c13n #35

c13n

2025 年 11 月 19 日

第 I 部

深入理解并实现基本的管道操作符（Pipe Operator）原理与实现

黄梓淳

Oct 08, 2025

在 JavaScript 开发中，处理链式数据转换是一个常见场景，但往往伴随着代码可读性差和维护成本高的问题。例如，当我们需要对数据进行多次函数调用时，传统写法如 `func3(func2(func1(data)))` 会导致深层嵌套，从内到外的阅读顺序不符合人类直觉，同时使用临时变量存储中间结果也会增加代码复杂度。为了解决这些痛点，函数式编程中的管道（Pipe）概念提供了一种优雅方案，它允许数据像在流水线上一样依次通过处理函数，从而提升代码的清晰度。这种模式在 F#、Elm 和 RxJS 等语言和库中已经得到广泛应用。本文旨在深入解析管道操作符的核心原理，并引导读者用纯 JavaScript 从头实现一个功能完善的管道工具函数，涵盖从基础到进阶的完整知识体系。

1 什么是管道操作符？—— 一种数据流的思想

管道操作符的核心思想是将数据视为在管道中流动的实体，依次通过一系列处理函数进行转换。具体来说，它接受一个初始数据作为输入，然后将其传递给第一个函数处理，再将结果传递给下一个函数，如此反复，直到所有函数执行完毕，输出最终结果。这种模式强调数据的单向流动，类似于工厂中的流水线作业，其中每个函数代表一个加工工序。例如，在传统嵌套写法中，代码 `func3(func2(func1(data)))` 需要从内向外阅读，而管道式写法如 `pipe(func1, func2, func3)(data)` 或使用提案中的 `data |> func1 |> func2 |> func3` 则允许从左向右线性阅读，更符合人类的思维习惯。一个简单的生活化比喻是，将原材料放入加工流水线，经过多道工序后变成成品，管道操作符正是这种思想的代码体现。

2 为什么我们需要管道？—— 提升代码的可读性与可维护性

管道操作符能显著提升代码质量，主要体现在几个方面。首先，它促进声明式编程风格，让代码更专注于表达“做什么”而非“怎么做”，从而减少实现细节的干扰。其次，管道能消除临时变量，避免为中间状态命名的负担，使代码更简洁易读。例如，在复杂的数据处理链中，无需定义多个变量来存储每一步的结果。此外，管道是函数组合的优雅实践，它鼓励开发者将逻辑拆分为小而纯的函数，每个函数只负责单一职责，这不仅便于复用，还增强了代码的模块性。最后，管道模式使调试和测试变得更简单，因为每个步骤都是独立的函数，可以单独验证其行为，而不必关注整个链的上下文。

3 核心原理剖析：pipe 函数是如何工作的？

要理解管道操作符的实现，首先需要分析其函数签名和执行过程。典型的 `pipe` 函数签名是 `const pipe = (...fns) => (initialValue) => { ... }`，其中 `...fns` 使用剩余参数语法接收一个函数列表，而返回的函数接受初始值 `initialValue`。执行过程可以描述为：从第一个函数开始，将 `initialValue` 传入，然后将上一个函数的返回值作为下一个函数的输入值，依次迭代所有函数，最终返回最后一个函数的执行结果。这里的关键在于数据传递的连续性，前一个函数的输出必须与后一个函数的输入类型兼容，且每个函数最好是“一元函数”（即只接受一个参数），这符合函数式编程的最佳实践，能确保数据流的可预测性和简洁性。从数学角度看，管道操作符实现了函数的左结合，类似于函数组合 $f \circ g \circ h$ 但顺序更直观。

4　动手实现：从零构建我们的 pipe 函数

我们将分步骤实现管道函数，从基础版开始，逐步增强其功能以处理错误和异步场景。基础版实现使用 Array.prototype.reduce 方法，这是一种简洁的函数式编程方式。代码如下：

```
const pipe = (...fns) => (value) => fns.reduce((acc, fn) => fn(acc),
    ↪ value);
```

