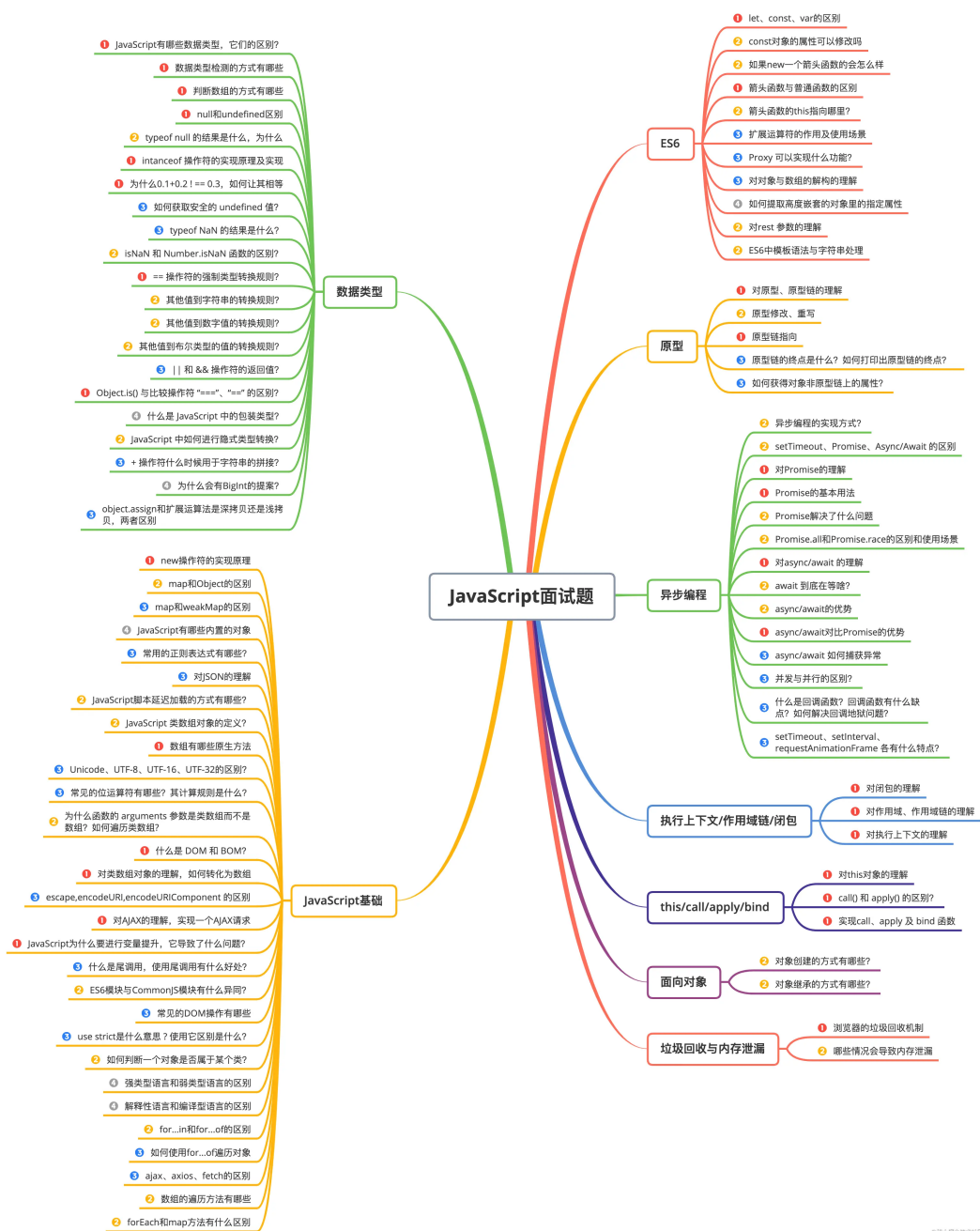


## 前端面试题系列文章：

- 【1】 [「2021」高频前端面试题汇总之HTML篇](#)
- 【2】 [「2021」高频前端面试题汇总之CSS篇](#)
- 【3】 [「2021」高频前端面试题汇总之JavaScript篇（上）](#)
- 【4】 [「2021」高频前端面试题汇总之JavaScript篇（下）](#)
- 【5】 [「2021」高频前端面试题汇总之Vue篇（上）](#)
- 【6】 [「2021」高频前端面试题汇总之Vue篇（下）](#)
- 【7】 [「2021」高频前端面试题汇总之React篇（上）](#)
- 【8】 [「2021」高频前端面试题汇总之React篇（下）](#)
- 【9】 [「2021」高频前端面试题汇总之计算机网络篇](#)
- 【10】 [「2021」高频前端面试题汇总之浏览器原理篇](#)
- 【11】 [「2021」高频前端面试题汇总之性能优化篇](#)
- 【12】 [「2021」高频前端面试题汇总之手写代码篇](#)
- 【13】 [「2021」高频前端面试题汇总之代码输出结果篇](#)

图例	
1	考察很多
2	考察较多
3	考察较少
4	考察很少



# 一、数据类型

## 1. JavaScript有哪些数据类型，它们的区别？

JavaScript共有八种数据类型，分别是 Undefined、Null、Boolean、Number、String、Object、Symbol、BigInt。

其中 Symbol 和 BigInt 是ES6 中新增的数据类型：

- Symbol 代表创建后独一无二且不可变的数据类型，它主要是为了解决可能出现的全局变量冲突的问题。
- BigInt 是一种数字类型的数据，它可以表示任意精度格式的整数，使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了 Number 能够表示的安全整数范围。

这些数据可以分为原始数据类型和引用数据类型：

- 栈：原始数据类型（Undefined、Null、Boolean、Number、String）
- 堆：引用数据类型（对象、数组和函数）

两种类型的区别在于**存储位置的不同**：

- 原始数据类型直接存储在栈（stack）中的简单数据段，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储；
- 引用数据类型存储在堆（heap）中的对象，占据空间大、大小不固定。如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。

堆和栈的概念存在于数据结构和操作系统内存中，在数据结构中：

- 在数据结构中，栈中数据的存取方式为先进后出。
- 堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。

在操作系统中，内存被分为栈区和堆区：

- 栈区内内存由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
- 堆区内存一般由开发者分配释放，若开发者不释放，程序结束时可能由垃圾回收机制回收。

## 2. 数据类型检测的方式有哪些

### (1) typeof

```
console.log(typeof 2);           // number
console.log(typeof true);        // boolean
console.log(typeof 'str');       // string
console.log(typeof []);          // object
console.log(typeof function(){}); // function
console.log(typeof {});          // object
console.log(typeof undefined);   // undefined
console.log(typeof null);        // object
```

复制代码

其中数组、对象、null都会被判断为object，其他判断都正确。

### (2) instanceof

`instanceof` 可以正确判断对象的类型，其内部运行机制是判断在其原型链中能否找到该类型的原型。

```
console.log(2 instanceof Number); // false
console.log(true instanceof Boolean); // false
console.log('str' instanceof String); // false

console.log([] instanceof Array); // true
console.log(function(){} instanceof Function); // true
console.log({} instanceof Object); // true
```

复制代码

可以看到，`instanceof` 只能正确判断引用数据类型，而不能判断基本数据类型。`instanceof` 运算符可以用来测试一个对象在其原型链中是否存在一个构造函数的 `prototype` 属性。

### (3) constructor

```
console.log((2).constructor === Number); // true
console.log((true).constructor === Boolean); // true
console.log(('str').constructor === String); // true
console.log([]).constructor === Array); // true
console.log((function() {})).constructor === Function); // true
console.log({}).constructor === Object); // true
```

复制代码

`constructor` 有两个作用，一是判断数据的类型，二是对象实例通过 `constructor` 对象访问它的构造函数。需要注意，如果创建一个对象来改变它的原型，`constructor` 就不能用来判断数据类型了：

```
function Fn(){};

Fn.prototype = new Array();

var f = new Fn();

console.log(f.constructor===Fn); // false
console.log(f.constructor===Array); // true
```

复制代码

#### (4) Object.prototype.toString.call()

`Object.prototype.toString.call()` 使用 `Object` 对象的原型方法 `toString` 来判断数据类型：

```
var a = Object.prototype.toString;

console.log(a.call(2));
console.log(a.call(true));
console.log(a.call('str'));
console.log(a.call([]));
console.log(a.call(function(){}));
console.log(a.call({}));
console.log(a.call(undefined));
console.log(a.call(null));
```

复制代码

同样是检测对象obj调用toString方法，obj.toString()的结果和Object.prototype.toString.call(obj)的结果不一样，这是为什么？

这是因为toString是Object的原型方法，而Array、function等类型作为Object的实例，都重写了**toString方法**。不同的对象类型调用toString方法时，根据原型链的知识，调用的是对应的重写之后的toString方法（function类型返回内容为函数体的字符串，Array类型返回元素组成的字符串...），而不会去调用Object上原型toString方法（返回对象的具体类型），所以采用obj.toString()不能得到其对象类型，只能将obj转换为字符串类型；因此，在想要得到对象的具体类型时，应该调用Object原型上的toString方法。

### 3. 判断数组的方式有哪些

- 通过Object.prototype.toString.call()做判断

```
Object.prototype.toString.call(obj).slice(8,-1) === 'Array';
```

复制代码

- 通过原型链做判断

```
obj.__proto__ === Array.prototype;
```

复制代码

- 通过ES6的Array.isArray()做判断

```
Array.isArray(obj);
```

复制代码

- 通过instanceof做判断

```
obj instanceof Array
```

复制代码

- 通过Array.prototype.isPrototypeOf

```
Array.prototype.isPrototypeOf(obj)
```

复制代码

## 4. null和undefined区别

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。

undefined 代表的含义是**未定义**，null 代表的含义是**空对象**。一般变量声明了但还没有定义的时候会返回 undefined，null主要用于赋值给一些可能会返回对象的变量，作为初始化。

undefined 在 JavaScript 中不是一个保留字，这意味着可以使用 undefined 来作为一个变量名，但是这样的做法是非常危险的，它会影响对 undefined 值的判断。我们可以通过一些方法获得安全的 undefined 值，比如说 void 0。

当对这两种类型使用 typeof 进行判断时，Null 类型化会返回 "object"，这是一个历史遗留的问题。当使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

## 5. typeof null 的结果是什么，为什么？

typeof null 的结果是Object。

在 JavaScript 第一个版本中，所有值都存储在 32 位的单元中，每个单元包含一个小的 **类型标签(1-3 bits)** 以及当前要存储值的真实数据。类型标签存储在每个单元的低位中，共有五种数据类型：

000: object	- 当前存储的数据指向一个对象。
1: int	- 当前存储的数据是一个 31 位的有符号整数。
010: double	- 当前存储的数据指向一个双精度的浮点数。
100: string	- 当前存储的数据指向一个字符串。
110: boolean	- 当前存储的数据是布尔值。

复制代码

如果最低位是 1，则类型标签标志位的长度只有一位；如果最低位是 0，则类型标签标志位的长度占三位，为存储其他四种数据类型提供了额外两个 bit 的长度。

有两种特殊数据类型：

- undefined的值是 (-2)<sup>30</sup>(一个超出整数范围的数字)；

- null 的值是机器码 NULL 指针(null 指针的值全是 0)

那也就是说null的类型标签也是000，和Object的类型标签一样，所以会被判定为Object。

## 6. instanceof 操作符的实现原理及实现

instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

```
function myInstanceOf(left, right) {  
  // 获取对象的原型  
  let proto = Object.getPrototypeOf(left)  
  // 获取构造函数的 prototype 对象  
  let prototype = right.prototype;  
  
  // 判断构造函数的 prototype 对象是否在对象的原型链上  
  while (true) {  
    if (!proto) return false;  
    if (proto === prototype) return true;  
    // 如果没有找到，就继续从其原型上找，Object.getPrototypeOf方法用来获取指定对象的原型  
    proto = Object.getPrototypeOf(proto);  
  }  
}
```

复制代码

## 7. 为什么0.1+0.2 !== 0.3，如何让其相等

在开发过程中遇到类似这样的问题：

```
let n1 = 0.1, n2 = 0.2  
console.log(n1 + n2) // 0.30000000000000004  
复制代码
```

这里得到的不是想要的结果，要想等于0.3，就要把它进行转化：

```
(n1 + n2).toFixed(2) // 注意，toFixed为四舍五入  
复制代码
```

toFixed(num) 方法可把 Number 四舍五入为指定小数位数的数字。那为什么会出现这样的结果呢？

计算机是通过二进制的方式存储数据的，所以计算机计算0.1+0.2的时候，实际上是计算的两个数的二进制的和。0.1的二进制是 0.0001100110011001100...（1100循环），0.2的二进制是：

0.00110011001100...（1100循环），这两个数的二进制都是无限循环的数。那JavaScript是如何处理无限循环的二进制小数呢？

一般我们认为数字包括整数和小数，但是在 JavaScript 中只有一种数字类型：Number，它的实现遵循 IEEE 754标准，使用64位固定长度来表示，也就是标准的double双精度浮点数。在二进制科学表示法中，双精度浮点数的小数部分最多只能保留52位，再加上前面的1，其实就是保留53位有效数字，剩余的需要舍去，遵从“0舍1入”的原则。

