三分钟学前端 JS 篇

每日三分钟，进阶一个前端小 Tip！从入门到进阶到资深，循序渐进加深你的知识领域！

项目地址：https://github.com/Advanced-Frontend

如果感觉还不错，欢迎分享给好友，你的每次分享都是对我们最好的支持❤️

最近开源了一个github仓库：百问百答，在工作中很难做到对社群问题进行立即解答，所以可以将问题提交至 https://github.com/Advanced-Frontend/Just-Now-QA ，会定期解答，更多的是鼓励与欢迎更多人一起参与探讨与解答🌹

目录

indexOf 和 findIndex 的区别

`indexOf` 与 `findIndex` 都是查找数组中满足条件的第一个元素的索引

indexOf

> Array.prototype.indexOf()：
>
> `indexOf()` 方法返回在数组中可以找到一个给定元素的第一个索引，如果不存在，则返回-1。
>
> 来自：MDN

例如：

```
const sisters = ['a', 'b', 'c', 'd', 'e'];
console.log(sisters.indexOf('b'));
// 1
```

请注意： **`indexOf()` 使用严格等号(与 `===` 或 triple-equals 使用的方法相同)来比较 `searchElement` 和数组中的元素**

所以，`indexOf` 更多的是用于查找基本类型，如果是对象类型，则是判断是否是同一个对象的引用

```
let sisters = [{a: 1}, {b: 2}];
console.log(sisters.indexOf({b: 2}));
// -1

const an = {b: 2}
sisters = [{a: 1}, an];
console.log(sisters.indexOf(an));
// 1
```

findIndex

> Array.prototype.findIndex()：
>
> `findIndex()` 方法返回数组中满足提供的测试函数的第一个元素的**索引**。若没有找到对应元素则返回-1。
>
> 来自：MDN

```
const sisters = [10, 9, 12, 15, 16];
const isLargeNumber = (element) => element > 13;
console.log(sisters.findIndex(isLargeNumber));
// 3
```

`findIndex` 期望回调函数作为第一个参数。如果你需要非基本类型数组(例如对象)的索引，或者你的查找条件比一个值更复杂，可以使用这个方法。

indexOf 与 findIndex 区别（总结）

- `indexOf`：查找值作为第一个参数，采用 `===` 比较，更多的是用于查找基本类型，如果是对象类型，则是判断是否是同一个对象的引用
- `findIndex`：比较函数作为第一个参数，多用于非基本类型(例如对象)的数组索引查找，或查找条件很复杂

源码实现（加深）

`indexOf`：

```
if (!Array.prototype.indexOf) {
  Array.prototype.indexOf = function(searchElement, fromIndex) {

    var k;
    if (this == null) {
      throw new TypeError('"this" is null or not defined');
    }

    var O = Object(this);
    var len = O.length >>> 0;
    if (len === 0) {
      return -1;
    }

    var n = +fromIndex || 0;
    if (Math.abs(n) === Infinity) {
      n = 0;
    }
    if (n >= len) {
      return -1;
    }
    k = Math.max(n >= 0 ? n : len - Math.abs(n), 0);

    while (k < len) {
      if (k in O && O[k] === searchElement) { // === 匹配
        return k;
      }
      k++;
    }
    return -1;
  };
}
```

`findIndex` :

```
if (!Array.prototype.findIndex) {
  Object.defineProperty(Array.prototype, 'findIndex', {
    value: function(predicate) {
      if (this == null) {
        throw new TypeError('"this" is null or not defined');
      }

      var o = Object(this);
      var len = o.length >>> 0;

      if (typeof predicate !== 'function') {
        throw new TypeError('predicate must be a function');
      }

      var thisArg = arguments[1];

      var k = 0;
      while (k < len) {
        var kValue = o[k];
        if (predicate.call(thisArg, kValue, k, o)) { // 比较函数判断
          return k;
        }
        k++;
      }

      return -1;
    }
  });
}
```

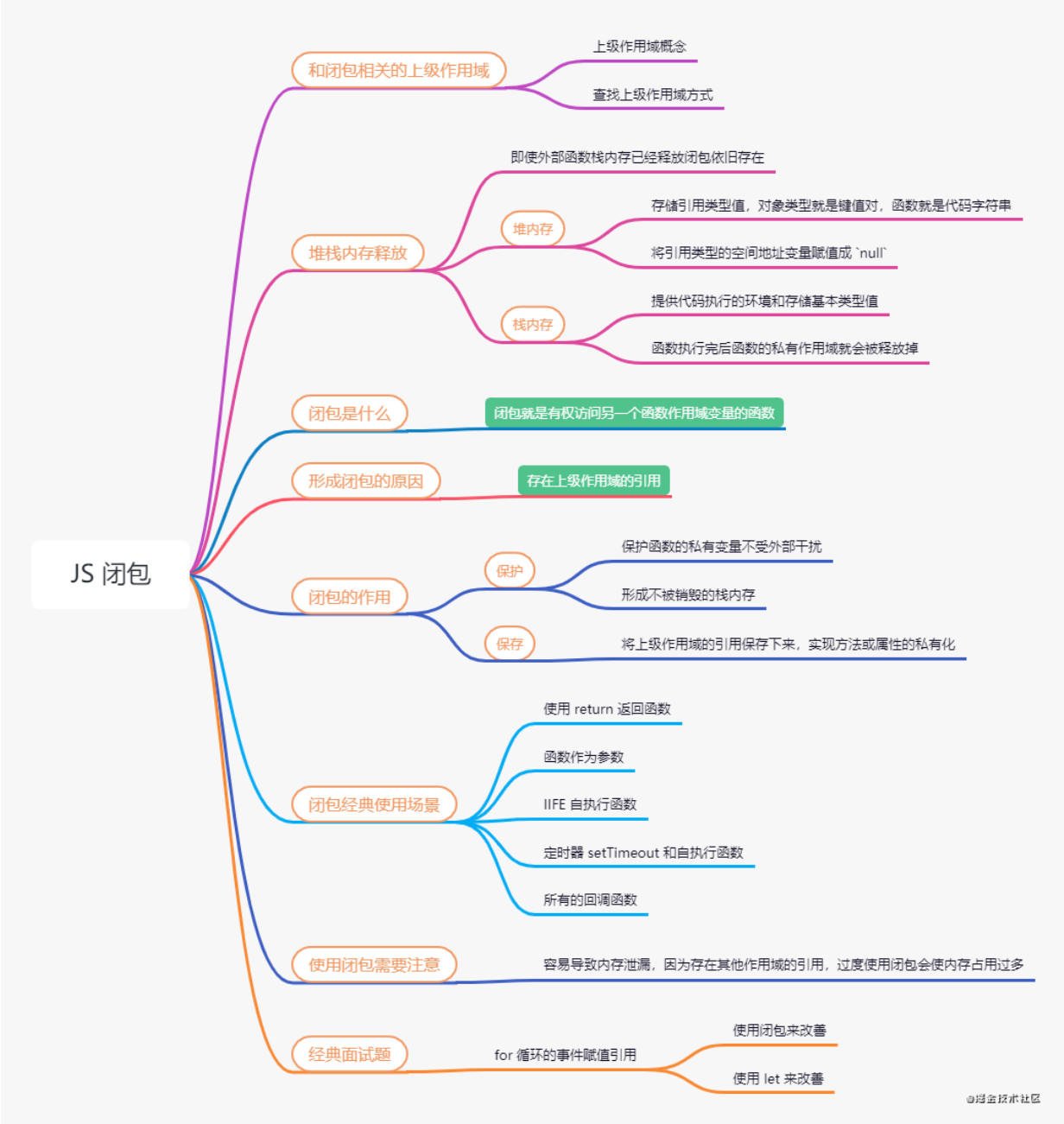
闭包 JS基础 编程题 (字节)

```
var foo = function(...args) {
    // 要求实现函数体
}
var f1 = foo(1,2,3);
f1.getValue(); // 6 输出是参数的和
var f2 = foo(1)(2,3);
f2.getValue(); // 6
var f3 = foo(1)(2)(3)(4);
f3.getValue(); // 10
```

解答@Ishmael-Yoko

```
function foo(...args) {
  const target = (...arg1s) => foo(...[...args, ...arg1s])
  target.getValue = () => args.reduce((p, n) => p+ n, 0)
  return target
}
```

闭包的使用场景，使用闭包需要注意什么



闭包

什么是闭包

闭包很简单，就是能够访问另一个函数作用域变量的函数，更简单的说，闭包就是函数，只不过是声明在其它函数内部而已。

例如：

```
function getOuter(){
  var count = 0
  function getCount(num){
    count += num
    console.log(count) //访问外部的date
  }
  return getCount //外部函数返回
}
var myfunc = getOuter()
myfunc(1) // 1
myfunc(2) // 3
```

`myfunc` 就是闭包， `myfunc` 是执行 `getOuter` 时创建的 `getCount` 函数实例的引用。 `getCount` 函数实例维护了一个对它的词法环境的引用，所以闭包就是函数+词法环境

当 `myfunc` 函数被调用时，变量 `count` 依然是可用的，也可以更新的

```
function add(x){
    return function(y){
        return x + y
    };
}

var addFun1 = add(4)
var addFun2 = add(9)

console.log(addFun1(2)) //6
console.log(addFun2(2))  //11
```

`add` 接受一个参数 `x` ，返回一个函数,它的参数是 `y` ，返回 `x+y`

`add` 是一个函数工厂，传入一个参数，就可以创建一个参数和其他参数求值的函数。

`addFun1` 和 `addFun2` 都是闭包。他们使用相同的函数定义，但词法环境不同， `addFun1` 中 `x` 是 `4` ，后者是 `5`

即：

- 闭包可以访问当前函数以外的变量
- 即使外部函数已经返回，闭包仍能访问外部函数定义的变量与参数
- 闭包可以更新外部变量的值

所以，闭包可以：

- 避免全局变量的污染
- 能够读取函数内部的变量
- 可以在内存中维护一个变量

使用闭包应该注意什么

- **代码难以维护：** 闭包内部是可以访问上级作用域，改变上级作用域的私有变量，我们使用的使用一定要小心，不要随便改变上级作用域私有变量的值
- **使用闭包的注意点：** 由于闭包会使得函数中的变量都保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄漏。解决方法是，在退出函数之前，将不使用的局部变量全部删除（引用设置为 `null` ，这样就解除了对这个变量的引用，其引用计数也会减少，从而确保其内存可以在适当的时机回收）
- **内存泄漏：** 程序的运行需要内存。对于持续运行的服务进程，必须及时释放不再用到的内存，否则占用越来越高，轻则影响系统性能，重则导致进程崩溃。不再用到的内存，没有及时释放，就叫做内存泄漏
- **this指向：** 闭包的this指向的是window

应用场景

闭包通常用来创建内部变量，使得这些变量不能被外部随意修改，同时又可以通过指定的函数接口来操作。例如 `setTimeout` 传参、回调、IIFE、函数防抖、节流、柯里化、模块化等等

`setTimeout` 传参

```
//原生的setTimeout传递的第一个函数不能带参数
setTimeout(function(param){
    alert(param)
},1000)


//通过闭包可以实现传参效果
function myfunc(param){
    return function(){
        alert(param)
    }
}
var f1 = myfunc(1);
setTimeout(f1,1000);
```

回调

大部分我们所写的 JavaScript 代码都是基于事件的 — 定义某种行为，然后将其添加到用户触发的事件之上（比如点击或者按键）。我们的代码通常作为回调：为响应事件而执行的函数。

例如，我们想在页面上添加一些可以调整字号的按钮。可以采用css，也可以使用：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>test</title>
    <link rel="stylesheet" href="">
```

```
</head>
<style>
    body{
        font-size: 12px;
    }
    h1{
        font-size: 1.5rem;
    }
    h2{
        font-size: 1.2rem;
    }
</style>
<body>

    <p>测试</p>

    <a href="#" id="size-12">12</a>
    <a href="#" id="size-14">14</a>
    <a href="#" id="size-16">16</a>

<script>
    function changeSize(size){
        return function(){
            document.body.style.fontSize = size + 'px';
        };
    }

    var size12 = changeSize(12);
    var size14 = changeSize(14);
    var size16 = changeSize(16);

    document.getElementById('size-12').onclick = size12;
    document.getElementById('size-14').onclick = size14;
    document.getElementById('size-16').onclick = size16;
</script>
</body>
</html>
```

IIFE

```
  var arr = [];
    for (var i=0;i<3;i++){
      //使用IIFE
      (function (i) {
        arr[i] = function () {
          return i;
        };
      })(i);
    }
    console.log(arr[0]()) // 0
    console.log(arr[1]()) // 1
    console.log(arr[2]()) // 2
```

函数防抖、节流

`debounce` 与 `throttle` 是开发中常用的高阶函数，作用都是为了防止函数被高频调用，换句话说就是，用来控制某个函数在一定时间内执行多少次。

使用场景

比如绑定响应鼠标移动、窗口大小调整、滚屏等事件时，绑定的函数触发的频率会很频繁。若稍处理函数微复杂，需要较多的运算执行时间和资源，往往会出现延迟，甚至导致假死或者卡顿感。为了优化性能，这时就很有必要使用 `debounce` 或 `throttle` 了。

debounce 与 throttle 区别

防抖 （**debounce**） ：多次触发，只在最后**一次触发**时，执行目标函数。

节流（**throttle**）：限制目标函数调用的**频率**，比如：1s内不能调用2次。

源码实现

debounce

```
// 这个是用来获取当前时间戳的
function now() {
  return +new Date()
}
/**
 * 防抖函数，返回函数连续调用时，空闲时间必须大于或等于 wait，func 才会执行
 *
 * @param  {function} func        回调函数
 * @param  {number}   wait        表示时间窗口的间隔
 * @param  {boolean}  immediate   设置为ture时，是否立即调用函数
 * @return {function}             返回客户调用函数
 */
function debounce (func, wait = 50, immediate = true) {
  let timer, context, args

  // 延迟执行函数
```

```
  const later = () => setTimeout(() => {
    // 延迟函数执行完毕，清空缓存的定时器序号
    timer = null
    // 延迟执行的情况下，函数会在延迟函数中执行
    // 使用到之前缓存的参数和上下文
    if (!immediate) {
      func.apply(context, args)
      context = args = null
    }
  }, wait)

  // 这里返回的函数是每次实际调用的函数
  return function(...params) {
    // 如果没有创建延迟执行函数（later），就创建一个
    if (!timer) {
      timer = later()
      // 如果是立即执行，调用函数
      // 否则缓存参数和调用上下文
      if (immediate) {
        func.apply(this, params)
      } else {
        context = this
        args = params
      }
    // 如果已有延迟执行函数（later），调用的时候清除原来的并重新设定一个
    // 这样做延迟函数会重新计时
    } else {
      clearTimeout(timer)
      timer = later()
    }
  }
}
```

throttle

```
/**
 * underscore 节流函数，返回函数连续调用时，func 执行频率限定为 次 / wait
 *
 * @param  {function}   func      回调函数
 * @param  {number}     wait      表示时间窗口的间隔
 * @param  {object}     options   如果想忽略开始函数的的调用，传入{leading:
false}。
 *                                如果想忽略结尾函数的调用，传入{trailing:
false}
 *                                两者不能共存，否则函数不能执行
 * @return {function}             返回客户调用函数
 */
_.throttle = function(func, wait, options) {
```

```
    var context, args, result;
    var timeout = null;
    // 之前的时间戳
    var previous = 0;
    // 如果 options 没传则设为空对象
    if (!options) options = {};
    // 定时器回调函数
    var later = function() {
      // 如果设置了 leading，就将 previous 设为 0
      // 用于下面函数的第一个 if 判断
      previous = options.leading === false ? 0 : _.now();
      // 置空一是为了防止内存泄漏，二是为了下面的定时器判断
      timeout = null;
      result = func.apply(context, args);
      if (!timeout) context = args = null;
    };
    return function() {
      // 获得当前时间戳
      var now = _.now();
      // 首次进入前者肯定为 true
      // 如果需要第一次不执行函数
      // 就将上次时间戳设为当前的
      // 这样在接下来计算 remaining 的值时会大于0
      if (!previous && options.leading === false) previous = now;
      // 计算剩余时间
      var remaining = wait - (now - previous);
      context = this;
      args = arguments;
      // 如果当前调用已经大于上次调用时间 + wait
      // 或者用户手动调了时间
      // 如果设置了 trailing，只会进入这个条件
      // 如果没有设置 leading，那么第一次会进入这个条件
      // 还有一点，你可能会觉得开启了定时器那么应该不会进入这个 if 条件了
      // 其实还是会进入的，因为定时器的延时
      // 并不是准确的时间，很可能你设置了2秒
      // 但是他需要2.2秒才触发，这时候就会进入这个条件
      if (remaining <= 0 || remaining > wait) {
        // 如果存在定时器就清理掉否则会调用二次回调
        if (timeout) {
          clearTimeout(timeout);
          timeout = null;
        }
        previous = now;
        result = func.apply(context, args);
        if (!timeout) context = args = null;
      } else if (!timeout && options.trailing !== false) {
        // 判断是否设置了定时器和 trailing
        // 没有的话就开启一个定时器
        // 并且不能不能同时设置 leading 和 trailing
```

```
        timeout = setTimeout(later, remaining);
      }
      return result;
    };
  };
```

柯里化

在计算机科学中，柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数(最初函数的第一个参数)的函数，并且返回接受余下的参数且返回结果的新函数的技术。这个技术由 Christopher Strachey 以逻辑学家 Haskell Curry 命名的，尽管它是 Moses Schnfinkel 和 Gottlob Frege 发明的。

```
var add = function(x) {
  return function(y) {
    return x + y;
  };
};
var increment = add(1);
var addTen = add(10);
increment(2);
// 3
addTen(2);
// 12
add(1)(2);
// 3
```

这里定义了一个 `add` 函数，它接受一个参数并返回一个新的函数。调用 `add` 之后，返回的函数就通过闭包的方式记住了 `add` 的第一个参数。所以说 `bind` 本身也是闭包的一种使用场景。

柯里化是将 `f(a,b,c)` 可以被以 `f(a)(b)(c)` 的形式被调用的转化。JavaScript 实现版本通常保留函数被正常调用和在参数数量不够的情况下返回偏函数这两个特性。

模块化

模块化的目的在于**将一个程序按照其功能做拆分，分成相互独立的模块，以便于每个模块只包含与其功能相关的内容，模块之间通过接口调用**。

模块化开发和闭包息息相关，通过模块模式需要具备两个必要条件可以看出：

- 外部必须是一个函数,且函数必须至少被调用一次(每次调用产生的闭包作为新的模块实例)
- 外部函数内部至少有一个内部函数, 内部函数用于修改和访问各种内部私有成员

```
function myModule (){
    const moduleName = '我的自定义模块'
    var name = 'sisterAn'

    // 在模块内定义方法(API)
```

```
    function getName(){
        console.log(name)
    }
    function modifyName(newName){
        name = newName
    }

    // 模块暴露：  向外暴露API
    return {
        getName,
        modifyName
    }
}

// 测试
const md = myModule()
md.getName()    // 'sisterAn'
md.modifyName('PZ')
md.getName()    // 'PZ'

// 模块实例之间互不影响
const md2 = myModule()
md2.sayHello = function () {
    console.log('hello')
}
console.log(md) // {getName: ƒ, modifyName: ƒ}
```

常见错误

在循环中创建闭包

```
var data = []

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

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

这里的 `i` 是全局下的 `i`，共用一个作用域，当函数被执行的时候这时的 `i=3`，导致输出的结构都是3

方案一：闭包

```
var data = []

function myfunc(num) {
  return function(){
    console.log(num)
  }
}

for (var i = 0; i < 3; i++) {
  data[i] = myfunc(i)
}

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

方案二：let

如果不想使用过多的闭包，你可以用 ES6 引入的 let 关键词：

```
var data = []

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

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

方案三：forEach

如果是数组的遍历操作（如下例中的 `arr` ），还有一个可选方案是使用 forEach()来遍历：

```
var data = []

var arr = [0, 1, 2]
arr.forEach(function (i) {
  data[i] = function () {
    console.log(i)
  }
})

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

原文

了解词法环境吗？它和闭包有什么联系？

词法环境（Lexical Environment）

官方定义

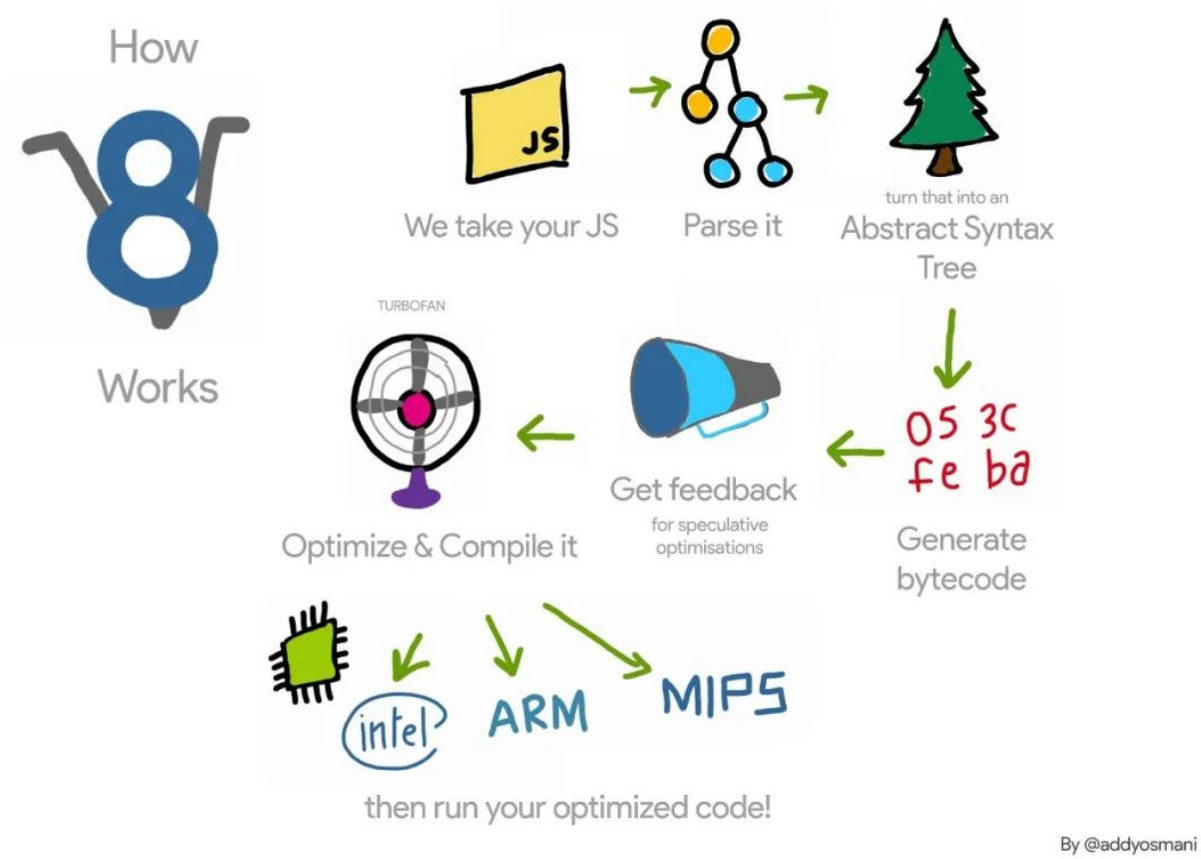
官方 ES2020 这样定义词法环境（Lexical Environment）：

> A Lexical Environment is a specification type used to define the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code. A Lexical Environment consists of an Environment Record and a possibly null reference to an *outer* Lexical Environment.

词法环境是一种规范类型（specification type），它基于 ECMAScript 代码的词法嵌套结构，来定义标识符与特定变量和函数的关联关系。词法环境由环境记录（environment record）和可能为空引用（null）的外部词法环境组成。

说的很详细，可是很难理解嘛🤔

下面，我们通过一个 V8 中 JS 的编译过程来更加直观的解释。



V8 中 JS 的编译过程来更加直观的解释

大致分为三个步骤：

- **第一步 词法分析**：V8 刚拿到执行上下文的时候，会把代码从上到下一行一行的进行分词/词法分析（Tokenizing/Lexing），例如 `var a = 1;`，会被分成 `var` 、 `a` 、 `1` 、 `;` 这样的

原子符号（(atomic token）。词法分析=指登记变量声明+函数声明+函数声明的形参。

- **第二步 语法分析**：在词法分析结束后，会做语法分析，引擎将 token 解析成一个抽象语法树（AST），在这一步会检测是否有语法错误，如果有则直接报错不再往下执行

