

# 关于JavaScript的一些问题

## JS和nodeJS执行环境问题

nodeJs没有window对象，document对象，因为nodejs是一个 `JavaScript runtime built on Chrome's V8 JavaScript engine.` 详细参考：[前端是什么 > 浏览网页做了什么。](#)

BOM (browser object model) 是浏览器对象模型，提供与浏览器交互的方法和接口；DOM (document object model) 是浏览器文档对象模型，处理网页内容的方法和接口

详情参考：[web-api](#)

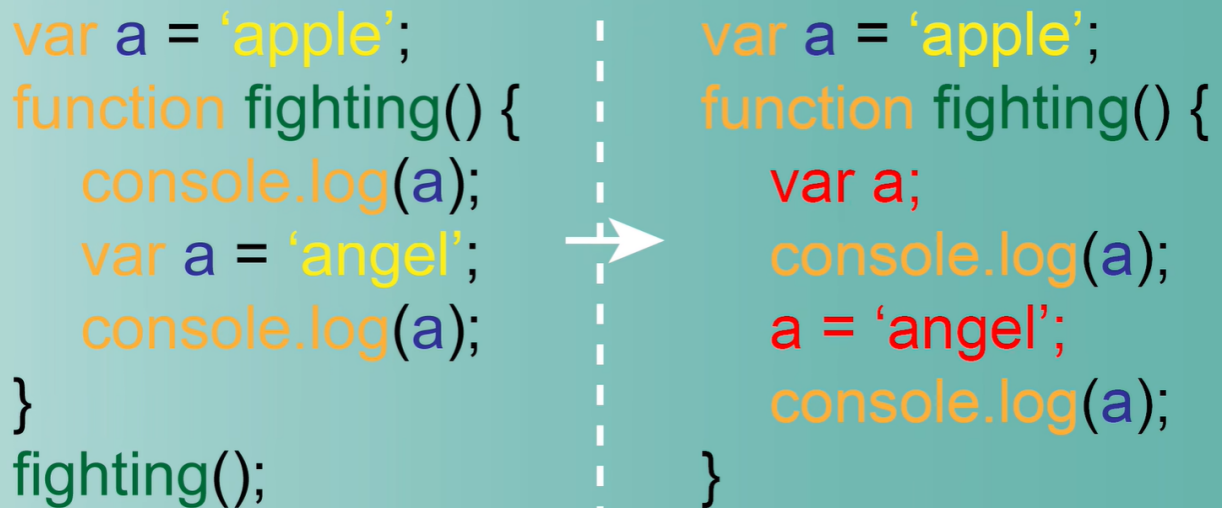
- 如果使用window对象需要创建一个
- 没有document对象，使用jsdom之类的库

## 基础

## 变量声明

- JavaScript变量申明可以提升
- let 声明一个块作用域中的局部变量，不允许重复声明
- var 声明一个变量，允许重复声明
- 当声明变量类型时，可以使用关键词 "new" 来声明其类型

- let定义的变量没有赋值之前是不可以使用、var可以使用是undefined



```
var a = 'apple';
function fighting() {
  console.log(a);
  var a = 'angel';
  console.log(a);
}
fighting();
```

```
var a = 'apple';
function fighting() {
  var a;
  console.log(a);
  a = 'angel';
  console.log(a);
}
```

## 操作符优先级

[https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)

## 弱类型语言--类型转换问题 隐式转换

某些运算符被执行时，系统内部自动将数据类型进行转换，这种转换称为隐式转换。

比如：

- 用+, -, -0运算转换：如+str, -str, str-0将字符串转为数值（隐式），做Number()操作
- !obj（隐式）
  - 除空字符串, null, undefined, NaN, 0 会被转为false，其他都会被转成true。注意空对象不是False

## 显式转换

- 转换为字符串类型
  - obj.toString()
- 转换为数值类型
  - Number(obj)：可以将任何值转为数值，但是必须obj中全是数值。

- parseInt(obj): 转换字符串中的整数，到小数点或者字符串结束，如果字符串开头，就会变成NaN。
- parseFloat(obj): 转换字符串中的小数，到第二个小数点或字符串结束。
- 转换为布尔类型
  - Boolean(obj)

## 函数的作用域

函数提升与变量提升比较类似，是指函数在声明之前即可被调用。注意：  
函数表达式不存在提升现象

```
function foo() {  
  console.log(a);  
  var a = 1;  
  function a() {}  
}  
foo();  
// f a() {} 假如函数名和变量名重复，函数的优先级比变量的高  
  
function foo() {  
  console.log(a);  
  var a = 1;  
  a()  
  function a() {console.log(a)}  
}  
foo(); // 报错 a 不是函数
```

```
var a = 'apple';
function fighting() {
  console.log(a);
  var a = 'angel';
  console.log(ss());
  function ss() {
    return a;
  }
}
fighting();
```



```
var a = 'apple';
function fighting() {
  function ss() {
    return a;
  }
  var a;
  console.log(a);
  a = 'angel';
  console.log(ss());
}
```

```
a() // 成功
```

```
b()
```

```
c()
```

```
e()
```

```
function a(){
  console.log('a')
};
```

```
var b = function(){
  console.log('b')
};
```

```
var c = function(){
  console.log('c')
};
```

```
var e = () => {
  console.log('e')
```

```
}
```

