c13n #36

c13n

2025 年 11 月 19 日

第 I 部

基本的模数转换器（ADC）原理与实现

黄京

Oct 14, 2025

探索采样、量化与编码的奥秘，并用电路和代码亲手实现一个简单的 ADC。

我们生活在一个充满连续变化的模拟世界中，温度的高低、声音的强弱、光线的明暗都以模拟形式存在。然而，计算机和数字系统只能处理离散的二进制数据，即 0 和 1。模数转换器（ADC）正是连接这两个世界的桥梁，它负责将连续的模拟信号转换为离散的数字信号，使得物理世界的信息能够被数字设备理解和处理。ADC 在现代技术中无处不在，例如智能手机的触摸屏通过 ADC 检测触摸位置，麦克风录音时将声波转换为数字音频流，数字体温计测量体温并显示数值，物联网传感器采集环境数据并上传到云端。这些应用都依赖于 ADC 的精确转换能力。本文旨在深入讲解 ADC 的核心工作原理，包括采样、量化和编码三个关键步骤，并介绍几种主流 ADC 类型及其优缺点。最终，我们将引导读者通过软件模拟和硬件理解的方式，实现一个基本的逐次逼近型 ADC 模型，从而将理论知识转化为实践技能。

1 ADC 的核心三部曲：采样、量化、编码

模数转换过程可以概括为三个基本步骤：采样、量化和编码。这些步骤共同作用，将连续的模拟信号转换为离散的数字代码。首先，采样阶段负责在时间维度上捕捉模拟信号的瞬时值。采样以固定的时间间隔进行，这个间隔的倒数称为采样频率。采样频率的选择至关重要，它必须遵循奈奎斯特-香农采样定理，即采样频率至少是信号最高频率的两倍，以避免混叠现象。混叠会导致高频信号被错误地表示为低频信号，从而失真。例如，如果一个模拟信号的最高频率成分是 1000 Hz，那么采样频率至少应为 2000 Hz 才能准确重建信号。在实际应用中，如果采样频率过低，原本的高频正弦波可能会被误判为低频波形，造成数据错误。

接下来是量化阶段，它将采样得到的连续电压值映射到有限个离散的电压等级中。量化过程引入了分辨率的概念，分辨率通常用位数表示，例如一个 8 位 ADC 可以将电压范围划分为 (2^8 = 256) 个等级。假设一个 3 位 ADC 的参考电压范围为 0 到 5 V，那么它会将这个范围分成 8 个等间隔的区间，每个区间对应一个离散等级。例如，0 到 0.625 V 对应等级 0，0.625 到 1.25 V 对应等级 1，依此类推，直到 4.375 到 5 V 对应等级 7。量化过程中不可避免会产生量化误差，这是因为连续的电压值被“四舍五入”到最接近的离散等级。量化误差的最大值为 ±1/2 LSB（最低有效位），例如在上述 3 位 ADC 中，每个等级的步长为 0.625 V，因此最大误差为 ±0.3125 V。这种误差是 ADC 固有的，无法完全消除，但可以通过提高分辨率来减小。

最后，编码阶段将量化后的等级值转换为二进制代码，以便数字系统处理。继续以 3 位 ADC 为例，量化等级 0 可能编码为二进制 000，等级 1 为 001，一直到等级 7 为 111。编码过程通常使用标准二进制或补码格式，具体取决于应用需求。通过这三个步骤，ADC 成功地将模拟信号转换为数字形式，为后续的数字处理和分析奠定了基础。

2 常见 ADC 类型巡礼

在模数转换领域，有多种 ADC 架构可供选择，每种都有其独特的优缺点和适用场景。Flash ADC 是一种高速转换器，它使用一列比较器并行比较输入电压与参考电压，从而实现极快的转换速度。然而，Flash ADC 的电路复杂度随分辨率指数增长，例如一个 8 位 Flash ADC 需要 255 个比较器，导致高功耗和高成本，因此它主要用于超高速应用如示波器和雷达系统。

逐次逼近型 ADC（SAR ADC）在速度、精度和成本之间取得了良好平衡，因此成为最通用的 ADC 类型之一。它的工作原理类似于天秤称重，从最高位开始逐位试探和比较输入电压。SAR ADC 通过一个数模转换器（DAC）生成试探电压，并与输入电压比较，根据比较结果调整二进制代码。这种架构速度适中，广泛应用于微控制器和数据采集系统。本文将重点介绍并实现这种 ADC 类型。

双积分型 ADC 采用电压-时间转换原理，通过两次积分过程将输入电压转换为时间间隔，再通过计时得到数字值。这种 ADC 具有高精度和强抗干扰能力，但转换速度较慢，因此适用于数字万用表和高精度测量仪器。此外，Sigma-Delta ADC 使用过采样和噪声整形技术，以高速 1 位 ADC 为基础实现高分辨率。它成本较低，但建立时间慢，常用于音频采集和高精度传感器应用。

3　动手实践：实现一个简易的逐次逼近型 ADC

逐次逼近型 ADC 的核心思想是二分搜索算法，它通过逐位比较来逼近输入电压值。首先，ADC 从最高位开始，将该位置 1，其余位为 0，生成一个试探电压。然后，通过 DAC 将试探值转换为模拟电压，并与输入电压比较。如果试探电压小于或等于输入电压，则保留该位为 1；否则，清除该位为 0。接着，移动到下一位，重复这个过程，直到所有位都被处理完毕。最终，得到的二进制代码就是输入电压的数字表示。例如，在一个 4 位 ADC 中，如果输入电压为 2.7 V，参考电压为 5 V，那么转换过程可能从二进制 1000（对应 2.5 V）开始，逐步调整到 1010（对应 3.125 V），最后得到 1001（对应 2.8125 V），完成逼近。

