构建编译器的基础原理

杨子凡

Dec 10, 2025

编译器是一个将高级编程语言源代码转换为目标机器可执行代码的复杂程序。它通过多阶段处理，确保源代码的语义被精确映射到底层硬件指令。与解释器不同，解释器如 Python 的 CPython 在运行时逐行执行代码，而编译器则预先完成整个翻译过程，生成独立的可执行文件。这种静态编译方式通常带来更高的运行时性能，因为所有分析和优化都在编译期完成。

编译器的历史可以追溯到 1950 年代的 Fortran 编译器，那是第一个将科学计算公式转化为机器指令的工具。随后，经典的《编译原理》（Dragon Book）奠定了现代编译理论基础。今天，GCC 和 LLVM 等工具主导了开源领域，支持从 C++ 到 Rust 的多种语言。

学习构建编译器不仅仅是技术挑战，更是培养深刻编程思维的过程。它帮助你理解语言设计决策背后的权衡，比如类型系统或垃圾回收机制；同时掌握性能优化技巧，在面试中脱颖而出，如 Google 或字节跳动常考编译原理。本文面向有 C 或其他编程基础的读者，使用 Go 语言实现示例，所有代码可在配套 GitHub 仓库中找到。我们的目标是循序渐进，从零构建一个支持基本算术表达式的迷你编译器。

经典编译器模型分为四个核心阶段：前端处理源语言特有分析，中端进行机器无关优化，后端生成平台特定代码，还有贯穿始终的优化管道。本文将严格遵循这一 pipeline，从词法分析起步，直至代码生成，并穿插高级主题。想象一个流程图：源代码流经 lexer、parser、语义检查、IR 生成、优化器，最终输出机器码 —— 这将是我们的路线图。

1　编译器的整体架构

编译器的整体架构可以分为前端、中端和后端三个主要部分，前端紧密依赖源语言特性，负责将人类可读的源代码转化为结构化的中间表示；中端则在这一表示上执行优化，这些优化与具体硬件无关；后端最终将优化后的表示映射到目标平台的机器指令。

前端包括词法分析、语法分析和语义分析，生成抽象语法树或初步中间代码。中端使用统一的中间表示如三地址码，进行数据流分析和变换。后端涉及指令选择、寄存器分配和汇编生成。工具链极大简化了开发：Lex 和 Yacc（或现代的 Flex 和 Bison）自动化生成词法和语法分析器，而 LLVM 提供成熟的中后端基础设施，支持从 IR 到多种架构的代码生成。

考虑一个简单示例：源代码 `int main() { return 1 + 2; }` 前端解析为 AST，中端优化为常量折叠 `return 3;`，后端生成 x86 汇编如 `mov eax, 3; ret`。这一转换流程体现了架构的模块化：每个阶段独立测试，便于维护和扩展。

2 第一阶段：词法分析

词法分析是编译 pipeline 的入口，它将连续的源代码字符流分解为离散的 Token 序列，例如关键字 `if`、标识符 `foo`、数字 `42` 或运算符 `+`。这一过程依赖正则表达式驱动的有限状态机，通常实现为确定性有限自动机（DFA），确保高效线性扫描。

实现词法分析器的第一步是定义 Token 类型。在 Go 中，我们可以用枚举模拟这一概念：

```
type TokenType string

const (
    IDENTIFIER TokenType = "IDENTIFIER"
    NUMBER TokenType = "NUMBER"
    PLUS TokenType = "PLUS"
    EOF TokenType = "EOF"
)

type Token struct {
    Type TokenType
    Value string
}
```

这里，`TokenType` 是字符串常量模拟的枚举，`Token` 结构体携带类型和字面值。接下来构建状态机，手写一个简单 scanner 函数：

```
func (l *Lexer) NextToken() Token {
    for {
        switch l.ch {
        case '+':
            tok := Token{PLUS, "+"}
            l.readChar()
            return tok
        case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
            start := l.position
            for isDigit(l.ch) {
                l.readChar()
            }
            return Token{NUMBER, l.input[start:l.position]}
        default:
            if isLetter(l.ch) {
                start := l.position
                for isLetter(l.ch) {
```

```
                l.readChar()
            }
            return Token{IDENTIFIER, l.input[start:l.position]}
        }
        l.readChar()
    }
  }
}
```

这个 Lexer 结构体维护输入字符串 input、当前字符 ch 和位置 position。NextToken 循环读取字符：遇到 + 直接返回 PLUS Token 并推进位置；数字则收集连续数字字符，使用 isDigit 检查（未示出，为简单 ASCII 检查）；标识符类似，收集字母序列。readChar 方法递增位置并更新 ch，处理空白通过循环忽略。这一手写 DFA 模拟了状态转换：从初始状态跳转到数字状态或运算符状态，避免回溯。

