

# 目录

Introduction	1.1
--------------	-----

## 【 JS内功修炼 】

### 第一节: this/闭包/作用域

专业术语	3.1
执行上下文	3.2
作用域	3.3
闭包	3.4
this	3.5
相关面试题	3.6

### 第二节: 面向对象/原型链/继承

专业术语	4.1
面向对象	4.2
原型链	4.3
继承	4.4
相关面试题	4.5

### 第三节: es6深入理解

专业术语	5.1
babel	5.2
块级作用域	5.3
函数	5.4
扩展对象的功能性	5.5
解构	5.6
Set与Map	5.7
js的类	5.8
改进的数组功能	5.9
Promise与异步编程	5.10
代理和反射	5.11
用模块封装代码	5.12

---

async-await	5.13
Decorator装饰器	5.14

---

## 第四节: 模块化/浏览器事件模型

[专业术语]	6.1
[模块化]	6.2
[浏览器事件模型]	6.3

---

## 第五节: VUE基础

数据绑定	7.1
指令	7.2
计算属性	7.3
过滤器	7.4
Class与Style绑定	7.5
过渡	7.6
事件处理	7.7
生命周期	7.8
组件	7.9

# 爪哇教育-前端课程

- 讲师：袁鑫
- 2019-8

# JS内功修炼

## 专业术语

- 常量、变量、数据类型
- 形参、实参
- 匿名函数、具名函数、自执行函数
- 函数声明、函数表达式
- 堆、栈
- 同步、异步、进程、线程

## 执行上下文

当函数执行时，会创建一个称为执行上下文（execution context）的环境，分为创建和执行2个阶段

### 创建阶段

创建阶段，指函数被调用但还未执行任何代码时，此时创建了一个拥有3个属性的对象：

```
executionContext = {
  scopeChain: {}, // 创建作用域链 (scope chain)
  variableObject: {}, // 初始化变量、函数、形参
  this: {} // 指定this
}
```

### 代码执行阶段

代码执行阶段主要的工作是：1、分配变量、函数的引用，赋值。2、执行代码。

举个栗子

```
// 一段这样的代码
function demo(num) {
  var name = 'xiaowa';
  var getData = function getData() {};
  function c() {}
}
demo(100);

// 创建阶段大致这样，在这个阶段就出现了【变量提升(Hoisting)】
executionContext = {
  scopeChain: { ... },
  variableObject: {
```

```

arguments: { // 创建了参数对象
  0: 100,
  length: 1
},
num: 100, // 创建形参名称，赋值/或创建引用拷贝
c: pointer to function c() // 有内部函数声明的话，创建引用指向函数体
name: undefined, // 有内部声明变量a，初始化为undefined
getData: undefined // 有内部声明变量b，初始化为undefined
},
this: { ... }
}

// 代码执行阶段，在这个阶段主要是赋值并执行代码
executionContext = {
  scopeChain: { ... },
  variableObject: {
    arguments: {
      0: 100,
      length: 1
    },
    num: 100,
    c: pointer to function c()
    name: 'xiaowa', // 分配变量，赋值
    getData: pointer to function getData() // 分配函数的引用，赋值
  },
  this: { ... }
}

```

## 执行上下文栈

- 浏览器中的JS解释器是单线程的，相当于浏览器中同一时间只能做一个事情。
- 代码中只有一个全局执行上下文，和无数个函数执行上下文，这些组成了执行上下文栈（Execution Stack）。
- 一个函数的执行上下文，在函数执行完毕后，会被移出执行上下文栈。

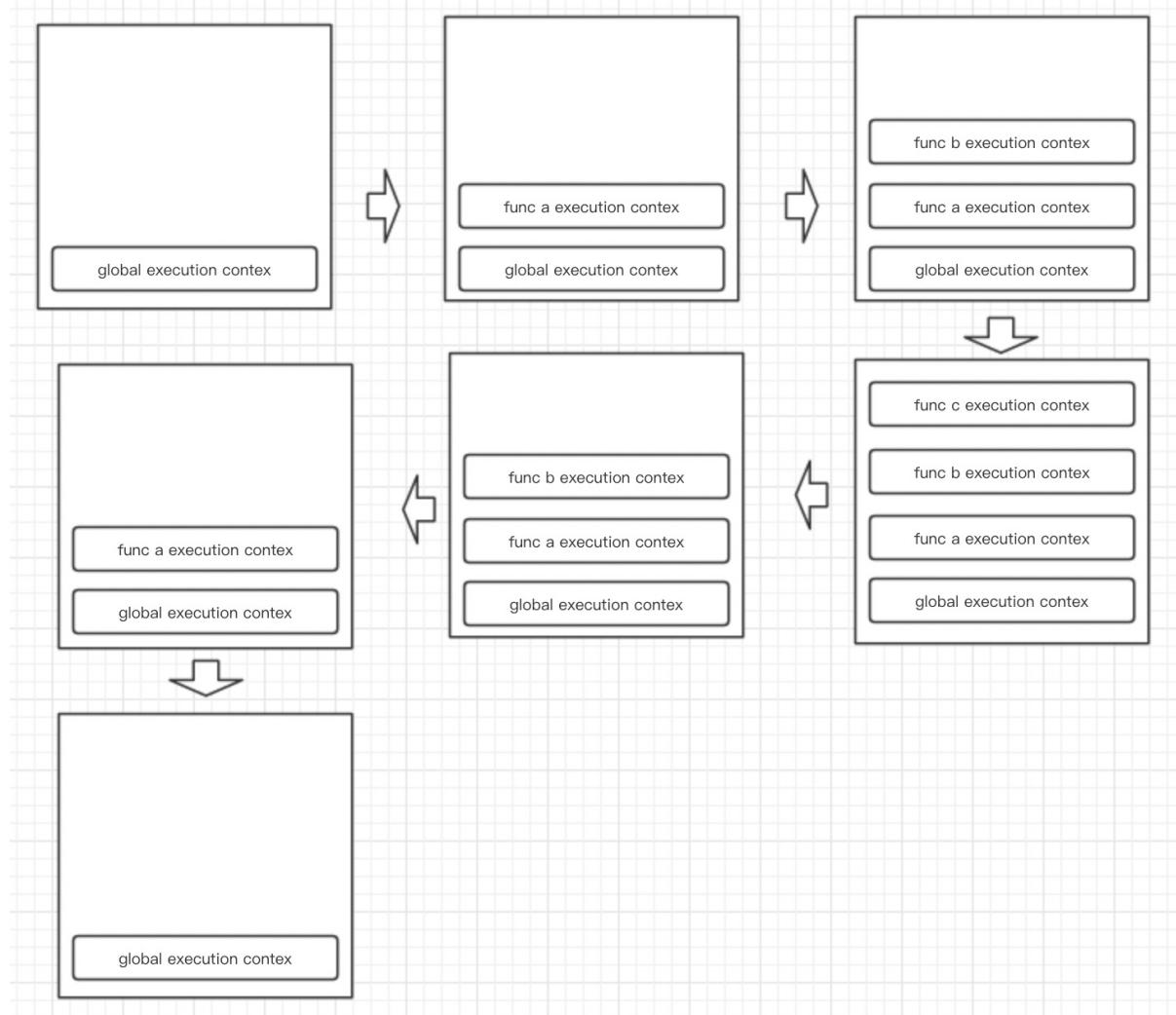
举个栗子

```

function c(){
  console.log('ok');
}
function b(){
  c();
}
function a(){
  b();
}
a();

```

这个栗子的执行上下文栈是这样的



## 作用域

js中有全局作用域、函数作用域，es6中又增加了块级作用域。作用域的最大用途就是隔离变量或函数，并控制他们的生命周期。作用域是在函数执行上下文创建时定义好的，不是函数执行时定义的。

举个栗子

```
// 不要看晕了哦~
function a () {
    return function b() {
        var myname = 'b';
        console.log(myname); // b
    }
}
function c() {
    var myname = 'c';
    b();
}
```

```

var b = a();
c();

// 去掉函数b中的myname声明后
function a () {
    return function b() {
        // var myname = 'b';
        console.log(myname); // 这里会报错
    }
}
function c() {
    var myname = 'c';
    b();
}
var b = a();
c();

```

## 作用域链

当一个块或函数嵌套在另一个块或函数中时，就发生了作用域的嵌套。在当前函数中如果js引擎无法找到某个变量，就会往上一级嵌套的作用域中去寻找，直到找到该变量或抵达全局作用域，这样的链式关系就称为作用域链(Scope Chain)

## 闭包

高级程序设计三中:闭包是指有权访问另外一个函数作用域中的变量的函数.可以理解为(能够读取其他函数内部变量的函数)

wiki百科的解释: [https://en.wikipedia.org/wiki/Closure\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Closure_(computer_programming))

In programming languages, a closure, also lexical closure or function closure, is a technique for implementing lexically scoped name binding in a language with first-class functions. Operationally, a closure is a record storing a function[a] together with an environment.[1] The environment is a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.[b] Unlike a plain function, a closure allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

```

function outer() {
    var top = xxxx;
    function inner() {
        xxx.innerHTML = top;
    }
}

```

平时用在哪儿?

