

JavaScript进阶

1 内置类型

- JS 中分为七种内置类型，七种内置类型又分为两大类型：基本类型和对象（Object）。
- 基本类型有六种： `null`，`undefined`，`boolean`，`number`，`string`，`symbol`。
- 其中 JS 的数字类型是浮点类型的，没有整型。并且浮点类型基于 IEEE 754 标准实现，在使用中会遇到某些 Bug。NaN 也属于 `number` 类型，并且 NaN 不等于自身。
- 对于基本类型来说，如果使用字面量的方式，那么这个变量只是个字面量，只有在必要的时候才会转换为对应的类型。

```
1 let a = 111 // 这只是字面量，不是 number 类型
2 a.toString() // 使用时候才会转换为对象类型
```

对象（Object）是引用类型，在使用过程中会遇到浅拷贝和深拷贝的问题。

```
1 let a = { name: 'FE' }
2 let b = a
3 b.name = 'EF'
4 console.log(a.name) // EF
```

2 Typeof

`typeof` 对于基本类型，除了 `null` 都可以显示正确的类型

```
1 typeof 1 // 'number'
2 typeof '1' // 'string'
3 typeof undefined // 'undefined'
4 typeof true // 'boolean'
5 typeof Symbol() // 'symbol'
6 typeof b // b 没有声明，但是还会显示 undefined
```

```
1 typeof ` 对于对象，除了函数都会显示 `object
```

```
1 | typeof [] // 'object'
2 | typeof {} // 'object'
3 | typeof console.log // 'function'
```

对于 `null` 来说，虽然它是基本类型，但是会显示 `object`，这是一个存在很久的 Bug

```
1 | typeof null // 'object'
```

PS：为什么会出现这种情况呢？因为在 JS 的最初版本中，使用的是 32 位系统，为了性能考虑使用低位存储了变量的类型信息，000 开头代表是对象，然而 `null` 表示为全零，所以将它错误的判断为 `object`。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来。

- 如果我们想获得一个变量的正确类型，可以通过 `Object.prototype.toString.call(xx)`。这样我们就可以获得类似 `[object Type]` 的字符串

```
1 | let a
2 | // 我们也可以这样判断 undefined
3 | a === undefined
4 | // 但是 undefined 不是保留字，能够在低版本浏览器被赋值
5 | let undefined = 1
6 | // 这样判断就会出错
7 | // 所以可以用下面的方式来判断，并且代码量更少
8 | // 因为 void 后面随便跟上一个组成表达式
9 | // 返回就是 undefined
10 | a === void 0
```

3 类型转换

转Boolean

在条件判断时，除了 `undefined`，`null`，`false`，`NaN`，`''`，`0`，`-0`，其他所有值都转为 `true`，包括所有对象

对象转基本类型

对象在转换基本类型时，首先会调用 `valueOf` 然后调用 `toString`。并且这两个方法你是可以重写的

```

1 let a = {
2     valueOf() {
3         return 0
4     }
5 }

```

四则运算符

只有当加法运算时，其中一方是字符串类型，就会把另一个也转为字符串类型。其他运算只要其中一方是数字，那么另一方就转为数字。并且加法运算会触发三种类型转换：将值转换为原始值，转换为数字，转换为字符串

```

1 1 + '1' // '11'
2 2 * '2' // 4
3 [1, 2] + [2, 1] // '1,22,1'
4 // [1, 2].toString() -> '1,2'
5 // [2, 1].toString() -> '2,1'
6 // '1,2' + '2,1' = '1,22,1'

```

对于加号需要注意这个表达式 `'a' + + 'b'`

```

1 'a' + + 'b' // -> "aNaN"
2 // 因为 + 'b' -> NaN
3 // 你也许在一些代码中看到过 + '1' -> 1

