

frontendPrep

JS 进阶

js 进阶

数据类型

基础类型

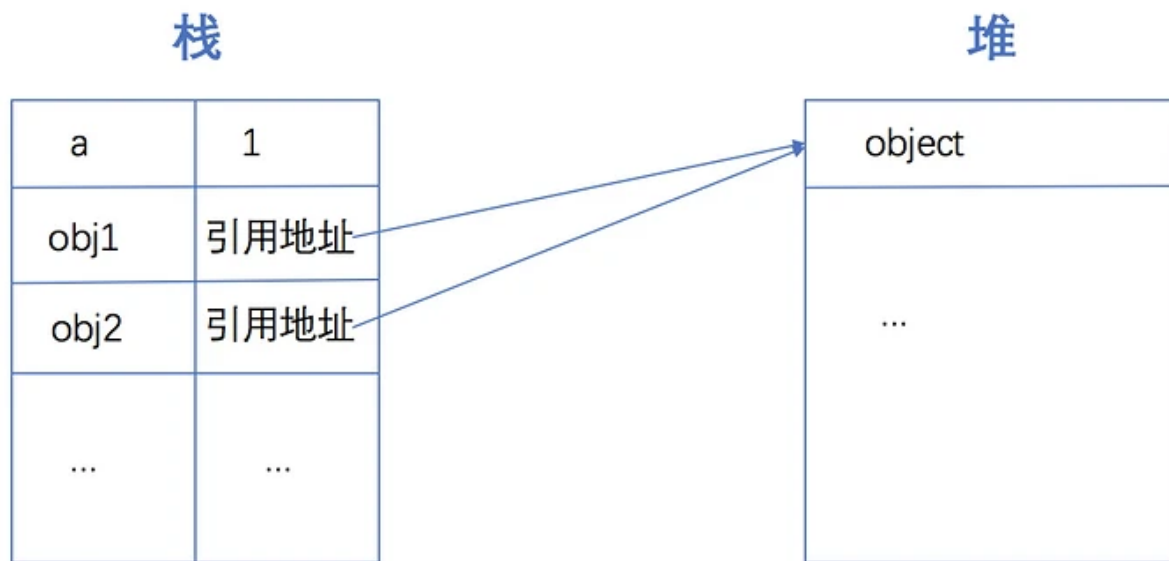
1. undefined
2. Null
3. Symbol
4. String
5. BigInt
6. Boolean
7. Number

引用类型 (Object)

1. Array
2. Function
3. RegExp
4. Date
5. Math

内存的表现方式

基础类型的数据的值保存在栈。引用类型的值保存在堆且返回在堆上的内存地址给栈，即我们常说的引用。上个图帮助理解引用类型在内存的存在方式。



图片来源：<https://segmentfault.com/a/1190000015830451>

引用类型在栈只是保存堆上的内存地址而不是值，如图当把一个引用类型对象赋值给另外一个引用类型对象，只是让这2个对象有相同的内存地址。因为有相同的内存地址，所以当任意一个对象修改了堆内存中的值，那么另一个对象在堆内存中的值则会跟着改变。

```
1 let a = {  
2   hello: 'abc'  
3 }  
4  
5 let b = a  
6 b.hello = 'def'  
7  
8 console.log(a)  
9 console.log(b)
```

那么我们有时候确实是希望能深度拷贝一个对象，即两个对象不仅在堆内存中的值相同且分别有自己的内存地址空间。与上面相反，改变一个对象的在堆内存中的值，不会影响另外一个对象。

我们来看看如何实现一个对象深度拷贝

对象深度拷贝

我们已经了解了不同数据类型在内存中的存在方式，承接上一节的内容，我们完成一个对象的深度拷贝方法。

开始编码之前，我们梳理一下需要完成什么才是一个合格的深度拷贝和编码逻辑

- 当对象的属性值是基本类型的值则直接通过"="赋值
- 当对象的属性值是Function引用类型, 则直接通过"="赋值
- 当对象的属性值是RegExp / Date 等引用类型，直接new一个新对象然后把新对象返回
- 除了上述的3种，一般对象属性的值使用递归完成。使用递归为了解决对象里面有对象的内嵌情况。这里也可以清晰的知道，递归的结束条件就是上述3种条件+1种比较特殊的条件（对象自引用，即 `a.property = a`）
- 对象的属性值与对象可能存在循环引用，即 `a.property = a`, 如果不处理, 当使用递归编码会出现无限递归。这里需要使用一种机制可以让我们检测循环引用，所以我们使用了map数据结构作为容器，当前对象作为key。那么每次开始递归之前，首先判断是否即将被递归的当前对象是否已经在map里面存在。如果存在，则直接返回map里面的，即避免无限递归。但是这样会有一个缺点，即map里面的key保存的是对象，即使我们消除了对某个key的引用且手动释放，可能gc依然不会回收map中的对象。所以我们需要找到一个map替代的方法，即使用weakmap
- weakmap 和 map 基本相同。weakmap 保存的是对象的弱引用，即当没有对该对象有任何引用gc可以在合适的时候回收weakmap里面的对象, 释放内存。

演示代码

```
1  const deepClone = function (obj, container = new WeakMap()) {
2    if (obj.constructor === Date) return new Date(obj);
3    if (obj.constructor === RegExp) return new RegExp(obj);
4    if (container.has(obj)) return container.get(obj);
5
6    const objDesc = Object.getOwnPropertyDescriptors(obj);
7    const objCloned = Object.create(Object.getPrototypeOf(obj), objDesc);
8    container.set(obj, objCloned);
9
10   for (const key of Reflect.ownKeys(obj)) {
11     objCloned[key] =
12       typeof obj[key] === 'object' &&
13       typeof obj[key] !== 'function' &&
14       obj[key] !== null
```

```

15         ? deepClone(obj[key], container)
16         : obj[key];
17     }
18     return objCloned;
19 };
20
21 // test
22 const obj = {
23     _number: 100,
24     _string: 'hello world',
25     _undefined: undefined,
26     _null: null,
27     _bool: true,
28     _obj: {
29         title: 'clone object deeply',
30         isTested: true,
31     },
32     _arr: [0],
33     _func: function (a = 0) {
34         console.log(arguments);
35     },
36     _date: new Date(),
37     _reg: new RegExp('/test reg obj/'),
38     [Symbol('any')]: 123,
39 };
40 Object.defineProperty(obj, '_inumerableTest', {
41     enumerable: false,
42     value: 'notEnumerableVal',
43 });
44 obj.selfReference = obj;
45 console.log('original obj', obj);
46 console.log('cloned obj', deepClone(obj));

```

我们提到了使用weakmap为了配合gc在合适的时候自动释放内存，让程序能够表现的更好。那我们如何平时判断javascript项目是否存在内存泄漏和常见的内存泄漏的原因和解决方法么？

内存自动回收与手动管理

承接上一节，我们实现了对对象的深度拷贝。使用了weakmap来优化我们的代码，配合gc释放内存。

那是否map真的无法配合gc释放内存，我们用代码自测一下。

```
1  global.gc();
2
3  function memUsed() {
4      const usedMem = process.memoryUsage().heapUsed;
5      return Math.round((usedMem / 1024 / 1024) * 100) / 100 + 'M';
6  }
7  console.log(memUsed());
8  var arr = new Array(1000000);
9  var map = new Map();
10 map.set(arr, 1);
11 global.gc();
12 console.log(memUsed());
13
14 arr = null;
15 global.gc();
16 console.log(memUsed());
```

```
→ test node --expose-gc memory.js
2.75M
10.36M
10.36M
```

如果使用weakmap呢？把new Map() -> new WeakMap(), 自己用代码来自测吧~



所有使用gc方法的代码需要在服务端环境运行, 下面不再提示

javascript 是一门高级语言，所以很多底层的交互api没有提供，比如内存分配与内存回收。与其花大量时间了解gc的回收算法，不如我们了解一下在平时做项目中，最有可能产生内存泄漏的可能

1. 没有释放闭包
2. 没有消除定时器等回调
3. 全局变量
4. 还有很多

那让我们来演示一下闭包的内存泄漏和解决方法

```
1  global.gc();
2
3  console.log(memUsed());
4  let closure = testClosure();
5  closure();
6  global.gc();
7  console.log(memUsed());
8  closure();
9  global.gc();
10 console.log(memUsed());
11 closure();
12 global.gc();
13 console.log(memUsed());
14 closure();
15 global.gc();
16 console.log(memUsed());
17
18 closure = null;
19 global.gc();
20 console.log(memUsed());
21
22 function memUsed() {
23   const usedMem = process.memoryUsage().heapUsed;
24   return Math.round((usedMem / 1024 / 1024) * 100) / 100;
25 }
26 function testClosure() {
27   const hooked = new Array(10);
28   return function () {
29     hooked.push(...new Array(100000));
30   };
31 }
32
```

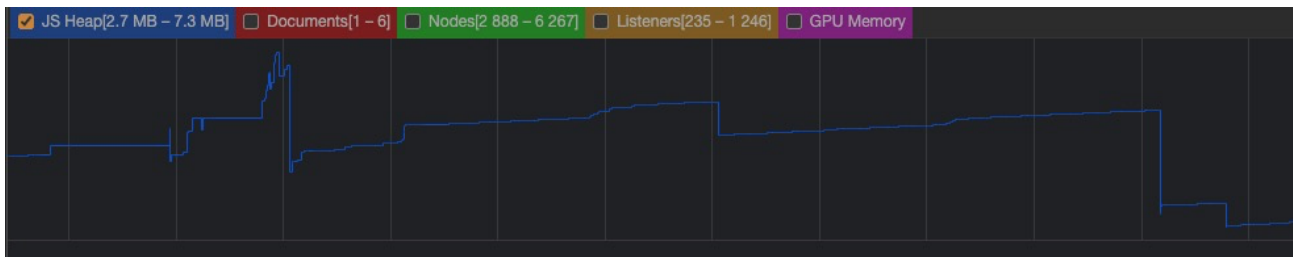
```
→ test node --expose-gc closure.js
当前内存: 2.75
当前内存: 3.86
当前内存: 4.42
当前内存: 5.27
当前内存: 6.53
当前内存: 2.74
```

闭包的内存演示和释放

正如上面的演示，内存会不断的提升。如果是生产环境恰好数据量再大点，那么会存在内存泄漏的可能。gc在这里是无法自动回收的因为闭包的作用域关系，hooked变量一直有引用，即使当前函数运行完也无法释放函数使用的内存。那么请自己动手试试如何自测其他情况的内存泄漏与解决方法。

因为内存泄漏有时候自己也不知道，但是有迹可循，比如，网站刚打开非常顺滑，当用户一直没有关闭网站而网站随着时间的长度运行速度越来越卡。

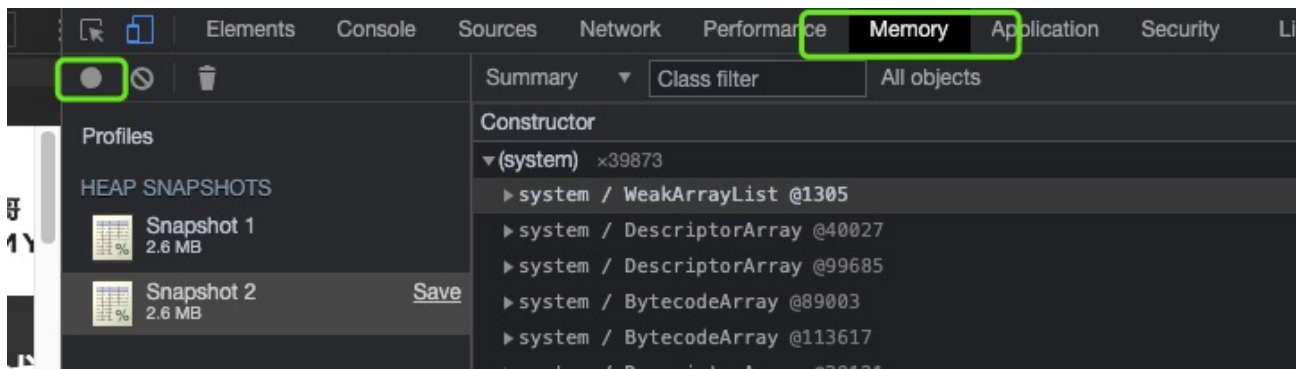
首先，我们可以使用的工具是chrome devtools 的performance 标签，然后找到如下图的位置



javascript 堆内存分配图

图片的内存分配图如何理解呢，其实非常简单。上图的图中内存的分配有高有低，这是非常好的，因为使用了内存，gc释放了内存。如果网站一直都处于上升没有降低，或降低幅度对比上升幅度小很多很多，那么网站很有可能内存存在泄漏。那么如何判断是哪里呢？一般我们会先从在堆内存中占有最大内存尺寸的对象开始。

打开chrome devtools 的memory 标签内的 heap snapshot, 然后拍一下当前的heap的快照用来定位。如图



heap snapshot 的位置

搭配我们用代码来自测

ⓘ 注意！运行下面的代码可能会让机器卡住，测试的代码已经收敛了。可以根据你的机器实际情况来更改下面代码。

```
1  function test() {
2    let arr = new Array(10000000);
3    return function () {
4      return arr.push(...new Array(100000));
5    };
6  }
7  let t = test();
8  t();
9  t();
10 t();
11 t();
12 t();
13
```

The screenshot shows the Chrome DevTools Memory tab. On the left, there are three heap snapshots: Snapshot 1 (40.9 MB), Snapshot 2 (41.0 MB), and Snapshot 3 (41.0 MB). The main panel displays a list of objects under the 'Constructor' column, with columns for 'Distance', 'Shallow Size', and 'Retained Size'. The first object is an array of 168 elements, with a shallow size of 40,235,276 bytes and a retained size of 40,414,456 bytes. Below this, a list of internal arrays and object properties is shown. The 'Retainers' section at the bottom shows the call stack leading to the selected object, including 'varr' in 'system / Context' and 'test' in 'Window / 127.0.0.1:5500'.

