1. 作用域、var、let、const

> 作用域指一个变量的作用的范围

(1.1) 在JS中一共有两种作用域【JS的ES6之前没有块级作用域】

- 全局作用域
  - 全局作用域中有一个全局对象window，可以直接使用
  - 在全局作用域中
    - 创建的变量都会作为window对象的属性保存
    - 创键的函数都会作为window对象的方法保存
- 函数作用域
  - 调用函数时创建函数作用域，函数执行完毕以后，函数作用域销毁
  - 每调用一次函数就会创建一个新的函数作用域，他们之间是互相独立的
  - 在函数作用域中可以访问到全局作用域的变量，在全局作用域中无法访问到函数作用域的变量
  - 当在函数作用域中操作一个变量时，它会现在自身作用域中寻找，如果有就直接使用，如果没有则向上一级作用域中寻找

```
var a = 10;

function fun() {
  var b = 20;

  console.log("a = " + a);
}

fun();                      // a = 10
console.log("b = " + b);    // Uncaught ReferenceError: b is not defined
```

(1.2) 变量声明提前

- **1. var会变量提升【下面这两个等价】| let不会变量提升**

```
console.log(a);    // undefined

var a = 5;
```

```
var a;

console.log(a);     // undefined
```

```
console.log(a);     // Uncaught ReferenceError: a is not defined

a = 5;
```

- **2. let 块级作用域**

```
{
  let b = 10;
}

console.log(b);     // Uncaught ReferenceError: b is not defined
```

- **3. 使用let，变量不能重复声明**

```
let x = 10;
let x = 20;         // 浏览器报错：Uncaught SyntaxError: Identifier 'x' has
already been declared
```

- **4. 暂时性死区：在当前作用域 不允许同名的变量进来**

```
let c = 10;
function d() {
  console.log(c);
}
d();                 // c = 10
```

```
let c = 10;
function d() {
  console.log(c);

  let c = 20;       // 因为let有暂时性死区特性，在函数内定义了let c，外面的全局c就进不来
了，然后又因为let没有变量提升特性，所以console.log()时候浏览器没有找到c，就报错了
}
d();                // 浏览器报错 Uncaught ReferenceError: c is not defined
```

若上面函数内的let c改成var c，则输出是undefined

```
let c = 10;
```

```
function d() {
  console.log(c);

  var c = 20;
}
d();
```

等价于：

```
let c = 10;
function d() {
  var c;                  // c = undefined，var的变量提升特性

  console.log(c);

  var c = 20;
}
d();
```

- **5. const具有上面let的全部特性**

```
const e = 10;

e = 20;

console.log(e);       // 报错，不能修改const的值
```

但是，下面这个例子怎么就能改了呢?

是因为arr里存的是指针(地址)，我们并没有该指针啊，我们改的是指针指向位置里的那个值，arr里指的位置我们没有改

```
const arr = [1,2,3];
arr[0] = 0;
console.log(arr);       // arr=[0,2,3]

arr = ['a'];
console.log(arr);       // 报错 Uncaught TypeErrror: Assignment to constant variable，因为arr=['a']新的对象地址，所以报错
```

(1.3) 函数声明提前

- 使用**函数声明**形式创建的函数 `function 函数名(){}` 它会在所有的代码执行之前就被创建【函数提升】
- **函数表达式**本质是把匿名函数赋值给变量，这里如果在上面打印fun2，会是undefined，说明var声明的fun2变量确实提升了。当代码执行到 `var fun2 = function() {...}` 这行时才会把函数赋个这个变量fun2【使用函数表达式创建的函数，不会被声明提前，所以不能在声明前调用】

```
fun();              // 输出：我是一个fun函数
fun2();             // 报错：Uncaught TypeError: undefined is not a function

function fun() {
  console.log('我是一个fun函数');
}

var fun2 = function() {
  console.log('我是fun2函数');
}
```

(1.4) var、let、const

var

- 可以重复声明
- 无法限制修改
- 没有块级作用域

let

- 不能重复声明
- 变量可以修改
- 有块级作用域

const

- 不能重复声明
- 变量不可以修改
- 有块级作用域

2. this

- 解析器在调用函数 每次都会向函数内部传递一个隐含的参数，就是this，this指向的是一个对象，这个对象我们称为函数执行的上下文对象
- 根据函数的调用方式的不同，this会指向不同的对象
  - 以函数形式调用时，this永远都是window
  - 以方法形式调用时，this是调用方法的对象
  - 以构造函数形式调用时，this是新创建的那个对象
  - 使用call和apply调用时，this是指定的那个对象

(1) this指向问题

- 函数中 => window

```
functioon a() {
  console.log(this);
}

a();    // window
```

- 定时器中 => window

```
setInterval(function() {
  console.log(this);
}, 1000)

setTimeout(function() {
  console.log(this);
}, 1000)
```

- 对象 => this指向对象

```
obj = {
  name: 'haha',
  say: function() {
    console.log(this);
  }
}

obj.say();
```

- 事件中 => this指向执行点击的元素
- 箭头函数 => this指向声明时候的this，也就是指向父作用域
- 在类中 => this指向实例化对象

(2) 如何改变this指向

- `call()`、`apply()`、`bind()`

```
functiona a(a, b) {
  console.log(this, a, b);
}

obj = {
  name: 'lasa';
}

a.call(obj, '1', '2');         // 第一个是this指向，后面是参数
a.apply(obj, ['1', '2']);      // 第一个是this指向，后面是参数
a.bind(obj, '1', '2')();       // 第一个是this指向，后面是参数
```

2.1 箭头函数

难点：this指向怎么找

箭头函数中this指向声明时的this，而不是执行时的this

=> 找父作用域中的this。例2中，setTimeout写成箭头函数，而箭头函数本身没有this，所以去它的父作用域中找，它的父作用域就是aLi[i].onclick=function(){...}里，这里的this就是指向被点击的dom

```
<ul>
  <li>1111</li>
  <li>2222</li>
  <li>3333</li>
</ul>

// 例1
<script>
  var aLi = document.getElementsByTagName('li');
```

```
  for(var i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      setTimeout = function() {
        console.log(this);            // windows，定时器中this指向windows
      }
    }
  }
</script>

// 如果我们还想this指向谁的点击事件
// 例2-1：箭头函数写法
<script>
  var aLi = document.getElementsByTagName('li');
  for(var i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      setTimeout(() => {
        console.log(this);          // 点击谁，显示谁的dom
      })
    }
  }
</script>

// 例2-2：bind写法
<script>
  var aLi = document.getElementsByTagName('li');
  for(var i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      setTimeout(function() {
        console.log(this);          // 点击谁，显示谁的dom
      }.bind(this))
    }
  }
</script>
```

例3

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

obj.say();            // this指向obj {name:'lisa', say: f}
```

改写

```
var obj = {
  name: 'lisa',
  say: function() {
    setTimeout(function() {
      console.log(this);
    })
  }
}

obj.say();              // this指向windows
```

继续改写：【setTimeout是箭头函数，所有向父作用域中找this，父作用域是say:function(){...}，这里面的this是指向obj的】