```

== 操作符

这里来解析一道题目 `[] == ![] // -> true`，下面是这个表达式为何为 `true` 的步骤

```

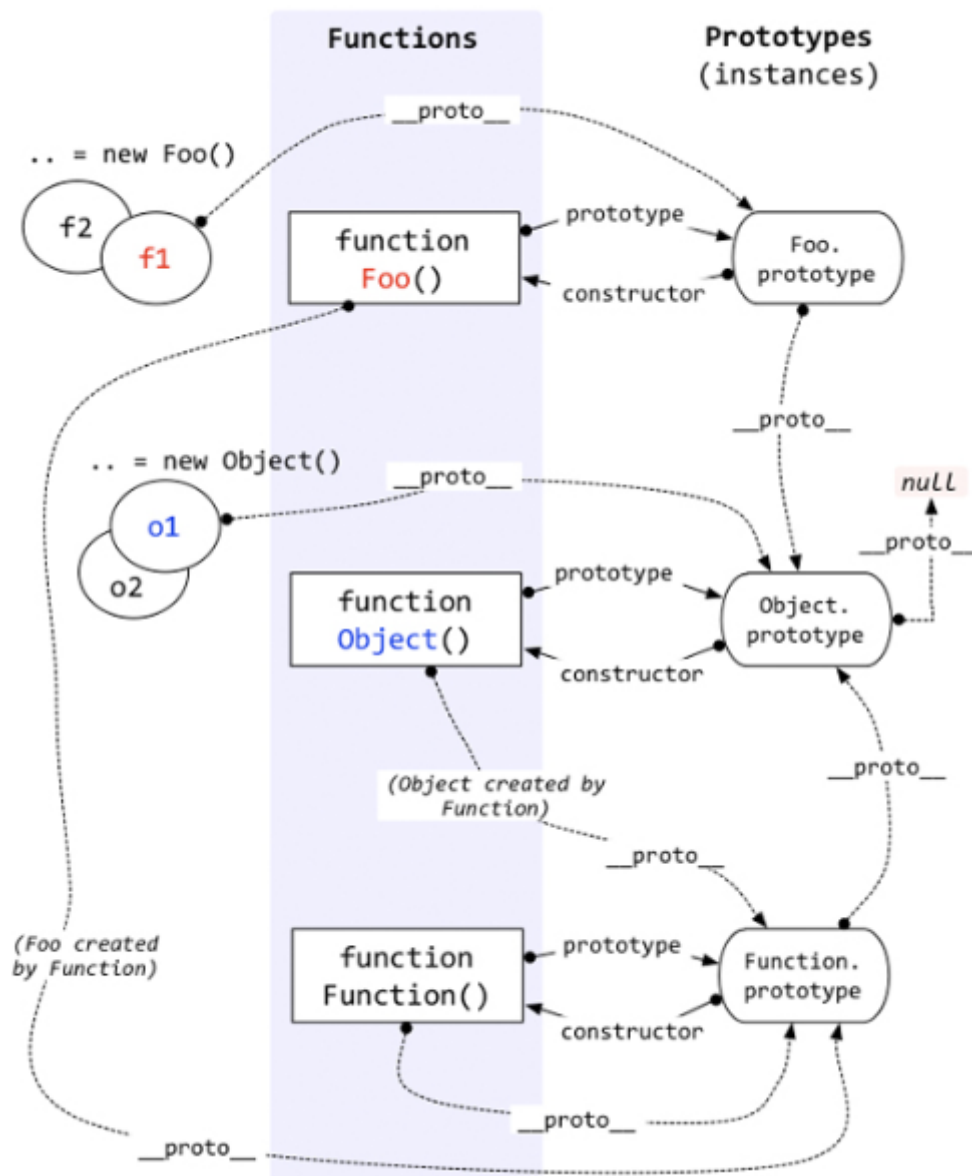
1 // [] 转成 true，然后取反变成 false
2 [] == false
3 // 根据第 8 条得出
4 [] == ToNumber(false)
5 [] == 0
6 // 根据第 10 条得出
7 ToPrimitive([]) == 0
8 // [].toString() -> ''
9 '' == 0
10 // 根据第 6 条得出
11 0 == 0 // -> true

```

比较运算符

- 如果是对象，就通过 `toPrimitive` 转换对象
- 如果是字符串，就通过 `unicode` 字符索引来比较

4 原型



- 每个函数都有 `prototype` 属性，除了 `Function.prototype.bind()`，该属性指向原型。
- 每个对象都有 `__proto__` 属性，指向了创建该对象的构造函数的原型。其实这个属性指向了 `[[prototype]]`，但是 `[[prototype]]` 是内部属性，我们并不能访问到，所以使用 `__proto__` 来访问。
- 对象可以通过 `__proto__` 来寻找不属于该对象的属性，`__proto__` 将对象连接起来组成了原型链

5 new

- 生成了一个对象
- 链接到原型
- 绑定 `this`
- 返回新对象

在调用 `new` 的过程中会发生以上四件事情，我们也可以试着来自己实现一个 `new`

```
1 function create() {
2     // 创建一个空的对象
3     let obj = new Object()
4     // 获得构造函数
5     let Con = [].shift.call(arguments)
6     // 链接到原型
7     obj.__proto__ = Con.prototype
8     // 绑定 this, 执行构造函数
9     let result = Con.apply(obj, arguments)
10    // 确保 new 出来的是个对象
11    return typeof result === 'object' ? result : obj
12 }
```

6 instanceof

- 1 `instanceof`` 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 ``prototype`

我们也可以试着实现一下 `instanceof`

```
1 function instanceof(left, right) {
2     // 获得类型的原型
3     let prototype = right.prototype
4     // 获得对象的原型
5     left = left.__proto__
6     // 判断对象的类型是否等于类型的原型
7     while (true) {
8         if (left === null)
9             return false
10        if (prototype === left)
11            return true
12        left = left.__proto__
13    }
14 }
```