## 1、封装私有变量(amd的框架等都使用)

```
// 普通的定义类的方式
function Person() {
    this._attackVolume = 100;
}

Person.prototype = {
    attack(body) {
        body.bloodVolume -= this.attackVolume - body.defenseVolume;
    }
};

var person = new Person();
console.log(person._attackVolume);

// 工厂方法
function Person() {
    var _attackVolume = 100;
    return {
        attack() {
            body.bloodVolume -= attackVolume - body.defenseVolume;
        }
    };
}

var person = new Person();
console.log(person._attackVolume);
```

## 2、存储变量

```
// 封装的时候
function getListDataManager() {
    // 外层scope中定义一个变量
    let localData = null;

    return {
        getData() {
            // 里面的函数使用外层的变量，而且是反复使用
            if (localData) {
                return Promise.resolve(localData);
            }
            return fetch('xxxx')
                .then(data => localData = data.json());
        }
    };
}

// 用的时候
```

```
const listDataManager = getListDataManager();

button.onclick = () => {
    // 每次都会去获取数据，但是有可能是获取的缓存的数据
    text.innerHTML = listDataManager.getData();
};

window.onscroll = () => {
    // 每次都会去获取数据，但是有可能是获取的缓存的数据
    text.innerHTML = listDataManager.getData();
};
```

## this

一共有5种场景。

### 场景1：函数直接调用时

```
function myfunc() {
    console.log(this) // this是window
}
var a = 1;
myfunc();
```

### 场景2：函数被别人调用时

```
function myfunc() {
    console.log(this) // this是对象a
}
var a = {
    myfunc: myfunc
};
a.myfunc();
```

### 场景3：new一个实例时

```
function Person(name) {
    this.name = name;
    console.log(this); // this是指实例p
}
var p = new Person('zhaowa');
```

### 场景4：apply、call、bind时

```
function getColor(color) {
    this.color = color;
    console.log(this);
}

function Car(name, color){
    this.name = name;    // this指的是实例car
    getColor.call(this, color); // 这里的this从原本的getColor, 变成了car
}

var car = new Car('卡车', '绿色');
```

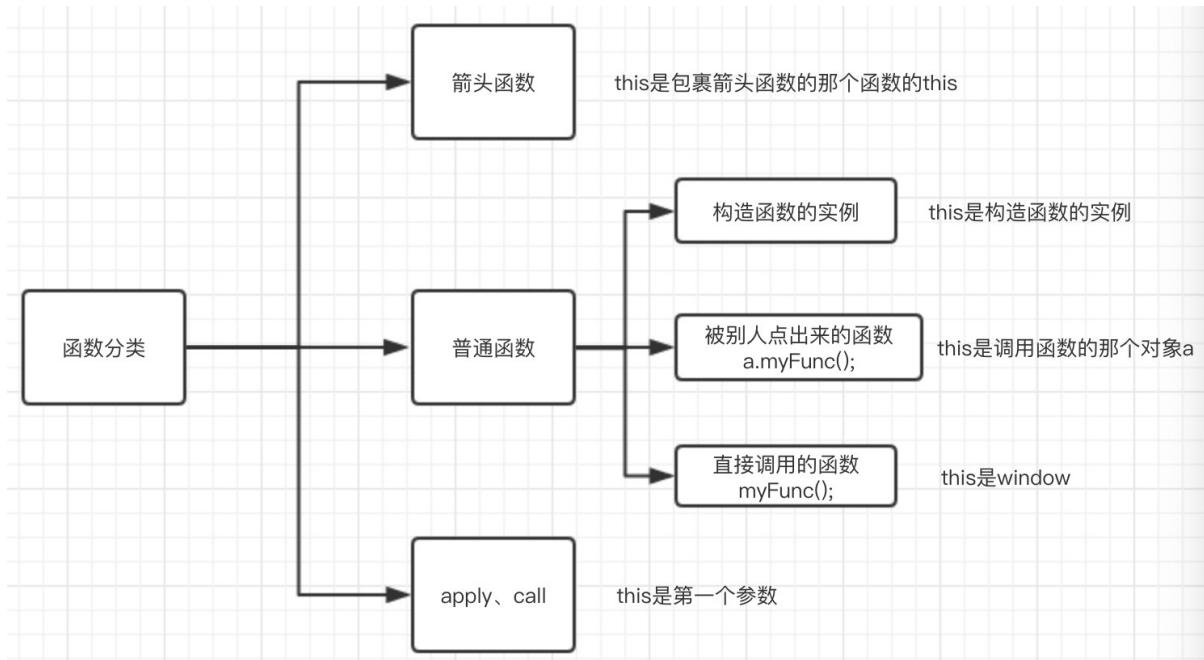
## 场景5: 箭头函数时

```
// 复习一下场景1
var a = {
    myfunc: function() {
        setTimeout(function(){
            console.log(this); // this是a
        }, 0)
    }
};
a.myfunc();

// 稍微改变一下
var a = {
    myfunc: function() {
        var that = this;
        setTimeout(function(){
            console.log(that); // this是a
        }, 0)
    }
};
a.myfunc();

// 箭头函数
var a = {
    myfunc: function() {
        setTimeout(() => {
            console.log(this); // this是a
        }, 0)
    }
};
a.myfunc();
```

总结一下



- 对于直接调用的函数来说，不管函数被放在了什么地方，this都是window
- 对于被别人调用的函数来说，被谁点出来的，this就是谁
- 在构造函数中，类中(函数体中)出现的this.xxx=xxx中的this是当前类的一个实例
- call、apply时，this是第一个参数。bind要优于call/apply哦，call参数多，apply参数少
- 箭头函数没有自己的this，需要看其外层的是否有函数，如果有，外层函数的this就是内部箭头函数的this，如果没有，则this是window

## 相关面试题

### 1. 考察this三板斧

#### 1.1

```

function show () {
  console.log('this:', this);
}
var obj = {
  show: show
};
obj.show();
  
```

```

function show () {
  console.log('this:', this);
}
  
```

```

var obj = {
  
```

```
show: function () {
    show();
}
};

obj.show();
```

## 1.2

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};

(0, obj.show)();
```

## 1.3

```
var obj = {
    sub: {
        show: function () {
            console.log('this:', this);
        }
    }
};
```

## 1.4

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};

var newobj = new obj.show();
```

## 1.5

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};

var newobj = new (obj.show.bind(obj))();
```

## 1.6

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
var newobj = new (obj.show.bind(obj))();
```

## 1.7

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
var elem = document.getElementById('book-search-results');
elem.addEventListener('click', obj.show);
elem.addEventListener('click', obj.show.bind(obj));
elem.addEventListener('click', function () {
    obj.show();
});
```

# 2. 作用域

## 2.1

```
var person = 1;
function showPerson() {
    var person = 2;
    console.log(person);
}
showPerson();
```

## 2.2

```
var person = 1;
function showPerson() {
    console.log(person);
    var person = 2;
}
showPerson();
```

## 2.3

```
var person = 1;
function showPerson() {
```

```
console.log(person);

var person = 2;
function person() {}

showPerson();
```

## 2.4

```
var person = 1;
function showPerson() {
    console.log(person);

    function person() {}
    var person = 2;
}

showPerson();
```

## 2.5

```
for(var i = 0; i < 10; i++) {
    console.log(i);
}

for(var i = 0; i < 10; i++) {
    setTimeout(function(){
        console.log(i);
    }, 0);
}

for(var i = 0; i < 10; i++) {
    (function(i){
        setTimeout(function(){
            console.log(i);
        }, 0)
    })(i);
}

for(let i = 0; i < 10; i++) {
    console.log(i);
}
```

## JS内功修炼

### 专业术语

- 类、封装、继承、多态
- 构造函数、实例、对象字面量
- 命名空间
- 内置对象、宿主对象、本地对象

### 面向对象

js里还是没有'类'，js的面向对象，还是基于原型的，无论是ES5/还是ES6，ES6中引入的class，只是基于原型继承模型的语法糖。

JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

有兴趣的同学可以看一下ECMAScript的文档：

ECMA - 256页 - 开始定义类

ECMA - 260页 - 定义了什么是继承

### 创建对象的方法

#### 方法1：工厂模式(没有对象识别)

举个栗子

```
function body() {  
    var o = new Object();  
    o._bloodVolume = 100;  
    o._attackVolume = 500;  
    return o;  
}  
  
var monster = body();  
var monster2 = body();
```

#### 方法2：构造函数模式

js中可以使用方法(function)作为类，定义一个类与定义一个函数一致。使用new操作符，来创建新的对象。该function不显式的创造对象。

new的几个步骤：

- 1、创建一个新对象
- 2、将构造函数中的作用域指向该对象
- 3、执行构造函数中的代码
- 4、返回新对象

举个栗子