```
var obj = {
  name: 'lisa',
  say: function() {
    setTimeout(() => {
      console.log(this);
    })
  }
}

obj.say();          // this指向obj, {name:'lisa', say:f}
```

继续改写：【setTimeout箭头函数，向父作用域say:() => {}中找this，但是不巧父作用域也是箭头函数，say的this也要向父作用域找，就到了全局作用域，于是say里和setTimeout里的this都是指向windows。】

```
var obj = {
  name: 'lisa',
  say: () => {
    setTimeout(() => {
      console.log(this);
    })
  }
}

obj.say();          // this指向windows
```

3. 使用工厂方式创建对象

```
function createPerson(name, age, gender) {    // 构造函数，其实就是普通函数，只是它的功能比较特别，这个函数的功能是构造一个对象，所以叫构造函数
  var obj = new Object();
  obj.name = name;
  obj.age = age;
  obj.gender = gender;
  obj.sayName = function() {
    console.log(this.name);
  }

  return obj;
}

var obj = createPerson("猪八戒", 28, "男");
var obj1 = createPerson("唐僧", 38, "男");
```

使用工厂方法创建的对象，使用的构造函数都是Object，所以创建的对象都是Object这个类型，就导致我们无法区分出多种不同类型的对象

之前:

```
var person = new Object()
var dog = new Object()
// Object { name: '', age: '', gender: '' }
```

我们想改成:

```
var person = new Person()
var dog = new Dog()

// Object { name: '', age: '', gender: '' }
```

进化一：于是我们使用构造函数

- 创建一个构造函数，专门用来创建Person对象【构造函数就是一个普通的函数，创建方式和普通函数没有区别】【构造函数习惯首字母大写】
- **构造函数和普通函数的区别就是调用方式不同**
  - **普通函数是直接调用，而构造函数需要使用new关键字来调用**
- **构造函数的执行流程**
  - 1. 立即创建一个新的对象
  - 2. 将新建的对象设置为函数中this，在构造函数中可以使用this来引用新建的对象
  - 3. 逐行执行函数中的代码
  - 4. 将新建的对象作为返回值返回
- 使用同一个构造函数创建的对象，称为一类对象，也将一个构造函数称为一个类
- 将通过构造函数创建的对象，称为是该类的实例

```
function Person(name, age, gender) {
  this.name = name;         // this就是这个per
  this.age = age;
  this.gender = gender;
  this.sayName = function() {
    console.log(this.name);
  }
}

var per = new Person('孙悟空', 18, '男');
var per2 = new Person('唐僧', 38, '男');
// Person{ name: '孙悟空', age: 18, gender: '男' }
// Person{ name: '唐僧', age: 38, gender: '男' }

function Dog () {

}

var dog = new Dog();
// Dog { }

console.log(per instanceof Person)        // true
console.log(dog instanceof of Dog)          // false
```

进化二：把共用的方法添加在原型上而不是放在构造函数中

> 结论：以后创建构造函数时，可以将这些对象共有的属性和方法，统一添加到构造函数的原型对象中，这样不用分别为每一个对象添加，也不会影响到全局作用域，就可以使每个对象都具有这些属性和方法了

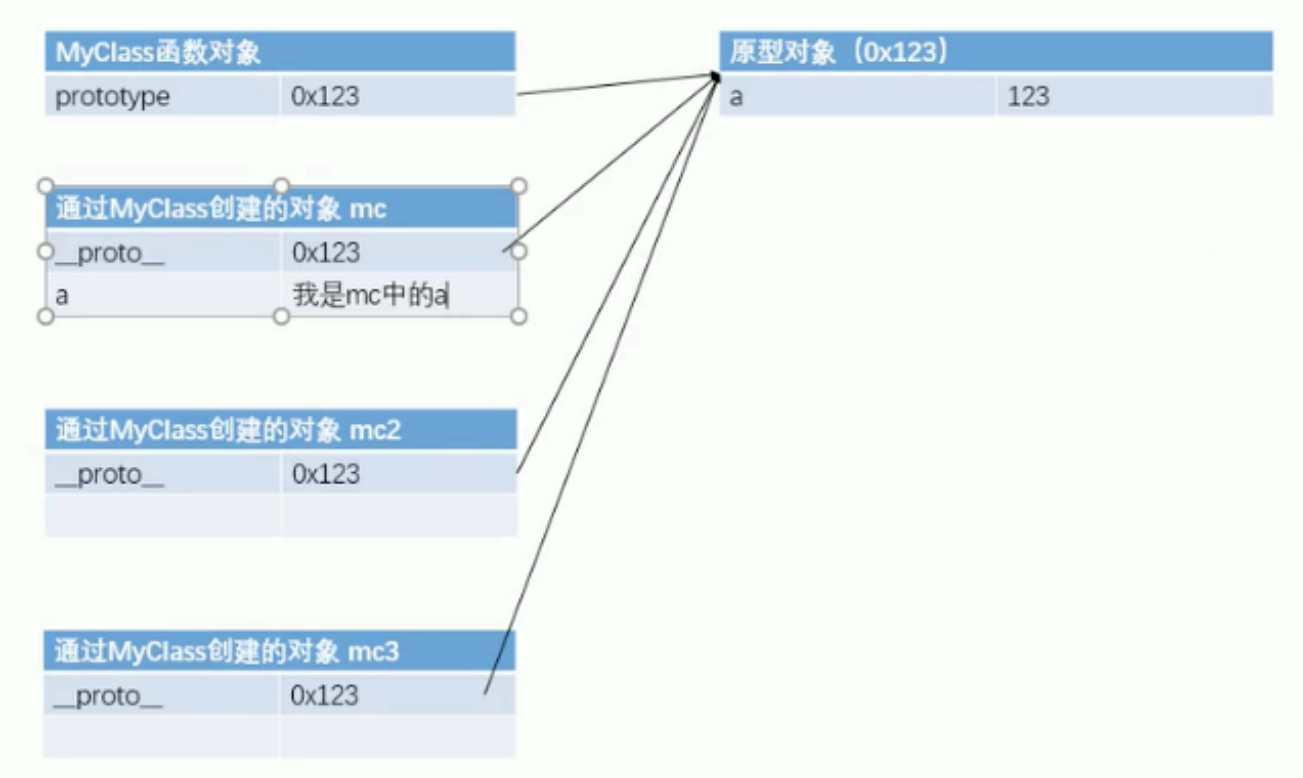
- 上面的Person构造函数中，我们发现我们为每一个对象都添加了一个sayName方法
  - 上面的写法把方法放在了构造函数中，也就是构造函数每执行一次就会创建一个新的sayName方法，这样就导致了构造函数执行一次就会创建一个新的方法，执行10000次就会创建10000个新的方法，因为这10000次创建的是同样的方法这就没有必要了，完全可以共享这个方法
- **把共用的方法放到原型对象中**

```
function Person(name, age, gender) {
  this.name = name;         // this就是这个per
  this.age = age;
  this.gender = gender;
  // this.sayName = function() {
    //console.log("Hello 大家好，我是：" + this.name);
  // }
}