7 this

```
1 function foo() {
2     console.log(this.a)
3 }
4 var a = 1
5 foo()
6
7 var obj = {
8     a: 2,
9     foo: foo
10 }
11 obj.foo()
12
13 // 以上两者情况 `this` 只依赖于调用函数前的对象，优先级是第二个情况大于第一个情况
14
15 // 以下情况是优先级最高的，`this` 只会绑定在 `c` 上，不会被任何方式修改 `this` 指向
16 var c = new foo()
17 c.a = 3
18 console.log(c.a)
19
20 // 还有种就是利用 call, apply, bind 改变 this，这个优先级仅次于 new
```

看看箭头函数中的 this

```
1 function a() {
2     return () => {
3         return () => {
4             console.log(this)
5         }
6     }
7 }
8 console.log(a()()())
```

箭头函数其实是没有 `this` 的，这个函数中的 `this` 只取决于他外面的第一个不是箭头函数的函数的 `this`。在这个例子中，因为调用 `a` 符合前面代码中的第一个情况，所以 `this` 是 `window`。并且 `this` 一旦绑定了上下文，就不会被任何代码改变

8 执行上下文

当执行 JS 代码时，会产生三种执行上下文

- 全局执行上下文
- 函数执行上下文
- `eval` 执行上下文

每个执行上下文中都有三个重要的属性

- 变量对象 (`vo`)，包含变量、函数声明和函数的形参，该属性只能在全局上下文中访问
- 作用域链 (`JS` 采用词法作用域，也就是说变量的作用域是在定义时就决定了)
- `this`

```
1 var a = 10
2 function foo(i) {
3     var b = 20
4 }
5 foo()
```

对于上述代码，执行栈中有两个上下文：全局上下文和函数 `foo` 上下文。

```
1 stack = [
2     globalContext,
3     fooContext
4 ]
```

对于全局上下文来说，`vo` 大概是这样的

```
1 globalContext.v0 === globe
2 globalContext.v0 = {
3     a: undefined,
4     foo: <Function>,
5 }
```

对于函数 `foo` 来说，`vo` 不能访问，只能访问到活动对象 (`AO`)

```

1 fooContext.VO === foo.AO
2 fooContext.AO {
3   i: undefined,
4   b: undefined,
5   arguments: <>
6 }
7 // arguments 是函数独有的对象(箭头函数没有)
8 // 该对象是一个伪数组，有 `length` 属性且可以通过下标访问元素
9 // 该对象中的 `callee` 属性代表函数本身
10 // `caller` 属性代表函数的调用者

```

对于作用域链，可以把它理解成包含自身变量对象和上级变量对象的列表，通过 `[[Scope]]` 属性查找上级变量

```

1 fooContext. [[Scope]] = [
2   globalContext.VO
3 ]
4 fooContext.Scope = fooContext. [[Scope]] + fooContext.VO
5 fooContext.Scope = [
6   fooContext.VO,
7   globalContext.VO
8 ]

```

接下来让我们看一个老生常谈的例子，`var`

```

1 b() // call b
2 console.log(a) // undefined
3
4 var a = 'Hello world'
5
6 function b() {
7   console.log('call b')
8 }

```

想必以上的输出大家肯定都已经明白了，这是因为函数和变量提升的原因。通常提升的解释是说将声明的代码移动到了顶部，这其实没有什么错误，便于大家理解。但是更准确的解释应该是：在生成执行上下文时，会有两个阶段。第一个阶段是创建的阶段（具体步骤是创建 `VO`），`JS` 解释器会找出需要提升的变量和函数，并且给他们提前在内存中开辟好空间，函数的话会将整个函数存入内存中，变量只声明并且赋值为 `undefined`，所以在第二个阶段，也就是代码执行阶段，我们可以直接提前使用。

- 在提升的过程中，相同的函数会覆盖上一个函数，并且函数优先于变量提升


```

1 b() // call b second
2
3 function b() {
4     console.log('call b fist')
5 }
6 function b() {
7     console.log('call b second')
8 }
9 var b = 'Hello world'

```

`var` 会产生很多错误，所以在 ES6 中引入了 `let`。`let` 不能在声明前使用，但是这并不是常说的 `let` 不会提升，`let` 提升了声明但没有赋值，因为临时死区导致了并不能在声明前使用。

- 对于非匿名的立即执行函数需要注意以下一点

```

1 var foo = 1
2 (function foo() {
3     foo = 10
4     console.log(foo)
5 })() // -> f foo() { foo = 10 ; console.log(foo) }

```

因为当 JS 解释器在遇到非匿名的立即执行函数时，会创建一个辅助的特定对象，然后将函数名称作为这个对象的属性，因此函数内部才可以访问到 `foo`，但是这个值又是只读的，所以对它的赋值并不生效，所以打印的结果还是这个函数，并且外部的值也没有发生更改。

```

1 specialObject = {};
2
3 Scope = specialObject + Scope;
4
5 foo = new FunctionExpression;
6 foo.[[Scope]] = Scope;
7 specialObject.foo = foo; // {DontDelete}, {ReadOnly}
8
9 delete Scope[0]; // remove specialObject from the front of
  scope chain

