


# 挑战一轮大厂后的面试总结 (含六个方向) - javascript 篇(万字长文)

这篇文章是对 `javascript` 相关的题目做总结，内容有点长，大致算了下，有接近 2W 字，推荐用电脑阅读，欢迎朋友们先收藏在看。

先看看目录（这长图在手机上比较模糊，可点击图片看大图）

javascript.md >  ### Q:箭头函数有没有 arguments 对象?

> ### Q:介绍下原型链 ...

> ### Q:介绍下构造函数是什么? ...

> ### Q:typeof 和 instanceof 有什么区别 ...

> ### Q:数据类型有哪几种? ...

> ### Q:JS中基本数据类型和引用类型在内存上有什么区别? ...

> ### Q:描述 NaN 指的是什么 ...

> ### Q:描述 null ...

> ### Q:什么是包装对象 ...

> ### Q:class 和 function 的区别 ...

> ### Q:实现继承的几种方法 ...

> ### Q:谈谈作用域链机制 ...

> ### Q:let var const 的区别 ...

> ### Q:数据属性和访问器属性的区别 ...

> ### Q:toString 和 valueOf 有什么区别 ...

> ### Q:箭头函数有没有 arguments 对象? ...

> ### Q:js 精度丢失问题 ...

> ### Q: toFixed 可以做到四舍五入吗 ...

> ### Q: js中不同进制怎么转换 ...

> ### Q:对js处理二进制有了解吗 ...

> ### Q:异步有哪些解决方案 ...

> ### Q:简单介绍Generator ...

> ### Q: 讲一讲 Promise ...

> ### Q: co库的执行原理 ...

> ### Q:介绍下浏览器的事件循环 ...

> ### Q:介绍下模块化方案 ...

> ### Q:垃圾回收机制 ...

> ### Q:什么是严格模式 ...

> ### Q:map 和 weakMap 的区别 ...

> ### Q:String 和 Array 有哪些常用函数 ...

Q:介绍下原型链

> ### Q:判断数组的几种方法 ...

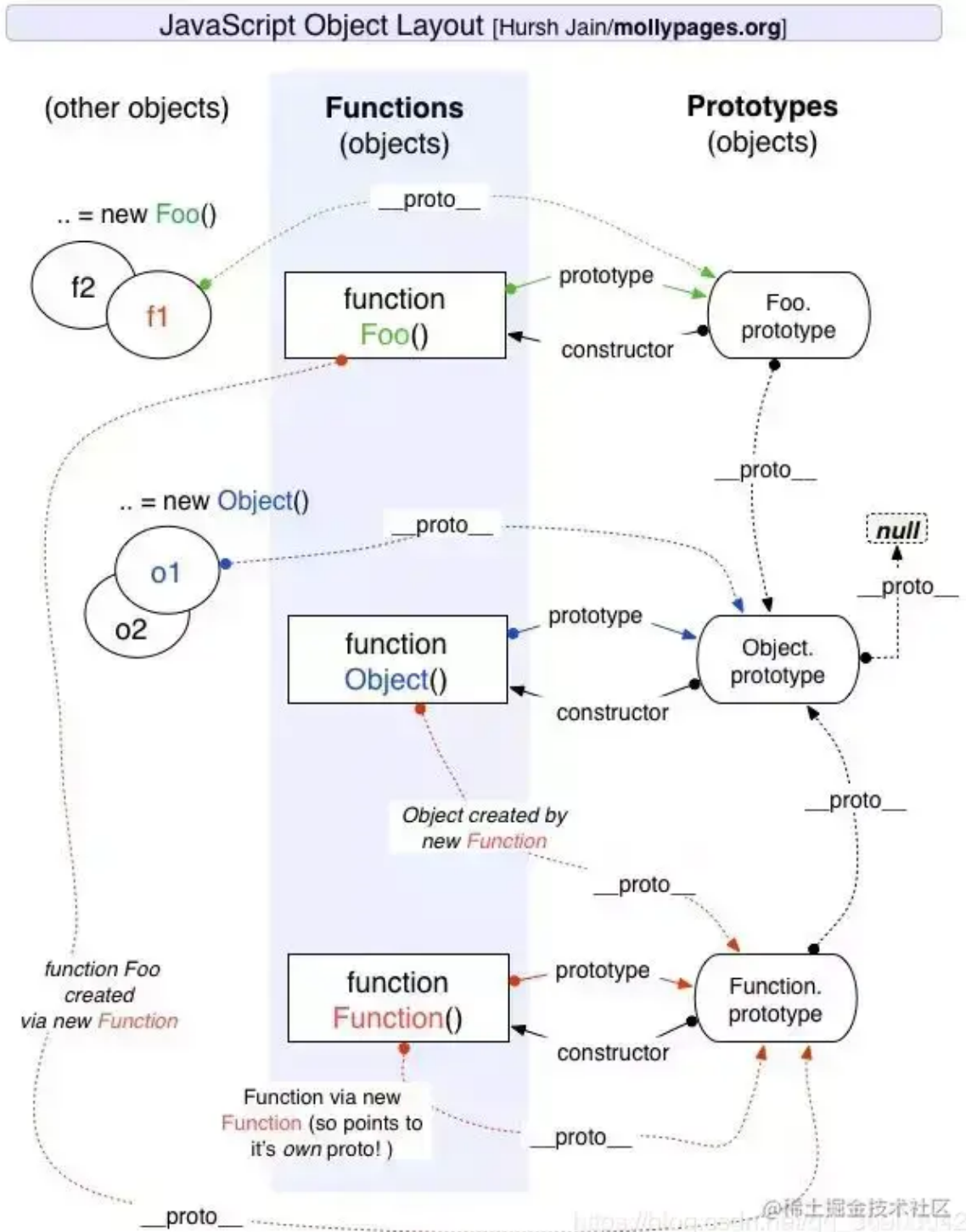
> ### Q:循环有几种方式，是否支持中断和默认情况下是否支持async/await ...

原型链这东西，基本上是无脑必记，而且不是知识点还都是基于原型链扩展的，所以我们先把

> ### Q:闭包的使用场景列举 ...

原先链整明白。我们看一张网上非常流行的图

> ### Q:扩展运算符 ...

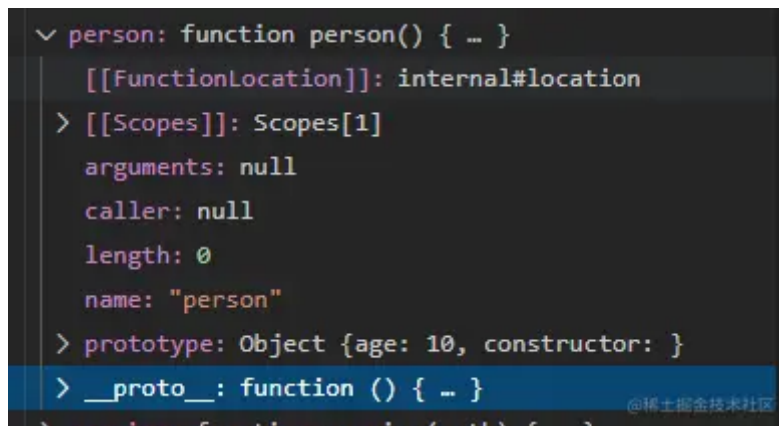


嗯，箭头有点多且有点绕，没关系，我们可逐步分析，我们从结果倒推结论，这样更直观些，看代码

```
function person() {  
  this.name = 10  
}  
person.prototype.age = 10  
const p = new person()
```

