1. Object.create 和 Object.setPrototypeof有什么区别？

Object.create

会创建一个一个新的对象。

Object.setPrototype(a, b)

是把一个现存的对象\_\_proto\_\_指向一个原型对象。（将a的\_\_proto\_\_指向b）

Object.getPrototypeOf(a)

// 获取a对象的原型，可以用来模拟super，一旦这个a里面没有某个方法，会往上

# prototype和 \_ \_proto\_ \_

function Super () {

...

}

Super.prototype.func = function () {

...

}

var sub = new Super();

1. sub上并没有prototype，取而代之的是\_\_proto\_\_，同时原型链上的func方法也被挂载在了\_\_proto\_\_属性上。

问题：构造器上的prototype去哪儿了，窃取了func的\_\_proto\_\_是什么?

1. 通过proto实现的继承；

随意打印一个对象，发现它们都有\_\_proto\_\_属性，而\_\_proto\_\_中也有\_\_proto\_\_，一级一级追溯，会发现最终的\_\_proto\_\_都是Object

* 在 js 中并没有真正意义上的继承关系，**通过new实现的继承可以通过call方式来实现；**

var sub1 = {}

sub1.\_\_proto = Super.prototype

Super.call(sub1)

* call方式是将sub1的\_\_proto\_\_指向了Super的prototype，再改变执行环境，从而实现继承。
* 问：那new的对象的\_\_proto\_\_是不是也等于Super的prototype呢？ true

当调用一个方法时，会先查询这个对象本身的属性，如果没有这个方法，再到它的\_\_proto\_\_上查找，如果依然没有，就到\_\_proto\_\_的\_\_proto\_\_上查找，这样一级一级往上也就形成了原型链。

对象的继承本质上通过\_\_proto\_\_实现，而prototype在整个原型链的实现中只是起到了一个辅助的作用。

# basic

## 声明

1. let
2. 块级作用域，不能在同一作用域重复声明；(让声明的变量在指定块的作用域外无法被访问)
3. let声明的变量,只在let命令声明时所在的代码块内有效{};
4. **var**声明的变量**没有块级作用域**，**只有函数作用域**；

var无论实际声明位置在何处，都会被视为声明于所在函数的顶部（如果声明不在任意函数内，则视为在全局作用域的顶部）；

**let和const不会进行变量提升**；

使用let和const声明的变量，在声明处之前无法被访问，即使安全的运算，如typeof；

1. const

1) 定义常量，一旦定义，只能读取，不能进行其它操作；

2) 需要在声明时进行初始化；

3) let 和 const 声明的变量，作用域在声明它们的语句块中。

## 字符串模板

1) 拼接字符串

以前： 使用+号连接 var message = "The user "+ user.firstName

现在： 使用 var message = `The user ${user.firstName}`;

2) 模板字符串表达式

## 扩展（剩余）运算符

作用： 将一个数组转为用逗号分隔的参数序列。

该操作符把数组分成单个变量，最简单的用例是串联数组。

1. 作为函数的形式参数时，称之为剩余运算符（将剩余的内容放到一个数中）

setTimeout((...args)=>{

console.log(args, args.length) // [20, 30] 2

}, 1000, '20','30')

函数调用: myFunction(...iterableObj);

2. 数组/对象 字面量 （扩展运算符）

[...[1, 2, 3], 4, 5, 6];

{...{name: 'zhangdan'}, ...{age: '20'}}

=> {name: 'zhangsan', age: '20'} //es7语法

【应用】

* 函数传参

function printPerson(first, last, age) {

console.log(first, last age);

}

printPerson(...["Ted", "Neward", 45]);

* 替代Function.prototype.apply方法，将一个数组展开成多个参数:

1. Math.max(1,2,3) =>

Math.max.apply(null, [1,2,3]) => Math.max(...[1,2,3]))

2) function myFunction(x, y, z) { }

myFunction.apply(null, [0, 1, 2]);

=>

myFunction(...[0, 1, 2]);

* 数据解构

*let* cold = ['autumn', 'winter'];  
*let* [autumn, ...otherSeasons] = cold;  
console.log(autumn) *// "autumn"*console.log(otherSeasons) *// => ['winter']*