根据这个原则，0.1和0.2的二进制数相加，再转化为十进制数就是：0.30000000000000004。

下面看一下**双精度数是如何保存的**：



- 第一部分（蓝色）：用来存储符号位（sign），用来区分正负数，0表示正数，占用1位
- 第二部分（绿色）：用来存储指数（exponent），占用11位
- 第三部分（红色）：用来存储小数（fraction），占用52位

对于0.1，它的二进制为：

```
0.000110011001100110011001100110011001100110011001100110011001 10011...
```

复制代码

转为科学计数法（科学计数法的结果就是浮点数）：

```
1.100110011001100110011001100110011001100110011001100110011001*2^-4
```

复制代码

可以看出0.1的符号位为0，指数位为-4，小数位为：

```
100110011001100110011001100110011001100110011001100110011001
```

复制代码

那么问题又来了，**指数位是负数，该如何保存呢？**

IEEE标准规定了一个偏移量，对于指数部分，每次都加这个偏移量进行保存，这样即使指数是负数，那么加上这个偏移量也就是正数了。由于JavaScript的数字是双精度数，这里就以双精度数为例，它的指数部分为11位，能表示的范围就是0~2047，IEEE固定**双精度数的偏移量为1023**。

- 当指数位不全是0也不全是1时(规格化的数值)，IEEE规定，阶码计算公式为  $e - \text{Bias}$ 。此时e最小值是1，则  $1 - 1023 = -1022$ ，e最大值是2046，则  $2046 - 1023 = 1023$ ，可以看到，这种情况下取值范围是  $-1022 \sim 1013$ 。
- 当指数位全部是0的时候(非规格化的数值)，IEEE规定，阶码的计算公式为  $1 - \text{Bias}$ ，即  $1 - 1023 = -1022$ 。
- 当指数位全部是1的时候(特殊值)，IEEE规定这个浮点数可用来表示3个特殊值，分别是正无穷，负无穷，NaN。具体的，小数位不为0的时候表示NaN；小数位为0时，当符号位s=0时表示正无穷，s=1时候表示负无穷。

对于上面的0.1的指数位为-4， $-4 + 1023 = 1019$  转化为二进制就是：1111111011。

所以，0.1表示为：

```
0 1111111011 100110011001100110011001100110011001100110011001100110011001
```

复制代码

说了这么多，是时候该最开始的问题了，如何实现  $0.1 + 0.2 = 0.3$  呢？

对于这个问题，一个直接的解决方法就是设置一个误差范围，通常称为“机器精度”。对JavaScript来说，这个值通常为  $2^{-52}$ ，在ES6中，提供了 `Number.EPSILON` 属性，而它的值就是  $2^{-52}$ ，只要判断  $0.1 + 0.2 - 0.3$  是否小于 `Number.EPSILON`，如果小于，就可以判断为  $0.1 + 0.2 === 0.3$



```
function numberepsilon(arg1,arg2){
  return Math.abs(arg1 - arg2) < Number.EPSILON;
}

console.log(numberepsilon(0.1 + 0.2, 0.3)); // true
```

复制代码

## 8. 如何获取安全的 undefined 值？

因为 undefined 是一个标识符，所以可以被当作变量来使用和赋值，但是这样会影响 undefined 的正常判断。表达式 void \_\_ 没有返回值，因此返回结果是 undefined。void 并不改变表达式的结果，只是让表达式不返回值。因此可以用 void 0 来获得 undefined。

## 9. typeof NaN 的结果是什么？

NaN 指“不是一个数字”（not a number），NaN 是一个“警戒值”（sentinel value，有特殊用途的常规值），用于指出数字类型中的错误情况，即“执行数学运算没有成功，这是失败后返回的结果”。

```
typeof NaN; // "number"
```

复制代码

NaN 是一个特殊值，它和自身不相等，是唯一一个非自反（自反，reflexive，即  $x === x$  不成立）的值。而  $NaN !== NaN$  为 true。

## 10. isNaN 和 Number.isNaN 函数的区别？

- 函数 isNaN 接收参数后，会尝试将这个参数转换为数值，任何不能被转换为数值的值都会返回 true，因此非数字值传入也会返回 true，会影响 NaN 的判断。
- 函数 Number.isNaN 会首先判断传入参数是否为数字，如果是数字再继续判断是否为 NaN，不会进行数据类型的转换，这种方法对于 NaN 的判断更为准确。

## 11. == 操作符的强制类型转换规则？

对于 == 来说，如果对比双方的类型**不一样**，就会进行**类型转换**。假如对比 x 和 y 是否相同，就会进行如下判断流程：

1. 首先会判断两者类型是否**相同**，相同的话就比较两者的大小；
2. 类型不相同的话，就会进行类型转换；
3. 会先判断是否在对 `null` 和 `undefined`，是的话就会返回 `true`
4. 判断两者类型是否为 `string` 和 `number`，是的话就会将字符串转换为 `number`

```
1 == '1'
  ↓
1 == 1
```

复制代码

1. 判断其中一方是否为 `boolean`，是的话就会把 `boolean` 转为 `number` 再进行判断



```
'1' == true
  ↓
'1' == 1
  ↓
1 == 1
```

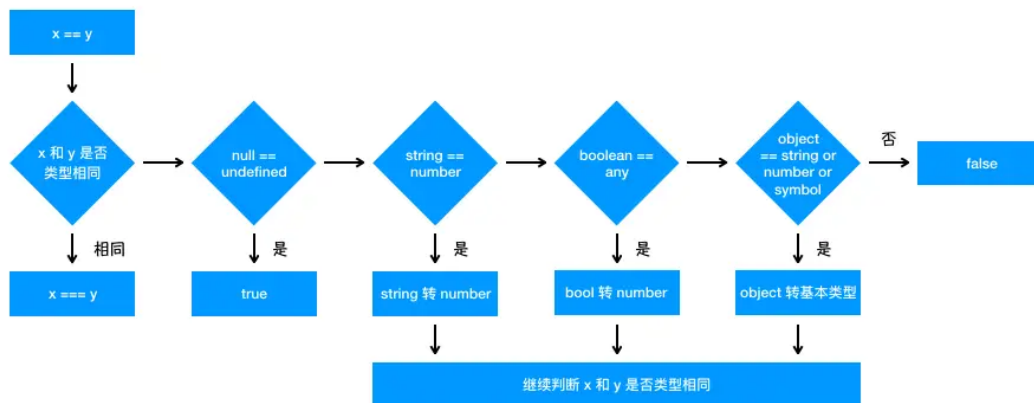
复制代码

1. 判断其中一方是否为 `object` 且另一方为 `string`、`number` 或者 `symbol`，是的话就会把 `object` 转为原始类型再进行判断

```
'1' == { name: 'js' }      ↓ '1' == '[object Object]'
```

复制代码

其流程图如下：



@稀土掘金技术社区

## 12. 其他值到字符串的转换规则？

- Null 和 Undefined 类型，null 转换为 "null"，undefined 转换为 "undefined"，
- Boolean 类型，true 转换为 "true"，false 转换为 "false"。
- Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。
- Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。
- 对普通对象来说，除非自行定义 toString() 方法，否则会调用 toString() (Object.prototype.toString()) 来返回内部属性 [[Class]] 的值，如 "[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值。

## 13. 其他值到数字值的转换规则？

- Undefined 类型的值转换为 NaN。
- Null 类型的值转换为 0。
- Boolean 类型的值，true 转换为 1，false 转换为 0。
- String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。
- Symbol 类型的值不能转换为数字，会报错。
- 对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。

为了将值转换为相应的基本类型值，抽象操作 ToPrimitive 会首先（通过内部操作 DefaultValue）检查该值是否有 valueOf() 方法。如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。

如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

## 14. 其他值到布尔类型的值的转换规则？

以下这些是假值：• undefined • null • false • +0、-0 和 NaN • ""

假值的布尔强制类型转换结果为 false。从逻辑上说，假值列表以外的都应该是真值。

## 15. || 和 && 操作符的返回值？

|| 和 && 首先会对第一个操作数执行条件判断，如果其不是布尔值就先强制转换为布尔类型，然后再执行条件判断。

- 对于 || 来说，如果条件判断结果为 true 就返回第一个操作数的值，如果为 false 就返回第二个操作数的值。
- && 则相反，如果条件判断结果为 true 就返回第二个操作数的值，如果为 false 就返回第一个操作数的值。

|| 和 && 返回它们其中一个操作数的值，而非条件判断的结果

## 16. Object.is() 与比较操作符 “===”、“==” 的区别？

- 使用双等号 (==) 进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。
- 使用三等号 (===) 进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。
- 使用 Object.is 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。

## 17. 什么是 JavaScript 中的包装类型？

在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作基本类型的值，在调用基本类型的属性或方法时 JavaScript 会在后台隐式地将基本类型的值转换为对象，如：

```
const a = "abc";
a.length; // 3
a.toUpperCase(); // "ABC"
```

复制代码

在访问 'abc'.length 时，JavaScript 将 'abc' 在后台转换成 String('abc')，然后再访问其 length 属性。

JavaScript 也可以使用 Object 函数显式地将基本类型转换为包装类型：

```
var a = 'abc'
Object(a) // String {"abc"}
```

复制代码

也可以使用 valueOf 方法将包装类型倒转成基本类型：

```
var a = 'abc'
var b = Object(a)
var c = b.valueOf() // 'abc'
```

复制代码

看看如下代码会打印出什么：

```
var a = new Boolean( false );
if (!a) {
    console.log( "oops" ); // never runs
}
```

复制代码

答案是什么都不会打印，因为虽然包裹的基本类型是 `false`，但是 `false` 被包裹成包装类型后就成了对象，所以其非值为 `false`，所以循环体中的内容不会运行。

## 18. JavaScript 中如何进行隐式类型转换？

首先要介绍 `ToPrimitive` 方法，这是 JavaScript 中每个值隐含的自带的方法，用来将值（无论是基本类型值还是对象）转换为基本类型值。如果值为基本类型，则直接返回值本身；如果值为对象，其看起来大概是这样：

```
/**
 * @obj 需要转换的对象
 * @type 期望的结果类型
 */
ToPrimitive(obj, type)
```

复制代码

`type` 的值为 `number` 或者 `string`。

### (1) 当 `type` 为 `number` 时规则如下：

- 调用 `obj` 的 `valueOf` 方法，如果为原始值，则返回，否则下一步；
- 调用 `obj` 的 `toString` 方法，后续同上；
- 抛出 `TypeError` 异常。

### (2) 当 `type` 为 `string` 时规则如下：

- 调用 `obj` 的 `toString` 方法，如果为原始值，则返回，否则下一步；
- 调用 `obj` 的 `valueOf` 方法，后续同上；
- 抛出 `TypeError` 异常。

可以看出两者的主要区别在于调用 `toString` 和 `valueOf` 的先后顺序。默认情况下：

- 如果对象为 `Date` 对象，则 `type` 默认为 `string`；
- 其他情况下，`type` 默认为 `number`。

总结上面的规则，对于 `Date` 以外的对象，转换为基本类型的大概规则可以概括为一个函数：

```
var objToNumber = value => Number(value.valueOf().toString())
objToNumber([]) === 0
objToNumber({}) === NaN
```

复制代码

而 JavaScript 中的隐式类型转换主要发生在 `+`、`-`、`*`、`/` 以及 `==`、`>`、`<` 这些运算符之间。而这些运算符只能操作基本类型值，所以在进行这些运算前的第一步就是将两边的值用 `ToPrimitive` 转换成基本类型，再进行操作。

以下是基本类型的值在不同操作符的情况下隐式转换的规则（对于对象，其会被 `ToPrimitive` 转换成基本类型，所以最终还是要应用基本类型转换规则）：

### 1. `+` 操作符

+ 操作符的两边有至少一个 `string` 类型变量时，两边的变量都会被隐式转换为字符串；其他情况下两边的变量都会被转换为数字。

```
1 + '23' // '123'
1 + false // 1
1 + Symbol() // Uncaught TypeError: Cannot convert a symbol value to a number
'1' + false // '1false'
false + true // 1
```

复制代码

### 1. `-`、`*`、`\` 操作符

`NaN` 也是一个数字

```
1 * '23' // 23
1 * false // 0
1 / 'aa' // NaN
```

复制代码

### 1. 对于 `==` 操作符

操作符两边的值都尽量转成 `number`：

```
3 == true // false, 3 转为number为3, true转为number为1
'0' == false //true, '0'转为number为0, false转为number为0
'0' == 0 // '0'转为number为0
```

复制代码

### 1. 对于 `<` 和 `>` 比较符

如果两边都是字符串，则比较字母表顺序：

```
'ca' < 'bd' // false
'a' < 'b' // true
```

复制代码

其他情况下，转换为数字再比较：

```
'12' < 13 // true
false > -1 // true
```

复制代码

以上说的是基本类型的隐式转换，而对象会被 `ToPrimitive` 转换为基本类型再进行转换：

```
var a = {}
a > 2 // false
```

