
JavaScript
一、数据类型
1.数据类型
2.检测类型的方式
typeof
在判断 对象、数组、null时都是 object
只能判断引用类型数据类型，而不能判断基本数据类型
instanceof
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
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
constructor
判断数据的类型
对象实例通过constructor 对象访问它的构造函数
如果创建一个对象来改变它的原型，constructor 则不能用来判断数据类型了
Object.prototype.toString.call()
3.判断数组的方式
Object.prototype.toString.call([])
[] _proto_ === Array.prototype
Array.isArray([])
[] instanceof Array
Array.prototype.isPrototypeOf([])
4.null 和 undefined 的区别
null：空对象
undefined: 未定义
5.instanceof 操作符的实现原理及实现
6.typeof NaN
7.Object.is() 、"===" 、"=="
8.JavaScript 中包装类型
9.object.assign、扩展运算符，两者区别
10.判断一个对象是空对象
引用数据类型
Object
Array
两种类型的区别
原始类型直接存储在栈中，占据空间小，大小固定，属于被频繁使用的数据，所以放入栈中存储。
引用类型存储在堆中，占据空间大，大小不固定。
基本数据类型
Undefined
Null
Boolean
Number
String
Symbol
BigInt
1.数据类型
二、ES6
1.let、const、var的区别
是否有块级作用域
是否存在变量提升
是否添加全局属性
是否重复声明变量
是否存在暂时性死区
是否必须设置初始值
能否改变指针指向
2.const 对象的属性可以修改吗
变量执行的地址不能改动
3.箭头函数与普通函数的区别
箭头函数更简洁
箭头函数没有自己的this，在定义地方的上层作用域继承
箭头函数this执行无法更改
箭头函数无法作为构造函数使用
箭头函数没有自己的arguments
箭头函数没有 prototype
4.扩展运算符的作用及使用场景
取出对象所有可遍历属性并拷贝到对象
将数组转为逗号分隔的参数序列，每次最多一层
5.Proxy 可以实现什么功能
6.对象与数组的解构
7.对 rest 参数的理解
可以把一个分离的参数序列整合成一个数组
三、this/call/bind/apply
理解this对象
是执行上下文中的一个属性，指向最后一次调用这个方法的对象
函数调用模式：直接调用，指向全局对象
方法调用模式：作为对象的方法调用，指向这个对象
构造器调用模式：函数用new调用，则执行前创建一个新对象，指向这个新对象
apply、call、bind
apply、call、bind区别
传参：apply入参 一个数组，call、bind入参非数组
返回值：bind返回一个绑定了传入对象的新函数，call、apply返回改变指向函数返回的结果
call、apply、bind的实现
四、原型与原型链
原型
每个构造函数都有一个prototype属性，它的属性值是一个对象
这个对象包含了改构造函数所有实例共享的属性和方法
实例的__proto__指向构造函数的prototype
原型链
当访问一个对象的属性时，如果这个对象不存在这个属性，那么它就会去它的原型链里查找这个属性
这个原型对象又会有自己的原型，于是就这样一找下去
原型链的尽头一般都是Object.prototype， 所以Object.prototype.__proto__ 为null
可以通过Object.getPrototypeOfZ()获取
六、JavaScript基础
new 操作符的实现原理
Set
size
add
delete
clear
for of
forEach
keys
values
entries
Map
size
set(key,value)
has(key)
delete(key)
clear()
keys()
values()
entries()
forEach()
for of
XMLHttpRequest
异步编程
Promise
作用
解决地狱回调
三个状态
Pendding (进行中)
Resolved(已完成)
Rejected(已拒绝)
当把一件事交给promise时，它的状态就是 Pending，任务完成了状态就成了Resolved，失败就成了Rejected。
两个过程
pending -> fulfilled: Resolved(已完成)
pending -> rejected: Rejected(已拒绝)
注意：旦从进行状态变成为其他状态就永远不能更改状态了。
缺点
无法取消 Promise，一旦新建它就会立即执行，无法中途取消
当处于pending状态时，无法得知目前进展到哪一个阶段
async & await
async 函数
async 函数返回的是一个 Promise 对象
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，绝不会阻塞后面的语句
async 的返回值会作为：Promise.resolve(返回值)，如果没有返回值则相当于：Promise.resolve(undefined)
await
await 等待的是一个表达式，这个表达式的计算结果是Promise对象或者其他值（即没有特殊限定）
如果它等到不是一个Promise对象，那么 await 表达式的运算结果就是它等到的东西
如果它等到的是一个Promise对象，await 就忙起来了，它会阻塞后面的代码，等着 Promise 对象 resolve，然后得到 resolve 的值，作为 await 表达式的运算结果
五、闭包、作用域链、执行上下文
闭包
概念
闭包是指有权访问另一个函数作用域中变量的函数
常用用途
创建私有变量
保留引用，使变量不会回收
作用域
全局作用域
是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
函数作用域
块级作用域
let和const指令可以声明块级作用域
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）
不会有变量提升，也不可以重复声明
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
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，这就是作用域链
作用：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
执行上下文
执行 上下文类型
全局执行上下文
函数执行上下文
eval 执行 上下文
执行上下文栈
首先入栈全局上下文
遇到函数调用，函数执行上下文入栈
先进后出
创建执行上下文
解析阶段
this 绑定
创建词法环境组件
创建变量环境组件
执行阶段
Presented with xmind

一、数据类型

1.数据类型

- 基本数据类型
  - Undefined
  - Null
  - Boolean
  - Number
  - String
  - Symbol
  - BigInt
- 引用数据类型
  - Object
  - Array
- 两种类型的区别在与存储位置的不同
  - 原始类型直接存储在栈中，占据空间小，大小固定，属于被频繁使用的数据，所以放入栈中存储。
  - 引用类型存储在堆中，占据空间大，大小不固定。

2. 数据类型检测的方式有哪些

- typeof
- instanceof
- constructor
- Object.prototype.toString.call()

