深入理解并实现基本的 JIT 编译器原理与优化

杨子凡

Oct 13, 2025

1 手把手带你构建一个迷你的 JIT 编译器，并探索现代运行时（如 JVM、V8）的性能奥秘

在软件开发领域，性能优化始终是一个核心议题。解释型语言如 Python 或 JavaScript 依赖于解释器逐条执行字节码，虽然启动速度快，但执行效率较低；而静态编译语言如 C++ 通过提前编译（AOT）生成高效的本地机器码，执行速度快却启动较慢，且缺乏运行时优化能力。JIT 编译器的诞生正是为了结合两者的优势：它在程序运行过程中，将字节码或中间表示编译成本地机器码，从而实现快速启动和高性能执行。JIT 编译器广泛应用于 Java 虚拟机（JVM）、JavaScript V8 引擎、.NET CLR 和 PyPy 等现代运行时环境中，成为动态语言性能提升的关键技术。

2 工作流程总览

JIT 编译器的工作流程始于源代码被解释器执行以进行“热身”，随后热点代码探测器识别出频繁执行的代码段，JIT 编译器将这些代码编译成本地机器码，并存储到代码缓存中供后续直接调用。这一流程实现了从解释执行到本地代码执行的平滑过渡，有效平衡了启动速度和运行效率。

3 关键组件详解

JIT 编译器的关键组件包括解释器、中间表示、热点代码探测器、编译器核心和代码缓存。解释器负责程序的初始执行，逐条解释字节码并为 JIT 编译提供运行数据。中间表示（IR）作为 JIT 编译器的工作对象，常见形式包括 Java 字节码或 JavaScript 的抽象语法树（AST）。热点代码探测器通过方法调用计数器和回边计数器来识别热点代码；方法调用计数器统计方法被调用的次数，而回边计数器则用于检测热循环。编译器核心将 IR 编译为目标机器码，涉及代码生成和优化过程。代码缓存存储已编译的机器码，以避免重复编译，提升整体效率。

4 目标设定

我们的目标是实现一个能 JIT 编译并执行简单算术表达式如 `(a + b) * c` 的迷你系统。通过这个实践示例，读者可以直观理解 JIT 编译的基本过程及其性能优势。

5 步骤一：定义中间表示

我们设计一个极简的基于栈的字节码指令集，包括指令如 PUSH、ADD、MUL 和 RET。PUSH 指令用于将值压入操作数栈，ADD 和 MUL 分别执行加法和乘法操作，RET 则返回结果。这种设计简化了字节码的解释和编译过程，便于后续实现。

6 步骤二：实现解释器

我们编写一个栈式解释器来执行上述字节码。解释器维护一个操作数栈，并逐条解释字节码指令。例如，当遇到 PUSH 指令时，它将指定值压入栈；遇到 ADD 指令时，它弹出栈顶两个值，相加后压回栈。以下是一个简单的 C 代码示例：

```
typedef enum {
   PUSH,
   ADD,
   MUL,
   RET
} OpCode;

int interpret(OpCode* code, int* constants) {
   int stack[100];
   int sp = 0;
   int ip = 0;
   while (1) {
      OpCode op = code[ip++];
      switch (op) {
         case PUSH:
            stack[sp++] = constants[ip++];
            break;
         case ADD: {
            int b = stack[--sp];
            int a = stack[--sp];
            stack[sp++] = a + b;
            break;
         }
         case MUL: {
            int b = stack[--sp];
            int a = stack[--sp];
            stack[sp++] = a * b;
            break;
```

```
29          }
            case RET:
31              return stack[--sp];
        }
33    }
}
```

这段代码定义了一个简单的解释器，它通过循环处理字节码指令。PUSH 指令从常量数组中读取值并压栈，ADD 和 MUL 指令执行相应的算术操作，RET 指令返回栈顶值作为结果。该解释器为 JIT 编译提供了基础执行环境。

7 步骤三：分配可执行内存

在 JIT 编译中，我们需要分配一块可写且可执行的内存来存储生成的机器码。在 Linux 系统中，可以使用 mmap 系统调用；在 Windows 系统中，可以使用 VirtualAlloc 函数。以下是一个 Linux 下的示例代码：

```
#include <sys/mman.h>
2
void* allocate_executable_memory(size_t size) {
4   return mmap(NULL, size, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE |
        ↪ MAP_ANONYMOUS, -1, 0);
}
```

这段代码使用 mmap 分配一块内存，并设置权限为可读、可写和可执行。这是 JIT 编译的核心步骤，因为它允许我们在运行时动态生成并执行机器码，突破了传统静态编译的限制。