复制代码

其对比过程如下：

```
a.valueOf() // {}, 上面提到过, ToPrimitive默认type为number, 所以先valueOf, 结果还是个对象, 下一步
a.toString() // "[object Object]", 现在是一个字符串了
Number(a.toString()) // NaN, 根据上面 < 和 > 操作符的规则, 要转换成数字
NaN > 2 //false, 得出比较结果
复制代码
```

又比如:

```
var a = {name: 'Jack'}
var b = {age: 18}
a + b // "[object Object][object Object]"
复制代码
```

运算过程如下:

```
a.valueOf() // {}, 上面提到过, ToPrimitive默认type为number, 所以先valueOf, 结果还是个对象, 下一步
a.toString() // "[object Object]"
b.valueOf() // 同理
b.toString() // "[object Object]"
a + b // "[object Object][object Object]"
复制代码
```

## 19. + 操作符什么时候用于字符串的拼接?

根据 ES5 规范, 如果某个操作数是字符串或者能够通过以下步骤转换为字符串的话, + 将进行拼接操作。如果其中一个操作数是对象(包括数组), 则首先对其调用 ToPrimitive 抽象操作, 该抽象操作再调用 [[DefaultValue]], 以数字作为上下文。如果不能转换为字符串, 则会将其转换为数字类型来进行计算。

简单来说就是, 如果 + 的其中一个操作数是字符串(或者通过以上步骤最终得到字符串), 则执行字符串拼接, 否则执行数字加法。

那么对于除了加法的运算符来说, 只要其中一方是数字, 那么另一方就会被转为数字。

## 20. 为什么会有BigInt的提案?

JavaScript中Number.MAX\_SAFE\_INTEGER表示最大安全数字, 计算结果是9007199254740991, 即在这个数范围内不会出现精度丢失(小数除外)。但是一旦超过这个范围, js就会出现计算不准确的情况, 这在大数计算的时候不得不依靠一些第三方库进行解决, 因此官方提出了BigInt来解决此问题。

## 21. object.assign和扩展运算是深拷贝还是浅拷贝, 两者区别

扩展运算符:

```
let outObj = {
  inObj: {a: 1, b: 2}
}
let newObj = {...outObj}
newObj.inObj.a = 2
console.log(outObj) // {inObj: {a: 2, b: 2}}
复制代码
```

Object.assign():

```
let outObj = {
  inObj: {a: 1, b: 2}
}
let newObj = Object.assign({}, outObj)
newObj.inObj.a = 2
console.log(outObj) // {inObj: {a: 2, b: 2}}
```

复制代码

可以看到，两者都是浅拷贝。

- Object.assign()方法接收的第一个参数作为目标对象，后面的所有参数作为源对象。然后把所有的源对象合并到目标对象中。它会修改了一个对象，因此会触发 ES6 setter。
- 扩展操作符 (...) 使用它时，数组或对象中的每一个值都会被拷贝到一个新的数组或对象中。它不复制继承的属性或类的属性，但是它会复制ES6的 symbols 属性。

## 二、ES6

### 1. let、const、var的区别

**(1) 块级作用域：**块作用域由 `{ }` 包括，let和const具有块级作用域，var不存在块级作用域。块级作用域解决了ES5中的两个问题：

- 内层变量可能覆盖外层变量
- 用来计数的循环变量泄露为全局变量

**(2) 变量提升：**var存在变量提升，let和const不存在变量提升，即在变量只能在声明之后使用，否在会报错。

**(3) 给全局添加属性：**浏览器的全局对象是window，Node的全局对象是global。var声明的变量为全局变量，并且会将该变量添加为全局对象的属性，但是let和const不会。

**(4) 重复声明：**var声明变量时，可以重复声明变量，后声明的同名变量会覆盖之前声明的遍历。const和let不允许重复声明变量。

**(5) 暂时性死区：**在使用let、const命令声明变量之前，该变量都是不可用的。这在语法上，称为**暂时性死区**。使用var声明的变量不存在暂时性死区。

**(6) 初始值设置：**在变量声明时，var 和 let 可以不用设置初始值。而const声明变量必须设置初始值。

**(7) 指针指向：**let和const都是ES6新增的用于创建变量的语法。let创建的变量是可以更改指针指向（可以重新赋值）。但const声明的变量是不允许改变指针的指向。

区别	var	let	const
是否有块级作用域	×	✓	✓
是否存在变量提升	✓	×	×
是否添加全局属性	✓	×	×
能否重复声明变量	✓	×	×
是否存在暂时性死区	×	✓	✓
是否必须设置初始值	×	×	✓
能否改变指针指向	✓	✓	×

## 2. const对象的属性可以修改吗

const保证的并不是变量的值不能改动，而是变量指向的那个内存地址不能改动。对于基本类型的数据（数值、字符串、布尔值），其值就保存在变量指向的那个内存地址，因此等同于常量。

但对于引用类型的数据（主要是对象和数组）来说，变量指向数据的内存地址，保存的只是一个指针，const只能保证这个指针是固定不变的，至于它指向的数据结构是不是可变的，就完全不能控制了。

## 3. 如果new一个箭头函数的会怎么样

箭头函数是ES6中的提出来的，它没有prototype，也没有自己的this指向，更不可以使用arguments参数，所以不能New一个箭头函数。

new操作符的实现步骤如下：

1. 创建一个对象
2. 将构造函数的作用域赋给新对象（也就是将对象的proto属性指向构造函数的prototype属性）
3. 指向构造函数中的代码，构造函数中的this指向该对象（也就是为这个对象添加属性和方法）
4. 返回新的对象

所以，上面的第二、三步，箭头函数都是没有办法执行的。

## 4. 箭头函数与普通函数的区别

### (1) 箭头函数比普通函数更加简洁

- 如果没有参数，就直接写一个空括号即可
- 如果只有一个参数，可以省去参数的括号
- 如果有多个参数，用逗号分割
- 如果函数体的返回值只有一句，可以省略大括号
- 如果函数体不需要返回值，且只有一句话，可以给这个语句前面加一个void关键字。最常见的就是调用一个函数：

```
let fn = () => void doesNotReturn();
```

复制代码

### (2) 箭头函数没有自己的this

箭头函数不会创建自己的this，所以它没有自己的this，它只会在自己作用域的上一层继承this。所以箭头函数中this的指向在它在定义时已经确定了，之后不会改变。

### (3) 箭头函数继承来的this指向永远不会改变



```

var id = 'GLOBAL';
var obj = {
  id: 'OBJ',
  a: function(){
    console.log(this.id);
  },
  b: () => {
    console.log(this.id);
  }
};
obj.a();    // 'OBJ'
obj.b();    // 'GLOBAL'
new obj.a() // undefined
new obj.b() // Uncaught TypeError: obj.b is not a constructor
复制代码

```

对象obj的方法b是使用箭头函数定义的，这个函数中的this就永远指向它定义时所处的全局执行环境中的this，即便这个函数是作为对象obj的方法调用，this依旧指向Window对象。需要注意，定义对象的大括号 {} 是无法形成一个单独的执行环境的，它依旧是处于全局执行环境中。

#### (4) call()、apply()、bind()等方法不能改变箭头函数中this的指向

```

var id = 'Global';
let fun1 = () => {
  console.log(this.id)
};
fun1();                // 'Global'
fun1.call({id: 'Obj'}); // 'Global'
fun1.apply({id: 'Obj'}); // 'Global'
fun1.bind({id: 'Obj'})(); // 'Global'
复制代码

```

#### (5) 箭头函数不能作为构造函数使用

构造函数在new的步骤在上面已经说过了，实际上第二步就是将函数中的this指向该对象。但是由于箭头函数时没有自己的this的，且this指向外层的执行环境，且不能改变指向，所以不能当做构造函数使用。

#### (6) 箭头函数没有自己的arguments

箭头函数没有自己的arguments对象。在箭头函数中访问arguments实际上获得的是它外层函数的arguments值。

#### (7) 箭头函数没有prototype

#### (8) 箭头函数不能用作Generator函数，不能使用yeild关键字

## 5. 箭头函数的this指向哪里？

箭头函数不同于传统JavaScript中的函数，箭头函数并没有属于自己的this，它所谓的this是捕获其所在上下文的 this 值，作为自己的 this 值，并且由于没有属于自己的this，所以是不会被new调用的，这个所谓的this也不会被改变。

可以用Babel理解一下箭头函数：

```
// ES6
const obj = {
  getArrow() {
    return () => {
      console.log(this === obj);
    };
  }
}
```

复制代码

转化后：

```
// ES5, 由 Babel 转译
var obj = {
  getArrow: function getArrow() {
    var _this = this;
    return function () {
      console.log(_this === obj);
    };
  }
};
```

复制代码

## 6. 扩展运算符的作用及使用场景

### (1) 对象扩展运算符

对象的扩展运算符(...)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中。

```
let bar = { a: 1, b: 2 };
let baz = { ...bar }; // { a: 1, b: 2 }
```

复制代码

上述方法实际上等价于：

```
let bar = { a: 1, b: 2 };
let baz = Object.assign({}, bar); // { a: 1, b: 2 }
```

复制代码

`Object.assign` 方法用于对象的合并，将源对象（`source`）的所有可枚举属性，复制到目标对象（`target`）。`Object.assign` 方法的第一个参数是目标对象，后面的参数都是源对象。（如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性）。

同样，如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

```
let bar = {a: 1, b: 2};
let baz = {...bar, ...{a:2, b: 4}}; // {a: 2, b: 4}
```

复制代码

利用上述特性就可以很方便的修改对象的部分属性。在 `redux` 中的 `reducer` 函数规定必须是一个纯函数，`reducer` 中的 `state` 对象要求不能直接修改，可以通过扩展运算符把修改路径的对象都复制一遍，然后产生一个新的对象返回。

需要注意：扩展运算符对对象实例的拷贝属于浅拷贝。

## (2) 数组扩展运算符

数组的扩展运算符可以将一个数组转为用逗号分隔的参数序列，且每次只能展开一层数组。

```
console.log(...[1, 2, 3])  
// 1 2 3  
console.log(...[1, [2, 3, 4], 5])  
// 1 [2, 3, 4] 5  
复制代码
```

下面是数组的扩展运算符的应用：

- 将数组转换为参数序列

```
function add(x, y) {  
  return x + y;  
}  
const numbers = [1, 2];  
add(...numbers) // 3  
复制代码
```

- 复制数组

```
const arr1 = [1, 2];  
const arr2 = [...arr1];  
复制代码
```

要记住：**扩展运算符(...)**用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中，这里参数对象是个数组，数组里面的所有对象都是基础数据类型，将所有基础数据类型重新拷贝到新的数组中。

- 合并数组

如果想在数组内合并数组，可以这样：

```
const arr1 = ['two', 'three'];const arr2 = ['one', ...arr1, 'four', 'five'];  
// ["one", "two", "three", "four", "five"]  
复制代码
```

- 扩展运算符与解构赋值结合起来，用于生成数组

```
const [first, ...rest] = [1, 2, 3, 4, 5];first // 1rest // [2, 3, 4, 5]  
复制代码
```

需要注意：如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

```
const [...rest, last] = [1, 2, 3, 4, 5]; // 报错const [first, ...rest,  
last] = [1, 2, 3, 4, 5]; // 报错  
复制代码
```

- 将字符串转为真正的数组

```
[...'hello'] // [ "h", "e", "l", "l", "o" ]  
复制代码
```

- 任何 `Iterator` 接口的对象，都可以用扩展运算符转为真正的数组

比较常见的应用是可以将某些数据结构转为数组：

```
// arguments对象
function foo() {
  const args = [...arguments];
}
```

复制代码

用于替换 es5 中的 `Array.prototype.slice.call(arguments)` 写法。

- 使用 `Math` 函数获取数组中特定的值

```
const numbers = [9, 4, 7, 1];
Math.min(...numbers); // 1
Math.max(...numbers); // 9
```

复制代码

## 7. Proxy 可以实现什么功能？

在 Vue3.0 中通过 `Proxy` 来替换原本的 `Object.defineProperty` 来实现数据响应式。

`Proxy` 是 ES6 中新增的功能，它可以用来自定义对象中的操作。

```
let p = new Proxy(target, handler)
```

复制代码

`target` 代表需要添加代理的对象，`handler` 用来自定义对象中的操作，比如可以用来自定义 `set` 或者 `get` 函数。

下面来通过 `Proxy` 来实现一个数据响应式：

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver)
    },
    set(target, property, value, receiver) {
      setBind(value, property)
      return Reflect.set(target, property, value)
    }
  }
  return new Proxy(obj, handler)
}

let obj = { a: 1 }
let p = onWatch(
  obj,
  (v, property) => {
    console.log(`监听到属性${property}改变为${v}`)
  },
  (target, property) => {
    console.log(`${property} = ${target[property]}`)
  }
)
```

