一般只有正确的表，但有时不止一份。Google 的工程师们在 BigTable 技术 [7] 中，用布隆过滤器降低写入前方法 (write-ahead approach) 的开销，并用它快速确定，数据请求应在最近变化的表中查找，还是应直接进入后备存储。

在关系数据库中，如果索引的存在有利于多个查询，就会在运行时将其

添加到表中。对于面向数据方法，总有办法能加快查找速度，但只能通过查看数据。如果数据尚未排序，那就通过索引找到所需的特定项目。如果数据有序，但需求更快的访问速度，那针对缓存行大小优化过的查找树就有所帮助。

大多数数据没法那么简单就优化掉。重要的是，有大量数据时，通常很容易从其中学习到模式。很多时候，我们必须与空间数据打交道，但因为用对象，就很难事后再捆绑有效的空间参考容器 (spatial reference container)。运行时，给外部定义的对象类添加空间参考容器几乎是不可能的。

如果数据是像行这样简单的格式，添加空间分区后，就能够生成空间容器、查询系统。它们易于维护，易于优化。由于面向数据变换本身固有的可复用性，我们进而也为上层程序员提供了超高优化的代码。

6.3 寻找极值是排序问题

有时，其实不用真的去查找。如果是为了在某个范围内找到一些东西，如附近的食物、住所、掩体等，就不是查找问题，而是排序问题。在查询的前几次，可能真的会做一次查找以定位结果，但如果执行得够频繁，那为什么不把查询放进运行时更新的排序子集的表格数据中呢？如果需要最近的三个元素，那就保留最近三个的排序列表，当有元素更新、插入、删除时，就能根据信息，更新那三个元素。要对元素分布稀疏的集合做插入或修改，可以先检查该元素是否更近，并在添加新元素时pop出最小值，保证最佳排序。删除或修改时，想要从有序集合中找到一个元素，替代被删除的那个，就得快速检查其余元素，找到新的最佳集合。但如果保留的最佳值多于必要数量呢？估计这种情况永远不会发生。

```
Array<int> bigArray;
Array<int> bestValue;
const int LIMIT = 3;
void AddValue( int newValue ) {
   bigArray.push( newValue );
   bestValue.sortedinsert ( newValue );
   if( bestValue.size() > LIMIT )
      bestValue.erase(bestValue.begin());
}
void RemoveValue ( int deletedValue ) {
   bigArray.remove( deletedValue );
   bestValue.remove( deletedValue );
}
int GetBestValue () {
   if( bestValue.size () ) {
      return bestValue.top();
   } else {
```

```
        int best = bigArray.findbest();
        bestvalue.push( best );
        return best;
    }
}
```

Listing 6.4: 保存更多信息

诀窍就是，在运行时找到方案需求的最佳值。于是，唯一要做的就是运行时检查数据。为此，要么保留日志，要么根据查询优化器的反馈，动态调整大小并测试。

6.4 查找随机是哈希 (或树) 问题

有些表的数值变化非常频繁。为了确保树描述具备高性能，最好不要有大量修改，因为每次修改都可能导致重平衡。当然，如果在某个处理阶段做所有修改，然后再平衡，然后在另一处理阶段做所有读取，那么也仍可以使用树形态。

C++ 标准模板库中的 `map` 实现，即便一次性提交所有修改，可能也没法很好地帮到编译器。但一个考虑到缓存行的树实现 (如 B-树)，可能会有所帮助。B-树的节点更宽，也更浅。由于每个节点容量都更大，插入和删除操作一次发生多个变动的概率也更低。通常，红黑树每隔一次插入或删除，就会触发某种形式的平衡。但大多数 B-树的实现中，可能会有与节点宽度相关的旋转，而节点可以非常宽。如，有些节点有 8 个子节点，并且很常见。

如果某些数据有许多不同查询，最终会有多个不同索引。条目变化的频率应该影响索引数据的存储方式。每次查询都保留一棵树开销可能很高，但许多实现中，反而会比哈希表开销更小。哈希表在有很多修改和查找穿插的情况下，开销更小；而树在数据基本静态时，或着说，至少在多次读取时，以某种形式存在一段时间，这种情况反而更优。

数据为常量时，完美哈希表能胜过树。完美哈希表会用预计算的哈希函数生成索引，除了用于存储常量、指针数组、到原始表的偏移量外，不需要额外空间。如果时间充足，就有可能找到完美哈希，返回实际的索引。虽然通常时间没那么充裕。

例如，要怎样根据某人的名字找到他的位置？通常不会按名字排序玩家，所以要发起一次从名字到玩家的查找。这个数据在游戏中基本不会变，所以最好能找个方法直接访问。单一的查找几乎总是会带出指针链，所以用哈希寻找数组条目或许再合适不过了。假设有个普通哈希表，条目里有目标

和其他元素，并且提供方法计算出下一个要要查找的条目。若确定要做一次，且只有一次查找，就可以把哈希桶做得和缓存行一样大。这样就能免去额外的内存开销，何乐而不为。

第七章 排序

排序，对于某些子系统非常重要。排序图元渲染的调用，不透明对象从前向后渲染，对 GPU 性能有巨大影响，值得一做。同样，alpha 混合对象从后往前渲染，也有必要排序。按响度和采样位置排序音频通道，也能作为很好的优先级指标。

无论要对什么排序，都要先确认排序的对象，因为通常排序是一项高内存负载的任务。

7.1 真的必要吗？

有些算法看似需要数据有序，实则不用；有些则需要数据有序，但又看不太出来。做出任何错误举措前，最好先确认是否真的有必要。

游戏中，排序的常见用途之一是在渲染管线中。有些引擎程序员提议，将所有渲染调用按一个若干比特的键排序，该键是由深度、网格、材质、着色器、其他标志等（如是否 alpha 混合）组合而成。这样一来，渲染器就能在运行时调整排序，最大限度利用带宽。在排序渲染列表时，可以用常规排序算法来处理。但实际上，并不用排序 alpha 混合对象与不透明对象。所以很多情况下，可以直接采取第一步，将列表放入两个独立的桶中，总体上省掉一些工作。另外，排序算法也要谨慎选择。对于不透明物体，最重要的通常是：先按纹理排序，再按深度排序。但是，随着同一像素被多次覆盖，进而破坏填充率，情况可能会依破坏程度而改变。如果是纹理上传的重要程度高于过度绘制 (overdraw)，那么对调用做基数排序或许会比较好。而对于 alpha 混合，只需要按深度排序，所以只要适用当前情况即可。另外，还要注意数据排序的准确程度。有些排序是稳定的，有些不稳定。不稳定的排序通常会快一些。对于模拟的范围，快速排序或合并排序通常缓慢但准确。如果是 n 很大的离散区间，基数排序就难以超越。如果知道数值范围，那计数

排序是一种非常快速的 two-pass 排序，如，按材质、着色器、其他输入缓冲索引等排序。

排序时，也要注意那些只能在某个范围进行部分排序的算法。如果只需要一个 m 长的数组中最低（或最高）的 n 个元素，可以用不同类型的算法找到第 n 个元素作为枢轴，然后排序所有小于（或大于）它的元素。一些选择算法会带来数据的额外特征。比如，快速选择算法中，根据排序标准，第 n 个元素会保留在第 n 个位置。选择完成后，两边的所有元素在其子范围内都保持未排序状态，但根据其相对枢轴的方位，一定小于或大于枢轴。

假设有一组常规范围的元素，需要以两种不同的方式排序。可以用专门的比较函数一次性排序，也可以分层级排序。若整个范围的元素的任意子集的顺序不太重要，也能加以利用。此处仍用渲染队列举例。如果把排序分成不同的子排序，就有可能剖析排序的每个部分，或许就能发现有用的点。

当然也不需要自己实现算法。这里提出的大多数想法，STL 都已经实现过了，直接使用即可。例如，可以用 `std::partial_sort` 寻找和排序前 n 个元素，用`std::nth_element` 寻找第 n 个值，就跟排序了的容器一样。用 `std::partition` 和 `std::stable_partition` 根据某些准则，能有效地对某个范围的元素排序，并可以分成两个子范围。

了解这些算法的标准也很重要，因为像 `erase/remove` 这些简单操作，也可能会因为不自知的用法将整个数据打乱重排，因为它们会维持数据的有序状态。如果要给自己的工具集里添加新算法，那最好也实现自己版本的 `remove`，不去保证有序状态。代码 7.1 就是这样一个例子。

```
template <class It, class T>
It unstable_remove( It begin, It end, const T& value )
{
   begin = find(begin, end, value);
   if (begin ! = end) {
      --end;
      *begin = move( *end );
   }
   return end;
}
```

Listing 7.1: `unstable_remove` 的一种基本实现

7.2 插入排序与并行归并排序

根据排序的目的，修改时也能执行排序。如果排序针对需要优先级的 AI 函数，那可以用插入排序，因为基础启发式的，通常会有完全正交的输

入。如果输入是相关的，那么插入后再整表排序能保证有序，但基本没必要全排。

如果确实需要完整地排序，最好尝试倾向于并行的算法。归并排序和快速排序在某种程度上是串行的，它们开始和结束工作都位于单线程上，但也有些变体在多线程中工作良好。对于小数据集，有一些特殊的排序网络技术，可以比好的算法更快，只因为它们更适用于硬件[1]。

7.3 针对平台排序

时刻记住，在面向数据开发中，必须先从数据中寻找信息，再决定怎样写代码。数据是什么样的？在渲染中，有大量数据，有不同的维度可用于排序。如果渲染器是按照网格、材质排序，以减少顶点和纹理上传，那数据看起来就是：有些渲染调用共享纹理数据，还有一些共享顶点数据。通过计算上传纹理、网格的时间；各自又需要上传多少额外信息；然后计算场景的总时间；就能知道先排序哪边了。不过大多数情况下，分析报告是唯一能够确定的方式。如果希望能快速剖析并获取反馈，或者游戏中有相当多的资源要剖析，以至于没有方案适合所有的情况，最后不得不允许运行时做调整。那最好有些灵活的排序标准，有时候甚至是必须的。好在它只是需要一点设置成本，就可以与任何不灵活的排序技术一样快。

基数排序是最快的串行排序。如果能做到，在第一轮中为不同值的数据生成一个起点列表，然后在第二轮中用这些数据执行操作，基数排序是相当快的。排序器可以根据译表将内容放入容器中，其中译表是一个为给定数据值返回偏移量的表。如果从一个已知的小的数值空间建列表，那基数排序可以非常快速地在第一轮就给出粗略的结果。之所以是串行，是因为它必须修改正在读取的表，以便更新下一个元素的偏移量，这些元素会被放进同一个桶里。如果有多个线程，让它们各自承担一部分工作，就会发现它们的吞吐量是非线性增长的，它们会争相在同一块内存上读写，而我们并不希望在排序算法中用上原子更新。

通过让每个排序器忽略它读到的超出其工作集的值，能让这个过程的最后阶段变为并行。也就意味着为了填充自己地桶，每个 `worker` 都会读完整个值集。但由于必须在不同线程上向临近的内存写入，仍有小概率出现非线性的性能问题。在 `worker` 收集元素的过程中，它可能正在为序列中的下一

[1]Tony Albrecht 在他关于分类网络的文章中证明了这点
http://seven-degrees-of-freedom.blogspot.co.uk/2010/07/question-ofsorts.html

个基数生成计数，只需在下一轮之前求和，就能减轻每个 `worker` 在整个集合上迭代的成本。

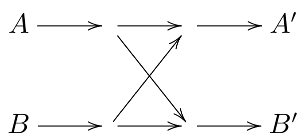
如果数据相对复杂，无法使用基数排序，那可以考虑归并排序和快速排序，但如果在编译期就知道可排序缓冲区的长度，那就还有其他替代方案，如排序网络。归并排序本身不是并发的算法，但很多前期归并能够并行，只有最后是串行的。通过快速预解析要归并的数据，就能用两个线程而非一个，从两端开始直至最终完成（需确保合并的数据不会耗尽）。尽管快速排序也不是并发算法，但每个子阶段都可以并行。它们本质上是串行的，但可以变成部分可并行的算法，延迟为 $O(\log n)$。

n 足够小时，原地冒泡排序通常也不错。它简单到很难写错，而且由于需要交换的数量很少，优化排序的功夫可以用在其他地方。重写这类简单的代码时，还有一种说法，内联实现能小到整个数据和算法都可以放进缓存[2]。这么小的 n，冒泡排序的低效带来的负面影响微乎其微，因此基本没人会反对。某些情况下，减少指令本身或许要比操作效率更重要，因为指令被逐出缓存的开销，可能比优化算法省下的还多。跟以前一样，测一下就知道了。

如果读者已经接触过面向数据开发，一定处理过排序有 n 个元素的表的变换。用到的算法不一定要优于冒泡排序，但是注意，用更好的算法并不需要额外的开发时间，因为数据已经是正确的格式了。面向数据的开发自然会引导我们复用好的算法。

寻找正确的算法时，学到的东西远比课程作业中所介绍的更多，我们能探究到更深奥的形式。对于排序，有时想要一个总是在相同时间内排序的算法，这样的话，标准的快速排序、基数排序、冒泡排序等都无法胜任。归并排序通常性能很不错，但为了确保时间真正稳定，可能还需借助于排序网络。

排序网络是以静态的方式实现。它们有输入数据，并在输出最终结果之前，对输入数据的数值对执行交换函数（必要的话）。最简单的排序网络是两个输入。



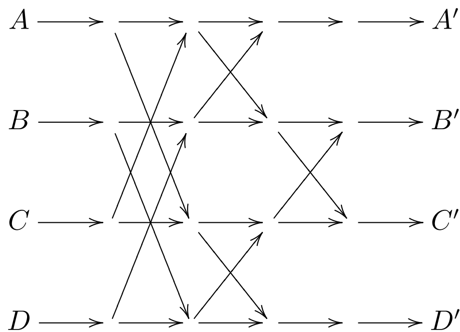
如果输入的值符合次序，则排序交叉什么都不做；如果不符合，那么排序交叉就互换数值；这样就能实现无分支写入。

[2] 在自己工具类的头文件里放些内联的排序函数模板或许还不错，这样就能利用上小型化的好处。但别把臃肿的 `std::sort` 放进去

```
a' <= MAX(a, b)
b' <= MIN(a, b)
```

这种操作在各类硬件上都很快。对于不同平台和数据类型，`MAX` 和 `MIN` 函数需要不同的实现，但通常来讲，无分支的代码比有分支的快一些。当前大多数编译器中，如果可以，`MAX` 和 `MIN` 函数都会被提升为内部函数 (intrinsic)，但我们可能需要微调一下数据，使其值成为键的一部分，所以它会与键一起排序。

引入更多元素：



可以看到，这里的关键路径并不长（共只有三个阶段），第一阶段：并发排序 A/C 和 B/D 对。第二阶段：排序 A/B 和 C/D 对。最后：排序 B/C 对。这些都是无分支函数，因此在整个数据处理序列里，性能有规律可循。有了这样一个有规律的性能曲线，就可以在排序时长易变的地方应用它，如对渲染的子模块执行即时 (just-in-time) 排序。如果对可渲染数据用了基数排序，就可以在任何最终需要有序的地方使用网络排序，因为可以保证时间稳定。

```
a’ <= MAX(a,c)
b’ <= MIN(b,d)
c’ <= MAX(a,c)
d’ <= MIN(b,d)
a’’ <= MAX(a’ ,b’ )
b’’ <= MIN(a’ ,b’ )
c’’ <= MAX(c’ ,d’ )
d’’ <= MIN(c’ ,d’ )
b’’’ <= MIN(b’’ ,c’’ )
c’’’ <= MAX(b’’ ,c’’ )
```

