2024前端高频— JS篇

1.js基本数据类型有哪些及它们的区别

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、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着分配释放，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

```
javascript
复制代码console.log(typeof 2);
console.log(typeof true);
console.log(typeof 'str');
console.log(typeof []);
console.log(typeof function(){});
console.log(typeof {});
console.log(typeof undefined);
console.log(typeof null);
```

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**。

```
javascript
复制代码console.log(2 instanceof Number);
console.log(true instanceof Boolean);
console.log('str' instanceof String);

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

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

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

`constructor` 有两个作用，一是判断数据的类型，二是对象实例通过 `constrcutor` 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：

```
javascript
复制代码function Fn(){}

Fn.prototype = new Array()

var f = new Fn()

console.log(f.constructor===Fn)
console.log(f.constructor===Array)
```

（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：

```
javascript
复制代码var a = Object.prototype.toString;
```

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

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

```
javascript
复制代码Object.prototype.toString.call(obj).slice(8,-1) === 'Ar
ray';
```

- 通过原型链做判断

```
javascript
复制代码obj.__proto__ === Array.prototype
```

- 通过ES6的Array.isArray()做判断

```
javascript
复制代码Array.isArrray(obj)
```

- 通过instanceof做判断

```
javascript
复制代码obj instanceof Array
```

- 通过Array.prototype.isPrototypeOf

```
javascript
复制代码Array.prototype.isPrototypeOf(obj)
```

4.请简述JavaScript中的this

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。

- 第一种是**函数调用模式**，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。
- 第二种是**方法调用模式**，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。
- 第三种是**构造器调用模式**，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。
- 第四种是 **apply 、 call 和 bind 调用模式**，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

这四种方式，使用构造器调用模式的优先级最高，然后是 apply、call 和 bind 调用模式，然后是方法调用模式，然后是函数调用模式。

3.箭头函数与普通函数有什么区别

（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：

```
javascript
复制代码let fn = () => void doesNotReturn();
```

（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

```
javascript
复制代码var id = 'GLOBAL';
var obj = {
  id: 'OBJ',
  a: function(){
    console.log(this.id);
  },
  b: () => {
    console.log(this.id);
  }
};
obj.a();
```

```
obj.b();
new obj.a()
new obj.b()
```

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

```
javascript
复制代码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'
```

（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.AMD和CommonJS的区别

它们都是实现模块体系的方式，直到 `ES2015` 出现之前，`JavaScript` 一直没有模块体系。`CommonJS` 是同步的，而 `AMD (Asynchronous Module Definition)` 从全称中可以明显看出是异步的。`CommonJS` 的设计是为服务器端开发考虑的，而 `AMD` 支持异步加载模块，更适合浏览器。

我发现 `AMD` 的语法非常冗长，`CommonJS` 更接近其他语言 `import` 声明语句的用法习惯。大多数情况下，我认为 `AMD` 没有使用的必要，因为如果把所有 `JavaScript` 都捆绑进一个文件中，将无法得到异步加载的好处。此外，`CommonJS` 语法上更接近 `Node` 编写模块的风格，在前后端都使用 `JavaScript` 开发之间进行切换时，语境的切换开销较小。

我很高兴看到 `ES2015` 的模块加载方案同时支持同步和异步，我们终于可以只使用一种方案了。虽然它尚未在浏览器和 `Node` 中完全推出，但是我们可以使用代码转换工具进行转换。

6.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。

7\. 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	✓	✓	×

8.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;

  if (typeof constructor !== "function") {
    console.error("type error");
    return;
  }

  newObject = Object.create(constructor.prototype);

  result = constructor.apply(newObject, arguments);

  let flag = result && (typeof result === "object" || typeof
result === "function");

  return flag ? result : newObject;
}

objectFactory(构造函数, 初始化参数);
```

9.数组有哪些原生方法？

数组和字符串的转换方法：toString()、toLocalString()、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.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。

11.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，值是基本类型, 改变不了;如果是引用类型分两种情况：1、没有修改形参元素的地址值, 只是修改形参元素内部的某些属性，会改变原数组；2、直接修改整个元素对象时，无法改变原数组，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