```
var a = 1;
console.log(a);
a = ;
// Uncaught SyntaxError: Unexpected token ;
// 代码并没有打印出来 1 ，而是直接报错，说明在代码执行前进行了词法分析、语法分析
```

- **注意：** 词法分析跟语法分析不是完全独立的，而是交错运行的。也就是说，并不是等所有的 token 都生成之后，才用语法分析器来处理。一般都是每取得一个 token ，就开始用语法分析器来处理了
- **第三步 代码生成**：最后一步就是将 AST 转成计算机可以识别的机器指令码

在第一步中，我们看到有词法分析，它用来登记变量声明、函数声明以及函数声明的形参，后续代码执行的时候就可以知道要从哪里去获取变量值与函数。这个登记的地方就是词法环境。

词法环境包含两部分：

- **环境记录**：存储变量和函数声明的实际位置，真正用来登记变量的地方
- **对外部环境的引用**：意味着它可以访问其外部词法环境，是作用域链能够连接起来的关键

每个环境能访问到的标识符集合，我们称之为“作用域”。我们将作用域一层一层嵌套，形成了“作用域链”。

词法环境有两种 **类型**：

- **全局环境**：是一个没有外部环境的词法环境，其外部环境引用为 **null**。拥有一个全局对象（window 对象）及其关联的方法和属性（例如数组方法）以及任何用户自定义的全局变量，`this` 的值指向这个全局对象。
- **函数环境**：用户在函数中定义的变量被存储在**环境记录**中，包含了`arguments` 对象。对外部环境的引用可以是全局环境，也可以是包含内部函数的外部函数环境。

环境记录 同样有两种类型：

- **声明性环境记录**：存储变量、函数和参数。一个函数环境包含声明性环境记录。
- **对象环境记录**：用于定义在全局执行上下文中出现的变量和函数的关联。全局环境包含对象环境记录。

如果用伪代码的形式表示，词法环境是这样哒：

```
GlobalExectionContext = {  // 全局执行上下文
  LexicalEnvironment: {        // 词法环境
    EnvironmentRecord: {         // 环境记录
      Type: "Object",               // 全局环境
      // ...
      // 标识符绑定在这里
    },
    outer: <null>                // 对外部环境的引用
  }
```

```
}

FunctionExectionContext = { // 函数执行上下文
  LexicalEnvironment: {        // 词法环境
    EnvironmentRecord: {         // 环境记录
      Type: "Declarative",          // 函数环境
      // ...
      // 标识符绑定在这里              // 对外部环境的引用
    },
    outer: <Global or outer function environment reference>
  }
}
```

例如：

```
let a = 20;
const b = 30;
var c;

function multiply(e, f) {
 var g = 20;
 return e * f * g;
}

c = multiply(20, 30);
```

对应的执行上下文、词法环境：

```
GlobalExectionContext = {

  ThisBinding: <Global Object>,

  LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Object",
      // 标识符绑定在这里
      a: < uninitialized >,
      b: < uninitialized >,
      multiply: < func >
    }
    outer: <null>
  },

  VariableEnvironment: {
    EnvironmentRecord: {
      Type: "Object",
      // 标识符绑定在这里
```

```
      c: undefined,
    }
    outer: <null>
  }
}

FunctionExectionContext = {

  ThisBinding: <Global Object>,

  LexicalEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // 标识符绑定在这里
      Arguments: {0: 20, 1: 30, length: 2},
    },
    outer: <GlobalLexicalEnvironment>
  },

  VariableEnvironment: {
    EnvironmentRecord: {
      Type: "Declarative",
      // 标识符绑定在这里
      g: undefined
    },
    outer: <GlobalLexicalEnvironment>
  }
}
```

词法环境与我们自己写的代码结构相对应，也就是我们自己代码写成什么样子，词法环境就是什么样子。词法环境是在代码定义的时候决定的，跟代码在哪里调用没有关系。所以说 JS 采用的是词法作用域（静态作用域），即它在代码写好之后就被静态决定了它的作用域。

静态作用域 vs 动态作用域

动态作用域是基于栈结构，局部变量与函数参数都存储在栈中，所以，变量的值是由代码运行时当前栈的栈顶执行上下文决定的。而静态作用域是指变量创建时就决定了它的值，源代码的位置决定了变量的值。

```
var x = 1;

function foo() {
  var y = x + 1;
  return y;
}

function bar() {
  var x = 2;
```

```
  return foo();
}

foo(); // 静态作用域：2；动态作用域：2
bar(); // 静态作用域：2；动态作用域：3
```

在此例中，静态作用域与动态作用域的执行结构可能是不一致的，`bar` 本质上就是执行 `foo` 函数，如果是静态作用域的话，`bar` 函数中的变量 `x` 是在 `foo` 函数创建的时候就确定了，也就是说变量 `x` 一直为 `1`，两次输出应该都是 `2`。而动态作用域则根据运行时的 `x` 值而返回不同的结果。

所以说，动态作用域经常会带来不确定性，它不能确定变量的值到底是来自哪个作用域的。

大多数现在程序设计语言都是采用静态作用域规则，如C/C++、C#、Python、Java、JavaScript等，采用动态作用域的语言有Emacs Lisp、Common Lisp（兼有静态作用域）、Perl（兼有静态作用域）。C/C++的宏中用到的名字，也是动态作用域。

词法环境与闭包

> 一个函数和对其周围状态（**lexical environment，词法环境**）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是**闭包**（**closure**）
>
> ——MDN

也就是说，闭包是由 **函数** 以及声明该函数的 **词法环境** 组合而成的

```
var x = 1;

function foo() {
  var y = 2; // 自由变量
  function bar() {
    var z = 3; //自由变量
    return x + y + z;
  }
  return bar;
}

var test = foo();

test(); // 6
```

基于我们对词法环境的理解，上述例子可以抽象为如下伪代码：

```
GlobalEnvironment = {
  EnvironmentRecord: {
    // 内置标识符
    Array: '<func>',
    Object: '<func>',
```

```
    // 等等..

    // 自定义标识符
    x: 1
  },
  outer: null
};

fooEnvironment = {
  EnvironmentRecord: {
    y: 2,
    bar: '<func>'
  }
  outer: GlobalEnvironment
};

barEnvironment = {
  EnvironmentRecord: {
    z: 3
  }
  outer: fooEnvironment
};
```

前面说过，词法作用域也叫静态作用域，变量在词法阶段确定，也就是定义时确定。虽然在 `bar` 内调用，但由于 `foo` 是闭包函数，即使它在自己定义的词法作用域以外的地方执行，它也一直保持着自己的作用域。所谓闭包函数，即这个函数封闭了它自己的定义时的环境，形成了一个闭包，所以 `foo` 并不会从 `bar` 中寻找变量，这就是静态作用域的特点。

为了实现闭包，我们不能用动态作用域的动态堆栈来存储变量。如果是这样，当函数返回时，变量就必须出栈，而不再存在，这与最初闭包的定义是矛盾的。事实上，外部环境的闭包数据被存在了“堆”中，这样才使得即使函数返回之后内部的变量仍然一直存在（即使它的执行上下文也已经出栈）。

[原文]

从页面 A 打开一个新页面 B，B 页面关闭（包括意外崩溃），如何通知 A 页面?

本题是 html 页面通信题，可以拆分成：

- A 页面打开 B 页面，A、B 页面通信方式?
- B 页面正常关闭，如何通知 A 页面?
- B 页面意外崩溃，又该如何通知 A 页面?

A 页面打开 B 页面，A、B 页面通信方式

据我所知，A、B 页面通信方式有：

- url 传参
- postmessage
- localStorage
- WebSocket
- SharedWorker
- Service Worker

url 传参

url 传参数没什么可说的

```
<!-- A.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>A</title>
</head>
<body>
    <h1>A 页面</h1>
    <button type="button" onclick="openB()">B</button>
    <script>
        window.name = 'A'
        function openB() {
            window.open("B.html", "B")
        }

        window.addEventListener('hashchange', function () {// 监听 hash
            alert(window.location.hash)
        }, false);
    </script>
</body>
</html>
```

B:

```
<!-- B.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>B</title>
    <button type="button" onclick="sendA()">发送A页面消息</button>
</head>
<body>
    <h1>B 页面</h1>
    <span></span>
    <script>
        window.name = 'B'
        window.onbeforeunload = function (e) {
            window.open('A.html#close', "A")
            return '确定离开此页吗? ';
        }
    </script>
</body>
</html>
```

A 页面通过 url 传递参数与 B 页面通信，同样通过监听 **hashchange** 事件，在页面 B 关闭时与 A 通信

postmessage

`postMessage` 是 `h5` 引入的 API，`postMessage()` 方法允许来自不同源的脚本采用异步方式进行有效的通信，可以实现跨文本文档、多窗口、跨域消息传递，可在多用于窗口间数据通信，这也使它成为跨域通信的一种有效的解决方案，简直不要太好用

A 页面打开 B 页面，B 页面向 A 页面发送消息：

```
<!-- A.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>A</title>
</head>
<body>
    <h1>A 页面</h1>
    <button type="button" onclick="openB()">B</button>
    <script>
```

```
        window.name = 'A'
        function openB() {
            window.open("B.html?code=123", "B")
        }
        window.addEventListener("message", receiveMessage, false);
        function receiveMessage(event) {
            console.log('收到消息：', event.data)
        }
    </script>
</body>
</html>
```

```
<!-- B.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>B</title>
    <button type="button" onclick="sendA()">发送A页面消息</button>
</head>
<body>
    <h1>B 页面</h1>
    <span></span>
    <script>
        window.name = 'B'
        function sendA() {
            let targetWindow = window.opener
            targetWindow.postMessage('Hello A', "http://localhost:3000");
        }
    </script>
</body>
</html>
```

```
收到消息：  Hello A                                    A.html:18
>
```

localStorage

```
// A
localStorage.setItem('testB', 'sisterAn');

// B
let testB = localStorage.getItem('testB');
console.log(testB)
// sisterAn
```

注意： `localStorage` 仅允许你访问一个 `Document` 源（origin）的对象 `Storage`；存储的数据将保存在浏览器会话中。如果 A 打开的 B 页面和 A 是不同源，则无法访问同一 `Storage`

WebSocket

基于服务端的页面通信方式，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于服务器推送技术的一种

SharedWorker

`SharedWorker` 接口代表一种特定类型的 worker，可以从几个浏览上下文中*访问*，例如几个窗口、iframe 或其他 worker。它们实现一个不同于普通 worker 的接口，具有不同的全局作用域，`SharedWorkerGlobalScope` 。

```
// A.html
var sharedworker = new SharedWorker('worker.js')
sharedworker.port.start()
sharedworker.port.onmessage = evt => {
    // evt.data
    console.log(evt.data) // hello A
}

// B.html
var sharedworker = new SharedWorker('worker.js')
sharedworker.port.start()
sharedworker.port.postMessage('hello A')

// worker.js
const ports = []
onconnect = e => {
const port = e.ports[0]
   ports.push(port)
   port.onmessage = evt => {
       ports.filter(v => v!== port) // 此处为了贴近其他方案的实现，剔除自己
       .forEach(p => p.postMessage(evt.data))
   }
}
```

Service Worker

Service Worker 是一个可以长期运行在后台的 Worker，能够实现与页面的双向通信。多页面共享间的 Service Worker 可以共享，将 Service Worker 作为消息的处理中心（中央站）即可实现广播效果。

```
// 注册
navigator.serviceWorker.register('./sw.js').then(function () {
    console.log('Service Worker 注册成功');
})

// A
navigator.serviceWorker.addEventListener('message', function (e) {
    console.log(e.data)
});

// B
navigator.serviceWorker.controller.postMessage('Hello A');
```

B 页面正常关闭，如何通知 A 页面

页面正常关闭时，会先执行 `window.onbeforeunload` ，然后执行 `window.onunload` ，我们可以在这两个方法里向 A 页面通信

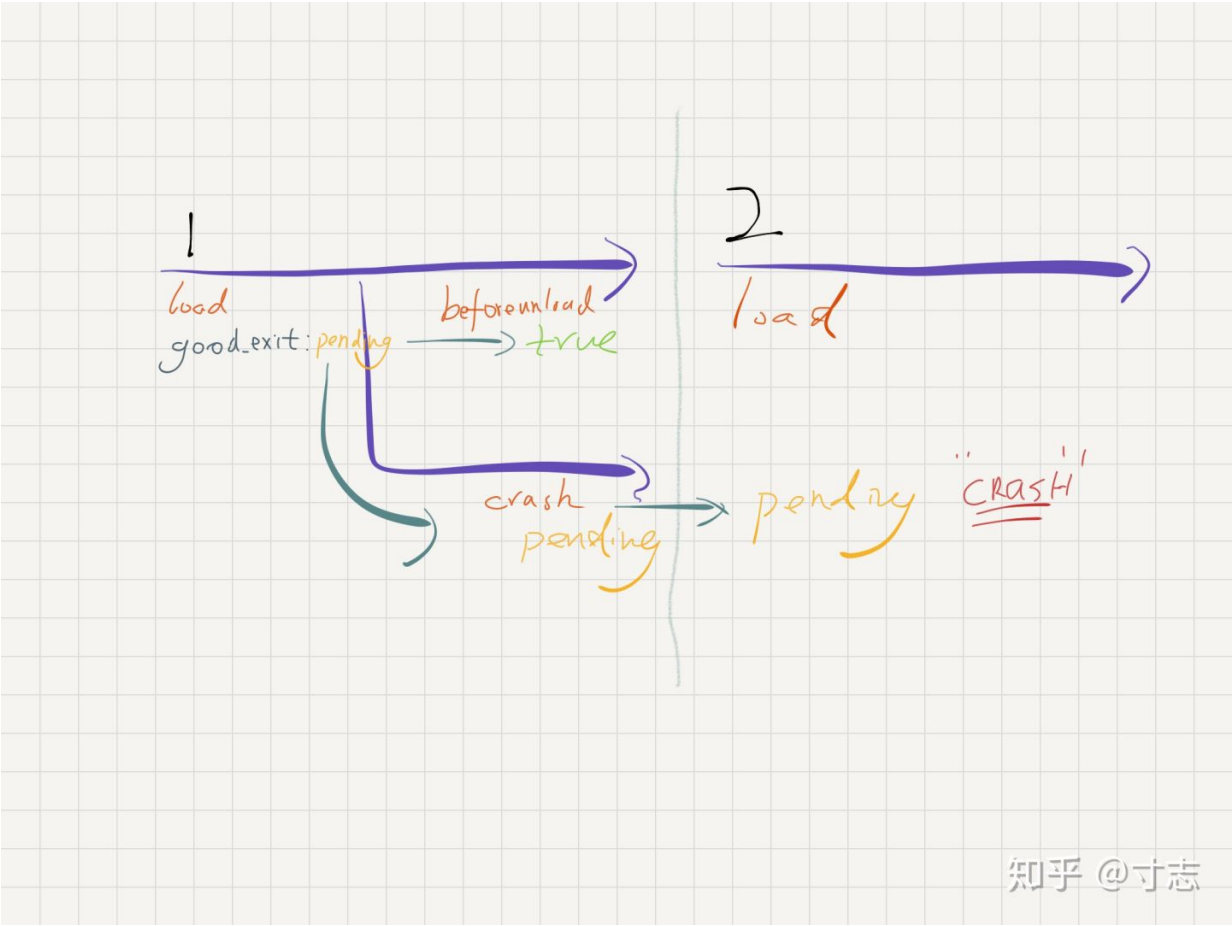
B 页面意外崩溃，又该如何通知 A 页面

页面正常关闭，我们有相关的 API，崩溃就不一样了，页面看不见了，JS 都不运行了，那还有什么办法可以获取B页面的崩溃?

全网搜索了一下，发现我们可以利用 window 对象的 `load` 和 `beforeunload` 事件，通过心跳监控来获取 B 页面的崩溃

```
 window.addEventListener('load', function () {
      sessionStorage.setItem('good_exit', 'pending');
      setInterval(function () {
         sessionStorage.setItem('time_before_crash', new
Date().toString());
      }, 1000);
   });

   window.addEventListener('beforeunload', function () {
      sessionStorage.setItem('good_exit', 'true');
   });

   if(sessionStorage.getItem('good_exit') &&
      sessionStorage.getItem('good_exit') !== 'true') {
      /*
         insert crash logging code here
     */
      alert('Hey, welcome back from your crash, looks like you crashed on:
' + sessionStorage.getItem('time_before_crash'));
   }
```

使用 load 和 beforeunload 事件实现崩溃监控过程如下：



图片来自：https://zhuanlan.zhihu.com/p/40273861

这个方案巧妙的利用了页面崩溃无法触发 **beforeunload** 事件来实现的。

在页面加载时（**load** 事件）在 **sessionStorage** 记录 **good_exit** 状态为 **pending**，如果用户正常退出（**beforeunload** 事件）状态改为 **true**，如果 **crash** 了，状态依然为 **pending**，在用户第2次访问网页的时候（第2个**load**事件），查看 **good_exit** 的状态，如果仍然是 **pending** 就是可以断定上次访问网页崩溃了！

但有一个问题，本例中用 sessionStorage 保存状态，在用户关闭了B页面，sessionStorage 值就会丢失，所以换种方式，使用 Service Worker 来实现：

- Service Worker 有自己独立的工作线程，与网页区分开，网页崩溃了，Service Worker 一般情况下不会崩溃；
- Service Worker 生命周期一般要比网页还要长，可以用来监控网页的状态；
- 网页可以通过 **navigator.serviceWorker.controller.postMessage** API 向掌管自己的 SW 发送消息

基于以上几点优势，完整设计一套流程如下：

- B 页面加载后，通过 **postMessage** API 每 **5s** 给 sw 发送一个心跳，表示自己的在线，sw 将在线的网页登记下来，更新登记时间；
- B 页面在 **beforeunload** 时，通过 **postMessage** API 告知自己已经正常关闭，sw 将登记的网页清除；
- 如果 B页面在运行的过程中 crash 了，sw 中的 **running** 状态将不会被清除，更新时间停留在奔溃前的最后一次心跳；

- A 页面 Service Worker 每 **10s** 查看一遍登记中的网页，发现登记时间已经超出了一定时间（比如 15s）即可判定该网页 crash 了。

代码如下：

```
// B
if (navigator.serviceWorker.controller !== null) {
  let HEARTBEAT_INTERVAL = 5 * 1000 // 每五秒发一次心跳
  let sessionId = uuid() // B页面会话的唯一 id
  let heartbeat = function () {
    navigator.serviceWorker.controller.postMessage({
      type: 'heartbeat',
      id: sessionId,
      data: {} // 附加信息，如果页面 crash，上报的附加数据
    })
  }
  window.addEventListener("beforeunload", function() {
    navigator.serviceWorker.controller.postMessage({
      type: 'unload',
      id: sessionId
    })
  })
  setInterval(heartbeat, HEARTBEAT_INTERVAL);
  heartbeat();
}
```

```
// 每 10s 检查一次，超过15s没有心跳则认为已经 crash
const CHECK_CRASH_INTERVAL = 10 * 1000
const CRASH_THRESHOLD = 15 * 1000
const pages = {}
let timer
function checkCrash() {
  const now = Date.now()
  for (var id in pages) {
    let page = pages[id]
    if ((now - page.t) > CRASH_THRESHOLD) {
      // 上报 crash
      delete pages[id]
    }
  }
  if (Object.keys(pages).length == 0) {
    clearInterval(timer)
    timer = null
  }
}
```

```
worker.addEventListener('message', (e) => {
  const data = e.data;
  if (data.type === 'heartbeat') {
    pages[data.id] = {
      t: Date.now()
    }
    if (!timer) {
      timer = setInterval(function () {
        checkCrash()
      }, CHECK_CRASH_INTERVAL)
    }
  } else if (data.type === 'unload') {
    delete pages[data.id]
  }
})
```

参考：

- 如何监控网页崩溃?
- 腾讯面试四问，Are you OK?

原文

监听一个变量的变化，需要怎么做

监听一个变量的变化，当变量变化时执行某些操作，这类似现在流行的前端框架（例如 React、Vue 等）中的数据绑定功能，在数据更新时自动更新 DOM 渲染，那么如何实现数据绑定喃?

本文给出两种思路：

- ES5 的 Object.defineProperty
- ES6 的 Proxy

ES5 的 Object.defineProperty

> `Object.defineProperty()` 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象
>
> ——MDN