Constructor	Distance	Shallow Size	Retained Size
(array) x168	2	40 235 276 98 %	40 414 456 99 %
(object elements)[] @99415	7	40 000 008 98 %	40 000 008 98 %
(internal array)[] @81003	-	131 080 0 %	131 080 0 %
(object properties)[] @92589	2	8 220 0 %	45 692 0 %
(internal array)[] @90935	3	4 104 0 %	39 736 0 %
(internal array)[] @29101	-	3 412 0 %	36 340 0 %
(object properties)[] @97123	6	24 604 0 %	24 604 0 %
(internal array)[] @37353	-	2 236 0 %	23 836 0 %
(internal array)[] @42183	-	2 236 0 %	20 668 0 %
(internal array)[] @40657	-	2 236 0 %	20 380 0 %
(internal array)[] @37367	-	2 236 0 %	19 900 0 %
(internal array)[] @91141	3	4 116 0 %	11 884 0 %
(internal array)[] @15793	6	3 172 0 %	6 504 0 %
(internal array)[] @23697	-	3 172 0 %	6 504 0 %
(object properties)[] @92017	5	6 172 0 %	6 204 0 %
(internal array)[] @15791	6	3 532 0 %	4 396 0 %
(internal array)[] @23695	-	3 532 0 %	4 396 0 %
(internal array)[] @48783	3	4 116 0 %	4 116 0 %
(internal array)[] @48577	3	4 104 0 %	4 104 0 %
(internal array)[] @21197	9	3 412 0 %	4 084 0 %
(internal array)[] @46517	-	928 0 %	3 904 0 %
(internal array)[] @1315	-	24 0 %	1 804 0 %
(object properties)[] @51259	2	540 0 %	1 780 0 %
(object properties)[] @91521	6	1 564 0 %	1 564 0 %
(internal array)[] @48127	-	12 0 %	1 296 0 %
(internal array)[] @1255	-	1 024 0 %	1 192 0 %
(internal array)[] @1249	-	1 032 0 %	1 032 0 %
(internal array)[] @1251	-	1 032 0 %	1 032 0 %
(internal array)[] @1253	-	1 032 0 %	1 032 0 %
(internal array)[] @23343	9	928 0 %	928 0 %
(object properties)[] @90409	6	796 0 %	796 0 %
(object properties)[] @91459	5	796 0 %	796 0 %
(internal array)[] @48207	-	136 0 %	740 0 %
(internal array)[] @48715	3	520 0 %	520 0 %

Object	Distance	Shallow Size	Retained Size
elements in Array @99415	6	16 0 %	40 000 024 98 %
varr in system / Context @98979	5	20 0 %	40 000 044 98 %
context in () @98973	4	32 0 %	40 000 172 98 %
vt in system / Context @98889	3	24 0 %	40 000 196 98 %
context in test() @92291	2	32 0 %	108 0 %
test in Window / 127.0.0.1:5500 @90245	1	36 0 %	40 370 396 98 %
value in system / PropertyCell @93501	3	20 0 %	20 0 %
2 in (internal array)[] @91123	3	24 0 %	44 0 %
previous in system / Context @98979	5	20 0 %	40 000 044 98 %

heap snap

一目了然，我们可以找到在堆内存中占有最大几个尺寸的对象然后进行分析

测试的代码基本使用了闭包的作用域和保存的特点，让内存无法通过gc自动回收。那闭包底层原理到底是什么呢？下一节，我们来研究一下闭包与作用域

扩展

当然如果你是高手有充足的时间，可以上网查gc的算法实现，比如 Scavenge

闭包的原理和实际应用

往往闭包的知识点运用到的一个更基本的知识点，就是作用域。那我们先来了解下作用域。

作用域字面的意思已经非常清晰了，就是作用的范围，作用的范围在javascript表达可以访问的范围。

javascript作用域分为3种，即全局作用域，函数作用域，块级作用域。

全局作用域

在javascript里面，全局作用域是指在window对象的变量，且可以在项目的任何位置都可以访问。代码自测：

```
1  function globalScope() {  
2      this.nameTest = 'hello world'  
3      this.titleTest = 'hello javascript'  
4  }  
5  globalScope()  
6  
7  console.log(name) // hello world  
8  console.log(title) // hello javascript
```

上面的代码如果你不理解name 和 title 2个变量为什么会是全局变量，那么也说明了平时代码你已经可能不知不觉的有内存泄漏。回想上一节我们提到的内存泄漏最有可能的几种场景其中就有非意图的设置全局变量。

原理底层我们分析一下。当函数执行的时候，访问nameTest，会发现当前函数作用域没有nameTest变量，那么默认的会往当前作用域的上一级寻找，即globalScope 的上一级就是window，我们标记 globalScope->window，即作用域链。window 上面也无法找到nameTest, 那么默认的会往window上面挂载这个变量，即 window.nameTest = 'hello world'。titleTest 也是如此，即window.titleTest = 'hello javascript'。那么当我们在globalScope 外面访问nameTest 和 titleTest当然就可以访问了，即如我们上面所说，全局变量在项目的任何地方都可访问, 全局变量有全局作用域。

 作用域的查找方式是从内向外的，即不是从外向内的查找

那么上面的代码我们应该如何修改让nameTest 和 titleTest的作用域限定在globalScope函数作用域内呢？

函数作用域

现在我们来修改代码，让nameTest和titleTest变成函数作用域

```
1  function globalScope() {
2      var nameTest = 'hello world'
3      var titleTest = 'hello javascript'
4      console.log(nameTest) // hello world
5      console.log(titleTest) // hello javascript
6  }
7  globalScope()
8
9  console.log(nameTest) // nameTest is not defined
10 console.log(titleTest) // titleTest is not defined
```

好了，现在的nameTest 和 titleTest 是函数作用域，即在函数内部可以访问。当globalScope 运行完之后，内部分配的内存会回收比如nameTest，titleTest。

es6 之前我们的作用域就这2种，但是在let 关键字的出现，让我们有了第三种作用域，块级作用域。代码自测：

```
1  function globalScope() {
2      for (let i = 0; i<6; i++) {
3          console.log(i)
4      }
5
6      console.log(i) // i is not defined
7  }
8
9  console.log(i) // i is not defined
```

块级作用域字面意思已经解释比较清晰了，即在当前直接包裹的范围内可以访问，在我们的代码中包裹变量i的正是for，即只在for循环内可以访问。

回归我们今天的正题，闭包。正如前面我们已经提到，一般当函数运行之后会自动回收该函数用到的内存比如函数内部定义的变量等等。

那么，是否有一种机制让函数即使运行好之后也不回收其内存，即保存此函数的内存不被自动销毁，可以进一步利用这个机制让所有调用函数的调用者共享函数内的变量来更方便的完成某个使用场景。

使用场景一

比如某销售公司员工工资运算程序。业务需求:

1. 部门内的所有员工都有底薪，且底薪相同, 2000
2. 部门内的所有员工都有提成，但是每个人根据销售业绩提成金额不同

这个是平时很常见的销售人员的薪资结构，我们根据这个业务用js完成一个函数完成这个业务需求，代码：

```
1  // 实例代码为了演示，没有考虑边界检查比如传入的值可能为负数的检查等等
2  function calcSalary(base, commission) {
3      return base + commission
4  }
5  let sales1 = calcSalary(2000, 10000) // 12000
6  let sales2 = calcSalary(2000, 20000) // 22000
7  let sales3 = calcSalary(2000, 30000) // 32000
8  let sales4 = calcSalary(2000, 40000) // 42000
9  let sales5 = calcSalary(2000, 50000) // 52000
10 let sales6 = calcSalary(2000, 60000) // 62000
```

非常简单就能轻松搞定！是不是觉得很有成就感。不妨我们自测考虑一下，既然每个人拥有相同的底薪，可以不用每次调用的时候都传入一个相同的底薪就能完成相同的功能么？

比如，我们想要下面的这种调用方式？

```
1  let sales1 = calcSalary(10000)
```



```
2 let sales2 = calcSalary(20000)
3 let sales3 = calcSalary(30000)
4 let sales4 = calcSalary(40000)
5 let sales5 = calcSalary(50000)
6 let sales6 = calcSalary(60000)
```

完成改方法一定让你觉得非常简单，可以如下：

```
1 function calcSalary(commission) {
2     return 2000 + commission
3 }
```

你看这种谁不会呢？！这个时候其他部门B管理来电话了因为觉得你的程序用的很好给之前那部门节省了很多时间，所以想要给他们部门也弄一个，他们部门唯一不同的就是底薪资从2000 变成 3000。聪明的你想了想太简单了，可以这样完成

```
1 function calcSalaryB(commission) {
2     return 3000 + commission
3 }
```

是吧，太简单了，只需要添加多一个函数就可以了。这时候，老板来电话了，因为你的程序给2个部门节省了很多时间，所以决定让你给其他100000000部门也来一个，顺便说一下，这公司真的很大就有那么多部门 ~ 每个部门的薪资结构相同不同的就是底薪。

简单啊，跑路啰，机智如我~

按照上面的逻辑需要再添加 100000000个函数？等等，确定么，这么多函数咋么管理呀。如果老板还有其他公司，每个公司也有那么多部门呢？？？只能跑路了：)

手机打开在招聘网站上已经在搜索下一家工作合适岗位。。。电话响了。。。

“喂，小炮，那个前段时间你和我提的那个事情，我考虑过了，我觉的靠谱，我俩找个合适的地方深入了解一下吧”

“啊，真的吗，太好了，行呀，稍晚联系”

太好了，公司心仪的对象总算是答应那个了，嘿嘿嘿~~~

稍等，刚不是已经准备跑路了么，咋办好呀，走也可以，但是公司里面有小a, 小b。。。小z都盯着她呢，如果换工作了，近水楼台拱手让人这事还能不能行了？？？

冷静一下抽根烟~~~

不行，不能走，咋么可能嘴里的肥肉让给其他人，一个坚定的声音在内心想起！

行吧，不走就不走呗。想想前面老板的那个问题咋么解决呢？理理思路，

1. 所有部门的薪资结构相同
2. 每个部门的底薪相同，部门与部门之间底薪不同
3. 每个员工的提成不同

这个好像可以使用闭包，让同一个部门的多个调用共享相同的底薪，只有提成不同。以此类推，其他所有 10000000个部门也是如此。那么可以这样：

```
1  function calcSalary(base) {
2      return function (commission) {
3          return base + commission
4      }
5  }
6
7  let deptA = calcSalary(2000)
8  // 同一个部门的多个调用共享一个底薪
9  deptA(10000) // 12000
10 deptA(20000) // 22000
11 deptA(30000) // 32000
12 deptA(40000) // 42000
13 deptA(50000) // 52000
14 deptA(60000) // 62000
15
16 let deptB = calcSalary(3000)
17 // 同一个部门的多个调用共享一个底薪
18 deptB(10000)
19 deptB(20000)
20 deptB(30000)
21 deptB(40000)
22 deptB(50000)
23 deptB(60000)
24
25 //手动回收内存，注意闭包无法自动回收内存，正如我们上一节已经讲述了
```

```
26 deptA = null
27 deptB = null
28
29 // ... 找合适酒店深入了解
```

太好了，这不就解决了么，利用闭包的性质，calcSalary 内部返回的函数引用了其外部的变量base，即使calcSalary 运行好也无法回收其内存。所以base 会一直存在于内存中。这样不仅不需要每次都传入一个base参数，且总的函数也就一个方便维护。

太好了，哪个酒店比较好呢？

使用场景二

debounce / throttle 常见的面试经典题目，考点之一就是闭包的理解。我们首先来看看debounce实现方式

```
1 // debounce
2 function _debounce(fn, timeOut) {
3   var timer = null;
4
5   return (...args) => {
6     clearTimeout(timer);
7     timer = setTimeout(() => fn.apply(this, args), timeOut);
8   };
9 }
10
11 const fn = function (params) {
12   console.log(params);
13 };
14 const d = _debounce(fn, 2000);
15 d('hello world');
16 d('hello world');
17 d('hello world');
18 d('hello world');
19 d('hello world');
20
21
22 // 2秒之后，输出：hello world
23 // 对的，只有一个hello world 会输出而不是5个
24 // 2秒是最后一个d('hello world')调用开始之后2秒
25 // 所有在最后一个d('hello world') 之前的调用都会被取消
26
27 // 手动释放内存因为其无法自动被内存回收
```

debounce的使用场景有很多，防抖的作用是指在指定时间内多次的触发以最后一个为准。比如支付按钮的防抖使用场景，不管用户按几次支付我们只想支付按钮的绑定函数运行一次。因网络延迟或客户端原因产生的延迟让用户以为卡住了用户很有可能在短时间内按多次支付按钮的。