12.forEach和map的区别

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；

13.原型和原型链

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。

2\. 原型修改、重写

```
javascript
复制代码function Person(name) {
    this.name = name
}

Person.prototype.getName = function() {}
var p = new Person('hello')
console.log(p.__proto__ === Person.prototype)
console.log(p.__proto__ === p.constructor.prototype)

Person.prototype = {
```

```
    getName: function() {}
}
var p = new Person('hello')
console.log(p.__proto__ === Person.prototype)
console.log(p.__proto__ === p.constructor.prototype)
```

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：

```
javascript
复制代码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
```

3\. 原型链指向

```
javascript
复制代码p.__proto__
Person.prototype.__proto__
p.__proto__.__proto__
p.__proto__.constructor.prototype.__proto__
Person.prototype.constructor.prototype.__proto__
p1.__proto__.constructor
Person.prototype.constructor
```

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：

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

14\. 对执行上下文,作用域(链),闭包的理解

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。

```
javascript
复制代码function A() {
  let a = 1
  window.B = function () {
      console.log(a)
  }
}
A()
B()
```

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

```
javascript
复制代码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

```
javascript
复制代码for (var i = 1; i <= 5; i++) {  ;(function(j) {    setT
imeout(function timer() {      console.log(j)    }, j * 1000)
})(i)}
```

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。

```
javascript
复制代码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` 了来解决问题了，这个也是最为推荐的方式

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

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零，一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\. 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

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。

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。

```
javascript
复制代码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()
```

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`

15\. 实现call、apply 及 bind 函数

（1）call 函数的实现步骤：

- 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。
- 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
- 处理传入的参数，截取第一个参数后的所有参数。
- 将函数作为上下文对象的一个属性。
- 使用上下文对象来调用这个方法，并保存返回结果。
- 删除刚才新增的属性。

- 返回结果。

```
Function.prototype.myCall = function(context) {

  if (typeof this !== "function") {
    console.error("type error");
  }

  let args = [...arguments].slice(1),
    result = null;

  context = context || window;

  context.fn = this;

  result = context.fn(...args);

  delete context.fn;
  return result;
};
```

（2）apply 函数的实现步骤：

- 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。
- 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
- 将函数作为上下文对象的一个属性。
- 判断参数值是否传入
- 使用上下文对象来调用这个方法，并保存返回结果。
- 删除刚才新增的属性
- 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）bind 函数的实现步骤：

- 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。
- 保存当前函数的引用，获取其余传入参数值。
- 创建一个函数返回
- 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) {

  if (typeof this !== "function") {
```

```
    throw new TypeError("Error");
  }

  var args = [...arguments].slice(1),
    fn = this;
  return function Fn() {

    return fn.apply(
      this instanceof Fn ? this : context,
      args.concat(...arguments)
    );
  };
};
```

16\. call、apply 函数的区别

它们的作用一模一样，区别仅在于传入参数的形式的不同。

- apply 接受两个参数，第一个参数指定了函数体内 this 对象的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，apply 方法把这个集合中的元素作为参数传递给被调用的函数。
- call 传入的参数数量不固定，跟 apply 相同的是，第一个参数也是代表函数体内的 this 指向，从第二个参数开始往后，每个参数被依次传入函数。

17\. 异步编程的实现方式？

JavaScript中的异步机制可以分为以下几种：

- **回调函数** 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，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。但是使用这种方法，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** 的方式，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。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

- **async 函数** 的方式，async 函数是 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。

18\. setTimeout、Promise、Async/Await 的区别

（1）setTimeout

```
javascript
复制代码console.log('script start')
setTimeout(function(){
    console.log('settimeout')
})
console.log('script end')
```

（2）Promise

Promise本身是**同步的立即执行函数**， 当在executor中执行resolve或者reject的时候, 此时是异步操作， 会先执行then/catch等，当主栈完成后，才会去调用resolve/reject中存放的方法执行，打印p的时候，是打印的返回结果，一个Promise实例。

```
javascript
复制代码console.log('script start')
let promise1 = new Promise(function (resolve) {
    console.log('promise1')
```