```
Object.defineProperty(obj, prop, descriptor)
```

其中：

- `obj`： 要定义属性的对象
- `prop`：要定义或修改的属性的名称或 `Symbol`
- `descriptor`：要定义或修改的属性描述符

```
var user = {
    name: 'sisterAn'
}

Object.defineProperty(user, 'name', {
    enumerable: true,
    configurable:true,
    set: function(newVal) {
        this._name = newVal
        console.log('set: ' + this._name)
    },
    get: function() {
        console.log('get: ' + this._name)
        return this._name
    }
})

user.name = 'an' // set: an
console.log(user.name) // get: an
```

如果是完整的对变量的每一个子属性进行监听：

```
// 监视对象
function observe(obj) {
   // 遍历对象，使用 get/set 重新定义对象的每个属性值
    Object.keys(obj).map(key => {
        defineReactive(obj, key, obj[key])
    })
}

function defineReactive(obj, k, v) {
    // 递归子属性
    if (typeof(v) === 'object') observe(v)

    // 重定义 get/set
    Object.defineProperty(obj, k, {
        enumerable: true,
        configurable: true,
        get: function reactiveGetter() {
            console.log('get: ' + v)
            return v
        },
        // 重新设置值时，触发收集器的通知机制
        set: function reactiveSetter(newV) {
            console.log('set: ' + newV)
            v = newV
        },
    })
}

let data = {a: 1}
// 监视对象
observe(data)
data.a // get: 1
data.a = 2 // set: 2
```

通过 `map` 遍历，通过深度递归监听子子属性

注意， `Object.defineProperty` 拥有以下缺陷：

- IE8 及更低版本 IE 是不支持的
- 无法检测到对象属性的新增或删除
- 如果修改数组的 `length` （ `Object.defineProperty` 不能监听数组的长度），以及数组的 `push` 等变异方法是无法触发 `setter` 的

对此，我们看一下 vue2.x 是如何解决这块的?

vue2.x 中如何监测数组变化

使用了函数劫持的方式，重写了数组的方法，Vue 将 data 中的数组进行了原型链重写，指向了自己定义的数组原型方法。这样当调用数组 api 时，可以通知依赖更新。如果数组中包含着引用类型，会对数组中的引用类型再次递归遍历进行监控。这样就实现了监测数组变化。

对于数组而言，Vue 内部重写了以下函数实现派发更新

```
// 获得数组原型
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// 重写以下函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 缓存原生函数
  const original = arrayProto[method]
  // 重写函数
  def(arrayMethods, method, function mutator (...args) {
  // 先调用原生函数获得结果
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    // 调用以下几个函数时，监听新数据
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // 手动派发更新
    ob.dep.notify()
    return result
  })
})
```

vue2.x 怎么解决给对象新增属性不会触发组件重新渲染的问题

受现代 JavaScript 的限制 (`Object.observe` 已被废弃)，Vue 无法检测到对象属性的添加或删除。

由于 Vue 会在初始化实例时对属性执行 `getter/setter` 转化，所以属性必须在 `data` 对象上存在才能让 `Vue` 将它转换为响应式的。

对于已经创建的实例，Vue 不允许动态添加根级别的响应式属性。但是，可以使用 Vue.set(object, propertyName, value) 方法向嵌套对象添加响应式属性。

vm.$set()实现原理

```
export function set(target: Array<any> | Object, key: any, val: any): any {
  // target 为数组
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    // 修改数组的长度, 避免索引>数组长度导致 splice() 执行有误
    target.length = Math.max(target.length, key);
    // 利用数组的 splice 方法触发响应式
    target.splice(key, 1, val);
    return val;
  }
  // target 为对象, key 在 target 或者 target.prototype 上 且必须不能在
Object.prototype 上,直接赋值
  if (key in target && !(key in Object.prototype)) {
    target[key] = val;
    return val;
  }
  // 以上都不成立, 即开始给 target 创建一个全新的属性
  // 获取 Observer 实例
  const ob = (target: any).__ob__;
  // target 本身就不是响应式数据, 直接赋值
  if (!ob) {
    target[key] = val;
    return val;
  }
  // 进行响应式处理
  defineReactive(ob.value, key, val);
  ob.dep.notify();
  return val;
}
```

- 如果目标是数组,使用 vue 实现的变异方法 `splice` 实现响应式
- 如果目标是对象,判断属性存在,即为响应式,直接赋值
- 如果 `target` 本身就不是响应式,直接赋值
- 如果属性不是响应式,则调用 `defineReactive` 方法进行响应式处理

ES6 的 Proxy

众所周知，尤大大的 vue3.0 版本用 `Proxy` 代替了 `defineProperty` 来实现数据绑定，因为 `Proxy` 可以直接监听对象和数组的变化，并且有多达 `13` 种拦截方法。并且作为新标准将受到浏览器厂商重点持续的性能优化。

Proxy

> **Proxy** 对象用于创建一个对象的代理，从而实现基本操作的拦截和自定义（如属性查找、赋值、枚举、函数调用等）
>
> — MDN

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

其中：

- `target`：要使用 `Proxy` 包装的目标对象（可以是任何类型的对象，包括原生数组，函数，甚至另一个代理）
- `handler`：一个通常以函数作为属性的对象，各属性中的函数分别定义了在执行各种操作时代理 `p` 的行为

```
var handler = {
    get: function(target, name){
        return name in target ? target[name] : 'no prop!'
    },
    set: function(target, prop, value, receiver) {
        target[prop] = value;
        console.log('property set: ' + prop + ' = ' + value);
        return true;
    }
};

var user = new Proxy({}, handler)
user.name = 'an' // property set: name = an

console.log(user.name) // an
console.log(user.age) // no prop!
```

上面提到过 `Proxy` 总共提供了 `13` 种拦截行为，分别是：

- `getPrototypeOf` / `setPrototypeOf`
- `isExtensible` / `preventExtensions`
- `ownKeys` / `getOwnPropertyDescriptor`
- `defineProperty` / `deleteProperty`
- `get` / `set` / `has`
- `apply` / `construct`

感兴趣的可以查看 MDN，一一尝试一下，这里不再赘述

另外考虑两个问题：

- Proxy只会代理对象的第一层，那么又是怎样处理这个问题的呢?
- 监测数组的时候可能触发多次get/set，那么如何防止触发多次呢（因为获取push和修改length的时候也会触发）

Vue3 Proxy

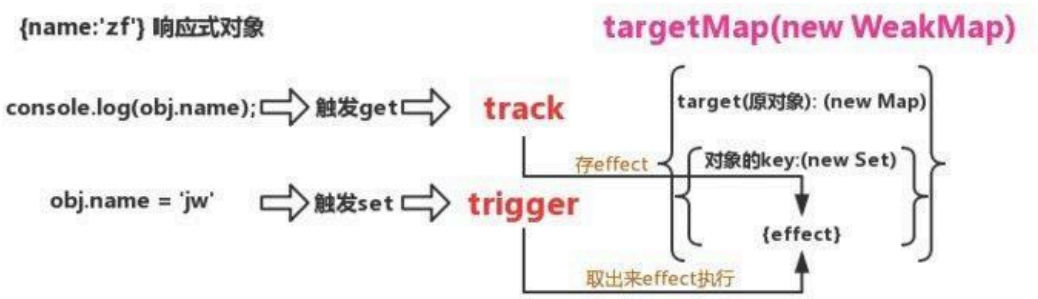
对于第一个问题，我们可以判断当前 `Reflect.get` 的返回值是否为 `Object` ，如果是则再通过 `reactive` 方法做代理， 这样就实现了深度观测。

对于第二个问题，我们可以判断是否是 `hasOwProperty`

下面我们自己写个案例，通过proxy 自定义获取、增加、删除等行为

```
const toProxy = new WeakMap(); // 存放被代理过的对象
const toRaw = new WeakMap(); // 存放已经代理过的对象
function reactive(target) {
  // 创建响应式对象
  return createReactiveObject(target);
}
function isObject(target) {
  return typeof target === "object" && target !== null;
}
function hasOwn(target,key){
  return target.hasOwnProperty(key);
}
function createReactiveObject(target) {
  if (!isObject(target)) {
    return target;
  }
  let observed = toProxy.get(target);
  if(observed){ // 判断是否被代理过
    return observed;
  }
  if(toRaw.has(target)){ // 判断是否要重复代理
    return target;
  }
  const handlers = {
    get(target, key, receiver) {
        let res = Reflect.get(target, key, receiver);
        track(target,'get',key); // 依赖收集==
        return isObject(res)
        ?reactive(res):res;
    },
    set(target, key, value, receiver) {
        let oldValue = target[key];
        let hadKey = hasOwn(target,key);
        let result = Reflect.set(target, key, value, receiver);
        if(!hadKey){
          trigger(target,'add',key); // 触发添加
        }else if(oldValue !== value){
          trigger(target,'set',key); // 触发修改
        }
        return result;
```

```
    },
    deleteProperty(target, key) {
      console.log("删除");
      const result = Reflect.deleteProperty(target, key);
      return result;
    }
  };

  // 开始代理
  observed = new Proxy(target, handlers);
  toProxy.set(target,observed);
  toRaw.set(observed,target); // 做映射表
  return observed;
}
```



总结

Proxy 相比于 defineProperty 的优势：

- 基于 `Proxy` 和 `Reflect` ，可以原生监听数组，可以监听对象属性的添加和删除
- 不需要深度遍历监听：判断当前 `Reflect.get` 的返回值是否为 `Object` ，如果是则再通过 `reactive` 方法做代理， 这样就实现了深度观测
- 只在 `getter` 时才对对象的下一层进行劫持(优化了性能)

所以，建议使用 `Proxy` 监测变量变化

原文

讲下 V8 sort 的大概思路，并手写一个 sort 的实现

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

以上是常见的几种排序算法，首先思考一下，`Array.prototype.sort()` 使用了上面的那种算法嘛?

Array.prototype.sort()

> `sort()` 方法用原地算法对数组的元素进行排序，并返回数组。默认排序顺序是在将元素转换为字符串，然后比较它们的UTF-16代码单元值序列时构建的
>
> — MDN

```
const array = [1, 30, 4, 21, 100000];
array.sort();
console.log(array);
// [1, 100000, 21, 30, 4]

const numbers = [4, 2, 5, 1, 3];
numbers.sort((a, b) => a - b);
console.log(numbers)
// [1, 2, 3, 4, 5]
```

V8 种的 Array.prototype.sort()

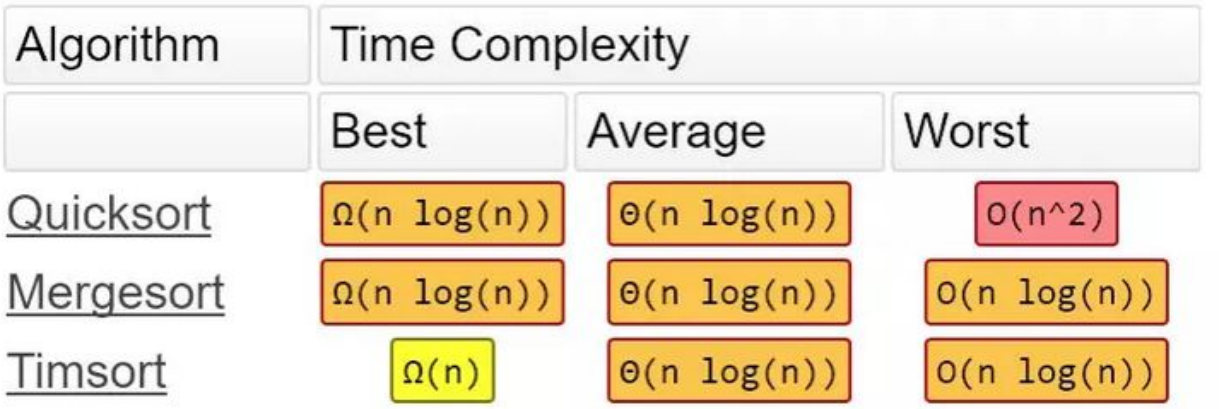
关于 `Array.prototype.sort()` ，ES 规范并没有指定具体的算法，在 V8 引擎中， **7.0 版本之前** ，数组长度小于10时， `Array.prototype.sort()` 使用的是插入排序，否则用快速排序。

在 V8 引擎 **7.0 版本之后** 就舍弃了快速排序，因为它不是稳定的排序算法，在最坏情况下，时间复杂度会降级到 O(n^2)。

于是采用了一种混合排序的算法：**TimSort** 。

这种功能算法最初用于Python语言中，严格地说它不属于以上10种排序算法中的任何一种，属于一种混合排序算法：

在数据量小的子数组中使用**插入排序**，然后再使用**归并排序**将有序的子数组进行合并排序，时间复杂度为 `O(nlogn)` 。



什么是 TimSort？

在 解答 v8 sort 源码前，我们先看看 TimSort 具体是如何实现的，帮助我们阅读源码

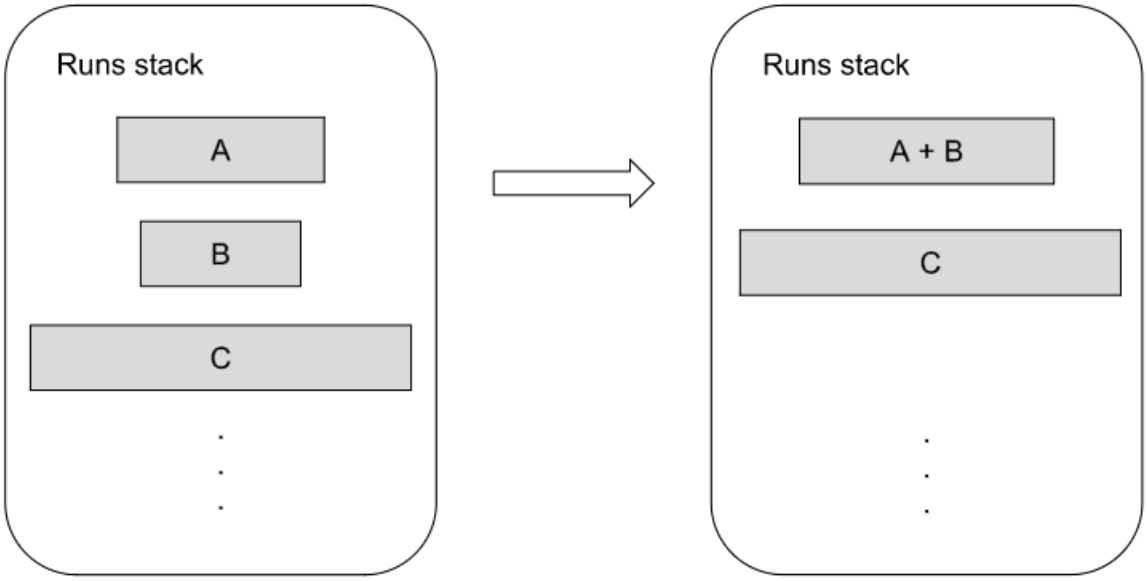
Timsort 是 Tim Peter 在 2001 年为 Python 语言特意创造的，主要是 **基于现实数据集中存在者大量的有序元素（不需要重新排序）** 。 Timsort 会遍历所有数据，找出数据中所有有序的分区（run），然后按照一定的规则将这些分区（run）归并为一个。

具体过程为：

- 扫描数组，并寻找所谓的 `_runs_` ，一个 run 可以认为是已经排序的小数组，也包括以逆向排序的，因为这些数组可以简单地翻转（reverse）就成为一个run
- 确定最小 run 长度，小于的 run 会通过 **插入排序** 归并成长度高于最小长度的 run
- 反复归并一些相邻 run ，过程中避免归并长度相差很大的片段，直至整个排序完成

如何避免归并长度相差很大 run 呢？在 Timsort 排序过程中，会存在一个栈用于记录每个 run 的起始索引位置与长度， 依次将 run 压入栈中，若栈顶 A 、B、C 的长度

- `|C| > |B| + |A|`
- `|B| > |A|`



在上图的例子中，因为 `| A |> | B |` ，所以B被合并到了它前后两个runs（A、C）中较小的一个 `| A |` ，然后 `| A |` 再与 `| C |` 。依据这个法则，能够尽量使得大小相同的 run 合并，以提高性能。注意Timsort是稳定排序故只有相邻的 run 才能归并。

所以，对于已经排序好的数组，会以 O(n) 的时间内完成排序，因为这样的数组将只产生单个 run ，不需要合并操作。最坏的情况是 O(n log n) 。这样的算法性能参数，以及 Timsort 天生的稳定性是我们最终选择 Timsort 而非 Quicksort 的几个原因。

手写一个 Array.prototype.sort() 实现

了解的 Timsort 的基本思想与排序过程后，我们手写一个简易版的 Timsort：

```
// 顺序合并两个小数组left、right 到 result
function merge(left, right) {
  let result = [],
      ileft = 0,
      iright = 0
  while(ileft < left.length && iright < right.length) {
    if(left[ileft] < right[iright]){
      result.push(left[ileft ++])
    } else {
      result.push(right[iright ++])
    }
  }
  while(ileft < left.length) {
    result.push(left[ileft ++])
  }
  while(iright < right.length) {
    result.push(right[iright ++])
  }
  return result
}
```

```
// 插入排序
function insertionSort(arr) {
    let n = arr.length;
    let preIndex, current;
    for (let i = 1; i < n; i++) {
        preIndex = i - 1;
        current = arr[i];
        while (preIndex >= 0 && arr[preIndex] > current) {
            arr[preIndex + 1] = arr[preIndex];
            preIndex--;
        }
        arr[preIndex + 1] = current;
    }
    return arr;
}

// timsort
function timsort(arr) {
    // 空数组或数组长度小于 2，直接返回
    if(!arr || arr.length < 2) return arr
    let runs = [],
        sortedRuns = [],
        newRun = [arr[0]],
        length = arr.length
    // 划分 run 区，并存储到 runs 中，这里简单的按照升序划分，没有考虑降序的run
    for(let i = 1; i < length; i++) {
        if(arr[i] < arr[i - 1]) {
            runs.push(newRun)
            newRun = [arr[i]]
        } else {
            newRun.push(arr[i])
        }
        if(length - 1 === i) {
            runs.push(newRun)
            break
        }
    }
    // 由于仅仅是升序的run，没有涉及到run的扩充和降序的run，因此，其实这里没有必要使用
insertionSort 来进行 run 自身的排序
    for(let run of runs) {
        insertionSort(run)
    }
    // 合并 runs
    sortedRuns = []
    for(let run of runs) {
        sortedRuns = merge(sortedRuns, run)
    }
    return sortedRuns
```

```
}

// 测试
var numbers = [4, 2, 5, 1, 3]
timsort(numbers)
// [1, 2, 3, 4, 5]
```

简易版的，完整的实现可以查看 v8 array-sort 实现，下面我们就来看一下

v8 中的 Array.prototype.sort() 源码解读

即 TimSort 在 v8 中的实现，具体实现步骤如下：

1. 判断数组长度，小于2直接返回，不排序
2. 开始循环
3. 找出一个有序子数组，我们称之为 “run”，长度 currentRunLength
4. 计算最小合并序列长度 minRunLength （这个值会根据数组长度动态变化，在32~64之间）
5. 比较 currentRunLength 和 minRunLength，如果 currentRunLength >= minRunLength，否则采用插入排序补足数组长度至 minRunLength，将 run 压入栈 pendingRuns 中
6. 每次有新的 run 被压入 pendingRuns 时保证栈内任意 3 个连续的 run（run0, run1, run2）从下至上满足run0 > run1 + run2 && run1 > run2，不满足的话进行调整直至满足
7. 如果剩余子数组为 0，结束循环
8. 合并栈中所有 run，排序结束

核心源码解读

下面重点解读 3 个核心函数：

- `ComputeMinRunLength`：用来计算 `minRunLength`
- `CountAndMakeRun`：计算第一个 `run` 的长度
- `MergeCollapse`：调整 `pendingRuns`，使栈长度大于 `3` 时，所有 `run` 都满足 `run[n]>run[n+1]+run[n+2]` 且 `run[n+1]>run2[n+2]`

```
// 计算最小合并序列长度 minRunLength
macro ComputeMinRunLength(nArg: Smi): Smi {
  let n: Smi = nArg;
  let r: Smi = 0;  // Becomes 1 if any 1 bits are shifted off.

  assert(n >= 0);
  // 如果小于 64 ，则返回n（该值太小，无法打扰那些奇特的东西）
  // 否则不断除以2，得到结果在 32~64 之间
  while (n >= 64) {
    r = r | (n & 1);
    n = n >> 1;
  }

  const minRunLength: Smi = n + r;
```

```
  assert(nArg < 64 || (32 <= minRunLength && minRunLength <= 64));
  return minRunLength;
}
```

```
// 计算第一个 run 的长度
macro CountAndMakeRun(implicit context: Context, sortState: SortState)(
    lowArg: Smi, high: Smi): Smi {
  assert(lowArg < high);
  // 这里保存的才是我们传入的数组数据
  const workArray = sortState.workArray;

  const low: Smi = lowArg + 1;
  if (low == high) return 1;

  let runLength: Smi = 2;

  const elementLow = UnsafeCast<JSAny>(workArray.objects[low]);
  const elementLowPred = UnsafeCast<JSAny>(workArray.objects[low - 1]);
  // 调用比对函数来比对数据
  let order = sortState.Compare(elementLow, elementLowPred);

  // TODO(szuend): Replace with "order < 0" once Torque supports it.
  //               Currently the operator<(Number, Number) has return type
  //               'never' and uses two labels to branch.
  const isDescending: bool = order < 0 ? true : false;

  let previousElement: JSAny = elementLow;
  // 遍历子数组并计算 run 的长度
  for (let idx: Smi = low + 1; idx < high; ++idx) {
    const currentElement = UnsafeCast<JSAny>(workArray.objects[idx]);
    order = sortState.Compare(currentElement, previousElement);

    if (isDescending) {
      if (order >= 0) break;
    } else {
      if (order < 0) break;
    }

    previousElement = currentElement;
    ++runLength;
  }

  if (isDescending) {
    ReverseRange(workArray, lowArg, lowArg + runLength);
  }
```

```
  return runLength;
}
```

```
// 调整 pendingRuns ，使栈长度大于3时，所有 run 都满足 run[n]>run[n+1]+run[n+2]
且 run[n+1]>run2[n+2]
transitioning macro MergeCollapse(context: Context, sortState: SortState) {
  const pendingRuns: FixedArray = sortState.pendingRuns;