Person.prototype.sayName = function() {
  console.log("Hello 大家好，我是：" + this.name);
}

var per = new Person('孙悟空', 18, '男');
var per2 = new Person('唐僧', 38, '男');

// console.log(per.sayName == per2.sayName);      // false
```

3.1 原型prototype【原型对象就相当于一个公共的区域】

> 构造函数也是函数，和普通函数区别在于，定义时开头字母大写，并且通过new关键字实例化对象

- 我们创建的每一个函数，解析器都会向**函数中**添加一个属性prototype；这个属性对应着一个对象，这个对象就是我们所谓的原型对象
- 如果函数作为普通函数调用，prototype没有任何作用
- 当函数以构造函数的形式调用时，它所创建的对象中都会有一个隐含的属性，指向该构造函数的原型对象，我们可以通过 `__proto__` 来访问该属性
- **原型对象就相当于一个公共的区域，所有同一个类的实例都可以访问到这个原型对象，我们可以将对象中共有的内容，统一设置到原型对象中**
- 当我们访问对象的一个属性或方法时，它会先在对象自身中寻找，如果有则直接使用，如果没有则去原型对象中寻找，如果找到则直接使用




```
function MyClass() {            // 构造函数

}

// 向MyClass的原型中添加属性a
MyClass.prototype.a = 123;

// 向MyClass的原型中添加一个方法
MyClass.prototype.sayHello = function() {
  alert('Hello');
}

var mc = new MyClass();
var mc2 = new MyClass();

// console.log(mc.__proto__ == MyClass.prototype);      // true
// console.log(mc2.__proto__ == MyClass.prototype);      // true

// 向mc中添加a属性
mc.a = '我是mc中的a';
console.log(mc.a);                  // 我是mc中的a

mc.sayHello();                      // Hello

// console.log(mc.a);               // 前提是没有在mc中添加a属性  123
```

3.2 再谈原型

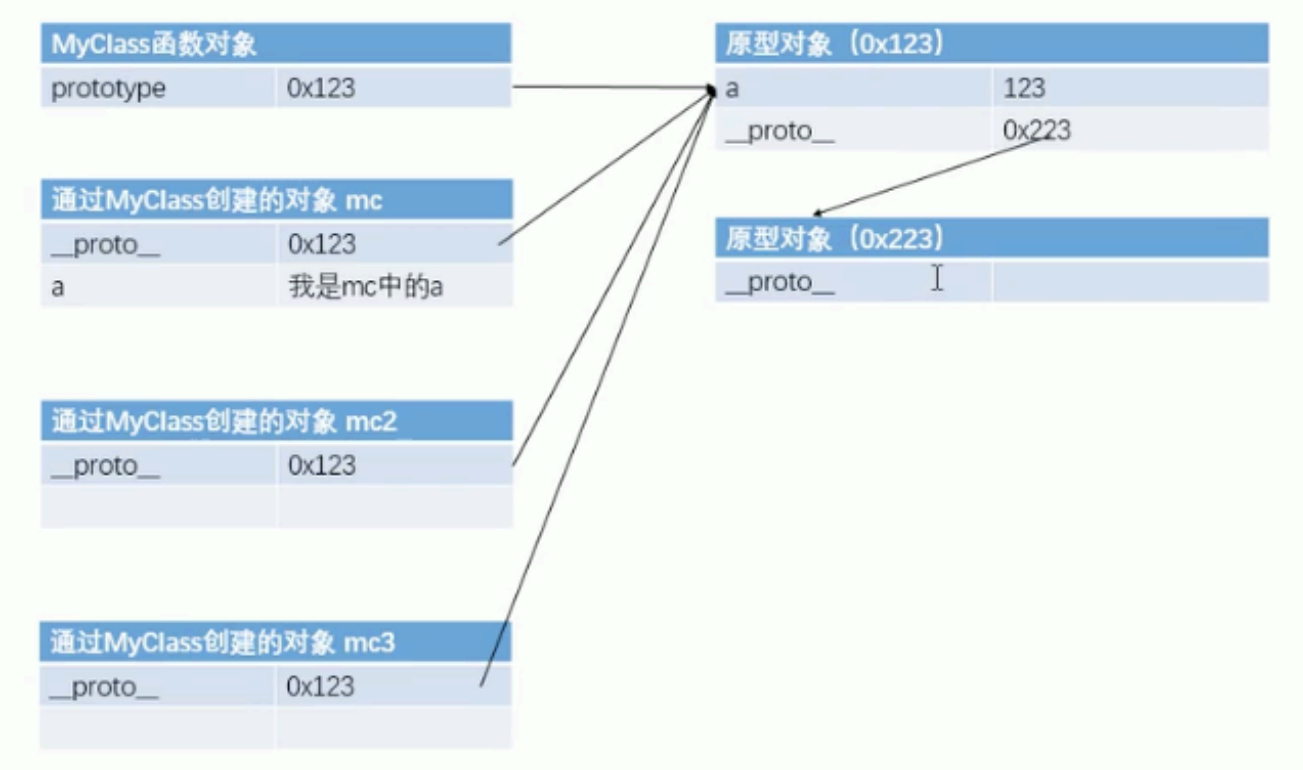
```
function MyClass() {

}

MyClass.prototype.name = '我是原型中的名字';

var mc = new MyClass();
mc.age = 18;

// 使用in检查对象中是否含有某个属性时，如果对象中没有但是原型中有，也会返回true
console.log("name" in mc);          // true

// 可以使用对象的hasOwnProperty()来检查对象自身是否含有该属性
// 使用该方法只有当对象自身中含有属性时，才会返回true
console.log(mc.hasOwnProperty('age'));      // true
```

我们好奇 hasOwnProperty这个API哪来的

- 发现也不mc的原型上，那么哪来的呢?

```
console.log(mc.__proto__.hasOwnProperty("hasOwnProperty"));      // false
```

我们发现原型对象也是对象，所以它也有原型



- 当我们使用一个对象的属性或方法时，会先在自身中寻找，自身中如果有，则直接使用；如果没有则去原型对象中寻找，如果原型对象中有，则使用；如果没有则去原型的原型中寻找，直到找到Object对象的原型，Object对象的原型没有原型，如果在Object中依然没有找到，则返回undefined

```
console.log(mc.__proto__.__proto__)    // 原型对象的原型
console.log(mc.__proto__.__proto__.hasOwnProperty("hasOwnProperty"))    // true
```

3.3 Blue讲面向对象、原型、继承

(1) 工厂方式的问题：

- 没有new
- 方法重复，资源浪费。如果有100个对象，那么就要生成100个sayName，但其实sayName是一样的，共享这个方法就完事了

解决方案：

总结：用构造函数加属性，用原型加方法

- 解决new问题

原本写法

```
function createPerson(name, qq) {       // 工厂方式
  var obj = new Object();
  obj.name = name;
  obj.qq = qq;

  obj.showName = function() {
    alert('我的名字叫: ' + this.name);
  }

  obj.showQQ = function() {
    alert('我的QQ号: ' + this.qq);
  }

  return obj;
}
```

更改后写法

```
function createPerson(name, qq) {
  // var obj = new Object();

