第四次课 JavaScript核心技术解密

⏰ 回顾与补充

🐻 引入JavaScript

在 `HTML` 文件中使用 `JavaScript` ，首推外部引用，语法格式如下：

```
1 <script type="text/javascript" src="路径/文件名.js"></script>
```

🥑 数据类型

	A	B	C	D	E
1	Number	String	Boolean	Null	Undefined
2	Object	Array	Function		

其中，前五种：字符串 `String` 、数字 `Number` 、布尔 `Boolean` 、空 `Null` 、未定义 `Undefined` 为基本数据类型，后三种：对象 `Object` 、数组 `Array` 、函数 `Function` 为引用数据类型。

基本数据类型和引用数据类型的区别可以看如下例子：

```
1 let a = 'lanshan';
```

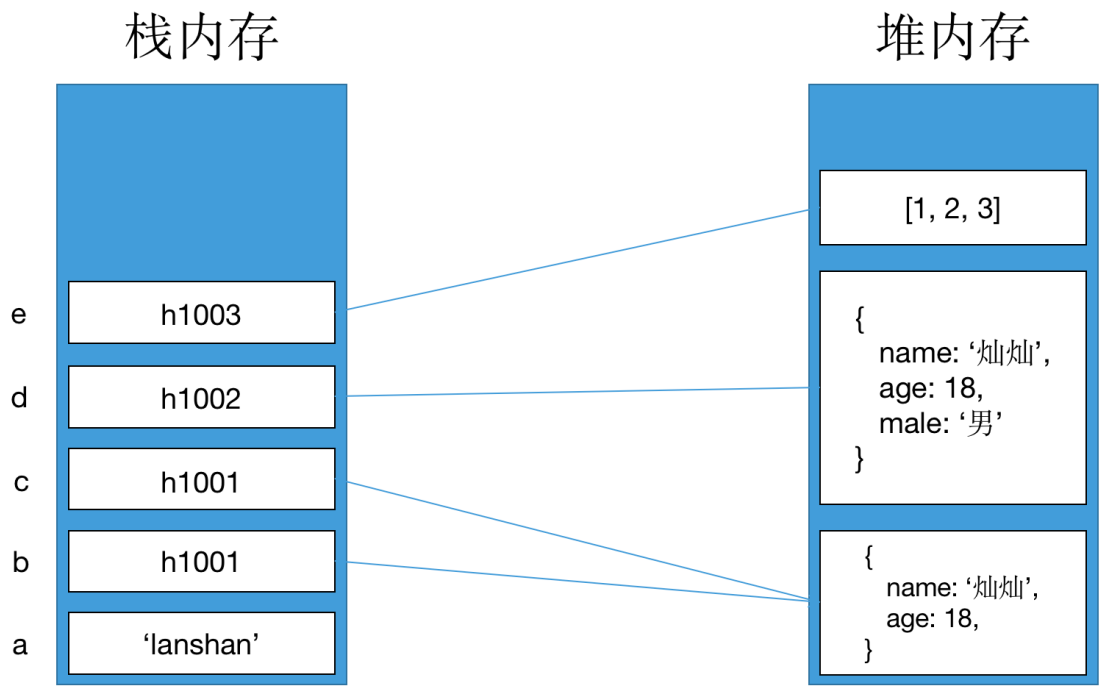
```
let b = {
  name: '灿灿',
  age: 18
}

let temp = a;
temp = 'frontend';
console.log(temp); // 'frontend'
console.log(a); // 'lanshan'

let c = b;
console.log(c);
// 打印:
// {
//   name: '灿灿',
//   age: 18
// }
c.male = '男';
console.log(c);
console.log(b);
// 20、21行都打印:
// {
//   name: '灿灿',
//   age: 18,
//   male: '男'
// }
```

从上例我们可以看到：

基本数据类型的变量保存的是原始值，他代表的值就是我们赋予的原始值；

引用数据类型的变量保存的是引用值，该引用值是指向堆内存空间的**一个地址**，而不是对象本身。

故上例我们将 `b` 赋值给 `c` 的时候，是将 `b` 变量存储的地址赋值给了 `c` 变量，从而调用 `c` 为对象添加属性，实际上也是为 `b` 所指向的对象添加属性。



const、let、var

1. 使用 `var` 声明的变量，其作用域为该语句所在的函数内，且存在变量提升现象。

   作用域为函数内：如下例所示，在函数中使用 `var` 声明的变量 `a` 可以访问到，在函数执行结束后，就会无法访问到。

```
function fun(){
  var a = 'lanshan'
  console.log(a) // "lanshan"
}
fun()
console.log(a) // 报错，Uncaught ReferenceError: a is not defined
```

```
"lanshan"
Uncaught ReferenceError: a is not defined
```

   变量提升：如下例所示。在 `var` 定义之前使用该变量，其值为 `undefined` 。

   变量提升的过程为：将要开始执行程序时，首先会找出所有用 `var` 定义的变量，将其声明并赋值为 `undefined` ，这阶段为创建阶段。之后再按照顺序依次执行代码。

   但要注意一点，函数声明提高的优先级比 `var` 声明提高的优先级高，即创建阶段之前函数就被声明了。

```
//===== 代码原本的样子 =====
console.log(a); // undefined
```

```
var a = 'lanshan';
console.log(a); // "lanshan"

//===== 代码实际执行的顺序 =====
var a = undefined;
console.log(a);
a = 'lanshan';
console.log(a);

//===== 函数优先级更高 =====
// 执行结果如下图所示
console.log(a);
var a = 'lanshan';
console.log(a);
function a () {
  console.log('优先级更高');
}
console.log(a);
```

```
function a() {
  console.log('优先级更高');
}
```

```
"lanshan"
```

```
"lanshan"
```

2. 使用 `let` 声明的变量，其作用域为该语句所在的代码块内（大括号内），不存在变量提升。

如下例所示，我们在 `if` 语句块内使用 `let` 声明的变量，在外部不能访问：

```
if(true) {
  let team = 'lanshan';
}
console.log(team); // Uncaught ReferenceError: team is not defined
```

```
if(true) {
  var number = '家余哥哥'
}
console.log(number); // '家余哥哥'
```

3. 使用 `const` 声明的是常量，在后面出现的代码中不能再修改该常量的值（如果是引用类型，可以修改引用类型所指向的数据）

```js
const arr = [0, 1, 2, 3];
const len = arr.push(4); // 数组末尾添加一个4，并返回数组添加后的长度
console.log(arr, len) // [0,1,2,3,4] 5

const obj = {
  name: '灿灿',
  age: 19
}
obj.male='男'; // 新设置性别的属性
console.log(obj); // 打印结果如下图所示
```

```
// [object Object]
{
  "name": "灿灿",
  "age": 19,
  "male": "男"
}
```

使用 `var` 可以重复声明、赋值变量，使用 `let` 、 `const` 不能重复声明变量。

```
var a = 1
var a = 2
let b = '1'     无法重新声明块范围变量“b”。
let b = '2'     无法重新声明块范围变量“b”。
const c = '1'     无法重新声明块范围变量“c”。
const c = '2'     无法重新声明块范围变量“c”。
```