排序网络有点像是低阶断言 (predication)[3]，是处理条件计算的无分支形式。因为这里用到的是`min / max`函数，而非条件交换。所以涉及到单个元素的实际排序时，同样具备优势。鉴于排序网络在某些实现上比基数排序更快，因此，对于某些类型的计算，低阶断言，甚至是长链的低阶断言，都要比像借由分支节省处理时间的代码快。正是这样一个例子存在于 *Pitfalls of Object Oriented Programming* [17] 的介绍中，结论是惰性求值比通过它尝试避免的工作开销更大。还没有确凿的证据，但我相信很多 AI 代码也能从中受益，因为即便不确定是否需要，收集信息也很明智，收集可能比确定不用收集要快。例如，确认某人是否在玩家视野中，或够近，或很小，不管是不是每一个 AI 都需要这样的信息，我们为所有 AI 执行该检查。

[3]译注：同前文中的 predication 一样，尚不确定这个词的术语，用法与一阶谓词逻辑中 (或低阶断言逻辑) 相同

第八章　优化和实现

优化软件时，必须了解是什么导致软件的速度不及预期。大多数情况下，开销最大的是数据移动。它也是处理数据时最耗能的部分。而计算函数结果，或用算法处理数据，耗能都较少。优先提供所需的数据，似乎就是最大的成本。在当前的架构中确实如此，但我们发现隐式、可计算的信息，往往比缓存的值或显式的状态更有用。

如果开发游戏时，将数据组织成数组，就能带来诸多优化的机会。从这种与问题无关的布局开始，我们可以挑选针对其他任务创建的工具，最坏也是将方案提升为模板或策略，然后再将其应用于新旧用例。

在 *Out of the Tar Pit*[13] 中提到，除非是在方案的开发末期，否则不推荐为了性能而增加状态和复杂度。通过数组解决这个问题，并在没有副作用的情况下变换这些表，能够改进整个系统的性能。它们可以在程序的许多地方应用，也不需要担心兼容问题，同时保证不会增加状态，实际增强的只有当下工作的语言。

但许多项目的祸根，延期的原因，就是坚持不过早优化。后期优化之所以如此困难，是因为许多软件里到处都是对象的实例，甚至有时候都不是必需的。实例作为处理的单元，即是面向对象设计诸多问题的起源。面向对象的开发实践，倾向于认为实例是代码的工作单元，技术与实践标准将对象的集合视为个体的集合。

假设对象是某个独特的东西，有自己的用处，那么让它做事情的指令，必然以某种方式依赖于对象。通常会通过虚表查找访问实际的操作。更大的威胁是，当原本可以分组，或是以独立值结合增量的形式表示 5 、10、乃至 100 个单独的实例，现在却当作个体序列来处理。很多情况下，某个对象存在，只是因为在开发者实现的层面，它的体量看起来与现实世界的概念匹配，而不是因为在用户可见的层面，它需要作为独特的个体元素发挥作用。我们很容易陷入这种情况：从它们是什么的角度来实现功能，而不是应该如

何看待它们。

8.1 什么时候应该做优化?

什么时候应该优化？什么时候算过早？答案在另一种形式的数据里。过早优化是指在不知道是否会产生影响的情况下，优化某些内容。如果只是因为觉得它能”提高点速度”，就尝试优化，那就可能属于过早优化，因为此时还看不出有需要优化的迹象。

这里要说明，如果没有数据显示游戏运行缓慢或内存耗尽，那所有优化都是过早优化。如果应用程序没有被分析，但感觉很慢、很迟钝、不稳定，那做的任何事情都不能客观地定义为改进，而任何改进的尝试都只能算是过早优化。只有从真实的数据着手，才能停止过早优化。如果程序看起来很慢，且已经分析过，基于数据能够明确定义什么无法接受，那么任何为改进解决方案的行为都不会视为过早，因为它已经被测量过，可以在失败、成功、进步等方面做出了评估。

鉴于我们已经确认在某些时候需要优化，且知道不做分析的话，就不是优化，下一个问题就很清楚了。应该什么时候开始分析？应该什么时候开始分析框架的工作？要有多少游戏内容才足以保证测试性能？开始测试游戏的性能峰值前，应有多少玩法实现？

考虑另一个问题。最终的产品中，性能是否是可选项？如果知道游戏有模块在某些硬件上只有 5fps，还能发布吗？如果觉得 30fps 可行，即便相当不精确，也算是个衡量标准。那怎么知道游戏在目标受众的某个硬件配置上已经不是 5fps 了？只要相信帧率有下限，内存有上限，关卡有预期的最长加载时间，或确信游戏运行时至少应该有合理的耗电量，某种意义上，我们已经达成共识：性能不是可有可无的。

若性能并非可有可无，且需要实际工作来优化，那么就问自己一组不同的问题。分析能够推迟多久？能承受多少美术资源被重做（或其他内容）？尚不确定最终能否上线时，愿意实现多少功能？在没有反馈，且同样不知道能否进入最终产品时，愿意工作多久？

8.2 反馈

写出性能较差的代码却不自知，损害的不仅是程序。由于没有人对他们的工作提出反馈，开发人员就无法进步，反而会强化并延续那些不起作用的神话和技术。Daniel Kahneman 在 *Think, Fast and Slow*[8] 一书中有过相关论述：人们能够很好地在即时反应中学习，但当反馈时间较长，掌握技能就没那么容易了。书中提到：心理医生在与病人的互动中，有机会运用强大的直觉技能，因为他们能够观察到病人的即时反应；但在确定合适的治疗方案时，就难以建立强大的直觉，他们无法确保反馈总是完整且可用的，并且反馈往往还会滞后。决定在没有反馈的情况下工作毫无意义可言。但对许多游戏开发者，又几乎别无选择，第三方引擎为初学者提供的反馈机制非常少。除了 CPU、GPU、物理、渲染等粗略的粒度之外，没有提供其他机制，能够对其引擎的不同方面做预算。他们提供了很多工具用以帮助解决性能问题，但提供的反馈往往不完整；又或无法精确匹配最终产品的形式，因为在完全优化的发布版中，内置的分析工具也常常用不了。

当下的事情必须有所反馈，不然，原本该用来打磨的时间都得用来优化了。尽可能确保反馈完整，即时。为了实现这一点，可以添加游戏性能状态的指标。及时反馈优化结果，有助于减轻沉没成本带来的谬误，进而避免干扰理性的方向决断。如果开发者相信某种行事方式有用，但事实又不尽然，那最好尽早知道。即使是最固执己见的人，在数据面前也可以变得平易近人。好奇心，对于自尊心受创的开发者来说，是一剂良药。如果没有在某种方法上投入大量时间和精力，反馈就更容易整合了，开发者会更愿意放下当下的工作，找出不同的方法。

当然还要确保反馈的正确性。比如自己一直在优化一款游戏，想保证帧率丝滑，平均帧率保持在 60fps，但客户和测试人员却不断地反馈掉帧和帧峰值问题，那有可能是分析的内容或方法有误。有时候，还是有必要在游戏运行时做性能分析。除了平均帧数，只需要再记录每帧的数据就好。

性能分析也不一定只针对帧率。慢的不是帧，是帧内执行的内容。软件开发中，有个老式但强大的方法：为系统和模块做预算。这里讨论的并非财务预算，而是时间、内存、带宽、磁盘空间，或是其他直接影响最终产品的限制。如果每帧预算是 16ms，且绝不超过它，就能保证游戏有 60fps。没有如果，没有但是。假设觉得保证关卡加载时长令人舒适，并设定了 4 秒预算，只要不超时，就不会有人抱怨。

不止是游戏，比如在线零售网站，还要注意延迟问题，它也会影响到用

户。Greg Linden 在 2008 年的一次演讲中透露，延迟每增加 100 毫秒，亚马逊就会损失 1% 的销售额。根据谷歌的统计数据，有人透露，当他们在生成页面时只增加半秒的延迟，网站流量就会下降 20%。更有甚者，TABB 集团在 2008 年的评论中，提到了陡增的成本：

> TABB 集团估计，如果一个经纪人的电子交易平台比竞争对手慢 5 毫秒，他可能会损失至少 1% 的流量；平均每毫秒收入为 400 万美元。高达 10 毫秒的延迟可能会导致收入损失 10%。这里开始，情况越来越差。如果经纪人比最快的慢 100 毫秒，还不如直接关掉 FIX 引擎，去当场内经纪人了。[1]

如果延迟、吞吐量、帧时间、内存等资源会变为限制，就给它们做预算。什么会削弱业务？是否有在衡量它？多久可以不检查游戏是否超预算了？

8.2.1 了解极限

将预算纳入工作考量，就能在早期就为系统设置切实的预算，在开发过程中维持在一定水平上工作，避免开发后期带来困扰。没有预算的项目，帧峰值可能只在临近发布时才逐渐显现，只有那会儿，所有的系统才会共同创造出最终产品。产品发布前，没有任何迹象表现出可能的帧峰值问题，结果可能就是直接导致系统看起来很廉价。直到最终找到引发峰值问题的子系统，或许就是很久前的一次改动导致的。同时，在项目早期，资源充足，系统引起的峰值完全没法引起注意，因而躲过了监测。如果系统有预算，不合理的行为就能记录下来，并立即做出响应。这样，问题就可以在发生的时立马被逮住，定位 bug 也更容易。

无论是自己实现，还是找一个，必须有个能够持续运行的性能分析器。确保分析器能在框架时间超预算时，报告游戏的整体状态。对所有超预算的子系统做出响应，都会很有用。要想搞清楚究竟发生了什么，还需要截取问题发生时的若干帧。如果游戏中有 AI，要捕捉性能问题，就试试持续运行测试，就跟版本机构建测试版本一样。当然所有情况下，除非让真正的测试人员运行分析器，否则很难得到真实的数据。如果由测试人员使用，还得考虑如何收集数据。可以的话，看看否可以把自动生成的分析数据，送回分析（或度量）服务器，以便自动化捕获问题。

[1]摘自 Viraf (Willy) Reporter 撰写的 *THE VALUE OF A MILLISECOND: FINDING THE OPTIMAL SPEED OF A TRADING INFRASTRUCTURE*。

8.3 优化策略

优化不是只要打开编辑器就能马上着手的。先得有个策略。我们会在本节介绍一些。在游戏行业外，这些步骤也有相似之处。这些行业中，有些大公司 (如丰田) 会把优化作为其商业模式的一部分。为了确保最大的性能和增长，丰田的技术已经相当完善。其背后的驱动理念，是丰田生产方式[2]，能有效减少浪费。它还有其他可用的技术，本节中介绍的步骤与它们有很多共同之处。

8.3.1 定义问题

定义问题。找出什么是不好的。用事实支撑定义，并假设好的解决方案最终应是什么。简单举例来说，问题是游戏以 25fps 运行，而最终需要它以 30fps 运行。坚持使用清晰、客观的语言。

重要的是，在这一步不要包含任何猜测，所以应避免如：优化什么、如何优化这类陈述。考虑从用户角度来写，而非从开发者的角度。所以它有时候也被称作：质量标准 (或客户需求)。

8.3.2 测量

只测量有需求地部分。不同于随机测量，有针对性的测更有利于理清实际情况，毕竟不太可能在不相干的数据中找到模式。数据挖掘 (data dredging, 又称 P-hacking) 在这里会导致错判问题。

这个阶段，还需要对测量的质量有个概念。运行测试，然后再次运行，以确保可重复。如果改动前，无法重现结果，那要怎么能判断改动有效呢？

8.3.3 分析

多数非正式优化策略的第一步：猜测阶段。这时会想想哪些可能是问题所在，并提出不同的方法来解决。在非正式的优化过程中，通常先实施看起来最好的，或至少是最有趣的想法。

相对正式的策略中，会去分析前面测量的东西。有时从这一步就能看出，测量结果有没有提供足够的方向，支持提出好的优化方案。如果分析结

[2] 译注：丰田生产方式 (TPS:Toyota Production System) 是由丰田提出的一套整合的社会-技术系统，包含一套管理理念和实践。(摘自维基百科)

果表明数据不够好，下一步就应该先提高获取有用数据的能力。不了解误解问题的成本时，不要着急优化。

这一步也是做预测的阶段。评估一下计划预期的的改进效果。不是轻率地猜测，要通过一些计算来背书。等计划实施后，就没法再做了，会有太多的新的知识帮我们做可靠的推断。当然，也可能经历大家口中的知识的诅咒。这一步，可以了解你在估计优化效果方面的表现，还能在着手工作之前，了解改动相对会产生的影响。

8.3.4 实现

多数非正式优化策略的第二步：实现阶段。这一步会做可能解决问题改动。

可以的话，先试验性地实现一下方案。程序作为问题的解决方案，是一个解决数据变换的策略。设计实验请牢记这一点。

在得出本地的版本是否有效且确实值得做前，得证明它真的有用。检查本地实验中得到的测量结果，看它是否能保证在集成版本中测得的结果有相同预期。

如果优化第一次就很完美，那实验性的实现只能作为一个证明，证明这个过程可重复，适用于其他情况。但也只能作为一个教学工具，帮助他人了解原先实现的成本，以及在相似约束条件下预期的改进。

如果不确定优化第一次就成功，还可以做一些局部试验，避免完整实施，能省下不少时间。当然，为了给第三方提供支持而实现例子，也是个不错的开始，用较小的例子展示问题，会更容易沟通。

8.3.5 确认

这一步可能比预想的还要关键。有些人觉得它可有可无，但其实，它对于保留做优化时，产生的有价值的信息至关重要。

撰写一份报告，说明做了什么，发现了什么。这样做有两个好处。首先，分享优化技术知识，显然可以帮到其他遇到同类问题的人。其次，撰写报告能确定是否存在测量的错误，并确保测试的每一步都与最终的变化相关。

报告中，其他人可以指出不合逻辑的跳跃性推理，进而加深对问题地理解；也有助于排除错误的假设，并加深对机器工作原理的理解。写报告也是很强的体验，能锻炼思考能力，让我们能更好地解释某些条件下发生的事情。

8.3.6 总结

最重要的是，保持追踪。尽量在独立的工作测试平台上优化。即便因为要处理一个错误或功能，不得不与项目的其他部分同步，也要确保测试出的时间是可重复的。记录当下做的事情，这样在做早期修改时，可以帮助理解脑子里想的是什么，或可能没想到的事情。

还有就是，继续努力提高观察能力。如果不测量，就无法取得可衡量的进展；如果没有辨别改进的工具，就无法知道是否已经取得改进。还要适时改进测量工具。要寻找看的方法。每当发现当下的工具没法满足需求时，就去找满足需求的工具，找不到的话就自己实现。如果自己没法实现，就提需求，或委托别人实现。不要沉溺于向充满希望的乐观主义，会养成坏习惯，那样会随机证明你是对的，让人学到错误的真相。

8.4 表格

多方建议表明，将数据保持为矢量能让数据处理起来更方便。虽然有时要使用 STL 之外的内容，但学习它的特点，可以避免很多问题。无论是用 std::vector，还是实现自己的动态数组，都可以为将来的优化起个好头。大部分要做的处理，都只是读取一个数组；将数组转化为另一个；或原地修改一个表。一个简单的数组，就足以应付上述情况中的大部分任务。

转移到数组，算是开个好头，再进一步，转移到数组的结构 (structure-of-arrays) 还能更好。不过也不一定。主要还是得考虑数据的访问模式。如果不能考虑访问模式，而改变又很耗费，那就根据其他标准做选择，例如可读性。

