基本的 CHIP-8 虚拟机

李睿远

Nov 09, 2025

探索早期游戏机的灵魂，用代码重现经典「PONG」和「太空入侵者」。

想象一个简单的黑白屏幕上，两个拍子来回击打一个像素点，这就是 CHIP-8 上的经典游戏 PONG。CHIP-8 并非真正的硬件，而是诞生于 1970 年代中期的解释性编程语言或虚拟机，最初在 COSMAC VIP 和 HP-48 计算器等设备上运行。它的设计简单且教育性强，是学习虚拟机、模拟器和计算机体系结构的完美入门项目。在本博客中，我们将使用你熟悉的编程语言，如 C++、Rust、JavaScript 或 Python，一步步实现一个完整的 CHIP-8 虚拟机，深入其核心设计并重现经典游戏。

1 CHIP-8 核心架构剖析

CHIP-8 虚拟机的核心架构包括内存、寄存器、栈、程序计数器、定时器、输入系统和显示系统。内存大小为 4KB，即 4096 字节，其布局从地址 0x000 到 0x1FF 保留给解释器本身，历史上用于存放 HP-48 计算器的 ROM。地址 0x050 到 0x0A0 用于内置的 4x5 像素十六进制字体集，而 0x200 到 0xFFF 是程序 ROM 和工作内存的起始点，大多数 CHIP-8 程序从这里开始执行。

寄存器系统包括 16 个 8 位通用寄存器，从 V0 到 VF。其中 VF 寄存器作为特殊标志寄存器，用于处理进位、借位和碰撞检测等操作。此外，还有一个 16 位地址寄存器 I，用于存储内存地址，方便数据存取。栈和栈指针用于子程序调用时存储返回地址，实现上通常使用一个数组作为栈，并有一个栈指针 SP 来跟踪栈顶位置。程序计数器 PC 指向当前执行的指令地址，初始值为 0x200，确保程序从正确位置开始。

定时器包括延迟定时器和声音定时器，两者都以 60Hz 频率递减。延迟定时器用于游戏逻辑控制，如移动速度；声音定时器当不为零时，会发出蜂鸣声，提供音频反馈。输入系统模拟一个 16 键的十六进制键盘，按键从 0 到 9 和 A 到 F，按键状态通常用一个 16 位的位掩码或布尔数组表示，便于检测按键事件。

显示系统是单色的，分辨率为 64x32 像素。绘制原理基于 XOR 操作：当绘制精灵时，精灵数据与屏幕像素进行 XOR 运算。如果某个像素从 1 翻转为 0，则表示发生碰撞，VF 寄存器被置为 1。这种机制简单高效，是 CHIP-8 图形渲染的核心。例如，在数学上，XOR 操作可以表示为 $a \oplus b$，其中 a 和 b 是二进制值，结果在它们不同时为 1。

2 实现我们的虚拟机

实现 CHIP-8 虚拟机首先需要搭建项目骨架和初始化所有组件。我们创建一个结构体或类来封装虚拟机的状态，包括内存数组、寄存器数组、栈数组、程序计数器 PC、地址寄存器 I、定时器等。初始化函数负责清空内存、重置寄存器为零，并将内置字体数据加载到内存地址 0x050 处。内置字体是 4x5 像素的字符集，用于显示十六进制数字。

指令循环是虚拟机的心脏，我们实现一个名为 emu_cycle 的函数，在主循环中不断调用。其伪代码可以描述为：首先，从内存中 PC 指向的地址读取两个字节，组合成一条 16 位指令；然后，解码指令的高 4 位操作码，确定指令类型；接着，执行对应的指令处理函数；之后，独立更新定时器，以 60Hz 频率递减；同时处理输入事件；最后，如果绘图指令被触发，则更新显示。这个循环确保虚拟机持续运行。

在核心模块实现中，指令解码与分发是关键。我们使用位操作来提取指令中的字段，例如，用掩码和移位获取 x、y、n、nnn 和 kk 等参数。例如，对于一个指令，高 4 位是操作码，接下来的 4 位可能是寄存器索引 x，再接下来是 y，而低 8 位可能是立即数 kk。通过位操作，我们可以高效地解析指令。代码示例如下：

```
uint16_t opcode = (memory[PC] << 8) | memory[PC + 1];
uint8_t op = (opcode & 0xF000) >> 12;
uint8_t x = (opcode & 0x0F00) >> 8;
uint8_t y = (opcode & 0x00F0) >> 4;
uint8_t n = opcode & 0x000F;
uint16_t nnn = opcode & 0x0FFF;
uint8_t kk = opcode & 0x00FF;
```