三种变量的使用优先级：

- 不使用 `var` （容易污染全局作用域）
- `const` 优先， `let` 次之

🚢 运算符与类型转换

对于基本的算数运算符（ `+` , `-` , `*` , `/` , `%` 等），这里就不详细讲解了。

但对于字符串的相加有几点需要注意的地方：

```js
const str = '蓝山工作室';
const num = 888;
const boolean = true;
const nullVar = null;
const undefinedVar = undefined;

console.log(str + num);
```

```
8 console.log(str + boolean);
9 console.log(str + nullVar);
10 console.log(str + undefinedVar);
```

输出结果如下图所示：

```
"蓝山工作室888"
"蓝山工作室true"
"蓝山工作室null"
"蓝山工作室undefined"
```

重点需要掌握**与或运算符**。

或运算符 `||`

首先来看一下或运算符。两个竖线符号表示“或”运算符，如果参与运算的任意一个参数为 `true`，返回的结果就为 `true`，否则返回 `false`。

```
1 console.log( true || true );   // true
2 console.log( false || true );  // true
3 console.log( false || false ); // false
```

对于或运算符的实际应用场景，通常，逻辑或 `||` 会被用在 `if` 语句中，用来测试是否有任何给定的条件为 `true`。

```
1 let hour = 12;
2 let isWeekend = true;
3
4 if (hour < 7 || hour > 22 || isWeekend) {
5   console.log('开摆! '); // 非学习时间或者周末
6 }else{
7   console.log('开卷! ')
8 }
```

或运算符的返回值：或运算寻找并返回第一个真值或返回最后一个值。

该特性是 `JavaScript` 特有的。

```
let result = value1 || value2 || value3
```

或运算符的**执行步骤**如下：

- 从左到右依次计算操作数。
- 处理每一个操作数时，都将其转化为布尔值。如果结果是 `true` ，就停止计算，返回这个操作数。
- 如果所有的操作数都被计算过（也就是，转换结果都是 `false` ），则返回最后一个操作数。

与“纯粹的、传统的、仅仅处理布尔值的或运算”相比，这个规则就引起了一些很有趣的用法。

a. 获取变量列表或者表达式中的第一个真值

例如，我们有变量 `firstName` 、 `lastName` 和 `nickName` 。

我们用或运算 `||` 来选择有数据的那一个，并显示出来：

```
let firstName = "";
let lastName = "";
let nickName = "孙小龙";

alert( firstName || lastName || nickName || "找不到名字");
```

如果所有变量的值都为假，结果就是 "找不到名字"。

b. 短路求值

或运算符 `||` 的另一个用途是所谓的“短路求值”。

这指的是， `||` 对其参数进行处理，直到达到第一个真值，然后立即返回该值，而无需处理其他参数。

如果操作数不仅仅是一个值，而是一个有副作用的表达式，例如变量赋值或函数调用，那么这一特性的重要性就变得显而易见了。

```
true || alert("不会打印蓝山工作室"); // 不会打印
false || alert("会打印蓝山工作室"); // 会打印
```

与运算符 `&&`

之后我们再来学习一下与运算符。

两个 & 符号表示 `&&` 与运算符，当两个操作数都是真值时，与运算返回 `true` ，否则返回 `false` 。

```
alert( true && true );   // true
```

```
alert( true && false );  // false
alert( false && false ); // false
```

实际运用场景：

```
let hour = 12;
let minute = 30;
if (hour == 12 && minute == 30) {
  alert( '时间是12:30' );
}
```

与运算符的返回值：与运算寻找第一个假值或返回最后一个值。

与运算 `&&` 做了如下的事：

- 从左到右依次计算操作数。
- 处理每一个操作数时，都将其转化为布尔值。如果结果是 `false`，就停止计算，并返回这个操作数。
- 如果所有的操作数都是真值，则返回最后一个操作数。

与运算符在函数调用的条件下可以做出如下操作：

```
true && alert("李学长"); // 会打印
false && alert("吴学长"); // 不会打印
```

🏠 数组方法

首先我们创建一个新数组：

```
let array = [
  { id: 1, title: 'js数组方法1' },
  { id: 2, title: 'js数组方法2' }
]
```

增删改

`arr.push( )`：数组尾部添加元素，返回数组的新长度。

```
let obj = { id: 3, title: 'js数组方法3' }
```

```
array.push(obj)    // 返回数组长度: 3
console.log(array)
// [{ id: 1, title:'js数组方法1' },{ id: 2, title:'js数组方法2' },{ id: 3, title:'
```

`arr.unshift( )`：数组头部添加元素，返回数组的新长度。

```
let obj =  { id: 3, title:'js数组方法3' },
array.unshift(obj);  // 返回数组长度: 3
console.log(array)   // [{ id: 3, title:'js数组方法3' },{ id: 1, title:'js数组方法
```

`arr.pop( )`：从数组中删除最后一个元素，并返回该元素的值（删除的元素）。

```
array.pop();       // 返回删除元素: { id: 2, title:'js数组方法2' }
console.log(array)// [{ id: 1, title:'js数组方法1' }]
```

`arr.shift( )`：从数组中删除第一个元素，并返回该元素的值（删除的元素）。

```
array.shift();     // 返回删除元素: { id: 1, title:'js数组方法1' }
console.log(array) // [{ id: 2, title:'js数组方法2' }]
```

`arr.splice( )`：数组删除、插入和替换一个或多个元素；并返回删除元素的值。

实现**删除**：需要指定2个参数，要删除的项的位置和要删除的个数。

实现**插入**：向指定位置插入任意数量的元素，需提供3个参数：起始位置、 0（要删除的项数）和要插入的元素。

实现**替换**：向指定位置插入任意数量的项，且同时删除任意数量的项，需指定 3 个参数：起始位置、要删除的项数和要插入的任意数量的项。插入的项数不必与删除的项数相等。

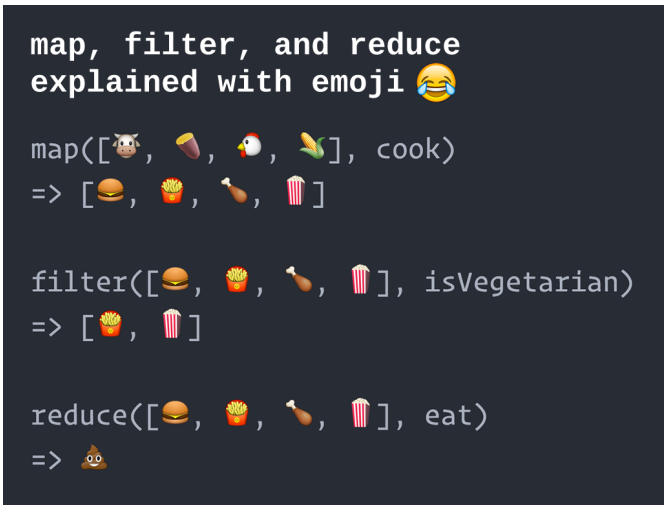
```
let array = [
  { id: 1, title: 'js数组方法1' },
  { id: 2, title: 'js数组方法2' },
  { id: 3, title: 'js数组方法3' }
]
// 删除
array.splice(0,1);    // 从索引为0开始删除1项
console.log(array);   // [{ id: 2, title:'js数组方法2' }, { id: 3, title: 'js数组
// 插入
```

```
array.splice(1,0,{ id: 3, title:'js数组方法3' });
console.log(array);
// [
//   { id: 2, title: 'js数组方法2' },
//   { id: 3, title: 'js数组方法3' },
//   { id: 3, title: 'js数组方法3' }
// ]

// 替换
array.splice(1,1,{ id: 3, title:'js数组方法3' });
console.log(array);
// [
//   { id: 2, title: 'js数组方法2' },
//   { id: 3, title: 'js数组方法3' },
//   { id: 3, title: 'js数组方法3' }
// ]
```