  // Reload the stack size because MergeAt might change it.
  while (GetPendingRunsSize(sortState) > 1) {
    let n: Smi = GetPendingRunsSize(sortState) - 2;

    if (!RunInvariantEstablished(pendingRuns, n + 1) ||
        !RunInvariantEstablished(pendingRuns, n)) {
      if (GetPendingRunLength(pendingRuns, n - 1) <
          GetPendingRunLength(pendingRuns, n + 1)) {
        --n;
      }

      MergeAt(n); // 将第 n 个 run 和第 n+1 个 run 进行合并
    } else if (
        GetPendingRunLength(pendingRuns, n) <=
        GetPendingRunLength(pendingRuns, n + 1)) {
      MergeAt(n); // 将第 n 个 run 和第 n+1 个 run 进行合并
    } else {
      break;
    }
  }
}
```

原文

了解继承吗？该如何实现继承？

引言

JS 继承主要由六种实现方式：

- 原型链继承
- 构造函数继承
- 组合继承
- 寄生组合继承
- 原型式继承
- ES6 继承

ES5 继承

先定义一个父类

```
function SuperType () {
  // 属性
  this.name = 'SuperType';
}
// 原型方法
SuperType.prototype.sayName = function() {
  return this.name;
};
```

一、 原型链继承

将父类的实例作为子类的原型

```
// 父类
function SuperType () {
  this.name = 'SuperType'; // 父类属性
}
SuperType.prototype.sayName = function () { // 父类原型方法
  return this.name;
};

// 子类
function SubType () {
  this.subName = "SubType"; // 子类属性
};

SubType.prototype = new SuperType(); // 重写原型对象，代之以一个新类型的实例
```

```
// 这里实例化一个 SuperType 时， 实际上执行了两步
// 1，新创建的对象复制了父类构造函数内的所有属性及方法
// 2，并将原型 __proto__ 指向了父类的原型对象

SubType.prototype.saySubName = function () { // 子类原型方法
  return this.subName;
}

// 子类实例
let instance = new SubType();

// instanceof 通过判断对象的 prototype 链来确定对象是否是某个类的实例
instance instanceof SubType; // true
instance instanceof SuperType; // true

// 注意
SubType instanceof SuperType; // false
SubType.prototype instanceof SuperType ; // true
```

```
> instance
< ▼SubType {subName: "SubType"}
    subName: "SubType"
  ▼__proto__: SuperType
    name: "SuperType"
  ▶saySubName: ƒ ()
  ▼__proto__:
    ▶sayName: ƒ ()
    ▶constructor: ƒ SuperType()
    ▶__proto__: Object
> instance.constructor
< ƒ SuperType() {
    this.name = 'SuperType'; // 子类属性
  }
> instance.sayName()
< "SuperType"
```

- **特点**：利用原型，让一个引用类型继承另一个引用类型的属性及方法
- **优点**：**继承了父类的模板，又继承了父类的原型对象**
- **缺点**：
  - 可以在子类构造函数中，为子类实例增加实例属性。如果要**新增原型属性和方法**，则必须放在 `SubType.prototype = new SuperType('SubType');` 这样的语句**之后**执行。

- 无法实现多继承
- 来自原型对象的所有属性被**所有实例共享**

```
// 父类
function SuperType () {
  this.colors = ["red", "blue", "green"];
  this.name = "SuperType";
}
// 子类
function SubType () {}

// 原型链继承
SubType.prototype = new SuperType();

// 实例1
var instance1 = new SubType();
instance1.colors.push("blcak");
instance1.name = "change-super-type-name";
console.log(instance1.colors); // ["red", "blue", "green", "blcak"]
console.log(instance1.name); // change-super-type-name
// 实例2
var instance2 = new SubType();
console.log(instance2.colors); // ["red", "blue", "green", "blcak"]
console.log(instance2.name); // SuperType
```

```
▶ (4) ["red", "blue", "green", "blcak"]
change-super-type-name
▶ (4) ["red", "blue", "green", "blcak"]
SuperType
undefined
> instance1
▼ SubType {name: "change-super-type-name"} ⓘ
    name: "change-super-type-name"
  ▼ __proto__: SuperType
    ▶ colors: (4) ["red", "blue", "green", "blcak"]
      name: "SuperType"
    ▶ __proto__: Object
> instance2
▼ SubType {} ⓘ
  ▼ __proto__: SuperType
    ▶ colors: (4) ["red", "blue", "green", "blcak"]
      name: "SuperType"
    ▶ __proto__: Object
```

注意：更改 `SuperType` **引用类型属性**时，会使 `SubType` 所有实例共享这一更新。基础类型属性更新则不会。

- **创建子类实例时，无法向父类构造函数传参**，或者说是，没办法在不影响所有对象实例的情况下，向超类的构造函数传递参数

二、 构造继承

基本思想：在子类型的构造函数内部调用父类型构造函数。

注意：函数只不过是在特定环境中执行代码的对象，所以这里使用 apply/call 来实现。

使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

```
// 父类
function SuperType (name) {
  this.name = name; // 父类属性
}
SuperType.prototype.sayName = function () { // 父类原型方法
  return this.name;
};

// 子类
function SubType () {
  // 调用 SuperType 构造函数
  SuperType.call(this, 'SuperType'); // 在子类构造函数中，向父类构造函数传参
  // 为了保证子父类的构造函数不会重写子类的属性，需要在调用父类构造函数后，定义子类的属性
  this.subName = "SubType"; // 子类属性
};
// 子类实例
let instance = new SubType(); // 运行子类构造函数，并在子类构造函数中运行父类构造函数，this绑定到子类
```

```
> instance
< ▼SubType {name: "SuperType", subName: "SubType"}
    name: "SuperType"
    subName: "SubType"
  ▼__proto__:
    ▶constructor: ƒ SubType()
    ▶__proto__: Object
```

```
> instance.constructor
< ƒ SubType () {
    // 调用 SuperType 构造函数
    SuperType.call(this, 'SuperType');
    this.subName = "SubType"; // 子类属性
  }
```

- **优点：解决了1中子类实例共享父类引用对象的问题，实现多继承，创建子类实例时，可以向父类传递参数**

- **缺点**：
  - 实例并不是父类的实例，只是子类的实例
  - **只能**继承父类的实例属性和方法，**不能**继承原型属性/方法
  - **无法实现函数复用**，每个子类都有父类实例函数的副本，影响性能

三. 组合继承

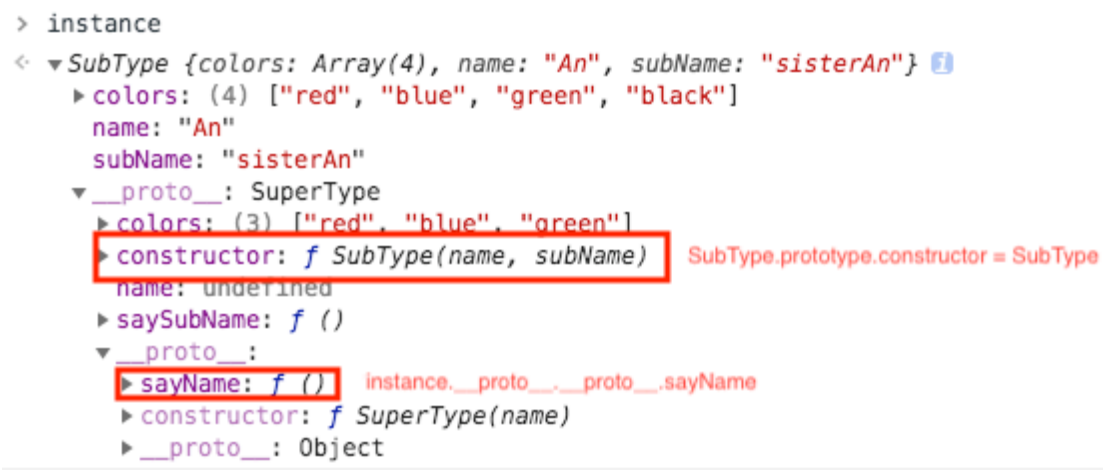
顾名思义，组合继承就是将原型链继承与构造函数继承组合在一起，从而发挥两者之长的一种继承模式。

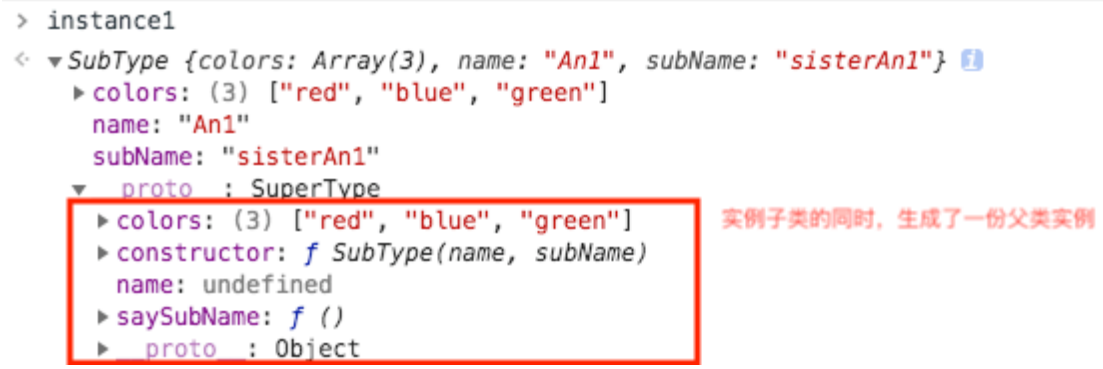
基本思想：使用**原型链**继承使用对原型属性和方法的继承，通过**构造函数**继承来实现对实例属性的继承。这样既能通过在原型上定义方法实现函数复用，又能保证每个实例都有自己的属性。

通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

```
// 父类
function SuperType (name) {
  this.colors = ["red", "blue", "green"];
  this.name = name; // 父类属性
}
SuperType.prototype.sayName = function () { // 父类原型方法
  return this.name;
};

// 子类
function SubType (name, subName) {
  // 调用 SuperType 构造函数
  SuperType.call(this, name); // ----第二次调用 SuperType----
  this.subName = subName;
};

// ----第一次调用 SuperType----
SubType.prototype = new SuperType(); // 重写原型对象，代之以一个新类型的实例

SubType.prototype.constructor = SubType; // 组合继承需要修复构造函数指向
SubType.prototype.saySubName = function () { // 子类原型方法
  return this.subName;
}

// 子类实例
let instance = new SubType('An', 'sisterAn')
instance.colors.push('black')
console.log(instance.colors) // ["red", "blue", "green", "black"]
instance.sayName() // An
instance.saySubName() // sisterAn

let instance1 = new SubType('An1', 'sisterAn1')
```

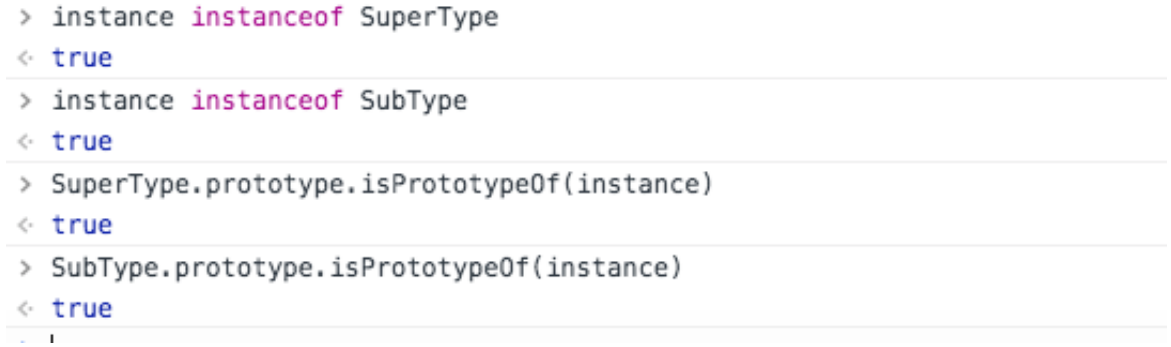
```
console.log(instance1.colors) //  ["red", "blue", "green"]
instance1.sayName() // An1
instance1.saySubName() // sisterAn1
```





第一次调用 `SuperType` 构造函数时，`SubType.prototype` 会得到两个属性 `name` 和 `colors`；当调用 `SubType` 构造函数时，第二次调用 `SuperType` 构造函数，这一次又在新对象属性上创建了 `name` 和 `colors`，这两个属性就会屏蔽原型对象上的同名属性。

```
// instanceof：instance 的原型链是针对 SuperType.prototype 进行检查的
instance instanceof SuperType // true
instance instanceof SubType // true

// isPrototypeOf：instance 的原型链是针对 SuperType 本身进行检查的
SuperType.prototype.isPrototypeOf(instance) // true
SubType.prototype.isPrototypeOf(instance) // true
```



- **优点：弥补了方式2的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法，不存在引用属性共享问题，可传参，可复用**
- **缺点**：
    - 调用了两次父类构造函数，生成了**两份实例**（子类实例将子类原型上的那份屏蔽了）

四. 寄生组合继承

在组合继承中，调用了两次父类构造函数，这里 **通过通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点**

主要思想：借用 **构造函数** 继承 **属性** ，通过 **原型链的混成形式** 来继承 **方法**

```
// 父类
function SuperType (name) {
  this.colors = ["red", "blue", "green"];
  this.name = name; // 父类属性
}
SuperType.prototype.sayName = function () { // 父类原型方法
  return this.name;
};

// 子类
function SubType (name, subName) {
  // 调用 SuperType 构造函数
  SuperType.call(this, name); // ----第二次调用 SuperType，继承实例属性----
  this.subName = subName;
};

// ----第一次调用 SuperType，继承原型属性----
SubType.prototype = Object.create(SuperType.prototype)

SubType.prototype.constructor = SubType; // 注意：增强对象

let instance = new SubType('An', 'sisterAn')
```

```
> instance
< ▼SubType {colors: Array(3), name: "An", subName: "sisterAn"}
    ▶colors: (3) ["red", "blue", "green"]
     name: "An"
     subName: "sisterAn"
    ▼__proto__: SuperType
      ▶constructor: ƒ SubType(name, subName)
      ▼__proto__:
        ▶sayName: ƒ ()
        ▶constructor: ƒ SuperType(name)
        ▶__proto__: Object
```

优点：

- 只调用一次 `SuperType` 构造函数，只创建一份父类属性
- 原型链保持不变
- 能够正常使用 `instanceof` 与 `isPrototypeOf`

五. 原型式继承

实现思路就是将子类的原型设置为父类的原型

```
// 父类
function SuperType (name) {
  this.colors = ["red", "blue", "green"];
  this.name = name; // 父类属性
}
SuperType.prototype.sayName = function () { // 父类原型方法
  return this.name;
};

/** 第一步 */
// 子类，通过 call 继承父类的实例属性和方法，不能继承原型属性/方法
function SubType (name, subName) {
  SuperType.call(this, name); // 调用 SuperType 的构造函数，并向其传参
  this.subName = subName;
}

/** 第二步 */
// 解决 call 无法继承父类原型属性/方法的问题
// Object.create 方法接受传入一个作为新创建对象的原型的对象，创建一个拥有指定原型和若干个指定属性的对象
// 通过这种方法指定的任何属性都会覆盖原型对象上的同名属性
SubType.prototype = Object.create(SuperType.prototype, {
  constructor: { // 注意指定 SubType.prototype.constructor = SubType
    value: SubType,
    enumerable: false,
    writable: true,
    configurable: true
  },
  run : {
    value: function(){ // override
      SuperType.prototype.run.apply(this, arguments);
        // call super
        // ...
    },
    enumerable: true,
    configurable: true,
    writable: true
  }
})
```

```
/** 第三步 */
// 最后：解决 SubType.prototype.constructor === SuperType 的问题
// 这里，在上一步已经指定，这里不需要再操作
// SubType.prototype.constructor = SubType;

var instance = new SubType('An', 'sistenAn')
```

```
> instance
< ▼SubType {colors: Array(3), name: "An", subName: "sistenAn"}
    ▶colors: (3) ["red", "blue", "green"]
     name: "An"
     subName: "sistenAn"
    ▼__proto__: SuperType
      ▶run: ƒ ()
      ▶constructor: ƒ SubType(name, subName)
      ▼__proto__:
        ▶sayName: ƒ ()
        ▶constructor: ƒ SuperType(name)
        ▶__proto__: Object
```

如果希望能 **多继承**，可使用 **混入** 的方式

```
// 父类 SuperType
function SuperType () {}
// 父类 OtherSuperType
function OtherSuperType () {}

// 多继承子类
function AnotherType () {
    SuperType.call(this) // 继承 SuperType 的实例属性和方法
    OtherSuperType.call(this) // 继承 OtherSuperType 的实例属性和方法
}

// 继承一个类
AnotherType.prototype = Object.create(SuperType.prototype);

// 使用 Object.assign 混合其它
Object.assign(AnotherType.prototype, OtherSuperType.prototype);
// Object.assign 会把  OtherSuperType 原型上的函数拷贝到 AnotherType 原型上，使
AnotherType 的所有实例都可用 OtherSuperType 的方法

// 重新指定 constructor
AnotherType.prototype.constructor = AnotherType;

AnotherType.prototype.myMethod = function() {
     // do a thing
};

let instance = new AnotherType()
```

最重要的部分是：

- `SuperType.call` 继承实例属性方法
- 用 `Object.create()` 来继承原型属性与方法
- 修改 `SubType.prototype.constructor` 的指向

ES6 继承

首先，实现一个简单的 ES6 继承：

```
class People {
    constructor(name) {
        this.name = name
    }
    run() { }
}

// extends 相当于方法的继承
// 替换了上面的3行代码
class Man extends People {
    constructor(name) {
        // super 相当于属性的继承
        // 替换了 People.call(this, name)
        super(name)
        this.gender = '男'
    }
    fight() { }
}
```

核心代码

`extends` 继承的核心代码如下，其实现和上述的寄生组合式继承方式一样

```
function _inherits(subType, superType) {
    // 创建对象，Object.create 创建父类原型的一个副本
    // 增强对象，弥补因重写原型而失去的默认的 constructor 属性
    // 指定对象，将新创建的对象赋值给子类的原型 subType.prototype
    subType.prototype = Object.create(superType && superType.prototype, {
        constructor: { // 重写 constructor
            value: subType,
            enumerable: false,
            writable: true,
            configurable: true
        }
    });
    if (superType) {
```

```
        Object.setPrototypeOf
            ? Object.setPrototypeOf(subType, superType)
            : subType.__proto__ = superType;
    }
}
```

继承的使用场景

- 不要仅仅为了使用而使用它们，这只是在浪费时间而已。
- 当需要创建 **一系列拥有相似特性的对象** 时，那么创建一个包含所有共有功能的通用对象，然后在更特殊的对象类型中继承这些特性。
- 应避免多继承，造成混乱。

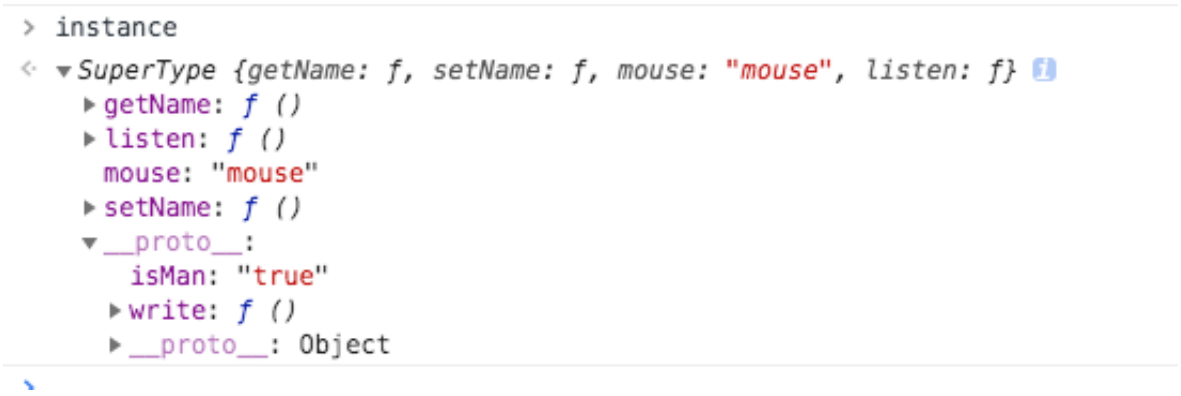
注: 考虑到JavaScript的工作方式，由于原型链等特性的存在，在不同对象之间功能的共享通常被叫做 **委托** - 特殊的对象将功能委托给通用的对象类型完成。这也许比将其称之为继承更为贴切，因为“被继承”了的功能并没有被拷贝到正在“进行继承”的对象中，相反它仍存在于通用的对象中。

扩展：new

new 关键字创建的对象**实际上是对新对象 this 的不断赋值，并将 prototype 指向类的 prototype 所指向的对象。**

```
var SuperType = function (name) {
    var nose = 'nose' // 私有属性
    function say () {} // 私有方法

    // 特权方法
    this.getName = function () {}
    this.setName = function () {}

    this.mouse = 'mouse' // 对象公有属性
    this.listen = function () {} // 对象公有方法

    // 构造器
    this.setName(name)
}

SuperType.age = 10 // 类静态公有属性（对象不能访问）
SuperType.read = function () {} // 类静态公有方法（对象无法访问）

SuperType.prototype = { // 对象赋值（也可以一一赋值）
    isMan: 'true', // 公有属性
    write: function () {} // 公有方法
}
```

