

# Game Programming Patterns

Robert Nystrom

游戏编程模式

gb

# Table of Contents

---

1. [Introduction](#)
2. [介绍](#)
  - i. 架构, 性能和游戏
3. [再探设计模式](#)
  - i. 命令模式
  - ii. 享元模式
  - iii. 观察者模式
  - iv. 原型模式
  - v. 单例模式
  - vi. 状态模式
4. [序列模式](#)
  - i. 双缓冲
  - ii. 游戏循环
  - iii. 更新方法
5. [行为模式](#)
  - i. 字节码
  - ii. 子类沙盒
  - iii. 对象类型
6. [解耦模式](#)
  - i. 组件
  - ii. 事件队列
  - iii. 服务定位器
7. [优化模式](#)
  - i. 数据局部性
  - ii. 脏标记
  - iii. 对象池
  - iv. 空间划分

# Game-Programming-Patterns-CN

## 游戏编程模式中文

[Gitbook地址](#)

### 嘿，游戏开发伙伴们！

- 你是否还在为代码整体规划而苦苦挣扎？
- 是否发现随着代码库的增长却不容易做出些改动？
- 是否感觉到你的游戏就是一个纷乱交杂的巨大的毛球？
- 又或者不知如何将设计模式应用到游戏？
- 听说过“缓存一致性”和“对象池”，但却不知道如何使用它们来提升你的游戏的性能？

你们的救星来啦！我撰写了这本书来解答这些问题。这是我在游戏中所使用的模式总结，这些模式能让我们的代码更整洁，更清晰易懂，以及运行更快！

### 免费在线阅读

当我开始编写游戏时，我希望我有一本这样的书。现在，我希望你能够有一本。

[开始阅读](#)

### 我是谁？

我叫 Bob Nystrom。当我在EA工作的时候，我便开始写这本书了。在EA工作的8年时间里，我见过很多优美的代码，也见过很多着实可怕的代码。我希望我能够将我从这些优美的代码设计中学到的东西，在这里写下来，并教给大家如何写出这样好的代码来。

如果你想要联系我，你可以在[网站写email给我](#)，或者直接在twitter上[@munificentbob](#)都可以。



在线写作书籍的一大好处就是方便修改。如果你发现了错误或者有什么建议，不要犹豫，给我[报告bug](#)或者发送一个 pull request。

### 目录

- [介绍](#)
  - 架构，性能和游戏
- [再探设计模式](#)
  - 命令模式
  - 享元模式
  - 观察者模式
  - 原型模式
  - 单例模式
  - 状态模式
- [序列模式](#)

- 双缓冲
- 游戏循环
- 更新方法
- 行为模式
  - 字节码
  - 子类沙盒
  - 对象类型
- 解耦模式
  - 组件
  - 事件队列
  - 服务定位器
- 优化模式
  - 数据局部性
  - 脏标记
  - 对象池
  - 空间划分

=====

## 翻译

欢迎朋友们阅读并斧正，提交 Issue 或者 send pull request :)。

能有所收获，便是我们翻译中文的意义所在。

## 如何参与？

- 翻译：（目前已经都领取完毕）
  - 在[Issue列表](#)中查看尚未领取的章节
  - 然后在[Issue1](#)中留言回复，我会更新Issue列表状态。
  - Fork项目后开始翻译，提交PR。
- 校正：由于能力所及，总有些地方翻译欠妥，所以校正会一直进行中，朋友们看到有翻译错误，可以在[Issue列表](#)中新建 issue 提出来，更鼓励欢迎直接发送 pull request。

## 参考资料

- [术语参考](#)

## 贡献人列表(排名不分先后)

注：所有贡献人的名字都会在此列出，欢迎大家踊跃参与翻译、校正。

- [子龙山人](#)
- [kislyl](#)
- [Henry-T](#)
- [lazyqiang](#)
- [zhizhen](#)
- [Gwill](#)
- [Tsiannian](#)
- [cone0](#)
- [Gizmosir](#)
- [jptiancai](#)
- [ChildhoodAndy](#)

# 序言

在五年级的时候，我和我的小伙伴们被获准使用一个容放着一些非常破旧的 TRS-80s（译者注：见[维基百科TRS-80s](#)）的废弃教室。为了激励我们，一个老师找到了一些简单的 BASIC 程序的打印输出让我们鼓捣玩耍。

电脑上的音频盒式磁带驱动器当时是坏掉的，所以每次我们想要运行一些代码的时候，我们不得不仔细的从头开始键入。这使得我们更喜欢那些只有几行代码的程序：

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

## 注解

如果计算机打印足够多的次数，或许它会神奇的变成现实哦。（译者注：这里指的是计算机反复打印第10行代码的语句 `BOBBY IS RADICAL!!!`，作者开玩笑的说会变成现实。）

即便如此，整个过程还是比较艰辛。我们不懂得如何去编程，所以一个小的语法错误便让我们感到很费解。程序出毛病是家常便饭，而那是我们只能重头再来。

在有了这些小程序的经验积累之后，我们遇到了个大BOSS：一个代码密密麻麻占去好几页纸的程序。我们光是鼓起勇气决定去尝试它就花了不少时间，光是它的标题“隧道与巨人”（"Tunnels and Trolls"）就令人捉摸不透。这听起来像是个游戏，而还有什么比亲手编写一款电脑游戏更酷？

我们从没让它实际运行起来过。一年后，我们搬出了那个教室。（后来随着我更多地接触BASIC，才意识到它只是一个为桌面游戏使用的角色生成器，而并非整个游戏。）但木已成舟，从那之后，我立志要成为一个游戏开发者。

在我十几岁时，我的家人搞了一台装有 QuickBASIC 的 Macintosh，之后又装了 THINK C。我几乎整个暑假都在那上面倒腾游戏。自学是缓慢而痛苦的。我希望能让程序轻松地跑一些功能-一张地图或者是个小的猜谜游戏-但是随着程序的扩大，这越来越难了。

## 注解

我的许多夏天都是在路易斯安那州南部的沼泽中捕蛇和乌龟来度过的。如果户外不是这样酷热，很有可能，这将是一本爬虫学的书，而不是编程书。

起初，我的挑战在于让程序跑起来。后来，我开始思考如何让程序做一些超越我脑袋所想的工作。除了阅读一些关于“如何用 C++ 编程”的书籍，我开始试图寻找一些关于如何组织程序的书籍。

又过了几年，一个朋友给了我一本书：《设计模式：可复用面向对象软件的基础》。终于来了！这就是我从青少年开始便一直寻找的那本书！我们刚碰面，我就把书从头到尾读了一遍。我之前仍然在为我自己写的程序挣扎犯愁，但是看到别人也如此挣扎并提出了解决方案，我便解脱了。我感觉我终于有了一些武器用来挥舞而不再是赤手空拳了。

## 注解

这是我们第一次见面，5分钟自我介绍之后，我坐在他的沙发上，并在接下来的几个小时里，我聚精会神地阅读而完全忽视了他。我感觉那时候自己的社交能力还是稍有那么一丁点提升的。

在2001年，我得到了我梦想中的工作：EA(Electronic Arts) 的软件工程师。我迫不及待的想看下真正的游戏，以及工程师是如何组织它们的。像 Madden Football 这样的大型游戏到底是一个什么样的架构？他们是怎么让一套代码库在不同平台上运行的？

破解开放的源码是一个震撼人心和令人惊奇的体验。图形、人工智能、动画和视觉效果等各个方面的代码都十分出众。我们公司有人懂得如何榨取CPU的每一个周期并得以善用。甚至一些我认为不可能的东西，这些家伙一个早上就能搞定。

但这种优秀代码的结构往往是事后想出来的。他们太专注于功能以至于忽视了组织架构。模块之间耦合很严重。只要是有作

为的新功能都会被扔进代码库里。我的幻想破灭了，在我看来，恐怕很多程序员，从没翻过设计模式，不曾了解[单例模式](#)。

当然，实际并非想象的那么糟。我曾设想游戏程序员们坐在放满白板的象牙塔中，从头到尾的几周时间里都在想当然地讨论代码架构细节。实际上，我所见到的代码是出自那些被上司催着进度的人之手。他们竭尽全力，而且，我逐渐意识到，他们竭尽全力的结果往往是很棒的。我越深入这些代码，便越是这么觉得。

不幸的是，“隐藏”一词恰恰反映了这样的情况：宝藏埋在代码深处，而许多人只从地表踏过。我看到同事在努力重塑更好的解决方案时，他们所寻求的办法正藏在他们脚下的代码库之中。

这样的问题正是这本书所关注的。我挖掘并打磨出自己在游戏中所发现的最好的设计模式，在此一一呈现给大家，以便我们将时间节省下来发现新大陆，而不是重新造轮子。

## 市面上的书籍

目前市面已经有数十本游戏编程的书籍。为什么还要再写一本？

我见过的大多数游戏编程书籍无非下列两类：

- 关于特定领域的书籍。这些针对性较强的书籍为你的游戏开发打开一个独特而深刻的视角。他们会教你3D图形，实时渲染，物理模拟，人工智能，或音频处理。这些是多数游戏程序员在自己的职业生涯中所专注的领域。
- 关于整个游戏引擎的书籍。相反，这些书尝试涵盖整个游戏引擎的各部分。它们被组织起来形成一个完整的引擎以适用于一些特定类型的游戏，通常是3D第一人称射击游戏。

我喜欢这两类书，但我觉得它们仍留有一些空白。特定领域的书很少会写你的代码块如何与游戏的其他部分交互。你可能擅长物理和渲染（译者注：这里指的是在某个领域特别擅长的人），是你知道如何优雅的将它们拼合起来？

第二类书籍写到了这些，但我通常发现这类书都太单一，太泛泛而谈。特别是随着移动和休闲游戏的兴起，我们正处在充斥着大量不同类型游戏的时代。我们不再只是克隆 Quake（译者注：雷神之锤，第一个真3D实时演算的FPS游戏）了。当你的游戏不适合这个模型时，这类阐述单个引擎的书籍就不再合适了。

相反，这里我想要做的，更倾向于“引导”。在这本书中，每个章节都是一个独立的思想，而你可以将它应用到你的代码里。藉此，你可以混用它们以令其在你制作的游戏中发挥最好的效果。

### 注解

这种“向导”风格的另外一个例子，就是广受大家喜爱的《游戏编程精粹》系列。

## 设计模式相关

任何名字中带有“模式”的编程书籍都和经典图书《设计模式：可复用面向对象软件的基础》脱不了干系。这本书由 Erich Gamma, Richard Helm, Ralph Johnson 和 John Vlissides 完成。（俗称“Gang of Four”--四人组）

### 注解

设计模式一书本身也源自前人的灵感。使用模式语言来描述开放式解决问题的想法来自《A Pattern Language》，由 Christopher Alexander（以及Sarah Ishikawa, Murray Silverstein）完成。

他们的书是关于架构（就像真正的建筑结构有着建筑和围墙之类的东西），但他们希望他人会使用相同的结构来描述在其他领域的解决方案。设计模式（Design Patterns）是 Gang of Four 在软件领域的一个尝试。

本书以“游戏编程模式”命名，并不是说 Gang of Four 的书不适用于游戏。恰恰相反，在[再探设计模式](#)一节中覆盖众多来自 GoF 著作的设计模式，同时强调了在游戏开发中的运用。

从另一面说，我觉得这本书也适用于非游戏软件。我也可以把这本书命名为 *More Design Patterns*，但我认为游戏制作例子更吸引人。难道你真的想要阅读一本关于员工记录和银行账户例子的书么？

话虽这么说，这里介绍的模式在其他软件中是有用的，我觉得他们是特别适合于软件开发，就像在游戏中经常遇到的挑战一样：

- 时间和序列化往往是一个游戏的架构的核心部分。事情必须依照正确的顺序和正确的时间发生。
- 开发周期被高度压缩，程序员们需要能够快速构建和迭代一组丰富且相异的行为，同时不牵涉到他人或者弄乱代码库。
- 所有这些行为被定义后，游戏便开始互动。怪物撕咬英雄，药水混合在一起，炸弹炸到敌人和朋友...诸如此类。这些交互必须很好地进行下去，同时代码库需要保持干净不能纷乱无章得像个交织错乱的毛线球。
- 最后，性能是游戏的关键。游戏开发者们总在不断比赛着看谁能够充分利用平台的性能。游戏周期处理的不同可能意味着产品是成为数以百万计销售的A级游戏，或者满是掉帧且贴满愤怒评论的废铁。

## 如何阅读本书

---

游戏编程模式分为三大部分。第一部分是序言和书的概括。这正是你现在阅读的章节以及[下一章节](#)。

第二部分，再探设计模式，回顾了 Gang of Four 中的一些设计模式。在每个章节中，我会提及自己对该模式的认识，及将该模式运用到游戏中的方法。

最后部分是这本书的重头戏。这部分代表了我认为十分有用的13种设计模式。它们分为四类：序列模式，行为模式，解耦模式，优化模式。

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

- 目的 部分简述了此模式旨在解决的问题。这样你很容易根据自己遇到的问题来快速确定该用哪个模式。
- 动机 部分描述了一个示例、该示例存在问题、我们将要对之采用设计模式。不同于具体的算法，模式通常是无形的，除非针对一些特定的问题。学设计模式离不开示例正如学烘焙离不开面团一样。这个部分提供面团，之后的部分将会教你如何烘焙。
- 模式 部分会提炼出前面示例中的模式本质。如果你想了解该模式的书面描述，就是这部分了。如果你已经熟悉了，这部分也是一个很好的复习，确保你没有忘记该模式。
- 到目前为止，该模式只是就一个单一的例子来解释的。但你怎么该模式是否适用于你的问题呢？使用情境 针对模式使用的情境以及何时避免使用它提供一些指引。使用须知 部分会指出使用该模式时面临的后果和风险。
- 如果像我一样，需要借助具体的实例才能真正的理解，那么示例 正满足你的需要。示例会一步步实现模式，所以你可以清楚的看到模式是如何工作的。
- 模式和单一的算法不同，因为模式是开放式的。每次使用模式的时候，你实现的方式有可能会不同。接下来设计决策 部分，会探讨这个问题，并告诉你在应用模式时要考虑的不同因素。
- 结束部分，参考 部分会告诉你该模式和其他模式的关联并指出使用该模式的一些真实世界中的开源代码。

## 关于示例代码

---

这本书中的示例代码用 C++ 编写，但是这并不意味着这些模式仅在该语言中有用或者说 C++ 比其他语言要好。几乎所有的语言都适用，虽然有些模式确实倾向于面向对象语言。

我选择 C++ 有几个原因。首先，它是商业游戏中最流行的语言，是该行业的通用语言。另外，C++ 基于 C 的语法也是 Java, C#, JavaScript 和许多其他语言的基础。即使你不懂 C++，也没有关系，你可以很轻松的明白示例代码的含义。

这本书的目的，不是教你学习 C++。示例会尽可能保持简单，但是这并不代表良好的 C++ 编码风格或使用就是这样。阅读代码时要理解代码所传达的思想，而不是代码本身的表达。

特别一提的是，示例代码没有采用“现代” C++ -- C++11 或更高版本风格。它没使用标准库并很少使用模板。这是“糟糕”的

C++ 代码，但我仍希望保留下来，这样会对那些从C, Objective-C, Java和其他语言转来的人们更加的友好。

为了避免浪费页面空间，你已经看到了，和模式不相关的代码，有时会在例子中省略，通常用省略号来表示省去的代码。

举个例子，有一个函数，它会处理一些工作，并且会返回一个值。被解释的模式是只关心返回值，不关心处理的工作。在这种情况下，示例代码看起来像这样：

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

## 下一步

---

模式是软件开发中一个不断变化和扩展的部分。这本书从 Gang of Four 文献开始，并分享他们了解的软件设计模式，当书页发布之后过程仍然会继续。

你是这个过程的核心部分。只要你开发了你自己的模式并细化（或者反驳！）这本书中提到的模式，你就是在为软件社区贡献力量。如果你关于书中内容有任何建议，修正或者其他反馈，请与我联系。

===== 目录

[下一节](#)

# Architecture, Performance, and Games

---

Before we plunge headfirst into a pile of patterns, I thought it might help to give you some context about how I think about software architecture and how it applies to games. It may help you understand the rest of this book better. If nothing else, when you get dragged into an argument about how terrible (or awesome) design patterns and software architecture are, it will give you some ammo to use.

note: Note that I didn't presume which side you're taking in that fight. Like any arms dealer, I have wares for sale to all combatants.

## What is Software Architecture?

---

If you read this book cover to cover, you won't come away knowing the linear algebra behind 3D graphics or the calculus behind game physics. It won't show you how to alpha-beta prune your AI's search tree or simulate a room's reverberation in your audio playback.

note:Wow, this paragraph would make a terrible ad for the book.

Instead, this book is about the code between all of that. It's less about writing code than it is about organizing it. Every program has some organization, even if it's just "jam the whole thing into main() and see what happens", so I think it's more interesting to talk about what makes for good organization. How do we tell a good architecture from a bad one?

I've been mulling over this question for about five years. Of course, like you, I have an intuition about good design. We've all suffered through codebases so bad, the best you could hope to do for them is take them out back and put them out of their misery.

note:Let's admit it, most of us are responsible for a few of those.

A lucky few have had the opposite experience, a chance to work with beautifully designed code. The kind of codebase that feels like a perfectly appointed luxury hotel festooned with concierges waiting eagerly on your every whim. What's the difference between the two?

## What is good software architecture?

---

For me, good design means that when I make a change, it's as if the entire program was crafted in anticipation of it. I can solve a task with just a few choice function calls that slot in perfectly, leaving not the slightest ripple on the placid surface of the code.

That sounds pretty, but it's not exactly actionable. "Just write your code so that changes don't disturb its placid surface." Right.

Let me break that down a bit. The first key piece is that architecture is about change. Someone has to be modifying the codebase. If no one is touching the code—whether because it's perfect and complete or so wretched no one will sully their text editor with it—its design is irrelevant. The measure of a design is how easily it accommodates changes. With no changes, it's a runner who never leaves the starting line.

## How do you make a change?

---

Before you can change the code to add a new feature, to fix a bug, or for whatever reason caused you to fire up your editor, you have to understand what the existing code is doing. You don't have to know the whole program, of course, but you need to load all of the relevant pieces of it into your primate brain.

note: It's weird to think that this is literally an OCR process.

We tend to gloss over this step, but it's often the most time-consuming part of programming. If you think paging some data from disk into RAM is slow, try paging it into a simian cerebrum over a pair of optical nerves.

Once you've got all the right context into your wetware, you think for a bit and figure out your solution. There can be a lot of back and forth here, but often this is relatively straightforward. Once you understand the problem and the parts of the code it touches, the actual coding is sometimes trivial.

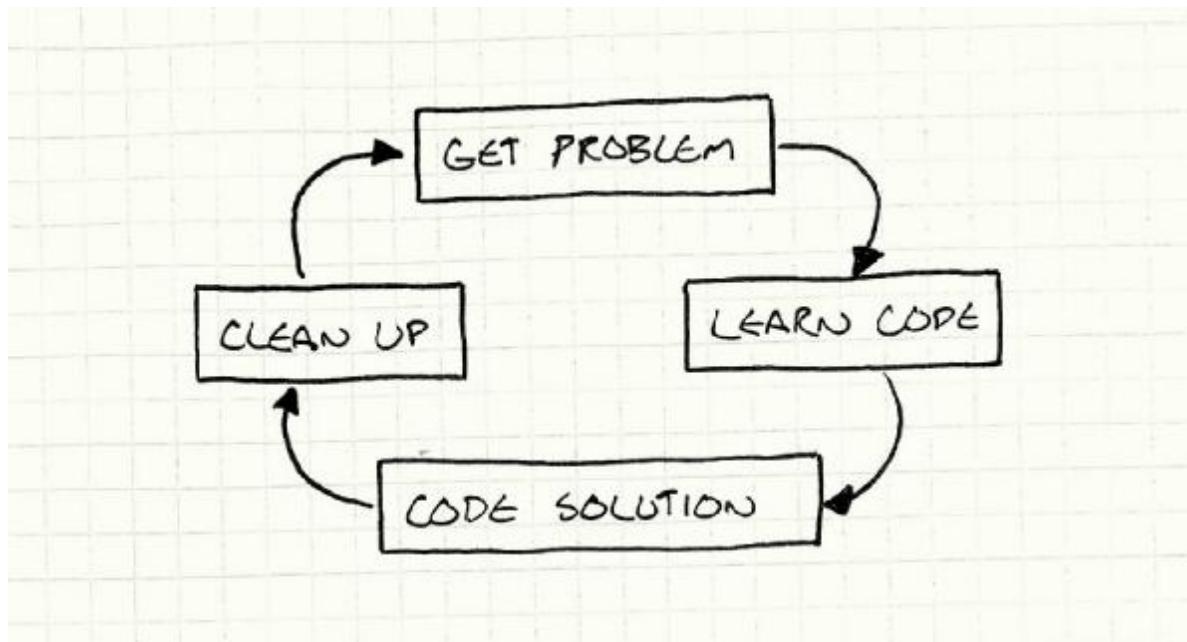
You beat your meaty fingers on the keyboard for a while until the right colored lights blink on screen and you're done, right? Not just yet! Before you write tests and send it off for code review, you often have some cleanup to do.

note: Did I say "tests"? Oh, yes, I did. It's hard to write unit tests for some game code, but a large fraction of the codebase is perfectly testable.

I won't get on a soapbox here, but I'll ask you to consider doing more automated testing if you aren't already. Don't you have better things to do than manually validate stuff over and over again?

You jammed a bit more code into your game, but you don't want the next person to come along to trip over the wrinkles you left throughout the source. Unless the change is minor, there's usually a bit of reorganization to do to make your new code integrate seamlessly with the rest of the program. If you do it right, the next person to come along won't be able to tell when any line of code was written.

In short, the flow chart for programming is something like:



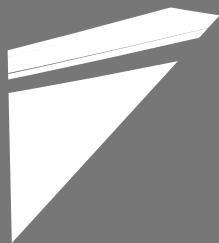
note: The fact that there is no escape from that loop is a little alarming now that I think about it.

## How can decoupling help?

While it isn't obvious, I think much of software architecture is about that learning phase. Loading code into neurons is so painfully slow that it pays to find strategies to reduce the volume of it. This book has an entire section on decoupling patterns, and a large chunk of Design Patterns is about the same idea.

You can define "decoupling" a bunch of ways, but I think if two pieces of code are coupled, it means you can't understand one without understanding the other. If you de-couple them, you can reason about either side independently. That's great because if only one of those pieces is relevant to your problem, you just need to load it into your monkey brain and not the other half too.

To me, this is a key goal of software architecture: minimize the amount of knowledge you need to have in-cranium before



# Performance and Speed

---

There's another critique of software architecture and abstraction that you hear sometimes, especially in game development: that it hurts your game's performance. Many patterns that make your code more flexible rely on virtual dispatch, interfaces, pointers, messages, and other mechanisms that all have at least some runtime cost.

note: One interesting counter-example is templates in C++. Template metaprogramming can sometimes give you the abstraction of interfaces without any penalty at runtime.

There's a spectrum of flexibility here. When you write code to call a concrete method in some class, you're fixing that class at author time—you've hard-coded which class you call into. When you go through a virtual method or interface, the class that gets called isn't known until runtime. That's much more flexible but implies some runtime overhead.

Template metaprogramming is somewhere between the two. There, you make the decision of which class to call at compile time when the template is instantiated.

There's a reason for this. A lot of software architecture is about making your program more flexible. It's about making it take less effort to change it. That means encoding fewer assumptions in the program. You use interfaces so that your code works with any class that implements it instead of just the one that does today. You use observers and messaging to let two parts of the game talk to each other so that tomorrow, it can easily be three or four.

But performance is all about assumptions. The practice of optimization thrives on concrete limitations. Can we safely assume we'll never have more than 256 enemies? Great, we can pack an ID into a single byte. Will we only call a method on one concrete type here? Good, we can statically dispatch or inline it. Are all of the entities going to be the same class? Great, we can make a nice contiguous array of them.

This doesn't mean flexibility is bad, though! It lets us change our game quickly, and development speed is absolutely vital for getting to a fun experience. No one, not even Will Wright, can come up with a balanced game design on paper. It demands iteration and experimentation.

The faster you can try out ideas and see how they feel, the more you can try and the more likely you are to find something great. Even after you've found the right mechanics, you need plenty of time for tuning. A tiny imbalance can wreck the fun of a game.

There's no easy answer here. Making your program more flexible so you can prototype faster will have some performance cost. Likewise, optimizing your code will make it less flexible.

My experience, though, is that it's easier to make a fun game fast than it is to make a fast game fun. One compromise is to keep the code flexible until the design settles down and then tear out some of the abstraction later to improve your performance.

## The Good in Bad Code

---

That brings me to the next point which is that there's a time and place for different styles of coding. Much of this book is about making maintainable, clean code, so my allegiance is pretty clearly to doing things the "right" way, but there's value in slapdash code too.

Writing well-architected code takes careful thought, and that translates to time. Moreso, maintaining a good architecture over the life of a project takes a lot of effort. You have to treat your codebase like a good camper does their campsite: always try to leave it a little better than you found it.

This is good when you're going to be living in and working on that code for a long time. But, like I mentioned earlier, game design requires a lot of experimentation and exploration. Especially early on, it's common to write code that you know you'll throw away.

If you just want to find out if some gameplay idea plays right at all, architecting it beautifully means burning more time before you actually get it on screen and get some feedback. If it ends up not working, that time spent making the code elegant goes to waste when you delete it.

Prototyping—slapping together code that's just barely functional enough to answer a design question—is a perfectly legitimate programming practice. There is a very large caveat, though. If you write throwaway code, you must ensure you're able to throw it away. I've seen bad managers play this game time and time again:

Boss: "Hey, we've got this idea that we want to try out. Just a prototype, so don't feel you need to do it right. How quickly can you slap something together?"

Dev: "Well, if I cut lots of corners, don't test it, don't document it, and it has tons of bugs, I can give you some temp code in a few days."

Boss: "Great!"

A few days pass...

Boss: "Hey, that prototype is great. Can you just spend a few hours cleaning it up a bit now and we'll call it the real thing?"

You need to make sure the people using the throwaway code understand that even though it kind of looks like it works, it cannot be maintained and must be rewritten. If there's a chance you'll end up having to keep it around, you may have to defensively write it well.

note: One trick to ensuring your prototype code isn't obliged to become real code is to write it in a language different from the one your game uses. That way, you have to rewrite it before it can end up in your actual game.

## Striking a Balance

We have a few forces in play:

1. We want nice architecture so the code is easier to understand over the lifetime of the project.
2. We want fast runtime performance.
3. We want to get today's features done quickly.

note: I think it's interesting that these are all about some kind of speed: our long-term development speed, the game's execution speed, and our short-term development speed.

These goals are at least partially in opposition. Good architecture improves productivity over the long term, but maintaining it means every change requires a little more effort to keep things clean.

The implementation that's quickest to write is rarely the quickest to run. Instead, optimization takes significant engineering time. Once it's done, it tends to calcify the codebase: highly optimized code is inflexible and very difficult to change.

There's always pressure to get today's work done today and worry about everything else tomorrow. But if we cram in features as quickly as we can, our codebase will become a mess of hacks, bugs, and inconsistencies that saps our future productivity.

There's no simple answer here, just trade-offs. From the email I get, this disheartens a lot of people. Especially for novices who just want to make a game, it's intimidating to hear, "There is no right answer, just different flavors of wrong."

But, to me, this is exciting! Look at any field that people dedicate careers to mastering, and in the center you will always find a set of intertwined constraints. After all, if there was an easy answer, everyone would just do that. A field you can master in a week is ultimately boring. You don't hear of someone's distinguished career in ditch digging.

note: Maybe you do; I didn't research that analogy. For all I know, there could be avid ditch digging hobbyists, ditch

digging conventions, and a whole subculture around it. Who am I to judge?

To me, this has much in common with games themselves. A game like chess can never be mastered because all of the pieces are so perfectly balanced against one another. This means you can spend your life exploring the vast space of viable strategies. A poorly designed game collapses to the one winning tactic played over and over until you get bored and quit.

## Simplicity

---

Lately, I feel like if there is any method that eases these constraints, it's simplicity. In my code today, I try very hard to write the cleanest, most direct solution to the problem. The kind of code where after you read it, you understand exactly what it does and can't imagine any other possible solution.

I aim to get the data structures and algorithms right (in about that order) and then go from there. I find if I can keep things simple, there's less code overall. That means less code to load into my head in order to change it.

It often runs fast because there's simply not as much overhead and not much code to execute. (This certainly isn't always the case though. You can pack a lot of looping and recursion in a tiny amount of code.)

However, note that I'm not saying simple code takes less time to write. You'd think it would since you end up with less total code, but a good solution isn't an accretion of code, it's a distillation of it.

note: Blaise Pascal famously ended a letter with, "I would have written a shorter letter, but I did not have the time."

Another choice quote comes from Antoine de Saint-Exupery: "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

Closer to home, I'll note that every time I revise a chapter in this book, it gets shorter. Some chapters are tightened by 20% by the time they're done.

We're rarely presented with an elegant problem. Instead, it's a pile of use cases. You want the X to do Y when Z, but W when A, and so on. In other words, a long list of different example behaviors.

The solution that takes the least mental effort is to just code up those use cases one at a time. If you look at novice programmers, that's what they often do: they churn out reams of conditional logic for each case that popped into their head.

But there's nothing elegant in that, and code in that style tends to fall over when presented with input even slightly different than the examples the coder considered. When we think of elegant solutions, what we often have in mind is a general one: a small bit of logic that still correctly covers a large space of use cases.

Finding that is a bit like pattern matching or solving a puzzle. It takes effort to see through the scattering of example use cases to find the hidden order underlying them all. It's a great feeling when you pull it off.

## Get On With It, Already

---

Almost everyone skips the introductory chapters, so I congratulate you on making it this far. I don't have much in return for your patience, but I'll offer up a few bits of advice that I hope may be useful to you:

- Abstraction and decoupling make evolving your program faster and easier, but don't waste time doing them unless you're confident the code in question needs that flexibility.
- Think about and design for performance throughout your development cycle, but put off the low-level, nitty-gritty optimizations that lock assumptions into your code until as late as possible.

note: Trust me, two months before shipping is not when you want to start worrying about that nagging little "game only runs at 1 FPS" problem.

- Move quickly to explore your game's design space, but don't go so fast that you leave a mess behind you. You'll have to live with it, after all.
- If you are going to ditch code, don't waste time making it pretty. Rock stars trash hotel rooms because they know they're going to check out the next day.
- But, most of all, if you want to make something fun, have fun making it.

# 再探设计模式

---

《设计模式：可复用面向对象软件的基础》一书已经出版了将近20年。不过通过阅读本章，你将有机会重温和理解设计模式。软件行业发展迅速，这本书确实有些古老了。这本书经久不衰说明比起许多框架和方法来说，设计模式更加永恒。

然而我认为设计模式到了今天仍然重要，我们从过去几十年中学习到了许多东西。在这个章节中，我们将重温四人帮（Gang of Four）的几个原作设计模式。针对每一种模式，我都希望能写出一些有用、有趣的东西。

我认为有些模式被过度使用（[单例模式](#)），而另一些却又被冷落（[命令模式](#)）。书中还涉及到两个，我想探讨他们与游戏的相关性([享元模式](#)和[观察者模式](#))。最后，有时候我只是觉得了解设计模式在更大点的编程领域的表现是蛮有趣的（[原型模式](#)和[状态模式](#)）。

## 模式

---

- [命令模式](#)
- [享元模式](#)
- [观察者模式](#)
- [原型模式](#)
- [单例模式](#)
- [状态模式](#)

===== [上一节](#)

[目录](#)

[下一节](#)

# 命令模式

## 目的

命令模式是我最喜爱的模式之一。在我写过的许多大型游戏或者其他程序中，都有用到它。正确的使用它，会让你的代码变得更加优雅。对于这个模式，Gang of Four 有着一个预见性的深奥说明：

将一个请求（request）封装成一个对象，从而让你使用不同的请求，请求队列或请求日志来参数化客户端，同时支持请求操作的撤销与恢复。

我想你也和我一样觉得这句话晦涩难明。首先，它分割了自己试图建立的物象。在软件世界之外，一词往往多义。“客户（client）”就是一个人的意思—一个你与它做生意的人。据我查证，人类（human beings）是不可“参数化”的。（译者注：作者在这里的意思是Gang of Four 的说明因为太具概括性，涵盖了软件开发之外的一些定义，使得句子很难理解。）

然后，句子的剩余部分就像你可能使用的模式的一串列表一样。不是特别明朗，除非你的用例恰巧在列表中。我对命令模式的精炼（pithy）概括如下：

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

当然，“精炼”（pithy）通常意味着“令人费解的简洁”，所以我可能改得不是很好。让我解释一下：你可能没听过“Reify”一词，意即“具象化”（make real）。另一个术语reifying的意思是使一些事物成为“第一类”（first-class）。（译者注：你可能在其他书籍中见到说是“[第一类值](#)”的类似说法）

### 注解

“Reify”出自拉丁文“res”，意思为“thing”，加上英语后缀“-fy”，所以就成为了“thingify”，坦白说，这是个很有趣的单词。

这两个术语都意味着，将某个概念（concept）转化为一块数据（data），即一个对象，你可以认为是传入函数的变量等等。所以说命令模式是一个“对象化的方法调用”，我的意思就是封装在一个对象的一个方法调用。

你可能对“回调”（callback），“第一类函数”（first-class function），“函数指针”（function pointer），“闭包”（closure），“局部函数”（partially applied function）更耳熟，至于耳熟哪个就取决于你所使用的语言，而它们都具共性。The Gang of Four 之后这样阐述：

命令就是回调的面向对象化。（Commands are an object-oriented replacement for callbacks.）

这个比他们对模式的概括要好多了。

### 注解

一些语言的反射系统（译者注：如.NET）可以让你在运行时使用类型处理。你可以得到一个对象，它代表着某些其他对象的类，你也可以玩玩看类型可以处理哪些问题。换句话说，反射是一个具体化的类型系统。

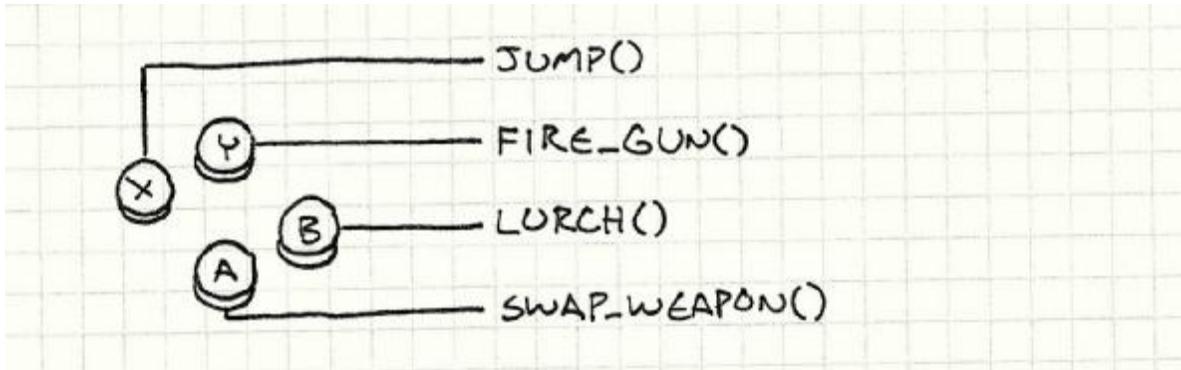
但是这些都比较抽象和模糊。正如我所推崇的那样，我喜欢用一些具体点的东西来开头。为弥补这点，现在开始我将举例，它们都非常适合命令模式。

## 动机

## 输入配置

每个游戏都有一处代码块用来读取用户原始输入—按钮点击，键盘事件，鼠标点击，或者其他等等。它记录每次的输入，并将

之转换为游戏中一个有意义的动作 (action) :



#### 注解

专业级贴士:请勿常按B键。

下面是个简单的实现 :

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

这个函数通常会通过[游戏循环](#)被每帧调用，我想你能理解这段代码在干些什么。如果我们将用户的输入硬关联到游戏的动作 (game actions)，上面的代码是有效的，但是许多游戏允许用户配置他们的按钮与动作的映射。

为了支持自定义配置，我们需要把那些对 jump() 和 fireGun() 的直接调用转换为我们可以换出 (swap out) 的东西。“换出” (swapping out) 听起来很像分配变量，所以我们需要个对象来代表一个游戏动作。这就用到了命令模式。

我们定义了一个基类用来代表一个可激活的游戏命令：

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

#### 注解

当你的接口仅有一个无返回值的方法时，很有可能就会用到命令模式。

然后我们为每个不同的游戏动作创建一个子类：

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

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

// You get the idea...
```

在我们的输入处理中，我们为每个按钮存储一个指针指向他们。

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

    // Methods to bind commands...

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

现在输入处理成了下面这样：

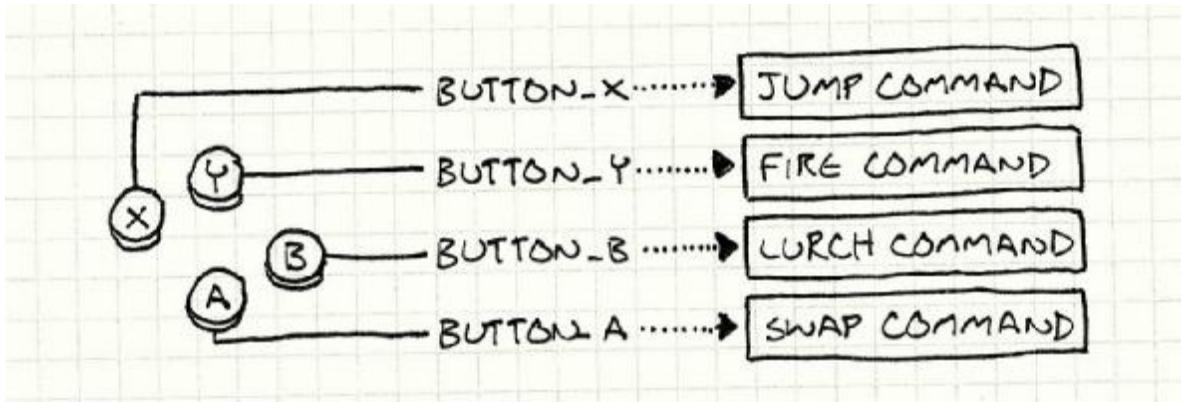
```
void InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) buttonX_->execute();
    else if (isPressed(BUTTON_Y)) buttonY_->execute();
    else if (isPressed(BUTTON_A)) buttonA_->execute();
    else if (isPressed(BUTTON_B)) buttonB_->execute();
}
```

注解

注意到我们这里没有检查命令是否为 `null` 没？这里假设了每个按钮有某个命令与之对应关联。

如果你想要支持不处理任何事情的按钮，而不用明确检查是否为 `null`，我们可以定义一个命令类，这个命令类中的 `execute()` 方法不做任何事情。然后，我们将按钮处理器（button handler）指向一个空对象（null object）代替指向 `null`。这个模式叫[空对象（Null Object）](#)。

以前每个输入都会直接调用一个函数，现在则会有一个间接调用层。



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

## 模式

### 关于角色的说明

我们刚才定义的命令类在上个例子中是有效的，但他们很受限。问题在于，他们假设存在 `jump()`, `fireGun()` 等这样的能找到玩家的头像，使得玩家像木偶一样进行动作处理的顶级函数。

这种假设耦合限制了这些命令的效用。**JumpCommand**类唯一能做的事情就是使得 player 进行跳跃。让我们放宽限制。我们传进去一个我们想要控制的对象而不是用命令对象自身来调用函数：

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

这里，**GameActor**是我们用来表示游戏世界中的角色的“游戏对象”类。我们将它传入 `execute()` 中，以便子类化的命令可以针对我们选择的角色进行调用，就像这样：

```
class JumpCommand : public Command
{
public:
    virtual void execute(GameActor& actor)
    {
        actor.jump();
    }
};
```

现在，我们可以使用这个类来让游戏中的任何角色进行来回跳动。在输入处理和记录命令以及调用正确的对象之间，我们缺少了一部分。首先，我们改变下 `handleInput()`，像下面这样返回一个命令(commands)：

```
Command* InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) return buttonX_;
    if (isPressed(BUTTON_Y)) return buttonY_;
    if (isPressed(BUTTON_A)) return buttonA_;
    if (isPressed(BUTTON_B)) return buttonB_;

    // Nothing pressed, so do nothing.
    return NULL;
}
```

它不能直接执行命令，因为它并不知道该传入那个角色对象。命令是一个具体化的调用，这里正是我们可以利用的地方-我们可以延迟调用。

然后，我们需要一些代码来保存命令并且执行对玩家角色的调用。像下面这样：

```
Command* command = inputHandler.handleInput();
if (command)
{
    command->execute(actor);
}
```

假设 `actor` 是玩家角色的一个引用，这将会基于用户的输入来驱动角色，所以我们可以赋予角色与前例一致的行为。在命令和角色之间加入的间接层使得我们可以让玩家控制游戏中的任何角色，只需通过改变命令执行时传入的角色对象即可。

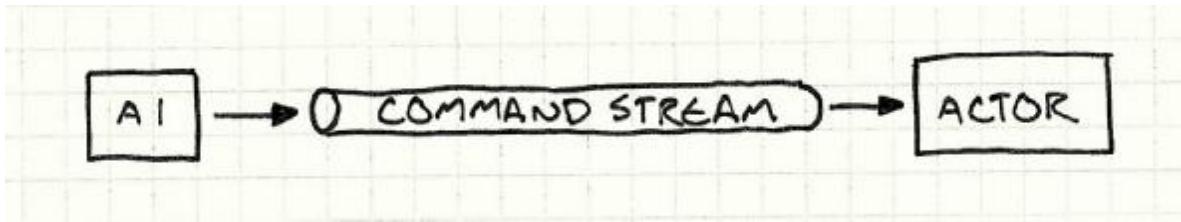
在实践中，这并不是一个常见的功能，但是有一种情况却经常见到。迄今为止，我们只考虑了玩家驱动角色（player-driven character），但是对于游戏世界中的其他角色呢？他们由游戏的AI来驱动。我们可以使用相同的命令模式来作为AI引擎和角色的接口；AI代码部分提供命令（Command）对象用来执行。（译者注：`command->execute(AI对象);`）

AI选择命令，角色执行命令，它们之间的解耦给了我们很大的灵活性。我们可以为不同的角色使用不同的AI模块。或者我们可以为不同种类的行为混合AI。你想要一个更加具有侵略性的敌人？只需要插入一段更具侵略性的AI代码来为它生成命令。事实上，我们甚至可以将AI使用到玩家的角色身上，这对于像游戏需要自动运行的demo模式是很有用的。

通过将控制角色的命令作为第一类对象，我们便去掉了直接的函数调用这样的紧耦合。相反的，把它想象成一个队列或者一个命令流（queue or stream of commands）：

注解

关于队列更多信息，见事件队列



注解

为什么我感觉有必要通过图片来解释“流（stream）”呢？为什么它看起来就像一个管道（tube）一样？

一些代码（输入处理（the input handler）或者AI）生成命令并将它们放置于命令流中，一些代码（发送者（the dispatcher）或者角色自身（actor））执行命令并且调用它们。通过中间的队列，我们解耦了一端的生产者和另一端的消费者。

注解

如果我们把这些命令序列化，我们便可以通过互联网发送数据流。我们可以把玩家的输入，通过网络发送到另外一台机器上，然后进行回放。这是多人网络游戏很重要的一块。

## 撤销和重做（Undo and Redo）

最后这个例子（译者注：作者指的是撤销和重做）是命令模式的成名应用了。如果一个命令对象可以 do 一些事情，那么应该可以很轻松的 undo（撤销）它们。撤销这个行为经常在一些策略游戏中见到，在游戏中如果你不喜欢的话可以回滚一些步骤。在创建游戏时这是一个很常见的工具。如果你想让你的游戏设计师们讨厌你，最可靠的办法就是在关卡编辑器中不要提供撤销命令，让他们不能撤销不小心犯的错误。

注解

这里可能是我的经验之谈。

如果没有命令模式，实现撤销是很困难的。有了它，小菜一碟啊。我们假定一个情景，我们在制作一款单人回合制的游戏，我们想让我们的玩家能够撤销一些行动以便他们能够更多的专注于策略而不是猜测。

我们已经可以很方便的使用命令模式来抽象输入处理，所以每次对角色的移动要封装起来。例如，像下面这样来移动一个单位：

```
class MoveUnitCommand : public Command
{
public:
    MoveUnitCommand(Unit* unit, int x, int y)
        : unit_(unit),
        x_(x),
        y_(y)
    {}

    virtual void execute()
    {
        unit_->moveTo(x_, y_);
    }

private:
    Unit* unit_;
    int x_, y_;
};
```

注意到这个和我们上一个命令不太相同。在上个例子中，我们想要抽象出命令，执行命令时可以针对不同的角色。在这个例

子中，我们特别希望将命令绑定到移动的单位上。这个命令的实例不是一般性质的“移动某些物体”这样适用于很多情境下的操作，在游戏的回合次序中，它是一个特定具体的移动。

这凸显了命令模式在实现时的一个变化。在某些情况下，像我们的第一对的例子，一个命令代表了一个可重用的对象，表示 *a thing that can be done*（一件可完成的事情）。我们前面的输入处理程序仅针对单一的命令对象，并要求在对应按钮被按下时候其 `execute()` 方法被调用。

这里，这些命令更加具体。他们表示 *a thing that can be done at a specific point in time*（一件可在特定时间点完成的事情）。这意味着每次玩家选择移动，输入处理程序代码都会创建一个命令实例。像下面这样：

```
Command* handleInput()
{
    // Get the selected unit...
    Unit* unit = getSelectedUnit();

    if (isPressed(BUTTON_UP)) {
        // Move the unit up one.
        int destY = unit->y() - 1;
        return new MoveUnitCommand(unit, unit->x(), destY);
    }

    if (isPressed(BUTTON_DOWN)) {
        // Move the unit down one.
        int destY = unit->y() + 1;
        return new MoveUnitCommand(unit, unit->x(), destY);
    }

    // Other moves...

    return NULL;
}
```

一次性命令的特质很快能为我们所用。为了撤销命令，我们定义了一个操作，每个命令类都需要来实现它：

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

### 注解

当然了，在没有垃圾回收的语言如C++中，这意味着执行命令的代码也要负责释放它们申请的内存。

`undo()` 方法会反转由对应的 `execute()` 方法改变的游戏状态。下面我们针对上一个移动命令加入了撤销支持：

```
class MoveUnitCommand : public Command
{
public:
    MoveUnitCommand(Unit* unit, int x, int y)
        : unit_(unit),
          xBefore_(0),
          yBefore_(0),
          x_(x),
          y_(y)
    {}

    virtual void execute()
    {
        // Remember the unit's position before the move
        // so we can restore it.
        xBefore_ = unit_->x();
        yBefore_ = unit_->y();

        unit_->moveTo(x_, y_);
    }
}
```

```

virtual void undo()
{
    unit_->moveTo(xBefore_, yBefore_);
}

private:
    Unit* unit_;
    int xBefore_, yBefore_;
    int x_, y_;
};

```

注意到我们在类中添加了一些状态。当单位移动时，它会忘记它刚才在哪。如果我们要撤销移动，我们得记录单位的上一次位置，正是 `xBefore_` 和 `yBefore_` 变量的功能。

#### 注解

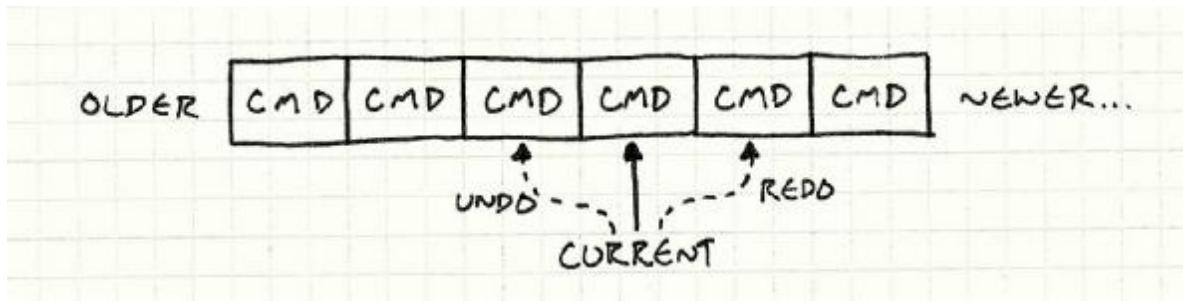
这看起来挺像[备忘录模式](#)的，但是我发现备忘录模式用在这里并不能有效的工作。因为命令试图去修改一个对象状态的一小部分，而为对象的其他数据创建快照是浪费内存。只手动存储被修改的部分相对来说就节省很多内存了。

[持久化数据结构](#)是另一个选择。通过它们，每次对一个对象进行修改都会返回一个新的对象，保留原对象不变。通过这样明智的实现，这些新对象与原对象共享数据，所以比拷贝整个对象的代价要小的多。

使用持久化数据结构，每个命令存储着命令执行前对象的一个引用，所以撤销意味着切换到原来老的对象。

为了让玩家能够撤销一次移动，我们保留了他们执行的上一个命令。当他们敲击 `Control-Z` 时，我们便会调用 `undo()` 方法。（如果他们已经撤销了，那么会变为“重做”，我们会再次执行那个命令。）

支持多次撤销并不难。这次我们不再保存最后一个命令，取而代之的是，我们保存了一个命令列表和“current”（当前）命令的一个引用。当玩家执行了一个命令，我们将这个命令添加到列表中，并将“current”指向它。



当玩家选择“撤销”时，我们撤销当前的命令并且将当前的指针移回去。当他们选择“重做”，我们将指针前移然后执行命令。如果他们在撤销之后选择了一个新的命令，列表中位于当前命令之后的所有命令被舍弃掉。

我第一次在一个关卡编辑器中实现了这一点，顿时自我感觉良好。我很惊讶它是如此的简单而且高效。我们需要指定规则来确保每个数据的更改都经由一个命令实现，但只要定了规则，剩下的就容易得多。

#### 注解

重做在游戏中并不常见，但回放（re-play）却不是。一个很老实的实现方法就是记录每一帧的游戏状态以便能够回放，但是这样会使用大量的内存。

相反，许多游戏会记录每一帧每个实体所执行的一系列命令。为了回放游戏，引擎只需要运行正常游戏的模拟，执行预先录制的命令。

## 设计决策

### 类风格化还是函数风格化？

此前，我说命令（commands）和第一类函数或者闭包相似，但是这里我举的每个例子都用了类定义。如果你熟悉函数式编程，你可能想知道如何用函数式风格实现命令模式。

我用这种方式写例子是因为 C++ 对于第一类函数的支持非常有限。函数指针无须过多阐述，仿函数（译者注：关于仿函数可以看[百科的介绍](#)）看起来比较怪异，还需要定义一个类，C++11 中的闭包使用起来比较棘手因为要手动管理内存。

这并不是说在其他语言中你不应该使用函数来实现命令模式。如果你使用的语言中有闭包的实现，无论怎样，使用它们！在某些方面（In some ways），命令模式对于没有闭包的语言来说是模拟闭包的一种方式。

#### 注解

我在某些方面（In some ways），是因为即使在有闭包的语言中为命令构建实际的类或结构仍然是有用的。如果你的命令有多个操作（如可撤销命令），映射到一个单一函数是比较尴尬的。

定义一个实际的附带字段的类也有助于读者很容易分辨该命令中包含哪些数据。闭包自动包装一些状态是比较简洁，但它们太过于自动化了以至于很难分辨出它们实际上持有的状态。

举个例子，如果我们在用 JavaScript 编写游戏，我们可以像下面这样创建一个单位移动命令：