在这段代码中，我们首先从内存中读取两个字节，通过左移和或操作组合成一个 16 位指令 opcode。然后，使用位掩码和移位提取各个字段：op 是操作码，x 和 y 是寄存器索引，n 是低 4 位，nnn 是 12 位地址，kk 是 8 位立即数。这种解码方式确保了指令的正确解析。

关键指令集的实现需要分类处理。显示和图形指令如 Dxyn 用于绘制精灵，这是最复杂的指令之一。它从地址寄存器 I 指向的内存位置读取 n 字节的精灵数据，然后在坐标 (Vx, Vy) 处绘制到屏幕上。绘制时，每个字节的位与屏幕像素进行 XOR 操作，如果像素被从 1 翻转为 0，则设置 VF 寄存器为 1，表示碰撞。代码实现中，我们需要一个嵌套循环来处理每个像素。

流程控制指令包括 1nnn 用于跳转到地址 nnn，2nnn 用于调用子程序，00EE 用于从子程序返回。这些指令通过修改 PC 和栈来管理程序流程。例如，在调用子程序时，我们将当前 PC 压入栈，然后跳转到指定地址。

算术和逻辑指令如 8xy4 实现加法：将寄存器 Vx 和 Vy 的值相加，结果存入 Vx，如果发生进位，则设置 VF 为 1。类似地，8xy5 实现减法，8xy1 和 8xy2 分别处理 OR 和 AND 操作。这些指令通常涉及位运算，例如加法操作可以表示为 $V_x = V_x + V_y$，如果结果超过 255，则 VF 设为 1。

寄存器和内存操作指令中，Fx33 是 BCD 码转换的经典例子：它将寄存器 Vx 的值转换为三位 BCD 码，并存储到内存中 I、I+1 和 I+2 的位置。Fx55 和 Fx65 用于将寄存器 V0 到 Vx 存储到内存或从内存加载，这些指令涉及块数据传输。

定时器模块需要以固定频率更新，通常与主循环解耦。我们可以使用一个独立线程或定时器事件，每 1/60 秒递减延迟定时器和声音定时器。如果声音定时器大于零，则播放一个简单的蜂鸣声。图形和音频输出依赖于所选平台。例如，使用 SDL 库时，我们可以创建一个 64x32 的纹理，将虚拟机的显示缓冲区映射到屏幕上。音频输出可以通过播放一个短促的波形文件或使用 API 生成声音来实现。

3 测试与调试：让虚拟机活起来

测试虚拟机时，首先需要加载 CHIP-8 游戏 ROM。这些 ROM 可以从在线资源获取，如经典游戏 PONG、BRIX 或 INVADERS。我们编写一个函数，将 ROM 文件读入内存的 0x200 位置，确保程序正确加载。代码示例如下：

```
void load_rom(const char* filename) {
```

```
    FILE* file = fopen(filename, "rb");
    if (file) {
        fread(&memory[0x200], 1, sizeof(memory) - 0x200, file);
        fclose(file);
    }
}
```

在这段代码中，我们打开 ROM 文件并以二进制模式读取，将数据直接加载到内存起始地址 0x200 处。这确保了程序代码被正确放置，虚拟机可以开始执行。需要注意的是，文件大小不应超过可用内存空间，否则可能导致溢出。

调试是实现过程中的关键环节。我们可以添加日志输出，在每个循环周期打印 PC、当前指令和关键寄存器状态，便于跟踪执行流程。实现逐条执行模式，允许单步调试，帮助识别指令解码错误。常见问题包括字节序错误，例如在组合 16 位指令时顺序错误；定时器速度不匹配导致游戏运行过快或过慢；以及 XOR 绘图逻辑错误，导致显示异常。通过仔细检查代码和对比参考实现，可以逐步解决这些问题。例如，在绘图指令中，如果碰撞检测不正确，可能是 XOR 操作或像素翻转逻辑有误。

通过本博客的指导，我们成功实现了一个功能完整的 CHIP-8 虚拟机，能够运行几十年前的经典游戏如 PONG 和太空入侵者。这一过程让我们深入理解了虚拟机的基本工作原理，包括指令解码、执行、内存管理和 I/O 模拟。这些知识是学习更复杂系统如 NES 或 Game Boy 模拟器的坚实基础。

未来，我们可以尝试支持 SUPER-CHIP 扩展，提供更高分辨率和更多指令；或者优化性能，例如使用即时编译技术将 CHIP-8 指令转换为本地代码。此外，以此为起点，探索其他 8 位机体系结构，将进一步扩展我们的技能。虚拟机开发不仅是对历史的回顾，更是对计算机科学核心概念的实践，值得每一位开发者深入探索。