```
p.a = 2 // 监听到属性a改变
p.a // 'a' = 2
复制代码
```

在上述代码中，通过自定义 `set` 和 `get` 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了对对象任何属性进行读写时发出通知。

当然这是简单版的响应式实现，如果需要一个 Vue 中的响应式，需要在 `get` 中收集依赖，在 `set` 派发更新，之所以 Vue3.0 要使用 `Proxy` 替换原本的 API 原因在于 `Proxy` 无需一层层递归为每个属性添加代理，一次即可完成以上操作，性能上更好，并且原本的实现有一些数据更新不能监听到，但是 `Proxy` 可以完美监听到任何方式的数据改变，唯一缺陷就是浏览器的兼容性不好。

## 8. 对对象与数组的解构的理解

解构是 ES6 提供的一种新的提取数据的模式，这种模式能够从对象或数组里有针对性地拿到想要的数值。1) **数组的解构** 在解构数组时，以元素的位置为匹配条件来提取想要的数据的：

```
const [a, b, c] = [1, 2, 3]
复制代码
```

最终，a、b、c 分别被赋予了数组第 0、1、2 个索引位的值：

> a

< 1

> b

< 2

> c

< 3

数组里的 0、1、2 索引

@稀土掘金技术社区

位的元素值，精准地被映射到了左侧的第 0、1、2 个变量里去，这就是数组解构的工作模式。还可以通过给左侧变量数组设置空占位的方式，实现对数组中某几个元素的精准提取：

```
const [a,,c] = [1,2,3]
```

复制代码

通过把中间位留空，可以顺利地把数组第一位和最后一位的值赋给 a、c 两个变量：

> a

< 1

> c

< 3

@掘金技术社区

**2) 对象的解构** 对象解构比数组结构稍微复杂一些，也更显强大。在解构对象时，是以属性的名称为匹配条件，来提取想要的数据的。现在定义一个对象：

```
const stu = {  
  name: 'Bob',  
  age: 24  
}
```

复制代码

假如想要解构它的两个自有属性，可以这样：

```
const { name, age } = stu
```

复制代码

这样就得到了 name 和 age 两个和 stu 平级的变量：

> name

< "Bob"

> age

< 24

@稀土掘金技术社区

注意，对象解构严格以属性名作为定位依据，所以就算调换了 name 和 age 的位置，结果也是一样的：

```
const { age, name } = stu
```

复制代码

## 9. 如何提取高度嵌套的对象里的指定属性？

有时会遇到一些嵌套程度非常深的对象：

```
const school = {  
  classes: {  
    stu: {  
      name: 'Bob',  
      age: 24,  
    }  
  }  
}
```

复制代码

像此处的 name 这个变量，嵌套了四层，此时如果仍然尝试老方法来提取它：

```
const { name } = school
```

复制代码

显然是不奏效的，因为 school 这个对象本身是没有 name 这个属性的，name 位于 school 对象的“儿子的儿子”对象里面。要想把 name 提取出来，一种比较笨的方法是逐层解构：



```
const { classes } = school
const { stu } = classes
const { name } = stu
name // 'Bob'
```

复制代码

但是还有一种更标准的做法，可以用一行代码来解决这个问题：

```
const { classes: { stu: { name } } } = school

console.log(name) // 'Bob'
```

复制代码

可以在解构出来的变量名右侧，通过冒号+{目标属性名}这种形式，进一步解构它，一直解构到拿到目标数据为止。

## 10. 对 rest 参数的理解

扩展运算符被用在函数形参上时，它还可以把一个分离的参数序列整合成一个数组：

```
function mutiple(...args) {
  let result = 1;
  for (var val of args) {
    result *= val;
  }
  return result;
}

mutiple(1, 2, 3, 4) // 24
```

复制代码

这里，传入 mutiple 的是四个分离的参数，但是如果在 mutiple 函数里尝试输出 args 的值，会发现它是一个数组：

```
function mutiple(...args) {
  console.log(args)
}

mutiple(1, 2, 3, 4) // [1, 2, 3, 4]
```

复制代码

这就是 ... rest运算符的又一层威力了，它可以把函数的多个入参收敛进一个数组里。这一点经常用于获取函数的多余参数，或者像上面这样处理函数参数个数不确定的情况。

## 11. ES6中模板语法与字符串处理

ES6 提出了“模板语法”的概念。在 ES6 以前，拼接字符串是很麻烦的事情：

```
var name = 'css'
var career = 'coder'
var hobby = ['coding', 'writing']
var finalString = 'my name is ' + name + ', I work as a ' + career + ', I love '
+ hobby[0] + ' and ' + hobby[1]
```

复制代码

仅仅几个变量，写了这么多加号，还要时刻小心里面的空格和标点符号有没有跟错地方。但是有了模板字符串，拼接难度直线下降：

```
var name = 'css'
var career = 'coder'
var hobby = ['coding', 'writing']
var finalString = `my name is ${name}, I work as a ${career} I love ${hobby[0]}
and ${hobby[1]}`
```

复制代码

字符串不仅更容易拼了，也更易读了，代码整体的质量都变高了。这就是模板字符串的第一个优势——允许用`\${}`的方式嵌入变量。但这还不是问题的关键，模板字符串的关键优势有两个：

- 在模板字符串中，空格、缩进、换行都会被保留
- 模板字符串完全支持“运算”式的表达式，可以在`\${}`里完成一些计算

基于第一点，可以在模板字符串里无障碍地直接写 html 代码：

```
let list = `
  <ul>
    <li>列表项1</li>
    <li>列表项2</li>
  </ul>
`;
console.log(message); // 正确输出，不存在报错
```

复制代码

基于第二点，可以把一些简单的计算和调用丢进`\${}`来做：

```
function add(a, b) {
  const finalString = `${a} + ${b} = ${a+b}`
  console.log(finalString)
}
add(1, 2) // 输出 '1 + 2 = 3'
```

复制代码

除了模板语法外，ES6中还新增了一系列的字符串方法用于提升开发效率：

(1) **存在性判定**：在过去，当判断一个字符/字符串是否在某字符串中时，只能用 `indexOf > -1` 来做。现在 ES6 提供了三个方法：`includes`、`startsWith`、`endsWith`，它们都会返回一个布尔值来告诉你是否存在。

- **includes**：判断字符串与子串的包含关系：

```
const son = 'haha'
const father = 'xixi haha hehe'
father.includes(son) // true
```

复制代码

- **startsWith**：判断字符串是否以某个/某串字符开头：

```
const father = 'xixi haha hehe'
father.startsWith('haha') // false
father.startsWith('xixi') // true
```

复制代码

- **endsWith**: 判断字符串是否以某个/某串字符结尾:

```
const father = 'xixi haha hehe'
father.endsWith('hehe') // true
```

复制代码

- (2) **自动重复**: 可以使用 repeat 方法来使同一个字符串输出多次 (被连续复制多次):

```
const sourceCode = 'repeat for 3 times;'
const repeated = sourceCode.repeat(3)
console.log(repeated) // repeat for 3 times;repeat for 3 times;repeat for 3
times;
```

复制代码

## 三、JavaScript基础

### 1. new操作符的实现原理

**new操作符的执行过程:**

- (1) 首先创建了一个新的空对象
- (2) 设置原型, 将对象的原型设置为函数的 prototype 对象。
- (3) 让函数的 this 指向这个对象, 执行构造函数的代码 (为这个新对象添加属性)
- (4) 判断函数的返回值类型, 如果是值类型, 返回创建的对象。如果是引用类型, 就返回这个引用类型的对象。

具体实现:

```
function objectFactory() {
  let newObject = null;
  let constructor = Array.prototype.shift.call(arguments);
  let result = null;
  // 判断参数是否是一个函数
  if (typeof constructor !== "function") {
    console.error("type error");
    return;
  }
  // 新建一个空对象, 对象的原型为构造函数的 prototype 对象
  newObject = Object.create(constructor.prototype);
  // 将 this 指向新建对象, 并执行函数
  result = constructor.apply(newObject, arguments);
  // 判断返回对象
  let flag = result && (typeof result === "object" || typeof result ===
"function");
  // 判断返回结果
  return flag ? result : newObject;
}
// 使用方法
objectFactory(构造函数, 初始化参数);
```

复制代码

## 2. map和Object的区别

	Map	Object
意外的键	Map默认情况不包含任何键，只包含显式插入的键。	Object 有一个原型, 原型链上的键名有可能和自己在对象上的设置的键名产生冲突。
键的类型	Map的键可以是任意值，包括函数、对象或任意基本类型。	Object 的键必须是 String 或是Symbol。
键的顺序	Map 中的 key 是有序的。因此，当迭代的时候， Map 对象以插入的顺序返回键值。	Object 的键是无序的
Size	Map 的键值对个数可以轻易地通过size 属性获取	Object 的键值对个数只能手动计算
迭代	Map 是 iterable 的，所以可以直接被迭代。	迭代Object需要以某种方式获取它的键然后才能迭代。
性能	在频繁增删键值对的场景下表现更好。	在频繁添加和删除键值对的场景下未作出优化。

## 3. map和weakMap的区别

**(1) Map** map本质上就是键值对的集合，但是普通的Object中的键值对中的键只能是字符串。而ES6提供的Map数据结构类似于对象，但是它的键不限制范围，可以是任意类型，是一种更加完善的Hash结构。如果Map的键是一个原始数据类型，只要两个键严格相同，就视为是同一个键。

实际上Map是一个数组，它的每一个数据也都是一个数组，其形式如下：

```
const map = [
  ["name", "张三"],
  ["age", 18],
]
```

复制代码

Map数据结构有以下操作方法：

- **size**： `map.size` 返回Map结构的成员总数。
- **set(key,value)**： 设置键名key对应的键值value，然后返回整个Map结构，如果key已经有值，则键值会被更新，否则就新生成该键。（因为返回的是当前Map对象，所以可以链式调用）
- **get(key)**： 该方法读取key对应的键值，如果找不到key，返回undefined。
- **has(key)**： 该方法返回一个布尔值，表示某个键是否在当前Map对象中。
- **delete(key)**： 该方法删除某个键，返回true，如果删除失败，返回false。
- **clear()**： `map.clear()`清除所有成员，没有返回值。

Map结构原生提供三个遍历器生成函数和一个遍历方法

- **keys()**： 返回键名的遍历器。

- values(): 返回键值的遍历器。
- entries(): 返回所有成员的遍历器。
- forEach(): 遍历Map的所有成员。

```
const map = new Map([
  ["foo", 1],
  ["bar", 2],
])
for(let key of map.keys()){
  console.log(key); // foo bar
}
for(let value of map.values()){
  console.log(value); // 1 2
}
for(let items of map.entries()){
  console.log(items); // ["foo",1] ["bar",2]
}
map.forEach( (value,key,map) => {
  console.log(key,value); // foo 1 bar 2
})
```

复制代码

(2) **WeakMap** WeakMap 对象也是一组键值对的集合，其中的键是弱引用的。**其键必须是对象**，原始数据类型不能作为key值，而值可以是任意的。

该对象也有以下几种方法：

- **set(key,value)**: 设置键名key对应的键值value，然后返回整个Map结构，如果key已经有值，则键值会被更新，否则就新生成该键。（因为返回的是当前Map对象，所以可以链式调用）
- **get(key)**: 该方法读取key对应的键值，如果找不到key，返回undefined。
- **has(key)**: 该方法返回一个布尔值，表示某个键是否在当前Map对象中。
- **delete(key)**: 该方法删除某个键，返回true，如果删除失败，返回false。

其clear()方法已经被弃用，所以可以通过创建一个空的WeakMap并替换原对象来实现清除。

WeakMap的设计目的在于，有时想在某个对象上面存放一些数据，但是这会形成对于这个对象的引用。一旦不再需要这两个对象，就必须手动删除这个引用，否则垃圾回收机制就不会释放对象占用的内存。

而WeakMap的**键名所引用的对象都是弱引用**，即垃圾回收机制不将该引用考虑在内。因此，只要所引用的对象的其他引用都被清除，垃圾回收机制就会释放该对象所占用的内存。也就是说，一旦不再需要，WeakMap 里面的**键名对象和所对应的键值对会自动消失，不用手动删除引用**。

**总结：**

- Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。
- WeakMap 结构与 Map 结构类似，也是用于生成键值对的集合。但是 WeakMap 只接受对象作为键名（null 除外），不接受其他类型的值作为键名。而且 WeakMap 的键名所指向的对象，不计入垃圾回收机制。

## 4. JavaScript有哪些内置对象

全局的对象（global objects）或称标准内置对象，不要和“全局对象（global object）”混淆。这里说的全局的对象是说在 全局作用域里的对象。全局作用域中的其他对象可以由用户的脚本创建或由宿主程序提供。

## 标准内置对象的分类：

- (1) 值属性，这些全局属性返回一个简单值，这些值没有自己的属性和方法。例如 Infinity、NaN、undefined、null 字面量
- (2) 函数属性，全局函数可以直接调用，不需要在调用时指定所属对象，执行结束后会将结果直接返回给调用者。例如 eval()、parseFloat()、parseInt() 等
- (3) 基本对象，基本对象是定义或使用其他对象的基础。基本对象包括一般对象、函数对象和错误对象。例如 Object、Function、Boolean、Symbol、Error 等
- (4) 数字和日期对象，用来表示数字、日期和执行数学计算的对象。例如 Number、Math、Date
- (5) 字符串，用来表示和操作字符串的对象。例如 String、RegExp
- (6) 可索引的集合对象，这些对象表示按照索引值来排序的数据集合，包括数组和类型数组，以及类数组结构的对象。例如 Array
- (7) 使用键的集合对象，这些集合对象在存储数据时会使用到键，支持按照插入顺序来迭代元素。例如 Map、Set、WeakMap、WeakSet
- (8) 矢量集合，SIMD 矢量集合中的数据会被组织为一个数据序列。例如 SIMD 等
- (9) 结构化数据，这些对象用来表示和操作结构化的缓冲区数据，或使用 JSON 编码的数据。例如 JSON 等
- (10) 控制抽象对象 例如 Promise、Generator 等
- (11) 反射。例如 Reflect、Proxy
- (12) 国际化，为了支持多语言处理而加入 ECMAScript 的对象。例如 Intl、Intl.Collator 等
- (13) WebAssembly
- (14) 其他。例如 arguments

**总结：**js 中的内置对象主要指的是在程序执行前存在全局作用域里的由 js 定义的一些全局值属性、函数和用来实例化其他对象的构造函数对象。一般经常用到的如全局变量值 NaN、undefined，全局函数如 parseInt()、parseFloat() 用来实例化对象的构造函数如 Date、Object 等，还有提供数学计算的单体内置对象如 Math 对象。

## 5. 常用的正则表达式有哪些？