  // 通过new创建实例时
  // 系统会偷偷替我们做:
  // var this = new object();

  this.name = name;
  this.qq = qq;

  this.showName = function() {
    alert('我的名字叫: ' + this.name);
  }

  this.showQQ = function() {
    alert('我的QQ号: ' + this.qq);
  }

  // 也会偷偷做:
  // return this;
}
```

- 解决方法重复问题

```
createPerson.prototype.showName = function() {
  alert('我的名字叫：' + this.name);
}

createPerson.prototype.showQQ = function() {
  alert('我的名字叫：' + this.qq);
}
```

(2) 原型

ss JS

Class 一次给一组元素加样式 原型

行间样式 一次给一个元素加样式 给对象加东西

4. ES5面向对象、继承 VS ES6面向对象、继承 写法

(4.1) ES5面向对象【构造函数 + 原型】

```
function User(name, pass) {
  this.name = name;
  this.pass = pass;
}

User.prototype.showName = function() {
  alert(this.name);
}

User.prototype.showPass = function() {
  alert(this.pass);
}

var u1 = new User('blue', '123456');
u1.showName();
u1.showPass();
```

(4.2) ES6面向对象【class + constructor】

```
class User {
  constructor(name, pass) {
    this.name = name;
    this.pass = pass;
  }

  showName() {
    alert(this.name);
  }

  // 方法直接会加到User原型上
  showPass() {
    alert(this.name);
  }
}

var u1 = new User('blue', '123456');

u1.showName();
u1.showPass();
```

(4.3) ES5 继承

```
function VipUser(name, pass, level) {
  User.call(this, name, pass);
  this.level = level;
}

VipUser.prototype = new User();     // 等价VipUser.prototype = User.prototype;
VipUser.prototype.constructor = VipUser;

VipUser.prototype.showLevel = function() {
  alert(this.level);
}

var v1 = new VipUser('blue', '123456', 3);
v1.showName();
v1.showPass();
v1.showLevel();
```

(4.4) ES6 继承

```
class VipUser extends User {
  constructor(name, pass, level) {
    super(name, pass);
    this.level = level;
  }

  showLevel() {
    alert(this.level);
  }
}

var v1 = new VipUser('blue', '123456', 3);
v1.showName();
v1.showPass();
v1.showLevel();
```

4.【并列4】 ES5的继承【用的最多：原型链 + 对象冒充的组合继承模式】

- 继承原理：原型链 + 对象冒充的组合继承模式
- 在子类里实例化一个父类：**冒充是Person类的实例，我们知道实例拥有这个类构造函数里的所有属性和方法**
    - **注意：对象冒充可以继承构造函数里面的属性和方法，但是没法继承原型链上的属性和方法**

(1) 对象冒充实现继承：子类只能继承父类构造函数里的属性和方法；问题： 【没法继承父类原型链上的属性和方法】

```
// Person类
function Person() {
  this.name = '张三';                // 构造函数中定义属性
  this.age = 20;
```

```
  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';      // 原型链上扩展属性
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}


// Web类 继承Person类：原型链 + 对象冒充的组合继承模式
function Web() {
  Person.call(this);       // 对象冒充实现继承【冒充是Person类的实例，我们知道实例拥有这个类的所有属性和方法】
}

var w = new Web();
w.run();  正确      // 对象冒充可以继承构造函数里面的属性和方法
w.work(); 错误     // 对象冒充可以继承构造函数里面的属性和方法 但是没法继承原型链上的属性和方法
```

(2) 原型链实现继承：既可以继承父类构造函数里的属性和方法，也可以继承原型链上的属性和方法

```
// Person类
function Person() {
  this.name = '张三';                // 构造函数中定义属性
  this.age = 20;

  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';      // 原型链上扩展属性
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}


// Web类 继承Person类：原型链 + 对象冒充的组合继承模式
function Web() {
```

```
}
// 父类的实例挂载到子类的原型上，这样子类就拥有了父类的构造函数和原型链上的所有属性和方法
Web.prototype = new Person();    // 原型链实现继承 可以继承构造函数里的属性和方法，也可以继承原型链上的属性和方法

var w = new Web();
w.run();            // 正确
```

(3) 原型链实现继承的问题【实例化子类的时候没法给父类传参】

```
// Person类
function Person(name, age) {
  this.name = name;                // 构造函数中定义属性
  this.age = age;

  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';      // 原型链上扩展属性
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}

var p = new Person('李四', 20);

p.run();         // 李四在运动


// 子类
function Web(name, age) {

}

Web.prototype = new Person();

var w = new Web('赵四', 20);       // 实例化子类的时候没法给父类传参
w.run();         // undefined在运动
```

(4) 原型链 + 对象冒充的组合继承模式

```
// Person类
function Person(name, age) {
  this.name = name;                // 构造函数中定义属性
  this.age = age;

  this.run = function() {
    alert(this.name + '在运动');    // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';     // 原型链上扩展属性
Person.prototype.work = function() {     // 原型链上扩展方法
  console.log(this.name + '在工作');
}

var p = new Person('李四', 20);

p.run();         // 李四在运动


// 子类
function Web(name, age) {
  Person.call(this, name, age);   // 对象冒充 => 可以继承构造函数里面的属性和方法 => 实例化子类时可以给父类传参
}

Web.prototype = new Person();     // 原型链继承

var w = new Web('赵四', 20);
w.run();         // 赵四在运动
w.work();        // 赵四在工作
```

(5) 原型链 + 对象冒充组合继承的另一种方式

- 把`Web.prototype = new Person()`改成`Web.prototype = Person.prototype`
- 因为`Person.call(this, name, age)`对象冒充已经把父类构造函数里的属性和方法继承过来了，只需要再继承父类的原型链就行了

```
// Person类
function Person(name, age) {
  this.name = name;                // 构造函数中定义属性
  this.age = age;
```

```
  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';      // 原型链上扩展属性
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}

var p = new Person('李四', 20);

p.run();         // 李四在运动


// 子类
function Web(name, age) {
  Person.call(this, name, age);   // 对象冒充 => 可以继承构造函数里面的属性和方法 => 实例化子类时可以给父类传参
}