2.1 typeof

```
// typeof
console.log(typeof 2) // number
console.log(typeof true) // boolean
console.log(typeof 'str') // string
console.log(typeof undefined) // undefined
console.log(typeof Symbol(5)) // symbol
console.log(typeof BigInt(5)) // bigint
console.log(typeof function(){}) // function

console.log(typeof {}) //object
console.log(typeof []) //object
console.log(typeof null) //object
```

- 缺点：在判断 对象、数组、null时都是 object

2.2 instanceof

```
// instanceof
console.log(2 instanceof Number) // false
console.log([] instanceof Array) // true
console.log(function(){} instanceof Function) // true
console.log({} instanceof Object) // true
```

- instanceof 只能判断引用类型数据类型，而不能判断基本数据类型
- 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
- 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

2.3 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
  - 判断数据的类型
  - 对象实例通过constructor 对象访问它的构造函数
- 缺点：如果创建一个对象来改变它的原型，constructor则不能用来判断数据类型了

2.4 Object.prototype.toString.call()

```
const ObString = Object.prototype.toString
console.log(ObString.call(2)) // [object Number]
console.log(ObString.call(true)) // [object Boolean]
console.log(ObString.call('str')) // [object String]
console.log(ObString.call([])) // [object Array]
console.log(ObString.call(function(){})) // [object Function]
console.log(ObString.call({})) // [object Object]
console.log(ObString.call(undefined)) // [object Undefined]
console.log(ObString.call(null)) // [object 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...）。

3. 判断数组的方式有哪些?

```
console.log(Object.prototype.toString.call([]).slice(8,-1) === 'Array') // true
console.log([].__proto__ === Array.prototype) // true
console.log(Array.isArray([])) // true
console.log([] instanceof Array) // true
console.log(Array.prototype.isPrototypeOf([])) // true
```

4. null 和 undefined 的区别?

- 定义
  - null：空对象
  - undefined: 未定义
- 判断

```
null == undefined // true
null === undefined // false
```

5. typeof null 的结果为什么是object?

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

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

- null 的值是机器码 NULL 指针(null 指针的值全是 0)
- 就是说null的类型标签也是000，和Object的类型标签一样，所以会被判定为Object。

6. instanceof 操作符的实现原理及实现

```
function myInstanceof (left, right) {
    // 获取对象的原型
    let proto = Object.getPrototypeOf(left)
    // 获取构造函数的 prototype 对象
    let prototype = right.prototype

    // 判断构造函数的 prototype 对象是否在对象的原型链上
    while (true) {
        if (!proto) return false
        if (proto === prototype) return true
        // 如果没有找到，就继续从其原型上找，Object.getPrototypeOf方法用来获取指定对象的原
型
        proto = Object.getPrototypeOf(proto)
    }
}
console.log(myInstanceof(2, Number)) // true
console.log(myInstanceof([], Array)) // true
```

7.typeof NaN的结果是什么?

```
typeof NaN // 'number'
```

8. Object.is() 与比较操作符 “===” 或 “==”的区别?

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

```
+0 === -0 // true
Object.is(+0, -0) // false
```

9.什么是 JavaScript 中包装类型?

```
const a = 'abc'
a.length; // 3
a.toUpperCase(); // "ABC"
Object(a) // String {"abc"}
Object(a).valueOf() // 'abc'
```

- 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。
- 在访问 a.length 时，JavaScript 将`'abc'`在后台转换成`String('abc')`，然后再访问其`length`属性
- 可以使用`Object`函数显式地将基本类型转换为包装类型
- 可以使用`valueOf`方法将包装类型倒转成基本类型

10.object.assign和扩展运算法是深拷贝还是浅拷贝，两者区别

```
let outObj = {
    inObj: {a: 1, b: 2}
}
let newObj = {...outObj}
let newObjAs = Object.assign({}, outObj)
newObj.inObj.a = 2
newObjAs.inObj.b = 5
console.log(outObj) // {inObj: {a: 2, b: 5}}
```

- 两者都是浅拷贝
- Object.assign()方法接收的第一个参数作为目标对象，后面的所有参数作为源对象。然后把所有的源对象合并到目标对象中。它会修改了一个对象，因此会触发 ES6 setter。
- 扩展操作符（...）使用它时，数组或对象中的每一个值都会被拷贝到一个新的数组或对象中。它不复制继承的属性或类的属性，但是它会复制ES6的 symbols 属性。

11. 如何判断一个对象是空对象

```
// 1.使用 JSON.stringify 方法来判断
console.log(JSON.stringify({}) === '{}')
// Object.keys()
console.log(Object.keys({})) // []
console.log(Object.keys({}).length) // 0
```

二、ES6

12.let、const、var的区别

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	✔	✔	×

13. 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，而是变量指向的那个内存地址不能改动。对于基本类型的数据（数值、字符串、布尔值），其值就保存在变量指向的那个内存地址，因此等同于常量。
- 但对于引用类型的数据（主要是对象和数组）来说，变量指向数据的内存地址，保存的只是一个指针，const只能保证这个指针是固定不变的，至于它指向的数据结构是不是可变的，就完全不能控制了。

14. 箭头函数与普通函数的区别

1. **箭头函数比普通函数更加简洁**
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**
4. **call()、apply()、bind()等方法不能改变箭头函数中this的指向**
5. **箭头函数不能作为构造函数使用**
6. **箭头函数没有自己的arguments**
7. **箭头函数没有prototype**
8. **箭头函数没有自己的arguments**

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

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。

15.扩展运算符的作用及使用场景

15.1 对象扩展运算符

```
// 对象的扩展运算符
let bar = {
    a: 1,
    b: 2
}
console.log({...bar})
```

- 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。

15.2 数组的扩展运算函数

```
// 数组的扩展运算符
const arr = [1,2,3]
console.log(...arr) // 1,2,3
```

- 可以将一个数组**转为用逗号分隔的参数序列**，且每次只能展开一层数组

```
// 作用--复制数组
const arrCopy = [...arr]
console.log(arrCopy) // [ 1, 2, 3 ]
// 作用--合并数组
const arr1 = ['one', 'two', ...arr]
console.log(arr1) // [ 'one', 'two', 1, 2, 3 ]
// 与解构赋值结合，用于生成数组
const [first, ...rest] = arr1
console.log(first) // one
console.log(rest) // [ 'two', 1, 2, 3 ]
```

16. Proxy 可以实现什么功能?

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