之所以摒弃对象的数组 (arrays-of-objects) 和结构的数组 (array-of-structures)，是为了让内存访问更符合其任务。试想要如何组织数据，重点在于，哪些数据要加载，哪些数据要存储。CPU 是针对某些内存活动模式做优化的。许多 CPU 从读操作转到写操作时，都有一定的成本。如果能以一种非常可预测的串行方式，安排对内存的写入，或许就能帮助到 CPU 避免读写切换。例如代码 8.1，没有考虑到写的重要性，也没有冷热分离的例子。该代码试图更新那些既可读又可写的值，但相邻的数据却做只读操作。

```
struct PosInfo
{
   vec3 pos;
   vec3 velocity;
   PosInfo():
```

```
        pos(1.0f, 2.0f, 3.0f),
        velocity (4.0f, 5.0f, 6.0f)
    {
    }
};
struct nodes
{
    std::vector<PosInfo> posInfos;
    std::vector<vec3> colors;
    std::vector<LifetimeInfo> lifetimeInfos;
}nodesystem;
// ...
for (size_t times = 0; times < trialCount; times ++)
{
    std::vector<PosInfo>& posInfos = nodesystem.posInfos;
    for (size_t i = 0; i < node_count; ++i)
    {
        posInfos[i]. pos += posInfos[i]. velocity * deltaTime;
    }
}
```

Listing 8.1: 热读与热写和冷写相混合

代码 8.2 显著优化了性能。

```
struct nodes
{
    std::vector<vec3> positions;
    std::vector<vec3> velocities;
    std::vector<vec3> colors;
    std::vector<LifetimeInfo> lifetimeInfos;
};
// ...
nodes nodesystem;
// ...
for (size_t times = 0; times < trialCount; times ++)
{
    for (size_t i = 0; i < node_count; ++i)
    {
        nodesystem.positions[i] += nodesystem.velocities[i] * deltaTime;
    }
}
```

Listing 8.2: 确保每个流都是连续的

如果数据的读与写之间没有强关联，那数组的结构 (SoA) 就更利于缓存。记住，只有当数据不总是作为一个单元被访问时，才会如此。面向数据设计的倡导者之一认为，数组的结构本质上有利于缓存，于是把`x`，`y`，`z` 坐标分别放进独立的 `float` 数组中。列表足够大时，且每个元素位于自己的数组中，使用 SIMD 就能从中获益。然而，通常如果要访问数组中某个元素的其中一个维度，很可能同时也要访问另外两个。于是，对每个元素而言，会加载不止一行，而是三行缓存的浮点数据。如果操作涉及到很多其他值，那缓冲区可能会过满。这就是为什么要考虑数据从哪来，如何关联，如何使用。面向数据设计不单单是一套，从一种风格转向另一种的简单规则。要学

会看到数据间的联系。这个案例中，我们看到某些情况下，如果向量作为一个操作值，不太会被 SIMD 指令优化，那将其保持为 3、4 个 `float` 会更好。

还有一些其他情况，SoA 格式并不适用。如经常插入、删除数据。可以在周围保留一些，避免删除操作导致数组变换，进而减轻压力。但这样就无法避免元素在处理过程中，用到简单的同质变换。因为同质变换，往往就是这种数据布局变化的核心。

如果使用动态数组，并需要从其中删除元素，并且这些表通过一些 ID 相互引用。那就得想办法将这些表拼接在一起，方便处理。如让它们维持有序，以便压缩。如果按相同的值排序，可以实现为一个简单的合并操作，如代码 8.3。

```
ProcessJoin( Func functionToCall ) {
   TableIterator A = t1Table.begin();
   TableIterator B = t2Table.begin();
   TableIterator C = t3Table.begin();
   while( ! A.finished && ! B.finished && ! C.finished ) {
      if( A == B && B == C ) {
         functionToCall( A, B, C );
         ++A; ++B; ++C;
      } else {
         if( A < B || A < C ) ++A;
         if( B < A || B < C ) ++B;
         if( C < A || C < B ) ++C;
      }
   }
}
```

Listing 8.3: 合并多个表

只要 `==` 操作符知道表的类型，能找到要检查的特定列，且只要表基于这个列排序，就可以了。但如果这些表被压缩在一起，却没有按相同的列排序，该怎么办？如，有很多实体引用了 `modelID`，且很多*网格-纹理*组也引用了同一个 `modelID`。那这里就需要将实体朝向、实体渲染数据中的 `modelID`、*网格-纹理*组，这三个列表中匹配的行压缩在一起。最简单办法是，依次遍历每个表并寻找匹配，如代码 8.4 所示。这种方案虽然写起来很简单，但效率非常低，应极力避免。不过凡事总有例外。某些情况下，非常小的表或许用这种方式更有效率，它们能保持常驻，而对其排序可能会花更多时间。

```
ProcessJoin( Func functionToCall ) {
   for( auto A : orientationTable ) {
      for( auto B : entityRenderableTable ) {
         if( A == B ) {
            for( auto C : meshAndTextureTable ) {
               if( A == C ) {
                  functionToCall( A, B, C );
               }
            }
         }
```

```
    }
  }
}
```

Listing 8.4: 遍历所有表并连接

处理添加在不同列上的数据时，还得了解连接策略 (join strategies) 的用法。数据库中，连接策略用于减少在多个表中查询时的总操作数。在一个列（或多个列组成的键）上连接表时，有很多处理可供选择。之前的尝试中，我们只是简单地遍历了连接中涉及的每个表。对于大小相当的表，最终会是 $O(nmo)$ 或 $O(n^3)$。对小表来说可以接受，但大表就不行了。因此需要了解数据，判断其究竟是大是小[3]。如果太大，无法使用上文那样的常规连接，就得有替代策略了。

迭代或查找[4]都可以用于连接，甚至还能连接一次并缓存。维护一份连接缓存，似乎可以保证表能以多种方式排序的同时，还能对其执行操作。

当然，也完全可以添加辅助数据，然后以不同的顺序来遍历表。添加连接缓存，就像数据库允许一个表有任意数量的索引一样。创建索引，并在修改表时更新。我们的案例中，会根据需求实现索引。有些表可能会突然有大量写入，插入式排序就很慢。此时，在第一次读取时排序可能更好，或者在修改时丢弃整个索引。其他情况下，排序最好在写入时做，因为写的次数不多，或常与许多读操作交错进行。

8.5 变换

现在，我们来推广一下模式 (Schema) 的概念，静态的模式定义允许对迭代器采取不同的方法。模式迭代器可以用于访问一组表，取代通常那种，在容器内迭代以获取元素的访问权。这表示合并工作可以在迭代过程中完成，同时生成变换所依赖的环境 (context)。这种形式很适合数据处理不多的大型复杂合并，因为创建临时表占用的内存更少。而复杂的变换没法从中获益，因为它会降低下一组数据在缓存中，用于下一个周期的几率。

变换的另一方面是分离“什么”与“如何”。也就是说，将要变换的数据的收集、加载，与最终对数据执行操作的代码分开。有些语言会在基础章节中介绍 `map` 和 `reduce` ，但 C++ 中很少这么做。可能是因为列表不属于语言的基础部分，但如果没有这些，就很难引入并用到它们强大的工具。`map`

[3]取决于目标硬件，有多少行和列，以及你是否希望在不破坏太多缓存的情况下运行该进程

[4]通常查找连接也叫散列连接，如果足够了解数据，还可以用比散列更好的行搜索算法。

和 `reduce`，可以作为纯粹的变换与流程驱动程序 (flow driven program) 的基础。将大数据集变成单一的结果，听起来显然是串行。然而，只要其中的一个步骤 (即 `reduce`) 是关联的，那就可以并行处理掉相当一部分 `reduce`。

举例来说，一个简单的 `reduce`，为所有匹配的元素生成值为 0 或 1 的映射，产生一个最终的总值，再进一步处理成一个越来越少的平行归约树。第一步，对所有元素做归约 (reduction) 操作，获得奇偶对，并用相同的过程产生一个新列表。归约操作会一直执行，直到仅剩一个条目。当然，这种特殊归约法用处不大，因为每次归约都很琐碎，最好根据核心的数量将工作分配出去，最后做一次求和。还有一种更复杂，但同样有用的归约法：将一连串的矩阵串联起来。矩阵运算满足结合律，但不满足交换律。因此，矩阵链可以用与计算总工作量相同的方式归约。只要在归约过程中保持顺序，并且归约步骤满足结合律，就可以在许多通常看起来是串行的工作中应用并行处理。除了矩阵串联，浮点乘法也可以，例如多种颜色调制，如光、漫反射、游戏相关的着色。建立字符串、列表也都可以利用到结合律。

8.6 碰撞的空间集

碰撞检测中，通常有一个全面阶段，会大量减少碰撞检测的数量。投出射线时，通过八叉树、BSP、又或者其他空间查询加速器，找到潜在的交点，通常都会很有帮助。寻路时，也可以通过查询局部节点，帮助选择旅程的起始节点。

所有空间数据存储都是通过减少处理量来加速查询的。它们基于一些空间标准，返回缩小的集合。并且由于更短了，变换为新数据的成本也更低。

当前支持空间划分的库，需要适配任意结构。但由于我们所有的数据已经按表组织，因此，为任意表布局编写适配器就容易得多。通用算法也容易实现，并且实现可能会用于多处的代码时，也不会引入副作用。使用基于表的方法，由于不清楚其意图（也就是说，并不知道空间系统，是否用在技术层面上不属于空间的数据上），于是我们可以在意想不到的地方使用空间划分算法。例如分配音频通道，不仅可以通过它们与听众的距离，还可以通过音量和重要性。制作一个五维 (甚至 n 维) 空间划分系统，未来可期。只需要写一次，实现一次单元测试，就可以用了，即便用于一些奇怪的事情也值得托付。按任务进度进行空间划分或许就有点矫枉过正了，但按位置、威胁、奖励获得附近所有能当作兴趣点的实体集合，对于 AI 来说或许就非常有用

了。

8.7 针对大量数据惰性求值

优化面向对象代码时，经常发现隐藏在可变成员变量中，有已完成计算的局部缓存。大多数层级更新都会用到一个技巧：脏位（dirty bit），用于表示经由树的子成员或父成员，决定该对象是否需要更新。遍历层级时，脏位会基于刚刚加载的数据触发分支，意味着无法去猜测结果。因此大多数情况下，在准备阶段会读取不必要的内存。

如果计算的开销很大，那可能想要避免渲染引擎当下使用的路线。渲染引擎中，即便每帧都执行每个场景的矩阵连接，往往也比找到必要的矩阵并计算的开销要低。

例如，Tony Albrecht 在他的 *Pitfalls of Object-Oriented Programming* [17] 演讲的幻灯片中表示：检查脏位的作用不如不检查，检查失败时（对象不脏的情况），原本只要 12 个周期的计算与分支错误预测的代价（23-24 个周期）相形见绌。事物总是在发展，在后来的演讲 *Pitfalls revisited* [18] 中，他指出之前通过手动去虚拟化获得的改进不再有效。无论是编译器的改进，还是硬件的变化，现实总会打败经验。

如果计算的开销很大，尽量避免用大量的检查去查看数值是否需要更新，进而使游戏陷入困境。这是基于存在的处理再次发挥作用的时候，因为存在于脏表中，就意味着它需要更新。更新一个脏元素时，它可以把新的脏元素 `push` 到表的末端，如果能改善带宽，甚至可以预取。

8.8 必要性-只取所需

标准化数据，同时会降低遇到面向对象开发的另一个多方面问题的几率。C++ 中对象的实现，迫使不相关的数据共享高速缓存行。

对象按类收集数据，但许多对象设计上包含了多于单一职能的数据。这是因为面向对象开发，并不天然允许对象根据其职能重新组合，同时也因为 C++ 需要提供一种方法，让我们能够以一种简单的方式，保证可以重载系统级的内存分配，同时还能以面向对象编程。多数类包含的内容都不只是最低限度，或因为继承，或因为对象需要应对多种情况。除非很仔细地去布局类，否则许多只需从类中获取少量信息的操作，都会在缓存中加载许多不必

要的数据。只使用极少量加载的数据，是面向对象程序员最常见的过失之一。

每次虚拟调用，都会加载包含实例的虚表指针的缓存行。如果这个函数没有使用该类中位于虚表指针之前的数据，那缓存行的利用率可能就只有不到 4% 了。这是对内存吞吐量的浪费，如果不重新考虑如何调度函数，是无法恢复的。类调用自己的虚函数时，添加一个 `final` 关键字会有所帮助，但如果这些函数通过基类调用，就没用了。

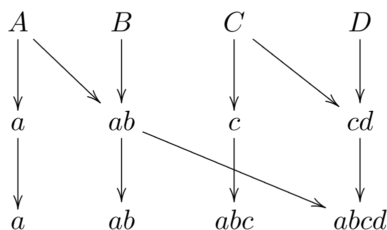
实践中，只有加载函数后，CPU 才能加载要处理的数据，这些数据也可能分散在为类分配的内存中。解码出虚表指向的函数指令前，它不知道自己需要什么数据。

8.9 变化的长度集

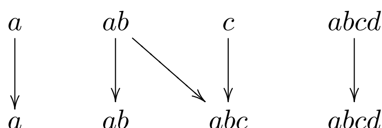
目前为止，我们用到的技术中，数据一直会隐含一个表结构。根据变换的需求，每行都可以是一个结构，又或是每张表都是一列列数据。使用流处理时，如，着色器通常会使用固定大小的缓冲区。大多数流处理的工作也有类似的限制，这里我们倾向于，两个方向都固定元素数目。

已知输入是输出的超集的过滤，或许很适合采用退火结构。先输出到多个独立向量，并在最后归并时将他们串联在一起。每个变换线程都有自己的输出向量，归并步骤会先为每个条目生成总数和起始位置，然后处理归并列表，到最终连续的存储器上。并行的前缀和 (prifix sum) 会比较有用，但简单的线性传递其实已经够了。

若过滤是径向排序、计数排序、或是使用类似的直方图生成偏移量的某个阶段，还可以用并行的前缀和，减少生成偏移量的延迟。前缀和是运行的值列表的总和。举例来说，径向排序输出直方图，桶计数会以所有直方图桶的总和作为起始位置。$o_n = \sum_{i=0}^{n-1} b_i$ 。这样很容易以串行形式生成，但如果是并行，就必须考虑生成结果所需的最少操作。这种情况下，记住，最长链会是最后一个偏移量的值，即所有元素的总和。通常优化会以二叉树的形式求和。分而治之：先将所有奇数槽与所有偶数槽相加，然后针对前一阶段的输出，执行相同操作。



然后，获得最后一个元素之后，回填在生成最后元素的过程中尚未完成的其他元素。实现时，我们发现这些回填的值，可以与生成最长链的处理并行。它们没有依赖最终值，所以能交给其他进程，或巧妙运用 SIMD 管理。



并行前缀和虽然能够减少延迟，但不是比线性前缀和更好的通用方案。同样的工作，线性版本用到的资源更少。所以如果能处理延迟，就简化代码，以线性方式求和。

另外，实体的数量也可能增减，这样的话，就需要能够无障碍地添加和删除。若要就地变换数据，就得处理这种情况：一个线程可以读取和使用正在删除的数据。要在通过内存分配来确定对象是否存在的系统中做到这一点，就很难删除被其他变换引用的对象。当然，还有智能指针。但在多线程环境中，智能指针需要一个 `mutex` 来保证每次引用和解引用都是线程安全的。这样成本就很高了，那要如何避免它呢？至少有两种方法。

不用 `mutex`。一种是将使用的智能指针类型绑定到某个线程上。有些游戏引擎中的智能指针，并不维护 `mutex`，而是存一个它们所属线程的标识。这样就可以断定，每次访问都是在同一线程上。性能方面，这些数据无需出现在发布版中，因为检查是为了防止运行时出现由编译期的决定导致的误用。例如，如果知道数据音频子系统使用，而音频子系统是在自己的单线程上运行，那就锁起来，把内存分配与音频线程关联起来。任何时候，音频系统的内存在音频线程外被访问：要么就是因为音频系统的内存暴露给外部了；要么是它在某些回调函数中做了本职外的工作。无论哪种情况，不良行为都会被断言捕获，进而可以修复代码，以应对常规问题，而不是什么特定情况。