- 全局作用域
  - 作用域所有代码执行的环境
- 局部作用域
  - 作用域函数内的环境
- 块级作用域
  - `{ }` 包含的, 包括if, for

## 作用域链

作用域链本质上是底层的变量查找机制, 在函数被执行时, 会优先查找当前函数作用域中查找变量, 如果当前作用域查找不到则会依次逐级查找父级作用域直到全局作用域。

闭包是一种比较特殊和函数, 使用闭包能够访问函数作用域中的变量。从代码形式上看闭包是一个做为返回值的函数

```
function foo() {  
  
    let i = 0;  
  
    // 函数内部分函数  
    function bar() {  
        console.log(++i);  
    }  
  
    // 将函数做为返回值  
    return bar;  
}
```

```
// fn 即为闭包函数
```

```
let fn = foo();
```

```
fn();
```

# 箭头函数

注意：没有 `this`、`super`、`arguments` 和 `new.target` 绑定。

- 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。
- 箭头函数继承来的`this`指向永远不会改变
- 箭头函数不能作为构造函数使用
  - 由于箭头函数时没有自己的`this`，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。
- 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。

```
var obj = {  
  name: "xx",  
  show: function() {  
    console.log(this); //this表示当前对象  
  },  
  say: () => {  
    console.log(this); //this表示全局window对象  
  }  
}
```

```
var People = ()=>{  
    console.log(this);  
}  
  
obj.show()  
obj.say()  
var obj = {  
    birth: 1990,  
    getAge: function () {  
        var b = this.birth; // 1990  
        var fn = function () {  
            console.log(this.birth)  
            return new Date().getFullYear() - this.birth;  
        }  
        // this指向window或undefined  
        };  
        return fn();  
    }  
};  
  
x = obj.getAge();  
console.log(x)  
  
结果:  
undefined  
NaN
```

## 函数柯里化

柯里化(Currying)是一种函数转化方法，是 高阶函数(接收函数作为参数的函数)的一种，它将一个接收多参数的函数转化为接收部分参数的函数。柯里化后的函数只传递部分参数来调用，并返回一个新的函数 去处理剩余的参数，是逐步传参的过程。

```
function add(){
  let args = Array.prototype.slice.call(arguments)
  let inner = function(){
    args.push(...arguments)
    return inner
  }
  inner.toString = () => {
    return args.reduce((pre, cur) => {
      return pre + cur
    })
  }
  return inner
}
console.log(add(0,1,2,3) + '')
```

## js的对象模型

JavaScript 是一种基于原型而不是基于类的面向对象语言参考：

[https://developer.mozilla.org/zh-](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Details_of_the_Object_Model)

[CN/docs/Web/JavaScript/Guide/Details of the Object Model](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Details_of_the_Object_Model)基于类 vs 基于

原型的语言：基于类的面向对象语言，比如 Java 和 C++，是构建在两个不同实体的概念之上的：即类和实例。类可以定义属性，这些属性可使特定的对象集合特征化（可以将 Java 中的方法和变量以及 C++ 中的成员都视作属性）。类是抽象的，而不是其所描述的对象集合中的任何特定的个体。例如 Employee 类可以用来表示所有雇员的集合。另一方面，一个实例是一个类的实例化；也就是其中一名成员。例如，Victoria 可以是 Employee 类的一个实例，表示一个特定的雇员个体。实例具有和其父类完全一致的属性。基于原型的语言（如 JavaScript）并不存在这种区别：它只有对象。基于原型的语言具有所谓原型对象的概念。原型对象可以作为一个模板，新对象可以从中获得原始的属性。任何对象都可以指定其自身的属性，既可以是创建时也可以在运行时创建。而且，任何对象都可以作为另一个对象的原型，从而允许后者共享前者的属性。



# 原型

<https://juejin.cn/post/6844903837623386126#heading-5>

## 原型对象

实际上每一个构造函数都有一个名为 `prototype` 的属性，译成中文是原型的意思，`prototype` 的是对象类据类型，称为构造函数的原型对象，每个原型对象都具有 `constructor` 属性代表了该原型对象对应的构造函数。

JavaScript 中对象的工作机制：**当访问对象的属性或方法时，先在当前实例对象是查找，然后再去原型对象查找，并且原型对象被所有实例共享。**

```
function Person() {}  
// 每个函数都有 prototype 属性  
console.log(Person.prototype);  
1.  constructor: f Person()  
2.  [[Prototype]]: Object // 该属性指向实例对象的原型对象  
1.  constructor: f Object() // 原型对象默认只会取得一个  
   constructor属性，指向该原型对象对应的构造函数。至于其他方法，则是  
   从Object继承来的  
2.  hasOwnProperty: f hasOwnProperty() // 通过  
   hasOwnProperty()方法可以确定该属性是自有属性还是继承属性  
3.  isPrototypeOf: f isPrototypeOf() // 可以通过  
   isPrototypeOf()方法来确定对象之间是否是实例对象和原型对象的关系  
4.  propertyIsEnumerable: f propertyIsEnumerable()  
5.  toLocaleString: f toLocaleString()  
6.  toString: f toString()  
7.  valueOf: f valueOf()  
8.  __defineGetter__: f __defineGetter__()  
9.  __defineSetter__: f __defineSetter__()  
10. __lookupGetter__: f __lookupGetter__()  
11. __lookupSetter__: f __lookupSetter__()  
12. __proto__: Object // 实例对象内部包含一个proto属性指向该实  
   例对象对应的原型对象  
13.  get __proto__: f __proto__()  
14.  set __proto__: f __proto__()
```