```
/**
 * obj: 源对象
 * setBind: set回调
 * getLogger: get 回调
 */
let onWatch = (obj, setBind, getLogger) => {
    let handler = {
        set (target, property, value, receiver) {
            setBind(target, property, value, receiver)
            return Reflect.set(target, property, value)
        },
        get (target, property, receiver) {
            getLogger(target, property, receiver)
```

```
            return Reflect.get(target, property, receiver)
        }
    }
    return new Proxy(obj, handler)
}

let obj = {
    a: 1
}
let p = onWatch(
    obj,
    (target, property, value, receiver) => {
        console.log(target, property, value, receiver) // { a: 1 } a 2 { a: 1 }
    },
    (target, property, receiver) => {
       console.log(target, property, receiver) // { a: 2 } a { a: 2 }
    }
)

p.a = 2
p.a
```

17. 对象与数组的解构

```
// 数组解构
const [a, b, c] = [1, 2, 3]

// 对象解构
const stu = {
    name: 'well',
    age: 15
}
const { name } = stu
```

18.

扩展运算符被用在函数形参上时，**它还可以把一个分离的参数序列整合成一个数组**

```
const mutiple = (...args) => {
    let result = 1
    for (let value of args) {
        result *= value
    }
    return result
}

console.log(mutiple(1,2,3,4)) // 24
```

三、this/call/apply/bind

19. 对this 对象的理解

- 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. 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。
2. 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。
3. 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。
4. 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。

- apply、call、bind的区别?
  - apply接收的是数组，call、bind接受的非数组。
  - 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。

```
global.name = 'liuguowei'
global.age = 18

// 函数调用模式
const sayName = function () {
    return this.name
}
console.log('函数调用模式:', sayName()) // 函数调用模式: liuguowei

// 方法调用模式
const nameObj = {
    name: 'well',
    sayName () {
        return this.name
    }
}
console.log('方法调用模式:', nameObj.sayName()) // 方法调用模式: well

// 构造器调用模式
function AgeFun (age) {
    this.age = age
    this.sayAge = function () {
        return this.age
    }
}
const ageInstance = new AgeFun(25)
console.log('构造器调用模式:', ageInstance.sayAge()) // 构造器调用模式: 25
```

20. apply、call、bind

```
const foo = {
    a: 1,
    fn (x, y) {
        console.log(this.a, x, y)
    }
}

const obj = {
    a: 10
}
```

```
// apply
foo.fn.apply(obj, [2, 3]) // 10 2 3
// call
foo.fn.call(obj, 2, 3) // 10 2 3
// bind
foo.fn.bind(obj, 2, 3)() // 10 2 3
```

21. call 函数的实现及步骤

1. 判断调用对象是否为函数
2. 获取参数
3. 判断 context 是否传入，如果不传入则设置为 window
4. 将函数设为此对象的方法（把调用函数作为传入对象的属性）
5. 调用函数
6. 返回结果

```
Function.prototype.MyCall = function (context) {
    console.log('this>>>', this) // this>>> [Function: log]
    // 判断调用对象
    if (typeof this !== 'function') {
        console.error('type error')
    }
    // 获取参数
    const 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
    // 将调用函数设为此对象的方法
    context.fn = this
    // 调用函数
    result = context.fn(...args)
    // 将属性删除
    delete context.fn
    return result
}

const foo = {
    a: 1,
    log (x, y) {
        console.log(this.a, x, y)
    }
}
const obj = {
    a: 10
}

foo.log.MyCall(obj, 5, 6) // 10, 5, 6
```

22. apply 函数的实现

```
Function.prototype.MyApply = function (context) {
    // 判断调用对象是否为函数
    if (typeof this !== 'function') {
        throw new TypeError('Error')
    }
```

```
    let result = null
    // 判断 context 是否传入，如果为传入则设置window
    context = context || window
    context.fn = this
    // 判断是否有传参
    if (arguments[1]) {
        result = context.fn(...arguments[1])
    } else {
        result = context.fn()
    }
    // 将属性删除
    delete context.fn
    return result
}

const foo = {
    a: 1,
    log (x, y) {
        console.log(this.a, x, y)
    }
}
const obj = {
    a: 10
}

