> 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。

```js
specialObject = {};

Scope = specialObject + Scope;

foo = new FunctionExpression;
foo. [ [Scope]] = Scope;
specialObject.foo = foo; // {DontDelete}, {ReadOnly}

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。

```js
function A() {
  let a = 1
  function B() {
      console.log(a)
  }
  return B
}
```

> 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

```js
for ( var i=1; i<=5; i++) {
    setTimeout( function timer() {
        console.log( i );
```

```
    }, i*1000 );
}
```

- 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

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

- 第二种就是使用 `setTimeout` 的第三个参数

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

> 第三种就是使用 `let` 定义 `i` 了

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

> 因为对于 `let` 来说，他会创建一个块级作用域，相当于

```js
{ // 形成块级作用域
  let i = 0
  {
    let ii = i
```

```
    setTimeout( function timer() {
        console.log( i );
    }, i*1000 );
  }
  i++
  {
    let ii = i
  }
  i++
  {
    let ii = i
  }
  ...
}
```

10 深浅拷贝

```js
letet a a = {
    age     : 1
}
let b = a
a.age = 2
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

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

> 当然我们也可以通过展开运算符 `(...)` 来解决

```js
let a = {
    age: 1
}
let b = {...a}
a.age = 2
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

```js
let a = {
    age: 1,
    jobs: {
        first: 'FE'
    }
}
let b = {...a}
a.jobs.first = 'native'
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

```js
let a = {
    age: 1,
    jobs: {
        first: 'FE'
    }
}
let b = JSON.parse(JSON.stringify(a))
```

```
a.jobs.first = 'native'
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

```js
let obj = {
  a: 1,
  b: {
    c: 2,
    d: 3,
  },
}
obj.c = obj.b
obj.e = obj.a
obj.b.c = obj.c
obj.b.d = obj.b
obj.b.e = obj.b.c
let newObj = JSON.parse(JSON.stringify(obj))
console.log(newObj)
```

> 如果你有这么一个循环引用对象，你会发现你不能通过该方法深拷贝

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

```js
let a = {
    age: undefined,
    jobs: function() {},
    name: 'poetries'
}
let b = JSON.parse(JSON.stringify(a))
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

```js
// file a.js
export function a() {}
export function b() {}
// file b.js
export default function() {}

import {a, b} from './a.js'
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。

```js
// a.js
module.exports = {
    a: 1
}
// or
exports.a = 1

// b.js
var module = require( './a.js')
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

```js
var module = require( './a.js')
module.a
// 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，
```

```
// 重要的是 module 这里，module 是 Node 独有的一个变量
module.exports = {
    a: 1
}
// 基本实现
var module = {
  exports: {} // exports 就是个空对象
}
// 这个是为什么 exports 和 module.exports 用法相似的原因
var exports = module.exports
var load = function (module) {
    // 导出的东西
    var a = 1
    module.exports = a
    return module.exports
};
```

> 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

```
// AMD
define( [ './a', './b'], function(a, b) {
```

```
    a.do()
    b.do()
})
define(function(require, exports, module) {
    var a = require( './a')
    a.doSomething()
    var b = require( './b')
    b.doSomething()
})
```

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

```js
// 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()
    }
  }
}
```

- 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

```
/**
 * 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: fals
 *                                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;
    };
  };
```

14 继承

> 在 ES5 中， 我们可以使用如下方式解决继承的问题

```
function Super() {}
Super.prototype.getNumber = function() {
  return 1
}

function Sub() {}
let s = new Sub()
Sub.prototype = Object.create(Super.prototype, {
  constructor: {
    value: Sub,
    enumerable: false,
    writable: true,
    configurable: true
  }
})
```

- 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

```js
class MyDate extends Date {
  test() {
    return this.getTime()
  }
}
let myDate = new MyDate()
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

```js
function MyData() {

}
MyData.prototype.test = function () {
  return this.getTime()
}
let d = new Date()
Object.setPrototypeOf(d, MyData.prototype)
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

```js
let a = {
    value: 1
```

```
}
function getValue(name, age) {
    console.log(name)
    console.log(age)
    console.log(this.value)
}
getValue.call(a, 'yck', '24')
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

