视频地址：链接：https://pan.baidu.com/s/17EPb69bcJDFZ97DWV0bs9Q

提取码：ksx3

认识ECMAScript

它是一种由ECMA组织（前身为欧洲计算机制造商协会）制定和发布的脚本语言规范。而我们学的JavaScript是ECMA的实现, 但术语ECMAScript和JavaScript平时表达同一个意思。

JS包含三个部分：

1). ECMAScript（核心）

2). 浏览器端扩展：DOM（文档对象模型）和 BOM（浏览器对象模型）

3). 服务器端扩展：Node

ES的几个重要版本：

ES5 : 09年发布

ES6(ES2015) : 15年发布, 也称为ECMA2015

ES7(ES2016) : 16年发布, 也称为ECMA2016 (变化不大)

扩展学习参考:

1). ES5 :

http://www.zhangxinxu.com/wordpress/2012/01/introducing-ecmascript-5-1/

http://www.ibm.com/developerworks/cn/web/wa-ecma262/

2). ES6

http://es6.ruanyifeng.com/

3). ES7

http://www.w3ctech.com/topic/1614

ECMAScript5

浏览器对ES5的支持

\1. IE8只支持defineProperty、getOwnPropertyDescriptor的部分特性和JSON的新特性

\2. IE9不支持严格模式, 其它都可以

\3. IE10和其他主流浏览器都支持了

\4. PC端开发需要注意IE9以下的兼容, 但移动端开发时不需要

严格模式

\1. 理解：

除了正常运行模式(混杂模式)，ES5添加了第二种运行模式："严格模式"（strict mode）。

顾名思义，这种模式使得Javascript在更严格的语法条件下运行。

\2. 目的/作用：

消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为。

消除代码运行的一些不安全之处，保证代码运行的安全。

为未来新版本的Javascript做好铺垫。

\3. 使用：

在全局或函数的第一条语句定义为: 'use strict'。

如果浏览器不支持, 只解析为一条简单的语句, 没有任何副作用。

\4. 语法和行为改变：

必须用var声明变量；

创建eval作用域；

禁止this指向window；

对象不能有重名的属性；

函数不能有重名的形参。

\5. 学习参考：

http://www.ruanyifeng.com/blog/2013/01/javascript_strict_mode.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>01_严格模式</title>
</head>
<body>
<script type="text/javascript">
    'use strict';
    var age = 12;
    console.log(age);
    function Person(name, age) {
        this.name = name;
        this.age = age;
    }
    // Person('kobe', 39); // 报错
    new Person('kobe', 39);
    setTimeout(function () {
        console.log(this);//window
    }, 1000);

    //* 创建eval作用域
    var name = 'kobe';
    eval('var name = "anverson";alert(name)'); // 打印anverson
    console.log(name); // 打印anverson

    var obj = {
        name : 'kobe',
        name : 'weide' // 重名了会报错
    };
```

```
    console.log(obj);

</script>

</body>
</html>
```

JSON对象

\1. JSON.stringify(obj/arr)： js对象(数组)转换为json对象(数组)

\2. JSON.parse(json)：json对象(数组)转换为js对象(数组)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>02_JSON对象</title>
</head>
<body>

<script type="text/javascript">
    var obj = {
        name : 'kobe',
        age : 39
    };
    obj = JSON.stringify(obj);
    console.log( typeof obj);// String
    obj = JSON.parse(obj);
    console.log(obj);// Object

</script>
</body>
</html>
```

Object扩展

ES5给Object扩展了好一些静态方法, 常用的3个：

\1. Object.create(prototype[, descriptors])： 创建一个新的对象

1). 以指定对象为原型创建新的对象；

2). 指定新的属性, 并对属性进行描述：

value : 指定值；

writable : 标识当前属性值是否是可修改的, 默认为false；

configurable: 标识当前属性是否可以被删除 默认为false；

enumerable： 标识当前属性是否能用for in 枚举 默认为false。

\2. Object.defineProperties(object, descriptors) : 为指定对象定义扩展多个属性

get : 用来得到当前属性值的回调函数;

set : 用来监视当前属性值变化的回调函数;

存取器属性：setter,getter一个用来存值，一个用来取值。-

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>03_Object扩展</title>
</head>
<body>
<script type="text/javascript">
    var obj = {name : 'curry', age : 29}
    var obj1 = {};
    obj1 = Object.create(obj, {
        sex : {
            value : '男',
            writable : true,
            configurable : true,
            enumerable : true
        }
    });
    obj1.sex = '女';
    console.log(obj1.sex);

    // delete obj1.sex;
    console.log(obj1);
    for(var i in obj1){
        console.log(i);
    }

    //Object.defineProperties(object, descriptors)
    var obj2 = {
        firstName : 'curry',
        lastName : 'stephen'
    };
    Object.defineProperties(obj2, {
        fullName : {
            get : function () { // 获取扩展属性的值
                return this.firstName + '-' + this.lastName
            },
            set : function (data) { // 监听扩展属性，当扩展属性发生变化时自动回调，自动
调用后会将变化的值作为实参传进来
                var names = data.split('-');
                this.firstName = names[0];
                this.lastName = names[1];
            }
        }
    });
    console.log(obj2.fullName);
    obj2.firstName = 'tim';
    obj2.lastName = 'duncan';
    console.log(obj2.fullName);
    obj2.fullName = 'kobe-bryant';
    console.log(obj2.fullName);
</script>
```