```
var instance = new SuperType()
```



所以类的构造函数内定义的 **私有变量或方法** ，以及类定义的 **静态公有属性及方法** ，在 **new** 的实例对象中都将 **无法访问** 。

扩展：继承机制的设计思想

关于继承机制的设计思想，请参见 Javascript继承机制的设计思想 。

什么变量保存在堆/栈中?

看到这个问题，第一反应表示很简单，基本类型保存在栈中，引用类型保存到堆中✌✌✌，但仅仅就如此简单吗？我们接下来详细看一看

JS 数据类型

我们知道 JS 就是动态语言，因为在声明变量之前并不需要确认其数据类型，所以 **JS 的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据** 。

JS 值有 8 种数据类型：

- Boolean：有 `true` 和 `false`
- Undefined：没有被赋值的变量或变量被提升时的，都会有个默认值 `undefined`
- Null：只有一个值 `null`
- Number：数字类型
- BigInt（ES10）：表示大于 `253 - 1` 的整数
- String：字符类型
- Symbol（ES6）
- Object：对象类型

其中前 7 种数据类型称为基本类型，把最后一个对象类型称为引用类型

JS中的变量存储机制

JS 内存空间分为栈(stack)空间、堆(heap)空间、代码空间。其中代码空间用于存放可执行代码。

栈空间

栈是内存中一块用于存储局部变量和函数参数的线性结构，遵循着先进后出 (LIFO / Last In First Out) 的原则。栈由内存中占据一片连续的存储空间，出栈与入栈仅仅是指针在内存中的上下移动而已。

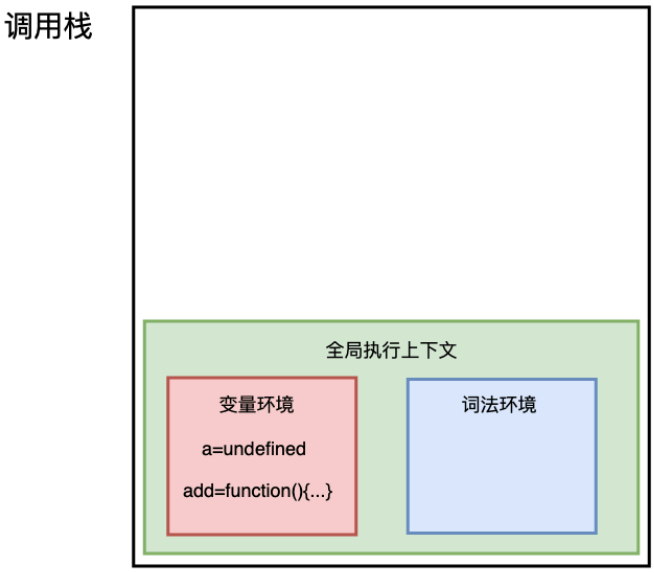
JS 的栈空间就是我们所说的调用栈，是用来存储执行上下文的，包含变量空间与词法环境，var、function保存在变量环境，let、const 声明的变量保存在词法环境中。

```
var a = 1
function add(a) {
  var b = 2
  let c = 3
  return a + b + c
}

// 函数调用
add(a)
```

这段代码很简单，就是创建了一个 `add` 函数，然后调用了它。

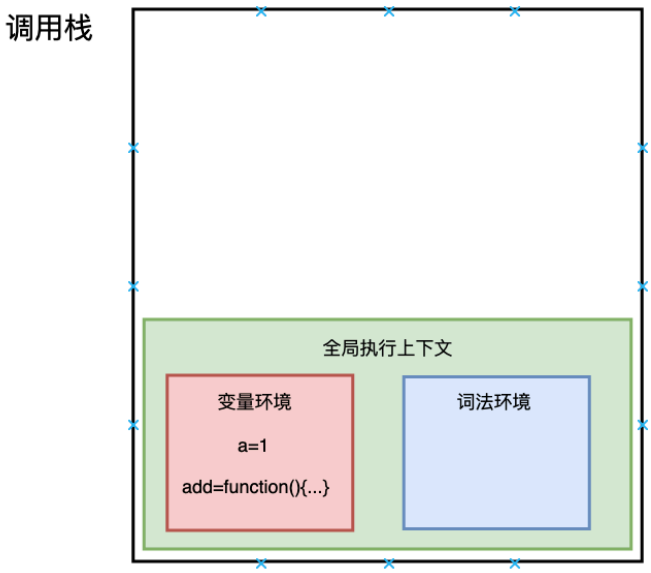
下面我们就一步步的介绍整个函数调用执行的过程。

在执行这段代码之前，JavaScript 引擎会先创建一个全局执行上下文，包含所有已声明的函数与变量：



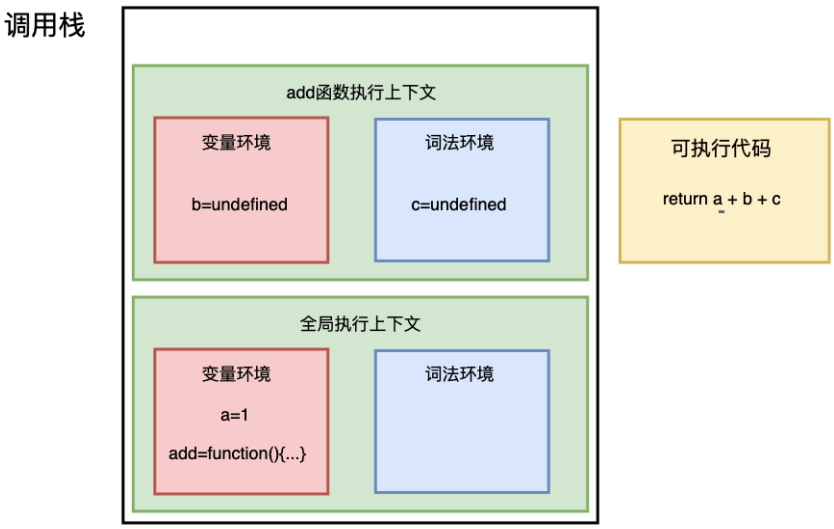
从图中可以看出，代码中的全局变量 `a` 及函数 `add` 保存在变量环境中。

执行上下文准备好后，开始执行全局代码，首先执行 `a = 1` 的赋值操作，

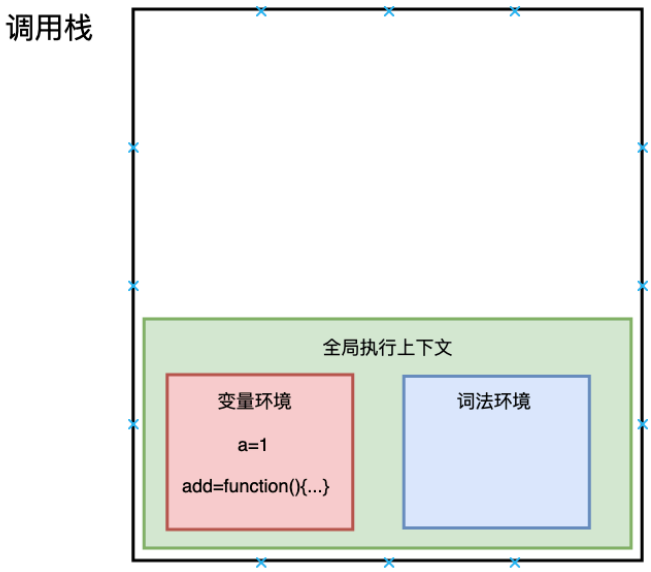


赋值完成后 a 的值由 undefined 变为 1，然后执行 `add` 函数，JavaScript 判断出这是一个函数调用，然后执行以下操作：

- 首先，从全局执行上下文中，取出 add 函数代码
- 其次，对 add 函数的这段代码进行编译，并创建该函数的执行上下文和可执行代码，并将执行上下文压入栈中



- 然后，执行代码，返回结果，并将 add 的执行上下文也会从栈顶部弹出，此时调用栈中就只剩下全局上下文了。



至此，整个函数调用执行结束了。

上面需要注意的是：函数（add）中存放在栈区的数据，在函数调用结束后，就已经自动的出栈，换句话说：栈中的变量在函数调用结束后，就会自动回收。

所以，通常栈空间都不会设置太大，而基本类型在内存中占有固定大小的空间，所以它们的值保存在栈空间，我们通过 **按值访问** 。它们也不需要手动管理，函数调时创建，调用结束则消失。

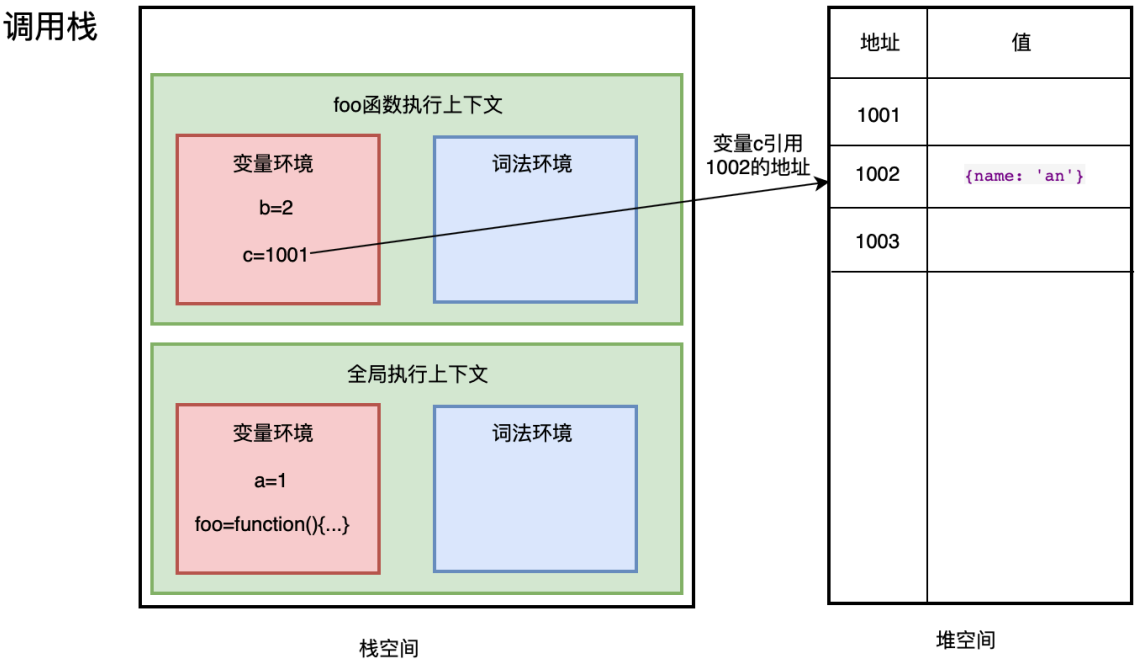
堆空间

堆数据结构是一种树状结构。它的存取数据的方式与书架和书非常相似。我们只需要知道书的名字就可以直接取出书了，并不需要把上面的书取出来。

在栈中存储不了的数据比如对象就会被存储在堆中，在栈中只是保留了对象在堆中的地址，也就是对象的**引用** ，对于这种，我们把它叫做 **按引用访问** 。

举个例子帮助理解一下：

```
var a = 1
function foo() {
  var b = 2
  var c = { name: 'an' } // 引用类型
}

// 函数调用
foo()
```



所以，堆空间通常很大，能存放很多大的数据，不过缺点是分配内存和回收内存都会占用一定的时间

JS中的变量存储机制与闭包

对以上总结一下，JS 内存空间分为栈(stack)空间、堆(heap)空间、代码空间。其中代码空间用于存放可执行代码

- 基本类型：保存在栈内存中，因为这些类型在存中分别占有固定大小的空间，通过按值来访问。
- 引用类型：保存在堆内存中，因为这种值的大小不固定，因此不能把它们保存到栈内存中，但内存地址大小的固定的，因此保存在堆内存中，在栈内存中存放的只是该对象的访问地址。当查询引用类型的变量时， 先从栈中读取内存地址， 然后再通过地址找到堆中的值。对于这种，我们把它叫做按引用访问。

闭包

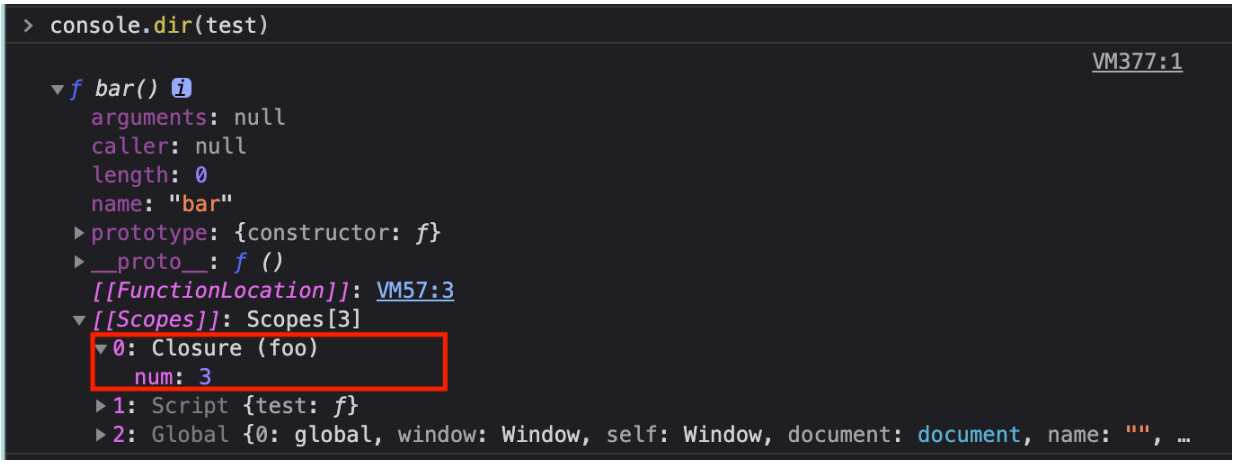
那么闭包喃？既然基本类型变量存储在栈中，栈中数据在函数执行完成后就会被自动销毁，那执行函数之后为什么闭包还能引用到函数内的变量？

```
function foo() {
  let num = 1 // 创建局部变量 num 和局部函数 bar
  function bar() { // bar()是函数内部方法，是一个闭包
      num++
      console.log(num) // 使用了外部函数声明的变量，内部函数可以访问外部函数的变量
  }
  return bar // bar 被外部函数作为返回值返回了,返回的是一个闭包
}

// 测试
let test = foo()
test() // 2
test() // 3
```

在执行完函数 `foo` 后，`foo` 中的变量 `num` 应该被弹出销毁，为什么还能继续使用喃？

这说明闭包中的变量没有保存在栈中，而是保存到了堆中：

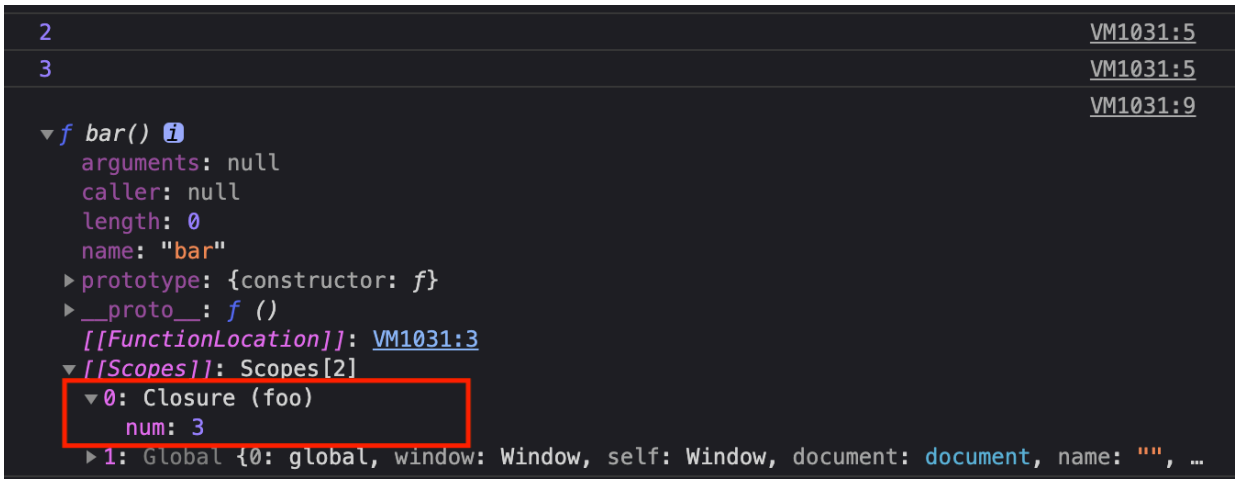
```
console.dir(test)
```



所以 JS 引擎判断当前是一个闭包时，就会在堆空间创建换一个“closure(foo)”的对象（这是一个内部对象，JS 是无法访问的），用来保存 num 变量

注意，即使不返回函数（闭包没有被返回）：

```
function foo() {
  let num = 1 // 创建局部变量 num 和局部函数 bar
  function bar() { // bar()是函数内部方法，是一个闭包
      num++
      console.log(num) // 使用了外部函数声明的变量，内部函数可以访问外部函数的变量
  }
  bar() // 2
  bar() // 3
  console.dir(bar)
}

foo()
```



总结

JS 就是动态语言，因为在声明变量之前并不需要确认其数据类型，所以 JS 的变量是没有数据类型的，值才有数据类型，变量可以随时持有任何类型的数据。

JS 值有 8 种数据类型，它们可以分为两大类——基本类型和引用类型。其中，基本类型的数据是存放在栈中，引用类型的数据是存放在堆中的。堆中的数据是通过引用和变量关联起来的。

闭包除外，JS 闭包中的变量值并不保存中栈内存中，而是保存在堆内存中。

[原文]

实现颜色转换 'rgb(255, 255, 255)' -> '#FFFFFF' 的多种思路

仔细观察本题，本题可分为三个步骤：

- 从 `rgb(255, 255, 255)` 中提取出 `r=255` 、 `g=255` 、 `b=255`
- 将 `r` 、 `g` 、 `b` 转换为十六进制，不足两位则补零
- 组合 `#`

提取 r、g、b

方式一：利用 match

利用 `match()` 方法，读取出 `r` 、 `g` 、 `b`

```
function rgb2hex(sRGB) {
    const reg = /^(rgb|RGB)\(\s*(\d{1,3})\s*,\s*(\d{1,3}\s*),\s*
(\d{1,3})\s*\)$/
    const rbg = sRGB.match(reg)
    return rbg
}

// 测试
rgb2hex('rgb(255, 255, 255)')
// ["rgb(255, 255, 255)", "rgb", "255", "255", "255", index: 0, input:
"rgb(255, 255, 255)", groups: undefined]
rgb2hex('rgb(16, 10, 255)')
// ["rgb(16, 10, 255)", "rgb", "16", "10", "255", index: 0, input: "rgb(16,
10, 255)", groups: undefined]
```

`r = rgb[2]` 、 `g = rgb[3]` 、 `b = rgb[4]`

方式二：利用 `match()` 方法（2）

`rgb(255, 255, 255)` 中 `r` 、 `g` 、 `b` 分别为连续的数字，所以我们可以利用正则 `/\d+/g` 获取取所有连着的数字

```
function rgb2hex(sRGB) {
    const rgb = sRGB.match(/\d+/g);
    return rgb
}

// 测试
rgb2hex('rgb(255, 255, 255)')
// ["255", "255", "255"]
rgb2hex('rgb(16, 10, 255)')
// ["16", "10", "255"]
```

方式三：replace + 利用 split

观察 `rgb(255, 255, 255)` 的每一个色值是透过 `,` 连接一起的，所以我们考虑是否能通过 `split(',')` 拆分出每一个色值，主要考虑两步

- 替换 `rgb(255, 255, 255)` 的部分字符（`rgb`、`(`、`)`）为 `''`
- 拆分出每一个色值

```
function rgb2hex(sRGB) {
    const rgb = sRGB.replace(/(?:\(|\)|rgb|RGB)*/g, '').split(',')
    return rgb
}
// 测试
rgb2hex('rgb(255, 255, 255)')
// ["255", " 255", " 255"]
rgb2hex('rgb(16, 10, 255)')
// ["16", " 10", " 255"]
```

转换为十六进制，不足补零

转换为十六进制，可采用：

- `(+n).toString(16)` 或 `Number(n).toString(16)` `

不足两位则补零：

- `('0' + r16).slice(-2)`
- `r16.padStart(2, '0')`
- `(r < 16? '0':'') + r16`
- `r16.length < 2 ? '0' + r16 : r16`
- `((1 << 24) + (Number(r) << 16) + (Number(g) << 8) + Number(b)).toString(16).slice(1)`

方式多种多样，发散思维，🤔更多

组合

reduce

注意，输出为可为大写（`#FFFFFF`）或小写字符（`#ffffff`），本题为大写

```
rgb.reduce((acc, cur) => acc + hex, '#').toUpperCase()
```

+

也可以通过`+`连接，按情况而定

方式多种多样，发散思维，🤔更多

总结

把本题拆分成以上三步，选取每步的一种实现方式组合实现本题，最终实现方案多种多样，简单这里列一下 **其中的部分组合**

组合一

```
function rgb2hex(sRGB) {
    var rgb = sRGB.replace(/(?:\(|\)|rgb|RGB)*/g, '').split(',')
    return rgb.reduce((acc, cur) => {
        var hex = (cur < 16? '0':'') + Number(cur).toString(16)
        return acc + hex
    }, '#').toUpperCase()
}

// 测试
rgb2hex('rgb(255, 255, 255)')
// "#FFFFFF"
rgb2hex('rgb(16, 10, 255)')
// "#100AFF"
```

组合二

```
function rgb2hex(rgb) {
    const rgb = rgb.match(/\d+/g);
    const hex = (n) => {
        return ("0" + Number(n).toString(16)).slice(-2);
    }
    return rgb.reduce((acc, cur) => acc + hex, '#').toUpperCase()
}

// 测试
rgb2hex('rgb(255, 255, 255)')
// "#FFFFFF"
rgb2hex('rgb(16, 10, 255)')
// "#100AFF"
```

组合三

```
function rgb2hex(sRGB) {
    const rgb = sRGB.replace(/(?:\(|\)|rgb|RGB)*/g, '').split(',')
    return "#" + ((1 << 24) + (Number(rgb[0]) << 16) + (Number(rgb[1]) <<
8) + Number(rgb[2])).toString(16).slice(1).toUpperCase()
}

// 测试
rgb2hex('rgb(255, 255, 255)')
// "#FFFFFF"
rgb2hex('rgb(16, 10, 255)')
// "#100AFF"
rgb2hex('rgb(1, 2, 3)')
// "#010203"
```

原文

实现一个异步求和函数

简化：两数之和

我们先来简单的实现一个异步两数之和函数

```
function sumT(a, b) {
    return await new Promise((resolve, reject) => {
        asyncAdd(a, b, (err, res) => {
            if(!err) {
                resolve(res)
            }
            reject(err)
        })
    })
}

// 测试
const test = await sumT(1, 2)
console.log(test)
// 3
```

加深：多数之和

上面我们实现了两数之和，然后扩展到多数之和喃?

提到数组求和问题，我们首先想到的是 `reduce`

> `reduce()` 方法对数组中的每个元素执行一个由您提供的**reducer**函数(升序执行)，将其结果汇总为单个返回值。
>
> —— MDN

