c13n #40

c13n

2025 年 11 月 19 日

第 I 部

基本的布隆过滤器（Bloom Filter）数据结构

王思成

Nov 04, 2025

在当今数据爆炸的时代，处理海量信息的存在性判断成为一个常见挑战。例如，在恶意 URL 检测场景中，我们可能有一个包含十亿个 URL 的黑名单，如何快速判断一个新 URL 是否在其中？如果使用传统的哈希表，内存占用可能高达数十 GB，这在实际应用中往往不可行。数据库查询虽然能节省内存，但查询速度较慢，无法满足实时性要求。布隆过滤器应运而生，它通过巧妙的概率设计，用极小的空间开销和恒定的查询时间，解决了「可能存在」和「肯定不存在」的问题。其代价是允许一定的误判率，但绝不漏判，这使得它在许多场景中成为理想选择。

1 什么是布隆过滤器?

布隆过滤器的核心思想可以概括为一个由随机映射函数和二进制向量组成的概率型数据结构。想象一个多指纹采集器，每个元素进来时，会通过多个哈希函数生成多个指纹，并将这些指纹记录在一个比特数组中；判断时，只需检查所有指纹是否都存在。这种设计使得布隆过滤器在空间效率和查询速度上远超一般算法，同时不需要存储元素本身，具有一定保密性。然而，它也存在缺点，例如存在误判率，无法删除元素，且不支持获取所有元素。这些特性决定了它的适用边界，在允许一定误差的场景中表现卓越。

2 深入原理：布隆过滤器如何工作?

布隆过滤器的核心组件包括一个长度为 m 的比特数组和 k 个相互独立且分布均匀的哈希函数。比特数组初始状态所有位都为 0，而哈希函数将输入元素映射到 [0, m-1] 范围内的整数索引。添加元素时，例如添加字符串 geekhour，首先将其输入 k 个哈希函数，得到 k 个哈希值 h1, h2, ..., hk，然后将比特数组中对应这些位置的位全部设置为 1。查询元素时，同样计算 k 个哈希值，并检查这些位置是否全部为 1；如果全是 1，则返回「可能存在」；如果有任何一位为 0，则返回「肯定不存在」。这种设计确保了绝不漏判，因为如果一个元素确实被添加过，它设置的位不可能被清零，查询时一定能看到所有位都是 1。误判的发生是由于哈希冲突，导致不同元素的哈希值可能覆盖相同位置。

3 关键参数与误判率分析

布隆过滤器的误判率受三个核心参数影响：预期添加的元素数量 n、比特数组长度 m 和哈希函数个数 k。误判率的近似公式为 P ≈ (1 - e^(-k * n / m))^k，这个公式直观地展示了参数间的权衡。当 m 增大时，比特数组空间更大，冲突减少，误判率降低；当 n 增大时，元素越多，数组越满，误判率升高；而 k 的个数需要平衡，太少容易冲突，太多则数组快速饱和。实践中，最优哈希函数个数可通过公式 k = (m / n) * ln(2) 计算。例如，如果要求误判率为 1%，通常推荐 m 约为 n 的 10 倍，此时最优 k 值约为 7。这种参数选择帮助用户在空间和准确性之间找到平衡点。

4 动手实现：一个简单的布隆过滤器

我们将使用 Python 实现一个简单的布隆过滤器类，命名为 SimpleBloomFilter。这个类包含三个成员变量：size 表示比特数组大小 m，hash_count 表示哈希函数个数 k，以及

bit_array 用于模拟比特数组。在 Python 中，我们可以使用内置的 bytearray 或第三方库如 bitarray 来高效管理位操作，但为了简单起见，这里用整数列表模拟，每个整数代表一个位。

首先，我们需要模拟多个哈希函数。一个高效的方法是使用双重哈希技术，基于一个基础哈希函数生成 k 个不同的哈希值。具体公式为 hash_i(x) = (hash1(x) + i * hash2(x)) % size，其中 hash1 和 hash2 可以是简单的哈希函数如 FNV-1a 或 MurmurHash 的变体。这种方法确保了哈希值的独立性和均匀分布，同时避免了实现多个独立哈希函数的复杂性。

初始化方法 `__init__(self, size, hash_count)` 负责设置比特数组大小和哈希函数个数，并将比特数组初始化为全 0。在代码中，我们用一个长度为 size 的列表表示，每个元素初始为 0。

添加方法 `add(self, item)` 首先将输入项转换为字符串或字节，以确保一致性。然后，计算 k 个位位置，通过循环调用哈希函数生成索引，并将比特数组中对应位置设为 1。在实现中，我们使用位操作来高效设置位，例如通过位或运算。

查询方法 `contains(self, item)` 类似地计算 k 个位位置，并检查这些位置是否全为 1；如果是，返回 True，表示可能存在；否则返回 False，表示肯定不存在。

