面试题

——By 郝荣政

javascript 有几种类型的值，以及内置对象？

栈:原始数据类型（undefined,null,boolean,number,string,symbol）

堆:引用数据类型（object,array,function）

两种类型的区别是：存储位置不同；

原始数据类型直接存储在栈中的简单数据段，占据空间小，大小固定，属于被频繁使用数据，所以放在栈中存储，

引用数据类型存储在堆中的对象，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取实体

Object 是 JavaScript 中所有对象的父对象

数据封装类对象：Object,Array,Boolean,Number 和 String

其他对象：Function,Arguments,Math,Date,RegExp,Error

堆，栈的区别？

申请方式：

内存中的堆和栈第一个区别就是申请的方式不同：栈是系统自动分配空间的，而堆则是程序员根据需要自己申请的空间，由于栈上的空间是自动分配自动回收的，所以栈上的

数据的生存周期只是在函数运行过程中，运行后就会释放掉，不可以再访问，而堆上的数据只要程序员不释放空间，就一直可以访问，不过缺点是一旦忘记释放会内存泄漏。

申请效率的比较：栈由系统自动分配，速度较快，但程序员无法控制，堆由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片，不过用起来最方便

申请大小的限制：

栈：在 windows 下，栈是向低地址扩展的数据结构，是一块连续的内存的区域，这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 windows 下，栈的大小是 2M（也有的说是 1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 overflow，因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域，这是由于系统是用链表来存储空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址，堆的大小

受限于计算机系统中有效的虚拟内存，由此可见，堆获得的内存空间比较灵活，也比较大。

宏队列、微队列的区别？

Event Loop 就是事件循环，是浏览器和 NodeJS 用来解决 Javascript 单线程运行带来的问题的一种运行机制。

针对于浏览器和 NodeJS 两种不同环境，Event Loop 也有不同的实现：

浏览器的 Event Loop 是在 html5 的规范中明确定义，NodeJS 的 Event Loop 是基于 libuv 实现的，libuv 已经对 Event Loop 做出了实现，而 HTML5 规范中只是定义了浏览器中 Event Loop 的模型，具体的实现留给了浏览器厂商，因此浏览器和 NodeJS 的 Event Loop 是两种不同的概念。

进程(process)是系统分配的独立资源，是 CPU 资源分配的基本单位，进程是由一个或者多个线程组成的。

程序是指令和数据的有序集合，进程中的文本区域就是代码区就是程序。

线程(thread)是进程的执行流，是 CPU 调度和分派的基本单位。

进程是包含程序的，进程的执行离不开程序，程序本身没有任何运行的含义，是一个静态的概念。而进程则是在处理机上的一次执行过程，它是一个动态的概念。同一个程序包含多个进程。

进程和线程独立运行，并可能同时运行。在不同的处理器，甚至不同的计算机上，但多个线程能够共享单个相同进程的内存。不同的进程则不能。

在 javascript 中，任务被分为两种，一种宏任务也叫 task，一种叫微任务

1.宏任务，macrotask 也叫 task，一些异步任务的回调会依次进入 macro task queue，等待后续被调用，这些异步任务包括

- setTimeout
- setInterval
- setImmediate(Node 独有)
- requestAnimationFrame(浏览器独有)
- I/O
- UI rendering(浏览器独有)

2.微任务，microtask，也叫 jobs，另一些异步任务的回调会依次进入 micro task queue，等待后续被调用，这些异步任务包括

- process.nextTick(Node 独有)
- Promise
- Object.observe
- MutationObserver

浏览器的 Evnet Loop

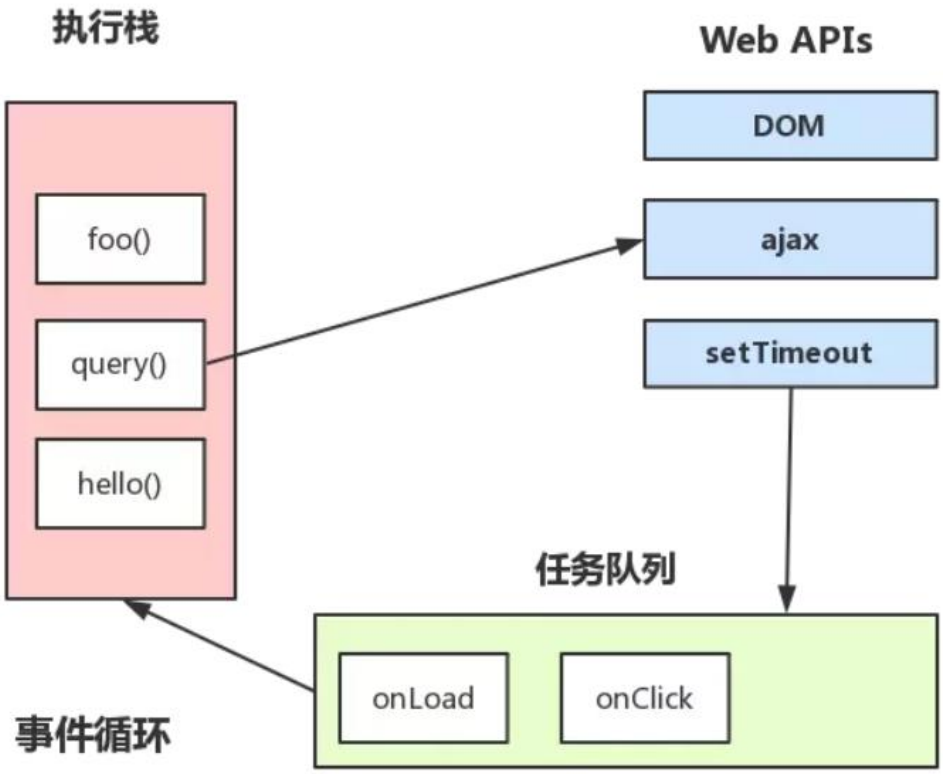
javascript 有一个 main thread 主线程和 call-stack 调用栈(执行栈)，所有的任务都会被放到调用栈等待主线程执行