```
function Body() {  
    this._bloodVolume = 100;  
    this._attackVolume = 500;  
}  
  
var monster = new Body();
```

手写一个【new函数】

```
function Body() {  
    this._bloodVolume = 100;  
    this._attackVolume = 500;  
}  
  
function newOperation(constructFunc) {  
    const newObj = Object.create(null);  
    constructFunc.call(newObj);  
    return newObj;  
}  
  
var monster = newOperation(Body);
```

### 方法3：原型模式

我们创建的每个函数都有个prototype属性，这个属性是个指针，指向一个对象，而这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法。那么prototype就是通过调用构造函数而创建的那个对象实例的原型对象。

举个栗子

```
function Body() {}  
Body.prototype._bloodVolume = 100;  
Body.prototype._attackVolume = 500;  
  
var monster = new Body();
```

什么是原型？

1. 真正的原型，在构造函数中

2. 我们每声明一个函数，浏览器就会在内存中创建一个对象，在这个对象中增加一个属性，叫 `constructor` 指向我们的函数，并把我们的函数的 `prototype` 属性指向这个对象。
3. 用这个构造函数创建的对象都有一个不可访问的属性 `[[prototype]]`，这个属性就指向了构造函数的 `prototype`，在浏览器中，支持使用 `__proto__` 来访问这个对象。

使用原型对象的好处是，可以在创建出来的多个对象中，共享属性和方法。

手写一个【new函数】

```
function Body() {}
Body.prototype._bloodVolume = 100;
Body.prototype._attackVolume = 500;

function newOperation(constructFunc) {
    const newObj = Object.create(constructFunc.prototype);
    return newObj;
}

var monster = newOperation(Body);
```

这种模式下，每一个生成的对象都有一个属性 `[[Prototype]]`，浏览器厂商在具体实现的时候，保留了一个 `__proto__` 属性，用以让我们访问 `[[Prototype]]`

[Prototype]的ECMA解释

19.2.4.3 prototype Function instances that can be used as a constructor have a prototype property. Whenever such a Function instance is created another ordinary object is also created and is the initial value of the function's prototype property. Unless otherwise specified, the value of the prototype property is used to initialize the `[[Prototype]]` internal slot of the object created when that function is invoked as a constructor.

#### 方法4：组合模式

组合模式是构造函数和原型模式一起使用，构造函数模式用于定义实例属性，原型模式用于定义方法和共享的属性。

举个栗子

```
function Person() {
    this._attackVolume = 100;
}
Person.prototype = {
    attack(body) {
        body.bloodVolume -= this.attackVolume - body.defenseVolume;
    }
};
var hero = new Person();
```

手写一个【new函数】

```

function Person() {
    this._attackVolume = 100;
}
Person.prototype = {
    attack(body) {
        body.bloodVolume -= this.attackVolume - body.defenseVolume;
    }
};

function newOperation(constructFunc) {
    const newObj = Object.create(constructFunc.prototype);
    constructFunc.call(newObj);
    return newObj;
}

var hero = newOperation(Person);

```

es6的写法

```

// class expression
var Person = class {
    constructor(height, width) {}
}

// class declaration
class Person {
    constructor(height, width) {}
}

```

这里我们看下babel编译后的样子

## 原型链

利用js的原型模型，代码读取某个对象的某个属性的时候，都会按照属性名执行一次搜索，在实例中找到了则返回，如果没找到，则继续在当前实例的原型对象中搜索，直到找到为止。如果还没找到，则继续该原型对象的原型对象，以此类推，直到搜索到Object对象为止，这样就形成了一个原型指向的链条，专业术语称之为原型链。

## 继承

### 方法1：原型链继承

```

//父类型
function Body() {
    this.volumes = {
        _bloodVolume: 1000,

```

```

        _attackVolume: 500,
        _defenseVolume: 200
    );
}
Body.prototype.attacked = function (body) {
    this.volumes._bloodVolume -= body.getAttackVolume() - this.volumes._defenseVolume;
};

//子类型
function Monster() {};
Monster.prototype = new Body();
Monster.prototype.attacked = function () {
    this.volumes._bloodVolume -= 1;
};

var monster = new Monster();
var monster2 = new Monster();
monster.attacked();
console.log(monster2.volumes._bloodVolume);

```

注意：这种方式，monster改变`_bloodVolume`后，monster2的`_bloodVolume`也被改变了。

## 方法2：借用构造函数继承

```

// 父类
function Body() {
    this._bloodVolume = 1000;
    this._attackVolume = 500;
    this._defenseVolume = 200;
}
Body.prototype.attacked = function (body) {
    this._bloodVolume -= body.getAttackVolume() - this._defenseVolume;
};

// 子类
function Monster() {
    Body.call(this);
};

var monster = new Monster();

```

注意：这种方式，Monster无法继承父类prototype上的方法和属性

## 方法3：原型链+借用构造函数的组合继承

```

// 父类
function Body() {

```

```
this._bloodVolume = 1000;
this._attackVolume = 500;
this._defenseVolume = 200;
}
Body.prototype.attacked = function (body) {
    this._bloodVolume -= body.getAttackVolume() - this._defenseVolume;
};

// 子类
function Monster() {
    Body.call(this);
}
Monster.prototype = new Body();

var monster = new Monster();
```

## 方法4：寄生组合继承

```
// 父类
function Body() {
    this._bloodVolume = 1000;
    this._attackVolume = 500;
    this._defenseVolume = 200;
}
Body.prototype.attacked = function (body) {
    this._bloodVolume -= body.getAttackVolume() - this._defenseVolume;
};

// 子类
function Monster() {
    this.name = 'asd';
    Body.call(this);
}
Monster.prototype = Object.create(Body.prototype);

var monster = new Monster();
```

## 方法5：es6的class

```
// 父类
class Body {
    constructor() {
        this._bloodVolume = 1000;
        this._attackVolume = 500;
        this._defenseVolume = 200;
    }
}
```

```
attacked(body) {
    this._bloodVolume -= body.getAttackVolume() - this._defenseVolume;
}
}

// 子类
class Monster extends Body {
    constructor() {
        super();
    }

    attacked() {
        this._bloodVolume -= 1;
    }
}

var monster = new Monster();
```

这里我们看下babel编译后的样子

## 相关面试题

### 1. 面向对象

#### 1.1

```
function Person() {
    this.name = 1;
    return {};
}
var person = new Person();
console.log('name:', person.name);
```

#### 1.2

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    show: function () {
        console.log('name is:', this.name);
    }
};
var person = new Person();
person.show();
```

### 1.3

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    name: 2,
    show: function () {
        console.log('name is:', this.name);
    }
};
var person = new Person();

Person.prototype.show = function () {
    console.log('new show');
};

person.show();
```

### 1.4

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    name: 2,
    show: function () {
        console.log('name is:', this.name);
    }
};
var person = new Person();
var person2 = new Person();

person.show = function () {
    console.log('new show');
};

person2.show();
person.show();
```

## 2、综合题

```
function Person() {
    this.name = 1;
}

Person.prototype = {
    name: 2,
```

```
show: function () {
    console.log('name is:', this.name);
}
};

Person.prototype.show();

(new Person()).show();
```

## es6深入理解

### 专业术语

- 严格模式、非严格模式

### babel

ECMAScript 6(ES6)的发展速度非常之快，但现代浏览器对ES6新特性支持度不高，所以要想在浏览器中直接使用ES6的新特性就得借助别的工具来实现。

### babel原理

- babel 本质上就是在操作抽象语法树（Abstract Syntax Tree, AST）来完成代码的转译。
- babel的工作步骤：解析-转换-生成器
- babel 对于 AST 的遍历是深度优先遍历，对于 AST 上的每一个分支 babel 都会先向下遍历走到尽头，然后再向上遍历退出刚遍历过的节点，然后寻找下一个分支。

### 在线工具

<http://babeljs.io/repl/>

### 安装步骤

- npm install
  - "@babel/cli": "^7.5.5",
  - "@babel/core": "^7.5.5",
  - "@babel/preset-env": "^7.5.5",
  - "babel-plugin-proxy": "^1.1.0",
  - "babel-polyfill": "6.26.0",
- 安装语法插件
  - .babelrc 文件配置
  - babel src -d dist

### 语法插件

```
# ES2015转码规则
npm install --save-dev babel-preset-es2015

# react转码规则
```

```

npm install --save-dev babel-preset-react

# ES7不同阶段语法提案的转码规则（共有4个阶段），选装一个
npm install --save-dev babel-preset-stage-0
npm install --save-dev babel-preset-stage-1
npm install --save-dev babel-preset-stage-2
npm install --save-dev babel-preset-stage-3

```

### .babelrc

```

{
  "presets": [
    [
      "@babel/env",
      {
        "targets": {
          "edge": "7",
          "firefox": "60",
          "chrome": "50",
          "safari": "11.1",
        },
        "useBuiltIns": "usage"
      }
    ]
  ],
  "plugins": ["proxy"]
}