```
// 三种状态
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";
// promise 接收一个函数参数，该函数会立即执行
function MyPromise(fn) {
  let _this = this;
  _this.currentState = PENDING;
  _this.value = undefined;
  // 用于保存 then 中的回调， 只有当 promise
  // 状态为 pending 时才会缓存，并且每个实例至多缓存一个
  _this.resolvedCallbacks = [];
  _this.rejectedCallbacks = [];

  _this.resolve = function (value) {
    if (value instanceof MyPromise) {
      // 如果 value 是个 Promise，递归执行
      return value.then(_this.resolve, _this.reject)
    }
    setTimeout(() => { // 异步执行，保证执行顺序
      if (_this.currentState === PENDING) {
        _this.currentState = RESOLVED;
        _this.value = value;
        _this.resolvedCallbacks.forEach(cb => cb());
```

```
      }
    })
  };

  _this.reject = function (reason) {
    setTimeout(() => { // 异步执行，保证执行顺序
      if (_this.currentState === PENDING) {
        _this.currentState = REJECTED;
        _this.value = reason;
        _this.rejectedCallbacks.forEach(cb => cb());
      }
    })
  }
  // 用于解决以下问题
  // new Promise(() => throw Error('error'))
  try {
    fn(_this.resolve, _this.reject);
  } catch (e) {
    _this.reject(e);
  }
}

MyPromise.prototype.then = function (onResolved, onRejected) {
  var self = this;
  // 规范 2.2.7，then 必须返回一个新的 promise
  var promise2;
  // 规范 2.2.onResolved 和 onRejected 都为可选参数
  // 如果类型不是函数需要忽略，同时也实现了透传
  // Promise.resolve(4).then().then((value) => console.log(value))
  onResolved = typeof onResolved === 'function' ? onResolved : v => v;
  onRejected = typeof onRejected === 'function' ? onRejected : r => throw r

  if (self.currentState === RESOLVED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      // 规范 2.2.4，保证 onFulfilled，onRjected 异步执行
      // 所以用了 setTimeout 包裹下
      setTimeout(function () {
        try {
          var x = onResolved(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (reason) {
          reject(reason);
        }
      });
    }));
  }
```

```
  if (self.currentState === REJECTED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      setTimeout(function () {
        // 异步执行onRejected
        try {
          var x = onRejected(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (reason) {
          reject(reason);
        }
      });
    }));
  }

  if (self.currentState === PENDING) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      self.resolvedCallbacks.push(function () {
        // 考虑到可能会有报错，所以使用 try/catch 包裹
        try {
          var x = onResolved(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (r) {
          reject(r);
        }
      });

      self.rejectedCallbacks.push(function () {
        try {
          var x = onRejected(self.value);
          resolutionProcedure(promise2, x, resolve, reject);
        } catch (r) {
          reject(r);
        }
      });
    }));
  }
};
// 规范 2.3
function resolutionProcedure(promise2, x, resolve, reject) {
  // 规范 2.3.1，x 不能和 promise2 相同，避免循环引用
  if (promise2 === x) {
    return reject(new TypeError("Error"));
  }
  // 规范 2.3.2
  // 如果 x 为 Promise，状态为 pending 需要继续等待否则执行
  if (x instanceof MyPromise) {
    if (x.currentState === PENDING) {
```

```
      x.then(function (value) {
        // 再次调用该函数是为了确认 x resolve 的
        // 参数是什么类型，如果是基本类型就再次 resolve
        // 把值传给下个 then
        resolutionProcedure(promise2, value, resolve, reject);
      }, reject);
    } else {
      x.then(resolve, reject);
    }
    return;
  }
  // 规范 2.3.3.3
  // reject 或者 resolve 其中一个执行过得话， 忽略其他的
  let called = false;
  // 规范 2.3.3，判断 x 是否为对象或者函数
  if (x !== null && (typeof x === "object" || typeof x === "function")) {
    // 规范 2.3.3.2，如果不能取出 then，就 reject
    try {
      // 规范 2.3.3.1
      let then = x.then;
      // 如果 then 是函数，调用 x.then
      if (typeof then === "function") {
        // 规范 2.3.3.3
        then.call(
          x,
          y => {
            if (called) return;
            called = true;
            // 规范 2.3.3.3.1
            resolutionProcedure(promise2, y, resolve, reject);
          },
          e => {
            if (called) return;
            called = true;
            reject(e);
          }
        );
      } else {
        // 规范 2.3.3.4
        resolve(x);
      }
    } catch (e) {
      if (called) return;
      called = true;
      reject(e);
    }
  } else {
```