```
function makeMoveUnitCommand(unit, x, y) {
    // This function here is the command object:
    return function() {
        unit.moveTo(x, y);
    }
}
```

我们也可以通过闭包来添加对撤销的支持：

```
function makeMoveUnitCommand(unit, x, y) {
    var xBefore, yBefore;
    return {
        execute: function() {
            xBefore = unit.x();
            yBefore = unit.y();
            unit.moveTo(x, y);
        },
        undo: function() {
            unit.moveTo(xBefore, yBefore);
        }
    };
}
```

如果你熟悉函数式风格，上面这么做你会感到很自然。如果不熟悉，我希望这个章节能够帮助你了解一些。对我来说，命令模式的作用能够真正的显示函数式编程在解决许多问题时是多么的高效。

## 参考

- 你可能最终会有很多不同的命令类。为了更容易地实现这些类，定义一个具体的基类，里面有着一些方便的高层次的方法，这样派生的命令可以将它们组合来定义自身的行为，这么做通常是有帮助的。它会将命令的主要方法 `execute()` 变成子类沙盒。
- 在我们的例子中，我们明确地选择了那些角色会执行一个命令。在某些情况下，尤其是在对象模型是分层的情况下，它可能没这么直观。一个对象可以响应一个命令，或者它可以决定于关闭一些从属对象。如果你这样做，你需要了解下[责任链\(Chain of Responsibility\)](#)。
- 一些命令如第一个例子中的 `JumpCommand` 是一些纯行为的代码块，无需过多阐述。在类似情况下，拥有不止一个这样命令类的实例会浪费内存，因为所有的实例是等价的。[享元模式](#)就是解决这个问题的。

#### 注解

你可以用[单例模式](#)实现它，但我奉劝你别这么做。

===== 上一节

目录

[下一节](#)

# 享元模式

云雾散去，露出一片巨大而又古老、绵延的深林。数不清的远古铁杉，高出你而形成一个绿色的大教堂。叶子像彩色玻璃窗一样，将阳光打碎成一束束金黄色的雾气。巨大的树干之间，你能在远出制作出巨大的深林。对于我们游戏开发者来说，这是一种超凡脱俗的梦境设置，这样的场景我们经常通过一个模式来实现，它的名字可能并不常见，那就是享元模式。

## 构成深林的树

我能用简短的语句描述绵延不绝的深林，然而在一个虚拟现实游戏中去实现它却又是另一回事。当你看到各种形状不一的树填满整个屏幕时，在图形程序员眼里看到的，却是每隔60分之一秒就必须渲染进GPU的数以百万计的多边形。

我们谈到成千上万的树，每一颗的几何结构又都包含了成千上万的多边形。即便你有足够的内存来描述这片深林，到了需要渲染的时候，这些数据还必须像搭乘巴士一样从CPU传输到GPU。每一棵树都有一堆与之相关联的数据：

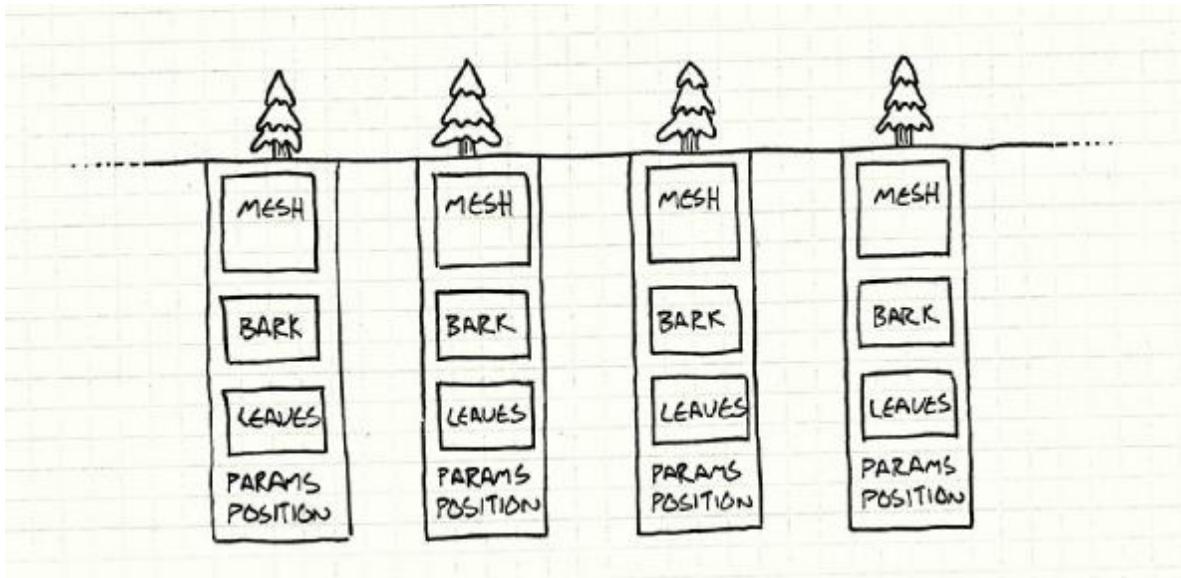
- 交织成网状的定义了树的主干、树枝以及树叶的多边形。
- 树皮和树叶的贴图。
- 它在深林中的位置以及朝向。
- 大量的参数像大小、颜色等，这样每棵树看起来都不一样。

如果你将它在代码中表述出来，那么你将得到下面的东西：

```
class Tree
{
private:
    Mesh mesh_;
    Texture bark_;
    Texture leaves_;
    Vector position_;
    double height_;
    double thickness_;
    Color barkTint_;
    Color leafTint_;
};
```

数据很多，并且网格和贴图数据还挺大。整个深林的物体在一帧中被扔进去GPU就太多了，所幸的是，有一个很古老的窍门来处理这个。

关键的是，即便深林中有成千上万的树，它们大部分看起来相似。它们大部分都用同样的网格和贴图。这就意味着远近许多物体间都有着许多共性。



我们通过把物体分块能很明显地模拟这一切，首先，我们取出所有树共有的数据放到一个单独的类：

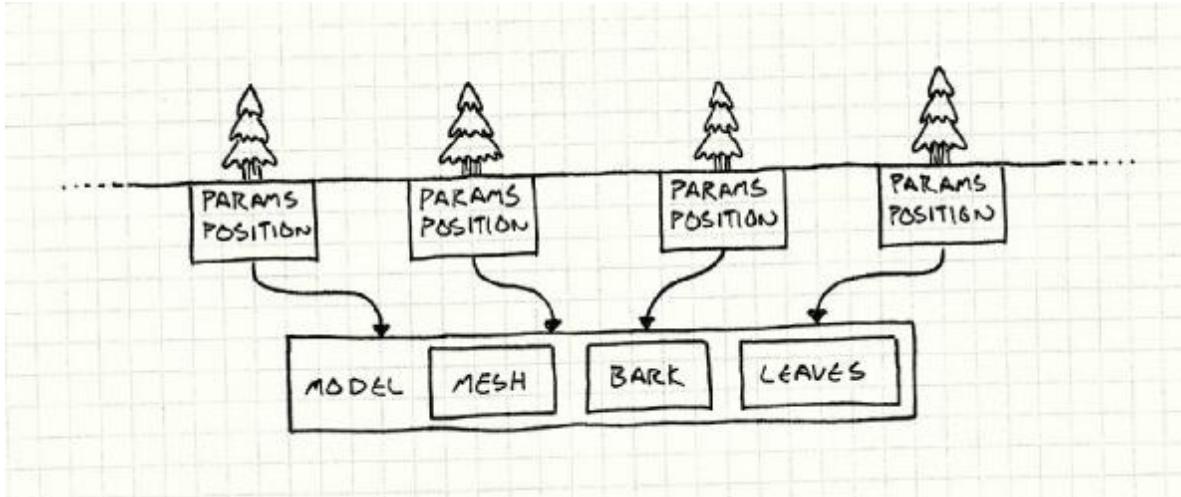
```
class TreeModel
{
private:
    Mesh mesh_;
    Texture bark_;
    Texture leaves_;
};
```

游戏只需要一个这样的类，因为没有道理在内存中存有成千上万个同样的网格和贴图。因此，每一个树的实例，都用一个指向共享的TreeModel这个类的引用。这就意味着，在Tree这个类里有这么一个指针：

```
class Tree
{
private:
    TreeModel* model_;

    Vector position_;
    double height_;
    double thickness_;
    Color barkTint_;
    Color leafTint_;
};
```

你可以这样形象地描述：



这样能很好的在主内存中存储物体，不过这却对渲染没太大帮助。在树林搬到屏幕上之前，它必须自己搬运到GPU，我们必须用显卡能理解的方式来表达资源共享。

## 一千个实例

# Observer

---

^title Observer ^section Design Patterns Revisited

You can't throw a rock at a computer without hitting an application built using the [Model-View-Controller](#) architecture, and underlying that is the Observer pattern. Observer is so pervasive that Java put it in its core library ([java.util.Observer](#)) and C# baked it right into the *language* (the `event` keyword).

Like so many things in software, MVC was invented by Smalltalkers in the seventies. Lispers probably claim they came up with it in the sixties but didn't bother writing it down.

Observer is one of the most widely used and widely known of the original Gang of Four patterns, but the game development world can be strangely cloistered at times, so maybe this is all news to you. In case you haven't left the abbey in a while, let me walk you through a motivating example.

## Achievement Unlocked

---

Say we're adding an achievements system to our game. It will feature dozens of different badges players can earn for completing specific milestones like "Kill 100 Monkey Demons", "Fall off a Bridge", or "Complete a Level Wielding Only a Dead Weasel".



I swear I had no double meaning in mind when I drew this.

This is tricky to implement cleanly since we have such a wide range of achievements that are unlocked by all sorts of different behaviors. If we aren't careful, tendrils of our achievement system will twine their way through every dark corner of our codebase. Sure, "Fall off a Bridge" is somehow tied to the physics engine, but do we really want to see a call to `unlockFallOffBridge()` right in the middle of the linear algebra in our collision resolution algorithm?

This is a rhetorical question. No self-respecting physics programmer would ever let us sully their beautiful mathematics with something as pedestrian as *gameplay*.

What we'd like, as always, is to have all the code concerned with one facet of the game nicely lumped in one place. The challenge is that achievements are triggered by a bunch of different aspects of gameplay. How can that work without coupling the achievement code to all of them?

That's what the observer pattern is for. It lets one piece of code announce that something interesting happened *without actually caring who receives the notification*.

For example, we've got some physics code that handles gravity and tracks which bodies are relaxing on nice flat surfaces and which are plummeting toward sure demise. To implement the "Fall off a Bridge" badge, we could just jam the achievement code right in there, but that's a mess. Instead, we can just do:

^code physics-update

All it does is say, "Uh, I don't know if anyone cares, but this thing just fell. Do with that as you will."

The physics engine does have to decide what notifications to send, so it isn't entirely decoupled. But in architecture, we're most often trying to make systems *better*, not *perfect*.

The achievement system registers itself so that whenever the physics code sends a notification, the achievement system receives it. It can then check to see if the falling body is our less-than-graceful hero, and if his perch prior to this new, unpleasant encounter with classical mechanics was a bridge. If so, it unlocks the proper achievement with associated fireworks and fanfare, and it does all of this with no involvement from the physics code.

In fact, we can change the set of achievements or tear out the entire achievement system without touching a line of the physics engine. It will still send out its notifications, oblivious to the fact that nothing is receiving them anymore.

Of course, if we *permanently* remove achievements and nothing else ever listens to the physics engine's notifications, we may as well remove the notification code too. But during the game's evolution, it's nice to have this flexibility.

## How it Works

---

If you don't already know how to implement the pattern, you could probably guess from the previous description, but to keep things easy on you, I'll walk through it quickly.

### The observer

We'll start with the nosy class that wants to know when another object does something interesting. These inquisitive objects are defined by this interface:

`^code observer`

The parameters to `onNotify()` are up to you. That's why this is the *Observer pattern* and not the Observer "ready-made code you can paste into your game". Typical parameters are the object that sent the notification and a generic "data" parameter you stuff other details into. If you're coding in a language with generics or templates, you'll probably use them here, but it's also fine to tailor them to your specific use case. Here, I'm just hardcoding it to take a game entity and an enum that describes what happened.

Any concrete class that implements this becomes an observer. In our example, that's the achievement system, so we'd have something like so:

`^code achievement-observer`

### The subject

The notification method is invoked by the object being observed. In Gang of Four parlance, that object is called the "subject". It has two jobs. First, it holds the list of observers that are waiting oh-so-patiently for a missive from it:

`^code subject-list`

In real code, you would use a dynamically-sized collection instead of a dumb array. I'm sticking with the basics here for people coming from other languages who don't know C++'s standard library.

The important bit is that the subject exposes a *public API* for modifying that list:

`^code subject-register`

That allows outside code to control who receives notifications. The subject communicates with the observers, but it isn't *coupled* to them. In our example, no line of physics code will mention achievements. Yet, it can still talk to the achievements system. That's the clever part about this pattern.

It's also important that the subject has a *list* of observers instead of a single one. It makes sure that observers aren't implicitly coupled to *each other*. For example, say the audio engine also observes the fall event so that it can play an appropriate sound. If the subject only supported one observer, when the audio engine registered itself, that would *unregister* the achievements system.

That means those two systems would interfere with each other -- and in a particularly nasty way, since the second would disable the first. Supporting a list of observers ensures that each observer is treated independently from the others. As far as they know, each is the only thing in the world with eyes on the subject.

The other job of the subject is sending notifications:

`^code subject-notify`

Note that this code assumes observers don't modify the list in their `onNotify()` methods. A more robust implementation would either prevent or gracefully handle concurrent modification like that.

## Observable physics

Now, we just need to hook all of this into the physics engine so that it can send notifications and the achievement system can wire itself up to receive them. We'll stay close to the original *Design Patterns* recipe and inherit `Subject`:

`^code physics-inherit`

This lets us make `notify()` in `Subject` protected. That way the derived physics engine class can call it to send notifications, but code outside of it cannot. Meanwhile, `addObserver()` and `removeObserver()` are public, so anything that can get to the physics system can observe it.

In real code, I would avoid using inheritance here. Instead, I'd make `Physics` have an instance of `Subject`. Instead of observing the physics engine itself, the subject would be a separate "falling event" object. Observers could register themselves using something like: `^code physics-event` To me, this is the difference between "observer" systems and "event" systems. With the former, you observe *the thing that did something interesting*. With the latter, you observe an object that represents *the interesting thing that happened*.

Now, when the physics engine does something noteworthy, it calls `notify()` like in the motivating example before. That walks the observer list and gives them all the heads up.



Pretty simple, right? Just one class that maintains a list of pointers to instances of some interface. It's hard to believe that something so straightforward is the communication backbone of countless programs and app frameworks.

But the Observer pattern isn't without its detractors. When I've asked other game programmers what they think about this pattern, they bring up a few complaints. Let's see what we can do to address them, if anything.

## "It's Too Slow"

---

I hear this a lot, often from programmers who don't actually know the details of the pattern. They have a default assumption that anything that smells like a "design pattern" must involve piles of classes and indirection and other creative ways of squandering CPU cycles.

The Observer pattern gets a particularly bad rap here because it's been known to hang around with some shady characters named "events", "messages", and even "data binding". Some of those systems *can* be slow (often deliberately, and for good reason). They involve things like queuing or doing dynamic allocation for each notification.

This is why I think documenting patterns is important. When we get fuzzy about terminology, we lose the ability to communicate clearly and succinctly. You say, "Observer", and someone hears "Events" or "Messaging" because either no one bothered to write down the difference or they didn't happen to read it. That's what I'm trying to do with this book. To cover my bases, I've got a chapter on events and messages too: [Event Queue](#).

But, now that you've seen how the pattern is actually implemented, you know that isn't the case. Sending a notification is simply walking a list and calling some virtual methods. Granted, it's a *bit* slower than a statically dispatched call, but that cost is negligible in all but the most performance-critical code.

I find this pattern fits best outside of hot code paths anyway, so you can usually afford the dynamic dispatch. Aside from

that, there's virtually no overhead. We aren't allocating objects for messages. There's no queueing. It's just an indirection over a synchronous method call.

## It's too fast?

In fact, you have to be careful because the Observer pattern *is* synchronous. The subject invokes its observers directly, which means it doesn't resume its own work until all of the observers have returned from their notification methods. A slow observer can block a subject.

This sounds scary, but in practice, it's not the end of the world. It's just something you have to be aware of. UI programmers -- who've been doing event-based programming like this for ages -- have a time-worn motto for this: "stay off the UI thread".

If you're responding to an event synchronously, you need to finish and return control as quickly as possible so that the UI doesn't lock up. When you have slow work to do, push it onto another thread or a work queue.

You do have to be careful mixing observers with threading and explicit locks, though. If an observer tries to grab a lock that the subject has, you can deadlock the game. In a highly threaded engine, you may be better off with asynchronous communication using an [Event Queue](#).

## "It Does Too Much Dynamic Allocation"

---

Whole tribes of the programmer clan -- including many game developers -- have moved onto garbage collected languages, and dynamic allocation isn't the boogie man that it used to be. But for performance-critical software like games, memory allocation still matters, even in managed languages. Dynamic allocation takes time, as does reclaiming memory, even if it happens automatically.

Many game developers are less worried about allocation and more worried about *fragmentation*. When your game needs to run continuously for days without crashing in order to get certified, an increasingly fragmented heap can prevent you from shipping. The [Object Pool](#) chapter goes into more detail about this and a common technique for avoiding it.

In the example code before, I used a fixed array because I'm trying to keep things dead simple. In real implementations, the observer list is almost always a dynamically allocated collection that grows and shrinks as observers are added and removed. That memory churn spooks some people.

Of course, the first thing to notice is that it only allocates memory when observers are being wired up. *Sending* a notification requires no memory allocation whatsoever -- it's just a method call. If you hook up your observers at the start of the game and don't mess with them much, the amount of allocation is minimal.

If it's still a problem, though, I'll walk through a way to implement adding and removing observers without any dynamic allocation at all.

## Linked observers

In the code we've seen so far, `Subject` owns a list of pointers to each `observer` watching it. The `observer` class itself has no reference to this list. It's just a pure virtual interface. Interfaces are preferred over concrete, stateful classes, so that's generally a good thing.

But if we are willing to put a bit of state in `observer`, we can solve our allocation problem by threading the subject's list *through the observers themselves*. Instead of the subject having a separate collection of pointers, the observer objects become nodes in a linked list:



To implement this, first we'll get rid of the array in `Subject` and replace it with a pointer to the head of the list of observers:

<sup>^code linked-subject</sup>

Then we'll extend `Observer` with a pointer to the next observer in the list:

`^code linked-observer`

We're also making `subject` a friend class here. The subject owns the API for adding and removing observers, but the list it will be managing is now inside the `observer` class itself. The simplest way to give it the ability to poke at that list is by making it a friend.

Registering a new observer is just wiring it into the list. We'll take the easy option and insert it at the front:

`^code linked-add`

The other option is to add it to the end of the linked list. Doing that adds a bit more complexity. `subject` has to either walk the list to find the end or keep a separate `tail` pointer that always points to the last node.

Adding it to the front of the list is simpler, but does have one side effect. When we walk the list to send a notification to every observer, the most *recently* registered observer gets notified *first*. So if you register observers A, B, and C, in that order, they will receive notifications in C, B, A order.

In theory, this doesn't matter one way or the other. It's a tenet of good observer discipline that two observers observing the same subject should have no ordering dependencies relative to each other. If the ordering *does* matter, it means those two observers have some subtle coupling that could end up biting you.

Let's get removal working:

`^code linked-remove`

Removing a node from a linked list usually requires a bit of ugly special case handling for removing the very first node, like you see here. There's a more elegant solution using a pointer to a pointer. I didn't do that here because it confuses at least half the people I show it to. It's a worthwhile exercise for you to do, though: It helps you really think in terms of pointers. Because we have a singly linked list, we have to walk it to find the observer we're removing. We'd have to do the same thing if we were using a regular array for that matter. If we use a *doubly* linked list, where each observer has a pointer to both the observer after it and before it, we can remove an observer in constant time. If this were real code, I'd do that.

The only thing left to do is send a notification. That's as simple as walking the list:

`^code linked-notify`

Here, we walk the entire list and notify every single observer in it. This ensures that all of the observers get equal priority and are independent of each other. We could tweak this such that when an observer is notified, it can return a flag indicating whether the subject should keep walking the list or stop. If you do that, you're pretty close to having the [Chain of Responsibility pattern](#).

Not too bad, right? A subject can have as many observers as it wants, without a single whiff of dynamic memory.

Registering and unregistering is as fast as it was with a simple array. We have sacrificed one small feature, though.

Since we are using the observer object itself as a list node, that implies it can only be part of one subject's observer list. In other words, an observer can only observe a single subject at a time. In a more traditional implementation where each subject has its own independent list, an observer can be in more than one of them simultaneously.

You may be able to live with that limitation. I find it more common for a `subject` to have multiple `observers` than vice versa. If it *is* a problem for you, there is another more complex solution you can use that still doesn't require dynamic allocation. It's too long to cram into this chapter, but I'll sketch it out and let you fill in the blanks...

## A pool of list nodes

Like before, each subject will have a linked list of observers. However, those list nodes won't be the observer objects themselves. Instead, they'll be separate little "list node" objects that contain a pointer to the observer and then a pointer to the next node in the list.

Since multiple nodes can all point to the same observer, that means an observer can be in more than one subject's list at the same time. We're back to being able to observe multiple subjects simultaneously.

Linked lists come in two flavors. In the one you learned in school, you have a node object that contains the data. In our previous linked observer example, that was flipped around: the `data` (in this case the observer) contained the `node` (i.e. the `next_` pointer). The latter style is called an "intrusive" linked list because using an object in a list intrudes into the definition of that object itself. That makes intrusive lists less flexible but, as we've seen, also more efficient. They're popular in places like the Linux kernel where that trade-off makes sense.

The way you avoid dynamic allocation is simple: since all of those nodes are the same size and type, you pre-allocate an [Object Pool](#) of them. That gives you a fixed-size pile of list nodes to work with, and you can use and reuse them as you need without having to hit an actual memory allocator.

## Remaining Problems

---

I think we've banished the three boogie men used to scare people off this pattern. As we've seen, it's simple, fast, and can be made to play nice with memory management. But does that mean you should use observers all the time?

Now, that's a different question. Like all design patterns, the Observer pattern isn't a cure-all. Even when implemented correctly and efficiently, it may not be the right solution. The reason design patterns get a bad rap is because people apply good patterns to the wrong problem and end up making things worse.

Two challenges remain, one technical and one at something more like the maintainability level. We'll do the technical one first because those are always easiest.

### Destroying subjects and observers

The sample code we walked through is solid, but it side-steps an important issue: what happens when you delete a subject or an observer? If you carelessly call `delete` on some observer, a subject may still have a pointer to it. That's now a dangling pointer into deallocated memory. When that subject tries to send a notification, well... let's just say you're not going to have a good time.

Not to point fingers, but I'll note that *Design Patterns* doesn't mention this issue at all.

Destroying the subject is easier since in most implementations, the observer doesn't have any references to it. But even then, sending the subject's bits to the memory manager's recycle bin may cause some problems. Those observers may still be expecting to receive notifications in the future, and they don't know that that will never happen now. They aren't observers at all, really, they just think they are.

You can deal with this in a couple of different ways. The simplest is to do what I did and just punt on it. It's an observer's job to unregister itself from any subjects when it gets deleted. More often than not, the observer *does* know which subjects it's observing, so it's usually just a matter of adding a `removeObserver()` call to its destructor.

As is often the case, the hard part isn't doing it, it's *remembering* to do it.

If you don't want to leave observers hanging when a subject gives up the ghost, that's easy to fix. Just have the subject send one final "dying breath" notification right before it gets destroyed. That way, any observer can receive that and take whatever action it thinks is appropriate.

Mourn, send flowers, compose elegy, etc.

People -- even those of us who've spent enough time in the company of machines to have some of their precise nature rub off on us -- are reliably terrible at being reliable. That's why we invented computers: they don't make the mistakes we so often do.

A safer answer is to make observers automatically unregister themselves from every subject when they get destroyed. If you implement the logic for that once in your base observer class, everyone using it doesn't have to remember to do it themselves. This does add some complexity, though. It means each observer will need a list of the `subjects` it's observing. You end up with pointers going in both directions.

## Don't worry, I've got a GC

All you cool kids with your hip modern languages with garbage collectors are feeling pretty smug right now. Think you don't have to worry about this because you never explicitly delete anything? Think again!

Imagine this: you've got some UI screen that shows a bunch of stats about the player's character like their health and stuff. When the player brings up the screen, you instantiate a new object for it. When they close it, you just forget about the object and let the GC clean it up.

Every time the character takes a punch to the face (or elsewhere, I suppose), it sends a notification. The UI screen observes that and updates the little health bar. Great. Now what happens when the player dismisses the screen, but you don't unregister the observer?

The UI isn't visible anymore, but it won't get garbage collected since the character's observer list still has a reference to it. Every time the screen is loaded, we add a new instance of it to that increasingly long list.

The entire time the player is playing the game, running around, and getting in fights, the character is sending notifications that get received by *all* of those screens. They aren't on screen, but they receive notifications and waste CPU cycles updating invisible UI elements. If they do other things like play sounds, you'll get noticeably wrong behavior.

This is such a common issue in notification systems that it has a name: the *lapsed listener problem*. Since subjects retain references to their listeners, you can end up with zombie UI objects lingering in memory. The lesson here is to be disciplined about unregistration.

An even surer sign of its significance: it has a [Wikipedia article](#).

## What's going on?

The other, deeper issue with the Observer pattern is a direct consequence of its intended purpose. We use it because it helps us loosen the coupling between two pieces of code. It lets a subject indirectly communicate with some observer without being statically bound to it.

This is a real win when you're trying to reason about the subject's behavior, and any hangers-on would be an annoying distraction. If you're poking at the physics engine, you really don't want your editor -- or your mind -- cluttered up with a bunch of stuff about achievements.

On the other hand, if your program isn't working and the bug spans some chain of observers, reasoning about that communication flow is much more difficult. With an explicit coupling, it's as easy as looking up the method being called. This is child's play for your average IDE since the coupling is static.

But if that coupling happens through an observer list, the only way to tell who will get notified is by seeing which observers happen to be in that list *at runtime*. Instead of being able to *statically* reason about the communication structure of the program, you have to reason about its *imperative, dynamic* behavior.

My guideline for how to cope with this is pretty simple. If you often need to think about *both* sides of some communication in order to understand a part of the program, don't use the Observer pattern to express that linkage. Prefer something more explicit.

When you're hacking on some big program, you tend to have lumps of it that you work on all together. We have lots of terminology for this like "separation of concerns" and "coherence and cohesion" and "modularity", but it boils down to "this stuff goes together and doesn't go with this other stuff".

The observer pattern is a great way to let those mostly unrelated lumps talk to each other without them merging into one big lump. It's less useful *within* a single lump of code dedicated to one feature or aspect.

That's why it fits our example well: achievements and physics are almost entirely unrelated domains, likely implemented by different people. We want the bare minimum of communication between them so that working on either one doesn't require much knowledge of the other.

## Observers Today

---

*Design Patterns* came out in the 90s. Back then, object-oriented programming was *the* hot paradigm. Every programmer on Earth wanted to "Learn OOP in 30 Days," and middle managers paid them based on the number of classes they created. Engineers judged their mettle by the depth of their inheritance hierarchies.

That same year, Ace of Base had not one but *three* hit singles, so that may tell you something about our taste and discernment back then.

The Observer pattern got popular during that zeitgeist, so it's no surprise that it's class-heavy. But mainstream coders now are more comfortable with functional programming. Having to implement an entire interface just to receive a notification doesn't fit today's aesthetic.

It feels heavyweight and rigid. It *is* heavyweight and rigid. For example, you can't have a single class that uses different notification methods for different subjects.

This is why the subject usually passes itself to the observer. Since an observer only has a single `onNotify()` method, if it's observing multiple subjects, it needs to be able to tell which one called it.

A more modern approach is for an "observer" to be only a reference to a method or function. In languages with first-class functions, and especially ones with closures, this is a much more common way to do observers.

These days, practically every language has closures. C++ overcame the challenge of closures in a language without garbage collection, and even Java finally got its act together and introduced them in JDK 8.

For example, C# has "events" baked into the language. With those, the observer you register is a "delegate", which is that language's term for a reference to a method. In JavaScript's event system, observers *can* be objects supporting a special `EventListener` protocol, but they can also just be functions. The latter is almost always what people use.

If I were designing an observer system today, I'd make it function-based instead of class-based. Even in C++, I would tend toward a system that let you register member function pointers as observers instead of instances of some `Observer` interface.

[Here's][delegate] an interesting blog post on one way to implement this in C++: [delegate]:

<http://molecularmusings.wordpress.com/2011/09/19/generic-type-safe-delegates-and-events-in-c/>

## Observers Tomorrow

---

Event systems and other observer-like patterns are incredibly common these days. They're a well-worn path. But if you write a few large apps using them, you start to notice something. A lot of the code in your observers ends up looking the same. It's usually something like:

1. Get notified that some state has changed.
2. Imperatively modify some chunk of UI to reflect the new state.

It's all, "Oh, the hero health is 7 now? Let me set the width of the health bar to 70 pixels." After a while, that gets pretty tedious. Computer science academics and software engineers have been trying to eliminate that tedium for a *long* time. Their attempts have gone under a number of different names: "dataflow programming", "functional reactive programming", etc.

While there have been some successes, usually in limited domains like audio processing or chip design, the Holy Grail still hasn't been found. In the meantime, a less ambitious approach has started gaining traction. Many recent application frameworks now use "data binding".

Unlike more radical models, data binding doesn't try to entirely eliminate imperative code and doesn't try to architect your entire application around a giant declarative dataflow graph. What it does do is automate the busywork where you're tweaking a UI element or calculated property to reflect a change to some value.

Like other declarative systems, data binding is probably a bit too slow and complex to fit inside the core of a game engine. But I would be surprised if I didn't see it start making inroads into less critical areas of the game like UI.

In the meantime, the good old Observer pattern will still be here waiting for us. Sure, it's not as exciting as some hot technique that manages to cram both "functional" and "reactive" in its name, but it's dead simple and it works. To me, those are often the two most important criteria for a solution.

# Prototype

---

`^title Prototype ^section Design Patterns Revisited`

The first time I heard the word "prototype" was in *Design Patterns*. Today, it seems like everyone is saying it, but it turns out they aren't talking about the [design pattern](#). We'll cover that here, but I'll also show you other, more interesting places where the term "prototype" and the concepts behind it have popped up. But first, let's revisit the original pattern.

I don't say "original" lightly here. *Design Patterns* cites Ivan Sutherland's legendary [Sketchpad](#) project in 1963 as one of the first examples of this pattern in the wild. While everyone else was listening to Dylan and the Beatles, Sutherland was busy just, you know, inventing the basic concepts of CAD, interactive graphics, and object-oriented programming. Watch [the demo](#) and prepare to be blown away.

## The Prototype Design Pattern

---

Pretend we're making a game in the style of Gauntlet. We've got creatures and fiends swarming around the hero, vying for their share of his flesh. These unsavory dinner companions enter the arena by way of "spawners", and there is a different spawner for each kind of enemy.

For the sake of this example, let's say we have different classes for each kind of monster in the game -- `Ghost`, `Demon`, `sorcerer`, etc., like:

`^code monster-classes`

A spawner constructs instances of one particular monster type. To support every monster in the game, we *could* brute-force it by having a spawner class for each monster class, leading to a parallel class hierarchy:



I had to dig up a dusty UML book to make this diagram. The  means "inherits from". Implementing it would look like this:

`^code spawner-classes`

Unless you get paid by the line of code, this is obviously not a fun way to hack this together. Lots of classes, lots of boilerplate, lots of redundancy, lots of duplication, lots of repeating myself...

The Prototype pattern offers a solution. The key idea is that *an object can spawn other objects similar to itself*. If you have one ghost, you can make more ghosts from it. If you have a demon, you can make other demons. Any monster can be treated as a *prototypal* monster used to generate other versions of itself.

To implement this, we give our base class, `Monster`, an abstract `clone()` method:

`^code virtual-clone`

Each monster subclass provides an implementation that returns a new object identical in class and state to itself. For example:

`^code clone-ghost`

Once all our monsters support that, we no longer need a spawner class for each monster class. Instead, we define a single one:

`^code spawner-clone`

It internally holds a monster, a hidden one whose sole purpose is to be used by the spawner as a template to stamp out

more monsters like it, sort of like a queen bee who never leaves the hive.



To create a ghost spawner, we create a prototypal ghost instance and then create a spawner holding that prototype:

`^code spawn-ghost-clone`

One neat part about this pattern is that it doesn't just clone the *class* of the prototype, it clones its *state* too. This means we could make a spawner for fast ghosts, weak ghosts, or slow ghosts just by creating an appropriate prototype ghost.

I find something both elegant and yet surprising about this pattern. I can't imagine coming up with it myself, but I can't imagine *not* knowing about it now that I do.

## How well does it work?

Well, we don't have to create a separate spawner class for each monster, so that's good. But we *do* have to implement `clone()` in each monster class. That's just about as much code as the spawners.

There are also some nasty semantic ratholes when you sit down to try to write a correct `clone()`. Does it do a deep clone or shallow one? In other words, if a demon is holding a pitchfork, does cloning the demon clone the pitchfork too?

Also, not only does this not look like it's saving us much code in this contrived problem, there's the fact that it's a *contrived problem*. We had to take as a given that we have separate classes for each monster. These days, that's definitely *not* the way most game engines roll.

Most of us learned the hard way that big class hierarchies like this are a pain to manage, which is why we instead use patterns like [Component](#) and [Type Object](#) to model different kinds of entities without enshrining each in its own class.

## Spawn functions

Even if we do have different classes for each monster, there are other ways to decorticate this *Felis catus*. Instead of making separate spawner *classes* for each monster, we could make spawner *functions*, like so:

`^code callback`

This is less boilerplate than rolling a whole class for constructing a monster of some type. Then the one spawner class can simply store a function pointer:

`^code spawner-callback`

To create a spawner for ghosts, you do:

`^code spawn-ghost-callback`

## Templates

By now, most C++ developers are familiar with templates. Our spawner class needs to construct instances of some type, but we don't want to hard code some specific monster class. The natural solution then is to make it a *type parameter*, which templates let us do:

I'm not sure if C++ programmers learned to love them or if templates just scared some people completely away from C++. Either way, everyone I see using C++ today uses templates too.

`^code templates`

Using it looks like:

`^code use-templates`

The `Spawner` class here is so that code that doesn't care what kind of monster a spawner creates can just use it and work with pointers to `Monster`. If we only had the `SpawnerFor<T>` class, there would be no single supertype the instantiations of that template all shared, so any code that worked with spawners of any monster type would itself need to take a template parameter.

## First-class types

The previous two solutions address the need to have a class, `Spawner`, which is parameterized by a type. In C++, types aren't generally first-class, so that requires some gymnastics. If you're using a dynamically-typed language like JavaScript, Python, or Ruby where classes are regular objects you can pass around, you can solve this much more directly.

In some ways, the [Type Object](#) pattern is another workaround for the lack of first-class types. That pattern can still be useful even in languages with them, though, because it lets *you* define what a "type" is. You may want different semantics than what the language's built-in classes provide.

When you make a spawner, just pass in the class of monster that it should construct -- the actual runtime object that represents the monster's class. Easy as pie.

With all of these options, I honestly can't say I've found a case where I felt the Prototype *design pattern* was the best answer. Maybe your experience will be different, but for now let's put that away and talk about something else: prototypes as a *language paradigm*.

## The Prototype Language Paradigm

---

Many people think "object-oriented programming" is synonymous with "classes". Definitions of OOP tend to feel like credos of opposing religious denominations, but a fairly non-contentious take on it is that *OOP lets you define "objects" which bundle data and code together*. Compared to structured languages like C and functional languages like Scheme, the defining characteristic of OOP is that it tightly binds state and behavior together.

You may think classes are the one and only way to do that, but a handful of guys including Dave Ungar and Randall Smith beg to differ. They created a language in the 80s called Self. While as OOP as can be, it has no classes.

### Self

In a pure sense, Self is *more* object-oriented than a class-based language. We think of OOP as marrying state and behavior, but languages with classes actually have a line of separation between them.

Consider the semantics of your favorite class-based language. To access some state on an object, you look in the memory of the instance itself. State is *contained* in the instance.

To invoke a method, though, you look up the instance's class, and then you look up the method *there*. Behavior is contained in the *class*. There's always that level of indirection to get to a method, which means fields and methods are different.



For example, to invoke a virtual method in C++, you look in the instance for the pointer to its vtable, then look up the method *there*.

Self eliminates that distinction. To look up *anything*, you just look on the object. An instance can contain both state and behavior. You can have a single object that has a method completely unique to it.



No man is an island, but this object is.

If that was all Self did, it would be hard to use. Inheritance in class-based languages, despite its faults, gives you a useful mechanism for reusing polymorphic code and avoiding duplication. To accomplish something similar without classes, Self has *delegation*.

To find a field or call a method on some object, we first look in the object itself. If it has it, we're done. If it doesn't, we look at the object's *parent*. This is just a reference to some other object. When we fail to find a property on the first object, we try its parent, and its parent, and so on. In other words, failed lookups are *delegated* to an object's parent.

I'm simplifying here. Self actually supports multiple parents. Parents are just specially marked fields, which means you can do things like inherit parents or change them at runtime, leading to what's called *dynamic inheritance*.



Parent objects let us reuse behavior (and state!) across multiple objects, so we've covered part of the utility of classes. The other key thing classes do is give us a way to create instances. When you need a new thingamabob, you can just do `new Thingamabob()`, or whatever your preferred language's syntax is. A class is a factory for instances of itself.

Without classes, how do we make new things? In particular, how do we make a bunch of new things that all have stuff in common? Just like the design pattern, the way you do this in Self is by *cloning*.

In Self, it's as if every object supports the Prototype design pattern automatically. Any object can be cloned. To make a bunch of similar objects, you:

1. Beat one object into the shape you want. You can just clone the base `Object` built into the system and then stuff fields and methods into it.
2. Clone it to make as many... uh... clones as you want.

This gives us the elegance of the Prototype design pattern without the tedium of having to implement `clone()` ourselves; it's built into the system.

This is such a beautiful, clever, minimal system that as soon as I learned about it, I started creating a prototype-based language to get more experience with it.

I realize building a language from scratch is not the most efficient way to learn, but what can I say? I'm a bit peculiar. If you're curious, the language is called [Finch](#).

## How did it go?

I was super excited to play with a pure prototype-based language, but once I had mine up and running, I discovered an unpleasant fact: it just wasn't that fun to program in.

I've since heard through the grapevine that many of the Self programmers came to the same conclusion. The project was far from a loss, though. Self was so dynamic that it needed all sorts of virtual machine innovations in order to run fast enough. The ideas they invented for just-in-time compilation, garbage collection, and optimizing method dispatch are the exact same techniques -- often implemented by the same people! -- that now make many of the world's dynamically-typed languages fast enough to use for massively popular applications.

Sure, the language was simple to implement, but that was because it punted the complexity onto the user. As soon as I started trying to use it, I found myself missing the structure that classes give. I ended up trying to recapitulate it at the library level since the language didn't have it.

Maybe this is because my prior experience is in class-based languages, so my mind has been tainted by that paradigm. But my hunch is that most people just like well-defined "kinds of things".

In addition to the runaway success of class-based languages, look at how many games have explicit character classes and a precise roster of different sorts of enemies, items, and skills, each neatly labeled. You don't see many games where each monster is a unique snowflake, like "sort of halfway between a troll and a goblin with a bit of snake mixed in".

While prototypes are a really cool paradigm and one that I wish more people knew about, I'm glad that most of us aren't actually programming using them every day. The code I've seen that fully embraces prototypes has a weird mushiness to it that I find hard to wrap my head around.

It's also telling how *little* code there actually is written in a prototypal style. I've looked.

## What about JavaScript?

OK, if prototype-based languages are so unfriendly, how do I explain JavaScript? Here's a language with prototypes used by millions of people every day. More computers run JavaScript than any other language on Earth.

Brendan Eich, the creator of JavaScript, took inspiration directly from Self, and many of JavaScript's semantics are prototype-based. Each object can have an arbitrary set of properties, both fields and "methods" (which are really just functions stored as fields). An object can also have another object, called its "prototype", that it delegates to if a field access fails.

As a language designer, one appealing thing about prototypes is that they are simpler to implement than classes. Eich took full advantage of this: the first version of JavaScript was created in ten days.

But, despite that, I believe that JavaScript in practice has more in common with class-based languages than with prototypal ones. One hint that JavaScript has taken steps away from Self is that the core operation in a prototype-based language, *cloning*, is nowhere to be seen.

There is no method to clone an object in JavaScript. The closest it has is `Object.create()`, which lets you create a new object that delegates to an existing one. Even that wasn't added until ECMAScript 5, fourteen years after JavaScript came out. Instead of cloning, let me walk you through the typical way you define types and create objects in JavaScript. You start with a *constructor function*:

```
:::javascript
function Weapon(range, damage) {
  this.range = range;
  this.damage = damage;
}
```

This creates a new object and initializes its fields. You invoke it like:

```
:::javascript
var sword = new Weapon(10, 16);
```

The `new` here invokes the body of the `Weapon()` function with `this` bound to a new empty object. The body adds a bunch of fields to it, then the now-filled-in object is automatically returned.

The `new` also does one other thing for you. When it creates that blank object, it wires it up to delegate to a prototype object. You can get to that object directly using `Weapon.prototype`.

While state is added in the constructor body, to define *behavior*, you usually add methods to the prototype object. Something like this:

```
:::javascript
Weapon.prototype.attack = function(target) {
  if (distanceTo(target) > this.range) {
    console.log("Out of range!");
  } else {
    target.health -= this.damage;
  }
}
```

This adds an `attack` property to the weapon prototype whose value is a function. Since every object returned by `new` `Weapon()` delegates to `Weapon.prototype`, you can now call `sword.attack()` and it will call that function. It looks a bit like this:



Let's review:

- The way you create objects is by a "new" operation that you invoke using an object that represents the type -- the constructor function.
- State is stored on the instance itself.
- Behavior goes through a level of indirection -- delegating to the prototype -- and is stored on a separate object that represents the set of methods shared by all objects of a certain type.

Call me crazy, but that sounds a lot like my description of classes earlier. You *can* write prototype-style code in JavaScript (*sans* cloning), but the syntax and idioms of the language encourage a class-based approach.

Personally, I think that's a good thing. Like I said, I find doubling down on prototypes makes code harder to work with, so I like that JavaScript wraps the core semantics in something a little more classy.

## Prototypes for Data Modeling

---

OK, I keep talking about things I *don't* like prototypes for, which is making this chapter a real downer. I think of this book as more comedy than tragedy, so let's close this out with an area where I *do* think prototypes, or more specifically *delegation*, can be useful.

If you were to count all the bytes in a game that are code compared to the ones that are data, you'd see the fraction of data has been increasing steadily since the dawn of programming. Early games procedurally generated almost everything so they could fit on floppies and old game cartridges. In many games today, the code is just an "engine" that drives the game, which is defined entirely in data.

That's great, but pushing piles of content into data files doesn't magically solve the organizational challenges of a large project. If anything, it makes it harder. The reason we use programming languages is because they have tools for managing complexity.

Instead of copying and pasting a chunk of code in ten places, we move it into a function that we can call by name. Instead of copying a method in a bunch of classes, we can put it in a separate class that those classes inherit from or mix in.

When your game's data reaches a certain size, you really start wanting similar features. Data modeling is a deep subject that I can't hope to do justice here, but I do want to throw out one feature for you to consider in your own games: using prototypes and delegation for reusing data.

Let's say we're defining the data model for the shameless Gauntlet rip-off I mentioned earlier. The game designers need to specify the attributes for monsters and items in some kind of files.

I mean completely original title in no way inspired by any previously existing top-down multi-player dungeon crawl arcade games. Please don't sue me.

One common approach is to use JSON. Data entities are basically *maps*, or *property bags*, or any of a dozen other terms because there's nothing programmers like more than inventing a new name for something that already has one.

We've re-invented them so many times that Steve Yegge calls them "[The Universal Design Pattern](#)".

So a goblin in the game might be defined something like this:

```
 ::=json
{
  "name": "goblin grunt",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"]
}
```

This is pretty straightforward and even the most text-averse designer can handle that. So you throw in a couple of sibling branches on the Great Goblin Family Tree:

```

:::json
{
  "name": "goblin wizard",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"],
  "spells": ["fire ball", "lightning bolt"]
}

{
  "name": "goblin archer",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"],
  "attacks": ["short bow"]
}

```

Now, if this was code, our aesthetic sense would be tingling. There's a lot of duplication between these entities, and well-trained programmers *hate* that. It wastes space and takes more time to author. You have to read carefully to tell if the data even *is* the same. It's a maintenance headache. If we decide to make all of the goblins in the game stronger, we need to remember to update the health of all three of them. Bad bad bad.

If this was code, we'd create an abstraction for a "goblin" and reuse that across the three goblin types. But dumb JSON doesn't know anything about that. So let's make it a bit smarter.

We'll declare that if an object has a `"prototype"` field, then that defines the name of another object that this one delegates to. Any properties that don't exist on the first object fall back to being looked up on the prototype.

This makes the `"prototype"` a piece of *metadata* instead of data. Goblins have warty green skin and yellow teeth. They don't have prototypes. Prototypes are a property of the *data object representing the goblin*, and not the goblin itself.

With that, we can simplify the JSON for our goblin horde:

```

:::json
{
  "name": "goblin grunt",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"]
}

{
  "name": "goblin wizard",
  "prototype": "goblin grunt",
  "spells": ["fire ball", "lightning bolt"]
}

{
  "name": "goblin archer",
  "prototype": "goblin grunt",
  "attacks": ["short bow"]
}

```

Since the archer and wizard have the grunt as their prototype, we don't have to repeat the health, resists, and weaknesses in each of them. The logic we've added to our data model is super simple -- basic single delegation -- but we've already gotten rid of a bunch of duplication.

One interesting thing to note here is that we didn't set up a fourth "base goblin" *abstract* prototype for the three concrete goblin types to delegate to. Instead, we just picked one of the goblins who was the simplest and delegated to it.

That feels natural in a prototype-based system where any object can be used as a clone to create new refined objects, and I think it's equally natural here too. It's a particularly good fit for data in games where you often have one-off special entities in the game world.

Think about bosses and unique items. These are often refinements of a more common object in the game, and prototypal delegation is a good fit for defining those. The magic Sword of Head-Detaching, which is really just a longsword with some bonuses, can be expressed as that directly:

```
:::json
{
  "name": "Sword of Head-Detaching",
  "prototype": "longsword",
  "damageBonus": "20"
}
```

A little extra power in your game engine's data modeling system can make it easier for designers to add lots of little variations to the armaments and beasties populating your game world, and that richness is exactly what delights players.

# Singleton 单件模式

---

This chapter is an anomaly. Every other chapter in this book shows you how to use a design pattern. This chapter shows you how *not* to use one.

这节有点反常。其他章节都是告诉你如何使用一个模式。本节却是告诉你如何不使用一个模式。

Despite noble intentions, the [Singleton](#) pattern described by the Gang of Four usually does more harm than good. They stress that the pattern should be used sparingly, but that message was often lost in translation to the game industry.

尽管一再告诫，在四人帮的[单件](#)模式描述中，它通常缺点大于优点。他们一再强调这个模式应当谨慎的使用，但是当应用在游戏行业时，这个强调通常被忽略了。

Like any pattern, using Singleton where it doesn't belong is about as helpful as treating a bullet wound with a splint. Since it's so overused, most of this chapter will be about *avoiding* singletons, but first, let's go over the pattern itself.

和其他模式一样，在不合适的地方使用单件模式，就像用夹板来治疗枪伤一样毫无用处。既然它已经被过度使用了，本节的大部分内容都是关于避免使用单件。不过首先，我们来看看模式本身。

>When much of the industry moved to object-oriented programming from C, one problem they ran into was "how do I get an instance?" They had some method they wanted to call but didn't have an instance of the object that provides that method in hand. Singletons (in other words, making it global) were an easy way out. >自从工业界大部分从C转向面向对象编程之后，一个摆在面前的问题就是“如何得到一个实例？”，他们有一些想要调用的方法，但是手上却没有这个对象的实例。单件(或者，使之全局化)是最简单的解决方法。

## The Singleton Pattern 单件模式

---

*Design Patterns* summarizes Singleton like this:

设计模式这样总结单件：

Ensure a class has one instance, and provide a global point of access to it.

确保一个类只有一个实例，并提供一个全局的指针访问它。

We'll split that at "and" and consider each half separately.

我们将分别讨论“并”前后的两点。

### Restricting a class to one instance 确保一个类只有一个实例

There are times when a class cannot perform correctly if there is more than one instance of it. The common case is when the class interacts with an external system that maintains its own global state.

在有些情况下，一个类如果有多个实例就不能正常运作。最常见的情况就是这个类和一些关联全局状态的额外类进行交互。

Consider a class that wraps an underlying file system API. Because file operations can take a while to complete, our class performs operations asynchronously. This means multiple operations can be running concurrently, so they must be coordinated with each other. If we start one call to create a file and another one to delete that same file, our wrapper needs to be aware of both to make sure they don't interfere with each other.

比如说一个封装了底层文件API的类。因为文件操作需要一定时间去完成，我们的类将异步地处理。这意味着许多操作可以同时进行，所以他们必须相互协调。如果我们一方面创建文件，一方面去删除这个文件，我们的封装类就必须全部感知，并确保他们不会相互干扰。

To do this, a call into our wrapper needs to have access to every previous operation. If users could freely create instances of our class, one instance would have no way of knowing about operations that other instances started. Enter the singleton. It provides a way for a class to ensure at compile time that there is only a single instance of the class.

为了做到这点，对封装类的调用必须能够访问之前的操作。如果使用者能够自由的创建这个类的实例，一个实例就不能够知道其他实例所做的操作。在单件模式中，他提供了一个编译期能确保某个类只有一个实例的方法。

## Providing a global point of access 提供一个全局指针访问

Several different systems in the game will use our file system wrapper: logging, content loading, game state saving, etc. If those systems can't create their own instances of our file system wrapper, how can they get ahold of one?

游戏中一些其他系统需要用到我们的文件系统封装：日志、文件加载、游戏保存等等。如果这些系统不能够创建他们各自的文件封装实例，他们如果去得到一个呢？

Singleton provides a solution to this too. In addition to creating the single instance, it also provides a globally available method to get it. This way, anyone anywhere can get their paws on our blessed instance. All together, the classic implementation looks like this:

单件提供了一个解决方法。除了创建一个单独的实例外，他还提供一个全局的方法去得到这个实例。这样，就能在其他任何地方都能够等到这个实例了。总体说来，这个类的实现起来像如下这个样子：

```
class FileSystem
{
public:
    static FileSystem& instance()
    {
        // Lazy initialize.
        if (instance_ == NULL) instance_ = new FileSystem();
        return *instance_;
    }

private:
    FileSystem() {}

    static FileSystem* instance_;
};
```

The static `instance_` member holds an instance of the class, and the private constructor ensures that it is the *only* one. The public static `instance()` method grants access to the instance from anywhere in the codebase. It is also responsible for instantiating the singleton instance lazily the first time someone asks for it.

`instance_` 这个静态成员保存这个类的一个实例，私有的构造函数确保他是唯一的一个实例。静态函数 `instance()` 提供了一个方法能在其他地方得到这个实例。它也负责在第一次访问的时候初始化这个实例，这也叫延时创建。

A modern take looks like this:

实现起来如下：

```
class FileSystem
{
public:
    static FileSystem& instance()
    {
        static FileSystem *instance = new FileSystem();
        return *instance;
    }

private:
    FileSystem() {}
};
```

C++11 mandates that the initializer for a local static variable is only run once, even in the presence of concurrency. So,

assuming you've got a modern C++ compiler, this code is thread-safe where the first example is not.

C++11 初始化一个局部静态变量时只会运行一次，哪怕是在多线程的情况下也是一样。所以，如果你有一个现代C++编译器的话，下面的代码是线程安全的，而上面的例子却不是：

>Of course, the thread-safety of your singleton class itself is an entirely different question! This just ensures that its *initialization* is. >当然，你的单件类本身的线程安全行完全是另外一个问题！这只是确保他的初始化是线程安全的。

## Why We Use It 为什么需要使用

It seems we have a winner. Our file system wrapper is available wherever we need it without the tedium of passing it around everywhere. The class itself cleverly ensures we won't make a mess of things by instantiating a couple of instances. It's got some other nice features too:

看起来我们取得了成效。我们的文件封装能够在任何地方使用而不必将它传递的到处都是。这个类本身机智的保证了我们不会初始化多个实例而将事情弄糟。它还具有一些额外的优良特性。

- **It doesn't create the instance if no one uses it.** Saving memory and CPU cycles is always good. Since the singleton is initialized only when it's first accessed, it won't be instantiated at all if the game never asks for it.
- 如果我们不使用，就不会创建实例 节省内存和CPU周期始终是好的。既然单件只在第一次访问的时候初始化，如果我们游戏始终不使用就不会初始化。
- **It's initialized at runtime.** A common alternative to Singleton is a class with static member variables. I like simple solutions, so I use static classes instead of singletons when possible, but there's one limitation static members have: automatic initialization. The compiler initializes statics before `main()` is called. This means they can't use information known only once the program is up and running (for example, configuration loaded from a file). It also means they can't reliably depend on each other -- the compiler does not guarantee the order in which statics are initialized relative to each other.
- 他在运行期初始化一个单件的变种是包含多个静态成员的类。我喜欢简单的解决方案，所以我多会使用静态类而不是单件。但是静态类有一个缺点：自动初始化。编译器早在 `main()` 函数调用之前就初始化静态成员了。这以为着他们不能使用只有游戏运行起来才能知道的信息（比如，文件配置）。它还意味着他们之间不能相互依赖——编译器不能保证他们之间的初始化的顺序。

Lazy initialization solves both of those problems. The singleton will be initialized as late as possible, so by that time any information it needs should be available. As long as they don't have circular dependencies, one singleton can even refer to another when initializing itself.

延时初始化解决了以上所有问题。单件会尽可能的延时创建，所以他们需要的信息都是可以得到的。只要不是循环依赖，一个单件在初始化的时候可以依赖另外一个单件。

- **You can subclass the singleton.** This is a powerful but often overlooked capability. Let's say we need our file system wrapper to be cross-platform. To make this work, we want it to be an abstract interface for a file system with subclasses that implement the interface for each platform. Here is the base class:
- 你可以继承单件 这是一个强大但是过多使用的能力。假设我们需要我们的文件封装跨平台。为了实现这一点，我们将它实现为一个抽象接口，他的子类提供各个平台上的实现。下面是基本的结构：

```
class FileSystem
{
public:
    virtual ~FileSystem() {}
    virtual char* readFile(char* path) = 0;
    virtual void writeFile(char* path, char* contents) = 0;
};
```

Then we define derived classes for a couple of platforms: 之后，我们为不同平台定义派生类：

```

class PS3FileSystem : public FileSystem
{
public:
    virtual char* readFile(char* path)
    {
        // Use Sony file IO API...
    }

    virtual void writeFile(char* path, char* contents)
    {
        // Use sony file IO API...
    }
};

class WiiFileSystem : public FileSystem
{
public:
    virtual char* readFile(char* path)
    {
        // Use Nintendo file IO API...
    }

    virtual void writeFile(char* path, char* contents)
    {
        // Use Nintendo file IO API...
    }
};

```

Next, we turn `FileSystem` into a singleton:

接下来，我们将 `FileSystem` 变为一个单件：

```

class FileSystem
{
public:
    static FileSystem& instance();

    virtual ~FileSystem() {}
    virtual char* readFile(char* path) = 0;
    virtual void writeFile(char* path, char* contents) = 0;

protected:
    FileSystem() {}
};

```

The clever part is how the instance is created:

机智之处是如何创建实例的：

```

FileSystem& FileSystem::instance()
{
#if PLATFORM == PLAYSTATION
    static FileSystem *instance = new PS3FileSystem();
#elif PLATFORM == WII
    static FileSystem *instance = new WiiFileSystem();
#endif

    return *instance;
}

```

With a simple compiler switch, we bind our file system wrapper to the appropriate concrete type. Our entire codebase can access the file system using `FileSystem::instance()` without being coupled to any platform-specific code. That coupling is instead encapsulated within the implementation file for the `FileSystem` class itself.

随着一个简单的编译跳转，我们将文件封装编译到正确的系统上。我们整个代码可以通过 `FileSystem::instance()` 来访问文件系统，而不必加上任何平台相关的代码。关联的代码封装在实现 `FileSystem` 这个类的文件之中了。

This takes us about as far as most of us go when it comes to solving a problem like this. We've got a file system wrapper. It

works reliably. It's available globally so every place that needs it can get to it. It's time to check in the code and celebrate with a tasty beverage.

它花费了我们我们之中绝大数人解决这类问题所花费的时间(译注：绝大部分人解决这类问题到此为止)。我们得到了一个文件封装。他工作的很好，它全局可用，每处需要使用的地方都能访问它。是时候提交代码，来点美味的饮料庆祝了。

## Why We Regret Using It 为什么后悔使用

---

In the short term, the Singleton pattern is relatively benign. Like many design choices, we pay the cost in the long term. Once we've cast a few unnecessary singletons into cold hard code, here's the trouble we've bought ourselves:

在短期内，单件模式是相对温和的。像其他一些设计取舍一样，我们在长时间内付出代价。一旦我们将一些不必要的单件扔到了冰硬的代码之中，我们就为自己带来了一系列的麻烦。

### It's a global variable 他是一个全局变量

When games were still written by a couple of guys in a garage, pushing the hardware was more important than ivory-tower software engineering principles. Old-school C and assembly coders used globals and statics without any trouble and shipped good games. As games got bigger and more complex, architecture and maintainability started to become the bottleneck. We struggled to ship games not because of hardware limitations, but because of *productivity* limitations.

当游戏还是车库里的借个家伙写的时候，硬件要比软件工程准则更为总要。随着游戏变得更大更复，架构和开始变为短板。我们挣扎这放弃游戏不是应为硬件限制，而是因为开发效率

So we moved to languages like C++ and started applying some of the hard-earned wisdom of our software engineer forebears. One lesson we learned is that global variables are bad for a variety of reasons:

所以我们开始学习C++这样的语言，并且应用我们软件开发前驱总结的智慧。我们学到的一个教训就是，全局变量是有害的。理由如下：

- **They make it harder to reason about code.** Say we're tracking down a bug in a function someone else wrote. If that function doesn't touch any global state, we can wrap our heads around it just by understanding the body of the function and the arguments being passed to it.
- 他们导致能更难理解的代码 假设我们正在跟踪一个bug。如果这个函数不使用全局状态，我们可以将精力集中起来，只要理解他的函数体就可以了，和传递给他的参数就可以了。

>Computer scientists call functions that don't access or modify global state "pure" functions. Pure functions are easier to reason about, easier for the compiler to optimize, and let you do neat things like memoization where you cache and reuse the results from previous calls to the function. >计算机科学家称不访问或者不修改全局状态的函数为“纯函数”。纯函数易于理解，利于编译器优化。>While there are challenges to using purity exclusively, the benefits are enticing enough that computer scientists have created languages like Haskell that *only* allow pure functions. >因为全部使用纯函数有不少挑战，但是足够诱使计算机科学家发明Haskell这样只允许存函数的语言。

Now, imagine right in the middle of that function is a call to `SomeClass::getSomeGlobalData()`. To figure out what's going on, we have to hunt through the entire codebase to see what touches that global data. You don't really hate global state until you've had to `grep` a million lines of code at three in the morning trying to find the one errant call that's setting a static variable to the wrong value.

现在，让我们来看这个函数中间的 `SomeClass::getSomeGlobalData()` 这个调用。为了搞清楚其中发生了什么，我们需要查看整个代码库来看是谁访问了全局状态。在你不得不大清晨 `grep` 百万行代码来找出究竟是那一个错误的调用将一个静态变量设置错了之前，你是不会真正痛恨全局状态的。

- **They encourage coupling.** The new coder on your team isn't familiar with your game's beautifully maintainable loosely coupled architecture, but he's just been given his first task: make boulders play sounds when they crash onto the ground. You and I know we don't want the physics code to be coupled to `audio` of all things, but he's just trying to get his task done. Unfortunately for us, the instance of our `AudioPlayer` is globally visible. So, one little `#include` later, and our new guy has compromised a carefully constructed architecture.

- 这促进了耦合。你团队的开发新手不熟悉游戏优美的松耦合架构，但是他却有了第一项任务：让石头撞在地上时发出声音。你我都知道，我们不想让物理引擎代码和音频代码耦合起来，但是新手只是一心想完成任务。不幸的是，我们的 `AudioPlayer` 这个类实例是全局可见的。所以，在一小段 `#include` 之后，我们的新伙伴搞乱了一个仔细构建的架构。

Without a global instance of the audio player, even if he *did #include* the header, he still wouldn't be able to do anything with it. That difficulty sends a clear message to him that those two modules should not know about each other and that he needs to find another way to solve his problem. *By controlling access to instances, you control coupling.*

如果没有音频播放器的全局实例，即使他确实 `#include` 头文件，他也不能做任何事情。这个困难度给他传递了一个明确的消息，这两个模块不应该相互了解，他应该找另外的方式去解决这个问题。通过控制实例的访问，你控制了耦合。

- **They aren't concurrency-friendly.** The days of games running on a simple single-core CPU are pretty much over. Code today must at the very least *work* in a multi-threaded way even if it doesn't take full advantage of concurrency. When we make something global, we've created a chunk of memory that every thread can see and poke at, whether or not they know what other threads are doing to it. That path leads to deadlocks, race conditions, and other hell-to-fix thread-synchronization bugs.
- 它对并发不友好现在在单核上运行游戏的日子已经很远了。即使他们有利用到并发的全部优势。当我们设置为全局时，我们创建了一段内存，每个线程都能够查看和修改它，不管他们时候知道其他线程正在操作它。这有可能导致死锁，条件竞争，和其他一些难以修复的线程同步的Bug。

Issues like these are enough to scare us away from declaring a global variable, and thus the Singleton pattern too, but that still doesn't tell us how we *should* design the game. How do you architect a game without global state?

上面这些问题足够吓退我们去声明一个全局变量了，同样也适用于单件模式，但是现在还是没有告诉我们该如何设计游戏。在没有全局状态的情况下，该如何构建游戏呢？

There are some extensive answers to that question (most of this book in many ways *is* an answer to just that), but they aren't apparent or easy to come by. In the meantime, we have to get games out the door. The Singleton pattern looks like a panacea. It's in a book on object-oriented design patterns, so it *must* be architecturally sound, right? And it lets us design software the way we have been doing for years.

这个问题有几个拓展的答案(本书的绝大部分从某些方面来说就是这个)，但是他们不是和明显或者简单能够得到。与此同时，我们需要发布我们的游戏。单件模式就像一帖万能药。他在一本关于面向对象设计模式中，所以它肯定是架构合理的，对吧？并且他能像之前我们开发了N年那样去设计软件。

Unfortunately, it's more placebo than cure. If you scan the list of problems that globals cause, you'll notice that the Singleton pattern doesn't solve any of them. That's because a singleton *is* global state -- it's just encapsulated in a class.

不幸的是，它更多的是一种安慰而不是解决方法。如果你浏览一边全局对象造成的问题，你会注意到单件模式没有解决任何一个。这是因为，一个单件就是全局状态——他只是被封装到了一个类中而已。

## **It solves two problems even when you just have one** 即便你只有一个问题，它却解决了两个

The word "and" in the Gang of Four's description of Singleton is a bit strange. Is this pattern a solution to one problem or two? What if we have only one of those? Ensuring a single instance is useful, but who says we want to let everyone poke at it? Likewise, global access is convenient, but that's true even for a class that allows multiple instances.

在四人帮的单件模式中那个“和”这个词有点奇怪。这个模式解决的是一个问题还是两个问题？如果我们只用其中的一个问题怎么办？确保一个实例是很有用的，但是谁说我们需要所有的东西都像这样？就好比，全局访问是很方便，但是允许有多个实例却是很常见的。

The latter of those two problems, convenient access, is almost always why we turn to the Singleton pattern. Consider a logging class. Most modules in the game can benefit from being able to log diagnostic information. However, passing an instance of our `Log` class to every single function clutters the method signature and distracts from the intent of the code.

这两个问题的后者，便利的访问，是我们使用单件模式的主要原因。比如一个日志类。许多游戏中的模块都能够从日子模块

中获得好处，但是，

The obvious fix is to make our `Log` class a singleton. Every function can then go straight to the class itself to get an instance. But when we do that, we inadvertently acquire a strange little restriction. All of a sudden, we can no longer create more than one logger.

很显然，修正这点就是让我们的 `Log` 变为一个单件。每个函数都能直接通过这个类得到这个类的实例。但是当我们这样做是，我们奇怪的得到了一个限制。突然的，我们不能够创建更多的日志器了。

At first, this isn't a problem. We're writing only a single log file, so we only need one instance anyway. Then, deep in the development cycle, we run into trouble. Everyone on the team has been using the logger for their own diagnostics, and the log file has become a massive dumping ground. Programmers have to wade through pages of text just to find the one entry they care about.

起初，这并不是一个问题，我们只写一个日志文件，所以我们只需要一个日志实例。之后，随着开发的深入，我们陷入了麻烦。团队的每个人都使用这个日志器来记录他们自己的日志。

We'd like to fix this by partitioning the logging into multiple files. To do this, we'll have separate loggers for different game domains: online, UI, audio, gameplay. But we can't. Not only does our `Log` class no longer allow us to create multiple instances, that design limitation is entrenched in every single call site that uses it:

我们可以通过将日志分割为不同的文件来修正。我们将日志分为不同的游戏区域：在线、界面、音频、游戏。但是我们不能够。不仅仅是应为我们的 `Log` 类不允许我们创建多个实例，还有这个模式的每个单次调用都是像如下这样使用的。

```
Log::instance().write("Some event.");
```

In order to make our `Log` class support multiple instantiation (like it originally did), we'll have to fix both the class itself and every line of code that mentions it. Our convenient access isn't so convenient anymore.

为了是我们的 `Log` 类能够支持多个初始化（想他原来的那样）。我们需要修改这个类的本身和每处调用这个类的地方。我们便利的访问也不那么便利了。

>It could be even worse than this. Imagine your `Log` class is in a library being shared across several games. Now, to change the design, you'll have to coordinate the change across several groups of people, most of whom have neither the time nor the motivation to fix it. >情况也许会比这样更为糟糕。假如你的 `Log` 在多个游戏共享的一个库文件中。现在，修改设计，你需要考虑的不同团队的人，他们之中的大部分人都没有时间也没有动机去修改它。

## Lazy initialization takes control away from you 延迟初始化脱离了你的控制

In the desktop PC world of virtual memory and soft performance requirements, lazy initialization is a smart trick. Games are a different animal. Initializing a system can take time: allocating memory, loading resources, etc. If initializing the audio system takes a few hundred milliseconds, we need to control when that's going to happen. If we let it lazy-initialize itself the first time a sound plays, that could be in the middle of an action-packed part of the game, causing visibly dropped frames and stuttering gameplay.

为了满足PC游戏内存和软件效率的需求，延时实例化是一个聪明的技巧。游戏是个不同的怪兽。实例化一个系统需要花费时间：分配内存，加载资源等等。如果实例化音频系统需要花费几百毫秒，我们需要控制住何时实例化。如果我们让他在第一次播放声音的时候延时实例化，这有可能在游戏正酣的时候，导致明显的掉帧和游戏卡顿。

Likewise, games generally need to closely control how memory is laid out in the heap to avoid fragmentation. If our audio system allocates a chunk of heap when it initializes, we want to know *when* that initialization is going to happen, so that we can control *where* in the heap that memory will live.

同样的，游戏通常需要仔细的控制内存存在堆中的布局来防止分段。如果我们的音频系统在实例化时分配了内存，我们需要知道实例化发生的时间，以便让我们控制它在堆中的内存布局。

>See [Object Pool](#) for a detailed explanation of memory fragmentation. >查看 [对象池](#) 活的内存分段的详细解释。

Because of these two problems, most games I've seen don't rely on lazy initialization. Instead, they implement the Singleton pattern like this:

介于这两点问题，我见过的大部分游戏都不依赖延时初始化。相反，他们想这样实现单件模式。

```
class Filesystem
{
public:
    static FileSystem& instance() { return instance_; }

private:
    FileSystem() {}

    static FileSystem instance_;
};
```

That solves the lazy initialization problem, but at the expense of discarding several singleton features that *do* make it better than a raw global variable. With a static instance, we can no longer use polymorphism, and the class must be constructible at static initialization time. Nor can we free the memory that the instance is using when not needed.

这解决的延时初始化的问题，但是这也丢失了单件比一个全局变量更好的几个特性。随着一个静态实例，我们不能够使用多态了，并且这个类必须能够在静态初始化的时候构造。我们也不能够在不需要这个类的时候释放这段内存。

Instead of creating a singleton, what we really have here is a simple static class. That isn't necessarily a bad thing, but if a static class is all you need, why not get rid of the `instance()` method entirely and use static functions instead? Calling `Foo::bar()` is simpler than `Foo::instance().bar()`, and also makes it clear that you really are dealing with static memory.

与创建单件相反，这里我们真正需要的是一个静态类。这不完全是一件坏事，如果你想要的仅仅是静态类，何不消除 `instance()` 这个方法而使用简单函数呢？调用 `Foo::bar()` 要比 `Foo::instance().bar()` 简单不说，还能澄清你正在使用静态内存。

The usual argument for choosing singletons over static classes is that if you decide to change the static class into a non-static one later, you'll need to fix every call site. In theory, you don't have to do that with singletons because you could be passing the instance around and calling it like a normal instance method. 通常关于静态类和单件的争论是，如果之后你决定将一个静态类转变为非静态类，你必须修改没处调用的地方。理论上，对于单件，你可以不必这样做，因为你可以将实例相互传递并且像一个普通实例一样去调用。In practice, I've never seen it work that way. Everyone just does `Foo::instance().bar()` in one line. If we changed Foo to not be a singleton, we'd still have to touch every call site. Given that, I'd rather have a simpler class and a simpler syntax to call into it. 在实践中，我从没有见过这么做过。每个人都 是 `Foo::instance().bar()` 这样调用的。如果我们将 `Foo` 改为非单件，我们必须修改每处调用的地方。有鉴于此，我更倾向于 使用一个简单的类和一个简单的语法去调用它。

## What We Can Do Instead 有何替代

If I've accomplished my goal so far, you'll think twice before you pull Singleton out of your toolbox the next time you have a problem. But you still have a problem that needs solving. What tool *should* you pull out? Depending on what you're trying to do, I have a few options for you to consider, but first...

如果现在我完成了目标，在下次你遇到问题时，在你祭出单件大法是会多考虑两次。但你还有一个问题有待解决。你需要什么样的工具？这要取决于你想要做什么，我有几个建议可供参考，不过首先...

### See if you need the class at all 看你究竟是否需要类

Many of the singleton classes I see in games are "managers" -- those nebulous classes that exist just to babysit other objects. I've seen codebases where it seems like every class has a manager: Monster, MonsterManager, Particle, ParticleManager, Sound, SoundManager, ManagerManager. Sometimes, for variety, they'll throw a "System" or "Engine" in there, but it's still the same idea.

游戏中的许多单件类都是"managers"——这些保姆类存在就是为了管理其他对象。我见识过一个代码库，里面好像每个类都

有一个管理者:Monster, MonsterManager, Particle, ParticleManager, Sound, SoundManager, ManagerManager。有时，为了区别，他们叫做"System'或者"Engine",却是换汤不换药。

While caretaker classes are sometimes useful, often they just reflect unfamiliarity with OOP. Consider these two contrived classes:

尽管保姆类有时是有用的，通常这反应他们对OOP不熟悉。考虑这两个我构造的类：

```
class Bullet
{
public:
    int getX() const { return x_; }
    int getY() const { return y_; }

    void setX(int x) { x_ = x; }
    void setY(int y) { y_ = y; }

private:
    int x_, y_;
};

class BulletManager
{
public:
    Bullet* create(int x, int y)
    {
        Bullet* bullet = new Bullet();
        bullet->setX(x);
        bullet->setY(y);

        return bullet;
    }

    bool isOnScreen(Bullet& bullet)
    {
        return bullet.getX() >= 0 &&
               bullet.getX() < SCREEN_WIDTH &&
               bullet.getY() >= 0 &&
               bullet.getY() < SCREEN_HEIGHT;
    }

    void move(Bullet& bullet)
    {
        bullet.setX(bullet.getX() + 5);
    }
};
```

Maybe this example is a bit dumb, but I've seen plenty of code that reveals a design just like this after you scrape away the crusty details. If you look at this code, it's natural to think that `BulletManager` should be a singleton. After all, anything that has a `Bullet` will need the manager too, and how many instances of `BulletManager` do you need?

或许这个例子有点愚蠢，如果你查看这段代码，将 `BulletManager` 当作单件是很自然的事情。毕竟，一个 `Bullet` 需要用一个东西来管理，而你需要有多个管理器呢？

The answer here is zero, actually. Here's how we solve the "singleton" problem for our manager class:

答案是零，实际上，我们是这样解决我们管理类的"单件"问题的：

```
class Bullet
{
public:
    Bullet(int x, int y) : x_(x), y_(y) {}

    bool isOnScreen()
    {
        return x_ >= 0 && x_ < SCREEN_WIDTH &&
               y_ >= 0 && y_ < SCREEN_HEIGHT;
    }

    void move() { x_ += 5; }
```

```
private:  
    int x_, y_;  
};
```

There we go. No manager, no problem. Poorly designed singletons are often "helpers" that add functionality to another class. If you can, just move all of that behavior into the class it helps. After all, OOP is about letting objects take care of themselves.

就这样。没有管理器也没有问题。错误的设计单件通常会“帮助”你将功能添加到别的类中。如果可以，你只需将这些功能移动到它帮助的类中去就可以了。比较，面向对象就是让对象自己管理自己。

Outside of managers, though, there are other problems where we'd reach to Singleton for a solution. For each of those problems, there are some alternative solutions to consider.

除了管理器，毕竟，这里还有别的问题我们需要求助单件模式去解决。对于这些问题，这里有一些额外的解决方案可供考虑。

## To limit a class to a single instance 限制类只有一个实例

This is one half of what the Singleton pattern gives you. As in our file system example, it can be critical to ensure there's only a single instance of a class. However, that doesn't necessarily mean we also want to provide *public, global* access to that instance. We may want to restrict access to certain areas of the code or even make it private to a single class. In those cases, providing a public global point of access weakens the architecture.

这是单件模式给你解决的另外一个问题。在我们的文件系统例子中，保证这个类只有一个实例是很有必要的。但是，这不意味着我们也想提供这个实例公共的全局的访问。我们也许想要严格限制在某一部分代码中，或者干脆将它作为一个类的私有成员。在这种情况下，提供一个全局的指针访问削弱了整体框架。

For example, we may be wrapping our file system wrapper inside *another* layer of abstraction. 比如，我们可以将我们的文件系统包装在另外一个抽象层中。

We want a way to ensure single instantiation *without* providing global access. There are a couple of ways to accomplish this. Here's one:

我们提供一种方法来保证单个实例，并且不提供全局访问。这里有几种方法可以达到这点，下面就是一例：

```
class FileSystem  
{  
public:  
    FileSystem()  
    {  
        assert(!instantiated_);  
        instantiated_ = true;  
    }  
  
    ~FileSystem() { instantiated_ = false; }  
  
private:  
    static bool instantiated_;  
};  
  
bool FileSystem::instantiated_ = false;
```

This class allows anyone to construct it, but it will assert and fail if you try to construct more than one instance. As long as the right code creates the instance first, then we've ensured no other code can either get at that instance or create their own. The class ensures the single instantiation requirement it cares about, but it doesn't dictate how the class should be used.

这个类允许所有人创建它，但是如果你想要创建操作一个实例，它会断言并且失败。只要代码创建了第一个实例，我们保证其他代码要么得到这个实例要么创建一个自己的实例。这个类保证了它的单个实例，但是它不能保证这个类如何使用。

>An *assertion* function is a way of embedding a contract into your code. When `assert()` is called, it evaluates the

expression passed to it. If it evaluates to `true`, then it does nothing and lets the game continue. If it evaluates to `false`, it immediately halts the game at that point. In a debug build, it will usually bring up the debugger or at least print out the file and line number where the assertion failed. >一个断言函数就是在我们代码中嵌入一份契约。当 `assert()` 调用时，他计算传递给它的表达式。当表达式为 `true` 时，它什么都不做，并让游戏继续。当表达式为 `false` 时，它在此处立刻挂断游戏。在一个debug构建中，它通常会启动调试器或者至少将断言失败的文件名和行号打印出来。>An `assert()` means, "I assert that this should always be true. If it's not, that's a bug and I want to stop now so you can fix it." This lets you define contracts between regions of code. If a function asserts that one of its arguments is not `NULL`, that says, "The contract between me and the caller is that I will not be passed `NULL`." >一个 `assert()` 意味着，“我确保这个应该始终为true，如果不是，这就是一个bug，并且我想立刻停止以便让我修复它。”这可以让你在代码区间定义约定。如果一个函数断言它的某个参数不为 `NULL`，也就是说，“函数和调用者之间的契约就是不能够传递 `NULL`。”>Assertions help us track down bugs as soon as the game does something unexpected, not later when that error finally manifests as something visibly wrong to the user. They are fences in your codebase, corralling bugs so that they can't escape from the code that created them. >断言帮助我们在游戏中做一些未预料的事情时离开开始追踪bug，而不是等到错误体现在用户可见的错误上。它们是代码库的围栏，以防bug在发生的代码之处逃离出去。

The downside with this implementation is that the check to prevent multiple instantiation is only done at *runtime*. The Singleton pattern, in contrast, guarantees a single instance at compile time by the very nature of the class's structure.

这份实现的不足之处在于它只在运行期检测来防止多个实例。单件模式，相反的，在编译期就通过类结构自然的保证了单个实例。

## To provide convenient access to an instance 提供一个便利的方法访问实例

Convenient access is the main reason we reach for singletons. They make it easy to get our hands on an object we need to use in a lot of different places. That ease comes at a cost, though -- it becomes equally easy to get our hands on the object in places where we *don't* want it being used.

便利的访问是我们使用单件的主要原因。它让我们在许多不同地方得到一个对象变得简单。这种便利也有代价，——它也是我们在不想使用的地方也可以轻松的得到这个对象。

The general rule is that we want variables to be as narrowly scoped as possible while still getting the job done. The smaller the scope an object has, the fewer places we need to keep in our head while we're working with it. Before we take the shotgun approach of a singleton object with *global* scope, let's consider other ways our codebase can get access to an object:

通用的原则是，在保证功能的情况下将变量限制在一个狭窄的范围内。对象的作用域越小，我们需要用到它的地方就越少。在我们直接了当的通过全局作用域来访问一个单件对象时，让我们考察一下我们代码访问一个对象的其他方式：

- **Pass it in.** The simplest solution, and often the best, is to simply pass the object you need as an argument to the functions that need it. It's worth considering before we discard it as too cumbersome.
- 传递进去 最简单，通常也是最好的方法就是简单的将这个对象当作一个参数传递给需要他的函数。

>Some use the term "dependency injection" to refer to this. Instead of code reaching *out* and finding its dependencies by calling into something global, the dependencies are pushed *in* to the code that needs it through parameters. Others reserve "dependency injection" for more complex ways of providing dependencies to code. >有些人使用术语“依赖注入”来指代这点。与在外部通过调用全局对象来查找依赖不同，依赖通过参数传递到需要的代码“里面”。其他通过储备“依赖注入”来为代码依赖提供更复杂的方式。

Consider a function for rendering objects. In order to render, it needs access to an object that represents the graphics device and maintains the render state. It's very common to simply pass that in to all of the rendering functions, usually as a parameter named something like `context`.

假设一个渲染物体的函数。为了渲染，他需要访问这个物体的图形设备的表象并维持渲染状态。简单地将他们全部传递到渲染函数中是很普遍的做法，通常这个参数叫做 `context`。

On the other hand, some objects don't belong in the signature of a method. For example, a function that handles AI may need to also write to a log file, but logging isn't its core concern. It would be strange to see `Log` show up in its argument list, so for cases like that we'll want to consider other options.

另一方面，一个对象不属于某个函数的签名。举个例子，一个操作AI的函数可能也需要写一个日志文件，但是记录日志并不是它主要关系的东西。在它的参数列表中发现有 `Log` 会很奇怪，所以为了这些情况，我们需要参考其他方法。

>The term for things like logging that appear scattered throughout a codebase is "cross-cutting concern". Handling cross-cutting concerns gracefully is a continuing architectural challenge, especially in statically typed languages. >描述想日志这种分散的出现在代码库的术语称为“横切关注点”。优雅的处理横切关注点是可持续架构的挑战。尤其是在静态类型语言中。>[Aspect-oriented programming](#) was designed to address these concerns. >[面向方面程序设计](#)就是设计用来解决这些问题。

- **Get it from the base class.** Many game architectures have shallow but wide inheritance hierarchies, often only one level deep. For example, you may have a base `GameObject` class with derived classes for each enemy or object in the game. With architectures like this, a large portion of the game code will live in these "leaf" derived classes. This means that all these classes already have access to the same thing: their `GameObject` base class. We can use that to our advantage:
- 在基类中访问它。许多游戏架构有浅层次但是广泛的继承，通常只有一层继承。举个例子，你可能有一个 `GameObject` 基类，每个地方或者游戏物体都派生只这个类。有了这样的架构，游戏代码的绝大部分都在资额些派生类的“叶子”上。这意味着所有这些类都能访问同样的东西：他们的 `GameObject` 基类。我们可以利用这点：

```
class GameObject
{
protected:
    Log& getLog() { return log_; }

private:
    static Log& log_;
};

class Enemy : public GameObject
{
    void doSomething()
    {
        getLog().write("I can log!");
    }
};
```

This ensures nothing outside of `GameObject` has access to its `Log` object, but every derived entity does using `getLog()`. This pattern of letting derived objects implement themselves in terms of protected methods provided to them is covered in the [Subclass Sandbox](#) chapter.

这保证了在 `GameObject` 之外没有访问 `Log` 对象的代码，但是每个派生类能够通过 `getLog()` 访问。这种让派生类在所提供的保护方法中提供实现的模式在 [子类沙盒](#) 章节中讨论。

This raises the question, "how does `GameObject` get the `Log` instance?" A simple solution is to have the base class simply create and own a static instance.

这提出了新的问题。“`GameObject` 如何得到 `Log` 实例？”一个简单的方案是，将基类创建出来，并拥有一个自己的实例。

If you don't want the base class to take such an active role, you can provide an initialization function to pass it in or use the [Service Locator](#) pattern to find it.

如果我们不想让基类承当这个角色，你可以提供一个初始化函数将它传递进去，或者使用 [服务定位器](#) 模式来得到它。

```
</aside>
```

- **Get it from something already global.** The goal of removing *all* global state is admirable, but rarely practical. Most codebases will still have a couple of globally available objects, such as a single `Game` or `World` object representing the entire game state.
- 通过其他全局对象访问它。将所有全局状态都移除令人敬佩，但是不够实际。许多代码库仍然有一些全局对象，比如一

个单独的 `Game` 或者 `World` 对象来代表整个游戏状态。

We can reduce the number of global classes by piggybacking on existing ones like that. Instead of making singletons out of `Log`, `FileSystem`, and `AudioPlayer`, do this: 我们能够通过打包到一个已知的全局对象类中来减少全局对象的数量。与将 `Log`, `FileSystem`, 和 `AudioPlayer` 变为单件不同：

```
class Game
{
public:
    static Game& instance() { return instance_; }

    // Functions to set log_, et. al. ...

    Log& getLog() { return *log_; }
    FileSystem& getFileSystem() { return *fileSystem_; }
    AudioPlayer& getAudioPlayer() { return *audioPlayer_; }

private:
    static Game instance_;

    Log *log_;
    FileSystem *fileSystem_;
    AudioPlayer *audioPlayer_;
};
```

With this, only `Game` is globally available. Functions can get to the other systems through it:

通过这点，只有 `Game` 是全局可见的。函数能够通过这个来访问其他系统：

```
Game::instance().getAudioPlayer().play(VERYLOUD_BANG);
```

>Purists will claim this violates the Law of Demeter. I claim that's still better than a giant pile of singletons. >纯粹主义者会声称这违反了迪米特法则。我坚持这仍然要比一对单件要好。

If, later, the architecture is changed to support multiple `Game` instances (perhaps for streaming or testing purposes), `Log`, `FileSystem`, and `AudioPlayer` are all unaffected -- they won't even know the difference. The downside with this, of course, is that more code ends up coupled to `Game` itself. If a class just needs to play sound, our example still requires it to know about the world in order to get to the audio player.

如果，随后，架构会变得支持多个 `Game` 实例(也许是为流或者测试目的)，`Log`, `FileSystem` 和 `AudioPlayer` 都不会影响。——他们甚至不知道任何不同。这个副作用，当然，就是更多的代码耦合在了 `Game` 当中。如果一个类只是为了播放声音，我们的例子仍然需要知道全部，以便能够得到声音播放器。

We solve this with a hybrid solution. Code that already knows about `Game` can simply access `AudioPlayer` directly from it. For code that doesn't, we provide access to `AudioPlayer` using one of the other options described here.

我们通过一个混合方案解决这点。如果代码已经知道了 `Game` 就直接通过它来访问 `AudioPlayer`。如果代码不知道，我们通过这里讨论的其他方法来访问 `AudioPlayer`。

- **Get it from a Service Locator.** So far, we're assuming the global class is some regular concrete class like `Game`. Another option is to define a class whose sole reason for being is to give global access to objects. This common pattern is called a [Service Locator](#) and gets its own chapter.
- 通过服务定位器来访问。到现在位置，我们假设全局类就是像 `Game` 那样的具体类。另外一个选择就是定义一个类专门用来给对象做全局访问。这个模式被称之为 [服务定位器](#) 并有单独的章节。

## What's Left for Singleton 剩下的问题

The question remains, where *should* we use the real Singleton pattern? Honestly, I've never used the full Gang of Four implementation in a game. To ensure single instantiation, I usually simply use a static class. If that doesn't work, I'll use a

static flag to check at runtime that only one instance of the class is constructed.

我们还有一个问题，我们什么情况下使用真正的单件呢？老实说，我从来没有在游戏中使用四人帮的全部实现。为了简单原则，我一般简单的使用一个静态类。如果这不能够满足，我将会使用一个静态的标志来在运行期检查只有一个类被创建了。

There are a couple of other chapters in this book that can also help here. The [Subclass Sandbox](#) pattern gives instances of a class access to some shared state without making it globally available. The [Service Locator](#) pattern *does* make an object globally available, but it gives you more flexibility with how that object is configured.

本书的一些其他章节也会有所帮助。沙箱模式能够提供一些共享状态的访问指针而不必全局可见。本地模式是的一个对象全局可见，但是他给你这个物体更多弹性的配置。

# 状态模式

Confession time: I went a little overboard and packed way too much into this chapter. It's ostensibly about the State design pattern, but I can't talk about that and games without going into the more fundamental concept of finite state machines (or "FSMs"). But then once I went there, I figured I might as well introduce hierarchical state machines and pushdown automata. 交待时间：我有点走远了，我往本章里面添加了太多东西。表面上这一章是介绍状态模式的，但是我不能抛开游戏里面的FSM（有限状态机）而单独只谈“状态模式”。不过，当我讲到FSM的时候，我发现我还有必要再介绍一下层级状态机(hierarchical state machine)和下推自动机(pushdown automata).

That's a lot to cover, so to keep things as short as possible, the code samples here leave out a few details that you'll have to fill in on your own. I hope they're still clear enough for you to get the big picture. 因为有太多东西需要讲，所以，我试图压缩本章的内容。本章中的代码片断没有涉及很细节的东西，所以，这些省略的部分需要靠读者自己脑补一下。我希望它们还是足够清楚到能让你掌握关键点 (big picture) .

Don't feel sad if you've never heard of a state machine. While well known to AI and compiler hackers, they aren't that familiar to other programming circles. I think they should be more widely known, so I'm going to throw them at a different kind of problem here. 如果你从未听说过状态机，也不要感到沮丧。它们对于人工智能领域和编译器黑客来讲非常熟悉，不过对于其它编程领域的人可能不是那么被人熟知了。我觉得它们应该被更多的人所了解，因此，我将通过一个新的问题领域的应用来介绍它。

## We've All Been There

我们曾经相遇过 We're working on a little side-scrolling platformer. Our job is to implement the heroine that is the player's avatar in the game world. That means making her respond to user input. Push the B button and she should jump. Simple enough: 假如我们现在正在开发一款横版游戏。我们的任务是实现女主角--即我们游戏世界里面的主人翁。我们需要根据玩家的输入来控件主角的行为。当按下B键的时候，她应该跳跃。我们可以这样实现：

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        yVelocity_ = JUMP_VELOCITY;
        setGraphics(IMAGE_JUMP);
    }
}
```

Spot the bug? 找找看，bug在哪里？

There's nothing to prevent "air jumping"—keep hammering B while she's in the air, and she will float forever. The simple fix is to add an isJumping\_ Boolean field to Heroine that tracks when she's jumping, and then do: 我们没有防止主角“在空中跳跃”--当主角跳起来后持续按下B键。这样会导致她一直飘在空中，简单地修复方法可以是：添加一个 isJumping布尔值变量。当主角跳起来后，就把该变量设置为True.只有当该变量为False时，才让主角跳跃。

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        if (!isJumping_)
        {
            isJumping_ = true;
            // Jump...
        }
    }
}
```

Next, we want the heroine to duck if the player presses down while she's on the ground and stand back up when the button is released: 接下来，我们想实现主角的闪避动作。当主角站在地面上的时候，如果玩家按下方向“下键”，则躲避，如果松开

此键，则起立。

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        // Jump if not jumping...
    }
    else if (input == PRESS_DOWN)
    {
        if (!isJumping_)
        {
            setGraphics(IMAGE_DUCK);
        }
    }
    else if (input == RELEASE_DOWN)
    {
        setGraphics(IMAGE_STAND);
    }
}
```

Spot the bug this time? 找找看，有bug在哪里？

With this code, the player could: 通过上面的代码，玩家可以：

1. Press down to duck.
2. 按下键->闪避
3. Press B to jump from a ducking position.
4. 按B键从闪避的状态直接跳起来
5. Release down while still in the air.
6. 玩家还在空中的时候松开下键

The heroine will switch to her standing graphic in the middle of the jump. Time for another flag... 此时，当女主角在跳跃的状态的时候，显示的是站立的图像。是时候添加另外一个flag来解决该问题了...