以下是一个简单的 Python 实现示例：

```
class SimpleBloomFilter:
    def __init__(self, size, hash_count):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = [0] * size # 初始化比特数组为全 0

    def _hashes(self, item):
        # 模拟双重哈希生成 k 个哈希值
        item = str(item).encode('utf-8')
        h1 = hash(item) # 使用 Python 内置哈希作为基础，实际应用中应使用更均匀的哈希函数
        h2 = h1 ^ 0xFFFFFFFF # 简单变换生成第二个哈希值
        hashes = []
        for i in range(self.hash_count):
            hashes.append((h1 + i * h2) % self.size)
        return hashes

    def add(self, item):
        for pos in self._hashes(item):
            self.bit_array[pos] = 1 # 设置对应位为 1

    def contains(self, item):
        for pos in self._hashes(item):
            if self.bit_array[pos] == 0:
                return False # 如果任何一位为 0，肯定不存在
```

```
        return True # 所有位为 1，可能存在
```

在这段代码中，初始化方法创建了一个指定大小的列表作为比特数组。哈希生成方法 _hashes 将输入项转换为字节后，使用 Python 内置哈希函数生成基础值，并通过线性组合生成 k 个索引；实际应用中，建议使用更均匀的哈希函数如 FNV-1a 以减少冲突。添加方法遍历哈希值，将对应位置设为 1；查询方法检查所有位置，如果任何一位为 0 则立即返回 False。这种实现虽然简单，但清晰地展示了布隆过滤器的核心逻辑。

为了测试这个实现，我们可以创建一个布隆过滤器实例，设置 m=1000 和 k=7，然后添加一组单词如 [hello, world, bloom, filter]。测试 contains(hello) 应返回 True，而 contains(foo) 应返回 False。随着添加元素增多，误判可能发生，例如在添加大量元素后，查询一个未添加项可能返回 True，这演示了布隆过滤器的概率特性。

5 进阶话题与优化

布隆过滤器有多种变体，例如计数布隆过滤器，它将比特位扩展为计数器，支持元素的删除操作。在计数布隆过滤器中，添加元素时计数器加一，删除时减一，但这增加了空间开销，并可能引发计数溢出问题。应用场景方面，布隆过滤器广泛用于缓存穿透防护，通过在查询数据库前判断键是否存在，避免无效查询；在爬虫 URL 去重中，快速判断 URL 是否已抓取；在垃圾邮件过滤中，检查发件人是否在黑名单内；以及在数据库如 LevelDB 中优化查询，减少不必要的磁盘访问。这些应用凸显了布隆过滤器在高效处理大规模数据时的价值。

回顾布隆过滤器的核心，它通过空间换时间和概率交换确定性的哲学，解决了传统方法在高并发和大数据场景下的瓶颈。使用时需权衡其优缺点：适用于对空间敏感、允许误判的快速查询场景；而不适用于要求百分百准确、需要删除操作或存储元素本身的场景。鼓励读者在项目中探索布隆过滤器的应用，例如在分布式系统中使用变体优化性能。进一步学习可关注高效哈希函数或分布式布隆过滤器的实现，以深化对这一数据结构的理解。

第 II 部

基本的动态脚本执行引擎

王思成

Nov 05, 2025

动态脚本在现代软件开发中无处不在。从游戏中的技能脚本到网页中的 JavaScript，再到办公软件中的宏和持续集成中的 Pipeline 脚本，这些功能都依赖于动态脚本执行引擎。动态脚本的核心价值在于其灵活性：它允许程序在运行时改变逻辑，无需重新编译整个应用程序。这带来了可扩展性，使用户或开发者能够定制和扩展功能；支持热更新，可以在线修复问题或更新玩法；以及促进了领域特定语言（DSL）的发展，为特定场景创造更高效的工具。本文旨在带领读者从理论到实践，深入理解动态脚本引擎的内部机制。在理论层面，我们将探讨脚本引擎的三大核心阶段：词法分析、语法分析和解释执行。在实践层面，我们将使用 Python 语言实现一个微型脚本引擎，能够执行四则运算、变量赋值和条件判断，例如处理像 `x = 10 + 2 * (3 - 1); if (x > 15) { y = 1; } else { y = 0; }` 这样的脚本。

6 理论基石：脚本引擎的三大核心阶段

一个脚本引擎的工作流程可以概括为三个核心阶段：词法分析、语法分析和解释执行。词法分析负责将源代码字符串转换为一系列有意义的「单词」或 Token。每个 Token 包含类型（如标识符、数字、运算符、关键字）和值（如变量名「x」、数字「10」）。这一过程类似于人类阅读时识别单词，它使用有限自动机（DFA）作为理论模型来识别 Token 序列。例如，源代码 `position = 10 + 20` 会被转换为 Token 列表：`[IDENTIFIER(position), OPERATOR(=), NUMBER(10), OPERATOR(+), NUMBER(20)]`。

语法分析阶段则根据预定义的语法规则，将 Token 流组织成抽象语法树（AST）。AST 是一种树形数据结构，它忽略源代码中的标点等细节，只保留程序的逻辑结构。上下文无关文法（CFG）常用于定义语言的语法规则，而递归下降法是一种简单直观的语法分析方法，适合手动实现。例如，对于表达式 `10 + 2 * 3`，AST 会确保乘法运算符的优先级高于加法，通过树结构正确表示计算顺序。