不删除。如果在一个不断变化的系统上执行删除，通常会用到一个池 (pool)。主动不删除，而是做其他事情，就能改变代码访问数据的方式。改变了数据所代表的内容。如果一个实体需要存在，比如 `CarDriverAI`，那在

使用时，可以存在 `CarDriverAI` 表里，不用时也不删除，只是标记为未使用。不同于删除，这个实体仍然有效，不会在变换时崩溃，但直到有最新的 `CarDriverAI` 请求覆盖它前，都可以当它不存在，直接跳过。表中只有少量“死去”的实体，开销就跟维护组件池差不多。

8.10 间接连接

有时，标准化需要将表连接起来，方便创建查询。不同于 RDBMS 查询，我们能更仔细地组织查询，并通过合并排序算法合并两个表。还有一种替代方案，不一定要输出到一个表里，也可以通过传递式的变换，输入多个表，为另一个变换生成新的数据流。如，每个 `entityRenderable`，通过 `entityID` 与 `entityPosition` 连接，用 `AddRenderCall(Renderable, Position)` 变换。

8.11 数据驱动的技术

除有限状态机外，还有一些常见的数据驱动编码实践。不太明显的如回调；明显的如脚本。两种情况都会改变代码流的数据，也都会引发同虚拟调用、有限状态机中一样的缓存和管线问题。

事件订阅表的触发器能让回调更安全。与其在工作完成时启动回调，不如维护一个已完成的事件表，回调就可以在整个运行完成后被触发。例如，假设一个评分系统有来自 `badGuyDies` 的回调，在面向对象的消息监听中，评分器只要收到 `badGuyDies` 的消息，就会在内部使分数上涨。相对的，只要确认 `badGuys` 集全部死亡，就运行回调表中的每个回调。如果一直这样，并且每次都要检查 `badGuys` 后再执行，那其实可以一次性为所有杀死 `badGuys` 加一次分。内部状态只执行了一次读取和一次写入。这比通过多次读写累加出最后分数强得多。

对于脚本，如果有在多个实体上运行的脚本，考虑一下图形内核是如何操作分支的。有时会用预测法：做出选择之前，分支两边都会执行。这样就会减少因按需解释脚本带来的分支数量。再进一步，将 SIMD 真正构建到脚本核心中，相较于传统的逐实体的串行脚本，或许就能发现，可以在更多实体上使用脚本。如果 SIMD 操作是在整个实体集合上执行，那解释脚本

几乎没有额外代价[5]。

8.11.1 SIMD

只要有大量任务需要处理，如更新粒子位置的操作（见代码8.5），SIMD就非常有用。这个 SIMD 化的例子很简单，根据测试结果，它比结构的数组(AoS)、单纯的数组的结构 (SoA) 都要快四倍以上。

```
void SimpleUpdateParticles( particle_buffer *pb, float delta_time
) {
   float g = pb->gravity;
   float gd2 = g * delta_time * delta_time * 0.5f;
   float gd = g * delta_time;
   for( int i = 0; i < NUM_PARTICLES ; ++i ) {
      pb->posx[i] += pb->vx[i] * delta_time;
      pb->posy[i] += pb->vy[i] * delta_time + gd2;
      pb->posz[i] += pb->vz[i] * delta_time;
      pb->vy[i] += gd;
   }
}
void SIMD_SSE_UpdateParticles ( particle_buffer *pb , float delta_time ) {
   float g = pb->gravity;
   float f_gd = g * delta_time;
   float f_gd2 = pb ->gravity * delta_time * delta_time * 0.5f;
   __m128 mmd = _mm_setr_ps( delta_time, delta_time, delta_time, delta_time );
   __m128 mmgd = _mm_load1_ps( &f_gd );
   __m128 mmgd2 = _mm_load1_ps( &f_gd2 );
   __m128 *px = (__m128 *)pb->posx;
   __m128 *py = (__m128 *)pb->posx;
   __m128 *pz = (__m128 *)pb->posz;
   __m128 *vx = (__m128 *)pb->vx;
   __m128 *vy = (__m128 *)pb->vy;
   __m128 *vz = (__m128 *)pb->vz;
   int iterationCount = NUM_PARTICLES / 4;
   for( int i = 0; i < iterationCount ; ++i ) {
      __m128 dx = _mm_mul_ps(vx[i], mmd );
      __m128 dy = _mm_add_ps(_mm_mul_ps(vy[i], mmd ), mmgd2 );
      __m128 dz = _mm_mul_ps(vz[i], mmd );
      __m128 newx = _mm_add_ps(px[i], dx);
      __m128 newy = _mm_add_ps(py[i], dy);
      __m128 newz = _mm_add_ps(pz[i], dz);
      __m128 newvy = _mm_add_ps(vy[i], mmgd);
      _mm_store_ps(( float *)(px+i), newx);
      _mm_store_ps(( float *)(py+i), newy);
      _mm_store_ps(( float *)(pz+i), newz);
      _mm_store_ps(( float *)(vy+i), newvy);
   }
}
```

Listing 8.5: 使用 SIMD 更新粒子的简单实现

许多编译器优化中，默认会做简单的矢量化，但也仅限于编译器清楚的部分。要搞清楚这些也没那么容易。

[5]请看 BitSquid 中题为 The Massively Vectorized Virtual Machine 的博客：http://bitsquid.blogspot.co.uk/2012/10/a-dataoriented-data-driven-system-for.html。

在支持 SSE 的机器上执行 SIMD 运算，能一次性向 CPU 输入更多数据。许多人一开始就把 3D 向量放到 SIMD 单元中，但其实并没有完全利用到 SIMD 管线。本例同时加载了四种不同粒子，并同时更新。这种技术同时也不需要对数据布局做什么花哨的处理，只需要为 SIMD 准备一个单纯的数组的结构，就能规避掉数据瓶颈。

8.12 数组的结构

保持运行时数据处于数据库形式的格式，除了之前提到的好处外，还有机会利用到数组的结构。SoA 已经成为一个描述对象数据访问模式的术语。可以在 SoA 对象中同时保留冷、热数据，数据会按需载入缓存，避免按容易出现意外的物理地址。

比如有一个动画的 timekey/value 类是这样：

```
struct Keyframe
{
   float time, x, y, z;
};
struct Stream
{
   Keyframe *keyframes;
   int numKeys;
};
```

Listing 8.6: 动画的 timekey/value 类

那如果在大集合上迭代时，所有数据都得一次性载入缓存中。假设一条缓存行是 64 字节，`float` 4 字节，那 `Keyframe` 结构就是 16 字节。于是每次查询一个键的时间，都会不小心载入四个键，还有所有相关的 `Keyframe` 数据。如果在对一组 128 个键的 `Stream` 执行二分查找，就意味着可能最后加载了 64 字节数据，但在多达 5 个步骤中，只用了其中 4 字节。如果改变数据布局，让查找发生在同一个数组中，且分开存储数据，那么得到的结构会是这样：

```
struct KeyData
{
   float x,y,z;
   // consider padding out to 16 bytes long
};
struct stream
{
   float *times;
   KeyData *values;
   int numKeys;
};
```

Listing 8.7: SoA

这样做，对于 128 个键的 stream，times 总共只占 8 条缓存行，二分查找最多只需要载入其中 3 个，而数据查只需要一个。如果数据跨越了两个缓存行，但由于选择内存空间效率而非性能，因此最多也只需要两个。

这里是先有的数据库技术。DBMS 术语中，它被称为面向列的数据库，与传统的面向行的关系数据库相比，数据处理拥有更大的吞吐量。原因也很简单，因为在执行列聚合、过滤时，无关数据不会被加载。还有其他一些功能让列存储数据库更有效率，比如允许它们在一个值下收集许多键，替代键值一对一的映射。数据库一直在进步，所以值得去查阅最新的文献，看看还有什么值得迁移到自己的代码库里。

第九章　帮助编译器

编译器在优化代码方面，能做到相当出色。但有些编码方式会让编译器难以处理。一些技巧会打破编译器能做出的假设。本节中，我们会探讨一些应尽量避免的事情，以及如何养成习惯，让编译器更容易遵从我们的本意。

9.1　减少顺序依赖

如果不管编译器能否推断出操作顺序，那它就无法提前完成工作。将翻译好的代码合成为中间形式，有些编译器会用到一个指标：静态单赋值形式 (static single assignment form，SSA)。它基于这样的想法：一旦变量被初始化赋值后，就不再修改，如果确实需要修改，就创建新变量。不过在循环中还无法应用，因为只要是传输操作，所赋的值都需要改变。不过我们可以尝试去逼近，这样就能帮助编译器理解：修改、分配值时的意图。粗略过一遍 Haskell、Erlang、Single-Assignment C 等语言现在的功能和教程，就有足够的线索，学会以单赋值的方式写代码。

这样更容易看出编译器在哪里要分支，并且可以让写入更加明确。同时也意味着在编译器不得不停止写入内存的位置，我们可以强迫它在所有情况下写入。这样处理更同质化，因而保证流[1]的效率。

9.2　减少内存依赖

由于依赖关系存在，链表的开销很大，不过这属于另一种依赖关系。内存很慢，我们当然希望执行操作时它能及时加载。但是需要加载的地址本身也还在加载时，就没法作弊了。指针驱动的树形算法很慢，不是因为内存查找，而是因为链式内存查找。

[1] 译注：stream

如果想让 `map`、`set` 的实现更快，可以尝试宽节点算法，如 B-树，或 B*-树。希望将来某天，STL 中的 `std::map` 和 `std::set` 的实现可以自由选择。

如果有一个组合风格的实体-组件-系统，不过是基于指针的组合，那通过两层指针来获取组件的速度就会变慢。如果这些组件里还有指针，问题只会更复杂。

尽量减少获取所需数据的跳转次数。每一次依赖于先前数据的跳转，都可能会导致读取主内存引发的停顿。

9.3 察觉缓冲区写入

写入时要考虑的问题，与读取时相同。尽可能保持连续。尽量让可修改的值与只读的值分开，也与只写的值分开。

简而言之，连续写入，单次大量写入，用上所有字节，避免零散的字节。之所以这样尝试，因为它不仅有助于激活、停用不同内存页，也能帮助编译器做优化。

假设有一块缓存，有时知道如何绕过它也很重要。如果知道当前加载的数据只会用一次，或至少短时间内不会从缓存中获益，那了解如何避免污染缓存就很有用了。用简单的方法实现变换，能够帮助编译器将污染缓存的操作，修正为完全绕过缓存的指令。这些流式操作通过避免挤出随机访问的内存，从而让缓存获益。

Ulrich Drepper 在 *What every programmer should know about memory* [6] 一文中谈到了内存的方方面面，其中对于如何充分利用计算机硬件很有意思。这篇文章中，他用非时间性 (non-temporal) 这个词来描述我们称之为流的各种操作。这些非时间性的内存操作之所以有用，是因为它们完全绕过了缓存。粗看似乎是个糟糕的选择，但正如其名称暗示的，流式数据短期不可能被召回到寄存器中，所以在缓存中保有这些数据毫无意义，反倒会挤出潜在有用的数据。因此，流式操作允许一定程度上决定哪些重要，哪些肯定算不上。

9.4 别名 (Aliasing)

别名是指相同的内存地址可能被多个指针引用。因此，如果写入一个指针，则需要在读取前重新加载。举个例子，假设要找的值是通过引用而不是值确定的，如果有任何函数，有几率影响到该引用指向的内存，那在做比较之前，就必须重读该引用。实际上，正是因为它是指针，而非值，才导致了这样的问题。

以不可变的方式处理数据的原因之一，就是为优化做准备。C++ 给程序员提供了很多自讨苦吃的办法，其中最厉害的就是，不仔细使用内存指针时，能引发意外的副作用。考虑如下代码：

```
char buffer[ 100 ];
buffer[0] = 'X';
memcpy( buffer +1, buffer, 98 );
buffer[ 99 ] = '\0';
```

Listing 9.1: 拷贝字节

如果只是想得到一个长度 99，且都是 `'X'` 的字符串，这份代码完全没问题。然而，有这种可能，`memcpy` 需要一个一个字节地复制。为了加快速度，通常会一次载入大量内存地址，在它们全部进入缓存后再保存出来。如果写入输出数据可能会修改到输入的缓冲区，那就得非常谨慎了。现在考虑这个问题：

```
int q=10;
int p[10];
for( int i = 0; i < q; ++i )
   p[i] = i;
```

Listing 9.2: 可并行的代码

编译器能发现 `q` 不受影响，且很乐意解开循环，或是用一个寄存器的值替换对 `q` 的检查。但反过来看这段代码：

```
void foo( int* p, const int &q )
{
   for( int i = 0; i < q; ++i)
      p[i] = i;
}
int q=10;
int p[10];
foo( p, q );
```

Listing 9.3: 潜在的别名 (int)

编译器无法判断 `q` 是否会受对 `p` 的操作的影响，所以它每次检查循环的结束时都要存储 `p` 并重新加载 `q`。这就是所谓的别名，不知道两个正在使

用的变量的地址是否不同，所以为了保证代码功能正确，必须把这些变量当作可能位于相同地址来处理。

9.5 返回值优化

如果要返回多个值，通常是通过引用参数，或是填充一个对象的引用。许多情况下，通过值返回开销其实很低，许多编译器可以把它变成一次非拷贝操作。

如果一个函数试图通过在返回过程中原地构造一个结构，它可以将构造过程直接转移到将接收它的值中，无需拷贝。

利用 `std::pair` 或其他小的临时结构能有效帮助更多代码以值的形式运行，不仅从本质上让推理变得更容易，编译器也更容易优化。

9.6 缓存行利用率

我们知道，一次内存请求总是至少会读到一个完整的高速缓存行。这个完整缓存行包含多个字节数据。写这本书时，常见的缓存行大小似乎已经稳定在了 64 字节。基于这些信息，就可以推测出哪些数据只通过与其他数据的相对位置来访问开销更低。

在第 6 章 (查找)，我们用这些信息决定数据的位置和数量，通过示例，创建了快速查找表，该表使用两层线性查找，结果比二分查找要快。

当有一个要加载到内存中的对象，就需要计算出缓存行与对象的大小之间的差值。这个差值就是可以存入免费读取的数据的大小。利用这个空间去回答该类的常见问题，就能显著提升速度，因为不会再额外访问内存了。

例如，考虑有一部分要迁移到组件的代码，但仍有一个实体类指向组件数组中的一条可选行。这种情况下，我们可以将这些实体在用的元素，整合成比特集，缓存在实体类的后半部分。于是实体与实体间的交互，就可以省去在未匹配行时查找数组的过程。它还可以提高渲染性能，渲染器可以立即知道没有受伤，然后只显示满血图标，或什么都不显示。

第 16 章[2]中，代码 16.11[3] 尽量利用对象的初始缓存行，应对对象其余部分的问题，从结果来看，获得了不同程度的收益。完全缓存结果的情况下，

[2]译注：英文付费版中的源码章节，可以在本书的 github 中看到。

[3]译注：本次翻译基于免费章节，16 章为付费章节。

则有巨大改进。如果结果无法快速计算，并又急需，缓存也被占用，也能有 25% 的改进。能够缓存结果时，根据缓存命中情况，也有不同程度的性能改进。总而言之，利用缓存行上的额外数据总是比基础的检查有进步。

```
i5-4430  @ 3.00GHz
Average  11.31ms  [基础-直接检查 map]
Average   9.62ms  [部分缓存 (25%)]
Average   8.77ms  [部分缓存 (50%)]
Average   3.71ms  [基础-有缓存]
Average   1.05ms  [部分缓存 (95%)]
Average   0.30ms  [完整缓存]
```

