35. JS 中常见继承方式对比总结

1. 核心概念 (Core Concept)

JavaScript 中的继承是一种机制，允许一个对象（或构造函数）“继承”另一个对象（或构造函数）的属性和方法。这意味着子对象可以访问父对象定义的属性和方法，从而实现代码的复用和构建对象之间的层级关系。

2. 为什么需要它？ (The "Why")

- **代码复用 (Code Reusability):** 避免在多个相似对象中重复定义相同的属性和方法。
- **构建层级结构 (Building Hierarchies):** 组织和管理复杂代码，建立对象之间的父子关系。
- **多态 (Polymorphism):** 允许不同的对象对同一个方法调用做出不同的响应（虽然 JS 的原型继承实现多态的方式与传统的类继承略有不同）。

3. API 与用法 (API & Usage)

JavaScript 提供了多种实现继承的方式，主要包括：

- **原型链继承 (Prototype Chain Inheritance):** 通过将子类型的原型指向父类型的实例来实现。
- **构造函数继承 (Constructor Function Inheritance):** 在子类型的构造函数内部调用父类型的构造函数。
- **组合继承 (Combination Inheritance):** 结合原型链继承和构造函数继承。
- **原型式继承 (Prototypal Inheritance):** 使用 `Object.create()` 方法。
- **寄生式继承 (Parasitic Inheritance):** 在原型式继承的基础上增强对象。
- **寄生组合式继承 (Parasitic Combination Inheritance):** 结合寄生式继承和构造函数继承，是公认的最优方案。
- **ES6 Class 的 `extends` / `super` 关键字:** 语法糖，底层仍然是原型链和构造函数的组合应用。

以下是几种常见方式的**纯粹示例**，展示核心机制：

原型链继承 (Prototype Chain Inheritance):

```
function Parent() {
  this.name = 'parent';
  this.colors = ['red', 'blue'];
}

Parent.prototype.sayName = function() {
  console.log(this.name);
};
```

```
function Child() {
  this.age = 10;
}

// 核心：将子类型的原型指向父类型的实例
Child.prototype = new Parent(); // 注意：这里会创造一个不必要的 Parent 实例
Child.prototype.constructor = Child; // 修复 constructor 指向

Child.prototype.sayAge = function() {
  console.log(this.age);
};

const instance1 = new Child();
instance1.sayName(); // 输出 'parent'
console.log(instance1.colors); // 输出 ['red', 'blue']
instance1.colors.push('green');
console.log(instance1.colors); // 输出 ['red', 'blue', 'green']

const instance2 = new Child();
console.log(instance2.colors); // 输出 ['red', 'blue', 'green'] - 说明引用类型的属性被所有实例共享
```

构造函数继承 (Constructor Function Inheritance):

```
function Parent() {
  this.name = 'parent';
  this.colors = ['red', 'blue'];
}

function Child() {
  // 核心：在子类型构造函数中调用父类型构造函数
  Parent.call(this); // or Parent.apply(this)
  this.age = 10;
}

const instance1 = new Child();
instance1.colors.push('green');
console.log(instance1.colors); // 输出 ['red', 'blue', 'green']

const instance2 = new Child();
console.log(instance2.colors); // 输出 ['red', 'blue'] - 引用类型属性不再共享
console.log(instance1.name); // 输出 'parent'
console.log(instance2.name); // 输出 'parent'

// 注意：Parent.prototype 上的方法不会被继承
// instance1.sayName(); // 报错：instance1.sayName is not a function
```

组合继承 (Combination Inheritance):

```
function Parent() {
  this.name = 'parent';
  this.colors = ['red', 'blue'];
}

Parent.prototype.sayName = function() {
  console.log(this.name);
};

function Child() {
  // 继承属性
  Parent.call(this);
  this.age = 10;
}

// 继承方法
Child.prototype = new Parent(); // 问题：调用了两次 Parent 构造函数
Child.prototype.constructor = Child;

Child.prototype.sayAge = function() {
    console.log(this.age);
};

const instance1 = new Child();
instance1.colors.push('green');
console.log(instance1.colors); // 输出 ['red', 'blue', 'green']
instance1.sayName(); // 输出 'parent'

const instance2 = new Child();
console.log(instance2.colors); // 输出 ['red', 'blue']
instance2.sayName(); // 输出 'parent'
```

寄生组合式继承 (Parasitic Combination Inheritance): (公认最优方案)

```
function inheritPrototype(child, parent) {
    // 创建一个空对象作为中介，避免直接实例化父类
    const prototype = Object.create(parent.prototype);
    prototype.constructor = child; // 增强对象，指定 constructor
    child.prototype = prototype; // 将子类型的原型指向这个中介对象
}

function Parent() {
  this.name = 'parent';
  this.colors = ['red', 'blue'];
}
```

```
Parent.prototype.sayName = function() {
  console.log(this.name);
};

function Child() {
  // 继承属性
  Parent.call(this);
  this.age = 10;
}

// 继承方法
inheritPrototype(Child, Parent);

Child.prototype.sayAge = function() {
    console.log(this.age);
};

const instance1 = new Child();
instance1.colors.push('green');
console.log(instance1.colors); // 输出 ['red', 'blue', 'green']
instance1.sayName(); // 输出 'parent'

const instance2 = new Child();
console.log(instance2.colors); // 输出 ['red', 'blue']
instance2.sayName(); // 输出 'parent'

console.log(instance1 instanceof Child); // true
console.log(instance1 instanceof Parent); // true
```

ES6 Class (语法糖):

```
class Parent {
  constructor() {
    this.name = 'parent';
    this.colors = ['red', 'blue'];
  }

  sayName() {
    console.log(this.name);
  }
}

class Child extends Parent {
  constructor() {
    // 核心：调用父类构造函数
    super();
    this.age = 10;
  }
```

```
  sayAge() {
    console.log(this.age);
  }
}

const instance1 = new Child();
instance1.colors.push('green');
console.log(instance1.colors); // 输出 ['red', 'blue', 'green']
instance1.sayName(); // 输出 'parent'

const instance2 = new Child();
console.log(instance2.colors); // 输出 ['red', 'blue']
instance2.sayName(); // 输出 'parent'

console.log(instance1 instanceof Child); // true
console.log(instance1 instanceof Parent); // true
```

4. 关键注意事项 (Key Considerations)

- **原型链继承的问题:** 引用类型的属性会被所有实例共享；创建子类型实例时无法向父类型构造函数传参；修改子类型原型会影响所有实例。
- **构造函数继承的问题:** 方法必须在构造函数中定义，不能复用；父类型原型上定义的方法对子类型不可见。
- **组合继承的问题:** 调用了两次父类型构造函数，造成不必要的开销。
- **寄生组合式继承的优点:** 解决了组合继承调用两次父构造函数的问题，同时也保留了继承属性和方法的优点，是较为完美的解决方案（在 ES6 之前）。
- **ES6 Class 的本质:** `class` 语法是 `extends` 属性的语法糖，底层仍然基于原型链和构造函数实现，使用了类似于寄生组合继承的机制。它解决了传统原型继承的许多繁琐之处，是现代 JavaScript 中实现继承的首选方式。

5. 参考资料 (References)

- MDN Web Docs - Inheritance and the prototype chain
- MDN Web Docs - Classes
- JavaScript 高级程序设计 (第4版) (Chapter 6 - Object-Oriented Programming)