回归我们的题目我们来一行一行的理解一下闭包如何在上面的代码发挥作用的

首先`_debounce(fn, timeOut)` 函数满足

1. 接受2个参数，第一个参数是一个函数, `fn`
2. 第二个参数是毫秒即多少毫秒之后运行作为第一个参数传入的函数, `timeOut`
3. 对`_debounce`在`timeOut`毫秒之内多次调用，只有最后一次调用会运行作为第一个参数传入的函数

那么这里的关键几点我们已经分析过了。那么实现方法可以整理成

1. 每次在`timeOut`毫秒内调用`_debounce` 我们都首先清除前一个 `_debounce`调用产生的 `setTimeout`
2. 一直重复第一条，直到最后一次的调用
3. 按照前2条，因为最后一条产生的`setTimeout`没有清除所以会在`timeOut`毫秒之后，运行作为第一个参数传入的函数

这里的关键点又是共享，共享也可以理解为预先保存，在这里预先保存的是`timer`变量，保存的方法是使用闭包的特点。

`_debounce`函数内部返回的函数引用了一个外部的变量，即`timer`变量，所以正如我们前面描述的，即使运行好`_debounce` 函数该函数的内存也不会被自动回收，所有该函数的调用都会分享这个`timer`变量。即实现了下一个调用可以清除上一个调用产生的`setTimeout`。

理解好了`debounce`，我们来了解一下`throttle`。

很多人可能都分不清与`debounce`的区别。`throttle` 在指定的时间内只运行一次。与`debounce`只运行最后一次的特点，可以现在清晰的了解2者的区别了吧。`throttle` 的使用场景也非常多，比如判断页面是否滚动在想要的某个位置。我们可以每次滚动都判断，但是这个是非常消耗性能的，那么我们就可以使用`throttle`, 每经过一段指定时间运行一个函数来确认是否已

经在想要的位置。那么既然已经了解了工作的原理，自测一下吧，试试是否可以完成一个 throttle 函数

总结

我们在这里讲述了下面的关键点

1. 作用域的理解
2. 闭包的产生
3. 闭包的特点
4. 闭包的实用性与手动回收内存提高性能
5. debounce
6. throttle

在本章我们在代码里使用了 apply 函数，与它类似还有 call 和 bind，我们可以在几乎所有比较常用的前端框架源码都有使用。那么我们在下一节来了解一下其使用的方法和实现的原理。

借力 - apply, bind, call的底层原理与实现

承接我们上一节所讲，我们在上一节练习中使用了apply方法，

```
1 // debounce
2 function _debounce(fn, timeOut) {
3   var timer = null;
4
5   return (...args) => {
6     clearTimeout(timer);
7     timer = setTimeout(() => fn.apply(this, args), timeOut);
8   };
9 }
10
```

apply, call, bind 也是面试常常会被考的题目。他们3个都有同一个作用，即强制改变this的指向。我们不妨回想一下3个问题，

1. 自己的开发经验中是否有用到过它们？
2. 在什么场景中需要使用它们？
3. 使用它们能够带来什么好处呢？

带着这几个问题我们来探索一下。

使用场景一

我们需要一个函数，用来找出数组中最大的值然后返回这个值。假设数组的元素都是整数且每个元素最大值不超过Number.MAX_SAFE_INTEGER, 数组至少有一个元素。这个函数实现非常简单，我们可以完成如下

```
1 function max(arr) {
2   let num = arr[0];
3   let arrLength = arr.length;
4   for (let i = 1; i < arrLength; i++) {
5     if (arr[i] > num) num = arr[i];
6   }
7
8   return num;
9 }
```

```
9 }
```

稍等，记得Math里面有个max函数, 使用方法如下，

```
Math.max(1,2,3) // 3
```

我们想想，是否可以借助使用Math.max方法直接借助该方法获取数组内元素的最大值？直接代码自测一下

```
Math.max([1,2,3]) // NaN
```


果然，这么做是不行的，查看math max的使用方法可以得知确实参数只能是0或多个数字不能是数组。

不饶了，直接上解决方法吧

```
Math.max.apply(null, [1,2,3]) // 3
```

我们根据这个倒过来试试如何实现apply函数，为什么使用apply 就可以了呢？第一个参数传入null有什么作用呢？


我们查看一下官方的apply定义，

 **apply()** 方法调用一个具有给定 this 值的函数，以及以一个数组（或类数组对象）的形式提供的参数。

注意：call()方法的作用和 apply() 方法类似，区别就是 call() 方法接受的是参数列表，而 apply() 方法接受的是一个参数数组。

行我们开始尝试，首先我们知道`apply` 参数接受2个参数，第一个是`this`的方向，第二个是一个数组或者类数组，数组的每一个元素就是函数的参数。我们可以得到

```
Function.prototype._apply = function(thisArg, args) {}
```

 我们为什么需要挂载在`Function`的原型下面呢？因为`apply`本身就是`Function` 原型里的一个属性，我们可以验证一下

```
Reflect.ownKeys(Function.prototype)
```

为了和`apply`区分开来，所以我们使用了 `_apply`

先来看一下比如平常的时候我们使用`apply`是如何使用的，

```
1  let a = {
2    hello: 'Hello'
3  }
4
5  let b = {
6    sayHello: function() {
7      console.log(this.hello)
8    }
9  }
10
11 b.sayHello.apply(a) // Hello
```

`b`对象里面其实没有`hello`，但是我们使用`apply`方法传入`a`就可以借用`a`对象的`hello`通过`b`对象里面的`sayHello`方法打印出来了。那我们分析一下是否可以理解这个等价于下面这样

```
1  // b.sayHello.apply(a)
2  // 等价于
3  let a = {
4    hello: 'Hello',
5    sayHello: function() {
```



```

6     console.log(this.hello)
7   }
8 }
9
10 a.sayHello() // Hello

```

自测的代码确实和我们推测的一样，上面的解决方法逻辑也就是解决this方向的内部原理，那么我们按照这个逻辑如何完成apply函数？如下，

```

1  Function.prototype._apply = function (thisArg, args=[]) {
2    thisArg.fn = this;
3    var result = args.length ? thisArg.fn(...args) : thisArg.fn();
4    delete thisArg.fn;
5    return result;
6  };

```

是不是没有你想的那么难！我们来分析一下代码

1. 首先在这thisArg上面添加一个属性fn，让调用apply的函数的函数即this等于thisArg.fn
2. 接着运行thisArg.fn 根据args是否是空来决定是否需要参数传给thisArg.fn, 且保存运行后的值
3. 使用delete 删除临时的thisArg.fn, 保持 thisArg和原来一样
4. 返回保存的值

我们来自测一下代码是否可以如我们想的那样

```

1  Function.prototype._apply = function (thisArg, args=[]) {
2    thisArg.fn = this;
3    var result = args.length ? thisArg.fn(...args) : thisArg.fn();
4    delete thisArg.fn;
5    return result;
6  };
7
8  let a = {
9    hello: 'Hello'
10 }
11
12 let b = {
13   sayHello: function() {

```

```

14     console.log(this.hello)
15   }
16 }
17
18 b.sayHello._apply(a) // Hello

```

确实，我们得到了我们想要的，输出了 Hello，太好了！

那么既然我们已经完成了自己的`_apply`, 完成`call`是不是更简单了，如下

```

1  Function.prototype._call = function (thisArg, ...args) {
2    thisArg.fn = this;
3    var result = thisArg.fn(...args);
4    delete thisArg.fn;
5    return result;
6  };
7
8  let a = {
9    hello: 'Hello'
10 }
11
12 let b = {
13   sayHello: function() {
14     console.log(this.hello)
15   }
16 }
17
18 b.sayHello._call(a) // Hello

```

自测一下是否可以期望的。

那么我们完成`apply`，`call`，还有一个`bind`相信应该不会难倒你，自测一下是否可以完成。注意`bind`和`apply`和`call`稍微不同，返回的是一个函数，而不是函数运行后的值。这个可以在函数外面包一层就可以。因为我们返回的是自己包一层的函数，等于是一个新对象，那么我们需要把这个新创建的对象和调用`bind`函数的`this`对象的原型相同。这2个的最中心代码如下

```

1  Function.prototype._bind=function(thisArg, ...args) {
2    ...
3    var fn = () => {
4      this.apply(thisArg, args.concat(Array.prototype.slice.call(arguments, 1)))

```

```
5     }
6     fn.prototype = Object.create(this.prototype)
7     return fn
8 }
```

好了，理解了call, apply, bind 的底层原理，我们回到最开始的函数

```
Math.max.apply(null, [1,2,3]) // 3
```

我们借用了null作为 this的方向，因为null的类型是object，所以可以作为第一个参数，第二个参数虽然是数组，但是在apply底层源码里面会自动展开，再来对比我们完成的apply代码

```
1  Function.prototype._apply = function (thisArg, args=[]) {
2    thisArg.fn = this;
3    var result = args.length ? thisArg.fn(...args) : thisArg.fn();
4    delete thisArg.fn;
5    return result;
6  };
7
8  let a = {
9    hello: 'Hello'
10 }
11
12 let b = {
13   sayHello: function() {
14     console.log(this.hello)
15   }
16 }
17
18 b.sayHello._apply(a) // Hello
```

apply, bind, call 改变this的方向解决了借力的问题，即节省了不必要的内存开销，增加性能。好比在我们这里，我们其实不需要自己完成一个找出数组中最大值的函数，而是可以借助Math.max 函数来完成。

如果对本章中的this方向不太理解，那我们的下一节就是对this方向的深入探索

this

this在js中使用的非常普遍，默认的，全局的this指向的是 window, 我们自测一下如下，

```
1  function func() {  
2    this.a = 1;  
3    this.b = 2;  
4    this.c = 3;  
5    this.d = 4;  
6  }  
7  func();  
8  console.log(window.a);  
9  console.log(window.b);  
10 console.log(window.c);  
11 console.log(window.d);  
12
```

```
> function func() {  
    this.a = 1;  
    this.b = 2;  
    this.c = 3;  
    this.d = 4;  
}  
func();  
console.log(window.a);  
console.log(window.b);  
console.log(window.c);  
console.log(window.d);
```

1

2

3

4

< undefined

>

可能你会觉得这个非常简单，那么我们再自测一下，看一下是否你也可以回答正确，题目如下

```
1 let a = {  
2   testfunc: (function () {  
3     console.log(this);  
4   })(),  
5 };
```

上面代码的this是谁你知道吗？

对的，上面代码的this是window，

```
> let a = {  
    testfunc: (function () {  
      console.log(this);  
    })(),  
  };
```

```
Window {0: global, window: Window, self: Window, document:  
  cation, ...}
```

因为testfunc 是一个自执行函数所以会先运行这个函数，当运行的时候不就是和我们最开始举例子的代码一样了么。

从这里也可以验证出来，this是谁和this在哪里被定义无关。上面就是最好的一个验证，我们在a对象的testfunc属相定义的这个函数，但是this确是window。

聪明的你肯定想到了，this是谁取决于被调用的时候。被调用的含义是指比如，a.b(), 即b函数被a调用。因为上面代码运行的时候没有被任何调用者调用，开始的时候已经阐述了，this的默认就是window。现在相信能更加比之前对this是谁有一个更清晰的辨别方法了。

我们接着来验证一下，上文提到的“this是谁取决于被调用的时候”这个论点，

```
1 let a = {  
2   name: 'abc',
```

```

3   testfunc: function () {
4       console.log(this.name);
5   },
6   };
7   a.testfunc(); // abc
8

```

使用了 `a.testfunc()` 调用函数的时候，我们内部尝试访问 `name` 这个属性，而且我们能访问成功。那么 `this` 现在就是 `a`。如果不是 `a`，那么就无法顺利打印出 `a.name`。从这里我们可以推断出，`this` 是谁取决于被调用的时候，而且规律就是找 `"."` 前面的那个，比如 `a.testfunc()` 那么 `this` 就是 `a`。我们再自测一下，

```

1   function hi() {
2       name = 'a'
3       console.log(this.name)
4   }
5
6   let a = {
7       name: 'b',
8       test: hi
9   }
10
11  hi() // a
12  a.test() //b

```

对吧，第一个打印 `a`，`hi` 函数调用的时候前面没有 `'` 那么默认 `hi` 函数里面的 `this` 就是全局 `window`。因为默认如果没有添加任何修饰符比如 `var let` 那么默认就是在 `window` 上面添加。

当 `a.test()` 调用的时候，`'` 前面的是 `a`，所以 `hi` 函数的 `this` 是 `a`，那么自然就会打印出 `a` 里面的 `name`，即 `b`。

