41. 实现一个深比较 isEqual 函数

1. 核心概念 (Core Concept)

深比较（Deep Comparison）是指递归地比较两个对象的属性（包括嵌套的对象、数组等）是否完全相等，而不仅仅是比较它们的引用地址。实现一个 `isEqual` 函数的目的就是为了进行这种深度的相等性检查。

2. 为什么需要它？ (The "Why")

1. **比较复杂数据结构:** JavaScript 的 `==` 或 `===` 操作符对于非原始值（如对象和数组）只比较引用标识，而不是值本身。当需要判断两个包含复杂嵌套结构的对象或数组的内容是否相同时，必须使用深比较。
2. **状态管理与优化:** 在前端应用（如 React/Redux）中，经常需要比较新旧状态或 props 是否发生了实质性变化，以便决定是否进行渲染或更新。深比较可以确保只有内容真正改变时才触发相应的操作，提高性能。
3. **单元测试与断言:** 在编写测试用例时，常常需要断言两个复杂对象或数组的内容是否相等，深比较是实现此类断言的必备工具。

3. API 与用法 (API & Usage)

由于 `isEqual` 并非 JavaScript 内置的标准 API，其实现方式和具体签名可能因库而异（如 Lodash 的 `_.isEqual`）。但核心功能是接收两个任意类型的输入，返回一个布尔值表示它们是否深度相等。

典型函数签名:

```
function isEqual(value1: any, value2: any): boolean;
```

核心逻辑示例（简化的实现思路，不包含所有边界情况，仅为说明用途）：

一个简单的递归实现思路：
基础情况:
如果 `value1` 和 `value2` 全等 (`===`)，返回 `true`。(处理原始值、`null`、`undefined`、以及同一对象引用的情况)
如果它们的类型不同，返回 `false`。
如果其中一个是原始类型而另一个不是，返回 `false`。
对象/数组比较:
检查它们是否都是对象且非 `null`。
获取它们的键（包括不可枚举的属性，根据需求）。
如果键的数量不同，返回 `false`。
递归地比较每个键对应的值。如果在任何一个键上发现值不相等，立即返回 `false`。

如果所有键的值都相等，返回 `true` 。

特殊对象比较 (可选，取决于需求):

比较日期对象：检查它们是否都是 Date 实例，然后比较其时间戳。

比较正则表达式：检查是否都是 RegExp 实例，然后比较其 `source` 和 `flags` 。

* 比较函数：通常认为函数不同构成的对象不同（即使代码相同），或者简单返回 `false` 。

```
/**
 * 简化的 isEqual 实现示例（不处理循环引用等复杂情况）
 * 仅用于说明核心递归比较思路
 */
function isEqual(value1, value2) {
    // 1. 基础情况：原始值、null、undefined、同一引用
    if (value1 === value2) {
        return true;
    }

    // 2. 其中一个为基本类型而另一个不是，或都为 null/undefined 但不 === 的情况
(已被上个判断处理)
    //   在这里检查 typeof 是否都是 'object' 且 非 null
    if (typeof value1 !== 'object' || value1 === null || typeof value2 !==
'object' || value2 === null) {
        return false;
    }

    // 3. 调用构造函数是否一致（可选，更严谨的比较）
    // if (value1.constructor !== value2.constructor) {
    //     return false;
    // }

    // 4. 比较 Date 对象
    if (value1 instanceof Date && value2 instanceof Date) {
        return value1.getTime() === value2.getTime();
    }

    // 5. 比较 RegExp 对象
    if (value1 instanceof RegExp && value2 instanceof RegExp) {
        return value1.source === value2.source && value1.flags ===
value2.flags;
    }

    // 6. 比较 Map/Set（更完整的实现需要处理迭代器）
    // ...

    // 7. 比较普通对象或数组
    const keys1 = Object.keys(value1); // 简单示例：只比较可枚举自有属性
    const keys2 = Object.keys(value2);

    if (keys1.length !== keys2.length) {
```

```
            return false;
        }

        for (const key of keys1) {
            // 递归比较属性值
            // 注意：实际的 Lodash 等库会处理原型链、Symbol 属性等
            if (!Object.prototype.hasOwnProperty.call(value2, key) ||
!isEqual(value1[key], value2[key])) {
                return false;
            }
        }

        // 如果所有属性都深度相等
        return true;
}

// 示例用法
const objA = { a: 1, b: { c: 2 } };
const objB = { a: 1, b: { c: 2 } };
const objC = { a: 1, b: { c: 3 } };
const arrA = [1, [2, 3]];
const arrB = [1, [2, 3]];
const arrC = [1, [2, 4]];

console.log(isEqual(objA, objB)); // 输出: true
console.log(isEqual(objA, objC)); // 输出: false
console.log(isEqual(arrA, arrB)); // 输出: true
console.log(isEqual(arrA, arrC)); // 输出: false
console.log(isEqual(null, undefined)); // 输出: false
console.log(isEqual(1, '1'));     // 输出: false
console.log(isEqual(new Date('2023-01-01'), new Date('2023-01-01'))); //
输出: true
console.log(isEqual(/a/g, /a/g)); // 输出: true
```

4. 关键注意事项 (Key Considerations)

1. **循环引用 (Circular References):** 简单的递归实现会在遇到循环引用时导致无限循环和栈溢出。健壮的 `isEqual` 实现需要追踪已经比较过的对象对（例如，使用 Set 或 Map 来存储 `{obj1, obj2}` 对），如果在比较途中遇到已经追踪过的对，则认为它们相等。
2. **特殊对象类型:** 除了普通对象和数组，还需要考虑 `Date`、`RegExp`、`Map`、`Set`、`TypedArray`、`Error` 等内置对象的比较逻辑。它们需要特定的处理方法（如比较时间戳、Regex 标志、Map/Set 元素等）。
3. **属性的比较范围:** 是只比较可枚举的自有属性 (`Object.keys`)，还是包括 Symbol 属性 (`Object.getOwnPropertySymbols`)，或是包括原型链上的属性 (`for...in`)？标准的深比较通常只比较对象的可枚举自有属性，但具体取决于需求。
4. **性能:** 深比较尤其是对大型、深度嵌套的数据结构进行比较时，性能开销可能很大。在性能敏感的场景下，应权衡是否真的需要深比较，或者考虑使用不可变数据结构并进行引用

比较。

5. 参考资料 (References)

- **Lodash _.isEqual Documentation:** https://lodash.com/docs/#isEqual (提供了一个广泛使用的、健壮的实现参考)
- **MDN Web Docs - Equality comparisons and sameness:** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness (解释了 `==`, `===`, `Object.is` 等的比较区别，为理解深比较的必要性提供基础)
- **Stack Overflow / 教程关于实现 Deep Equal 的讨论:** 业界有大量关于如何实现健壮的深比较函数的讨论，可以搜索 "implement deep equal javascript" 找到相关资源，但注意甄别其是否处理了循环引用和各种特殊类型。