常见挑战包括处理嵌套注释、多字符 Token（如 ==）和最大 Munch 规则，即优先匹配最长 Token，如 if 而非单个 i。字符串转义如 \ 需要栈模拟。相比手写，Flex 生成器从正则定义 .l 文件自动构建 DFA，但手写更易理解边界情况。

实践一个完整 lexer，输入 Hello+1，输出 Tokens：{IDENTIFIER, Hello}、{PLUS, +}、{NUMBER, 1}、{EOF, }。测试用例验证忽略空格和注释，确保 Token 流准确无误。这一阶段奠定了解析基础，下节我们将这些 Token 组织成树状结构。

3 第二阶段：语法分析

语法分析接收词法分析器输出的 Token 流，根据上下文无关文法（CFG）规则验证其结构，并构建抽象语法树（AST）。CFG 用 BNF 表示，如算术表达式文法：$E \to E + T \mid T$，$T \to T * F \mid F$，$F \to \text{number} \mid (\text{E})$。为避免左递归，我们改写为 LL(1) 兼容形式。

解析方法主要分为自顶向下和自底向上。自顶向下如递归下降解析器，直观递归匹配非终结符，适合 LL 文法；自底向上如 LR 解析器，使用栈和状态表处理更复杂文法。

实现递归下降解析器，先定义 AST 节点：

```
type Node interface {
    TokenLiteral() string
}

type InfixExpression struct {
    Token Token
    Left Node
    Operator string
    Right Node
}

func (ie *InfixExpression) TokenLiteral() string { return ie.Token.Literal }
```

Node 接口统一访问 Token 值。解析函数如：

```
func (p *Parser) parseExpression() Node {
    left := p.parseTerm()
    for p.peekToken.Type == PLUS {
        token := p.peekToken
        p.nextToken()
        right := p.parseTerm()
        node := &InfixExpression{Token: token, Left: left, Operator: token.Literal,
            ↪ Right: right}
        left = node
    }
    return left
}
```

Parser 维护当前 curToken 和预读 peekToken，nextToken 推进。parseExpression 先解析 term（优先级低），循环检查 +，构建左结合的 InfixExpression 树。类似地，parseTerm 处理 * 以体现优先级。输入 1+2*3，先解析 1，遇 + 暂存，解析 2*3 为右子树，最终 AST 为 (1 + (2 * 3))。
移进-归约冲突通过优先级表解决：* 优先于 +，栈模拟如 LR 状态机。错误恢复采用 panic 模式：跳过无效 Token 至同步点。
实践输入 1+2*3，输出 AST：根节点 +，左 1，右 *（左 2 右 3）。解析栈从 Tokens 逐步归约，生成树状结构。这一阶段确保语法正确，下节引入语义检查。

4 第三阶段：语义分析

语义分析在 AST 上执行静态检查，如类型兼容、作用域解析和符号表管理。它不改变程序结构，但标注错误或丰富节点属性。核心是符号表，一个哈希表存储标识符信息：

```
type Symbol struct {
    Name string
    Type string
    Scope int
}

type SymbolTable map[string]Symbol
```

符号表按作用域层级管理。遍历 AST 构建表：

```
func (s *SemanticAnalyzer) Visit(node Node) {
    switch n := node.(type) {
    case *VarDecl:
        if _, exists := s.symTable[n.Name]; exists {
            s.errors = append(s.errors, "redeclared␣variable:␣"+n.Name)
```

```
            return
        }
        s.symTable[n.Name] = Symbol{Name: n.Name, Type: n.Type, Scope: s.scope}
    case *Identifier:
        sym, exists := s.symTable[n.Value]
        if !exists {
            s.errors = append(s.errors, "undeclared␣variable:␣"+n.Value)
            return
        }
        n.Type = sym.Type // 类型传播
    }
}
```

SemanticAnalyzer 后序遍历 AST：声明节点插入符号表，检查重定义；使用节点查询表，报告未声明错误，并传播类型。作用域用栈模拟，进入块增层，出块回退。类型检查如 `int + string` 触发不兼容错误。
属性文法通过合成属性（如从子树推导表达式类型）实现：二元运算节点类型为左右最小公类型。实践添加类型到 `int x = 1 + 2` 的 AST，报告 `x = 1 + 2` 类型错误。符号表从全局到局部嵌套，类型推导自底向上。这一阶段桥接语法与代码生成，下节生成中间表示。

5 第四阶段：中间代码生成

中间表示（IR）是机器无关的桥梁，便于优化和多后端支持。常见 IR 如三地址码（TAC，每指令最多三操作数）、静态单赋值（SSA）和 LLVM IR。
TAC 示例：`t1 = a + b`。从 AST 生成：

```
func (cg *CodeGenerator) Generate(node Node) string {
    switch n := node.(type) {
    case *InfixExpression:
        left := cg.Generate(n.Left)
        right := cg.Generate(n.Right)
        temp := cg.NewTemp()
        cg.Emit(fmt.Sprintf("%s␣=␣%s␣%s␣%s", temp, left, n.Operator, right))
        return temp
    case *NumberLiteral:
        return n.Value
    }
    return ""
}
```