现在，我们通过 Python 代码模拟 SAR ADC 的逻辑。以下代码定义了一个简单的 SAR ADC 函数，它接受输入电压、参考电压和分辨率作为参数，并返回数字输出。代码首先初始化数字输出和当前位，然后通过循环逐位进行试探和比较。在每次迭代中，代码生成试探值，将其转换为模拟电压（通过简单的计算模拟 DAC），并与输入电压比较，根据结果更新数字输出。最后，函数返回最终的二进制代码。

```
def sar_adc(input_voltage, vref, bits):
    digital_output = 0
    current_bit = bits - 1 # 从最高位开始
    while current_bit >= 0:
        # 将当前位置 1，生成试探值
        trial = digital_output | (1 << current_bit)
        # 将试探值转换为模拟电压（模拟 DAC 输出）
        trial_voltage = (trial / (2**bits)) * vref
        # 与输入电压比较
        if trial_voltage <= input_voltage:
            digital_output = trial # 保留该位
        current_bit -= 1 # 移至下一位
    return digital_output

# 示例使用
vref = 5.0 # 参考电压 5V
```

```
bits = 4 # 4 位分辨率
input_voltage = 2.7 # 输入电压 2.7V
result = sar_adc(input_voltage, vref, bits)
print(f"输入电压 {input_voltage} V 的数字输出为 : {bin(result)}")
```

在这段代码中，我们首先定义函数 sar_adc，它使用 digital_output 变量存储当前数字值，current_bit 表示当前处理的位位置。循环从最高位（bits-1）开始，到最低位（0）结束。在每次循环中，我们使用位操作 (1 << current_bit) 将当前位设为 1，然后通过 trial_voltage = (trial / (2**bits)) * vref 计算试探电压，这模拟了 DAC 的转换过程。接着，比较试探电压与输入电压，如果试探电压小于或等于输入电压，则更新 digital_output 以保留该位。最后，函数返回数字输出。运行示例后，对于输入电压 2.7 V，代码可能输出二进制 0b1001，表示数字值 9，对应电压约 2.8125 V，体现了量化误差。在硬件层面，SAR ADC 由逐次逼近寄存器、数模转换器、比较器和控制逻辑组成。工作时，时钟信号驱动控制逻辑，逐位生成试探代码，DAC 将其转换为模拟电压，比较器判断大小，并根据结果更新寄存器。这种电路结构在微控制器中常见，例如 Arduino 的 analogRead() 函数内部就使用了类似的 ADC。通过在线仿真工具如 Falstad Circuit Simulator，可以搭建并观察 SAR ADC 的时序行为，加深理解。

4 ADC 的关键性能参数解读

ADC 的性能由多个参数描述，这些参数直接影响转换精度和速度。分辨率是 ADC 能够区分的最小电压变化，通常用位数表示，例如一个 10 位 ADC 在 0 到 5 V 范围内的分辨率约为 4.88 mV。分辨率越高，量化误差越小，但转换可能更慢或成本更高。采样率指每秒采样的次数，它必须满足奈奎斯特定理以避免混叠，例如在音频应用中，采样率通常设为 44.1 kHz 以覆盖人耳可闻频率范围。

信噪比（SNR）衡量 ADC 输出信号与噪声的比率，它与分辨率密切相关，近似公式为 (SNR \approx 6.02N + 1.76) dB，其中 N 是 ADC 的位数。例如，一个 12 位 ADC 的理想 SNR 约为 74 dB，表示信号强度远高于噪声。有效位数（ENOB）则反映 ADC 在实际应用中的真实性能，它可能低于标称位数，因为实际电路会引入噪声和非线性误差。理解这些参数有助于在选择 ADC 时权衡速度、精度和成本。

本文从模拟世界与数字世界的连接需求出发，详细讲解了 ADC 的核心原理，包括采样、量化和编码三个步骤，并介绍了多种 ADC 类型，重点实现了逐次逼近型 ADC 的软件模拟和硬件理解。ADC 作为物理世界与数字系统交互的关键组件，其重要性不言而喻，它使得温度、声音和光线等模拟量能够被计算机处理和分析。展望未来，读者可以进一步在真实硬件上实践，例如使用 Arduino 的 analogRead() 函数测量光敏电阻或电位器，将理论知识应用于实际项目。通过不断探索和实验，我们能够更深入地理解 ADC 在现代技术中的核心作用，并推动创新应用的发展。如果您有任何问题或项目经验，欢迎分享和讨论。

第 II 部

基本的垃圾回收（Garbage Collection）机制

李睿远

Oct 15, 2025

告别内存泄漏，亲手构建你的第一个微型垃圾收集器。

在软件开发中，内存管理一直是一个核心且复杂的问题。想象一下，你编写了一个 C 语言程序，动态分配了内存却忘记释放，就像打开水龙头后没有关闭一样，内存使用量会不断攀升，最终导致程序崩溃。例如，一段简单的 C 代码中，如果重复调用 `malloc` 而不调用 `free`，内存泄漏会逐渐累积，系统资源被耗尽。相比之下，像 Java 或 Python 这样的语言内置了垃圾回收机制，开发者无需手动管理内存，从而大大提升了开发效率。垃圾回收的核心价值在于自动化内存管理，它让程序员能专注于业务逻辑，而不是底层细节。同时，它还能避免悬挂指针、重复释放等经典内存错误，增强程序的健壮性。本文的目标是引导读者从理论出发，理解垃圾回收的基本概念和经典算法，并最终通过 Python 实现一个简化的标记-清除垃圾回收器，从而在实践中深化理解。

5 垃圾回收的基石：概念与术语

在垃圾回收领域，「垃圾」指的是程序中无法再被访问到的对象所占用的内存。判断垃圾的唯一标准是「可达性」，即一个对象是否还能通过引用链被访问到。根对象是垃圾收集器遍历的起点，常见的根对象类型包括全局变量、当前执行函数的栈上的变量以及寄存器中的引用。这些根对象构成了对象图的入口点。对象图是一个有向图，其中节点代表内存中的对象，边代表对象之间的引用关系。活动对象是指从根对象出发，通过引用链可以访问到的所有对象，而垃圾对象则是从任何根对象都无法到达的对象。理解这些基本概念是掌握垃圾回收机制的前提。