`arr.slice( )`：用于从数组中截取任意个元素，并返回截取后的新数组。

```
const arr = ['a', 'b', 'c', 'd', 'e', 'f'];
arr.slice(2, 4); // 提取索引为2的元素(包括)到索引为4的元素(不包括)
arr.slice(4, 2); // 空
arr.slice(2); // 从索引为2的元素开始提取(包含)，直到末尾
arr.slice(-2); // 负数会将其与长度相加，-2+6=4，也就是从第四个值开始（不包括第四个值）
```

循环、筛选



`arr.forEach( )` 方法用于调用数组的每一个元素，并将元素传递给回调函数。这个方法没有返回值。

第一个参数代表当前遍历的数组元素，第二个参数代表该数组元素在数组中的索引，第三个参数代表被遍历的数组。

```
array.forEach((value, index, arr) => {
  console.log(value,index,arr);
})
```

```
{ id: 1, title: 'js数组方法1' } 0 [
  { id: 1, title: 'js数组方法1' },
  { id: 2, title: 'js数组方法2' },
  { id: 3, title: 'js数组方法3' }
]
{ id: 2, title: 'js数组方法2' } 1 [
  { id: 1, title: 'js数组方法1' },
  { id: 2, title: 'js数组方法2' },
  { id: 3, title: 'js数组方法3' }
]
{ id: 3, title: 'js数组方法3' } 2 [
  { id: 1, title: 'js数组方法1' },
  { id: 2, title: 'js数组方法2' },
  { id: 3, title: 'js数组方法3' }
]
```

`arr.filter( )` 方法用于对数组进行过滤。它创建一个新数组，并返回新数组，新数组中的元素是通过检查原数组中符合条件的所有元素的集合。

同样需要接受三个参数，和 `forEach` 相同。这里由于不需要其他参数，只传入第一个参数。

```
let array = [
  { id: 1, title: 'js数组方法1' },
  { id: 2, title: 'js数组方法2' },
  { id: 3, title: 'js数组方法3' }
]

let res = array.filter((item) => {
  if (item.id === 1) return item
})
console.log(res) // [{ id: 1, title:'js数组方法1' }]
```

`arr.map( )` 方法返回一个新的数组，数组中的元素为原始数组调用函数处理后的值，也就是 `map()` 方法进行处理之后返回一个新的数组。

```
let array = [
  { id: 1, title: 'js数组方法1' },
```

```
  { id: 2, title: 'js数组方法2' },
  { id: 3, title: 'js数组方法3' }
]

let res = array.map(item => item.id );
console.log(res); // [1,2,3]
```

`arr.reduce( )` 方法对数组中的每个元素按序执行一个由你提供的回调函数，每一次运行回调函数时会将先前元素的计算结果作为参数传入，最后将其结果汇总为单个返回值。

```
const array1 = [1, 2, 3, 4];

const initialValue = 0;
const sumWithInitial = array1.reduce(
  (previousValue, currentValue) => previousValue + currentValue,
  initialValue
);

console.log(sumWithInitial); // 10
```

`arr.some( )` 方法用于检测数组中的元素是否满足指定条件，如果有一个元素满足条件，则表达式返回 `true` , 剩余的元素不会再执行检测。如果没有满足条件的元素，则返回 `false` 。

`arr.every( )` 方法方法用于检测数组所有元素是否都符合指定条件。如果数组中检测到有一个元素不满足，则整个表达式返回 `false` ，且剩余的元素不会再进行检测。如果所有元素都满足条件，则返回 `true` 。

`arr.findIndex( )` 方法返回数组中满足条件的第一个元素的索引（下标）, 如果没有找到，返回-1。

`arr.find( )` 方法返回数组中满足条件的第一个元素的值，如果没有，返回 `undefined` 。

`arr.includes( )` 方法用来判断一个数组是否包含一个指定的值，如果是返回 `true` ，否则 `false` 。

`arr.indexOf( )` 方法判断数组中是否存在某个值，如果存在返回数组元素的下标，否则返回-1。

合并

`arr.concat( )` 方法用于合并两个或多个数组。此方法不会更改现有数组，返回值是一个新数组。

```
let array2 = [
    { id: 3, title:'js数组方法3' },
    { id: 4, title:'js数组方法4' }
]
```

```
array.concat(array2)
console.log(array)
// [{ id: 1, title:'js数组方法2' },{ id: 2, title:'js数组方法2' }{ id: 3, title:'j
```

还可以利用ES6语法中的扩展运算符进行数组的合并。

数组排序

`arr.sort( )` 方法将数组里的项从小到大排序。注意： `sort( )` 方法**比较的是字符串，没有按照数值的大小对数字进行排序，传入的若是数组，会先转化为字符串再排序**。

```
let arr1 = ["a", "d", "c", "b"]
console.log(arr1.sort()) // ["a", "b", "c", "d"]

let arr2 = [99, 23, 11, 4, 5, 1]
console.log(arr2.sort()) // [ 1, 11, 23, 4, 5, 99 ]
```

那么我们要实现数值的排序应该怎么办呢？这时候就要引入我们的 `sort` 函数自定义功能了。

我们可以在 `sort( )` 中传入一个自定义函数，构造规则一般如下：

- 设置两个参数，一般为 `x` 、 `y` ，可以简单地理解为x代表前面的元素，y代表后面的元素。
- 用分支语句进行条件判断
- 如果不符合预期排序前后顺序的则 `return 1` ，符合预期顺序的 `return -1` ，相等时 `return 0` 。

```
let arr = [99, 23, 11, 4, 5, 1]
arr.sort(function (x, y) {
  if (x < y) {
    return -1
  }
  if (x > y) {
    return 1
  }
  return 0
})

console.log(arr)
// [ 1, 4, 5, 11, 23, 99 ]
```

在这里我们发现，如果`x<y`，这说明了小的在前面，符合了我们的预期，于是`return -1`。`x>y`是大的数在前面，不符合预期，我们返回1。

我们还可以简写为：

```
let arr = [99, 23, 11, 4, 5, 1]
arr.sort((a, b) => a - b)
console.log(arr)
// [ 1, 4, 5, 11, 23, 99 ]
```

