c13n #32

c13n

2025 年 11 月 19 日

第 I 部

深入理解并实现基本的自动微分（Automatic Differentiation）算法

杨其臻

Sep 18, 202

自动微分（Automatic Differentiation，AD）是现代深度学习框架的核心技术，它使得梯度计算变得高效且精确。本文将带领您从第一性原理出发，深入探讨自动微分的两种核心模式：前向模式与反向模式。我们将从数学基础讲起，通过 Python 代码一步步实现一个基于计算图的反向模式自动微分微型库，并理解其如何高效地计算梯度。文章将避免使用图片和列表，专注于文字描述和代码解读，以确保内容的清晰性和专业性。
在机器学习和深度学习中，模型的训练过程本质上是通过梯度下降算法来优化参数。梯度计算是关键步骤，但传统方法存在显著局限性。手动求导虽然精确，但耗时且容易出错，尤其是当模型结构变化时，需要重新推导公式。数值微分（如有限差分法）虽然自动化，但计算成本高，且存在精度问题，如舍入误差和截断误差，无法达到机器精度。自动微分结合了数值微分的自动化性和符号微分的精确性，能够高效计算梯度，且精度极高。本文的目标是抛开大型框架如 PyTorch 或 TensorFlow，从零实现一个简单的自动微分引擎，以透彻理解其工作原理。

1 自动微分的基本思想

自动微分的核心思想是将任何复杂函数分解为一系列基本初等函数（Primitive Functions）的组合，例如加法、乘法、指数函数和对数函数等。这个过程可以通过计算图（Computational Graph）来表示，其中节点代表中间变量或值，边代表基本运算操作。计算图是一个有向无环图（DAG），它清晰地展示了函数计算的依赖关系。例如，函数 `f(x, y) = exp(x) + x * y` 可以分解为多个步骤：首先计算 `exp(x)`，然后计算 `x * y`，最后将两者相加。数学上，自动微分依赖于链式法则（Chain Rule），这是多元微积分中的基本原理，用于计算复合函数的导数。链式法则允许我们将梯度计算分解为每个基本操作的局部梯度乘积，从而高效地传播梯度。

2 两种模式的深度解析

自动微分主要有两种模式：前向模式和反向模式。每种模式有其适用场景和特点。

2.1 前向模式自动微分

前向模式自动微分从输入变量开始，沿着计算图向前推进，同时计算当前变量对某一个输入变量的导数。这种模式的核心是二元数（Dual Number）理论，二元数将函数值和高阶微分的计算合并在一起，通过扩展数字系统来同时跟踪值和导数。前向模式的特点是适合输入变量少、输出变量多的情况，因为计算梯度的时间与输入维度成正比。例如，对于一个函数 `f(x, y)`，前向模式会逐个计算对每个输入的偏导数。在实际计算中，我们初始化输入变量的导数为 1（对于自身）或 0（对于其他），然后逐步应用链式法则向前传播。

2.2 反向模式自动微分

反向模式自动微分是本文的重点，因为它更适用于深度学习场景，其中输入维度高（如大量参数），输出维度低（如损失函数为标量）。反向模式分为两个阶段：首先进行前向计算（Forward Pass）得到所有中间变量的值，然后从最终输出开始，反向遍历计算图，应用链式法则计算输出对所有输入变量的导数。关键概念是雅可比向量积（Jacobian-vector

Product)，它允许我们高效地累积梯度。反向模式的特点是计算梯度的时间与输出维度成正比，这使得它在高维输入情况下非常高效。反向传播（Backpropagation）算法是反向模式的一个特例，广泛应用于神经网络训练。在反向过程中，每个节点接收来自后续节点的梯度，并将其乘以局部雅可比矩阵，然后累加到父节点的梯度上。

3 动手实现：构建一个微型自动微分库

现在，我们将动手实现一个基于反向模式的自动微分微型库。这个库的核心是一个 Tensor 类，它存储数据、梯度、父节点和操作信息。我们将重载基本运算符来实现前向计算和反向传播。

3.1 设计核心类：Tensor 类

首先，我们定义 Tensor 类，它具有以下属性：data 存储数值，grad 存储梯度，_prev 存储父节点（用于构建计算图），_op 存储产生该节点的操作。此外，我们实现 backward() 方法来触发反向传播。

```
class Tensor:
   def __init__(self, data, _children=(), _op=''):
      self.data = data
      self.grad = 0.0
      self._prev = set(_children)
      self._op = _op
      self._backward = lambda: None

   def backward(self):
      topo = []
      visited = set()
      def build_topo(v):
         if v not in visited:
            visited.add(v)
            for child in v._prev:
               build_topo(child)
            topo.append(v)
      build_topo(self)
      self.grad = 1.0
      for v in reversed(topo):
         v._backward()
```

在这个代码中，Tensor 类初始化时设置数据、梯度、父节点和操作。backward 方法使用深度优先搜索（DFS）构建拓扑排序，然后反向遍历节点，调用每个节点的 _backward 方法来计算梯度。拓扑排序确保我们以正确的顺序处理节点，避免循环依赖。

