在以下情形中，有限状态机十分有用：

* 当你有一个实体，其行为会依据某些内部状态而改变。
* 该状态能够被严格划分为相对较少的若干不同选项之一。
* 该实体随时间推移对一系列输入或事件做出响应。

测序模式

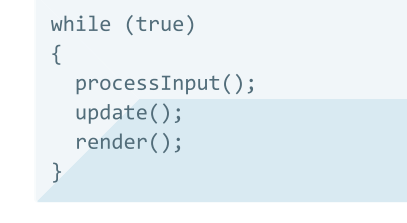
双缓冲

游戏循环

更新方法.

在大多数计算机中，答案是它从帧缓冲区中获取颜色信息。帧缓冲区是内存中的一个像素数组，是一段随机存取存储器（RAM），其中每几个字节代表一个像素的颜色。当软管在显示器上扫描时，它从这个数组中逐个读取颜色值。

而通过设置两个舞台，一个用于展示当前场景，另一个用于准备下一个场景，在场景切换时快速切换灯光，就如同计算机图形显示中的双缓冲技术，通过两个缓冲区（类似两个舞台），一个用于显示（类似正在展示场景的舞台），另一个用于绘制新画面（类似在黑暗中准备下一场景的舞台），在合适的时机切换（类似切换灯光），从而实现画面的无缝切换，避免观众察觉到场景之间的间隔，就像计算机图形显示中避免出现画面撕裂等问题，给用户流畅的视觉体验。



. 文本

AI 生成的内容可能不正确。

图形用户界面, 文本, 应用程序

AI 生成的内容可能不正确。

图形用户界面, 文本, 应用程序

AI 生成的内容可能不正确。

游戏循环

这些是批处理模式的程序 —— 一旦工作完成，程序就停止运行。



文本, 信件

AI 生成的内容可能不正确。

**可变时间步长的概念**

其思路是根据自上一帧以来实际经过的时间来选择推进游戏的时间步长。一帧所花费的时间越长，游戏推进的步长就越大。它总能跟上实时时间，因为它会通过越来越大的步长来实现这一点。

**可变时间步长方式**

这种方式既参考游戏内时间，也参考现实世界时间，但重点是依据现实世界时间来动态调整游戏更新的步长。

* **时间参考逻辑**：每一帧获取现实世界中自上一帧以来经过的时间（elapsed），然后将这个时间值传入游戏更新函数，让游戏状态根据这个实际经过的时间进行推进。比如屏幕上飞行的子弹，其移动距离会根据这一帧实际经过的时间来缩放速度，从而确定移动距离。
* **固定时间步长追赶策略**
* 该方式也是结合了游戏内时间和现实世界时间，不过是以固定的游戏内时间步长为基础，去追赶现实世界的时间。

**解决思路：利用残留延迟进行外推**

1. **获取残留延迟信息**：在退出更新循环时，lag 中存储的是当前距离下一帧更新的残留延迟。通过 lag / MS\_PER\_UPDATE 对这个值进行归一化处理，得到一个 0 到 1 之间的数值，表示当前处于两次更新帧之间的相对位置。
2. **传递归一化值给渲染器**：将归一化后的 lag / MS\_PER\_UPDATE 值传递给 render() 函数。渲染器根据这个值以及游戏对象的当前位置和速度，对游戏对象的位置进行外推计算。例如，当传递的值为 0.5 时，渲染器会将子弹绘制在提前半帧的位置，从而实现平滑的运动效果。

图形用户界面, 文本, 应用程序

AI 生成的内容可能不正确。

图形用户界面, 文本, 应用程序, 电子邮件

AI 生成的内容可能不正确。

**固定更新时间步长、可变渲染**

1. **原理**：以固定的时间步长更新游戏的核心状态（如物理模拟、AI 逻辑等），而渲染操作则根据实际情况灵活调整。当游戏更新跟不上实时时间时，可以选择丢弃一些渲染帧，优先保证游戏状态的更新。
2. **优点**：
   * **适应性强**：无论游戏运行过快还是过慢，只要游戏的更新能够实时进行，就不会出现游戏落后于现实时间的情况。在高性能设备上，游戏可以以更流畅的画面呈现，提供更好的游戏体验。
   * **稳定性好**：固定的更新时间步长保证了游戏核心逻辑的稳定性，物理模拟和网络交互等方面更容易处理。
3. **缺点**：实现复杂度较高。需要精心调整更新时间步长，以平衡高端设备和低端设备的性能。如果时间步长设置得太小，低端设备可能无法及时完成更新；如果设置得太大，高端设备上的游戏体验可能会受到影响。