## 分析构造函数

我们通过断点看下 person 这个函数的内容



```
person: function person() { ... }  
  [[FunctionLocation]]: internal#location  
> [[Scopes]]: Scopes[1]  
  arguments: null  
  caller: null  
  length: 0  
  name: "person"  
> prototype: Object {age: 10, constructor: }  
> __proto__: function () { ... }
```

它是一个自定义的函数类型，看关键的两个属性 `prototype` 和 `__proto__`，我们——分析

### 1. prototype 分析

对 `prototype` 展开看，是个自定义的对象，这个对象有三个属性 `age` `constructor` `__proto__`，`age` 的值是 10，那么可以得出通过 `person.prototype` 赋值的参数都是在 `prototype` 这个对象中的。

点开 `constructor`，发现这个属性的值就是指向构造器 `person` 函数，其实就是循环引用，这时候就有点套娃的意思了

```

    ✓ person: function person() { ... }
      [[FunctionLocation]]: internal#location
    > [[Scopes]]: Scopes[1]
      arguments: null
      caller: null
      length: 0
      name: "person"
    ✓ prototype: Object {age: 10, constructor: }
      age: 10
    ✓ constructor: function person() { ... }
      [[FunctionLocation]]: internal#location
    > [[Scopes]]: Scopes[1]
      arguments: null
      caller: null
      length: 0
      name: "person"
    > prototype: Object {age: 10, constructor: }
    > __proto__: function () { ... }
    > __proto__: Object {constructor: , __defineGett...
    > __proto__: function () { ... }

```

那么，根据字面意思， `prototype` 可以翻译成，原先对象，用于扩展属性和方法。

## 2. `__proto__` 分析 对 `__proto__` 展开看看

```

    > person: function person() { ... }
      [[FunctionLocation]]: internal#location
    > [[Scopes]]: Scopes[1]
      arguments: null
      caller: null
      length: 0
      name: "person"
    > prototype: Object {age: 10, constructor: }
    > __proto__: function () { ... }
      [[FunctionLocation]]: internal#location
    > [[Scopes]]: Scopes[0]
    > apply: function apply() { ... }
      arguments: TypeError: 'caller', 'callee', and ...
    > bind: function bind() { ... }
    > call: function call() { ... }
      caller: TypeError: 'caller', 'callee', and 'ar...
    > constructor: function Function() { ... }
      length: 0
      name: ""
    > Symbol(Symbol.hasInstance): function [Symbol.h...
    > toString: function toString() { ... }
    > __proto__: Object {constructor: , __defineGett...
      > __defineGetter__: function __defineGetter__()-
      > __defineSetter__: function __defineSetter__()-
      > __lookupGetter__: function __lookupGetter__()-
      > __lookupSetter__: function __lookupSetter__()-
      > constructor: function Object() { ... }
      > hasOwnProperty: function hasOwnProperty() { ... }
      > isPrototypeOf: function isPrototypeOf() { ... }
      > propertyIsEnumerable: function propertyIsEnum...
      > toLocaleString: function toLocaleString() { ... }
      > toString: function toString() { ... }
      > valueOf: function valueOf() { ... }
      __proto__: null

```

person 中的 `__proto__` 是一个原始的 function 对象，在 function 对象中，又看到了 `__proto__` 这个属性，这时候它的值是原始的 Object 对象，在 Object 对象中又再次发现了 `__proto__` 属性，这时候 `__proto__` 等于 null

js 中数据类型分为两种，基本类型和对象类型，所以我们可以这么猜测，person 是一个自定义的函数类型，它应该是属于函数这一家族下的，对于函数，我们知道它是属于对象的，那么它们几个是怎么关联起来的呢？

没错，就是通过 `__proto__` 这个属性，而由这个属性组成的链，就叫做原型链。



根据上面的例子我们，可得出，原型链的最顶端是 `null`，往下是 `Object` 对象，而且只要是对象或函数类型都会有 `__proto__` 这个属性，毕竟大家都是 js-family 的一员嘛。

## 分析生成的对象

上面我们已经知道了原型和原型链，那么对于 `new` 出来的对象，它们的关系又是怎样的呢？继续断点分析

```

  ▾ p: person {name: 10}
    name: 10
    ▾ __proto__: Object {age: 10, constructor: }
      age: 10
      > constructor: function person() { ... }
    ▾ __proto__: Object {constructor: , __defineGett...
      > __defineGetter__: function __defineGetter__()...
      > __defineSetter__: function __defineSetter__()...
      > __lookupGetter__: function __lookupGetter__()...
      > __lookupSetter__: function __lookupSetter__()...
      > constructor: function Object() { ... }
      > hasOwnProperty: function hasOwnProperty() { .....
      > isPrototypeOf: function isPrototypeOf() { ... }
      > propertyIsEnumerable: function propertyIsEnum...
      > toLocaleString: function toLocaleString() { .....
      > toString: function toString() { ... }
      > valueOf: function valueOf() { ... }
      __proto__: null
    
```

`p` 对象中有个 `__proto__` 属性，我们已经知道这是个原型链，通过它可以找到我们的祖先，展开 `__proto__`，大家看到这里有没有发现很眼熟，在看一张图，

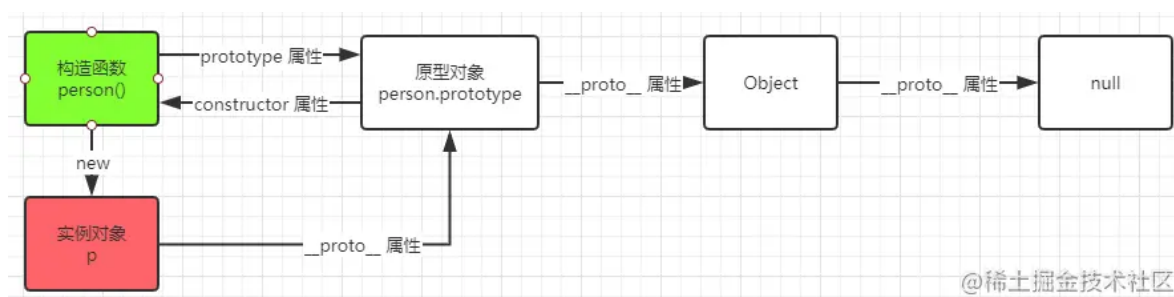
```

  p: person {name: 10}
    name: 10
    __proto__: Object {age: 10, constructor: }
      age: 10
      > constructor: function person() { ... }
      > __proto__: Object {constructor: , __defineGett...
    person: function person() { ... }
      [[FunctionLocation]]: internal#location
      > [[Scopes]]: Scopes[1]
      arguments: null
      caller: null
      length: 0
      name: "person"
    prototype: Object {age: 10, constructor: }
      age: 10
      > constructor: function person() { ... }
      > __proto__: Object {constructor: , __defineGett...
      > __proto__: function () { ... }

```

没错！`p.__proto__` 就是 `person` 函数的 `prototype`，这一步也就是 `new` 的核心点(下个题目我们会说到)。

那么 `p` 这实例的原型链是怎么样的？  
`p.__proto__ => {constructor:func}.__proto__ => Object => null`



对于实例对象来说，原型链主要用来做什么呢？

- 实现继承：如果没有原型链，每个对象就都是孤立的，对象间就没有关联，所以原型链就像一颗树干，从而可以实现面对对象中的继承
- 属性查找：首先在当前实例对象上查找，要是没找到，那么沿着 `__proto__` 往上查找
- 实例类型判断：判断这个实例是否属于某类对象