```
</body>
</html>
```

对象本身的两个方法:

* get propertyName(){} 用来得到当前属性值的回调函数

* set propertyName(){} 用来监视当前属性值变化的回调函数

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<script type='text/javascript'>
    var obj = {
        firstName : 'kobe',
        lastName : 'bryant',
        get fullName(){
            return this.firstName + ' ' + this.lastName
        },
        set fullName(data){
            var names = data.split(' ');
            this.firstName = names[0];
            this.lastName = names[1];
        }
    };
    console.log(obj.fullName);
    obj.fullName = 'curry stephen';
    console.log(obj.fullName);

</script>
</body>
</html>
```

Array扩展

ES5给数组对象添加了一些方法, 常用的5个:

\1. Array.prototype.indexOf(value) : 得到值在数组中的第一个下标。

\2. Array.prototype.lastIndexOf(value) : 得到值在数组中的最后一个下标。

\3. Array.prototype.forEach(function(item, index){}) : 遍历数组。

\4. Array.prototype.map(function(item, index){ }) : 遍历数组返回一个新的数组，返回加工之后的值。

\5. Array.prototype.filter(function(item, index){ }) : 遍历过滤出一个新的子数组， 返回条件为true的值。

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>04_Array扩展</title>
</head>
<body>
<script type="text/javascript">
  /*
   需求:
   1. 输出第一个6的下标
   2. 输出最后一个6的下标
   3. 输出所有元素的值和下标
   4. 根据arr产生一个新数组,要求每个元素都比原来大10
   5. 根据arr产生一个新数组, 返回的每个元素要大于4
   */

    var arr = [1, 4, 6, 2, 5, 6];
    console.log(arr.indexOf(6));//2
    //Array.prototype.lastIndexOf(value) : 得到值在数组中的最后一个下标
    console.log(arr.lastIndexOf(6));//5

    //Array.prototype.forEach(function(item, index){}) : 遍历数组
    arr.forEach(function (item, index) {
        console.log(item, index);
    });

    //Array.prototype.map(function(item, index){}) : 遍历数组返回一个新的数组，返回加工之后的值
    var arr1 = arr.map(function (item, index) {
        return item + 10
    });
    console.log(arr, arr1);

    //Array.prototype.filter(function(item, index){}) : 遍历过滤出一个新的子数组， 返回条件为true的值
    var arr2 = arr.filter(function (item, index) {
        return item > 4
    });
    console.log(arr, arr2);


</script>
</body>
</html>
```

Function扩展

Function.prototype.bind(obj)： 将函数内的this绑定为obj, 并将函数返回。

面试题: 区别bind()与call()和apply()

* 都能指定函数中的this;

* call()/apply()是立即调用函数;

* bind()是将函数返回。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>05_Function扩展</title>
</head>
<body>
<script type="text/javascript">
    function fun(age) {
        this.name = 'kobe';
        this.age = age;
        console.log('ddddddddddddddd');
    }
    var obj = {};
    fun.bind(obj, 12)();
    console.log(obj.name, obj.age);

    var test = {username: 'kobe'};
    function foo(data){
        console.log(this,data);
    }
    // call和apply传入参数的形式不一样
    foo.call(obj,33); // 直接从第二个参数开始，依次传入
    foo.apply(obj,[33]);// 第二参数必须是数组，传入放在数组用

</script>
</body>
</html>
```

ECMAScript6

常用

关键字扩展

① let关键字

作用：与var类似, 用于声明一个变量。

特点： 在块作用域内有效;

不能重复声明;

不会预处理, 不存在提升。

应用： 循环遍历加监听，使用let取代var是趋势。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>01_let关键字</title>
</head>
<body>
```

```
<button>测试1</button>
<br>
<button>测试2</button>
<br>
<button>测试3</button>
<br>

<script type="text/javascript">
    //console.log(age);// age is not defined
    let age = 12;
    //let age = 13;不能重复声明
    console.log(age);
    let btns = document.getElementsByTagName('button');
    for(let i = 0;i<btns.length;i++){
            btns[i].onclick = function () {
                alert(i);
            }
    }

</script>
</body>

</html>
```

② const关键字

作用：定义一个常量。

特点：不能修改；

也是块作用域有效。

应用：保存应用需要的常量数据。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>02_const关键字</title>
</head>
<body>
<script type="text/javascript">
    const sex = '男';
    console.log(sex);
    //sex = '女';//不能修改
    console.log(sex);
</script>
</body>
</html>
```

变量的解构赋值

理解：从数组或对象中提取值, 对多个变量进行赋值。

数组的解构赋值：let [a,b] = [1, 'atguigu'];

对象的解构赋值：let {n, a} = {n:'tom', a:12}

用途： 给多个形参赋值;

交换2个变量的值;

从函数返回多个值。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>03_变量的解构赋值</title>
</head>
<body>
<script type="text/javascript">
    let obj = {name : 'kobe', age : 39};
//    let name = obj.name;
//    let age = obj.age;
//    console.log(name, age);
    //对象的解构赋值
    let {age} = obj;
    console.log(age);
//    let {name, age} = {name : 'kobe', age : 39};
//    console.log(name, age);

