[笔记][ES567]

JavaScript

---

01. ECMAScript入门介绍

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>ECMAScript理解</title>
    </head>
    <body>
        <button id="testBtn">测试</button>

        <!--
1. 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
2. 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
3. JS包含三个部分：
    1). ECMAScript（核心）
    2). 扩展==>浏览器端
        * BOM（浏览器对象模型）
        * DOM（文档对象模型）
    3). 扩展==>服务器端
        * Node
4. 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）
-->
        <script type="text/javascript"></script>
    </body>
```

```
</html>
```

02. ES5-严格模式

可以看看这里：https://www.runoob.com/js/js-strict.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>01_严格模式</title>
    </head>
    <body>
        <!--
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, 为代码的安全运行保驾护航
    * 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
    * 禁止使用 delete 删除变量、对象、函数
    * 禁止自定义的函数中的 this 指向 window
    * 创建 eval 作用域, 禁止使用 eval 作为变量名
    * 函数不能有同名形参
    * 禁止使用 arguments 作为变量名
    * 禁止使用0前缀八进制和八进制转义字符串'\070', 需要使用0o和0O前缀与十六进制转义字符串代替
    * 禁止使用 with 语句
    * 新增保留关键字:
```

```
    implements
    interface
    let
    package
    private
    protected
    public
    static
    yield
-->

        <script type="text/javascript">
            "use strict";

            // 报错 Reference Error
            // username = "kobe";
            var username = "kobe";
            console.log(username);

            function Person(name, age) {
                this.name = name;
                this.age = age;
            }

            // Person('kobe', 39);  // 报错
            new Person("kobe", 39);

            var str = "nba";
            eval("var str = 'CBA'; alert(str);"); // 输出
 CBA

            // 不使用严格模式的话，eval没有自己的作用域
            // 在 eval 里面声明的变量会影响到全局
            // 所以下面输出是 CBA
            // 但是如果使用严格模式，下面输出是 nba
            alert(str);

            // 非严格模式中函数可以有同名形参
            // 严格模式会报错
            // function fun(arg1, arg1) {
            //     console.log(arg1);
```

```
            // }
            // fun(1, 4);

            // 禁止使用0前缀八进制和八进制转义字符串
            // var number = 070;
            // var number = "\070";

            // eval 不能作为变量名
            // var eval = 666;
        </script>
    </body>
</html>
```

03. ES5-json对象扩展

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

-->
        <script type="text/javascript">
            var obj = { username: "kobe" };
            obj = JSON.stringify(obj);
            console.log(typeof obj); // "string"
            console.log(obj); // {"username": "kobe"}

            obj = JSON.parse(obj);
```

```
            console.log(typeof obj); // "object"
            console.log(obj); // Object { username: "kob
e" }

            // json 字符串只有两种，要么是 json 对象，要么是 js
on 数组
        </script>
    </body>
</html>
```

04. ES5-Object对象方法扩展

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>03_Object扩展</title>
    </head>
    <body>
        <!--
ES5给Object扩展了一些静态方法，常用的2个：
1. Object.create(prototype, [descriptors])
  * 作用：以指定对象为原型创建新的对象
  * 为新的对象指定新的属性，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)
  * 作用：为指定对象定义扩展多个属性
    * get ：用来获取当前属性值得回调函数
    * set ：修改当前属性值时触发的回调函数，并且实参即为修改后的值
   * 存取器属性：setter,getter一个用来存值，一个用来取值
-->
        <script type="text/javascript">
            var obj = { username: "damu", age: 30 };
```

```
            var obj1 = {};
            obj1 = Object.create(obj, {
                sex: {
                    value: "男",
                    writable: true,
                    configurable: true,
                    enumerable: true
                }
            });

            console.log(obj1.sex);
            obj1.sex = "女";
            console.log(obj1.sex);
            // delete obj1.sex;
            // console.log(obj1);

            console.log("------------");
            for (var i in obj1) {
                console.log(i);
            }

            console.log("*********");
            var obj2 = { firstName: "kobe", lastName: "br
yant" };
            Object.defineProperties(obj2, {
                fullName: {
                    get: function() {
                        // 获取扩展属性值的时候，get方法自动调
用
                        // 它是惰性求值的
                        console.log("get()");
                        return this.firstName + " " + thi
s.lastName;
                    },
                    set: function(data) {
                        // 监听扩展属性，当扩展属性发生变化的时
候会自动调用
                        // 自动调用时会将变化的值作为实参传递给
 set 指定的函数
                        console.log("set()", data);
                        var names = data.split(" ");
```

```
                        this.firstName = names[0];
                        this.lastName = names[1];
                    }
                }
            });
            console.log(obj2.fullName);
            obj2.fullName = "tim duncan";
            console.log(obj2.fullName);
        </script>
    </body>
</html>
```