其实难么，掌握好方法就不难了。那么这里还有一个比较特殊的需要了解，箭头函数

箭头函数有如下特点

1. 没有 `this`
2. 没有 `arguments`

可能你会说，箭头函数里面可以使用this。那么这里说的没有this，说的是箭头函数自身没有this，如果在箭头函数内使用this，那么this就是箭头函数之外的上下文，上代码

```
1  name = 'abc'
2  let a = {
3      name: 'b',
4      test: ()=>{
5          console.log(this.name)
6      }
7  }
8
9
10 a.test() // 'abc'
```

是不是你会反驳上文提出this是谁按照‘’前面来决定的么？那么这里验证了上文提出的箭头函数没有this，箭头函数使用this，this就是箭头函数外的上下文。那么这里就是a之外的window，打印了window上面的name，即 abc

那么我们上节课程正好了解了可以强制改变this，使用call apply。那么我们来验证一下是否可以强制改变箭头函数的this呢，代码

```
1  name = 'abc'
2  let a= {
3      name: 'b'
4      test: ()=>{
5          console.log(this.name)
6      }
7  }
8
9  a.test.call(a) // abc
```

啊呀，强制改变也改变不了，因为箭头函数函数本身没有this，没有this咋么改变呢？对吧！

那么有方法么，就是想要打印 b，

```
1  name = 'abc'
2  let a= {
```



```

3     name: 'b',
4     test: function () {
5         const func = ()=>{
6             console.log(this.name)
7         }
8         func()
9     }
10 }
11
12 a.test() // b

```

原理解释，当调用就会找上下文，这里的上下文就是包裹func箭头函数的test上下文，即a，那么输出b就是顺利成章了。

明面上的应该都懂了，如果题目稍微改变一下自测一下是否可以回答正确呢，

```

1  let a = {
2      name: 'b',
3      test: function() {
4
5          setTimeout(()=>{
6              console.log(this.name)
7          }, 2000)
8      }
9  }
10
11 a.test() // b

```

是的还是打印出来的还是b。只需要记住上文的技巧，应该能够搞懂this。

那么如果用一句话来描述this，那么可以理解为 `()=>{} 不改变this`，因为this始终是其上下文内的。

好了，我们下一章开始讲讲异步的原理比如promise / generator / async await，让我们的代码更加的优雅。

promise A+ 原理实现

promise 是经常在工作会使用得到的，其解决了多个回调产生的代码难于维护和理解的问题。那么我们就来实现一个自己的promise来加深我们对其的内部机制和实现原理

```
1  function Promise() {
2      const PENDING = 'PENDING'
3      const RESOLVE = 'RESOLVE'
4      const REJECT = 'REJECT'
5      const resolveCallback = []
6      const rejectCallback = []
7
8      this.status = PENDING
9      this.value = undefined
10     this.reason = undefined
11     constructor(runner) {
12         resolve(value) {
13             if (this.status === PENDING) {
14                 this.value = value
15                 this.status = RESOLVE
16                 resolveCallback.forEach(cb=>cb())
17             }
18         }
19     }
20
21     reject(resson) {
22         if (this.value === PENDING) {
23             this.reason = reason
24             this.status = REJECT
25             rejectCallback.forEach(cb=>cb())
26         }
27     }
28     try {
29         runner(resolve, reject)
30     } catch (e) {
31     }
32 }
33
34 }
```

上面的代码是第一部分，我们会一部分一部分来演示实现的过程可以降低理解的难度和时间成本，我们先来了解一下上面的代码产出了什么，

首先，promise 规范了内部一共有3种状态，pending, resolve, reject。当状态发生改变之后则无法再更改其状态。即 pending -> resolve 之后，resolve->reject 就不再可以了，因为状态发生改变之后不再允许任何方式再次对状态更改。所以我们一开始使用const定义了这3个状态，而且我们定义了起始状态为pending，即等待。

然后，构造函数接受一个函数，我们的代码是runner函数，这个函数接受2个函数为参数，即 resolve 和 reject，回想一下平时使用promise的场景，

```
1 let p = new Promise((resolve, reject)=>{
2     ...
3 })
```

这里传给Promise 的回调函数就是我们代码中传给constructor的runner函数，我们平时使用的时候可能类似下面这样

```
1 let p = new Promise((resolve, reject)=>{
2     // resolve(1) 或
3     // reject('error')
4 })
```

resolve 和 reject 都接受一个参数，reject 字面意思表示拒绝, 它的参数可以是描述为什么拒绝的缘由。相反，resolve 则是成功，它的参数就是promise成功返回的值。当我们调用2者之中任意一个的时候，代表状态已经发生了改变，就2种可能，即 pending-> resolve / pending->reject。我们已经说明，状态改变之后是不可以再次改变的。

代码中我们定义了2个数组 resolveCallback / rejectCallback 用来收集then的回调，我们先来添加收集then方法然后慢慢解释其作用，

```
1 function Promise() {
2     // 省略上面的代码
3
4     // then方法
5     then(onFulfilled, onRejected) {
6         if (this.status===PENDING) {
7             this.onResolvedCallback.push(()=>{
```

```

8         onFulfilled(this.value)
9     })
10    this.onRejectedCallback.push(()=>{
11        onRejected(this.value)
12    })
13 }
14 }
15
16 }

```

这里收集的前提是在当前的promise的状态还是pending的时候。那么，换句话说，当前的promise也可能已经不是pending了，确实是这样的比如，

```

1  let p = new Promise((resolve, reject)=>{
2      resolve(1)
3  }).then((data)=>{
4      console.log(data) // 1
5  })

```

上面的代码在then调用之前已经resolve(1)，resolve是我们之前定义的方法。在then里面status 现在已经变成 RESOLVE，

```

1  resolve(value) {
2      if (this.status === PENDING) {
3          this.value = value
4          this.status = RESOLVE
5          resolveCallback.forEach(cb=>cb())
6      }
7  }

```

上面的代码让保存resolve的值给value，改变的当前promise的status，循环了resolveCallback 数组里面之前收集的then回调函数，然后逐一运行里面的函数。我们接着来看resolveCallback 里面的值是如何收集的，

```

1  // then方法
2  then(onFulfilled, onRejected) {

```

```

3     if (this.status===PENDING) {
4         this.onResolvedCallback.push(()=>{
5             onFulfilled(this.value)
6         })
7         this.onRejectedCallback.push(()=>{
8             onRejected(this.value)
9         })
10    }
11 }

```

答案就是在then 方法，但是有一个条件即then方法被调用的时候当前promise的status是PENDING。我们举个例子，

```

1  let p = new Promise((resolve ,reject)=>{
2      // 异步操作需要时间完成
3      setTimeout(()=>{resolve(123)}, 200)
4  }).then(data=>{
5      console.log(data) // 123
6  })

```

虽然我们模拟了使用了延迟调用resolve, 可是我们的then里面一样可以得到resolve 的值。实现的原理就是 当then调用的时候，当前的status 还是PENDING, 那么我们可以让其保存起来 then的回调函数等待status 转变成resolve 之后再从保存的地方取出然后依次运行收集的回调函数。这就有了我们上面的then方法代码。

现在我们来实现链式调用，我们先来演示平时的promise then 链式使用方法

```

1  let a = new Promise((resolve, reject)=>{
2      setTimeout(()=>{resolve(100)}, 200)
3  }).then((data)=>{
4      console.log(data)
5      new Promise((resolve, reject)=>{
6          setTimeout(()=>{resolve(200)}, 200)
7      })
8  })
9  ).then((data)=>{
10     console.log(data)
11 })

```

链式调用底层原理就是当then被调用，then 函数内新生成一个promise对象，然后最后返回这个新建promise对象，比如

```
1  // ... 省略代码
2
3  then(onFulfilled, onRejected) {
4      let promise2 = new Promise((resolve, reject)=>{
5          if (this.status === RESOLVED) {
6              setTimeout(()=>{
7                  try {
8                      let x = onFulfilled(this.value)
9                      resolvePromise(promise2, x, resolve, reject)
10                 } catch (e) {
11                     reject(e)
12                 }
13             },0)
14         }
15     })
16 })
17
18 return promise2
19 }
```

因为每次返回的都是promise对象所以可以实现链式调用，上面的代码实现了当then被调用，已经resolve的代码，我们来慢慢解释。首先我们在then里面新创建了一个promise2对象, 我们然后判断当前的promise（注意，这里是当前的promise不是 promise2）状态是否已经resolved, 如果已经resolved，我们就直接调用onFulfilled(this.value)，这里的onFulfilled就是then的第一个参数，比如

```
1  let p = new Promise((resolve, reject)=>{
2      resolve(1)
3  })
4  p.then((data)=>{
5      console.log(data)
6  })
```

then函数接受2个参数，第一个是当状态成功的回调，另外一个错误地回调，所以我们刚刚的代码onFulfilled(this.value) 就是成功的回调。this.value 是当前promise（注意这里是

promsie 不是promise2) 调用resolve函数之后把值给了value，代码我们已经在上面实现了，

```
1 let resolve = (value) => {
2   if (this.status === PENDING) {
3     this.value = value; // 就是这里
4     this.status = RESOLVED;
5     this.onResolvedCallbakcs.forEach((fn) => fn());
6   }
7 };
```

我们刚刚的代码使用了

```
1 // ... 省略代码
2
3 then(onFulfilled, onRejected) {
4   let promise2 = new Promsie((resolve, reject)=>{
5     if (this.status === RESOLVED) {
6       setTimeout(()=>{
7         try {
8           let x = onFulfilled(this.value)
9           resolvePromise(promise2, x, resolve, reject)
10          } catch (e) {
11            reject(e)
12          }
13        },0)
14      }
15    })
16  })
17
18  return promise2
19 }
```

这里使用的异步操作为什么呢？因为如果不使用异步操作我们在resolvePromise(promise2, x, resolve, reject) 内的promise2是空。空的原因是new Promise 还没有完成赋值给promise2，就已经使用了promise2。解决方法就是放在下一个宏任务就可以解决了，这里也可以使用微任务比如 mutationObserver 来解决，我们就不做展开了。这里还有一个关键是resolvePromise函数的 resolve, reject 参数，这2个参数是当前promise内的resolve和reject。这一点非常重要，因为就是这个让promise与promise之间串接起来了。

我们现在来实现resolvePromise函数，

```
1  const resolvePromise = (promise2, x, resolve, reject) => {
2    // 解决循环引用的问题,比如
3    // let p = new Promise((resolve, reject)=>{resolve(1)})
4    // let promise2 = p.then(data=>{return promise2 })
5    // promise2.then(()=>{}),err=>{console.log(err)})
6
7    // 这里是通过这个值来实现 resolve 和 reject 只能运行其中一个
8    let called = false;
9    if (promise2 === x) {
10      return reject(
11        new TypeError('Chaining cycle detected for promise #<Promise>')
12      );
13    }
14    if ((typeof x === 'object' && x !== null) || typeof x === 'function') {
15      //这里判断x可能是一个promise
16
17      // 这里的then需要用try catch 因为可能有
18      // Object.defineProperty(x, then, {
19      //   get() {throw error}
20      // })
21      //
22      try {
23        let then = x.then;
24        if (typeof then === 'function') {
25          // 如果进到这里我们就认为是promise了，
26          // 即使只是一个普通函数空的我们也只能判断成promise
27
28          // 没有使用 x.then 因为可能会有
29          // Object.defineProperty(x,then,{
30          //   get() {
31          //     if (++index==2) {throw new Error()}
32          //   }
33          // })
34          // 第一次不报错第二个次报错，所以我们使用下面的方法
35          then.call(
36            x,
37            (y) => {
38              if (called) return;
39              called = true;
40              // 这里使用递归因为如果可能返回promise，我们
41              // 需要一直直到不是promise
42              // 比如，
43              //let p= new Promise((resolve, reject)=>{
44
45              // })
46              // p.then((data)=>{
```

```

47         // return new Promise((resolve, reject)=>{
48         // resolve(new Promise((resolve, reject)=>{}))
49         // })
50         // })
51         resolvePromise(promise2, y, resolve, reject);
52     },
53     (e) => {
54         if (called) return;
55         called = true;
56         reject(e);
57     }
58 );
59 } else {
60     resolve(x);
61 }
62 } catch (e) {
63     if (called) return;
64     called = true;
65     reject(e);
66 }
67 } else {
68     resolve(x);
69 }
70 };

```

这里首先判断是否x与promise2 相等来确认是不是自己给自己赋值了，如果是我们就直接使用传进来的reject 拒绝。

called 变量是用来控制resolve 和 reject 只能调用其中之一。

我们接着判断x是不是一个promise，x的可能就2种，普通值和promise。如果是promise，那么我们还需要关注一下是否可能promise里面返回promise，所以我们使用了递归的方法一直解析，但是这里关键点记住，我们始终用的是当初传进来的resolve, reject。那么为什么这么设计呢？因为这样无论当前的promise里面是否是普通值还是promise或者是promise里面有多层promise，最外层的promise都会等待完成之后再运行下一层的promise，比如，

```

1  let b = new Promise((resolve, reject)=>{
2      resolve(1)
3      console.log(1)
4  })
5  b.then((data)=>{
6      return new Promise((resolve, reject)=>{
7          resolve(new Promise((resolve, reject)=>{

