# 3.值类型 vs 引用类型

## 1. 核心概念 (Core Concept)

JavaScript 的数据类型根据其在内存中的存储方式，分为**值类型 (Value Types)** 和**引用类型 (Reference Types)**。

- **值类型 (原始类型)**: 变量直接存储数据的值。当将一个值类型的变量赋给另一个变量时，计算机会为新变量分配一个全新的内存空间，然后将原始值复制一份过去。
- **引用类型 (对象类型)**: 变量存储的是一个指向数据在内存中存放位置的**指针（或引用）**。当将一个引用类型的变量赋给另一个变量时，只是将这个指针复制了一份，两个变量最终指向堆内存中的同一个对象。

## 2. 为什么需要理解它们?

1. **预测变量行为**: 理解值类型和引用类型的差异是预测变量在赋值、传参等操作后行为的基础。错误地认为引用类型是按值复制，会导致难以调试的 bug。
2. **理解函数传参机制**: JavaScript 函数的参数传递**始终是"按值传递" (pass by value)**。对于值类型，是传递值的副本；对于引用类型，是传递引用的副本。这解释了为什么在函数内部修改引用类型参数的属性会影响到外部对象。
3. **掌握深浅拷贝的基础**: 深拷贝和浅拷贝的概念完全建立在值类型和引用类型的基础之上。只有理解了引用，才能明白为什么需要深拷贝。

## 3. API 与用法

### 值类型 (Value Types)

包括 `String`, `Number`, `Boolean`, `Null`, `Undefined`, `Symbol`, `BigInt`。

```
let a = 100;
let b = a; // b 是 a 的一个副本

a = 200; // 修改 a 的值

console.log(a); // 输出: 200
console.log(b); // 输出: 100 (b 的值不受影响)
```

**内存图解**:

- `a` 在栈内存中占据一个位置，存储值 `100`。
- `b = a` 时，在栈中为 `b` 开辟一个新位置，将 `a` 的值 `100` 复制过来。
- `a = 200` 时，只修改了 `a` 所在内存位置的值，与 `b` 无关。

# 引用类型 (Reference Types)

包括 `Object`, `Array`, `Function` 等。

```
let obj1 = { name: 'Alice' };
let obj2 = obj1; // obj2 复制的是指向 { name: 'Alice' } 的引用

obj1.name = 'Bob'; // 通过引用修改堆内存中的对象

console.log(obj1.name); // 输出: 'Bob'
console.log(obj2.name); // 输出: 'Bob' (obj2 也指向同一个对象，因此也受到影响)
```

**内存图解**:

- 对象 `{ name: 'Alice' }` 被创建在**堆内存**中。
- `obj1` 在**栈内存**中存储一个地址，该地址指向堆中的对象。
- `obj2 = obj1` 时，在栈中为 `obj2` 开辟新位置，将 `obj1` 中存储的地址复制过来。
- `obj1.name = 'Bob'` 时，通过 `obj1` 的地址找到堆中的对象，并修改其 `name` 属性。由于 `obj2` 也持有相同的地址，所以访问 `obj2.name` 会看到修改后的结果。

# 4. 关键注意事项 (Key Considerations)

1. **函数参数传递**:
   - 传递值类型：函数内部对参数的修改不会影响外部变量。
   - 传递引用类型：函数内部修改参数对象的**属性**会影响外部对象。但如果函数内部将参数**重新赋值**为一个新对象，则会切断与外部对象的联系。

   ```
   function modify(obj) {
     // 修改属性，会影响外部
     obj.name = 'Modified';

     // 重新赋值，不会影响外部
     obj = { name: 'New Object' };
   }

   let myObj = { name: 'Initial' };
   modify(myObj);
   console.log(myObj.name); // 输出: 'Modified'
   ```

2. **`const` 和引用类型**: `const` 声明一个引用类型的变量时，仅保证变量存储的**引用（地址）不可变，但该引用指向的对象内容**是可变的。

```
const user = { name: 'John' };
user.name = 'Jane'; // 合法
// user = { name: 'Doe' }; // 非法, TypeError
```

3. **判断相等 ( `==` 或 `===` )**:
   - 对于值类型，比较的是它们的值。
   - 对于引用类型，比较的是它们是否指向同一个内存地址（引用）。两个内容完全相同的独立对象，比较结果也是 `false`。

```
console.log({} === {}); // false
```

# 5. 参考资料 (References)

- MDN Web Docs: JavaScript data structures - Primitives
- JavaScript Visualized: Hoisting, Scope, and Closures (related concepts) (Visualizations help understand stack/heap)