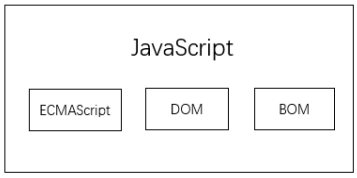
读书总结

<<JavaScript 高级程序设计 第 4 版>>

1. JavaScript 与 ECMAScript

- ECMAScript 是 JavaScript 的核心，各个浏览器均以 ECMAScript 作为自己 JavaScript 实现的依据，具体实现各有不同。完整的 JavaScript 包含三个部分：（1）核心（ECMAScript）；（2）文档对象模型（DOM）；（3）浏览器对象模型（BOM）。如下图所示。



2. ECMAScript 版本

- ES6/ES2015：新增类、模块、迭代器、生成器、箭头函数、期约、反射、代理和众多新的数据类型；
- ES7：新增 Array.prototype.includes 和指数操作符；
- ES8：新增异步函数（async/await）、SharedArrayBuffer、Atomics API、Object.values()/Object.entries()/Object.getOwnPropertyDescriptors()和字符串填充方法；
- ES9：新增异步迭代、Promise finally()；
- ES10：
- ES11：

3. DOM

- DOM（文档对象模型）：提供与网页内容交互的方法和接口

4. BOM

- BOM（浏览器对象模型）：提供与浏览器交互的方法和接口

5. script 标签

- 将 JavaScript 插入 HTML 的主要方法是使用 script 标签，script 标签共有 8 个属性：async、charset、crossorigin、defer、integrity、language、src 和 type。
- 如果 type 属性值为 module，则代码会被当成 ES6 模块，只有这时候代码中才能出现 import 和 export 关键字；
- script 标签的使用方式有两种：（1）直接在网页中嵌入 JavaScript 代码；（2）在网页中包含外部 JavaScript 文件（更推荐使用外部文件的方式），例如：

```
<script src="example.js"></script>
```

6. ECMAScript 语法

- ECMAScript 中一切都区分大小写，例如，变量 test 和变量 Test 是两个不同的变量；

- 标识符，即变量、函数、属性或函数参数的名称，根据惯例，ECMAScript 标识符使用驼峰大小写的形式，即第一个单词的首字母小写，后面每个单词的首字母大写；
- 严格模式："use strict"。

7. ECMAScript 变量

- ECMAScript 变量是松散类型的，即变量可以保存任何类型的数据，声明变量可以使用 var、let 和 const（let 和 const 为 ES6 新增 ）这 3 个关键字，不同数据类型的变量初始化可以在同一条语句中声明；
- let 声明的是块作用域，var 声明的是函数作用域，例如：

```
if (true) {
  var name = "Tom"; // 函数作用域
  console.log(name); // Tom
}
console.log(name); // Tom
```

```
if (true) {
  let name = "Tom"; // 块作用域
  console.log(name); // Tom
}
console.log(name); // ReferenceError: name未定义
```

- let 在全局作用域中声明的变量不会成为 window 的属性，而 var 声明的变量会；
- 只使用 let 和 const 有助于提高代码质量，const 优先，let 次之。

8. ECMAScript 数据类型

- 6 种简单数据类型/原始类型：Undefined、Null、Boolean、Number、String 和 Symbol（ES6 新增）；
- 1 种复杂数据类型/引用类型：Object/对象。
- typeof 操作符：注意 typeof null 返回"object"；
- Undefined 类型只有一个值即 undefined， 值为 undefined 的变量是 指该变量声明了但未初始化；
- Null 类型也只有一个值即 null，null 值表示一个空对象指针，这就是为什么 typeof null 返回的是"object"；
- Boolean 类型有两个字面值：true 和 false。这两个布尔值不同于数值，即 true 不等于 1，false 不等于 0，但是其他类型的值都能与布尔值进行转换，在 if 等流控制语句中其他类型值会自动转换为布尔值；
- Number 类型
- String 类型
- Symbol 类型：用来确保对象属性使用唯一标识符，不会发生属性冲突。使用 Symbol()函数进行初始化，Symbol()函数不能与 new 关键字一起作为构造函数使用，例如：

```
let n = new Number();
console.log(typeof n); // "object"

let s = new Symbol(); // TypeError: Symbol is not a constructor
```