    //3. 数组的解构赋值  不经常用
    let arr = ['abc', 23, true];
    let [a, b, c, d] = arr;
    console.log(a, b, c, d);
    //console.log(e);
    function person(p) {//不用解构赋值
        console.log(p.name, p.age);
    }
    person(obj);

    function person1({name, age}) {
     console.log(name, age);
    }
    person1(obj);

</script>
</body>
</html>
```

模板字符串

\1. 模板字符串： 简化字符串的拼接。

1). 模板字符串必须用``;

2). 变化的部分使用${xxx}定义。

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>04_模板字符串</title>
</head>
<body>
<!--
1. 模板字符串 : 简化字符串的拼接
  * 模板字符串必须用 `` 包含
  * 变化的部分使用${xxx}定义
-->
<script type="text/javascript">
    let obj = {
        name : 'anverson',
        age : 41
    };
    console.log('我叫:' + obj.name + ', 我的年龄是: ' + obj.age);

    console.log(`我叫:${obj.name}, 我的年龄是: ${obj.age}`);
</script>
</body>
</html>
```

对象增强表达

简化的对象写法:

* 省略同名的属性值;

* 省略方法的function。

例如:

let x = 1;

let y = 2;

let point = {

x,

y,

setX (x) {this.x = x}

};

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>05_简化的对象写法</title>
</head>
<body>
<script type="text/javascript">

    let x = 3;
    let y = 5;
```

```
    //普通额写法
//    let obj = {
//        x : x,
//        y : y,
//        getPoint : function () {
//            return this.x + this.y
//        }
//    };
    //简化的写法
    let obj = {
        x,
        y,
        getPoint(){
            return this.x
        }
    };
    console.log(obj, obj.getPoint());
</script>
</body>
</html>
```

函数扩展

1. 箭头函数

var fun = function (v) {

return v+3;

}

var fun2 = v => v+3;

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>06_箭头函数</title>

</head>
<body>
    <button id="btn">测试箭头函数this_1</button>
    <button id="btn2">测试箭头函数this_2</button>


<!--
* 作用: 定义匿名函数
* 基本语法:
  * 没有参数: () => console.log('xxxx')
  * 一个参数: i => i+2
  * 大于一个参数: (i,j) => i+j
  * 函数体不用大括号: 默认返回结果
  * 函数体如果有多个语句, 需要用{}包围, 若有需要返回的内容, 需要手动返回
* 使用场景: 多用来定义回调函数

* 箭头函数的特点:
```

```
    1、简洁
    2、箭头函数没有自己的this，箭头函数的this不是调用的时候决定的，而是在定义的时候处在的对象就是它的this
   3、扩展理解：  箭头函数的this看外层的是否有函数，
         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。
-->
<script type="text/javascript">
  let fun = function () {
       console.log('fun()');
   };
   fun();
   //没有形参，并且函数体只有一条语句
   let fun1 = () => console.log('fun1()');
    fun1();
   console.log(fun1());
    //一个形参，并且函数体只有一条语句
    let fun2 = x => x;
    console.log(fun2(5));
    //形参是一个以上
    let fun3 = (x, y) => x + y;
    console.log(fun3(25, 39));//64

    //函数体有多条语句
    let fun4 = (x, y) => {
        console.log(x, y);
        return x + y;
    };
    console.log(fun4(34, 48));//82

    setTimeout(() => {
        console.log(this);
    },1000)

   let btn = document.getElementById('btn');
   //没有箭头函数
   btn.onclick = function () {
       console.log(this);//btn
   };
   //箭头函数
   let btn2 = document.getElementById('btn2');

    let obj = {
        name : 'kobe',
        age : 39,
        getName : () => {
            btn2.onclick = () => {
                console.log(this);//obj
            };
        }
    };
    obj.getName();


 function Person() {
     this.obj = {
         showThis : () => {
             console.log(this);
```

```
        }
    }
}
    let fun5 = new Person();
    fun5.obj.showThis();

</script>

</body>
</html>
```

2. 形参的默认值

function Point(x = 1,y = 2) {

this.x = x;

this.y = y;

}

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>08_形参默认值</title>
</head>
<body>
<!--
    * 形参的默认值----当不传入参数的时候默认使用形参里的默认值
    function Point(x = 1,y = 2) {
    this.x = x;
    this.y = y;
    }
-->
<script type="text/javascript">
    //定义一个点的坐标
    function Point(x=12, y=12) {
        this.x = x;
        this.y = y;
    }
    let point = new Point(25, 36);
    console.log(point);
    let p = new Point();
    console.log(p);
    let point1 = new Point(12, 35);
    console.log(point1);

