38.实现 Array.prototype.map、reduce

1. 核心概念 (Core Concept)

`Array.prototype.map()` 和 `Array.prototype.reduce()` 是 JavaScript 数组的两个非常重要且常用的高阶函数，它们都用于处理数组元素并返回新的结果，但处理方式和返回结果的形态不同。`map` 用于将数组的每个 ელემენტი 映射到 ꯑꯣ 값이에 对应，返回一个相同长度的新数组；`reduce` 用于将数组 ელემენტი 累积到一个单一值（或其他类型的值）。

2. 为什么需要它？ (The "Why")

1. **函数式编程风格 (Functional Programming Style):** 它们是函数式编程范式的体现，通过纯函数处理数据，避免副作用，使代码更可读、易于测试和维护。
2. **简洁高效 (Concise & Efficient):** 它们提供了一种声明式的方式来处理数组，相比传统的 `for` 或 `while` 循环，代码更简洁，意图更清晰。
3. **通用性 (Versatility):** `map` 和 `reduce` 可以应对各种数组转换和聚合的场景，是处理数组的强大工具。

3. API 与用法 (API & Usage)

本节将基于 ECMAScript 规范和 MDN 文档来描述这两个方法的实现原理和核心用法。

3.1 `Array.prototype.map()`

签名 (Simplified):

```
array.map(callback(currentValue, index, array), thisArg)
```

参数:

- `callback`: 回调函数，为数组中每个元素执行的函数。其返回值将构成新数组中对应位置的值。
  - `currentValue`: 数组中正在处理的当前元素。
  - `index` (可选): 数组中正在处理的当前元素的索引。
  - `array` (可选): `map` 方法被调用的数组。
- `thisArg` (可选): 执行 `callback` 函数时使用的 `this` 值。

返回值:

一个由原数组中的每个 ელემენტი 调用 `callback` 函数的返回值组成的新数组。

实现原理 (Simplified):

1. 创建一个新的空数组 `resultArray`。
2. 遍历原数组的 ელემენტი。
3. 对于每一个 ელემენტი，以 `currentValue`，`index`，`array` 为参数调用 `callback` 函数（若提供了 `thisArg`，则绑定 `this`）。
4. 将 `callback` 函数的返回值添加到 `resultArray` 中。
5. 遍历结束后，返回 `resultArray`。

代码示例 (模拟实现):

```
if (!Array.prototype.map) {
  Array.prototype.map = function(callback, thisArg) {
    'use strict';

    // 确保 this 是一个对象，并处理 null 或 undefined 的情况
    if (this == null) {
      throw new TypeError('Array.prototype.map called on null or
undefined');
    }

    // 将 this 转换为对象
    var O = Object(this);

    // 获取数组长度，确保 length 是一个无符号 32 位整数
    var len = O.length >>> 0;

    // Ensure callback is a function
    if (typeof callback !== 'function') {
      throw new TypeError(callback + ' is not a function');
    }

    // 创建结果数组
    var A = new Array(len);

    // 遍历数组
    var k = 0;
    while (k < len) {
      var kValue, mappedValue;

      // 检查属性是否存在（处理稀疏数组）
      if (k in O) {
        kValue = O[k];

        // 调用回调函数，并传入适当的 this 值
        // 使用 call 或 apply 来设置 this 和参数
        mappedValue = callback.call(thisArg, kValue, k, O);

        // 将结果赋值给新数组
        A[k] = mappedValue;
```

```
      }
      // else { empty slots in array-likes are dropped by
Array.prototype.map after ES5 }

      k++;
    }

    // 返回新数组
    return A;
  };
}

// 示例用法
const numbers = [1, 4, 9];
const roots = numbers.map(Math.sqrt);
console.log(roots); // [1, 2, 3]

const listItems = ['apple', 'banana'];
const htmlList = listItems.map(item => `<li>${item}</li>`);
console.log(htmlList); // ['<li>apple</li>', '<li>banana</li>']
```

3.2 `Array.prototype.reduce()`

签名 (Simplified):

```
array.reduce(callback(accumulator, currentValue, currentIndex, array),
initialValue)
```

参数:

- `callback`: 回调函数，对数组中的每个元素按顺序执行。
  - `accumulator`: 累积器，存储上一次调用回调函数的返回值。或者 `initialValue`（如果提供的话）。
  - `currentValue`: 数组中正在处理的当前元素。
  - `currentIndex` (可选): 数组中正在处理的当前元素的索引。如果 `initialValue` 存在，则从索引 0 开始；否则从索引 1 开始。
  - `array` (可选): `reduce` 方法被调用的数组。
- `initialValue` (可选): 作为第一次调用 `callback` 函数时的 `accumulator` 的初始值。如果未提供，则使用数组的第一个元素作为 `initialValue`，并从第二个元素开始执行 `callback`。

