一、数据类型

1、JavaScript有哪些数据类型，它们的区别?

> JavaScript一个有8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、symbol、BigInt）
- **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不同：

> 原始类型存储的是值，对象类型存储的是地址（指针）。

- **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，所以放入栈中存储数据
- **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。

2、数据类型检测的方式有哪些?

1、typeof

```
console.log(typeof 3);   //number
console.log(typeof []);  //object
console.log(typeof function(){});  //function
console.log(typeof null);    //object
console.log(typeof 'str');    //string
```

其中数组、对象、null都会被判断为object，其他判断都正确。函数类型会被判断为function

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
```

- `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
```

`constructor`有两个作用，一是判断数据的类型，二是实例对象通过`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
```

4、object.prototype.toString.call()

`Object.prototype.tpString.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)); //[object Number]
console.log(a.call(true)); //[object Boolean]
console.log(a.call('str')); //[object String ]
console.log(a.call([])); //[object Array]
console.log(a.call(function(){})); //[object Function]
console.log(a.call({}));//[object Object]
console.log(a.call(undefined)); //[object undefined]
console.log(a.call(null)); //[object null]
```

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toSting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、判断数组的方法有哪些?

1、通过 object.prototype.toString.call()做判断

```
Object.prototype.toString.call(obj) === 'Array';
```

2、通过原型链判断

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

3、通过ES6的Array.isArray()做判断

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

4、通过Array.prototype.isPrototypeOf(obj)

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

4、null和undefined的区别

- undefined和null都是基本数据类型，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，这意味着可以用undefined作为一个变量名，但是这样做非常危险，它会影响对undefined值的判断。我们可以通过一些方法安全的获得undefined值，比如说void 0.
- 当对这两种类型使用typeof进行判断时，null类型会返回“object”，这是一个历史遗留问题，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、为什么0.1+0.2！=0.3，如何让其相等

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

想要等于0.3，就要把它转化：

```
(n1+n2).toFixed(1)   //注意：toFixed为四舍五入，括号里表示保留几位小数
```

计算机是通过二进制的方式存储数据的，所以计算0.1+0.2是计算两个数的二进制的和。这两个数的二进制都是无限循环的数。

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IEEE754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

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对像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);
    }
}
```

7、如何安全的获取undefined的值?

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来获取undefined的值。

```
let a = void 0;
console.log(a);
```

8、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”，用于指出数字类型中的错误情况，即“执行属性运算没有成功，这是失败后的返回结果“。

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

NaN是一个特殊值，它和自身不相等，是唯一一个非自反的值，而 NaN！=NaN为true。

9、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会首先判断传入的参数是否为数字，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的更加准确。

10、==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，相同的话就会比较大小
2. 类型不相同的话，就会进行类型转换
3. 会先判断是否在对比 `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`
5. 判断其中一方是否为 `Boolean`，是的话就会将 `Boolean` 转换为 `number` 再进行判断
6. 判断其中一方是否为 `object` 且另一方为 `string`、`number` 或者 `symbol`，是的化就会把 `object` 转为原始类型再判断

11、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换后再进行比较
- 使用三等号（===）进行相等判断时，如果两边类型不一致时，不会做强制类型转换，直接返回false
- Object.is()来进行相等判断，一般情况下和三等号的判断相等相同，它处理了一些特殊情况，比如+0和-0不再相等，两个NaN是相等的。

12. 什么是JavaScript中的包装类型?

（1）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'
```

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。

（2）JavaScript也可以使用 `Object` 函数**显示地**将基本类型转换为**包装类型**：

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

（3）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'
```

一个栗子：

看看下面代码会打印出什么：

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

答案是什么也不会打印，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执行。

13、JavaScript中如何进行隐式类型转换?

了解一下 `ToPrimitive` 方法，这是JavaScript中每个隐含的自带方法，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，则是下面这样的：

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

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
- 抛出异常 `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。默认情况如下：

- 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
```

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。

14、隐式转换规则

JavaScript中的隐式转换主要发生在 `+、-、*、/` 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` 转换为基本类型，再进行操作。

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
```

2、`-`、`*`、`\` 操作符：`NaN` 也是一个数字

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

3、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
```

4、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
```

其他情况下，转换为数字再比较

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

> 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化：

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

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
```

又一个栗子：

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

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]"
```

15、+ 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

16、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。

17、如何判断一个对象是空对象?

（1）使用JSON自带的.stringify方法来判断：

```
if(JSON.stringify(Obj)=='{}'){
    console.log('空对象');
}
```

（2）使用ES6新增的方法 Object.keys()来判断：

```
if(Object.keys(Obj).length < 0){
    console.log('空对象');
}
```

二、ES6

1、let、const、var的区别

（1）作用域不同：块级作用域由`{}`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变为全局变量

（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，否则会报错。

（3）给全局添加属性：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，后声明的同名变量会覆盖之前声明的变量。const和let不允许重复声明变量。

（5）暂时性死区：在使用let、const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：在声明变量时，var和let可以不用设置初始值，但是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创建的变量是可以改变指针指向（可以重新赋值），但是const声明的变量是不允许改变指针的指向。

> 总结来说就是：var声明的变量是全局或者整个函数块的，而let和const声明的变量是块级的变量，var声明的变量存在变量提升，let、const不存在；let声明的变量允许重新赋值，const不允许。

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，因此就等同于常量。

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一个箭头函数会怎样?

箭头函数是es6中提出来的，它没有prototype，也没有自己的this指向，更不可以适用arguments参数，所以不能new一个箭头函数。

new操作符的步骤如下：

1. 创建一个对象
2. 将构造函数的作用域赋值给新对象（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、第三步，箭头函数都是没有办法执行的。

```
let b = ()=> console.log('aaa')
b()
new c = ()=> console.log('ccc')
c()
```

4、箭头函数与普通函数的区别

- 箭头函数更加简洁

- **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，他只会在自己作用域的上一层继承this。所以箭头函数中this的指向在它定义时就已经确定了，之后不会改变。
- **继承来的this指向永远不会改变**
- call()、apply()、bind()等方法不能改变箭头函数中this的指向
- 箭头函数不能作为构造函数使用
- 箭头函数没有自己的arguments和prototype
- 箭头函数不能用作Generator函数，不能使用 `yeild` 关键字。

5、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,handlder)
```