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

```
            };
            console.log(obj);
            obj.fullName = "kobe bryant";
            console.log(obj.fullName); // kobe bryant
        </script>
    </body>
</html>
```

补充

> 团子注：参考https://segmentfault.com/a/1190000002979437

`Object.create(proto, [propertiesObject])`

- `proto` 原型对象，会作为新创建对象的原型，也就是 `obj.__proto__` 的值。可以是 `null`（此时创建一个干净的对象，所谓干净指的是原型链上没东西），也可以是一个对象值，如果不是，会抛出 `TypeError`。
- `propertiesObject` 属性对象，是一个对象，该对象的属性为新建对象的属性名，属性值为新建对象属性的描述符对象。
- 描述符可以是：`value`、`writable`、`enumerable`、`configurable`、`get`、`set`
- 使用 `set`，`get` 函数的时候，不能和 `value` 属性和 `writable` 属性一起用，会报错

---

05. ES5-数组的扩展

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>04_Array扩展</title>
    </head>
```

```
    <body>
        <!-- 注意：Array.prototype 上面的方法是给 Array 实例使用的！ -->
        <!--
        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
        -->
        <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
            */
            arr = [1, 6, 4, 2, 7, 5, 8, 6];

            // 1. 输出第一个6的下标
            var first6 = arr.indexOf(6);
            console.log(first6); // 1

            // 2. 输出最后一个6的下标
            var last6 = arr.lastIndexOf(6);
            console.log(last6); // 7

            // 3. 输出所有元素的值和下标
            arr.forEach(function(value, index, array) {
                console.log(value, index, array);
            });

            // 4. 根据arr产生一个新数组,要求每个元素都比原来大10
```

```
            var arr1 = arr.map(function(value, index, arr
ay) {
                return value + 10;
            });
            console.log(arr1);

            // 5. 根据arr产生一个新数组，返回的每个元素要大于4
            var arr2 = arr.filter(function(value, index,
 arr) {
                if (value > 4) {
                    return true;
                }
                // 简化写法
                // return value > 4;
            });
            console.log(arr2);
        </script>
    </body>
</html>
```

06. ES5-Function扩展

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>05_Function扩展</title>
    </head>
    <body>
        <!--
        1. Function.prototype.bind(obj) :
            * 作用: 将函数内的this绑定为obj, 并将函数返回
        2. 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
```

```
        -->
        <script type="text/javascript">
            var obj = { username: "kobe" };
            function foo(data) {
                console.log(this, data);
            }
            // call和apply在不传递额外参数的情况下，使用方法是一模一样的
            // foo.apply(obj);
            // foo.call(obj);

            // call 从第二个参数依次传递
            foo.call(obj, 33);
            // apply 第二个参数是一个数组
            foo.apply(obj, [33]);

            // bind 也可以绑定 this
            // 绑定完 this 以后，不会立即调用函数，而是会将绑定了 this 的函数返回
            var bar = foo.bind(obj);
            console.log(bar); // function foo()
            bar();
            // 也可以这么调用
            // foo.bind(obj)();

            // bind 传参的方式，跟 call 一样
            foo.bind(obj)(66);

            // 通常用 bind 来指定回调函数，因为回调函数的本质不是立即调用的
            setTimeout(
                function() {
                    console.log(this);
                }.bind(obj),
                1000
            );
        </script>
    </body>
</html>
```