这段代码的核心是 reduce 方法，它遍历函数列表 fns，以 value 作为初始累积值 acc，然后对每个函数 fn 应用 fn(acc)，并将结果更新为新的累积值。这样，数据就像在管道中流动一样，依次通过每个函数。例如，假设我们有三个函数：addPrefix 用于添加前缀，toUpperCase 用于转换为大写，addExclamation 用于添加感叹号。通过 pipe 组合后，调用 processName('world') 会依次执行这些函数，最终输出 HELLO, WORLD!。这种实现的优点在于其简洁性和函数式思想的体现，但它假设所有函数都是同步且不会抛出错误，因此在生产环境中可能需要扩展。

接下来，我们考虑增强版实现，添加错误处理与异步支持。在基础版中，如果某个函数抛出错误，整个管道会中断，且没有捕获机制。我们可以使用 try...catch 包裹 reduce 逻辑来改进：

```
const pipeWithErrorHandling = (...fns) => (value) => {
  try {
    return fns.reduce((acc, fn) => fn(acc), value);
  } catch (error) {
    console.error('管道执行错误 :', error);
    throw error;
  }
};
```

这个版本在 reduce 外部添加了 try...catch 块，当任何函数抛出异常时，会捕获并记录错误，然后重新抛出以保持调用方感知。但更常见的需求是处理异步函数，例如那些返回 Promise 的操作。我们可以实现一个异步版本的 pipeAsync，使用 Promise 链来确保顺序执行：

```
const pipeAsync = (...fns) => (initialValue) =>
  fns.reduce((promise, fn) => promise.then(fn), Promise.resolve(
      ↪ initialValue));
```

这里，Promise.resolve(initialValue) 将初始值转换为 Promise，然后通过 reduce 和 promise.then(fn) 链式调用每个函数。每个 fn 都返回一个 Promise，确保异步操作按顺序进行。例如，在数据获取场景中，fetchData 异步获取数据，parseJSON 解析响应，processData 处理结果，通过 pipeAsync 组合后，调用 getProcessedData('/api/data') 会依次执行这些异步步骤，最终输出处理后的数据。这种实现自动处理了 Promise 链，使异步代码更清晰。

5 进阶话题与扩展

在深入管道操作符后，有必要探讨其与相关概念的差异和应用。首先，pipe 与 compose 都是函数组合工具，但执行顺序相反。pipe(f, g, h)(x) 按从左到右顺序执行，相当于数学上的 $h(g(f(x)))$，而 compose(f, g, h)(x) 按从右到左顺序执行，相当于 $f(g(h(x)))$。compose 的实现可以使用 reduceRight：const compose = (...fns) => (value) => fns.reduceRight((acc, fn) => fn(acc), value);。这种区别源于函数组合的数学定义，但在实际编程中，pipe 的更直观顺序使其更受欢迎。其次，管道在流行库中广泛应用，例如 RxJS 的 pipe 方法用于组合观察者操作符，而 Lodash 的函数式编程版本提供了 _.flow 函数，其行为等同于 pipe。这些库的集成展示了管道模式在复杂数据流处理中的价值。最后，JavaScript 语言本身也在演进，TC39 提案中的原生管道操作符 ↑ 旨在提供语法级支持，例如 data ↑ func1 ↑ func2 ↑ func3，这将进一步简化代码书写，但目前仍处于提案阶段，需要关注其进展。

管道操作符通过将数据流线性化，显著提升了代码的可读性和可维护性，是函数式编程中的核心模式之一。本文从原理剖析到实践实现，详细讲解了如何用 JavaScript 构建管道函数，包括基础版、错误处理版和异步版。实现精髓在于利用 reduce 方法迭代函数列表并传递数据，这体现了函数组合的优雅性。我们鼓励读者在项目中尝试这种模式，从小函数和管道组合开始，体验声明式编程的优势。通过将复杂逻辑拆分为可测试的单元，管道不仅能减少代码冗余，还能促进更健壮的软件设计。

第 II 部

深入理解并实现基本的深度优先搜索（DFS）算法

杨岢瑞

Oct 10, 2025

想象你身处一个巨大的迷宫，没有地图，你该如何系统地探索每一条路径，确保不重不漏？一种最直观的策略便是“一条路走到黑”。遇到岔路时，选择一条路走到底，直到死胡同，然后返回上一个岔路口选择另一条路。这种“不撞南墙不回头”的策略，正是我们今天要深入探讨的深度优先搜索（Depth-First Search，DFS）算法的精髓。本文将带你从本质理解 DFS 的思想，掌握其递归与非递归两种实现方式，并通过经典问题学会如何应用它，从而在算法世界中游刃有余。