* 数据构造

1. 合并对象 (两个对象连接返回新的对象)

*var* obj = {...{name: 'zhangsan'},...{name: 'wangwu', age:20}}  
console.log(obj) *//{ name: 'wangwu', age: 20 }*

1. 串联数组（合并数组）

即：两个数组连接返回新的数组

*var* arr1 = [0, 1, 2]  
*var* arr2 = [...arr1, 3, 4, 5]  
console.log(arr2) *//[ 0, 1, 2, 3, 4, 5 ]  
  
var* arr3 = [0, 1].concat(['a', 'b'], ['x', 'y'])  
*var* arr4 = [...[0, 1], ...['a', 'b'], ...['x', 'y']]  
console.log(arr3) *// [ 0, 1, 'a', 'b', 'x', 'y' ]*console.log(arr4) *// [ 0, 1, 'a', 'b', 'x', 'y' ]  
  
//在中间插入数组  
var* parts = ['shoulder', 'knees']  
*var* lyrics = ['head', ...parts, 'toes']  
console.log(lyrics) *// [ 'head', 'shoulder', 'knees', 'and', 'toes' ]  
  
//将一个数组添加到另一个数组的尾部  
var* arr5 = [1, 2]  
*var* arr6 = ['a', 'b']  
*//Array.prototype.push.apply(arr5, arr6)*arr5.push(...arr6)  
console.log(arr5) *//[ 1, 2, 'a', 'b' ]*

# Array

es5数组

pop shift unshift push splice reverse sort (改变原数组)

slice concat indexOf lastIndexof

* 问：for, forEach, for in, for of之间的区别？\*\*

1. for(let i = 0; i < arrlength; i++) {}

2. arr.forEach((item, index) => {}) //不支持return，无返回值

3. for(let key in arr) {} //key会变成String类型, 能遍历到数组的私有属性

4. for(let key of arr) {} //可以return,又不会遍历到私有属性，不能遍历对象

## es6新增数组方法

|  |  |  |  |
| --- | --- | --- | --- |
|  | **作用** | **返回结果** | **回调函数的返回结果** |
| filter | 过滤 | 过滤后的新数组 | 返回true，表示这一项放到新数组中  （一般用于删除数组的某项） |
| map | 映射 | 新数组 | 返回什么，这一项就是什么  （一般用于更新数组） |
| some |  |  | 找到立即停止返回true，找不到返回false |
| every |  |  | 找到停止返回false 。  每一项都返回true,才返回true |
| includes |  | true/false | 找到返回true，否则返回false |
| find | 找到具体的某一项 | 返回找到的那一项 | 找到立即停止，返回true，未找到返回undefined; |
| reduce | 收敛 | 返回的是叠加后的结果； |  |

[1,2,3,4,5].filter((item) => item>1 && item <5);

[1,2,3,4,5].map(function(item){

return `<li>${item}</li>`;

}).join('');

[1,2,3,4,55,555].some((item,idx) => {

return item.toString().indexOf("5") > -1;

}) //true，不会返回idx之类的其它信息

[1,2,3,4,55,555].find((item,idx) => {

return item.toString().indexOf("5") > -1

});

[1,2,3,4].includes(5)

[].reduce((prev, cur, nIndex, arr) => {}, initValue)

/\*\*返回叠加后的结果

\* callback 函数接受4个参数：之前值、当前值、索引值以及数组本身；

\* initValue 参数可选，表示初始值；

\* 若指定，则当作最初使用的previous值；

\* 若缺省，则使用数组的第一个元素作为prev初始值，同时cur往后排一位，相比有initValue值少一次迭代；

\*\*/

[1,2,3,4,5].reduce(function(prev,current,idx,arr){ // 求数组项和

return prev+next; //如果无返回值，prev下次是undefined

});

/\*\*

\* 说明：

\* 1. 因为initValue不存在，因此一开始的prev值等于数组的第一个元素;

\* 2. 从而current值在第一次调用的时候就是2;

\*\*/

urls.reduce((promise, url) => {

return promise.then(() => loadImg(url)).then(addToHtml)

}, Promise.resolve())

arr.reduce((prev, cur) => prev.concat(cur)