6 经典垃圾回收算法巡礼

引用计数法是一种直观的垃圾回收算法，其核心思想是为每个对象维护一个引用计数器。每当有新的引用指向该对象时，计数器加一；当引用消失时，计数器减一。一旦计数器归零，对象便立即被回收。这种方法的优点在于简单和实时性高，但存在致命的循环引用问题。例如，如果两个对象互相引用，它们的计数器永远不会归零，导致内存泄漏。此外，频繁更新计数器也会带来性能开销。

标记-清除法通过两个阶段来解决循环引用问题。在标记阶段，垃圾收集器从所有根对象开始，递归地遍历对象图，为所有可达的对象打上活动标记。在清除阶段，收集器遍历整个堆，回收未被标记的对象，并将它们的内存加入空闲链表以备后续分配。这种方法虽然能有效处理循环引用，但缺点包括「停止世界」现象，即标记和清除过程中应用程序需要暂停，以及内存碎片化问题，可能导致后续无法分配大对象。

标记-整理法在标记-清除的基础上增加了整理阶段，以解决内存碎片化。标记阶段与标记-清除法相同，随后收集器将所有活动对象向内存的一端移动，紧凑排列，并更新所有指向被移动对象的引用。这样消除了碎片，但移动对象和更新引用的开销较大。

复制算法将堆内存分为两个相等部分：From 空间和 To 空间。新对象只分配在 From 空间，当 From 空间满时，程序暂停，所有活动对象被复制到 To 空间并紧凑排列，然后交换 From 和 To 空间的角色。这种方法的优点是分配速度快且无碎片，但内存利用率只有 50%。

分代收集法基于「弱代假说」，即大多数对象生命周期短，只有少数对象能长期存活。它将堆划分为不同代，如年轻代和老年代。新对象分配在年轻代，年轻代垃圾回收频繁且快速，

通常使用复制算法。存活多次回收的对象晋升到老年代，老年代回收不频繁但耗时较长，常用标记-清除或标记-整理法。这种方法平衡了性能和内存使用。

7　动手实践：用 Python 实现一个微型标记-清除 GC

为了清晰展示垃圾回收原理，我们将用 Python 模拟一个虚拟的「堆」和「对象」，实现一个极简的标记-清除垃圾回收器。首先，我们设计一个「微型世界」，包括一个 VM 类来模拟虚拟机和一个 Object 类来代表对象。VM 类包含栈（模拟根集合）、堆（用字典存储对象）和下一个可用对象 ID。Object 类则包含对象 ID、标记位和引用列表。

在核心功能实现中，VM.new_object 方法用于在堆上分配新对象，它创建一个 Object 实例并分配唯一 ID。VM.push 和 VM.pop 方法分别用于将对象压入栈和弹出栈，模拟根集合的变化。VM.mark 方法实现标记阶段，从栈中的根对象开始，递归遍历所有可达对象，并将它们的标记位设为 True。VM.sweep 方法实现清除阶段，遍历堆中的所有对象，删除未标记的对象，并重置已标记对象的标记位。VM.gc 方法是垃圾回收的入口，依次调用 mark 和 sweep。

下面是一个简单的代码示例，展示如何创建和回收普通对象。首先，我们初始化虚拟机，创建对象 A 并将其压入栈，然后创建对象 B 并由 A 引用。接着，将 A 弹出栈，触发垃圾回收，观察 A 和 B 是否被正确回收。

```
class Object:
    def __init__(self, obj_id):
        self.id = obj_id
        self.marked = False
        self.refs = []

class VM:
    def __init__(self):
        self.stack = []
        self.heap = {}
        self.next_id = 0

    def new_object(self):
        obj_id = self.next_id
        self.next_id += 1
        obj = Object(obj_id)
        self.heap[obj_id] = obj
        return obj

    def push(self, obj):
        self.stack.append(obj)

    def pop(self):
```

```
        return self.stack.pop() if self.stack else None

    def mark(self):
        for obj in self.stack:
            self._mark_recursive(obj)

    def _mark_recursive(self, obj):
        if obj is None or obj.marked:
            return
        obj.marked = True
        for ref_id in obj.refs:
            ref_obj = self.heap.get(ref_id)
            self._mark_recursive(ref_obj)

    def sweep(self):
        to_remove = []
        for obj_id, obj in self.heap.items():
            if not obj.marked:
                to_remove.append(obj_id)
            else:
                obj.marked = False
        for obj_id in to_remove:
            del self.heap[obj_id]

    def gc(self):
        self.mark()
        self.sweep()
```

在这段代码中，_mark_recursive 方法递归地标记所有从根对象可达的对象，确保活动对象不被误删。sweep 方法则清理未标记对象，释放内存。通过这个实现，我们可以测试两种场景：普通对象的回收和循环引用的处理。在循环引用场景中，创建两个对象 C 和 D，让它们互相引用，但不放入栈中，触发垃圾回收后，它们会被正确回收，证明标记-清除法解决了循环引用问题。

随着计算需求的增长，垃圾回收技术不断演进。并行垃圾回收使用多个线程并行工作，但应用程序仍需暂停；而并发垃圾回收允许 GC 线程与应用程序线程同时运行，显著减少了暂停时间，这在现代高性能 GC 如 G1 和 ZGC 中广泛应用。三色标记抽象是一种优雅的建模方法，将对象分为白色（未访问）、灰色（已标记但子对象未检查）和黑色（完全标记），这有助于实现并发标记过程。

总之，垃圾回收的核心思想是自动化内存管理，基于可达性判断对象生命周期。不同算法在时间、空间和复杂度上各有权衡，理解这些原理有助于在使用带 GC 的语言时编写高效、健壮的代码。通过本文的理论学习和实践，读者可以更深入地掌握内存管理的艺术。