```
    // 规范 2.3.4, x 为基本类型
    resolve(x);
  }
}
```

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

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

```js
// cb 也就是编译过的 test 函数
function generator(cb) {
  return (function() {
    var object = {
      next: 0,
      stop: function() {}
    };

    return {
      next: function() {
        var ret = cb(object);
        if (ret === undefined) return { value: undefined, done: true };
```

```
          return {
            value: ret,
            done: false
          };
        }
      };
    })();
}
// 如果你使用 babel 编译后可以发现 test 函数变成了这样
function test() {
  var a;
  return generator(function(_context) {
    while (1) {
      switch ((_context.prev = _context.next)) {
        // 可以发现通过 yield 将代码分割成几块
        // 每次执行 next 函数就执行一块代码
        // 并且表明下次需要执行哪块代码
        case 0:
          a = 1 + 2;
          _context.next = 4;
          return 2;
        case 4:
          _context.next = 6;
          return 3;
        // 执行完毕
        case 6:
        case "end":
          return _context.stop();
      }
    }
  });
}
```

18 Proxy

> Proxy 是 ES6 中新增的功能， 可以用来自定义对象中的操作

```
let p = new Proxy(target, handler);
// `target` 代表需要添加代理的对象
// `handler` 用来自定义对象中的操作
```

可以很方便的使用 Proxy 来实现一个数据绑定和监听

```
let onWatch = (obj, setBind, getLogger) => {
```

```
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver);
    },
    set(target, property, value, receiver) {
      setBind(value);
      return Reflect.set(target, property, value);
    }
  };
  return new Proxy(obj, handler);
};

let obj = { a: 1 }
let value
let p = onWatch(obj, (v) => {
  value = v
}, (target, property) => {
  console.log(`Get '${property} ' = ${target[property]}`);
})
p.a = 2 // bind `value` to `2`
p.a // -> Get 'a' = 2
```

二、浏览器

1 事件机制

事件触发三阶段

- document 往事件触发处传播， 遇到注册的捕获事件会触发
- 传播到事件触发处时触发注册的事件
- 从事件触发处往 document 传播， 遇到注册的冒泡事件会触发

> 事件触发一般来说会按照上面的顺序进行，但是也有特例， 如果给一个目标节点同时注册冒泡和捕获事件，事件触发会按照注册的顺序执行

```
// 以下会先打印冒泡然后是捕获
node.addEventListener( 'click',(event) =>{
    console.log( '冒泡 ')
},false);
node.addEventListener( 'click',(event) =>{
```

```
    console.log( '捕获 ')
},true)
```

注册事件

- 通常我们使用 addEventListener 注册事件，该函数的第三个参数可以是布尔值，也可以是对象 。对于布尔值 useCapture 参数来说，该参数默认值为 false 。 useCapture 决定了注册的事件是捕获事件还是冒泡事件
- 一般来说， 我们只希望事件只触发在目标上， 这时候可以使用 stopPropagation 来阻止事件的进一步传播 。通常我们认为 stopPropagation 是用来阻止事件冒泡的， 其实该函数也可以阻止捕获事件 。 stopImmediatePropagation 同样也能实现阻止事件，但是还能阻止该事件目标执行别的注册事件

```js
node.addEventListener( 'click',(event) =>{
    event.stopImmediatePropagation()
    console.log( '冒泡 ')
},false);
// 点击 node 只会执行上面的函数，该函数不会执行
node.addEventListener( 'click',(event) => {
    console.log( '捕获 ')
},true)
```

事件代理

> 如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上

```html
<ul id="ul">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
<script>
    let ul = document.querySelector( '##ul')
    ul.addEventListener( 'click', (event) => {
        console.log(event.target);
    })
</script>
```