JS 调用栈

JS 调用栈是一种后进先出的数据结构，当函数被调用时，会被添加到栈中的顶部，执行完之后从栈定移出该函数，直到栈内被清空

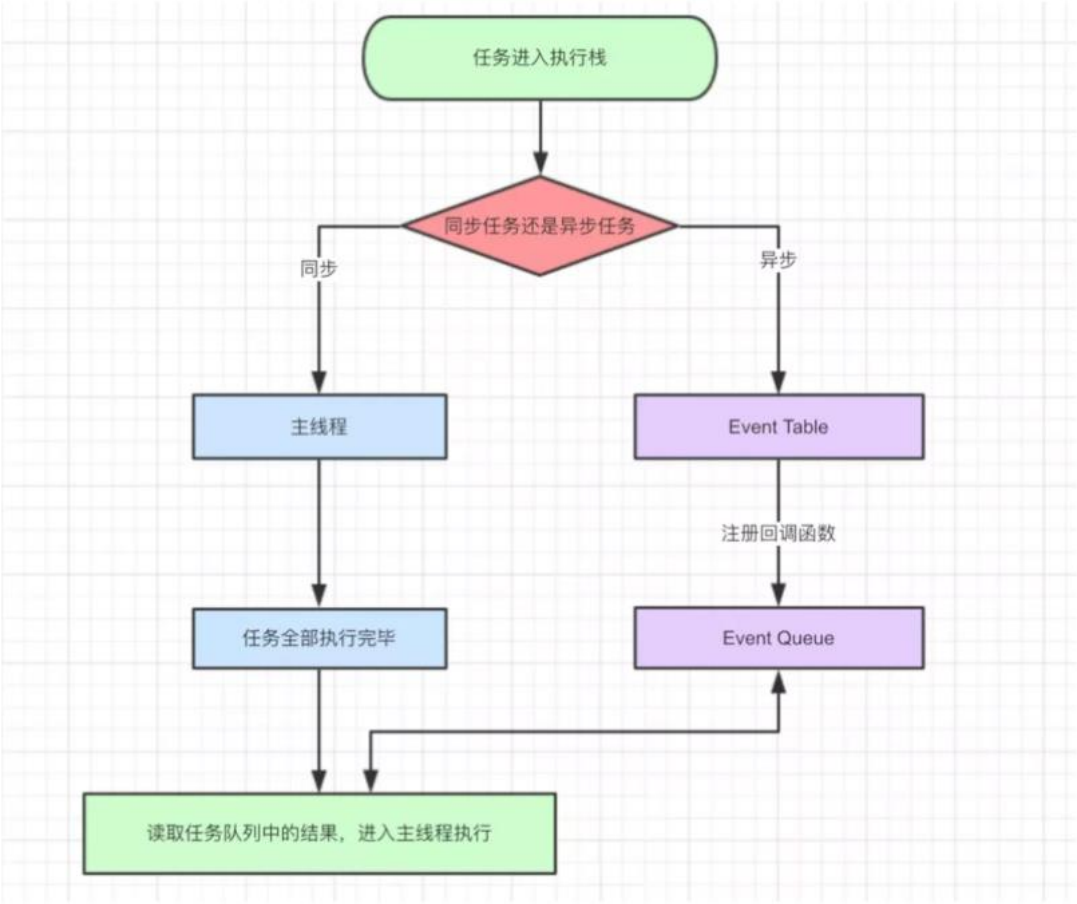
同步任务，异步任务

javascript 单线程中的任务分为同步任务和异步任务，同步任务会在调用栈中按照顺序排队等待主线程执行，异步任务则会在异步有了结果后将注册的回调函数添加到任务对列(消息队列)中等待主线程空闲的时候，也就是栈内被清空的时候，被读取到栈中等待主线程执行，任务队列是先进先出的数据结构



事件循环

调用栈中的同步任务都执行完毕，栈内被清空了，就代表主线程空闲了，这个时候就会去任务队列中按照顺序读取一个任务读入到栈中执行，每次栈内被清空，都会去读取任务队列有没有任务，有就读取执行，一直循环读取-执行的操作，就形成了事件循环



定时器

定时器会开启一条定时器触发线程来触发计时，定时器会在等待了指定的时间后将事件放入到任务队列中读取到主线程执行，例如 setTimeout：定时器的精度可能不高，大体看来，只要确保回调函数不会在指定的时间间隔之前运行，但可能会在那个时刻执行，也可能在那之后运行，要根据事件队列的状态而定，定时 0 的话是 4ms,程序通常被分成了很多小块，在事件循环队列中一个接一个地执行，严格地说，和你的程序不直接相关的其他事件也可能会插入到队列中

Javascript 代码的具体流程

1. 执行全局 script 同步代码，这些同步代码有一些是同步语句，有一些是异步语句（比如 setTimeout）等；遇到了 setTimeout，就会等到过了指定的时间后将回调函数放入到宏队列的任务队列中，遇到 Promise，将 then 函数放入到微任务的任务队列中。
2. 全局 script 代码执行完毕后，调用栈 stack 会清空。
3. 去检测微任务的任务队列是否存在任务，存在就执行，从微任务队列 microtask queue 中取出位于队首的回调任务，放入调用栈 stack 中执行，执行完 microtask queue 长度减 1。
4. 继续取出位于队首的任务，放入调用栈 stack 中执行，以此类推，直到把 microtask queue 中的所有任务都执行完毕，注意：如果在执行 microtask 的过程中，又产生了 microtask，那么会加入到队列的末尾，也会在这个周期被调用执行。
5. Microtask queue 中的所有任务都执行完毕，此时 microtask queue 为空队列，调用栈 stack 也为空。

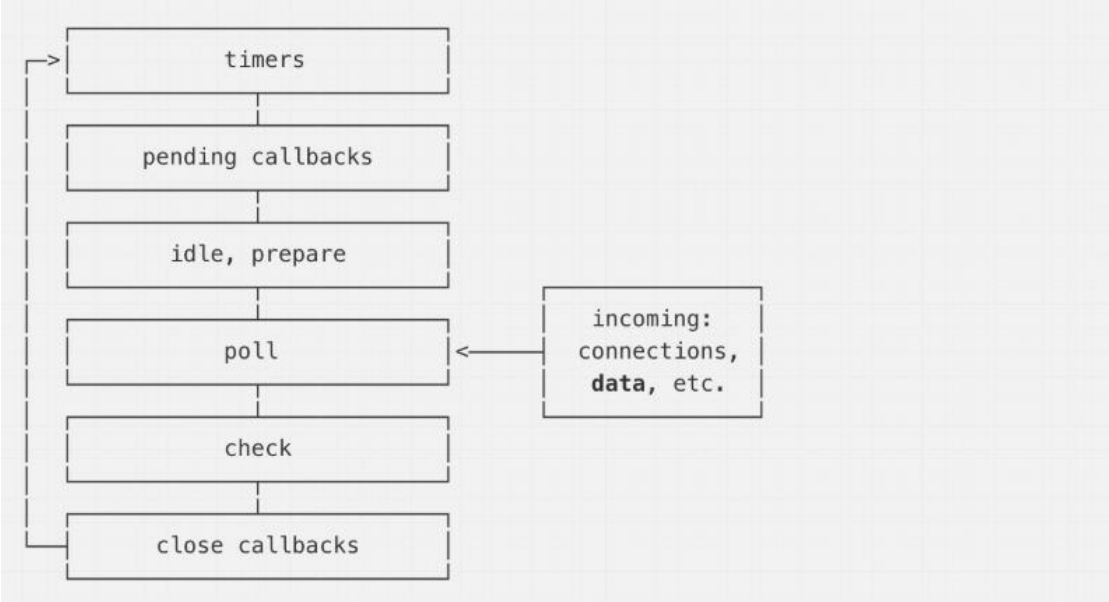
6. 取出宏队列 macrotask queue 中对于队首的任务，放入 stack 中执行，发现在这次循环中不存在微任务，存在就进行第三个步骤，不存在就进行第七个步骤
7. 宏任务执行完后，macrotask queue 为空
8. 全部执行完后，stack queue 为空，macrotask queue 为空，micro queue 为空

示例：输出以下内容的结果

```
console.log('start');
setTimeout(function() {
    console.log(1);
}, 0);
Promise.resolve().then(function() {
    console.log(2);
}).then(function() {
    console.log(3);
});
setTimeout(() => {
    console.log(4);
    Promise.resolve().then(() => {
        console.log(5)
    });
});
new Promise((resolve, reject) => {
    console.log(6),
    resolve(7)
}).then((data) => {
    console.log(data);
})
console.log('end');
// start,6,end,2,7,3,1,4,5
```

NodeJs 的 Event Loop

Node 的 Event Loop 一共分为 6 个阶段，每个阶段都有自己的任务队列，当本阶段的任务队列都执行完毕或者达到了执行的最大任务数，就回进入到下一个阶段。



Timers 阶段

这个阶段会执行被 setTimeout 和 setInterval 设置的定时任务，当然这个定时并不是准确的，而是在超过了定时时间后，一旦得到执行机会就立刻执行

Pending callbacks 阶段

这个阶段会执行一些和底层系统有关的操作，例如 TCP 连接返回的错误等，这些错误发生时，会被 Node 推迟到下一个循环中执行

轮询阶段

这个阶段是用来执行和 IO 操作有关的回调的，Node 会向系统询问是否有新的 IO 事件已经触发，然后会执行响应的事件回调，几乎除了定时器事件，setImmediate()和 close callbacks 之外操作都会在这个阶段执行

Check 阶段

这个阶段会执行 setImmediate()设置的任务

Close callbacks 阶段

如果一个 socket 或 handle(句柄)突然被关闭了，例如通过 socket.destroy()关闭了，close 事件将会在这个阶段发出

宏队列，微队列

宏队列：

在浏览器中，可以认为下面只有一个宏队列，所有的 macrotask 都会被加到这一个宏队列中，但是在 NodeJS 中，不同的 macrotask 会被放置在不同的宏队列中。

1. Timers Queue
2. IO Callbacks Queue
3. Check Queue
4. Close Callbacks Queue

微队列

在浏览器中，也可以认为只有一个微队列，所有的 microtask 都会被加到这个微队列中，但是在 NodeJS 中，不同的 microtask 会被放置在不同的微队列中