3.2 实现基本运算操作

接下来，我们重载基本运算符，如加法、乘法、指数函数等。每个操作需要实现前向计算和反向传播规则。

```
def add(self, other):
    other = other if isinstance(other, Tensor) else Tensor(other)
    out = Tensor(self.data + other.data, (self, other), '+')
    def _backward():
        self.grad += out.grad
        other.grad += out.grad
    out._backward = _backward
    return out

Tensor.__add__ = add
```

加法操作的前向计算简单地将两个张量的数据相加。反向传播规则是：梯度从输出节点 out 传播回输入节点 self 和 other，由于加法操作的导数为 1，所以直接将 out.grad 累加到输入节点的梯度上。这体现了链式法则的应用。

```
def mul(self, other):
    other = other if isinstance(other, Tensor) else Tensor(other)
    out = Tensor(self.data * other.data, (self, other), '*')
    def _backward():
        self.grad += other.data * out.grad
        other.grad += self.data * out.grad
    out._backward = _backward
    return out

Tensor.__mul__ = mul
```

乘法操作的前向计算将数据相乘。反向传播规则更复杂：对于 self，梯度是 other.data * out.grad，因为乘法的偏导数是另一个变量的值；同样对于 other，梯度是 self.data * out.grad。这确保了梯度正确传播。

类似地，我们可以实现其他操作，如指数函数。

```
def exp(self):
    out = Tensor(math.exp(self.data), (self,), 'exp')
    def _backward():
        self.grad += out.data * out.grad
    out._backward = _backward
    return out

Tensor.exp = exp
```

指数函数的前向计算使用 math.exp。反向传播规则是：梯度是输出值乘以输出梯度，因为指数函数的导数是其自身。这通过 out.data * out.grad 实现。

3.3 核心引擎：反向传播算法

反向传播算法在 backward 方法中实现。它首先构建拓扑排序来确保节点按依赖顺序处理，然后从输出节点开始，梯度初始化为 1.0（因为输出对自身的导数为 1），反向调用每个节点的 _backward 方法。这个过程应用链式法则，将梯度累加到父节点。

3.4 代码演示与测试

让我们测试这个微型库。例如，计算函数 f(x) = exp(x) + x * x 在 x=2 处的梯度。

```
x = Tensor(2.0)
y = x.exp() + x * x
y.backward()
print(x.grad) # 应该输出 exp(2) + 2*2 的导数，即 exp(2) + 4
```

在这个测试中，我们创建张量 x，计算 y，然后调用 backward。梯度计算应该正确，我们可以与手动计算对比：f'(x) = exp(x) + 2*x，在 x=2 时，值为 exp(2) + 4，约等于 7.389 + 4 = 11.389。我们的库应该输出类似值。

4 现代框架中的自动微分

在现代深度学习框架中，自动微分有静态图和动态图两种实现方式。静态图（如 TensorFlow 1.x）先定义计算图再执行，图是固定的；动态图（如 PyTorch 和 TensorFlow 2.x Eager Mode）边定义边执行，图是即时构建的。我们的实现属于动态图模式。现代框架还进行了大量优化，如内核融合、高效内存管理和数据结构，以提高性能和可扩展性。

本文深入探讨了自动微分的核心思想、两种模式的区别，以及如何从零实现一个反向模式自动微分库。通过代码实现，我们理解了梯度计算如何通过计算图和链式法则高效完成。自动微分是深度学习的基础，掌握其原理有助于更深入地理解模型训练过程。展望未来，可以扩展实现高阶导数、处理控制流（如 if 和 while 语句），或应用于随机梯度估计等领域。鼓励读者在此基础上继续探索，以加深理解。

5 参考资料与延伸阅读

建议阅读一些经典资源，如 Baydin et al. 的论文「Automatic Differentiation in Machine Learning: a Survey」，以及书籍如「Deep Learning」by Ian Goodfellow et al.。在线教程如 PyTorch 官方文档也提供了丰富资料。

第 II 部

深入理解并实现基本的蒙特卡洛方法（Monte Carlo Method）

杨其臻

Sep 19, 2025

蒙特卡洛方法是一种基于随机抽样的数值计算技术，用于解决复杂积分、概率模拟和优化等问题。本文将从理论入手，讲解其核心原理，并通过 Python 实现几个经典案例，帮助读者从实践角度深入理解这一方法。
蒙特卡洛方法得名于摩纳哥的蒙特卡洛赌场，源于 20 世纪 40 年代曼哈顿计划中冯·诺依曼和乌拉姆等人的工作。其核心思想是利用随机性来解决确定性问题，例如，通过投掷飞镖来估算圆周率 π：在一个单位圆的外接正方形内随机投点，统计落在圆内的点的比例，从而近似 π 的值。这种方法直观易懂，因为它是基于「用频率逼近概率，用样本均值逼近理论积分」的基本统计原理。蒙特卡洛方法的主要优点包括实现简单、不受问题维度限制，能够有效克服「维度灾难」；但其缺点在于计算精度依赖于采样数量，收敛速度较慢，为 $O(1/\sqrt{N})$，这意味着需要大量样本才能获得高精度结果。本文将逐步引导读者从理论基础到实际编码，全面掌握蒙特卡洛方法的应用。