第 III 部

基本的 IPv6 协议栈

李睿远

Oct 16, 2025

从数据包结构到代码实现，亲手构建网络核心

随着互联网的快速发展，IPv4 地址的枯竭已成为不争的事实。IPv4 仅提供约 43 亿个地址，而全球设备数量早已远超这一数字，导致地址分配紧张和各种过渡技术的出现。相比之下，IPv6 采用 128 位地址长度，理论上可提供 2^{128} 个地址，这一数量级足以满足未来数十年的需求。根据最新数据，全球 IPv6 部署率持续上升，许多大型网络和服务商已全面支持 IPv6。因此，对于现代开发者和网络工程师而言，深入理解 IPv6 不再是可选技能，而是必备知识。

IPv6 的核心优势远不止于地址空间的扩展。其报头格式经过简化，固定为 40 字节，去除了 IPv4 中的可选字段，转而使用扩展报头链式处理，这大大提升了路由效率。此外，IPv6 内置了对 IPsec 的支持，增强了端到端的安全性，同时更好地支持移动性和无状态地址自动配置（SLAAC），使得设备能够快速接入网络。本文旨在通过理论与实践相结合的方式，引导读者从零开始实现一个用户态的 IPv6 协议栈。我们将聚焦于 IPv6 层、ICMPv6 和邻居发现协议（NDP），暂不涉及 TCP/UDP 等上层协议，最终目标是实现一个能响应 Ping（ping6）并进行邻居发现的微型协议栈。

8 理论基础：深入剖析 IPv6 核心机制

IPv6 数据包结构是理解其设计理念的基础。IPv6 基本报头包含多个字段：版本（Version）固定为 6，流量类别（Traffic Class）用于 QoS，流标签（Flow Label）标识特定流，载荷长度（Payload Length）指示扩展报头和数据的总长度，下一个报头（Next Header）指定后续内容类型，跳数限制（Hop Limit）类似 IPv4 的 TTL，以及源和目的地址各 128 位。与 IPv4 报头相比，IPv6 报头更加简化，去除了校验和、分片等相关字段，转而依赖上层协议处理。扩展报头以链式结构存在，例如 Hop-by-Hop Options 和 ICMPv6（下一个报头值为 58），它们允许灵活地添加功能而不改变基本结构。

ICMPv6 在 IPv6 中扮演着多重角色，不仅是错误报告和网络诊断的工具，更是邻居发现协议（NDP）的载体。NDP 替代了 IPv4 中的 ARP，用于解析 IPv6 地址到 MAC 地址的映射。关键报文类型包括 Echo Request 和 Echo Reply 用于实现 ping6，Neighbor Solicitation 和 Neighbor Advertisement 用于地址解析，以及 Router Solicitation 和 Router Advertisement 用于无状态地址自动配置。NDP 的状态机涉及多个状态：INCOMPLETE、REACHABLE、STALE、DELAY 和 PROBE，它们根据超时和事件进行转换，确保邻居关系的有效管理。

在地址体系方面，IPv6 定义了单播、组播和任播地址。单播地址包括全球单播（如 2000::/3）和链路本地地址（fe80::/10），其中链路本地地址在 NDP 通信中至关重要。无状态地址自动配置（SLAAC）允许设备通过接收 Router Advertisement 报文并结合 EUI-64 格式生成接口标识符，自动配置全球单播地址，这简化了网络部署过程。

9 实战准备：搭建开发环境与架构设计

在技术选型上，我们推荐使用 C 语言进行实现，因为它贴近系统底层，能够直接操作内存和网络接口，适合协议栈开发。开发平台选择 Linux，因其原生支持 TUN/TAP 设备，允许用户态程序模拟网络接口。关键工具包括 tcpdump 或 Wireshark 用于抓包分析，ping6 和 traceroute6 用于测试，而 TUN/TAP 设备则是核心，它通过字符设备接口提供原始数

据包的读写能力。

协议栈的软件架构应模块化设计。TUN 驱动模块负责从 TUN 设备读取和写入原始以太网帧；以太网帧处理模块解析帧类型，识别 IPv6 数据包（以太网类型 0x86DD）；IPv6 解包/组包模块处理基本报头和扩展报头；ICMPv6 处理模块响应 Echo 请求和邻居发现报文；邻居缓存表存储 IPv6 地址到 MAC 地址的映射及其状态；定时器模块处理 NDP 状态超时和重传。数据流从 TUN 设备进入，经以太网帧解析后，如果是 IPv6 包，则递交给 IPv6 模块，再根据下一个报头调用 ICMPv6 模块，最终可能更新邻居缓存或返回响应。

10　代码实现：构建核心模块

首先，我们搭建项目骨架并初始化 TUN 接口。以下代码展示如何在 C 语言中创建和配置 TUN 设备。

```
#include <fcntl.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <string.h>

int tun_alloc(char *dev) {
   struct ifreq ifr;
   int fd, err;
   if ((fd = open("/dev/net/tun", O_RDWR)) < 0) {
      return -1;
   }
   memset(&ifr, 0, sizeof(ifr));
   ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
   if (dev) {
      strncpy(ifr.ifr_name, dev, IFNAMSIZ);
   }
   if ((err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0) {
      close(fd);
      return err;
   }
   strcpy(dev, ifr.ifr_name);
   return fd;
}
```

这段代码通过打开 /dev/net/tun 设备文件并使用 ioctl 调用配置 TUN 接口。IFF_TUN 标志表示创建 TUN 设备（三层网络设备），IFF_NO_PI 表示不提供包信息，从而简化数据包处理。函数返回文件描述符，用于后续读写操作。初始化后，程序可以进入数据包读写循环，不断从文件描述符读取原始数据包并进行处理。