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        if (!isJumping_ && !isDucking_)
        {
            // Jump...
        }
    }
    else if (input == PRESS_DOWN)
    {
        if (!isJumping_)
        {
            isDucking_ = true;
            setGraphics(IMAGE_DUCK);
        }
    }
    else if (input == RELEASE_DOWN)
    {
        if (isDucking_)
        {
            isDucking_ = false;
            setGraphics(IMAGE_STAND);
        }
    }
}
```

Next, it would be cool if the heroine did a dive attack if the player presses down in the middle of a jump: 接下来，如果我们的主角可以在跳起来的时候按Down键进行一次俯冲攻击那就太帅了，代码如下：

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
```

```

{
    if (!isJumping_ && !isDucking_)
    {
        // Jump...
    }
    else if (input == PRESS_DOWN)
    {
        if (!isJumping_)
        {
            isDucking_ = true;
            setGraphics(IMAGE_DUCK);
        }
        else
        {
            isJumping_ = false;
            setGraphics(IMAGE_DIVE);
        }
    }
    else if (input == RELEASE_DOWN)
    {
        if (isDucking_)
        {
            // Stand...
        }
    }
}

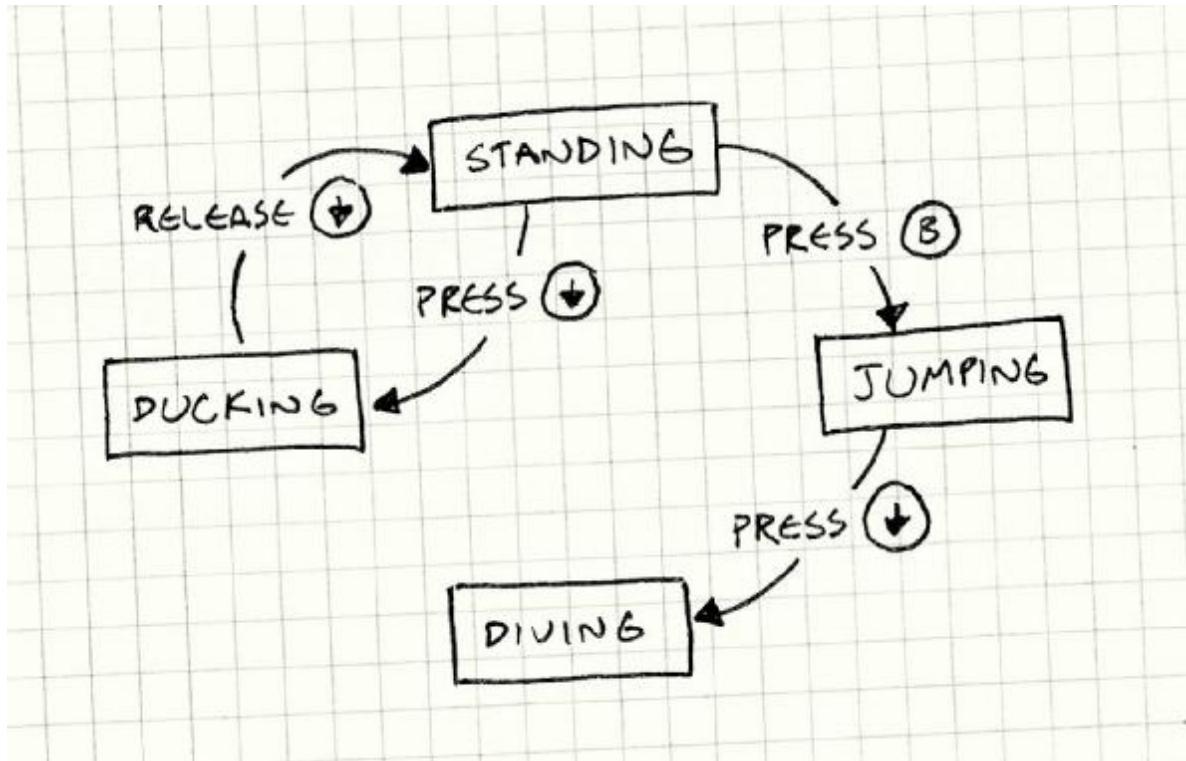
```

Bug hunting time again. Find it? 又是Bug解决时间到了。找到了吗？

We check that you can't air jump while jumping, but not while diving. Yet another field... 我们需要添加一个判断，让主角在跳跃状态的时候不能再跳，但是在俯冲攻击的时候是可以跳跃的。又要添加一个成员变量。。。

Something is clearly wrong with our approach. Every time we touch this handful of code, we break something. We need to add a bunch more moves—we haven't even added walking yet—but at this rate, it will collapse into a heap of bugs before we're done with it. 很明显，我们的这种做法有问题。每次我们添加一些功能的时候，都会不经意地破坏已有代码的功能。而且，我们还没有添加“行走”的状态。如果我们还是采用类似的做法，那bug可能会更多。

## Finite State Machines to the Rescue



In a fit of frustration, you sweep everything off your desk except a pen and paper and start drawing a flowchart. You draw a box for each thing the heroine can be doing: standing, jumping, ducking, and diving. When she can respond to a button press in one of those states, you draw an arrow from that box, label it with that button, and connect it to the state she changes to. 为了消除你心中的疑惑，你可以准备一张纸和一支笔，让我们一起来画一张流程图。对于，女主角能够进行的动作画一个“矩形”：站立、跳跃、躲避和俯冲。当你可以按下一个键让主角从一个状态切换到另一个状态的时候，我们画一个箭头，让它从一个矩形指向另一个矩形。同时在箭头上面添加文本，表示我们按下的按钮。

Congratulations, you've just created a finite state machine. These came out of a branch of computer science called automata theory whose family of data structures also includes the famous Turing machine. FSMs are the simplest member of that family. 恭喜，你刚刚已经成功创建了一个有限状态机。FSM是借鉴了计算机科学里的自动机理论（automata theory）中的一种数据结构(图灵机)的思想。FSM可以看作是最简单的图灵机。

The gist is: 这个FSM表达的是：

- You have a fixed set of states that the machine can be in. For our example, that's standing, jumping, ducking, and diving.
- 你拥有一组状态，并且可以在这组状态之间进行切换。比如：站立、跳跃、躲避和俯冲。
- The machine can only be in one state at a time. Our heroine can't be jumping and standing simultaneously. In fact, preventing that is one reason we're going to use an FSM.
- 这个机器一次只能处于一种状态。
- A sequence of inputs or events is sent to the machine. In our example, that's the raw button presses and releases. 有一组输入或者事件发送给状态机。在我们这个例子中，它们就是按钮的按下和释放。

Each state has a set of transitions, each associated with an input and pointing to a state. When an input comes in, if it matches a transition for the current state, the machine changes to the state that transition points to. 每一个状态有一组转换，每一个转换都关联着一个输入并指向另一个状态。当有一个输入进来的时候，如果有转换与此状态和输入事件对应，则状态机便会转换状态到输入事件所指的状态。

For example, pressing down while standing transitions to the ducking state. Pressing down while jumping transitions to diving. If no transition is defined for an input on the current state, the input is ignored. 在我们的例子中，在站立状态的时候如果按下down键，则状态转换到躲避状态。如果在跳跃状态的时候按下down键，则会转换到俯冲攻击状态。如果对于每一个输入事件没有对应的转换，则这个转入会被忽略。

In their pure form, that's the whole banana: states, inputs, and transitions. You can draw it out like a little flowchart. Unfortunately, the compiler doesn't recognize our scribbles, so how do we go about implementing one? The Gang of Four's State pattern is one method—which we'll get to—but let's start simpler. 简而言之，整个状态机可以分为：状态，输入和转换。你可以通过画状态流程图来表示它们。但是，我们的编译器并不认识状态图，所以，我们接下来要介绍一个实现。四人帮(Gof)的状态模式是一种实现方法，但是让我们先从简单的方法开始。

## Enums and Switches

---

### 枚举和分支

---

One problem our Heroine class has is some combinations of those Boolean fields aren't valid: `isJumping` and `isDucking` should never both be true, for example. When you have a handful of flags where only one is true at a time, that's a hint that what you really want is an enum. 我们的女主角类有一些布尔类型的成员变量：`isJumping_` 和 `isDucking`，但是这两个变量永远不可能同时为True。当你有一系列的标记成员变量，而它们只能有且仅有一个为True时，这表明我们需要把它们定义成枚举(enum)。

In this case, that enum is exactly the set of states for our FSM, so let's define that: 在这个例子当中，我们的FSM的每一个状态可以用一个枚举来表示，所以，让我们定义以下枚举：

```

enum State
{
    STATE_STANDING,
    STATE_JUMPING,
    STATE_DUCKING,
    STATE_DIVING
};

```

Instead of a bunch of flags, Heroine will just have one state *field*. We also flip the order of our branching. In the previous code, we switched on input, then on state. This kept the code for handling one button press together, but it smeared around the code for one state. We want to keep that together, so we switch on state first. That gives us: 这里没有大量的flags，女主角类只有一个state成员。我们还需要调换分支语句的顺序。在前面的代码中，我们先判断输入事件，然后才是状态。那种代码可以让我们集中处理与按键相关的逻辑，但是，它也让每一种状态的处理代码变得很乱。我们想把它们放在一起处理，因此，我们先判断状态。代码如下：

```

void Heroine::handleInput(Input input)
{
    switch (state_)
    {
        case STATE_STANDING:
            if (input == PRESS_B)
            {
                state_ = STATE_JUMPING;
                yVelocity_ = JUMP_VELOCITY;
                setGraphics(IMAGE_JUMP);
            }
            else if (input == PRESS_DOWN)
            {
                state_ = STATE_DUCKING;
                setGraphics(IMAGE_DUCK);
            }
            break;

        case STATE_JUMPING:
            if (input == PRESS_DOWN)
            {
                state_ = STATE_DIVING;
                setGraphics(IMAGE_DIVE);
            }
            break;

        case STATE_DUCKING:
            if (input == RELEASE_DOWN)
            {
                state_ = STATE_STANDING;
                setGraphics(IMAGE_STAND);
            }
            break;
    }
}

```

This seems trivial, but it's a real improvement over the previous code. We still have some conditional branching, but we simplified the mutable state to a single field. All of the code for handling a single state is now nicely lumped together. This is the simplest way to implement a state machine and is fine for some uses. 这样看起来也挺普通的，但是它却是对前面的代码的一个提升。我们仍然有一些条件分支语句，但是我们简化了状态的处理。所有处理单个状态的代码都集中在一起了。这样做是最简单的方法来实现状态机，而且在某些情况下，这样做也挺好的。

In particular, the heroine can no longer be in an invalid state. With the Boolean flags, some sets of values were possible but meaningless. With the enum, each value is valid. Your problem may outgrow this solution, though. Say we want to add a move where our heroine can duck for a while to charge up and unleash a special attack. While she's ducking, we need to track the charge time. 不过，这样我们的女主角就不可能处于一个无效的状态。通过布尔值标识，我们可以设置一些没有意思的值。但是，使用枚举，每一个枚举值都是有意义的。你的问题可能也会超过此方案能解决的范围。比如，我们想在主角下蹲躲避的时候“蓄能”，然后等蓄满能量之后可以释放出一个特殊的技能。那么，当主角处理躲避状态的时候，我们需要添加一个变量来记录蓄能时间。

We add a chargeTime field to Heroine to store how long the attack has charged. Assume we already have an update() that gets called each frame. In there, we add: 我们可以添加一个chargeTime成员来记录主角蓄能的时间长短。假设，我们已经

有一个update方法了，并且这个方法会在每一帧被调用。在那里，我们可以使用如下代码片断能记录蓄能的时间：

```
void Heroine::update()
{
    if (state_ == STATE_DUCKING)
    {
        chargeTime_++;
        if (chargeTime_ > MAX_CHARGE)
        {
            superBomb();
        }
    }
}
```

We need to reset the timer when she starts ducking, so we modify handleInput(): 我们需要在主角躲避的时候重置这个蓄能时间，所以，我们还需要修改handleInput方法：

```
void Heroine::handleInput(Input input)
{
    switch (state_)
    {
        case STATE_STANDING:
            if (input == PRESS_DOWN)
            {
                state_ = STATE_DUCKING;
                chargeTime_ = 0;
                setGraphics(IMAGE_DUCK);
            }
            // Handle other inputs...
            break;

        // Other states...
    }
}
```

All in all, to add this charge attack, we had to modify two methods and add a chargeTime field onto Heroine even though it's only meaningful while in the ducking state. What we'd prefer is to have all of that code and data nicely wrapped up in one place. The Gang of Four has us covered. 总之，为了添加蓄能攻击，我们不得不修改两个方法，并且添加一个chargeTime 成员给主角，尽管这个成员变量只有在主角处于躲避状态的时候才有效。我们其实真正想要的是把所有这些与状态相关的数据和代码封装起来。接下来，我们介绍四人帮的状态模式来解决这个问题。

## The State Pattern

### 状态模式

For people deeply into the object-oriented mindset, every conditional branch is an opportunity to use dynamic dispatch (in other words a virtual method call in C++). I think you can go too far down that rabbit hole. Sometimes an if is all you need. 对于熟知面向对象方法的人来说，每一个条件分支都可以用动态分发来解决（换句话说，都可以用c++里面的虚函数来解决）。但是，如果你这样做，可能会走远了。有时候，一个简单的if语句就足够了。

There's a historical basis for this. Many of the original object-oriented apostles like Design Patterns' Gang of Four, and Refactoring's Martin Fowler came from Smalltalk. There, ifThen: is just a method you invoke on the condition, which is implemented differently by the true and false objects. 状态模式的由来也有一些历史原因。许多面向对象设计的信徒--四人帮和重构的作者Martin Fowler都是Smalltalk出生。在那里，如果有一个if语句，我们便可以用一个表示true和false的对象来操作。

But in our example, we've reached a tipping point where something object-oriented is a better fit. That gets us to the State pattern. In the words of the Gang of Four: 但是，在我们这个例子当中，我们也达到了设计模式的应用场景。在四人帮的状态模式中是这么描述的：

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. 当一

一个对象在其内部状态改变时改变它的行为，对象看起来好像是修改其类

That doesn't tell us much. Heck, our switch does that. The concrete pattern they describe looks like this when applied to our heroine: 这句话并没有给我们太多信息。真见鬼，我们的switch却做到了。这个模式应用到我们的主角中，恰好和模式定义中描述的差不多。

## A state interface

### 一个状态接口

First, we define an interface for the state. Every bit of behavior that is state-dependent—every place we had a switch before—becomes a virtual method in that interface. For us, that's handleInput() and update(): 首先，我们为状态定义一个接口。每一个与状态相关的行为都定义成虚函数。对于我们而言，就是handleInput和update函数。

```
class HeroineState
{
public:
    virtual ~HeroineState() {}
    virtual void handleInput(Heroine& heroine, Input input) {}
    virtual void update(Heroine& heroine) {}
};
```

## Classes for each state

### 为每一个状态定义一个类

For each state, we define a class that implements the interface. Its methods define the heroine's behavior when in that state. In other words, take each case from the earlier switch statements and move them into their state's class. For example: 对于每一个状态，我们定义了一个类并继承至此状态接口。它覆盖的方法定义主角对应此状态的行为。换句话说，把之前的switch语句里面的每一个case语句里的内容放置到它们对应的状态类里面去。比如：

```
class DuckingState : public HeroineState
{
public:
    DuckingState()
    : chargeTime_(0)
    {}

    virtual void handleInput(Heroine& heroine, Input input) {
        if (input == RELEASE_DOWN) {
            // Change to standing state...
            heroine.setGraphics(IMAGE_STAND);
        }
    }

    virtual void update(Heroine& heroine) {
        chargeTime_++;
        if (chargeTime_ > MAX_CHARGE) {
            heroine.superBomb();
        }
    }

private:
    int chargeTime_;
};
```

Note that we also moved chargeTime out of Heroine and into the DuckingState class. This is great—that piece of data is only meaningful while in that state, and now our object model reflects that explicitly. 注意，我们这里把chargeTime也放到了DuckingState(躲避状态)中。这样非常好，因为这个变量只是对躲避状态有意义，现在把它定义在这里，正好显式地反应了我们的对象模型。

## Delegate to the state

### 状态委托

Next, we give the Heroine a pointer to her current state, lose each big switch, and delegate to the state instead: 接下来，我们在主角类中定义一个指针变量，让它指向当前的状态。我们把之前那个很大的switch语句去掉了，然后让它去调用状态接口的虚函数，最终这些虚方法就会动态地调用具体子状态的相应函数了。

```
class Heroine
{
public:
    virtual void handleInput(Input input)
    {
        state_->handleInput(*this, input);
    }

    virtual void update()
    {
        state_->update(*this);
    }

    // Other methods...
private:
    HeroineState* state_;
};
```

In order to “change state”, we just need to assign state to point to a different HeroineState object. That’s the State pattern in its entirety. 为了修改状态，我们需要把state指针指向另一个不同的状态对象。至此，我们的状态模式就讲完了。

## Where Are the State Objects?

### 状态对象应该放在哪里呢？

I did gloss over one bit here. To change states, we need to assign state to point to the new one, but where does that object come from? With our enum implementation, that was a no-brainer—enum values are primitives like numbers. But now our states are classes, which means we need an actual instance to point to. There are two common answers to this: 我这里忽略了一些细节。为了修改一个状态，我们需要给state指针赋值为一个新的状态，但是这个新的状态对象要从哪里来呢？我们的之前的枚举方法是一些数字定义。但是，现在我们的状态是类，我们需要获取这些类的实例。通常来说，有两种实现方法：

#### Static states

##### 静态状态

If the state object doesn’t have any other fields, then the only data it stores is a pointer to the internal virtual method table so that its methods can be called. In that case, there’s no reason to ever have more than one instance of it. Every instance would be identical anyway. 如果一个状态对象没有任何数据成员，那么它的唯一数据成员便是虚表指针了。那样的话，我们就没有必要创建此状态的多个实例了，因为它们的每一个实例都是相等的。

If your state has no fields and only one virtual method in it, you can simplify this pattern even more. Replace each state class with a state function—just a plain vanilla top-level function. Then, the state field in your main class becomes a simple function pointer. 如果你的状态类没有任何数据成员，并且它只有一个函数方法在里面。那么我们还可以进一步简化此模式。我们可以通过一个状态函数来替换状态类。这样的话，我们的state变量只需要变成一个状态函数指针就可以了。

In that case, you can make a single static instance. Even if you have a bunch of FSMs all going at the same time in that same state, they can all point to the same instance since it has nothing machine-specific about it. 在那种情况下，我们可以定义一个静态实例。即使你有一系列的FSM在同时运转，所有的状态机都同时指向这一个唯一的实例。

This is the Flyweight pattern. 这个就是享元模式。

Where you put that static instance is up to you. Find a place that makes sense. For no particular reason, let's put ours inside the base state class: 你把静态方法放置在哪里，这个由你自己来决定。如果没有任何特殊原因的话，我们可以把它放到基类状态类中：

```
class HeroineState
{
public:
    static StandingState standing;
    static DuckingState ducking;
    static JumpingState jumping;
    static DivingState diving;

    // Other code...
};
```

Each of those static fields is the one instance of that state that the game uses. To make the heroine jump, the standing state would do something like: 每一个静态成员变量都是对应状态类的一个实例。如果我们想让主角跳跃，那么站立状态应该是这样子：

```
if (input == PRESS_B)
{
    heroine.state_ = &HeroineState::jumping;
    heroine.setGraphics(IMAGE_JUMP);
}
```

## Instantiated states

### 实例化状态

Sometimes, though, this doesn't fly. A static state won't work for the ducking state. It has a chargeTime field, and that's specific to the heroine that happens to be ducking. This may coincidentally work in our game if there's only one heroine, but if we try to add two-player co-op and have two heroines on screen at the same time, we'll have problems. 有时候上面的方法可能不行。一个静态状态对于躲避状态而言是行不通的。因为它有一个chargeTime成员变量，这个是专属于每一个主角类在躲避状态下的。如果我们的游戏里面只有一个主角的话，那么定义一个静态类也是没有啥问题的。但是，如果我们想加入多个玩家，那么此方法就行不通了。

In that case, we have to create a state object when we transition to it. This lets each FSM have its own instance of the state. Of course, if we're allocating a new state, that means we need to free the current one. We have to be careful here, since the code that's triggering the change is in a method in the current state. We don't want to delete this out from under ourselves. 在那种情况下面，我们不得不在状态切换的时候动态地创建一个躲避状态实例。这样，我们的FSM并拥有了它自己的实例。当然，如果我们又动态分配了一个新的状态实例，我们需要负责清理老的状态实例。我们这里必须要相当小心，因为当前状态修改的函数是处在当前状态里面，我们需要小心地处理删除的顺序。

Instead, we'll allow handleInput() in HeroineState to optionally return a new state. When it does, Heroine will delete the old one and swap in the new one, like so: 另外，我们也会在handleInput方法里面可选地返回一个新的状态。当这个状态返回的时候，主角将会删除老的状态并切换到这个新的状态，如下所示：

```
void Heroine::handleInput(Input input)
{
    HeroineState* state = state_->handleInput(*this, input);
    if (state != NULL)
    {
        delete state_;
        state_ = state;
    }
}
```

That way, we don't delete the previous state until we've returned from its method. Now, the standing state can transition to

ducking by creating a new instance: 那样的话，我们只有在从handleInput方法返回的时候才有可能去删除前面的对象。现在，站立状态可以通过创建一个躲避状态的实例来切换状态了。

```
HeroineState* StandingState::handleInput(Heroine& heroine,
                                         Input input)
{
    if (input == PRESS_DOWN)
    {
        // Other code...
        return new DuckingState();
    }

    // Stay in this state.
    return NULL;
}
```

When I can, I prefer to use static states since they don't burn memory and CPU cycles allocating objects each state change. For states that are more, uh, stateful, though, this is the way to go. 当我在做选择的时候，我倾向于使用静态状态。因数它们不会占用太多的CPU和内存资源。

## Enter and Exit Actions

### 进入状态和退出状态的行为

The goal of the State pattern is to encapsulate all of the behavior and data for one state in a single class. We're partway there, but we still have some loose ends. 状态模式的目标就是封装所有的数据和行为到一个状态类里面。万里长征，我们仅仅是迈出去了一步，我们还可以走地更远。

When the heroine changes state, we also switch her sprite. Right now, that code is owned by the state she's switching from. When she goes from ducking to standing, the ducking state sets her image: 当主角更改状态的时候，我们也会切换它的精灵。现在，这段代码是包含在它要切换的状态的上一个状态里面。当她从躲避状态切换到站立状态的时候，躲避状态将会修改它的图像：

```
HeroineState* DuckingState::handleInput(Heroine& heroine,
                                         Input input)
{
    if (input == RELEASE_DOWN)
    {
        heroine.setGraphics(IMAGE_STAND);
        return new StandingState();
    }

    // Other code...
}
```

What we really want is each state to control its own graphics. We can handle that by giving the state an enter action: 我们希望的是，每一个状态来自己控件自己的图像。我们可以通过给每一个状态添加一个enter行为。

```
class StandingState : public HeroineState
{
public:
    virtual void enter(Heroine& heroine)
    {
        heroine.setGraphics(IMAGE_STAND);
    }

    // Other code...
};
```

Back in Heroine, we modify the code for handling state changes to call that on the new state: 回到女主角的例子，我们修改代码来处理状态切换的情况：

```

void Heroine::handleInput(Input input)
{
    HeroineState* state = state_->handleInput(*this, input);
    if (state != NULL)
    {
        delete state_;
        state_ = state;

        // Call the enter action on the new state.
        state_->enter(*this);
    }
}

```

This lets us simplify the ducking code to: 这样也可以让我们简化躲避状态的代码：

```

HeroineState* DuckingState::handleInput(Heroine& heroine,
                                         Input input)
{
    if (input == RELEASE_DOWN)
    {
        return new StandingState();
    }

    // Other code...
}

```

All it does is switch to standing and the standing state takes care of the graphics. Now our states really are encapsulated. One particularly nice thing about entry actions is that they run when you enter the state regardless of which state you're coming from. 它所做的就是切换到站立状态，然后站立状态会自己设置自己的图像。现在，我们的状态已经封装好了。entry 动作的一个最大的好处就是它一个状态进来的时候，它不用关心上一个状态是什么，它只需要根据自己的状态来处理图像和行为就ok了。

Most real-world state graphs have multiple transitions into the same state. For example, our heroine will also end up standing after she lands a jump or dive. That means we would end up duplicating some code everywhere that transition occurs. Entry actions give us a place to consolidate that. 大部分的真实状态图里面，我们有多个状态对应同一个状态。比如，我们的女主角会在她俯冲或者跳跃之后站立在地面上。这意味着，我们可能会在每一个状态发生变化的时候重复写很多代码。但是，entry 动作帮我们很好地解决了这个问题。

We can, of course, also extend this to support an exit action. This is just a method we call on the state we're leaving right before we switch to the new state. 当然，我们也可以扩展这个功能来支持退出状态的行为。我们可以定义一个exit函数来定义一些在状态改变后的处理。

## 发现什么问题了吗？

I've spent all this time selling you on FSMs, and now I'm going to pull the rug out from under you. Everything I've said so far is true, and FSMs are a good fit for some problems. But their greatest virtue is also their greatest flaw. 我已经花了大量的时间来向你兜售FSM了。现在，我将要把你拉回来。到目前为止，我跟你讲的任何事情都是对的，FSM对于某些应用来讲是非常合适的。但是，往往最大的优点也是最大的缺点。

State machines help you untangle hairy code by enforcing a very constrained structure on it. All you've got is a fixed set of states, a single current state, and some hardcoded transitions. 状态机帮助你把千丝万缕的逻辑判断代码封装起来了。你需要的只是一组状态，一个当前状态和一些硬编码的状态切换。

A finite state machine isn't even Turing complete. Automata theory describes computation using a series of abstract models, each more complex than the previous. A Turing machine is one of the most expressive models. 一个有限状态机甚至都不是一个图灵机。自动化理论使用一系列抽象的模型来描述计算，并且每一个模型都比先前的模型更复杂。而图灵机只是这里面最具有表达力的模型。

"Turing complete" means a system (usually a programming language) is powerful enough to implement a Turing machine in it, which means all Turing complete languages are, in some ways, equally expressive. FSMs are not flexible enough to be

in that club. “图灵完备”意味着一个系统（通常指的是一门编程语言）是足够强大的，强大到它可以实现一个图灵机。这也意味着，所有图灵完备的编程语言，在某些程度上来也是一种FSM。

If you try using a state machine for something more complex like game AI, you will slam face-first into the limitations of that model. Thankfully, our forebears have found ways to dodge some of those barriers. I'll close this chapter out by walking you through a couple of them. 如果你想要用一个状态机来表示一些复杂的游戏AI，你可能会面临这个模型的一些限制。幸运的是，我们的先辈们已经发现一些解决方案可以解决些问题。我将会在本章的最后简单地提到它们。

## 并发状态机

We've decided to give our heroine the ability to carry a gun. When she's packing heat, she can still do everything she could before: run, jump, duck, etc. But she also needs to be able to fire her weapon while doing it. 我们已经决定给我们的主角添加持枪功能。当她手持枪的时候，她仍然可以：跑、跳和躲避。但是，她同时也能够在这些状态过程中开火。

If we want to stick to the confines of an FSM, we have to double the number of states we have. For each existing state, we'll need another one for doing the same thing while she's armed: standing, standing with gun, jumping, jumping with gun, you get the idea. 如果你执着于传统的FSM，我们可能需要把之前的状态加倍。对于每一个已经存在的状态，我们需要定义另一个状态，它做的事情也差不多，不过就是多了持枪的操作。比如站立状态和状态开火状态，跳跃状态和跳跃开火状态等。

Add a couple of more weapons and the number of states explodes combinatorially. Not only is it a huge number of states, it's a huge amount of redundancy: the unarmed and armed states are almost identical except for the little bit of code to handle firing. 如果我们添加更多的武器种类，那么这个状态种类会爆炸的。而且不仅仅是增加了大量的状态类实例而已，它还会重复编写相当多的重复代码。

The problem is that we've jammed two pieces of state—what she's doing and what she's carrying—into a single machine. To model all possible combinations, we would need a state for each pair. The fix is obvious: have two separate state machines. 这里的问题是，我们把两种状态杂合在一起了。我们把两种不同的状态硬塞到一个状态机里面去了。为了建模所有可能的组合，我们可能需要为每一种状态准备一组状态。解决方法比较直观，就是我们提供两个状态机。

If we want to cram n states for what she's doing and m states for what she's carrying into a single machine, we need  $n \times m$  states. With two machines, it's just  $n + m$ . 如果我们可以为主角定义n种状态，那么就可以为它所持装备定义m种状态，并把它们分别放入不同的状态机。因此，我们只需要 $n * m$ 个状态就够了。如果有两个状态机，那么状态组合是 $n+m$ 。

We keep our original state machine for what she's doing and leave it alone. Then we define a separate state machine for what she's carrying. Heroine will have two “state” references, one for each, like: 为了保持我们原来的状态机，我们这里先不管。接下来，我们定义了一个单独的状态机，用来处理主角携带的武器。现在，我们的主角会有两个状态索引，其中一个看起来如下所示：

```
class Heroine
{
    // Other code...

private:
    HeroineState* state_;
    HeroineState* equipment_;
};
```

When the heroine delegates inputs to the states, she hands it to both of them: 当主角派发输入事件给状态类的，需要给两种状态都派发一下。

```
void Heroine::handleInput(Input input)
{
    state_->handleInput(*this, input);
    equipment_->handleInput(*this, input);
}
```

Each state machine can then respond to inputs, spawn behavior, and change its state independently of the other machine. When the two sets of states are mostly unrelated, this works well. 这样每一个状态机都可以响应输入事件并以此切换状态而不考虑其它状态机的内部。当两个状态没什么关系的时候，这种方法工作地很好。

In practice, you'll find a few cases where the states do interact. For example, maybe she can't fire while jumping, or maybe she can't do a dive attack if she's armed. To handle that, in the code for one state, you'll probably just do some crude if tests on the other machine's state to coordinate them. It's not the most elegant solution, but it gets the job done. 在实际中，你可能会发现你需要对某些状态处理进行干预。比如，如果主角不能够在跳跃的过程中开火，或者她在装备武器的时候不能俯冲。为了处理这种情况，在代码里面，对于每一个状态，你可能需要做一些简单的if判断并做出特殊处理。虽然这可能不是最好的解决方案，但是至少它可以完成任务。

## 层级状态机

After fleshing out our heroine's behavior some more, she'll likely have a bunch of similar states. For example, she may have standing, walking, running, and sliding states. In any of those, pressing B jumps and pressing down ducks. 在我们把主角的行为更加具象化以后，她可能会包含大量相似的状态。比如，她可能有站立、走路、跑步和滑动状态。在这些状态中的任何一个状态按下B键，我们的主角要跳跃，按下down键，我们的主角要躲避。

With a simple state machine implementation, we have to duplicate that code in each of those states. It would be better if we could implement that once and reuse it across all of the states. 如果只是使用一个简单的状态机实现，我们可能会在这些状态中重复不少代码。更好的解决方案是，我们只需要实现一次那么它便可以在所有的状态下都有效。

If this was just object-oriented code instead of a state machine, one way to share code across those states would be using inheritance. We could define a class for an "on ground" state that handles jumping and ducking. Standing, walking, running, and sliding would then inherit from that and add their own additional behavior. 如果我们抛开状态机来谈面向对象，有一种共享代码的方式便是继承。我们可以定义一个类来代码在地上的状态，它用来处理跳跃状态和躲避状态。站立，走跳，跑步和滑行状态从这个在地面上的状态继承而来，并且在其类里面实现一些特殊行为。

This has both good and bad implications. Inheritance is a powerful means of code reuse, but it's also a very strong coupling between two chunks of code. It's a big hammer, so swing it carefully. It turns out, this is a common structure called a hierarchical state machine. A state can have a superstate (making itself a substate). When an event comes in, if the substate doesn't handle it, it rolls up the chain of superstates. In other words, it works just like overriding inherited methods. 这可能既是一个好的设计，也可能是一个坏的设计。继承是一种强大的代码重用方式，但是，它也会使得子类与基类之间的代码变得紧耦合。它是一个很大的锤子，需小心使用才行。这里，我们通常把这种状态机叫做层级状态机。一个状态有一个父状态。当有一个事件进来的时候，如果子状态不处理它，那么并沿着继承链传给它的父状态来处理。换句话说，它有点像覆盖继承的方法。

in fact, if we're using the State pattern to implement our FSM, we can use class inheritance to implement the hierarchy. Define a base class for the superstate: 实际上，如果我们正在使用状态模式来实现FSM，我们可以使用类层次来实现层级状态机。我们首先定义一个基类来表示父状态：

```
class OnGroundState : public HeroineState
{
public:
    virtual void handleInput(Heroine& heroine, Input input)
    {
        if (input == PRESS_B)
        {
            // Jump...
        }
        else if (input == PRESS_DOWN)
        {
            // Duck...
        }
    }
};
```

And then each substate inherits it: 然后，每一个子状态都继承至它：

```

class DuckingState : public OnGroundState
{
public:
    virtual void handleInput(Heroine& heroine, Input input)
    {
        if (input == RELEASE_DOWN)
        {
            // Stand up...
        }
        else
        {
            // Didn't handle input, so walk up hierarchy.
            OnGroundState::handleInput(heroine, input);
        }
    }
};

```

This isn't the only way to implement the hierarchy, of course. If you aren't using the Gang of Four's State pattern, this won't work. Instead, you can model the current state's chain of superstates explicitly using a stack of states instead of a single state in the main class. 当然，这不是实现层级状态机的惟一方式。如果你没有使用GoF的状态模式，这种做法可能并不奏效。相反，你也可以使用栈结构来显示地模拟当前状态的父级状态的状态链。

The current state is the one on the top of the stack, under that is its immediate superstate, and then that state's superstate and so on. When you dish out some state-specific behavior, you start at the top of the stack and walk down until one of the states handles it. (If none do, you ignore it.) 我们当前的状态总是处于栈顶，栈顶下面的第一个元素则是它的父状态，再下一个状态则是它的爷爷状态，等等。如果你要进行一些与状态相关的行为操作，那么首先从栈顶状态开始。如果它不处理，则往上继承寻找一个能处理此事件的状态为止。（如果找遍整个栈了，还是没人处理，则此事件忽略掉）.

## 下推自动机

---

There's another common extension to finite state machines that also uses a stack of states. Confusingly, the stack represents something entirely different, and is used to solve a different problem. 这里还有一种有限状态机的变种，它们使用状态栈。可能听起来有些奇怪，栈完全是另外一种截然不同的东西。

The problem is that finite state machines have no concept of history. You know what state you are in, but have no memory of what state you were in. There's no easy way to go back to a previous state. 这里要解决的问题是，因为有限状态机是没有历史记录这个概念的。我们知道一个状态进来了，但是，我们并不知道这个状态的上一个状态是什么。而且，我们也没有简便地方法可以获取当前状态的上一个状态。

Here's an example: Earlier, we let our fearless heroine arm herself to the teeth. When she fires her gun, we need a new state that plays the firing animation and spawns the bullet and any visual effects. So we slap together a FiringState and make all of the states that she can fire from transition into that when the fire button is pressed. 这里有一个例子：之前，我们让我们无畏的主角全副武装。when她开枪的时候，我们需要一种新的状态来播放开枪的动画，但是发射子弹并显示一些特效。因此，我们需要定义一个FiringState，并且所有的状态都可以切换到这个状态，只要有玩家按下开火按键就行了。

Since this behavior is duplicated across several states, it may also be a good place to use a hierarchical state machine to reuse that code. The tricky part is what state she transitions to after firing. She can pop off a round while standing, running, jumping, and ducking. When the firing sequence is complete, she should transition back to what she was doing before. 因为这个行为在许多状态里面都重复了，所以，我们需要使用层级状态机来解决这个问题。这里的问题来了，当她开完枪后，她要回到什么状态呢？主角可以处于站立、躲避、俯冲和跳跃状态。但开火的动画播放完以后，她应该要回到之前的状态。

If we're sticking with a vanilla FSM, we've already forgotten what state she was in. To keep track of it, we'd have to define a slew of nearly identical states — firing while standing, firing while running, firing while jumping, and so on — just so that each one can have a hardcoded transition that goes back to the right state when it's done. 如果我们仍然坚持使用以前的FSM，那么我们将无法获得上一个状态的信息。为了保留上一个状态的信息，我们不得不定义一些几乎对等的状态，比如站立开火状态，跑步开火状态等。这样的话，当我们的开火状态完成以后，就可以切换回之前的状态了。

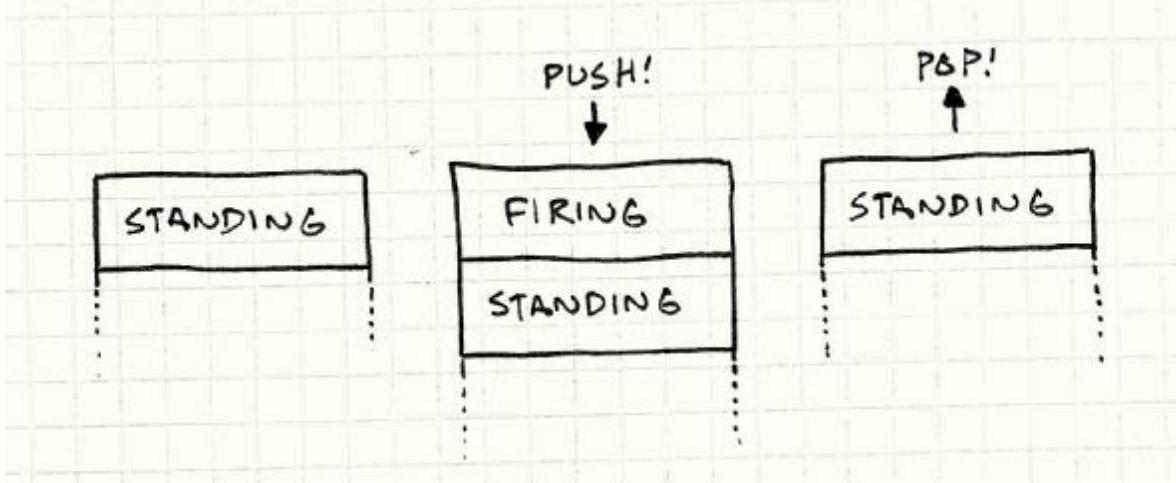
What we'd really like is a way to store the state she was in before firing and then recall it later. Again, automata theory is here to help. The relevant data structure is called a pushdown automaton. 我们需要的仅仅是提供一种方式，让我们可以保存开火前的状态，这样在开火状态完成之后可以回去。好了，自动机理论可以帮助我们。相关的数据结构叫做下推自动机

(pushdown automata) .

Where a finite state machine has a single pointer to a state, a pushdown automaton has a stack of them. In an FSM, transitioning to a new state replaces the previous one. A pushdown automaton lets you do that, but it also gives you two additional operations: 本来，一个有限状态机有一个指针指向一个状态。而下推自动机则有一个状态栈。在一个FSM里面，当有一个状态切进来，则替换掉之前的状态。下推自动机可以让你这样做，同时它还提供其它选择：

You can push a new state onto the stack. The “current” state is always the one on top of the stack, so this transitions to the new state. But it leaves the previous state directly under it on the stack instead of discarding it. 你可以把这个新的状态放入栈里面。那么当前的状态就永远存在栈顶，当你需要回退到上一个状态的时候，只需要栈顶出栈就可以得到上一个状态了。

此时，上一个状态就变成了新的栈顶状态了。



This is just what we need for firing. We create a single firing state. When the fire button is pressed while in any other state, we push the firing state onto the stack. When the firing animation is done, we pop that state off, and the pushdown automaton automatically transitions us right back to the state we were in before. 这个就是我们的开火状态所需要的。当开火按钮在任何一种状态下被按下的时候，我们把开火状态push到栈顶。当开火动画结束的时候，我们把这个开火状态pop出去。此时，状态机会自动切换到我们开火前的上一个状态。

## 现在知道它们有多有用了吧？

Even with those common extensions to state machines, they are still pretty limited. The trend these days in game AI is more toward exciting things like behavior trees and planning systems. If complex AI is what you're interested in, all this chapter has done is whet your appetite. You'll want to read other books to satisfy it. 即使有了这些通用的状态机扩展，它们的使用范围仍然是有限的。在游戏的AI领域的最近的趋势是越来越倾向于行为树和规划系统。如果你对复杂的AI感兴趣的话，那么本章所有这些内容只是在吊你的胃口。你可能还想通过阅读其它的书籍来了解它们。

This doesn't mean finite state machines, pushdown automata, and other simple systems aren't useful. They're a good modeling tool for certain kinds of problems. Finite state machines are useful when: 但是这并不意味着有限状态机，下推自动机和其它简单的状态机没有用。它们对于解决某些特定的问题是一个很好的建模工具。当你的问题满足以下几点要求的时候，有限状态机将会非常有用：

- You have an entity whose behavior changes based on some internal state.
- 你有一个游戏裸体，它的行为基于它的内部状态而改变
- That state can be rigidly divided into one of a relatively small number of distinct options.
- 这些状态被严格划分为为小个有限的小集合。
- The entity responds to a series of inputs or events over time.
- 游戏实体随着时间的变化会响应用户输入和一些游戏事件。

In games, they are most known for being used in AI, but they are also common in implementations of user input handling, navigating menu screens, parsing text, network protocols, and other asynchronous behavior. 在游戏里面，它们在AI里面被广泛使用，但是它们也经常被应用于“用户输入处理”，“浏览菜单屏幕”，“解析文件”，“网络协议”和其它异步的行为。

## 序列模式

---

Videogames are exciting in large part because they take us somewhere else. For a few minutes (or, let's be honest with ourselves, much longer), we become inhabitants of a virtual world. Creating these worlds is one of the supreme delights of being a game programmer.

视频游戏很大程度会让我们兴奋是因为它们把我们带到了其他地方。在几分钟（或者，坦率讲，时间更长）里，我们成为了虚拟世界的人。创建这些世界是作为游戏程序员的最高乐趣之一。

One aspect that most of these game worlds feature is time—the artificial world lives and breathes at its own cadence. As world builders, we must invent time and craft the gears that drive our game's great clock

从一方面来讲，大多数游戏世界的特征便是时间--虚拟世界按照自己的节奏运行着。作为世界的建造者，我们必须创造时间和打磨用来驱动游戏巨大时钟的齿轮。（译者注：这里作者用了比喻来说明问题）

The patterns in this section are tools for doing just that. A Game Loop is the central axle that the clock spins on. Objects hear its ticking through Update Methods. We can hide the computer's sequential nature behind a facade of snapshots of moments in time using Double Buffering so that the world appears to update simultaneously.

在本节中的模式便是用来做那样工作的工具。[游戏循环](#)是时钟旋转的中心轴，对象通过[更新方法](#)来聆听它的滴答声。我们可以通过[双缓冲](#)来及时的将计算机的连续性隐藏在时刻快照之后，从而使得游戏世界能够同步更新。

## 本章模式

---

- [双缓冲](#)
- [游戏循环](#)
- [更新方法](#)

# 双缓冲模式

---

#

---

## 目的

---

进行一系列序列化操作来表现出瞬发性或同步性。

## 动机

---

计算机的心脏里藏着凶残的序列化处理能力。其力量源于它们能够将庞大的任务分解为许多细小的步骤以便逐个处理。尽管通常对于我们的用户而言,他们希望看到的是问题能够即刻被处理,或者多个任务能同时被执行。

注解: 虽然线程和多核技术在不断进步,但即便在多核环境下,也仅有少数操作能真正同步地执行

举个典型的例子,每个游戏引擎所必会涉及的——渲染。当引擎为用户渲染出可见的世界时,它是分步骤来完成渲染任务的:远处的山峰,起伏的山脉,树木,这些被轮流渲染。假如用户也跟着逐步地观察引擎的渲染,那么这个连续游戏世界的幻像将会破碎。场景必须快速而平滑地进行更新,显示一系列完整的帧,而每帧都应当瞬间显示出来。

双缓冲模式解决了上述问题,但为便于理解,首先让我们回顾一下计算机是如何显示图形的。

## 计算机图形系统工作原理概述

---

诸如计算机显示屏的显示设备在每一时刻仅绘制一个像素。显示设备从左至右地扫描屏幕第一行的每个像素,并如此从上至下地扫描屏幕上的每一行。直到扫描至屏幕的右下角,它将重置回屏幕的左上角并如前述那样地重复扫描屏幕。这一扫描过程是如此地快速(大概每秒60次),以至于我们的眼睛无法察觉这一过程。对于我们而言,扫描的结果就是一块静态的彩色像素区域,即一张图片。

注解: 这样的阐述不太妥当——它过于简单了。假如你从事底层硬件开发我想你大概已经笑了,你可以轻松地跳过后面的部分,并完全能够理解本章余下的内容。但假如你并非这样的人物,那么在此我的目的是给予你足够的背景知识以便你能理解我们随后要讨论的设计模式。

你可以将上述过程想象成一根细小的软管在向显示区域不断喷洒出像素。单个颜色像素到达软管的末端,软管将它们喷射到显示区域中,每次往每个像素上喷洒一点。那么它如何知道哪个像素该往哪喷呢?

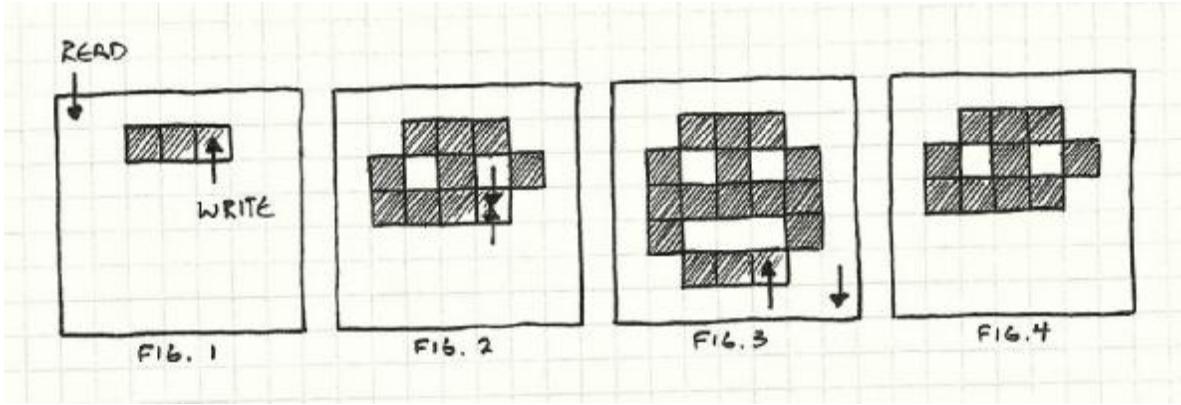
在多数计算机中答案是:它从帧缓冲区(framebuffer)中获知这些信息。帧缓冲区是一块内存,是存储着像素的数组(它是RAM中的一个块,其中每两个字节表示一个像素)。当软管往显示区域喷洒时,它从这个数组中读取颜色值,每次读取1字节。

注解: 字节值与颜色之间的特殊映射关系是通过系统中的像素格式以及色彩深度来描述的。在当今的多数控制台游戏平台,每个像素占32位:红绿蓝色彩通道各占8位,剩余的8位则用于其他多种用途。

基本上讲,为了让游戏在屏幕上显示出来,我们只需要往这个数组里写东西。我们熬夜折腾出来的那些先进图形算法,其根本都只是在往帧缓冲区里设置字节的值。但这里有个小问题。

前面我说计算机的处理是序列化的。假设计算机正在处理我们的一段渲染代码,我们便不希望计算机同时在做其他不相干的事。这几乎是对的,然而在我们的程序运行过程中间还是会穿插着许多其他的事情:比如当我们的游戏在运行时,显示设备会从帧缓冲区中读取内存中的像素信息。这就为我们带来了问题。

比如我们希望在屏幕上显示一张笑脸。我们的程序开始循环访问帧缓冲并对像素进行渲染。出乎我们意料的是,显卡正是在我们往帧缓冲区中写入数据的同时进行数据读取的。一开始它扫描到那些我们已经写入的数据,笑脸便开始在屏幕上浮现,但它渐渐超过我们的写入速度而访问了帧缓冲区中那些未写入数据的部分——悲惨的结局,屏幕上留下了一个半成品,这是个能看得一清二楚的BUG。



注解: 如图,我们在显卡设备开始从帧缓存读取数据的同时进行像素数据的写入(图16.1)。最终显卡赶上并超过了渲染器并访问了我们尚未写入数据的帧缓存区域(图16.2)。我们结束绘制(图16.3)时,那些在被显卡读取后才写入的数据就没有被它读取到。结果用户看到的是渲染的半成品(图16.4)。我称它是“哭丧脸”——笑脸的下半边像是被撕掉了一样。

这就是我们需要本设计模式的原因。我们的程序一次性渲染所有的像素,同时我们要求显示器也一次性将其显示出来——可能这一帧看不到任何东西,但下一帧显示的就是完整的笑脸。双缓冲解决了这一问题。下面我会以类比的形式来阐述。

## 场景1,幕1

设想我们的用户正在观看我们创作的一场表演。当第一个场景谢幕后第二个场景跟着上映,这时候我们需要切换场景。如果我们在场景后台控制舞台管理设备直接开始收起场景道具,那么场景在视觉上的连续性会被破坏。我们可以在收拾场景的同时将灯光变暗(这也正是影剧院所做的),而观众们依然知道黑暗中戏剧仍在继续。我们希望在剧幕之间不会产生间隙。

在资源允许的情况下,我们想到了这个好办法:我们建立两个舞台以便它们都能为观众所见。它们各有各的光源设置。我们称其为A舞台和B舞台。场景1正在A舞台上上演,同时舞台B正处在黑暗中并正由场景后台进行着场景2的准备。一旦场景1结束,我们就关掉A舞台的灯光并将灯光转移到B舞台,观众们便立即聚焦到新舞台并看到了第二幕场景上映。

与此同时,我们的场景后台正在清理舞台A,它清理场景1并为场景3做准备。一旦场景2结束,我们再将光线聚焦到A舞台上。我们在整场表演过程中重复上述过程,将黑暗中的舞台作为工作区来为下个场景做准备。每次场景切换,我们只是将灯光在两个舞台之间来回切换。我们的观众于是就看到了衔接流畅而无缝的场景转换。他们从不会看到舞台的后台。

注解: 借助单面镜以及其他一些巧妙的布局,实际上你能够在同一个舞台进行场景之间的无缝切换。当灯光转移时,观众们可能会聚焦到另一个舞台上,但他们并不一定要转移视线。如何做到这一点就留给读者思考吧。

## 回到图形上

上面就是双缓冲的工作原理,你所见到的任何一款游戏其渲染系统中都重复着这样的过程,我们也是。如我们所类比的,双缓冲中的一个缓存用于展示当前帧,即A舞台。它就是显示硬件读取像素数据的地方,GPU对其进行扫描,整个缓冲区的数据都是它的。

注解: 然而并非所有的游戏和控制台都这么做。早前比较简单的控制台游戏受到内存的局限,小心翼翼地将渲染与机器刷新操作进行同步来取代双缓冲,这可是要技巧的。

于此同时,我们的渲染代码正在往另一个帧缓冲区中写入数据,它就是我们黑暗中的B舞台。当渲染代码完成场景2的渲染时,它通过交换两个缓冲区来“切换光线”。这使得显卡驱动开始从第一个缓冲区转向第二个缓冲区以读取其数据。只要它掌握好时机在每次刷新显示结束时进行切换,我们就不会看到任何衔接的裂隙,且整个场景能一次性显示出来。这时候,旧的帧缓冲变得可用后,我们就开始往它的内存区域渲染入下一帧。这真棒!