foo.log.MyApply(obj, [5, 6]) // 10, 5, 6
```

23. bind 函数的实现及步骤

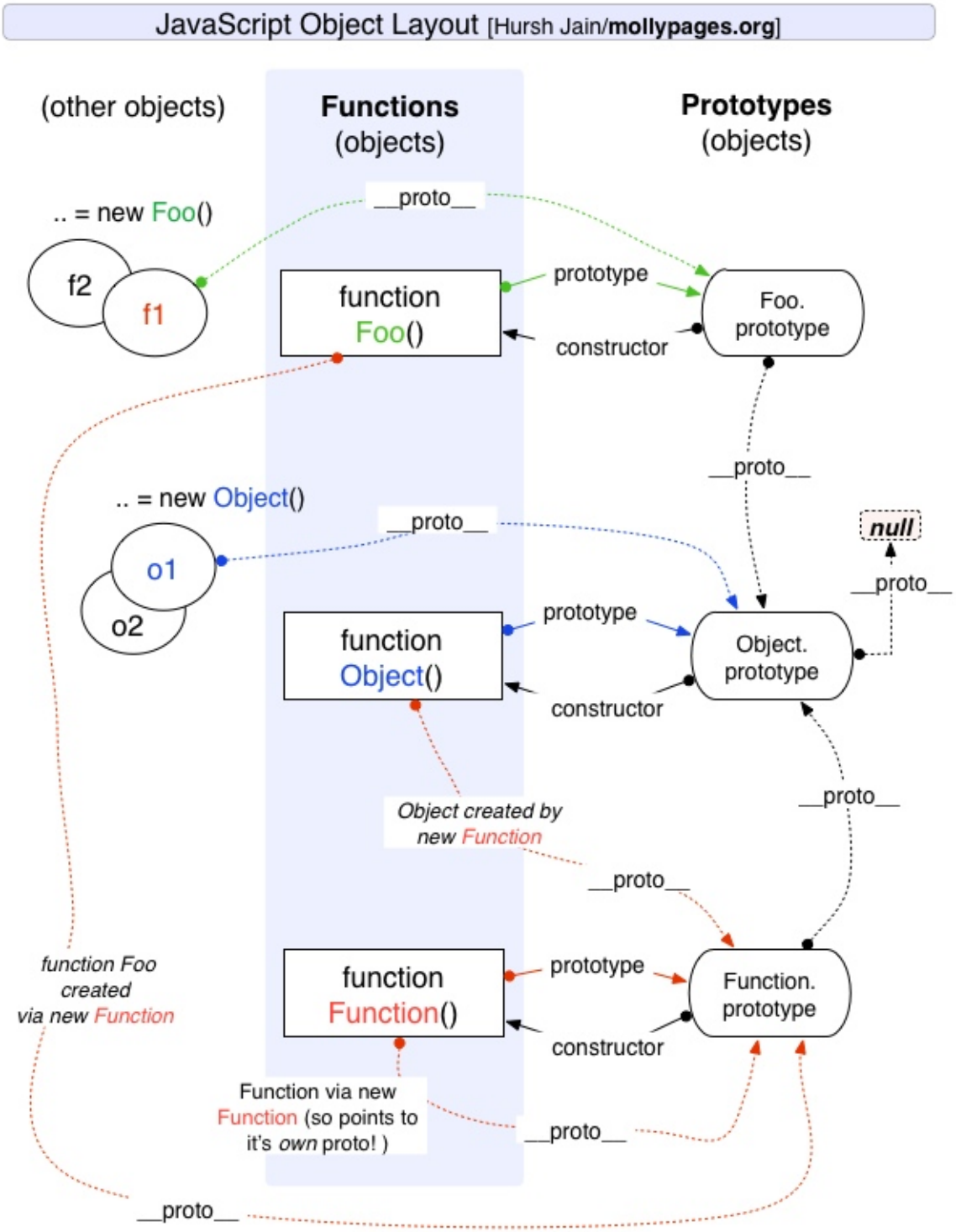
```
Function.prototype.myBind = function (context) {
    // 判断调用对象是否为函数
    if (typeof this !== 'function') {
        throw new TypeError('Error')
    }
    // 获取参数
    const args = [...arguments].slice(1)
    const fn = this
    return function Fn () {
        // 根据调用方式，传入不同绑定值
        return fn.apply(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

const foo = {
    a: 1,
    log (x, y) {
        console.log(this.a, x, y)
    }
}
const obj = {
    a: 10
}

foo.log.myBind(obj, 5, 6)() // 10, 5, 6
```

四、原型与原型链

24.对原型链的理解

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。



- 每个构造函数都有一个prototype属性，其指向对象的原型。
- 对象的原型包含了可以由该构造函数的所有实例共享的属性和方法。而且其还有一个constructor属性，其指向对应的构造函数。
- 通过构造函数实例化一个对象后，这个对象包含 一个指针`__proto__:非标准，由浏览器实现`，指向构造函数的 prototype 属性对应的值，即对象的原型。
- 原型链的尽头一般来说都是 Object.prototype(对象)来获取。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
```

```
</head>
<body>
    <script>
        function Person (age) {
            this.age = age
        }
        const well = new Person(18)
        /**
         * {constructor: ƒ}
            constructor: ƒ Person(age)
            [[Prototype]]: Object
         */
        console.log(Object.getPrototypeOf(well))
        /**
         * {constructor: ƒ}
            constructor: ƒ Person(age)
            [[Prototype]]: Object
         */
        console.log(well.__proto__)
        /**
         * {constructor: ƒ}
            constructor: ƒ Person(age)
            [[Prototype]]: Object
         */
        console.log(Person.prototype)
    </script>
</body>
</html>
```

25. 原型链指向

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        function Person(name) {
            this.name = name
        }
        const p = new Person('well')
        console.log('p.__proto__>>>', p.__proto__) // // Person.prototype
        console.log('p.__proto__.constructor>>>', p.__proto__.constructor) // f
Person(name)
        console.log('Person.prototype.__proto__>>>', Person.prototype.__proto__)
// Object.prototype
        console.log('p.__proto__.__proto__>>>', p.__proto__.__proto__) // //
Object.prototype
        console.log('p.__proto__.constructor.prototype.__proto__>>>',
p.__proto__.constructor.prototype.__proto__) // Object.prototype
        console.log('Person.prototype.constructor.prototype.__proto__>>>',
Person.prototype.constructor.prototype.__proto__) // Object.prototype
```

```
        console.log('Person.prototype.constructor>>>',
Person.prototype.constructor) // f Person(name)
    </script>
</body>
</html>
```

26.原型链的终点是什么？null

```
console.log(Object.prototype.__proto__) //
```

27.如何获得对象非原型链上的属性?

`hasOwnProperty()` 方法会返回一个布尔值，指示对象自身属性中是否具有指定的属性（也就是，是否有指定的键）。

```
function iterate(obj) {
    const res = []
    for (let key in obj) {
        if (obj.hasOwnProperty(key)) {
            res.push(`${key}:${obj[key]}`)
         }
     }
     return res
}
function Person (name) {
    this.name = name
}
console.log(iterate(new Person('well'))) // ['name:well']
```

五、闭包、作用域链、执行上下文

28. 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**
- 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。

```
const name = 'well'
const age = 25

function showName () {
    const name = 'liuguowei'
    return function () {
        return name
    }
}
console.log(showName()()) // liuguowei
```

```
function myAge () {
    return age
}
function showAge (fn) {
    const age = 18
    return fn()
}
console.log(showAge(myAge)) // 25
```

29.对作用域、作用域链的理解

- 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。
- 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
- 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。

30.对执行上下文的理解

- 执行上下文类型
  - **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。
  - **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。
  - **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。
- 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。

六、JavaScript 基础

31. new 操作符的实现原理

new 操作符的执行过程

- 首先创建一个新的空对象
- 设置原型，将对象的原型设置为函数的prototype 对象。（实例的原型执行构造函数的原型）
- 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）
- 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。