还有就是，光看文字的解释还是有点费解的，要想深入理解，还是需要多动手断点调试，才能很快理顺。



若还是不太理解实例对象的原型链关系，可以看下一题：解释构造函数

## Q:介绍下构造函数是什么？

构造函数与普通函数在编码上没有区别，只要可以通过 `new` 来调用的就是构造函数。

那么什么函数不可以作为构造函数呢？

箭头函数不可以作为构造函数。

`new` 是一个语法糖，对执行的原理一步步拆分并自己写一个模拟 `new` 的函数: 0. 自定义一个 `objectFactory` 模拟 `new` 语法糖，函数可以接受多个参数，但要求第一个参数必须为构造函数

1. 创建一个空对象 `obj`，分配内存空间
2. 从参数列表中获取构造函数，并将 `obj` 的 `__proto__` 属性指向构造函数的 `prototype`
3. 通过 `apply` 执行构造，并将当前 `this` 的指向改为 `obj`
4. 返回构造函数的执行结果，或者当前的 `obj` 对象

ini 复制代码

```
function objectFactory() {
  var obj = {},
  Constructor = [].shift.call(arguments);
  obj.__proto__ = Constructor.prototype;
  var ret = Constructor.apply(obj, arguments);
  return typeof ret === 'object' ? ret : obj;
};
function fnf() {
  this.x = 123
}
let a2 = objectFactory(fnf) // 模拟 new fnf()
console.log(a2.x) // 123
```

可看出并不复杂，关键点在第二步，设置对象的原型链，这也是创建实例对象的核心点。

## Q:typeof 和 instanceof 有什么区别

js 中数据类型分为两类，一类是基本数据类型，一类是对象类型。

基本数据类型有： `Number String Boolean Null Undefined BigInt Symbol`

对象类型: `Object` 也叫引用类型

1. `typeof(a)` 用于返回值的类型, 有 "number"、"string"、"boolean"、"null"、"function" 和 "undefined"、"symbol"、"object"

vbnet 复制代码

```
let a = 1
let a1 = '1'
let a2 = true
let a3 = null
let a4 = undefined
let a5 = Symbol
let a6 = {}
console.log(typeof(a),typeof(a1),typeof(a2),typeof(a3),typeof(a4),typeof(a5),typeof(a6))
// number string boolean object undefined function object
```

2. `instanceof` 用于判断该对象是否是目标实例, 根据原型链 `__proto__` 逐层向上查找, 通过 `instanceof` 也可以判断一个实例是否是其父类型或者祖先类型的实例。

有这么个面试题

javascript 复制代码

```
function person() {
  this.name = 10
}
console.log(person instanceof person)
```

结果是 `false` , 看下 `person` 函数的原型链 `person.prototype => Function.prototype => Object.prototype => null` , 所以在原型链上是找不到 `person` 的

## Q:数据类型有哪几种?

- 7 种原始数据类型: `Null` `Undefined` `String` `Number` `Boolean` `BigInt` `Symbol`
- `Object` 对象类型, 也称为引用类型

## Q:JS中基本数据类型和引用类型在内存上有什么区别?

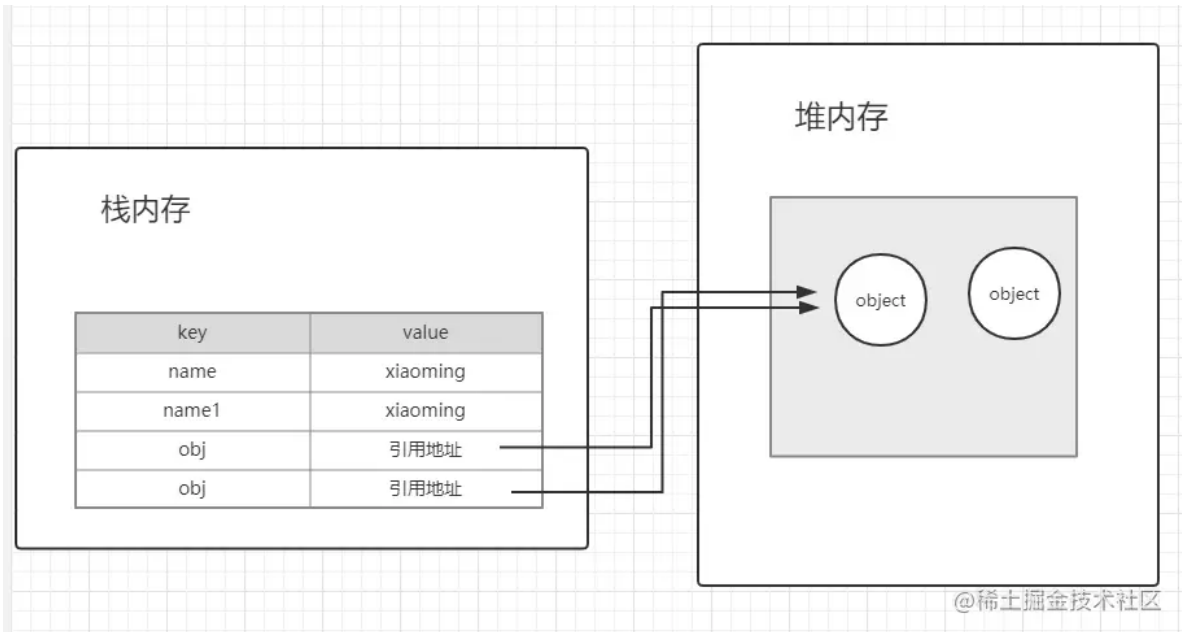
基本类型: 存储在栈内存中, 因为基本类型的大小是固定, 在栈内可以快速查找。

引用类型: 存储在堆内存中, 因为引用类型的大小是不固定的, 所以存储在堆内存中, 然后栈内存中仅存储堆中的内存地址。

我们在查找对象是从栈中查找，那么可得知，对于基本对象我们是对它的值进行操作，而对于引用类型，我们是对它的引用地址操作。

ini 复制代码

```
var name = 'xiaoming'  
var name1 = name; // 值拷贝  
var obj = {age:10}  
var obj1 = obj // 引用地址的拷贝，所以这两个对象指向同一个内存地址，那么他们其实是同一个对象
```



关于函数的传参是传值还是传引用呢？

很多人说基本类型传值，对象类型传引用，但严格来说，函数参数传递的是值，上图可以看出，就算是引用类型，它在栈中存储的还是一串内存地址，所以也是一个值。不过我觉得没必要过于纠结这句话，理解就行。

## Q:描述 NaN 指的是什么

NaN 属性是代表非数字值的特殊值，该属性用于表示某个值不是数字。

NaN 是 Number 对象中的静态属性

scss 复制代码

```
typeof(NaN) // "number"  
NaN == NaN // false
```

那怎么判断一个值是否是 NaN 呢？若支持 es6 ，可直接使用 `Number.isNaN()`

若不支，可根据 `NAN !== NAN` 的特性

javascript 复制代码

```
function isReallyNaN(val) {  
    let x = Number(val);  
    return x !== x;  
}
```

## Q:描述 null

null 是基本类型之一，不是 Object 对象，至于为什么？答曰：历史原因，咱也不敢多问

javascript 复制代码

```
typeof(null) // "object"  
null instanceof Object // false
```