- Object 类型，Object 是所有对象的基类。

9. ECMAScript 语句

- for-in 语句:一种严格的迭代语句，用于枚举对象中的非符号键属性。由于 ECMAScript 中对象的属性是无序的，因此 for-in 语句不能保证返回对象属性的顺序。
- for-of 语句：一种严格的迭代语句，用于遍历可迭代对象的元素。for-of 循环会按照可迭代对象的 next() 方法产生值的顺序迭代元素。ES2018 对 for-of 语句进行了扩展，增加了 for-await-of 循环，以支持生成期约（promise）的异步可迭代对象。示例：

```
// 这里控制语句中的const不是必需的，但为了确保这个局部变量不被修改，推荐使用const
for (const el of [2, 4, 6, 8]) {
  console.log(el);
}
```

10. 原始值与引用值

- 原始值保存在栈内存上，引用值保存在堆内存上；
- 保存原始值的变量是按值访问的，保存引用值的变量是按引用访问的；
- 原始值不能有属性，只有引用值可以动态添加后面可以使用的属性；
- 原始类型的初始化只能使用原始字面量形式，如果使用 new 关键字，会创建一个 Object 类型的实例；
- 通过变量把原始值赋值给另一个变量时，两个变量是完全独立的；

```
let num1 = 5;
let num2 = num1;
```

变量复制前

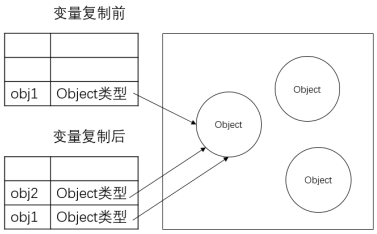
num1	5（Number类型）

变量复制后

num2	5（Number类型）
num1	5（Number类型）

- 通过变量把引用值赋值给另一个变量时，存储在变量中的值也会被复制到新变量所在的位置，这里复制的实际上是一个指针，它指向存储在堆内存中的对象；

```
let obj1 = new Object();
let obj2 = obj1;
obj1.name = "Tom";
console.log(obj2.name); // "Tom"
```



- ECMAScript 所有函数的参数都是按值传递的，ECMAScript 中函数的参数就是局部变量。

11. 变量作用域

- 搜索标识符表示什么过程如下：搜索开始于作用域前端，如果在局部上下文找到该标识符，则搜索停止，变量确定；如果没有找到变量名，则继续沿作用域链搜索。如果搜索到全局上下文，仍然没有找到该标识符，则说明该变量未声明。例如：

```
var color = "blue";
function getColor() {
  let color = "red";
  {
    let color = "green";
    return color;
  }
}

console.log(getColor()); // "green"
```

12. 垃圾回收与内存管理

- JavaScript 是使用垃圾回收的语言，垃圾收集器会定时、周期性地找出不再继续使用的变量，然后释放其占用的内存。标识无用变量的策略通常有两种：
  （1）标记清除（最常用）：垃圾收集器会给内存中所有变量都加上标识，然后去掉环境中的变量和被环境中变量引用的变量的标识，之后仍然被加上标识的变量就会被垃圾收集器回收所占用的内存。
  （2）引用计数：跟踪每个值被引用的次数，当这个值的引用次数变为 0，其占用的内存空间就会被回收。

- 将内存占用量保持在一个较小的值可以让页面性能更好，优化内存占用的最佳手段就是保证在执行代码时只保存必要的数据。如果数据不再必要，就解除引用，比如把它设置为 null；

13. 原始值包装类型

- ECMAScript 提供了 3 种特殊的引用类型：Boolean、Number 和 String。

```
let s1 = "hello world";
let s2 = s1.substring(2); // "llo world"
```