6 蒙特卡洛方法的理论基石

蒙特卡洛方法的有效性建立在两大统计定律之上：大数定律和中心极限定理。大数定律指出，当试验次数足够多时，样本的平均值将无限接近理论的期望值，这为蒙特卡洛方法的收敛性提供了根本保证。例如，如果我们通过随机采样来估计一个积分，样本均值会随着采样数量的增加而趋近于真实值。中心极限定理则说明，大量独立随机变量的和近似服从正态分布，这使得我们能够估计蒙特卡洛方法的误差和置信区间；具体来说，误差正比于 $1/\sqrt{N}$，其中 N 是样本数量。从数学角度，蒙特卡洛方法将积分问题转化为期望计算：假设我们需要计算积分 $I = \int_a^b f(x)dx$，这可以视为随机变量 $f(X)$ 的期望 $E[f(X)]$，其中 X 在区间 [a, b] 上均匀分布。通过生成大量随机样本 X_i，我们可以用样本均值来近似积分：$I \approx (b-a) \cdot \frac{1}{N} \sum_{i=1}^{N} f(X_i)$。这种转换使得复杂积分变得可计算，只需通过随机采样即可。

7 动手实战 - Python 实现经典案例

在开始编码前，我们需要导入必要的 Python 库，包括 numpy 用于数值计算和随机数生成，以及 matplotlib 用于可视化。我们将通过三个案例来演示蒙特卡洛方法的实际应用。

7.1 案例一：计算圆周率 π

计算圆周率 π 是蒙特卡洛方法的经典示例。问题基于几何原理：单位圆的面积是 π，而外接正方形的面积是 4，因此圆的面积与正方形面积之比为 π/4。算法思路是在正方形区域内随机生成点，并统计落在圆内的点的数量，从而估算 π 值。具体地，我们生成在 [-1, 1] x [-1, 1] 范围内的随机点 (x, y)，计算每个点到原点的距离 $\sqrt{x^2+y^2}$，如果距离小于或等于 1，则点落在圆内。最终，π 的估计值为 4 乘以圆内点数与总点数的比值。
以下是 Python 代码实现。首先，我们导入库并设置参数，如样本数量 N。然后，使用 numpy 的 random.uniform 函数生成均匀分布的随机点。接着，通过向量化操作计算每个点的距离，并统计圆内的点数。最后，计算 π 的估计值并输出结果。

```
import numpy as np
import matplotlib.pyplot as plt
```

```
N = 100000 # 样本数量
x = np.random.uniform(-1, 1, N)
y = np.random.uniform(-1, 1, N)
distance = np.sqrt(x**2 + y**2)
inside_circle = distance <= 1
pi_estimate = 4 * np.sum(inside_circle) / N
print(f"估计的␣π␣值␣:␣{pi_estimate}")
```

代码解读：这段代码首先生成 N 个在 [-1, 1] 区间内的随机点 x 和 y，然后计算每个点到原点的欧几里得距离。条件判断 inside_circle 是一个布尔数组，表示点是否在圆内。求和操作 np.sum(inside_circle) 统计圆内点的数量，最终乘以 4 得到 π 的估计。由于随机性，每次运行结果会略有不同，但随着 N 增大，估计值会趋近于真实 π。为了分析收敛性，我们可以计算误差并绘制图表，例如误差随 N 变化的曲线，验证 $O(1/\sqrt{N})$ 的收敛速度。

7.2 案例二：计算定积分

蒙特卡洛方法可用于计算复杂定积分，例如积分 $I = \int_0^2 \left(\sin(x) + \cos(x^2)\right) dx$。算法思路是在积分区间 [0, 2] 上均匀采样，生成随机点 x_i，然后计算函数值的平均值，并乘以区间长度 (2 - 0) = 2 来估计积分值。这种方法简单直接，但精度依赖于采样数量。

Python 代码实现如下。我们定义被积函数 f(x)，然后生成均匀分布的随机样本，计算函数值的均值，并乘以区间长度得到积分估计。

```
def f(x):
    return np.sin(x) + np.cos(x**2)

a = 0
b = 2
N = 100000
x_samples = np.random.uniform(a, b, N)
f_values = f(x_samples)
integral_estimate = (b - a) * np.mean(f_values)
print(f"积分估计值␣:␣{integral_estimate}")
```

代码解读：这里，np.random.uniform(a, b, N) 生成在 [a, b] 区间内的 N 个随机点。函数 f(x) applied 到这些点上，得到 f_values 数组。np.mean(f_values) 计算样本均值，然后乘以区间长度 (b - a) 来近似积分。与 scipy.integrate.quad 等数值积分方法对比，蒙特卡洛方法在低维问题中可能效率较低，但它的优势在于高维积分，其中传统方法会遇到困难。进阶地，我们可以讨论重要性采样来优化方差，例如通过调整采样分布来聚焦于函数值较大的区域，从而提高效率。