## instanceof

instanceof操作符可以用来鉴别对象的类型

# constructor

每个对象在创建时都自动拥有一个构造函数属性constructor，其中包含了一个指向其构造函数的引用。而这个constructor属性实际上继承自原型对象，而constructor也是原型对象唯一的自有属性

## 属性查找

当读取一个对象的属性时，javascript引擎首先在该对象的自有属性中查找属性名字。如果找到则返回。如果自有属性不包含该名字，则javascript会搜索proto中的对象。如果找到则返回。如果找不到，则返回undefined

```
let obj = { name : 'harry', age:18 }
console.log(obj.address); //undefined
obj.__proto__ = {
    //不可变原型对象"#<object>"不能有其原型集，不能直接写
    obj.__proto__.__proto__ = {}
}
obj.__proto__.__proto__ = {
    address: '上海'
}
console.log(obj.address); //上海
```

```
let obj = {
    name : 'harry',
    age:18
}

obj.__proto__ = {
}
obj.__proto__.__proto__ = {
}
obj.__proto__.__proto__.__proto__ = {
    address: '上海'
}
console.log(obj.address); //上海
```

**\*\*js引擎会顺着这些原型不断的往上找，直到address这个属性。这些原型构成了原型链。\*\***

# 原型链

## es6中class

1. 类的定义使用class关键字，创建的本质还是函数，是一个特殊函数
2. 一个类只能拥有一个constructor构造器方法，如没有显示的定义一个构造方法，会添加一个默认的
3. 基础用extends
4. 一个构造器可以使用super关键字调用一个父类构造方法
5. 类没有私有属性

```
class Person {
  constructor(name) {
    console.log('constructor')
    this.name = name
  }

  say() {
    console.log('My name is ',this.name)
  }
}
const b = new Person("b");
```

## this的坑

代码在nodejs中结果不一样：

```
var x = 10 // 相当于 window.x = 10
var obj = {
  x: 20,
  f: function(){
    console.log(this.x)
  },
  s: function(){
    console.log(this.x) // 20
    function fn(){
      console.log(this.x)
    }
    return fn // 函数f虽然是在obj.fn内部定义的，但是它
```

仍然是一个普通的函数，this仍然指向window

```
    }
  }
  var fn = obj.f
  fn() // 10 函数执行 fn = function(){
console.log(this.x)}
obj.f() // 20 调用方法 `obj.f()` 中this 指向obj
obj.s()( ) // 10 函数执行

// 第一种情况
var x= 10
var obj ={
  x: 20,
  f1: function(){
    console.log(this.x)
  },
  f2: () => {
    console.log(this.x) // 指向 window
  }
}
obj.f1() // 20
obj.f2() // 10

// 第二种情况
var name = "jack"
var man = {
  name: 'tom',
  f1: function(){
    // 这边的this和下面的setTimeout函数下的this相等
    var that = this
    setTimeout(()=>{
      console.log(this.name, that === this) // 'tom'
true
    }, 0)
  },
  f2: function(){
    // 这边的this和下面的setTimeout函数下的this不相等
    var that = this
    setTimeout(function(){
      console.log(this.name, that === this) // 'jack'
false
    }, 0)
  }
}
man.f1() // 'tom' true
```

```
man.f2() // 'jack' false
```

/\* setTimeout默认指向window，但是，在箭头函数中，this的作用域环境在man内，故this指向了man。也就是说此时的this可以忽略外围包裹的setTimeout定时器函数，看上一层及的作用域。\*/

### 改变this三个方法总结：

call: fun.call(this, arg1, arg2,.....)

apply: fun.apply(this, [arg1, arg2,.....])

bind: fun.bind(this, arg1, arg2,.....)

相同点：

都可以用来改变this指向，第一个参数都是this指向的对象

区别：

call和apply：都会使函数执行，但是参数不同

bind：不会使函数执行，参数同call

## this指向问题

this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底指向谁，一般情况下this的最终指向的是那个调用它的对象。

1. 全局作用域或者普通函数中this指向全局对象window（注意定时器里面的this指向window）
2. 方法调用中谁调用this指向谁
3. 构造函数中this指向构造函数的实例
4. 箭头函数的this是在定义函数时绑定的，不是在执行过程中绑定的，箭头函数中的this取决于该函数被创建时的作用域环境。
5. 回调函数会丢失this的指向

```
let normal = {  
  'name': 'Jack',  
  'desc': function() {  
    // setTimeout(function(){console.log("普通函数: " +  
this.name)}})  
    return function(){console.log("普通函数: " +  
this.name)}  
  }  
}
```

```

        // 回调函数丢失this的指向

    }

}

let arrow = {

    'name': 'Nancy',

    'desc': function() {

        setTimeout(() => {console.log("箭头函数:  " +
this.name)})

    }

}

normal.desc()

arrow.desc()

// 普通函数:  undefined

// 箭头函数:  Jack

```

```

<button>点击</button>
<script>
    // this 指向问题 一般情况下this的最终指向的是那个调用它的
    对象

    // 1. 全局作用域或者普通函数中this指向全局对象window (
    注意定时器里面的this指向window)
    console.log(this);
    function fn() {
        console.log(this);
    }
    window.fn();
    window.setTimeout(function() {
        console.log(this);
    }, 1000);
    // 2. 方法调用中谁调用this指向谁
    var o = {

        call: function() {