1. Next Tick Queue:是放置 process.nextTick(callback)的回调任务的
2. Other Micro Queue:放置其他 microtask，比如说 Promise 等

NodeJS 的 Event Loop 过程

1. 执行全局 Script 的同步代码
2. 执行 microtask 微任务，先执行所有 Next Tick Queue 中的所有任务，再执行 Other Microtask Queue 中的所有任务
3. 开始执行 macrotask 宏任务，共六个阶段，从第一个阶段开始执行相应每一个阶段 macrotask 中的所有任务，注意，这里是所有每个阶段宏任务队列中的所有任务，在浏览器的 Event Loop 中是只取宏队列的第一个任务出来执行，每一个阶段的 macrotask 任务执行完毕后，开始执行微任务，也就是步骤二
4. Timer Queue ->步骤二->I/O Queue->步骤二->Check Queue ->步骤二->Close Callback Queue->步骤二->Timers Queue……
5. 这就是 Node 的 Event Loop

示例：输出以下内容的结果

```
console.log('start');
setTimeout(() => {
    console.log(1);
    setTimeout(() => {
        console.log(2);
    }, 0);
    setImmediate(() => {
        console.log(3);
    })
    process.nextTick(() => {
        console.log(4);
    })
}, 0);
setImmediate(() => {
    console.log(5);
    process.nextTick(() => {
        console.log(6);
    })
})
setTimeout(() => {
    console.log(7);
    process.nextTick(() => {
        console.log(8);
    })
}, 0);
process.nextTick(() => {
    console.log(9);
})
console.log('end');
// start，end，9，1，7，4，8，5，3，6，2
```

setTimeout 对比 setImmediate

1. setTimeout(fn,0)在 Timers 阶段执行, 并且是在 poll 阶段进行判断是否达到指定的 timer 时间才会执行
2. setImmediate（fn）在 check 阶段执行，两者的执行顺序要根据当前的执行环境才能确定
3. 如果两者都在主模块调用，那么执行先后取决于进程性能，顺序随机
4. 如果两者都不在主模块调用，即在一个 I/O circle 中调用，那么 setImmediate 的回调永远先执行，因为会先到 check 阶段

setImmediate 对比 process.nextTick
1. setImmediate（fn）的回调任务会插入到宏队列 Check Queue 中
2. process.nextTick(fn)的回调任务会插入到微队列 Next Tick Queue 中
3. process.nextTick(fn)调用深度有限制，上限是 1000，而 setImmediate 则没有

有以下 3 个判断数组的方法，请分别介绍它们之间的区别和优劣

Object.prototype.toString.call(),instanceof 以及 Array.isArray()

性能方面：Array.isArray()性能最好，instanceof 次之，Object.prototype.toString,call()第三

功能方面：Object.prototype.toString.call()所有类型都可以判断，instanceof 只能判断对象原型，原始类型不可以。Array.isArray 优于 instanceof ，因为 Array.isArray 可以检测出 iframes， Array.isArray() 是 ES5 新增的方法，当不存在 Array.isArray() ，可以用 Object.prototype.toString.call() 实现

Array(…)和 Array.of(…)的区别

Array(…)的作用是接受参数返回一个数组，但是有一个陷阱，如果只传入一个参数，并且这个参数是数字的话，那么不会构造一个值为这个数字的单个元素的数组，而是构造一个空数组

```
Var a = Array( 3 );
a.length //3
a[0]; //undefind
```

Array.of(…)解决掉了这个陷阱

```
Var b = Array.of( 3 );
b.length; // 1
b[0]; // 3
var c = Array.of(1,2,3);
c.length; // 3
c; //1,2,3
```

类数组对象转换为数组

类数组对象：只包含使用从零开始，且自然递增的整数做键名，并且定义了 length 表示元素个数的对象，我们就认为他是类数组对象，类数组对象可以进行读写操作和遍历操作

```
var arrLike = {
        length: 4,
        2: "foo"
    };
```

要将其转换为真正的数组可以使用各种 Array.prototype 方法(map(…),indexOf(…)等)

```
var arr = Array.prototype.slice.call( arrLike );
var arr2 = arr.slice();
var arr = Array.from( arrLike );
```

常见的类数组

1. arguments

```
console.log(Array.isArray(arguments));
//false
console.log(arguments);
//node 打印: {}
//chrome 打印: Arguments [callee: ƒ, Symbol(Symbol.iterator): ƒ]
```

//因为 arguments 具有 Symbol.iterator 属性，所以它可以用扩展运算符(...arguments) 或其他使用迭代器的方法

2. 仅限浏览器中

```
const nodeList = document.querySelectorAll('*');
console.log(Array.isArray(nodeList));
```

3. 字符串 String

```
const array = Array.from('abc');
console.log(array);
//["a", "b", "c"]
```

4. TypeDArray

```
const typedArray = new Int8Array(new ArrayBuffer(3));
console.log(Array.isArray(typedArray));
//false
```

5. {length:0} 是类数组的特殊情况，转换时可以执行成功，返回空数组[]

```
console.log(Array.from({length:0}));
//[]
console.log(Array.from(''));
//[]
```

从内存来看 null 和 undefined 本质的区别是什么

给一个全局变量赋值为 null，相当于将这个变量的指针对象以及值清空，如果是给对象的属性赋值为 null，或者局部变量赋值为 null，相当于给这个属性分配了一块空的内存，然后值为 null，js 会回收全局变量为 null 的对象

给一个全局变量赋值为 undefined，相当于将这个对象的值清空，但是这个对象依旧存在，如果是给对象的属性赋值为 undefined，说明这个值为空值

在验证 null 时，一定要用===，因为==无法分别 null 和 undefined

扩展：null，undefined 或者 undeclared？该如何检测它们

未定义的属性，定义未赋值的为 undefined，JavaScript 访问不会报错，null 是一种特殊的 object；NaN 是一种特殊的 Number；undeclared 是未声明也未赋值的变量，JavaScript 访问会报错

为什么普通 for 循环的性能远远高于 forEach 的性能，请解释其中的原因

区别：

一个按顺序遍历，一个使用 iterator 迭代器遍历；

从数据结构来说，for 循环是随机访问元素，foreach 是顺序链表访问元素

从性能上来说，对于 arraylist 是顺序表，使用 for 循环可以顺序访问，速度较快，使用 foreach 会比 for 循环稍慢一些。对于 linkedlist 是单链表，使用 for 循环每次都要从第一个元素读取 next 域来读取，速度非常慢，使用 foreach 可以直接读取当前结点，数据较快

箭头函数与普通函数的区别是什么?构造函数可以使用 new 生成示例，那么箭头函数可以吗?为什么?

箭头函数是普通函数的简写，可以更优雅的定义一个函数，和普通函数相比区别:

1. 函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象，它会从自己的作用域链的上一层继承 this（因此无法使用 apply/call/bind 进行绑定 this 值）
2. 箭头函数不可以使用 arguments 对象，该对象在函数体内不存在，如果要用，可以用 rest 参数代替
3. 不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数
4. 不绑定 super 和 new.target
5. 不可以使用 new 命令，因为没有自己的 this，无法调用 call,apply，没有 prototype 属性，而 new 命令在执行时需要将构造函数的 prototype 赋值给新的对象的 proto

请解释为什么 function foo(){ }();不是 IIFE (立即调用的函数表达式)，要做哪些改动使它变成 IIFE

以 function 关键字开头的语句会被解析成函数声明，而函数声明是不允许直接运行的。

原因:

Js"预编译"的特点：js 在预编译阶段，会解释函数声明，但却会忽略表达式

当 js 执行到 function(){//code}();时，由于 function(){//code};在预编译阶段已经被解释过，js 会跳过 function(){//code};试图去执行()，故而报错

只有当解析器把这句话解析为函数表达式，才能够直接运行

实现:

```
(function(){//code})()
! function(){//code}()
+ function(){//code}()
```