```

```

8         resolve(2)
9         console.log(2)
10    })
11
12    console.log(2.1)
13  })
14  }).then((data)=>{
15    console.log(3)
16  })
17
18  // 依次输出 1 / 2 / 2.1 / 3

```

上面的resolvePromise做了备注和理解，可以慢慢参看。

我们接着讲then函数，then函数的2个参数都是可选的，也就是即使给出一个空的then也是可以的。重点部分是即使是then函数为空函数则依然可以把值传递给下一个then, 比如

```

1  let p = new Promise((resolve, reject)=>{
2    reject(1)
3  })
4  p.then().catch(err=>console.log(err)) // 1

```

那么这样是如何实现的呢？其实我们可以这样想，既然then函数的2个参数是可选的，那么我们需要判断是否then有参数，如果没有，那么我们自己在then内部创建这2个函数就可以了，比如

```

1  then(onFulfilled, onRejected) {
2    onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : val=>
3    onRejected = typeof onRejected === 'function' ? onRejected : err=> {th
4  }

```

这样就可以解决了。

我们已经整理了如何实现一个promise A+，过程还是需要点时间才吃透的。

我们下一章讲解如何实现promise各种api，比如 promise.all 等等

promise api 的实现

承接上一节我们已经实现了一个promise A+，那么来完善一下promise 的api 实现原理

```
1  //promise all
2
3  function Promise() {
4      static all(arr) {
5          if (!Array.isArray(arr)) return new TypeError(`${arr} is not array`);
6          return new Promise((resolve, reject) => {
7              let arr = []
8              let len = promises.length
9              for (const promise of promises) {
10                 if (typeof promise.then === 'function' && promise !== null) {
11                     try {
12                         promise.then((val) => {
13                             arr.push(val)
14                         })
15                     } catch(e) {
16                         reject(e)
17                     }
18                 } else {
19                     arr.push(promise)
20                 }
21             }
22             resolve(arr)
23         })
24     }
25 }
26 }
```

内部原理非常简单，循环遍历promise数组，依次调用then然后保存其值在arr数组。如果其中某一个promise 有错误则直接reject。如果全部正确则调用resolve(arr)

这里的重点是最外层的promise，它控制了内部n个promise最终的返回结果通过调用最外层的resolve 或者 reject

那么我们接着实现Promise.prototype.catch

```
1  Promise.prototype.catch = function (cb) {
```

```
2     return this.then(null, cb)
3 }
```

是的catch 其实就是then （如果想知道then的实现方法可以参看上一节）

再来一个 Promise.resolve

```
1 // Promise.resolve
2 function Promise {
3     static resolve(data) {
4         return new Promise((resolve, reject)=>{
5             resolve(data)
6         })
7     }
8 }
```

promise 上面还有其他的api，内部实现原理如果能掌握上面的三种，其他应该也不难了。就当作练习自己自测一下是否理解了。

了解了Promise，接着我们来了解常用的js数据结构 - 数组

数组api整理

js数组是我们常用的一种数据结构，我们在这一章来梳理一下js的数组api。

```
1 let a = new Array(6)
2
3 let b = []
4 b.length = 6
5
6 let _c = {
7   0: 'a',
8   1: 'b',
9   length: 2
10
11 }
12 let c = Array.from(_c)
```

上面的3种都可以定义一个数组，第一和第二个可能你很熟悉。那么第三个是什么呢？是类数组，类数组本身不是数组但是内部的结构很类似于数组比如有索引和长度length。Array.from 是es6 新增的函数，可以把类数组转换成真正意义上的数组。obj / set / map 都可以转换成真正的数组。

```
1 const map = new Map([[1, 2], [2, 4], [4, 8]]);
2 Array.from(map);
3 // [[1, 2], [2, 4], [4, 8]]
4
5 const set = new Set(['foo', 'bar', 'baz', 'foo']);
6 Array.from(set);
```

有时候需要在不同的数据结构之间转换成数组为了使用数组上面的api，比如map -> array, set->array, 在算法题目中是经常使用。

好了，然后自测一下是否相有时候会搞混哪几个数组api函数会改变数组本身，哪几个不会？我们现在来梳理一下

```
1 // 会改变数组本身
```

```

2
3 // pop
4 let a = [1,2,3,4,5,6]
5 let item = a.pop() // [1,2,3,4,5]
6 console.log(a) [1,2,3,4,5]
7 console.log(item) // 6
8
9 // shift
10 let b = [1,2,3,4,5,6]
11 let item = b.shift()
12 console.log(b) // [2,3,4,5,6]
13 console.log(item) // 1
14
15 //reverse
16 let c = [1,2,3,4,5,6]
17 let item = c.reverse()
18 console.log(c) // [6,5,4,3,2,1]
19 console.log(item) // [6,5,4,3,2,1]
20
21 // unshift
22 let d = [1,2,3,4,5,6]
23 let item = d.unshift(-1,0)
24 console.log(d) // [-1,0,1,2,3,4,5,6]
25 console.log(item) // 8
26
27 //splice
28 let e = [1,2]
29 let item = e.splice(1,1)
30 console.log(e) // [1]
31 console.log(item) // [2]
32
33 // copyWithin
34 let f = [1,2,3,4,5,6]
35 let item = f.copyWithin(0, 3)
36 console.log(f) // [4,5,6,4,5,6]
37 console.log(item) // [4,5,6,4,5,6]
38
39 // fill
40 let g = [1,2,3,4,5,6]
41 let item = g.fill(100,2,3) // [1,2,100,100,100,6]
42 console.log(g) // [1,2,100,4,5,6]
43 console.log(item) // [1,2,100,4,5,6]

```

```

1 // 不改变数组的方法
2
3 // concat
4 let a = [1,2,3]
5 let item = a.concat(100,[200,300])

```

```

6 console.log(a) // [1,2,3]
7 console.log(item) // [1,2,3,100,200,300]
8
9 // join
10 let b = [1,2,3]
11 let item = b.join()
12 console.log(b) // [1,2,3]
13 console.log(item) // "1,2,3"
14
15
16 let c = [1,2,3]
17 let item = c.slice(1,2)
18 console.log(c) // [1,2,3]
19 console.log(item) // [2]
20
21 let d = [1,2,3]
22 let item = d.toString()
23 console.log(d) // [1,2,3]
24 console.log(item) // "1,2,3"
25
26 let e = [1,2,3]
27 let item = e.indexOf(1)
28 console.log(e) // [1,2,3]
29 console.log(item) // 0
30
31 let f = [1,2,3]
32 let item = f.includes(1)
33 console.log(f) // [1,2,3]
34 console.log(item) // true
35

```

上面分类了2种数组api，即会改变自身&不会改变自身。下面我们来梳理一下数组的遍历方法

```

1 // foreach
2 let arr = [1, 2, 3];
3 let b = {
4   val: 2,
5   test(item) {
6     this.val += item;
7     return this.val;
8   },
9 };
10 arr.forEach(function (item) {
11   console.log(this.test(item)); // 3 5 8
12 }, b)

```



```

13
14 // every
15 arr = [1, 2, 3];
16 b = arr.every((item) => item > 0);
17 console.log(b); // true
18
19 // some
20 arr = [1, 2, 3];
21 b = arr.every((item) => item > 2);
22 console.log(b); // true
23
24 //map, 不改变原数组, 返回处理过的数组
25 arr = [1, 2, 3];
26 b = arr.map((item) => item + 100);
27 console.log(arr); // [1,2,3]
28 console.log(b); // [101,102,103]
29
30 //filter, 不改变原有数组, 返回处理过的数组
31 arr = [1, 2, 3];
32 b = arr.filter((item) => item > 2);
33 console.log(arr); // [1,2,3]
34 console.log(b) // [3]
35
36 //reducer, 不改变原有数组, 返回处理过的值
37 arr = [1, 2, 3];
38 b = arr.reduce((prev, current) => prev+current);
39 console.log(arr); // [1,2,3]
40 console.log(b); // 6
41
42 //findIndex
43 arr = [1, 2, 3];
44 b = arr.findIndex((item) => item > 2);
45 console.log(arr); // [1,2,3]
46 console.log(b); [2]
47
48 //find
49 arr = [1, 2, 3];
50 b = arr.find((item) => item > 2);
51 console.log(arr); // [1,2,3]
52 console.log(b); //3
53
54 //keys
55 arr = [1, 2, 3];
56 b = arr.keys();
57 console.log(arr);
58 console.log(b.next().value); //0
59 console.log(b.next().value); //1
60 console.log(b.next().value); //2
61 console.log(b.next().value) //undefined
62
63

```



类数组

承接上一节，我们梳理了常用的数组api和演示了使用的方法，我们这一节了解一下类数组

类数组，类似数组可以又不相同，我们编码的时候可能你已经用到了确不知道，比如 arguments 和 string

```
1 function abc() {
2     console.log(typeof arguments) // object
3 }
4 abc() // object
5
6 function def() {
7     for (let i = 0; i < arguments.length; i++) {
8         console.log(arguments[i]);
9     }
10 }
11 def(1, 2, 3); // 1 2 3
12
13 let str = 'abcdef';
14 for (let i = 0; i < str.length; i++) {
15     console.log(str[i]); // a b c d e f
16 }
17
```

我们上面的代码可以自测 arguments 是对象，不是数组，但是一定功能却能像数组一样比如遍历，按照索引访问。string 一样。那么底层的数据结构是咋么实现的呢，

```
1 let abc = {
2     0: 'abc',
3     1: 'def',
4     'length': 2
5 }
```

就如上面的代码，内部的数据结构key都是有序数字，最后添加一个length属性。这样就可以实现使用索引访问，那么为什么也可以类似于数组那样遍历呢，

```
1 function abcdef() {
2     console.log(Reflect.ownKeys(arguments))
3 }
4 abcdef() // ['length', 'callee', Symbol(Symbol.iterator)]
```

上面的代码我们已经了解arguments 上面的keys，其中有一个 Symbol.iterator, 这就是为什么可以遍历的由来。我们可以再来测试一下string，

```
1 function abcdef2() {
2     console.log(typeof 'abc'[Symbol.iterator] === 'function');
3 }
4 abcdef2(); // true
```

string不是object，我们使用了typeof 来判断是否某个非对象数据是带有iterator的。

那么数组的遍历方法可以使用在arguments类数组对象上面么，

```
1 function abcdef3() {
2     arguments.forEach((item) => console.log(item));
3 }
4 abcdef3(); //TypeError: arguments.forEach is not a function
```

报错了，forEach是数组原型上面的方法没有办法直接在类数组arguments 上面使用，那么如何让其转换成真正的数组从而可以使用数组上面的方法呢，

```
1 function abcdef4() {
2     let arg = Array.prototype.slice.call(arguments);
3     console.log(Object.prototype.toString.call(arg));
4 }
5 abcdef4(1, 2, 3); // [object Array]
6
```

这里不使用typeof 来自测是否是数组，因为数组本身也是对象，所以使用typeof会返回object。使用Object.prototype.toString 可以区分不同的object类型比如数组[object Array]这

在前几节我们已经讲过了。

转换之后就可以使用所有数组的方法了。那么问题来了，javascript 为什么需要类数组呢？我觉得数组的遍历和下标是开发者经常使用的，但是数组的方法比如pop，push有时候是不必要的比如

```
1 function abcdef6() {  
2   console.log(arguments.push('any'));  
3 }  
4 abcdef6(1, 2, 3); // TypeError: arguments.push is not a function
```

我们尝试在给定的参数1，2，3之后直接添加多一个参数，这个在大部分情况下面是有混乱的，所以这个可能是类数组的使用场景和当初设计的由来（猜的）

如果你有更好的解释欢迎随时讨论。

我们下一节接着来讨论数组，讨论数组的扁平化。

数组扁平化

扁平化，即把可能存在的嵌套数组展开，比如

```
1 // 原始数组
2 [1,2,[1,2,[1,2,3]]]
3 // 扁平化之后的数组
4 [1,2,1,2,1,2,3]
```


可以实现扁平化的方法有很多，

方法一

```
1 let arr = [1, 2, [1, 2, 3, [1, 2, [1, 2, 3]]]];
2 arr = arr.toString();
3 arr = arr.split(',');
4 //['1', '2', '1', '2','3', '1', '2', '1','2', '3']
5 console.log(arr);
```

方法二

```
1 let arr2 = [1, 2, [1, 2, 3, [1, 2, [1, 2, 3]]]];
2 //[1,2,1,2,3,1,2,1,2,3]
3 // 这里的Infinity 是flat的参数，表示不管内部嵌套多深都会扁平化，
4 // 是的，这个参数提供开发者可以控制最多展开层级数
5 console.log(arr2.flat(Infinity));
```

 对比上面的2个方法，我们可以知道略有区别，方法一的原始数组里的元素都是数字，转换后变成string。相反的，方法二转换后的数组内部原素和转换前的数组元素类型一致。