解释执行是最终阶段，它遍历 AST 并执行其中蕴含的操作。有两种主要策略：遍历解释和编译到字节码。遍历解释直接访问 AST 节点并执行相应操作，简单易实现；编译到字节码则先将 AST 编译成中间指令，由虚拟机执行，效率更高。在本文中，我们将采用遍历解释的方式，为每个节点类型定义执行逻辑，同时维护一个作用域来管理变量。

7 动手实现：构建微型脚本引擎

我们将使用 Python 语言实现一个微型脚本引擎，支持数字、字符串、布尔值数据类型，以及四则运算、逻辑比较、变量赋值、if-else 条件语句和语句块。实现过程分为五个步骤：定义语言语法、实现词法分析器、实现语法分析器、实现解释器，以及组装测试。

首先，我们定义迷你脚本语言的语法。它支持基本的表达式和语句，例如变量赋值使用 = 运算符，条件语句使用 `if-else` 结构，表达式可以包含加减乘除和比较操作。数据类型包括数字（如 `10`）、字符串（如 `hello`）和布尔值（如 `true` 和 `false`）。

接下来，实现词法分析器（Lexer）。词法分析器的输入是源代码字符串，输出是 Token 列表。我们定义 Token 类型枚举，包括 `IDENTIFIER`、`NUMBER`、`OPERATOR`、`KEYWORD` 等。在实现中，我们使用循环逐个字符扫描源代码，识别标识符、数字、运算符和关键字，并跳过空白字符。例如，以下 Python 代码展示了词法分析器的核心逻辑：

```
class TokenType:
```

```
    IDENTIFIER = 'IDENTIFIER'
    NUMBER = 'NUMBER'
    OPERATOR = 'OPERATOR'
    KEYWORD = 'KEYWORD'

class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

def lexer(source):
    tokens = []
    pos = 0
    while pos < len(source):
        char = source[pos]
        if char.isspace():
            pos += 1
            continue
        elif char.isalpha():
            start = pos
            while pos < len(source) and source[pos].isalnum():
                pos += 1
            value = source[start:pos]
            if value in ['if', 'else', 'true', 'false']:
                tokens.append(Token(TokenType.KEYWORD, value))
            else:
                tokens.append(Token(TokenType.IDENTIFIER, value))
        elif char.isdigit():
            start = pos
            while pos < len(source) and source[pos].isdigit():
                pos += 1
            value = int(source[start:pos])
            tokens.append(Token(TokenType.NUMBER, value))
        elif char in '=+-*/(){}<>!=':
            if char == '=' and pos + 1 < len(source) and source[pos+1]
                ↪ == '=':
                tokens.append(Token(TokenType.OPERATOR, '=='))
                pos += 2
            else:
                tokens.append(Token(TokenType.OPERATOR, char))
                pos += 1
        else:
```

```
            raise SyntaxError(f"未知字符␣:␣{char}")
    return tokens
```

在这段代码中，我们定义了 Token 类和 TokenType 枚举来表示 Token 的类型和值。lexer 函数通过遍历源代码字符串，使用条件分支识别不同字符类型。例如，当遇到字母时，它读取连续的字母数字字符，判断是否为关键字或标识符；遇到数字时，读取连续数字并转换为整数；遇到运算符时，处理单字符或多字符情况（如 ==）。错误处理部分在遇到未知字符时抛出异常，为后续完善错误信息奠定基础。