Web.prototype = Person.prototype;      // 原型链继承

var w = new Web('赵四', 20);
w.run();         // 赵四在运动
w.work();        // 赵四在工作
```

5. 闭包

定义：闭包是指外部函数里声明内部函数，内部函数引用外部函数里的局部变量，这样当外部函数调用完毕以后，局部变量不被释放，可以一直使用

- 浏览器刷新完for循环就执行完了此时i=4，这个时候我们才去点击任意一个dom，才执行console.log()这个命令，此时的 i = 4，所以输出4 【原本写法】
- 浏览器刷新完for循环就执行了，每个for循环一进去就立马执行**立即执行函数**，把当前的 `i` 传进去，然后aLi[i] 再绑定onclick，里面的console现在并不会执行，只是绑定了事件
    - **另种理解方式【从闭包定义角度】：`(function(idx) {...})` 这个是外部函数，内部函数 `aLi[idx].onclick=function(){...}` 中引用了外部函数中的局部变量 `idx`，这样外部函数执行完后，它的局部变量 `idx` 不会被释放，一直存在内存中，这样内部函数就可以拿到这个局**

部变量。外部函数中的四个局部变量idx=0, 1, 2, 3都被保存下来了，没有释放。那为什外部函数的局部变量能保存下来呢？这不就是利用了函数有块级作用域嘛！！所以引出另种 es6 let写法，es6的let使得这个for表达式就有块级作用域了！所以当我们点击任意一个dom时，`console.log(i)` 能拿到哪个i就会输出哪个i

- 在原本写法中，当我们点击那一刻，才执行 `console.log()`，但是此时只能拿到外部的 `i=4`；在闭包写法中，因为内部函数要使用外部函数的 `idx` 局部变量，所以外部函数执行完后不会释放局部变量 `idx`，这样当我们触发点击动作的时候，`console.log(idx)` 能拿到外部函数的局部变量 `idx`，也就是每个dom的index，所以能正确输出

```
<ul id="ul1">
  <li>11</li>
  <li>22</li>
  <li>33</li>
  <li>44</li>
</ul>

// 原本写法
<script>
  var oUl = document.getElementById('ul1');
  var aLi = oUl.getElementsByTagName('li');

  for(var i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      console.log(i);
    }
  }
</script>

// 闭包写法
<script>
  var oUl = document.getElementById('ul1');
  var aLi = oUl.getElementsByTagName('li');

  for(var i = 0; i < aLi.length; i++) {
    (function(idx) {
      aLi[idx].onclick = function() {
        console.log(idx);
      }
    })(i)
  }
</script>

// es6 let写法 [因为let有块级作用域，每个i=0,1,2,3都保存在for内部]
<script>
  var oUl = document.getElementById('ul1');
```

```
  oUl.getElementsByTagName('li');

  for(let i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      console.log(i);
    }
  }
</script>
```

6. ES6 / TS中类的 定义、继承、修饰符、静态属性、静态方法、继承多态、抽象类、

(1) TS中定义类

- `constructor(){}` 其实就是一个**钩子**，在实例化类的时候触发的方法

```
class Person {
  name: string;    // 属性 前面省略了public关键词
  constructor(n: string) {  // 构造函数 实例化类的时候触发的方法
    this.name = n;
  }

  run(): void {
    alert(this.name);
  }
}

var p = new Person('张三');

p.run();
```

```
class Person {
  name: string;    // 属性 前面省略了public关键词
  constructor(name: string) {  // 构造函数 实例化类的时候触发的方法
    this.name = name;
  }

  getName(): string {
    return this.name;
```

```
  }

  setName(name: string): void {
    this.name = name;
  }
}

var p = new Person('张三');
console.log(p.getName());      // 张三

p.setName('李四');
console.log(p.getName());      // 李四
```

(2) TS中如何实现继承：extends / super

```
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

var p = new Person('王五');
console.log(p.run());          // 王五在运动


// 子类
class Web extends Person {
  constructor(name: string) {     // 子类里不写constructor的话，默认使用父类的
consstructor
    super(name);     // 初始化父类的构造函数
  }
}

var w = new Web('李四');
console.log(w.run());         // 李四在运动
```

(3) TS中继承的探讨 父类的方法和子类的方法一致

```
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

var p = new Person('王五');
console.log(p.run());          // 王五在运动


// 子类
class Web extends Person {
  constructor(name: string) {     // 子类里不写constructor的话，默认使用父类的
consstructor
    super(name);      // 初始化父类的构造函数
  }

  run(): string {
    return `${this.name}在运动 - 子类`;
  }

  work() {
    console.log(`${this.name}在工作`);
  }
}

var w = new Web('李四');
w.work();           // 李四在工作
w.run();            // 李四在运动馆 - 子类
```

(4) 类里面的修饰符【TS里面定义属性的时候给我们提供了三种修饰符】

> public, protected, private

- **属性如果不加修饰符 默认就是public**
- public：公有 => 在当前类里面、子类、类外面都可以访问
- protected：保护类型 => 在当前类里面、子类里面可以访问；在类外部没法访问
- private：私有 => 在当前类里面可以访问，子类和类外部都没法访问

```
class Person {
  public name: string;            /* 公有属性 */

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

var p = new Person('王五');
console.log(p.run());           // 王五在运动


// 子类
class Web extends Person {
  constructor(name: string) {     // 子类里不写constructor的话，默认使用父类的
constructor
    super(name);      // 初始化父类的构造函数
  }

  run(): string {
    return `${this.name}在运动 - 子类`;
  }

  work() {
    console.log(`${this.name}在工作`);
  }
}

var w = new Web('李四');
w.work();                 // 李四在工作
```

- **public 类外部访问公有属性**

```
class Person {
  public name: string;            /* 公有属性 */

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

// 类外部访问公有属性
var p = new Person('哈哈哈');
console.log(p.name);           // 哈哈哈
```

- **protected 类外部没法访问到保护类型的属性【子类里可以访问】 ts编译会报错提示**

```
class Person {
  protected name: string;            /* 保护类型 */

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

var p = new Person('哈哈哈');
console.log(p.name);            // ts编译报错
```

- **private 只能在当前类使用【子类里不能访问】**

```
class Person {
  private name: string;            /* 私有属性 */

  constructor(name: string) {
    this.name = name;
  }
```

```
  run(): string {
    return `${this.name}在运动`;
  }
}

class Web extends Person {
  constructor(name: string) {
    super(name);
  }

  work() {
    console.log(`${this.name}在工作`);      // ts编译报错，私有属性只能在定义的类里使用
  }
}
```

(5) 静态属性、静态方法

ES5 中的静态属性、静态方法写法

```
function Person() {
  this.name1 = 'hehe';
  this.run1 = function() {          /* 实例方法 */

  }
}

Person.name2 = "haha";
Person.run2 = function() {          /* 静态方法 */

}

// 两种方法的调用方法不同
var p = new Person();
p.name1;                        // 实例属性的调用方法
p.run1();                       // 实例方法的调用方法

Person.name2;                   // 静态属性的调用方法
Person.run2();                  // 静态方法的调用方法
```

ES6/TS 中的静态属性、静态方法写法

- 静态方法里面没法直接调用类里面的属性；若想调用的话，把类里的属性写成 `static age` 这样
- 因为静态方法随着类的加载而加载，对象是在类存在后才能创建的，所以静态方法优先于对象存在，所以不能在静态方法里面访问成员变量