6 DFS 核心思想解析

深度优先搜索的核心思想可以用一个形象化的比喻来概括：“一条路走到黑”加上“后悔机制”（即回溯）。具体来说，DFS 在探索图或树结构时，会优先沿着一条路径深入到底，直到无法继续前进，然后回溯到上一个节点，尝试其他未探索的分支。这种策略依赖于栈（Stack）数据结构，无论是隐式的系统调用栈（在递归中）还是显式维护的栈（在迭代中），都利用了栈的“后进先出”特性来实现回溯过程。

DFS 的核心操作可以分解为几个关键步骤。首先，访问当前节点，例如打印节点信息或判断是否达到目标。其次，标记当前节点为已访问，这是防止程序在存在环的图中陷入无限循环的关键步骤。然后，对于当前节点的每一个未被访问的邻居节点，重复上述访问和标记过程，实现深度探索。最后，当所有邻居都被探索完毕时，回溯到上一个节点，继续探索其他分支。整个过程就像在迷宫中系统地尝试每一条路径，确保不会遗漏任何可能性。

7 DFS 的两种实现方式

7.1 递归实现

递归实现是 DFS 最直观和常用的方式，它利用函数调用栈来隐式地管理回溯过程。以下是一个基于图的邻接表表示的递归 DFS 实现框架，使用 Python 风格的伪代码。

```
def dfs_recursive(node, visited):
    # 标记当前节点为已访问
    visited[node] = True
    # 处理当前节点，例如打印节点信息
    print(f"Visiting␣node␣{node}")
    # 遍历当前节点的所有邻居
    for neighbor in graph[node]:
        # 如果邻居未被访问，递归调用 DFS
        if not visited[neighbor]:
            dfs_recursive(neighbor, visited)
```

在这段代码中，`visited` 数组用于记录每个节点的访问状态，防止重复访问。递归调用 `dfs_recursive` 实现了深度优先的探索：每次函数调用都会处理当前节点，然后对其未访问的邻居递归调用自身，从而沿着一条路径深入到底。递归的终止条件隐含在循环中：当所有邻居都已被访问时，函数会自然返回，实现回溯。这种实现方式代码简洁，但需要注意递归深度过大时可能导致栈溢出。

7.2 迭代实现

迭代实现通过显式使用栈来模拟递归过程，避免了递归可能带来的栈溢出问题，并提供了更好的控制力。以下是迭代 DFS 的实现框架。

```
def dfs_iterative(start_node):
    # 初始化栈和已访问集合
    stack = []
    visited = set()
    # 将起始节点压入栈中
    stack.append(start_node)
    while stack: # 当栈不为空时循环
        # 弹出栈顶元素
        node = stack.pop()
        if node not in visited:
            # 标记节点为已访问
            visited.add(node)
            # 处理当前节点
            print(f"Visiting node {node}")
            # 将未访问的邻居逆序压入栈中
            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)
```

在迭代实现中，栈用于存储待访问的节点，visited 集合记录已访问节点。关键点在于，我们在弹出节点后才检查其访问状态，这是因为同一节点可能被多次压入栈中（例如通过不同路径）。逆序压入邻居节点是为了保持与递归实现一致的遍历顺序，否则由于栈的后进先出特性，遍历顺序可能会颠倒。迭代实现虽然代码稍复杂，但能有效控制内存使用，适用于深度极大的场景。

递归实现和迭代实现在可读性、性能和适用场景上各有优劣。递归实现代码简洁，更符合 DFS 的思维模型，但在深度过大时可能引发栈溢出。迭代实现通过显式栈避免了这一问题，且便于控制执行流程，但代码可读性稍低。在实际应用中，如果图深度可控，递归实现是首选；对于深度极大或需要精细控制栈的场景，迭代实现更为合适。无论哪种方式，核心都是利用栈的后进先出特性来实现深度优先和回溯。