7.3 案例三：赌博游戏模拟 - 赌徒的破产

赌徒的破产问题是一个概率模拟示例，演示蒙特卡洛方法在随机过程中的应用。问题描述：一个赌徒有初始本金，每次赌博以概率 p 赢 1 元，概率 q = 1 - p 输 1 元，目标是模拟他最

终破产（本金为 0）或达到目标金额的过程。算法思路是通过多次独立实验（即模拟多个赌徒的赌博过程），统计破产的次数，从而估计破产概率。每次实验模拟一个赌徒的序列直到破产或成功。

Python 代码实现中，我们定义一个函数来模拟单个赌徒的命运，然后循环多次实验来统计破产概率。

```
def simulate_gambler(initial_capital, target, p):
    capital = initial_capital
    while capital > 0 and capital < target:
        if np.random.rand() < p:
            capital += 1
        else:
            capital -= 1
    return capital == 0 # 返回是否破产

initial_capital = 10
target = 20
p = 0.5
num_experiments = 10000
bankrupt_count = 0
for _ in range(num_experiments):
    if simulate_gambler(initial_capital, target, p):
        bankrupt_count += 1
bankrupt_probability = bankrupt_count / num_experiments
print(f"破产概率 : {bankrupt_probability}")
```

代码解读：simulate_gambler 函数使用 while 循环来模拟赌博过程，直到本金为 0（破产）或达到目标金额。np.random.rand() 生成 [0,1) 之间的随机数，用于模拟赢的概率 p。主循环中，我们进行 num_experiments 次实验，统计破产次数，并计算破产概率。这种方法通过大量随机实验来逼近理论概率，体现了蒙特卡洛模拟的核心。可视化方面，我们可以绘制破产概率随初始本金或 p 变化的曲线，例如使用 matplotlib 的 plot 函数来展示趋势。

8 蒙特卡洛方法的优化与扩展方向

尽管基础蒙特卡洛方法简单有效，但其收敛速度较慢，因此优化方差缩减技术至关重要。重要性采样是一种常见优化，通过调整采样分布来更频繁地对重要区域采样，从而减少方差。对偶变量法利用随机数的对称性来抵消方差，例如在积分计算中使用配对样本。控制变量法则用一个已知期望的变量来修正估计量，提高精度。这些技术都能显著降低计算成本，使蒙特卡洛方法更高效。扩展应用领域包括强化学习中的蒙特卡洛控制、金融工程中的期权定价、计算机图形学中的路径追踪渲染，以及物理学中的粒子模拟。这些高级应用展示了蒙特卡洛方法的广泛适用性和强大能力。

本文回顾了蒙特卡洛方法的核心原理，包括大数定律和中心极限定理的理论基础，并通过

Python 实战演示了其在圆周率计算、积分估算和概率模拟中的应用。蒙特卡洛方法的优势在于其通用性和简单性，但局限性在于收敛速度慢和有时需要大量计算。未来，读者可以探索更先进的方差缩减技术或应用于特定领域如机器学习。鼓励读者在实践中尝试这一方法，因为它就像一把「瑞士军刀」，能灵活解决各种复杂问题。深入学习方向包括马尔可夫链蒙特卡洛（MCMC）等高级主题。

第 III 部

使用 Haskell 解决逻辑谜题的编程实践

黄梓淳

Sep 20, 2025

逻辑谜题一直是考验人类推理能力的经典方式，例如著名的「骑士与骑士」谜题：在一个岛上，居民要么总是说真话（骑士），要么总是说假话（骑士），而你需要通过一系列陈述来推断真相。这类问题的核心在于定义约束条件，而不是描述具体的解决步骤，这正是声明式编程的用武之地。Haskell 作为一种纯函数式编程语言，以其独特的特性成为解决逻辑谜题的理想工具。Haskell 的纯函数性确保了推理过程的确定性，无副作用干扰；强大的类型系统帮助精确建模问题域，让非法状态无法表示；卓越的列表处理能力允许轻松生成和过滤解空间；而惰性求值则能高效处理甚至无限的可能性，只计算所需部分。通过 Haskell，我们可以像数学家一样定义问题，并让编译器自动找出答案。

9 热身：Haskell 武器库速览

在深入解决逻辑谜题之前，让我们快速回顾 Haskell 中一些关键工具。列表推导式是生成和过滤解的强大工具，其语法类似于数学中的集合表示法，例如 `[x | x <- [1..10], even x]` 会生成所有偶数，这让我们能够直观地表达解空间。Maybe 类型用于处理可能不存在的值，例如 `Just 5` 表示有值，而 `Nothing` 表示无值，这在检查约束时非常有用。Bool 类型表示真值，用于谓词函数。元组允许将多个值组合在一起，例如 `(1, hello)` 可以表示一个简单的实体。模式匹配则用于解构数据，例如在函数定义中匹配特定模式以执行不同逻辑。这些工具组合起来，为我们提供了声明式解决问题的基础。