接下来，实现以太网帧与 IPv6 报文的解析。我们定义相关结构体并处理字节序转换。

```
#include <arpa/inet.h>
#include <stdint.h>

struct ethhdr {
    unsigned char h_dest[6];
    unsigned char h_source[6];
    uint16_t h_proto;
};

struct ip6_hdr {
    uint32_t v_tc_fl;
    uint16_t plen;
    uint8_t nxt;
    uint8_t hlim;
    struct in6_addr src;
    struct in6_addr dst;
};

void parse_ethernet(unsigned char *buffer, int length) {
    struct ethhdr *eth = (struct ethhdr *)buffer;
    uint16_t proto = ntohs(eth->h_proto);
    if (proto == 0x86DD) {
        parse_ipv6(buffer + sizeof(struct ethhdr), length - sizeof(
            ↪ struct ethhdr));
    }
}
```

在这段代码中，我们定义了以太网帧头和 IPv6 报头的结构体。`ethhdr` 包含目的和源 MAC 地址以及协议类型字段；`ip6_hdr` 使用组合字段处理版本、流量类别和流标签，注意这些字段需要字节序转换，例如 `ntohs` 用于将网络字节序转换为主机字节序。在 `parse_ethernet` 函数中，我们检查协议类型是否为 IPv6（0x86DD），如果是，则调用 IPv6 解析函数。IPv6 报头解析需要校验版本字段是否为 6，并处理载荷长度以确保数据完整性。

现在，实现 ICMPv6 协议处理，首先是响应 Ping 请求。

```
struct icmp6_hdr {
    uint8_t type;
    uint8_t code;
    uint16_t checksum;
};
```

```
void handle_icmp6(unsigned char *buffer, int length, struct in6_addr
    ↪ src, struct in6_addr dst) {
   struct icmp6_hdr *icmp6 = (struct icmp6_hdr *)buffer;
   if (icmp6->type == 128) {
      icmp6->type = 129;
      struct in6_addr tmp = src;
      src = dst;
      dst = tmp;
      icmp6->checksum = 0;
      icmp6->checksum = compute_checksum(src, dst, buffer, length);
      send_packet(buffer, length);
   }
}
```

这里，我们定义 ICMPv6 报头结构，类型字段为 128 表示 Echo Request，129 表示 Echo Reply。处理时，我们改变类型，交换源和目的地址，并重新计算校验和。校验和计算涉及 IPv6 伪首部，包括源地址、目的地址、载荷长度和下一个报头值。计算函数 compute_checksum 需要实现，它基于标准算法，对伪首部和 ICMPv6 报文进行求和。校验和计算是 ICMPv6 的关键部分。IPv6 的校验和使用伪首部，其结构包括源地址（16 字节）、目的地址（16 字节）、上层包长度（32 位）、下一个报头（8 位，后跟 24 位零）。公式可表示为：校验和 = ~(sum(伪首部) + sum(ICMPv6 报文))，其中 sum 是 16 位字的补码和。在代码中，我们需要遍历这些数据并计算。

```
uint16_t compute_checksum(struct in6_addr src, struct in6_addr dst,
    ↪ unsigned char *data, int len) {
   uint32_t sum = 0;
   for (int i = 0; i < 16; i += 2) {
      sum += (src.s6_addr[i] << 8) | src.s6_addr[i+1];
   }
   for (int i = 0; i < 16; i += 2) {
      sum += (dst.s6_addr[i] << 8) | dst.s6_addr[i+1];
   }
   sum += len;
   sum += 58;
   for (int i = 0; i < len; i += 2) {
      if (i+1 < len) {
         sum += (data[i] << 8) | data[i+1];
      } else {
         sum += data[i] << 8;
      }
   }
   while (sum >> 16) {
      sum = (sum & 0xFFFF) + (sum >> 16);
```

```
    }
    return ~sum;
}
```

这段代码实现了校验和计算。我们首先处理伪首部：源和目的地址各作为 8 个 16 位字处理，上层包长度作为 32 位值但以 16 位字形式添加，下一个报头值 58 也作为 16 位字添加。然后处理 ICMPv6 报文数据，以 16 位为单位求和，并处理进位。最终返回补码。注意，在实现中，我们假设数据是网络字节序，且长度参数 `len` 是 ICMPv6 报文的字节长度。

11 实现邻居发现协议

邻居发现协议（NDP）是 IPv6 的核心组件，用于解析链路层地址。我们首先设计邻居缓存表，存储 IPv6 地址到 MAC 地址的映射及其状态。

```
#include <time.h>

#define ND6_INCOMPLETE 0
#define ND6_REACHABLE 1

struct neighbor_cache {
    struct in6_addr ip6_addr;
    unsigned char mac_addr[6];
    int state;
    time_t last_updated;
};

struct neighbor_cache ncache[100];
int ncache_size = 0;

void handle_neighbor_solicitation(unsigned char *buffer, int length,
    ↪ struct in6_addr src, struct in6_addr dst) {
    struct icmp6_ns *ns = (struct icmp6_ns *)buffer;
    if (is_my_address(ns->target)) {
        send_neighbor_advertisement(ns->target, my_mac_addr);
    }
}

void send_neighbor_solicitation(struct in6_addr target) {
}
```

在这段代码中，我们定义了邻居缓存表的结构，包括 IPv6 地址、MAC 地址、状态和时间戳。`handle_neighbor_solicitation` 函数处理收到的 NS 报文，如果目标地址匹配本机地址，则发送 NA 响应。发送 NA 时，需要设置相关标志，如覆盖位（O）和请求位（S），

以指示响应的性质。同时，当协议栈需要发送数据包但缓存中无对应条目时，应主动发送 NS 报文，并将状态设置为 INCOMPLETE，等待 NA 响应后更新为 REACHABLE。

12 测试与验证

搭建测试环境时，需要将协议栈程序与主机网络连接。通过配置 TUN 设备并设置路由，使主机的 IPv6 流量指向该设备。例如，在 Linux 中，可以使用 `ip link set dev tun0 up` 和 `ip route add local 2001:db8::/64 dev tun0` 等命令。