总的来说，只要知道，加载任意内存都是在加载一个完整的高速缓存行。目前，64 字节的缓存行，相当于一个 4x4 的浮点数矩阵，8 个 `double`，16 个 `int`，一个 64 字符的 ASCII 字符串，或者 512 位。

9.7 伪共享 (False Sharing)

如果某个 CPU 核心不与其他核心共享资源，那它一定可以独立全速运行，是么？不一定。即使 CPU 核心是在独立数据上工作，有时它也会被缓存阻塞。

与线性写入遇到的问题相反，在向同一缓存行写入数据时，会干扰到线程。不过随着编译器进步，这种情况已经相对少了很多。想要重现它，观察其影响的话，只有关闭优化才有极小的几率验证。

这个想法源于多个线程可能读取、写入同一个缓存行，但不一定是缓存行中的相同地址。确保任意快速更新的变量保存在线程内，不管是堆栈还是线程局部存储，都可以相对容易地避免这种情况。其他的数据，只要不是定期更新，就很难引起冲突。

```
void FalseSharing () {
   int sum =0;
   int aligned_sum_store[NUM_THREADS] __attribute__((aligned (64)));
#pragma omp parallel num_threads(NUM_THREADS)
   {
      int me = omp_get_thread_num ();
      aligned_sum_store[me] = 0;
      for (int i = me; i < ELEMENT_COUNT; i += NUM_THREADS) {
         aligned_sum_store[me] += CalcValue( i );
      }
#pragma omp atomic
      sum += aligned_sum_store[me];
   }
```

```
}
void LocalAccumulator() {
    int sum =0;
#pragma omp parallel num_threads(NUM_THREADS)
    {
        int me = omp_get_thread_num ();
        int local_accumulator = 0;
        for(int i = me; i < ELEMENT_COUNT; i += NUM_THREADS ) {
            local_accumulator += CalcValue( i );
        }
#pragma omp atomic
        sum += local_accumulator;
    }
}
```

Listing 9.4: 伪共享

这个特殊的问题，已经有很多人在谈论，但实际情况与实际问题所呈现的不同。在优化前后持续跟进，确保真的是这个问题，即便能力很强的人也可能在此跌跟头。

那么，如何判断是这个问题？如果多线程代码在增加内核时，处理速度不是线性增长，那就可能被伪共享所扰。看看线程在哪里写，尽量彻底从共享内存移除写操作。常见的例子，如将一些数组相加，并在全局共享位置更新总和，如代码 9.4。

`FalseSharing` 函数中，`sum` 作为共享资源被写入，每个线程都会触发缓存清理，并在其他核心更新自己在缓存行中的元素之前，将缓存行标记为脏。在第二个函数 `LocalAccumulator`，每个线程在写结果前，都会先在内部求和。

9.8 注意推测执行

推测执行，能提前准备可能用到的指令和数据，有效地在确定需要之前，就完成工作，但有时也会带来不利影响。如，之前提到的代码，已经部分以组件形式实现。目前用来表示可选行的比特集，可能通过推测，载入这些数组的信息。因为有推测执行，所以需要注意代码是否意外预取了数据，同时在等待某个比较结果。这些推测操作在 SPECTRE 和 MELTDOWN 漏洞的新闻中都有所提及。

如果可以，通过预先计算低阶断言减少分支预测引发的读取，将普通查询的结果存储在行中，对于大多数机器来说都有很大收益，而对于内存延迟高、CPU 与内存带宽比率高的机器，就是更大的收益。尝试降低分支预测错误对数据带来的副作用的技术，一般来说也不错。即便只在可行时缓存，

将结果存在初始部分，随时间推移，也能节省带宽。

在缓存行利用率部分，数据显示，获取数据的可能性，似乎也会影响进程的速度，并且比预期的还要多得多。因而人们相信，非必要数据的推测性负载可能不利于整体吞吐量。

即便只能缓存一个请求是否会返回结果，那也有用。保留是否有符合该描述的条目的数据，以避免查询复杂的数据结构，能够提高速度，且很少带来副作用。

9.9 分支预测

CPU 停顿的主要原因之一，归根结底，要么是无事可做；要么是预测得不好，不得已去分解已经完成的工作。如果代码推测执行，请求了不必要的内存，那负载就浪费了内存带宽。已完成的工作会被否决，重新开始（或继续）正确的工作。要解决这个问题，有些方法可以让代码无分支；还有方法是了解 CPU 的分支预测机制，并帮助它解决。

如果预测足够明显 (trivial)，大多数时候预测器都能做对。如果确保大块的条件始终是为 `true` 或 `false`，那预测器犯的错误会更少。举个简单的例子，如代码 9.5，根据传入的数据，预测是否需要累加。如果 CPU 支持，大多数编译器可以用条件移动 (conditional move) 指令优化这里。如果把工作做得更真实些，那就算开了全局优化，在对数据排序以提高分支可预测性的情况下，也能看到非常大的差别 [4]。另一个重点，如果编译器能提供帮助，就让它来吧。优化后的明显的例子只在与常规工作负载对比时才凸显其明显，但如果实际工作被优化为条件执行，那排序数据就是浪费了。

```
int SumBasedOnData () {
   int sum =0;
   for (int i = 0; i < ELEMENT_COUNT; i++) {
      if( a[i] > 128 ) {
         sum += b[i];
      }
   }
   return sum;
}
```

Listing 9.5: 基于数据执行任务

[4] 在 i5-4430 上，未排序的求和执行了 4.2ms，排序版本执行了 0.8ms。明显版本，可能主要被编译成 CMOVS。排序和未排序版本均能在 0.4ms 内完成。

```
i5-4430  @ 3.00GHz
Average  4.40ms  [随机分支]
Average  1.15ms  [有序分支]
Average  0.80ms  [明显随机分支]
Average  0.76ms  [明显有序分支]
```

分支是因为数据产生的，记住，分支不好的原因不是因为跳转开销大，而是因为预测错误所做的工作必须撤销。因此，务必记住虚表指针也是数据。不批量更新，就无法最大化利用分支预测器，但即使根本不触发分支预测器，也可能会基于数据提交指令序列。

9.10 避免被驱逐

如果读者常与其他人合作 (很多人如此)，那针对很多缓存性能差的问题, 最简单的办法就是考虑到其他部分的代码。如果工作在一台多核机器上 (显然是，除非能回到过去)，那很可能所有的进程都在分享、争夺机器上的缓存。毫无疑问，你的代码会被从缓存中驱逐。数据也是如此。要降低代码和数据被驱逐的几率，尽量保持代码和数据处于小规模，如果可以，还要处理突发情况。

这个建议其实很简单。短小的代码不仅不易被驱逐，并且，如果能迅速完成，那在被覆盖前就有机会完成适当的工作量。有些缓存架构无法判断缓存中的元素最近是否被用过，所以他们依靠这些元素的添加时间，作为优先驱逐的标准。特别是，有些 Intel 的 CPU 会因为 L3 需要驱逐，就去驱逐 L1 和 L2 缓存行，但 L3 并不能完全访问 LRU 信息。Intel 的 CPU 还有些其他魔法，能减少这种情况，但确实仍会发生。

为此，尽量想办法向编译器保证在处理的数据是对齐的，是位于 4、8、16 的倍数的数组中。这样编译器就无需添加预处理和后处理代码来处理不对齐、不规则大小的数组。一个数组中多出 3 个无用元素，那将其作为长度为 `N * 4` 的数组来处理可能会更好。

9.11 自动矢量化

自动矢量化能让应用程序运行得更快，只需启用它组织代码，编译器就可以做出安全假设，并将指令从标量变为矢量。

有许多常规的例子可以被彻底矢量化。如代码 9.6，常规到大多数编译器打开优化时都会对其矢量化。问题是这段代码没什么保证，所以即使处理数据的速度相当快，它在指令缓存中占用的空间，也多于必要。

```
void Amplify( float *a, float mult, int count)
{
   for( int i = 0; i < count; ++i) {
      a[i] *= mult;
   }
}
```

Listing 9.6: 常规的放大实现

如果做些简单保证（如对齐指针），并提供给编译器一些元素数量的保证，就能够减少分发的汇编指令。虽然从单个案例的尺度看没什么帮助，但对于大型代码库，要解码的指令数量削减了，因而指令缓存效果也会增强。代码 9.7 在孤立测试中并不快，但最终可执行文件更小，生成的代码不到原先的一半。这属于微型基准测试的一个问题，它们没法每次都展示系统是如何一起工作、相互斗争的。现实的测试中，修正指针对齐能极大提高性能。而在小型测试平台中，内存吞吐量通常是唯一瓶颈。

```
typedef float f16 __attribute__ (( aligned (16)));
void Amplify( f16 *a, float mult , int count)
{
   count &= -4;
   for( int i = 0; i < count; ++i) {
      a[i] *= mult;
   }
}
```

Listing 9.7: 带对齐的放大实现

需要注意，确保循环足够简单，并且总能执行其循环体。如果循环必须基于数据中断，那就不能确保完成所有元素的处理，意味着它必须逐个处理每个元素。代码 9.8 中，引入基于数据的中断，使得该函数从一个快速并行的 SIMD 操作的自动矢量循环，变成了一个单步循环。这里要注意，分支本身并不会导致矢量化崩溃，基于数据退出循环才会。如，代码9.9，分支可以变为其他操作。还有一种情况，调用函数往往也会破坏矢量化，因为通常无法保证副作用。而如果该函数是`constexpr`，那它就更有可能被放进循环体中，且不会破坏矢量化。某些编译器中，一些数学函数能够以矢量形式呈现，如 `minabs`、`sqrt`、`tan`、`pow` 等。可以试着找出当前编译器可以矢量化的内容。某种程度上，手动去写要做的一些列操作往往也有帮助，因为尝试缩短 C++ 代码的话，可能会导致编译器允许做的事情出现轻微歧义。但要特别注意，确保完全展开。如果只写入部分输出流，就没法输出整个 SIMD

数据，所以要写输出变量，哪怕读取只是为了去写输出。

```
typedef float f16 __attribute__ (( aligned (16)));
void Amplify( f16 *a, float mult , int count)
{
   count &= -4;
   for( int i = 0; i < count; ++i) {
      if( a[i] < 0 )
         break;
      a[i] *= mult;
   }
}
```

Listing 9.8: 用 break 跳出循环，破坏矢量化

```
typedef float f16 __attribute__ (( aligned (16)));
void Amplify( f16 *a, float mult , int count)
{
   count &= -4;
   for( int i = 0; i < count; ++i) {
      f16 val = a[i] * mult;
      if( val > 0 )
         a[i] = val;
      else
         a[i] = 0;
   }
}
```

Listing 9.9: 矢量化 if

别名也会影响自动矢量化，指针可以重叠时，同一 SIMD 寄存器的不同成员间，就可能存在依赖关系。考虑代码 9.10，该函数的第一个版本是每个成员会加它后面的一个相邻成员。这个函数没什么意义，只是用来举例。它成对累加，直到最后一个 `float`。因此，无法简单地矢量化。第二个函数，虽然同样没有意义，但其步长够大，自动矢量化就能找到方法，计算每步的多个值。

```
void CombineNext( float *a, int count )
{
   for( int i = 0; i < count - 1; ++i ) {
      a[i] += a[i+1];
   }
}
void CombineFours( float *a, int count )
{
   for( int i = 0; i < count - 4; ++i ) {
      a[i] += a[i+4];
   }
}
```

Listing 9.10: 矢量化 if

不同的编译器会根据写代码的方式，管理不同数量的矢量。一般来说，代码越简单，编译器就越有可能优化它。

未来十年，编译器会越来越好。Clang 已经比 GCC 尝试解开更多循环，可能也会出现许多检测、优化常规代码的新方法。写作本书时，Matt Godbolt 实现的在线 Compiler Explorer[5] 就是个很好的工具。能够看到代码如何被编译成汇编，也就能看到什么可以，也会被矢量化、优化、重排、以其他方式转变成机器可读形式。记住，汇编指令的数量，并不是衡量代码快慢的好标准，SIMD 操作并非在所有情况下都更快。只要不是通过摸着下巴想想这段指令是不是很牛皮[6]来衡量代码的运行情况，应该就没事了。

[5] https://godbolt.org

[6] 捋胡须、咬笔头也不行！————某位读者提议。

第十章　维护与复用

最初推广面向对象设计时，据说它比传统的过程式方法更易修改、扩展。虽然实践证明并非如此，但看到其他编程范式时，面向对象的开发者常常引用这一点。不管他们的专业水平如何，但凡涉及到大型项目，面向对象的程序员很可能会将可扩展性、封装性作为其优势。

经验丰富但较为客观的开发者已经承认，甚至写到，面向对象的 C++ 并非高度适用于有大量依赖关系的大项目，但只要严格遵循诸如 Large-scale C++ 一书中的准则，就可以 [10]。如果无法立即看到面向数据开发范式，在维护、发展方面优势，本章中会阐明，为什么它比用对象工作更容易。

10.1　宇宙层级

不管怎么称呼：宇宙基类、万恶之源、Gotcha #97，还是 `CObject`。大型 C++ 项目中，所有东西都衍生自一个基类，近乎是个普遍的失败点。C++ 原生不支持内省或鸭子类型，所以很难有效利用`CObjects`。如果用数据库驱动，那就能在一开始就巧妙地引入宇宙基类，作为所有其他组件描述出的实体，从而保证所有东西都是实体。尽管基于组件的引擎经常会用一个 `EntityID` 作为所有者，但并非所有的引擎都有必要。也不是所有引擎都只有一个所有者。标准化数据库时，就会发现有不同的实体类型的集合。在前面关卡文件的例子中，可以看到一开始的对象，如何一步步变成`MeshID`、`TextureID`、`RoomID`、`PickupID`。甚至出现了 `DoorID`，当然也有必要。如果把所有这些 `ID` 集中到 `EntityID` 中，系统或许仍能正常工作，但却没什么必要。很多实体系统的确这么做了。但跟大多数运动一样，第一时间尝试找回平衡，动作幅度往往太大。可以在数据库行业提供的数据标准化的实例中找到平衡点。

10.2 调试

导致错误的主要原因：是变换带来的意外的副作用，或是一个条件没有返回正确值的意外边界。面向对象编程中，可以表现在许多方面：从解引用空值引起的异常，到忽略玩家的交互 (游戏逻辑并不知道自己需要互动)。

记着系统状态，玩电脑以搞清楚状况，即：程序员绝对要在这个领域里，才能完成真正的工作。现实可能远没那么惊心动魄。实际情况更类似于对于这种情况的恐惧，只有代码复杂到了近乎致命时，程序员才需要进到这个领域。

10.2.1 生命周期

最常见的导致解引用 `null` 的原因之一，是控制生命周期与使用它的对象是分离的。如，假设有个敌人会死的游戏，每当敌人被删除时，那就必须小心翼翼地更新所有使用他们的对象；否则，最终就可能解引用到无效内存，进而可能解引用空指针。该类已经被破坏了。面向数据开发则倾向于避免这类情况。因为实体存在于数组中，就表示需要处理，如果表中只留下实体的一部分，没有完全删除它。那这又另一个不同的 bug，但不是崩溃，而且更容易发现、解决。因为它只要确保在实体被销毁时，所有包含它的表也会对应销毁其中的元素。

10.2.2 远离指针

在寻找面向数据编程问题的解决方案时，我们常发现指针不是必需的，而且常常导致解决方案更难扩展。在可能出现空值的地方使用指针，意味着每个指针不仅表示被指向的对象的值，也意味着它隐含着布尔属性：确定实例是否存在。去除这个不必要的额外功能可以消除错误，节省时间，并降低复杂度。