语法分析器（Parser）的输入是 Token 列表，输出是 AST 根节点。我们定义 AST 节点类型，如 NumberLiteral、BinaryExpression、AssignmentStatement、IfStatement 等。使用递归下降法，我们编写解析函数来处理表达式和语句。例如，parse_expression 函数处理表达式，考虑运算符优先级；parse_statement 处理语句；parse_program 解析整个程序。以下代码展示了部分实现：

```
class ASTNode:
    pass

class NumberLiteral(ASTNode):
    def __init__(self, value):
        self.value = value

class BinaryExpression(ASTNode):
    def __init__(self, left, operator, right):
        self.left = left
        self.operator = operator
        self.right = right

class AssignmentStatement(ASTNode):
    def __init__(self, identifier, expression):
        self.identifier = identifier
        self.expression = expression

class IfStatement(ASTNode):
    def __init__(self, condition, then_branch, else_branch):
        self.condition = condition
        self.then_branch = then_branch
        self.else_branch = else_branch

class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0
```

```
    def current_token(self):
        if self.pos < len(self.tokens):
            return self.tokens[self.pos]
        return None

    def eat(self, token_type):
        token = self.current_token()
        if token and token.type == token_type:
            self.pos += 1
            return token
        else:
            raise SyntaxError(f"期望␣{token_type}，但得到␣{token.type␣if␣
                ↪ token␣else␣'EOF'}")

    def parse_expression(self):
        left = self.parse_term()
        while self.current_token() and self.current_token().type ==
            ↪ TokenType.OPERATOR and self.current_token().value in ['+'
            ↪ , '-']:
            operator = self.eat(TokenType.OPERATOR).value
            right = self.parse_term()
            left = BinaryExpression(left, operator, right)
        return left

    def parse_term(self):
        left = self.parse_factor()
        while self.current_token() and self.current_token().type ==
            ↪ TokenType.OPERATOR and self.current_token().value in ['*'
            ↪ , '/']:
            operator = self.eat(TokenType.OPERATOR).value
            right = self.parse_factor()
            left = BinaryExpression(left, operator, right)
        return left

    def parse_factor(self):
        token = self.current_token()
        if token.type == TokenType.NUMBER:
            self.eat(TokenType.NUMBER)
            return NumberLiteral(token.value)
        elif token.type == TokenType.IDENTIFIER:
            self.eat(TokenType.IDENTIFIER)
            return Identifier(token.value)
```

```
        elif token.value == '(':
            self.eat(TokenType.OPERATOR)
            expr = self.parse_expression()
            self.eat(TokenType.OPERATOR)
            return expr
        else:
            raise SyntaxError("期望表达式")

    def parse_statement(self):
        token = self.current_token()
        if token.type == TokenType.KEYWORD and token.value == 'if':
            return self.parse_if_statement()
        elif token.type == TokenType.IDENTIFIER and self.pos + 1 < len(
            ↪ self.tokens) and self.tokens[self.pos+1].value == '=':
            return self.parse_assignment()
        else:
            raise SyntaxError("未知语句")

    def parse_if_statement(self):
        self.eat(TokenType.KEYWORD)
        self.eat(TokenType.OPERATOR)
        condition = self.parse_expression()
        self.eat(TokenType.OPERATOR)
        self.eat(TokenType.OPERATOR)
        then_branch = self.parse_statement()
        self.eat(TokenType.OPERATOR)
        else_branch = None
        if self.current_token() and self.current_token().type ==
            ↪ TokenType.KEYWORD and self.current_token().value == 'else
            ↪ ':
            self.eat(TokenType.KEYWORD)
            self.eat(TokenType.OPERATOR)
            else_branch = self.parse_statement()
            self.eat(TokenType.OPERATOR)
        return IfStatement(condition, then_branch, else_branch)

    def parse_assignment(self):
        identifier = self.eat(TokenType.IDENTIFIER).value
        self.eat(TokenType.OPERATOR)
        expression = self.parse_expression()
        return AssignmentStatement(identifier, expression)
```

```
    def parse_program(self):
        statements = []
        while self.current_token():
            statements.append(self.parse_statement())
        return statements
```

在这段代码中，我们定义了多种 AST 节点类来表示程序结构。Parser 类使用递归下降法解析 Token 流，通过 parse_expression、parse_term 和 parse_factor 函数确保运算符优先级正确。例如，乘除法在 parse_term 中处理，优先级高于加减法；parse_if_statement 函数解析条件语句，处理 if 关键字、条件表达式和分支语句。eat 方法用于验证和消耗 Token，提供基本错误检查。

解释器（Interpreter）负责执行 AST，它使用作用域来管理变量。作用域是一个字典，存储变量名和值的映射。我们编写 evaluate 函数，递归遍历 AST 节点并执行操作。例如：

```
class Scope:
    def __init__(self, parent=None):
        self.variables = {}
        self.parent = parent

    def get(self, name):
        if name in self.variables:
            return self.variables[name]
        elif self.parent:
            return self.parent.get(name)
        else:
            raise NameError(f"未定义变量␣:␣{name}")

    def set(self, name, value):
        self.variables[name] = value

def evaluate(node, scope):
    if isinstance(node, NumberLiteral):
        return node.value
    elif isinstance(node, Identifier):
        return scope.get(node.value)
    elif isinstance(node, BinaryExpression):
        left_val = evaluate(node.left, scope)
        right_val = evaluate(node.right, scope)
        if node.operator == '+':
            return left_val + right_val
        elif node.operator == '-':
            return left_val - right_val
        elif node.operator == '*':
```

```
            return left_val * right_val
        elif node.operator == '/':
            return left_val / right_val
        elif node.operator == '>':
            return left_val > right_val
        elif node.operator == '<':
            return left_val < right_val
        elif node.operator == '==':
            return left_val == right_val
        else:
            raise RuntimeError(f"未知运算符 : {node.operator}")
    elif isinstance(node, AssignmentStatement):
        value = evaluate(node.expression, scope)
        scope.set(node.identifier, value)
        return value
    elif isinstance(node, IfStatement):
        condition_val = evaluate(node.condition, scope)
        if condition_val:
            return evaluate(node.then_branch, scope)
        else:
            if node.else_branch:
                return evaluate(node.else_branch, scope)
            else:
                return None
    else:
        raise RuntimeError(f"未知节点类型 : {type(node)}")
```

evaluate 函数根据节点类型执行相应操作。对于 BinaryExpression，它递归计算左右子树的值，然后应用运算符；对于 AssignmentStatement，它计算表达式值并存入作用域；对于 IfStatement，它评估条件并决定执行哪个分支。作用域支持简单的嵌套，通过 parent 属性实现变量查找链。

最后，我们组装整个引擎，编写 run 函数串联所有步骤：

```
def run(source):
    tokens = lexer(source)
    parser = Parser(tokens)
    program = parser.parse_program()
    scope = Scope()
    for statement in program:
        result = evaluate(statement, scope)
    return scope
```