功能测试包括 Ping 测试和邻居发现测试。从主机执行 `ping6 fe80::1%tun0`（假设协议栈配置了链路本地地址），同时使用 Wireshark 抓包验证请求和回复报文。此外，通过 `ip -6 neighbor show` 命令观察邻居表，确认协议栈的条目出现并达到 REACHABLE 状态。抓包结果应显示完整的 NS/NA 和 Echo 交互过程：例如，当发送 Ping 请求时，Wireshark 应显示 Echo Request 和 Echo Reply 报文，且邻居发现报文应显示 NS 和 NA 的交换，确保地址解析成功。

通过本文的实现，我们构建了一个具备 IPv6 基本报头处理、ICMPv6 Echo 响应和邻居发现功能的用户态协议栈。这个过程不仅加深了对 IPv6 协议的理解，还展示了如何将理论转化为实践。未来，可以扩展支持更多扩展报头如分片、实现 DHCPv6 客户端、添加 TCP/UDP 上层协议支持，以及优化性能和多线程处理。网络协议的设计精巧而实用，亲手实现是掌握它的最佳途径。鼓励读者尝试实现，并参考相关开源项目进一步探索。

第 IV 部

基本的基数排序（Radix Sort）算法

杨子凡

Oct 18, 2025

你是否听说过一种不基于比较的排序算法？它像整理档案一样，逐位、逐层地对数字进行归类，最终神奇地实现有序。本文将带你深入探索基数排序的独特魅力，从核心思想到代码实现，彻底掌握这一高效的非比较排序算法。

在排序算法的广阔领域中，主流比较排序算法如快速排序和归并排序的时间复杂度下限为 $O(n \log n)$，这是基于比较操作的固有限制。然而，基数排序作为一种非比较型整数排序算法，打破了这一瓶颈，其时间复杂度可以达到 $O(k \times n)$，其中 k 代表数字的最大位数。这种算法在特定场景下，例如处理固定位数的整数数据时，能够展现出超越传统排序方法的效率。本文旨在帮助读者彻底理解基数排序的工作原理，掌握最低位优先（LSD）的实现方式，并深入探讨其优势、局限性以及适用场景，从而为实际应用提供坚实的技术基础。

13 基数排序的核心思想

基数排序的核心思想可以通过一个生动的比喻来理解：想象整理一副扑克牌，我们首先按照花色（如红心、黑桃等）将牌分成几个大类，然后在每个花色内部按照点数（从 A 到 K）进行排序。类似地，基数排序通过逐位处理数字，先按最低位（如个位数）将数字分配到不同的「桶」中，然后按顺序收集，再处理更高位（如十位数、百位数），依次类推，直到最高位处理完毕，最终实现整体有序。这里涉及两个关键概念：基数和关键码。基数指的是每一位数字的取值范围，例如对于十进制数，基数为 10（涵盖 0 到 9）；关键码则是排序所依据的每一位数字本身。基数排序主要有两种策略：最低位优先（LSD）和最高位优先（MSD）。LSD 从数字的最低位开始排序，是本文的重点；而 MSD 则从最高位开始，采用递归分治的方式，本文仅作简要提及以供拓展。

14 算法步骤详解（以 LSD 为例）

基数排序的 LSD 实现过程可以分解为几个清晰的步骤，无需依赖图示，我们通过文字描述来构建完整的理解。首先，进行准备工作：确定待排序数组中的最大值 `max_val`，并计算最大数字的位数 k，公式为 $k = \lfloor \log_{10}(\text{max\_val}) \rfloor + 1$，这决定了排序所需的轮数。接下来，从最低位（个位）开始，逐位进行排序。例如，在第一轮中，我们创建 10 个桶，分别对应数字 0 到 9，然后遍历数组，根据每个数字的个位数将其放入对应桶中。完成后，按桶的编号顺序（从 0 到 9）依次将元素收集回原数组。这时，数组在个位数上已经达到局部有序。进入第二轮，我们清空桶，基于十位数重复上述分发和收集过程。关键点在于，必须从低到高排序，因为高位相同的数字，其顺序由低位的排序结果决定，这依赖于排序的稳定性。稳定性确保前一轮的排序成果不被破坏，例如如果两个数字的十位数相同，它们的相对顺序由个位数的排序结果保持。重复这一过程，直到处理完最高位，数组便完全有序。整个算法强调稳定性的重要性，它是基数排序正确性的基石。

15 代码实现（以 Python 为例）

以下是一个完整的 Python 实现基数排序的代码示例，我们将逐段进行详细解读，帮助读者理解每一部分的功能。

```
def radix_sort(arr):
    # 特殊情况处理：如果数组为空或只有一个元素，直接返回
```

```
    if len(arr) <= 1:
        return arr

    # 寻找数组中的最大值，以确定最大位数
    max_val = max(arr)
    # 计算最大位数 k，使用对数函数并取整
    exp = 1
    while max_val // exp > 0:
        # 创建 10 个桶，每个桶是一个空列表，用于存放对应数字的元素
        buckets = [[] for _ in range(10)]

        # 分发阶段：遍历数组，根据当前位数将元素放入对应桶中
        for num in arr:
            digit = (num // exp) % 10 # 计算当前位数字
            buckets[digit].append(num) # 将数字放入对应桶

        # 收集阶段：按桶顺序（0 到 9）将元素覆盖回原数组
        arr = []
        for bucket in buckets:
            arr.extend(bucket) # 将每个桶中的元素依次添加到数组

        # 更新指数，进入下一位处理
        exp *= 10

    return arr
```