IIFE 的一些作用:v

创建作用域，内部保存一些大量临时变量的代码防止命名冲突。

一些库的外层用这种形式包起来防止作用域污染。

运行一些只执行一次的代码

for of , for in 和 forEach,map 的区别

for…of：循环：具有 iterator 接口，就可以用 for…of 循环遍历它的成员(属性值)。for…of 循环可以使用的范围包括数组，Set 和 Map 结构，某些类似数组的对象，Generator 对象，以及字符串，for…of 循环调用遍历器接口，数组的遍历器接口只返回具有数字索引的属性，对于普通的对象，for…of 结构不能直接使用，会报错，必须部署了 Iterator 接口后才能使用，可以中断循环

for…of：循环：遍历对象自身的和继承的可枚举的属性，不能直接获取属性值，可以中断循环

forEach：只能遍历数组，不能中断，没有返回值（或认为返回值是 undefined）

map：只能遍历数组，不能中断，返回值是修改后的数组

介绍下深度优先遍历和广度优先遍历，如何实现

深度优先遍历：

深度优先遍历，是搜索算法的一种，它沿着树的深度遍历树的节点，尽可能深地搜索树的分支，当节点 v 的所有边都已被探寻过，将回溯到发现节点 v 的那条边的起始节点，这一过程一直进行到已探寻源节点到其他所有节点为止，如果还有未被发现的节点，则选择其中一个未被发现的节点为源节点并重复以上操作，直到所有节点都被探寻完成。

简单的说，DFS 就是从图中的一个节点开始追溯，直到最后一个节点，然后回溯，继续追溯下一条路径，直到到达所有的节点，如此往复，直到没有路径为止。

DFS 可以产生相应图的拓扑排序表，利用拓扑排序表可以解决很多问题，例如最大路径问题。一般用堆数据结构来辅助实现 DFS 算法。

注意：深度 DFS 属于盲目搜索，无法保证搜索到的路径为最短路径，也不是在搜索特定的路径，而是通过搜索来查看图中有哪些路径可以选择。

步骤：

访问顶点 v

依次从 v 的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和 v 有路径相通的顶点都被访问

若此时途中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到所有顶点均被访问过为止。

实现：

```
Graph.prototype.dfs = function(){
    Var marked = []
    For(var i=0;i<this.vertices.length;i++){
        If(!marked[this.vertices[i]]){
            dfsVisit(this.vertices[i])
        }
    }
    function dfsVisit(u) {
        let edges = this.edges
        marked[u] = true
        console.log(u)
        var neighbors = edges.get(u)
        for (var i=0; i<neighbors.length; i++) {
            var w = neighbors[i]
            if (!marked[w]) {
                dfsVisit(w)
            }
        }
    }
}
```

广度优先遍历

广度优先遍历是从根节点开始，沿着图的宽度遍历节点，如果所有节点均被访问过，则算法终止，BFS 同样属于盲目搜索，一般用队列数据结构来辅助实现 BFS

BFS 从一个节点开始，尝试访问尽可能靠近它的目标节点。本质上这种遍历在图上是逐层移动的，首先检查最靠近第一个节点的层，再逐渐向下移动到离起始节点最远的层
步骤：
创建一个队列，并将开始节点放入队列中
若队列非空，则从队列中取出第一个节点，并检测它是否为目标节点
若是目标节点，则结束搜寻，并返回结果
若不是，则将它所有没有被检测过的字节点都加入队列中
若队列为空，表示图中并没有目标节点，则结束遍历
实现：

```
Graph.prototype.bfs = function(v) {
    var queue = [], marked = []
    marked[v] = true
    queue.push(v) //  添加到队尾
    while(queue.length > 0) {
        var s = queue.shift() //  从队首移除
        if (this.edges.has(s)) {
            console.log('visited vertex: ', s)
        }
        let neighbors = this.edges.get(s)
        for(let i=0;i<neighbors.length;i++) {
            var w = neighbors[i]
            if (!marked[w]) {
                marked[w] = true
                queue.push(w)
            }
        }
    }
}
```

Object.assion()和...解构的区别
Object.assion()用于对象合并，并返回一个新对象
…三点解构也用于返回一个新对象。两者都可以进行部分深拷贝对象

```
var obj = {a: 1, b: 2, c: { a: 3 },d: [4, 5]}
var obj1 = obj
var obj2 = JSON.parse(JSON.stringify(obj))//深拷贝常用方法
var obj3 = {...obj}
var obj4 = Object.assign({},obj)
obj.a = 999
obj.c.a = -999
obj.d[0] = 123
console.log(obj1) //{a: 999, b: 2, c: { a: -999 },d: [123, 5]}
console.log(obj2) //{a: 1, b: 2, c: { a: 3 },d: [4, 5]}
console.log(obj3) //{a: 1, b: 2, c: { a: -999 },d: [123, 5]}
```

```
console.log(obj4) //{a: 1, b: 2, c: { a: -999 },d: [123, 5]}
```

数组

```
var arr = [1, 2, 3, [4, 5], {a: 6, b: 7}]
var arr1 = JSON.parse(JSON.stringify(arr))//深拷贝常用方法
var arr2 = arr
var arr3 = [...arr]
var arr4 = Object.assign([],arr)
console.log(arr === arr1) //false
console.log(arr === arr2) //true
console.log(arr === arr3) //false
console.log(arr === arr4) //false
arr[0]= 999
arr[3][0]= -999
arr[4].a = 123
console.log(arr1) //[1, 2, 3, [4, 5], {a: 6, b: 7}]
console.log(arr2) //[999, 2, 3, [-999, 5], {a: 123, b: 7}]
console.log(arr3) //[1, 2, 3, [-999, 5], {a: 123, b: 7}]
console.log(arr4) //[1, 2, 3, [-999, 5], {a: 123, b: 7}]
```

什么是内存泄露,该如何避免?

内存泄漏指任何对象在您不再拥有或需要它之后仍然存在。

4 类常见内存泄漏:

1、意外的全局变量

解决办法在 JavaScript 文件头部加上 'use strict'，使用严格模式避免意外的全局变量，此时上例中的 this 指向 undefined。如果必须使用全局变量存储大量数据时，确保用完以后把它设置为 null 或者重新定义。

2.被遗忘的计时器或回调函数

解决办法：现代的浏览器（包括 IE 和 Microsoft Edge）使用了更先进的垃圾回收算法（标记清除），已经可以正确检测和处理循环引用了。即回收节点内存时，不必非要调用 removeEventListener 了。

3、脱离 DOM 的引用

4、闭包

使用后给变量复制 null

5.事件监听：没有正确销毁（低版本浏览器可能出现）

XML 和 JSON 的区别?

数据体积方面

JSON 相对于 XML 来讲，数据的体积小，传递的速度更快些。

数据交互方面

JSON 与 JavaScript 的交互更加方便，更容易解析处理，更好的数据交互

数据描述方面

JSON 对数据的描述性比 XML 较差

传输速度方面

JSON 的速度要远远快于 XML

附加：关于序列化，有下面五点注意事项

1. 非数组对象的属性不能保证以特定的顺序出现在序列化后的字符串中。
2. 布尔值，数组，字符串的包装对象在序列化过程中会自动转换成对应的原始值
3. Undefined,任意的函数以及 symbol 值，在序列化过程中会被忽略(出现在非数组对象的属性值中时)或者被转换成 null(出现在数组中时)。
4. 所有以 symbol 为属性键的属性都会被忽略掉，即使 replacer 参数中强制指定包含了它们。
5. 不可枚举的属性都会被忽略

new 的原理是什么?通过 new 的方式创建对象和通过字面量创建有什么区别

创建一个新对象。

这个新对象会被执行[[原型]]连接。

将构造函数的作用域赋值给新对象，即 this 指向这个新对象.