</script>

</body>
</html>
```

3. rest(可变)参数

function add(... values) {

let sum = 0;

for(value of values) {

sum += value;

}

return sum;

}

4. 扩展运算符(...)

var arr1 = [1,3,5];

var arr2 = [2,...arr1,6];

arr2.push(...arr1);

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>07_3点运算符</title>
</head>
<body>
<!--
* 用途
1. rest(可变)参数
    * 用来取代arguments 但比arguments灵活,只能是最后部分形参参数
  function add(...values) {
    let sum = 0;
    for(value of values) {
      sum += value;
    }
    return sum;
  }
2. 扩展运算符
  let arr1 = [1,3,5];
  let arr2 = [2,...arr1,6];
  arr2.push(...arr1);
-->
<script type="text/javascript">
    function fun(...values) {
        console.log(arguments);
//        arguments.forEach(function (item, index) {
//            console.log(item, index);
//        });
        console.log(values);
        values.forEach(function (item, index) {
            console.log(item, index);
        })
    }
    fun(1,2,3);

    let arr = [2,3,4,5,6];
    let arr1 = ['abc',...arr, 'fg'];
```

```
    console.log(arr1);


</script>

</body>
</html>
```

class类

\1. 通过class定义类；

\2. 在类中通过constructor定义构造方法；

\3. 通过new来创建类的实例；

\4. 通过extends来实现类的继承；

\5. 通过super调用父类的构造方法。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>12_class</title>
</head>
<body>
</body>
<!--
1. 通过class定义类/实现类的继承
2. 在类中通过constructor定义构造方法
3. 通过new来创建类的实例
4. 通过extends来实现类的继承
5. 通过super调用父类的构造方法
6. 重写从父类中继承的一般方法
-->
<script type="text/javascript">
    class Person {
        //调用类的构造方法
        constructor(name, age){
            this.name = name;
            this.age = age;

        }
        //定义一般的方法
        showName(){
            console.log(this.name, this.age);
        }
    }
    let person = new Person('kobe', 39);
    console.log(person, person.showName());

    //定义一个子类
    class StrPerson extends Person{
        constructor(name, age, salary){
            super(name, age);//调用父类的构造方法
```

```
            this.salary = salary;
        }
        showName(){//在子类自身定义方法
            console.log(this.name, this.age, this.salary);
        }
    }
    let str = new StrPerson('weide', 38, 1000000000);
    console.log(str);
    str.showName();
</script>
</html>
```

Promise对象

1、 理解:

Promise对象: 代表了未来某个将要发生的事件(通常是一个异步操作);

有了promise对象, 可以将异步操作以同步的流程表达出来, 避免了层层嵌套的回调函数(俗称'回调地狱');

ES6的Promise是一个构造函数, 用来生成promise实例。

2.、使用promise基本步骤(2步):

① 创建promise对象:

```
    let promise = new Promise((resolve, reject) => {
      //执行异步操作
      if(异步操作成功) {
        resolve(value);
      } else {
        reject(errMsg);
      }
    })
```

② 调用promise的then()

```
    promise.then(function(
      result => console.log(result),
      errorMsg => alert(errorMsg)
    ))
```

3、promise对象的3个状态:

pending: 初始化状态;

fullfilled: 成功状态;

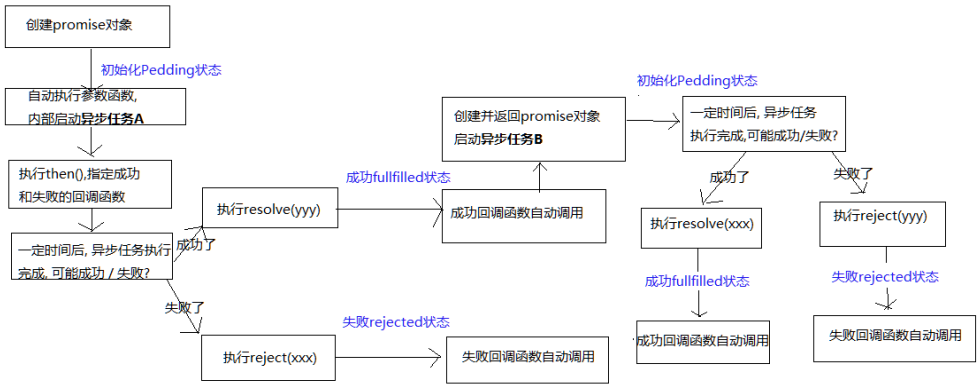
rejected: 失败状态。

4、应用:

使用promise实现超时处理;

使用promise封装处理ajax请求：

```
    let request = new XMLHttpRequest();
    request.onreadystatechange = function () {
    }
    request.responseType = 'json';
    request.open("GET", url);
    request.send();
```




```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>11_Promise对象</title>
</head>
<body>
<script type="text/javascript">

    //创建一个promise实例对象
    let promise = new Promise((resolve, reject) => {
        //初始化promise的状态为pending---->初始化状态
        console.log('1111');//同步执行
        //启动异步任务
        setTimeout(function () {
            console.log('3333');
            //resolve('atguigu.com');//修改promise的状态pending---->fullfilled（成
功状态）
            reject('xxxx');//修改promise的状态pending----->rejected(失败状态)
        },1000)
    });
    promise.then((data) => {
        console.log('成功了。。。' + data);
    }, (error) => {
        console.log('失败了' + error);
    });
    console.log('2222');