那怎么判断一个值是 null 呢？可根据上面描述的特性，得

javascript 复制代码

```
function isNull(a) {  
    if (!a && typeof (a) === 'object') {  
        return true  
    }  
    return false  
}  
  
console.log(isNull(0))    // false  
console.log(isNull(false))// false  
console.log(isNull(''))  // false  
console.log(isNull(null))// true
```

## Q:什么是包装对象

包装对象，只要是为了便于基本类型调用对象的方法。

包装对象有三种： `String Number Boolean`

这三种原始类型可以与实例对象进行自动转换，可把原始类型的值变成（包装成）对象，比如在字符串调用函数时，引擎会将原始类型的值转换成只读的包装对象，执行完函数后就销毁。

## Q:class 和 function 的区别

class 也是一个语法糖，本质还是基于原型链，class 语义化和编码上更加符合面向对象的思维。

对于 `function` 可以用 `call apply bind` 的方式来改变他的执行上下文，但是 `class` 却不可以，class 虽然本质上也是一个函数，但在转成 es5 (babel) 做了一层代理，来禁止了这种行为。

- class 中定义的方法不可用 `Object.keys()` 遍历
- class 不可以定义私有的属性和方法，function 可以，只要不挂载在 this 作用域下就行
- class 只能通过类名调用
- class 的静态方法，this 指向类而非实例

## Q:实现继承的几种方法

因为涉及的代码较多，所以独立写一篇文章来总结，传送门: [js-实现继承的几种方式](#)

## Q:谈谈作用域链机制

先说下作用域的这个概念，作用域就是变量和函数的可访问范围，控制这个变量或者函数可访问行和生命周期（这个很重要）。

在 js 中是词法作用域，意思就是你的变量函数的作用域是由你的编码中的位置决定的，当然可以通过 `apply bind` 等函数进行修改。

在 ES6 之前，js 中的作用域分为两种：函数作用域和全局作用域。

全局作用域顾名思义，浏览器下就是 `window`，作用域链的顶级就是它，那么只要不是被函数包裹的变量或者函数，它的作用域就是全局。

而函数作用域，就是在函数的体内声明的变量、函数及函数的参数，它们的作用域都是在这个函数内部。那么函数中的未在该函数内定义的变量呢？这个变量怎么获取呢？这就是作用域链的概念了。

我们知道函数在执行时是有个执行栈，在函数执行的时候会创建执行环境，也就是执行上下文，在上下文中有个大对象，保存执行环境定义的变量和函数，在使用变量的时候，就会访问这个大对象，这个对象会随着函数的调用而创建，函数执行结束出栈而销毁，那么这些大对象组成一个链，就是作用域链。

那么函数内部未定义的变量，就会顺着作用域链向上查找，一直找到同名的属性。

## 看下面这个栗子

ini 复制代码

```
var a = 10;
function fn() {
  var b = 20;
  function bar() {
    console.log(a + b) // a 一直往上找，直到最高层级找到了， b 往上找，在函数 fn 这一层级的上下文中找到了
  }
  return bar
}
b = 200;
var x = fn();
x()
```

在看看闭包的作用域，只要存在函数内部调用，执行栈中就会保留父级函数和函数对于的作用域，所以父函数的作用域在子函数的作用域链中，直到子函数被销毁，父级作用域才会释放，来个很常见的面试题

scss 复制代码

```
function test() {
  for (var index = 0; index < 3; index++) {
    setTimeout(() => {
      console.log('index:' + index)
    })
  }
}

test()
// index:3
// index:3
// index:3
```

执行结果是 3个3，因为js的事件循环机制，就不细说，那么我们想让它按顺序输出，咋办呢？

思路就是，因为定时器的回调肯定是在循环结束后才执行，那时候 index 已经是3了，那么可以利用上面说的闭包中的作用域链，在子函数中去引用父级的变量，这样子函数没有被销毁前，这个变量是会一直存在的，所以我们可以这么改。

javascript 复制代码

```
function test() {
  for (var index = 0; index < 3; index++) {
    ((index) => {
      setTimeout(() => {
        console.log('index:' + index)
      })
    })(index)
  }
}
```



```

    })
  })(index)
}
}

```

## 我们在看一道面试题

scss 复制代码

```

function f(fn, x) {
  console.log('into')
  if (x < 1) {
    f(g, 1);
  } else {
    fn();
  }
  function g() {
    console.log('x' + x);
  }
}

function h() {
}

f(h, 0) // x 0

```

逻辑很简单，但面试题就是这么鬼精，越是简单越有坑。

g 函数中的 x 变量是引用父级的，而 f 函数执行了两次，x 变量依次为 0 1，在 f(h,0) 这个函数执行的时候，这个函数的作用域中的 x=0，这个时候 g 函数中引用的 x 就是当前执行上下文中的 x=0 这个变量，但这个函数还没被执行，接着到了 f(g, 1) 执行，这一层执行上下文中的 x=1，但注意两次 f 执行的作用域不是同一个对象，是作用域链上两个独立的对象，最后到了 fn()，这个 fn 是一个参数，也就是在 f(h,0) 执行的时候 g 函数，那么 g 函数在这里被执行，g 打印出来的 x 就是 0。

块级作用域： `let` `const` 的出现就是为了解决 js 中没有块级作用域的弊端。

其他小点：

- for 循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域
- 函数中的变量可以分为自由变量(当前作用域没有定义的变量)和本作用域变量，自由变量的取值要到创建这个函数的那个域（非常重要），也叫做静态作用域。
- 作用域和执行上下文的区别，看下引擎执行脚本的两个阶段

解释阶段：词法分析 -> 语法分析 -> 作用域规则确定 执行阶段：创建执行上下文 -> 执行函数代码 -> 垃圾回收

参考连接： [segmentfault.com/a/1190000001...](https://segmentfault.com/a/1190000001...) [www.cnblogs.com/dolphinX/p/...](http://www.cnblogs.com/dolphinX/p/...)

## Q:let var const 的区别

var: 解析器在对js解析时，会将脚本扫描一遍，将变量的声明提前到代码块的顶部，赋值还是在原先的位置，若在赋值前调用，就会出现暂时性死区，值为 `undefined`

let const: 不存在在变量提升，且作用域是存在于块级作用域下，所以这两个的出现解决了变量提升的问题，同时引用块级作用域。 注：变量提升是为了解决函数互相调用的问题。

## Q:数据属性和访问器属性的区别

其实就是问对 `Object.defineProperty` 的掌握程度。

### 1. 数据属性（数据描述符）

相关的属性如下：

[[Configurable]]：表示能否通过 `delete` 删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为访问器属性。

[[Enumerable]]：表示能否通过 `for-in` 循环返回属性。

[[Writable]]：表示能否修改属性的值。

[[Value]]：包含这个属性的值。读取属性值的时候，从这个位置读；写入属性值的时候，把新值保存在这个位置。这个特性的默认值为 `undefined`。

数据属性可以直接定义，如 `var p = {name:'xxx'}` 这个 `name` 就是数据属性，直接定义下，相关属性值都是 `true`，如果要修改默认的定义值，那么使用 `Object.defineProperty()` 方法，如下面这个栗子

css 复制代码

```
var p = {  
  name: 'dage'  
}  
  
Object.defineProperty(p, 'name', {
```

```

    value: 'xxx'
  })
  p.name = '4rrr'
  console.log(p.name) // 4rrr
  Object.defineProperty(p, 'name', {
    writable: false,
    value: 'again'
  })
  p.name = '4rrr'
  console.log(p.name) // again