测试时，我们可以执行脚本如 `x = 10 + 2 * (3 - 1);  if (x > 15) { y = 1; }`

`else { y = 0; }`，并检查变量 x 和 y 的值。例如，计算 x 时，先计算 `3 - 1` 得 2，再乘以 2 得 4，然后加 10 得 14，由于 14 不大于 15，因此 y 被赋值为 0。

8 进阶与优化

在基本实现基础上，我们可以添加错误处理来提升用户体验。例如，在词法分析、语法分析和运行时阶段，记录行号和列号，提供详细的错误信息。这有助于开发者快速定位问题，例如在语法错误时指出具体位置。

性能优化方面，可以考虑编译到字节码。字节码是一种紧凑的中间表示，由虚拟机执行，比直接解释 AST 更高效。实现时，我们需要定义字节码指令集（如 LOAD_CONST、STORE_NAME、BINARY_ADD 等），并将 AST 编译成字节码序列。虚拟机使用栈来执行指令，例如对于表达式 `a + b`，字节码可能是 `LOAD a`、`LOAD b`、`ADD`，这减少了递归开销。

Just-In-Time（JIT）编译是更高级的优化，它将热点代码直接编译成本地机器码，大幅提升执行速度。但这涉及复杂的技术，如动态代码生成和优化，通常用于生产级引擎。

功能扩展上，我们可以添加循环语句（如 `while`）、函数定义与调用、内置函数（如 `print`）和复杂数据类型（如数组和对象）。这些扩展需要修改词法分析、语法分析和解释器，以支持新语法和语义。例如，添加函数调用时，需引入调用栈和作用域链。

通过本文的旅程，我们深入理解了动态脚本引擎的核心组件：词法分析将源代码转换为 Token 流，语法分析构建 AST，解释执行遍历 AST 实现功能。我们的微型引擎虽然简单，但揭示了真实引擎如 V8（JavaScript）、CPython 和 Lua 的基本原理。这些成熟引擎处理了更多边界情况，并集成了高级优化。

鼓励读者在此基础上继续探索，例如实现字节码虚拟机或添加新功能。进一步学习资源包括《编译原理》（龙书）和《Crafting Interpreters》书籍。现在，您已经拥有了理解更复杂脚本引擎的基石，去构建属于自己的「微缩宇宙」吧！

第 III 部

基本的队列（Queue）数据结构

李睿远

Nov 07, 2025

从“先来后到”的哲学，到计算机科学的核心基石

想象一下，您在咖啡店点单时，前面已经排了几位顾客，您会自然地站到队尾等待。同样，打印机处理多个文档任务时，会按照接收顺序依次打印；客服热线中，来电者也会被放入等待队列，按先后顺序接听。这些场景都遵循一个共同原则 —— 先进先出（First-In-First-Out，FIFO），即先来的对象先被服务。在计算机科学中，这种“排队”思想被抽象为队列数据结构，用于处理需要顺序执行的任务。队列是许多系统的基础组件，从操作系统调度到网络数据包管理，都离不开它的身影。本文将带您从零开始，深入探索队列的概念、实现方式及其广泛应用，帮助您不仅理解“是什么”，更掌握“为什么”和“怎么做”。

9 队列的核心概念与特性

队列是一种操作受限的线性表，它只允许在表的前端进行删除操作，在表的后端进行插入操作。前端通常称为队首（front），后端称为队尾（rear）。队列的核心特性是 FIFO，即最先进入队列的元素将最先被移出。这与栈的后进先出（LIFO）特性形成鲜明对比，例如栈像电梯，最后进入的人最先出去；而队列像隧道，车辆按进入顺序依次通过。

队列的基本操作包括入队、出队、获取队首元素、检查是否为空以及返回大小。入队操作（常用方法名如 enqueue 或 offer）用于向队列尾部添加新元素；出队操作（如 dequeue 或 poll）移除并返回队列头部的元素；获取队首元素操作（如 front 或 peek）返回头部元素但不移除它；检查是否为空操作（isEmpty）判断队列是否没有元素；返回大小操作（size）给出元素个数。对于有界队列，还可以添加检查是否已满的操作（isFull）。这些操作共同定义了队列的行为，确保数据处理的顺序性。

10 队列的实现方式

队列可以通过多种底层数据结构实现，最常见的是基于数组和基于链表的方式。每种方式都有其优缺点，适用于不同场景。下面我们将详细探讨这两种实现，并提供代码示例和解读。

10.1 基于数组的实现（顺序队列）

基于数组的队列使用一个固定大小的数组和两个指针（front 和 rear）来跟踪队列的头部和尾部。初始时，front 和 rear 都指向数组起始位置。入队操作将元素添加到 rear 指向的位置，并后移 rear；出队操作返回 front 指向的元素，并前移 front。然而，这种简单实现会遇到“假溢出”问题：随着元素入队和出队，front 和 rear 指针不断后移，导致数组前半部分空间无法利用，即使数组未满，也无法添加新元素。