8 DFS 的实战应用

8.1 应用一：二叉树的前序遍历

二叉树的前序遍历是 DFS 在树结构上的直接体现，访问顺序为“根节点 -> 左子树 -> 右子树”。以下分别用递归和迭代实现前序遍历。

递归实现代码如下：

```
def preorder_recursive(root):
```

```
    if root is None:
        return
    # 访问根节点
    print(root.val)
    # 递归遍历左子树
    preorder_recursive(root.left)
    # 递归遍历右子树
    preorder_recursive(root.right)
```

在这段代码中，我们首先处理当前节点（根节点），然后递归处理左子树和右子树，这正符合 DFS 的深度优先思想。递归调用栈确保了回溯的正确性。

迭代实现代码如下：

```
def preorder_iterative(root):
    if root is None:
        return
    stack = [root]
    while stack:
        node = stack.pop()
        # 访问当前节点
        print(node.val)
        # 先将右子节点压栈，再将左子节点压栈，以确保左子树先被处理
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
```

迭代实现使用栈来模拟递归过程。由于栈的后进先出特性，我们先将右子节点压栈，再将左子节点压栈，这样左子节点会先被弹出和处理，实现了前序遍历的顺序。这种方法展示了 DFS 如何通过栈管理遍历路径。

8.2 应用二：查找路径

在图中查找从节点 A 到节点 B 的路径是 DFS 的经典应用。我们可以通过 DFS 探索所有可能路径，并记录访问顺序。以下是一个查找路径的递归实现示例。

```
def find_path_dfs(start, target, visited, path):
    # 标记当前节点为已访问
    visited[start] = True
    # 将当前节点加入路径
    path.append(start)
    if start == target:
        # 找到目标，返回路径
        return path.copy()
```

```
    # 遍历邻居节点
    for neighbor in graph[start]:
        if not visited[neighbor]:
            # 递归搜索
            result = find_path_dfs(neighbor, target, visited, path)
            if result:
                return result
    # 回溯：从路径中移除当前节点
    path.pop()
    return None
```

在这个实现中，我们使用 visited 数组避免重复访问，path 列表记录当前路径。当找到目标节点时，返回路径副本；否则，在回溯时从路径中移除当前节点。这体现了 DFS 的回溯机制，但注意 DFS 找到的路径不一定是最短路径，因为它优先深度探索。

9 DFS 的复杂度分析与常见陷阱

深度优先搜索的时间复杂度通常为 $O(V+E)$，其中 V 是顶点数，E 是边数。这是因为每个节点和边最多被访问一次。空间复杂度主要取决于 visited 数据结构和栈的使用，最坏情况下为 $O(V)$，例如当图呈链状结构时，栈可能需要存储所有节点。

在实际实现中，常见的陷阱包括忘记标记节点为已访问，这会导致在存在环的图中陷入无限循环；在迭代实现中，在错误的位置标记已访问可能导致节点被重复处理；此外，混淆 DFS 与广度优先搜索（BFS）的适用场景也是一个常见错误，例如 DFS 不适合求解非加权图的最短路径问题。为了避免这些陷阱，务必确保在访问节点时立即标记，并根据问题特性选择合适的算法。

深度优先搜索的核心在于其“深度优先”和“回溯”思想，以及栈数据结构的巧妙运用。DFS 的优点包括实现简单、空间效率较高，并且是许多高级算法（如回溯和 Tarjan 算法）的基础；缺点则是可能无法找到最优解（如最短路径），且递归实现有深度限制。

进一步思考，DFS 与回溯算法密切相关，回溯本质上是 DFS 加上剪枝优化。与 BFS 相比，DFS 更适用于探索所有可能解或路径存在的场景，而 BFS 则擅长寻找最短路径。鼓励读者动手实现代码，并尝试应用 DFS 解决更复杂的问题，如数独或 N 皇后问题，从而深化对算法的理解。通过不断实践，你将能灵活运用 DFS 应对各种挑战。

第 III 部

深入理解并实现基本的基数排序（Radix Sort）算法

叶家炜

Oct 11, 2025

在计算机科学中，排序算法是基础且重要的研究主题。想象一下，当我们面对一组数字，例如学生的学号「102」、「031」、「215」、「123」，如何高效地对它们进行排序？传统的基于比较的排序算法，如快速排序或归并排序，虽然通用性强，但其时间复杂度下界为 $O(n \log n)$。是否存在一种方法能够突破这一限制？答案是肯定的，基数排序（Radix Sort）作为一种非比较型整数排序算法，在某些条件下可以达到 $O(n \times k)$ 的线性时间复杂度，其中 k 是数字的最大位数。本文将带领读者深入理解基数排序的核心思想、完整流程、代码实现，并分析其优劣与应用场景。

10　为什么我们需要基数排序?