文本, 信件

AI 生成的内容可能不正确。

文本

AI 生成的内容可能不正确。

文本, 信件

AI 生成的内容可能不正确。

行为模式

**相关模式**

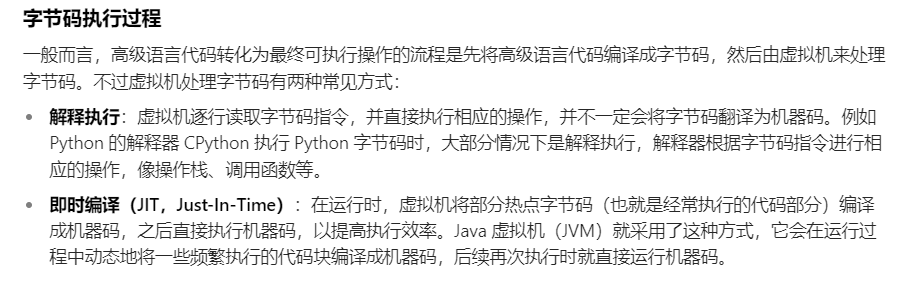
* 字节码
* 子类沙盒
* 类型对象

文本

AI 生成的内容可能不正确。

手机屏幕截图

AI 生成的内容可能不正确。



沙盒子类

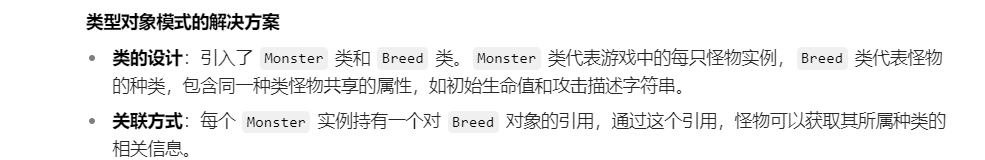
文本

AI 生成的内容可能不正确。

文本, 信件

AI 生成的内容可能不正确。

类型对象

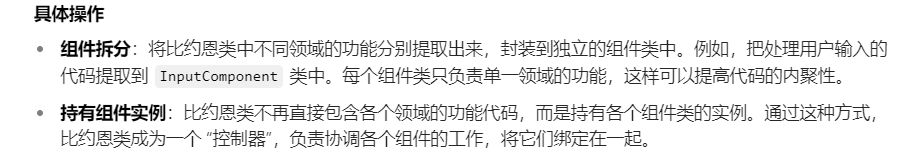


解耦模式

组件

事件队列

服务定位器



1. 你有一个类涉及多个领域，而你希望这些领域彼此保持解耦。
2. 一个类变得非常庞大，难以处理。
3. 你希望能够定义各种具有不同功能组合的对象，但使用继承无法让你足够精确地选择想要复用的部分。

**通过发送消息**

* **方式概述**：这是最复杂的一种选择。我们实际上可以在容器对象中构建一个小型消息系统，让组件相互广播信息

**中央事件总线**

* **游戏系统间的解耦通信**：在游戏开发中，不同的游戏系统（如战斗系统、教程系统等）可能需要进行通信，但又希望保持彼此的独立性和可维护性。中央事件队列作为一个全局的通信机制，允许任何游戏系统向队列中发送事件，也允许其他系统从队列中接收事件。

**问题 1：API 会阻塞调用者，直到音频引擎完全处理完请求**

**问题 2：无法对请求进行汇总处理**

**问题 3：请求在错误的线程上处理**

文本, 应用程序

AI 生成的内容可能不正确。

我已经提到过几次，从很多方面来看，这个模式是著名的观察者（Observer）模式的异步版本。

**与观察者模式的关系**

事件队列模式是观察者模式的异步版本。观察者模式是同步的，当一个对象的状态发生变化时，会立即通知所有观察者；而事件队列模式通过队列实现了异步通知，发送者将事件放入队列，接收者在合适的时间从队列中取出事件进行处理。

服务定位器

有一个更好的解决方案：电话簿。需要联系我们的人可以通过姓名查找我们，并获取我们当前的地址。当我们搬家时，我们告知电话公司。他们更新电话簿，然后每个人都能得到新地址。实际上，我们甚至根本不需要公布我们的真实地址。我们可以列出一个邮政信箱或其他形式的 “代表地址”。通过让来电者通过电话簿找到我们，我们就有了一个方便的单一控制点，用于管理如何被找到。

这就是服务定位器模式的核心思想 —— 它将需要服务的代码与服务的具体实现类型以及获取服务实例的方式解耦。

// 主函数中使用服务定位器

图形用户界面, 文本, 应用程序, 电子邮件

AI 生成的内容可能不正确。

文本

AI 生成的内容可能不正确。

优化模式

文本, 信件

AI 生成的内容可能不正确。.

**模式意图**

此模式的核心意图是采用延迟执行的策略，把工作推迟到真正需要结果的时刻再去执行，以此避免不必要的计算和资源消耗，提高程序的效率。

对象池

文本, 信件

AI 生成的内容可能不正确。

文本

AI 生成的内容可能不正确。

文本

AI 生成的内容可能不正确。

空间分区