`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

6、Symbol

- 不是构造函数，不能通过new创建。而是直接调用Symbol()，返回的变量值永远不变。
- Symbol.for()全局搜索被登记的Symbol中是否有该字符串参数作为名称的Symbol值，如果有即返回该Symbol值，若没有则新建并返回一个以该字符串参数为名称的Symbol值，并登记在全局环境中供搜索。
- Symbol.keyFor（参数为Symbol变量）返回一个已登记的Symbol类型的值的key，用来检测该字符串参数作为名称的Symbol值是否已被登记。。

7、Map和Set

- Map是一个映射数据结构

  API：Get、Set、Delete、Has（键）、Clear、Size、可以forEach遍历

- Set是一个集合数据结构（自带去重效果）

  API：Add、Has、Delete、Clear

8、Promise

定义：

promise是一种异步编程的解决方案，能够将异步操作封装，并将其返回的结果或者错误原因同Promise实例的then方法传入的处理函数联系起来。

New Promise:

传入构造器函数，两个参数，都是敲定函数，第一个成功的敲定，第二个失败的敲定，构造器同步执行，但是可以一步异步的执行敲定函数。

构造器中调用对应的敲定函数，返回的promise实例的对象状态就是那种，当然中间可能会抛出异常throw，那么状态就是拒绝。

状态的变化只有两种pending=> fulfilled,pending => rejected,一旦发生变化就不会再改变。

Promise.then返回的Promise状态:

- 非Promise对象，直接包裹称为Promise（Promise.resolve)
- Promise对象，状态跟随，值跟随，通过这一点可以串联Promise

- 报错返回拒绝的Promise，错误对象是Promise的错误原因

异常穿透：

如果实例.then方法没有处理实例状态对于的回调函数，那么.then返回的promise实例状态跟随调用then方法的promise，Catch在最后进行异常捕获。

中断Promise链：

处理函数返回等待状态的Promise

判断Promise的执行顺序：

- 注意，知道前面的Promise状态发生变化时，才能放进微任务队列
- async返回Promise和Promise.then返回Promise都会导致中间有两个层级的微任务队列间隔
- async每有await延后一个层次发生，如果返回promise再加两个层级

```
new Promise((resolve)=>{
    resolve()
}).then(()=>{

})

Promise.resolve().then(()=>{

})
//这两个Promise的then发生的层级一致
```

9. 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：

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

最终，a，b，c分别被赋予了数组第0,1,2个索引的值。

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

（2）对象的解构：

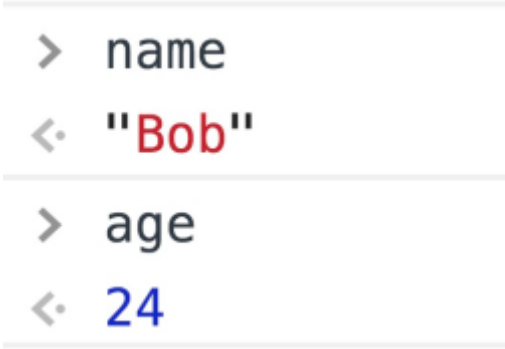
在解构对象时，是以属性的名称为匹配条件，来提取想要的数据。

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

假如想要解构他的两个属性，可以这样：

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

这样就得到了name和age两个和stu平级的变量



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;
```

10、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}
```

还可以用来修改对象的属性。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}
```

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换为用以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
```

> 数组的扩展运算符的应用：

（1）将数组转换为参数序列：

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

（2）复制数组

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

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。

（3）合并数组

在数组内合并数组：

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

（4）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 //1
rest  // [2,3,4,5]
```

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
```

（5）将字符串转为真正的数组

```
[...'hello'];  //['h','e','l','l','o']
```

（6）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换为数组：

```
//将arguments对象转换为数组
function foo(){
    const args = [...arguments];
}
```

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

（7）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
```

11、ES6中模板语法与字符串处理

ES6模板语法

> ``和 ${}

```
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]}`
```

- 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

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`、`startWith`、`endsWith`，他们都会返回一个布尔值来告诉你是否存在。

(1) 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
```

(2) startWith：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
```

(3) 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
```

(4) 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;
```

三、JS基础

1. new操作符的执行过程

1. 首先创建一个新的空对象
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。

2. for in、for of 、forEach、map

- `map`：数组方法，不改变元数组，返回新数组，可以使用break中断，可以return到外层函数
- `forEach`：数组方法，不可以使用break中断，不可以return到外层函数
- `for in`：用于遍历数组和对象，遍历对象键值（key），或者数组下标，不推荐循环一个数组
- `for of`：不能遍历对象，需要通过和Object.keys()搭配使用

  `for of`遍历列表值，允许遍历Arrays（数组），String（字符串），Maps（映射），Sets（集合）等可迭代的数据结构等。

  在ES6中引入的`for of`循环，以替代`for in`和`forEach()`，并支持新的迭代协议
- `for in`循环出的是key，`for of`循环出的是value

3. 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。

（1）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)
}
```

（2）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 conut = 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);
}
```

3.5、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（对象、数组）等并且返回各项的值。

- for of 遍历获取的是对象的键值value，for in 获取的是对象的键名key
- 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 主要用于遍历对象而生，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以及生成器对象等可迭代的对象

4、Map和Object的区别

	Map	Object
意外的键	Map默认情况不包含任何键，只包含显示插入的键	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 Iterator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	在频繁添加或删除键值对的场景下未作出优化

5、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中的键值对中键只能是string或symbol类型。**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":19]
]
```

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结构原生提供是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其

> - 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
})
```

(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 **set(key,value) | get(key) | has(key) | delete(key)** 方法。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：
>
> 有时想在某个对象上面存放一些数据，但是这会形成对这个对象的引用。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，不用手动清除引用。**

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。

6、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有用户的脚本创建或宿主程序提供。

标准内置对象的分类：值属性、函数属性、基本对象、数字和日期对象、字符串，用来表示和操作字符串的对象、可索引的集合对象、使用键的集合对象、矢量集合、结构化数据、控制抽象对象、反射、国际化、WebAssembly、其他。

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。

7、常用的正则表达式

