18. 手写柯里化函数

1. 核心概念 (Core Concept)

柯里化（Currying）是函数式编程中的一个重要概念，指将一个接受多个参数的函数，转换成一系列只接受一个参数的函数序列，并返回一个新的函数。每次调用新的函数时，它会接收一个参数，并返回一个新的函数，直到接收到最后一个参数时，才会执行原始函数并返回结果。

2. 为什么需要它？ (The "Why")

柯里化带来了多种优势：

- **延迟执行和部分应用 (Deferred Execution & Partial Application):** 柯里化允许你提前固定函数的部分参数，生成一个新的函数，而无需立即执行。这使得你可以创建专注于处理特定参数的更简单的函数，或者根据需要在后续步骤中逐步提供参数。
- **提高函数的可复用性 (Enhanced Function Reusability):** 通过创建具有预设参数的新函数，你可以更容易地在不同的场景下复用同一个基础函数，避免重复编写相似的函数实现。
- **更好地组合函数 (Easier Function Composition):** 柯里化函数通常更容易与其他函数组合，形成更复杂的逻辑流，这在函数式编程范式中尤其有用。

3. API 与用法 (API & Usage)

柯里化本身不是 JavaScript 内置的 API，而是一种函数转换的技术。手动实现柯里化通常需要一个**高阶函数**，该函数接收一个多参数函数作为输入，并返回一个柯里化后的函数。

以下是一个经典的、简化版的**手动实现柯里化的高阶函数**示例：

```
/**
 * 手动实现柯里化函数
 * @param {Function} fn - 待柯里化的函数
 * @returns {Function} - 柯里化后的函数
 */
function curry(fn) {
  // 返回一个新的函数，这个函数是柯里化后的入口
  return function curried(...args) {
    // 如果当前收集的参数数量已经达到了原始函数所需的参数数量
    if (args.length >= fn.length) {
      // 直接调用原始函数，并应用当前收集到的参数
      return fn.apply(this, args);
    } else {
      // 如果参数数量不够，则返回一个新的函数
      // 新函数会接收剩余的参数，并与之前收集的参数合并后，再次调用 curried 函数自身
      return function(...restArgs) {
```

```
        return curried.apply(this, args.concat(restArgs));
      };
    }
  };
}

// 示例用法：一个简单的求和函数
function sum(a, b, c) {
  return a + b + c;
}

// 对 sum 函数进行柯里化
const curriedSum = curry(sum);

// 逐步调用柯里化后的函数
console.log(curriedSum(1)(2)(3)); // 输出: 6
console.log(curriedSum(1, 2)(3)); // 输出: 6
console.log(curriedSum(1)(2, 3)); // 输出: 6
console.log(curriedSum(1, 2, 3)); // 输出: 6
```

解释：

- `curry(fn)` 函数接收原始函数 `fn`。
- 它返回一个新的函数 `curried`，这个函数是柯里化后的入口。
- `curried` 函数内部，使用 `...args` 来收集当前调用接收到的参数。
- `fn.length` 可以获取原始函数形参的数量。
- 如果 `args.length` 已经等于或大于 `fn.length`，说明所有参数都已收集完毕，此时使用 `fn.apply(this, args)` 执行原始函数并返回结果。`apply` 用于确保 `this` 指向正确（尽管在纯函数场景下 `this` 不重要）。
- 如果参数数量不足，则返回一个新的函数。这个新函数会接收剩余的参数 `...restArgs`，并使用 `args.concat(restArgs)` 将新接收的参数与之前收集的参数合并。然后，递归地调用 `curried.apply(this, ...)` 自身，继续等待或处理更多参数。

4. 关键注意事项 (Key Considerations)

- **参数数量的判断:** 上述实现依赖于函数的 `length` 属性来确定原始函数期望的参数数量。请注意，参数默认值、剩余参数 (`...rest`) 和解构赋值参数不会计入 `fn.length`。对于包含这些特性的函数，标准的基于 `fn.length` 的柯里化实现可能需要调整。
- **`this` 的处理:** 示例中的 `apply(this, ...)` 尝试保留调用时的 `this` 上下文，但在纯粹的函数式编程场景中，通常不依赖 `this`。如果你的函数依赖特定的 `this`，柯里化实现需要确保正确地传递 `this`。
- **实现复杂性:** 完整的柯里化实现需要考虑更多细节，例如支持占位符参数（允许跳过某些参数并在后续提供），以及更健壮的参数收集和应用逻辑。上述示例是一个基础实现，用于理解核心原理。

- **与偏函数的区别:** 柯里化是将多参数函数转化为一系列单参数函数，直到收到最后一个参数才执行。偏函数（Partial Application）是固定函数的部分参数，返回一个新的函数，这个新函数接受剩余参数并执行原始函数。虽然柯里化可以实现偏函数的效果，但两者的概念和常见实现方式略有不同。

5. 参考资料 (References)

- **MDN Web Docs - 柯里化 (Currying):** (作为概念性参考，MDN 在函数式编程部分会提及柯里化) https://developer.mozilla.org/zh-CN/docs/Glossary/Currying
- **Effective JavaScript by David Herman:** (函数式编程章节通常会深入讨论柯里化和偏函数)
- **Lodash/Ramda 库文档:** (这些实用工具库提供了成熟的 `_.curry` 或 `R.curry` 实现，可以学习其用法和原理)
  - Lodash: https://lodash.com/docs/#curry
  - Ramda: https://ramdajs.com/docs/#curry
- **JavaScript 函数式编程相关技术博客:** (许多技术博客会深入探讨柯里化的实现细节和应用场景，需选择业界公认的优秀博客)