```

```

        sayHi: function() {
            console.log(this); // this指向的是 o 这个对象
        }
    }
    o.sayHi();
    var btn = document.querySelector('button');
    btn.addEventListener('click', function() {
        console.log(this); // 事件处理函数中的this指向的是btn这个按钮对象
    })
    // 3. 构造函数中this指向构造函数的实例
    function Fun() {
        console.log(this); // this 指向的是fun 实例对象
    }
    var fun = new Fun();
</script>

```

这里举一个简单的例子

```
var cat={name:'小猫',eat:function(){console.log(this.name+'在吃饭')}}

```

小猫对象拥有name变量代表名字以及eat的吃饭函数

如果我们调用cat.eat()会执行对应的函数，this作用域拿到cat对象，并获取了cat.name跟在吃饭拼接，变成了小猫在吃饭

但是如果这时候我们创建一个小狗

```
var dog={name:'大黄'}

```

想让小狗也吃饭，其实并不需要再声明一次函数，直接使用cat.eat.call(dog)就可以了

cat.eat获取了cat对象里的eat函数

call可以改变函数内部的this作用域得指向，call的括号内部就是指向哪个对象，这里我们指向了dog，所以在执行这句代码的过程中cat.eat的this作用域由于使用了call(dog)而变成了dog

call与apply的功能一致，唯一的区别就是apply是数组传入参数而已。

```

for(let i = 0;i<5;i++){
    setTimeout(() => {
        console.log(i++)
    }, 2000)
}

```

```
}
console.log('it=》 ' + i)

for(var i = 0;i<5;i++){
    setTimeout(() => {
        console.log(i++)
    }, 2000)
}
console.log('it2=》 ' + i)

var x = 1

function name1() {
    console.log(x)
    var x = 2000
}

name1()
function name2() {
    var x = 2000
    console.log(x)
}

name2()

function name3() {
    var x = 2000
    console.log(this.x)
}

name3()
```

# 任务

JS中所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。

同步任务指的是：在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；

异步任务指的是：不进入主线程、而进入"任务队列"的任务，当主线程中的任务运行完了，才会从"任务队列"取出异步任务放入主线程执行。



# JS执行机制（事件循环）

1. 先执行执行栈中的同步任务。
2. 异步任务（回调函数）放入任务队列中。
3. 一旦执行栈中的所有同步任务执行完毕，系统就会按次序读取任务队列中的异步任务，于是被读取的异步任务结束等待状态，进入执行栈，开始执行。[https://blog.csdn.net/qq\\_57406825/article/details/124290457](https://blog.csdn.net/qq_57406825/article/details/124290457)

## Promise概念

多层回调函数的相互嵌套，就形成了回调地狱。回调地狱的缺点：

- 代码耦合性太强，牵一发而动全身，难以维护
  - 大量冗余的代码相互嵌套，代码的可读性变差
- Promise对象用于一个异步操作的最终完成（包括成功和失败）以及结果值的表示
- 对比：

```
$.ajax({
  url: 'xxx1',
  success: function (data) {
    $.ajax({
      url: 'xxx2',
      success: function (data) {
        $.ajax({
          url: 'xxx3',
          success: function (data) {
            console.log(data)
          }
        });
      }
    });
  }
});
```

```
function queryAjax(url) {
  var p = new Promise(function (resolve, reject) {
```

```

var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function () {
    if (xhr.readyState !== 4) return;
    if (xhr.readyState === 4 && xhr.status === 200)
{
        resolve(xhr.responseText);
    } else {
        reject('err');
    }
};
xhr.open('get', url);
xhr.send(null);
});
return p;
}

queryAjax('xxx1')
    .then(function (data) {
        return queryData('xxx2');
    })
    .then(function (data) {
        return queryData('xxx3');
    })
    .then(function (data) {
        console.log(data)
    });

```

demo:

```

const imgUrl =
'https://img2.woyaogexing.com/2022/11/21/9e0aed1b5c3c2832!
400x400.jpg'

const imgPromise = (url) => {

    return new Promise((resolve, reject) => {

        const img = new Image();

        img.src = url;

        img.onload = () => {

            resolve(img)

```

```

    }

    img.onerror = () => {

        reject(new Error('图片加载失败! '))

    }

    })

}

imgPromise(imgUrl).then(

    img => {

        document.body.appendChild(img)

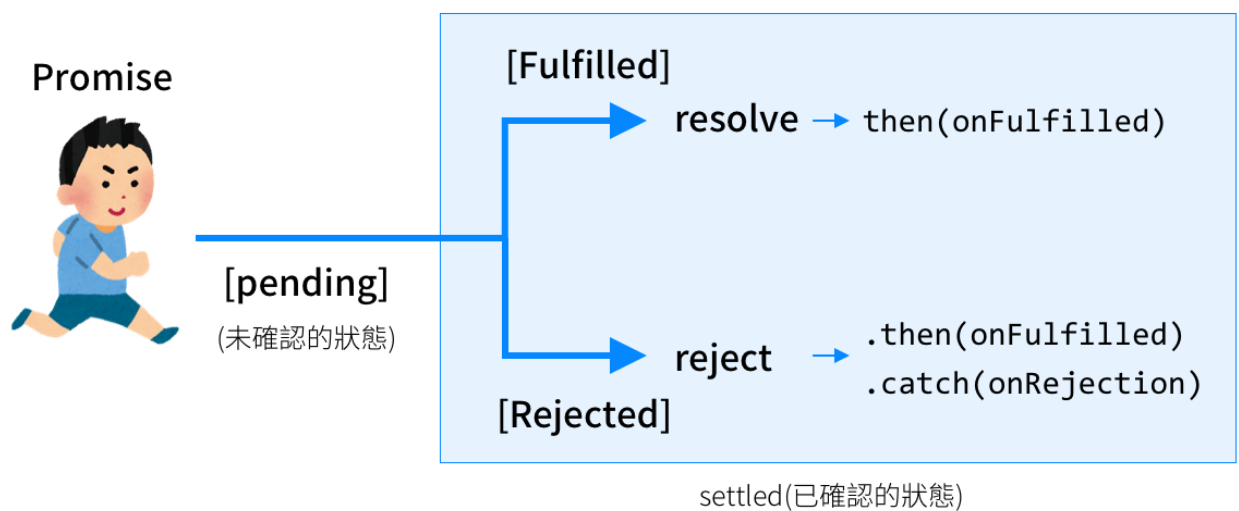
    }

).catch(err => {

    document.body.innerHTML = err

})