- 在上面的示例中，s1 是一个原始值，从逻辑上讲不应该有方法，实际上后台进行了特殊的处理：
  （1）创建一个 String 包装类型的示例；

（2）调用实例上的 substring 方法；
（3）销毁实例。

- 对象的生命周期：通过 new 实例化引用类型后，得到的实例会在离开作用域时被销毁，而自动创建的原始值包装对象只存在于访问它的那行代码执行期间，这就意味着不能给原始值添加属性和方法。
- 使用 new 调用原始值包装类型的构造函数，和调用同名的转型函数并不一样。

```
let value = "25";
let number = Number(value); // 转型函数
console.log(typeof number); // "number"
let obj = new Number(value); // 构造函数，但不推荐显式创建原始值包装类型的示例
console.log(typeof obj); // "object"
```

14. 内置对象

- 除了 Object、Array、Function 和原始值包装对象，ECMAScript 还定义了另外 2 种内置对象：Global 和 Math；
- Global：
- Math：

15. 引用类型

- Object
- Array：数组中每个槽位可以存储任意类型的数据，即可以创建一个数组，它的第一个元素是字符串，第二个元素是数值，第三个元素是对象。ECMAScript 数组是动态大小的，会随着数据添加而自动增长；
- 定型数组
- Map（ES6 新增）：键/值对，与 Object 只能使用数值、字符串和符号作为键不同，Map 可以使用任何 JavaScript 数据类型作为键；
- WeakMap（ES6 新增）：弱映射，WeakMap 中的 “weak” 描述的是 JavaScript 垃圾回收程序对待弱映射中键的方式。弱映射中的键只能是 Object 或继承自 Object 的类型，使用非对象设置键会抛出 TypeError，值类型没有限制；WeakMap 中的键不属于正式的引用，不会阻止垃圾回收；WeakMap 无法对键/值对进行迭代。
- Set（ES6 新增）：一种新的集合类型，Set 可以包含任何 JavaScript 数据类型作为值；
- WeakSet（ES6 新增）：弱集合，WeakSet 中的 “weak” 描述的是 JavaScript 垃圾回收程序对待弱集合中值的方式。弱集合中的值只能是 Object 或继承自 Object 的类型，使用非对象设置键会抛出 TypeError。WeakSet 中的值不属于正式的引用，不会阻止垃圾回收。WeakSet 无法对值进行迭代。
- Function

16. 迭代器

- ES5 新增 Array.prototype.forEach()进行迭代，但 forEach()方法无法标识迭代何时终止；

```
let arr = ["foo", "bar", "app"];
arr.forEach(item => {
  console.log(item);
});
```

- ES6 新增迭代器模式，字符串、数组、Map、Set、arguments 对象和 NodeList 等 DOM 集合类型都实现了 Iterable 接口（可迭代协议）。任何实现 Iterable 接口的对象都有 Symbol.iterator 属性。如果对象原型链上父类实现了 Iterable 接口，那么这个对象也实现了这个接口。通过调用迭代器工厂函数会生成一个迭代器，例如：

```
let str = "abc";
console.log(str[Symbol.iterator]()); // StringIterator {}
```

- 迭代器使用 next()方法在可迭代对象中遍历数据，成功返回的迭代器对象 IteratorResult 包含两个属性：done 和 value。done 是布尔值，表示是否还可以再次调用 next()取下一个值，取 true 是表示“耗尽”，value 包含迭代对象的下一个值。

17. 生成器

- ES6 新增的结构，可以在一个函数块内暂停和恢复代码的执行；
- 生成器的形式是一个函数，函数名称前加一个星号（*）表示它是一个生成器（标识生成器函数的星号不受两侧空格的影响），箭头函数不能用来定义生成器函数，示例：

```
// 生成器函数声明
function* generatorFn() {}

// 生成器函数表达式
let generatorFn = function*() {};

// 作为对象字面量方法的生成器函数
let foo = {
  *generatorFn() {}
};

// 作为类实例方法的生成器函数
class Foo {
  *generatorFn() {}
}

// 作为类静态方法的生成器函数
class Foo {
  static *generatorFn() {}
}
```