```
// (1) 匹配 16 进制颜色值
var regex = /#[0-9a-fA-F]{6}|[0-9a-fA-F]{3}/g;

// (2) 匹配日期，如 yyyy-mm-dd 格式
var regex = /^([0-9]{4})-([01-9]|1[0-2])-([01-9]|12|[0-9]|3[01])$/;

// (3) 匹配 qq 号
var regex = /^[1-9][0-9]{4,10}$/g;

// (4) 手机号码正则
var regex = /^1[34578]\d{9}$/g;

// (5) 用户名正则
var regex = /^[a-zA-Z$][a-zA-Z0-9_\$]{4,16}$/;
复制代码
```

## 6. 对JSON的理解

JSON 是一种基于文本的轻量级的数据交换格式。它可以被任何的编程语言读取和作为数据格式来传递。

在项目开发中，使用 JSON 作为前后端数据交换的方式。在前端通过将一个符合 JSON 格式的数据结构序列化为 JSON 字符串，然后将它传递到后端，后端通过 JSON 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。

因为 JSON 的语法是基于 js 的，因此很容易将 JSON 和 js 中的对象弄混，但是应该注意的是 JSON 和 js 中的对象不是一回事，JSON 中对象格式更加严格，比如说在 JSON 中属性值不能为函数，不能出现 NaN 这样的属性值等，因此大多数的 js 对象是不符合 JSON 对象的格式的。

在 js 中提供了两个函数来实现 js 数据结构和 JSON 格式的转换处理，

- JSON.stringify 函数，通过传入一个符合 JSON 格式的数据结构，将其转换为一个 JSON 字符串。如果传入的数据结构不符合 JSON 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使其符合规范。在前端向后端发送数据时，可以调用这个函数将数据对象转化为 JSON 格式的字符串。
- JSON.parse() 函数，这个函数用来将 JSON 格式的字符串转换为一个 js 数据结构，如果传入的字符串不是标准的 JSON 格式的字符串的话，将会抛出错误。当从后端接收到 JSON 格式的字符串时，可以通过这个方法将其解析为一个 js 数据结构，以此来进行数据的访问。

## 7. JavaScript脚本延迟加载的方式有哪些？

延迟加载就是等页面加载完成之后再加载 JavaScript 文件。js 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

- **defer 属性：** 给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。
- **async 属性：** 给 js 脚本添加 async 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。
- **动态创建 DOM 方式：** 动态创建 DOM 标签的方式，可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。
- **使用 setTimeout 延迟方法：** 设置一个定时器来延迟加载js脚本文件
- **让 JS 最后加载：** 将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。

## 8. JavaScript 类数组对象的定义？

一个拥有 length 属性和若干索引属性的对象就可以被称为类数组对象，类数组对象和数组类似，但是不能调用数组的方法。常见的类数组对象有 arguments 和 DOM 方法的返回结果，还有一个函数也可以被看作是类数组对象，因为它含有 length 属性值，代表可接收的参数个数。

常见的类数组转换为数组的方法有这样几种：

- (1) 通过 call 调用数组的 slice 方法来实现转换

```
Array.prototype.slice.call(arrayLike);  
复制代码
```

- (2) 通过 call 调用数组的 splice 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);  
复制代码
```



(3) 通过 `apply` 调用数组的 `concat` 方法来实现转换

```
Array.prototype.concat.apply([], arrayLike);
```

复制代码

(4) 通过 `Array.from` 方法来实现转换

```
Array.from(arrayLike);
```

复制代码

## 9. 数组有哪些原生方法？

- 数组和字符串的转换方法：`toString()`、`toLocaleString()`、`join()` 其中 `join()` 方法可以指定转换为字符串时的分隔符。
- 数组尾部操作的方法 `pop()` 和 `push()`，`push` 方法可以传入多个参数。
- 数组首部操作的方法 `shift()` 和 `unshift()` 重排序的方法 `reverse()` 和 `sort()`，`sort()` 方法可以传入一个函数来进行比较，传入前后两个值，如果返回值为正数，则交换两个参数的位置。
- 数组连接的方法 `concat()`，返回的是拼接好的数组，不影响原数组。
- 数组截取办法 `slice()`，用于截取数组中的一部分返回，不影响原数组。
- 数组插入方法 `splice()`，影响原数组查找特定项的索引的方法，`indexOf()` 和 `lastIndexOf()` 迭代方法 `every()`、`some()`、`filter()`、`map()` 和 `forEach()` 方法
- 数组归并方法 `reduce()` 和 `reduceRight()` 方法

## 10. Unicode、UTF-8、UTF-16、UTF-32的区别？

### (1) Unicode

在说 `Unicode` 之前需要先了解一下 `ASCII` 码：`ASCII` 码（`American Standard Code for Information Interchange`）称为美国标准信息交换码。

- 它是基于拉丁字母的一套电脑编码系统。
- 它定义了一个用于代表常见字符的字典。
- 它包含了"A-Z"(包含大小写)，数据"0-9" 以及一些常见的符号。
- 它是专门为英语而设计的，有128个编码，对其他语言无能为力

`ASCII` 码可以表示的编码有限，要想表示其他语言的编码，还是要使用 `Unicode` 来表示，可以说 `Unicode` 是 `ASCII` 的超集。

`Unicode` 全称 `Unicode Translation Format`，又叫做统一码、万国码、单一码。`Unicode` 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。

`Unicode` 的实现方式（也就是编码方式）有很多种，常见的是 `UTF-8`、`UTF-16`、`UTF-32` 和 `USC-2`。

### (2) UTF-8

`UTF-8` 是使用最广泛的 `Unicode` 编码方式，它是一种可变长的编码方式，可以是1—4个字节不等，它可以完全兼容 `ASCII` 码的128个字符。

**注意：** `UTF-8` 是一种编码方式，`Unicode` 是一个字符集合。

`UTF-8` 的编码规则：

- 对于单字节的符号，字节的第一位为0，后面的7位为这个字符的 `Unicode` 编码，因此对于英文字母，它的 `Unicode` 编码和 `ASCII` 编码一样。

- 对于n字节的符号，第一个字节的前n位都是1，第n+1位设为0，后面字节的前两位一律设为10，剩下的没有提及的二进制位，全部为这个符号的 unicode 码。

来看一下具体的 unicode 编号范围与对应的 UTF-8 二进制格式：

编码范围（编号对应的十进制数）	二进制格式
0x00—0x7F (0-127)	0xxxxxxx
0x80—0x7FF (128-2047)	110xxxxx 10xxxxxx
0x800—0xFFFF (2048-65535)	1110xxxx 10xxxxxx 10xxxxxx
0x10000—0x10FFFF (65536以上)	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

那该如何通过具体的 unicode 编码，进行具体的 UTF-8 编码呢？步骤如下：

- 找到该 unicode 编码的所在的编号范围，进而找到与之对应的二进制格式
- 将 unicode 编码转换为二进制数（去掉最高位的0）
- 将二进制数从右往左一次填入二进制格式的 x 中，如果有 x 未填，就设为0

来看一个实际的例子：“马”字的 unicode 编码是：0x9A6C，整数编号是 39532（1）首选确定了该字符在第三个范围内，它的格式是 1110xxxx 10xxxxxx 10xxxxxx（2）39532对应的二进制数为 1001 1010 0110 1100（3）将二进制数填入X中，结果是：11101001 10101001 10101100

### (3) UTF-16

#### 1. 平面的概念

在了解 UTF-16 之前，先看一下平面的概念：unicode 编码中有很多很多的字符，它并不是一次性定义的，而是分区进行定义的，每个区存放65536（216）个字符，这称为一个平面，目前总共有17个平面。

最前面的一个平面称为基本平面，它的码点从0 — 216-1，写成16进制就是 U+0000 — U+FFFF，那剩下的16个平面就是辅助平面，码点范围是 U+10000-U+10FFFF。

#### 2. UTF-16 概念：

UTF-16 也是 unicode 编码集的一种编码形式，把 unicode 字符集的抽象码位映射为16位长的整数（即码元）的序列，用于数据存储或传递。Unicode 字符的码位需要1个或者2个16位长的码元来表示，因此 UTF-16 也是用变长字节表示的。

#### 3. UTF-16 编码规则：

- 编号在 U+0000-U+FFFF 的字符（常用字符集），直接用两个字节表示。
- 编号在 U+10000-U+10FFFF 之间的字符，需要用四个字节表示。

#### 4. 编码识别

那么问题来了，当遇到两个字节时，怎么知道是把它当做一个字符还是和后面的两个字节一起当做一个字符呢？

UTF-16 编码肯定也考虑到了这个问题，在基本平面内，从 U+D800 — U+DFFF 是一个空段，也就是说这个区间的码点不对应任何的字符，因此这些空段就可以用来映射辅助平面的字符。

辅助平面共有 220 个字符位，因此表示这些字符至少需要 20 个二进制位。UTF-16 将这 20 个二进制位分成两半，前 10 位映射在 U+D800 — U+DBFF，称为高位（H），后 10 位映射在 U+DC00 — U+DFFF，称为低位（L）。这就相当于，将一个辅助平面的字符拆成了两个基本平面的字符来表示。

因此，当遇到两个字节时，发现它的码点在 U+D800 -U+DBFF 之间，就可以知道，它后面的两个字节的码点应该在 U+DC00 - U+DFFF 之间，这四个字节必须放在一起进行解读。

## 5. 举例说明

以 "搞" 字为例，它的 Unicode 码点为 0x21800，该码点超出了基本平面的范围，因此需要用四个字节来表示，步骤如下：

- 首先计算超出部分的结果：0x21800 - 0x10000
- 将上面的计算结果转为20位的二进制数，不足20位就在前面补0，结果为：00010001100000000000
- 将得到的两个10位二进制数分别对应到两个区间中
- U+D800 对应的二进制数为 1101100000000000，将 0001000110 填充在它的后10个二进制位，得到 1101100001000110，转成16进制数为 0xD846。同理，低位为 0xDC00，所以这个字的 UTF-16 编码为 0xD846 0xDC00

## (4) UTF-32

UTF-32 就是字符所对应编号的整数二进制形式，每个字符占四个字节，这个是直接进行转换的。该编码方式占用的储存空间较多，所以使用较少。

比如“马”字的Unicode编号是：U+9A6C，整数编号是 39532，直接转化为二进制：1001 1010 0110 1100，这就是它的UTF-32编码。

## (5) 总结

Unicode、UTF-8、UTF-16、UTF-32有什么区别？

- Unicode 是编码字符集（字符集），而 UTF-8、UTF-16、UTF-32 是字符集编码（编码规则）；
- UTF-16 使用变长码元序列的编码方式，相较于定长码元序列的 UTF-32 算法更复杂，甚至比同样是变长码元序列的 UTF-8 也更为复杂，因为其引入了独特的代理对这样的代理机制；
- UTF-8 需要判断每个字节中的开头标志信息，所以如果某个字节在传送过程中出错了，就会导致后面的字节也会解析出错；而 UTF-16 不会判断开头标志，即使错也只会错一个字符，所以容错能力较强；
- 如果字符内容全部英文或英文与其他文字混合，但英文占绝大部分，那么用 UTF-8 就比 UTF-16 节省了很多空间；而如果字符内容全部是中文这样类似的字符或者混合字符中中文占绝大多数，那么 UTF-16 就占优势了，可以节省很多空间；