```
arr.reduce(callback(acc, cur[, idx[, arr]])[, initialValue])
```

`callback` 函数接收4个参数:

- `acc` ：累计器
- `cur` ：当前值
- `idx` ： 当前索引
- `arr` ：源数组

其中， `initialValue` 可选，

- 如果有 `initialValue` ： `acc` 取值为 `initialValue` ， `cur` 取数组中的第一个值

- 如果没有：`acc` 取数组中的第一个值，`cur` 取数组中的第二个值

```
const arr = [1, 2, 3, 4];
const reducer = (acc, cur) => acc + cur;

// 1 + 2 + 3 + 4
console.log(arr.reduce(reducer));
// 输出：10

// 5 + 1 + 2 + 3 + 4
console.log(arr.reduce(reducer, 5));
// 输出：15
```

关于本题：来自@champkeh

设置初始值为 `Promise.resolve(0)`，经历 `5` 次求和：

```
function sum(...args) {
    return new Promise(resolve => {
        args.reduce((acc, cur) => acc.then(total => sumT(total, cur)),
Promise.resolve(0)).then(resolve)
    })
}

// 测试
await sum(1, 2, 3, 4, 5)
// 15
```

但这存在一个耗时较长的问题，我们可以计算下时间：

```
console.time("sum")
// 测试
await sum(1, 2, 3, 4, 5)
// 15
console.timeEnd("sum")
```

```
> console.time("sum")
  // 测试
  await sum(1, 2, 3, 4, 5)
  // 15
  console.timeEnd("sum")
  sum: 5010.7509765625 ms                                    VM2420:5
```

也就是说，我们每次求和都会花费 1s，串行异步求和，这显然不是最优的

优化：使用 Promise.all

我们可以两两一组，使用 `Promise.all` 求和，再把和两两一组继续求和.....，知道只剩余一个就是最终的结果

```
async function sum(...args) {
    // 用于考察每次迭代的过程
    console.log(args)

    // 如果仅有一个，直接返回
    if(args.length === 1) return args[0]
    let result = []
    // 两两一组，如果有剩余一个，直接进入
    for(let i = 0; i < args.length - 1; i+=2) {
        result.push(sumT(args[i], args[i + 1]))
    }
    if(args.length%2)  result.push(args[args.length-1])
    // Promise.all 组内求和
    return sum(...await Promise.all(result))
}

// 测试
test = await sum(1, 2, 3, 4, 5)
// 15
```

```
test = await sum(1, 2, 3, 4, 5)
▶ (5) [1, 2, 3, 4, 5]                    VM722:2
▶ (3) [3, 7, 5]                          VM722:2
▶ (2) [10, 5]                            VM722:2
▶ [15]                                   VM722:2
← 15
```

```
console.time("sum")
await sum(1, 2, 3, 4, 5)
console.timeEnd("sum")
```

```
▶ [15]                                   VM2841:3
sum: 3017.781005859375 ms                VM2855:3
← undefined
```

一道腾讯手写题，如何判断 url 中只包含 qq.com

例如：

```
http://www.qq.com  // 通过

http://www.qq.com.cn  // 不通过

http://www.qq.com/a/b  // 通过

http://www.qq.com?a=1  // 通过

http://www.123qq.com?a=1  // 不通过
```

解答：正则

```
function check(url){
  if(/\/\/w+\.qq\.com[^.]*$/.test(url)){
    return true;
  }else{
    return false;
  }

}
check('http://www.qq.com')
// true

check('http://www.qq.com.cn')
// false

check('http://www.qq.com/a/b')
// true

check('http://www.qq.com?a=1')
// true

check('http://www.123qq.com?a=1')
// false
```

这个正则很简单，包含 `.qq.com` 就可以，但是有一种情况，如果域名不是包含 `qq.com` 而仅仅是参数后面包含了 `qq.com` 怎么办？例如 `http://www.baidu.com?redirect=http://www.qq.com/a`

```
check('http://www.baidu.com?redirect=http://www.qq.com/a')
// true
```

如何排除这种情况?

```
function check(url){
  if(/^https?:\/\/w+\.qq\.com[^.]*$/.test(url)){
    return true;
  }else{
    return false;
  }

}
check('http://www.qq.com')
// true

check('http://www.qq.com.cn')
// false

check('http://www.qq.com/a/b')
// true

check('http://www.qq.com?a=1')
// true

check('http://www.123qq.com?a=1')
// false

check('http://www.baidu.com?redirect=http://www.qq.com/a')
// true
```

若有收获，就点个赞吧

由一道bilibili面试题看Promise异步执行机制

```
var date = new Date()

console.log(1, new Date() - date)

setTimeout(() => {
    console.log(2, new Date() - date)
}, 500)

Promise.resolve().then(console.log(3, new Date() - date))

while(new Date() - date < 1000) {}

console.log(4, new Date() - date)
```

求上面的输出顺序和输出值，为什么?

答案:

```
1 0
3 1
4 1000
2 1000
```

其中，关于时间差结果可能因为计算机性能造成的微小差异，可忽略不计

你答对了吗？下面我们由浅入深探索本题

由浅入深探索 Promise 异步执行

首先，看一下 `event loop` 的基础必备内容

`event loop` 执行顺序:

- 首先执行 `script` 宏任务
- 执行同步任务，遇见微任务进入微任务队列，遇见宏任务进入宏任务队列
- 当前宏任务执行完出队，检查微任务列表，有则依次执行，直到全部执行完
- 执行浏览器 UI 线程的渲染工作
- 检查是否有`Web Worker`任务，有则执行
- 执行下一个宏任务，回到第二步，依此循环，直到宏任务和微任务队列都为空

微任务包括: `MutationObserver`、`Promise.then()或catch()`、`Promise为基础开发的其它技术，比如fetch API`、`V8`的垃圾回收过程、`Node独有的process.nextTick`、`Object.observe`（已废弃； `Proxy` 对象替代）

宏任务包括：`script` 、`setTimeout`、`setInterval` 、`setImmediate` 、`I/O` 、`UI rendering` 、 `postMessage` 、 `MessageChannel`

注意： 下面的题目都是执行在浏览器环境下

遇到不好理解的，可结合 `promise` 源码 进行理解，就很简单了

1. 同步 + Promise

题目一：

```
var promise = new Promise((resolve, reject) => {
    console.log(1)
    resolve()
    console.log(2)
})
promise.then(()=>{
    console.log(3)
})
console.log(4)
// 1
// 2
// 4
// 3
```

解析：

- 首先明确， `Promise` 构造函数是同步执行的， `then` 方法是异步执行的
- 开始 `new Promise` ，执行构造函数同步代码，输出 `1`
- 再 `resolve()` ， 将 `promise` 的状态改为了 `resolved` ，并且将 `resolve` 值保存下来，此处没有传值
- 执行构造函数同步代码，输出 `2`
- 跳出`promise`，往下执行，碰到 `promise.then` 这个微任务，将其加入微任务队列
- 执行同步代码，输出 `4`
- 此时宏任务执行完毕，开始检查微任务队列，执行 `promise.then` 微任务，输出 `3`

题目二：

```
var promise = new Promise((resolve, reject) => {
    console.log(1)
})
promise.then(()=>{
    console.log(2)
})
console.log(3)
// 1
// 3
```

解析：

- 开始 `new Promise` ，执行构造函数同步代码，输出 `1`
- 再 `promise.then` ，因为 `promise` 中并没有 `resolve` ，所以 `then` 方法不会执行
- 执行同步代码，输出 `3`

题目三：

```
var promise = new Promise((resolve, reject) => {
    console.log(1)
})
promise.then(console.log(2))
console.log(3)
// 1
// 2
// 3
```

解析：

- 首先明确， `.then` 或者 `.catch` 的参数期望是函数，传入非函数则会发生值透传（ `value => value` ）
- 开始 `new Promise` ，执行构造函数同步代码，输出 `1`
- 然后 `then()` 的参数是一个 `console.log(2)` （注意：并不是一个函数），是立即执行的，输出 `2`
- 执行同步代码，输出 `3`

题目四：

```
Promise.resolve(1)
  .then(2)
  .then(Promise.resolve(3))
  .then(console.log)
// 1
```

解析：

- `then(2)` 、 `then(Promise.resolve(3))` 发生了值穿透，直接执行最后一个 `then` ，输出 `1`

题目五：

```
var promise = new Promise((resolve, reject) => {
    console.log(1)
    resolve()
    reject()
})
promise.then(()=>{
    console.log(2)
```

```
}).catch(()=>{
    console.log(3)
})
console.log(4)
// 1
// 4
// 2
```

解析：

- 开始 `new Promise` ，执行构造函数同步代码，输出 `1`
- 再 `resolve()` ， 将 `promise` 的状态改为了 `resolved` ，并且将 `resolve` 值保存下来，此处没有传值
- 再 `reject()` ，此时 `promise` 的状态已经改为了 `resolved` ，不能再重新翻转（状态转变只能是pending —> resolved 或者 pending —> rejected，状态转变不可逆）
- 跳出 `promise` ，往下执行，碰到 `promise.then` 这个微任务，将其加入微任务队列
- 往下执行，碰到 `promise.catch` 这个微任务，此时 `promise` 的状态为 `resolved` （非 `rejected` ），忽略 `catch` 方法
- 执行同步代码，输出 `4`
- 此时宏任务执行完毕，开始检查微任务队列，执行 `promise.then` 微任务，输出 `2`

题目六：

```
Promise.resolve(1)
  .then(res => {
    console.log(res);
    return 2;
  })
  .catch(err => {
    return 3;
  })
  .then(res => {
    console.log(res);
  });
// 1
// 2
```

解析：

- 首先 `resolve(1)` ， 状态改为了 `resolved` ，并且将 `resolve` 值保存下来
- 执行 `console.log(res)` 输出 `1`
- 返回 `return 2` 实际上是包装成了 `resolve(2)`
- 状态为 `resolved` ， `catch` 方法被忽略
- 最后 `then` ，输出 `2`

2. 同步 + Promise + setTimeout

题目一：

```
setTimeout(() => {
  console.log(1)
})
Promise.resolve().then(() => {
  console.log(2)
})
console.log(3)
// 3
// 2
// 1
```

解析：

- 首先 `setTimout` 被放入宏任务队列
- 再 `Promise.resolve().then` ， `then` 方法被放入微任务队列
- 执行同步代码，输出 `3`
- 此时宏任务执行完毕，开始检查微任务队列，执行 `then` 微任务，输出 `2`
- 微任务队列执行完毕，检查执行一个宏任务
- 发现 `setTimeout` 宏任务，执行输出 `1`

题目二：

```
var promise = new Promise((resolve, reject) => {
  console.log(1)
  setTimeout(() => {
    console.log(2)
    resolve()
  }, 1000)
})

promise.then(() => {
  console.log(3)
})
promise.then(() => {
  console.log(4)
})
console.log(5)
// 1
// 5
// 2
// 3
// 4
```

解析：

- 首先明确，当遇到 `promise.then` 时，如果当前的 `Promise` 还处于 `pending` 状态，我们

并不能确定调用 `resolved` 还是 `rejected`，只有等待 `promise` 的状态确定后，再做处理，所以我们需要把我们的两种情况的处理逻辑做成 `callback` 放入 `promise` 的回调数组内，当 `promise` 状态翻转为 `resolved` 时，才将之前的 `promise.then` 推入微任务队列

- 开始，`Promise` 构造函数同步执行，输出 `1`，执行 `setTimeout`
- 将 `setTimeout` 加入到宏任务队列中
- 然后，第一个 `promise.then` 放入 `promise` 的回调数组内
- 第二个 `promise.then` 放入 `promise` 的回调数组内
- 执行同步代码，输出 `5`
- 检查微任务队列，为空
- 检查宏任务队列，执行 `setTimeout` 宏任务，输入 `2`，执行 `resolve`，`promise` 状态翻转为 `resolved`，将之前的 `promise.then` 推入微任务队列
- `setTimeout` 宏任务出队，检查微任务队列
- 执行第一个微任务，输出 `3`
- 执行第二个微任务，输出 `4`

回到开头

现在看，本题就很简单了

```
var date = new Date()

console.log(1, new Date() - date)

setTimeout(() => {
    console.log(2, new Date() - date)
}, 500)

Promise.resolve().then(console.log(3, new Date() - date))

while(new Date() - date < 1000) {}

console.log(4, new Date() - date)
```

解析：

- 首先执行同步代码，输出 `1 0`
- 遇到 `setTimeout`，定时 `500ms` 后执行，此时，将 `setTimeout` 交给异步线程，主线程继续执行下一步，异步线程执行 `setTimeout`
- 主线程执行 `Promise.resolve().then`，`.then` 的参数不是函数，直接执行（`value => value`），输出 `3 1`
- 主线程继续执行同步任务 `whlie`，等待 `1000ms`，在此期间，`setTimeout` 定时 `500ms` 完成，异步线程将 `setTimeout` 回调事件放入宏任务队列中
- 继续执行下一步，输出 `4 1000`
- 检查微任务队列，为空
- 检查宏任务队列，执行 `setTimeout` 宏任务，输入 `2 1000`

总结

- `Promise` 构造函数是同步执行的， `then` 方法是异步执行的
- `.then` 或者 `.catch` 的参数期望是函数，传入非函数则会直接执行
- `Promise`的状态一经改变就不能再改变，构造函数中的 `resolve` 或 `reject` 只有第一次执行有效，多次调用没有任何作用
- `.then`方法是能接收两个参数的，第一个是处理成功的函数，第二个是处理失败的函数，再某些时候你可以认为`catch`是`.then`第二个参数的简便写法
- 当遇到 `promise.then` 时， 如果当前的 `Promise` 还处于 `pending` 状态，我们并不能确定调用 `resolved` 还是 `rejected` ，只有等待 `promise` 的状态确定后，再做处理，所以我们需要把我们的两种情况的处理逻辑做成 `callback` 放入 `promise` 的回调数组内，当 `promise` 状态翻转为 `resolved` 时，才将之前的 `promise.then` 推入微任务队列

[原文]

es6 及 es6+ 的能力集，你最常用的，这其中最有用的，都解决了什么问题

我最常用的

ES6 的特性是使用最多的，包括类、模块化、箭头函数、函数参数默认值、模板字符串、解构赋值、延展操作符、Promise、let 与 const等等，这部分已经是开发必备了，没什么好说的

另外还有：

- ES7 的 `Array.prototype.includes()`
- ES8 的 async/await 、String padding: `padStart()`和`padEnd()` 、 `Object.values()`
- ES9 的 Rest/Spread 属性、for await of、 `Promise.finally()`
- ES10 的 `Array.prototype.flat()` 、 `Array.prototype.flatMap()` 、String的 `trimStart()` `trimEnd()`
- ES11 的 `Promise.allSettled` 、空值处理（ `??` ）及可选链（ `?.` ）
- ES12 的逻辑赋值操作符（ `||=` 、 `&&=` 、 `??=` ）、数字分隔符（ `1_000_000_000` ）、 `Promise.any()`

最有用的

ES6 的特性都很有用，ES7-ES11中，我比较感兴趣的是：

- ES8 的 async/await
- ES9 的 for await of
- ES11 的 `Promise.allSettled` 、ES9 的 `Promise.finally()` 、ES12 的 `Promise.any()`
- 还有常用的逻辑操作：逻辑赋值操作符、数字分隔符、空值处理及可选链等都很大的简洁优化了我们的代码

其中，async/await 异步终极解决方案， `for await of` 异步串行， `Promise.allSettled` 解决了 `Promise.all` 的只要一个请求失败了就会抛出错误的问题，当我们一次发起多个请求时，所有结果都能返回，无论成功或失败，等等等，不了解的可以往下查找

下面列一下所有的特性，查漏补缺

ES6（ES2015）

- 类
- 模块化
- 箭头函数
- 函数参数默认值
- 模板字符串
- 解构赋值
- 扩展操作符
- 对象属性简写

- Promise
- let 与 const

具体不再冗余介绍，这个属于前端基础

ES7（ES2016）

- Array.prototype.includes()
- 指数操作符

Array.prototype.includes()

```
[1, 2].includes(1) // true
```

指数操作符

```
2**5 // 32
```

ES8（ES2017）

- async/await
- Object.values()
- Object.entries()
- String padding: `padStart()`和`padEnd()`，填充字符串达到当前长度
- Object.getOwnPropertyDescriptors()
- 函数参数列表结尾允许逗号
- SharedArrayBuffer对象
- Atomics对象

async/await

异步终极解决方案

```
async getInfo(){
    const res = await api.getData()
    // ...
}
```

Object.values()

```
Object.values({a: 1, b: 2, c: 3})
// [1, 2, 3]
```

Object.entries()

```
Object.values({a: 1, b: 2, c: 3})
// [["a", 1], ["b", 2], ["c", 3]]
```

String padding: `padStart()` 和 `padEnd()`

padStart() 方法用另一个字符串填充当前字符串(如果需要的话，会重复多次)，以便产生的字符串达到给定的长度。从当前字符串的左侧开始填充

```
// padStart
'sister'.padStart(7, '0') // "0sister"
// padEnd
'sister'.padEnd(7, '0') // "sister0"
```

Object.getOwnPropertyDescriptors()

`Object.getOwnPropertyDescriptors()` 函数用来获取一个对象的所有自身属性的描述符，如果没有任何自身属性，则返回空对象

函数参数列表结尾允许逗号

这是一个不痛不痒的更新，主要作用是方便使用git进行多人协作开发时修改同一个函数减少不必要的行变更

ES9（ES2018）

- 异步迭代（for await of）
- Promise.finally()
- Rest/Spread 属性
- 新的正则表达式特性
    - 正则表达式反向断言（lookbehind）
    - 正则表达式dotAll模式
    - 正则表达式命名捕获组（Regular Expression Named Capture Groups）
    - 正则表达式 Unicode 转义
    - 非转义序列的模板字符串

异步迭代（for await of）

`await` 可以和 `for...of` 循环一起使用，以串行的方式运行异步操作

```
async function getInfos(arr) {
  for await (let i of arr) {
    getData(i)
  }
}
```

Promise.finally()

无论 `Promise` 运行成功还是失败，都会运行 `finally`

```
function getInfos() {
  getData1()
  .then(getData2)
  .catch(err => {
    console.log(err);
  })
  .finally(() => {
    // ...
  });
}
```

Rest/Spread 属性

```
const values = [1, 2, 3]
console.log( Math.min(...values) ) // 1
```

新的正则表达式特性

正则表达式命名捕获组（Regular Expression Named Capture Groups）

在一些正则表达式模式中，使用数字进行匹配可能会令人混淆。例如，使用正则表达式 `/(\d{4})-(\d{2})-(\d{2})/` 来匹配日期。因为美式英语中的日期表示法和英式英语中的日期表示法不同，所以很难区分哪一组表示日期，哪一组表示月份

```
const re = /(\d{4})-(\d{2})-(\d{2})/;
const match= re.exec('2019-01-01');
console.log(match[0]);    // → 2019-01-01
console.log(match[1]);    // → 2019
console.log(match[2]);    // → 01
console.log(match[3]);    // → 01
```

ES9引入了命名捕获组，允许为每一个组匹配指定一个名字，既便于阅读代码，又便于引用。

```
const re = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/;
const match = re.exec('2019-01-01');
console.log(match.groups);          // → {year: "2019", month: "01", day:
"01"}
console.log(match.groups.year);     // → 2019
console.log(match.groups.month);    // → 01
console.log(match.groups.day);      // → 01
```

正则表达式反向断言（lookbehind）

```
let test = 'hello sisteran'
console.log(test.match(/(?<=sisteran\s)hello/))
// ["hello", index: 6, input: "sisteran hello", groups: undefined]
```

正则表达式dotAll模式

正则表达式中，点（.）是一个特殊字符，代表任意的单个字符，但是有两个例外。一个是四个字节的UTF-16 字符，这个可以用u修饰符解决；另一个是行终止符,如换行符(\n)或回车符(\r),这个可以通过ES9的s(dotAll)flag，在原正则表达式基础上添加s表示:

```
console.log(/foo.bar/.test('foo\nbar')) // false
console.log(/foo.bar/s.test('foo\nbar')) // true
```

那如何判断当前正则是否使用了 dotAll 模式呢?