10 实战演练：解构一个经典谜题

我们选择爱因斯坦逻辑谜题（又称「谁养鱼」谜题）作为案例，这是一个涉及多个属性（如国籍、颜色、饮料、宠物和香烟品牌）的复杂推理问题。解决过程分为几个步骤：首先，将谜题转化为规格说明，列出所有实体和线索；其次，用 Haskell 数据类型建立模型；然后，将约束条件编码为函数；最后，生成并筛选解空间。

第一步是定义数据类型来表示各种属性。我们使用代数数据类型来确保类型安全。例如：

```haskell
data Nationality = Norwegian | Englishman | Swede | Dane | German deriving (Show, Eq)
data Color = Red | Green | White | Yellow | Blue deriving (Show, Eq)
data Drink = Tea | Coffee | Milk | Beer | Water deriving (Show, Eq)
data Pet = Dog | Birds | Cats | Horse | Fish deriving (Show, Eq)
data Cigarette = PallMall | Dunhill | Blend | BlueMaster | Prince deriving (Show, Eq)
```

这里，`deriving (Show, Eq)` 允许这些类型可显示和比较。每个数据类型代表谜题中的一个属性类别，确保我们只能使用有效的值。

接下来，我们定义表示一个房子的类型，通常使用元组或记录，但为了简洁，我们使用元组。一个房子由五个属性组成：国籍、颜色、饮料、宠物和香烟品牌。

```haskell
type House = (Nationality, Color, Drink, Pet, Cigarette)
```

一个解决方案是五个房子的列表，代表一排房子。

```haskell
type Solution = [House]
```

现在，我们需要编码所有线索作为约束函数。每个函数类型为 Solution → Bool，检查一个可能解是否满足条件。例如，第一条线索是「挪威人住在第一间房子」，我们可以这样写：

```
constraint1 :: Solution -> Bool
constraint1 ((Norwegian, _, _, _, _) : _) = True
constraint1 _ = False
```

这里，我们使用模式匹配来检查列表的第一个元素是否是挪威人。_ 是通配符，表示我们忽略其他属性。

另一条线索是「英国人住在红房子里」，我们需要检查整个列表中是否有这样一个房子。

```
constraint2 :: Solution -> Bool
constraint2 solution = any (\(nat, col, _, _, _) -> nat == Englishman
    ↪  && col == Red) solution
```

any 函数遍历列表，检查是否存在满足条件的元素。lambda 函数解构每个房子，比较国籍和颜色。

对于关系约束，如「绿房子在白房子的左边」，我们需要比较位置。

```
constraintLeftOf :: Solution -> Bool
constraintLeftOf houses = or [ greenIndex + 1 == whiteIndex | (
    ↪ greenIndex, (_, Green, _, _, _)) <- indexed, (whiteIndex, (_,
    ↪ White, _, _, _)) <- indexed ]
  where indexed = zip [0..] houses
```

这里，我们使用列表推导式和 zip 来获取索引，然后检查绿房子索引加一是否等于白房子索引。or 确保至少一对满足条件。

最后，我们生成所有可能解并应用约束。由于解空间巨大，我们利用列表推导式和惰性求值。

```
allHouses :: [House]
allHouses = [ (nat, col, drink, pet, cig) | nat <- [Norwegian,
    ↪ Englishman, Swede, Dane, German],
                                  col <- [Red, Green, White, Yellow,
                                      ↪ Blue],
                                  drink <- [Tea, Coffee, Milk, Beer,
                                      ↪ Water],
                                  pet <- [Dog, Birds, Cats, Horse, Fish
                                      ↪ ],
                                  cig <- [PallMall, Dunhill, Blend,
                                      ↪ BlueMaster, Prince] ]

solutions :: [Solution]
solutions = [ [h1, h2, h3, h4, h5] | h1 <- allHouses, h2 <- allHouses,
    ↪  h3 <- allHouses, h4 <- allHouses, h5 <- allHouses,
                             constraint1 [h1, h2, h3, h4, h5],
```

```
                                  constraint2 [h1, h2, h3, h4, h5],
                                  -- 应用所有其他约束
                                  constraintLeftOf [h1, h2, h3, h4, h5] ]
```

这个列表推导式生成所有可能的房子排列，然后逐个应用约束函数过滤。由于 Haskell 的惰性求值，它只会在需要时计算，避免不必要的开销。

11 优化与思考：超越暴力破解

虽然上述暴力枚举方法正确，但效率低下，因为解空间随属性数量指数级增长。Haskell 提供了多种优化策略。首先，尽早过滤：在列表推导式中尽早应用约束，减少后续组合。例如，我们可以在生成房子时就应用部分约束，而不是生成所有后再过滤。其次，使用高级抽象如 State Monad 或 Logic Monad：库如 `logict` 提供智能回溯，类似于 Prolog 的搜索机制，能更高效地探索解空间。例如，Logic Monad 允许我们定义非确定性计算，并自动处理分支。第三，对称性剪枝：通过消除等价解（如颜色标签可互换）来减少搜索空间。尽管优化重要，但核心价值在于思维模式：我们声明问题是什么，而非如何解决，计算机负责繁琐搜索。这体现了函数式编程的优雅和抽象能力。