```
function objectFactory () {
    // 声明要创建的对象
    let newObject = null
    // 取出构造函数
    let constructor = Array.prototype.shift.call(arguments)
    console.log('constructor>>>', constructor) // constructor>>> [Function:
Person]
    let result = null
    // 判断参数是否是函数
    if (typeof constructor !== 'function') {
        throw TypeError('error')
    }
    // 新建一个空对象，对象的原型为构造函数的 prototype 对象
    newObject = Object.create(constructor.prototype)
    console.log('newObject>>>', newObject) // newObject>>> Person {}
    // 将 this 指向新建对象，并指向函数
    result = constructor.apply(newObject, arguments)
    console.log('result>>>', result)
    console.log('newObject>>>', newObject) // newObject>>> Person { name: 'well',
age: 18 }
    // 判断返回对象
    let flag = result && (typeof result === 'object' || typeof result ===
'function')
    // 判断返回结果
    return flag ? result : newObject
}

function Person (name, age) {
    this.name = name
    this.age = age
}
const well = objectFactory(Person, 'well', 18)
console.log(well.name)
```

32.map

1. Map 与 object 的区别?

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

2. Map

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

```
const map = new Map()
// map.set(key, value)
map.set('a', 1).set({a: 1}, 2)
// map.get(key)
console.log(map.get('a')) // 1
console.log(map.size) // 2
// map.has(key)
console.log(map.has('a')) // true
// map.delete(key)
console.log(map.delete('a')) // true
console.log(map) // Map(1) { { a: 1 } => 2 }
// map.clear()
map.clear()
console.log(map) // Map(0) {}

// 验证遍历器
```

```
map.set('foo', 1).set('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
})
```

3.WeakMap

- **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。

```
const weakmap = new WeakMap()
const a = {
    a: 1
}
const b = {
    b: 2
}
weakmap.set(a, 1).set(b, 2)
console.log(weakmap) // WeakMap { <items unknown> }
console.log(weakmap.get(a)) // 1
console.log(weakmap.has(a)) // true
weakmap.delete(a)
console.log(weakmap.has(a)) // false
```

- 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。

33. 为什么函数的 arguments 参数是类数组而不是数组？如果遍历类数组

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

```
function foo () {
    Array.prototype.forEach.call(arguments, a=> console.log(a)) // 1 2 3 4 5
}
function foo1 () {
    const arrArgs = Array.from(arguments)
    console.log(arrArgs) // [ 1, 2, 3, 4, 5 ]
```

```
}
function foo2 () {
    const arrArgs = [...arguments]
    console.log(arrArgs) // [ 1, 2, 3, 4, 5 ]
}

foo(1,2,3,4,5)
foo1(1,2,3,4,5)
foo2(1,2,3,4,5)
```

34. XMLHttpRequest

```
const SERVER_URL = 'http://localhost:3000/todos'
const xhr = new XMLHttpRequest()
// 创建 Http 请求
xhr.open('GET', SERVER_URL, true)
// 设置状态监听函数
xhr.onreadystatechange = function () {
    if (this.readyState !== 4) return
    // 当请求成功时
    if (this.status === 200) {
        console.log(this.response)
    } else {
        console.error(this.statusText)
    }
}
// 设置请求失败时的监听函数
xhr.onerror = function () {
    console.error(this.statusText)
}
// 设置请求头信息
xhr.responseType = 'json'
xhr.setRequestHeader('Accept', 'application/json')
xhr.send()
```

35 XMLHttpRequest Promise

```
function getJSON (url) {
    let promise = new Promise(function (resolve, reject) {
        const xhr = new XMLHttpRequest()
        // 创建 Http 请求
        xhr.open('GET', url, true)
        // 设置状态监听函数
        xhr.onreadystatechange = function () {
        if (this.readyState !== 4) return
            // 当请求成功时
            if (this.status === 200) {
                resolve(this.response)
            } else {
                reject(new Error(this.statusText))
            }
        }
        // 设置请求失败时的监听函数
        xhr.onerror = function () {
            reject(new Error(this.statusText))
```

```
        }
        // 设置请求头信息
        xhr.responseType = 'json'
        xhr.setRequestHeader('Accept', 'application/json')
        xhr.send()
    })
    return promise
}
const SERVER_URL = 'http://localhost:3000/todos'
getJSON(SERVER_URL)
```

七、异步编程

36.对 Promise 的理解

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。

- Promise 实例有三个状态
  - Pendding (进行中)
  - Resolved(已完成)
  - Rejected(已拒绝)
  - 当把一件事交给promise时，它的状态就是 Pending，任务完成了状态就成了Resolved，失败就成了Rejected。
- Promise的实例有两个过程
  - pending -> fulfilled: Resolved(已完成)
  - pending -> rejected: Rejected(已拒绝)
  - 注意：旦从进行状态变成为其他状态就永远不能更改状态了。
- Promise 的缺点
  - 无法取消 Promise，旦新建它就会立即执行，无法中途取消
  - 旦新建它就会立即执行，无法中途取消
  - 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）。

37.Promise 的 基本用法

- then、catch、finally
- all：全部resolve 才 执行then，否则执行 catch
- race：“竞赛”，最先resolve的作为then，使用场景：当一件事超过一定时间就不做了

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

```
// new Promise() promise.then() promise.catch
function getData (success) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (success) {
                resolve(1)
            } else {
                reject(0)
            }
        })
    })
}
```

```
getData(1).then((res) => console.log('promise.then()>>>', res)) //
promise.then()>>> 1
getData().then((res) => console.log(res)).catch(err =>
console.log('promise.catch()>>>', err)) // promise.catch()>>> 0
// promise.all()、 promise.race()
let promise1 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
       resolve(1);
    },2000)
});
let promise2 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
       resolve(2);
    },1000)
});
let promise3 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
       resolve(3);
    },3000)
});
Promise.all([promise1,promise2,promise3]).then(res=>{
    console.log('promise.all>>>', res); // promise.all>>> [ 1, 2, 3 ]
})
Promise.race([promise1,promise2,promise3]).then(res=>{
    console.log('promise.race>>>', res); // promise.race>>> 2
})
```

38. async & await

1. async 函数

- async 函数返回的是一个 Promise 对象。
- 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，绝不会阻塞后面的语句。
- async 的返回值会作为：Promise.resolve(返回值)，如果没有返回值则相当于：Promise.resolve(undefined)