```

## 块级作用域绑定

### 变量提升机制

在函数作用域或者全局作用域中，通过var声明的变量，无论在哪里声明，都会被当成在当前作用域顶部声明，这就是变量提升（Hoisting）

### 块级作用域

块级声明用于声明在制定块的作用域之外无法访问的变量。块级作用域存在于：

- 函数内部
- 块中（大括号之间的区域）

### let声明

对变量的声明，不能变量提升。

举个栗子

```
if(condition) {  
    let value = 'bule';  
    console.log(value);  
} else {  
    // ...  
}  
console.log(value);
```

## 注意

### 1、禁止重复声明

```
var abc = 30;  
let abc = 40; // 会抛错  
  
var abc = 30;  
if (condition) {  
    let abc = 40; // 不会抛错  
}
```

### 2、块级不会提升

```
console.log(abc); // 会抛错 (临时死区Temporal Dead Zone)  
console.log(typeof abc); // 会抛错  
let abc = 30;  
  
console.log(abc); // undefined  
var abc = 30;
```

## const声明

对常量的声明，同样也不会变量提升。目前的普遍推荐方式是：默认使用const，只有确实需要改变的变量时，才用let。

### 举个栗子

```
const name = 'zhaowa';
```

## 注意

### 1、有效的常量声明

```
const abc = 30; // 有效  
const abc; // 无效  
const abc = 40; // 重复声明，会报错
```

### 2、用const声明对象时，不允许修改绑定，但允许修改值

```
const persion = {
  age: 18
};
persion.age = 19; // 有效
```

## 循环中的块级作用域

长久以来，var声明让开发者在循环中创建函数变的异常困难，因为变量到了循环之外仍能访问。为了解决这个问题，开发者在循环中使用立即调用函数表达式。

举个栗子

```
var funcs = [];
for (var i = 0; i < 10; i++) {
  funcs.push(function() {
    console.log(i);
  });
}
funcs.forEach(function(func) {
  func(); // 输出10次10
});

// 使用自执行函数
var funcs = [];
for (var i = 0; i < 10; i++) {
  funcs.push(function(i) {
    return function() {console.log(i);}
  )(i);
}
funcs.forEach(function(func) {
  func(); // 输出0到9
});

// 使用let
var funcs = [];
for (let i = 0; i < 10; i++) {
  funcs.push(function() {
    console.log(i);
  });
}
funcs.forEach(function(func) {
  func(); // 输出0到9
});
```

## 全局块作用域绑定

let和const与var的另外一个区别是他们在全局作用域中的行为。这意味着var很可能无意间覆盖了一个已经存在的全局属性。

举个栗子

```
var RegExp = 'hello';
console.log(window.RegExp); // hello

let RegExp = 'hello';
console.log(RegExp); // hello
console.log(window.RegExp === RegExp); // false
```

## 函数

### 函数形参的默认值

举个栗子

```
// es5效果
function request(url, timeout, callback) {
    timeout = (typeof timeout !== 'undefined') ? timeout : 1000;
    callback = (typeof callback !== 'undefined') ? callback : function() {};
}

// es6效果
function request(url, timeout = 1000, callback = function() {}) {
    console.log(arguments.length); // 如果只传了url, 那arguments.length为1
}
```

注意

1、前面的参数默认值不能引用后面的参数值

```
function add(a,b=1){return a+b};
console.log(add(1,1)); //2

function addition(a=b,b){return a+b};
console.log(addition(undefined,1)); // 报错
```

### 处理无命名参数

在函数的命名参数前添加三个点(...)就标识这是一个不定参数，是个数组。

举个栗子

```
function checkArgs(...keys) {
    console.log(args.length); // 2
    console.log(arguments.length); // 2
    console.log(args[0], arguments[0]); // 1 1
    console.log(args[1], arguments[1]); // 2 2
```

```
    }
    checkArgs(1, 2);
```

## 注意

1、...必须是最后一个参数

```
function demo(object, ...keys, last) { // 报错
}
```

2、setter中不能使用...

```
let object = {
    set name(...value) { // 报错
        }
}
```

## 扩展运算符

执行函数时，传入展开后的参数

举个栗子

```
abc(...value, 0);
```

## 箭头函数

es6中的新特性，通过 `参数 => 函数体` 的方式调用。

特性

- 没有this、super、arguments, new.target绑定。this、super、arguments以及内部函数的new.target的值由所在的最近的外部非箭头函数来决定。
- 不能使用new来调用。箭头函数没有[[Construct]]方法，因此不能被用为构造函数，使用new调用函数会抛出错误。
- 没有原型。没有使用new，因此没有prototype属性。
- 不能修改this的绑定。不能通过call(),apply()以及bind()方法修改this。
- 没有arguments对象。需要通过命名参数和不定参数这两种形式访问函数的参数。
- 不允许使用重复的具名参数。箭头函数不允许拥有重复的具名参数，无论是否在严格模式下。

举个栗子

```
// es6
let sum = (num1, num2) => num1 + num2;
// es5
```

```

var sum = function(num1, num2) {
    return num1 + num2;
}

// es6
let person = ((name) => {
    return {
        getName: function() {
            return name;
        }
    };
})('zhaowa');
console.log(person.getName());
// es5
var person = function(name) {
    return {
        getName: function() {
            return name;
        }
    };
}('zhaowa');
console.log(person.getName());

```

## 注意

1、箭头函数没有arguments对象，但可以访问上一层函数的arguments对象

```

function outer() {
    return () => arguments[0];
}
var result = outer(1,2,3);
console.log(result()); // 1

```

2、如果需要返回对象的话，需要使用圆括号()将对象包裹起来，为了防止对象字面量被认为是函数体语句。

## 尾调用优化

主要是为了解决递归函数的爆栈问题，降低内存占用。需要满足以下条件：

- 尾调用不能引用当前栈帧中的变量。
- 进行尾调用的函数在尾调用返回结果后不能做额外任何操作。
- 尾调用的结果作为当前函数的返回值。

## 扩展对象的功能性

### 对象字面量语法扩展

## 举个栗子

### 1、属性初始值的简写

```
// es5
function demo(name, age) {
    return {
        name: name,
        age: age
    }
}

// es6
function demo(name, age) {
    return {
        name,
        age
    }
}
```

### 2、对象方法的简写

```
// es5
var demo = {
    name: 'zhaowa',
    getName: function() {
        console.log(this.name);
    }
}

// es6
let demo = {
    name: 'zhaowa',
    getName() {
        console.log(this.name);
    }
}
```

### 3、可计算属性名

```
// es5
var demo = {
    'first name': 'xiaowa'
};
console.log(demo['first name']);

// es6
let firstName = 'first name';
```

```
let demo = {
  [firstName]: 'xiao'
};
console.log(demo[firstName]);
```

## es6新增方法

Object.is()

Object.assign()

## 重复的对象字面量属性

举个栗子

```
// es5
var person = {
  name: 'xiaowa',
  name: 'yuanxin' // 严格模式下报错
};

// es6
var person = {
  name: 'xiaowa',
  name: 'yuanxin' // 严格模式和非严格模式都正常
};
console.log(person.name); // yuanxin
```

## 增强对象原型

1、使用setPrototypeOf修改对象原型

```
let person = {
  getName(){
    return 'hello';
  }
}
let dog ={
  getName(){
    return 'world';
  }
}

let friend = Object.create(person);
console.log(friend.getName()); //hello
console.log(Object.getPrototypeOf(friend)===person); //true

Object.setPrototypeOf(friend,dog);
```

```
console.log(friend.getName()); //world
console.log(Object.getPrototypeOf(friend) === dog); //true
```

## 2、super的引用，来访问原型中的方法

```
let person = {
  getName(){
    return 'hello';
  }
}
let dog ={
  getName(){
    return super.getName() + ' world';
  }
}
Object.setPrototypeOf(dog, person);
console.log(dog.getName()); //hello world
```

# 解构

## 对象的解构

### 解构语法

#### 举个栗子

```
let person ={
  name: 'hello',
  age: 18
}
let {name, age} = person;
console.log(name); //hello
console.log(age); //18
```

#### 解构赋值与默认值

#### 举个栗子

```
let person ={
  name: 'hello',
  age: 18
}
let name = 'world';
let age = 20;
({name, age, value = true} = person); // 这里是圆括号包裹
console.log(name); // hello
console.log(age); // 18
```

```
console.log(value); // true
```

## 为不同名的变量赋值

举个栗子

```
let person = {  
    name:'hello',  
    age:18  
};  
({name: localName, age: localAge, value = true} = person); // 等号是赋默认值, 冒号是为别名赋  
值  
console.log(localName); //hello  
console.log(localAge); //18  
console.log(value); // true
```

## 嵌套的对象解构

举个栗子

```
let person = {  
    name: 'zhaowa',  
    school: {  
        primary: {  
            start: 2000,  
            column: 2007  
        },  
        middle: {  
            start: 2007,  
            end: 2010  
        },  
        university: {  
            start: 2010,  
            end: 2014  
        }  
    }  
};  
let{school: {university}} = person;  
console.log(university.start); // 2010  
console.log(university.end); // 2014
```