- 调用生成器函数会产生一个生成器对象，生成器对象开始处于暂停执行（suspended）状态，生成器对象也实现了 Iterator 接口，具有 next()方法，调用 next()方法会让生成器开始或恢复执行；
- yield 关键字可以让生成器停止和开始执行，生成器函数在遇到 yield 之前会正常执行，遇到 yield 之后执行会停止，函数作用域的状态会被保留，只能通过调用 next()方法来恢复执行；
- yield 关键字只能在生成器函数内部使用，出现在嵌套的非生成器函数中会抛出语法错误；

18. 对象

- ECMAScript 将对象定义为一组属性的无序集合，对象的每个属性和方法都由一个名称标识，这个名称映射到一个值，可以把 ECMAScript 中的对象想象为一张散列表，其中内容就是名/值对，值可以是数据或函数；
- 对象的属性分为两种：数据属性和访问器属性。
- 数据属性有 4 个特性描述它们的行为：[[Configurable]]、[[Enumerable]]、[[Writable]]和[[Value]]，要修改属性的默认特征，必须使用 Object.defineProperty()方法，示例：

```
let person = {};
Object.defineProperty(person, "name", {
  writable: false, // 只读
  value: "Nicholas"
});
console.log(person.name); // "Nicholas"
person.name = "Greg";
console.log(person.name); // "Nicholas"
```

- 虽然对于同一属性可以多次调用 Object.defineProperty()，但在 configurable 值设为 false 后再次调用就会报错。同时在调用 Object.defineProperty()时，configurable、enumerable 和 writable 的值如果不指定，就默认为 false；
- 访问器属性不包含数据值，有 4 个特性描述它们的行为：[[Configurable]]、[[Enumerable]]、[[Get]]和[[Set]]，访问器属性是不能直接定义的，必须使用 Object.defineProperty()；
- 可以通过 Object.defineProperties()方法一次性定义多个属性；
- 可以使用 Object.getOwnPropertyDescriptor()和 Object.getOwnPropertyDescriptors()（ES7 新增）方法读取属性的特性；
- ES6 新增 Object.assign()方法用来合并对象，该方法接受一个目标对象和一个或多个源对象作为参数，然后将每个源对象中的可枚举（Object.propertyIsEnumerable()返回 true）和只有（Object.hasOwnProperty()返回 true）属性复制到目标对象。如果多个源对象都有相同的属性，则使用最后一个复制的值。示例：

```
let dest = {},
  a = { id: "a" },
  b = { id: "b" };
// 浅拷贝
let res = Object.assign(dest, a, b);
console.log(res === dest); // true
```

- 对象相等判定：ES6 之前使用 === 操作符进行判断，但有些特殊情况 === 无能为力，因此，ES6 新增 Object.is()方法来判定对象是否相等。
- 增强的对象语法

- 对象解构：涉及多个属性的解构赋值是一个输出无关的顺序化操作，如果一个解构表达式涉及多个赋值，开始的赋值成功而后面的赋值出错，则整个解构赋值只会完成一部分。

19. 创建对象

- Object 构造函数；
- 对象字面量；
- 工厂模式：按照特定接口创建对象的方式，虽然可以解决创建多个类似对象的问题，但没有解决对象标识问题（即新创建的对象是什么类型）。例如：

```
function createPerson(name, age, job) {
  let o = new Object();
  o.name = name;
  o.age = age;
  o.job = job;
  o.sayName = function() {
    console.log(this.name);
  };
  return o;
}

let p1 = createPerson("Nicholas", 20, "Doctor");
```

- 构造函数模式：用于创建特定类型对象。例如：

```
function Person(name, age, job) {
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = function() {
    console.log(this.name);
  };
}

let p1 = new Person("Nicholas", 20, "Doctor");
```

Person() 构造函数与 createPerson() 工厂函数有如下区别：
（1）没有显示创建对象；
（2）属性和方法直接赋值给 this；
（3）函数没有 return。
同时构造函数名称的首字母都是大写的，非构造函数则以小写字母开头。

使用 new 操作符调用构造函数会执行以下操作：
（1）在内存中创建一个新对象；
（2）新对象的[[Prototype]]特性被赋值为构造函数的 prototype 属性；
（3）构造函数内部的 this 被赋值为这个新对象（即 this 指向新对象）；