还有很多其他方法可以实现比如递归，我们这里就不展开了。这里再说明一下展开运算符不能解决多于一层的扁平，比如

```
> [...[1,2,3],6]
< ▶ (4) [1, 2, 3, 6]
> [...[1,2,3,[4,5,6]],6]
< ▶ (5) [1, 2, 3, Array(3), 6]
>
```

使用展开运算符完成多层的嵌套扁平的正确姿势，

```
1 let arr6 = [1, 2, 3, [1, 2, 3, [1, 2, 3]]];
2 while (arr6.some((item) => Array.isArray(item))) {
3   arr6 = [].concat(...arr6);
4 }
5 // [1,2,3,1,2,3,1,2,3]
6 console.log(arr6);
```

上面的代码使用循环判断数组内部是否还有数组，如果存在我们则扁平一层，然后这样一直判断，就可以一层一层的扁平。

那么下一章我们来了解一下数组排序的常见使用方法

数组排序算法

冒泡排序

```
1 function bubbleSort(arr) {
2   if (!Array.isArray(arr)) throw new TypeError('need array');
3   let arrLen = arr.length >>> 0;
4   if (arrLen < 2) return arr;
5   if (arrLen > 2 ** 53 - 1)
6     throw new TypeError(
7       'the number of elements in array exceed the max length'
8     );
9
10  while (arrLen > 0) {
11    for (let i = 0; i < arrLen - 1; i++) {
12      if (arr[i] > arr[i + 1]) {
13        arr[i] ^= arr[i + 1];
14        arr[i + 1] ^= arr[i];
15        arr[i] ^= arr[i + 1];
16      }
17    }
18    --arrLen;
19  }
20  return arr;
21 }
22 // [1,1,2,3,5,5,10]
23 console.log(bubbleSort([2, 1, 5, 1, 10, 3, 5]));
```

1. 冒泡排序比较相邻2个元素大小，后面那个数如果小于前面那个数则让他们2个数交换
2. 当第一次for循环完成之后，就会找出数组中的最大值，位置也已经排在了数组的最右边
3. 当第二次for循环完成后，就会找出数组中第二最大值，位置也已经排在了最大值的前面那位，以此类推。
4. 时间复杂度为 $O(n^2)$ 。
5. 代码中使用了位运算代替中间值完成2个数的交换，当然也可以使用中间值。

快速排序

```
1 function quickSort(arr) {
2   if (!Array.isArray(arr)) throw new TypeError('need array');
```



```

3   let arrLen = arr.length >>> 0;
4   if (arrLen < 2) {
5       return arr;
6   }
7   if (arrLen > 2 ** 53 - 1)
8       throw new TypeError(
9           'the number of elements in array exceed the max length'
10      );
11
12  var pivotIndex = Math.floor(arrLen / 2);
13
14  var pivot = arr.splice(pivotIndex, 1)[0];
15
16  var left = [];
17
18  var right = [];
19
20  for (var i = 0; i < arr.length; i++) {
21      if (arr[i] < pivot) {
22          left.push(arr[i]);
23      } else {
24          right.push(arr[i]);
25      }
26  }
27
28  return quickSort(left).concat([pivot], quickSort(right));
29  }
30  // [1,1,2,3,5,5,10]
31  console.log(quickSort([2, 1, 5, 1, 10, 3, 5]));

```

1. 找出数组的中间索引值，比如5个元素就是 $5/2 = 2.5 \Rightarrow 2$ ，即数组的第3个元素
2. 比如6个元素就是 $6/2 = 3 \Rightarrow 3$ ，即数组的第4个元素
3. 然后取出这个元素让数组剩余的元素都和这个元素比大小
4. 如果比这个元素大，则放入 right 数组
5. 如果比这个元素小或者等于这个元素，则放入 left 数组
6. 目前会有left 数组， right 数组，和这个中间索引值元素
7. left数组包含了所有小于或等于这个中间索引值元素
8. right数组包含了所有大于这个中间索引值元素
9. 然后以left数组作为一个整体递归步骤1-6
10. 然后以right数组作为一个整体递归步骤1-6
11. 使用concat拼接left数组 + 中间索引值 + right数组
12. 时间复杂度 $O(n\log n)$

插入排序

```
1 function insertSortTest(arr) {
2   if (!Array.isArray(arr)) throw new TypeError('need array');
3   let arrLen = arr.length >>> 0;
4   if (arrLen < 2) {
5     return arr;
6   }
7   if (arrLen > 2 ** 53 - 1)
8     throw new TypeError(
9       'the number of elements in array exceed the max length'
10    );
11
12   let onHand = [];
13   onHand.push(arr[0]);
14
15   for (let i = 1; i < arr.length; i++) {
16     for (let j = onHand.length - 1; j >= 0; j--) {
17       if (arr[i] > onHand[j]) {
18         onHand.splice(j + 1, 0, arr[i]);
19         break;
20       }
21
22       if (j === 0) {
23         onHand.unshift(arr[i]);
24       }
25     }
26   }
27
28   return onHand;
29 }
30 console.log(insertSortTest([2, 1, 5, 1, 10, 3, 5]))
31
```

插入排序的理解非常容易，我们创建一个空数组，把待排序的数组的第一个元素添加进空数组，好比打牌的时候摸第一张牌在手里。这里的比喻待排序的数组就是牌堆，手里的牌就是上面的代码onHand数组。搞清楚这个后面就简单了。

因为我们已经把待排序数组第一个元素加进了手里，即onHand数组。所以待排序数组从第二个元素开始遍历。遍历的时候好比打牌的时候从牌堆里面拿一张牌和现在手上有的牌做比较。从牌堆里面取一张牌代表的就是从待排序数组里面取一个数。


比较的时候我们把从待排序数组里面取出的数和onHand 数组里面的每一个数做比较。上面的代码里面选择的是onHand数组内的数从大至小依次和取出的数对比较直到2个情况会停止比较。

第一种，当取出的数比onHand里面的某个数大的时候，我们记录位置然后就可以把取出的数插进那个位置

第二种，当手里的数都大于取出的数，则取出的数插进onHand数组的第一位，即最小位。

最后我们返回onHand数组就好，里面的数已经全部排序好了。

时间复杂度 $O(n^2)$

 这里借用了新开一个数组onHand来完成插入排序因为方便理解，只不过这样做需要更多的空间即onHand数组

其实，插入算法也可以不用额外使用一个数组直接在原数组里面完成排序，这个就作为练习给各位小伙伴

选择排序

```
1 function selectSort(arr = []) {
2   if (!Array.isArray(arr)) throw new TypeError('need array');
3   let arrLen = arr.length >>> 0;
4   if (arrLen < 2) {
5     return arr;
6   }
7   if (arrLen > 2 ** 53 - 1)
8     throw new TypeError(
9       'the number of elements in array exceed the max length'
10    );
11   for (let i = 0; i < arrLen - 1; i++) {
12     let minIndex = i;
13     for (let j = i + 1; j < arrLen; j++) {
14       if (arr[j] < arr[minIndex]) minIndex = j;
15     }
16     if (j !== minIndex) {
17       arr[i] ^= arr[minIndex];
18       arr[minIndex] ^= arr[i];
```

```

19     arr[i] ^= arr[minIndex];
20 }
21
22 }
23 return arr;
24 }
25 console.log(selectSort([2, 1, 5, 1, 10, 3, 5]))
26

```

选择排序应该比上面几个更容易理解，选择排序是每一次循环从数组里面找出剩余数里面最小的那个数的索引然后和数组的内剩余数的第一个元素交换位置，那么每一个循环就会依次找出剩余数的最小值然后放在相对应的位置。

代码里面的外层for循环只需要比较数组长度-1，因为最后的那个数已经是最小值所以不需要再比较。

时间复杂度是 $O(n^2)$

MergeSort

```

1  function mergeSort(arr) {
2    const merge = (right, left) => {
3      let lp = 0;
4      let rp = 0;
5      const container = [];
6      while (lp < left.length && rp < right.length) {
7        if (right[rp] < left[lp]) container.push(right[rp++]);
8        else container.push(left[lp++]);
9      }
10     while (lp < left.length) {
11       container.push(left[lp++]);
12     }
13     while (rp < right.length) {
14       container.push(right[rp++]);
15     }
16     return container;
17   };
18
19   const mergeSort = (arr = []) => {
20     if (arr.length === 1) return arr;
21     const middle = Math.floor(arr.length / 2);
22     const leftContainer = arr.slice(0, middle);
23     const rightContainer = arr.slice(middle, arr.length);

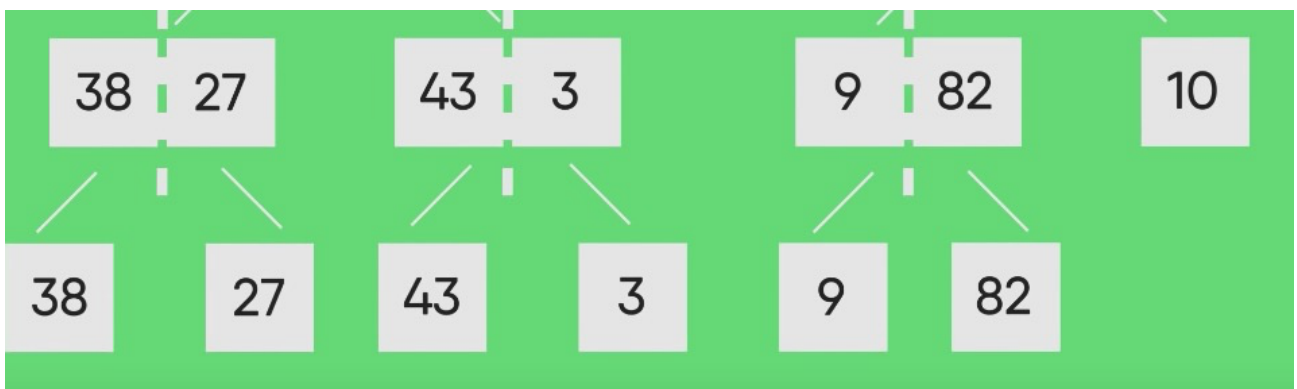
```

```

24     return merge(mergeSort(leftContainer), mergeSort(rightContainer));
25 };
26
27     return mergeSort(arr);
28 }
29
30 console.log(mergeSort([2, 1, 5, 1, 10, 3, 5]));

```

mergesort 比起前面的稍微复杂一点，其特点利用递归和分治的概念来排序，分治的概念可以理解为分开治理。在这里的实际意思为把数组对半拆分一直这样对半拆分直至数组内只有一个元素。当拆分好了之后，我们就开始治理。大家可以用笔画出来，会发现就是树结构，



跳过了部分中间的拆分步骤来到了最后的形态

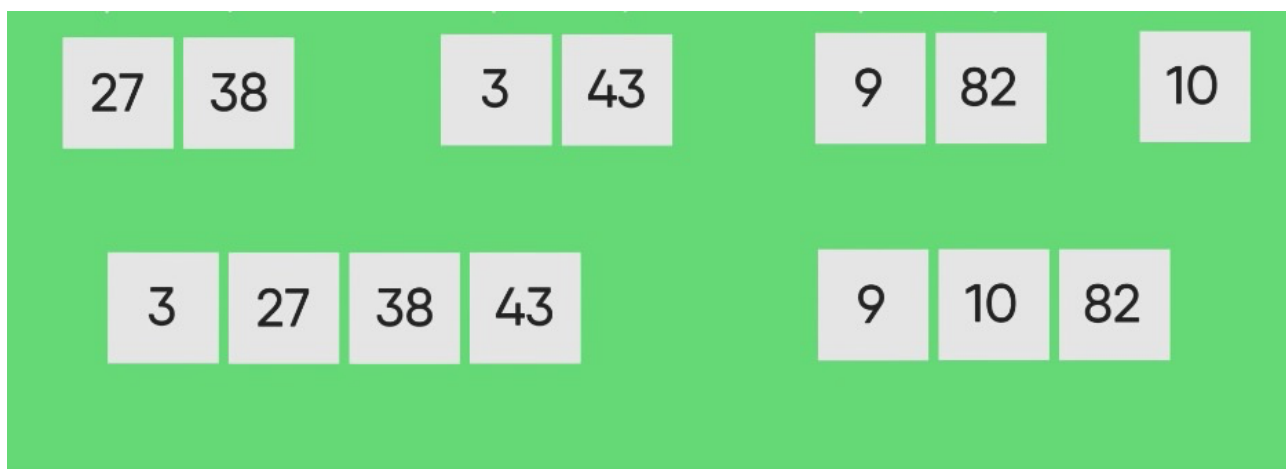
然后递归在这里会结束，因为递归的特性，这里将会开始递归的回溯。回溯的过程就是开始真正排序的过程其逻辑封装在了merge 函数。merge 函数首先会进行比较，小的那个数将会先放入一个第三方数组container, 然后再看同级是否有其他数，如果有，小的那个数所在的数组将会将下一个数和之前大的那个数做比较，以此循环，直至左右2个数组任何一方里面的数全部处理好。这一部分的处理逻辑就是下面这部分代码，

```

1  ...
2  const container = [];
3  while (lp < left.length && rp < right.length) {
4      if (right[rp] < left[lp]) container.push(right[rp++]);
5      else container.push(left[lp++]);
6  }
7  ...