排序问题在现实世界中无处不在，从数据库查询到大数据处理，高效排序至关重要。基于比较的排序算法通过元素间的直接比较来确定顺序，但其性能受限于 $O(n \log n)$ 的理论下界。当数据量巨大时，这一限制可能成为瓶颈。基数排序则另辟蹊径，它不依赖于元素间的直接比较，而是利用数字的位结构进行排序，从而在特定场景下实现线性时间复杂度。例如，对于整数数据集，尤其是当数字范围相对集中且位数较小时，基数排序的性能优势显著。这不仅提升了效率，还拓宽了排序算法的应用边界。

基数排序的核心价值在于其非比较特性。它通过逐位处理元素，将排序问题分解为多个稳定的子排序过程。这种方法的灵感来源于日常生活中的分类逻辑，例如在整理文件时先按类别再按日期排序。基数排序正是将这种分层思想应用于数字排序，从而避免了直接比较的整体复杂度。理解基数排序不仅有助于掌握一种高效算法，还能深化对数据结构和算法设计的认识。

11　什么是基数排序?

基数排序中的「基数」指的是数字的进制基数。在十进制系统中，基数为 10，每一位数字的取值范围是 0 到 9。基数排序的基本思想是将待排序的整数视为由多个位组成的序列，然后从最低位到最高位依次对每一位进行排序。每一轮排序必须使用稳定的排序算法，以确保在后续排序中低位顺序不被破坏。

稳定性是基数排序成功的关键。例如，假设有两个数字 23 和 25，在对十位排序时，如果使用非稳定排序，可能会破坏个位已排好的顺序。稳定排序保证了相同高位的元素其低位的相对顺序不变。这类似于扑克牌排序：先按花色排序，再按数字排序，第二次排序时必须确保相同数字的牌其花色的顺序保持不变。基数排序通过这种逐位稳定的方式，最终实现整体有序。

基数排序适用于整数或可分解为位序列的数据类型。其核心在于将复杂排序问题简化为多个简单的子问题，每一子问题仅处理一个位上的数字。这种分解不仅降低了时间复杂度，还使得算法易于理解和实现。需要注意的是，基数排序通常从最低位开始（LSD，Least Significant Digit），但也可以从最高位开始（MSD，Most Significant Digit），后者适用于递归实现。

12 算法流程详解：一步步拆解基数排序

基数排序的流程可以分为几个关键步骤。首先，需要确定待排序数组中的最大值，以计算最大位数 d。例如，对于数组 [170，45，75，90，2，802，24，66]，最大值为 802，其位数为 3，因此需要进行三轮排序。接下来，初始化一个过程数组用于存放中间结果。然后，从最低位（个位）开始，到最高位（百位）结束，依次执行分配和收集操作。

分配阶段，我们根据当前位的数字将元素分配到对应的桶中。桶的数量等于基数，在十进制中为 10 个（0 到 9）。收集阶段，则按照桶的顺序（从 0 到 9）将元素依次取出，放回过程数组。以数组 [170，45，75，90，2，802，24，66] 为例，第一轮按个位排序：170 的个位是 0，45 的个位是 5，75 的个位是 5，90 的个位是 0，2 的个位是 2，802 的个位是 2，24 的个位是 4，66 的个位是 6。分配后，桶 0 包含 170 和 90，桶 2 包含 2 和 802，桶 4 包含 24，桶 5 包含 45 和 75，桶 6 包含 66。收集后数组变为 [170，90，2，802，24，45，75，66]。

第二轮按十位排序：170 的十位是 7，90 的十位是 9，2 的十位是 0（视为 0），802 的十位是 0，24 的十位是 2，45 的十位是 4，75 的十位是 7，66 的十位是 6。分配后，桶 0 包含 2 和 802，桶 2 包含 24，桶 4 包含 45，桶 6 包含 66，桶 7 包含 170 和 75，桶 9 包含 90。收集后数组变为 [802，2，24，45，66，170，75，90]。第三轮按百位排序：802 的百位是 8，2 的百位是 0，24 的百位是 0，45 的百位是 0，66 的百位是 0，170 的百位是 1，75 的百位是 0，90 的百位是 0。分配后，桶 0 包含 2、24、45、66、75、90，桶 1 包含 170，桶 8 包含 802。收集后最终数组为 [2，24，45，66，75，90，170，802]，排序完成。