```
//(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}-(0[1-9]|1[0-2])-(0[0-9]|[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}$/g;
//(5) 用户名正则
var regex = /^[a-zA-Z\$][a-zA-Z0-9_\$]{4-16}$/;
```

8、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。在前端通过一个符合JSON格式的数据序列化为JSON字符串，然后将它传递给后端，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，但是注意JSON和js中的对象并不是一回事，JSON中对象格式更加严格，比如说在JSON中属性值不能作为函数，不能出现NaN这样的属性值，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()`：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，是其符合规范。在前后端发送数据时，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()`：这个函数用来将JSON格式的字符串转换为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，将会抛出错误。

9、如何让JavaScript脚本延迟加载?

> 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。

- **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加载执行。

10、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，代表可以接收的参数个数。

将类数组转换为数组的方法：

（1）通过call调用数组的slice方法来实现转换：不会改变原数组

> 关于为什么Array.prototype.slice可以使用call方法，是这样的：先把arraylike转换为数组后（使用call），才能使用数组的slice方法 。详细解释请看

```
arraylike = {
    0:'33',
    1:'232',
    length:2
}
var arr = Array.prototype.slice.call(arraylike);
console.log(arraylike);
console.log(arr);
```

（2）通过call调用数组的splice方法：会改变原数组

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

（3）通过apply调用数组的concat方法

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

（4）通过Array.from方法

```
Array.from(arraylike);
```

11、数组的原生方法

- 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()移除`和`unshift()添加`，返回长度。
- 数组重排序方法`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`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

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开始依次递增的数组，还有`callee`和`length`等属性，与数组类似；但是它没有数组常见的方法属性，如`forEach`、`reduce`等，所以叫他们类数组。

要想遍历类数组，可以先将它转化为数组。

- 使用 call 和apply
- 使用Array,from方法
- 使用展开运算符将类数组转化为数组

13、什么是DOM和BOM?

- **DOM**指的是文档对象类型，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，他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、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，无论在函数中任何位置声明的变量，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的作用域链中去查找，而作用域链的首段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，这个对象都是在代码解析的时候创建的。

变量提升主要有以下两个原因：

- 提高性能
- 容错性更好

变量提升可能会造成的问题：

- 内层定义的变量可能会覆盖外层变量
- 如果是在块级作用域里定义var，那么运行完毕后不会被销毁。

15、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，代码执行时基于执行栈的，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，然后再新创建另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。

16、ES6模块与CommonJS模块有什么异同?

CommonJS模块

ES6模块化

ES6 Module 和 CommonJS模块的区别：

- CommonJS是对模块的浅拷贝，ES6 Module是对模块的应用，即ES6模块只存只读，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ES6模块赋值会编译报错。

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。

17、常见的DOM操作

- 获取：getElementById、querySelectorAll、等
- 创建：createElement、appendChild
- 删除：removeChild
- 修改：insertBefore

18、数组扁平化

参考文章

> 数组扁平化是指将一个多维数组变为一维数组。
>
> 例：[1,[2,3,[4,5]]] ----> [1,2,3,4,5]

（1）通过concat将二维数组转化为一维数组

原理：通过将扩展运算符，将数组内部展开，并通过concat连接两个字符串的方式返回数组。

```
let a = [12,3,45,[6,7,8]];
let b = [].concat(...a);
console.log(b);  //[12, 3, 45, 6, 7, 8]
```

（2）使用数组方法join 和 字符串方法 split 进行数组扁平化

原理：通过join方法将数组转化为以点隔开的字符串，再使用split把转化的字符串转化为数组。通过map方法将内部字符串转化为数字类型的。

```
let a = [1,2,[3,4,[5,6,7,8,[9,10]]]];
console.log(a.join(',')); //1,2,3,4,5,6,7,8,9,10
console.log(a.join(',').split(','));
/*[
  '1',  '2', '3',
  '4',  '5', '6',
  '7',  '8', '9',
  '10'
]*/
let b = a.join(',').split(',').map(Number);
console.log(b); //[1,2,3,4,5,6,7,8,9,10]
```

（3）通过正则方法和JSON.stringify方法和数组方法

原理：首先将数组转化为字符串，使用字符串匹配正则规则，替换所有的'[' ']' 和方法类似。

split主要是将字符串转化为数组，map将字符串数组转换为数字。

```
let a = [1,2,[3,4,[5,6,7,8,[9,10]]]];
let c = JSON.stringify(a).replace(/\[|\]/g,'').split(',').map(Number);
console.log(c)//[1,2,3,4,5,6,7,8,9,10]
```

（4）函数递归

原理：判断获取的当前值是不是数组，是数组就递归调用

```
let a = [1,2,[3,4,[5,6,7,8,[9,10]]]];
let d = [];
function fn(arr){
    for(let i=0;i<arr.length;i++){
        if(Array.isArray(arr[i])){
            fn(arr[i]);
        }else{
            d.push(arr[i]);
        }
    }
}
fn(a);
console.log(d);  //[1,2,3,4,5,6,7,8,9,10]
```

（5）通过reduce方法进行数组扁平化

原理：主要通过reduce的依次进行，判断当前拿到的是不是数组，是数组就进行递归将内部所有数组扁平化（与方法四类似）

```
let a = [1,2,[3,4,[5,6,7,8,[9,10]]]];
function flatten(arr){
    return arr.reduce((result,item)=>{
       console.log(result,item);
       return result.concat(Array.isArray(item) ? flatten(item):item);
    },[])
};
console.log(flatten(a));  //[1,2,3,4,5,6,7,8,9,10]
```

（6）ES6新增方法flat()

```
let a = [4, 1, 2, 3, 6, [7, 8, [3, 9, 10, [4, 6, 11]]]];
let e = a.flat()   // 不传参的时候 表示将二维数组转一维数组
console.log(e)     // [4, 1, 2, 3, 6, 7, 8, Array(4)]
let f = a.flat(2)  // 传入2 表示将两层嵌套数组 转化为一维数组
console.log(f)     // [4, 1, 2, 3, 6, 7, 8, 3, 9, 10, Array(3)]
let g = a.flat(Infinity) // Infinity 使用这个关键字可以将所包含的所谓数组转化为一维数组
console.log(g)     // [4, 1, 2, 3, 6, 7, 8, 3, 9, 10, 4, 6, 11]
```

19、use strict是什么意思？使用它区别是什么？

use strict 是一种ES5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。

- 提供编译器的效率，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

20、如何判断一个对象是否属于某个类?

1. 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。
2. 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
3. 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。

21、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。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，那么它就永远是那个类型。例java和c++等语言。
- **弱类型语言**：弱类型语言也被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。就是一种变量的类型可以被忽略的语言。

22、ajax、axios、fetch的区别

（1）ajax

ajax即（异步JavaScript和XML）。是一种在无需加载整个网页的情况下，能够更新部分网页的技术。

缺点：

- 本身是针对MVC编程的，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('http://aaa.com/text.json')
.then(response => response.json())
.then(data => console.log(data));
```

fetch的优点：

- 语法简洁，更加语义化
- 基于标准的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没有方法原生检测请求的进度，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

23、数组的遍历方法有哪些?