解决假溢出的方法是使用循环队列。循环队列将数组视为一个环形结构，当指针移动到数组末尾时，自动绕回开头。关键计算包括：入队时，rear 更新为 $rear = (rear+1)\%capacity$；出队时，front 更新为 $front = (front + 1)\%capacity$。队列空的条件是 front 等于 rear；队列满的条件是 $(rear + 1)\%capacity == front$，这里牺牲一个存储单元以区分空和满状态。

以下是用 Python 实现循环队列的代码示例。我们定义一个 ArrayQueue 类，包含 items 数组、front 和 rear 指针以及容量 capacity。

```
class ArrayQueue:
```

```
    def __init__(self, capacity):
        self.capacity = capacity
        self.items = [None] * capacity
        self.front = 0
        self.rear = 0

    def enqueue(self, item):
        if self.is_full():
            raise Exception("Queue is full")
        self.items[self.rear] = item
        self.rear = (self.rear + 1) % self.capacity

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.items[self.front]
        self.front = (self.front + 1) % self.capacity
        return item

    def peek(self):
        if self.is_empty():
            return None
        return self.items[self.front]

    def is_empty(self):
        return self.front == self.rear

    def is_full(self):
        return (self.rear + 1) % self.capacity == self.front

    def size(self):
        return (self.rear - self.front + self.capacity) % self.capacity
```

在这段代码中，构造函数初始化一个固定容量的数组，并将 front 和 rear 都设为 0。enqueue 方法首先检查队列是否已满，如果未满，则将元素放入 rear 位置，并更新 rear 指针使用模运算实现循环。dequeue 方法检查队列是否为空，如果不空，则返回 front 位置的元素，并更新 front 指针。peek 方法返回队首元素但不移除它。is_empty 和 is_full 方法分别通过比较 front 和 rear 来判断状态。size 方法计算当前元素个数，考虑循环情况。这种实现确保了所有基本操作的时间复杂度为 O(1)，但容量固定，可能不适合动态场景。

10.2 基于链表的实现（链式队列）

基于链表的队列使用节点来存储元素，每个节点包含数据域和指向下一个节点的指针。队列维护两个指针：head 指向链表头节点（队首），tail 指向链表尾节点（队尾）。入队操作在 tail 节点后添加新节点，并更新 tail；出队操作移除 head 节点，并更新 head。链表实现天然支持动态扩容，没有假溢出问题，因为节点可以随时分配。

以下是用 Python 实现链式队列的代码示例。首先定义 Node 类表示链表节点，然后定义 LinkedListQueue 类。

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedListQueue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0

    def enqueue(self, item):
        new_node = Node(item)
        if self.is_empty():
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.count += 1

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.head.data
        self.head = self.head.next
        if self.head is None:
            self.tail = None
        self.count -= 1
        return item

    def peek(self):
        if self.is_empty():
```

```
        return None
35      return self.head.data

37  def is_empty(self):
        return self.head is None
39
    def size(self):
41      return self.count
```

在这段代码中，Node 类包含 data 和 next 指针。LinkedListQueue 的构造函数初始化 head 和 tail 为 None，count 记录元素个数。enqueue 方法创建新节点，如果队列为空，则 head 和 tail 都指向新节点；否则将新节点链接到 tail 后，并更新 tail。dequeue 方法检查队列是否为空，如果不空，则返回 head 的数据，更新 head 到下一个节点，如果 head 变为 None，则 tail 也设为 None。peek 方法返回 head 的数据但不移除。is_empty 和 size 方法分别通过 head 和 count 判断状态。链表实现的所有操作时间复杂度也为 O(1)，且容量无限，但每个节点有额外指针开销。

10.3 两种实现方式的对比

基于数组和基于链表的队列实现各有优劣。在时间复杂度上，两种实现的入队、出队操作均为 O(1)，因为都只涉及指针更新。在空间复杂度上，数组实现使用固定容量，内存连续，访问效率高，但可能浪费空间或需要扩容；链表实现容量动态，无浪费，但每个节点有额外指针开销，内存不连续。数组实现适合已知最大大小的场景，例如嵌入式系统；链表实现适合大小变化频繁的应用，如任务调度。选择时需权衡内存使用和性能需求。

11 队列的变体与应用场景

队列不仅限于基本 FIFO 形式，还有多种变体适应不同需求。双端队列（Deque）允许在队列两端进行插入和删除操作，可以同时模拟栈和队列的行为，例如用于实现滑动窗口算法。优先队列（Priority Queue）出队顺序由元素优先级决定，而非入队顺序，通常用堆（Heap）实现，应用在任务调度和 Dijkstra 算法中，其中高优先级任务先处理。
阻塞队列（Blocking Queue）是一种同步工具，当队列为空时，获取操作阻塞线程直到有元素可用；当队列满时，插入操作阻塞直到空间空闲。这在生产者-消费者模型中非常有用，例如多线程环境下，生产者线程生成数据放入队列，消费者线程从队列取出数据，阻塞机制确保线程安全协调。这些变体扩展了队列的应用范围，使其成为操作系统、消息中间件和网络通信的核心组件。

12 实战案例：用队列解决“击鼓传花”游戏

“击鼓传花”游戏是一个经典问题，可以用队列优雅解决。问题描述：一群人围成一圈，传花同时计数，数到特定数字的人被淘汰，最后剩下的人获胜。解决方案利用队列的 FIFO 特性模拟传花过程。
首先，将所有玩家入队。然后循环模拟传花：将队首玩家出队并立刻入队，相当于安全传花

一次；当计数达到指定值时，将当前队首玩家出队（淘汰），不再入队。重复直到队列只剩一人。

以下是用之前实现的 LinkedListQueue 解决此问题的代码示例。假设玩家列表为 [Alice, Bob, Charlie, Diana]，计数为 3。

