代数效应（Algebraic Effects）的原理与实现

黄梓淳

Sep 07, 2025

1　导言

在编写复杂异步逻辑、异常处理或状态管理时，我们常常面临代码嵌套深厚、难以阅读和维护的问题。例如，回调地狱使得代码层层嵌套，繁琐的 try ／ catch 块增加了冗余，而层层传递的状态则导致逻辑分散。我们不禁思考：能否像写同步代码一样写异步代码？能否简单地「抛出」一个任意复杂的操作（如登录弹窗），并由调用栈上层的某个 handler 来接管？答案是肯定的，这正是代数效应（Algebraic Effects）这一编程语言概念所要解决的问题。代数效应被誉为「下一代异步编程和错误处理模型」，在 React、Koka 等语言和框架中已有深入探索。本文旨在从零开始，深入剖析代数效应的核心思想、运行原理，并探讨其在不同语言中的实现方式，帮助读者不仅知其然，更知其所以然。

2　什么是代数效应？—— 超越 Monad 的优雅控制流

代数效应的核心思想在于将程序中的「效应」与纯计算分离。效应指的是程序除了返回值外可能产生的操作，如抛出异常、发起网络请求或读取环境变量，这些效应是对纯计算的一种「打扰」。代数一词指代效应的组合方式具有代数结构（如可组合性和可交换性），使得多个效应可以优雅地组合在一起。简单来说，代数效应允许函数发起（perform）一个效应，并由其调用者的效应处理程序（handler）来响应该效应，从而将「做什么」与「怎么做」彻底分离。

一个简单的类比是异常处理机制。在异常处理中，throw 类似于 perform，catch 块类似于效应处理程序。然而，关键区别在于，效应处理程序可以携带一个恢复函数（resumption），在处理完效应后，可以选择回到中断的地方继续执行，并注入一个结果。这使得代数效应比异常处理更强大和灵活。

与其他技术相比，代数效应展现出显著优势。与回调函数相比，它避免了回调地狱，保持代码扁平化和可读性。与 Promise 或 Async ／ Await 相比，Async ／ Await 仅是代数效应的一种特例（仅用于异步效应），而代数效应是更通用、更强大的抽象。与 Monad（如 Haskell 中的 IO Monad）相比，Monad 通过类型系统顺序组合效应，需要在语法上进行包装（如 do notation），而代数效应在语法上更轻量，对控制流的操作更直观，对类型系统的侵入性更小。

3　核心原理与运行机制剖析

代数效应的核心操作包括 perform(effect, argument) 和 try...with handler。perform 用于在执行中发起一个效应，而 try...with 用于建立一个效应处理程序的作用域。执行流程的关键在于续延（Continuation）

的概念，续延表示「接下来要做什么」，即当前的执行状态。当函数 f perform 一个效应时，运行时会中断执行，捕获从 perform 点之后直到 try 块结束的续延，并将其包装成一个函数 k。然后，运行时沿着调用栈向上查找能处理该效应的 handler。找到后，将数据和续延 k 传递给 handler。handler 可以自由选择：调用 resume(k, result) 来恢复执行，注入结果；直接返回一个值来终止 try 块；或再次发起其他效应。如果 resume 被调用，续延 k 被执行，函数 f 接收到结果并继续执行。

这种机制的优势在于抽象泄漏最小化，调用者无需关心被调用函数的具体效应；极强的表达能力，可以轻松实现可恢复异常、协作式多任务等模式；以及代码极度简洁，业务逻辑和效应处理逻辑分离。例如，在数学上，续延可以表示为函数 $k : A \to B$，其中 A 是当前状态，B 是未来计算结果，handler 通过操作 k 来实现控制流跳转。

4 如何实现代数效应?