> 细数JavaScript中那些遍历和循环

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

四、原型和原型链

> - 构造函数原型prototype
> - 对象原型 __ proto __
> - constructor构造函数

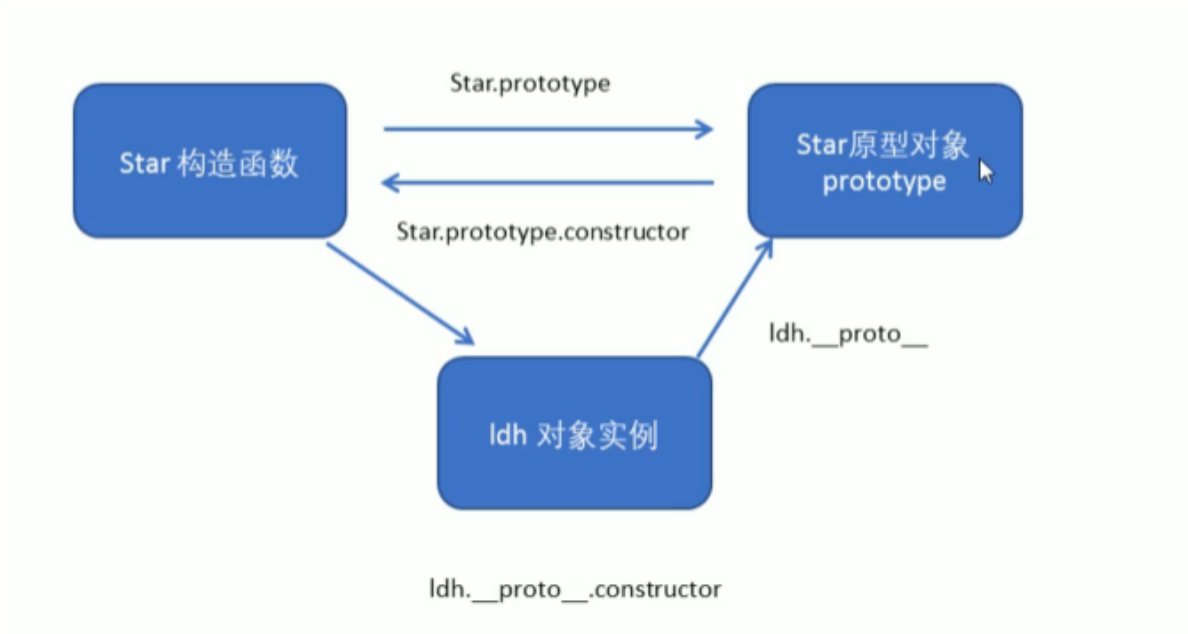
原型：

要弄清楚什么是原型和原型链，就首先要知道什么叫**构造函数**：

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，即prototype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。



> 构造函数的prototype===对象实例的__ proto __

关于原型修改、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
```

可以看到**重写原型**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，它的构造函数的指向了**根构造函数Object**，所以这时候 `p.constructor === Object`，而不是 `p.constructor === Person`。想要成立，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
```

原型链：

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。

原型链就是多个对象通过 `__ proto __` 的方式连接了起来。

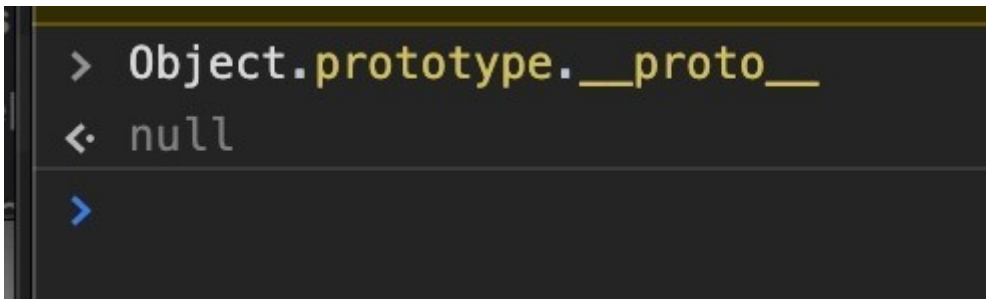
原型链指向

```
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
```

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__`



如何获得对象非原型链上的属性？

使用 `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;
}
```

五、执行上下文/作用域链/闭包

1. 闭包

闭包是指有权访问另一个函数作用域中变量的 `函数` 。创建闭包最常见的方式就是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。

闭包的两个常用用途：

1. 使我们可以在函数外部能够访问函数内部的变量。通过在外部调用闭包，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。
2. 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。

也就是函数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
```

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)
}
```

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 `i` 就是6了，所以会输出一堆6。解决的方法有三种：

- **第一种就是使用闭包的方式**

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

在上述代码中，首先使用了立即实行函数将 `i` 传入函数内部，这个时候值就被固定在了 `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)
}
```

- 第三种就是使用 `let` 定义 `i` 来解决问题，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)
}
```

2. 作用域、作用域链

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

> 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到外层作用域，反之不行。

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（就是一定要在使用前声明），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。**

作用域链的本质上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. 执行上下文

深入理解执行上下文

> 执行上下文会创建变量对象。
>
> 执行上下文可以理解为当前代码的运行环境

执行上下文的类型

（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

执行在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？？？

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()
```

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。如果他被一个引用对象调用，那么this会被设置为那个对象，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**：用来存储变量给函数声明的实际位置 **外部环境引用**：可以访问父级作用域。

（3）创建环境变量组件

- 变量环境也是一个词法环境，器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

六、事件循环、微任务和宏任务

> 事件循环（Event Loop），宏任务（macro-task），微任务（micro-task），执行栈和任务队列等。

前言

JavaScript是单线程的，同一时间只能做一件事。如果碰到某个耗时长的任务，那么后续的任务都要等待，这种效果是无法接受的，这时就引入了**异步任务**的概念。