    //定义一个请求news的方法
    function getNews(url) {
        //创建一个promise对象
```

```
        let promise = new Promise((resolve, reject) => {
            //初始化promise状态为pending
            //启动异步任务
            let request = new XMLHttpRequest();
            request.onreadystatechange = function () {
                if(request.readyState === 4){
                    if(request.status === 200){
                        let news = request.response;
                        resolve(news);
                    }else{
                        reject('请求失败了。。。');
                    }
                }
            };
            request.responseType = 'json';//设置返回的数据类型
            request.open("GET", url);//规定请求的方法，创建链接
            request.send();//发送
        })
        return promise;
    }

    getNews('http://localhost:3000/news?id=2')
            .then((news) => {
                console.log(news);
                document.write(JSON.stringify(news));
                console.log('http://localhost:3000' + news.commentsUrl);
                return getNews('http://localhost:3000' + news.commentsUrl);
            }, (error) => {
                alert(error);
            })
            .then((comments) => {
                console.log(comments);
                document.write('<br><br><br><br><br>' +
JSON.stringify(comments));
            }, (error) => {
                alert(error);
            })
</script>

</body>

</html>
```

Symbol

前言：ES5中对象的属性名都是字符串，容易造成重名，污染环境。

Symbol：

概念：ES6中的添加了一种原始数据类型symbol(已有的原始数据类型：String, Number, boolean, null, undefined, 对象)

特点：

1、Symbol属性对应的值是唯一的，解决命名冲突问题

2、Symbol值不能与其他数据进行计算，包括同字符串拼串

3、for in, for of遍历时不会遍历symbol属性。

使用：

1、调用Symbol函数得到symbol值

```
let symbol = Symbol();
let obj = {};
obj[symbol] = 'hello';
```

2、传参标识

```
let symbol = Symbol('one');
let symbol2 = Symbol('two');
console.log(symbol);// Symbol('one')
console.log(symbol2);// Symbol('two')
```

3、内置Symbol值

* 除了定义自己使用的Symbol值以外，ES6还提供了11个内置的Symbol值，指向语言内部使用的方法。

- Symbol.iterator

* 对象的Symbol.iterator属性，指向该对象的默认遍历器方法(后边讲)。

```
// 等同于在指定的数据内结构上部署了iterator接口，
// 当使用for of去遍历某一个数据结构的时候，首先去找Symbol.iterator， 找到了就去遍历，没有找到的话不能遍历 xxx is not
let targetData = {
  [Symbol.iterator]: function () {
    let nextIndex = 0; // 记录指针的位置；
    return { // 遍历器对象
      next: function () {
        return nextIndex < this.length?{value: this[nextIndex++], done: false}:{value: undefined, done: true}
      }
    }
  }
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Symbol</title>
</head>
<body>
<script type="text/javascript">
    window.onload = function () {
      let symbol = Symbol();
      console.log(typeof symbol);
      console.log(symbol);

      // 用作对象的属性(唯一)
      let obj = {username: 'kobe', age: 39};
      obj[symbol] = 'hello';
      obj[symbol] = 'symbol';
      console.log(obj);
      for(let i in obj){
        console.log(i);
```

```
      }
    }
</script>

</body>
</html>
```

iterator遍历器

概念： iterator是一种接口机制，为各种不同的数据结构提供统一的访问机制

作用：

1、为各种数据结构，提供一个统一的、简便的访问接口；

2、使得数据结构的成员能够按某种次序排列；

3、ES6创造了一种新的遍历命令for...of循环，Iterator接口主要供for...of消费。

工作原理：

- 创建一个指针对象，指向数据结构的起始位置；

- 第一次调用next方法，指针自动指向数据结构的第一个成员；

- 接下来不断调用next方法，指针会一直往后移动，直到指向最后一个成员；

- 每调用next方法返回的是一个包含value和done的对象，{value: 当前成员的值,done: 布尔值}。

* value表示当前成员的值，done对应的布尔值表示当前的数据的结构是否遍历结束。

* 当遍历结束的时候返回的value值是undefined，done值为false。

原生具备iterator接口的数据(可用for of遍历)：

1、Array

2、arguments

3、set容器

4、map容器

5、String

。。。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Iterator遍历器</title>
</head>
<body>
  <script type="text/javascript">
      window.onload = function () {
        // 自定义iterator生成指针对象
        function mockIterator(arr) {
          let nextIndex = 0;
          return {
```

```
            next: function () {
              return nextIndex<arr.length?{value: arr[nextIndex++], done:
false}:{value: undefined, done: true}
            }
          }
        }

        let arr = [1,2,3,4,5];
        let iteratorObj = mockIterator(arr);
        console.log(iteratorObj.next());
        console.log(iteratorObj.next());
        console.log(iteratorObj.next());


        // 使用解构赋值以及...三点运算符时会调用iterator接口
        let arr1 = [1,2,3,4,5];
        let [value1, ...arr2] = arr1;
        // yield*语句
        function* generatorObj() {
          yield '1'; // 可遍历数据，会自动调用iterator函数
          yield '3';
        }
        let Go = generatorObj();
        console.log(Go.next());
        console.log(Go.next());
        console.log(Go.next());


        // 原生测试  数组
        let arr3 = [1, 2, 'kobe', true];
        for(let i of arr3){
          console.log(i);
        }
        // 字符串  string
        let str = 'abcdefg';
        for(let item of str){
          console.log(item);
        }


      }
  </script>
</body>
</html>
```