```

当左右2个数组内任意一个元素全部处理好了之后，就会跳出while循环接着把2者所剩余的数依次添加进container，这个时候就是当前递归回溯的这一层里面左边的数组已经排好序，接着同样的逻辑处理这一层里面的右边的数组，



这一层的所有左边和所有右边所在数据都以排序

然后就会回溯一层重复上面的相同逻辑步骤，比较左边数组和右边数组，直至全部完成排序。时间复杂度为 $O(n \log n)$

i mergeSort 因为递归和回溯比较难于具像化理解，需要小伙伴们自己动手多写几遍体会一下方能帮助理解。网上也有更多资料可以查找帮助理解，但是都不是简单的可以理解。如果不会也不用气馁，把上面的几种算法熟记

HeapSort

```
1 function heapSort(arr) {
2   function sort(arr) {
3     for (var i = parseInt(arr.length / 2) - 1; i >= 0; i--) {
4       makeHeap(i, arr, arr.length);
5     }
6     for (var j = arr.length - 1; j > 0; j--) {
7       swap(arr, 0, j);
8       makeHeap(0, arr, j);
9     }
10
11    return arr;
12  }
```

```

13 //实现堆结构
14 function makeHeap(i, arr, length) {
15     let temp = arr[i];
16     for (var k = i * 2 + 1; k < length; k = k * 2 + 1) {
17         if (k + 1 < length && arr[k] < arr[k + 1]) {
18             k++;
19         }
20         if (arr[k] > temp) {
21             arr[i] = arr[k];
22             i = k;
23         } else {
24             break;
25         }
26     }
27     arr[i] = temp;
28 }
29
30 function swap(arr, i, j) {
31     let empty = arr[i];
32     arr[i] = arr[j];
33     arr[j] = empty;
34 }
35
36 return sort(arr);
37 }
38 console.log(heapSort([2, 1, 5, 1, 10, 3, 5]));

```

堆排序的实现方法整理一下

1. 首先需要把数组变成最大堆。即所有的父节点大于或等于自己的子节点。代码里面是从底部往上推，一层一层实现
2. 当完成第一步的时候，堆顶就是当前数组的最大元素，我们把最大元素和数组最末尾元素交换位置，然后去除最大元素，对其他元素接着排序。代码里面是通过swap方法实现的交换
3. 现在的堆顶元素在第二步交换之后已经不是最大元素，即产生了一个新的堆只不过不是最大堆。就这样我们每一次都生成一个最大堆，最后便能得到一个排序好的数组。

堆排序的时间复杂度为 $O(n\log n)$ 。代码不太容易理解，建议画出来。

我们对数组的使用和排序通过3个章节做了整理，但是我们也处在使用的阶段。为了更进一步提高技术和了解数组的常用方法，所以我们在下一章节尝试自己完成几个常用的数组方法。

经典数组原生方法实现

Reduce

reduce 方法让数组元素运行一个开发者传入的reducer函数，然后返回结果。非常简单的理解，我们先使用一下，

```
1 let arr = [1,2,3,4,5,6]
2 console.log(arr.reduce((accumulator,initialVal)=>accumulator+initialVal))
```

上面简单演示使用方法，传入了一个callback函数就是让数组内的所有元素相加。

 顺便说一下，reduce函数接受2个参数，第二个参数为可选的。

那么，现在让我们自己从0来完成这样一个函数

1. 定义一个函数其接受2个参数
2. 如果第一个参数类型不是function则直接抛错
3. 如果调用_reduce的为undefined或者null，则直接抛错
4. 如果调用_reduce的数组长度为0即空数组则直接返回
5. 如果第二个参数提供了且不为undefined, 则让 res 的值为第二个参数的值, , 否则 accumulator的值就是数组的第一个参数的值
6. 使用循环让每一个数组的元素都作为传入的cb函数的参数

```
1 Array.prototype._reduce = function (cb, initialVal) {
2   if (this === null || this === undefined)
3     throw new TypeError('need array');
4   if (Object.prototype.toString.call(cb) !== '[object Function]')
5     throw new TypeError('first argument need a function');
6   let arrLen = this.length >>> 0;
7   if (arrLen === 0)
8     throw new TypeError('Uncaught TypeError: Reduce of empty array is inva
9   let _this = Object(this);
10  let i;
```



```

11   let res =
12     initialVal === undefined ? ((i = 1), this[0]) : ((i = 0), initialVal);
13   for (; i < arrLen; i++) {
14     res = cb.call(undefined, res, this[i])
15   }
16   return res;
17 };
18 console.log([1, 2, 3]._reduce((prev, cur) => prev + cur)); // 6

```

Concat

concat 函数可以让多个数组或者数字组合起来，返回一个新数组，比如

```

1  let arr = [1,2,3]
2  let arr2 = [4,5,6]
3  let arr3 = [10,[11],12]
4
5  console.log(arr.concat(arr2,arr3))

```

```

> let arr = [1,2,3]
   let arr2 = [4,5,6]
   let arr3 = [10,[11],12]

   console.log(arr.concat(arr2,arr3))

▶ (9) [1, 2, 3, 4, 5, 6, 10, Array(1), 12]
< undefined
>

```

那么，我们从0开始完成一个concat函数

1. 我们首先定一个函数，其可以接受0-多个参数，参数的类型可以是数组或者单个值或者2者混合
2. 如果是0个参数，则返回调用concat方法的数组的浅拷贝
3. 如果参数的类型是引用类型，则直接把引用复制进去，即原始数组和创建的新数组都引用同一个元素
4. 如果是普通的值，则直接复制值
5. 返回一个新数组

```

1  Array.prototype._concat = function (...items) {

```

```

2   if (this === null || this === undefined) throw new TypeError('need array
3   if (items.length >>> 0 === 0) {
4       let arr = [...this];
5       return arr;
6   }
7   const isArray = (item) => {
8       return Object.prototype.toString.call(item) === '[object Array]';
9   };
10  items.forEach((item) => {
11      if (isArray(item)) this.push(...item);
12      else this.push(item);
13  });
14
15  return this;
16 };
17 console.log([1, 2, 3]._concat(4, 5, 6));
18 console.log([1, 2, 3]._concat(4, [1, 2, 3], 5, 6));
19 console.log([1, 2, 3]._concat(1, 2, [1, 2, 3, [4, 5, 6, [10, 11, 12]]]));

```

测试一下是否和原生的一样

```

> Array.prototype._concat = function (...items) {
    if (this === null || this === undefined) throw new TypeError('need array');
    if (items.length >>> 0 === 0) {
        let arr = [...this];
        return arr;
    }
    const isArray = (item) => {
        return Object.prototype.toString.call(item) === '[object Array]';
    };
    items.forEach((item) => {
        if (isArray(item)) this.push(...item);
        else this.push(item);
    });

    return this;
};
console.log([1, 2, 3]._concat(4, 5, 6));
console.log([1, 2, 3]._concat(4, [1, 2, 3], 5, 6));
console.log([1, 2, 3]._concat(1, 2, [1, 2, 3, [4, 5, 6, [10, 11, 12]]]));

▶ (6) [1, 2, 3, 4, 5, 6]
▶ (9) [1, 2, 3, 4, 1, 2, 3, 5, 6]
▶ (9) [1, 2, 3, 1, 2, 1, 2, 3, Array(4)]
< undefined
> console.log([1, 2, 3].concat(4, 5, 6));
console.log([1, 2, 3].concat(4, [1, 2, 3], 5, 6));
console.log([1, 2, 3].concat(1, 2, [1, 2, 3, [4, 5, 6, [10, 11, 12]]]));

▶ (6) [1, 2, 3, 4, 5, 6]
▶ (9) [1, 2, 3, 4, 1, 2, 3, 5, 6]
▶ (9) [1, 2, 3, 1, 2, 1, 2, 3, Array(4)]

```

POP

pop 方法取出一个数组的最后一个元素，且返回那个数。该方法已经改变原数组且返回数组的长度已经更新

```
1 let arr = [1, 2, 3];
2 let res = arr.pop();
3 console.log(arr);
4 console.log(res);
```

```
> let arr = [1, 2, 3];
   let res = arr.pop();
   console.log(arr);
   console.log(res);

▶ (2) [1, 2]

3
```

让我们从0来实现一个pop函数吧，

1. 如果调用的是null或者undefined，直接抛出类型错误
2. 如果调用的是一个空数组,返回undefined
3. 如果调用的不是一个数组，抛出类型错误
4. 使用splice拿出最后一个元素
5. splice函数返回一个数组，我们返回该数组的第一个数就是我们想要的

```
1 Array.prototype._pop = function () {
2   if (this === null || this === undefined) throw new TypeError('need array')
3   let len = this.length >>> 0;
4   if (len === 0) return undefined;
5
6   if (Object.prototype.toString.call(this) !== '[object Array]')
7     throw new TypeError('need array');
8
9   return arr.splice(len - 1, 1)[0];
10 };
11 let arr = [1, 2, 3];
12 let res = arr._pop();
13 console.log(arr);
14 console.log(res);
15 console.log(arr.length);
```

```
> Array.prototype._pop = function () {
  if (this === null || this === undefined) throw new TypeError('need array');
  let len = this.length >>> 0;
  if (len === 0) return undefined;

  if (Object.prototype.toString.call(this) !== '[object Array]')
    throw new TypeError('need array');

  return arr.splice(len - 1, 1)[0];
};
let arr = [1, 2, 3];
let res = arr._pop();
console.log(arr);
console.log(res);
console.log(arr.length)

▶ (2) [1, 2]
3
2
```

那么做为练习，请自己实现一下push的方法，push方法的实现非常类似pop。

我们这章不仅尝试了完成原生方法的手写过程，也从这个过程更加了解了其底层的方法也对我们js的知识点有了更多的认识比如this。

那么从下一个章节开始，我们会开始了解browser相关的知识比如browser是如何组织和运行js代码的等等，那么一起加油吧！

js 与 v8引擎

v8 是目前各方面都做的比较好的js引擎，在这里的引擎指的就是js源代码的处理工具。处理的意思就是js代码是让开发者使用的接近人类语言的，但是机器不懂，机器只懂机器码。那么这个之间就会有一个转换的过程，v8就是一个可以完成这个步骤的工具。当然除了v8还有其他的js引擎比如chakra，JavascriptCore 等等。我们本文将会专注在v8。

一般比较主要的编程语言有2种类型，解释型比如js，编译型比如java。

在运行速度上面，编译型的语言是更加效率的，原理是因为在运行之前代码已经完成了编译，源代码转换成了机器码，即在运行的时候无需再转换成机器码。相对的解释型就是在运行的时候进行源代码和机器码的转换，js代码以文本格式传给browser，然后再变解释边运行。

那么解释型语言存在的目的是什么呢？

相信前端开发人员在使用js的时候感觉非常畅快不受一定的限制相对于使用ts，有没有？这也是解释型语言的优势之一，即支持动态类型。使用一个变量不需要先给这个变量指定类型，这样代码编写的时候相比较java等编译型语言效率就会高很多。再有，解释型语言是平台独立，即一个程序可以在不同的平台运行比如php程序可以在linux 和 windows，从Arm架构和x86都可以。但是c，c++ 在不同平台下面可能需要重新编译的，本质上可以理解它们代码是可以跨平台的，编译后的代码不能跨平台，所以需要重新编译。

上面的知识是为了下面的内容做铺垫，那么我们现在来解释一下v8是如何运行js的。

v8 把js运行可以分为4个步骤

1. 生成ast，解析阶段
2. ast转成字节码，转换阶段
3. 根据生成的字节码优化编译，生成机器码
4. 内存空间回收，把程序中不需要的内存进行回收

ast生成

ast的生成可以想象为把你的代码以描述的方式描述出来，里面关键的2个地方需要理解，即词法分析和语法分析。

分析词法，比如 `var a = 1`，词法分析会把这句语言分析为 `var, a, =, 1`。接着通过语法分析开始转换之后的样子类似于

```
1  {
2    "type": "Program",
3    "start": 0,
4    "end": 10,
5    "body": [
6      {
7        "type": "VariableDeclaration",
8        "start": 0,
9        "end": 10,
10       "declarations": [
11         {
12           "type": "VariableDeclarator",
13           "start": 4,
14           "end": 9,
15           "id": {
16             "type": "Identifier",
17             "start": 4,
18             "end": 5,
19             "name": "a"
20           },
21           "init": {
22             "type": "Literal",
23             "start": 8,
24             "end": 9,
25             "value": 1,
26             "raw": "1"
27           }
28         }
29       ],
30       "kind": "var"
31     }
32   ],
33   "sourceType": "module"
34 }
```

那么生成ast可以为我们做什么呢？比如，es6语法转换成其他版本就可以通过ast来转换其原理就是通过ast对代码的描述来生成其他版本的语法。因为ast只是描述了代码是什么样