```

9 闭包

闭包的定义很简单：函数 A 返回了一个函数 B，并且函数 B 中使用了函数 A 的变量，函数 B 就被称为闭包。

```

1 function A() {
2   let a = 1
3   function B() {
4     console.log(a)
5   }
6   return B
7 }

```

你是否会疑惑，为什么函数 A 已经弹出调用栈了，为什么函数 B 还能引用到函数 A 中的变量。因为函数 A 中的变量这时候是存储在堆上的。现在的 JS 引擎可以通过逃逸分析辨别出哪些变量需要存储在堆上，哪些需要存储在栈上。

经典面试题，循环中使用闭包解决 var 定义函数的问题

```

1 for ( var i=1; i<=5; i++) {
2   setTimeout( function timer() {
3     console.log( i );
4   }, i*1000 );
5 }

```

- 首先因为 `setTimeout` 是个异步函数，所有会先把循环全部执行完毕，这时候 `i` 就是 6 了，所以会输出一堆 6。
- 解决办法两种，第一种使用闭包

```

1 for (var i = 1; i <= 5; i++) {
2   (function(j) {
3     setTimeout(function timer() {
4       console.log(j);
5     }, j * 1000);
6   })(i);
7 }

```

- 第二种就是使用 `setTimeout` 的第三个参数

```

1 for ( var i=1; i<=5; i++) {
2   setTimeout( function timer(j) {
3     console.log( j );
4   }, i*1000, i);
5 }

```

第三种就是使用 `let` 定义 `i` 了

```

1 for ( let i=1; i<=5; i++) {
2     setTimeout( function timer() {
3         console.log( i );
4     }, i*1000 );
5 }

```

因为对于 `let` 来说，他会创建一个块级作用域，相当于

```

1 { // 形成块级作用域
2     let i = 0
3     {
4         let ii = i
5         setTimeout( function timer() {
6             console.log( i );
7         }, i*1000 );
8     }
9     i++
10    {
11        let ii = i
12    }
13    i++
14    {
15        let ii = i
16    }
17    ...
18 }

```

10 深浅拷贝

```

1 let a = {
2     age : 1
3 }
4 let b = a
5 a.age = 2
6 console.log(b.age) // 2

```

- 从上述例子中我们可以发现，如果给一个变量赋值一个对象，那么两者的值会是同一个引用，其中一方改变，另一方也会相应改变。
- 通常在开发中我们不希望出现这样的问题，我们可以使用浅拷贝来解决这个问题

浅拷贝

首先可以通过 `Object.assign` 来解决这个问题

```
1 let a = {
2   age: 1
3 }
4 let b = Object.assign({}, a)
5 a.age = 2
6 console.log(b.age) // 1
```

当然我们也可以通过展开运算符 `(...)` 来解决

```
1 let a = {
2   age: 1
3 }
4 let b = {...a}
5 a.age = 2
6 console.log(b.age) // 1
```

通常浅拷贝就能解决大部分问题了，但是当我们遇到如下情况就需要使用到深拷贝了

```
1 let a = {
2   age: 1,
3   jobs: {
4     first: 'FE'
5   }
6 }
7 let b = {...a}
8 a.jobs.first = 'native'
9 console.log(b.jobs.first) // native
```

浅拷贝只解决了第一层的问题，如果接下去的值中还有对象的话，那么就又回到刚开始的话题了，两者享有相同的引用。要解决这个问题，我们需要引入深拷

深拷贝

这个问题通常可以通过 `JSON.parse(JSON.stringify(object))` 来解决

```

1 let a = {
2   age: 1,
3   jobs: {
4     first: 'FE'
5   }
6 }
7 let b = JSON.parse(JSON.stringify(a))
8 a.jobs.first = 'native'
9 console.log(b.jobs.first) // FE

```

但是该方法也是有局限性的：

- 会忽略 `undefined`
- 不能序列化函数
- 不能解决循环引用的对象

```

1 let obj = {
2   a: 1,
3   b: {
4     c: 2,
5     d: 3,
6   },
7 }
8 obj.c = obj.b
9 obj.e = obj.a
10 obj.b.c = obj.c
11 obj.b.d = obj.b
12 obj.b.e = obj.b.c
13 let newObj = JSON.parse(JSON.stringify(obj))
14 console.log(newObj)

```

如果你有这么一个循环引用对象，你会发现你 cannot 通过该方法深拷贝

- 在遇到函数或者 `undefined` 的时候，该对象也不能正常的序列化

```

1 let a = {
2   age: undefined,
3   jobs: function() {},
4   name: 'poetries'
5 }
6 let b = JSON.parse(JSON.stringify(a))
7 console.log(b) // {name: "poetries"}

```

- 你会发现在上述情况中，该方法会忽略掉函数和 `undefined`。

- 但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题，并且该函数是内置函数中处理深拷贝性能最快的。当然如果你的数据中含有以上三种情况下，可以使用 `lodash` 的深拷贝函数。