## (双缓冲)模式

定义一个缓冲区类来代表一个缓冲区:一系列能被修改的状态。这块缓冲区能被一步步地修改,但我们希望任何外部的代码对该缓冲区的修改都是原子操作。为实现这一点,此类中维护两个缓冲区实例:当前缓冲区和后台缓冲区。

当要从缓冲区中读取信息时,总是从当前缓冲区读取。当要往缓冲区中写数据时,则总在后台缓冲区上进行。当改动完成后,则执行“交换”操作来讲当前缓冲区与后台缓冲区交换,以便让新的缓冲区为我们所见,同时刚被换下来的当前缓冲区则成为现在的后台缓冲区以供复用。

## 使用情境

---

这是个到需要时你自然会想起的设计模式之一。假如你的系统不支持双缓冲,那么显然是没法用了(比如会出现“撕裂”现象),或者显示将表现出异常。但是说“需要的时候你自然会想起”还是太宽泛了,更准确地说,当下面这些条件都成立时,适用双缓冲模式:

- 我们需要维护一些能够不断被修改的状态量。
- 同个状态可能会在其被修改的同时被访问到。
- 我们希望改变状态的工作进程对正在访问这些状态的外部代码透明。
- 我们希望能够读取到这些状态,而无需在其被写入时等待。

## 使用须知

---

不同于其他大架构的设计模式,双缓冲模式的实现处于较底层。因此,它对代码库的影响较少——甚至多数游戏都不会在意这些差别。当然,下面这些附加说明还是值得一提的。

### 交换操作本身是耗时的

---

双缓冲模式需要在状态写入完成后进行一个交换缓冲区的动作。这个操作必须是原子性的:也就是说任何代码都无法在这个操作其间对任何一块缓冲区内的状态进行访问。通常这个交换过程和分配一个指针的速度差不多,但万一交换花去了比修改初始状态更多的时间,那这模式就毫无助益了。

### 必须要有两个缓冲区

---

使用此模式的另一结果是导致内存占用率增加。正如其名,此模式要求你在任何时刻都维护着两份存储着状态的内存区域。在内存受限的硬件上,这可是个很苛刻的要求。假如你无法分配出两份内存,你就必须想其他办法来避免你的状态在修改时被访问。

## 示例

---

说完理论,让我们来结合实践,看看它是如何工作的。我们将写一个及其简单的图形系统以供我们在帧缓存上绘制像素。在多数控制台和PC上,显卡驱动提供了这一底层部分的图形系统,而这里通过手动实现它,我们将能窥其全貌。首先是缓冲区:

```
class Framebuffer
{
public:
    Framebuffer() { clear(); }
    void clear()
    {
        for (int i = 0; i < WIDTH * HEIGHT; i++)
        {
            pixels_[i] = WHITE;
        }
    }

    void draw(int x, int y)
    {
        pixels_[((WIDTH * y) + x)] = BLACK;
    }

    const char* getPixels()
    {
        return pixels_;
    }
}
```

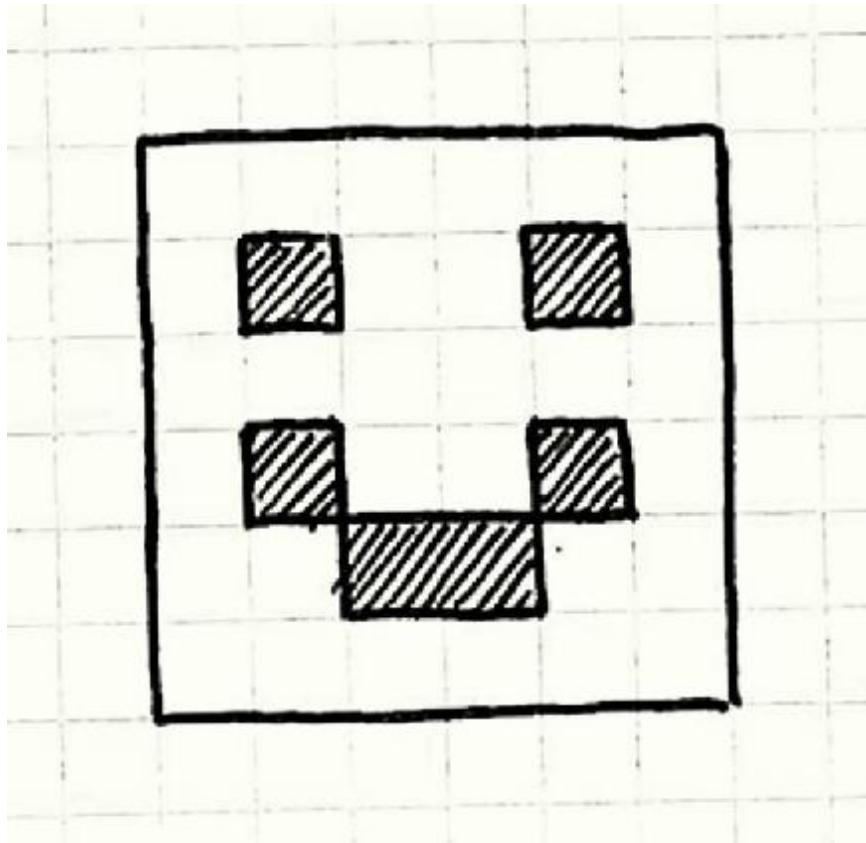
```
private:  
    static const int WIDTH = 160;  
    static const int HEIGHT = 120;  
  
    char pixels_[WIDTH * HEIGHT];  
};
```

缓冲区拥有一些基本操作:将整个缓冲区清理为默认颜色,对指定位置的像素颜色值进行设置。它还包含了 `getPixels()` 函数,用于外部访问缓冲区持有的整个原始像素数组。我们并不会在例子中看到它,但实际中,显卡驱动会频繁地调用这个函数来将缓冲区的内存流式地输出到屏幕上。

我们在 `Scene` 类里包装这个原始的缓冲区。此类的任务在于对其缓冲区进行一系列的 `draw()` 函数调用来渲染出图形。

```
class Scene  
{  
public:  
    void draw()  
    {  
        buffer_.clear();  
        buffer_.draw(1, 1);  
        buffer_.draw(4, 1);  
        buffer_.draw(1, 3);  
        buffer_.draw(2, 4);  
        buffer_.draw(3, 4);  
        buffer_.draw(4, 3);  
    }  
  
    Framebuffer& getBuffer() { return buffer_; }  
  
private:  
    Framebuffer buffer_;  
};
```

注解: 具体来说,它画出了这样一幅杰作:



游戏在每帧通知场景进行绘制。场景清理缓冲区接着一次性地绘制一系列像素。它也通过方法 `getBuffer()` 提供了对内部缓冲区的访问,以便显卡驱动能够获取到它。

这听起来直接了当,但假如我们就这么结束了,那么就会出现问题:显卡驱动可以在任何时刻对缓冲区调用 `getPixels()`,甚至是在下面这样的时机调用:

```
buffer_.draw(1, 1);
buffer_.draw(4, 1);
// <- Video driver reads pixels here!
buffer_.draw(1, 3);
buffer_.draw(2, 4);
buffer_.draw(3, 4);
buffer_.draw(4, 3);
```

当上述情况发生时,用户将看到笑脸的眼睛部分,但单对这一帧而言它的嘴巴却没了。在下一帧它又可能在其他某个地方受到干扰。结果是可怕的频闪图像。我们可以用双缓冲来修正它:

```
class Scene
{
public:
    Scene()
        : current_(&buffers_[0]),
        next_(&buffers_[1])
    {}

    void draw()
    {
        next_->clear();
        next_->draw(1, 1);
        // ...
        next_->draw(4, 3);
        swap();
    }

    Framebuffer& getBuffer() { return *current_; }

private:
    void swap()
    {
        // Just switch the pointers.
        Framebuffer* temp = current_;
        current_ = next_;
        next_ = temp;
    }

    Framebuffer buffers_[2];
    Framebuffer* current_;
    Framebuffer* next_;
};
```

现在 `Scene` 拥有两个缓冲区,它们置于 `buffers_` 数组中。我们并不直接从数组中引用它们,而是通过 `next_` 和 `current_` 这两个成员来指向数组。当我们绘图时,我们往 `next` 这个缓冲区(通过 `next_->draw()` 访问)里绘制,而当显卡驱动需要获取像素信息时,它总是从 `current_` 所指向的 `current` 缓冲区中获取。

由此,显卡驱动将不会访问到我们所正在进行处理的缓冲区。剩下的问题就在于在场景完成帧绘制后,对 `swap()` 方法的调用。它简单地通过交换 `next_` 与 `current_` 这两个指针的指向来交换两个缓冲区。当下一次显卡驱动调用 `getBuffer()` 函数时,它将获取到我们刚刚完成绘制的那块新的缓冲区,并将其内容绘制到屏幕上。再也不会有图形撕裂和不美观的问题了。

## 并非只针对图形

双缓冲模式所解决的核心问题在于对状态同时进行修改与访问的冲突。造成此问题的情况通常有两个,我们已经通过上述图形例子描述了第一种情况——状态直接被另一个线程的代码所直接访问或者打断。

还有另一种很类似且常见的情况。一个状态同时被两段代码进行修改。这会在很多地方发生:尤其是实体的AI和物理部分,在它与其他实体进行交互时会发生这样的情况,双缓冲模式往往能在此奏效。

## 没智商的AI

假设我们在为所有实体构建行为系统,这是个基于打斗漫画的游戏。游戏包含一个舞台,许多角色在其中追逐打闹。下面是我们 的演员角色类:

```
class Actor
{
public:
    Actor() : slapped_(false) {}

    virtual ~Actor() {}
    virtual void update() = 0;

    void reset()     { slapped_ = false; }
    void slap()      { slapped_ = true; }
    bool wasSlapped() { return slapped_; }

private:
    bool slapped_;
};
```

游戏需要在每一帧对演员实例调用 `update()` 以让其进行自身的处理。从用户的角度严格来说,所有的角色必须看起来是同步 地进行更新。

注解: 这是一个[Update Method](#)的例子

演员也可以通过“相互作用”与其他角色进行交互,这里的相互作用指他们可以互相扇对方巴掌。当更新时,角色可以对其他角色 调用自身的 `slap()` 方法来扇巴掌并通过调用 `wasSlapped()` 方法来获知对方是否已经被扇过巴掌。

这些角色需要一个可以交互的舞台,我们下面构建它:

```
class Stage
{
public:
    void add(Actor* actor, int index)
    {
        actors_[index] = actor;
    }

    void update()
    {
        for (int i = 0; i < NUM_ACTORS; i++)
        {
            actors_[i]->update();
            actors_[i]->reset();
        }
    }

private:
    static const int NUM_ACTORS = 3;
    Actor* actors_[NUM_ACTORS];
};
```

`Stage`允许我们往里添加角色,并提供一个简单的 `update()` 方法来更新所有角色。对于用户而言,角色开始同步地各自移动,但 从内部看,一个时刻仅有一个角色被更新。

另一点需要注意的是,每个角色“被扇巴掌”的状态在其更新结束后立即被清空重置。这是为了确保一个角色只会对一个巴掌作 出响应。

为了推动事情的进展,我们来为角色创建一个具体的子类。我们的内容很简单,它面对一个角色,不论谁给了它一巴掌,它就冲着 这个角色扇巴掌。

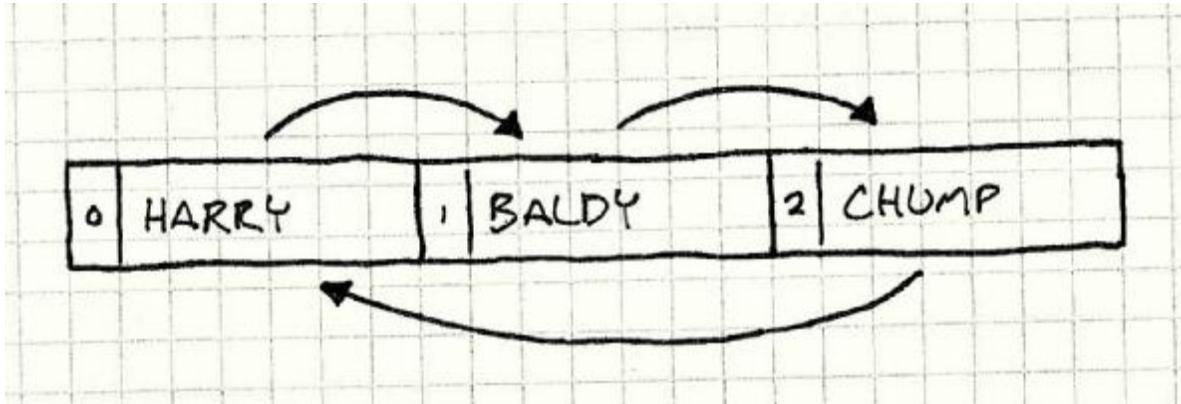
```
class Comedian : public Actor
{
public:  void face(Actor* actor) { facing_ = actor; }
virtual void update()
{
    if (wasSlapped()) facing_->slap();
}
```

```
private:  
    Actor* facing_;  
};
```

现在,让我们往舞台里放一些角色来看看会发生什么。对三个角色进行恰当的设置,使他们每个都面对着下一个,而最后一个面向第一个,组成一个圈。(译者注:即构成一个小的单循环链表, facing\_成员即为next)

```
Stage stage;  
  
Comedian* harry = new Comedian();  
Comedian* baldy = new Comedian();  
Comedian* chump = new Comedian();  
  
harry->face(baldy);  
baldy->face(chump);  
chump->face(harry);  
  
stage.add(harry, 0);  
stage.add(baldy, 1);  
stage.add(chump, 2);
```

现在舞台的布局如下图所示。箭头指明了角色所面朝的另一个角色,而数字表示角色在舞台数组中的索引号。



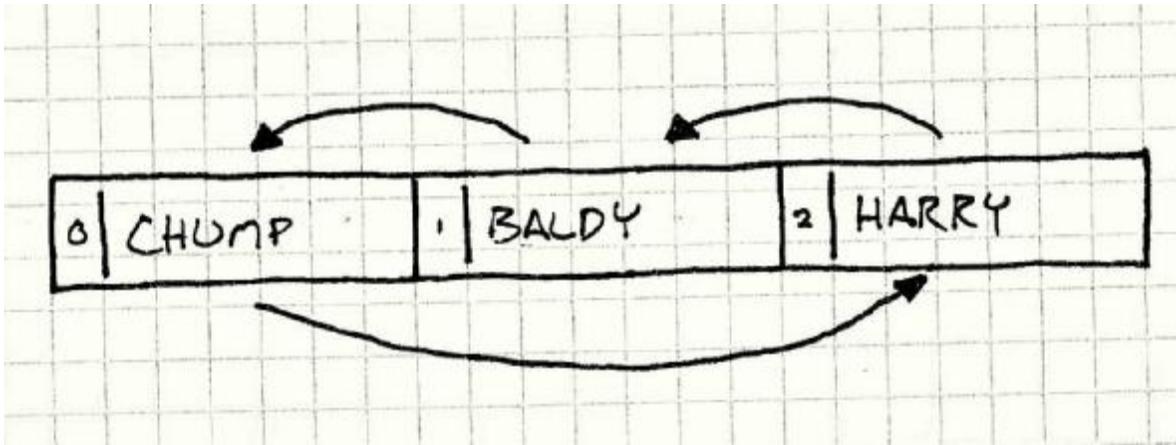
现在我们往harry脸上扇一巴掌来启动舞台,看看现在会发生些什么:

```
harry->slap();  
stage.update();
```

切记 stage 中的 update() 方法轮流对每个角色进行更新,所以假如我们浏览一遍代码,我们将推测舞台上表演的进展过程:

```
Stage updates actor 0 (Harry)  
    Harry was slapped, so he slaps Baldy  
Stage updates actor 1 (Baldy)  
    Baldy was slapped, so he slaps Chump  
Stage updates actor 2 (Chump)  
    Chump was slapped, so he slaps Harry  
Stage update ends
```

在单独一帧内,我们最开始给Harry的一巴掌传递给了所有演员。现在为了让事情更复杂些,我们把舞台上的这些演员在数组中的顺序打乱但不改变他们脸的朝向。



我们将剩余的部分交给舞台自己处理,但要将上面添加三个角色的代码替换为以下:

```
stage.add(harry, 2);
stage.add(baldy, 1);
stage.add(chump, 0);
```

让我们再来实验看看会发生什么:

```
Stage updates actor 0 (Chump)
    Chump was not slapped, so he does nothing
Stage updates actor 1 (Baldy)
    Baldy was not slapped, so he does nothing
Stage updates actor 2 (Harry)
    Harry was slapped, so he slaps Baldy
Stage update ends
```

哦!完全不一样了。问题很明显,当我们更新角色时,我们修改它们的“被掴巴掌”状态,我们也在更新中同时读取这些状态。因此在同一次舞台更新循环中,状态的修改仅仅会影响到在其后更新的那些角色。

**注解:**假如你继续更新舞台,你将看到扇巴掌的动作开始在角色之间传递,每帧传递一个。在第一帧,Harry扇了Baldy一巴掌,下一帧Baldy扇了Chump一巴掌,如此递推。

最终的结果是某个角色可能不会在被扇巴掌的这一帧做出反应也不会在下一帧做出反应——这完全取决于两个角色在舞台中的顺序。这违背了我们对角色的需求:我们希望它们同步地运转,而他们在某帧更新中的顺序是不应该对结果产生影响的。

## 缓存这些巴掌

幸运的是,我们的双缓冲模式能帮上忙。这一次,我们将缓存一系列粒度更恰当的数据:每个角色的“被掴”状态,而不是先前的那两个巨大的缓冲区对象:

```
class Actor
{
public:
    Actor() : currentSlapped_(false) {}

    virtual ~Actor() {}
    virtual void update() = 0;

    void swap()
    {
        // Swap the buffer.
        currentSlapped_ = nextSlapped_;

        // Clear the new "next" buffer.
        nextSlapped_ = false;
    }

    void slap() { nextSlapped_ = true; }
```

```

    bool wasSlapped() { return currentSlapped_; }

private:
    bool currentSlapped_;
    bool nextSlapped_;
};


```

现在每个角色有两个 `slapped_` 状态而不是一个。正如先前图形的例子一样,当前的状态用于读取,下一个状态用于写入。

`reset()` 函数被 `swap()` 方法所替换。现在,在清除交换的状态之前,角色先将下一状态复制到当前状态中,使其成为当前状态,这里还需要在 `stage` 中进行一些小改动:

```

class Stage
{
    void update()
    {
        for (int i = 0; i < NUM_ACTORS; i++)
        {
            actors_[i]->update();
        }

        for (int i = 0; i < NUM_ACTORS; i++)
        {
            actors_[i]->swap();
        }
    }

    // Previous Stage code...
};


```

现在 `update()` 函数更新所有的角色接着对他们的状态进行交换。

这样的结果是,每个角色在其被扇巴掌的那一帧中仅会看到一个巴掌。这样一来,这些角色就会表现一致而不受他们在舞台上顺序的影响。对于用户和外部的代码而言,这些角色在一帧中就是同步更新的。

## 设计决策

双缓冲模式很直白,我们上面所看到的例子也几乎将你可能遇到的问题都涵盖到了。当实现这种模式时主要会有如下两点的讨论:

### 缓冲区如何进行交换?

交换缓冲区的操作是整个过程最关键的第一步,因为在这一过程中我们必须封锁对两个缓冲区所有的读写操作。为达到最优性能,我们希望这个过程越快越好。

- 交换指向两个缓冲区的指针。

这是我们图形例子中的做法,也是处理图形双缓冲最通用的解决方案。

- 这很快。它无视缓冲区的大小,交换操作只是两个指针分配的动作。没办法让这个过程更加简化或更快了。
- 外部代码无法永久存储指向某块缓冲区的指针。这是该方法主要的约束。因为我们并没有实际移动数据,我们实际上做的是周期性地告诉其他代码库去另外一些地方找缓冲区,就像我们最初所比的舞台那样。这意味着其他代码库无法直接存储指向某个缓冲区的指针,因为过一会儿它就可能指向错误的缓冲区了。
- 这对于那些显卡希望帧缓冲区在内存中固定地址的系统来说尤其会造成麻烦。如果是那样,我们就不能采用这种方法。
- 缓冲区中现存的数据会来自两帧之前而不是上一帧的。连续几帧里在交替的两个缓冲区中进行绘制而在它们之间进行数据复制,如下:

- 你将会注意到当我们绘制第三帧时,在缓冲区中的数据来自第一帧的,而不是来自最近的第二帧。在多数情况下,这并不是问题—我们往往在绘制前会清理整个



注解: 双缓冲一个经典的用法是处理动态模糊。当前帧与先前渲染帧的一部分进行混合,以便让产生的图像更接近于真实摄像机拍摄产生的效果。

- 在两个缓冲区之间进行数据的拷贝:

假如我们无法对缓冲区进行指针重定向,那么唯一的办法就是将数据从后台缓冲区实实在在地拷贝到当前缓冲区。这就是我们在打斗喜剧中所做的。在这一情况下,我们选择此方法是因为其缓冲区仅仅是一个简单的布尔值标志位——它并不会比复制指向缓冲区的指针花去更长的时间。

位于后台缓冲区里的数据与当前的数据就只差一帧时间。这是拷贝数据方法的优点,它就像打乒乓球那样一来一回通过两个缓冲区的翻转来推进画面。假如我们需要访问先前缓冲区的数据,此方法会提供更加实时的数据以供我们使用。

- 交换操作可能会花去更多时间。这当然是个大缺点。这里的交换就意味着拷贝内存中的整个缓冲区数据块。假如缓冲区很大,比如是一整个帧缓冲区,那么进行交换就会很明显地花去一整块时间。由于在交换期间无法对缓冲区进行任何读写操作,故这是个很大的局限。

## 缓冲区的粒度如何?

另一个问题在于缓冲区其自身是如何组织的:它是单个庞大数据块还是分布在某个集合里的每个对象之中?我们在图形的例子中使用了前一形式而演员类中使用了后者。多数时候,你所要缓存的内容将会告诉你答案,当然也会有些变数。例如,我们的演员也都可以将他们的信息集中存储在一个独立的信息块中,并让演员们通过他们的索引指向其中各自的状态。

- 假如缓冲区是单个大块

交换操作很简单,因为全局只有一对缓冲区,只需要进行一次交换操作。假如你通过交换指针来交换缓冲区,那么你就可以交换整个缓冲区而无视其大小,只是两次指针分配而已。

- 假如许多对象都持有一块数据

- 交换较慢。为实现交换,我们需要遍历对象集合并通知每个对象进行交换。
- 在我们的打斗喜剧中,这是没啥问题的,因为我们总需要清理后台“被扇巴掌”的状态——每帧都必须访问到每个对象所缓存的状态。假如我们不需要访问缓存的状态,那么我们就可以对其进行优化来使其达到与使用单块大缓冲区存储一系列对象状态一样的效率。——此时的办法就是使用“当前”和“下一个”指针的概念并以此建立对象之间的联系(译者注:类似建立链表)。如下:

```
class Actor
{
public:
    static void init() { current_ = 0; }
    static void swap() { current_ = next(); }

    void slap() { slapped_[next()] = true; }
    bool wasSlapped() { return slapped_[current_]; }

private:
    static int current_;
    static int next() { return 1 - current_; }

    bool slapped_[2];
};
```

- 演员们通过current变量来访问状态数组。下个状态总是数组中的另一个索引,故我们可以通过next()来计算它。此时交换状态只需变换current的索引。聪明的地方在于swap()现在是一个静态方法——只需要调用一次,则每个演员的状态都会被交换。

## 参考

- 你几乎能在任何一个图形API中找到双缓冲模式的应用。例如OpenGL中的 swapBuffers() 函数, Direct3D中的“swap

chains",微软XNA框架在 `endDraw()` 方法中也进行的帧缓冲区交换。

# 游戏循环

## 目的

实现用户输入和处理器速度在游戏行进时间上的分离。

## 动机

假如有哪个模式是本书无法删去的，那么非游戏循环模式莫属。游戏循环模式是游戏编程模式中的精髓。几乎所有的游戏都包含着它，无一雷同，相比而言那些非游戏程序却难见它的身影。

为了解循环模式是如何大有作为，我们先来快速回顾一下内存的发展史。在那个大家都还留着络腮胡的编程年代，程序工作起来就像你家里的洗碗机——你塞进一段代码给机器，按下按钮，等待，获得输出结果，完成。那就是批处理程序——活干完了，程序也就终止了。

### 注解

络腮胡：Ada Lovelace(十九世纪中期的数学家，世界上第一位程序设计师)和海军少将 Grace Hopper(二十世纪的海军将军和计算机科学家)都留着极具纪念意义的络腮胡

今天你依然见得到它们，幸运的是今天的我们不再使用穿孔卡片来写代码。Shell脚本，命令行程序，甚至是将一堆标记性语言(Markdown)转变成这本书的那些小Python脚本都属于批处理程序。

## CPU探秘

最终程序员们意识到，这种把批处理代码丢给计算机，离开几个小时后再回来查看结果的方式在程序排错上简直慢得可怕。他们需要实时的反馈——于是交互式编程诞生了。最早的一批交互式程序就包括了像下面这样的游戏：

```
YOU ARE STANDING AT THE END OF A ROAD BEFORE A SMALL BRICK  
BUILDING . AROUND YOU IS A FOREST. A SMALL  
STREAM FLOWS OUT OF THE BUILDING AND DOWN A GULLY.  
  
> GO IN  
YOU ARE INSIDE A BUILDING, A WELL HOUSE FOR A LARGE SPRING.
```

### 注解

洞穴冒险：上面这个被称为“洞穴探险”(Colossal Cave Adventure)，史上首个冒险游戏。

你可以和这个程序面对面地交谈。它等待你的输入，并对你的操作进行响应。你也许还会回应它的反馈，你们就这么一唱一和，就像你在幼儿园里所学的那样。当轮到你时，机器就静静地呆在那儿啥也不做，就像：

```
while (true)  
{  
    char* command = readCommand();  
    handleCommand(command);  
}
```

### 注解

退出游戏：这个程序永远地循环着，因此你无法退出游戏。真实的游戏会改为诸如while (!done) 并通过设置done标志的值来退出游戏。我省去了这些来让例子看上去更简单。

## 事件循环

如果剥去现代的图形UI应用程序的外衣，你将发现它们和旧的冒险游戏是如此地相似。你的文字处理器通常什么也不做地呆着，直到你按下了某个键或者点击了鼠标：

```
while (true)
{
    Event* event = waitForEvent();
    dispatchEvent(event);
}
```

这与文本指令的主要差异在于，事件循环程序等待用户的输入事件，包括鼠标点击和键盘按键。基本上它还是像旧的文字冒险游戏那样运作，阻塞着自己等待用户输入，这是个大问题。

不同于其他大多数软件，游戏即便在用户不提供输入时也一直在跑。假如你坐下来愣盯着屏幕，游戏也不该卡住。动画依旧在播放，各种效果也在闪动跳跃，假如你运气不佳，怪物们则可能在不断地啃咬你的英雄！

### 注解

空闲状态：多数事件循环都包含一个“空闲”(“idle”)事件以便在没有用户输入时也能间歇性地处理事务，这对于闪烁的光标或者一个进度条而言已经足够了，但对于游戏而言远远不够。

这是实际游戏循环的第一个关键点：它处理用户的输入，但并不等待输入。游戏循环始终在运转：

```
while (true)
{
    processInput();
    update();
    render();
}
```

上面是最基本的结构，我们稍后再改善它。`processInput()`处理相邻两次循环调用之间的所有用户输入。接着`update()`让游戏(数据)模拟迭代一步，它执行游戏AI和物理计算(这是常见顺序)。最后`render()`对游戏进行渲染以将游戏内容展现给玩家。

### 注解

见名知意，你可能已经猜到了，`update()`方法里正是个使用[Update模式](#)的好地方。

## 时间之外的世界

假如循环不因输入而阻塞，那么试问：它运转得多快呢？游戏循环的每次执行通过某些值更新了游戏状态，从游戏世界中某个人物的视角来看，他们的时钟便往前走了一个单位。

### 注解

帧：游戏循环的一次更新可以用术语“tick”(“tick”)或“帧”(“frame”)来描述。

与此同时，玩家的实际时间也在流逝。假如用现实时间来衡量游戏循环的速度，我们就得到了游戏的“帧率”(FPS， frames per second)。假如游戏循环得很快，FPS的值便很高，游戏将会运行得十分快而流畅。反之，游戏就会拖拉得像场定格电影(stop motion movie)。

对于现在这个粗糙的游戏循环，它以其尽可能快的速度在运转。两个因素决定了帧率：

1. 循环每一帧要处理的信息量。复杂的物理运算，一堆对象的数据更新，许多图形细节等等都将让你的CPU和GPU忙个不停，这都会让一帧消耗更多的时间。

2. 平台的底层速度。越快的芯片在相同时间内处理更多的代码。多核，多GPU，专用声卡以及操作系统的定时器都影响着你

在一帧里能干多少事情。

## 秒的长短

在早期的电视游戏中，这个秒数因子是被固定的。假如你为红白机(NES)或者苹果二代电脑(Apple IIe)写游戏，你就必须对执行你游戏的CPU有深度的了解，而且你要能(且必须)为它写专门的代码。你需要好好考虑游戏的每一帧都该做些什么。

早些的游戏被精心地编写成每帧仅执行必须的任务，以便它能够在开发者期望的速度下运行。但假如你在更快或更慢一些的机器上跑这样的游戏，游戏本身会发生变速的现象。

### 注解

“turbo”：这也就是那些旧的个人电脑总带着“加速”(“turbo”)按钮的原因。新一代的个人电脑变得更快，它们将无法运行那些旧的游戏——因为这些游戏跑起来会变得很快。关闭加速按钮可以减缓它们的运行速度以便进行游戏。

而今很少有开发者对他们游戏所运行的硬件平台有深度的了解。取而代之的是，我们必须让游戏智能地(在速度上)适配于多种硬件机型。

这就是游戏循环模式的另一个要点：这一模式让游戏在一个与硬件无关的速度常量下运行。

## (游戏循环)模式

游戏循环在游戏过程中持续运转。每循环一次，它非阻塞地处理用户的输入，更新游戏状态，并渲染游戏。它跟踪流逝的时间并控制游戏的速率。

## 使用情境

对于设计模式，宁可不用也不能错用，故每一章你都能看到这一部分，以便让我们冷静下来思考。设计模式的目标可不是为了让你毫无节制地往你的程序里塞代码。

但这一模式有所不同。我拍着胸脯说你会在你的游戏里使用它。假如你使用了游戏引擎，那么这一模式无需你亲手实现，但它依然存在(于引擎中)。

### 注解

于我而言，这就是“引擎”和“库”之间的差别。使用库时，你自己把握游戏循环并在其中调用库函数，而使用引擎时它自己掌握着游戏主循环并调用你的代码。

你可能会想，我的回合制游戏应该不需要这家伙吧？不，尽管回合制游戏中，游戏状态总是随着双方回合的轮转而更新，但游戏中视觉和听觉的模块却也一直在运转，即便当你正在自己的回合犹豫着下一步行动时，动画和音效也依旧在运转。

## 使用须知

我们这里所讨论的循环是游戏中举足轻重的部分。正所谓程序90%的时间都花在10%的代码上——而游戏循环部分的代码就在这10%之中。你必须小心翼翼，并时刻考虑它的效率。

### 注解

谈论这些听起来不靠谱的统计，正是那些正牌机械或电气工程师不把我们当回事的原因吧！

## 你可能需要与操作系统的事件循环进行协调

假如你在一个带有图形UI和内置事件循环的操作系统或平台上构建游戏，那么在游戏运行时就有两个应用程序循环在执行。

它们需要很好地协作。

有时你可以对其进行控制使得游戏只执行你的游戏循环。例如，你放弃珍贵的Windows API来开发游戏，那么你的main()函数可以简化为一个游戏循环。其中你可以调用PeekMessage()处理并从操作系统中分派事件。不同于GetMessage()，PeekMessage()并不阻塞等待用户输入，所以你的游戏循环会持续地运转。

其他平台并不会轻易地让你退出事件循环。假如你以浏览器为平台，事件循环也已根植在浏览器执行模式的底层。其中事件循环负责显示，你同样要使用它来作为你的游戏循环。你可能会调用requestAnimationFrame()之类的函数以便浏览器回调回你的程序，并维持游戏的运转。

## 示例

做了这么长的介绍，游戏循环模式的代码已经不言而喻。我们将看到两个不同的实现版本，并比较它们的好坏。

游戏循环驱动着AI，渲染，和其他游戏系统，但这并不是模式本身的关键，所以这里我们将这些部分都虚化。实际上render()，update()等这些部分留给读者作为练习(值得一试!)。

## 跑，能跑多快就跑多快

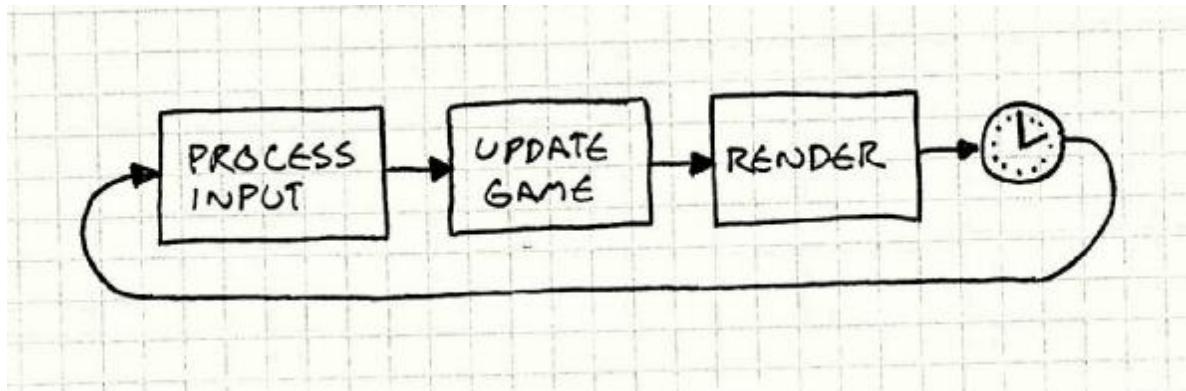
我们已经看到最简单的游戏循环：

```
while (true)
{
    processInput();
    update();
    render();
}
```

它的问题在于你无法控制游戏运转的快慢。在较快的机器上游戏循环可能会快得令玩家看不清游戏在做些什么，在慢的机器上游戏则会一样变慢变卡。假如你还加入了重量级的模块或者进行AI，物理运算，游戏实际上会更卡。

## 小睡一会儿

我们首先来看看做一点小改动会如何。假设你希望让游戏以60帧/秒运行，也就是说你大概有16毫秒的时间来处理每一帧。假如你确实能够在这16毫秒以内进行所有的游戏更新与渲染工作，你就可以以一个稳定的帧率来跑游戏。你所需要做的就是处理这一帧，接着等待下一帧的到来，如下图：



注解

1000 ms/FPS = 毫秒每帧

代码如下：

```

while (true)
{
    double start = getCurrentTime();
    processInput();
    update();
    render();

    sleep(start + MS_PER_FRAME - getCurrentTime());
}

```

这里sleep()的方法确保即便过快地处理完一帧，游戏也不会运转得太快。但这办法在游戏运行过慢时并无作为。假如一帧的更新渲染时间超过了16毫秒，睡眠的时间为负——如果我们有让时间逆流的电脑，那许多事情都会很容易，遗憾的是并没有。

这时候游戏便慢下来。你可以通过减少每帧的工作量——减少图形处理量或者在AI上耍点小聪明，甚至直接砍了AI。但即便是在一台很快的机器上，这样做也会影响游戏的质量。

## 小改动大进步

让我们再试试稍复杂点的办法。我们目前的问题可以归结为：

1.每次更新游戏花去一个固定的时间值。

2.需要花些实际的时间来进行更新。(译者注：而这个“实际时间”是机器相关的)

假如第二步的时间长于第一步，那么游戏就会变慢。例如当需要16毫秒以上的时间来更新帧速为16毫秒每帧的游戏时，就可能无法维持运行速度。但假如我们能在每一帧中进行超过16毫秒的游戏状态更新，那么我们可以不那么频繁地更新游戏并且能够追赶上游戏的行进速度。

具体想法是计算这一帧距离上一帧的实际时间间隔以作为更新步长。帧处理花费的实际时间越长，这个步长也就越长(译者注：这个步长实际上等值于帧处理花费的实际时间)。这个办法使得游戏总会越来越接近于实际时间。他们称此为浮动时间迭代(或者变值时间迭代)，代码如下：

```

double lastTime = getCurrentTime();
while (true)
{
    double current = getCurrentTime();
    double elapsed = current - lastTime;
    processInput();
    update(elapsed);
    render();
    lastTime = current;
}

```

在每一帧里，我们计算出自上次更新至今所花费的实际时间，即变量elapsed。当我们更新游戏状态时，将这个时间值传入。接下来游戏引擎必须负责将游戏世界更新到这个时间增量的下一个状态。

假设我们从屏幕左边向右边开了一枪。在固定时间迭代方法下，每帧中你根据子弹的速度移动它。在浮动时间迭代方法下，你通过时间差可以调整这个子弹的速度。随着迭代步长增加，子弹在每一帧越飞越快。于是子弹将在等同的实际时间中移动同样的距离，不论它花了20小步(较快的机器上)来完成的或4大步(较慢的机器上)来完成。

这办法看起来成功了：

1.这样一来游戏在不同的硬件上以相同的速率运行。 2.高端机器的玩家能够得到一个更流畅的游戏体验。

但，哎，我们面前还埋着个大坑：我们使得游戏变得不确定且不稳定。举个例子来说说我们自埋的坑吧：

### 注解

“确定性”表示每次你运行程序，假如给与同样的输入，那么你将得到完全一致的输出。如你所想，在具有确定性的程序

上排错要容易多了——一旦找到导致错误的输入，那么它每次都能重现BUG。

计算机天生具有确定性，它们机械地执行程序。当混乱的现实世界参杂进来时它们就比变得不确定。例如，网络，系统时钟，线程定时器等都很大程度地依赖于程序控制之外的真实世界。

假设在一个双玩家的网络游戏，Fred使用的是强大的游戏机而George用的是他祖母的古董PC机，我们之前讨论的子弹在他们的屏幕上飞来飞去。在Fred的机器上，游戏运行得飞快，也就是说每一帧处理所需的时间都极短。让我们把帧填满：假设在Fred的机器上子弹飞过屏幕共执行了50帧，那么George那苦逼的机器可能只能在这样的时间里执行5帧。

这意味着在Fred的机器上，游戏的物理引擎更新了子弹的位置50次，而George的机器只执行了5次。多数游戏采用浮点数，而它们会带来舍入误差。你每次将两个浮点数相加，其返回的结果都可能出现左右偏差。Fred的机器做了比George机器10倍多的运算，所以他累计了更多的误差。在他们的机器上，子弹将在不同的位置消失。

这只是变时迭代方法可能导致的麻烦之一，问题还多着呢。为了以实时来运行，游戏的物理引擎会做实际物理规则的近似。为了防止这近似计算“炸飞上天”，系统进行了减幅运算。这个减幅运算被小心地安排成以某个固定时长迭代进行。因此，物理引擎也将变得不稳定。

注解

“炸飞上天”(“Blowing up”)在这里取字面意思。当物理引擎出问题时，游戏中的对象可能已完全错误的速度飞到天上去。

这个例子其不稳定性只是作为一个警醒我们的例子，它会引导我们更进一步。

## 把时间追回来

渲染，是引擎中通常不会受变时迭代影响的部分。由于渲染引擎表现的是游戏时间中的一瞬间，所以它并不关心距离上次渲染过去了多少时间。它只是把当前的游戏状态渲染出来而已。

注解

这很大程度上是成立的。诸如动态模糊等效果可能受到时间迭代的影响，但假如它们出现一些偏差，玩家往往也注意不到。

这一事实可以利用。我们将使用固定时间更新，因为它使得物理引擎和AI都更加稳定。但我们允许在渲染的时候进行一些机动的调整以释放出一些处理器时间。

它这样运作：距离上次的游戏循环已经过去了一段(真实的)时间。这一段时间就是我们需要模拟的游戏“当前时间”，以便赶上玩家的实际时间。我们通过一系列的定时步骤来实现它。代码大致如下：

```
double previous = getCurrentTime();
double lag = 0.0;
while (true)
{
    double current = getCurrentTime();
    double elapsed = current - previous;
    previous = current;
    lag += elapsed;

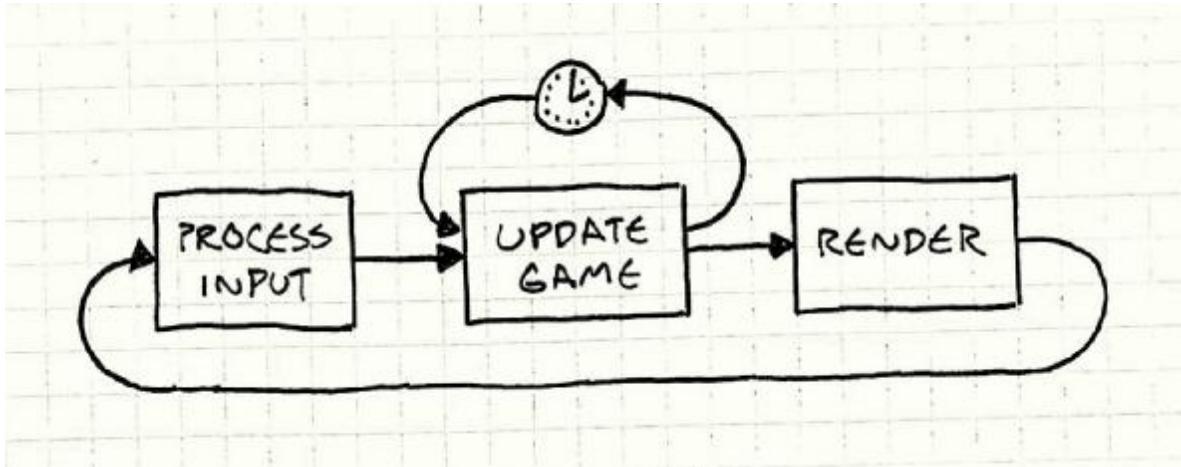
    processInput();

    while (lag >= MS_PER_UPDATE)
    {
        update();
        lag -= MS_PER_UPDATE;
    }

    render();
}
```

上述代码可分为几部分：在每帧的开始，我们基于实际流逝的时间更新变量lag。这一变量表示了游戏时钟相对现实时间落后

的量。接着我们使用一个内部循环来更新游戏，每次以固定时间进行，直到它追赶上现实时间。一旦赶上，我们渲染并进行下一次游戏循环。你可以将上述过程画图如下：



注意此时的时间步长不再是视觉上的帧率。常量MS\_PER\_UPDATE只是我们更新游戏的间隔。这一间隔越短，追赶上实际时间所花费的处理次数就越多。间隔越大，游戏跳帧越明显。理论上，你希望它足够短，通常快于60FPS，以使游戏在快的机器上维持高保真度。

但要注意的是别让它过短。你必须保证这个时间步长大于每次update()函数的处理时间，即便在最慢的机器上也须如此。否则，你的游戏便跟不上现实时间。

#### 注解

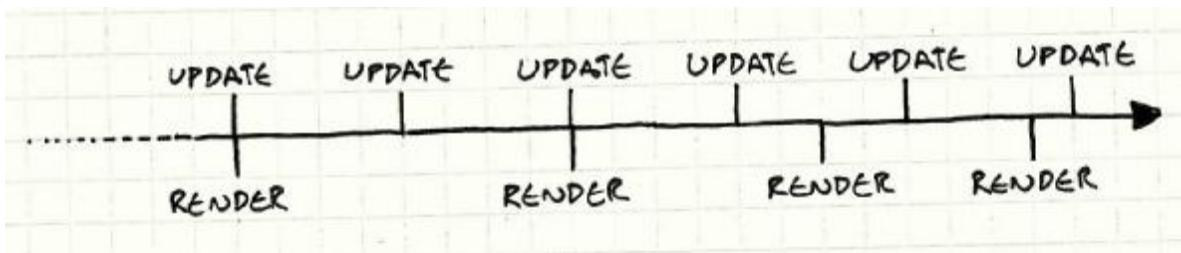
我把它就这么留在这，但你可以对其采取一些安全措施：当内部更新循环次数超出一定迭代上限时，让循环终止。这样游戏可能会变慢，但总比完全卡死好。

幸运的是，我们给予了自己一些喘息的空间。我们通过将渲染拉出更新循环之外来实现了这一点。这一方法解放了大量的CPU时间。最后的结果是，游戏通过定时步长更新，实现了在多硬件平台上以恒定速率进行游戏模拟。只不过在低端机器上玩家会看到游戏窗口里出现跳帧的情况。

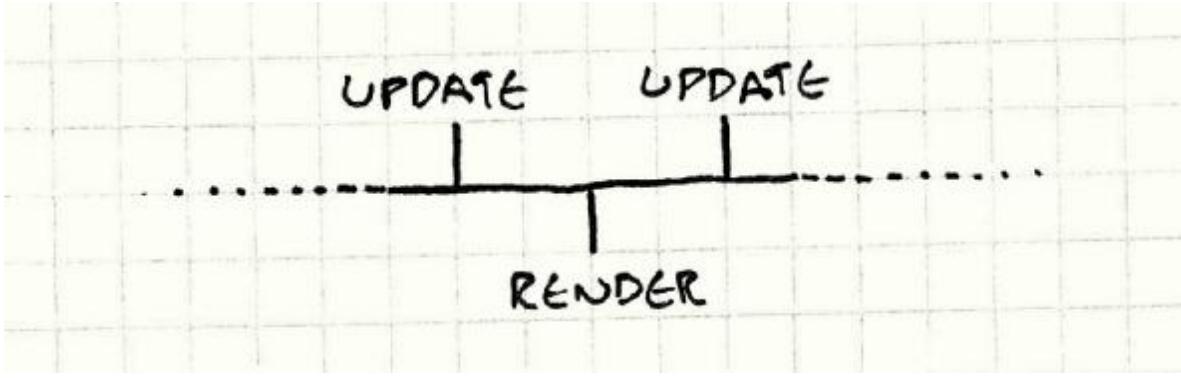
## 留在两帧之间

眼下还留有一个问题，也就是残留的延迟。我们以固定的时间步长更新游戏，但在随机的时间点进行渲染。这意味着从玩家的角度来看，游戏常会在两次更新之间展现出完全相同的画面。

让我们看看时间线：



如你所见，我们的更新十分紧凑而固定，同时我们在任何可能的时间进行渲染。渲染的频度低于更新，且不稳定。这些都没问题。问题在于我们并不总在更新的时间点进行渲染。看看第三次渲染，它介于两次更新之间：



设想一个子弹正横穿屏幕，首次更新时它在左侧，而第二次更新将它移动到屏幕右端。渲染在两次更新之间的某个时间点进行。以我们现在的实现方式，它将依然在屏幕左端。这意味着动作看起来会显得卡顿而不流畅。

顺带一提，我们实际上知道渲染时相邻两帧之间的间隔长度：也就是变量lag。当这个值小于更新时间步长时，我们跳出更新循环，而不是当lag为0时跳出。那么此时lag剩余的量呢？其实这个量就是我们进入下一帧的时间间隔。

当进行渲染时，我们将其传入：

```
render(lag / MS_PER_UPDATE);
```

#### 注解

**标准化**：这里我们将它除以MS\_PER\_UPDATE是为了将值标准化。这样传入render()的值将在0(恰好在前一帧)到1(恰好在后一帧)之间(忽略更新时间步长)。通过这一方法，渲染引擎无需担心帧率。它仅仅处理0-1值之间的情况。

渲染器知道每个游戏对象的属性以及其当前速度。假设子弹在距离屏幕左侧20像素的地方并以400像素每帧的速率向右移动，假设我们在两帧的正中间渲染，传入render()的参数值即为0.5。故它绘制了下半帧的子弹飞行情况，也就是在距离屏幕左侧220的位置。锵！流畅的动作。

当然，可能会遇到推断错误的情况。当计算下一帧时，子弹可能撞上了障碍物，或者减速了等等。我们只是设想其前一帧的位置以及下一帧可能所在的位置并在两者之间插值地渲染其位置。但除非物理引擎和AI更新完成，否则我们并不能确切地知道子弹究竟会在哪。

故推断含有猜测的成分，有时将会出错。幸运的是，这些程度的修正通常并不引人注目。至少，比起你完全不做预测时的卡顿要不起眼得多。

## 设计决策

尽管这章已经写得够长了，但我还是留下了许多额外的问题。一旦你考虑诸如与显示刷新速率的同步，多线程，GPU等因素，实际的游戏循环将会变得复杂许多。在这样的高级层面上，你可能需要考虑以下这些问题：

### 谁来控制游戏循环，你还是平台？

这是你或多或少都要回答的一个问题。假如你的游戏嵌入在浏览器里，那么你往往无法自己来写游戏循环。浏览器自带基于事件的机制已经预先包含了这一循环。类似地，假如你使用了现成的游戏引擎，你也将依赖于它的游戏循环而不是自己来控制。

- 使用平台的事件循环：
  - 这相对简单，你无须担心游戏核心循环的代码和优化问题。
  - 它与平台协作得很好。你显然无需担心它何时处理事件，如何捕获事件，或者如何处理平台与你的输入模型之间不匹配的问题等等。
  - 你失去了对时间的控制。平台将在其认为合适的时间调用你的代码。假如其频度无法达到你所预期，很遗憾。更糟

的是，许多应用程序的事件循环在概念上的设计并不同与游戏——它们通常很慢并且不连续。

- 使用游戏引擎的游戏循环：

- 你无需自己写代码。写游戏循环需要不少技巧。由于其核心代码每一帧都会执行，其微小的错误或性能问题都可能对你的游戏产生很大的影响。具有一个紧凑靠谱的游戏循环是考虑使用现存引擎的重要原因。
- 你不需要亲自来写。当然，坏消息是当出现一些与引擎循环不那么合拍的需求时，你却无法获得循环的控制权。

- 自己写游戏循环：

- 你来掌控一切。你可以做你想做的任何事。你可以完全依照你游戏的需求来设计它。
- 你需要实现平台的接口。应用程序框架和操作系统通常希望你能划分出一些时间来供它们处理事件并做一些其他事。假如你掌控你程序的核心循环，那么它们便得不到这些时间。你显然周期性地将控制权交给系统以保证应用程序的框架不会混乱。

## 你如何解决能量耗损？

五年前我们无须讨论这个问题。那时游戏运行在电视设备或专用手持设备上。但随着智能手机，笔记本电脑，移动游戏的大发展，你现在是该好好考虑这个问题了。一个跑起来很炫的游戏，但它却将玩家的手机变成一个3分钟将果汁蒸发的加热器，这可不是个让人们开心的好游戏啊。

现在你需要考虑不但要让你的游戏看来很棒，并且尽可能低减少CPU的使用率。当你完成了一帧中所做的所有工作时，你可能需要一个性能的上限来控制CPU进行休眠。

- 让它能跑多快跑对快：

你最好只在PC游戏上这么做(尽管越来越多的玩家在笔记本上跑PC游戏)。你的游戏循环从不明确地让系统休眠。任何空余的循环都要用于避免FPS或者图形保真度的不稳定。

这给予你最好的游戏体验，但它会吞噬电量。假如玩家在笔记本电脑上玩，他们需要一个很好的供电设备。

- 限制帧率：

移动游戏通常更关注游戏的运转质量而不是最高的画质。许多移动游戏会设置帧率上限(60或30FPS)。假如游戏循环在本时间片内已经完成了处理，剩余的时间它将休眠。

这给予了玩家一个足够好的体验并帮他们节省的设备能耗。

## 如何来控制游戏速度？

一个游戏循环具有两个关键部分：非阻塞的用户输入和帧时间适配。输入的问题好解决。所以关键在于你如何解决时间的问题。游戏可运行的平台数目是有限的，且多数游戏只能在其中几个平台上跑。如何适应平台变化便是关键。

### 注解

做游戏看起来像是人类的天赋之一，因为每创造出一个能进行计算的机器，我们最先做的就是在它上面开发游戏。PDP-1是一台主频2kHz的机器，仅有4096字的内存，即便如此Steve Russell和他的几个同学还是在它身上创造出了Spacewar!(译者注：世界上第一款真正意义上的娱乐性游戏，双人飞行射击游戏)。

- 非同步的定时迭代：见我们的第一个代码样例。你只需要尽可能快地执行游戏循环。

- 简单。这是这一情况的主要(呃，也是唯一的)优点。
  - 游戏速度直接取决于硬件和游戏的复杂程度。其主要缺点是假如硬件出现任何变化，将直接影响游戏速度。它就像带着死飞的游戏循环。
- 同步的定时迭代：在复杂平台上所要做的下一步是让游戏进行定时迭代，同时在循环的末尾增加一个延时，或者是同步点以防游戏运行得过快。
- 依然很简单。比起最简单的例子，只需要追加一行代码。在多数游戏中，你都希望进行同步。或许你会为图形引擎增加双缓存并让翻转缓存的操作与显示的刷新率同步。

- 这是省电的。这是移动游戏十分在意的一点。你不会希望非必要地耗损用户的电量。通过几毫秒的休眠而不是将每一帧都塞满操作，你能够省电。
- 游戏不会跑得很快。它的速度可能是填满操作的游戏循环的一半。
- 游戏可能会跑得很慢。假如更新一帧的更新和渲染花去过多的时间，游戏反馈将会变慢。由于这一模式并不将更新与渲染分离，在没有进一步优化的情况下它将很容易显露出这一缺陷。不进行外置帧渲染并同步时，游戏会变慢。
- 变时迭代：我在此提到诸多解决方法中的这一种以警示那些我曾经建议避免使用它的游戏开发者们。记住这个方法为何不好，总有助益。
  - 它能适应过快或过慢的硬件平台。加入游戏无法跟上现实速度，它将以越来越快的步伐跟上。
  - 它使得游戏变得不确定且不稳定。当然这才是根本问题。物理和网络模块往往在变时迭代下变得运转困难。
- 定时更新迭代，变时渲染：我们提及的例子中的最后一个办法也是最复杂，最具适配性的一个。它以固定时间步长进行更新，但将渲染与更新分离，并让渲染来跟进玩家的时钟。
  - 它也能适应过快或过慢的硬件平台。因为游戏能够实时更新，所以游戏状态不会落后于现实时间。假如玩家拥有顶尖的机器，它则将带来一个十分流畅的游戏体验。
  - 它很复杂。它的主要缺陷在于实际的实现还有需要工作要做。你需要协调更新迭代的步长使其在高端机上足够小(足够平滑)，同时在低端机上不会让游戏跑得太慢。