如果函数没有返回其他对象，那么 new 表达式中的函数调用会自动返回这个新对象。

字面量创建对象，不会调用 Object 构造函数，简洁且性能更好;

new Object() 方式创建对象本质上是方法调用，涉及到在 proto 链中遍历该方法，当找到该方法后，又会生产方法调用必须的 堆栈信息，方法调用结束后，还要释放该堆栈，性能不如字面量的方式。

通过对象字面量定义对象时，不会调用 Object 构造函数

手动实现 new 操作

```
function Foo(name, age) {
   this.name = name;
   this.age = age
}
var nar = new Foo('tom', 18)
console.log(nar.name)
function OBK() {
   var obj = new Object(),//从 Object.prototype 上克隆一个对象
    Constructor = [].shift.call(arguments);//取得外部传入的构造器
   var F=function(){};
   F.prototype= Constructor.prototype;
   obj=new F();//指向正确的原型
   var ret = Constructor.apply(obj, arguments);//借用外部传入的构造器给 obj 设置属性
   return typeof ret === 'object' ? ret : obj;//确保构造器总是返回一个对象
}
var bar = OBK(Foo, 'jim', 15)
console.log(bar.age)
```

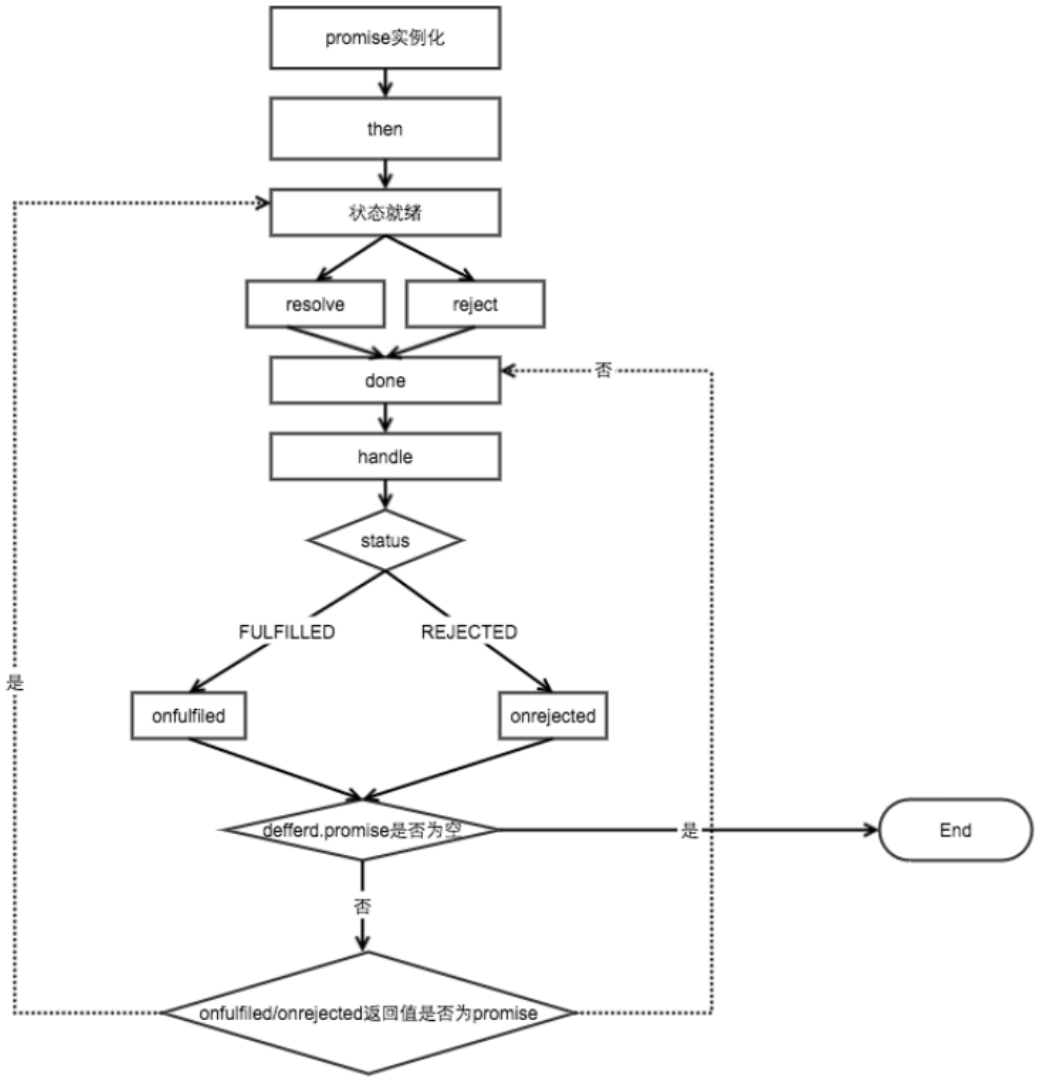
从头手写 promise

相关概念:

1. 术语

解决(fulfill)：指一个 promise 成功时进行的一系列操作，如状态的改变，回调的执行，虽然

规范中有 fulfill 来表示解决，但在后世的 promise 实现多以 resolve 来指代
拒绝(reject)：指一个 promise 失败时进行的一系列操作
终值(eventual value)：所谓终值，指的是 promise 被解决时传递给解决回调的值，由于 promise 有一次性的特征，因此当这个值被传递时，标志着 promise 等待态的结束，故称之终值，有时也直接简称为值(value)
拒因(reason)：也就是拒绝的原因，指在 promise 被拒绝时传递给拒绝回调的值
2. 执行流程



每个 promise 后面链一个对象，该对象包含 onfulfiled,onrejected,子 promise 三个属性，当父 promise 状态改变完毕，执行完相应的 onfuliled/onfuliled 的时候呢，拿到子 promise，在等待这个子 promise 状态改变，再执行相应的 onfuliled/onfuliled，以此循环直到当前 promise 没有子 promise
3. 状态机制切换
状态只能由 pengding-->fulfilled，或者由 pending-->rejected 这样转变。
只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的
4. 基本结构

```
function Promise(executor) {
        this.state = 'pending'; //状态
        this.value = undefined; //成功结果
        this.reason = undefined; //失败原因
        function resolve(value) { }
        function reject(reason) { }
        executor(resolve, reject) //立即执行
    }
```

接收一个 executor 函数, executor 函数传入就执行（当我们实例化一个 promise 时, executor 立即执行），执行完同步或异步操作后，调用它的两个参数 resolve 和 reject。其中 state 保存了 promise 的状态, 包含三个状态: 等待态(pending)成功态(resolved)和失败态(rejected)。promise 执行成功后的结果由 value 保存，失败后的原因由 reason 保存。

5. 完善 resolve 与 reject

new Promise((resolve, reject)=>{resolve(value)}) resolve 为成功，接收参数 value，状态改变为 fulfilled，不可再次改变。

new Promise((resolve, reject)=>{reject(reason)}) reject 为失败，接收参数 reason，状态改变为 rejected，不可再次改变。

若是 executor 函数报错 直接执行 reject()

我们可以这样实现:

```
function Promise(executor) {
        this.state = 'pending'; //状态
        this.value = undefined; //成功结果
        this.reason = undefined; //失败原因
        resolve = (value) => {
            // state 改变,resolve 调用就会失败
            if (this.state === 'pending') {
                // resolve 调用后，state 转化为成功态
                this.state = 'fulfilled';
                //  储存成功的值
                this.value = value;
            }
        }
        reject = (reason) => {
            // state 改变,reject 调用就会失败
            if (this.state === 'pending') {
                // reject 调用后，state 转化为失败态
                this.state = 'rejected';
                //  储存失败的原因
                this.reason = reason;
            }
        }
        //如果 executor 执行报错，直接执行 reject
        try {
            executor(resolve, reject)
```

```
        } catch (err) {
            reject(err)  // executor 出错就直接调用
        }
    }
```

6. 实现 then 方法

每一个 Promise 实例都有一个 then 方法，接收两个为函数的参数，它用来处理异步返回的结果，它是定义在原型上的方法。

```
Promise.prototype.then = function (onFulfilled, onRejected) {};
```