11 模块化

在有 `Babel` 的情况下，我们可以直接使用 `ES6` 的模块化

```
1 // file a.js
2 export function a() {}
3 export function b() {}
4 // file b.js
5 export default function() {}
6
7 import {a, b} from './a.js'
8 import xxx from './b.js'
```

CommonJS

`CommonJS` 是 `Node` 独有的规范，浏览器中使用就需要用到 `Browserify` 解析了。

```
1 // a.js
2 module.exports = {
3   a: 1
4 }
5 // or
6 exports.a = 1
7
8 // b.js
9 var module = require('./a.js')
10 module.a // -> log 1
```

在上述代码中，`module.exports` 和 `exports` 很容易混淆，让我们来看看大致内部实现

```
1 var module = require('./a.js')
2 module.a
3 // 这里其实就是包装了一层立即执行函数，这样就不会污染全局变量了，
4 // 重要的是 module 这里，module 是 Node 独有的一个变量
5 module.exports = {
6   a: 1
7 }
8 // 基本实现
9 var module = {
```

```

10   exports: {} // exports 就是个空对象
11 }
12 // 这个是因为 exports 和 module.exports 用法相似的原因
13 var exports = module.exports
14 var load = function (module) {
15     // 导出的东西
16     var a = 1
17     module.exports = a
18     return module.exports
19 };

```

再来说说 `module.exports` 和 `exports`，用法其实是相似的，但是不能对 `exports` 直接赋值，不会有任何效果。

对于 `CommonJS` 和 `ES6` 中的模块化的两者区别是：

- 前者支持动态导入，也就是 `require(`${path}/xx.js`)`，后者目前不支持，但是已有提案，前者是同步导入，因为用于服务端，文件都在本地，同步导入即使卡住主线程影响也不大。
- 而后者是异步导入，因为用于浏览器，需要下载文件，如果也采用同步导入会对渲染有很大影响
- 前者在导出时都是值拷贝，就算导出的值变了，导入的值也不会改变，所以如果想更新值，必须重新导入一次。
- 但是后者采用实时绑定的方式，导入导出的值都指向同一个内存地址，所以导入值会跟随导出值变化
- 后者会编译成 `require/exports` 来执行的

AMD

AMD 是由 `RequireJS` 提出的

```

1  // AMD
2  define(['./a', './b'], function(a, b) {
3      a.do()
4      b.do()
5  })
6  define(function(require, exports, module) {
7      var a = require('./a')
8      a.doSomething()
9      var b = require('./b')
10     b.doSomething()
11 })

```

12 防抖

你是否在日常开发中遇到一个问题，在滚动事件中需要做个复杂计算或者实现一个按钮的防二次点击操作。

- 这些需求都可以通过函数防抖动来实现。尤其是第一个需求，如果在频繁的事件回调中做复杂计算，很有可能导致页面卡顿，不如将多次计算合并为一次计算，只在一个精确点做操作
- PS：防抖和节流的作用都是防止函数多次调用。区别在于，假设一个用户一直触发这个函数，且每次触发函数的间隔小于 `wait`，防抖的情况下只会调用一次，而节流的情况会每隔一定时间（参数 `wait`）调用函数

```
1 // 这个是用来获取当前时间戳的
2 function now() {
3   return +new Date()
4 }
5 /**
6  * 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
7  *
8  * @param {function} func      回调函数
9  * @param {number}   wait      表示时间窗口的间隔
10  * @param {boolean}  immediate  设置为true时，是否立即调用函数
11  * @return {function}          返回客户调用函数
12  */
13 function debounce (func, wait = 50, immediate = true) {
14   let timer, context, args
15
16   // 延迟执行函数
17   const later = () => setTimeout(() => {
18     // 延迟函数执行完毕，清空缓存的定时器序号
19     timer = null
20     // 延迟执行的情况下，函数会在延迟函数中执行
21     // 使用到之前缓存的参数和上下文
22     if (!immediate) {
23       func.apply(context, args)
24       context = args = null
25     }
26   }, wait)
27
28   // 这里返回的函数是每次实际调用的函数
29   return function(...params) {
30     // 如果没有创建延迟执行函数（later），就创建一个
31     if (!timer) {
```



```

32     timer = later()
33     // 如果是立即执行，调用函数
34     // 否则缓存参数和调用上下文
35     if (immediate) {
36         func.apply(this, params)
37     } else {
38         context = this
39         args = params
40     }
41     // 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
42     // 这样做延迟函数会重新计时
43 } else {
44     clearTimeout(timer)
45     timer = later()
46 }
47 }
48 }

```

- 对于按钮防点击来说的实现：如果函数是立即执行的，就立即调用，如果函数是延迟执行的，就缓存上下文和参数，放到延迟函数中去执行。一旦我开始一个定时器，只要我定时器还在，你每次点击我都重新计时。一旦你点累了，定时器时间到，定时器重置为 `null`，就可以再次点击了。
- 对于延时执行函数来说的实现：清除定时器 ID，如果是延迟调用就调用函数