```

- 调用Object.defineProperty()方法时，如果不显示指定configurable，enumerable，writable的值，就默认为false
- 如果 writable 为 false，但是 configurable 为 true，还是可以对属性重新赋值的。

## 2. 访问器属性（存取描述符）

访问器属性不包含数据值，没有 value 属性，有 `get set` 属性，通过这两个属性来对值进行自定义的读和写，可以理解为取值和赋值前的拦截器，相关属性如下：

[[Configurable]]：表示能否通过 delete 删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为数据属性，默认 false

[[Enumerable]]：表示能否通过 for-in 循环返回属性，默认 false

[[Get]]：在读取属性时调用的函数。默认值为 undefined

[[Set]]：在写入属性时调用的函数。默认值为 undefined

- 访器属性不能直接定义，必须使用 Object.defineProperty() 来定义。
- 根据 `get set` 的特性，可以实现对象的代理，`vue` 就是通过这个实现数据的劫持。

两者的相同点：都有 Configurable 和 Enumerable 属性。

## 一个简单的小demo

```

var p = {
  name: ''
}
Object.defineProperty(p, 'name', {
  get: function(){
    return 'right yeah !'
  },

```

javascript 复制代码

```
    set:function(val){
        return 'handsome '+val
    }
})
p.name = `xiaoli`
console.log(p.name) // right yeah !
```

参考连接:

[cloud.tencent.com/developer/a...](https://cloud.tencent.com/developer/a...)

[developer.mozilla.org/zh-CN/docs/...](https://developer.mozilla.org/zh-CN/docs/...)

## Q:toString 和 valueOf 有什么区别

在 Object 中存在这两个方法，继承Object的对象可以重写方法。这两个方法主要用于隐式转换，比如

js 不同于其他语言，两个不同的数据类型可以进行四则运算和判断，这就归功于隐式转换了，隐式转换我就不详细介绍了，因为我没有被问到~

复制代码

```
1 + '1' // 11 : 整型 1 被转换成字符串 '1'，变成了 '1' + '1' = '11'
2 * '3' // 6 : 字符串 '3' 被转换成整型 3，变成了 2 * 3 = 6
```

那么我们也可以对自定义的对象重写这两个函数，以便进行隐式转换

javascript 复制代码

```
let o = function () {
    this.toString = () => {
        return 'my is o,'
    }
    this.valueOf = () => {
        return 99
    }
}
let n = new o()
console.log(n + 'abc') // 99abc
console.log(n * 10) // 990
// 有没有很酷炫
```

当这两个函数同时存在时候，会先调用 `valueOf`，若返回的不是原始类型，那么会调用 `toString` 方法，如果这时候 `toString` 方法返回的也不是原始数据类型，那么就会报错 `TypeError: Cannot convert object to primitive value` 如下

javascript 复制代码

```
let o = function () {
  this.toString = () => {
    console.log('into toString')
    return { 'string': 'ssss' }
  }
  this.valueOf = () => {
    console.log('into valueOf')
    return { 'val': 99 }
  }
}
let n = new o()
console.log(n + 'xx')
//into valueOf
//into toString
// VM1904:12 Uncaught TypeError: Cannot convert object to primitive value
```

## Q:箭头函数有没有 arguments 对象?

(非常感谢评论区伙伴的提醒)

`arguments` 是一个类数组对象，可以获取到参数个数和参数列表数组，对于不定参数的函数，可以用 `arguments` 获取参数。

那么对于箭头函数有没有 `arguments` 呢？ 需要看具体执行的场景了

javascript 复制代码

```
// 箭头函数
let aa1 = (...args) => {
  let bb = [].slice.call(arguments, 0)
  let a = arguments[0]
  let b = arguments[1]
  let c = arguments[2]
  console.log(a + b + c)
}

// 正常的函数
let aa = function (...args) {
  let bb = [].slice.call(arguments, 0)
  let a = arguments[0]
  let b = arguments[1]
  let c = arguments[2]
```

```
    console.log(a + b + c)
  }
  aa(1, 2, 3)
  aa1(1, 2, 3)
```

分别观察以下两个场景的执行结果

## 浏览器中执行

直接看结果

```
> let aa1 = (...args)=>{
    let bb = [].slice.call(arguments,0)
    let a = arguments[0]
    let b = arguments[1]
    let c = arguments[2]
    console.log(a+b+c)
  }
  aa1()

✖ ▶ Uncaught ReferenceError: arguments is not defined
    at aa1 (<anonymous>:2:29)
    at <anonymous>:9:1

> let aa = function(...args){
    let bb = [].slice.call(arguments,0)
    let a = arguments[0]
    let b = arguments[1]
    let c = arguments[2]
    console.log(a+b+c)
  }
  aa(1,2,3)

6
```

很明显，在浏览器中 `arguments` 是不存在的

## nodejs 中执行

结果（为了辨认，输出前加了段字符串）

```
$ node test/tt.js
normal func:6
jiantou func:[object Object]function require(path) {
  try {
    exports.requireDepth += 1;
    return mod.require(path);
  } finally {
    exports.requireDepth -= 1;
  }
}[object Object]
```

执行过程没有报错，说明 `arguments` 是存在的，那为啥结果不是预期的 6 呢？



我们对箭头函数打断点看看

```
Arguments(3) [1, 2, 3]
  callee: TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on
  length: 3
  > Symbol(Symbol.iterator): function values() { ... }
  > __proto__: Object {constructor: , __defineGetter__: , __defineSetter__: , ...}
  0: 1
  1: 2
  2: 3
```

@稀土掘金技术社区

arguments 对象看着没啥问题，传入的参数也看到了

我们看看通过数组方式获取到的值

```
Block
  this: undefined
  > a: Object {}
  > b: function require(path) { ... }
  > bb: Array(5) [Object, , Module, ...]
    length: 5
    > __proto__: Array(0) [, ...]
    > 0: Object {}
    > 1: function require(path) { ... }
    > 2: Module {id: ".", exports: Object, p...
      3: "e:\code\practice\test\tt.js"
      4: "e:\code\practice\test"
    > c: Module {id: ".", exports: Object, pa...
      > children: Array(0) []
      > exports: Object {}
      filename: "e:\code\practice\test\tt.js"
      id: "."
      loaded: false
      parent: null
      > paths: Array(4) ["e:\code\practice\tes...
      > __proto__: Object {load: , require: , ...}
```

竟然是这些东西，这些是当前脚本执行的模块信息，并不是我们预期的参数列表

## 结论

1. 在浏览器中箭头函数没有 `arguments`
2. 在 nodejs 中，有 `arguments`，可通过其获取参数长度，但不能通过改对象获取参数列表

(我也不太懂这个对象的原理，还请知道的伙伴在评论区告知，谢谢)

## Q:js 精度丢失问题

浮点数的精度丢失不仅仅是js的问题，java 也会出现精度丢失的问题（没有黑java），主要是因为数值在内存是由二进制存储的，而某些值在转换成二进制的时候会出现无限循环，由于位数限制，无限循环的值就会采用“四舍五入法”截取，成为一个计算机内部很接近数字，即使很接近，但是误差已经出现了。

举个栗子

arduino 复制代码

```
0.1 + 0.2 = 0.30000000000000004
// 0.1 转成二进制会无限循环
// "0.000110011001100110011001100110011001100110011001100..."
```