`arr.reverse( )`方法用于颠倒数组中元素的顺序。

数组转字符串

`arr.join( )`方法就是把数组转换成字符串，然后给他规定个连接字符，默认的是逗号`,`。原数组不会被改变。

```
let arr = [1,2,3];
console.log(arr.join());        // 1,2,3
console.log(arr.join("-"));      // 1-2-3
```

🚗 ES6部分语法

箭头函数

ES6 允许使用“箭头”`=>`定义函数。

箭头函数相当于一个匿名函数，并且简化了函数定义。

括号内表达传入的参数，如果只有一个参数，括号可以省略，如果有多个参数或没有参数，则不能省略。

箭头后面为函数体，用大括号包裹。如果只含返回值，大括号可以省略。

```
let f = () => 5;
// 等同于
let f = function () { return 5 };

let sum = (num1, num2) => num1 + num2;
// 等同于
let sum = function(num1, num2) {
  return num1 + num2;
};
```

```
10
11 let foo = (name) => {
12     if(name === "蓝山工作室"){
13         return "是蓝山工作室"
14     }
15     return '不是蓝山工作室'
16 }
17 foo('蓝山工作室') // 是蓝山工作室
```

rest参数

ES6 引入 `rest` 参数（形式为 `...变量名`），用于获取函数的多余参数。

rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

```
1 const add = (.。 .nums) => {
2   let sum = 0;
3   for (let i of nums) {
4     sum += i;
5   }
6   return sum;
7 }
8
9 console.log(add(2, 5, 3, 4)) // 14
```

```
1 const add = (a, b, ...nums) => {
2   let sum = 0;
3   for (let i of nums) {
4     sum += i;
5   }
6   return sum;
7 }
8
9 console.log(add(2, 5, 3, 4)) // 7
```

扩展运算符

当你在JavaScript中看到 `...` 时，它还可能代表扩展运算符。

```
1 function test (a, b, c) {
2   console.log(a)
3   console.log(b)
```

```
  console.log(c)
}

let arr = [1, 2, 3]
test(...arr)
// 打印结果
// 1
// 2
// 3
```

也可以用来**拼接数组**：

```
let arr1 = [1, 2, 3]
let arr2 = [...arr1, 4, 5, 6]
console.log(arr2)
// 打印结果
// [1, 2, 3, 4, 5, 6]
```

字符串转数组：

```
let str = 'test'
let arr3 = [...str]
console.log(arr3)
// 打印结果
//  ["t", "e", "s", "t"]
```

结构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构赋值。

数组解构赋值：

```
let a = 1;
let b = 2;
let c = 3;

let [a, b, c] = [1, 2, 3];
```

数组解构赋值的部分例子如下：

```
let [foo, [[bar], baz]] = [1, [[2], 3]];
foo // 1
bar // 2
baz // 3

let [ , , third] = ["重邮", "蓝山", "工作室"];
third // "工作室"

let [x, , y] = [1, 2, 3];
x // 1
y // 3

let [head, ...tail] = [1, 2, 3, 4];
head // 1
tail // [2, 3, 4]
```

也可以在数组解构赋值中设定默认值：

```
let [x, y = 'b'] = ['a']; // x='a', y='b'
let [x, y = 'b', z] = ['a', , '工作室'] // x='a', y='b', z='工作室'
```

对象解构赋值：

```
let { foo, bar } = { foo: 'aaa', bar: 'bbb' };
foo // "aaa"
bar // "bbb"
```

和数组一样，对象的解构赋值也可以嵌套使用：

```
let obj = {
  p: [
    'Hello',
    { y: 'World' }
  ]
};

let { p: [x, { y }] } = obj;
x // "Hello"
y // "World"
```

对象解构赋值也可以设置默认值：

```
let {x, y = 5} = {x: 1};
x // 1
y // 5
```

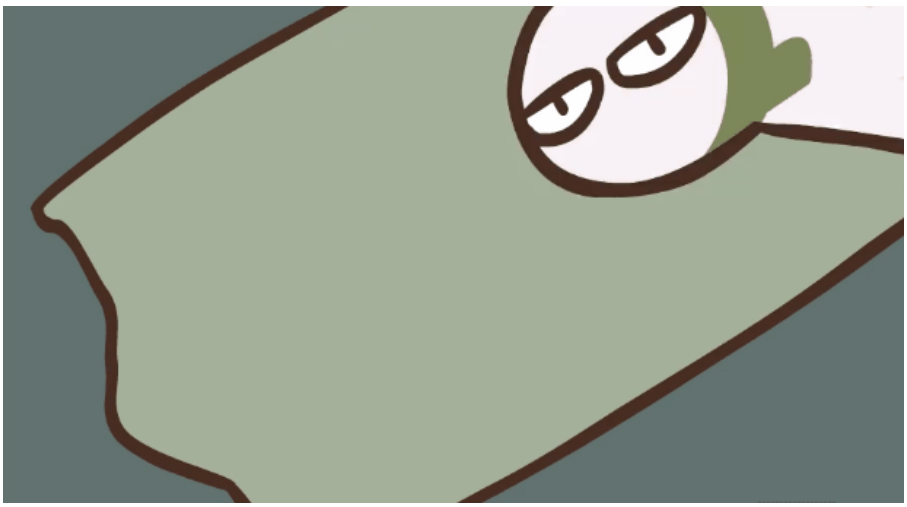
🧩 作用域与作用域链

🍔 栈、堆、队列

栈

栈是一种数据结构，它表示的是数据的一种存取方式。栈可以用来规定代码的执行顺序，在`JavaScript`中叫做**函数调用栈（call stack）**。

如图所示，羽毛球盒中依次放入羽毛球，当想取出来的时候，首先取走的是处于顶层的羽毛球4，它一定是最后被放进去且最先被取出来的。而若想取出羽毛球1，必须先将上面的所有羽毛球全部取出后才能取出，但羽毛球1是最先被放进去的。



这种存取方式的特点可以总结为**先进后出，后进先出（LIFO，last in，first out）**。

在`JavaScript`中，数组提供了两个栈方法来对应这种存取方式：

`push`：向数组末尾添加元素（进栈方法）

`pop`：弹出数组最末尾的一个元素（出栈方法）

```
let arr = [1, 2, 3, 4];
arr.push(5); // [1, 2, 3, 4, 5]
arr.pop(); // [1, 2, 3, 4]
```

堆

堆通常是一种树状结构。

堆的存取方式与在书架上取书非常相似。书摆放在书架上，我们只要知道书的名字，在书架上找到之后就可以很方便地取出，我们甚至不用关系书的存放顺序，即不用像从乒乓球盒子中取乒乓球一样，需要有先后顺序。

我们可以用对象的形式体现出来：

```
let bookcase = {
  Hamlet: 'C12311',
  theHobbit: 'B1231',
        harryPotter: {
    part1: 'A2113',
    part2: 'A2131',
    part3: 'A2132'
  }
}
```