13 关键实现：使用计数排序作为子程序

在基数排序中，每一轮的排序子程序必须稳定且高效。计数排序（Counting Sort）是理想选择，因为当数据范围较小（0 到 9）时，其时间复杂度为 $O(n+k)$，且具有稳定性。计数排序通过统计每个数字的出现次数，并利用前缀和确定元素位置，从而保证排序的稳定性。以下是基数排序的 Python 实现，附有详细注释。代码包括主函数 `radix_sort` 和子函数 `counting_sort_for_radix`，后者用于对特定位进行排序。

```
def radix_sort(arr):
    # 查找数组中的最大值，以确定最大位数
    max_val = max(arr)
    exp = 1 # 从个位开始
    # 循环直到处理完最高位
    while max_val // exp > 0:
        counting_sort_for_radix(arr, exp)
        exp *= 10 # 移动到下一位（十位、百位等）

def counting_sort_for_radix(arr, exp):
    n = len(arr)
    output = [0] * n # 输出数组，用于存放排序结果
```

```
    count = [0] * 10 # 计数数组，索引 0 到 9，对应当前位的数字
    # 统计当前位上每个数字的出现次数
    for i in range(n):
        index = (arr[i] // exp) % 10 # 提取当前位的数字
        count[index] += 1
    # 将计数数组转换为前缀和数组，以确定每个数字在输出数组中的最终位置
    for i in range(1, 10):
        count[i] += count[i - 1]
    # 逆向遍历原数组，将元素放置到输出数组的正确位置，保证稳定性
    for i in range(n - 1, -1, -1):
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i] # 根据计数数组放置元素
        count[index] -= 1 # 更新计数，为相同数字的下一个元素预留位置
    # 将排序好的输出数组拷贝回原数组
    for i in range(n):
        arr[i] = output[i]
```

在 radix_sort 函数中，首先通过 max(arr) 获取最大值，并初始化 exp 为 1，表示从个位开始处理。循环条件 max_val // exp > 0 确保处理完所有位数。每次循环调用 counting_sort_for_radix 对当前位进行排序，然后 exp 乘以 10 以移动到更高位。

在 counting_sort_for_radix 函数中，首先初始化输出数组和计数数组。遍历原数组，使用 (arr[i] // exp) % 10 提取当前位的数字，并更新计数数组。接着，将计数数组转换为前缀和数组，这样 count[i] 就表示数字 i 及更小数字的累计次数，从而确定元素在输出数组中的位置。逆向遍历原数组是为了保证稳定性：当多个元素具有相同当前位数字时，后出现的元素在输出数组中会被放置在较后位置，从而维持原有顺序。最后，将输出数组拷贝回原数组，完成当前位的排序。

14 深度分析与探讨

基数排序的时间复杂度为 $O(d \times (n + k))$，其中 d 是最大位数，n 是元素个数，k 是基数（例如 10）。当 d 较小且 n 较大时，基数排序的性能优于基于比较的排序算法。空间复杂度为 $O(n + k)$，主要来自计数排序中的输出数组和计数数组。

基数排序的优点包括速度快（在特定条件下）、稳定性高，适用于整数排序。缺点则是只能处理整数或可分解为位的数据类型，需要额外空间，且当数字范围极大（d 很大）时，效率可能下降。与快速排序和归并排序相比，基数排序在整数排序场景下更具优势，但通用性较差。

基数排序的变体包括 MSD（最高位优先）基数排序，它从最高位开始排序，适用于递归实现，但可能更复杂。处理负数时，可以将数组分为正负两部分，负数取绝对值排序后再反转。基数排序还可扩展至字符串排序，通过逐字符处理实现字典序排序。

基数排序通过逐位稳定的排序方式，实现了整数的高效排序。其核心思想是将排序问题分解为多个子问题，每一子问题处理一个位上的数字。使用计数排序作为子程序，保证了算法的稳定性和性能。基数排序在特定场景下展现出线性时间复杂度的优势，为大数据处理和数据