的，所以具体的实现可以由我们自己完成。babel等让代码自己转换的实现原理也就是通过这个来完成。

生成字节码

字节码是在ast与机器码中间的一个概念，那么需要有这个步骤呢？因为内存问题。如果直接生成机器码那么机器码的体积会非常大，因为browser都有缓存功能，即下一次使用可以直接从缓存里面获取增加效率。可是内存的空间是有限的，如果想缓存的内容超出了可以缓存的空间则会放弃缓存。解决方法就是先生成字节码。字节码通过一定的编码可以让体积更紧凑即占用的内存空间会少可以缓存更多的代码。

生成优化后的机器码

字节码转换生成机器码之前会进行优化，这个优化的工作会交给turbofan编译器。这里的优化比如如果代码其中一部分会重复的运行那么可以让代码提前编译好，下次再运行此段代码时候不需要再解释而是直接运行了。

上面的步骤就是v8拿到js文件之后处理的过程和步骤，希望本文让你对v8有进一步的认识。

我们下一章讨论一下v8的微宏任务。

宏微任务 + async/await/promise

js是单线程的，设计成单线程也是因为可以免去一部分的复杂度，这里的复杂度指的是不会产生dom的冲突。这句话的解释就是js是可以操作dom的，如果js是多线程那么最终的dom可能会产生互斥。如果上锁的概念就完全让js本身的难度和复杂度上升了一个档次，js作者当初也只是把它作为一个脚本语言。

那么如果是单线程的，是不是就会有阻塞的问题呢？确实，单线程容易产生阻塞问题，即阻塞后续的任务运行如果当前的任务没有完成。为了解决这个问题，js出现了异步任务。异步任务不会阻塞js主线程的任务，而是当异步任务完成的时候会按照一定的顺序运行。这一章里面讲解的一个重点就是这个规则是咋么样的。

先上一段代码，通过代码的方式来了解

```
1 console.log('script start');
2 new Promise((resovle, reject)=>{
3   console.log('promise');
4   resovle()
5 }).then(()=>{
6   console.log('then');
7 })
8 console.log('script end')
```

大家花3分钟自测一下运行的顺序是什么？

先上答案，大家看看是否自己做对了。如果做对了，那么恭喜，最基础的【宏微任务 + Promise】的运行机制掌握了。

- ☒ script start
- ☒ promise
- ☒ script end
- ☒ then

接下来分析一下，首先打印 script start，这个相信没有问题。接下来打印了promise，可能有人会疑惑，promise不是异步任务么，这里应该打印 script end。如果你也有这个疑惑，那么请记住，promsie规范里面，promise的构造器里的代码是同步任务，而then里面的

callback才是异步微任务，所以这里先打印了promise。接着打印了script end，正如我们已经描述了，then里面的callback是异步微任务，所以这个微任务会被加载进当前宏任务的微任务队列中。

- ① 每一个宏任务会有一个对应的微任务队列，一一对应的关系。当前宏任务内的微任务不会跑进其他的宏任务里的微任务队列里面。

微任务的运行时机是当前宏任务队列里面已经没有其他任务的时候，那么就会按照先进先出的顺序让微任务队列里面的任务一个一个顺序运行。所以最后打印then。

- ① 当前宏任务的运行时间也会被微任务运行的时间影响。即如果当前的微任务没有完成则不会接着处理下一个宏任务。

那么，升级一下题目难度，

```
1  async function abc() {
2    console.log('async abc');
3    await def();
4    setTimeout(() => {
5      console.log('async abc timeout');
6    }, 100);
7    console.log('async abc end');
8  }
9  async function def() {
10   console.log('async def');
11 }
12 console.log('script starts');
13 abc();
14 setTimeout(() => {
15   new Promise((resolve, reject) => {
16     console.log('promise in timeout');
17   });
18   console.log('timeout end');
19 }, 100);
20 setTimeout(() => {
21   console.log('timeout with no promise');
22 }, 0);
```

上面的题目给自己5分钟时间考虑一下输出的是什么？

如果不能给出答案，那么我们来讲解一下运行的机制，

- ☒ script starts
- ☒ async abc

上面这2个会先输出，这个相信基本都会，那么接下来运行在这里的时候我们遇见了await，所以必须等待await后面的语句运行好。await 可以想象成generator 里的 yield，即让出当前线程。所以会接着打印

- ☒ async def

当await的任务完成之后，会回到原来的线程继续运行，那么接下来是否会运行 await def() 后面的代码？答案可能很多人会出乎意料，不会立即运行，而是把后面的代码放进了异步微任务队列。为什么呢？async await 其实就是generator + promise的语法糖。前面刚刚描述过 promise 的then 是异步微任务，这里的await def() 代码之后的语句等价于promise 里的 then 异步微任务。

那么，放入异步微任务之后，我们接着运行后续代码，这时我们来到了

```
1  setTimeout(() => {
2    new Promise((resolve, reject) => {
3      console.log('promise in timeout');
4    });
5    console.log('timeout end');
6  }, 100);
```

很明显setTimeout有一个100ms的缓冲时间，即等待100毫秒再放入宏任务队列。接着往下运行，我们遇见了

```
1  setTimeout(() => {
2    console.log('timeout with no promise');
3  }, 0);
```

这个和上面的区别是没有任何缓冲时间，即立即放入宏任务队列，所以虽然代码顺序是这个更加靠后但是放入宏任务队列的时间是比之前那个100ms缓冲时间更快的。

这个时候明面上的代码基本都走过了，那么我们会开始运行当前宏任务对应的异步微任务队列。即回到之前的`async abc`函数里面的 `await def` 后面的代码，

```
1  async function abc() {
2    console.log('async abc');
3    await def();
4
5    // 下面这部分代码
6    setTimeout(() => {
7      console.log('async abc timeout');
8    }, 100);
9    console.log('async abc end');
10 }
```

首先会有一个`setTimeout`和100ms的缓冲时间，这个`setTimeout`也会在100ms之后加入宏任务队列，所以这里会运行

☑ `async abc end`

那么当前宏任务完成了，接着我们开始运行下一个宏任务。运行之前我们搞清楚当前宏任务队列之中有哪几个宏任务，以及顺序如何排列的。

一共我们有3个宏任务，分别是2个有100ms缓冲的和没有时间缓冲的。先后顺序就是

```
1  setTimeout(() => {
2    console.log('timeout with no promise');
3  }, 0);
4
5  setTimeout(() => {
6    new Promise((resolve, reject) => {
7      console.log('promise in timeout');
8    });
9    console.log('timeout end');
10 }, 100);
11
12 setTimeout(() => {
13   console.log('async abc timeout');
```

虽然后两个都有100ms的相同缓冲时间，但是先开始缓冲的是带有Promise的因为这个是在当前宏任务里的任务而后面那个确是当前宏任务对应的微任务队列里面的。

最后整理一下完整输出顺序

- ☒ scripts starts
- ☒ async abc
- ☒ async def
- ☒ async abc end
- ☒ timeout with no promise
- ☒ promise in timeout
- ☒ timeout end
- ☒ async abc timeout

相信通过这2个题目可以让你对宏微任务 + async/await/promise的组合面试题更有信心

数据结构 - Set, Map

set 与 map 是js中默认提供的数据结构，使用的好则可以在开发过程之中帮 助我们把复杂的问题简单化。深入讲解这2个数据机构之前，我们先来了解一下如何使用。

Map

```
1  const map1 = new Map();
2
3  map1.set('a', 1);
4  map1.set('b', 2);
5  map1.set('c', 3);
6
7  console.log(map1.get('a'));
8  // expected output: 1
9
10 map1.set('a', 97);
11
12 console.log(map1.get('a'));
13 // expected output: 97
14
15 console.log(map1.size);
16 // expected output: 3
17
18 map1.delete('b');
19
20 console.log(map1.size);
21 // expected output: 2
```

Set

```
1  const mySet1 = new Set()
2
3  mySet1.add(1)           // Set [ 1 ]
4  mySet1.add(5)           // Set [ 1, 5 ]
5  mySet1.add(5)           // Set [ 1, 5 ]
6  mySet1.add('some text') // Set [ 1, 5, 'some text' ]
7  const o = {a: 1, b: 2}
8  mySet1.add(o)
9
10 mySet1.add({a: 1, b: 2}) // o is referencing a different object, so this
```

```

11
12 mySet1.has(1)           // true
13 mySet1.has(3)           // false, since 3 has not been added to the set
14 mySet1.has(5)           // true
15 mySet1.has(Math.sqrt(25)) // true
16 mySet1.has('Some Text'.toLowerCase()) // true
17 mySet1.has(o)           // true
18
19 mySet1.size              // 5
20
21 mySet1.delete(5)        // removes 5 from the set
22 mySet1.has(5)           // false, 5 has been removed
23
24 mySet1.size              // 4, since we just removed one value
25
26 console.log(mySet1)
27 // logs Set(4) [ 1, "some text", {...}, {...} ] in Firefox
28 // logs Set(4) { 1, "some text", {...}, {...} } in Chrome

```

上面是mdn的使用方法介绍。那么，我想你们最感兴趣的部分是为什么需要有这2种数据结构，以及在什么场景下使用这2种数据结构。

首先，我们来一个算法题来讲解map，不难。

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 和为目标值 `target` 的那 两个 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

示例 1：

输入：`nums = [2,7,11,15]`, `target = 9` 输出：`[0,1]` 解释：因为 `nums[0] + nums[1] == 9`，返回 `[0, 1]`。

示例 2：

输入：`nums = [3,2,4]`, `target = 6` 输出：`[1,2]`

示例 3：

输入：nums = [3,3], target = 6 输出：[0,1]

那么我们可以非常容易的想到使用数组循环的方式逐个确认，我们先来实现一下

```
1 function sum(arr, target) {
2   let arrLen = arr.length;
3   for (let i = 0; i < arrLen; i++) {
4     for (let j = i + 1; j < arrLen; j++) {
5       const sum = arr[i] + arr[j];
6       if (sum === target) {
7         return [i, j];
8       }
9     }
10  }
11  return [];
12 }
13 console.log(twoSum([3, 2, 4], 6));
14 console.log(twoSum([3, 3], 6));
```

非常简单就能使用暴力解法解出此题，可是就像你想的那样，这样的时间复杂度为 $O(n^2)$ ，如果该数组的元素有1000000万个那么时间上面的消耗不可小看。

那么，是否可以有更优解法呢。答案是肯定。使用map的数据结构可以让时间复杂度降低，原理是什么呢？

map的每一个元素由一个key和value组成 (key, value)。那么我们在暴力破解的时候，之所以时间消耗比较大因为数组的每一个数都需要再循环数组一次确认是否有2个数的和等于target。使用map我们可以使用map.get(key) 确认是否有我们想要的数就好了，不需要向数组那样每一次都循环一遍。那么这里的key就可以是arr数组的元素，比如 map.set(arr[0], 0)。这里的arr[0] 就是数组arr的第一个元素，第二个参数就可以是数组arr的索引index。

那么每一次arr数组取出一个数，然后比对map里面是否有另一个我们想要的数让2数的和等于target。如果没有，则把当前的数存进map, 等待下一次使用。如果有，直接返回那个数的索引。代码实现，


```
1 function sum(arr, target) {
```

```

2   let arrLen = arr.length;
3   let map = new Map();
4   map.set(arr[0], 0);
5   for (let i = 1; i < arrLen; i++) {
6       if (map.has(target - arr[i])) {
7           return [map.get(target - arr[i]), i];
8       }
9       map.set(arr[i], i);
10  }
11  return [];
12 }
13 console.log(twoSum([3, 2, 4], 6));
14 console.log(twoSum([3, 3], 6));

```

对比暴力解法，我们只有一个for循环，时间复杂度从 $O(n^2)$ -> $O(n)$ 。现在是否对数据结构能为我们工作中带来什么好处有更深的体会了吧。

 本题是为了演示map的使用，所以没有对边界做检查。比如是否传入的是数组，数组的长度是否在允许的范围之内等等。

上面演示的题目来自于力扣（LeetCode）著作权归领扣网络所有。

我们接着来看看set的数据结构。set很类似于数组，但是set里面是不会有重复的值。那么简单的来说就是可以让数组把重复的元素去除。

表面上的理解非常简单。但是实际的使用场景确有很多可能性。比如

```

1  // 数组去重
2  let arr = [1, 1, 2, 3];
3  let unique = [... new Set(arr)];
4
5  let a = new Set([1, 2, 3]);
6  let b = new Set([4, 3, 2]);
7
8  // 并集
9  let union = [...new Set([...a, ...b])];
10
11 // 交集
12 let intersect = [...new Set([...a].filter(x => b.has(x)))];

```



```
13  
14 // 差集  
15 let difference = Array.from(new Set([...a].filter(x => !b.has(x))));
```

利用去重的特点可以简单的完成上面的操作。如果没有set，确实也是可以使用数组完成的。类似于我们刚刚才优化过的算法题，时间效率上面消耗会多，因为每一次都需要判断是否数组内有重复的元素等于需要每次都再遍历一次数组。

本节通过题目和使用场景对javascript的map和set做了讲解，希望能够让小伙伴们对数据结构在实际应用场景有更深的体会。数据结构的精通是一个需要时间来沉淀的过程。