10.2.3 不良状态

bug 与处于错误状态有很大关系。因此，调试就变成了找出游戏是如何进入到当前错误状态的。

要给一个变量赋值时，就会破坏历史。以代码 10.1 为例。一个函数中只有一个返回语句，这种理想状态会导致这种错误，并且比预想的次数要多。

有一个以上的返回点也有自己的问题。重点是，一旦走到了函数末尾，就很难搞清楚导致它验证失败的原因。甚至没法对问题点下断点。递归的例子就更危险，因为有一整棵树的对象，不管值是多少，都会在返回前对所有对象执行递归，并且也没法下断点。

```
void CombineNext( float *a, int count )
{
   for( int i = 0; i < count - 1; ++i ) {
      a[i] += a[i+1];
   }
}
void CombineFours( float *a, int count )
{
   for( int i = 0; i < count - 4; ++i ) {
      a[i] += a[i+4];
   }
}
```

Listing 10.1: 修改状态可能会隐藏历史

封装状态会隐藏内部变化。很快就会引入大量调试日志。相较于隐藏，面向数据则倾向以简单形式保留数据。有可能维护数据超过预期时间，然而会高度简化对变换的检查。假设有个将将可以工作的变换，在某些奇怪的情况下会出错，那通过添加断言，或者不删除输入数据，这类简单做法能一定程度减少辛劳和猜测，进而更好地理解问题并重现。如果大部分变换保持单向，即，从一个来源获取，并生成、更新另一个来源，那即使多次运行代码，仍然会产生与第一次相同的结果。变换是幂等的。这个属性可以确保找到错误症状，然后倒回去，追溯原因，而不必尝试重建初始状态。

要保持代码的可操作性，一种办法是用赋值实现变换。如果用多个变换操作，但都指向预定的连接点，节奏就由自己掌握，也可以避免回溯，回顾导致最终状态的原因。如果条件是一个条件表，只要在检查完成前保留输入，就能进入任何实时系统，并检查如何进入当前状态。仅此一点，就可以将 debug 时间缩减到最低限度。

10.3 复用性

面向对象开发人员常提到：面向数据开发中看起来缺乏复用性。这种想法在于，不能再次使用已经写好的代码库，因为设计部分体现在了实现中。当然，一旦开始针对某个软件项目的特殊功能优化代码，确实会出现无法复用的情况。开发面向数据的项目时，会假定无法复用源代码的情况会很严重，但实际并非如此。仔细想想复用性的真正含义，就能发现真相。

复用性从根本上说，并非指复用源文件或库。复用性是维持信息投资的能力，如创造更多表达，用它们传达意图，例如 STL，或其他的结构代码库。作为复用操作序列的案例，它们是有开发知识产权的实体的知识财富，非常接近专利的根基。后者中，表达往往是偶然发现，而非真正发明的。

版权法让人很难看出哪些源码有复用价值。它讨论源码，而非源码代表的知识产权。仅仅一个想法是无法版权化的。借由保持这种立场，版权方就能通过这种微弱联系维持保留信息的权利。复用性源自对它所存储的媒介中所含信息的认识。在我们的例子里，它通常以源码形式存储，但信息不是源码。通过面向对象开发，源码可以修改（适配器模式）到任何希望投产的项目。但是，源码也不是信息。现在以及将来能在数据上执行的任务的顺序及其存在本身，才是信息。可以理解为，一个编程技术所产生的任何复用性，都归结于其输入和输出的可变性。将一组时间上耦合的任务调整到新的使用框架中的难易程度，就是评判复用性的标准。

面向对象开发中，可以通过修改执行该任务的类，来应用代码中固有的信息 (也可以包装 (`wrapper`)，或使用代理 (`agent`))。面向数据开发中，则直接复制函数和架构，并在应用面向数据的变换中包含的信息时，围绕输入、输出数据结构进行变换。

尽管从外表看，面向数据的代码复用性不高。但实际上，它以更简单的形式维护同样数量的信息，所以更具复用性。因为，它不像面向对象编程那样，携带相关数据、相关函数的包袱；也不像过程式编程那样，出于标准化倾向，会产生复杂的变换来生成输入，提取输出。

由于数据间的接口有一套更严格的规则，在面向对象编程中通常不能使用鸭子类型。但可以用模板实现，而且效果很好。只要保持统一命名风格，就可以把或许不明显的可复用代码，变成简单的策略，或一连串可以应用于任何类型数据或结构的变换。

面向对象的 C++ 的复用是信息和架构的混合。从围绕面向数据变换的角度开发，架构似乎只是很多花哨的代码。唯一值得保存的好架构是数据流和变换的实现。少有的情况下，可以复用面向对象的模块，毕竟面向对象项目之间的接口天生就会带来难度。

最可复用的面向对象代码是作为代理接口，出现在更复杂的系统中。最好的例子是 *stdio.h* 中的 `FILE` 类型，这种方法让一切都更易处理，高度可复用，并且完全封装。它用于代理进入任何平台和操作系统都需要打开、访问、写入、读出系统上的文件。

10.4 可复用的函数

除了在将所有数据保持在简单的线性排列时，可以自由扩展的优势，还有种隐含：意外发现可重复使用的解决方案。由于数据更加严格地格式化了，因此在适合的时候，几乎可以作为某种鸭子类型。如果数据适合一个变换，那么该变换就理应能应用于它。有人会说，仅仅因为类型匹配，并不表示函数就会产生预期结果。但这一点可以通过通过避免使用不理解的代码来规避。并且某些情况下，只要知道签名，就能理解变换。一个极端的例子，我们可以纯粹根据参数来理解相当数量的 `Haskell` 函数。最后，由于代码变得更易理解，所以决定复用一个变换之前，了解其功能花费的时间更少。

数据每次都以相同方式建立，再变换，并且总是保存在相同类型的容器中。所以很可能存在多种与设计无关的优化，能应用于代码中的许多位置。通用的排序、计数、查找、空间感知系统，可以在不调用 OOP 适配器，不实现接口的情况下，附加到新数据上。策略 *(Strategies)* 得以执行。这也是为什么在数据库中，可以有通用查询优化。以这种形式开发，就可以在更多的项目中引入优化。

10.5 单元测试

单元测试在开发游戏时作用重大。但面向对象范式，会使程序员把代码看作是对象的表示，而非数据的变换，所以很难看到能测试什么。将不相关的概念链接到同一个对象中，并且在测试前需要设置复杂的状态，导致单元测试在游戏开发中有着先天劣势。因而简单的测试在面向对象编程中也难以编写。同时，由于在游戏世界中的对象，以某种不显著的方式变换为实体，测试变得更加复杂。除非开发者已经相当熟悉，否则很难写好单元测试。但单元测试的作用之一，是让尚未完全摸透系统的人可以修改，且不至搞得一团糟。

重构中常会用到单元测试，把游戏或引擎从一种代码、数据布局变成另一种，应对新的变化。重构，通常是因为数据的格式不对。如果此时再去标准化数据，本身就更难，反而更可能让数据处于未配置的形式。有时即便数据已经标准化了也不够。例如，当游戏设计发生变化，足以使原来的数据分析不再正确，或至少变得低效甚至无效。

在面向数据技术中，单元测试相对简单。因为精力已经集中在变换上了。如果尚未处于游戏开发过程中，那生成测试数据的表就是开发的一部

分。保留一些表，用作单元测试也很容易。用单元测试帮助指导代码，相当于部分遵循了测试驱动开发技术，而它已经被证明是一种产生高效和清晰代码的好方法。

记住，在做面向数据开发时，游戏完全是由**有状态的数据**和**无状态的变换**驱动的。为变换实现单元测试非常简单。甚至不需要框架，只要有输入/输出表，然后再用一个比较函数，检查变换的结果是否正确即可。

10.6 重构

重构过程中，知道是否因为代码改变带来破坏，始终都很重要。引入简单的单元测试就成功了一半。面向数据开发的另一个优点，它每次都能剥离出不必要的元素。或许我们会发现，重构更多是改变变换顺序，而非改变数据的表现形式。重构通常会涉及一些新的数据表示，但只要在构建结构时考虑到标准化，需求就不会那么多。真正需要时，从一种模式变换到另一种模式的工具可以一次实现，多次使用。

使用标准化的数据时，开发者可能就会意识到：一开始就重构了这么多，是因为意义嵌入到了代码中，数据放进了有名字的对象中，方法是针对对象执行，而非为了变换数据。

第十一章　问题出在哪？

面向对象设计有什么问题？它有哪些危害？

多年来，游戏开发者已经陷入一种 C++ 风格，它如此不讨好硬件，以至于对比托管型语言也没有快多少。游戏开发中，C++ 的使用模式与 PlayStation 3 和 Xbox 360 这代硬件的适配性惊人地差，也难怪解释型语言在正常使用下只慢了 50%，在其专业领域有时还更快 [1]。为什么这种奇怪的语言风格，会嵌入到 C++ 游戏开发者的头脑中？流行的编写游戏的方式，为什么会变成最差的利用目标机器的方式？本质上，游戏开发风格中面向对象的 C++，危害在哪里？

其中一些，来自对面向对象的最初解释。很多游戏开发者认为，面向对象表示必须将在意的所有实例，都变成对象的实例映射到代码中。这种开发形式可以解释为面向实例开发，它把单一的独特实体放在整个程序之先。这样更易发现一些签注的问题。个例的性能很难说是差；对象方法难以准确计时，甚至可以说近乎不可能。当开发实践鼓励个体元素高于程序整体，就会消耗心智，因为开发者必须从角色角度，而非价值语义的角度，用它们的隐藏状态，考虑所有操作。

另一个问题：语言设计者并没有忽视性能，但可能是在孤立的情况下测试的。可能由于 C++ 的实际用途与库提供者的期望有大不同，也可能库的提供者按照内部指标工作，而不是确保理解客户。作者认为，在开发库或用于 C++ 的模板时，代码的性能不应只是一个可调整的选项，它应该是默认的。如果可以调整性能，就会用功能换取理解和性能。对于游戏开发者，这笔交易并不划算，但已经被广泛接受了；毕竟通用语言的好处非常诱人。

[1] http://keithlea.com/javabench/ 讲了服务器 JVM 比 C++ 快的故事。有些观点反对这个结果，也有一些观点支持它。请自行理解。

11.1 危害

主张：虚函数开销并不高，但若频繁调用，开销会累积。

或——千里之堤，溃于蚁穴 [2]

简单的测试中，虚拟调用的开销可以忽略不计。对比虚拟调用，额外的开销近乎微不足道，除了解引用的开销和虚表指针占的额外空间外，很可能不再有其他明显的副作用。在获得某个特定实例中要调用的函数指针之前，额外的解引用也只增加了一点点负担，但让我们仔细看看到底发生了什么。

派生自基类，有虚方法的类都具备一定的结构。在类中添加虚方法，会立即在可执行文件中添加一个虚表。虚表指针会隐式地成为该类的第一个数据成员。这点没办法。语言规范中，允许编译器决定类的数据布局。为了实现虚方法，它们可以添加隐藏成员，在幕后生成新的函数指针数组。虽然也可以通过其他方式实现，但似乎大多数编译器都是通过虚表。重点是，虚拟调用不是操作系统级别的概念，就 CPU 而言，它们不存在，只是 C++ 的一个实现细节。

调用类的虚方法时，必须得知道会运行什么代码，通常得知道要访问虚表的哪一条。因此，要读取第一个数据成员，以便访问并调用正确的虚表。先从类的地址加载到寄存器，再加上偏移。每次“没什么开销”的虚方法调用，都是一次查表，所以在编译后的代码中，所有虚拟调用实际上都在解引用函数指针数组。偏移量用在这里，它是函数指针数组的偏移量。得到真正的函数指针地址后，才能开始解码指令。有些方法可以不调用虚表，特别是 C++11 中，`final` 关键字有了些进步，能帮那些无法重写的类知道，如果它们调用自己，就可以直接调用函数。但它对多态调用、不知道具体类型的接口访问都没有帮助（见代码 11.1）。不过偶尔在一些习惯性做法中也能发挥作用，如私有实现（`pImpl`），以及奇怪的常见的模板模式。

```cpp
#include <stdio.h>
class B {
public:
   B() {}
   virtual ~B() {}
   virtual
   void Call () { printf( "Base\n" ); }
   void LocalCall () {
      Call ();
   }
};
class D final : public B {
```

[2] 译注：原文为 death by a thousand paper cuts

```
public:
    D() {}
    ~D() {}
    virtual void Call () { printf( "Derived\n" ); }
    void LocalCall () {
        Call ();
    }
};
B *pb;
D *pd;
int main () {
    D *d = new D;
    pb = pd = d;

    // prints "Derived" via virtual call
    pb ->LocalCall ();

    // prints "Derived" via direct call
    pd ->LocalCall ();
}
```

Listing 11.1: 简单的派生类

多重继承会稍复杂些，但基本仍然是虚表，只不过每个函数会定义要引用哪个类的虚表。

现在，我们来计算一下这个调用涉及的实际操作：先是 load，然后 add，再一次 load，最后是 branch。几乎所有的程序员看来，为了运行时多态，这都算不上什么沉痛代价。每次调用有四个操作，所以可以把所有实体扔进一个数组，然后循环更新、渲染、收集碰撞状态、生成声音效果。乍一看似乎是不错的权衡，但仅限这些特定指令开销很低的时候。

四条指令中，有两条是加载，开销似乎不应该太大。但除非命中附近的缓存，否则加载就需要很长时间，解码指令也需要时间。加法开销很低 [3]，它会修改寄存器的值，寻址正确的函数指针。但是分支开销就不一定了，在第二次加载完成前，不知道下一步会去哪。可能会导致指令缓存未命中。总而言之，由于虚拟调用而浪费了大把时间，在任何显而易见的大规模游戏中都很常见。在这一大块时间里，光是浮点单元就能完成大量点积、方根运算。最好的情况，虚表指针已经在内存中，对象类型与上次相同，所以函数指针地址也相同，因此函数指针也在缓存中。这样的话，分支很可能不会停顿，因为指令也可能还在缓存中。但对于各种类型的数据，并不是每次都能遇到“最佳情况”。

考虑另一种情况：函数末尾，需要要返回值，然后调用另一个函数。指令顺序相当清楚，在 CPU 看来近乎于一条直线。获得指令方面没有任何偏差，只是沿着每个函数，依次跟随程序计数器。于是可能提前很长时间猜到

[3] 在大多数平台上，在访问内存前添加到寄存器没有开销

将要调用的新函数地址，因为它们都不依赖于数据。即使有大量函数调用，在编译时也可推断，于是很容易被预取 (prefetch)、预翻译 (precompile)。

C++ 的实现不喜欢我们迭代对象的方式。迭代一组异质对象的标准方法，就是字面意思上：拿到迭代器，依次调用每个对象的虚函数。常规的游戏代码中，会涉及到为每个对象加载虚表指针。在加载缓存行的时候会导致等待，而且难以轻易避免。加载到虚表指针后，就可以用常数偏移量（虚方法的索引）找到目标函数指针。然而，鉴于游戏中常见的虚函数的大小，该表不会出现在缓存中。自然，另一次等待加载少不了。等这次加载完成，我们只能寄希望于，该对象实际上与前一个元素的类型相同；否则，又不得不再等待一些加载指令。

即使没有加载，数据加载前也不知道哪个函数会被调用。意味着需要依赖缓存行的信息，才能确定正在解码的是正确的指令。

游戏中的虚函数之所以大，是因为开发者已经被灌输了这样的观念：只要不在紧密循环中使用虚函数就好了。但无一例外，导致它们被用于更多其他架构考量中去：如对象类型的层次结构，或树状问题解决系统（如寻路，或行为树）中的 `helper` 类。