所以JavaScript执行主要包括同步任务和异步任务：

同步任务：会放到执行栈中，他们是要按顺序执行的任务；

异步任务：会放到任务队列中，这些异步任务一定要等到执行栈清空后才会执行，也就是说异步任务一定是在同步任务之后执行的。

> JavaScript事件循环机制，主要和异步任务有关。

任务队列

事件循环主要和 任务队列有关，所以必须先知道宏任务和微任务。

在任务队列中，有两种任务：宏任务和微任务。

宏任务：script标签中的整体代码、setTimeout、setInterval、setImmediate、I/O、UI渲染等

微任务：process.nextTick(Node.js)、Promise.then()、Object.observe(不常用)、MutationObserver（NodeJS）

JS中用来存储执行回调函数的队列包含2个不同特定的队列

宏队列：用来保存待执行的宏任务（回调），比如：定时器回调/DOM事件回调/ajax回调

微队列：用来保存待执行的微任务（回调），比如：promise的回调/MutationsObserver的回调

JS执行时会区别这2个队列

1. JS引擎首先必须先执行所有的初始化同步任务代码
2. 每次准备取出第一个宏任务执行前，都要将所有的微任务一个一个取出来执行

> 执行的顺序是 同步代码》微任务》宏任务
>
> 微任务永远是先执行的，如果微任务中有宏任务，那么将这个宏任务放到宏队列中进行排队等待。

事件循环

- 一开始整个脚本（script标签中的整体代码）作为一个宏任务执行；
- 执行过程中同步代码直接执行，宏任务进入宏队列，微任务进入微队列；
- 当前宏任务（同步代码）执行完毕后，立即执行当前微任务队列中的所有微任务（依次执行）
- 当前宏任务执行完毕，开始检查渲染，然后GUI线程接管渲染
- 渲染完毕后，JS线程继续接管，开始下一个宏任务（从事件队列中获取）

> 简单来说，事件循环大致分为以下几个步骤：

1. 所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）
2. 主线程之外，还存在一个 “任务队列”（task queue）。只要异步任务有了运行结果，就在“任务队列”之中放置一个事件。
3. 一旦 “执行栈”中的所有同步任务执行完毕，系统就会读取“任务队列”，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
4. 主线程不断重复上面的第三步。

七、深拷贝和浅拷贝

> 这里深浅拷贝针对的是**引用数据类型**。因为原始数据类型，从一个变量拷贝到另一个变量，只是很单纯的赋值过程，两者是不会有什么影响的。
>
> 而对于引用数据类型，从一个变量拷贝到另一个变量，本质是拷贝了对象的存储地址，两个变量最终都还是指向同一个对象。

概念

为什么存在 `浅拷贝` 和 `深拷贝`？

“为了防止父对象数据被篡改”。也就是不希望我拷贝完的对象修改后会影响到原来的对象。

- 浅拷贝：复制引用的拷贝方法（可以解决一层，但是多层就不行了）
- 深拷贝：完全拷贝一个对象。完成拷贝之后彼此之间操作绝对不会有互相影响的就是深拷贝。（可以解决多层）

数组的浅拷贝和深拷贝

浅拷贝

- `Array.prototype.slice()`
  - 运行slice得到的结果是一个对原数组的浅拷贝，原数组不会修改。所以如果修改两个数组中的任意一个，另一个都不会受影响
- `Array.prototype.concat()`
  - concat方法是用来合并数组的，它也不会更改原有的数组，而是返回一个新数组。
- `ES6扩展语法...`
- `Array.from()`

> 上面这些方法对于**一层 深度** 且只是**普通的原始数据类型值**来说，确实不会改变原数组。但是如果是多层拷贝的化，就不行了：

```
// 数组
let arr = [1, 2, [1]];
let newArr = arr.concat();
newArr[2][0] = 10;
console.log(arr, newArr); // [1, 2, [10]] [1, 2, [10]]

let arr1 = [1, 2, [1]];
let newArr1 = arr1.slice();
newArr1[2][0] = 10;
console.log(arr1, newArr1); // [1, 2, [10]] [1, 2, [10]]
```

这就需要深拷贝了：

深拷贝

数组深拷贝：`JSON.Parse()`，`JSON.stringify()`

```
let arr = [1, 2, [1]];
let newArr = JSON.parse(JSON.stringify(arr));
newArr[2][0] = 10;
console.log(arr, newArr); // [1, 2, [1]] [1, 2, [10]]
```

对象的浅拷贝和深拷贝

浅拷贝

- `Object.assign(target,...sources)`
- 也可以通过扩展运算符 `...` 来实现

```
const person = {
   name: 'Wes Bos',
   age: 80
 };
const cap2 = Object.assign({}, person, { number: 99, age: 12 });
console.log(cap2); // Object {name: "Wes Bos", age: 12, number: 99}
```

浅拷贝只解决了第一层的问题，如果接下去的值中还有对象的话，那么就又回到最开始的话题了，两者享有相同的地址。要解决这个问题，我们就得使用深拷贝了。

深拷贝

`JSON.Parse()`, `JSON.stringify()`

JSON对象中的stringify可以把一个js对象序列化为一个JSON字符串，parse可以把JSON字符串反序列化为一个对象。

```
const wes = {
  name: 'Wes',
  age: 100,
  social: {
    twitter: '@wesbos',
    facebook: 'wesbos.developer'
  }
};

const dev = Object.assign({}, wes);
const dev2 = JSON.parse(JSON.stringify(wes));
console.log(wes);
console.log(dev);
console.log(dev2);
```

不过 **这种方法也是有局限的**:

- 会忽略 `undefined`
- 会忽略 `symbol`
- 不能序列化函数
- 不能解决循环引用的对象

loader中使用 deepclone可以解决

八、防抖和节流

防抖（debounce）

> 多次点击只执行一次。
>
> 只执行最后一个被触发的，清除之前的异步任务，核心在于 `清零`

所谓防抖，就是指触发事件后，把触发非常频繁的事件合并成一次去执行。即在指定时间内只执行一次回调函数，如果在指定的时间内又触发了该事件，则回调函数的执行时间会基于此刻重新开始计算。

简单来说就是，执行一次函数后（给下一次执行设定一个延迟时间），如果再次触发，那么不会响应，而这个时间重新计时，直到在这个时间内没有触发事件才可以执行下一次函数。

```
const button = document.querySelector('input');

function inputhhh(){
    console.log('hhh');
}

//防抖函数
//传入两个参数：要防抖的函数，要延迟的时间
function debounce(func,delay){
    let timer;
    return function(){
        let context = this;
        let args = arguments;
        clearTimeout(timer);
        timer = setTimeout(function(){
            func.apply(context,args);
        },delay);
    }
}

button.addEventListener('click',debounce(inputhhh,1000));
```