# 函数

函数参数默认值

let sayHello = function (message = "Hi") {

console.log(message);

}

箭头函数

var fun = (arg1, arg2) => {

//do something

自己没有this，没有arguments，this指向函数所在的上下文环境，找上一级的this

}

## bind apply call 三者区别和联系

JavaScript 的一大特点是，函数存在「定义时上下文」和「运行时上下文」以及「上下文是可以改变的」。

### call & apply

call和apply都是为了改变某个函数运行时的上下文（context）而存在的（改变函数体内部 this 的指向）

当一个 object 没有某个方法，但其他的有，可以借助call或apply用其它对象的方法来操作。

* **call与apply的区别：**

func.call(this, arg1, arg2);

func.apply(this, [arg1, arg2]);

this 是你想指定的上下文，可以是任何一个JS对象;call 需要把参数按顺序传递进去，而 apply 则是把参数放在数组里；

* 【常见应用】：

1) 数组之间追加

var array1 = [12 , "foo" , {name "Joe"} , -2458];

var array2 = ["Doe" , 555 , 100];

Array.prototype.push.apply(array1, array2); /\* array1 值为 [12 , "foo" , {name "Joe"} , -2458 , "Doe" , 555 , 100] \*/

2) 获取数组中的最大值和最小值

var numbers = [5, 458 , 120 , -215 ];

var maxInNumbers = Math.max.apply(Math, numbers), //458

maxInNumbers = Math.max.call(Math,5, 458 , 120 , -215); //458

//number 本身没有 max 方法，但是 Math 有，我们就可以借助 call 或者 apply 使用其方法。

1. 验证是否是数组（前提是toString()方法没有被重写过）

function isArray(obj){

returnObject.prototype.toString.call(obj) === '[object Array]' ;

}

2. 类（伪）数组使用数组方法

var domNodes = Array.prototype.slice.call(document.getElementsByTagName("\*"));

调用 getElementsByTagName , document.childNodes 之类的，返回NodeList对象都属于伪数组。不能应用 Array下的 push , pop 等方法。（arguments 对象也是伪数组）。

通过 Array.prototype.slice.call 转换为真正的数组的带有 length 属性的对象，这样 domNodes 就可以应用 Array 下的所有方法了.

3. 定义一个 log 方法，让它可以代理 console.log 方法；并给每个log方法加一个前缀；

function log(){

var args = Array.prototype.slice.call(arguments);

//arguments参数是个伪数组,转化为标准数组，从而应用apply方法

args.unshift('(app)');

console.log.apply(console, args);

};

### bind

与apply及call方法类似，也是改变函数体内this的指向

MDN的解释是：

bind()方法会 \*\* 创建一个新函数 \*\*，称为绑定函数。当调用这个绑定函数时，绑定函数会以创建它时传入 bind()方法的第一个参数作为 this，传入 bind() 方法的第二个以及以后的参数加上绑定函数运行时本身的参数按照顺序作为原函数的参数来调用原函数。

在单例模式中，通常我们会使用 \_this , that , self 等保存 this ，这样我们可以在改变了上下文之后继续引用到它

var foo = {

bar : 1,

eventBind: function() {

var \_this = this;

$('.someClass').on('click',function(event) {

/\* Act on the event \*/

console.log(\_this.bar); //1

});

}

}

由于 Javascript 特有的机制:

上下文环境在 eventBind:function(){ } 过渡到

$('.someClass').on('click',function (event) { }) 发生了改变;

上述使用变量保存 this 这些方式都是有用的，也没有什么问题。当然使用 bind() 可以更加优雅的解决这个问题

var foo = {

bar : 1,

eventBind: function(){

$('.someClass').on('click',function(event) {

/\* Act on the event \*/

console.log(this.bar); //1

}.bind(this));