当我们想要访问哈姆雷特时，只需要通过 `bookcase.Hamlet` 来访问即可，或者想访问哈利波特第二部，只需要调用 `bookcase.harryPotter.part2` 来访问。而不用关心书籍和部数的具体顺序。

队列

队列是一种先进先出（**FIFO**，fisrt in，first out）的数据结构。正如排队过安检一样，排在队伍前面的人一定是最先过安检的人。

在 `JavaScript` 中，理解队列数据结构的目的是为了搞清楚事件循环（`Event loop`）的机制。在后面学长应该会讲到。

♻ 内存空间

基本数据类型和引用数据类型

垃圾回收机制

`JavaScript` 有垃圾自动回收机制，所以对于前端开发人员来说，内存分配并不是一个经常被提及的概念。

```
let a = 20;
alert(a + 100);
a = null;
```

分别对应如下三个过程：

- 分配内存
- 使用分配到的内存
- 不需要时释放内存

这里要重点理解第三个过程，`JavaScript`的垃圾回收主要依赖“引用”的概念，当一块内存空间中的数据能够被访问时，垃圾回收器就认为“该数据能够被获得”，不能被获得的数据，就会被打上标记，并回收内存空间，即为**标记清除算法**。

这个算法使垃圾回收器会找到所有可以获得与不能访问的数据。

如上面那个例子，我们将`a`设置为`null`时，最开始给其分配的20，就无法被访问到，很快会被自动回收。如果一个引用数据类型的值被改变，那么其指向的数据也无法被访问到，很快会被自动回收掉。

在局部作用域中，当函数执行完毕后，局部变量就会被清除，因此垃圾回收集器很容易做出判断并回收。但是在全局作用域中，变量什么时候被释放垃圾收集器很难判断，所以我们需要尽可能避免使用全局变量，尽量不使用`var`声明变量。如果使用了全局变量，则建议不使用它时，通过`a=null`来手动释放。

🎱 执行上下文

`JavaScript`代码在执行时，会进入一个执行上下文中。执行上下文可以理解为当前代码的运行环境。

当代码在`JavaScript`中运行时，执行代码的环境非常重要，可以分为以下两类：

- 全局环境：代码运行起来后会首先进入全局环境。
- 函数环境：当函数被调用执行时，会进入当前函数中执行代码。

JavaScript引擎会以栈的方式来处理它们，这个栈就是之前提到过的函数调用栈，其规定了`JavaScript`代码的执行顺序。**栈底永远是全局上下文，栈顶是当前正在执行的上下文。**

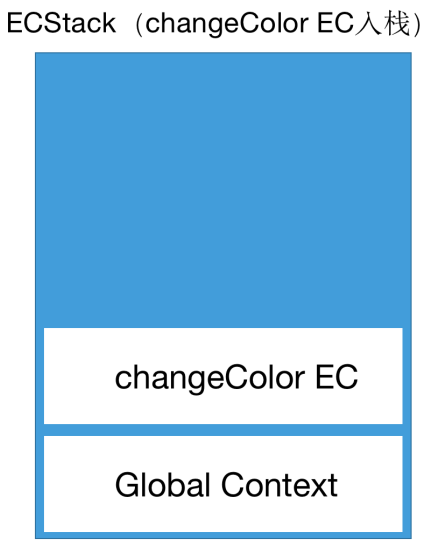
```
let color = 'blue'

const changeColor = () => {
  let anotherColor = 'red'

  const swapColors = () => {
    [color, anotherColor] = [anotherColor, color]
    console.log(color, anotherColor) // red blue
  }

  const setColor = () => {
```

```
12            color = 'yellow'
13      console.log(color)
14    }
15
16    swapColors()
17    setColor()
18  }
19  changeColor()
```

我们用ECStack（Execution Context Stack，执行上下文堆栈）来表示处理执行上下文的堆栈。

1. 第一步，全局上下文（Global Context）入栈，并一直处于栈底。

2. 第二步，从可执行代码开始执行，直到遇到`changeColor( )`，从而创建`changeColor`自己的执行上下文，changeColor EC入栈。

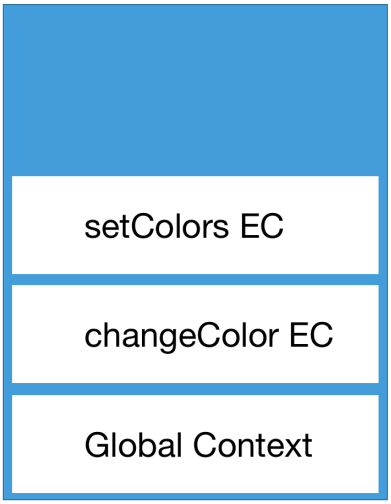


3. 第三步，changeColors EC入栈后，开始执行changeColors内可执行的代码。遇到 `swapColors( )` 这句代码后激活 `swapColors( )` ，swapColors EC入栈。

4. 第四步，执行swapColors内可执行的代码，其中没有能生成其他能生成执行上下文的情况，因此这段代码顺利执行完毕 🎉，swapColors的执行上下文从栈中弹出。

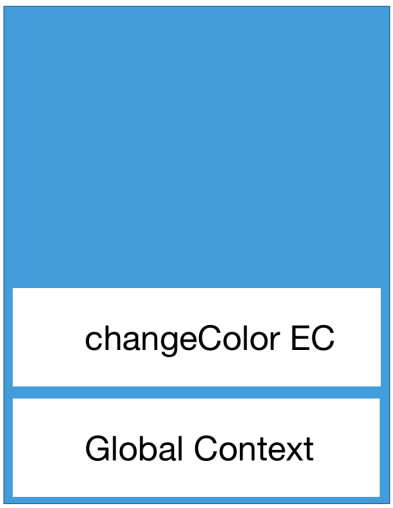
5. 第五步，继续执行之前changeColor内的可执行代码，遇到 `setColor( )` ，生产setColor的执行上下文，setColor EC入栈。

ECStack（setColors EC入栈）



6. 第六步，执行setColor内可执行的代码，其中没有能生成其他能生成执行上下文的情况，因此这段代码顺利执行完毕 🎉，setColors的执行上下文从栈中弹出。

ECStack（setColors EC出栈）



7. 第七步，继续执行之前changeColor的可执行代码，没有再遇到其他执行上下文，顺利执行完毕后弹出。这样ECStack中就只剩全局上下文了。

8. 最后，全局上下文在浏览器窗口关闭后出栈。

🪵 作用域与作用域链

在 `JavaScript` 中，作用域是用来规定变量与函数可访问范围的一套规则。

作用域

常见的作用域有三种，分别是全局作用域、函数作用域、块级作用域。

全局作用域：

全局作用域中声明的变量与函数可以在代码的任何地方被访问。

一般来说一下三种情况拥有全局作用域：

1. 全局对象下拥有的属性和方法

```
window.name
window.location
window.top
```

2. 在最外层声明的变量与方法

   我们知道全局上下文的变量对象实际上就是 `window` 对象，因此在全局上下文中声明的变量与方法就是 `window` 的属性和方法，所以也具有全局作用域。

3. 在非严格模式下，函数作用域中定义但没有声明的变量和方法，会自动变成全局对象 `window` 的属性。

```
function foo() {
  bar = 20;
```

```
3 }
```