## 参考

---

- 讲述游戏循环模式的一篇经典文章是来自Glenn Fiedler的“[Fix Your Timestep](#)”。没有它这一章节就没法写成现在这样。
- Witters的文章 [game loops](#) 也值得一看。
- [Unity](#)的框架具有一个复杂的游戏循环，[这里](#)有一个对其很详尽的阐述。

# 更新方法

## 目的

通过对所有对象实例同时进行帧更新来模拟一系列相互独立的游戏对象。

## 动机

玩家所操控的强大女武神在执行任务，目标是从法师之王所长眠的埋骨地里盗取珍贵珠宝。她试探性地接近法师那法力强大的地穴入口，并将受到攻击——可实际上什么也没有，没有被诅咒的雕像向她发射光线，也没有亡灵士兵在入口巡逻。她长驱直入，轻取珠宝，然后你赢了，然后游戏结束。

嗯，这真没劲。

这个地穴需要一些守卫来绊住我们的英雄。首先，我们希望让一个复活的骷髅兵在门口来回巡逻。我想你已经猜到该怎么写代码了，你可以这样要让它来回巡逻：

注解

假如法师之王希望仆从们有更机智的表现，那么他需要复活一些聪明的家伙们

```
while (true)
{
    // Patrol right.
    for (double x = 0; x < 100; x++)
    {
        skeleton.setX(x);
    }

    // Patrol left.
    for (double x = 100; x > 0; x--)
    {
        skeleton.setX(x);
    }
}
```

此代码的问题在于，虽然怪物来回走着但玩家却看不到它。程序被一个死循环锁住，这显然会带来很差劲的游戏体验。我们所希望的是骷髅兵每一帧走一步，以保证在骷髅守卫巡逻时，游戏能持续地进行渲染并对玩家的输入做出反应。如：

注解

当然，游戏循环(Game Loop)是本书介绍的另一种设计模式

```
Entity skeleton;
bool patrollingLeft = false;
double x = 0;

// Main game loop:
while (true)
{
    if (patrollingLeft)
    {
        x--;
        if (x == 0) patrollingLeft = false;
    }
    else
    {
        x++;
        if (x == 100) patrollingLeft = true;
    }
}
```

```
    skeleton.setX(x);

    // Handle user input and render game...
}
```

我之所以列出前后两个版本，是为了告诉读者代码是如何变复杂的。向左,右巡逻本是两个相互独立的循环，骷髅依赖于循环的执行来保持对自己巡逻方向的跟踪。为达到逐帧处理的目的，我们必须逐帧跳出游戏循环并随后(在下一帧时)返回循环内以继续，在此必须借助变量 `patrollingLeft` 以在循环内外维持对其方向的跟踪。

但这至少奏效，我们接着前进。一堆无脑的骨头可不会对你的女武神造成什么威胁，于是接下来我们为它加入一些魔法状态，这将使它能频繁地向我们的女武神释放闪电和火光，让她措手不及。

时刻记得我们的风格——“最简单地写代码”，于是我们这么写：

```
// Skeleton variables...
Entity leftStatue;
Entity rightStatue;
int leftStatueFrames = 0;
int rightStatueFrames = 0;

// Main game loop:
while (true)
{
    // Skeleton code...

    if (++leftStatueFrames == 90)
    {
        leftStatueFrames = 0;
        leftStatue.shootLightning();
    }

    if (++rightStatueFrames == 80)
    {
        rightStatueFrames = 0;
        rightStatue.shootLightning();
    }

    // Handle user input and render game...
}
```

你会发现这代码的可维护性不高。我们维护着一堆其值不断增长的变量，并不可避免地将所有代码都塞进游戏循环里，每段代码处理一个游戏中特殊的实体。为达到让所有实体同时运转的目的，我们把它们给糊成一团了。

#### 注解

一旦当你的代码构架可以确切地用“糊作一团”来形容，那你可遇到麻烦了。

你可能猜到我们所要运用的设计模式该干些什么了：它要为游戏中的每个实体封装其自身的行为。这将使游戏循环保持整洁并便于往循环中增加或移除实体。

为了做到这一点，我们需要一个抽象层，为此定义一个 `update()` 的抽象方法。游戏循环维护对象集合，但它并不关心这些对象的具体类型。它只是更新它们。这将每个对象的行为从游戏循环以及其他对象那里分离了出来。

每一帧，游戏循环遍历游戏对象集合并调用它们的 `update()`。这在每帧都给予每个对象一次更新自己行为的机会。通过逐帧调用 `update` 方法，这些对象的表现得到同步。

#### 注解

有些爱挑刺的人会说，它们并不是真正意义上的行为同步，因为一个对象更新时其他对象都不在更新——让我们后面再来深入这个问题。

游戏循环维护一个动态对象集合，这使得向关卡里添加或移除对象十分便捷——只要往集合里增加或移除就好。问题已解决，我们甚至可以将关卡文件用某种文件格式存储，以供我们的关卡设计师们使用。

# (更新方法)模式

游戏世界维护一个对象集合。每个对象实现一个更新方法来在每帧模拟自己的行为。而游戏循环在每帧对集合中所有的对象调用其更新方法以实现同步的游戏世界更新。

## 使用情境

假如把游戏循环比作有史以来最好的东西，那么更新方法模式就会让它锦上添花。许多游戏都通过这样或那样的形式来使用这一设计模式，以构造出许多鲜活的游戏实体来与玩家进行交互。像游戏里的太空战士，龙，火星人，幽灵或者运动员们，他们正适合使用这一设计模式。

然而，假如这个游戏更加抽象，那些移动的对象并不像是生物而更像是西洋棋子，那么这一模式就不那么适用了。在一个类似西洋棋的游戏里，你并不需要同时模拟所有对象，而且你很可能也没必要让棋子们逐帧地更新自身。

### 注解

或许你无须逐帧更新它们的行为，但即便是在棋类游戏中，你也很可能需要逐帧更新它们的动画。这一设计模式同样可以帮到你。

更新方法模式在如下情境最为适用：

- 你的游戏中含有一系列对象或系统需要同步地运转。
- 各个对象之间的行为几乎是相互独立的。
- 对象的行为与时间相关。

## 使用须知

这一设计模式相当简单，所以它并没有什么值得发现的惊喜。当然，每行代码也都有它的意义在。

### 将代码划分至单帧之中使其变得复杂

比较先前的两个代码块，第二个显得更加复杂。二者都只是让骷髅守卫来回行走，但第二个代码块将控制权分派给了游戏循环的每一帧。

这一变化几乎在处理用户输入，渲染以及其它游戏循环所关心的事情时是必不可少的，所以第一个例子并不实用。例一的价值在于让我们牢记，这样处理你对象的表现，你将面临着复杂而巨大的开销。

### 注解

我所说“几乎”，是因为有时你也可以兼得鱼与熊掌。你可以直接为你的对象行为编码而不让这些函数返回，同时使这样一系列的对象与游戏循环保持同步运转。

要想实现这一点，你就必须使用多线程来让这些对象同时运转。假如一个对象可以在处理时中途暂停并继续，你可以用更强制的方式来执行而不必完全让函数结束返回。

实际中的线程往往对我们的例子而言过于繁重，但假如你的语言支持轻量的并发性组件诸如生成器，协程，Fibers(Node.js)，那可以考虑使用它们。

Bytecode设计模式是在应用程序层创建多线程的另一种选择。

### 你需要在每帧结束前存储游戏状态以便下一帧继续

在第一段示例代码中，我们并无任何指明守卫移动方向的变量。方向完全取决于当前执行的是哪一段代码。

当我们将其改造为逐帧更新的形式时，我们需要创建一个`patrollingLeft`变量来跟踪这个行走方向。当我们脱离内部代码，我们就无法获知行走的朝向，所以说我们需要存储足够的帧信息以便下一帧能够继续执行。

State设计模式在这里通常能帮上忙，因为状态机（正如其名）存储了那些能够让你在下一帧继续处理的游戏信息。

## 所有对象都在每帧进行模拟，但并非真正同步

在本设计模式中，游戏循环在每帧遍历对象集并逐个更新对象。在`update()`的调用中，多数对象能够访问到游戏世界的其他部分，包括那些正在被更新的其他对象。这意味着，游戏循环遍历更新对象的顺序意义重大。

假如A对象在对象列表中位于B对象的前面，那么当A更新时，它将会看到B停留在前一帧的状态。但当B更新时，它看到的却是A在这一帧的新状态，因为A在这一帧已经被更新了。尽管从玩家的视角来看，所有的事物都同时在运转，但游戏的核心仍然是回合制的——只不过这时两回合之间的间隔只有一帧的时间。

### 注解

假如由于某些原因你希望回避这一有序性，你可能会需要Double Buffer模式的帮助。这一模式将使得A, B的更新顺序不再重要，因为它们都能够获取到前一帧的状态。

考虑到游戏逻辑，更新分先后顺序这是件好事。平行地更新所有对象会将你带向语义死角，设想西洋棋盘上黑白棋子同时移动，它们都想往一个当前空白的位置移动，这该怎么办？

序列化地更新解决了这一问题——每次更新增量式地改变游戏世界，从一个有效的状态到下一个，不会产生对象状态的歧义而需要去进行调解。

### 注解

这同样在线游戏模式起作用，因为你需要一串序列化的动作数据来在网际间进行传输。

## 在更新期间修改对象列表时必须谨慎

当你使用这一模式时，大量的游戏表现将在这些更新方法中完成。这里面常常包含着从游戏中增加或移除对象的代码。

例如，假设一个骷髅卫兵被杀死时会掉落一个物品，对于一个新对象，你通常可以直接将它加入到列表的尾部而不会产生问题。循环继续，最终你能够在循环的末尾找到这个新对象并更新它。

但这意味着在这个新对象产生的那一帧它也有机会进行更新，而此时玩家尚未看到这个物品。假如你不希望这样的情况发生，一个简单的办法就是在遍历之前存储当前对象列表的长度，而在这一次循环仅更新列表前面这么多的对象：

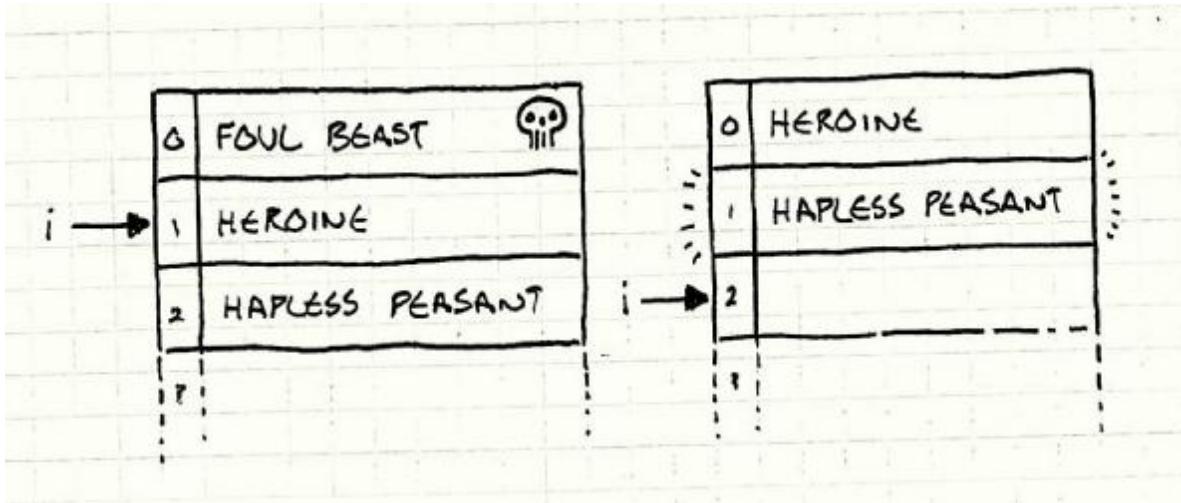
```
int numObjectsThisTurn = numObjects_;
for (int i = 0; i < numObjectsThisTurn; i++)
{
    objects_[i]->update();
}
```

上例中，`objects`是游戏中可更新对象的数组而`numObjects`是它的长度。当增加新的对象时，这个长度变量增长。我们在循环的一开始将长度缓存在`numObjectsThisTurn`变量中，从而使这一帧的循环迭代在遍历到任何新增对象之前停止。

一个令人担忧的问题是在迭代时移除对象。你希望让一只肮脏的野兽从游戏中消失，而这时候需要从对象列表中移除它。假如在对象表中，它碰巧位于你当前所更新的对象之前，你可以投机地跳过它。

```
for (int i = 0; i < numObjects_; i++)
{
    objects_[i]->update();
}
```

这一简单的循环通过对象下标索引的递增来更新每个对象。下面的示例图中左侧展示了当我们更新女主角时对象数组的变化。



我们更新她时， $i$ 等于1，她斩杀了肮脏的野兽所以它从数组中被移除。而女武神移动到 $i$ 为0的位置，而倒霉的农夫被前移到1的位置。在女武神更新结束后， $i$ 增长到2。如上图右侧所示，倒霉的农夫在循环中被跳过并且永远也不会更新了。

#### 注解

一个简便的解决方法是当你更新时从表的末尾开始遍历。此方法下移除对象，只会让已经更新的物品发生移动。

另一方法是在移除对象时多加小心，并在更新任何计数器时把被移除的对象也算在内。还有一个办法是将移除操作推迟到本次循环遍历结束之后。将要被移除的对象标记为“死亡”，但并不从列表中移除它。在更新期间，确保跳过那些被标记死亡的对象，接着等到遍历更新结束，再次遍历列表来移除这些“尸体”。

假如在更新循环中你加入了多线程，采用延迟修改的方法较好，因为这可以避免更新期间线程同步带来的巨大开销。

## 示例

这一模式十分浅显，从例子里我们就能看出其要点。这并不意味着它没用，而正因为它的简单才使得它好用——它是一个简明而不加任何修饰的解决方案。。但为了更具体地阐明此方法，我们还是来看一个基本的实现例子。让我们从这个代表着骷髅和雕像的Entity类来开始吧：

```
class Entity
{
public:
    Entity()
        : x_(0), y_(0)
    {}

    virtual ~Entity() {}
    virtual void update() = 0;

    double x() const { return x_; }
    double y() const { return y_; }

    void setX(double x) { x_ = x; }
    void setY(double y) { y_ = y; }

private:
    double x_;
    double y_;
};
```

在这个类里我并没有加入太多东西，只有那些后面能用到的成员。实际的项目中还将包含有诸如图形和物理的部分。而上面的类中最重要的部分就是这一设计模式所要求的update()抽象方法。

游戏维护一系列这样的实体，在我们的例子中，我们将它们置入一个代表游戏世界的类中：

```
class World
{
public:
    World()
        : numEntities_(0)
    {}

    void gameLoop();

private:
    Entity* entities_[MAX_ENTITIES];
    int numEntities_;
};
```

注解

在一个实际的游戏项目中，你可能会用到一个实际的集合类，但在此我仅使用普通的数组来让事情简单些。

一切准备就绪，遍历实体逐帧更新的游戏实现如下：

```
void World::gameLoop()
{
    while (true)
    {
        // Handle user input...

        // Update each entity.
        for (int i = 0; i < numEntities_; i++)
        {
            entities_[i]->update();
        }

        // Physics and rendering...
    }
}
```

注解

见名知意，这就是游戏循环(Game Loop)模式的例子。

## 子类化的实体？！

肯定有些读者现在很不舒服，因为我在那里对主要的Entity类采用了继承的方式来定义不同的行为。假如你碰巧遇到了问题，我将会提供一些文章。

随着游戏产业从最初6502汇编语言和VBLANK(老式的阴极射线管)显示器的海洋到OOP(面向对象)上岸，开发者们陷入了一场软件架构的狂热，其中之一就是对继承的使用。高耸而错综复杂的类继承大厦被建立起来，遮天盖地。

而事实证明继承真是个恐怖的想法，没人能够在不拆解的情况下维护一个庞大的继承关系，甚至连GoF(最杰出的4个软件设计师，合著有《设计模式：可复用面向对象软件的基础》一书)都在1994年发现了这一点，并写道：

Favor ‘object composition’ over ‘class inheritance’. (组合对象，而不是类继承)

注解

在我之间，我想子类继承的问题离我们甚远。我几乎避开了它，但执着于避免使用继承就和执着于使用它一样糟。  
你完全可以适度使用它而不必完全禁用。

当游戏产业中的人们纷纷意识到类继承糟糕的一面时，Component设计模式应运而生。借此，update()方法能够置于实体的组件之中而非依附Entity本身。这将帮助你避免为了定义和复用不同表现的实体类而构建出复杂的实体类继承关系。取而代之的是用各种组件来组装这些子类。

假如我在实际开发一款游戏，我也会这么做。但这一章并不讨论组件模式而是update()方法，我尽可能简洁并快速地表达出它们，而将这个方法直接放在Entity类里并进行一两个子类的继承就是最快的方法。

注解

组件模式请看[这里](#)

## Defining entities 定义实体

回到正题，我们最初的动机是要定义一个骷髅守卫，和能放出电光石火的魔法雕像。从我们的骷髅朋友开始吧。为了定义其巡逻行为，我们通过恰当地实现 update() 方法来创建新的实体类。

```
class Skeleton : public Entity
{
public:
    Skeleton()
    : patrollingLeft_(false)
    {}

    virtual void update()
    {
        if (patrollingLeft_)
        {
            setX(x() - 1);
            if (x() == 0) patrollingLeft_ = false;
        }
        else
        {
            setX(x() + 1);
            if (x() == 100) patrollingLeft_ = true;
        }
    }

private:
    bool patrollingLeft_;
};
```

如你所见，我们所做仅仅是将游戏循环中复制代码并将它粘贴到Skeleton类的 update() 方法中。一个小差异在于这里 patrollingLeft\_ 被变成了一个类成员而非局部变量。借此便能保证，这一变量在 update() 方法调用期间有效。

我们对Statue类如法炮制：

```
class Statue : public Entity
{
public:
    Statue(int delay)
    : frames_(0),
      delay_(delay)
    {}

    virtual void update()
    {
        if (++frames_ == delay_)
        {
            shootLightning();

            // Reset the timer.
            frames_ = 0;
        }
    }

private:
    int frames_;
    int delay_;

    void shootLightning()
    {
        // Shoot the lightning...
    }
};
```

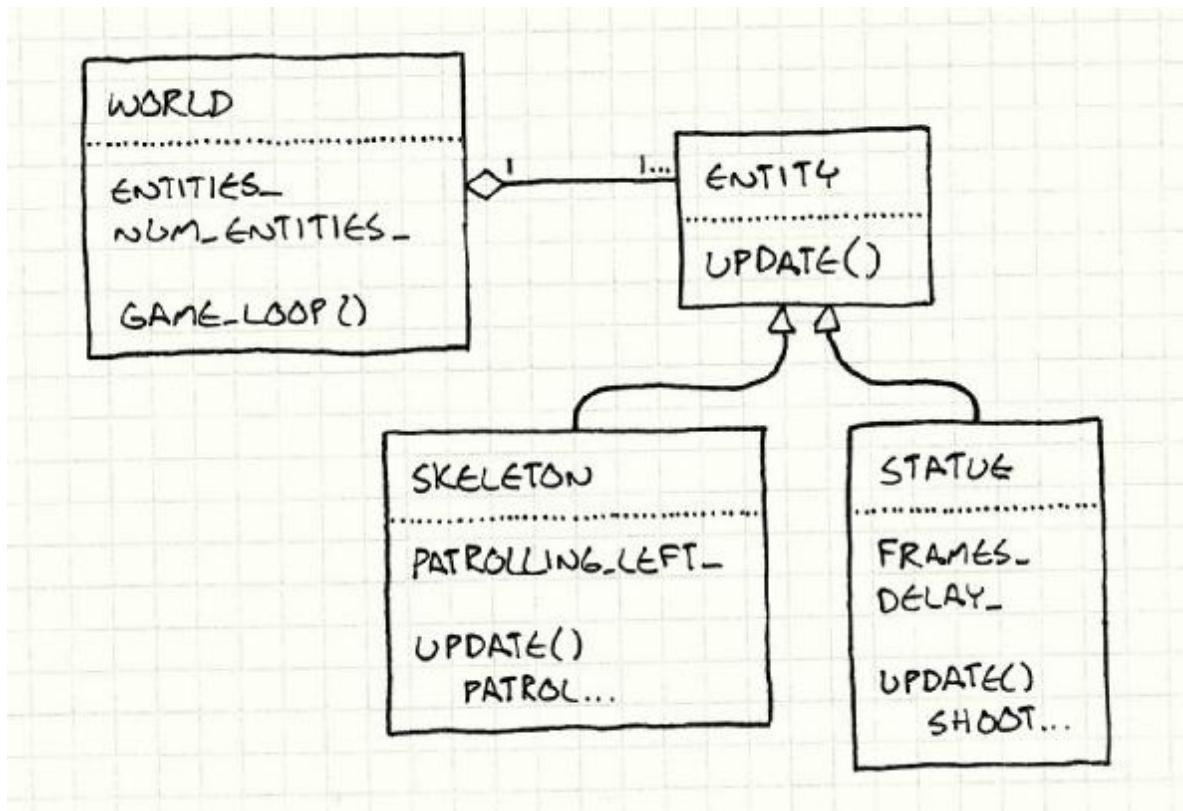
再一次，最大的改动就是从游戏循环中移出一些代码以及对变量重命名。这样一来，我们让代码更加简洁——在原来不可避免杂乱的代码中，本会出现许多存储不同雕像帧计数器和开火频率的局部变量。

既然这些都已经被移动到Statue类之中，你可以随心所欲地创建Statue的实例，而它们各有自己的计时器。这正是本设计方法背后的本意——现在向游戏世界中添加实体更加容易了，因为每个实体都携带着所有自己所必须的东西，自给自足。

这一模式让我们避免了在扩展游戏时采用继承。这一模式反倒让我们可以单独地使用数据文件或者关卡编辑器来扩展游戏世界。

#### 注解

还有人关心UML图么？如果还有那上图就对应着我们所创建的类结构的UML图。



## 逝去的时间

这是游戏核心的设计模式，但我只是做了其最常用部分的提炼。至此，我们都假设每次对`update()`的调用都让整个游戏世界向前推进相同（固定）的时间长度。

我更喜欢这种方式，但多数游戏使用变时长的方式。那样做，每次游戏循环可能会占用更多或更少的时间，具体取决于其处理更新和渲染前一帧所的开销。

#### 注解

Game Loop 这一章详述了定时和变时步长的优劣。

这意味着每次 `update()` 的调用需要知道虚拟时钟所流逝的时间，于是你常会看到流逝的时间会被作为参数传入。例如，我们可以像下面那样让骷髅卫兵处理一个变时步长更新：

```
void Skeleton::update(double elapsed)
{
    if (patrollingLeft_)
```

```

{
    x -= elapsed;
    if (x <= 0)
    {
        patrollingLeft_ = false;
        x = -x;
    }
}
else
{
    x += elapsed;
    if (x >= 100) {
        patrollingLeft_ = true;
        x = 100 - (x - 100);
    }
}
}

```

现在，骷髅移动的距离随着时间间隔而增长。你同样能看到处理变时步长时额外增加的复杂度。骷髅可能在很长的时间差下超出其巡逻范围，我们需要小心地对这一情况进行处理。

## 设计决策

---

这样一个简单的设计模式，并无太多可选项。但它也仍有选择的余地~

### update方法依存于何类中？

---

你显然必须决定好该把update()放哪。

#### 1. 实体类(父类)中

假如你已经创建了实体类，那么这是最简单的选项。因为这不会往游戏中增加额外的类。假如你不需要很多种类的实体，那么这种方法可行，但实际项目很少这么做。

每当希望实体有新的表现时就创建子类，这会积累大量的类而导致项目难以维护。你最终会发现你希望通过一种单一继承层次的优雅映射方式来复用代码模块，那时候你就该傻眼了。

#### 2. 组件类中

假如你已经使用了Component模式，那么傻瓜也知道怎么做了——让每个组件更新其自身。和更新方法模式一样地将每个游戏实体从游戏世界中分离出来，组件方法也让各个组件部分分离开来。渲染，物理，AI可以各自顾好自己。

#### 3. 代理类中

将一个类的表现代理给另一个类这涉及到其他几种设计模式。应用State 设计模式可以让你通过改变一个对象的代理来改变其行为。[Type Object设计模式](#)可以让你在多个同类型的实体之间共享行为表现。

假如你使用上述任一种设计模式，那么自然而然地需要将 update() 方法置于代理类中。这么一来，你可能在主(父)类中仍保留 update() 方法，但它成为非虚的方法并将指向代理类对象的update方法，如：

```

void Entity::update()
{
    // Forward to state object.
    state_->update();
}

```

这么做让你能在代理类之外定义新的行为方式。正像使用[Component模式](#)那样，这为你开辟了灵活定义新类和新的行为方式的途径。

### 那些未被利用的对象该如何处理？

---

你常需要在游戏中维护这样一些对象：不论出于何种原因，它们暂时无需被更新。它们可能被禁用，被移除出屏幕，或者至今尚未解锁。假如大量的对象处于这种状态，这可能会导致CPU每一帧都浪费许多时间来遍历这些对象同时又对他们毫无作为。

一种方法是单独维护一个需要被更新的“存活”对象表。当一个对象被禁用时，将它从其中移除。当它重新被启用时，把它加回表中。这样做，你只需遍历那些实际上有作为的对象。

- 假如你使用单个集合来存储所有游戏对象：

- 你在浪费时间。对于暂时无用的对象，你需要检查它们“是否死亡”的标志，或者调用一个空方法。

#### 注解

除了浪费CPU循环来检查对象是否被激活并跳过它的问题，空指针问题还可能搞坏你的缓存区。CPU通过加载读取RAM中的数据到更快的单片缓存来加载内存，这是基于投机地假设在一段时间内你读取的内存是连续的情况下进行的。当你跳过一个对象时，你可能会跳过缓存区(存储当前这个对象)的末尾，而让CPU再去另一块内存寻址。

- 假如你对活跃对象单独用一个集合来维护：

- 你将使用额外的内存来维护这第二个集合。因为往往你需要一个主集合来维护所有的对象，以便在需要所有对象能够访问它们全部。这么说来，这额外的集合在技术上是多余的。当游戏对速度的要求比对内存的要求高时(往往是这样的)，这样的取舍还是值得的。
  - 另一种缓和此问题的办法是，同样维护两个集合，但另一个只维护那些未被激活的对象，而不是维护所有对象。
  - 你必须保持两个集合同步。当对象被创建或者销毁(并非临时禁用而是永久销毁)时，你必须记住同时修改主集合和活跃对象集合。

这里该使用什么方法，取决于你对非激活对象数目的预估。其数目越多，就越需要创建一个独立的集合来存储他们以便在游戏循环时避免处理这些非激活对象。

## 参考

- 这一模式与[Game Loop](#)和[Component](#)模式共同构成了多数游戏引擎的核心部分。
- 当你开始考虑实体集合或循环中组件在更新时的缓存效能，并希望它们更快地运转，[Data Locality模式](#)将会有所帮助。
- [Unity](#)的引擎框架在许多类模块中使用了本模式，包括[MonoBehaviour](#)类。
- 微软的[XNA](#)平台在Game和GameComponent类中均使用了这一模式。
- [Quintus](#)的基于JavaScript的游戏引擎在其主Sprite类中使用了这一模式。

# 行为模式

---

搭好了游戏框架，并加入各种角色和物件之后，剩下的事就是让场景活动起来。为此你需要定义行为——让游戏中的每个实体知道该做些什么。

当然了，任何代码都是行为，所有的软件都在定义行为，但游戏中的行为如呼吸一样必要。一个文本编辑软件有各种各样的功能，但比起一般RPG游戏中的各种人物、物品和任务来说，就显得无趣多了。

本章中的模式，可以帮助你快速提炼并定义大量可维护的行为。[类型对象](#)无需定义实际的类，就可以创建各种类型的行为。[子类沙箱](#)提供了一批安全的原子方法去定义各种行为。最高级的选择是[字节码](#)，它可以将行为从代码完全转移到数据中去。

## 模式

---

- [字节码](#)
- [子类沙箱](#)
- [类型对象](#)

===== [上一节](#)

[目录](#)

[下一节](#)

# 字节码

---

## 目的

---

通过将行为编码成虚拟机指令，而使其具备数据的灵活性。

## 动机

---

我曾参与一款有600万行C++代码的游戏。比较起来火星探测车“好奇号”的控制软件的代码量还不及它的一半。

制作游戏很有趣，但也不容易。现在的游戏需要庞大复杂的代码库。主机厂商和应用商店有严格的质量要求，一个造成崩溃的Bug就可能导致你的游戏无法发布。

同时，我们希望将平台的性能发挥到极致。游戏的发展推动着硬件发展，我们当然要不遗余力地进行优化来赶上发展的脚步。

为了提高高稳定性和效率，我们会选择像C++这样的重量级语言。它们兼具充分利用硬件的能力以及可以阻止或拦截Bug的强类型系统。

我们可以为此感到骄傲，但它也有代价。多年的专业训练才能造就一个精通的程序员，随后你又必须面对庞大的代码库。大型游戏的编译时间可以短到“喝杯咖啡”，也可以长到把“自己煮咖啡豆、磨咖啡豆、倒咖啡、打奶泡、练拿铁的拉花”都给搞定。

除了这些挑战外，游戏还有个额外的苛求：有趣。玩家需要的是既新奇又具平衡性的体验。这就需要持续迭代。如果每个小修小改都得一个工程师去动底层代码，然后等待漫长的重编译，那实际上你已经毁了整个创作流程。

## 魔法大战！

比如说，我们在开发一款基于魔法的战斗游戏。两个对峙的法师不断向对方释放法术直到分出胜负。我们可以在代码中定义法术，但这意味着每次修改都需要工程师介入。当一个设计师想要改些数值并测试效果，就需要重新编译整个游戏，重启然后重新进入战斗。

在游戏发布之后，我们得像其他游戏一样去更新它，包括修正Bug以及添加内容等。如果所有的法术都被硬编码，一次更新就等价于发一次可执行文件的补丁。

进一步，设想提供一个MOD系统以供用户自己创建法术。如果它们都在代码里面，那这些用户都需要有完整的编译工具链去构建游戏，我们得公开所有源码。况且，如果他们的法术有Bug，其他玩家可能受到殃及而造成游戏崩溃。

## 先数据后编码

很明显，我们引擎所使用的编程语言不适合解决这个问题。我们需要把法术从游戏核心移动到安全沙箱中。我们要给让它们易于修改，易于重新加载并且在物理上与游戏的可执行文件相分离。

这种形式在我看来更像是种数据，你或许也会这么想。我们可以在单独的数据文件中定义行为，游戏引擎可以某种方式加载并“执行”它们，那么问题就解决了。

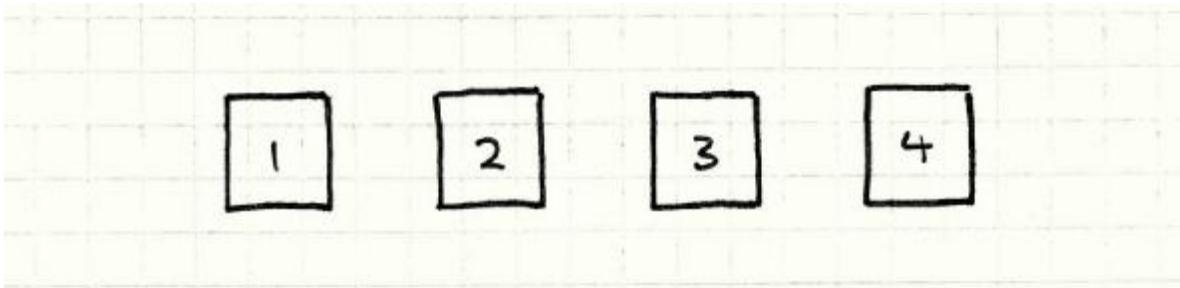
我们只需要弄明白对于数据，何谓“执行”。怎样才能让文件中的字节表示行为呢？有好几种方法。参照一下[解释器模式](#)，你就能对此模式的优缺点全貌有个大致了解。

## 解释器模式

本来这个模式我可以写成一整章的，但是Gof早已替我写了。所以这里我仅做简述。我们从一个语言开始——比如某种编程语言——你要执行它。例如它支持下面的数学表达式：

```
(1 + 2) * (3 - 4)
```

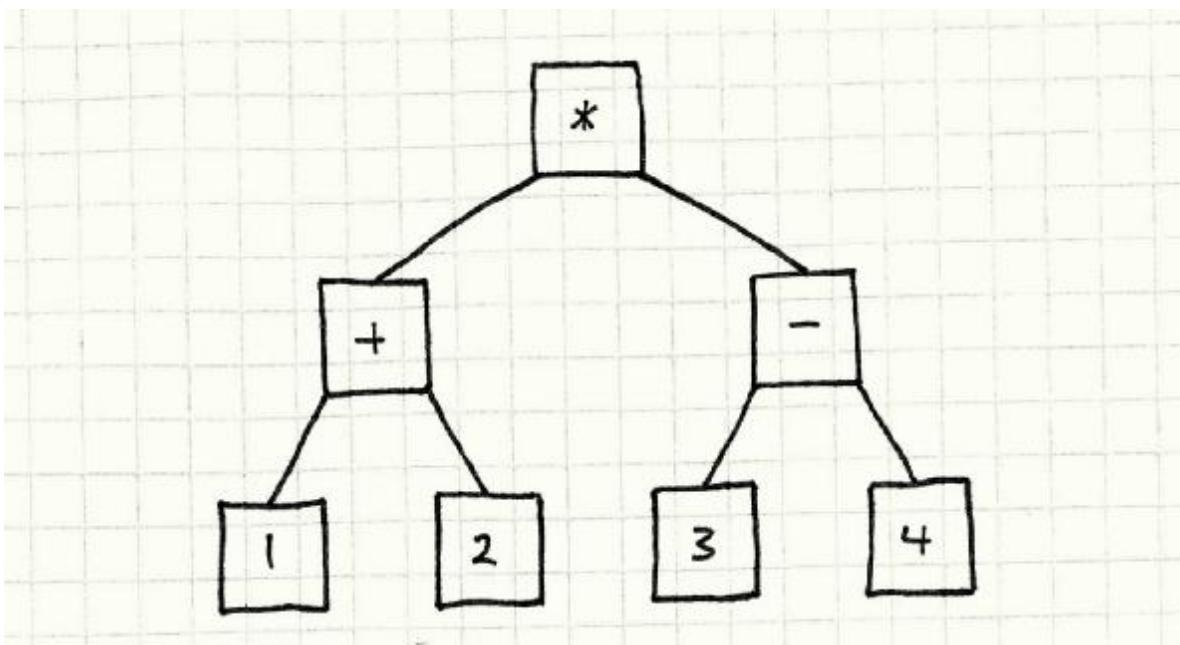
然后，你拿出表达式中的每个片段、语言语法中的每个规则，将它们变成对象。数字的对象就是它们的字面值。



简单来说，它们是在原始数值的基础上，做了个小封装。运算符也是对象，它们拥有对操作数的引用。如果你使用括号来控制优先级的话，这个表达式又变成了一棵小对象树：

这个“变化”究竟是什么？很简单——解析。解析器接受输入文本字符串，然后将它变成抽象的语法树，即一组用于表示文本语法结构的对象。

把上述内容堆积起来，你就完成了编译器一半的工作。



解释器模式与创建语法树无关，它只关心如何执行它。它的处理很聪明，树中的每个对象都是表达式或子表达式。在面向对象风格中，表达式会计算它们自己的值。

首先，定义一个所有表达式都要实现的基础接口。

```
class Expression
{
public:
    virtual ~Expression() {}
    virtual double evaluate() = 0;
};
```

然后为每一个语法定义实现这个接口的类。其中，最简单的是数字：

```
class NumberExpression : public Expression
```

```

{
public:
    NumberExpression(double value)
    : value_(value)
    {}

    virtual double evaluate()
    {
        return value_;
    }

private:
    double value_;
};

```

一个字面数字表达式的值就等同于它的数值。加法和乘法要稍微复杂一些，因为他们包含子表达式。它们需要先递归计算出所有子表达式的值，之后才能计算出它们自己的值。像这样：

```

class AdditionExpression : public Expression
{
public:
    AdditionExpression(Expression* left, Expression* right)
    : left_(left),
      right_(right)
    {}

    virtual double evaluate()
    {
        // Evaluate the operands.
        double left = left_->evaluate();
        double right = right_->evaluate();

        // Add them.
        return left + right;
    }

private:
    Expression* left_;
    Expression* right_;
};

```

显然，只要几个简单的类，就能够表达任何复杂的算术表达了。我们只用创建几个对象，并将它们正确得关联起来。

Ruby在大概15年前就是这么实现的。到了1.9版本，它们改成了本章所说的字节码。看我替你省了多少时间！

这个模式虽然简单漂亮，但是也有些问题。回头看看上面的插图，你看到了些什么？很多小盒子，以及它们之间的箭头。代码用一个微小对象构成的蔓生分形树来表达，会有一些副作用：

- 从磁盘加载需要实例化并串联成堆的小对象。
 

如果你想自己算算的话，别忘了算上虚函数表。
- 些对象和它们之间的指针占用大量内存。在32位机上，即使不考虑内存对齐，这个小小的表达式也要占用68字节(4字节/指针\*17个指针)。
 

要了解更多关于缓存以及它如何影响性能的原理，看看[数据局部性](#)这一章。
- 从每个指针遍历表达式都会废了你的数据缓存，而虚函数调用也会对指令缓存造成很大压力。

一个字概括，慢！大量的编程语言不采用解释器模式，就是因为它又慢又占内存。

## 虚拟机器码

回到我们的游戏。当它运行时，计算机并不会去遍历C++语法结构树，而是执行我们在编译期编译成的机器码。那么为什么要采用机器码呢？

- 高密度。它是坚实持续的二进制数据块，不浪费任何一个字节。
- 线性。指令被打包在一起顺序执行。不会在内存中跳跃访问（当然了，除非你确实在做流程控制）

- 底层。每个单独的指令仅仅完成一小个动作，各种有趣行为都是这些小动作的组合。
- 高速。以上几点让机器码疾行如风（当然还得算上机器码由硬件实现这一点了）。

听上去激动人心，但我们不想直接用机器码来编写法术。为用户提供游戏执行的机器码，简直是自找麻烦，这会带来很多安全问题。我们只能在机器码的效率和解释器模式的安全性之间取一个折中。

这就是为什么很多主机和iOS系统禁止程序在运行时生成或载入机器码的原因。这反倒是个拖累，因为最快的编程语言就是基于这个原理实现的。它们包含一个准时("just-in-time")编译器，或者叫JIT。它能飞快地把语言翻译成优化的机器码。

我们不要去加载执行真正的机器码，而去定义自己的虚拟机器码，会怎样呢？我们在游戏中实现一个执行它们的模拟器。这些虚拟机器码与机器码相似——高密度、线性、相对底层——同时它完全接受游戏安全的管理。

在编程语言的语境下，“虚拟机”和“解释器”是同义词，我正是交替地使用它们。如果说Gof的解释器模式的话，我会强调“模式”这个词，以避免混淆。

我们将这个小模拟器称为虚拟机（简称VM），这个虚拟机所执行的语义上的“二进制机器码”称为字节码。它具备从数据定义事物的灵活性和易用性，它也比解释器模式这种高级呈现方式更高效。

听上去挺吓人的。我在本章里剩下的目标，就是要给你展示一下，如果你控制好自己的功能清单，这个方案非常可行。即使最终你自己也没把这个模式用起来，至少你能对Lua以及别的采用这个原理的语言有更好的了解。

## (解释器)模式

指令集定义可以执行的底层操作。一系列指令被编码为字节序列。虚拟机逐条执行指令栈上的指令。通过组合指令，即可完成很多高级行为。

## 使用情境

这是本书中最复杂的模式，它可是不是轻易就能放进你的游戏里。仅当你的游戏中需要定义大量行为，并实现游戏的语言没法处理好下列事情时可以使用：

- 编程语言太底层了，编写起来繁琐易错
- 因编译时间太长或工具问题，导致迭代缓慢
- 它的安全性太依赖编译者。你想确保定义的行为不会让程序崩溃，就得把它们放进安全沙箱里。当然，这个列表符合大多数游戏的情况。谁不想提高迭代速度，让程序更安全？但那是有代价的。字节码比本地码要慢，所以它并不适合用作对性能要求极高的核心部分。

## 注意

建立你自己的语言或内嵌系统是一件很有吸引力的事。这里我只做个最小化示例，但在实际项目中，麻烦可多多了。

这也正是游戏开发吸引我的地方，二者不论是哪个，我都在努力创建虚拟世界，让别人进来玩或做创意。

每当我看到有人创造出一种小语言或脚本，他们会说“别担心，它会很小巧”。没法控制的是，他们会不断往里面添加小功能，直到它变成一个成熟的语言。但不像其他语言，它的发展是一些临时功能的有机组合，就像个精致的棚屋小镇。

任何一种的模板语言都是这样

当然，做个成熟的语言没什么错。只要保证你目标明确。否则，就控制好你的字节码要表达的东西，在它超出你控制之前设定好范围。

## 你需要个前端

低级的字节码对性能提升很大，但你没法让你的用户直接编写它们。我们将行为从代码中移出来的一个原因是想在更高一级

的层面表述它。C++已经很底层了，如果让你的用户用更高效的汇编语言编写——这根本不是种进步！

一个反例是有名的RoboWar。在这个游戏里，玩家使用一种类似汇编的语言编写小程序，来控制机器人。我们这里也会讨论指令集这种方式。这是我的第一篇汇编类语言的指南。

就像Gof的解释器模式一样，它假定你能够以某种方式生成字节码。通常，用户会在更高级的层次上编辑，一个工具负责将它转换成虚拟机能够理解的字节码。这个工具的名字，就是编译器。

我知道，听上去很可怕对不对。所以这里我先提出来了。如果你没有足够的资源去完成一个编辑工具，那么字节码不适合你。但你先别急继续往下看，也许没你想象中那么坏。

## 你会想念调试器的

编程很难。我们知道想让机器做什么，但是我们很难用正确的方式与之沟通——我们会写出bug。为此，我们搜集了一大堆工具来找出代码错在哪里，如何去改正。我们有调试器、静态分析器、反编译工具等等。所有这些工具都是为某种已经存在的语言而设计的：机器码或者是高级语言。

当你定义自己的字节码虚拟机时，你就没法用这些工具了。当然了，你可以用调试器步进到虚拟机的代码里，但那只能告诉你虚拟机在做什么，与它正在解释的字节码没什么关系。它也没法替你把字节码映射回对应的原始高级语言。

如果你定义的行为很简单，你可以勉强回避掉做各种辅助调试工具的事儿。但是随着内容规模增长，你得规划好如何让用户能实时看到他们的字节码有什么效果。这些功能可能不会随游戏发布，但是它们能确保你的游戏可以发布。

当然，如果你想让游戏支持MOD，你就得发布这些功能，这相当重要。

## 示例

在上面几节讨结束之后，你可能会很好奇如何直接实现它。首先，要为虚拟机设计一个指令集。在真正考虑字节码之类的东西前，可以先把它们当成是API。

### 法术API

假设我们要直接用C++代码去实现各种法术，我们需要让代码调用哪些API呢？为了定义法术，引擎中要定义哪些基础操作呢？

绝大多数法术会改变巫师的一个状态，我们就从这里开始：

```
void setHealth(int wizard, int amount);
void setWisdom(int wizard, int amount);
void setAgility(int wizard, int amount);
```

第一个参数定义受到影响的巫师，比如说用0代表玩家，用1代表对手。这样以来，治疗法术就能够施加到玩家自己的巫师身上，同时也可以伤害到对手。毋庸置疑，这三个小函数会神奇得支持非常广泛的法术效果。

然而如果法术只是静默得改变状态，游戏逻辑上不会有太大问题，但是玩这样的游戏会让玩家无聊到哭的。我们来做些调整：

```
void playSound(int soundId);
void spawnParticles(int particleType);
```

这些不会影响到玩法，但是会增加游戏的深度。我们还会添加摄像机抖动、动画等等。但是这些就足够我们开始了。

### 法术指令集

现在让我们看看如何将这些程序API转换成数据可控的形式。让我们由简入繁来完成整件事。首先拿掉这些函数中所有的参数。假设所有的set\_()函数都会影响玩家控制的法师并强化其对应属性。类似的，FX操作们会播放一个硬编码的音效或者粒

子特效。

在这个前提之下，法术就是一系列的指令。每个指令定义一个你想要执行的操作。我们可以枚举他们：

```
enum Instruction {
    INST_SET_HEALTH = 0x00,
    INST_SET_WISDOM = 0x01,
    INST_SET_AGILITY = 0x02,
    INST_PLAY_SOUND = 0x03,
    INST_SPAWN_PARTICLES = 0x04
};
```

为了将法术编码成数据，我们存储一系列枚举值在数组中。我们的仅有几种基本操作，所以枚举值长度取一个字节足矣，这意味着法术的代码都是一个字节列表——所谓的字节码。

一些字节码虚拟机使用多个字节去存储单个指令，这需要有更加复杂的解码规则。实际上常见芯片上的机器码，比如x86，就更加复杂了。

但是单字节对于Java Virtual Machine以及Microsoft .NET平台的基石Common Language Runtime来说已经够用了，所以对我们来说已经可以了。

执行一条指令时，我们首先找到对应的基础方法，然后调用正确的API：

```
class VM {
public:
    void interpret(char bytecode[], int size) {
        for (int i = 0; i < size; i++) {
            char instruction = bytecode[i];
            switch (instruction) {
                // Cases for each instruction...
            }
        }
    }
};
```

把这段代码写进去，你就完成了你的第一个虚拟机。可惜它还不够灵活。我们没办法去定义一个能够伤害到对手或者削弱某个属性的法术。我们只能播放段声音而已。

为了多一点真正语言的感觉，我们这里需要加入参数。

## 栈机

要执行一个复杂的嵌套表达式，你从最内层的子表达式开始。计算完的内层表达式的结果，就将结果作为包含它们的外层表达式的参数传给外层表达式进行计算，以此类推直到整个表达式计算完毕。

解释器模式将这一过程显式建模成一棵嵌套对象树，但我们想要获得像指令列表一样的高速度。同时要保证自表达式的结果能够正确的传入外层表达式。但由于我们的数据是被展平的，我们得通过指令的顺序去控制。我们会采用与你的CPU相同的方式——一个堆栈。

理所当然，这个架构就称为栈机。例如Forth、PostScript和Factor这类编程语言将这个模型直接暴露给了用户。

```
class VM {
public:
    VM() : stackSize_(0) {} // Other stuff... private:
    static const int MAX_STACK = 128;
    int stackSize_;
    int stack_[MAX_STACK];
};
```

这个虚拟机内部包含了一个值堆栈。在我们的例子中，与指令相关的唯一数据类型是数字，所以我们可以使用一个int型数

组。当一段数据要求指令一个一个执行下去时，实际上就是在遍历堆栈。

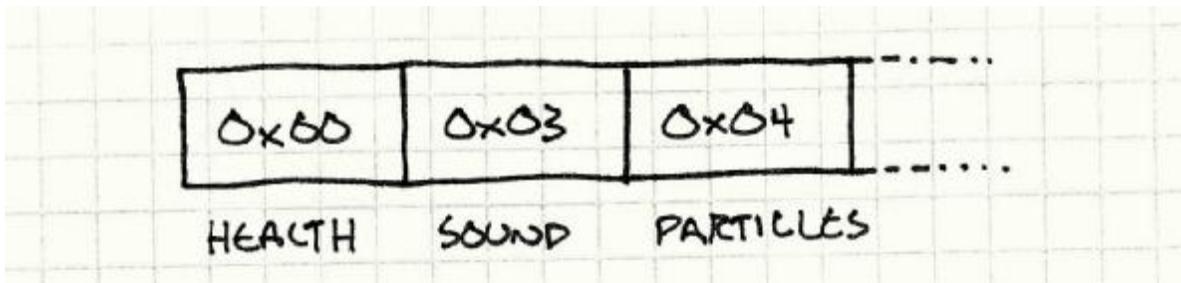
字面意思，数值可以被压入或者弹出这个堆栈。因此，让我们添加些方法来实现这个功能：

```
class VM {
    private:
    void push(int value) {
        // Check for stack overflow.
        assert(stackSize_ < MAX_STACK);
        stack_[stackSize_++] = value;
    }
    int pop() {
        // Make sure the stack isn't empty.
        assert(stackSize_ > 0);
        return stack_[--stackSize_];
    }
    // Other stuff...
};
```

当哪一个指令需要输入参数时，它会按照下面的方式从堆栈中弹出来：

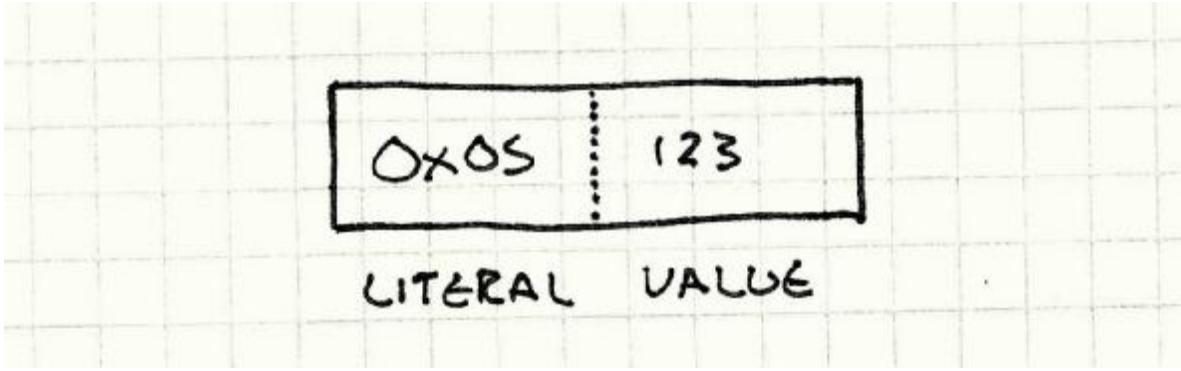
```
switch (instruction) {
    case INST_SET_HEALTH: {
        int amount = pop();
        int wizard = pop();
        setHealth(wizard, amount);
        break;
    }
    case INST_SET_WISDOM:
    case INST_SET_AGILITY:
        // Same as above...
    case INST_PLAY_SOUND:
        playSound(pop());
        break;
    case INST_SPAWN_PARTICLES:
        spawnParticles(pop());
        break;
}
```

为了向堆栈中添加一些数值，我们需要一个新的指令：字面值。它表示一个字面上的整数数值。但是它又从哪里获得这个值呢？这里究竟该如何避免无限循环呢？



这个小技巧就是利用指令流是字节序列的特性——我们可以将数字直接塞进字节数组。我们用如下方式定义一个字面数字的指令类型：

```
case INST_LITERAL: {
    // Read the next byte from the bytecode.
    int value = bytecode[++i];
    push(value);
    break;
}
```

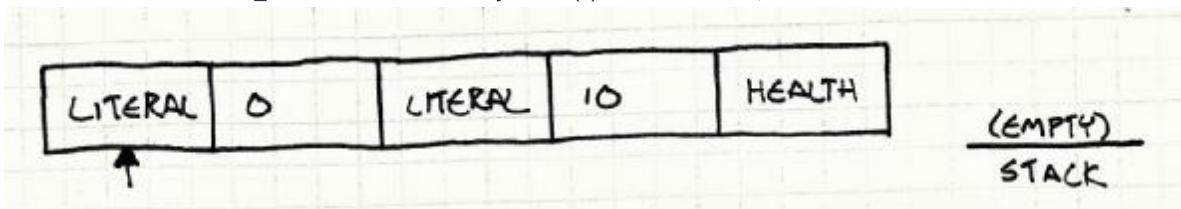


这里，为了避开处理多字节整型的情况，我仅读取单字节整数，但是在实际实现中，你肯定想要支持所有你所需范围的整数参数。

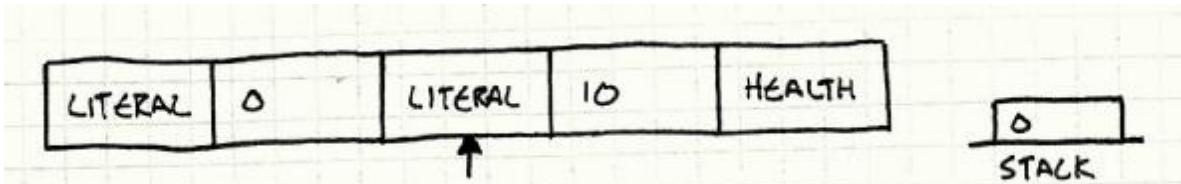
它读取了字节码流中的下一个字节，将它当作一个数字写入堆栈。

为了能够对堆栈的工作方式有个直观感受，我们把几条指令串起来，看看它们如何被解释器执行。从一个空栈开始，解释器指向第一个指令：

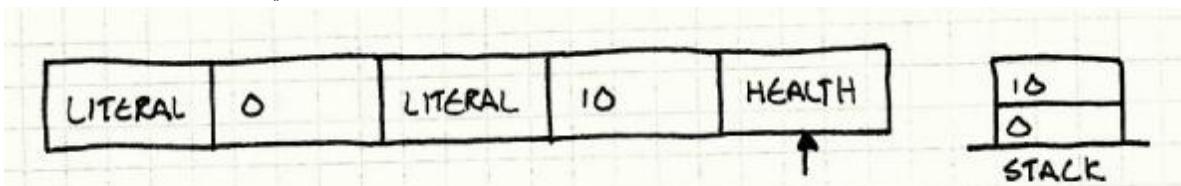
首先，它执行第一个 INST\_LITERAL。他会读取从 bytecode(0) 开始的下一个字节，并将它压入堆栈。



然后，它执行第二个 INST\_LITERAL。它读取数字 10，并将其压入堆栈。



最后，它执行 INST\_SET\_HEALTH。它会出栈 10 并将其存储到变量 amount 中，然后出栈 0 将其存储到 wizard 中。之后，使用这两个参数调用 setHealth()。



嗒啦！我们完成了一个将玩家巫师的生命值设定为 10 点的法术。现在，我们就拥有了足够的灵活性，来把任何巫师的状态设定到任何想要的值。我们也可以播放不同的音效以及发粒子。

但是，这感觉更像是数据结构。我们没法做到诸如将巫师的生命提高其法力值的一半。我们的设计师想要制定法术的计算规则，而不仅仅是数值。

## 组合就能得到行为

如果将我们的虚拟机看做是一种编程语言，它所支持的仅仅是些内置函数，以及它们的常量参数。为了让字节码感觉更像是行为，我们得加上组合。

我们的设计师想要创建一些表达式，能够将不同的值通过有趣的方式组合起来。举个简单的例子，他们想让一个法术对某种属性造成一个相对量的变化，而不是改变到一个绝对的量。

那就需要考虑状态的当前值。我们已经有了写入状态的指令，但还得加上些读取它们的指令：

```

case INST_GET_HEALTH: {
    int wizard = pop();
    push(getHealth(wizard));
    break;
}
case INST_GET_WISDOM:
case INST_GET_AGILITY:
// You get the idea...