```



# 语法

- Promise 是一个构造函数
  - 我们可以创建 Promise 的实例 `const p = new Promise()`
  - `new` 出来的 Promise 实例对象，代表一个异步操作
- Promise.prototype 上包含一个 `.then()` 方法
  - 每一次 `new Promise()` 构造函数得到的实例对象，
  - 都可以通过原型链的方式访问到 `.then()` 方法，例如 `p.then()`
- `.then()` 方法用来预先指定成功和失败的回调函数
  - `p.then(成功的回调函数, 失败的回调函数)`
  - `p.then(result => {}, error => {})`
  - 调用 `.then()` 方法时，成功的回调函数是必选的、失败的回调函数是可选的

```
new Promise( //executor function(resolve, reject) {...})
```

## race和call

call：并发处理多个任务，所有任务执行完才得到结果

race：并发处理多个任务，只要有一个任务执行完就能得到结果

### 1. race

```
Promise.race([p1,p2,p3]).then(function(result){  
  console.log(result)  
})
```

result只能得到最先结束异步任务的对象的结果

### 2. all

```
Promise.all([p1,p2,p3]).then(function(result){  
  console.log(result)  
})
```

## executor

- executor是一个带有resolve和reject两个参数的函数
- executor函数在Promise构造函数执行时同步执行，被传递的resolve和reject函数
- executor内部通常会执行一些异步操作，一旦完成，可以调用resolve函数来将promise；状态改为fulfilled即完成，或者在发生错误时将它的状态改成rejected即失败
- 如果在executor函数中抛出一个错误，那么该promise状态为rejected。executor函数返回值被忽略
- executor中，resolve或者reject只能执行其中一个函数

## Promise状态

- pending：初始状态，不是成功或者失败状态
- fulfilled：意味着操作成功
- rejected：操作失败

## Promise.then(onFulfilled,onRejected)

参数时两个函数，根据Promise状态调用不同的函数，fulfilled走onFulfilled函数，rejected走onRejected函数。then的返回值时一个新的promise对象。

```
promise()
  .then((success) => {
    console.log(success);
  }, (fail) => {
    console.log(fail);
  })
```

## catch (onRejected)

为当前Promise对象添加一个拒绝回调，返回一个新的Promise对象

## Promise.all() 方法

Promise.all() 方法会发起并行的 Promise 异步操作，等所有的异步操作全部结束后才会执行下一步的 .then 操作（等待机制）

# 生成器

[https://blog.csdn.net/m0\\_71485750/article/details/125471287](https://blog.csdn.net/m0_71485750/article/details/125471287)

- 在js中，生成器需要在function后面添加一个符号`*`，比如`function* func()`
- 生成器代码的执行可以被yield控制，当函数内部代码，遇到yield时会中断执行
- 生成器函数默认在执行时，返回的也是一个生成器
- 如果想要执行函数内部的代码，需要调用返回的生成器对象的next方法

```
function* makeRangeIterator(start = 0, end = 20, step = 1)
{
    for (let i = start; i < end; i += step) {
        yield i;
    }
}

var a = makeRangeIterator(1,10,2)

a.next() // {value: 1, done: false}
a.next() // {value: 3, done: false}
a.next() // {value: 5, done: false}
a.next() // {value: 7, done: false}
a.next() // {value: 9, done: false}
a.next() // {value: undefined, done: true}
```

## async/await

async和await是generator的语法糖，底层实现是带有执行器的 Generator 函数。个人理解async函数就是将 Generator 函数的星号`*`替换成async，将

`yield` 替换成 `await`，让异步操作同步化。

```
function fn(num) {
  return new Promise((resolve, reject) => {
    resolve(console.log(num))
  })
}

function *testGen(){
  fn(1)
  yield
  fn(2)
  yield
  fn(3)
}

let f = testGen()
f.next()
f.next()
f.next()
```

`async/await` 是 ES8 (ECMAScript 2017) 引入的新语法，用来简化 `Promise` 异步操作。在 `async/await` 出现之前，开发者只能通过链式 `.then()` 的方式处理 `Promise` 异步操作。