12 举一反三：其他谜题与模式

类似方法可应用于其他逻辑谜题。例如，数独问题可以用 Haskell 建模为网格，约束为行、列和子网格的数字唯一性；列表推导式可生成可能数字组合，并过滤无效解。八皇后问题则可通过生成皇后位置排列，并应用不攻击约束来解决。电路验证中，Haskell 的类型系统可确保连接正确，而约束函数检查逻辑一致性。共通模式是：定义问题域、生成解空间、施加约束、提取解。这种声明式方法不仅限于谜题，还适用于软件规范、测试用例生成和配置验证，展示 Haskell 在复杂问题解决中的通用性。

13 结论

通过 Haskell 解决逻辑谜题，我们展示了函数式编程的思维艺术：高度抽象、接近于问题描述的方式。这种方法强调定义而非执行，提升了代码的可读性和可维护性。价值 beyond 谜题，它提供了一种强大的问题解决框架，适用于多个领域。鼓励读者尝试用 Haskell 解决自己喜爱的谜题，或在社区中分享经验，进一步探索函数式编程的潜力。

第 IV 部

深入理解并实现基本的 JSON 解析器

黄京

Sep 21, 2025

14 副标题：抛开现成的库，亲手打造一个解析器，彻底掌握 JSON 的本质。

JSON（JavaScript Object Notation）作为现代数据交换的事实标准，几乎无处不在。从 Web API 的响应到配置文件的存储，JSON 以其轻量级和易读性赢得了广泛的应用。我们日常开发中经常使用如 Python 的 `json.loads()` 或 JavaScript 的 `JSON.parse()` 来解析 JSON 数据，但这些现成库的背后机制却往往被忽略。超越简单的导入和使用，亲手实现一个 JSON 解析器，不仅能帮助我们深入理解编译原理的基础知识如语法、词法和状态机，还能在遇到非标准或自定义数据格式时提供自主解决问题的能力。此外，这个过程极大地锻炼了编程技能、对细节的把握和调试能力。本文的目标是使用 Python 语言从零实现一个功能完备的 JSON 解析器，并通过官方的 JSONTestSuite 进行测试，以确保其正确性和健壮性。

15 背景知识：JSON 格式规范与解析器概述

JSON 的语法规范基于 RFC 7159，其基本结构包括对象（用花括号 {} 表示）和数组（用方括号 [] 表示），以及基本类型如字符串、数字、布尔值（`true` 或 `false`）和 `null`。重要规则包括键必须用双引号包裹、禁止尾部逗号等。解析 JSON 字符串通常涉及两个核心步骤：词法分析和语法分析。词法分析负责将字符流分解为有意义的词元（Token），类似于将句子拆分成单词；语法分析则根据 Token 流按照语法规则构建内存中的数据结构，如 Python 的字典或列表。整个解析器的架构可以概括为：JSON 字符串输入到词法分析器（Lexer）生成 Token 流，再传递给语法分析器（Parser）输出 Python 对象。

16 第一步：构建词法分析器（Lexer）

词法分析器的核心任务是识别和生成 Token。Token 类型包括结构字符如 {, }, [,], :, ,，以及值类型如 `STRING`, `NUMBER`, `TRUE`, `FALSE`, `NULL`。Lexer 的工作流程是循环遍历输入字符串，使用条件判断和状态机来识别这些 Token。实现过程中，需要处理空白字符（如空格、制表符、换行符）的跳过，以及解析字符串时的转义序列（例如 `\\`, `\`, `\n`, `\uXXXX`）。数字解析涉及识别负号、整数部分、小数部分和指数部分，策略通常是持续读取相关字符后统一用 `float()` 转换。字面量如 `true`, `false`, `null` 则通过匹配关键字来识别。

以下是一个简单的 Lexer 类框架代码示例：

```python
class Lexer:
    def __init__(self, input_string):
        self.input = input_string
        self.position = 0
        self.current_char = self.input[self.position] if self.input
            ↪ else None

    def advance(self):
        self.position += 1
```

```
        if self.position < len(self.input):
            self.current_char = self.input[self.position]
        else:
            self.current_char = None

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def next_token(self):
        # 实现 Token 识别逻辑
        pass
```

在这个代码中，`__init__` 方法初始化输入字符串和当前位置，`advance` 方法移动指针到下一个字符，`skip_whitespace` 用于跳过空白字符。`next_token` 方法是核心，它根据当前字符判断 Token 类型。例如，如果字符是 {，则返回一个表示左花括号的 Token；如果是双引号，则开始解析字符串。字符串解析需要处理转义序列，这是一个难点，因为必须正确识别和转换如 \n 为换行符。数字解析则涉及收集所有数字相关字符（包括符号、小数点、指数），然后使用 `float()` 进行转换，但需要注意错误处理，例如无效数字格式。