```

显然，它对堆栈做了双向操作。它首先出栈一个参数，来确定要获取哪个巫师的状态，然后找到这个状态值并入栈。

这使得我们能够编写任意拷贝状态值的法术。我们能够创造一个将巫师的敏捷设定为其智力甚至是复制对手生命值的古怪巫术。

好了一点，但是仍然很有限。接下来，我们需要算术。是时候让我们牙牙学语的虚拟机学 $1+1$ 了。我们得添加些新的指令。到现在为止，你应该已经发现它的规律并能够猜到它会是怎样的了。下面是加法：

```

case INST_ADD: {
    int b = pop();
    int a = pop();
    push(a + b);
    break;
}

```

和其他的指令一样，它出栈了一些数值，做了一些处理，然后将结果入栈。到现在为止，每个指令都提高了一点儿我们对表达式的支持，但这是个很大的跨越。它看起来不起眼，但我们能够处理各种复杂的，深层嵌套算术表达式。

让我们看看一个稍微复杂点的例子。比如说，要制作一个法术，能够将玩家法师的生命设定成他们敏捷和智力的平均值。在代码里面，是这样的：

```
simplyyetHealth(0, getHealth(0) + (getAgility(0) + getWisdom(0)) / 2);
```

你可能会认为我们需要指令来控制这个表达式里面由于括号造成的显式分组。但堆栈已经隐式支持它了。下面是手工求值的方法：

获取并保存法师当前的生命值。获取并保存法师的当前敏捷度。对智慧做同样的操作。获取后两个值，将他们相加并保留结果。除以2后保留结果。取回巫师的生命并加到结果里面去。获取结果，并赋值到巫师的生命属性。

你看到那些“保留”和“取回”了吗？每个“保留”对应于一个push，每个“取回”对应于一个pop。这意味着我们可以轻易将其转换为字节码。例如，第一行获取巫师的当前生命值：

```
LITERAL 0
GET_HEALTH
```

这段字节码将巫师的生命值入栈。如果我们重复这样的操作，最终会得到一段能计算出原表达式的字节码。为了让你体会指令是怎样组合的，我已经帮你做好了。

为了演示堆栈如何随时间变化，权且将巫师的初始状态设置为45点生命、7点敏捷和11点智力。跟在每个指令后面的是执行后的堆栈状态，以及这个指令作用的注释：

```

LITERAL 0 [0] # Wizard index
LITERAL 0 [0, 0] # Wizard index
GET_HEALTH [0, 45] # getHealth()
LITERAL 0 [0, 45, 0] # Wizard index
GET_AGILITY [0, 45, 7] # getAgility()
LITERAL 0 [0, 45, 7, 0] # Wizard index
GET_WISDOM [0, 45, 7, 11] # getWisdom()
ADD [0, 45, 18] # Add agility and wisdom
LITERAL 2 [0, 45, 18, 2] # Divide by two

```

```
DIVIDE [0, 45, 9] # Average agi li ty and wi sdom  
ADD [0, 54] # Add average to current health  
SET_HEALTH [] # Set health to result
```

如果你一步一步看完这个堆栈，你就会发现数据像魔法一样在它内部流动。我们在一开始入栈巫师的索引0，然后做了很多不同的操作，直到最后栈低设置巫师生命值时用到它。

也许我这里对“魔法”的定义有点宽。

## 一个虚拟机

我可以继续前进，添加更多各种各样的指令，但这是个停下来的好地方。像它现在这样，我们有了一个不错的小虚拟机，好让我们能使用简单又可压缩的数据根式来定义相对可扩展的指令。虽然“字节码”和“虚拟机”听起来有点吓人，但你会发现它们往往简单到一个堆栈、一个循环或是一个switch语句。

还记得我们最初目标是让字节码得到很好的沙箱化？现在你看过虚拟机的整个实现过程，很明显我们已经做到了。字节码没法深入引擎的各个部分做有恶意的事情，因为我们只定义了少量访问引擎局部的指令。

我们通过控制堆栈尺寸来限制它的可用内存，我们要当心以免内存溢出。我们甚至可以限制它的执行时间。在指令循环中，我们可以记录已经执行了多久，在它超出某个时间限制时，将它取消掉。

限制执行时间在我们的例子中并非必要，因为我们没有任何循环指令。我们可以通过限制字节码的总尺寸来限制执行时间。这也意味着字节码并非图灵完备。

只剩下一个问题了：真正去创建字节码。眼下我们将一段伪代码编译成了字节码。除非你真的很闲，否则这在实践中根本行不通。

## 语法转换工具

我们的一个最初目标是在较高的层次上编写行为，但是我们已经做了些比C++还底层的东西。它能兼顾我们需要的运行时性能和安全性，但是彻底缺乏对设计师友好的可用性。

为了填补这个缺陷，我们需要些工具。我们需要一个程序，让用户在高层次上定义法术的行为，并能够生成对应的低层次字节码。

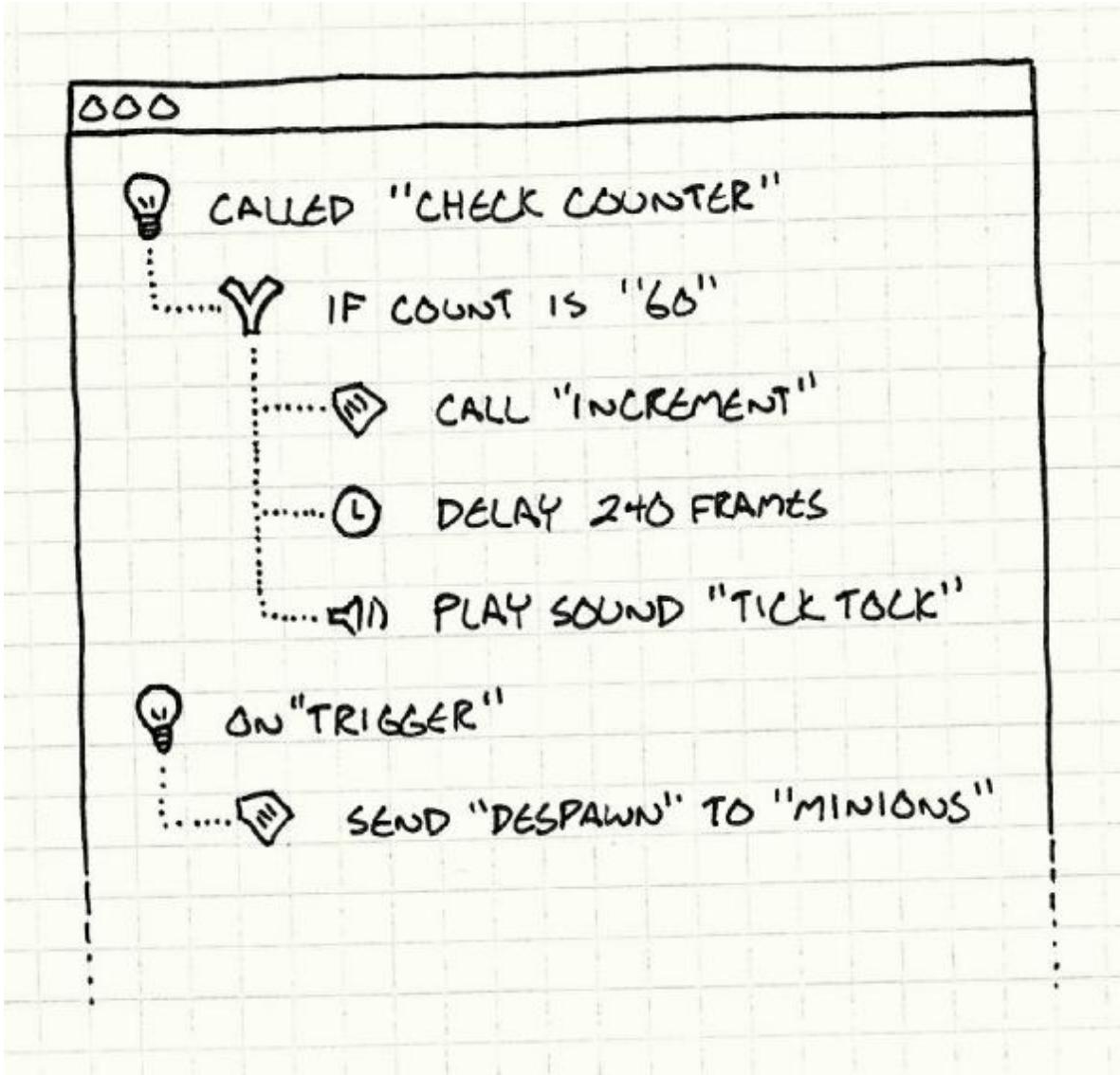
这听起来比创建一个虚拟机还难。很多程序员在大学的时候被扔到一个编译器课程中，其中所得除了看到封面上有条龙或者“lex”和“yacc”这些词时就犯的创伤后应激障碍之外，什么也没有。

我所说的，当然是这篇经典的[编译器：原则、技术和工具](#)

其实，编译一个基于文本的语言并非不能，只是这里篇幅有限。但是，你没必要这么做。我说你需要一个工具——并不一定得是个能编译输入文本的编译器。

恰恰相反，我希望你考虑做一个图形界面来让用户定义行为，他别是对于那些不太擅长技术的人。对于一个没有多年面对编译器各种报错的经验的人来说，书写语法正确的文本太难了。

反之，你可以创建一个应用，让用户通过点击和拖拽一些小方块、点选菜单或者其他任何对创建行为有意义的事情。



我为 [Henry Hatsworth in the Puzzling Adventure](#) 编写的脚本系统的原理就是这样的。

这么做的好处是你的UI让用户几乎难以创建“非法的”程序。你可以前瞻性地禁用按钮或者提供默认值来保证他们创建的东西在任何时候都是合法的，而不是丢出一大堆错误信息。

我要强调下错误处理的重要性。作为程序员，我们倾向于把人为错误看做是耻辱的人性缺陷而竭尽全力避免发生在自己身上。

为了做出一个用户喜欢的系统，你得拥抱他们的人性，这就包括了不可靠性。人们总是犯错，它是创造活动的组成部分。通过一些撤销之类的功能来优雅地处理掉这些问题能让你的用户更有创造力并更好地完成任务。

这让你免于为一个小语言设计语法并编写语法分析器。但我也清楚，有些人对UI编程同样很不习惯。那么，这里我也没别的办法。

最终，这个模式还是关于如何用一种用户友好并能在高层次编辑的条件下表达行为的。你得去创建用户表达式。为了获得高执行效率，你得把它翻译成低级形式。这就是真正的工作，但如果你接受这个挑战，它会给你回报的。

## 设计决定

我试图让这一章尽可能简单，但是我们真的是在创建一种语言。它是一个很开放的设计空间。在其中尝试非常有趣，所以，别忘了完成你的游戏。

因为这是本书中最长的一章，这个任务我失败了。

## 指令如何访问堆栈？

字节码虚拟机有两种大风格：基于栈和基于寄存器。在基于栈的虚拟机中，指令总是操作栈顶，正如我们的示例代码一样。例如，INST\_ADD出栈两个值，将它们相加，然后将结果入栈。

基于寄存器的虚拟机也有一个堆栈。唯一的区别是指令可以从栈的更深层次中读取输入。不像INST\_ADD那样总是出栈操作数，它在字节码中存储两个索引来表示应该从堆栈的那个位置读取操作数。

- 基于栈的虚拟机：

- 指令很小。因为每个指令都从隐式从栈顶寻找它的参数，你无需对任何数据做编码。这意味着每个指令都非常小，通常只用一个字节。
- 代码生成更简单。当你要编写一个编译器或生成字节码输出的工具时，你会发现机遇栈的虚拟机更简单。每个指令都隐式操作栈顶，你只需要以正确的顺序输出指令，来实现参数传递。
- 指令数更多。每个指令都只操作栈顶。这意味着生成类似 $a = b + c$ 这样的代码。你就得用分离指令把b和c分别放到栈顶，执行操作，然后将结果存入a。

- 基于寄存器的虚拟机：

- 指令更大。因为它需要记录参数在栈中的偏移量，单个指令需要更多的位数。例如，众所周知的寄存器虚拟机Lua中，每个指令占用32位。6位存储指令类型，剩下的存储参数。
- 指令更少。因为每个指令都能做更多的事情，相应的就没那么多。因为你无需把堆栈中的值拖来拖去，也可以说你获得了性能提升。

Lua的开发者并未明确Lua的字节码格式，它每个版本都在变化。我这里讲的是Lua5.1。想要看一篇精彩的Lua内部剖析，读读[这个](#)

那么你应该怎么选呢？我的建议是实现基于栈的虚拟机。它们更容易实现，生成代码也更加简单。寄存器虚拟机因Lua转换为它的格式之后执行效率更高而受到称赞，但这实际上依赖于你的虚拟机的实际指令集设计和其他很多东西。

## 应该有哪些指令？

你的指令集划定能用字节码表达与不能用字节码表达的边界，它也对虚拟机的性能有影响。

- 外部基本操作。它们是虚拟机之外、引擎内部的，做一些玩家能看到的事情的东西。它们决定字节码能够表达的真正行为。如果没有它们，你的虚拟机除了在循环中烧CPU之外，没有任何用处。
- 内部基本操作。它们操作虚拟机内部的值——例如字面值、算术运算符、比较运算符和操作栈的指令。
- 流程控制。我们的例子中没有这部分，但如果你想要让指令有选择地执行或是循环重复执行，那你就需要流程控制。在字节码这样的底层语言中，它们及其简单——跳转。在我们的指令循环中，我们有一个索引指向字节码堆栈的当前位置。换句话说，它是个goto。你可以用它来实现任何高级流程。
- 抽象。如果你的用户开始定义很多数据，最终他们会希望能重用字节码而不是反复复制粘贴。你也许会用到可调用过程。最简情况下，过程仅仅是一个跳转。唯一的不同是虚拟机维护另一个返回堆栈。当它执行到一个“call”指令时，它将当前指令压入返回栈中然后跳转到被调用的字节码。当它遇到一个“return”时，虚拟机从返回栈中弹出索引并跳转回索引的位置。

## 值应当如何表示？

我们的实例虚拟机只助理一种值，整数。这让答案变得很简单——这个堆栈仅仅是个存放int的栈。一个功能完善的虚拟机应当支持不同的数据类型：字符串、对象、列表等等。你必须决定如何在内部存储它们。

- 单一数据类型：
  - 它很简单。你不用担心标签、转换或者类型检查。
  - 你无法使用不同的数据类型。这个缺陷太明显了。将不同的类型填入到一种单一的呈现方式中——例如将数字存储成字符串——这是自找麻烦。
- 标签的一个变体：这是动态类型语言通用的形式。每个值都由两部分组成。第一部分是一个标签——一个枚举——用来标志所存储数据的类型。剩下的位根据这个类型来解析，例如：

```

enum ValueType
{
    TYPE_INT,
    TYPE_DOUBLE,
    TYPE_STRING
};

struct Value
{
    ValueType type;
    union
    {
        int intValue;
        double doubleValue;
        char* stringValue;
    };
};

```

- 值知道他们的类型。这种呈现方式的好处是，能够在运行时对值的类型做检查。这对动态调用很重要并能够保证你不会把操作执行到不支持它们的类型上。
- 占用内存更多。每一个值必须携带标志它们类型的额外位。在虚拟机这样的底层中，这几个位的占用增长得很快。
- 不带标签的联合体：与前一种方式类似，但使用联合体，它不会在每个值上携带一个累型标签。你有一个小数据块去表示多种类型，你需要自行确保值能得到正确解析，你不需要在运行时检查类型。

这也是无类型语言比如汇编和Forth的储值方式。这些语言让用户自己保证解析值的方式是正确的。玻璃心伤不起！

- 紧密。没有比只存储值本身更加高效的储值方式了。
- 快速。没有类型标签意味着你也无需再运行时检查它们。这也是静态类型语言比动态类型语言快的原因。
- 不安全。当然，这是真正的代价。一段错误的字节码，让你把一个数字当做指针或者反过来，都会违反游戏的安全性而造成崩溃。

如果你的字节码是从静态类型语言编译而来的，你可能会因编辑器不会生成不安全字节码而认为它是安全的。这也许是正确的，但是不要忘了用户可能绕过你的编译器去手工编写一些恶意的字节码。这就是Java虚拟机等要在加载程序时执行字节码检查的原因。

- 一个接口：确定值是一些类型中的一种的面向对象的解决方案是多态。一个接口提供进行各种类型测试和转换的虚方法，像下面这样：

```

class Value
{
public:
    virtual ~Value() {}
    virtual ValueType type() = 0;
    virtual int asInt() {
        // Can only call this on ints.
        assert(false);
        return 0;
    }
    // Other conversion methods...
};

```

你可能像下面这样定义数据的类：

```

class IntValue : public Value
{
public:
    IntValue(int value): value_(value)
    {}
    virtual ValueType type() { return TYPE_INT; }
    virtual int asInt() { return value_; }
private:
    int value_;
};

```

- 开放式。你可以再核心虚拟机之外定义任何实现基础接口的数据类型。
- 面向对象。如果你采用面向对象的准则，正确的做法是对类型采取多态性调度，而不是对类型标签是用switch。

- 累赘。你得为每一个数据类型定义一个类，并在里面填写一些重复性的内容。在前一个例子中，我们定义了所有的类型，这个例子里才只定义了一个！
- 低效。为了实现多态，你得借助于指针。这意味着像布尔和数字这种微小的值也要被封装到对象中，并在堆上面分配。每次访问一个值，你都是在做虚函数调用。在虚拟机核心中，这样影响效率的点会不断累加。事实上正因为这些问题，导致我们避免解释器模式，唯一的区别是解释器处理的是代码而我们处理的是值。

我的建议是，如果你能坚持使用单一数据类型，那就这么做。否则，使用标签联合体。这是几乎所有编程语言的解析的方式。

## 如何生成字节码？

我把最重要的问题留到了最后。我带你消化并分析了字节码，但是轮到你做些东西来生成它们了。标准的解决方案是编写一个编译器，但这并不是唯一的途径。

- 如果你定义了一种基于文本的语言：
  - 你得定义一种语法。无论业余或专业的设计师都容易想当然得低估这件事的难度。定义一种对分析器友好的语法很容易，但是定义一种对用户友好的很难。语法设计也是种用户界面设计，及时用户界面变成了一串字符，也容易不到哪儿去。
  - 你要实现一个分析器。不管它们的名声怎么样，这部分很简单。你可以使用ANTLR或Bison这样的解析器生成器，或者——跟我一样——自己写一个好用的递归分析，这样就行了。
  - 你必须处理语法错误。这是整个过程中最重要也是最难的部分。当用户出现语法或语义错误的时候——他们当然会，而且一直出错——将它们领回到正确的道路上是你的事情。当你都不知道解释器处于一个意外符号上时，提供帮助性的反馈不容易。
  - 对非技术人员没有亲和力。程序员喜欢文本文件。配合强大的命令行工具，我们将它们当做计算机里的乐高块——简单，却有无数种组合方式。多数程序员并不这样看待纯文本。对他们来说，文本文件如同给一个机器稽核员填写的纳税表，即使少填一个分好，它也会朝你大叫。
- 如果你定义一个图形化编辑工具：
  - 你要实现一个用户界面。按钮、点击、拖拽等诸如此类。这个方法感觉有点低三下四，但是我个人很喜欢它。如果你选择这个方向，设计好用户界面就是做好这件事情的关键——不是一件能应付了事的无聊事。这里你做的没一点儿额外工作都会使得工具更加易用而友好，这会直接提高你游戏内容的质量。如果你回头看看很多你喜欢的游戏，你常常会发现它们的秘密是有一个又去的编辑工具。
  - 不易出错。因为用户一步步交互式得构建行为，你的程序能够在发现错误时立刻引导他们改正。使用文本语言时，工具只有在提交整个文件时才能看到用户内容。这使得避免和控制错误都变得困难。
  - 可移植性差。文本编译器的一点好处是它是通用的。一个简单的编译器仅仅读取一个文件并输出另一个文件。在操作系统间移植是很容易的。

除了换行符和编码。

当你制作UI时，你得选择使用什么框架，很多框架都依赖于一种操作系统。也有一些跨平台的UI工具包，但是他们的代价在于亲切感——他们在所有的平台上都让人感到陌生。

## 参考

- 这个模式是四人帮[解释器模式](#)的姐妹版。它们都会给你一种用数据来组合行为的方法。事实上，你经常会两个模式一起使用。你用来生成字节码的工具通常会有一个内部对象树来表达代码。这正是解释器模式能做的事情。为了将它编译成字节码，你需要递归遍历整棵树，正如你在解释器模式中解析它那样。唯一的不同是你并不是直接执行一段代码而是将它们输出成字节码指令并在以后执行它们。
- [Lua](#)编程语言是游戏中广泛使用的编程语言。它内部实现了一个紧凑的基于寄存器的字节码虚拟机。
- [Kismet](#)是内置在UnrealEd（Unreal Engine 的编辑器）中的图形化脚本工具。
- 我自己的小脚本语言，[Wren](#)，是一个简单的基于堆栈的字节码解释器。

===== [上一节](#)

[目录](#)

[下一节](#)

# 子类沙盒

## 目的

使用一个基类提供的操作集合进而在子类中定义行为

## 动机

每个小孩都有一个成为超级英雄的梦想，但是很不幸，宇宙射线在地球上供应不足。游戏或许是令你成为超级英雄的最佳之地。因为我们的游戏设计师从来不会说，“不”，我们的超级英雄游戏意在提供至少十种或百种不同的能力以供玩家选择。

我们的计划是将有一个Superpower基类，然后，我们将有一个实现各个超级力量的继承类。我们将把设计文档分摊给团队中的程序员并进行编码。当我们完成的时候，我们将有数以百计的超级力量的类。

### 注解

当你发现自己像这个例子一样有大量的子类的时候，这意味着一种数据驱动的方法可能更适合。试着找到一种定义数据的行为的方法，而不是用大量的代码来定义不同的力量。

像模式[Type Object](#), [Bytecode](#) 和 [Interpreter](#) 或许能有所帮助。

我们想让玩家沉寘在一个复杂多变的世界里。无论他们小时候梦想过的什么力量，在我们的游戏里都有。这就意味着这些超级力量子类能够几乎做任何事情：播放音效，产生视觉效果，与AI交互，创建和销毁其他游戏实体以及产生物理效果。它们将涉及代码库的绝大部分内容。

让我们的团队就这么放手开始写这些子类，会发生什么呢？

- 会充满大量的冗余代码。尽管不同的力量将有所不同，我们也能料到其中必有不少冗余。他们中的多数将以同样的方式来产生视觉效果和播放音效。当你完成冰冻射线，热射线，第戎芥末射线这些射线时，会发现它们在实现上极其相似。如果人们在实现它们时没有整合起来，那么将会有大量重复的代码和付出。
- 游戏引擎的每个部分将与这些类产生耦合。在未深入了解之前，人们所写的代码会调用到那些可能与超级力量类毫无绑定关系的系统。如果我们的渲染器被组织成一些漂亮优雅的分层，只有其中的一层能够被图形引擎之外的代码使用，我们可以打赌最后将留下侵入到他们所有层的超级力量代码。
- 当这些外部系统需要改变的时候，超级力量代码将很可能被随机性地破坏。一旦我们的各种超级力量类与游戏引擎的各个零散部分产生耦合，改变这些系统无疑将影响这些超级力量类。这可不好玩，因为你的图形，音效，UI程序员可不想同时做游戏程序员的工作。
- 定义所有超级力量都遵守的约束条件很困难。比如说我们想保证所有我们超级力量播放的音效得到合理的排队和优先级处理。如果我们的百来个类都自己直接地调用音效引擎的话，这将很难实现。

我们需要的是给每个实现一个超级力量的游戏程序员一系列可用的基本元。你想要你的力量播放音效吗？那就提供给你 `playSound()` 函数。想要粒子效果吗？这里有 `spawnParticles()`。我们将保证这些操作覆盖你所有的需求，这样一来你就不必滥用 `#include` 来包含某些力量类，也不必去探究代码基的余下部分。

我们通过把这些操作设置成 Superpower 基类的保护方法来实现。把它们放在基类就能让每个力量子类直接简单地访问这些方法。把它们设置为保护状态(并且可能是非虚拟的)来交互，使得它们仅仅作为子类可调用的方法而存在。

我们已经有了玩偶，现在是时候把它们置入游戏中了。为此我们定义一个沙盒方法，这是一个子类必须实现的抽象保护方法。在有了这些之后，为了实现一种新的力量，你要做的就是：

- 创建一个继承自 Superpower 的新类。
- 覆盖沙盒函数 `activate()`。

### 3. 通过调用 Superpower 提供的保护函数来实现新类方法的函数体。

我们通过尽可能地提高将可用操作的层面来解决代码冗余的问题。当我们发现在大量子类中存在重复代码，我们可以把它向上移到 Superpower 中作为一个可用的新操作。

我们已经通过把耦合限制在一处来集中耦合问题。Superpower 最终将与不同的游戏系统耦合，但我们的上百个子类不会，它们仅与基类耦合。当这些游戏系统中的一个变化时，对 Superpower 进行修改可能是必须的，但是这些大量的子类不应被改动。

这个设计模式会催生一种浅而宽的类层次架构。你的继承链不会深，但是会有大量的类挂在 Superpower 上。通过生成一个有大量直接子类的单个类，我们在代码基里就有一只单点杠杆。我们在 Superpower 中所付出的心血都将对游戏中大量的类带来益处。

#### 注解

近来，你发现人们对面向对象语言的继承进行批判。继承是有问题的 -- 在代码基中没有比基类与子类之间更深的耦合了 -- 但是我发现宽的继承树比深的要表现的更好。

## (沙盒)模式

一个基类定义了一个抽象的沙盒方法和一些提供的操作。通过设置他们为保护状态来保证它们仅供子类使用。每个继承的沙盒子类针对父类提供的操作来实现沙盒函数。

## 使用情境

沙盒模式是运用在多数代码库里甚至游戏之外的一种非常简单通用的模式。如果你有一个非虚拟的保护函数，那么你很有可能正在使用与之相类似的模式。沙盒模式在以下情况比较适用：

- 你有一个带有大量子类的基类。
- 基类能够提供所有子类可能需要执行的操作集合。
- 在子类之间有重叠的代码，你想让它们之间更容易地共享代码。
- 你想使这些继承类与程序的其他代码之间的耦合最小化。

## 使用须知

“继承”一词在近代的一些程序圈子里被诟病，其中一个原因是基类会滋生越来越多的代码。这个模式尤其受这个因素的影响。

由于子类是通过它们的基类来实现剩下的游戏，基类最终会与那些需要与其子类交互的系统产生耦合。当然，这些子类也与他们的基类绑定。这个蜘蛛网式的耦合使得无损地改变基类是很困难的 -- 你遇到了[脆弱的基类问题](#)。

而从好的角度来说，你所有的耦合都被聚集到了基类，子类现在明显地与其他世界更加独立了。理想状态下，你的绝大部分操作都在子类中。这意味着你的大量的代码库是独立的，并且更易于维护。

如果你仍然发现这个模式正把你的基类浸入一大锅代码种时，请考虑把一些提供的操作提取到一个基类能够管理的独立的类中。这里[组件模式](#)能够有所帮助。

## 示例

由于这是一个如此简单的设计模式，并没有多少的示例代码。这不意味着它没有用 -- 这个模式的实现关乎的是其意义而不是其复杂度。

我们将从我们的 Superpower 基类开始：

```
class Superpower
{
public:
    virtual ~Superpower() {}

protected:
    virtual void activate() = 0;

    void move(double x, double y, double z)
    {
        // Code here...
    }

    void playSound(SoundId sound, double volume)
    {
        // Code here...
    }

    void spawnParticles(ParticleType type, int count)
    {
        // Code here...
    }
};
```

函数 `activate()` 就是沙盒函数。由于它是虚拟和抽象的，子类必须要覆盖它。这是为了让子类使用者能够明确他们该对自己的特殊力量子类做些什么。

另外的保护函数 `move()`，`playSound()` 和 `spawnParticles()` 都是提供的操作。这些就是子类需要在 `activate()` 函数实现时将调用的函数。

我们没有在这个示例中实现提供的操作，但是一个实际的游戏需要有真实的代码在那儿。这个函数是 `Superpower` 在游戏中与其他系统耦合的地方 -- `move()` 函数也许会调用物理引擎代码，`playSound()` 将与音效引擎通讯等等。由于所有的这些都是在基类的实现中，这就使得所有的耦合都封装在 `superpower` 自己中。

好啦，现在让我们放出放射性蜘蛛并创建一个力量。这就有一个：

```
class SkyLaunch : public Superpower
{
protected:
    virtual void activate()
    {
        // Spring into the air.
        playSound(SOUND_SPROING, 1.0f);
        spawnParticles(PARTICLE_DUST, 10);
        move(0, 0, 20);
    }
};
```

### 注解

好啦，也许能够跳跃并不足以算是超能力，但是这里我尝试保持事情基础化。

这个力量把超级英雄弹向空中，播放一段恰当的音效并踢开一缕拂尘。如果所有的超级力量都如此简单 -- 仅仅是音效，粒子效果和动作的组合，那么我们就不再需要这个模式了。反而，`Superpower` 可以自带一个 `activate()` 的实现，这个 `activate()` 是访问音效ID，粒子类型和移动的部分。但是这个在仅当所有的力量基本上以同样的方式来工作仅仅在数据上有一些差异的地方才有效。让我们更详细的看一下：

```
class Superpower
{
protected:
    double getHeroX()
    {
        // Code here...
    }
```

```

    double getHeroY()
    {
        // Code here...
    }

    double getHeroZ()
    {
        // Code here...
    }

    // Existing stuff...
};


```

这里我们添加了一个方法用于获取英雄的位置。我们的 SkyLaunch 子类现在可以使用这些：

```

class SkyLaunch : public Superpower
{
protected:
    virtual void activate()
    {
        if (getHeroZ() == 0)
        {
            // On the ground, so spring into the air.
            playSound(SOUND_SPROING, 1.0f);
            spawnParticles(PARTICLE_DUST, 10);
            move(0, 0, 20);
        }
        else if (getHeroZ() < 10.0f)
        {
            // Near the ground, so do a double jump.
            playSound(SOUND_SWOOP, 1.0f);
            move(0, 0, getHeroZ() - 20);
        }
        else
        {
            // Way up in the air, so do a dive attack.
            playSound(SOUND_DIVE, 0.7f);
            spawnParticles(PARTICLE_SPARKLES, 1);
            move(0, 0, -getHeroZ());
        }
    }
};

```

由于我们可以使用一些状态，现在我们的沙盒函数可以做一些实际的有趣的控制流。这里仍然仅仅是一些简单的if语句，但是你可以做任何你想做的事情。通过使沙盒函数成为一个包含任意代码的切实丰富的函数，将具备无限的潜力。

#### 注解

起初，我建议对力量类采用数据驱动的方式。此处就是一个你决定不采用它的原因。如果你的行为是复杂和紧急的，定义数据将更困难。

## 设计决策

正如你所见，子类沙盒模式是一个相当“弱化”的模式。它描述了一个基本的思想，但并没有给出过于详细的机制。这就意味着你每次应用它的时候将面临一些抉择，可能就是如下的几个问题：

### 需要提供什么操作？

这是最大的问题。这深深地影响了这个模式的样貌以及它的表现如何。从小来说，基类不提供任何操作。它仅仅有一个沙盒函数。为了实现它，你将不得不调用基类之外的系统。如果从这个角度来说，说你正在用这个模式恐怕有些牵强。

而从大来讲，基类提供了子类需要的所有操作。子类仅仅与基类耦合并且不调用任何外部系统。

#### 注解

具体来说，这意味着每个子类的源文件仅仅需要#include其基类的头文件即可。

在这两种极端之间，有一个很宽阔的中间地带。在这个空间里，一些操作由基类提供，另外一些则通过定义它的外部系统直接访问。基类提供越多的操作，子类与外部系统耦合越少，但是基类就耦合得越多。它去掉了继承类的耦合，但是它是通过把耦合聚集到基类自己来实现的。

如果你有一堆与外部系统耦合的继承类的话，那么就可以使用这个模式，通过把耦合向上移到一个提供的操作，你就把它聚集到了一个地方：基类。但是你这样做得越多，基类就变得越大和越来越难于维护。

因此你的准绳应该摆在何处？这里有一些经验法则：

- 如果所提供的操作仅仅被一个或者少数的子类使用，那么不必将它加入基类这只会给基类增加复杂度，同时将影响每个子类，而仅有少数子类从中受益。

使这个操作与其他操作保持一致或许有价值，而使这些特殊情况的子类直接调用外部系统或许更简单清晰。

- 当你调用游戏中一些其他部分的函数的时候，如果那个函数不修改任何状态那么它就不会具备侵入性。它仍然创建了耦合，但是这是一个“安全”的耦合，因为在游戏中它不带来任何破坏。

#### 注解

带引号的“安全”意指，即使是访问数据也能引起问题。如果你的游戏是多线程的，你可以在数据被修改的同时读取数据。如果你不小心，最终得到的将是错误的数据。

另一个令人不快的情况是如果你的游戏状态是严格准确的（许多在线游戏为了保持玩家同步），而你访问一些同步游戏状态之外的东西，则将引起非常严重的非确定性bug。

而另一方面，如果这些调用确实改变了状态，则将与代码库产生更深层次的绑定，你需要对它有更多的了解，因为这些方法更适合于成为在可见的基类中来提供。

- 如果提供的操作其实现仅仅是对一些外部系统调用的二次封装，那么它并没有带来多少价值。在这种情况下，直接调用外部系统更为简单。

然而，极其简单的转向调用也仍有用 -- 这些函数通常访问基类不想直接暴露给子类的状态。例如，让我们看看 Superpower 提供的这个：

```
void playSound(SoundId sound, double volume)
{
    soundEngine_.play(sound, volume);
}
```

它仅仅在Superpower中转向调用一些soundEngine\_区域。这样的好处是把这种区域封装在 Superpower，以免子类访问它。

## 是否直接提供函数，还是通过包含它们的对象来提供？

这个设计模式的挑战在于最终你的基类塞满了大量的方法。你能够通过转移一些函数到其他类中来缓解这种情况。在基类中提供的函数然后仅仅返回这些对象之一。

例如，为了使一个力量类播放音效，我们能够直接添加这些到 superpower 中：

```
class Superpower
{
protected:
    void playSound(SoundId sound, double volume)
    {
        // Code here...
    }
}
```

```

void stopSound(SoundId sound)
{
    // Code here...
}

void setVolume(SoundId sound)
{
    // Code here...
}

// Sandbox method and other operations...
};


```

但是如果 `Superpower` 已经变得臃肿不堪，我们或许想避免这样做。反而，我们创建一个 `SoundPlayer` 类来暴露这种功能：

```

class SoundPlayer
{
    void playSound(SoundId sound, double volume)
    {
        // Code here...
    }

    void stopSound(SoundId sound)
    {
        // Code here...
    }

    void setVolume(SoundId sound)
    {
        // Code here...
    }
};


```

然后 `Superpower` 提供它的访问：

```

class Superpower
{
protected:
    SoundPlayer& getSoundPlayer()
    {
        return soundPlayer_;
    }

// Sandbox method and other operations...

private:
    SoundPlayer soundPlayer_;
};


```

把提供的操作分流到一个像这样的辅助类中能给你带来一些东西：

- 减少基类的函数数量。在例子中，我们从三个函数变成仅仅一个获取函数。
- 在帮助类中的代码通常更容易维护。像 `Superpower` 这样的核心基类，不论是否处于我们的意思，都因大量的依赖于它们而难于修改。通过把功能转移到一个耦合更少的第二候选类，我们可以使它的代码在不破坏的情况下更易于访问。
- 减少了基类和其他系统之间的耦合。当 `playSound()` 是一个直接定义在 `Superpower` 上的函数时，无论实现中调用了什么音效代码，我们基类就直接地与 `SoundId` 绑定了。把它转移到 `SoundPlayer` 减少了 `Superpower` 对单个 `SoundPlayer` 类的耦合，`SoundPlayer` 会封装其他的依赖。

## 基类如何获取需要的状态？

---

你的基类经常需要一些数据来封装和保持对子类的隐藏。在我们的第一个例子中，`Superpower` 类提供了一个 `spawnParticles()` 方法。如果这个方法的实现需要一些粒子系统对象，它该如何获得？

- 把它传递到基类构造函数：

最简单的方案是让基类把粒子系统作为一个构造函数参数：

```
class Superpower
{
public:
    Superpower(ParticleSystem* particles)
    : particles_(particles)
    {}

    // Sandbox method and other operations...

private:
    ParticleSystem* particles_;
};
```

这安全地保证了每个 `superpower` 在它构造的时候有一个粒子系统。但是让我们看看一个子类：

```
class SkyLaunch : public Superpower
{
public:
    SkyLaunch(ParticleSystem* particles)
    : Superpower(particles)
    {}

};
```

这里我们看到了问题。每个继承类将需要一个构造函数，这个构造函数调用基类的构造函数并传递那个参数。这样就向一些我们所不期望的状态暴露了每个子类。

同样也存在维护负担。如果稍后我们在基类中添加另一份状态，每个继承类的构造函数将不得不被修改来传递它。

- 进行二级初始化：

为了避免通过构造函数传递所有的东西，我们可以把初始化拆分为两个步骤。构造函数将不带参数仅仅创建对象。然后，我们调用一个直接定义在基类中的函数来传递它需要的余下部分数据。

```
Superpower* power = new SkyLaunch();
power->init(particles);
```

这里注意我们没有为 `SkyLaunch` 的构造函数传递任何东西，它并没有与我们希望在 `Superpower` 保持隐藏的东西产生耦合。采用这种方法困难的地方在于你必须确保你记得调用 `init()`。如果你忘记了，你将拥有一个潜藏的半创建状态不能工作的力量实例。

你可以通过封装整个过程到单个函数中来修改它。像这样：

```
Superpower* createSkyLaunch(ParticleSystem* particles)
{
    Superpower* power = new SkyLaunch();
    power->init(particles);
    return power;
}
```

#### 注解

通过一点小技巧比如私有化构造函数和友元函数，你可以保证 `createSkyLaunch()` 函数是唯一能够实际创建力量实例的函数。通过那种方式，你就不会忘记任何的初始化步骤。

- 使状态静态化：

在之前的例子中，我们用一个粒子系统实例来初始化每个 `superpower` 实例。当每个力量实例需要它们唯一的状态时这是有意义的。但是让我们看看粒子系统是一个 [单例](#)，每一个力量实例都将共享同样的状态。

在这种情况下，我们可以使这个状态对基类来说是私有的，同样也是静态的。游戏将仍然不得不保证初始化了状态，但是它仅仅需要针对整个游戏初始化 `Superpower` 类一次，而不是每个实例。

### 注解

请记住，单例仍然有许多的问题。你已经使一些状态在大量的对象之前共享（所有的 `Superpower` 实例）。粒子系统被封装，因此它不是全局可见，这很棒，但是仍然使得合理化力量实例更困难，因为它们可以访问同一个对象。

```
class Superpower
{
public:
    static void init(ParticleSystem* particles)
    {
        particles_ = particles;
    }

    // Sandbox method and other operations...

private:
    static ParticleSystem* particles_;
};
```

此处注意 `init()` 和 `particles_` 都是静态的。只要游戏调用 `Superpower::init()` 稍早调用一次，所有的力量实例都可以访问粒子系统。与此同时，`Superpower` 实例可以通过调用正确的继承类构造函数被自由创建。

更棒的是，现在 `particles_` 是静态变量，我们不必为每个 `Superpower` 实例储存它，因此我们使得类占用更少的内存。

- 使用服务定位器：

之前的做法需要外部代码明确地牢记在使用基类所需的状前将这些状态传递进去，这给周围代码的初始化工作带来了负担。另外一个选择是让基类通过把它需要的状态拉进去来处理。实现这个的一个方法是使用[服务定位器](#)模式。

```
class Superpower
{
protected:
    void spawnParticles(ParticleType type, int count)
    {
        ParticleSystem& particles = Locator::getParticles();
        particles.spawn(type, count);
    }

    // Sandbox method and other operations...
};
```

这里，`spawnParticles()` 需要一个粒子系统。它从服务定位器获取了一个，而不是通过外部代码获取。

## 参考

- 当你使用[Update Method](#)模式的时候，你的更新函数通常也是一个沙盒函数。
- [模板函数](#)模式正与本模式相反。在这两个模式中，你使用一些列原始操作实现一个函数。通过子类沙盒模式，函数在继承类中，原始操作则在基类中。通过模板函数，基类有函数，原始操作被继承类实现。
- 你也能够在[Facade模式](#)中看到一种变化。那个模式把一些不同的系统隐藏在一个简单API后。通过子类沙盒，基类扮演了一个隐藏整个游戏引擎使子类不可见的外观。

# 类型对象

---

## 目的

允许

- 感叹号让一些都更加激动人心！???