07. ES6-let和const关键字

```
<!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 />
        <!--
          ***let
          1. 作用:
            * 与var类似，用于声明一个变量
          2. 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
          3. 应用:
            * 循环遍历加监听
            * 使用let取代var是趋势
        -->
        <script type="text/javascript">
            let username = "kobe";
            // let username = "kobe";
            console.log(username);

            // JS 代码在执行前会进行预解析（或者叫预处理）
            // let 声明的变量不会预解析
            console.log(age); // undefined
            var age = 11;

            // console.log(sex);  // 报错：ReferenceError
            // let sex = "男";
```

```
            let btns = document.getElementsByTagName("button");
            // 性能优化：可以在初始化的部分写上 length = btns.length
            for (var i = 0; i < btns.length; i++) {
                var btn = btns[i];
                btn.onclick = function() {
                    alert(i);
                    // 这里输出都是 3
                    // 因为点击事件定义的是回调函数
                    // 回调函数会被放到事件队列里面，等主线程代码执行完才会执行
                    // 才会通过钩子的形式把回调函数拿来执行
                };
            }

            // 解决方案一：闭包
            // 每次循环执行，都产生一个函数的作用域
            for (var i = 0; i < btns.length; i++) {
                var btn = btns[i];
                (function(i) {
                    btn.onclick = function() {
                        alert(i);
                    };
                })(i);
            }

            // 解决方案二：let
            // let 具有块级作用域，在使用 i 的时候，就是属于自己块级作用域的数据
            for (let i = 0; i < btns.length; i++) {
                var btn = btns[i];
                btn.onclick = function() {
                    alert(i);
                };
            }
        </script>
    </body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>02_const关键字</title>
    </head>
    <body>
        <!--
          1. 作用:
            * 定义一个常量
          2. 特点:
            * 不能修改
            * 其它特点同let
          3. 应用:
            * 保存不用改变的数据
        -->
        <script type="text/javascript">
            const KEY = "nba";
            // 下面会报错
            // KEY = 2;
            console.log(KEY);
        </script>
    </body>
</html>
```

08. ES6-变量的解构赋值

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>03_变量的解构赋值</title>
    </head>
    <body>
```

```
<!--
    1. 理解:
    * 从对象或数组中提取数据，并赋值给变量(多个)
    2. 对象的解构赋值
    let {n, a} = {n:'tom', a:12}
    3. 数组的解构赋值
    let [a,b] = [1, 'atguigu'];
    4. 用途
    * 给多个形参赋值
-->
<script type="text/javascript">
    // 解构就是解析结构
    let obj = { username: "kobe", age: 39 };
    // let username = obj.username;
    // let age = obj.age;

    // 左边以对象的结构来接收，对象中的属性相当于定义在全局的变量，并且要和右边的属性名相同
    let { username, age } = obj;
    // "kobe" 39
    console.log(username, age);

    let { u, a } = obj;
    // undefined undefined
    // 这样相当于 obj.u 和 obj.a，都是 undefined
    console.log(u, a);

    // 解构赋值可以不用全部解构出来，只挑选需要的属性

    // ********************
    // 数组解构赋值
    let arr = [1, 3, 5, "abc", true];
    let [x, y, z] = arr;
    // 根据下标来取值
    // 1 3 5
    console.log(x, y, z);
    let [, , , j, k] = arr;
    console.log(j, k); // "abc" true

    // 用途：给多个形参赋值
    /*
```

```
            function foo (obj) {
                console.log(obj.username, obj.age);
            }
            foo(obj);
            */

            function foo({ username, age }) {
                // 相当于:
                // let {username, age} = obj;
                console.log(username, age);
            }
            foo(obj);
        </script>
    </body>
</html>
```

09. ES6-模板字符串

```
<!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 = { username: "kobe", age: 39 };
            let str = "我的名字叫: " + obj.username + ", 我
的年龄是" + obj.age;
            console.log(str);
```

```
            let template_str = `我的名字
叫：${obj.username}，我的年龄是${obj.age}`;
            console.log(template_str);
        </script>
    </body>
</html>
```

10. ES6-对象的简写方式

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>05_简化的对象写法</title>
    </head>
    <body>
        <!--
          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}
            };
        -->
        <script type="text/javascript">
            let username = "kobe";
            let age = 39;
            /*
            let obj = {
                username: username,
                age: age
```

```
            };
            */
            // 对象的属性名和属性值一样，可以简写为
            let obj = {
                username, // 同名的属性可以省略不写
                age,
                /*
                getName: function() {
                    return this.username;
                }
                */
                // 删除：function，简写为
                getName() {
                    // 可以省略函数的冒号 function
                    return this.username;
                }
            };
            console.log(obj);
            console.log(obj.getName());
        </script>
    </body>
</html>
```