我们再重温一遍：许多开发者现在认为，使用虚拟的最好方法是把繁重的工作负载放到虚方法的主体中，这样可以降低虚拟调用机制的开销[4]。然而，这样做，几乎一定会导致，指令和数据缓存的很大一部分会被每次 `update()` 调用驱逐；并且大多数分支预测器槽也可能变脏，无法为下一次 `update()` 带来任何好处。假设虚拟调用不会累积，因为是在高级代码上调用的。很好，直到它变成通用编程风格，开发人员不再考虑应用程序如何被影响着，最终导致每秒数百万次调用。所有这些低效调用都会累加并影响硬件，但它们几乎从未出现在任何性能分析 (profile) 中。问题不在于它存在与否，而在于它在机器的整个处理过程中稀疏地分散在各处。总是会在代码调用的某个地方出现。

Carlos Bueno 的 *Mature Optimization Handbook*[2] 一书中提到，盲目追随低垂的果实，很容易错过拖慢速度的真正原因。这就证明先提出一个假设是有用的，当确定没有得到预期回报时，就能更快回溯、重整。例如 Facebook，他们追踪了导致驱逐的原因，并优化了这些功能，不是为了速度，而是为了尽可能消除缓存中驱逐其他数据的几率。

C++ 中，虚表中的函数指针按类存储。还有一种方法是，为每个函数

[4]这与任务系统有些类似，类似于想要降低设置、拆解任务的成本。

设置一个虚表，并根据调用类切换函数指针。实践中运行良好，也的确省了一些开销，在一组对象的一次迭代中，虚表对所有的调用都是一样的。然而，C++ 设计允许运行时链接到其他库，库中的新类可能继承自现有代码库。该设计必须允许能为运行时链接的类添加新的虚方法，并确保可以从原始运行代码中调用它们。如果 C++ 采用面向函数的虚表，那么每当链接一个新库，不管是静态编译的链接时，还是在动态链接库的运行时，语言都必须在运行时修补虚表。于是，为每个类使用一个虚表提供同样的功能，但避免了在链接时、运行时修改虚表；这些表是由类导向的，根据语言设计，它们在链接时不可变。

结合虚表的组织，游戏倾向的方法调用顺序，即使以高度可预测的方式执行列表，缓存未命中问题也很常见。不仅是因为类的实现，任何时候数据都是指令运行的决定性因素。游戏通常会实现脚本语言，它们通常被解释并运行于虚拟机上。不管虚拟机（或 JIT 编译器）如何实现，总有某种数据控制着指令调用。这就会导致分支预测错误。这也是为什么，通常解释型语言比较慢：在字节码解释器中，它们要么基于加载的数据执行代码，要么即时编译代码，虽然能生成更快的代码，但也带来了自身的其他问题。

开发者在不使用 C++ 中内置的虚函数、虚表、`this` 指针的情况下，实现面向对象的框架，除非按函数使用虚表，否则也没法提高缓存命中率。但即便特别小心，用开发者的访问模式进行面向对象编程，即在异质对象的数组上调用单一虚函数，仍会发生解码相同指令、缓存未命中的情况。也就是说，他们最希望的是每次虚拟调用减少一个基于数据的 CPU 状态变化。但却留下了两个预测错误的机会。

那么，面对所有这些明显的低效，为什么游戏开发者仍坚持使用面向对象编码实践呢？游戏开发者们常常作为计算机软件开发的前沿被提及，为什么他们不选择全盘脱离这个问题，完全停止使用面向对象开发实践呢？

11.2 映射问题

> 主张：对象为将现实世界的问题描述为最终代码层面的解决方案，能更好地映射。

游戏编程中，面向对象设计，始于从实体的角度考虑。设计中，为每个实体都赋予一个类，如 `boat`，`player`，`bullet`，`score`。每个对象都保持自己的状态，通过方法 (method) 与其他对象通信，并封装。因此，当某个实体的

实现有变化，使用它或为它提供效用的其他对象则不需要改变。开发者们喜欢抽象，回看历史，他们不单单要为一个目标平台编写游戏，一般有两个以上。而过去还只是在主机制造商之间。现在，开发者需要兼顾 $Windows^{TM}$、主机，以及移动端。过去的抽象主要针对硬件访问，少量玩法抽象。但随着游戏开发行业日趋成熟，物理、人工智能、玩家控制等领域也都有了常见的抽象形式。这些常见的抽象衍生出第三方库，其中许多也使用面向对象的设计。对于库，通常用代理与游戏交互。这些代理对象包含他们自己的，或隐藏，或公开的状态数据，实现了一些功能。通过这些功能，可以其所属系统的约束条件下操作它们。

游戏设计的灵感对象（如 `boat`，`player`，`bullet`）维护着代理，并通过它们获悉世界中发生了什么。玩家通过面向对象 API，与物理、输入、动画以及其他实体交互，隐藏了很多完成任务的实际的细节。

面向对象设计中的实体，是数据的容器，用于存放操作这些数据的函数。不要把这些实体与实体系统中的“实体”混为一谈，因为面向对象设计的实体，在其生命周期内是不可变的类。C++ 中没有原地重新构造类这一特性，所以面向对象的实体在其生命周期中不会改变类。不出所料，没有合适的工具的话，优秀的工程师就会尝试权宜之计。开发者不会在运行时改变对象的类型，但如果游戏实体需要，他们会创建新的并销毁旧的。但通常，由于语言中没有这个功能，即便有意义，也没法充分利用。

例如，在 FPS (第一人称射击游戏) 中，声明一个对象，表示玩家动画模型，玩家死亡时，用一份克隆表示尸体的布娃娃 (rag-doll)。可以隐藏动画对象，并移至下一个重生点，而尸体对象有不同的功能和数据，会留在死亡地点，好让玩家看到。于是，一旦玩家死亡，尸体对象就替代动画模型，因而需要定义复制构造函数。玩家重生时，动画模型再次可见，如果愿意，玩家还可以去看看尸体。这样效果不错，但如果不生成不同类型的克隆，而是将玩家类转换成死去的布娃娃，反倒没什么必要。而且还有其他潜在问题：克隆过程可能出错，导致其他问题；如果玩家死亡，但又能被复活，那么就得能将布娃娃变换回动画模型。这就更复杂了。

再举一个 AI 的例子。大多数游戏中运行的 AI 的有限状态机和行为树，都会为所有潜在状态维护必要数据。假设例中的 AI 有三种状态，`Idle`；`Making-a-stand`；`Fleeing-in-terror`，它就有所有状态所需的数据。假设 `Making-a-stand` 状态有个恐惧值累积，AI 会战斗，直到害怕到无法继续，而 `Fleeing-in-terror` 状态有个定时器，AI 会逃跑，但只有一段

时间，那么 `Idle` 状态也会有这两个非必须的属性。这个小例子中，AI 类有三个数据项，`state`；`how-scared`；`flee-time`，但只有一个用于所有状态。如果 AI 在状态转换时可以改变类型，那甚至不需要状态成员，虚表指针能承担这个功能。AI 只在适当状态下，才为每个状态的成员分配空间。C++ 中，能做到的极限就是手动改变虚表指针来伪造它，虽然很危险，但总归是能做到；或为每种转换实现复制构造函数。

除了类型不可变，面向对象开发还有个哲学问题。人是如何在现实生活中感知物体的？每一次观察总会有个背景。一张简陋的桌子，你看着它，可能会看到四条桌腿，木制，适度抛光。以上，还能看到它是棕色的，有些反光。能看到质地，颜色，并且觉得它是某种确定的颜色。然而，如果学习过艺术，可能就知道，我们看到的不是实际存在的东西。没什么纯色。盯着桌子，也没法看到它准确的形状，只能推断。如果是通过进入视网膜的平均光色，推断它是棕色，那如果关了灯，还是不是？如果光线太强，只看到抛光表面强烈的反射，又如何？如果闭上一只眼睛，从其中一条长边看着长方形桌面，看到的也不是直角，而是梯形。我们看到物体时，会自动调整、分类。会应用我们的成见，锁定，帮助做出推断。这就是为什么面向对象的开发有如此吸引力。然而，人容易理解的东西，对计算机可能并没有那么友好。将对象当作游戏实体，我们觉得是一个整体。但计算机没有对象的概念，只把对象看作是组织得很差的数据和随机调用的功能。

再以桌子为例，假设桌腿高约 90cm，即标准桌高。如果只有 30cm，那可能是张茶几。很矮，但仍可以扔杂志、放杯子。但要是桌腿只有 3cm，那就不是张桌子，只是块粘了几根棍子的大木头。不同尺寸的同一物品，轻易地就划分为三个不同类别。桌子、茶几、木头。但什么时候这块木头可以称作茶几？高度 15cm　30cm 的时候吗？跟沙子的问题一样，从沙粒到一堆沙子？多少粒是一堆，多少又算一个沙丘？答案必然是：无解。这样有助于理解计算机的思维方式。它不知道我们人类分类的具体区别，毕竟某些程度上，我们也不知道。

对象的类，难以用它是什么来定义，最好用它做什么来定义。这就是为什么鸭子类型化的方法很强大。我们也意识到，如果根据” 做什么” 能更好定义类型，那从根本了解多态类型时，就会发现，它只是在” 做什么” 方面有多态性。C++ 中，我们知道有虚函数的类，能当作运行时的多态实例调用；但如果它没有这些，就不清楚了，不过也没必要第一时间搞清楚。这就是多重继承发挥作用的点。多重继承，只是意味对象能响应某些信号。它在

声明中了签下了一份协议，能够执行一些多态函数。如果多态只是对象履行协议的能力，那就不需要每次都用虚拟调用处理。还有其他方法能让代码能根据不同对象，表现出不同效果。

大多数游戏引擎中，面向对象的方法会导致大量对象处于非常深的层次结构中。常见的实体的继承链可能是这样：PlayerEntity → CharacterEntity → MovingEntity → PhysicalEntity → Entity → Serialisable → ReferenceCounted → Base.

这些深层结构，必然导致使用虚方法时多次间接调用。涉及到交叉代码也产生诸多痛苦，即，与不相干的代码、层级相互影响。假设有个游戏，人物在场景移动。这个场景中，会有人物、世界，还有些粒子、灯光，有静态，有动态。所有这些，要么直接被渲染，要么渲染要用到。传统的方法，或是用多重继承, 或是确保每个实体的继承链上有 `Renderable` 基类。但会发出声响的实体呢？是否也要添加一个发声类？可序列化的实体，或可显式管理的实体呢？常见到需要一个独立内存管理器（如，粒子）的，或只需选择性渲染的（如，垃圾、花、远处的草），又该如何处理？这一点已经解决了无数次：将所有最常见功能，放到游戏的核心基类中。当然也有例外：如关卡有动画时；玩家角色处于开场或死亡画面时；又或是 Boss（特殊到值得多写点代码）。只有不使用多重继承时，这些 hack 才必要，不过一旦用上了多重继承，一张网就悄然编织，最终以虚拟继承及其带来的复杂状态告终。妥协的结果，往往总是成为某种形式的宇宙基类的相反模式。

面向对象开发擅长在源代码中，面向人类表述问题，但不擅长表述面向机器的解决方案。它难以提供创建最佳解决方案的框架，所以问题仍然存在：为什么游戏开发者仍然使用面向对象技术开发游戏？可能不是为了更好的设计，而是为了更容易修改代码。众所周知，开发者会随着设计更迭，不断修改代码，直到上线。面向对象开发是否能让修改和维护更简单、更安全？

11.3 内部化状态

> 主张：封装让代码更易复用。不影响使用的前提下，更易修改实现。维护和重构变得简单、快速、安全。

封装本质上是为用户提供一份协议，而不是提供一份原始实现。理论上，封装地好的面向对象代码，能避免因改变对象操作和数据的方式带来损害。如果所有使用该对象的代码都遵守协议，且不通过访问函数直接使用任

何数据成员，那无论对该类的协议内部实现怎样改变，都不会引入新 bug。理论上，只要不修改协议，只是扩展，就能任意改变实现。这就是开闭原则 (Open/Closed Principle)。类应该对扩展是开放的，但对修改是封闭的。

协议是为了保证复杂系统能工作。但在实践中，只有单元测试才能保证这点。

有时，程序员会不知不觉地依赖对象实现的隐藏特性。比如依赖的对象有个 bug，但正好符合使用情况。如果该 bug 被修复，那使用该对象的代码就不像预期般工作了。虽然被保留了协议使用，但并没能帮助另一份代码在不同的版本中正常工作。相反，还带来误导，一份错误地希望，即返回值不会改变。有时候不一定得是 bug。对象内部的时间耦合（或意外、未记录的特性）在后来的版本中消失，也会在未能报错的情况下带来损害。

如一个按顺序维护内部列表的实现，有个用例恰好依赖它（用户用例中不可预见的 bug，不是有意依赖）。维护者推送了一个*优化性能更新*，用户只得到一堆新 bug，他们很可能不会觉得是自己的问题，而是性能更新导致的。

再举一例：有个项目管理器，维护一份按名称排序的项目列表。有个函数返回所有符合过滤器的项目类型，调用者可以迭代返回列表，直到找到想要的项目。为了加速，可以在发现一个名字比要找的项目靠后时，提前退出，或也可以对返回的列表做二分查找。两种情况下，如果内部表示法改变，不再按名称排序，那应用代码就不工作了。如果内部表示变成了按哈希值排序，那提前退出、二分查找的实现都被破坏了。

在许多链表实现中，可以决定是否存储列表长度。选择存储一个计数成员会导致多线程访问变慢；但如果不存储，则会让查询链表长度变成 $O(n)$ 的操作。只想知道列表是否为空的话，如果对象协议只有一个 `get_count()` 函数，就没法确定是检查计数是否大于 0，还是检查 `begin()` 和 `end()` 是否相同，哪个更高效了。这也是个表明协议提供的信息太少了的例子。

封装看起来只是一种隐藏 bug 的方法，并且会导致程序员做假设。有句老话，除非能接触到源码，否则封装会阻止得到是或否的答案。如果有源码，且需要查看它以找出问题所在，那么封装做的只是在工作上又加了一层，并没什么额外的用处。

11.4　面向实例开发

主张：把每个对象都变成实例，能更容易地思考对象的责任、生命周期、及其在世界中的归属。

实例思维的第一个问题：一个条目只做一件事，这种思路必然导致性能不佳。

第二个，也更普遍的问题：让人抽象思考实例，以完整对象为思考的单元，只会让算法变低效。甚至对程序员用户，也隐藏项目内部表示，就会导致在不同对象的思维模式间来回跳转。比如有个条目，需要修改另一个对象，但发现当前环境下无法做到，所以不得不向其所在容器发送消息，以回答另一个实体相关的问题。然而，程序常常在这些线路上忽略了数据要求，在查询、响应中发送额外信息，不仅会放出不必要的权限，还会因为相关的系统状态，带来不必要的限制。

这里举个反例，比如有个城市建设类游戏，其中有人口幸福指数。若每个市民的幸福指数都是独立的，就需要一些计算。首先假设市民数量尚未严重超标，比如最多 1,000 座建筑，且每座最多住 10 人。若只在必要时计算幸福感，就能更快完成任务。至少在这个游戏里，这些数字相似，正确的做法就是用惰性求值。如果从市民个体角度，而非城市角度来计算，反而会比较棘手。比如一个市民的幸福感可能来自：工作离家近，周边生活设施齐全，远离工业园区，交通便利等。那幸福感计算可能会基于某种寻路算法。如果能缓存寻路结果，至少同一建筑物内的市民可以共享它；但每个建筑到其他建筑的距离总会有微小差异。在这么多实例上运行寻路的开销会很大。