那么如何避免这问题呢？解决办法：可在操作前，放大一定的倍数，然后再除以相同的倍数

ini 复制代码

```
(0.1 * 100 + 0.2 * 100) / 100 = 0.3
```

js 的 number 采用 64位双精度存储 JS 中能精准表示的最大整数是 `Math.pow(2, 53)`

推荐一个开源工具 (number-precision)[[github.com/nefe/number...](https://github.com/nefe/number-precision)]

## Q: toFixed 可以做到四舍五入吗

`toFixed` 对于四舍六入没问题，但对于尾数是 5 的处理就非常诡异

scss 复制代码

```
(1.235).toFixed(2) // "1.24" 正确
(1.355).toFixed(2) // "1.35" 错误
```

我也没明白为啥这么设计，严格的四舍五入可以采用以下函数

typescript 复制代码

```
// 使用 Math.round 可以四舍五入的特性，把数组放大一定的倍数处理
function round(number, precision) {
  return Math.round(+number + 'e' + precision) / Math.pow(10, precision);
}
```

原理是，`Math.round` 是可以做到四舍五入的，但是仅限于正整数，那么我们可以放大至保留一位小数，计算完成后再缩小倍数。

## Q: js中不同进制怎么转换

10 进制转其他进制: `Number(val).toString([2,8,10,16])`

其他进制转成10进制: `Number.parseInt("1101110",[2,8,10,16])`

其他进制互转: 先将其他进制转成 10 进制, 在把 10 进制转成其他进制

## Q:对js处理二进制有了解吗

ArrayBuffer: 用来表示通用的、固定长度的原始二进制数据缓冲区, 作为内存区域, 可以存放多种类型的数据, 它不能直接读写, 只能通过视图来读写。

同一段内存, 不同数据有不同的解读方式, 这就叫做“视图” (view), 视图的作用是以指定格式解读二进制数据。目前有两种视图, 一种是 `TypedArray` 视图, 另一种是 `DataView` 视图, 两者的区别主要是字节序, 前者的数组成员都是同一个数据类型, 后者的数组成员可以是不同的数据类型。

Blob: 也是存放二进制的容器, 通过 `FileReader` 进行转换。

之前有做过简单的总结, 大家可以看看: [nodejs 二进制与Buffer](#)

毕竟对这块应用的比较少, 推荐一篇文章给大家 [二进制数组](#)

## Q:异步有哪些解决方案

这个问题出场率很高呀! 常见的有如下几个:

- **回调函数**: 通过嵌套调用实现
- **Generator**: 异步任务的容器, 生成器本质上是一种特殊的迭代器, Generator 执行后返回的是个指针对象, 调用对象里的 next 函数, 会移动内部指针, 分阶段执行 **Generator** 函数, 指向 **yield** 语句, 返回一个对象 {value:当前的执行结果, done:是否结束}
- **promise**: 而是一种新的语法糖, Promise 的最大问题是代码冗余, 通过 then 传递执行权, 因为需求手动调用 then 方法, 当异步函数多的时候, 原来的语义变得很不清楚
- **co**: 把 Generator 和 Promise 封装, 达到自动执行
- **async\await**: 目前是es7草案, 可通过 **babel webpack** 等工具提前使用, 目前原生浏览器支持还不太好。其本质上是语法糖, 跟 co 库一样, 都是对 **generator+promise** 的封装, 不过相比 co, 语义化更好, 可以像普通函数一样调用, 且大概率是未来的趋势。

## Q:简单介绍Generator

Generator 函数就是一个封装的异步任务，或者说是异步任务的容器。

Generator 的核心是可以暂停函数执行，然后在从上一次暂停的位置继续执行，关键字 `yield` 标识暂停的位置。

Generator 函数返回一个迭代器对象，并不会立即执行函数里面的方法，对象中有 `next()` 函数，函数返回 `value` 和 `done` 属性，`value` 属性表示当前的内部状态的值，`done` 属性标识是否结束的标志位。

Generator 的每一步执行是通过调用 `next()` 函数，`next` 方法可以带一个参数，该参数就会被当作上一个 `yield` 表达式的返回值。执行的步骤如下：

(1) 遇到 `yield` 表达式，就暂停执行后面的操作，并将紧跟在 `yield` 后面的那个表达式的值，作为返回的对象的 `value` 属性的值。

(2) 下一次调用 `next` 方法时，再继续往下执行，直到遇到下一个 `yield` 表达式。

(3) 如果没有再遇到新的 `yield` 表达式，就一直运行到函数结束，直到 `return` 语句为止，并将 `return` 语句后面的表达式的值，作为返回的对象的 `value` 属性值。

(4) 如果该函数没有 `return` 语句，则返回的对象的 `value` 属性值为 `undefined`。注意：`yield` 表达式，本身是没有值的，需要通过 `next()` 函数的参数将值传进去。

javascript 复制代码

```
let go = function* (x) {
  console.log('one', x)
  let a = yield x * 2
  console.log('two', a)
  let b = yield x + 1
  sum = a + b
  return sum
}
let g = go(10)
let val = g.next()
while (!val.done) {
  val = g.next(val.value)
}
console.log(val)
```

可见 Generator 的弊端很明显，执行流程管理不方便，异步返回的值需要手动传递，编码上较容易出错。

## Q: 讲一讲 Promise

Promise 已经是 ES6 的规范了，相比 Generator，设计的更加合理和便捷。

看看Promise的规范：

1. 一个 Promise 的当前状态必须为以下三种状态中的一种：等待态（Pending）、执行态（Fulfilled）和拒绝态（Rejected），状态的改变只能是单向的，且变化后不可在改变。
2. 一个 Promise 必须提供一个 then 方法以访问其当前值、终值和据因。  
promise.then(onFulfilled, onRejected) 回调函数只能执行一次，且返回 promise 对象

promise 的每个操作返回的都是 promise 对象，可支持链式调用。通过 then 方法执行回调函数，Promise 的回调函数是放在事件循环中的微队列。

## Q: co库的执行原理

co 用 promise 的特性，将 Generator 包裹在 Promise 中，然后循环执行 next 函数，把 next 函数返回的 value 用 promise 包装，通过 then.resolve 调用下一个 next 函数，并将值传递给 next 函数，直到 done 为 true，最后执行包裹 Generator 函数的 resolve。

我们看下源码，源码做了截取

javascript 复制代码

```
function co(gen) {  
  return new Promise(function(resolve, reject) { // 最外层是一个 Promise 对象  
    if (typeof gen === 'function') gen = gen.apply(ctx, args);  
    if (!gen || typeof gen.next !== 'function') return resolve(gen);  
  
    onFulfilled();  
  
    function onFulfilled(res) {  
      var ret;  
      try {  
        ret = gen.next(res); // 将上一步的返回值传递给 next  
      } catch (e) {  
        return reject(e);  
      }  
      next(ret); // 将上一步执行结果转换成 promise  
      return null;  
    }  
  })  
}
```

```

/**
 * Get the next value in the generator,
 * return a promise.
 *
 * @param {Object} ret
 * @return {Promise}
 * @api private
 */

function next(ret) {
  if (ret.done) return resolve(ret.value); // done为true, 就表示执行结束, resolve结果出去
  var value = toPromise.call(ctx, ret.value); // toPromise 是个工具函数, 将对象转换成 promise, 可
  if (value && isPromise(value)) return value.then(onFulfilled, onRejected); // then 函数执行后
  return onRejected(new TypeError('You may only yield a function, promise, generator, array, c
    + 'but the following object was passed: ' + String(ret.value) + ''));
}
});
}