（4）执行构造函数内部的代码（给新对象添加属性）；
（5）如果构造函数返回非空对象，则返回该对象，否则返回刚创建的新对象。

构造函数和普通函数唯一的区别就是调用方式不同，任何函数只要使用 new 操作符调用就是构造函数，而不使用 new 操作符调用的函数就是普通函数。前面定义的 Person()也可以像下面这样调用：

```
// 作为函数调用
Person("Greg", 25, "Engineer");
window.sayName(); // "Greg"

// 在另一个对象的作用域中调用
let obj = new Object();
Person.call(obj, "Jerry", 30, "Nurse");
o.sayName(); // "Jerry"
```

在上面的例子中，没有使用 new 操作符调用 Person()，会将属性和方法添加到 window 对象。注意，在调用一个函数而没有明确设置 this 的情况（即没有作为对象的方法调用，也没有使用 call()/apply()调用），this 始终指向 Global 对象（在浏览器中就是 window 对象）。

构造函数有一个问题就是：其中定义的方法会在每个实例上都创建一遍，而如果将方法定义在构造函数外部，那个函数实际上只能在一个对象上调用。可以通过原型模式来解决。

- 原型模式：每个函数都会创建一个 prototype 属性，这个属性是一个对象，这个对象就是构造函数的原型，在它上面定义的方法和属性可以被对象实例共享。示例：

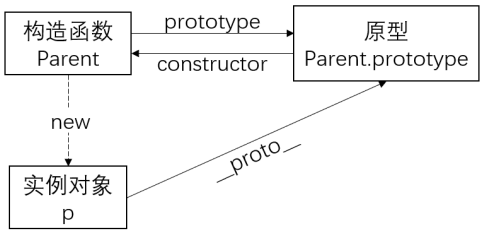
```
function Person() {}

Person.prototype.name = "Nicholas";
Person.prototype.age = 20;
Person.prototype.jog = "Doctor";
Person.sayName = function() {
  console.log(this.name);
};

let p1 = new Person(),
  p2 = new Person();
console.log(p1.sayName == p2.sayName); // true
```

20. 原型

- 原型：实例与构造函数原型之间有直接的联系，但实例与构造函数之间没有。



- 原型层级：在通过对象访问属性时，会按照这个属性的名称进行搜索。搜索开始于对象实例本身。如果在这个实例上发现了给定的名称，则返回该名称对应的值。如果没有找到这个属性，就会沿着指针进入原型对象，然后在原型对象上找到属性后，再返回相应的值。
  hasOwnProperty()方法可以用于确定某个属性是在实例上还是原型对象上。

- 其他原型语法：直接通过一个包含所有属性和方法的对象字面量来重写原型。示例：

```
function Person() {}

Person.prototype = {
  name: "Nicholas",
  age: 20,
  job: "doctor",
  sayName() {
    console.log(this.name);
  }
};
```

这样重写之后，只有一个问题就是：Person.prototype 的 constructor 属性就不指向 Person 了。因为在创建函数时，也会创建它的 prototype 对象，同时会自动给这个原型的 constructor 属性赋值。上面的写法完全重写了默认的 prototype 对象，因此其 constructor 属性也指向了完全不同的新对象（Object 构造函数）。可以像下面一样在重写原型对象时专门设置一下它的值：

```
function Person() {}

Person.prototype = {
  constructor: Person,
  name: "Nicholas",
  age: 20,
  job: "doctor",
  sayName() {
    console.log(this.name);
  }
};
```

- 原生对象原型：是实现所有原生引用类型的模式，所有原生引用类型的构造函数（包括 Object、Array、String 等）都在原型上定义了实例方法。例如数组实例的 sort()方法就是在 Array.prototype 上定义的。虽然可以像修改自定义对象原型一样修改原生对象类型，但不推挤这样做，推荐的做法是创建一个自定义的类，继承原生类型。

- 原型模式的问题：共享特性，由于原型上所有的属性是在实例间共享的，但一般来说，不同的实例应该有属于自己的属性副本，这就是实际开放中通常不单独使用原型模式的原因。