17 第二步：构建语法分析器（Parser）

语法分析器采用递归下降解析法，这是一种直观的方法，适合 JSON 这种上下文无关文法。每个语法规则对应一个解析函数。入口函数是 `parse()`，它根据第一个 Token 决定解析为对象或数组。`parse_object()` 函数处理花括号内的键值对，循环读取直到遇到右花括号；`parse_array()` 函数处理方括号内的值列表；`parse_value()` 是核心分发函数，根据当前 Token 类型调用相应的解析函数，如 `parse_string` 或 `parse_number`，或递归调用自身处理嵌套结构。

错误处理是关键部分，例如在预期冒号分隔键值对时却遇到其他 Token，应抛出清晰异常如“Expected ‘:’ after key”。解析器需要与 Lexer 协同工作，确保在解析完一个结构后，Lexer 的位置正确指向下一个 Token。以下是一个 Parser 类的框架代码：

```
class Parser:
    def __init__(self, lexer):
        self.lexer = lexer
        self.current_token = self.lexer.next_token()

    def eat(self, token_type):
        if self.current_token.type == token_type:
            self.current_token = self.lexer.next_token()
        else:
            raise Exception(f"Expected {token_type}, got {self.
```

```
            ↪ current_token.type}")

    def parse(self):
        if self.current_token.type == 'LBRACE':
            return self.parse_object()
        elif self.current_token.type == 'LBRACKET':
            return self.parse_array()
        else:
            return self.parse_value()

    def parse_object(self):
        # 解析对象逻辑
        pass
```

在这个代码中，`__init__` 方法接收 Lexer 实例并获取第一个 Token。`eat` 方法用于消耗预期类型的 Token，如果类型不匹配则抛出错误。`parse` 方法根据当前 Token 类型决定解析方向。`parse_object` 函数会循环读取键值对，每次读取一个字符串键、冒号、值，并处理逗号分隔。递归下降法的优势在于代码结构清晰，易于理解和调试，但需要 careful handling of recursive calls to avoid infinite loops。

18 整合与测试

将 Lexer 和 Parser 整合为一个函数 `my_json_loads(s)`，它接收 JSON 字符串并返回 Python 对象。基础测试用例包括简单 JSON 如 `{key: value}`，应解析为字典；数组如 `[1, true, null]`，应解析为列表。挑战性测试涉及嵌套结构，例如多层对象或数组，以验证递归处理的正确性。错误处理测试包括非法输入如缺少逗号或键名无双引号，解析器应提供有用的错误信息而非崩溃。最后，引入 JSONTestSuite 进行自动化测试，确保解析器符合官方标准，例如处理边缘情况如空字符串或超大数字。

通过实现 JSON 解析器，我们完整经历了从字符串到 Token 再到数据结构的解析过程，加深了对词法分析、语法分析和递归下降法的理解。未来优化方向包括性能提升（如使用生成器惰性产生 Token）、功能扩展（添加自定义参数如 parse_float）和增强鲁棒性（改进错误恢复机制）。鼓励读者以此为基础，探索更复杂格式如 XML 或 TOML 的解析，进一步提升编译原理技能。

19 附录 & 进一步阅读

完整代码可参考 GitHub 仓库示例。参考资源包括 RFC 7159 标准文档、JSONTestSuite 项目以及经典书籍《编译原理》（俗称龙书）的相关章节。这些资料有助于深入理解解析器设计和实现细节。

第 V 部

深入理解并实现基本的 HTTP/3 协议核心机制

马浩琨

Sep 27, 2025

HTTP 协议的演进史是一部不断解决性能瓶颈的奋斗史。HTTP/1.1 时代，每个 TCP 连接只能处理一个请求，导致严重的队头阻塞问题；HTTP/2 引入了多路复用技术，允许在单个连接上并行传输多个请求，但底层仍依赖于 TCP 协议。TCP 本身的队头阻塞和三次握手延迟成为了新的性能瓶颈，使得 HTTP/2 在许多场景下无法充分发挥优势。这种“补丁摞补丁”的困境促使我们需要一个更根本的解决方案。

HTTP/3 的核心理念是弃用传统的 TCP 协议，转而在 UDP 之上构建一个名为 QUIC 的新传输协议。这一变革旨在从根本上降低延迟并提升性能。本文将深入解析 HTTP/3 的核心机制，并探讨如何通过代码实现一个基本的 HTTP/3 交互模型，帮助读者从理论到实践全面掌握这一下一代 Web 协议。

20 基石：QUIC 协议深度解析

QUIC 协议是 HTTP/3 的基石，它本质上是一个位于传输层和应用层之间的“伪传输层”协议。QUIC 的核心优势在于其连接建立机制。与 TCP + TLS 需要 1 到 3 次 RTT（往返时间）不同，QUIC 将加密和传输握手合二为一，首次连接通常仅需 1-RTT，而再次连接时甚至可以实现 0-RTT，这显著降低了延迟。关键概念如 Connection ID（连接 ID）允许连接在不同网络环境（如从 Wi-Fi 切换到 5G）下无缝迁移，避免了传统 TCP 连接因 IP 变化而中断的问题。