}

}

a. bind() 创建了一个函数

b. 当这个click事件绑定在被调用的时候，它的 this 关键词会被设置成被传入的值（这里指调用bind()时传入的参数）。

c. 因此，这里我们传入想要的上下文 this(其实就是 foo )，到 bind() 函数中。然后，当回调函数被执行的时候， this 便指向 foo 对象。

d. 注意: 多次绑定bind()是无效的

var bar = function(){console.log(this.x);}

var foo = {x:3}

var sed = {x:4}

var func = bar.bind(foo).bind(sed);

func(); //3

### apply、call、bind 异同

var obj = {x: 81,};

var foo = {

getX: function() {return this.x;}

}

console.log(foo.getX.bind(obj)()); //81 注意看使用 bind() 方法的，他后面多了对括号

console.log(foo.getX.call(obj)); //81

console.log(foo.getX.apply(obj)); //81

```

区别：

1. 希望改变上下文环境之后并非立即执行，而是回调执行的时候，使用 bind() 方法。

2. apply/call 则会立即执行函数。

总结

1. apply 、 call 、bind 三者都是用来改变函数的this对象的指向的；

2. apply 、 call 、bind 三者第一个参数都是this要指向的对象，也就是想指定的上下文；

3. apply 、 call 、bind 三者都可以利用后续参数传参；

bind是返回对应函数，便于稍后调用；apply、call则是立即调用 。

# Generator函数

特点：

Generator函数分段执行，yield暂停，next恢复执行；

1. function关键字与函数名之间有个\*；
2. 函数体内部使用yield，定义状态；
3. 调用函数后，该函数不执行，返回的不是函数运行结果，而是一个指向内部状态的指针对象(遍历器对象)； （只有调用next，函数才会执行）
4. 调用next,指针移动到下一状态；(每次next,都从函数头部或上次停下来的地方执行，直到遇到yield或return)
5. yield只能用在Genrator函数中；

function\* hello () {

yield 'hello';

yield 'world';

return 'ending';

}

var hw = hello();

console.log('generator函数运行结果：', hw); //{}

// next()方法的返回值 就是yield表达式的值，done表示遍历还未结束

console.log(hw.next()); //{ value: 'hello', done: false }

console.log(hw.next()); //{ value: 'world', done: false }

console.log(hw.next()); //{ value: 'ending', done: true }

console.log(hw.next()); //{ value: undefined, done: true }

# async await

是Generator函数的语法糖；

* 特点

1. async、await基于promise，是非阻塞的；
2. async函数，始终**返回Promise对象**；
3. 定义async函数方式
4. 函数表达式

const foo = **async** function () {}

const foo = **async** () => {}

1. 作为对象的方法

let obj = {

**async** foo () {}

}

obj.foo().then()

1. Class 的方法

class Storage {

constructor() {

this.cachePromise = caches.open('avatars');

}

**async getAvatar(name)** {

const cache = **await this.cachePromise**;

return cache.match(`/avatars/${name}.jpg`);

}

}

const storage = new Storage();

storage.getAvatar('jake').then();

1. async函数
2. async 返回Promise，因此可以通过then添加回调函数，那么怎么向then中的回调添加参数呢？
   * + async函数中return的返回值，将将作为回调的参数；
     + 返回的promise对象，必须等到内部所有await命令后面的Promise执行完，才会发生状态改变，除非遇到return或抛出错误。
     + 抛出错误，会导致返回的Promise对象变为reject状态；

async function test () {

return 'this is a test async function'

}

test().then(

resolveArg => console.log(resolveArg);

)

1. 当async函数内部抛出一个错误时，也会被catch到，下面三种catch错误的方式都可以:

async function errorTest () {

throw new Error('this is an error');

}

* + 在 then 的回调中捕获错误

errorTest().then(

resolve => console.log(resolve),

error => console.log(error)

)

* + 在 Promise 的 catch 方法中捕获

errorTest().catch(

error => console.log(error)

)

* + 在 try...catch 语句中捕获

try{

errorTest()

} catch (error) {

console.log(error)

}