当 promise 的状态发生了变化，不论成功或失败都会调用 then 方法，因此 then 方法里面也会根据不同的状态来判断调用哪一个回调函数。

两个参数的注意事项：

onFulfilled 和 onRejected 都是可选参数，也就是说可以传也可以不传。传入的回调函数如不是一个函数类型，可以直接忽略。

两个参数在 promise 执行结束前其不可被调用，其调用次数不可超过一次。

```
Promise.prototype.then = function (onFulfilled, onRejected) {
        if (this.state === 'fulfilled') {
          //判断参数类型，是函数执行之，如果 onFulfilled 不是函数，其必须被忽略
            if (typeof onFulfilled === 'function') {
                onFulfilled(this.value);  // 传入成功的值
            }
        }
       // 如果 onRejected 不是函数，其必须被忽略
        if (this.state === 'rejected') {
            if (typeof onRejected === 'function') {
                onRejected(this.reason); // 传入失败的原因
            }
        }
    };
```

7. 支持异步实现

上述把 promise 的基本功能都实现了，但是还是会存在一个问题，就是 promise 不支持异步代码，当 resolve 或 reject 在 setTimeout 中实现时，调用 then 方法时，此时状态仍然是 pengding，then 方法即没有调用 onFulfilled 也没有调用 onRejected，也就运行没有任何结果。

我们可以参照发布订阅模式，在执行 then 方法时状态还是状态还是 pengding 时，把回调函数存储在一个数组中，当状态发生改变时依次从数组中取出执行就好了，首先在类上新增两个 Array 类型的数组，用于存放回调函数。

```
function Promise(executor) {
        this.state = 'pending'; //状态
        this.value = undefined; //成功结果
        this.reason = undefined; //失败原因
        this.onFulfilledFunc = [];//保存成功回调
        this.onRejectedFunc = [];//保存失败回调
        function resolve(value) {
          // ....
```

```
        }
        function reject(reason) {
            // ....
        }
        executor(resolve, reject) //立即执行
    }
```

并修改 then 方法

```
Promise.prototype.then = function (onFulfilled, onRejected) {
        if (this.state === 'pending') {
            if (typeof onFulfilled === 'function') {
                this.onFulfilledFunc.push(onFulfilled);//保存回调
            }
            if (typeof onRejected === 'function') {
                this.onRejectedFunc.push(onRejected);//保存回调
            }
        }
        if (this.state === 'fulfilled') {
            //判断参数类型，是函数执行之，如果 onFulfilled 不是函数，其必须被忽略
            if (typeof onFulfilled === 'function') {
                onFulfilled(this.value);  // 传入成功的值
            }
        }
        // 如果 onRejected 不是函数，其必须被忽略
        if (this.state === 'rejected') {
            if (typeof onRejected === 'function') {
                onRejected(this.reason); // 传入失败的原因
            }
        }
    };
```

修改 resolve 和 reject 方法：

```
function Promise(executor) {
        //  其他代码
        function resolve(value) {
            // state 改变,resolve 调用就会失败
            if (this.state === 'pending') {
                // resolve 调用后，state 转化为成功态
                this.state = 'fulfilled';
                // 储存成功的值
                this.value = value;
                this.onFulfilledFunc.forEach(fn => fn(value))
            }
        }
        function reject(reason) {
            // state 改变,reject 调用就会失败
```

```
                if (this.state === 'pending') {
                    // reject 调用后，state 转化为失败态
                    this.state = 'rejected';
                    //  储存失败的原因
                    this.reason = reason;
                    this.onRejectedFunc.forEach(fn => fn(reason))
                }
            }
            //   其他代码
        }
```

到这里 Promise 已经支持了异步操作了。

8. 链式调用实现

光是实现了异步操作可不行，我们常常用到 new Promise().then().then()这样的链式调用来解决回调地狱。

规范如何定义 then 方法：

每个 then 方法都返回一个新的 Promise 对象（原理的核心）

如果 then 方法中显示地返回了一个 Promise 对象就以此对象为准，返回它的结果

如果 then 方法中返回的是一个普通值（如 Number、String 等）就使用此值包装成一个新的 Promise 对象返回。

如果 then 方法中没有 return 语句，就视为返回一个用 Undefined 包装的 Promise 对象

若 then 方法中出现异常，则调用失败态方法（reject）跳转到下一个 then 的 onRejected

如果 then 方法没有传入任何回调，则继续向下传递（值的传递特性）

总的来说就是不论何时 then 方法都要返回一个 Promise，这样才能调用下一个 then 方法。

我们可以实例化一个 promise2 返回，将这个 promise2 返回的值传递到下一个 then 中。

```
Promise.prototype.then = function (onFulfilled, onRejected) {
    let promise2 = new Promise((resolve, reject) => {
    //  其他代码
    }
    return promise2;
};
```

接下来就处理根据上一个 then 方法的返回值来生成新 Promise 对象.

```
/**
 * 解析 then 返回值与新 Promise 对象
 * @param {Object} promise2  新的 Promise 对象
 * @param {*} x  上一个 then 的返回值
 * @param {Function} resolve promise2 的 resolve
 * @param {Function} reject promise2 的 reject
 */
function resolvePromise(promise2, x, resolve, reject) {
    //...
}
```

当 then 的返回值与新生成的 Promise 对象为同一个（引用地址相同），状态永远为等待态（pending），再也无法成为 resolved 或是 rejected，程序会死掉,则会抛出 TypeError 错误

```
let promise2 = p.then(data => {
```

```
    return promise2;
});
// TypeError: Chaining cycle detected for promise #<Promise>
```
因此需要判断 x。
x 不能和新生成的 promise 对象为同一个
x 不能是 null，可以是对象或者函数(包括 promise)，否则是普通值,那么直接 resolve(x)
当 x 是对象或者函数（默认 promise）则声明 then，let then = x.then
如果取 then 报错，则走 reject()
如果 then 是个函数，则用 call 执行 then，第一个参数是 this，后面是成功的回调和失败的回调，成功和失败只能调用一个 所以设定一个 called 来防止多次调用
如果成功的回调还是 pormise，就递归继续解析
小提示： 为什么取对象上的属性有报错的可能？Promise 有很多实现（bluebird，Q 等），Promises/A+只是一个规范，大家都按此规范来实现 Promise 才有可能通用，因此所有出错的可能都要考虑到，假设另一个人实现的 Promise 对象使用 Object.defineProperty()恶意的在取值时抛错，我们可以防止代码出现 Bug
resolvePromise 实现

```
function resolvePromise(promise2, x, resolve, reject) {
        if (promise2 === x) {   // 1.x 不能等于 promise2
            reject(new TypeError('Promise 发生了循环引用'));
        }
        let called;
        if (x !== null && (typeof x === 'object' || typeof x === 'function')) {
            // 2. 可能是个对象或是函数
            try {
                let then = x.then;// 3.取出 then 方法引用
                if (typeof then === 'function') {   // 此时认为 then 是一个 Promise 对象
                    //then 是 function，那么执行 Promise
                    then.call(x, (y) => {    // 5.使用 x 作为 this 来调用 then 方法，即 then 里面的 this 指向 x
                        if (called) return;
                        called = true;
                        // 6.递归调用，传入 y 若是 Promise 对象，继续循环
                        resolvePromise(promise2, y, resolve, reject);
                    }, (r) => {
                        if (called) return;
                        called = true;
                        reject(r);
                    });
                } else {
                    resolve(x);
                }
            } catch (e) {
                // 也属于失败
```

```
                    if (called) return;
                    called = true;
                    reject(e); // 4.取 then 报错，直接 reject
                }

            } else {
                //否则是个普通值
                resolve(x);
            }
        }
```

此时链式调用支持已经实现，在相应的地方调用 resolvePromise 方法即可。

9. 最后完善

规范还对 onFulfilled 和 onRejected 有规定

onFulfilled 返回一个普通的值，成功时直接等于 value => value

onRejected 返回一个普通的值，失败时如果直接等于 value => value，则会跑到下一个 then 中的 onFulfilled 中，所以直接扔出一个错误 reason => throw err

onFulfilled 或 onRejected 不能同步被调用，必须异步调用。我们就用 setTimeout 解决异步问题

如果 onFulfilled 或 onRejected 报错，则直接返回 reject()