## 数组的解构

解构语法

举个栗子

```
let arr = [1,2,3];
```

```
let [first, second] = arr;
console.log(first); // 1
console.log(second); // 2
```

## 解构赋值与默认值

举个栗子

```
let arr = [1, 2, 3];
[first, second, third = 100] = arr;
[first, second] = [second, first]; // 变量值互换
console.log(first); // 2
console.log(second); // 1
console.log(third); // 100
```

## 剩余项

举个栗子

```
let aaa = [1, 2, 3, 4, 5];
let [first, ...subArr] = aaa;
console.log(first); // 1
console.log(subArr.length); // 4
console.log(subArr[0]); // 2

// 深拷贝数组
let bbb = [1, 2, 3, 4, 5];
let [...clonedArr] = bbb;
console.log(clonedArr); // [1, 2, 3, 4, 5]
```

## 参数解构

对可选参数设置默认值

举个栗子

```
function setCookie(name, value,
{
  secure = false,
  path = "/",
  domain = "example.com",
  expires = new Date(Date.now() + 3600000000)
} = {}
) {
// ...
```

}

## Set与Map

### Set

ES6中新增了Set类型是一种有序列表，其中包含一些相互独立的非重复值。Set在存放对象时，实际上是存放的是对象的引用，如果所存储的对象被置为null，但是Set实例仍然存在的话，对象依然无法被垃圾回收器回收，从而无法释放内存。

举个栗子

```
// 创建、检验、删除
let set= new Set();
set.add(5);
set.add('5');
set.add(5);
console.log(set.size); // 2 只存放不重复的值
set.has(5); // true 检验某值是否存在
set.delete('5'); // 删除某值

// 与数组互转
let set = new Set([1, 2, 3]);
let arr = [...set];
console.log(arr); // [1,2,3]

let arr = [1, 2, 3];
let set = new Set(arr);
console.log(set.length); // 3

// forEach方法遍历
let set = new Set([1,2,3,3,3,3]);
set.forEach(function (value, key) { // 为了和Map一致，set的value和key的值一样
    console.log(value);
    console.log(key);
});

// 使用WeakSet解决无法垃圾回收的问题
let set = new WeakSet();
let key = {};
set.add(key);
console.log(set.has(key)); //true
set.delete(key);
console.log(set.has(key));
```

### 注意

- 对于Weak Set实例，若调用了add()方法时传入了非对象的参数，则会抛出错误。如果在has()或

者`delete()`方法中传入了非对象的参数则会返回`false`；

- `Weak Set`集合不可迭代，因此不能用于`for-of`循环；
- `Weak Set`集合不暴露任何迭代器（例如`keys()`与`values()`方法），所以无法通过程序本身来检测其中的内容；
- `Weak Set`集合不支持`forEach()`方法；
- `Weak Set`集合不支持`size`属性；

## Map

ES6中提供了`Map`数据结构，能够存放键值对。

举个栗子

```
// set、get、has、delete、clear
let map = new Map(['year', 2018]);
map.set('name', 'zhaowa');
console.log(map.get('title')); // zhaowa
console.log(map.has('title')); // true
map.delete('title');
map.clear();
console.log(map.size); // 0

// forEach
let map = new Map([['title', 'hello world'], ['year', '2018']]);
map.forEach((value, key) => {
  console.log(value);
  console.log(key);
});

// Weak Map
let map = new WeakMap([[key, 'hello'], [key2, 'world']]);
```

## js的类

回想一下es5如何创建类的？

注意

- 函数声明可以被提升，而类和`let`类似，不能被提升。真正执行声明语句之前，他们会一直存在于临时死区中。
- 类声明中的所有代码会自动运行在严格模式下，无法强行让代码脱离严格模式执行。
- 类的所有方法都是不可枚举的；
- 每个类都有一个名为`[[Constructor]]`的内部方法，通过`new`调用那些没有`[[Constructor]]`的方法会报错。
- 使用非`new`关键字调用构造函数会抛出错误；
- 在类中修改类名会抛出错误；

## 创建类

举个栗子

```
// 方式1: 类声明
class PersonClass{
    constructor(name){
        this.name = name;
    }
    sayName(){
        console.log(this.name);
    }
}
let person = new PersonClass("hello class");
person.sayName();

// 方式2: 类表达式
let PersonClass = class {
    constructor(name){
        this.name = name;
    }
    sayName(){
        console.log(this.name);
    }
}
let person = new PersonClass("hello class");
person.sayName(); //hello class

// 方式3: 当做参数传入
function createObj(classDef){
    return new classDef();
}
let person = createObj(class{
    sayName(){
        console.log('hello'); //hello
    }
});
person.sayName();

// 立即调用构造器
let person = new class{
    constructor(name){
        this.name = name;
    }
    sayName(){
        console.log(this.name);
    }
}('hello world');
person.sayName(); //hello world
```

## 访问器属性

自有属性需要在类构造器中创建，而类还允许创建访问器属性。为了创建一个getter，要使用get关键字，并要与后面的标识符之间留出空格；创建setter使用相同的方式，只需要将关键字换成set即可：

举个栗子

```
class PersonClass{
    constructor(name){
        this.name = name;
    }
    get name(){
        return name; //不要使用this.name会导致无限递归
    }

    set name(value){
        name=value; //不要使用this.value会导致无限递归
    }
}
let person = new PersonClass('hello');
console.log(person.name); // hello
person.name = 'world';
console.log(person.name); //world
let descriptor = Object.getOwnPropertyDescriptor(PersonClass.prototype, 'name');
console.log('get' in descriptor); //true
```

## 静态成员

ES6的类简化了静态成员的创建，只要在方法与访问器属性的名称前添加static关键字即可：

举个栗子

```
class PersonClass {
    // 等价于 PersonType 构造器
    constructor(name) {
        this.name = name;
    }
    static create(name) {
        return new PersonClass(name);
    }
}
let person = PersonClass.create("Nicholas");
```

## 类继承

使用关键字extends可以完成类继承，同时使用super关键字可以在派生类上访问到基类上的方法，包括构造器方法：

## 举个栗子

```

class Rec{
    constructor(width, height){
        this.width = width;
        this.height = height;
    }
    getArea(){
        return this.width * this.height;
    }
}
class Square extends Rec{
    constructor(width,height){
        super(width, height);
    }
}
let square = new Square(100, 100);
console.log(square.getArea()); //10000

```

## 注意

- 如果基类中包含了静态成员，那么这些静态成员在派生类中也是可以使用的。注意：静态成员只能通过类名进行访问，而不是使用对象实例进行访问；

## 从表达式中派生类

在ES6中最大的能力是从表达式中导出类的功能，只要一个表达式可以被解析成一个函数并具有`[[Constructor]]`属性以及原型，就可以用`extends`进行派生。由于`extends`后面能够接收任意类型的表达式，创造了更多的可能性，动态的确定类的继承目标。

## 举个栗子

```

let SerializableMixin = {
    serialize() {
        return JSON.stringify(this);
    }
};
let AreaMixin = {
    getArea() {
        return this.length * this.width;
    }
};
function mixin(...mixins) {
    var base = function() {};
    Object.assign(base.prototype, ...mixins);
    return base;
}
class Square extends mixin(AreaMixin, SerializableMixin) {
    constructor(length) {
        super();
    }
}

```

```
    this.length = length;
    this.width = length;
}
}
let x = new Square(3);
console.log(x.getArea()); // 9
console.log(x.serialize()); // {"length":3,"width":3}"
```

## 继承内置对象

在ES6中能够通过`extends`继承JS中内置对象，例如：

举个栗子

```
class MyArray extends Array {
// 空代码块
}
let colors = new MyArray();
colors[0] = "red";
console.log(colors.length); // 1
colors.length = 0;
console.log(colors[0]); // undefined
```

## 改进的数组功能

### 创建数组

- `Array.of()`
- `Array.from()`

### 数组上所有的新方法

- `find()`和`findIndex()`
- `fill()`
- `copyWithin()`

## Promise与异步编程

### Promise基础

每个 Promise 都会经历一个短暂的生命周期，初始为进行中状态（pending），也被认为是未处理的（unsettled）。一旦异步操作结束，Promise就会被认为是已处理的（settled），操作结束后Promise可能会进入下面两种状态之一：

- `fulfilled`: Promise 异步操作成功完成

- rejected: 由于程序错误或者其他原因, Promise 的异步操作未成功完成

内部属性[[PromiseState]]被用来表示3种状态: "pending"、"fulfilled"、"rejected", 属性不会暴露在Promise 对象上, , 当改变状态时, 通过then()方法采取一些特定的行动。

举个栗子

```
promise.catch(function(err) {
    // 拒绝
    console.error(err);
});

promise.then(null, function(err) {
    // 拒绝
    console.error(err);
});

promise.then(function(res) {
    console.log(res);
}).catch(function(err) {
    console.error(err);
});
```