```
class Person {
  public name: string;
  static age: number = 20;
  constructor(name) {
    this.name = name;
  }

  run() {                          // 实例方法
    console.log(`${this.name}在运动`);
  }

  work() {                         // 实例方法
    console.log(`${this.work}在工作`);
  }

  static print() {                 // 静态方法，里面没法直接调用类里面的属性
    console.log('print方法' + Person.age);
  }
}
```

(6) 多态

> 父类定义一个方法不去实现，让继承它的子类去实现，每一个子类有不同的表现

多态属于继承

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }

  eat() {              // 具体吃什么 不知道，具体吃什么 让继承它的子类去实现 每个子类的表现不一样
    console.log('吃的方法');
  }
```

```
}

class Dog extends Animal {
  constructor(name: string){
    super(name);
  }

  eat() {
    return this.name + '吃肉';
  }
}

class Cat extends Animal {
  constructor(name: string) {
    super(name);
  }

  eat() {
    return this.name + '吃老鼠'
  }
}
```

(7) 抽象方法

> TS中的抽象类：它是提供其他类继承的基类，不能直接被实例化

> 用abstract关键字定义抽象类和抽象方法，抽象类中的抽象方法不包含具体实现并且必须在派生类中实现

> abstract抽象方法只能放在抽象类里面

抽象类和抽象方法用来定义标准；eg: Animal这个类要求它的子类必须包含eat方法

```
abstract class Animal {
  public name: string;
  constructor(name: string) {
    this.name = name;
  }

  abstract eat():any;             // 抽象类中的抽象方法，不包含具体实现且必须在派生类中实现

  run() {
    console.log('其他方法可以不实现');
  }
```

```
}

class Dog extends Animal {
  // 抽象类的子类必须实现抽象类里的抽象方法

  constructor(name: any) {
    super(name);
  }

  eat() {
    console.log(this.name + '吃粮食');
  }
}

var d = new Dog('小狗');
d.eat();                        // 小狗吃粮食
```

7. JS中的数据类型

- 基本数据类型：`number`、`string`、`boolean`、`null`、`undefined`、`symbol`
  - **null**：空对象
    - `document.getElementById('div1')` 取不到的时候会返回null空对象
  - **undefined**：
    - `var a; console.log(a);` 变量定义了没有赋值
    - `var arr = [1,2,3]; arr[5]` 数组越界也会返回undefined
    - `var obj = { name: 'haha' }; obj.age` 访问对象没有的属性
    - `function a() { console.log(111); } console.log(a)` 函数默认返回undefined
  - **typeof**
    - `typeof 'abc'` "string"
    - `typeof 123` "number"
    - `typeof true` "boolean"
    - `typeof null` "object"
    - `typeof undefined` "undefined"
    - `typeof 函数` "function"
    - `typeof [1,2,3]` "object"
    - `typeof {name: 'haha'}` "object"
- 引用数据类型：`array`、`object`

typeof 判断不了数组和对象，那么如何判断呢?

```
var arr = [1,2,3];
var obj = {
  name: "haha"
}

法一:
console.log(arr.constructor == Array)    // .constructor返回实例的构造函数
console.log(arr.constructor == Object)    // .constructor返回实例的构造函数

法二:
console.log(arr instanceof Array)      // 判断arr是否是Array的实例化对象

法三:
Array.isArray(arr)                     // Array自带的判断方法
```

(7.1) 对象、类

- 对象

```
var obj = {
  name: 'haha',
  say: function() {
    ...
  }
}
```

- 类：具有相同属性和方法的对象的集合

  这里用原生js写法，不用es6

```
function Person(name, age) {       // 构造函数，也叫类
  this.name = 'haha';
  this.age = 20;
  this.eat = function() {
    console.log('...is eating');
  }
}
```

```
var person1 = new Person('lisa', 30);     // 实例化对象
console.log(person1);
// {'lisa', 30}

var person2 = new Person();
console.log(person2);
// {'haha', 20}
```

- 但是不建议把函数写在里面，若按照上面写法，每次new实例，在eat函数位置都会执行new Function
- 应该把函数放在prototype原型对象里 - **定义在原型对象下的所有属性和方法能被所有实例化对象共享**

```
function Person(name, age) {        // 构造函数，也叫类
  this.name = 'haha';
  this.age = 20;
}

Person.prototype.eat = function() {
  console.log(this.name + '...is eatting');
}
```

总结：属性写在构造函数了，方法写在原型里

8. ES6新特性

常见面试题

(8.1). 说说es6 新特性

let、const、promise、箭头函数、解构、class、set、map、proxy、数组、对象新方法、模板字符串

(8.2) 函数声明提前

- 使用**函数声明**形式创建的函数 `function 函数名(){}` 它会在所有的代码执行之前就被创键【函数提升】
- **函数表达式**本质是把匿名函数赋值给变量，这里如果在上面打印fun2，会是undefined，说明var声明的fun2变量确实提升了。当代码执行到 `var fun2 = function() {...}` 这行时才会把函数赋个这

个变量fun2【使用函数表达式创建的函数，不会被声明提前，所以不能在声明前调用】

```
fun();              // 输出：我是一个fun函数
fun2();             // 报错：Uncaught TypeError: undefined is not a function

function fun() {
  console.log('我是一个fun函数');
}

var fun2 = function() {
  console.log('我是fun2函数');
}
```

(8.3). 说说let 和 var 区别，const

- **1. var会变量提升【下面这两个等价】| let不会变量提升**

```
console.log(a);    // undefined

var a = 5;
```

```
var a;

console.log(a);    // undefined
```

```
console.log(a);    // Uncaught ReferenceError: a is not defined

a = 5;
```

- **2. let 块级作用域**

```
{
  let b = 10;
}

console.log(b);    // Uncaught ReferenceError: b is not defined
```

- **3. 使用let，变量不能重复声明**

```
let x = 10;
let x = 20;         // 浏览器报错：Uncaught SyntaxError: Identifier 'x' has
already been declared
```

- **4. 暂时性死区：在当前作用域 不允许同名的变量进来**

```
let c = 10;
function d() {
  console.log(c);
}
d();                 // c = 10
```

```
let c = 10;
function d() {
  console.log(c);

  let c = 20;       // 因为let有暂时性死区特性，在函数内定义了let c，外面的全局c就进不来
了，然后又因为let没有变量提升特性，所以console.log()时候浏览器没有找到c，就报错了
}
d();                // 浏览器报错 Uncaught ReferenceError: c is not defined
```

若上面函数内的let c改成var c，则输出是undefined

```
let c = 10;
function d() {
  console.log(c);

  var c = 20;
}
d();
```

等价于：

```
let c = 10;
function d() {
  var c;                    // c = undefined，var的变量提升特性

  console.log(c);

  var c = 20;
}
d();
```

- **5. const具有上面let的全部特性**

```
const e = 10;

e = 20;

console.log(e);       // 报错，不能修改const的值
```

但是，下面这个例子怎么就能改了呢?