```
    resolve()
    console.log('promise1 end')
}).then(function () {
    console.log('promise2')
})
setTimeout(function(){
    console.log('settimeout')
})
console.log('script end')
```

当JS主线程执行到Promise对象时：

- promise1.then() 的回调就是一个 task
- promise1 是 resolved或rejected: 那这个 task 就会放入当前事件循环回合的 microtask queue
- promise1 是 pending: 这个 task 就会放入 事件循环的未来的某个(可能下一个)回合的 microtask queue 中
- setTimeout 的回调也是个 task ，它会被放入 macrotask queue 即使是 0ms 的情况

（3）async/await

```
javascript
复制代码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')
```

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，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 async 函数体。

例如：

```
javascript
复制代码async function func1() {
    return 1
}
console.log(func1())
```

func1的运行结果其实就是一个Promise对象。因此也可以使用then来处理后续逻辑。

```
javascript
复制代码func1().then(res => {
    console.log(res);
})
```

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。

19\. Promise.all和Promise.race的区别的使用场景

（1）Promise.all `Promise.all` 可以将多个 `Promise` 实例包装成一个新的Promise实例。同时，成功和失败的返回值是不同的，成功的时候返回的是**一个结果数组**，而失败的时候则返回**最先被reject失败状态的值**。

Promise.all中传入的是数组，返回的也是是数组，并且会将进行映射，传入的promise对象返回的值是按照顺序在数组中排列的，但是注意的是他们执行的顺序并不是按照顺序的，除非可迭代对象为空。

需要注意，Promise.all获得的成功结果的数组里面的数据顺序和Promise.all接收到的数组顺序是一致的，这样当遇到发送多个请求并根据请求顺序获取和使用数据的场景，就可以使用Promise.all来解决。

（2）Promise.race

顾名思义，Promse.race就是赛跑的意思，意思就是说，Promise.race(\[p1, p2, p3\])里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。当要做一件事，超过多长时间就不做了，可以用这个方法来解决：

```
javascript
复制代码Promise.race([promise1,timeOutPromise(5000)]).then(res=
>{})
```

20\. 对async/await 的理解

async/await其实是`Generator`的语法糖，它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。从字面上来看，async是“异步”的简写，await则为等待，所以很好理解async 用于申明一个 function 是异步的，而 await 用于等待一个异步方法执行完成。当然语法上强制规定await只能出现在asnyc函数中，先来看看async函数返回了什么：

```
async function testAsy(){
   return 'hello world';
}
let result = testAsy();
console.log(result)
```

所以，async 函数返回的是一个 Promise 对象。async 函数（包含函数语句、函数表达式、Lambda表达式）会返回一个 Promise 对象，如果在函数中`return`一个直接量，async 会把这个直接量通过`Promise.resolve()`封装成 Promise 对象。

async 函数返回的是一个 Promise 对象，所以在最外层不能用 await 获取其返回值的情况下，当然应该用原来的方式：`then()`链来处理这个 Promise 对象，就像这样：

```
async function testAsy(){
   return 'hello world'
}
let result = testAsy()
console.log(result)
result.then(v=>{
    console.log(v)
})
```

那如果 async 函数没有返回值，又该如何？很容易想到，它会返回 `Promise.resolve(undefined)`。

联想一下 Promise 的特点——无等待，所以在没有 `await` 的情况下执行 async 函数，它会立即执行，返回一个 Promise 对象，并且，绝不会阻塞后面的语句。这和普通返回 Promise 对象的函数并无二致。

注意： `Promise.resolve(x)` 可以看作是 `new Promise(resolve => resolve(x))` 的简写，可以用于快速封装字面量对象或其他对象，将其封装成 Promise 实例。

21\. 浏览器的垃圾回收机制