完善 then 方法

```
Promise.prototype.then = function (onFulfilled, onRejected) {
        let promise2 = new Promise((resolve, reject) => {
            // onFulfilled 如果不是函数，就忽略 onFulfilled，直接返回 value
            onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : value => value;
            // onRejected 如果不是函数，就忽略 onRejected，直接扔出错误
            onRejected = typeof onRejected === 'function' ? onRejected : err => { throw err };
            if (this.state === 'pending') {
                this.onFulfilledFunc.push(() => {
                    //  异步
                    setTimeout(() => {
                        try {
                            let x = onFulfilled(this.value);
                            resolvePromise(promise2, x, resolve, reject);
                        } catch (e) {
                            reject(e);
                        }
                    }, 0);
                })
                this.onRejectedFunc.push(() => {
                    //  异步
                    setTimeout(() => {
                        try {
                            let x = onRejected(this.value);
```

```
                                resolvePromise(promise2, x, resolve, reject);
                            } catch (e) {
                                reject(e);
                            }
                        }, 0);
                    })
                }
                if (this.state === 'fulfilled') {
                    // 异步
                    setTimeout(() => {
                        try {
                            let x = onFulfilled(this.value);
                            resolvePromise(promise2, x, resolve, reject);
                        } catch (e) {
                            reject(e);
                        }
                    }, 0);
                }
                if (this.state === 'rejected') {
                    // 异步
                    setTimeout(() => {
                        // 如果报错
                        try {
                            let x = onRejected(this.reason);
                            resolvePromise(promise2, x, resolve, reject);
                        } catch (e) {
                            reject(e);
                        }
                    }, 0);
                }
            })
            return promise2;
        };
```

到这里手写一个 Promise 已经全部实现了

完整代码

```
function Promise(executor) {
            this.state = 'pending'; //状态
            this.value = undefined; //成功结果
            this.reason = undefined; //失败原因
            this.onFulfilledFunc = [];//保存成功回调
            this.onRejectedFunc = [];//保存失败回调
            resolve = (value) => {
                // state 改变,resolve 调用就会失败
                if (this.state === 'pending') {
```

```
                    // resolve 调用后，state 转化为成功态
                    this.state = 'fulfilled';
                    //  储存成功的值
                    this.value = value;
                    this.onFulfilledFunc.forEach(fn => fn(value))
                }
            }
            reject = (reason) => {
                // state 改变,reject 调用就会失败
                if (this.state === 'pending') {
                    // reject 调用后，state 转化为失败态
                    this.state = 'rejected';
                    //  储存失败的原因
                    this.reason = reason;
                    this.onRejectedFunc.forEach(fn => fn(reason))
                }
            }
            //如果 executor 执行报错，直接执行 reject
            try {
                executor(resolve, reject)
            } catch (err) {
                reject(err)    // executor 出错就直接调用
            }
        }
        Promise.prototype.then = function (onFulfilled, onRejected) {
            let promise2 = new Promise((resolve, reject) => {
                // onFulfilled 如果不是函数，就忽略 onFulfilled，直接返回 value
                onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : value => value;
                // onRejected 如果不是函数，就忽略 onRejected，直接扔出错误
                onRejected = typeof onRejected === 'function' ? onRejected : err =>
{ throw err };
                if (this.state === 'pending') {
                    this.onFulfilledFunc.push(() => {
                        //  异步
                        setTimeout(() => {
                            try {
                                let x = onFulfilled(this.value);
                                resolvePromise(promise2, x, resolve, reject);
                            } catch (e) {
                                reject(e);
                            }
                        }, 0);
                    })
                    this.onRejectedFunc.push(() => {
```

```
                // 异步
                setTimeout(() => {
                    try {
                        let x = onRejected(this.value);
                        resolvePromise(promise2, x, resolve, reject);
                    } catch (e) {
                        reject(e);
                    }
                }, 0);
            })
        }
        if (this.state === 'fulfilled') {
            // 异步
            setTimeout(() => {
                try {
                    let x = onFulfilled(this.value);
                    resolvePromise(promise2, x, resolve, reject);
                } catch (e) {
                    reject(e);
                }
            }, 0);
        }
        if (this.state === 'rejected') {
            // 异步
            setTimeout(() => {
                // 如果报错
                try {
                    let x = onRejected(this.reason);
                    resolvePromise(promise2, x, resolve, reject);
                } catch (e) {
                    reject(e);
                }
            }, 0);
        }
    })
    return promise2;
};
function resolvePromise(promise2, x, resolve, reject) {
    if (promise2 === x) {
        reject(new TypeError('Promise 发生了循环引用'));
    }
    let called;
    if (x !== null && (typeof x === 'object' || typeof x === 'function')) {
        //可能是个对象或是函数
```

```
                try {
                    let then = x.then;//取出 then 方法引用
                    if (typeof then === 'function') {  // 认为 then 是一个 Promise 对象
                        //then 是 function，那么执行 Promise
                        then.call(x, (y) => {
                            // 成功和失败只能调用一个
                            if (called) return;
                            called = true;
                            //递归调用，传入 y 若是 Promise 对象，继续循环
                            resolvePromise(promise2, y, resolve, reject);
                        }, (r) => {
                            // 成功和失败只能调用一个
                            if (called) return;
                            called = true;
                            reject(r);
                        });
                    } else {
                        resolve(x);
                    }
                } catch (e) {
                    // 也属于失败
                    if (called) return;
                    called = true;
                    reject(e);
                }

            } else {
                //否则是个普通值
                resolve(x);
            }
        }
```

但是只用构造函数实现当然是不够的，我们再用 class 来实现一个 Promise，基本原理同上 class 实现

```
class Promise {
            constructor(executor) {
                this.state = 'pending';
                this.value = undefined;
                this.reason = undefined;
                this.onResolvedCallbacks = [];
                this.onRejectedCallbacks = [];
                let resolve = value => {
                    if (this.state === 'pending') {
                        this.state = 'fulfilled';
                        this.value = value;
```

```
                this.onResolvedCallbacks.forEach(fn => fn());
            }
        };
        let reject = reason => {
            if (this.state === 'pending') {
                this.state = 'rejected';
                this.reason = reason;
                this.onRejectedCallbacks.forEach(fn => fn());
            }
        };
        try {
            executor(resolve, reject);
        } catch (err) {
            reject(err);
        }
    }
    then(onFulfilled, onRejected) {
        onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : value => value;
        onRejected = typeof onRejected === 'function' ? onRejected : err =>
{ throw err };
        let promise2 = new Promise((resolve, reject) => {
            if (this.state === 'fulfilled') {
                setTimeout(() => {
                    try {
                        let x = onFulfilled(this.value);
                        resolvePromise(promise2, x, resolve, reject);
                    } catch (e) {
                        reject(e);
                    }
                }, 0);
            };
            if (this.state === 'rejected') {
                setTimeout(() => {
                    try {
                        let x = onRejected(this.reason);
                        resolvePromise(promise2, x, resolve, reject);
                    } catch (e) {
                        reject(e);
                    }
                }, 0);
            };
            if (this.state === 'pending') {
                this.onResolvedCallbacks.push(() => {
                    setTimeout(() => {
```

```
                        try {
                            let x = onFulfilled(this.value);
                            resolvePromise(promise2, x, resolve, reject);
                        } catch (e) {
                            reject(e);
                        }
                    }, 0);
                });
                this.onRejectedCallbacks.push(() => {
                    setTimeout(() => {
                        try {
                            let x = onRejected(this.reason);
                            resolvePromise(promise2, x, resolve, reject);
                        } catch (e) {
                            reject(e);
                        }
                    }, 0)
                });
            };
        });
        return promise2;
    }
    catch(fn) {
        return this.then(null, fn);
    }
}
function resolvePromise(promise2, x, resolve, reject) {
    if (x === promise2) {
        return reject(new TypeError('Chaining cycle detected for promise'));
    }
    let called;
    if (x != null && (typeof x === 'object' || typeof x === 'function')) {
        try {
            let then = x.then;
            if (typeof then === 'function') {
                then.call(x, y => {
                    if (called) return;
                    called = true;
                    resolvePromise(promise2, y, resolve, reject);
                }, err => {
                    if (called) return;
                    called = true;
                    reject(err);
                })
```

```
                } else {
                    resolve(x);
                }
            } catch (e) {
                if (called) return;
                called = true;
                reject(e);
            }
        } else {
            resolve(x);
        }
}
//resolve 方法
Promise.resolve = function (val) {
    return new Promise((resolve, reject) => {
        resolve(val)
    });
}
//reject 方法
Promise.reject = function (val) {
    return new Promise((resolve, reject) => {
        reject(val)
    });
}
//race 方法
Promise.race = function (promises) {
    return new Promise((resolve, reject) => {
        for (let i = 0; i < promises.length; i++) {
            promises[i].then(resolve, reject)
        };
    })
}
//all 方法(获取所有的 promise，都执行 then，把结果放到数组，一起返回)
Promise.all = function (promises) {
    let arr = [];
    let i = 0;
    function processData(index, data) {
        arr[index] = data;
        i++;
        if (i == promises.length) {
            resolve(arr);
        };
    };
    return new Promise((resolve, reject) => {
```

```
            for (let i = 0; i < promises.length; i++) {
                promises[i].then(data => {
                    processData(i, data);
                }, reject);
            };
        });
    }
```