1. await
   * await只能用在async函数中, 表示等待promise返回结果；
   * await可以是Promise也可以是普通数据类型，但如果是普通数据类型的话，会自动转换成状态为resolve的Promise
   * 只要await后面的Promise变成reject,那么整个async函数都会中断执行,并抛出响应的错误；即使这里没有return，也会传入到错误的回调函数；
   * 所以当一个 async 函数中有多个 await命令时，如果不想因为一个出错而导致其与的都无法执行，应将await放在try...catch语句中执行
2. 并发执行await命令

当一个 async 函数中有多个await时，这些 await是继发执行的，只有当前一个await后面的方法执行完毕后，才会执行下一个。

如果我们前后的方法由依赖关系，继发执行是没有问题的，但是如果并没有任何关系的话，这样就会很耗时，所以需要让这些await命令同时执行，也就是并发执行。

* 方法 1

let [res1, res2] = await Promise.all([func1(), func2()])

* 方法 2

let func1Promise = func1()

let func2Promise = func2()

let res1 = await func1Promise

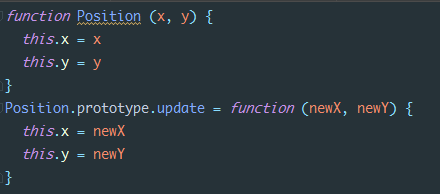
let res2 = await func2Promise

*var* movieDir = \_\_dirname + '/movies',  
 exts = ['.mkv', '.avi', '.mp4', '.rm', '.rmvb', '.wmv']  
  
*// 读取文件列表  
var* readFiles = *function* () {  
 *return new* Promise(*function* (resolve, reject) {  
 fs.readdir(movieDir, *function* (err, files) {  
 resolve(files.filter((v) => exts.includes(path.parse(v).ext)))  
 })  
 })  
}  
  
*// 获取海报  
var* getPoster = *function* (movieName) {  
 *let* url = `https://api.douban.com/v2/movie/search?q=${encodeURI(movieName)}`  
   
 *return new* Promise(*function* (resolve, reject) {  
 request({url: url, json: *true*}, *function* (error, response, body) {  
 *if* (error) {  
 *return* reject(error)  
 }  
 console.log('imageUrl: ', body.subjects[0].images.large)  
 resolve(body.subjects[0].images.large)  
 })  
 })  
}  
  
*// 保存海报  
var* savePoster = *function* (movieName, url) {  
 request.get(url)  
 .pipe(fs.createWriteStream(path.join(movieDir, movieName + '.jpg')))  
};  
  
(*async* () => {  
 *let* files = *await* readFiles()  
 *// await只能使用在原生语法  
 for* (*var* file *of* files) {  
 *let* name = path.parse(file).name  
 console.log(`正在获取【${name}】的海报`)  
 savePoster(name, *await* getPoster(name))  
 }  
 console.log('=== 获取海报完成 ===')  
})()

# Promise

# class

在 ES6 之前，js 中并没有 “类” 这个概念，而是通过原型链的方式实现了对象的继承。

1. 定义了一个Position类，里面的constructor方法是构造方法，this为实例对象;

ES5 的构造函数Position对应 ES6 Position类的构造方法。

1. Position类中还定义了一个update方法。

在定义类方法时，不需要function关键字，函数之间也不需要添加逗号，否则会报错。

1. constructor方法

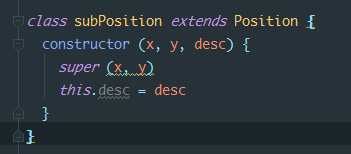
constructor是类的默认方法，new命令会调用该方法生成一个实例。

constructor方法必须存在，如果没有显示定义，则会被默认添加；

使用class定义类，不存在变量提升，所以在使用一个通过class定义的类时，必须保证这个类已经被定义过了。

1. 类的继承

通过extends关键字实现类的继承；



子类subPosition**必须在constructor中调用super方法**，否则将会报错，因为子类没有自己的this对象，而是继承父类的this后进行加工，如果不调用super方法，子类将得不到this，也因此，子类在调用super方法前，将不能使用this。

1. super关键字

super关键字既可以作为方法使用，也可以作为对象使用。

1. 作为方法

super()只能够在子类的constructor方法中调用，代表**调用父类的构造函数**。

1. 作为对象

在普通方法中，super指向父类的原型对象，可以通过super调用到父类原型上的方法

在静态方法中，super指向父类对象。

1. getter和setter

在class内部，可以通过get set关键字设置属性的取值和存值方法



1. 静态方法

在class内部方法的前面加上static关键字，就可以使这个方法成为静态方法；

**静态方法不会被实例所继承，并且可以直接通过类来调用。**