13 节流

防抖动和节流本质是不一样的。防抖动是将多次执行变为最后一次执行，节流是将多次执行变成每隔一段时间执行

```

1  /**
2   * underscore 节流函数，返回函数连续调用时，func 执行频率限定为 次 /
   wait
3   *
4   * @param {function} func      回调函数
5   * @param {number} wait        表示时间窗口的间隔
6   * @param {object} options     如果想忽略开始函数的调用，传入
   {leading: false}。
7   *                             如果想忽略结尾函数的调用，传入
   {trailing: false}
8   *                             两者不能共存，否则函数不能执行
9   * @return {function}         返回客户调用函数
10  */
11  _throttle = function(func, wait, options) {
12      var context, args, result;

```

```
13     var timeout = null;
14     // 之前的时间戳
15     var previous = 0;
16     // 如果 options 没传则设为空对象
17     if (!options) options = {};
18     // 定时器回调函数
19     var later = function() {
20         // 如果设置了 leading, 就将 previous 设为 0
21         // 用于下面函数的第一个 if 判断
22         previous = options.leading === false ? 0 : _.now();
23         // 置空一是为了防止内存泄漏, 二是为了下面的定时器判断
24         timeout = null;
25         result = func.apply(context, args);
26         if (!timeout) context = args = null;
27     };
28     return function() {
29         // 获得当前时间戳
30         var now = _.now();
31         // 首次进入前者肯定为 true
32         // 如果需要第一次不执行函数
33         // 就将上次时间戳设为当前的
34         // 这样在接下来计算 remaining 的值时会大于0
35         if (!previous && options.leading === false) previous =
now;
36         // 计算剩余时间
37         var remaining = wait - (now - previous);
38         context = this;
39         args = arguments;
40         // 如果当前调用已经大于上次调用时间 + wait
41         // 或者用户手动调了时间
42         // 如果设置了 trailing, 只会进入这个条件
43         // 如果没有设置 leading, 那么第一次会进入这个条件
44         // 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
45         // 其实还是会进入的, 因为定时器的延时
46         // 并不是准确的时间, 很可能你设置了2秒
47         // 但是他需要2.2秒才触发, 这时候就会进入这个条件
48         if (remaining <= 0 || remaining > wait) {
49             // 如果存在定时器就清理掉否则会调用二次回调
50             if (timeout) {
51                 clearTimeout(timeout);
52                 timeout = null;
53             }
54             previous = now;
55             result = func.apply(context, args);
56             if (!timeout) context = args = null;
```

```

57     } else if (!timeout && options.trailing !== false) {
58         // 判断是否设置了定时器和 trailing
59         // 没有的话就开启一个定时器
60         // 并且不能同时设置 leading 和 trailing
61         timeout = setTimeout(later, remaining);
62     }
63     return result;
64 };
65 };

```

14 继承

在 ES5 中，我们可以使用如下方式解决继承的问题

```

1  function Super() {}
2  Super.prototype.getNumber = function() {
3      return 1
4  }
5
6  function Sub() {}
7  let s = new Sub()
8  Sub.prototype = Object.create(Super.prototype, {
9      constructor: {
10         value: Sub,
11         enumerable: false,
12         writable: true,
13         configurable: true
14     }
15 })

```

- 以上继承实现思路就是将子类的原型设置为父类的原型
- 在 ES6 中，我们可以通过 `class` 语法轻松解决这个问题

```

1  class MyDate extends Date {
2      test() {
3          return this.getTime()
4      }
5  }
6  let myDate = new MyDate()
7  myDate.test()

```

- 但是 ES6 不是所有浏览器都兼容，所以我们需要使用 `Babel` 来编译这段代码。
- 如果你使用编译过得代码调用 `myDate.test()` 你会惊奇地发现出现了报错

因为在 JS 底层有限制，如果不是由 `Date` 构造出来的实例的话，是不能调用 `Date` 里的函数的。所以这也侧面的说明了：ES6 中的 `class` 继承与 ES5 中的一般继承写法是不同的。

- 既然底层限制了实例必须由 `Date` 构造出来，那么我们可以改变下思路实现继承

```
1 function MyData() {  
2  
3 }  
4 MyData.prototype.test = function () {  
5   return this.getTime()  
6 }  
7 let d = new Date()  
8 Object.setPrototypeOf(d, MyData.prototype)  
9 Object.setPrototypeOf(MyData.prototype, Date.prototype)
```

- 以上继承实现思路：先创建父类实例 => 改变实例原先的 `__proto__` 转而连接到子类的 `prototype` => 子类的 `prototype` 的 `__proto__` 改为父类的 `prototype`。
- 通过以上方法实现的继承就可以完美解决 JS 底层的这个限制