`async/await` 的使用注意事项

1. 如果在 `function` 中使用了 `await`，则 `function` 必须被 `async` 修饰。`await` 关键字会暂停异步函数的执行，并等待 `Promise` 执行，然后继续执行异步函数，并返回结果。
2. `await` 只在异步函数里面才起作用，如果 `await` 操作符后的表达式的值不是一个 `Promise`，则返回该值本身。
3. 在 `async` 方法中，第一个 `await` 之前的代码会同步执行，`await` 之后的代码会异步执行
4. `async/await` 虽然比 `Promise` 的 `then` 方式优化了很多，但是如果有大量的 `await` 的 `promises` 相继发生程序会变慢。如果想要所有的 `promises` 同时开始处理，最后才返回结果，此时可以借助 `setTimeout` 伪造异步进程；

```
function fn() {
  return new Promise((resolve, reject) => {
```

```

        resolve(Math.random())
    });
}
(async function asyncFn() {
    var random1 = await fn();
    console.log(random1);
    var random2 = await fn();
    console.log(random2);
})();

// 用同步的方法，调用异步的函数。
function fn(num){
    return new Promise((resolve,reject) => {
        setTimeout( () => {
            resolve(num)
        }, 1000)
    })
}
async function test(){
    let a = await fn(1)
    let b = await fn(2)
    let c = await fn(3)
    console.log(a,b,c) // 1 2 3
}
test()

```

## 两个属性 async（异步） 和 defer（推迟）

他们的功能是：

async：他是异步加载，不确定何时会加载好；页面加载时，带有 async 的脚本也同时加载，加载后会立即执行，如果有一些需要操作 DOM 的脚本加载比较慢时，这样会造成 DOM 还没有加载好，脚本就进行操作，会造成错误。

defer：页面加载时，带有 defer 的脚本也同时加载，加载后会等待页面加载好后，才执行。

## EventLoop

<http://latentflip.com/loupe/?code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXloKSB7CiAgICAgICAgY29uc29s>

[code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXloKSB7CiAgICAgICAgY29uc29s](http://latentflip.com/loupe/?code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXloKSB7CiAgICAgICAgY29uc29s)

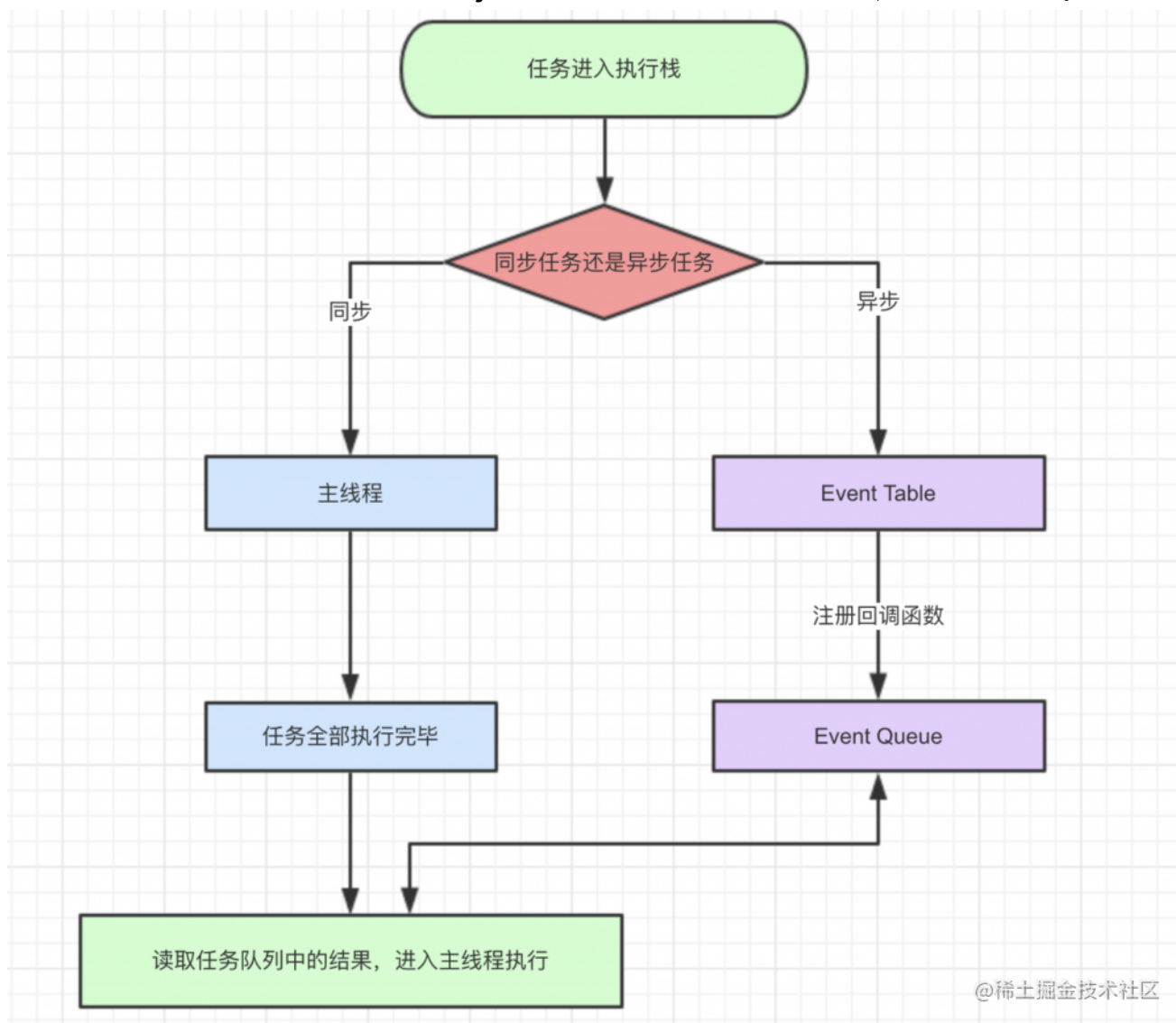


[ZS5sb2coJ1lvdSBjbGlja2VklHRoZSBidXR0b24hJyk7ICAglAogICAglCAgfSwgMjAwMCk7Cn0pOwoKY29uc29sZS5sb2colkhplSlpOwoKc2V0VGltZW91dChmdW5jdGlvb](#)  
[iB0aW1lb3V0KCkgewogICAglCAgY29uc29sZS5sb2colkNsaWNrIHRoZSBidXR0b24hli](#)  
[k7Cn0slDUwMDApOwoKY29uc29sZS5sb2colldlbGNvbWUgdG8gbG91cGUulik7!](#)  
[!!PGJ1dHRvb5DbGljayBtZSE8L2J1dHRvb4%3D](#)