实现代数效应的核心挑战在于如何捕获和管理续延，这通常需要运行时的深度支持。有几种不同的实现策略。策略一是一等续延（First-Class Continuations）。如果语言原生支持捕获当前续延，如 Scheme 的 call／cc，则可以在此基础上构建代数效应。效应处理程序本质上是对续延的捕获和重新调用。例如，在 Scheme 中，可以使用 call／cc 来模拟代数效应，通过捕获续延并将其传递给 handler 来实现恢复。

策略二是生成器函数或协程。Generator 函数可以暂停和恢复执行，这与代数效应中中断和恢复的模式相似。然而，Generator 的 yield 是「单向」的，只能向上返回值，无法像代数效应一样让调用者注入值并恢复。但可以通过一些技巧，如双向通信，来模拟。例如，在 JavaScript 中，使用 Generator 可以部分模拟代数效应，但效果有限，因为它无法完整捕获续延。

策略三是转换与运行时，以 React 为例。React 团队实现了 React Fiber，一个轻量级调用栈和调度器。Babel 转换将组件函数编译成状态机，使用 Generator 或 switch 语句实现。Fiber Reconciler 负责调度这些 Fiber 的执行。当遇到 Hook（效应的具体实现）时，React 能够暂停当前组件的渲染，先去完成效应（如状态更新、发起请求），然后在合适的时机恢复渲染。这是一种巧妙的工程实践，在不支持一等续延的语言中实现类似代数效应的能力。

策略四是原生语言支持。一些研究型语言或编译器原生支持代数效应，如 Koka，由微软研究院开发，将代数效应作为语言核心特性；Unison，一种新兴的分布式编程语言；以及 OCaml，其多态变体和未来计划对效应系统的支持。这些语言通过内置的运行时机制直接处理效应，提供更高效和类型安全的实现。

5 实际案例与代码演示

在实际应用中，代数效应可以用于多种场景。例如，可恢复异常。在解析用户输入时，如果格式错误，不是直接崩溃，而是弹窗让用户修改，然后恢复解析。以下用 Koka 语言的伪代码演示：

```
fun parseInput() : string
  val input = perform(GetInput)
  if not isValid(input) then
    perform(ShowError("Invalid input"))
    parseInput() // 递归调用以重试
  else
    input
```

```
handle
  with GetInput -> resume("user input") // 模拟获取输入
  with ShowError(msg) ->
    println(msg)
    resume(()) // 恢复执行
```

在这个代码中，perform(GetInput) 发起获取输入的效应，handler 通过 resume 提供输入并恢复执行。perform(ShowError) 显示错误消息，handler 打印消息后恢复，允许函数重试。这展示了代数效应如何将错误处理逻辑分离，使主逻辑更清晰。

另一个案例是异步操作即同步写法。编写一个完全同步风格的函数，内部发起了网络请求。以下用 JavaScript 伪代码演示：

```
function fetchData() {
  const data = perform('fetch', 'https://api.example.com/data');
  return process(data);
}

try {
  const result = fetchData();
  console.log(result);
} with (effect, arg) {
  if (effect === 'fetch') {
    fetch(arg).then(response => response.json()).then(data => resume(data));
  }
}
```

这里，perform('fetch') 发起网络请求效应，handler 使用 fetch API 异步获取数据，然后通过 resume 注入结果，恢复执行。这使得 fetchData 函数看起来是同步的，但实际上处理了异步操作，避免了回调嵌套。

依赖注入是另一个应用场景。函数需要访问配置或服务，但不希望硬编码。以下用 OCaml 伪代码演示：

```
let getConfig () = perform (GetConfig)

let handler =
  try
    getConfig ()
  with
    | GetConfig k -> resume k "production config"
```

在这个例子中，perform(GetConfig) 发起获取配置的效应，handler 提供配置值并通过 resume 恢复执行。这实现了灵活的依赖注入，允许在不同环境中切换配置，而无需修改业务逻辑。

代数效应的核心价值在于分离关注点，它提供了一种强大的控制流抽象能力，让代码更声明式、易于推理和维护。当前，虽然代数效应多为研究概念，但已开始在工业界产生巨大影响，如 React Hooks，证明了其价值。

未来，更多语言可能会在类型系统中集成效应类型（Effect Types），实现更安全的效应管理，它可能是解决复杂状态管理和异步并发问题的终极武器之一。鼓励读者用这种新的思维方式去思考程序的结构，即使当前使用的语言没有原生支持，也能从中获得架构上的启发。

6 附录

推荐资源包括 Koka 语言官网、React 团队 Sebastian Markbåge 的演讲和论文，以及 Oleg Kiselyov 等关于效应的经典文章。相关概念有续延传递风格（CPS）、函数式编程和效应类型（Effect Types）。这些资源可以帮助读者进一步深入理解代数效应及其应用。