```
const re = /foo.bar/s // Or, `const re = new RegExp('foo.bar', 's');`.
console.log(re.test('foo\nbar')) // true
console.log(re.dotAll) // true
console.log(re.flags) // 's'
```

正则表达式 Unicode 转义

引入了一种新的类的写法\p{...}和\P{...}，允许正则表达式匹配符合 Unicode 某种属性的所有字符

非转义序列的模板字符串

之前，`\u`开始一个 unicode 转义，`\x`开始一个十六进制转义，`\`后跟一个数字开始一个八进制转义。这使得创建特定的字符串变得不可能，例如Windows文件路径`C:\uuu\xxx\111`。

更多细节参考模板字符串。

ES10（ES2019）

- 新增了Array的 `flat()` 方法和 `flatMap()` 方法
- 新增了String的 `trimStart()` 方法和 `trimEnd()` 方法
- `Object.fromEntries()`
- `Symbol.prototype.description()`
- `Function.prototype.toString()` 现在返回精确字符，包括空格和注释
- 简化 `try {} catch {}` ，修改 `catch` 绑定

新增了Array的 `flat()` 方法和 `flatMap()` 方法

`flat()` 和 `flatMap()` 本质上就是是归纳（reduce） 与 合并（concat）的操作

```
[1, 2, [3, 4]].flat(Infinity); // [1, 2, 3, 4]

[1, 2, 3, 4].flatMap(a => [a**2]); // [1, 4, 9, 16]
```

新增了String的 `trimStart()` 方法和 `trimEnd()` 方法

分别去除字符串首尾空白字符

`Object.fromEntries()`

> 返回一个给定对象自身可枚举属性的键值对数组

Object.fromEntries() 是 Object.entries() 的反转

- 通过 Object.fromEntries， 可以将 Map 转化为 Object:

```
const map = new Map([ ['foo', 'bar'], ['baz', 42] ]);
const obj = Object.fromEntries(map);
console.log(obj); // { foo: "bar", baz: 42 }
```

- 通过 Object.fromEntries， 可以将 Array 转化为 Object:

```
const arr = [ ['0', 'a'], ['1', 'b'], ['2', 'c'] ];
const obj = Object.fromEntries(arr);
console.log(obj); // { 0: "a", 1: "b", 2: "c" }
```

`Symbol.prototype.description()`

> 只读属性，回 Symbol 对象的可选描述的字符串。

```
Symbol('description').description; // 'description'
```

`Function.prototype.toString()` 现在返回精确字符，包括空格和注释

```
function /* comment */ foo /* another comment */() {}

// 之前不会打印注释部分
console.log(foo.toString()); // function foo(){}

// ES2019 会把注释一同打印
console.log(foo.toString()); // function /* comment */ foo /* another
comment */ (){}

// 箭头函数
const bar /* comment */ = /* another comment */ () => {};

console.log(bar.toString()); // () => {}
```

简化 `try {} catch {}`,修改 `catch` 绑定

```
try {} catch(e) {}
```

现在是

```
try {} catch {}
```

ES11（ES2020）

- `Promise.allSettled()`
- 可选链（Optional chaining）
- 空值合并运算符（Nullish coalescing Operator）
- `import()`
- `globalThis`
- `BigInt`
- `String.prototype.matchAll`

Promise.allSettled

与 `Promise.all` 不同的是，它会返回所有的 promise 结果

```
const promise1 = Promise.resolve('hello')
const promise2 = new Promise((resolve, reject) => setTimeout(reject, 200,
'problem'))

Promise.allSettled([promise1, promise2])
    .then((values) => {
        console.log(values)
    })
```

可选链（Optional chaining）

`可选链` 可让我们在查询具有多层级的对象时，不再需要进行冗余的各种前置校验。

```
var name = user && user.info && user.info.name;
```

又或是这种

```
var age = user && user.info && user.info.getAge && user.info.getAge();
```

很容易命中 `Uncaught TypeError: Cannot read property...`

用了 Optional Chaining，上面代码会变成

```
var name = user?.info?.name;
```

```
var age = user?.info?.getAge?.();
```

空值合并运算符（Nullish coalescing Operator）

设置一个默认的值

```
var level =  user.data.level || '暂无等级';
```

来看看用空值合并运算符如何处理

```
// {
//   "level": 0
// }
var level = user.level ?? '暂无等级'; // level -> 0


// {
//   "an_other_field": 0
// }
var level = user.level ?? '暂无等级'; // level -> '暂无等级'
```

import()

按需加载

globalThis

globalThis 目的就是提供一种标准化方式访问全局对象，有了 globalThis 后，你可以在任意上下文，任意时刻都能获取到全局对象

BigInt

BigInt 是一种内置对象，它提供了一种方法来表示大于 `253 - 1` 的整数。这原本是 Javascript中可以用 `Number` 表示的最大数字。**BigInt** 可以表示任意大的整数

String.prototype.matchAll()

`matchAll()` 方法返回一个包含所有匹配正则表达式及分组捕获结果的迭代器

ES12（ES2021）

- String.prototype.replaceAll()
- Promise.any()
- WeakRef
- 逻辑赋值操作符（Logical Assignment Operators）
- 数字分隔符（Numeric separators）

String.prototype.replaceAll()

> 返回一个全新的字符串，所有符合匹配规则的字符都将被替换掉

```
const str = 'sisteran';
str.replaceAll('s', 'q'); // 'qiqteran'
```

Promise.any()

Promise.any() 接收一个Promise可迭代对象（例如数组），

- 只要其中的一个 promise 成功，就返回那个已经成功的 promise
- 如果可迭代对象中没有一个 promise 成功（即所有的 promises 都失败/拒绝），就返回一个失败的 promise

WeakRef

使用WeakRefs的Class类创建对对象的弱引用(对对象的弱引用是指当该对象应该被GC回收时不会阻止GC的回收行为)

逻辑赋值操作符（Logical Assignment Operators）

```
a ||= b
//等价于
a = a || (a = b)

a &&= b
//等价于
a = a && (a = b)

a ??= b
//等价于
a = a ?? (a = b)
```

数字分隔符（Numeric separators）

```
const money = 1_000_000_000 // 1000000000
```

参考：

- 种草 ES2020 新特性
- ES6、ES7、ES8、ES9、ES10新特性一览
- 盘点ES7、ES8、ES9、ES10新特性

原文

Promise.allSettled 的作用，如何自己实现 Promise.allSettled

引言

本文从四个方面循序渐进介绍 `Promise.allSettled`：

- `Promise.all()` 的缺陷
- 引入 `Promise.allSettled()`
- `Promise.allSettled()` 与 `Promise.all()` 各自的适用场景
- 手写 `Promise.allSettled()` 实现

下面正文开始 👇

Promise.all() 的缺陷

我们在之前的一篇文章中 第 80 题：介绍下 Promise.all 使用、原理实现及错误处理 已经介绍过，当我们使用 `Promise.all()` 执行过个 `promise` 时，只要其中任何一个 `promise` 失败都会执行 `reject`，并且 `reject` 的是第一个抛出的错误信息，只有所有的 `promise` 都 `resolve` 时才会调用 `.then` 中的成功回调

```
const p1 = Promise.resolve(1)
const p2 = Promise.resolve(2)
const p3 = new Promise((resolve, reject) => {
  setTimeout(reject, 1000, 'three');
});

Promise.all([p1, p2, p3])
.then(values => {
    console.log('resolve: ', values)
}).catch(err => {
    console.log('reject: ', err)
})

// reject:  three
```

注意：其中任意一个 `promise` 被 `reject`，`Promise.all` 就会立即被 `reject`，数组中其它未执行完的 `promise` 依然是在执行的，`Promise.all` 没有采取任何措施来取消它们的执行

但大多数场景中，我们期望传入的这组 `promise` 无论执行失败或成功，都能获取每个 `promise` 的执行结果，为此，ES2020 引入了 `Promise.allSettled()`

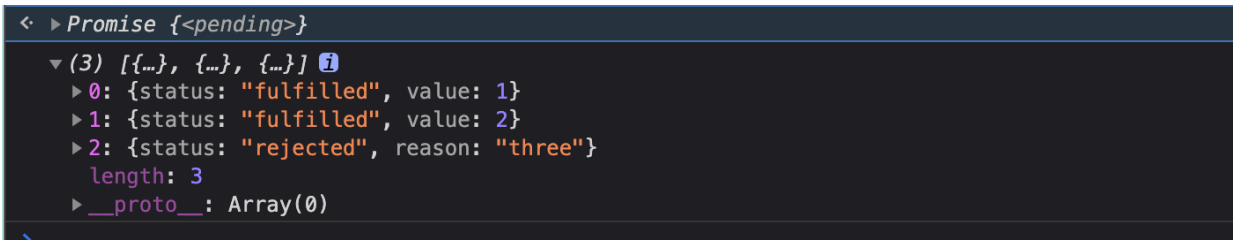
Promise.allSettled()

`Promise.allSettled()` 可以获取数组中每个 `promise` 的结果，无论成功或失败

```
const p1 = Promise.resolve(1)
const p2 = Promise.resolve(2)
const p3 = new Promise((resolve, reject) => {
  setTimeout(reject, 1000, 'three');
});

Promise.allSettled([p1, p2, p3])
.then(values => {
    console.log(values)
})

/*
[
  {status: "fulfilled", value: 1},
  {status: "fulfilled", value: 2},
  {status: "rejected", reason: "three"}
]
*/
```

当浏览器不支持 `Promise.allSettled` ，可以如此 polyfill：

```
if (!Promise.allSettled) {
  const rejectHandler = reason => ({status: "rejected", reason})
  const resolveHandler = value => ({status: "fulfilled", value})
  Promise.allSettled = promises =>
    Promise.all(
      promises.map((promise) =>
        Promise.resolve(promise)
          .then(resolveHandler, rejectHandler)
      )
      // 每个 promise 需要用 Promise.resolve 包裹下
      // 以防传递非 promise
    );
}

// 使用
const p1 = Promise.resolve(1)
const p2 = Promise.resolve(2)
const p3 = new Promise((resolve, reject) => {
  setTimeout(reject, 1000, 'three');
})
const promises = [p1, p2, p3]
Promise.allSettled(promises).then(console.log)
```



Promise.allSettled() 与 Promise.all() 各自的适用场景

`Promise.allSettled()` 更适合：

- 彼此不依赖，其中任何一个被 `reject` ，对其它都没有影响
- 期望知道每个 `promise` 的执行结果

`Promise.all()` 更适合：

- 彼此相互依赖，其中任何一个被 `reject` ，其它都失去了实际价值

手写 Promise.allSettled 源码

与 `Promise.all` 不同的是，当 `promise` 被 `reject` 之后，我们不会直接 `reject` ，而是记录下该 `reject` 的值和对应的状态 `'rejected'` ；

同样地，当 `promise` 对象被 `resolve` 时我们也不仅仅局限于记录值，同时也会记录状态 `'fulfilled'` 。

当所有的 `promise` 对象都已执行（解决或拒绝），我们统一 `resolve` 所有的 `promise` 执行结果数组

```
MyPromise.allSettled = function (promises) {
    return new MyPromise((resolve, reject) => {
      promises = Array.isArray(promises) ? promises : []
      let len = promises.length
      const argslen = len
      // 如果传入的是一个空数组，那么就直接返回一个resolved的空数组promise对象
      if (len === 0) return resolve([])
      // 将传入的参数转化为数组，赋给args变量
      let args = Array.prototype.slice.call(promises)
      // 计算当前是否所有的 promise 执行完成，执行完毕则resolve
      const compute = () => {
        if(--len === 0) {
          resolve(args)
        }
      }
      function resolvePromise(index, value) {
        // 判断传入的是否是 promise 类型
        if(value instanceof MyPromise) {
          const then = value.then
          then.call(value, function(val) {
            args[index] = { status: 'fulfilled', value: val}
```

```
            compute()
          }, function(e) {
            args[index] = { status: 'rejected', reason: e }
            compute()
          })
        } else {
          args[index] = { status: 'fulfilled', value: value}
          compute()
        }
      }

      for(let i = 0; i < argslen; i++){
        resolvePromise(i, args[i])
      }
    })
  }
```

总结

彼此相互依赖，一个失败全部失效（全无或全有）用 `Promise.all`；相互独立，获取每个结果用 `Promise.allSettled`

原文

Promise.any 的作用，如何自己实现 Promise.any

引言

本文从五个方面介绍 `Promise.any`：

- `Promise.any` 的作用
- `Promise.any` 应用场景
- `Promise.any` vs `Promise.all`
- `Promise.any` vs `Promise.race`
- 手写 `Promise.any` 实现

下面正文开始👇

Promise.any

`Promise.any()` 是 ES2021 新增的特性，它接收一个 `Promise` 可迭代对象（例如数组），

- 只要其中的一个 `promise` 成功，就返回那个已经成功的 `promise`
- 如果可迭代对象中没有一个 `promise` 成功（即所有的 `promises` 都失败/拒绝），就返回一个失败的 `promise` 和 `AggregateError` 类型的实例，它是 `Error` 的一个子类，用于把单一的错误集合在一起

```
const promises = [
  Promise.reject('ERROR A'),
  Promise.reject('ERROR B'),
  Promise.resolve('result'),
]

Promise.any(promises).then((value) => {
  console.log('value: ', value)
}).catch((err) => {
  console.log('err: ', err)
})

// value:  result
```

如果所有传入的 `promises` 都失败：

```
const promises = [
  Promise.reject('ERROR A'),
  Promise.reject('ERROR B'),
  Promise.reject('ERROR C'),
]
```

```
Promise.any(promises).then((value) => {
  console.log('value: ', value)
}).catch((err) => {
  console.log('err: ', err)
  console.log(err.message)
  console.log(err.name)
  console.log(err.errors)
})

// err:  AggregateError: All promises were rejected
// All promises were rejected
// AggregateError
// ["ERROR A", "ERROR B", "ERROR C"]
```

Promise.any 应用场景

- 从最快的服务器检索资源

  来自世界各地的用户访问网站，如果你有多台服务器，则尽量使用响应速度最快的服务器，在这种情况下，可以使用 `Promise.any()` 方法从最快的服务器接收响应

  ```
  function getUser(endpoint) {
    return fetch(`https://superfire.${endpoint}.com/users`)
      .then(response => response.json());
  }

  const promises = [getUser("jp"), getUser("uk"), getUser("us"),
  getUser("au"), getUser("in")]

  Promise.any(promises).then(value => {
    console.log(value)
  }).catch(err => {
    console.log(err);
  })
  ```

- 显示第一张已加载的图片（来自MDN）

  在这个例子，我们有一个获取图片并返回 `blob` 的函数，我们使用 `Promise.any()` 来获取一些图片并显示第一张有效的图片（即最先 resolved 的那个 promise）

  ```
  function fetchAndDecode(url) {
    return fetch(url).then(response => {
      if(!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      } else {
  ```

```
      return response.blob();
    }
  })
}

let coffee = fetchAndDecode('coffee.jpg');
let tea = fetchAndDecode('tea.jpg');

Promise.any([coffee, tea]).then(value => {
  let objectURL = URL.createObjectURL(value);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
})
.catch(e => {
  console.log(e.message);
});
```

Promise.any vs Promise.all

`Promise.any()` 和 `Promise.all()` 从返回结果来看，它们 **彼此相反**：

- `Promise.all()`：任意一个 `promise` 被 `reject`，就会立即被 `reject`，并且 `reject` 的是第一个抛出的错误信息，只有所有的 `promise` 都 `resolve` 时才会 `resolve` 所有的结果
- `Promise.any()`：任意一个 `promise` 被 `resolve`，就会立即被 `resolve`，并且 `resolve` 的是第一个正确结果，只有所有的 `promise` 都 `reject` 时才会 `reject` 所有的失败信息

另外，它们又有不同的 **重点**：

- `Promise.all()` 对所有实现都感兴趣。 相反的情况（至少一个拒绝）导致拒绝。
- `Promise.any()` 对第一个实现感兴趣。 相反的情况（所有拒绝）导致拒绝。

Promise.any vs Promise.race

`Promise.any()` 和 `Promise.race()` 的 **关注点** 不一样：

- `Promise.any()`：关注于 `Promise` 是否已经解决
- `Promise.race()`： 主要关注 `Promise` 是否已经解决，无论它是被解决还是被拒绝

手写 Promise.any 实现

`Promise.any` 只要传入的 `promise` 有一个是 `fullfilled` 则立即 `resolve` 出去，否则将所有 `reject` 结果收集起来并返回 `AggregateError`

```
MyPromise.any = function(promises){
  return new Promise((resolve,reject)=>{
```

```
    promises = Array.isArray(promises) ? promises : []
    let len = promises.length
    // 用于收集所有 reject
    let errs = []
    // 如果传入的是一个空数组，那么就直接返回 AggregateError
    if(len === 0) return reject(new AggregateError('All promises were
rejected'))
    promises.forEach((promise)=>{
      promise.then(value=>{
        resolve(value)
      },err=>{
        len--
        errs.push(err)
        if(len === 0){
          reject(new AggregateError(errs))
        }
      })
    })
  })
}
```

原文

Promise.prototype.finally 的作用，如何自己实现 Promise.prototype.finally

`Promise.prototype.finally()` 的作用

`Promise.prototype.finally()` 是 ES2018 新增的特性，它回一个 `Promise` ，在 `promise` 结束时，无论 `Promise` 运行成功还是失败，都会运行 `finally` ，类似于我们常用的 `try {...} catch {...} finally {...}`

`Promise.prototype.finally()` 避免了同样的语句需要在 `then()` 和 `catch()` 中各写一次的情况

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .then(result => console.log(result))
  .finally(() => console.log("Promise end"))

// result
// Promise end
```

`reject` :

```
new Promise((resolve, reject) => {
  throw new Error("error")
})
  .catch(err => console.log(err))
  .finally(() => console.log("Promise end"))

// Error: error
// Promise end
```

注意：

- `finally` 没有参数
- `finally` 会将结果和 error 传递

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .finally(() => console.log("Promise ready"))
  .then(result => console.log(result))

// Promise ready
// result
```

手写一个 `Promise.prototype.finally()`

不管 `Promise` 对象最后状态如何，都会执行的操作

```
MyPromise.prototype.finally = function (cb) {
  return this.then(function (value) {
    return MyPromise.resolve(cb()).then(function () {
      return value
    })
  }, function (err) {
    return MyPromise.resolve(cb()).then(function () {
      throw err
    })
  })
}
```

原文

typeof 可以判断哪些类型？instanceof 做了什么？null为什么被typeof错误的判断为了'object'

一、typeof

`typeof` 操作符唯一的目的就是检查数据类型

	类型	typeof 结果
基本类型	undefined	"undefined"
	Boolean	"boolean"
	Number	"number"
	String	"string"
	BigInt (ECMAScript 2020 新增)	"bigint"
	Symbol	"symbol"
	null	"object"
引用类型	Object（Object、Array、Map、Set等）	"object"
	Function	"function"

所以，但我们使用 `typeof` 来判断**引用类型变量**时，无论是什么类型的变量，它都会返回 `Object` 。

为此，引入了`instanceof`。

二、instanceof

`instanceof` 与 `typeof` 相比，`instanceof`方法要求开发者明确的确认对象为某特定类型。即`instanceof` 用于判断引用类型属于哪个构造函数的方法。

```
var arr = []
arr instanceof Array // true
typeof arr // "object"
// typeof 是无法判断类型是否为数组的
```

`instanceof` 操作符检测过程中也会将继承关系考虑在内，所以`instanceof` 可以在继承关系中用来判断一个实例是否属于它的父类型。

```
// 判断 f 是否是 Foo 类的实例 , 并且是否是其父类型的实例
function Aoo(){}
function Foo(){}
//JavaScript 原型继承
Foo.prototype = new Aoo();

var foo = new Foo();
console.log(foo instanceof Foo) // true
console.log(foo instanceof Aoo) // true
```

`f instanceof Foo` 的判断逻辑是：

- f 的 `__proto__` 一层一层往上，是否对应到 `Foo.prototype`
- 再往上，看是否对应着 `Aoo.prototype`
- 再试着判断 `f instanceof Object`

即 `instanceof` 可以用于判断多层继承关系。

三、instanceof 的内部实现原理

instanceof 的内部实现机制是：通过判断对象的原型链上是否能找到对象的 `prototype`，来确定 `instanceof` 返回值

1. 内部实现原理

```
// instanceof 的内部实现
function instance_of(L, R) {//L 表左表达式，R 表示右表达式，即L为变量，R为类型
  // 取 R 的显示原型
  var prototype = R.prototype
  // 取 L 的隐式原型
  L = L.__proto__
  // 判断对象（L）的类型是否严格等于类型（R）的显式原型
  while (true) {
    if (L === null) {
      return false
    }

    // 这里重点：当 prototype 严格等于 L 时，返回 true
    if (prototype === L) {
      return true
    }

    L = L.__proto__
  }
}
```

`instanceof` 运算符用来检测 `constructor.prototype` 是否存在于参数 `object` 的原型链上。

看下面一个例子，`instanceof` 为什么会返回 `true`？很显然，`an` 并不是通过 `Bottle()` 创建的。

```
function An() {}
function Bottle() {}
An.prototype = Bottle.prototype = {};

let an = new An();
console.log(an instanceof Bottle); // true
```

这是因为 `instanceof` 关心的并不是构造函数，而是原型链。

```
an.__proto__ === An.prototype; // true
An.prototype === Bottle.prototype; // true
// 即
an.__proto__ === Bottle.prototype; // true
```

即有 `an.__proto__ === Bottle.prototype` 成立，所以 `an instanceof Bottle` 返回了 `true`。

所以，按照 `instanceof` 的逻辑，真正决定类型的是 `prototype`，而不是构造函数。

2. Object.prototype.toString（扩展）

还有一个不错的判断类型的方法，就是 `Object.prototype.toString`，我们可以利用这个方法来对一个变量的类型来进行比较准确的判断

默认情况下(不覆盖 `toString` 方法前提下)，任何一个对象调用 `Object` 原生的 `toString` 方法都会返回 `"[object type]"`，其中 `type` 是对象的类型；

```
let obj = {};

console.log(obj); // {}
console.log(obj.toString()); // "[object Object]"
```

[[Class]]

每个实例都有一个 `[[Class]]` 属性，这个属性中就指定了上述字符串中的 `type` (构造函数名)。`[[Class]]` 不能直接地被访问，但通常可以间接地通过在这个值上借用默认的 `Object.prototype.toString.call(..)` 方法调用来展示。

```
Object.prototype.toString.call("abc"); // "[object String]"
Object.prototype.toString.call(100); // "[object Number]"
Object.prototype.toString.call(true); // "[object Boolean]"
Object.prototype.toString.call(null); // "[object Null]"
Object.prototype.toString.call(undefined); // "[object Undefined]"
Object.prototype.toString.call([1,2,3]); // "[object Array]"
Object.prototype.toString.call(/\w/); // "[object RegExp]"
```

可以通过 `Object.prototype.toString.call(..)` 来获取每个对象的类型。

```
function isFunction(value) {
  return Object.prototype.toString.call(value) === "[object Function]"
}
function isDate(value) {
  return Object.prototype.toString.call(value) === "[object Date]"
}
function isRegExp(value) {
  return Object.prototype.toString.call(value) === "[object RegExp]"
}

isDate(new Date()); // true
isRegExp(/\w/); // true
isFunction(function(){}); //true
```

或者可写为：

```
function generator(type){
  return function(value){
    return Object.prototype.toString.call(value) === "[object "+ type +"]"
  }
}

let isFunction = generator('Function')
let isArray = generator('Array');
let isDate = generator('Date');
let isRegExp = generator('RegExp');

isArray([]));    // true
isDate(new Date()); // true
isRegExp(/\w/); // true
isFunction(function(){}); //true
```

Symbol.toStringTag

`Object.prototype.toString` 方法可以使用 `Symbol.toStringTag` 这个特殊的对象属性进行自定义输出。

举例说明：

```
let bottle = {
  [Symbol.toStringTag]: "Bottle"
};

console.log(Object.prototype.toString.call(bottle)); // [object Bottle]
```

大部分和环境相关的对象也有这个属性。以下输出可能因浏览器不同而异：

```
// 环境相关对象和类的 toStringTag：
console.log(window[Symbol.toStringTag]); // Window
console.log(XMLHttpRequest.prototype[Symbol.toStringTag]); //
XMLHttpRequest

console.log(Object.prototype.toString.call(window)); // [object Window]
console.log(Object.prototype.toString.call(new XMLHttpRequest())); //
[object XMLHttpRequest]
```

输出结果和 `Symbol.toStringTag`（前提是这个属性存在）一样，只不过被包裹进了 `[object ...]` 里。

所以，如果希望以字符串的形式获取内置对象类型信息，而不仅仅只是检测类型的话，可以用这个方法来替代 `instanceof`。

3. 总结

	适用于	返回
typeof	基本数据类型	string
instanceof	任意对象	true/false
`Object.prototype.toString`	基本数据类型、内置对象以及包含 `Symbol.toStringTag` 属性的对象	string

`Object.prototype.toString` 基本上就是一增强版 `typeof`。

`instanceof` 在涉及多层类结构的场合中比较实用，这种情况下需要将类的继承关系考虑在内。

四、null为什么被typeof错误的判断为了'object'