JS代码虽然是单线程的，但是浏览器是多线程的啊。

所以人们就想到可以调用浏览器的API，把需要异步运行的代码扔给浏览器。

等到浏览器运行完，再塞回来js线程中，然后继续运行(异步的原理)。



**同步任务和异步任务**为了防止某个耗时任务导致程序假死的问题，

JavaScript 把待执行的任务分为了两类：

- ① 同步任务 (synchronous) 又叫做非耗时任务，指的是在主线程上排队执行的那些任务 只有前一个任务执行完毕，才能执行后一个任务
- ② 异步任务 (asynchronous) 又叫做耗时任务，异步任务由 JavaScript 委

托给宿主环境(浏览器或者nodejs)进行执行当异步任务执行完成后，会通知 JavaScript 主线程执行异步任务的回调函数

JavaScript 把异步任务又做了进一步的划分，异步任务又分为两类，分别是：

- ① 宏任务 (macrotask) 新程序或者子程序被直接执行 (比如 `<script>` 执行开始)、事件的回调函数、异步 Ajax 请求、`setTimeout`、`setInterval`、文件操作、其它宏任务
- ② 微任务 (microtask) `Promise.then`、`.catch` 和 `.finally`、`process.nextTick` (nodejs)、其它微任务

```
console.log('第一步')
new Promise((resolve, reject) => {

  console.log('第二步')

  setTimeout(() => {

    resolve('成功了')

    console.log('第四步')

  })

}).then(

  success => {console.log(success)},

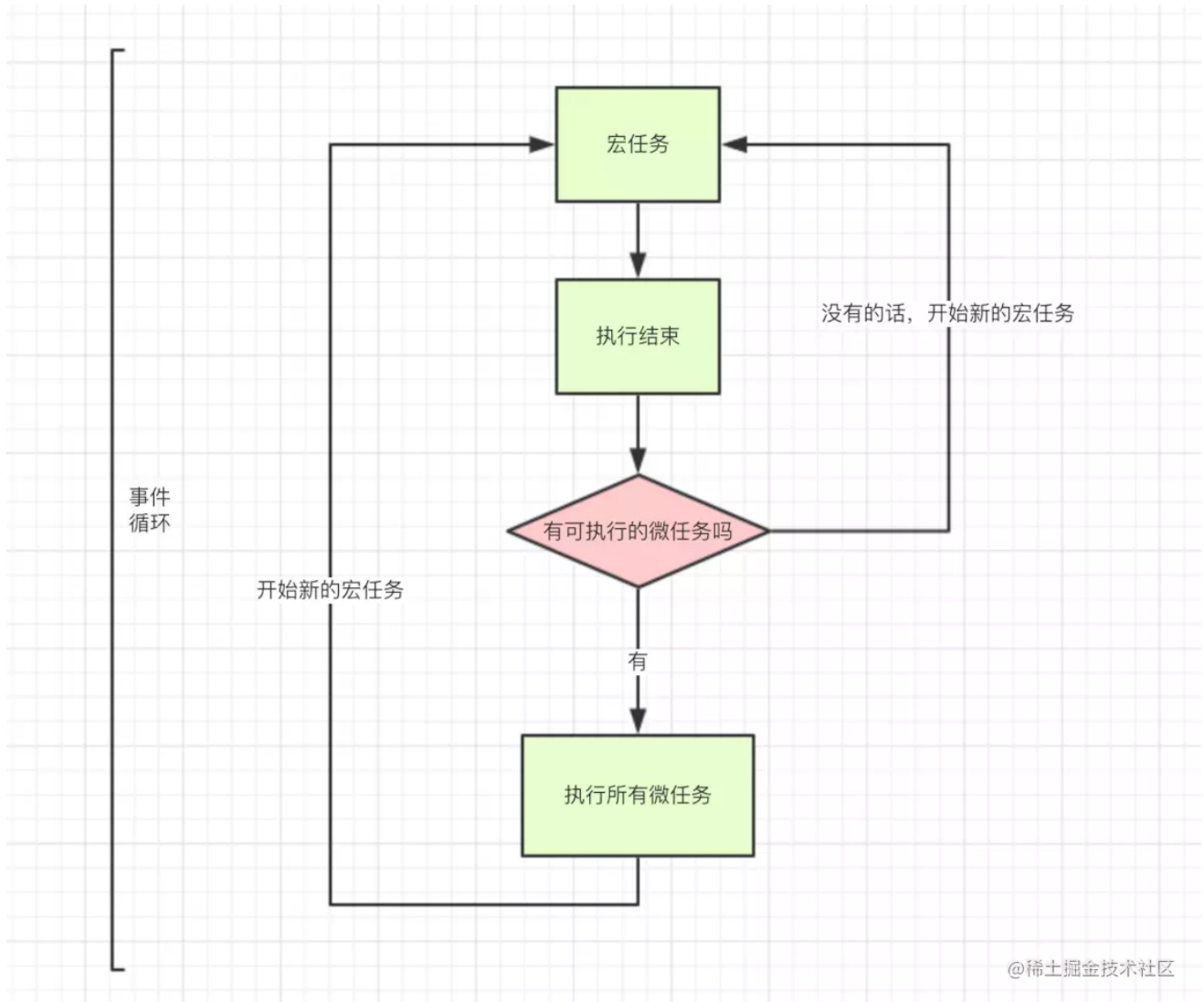
  failed => {console.log(failed.message)}

)

console.log('第三步')
```