```
####  （1）垃圾回收的概念

**垃圾回收**：JavaScript代码运行时，需要分配内存空间来储存变量和值。当变量不在参与运行时，就需要系统收回被占用的内存空间，这就是垃圾回收。

**回收机制**：

-    Javascript 具有自动垃圾回收机制，会定期对那些不再使用的变量、对象所占用的内存进行释放，原理就是找到不再使用的变量，然后释放掉其占用的内存。
-    JavaScript中存在两种变量：局部变量和全局变量。全局变量的生命周期会持续要页面卸载；而局部变量声明在函数中，它的生命周期从函数执行开始，直到函数执行结束，在这个过程中，局部变量会在堆或栈中存储它们的值，当函数执行结束后，这些局部变量不再被使用，它们所占有的空间就会被释放。
-    不过，当局部变量被外部函数使用时，其中一种情况就是闭包，在函数执行结
```

束后，函数外部的变量依然指向函数内部的局部变量，此时局部变量依然在被使用，所以不会回收。

（2）垃圾回收的方式

浏览器通常使用的垃圾回收方法有两种：标记清除，引用计数。 **1）标记清除**

- 标记清除是浏览器常见的垃圾回收方式，当变量进入执行环境时，就标记这个变量“进入环境”，被标记为“进入环境”的变量是不能被回收的，因为他们正在被使用。当变量离开环境时，就会被标记为“离开环境”，被标记为“离开环境”的变量会被内存释放。
- 垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后。垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。

2）引用计数

- 另外一种垃圾回收机制就是引用计数，这个用的相对较少。引用计数就是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减1。当这个引用次数变为0时，说明这个变量已经没有价值，因此，在在机回收期下次再运行时，这个变量所占有的内存空间就会被释放出来。
- 这种方法会引起**循环引用**的问题：例如：`obj1`和`obj2`通过属性进行相互引用，两个对象的引用次数都是2。当使用循环计数时，由于函数执行完后，两个对象都离开作用域，函数执行结束，`obj1`和`obj2`还将会继续存在，因此它们的引用次数永远不会是0，就会引起循环引用。

```
function fun() {
let obj1 = {};
let obj2 = {};
obj1.a = obj2; // obj1 引用 obj2
```

obj2.a = obj1; // obj2 引用 obj1
}

这种情况下，就要手动释放变量占用的内存：

obj1.a = null
obj2.a = null

（3）减少垃圾回收

虽然浏览器可以进行垃圾自动回收，但是当代码比较复杂时，垃圾回收所带来的代价比较大，所以应该尽量减少垃圾回收。

- **对数组进行优化：** 在清空一个数组时，最简单的方法就是给其赋值为[]，但是与此同时会创建一个新的空对象，可以将数组的长度设置为0，以此来达到清空数组的目的。
- **对**`object`**进行优化：** 对象尽量复用，对于不再使用的对象，就将其设置为null，尽快被回收。
- **对函数进行优化：** 在循环中的函数表达式，如果可以复用，尽量放在函数的外面。

22\. 哪些情况会导致内存泄漏

以下四种情况会造成内存的泄漏：

- **意外的全局变量：** 由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。
- **被遗忘的计时器或回调函数：** 设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。
- **脱离 DOM 的引用：** 获取一个 DOM 元素的引用，而后面这个元素被删

除，由于一直保留了对这个元素的引用，所以它也无法被回收。
- **闭包：** 不合理的使用闭包，从而导致某些变量一直被留在内存当中。

23\. ES6有哪些新特性

1.箭头函数

2.解构赋值

3.模板字符串

4.promise

5.symbol Symbol是ES6中引入的一种新的基本数据类型,用于表示一个独一无二的值,不能与其他数据类型进行运算

6.新的变量声明方式-let和const

7.模块化-es6新增了模块化，根据功能封装模块，通过import导入，然后通过export导出也可以使用export default导出

8.for...of 循环,用于遍历可迭代对象(如数组、Map 和 Set)中的元素

9.扩展运算符:使用 ... 可以将数组或对象展开成多个参数,或者将多个参数合并成一个数组

10.展开运算符:在ES6中用...来表示展开运算符,它可以将数组或者对象进行展开

11.Map 和 Set，引入了两种新的数据结构，分别用于存储键值对和唯一值

12.Proxy，允许在对象和函数调用等操作前后添加自定义的行为

13.类（Class），引入了面向对象编程中类的概念

14.默认参数（Default Parameter），在定义函数时可以给参数设置默认值

24\. 匿名函数的典型应用场景是什么？

匿名函数可以在 IIFE 中使用，来封装局部作用域内的代码，以便其声明的变量不会暴露到全局作用域。

```
(function () {

})();
```

匿名函数可以作为只用一次，不需要在其他地方使用的回调函数。当处理函数在调用它们的程序内部被定义时，代码具有更好地自闭性和可读性，可以省去寻找该处理函数的函数体位置的麻烦。

```
setTimeout(function () {
  console.log('Hello world!');
}, 1000);
```

匿名函数可以用于函数式编程或 Lodash（类似于回调函数）。