## 11. 常见的位运算符有哪些？其计算规则是什么？

现代计算机中数据都是以二进制的形式存储的，即0、1两种状态，计算机对二进制数据进行的运算加减乘除等都是叫位运算，即将符号位共同参与运算的运算。

常见的位运算有以下几种：

运算符	描述	运算规则
&	与	两个位都为1时，结果才为1
、	、	或
^	异或	两个位相同为0，相异为1
~	取反	0变1，1变0
<<	左移	各二进制位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃

## 1. 按位与运算符 (&)

**定义：** 参加运算的两个数据按**二进制位**进行“与”运算。 **运算规则：**

```
0 & 0 = 0
```

```
0 & 1 = 0
```

```
1 & 0 = 0
```

```
1 & 1 = 1
```

复制代码

总结：两位同时为1，结果才为1，否则结果为0。 例如：3&5 即：

```
0000 0011
```

```
0000 0101
```

```
= 0000 0001
```

复制代码

因此 3&5 的值为1。 注意：负数按补码形式参加按位与运算。

**用途：**

### (1) 判断奇偶

只要根据最末位是0还是1来决定，为0就是偶数，为1就是奇数。因此可以用 `if ((i & 1) == 0)` 代替 `if (i % 2 == 0)` 来判断a是不是偶数。

### (2) 清零

如果想将一个单元清零，即使其全部二进制位为0，只要与一个各位都为零的数值相与，结果为零。

## 2. 按位或运算符 (|)

**定义：** 参加运算的两个对象按二进制位进行“或”运算。

**运算规则：**

```
0 | 0 = 0
```

```
0 | 1 = 1
```

```
1 | 0 = 1
```

```
1 | 1 = 1
```

复制代码

总结：参加运算的两个对象只要有一个为1，其值为1。 例如：3|5即：

```
0000 0011
  0000 0101
= 0000 0111
复制代码
```

因此， $3|5$ 的值为7。注意：负数按补码形式参加按位或运算。

### 3. 异或运算符 (^)

**定义：** 参加运算的两个数据按二进制位进行“异或”运算。

**运算规则：**

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
复制代码
```

总结：参加运算的两个对象，如果两个相应位相同为0，相异为1。例如： $3|5$ 即：

```
0000 0011
  0000 0101
= 0000 0110
复制代码
```

因此， $3^5$ 的值为6。异或运算的性质：

- 交换律： $(a^b)^c == a^(b^c)$
- 结合律： $(a + b)^c == a^b + b^c$
- 对于任何数x，都有  $x^x=0$ ， $x^0=x$
- 自反性： $a^b^b=a^0=a$ ；

### 4. 取反运算符 (~)

**定义：** 参加运算的一个数据按二进制进行“取反”运算。

**运算规则：**

```
~ 1 = 0 ~ 0 = 1
复制代码
```

总结：对一个二进制数按位取反，即将0变1，1变0。例如： $\sim 6$  即：

```
0000 0110 = 1111 1001
复制代码
```

在计算机中，正数用原码表示，负数使用补码存储，首先看最高位，最高位1表示负数，0表示正数。此计算机二进制码为负数，最高位为符号位。当发现按位取反为负数时，就**直接取其补码**，变为十进制：

```
0000 0110 = 1111 1001反码: 1000 0110补码: 1000 0111
复制代码
```

因此， $\sim 6$ 的值为-7。

## 5. 左移运算符 (<<)

**定义：** 将一个运算对象的各二进制位全部左移若干位，左边的二进制位丢弃，右边补0。设  $a=1010\ 1110$ ， $a = a << 2$  将a的二进制位左移2位、右补0，即得 $a=1011\ 1000$ 。若左移时舍弃的高位不包含1，则每左移一位，相当于该数乘以2。

## 6. 右移运算符 (>>)

**定义：** 将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。例如： $a=a >> 2$  将a的二进制位右移2位，左补0 或者 左补1得看被移数是正还是负。操作数每右移一位，相当于该数除以2。

## 7. 原码、补码、反码

上面提到了补码、反码等知识，这里就补充一下。计算机中的**有符号数**有三种表示方法，即原码、反码和补码。三种表示方法均有符号位和数值位两部分，符号位都是用0表示“正”，用1表示“负”，而数值位，三种表示方法各不相同。

### (1) 原码

原码就是一个数的二进制数。例如：10的原码为0000 1010

### (2) 反码

- 正数的反码与原码相同，如：10 反码为 0000 1010
- 负数的反码为除符号位，按位取反，即0变1，1变0。

例如：-10

```
原码：1000 1010
反码：1111 0101
复制代码
```

### (3) 补码

- 正数的补码与原码相同，如：10 补码为 0000 1010
- 负数的补码是原码除符号位外的所有位取反即0变1，1变0，然后加1，也就是反码加1。

例如：-10

```
原码：1000 1010
反码：1111 0101
补码：1111 0110
复制代码
```

## 12. 为什么函数的 arguments 参数是类数组而不是数组？如何遍历类数组？

`arguments` 是一个对象，它的属性是从 0 开始依次递增的数字，还有 `callee` 和 `length` 等属性，与数组相似；但是它却没有数组常见的方法属性，如 `forEach`，`reduce` 等，所以叫它们类数组。

要遍历类数组，有三个方法：

- (1) 将数组的方法应用到类数组上，这时候就可以使用 `call` 和 `apply` 方法，如：

```
function foo(){
  Array.prototype.forEach.call(arguments, a => console.log(a))
}
```

复制代码

(2) 使用Array.from方法将类数组转化成数组:

```
function foo(){
  const arrArgs = Array.from(arguments)
  arrArgs.forEach(a => console.log(a))
}
```

复制代码

(3) 使用展开运算符将类数组转化成数组

```
function foo(){
  const arrArgs = [...arguments]
  arrArgs.forEach(a => console.log(a))
}
```

复制代码

## 13. 什么是 DOM 和 BOM?

- DOM 指的是文档对象模型，它指的是把文档当做一个对象，这个对象主要定义了处理网页内容的方法和接口。
- BOM 指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的方法和接口。BOM的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局）对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。window 对象含有 location 对象、navigator 对象、screen 对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象。

## 14. 对类数组对象的理解，如何转化为数组

一个拥有 length 属性和若干索引属性的对象就可以被称为类数组对象，类数组对象和数组类似，但是不能调用数组的方法。常见的类数组对象有 arguments 和 DOM 方法的返回结果，函数参数也可以被看作是类数组对象，因为它含有 length 属性值，代表可接收的参数个数。

常见的类数组转换为数组的方法有这样几种：

- 通过 call 调用数组的 slice 方法来实现转换

```
Array.prototype.slice.call(arrayLike);
```

复制代码

- 通过 call 调用数组的 splice 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);
```

复制代码

- 通过 apply 调用数组的 concat 方法来实现转换



```
Array.prototype.concat.apply([], arrayLike);
```

复制代码

- 通过 `Array.from` 方法来实现转换

```
Array.from(arrayLike);
```

复制代码

## 15. escape、encodeURIComponent、encodeURIComponent 的区别

- `encodeURIComponent` 是对整个 URI 进行转义，将 URI 中的非法字符转换为合法字符，所以对于一些在 URI 中有特殊意义的字符不会进行转义。
- `encodeURIComponent` 是对 URI 的组成部分进行转义，所以一些特殊字符也会得到转义。
- `escape` 和 `encodeURIComponent` 的作用相同，不过它们对于 unicode 编码为 0xff 之外字符的时候会有区别，`escape` 是直接在字符的 unicode 编码前加上 %u，而 `encodeURIComponent` 首先会将字符转换为 UTF-8 的格式，再在每个字节前加上 %。

## 16. 对AJAX的理解，实现一个AJAX请求

AJAX是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

创建AJAX请求的步骤：

- **创建一个 XMLHttpRequest 对象。**
- 在这个对象上使用 **open 方法创建一个 HTTP 请求**，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。
- 在发起请求前，可以为这个对象**添加一些信息和监听函数**。比如说可以通过 `setRequestHeader` 方法来为请求添加头信息。还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发 `onreadystatechange` 事件，可以通过设置监听函数，来处理请求成功后的结果。当对象的 `readyState` 变为 4 的时候，代表服务器返回的数据接收完成，这个时候可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候就可以通过 `response` 中的数据来对页面进行更新了。
- 当对象的属性和监听函数设置完成后，最后调用 **send 方法来向服务器发起请求**，可以传入参数作为发送的数据体。

```
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", url, true);
// 设置状态监听函数
xhr.onreadystatechange = function() {
  if (this.readyState !== 4) return;
  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};
// 设置请求失败时的监听函数
xhr.onerror = function() {
  console.error(this.statusText);
};
```

```
// 设置请求头信息
xhr.responseText = "json";
xhr.setRequestHeader("Accept", "application/json");
// 发送 Http 请求
xhr.send(null);
复制代码
```

使用Promise封装AJAX:

```
// promise 封装实现:
function getJSON(url) {
  // 创建一个 promise 对象
  let promise = new Promise(function(resolve, reject) {
    let xhr = new XMLHttpRequest();
    // 新建一个 http 请求
    xhr.open("GET", url, true);
    // 设置状态的监听函数
    xhr.onreadystatechange = function() {
      if (this.readyState !== 4) return;
      // 当请求成功或失败时, 改变 promise 的状态
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    // 设置错误监听函数
    xhr.onerror = function() {
      reject(new Error(this.statusText));
    };
    // 设置响应的数据类型
    xhr.responseText = "json";
    // 设置请求头信息
    xhr.setRequestHeader("Accept", "application/json");
    // 发送 http 请求
    xhr.send(null);
  });
  return promise;
}
复制代码
```

## 17. JavaScript为什么要进行变量提升, 它导致了什么问题?

变量提升的表现是, 无论在函数中何处位置声明的变量, 好像都被提升到了函数的首部, 可以在变量声明前访问到而不会报错。

造成变量声明提升的**本质原因**是 js 引擎在代码执行前有一个解析的过程, 创建了执行上下文, 初始化了一些代码执行时需要用到的对象。当访问一个变量时, 会到当前执行上下文中的作用域链中去查找, 而作用域链的首端指向的是当前执行上下文的变量对象, 这个变量对象是执行上下文的一个属性, 它包含了函数的形参、所有的函数和变量声明, 这个对象的是在代码解析的时候创建的。

首先要知道, JS在拿到一个变量或者一个函数的时候, 会有两步操作, 即解析和执行。

- 在解析阶段

，JS会检查语法，并对函数进行预编译。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值为undefined，函数先声明好可使用。在一个函数执行之前，也会创建一个函数执行上下文环境，跟全局执行上下文类似，不过函数执行上下文会多出this、arguments和函数的参数。

- 全局上下文：变量定义，函数声明
- 函数上下文：变量定义，函数声明，this，arguments
- **在执行阶段**，就是按照代码的顺序依次执行。

那为什么会进行变量提升呢？主要有以下两个原因：

- 提高性能
- 容错性更好

**(1) 提高性能** 在JS代码执行之前，会进行语法检查和预编译，并且这一操作只进行一次。这么做就是为了提高性能，如果没有这一步，那么每次执行代码前都必须重新解析一遍该变量（函数），而这是没有必要的，因为变量（函数）的代码并不会改变，解析一遍就够了。

在解析的过程中，还会为函数生成预编译代码。在预编译时，会统计声明了哪些变量、创建了哪些函数，并对函数的代码进行压缩，去除注释、不必要的空白等。这样做的好处就是每次执行函数时都可以直接为该函数分配栈空间（不需要再解析一遍去获取代码中声明了哪些变量，创建了哪些函数），并且因为代码压缩的原因，代码执行也更快了。

## **(2) 容错性更好**

变量提升可以在一定程度上提高JS的容错性，看下面的代码：

```
a = 1; var a; console.log(a);
```

复制代码

如果没有变量提升，这两行代码就会报错，但是因为有了变量提升，这段代码就可以正常执行。

虽然，在开发过程中，可以完全避免这样写，但是有时代码很复杂的时候。可能因为疏忽而先使用后定义了，这样也不会影响正常使用。由于变量提升的存在，而会正常运行。

## **总结：**

- 解析和预编译过程中的声明提升可以提高性能，让函数可以在执行时预先为变量分配栈空间
- 声明提升还可以提高JS代码的容错性，使一些不规范的代码也可以正常执行

变量提升虽然有一些优点，但是也会造成一定的问题，在ES6中提出了let、const来定义变量，它们就没有变量提升的机制。下面看一下变量提升可能会导致的问题：

```
var tmp = new Date();

function fn(){
  console.log(tmp);
  if(false){
    var tmp = 'hello world';
  }
}

fn(); // undefined
```

复制代码

在这个函数中，原本是要打印出外层的tmp变量，但是因为变量提升的问题，内层定义的tmp被提到函数内部的最顶部，相当于覆盖了外层的tmp，所以打印结果为undefined。

```
var tmp = 'hello world';

for (var i = 0; i < tmp.length; i++) {
  console.log(tmp[i]);
}

console.log(i); // 11
复制代码
```