11. ES6-箭头函数详解

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>06_箭头函数</title>
    </head>
    <body>
        <button id="btn1">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() {};
            let fun = () => console.log("我是箭头函数");
            // 箭头左边是形参列表，箭头右边是函数体

            // 形参的情况
            // 1. 没有形参，括号不能省略
            let fun1 = () => console.log("我是箭头函数");
            // 2. 只有一个形参，括号可以省略（只要有人在这里占着空位就行）
            let fun2 = a => console.log(a);
            fun2("aaa");
            // 3. 两个及两个以上的形参，小括号不能省略
            let fun3 = (x, y) => console.log(x, y);
            fun3(25, 36);

            // 函数体的情况
            // 1. 函数体只有一条语句或者表达式
            // 可以省略大括号，省略时会自动返回语句执行的结果或者表达式的结果
            let fun4 = () => console.log("我是箭头函数");
            fun4();
```

```
            let fun5 = (x, y) => x + y;
            console.log(fun5(24, 36));
            let fun6 = (x, y) => {
                return x + y;
            };
            console.log(fun6(24, 36));

            // 2. 函数体不止一条语句或者是表达式的情况，大括号不
能省略，省略会报错
            let fun7 = (x, y) => {
                console.log(x, y);
                return x + y;
            };
            let res = fun7(35, 49);
            console.log(res);

            // ****************
            // 测试箭头函数的this
            let btn1 = document.getElementById("btn1");
            let btn2 = document.getElementById("btn2");
            // 常规函数
            btn1.onclick = function() {
                alert(this);
            };
            btn2.onclick = () => alert(this);

            let obj = {
                name: "箭头函数",
                getName() {
                    btn2.onclick = () =>
console.log(this);
                }
            };
            obj.getName(); // 就是刚才定义的对象 obj

            let obj2 = {
                name: "箭头函数2",
                getName: () => {
                    btn2.onclick = () => {
                        console.log(this);
                    };
```

```
                }
            };
            obj2.getName(); // Window
            // 如果按照官方理解：看定义时所处的对象
            obj.getName = () => {
                // ...
            };
            // 这也是在 window 中定义的
        </script>
    </body>
</html>
```

12. ES6-三点运算符

```
<!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 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);
        2. 扩展运算符
            let arr1 = [1,3,5];
            let arr2 = [2,...arr1,6];
            arr2.push(...arr1);
        -->
        <script type="text/javascript">
            function foo(a, b) {
                // arguments 是一个伪数组
                console.log(arguments);

                // callee 是 arguments 的属性
                //  (不能打开注释，否则会递归到死机)
                // arguments.callee(); // arguments.calle
e 指向函数本身，这里是递归

                // 因为 arguments 是一个伪数组，所以没有 forEa
ch 方法，下面会报错
                /*
                arguments.forEach(function(value, index,
 array) {
                    console.log(item, index);
                });
                */
            }
            foo(2, 65);

            function bar(a, ...values) {
                console.log(arguments);
                // 三点运算符收集的实参是一个数组
                console.log(values);
                values.forEach(function(value, index, arr
ay) {
                    console.log(value, index);
                });
                // a 不会被收集，注意三点运算符必须放到最后面
                console.log(a);
            }
            bar(1, 2, 3);
```

```
            // ****************
            let arr = [1, 6];
            let arr1 = [2, 3, 4, 5];
            arr = [1, ...arr1, 6];
            console.log(arr);
        </script>
    </body>
</html>
```

13. ES6-形参默认值

```
<!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">
            // 定义一个点的坐标的构造函数
            function Point(x, y) {
                this.x = x;
                this.y = y;
            }
            let p = new Point(10, 20);
            console.log(p);