```
// async 返回的是什么？
const testAsync = async () => {
    return 'hello world'
}
let result = testAsync()
console.log('async 返回值>>>', result) // Promise { 'hello world' }
result.then(res => console.log(res)) // hello world
```

2.await 在等待什么?

- await 等待的是一个表达式，这个表达式的计算结果是Promise对象或者其他值（即没有特殊限定）
- 如果它等到不是一个Promise对象，那么 await 表达式的运算结果就是它等到的东西
- 如果它等到的是一个Promise对象，await 就忙起来了，它会阻塞后面的代码，等着 Promise 对象 resolve，然后得到 resolve 的值，作为 await 表达式的运算结果。

```
// await 在等待什么？
function getSomething () {
```

```
    console.log('await 会阻塞后面的代码，执行外面的同步代码')
    return 'something'
}
async function asyncFn () {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve('hello async')
        }, 3000)
    })
}
async function test () {
    const v1 = await getSomething()
    const v2 = await asyncFn()
    console.log('await 等到非Promise>>>', v1) // await 等到非Promise>>> something
    console.log('await 等到Promise>>>', v2) // await 等到Promise>>> hello async
}
test()
console.log('不在async 函数里的不会被阻塞') // 不在async 函数里的不会被阻塞
```

输出顺序

- await 会阻塞后面的代码，执行外面的同步代码
- 不在async 函数里的不会被阻塞
- await 等到非Promise>>> something
- await 等到Promise>>> hello async

遇到await会阻塞后面的代码，先执行async外面的同步代码，同步代码执行完，再回到async内部，继续执行await后面的代码。

await 后面的代码进入了微任务队列

1. 先进入 test()函数，进入getSomething, 打印**await 会阻塞后面的代码，执行外面的同步代码**
2. 阻塞后面的代码，执行async外面的代码，打印**不在async 函数里的不会被阻塞**
3. ……

3.深刻理解await

```
function sumTimeOUt  (a,b) {
  console.log('timeoutStart')
  setTimeout(() => {
    console.log('timeoutEnd')
    return a + b
  })
}

async function testAsync () {
  console.log('start')
  const ret = await sumTimeOUt(1,2)
  console.log(ret)
  console.log('await end')
}
testAsync()
console.log('end')
```

- start
- timeoutStart

- end
- undefined
- await end
- timeoutEnd

```
function sumTimeOUt  (a,b) {
  console.log('timeoutStart')
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('timeoutEnd')
      resolve(a+b)
    })
  })
}

async function testAsync () {
  console.log('start')
  const ret = await sumTimeOUt(1,2)
  console.log(ret)
  console.log('await end')
}
testAsync()
console.log('end')
```

- start
- timeoutStart
- end
- timeoutEnd
- 3
- await end

结论：

- 如果它等到的不是一个 Promise 对象，那 await 表达式的运算结果就是它等到的东西。
- 如果它等到的是一个 Promise 对象，await 就忙起来了，它会阻塞后面的代码，等着 Promise 对象 resolve，然后得到 resolve 的值，作为 await 表达式的运算结果。

40.async await 魔鬼细节

```
async function async1 () {
    await new Promise((resolve, reject) => {
        resolve()
    })
    console.log('A')
}

async1()

new Promise((resolve) => {
    console.log('B')
    resolve()
}).then(() => {
    console.log('C')
}).then(() => {
```

```
    console.log('D')
})

// B A C D
```

```
async function async1 () {
    await async2()
    console.log('A')
}

async function async2 () {
    return new Promise((resolve, reject) => {
        resolve()
    })
}

async1()

new Promise((resolve) => {
    console.log('B')
    resolve()
}).then(() => {
    console.log('C')
}).then(() => {
    console.log('D')
})

// B C D A
```

async 函数返回值

在讨论 `await` 之前，先聊一下 `async` 函数处理返回值的问题，它会像 `Promise.prototype.then` 一样，会对返回值的类型进行辨识。

根据返回值的类型，引起 `js引擎` 对返回值处理方式的不同

结论： `async` 函数在抛出返回值时，会根据返回值**类型**开启**不同数目的微任务**

- return结果值：非 `thenable`、非 `promise`（不等待）
- return结果值：`thenable`（等待 1个 `then` 的时间）
- return结果值：`promise`（等待 2个 `then` 的时间）

非`thenable`、非`promise`（不等待）

```
async function testA () {
    return 1;
}

testA().then(() => console.log(1));
Promise.resolve()
    .then(() => console.log(2))
    .then(() => console.log(3));

// (不等待)最终结果👉: 1 2 3
```

`thenable` (等待 1个`then`的时间)

```
async function testB () {
    return {
        then (cb) {
            cb();
        }
    };
}

testB().then(() => console.log(1));
Promise.resolve()
    .then(() => console.log(2))
    .then(() => console.log(3));

// (等待一个then)最终结果👉: 2 1 3
```

`promise` (等待 2个`then`的时间)

```
async function testC () {
    return new Promise((resolve, reject) => {
        resolve()
    })
}

testC().then(() => console.log(1));
Promise.resolve()
    .then(() => console.log(2))
    .then(() => console.log(3))
    .then(() => console.log(4))

// (等待两个then)最终结果👉: 2 3 1 4
```

await 右值类型区别

非`thenable`

```
async function test () {
    console.log(1);
    await 1;
    console.log(2);
}

test();
console.log(3);
// 最终结果👉: 1 3 2
```

```
function func () {
    console.log(2);
}

async function test () {
    console.log(1);
    await func();
```

```
    console.log(3);
}

test();
console.log(4);

// 最终结果👉: 1 2 4 3
```

```
async function test () {
    console.log(1);
    await 123
    console.log(2);
}

test();
console.log(3);