CodeGenerator 后序遍历：叶子数字直接返回值，二元节点生成临时变量 `t1`，emit TAC 指令如 `t1 = 1 + 2`。NewTemp 递增计数器。新鲜临时变量便于优化。

SSA 增强版引入版本号：`x1 = a + b; x2 = x1 + c`，消除伪依赖。LLVM IR 更丰富，支持 phi 节点。实践 `1+2*3` AST 转为 TAC：`t1 = 2 * 3; t2 = 1 + t1`。AST 到 IR 转换保留语义，剥离语法糖，为优化铺路。

6 第五阶段：优化

优化提升 IR 效率，分类为局部（单基本块，如常量折叠）和全局（跨块，如循环优化）。数据流分析如到达定义（Reaching Definitions）追踪变量来源。

常量折叠预计算：`2 + 3` 替换为 5。简单 pass：

```
func (opt *Optimizer) FoldConstants(ir []string) []string {
    var result []string
    for _, instr := range ir {
        if matchesConstFold(instr) {
            folded := fold(instr) // 如 "t1 = 2 + 3" -> "t1 = 5"
            result = append(result, folded)
        } else {
            result = append(result, instr)
        }
    }
    return result
}
```

`FoldConstants` 扫描 IR，模式匹配常量二元运算，用求值替换。死代码消除移除不可达块：分析控制流图（CFG），标记 live 代码。

公共子表达式消除（CSE）复用重复计算：`t1 = a + b; t2 = a + b` 共享 `t1`。循环不变式外提将 `i * c`（c 常量）移出循环。数据流方程如 $RD_{in}(B) = \bigcup_{p \in pred(B)} (OUT_p - GEN_p)$ 求解。

实践优化 `t1 = 1 + 2; t2 = t1 * 3` 为 `t1 = 3; t2 = 9`。优化前后 IR 对比凸显指令减少，CFG 节点间边标注 live-in/live-out。这一阶段抽象硬件细节，下节生成具体代码。

7 第六阶段：代码生成与汇编

代码生成将 IR 转为目标汇编，如 x86。过程包括指令选择（模式匹配树覆盖）和寄存器分配（图着色）。

简单栈机 TAC 到汇编：

```
func (cg *CodeGenerator) GenerateAsm(ir []string) string {
    var asm []string
    asm = append(asm, "section␣.text")
    asm = append(asm, "global␣_start")
    for _, instr := range ir {
        if strings.HasPrefix(instr, "t") && strings.Contains(instr, "=") {
            parts := strings.Split(instr, "␣=␣")
            left := parts[0]
```

```
            right := parts[1]
            cg.emitLoad(asm, right, "rax")
            cg.emitStore(asm, left, "rax")
        }
    }
    return strings.Join(asm, "\n")
}
```

解析 t1 = 1 + 2，加载操作数到寄存器 rax，执行 add，存储回栈或变量。寄存器分配建干扰图：冲突节点不同色（寄存器），Chaitin 算法贪心着色溢出至栈。
指令选择用 DAG 匹配：表达式树覆盖最优指令序列。实践生成 mov eax, 3; ret，链接为可执行文件运行输出 3。冲突图节点为 live 范围重叠变量，边表示分配约束。

8 高级主题与扩展

高级编译涉及运行时支持，如垃圾回收的 Mark-Sweep 接口，与 IR 交互管理堆。JIT 编译如 V8 将字节码动态优化为机器码，支持热路径加速。现代工具 LLVM 提供 IR 库，Cranelift 专注 WASM 快速后端。
调试依赖单元测试每个 pass 和 AST 可视化如 Graphviz。性能调优用基准如编译 1MB 代码时间，剖析瓶颈。
编译 pipeline 从字符串到机器码，体现了模块化设计：每个阶段独立演进，从 lexer 的 FSM 到后端的图算法。掌握它，你能设计 DSL 或优化现有语言。
实践构建 TinyC，支持 if/while/函数：扩展本文代码，添加控制流 IR 和跳转优化。推荐《编译原理》（Dragon Book）、《Crafting Interpreters》及 NandGame 在线课程。配套仓库：github.com/yourname/tiny-compiler。Fork 它，添加特性，分享你的编译器之旅！

9 附录

完整代码仓库：github.com/example/tiny-compiler（包含所有示例和测试）。
术语表：Token 是最小语法单元；AST 是树状源表示；IR 是优化中介。
参考文献：Dragon Book、Crafting Interpreters、LLVM 文档等。
常见问题：为什么不直接用 LLVM？手写理解原理，LLVM 适合生产。