21. 对象迭代

- ES2017 新增了两个静态方法 Object.values()和 Object.entries()，用于将对象内容转化为序列化的、可迭代的格式，返回它们内容的数组。Object.values()返回对象值的数组，Object.entries()返回键/值对的数组（非字符串属性会被转换为字符串输出）。

22. 继承

- 很多面向对象语言都支持两种继承：接口继承和实现继承。接口继承只继承方法签名，实现继承继承实际的方法。由于 ECMAScript 中函数没有签名，因此接口继承在 ECMAScript 中是不可能的，实现继承是 ECMAScript 唯一支持的继承方式，而这主要通过原型链实现。实现继承的方式有：
  （1）原型链； （2）盗用构造函数； （3）组合继承；

23. 原型链

- 原型链：每个构造函数都有一个原型对象，原型有一个 constructor 属性指向构造函数，而实例有一个内部指针指向原型。如果原型是另一个类型的实例，就意味着这个原型本身有一个内部指针指向另一个原型，相应地另一个原型也有一个指针指向另一个构造函数。这样就在实例和原型之间构成了一条原型链。示例：

```
function SuperType() {}

function SubType() {}

// 实现继承
SubType.prototype = new SuperType();
```

- 默认原型：默认情况下，所有引用类型都继承自 Object，这也是通过原型链实现的。任何函数的默认原型都是 Object 的一个实例，这也是自定义类型能够继承包括 toString()、valueOf()在内所有默认方法的原因。

- 原型和继承关系：原型和实例的关系可以通过两种方式来确定，
  （1）使用 instanceof 操作符；
  （2）使用 isPrototypeOf()方法，原型链中每个原型都可以调用这个方法，例如：

```
console.log(Object.isPrototypeOf(instance)); // true
console.log(SuperType.isPrototypeOf(instance)); // true
```

- 关于方法：子类有时候需要覆盖父类的方法，或者增加父类没有的方法。这些方法必须在原型赋值之后再添加到原型上。示例：

```
function SuperType() {
  this.property = true;
}
```

```
SuperType.prototype.getSuperValue = function() {
  return this.property;
};

function SubType() {
  this.subProperty = false;
}

// 继承
SubType.prototype = new SuperType();

// 覆盖已有方法
SubType.prototype.getSuperValue = function() {
  return false;
};

// 新方法
SubType.prototype.getSubValue = function() {
  return this.subProperty;
};
```

- 以对象字面量方式创建原型方法会破坏之前的原型链，因为这相当于重写了原型链。示例：

```
function SuperType() {}

SuperType.prototype.getSuperValue = function() {
  return true;
};


function SubType() {}

// 继承
SubType.prototype = new SuperType();
// 覆盖后的原型是Object的实例，不再是SuperType的实例，因此SubType与SuperType之间就没有关系了
SubType.prototype={
  getSubValue(){
    return false;
  }
  otherMethod(){
    return false;
  }
}

let instance = new SubType();
console.log(instance.getSuperValue());  // 报错!
```

- 原型链的问题：（1）共享特性，原型中的引用值会在所有实例间共享，这也是为什么属性通常会在构造函数中定义而不会定义在原型上的原因。
  （2）子类型在实例化时不能给父类型的构造函数传参。
  因此，原型链基本不会被单独使用。
  示例：

```
function SuperType() {
  this.colors = ["red", "blue", "white"];
}

function SubType() {}

SubType.prototype = new SuperType();

let instance1 = new SubType();
instance1.colors.push("black");

let instance2 = new SubType();
console.log(instance2.colors); //"red", "blue", "white","black"
```

24. 盗用构造函数

- 为解决原型包含引用值导致的继承问题，“盗用构造函数”的技术流行起来。思路：在子类构造函数中调用父类构造函数。因为函数是在特定上下文中执行代码的简单对象，所以可以使用 call()、apply()方法以新创建的对象为上下文执行构造函数。示例：

```
function SuperType() {
  this.colors = ["red", "green", "blue"];
}

function SubType() {
  SuperType.call(this);
}

let instance1 = new SubType();
instance1.colors.push("black");
console.log(instance1.colors);

let instance2 = new SubType();
console.log(instance2.colors);
```