要点：`setTimeout` | `clearTimeout`

节流（throttle）

> 在固定时间段内只执行一次。
>
> 只在开始执行一次，未执行完成过程中触发的忽略，核心在于`开关锁`

所谓节流，是指频繁触发事件时，只会在指定的时间段内执行事件回调，即触发事件间隔大于等于指定的时间才会执行回调函数。

简单来说，就是当事件被触发时执行，但是在指定时间内不管触发多少次都不会执行，当指定时间结束，触发才会被执行。

【使用定时器】

```
function throttle(func,delay){
    let timer;
    return function(){
        let context = this;
        let args = arguments;
        if(timer){
            return;
        }
        timer = setTimeout(function(){
            func.apply(context,args);
            timer = null;
        },delay);
    }
}
```

【使用Date()时间戳】

```
function throttle(func,delay){
    let pre = 0;
    return function(){
        let now = new Date();
        let context = this;
        let args = arguments;
        if(now - pre > delay){
            func.apply(context,args);
            pre = now;
        }
    }
}
```

总结

1. 防抖和节流只是减少了事件回调函数的执行次数，并不会减少事件的触发频率。
2. 防抖和节流并没有从本质上解决性能问题，我们还应该注意优化我们事件回调函数的逻辑功能，避免在回调中执行比较复杂的DOM操作，减少浏览器回流和重绘。

九、this/call/apply/bind

1. 对this的理解

> this关键字是函数运行时自动生成的一个内部对象，只能在函数内部使用，总指向调用它的对象。

this是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，this的指向可以通过四种调用模式来判断。

1. **函数调用模式（默认绑定）**：当一个函数不是一个对象的属性时，直接作为函数来调用时，this指向全局对象。
2. **方法调用模式（隐式绑定）**：如果一个函数作为一个对象的方法来调用时，this指向这个对象（这个函数如果被多级对象包含，尽管这个函数是被最外层的对象所调用，this的指向也只是它上一级对象）。例子
3. **构造器模式（new绑定）**：如果一个函数用new调用时，函数执行前会创建一个实例对象，this指向这个新创建的实例对象。
4. **apply、call、bind调用模式（显示绑定）**：这三个方法都可以显示的指定调用函数的this指向。

> 这四种调用的优先级：
>
> 构造器模式 》apply|call|bind 》方法调用模式》函数调用模式。

2. apply、call、bind的理解

作用

call、apply、bind作用是改变函数执行时的上下文，简而言之就是改变函数运行时的`this`指向。

apply

`apply`接受两个参数，第一个参数是`this`指向，第二个参数是函数接收的参数，以**数组**的形式传入。

改变this指向后原函数会立即执行，且此方法只是**临时改变this指向一次。**

```
function fn(...args){
  console.log(this,args);
}
let obj = {
  myname:"张三"
}

fn.apply(obj,[1,2]); // this会变成传入的obj，传入的参数必须是一个数组；
fn(1,2) // this指向window
```

> 当第一个参数为 null、undefined的时候，默认指向window（在浏览器中）

call

call方法的第一个参数是this的指向，后面传入的是一个参数列表

跟 `apply` 一样，改变this指向后原函数会立即执行，且此方法只是临时改变this指向一次。

```
function fn(...args){
    console.log(this,args);
}
let obj = {
    myname:"张三"
}

fn.call(obj,1,2); // this会变成传入的obj，传入的参数是一个列表；
fn(1,2) // this指向window
```

> 当第一个参数为 null、undefined的时候，默认指向window（在浏览器中）

bind

bind和call很类似，第一个参数也是 this 指向，后面传入的也是一个参数列表（但是这个参数列表可以分多次传入）

改变this指向后不会立即执行，而是返回一个永久改变 `this` 指向的函数。

```
function fn(...args){
    console.log(this,args);
}
let obj = {
    myname:"张三"
}

const bindFn = fn.bind(obj); // this 也会变成传入的obj ，bind不是立即执行,故需要执行一次
bindFn(1,2) // this指向obj
fn(1,2) // this指向window
```

总结

- 三者都可以改变函数的 `this` 对象指向
- 三者第一个参数都是 `this` 要指向的对象，如果没有这个参数或参数为 `undefined` 或 `null`，则默认指向全局 `window`
- 三者都可以传参，但是 `apply` 是数组，而 `call` 是参数列表，且 `apply` 和 `call` 是一次性传入参数，而 `bind` 可以分为多次传入
- `bind` 是返回绑定 `this` 之后的函数， `apply`、 `call` 则是立即执行

3. 手撕call、apply、bind函数

（1）实现call

1. 判断调用对象是否为函数，即使是定义在函数原型上的，**但是可能出现使用call等方式调用的情况**
2. 判断传入上下文对象是否存在，如果不存在，设置为window
3. 处理传入的参数，截取第一个参数后的所有参数
4. 将函数作为上下文对象的一个属性
5. 使用上下文对象来调用这个方法，并保存返回结果
6. 删除刚才新增的属性
7. 返回结果

```
Function.prototype.myCall = function(context){
    //判断调用对象
    if(typeof this!=="function"){
        console.log("type error");
    }
    //获取参数
    let args = [...arguments].slice(1);
    let result = null;
    //判断context 是否传入。如果未传入则设置为window
    context = context || window;
    //将调用函数设为对象的方法
    context.fn = this;
    //调用函数
    result = context.fn(...args);
    //将属性删除
    delete context.fn;
    return result;
};
```

（2）实现apply

1. 判断调用对象是否为函数，即使是定义在函数的原型上的，但是可能出现使用call等方式调用的情况。
2. 判断传入上下文对象是否存在，如果不存在，则设置为window
3. 将函数作为上下文对象的一个属性
4. 判断参数值是否传入
5. 使用上下文对象来调用这个方法，并保存返回结果
6. 删除刚才新增的属性
7. 返回结果

```
Function.prototype.myApply = function(context){
    if(typeof this!=="function"){
```

```
        throw new TypeError("Error");
    }
    let result = null;
    context = context || window;
    context.fn = this;
    if(arguments[1]){
        result = context.fn(...arguments[1]);
    }else{
        result = context.fn();
    }
    delete context.fn;
    return result;
};
```

