4.深浅拷贝的本质

1. 核心概念 (Core Concept)

浅拷贝 (Shallow Copy) 和 **深拷贝 (Deep Copy)** 都是针对**引用类型**（如对象和数组）的复制操作。

- **浅拷贝**: 创建一个新对象，然后将原始对象的**属性值**逐个复制到新对象。如果属性值是值类型，则复制值本身；如果属性值是引用类型，则只复制其**内存地址（引用）**。因此，新旧对象的引用类型属性仍然指向同一个底层对象。
- **深拷贝**: 创建一个新对象，并**递归地**复制原始对象的所有属性。不仅复制对象本身，还复制它所引用的所有对象。最终，新旧对象完全独立，互不影响。

2. 为什么需要它？ (The "Why")

1. **避免意外的副作用**: 在不希望修改原始数据的情况下操作其副本时，需要拷贝。如果使用浅拷贝处理包含嵌套对象的数据，修改副本的嵌套属性会意外地改变原始数据，导致难以追踪的 bug。
2. **实现状态的不可变性**: 在 React、Vue 等现代前端框架中，推崇状态的不可变性。当需要更新状态（特别是复杂的状态对象）时，通常会创建一个新的深拷贝副本进行修改，而不是直接修改原状态，这有助于框架进行高效的变更检测。
3. **隔离数据**: 在某些场景下，需要将一个对象的状态完全隔离出来，创建一个独立的快照，深拷贝是实现这一目标的唯一途径。

3. API 与用法 (API & Usage)

浅拷贝的常见实现

1. `Object.assign()`: 将所有可枚举属性的值从一个或多个源对象复制到目标对象。

```
const original = { a: 1, b: { c: 2 } };
const shallowCopy = Object.assign({}, original);

shallowCopy.a = 10;
shallowCopy.b.c = 20;

console.log(original.a);      // 1 (未受影响)
console.log(original.b.c);    // 20 (受到影响)
```

2. **扩展运算符 (`...`)**: ES6 提供的语法糖，效果与 `Object.assign()` 类似。

```
const original = { a: 1, b: { c: 2 } };
const shallowCopy = { ...original };
```

```
shallowCopy.a = 10;
shallowCopy.b.c = 20;

console.log(original.a);      // 1 (未受影响)
console.log(original.b.c);    // 20 (受到影响)
```

深拷贝的常见实现

1. **`JSON.parse(JSON.stringify(obj))`**：最简单但有局限性的深拷贝方法。
   - **优点**: 实现简单，能处理常见的 JSON 安全数据。
   - **缺点**:
     - 会忽略 `undefined`、`Symbol` 属性和函数。
     - 不能处理循环引用（会报错）。
     - 会把 `Date` 对象转换成字符串。

```
const original = { a: 1, b: { c: 2 }, d: new Date(), e: undefined };
const deepCopy = JSON.parse(JSON.stringify(original));

deepCopy.b.c = 20;

console.log(original.b.c); // 2 (未受影响)
console.log(typeof original.d); // "object"
console.log(typeof deepCopy.d);   // "string" (Date 被转换)
console.log('e' in deepCopy);   // false (undefined 被忽略)
```

2. **`structuredClone()` (现代推荐)**: 一个内置的、专门用于深拷贝的全局函数，支持更复杂的数据类型。
   - **优点**: 标准 API、高效、支持循环引用、能处理 `Date`, `RegExp`, `Map`, `Set` 等多种类型。
   - **缺点**: 不支持拷贝函数、`Error` 对象、DOM 节点。

```
const original = { a: 1, b: { c: 2 }, d: new Date() };
const deepCopy = structuredClone(original);

deepCopy.b.c = 20;
console.log(original.b.c); // 2 (未受影响)
console.log(original.d.getTime() === deepCopy.d.getTime()); // true
(Date 被正确克隆)
```

3. **手动递归实现** (面试常考)

```
function deepClone(obj, hash = new WeakMap()) {
    if (obj === null || typeof obj !== 'object') return obj;
```

```
    if (hash.has(obj)) return hash.get(obj); // 处理循环引用

    let cloneObj = Array.isArray(obj) ? [] : {};
    hash.set(obj, cloneObj); // 存储克隆的对象

    for (let key in obj) {
        if (Object.prototype.hasOwnProperty.call(obj, key)) {
            cloneObj[key] = deepClone(obj[key], hash);
        }
    }
    return cloneObj;
}
```

4. 关键注意事项 (Key Considerations)

1. **性能开销**: 深拷贝比浅拷贝消耗更多的计算资源和内存，因为它需要递归遍历整个对象树。对于层级很深或体积庞大的对象，应谨慎使用。
2. **选择合适的工具**:
   - 如果对象层级只有一层，浅拷贝 (`...` 或 `Object.assign`) 足够且高效。
   - 对于需要深拷贝的场景， **`structuredClone()` 是现代浏览器和 Node.js 环境下的首选**。
   - `JSON.parse(JSON.stringify())` 作为一个快速但不完美的备选方案，仅适用于数据结构简单且符合 JSON 格式的场景。
   - 对于需要处理函数等复杂情况的深拷贝，需要依赖 `lodash.cloneDeep` 等成熟的第三方库。
3. **循环引用**: `JSON.stringify` 无法处理循环引用的对象，会抛出 `TypeError` 。而 `structuredClone()` 和健壮的手动实现可以正确处理。
4. **特殊类型处理**: 注意 `Date` , `RegExp` , `Map` , `Set` , `Symbol` , `undefined` , `Function` 等特殊类型在不同深拷贝方法下的行为差异。

5. 参考资料 (References)

- MDN Web Docs: structuredClone()
- MDN Web Docs: Object.assign()
- MDN Web Docs: Spread syntax (...)
- Web.dev: Deep-copying in JavaScript using structuredClone