```
def hot_potato(players, num):
    queue = LinkedListQueue()
    for player in players:
        queue.enqueue(player)
    while queue.size() > 1:
        for _ in range(num - 1):
            queue.enqueue(queue.dequeue())
        eliminated = queue.dequeue()
        print(f"淘汰␣:␣{eliminated}")
    return queue.dequeue()

winner = hot_potato(["Alice", "Bob", "Charlie", "Diana"], 3)
print(f"获胜者␣:␣{winner}")
```

在这段代码中，hot_potato 函数初始化队列并填入所有玩家。while 循环继续直到队列大小大于 1。内层 for 循环执行 num-1 次传花：每次将队首玩家出队并立刻入队，相当于传递花束。当内循环结束，当前队首玩家被淘汰（出队并不再入队）。最后返回剩余的获胜者。例如，初始队列为 Alice、Bob、Charlie、Diana，计数 3；第一次循环传递两次后，Charlie 被淘汰；重复过程直到剩一人。这展示了队列如何自然建模顺序处理问题。

队列作为一种基础数据结构，其 FIFO 特性确保了数据处理的公平性和顺序性。我们从核心概念出发，探讨了基于数组和链表的实现方式，数组实现通过循环队列解决假溢出，链表实现支持动态扩容。队列的变体如双端队列、优先队列和阻塞队列，扩展了其应用场景，从算法到系统设计无处不在。通过击鼓传花案例，我们看到了队列在解决实际问题中的实用性。掌握队列不仅有助于理解计算机科学基础，还能为构建高效系统奠定基石。

13　互动与思考题

思考题一：如何用栈来实现一个队列？这需要两个栈协作，一个用于入队，一个用于出队，通过元素转移模拟 FIFO 行为。思考题二：除了文中提到的应用，队列还常见于消息队列系统如 RabbitMQ、事件循环处理等场景。欢迎在评论区分享您的想法和代码实现，共同探讨数据结构的魅力。

第 IV 部

基本的 C++ 移动语义（Move Semantics）

黄京

Nov 08, 2025

告别不必要的拷贝，拥抱高效资源转移。在 C++ 编程中，对象拷贝是常见操作，但深拷贝可能带来显著的性能开销。移动语义作为 C++11 引入的重要特性，旨在通过资源转移而非复制来提升效率。本文将系统性地介绍移动语义的核心概念、实现方式及最佳实践，帮助读者从基础到深入掌握这一技术。

在 C++ 中，对象拷贝操作常常涉及深拷贝，这在处理动态资源时效率低下。以一个简单的 MyString 类为例，该类包含一个动态分配的字符数组。当传递或返回这种对象时，深拷贝构造函数和赋值运算符会重新分配内存并复制所有数据，导致不必要的性能损失。例如，如果一个临时 MyString 对象即将被销毁，我们是否真的需要重新分配内存并复制其内容？移动语义应运而生，其核心思想是“窃取”临时对象的资源，而非执行昂贵的拷贝操作。这种机制在标准模板库（STL）容器如 std::vector 和 std::string 中广泛应用，显著提升了性能。

14 基石：左值、右值与将亡值

理解移动语义前，必须掌握 C++ 中的值类别。左值是有标识符、可以取地址的表达式，例如变量或函数返回的左值引用。右值通常是字面量、临时对象或表达式求值的中间结果，传统上不能被赋值或取地址。C++11 进一步细化了值类别，引入了将亡值，指即将被销毁的对象，它们是移动语义操作的最佳候选人。值类别可概括为：泛左值包括左值和将亡值，而将亡值又属于纯右值。这种分类帮助我们识别哪些对象适合进行资源转移。

15 关键工具：右值引用 T&&

右值引用是移动语义的语法基础，使用 && 声明，只能绑定到右值（包括将亡值）。其核心作用是延长将亡值的生命周期并允许修改。例如，int&& rref = 42 + 100; 是合法的，因为它绑定到一个临时表达式结果；而 int a = 10; int&& rref2 = a; 会编译错误，因为不能将右值引用绑定到左值。与左值引用 T&（仅绑定左值）和常左值引用 const T&（可绑定左右值但不允许修改）相比，右值引用专为资源转移设计，为移动操作提供了类型安全的基础。

16 实现移动语义：移动构造函数与移动赋值运算符

移动语义通过移动构造函数和移动赋值运算符实现。移动构造函数 MyClass(MyClass&& other) noexcept 的目标是从 other 对象“窃取”资源，并将 other 置于一个有效但可析构的状态。实现步骤包括将当前对象的指针指向 other 的资源，并将 other 的指针置为 nullptr，以确保 other 析构时不会释放已被转移的资源。以下是一个 MyString 类的移动构造函数示例：