是因为arr里存的是指针(地址)，我们并没有该指针啊，我们改的是指针指向位置里的那个值，arr里指的位置我们没有改

```
const arr = [1,2,3];
arr[0] = 0;
console.log(arr);       // arr=[0,2,3]

arr = ['a'];
console.log(arr);       // 报错 Uncaught TypeErrror: Assignment to constant variable，因为arr=['a']新的对象地址，所以报错
```

(8.4). 说说解构

```
[a, b] = [1, 2];
console.log(a);

const {a, b} = {a:1, b:2};
console.log(a, b);
```

(8.5). 说说set：自动去重

类似python里的集合set

```
// 数组去重
let arr = [1,2,3,2];
let arr2 = new Set(arr);
let arr3 = [...arr2];
console.log(arr3);       // [1,2,3]
```

(8.6). 箭头函数

难点：this指向怎么找

箭头函数中this指向声明时的this，而不是执行时的this

=> 找父作用域中的this。例2中，setTimeout写成箭头函数，而箭头函数本身没有this，所以去它的父作用域中找，它的父作用域就是aLi[i].onclick=function(){...}里，这里的this就是指向被点击的dom

```
<ul>
  <li>1111</li>
  <li>2222</li>
  <li>3333</li>
</ul>

// 例1
<script>
  var aLi = document.getElementsByTagName('li');
  for(var i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      setTimeout = function() {
        console.log(this);            // windows，定时器中this指向windows
      }
    }
  }
</script>

// 如果我们还想this指向谁的点击事件
// 例2-1：箭头函数写法
<script>
  var aLi = document.getElementsByTagName('li');
  for(var i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      setTimeout(() => {
        console.log(this);          // 点击谁，显示谁的dom
      })
```

```
    }
  }
</script>

// 例2-2：bind写法
<script>
  var aLi = document.getElementsByTagName('li');
  for(var i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      setTimeout(function() {
        console.log(this);          // 点击谁，显示谁的dom
      }.bind(this))
    }
  }
</script>
```

例3

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

obj.say();            // this指向obj {name:'lisa', say: f}
```

改写

```
var obj = {
  name: 'lisa',
  say: function() {
    setTimeout(function() {
      console.log(this);
    })
  }
}

obj.say();            // this指向windows
```

继续改写：【setTimeout是箭头函数，所有向父作用域中找this，父作用域是say:function(){...}，这里面的this是指向obj的】

```
var obj = {
  name: 'lisa',
  say: function() {
    setTimeout(() => {
      console.log(this);
    })
  }
}

obj.say();          // this指向obj, {name:'lisa', say:f}
```

继续改写：【setTimeout箭头函数，向父作用域say:() => {}中找this，但是不巧父作用域也是箭头函数，say的this也要向父作用域找，就到了全局作用域，于是say里和setTimeout里的this都是指向windows。】

```
var obj = {
  name: 'lisa',
  say: () => {
    setTimeout(() => {
      console.log(this);
    })
  }
}

obj.say();          // this指向windows
```

(8.7). class

- 声明一个类

```
class Person {
  constructor(name) {
    this.name = name;
  }
  say() {
    console.log(this.name + 'say...');
  }

  // 静态方法，子类继承不到
  static sleep() {
    console.log(this.name + 'sleep...');
```

```
  }
}

var person1 = new Person('lisa');
person1.say();
```

- 继承

```
class Coder extends Person {
  constructor(name, age) {
    super(name);       // 继承父类Person的name属性
    this.age = age;    // Coder类自己的age属性
  }

  // 自己的方法
  eat() {
    console.log(this.name + ' is eating...');
  }
}

var coder = new Coder('lisi', 20);
console.log(coder);      // Coder {name: "lisi", age: 20}

coder.say();         // lisisay...
coder.eating();      // lisi is eating
```

(8.8). promise：将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数

一个Promise配一个resolve和一个reject一个then一个catch

- 一个promise可能有三种状态：等待（pending）、已完成（resolved）、已拒绝（rejected）
- 一个promise的状态只可能从“等待”转到“完成”态或者“拒绝”态，不能逆向转换，同时”完成“态和”拒绝“态不能相互转换
- promise必须实现then方法，而且then必须返回一个promise，同一个promise的then可以调用多次，并且回调的执行顺序跟它们被定义时的顺序一致
- then方法接收两个参数，第一个参数是成功时的回调，在promise由”等待“态转到”完成“态时调用，另一个是失败时的回调，在promise由”等待“态转到”拒绝“态时调用。同时，then可以接受另一个promise传入，也接受一个”类then“的对象或方法，即thenable对象
- 解决函数的回调嵌套问题

- **不使用promise时候，回调地狱**

```
ajax(functionn(res) {
  if (res.a) {
    ajax(function(res) {
      if(res.id) {
        ajax(function() {
          ...
        })
      }
    })
  }
})
```

- **使用promise避免回调地狱**

```
let p1 = new Promse(function(resolve, reject) {
  setTimeout(function() {
    if(res.code === 1) {
      resolve();                // resolve代表成功 => 执行then里的代码
    } else {
      reject();                 // reject代表失败 => 执行catch里的代码
    }
  }, 1000)
}).then(function() {
  console.log('成功! ');
}).catch(function() {
  console.log('失败');
})
```

- **复杂点的promise调用**

```
let p1 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    let res = { code: 1 };
    if(res.code == 1) {
      resolve();              // 1. 成功走到这，跳转到第一个then
    } else {
      reject();
    }
  }, 1000)
}).then(function() {          // 2. 走到这resolve，跳转到第二个then
  return new Promise(function(resolve, reject) {
```

```
    setTimeout(function() {
      resolve();
    }, 500)
  })
}).then(function() {
  console.log('成功');       // 3. 走到这，输出成功
}).catch(function() {
  console.log('失败');
})
```

- **promise.all([])**

```
let P1 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    console.log(111);
    resolve();
  }, 500)
})
let P2 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    console.log(222);
    resolve();
  }, 800)
})
let P3 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    console.log(333);
    resolve();
  }, 300)
})

Promise.all([P1, P2, P3]).then(function() {
  console.log('over ok');
}).catch(function() {
  console.log('over no');
})

// 浏览器输出
333
111
222
over--
```

- **promise.race()**

```
Promise.race([P1, P2, P3]).then(function() {
  console.log('over ok');
}).catch(function() {
  console.log('over no');
})
```

9. Call、apply、arguments、forEach

(9.1). Call、apply

- 这两个方法都是函数对象的方法，需要通过函数对象来调用
- 当对函数调用call() 和apply() 都会调用函数执行
- 在调用call()和apply() 可以将一个对象指定为第一个参数
  - 此时这个对象将会成为函数执行时的this
- call() 方法可以将实参在对象之后依次传递
- apply() 方法需要将实参封装到一个数组中统一传递
- this的指向：
  - 以函数形式调用时，this永远都是window
  - 以方法形式调用时，this是调用方法的对象
  - 以构造函数形式调用时，this是新创建的那个对象
  - 使用call和apply调用时，this是指定的那个对象

```
function fun() {
  alert(this.name);
}

var obj = {
  name: 'obj',
  sayName: function() {
    alert(this.name);
  }
}

fun.call(obj, 2, 3);
fun.apply(obj, [2, 3]);
```

(9.2). arguments

在调用函数时，浏览器每次都会传递进两个隐含的参数：

- 函数的上下文对象this
- 封装实参的对象arguments
  - arguments是一个类数组对象，它也可以通过索引来操作数据，也可以获取长度
  - 在调用函数时，我们所传递的实参都会在arguments中保存
  - `arguments.length` 可以用来获取实参的长度
  - `arguments.callee` 这个属性对应一个函数对象，就是当前正在指向的函数的对象