相反，如果从城市角度计算，就可以为每种会影响幸福指数的建筑，通过 Flood Fill 生成距离图，使用 Floyd-Warshall 算法创建整个城市的通用距离图，帮市民确定到办公地点的距离。用 $O(n^3)$ 算法代替 $O(n^2)$ 算法可能有点傻，但寻路是为每个市民做的，所以变成了 $O(n^2m)$，虽然在算法层面也没什么优势。但在真实环境下，寻路本身还有其他开销，计算幸福指数前，执行 Floyd-Warshall 算法生成表格，后续的计算工作就能更简单（指数据存储方面），且进入功能代码之前的分支也更少。Floyd-Warshall 算法还能根据现有地图来确定需要更新的条目，实现部分更新。若从实例角度运行，知道拓扑结构变化，或是附近的建筑类型，就需要以某种形式，逐实例做距离检查。

总之，抽象是解决困难问题的基础。但在游戏中，我们通常不在玩法层

面上解决困难的算法问题。相反，我们往往倾向过早抽象。通过面向对象设计，通常也能以一种简单、可识别的方式抽象。但直到很久以后，对其过度依赖，因而无法在不影响其他代码的情况下清除它时，才逐渐看清其代价。

11.5 层设计与变革

主张：可以通过扩展继承来复用代码。添加新功能很简单。

通常认为，游戏程序员在 C++ 中使用类，主要是为了继承。最直观的好处，是能继承多个接口，获取系统对象 (如物理、动画、渲染) 的属性和代理权。早期使用 C++ 时，层次通常很浅，基本不会超过三层。但后来，诸如玩家、载具、AI 玩家这些主要类中，超过九层的继承链已经很常见了。例如，在《虚幻竞技场》(Unreal Tournament) 中，迷你机枪 (minigun) 的弹药对象是这样：

Miniammo → TournamentAmmo → Ammo → Pickup → Inventory → Actor → Object

游戏开发者借助继承，能稳定实现多态。因而能大量更新、渲染、查询游戏实体，无需手动编码做类型检查。从一个类继承的同时，也增加了功能，同时能减少复制粘贴，所以很受开发者欢迎。早期的混合继承还能减少代码里的 bug，很多时候，bug 之所以存在，只是因为程序员没能修复所有地方。渐渐地，多重继承淡化为只继承接口，只继承自一个真正的类，而任何其他的类都必须是按照 Java 中定义的纯虚接口类。

虽然看起来通过继承来扩展类的功能很安全，但很多情况下，重载方法时，类不一定会像预期那样。要扩展一个类，往往需要阅读源码，不光是该类本身，还有它继承的类。如果基类创建了一个纯虚方法，会强迫子类实现它。如果理由合理，就应强制执行，但又没法强制要求每个继承类都实现这个方法，只能是第一个继承它的可实例化的类。这样可能会导致一些不明显的错误，即一个新类有时会像它的父类那样行事，又或被当作父类。

C++ 中缺失非虚这一概念。没法声明一个函数不是虚函数。即，可以定义一个函数是 `override`，但没法定义它 `non-override`。如果用常用词汇组合，引入新的虚方法，就可能产生问题。如果它与有相同签名的现存函数重叠，或许就是个新 bug。

C++ 继承的另一个陷阱，是运行时与编译时的链接。一个典型的例子：方法调用的默认参，以及难以理解的覆盖规则。你觉得代码 11.2 中的程序

会输出什么？

```
class A {
   virtual void foo( int bar = 5 ) { cout << bar; }
};
class B : public A {
   void foo( int bar = 7 ) { cout << bar * 2; }
};
int main( int argc , char *argv [] ) {
   A *a = new B;
   a->foo();
   return 0;
}
```

Listing 11.2: 运行时？编译期？还是链接期？

如果发现输出是 10，会惊讶吗？有些代码依赖于编译状态，有些依赖于运行时。通过扩展类增加新功能，很快就变成玩火。两层以下的类就可能带来耦合的副作用，抛出异常（或更坏，不抛异常悄悄失败）；绕过改动；又或者因为命名空间已经占用、不兼容等问题，导致功能无法实现，比如要某种对齐，或是需要在某个 RAM Bank 中。

继承确实能干净地实现运行时多态，但正如前文所述，它并非唯一解。通过继承增加新特性，需要重新审视基类，提供默认实现（或是纯虚），然后在所有需要处理新特性的类中实现。因而需要修改基类，如果用纯虚的方式，可能会触及所有子类。因此，尽管编译器能帮我们定位到所有需要修改的代码，但也没能让修改来得更容易。

使用类型成员替代虚表指针，同样能利用到运行时链接，并且对缓存命中率更友好，也更容易添加新功能，更易推断。实现这些新功能时，它的包袱更少，相较于继承，混合与兼容变得更简单，多态代码也能放到一起。例如，假设有个虚函数 `go-forward`，`Car` 类会踩下油门。`Person` 类，需要设置方向向量。`UFO` 类中，也要个方向向量。看起来 `switch` 中的 `default:` 就能胜任。又比如虚函数 `re-fuel`，`Car` 和 `UFO` 会启动 `re-fuel` 计时器，并在播放加油动画时保持静止；而 `Person` 可以直接减耐力药剂库存，立即补充体能。同样，带 `default` 的 `switch` 语句能实现所有运行时多态，但不需要再为在每个类的每个功能尺度的细化实现动用多重继承。继承并不擅长在类中选择每个方法做什么，它仍有可取之处，也的确可以绕开继承实现多态。

使用继承的初衷，是不需要重新审视基类，也不会为了扩展而改变现有代码，但实际上极有可能必须查看基类实现。随着游戏规格变化，在基类层面上做改动也变得很常见。继承也会抑制某些类型分析，将人们的思维锁定

在：对象与游戏中的其他对象类型有是什么的关系。对象是功能的组合，程序员被锁在这一概念之外，灵活性随之一同束缚。把多重继承限制到接口层面，虽然有助于减少代码复杂度，但将类视作复合对象这一好办法也被掩藏。尽管它还会滥用缓存，本身也不是好方案。而 `switch` 类型也能提供类似于虚表的功能，并且没有相关的包袱。那为什么还要把东西放进类里呢？

11.6 分工

主张：模块化架构能减少耦合，更易测试。

人们认为，面向对象的范式也能保证代码质量。严格遵守开闭原则，始终通过访问器、方法、继承来使用和扩展对象，程序员实现的模块化代码明显多于纯过程化编码。每个对象的代码分离成单元。这些单元是所有数据，以及作用于数据的方法的集合。这些已经经过实践验证，测试也更简单，可以对每个对象孤立测试。

然而，我们知道这不是真的，数据因目的产生联系；目的，又是因数据连结在一起的，一系列偶然的关系。

面向对象设计在通信层面存在错误。对象并非系统，系统需要测试，系统不仅包括对象，还包括它们内在的通信。对象间的通信难以测试。实践中，很难隔离类之间的相互作用。面向对象的开发，导致从面向对象的角度去观察系统，因而像数据变换、通信、时间耦合这类非对象，难以独立出来。

模块化结构能限制变化带来的潜在损害，因此是好的。但就像前文中的封装一样，对任何模块的协议都必须足够明确，才能减少外部去依赖实现中非预期副作用。

面向对象的模块由对象的边界定义，而非基于更高层次的概念，因此效果并不好。而好的例子有：`stdio` 的 `FILE`，`CRT` 的 `malloc/free`，`NvTriStrip` 库的 `GenerateStrips`。每一个都是通过稳定、精炼、文档完备的函数集，访问那些的繁杂功能。

面向对象开发中的模块化，能避免不理解代码的程序员破坏代码。但为什么不理解代码的程序员，使用简化的接口也很安全？刚接触代码的人眼中，对象的方法就是它的说明书，所以把所有重要的操作方法写一块，就能为用户提供线索。模块化在这里很重要，游戏中的对象经常很大，不同方面的大量功能汇聚在一起。游戏对象没有尝试找方法解决跨领域的问题，反而想满足所有需求，避免将自己限制在最初的设计中。这种臃肿，模块化的方

法，即，在源代码中按关注度整合方法，对于刚开始接触对象的程序员很有帮助。要解决这个问题，初看应该使用能在基础层面支持跨领域的范式，但 C++ 中的面向对象的代码，似乎并不能胜任它。

如果面向对象开发不能这样提高模块化程度，比不过显式的模块化代码，那它能做什么？

11.7 可复用的通用代码

主张：复用通用代码可以加快开发。

复用旧代码，持续减少开发的成本，开发者们一直将其奉做圣杯。人们觉得，可以用现有代码组装应用程序，最多实现一些额外的小功能，就能避免浪费时间和精力。但事实上，任何有趣的新功能，都可能不兼容旧代码和旧数据。于是需要重写旧代码，好加入新功能，新数据布局。若软件项目能基于现有解决方案，从为旧项目的功能创建的对象中建立，那它可能不是很复杂。任何具有显著复杂度的项目都包括成百上千，甚至是成千上万的特例对象，满足项目所有的特殊需求。例如，绝大多数游戏都会有玩家类，但几乎没有游戏共享一套核心属性。扑克游戏中是否有世界坐标成员变量？赛车游戏的玩家有没有命中率成员变量？纯离线游戏中，需不需要有“玩家账户”(gamer-tag)？可以重复使用的通用类并不能更轻松地创建游戏，它只是将特化的内容转移到其他地方。一些游戏引擎通过用脚本扩展基本类来实现这一点。有些引擎则将玩法限制在某一类型，并通过数据驱动的方式扩展。目前为止，还没有人创建一套游戏 API，如果这样，它必须非常通用，因而没法提供比开发语言本身更多的内容。

复用，被制作人追捧，被没有游戏制作经验的人推崇。对于许多游戏开发者，复用本身已经变成一种目的。泛型的隐患在于，只注重保持一个类的泛型，使其能够最大限度复用，但从未考虑为什么复用，如何复用。首先，为什么？它是主要的绊脚石，开发者需要尽快知道。为了通用而生产，并非一个有效目标。一开始就做通用的东西，只会增加开发的时长，还不带来价值。有些开发者觉得这是短视行为，然而，同样的话也可以用来反驳。如果一个类只用在一处，怎么泛化它？类的实现只在有东西可测时才能保证测试有效，如果它只用在一处，就只能测试它在一种情况下的工作情况。在能复用类之前，本质上就无法测试类的复用性。通常的经验法则，除非至少有三处用它，否则就不是可复用的。如果泛化该类，但除了初始情况外，没有其

他的测试案例，就只能测试在泛化这个类时，有没有造成破坏。所以，如果不能保证这个类在其他类型、情况下都能工作，那么泛化这个类所做的，就只是增加代码量，藏一些 bug。这些 bug 现在隐藏在可以工作的代码中，甚至可能在隔离状态下通过测试，等同于，泛化过程为引入的任何 bug 都盖章，批准，认证。

测试驱动开发，隐式否认了通用编码，除非真的有用。只有把代码转移到更通用的状态，才算是好选择，即，通过复用通用功能减少冗余。

通用代码如果要适用于多数情况，就必须满足更多基本功能。如果写一个模板化的数组容器，方括号运算符访问数组就属于基本功能，还得实现迭代器，可能还要添加插入操作以避免在内存中整理数组。如果有重写这些函数的需求，各种微小的 bug 就悄悄冒出来，而链表在权宜的[5]情况下的实现，可谓是声名狼藉。为了适用所有用户，任何通用容器都应提供一整套操作的方法，`STL` 就是典型。在成为 `STL` 专家前，要理解数百种不同的函数。也必须成为 `STL` 专家，才能确保用 `STL` 写出高效的代码。而且 `STL` 的各种实现，有大量文档可用。大多数 `STL` 的实现即便功能不同，也会非常相似。即便如此，相当于还需要学习另一种语言，因此成为有价值的 `STL` 程序员可能需要花些时间。`STL` 是一种新的语言，它有自己的语法体系。为了限制这种情况，许多游戏公司会大幅缩减并重新解释 `STL` 的功能集，或提供更好的内存管理（因为硬件很笨），扩展可选的容器数目（除了`map`，还可以直接选 `hash-map`, `tree`, `b-tree` 等），或显式实现更简单的容器，如堆栈、单向链表还有它们的 `instrusive` 版本。这些库通常范围较小，因此比 `STL` 的变体更易学习、破解，但仍需要花时间。过去，这种行为算是很好的妥协，但现在 `STL` 有大量在线文档，很少有理由不使用 `STL`，除非内存开销非常大，如嵌入式领域，主内存是以千字节计算的，或者编译时间非常重要 [6]。

然而，我们能了解到的是，开发者仍需学习通用代码，以提高编码效率，不带来意外的性能瓶颈。如果采用 `STL`，至少手头上有大量文档。如果游戏公司实现了一套及其复杂的模板库，不要指望任何程序员在有足够时间学习它之前会去用。也即，如果写通用代码，反倒希望人们不要用它，除非是意外遇到，或被明确告知这是通用代码。并且可能也没法迅速获得信任。换句话说，一开始就写通用代码，是个可以快速产出大量代码，且不带来任何价值的好办法。

---

[5] 译注：quick and dirty

[6] STL 是很大，但比起一些操作系统的头文件也相形见绌，所以要先打对仗。

Bibliography

[1] Scott Bilas. *A Data-Driven Game Object System Gas Powered Games*. 2004. URL: http://gamedevs.org/uploads/data-%20driven-game-object-system.pdf.

[2] Carlos Bueno. *Mature Optimization Handbook*. Facebook, 2013. URL: http://carlos.bueno.org/optimization/.

[3] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: 13.6 (). ISSN: 0001-0782. URL: https://doi.org/10.1145/362384.362685.

[4] E. F. Codd. "Further Normalization of the Data Base Relational Model". In: *Research Report / RJ / IBM / San Jose, California* RJ909 (1971).

[5] Kevin Dickinson. *Why is it called a murder of crows?* URL: https://bigthink.com/life/why-group-murder-of-crows/.

[6] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2011. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[7] Fay Chang; Jeffrey Dean; Sanjay Ghemawat; Wilson C. Hsieh; Deborah A. Wallach Mike Burrows; Tushar Chandra; Andrew Fikes; Robert E. Gruber. *Bigtable: A Distributed Storage System for Structured Data*. Google, Inc., 2016.

[8] Daniel Kahneman. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011.

[9] William Kent. “A Simple Guide to Five Normal Forms in Relational Database Theory”. In: *Commun. ACM* 26.2 (), pp. 120–125. ISSN: 0001-0782. DOI: 10.1145/358024.358054. URL: https://doi.org/10.1145/358024.358054.

[10] John Lakos. *Large-scale C++ Software Design*. 1st. Addison Wesley, 1996.

[11] Noel Llopis. *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)*. URL: http://www.gamesfromwithin.com/data-oriented-design.

[12] Ernst Denert Manfred Broy. *Software Pioneers: Contributions to Software Engineering*. Science & Business Media. Springer, 2012.

[13] Ben Moseley and Peter Marks. “Out of the Tar Pit”. In: (2006).

[14] D. L. Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules”. In: *Communications of the ACM December 1972 Volume 15 Number 12* ().

[15] Peter Seibel. *Coders at Work*. Reflections on the Craft of Programming. Apress, 2009.

[16] Valentin Simonov. *10000 Update() calls*. 2015. URL: https://blogs.unity3d.com/2015/12/23/1k-update-calls/.

[17] Albrecht Tony. *Pitfalls of Object Oriented Programming*. URL: https://www.slideshare.net/EmanWebDev/pitfalls-of-object-oriented-programminggcap09.

[18] Albrecht Tony. *Pitfalls of Object Oriented Programming - Revisited*. URL: https://docs.google.com/presentation/d/1ST3mZgxmxqlpCFkdDhtgw116MQdCr2Fax edit#slide=id.p.