返回值:

累积计算的最终结果。

实现原理 (Simplified):

1. 判断是否提供 `initialValue`。
2. 如果提供了 `initialValue`，则 `accumulator` 初始化为 `initialValue`，遍历从索引 0 开始。
3. 如果没有提供 `initialValue`：
   - 检查数组是否为空。如果为空，抛出 `TypeError`。
   - `accumulator` 初始化为数组的第一个 ელემენტი，遍历从索引 1 开始。
4. 遍历数组的 ელემენტი (从适当的起始索引)。
5. 对于每个 ელემენტი，以当前的 `accumulator`，`currentValue`，`currentIndex`，`array` 为参数调用 `callback` 函数。
6. 将 `callback` 函数的返回值更新为新的 `accumulator`。
7. 遍历结束后，返回最终的 `accumulator` 值。

代码示例 (模拟实现):

```
if (!Array.prototype.reduce) {
  Object.defineProperty(Array.prototype, 'reduce', {
    value: function(callback /*, initialValue*/) {
      'use strict';

      if (this == null) {
        throw new TypeError('Array.prototype.reduce called on null or
undefined');
      }

      if (typeof callback !== 'function') {
        throw new TypeError(callback + ' is not a function');
      }

      var O = Object(this);
      var len = O.length >>> 0;
      var k = 0;
      var value;
      var isInitialValuePresent = arguments.length >= 2;

      if (isInitialValuePresent) {
        value = arguments[1];
      } else {
        // 如果没有提供初始值，使用数组的第一个元素
        while (k < len && !(k in O)) {
          k++;
        }
        // 如果数组是空的或者只有空槽，且没有提供初始值
        if (k >= len) {
          throw new TypeError('Reduce of empty array with no initial
value');
        }
        value = O[k++]; // 使用第一个存在的元素作为初始值，并从下一个元素开始遍历
```

```
      }

      while (k < len) {
        if (k in O) { // 只处理存在的属性
          // 调用回调函数更新累积值
          value = callback(value, O[k], k, O);
        }
        k++;
      }

      // 返回最终的累积值
      return value;
    }
  });
}


// 示例用法 1: 求和
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue) => accumulator +
currentValue, 0);
console.log(sum); // 10

// 示例用法 2: 展平数组
const nestedArray = [[1, 2], [3, 4], [5]];
const flatArray = nestedArray.reduce((accumulator, currentValue) =>
accumulator.concat(currentValue), []);
console.log(flatArray); // [1, 2, 3, 4, 5]

// 示例用法 3: 计算数组中每个元素出现的次数
const names = ['Alice', 'Bob', 'Tiff', 'Alice', 'Bob'];
const countedNames = names.reduce((allNames, name) => {
  const currentCount = allNames[name] ?? 0;
  return {
    ...allNames,
    [name]: currentCount + 1,
  };
}, {}); // 初始值是一个空对象
console.log(countedNames); // { Alice: 2, Bob: 2, Tiff: 1 }
```

4. 关键注意事项 (Key Considerations)

1. **`map` 的返回值是新数组:** `map` 方法总是返回一个**新的**数组，不会修改原数组。这是函数式编程“不可变性”原则的体现。
2. **`reduce` 的初始值:** 提供 `initialValue` 是推荐的最佳实践，特别是当数组可能为空时，可以避免因没有初始值而抛出 `TypeError`。同时，提供了初始值，`reduce` 的逻辑会更稳定和易于理解，它保证了回调函数被调用的次数与数组长度相同（对于非稀疏数组）。

3. **稀疏数组的处理:** 在 ES5+ 版本中，`map` 和 `reduce` (以及 `forEach`, `filter` 等) 在遍历时会跳过数组中的空槽（`empty` items，通过 `k in O` 判断是否存在属性）。模拟实现时需要注意这一点。
4. **修改原数组可能导致问题:** 在 `map` 或 `reduce` 的回调函数中修改正在遍历的原数组可能会导致意外行为，应尽量避免。

5. 参考资料 (References)

- **MDN Web Docs:**
  - `Array.prototype.map()`
  - `Array.prototype.reduce()`
- **ECMAScript® 2024 Language Specification:**
  - Array.prototype.map
  - Array.prototype.reduce