一个宏任务执行完之后，都会检查是否存在待执行的微任务，如果有，则执行完所有微任务之后，再继续执行下一个宏任务。（整个过程是在栈

中，先进先出)



@稀土掘金技术社区

## JS执行机制（事件循环）

1. 先执行执行栈中的同步任务。
2. 异步任务（回调函数）放入任务队列中。
3. 一旦执行栈中的所有同步任务执行完毕，系统就会按次序读取任务队列中的异步任务，于是被读取的异步任务结束等待状态，进入执行栈，开始执行。

[https://blog.csdn.net/qg\\_57406825/article/details/124290457](https://blog.csdn.net/qg_57406825/article/details/124290457)

## 事件

dom 事件流 三个阶段（1.事件捕获阶段 2.处于目标阶段 3.事件冒泡阶段）

1. JS 代码中只能执行捕获或者冒泡其中的一个阶段。

2. 捕获阶段 如果addEventListener 第三个参数是 true 那么则处于捕获阶段 document -> html -> body -> father -> son
3. 冒泡阶段 如果addEventListener 第三个参数是 false 或者 省略 那么则处于冒泡阶段 son -> father -> body -> html -> document

```
var son = document.querySelector('.son');
son.addEventListener('click', function() {
    alert('son');
}, true);
var father = document.querySelector('.father');
father.addEventListener('click', function() {
    alert('father');
}, true);
```

## 执行上下文



## 执行上下文的类型

JavaScript 中有三种执行上下文类型。

- **全局执行上下文** — 这是默认或者说基础的上下文，任何不在函数内部的代码都在全局上下文中。它会执行两件事：创建一个全局的 window 对象（浏览器的情况下），并且设置 `this` 的值等于这个全局对象。一个程序中只会有一个全局执行上下文。
- **函数执行上下文** — 每当一个函数被调用时，都会为该函数创建一个新的上下文。每个函数都有它自己的执行上下文，不过是在函数被调用时创

建的。函数上下文可以有任意多个。每当一个新的执行上下文被创建，它会按定义的顺序，执行一系列步骤。

- **Eval 函数执行上下文** — 执行在 `eval` 函数内部的代码也会有它属于自己的执行上下文，但由于 JavaScript 开发者并不经常使用 `eval`，所以在这里我不会讨论它。

## 执行栈



执行栈，也就是在其它编程语言中所说的“调用栈”，是一种拥有 LIFO（后进先出）数据结构的栈，被用来存储代码运行时创建的所有执行上下文。

当 JavaScript 引擎第一次遇到你的脚本时，它会创建一个全局的执行上下文并且压入当前执行栈。每当引擎遇到一个函数调用，它会为该函数创建一个新的执行上下文并压入栈的顶部。

引擎会执行那些执行上下文位于栈顶的函数。当该函数执行结束时，执行上下文从栈中弹出，控制流程到达当前栈中的下一个上下文。

# 闭包

**闭包** (closure) 是一个函数以及其捆绑的周边环境状态 (lexical environment, **词法环境**) 的引用的组合。换言之，闭包让开发者可以从**内部函数访问外部函数的作用域**。在 JavaScript 中，闭包会随着函数的创建而被同时创建。

作用：允许将函数与其所操作的某些数据（环境）关联起来

```
// var 全局作用域，作用域链
var data = [];

for (var i = 0; i < 3; i++) {
  data[i] = function () {
    console.log(i);
  };
}
```

```
data[0]();
data[1]();
data[2]();
// 3
// 3
// 3
```

```
for(var i = 0;i<5;i++){
  setTimeout(() => {
    console.log(i++)
  }, 10)
}
console.log('it=》' + this.i)
// it=》 5
// 5
// 6
// 7
// 8
// 9
```

// 解决闭包的问题，`let` 具有块级作用域，形成私有作用域都是互不干扰的。

```
for(let i = 0;i<5;i++){
```

```
        setTimeout(() => {
            console.log(i++)
        }, 10)
    }
    console.log('it=》 ' + this.i)

    for(var i = 0; i<10; i++){
        (function(j){
            setTimeout(function(){
                console.log(j)
            }, 1000)
        })(i)
    }
}
```

因为存在闭包的原因上面能依次输出1~10，闭包形成了10个互不干扰的私有作用域。将外层的自执行函数去掉后就不存在外部作用域的引用了，输出的结果就是连续的 10。

## 防抖-debounce

在一定的时间间隔内，多次触发只有一次触发产生。

## 节流-throttle

- 拖拽一个元素时，要随时拿到该元素的拖拽位置
- 直接使用drag事件，则会频繁触发，很容易导致卡顿
- 节流：无论拖拽速度多快，都会每隔100ms触发一次