8 步骤四：将字节码翻译成机器码

我们需要将字节码指令翻译成 x86-64 汇编指令，并编码为二进制机器码。例如，PUSH 指令对应 push 汇编指令，ADD 对应 add 指令。由于手动编码复杂，我们可以使用库如 AsmJit 来简化过程。以下是一个简化的手动编码示例：

```
1 unsigned char* generate_machine_code(OpCode* code, int* constants, size_t* code_size)
      ↪ {
    unsigned char* machine_code = allocate_executable_memory(100);
3   int offset = 0;
    // 示例：生成 mov rax, [rdi] 的机器码，假设参数在 rdi 寄存器
5   machine_code[offset++] = 0x48;
    machine_code[offset++] = 0x8b;
7   machine_code[offset++] = 0x07;
    // 添加更多指令编码 ...
9   *code_size = offset;
    return machine_code;
```

```
}
```

这段代码示意性地生成机器码，其中 `mov rax, [rdi]` 的机器码为 `0x48 0x8b 0x07`，用于从内存加载参数。实际 JIT 编译器需要根据字节码动态生成完整的函数体，包括算术运算和返回指令。

9 步骤五：执行 JIT 编译后的代码

一旦生成了机器码，我们可以将内存块转换为函数指针并直接调用它。以下是一个示例：

```
typedef int (*JITFunction)(int*);

JITFunction compile_to_function(OpCode* code, int* constants) {
    size_t code_size;
    unsigned char* machine_code = generate_machine_code(code, constants, &code_size);
    return (JITFunction)machine_code;
}

int main() {
    OpCode code[] = {PUSH, PUSH, ADD, PUSH, MUL, RET};
    int constants[] = {10, 20, 30};
    JITFunction func = compile_to_function(code, constants);
    int args[] = {10, 20, 30};
    int result = func(args);
    printf("Result: %d\n", result);
    return 0;
}
```

这段代码展示了如何将 JIT 编译后的机器码转换为函数指针并调用。通过这种方式，我们可以直接执行编译后的本地代码，避免了解释器的开销，从而提升性能。

10 成果演示

通过对比解释执行和 JIT 编译执行同一段代码，我们可以观察到显著的性能差异。例如，对于循环执行表达式 `(a + b) * c`，JIT 编译版本可能快数倍，因为它将字节码转换为高效的本地机器码，减少了运行时解释开销。

11 为什么要优化?

JIT 编译器初始生成的机器码往往是“直译”式的，效率不高。优化旨在提升代码质量，使其媲美甚至超越 AOT 编译器的性能。通过运行时信息，JIT 编译器可以进行动态优化，这是静态编译难以实现的。

12 经典的 JIT 优化策略

JIT 编译器采用多种优化策略，如方法内联、常量传播与折叠、逃逸分析、循环优化和本地优化。方法内联通过将短小方法体复制到调用处来消除函数调用开销，这是最重要的优化之一。常量传播与折叠在编译期计算常量表达式，例如将表达式 $2 + 3$ 直接替换为 5，减少运行时计算。逃逸分析判断对象是否逃逸方法作用域，如果未逃逸，则可以在栈上分配或消除对象，减少堆分配开销。循环优化包括循环展开和循环不变代码外提；循环展开减少循环控制指令，而循环不变代码外提将循环中不变的计算移到外部。本地优化如公共子表达式消除和死代码消除，则进一步提升代码效率。

13 基于性能分析的优化

现代 JIT 编译器使用分层编译和去优化技术。分层编译包括客户端编译器（如 HotSpot 的 C1）和服务端编译器（如 C2 或 Graal），前者快速编译但优化少，后者慢速但进行激进优化。去优化则是一种安全机制，当优化假设被打破时（如类继承变化），JIT 可以回退到解释器或低优化代码，确保正确性。这种基于性能分析的优化使 JIT 编译器能够自适应地调整编译策略。

14 现代 JIT 的发展

近年来，JIT 技术不断发展，例如 GraalVM 和 Truffle 框架基于 Java 实现高性能语言运行时，而 V8 引擎的 Ignition 解释器和 TurboFan 编译器架构则代表了 JavaScript JIT 的先进水平。这些创新推动了跨语言优化和即时编译的边界。

15 JIT 编译器的挑战

JIT 编译器面临编译开销、内存占用和预热时间等挑战。编译过程本身消耗 CPU 和内存，需要在编译时间和性能收益间平衡；代码缓存增加内存使用；程序在达到峰值性能前需要预热期，这可能影响短期任务。解决这些挑战需要精细的启发式算法和资源管理策略。

JIT 编译器通过运行时编译热点代码，巧妙结合了解释器和编译器的优势，为动态语言和托管环境提供了接近本地代码的性能。它是现代高性能运行时的基石，未来随着 AI 技术的引入，可能出现更智能的自适应 JIT 编译器，进一步推动软件性能优化。

推荐阅读周志明的「深入理解 Java 虚拟机」、V8 官方博客、OpenJDK HotSpot 文档，以及 AsmJit 和 LLVM JIT 等相关库的文档。这些资源提供了深入的理论和实践指导，帮助读者进一步探索 JIT 编译器的世界。