在这段代码中，我们首先处理边界情况，如果数组长度小于等于 1，则无需排序直接返回。接着，通过 `max(arr)` 找到最大值 `max_val`，并使用 `exp` 变量来表示当前处理的位数（初始为 1，代表个位）。主循环继续执行，只要 `max_val // exp` 大于 0，就意味着还有更高位需要处理。在每一轮循环中，我们创建 10 个桶，用于存放数字 0 到 9 对应的元素。分发阶段通过 `(num // exp) % 10` 计算每个数字在当前位的值，并将其添加到相应桶中。例如，如果 `exp` 为 1，则计算个位数；如果 `exp` 为 10，则计算十位数。收集阶段则按桶的顺序将元素重新组合到数组中，确保稳定性，因为桶内元素顺序保持不变。最后，通过 `exp *= 10` 更新指数，处理下一位。整个过程重复直到最高位处理完毕，返回排序后的数组。为了验证代码，我们可以使用测试用例，例如输入 `[170, 45, 75, 90, 2, 802, 24, 66]`，并观察每轮排序后的中间结果，例如第一轮后数组在个位数上有序，第二轮后在十位数上局部有序，最终完全有序。

16 算法分析

基数排序的时间复杂度为 $O(k \times n)$，其中 k 是最大数字的位数，n 是数组长度。详细来说，每一轮分发需要遍历所有 n 个元素，收集同样需要 $O(n)$ 时间，总共进行 k 轮，因此总时

间为 $O(k \times n)$。需要注意的是，当 k 远小于 n 时（例如数字范围较小），基数排序效率很高；但如果数字范围很大（如 2^{32}），k 值较大，效率可能不如 $O(n \log n)$ 的比较排序算法。空间复杂度为 $O(n + r)$，其中 r 是基数（十进制下为 10），因为需要额外的桶空间来存储 n 个元素。稳定性方面，基数排序是稳定的排序算法，这得益于每一轮的分发和收集过程都保持了元素的相对顺序，这对于多关键字排序至关重要。

17　进阶讨论与变体

除了基本的 LSD 实现，基数排序还有多种变体和扩展应用。最高位优先（MSD）基数排序从最高位开始处理，采用递归分治策略，适用于数据分布不均匀的场景，但可能需要额外处理空桶问题。对于负数处理，基本实现无法直接适用，因为负数的位表示不同。一种解决方案是将数组分割为负数和正数两部分，分别进行基数排序：对负数取绝对值排序后再反转顺序，最后与正数部分合并。另一种方法是使用偏移量将所有数转换为非负数后再排序。此外，基数排序可以扩展至其他数据类型，例如字符串或日期。对于字符串，可以将其视为字符序列，按位处理实现字典序排序；对于日期，可以分解为年、月、日等部分，逐部分排序。这些扩展体现了基数排序思想的通用性，关键在于将数据抽象为固定位的关键码序列。回顾基数排序的核心，它通过「按位分配、稳定收集」的方式，实现了高效的非比较排序。其优势包括线性时间复杂度（在 k 较小的情况下）、稳定性以及对整数排序的高效性。然而，基数排序也有局限性：它通常仅适用于整数或具有固定位键的数据类型，需要额外的内存空间，且当 k 值较大时效率可能下降。应用场景广泛，例如电话号码排序、身份证号排序或日期排序，这些场景中关键码由多位组成且位数固定。通过本文的讲解，读者应能掌握基数排序的基本原理和实现，为实际开发提供参考。

18　互动与思考

鼓励读者动手实践，尝试用其他编程语言实现基数排序，或扩展代码以处理负数情况。思考题包括：如果使用不稳定的子排序方法，会发生什么？例如，假设在分发过程中不保持桶内顺序，可能导致前一轮排序结果被破坏，从而无法保证最终有序。另一个思考题是如何修改算法对字符串数组进行字典序排序？这可以通过将字符串视为字符序列，按字符的 ASCII 值逐位处理来实现。通过这些互动，读者可以加深对基数排序的理解，并探索其更多可能性。

第 V 部

动态类型系统的设计原理与实现机制

杨其臻

Oct 18, 2025

想象一下，我们需要编写一个简单的函数来计算两个数的最大值。在静态类型语言如 Java 中，代码可能如下所示：

```
public static int max(int a, int b) {
   return a > b ? a : b;
}
```

而在动态类型语言如 Python 中，同样的逻辑可以写成：

```
def max(a, b):
   return a if a > b else b
```

表面上，Python 版本更简洁，但背后隐藏着一个根本性问题：为什么 Python 函数能处理不同类型的参数，比如整数、浮点数甚至字符串？这种灵活性源于动态类型系统的核心设计 —— 它将类型检查从编译时推迟到运行时。动态类型并非「没有类型」，而是构建在以名字、行为和运行时环境为核心的哲学基础上。本文将带领读者从基本概念出发，逐步深入动态类型系统的设计原理与实现机制，揭示其灵活性与性能背后的精巧权衡。

19 第一部分：基石篇 - 动态类型系统核心概念解析

在探讨动态类型系统之前，我们首先需要理解「类型」的本质。类型可以定义为一系列值及其允许操作的集合，例如整数类型包含所有整数值，并支持加法、比较等操作。静态类型与动态类型的根本分歧在于类型检查的时机：静态类型在编译时进行，而动态类型在运行时进行。类型声明的方式也各不相同，静态类型通常要求显式声明，而动态类型则依赖于隐式推断。

动态类型系统的核心特征之一是「变量无类型，值有类型」。以 Python 代码 `x = 5; x = hello` 为例，变量 `x` 本身并不绑定固定类型，它只是一个名字，可以先后指向整数 `5` 和字符串 `hello`。每个值在内部都携带自己的类型信息，系统在运行时根据这些信息执行操作。另一个关键概念是「鸭子类型」，它强调对象的行为而非继承关系。如果某个对象拥有所需的方法或属性，它就被视为合适的类型，例如一个对象只要有 `quack` 方法，就可以被当作鸭子处理。这自然引出了「运行时多态」，即方法调用的具体实现在运行时根据对象的实际类型决定。此外，动态类型系统还支持高度的灵活性和元编程能力，允许在运行时动态创建或修改类型结构，例如通过 Python 的装饰器或 Ruby 的 `method_missing` 机制。