每个从**Monster**派生的类传入初始生命并重写**getAttack()**来返回这个品种的攻击字符串。一些都和预想的一样，不久之后，我们的主人公就能够跑来跑去杀死各种怪物。我们继续编写代码，在我们意识到之前，就会发现有成堆的怪物派生类，从酸性史莱姆到僵尸山羊。

然后，事情很奇怪地变得缓慢起来。我们的设计师最终想要有上百个品种，我们会发现自己花费了所有的时间去编写那7行代码的小派生类。更糟糕的是——设计师要开始调整代码中已经有的品种。我们的日常工作流程变成了这样：

- 1. 收到设计师的邮件，要把巨魔的攻击力从48修改成52。
- 1. 签出并修改**Troll.h**。
- 1. 重新编译游戏。
- 1. 签入修改。
- 1. 邮件通知。
- 1. 重复上述步骤。

我们整天都很茫然，因为我们变成了填数据的猴子。我们的设计师也很茫然，因为要调整好一个数字就要花费大量的时间。我们需要的能力是在无需重新编译整个游戏的情况下，去修改品种的数值。如果设计师能在无需程序员介入的情况下创建并调整品种，那就更好了。

## 一个类的类

在一个比较高的层次上，我们要解决的问题非常简单。我们的游戏中有一堆不同的怪物，我们想要让它们共享一些特性。一个部落中的怪物想要击败主人公，我们想要它们在攻击时使用相同的文本。我们通过将它们定义成同样的“品种”来实现，那个品种决定了攻击字符串。

因为它们属于直觉上的类，因此我们决定使用派生来实现这个概念。一头龙是一只怪物，游戏中的没头龙是这个龙的“类”的实例。将每个种族定义成抽象基类**Monster**的派生类，让游戏中的每个怪物成为派生的品种类的实例来影射???他。我们最终会有下面这样的类层次：

- ??? 这里的意思是“从什么派生”

游戏中每个怪物的实例，都属于一个派生的怪物类型。我们拥有的品种越多，这个类层级就越加庞大。这就是问题的成因：添加新的品种意味着添加新的代码，每个品种必须被编译成它自己的类型。

这是可行的，但并不是唯一的选择。我们也可以将代码的架构调整成每个怪物具有一个品种。而不是为每个品种做一次从**Monster**的派生，我们有一个唯一的**Monster**类和一个唯一的品种类：

- ??? 这里，的意思是“被什么引用”

完成了。就两个类。注意这里没有任何派生。在这个系统里，游戏中的每个怪物是一个简单的**Monster**类的实例。**Breed**类包含了同一品种的所有怪物之间共享的信息：初始生命值和攻击字符串。

为了将怪物与品种关联起来，我们给每个**Monster**一个到包含了品种信息的**Breed**的引用。为了获得攻击字符串，一个怪物在只需在它的品种上调用一个方法。这个品种本质上定义了怪物的“类型”。每个品种实例是一个表述不类型概念差异的对象，即这个模式的名字：类型对象。

这个模式的特殊能力是在无需重新编译代码的情况下，添加新的类型。我们本质上是把硬编码的类型继承系统移动了位置，放到一个我们可以在运行时定义的数据里。

我们可以通过实例化更多的**Breed**实例来创建成百上千的不同品种。如果我们通过一些配置文件中的数据初始化品种，我们就能够完全在数据里定义新的怪物类型。这简单到设计师都能做！

## 模式

定义一个类型对象类和一个被指定类型对象类。每个类型对象实例表示一个不同的逻辑类型。每个被指定类型的对象存储一个描述它的类型的类型对象的引用。

实例相关的数据被存储在被指定类型对象实例中，而所有同概念类型所共享的数据和行为被存储在类型对象中。引用同一个类型对象的对象会表现得好像他们是同类。这让我们能够在一个相似对象集合中共享数据和行为，很像是类派生让我们做到的事，但是无需一批硬编码的派生类。

## 何时使用它

这个模式在任何你需要定义一系列不同“种”东西，但是又不想把那些种类硬写进类型系统中的时候都有用。详细来说，只要下列任意一项成立时它就有用：

- 你不知道将来会有什么类型。（例如，我们的游戏是否需要支持下载包含怪物新品种的内容？）
- 能够在不重新编译或修改代码的情况下，修改或添加类型。

## 记住

这个模式是关于把“类型”的定义从命令式???而僵硬的语言代码转移到更加灵活而且低行为式???的内存对象。灵活性是好的，但是把类型移动到数据里还是会失去些东西。

## 类型对象必须手工跟踪

一个使用类似C++类型系统的好处是编译器自动处理所有的类登记。定义类的数据自动编译到可执行程序的静态内存分段中，就能工作了。

通过类型对象模式，我们现在不仅与责任管理内存中的怪物，还包括它们的类型——我们要保证只要有怪物需要它们，所有的品种对象就应该初始化并保留在内存中。当我们创建一个新的怪物时，我们有责任保证他由一个对合法品种的引用正确得初始化。

我们将自己从编译器的一些限制中解放出来，但代价是我们得重新实现一些它曾为我们做的事情。

- ??? 在内部，C++ 虚方法通过一种叫做“虚函数表”(virtual function table)东西实现，简称“vtable”。一个虚函数表是一个包含了函数指针集合的简单结构体，每个函数指针指向类中的一个虚方法。每个类在内存中都有一个虚函数表。每个类实例都有一个指向其类虚函数表的指针。
- ??? 当你调用虚函数的时候，代码首先从对象的虚函数表中查找，然后调用存储在表里的恰当的函数。
- ??? 听起来很相似？虚函数表就是我们的品种对象，指向虚函数表的指针是怪物对其品种的引用。C++ 类是类型对象模式在C上的应用，由编译器自动处理。

- 未每个类型定义行为很难

通过类派生，你可以重写一个方法然后做任何你想让它做的事——用程序计算数值，调用其他代码等等。没有任何界限。如果我们想，我们可以定义一个怪物子类，使它的攻击字符串根据月相而变化。（对狼人来说很方便，我觉得。）

而当我们改用类型对象的时候，我们用成员变量替代了方法重写。不是写一个重写父类方法去计算攻击字符串的怪物派生类，而是定义一个品种对象把攻击字符串存进另一个变量里面。

这使得通过类型对象去定义类型相关的数据非常容易，但是定义类型相关的行为却很难。如果，比如说，不同的怪物品种需要使用不同的AI算法，使用这种模式就很有挑战性。

有几种方法可以绕过这个限制。一个简单的解决方案是有一个固定的预定义行为集并使用类型对象中的数据去选择其一。例如，我们的怪物AI总是处于“站着不动”、“追逐主人公”或者“在恐惧中瑟瑟发抖”（嘿，它们可不会都是巨龙）。我们可以定义函数来实现每种行为。然后，我们可以把品种通过一个指向特定方法的指针与AI算法关联起来。

- ??? 听起来也很熟悉？现在我们真正在类型对象中实现了虚函数表。

另一个更强的解决方案是支持完全在数据中定义行为。[解释器模式](#)和[字节码](#)模式都让我们编译代表行为的对象。如果我们读取数据文件并使用它来为其中一种模式创建一个数据结构，我们将行为定义完全移动到了代码之外，放进内容之中。

- ??? 随着时间前进???, 游戏变得越来越数据驱动。硬件变得更加强大，我们发现自己更多受到自己所能编辑的内容而不是硬件所困扰。使用一个64K卡带的挑战是把游戏塞进去，使用一个双面DVD的挑战是把里面塞满游戏。
- ??? 脚本语言和其他高级的定义游戏行为的方式能够为我们带来必要的生产力提升，其代价是运行时性能无法达到最优。因为硬件在变得越来越好，但是脑力并没有，这项交换变得越来越有意义。

## 实例代码

在我们的第一个实现中，我们从简单入手，实现动机一节中所描述的基础系统。我们首先从**Breed**类开始。：

```
class Breed
{
public:
    Breed(int health, const char* attack)
        : health_(health),
          attack_(attack)
    {}

    int getHealth() { return health_; }
    const char* getAttack() { return attack_; }

private:
    int health_; // Starting health.
    const char* attack_;
};
```

非常简单。它只是一个包含了两个数据字段的容器：初始生命值和攻击字符串。让我们看看怪物如何使用它：

```
class Monster
{
public:
    Monster(Breed& breed)
        : health_(breed.getHealth()),
          breed_(breed)
    {}

    const char* getAttack()
    {
        return breed_.getAttack();
    }

private:
    int health_; // Current health.
    Breed& breed_;
};
```

当我们构造一个怪物时，我们给它一个品种对象的引用。它定义怪物的品种，而不是用我们之前的类派生。在构造函数中，怪物使用品种来确定它的初始生命值。要获得攻击字符串，怪物只要转而调用它的品种。

这段简单的代码是这个模式的核心思想。从这里开始所有的东西都是额外奖励。

## 使类型对象更加像类型：构造函数

用我们现有的东西，我们直接构造了一个怪物并负责把它的品种传进去。这与大多数面向对象语言实例化对象的过程有点相反——我们通常不会分配一段空内存然后给它一个类型。反之，我们先在类上面调用了构造函数，它负责给我们一个新的实例。

我们可以将这个模式应用到类型对象上面：

```
class Breed
{
public:
    Monster* newMonster() { return new Monster(*this); }

    // Previous Breed code...
};
```

- ???“模式”在这里是个正确的字眼。我们说其实是经典设计模式中的一种：[工厂方法](#)
- ???在一些语言中，这个模式用来创建所有的对象。在Ruby、Smalltalk、Objective-C和其他语言里，类也是对象，你通过调用类对象上的的方法去构造新的实例。

使用它们的类：

```
class Monster
{
    friend class Breed;

public:
    const char* getAttack() { return breed_.getAttack(); }

private:
    Monster(Breed& breed)
        : health_(breed.getHealth()),
          breed_(breed)
    {}

    int health_; // Current health.
    Breed& breed_;
};
```

关键的区别是Breed类里面的newMonster()函数。他是我们的“构造”工厂方法。在我们的原始实现中，创建一个怪物看起来是这样的：

- ??? 这里有另一个小区别。因为在C++实例代码中，我们可以使用一个方便的小特性：友元类。
- ??? 我们将怪物的构造函数定为私有，使得任何人都不能直接调用它。友元类绕开了这个限制，因此Breed仍然能够访问到它。这意味着创建怪物的唯一方法是通过newMonster()。

```
Monster* monster = new Monster(someBreed);
```

在修改过之后，它看起来是这样的：

```
Monster* monster = someBreed.newMonster();
```

那么，为什么要这么做呢？创建一个对象分为两步：分配内存和初始化。怪物构造函数让我们能够做所有的初始化操作。在例子里它被存进了品种里，但是整个游戏会加载图形、初始化怪物的AI然后做一些其他设定工作。

但是，这都发生在内存分配之后。我们在怪物的构造函数被调用之前，就已经有了一段内存。在游戏里，我们也希望能控制

对象创建的另一方面：我们通常使用一些自定义分配器或者[对象池](#)模式来控制对象在内存中的哪个地方结束。

在**Breed**里定义一个“构造”函数让我们有个地方放置这个逻辑。并非简单得调用**new**, **newMonster()**函数能够在把控制权移交到初始化函数之前从一个池或者自定义堆栈里拉取。通过把此逻辑放进唯一能创建怪物的**Breed**里，我们保证所有的怪物都经过我们预想的内存管理体系。

## 通过继承共享数据

我们现在已经实现了一个完全可用的类型对象系统，但是它还很基本。我们的游戏最终会有上千个种族，每一个都有一堆属性。如果一个设计师想要调整30多个巨魔品种，使他们更强一点而，她将要面对的会是一段无聊的工作。

一个有效的办法是像多个怪物通过品种共享多种特性一样，让品种之间也能够共享特性。就像我们在最初的面向对象方案那样，我们可以通过派生来实现。只是，我们不采用语言本身的派生机制，而是自己在类型对象内部实现它。

简单起见，我们只支持单继承。和一个用于基类的类一样，我们允许品种拥有一个基品种：

???? 代码

当我们构造一个品种时，我们给它一个传入一个基品种。我们可以传入NULL表示它没有祖先。

为了让它更有用，一个品种需要控制哪些特性从父类继承，哪些特性需要用它自己的。举个例子，只继承基品种中非零生命值的以及非NULL的攻击字符串。

有两种实现方式。一个是在属性每次被请求的时候执行代理调用，像这样：

???? 代码

这么做可以当品种在运行时修改后，即使不再有继承关系也能够正确执行。但另一方面，它占用更多的内存（必须保留一个指向父级的指针），而且更加慢。它必须在派生链上走一遍来查找一个属性。

如果我们确定品种的属性不会改变，一个更快的解决方案是在构造时应用继承。这被称为“复制”代理，因为我们在创建一个类型的时候把继承的属性复制到了这个类型内部。代码如下：

???? 代码

注意我们不再需要基类中的字段。一旦构造结束，我们就可以忘掉基类，因为他的属性已经被拷贝下来了。要访问一个品种的特性，现在我们只要返回它的字段。

???? 代码

又好又快！

假设游戏引擎从JSON文件创建品种。示例如下：

???? 代码

我们要写一段代码去读取每个品种项，然后用它里面的数据去创建实例。例子里巨魔基类是“**Troll**”，**Throll Archer**和**Troll Wizard**都是派生类。

因为这两个派生类的生命值都是0，所以这个值从父类继承。这意味着设计师能在**Troll**类中调整这个值，所有的三个品种都会一起更新。随着品种的数量和每个品种内部属性的增加，这能够节省很多时间。现在，通过一个非常小的代码段，我们完成了控制权在设计师手让他们能有效利用时间的一个开放系统。同时，我们可以不被打扰得编写其他功能。

## 设计决定

类型对象模式让我们像在设计自己的编程语言一样射击一个类型系统。设计空间非常广阔，我们可以做很多有趣的事情。

事实上，有些事情限制了我们的美好期盼。时间和可维护性会阻止我们向任何特别复杂的方向走。更重要的是，无论我们设计了怎样的类型系统，我们的用户（通常是非程序员）需要能很容易地理解它。我们做的越简单，它就更加可用。所以，我

们这里讲到的其实是个被反复践踏了的领域，把更深入的方向交给学者和爱探索的人吧。

## 类型对象应该封装还是暴露？

我们的简单实现里，**Monster** 有一个对品种的引用，但这个引用不是公开的。外面的代码无法直接访问到怪物的品种。从核心代码????的角度来说，怪物都是无类型的，它们有品种这件事是实现细节。

我们可以做个修改，让**Monster**返回它的品种：

???? 代码

本书的另一个例子里，我们紧跟着进行了一个转换，返回引用而不是指针来告诉用户，永远不会返回NULL。

这么做修改了**Monster**的设计。这样怪物有品种这件事就在API中可见了。这对双方都有好处。

- 如果类型对象被封装：
  - 类型对象模式的复杂性对代码库的其他部分不可见。它成为一个设计细节，只有有类型对象才关心它。
  - 有类型对象可以选择性地重写类型对象的行为。比如说我们想把怪物濒死时的攻击字符串改掉。由于攻击字符串都是从**Monster**访问的，我们有个现成的位置可以写：???? 代码 如果外部代码直接调用品种上的**getAttack()**，我们就没有机会插入这段逻辑。
  - 我们得给类型对象暴露的所有内容提供外部访问接口。这部分很乏味。如果我们的类型对象有一大堆方法，对象类为了公开，也必须提供一一对应的一大堆方法。
- 如果类型对象被公开：
  - 类型对象现在是对象公共API的一部分。通常，窄接口比宽接口更容易维护——你暴露给代码库的越少，你要面对的复杂性和维护负担就越少。

-- TODO

## 类型能否改变？

现在，我们假定一旦对象创建完成，就与其类型对象进行绑定，并不再改变。并不是一定要这样

◦ 我们可以允许一个对象随时间改变类型。

回头看我们的例子。当一个怪物死的时候，设计师告诉我们他们希望尸体能够变成会动的僵尸。

我们可以通过重新产生一个带有僵尸品种的新怪，但另一个更简单的选择是获取现有的怪物并把

它的品种修改成僵尸。

- 如果类型不变：
  - 无论编码还是理解起来都更简单。在概念层面上，“类型”是大多数人都不希望改变的东西。这么做符合这条假定。

- 易于调试。如果我们在跟踪一个让怪物陷入奇怪状态的Bug时，能够直观地确定正在看

的品种肯定是怪物始终不变的品种，这件事就相对简单了。

- 如果类型改变：
  - 更少的对象创建。在我们的例子里，如果类型不能改变，我们不得不在CPU循环中创建新的僵尸怪物，把原怪物中需要保留的属性逐个拷贝过来，随后删除它。如果我们能改变类型，所有的工作就是个简单的赋值。

- 做假定时要更加小心。对象和其类型之间存在相对紧的耦合。例如，一个品种可能假

定怪物的当前血量永远不会超过初始血量。如果我们允许改变品种，我们需要确保现有对象能符合新类型的要求。当我们修改类型

时，我们可能会需要执行一些验证代码来保证对象现在的状态对新类型来说有意义。

## 支持何种类型的派生？

- 没有派生：
  - 更简单。简单是最好的选择。如果你没有成堆的需要共享的类型对象，何必自找麻烦

呢？

- 可能会导致重复劳动。我曾见过给设计师用的不支持派生的编辑系统。当你有50中精

灵，必须去50个地方把它们的血量修改成相同的数字非常无趣。

- 单继承：
  - 仍然相对简单。更容易实现，但是，更重要的是，它很容易理解。如果非技术用户使用这个系统，会动的部分越少，就越好。很多编程语言只支持单继承是有原因的。它看起来是强大和简单之间的不错的平衡点。

- 属性查找更慢。要获得类型对象中的特定数据，我们需要在派生链中找到其类型，才

能最终确定它的值。如果我们在编写性能苛刻的代码，我们可能不想在这里浪费时间。

- 多重派生：
  - 绝大多数的数据重复都能被避免。通过一个好的多继承系统，用户能够创建一个几乎没有冗余的继承体系。比如做调整数值这件事，我们可以避免大量的复制粘贴。

- 复杂。很不幸的是，它的优点更多停留在理论上而不是实践上。多重派生很难理解或

说明。如果我们的僵尸龙类型从僵尸和龙派生，哪些属性从僵尸获得，哪些从龙获得呢？为了

使用这个系统，用户必须理解派生图如何遍历并要有预见性地射击一个聪明的体系。我所见到的大多数现代C++编码标准倾向于禁用多重派生，Java和C#则完全不支持。这承

认了一件不幸的事情：太难让它正确地工作以至于干脆不要用它。虽然它值得考虑，但是你很少会希望在游戏的类型对象中使用多继承。常言道，越简单越好。

## 参考

- 这个模式引出的高级问题是如何在不同对象之间共享数据。另一个从另一个角度引出这个问题的模式是[原型](#)
- 类型对象与[享元](#)很接近。它们都让你在实例间共享数据。享元模式倾向于节约内存，并且共享的数据可能不会以实际的“类型”呈现。类型对象模式的重点在于组织性和灵活性。
- 这个模式与[状态](#)模式也有很多相似性。它们都把对象的部分定义交给另一个代理对象实现

。在类型对象中，我们通常代理的对象是：宽泛地描述对象的恒定数据。在状态中，我们代理的是对象现在是什么样的，即：描述对象当前

配置的临时数据。

当我们讨论到可改变类型对象的时候，你会发现此时的类型对象兼任了状态的任务。

## 解耦模式

---

Once you get the hang of a programming language, writing code to do what you want is actually pretty easy. What's hard is writing code that's easy to adapt when your requirements *change*. Rarely do we have the luxury of a perfect feature set before we've fired up our editor.

当你掌握了一门编程语言，你会发现写代码来实现某个你想要实现的功能是件相当容易的事情。难的是写出在此基础上容易添加或更改功能的代码，因为我们几乎没有可能不更改程序的功能或者特性。

A powerful tool we have for making change easier is *decoupling*. When we say two pieces of code are "decoupled", we mean a change in one usually doesn't require a change in the other. When you change some feature in your game, the fewer places in code you have to touch, the easier it is.

[Components](#) decouple different domains in your game from each other within a single entity that has aspects of all of them. [Event Queues](#) decouple two objects communicating with each other, both statically and *in time*. [Service Locators](#) let code access a facility without being bound to the code that provides it.

[组件](#)将游戏的不同方面互相分离开却仍然具备它们的特性。[事件队列](#)能够静态而且及时的将两个对象通信分离开。[服务定位器](#)让代码能够访问到设备却不需要被绑定到提供服务的代码上。

## 本章模式

---

- [组件](#)
- [事件队列](#)
- [服务定位器](#)

# 组件模式

---

## Intent 意图

---

*Allow a single entity to span multiple domains without coupling the domains to each other.*

允许一个单独的实体跨多个不同域而不耦合它们。

## Motivation 动机

---

Let's say we're building a platformer. The Italian plumber demographic is covered, so ours will star a Danish baker, Bjørn. It stands to reason that we'll have a class representing our friendly pastry chef, and it will contain everything he does in the game.

举个例子，假设我们准备要制作一个平台类游戏。意大利水管工（译者注：作者指的是超级玛丽，超级玛丽是个水管工，在最初设计时被设定为了意大利人）已经有了，那我们设计一个丹麦面包师 Bjorn。显而易见，我们将设计一个能够很好的代表面包师的类，这个类包含了面包师的所有动作跟特性。

注：我之所以是个程序猿而非设计师就是因为我总想要去实现这些很棒的想法。

Since the player controls him, that means reading controller input and translating that input into motion. And, of course, he needs to interact with the level, so some physics and collision go in there. Once that's done, he's got to show up on screen, so toss in animation and rendering. He'll probably play some sounds too.

因为玩家控制他，这就意味着需要读取控制器的输入并且将输入转换成动作。当然，角色类还需要跟平台相互作用，所以还需要一些物理和碰撞方面的东西。当这些都完成了，角色通过动画和渲染就显示在屏幕上。角色可能还会播放一些音效。

Hold on a minute; this is getting out of control. Software Architecture 101 tells us that different domains in a program should be kept isolated from each other. If we're making a word processor, the code that handles printing shouldn't be affected by the code that loads and saves documents. A game doesn't have the same domains as a business app, but the rule still applies.

且慢，事情似乎在往失控的方向发展。在第一章软件架构中我们曾经提到，一个程序中的不同域应该互相隔离。在我们设计一个文字处理器时，处理打印部分的代码不应该受到保存，读取文档的代码的任何影响。也许游戏的域与应用的域不完全相同，但这个道理是相通的。

As much as possible, we don't want AI, physics, rendering, sound and other domains to know about each other, but now we've got all of that crammed into one class. We've seen where this road leads to: a 5,000-line dumping ground source file so big that only the bravest ninja coders on your team even dare to go in there.

所以尽可能的，我们不应让AI，物理，渲染，声效已经其他域互相影响，但我们又必须将所有这些包含在一个类中。我们已经看到了，这是一个代码量能达5000行以上的巨大的源文件，以至于只有最勇敢的程序员才胆敢去尝试。

This is great job security for the few who can tame it, but it's hell for the rest of us. A class that big means even the most seemingly trivial changes can have far-reaching implications. Soon, the class collects *bugs* faster than it collects *features*.

如此庞大的工作量对于那些能够驯服它的人来说这件很棒的事情，但是对无能为力的其余我们来说则如同地狱。一个如此大的类意味着即使最微不足道的修改都可能产生深远的影响。所以很快，在这个类中你将会得到比功能更多的错误。

## The Gordian knot 难题

Even worse than the simple scale problem is the coupling one. All of the different systems in our game have been tied into a giant knotted ball of code like:

比简单规模问题更复杂的是耦合问题。我们游戏中所有不同的系统都被绑成一个像巨大的结球一样的代码，就像：

```
if (collidingWithFloor() && (getRenderState() != INVISIBLE))
{
    playSound(HIT_FLOOR);
}
```

Any programmer trying to make a change in code like that will need to know something about physics, graphics, and sound just to make sure they don't break anything.

任何试图想要修改以上代码的程序都必须要知道相关物理、图像以及声音的知识以避免破坏任何功能。

注：While coupling like this sucks in *any* game, it's even worse on modern games that use concurrency. On multi-core hardware, it's vital that code is running on multiple threads simultaneously. One common way to split a game across threads is along domain boundaries -- run AI on one core, sound on another, rendering on a third, etc.当这种耦合的设计在任何游戏中都是一种糟糕的设计，但是在使用并发性的现代游戏中尤其糟糕。代码是否能够运行在多个线程上对拥有多核的硬件来说至关重要。而一个常见的实现多线程并行设计的方法就是设置域隔阂，比如让AI计算在一个核中完成，声效在另外一核，渲染在第三个，以此类推。

Once you do that, it's critical that those domains stay decoupled in order to avoid deadlocks or other fiendish concurrency bugs. Having a single class with an `UpdateSounds()` method that must be called from one thread and a `RenderGraphics()` method that must be called from another is begging for those kinds of bugs to happen.而要实现以上所说的设置不同域之间的隔阂，最至关重要的就是让不同的域之间保持去耦来避免产生死锁以及其他致命的并发错误。一个单类中，尝试在一个线程上调用 `UpdateSounds()` 方法而在另一个线程上调用 `RenderGraphics()` 方法，这无疑就是自取灭亡。

These two problems compound each other; the class touches so many domains that every programmer will have to work on it, but it's so huge that doing so is a nightmare. If it gets bad enough, coders will start putting hacks in other parts of the codebase just to stay out of the hairball that this `Bjorn` class has become.这两个问题互相复合，一个包含了域很多的类将要求每个想要修改他的程序员做大量的工作，而这毫无疑问就是个噩梦。当代码变得足够糟糕，程序员都开始因为不想去理这团杂乱的毛团而开始放弃它。

## Cutting the knot 解决难题

We can solve this like Alexander the Great -- with a sword. We'll take our monolithic `Bjorn` class and slice it into separate parts along domain boundaries. For example, we'll take all of the code for handling user input and move it into a separate `InputComponent` class. `Bjorn` will then own an instance of this component. We'll repeat this process for each of the domains that `Bjorn` touches.

想要解决这个问题，我们应该像执剑的亚历山大大帝一样。将独立的Bjorn类依着不用的域切成相互独立的部分。举个例子，我们将所有用来处理用户输入的代码放到一个类中。而Bjorn将拥有这个类的一个实例。我们将循环对所有Bjorn类包含的领域做同样的工作。

When we're done, we'll have moved almost everything out of `Bjorn`. All that remains is a thin shell that binds the components together. We've solved our huge class problem by simply dividing it up into multiple smaller classes, but we've accomplished more than just that.

但我们完成这件工作后，我们几乎将Bjorn类中的所有东西都清除了。剩下的是一个将所有组件绑在一起的外壳。我们通过简单的将代码分割成小类的方式解决了这个复杂的巨大类问题。但是我们却又不仅仅只是解决了这个问题。

## Loose ends 宽松的末端

Our component classes are now decoupled. Even though `Bjorn` has a `PhysicsComponent` and a `GraphicsComponent`, the two don't know about each other. This means the person working on physics can modify their component without needing to know anything about graphics and vice versa.

现在我们的内容类是去耦的了。尽管Bjorn类仍然有物理以及图像不同两块的内容，但是这两块内容互不干涉。这意味着想要

修改物理块内容的程序员不再需要了解图像块的知识，反之亦然。

In practice, the components will need to have *some* interaction between themselves. For example, the AI component may need to tell the physics component where Bjørn is trying to go. However, we can restrict this to the components that *do* need to talk instead of just tossing them all in the same playpen together.

在实践中，这些组件需要互相之间有一些互动。例如，AI组件可以会告知物理组件 Bjorn将去哪里。然而，我们可以限制的只让组件之间进行交流而不是将他们全部放到一起。

## Tying back together 捆绑在一起

Another feature of this design is that the components are now reusable packages. So far, we've focused on our baker, but let's consider a couple of other kinds of objects in our game world. *Decorations* are things in the world the player sees but doesn't interact with: bushes, debris and other visual detail. *Props* are like decorations but can be touched: boxes, boulders, and trees. *Zones* are the opposite of decorations -- invisible but interactive. They're useful for things like triggering a cutscene when Bjørn enters an area.

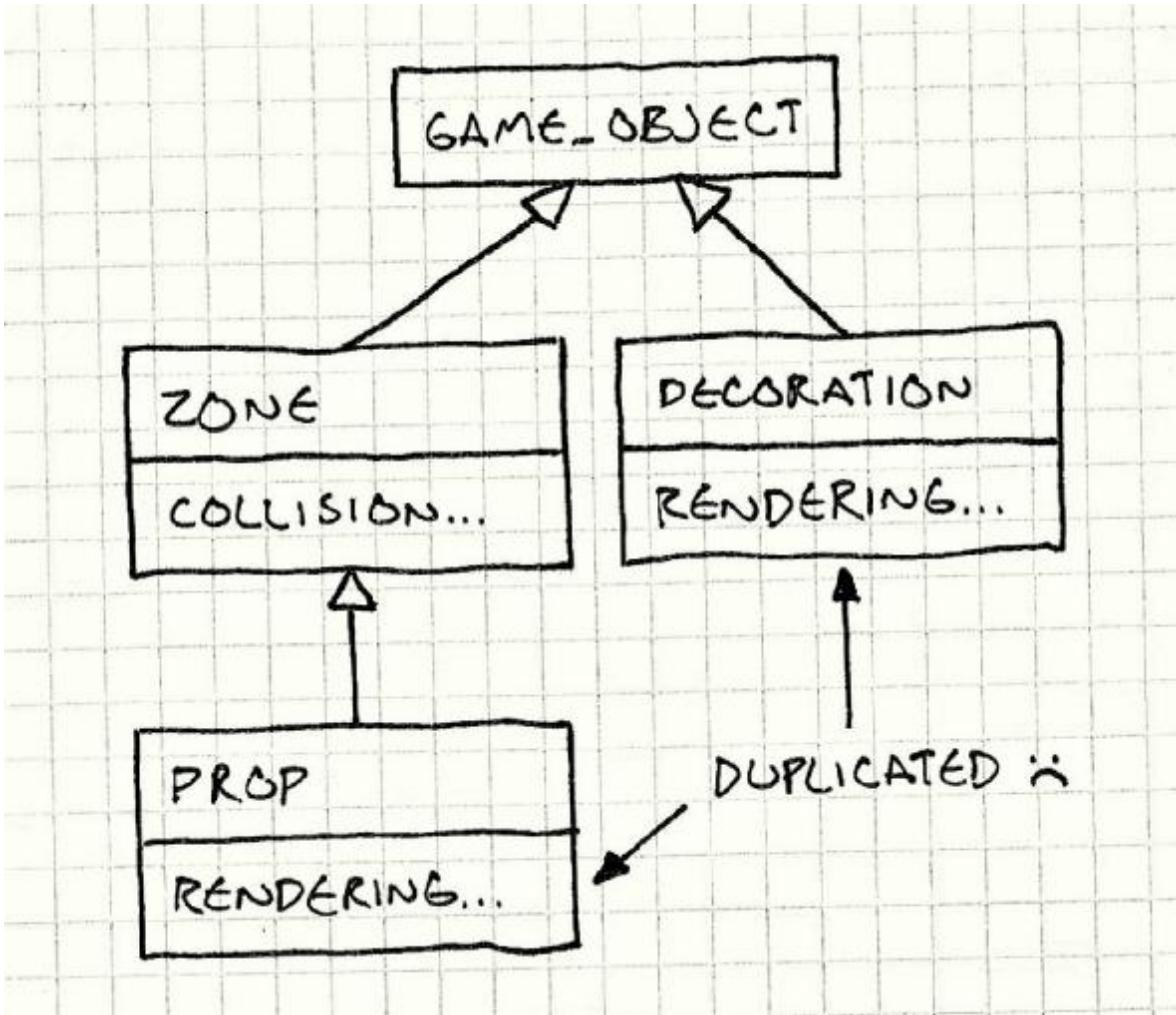
这个设计的另一种特性是可重用的组件包。到目前为止，我们都只是考虑了面包师一个角色，但游戏中可能出现的别的物品。例如灌木，碎片和其他的视觉细节等装饰是游戏中玩家能看到却不能交互的对象。而像盒子、巨石、树木等道具则是玩家可以与之交互的对象。分区则与装饰品正好相反——玩家看不到却能与之交互。上述这些对象将会在玩家的角色Bjorn进入一个区域时起作用。

注：When object-oriented programming first hit the scene, inheritance was the shiniest tool in its toolbox. It was considered the ultimate code-reuse hammer, and coders swung it often. Since then, we've learned the hard way that it's a heavy hammer indeed. Inheritance has its uses, but it's often too cumbersome for simple code reuse.当我们使用面对对象编程的时候，继承就是我们手上最有力的武器。它被认为是程序猿最喜欢用的终极武器。然而我们发现这个武器很多时候是把双刃剑，继承有它的用途，但是对某些代码重用来说实现起来太麻烦了。

Instead, the growing trend in software design is to use composition instead of inheritance when possible. Instead of sharing code between two classes by having them *inherit* from the same class, we do so by having them both *own an instance* of the same class.相反，软件设计的趋势应该是尽可能的使用组合而不是继承。我们应该让两个类拥有同一个类的实例而不是继承同一个类。

Now, consider how we'd set up an inheritance hierarchy for those classes if we weren't using components. A first pass might look like:

现在我们考虑如何在不用组件的情况下建立这些类的继承层次结构，它应该像：



We have a base `GameObject` class that has common stuff like position and orientation. `Zone` inherits from that and adds collision detection. Likewise, `Decoration` inherits from `GameObject` and adds rendering. `Prop` inherits from `Zone`, so it can reuse the collision code. However, `Prop` can't also inherit from `Decoration` to reuse the *rendering* code without running into the Deadly Diamond.

我们有一个基本游戏类，它包含像位置跟方向这种基本的元素。而区域继承了这个基本类并在其基础上加了碰撞。另外的，装饰也继承了基本类但是却添加了渲染。支柱继承区域，但是它重用了碰撞的代码。然后支柱不能继承装饰并重用渲染的代码，否则程序可能产生“致命的钻石”。

**注：**The "Deadly Diamond" occurs in class hierarchies with multiple inheritance where there are two different paths to the same base class. The pain that causes is a bit out of the scope of this book, but understand that they named it "deadly" for a reason. “致命的钻石”发生在对同一基类有不用路径的多重继承的类层次结构中。但是该错误的诱因不在这本书的讨论范畴内，但是请相信叫它致命的不是没有原因的。

We could flip things around so that `Prop` inherits from `Decoration`, but then we end up having to duplicate the *collision* code. Either way, there's no clean way to reuse the collision and rendering code between the classes that need it without resorting to multiple inheritance. The only other option is to push everything up into `GameObject`, but then `Zone` is wasting memory on rendering data it doesn't need and `Decoration` is doing the same with physics.

我们可以做些转变让支柱能够继承装饰类。但是我们将不能够复制碰撞的代码。无论如何，都没有办法不通过多重继承而在同一个类中重用碰撞跟渲染部分的代码。唯一的选择就是将这两段代码同时放到基本类中，然后这么做的结果就是区域类将会不需要渲染以及装饰类的代码而浪费了不少的内存。

Now, let's try it with components. Our subclasses disappear completely. Instead, we have a single `GameObject` class and two component classes: `PhysicsComponent` and `GraphicsComponent`. A decoration is simply a `GameObject` with a `GraphicsComponent` but no `PhysicsComponent`. A zone is the opposite, and a prop has both components. No code

duplication, no multiple inheritance, and only three classes instead of four.

现在，让我们试着用组件的方法。所有的子类完全消失，取而代之的是一个简单的基本类和两个组件：物理组件以及图像组件。所以装饰类就是一个同时包含基本类和图像组件的类，而区域则恰恰相反，支柱则是同时包含这两个组件，没有代码重复，没有多重继承，只有简单的三个类而不是四个。

注：A restaurant menu is a good analogy. If each entity is a monolithic class, it's like you can only order combos. We need to have a separate class for each possible *combination* of features. To satisfy every customer, we would need dozens of combos. 这好比一个餐厅的菜单，如果每个实体都是一个单独的类，那么也许你就只能点预定好的几个套餐。但是我们需要的一个能够结合不同特性的独立类。为了满足客户，我们可能需要数十个套餐。

Components are à la carte dining -- each customer can select just the dishes they want, and the menu is a list of the dishes they can choose from. 而组件就像按着菜单点菜用餐，每个客户能够选择那些他们喜欢的菜，而菜单则是一个他们选择菜品的列表。

Components are basically plug-and-play for objects. They let us build complex entities with rich behavior by plugging different reusable component objects into sockets on the entity. Think software Voltron.

## The Pattern 模式

A single entity spans multiple domains. To keep the domains isolated, the code for each is placed in its own **component class**. The entity is reduced to a simple **container of components**.

样式是一个跨多个域的单一实体。为了能够保持域之间相互隔离，每个域的代码都独立的放在自己的组件类中。实体则可以减少到一个的组件的容器。

> 注：“Component”，像“Object”，是那些在编程中意味着一切和什么都没有的词。因为这个，它被用来描述一些概念。在商业软件中，有一种“组件”设计模式，它描述了解耦服务，并能够通过网络进行通讯。> I tried to find a different name for this unrelated pattern found in games, but “Component” seems to be the most common term for it. Since design patterns are about documenting existing practices, I don't have the luxury of coining a new term. So, following in the footsteps of XNA, Delta3D, and others, “Component” it is. 我试图找到一个不同的名字来跟本文所说的游戏中的样式进行区别，但是“组件”仍然是最合适的名字。既然设计模式是用于记录已经存在的东西，我也没有那个荣幸能够创造一个新的术语。所以“组件”就随XNA，Delta3D之后，变成它现在的这个含义。

## When to Use It

Components are most commonly found within the core class that defines the entities in a game, but they may be useful in other places as well. This pattern can be put to good use when any of these are true:

组件一个最普遍的用法是在核心类中定义了一个游戏的实体，但是它们也能够用在别的地方。当如下条件成立时，样式就能够发挥它的作用：你有一个涉及到多个域的类，但是你想保持互相解耦。

- You have a class that touches multiple domains which you want to keep decoupled from each other. 你有一个涉及到多个域的类，但是你想保持互相解耦。
- A class is getting massive and hard to work with. 一个类越来越庞大，很难处理。
- You want to be able to define a variety of objects that share different capabilities, but using inheritance doesn't let you pick the parts you want to reuse precisely enough. 你希望定义能够共用不同功能的不同的类但却不通过继承来精确的重用代码。

## Keep in Mind 注意事项

The Component pattern adds a good bit of complexity over simply making a class and putting code in it. Each conceptual “object” becomes a cluster of objects that must be instantiated, initialized, and correctly wired together. Communication

between the different components becomes more challenging, and controlling how they occupy memory is more complex.

组件模式每次通过简单添加一点点的复杂的代码的方法来构成一个类。每个概念上的“对象”成为一个集群是必须被实例化，初始化以及正确地连接在一起。不同组件之间的通讯变得更具挑战性，并且限制它们占用内存将更加复杂。

For a large codebase, this complexity may be worth it for the decoupling and code reuse it enables, but take care to ensure you aren't over-engineering a "solution" to a non-existent problem before applying this pattern.

对于一个大型代码库，它的复杂性会让解耦以及代码重用变得有价值，但是请注意你没有在不存在问题的代码库中过度设计而使用这样一个“解决方案”。

Another consequence of using components is that you often have to hop through a level of indirection to get anything done. Given the container object, first you have to get the component you want, *then* you can do what you need. In performance-critical inner loops, this pointer following may lead to poor performance.

使用组件的另外一个后果是如果你需要经常跳过一个间接层来处理任何事情，考虑到容器对象，首先你必须得到你需要的组件，然后你才可以做你需要做的事情，在一些性能要求较高的代码中，这个指针可能会导致低劣的性能。

注：There's a flip side to this coin. The Component pattern can often *improve* performance and cache coherence. Components make it easier to use the [Data Locality](#) pattern to organize your data in the order that the CPU wants it. 就如硬币有正反面，而这就如同硬币的另外一面。组件模式通常能够提高性能和缓存一致性。组件让使用数据本地化模型来组织CPU所需要的数据顺序变得简单。

## Sample Code 范例代码

One of the biggest challenges for me in writing this book is figuring out how to isolate each pattern. Many design patterns exist to contain code that itself isn't part of the pattern. In order to distill the pattern down to its essence, I try to cut as much of that out as possible, but at some point it becomes a bit like explaining how to organize a closet without showing any clothes.

写这本书对我来说最大的挑战是找到如何隔离每个模式的方法。许多设计模式都包含了不属于与本模式无关的代码。为了提取模式的精华，我试着尽可能的简化，但是这就变得有点像是在展示一个没有任何衣服的衣柜

The Component pattern is a particularly hard one. You can't get a real feel for it without seeing some code for each of the domains that it decouples, so I'll have to sketch in a bit more of Bjørn's code than I'd like. The pattern is really only the component *classes* themselves, but the code in them should help clarify what the classes are for. It's fake code -- it calls into other classes that aren't presented here -- but it should give you an idea of what we're going for.

而组件模式则尤其困难。如果你没有见过任何解耦的代码，你将可能对组件的设计模式毫无头绪。所以我在我们Bjorn的代码上扩展开来向你们描述下。模式的确是只有组件本身，但是这其中的代码应该能够解释这个类。它是一段伪代码，它调用了其它不属于这里的类，但是它应该能够让你明白我们正在干什么。

### A monolithic class 一个单类

To get a clearer picture of how this pattern is applied, we'll start by showing a monolithic `Bjorn` class that does everything we need but *doesn't* use this pattern:

以便更清楚的了解这种模式是如何应用的，我们将从一个完成所有事情但是不使用组件模式的巨大单类Bjorn开始。

注：I should point out that using the actual name of the character in the codebase is usually a bad idea. The marketing department has an annoying habit of demanding name changes days before you ship. "Focus tests show males between 11 and 15 respond negatively to 'Bjørn'. Use 'Sven' instead." 我应该指出，使用实际名称中的字符代码库通常都是一个糟糕的想法。市场部有一个恼人的习惯就是要求你在发布应用前修改名字。“专注力测试的结果表示11到15岁之间的男生对Bjorn反响平平，请改为Sven。”

This is why many software projects use internal-only codenames. Well, that and because it's more fun to tell people you're working on "Big Electric Cat" than just "the next version of Photoshop." 这也是为什么许多软件项目使用只面向

内部的代号。因为告诉别人你正在开发一个“大电力猫”的程序比“新版本的Photoshop”要有趣多了。

```
class Bjorn
{
public:
    Bjorn()
        : velocity_(0),
          x_(0), y_(0)
    {}

    void update(World& world, Graphics& graphics);

private:
    static const int WALK_ACCELERATION = 1;

    int velocity_;
    int x_, y_;

    Volume volume_;

    Sprite spriteStand_;
    Sprite spriteWalkLeft_;
    Sprite spriteWalkRight_;
};
```

Bjorn has an `update()` method that gets called once per frame by the game:

Bjorn中有个update()方法来调用游戏中的每一帧：

```
void Bjorn::update(World& world, Graphics& graphics)
{
    // Apply user input to hero's velocity.
    switch (Controller::getJoystickDirection())
    {
        case DIR_LEFT:
            velocity_ -= WALK_ACCELERATION;
            break;

        case DIR_RIGHT:
            velocity_ += WALK_ACCELERATION;
            break;
    }

    // Modify position by velocity.
    x_ += velocity_;
    world.resolveCollision(volume_, x_, y_, velocity_);

    // Draw the appropriate sprite.
    Sprite* sprite = &spriteStand_;
    if (velocity_ < 0)
    {
        sprite = &spriteWalkLeft_;
    }
    else if (velocity_ > 0)
    {
        sprite = &spriteWalkRight_;
    }

    graphics.draw(*sprite, x_, y_);
}
```

It reads the joystick to determine how to accelerate the baker. Then it resolves its new position with the physics engine. Finally, it draws Bjørn onto the screen.它通过读取操纵杆来决定如何加速面包师。然后通过物理引擎来解决新位置的问题。最后将面包师显示到屏幕上。

The sample implementation here is trivially simple. There's no gravity, animation, or any of the dozens of other details that make a character fun to play. Even so, we can see that we've got a single function that several different coders on our team will probably have to spend time in, and it's starting to get a bit messy. Imagine this scaled up to a thousand lines and you can get an idea of how painful it can become.这个示例实现非常简单。没有重力，动画或者其他几十个能够让游戏变得有趣的细节。但即便如此，我们可以看到，我们有一个单一的函数让团队中不同的程序员都得花点时间，而且也开始有点混乱。

试想下如果代码扩展到一千行这将会是多么痛苦的一件事情。

## Splitting out a domain 分割域

Starting with one domain, let's pull a piece out of `Bjorn` and push it into a separate component class. We'll start with the first domain that gets processed: input. The first thing `Bjorn` does is read in user input and adjust his velocity based on it. Let's move that logic out into a separate class:从一个域开始，我们将一部分的Bjorn代码抽离出来并将它封装到一个独立的组件类中。我们首先从输入这个域开始。Bjorn类做的第一件事情就是读入用户的输入并相应的调整速度。让我们将这个逻辑封装到一个独立的类中：

```
class InputComponent
{
public:
    void update(Bjorn& bjorn)
    {
        switch (Controller::getJoystickDirection())
        {
            case DIR_LEFT:
                bjorn.velocity -= WALK_ACCELERATION;
                break;

            case DIR_RIGHT:
                bjorn.velocity += WALK_ACCELERATION;
                break;
        }
    }

private:
    static const int WALK_ACCELERATION = 1;
};
```

Pretty simple. We've taken the first section of `Bjorn`'s `update()` method and put it into this class. The changes to `Bjorn` are also straightforward:非常简单，我们只需要将Bjorn类中update方法放到一个新的类中就好了，而更改Bjorn类也相当简单：

```
class Bjorn
{
public:
    int velocity;
    int x, y;

    void update(World& world, Graphics& graphics)
    {
        input_.update(*this);

        // Modify position by velocity.
        x += velocity;
        world.resolveCollision(volume_, x, y, velocity);

        // Draw the appropriate sprite.
        Sprite* sprite = &spriteStand_;
        if (velocity < 0)
        {
            sprite = &spriteWalkLeft_;
        }
        else if (velocity > 0)
        {
            sprite = &spriteWalkRight_;
        }

        graphics.draw(*sprite, x, y);
    }

private:
    InputComponent input_;

    Volume volume_;

    Sprite spriteStand_;
    Sprite spriteWalkLeft_;
    Sprite spriteWalkRight_;
};
```

`Bjorn` now owns an `InputComponent` object. Where before he was handling user input directly in the `update()` method, now he delegates to the component:现在Bjorn拥有一个输入组件类，之前它通过调用update方法来处理用户的输入，现在它只需代理组件即可：

```
input_.update(*this);
```

We've only started, but already we've gotten rid of some coupling -- the main `Bjorn` class no longer has any reference to `controller`. This will come in handy later.我们仅仅才开始就已经摆脱了一些耦合——我们将逐步使得核心Bjorn类不再设计到任何控制器。

## Splitting out the rest 分割其余部分

Now, let's go ahead and do the same cut-and-paste job on the physics and graphics code. Here's our new `PhysicsComponent`:现在，让我们对物理以及图形的代码继续做同样的工作。这里给出了新的物理组件的代码：

```
class PhysicsComponent
{
public:
    void update(Bjorn& bjorn, World& world)
    {
        bjorn.x += bjorn.velocity;
        world.resolveCollision(volume_,
                               bjorn.x, bjorn.y, bjorn.velocity);
    }

private:
    Volume volume_;
};
```

In addition to moving the physics *behavior* out of the main `Bjorn` class, you can see we've also moved out the *data* too:除了将物理行为从核心类Bjorn中移除外，你还能看到我们同时将数据也移除了：现在音量对象是组件所拥有的。

Last but not least, here's where the rendering code lives now:最后但是同样重要的，是渲染部分的代码：

```
class GraphicsComponent
{
public:
    void update(Bjorn& bjorn, Graphics& graphics)
    {
        Sprite* sprite = &spriteStand_;
        if (bjorn.velocity < 0)
        {
            sprite = &spriteWalkLeft_;
        }
        else if (bjorn.velocity > 0)
        {
            sprite = &spriteWalkRight_;
        }

        graphics.draw(*sprite, bjorn.x, bjorn.y);
    }

private:
    Sprite spriteStand_;
    Sprite spriteWalkLeft_;
    Sprite spriteWalkRight_;
};
```

We've yanked almost everything out, so what's left of our humble pastry chef? Not much:我们几乎将所有东西都移除了，只剩下没有多少代码的Bjorn类：

```

class Bjorn
{
public:
    int velocity;
    int x, y;

    void update(World& world, Graphics& graphics)
    {
        input_.update(*this);
        physics_.update(*this, world);
        graphics_.update(*this, graphics);
    }

private:
    InputComponent input_;
    PhysicsComponent physics_;
    GraphicsComponent graphics_;
};

```

The `Bjorn` class now basically does two things: it holds the set of components that actually define it, and it holds the state that is shared across multiple domains. Position and velocity are still in the core `Bjorn` class for two reasons. First, they are "pan-domain" state -- almost every component will make use of them, so it isn't clear which component *should* own them if we did want to push them down. 现在Bjorn类只做两件很基础的事情，拥有一些能够真正工作起来的组件，以及拥有能够在不同的域共享的状态信息。位置和速度的信息之所以还保留在Bjorn类中主要有两个原因，首先它们是“pan-domain（最基础？）”状态，几乎所有的组件都必须使用它们，所以如果将它们放到组件中是不明智的。

Secondly, and more importantly, it gives us an easy way for the components to communicate without being coupled to each other. Let's see if we can put that to use. 第二点也是最重要的点是将位置与速度这两个状态信息保留在Bjorn类中使得我们轻松的在组件中传递信息而不需要耦合组件。让我们来看看应该如何使用。

## Robo-Bjørn 重构Bjorn

So far, we've pushed our behavior out to separate component classes, but we haven't *abstracted* the behavior out. `Bjorn` still knows the exact concrete classes where his behavior is defined. Let's change that. 到目前为止，我们已经将行为封装到单独的组件类中，但是我们没有将这些行为抽象化。Bjorn仍然精确的知道行为是在哪个类中被定义的。让我们来修改下。

We'll take our component for handling user input and hide it behind an interface. We'll turn `InputComponent` into an abstract base class: 我们将处理用户输入的组件隐藏到一个接口下，这样就能够将输入组件变成一个抽象的基类：

```

class InputComponent
{
public:
    virtual ~InputComponent() {}
    virtual void update(Bjorn& bjorn) = 0;
};

```

Then, we'll take our existing user input handling code and push it down into a class that implements that interface: 然后，我们将已经存在的用于处理用户输入的代码封装到一个实现了接口的类中：

```

class PlayerInputComponent : public InputComponent
{
public:
    virtual void update(Bjorn& bjorn)
    {
        switch (Controller::getJoystickDirection())
        {
            case DIR_LEFT:
                bjorn.velocity -= WALK_ACCELERATION;
                break;

            case DIR_RIGHT:
                bjorn.velocity += WALK_ACCELERATION;
                break;
        }
    }
};

```

```
private:  
    static const int WALK_ACCELERATION = 1;  
};
```

We'll change `Bjorn` to hold a pointer to the input component instead of having an inline instance: 我们改变Bjorn类，让它拥有一个指向输入组件的指针而不是拥有一个内联实例：

```
class Bjorn  
{  
public:  
    int velocity;  
    int x, y;  
  
    Bjorn(InputComponent* input)  
        : input_(input)  
    {}  
  
    void update(World& world, Graphics& graphics)  
    {  
        input_->update(*this);  
        physics_.update(*this, world);  
        graphics_.update(*this, graphics);  
    }  
  
private:  
    InputComponent* input_;  
    PhysicsComponent physics_;  
    GraphicsComponent graphics_;  
};
```

Now, when we instantiate `Bjorn`, we can pass in an input component for it to use, like so: 现在，当我们实例化Bjorn是，我们可以通过一个输入组件使用，像这样

```
Bjorn* bjorn = new Bjorn(new PlayerInputComponent());
```

This instance can be any concrete type that implements our abstract `InputComponent` interface. We pay a price for this -- `update()` is now a virtual method call, which is a little slower. What do we get in return for this cost? 这个实例可以是任何实现了我们抽象输入组件接口的具体类型。但是我们也因此付出代价，现在`update`方法是一个抽象方法调用方法，相对有点慢。我们应该反思，付出了这个代价我们得到了什么？

Most consoles require a game to support "demo mode." If the player sits at the main menu without doing anything, the game will start playing automatically, with the computer standing in for the player. This keeps the game from burning the main menu into your TV and also makes the game look nicer when it's running on a kiosk in a store. 大多数主机需要游戏支持“演示模式”。如果玩家停留在主菜单并且不做任何事情，电脑则会代替玩家让游戏则会自动的播放起来。这么做的目的是为了不要让游戏长时间的停留在主菜单画面，同时也为了在销售商店展示给顾客留下更好的印象。

Hiding the input component class behind an interface lets us get that working. We already have our concrete `PlayerInputComponent` that's normally used when playing the game. Now, let's make another one: 而将输入组件类隐藏到一个接口下有助于我们完成这个工作。我们已经有一个可供玩家正常游戏时的玩家输入组件。现在我们来制作另外一个输入组件：

```
class DemoInputComponent : public InputComponent  
{  
public:  
    virtual void update(Bjorn& bjorn)  
    {  
        // AI to automatically control Bjorn...  
    }  
};
```

When the game goes into demo mode, instead of constructing `Bjørn` like we did earlier, we'll wire him up with our new component: 当游戏进入演示模式时，我们像之前那样构建Bjorn类，取而代之的是将它连接到新的组件上：