库索引等领域提供了实用工具。

15 互动与思考题

请尝试用您熟悉的编程语言实现上述基数排序算法，并用一个包含负数的数组测试其表现。思考一下，如果要用基数排序给日期（格式为 YYYYMMDD）排序，应该如何设计算法？欢迎在评论区分享您的疑问或应用场景！

第 IV 部

深入理解并实现基本的管道操作符（Pipe Operator）原理与实现

杨岢瑞

Oct 12, 2025

16 从函数式编程的优雅，到揭开语法糖的神秘面纱

在软件开发中，我们常常遇到函数嵌套调用的场景，例如处理数据时写出类似 `func3(func2(func1(data)))` 的代码。这种写法不仅可读性差，还让调试变得困难，因为执行顺序与书写顺序相反，仿佛在解一个层层包裹的谜题。更糟糕的是，当使用数组方法链式调用如 `array.map(...).filter(...).reduce(...)` 时，中间步骤的嵌套会让代码逻辑支离破碎。为了解决这个问题，管道操作符应运而生，它允许我们将代码重写为 `data ↑ func1 ↑ func2 ↑ func3` 的线性形式，让数据处理过程像流水一样自然流动。
管道操作符的核心思想是将数据视为流动的介质，而函数则是处理这个介质的工具。通过 ↑ 符号连接，数据从左向右依次传递，每个函数接收前一个函数的输出作为输入。这种写法不仅符合人类从左到右的阅读习惯，还让代码的意图更加清晰。本文的目标是深入解析管道操作符的原理，并引导读者使用 JavaScript 实现一个基础的管道工具函数，从而理解其背后的机制。

17 什么是管道操作符?

管道操作符的本质是构建一条数据流管道，将数据处理过程线性化。想象一条流水线，数据是流动的水，每个函数是一个处理器，而 ↑ 就是连接这些处理器的管道。在语法上，管道操作符的基本形式是 `value ↑ function`，其规则是将左侧表达式的求值结果作为唯一参数传递给右侧函数。如果函数需要多个参数，可以通过柯里化或箭头函数包装来解决，例如 `value ↑ (x => func(x, arg2))`。
这种思想在多门编程语言中流行。例如，在 F#、Elixir 和 Elm 中，管道操作符是原生特性；JavaScript 社区也通过 TC39 提案推动其标准化，目前有两种主要风格：Hack 风格和 F# 风格。此外，类似概念也存在于其他领域，如 Unix Shell 中的 | 管道符，用于连接命令，以及 RxJS 库中的 `.pipe()` 方法，用于组合响应式操作符。这些实现都强调了数据流的线性处理，提升了代码的抽象层次。

18 为何需要管道? —— 优势分析

管道操作符的首要优势是提升代码的可读性和可维护性。通过将嵌套调用转化为线性序列，代码变成自上而下的叙事，而非从内到外的解谜游戏。例如，一个复杂的数据转换过程可以用管道清晰地表达每一步操作，让读者一目了然地理解数据流向。这种结构还便于修改和扩展，只需在管道中插入或删除函数即可调整逻辑。
其次，管道促进了函数组合，这是函数式编程的基石。通过将小型、纯函数组合成复杂操作，我们可以构建模块化且可复用的代码块。每个函数只负责单一职责，而管道则负责将它们串联起来，这降低了代码的耦合度，并提高了测试的便利性。此外，管道还改善了调试体验；我们可以在中间插入日志函数，例如 `data ↑ func1 ↑ tap(console.log) ↑ func2`，实时观察数据状态，而无需破坏原有结构。

19　核心原理剖析：语法糖的本质

管道操作符本质上是一种语法糖，它通过编译器或解释器转换为更基础的函数调用形式。例如，表达式 a ↑ b ↑ c 会被「脱糖」为 c(b(a))，这意味着管道并没有引入新功能，而是提供了更友好的语法抽象。理解这一点至关重要，因为它揭示了管道的实现依赖于函数组合的求值顺序。

关键实现要点包括确保左侧值先被求值，然后将结果传递给右侧函数。在 JavaScript 等语言中，还需要考虑函数上下文绑定问题；如果函数依赖 this，可能需要使用 .bind(this) 来维护正确的作用域。这些细节保证了管道操作符的语义一致性，使其在不同场景下都能可靠工作。