Generator函数

概念：

1、ES6提供的解决异步编程的方案之一；

2、Generator函数是一个状态机，内部封装了不同状态的数据；

3、用来生成遍历器对象；

4、可暂停函数(惰性求值), yield可暂停，next方法可启动。每次返回的是yield后的表达式结果。

特点：

1、function 与函数名之间有一个星号；

2、内部用yield表达式来定义不同的状态。

例如：

```
        function* generatorExample(){
          let result = yield 'hello'; // 状态值为hello
          yield 'generator'; // 状态值为generator
        }
```

3、generator函数返回的是指针对象(接11章节里iterator)，而不会执行函数内部逻辑。

4、调用next方法函数内部逻辑开始执行，遇到yield表达式停止，返回{value: yield后的表达式结果/undefined, done: false/true}。

5、再次调用next方法会从上一次停止时的yield处开始，直到最后。

6、yield语句返回结果通常为undefined， 当调用next方法时传参内容会作为启动时yield语句的返回值。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Generator函数</title>
</head>
<body>
<script type="text/javascript" src="./js/jquery-1.10.1.min.js"></script>
<script type="text/javascript">
    // 小试牛刀
    function* generatorTest() {
      console.log('函数开始执行');
      yield 'hello';
      console.log('函数暂停后再次启动');
      yield 'generator';
    }
    // 生成遍历器对象
    let Gt = generatorTest();
    // 执行函数，遇到yield后即暂停
    console.log(Gt); // 遍历器对象
    let result = Gt.next(); // 函数执行,遇到yield暂停
    console.log(result); // {value: "hello", done: false}
    result = Gt.next(); // 函数再次启动
    console.log(result); // {value: 'generator', done: false}
    result = Gt.next();
    console.log(result); // {value: undefined, done: true}表示函数内部状态已经遍历完毕

    // 对象的Symbol.iterator属性;
    let myIterable = {};
    myIterable[Symbol.iterator] = function* () {
      yield 1;
      yield 2;
```

```
      yield 4;
    };
    for(let i of myIterable){
      console.log(i);
    }
    let obj = [...myIterable];
    console.log(obj);

    console.log('------------------------------');
    // 案例练习
    /*
    * 需求:
    * 1、发送ajax请求获取新闻内容
    * 2、新闻内容获取成功后再次发送请求，获取对应的新闻评论内容
    * 3、新闻内容获取失败则不需要再次发送请求。
    *
    * */
    function* sendXml() {
      // url为next传参进来的数据
     let url = yield getNews('http://localhost:3000/news?newsId=2');
      yield getNews(url);
    }
    function getNews(url) {
      $.get(url, function (data) {
        console.log(data);
        let commentsUrl = data.commentsUrl;
        let url = 'http://localhost:3000' + commentsUrl;
        // 当获取新闻内容成功，发送请求获取对应的评论内容
        // 调用next传参会作为上次暂停是yield的返回值
        sx.next(url);
      })
    }

    let sx = sendXml();
    // 发送请求获取新闻内容
    sx.next();

</script>
</body>
</html>
```

async函数

概念： 真正意义上去解决异步回调的问题，同步流程表达异步操作。

本质： Generator的语法糖。

语法：

async function foo(){

await 异步操作;

await 异步操作;

}

特点：

1、不需要像Generator去调用next方法，遇到await等待，当前的异步操作完成就往下执行；

2、返回的总是Promise对象，可以用then方法进行下一步操作；

3、async取代Generator函数的星号*，await取代Generator的yield；

4、语意上更为明确，使用简单，经临床验证，暂时没有任何副作用。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>async函数</title>
</head>
<body>
<script type="text/javascript" src="./js/jquery-1.10.1.min.js"></script>
<script type="text/javascript">
    async function timeout(ms) {
      return new Promise(resolve => {
        setTimeout(resolve, ms);
      })
    }

    async function asyncPrint(value, ms) {
      console.log('函数执行', new Date().toTimeString());
      await timeout(ms);
      console.log('延时时间', new Date().toTimeString());
      console.log(value);
    }

    console.log(asyncPrint('hello async', 2000));

    // await
    async function awaitTest() {
      let result = await Promise.resolve('执行成功');
      console.log(result);
      let result2 = await Promise.reject('执行失败');
      console.log(result2);
      let result3 = await Promise.resolve('还想执行一次');// 执行不了
      console.log(result3);
    }
    awaitTest();


    // 案例演示
    async function sendXml(url) {
      return new Promise((resolve, reject) => {
        $.ajax({
          url,
          type: 'GET',
          success: data =>  resolve(data),
          error: error => reject(error)
        })
      })
    }
```

```
    async function getNews(url) {
      let result = await sendXml(url);
      let result2 = await sendXml(url);
      console.log(result, result2);
    }
    getNews('http://localhost:3000/news?id=2')

</script>

</body>
</html>
```

Module模块

ES6模块化语法将在后面JS模块化章节讲解。

\1. 通过export 关键字暴露模块；

\2. 通过import关键字引入模块。

其它

字符串扩展

给字符串扩展了几个方法：

\1. includes(str) : 判断是否包含指定的字符串；

\2. startsWith(str) : 判断是否以指定字符串开头；

\3. endsWith(str) : 判断是否以指定字符串结尾；

\4. repeat(count) : 重复指定次数。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>01_字符串扩展</title>
</head>
<body>
<script type="text/javascript">

    let str = 'abcdefg';
    console.log(str.includes('a'));//true
    console.log(str.includes('h'));//false