20 第二部分：原理篇 - 动态类型系统的设计哲学

动态类型系统的设计目标主要围绕开发效率与表现力。通过减少类型声明的负担，开发者可以快速进行原型开发，代码更简洁，更贴近人类思维流程。例如，一个通用的 `max` 函数可以处理任何可比较的类型，无需为每种类型编写重复代码。这种泛用性还促进了元编程的基石，使得动态代码生成和修改成为可能，如 Python 的装饰器能在运行时增强函数行为。

核心设计原理聚焦于名字、绑定与作用域。名字绑定机制描述了变量名如何在不同时间点与不同类型或值的对象关联。例如，在 Python 中，赋值语句 `x = 10` 将名字 `x` 绑定到整数对象 `10`，后续赋值 `x = text` 会重新绑定到字符串对象。这种绑定关系通过命名空间和环境模型管理，环境作为运行时查找变量值的依据，确保了动态查找的可行性。

动态类型系统的「动态」特性主要体现在延迟绑定上。方法绑定在调用时才查找具体实现，例如在 obj.method() 中，解释器在运行时根据 obj 的实际类型解析 method。类似地，类型检查也延迟到操作执行时，如表达式 a + b 只有在运行时才检查操作数类型是否支持加法操作，如果类型不匹配，则抛出异常。这种延迟机制赋予了系统极大的灵活性，但也带来了运行时开销。

21 第三部分：实现篇 - 揭开运行时类型系统的面纱

动态类型系统的实现依赖于精巧的数据结构和算法。首先，核心数据结构是如何表示一个「动态值」。通常，系统使用标签联合体，每个值包含一个类型标签和一个值载荷。例如，在 C 语言中模拟动态类型值，可以这样实现：

```
typedef struct {
   enum { INT, FLOAT, STR } tag;
   union {
      int int_val;
      double float_val;
      char* str_val;
   } payload;
} DynamicValue;
```

在这个结构中，tag 字段标识值的类型（如整数、浮点数或字符串），而 payload 联合体存储实际数据。当处理一个动态值时，系统通过检查 tag 来决定如何操作 payload，例如在加法运算中，根据标签选择整数加法或字符串连接函数。这种设计允许单一变量存储多种类型，但需要额外的内存和检查开销。

方法分派是动态类型系统的关键魔法。当调用 obj.method(arg) 时，系统必须找到正确的函数执行。在基于类的语言如 Python 中，这通常通过虚方法表实现：每个类有一个 vtable，存储方法指针，调用时根据对象的类查找 vtable。在基于原型的语言如 JavaScript 中，则通过字典查找直接在对象的属性中搜索方法。为了优化性能，系统使用内联缓存技术，缓存上一次查找的结果，如果下次调用类型相同，则直接使用缓存，避免重复查找。例如，V8 引擎通过这种机制大幅提升 JavaScript 方法调用速度。

运行时的类型检查与转换机制确保了操作的安全性。以表达式 a + b 为例，解释器或虚拟机首先检查 a 和 b 的类型标签，然后根据标签查找对应的函数（如 int_add 或 str_concat）。如果找不到匹配函数，则抛出类型错误如 TypeError。在弱类型语言如 JavaScript 中，还涉及隐式类型转换，例如 1 + 1 会转换为字符串连接，结果为 11，这通过内部检查类型并应用转换规则实现。

对象模型的内部表示因语言而异。在基于类的模型如 Python 中，对象通常包含 __dict__ 字典存储属性，和 __class__ 指针指向类对象，实现继承和方法解析。在基于原型的模型如 JavaScript 中，对象通过原型链进行属性查找，每个对象可能有一个原型指针，形成链式结构。这些设计确保了动态属性访问和修改的高效性。

22 第四部分：权衡与进化篇 - 代价与未来

动态类型系统虽然灵活，但也带来显著代价。性能开销是主要问题，类型标签、动态分派和运行时检查增加了额外成本，可能导致执行速度慢于静态类型语言。可靠性方面，类型错误在运行后期才暴露，需要更全面的测试覆盖，如单元测试和集成测试，以捕获潜在问题。工具支持与重构也面临挑战，IDE 的智能提示和自动重构功能不如静态语言精准，因为缺乏编译时类型信息。在大型长期项目中，类型的缺失可能增加代码理解成本，影响可维护性。

为应对这些挑战，现代优化技术和混合趋势不断涌现。即时编译技术如 PyPy、LuaJIT 和 V8 引擎通过自适应优化消除动态开销：JIT 编译器在运行时分析代码，推测类型并生成特化机器码，从而提升性能。例如，V8 引擎使用隐藏类和内联缓存优化 JavaScript 执行。

渐进式类型是另一重要趋势，允许在动态类型语言中可选地添加静态类型注解。Python 的 Type Hints、TypeScript 和 MyPy 工具实现了这一理念，在保留动态灵活性的同时引入静态检查优势，例如在 Python 中定义函数时添加类型注解：

```
def greet(name: str) -> str:
    return "Hello, " + name
```

这里，类型注解 `: str` 和 `-> str` 提供了编译时检查的可能，而运行时仍保持动态行为。此外，语言如 Python 3.11+ 通过专项优化进一步利用类型信息提升解释器效率。

23 结论

动态类型系统通过将类型信息与值绑定，并在运行时进行管理和检查，实现了极高的灵活性和开发效率。其核心实现依赖于标签值、环境模型和动态分派机制，这些设计使得系统能够适应快速变化的开发需求。然而，我们必须辩证看待这一技术：动态类型与静态类型是不同哲学下的产物，各自适用于不同场景。没有银弹，动态类型在原型开发和脚本任务中表现出色，而静态类型在大型系统和高性能计算中更具优势。展望未来，动态类型语言不会消失，而是通过与静态类型思想的融合不断进化，在 JIT 编译和渐进式类型等技术的推动下，持续寻求性能、可靠性和开发效率的最佳平衡点。