由于遍历时定义的*i*会变量提升成为一个全局变量，在函数结束之后不会被销毁，所以打印出来11。

## 18. 什么是尾调用，使用尾调用有什么好处？

尾调用指的是函数的最后一步调用另一个函数。代码执行是基于执行栈的，所以当在一个函数里调用另一个函数时，会保留当前的执行上下文，然后再新建另外一个执行上下文加入栈中。使用尾调用的话，因为已经是函数的最后一步，所以这时可以不必再保留当前的执行上下文，从而节省了内存，这就是尾调用优化。但是 ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。

## 19. ES6模块与CommonJS模块有什么异同？

ES6 Module和CommonJS模块的区别：

- CommonJS是对模块的浅拷贝，ES6 Module是对模块的引用，即ES6 Module只存只读，不能改变其值，也就是指针指向不能变，类似const；
- import的接口是read-only（只读状态），不能修改其变量值。即不能修改其变量的指针指向，但可以改变变量内部指针指向，可以对commonJS对重新赋值（改变指针指向），但是对ES6 Module赋值会编译报错。

ES6 Module和CommonJS模块的共同点：

- CommonJS和ES6 Module都可以对引入的对象进行赋值，即对对象内部属性的值进行改变。

## 20. 常见的DOM操作有哪些

### 1) DOM 节点的获取

DOM 节点的获取的API及使用：

```
getElementById // 按照 id 查询
getElementsByTagName // 按照标签名查询
getElementsByClassName // 按照类名查询
querySelectorAll // 按照 css 选择器查询

// 按照 id 查询
var imooc = document.getElementById('imooc') // 查询到 id 为 imooc 的元素
// 按照标签名查询
var pList = document.getElementsByTagName('p') // 查询到标签为 p 的集合
console.log(divList.length)
console.log(divList[0])
// 按照类名查询
var mooList = document.getElementsByClassName('moo') // 查询到类名为 moo 的集合
// 按照 css 选择器查询
var pList = document.querySelectorAll('.moo') // 查询到类名为 moo 的集合
复制代码
```

## 2) DOM 节点的创建

创建一个新节点，并把它添加到指定节点的后面。已知的 HTML 结构如下：

```
<html>
  <head>
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
    </div>
  </body>
</html>
```

复制代码

要求添加一个有内容的 span 节点到 id 为 title 的节点后面，做法就是：

```
// 首先获取父节点
var container = document.getElementById('container')
// 创建新节点
var targetSpan = document.createElement('span')
// 设置 span 节点的内容
targetSpan.innerHTML = 'hello world'
// 把新创建的元素塞进父节点里去
container.appendChild(targetSpan)
```

复制代码

## 3) DOM 节点的删除

删除指定的 DOM 节点，已知的 HTML 结构如下：

```
<html>
  <head>
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
    </div>
  </body>
</html>
```

复制代码

需要删除 id 为 title 的元素，做法是：

```
// 获取目标元素的父元素
var container = document.getElementById('container')
// 获取目标元素
var targetNode = document.getElementById('title')
// 删除目标元素
container.removeChild(targetNode)
```

复制代码

或者通过子节点数组来完成删除：

```
// 获取目标元素的父元素var container = document.getElementById('container')// 获取目标元素var targetNode = container.childNodes[1]// 删除目标元素container.removeChild(targetNode)
```

复制代码

## 4) 修改 DOM 元素

修改 DOM 元素这个动作可以分很多维度，比如说移动 DOM 元素的位置，修改 DOM 元素的属性等。

将指定的两个 DOM 元素交换位置，已知的 HTML 结构如下：

```
<html>
  <head>
    <title>DEMO</title>
  </head>
  <body>
    <div id="container">
      <h1 id="title">我是标题</h1>
      <p id="content">我是内容</p>
    </div>
  </body>
</html>
```

复制代码

现在需要调换 title 和 content 的位置，可以考虑 insertBefore 或者 appendChild：

```
// 获取父元素
var container = document.getElementById('container')

// 获取两个需要被交换的元素
var title = document.getElementById('title')
var content = document.getElementById('content')
// 交换两个元素，把 content 置于 title 前面
container.insertBefore(content, title)
```

复制代码

## 21. use strict是什么意思？使用它区别是什么？

use strict 是一种 ECMAScript5 添加的（严格模式）运行模式，这种模式使得 Javascript 在更严格的条件下运行。设立严格模式的目的如下：

- 消除 Javascript 语法的不合理、不严谨之处，减少怪异行为；
- 消除代码运行的不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的 Javascript 做好铺垫。

区别：

- 禁止使用 with 语句。
- 禁止 this 关键字指向全局对象。
- 对象不能有重名的属性。

## 22. 如何判断一个对象是否属于某个类？

- 第一种方式，使用 `instanceof` 运算符来判断构造函数的 `prototype` 属性是否出现在对象的原型链中的任何位置。
- 第二种方式，通过对象的 `constructor` 属性来判断，对象的 `constructor` 属性指向该对象的构造函数，但是这种方式不是很安全，因为 `constructor` 属性可以被改写。
- 第三种方式，如果需要判断的是某个内置的引用类型的话，可以使用 `Object.prototype.toString()` 方法来打印对象的`[[Class]]` 属性来进行判断。

## 23. 强类型语言和弱类型语言的区别

- **强类型语言**：强类型语言也称为强类型定义语言，是一种总是强制类型定义的语言，要求变量的使用要严格符合定义，所有变量都必须先定义后使用。Java和C++等语言都是强制类型定义的，也就是说，一旦一个变量被指定了某个数据类型，如果不经过强制转换，那么它就永远是这个数据类型了。例如你有一个整数，如果不显式地进行转换，你不能将其视为一个字符串。
- **弱类型语言**：弱类型语言也称为弱类型定义语言，与强类型定义相反。JavaScript语言就属于弱类型语言。简单理解就是一种变量类型可以被忽略的语言。比如JavaScript是弱类型定义的，在JavaScript中就可以将字符串'12'和整数3进行连接得到字符串'123'，在相加的时候会进行强制类型转换。

两者对比：强类型语言在速度上可能略逊色于弱类型语言，但是强类型语言带来的严谨性可以有效地帮助避免许多错误。

## 24. 解释性语言和编译型语言的区别

(1) 解释型语言 使用专门的解释器对源程序逐行解释成特定平台的机器码并立即执行。是代码在执行时才被解释器一行行动态翻译和执行，而不是在执行之前就完成翻译。解释型语言不需要事先编译，其直接将源代码解释成机器码并立即执行，所以只要某一平台提供了相应的解释器即可运行该程序。其特点总结如下

- 解释型语言每次运行都需要将源代码解释成机器码并执行，效率较低；
- 只要平台提供相应的解释器，就可以运行源代码，所以可以方便源程序移植；
- JavaScript、Python等属于解释型语言。

(2) 编译型语言 使用专门的编译器，针对特定的平台，将高级语言源代码一次性的编译成可被该平台硬件执行的机器码，并包装成该平台所能识别的可执行性程序的格式。在编译型语言写的程序执行之前，需要一个专门的编译过程，把源代码编译成机器语言的文件，如exe格式的文件，以后要再运行时，直接使用编译结果即可，如直接运行exe文件。因为只需编译一次，以后运行时不需要编译，所以编译型语言执行效率高。其特点总结如下：

- 一次性的编译成平台相关的机器语言文件，运行时脱离开发环境，运行效率高；
- 与特定平台相关，一般无法移植到其他平台；
- C、C++等属于编译型语言。

**两者主要区别在于：**前者源程序编译后即可在该平台运行，后者是在运行期间才编译。所以前者运行速度快，后者跨平台性好。

## 25. for...in和for...of的区别

`for...of` 是ES6新增的遍历方式，允许遍历一个含有`iterator`接口的数据结构（数组、对象等）并且返回各项的值，和ES3中的`for...in`的区别如下

- `for...of` 遍历获取的是对象的键值，`for...in` 获取的是对象的键名；
- `for... in` 会遍历对象的整个原型链，性能非常差不推荐使用，而 `for ... of` 只遍历当前对象不会遍历原型链；

- 对于数组的遍历，for...in 会返回数组中所有可枚举的属性(包括原型链上可枚举的属性)，for...of 只返回数组的下标对应的属性值；

**总结：** for...in 循环主要是为了遍历对象而生，不适用于遍历数组；for...of 循环可以用来遍历数组、类数组对象，字符串、Set、Map 以及 Generator 对象。

## 26. 如何使用for...of遍历对象

for...of是作为ES6新增的遍历方式，允许遍历一个含有iterator接口的数据结构（数组、对象等）并且返回各项的值，普通的对象用for..of遍历是会报错的。

如果需要遍历的对象是类数组对象，用Array.from转成数组即可。

```
var obj = {
  0: 'one',
  1: 'two',
  length: 2
};
obj = Array.from(obj);
for(var k of obj){
  console.log(k)
}
```

复制代码

如果不是类数组对象，就给对象添加一个[Symbol.iterator]属性，并指向一个迭代器即可。

```
//方法一：
var obj = {
  a:1,
  b:2,
  c:3
};

obj[Symbol.iterator] = function(){
  var keys = Object.keys(this);
  var count = 0;
  return {
    next(){
      if(count<keys.length){
        return {value: obj[keys[count++]],done:false};
      }else{
        return {value:undefined,done:true};
      }
    }
  }
};

for(var k of obj){
  console.log(k);
}
```

```
// 方法二
var obj = {
  a:1,
  b:2,
  c:3
```



```
};
obj[Symbol.iterator] = function*(){
    var keys = Object.keys(obj);
    for(var k of keys){
        yield [k,obj[k]]
    }
};

for(var [k,v] of obj){
    console.log(k,v);
}
```

复制代码

## 27. ajax、axios、fetch的区别

(1) **AJAX** Ajax 即“Asynchronous Javascript And XML”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发技术。它是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。通过在后台与服务器进行少量数据交换，Ajax 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页（不使用 Ajax）如果需要更新内容，必须重载整个网页页面。其缺点如下：

- 本身是针对MVC编程，不符合前端MVVM的浪潮
- 基于原生XHR开发，XHR本身的架构不清晰
- 不符合关注分离（Separation of Concerns）的原则
- 配置和调用方式非常混乱，而且基于事件的异步模型不友好。

(2) **Fetch** fetch号称是AJAX的替代品，是在ES6出现的，使用了ES6中的promise对象。Fetch是基于promise设计的。Fetch的代码结构比起ajax简单多。**fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象。**

fetch的优点：

- 语法简洁，更加语义化
- 基于标准 Promise 实现，支持 async/await
- 更加底层，提供的API丰富（request, response）
- 脱离了XHR，是ES规范里新的实现方式

fetch的缺点：

- fetch只对网络请求报错，对400，500都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。
- fetch默认不会带cookie，需要添加配置项：fetch(url, {credentials: 'include'})
- fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费
- fetch没有办法原生监测请求的进度，而XHR可以

(3) **Axios** Axios 是一种基于Promise封装的HTTP客户端，其特点如下：

- 浏览器端发起XMLHttpRequests请求
- node端发起http请求
- 支持Promise API
- 监听请求和返回
- 对请求和返回进行转化
- 取消请求
- 自动转换json数据
- 客户端支持抵御XSRF攻击

## 28. 数组的遍历方法有哪些

方法	是否改变原数组	特点
forEach()	否	数组方法，不改变原数组，没有返回值
map()	否	数组方法，不改变原数组，有返回值，可链式调用
filter()	否	数组方法，过滤数组，返回包含符合条件的元素的数组，可链式调用
for...of	否	for...of遍历具有Iterator迭代器的对象的属性，返回的是数组的元素、对象的属性值，不能遍历普通的obj对象，将异步循环变成同步循环
every() 和 some()	否	数组方法，some()只要有一个是true，便返回true；而every()只要有一个是false，便返回false.
find() 和 findIndex()	否	数组方法，find()返回的是第一个符合条件的值；findIndex()返回的是第一个返回条件的值的索引值
reduce() 和 reduceRight()	否	数组方法，reduce()对数组正序操作；reduceRight()对数组逆序操作

遍历方法的详细解释：[《细数JavaScript中那些遍历和循环》](#)

## 29. forEach和map方法有什么区别

这方法都是用来遍历数组的，两者区别如下：

- forEach()方法会针对每一个元素执行提供的函数，对数据的操作会改变原数组，该方法没有返回值；
- map()方法不会改变原数组的值，返回一个新数组，新数组中的值为原数组调用函数处理之后的值；