```
// JavaScript 诞生以来便如此
typeof null === 'object';
```

在 JavaScript 最初的实现中，JavaScript 中的值是由一个表示类型的标签和实际数据值表示的。对象的类型标签是 0。由于 `null` 代表的是空指针（大多数平台下值为 0x00），因此，null 的类型标签是 0，`typeof null` 也因此返回 `"object"`。（参考来源）

曾有一个 ECMAScript 的修复提案（通过选择性加入的方式），但被拒绝了。该提案会导致 `typeof null === 'null'`。

如果用 `instanceof` 来判断的话：

```
null instanceof null
// Uncaught TypeError: Right-hand side of 'instanceof' is not an object
```

原文

var、let、const 有什么区别

引言

本文主要介绍 `var` 、 `let` 、 `const` 关键字的含义，并从

- 作用域规则
- 重复声明/重复赋值
- 变量提升（hoisted）
- 暂时死区（TDZ）

四个方面对比 `var` 、 `let` 、 `const` 声明的变量差异

var

在 ES6 之前我们都是通过 `var` 关键字定义 JavaScript 变量。ES6 才新增了 `let` 和 `const` 关键字

```
var num = 1
```

在全局作用域下使用 `var` 声明一个变量，默认它是挂载在顶层对象 `window` 对象下（Node 是 `global` ）

```
var num = 1
console.log(window.num) // 1
```

用 `var` 声明的变量的作用域是它当前的执行上下文，可以是函数也可以是全局

```
var x = 1 // 声明在全局作用域下
function foo() {
    var x = 2 // 声明在 foo 函数作用域下
    console.log(x) // 2
}
foo()
console.log(x) // 1
```

如果在 `foo` 没有声明 `x` ，而是赋值，则赋值的是 `foo` 外层作用域下的 `x`

```
var x = 1 // 声明在全局作用域下
function foo() {
    x = 2 // 赋值
    console.log(x) // 2
}
foo()
console.log(x) // 2
```

如果赋值给未声明的变量，该变量会被隐式地创建为全局变量（它将成为顶层对象的属性）

```
a = 2
console.log(window.a) // 2

function foo(){
    b = 3
}
foo()
console.log(window.b) // 3
```

var 缺陷一：所有未声明直接赋值的变量都会自动挂在顶层对象下，造成全局环境变量不可控、混乱

变量提升（hoisted）

使用`var`声明的变量存在变量提升的情况

```
console.log(b) // undefined
var b = 3
```

注意，提升仅仅是变量声明，不会影响其值的初始化，可以与隐式的理解为：

```
var b
console.log(b) // undefined
b = 3
```

作用域规则

`var`声明可以在包含它的函数，模块，命名空间或全局作用域内部任何位置被访问，包含它的代码块对此没有什么影响，所以多次声明同一个变量并不会报错：

```
var x = 1
var x = 2
```

这种作用域规则可能会引发一些错误

```
function sumArr(arrList) {
    var sum = 0;
    for (var i = 0; i < arrList.length; i++) {
        var arr = arrList[i];
        for (var i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
    }

    return sum;
}
```

这里很容易看出一些问题，里层的 `for` 循环会覆盖变量 `i`，因为所有 `i` 都引用相同的函数作用域内的变量。 有经验的开发者们很清楚，这些问题可能在代码审查时漏掉，引发无穷的麻烦。

var 缺陷二：允许多次声明同一变量而不报错，造成代码不容易维护

捕获变量怪异之处

```
var a = [];
for (var i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 10
```

`i` 是全局变量，全局只有一个变量 `i`， `for` 循环结束时， `i=10` ，所以 `a[6]()` 也为 10 ，并且 `a` 的所有元素里面的 `i` 都为 10

而我们期望的是 `a[6]()` 输出 6，所以我们有了下面的块级作用域

let

`let` 与 `var` 的写法一致，不同的是它使用的是块作用域

```
let a = 1
```

块作用域变量在包含它们的块或 `for` 循环之外是不能访问的

```
{
    let x = 1
}
console.log(x) // Uncaught ReferenceError: x is not defined
```

所以：

```
var a = [];
for (let i = 0; i < 10; i++) { // 每一次循环的 i 其实都是一个新的变量
  a[i] = function () {
    console.log(i);
  };
} // JavaScript 引擎内部会记住上一轮循环的值，初始化本轮的变量i时，就在上一轮循环的基础上进行计算
a[6](); // 6
```

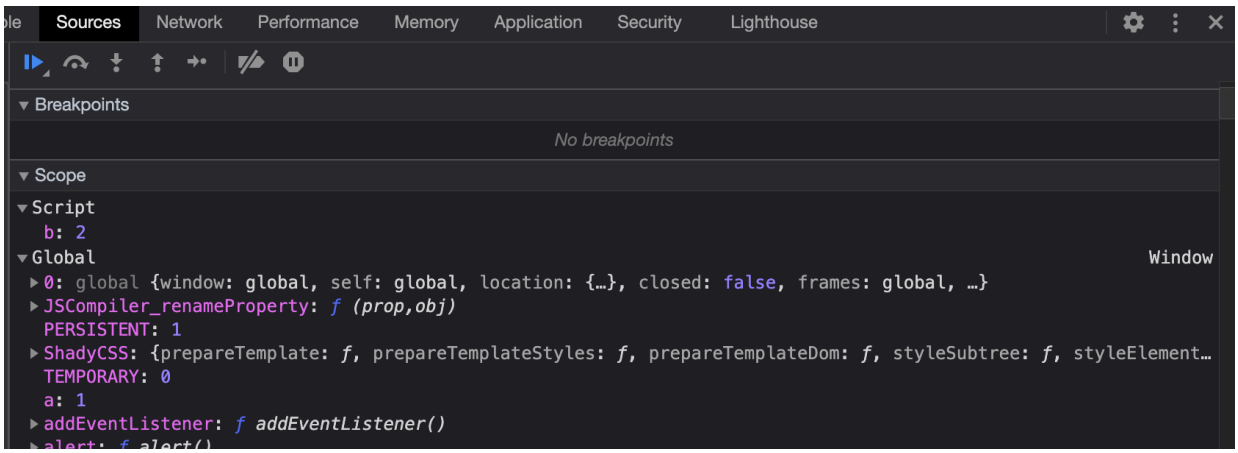
同时，`let` 解决了 `var` 的两个缺陷：

使用 let 在全局作用域下声明的变量也不是顶层对象的属性

```
let b = 2
window.b // undefined
```

那它在哪里嘀?

```
var a = 1
let b = 2
debugger
```



通过上图也可以看到，在全局作用域中，用 let 和 const 声明的全局变量没有在全局对象中，只是一个块级作用域（Script）中

不允许同一块中重复声明

```
let x = 1
let x = 2
// Uncaught SyntaxError: Identifier 'x' has already been declared
```

如果在不同块中是可以声明的

```
{
    let x = 1
    {
        let x = 2
    }
}
```

这种在一个嵌套作用域中声明同一个变量名称的行为称做 **屏蔽**，它可以完美解决上面的 `sumArr` 问题：

```
function sumArr(arrList) {
    let sum = 0;
    for (let i = 0; i < arrList.length; i++) {
        var arr = arrList[i];
        for (let i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
    }

    return sum;
}
```

此时将得到正确的结果，因为内层循环的 `i` 可以屏蔽掉外层循环的 `i`

通常来讲应该避免使用屏蔽，因为我们需要写出清晰的代码。 同时也有些场景适合利用它，你需要好好打算一下

暂时性死区（TDZ）

指 `let` 声明的变量在被声明之前不能被访问

```
console.log(x) // Uncaught ReferenceError: x is not defined
let x = 1
```

如果你在块中声明 `let`，它会报以下错误：

```
// let
{
    console.log(x) // Uncaught ReferenceError: Cannot access 'x' before
initialization
    let x = 2
}
```

即，在块级作用域下报错内容是未初始化，那 `let` 在块级作用域下有没有被提升喃?

TC39 的成员 **Rick Waldron** 在 [hoisting-vs-tdz.md] 中这么说：

> In JavaScript, all binding declarations are instantiated when control flow enters the scope in which they appear. Legacy var and function declarations allow access to those bindings before the actual declaration, with a "value" of `undefined`. That legacy behavior is known as "hoisting". let and const binding declarations are also instantiated when control flow enters the scope in which they appear, with access prevented until the actual declaration is reached; this is called the Temporal Dead Zone. The TDZ exists to prevent the sort of bugs that legacy hoisting can create.

翻译：

> 在 JavaScript 中，当控制流进入它们出现的范围时，所有绑定声明都会被实例化。 传统的 `var` 和 `function` 声明允许在实际声明之前访问那些绑定，并且其值（value）为 `undefined` 。 这种遗留行为被称为变量提升（hoisting）。 当控制流进入它们出现的范围时，`let` 和 `const` 声明也会被实例化，但在运行到实际声明之前禁止访问。 这称为暂时性死区（Temporal Dead Zone）。 TDZ 的存在是为了防止传统提升可能造成的那种错误。

即，通过 `let` 、 `const` 变量始终“存在”于它们的作用域里，但不能在 `let` 、 `const` 语句之前访问它们，所以不能称为变量提升，只能称为暂时性死区

const

`const` 声明一个只读的常量。一旦声明，常量的值就不能改变。

```
const a = 1

a = 2 // Uncaught TypeError: Assignment to constant variable.
```

因此， `const` 声明的变量不得改变值，这意味着， `const` 一旦声明变量，就必须立即初始化，不能留到以后赋值。

```
const s // 声明未赋值
// Uncaught SyntaxError: Missing initializer in const declaration
```

注意，这里 `const` 保证的不是变量的值不得改动，而是变量指向的那个内存地址不得改动，如果是基本类型的话，变量的值就保存在那个内存地址上，也就是常亮，如果是引用类型，它内部的值是可以变更的

```
const num = 1
const user = {
    name: "sisterAn",
    age: num,
}

user = {
    name: "pingzi",
    age: num
} // Uncaught TypeError: Assignment to constant variable.
```

```
// 下面这些都是运行成功的
user.name = "Hello"
user.name = "Kitty"
user.name = "Cat"
user.age--
```

其它 `const` 与 `let` 相同，例如：

- 作用域相同，只在声明所在的块级作用域内有效
- 常量也是不提升，同样存在暂时性死区

这里不再赘述

var vs let vs const

`var` 、 `let` 、 `const` 的不同主要有以下几个方面：

- 作用域规则
- 重复声明/重复赋值
- 变量提升（hoisted）
- 暂时死区（TDZ）

作用域规则

`let/const` 声明的变量属于块作用域，只能在其块或子块中可用。而 `var` 声明的变量的作用域是是全局或者整个封闭函数

重复声明/重复赋值

- `var` 可以重复声明和重复赋值
- `let` 仅允许重复赋值，但不能重复声明
- `const` 既不可以重复赋值，但不能重复声明

变量提升（hoisted）

`var` 声明的变量存在变量提升，即可以在变量声明前访问变量，值为 `undefined`

`let` 和 `const` 不存在变量提升，即它们所声明的变量一定要在声明后使用，否则报错 `ReferenceError`

var：

```
console.log(a) // undefined
var a = 1
```

let：

```
console.log(b) // Uncaught ReferenceError: b is not defined
let b = 2
```

const：

```
console.log(c) // Uncaught ReferenceError: c is not defined
let c = 3
```

暂时死区（TDZ）

`var`不存在暂时性死区， `let`和`const`存在暂时性死区，只有变量声明后，才能被访问或使用

编程风格

ES6 提出了两个新的声明变量的命令： `let` 和 `const` 。其中， `let` 完全可以取代 `var` ，因为两者语义相同，而且 `let` 没有副作用。所以，我们在开发中建议使用 `let` 、 `const` ，不使用 `var`

参考

- MDN
- 《阮一峰：ECMAScript 6入门》

原文

WeakMap 和 Map 的区别，WeakMap 原理，为什么能被 GC?

垃圾回收机制（GC）

我们知道，程序运行中会有一些垃圾数据不再使用，需要及时释放出去，如果我们没有及时释放，这就是内存泄露

JS 中的垃圾数据都是由垃圾回收（Garbage Collection，缩写为 GC）器自动回收的，不需要手动释放，它是如何做的喃?

很简单，JS 引擎中有一个后台进程称为垃圾回收器，它监视所有对象，观察对象是否可被访问，然后按照固定的时间间隔周期性的删除掉那些不可访问的对象即可

现在各大浏览器通常用采用的垃圾回收有两种方法：

- 引用计数
- 标记清除

引用计数

最早最简单的垃圾回收机制，就是给一个占用物理空间的对象附加一个引用计数器，当有其它对象引用这个对象时，这个对象的引用计数加一，反之解除时就减一，当该对象引用计数为 0 时就会被回收。

该方式很简单，但会引起内存泄漏：

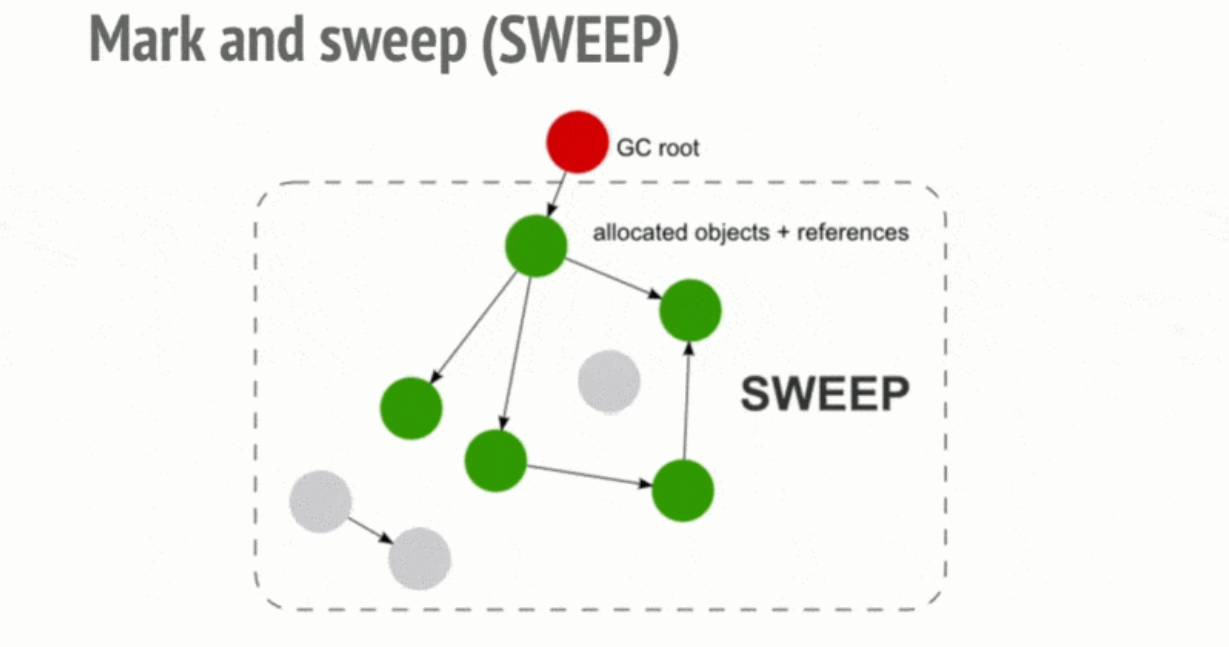
```
// 循环引用的问题
function temp(){
    var a={};
    var b={};
    a.o = b;
    b.o = a;
}
```

这种情况下每次调用 `temp` 函数，`a` 和 `b` 的引用计数都是 `2`，会使这部分内存永远不会被释放，即内存泄漏。现在已经很少使用了，只有低版本的 IE 使用这种方式。

标记清除

V8 中主垃圾回收器就采用标记清除法进行垃圾回收。主要流程如下：

- 标记：遍历调用栈，看老生代区域堆中的对象是否被引用，被引用的对象标记为活动对象，没有被引用的对象（待清理）标记为垃圾数据。
- 垃圾清理：将所有垃圾数据清理掉



（图片来源：How JavaScript works: memory management + how to handle 4 common memory leaks）

在我们的开发过程中，如果我们想要让垃圾回收器回收某一对象，就将对象的引用直接设置为 `null`

```
var a = {}; // {} 可访问，a 是其引用

a = null; // 引用设置为 null
// {} 将会被从内存里清理出去
```

但如果一个对象被多次引用时，例如作为另一对象的键、值或子元素时，将该对象引用设置为 `null` 时，该对象是不会被回收的，依然存在

```
var a = {};
var arr = [a];

a = null;
console.log(arr)
// [{}]
```

如果作为 `Map` 的键喃?

```
var a = {};
var map = new Map();
map.set(a, '三分钟学前端')

a = null;
console.log(map.keys()) // MapIterator {{}}
console.log(map.values()) // MapIterator {"三分钟学前端"}
```

如果想让 a 置为 `null` 时，该对象被回收，该怎么做喃?

WeakMap vs Map

ES6 考虑到了这一点，推出了： `WeakMap` 。它对于值的引用都是不计入垃圾回收机制的，所以名字里面才会有一个"Weak"，表示这是弱引用（对对象的弱引用是指当该对象应该被GC回收时不会阻止GC的回收行为）。

`Map` 相对于 `WeakMap` ：

- `Map` 的键可以是任意类型， `WeakMap` 只接受对象作为键（null除外），不接受其他类型的值作为键
- `Map` 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键； `WeakMap` 的键是弱引用，键所指向的对象可以被垃圾回收，此时键是无效的
- `Map` 可以被遍历， `WeakMap` 不能被遍历

下面以 `WeakMap` 为例，看看它是怎么上面问题的：

```
var a = {};
var map = new WeakMap();
map.set(a, '三分钟学前端')
map.get(a)

a = null;
```

上例并不能看出什么？我们通过 `process.memoryUsage` 测试一下：

```
//map.js
global.gc(); // 0 每次查询内存都先执行gc()再memoryUsage()，是为了确保垃圾回收，保证获取的内存使用状态准确

function usedSize() {
    const used = process.memoryUsage().heapUsed;
    return Math.round((used / 1024 / 1024) * 100) / 100 + "M";
}

console.log(usedSize()); // 1 初始状态，执行gc()和memoryUsage()以后，heapUsed值为 1.64M

var map = new Map();
var b = new Array(5 * 1024 * 1024);

map.set(b, 1);

global.gc();
console.log(usedSize()); // 2 在 Map 中加入元素b，为一个 5*1024*1024 的数组后，heapUsed为41.82M左右
```

```
b = null;
global.gc();

console.log(usedSize()); // 3 将b置为空以后，heapUsed 仍为41.82M，说明Map中的那个长度为5*1024*1024的数组依然存在
```

执行 `node --expose-gc map.js` 命令：

```
lilunahaijiaodeMacBook-Pro:yiti-mini-web lilunahaijiao$ node --expose-gc map.js
1.64M
41.82M
41.82M
```

其中，`--expose-gc` 参数表示允许手动执行垃圾回收机制

```
// weakmap.js
function usedSize() {
    const used = process.memoryUsage().heapUsed;
    return Math.round((used / 1024 / 1024) * 100) / 100 + "M";
}

global.gc(); // 0 每次查询内存都先执行gc()再memoryUsage()，是为了确保垃圾回收，保证获取的内存使用状态准确
console.log(usedSize()); // 1 初始状态，执行gc()和 memoryUsage()以后，heapUsed值为 1.64M
var map = new WeakMap();
var b = new Array(5 * 1024 * 1024);

map.set(b, 1);

global.gc();
console.log(usedSize()); // 2 在 Map 中加入元素b，为一个 5*1024*1024 的数组后，heapUsed为41.82M左右

b = null;
global.gc();

console.log(usedSize()); // 3 将b置为空以后，heapUsed 变成了1.82M左右，说明WeakMap中的那个长度为5*1024*1024的数组被销毁了
```

执行 `node --expose-gc weakmap.js` 命令：

```
lilunahaijiaodeMacBook-Pro:yiti-mini-web lilunahaijiao$ node --expose-gc weakmap.js
1.64M
41.82M
1.82M
```

上面代码中，只要外部的引用消失，WeakMap 内部的引用，就会自动被垃圾回收清除。由此可见，有了它的帮助，解决内存泄漏就会简单很多。

最后看一下 `WeakMap`

WeakMap

WeakMap 对象是一组键值对的集合，其中的**键是弱引用对象，而值可以是任意**。

注意，WeakMap 弱引用的只是键名，而不是键值。键值依然是正常引用。

WeakMap 中，每个键对自己所引用对象的引用都是弱引用，在没有其他引用和该键引用同一对象，这个对象将会被垃圾回收（相应的key则变成无效的），所以，WeakMap 的 key 是不可枚举的。

属性：

- constructor：构造函数

方法：

- has(key)：判断是否有 key 关联对象
- get(key)：返回key关联对象（没有则则返回 undefined）
- set(key)：设置一组key关联对象
- delete(key)：移除 key 的关联对象

```
let myElement = document.getElementById('logo');
let myWeakmap = new WeakMap();

myWeakmap.set(myElement, {timesClicked: 0});

myElement.addEventListener('click', function() {
  let logoData = myWeakmap.get(myElement);
  logoData.timesClicked++;
}, false);
```

除了 `WeakMap` 还有 `WeakSet` 都是弱引用，可以被垃圾回收机制回收，可以用来保存DOM节点，不容易造成内存泄漏

另外还有 ES12 的 `WeakRef` ，感兴趣的可以了解下，今晚太晚了，之后更新

最后

每日三分钟，进阶一个前端小 Tip！从入门到进阶到资深，循序渐进加深你的知识领域！

项目地址：https://github.com/Advanced-Frontend

如果感觉还不错，欢迎分享给好友，你的每次分享都是对我们最好的支持❤️

最近开源了一个github仓库：百问百答，在工作中很难做到对社群问题进行立即解答，所以可以将问题提交至 https://github.com/Advanced-Frontend/Just-Now-QA ，会定期解答，更多的是鼓励与欢迎更多人一起参与探讨与解答🌹