(3) 实现bind

1. 判断调用对象是否为函数，即使是定义在函数的原型上的，但是可能出现使用call等方式调用的情况。
2. 保存当前函数的引用，获取其传入的参数值
3. 创建一个函数返回
4. 函数内部使用apply来绑定函数调用，需要判断函数作为构造函数的情况，这个时候需要传入当前函数的this给apply调用，其余情况都传入指定的上下文对象。

```
Function.prototype.myBind = function(context){
    //判断调用对象是否为函数
    if(typeof this!=="function"){
        throw new TypeError("Error");
    }
    //获取参数
    var args = [...arguments].slice(1);
    fn = this;
    return function Fn(){
        //根据调用方式，传入不同绑定值
        return fn.apply(
            this instanceof Fn ? this : context,args.concat(...arguments)
        );
    };
};
```

十、异步编程

1. 异步解决方案

JavaScript中的异步机制可以分为以下几种：

- **回调函数**的方式，使用回调函数当有多个回调函数嵌套的时候会造成回调地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。
- **Promise**的方式，使用Promise的方式可以将嵌套的回调函数作为链式调用。但是，有时会造成多个then的链式调用，可能会造成代码的语义不够明确。
- **generator生成器**，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在generator内部对于异步操作的方式，可以以同步的顺序来书写。
- **async函数**，async是generator和promise实现的一个自动执行的语法糖。它内部自带执行器，当函数内部执行得到一个await语句的时候，如果语句返回一个promise对象，那么函数将会等待

promise对象的状态变为resolve后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

2. async/await（异步/等待）

> async/await其实就是 generator的语法糖，他能实现的效果都能用then链来实现，它是为优化then链而开发出来的。
>
> async/await的用处就是：用同步的方式，执行异步操作。理解

- `async`是一个位于function之前的前缀，**只有在`async函数`中，才能使用`await`。**
- 在 async函数中，`await`规定了异步操作只能一个一个排队执行，从而达到用同步的方式，执行异步的操作。

也就是说，`async`函数返回一个Promise对象，当函数执行的时候，一旦遇到await就会先返回，等到触发的异步操作完成，再执行函数体后面的语句。可以理解为，是让出线程，跳出了async函数体。

`await`的含义为等待，也就是async函数需要等待await后的函数执行完成并且有了返回结果（Promise对象）之后，才能继续执行下面的代码。await通过返回一个Promise对象来实现同步的效果。

【一个栗子】

```
function request(num){ //模拟接口请求
    return new Promise(resolve=>{
        setTimeout(()=>{
            resolve(num*2)
        },1000)
    })
}

async function fn(){
    await request(1);
    await request(2);
}
fn()
```

【一个栗子】

```
async function async1(){
   console.log('async1 start');
    await async2();
    console.log('async1 end')
}
async function async2(){
    console.log('async2')
}
console.log('script start');
async1();
console.log('script end')
// 输出顺序：script start->async1 start->async2->script end->async1 end
```

【总结】

- await只能在async函数中使用，不然会报错
- async函数返回的是一个Promise对象，有无值看有无return值
- await后面最好是接Promise，虽然接其他值也能达到排队的效果

- async/await作用是 **用同步方式，执行异步操作**

async/await如何捕获异常

```
async function fn(){
    try{
        let a = await Promise.reject('error')
    }catch(error){
        console.log(error)
    }
}
```

3. 生成器Generator

> 是ES6里面的知识。

迭代器Iterator

Iterator是一种接口，它为不同的数据结构提供统一的访问机制。任何数据结构只要部署了Iterator接口，就可以完成遍历操作。

Iterator可以认为是一个 **指针** 对象，提供 `next()` 对数据结构进行遍历，每次调用 `next()` 方法，指针就指向数组的下一个成员并返回数据结构的当前成员的信息。该信息是一个对象，包含 `value` 和 `done` 两个属性。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

Iterator接口部署在数据结构的Symbol.iterator属性

可迭代的数据结构：String、Array、Map、Set、类数组、DOM（Nodelist对象）

目的就是为了迭代有序的数据结构，方便访问和使用 **for...of**

理解

- `generator函数` 跟普通函数在写法上的区别就是，多了一个星号 `*`
- 在generator函数中才能使用 `yield`，相当于是函数指执行的 中途暂停点。
- 使用 `next方法` 继续执行，next方法执行后回返回一个对象，对象中有 `value和done` 两个属性
  - **value**：暂停点后面接的值，也就是yield后面接的值
  - **done**：是否generator函数已经走完，没走完为false，走完为true

```
function* gen() {
  yield 1
  yield 2
  return 4
}
const g = gen()
console.log(g.next()) // { value: 1, done: false }
console.log(g.next()) // { value: 2, done: false }
console.log(g.next()) // { value: 4, done: true }
```

最后一个value值是否为undefined取决于函数有无返回值。

关于yield

yield表达式就是用来 **划分generator函数的各个状态，他可以理解为函数暂停的标志。**

当执行`next()`方法，遇到yield表达式时，就暂停后面的操作，并将 **yield表达式的值作为next方法返回的信息对象的value属性值。**

下次调用next方法，继续执行yield表达式后面的操作。

4. 回调函数

理解

什么是回调函数?

按照MDN：回调函数是作为参数传递给另一个函数的函数，然后在外部函数内部调用该回调函数以完成某种例程或操作。

简单来说，回调函数是一个函数，将在另一个函数完成执行后立即执行。回调函数是一个作为参数传递给另一个JavaScript函数的函数。该回调函数在传递给它的函数内部执行。

为什么需要回调?