20　动手实现：构建我们自己的 pipe 函数

我们的目标是实现一个 pipe(...fns) 函数，它接受一系列函数作为参数，并返回一个新函数。这个新函数会将输入值依次传递给每个函数，从左到右执行。下面以 JavaScript 为例，逐步构建这个工具。

首先，我们实现基础版本。代码如下：

```
function pipe(...fns) {
  return function (initialValue) {
    return fns.reduce((acc, fn) => fn(acc), initialValue);
  };
}
```

这段代码使用 reduce 方法模拟管道的数据流动。pipe 函数接受任意数量的函数 fns，然后返回一个闭包函数。这个闭包函数以 initialValue 为起点，通过 reduce 迭代：acc 是累积值，初始为 initialValue，然后依次应用每个函数 fn，将 fn(acc) 的结果作为新的 acc。这样，数据就像在管道中流动，每个函数处理前一个的输出。

使用示例可以更好地理解其工作方式。假设我们有三个函数：add 用于加法，double 用于翻倍，square 用于平方。传统嵌套调用是 square(double(add(1, 2)))，而使用 pipe 可以这样写：

```
const add = (x, y) => x + y;
const double = x => x * 2;
const square = x => x * x;

const compute = pipe(
  (x, y) => x + y, // 初始函数处理多参数
  double,
  square
);

console.log(compute(1, 2)); // 输出：36
```

这里，compute 是一个组合函数，它先执行加法，然后翻倍，最后平方。注意，初始函数通过箭头函数处理了多参数情况，这体现了管道的灵活性。

然而，基础版本假设所有函数都是同步的。在实际应用中，我们可能遇到异步操作，例如调用 API。为此，我们实现增强版的 asyncPipe，支持异步函数。代码如下：

```
async function asyncPipe(...fns) {
  return async function (initialValue) {
    let result = initialValue;
    for (const fn of fns) {
      result = await fn(result); // 依次等待每个函数执行
    }
    return result;
  };
}
```

这个实现使用 async/await 语法来处理异步函数。它通过 for 循环遍历每个函数，并使用 await 确保前一个函数完成后再执行下一个。这样，管道可以处理 Promise 链，适用于从数据库查询到数据处理的完整异步流程。

21 进阶话题与展望

在更复杂的场景中，我们的 pipe 实现与库如 RxJS 的 pipe 有本质区别。我们的版本是「急求值」的，即立即执行所有函数；而 RxJS 的 pipe 用于组合 Observable 操作符，这些操作符是「惰性」的，只在订阅时执行。这种区别体现了响应式编程中数据流的延迟计算特性。

错误处理是管道中的一个重要话题。在默认情况下，一个函数的错误会中断整个管道。为了安全地处理异常，我们可以引入函数式编程中的 Monad 概念，例如 Maybe 或 Result 类型。这些类型封装了可能失败的计算，允许我们在管道中传播错误而不中断流程，从而编写出更健壮的代码。

在 TypeScript 中，为 pipe 函数添加类型推断可以提升开发体验。通过泛型和条件类型，我们可以确保每个函数的输入和输出类型正确衔接，实现优秀的类型安全和自动补全。例如，TypeScript 可以推断出 pipe(f, g) 的返回类型是 g 的返回类型，前提是 g 的输入类型匹配 f 的输出类型。这减少了运行时错误，并提高了代码的可维护性。

管道操作符通过将数据流线性化，极大地提升了代码的声明性和可读性。其核心原理是函数组合的语法糖，它将嵌套调用转化为直观的序列。即使在没有原生支持的语言中，我们也可以通过简单的工具函数如 pipe 来模拟这种体验，从而编写出更清晰、更易维护的代码。理解这些原理不仅有助于我们使用现有工具，还能激发我们在项目中应用函数式编程思想，推动软件质量的持续改进。

第 V 部

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

22 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。

23 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。

24 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。

25 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。

26 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。

27　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;
         }
         case RET:
            return stack[--sp];
      }
   }
}
```

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。

28 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>

void* allocate_executable_memory(size_t size) {
    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。

29 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：

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

30 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。

31 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。

32 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。

33 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。

34 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。

35 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。

36 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。