## 创建未完成的Promise

```
let promise = new Promise(function(resolve, reject) {
    console.log('hi, promise');
    resolve();
});
promise.then(() => {
    console.log('hi, then');
});
console.log('hi');
```

输出:  
hi, promise  
hi  
hi then

## 创建已完成的Promise

有以下2种方式:

```
let promise = Promise.resolve('zhaowa');
promise.then(res => {
    console.log(res); // zhaowa
});
```

```
let reject = Promise.reject('yuanxin');
reject.catch(err => {
    console.log(err); // yuanxin
})
```

## 执行器错误

如果执行器内部抛出一个错误，则Promise的拒绝处理会被调用

```
let promise = new Promise(function(resolve, reject) {
    throw new Error('Error!');
});
promise.catch(function(msg) {
    console.log(msg); // error
});
```

## 串联Promise

每次调用then()方法或者catch()方法会返回另外一个Promise，只有当第一个被拒绝或者完成后，第二个才会被解决。下面看下串联的几种场景。

举个栗子

```
// 捕获错误
let p1 = new Promise(function(resolve, reject) {
    resolve('zhaowa');
});
p1.then(res => {
    console.log(res);
    throw new Error('Error!');
}).catch(err => {
    console.log(err);
});

// 传递值
let p1 = new Promise(function(resolve, reject) {
    resolve(100);
});
p1.then(res => res + 1)
.then(res => {
    console.log(res); // 101
});

// 传递promise
let p1 = new Promise(function(resolve, reject){
    resolve('zhaowa');
});
```

```
let p2 = new Promise(function(resolve, reject){  
    resolve('yuanxin');  
})  
p1.then(res => {  
    console.log(res); // zhaowa  
    return p2;  
}).then(res => {  
    console.log(res); // yuanxin  
});
```

## 响应多个Promise

监听多个Promise来决定下一步的操作。

举个栗子

```
//Promise.all()  
let p1 = new Promise(function(resolve, reject) {  
    resolve(1);  
});  
let p2 = new Promise(function(resolve, reject) {  
    resolve(2);  
});  
let p3 = new Promise(function(resolve, reject) {  
    resolve(3);  
});  
let p4 = Promise.all([p1, p2, p3]);  
p4.then([p1res, p2res, p3res] => {  
    // 需要所有的都成功，才走到这里  
});  
  
// Promise.race()  
let p1 = new Promise(function(resolve, reject) {  
    resolve(1);  
});  
let p2 = new Promise(function(resolve, reject) {  
    resolve(2);  
});  
let p3 = new Promise(function(resolve, reject) {  
    resolve(3);  
});  
let p4 = Promise.race([p1, p2, p3]);  
p4.then(res => {  
    // 只要有一个先成功，就会走到这里  
});
```

## 核心Promises / A +规范

---

## 2.1 Promise状态

一个promise必须处于三种状态之一： 请求态（pending）， 完成态（fulfilled）， 拒绝态（rejected）

### 2.1.1 当promise处于请求状态（pending）时

2.1.1.1 promise可以转为fulfilled或rejected状态

### 2.1.2 当promise处于完成状态（fulfilled）时

2.1.2.1 promise不能转为任何其他状态

2.1.2.2 必须有一个值，且此值不能改变

### 2.1.3 当promise处于拒绝状态（rejected）时

2.1.3.1 promise不能转为任何其他状态

2.1.3.2 必须有一个原因（reason），且此原因不能改变

## 2.2 then方法

promise必须提供then方法来存取它当前或最终的值或者原因。

promise的then方法接收两个参数：

`promise.then(onFulfilled, onRejected)`

复制代码

2.2.1 onFulfilled和onRejected都是可选的参数：

2.2.1.1 如果 onFulfilled不是函数，必须忽略

2.2.1.1 如果 onRejected不是函数，必须忽略

2.2.2 如果onFulfilled是函数：

2.2.2.1 此函数必须在promise 完成(fulfilled)后被调用，并把promise 的值作为它的第一个参数

2.2.2.2 此函数在promise完成(fulfilled)之前绝对不能被调用

2.2.2.2 此函数绝对不能被调用超过一次

2.2.3 如果onRejected是函数：

2.2.2.1 此函数必须在promise rejected后被调用，并把promise 的reason作为它的第一个参数

2.2.2.2 此函数在promise rejected之前绝对不能被调用

2.2.2.2 此函数绝对不能被调用超过一次

2.2.4 在执行上下文堆栈（execution context）仅包含平台代码之前，不得调用 onFulfilled和onRejected

3.1

2.2.5 onFulfilled和onRejected必须被当做函数调用(i.e. with no this value-->这里不会翻.....).

3.2

2.2.6 then可以在同一个promise里被多次调用

2.2.6.1 如果/当 promise 完成执行（fulfilled），各个相应的onFulfilled回调

必须根据最原始的then 顺序来调用

2.2.6.2 如果/当 promise 被拒绝（rejected），各个相应的onRejected回调

必须根据最原始的then 顺序来调用

2.2.7 then必须返回一个promise 3.3

```
promise2 = promise1.then(onFulfilled, onRejected);
```

复制代码

2.2.7.1 如果onFulfilled或onRejected返回一个值x，运行

Promise Resolution Procedure [[Resolve]](promise2, x) 2.3

2.2.7.2 如果onFulfilled或onRejected抛出一个异常e,promise2

必须被拒绝 (rejected) 并把e当作原因

2.2.7.3 如果onFulfilled不是一个方法，并且promise1已经完成 (fulfilled) , promise2必须使用与promise1相同的值来完成 (fulfilled)

2.2.7.4 如果onRejected不是一个方法，并且promise1已经被拒绝 (rejected) , promise2必须使用与promise1相同的原因来拒绝 (rejected)

### 2.3 Promise解决程序

promise解析过程 是一个抽象操作，它将promise和value作为输入，我们将其表示为[[Resolve]] (promise, x)。

如果x是thenable的，假设x的行为至少有点像promise，

它会尝试让promise采用x的状态。不然就会用x来完成promise

只要它们公开一个Promises / A +兼容的方法，对thenables的这种处理允许promise实现进行互操作，

它还允许Promises / A +实现使用合理的then方法“同化”不一致的实现。

运行[[Resolve]](promise, x),执行以下步骤:

2.3.1 如果promise和x引用同一个对象，则用TypeError作为原因拒绝 (reject) promise。

2.3.2 如果x是一个promise,采用promise的状态3.4

2.3.2.1 如果x是请求状态(pending),promise必须保持pending直到xfulfilled或rejected

2.3.2.2 如果x是完成态(fulfilled), 用相同的值完成fulfillpromise

2.3.2.2 如果x是拒绝态(rejected), 用相同的原因rejectpromise

2.3.3另外，如果x是个对象或者方法

2.3.3.1 让x作为x.then. 3.5

2.3.3.2 如果取回的x.then属性的结果为一个异常e,用e作为原因reject promise

2.3.3.3 如果then是一个方法，把x当作this来调用它，

第一个参数为 resolvePromise, 第二个参数为rejectPromise, 其中:

2.3.3.3.1 如果/当 resolvePromise被一个值y调用，运行 [[Resolve]](promise, y)

2.3.3.3.2 如果/当 rejectPromise被一个原因r调用，用r拒绝 (reject) promise

2.3.3.3.3 如果resolvePromise和 rejectPromise都被调用，或者对同一个参数进行多次调用，第一次调用执行，任何进一步的调用都被忽略

2.3.3.3.4 如果调用then抛出一个异常e,

2.3.3.3.4.1 如果resolvePromise或 rejectPromise已被调用，忽略。

2.3.3.3.4.2 或者， 用e作为reason拒绝 (reject) promise

2.3.3.4 如果then不是一个函数，用x完成(fulfill)promise

2.3.4 如果 x既不是对象也不是函数，用x完成(fulfill)promise

如果一个promise被一个thenable resolve,并且这个thenable参与了循环的thenable环，

[[Resolve]](promise, thenable)的递归特性最终会引起[[Resolve]](promise, thenable)再次被调用。

遵循上述算法会导致无限递归，鼓励（但不是必须）实现检测这种递归并用包含信息的TypeError作为reason拒绝 (reject) 3.6

## 代理和反射

代理可以拦截js引擎内部目标的底层对象操作，这些底层操作被拦截后会触发相应特定操作的陷阱函数。调用 new proxy() 可创建代替其他目标 (target) 对象的代理，它虚拟化了目标，所以二者看起来功能一致。

反射api以Reflect对象的形式出现，对象中方法的默认特性与相同的底层操作一致。

思考一下：vue里是如何做到修改数据后，视图就变化了？

## 创建代理

举个栗子

```
// 不使用任何陷阱的处理等于转发
let target = {};
let proxy = new Proxy(target, {});
proxy.msg = 'zhaowa';
console.log(proxy.msg); // zhaowa
console.log(target.msg); // zhaowa
```