防止在单线程中执行某任务时间较长，而造成线程阻塞。

```
ajax(url,()=>{
    //处理逻辑
})
```

回调地狱

```
ajax(url, () => {
    // 处理逻辑
    ajax(url1, () => {
        // 处理逻辑
        ajax(url2, () => {
            // 处理逻辑
        })
    })
})
```

5. Promise

promise解析

什么是promise

Promise是异步编程的一种解决方案，它是一个对象，可以获取异步操作的消息。

所谓Promise，简单来说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。

- 从语法上说，Promise是一个构造函数
- 从功能上说：Promise是对象用来封装一个异步操作并可以获取其成功/失败的结果值。

> - promise简单理解就是回调
> - 成功回调一个函数，失败也回调一个函数，然后我们就可以在成功或失败里面写入一些代码去执行一些想要做的事。

Promise的三个状态

- Pending （进行中/未决定的）
- Resolved | fullfilled（已完成/成功）
- Rejected （已拒绝/失败）

promise只有两种状态，初始状态是pending，且一个promise对象只能改变一次。无论结果是成功还是失败，都会有一个**结果数据**，成功的结果一般称为 value，失败的结果数据一般称为 reason。

【promise使用的一个小例子】

```
const p = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        if(n <= 30){
            resolve(n);
        }else{
            reject(n);  //n传参
        }
     },1000);
});
//调用then方法
//第一个回调函数是成功的回调，第二个回调是失败的回调
//value 值； reason 理由
p.then((value)=> {  //成功resolve的执行函数
      alert("中奖了");
},(reason)=>{
      alert("没中奖失败了");
});
});
```

> 在使用promise时，构造函数内部的代码是立即执行的。

Promise常用API

① Promise.prototype.then()

指定用于得到成功value的成功回调和用于得到失败reason的失败回调**返回一个新的Promise对象。**第一个参数是resolved状态的回调函数，第二个参数是rejected状态的回调函数。

② Promise.prototype.catch()

用于指定发生错误时的回调函数

catch与then的第二个参数的区别是如果在then的第一个函数里抛出了异常，后面的catch能捕获到，而第二个函数捕获不到。

③ Promise.resolve()

返回一个成功或失败的promise对象。

- 如果传入的参数为 非promise对象，则返回的结果为成功的promise对象
- 如果传入的参数为 Promise对象 ， 则参数的结果决定了resolve 的结果

④ Promise.reject()

不管传入什么都会返回失败。

⑤ Promise.all()

多个promise任务并行执行

如果全部成功执行，则以数组的方式返回所有的Promise任务的执行结果。

如果有一个Promise任务是Rejected，则只返回rejected任务的结果。

⑥ Promise.race()

多个Promise任务并行执行。返回最先执行结束的Promise任务的结果，不管这个Promise结果是成功还是失败。

Promise的缺点

- 无法取消，一旦新建就会立即执行
- 如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
- 当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

十一、面向对象

面向对象理解

1. 对象创建的方式有哪些?

一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象时，会产生大量的重复代码。但JS和一般的面向对象的语言不用，在ES6之前它没有类的概念。但是可以使用函数来进行模拟，从未产生出可复用的对象创建方式，常见的：

① 工厂模式

主要的工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，**它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。**

```
function createPerson(name,age,job){
    var o = new Object;
    o.name = name;
    o.age = age;
    o.job = job;
    o.sayName = function(){
        alert(this.name);
    };
    return o;
}
var person1 = createPerson("jasonzhang",18,"singer");
var person2 = createPerson("yutongzhou",18,"actor");
```

② 构造函数模式

JS中每一个函数都可以作为构造函数，只要一个函数通过`new`来调用的，那么就可以把它称为构造函数。**执行构造函数首先会创建一个实例对象，然后对象的原型指向构造函数的prototype属性，然后将执行上下文中的this指向这个实例对象，然后再执行整个函数，如果返回值不是对象，则返回新建的对象。**因为this的值指向了新建的对象，因此可以使用this给对象赋值。

构造函数模式相对于工厂模式的**优点**是，所创建的对象和构造函数建立起了联系，因此可以通过原型来识别对象的类型。但是存在一个 **缺点**：造成了不必要的函数对象的创建，因为在js中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

```
function Person(name,age,job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = function(){
        alert(this.name);
    };
}
var person1 = new Person("jasonzhang",18,"singer");
var person2 = new Person("yutongzhou",18,"actor");
```

③ 原型模式

因为每一个函数都有一个prototype属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。

但是这个模式也存在一些问题，一个是没有办法通过出传入参数来初始化值，另一个是如果存在一个引用类型如Array这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。

```
function Person(){}
Person.prototype.name = "jasonzhang";
Person.prototype.age = 18;
Person.prototype.job = "singer";
Person.prototype.sayName = function(){
    alert(this.name);
};
var person1 = new Person();
person1.sayName();   //"jasonzhang"
var person2 = new Person();
person2.sayName();   //"jasonzhang"
```

④ 组合使用构造函数模式和原型模式

这是创建自定义类型最常见的方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此可以组合使用这两种模式，**通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。**

这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。

```
function Person(name,age,job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.friends = ["sandy","gally"];
}
Person.prototype = {
    constructor : Person,
    sayName : function(){
        alert(this.name);
    }
}
```

```
var person1 = new Person("jasonzhang",20,"singer");
var person2 = new Person("mary",13,"student");
```

⑤ 动态原型模式

把所有信息都封装在了构造函数中，而通过在构造函数中初始化原型（仅在必要的情况下），又保持了同时使用构造函数和原型的优点。简单的说，就是可以通过检查某个应该存在的方法是否有效，来决定是否需要初始化原型。

```
function Person(name,age,job){
    //属性
    this.name = name;
    this.age = age;
    this.job = job;
    //方法
    if(typeof this.sayName != "function"){
        Person.prototype.sayName = function(){
            alert(this.name);
        };
    }
}
var friend = new Person("jason",20,"singer");
friend.sayName();
```

⑥ 寄生构造函数模式

这一模式和工厂模式的实现基本相同，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。他的缺点和工厂模式一样，无法实现对象的识别。

2. 对象的继承方法

> 原型链作为实现继承的主要方法。

【继承的基本思想】：利用原型让一个引用类型继承另一个引用类型的属性和方法。

【原型链的基本概念】：假如让原型对象等于另一个类型的实例，此时的原型对象将会包含一个指向另一个原型的指针，相应的，另一个原型中也包含着一个指向另一个构造函数的指针。假如另一个原型又是另一个类型的实例，那么上述关系依然成立，如此层层递进，就构成了实例与原型的链条。

① 原型链的方式

这个方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改混乱。还有就是在创建子类型的时候不能向超类型传递参数。

② 借用构造函数的方式

这种方式是用过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有方法访问到。

③ 组合继承

组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

④ 原型式继承

型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象，然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，ES5 中定义的 Object.create() 方法就是原型式继承的实现。缺点与原型链方式相同。

⑤ 寄生式继承

寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是自定义类型时。缺点就是没有办法实现函数复用。

⑥ 寄生式组合继承

组合继承的缺点就是使用超类型的实例作为子类型的原型，导致添加了不必要的原型。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。