```
const arr = [1, 2, 3];
const double = arr.map(function (el) {
  return el * 2;
});
console.log(double);
```

25.你能举出一个柯里化函数（curry function）的例子吗？它有哪些好处？

柯里化（currying）是一种模式，其中具有多个参数的函数被分解为多个函数，当被串联调用时，将一次一个地累积所有需要的参数。这种技术帮助编写函数式风格的代码，使代码更易读、紧凑。值得注意的是，对于需要被 curry 的函数，它需要从一个函数开始，然后分解成一系列函数，每个函数都需要一个参数。

```
function curry(fn) {
  if (fn.length === 0) {
    return fn;
  }

  function _curried(depth, args) {
    return function (newArgument) {
      if (depth - 1 === 0) {
        return fn(...args, newArgument);
      }
```

```
      return _curried(depth - 1, [...args, newArgument]);
    };
  }

  return _curried(fn.length, []);
}

function add(a, b) {
  return a + b;
}

var curriedAdd = curry(add);
var addFive = curriedAdd(5);

var result = [0, 1, 2, 3, 4, 5].map(addFive);
```

26.什么是事件循环？调用堆栈和任务队列之间有什么区别？

事件循环是一个单线程循环，用于监视调用堆栈并检查是否有工作即将在任务队列中完成。如果调用堆栈为空并且任务队列中有回调函数，则将回调函数出队并推送到调用堆栈中执行。

27.js设计模式有哪些？

总体来说设计模式分为三大类：(`C5S7B11`)

1. **创建型模式**，共五种：**工厂方法模式**、抽象工厂模式、**单例模式**、建造者模式、**原型模式**。
2. **结构型模式**，共七种：**适配器模式**、**装饰器模式**、**代理模式**、外观模式、桥接模式、组合模式、享元模式。
3. **行为型模式**，共十一种：策略模式、模板方法模式、**观察者模式/发布订阅模式**、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

手写单例模式（创建模式）

```
javascript
复制代码let CreateSingleton = (function(){
    let instance;
    return function(name) {
        if (instance) {
            return instance;
        }
        this.name = name;
        return instance = this;
    }
})();
CreateSingleton.prototype.getName = function() {
    console.log(this.name);
}
```

代码测试

```
javascript
复制代码let Winner = new CreateSingleton('Winner')
let Looser = new CreateSingleton('Looser')

console.log(Winner === Looser)
console.log(Winner.getName())
console.log(Looser.getName())
```

手写观察者模式（行为模式）

```
javascript
复制代码// 定义observe
const queuedObservers = new Set()
const observe = fn => queuedObservers.add(fn)
```

```
const observable = obj => new Proxy(obj, {
  set(target, key, value, receiver) {
    const result = Reflect.set(target, key, value, receiver)
    // notify
    queuedObservers.forEach(observer => observer())
    return result
  }
})
```

代码测试

```
javascript
复制代码
obj = observable({
  name:'789'
})

observe(function test(){
  console.log('触发了')
})

obj.name ="前端柒八九"
```

手写发布订阅 （行为模式）

```
javascript
复制代码class Observer {
  caches = {};
```

```
  on (eventName, fn){
    this.caches[eventName] = this.caches[eventName] || [];
    this.caches[eventName].push(fn);
  }


  emit (eventName, data) {
    if (this.caches[eventName]) {
      this.caches[eventName]
      .forEach(fn => fn(data));
    }
  }

  off (eventName, fn) {
    if (this.caches[eventName]) {
      const newCaches = fn
        ? this.caches[eventName].filter(e => e !== fn)
        : [];
      this.caches[eventName] = newCaches;
    }
  }