Promise.resolve()
    .then(() => console.log(4))
    .then(() => console.log(5))
    .then(() => console.log(6))
    .then(() => console.log(7));

// 最终结果👉: 1 3 2 4 5 6 7
```

`await`**后面接非** `thenable` **类型，会立即向微任务队列添加一个微任务** `then`**，但不需等待**

`thenable`类型

```
async function test () {
    console.log(1);
    await {
        then (cb) {
            cb();
        },
    };
    console.log(2);
}

test();
console.log(3);

Promise.resolve()
    .then(() => console.log(4))
    .then(() => console.log(5))
    .then(() => console.log(6))
    .then(() => console.log(7));

// 最终结果👉: 1 3 4 2 5 6 7
```

`await` **后面接** `thenable` **类型，需要**等待一个 `then` 的时间之后**执行**

`Promise`类型

```
async function test () {
    console.log(1);
    await new Promise((resolve, reject) => {
        resolve()
    })
    console.log(2);
}

test();
console.log(3);

Promise.resolve()
    .then(() => console.log(4))
    .then(() => console.log(5))
    .then(() => console.log(6))
    .then(() => console.log(7));

// 最终结果👉: 1 3 2 4 5 6 7
```

为什么表现的和非 `thenable` 值一样呢？为什么不等待两个 `then` 的时间呢？

- TC 39(ECMAScript标准制定者) 对`await` 后面是 `promise` 的情况如何处理进行了一次修改，**移除**了额外的两个微任务，在**早期版本**，依然会等待两个 `then` 的时间
- 有大佬翻译了官方解释：**更快的 async 函数和 promises**，但在这次更新中并没有修改 `thenable` 的情况

这样做可以极大的优化 `await` 等待的速度👇

```
async function func () {
    console.log(1);
    await 1;
    console.log(2);
    await 2;
    console.log(3);
    await 3;
    console.log(4);
}

async function test () {
    console.log(5);
    await func();
    console.log(6);
}

test();
console.log(7);

Promise.resolve()
    .then(() => console.log(8))
    .then(() => console.log(9))
    .then(() => console.log(10))
    .then(() => console.log(11));

// 最终结果👉: 5 1 7 2 8 3 9 4 10 6 11
```

`await` 和 `Promise.prototype.then` 虽然很多时候可以在**时间顺序**上能等效，但是它们之间有**本质的区别**。

- `test` 函数中的 `await` 会等待 `func` 函数中所有的 `await` 取得 恢复函数执行 的命令并且整个函数执行完毕后才能获得取得 **恢复函数执行**的命令；
- 也就是说，`func` 函数的 `await` 此时**不能在时间的顺序上等效** `then`，而要等待到 `test` 函数完全执行完毕；
- 比如这里的数字 `6` 很晚才输出，**如果**单纯看成 `then` 的话，在下一个微任务队列执行时 `6` 就应该作为同步代码输出了才对。

所以我们可以合并两个函数的代码👇

```
async function test () {
    console.log(5);

    console.log(1);
    await 1;
    console.log(2);
    await 2;
    console.log(3);
    await 3;
    console.log(4);
    await null;

    console.log(6);
}

test();
console.log(7);

Promise.resolve()
    .then(() => console.log(8))
    .then(() => console.log(9))
    .then(() => console.log(10))
    .then(() => console.log(11));

// 最终结果👉: 5 1 7 2 8 3 9 4 10 6 11
```

因为将原本的函数融合，此时的 `await` 可以等效为 `Promise.prototype.then`，又完全可以等效如下代码👇

```
async function test () {
    console.log(5);
    console.log(1);
    Promise.resolve()
        .then(() => console.log(2))
        .then(() => console.log(3))
        .then(() => console.log(4))
        .then(() => console.log(6))
}

test();
console.log(7);

Promise.resolve()
    .then(() => console.log(8))
```

```
    .then(() => console.log(9))
    .then(() => console.log(10))
    .then(() => console.log(11));

// 最终结果👉: 5 1 7 2 8 3 9 4 10 6 11
```

以上三种写法在时间的顺序上完全等效，所以你 **完全可以将 `await` 后面的代码可以看做在 `then` 里面执行的结果**，又因为 `async` 函数会返回 `promise` 实例，所以还可以等效成👇

```
async function test () {
    console.log(5);
    console.log(1);
}

test()
    .then(() => console.log(2))
    .then(() => console.log(3))
    .then(() => console.log(4))
    .then(() => console.log(6))

console.log(7);

Promise.resolve()
    .then(() => console.log(8))
    .then(() => console.log(9))
    .then(() => console.log(10))
    .then(() => console.log(11));

// 最终结果👉: 5 1 7 2 8 3 9 4 10 6 11
```

真题演示

1.方式一

```
async function async2 () {
    new Promise((resolve, reject) => {
        resolve()
    })
}

async function async3 () {
    return new Promise((resolve, reject) => {
        resolve()
    })
}

async function async1 () {
    // 方式一：最终结果：B A C D
    // await new Promise((resolve, reject) => {
    //     resolve()
    // })

    // 方式二：最终结果：B A C D
    // await async2()

    // 方式三：最终结果：B C D A
```

```
    await async3()

    console.log('A')
}

async1()

new Promise((resolve) => {
    console.log('B')
    resolve()
}).then(() => {
    console.log('C')
}).then(() => {
    console.log('D')
})
```

- 首先，`async`**函数的整体返回值永远都是**`Promise`**，无论值本身是什么**
- 方式一：`await`的是`Promise`，无需等待
- 方式二：`await`的是`async`函数，但是该函数的返回值本身是**非**`thenable`，无需等待
- 方式三：`await`的是`async`函数，且返回值本身是`Promise`，需等待两个`then`时间

2.方式二

```
function func () {
    console.log(2);

    // 方式一：1 2 4  5 3 6 7
    // Promise.resolve()
    //     .then(() => console.log(5))
    //     .then(() => console.log(6))
    //     .then(() => console.log(7))