15 call, apply, bind

- `call` 和 `apply` 都是为了解决改变 `this` 的指向。作用都是相同的，只是传参的方式不同。
- 除了第一个参数外，`call` 可以接收一个参数列表，`apply` 只接受一个参数数组

```
1 let a = {  
2   value: 1  
3 }  
4 function getValue(name, age) {  
5   console.log(name)  
6   console.log(age)  
7   console.log(this.value)  
8 }  
9 getValue.call(a, 'yck', '24')  
10 getValue.apply(a, ['yck', '24'])
```

16 Promise 实现

- 可以把 `Promise` 看成一个状态机。初始是 `pending` 状态，可以通过函数 `resolve` 和 `reject`，将状态转变为 `resolved` 或者 `rejected` 状态，状态一旦改变就不能再次变化。

- `then` 函数会返回一个 `Promise` 实例，并且该返回值是一个新的实例而不是之前的实例。因为 `Promise` 规范规定除了 `pending` 状态，其他状态是不可以改变的，如果返回的是一个相同实例的话，多个 `then` 调用就失去意义了。
- 对于 `then` 来说，本质上可以把它看成是 `flatMap`

```
1 // 三种状态
2 const PENDING = "pending";
3 const RESOLVED = "resolved";
4 const REJECTED = "rejected";
5 // promise 接收一个函数参数，该函数会立即执行
6 function MyPromise(fn) {
7   let _this = this;
8   _this.currentState = PENDING;
9   _this.value = undefined;
10  // 用于保存 then 中的回调，只有当 promise
11  // 状态为 pending 时才会缓存，并且每个实例至多缓存一个
12  _this.resolvedCallbacks = [];
13  _this.rejectedCallbacks = [];
14
15  _this.resolve = function (value) {
16    if (value instanceof MyPromise) {
17      // 如果 value 是个 Promise，递归执行
18      return value.then(_this.resolve, _this.reject)
19    }
20    setTimeout(() => { // 异步执行，保证执行顺序
21      if (_this.currentState === PENDING) {
22        _this.currentState = RESOLVED;
23        _this.value = value;
24        _this.resolvedCallbacks.forEach(cb => cb());
25      }
26    })
27  };
28
29  _this.reject = function (reason) {
30    setTimeout(() => { // 异步执行，保证执行顺序
31      if (_this.currentState === PENDING) {
32        _this.currentState = REJECTED;
33        _this.value = reason;
34        _this.rejectedCallbacks.forEach(cb => cb());
35      }
36    })
37  }
38  // 用于解决以下问题
39  // new Promise(() => throw Error('error'))
```

```

40     try {
41         fn(_this.resolve, _this.reject);
42     } catch (e) {
43         _this.reject(e);
44     }
45 }
46
47 MyPromise.prototype.then = function (onResolved, onRejected)
48 {
49     var self = this;
50     // 规范 2.2.7, then 必须返回一个新的 promise
51     var promise2;
52     // 规范 2.2.onResolved 和 onRejected 都为可选参数
53     // 如果类型不是函数需要忽略, 同时也实现了透传
54     // Promise.resolve(4).then().then((value) =>
55     console.log(value))
56     onResolved = typeof onResolved === 'function' ? onResolved
57     : v => v;
58     onRejected = typeof onRejected === 'function' ? onRejected
59     : r => throw r;
60
61     if (self.currentState === RESOLVED) {
62         return (promise2 = new MyPromise(function (resolve,
63         reject) {
64             // 规范 2.2.4, 保证 onFulfilled, onRejected 异步执行
65             // 所以用了 setTimeout 包裹下
66             setTimeout(function () {
67                 try {
68                     var x = onResolved(self.value);
69                     resolutionProcedure(promise2, x, resolve, reject);
70                 } catch (reason) {
71                     reject(reason);
72                 }
73             });
74         }));
75     }
76
77     if (self.currentState === REJECTED) {
78         return (promise2 = new MyPromise(function (resolve,
79         reject) {
80             setTimeout(function () {
81                 // 异步执行onRejected
82                 try {
83                     var x = onRejected(self.value);
84                     resolutionProcedure(promise2, x, resolve, reject);

```

```

79         } catch (reason) {
80             reject(reason);
81         }
82     });
83 });
84 }
85
86 if (self.currentState === PENDING) {
87     return (promise2 = new MyPromise(function (resolve,
reject) {
88         self.resolvedCallbacks.push(function () {
89             // 考虑到可能会有报错，所以使用 try/catch 包裹
90             try {
91                 var x = onResolved(self.value);
92                 resolutionProcedure(promise2, x, resolve, reject);
93             } catch (r) {
94                 reject(r);
95             }
96         });
97
98         self.rejectedCallbacks.push(function () {
99             try {
100                 var x = onRejected(self.value);
101                 resolutionProcedure(promise2, x, resolve, reject);
102             } catch (r) {
103                 reject(r);
104             }
105         });
106     }));
107 }
108 };
109 // 规范 2.3
110 function resolutionProcedure(promise2, x, resolve, reject) {
111     // 规范 2.3.1, x 不能和 promise2 相同，避免循环引用
112     if (promise2 === x) {
113         return reject(new TypeError("Error"));
114     }
115     // 规范 2.3.2
116     // 如果 x 为 Promise，状态为 pending 需要继续等待否则执行
117     if (x instanceof MyPromise) {
118         if (x.currentState === PENDING) {
119             x.then(function (value) {
120                 // 再次调用该函数是为了确认 x resolve 的
121                 // 参数是什么类型，如果是基本类型就再次 resolve
122                 // 把值传给下个 then