```

## Q:介绍下浏览器的事件循环

这是必考题呀, 盆友们, 这个可阅读我以前写的一篇文章, 传送门: [js 事件循环](#)

## Q:介绍下模块化方案

这个东西有点多, 可以看我之前的一篇总结, 传送门: [面试官让我解释前端模块化](#)

## Q:垃圾回收机制

为什么需要垃圾回收: 因为对象需要占用内存, 而内存资源是有限的。

js 会周期性的对不在使用的对象销毁, 释放内存, 关键点就在于怎么识别哪些对象是垃圾。

垃圾对象: 对象没有被引用, 或者几个对象形成循环引用, 但是根访问不到他们, 这些都是可回收的垃圾。

垃圾回收的两种机制: 标记清除和引用计数

## 标记清除法



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。

比如说函数中声明了一个变量，就做一个标记，当函数执行完成，退出执行栈，这个变量的标记就变成已使用完。

目前主流浏览器采用的是这个策略

## 引用计数

跟踪每个值被引用的次数，声明一个变量后，这个变量每被其他变量引用一次，就加 1，如果变量引用释放了，就减 1，当引用次数为 0 的时候，对象就被清理。但这个有个循环引用的弊端，所以应用的比较少。

## 垃圾收集的性能优化

1. 分代回收，对象分成两组，新生带、老生带，
2. 增量回收
3. 空闲时间回收

## 编码可以做的优化

1. 避免重复创建对象。
2. 在适当的时候解除引用，是为页面获的更好性能的一个重要方式。
3. 全局变量什么时候需要自动释放内存空间则很难判断，因此在开发中，需要尽量避免使用全局变量。

## Q:什么是严格模式

通过在脚本的最顶端放上一个特定语句 `"use strict"`；整个脚本就可开启严格模式语法。

严格模式下有以下好处：

1. 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为；
2. 消除代码运行的一些不安全之处，保证代码运行的安全；

3. 提高编译器效率，增加运行速度；
4. 为未来新版本的Javascript做好铺垫。

如以下具体的场景：

1. 严格模式会使引起静默失败(silently fail,注:不报错也没有任何效果)的赋值操作抛出异常
2. 严格模式下的 eval 不再为上层范围(surrounding scope,注:包围eval代码块的范围)引入新变量
3. 严格模式禁止删除声明变量
4. 在严格模式中一部分字符变成了保留的关键字。这些字符包括implements, interface, let, package, private, protected, public, static和yield。在严格模式下，你不能再用这些名字作为变量名或者形参名。
5. 严格模式下 arguments 和参数值是完全独立的，非严格下修改是会相互影响的

## Q:map 和 weakMap 的区别

map 的 key 可以是任意类型，在 map 内部有两个数组，分别存放 key 和 value ，用下标保证两者的一一对应，在对 map 操作时，内部会遍历数组，时间复杂度O(n)，其次，因为数组会一直引用每个键和值，回收算法没法回收处理，可能会导致内存泄露。

相比之下， WeakMap 的键值必须是对象，持有的是每个键对象的 弱引用 ，这意味着在没有其他引用存在时垃圾回收能正确进行。

ini 复制代码

```
const wm1 = new WeakMap();
const o1 = {};
wm1.set(o1, 37); // 当 o1 对象被回收，那么 WeakMap 中的值也被释放
```

## Q:String 和 Array 有哪些常用函数

我也不知道为什么会有这种笔试题...

1. String:

split(): 方法使用指定的分隔符字符串将一个String对象分割成子字符串数组

slice(): 方法提取某个字符串的一部分，并返回一个新的字符串，且不会改动原字符串

substring(): 方法返回一个字符串在开始索引到结束索引之间的一个子集, 或从开始索引直到字符串的末尾的一个子集

## 2. Array:

slice(): 方法返回一个新的数组对象, 这一对象是一个由 begin 和 end 决定的原数组的浅拷贝 (包括 begin, 不包括end) 。原始数组不会被改变。

splice(): 方法通过删除或替换现有元素或者原地添加新的元素来修改数组,并以数组形式返回被修改的内容。此方法会改变原数组。

push(): 方法将一个或多个元素添加到数组的末尾, 并返回该数组的新长度。

pop(): 方法从数组中删除最后一个元素, 并返回该元素的值。此方法更改数组的长度。

shift():方法从数组中删除第一个元素, 并返回该元素的值。此方法更改数组的长度。

unshift(): 方法将一个或多个元素添加到数组的开头, 并返回该数组的新长度(该方法修改原有数组)。

## Q:判断数组的几种方法

这题主要还是考察对原型链的理解

1. `Array.isArray()` ES6 api
2. `obj instanceof Array` 原型链查找
3. `obj.constructor === Array` 构造函数类型判断
4. `Object.prototype.toString.call(obj) === '[object Array]'` `toString` 返回表示该对象的字符串, 若这个方法没有被覆盖, 那么默认返回 `"[object type]"` , 其中 `type` 是对象的类型。需要准确判断类型的话, 建议使用这种方法

## Q:循环有几种方式, 是否支持中断和默认情况下是否支持async/await

- for 支持中断、支持异步事件
- for of 支持中断、支持异步事件
- for in 支持中断、支持异步事件
- forEach 不支持中断、不支持异步事件
- map 不支持中断、不支持异步事件, 支持异步处理方法: map 返回promise数组, 在使用 Promise.all 一起处理异步事件数组

- reduce 不支持中断、不支持异步事件，支持异步处理方法：返回值返回 promise 对象

map 的比较简单就不写了，我写个 `reduce` 处理 `async/await` 的 demo

javascript 复制代码

```
const sleep = time => new Promise(res => setTimeout(res, time))
async function ff(){
  let aa = [1,2,3]
  let pp = await aa.reduce(async (re,val)=>{
    let r = await re;
    await sleep(3000)
    r += val;
    return Promise.resolve(r)
  },Promise.resolve(0))
  console.log(pp) // 6
}
ff()
```

## Q:闭包的使用场景列举

闭包：定义在一个函数内部的函数，内部函数持有外部函数内变量的引用，这个内部的函数有自己的执行作用域，可以避免外部污染。

关于闭包的理解，可以说是一千个读者就有一千个哈姆雷特，找到适合自己理解和讲述的就行。

场景有：

1. 函数式编程，compose curry
2. 函数工厂、单利
3. 私有变量和方法，面向对象编程

## Q:扩展运算符

这题面试官估计是想知道你是不是真的用过 es6 吧

扩展运算符 (...) 也会调用默认的 Iterator 接口。

扩展运算符主要用在不定参数上，可以将参数转成数组形式

```
function fn(...arg){  
  console.log(arg) // [ 1, 2, 3 ]  
}  
fn(1,2,3)
```

## Q:线程和进程分别是什么

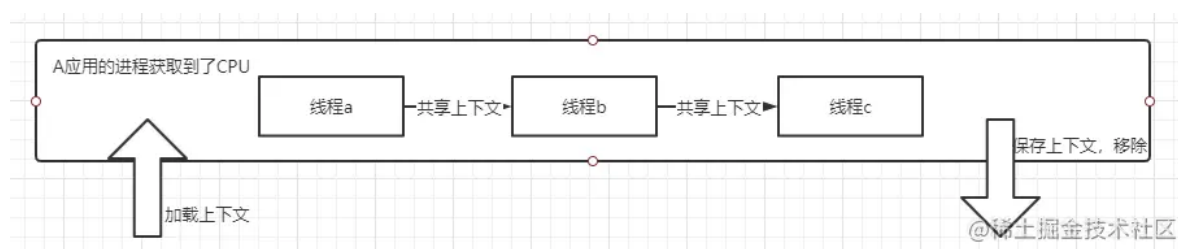
首先来一句话概括：进程和线程都是一个时间段的描述，都是对CPU工作时间段的描述。