    //startsWith(str) : 判断是否以指定字符串开头
    console.log(str.startsWith('a'));//true
    console.log(str.startsWith('d'));//false
    //endsWith(str) : 判断是否以指定字符串结尾
    console.log(str.endsWith('g'));//true
    console.log(str.endsWith('d'));//false
    //repeat(count) : 重复指定次数a
    console.log(str.repeat(5));
</script>
```

```
</body>
</html>
```

数值扩展

\1. 二进制与八进制数值表示法: 二进制用0b, 八进制用0o;

\2. Number.isFinite(i) : 判断是否是有限大的数;

\3. Number.isNaN(i) : 判断是否是NaN;

\4. Number.isInteger(i) : 判断是否是整数;

\5. Number.parseInt(str) : 将字符串转换为对应的数值;

\6. Math.trunc(i) : 直接去除小数部分。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>02_数值扩展</title>
</head>
<body>
<script type="text/javascript">
    console.log(0b1010);//10
    console.log(0o56);//46
    //Number.isFinite(i) : 判断是否是有限大的数
    console.log(Number.isFinite(NaN));//false
    console.log(Number.isFinite(5));//true
    //Number.isNaN(i) : 判断是否是NaN
    console.log(Number.isNaN(NaN));//true
    console.log(Number.isNaN(5));//falsse

    //Number.isInteger(i) : 判断是否是整数
    console.log(Number.isInteger(5.23));//false
    console.log(Number.isInteger(5.0));//true
    console.log(Number.isInteger(5));//true

    //Number.parseInt(str) : 将字符串转换为对应的数值
    console.log(Number.parseInt('123abc'));//123
    console.log(Number.parseInt('a123abc'));//NaN

    // Math.trunc(i) : 直接去除小数部分
    console.log(Math.trunc(13.123));//13

</script>
</body>
</html>
```

数组扩展

\1. Array.from(v) : 将伪数组对象或可遍历对象转换为真数组。

\2. Array.of(v1, v2, v3)： 将一系列值转换成数组。

\3. find(function(value, index, arr){return true})： 找出第一个满足条件返回true的元素。

\4. findIndex(function(value, index, arr){return true})： 找出第一个满足条件返回true的元素下标。

\5. keys()： 返回包含所有下标的可迭代对象。

\6. values()： 返回包含所有值的可迭代对象。

\7. entries()： 返回包含所有下标和值的可迭代对象。

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>03_数组扩展</title>
</head>
<body>
<button>测试1</button>
<br>
<button>测试2</button>
<br>
<button>测试3</button>
<br>
<script type="text/javascript">
    //Array.from(v) : 将伪数组对象或可遍历对象转换为真数组
    let btns = document.getElementsByTagName('button');
    console.log(btns.length);//3
    Array.from(btns).forEach(function (item, index) {
        console.log(item, index);
    });
    //Array.of(v1, v2, v3) : 将一系列值转换成数组
    let arr = Array.of(1, 'abc', true);
    console.log(arr);
    //find(function(value, index, arr){return true}) : 找出第一个满足条件返回true的
元素
    let arr1 = [1,3,5,2,6,7,3];
    let result = arr1.find(function (item, index) {
        return item >3
    });
    console.log(result);//5
    //findIndex(function(value, index, arr){return true}) : 找出第一个满足条件返回
true的元素下标
    let result1 = arr1.findIndex(function (item, index) {
        return item >3
    });
    console.log(result1);//2
</script>
</body>

</html>
```

对象扩展

\1. Object.is(v1, v2)： 判断2个数据是否完全相等。

\2. Object.assign(target, source1, source2..)： 将源对象的属性复制到目标对象上。

\3. 显式操作**proto**属性：

var obj2 = {};

obj2.**proto** = obj1;

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>04_对象扩展</title>
</head>
<body>
<script type="text/javascript">

    console.log(Object.is('abc', 'abc'));//true
    console.log(NaN == NaN);//false
    console.log(Object.is(NaN, NaN));//true

    console.log(0 == -0);//true
    console.log(Object.is(0, -0));//false

    //Object.assign(target, source1, source2..)
    let obj = {name : 'kobe', age : 39, c: {d: 2}};
    let obj1 = {};
    Object.assign(obj1, obj);
    console.log(obj1, obj1.name);

    //直接操作 __proto__ 属性
    let obj3 = {name : 'anverson', age : 41};
    let obj4 = {};
    obj4.__proto__ = obj3;
    console.log(obj4, obj4.name, obj4.age);
</script>
</body>

</html>
```

克隆函数

1、数据类型： 数据分为基本的数据类型(String, Number, boolean, Null, Undefined)和对象数据类型。

- 基本数据类型：

特点： 存储的是该对象的实际数据

- 对象数据类型：

特点： 存储的是该对象在栈中引用，真实的数据存放在堆内存里

2、复制数据

- 基本数据类型存放的就是实际的数据，可直接复制

let number2 = 2;

let number1 = number2;