```
class MyString {
public:
    // 移动构造函数
    MyString(MyString&& other) noexcept : data_(other.data_), size_(
        ↪ other.size_) {
        other.data_ = nullptr;
```

```
        other.size_ = 0;
    }
private:
    char* data_;
    size_t size_;
};
```

在这个示例中，data_ 和 size_ 被初始化为 other 的值，然后 other 的成员被重置为默认状态，从而安全地转移资源。移动操作通常应标记为 noexcept，因为标准库容器如 std::vector 在重新分配时会优先使用 noexcept 移动操作，否则回退到拷贝，影响性能。移动赋值运算符 MyClass& operator=(MyClass&& other) noexcept 的目标是释放当前对象的资源，并从 other “窃取”资源。实现步骤包括检查自赋值、释放当前资源、转移 other 的资源，并将 other 置为空状态。以下是 MyString 类的移动赋值运算符示例：

```
class MyString {
public:
    // 移动赋值运算符
    MyString& operator=(MyString&& other) noexcept {
        if (this != &other) {
            delete[] data_;
            data_ = other.data_;
            size_ = other.size_;
            other.data_ = nullptr;
            other.size_ = 0;
        }
        return *this;
    }
private:
    char* data_;
    size_t size_;
};
```

此代码首先检查自赋值，避免资源泄漏，然后释放当前内存，转移指针，并重置 other 状态。被移动后的对象必须处于“有效但可析构”状态，通常意味着成员变量被设置为空或默认值，例如 nullptr 或 0，从而确保可以安全析构或重新赋值。

17 催化剂：std::move - 将左值转化为右值

std::move 是一个类型转换工具，本质上是 static_cast<T&&>，它无条件地将参数转换为右值引用，从而启用移动语义。它本身不执行任何移动操作，而是作为启动移动的“开关”。使用场景包括当我们明确知道一个左值不再被需要时，例如 MyString s1 = std::move(s2); 会调用移动构造函数而非拷贝构造函数。但需注意，被 std::move 后的对象不应再被使用（除非析构或重新赋值），且不要对 const 对象使用 std::move，因为

它会阻止移动语义的发生，导致匹配到拷贝操作。

18 综合实践：一个完整的 MyVector 类示例

为了巩固理解，我们实现一个简单的 MyVector 类，展示拷贝语义与移动语义的差异。类定义包括构造函数、析构函数、拷贝构造/赋值和移动构造/赋值。以下是完整代码：

```
class MyVector {
public:
   // 构造函数
   MyVector(size_t size = 0) : data_(new int[size]), size_(size) {}
   // 析构函数
   ~MyVector() { delete[] data_; }
   // 拷贝构造函数
   MyVector(const MyVector& other) : data_(new int[other.size_]),
       ↪ size_(other.size_) {
      std::copy(other.data_, other.data_ + size_, data_);
   }
   // 拷贝赋值运算符
   MyVector& operator=(const MyVector& other) {
      if (this != &other) {
         delete[] data_;
         data_ = new int[other.size_];
         size_ = other.size_;
         std::copy(other.data_, other.data_ + size_, data_);
      }
      return *this;
   }
   // 移动构造函数
   MyVector(MyVector&& other) noexcept : data_(other.data_), size_(
       ↪ other.size_) {
      other.data_ = nullptr;
      other.size_ = 0;
   }
   // 移动赋值运算符
   MyVector& operator=(MyVector&& other) noexcept {
      if (this != &other) {
         delete[] data_;
         data_ = other.data_;
         size_ = other.size_;
         other.data_ = nullptr;
         other.size_ = 0;
```

```
        }
35      return *this;
    }
37  private:
    int* data_;
39  size_t size_;
};
```

在拷贝操作中，我们新分配内存并复制所有元素；而在移动操作中，我们直接转移指针和大小信息，并将源对象置空。在 main 函数中，通过返回局部 MyVector 或将其 push_back 到另一个容器，可以观察到移动语义带来的性能优势，例如避免不必要的内存分配和复制。移动语义通过资源窃取避免了不必要的深拷贝，提升了 C++ 程序的效率。核心要点包括：右值引用 && 是语法基础，移动构造函数和移动赋值运算符是具体实现，std::move 是启用移动的开关。编译器在特定条件下会自动生成移动操作，例如如果一个类没有用户声明的拷贝操作、移动操作和析构函数。最佳实践遵循 Rule of Five：如果声明了析构函数或拷贝操作之一，最好同时声明所有五个特殊成员函数（两个拷贝、两个移动、一个析构）。移动操作应标记为 noexcept，并明智地使用 std::move，同时理解被移动后对象的状态。

19 进一步阅读

为进一步深入学习，可探索 C++ 标准库中的完美转发 std::forward，它结合通用引用实现参数转发；引用折叠规则，解释了模板中引用的处理方式；以及智能指针如 std::unique_ptr 和 std::shared_ptr 的移动语义应用，这些主题将帮助读者更全面地掌握现代 C++ 资源管理技术。

第 V 部

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。

20 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。

21 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。

22　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。