    // 方式二：1 2 4  5 6 7 3
    return Promise.resolve()
        .then(() => console.log(5))
        .then(() => console.log(6))
        .then(() => console.log(7))
}

async function test () {
    console.log(1);
    await func();
    console.log(3);
}

test();
console.log(4);
```

- 方式一：
- - 同步代码输出`1、2`，接着将`log(5)`处的`then1`加入微任务队列，`await`拿到确切的`func`函数返回值`undefined`，将后续代码放入微任务队列（`then2`，可以这样理解）
  - 执行同步代码输出`4`，到此，所有同步代码完毕
  - 执行第一个放入的微任务`then1`输出`5`，产生`log(6)`的微任务`then3`
  - 执行第二个放入的微任务`then2`输出`3`

    - 然后执行微任务`then3`，输出`6`，产生`log(7)`的微任务`then4`
    - 执行`then4`，输出`7`
- 方式二：
-
    - 同步代码输出`1、2`，`await`拿到`func`函数返回值，但是并未获得**具体的结果**（由`Promise`本身机制决定），暂停执行当前`async`函数内的代码（跳出、让行）
    - 输出`4`，到此，所有同步代码完毕
    - `await`一直等到`Promise.resolve().then...`执行完成，再放行输出`3`

3.方式三

```
function func () {
    console.log(2);

    return Promise.resolve()
        .then(() => console.log(5))
        .then(() => console.log(6))
        .then(() => console.log(7))
}

async function test () {
    console.log(1);
    await func()
    console.log(3);
}

test();
console.log(4);

new Promise((resolve) => {
    console.log('B')
    resolve()
}).then(() => {
    console.log('C')
}).then(() => {
    console.log('D')
})

// 最终结果👉: 1 2 4    B 5 C 6 D 7 3
```

```
async function test () {
    console.log(1);
    await Promise.resolve()
        .then(() => console.log(5))
        .then(() => console.log(6))
        .then(() => console.log(7))
    console.log(3);
}

test();
console.log(4);

new Promise((resolve) => {
    console.log('B')
    resolve()
}).then(() => {
```

```
    console.log('C')
}).then(() => {
    console.log('D')
})

// 最终结果👉: 1 4    B 5 C 6 D 7 3
```

综上，`await`一定要等到右侧的表达式有**确切的值**才会放行，否则将一直等待（阻塞当前`async`函数内的后续代码），不服看看这个👇

```
function func () {
  return new Promise((resolve) => {
      console.log('B')
      // resolve() 故意一直保持pending
  })
}

async function test () {
  console.log(1);
  await func()
  console.log(3);
}

test();
console.log(4);
// 最终结果👉: 1 B 4 (永远不会打印3)


// ---------------------或者写为👇--------------------
async function test () {
  console.log(1);
  await new Promise((resolve) => {
      console.log('B')
      // resolve() 故意一直保持pending
  })
  console.log(3);
}

test();
console.log(4);
// 最终结果👉: 1 B 4 (永远不会打印3)
```

4.方式四

```
async function func () {
    console.log(2);
    return {
        then (cb) {
            cb()
        }
    }
}

async function test () {
    console.log(1);
    await func();
```

```
    console.log(3);
}

test();
console.log(4);

new Promise((resolve) => {
    console.log('B')
    resolve()
}).then(() => {
    console.log('C')
}).then(() => {
    console.log('D')
})

// 最终结果👉: 1 2 4 B C 3 D
```

- 同步代码输出`1、2`
- `await`拿到`func`函数的具体返回值`thenable`，将当前`async`函数内的后续代码放入微任务`then1`(但是需要等待一个`then`时间)
- 同步代码输出`4、B`，产生`log(C)`的微任务`then2`
- 由于`then1`滞后一个`then`时间，直接执行`then2`输出`C`，产生`log(D)`的微任务`then3`
- 执行原本滞后一个`then`时间的微任务`then1`，输出`3`
- 执行最后一个微任务`then3`输出`D`

5.经典面试题

```
async function async1 () {
    console.log('1')
    await async2()
    console.log('AAA')
}

async function async2 () {
    console.log('3')
    return new Promise((resolve, reject) => {
        resolve()
        console.log('4')
    })
}

console.log('5')

setTimeout(() => {
    console.log('6')
}, 0);

async1()

new Promise((resolve) => {
    console.log('7')
    resolve()
}).then(() => {
    console.log('8')
}).then(() => {
    console.log('9')
```

```
}).then(() => {
    console.log('10')
})
console.log('11')

// 最终结果👉: 5 1 3 4 7 11 8 9 AAA 10 6
```

步骤拆分👇:

1. 先执行同步代码，输出`5`
2. 执行`setTimeout`，是放入宏任务异步队列中
3. 接着执行`async1`函数，输出`1`
4. 执行`async2`函数，输出`3`
5. `Promise`构造器中代码属于同步代码，输出`4`

   > `async2`函数的返回值是`Promise`，等待`2`个`then`后放行，所以`AAA`暂时无法输出

6. `async1`函数**暂时**结束，继续往下走，输出`7`
7. 同步代码，输出`11`
8. 执行第一个`then`，输出`8`
9. 执行第二个`then`，输出`9`
10. 终于**等**到了两个`then`执行完毕，执行`async1`函数里面剩下的，输出`AAA`
11. 再执行最后一个微任务`then`，输出`10`
12. 执行最后的宏任务`setTimeout`，输出`6`

总结

`async`函数返回值

- 📝结论：`async`函数在抛出返回值时，会根据返回值**类型**开启**不同数目的微任务**
- - return结果值：非`thenable`、非`promise`（不等待）
  - return结果值：`thenable`（等待 1个`then`的时间）
  - return结果值：`promise`（等待 2个`then`的时间）

`await`右值类型区别

- 接非`thenable`类型，会立即向微任务队列添加一个微任务`then`，**但不需等待**
- 接`thenable`类型，需要**等待一个`then`的时间之后**执行
- 接`Promise`类型(有确定的返回值)，会立即向微任务队列添加一个微任务`then`，**但不需等待**
- - TC 39 对`await`后面是`promise`的情况如何处理进行了一次修改，**移除**了额外的两个微任务，在**早期版本**，依然会等待两个`then`的时