## 常用陷阱

结合使用代理陷阱和反射api方法可以过滤一些操作，他们默认执行内置行为，只在某些条件下才会表现不同的行为。

代理陷阱：

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

### set陷阱

参数

- trapTarget: 用于接受属性（代理的目标）对象；

- key: 要写入的属性键;
- value: 被写入属性的值;
- receiver: 操作发生的对象 (通常是代理)

举个栗子

```
//set陷阱函数
let target = {
    name: 'target'
}
let proxy = new Proxy(target, {
    set(tarpTarget, key, value, receiver){
        if(!tarpTarget.hasOwnProperty(key)){
            if(isNaN(value)){
                throw new Error('属性必须是数字');
            }
        }
        return Reflect.set(tarpTarget, key, value, receiver);
    }
});
proxy.msg = 'hello proxy'; // 属性必须是数字

proxy.const = 1;
console.log(proxy.const); // 1
console.log(target.const); // 1

proxy.name = 'zhaowa';
console.log(proxy.name); // zhaowa 可以给已有属性赋值成非数字
console.log(target.name); // zhaowa
```

## get陷阱

参数

- trapTarget: 被读取属性的原对象
- key: 要读取的属性键
- receiver: 操作发生的对象 (通常是代理)

举个栗子

```
let target = {
    name: 'hello world'
};
let proxy = new Proxy(target, {
    get(tarpTarget, key, receiver){
        if(!(key in tarpTarget)){ // 这里检查tarpTarget而不是receiver的原因，是为了避免
            receiver代理中有has陷阱
            throw new Error('不存在该对象');
        }
        return Reflect.get(tarpTarget, key, receiver);
    }
});
```

```
    }
});

console.log(proxy.name); //hello world
console.log(proxy.age); // Uncaught Error: 不存在该对象
```

## has陷阱

### 参数

- trapTarget: 读取属性的对象
- key: 要检查的属性键

### 举个栗子

```
let target = {
  value: 42,
  name: 'target'
};
let proxy = new Proxy(target, {
  has(trapTarget, key){
    if(Object.is(key, 'value')){
      return false;
    }
    Reflect.has(trapTarget, key);
  }
});
console.log('value' in proxy); //false
```

## delete陷阱

### 参数

- trapTarget: 读取属性的对象
- key: 要检查的属性键

### 举个栗子

```
let target = {
  name: "target",
  value: 42
};
let proxy = new Proxy(target, {
  deleteProperty(trapTarget, key) {
    if (key === "value") {
      return false; // 确保 value 属性不被删除
    } else {
      return Reflect.deleteProperty(trapTarget, key);
    }
  }
});
```

```
// 尝试删除 proxy.value
console.log("value" in proxy); // true
let result = delete proxy.value;
console.log(result); // false
```

## 可被撤销的代理

当 revoke() 函数被调用后，就不能再对该 proxy 对象进行更多操作

举个栗子

```
let target = {
  name: "target"
};
let { proxy, revoke } = Proxy.revocable(target, {});
console.log(proxy.name); // "target"
revoke();
// 抛出错误
console.log(proxy.name);
```

## 用模块封装代码

### 什么是模块

- 模块是自动运行在严格模式下并且没有本办法退出运行的js代码。
- 在模块顶部创建的变量不会自动被添加到全局作用域。
- 模块必须导出一些外部代码可以访问的元素。
- 模块可以从其他模块导入绑定。
- 在模块顶部， this 的值是 undefined
- 模块不支持html风格代码注释

### 导出基本语法

```
// 导出数据
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;
// 导出函数
export function sum(num1, num2) {
  return num1 + num2;
}
// 导出类
export class Rectangle {
  constructor(length, width) {
    this.length = length;
    this.width = width;
}
```

```
    }
}

// 此函数为模块私有
function subtract(num1, num2) {
    return num1 - num2;
}

// 定义一个函数.....
function multiply(num1, num2) {
    return num1 * num2;
}

// .....导出一个函数引用
export { multiply };
```

## 导入基本语法

```
// 导入多个绑定
import {sum, multiply, magicNumber} from "./example.js";
console.log(sum(1));

// 完全导入一个模块
import * as example from "./example.js";
console.log(example.sum(1, example.magicNumber)); // 8
console.log(example.multiply(1, 2)); // 2
```

## 导出和导入时重命名

```
function sum(num1, num2) {
    return num1 + num2;
}
export { sum as add };

import { add as sum } from './example.js'
console.log(typeof add); // "undefined"
console.log(sum(1, 2)); // 3
```

## 模块的默认值

```
// 不使用标识符
export default function(num1,num2){
    return num1+num2;
}

// 使用标识符
function sum(num1, num2) {
    return num1 + num2;
}
```

```

export default sum;

// 使用重命名语法
function sum(num1, num2) {
    return num1 + num2;
}
export {sum as default};

// 既导出了默认值，又导出非默认值
export let color = 'red';
export default function(num1,num2){
    return num1+num2;
}

```

## 导入默认值

```

// 只导入默认值
import sum from './example.js';

// 既导入默认值，又导入非默认值
import sum, { color } from './example.js';

// 对导入默认值重命名
import {default as sum, color} from './example.js'

```

## 对已导入的内容再导出

```

export {sum} from './example.js';
export { sum as add } from './example.js'
export * from './example.js';

```

## async-await

### async关键字

- 表明程序里面可能有异步过程：`async`关键字表明程序里面可能有异步过程，里面可以有`await`关键字；当然全部是同步代码也没关系，但是这样`async`关键字就显得多余
- 非阻塞：`async`函数里面如果有异步过程会等待，但是`async`函数本身会马上返回，不会阻塞当前线程，可以简单认为，`async`函数工作在主线程，同步执行，不会阻塞界面渲染，`async`函数内部由`await`关键字修饰的异步过程，会阻塞等待异步任务的完成再返回；
- `async`函数返回类型为Promise对象，相当于返回了`Promise.resolve()`；

### await关键字

- await只能在async函数内部使用
- await关键字后面跟Promise对象
- await不处理异步error: await是不管异步过程的reject(error)消息的，async函数返回的这个Promise对象的catch函数负责统一抓取内部所有异步过程的错误
- await的结果：如果得到的不是一个Promise对象，那么await表达式的运算结果就是它等到的东西；如果它等到的是一个Promise对象，await就忙起来了，它会阻塞其后面的代码，等着Promise对象resolve，然后得到resolve的值，作为await表达式的运算结果；虽然是阻塞，但async函数调用并不会造成阻塞，它内部所有的阻塞都被封装在一个Promise对象中异步执行，这也正是await必须用在async函数中的原因

举个栗子

```
async function demo() {
  let data = await asyncFunction();
  // ...
  return data;
}
```

举个栗子

```
// promise与async/await对比
function takeLongTime(n) {
  return new Promise(resolve => {
    setTimeout(() => resolve(n + 200), n);
  });
}

function step1(n) {
  console.log(`step1 with ${n}`);
  return takeLongTime(n);
}

function step2(n) {
  console.log(`step2 with ${n}`);
  return takeLongTime(n);
}

function step3(n) {
  console.log(`step3 with ${n}`);
  return takeLongTime(n);
}

// async/await方式
async function doIt() {
  console.time("doIt");
  const time1 = 300;
  const time2 = await step1(time1);
  const time3 = await step2(time2);
  const result = await step3(time3);
}
```

```

        console.log(`result is ${result}`);
        console.timeEnd("doIt");
    }
doIt();

// promise方式
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => step2(time2))
        .then(time3 => step3(time3))
        .then(result => {
            console.log(`result is ${result}`);
            console.timeEnd("doIt");
        });
}

doIt();

```

## Decorator装饰器

我们可以在不侵入原有代码的情况下，为代码增加一些额外的功能。一般都比较独立，不和原有逻辑耦合，只是做一层包装。

### 装饰器

- 首先它是一个函数。
- 这个函数会接收3个参数，分别是target、key和descriptor
- 它可以修改descriptor做一些额外的逻辑。

举个栗子

```

function memoize(target, key, descriptor) {
    ...
}
class Foo {
    @memoize;
    getFooById(id) {
        // ...
    }
}

```

### 装饰器的3个参数

举个栗子

```

function log(target, key, descriptor) {
    console.log(target);
    console.log(target.hasOwnProperty('constructor'));
    console.log(target.constructor);
    console.log(key);
    console.log(descriptor);
}

class Bar {
    @log;
    bar() {}
}

// {}

// true

// function Bar() { ... }

// bar

// {"enumerable":false,"configurable":true,"writable":true}

```

- key很明显就是当前方法名，我们可以推断出来用于属性的时候就是属性名
- descriptor显然是一个PropertyDescriptor，就是我们用于defineProperty时的那个东西。
- target是一个对象，然后是一个有constructor属性的对象，最后constructur指向的是Bar这个函数。所以这个就是Bar.prototype。