}
```

代码测试

```
javascript
复制代码ob = new Observer();

l1 = (data) => console.log(`l1_${data}`)
l2 = (data) => console.log(`l2_${data}`)

ob.on('event1',l1)
ob.on('event1',l2)
```

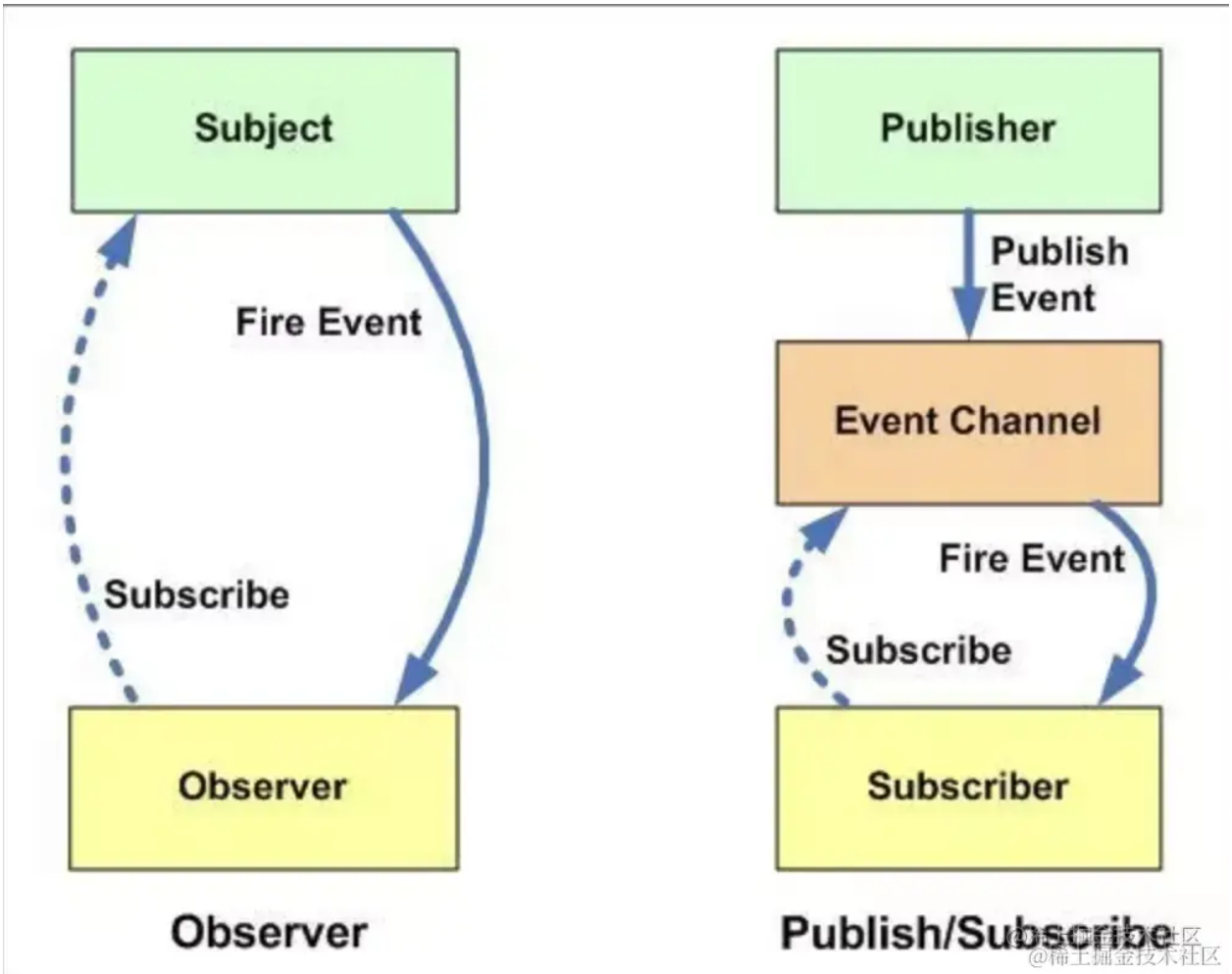
```
ob.emit('event1',789)



ob.off('event1',l1)

ob.emit('event1',567)
```

观察者模式 VS 发布订阅模式



1. 从表面上看：
    - 观察者模式里，只有两个角色 —— **观察者 \+ 被观察者**
    - 而发布订阅模式里，却不仅仅只有发布者和订阅者两个角色，还有一个经常被我们忽略的 —— {经纪人|Broker}
2. 往更深层次讲：
    - 观察者和被观察者，是`松耦合`的关系
    - 发布者和订阅者，则完全不存在耦合
3. 从使用层面上讲：
    - 观察者模式，多用于**单个应用内部**
    - 发布订阅模式，则更多的是一种{跨应用的模式|cross-application pattern} ，比如我们常用的消息中间件