## 四、原型与原型链

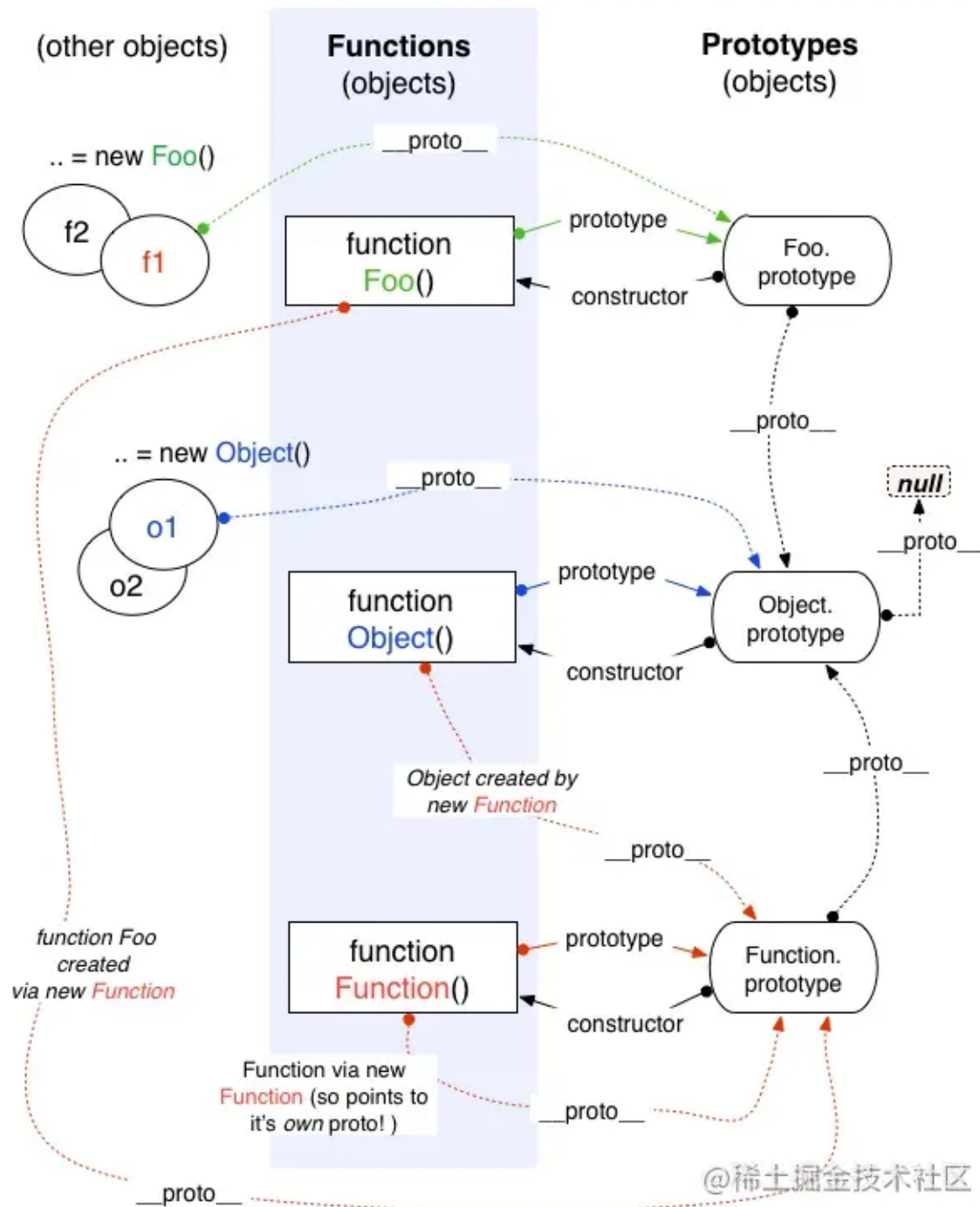
### 1. 对原型、原型链的理解

在JavaScript中是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性，它的属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说不应该能够获取到这个值的，但是现在浏览器中都实现了 **proto** 属性来访问这个属性，但是最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，可以通过这个方法来获取对象的原型。

当访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是新建的对象为什么能够使用 toString() 等方法的原因。

**特点：**JavaScript 对象是通过引用来传递的，创建的每个新对象实体中并没有一份属于自己的原型副本。当修改原型时，与之相关的对象也会继承这一改变。

## JavaScript Object Layout [Hursh Jain/[mollypages.org](http://mollypages.org)]



## 2. 原型修改、重写

```
function Person(name) {
    this.name = name
}

// 修改原型
Person.prototype.getName = function() {}
var p = new Person('hello')
console.log(p.__proto__ === Person.prototype) // true
console.log(p.__proto__ === p.constructor.prototype) // true
// 重写原型
Person.prototype = {
```

```

    getName: function() {}
}
var p = new Person('hello')
console.log(p.__proto__ === Person.prototype) // true
console.log(p.__proto__ === p.constructor.prototype) // false
复制代码

```

可以看到修改原型的时候p的构造函数不是指向Person了，因为直接给Person的原型对象直接用对象赋值时，它的构造函数指向的了根构造函数Object，所以这时候 `p.constructor === Object`，而不是 `p.constructor === Person`。要想成立，就要用constructor指回来：

```

Person.prototype = {
    getName: function() {}
}
var p = new Person('hello')
p.constructor = Person
console.log(p.__proto__ === Person.prototype) // true
console.log(p.__proto__ === p.constructor.prototype) // true
复制代码

```

### 3. 原型链指向

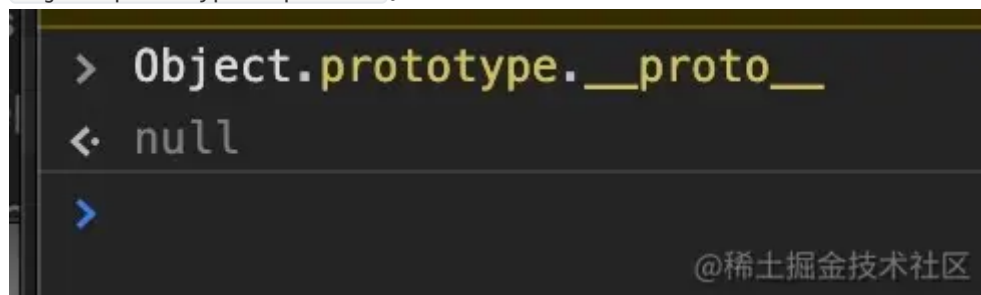
```

p.__proto__ // Person.prototype
Person.prototype.__proto__ // Object.prototype
p.__proto__.__proto__ // Object.prototype
p.__proto__.constructor.prototype.__proto__ // Object.prototype
Person.prototype.constructor.prototype.__proto__ // Object.prototype
p1.__proto__.constructor // Person
Person.prototype.constructor // Person
复制代码

```

### 4. 原型链的终点是什么？如何打印出原型链的终点？

由于Object是构造函数，原型链终点是 `Object.prototype.__proto__`，而 `Object.prototype.__proto__ === null // true`，所以，原型链的终点是 `null`。原型链上的所有原型都是对象，所有的对象最终都是由Object构造的，而 `Object.prototype` 的下一级是 `Object.prototype.__proto__`。



### 5. 如何获得对象非原型链上的属性？

使用后 `hasOwnProperty()` 方法来判断属性是否属于原型链的属性：

```
function iterate(obj){
  var res=[];
  for(var key in obj){
    if(obj.hasOwnProperty(key))
      res.push(key+' : '+obj[key]);
  }
  return res;
}
```

复制代码

## 五、执行上下文/作用域链/闭包

### 1. 对闭包的理解

**闭包是指有权访问另一个函数作用域中变量的函数**，创建闭包的最常见的方式就是在在一个函数内创建另一个函数，创建的函数可以访问到当前函数的局部变量。

闭包有两个常用的用途；

- 闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。
- 闭包的另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

比如，函数 A 内部有一个函数 B，函数 B 可以访问到函数 A 中的变量，那么函数 B 就是闭包。

```
function A() {
  let a = 1
  window.B = function () {
    console.log(a)
  }
}
A()
B() // 1
```

复制代码

在 JS 中，闭包存在的意义就是让我们可以间接访问函数内部的变量。经典面试题：循环中使用闭包解决 var 定义函数的问题

```
for (var i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

复制代码

首先因为 `setTimeout` 是个异步函数，所以会先把循环全部执行完毕，这时候 `i` 就是 6 了，所以会输出一堆 6。解决办法有三种：

- 第一种是使用闭包的方式

```
for (var i = 1; i <= 5; i++) { ;(function(j) {
  console.log(j)    }, j * 1000) })(i)}
  setTimeout(function timer() {
```

复制代码

在上述代码中，首先使用了立即执行函数将 `i` 传入函数内部，这个时候值就被固定在了参数 `j` 上面不会改变，当下次执行 `timer` 这个闭包的时候，就可以使用外部函数的变量 `j`，从而达到目的。

- 第二种就是使用 `setTimeout` 的第三个参数，这个参数会被当成 `timer` 函数的参数传入。

```
for (var i = 1; i <= 5; i++) {
  setTimeout(
    function timer(j) {
      console.log(j)
    },
    i * 1000,
    i
  )
}
```

复制代码

- 第三种就是使用 `let` 定义 `i` 来解决问题了，这个也是最为推荐的方式

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function timer() {
    console.log(i)
  }, i * 1000)
}
```

复制代码

## 2. 对作用域、作用域链的理解

### 1) 全局作用域和函数作用域

#### (1) 全局作用域

- 最外层函数和最外层函数外面定义的变量拥有全局作用域
- 所有未定义直接赋值的变量自动声明为全局作用域
- 所有window对象的属性拥有全局作用域
- 全局作用域有很大的弊端，过多的全局作用域变量会污染全局命名空间，容易引起命名冲突。

#### (2) 函数作用域

- 函数作用域声明在函数内部的变量，一般只有固定的代码片段可以访问到
- 作用域是分层的，内层作用域可以访问外层作用域，反之不行

### 2) 块级作用域

- 使用ES6中新增的`let`和`const`指令可以声明块级作用域，块级作用域可以在函数中创建也可以在一个代码块中的创建（由 `{ }` 包裹的代码片段）
- `let`和`const`声明的变量不会有变量提升，也不可以重复声明
- 在循环中比较适合绑定块级作用域，这样就可以把声明的计数器变量限制在循环内部。

**作用域链：** 在当前作用域中查找所需变量，但是该作用域没有这个变量，那这个变量就是自由变量。如果在自己作用域找不到该变量就去父级作用域查找，依次向上级作用域查找，直到访问到window对象就被终止，这一层层的关系就是作用域链。

作用域链的作用是**保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，可以访问到外层环境的变量和函数。**

作用域链的本质是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当查找一个变量时，如果当前执行环境中没有找到，可以沿着作用域链向后查找。

## 3. 对执行上下文的理解

### 1. 执行上下文类型

#### (1) 全局执行上下文

任何不在函数内部的都是全局执行上下文，它首先会创建一个全局的window对象，并且设置this的值等于这个全局对象，一个程序中只有一个全局执行上下文。

#### (2) 函数执行上下文

当一个函数被调用时，就会为该函数创建一个新的执行上下文，函数的上下文可以有任意多个。

#### (3) eval 函数执行上下文

运行在eval函数中的代码会有属于他自己的执行上下文，不过eval函数不常使用，不做介绍。

### 2. 执行上下文栈

- JavaScript引擎使用执行上下文栈来管理执行上下文
- 当JavaScript执行代码时，首先遇到全局代码，会创建一个全局执行上下文并且压入执行栈中，每当遇到一个函数调用，就会为该函数创建一个新的执行上下文并压入栈顶，引擎会执行位于执行上下文栈顶的函数，当函数执行完成之后，执行上下文从栈中弹出，继续执行下一个上下文。当所有的代码都执行完毕之后，从栈中弹出全局执行上下文。

```
let a = 'Hello world!';
function first() {
  console.log('Inside first function');
  second();
  console.log('Again inside first function');
}
function second() {
  console.log('Inside second function');
}
first();
//执行顺序
//先执行second(),在执行first()
复制代码
```

### 3. 创建执行上下文

创建执行上下文有两个阶段：**创建阶段**和**执行阶段**

#### 1) 创建阶段

##### (1) this绑定

- 在全局执行上下文中，this指向全局对象（window对象）
- 在函数执行上下文中，this指向取决于函数如何调用。如果它被一个引用对象调用，那么this会被设置成那个对象，否则this的值被设置为全局对象或者undefined

##### (2) 创建词法环境组件



- 词法环境是一种有**标识符——变量映射**的数据结构，标识符是指变量/函数名，变量是对实际对象或原始数据的引用。
- 词法环境的内部有两个组件：**加粗样式**：环境记录器:用来储存变量个函数声明的实际位置**外部环境的引用**：可以访问父级作用域

### (3) 创建变量环境组件

- 变量环境也是一个词法环境，其环境记录器持有变量声明语句在执行上下文中创建的绑定关系。

**2) 执行阶段** 此阶段会完成对变量的分配，最后执行完代码。

**简单来说执行上下文就是指：**

在执行一点JS代码之前，需要先解析代码。解析的时候会先创建一个全局执行上下文环境，先把代码中即将执行的变量、函数声明都拿出来，变量先赋值为undefined，函数先声明好可使用。这一步执行完了，才开始正式的执行程序。

在一个函数执行之前，也会创建一个函数执行上下文环境，跟全局执行上下文类似，不过函数执行上下文会多出this、arguments和函数的参数。

- 全局上下文：变量定义，函数声明
- 函数上下文：变量定义，函数声明， `this` , `arguments`

作者：CUGGZ

链接：<https://juejin.cn/post/6940945178899251230>

来源：稀土掘金

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。