## 有几种装饰器

- 放在class上的“类装饰器”。
- 放在属性上的“属性装饰器”，这需要配合另一个Stage 0的类属性语法提案，或者只能放在对象字面量上了。
- 放在方法上的“方法装饰器”。
- 放在getter或setter上的“访问器装饰器”。

## 装饰器在什么时候执行

- 装饰器是在声明期就起效的，并不需要类进行实例化。类实例化并不会致使装饰器多次执行，因此不会对实例化带来额外的开销。
- 按编码时的声明顺序执行，并不会将属性、方法、访问器进行重排序。

举个栗子

```

function randomize(target, key, descriptor) {
    let raw = descriptor.initializer;
    descriptor.initializer = function() {
        let value = raw.call(this);
        value += '-' + Math.floor(Math.random() * 1e6);
        return value;
    };
}
class Alice {

```

```
@randomize;
name = 'alice';
}
console.log((new Alice()).name); // alice-776521
```

## VUE基础

### vue是什么

一套用于构建用户界面的渐进式框架，用来开发web的前端库。通过简单的api提供高效的数据绑定和灵活的组件系统。

特点： 1、轻量级 2、数据绑定 3、指令：通过指令对应的表达式的值的变化，就可以修改对应的dom  
4、插件化： ajax、router等都可以以插件形式加载

### 数据绑定

数据绑定是将数据和视图相关联，当数据发生变化时，可以自动更新视图。

举个栗子

```
// 文本插值
<p>{{text}}</p>

// 插入html
<p v-html="myhtml"></p>
myhtml: <span>我是一个html片段</span>

// 使用表达式
<p>{{number + 1}}</p>
```

### 指令

指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM

- v-if (v-else v-else-if)
- v-bind:xxx (:xxx)
- v-on:click (@click)
- v-for
- v-model
- v-slot

### 计算属性

模板内的表达式常用于简单的运算，当其过长或逻辑复杂时，会难以维护。因此为了简化逻辑，当某个属性依赖其他属性的值时，我们可以使用计算属性。

举个栗子

```

<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // 计算属性的 getter
    reversedMessage: function () {
      // `this` 指向 vm 实例
      return this.message.split('').reverse().join('')
    }
  }
})

```

## 计算属性缓存

那么，既然使用methods就可以实现，为什么还需要计算属性呢？原因就是计算属性是基于它的依赖缓存的，一个计算属性所依赖的数据发生变化时，它才会重新取值，所以只有值不改变，计算属性就不会更新。使用计算属性还是methods取决于是否需要缓存，当便利大数组和做大量计算时，应当使用计算属性。

## 过滤器

Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化。过滤器可以用在两个地方：双花括号插值和 v-bind 表达式 (后者从 2.1.0+ 开始支持)。过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符号指示：

举个栗子

```

<!-- 在双花括号中 -->
{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>
你可以在一个组件的选项中定义本地的过滤器：

filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}

```

## Class与Style绑定

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是属性，所以我们可以用 v-bind 处理

### class

举个栗子

```
<div class="static"
  v-bind:class="{ active: isActive, 'text-danger': hasError }"
></div>

// 数据
data: {
  isActive: true,
  hasError: false
}

// 结果渲染为:
<div class="static active"></div>
```

### style

举个栗子

```
<div v-bind:style="styleObject"></div>

// 数据
data: {
  styleObject: {
    color: 'red',
    fontSize: '13px'
  }
}
```

## 过渡

在进入/离开的过渡中，会有 6 个 class 切换。

- v-enter: 定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。
- v-enter-active: 定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。

- **v-enter-to:** 2.1.8版及以上 定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 v-enter 被移除), 在过渡/动画完成之后移除。
- **v-leave:** 定义离开过渡的开始状态。在离开过渡被触发时立刻生效, 下一帧被移除。
- **v-leave-active:** 定义离开过渡生效时的状态。在整个离开过渡的阶段中应用, 在离开过渡被触发时立刻生效, 在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间, 延迟和曲线函数。
- **v-leave-to:** 2.1.8版及以上 定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 v-leave 被删除), 在过渡/动画完成之后移除。

## 事件处理

可以用 v-on 指令监听 DOM 事件，并在触发时运行一些 JavaScript 代码。

举个栗子

```
<div id="example-2">
  <!-- `greet` 是在下面定义的方法名 -->
  <button v-on:click="greet">Greet</button>
</div>

var example2 = new Vue({
  el: '#example-2',
  data: {
    name: 'Vue.js'
  },
  // 在 `methods` 对象中定义方法
  methods: {
    greet: function (event) {
      ...
    }
  }
})
```

## 事件修饰符

- .stop
- .prevent
- .capture
- .self
- .once
- .passive

## 按键修饰符

- .enter
- .tab
- .delete (捕获“删除”和“退格”键)
- .esc
- .space
- .up
- .down
- .left
- .right

## 系统修饰符

- .ctrl
- .alt
- .shift
- .meta

## 鼠标按钮修饰符

- .left
- .right
- .middle

## 生命周期

所有的生命周期钩子自动绑定 this 上下文到实例中，因此你可以访问数据，对属性和方法进行运算。这意味着你不能使用箭头函数来定义一个生命周期方法（例如 created: () => this.fetchTodos()）。这是因为箭头函数绑定了父上下文，因此 this 与你期待的 Vue 实例不同，this.fetchTodos 的行为未定义。

- beforeCreate
- created: 在实例创建完成后被立即调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调
- beforeMount
- mounted: el 被新创建的 vm.\$el 替换，挂载到实例上去
- beforeUpdate
- updated: 数据更改导致的虚拟 DOM 重新渲染
- activated: keep-alive 组件激活时调用
- deactivated
- beforeDestroy
- destroyed: 实例销毁之后调用
- errorCaptured

## 组件

组件是可复用的 Vue 实例，且带有一个名字。data、computed、watch、methods 以及生命周期钩子等，组件也是有的。仅有的例外是没有 el 这样根实例特有的选项，并且data必须是个函数。

## 通过 Prop 向子组件传递数据

举个栗子

```
Vue.component('button-counter', {
  data: function () {
    props: ['title'],
    return {
      count: 0,
      name: title
    }
  },
  template: '<button v-on:click="count++">{{name}} clicked me {{ count }} times.</button>'
})
new Vue({ el: '#components-demo' })

<div id="components-demo">
  <button-counter title="zhaowa"></button-counter>
</div>
```

## 监听子组件事件

在我们开发子组件时，它的一些功能可能要求我们和父级组件进行沟通。

举个栗子

```
// 子组件
<button v-on:click="$emit('enlarge-text', 0.1)">
  Enlarge text
</button>

// 父组件中的使用
<blog-post
  v-on:enlarge-text="onEnlargeText"
></blog-post>

methods: {
  onEnlargeText: function (enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
}
```

## 组件的注册

全局注册是指，子组件在任何组件里都可以使用。局部注册是指，父组件只能使用自己内部配置过的子组件。

全局注册所有的组件意味着即便你已经不再使用一个组件了，它仍然会被包含在你最终的构建结果中。这造成了用户下载的 JavaScript 的无谓的增加。

### prop

- prop 命名建议使用(短横线分隔命名) 命名
- 通常你希望每个 prop 都有指定的值类型。这时，你可以以对象形式列出 prop，这些属性的名称和值分别是 prop 各自的名称和类型：

```
props: {
  title: String,
  likes: Number,
  isPublished: Boolean,
  commentIds: Array,
  author: Object,
  callback: Function,
  contactsPromise: Promise // or any other constructor
}
```

### 举个栗子

```
<blog-post v-bind:likes="42"></blog-post>
<blog-post v-bind:is-published="false"></blog-post>
<blog-post v-bind:comment-ids="[234, 266, 273]"></blog-post>
<blog-post v-bind="post"></blog-post> // 传入整个post对象
```

### 单向数据流

所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。

### prop 验证

可以为组件的 prop 指定验证要求，例如你知道的这些类型。如果有一个需求没有被满足，则 Vue 会在浏览器控制台中警告你。这在开发一个会被别人用到的组件时尤其有帮助。

### 举个栗子

```
Vue.component('my-component', {
  props: {
    // 基础的类型检查 (`null` 和 `undefined` 会通过任何类型验证)
```

```
propA: Number,  
// 多个可能的类型  
propB: [String, Number],  
// 必填的字符串  
propC: {  
  type: String,  
  required: true  
},  
// 带有默认值的数字  
propD: {  
  type: Number,  
  default: 100  
},  
// 带有默认值的对象  
propE: {  
  type: Object,  
  // 对象或数组默认值必须从一个工厂函数获取  
  default: function () {  
    return { message: 'hello' }  
  }  
},  
// 自定义验证函数  
propF: {  
  validator: function (value) {  
    // 这个值必须匹配下列字符串中的一个  
    return ['success', 'warning', 'danger'].indexOf(value) !== -1  
  }  
}  
})
```

## 自定义事件

始终使用 kebab-case 的事件名

## 插槽

## 动态组件 & 异步组件