- 克隆数据：对象/数组

1、区别： 浅拷贝/深度拷贝

判断： 拷贝是否产生了新的数据还是拷贝的是数据的引用

知识点：对象数据存放的是对象在栈内存的引用，直接复制的是对象的引用

let obj = {username: 'kobe'}

let obj1 = obj; // obj1 复制了obj在栈内存的引用

2、常用的拷贝技术

1). arr.concat(): 数组浅拷贝

2). arr.slice(): 数组浅拷贝

3). JSON.parse(JSON.stringify(arr/obj)): 数组或对象深拷贝, 但不能处理函数数据

4). 浅拷贝包含函数数据的对象/数组

5). 深拷贝包含函数数据的对象/数组

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>对象的深度克隆</title>
</head>
<body>
<script type="text/javascript">
  // 复制的对象的方式
  // 浅度复制
  let obj = {username: 'kobe', age: 39, sex: {option1: '男', option2: '女'}};
  let obj1 = obj;
  console.log(obj1);
  obj1.sex.option1 = '不男不女'; // 修改复制的对象会影响原对象
  console.log(obj1, obj);

  console.log('-----------');
  // Object.assign();  浅复制
  let obj2 = {};
  Object.assign(obj2, obj);
  console.log(obj2);
  obj2.sex.option1 = '男'; // 修改复制的对象会影响原对象
  console.log(obj2, obj);

  // 深度克隆(复制)

  function getObjClass(obj) {
    let result = Object.prototype.toString.call(obj).slice(8, -1);
    if(result === 'Null'){
      return 'Null';
    }else if(result === 'Undefined'){
      return 'Undefined';
    }else {
      return result;
    }
  }
  // for in 遍历数组的时候遍历的是下标
```

```
  let testArr = [1,2,3,4];
  for(let i in testArr){
    console.log(i); // 对应的下标索引
  }

  // 深度克隆
  function deepClone(obj) {
    let result, objClass = getObjClass(obj);
    if(objClass === 'Object'){
      result = {};
    }else if(objClass === 'Array'){
      result = [];
    }else {
      return obj; // 如果是其他数据类型不复制，直接将数据返回
    }
    // 遍历目标对象
    for(let key in obj){
      let value = obj[key];
      if(getObjClass(value) === "Object" || 'Array'){
        result[key] = deepClone(value);
      }else {
        result[key] = obj[key];
      }
    }
    return result;
  }


  let obj3 = {username: 'kobe',age: 39, sex: {option1: '男', option2: '女'}};
  let obj4 = deepClone(obj3);
  console.log(obj4);
  obj4.sex.option1 = '不男不女'; // 修改复制后的对象不会影响原对象
  console.log(obj4, obj3);


</script>
</body>
</html>
```

Set和Map数据结构

\1. Set容器 : 无序不可重复的多个value的集合体。

1). Set()

2). Set(array)

3). add(value)

4). delete(value)

5). has(value)

6). clear()

7). size

\2. Map容器 : 无序key不重复的多个key-value的集合体。

1). Map()

2). Map(array)

3). set(key, value)

4). get(key)

5). delete(key)

6). has(key)

7). clear()

8). size

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>05_Set和Map数据结构</title>
</head>
<body>
<script type="text/javascript">

    let set = new Set([1,2,3,4,3,2,1,6]);
    console.log(set);
    set.add('abc');
    console.log(set, set.size);
    //delete(value)
    set.delete(2);
    console.log(set);
    //has(value)
    console.log(set.has(2));//false
    console.log(set.has(1));//true
    //clear()
    set.clear();
    console.log(set);

    let map = new Map([['abc', 12],[25, 'age']]);
    console.log(map);
    map.set('男', '性别');
    console.log(map);
    console.log(map.get(25));//age
    //delete(key)
    map.delete('男');
    console.log(map);
    console.log(map.has('男'));//false
    console.log(map.has('abc'));//true
    map.clear();
    console.log(map);
</script>
</body>

</html>
```

for...of循环

for(let value of target){}循环遍历

\1. 遍历数组

\2. 遍历Set

\3. 遍历Map

\4. 遍历字符串

\5. 遍历伪数组

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>06_for_of循环</title>
</head>
<body>
<button>按钮1</button>
<button>按钮2</button>
<button>按钮3</button>

<script type="text/javascript">

    let arr = [1,2,3,4,5];
    for(let num of arr){
        console.log(num);
    }
    let set = new Set([1,2,3,4,5]);
    for(let num of set){
        console.log(num);
    }
    let str = 'abcdefg';
    for(let num of str){
        console.log(num);
    }
    let btns = document.getElementsByTagName('button');
    for(let btn of btns){
        console.log(btn.innerHTML);
    }

</script>
</body>

</html>
```

ECMAScript7

\1. 指数运算符: **

\2. Array.prototype.includes(value) : 判断

区别方法的2种称谓:

静态(工具)方法：Fun.xxx = function(){}

实例方法

所有实例对象 : Fun.prototype.xxx = function(){} //xxx针对Fun的所有实例对象

某个实例对象 : fun.xxx = function(){} //xxx只是针对fun对象

数组的扩展、运算符扩展、await异步函数

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<!--
1. 指数运算符(幂): **
2. Array.prototype.includes(value) : 判断数组中是否包含指定value

-->
<script type="text/javascript">
    console.log(3 ** 3);//27
    let arr = [1,2,3,4, 'abc'];
    console.log(arr.includes(2));//true
    console.log(arr.includes(5));//false
</script>
</body>
</html>
```