- 优点：可以在子类构造函数中向父类构造函数传递参数。示例：

```
function SuperType(name) {
  this.name = name;
}
```

```
function SubType() {
  // 继承SuperType并传参
  SuperType.call(this, "Nicholas");

  this.age = 20;
}
```

- 盗用构造函数的问题：必须在构造函数中定义方法，因此函数不能重用。而且子类也不能访问父类原型上定义的方法。因此，盗用构造函数基本上也不单独使用。

25. 组合继承

- 组合继承（伪经典继承）综合了原型链和盗用构造函数。思路：使用原型链继承原型上的属性和方法，而通过盗用构造函数继承实例属性。组合继承是 JavaScript 中使用最多的继承模式，示例：

```
function SuperType(name) {
  this.name = name;
  this.colors = ["red", "green", "blue"];
}
SuperType.prototype.sayName = function() {
  console.log(this.name);
};

function SubType(name, age) {
  // 继承属性
  SuperType.call(this, name);

  this.age = age;
}

// 继承方法
SubType.prototype = new SuperType();

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

let instance1 = new SubType("Nicholas", 20);
instance1.colors.push("white");
console.log(instance1.colors);

let instance2 = new SubType();
console.log(instance2.colors);
```

26. 原型式继承

- 通过下面的 object()方法对传入的对象执行浅复制。

```

```

```
function object(o) {
  function F() {}
  F.prototpe = o;
  return new F();
}
```

- ES5 新增 Object.create()方法将原型式继承规范化，方法接收两个参数：作为新对象原型的对象，以及给新对象定义额外属性的对象（可选）。在只有一个参数时，Object.create()和上面的 object()方法效果相同。
- 原型式继承属性中包含的引用值始终会在相关对象间共享。

27. 寄生式继承

- 寄生式继承背后思路类似于寄生构造函数和工厂模式，具体思路：创建一个实现继承的函数，以某种方式增强对象，然后返回这个对象。

```
// object函数不是寄生式继承必需的，任何返回新对象的函数都能在这里使用
function createAnother(original) {
  let clone = object(original); // 通过调用object函数创建一个新对象
  clone.sayHi = function() {
    // 以某种方式增强这个对象
    console.log("Hi");
  };
  return clone; // 返回这个对象
}
```

使用示例：

```
let person = {
  name: "Nicholas",
  friends: ["Nick", "Van"]
};

let anotherPerson = createAnother(person);
anotherPerson.sayHi(); // "Hi"
```

- 缺点：与构造函数模式类似，寄生式继承给对象添加函数会导致函数难以重用。

28. 寄生式组合继承

- 寄生式组合继承：通过盗用构造函数继承属性，但使用混合式原型继承方法。

```
function inheritPrototype(subType, superType) {
  let prototype = object(superType.prototype); // 创建对象
  prototype.constructor = subType; // 增强对象
```

```
  subType.prototype = prototype; // 赋值对象
}
```

使用实例：

```
function SuperType(name) {
  this.name = name;
  this.colors = ["red", "green", "white"];
}

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

function SubType(name, age) {
  SuperType.call(this, name);
  this.age = age;
}

inheritPrototype(SubType, SuperType);
SubType.prototype.sayAge = function() {
  console.log(this.age);
};
```

- 寄生式组合继承算是引用类型继承的最佳模式。

29. 类

- 前面通过只使用 ES5 的特性来模拟类似于类的行为，ES6 新引入的 class 关键字具有正式定义类的能力。类（class）是 ECMAScript 中新的基础性语法糖结构，实际上背后使用的仍然是原型和构造函数的概念。
- 类的定义：与函数类型相似，定义类也主要有两种方式：类声明和类表达式。

```
// 类声明
class Person {}

// 类表达式
const Person = class {};
```