函数作用域：

函数作用域中声明的变量和方法，只能被下层子作用域访问，而不能被其他不相干的作用域访问。

```
function parent() {
  let bar = 20;
  function child() {
          console.log(bar);
  }
}
```

块级作用域：

在ES6之前，没有块级作用域，作用域范围为大括号内的区域。

```
let arr = [1, 2, 3, 4, 5]

for (var i = 0; i < arr.length; i++) {
  console.log(i)
}
console.log(i) // i = 5

for (let j = 0; j < arr.length; j++) {
  console.log(j)
}
console.log(j) // ReferenceError: j is not defined
```

尝试模拟一个作用域：

上面 `for` 循环例子中， `i` 值在作用域中仍可以被访问，那么这个值就会对作用域中其他同名变量造成干扰，因此我们通过函数自执行的方式来生成一个作用域。

```
var arr = [1, 2, 3, 4, 5]

(function () {
  for (var i = 0; i < arr.length; i++) {
    console.log(i)
  }
})();

console.log(i) // ReferenceError: i is not defined
```

当我们使用ES5时，常通过函数自执行来实现模块化，模块化是实际开发中需要重点掌握的开发思维。

作用域链

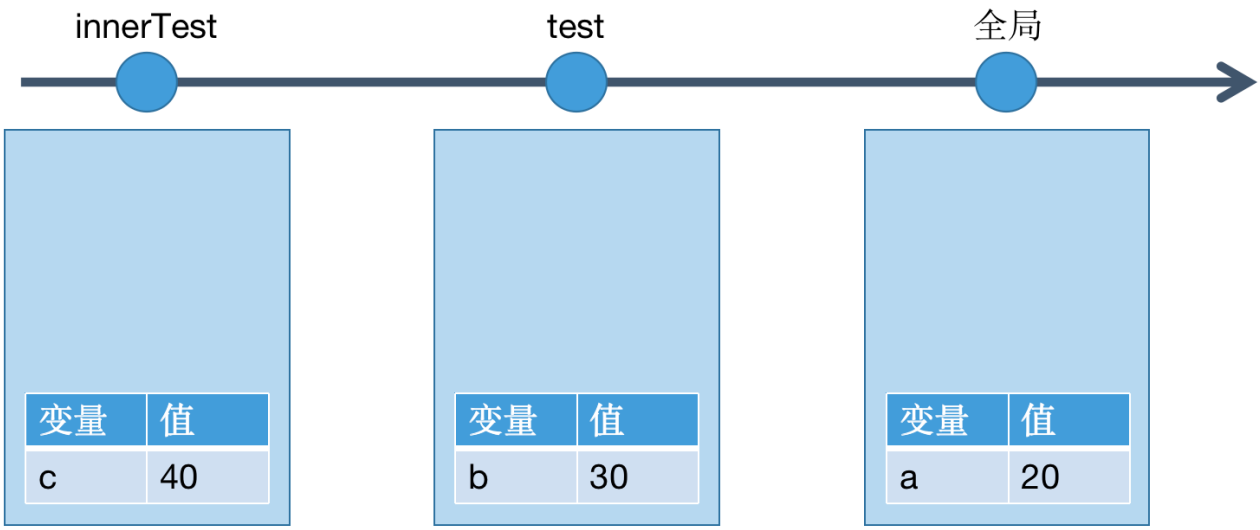
作用域链（Scope Chain）是由当前执行环境与上层执行环境的一系列变量对象组成的，它保证了当前执行环境对符合访问权限的变量和函数的有序访问。

如果是第一次接触作用域链的概念，可能对上面这句话的理解有一点困难，我们可以先来看一个例子：

```
let a = 20

function test () {
  let b = a + 10

  function innerTest () {
    let c = b + 10
    console.log(c) // 40
  }
  innerTest()
}

test()
```

在上面的例子中，先先后生成了全局上下文、函数 `test` 的执行上下文、 `innerTest` 的执行上下文。那么 `innerTest` 的作用域链则同时包含了这三个变量对象。

很多人会用父子关系或包含关系来理解当前作用域与上层作用域之间的关系，但用一个单方向通道来理解可能更加准确。



可以用一个数组来表示作用域链的有序性。数组的第一项 `scopeChain[0]` 为作用域链的最前端，而数组的最后一项则为作用域链的最末端。所有作用域链的最末端都是全局变量对象。

作用域链的有序性让我们知道其访问变量或函数的优先级顺序：

```
let a = 20

function test () {
  let b = a + 10

  function innerTest () {
    let b = 100 // 在innerTest中新定义一个b
    let c = b + 10
    console.log(c) // 110
  }
  innerTest()
}

test()
```

如上例，`innerTest` 中会先在自己的作用域中查找是否有 `b` 这个变量，有则直接使用，没有才会到之后的作用域中查找。

理解作用域链至关重要，但更多的知识需要结合闭包来理解。

🍞 闭包

大家在初学的时候对于闭包都是望而生畏的，面试必问，可理解起来又十分费劲。

🍋 概念

什么是闭包？我们可以从一个例子入手：

```
function fun () {
  let num = 0
  function add () {
    num++
    console.log(num)
  }
  return add
}

let addFun = fun()
addFun() // 1
addFun() // 2
addFun() // 3
console.log(num) // ReferenceError: num is not defined
```

这个例子中，我们在 `fun` 函数里又定义了 `add` 函数，并将 `add` 函数返回给全局上下文进行使用。

正常情况下，函数执行完成，内部变量会被销毁（释放内存空间）。

通过打印结果可以看到，当 `fun` 函数执行完成之后，其内部的变量 `num` 本该跟着函数执行完成一起被清除，但 `fun` 函数返回的 `add` 函数赋值给 `addFun` 后， `addFun` 内部可以却访问之前 `fun` 函数定义的 `num` ，但全局作用域无法访问。

这就形成了一个闭包。当一个函数 `a` 内部声明了另一个函数 `b` ，函数 `b` 中使用了函数 `a` 中声明的变量或者函数，并将函数 `b` 赋值给全局变量或者作为返回值传给全局变量，就形成了闭包。

🍰 特点

1. 可以引用外部函数的作用域
2. 可以创建私有作用域，且不会被回收内存。
3. 容易导致内存泄露

🍾 应用

可以利用闭包来实现模块化封装：

```
function create () {
  let num1 = 1
  let num2 = 1
  function add () {
```

```
    return num1 + num2
  }
  function multiply () {
    return num1 * num2
  }
  return {
    add,
    multiply
  }
}
const calculate = create()

let answer = calculate.add()
console.log(answer)
answer = calculate.multiply()
console.log(answer)
```

该段代码利用闭包封装了一个计算函数，并且外部访问不到内部的私有变量。

🐱 this

💘 什么是this

想要知道一个东西是什么，我们首先可以直接将其打印出来看看：

```
console.log(this)
```

```
index2.js:1
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, …}
```