```

```
123     resolutionProcedure(promise2, value, resolve,  
124     reject);  
125     }, reject);  
126     } else {  
127         x.then(resolve, reject);  
128     }  
129     return;  
130 }  
131 // 规范 2.3.3.3.3  
132 // reject 或者 resolve 其中一个执行过得话, 忽略其他的  
133 let called = false;  
134 // 规范 2.3.3, 判断 x 是否为对象或者函数  
135 if (x !== null && (typeof x === "object" || typeof x ===  
136 "function")) {  
137     // 规范 2.3.3.2, 如果不能取出 then, 就 reject  
138     try {  
139         // 规范 2.3.3.1  
140         let then = x.then;  
141         // 如果 then 是函数, 调用 x.then  
142         if (typeof then === "function") {  
143             // 规范 2.3.3.3  
144             then.call(  
145                 x,  
146                 y => {  
147                     if (called) return;  
148                     called = true;  
149                     // 规范 2.3.3.3.1  
150                     resolutionProcedure(promise2, y, resolve,  
151                     reject);  
152                 },  
153                 e => {  
154                     if (called) return;  
155                     called = true;  
156                     reject(e);  
157                 }  
158             );  
159         } else {  
160             // 规范 2.3.3.4  
161             resolve(x);  
162         }  
163     } catch (e) {  
164         if (called) return;  
165         called = true;  
166         reject(e);  
167     }  
168 }
```



```
165     } else {
166         // 规范 2.3.4, x 为基本类型
167         resolve(x);
168     }
169 }
```

17 Generator 实现

Generator 是 ES6 中新增的语法, 和 Promise 一样, 都可以用来异步编程

```
1 // 使用 * 表示这是一个 Generator 函数
2 // 内部可以通过 yield 暂停代码
3 // 通过调用 next 恢复执行
4 function* test() {
5     let a = 1 + 2;
6     yield 2;
7     yield 3;
8 }
9 let b = test();
10 console.log(b.next()); // > { value: 2, done: false }
11 console.log(b.next()); // > { value: 3, done: false }
12 console.log(b.next()); // > { value: undefined, done: true }
```

从以上代码可以发现，加上 `*` 的函数执行后拥有了 `next` 函数，也就是说函数执行后返回了一个对象。每次调用 `next` 函数可以继续执行被暂停的代码。以下是 `Generator` 函数的简单实现

```
1 // cb 也就是编译过的 test 函数
2 function generator(cb) {
3     return (function() {
4         var object = {
5             next: 0,
6             stop: function() {}
7         };
8
9         return {
10             next: function() {
11                 var ret = cb(object);
12                 if (ret === undefined) return { value: undefined,
done: true };
13                 return {
14                     value: ret,
15                     done: false
16                 };
17             }
18         };
19     })();
20 }
```

```

17     }
18   };
19   })();
20 }
21 // 如果你使用 babel 编译后可以发现 test 函数变成了这样
22 function test() {
23   var a;
24   return generator(function(_context) {
25     while (1) {
26       switch ((_context.prev = _context.next)) {
27         // 可以发现通过 yield 将代码分割成几块
28         // 每次执行 next 函数就执行一块代码
29         // 并且表明下次需要执行哪块代码
30         case 0:
31           a = 1 + 2;
32           _context.next = 4;
33           return 2;
34         case 4:
35           _context.next = 6;
36           return 3;
37         // 执行完毕
38         case 6:
39         case "end":
40           return _context.stop();
41       }
42     }
43   });
44 }

```

18 Proxy

Proxy 是 ES6 中新增的功能，可以用来自定义对象中的操作

```

1 let p = new Proxy(target, handler);
2 // `target` 代表需要添加代理的对象
3 // `handler` 用来自定义对象中的操作
4 可以很方便的使用 Proxy 来实现一个数据绑定和监听
5
6 let onwatch = (obj, setBind, getLogger) => {
7   let handler = {
8     get(target, property, receiver) {
9       getLogger(target, property)
10      return Reflect.get(target, property, receiver);
11    },
12    set(target, property, value, receiver) {

```

```
13     setBind(value);
14     return Reflect.set(target, property, value);
15   }
16 };
17 return new Proxy(obj, handler);
18 };
19
20 let obj = { a: 1 }
21 let value
22 let p = onwatch(obj, (v) => {
23   value = v
24 }, (target, property) => {
25   console.log(`Get '${property}' = ${target[property]}`);
26 })
27 p.a = 2 // bind `value` to `2`
28 p.a // -> Get 'a' = 2
```