```
function fun() {
  // console.log(Array.isArray(arguments));       // false
  console.log(arguments.length);
  console.log(arguments[0]);        // hello
  console.log(arguments.callee == fun);    // true
}

fun("hello", true);      // "2"
```

(9.3). forEach 遍历数组的一个API

> forEach() 方法需要一个函数作为参数
>
> 像这种函数，由我们创建但是不由我们调用的，称为回调函数

- 浏览器会在回调函数中传递三个参数
  - 第一个参数value，当前正在遍历的元素
  - 第二个参数index，当前正在遍历的元素的索引
  - 第三个参数obj，就是正在遍历的数组 `console.log(obj == arr) // true`

```
var arr = ["孙悟空", "猪八戒", "沙和尚", "唐僧", "白骨精"];

arr.forEach(function(value, index, obj) {
  console.log('hello');
})
```

10. 事件

(10.1). 事件模型：冒泡、捕获

冒泡：点击子元素，从子元素向父元素冒泡事件

捕获：点击子元素，从父元素向里子元素捕获事件

```
oDiv1.addEventListener('click', function(e) {
  console.log('div1冒泡');
}, false);

oDiv1.addEventListener('click', function(e) {
  console.log('div1捕获');
}, true);
```

(10.2). 如何绑定事件

```
var oDiv1 = document.getElementById('div1');

oDiv1.onclick = function() {
  console.log(111);
}

oDiv1.onclick = function() {
  console.log(222);
}
```

运行发现，如果使用.onclick方式绑定事件，在同个dom上后绑定的事件会覆盖前面的同个事件，所以使用addEventListenter()能在dom上绑定多个（同样的）事件：第一个参数是事件，第二个参数是事件处理函数，第三个参数是捕获或者冒泡（false冒泡，true捕获）

```
oDiv1.addEventListener('click', functionn() {
  console.log('111');
}, false);
oDiv1.addEventListener('click', functionn() {
  console.log('2');
}, false);
```

- `e.stopPropagation()` //阻止事件冒泡
- `e.cancelBubble = true` // IE阻止事件冒泡
- `e.preventDefault()` // 阻止默认行为
- `e.returnValue = false` // IE阻止默认行为

(10.3). 事件委托 【把事件绑定在父元素上，利用事件冒泡原理】

解决后生成元素的事件绑定问题

- 浏览器一刷新，for循环就执行完了，点击1, 2会打印出1，2
- 但是当我们点击button，新生成的dom上没有绑定事件的

```
<button id="btn">click</button>
<ul>
  <li>1</li>
  <li>2</li>
</ul>

<script>
  var oBtn = document.getElementById('btn');
  var aLi = document.getElementsByTagName('li');
  var oUl = document.getElementsByTagName('ul')[0];

  for(var i = 0; i < aLi.length; i++) {
    aLi[i].onclick = function() {
      console.log(this.innerHTML);
    }
  }

  oBtn.onclick = function() {
    var oLi = document.createElement('li');
    oLi.innerHTML = Math.random();

    Ul.appendChild(oLi);
  }
</script>
```

事件委托写法，把事件绑定在父元素上，在这里就是绑在ul上 【利用了事件冒泡原理】

```
// 点击子元素的时候，通过事件冒泡，会触发父元素的点击事件，通过e.target判断点击的元素
oUl.onclick = function(e) {
  if(e.target.tagName === 'LI') {
    console.log(e.target.innerHTML);
  }
}
```

11. 浏览器缓存

- cookie
  - 存cookie：

    ```
    var date = new Date();
    date.setDate(20);
    document.cookie = "name=11; expires="+date;
    document.cookie = "age=20; expires="+date;
    ```

  - 取cookie

    ```
    console.log(document.cookie);
    ```

    取某一个cookie

    ```
    function getCookie(key) {
      // "xxx='zs'; agee=20"
      var arr = document.cookie.split(';');       // ["name='zs'", "age=20"]
      for (var i = 0; i < arr.length; i++) {      // "name='zs'"  "age=20"
        var arr2 = arr[i].split('=');             // ["name", "zs"]
        if (key == arr2[0]) {
          return arr2[1];
        }
      }
    }
    ```

- sessionStorage
  - 存储和获取和删除

```
sessionStorage.setItem('name', 'zs');
```

```
sessionStorage.getItem('name');
```

```
sessionStorage.removeItem('name');
```

```
sessionStorage.clear();
```

- localStorage
  - 存储和获取和删除

```
localStorage.setItem('age', '111');
```

```
localStorage.getItem('age');
```

```
localStorage.removeItem('age');
```

总结 三者的共同点和区别

1. 共同点：都是保存在浏览器端，且同源的

2. 区别：

- **cookie数据始终在同源的http请求中携带（即使不需要）**，即cookie在浏览器和服务器间来回传递。而sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存。
- **存储大小限制也不同**，cookie数据不能超过4k，同时因为每次http请求都会携带cookie，所以cookie只适合保存很小的数据，如会话标识。sessionStorage和localStorage虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大。
- **生命周期不同**
  - sessionStorage：关闭浏览器就删除。仅在当前浏览器窗口关闭前有效，自然也就不可能持久保持
  - localStorage：一直有，除非手动删除。始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据
  - cookie：可以设置过期时间。cookie只在设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭
- **作用域不同**

- sessionStorage：不在不同的浏览器窗口中共享，即使是同一个页面
- localStorage：在所有同源窗口中都是共享的
- cookie：也是在所有同源窗口中都是共享的。

12. 跨域：同源策略的限制

- 理解跨域首先必须要了解同源策略。同源策略是浏览器为安全性考虑实施的非常重要的安全策略。
- 何谓同源?
  - URL由协议、域名、端口和路径组成，如果两个URL的协议、域名和端口相同，则表示它们是同源
- 同源策略：
  - 浏览器的同源策略，限制了来自不同源的“document”或脚本，对当前“document”读取或设置某些属性

13. EventLoop 事件循环

(1) 为什么JavaScript是单线程

- JavaScript语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。那么，为什么JavaScript不能有多个线程呢？这样能提高效率啊。
- JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准?
- 所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

(2) 宏任务&&微任务

- macro-task（宏任务）：包括整体代码script，setTimeout，setInterval
- micro-task（微任务）：Promise，process.nextTick
- 进入整体代码（宏任务）后，开始第一次循环。接着执行所有的微任务。然后再次从宏任务开始，找到其中一个任务队列执行完毕，再执行所有的微任务。

14. 垃圾回收（GC）

- 当一个对象没有任何的变量或属性对它进行引用，此时我们将永远无法操作该对象，此时这种对象就是一个垃圾，这种对象过多会占用大量的内存空间，导致程序运行变慢，所以这种垃圾必须进行清理。
- 在JS中拥有自动的垃圾回收机制，会自动将这些垃圾对象从内存中销毁，我们不需要也不能进行垃圾回收的操作
- 我们需要做的只是将不再使用的对象设置null即可