打印出来是 `window` 对象，即直接打印 `this` 指向全局对象。

如果我们在函数体内打印 `this` ：

```
function fun () {
        console.log(this); // 还是window对象
}
fun()
```

打印出来仍是 `window` 全局对象。

由于 `window` 对象扮演着浏览器中的全局对象的角色，因此所有在全局作用域中声明的变量，函数都会变成 `window` 对象的属性和方法。全局函数就是全局对象的方法。

```
window.fun()
```

现在我们调用对象中的方法打印 `this` 。

```
const name = '蓝山'
const robot = {
  name: '蓝妹',
  sayName () {
    console.log(this)
    console.log(`我是${this.name}`)
  }
}
robot.sayName()
```

```
▶ {name: '蓝妹', sayName: f}                    index2.js:4
我是蓝妹                                        index2.js:5
```

验证了我们的想法： `this` 指向调用这个方法的对象，即谁调用它就指向谁。

箭头函数中的this

对于谁调用 `this` 就 `this` 就指向谁，仅适用于非箭头函数的函数。对于特殊的箭头函数， `this` 指向需要单独来研究。

箭头函数外面指向谁，就指向谁。

实例如下，首先我们用普通函数的方式来实现如下代码：

```
let obj = {
  hp: 555,
  sayHp: function () {
    console.log(this.hp)
  }
}

obj.sayHp() // 555
```

如果我们使用箭头函数的方法来实现：

```
let obj = {
  hp: 555,
  sayHp: () => {
    console.log(this.hp)
  }
}

obj.sayHp() // undefined
```

会发现打印出来为 `undefined` 。

```
let obj = {
  hp: 555,
  that: this, // window对象
  sayHp: () => {
    console.log(this.hp)
    console.log(this) // window对象
  }
}
console.log(obj.that) // window对象
obj.sayHp()
```

🐶 call、apply、bind

`call` 、 `apply` 、 `bind` 都是改变this指向的函数的方法。

`call` :

1. `call` 可以立即调用并执行函数:

```
function fun () {
    console.log('蓝山工作室')
}
fun.call() // 蓝山工作室
```

2. `call` 可以改变函数中 `this` 的指向:

```
function fun () {
  console.log(this.name)
}

```

```
const cat = {
  name: '李荣浩'
}

fun.call(cat) // 李荣浩
```

```
const cat = {
  name: '黛玉'
}

const dog = {
  name: '芬达',
  sayName () {
    console.log(`我是${this.name}`)
  }
}

dog.sayName() // 我是芬达
dog.sayName.call(cat) // 我是黛玉
```

3. 考虑到传参的情况：

```
const dog = {
  name: '芬达',
  eat (food) {
    console.log(`我是${this.name}，我要吃${food}`)
  }
}

const cat = {
  name: '黛玉'
}

dog.eat('骨头') // 我是芬达，我要吃骨头
dog.eat.call(cat, '鱼') // 我是黛玉，我要吃鱼
```

`call` 的第一个参数是改变 `this` 指向的对象，从第二个参数开始，之后的参数作为函数调用的参数传递。

```
const dog = {
  name: '芬达',
```

```
  eat (food, hobby) {
    console.log(`我是${this.name}，我要吃${food}，喜欢${hobby}`)
  }
}

const cat = {
  name: '黛玉'
}

dog.eat.call(cat, '鱼', '睡觉') // 我是黛玉，我要吃鱼，喜欢睡觉
```

`apply`：

`call` 和 `apply` 的区别就在于传参的方式不同：

```
const dog = {
  name: '芬达',
  eat (food, hobby) {
    console.log(`我是${this.name}，我要吃${food}，喜欢${hobby}`)
  }
}

const cat = {
  name: '黛玉'
}

dog.eat.apply(cat, ['鱼', '睡觉']) // 我是黛玉，我要吃鱼，喜欢睡觉
```

`bind`：

```
const dog = {
  name: '芬达',
  eat (food, hobby) {
    console.log(`我是${this.name}，我要吃${food}，喜欢${hobby}`)
  }
}

const cat = {
  name: '黛玉'
}

dog.eat.bind(cat, '鱼', '睡觉') // 什么都没有被打印
```

`bind` 不会立即执行函数。

```
const dog = {
  name: '芬达',
  eat (food, hobby) {
    console.log(`我是${this.name}，我要吃${food}，喜欢${hobby}`)
  }
}

const cat = {
  name: '黛玉'
}

const fun = dog.eat.bind(cat, '鱼', '睡觉') // 什么都没打印
fun() // 我是黛玉，我要吃鱼，喜欢睡觉
```

`bind` 会返回一个已经改变好 `this` 指向的函数，方便我们在后面自行调用执行。

🧬 原型和原型链

🔧 构造函数

之前大家可能利用过构造函数 `new Object( )` 创建过对象，或者使用过 `new Array( )` 、`new Date( )` 等。但大家有没有想过为什么可以用这种方法创建对象呢？ `new` 到底干了啥？

实质上， `Object` 是一个构造函数。

构造函数本身就是一个函数，与普通函数没有任何区别，不过为了规范一般将其首字母大写。构造函数和普通函数的区别在于，使用了 new 生成实例的函数就是构造函数，直接调用的就是普通函数。

```
function Fun () {
        console.log('我是构造函数');
}

let temp = new Fun();
console.log(temp);
```

```
我是构造函数                                    index2.js:2
▶ Fun {}                                        index2.js:6
```

在这个例子中生成的实例就是 `temp` ，通过打印可以判断出类型为一个对象。首先可以确定的是 `new` 关键字执行了Fun函数内的语句，并返回了一个对象。那我们就可以清楚构造函数就是用来创建

对象的。

如果只是创建一个空对象肯定就没有什么意思，我们可以稍加更改：

```
function Father(name) {
    this.name = name;
}
let son = new Father('Lisa');
console.log(son); //Father {name: "Lisa"}
```

```
▶ Father {name: 'Lisa'}                                    index2.js:5
```