另一个重要机制是 QUIC 根除了队头阻塞。在 HTTP/2 中，虽然应用层实现了多路复用，但底层 TCP 流的包丢失会阻塞所有流。QUIC 在协议层原生支持多路复用的流，每个流独立传输，数据包丢失只影响所属流，其他流不受影响。同时，QUIC 在每个流内部保证了数据的可靠性和顺序性，类似于 TCP，但流与流之间完全独立，这为高性能传输奠定了基础。

21 HTTP/3 在 QUIC 之上的构建

HTTP/3 作为应用层协议，构建在 QUIC 之上，定义了如何利用 QUIC 的流来传输 HTTP 语义。与 HTTP/2 类似，HTTP/3 使用帧（Frames）来封装数据，如 HEADERS 帧和 DATA 帧，但载体从 TCP 流变为 QUIC 流。这种变化带来了新的挑战，特别是在头部压缩方面。HTTP/2 的 HPACK 依赖于全局有序的头部表，而 QUIC 流的独立性使得这种机制无法直接适用。

HTTP/3 引入了 QPACK 作为头部压缩解决方案。QPACK 使用静态表和动态表两个独立组件，并通过指令流来同步编码方和解码方的动态表状态，从而在无序的流上实现高效压缩。QPACK 的设计考虑了流之间的独立性，避免了全局依赖，确保了压缩效率。HTTP/3 连接的生命周期包括发现、建立和请求/响应阶段。客户端通过 `Alt-Svc` 响应头发现服务器支持 HTTP/3，然后通过 QUIC 握手建立连接。请求和响应通过控制流和双向流处理，每个请求通常分配一个新的流，实现了高效的资源管理。

22 实践：实现一个最简单的 HTTP/3 交互

实现 HTTP/3 交互时，不建议从零开始构建 QUIC 协议，而应使用成熟库如 Cloudflare 的 quiche 或 Node.js 的 `node:http3`。以下通过概念性伪代码演示核心步骤。首先，建立 QUIC 连接是基础，代码示例如下：

```
# 伪代码示例：建立 QUIC 连接
connection = quiche.connect(server_addr, server_cert_validation)
```

这段代码初始化一个 QUIC 连接对象，其中 server_addr 是服务器地址，server_cert_validation 用于证书验证，确保通信安全。QUIC 在连接建立时即集成加密，这与传统 TCP 分离握手和加密的方式不同。

接下来，创建并发送 HTTP/3 请求帧。需要打开一个双向流，并使用 QPACK 压缩头部：

```
# 伪代码示例：发送 HTTP/3 请求
stream_id = connection.open_bidirectional_stream()
headers = [(':method', 'GET'), (':path', '/'), ...]
compressed_headers = qpack.encode(headers)
connection.send_frame(stream_id, HEADERS_FRAME, compressed_headers)
```

这里，open_bidirectional_stream 方法创建一个双向流，流 ID 用于标识。QPACK 编码器将 HTTP 头部（如方法 GET 和路径 /）压缩为二进制格式，然后通过 send_frame 发送 HEADERS 帧。这一步体现了 HTTP/3 如何利用 QUIC 流传输应用层数据。

接收并解析响应时，从流中读取帧并处理：

```
# 伪代码示例：接收 HTTP/3 响应
frames = connection.receive_frames(stream_id)
for frame in frames:
   if frame.type == HEADERS_FRAME:
      headers = qpack.decode(frame.payload)
      status = get_header(headers, ':status')
   elif frame.type == DATA_FRAME:
      body += frame.payload
```

这段代码循环处理接收到的帧，如果是 HEADERS 帧，则使用 QPACK 解码获取状态码和头部；如果是 DATA 帧，则累加响应体。最后，关闭连接以释放资源。整个流程展示了 HTTP/3 如何通过 QUIC 流实现请求-响应交互，其效率源于流的独立性和快速连接建立。

23 HTTP/3 的现状与挑战

目前，HTTP/3 已得到主流浏览器如 Chrome 和 Firefox、云服务商如 Cloudflare 以及 Web 服务器如 Nginx 的支持。其优势包括更低的延迟、更好的弱网性能和连接迁移能力，这些特性使其在实时应用和移动互联网中具有广阔前景。然而，HTTP/3 也面临挑战。中间设备可能错误处理 UDP 流量，导致连接问题；用户态实现 QUIC 可能比内核态 TCP 消耗更多 CPU 资源；协议复杂性增加了调试难度。这些挑战需要在实际部署中通过优化和监控来应对。

HTTP/3 通过将传输层协议上移到用户空间，并用 QUIC 取代 TCP，从根本上解决了延迟和队头阻塞问题。它不仅提升了 Web 性能，还为未来实时应用和物联网奠定了基础。鼓励开发者在项目中尝试 HTTP/3，以充分利用其革新特性。随着技术普及，HTTP/3 有望成为下一代互联网的标准协议。

24 参考资料与延伸阅读

官方文档如 IETF RFC 9114（HTTP/3）和 RFC 9000（QUIC）是深入学习的基础。工具如 Wireshark 支持 QUIC 和 HTTP/3 解密，可用于协议分析。测试网站如 http3.check 提供便捷的验证平台。推荐阅读 Cloudflare 等公司的技术博客，获取实践洞见。