简单实现 async/await 中的 async 函数

```
function spawn(genF) {
    return new Promise(function(resolve, reject) {
        const gen = genF();
        function step(nextF) {
            let next;
            try {
                next = nextF();
            } catch (e) {
                return reject(e);
            }
            if (next.done) {
                return resolve(next.value);
            }
            Promise.resolve(next.value).then(
                function(v) {
                    step(function() {
                        return gen.next(v);
                    });
                },
                function(e) {
                    step(function() {
                        return gen.throw(e);
                    });
                }
            );
        }
        step(function() {
            return gen.next(undefined);
        });
    });
}
```

图片懒加载与预加载

图片懒加载的原理是暂时不设置图片的 src 属性，而是将图片的 url 隐藏起来，比如先写在 data-src 里面，等某些事件触发的时候（比如滚动到底部，点击加载图片）再将图片真实的

url 放进 src 属性里面，从而实现图片的延迟加载
图片预加载，是指在一些需要展示大量图片的网站，实现图片的提前加载，从而提升用户体验，常用的方式有两种，一种是隐藏在 css 的 background 的 url 属性里面，一种是通过 JavaScript 的 Image 对象设置实例对象耳朵 src 属性实现图片的预加载，相关代码如下：
css 实现

```
#preload-01 { background: url(http://domain.tld/image-01.png) no-repeat -9999px -9999px; }
#preload-02 { background: url(http://domain.tld/image-02.png) no-repeat -9999px -9999px; }
#preload-03 { background: url(http://domain.tld/image-03.png) no-repeat -9999px -9999px; }
```

Javascript 预加载图片的方式：

```
function preloadImg(url) {
    var img = new Image();
    img.src = url;
    if(img.complete) {
        //接下来可以使用图片了
        //do something here
    } else {
        img.onload = function() {
            //接下来可以使用图片了
            //do something here
        };
    }
}
```

预加载：提前加载图片，当用户需要查看时可直接从本地缓存中渲染。
懒加载：懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数。
两种技术的本质：两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。
懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力

input 搜索如何防抖，如何处理中文输入
通过 compositionstart & compositionend 做的中文输入处理

```
<input ref="input"
@compositionstart="handleComposition"
@compositionupdate="handleComposition"
@compositionend="handleComposition">
```

compositionstart 事件触发于一段文字的输入之前（类似于 keydown 事件，但是该事件仅在若干可见字符的输入之前，而这些可见字符的输入可能需要一连串的键盘操作、语音识别或者点击输入法的备选词）
每打一个拼音字母，触发 compositionupdate，最后将输入好的中文填入 input 中时触发 compositionend,
触发 compositionstart 时，文本框会填入“虚拟文本”（待确认文本），同时触发 input 事件.
填入实际内容后（已确认文本）,触发 compositionend,所以这里如果不想触发 input 事件的

话就得设置一个 bool 变量来控制。

['1', '2', '3'].map(parseInt) what & why ?

map 函数的第一个参数 callback：var new_array = arr.map(function callback(currentValue[, index[, array]]) { // Return element for new_array }[, thisArg])

这个 callback 一共可以接收三个参数，其中第一个参数代表当前被处理的元素，而第二个参数代表该元素的索引。而 parseInt 则是用来解析字符串的，使字符串成为指定基数的整数。

parseInt(string, radix)

接收两个参数，第一个表示被处理的值（字符串），第二个表示为解析时的基数。

了解这两个函数后，我们可以模拟一下运行情况

parseInt('1', 0) //radix 为 0 时，且 string 参数不以“0x”和“0”开头时，按照 10 为基数处理。这个时候返回 1

parseInt('2', 1) //基数为 1（1 进制）表示的数中，最大值小于 2，所以无法解析，返回 NaN

parseInt('3', 2) //基数为 2（2 进制）表示的数中，最大值小于 3，所以无法解析，返回 NaN

map 函数返回的是一个数组，所以最后结果为[1, NaN, NaN]

请分别用深度优先思想和广度优先思想实现一个拷贝函数?

```
//工具函数
let _toString = Object.prototype.toString
let map = {
   array: 'Array',
   object: 'Object',
   function: 'Function',
   string: 'String',
   null: 'Null',
   undefined: 'Undefined',
   boolean: 'Boolean',
   number: 'Number'
}
let getType = (item) => {
   return _toString.call(item).slice(8, -1)
}
let isTypeOf = (item, type) => {
   return map[type] && map[type] === getType(item)
}
深度复制
let DFSdeepClone = (obj, visitedArr = []) => {
   let _obj = {}
   if (isTypeOf(obj, 'array') || isTypeOf(obj, 'object')) {
      let index = visitedArr.indexOf(obj)
      _obj = isTypeOf(obj, 'array') ? [] : {}
      if (~index) { //  判断环状数据
         _obj = visitedArr[index]
      } else {
```

```
            visitedArr.push(obj)
            for (let item in obj) {
                _obj[item] = DFSdeepClone(obj[item], visitedArr)
            }
        }
    } else if (isTypeOf(obj, 'function')) {
        _obj = eval('(' + obj.toString() + ')');
    } else {
        _obj = obj
    }
    return _obj
}
广度复制
let BFSdeepClone = (obj) => {
    let origin = [obj],
        copyObj = {},
        copy = [copyObj]
        // 去除环状数据
    let visitedQueue = [],
        visitedCopyQueue = []
    while (origin.length > 0) {
        let items = origin.shift(),
            _obj = copy.shift()
        visitedQueue.push(items)
        visitedCopyQueue.push(_obj)
        if (isTypeOf(items, 'object') || isTypeOf(items, 'array')) {
            for (let item in items) {
                let val = items[item]
                if (isTypeOf(val, 'object')) {
                    let index = visitedQueue.indexOf(val)
                    if (!~index) {
                        _obj[item] = {}
                            //下次 while 循环使用给空对象提供数据
                        origin.push(val)
                            // 推入引用对象
                        copy.push(_obj[item])
                        visitedQueue.push(val)
                    } else {
                        _obj[item] = visitedCopyQueue[index]
                    }
                } else if (isTypeOf(val, 'array')) {
                    // 数组类型在这里创建了一个空数组
                    _obj[item] = []
                    origin.push(val)
```

```
                copy.push(_obj[item])
            } else if (isTypeOf(val, 'function')) {
                _obj[item] = eval('(' + val.toString() + ')');
            } else {
                _obj[item] = val
            }
        }
    } else if (isTypeOf(items, 'function')) {
        copyObj = eval('(' + items.toString() + ')');
    } else {
        copyObj = obj
    }

  }
  return copyObj
}
```

异步笔试题

```
async function async1() {
    console.log('async1 start');
    await async2();
    console.log('async1 end');
}
async function async2() {
    console.log('async2');
}
console.log('script start');
setTimeout(function() {
    console.log('setTimeout');
}, 0)
async1();
new Promise(function(resolve) {
    console.log('promise1');
    resolve();
}).then(function() {
    console.log('promise2');
});
console.log('script end');
/* script start
async1 start
async2
promise1
script end
promise2
```

```
async1 end
setTimeout */
```