19. “面试常问函数题”合集（组合、记忆化）

1. 核心概念 (Core Concept)

函数组合 (Function Composition): 函数组合是一种将多个简单函数链式连接起来，创建一个新函数的通用技术，其中一个函数的输出作为下一个函数的输入。

函数记忆化 (Memoization): 函数记忆化是一种优化技术，用于通过存储昂贵函数调用的结果并返回缓存的结果（当输入相同）来加速程序执行，避免重复计算。

2. 为什么需要它？ (The "Why")

- **函数组合:**
  - **提高代码可读性和可维护性:** 将复杂的逻辑分解为多个可复用的简单函数，通过组合清晰地表达执行流程。
  - **增强函数的可复用性:** 独立的小函数更容易在不同场景下复用。
- **函数记忆化:**
  - **提升性能:** 显著减少计算密集型或重复性调用的时间，特别是在递归或处理大量数据时。
  - **减少资源消耗:** 避免因为重复计算而消耗额外的 CPU 资源。

3. API 与用法 (API & Usage)

函数组合和记忆化更多是**编程模式**或**实现技巧**，而非特定的内置 API。以下是它们的经典实现方式：

3.1. 函数组合 (Function Composition)

通常通过一个 `compose` 或 `pipe` 函数来实现。

- `compose`：从右向左执行函数。
- `pipe`：从左向右执行函数。

经典实现示例 (使用 `compose`):

```
/**
 * 从右向左执行函数组合。
 * compose(f, g, h) 等价于 (...args) => f(g(h(...args)))
 * @param {...Function} funcs - 要组合的函数列表。
 * @returns {Function} 一个新函数，它是输入函数的组合。
 */
const compose = (...funcs) => {
  if (funcs.length === 0) {
    return arg => arg;
```

```
  }

  if (funcs.length === 1) {
    return funcs[0];
  }

  return funcs.reduce((a, b) => (...args) => a(b(...args)));
};

// 示例用法：
const add1 = x => x + 1;
const multiply2 = x => x * 2;
const subtract3 = x => x - 3;

const processedValue = compose(subtract3, multiply2, add1);

console.log(processedValue(5)); // 执行流程：add1(5) -> 6, multiply2(6) ->
12, subtract3(12) -> 9
// 输出：9
```

3.2. 函数记忆化 (Memoization)

通常通过一个高阶函数或装饰器来实现。

基本实现示例:

```
/**
 * 为一个函数提供记忆化能力。
 * 适用于纯函数（相同的输入总是产生相同的输出）。
 * 默认使用第一个参数作为缓存key。
 * @param {Function} fn - 要进行记忆化的函数。
 * @returns {Function} 一个具有记忆化能力的新函数。
 */
const memoize = (fn) => {
  const cache = {}; // 使用一个对象作为简单的缓存

  return function(...args) {
    // 创建一个简单的缓存key，这里只考虑第一个参数或多个参数的JSON字符串
    // 对于复杂参数（如对象引用）需要更复杂的key生成策略
    const key = args.length === 1 ? args[0] : JSON.stringify(args);

    if (cache.hasOwnProperty(key)) {
      console.log(`Fetching from cache for key: ${key}`);
      return cache[key];
    } else {
      console.log(`Calculating result for key: ${key}`);
      const result = fn.apply(this, args);
      cache[key] = result;
      return result;
```

```
    }
  };
};

// 示例：一个耗时的斐波那契数列计算函数
const fibonacci = (n) => {
  if (n <= 1) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
};

// 对斐波那契函数进行记忆化
const memoizedFibonacci = memoize(fibonacci);

console.log(memoizedFibonacci(10)); // 会进行计算并缓存中间结果
console.log(memoizedFibonacci(10)); // 直接从缓存读取，速度更快
console.log(memoizedFibonacci(15)); // 会进行计算并缓存中间结果
console.log(memoizedFibonacci(10)); // 再次为10，从缓存读取
```

- **注意:** 上述记忆化实现是一个基础版本。对于多参数、参数顺序敏感、参数类型复杂（如对象引用、函数）的场景，需要更健壮的缓存 key 生成策略（例如，使用 Map 或自定义哈希函数）。Redux Toolkit 的 `createSelector` 和 Lodash 的 `_.memoize` 提供了更强大的记忆化实现。

4. 关键注意事项 (Key Considerations)

- **函数组合:**
  - **函数签名匹配:** 组合的核心在于前一个函数的输出类型必须与后一个函数的输入类型兼容。
  - **错误处理:** 在组合函数链中，如何优雅地处理其中某个函数抛出的错误需要仔细考虑。
- **函数记忆化:**
  - **缓存策略:** 需要考虑使用哪种数据结构作为缓存 (对象、Map)，以及何时/如何清理缓存以避免内存泄漏（例如 LRU 策略）。
  - **缓存 key 的生成:** 这是实现记忆化最关键的部分。必须确保相同输入始终生成相同的 key，且不同的输入生成不同的 key。对于非原始类型的参数，简单的 JSON.stringify 可能不足够或效率低下。
  - **只对纯函数使用记忆化:** 记忆化假设一个函数对于相同的输入总是产生相同的输出（无副作用）。对于有副作用的函数，记忆化可能导致意外行为。

5. 参考资料 (References)

- MDN Web Docs: Function composition
- MDN Web Docs: Memoization

- Lodash Documentation: _.compose (虽然官方推荐 flow, 但 compose 概念相同)
- Lodash Documentation: _.memoize
- Redux Toolkit Documentation: createSelector (内部使用了 reselect，一个强大的记忆化库)