当一个任务得到 CPU 资源后，需要加载执行这个任务所需要的执行环境，也叫上下文，进程就是包含上下文切换的程序执行时间总和 = CPU加载上下文 + CPU执行 + CPU保存上下文。可见进程的颗粒度太大，每次都需要上下文的调入，保存，调出。

如果我们把进程比喻为一个运行在电脑上的软件，那么一个软件的执行不可能是一条逻辑执行的，必定有多个分支和多个程序段，就好比要实现程序A，实际分成 a, b, c等多个块组合而成。

那么这里具体的执行就是：程序A得到CPU => CPU加载上下文 => 开始执行程序A的a小段 => 然后执行A的b小段 => 然后再执行A的c小段 => 最后CPU保存A的上下文。这里a, b, c的执行共享了A的上下文，CPU在执行的时候没有进行上下文切换的。

a, b, c 我们就是称为线程，就是说线程是共享了进程的上下文环境，是更为细小的 CPU 执行时间段。



## Q:了解函数式编程吗

函数式编程的两个核心：合成和柯里化，之前对函数式编程做过总结，传送门：[【面试官问】你懂函数式编程吗？](#)

## Q:什么是尾递归？

先给面试官简单说下什么是递归函数：函数内部循环调用自身的就是递归函数，若函数没有执行完毕，执行栈中会一直保持函数相关的变量，一直占用内存，当递归次数过大的时候，就可能会出现内存溢出，也叫爆栈，页面可能会卡死。所以为了避免出现这种情况，可以采用尾递归。

尾递归：在函数的最后一步是调用函数，进入下一个函数不在需要上一个函数的环境了，内存空间  $O(n)$  到  $O(1)$  的优化，这就是尾递归。尾递归的好处：可以释放外层函数的调用栈，较少栈层级，节省内存开销，避免内存溢出。

网上很多用斐波那契数列作为栗子，但我偏不，我用个数组累加的栗子

scss 复制代码

```
function add1(arr) {
  if (arr.length === 0) {
    return 0
  }
  return add1(arr.slice(1)) + arr[0] // 还有父级函数中 arr[0] 的引用
}

function add(arr, re) {
  if (arr.length === 0) {
    return re + 0
  } else {
    return add(arr.slice(1), arr[0] + re) // 仅仅是函数调用
  }
}

console.log(add([1, 2, 3, 4], 0)) // 10
console.log(add1([1, 2, 3, 4])) // 10
```

## Q:观察者模式 发布-订阅模式 的区别

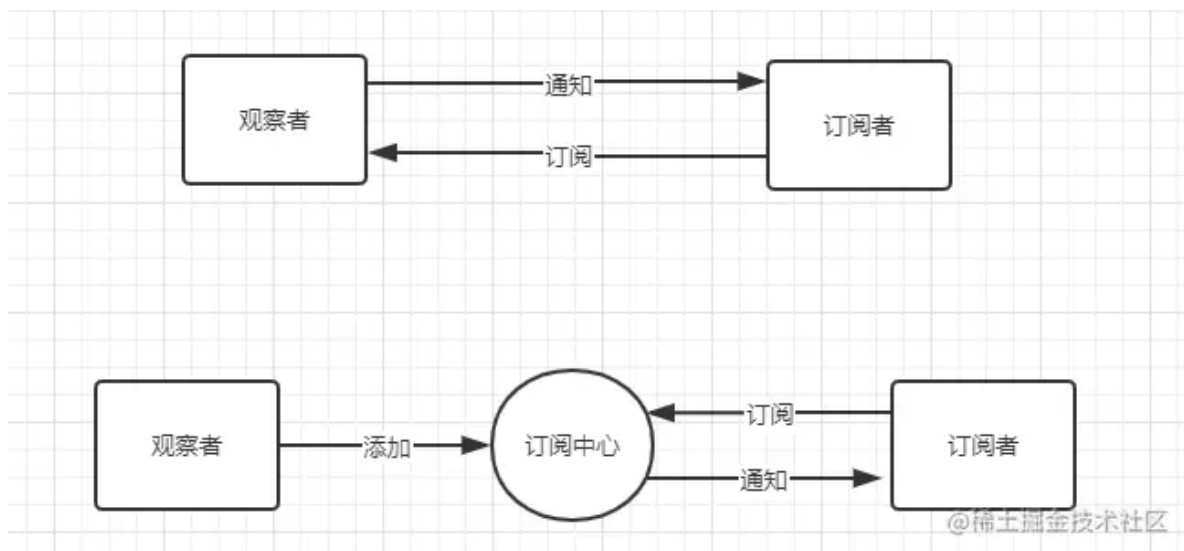
两者都是订阅-通知的模式，区别在于：

观察者模式：观察者和订阅者是互相知道彼此的，是一个紧耦合的设计

发布-订阅：观察者和订阅者是不知道彼此的，因为他们中间是通过一个订阅中心来交互的，订阅中心存储了多个订阅者，当有新的发布的时候，就会告知订阅者

设计模式的名词实在有点多且绕，我画个简单的图：

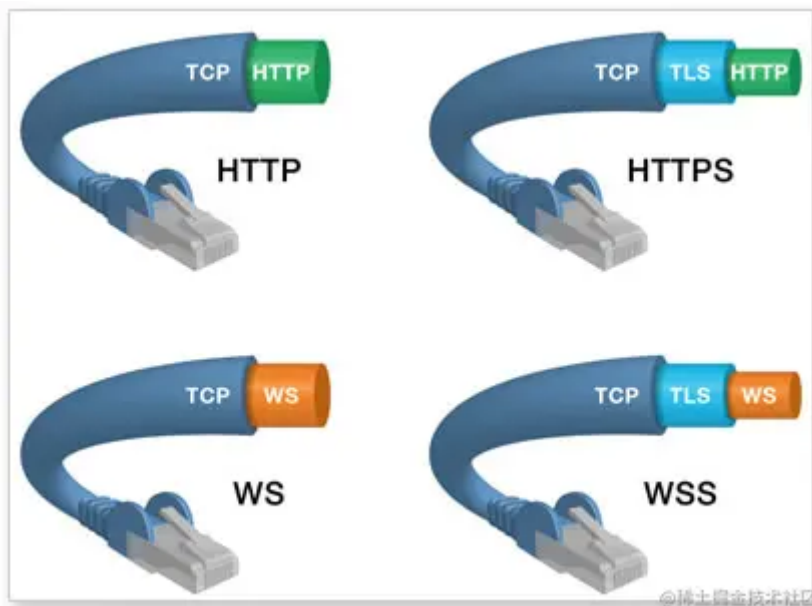




## Q:WebSocket

这个就问到了一次，所以简单进行了了解。

简单来说，WebSocket 是应用层协议，基于 tcp，与HTTP协议一样位于应用层，都是TCP/IP 协议的子集。



HTTP 协议是单向通信协议，只有客户端发起HTTP请求，服务端才会返回数据。而 WebSocket 协议是双向通信协议，在建立连接之后，客户端和服务端都可以主动向对方发送或接受数据。