这样使用构造函数创建对象，就能给对象赋予属性或方法。

这个例子中的 `this` 指向了 `son` ，也就是实例对象。

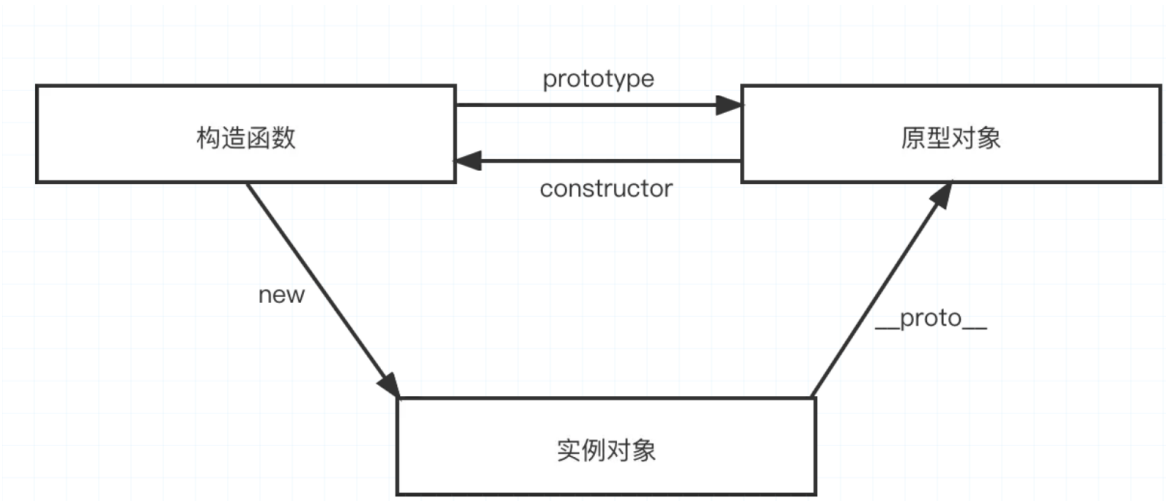
那我们就可以来总结一下 `new` 到底做了什么：

`new` 命令的作用，就是执行一个构造函数，并且返回一个对象实例。使用 `new` 命令时，依次执行下面的步骤：

1. 创建一个空对象，作为将要返回的对象实例。
2. 将空对象的原型指向了构造函数的 `prototype` 属性。
3. 使构造函数内部的 `this` 关键字指向创建的空对象。
4. 开始执行构造函数内部的代码，返回创建的实例对象。

大家到这里应该理解了1、2、4点，但是要能更加深入地理解构造函数，我们需要学习原型和原型链的相关知识。

🎥 原型概述



每个对象都有它的原型对象，它可以使用自己原型对象上的所有属性和方法。

这就引出了继承的概念。每个对象都有一个属性 `__proto__` 指向它的原型。示例如下：

```
const cat = {
  name: '黛玉'
}

cat.__proto__.sayName = function () {
  console.log(this.name)
}

cat.sayName() // 黛玉
```

该例子中， `cat` 对象本身并没有 `sayName` 的方法，我们通过 `__proto__` 在 `cat` 的原型上添加了 `sayName` 方法， `cat` 可以成功调用。

通过 `__proto__` 找到原型是第一种方法，我们还可以通过构造函数的 `prototype` 属性找到原型。

```
function Cat (name, age) {
  this.name = name
  this.age = age
}

const cat = new Cat('黛玉', 18)
Cat.prototype.sayName = function () {
  console.log(this.name)
}
cat.sayName()
```

通过对象本身的 `__proto__` 或者其构造函数的 `prototype` 获取的原型是相同的：

```
function Person(name, age){
    this.name = name;
    this.age = age;
}

Person.prototype.motherland = 'China'
// 这样就可以在Person上面添加motherland属性，值为china

let person1 = new Person('小明', 18); // 通过new关键字创建一个实例对象

console.log(Person.prototype.constructor == Person) // true
console.log(person1.__proto__ == Person.prototype) // true
```

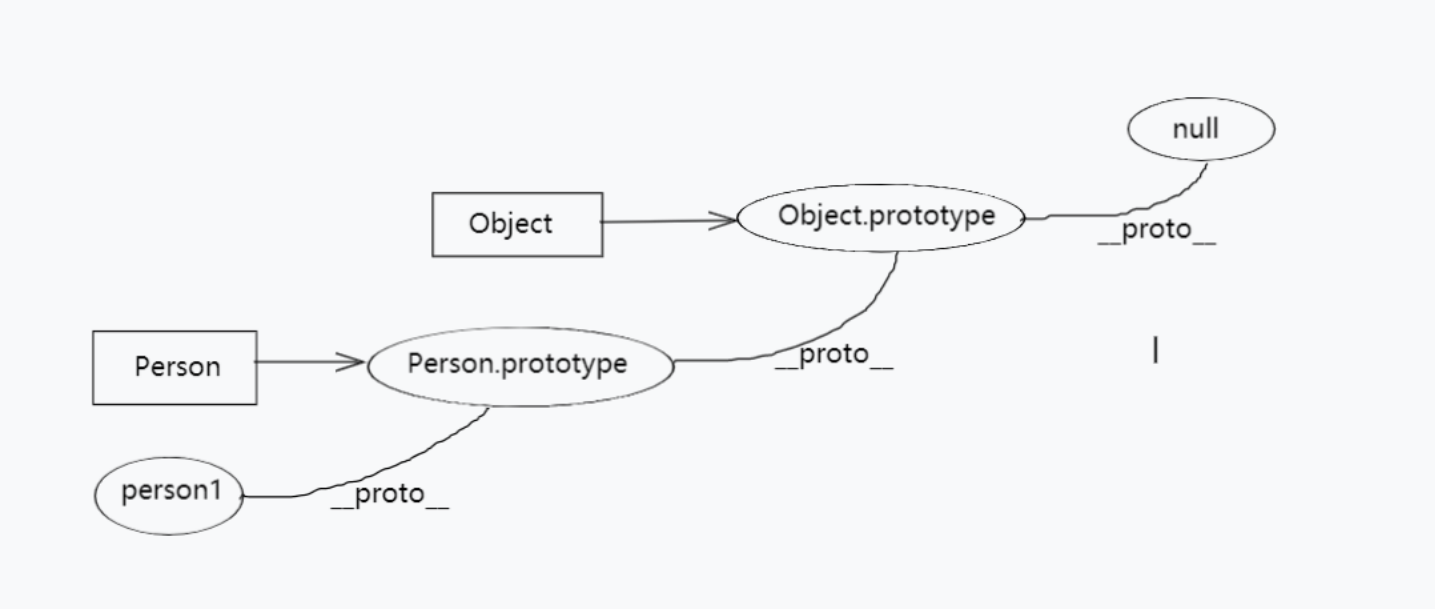
原型对象上还有一个属性：`constructor`，指向构造函数，用来记录实例是由哪一个构造函数创建的。

🔗 原型链

当你试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么它会去它的原型 `__proto__`（也就是它的构造函数的显式原型 `prototype`）中去寻找。

我们可以猜测：原型它是否能是一种链式结构?

原型对象也可能拥有原型，并从中继承方法和属性，一层一层、以此类推。这种关系常被称为原型链 (`prototype chain`)，它解释了为何一个对象会拥有定义在其他对象中的属性和方法。



如图所示，对象的原型链会一直延续到 `Object.prototype` 的 `__proto__`，即原型链的终点为 null。

📚 练习

作业一

用尽可能多的方法实现数组降维。

```
输入为[1, 2, [3, 4, 5, [6, 7], [8, 9, [10]]], [11, 12]]
输出为[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

作业二

利用rest参数，实现一个累积函数，不管传入多少参数都能够输出正确结果。

作业三

查阅函数柯里化的相关资料，通过自学，实现如下功能的函数：

```
add(1)(2)(3); // 6
```

作业四

自己封装（实现）call方法。（尽力去完成）

作业五

理解基本数据类型和引用数据类型后，查阅相关资料，尝试自己实现一个浅拷贝和深拷贝。（尽力去完成）