与函数表达式类似，类表达式在它们被求值前也不能引用。不过，与函数定义不同的是，虽然函数声明可以提升，但类定义不能。同时，函数受函数作用域限制，而类受块作用域限制。实例：

```
console.log(FunctionExpression); // undefined
var FunctionExpression = function() {};
console.log(FunctionExpression); // ƒ () {}
```

```
console.log(FunctionDeclaration); // FunctionDeclaration() {}

function FunctionDeclaration() {}
console.log(FunctionDeclaration); // FunctionDeclaration() {}

console.log(ClassExpression); // undefined
var ClassExpression = class {};
console.log(ClassExpression); // class {}

console.log(ClassDeclaration); // ReferenceError
class ClassDeclaration {}
console.log(ClassDeclaration); // class ClassDeclaration {}
```

- 类的构成：类可以包含构造函数方法、实例方法、获取函数、设置函数和静态类方法。但这些都不是必需的，空的类定义同样有效。

- 类构造函数：constructor 关键字。使用 new 操作符创建类的新实例时，会调用这个方法。具体分为以下几步：
  （1）在内存中创建一个新对象；
  （2）这个新对象的[[Prototype]]指针被赋值为构造函数的 prototype 属性；
  （3）构造函数内部的 this 被赋值为这个新对象（即 this 指向新对象）；
  （4）执行构造函数内部的代码（给新对象添加属性）；
  （5）如果构造函数返回非空对象，则返回该对象；否则返回刚创建的新对象。

```
class Person {
  constructor(override) {
    this.foo = "foo";
    if (override) {
      return {
        bar: "bar"
      };
    }
  }
}

let p1 = new Person(),
  p2 = new Person(true);

console.log(p1); // Person{foo:"foo"}
console.log(p1 instanceof Person); // true

console.log(p2); // Object{bar:"bar"}
console.log(p2 instanceof Person); // false
```

- 类构造函数和构造函数的主要区别：调用类构造函数必须使用 new 操作符，而普通构造函数如果不适用 new 调用，那么就会以全局的 this（通常是 window）作为内部对象。

```
function Person() {}

class Animal {}

// 把window作为this来创建实例
let p = Person();

let a = Animal(); // TypeError
```

- 类本身具有和普通构造函数一样的行为，

30. 扩展操作符

《工程数学线性代数 第六版》

1. 行列式

- 利用行列式求解二元线性方程组

$$ \begin{cases} a_{11}x_1+a_{12}x_2=b_1 \tag 1 \\ a_{21}x_1+a_{22}x_2=b_2 \\ \end{cases} $$

方程组(1)的解为：

$$ \begin{cases} x_1=\frac{b_1a_{22}-a_{12}b_2}{a_{11}a_{22}-a_{12}a_{21}} \\ & \text{if $a_{11}a_{22}-a_{12}a_{21} != 0$} \\ x_1=\frac{a_{11}b_2-b_1a_{21}}{a_{11}a_{22}-a_{12}a_{21}} \\ \end{cases} $$

使用行列式进行表示： $ D=\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ \end{vmatrix}, D_1=\begin{vmatrix} b_1 & a_{12} \\ b_2 & a_{22} \\ \end{vmatrix}, D_2=\begin{vmatrix} a_{11} & b_1 \\ a_{21} & b_2 \\ \end{vmatrix}$

则： $ x_1=\frac{D_1}{D}, x_2=\frac{D_2}{D} $

- 全排列：把 n 个不同的元素排成一列，叫做这 n 个元素的全排列。对于 n 个不同的元素，先规定各元素之间有个标准次序（例如规定从小到大为标准次序），当 1 对元素的先后次序与标准次序不同，就构成 1 个逆序。逆序数为奇数的排列叫做奇排列，逆序数为偶数的排列叫做偶排列。

- 对换：一个排列中任意两个元素对换，排列改变奇偶性。

- 行列式的一些性质：
  （1）行列式和它的转置行列式相等；
  （2）对换行列式的两行（列），行列式变号；
  （3）行列式的某一行（列）中所有元素都乘以同一数 K，等于数 K 乘此行列式；
  （4）行列式中有两行（列）元素成比例，则此行列式等于 0；
  （5）若行列式中某一行（列）的元素都是两数之和，则该行列式等于下列两个行列式之和；
  （6）把行列式的某一行（列）的各元素乘同一个数然后加到另一行（列）对应的元素上去，行列式不变。

- 代数余子式