            // 忘记传参，全是 undefined
```

```
            let point = new Point();

            function Point2(x = 0, y) {
                this.x = x;
                this.y = y;
            }
            let p2 = new Point2();
            console.log(p2);
        </script>
    </body>
</html>
```

---

14. ES6-promise对象原理详解

> `jQ` 的 `success` 和 `error` 也是有 `promise` 的思想在里面的

---

15. ES6-promise对象案例练习

设计看新闻的网站:

- 应该把新闻和评论分开返回? 还是一起返回?
- 分开返回的用户体验好!

在 `win` 文件管理器的地址栏输入 `cmd` 可以直接在当前文件夹中启动命令行。

`onreadystatechange` 的状态有 `0-4`，总共会调用 `4` 次。
第一次调用，`readystate` 为 `1`。

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

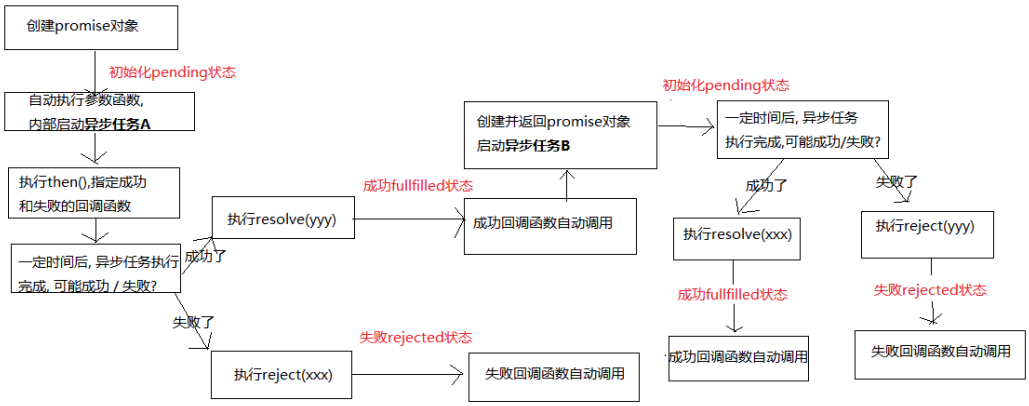
```
    </head>
    <body>
        <!--
            1. 理解:
            * Promise对象: 代表了未来某个将要发生的事件(通常是一个异步操作)
            * 有了promise对象, 可以将异步操作以同步的流程表达出来, 避免了层层嵌套的回调函数(俗称'回调地狱')
            * ES6的Promise是一个构造函数, 用来生成promise实例
            2. 使用promise基本步骤(2步):
            * 创建promise对象
                let promise = new Promise((resolve, reject) => {
                    //初始化promise状态为 pending
                //执行异步操作
                if(异步操作成功) {
                    resolve(value);//修改promise的状态为fullfilled
                } else {
                    reject(errMsg);//修改promise的状态为rejected
                }
                })
            * 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();
            -->
        <script type="text/javascript">
            // 创建 Promise 对象，构造函数里面要写一个 function（也算是一个回调）
            let promise = new Promise((resolve, reject) => {
                // 初始化 promise 状态为 pending（初始化状态）
                console.log("111");
                // 执行异步操作，通常是发送ajax请求、开启定时器
                setTimeout(() => {
                    console.log("333");
                    // 根据异步任务的返回结果来去修改promise的状态
                    // 假如异步任务执行成功，调用 resolve 函数
                    // （可以根据 http 状态码来判断是否成功）
                    // resolve("哈哈，成功了"); // 修改 promise 的状态为 fullfilled：成功的状态

                    // 假如异步任务执行失败，调用 reject 函数
                    reject("糟糕，失败了"); // 修改 promise 的状态为 rejected：失败的状态
                }, 2000);
            });
            // 输出111之后输出222，说明上面是同步代码
            console.log("222");

            promise.then(
                data => {
                    // 成功的回调
                    console.log(data, "成功了");
                },
                error => {
                    // 失败的回调
                    console.log(error, "失败了");
                }
            );

            // 定义获取新闻的功能函数
            // 要发两次请求，第二次请求拿评论受限于第一次
```

```
            function getNews(url) {
                let promise = new Promise((resolve, rejec
t) => {
                    // 状态：初始化
                    // 执行异步任务
                    // 原生AJAX
                    // 创建 xmlHttp 对象
                    let xmlHttp = new XMLHttpRequest();
                    // 刚初始化的时候，readystate 为 0
                    console.log(xmlHttp.readyState);
                    // 绑定监听 readyState
                    /*
                    xmlHttp.onreadystatechange = function
() {
                        if (xmlHttp.readyState === 4 && x
mlHttp.status === 200) {
                            //请求成功
                            console.log(xmlHttp.responseT
ext);
                            // 修改状态
                            resolve(xmlHttp.responseTex
t); // 修改 promise 的状态为成功的状态
                        } else {
                            // 请求失败
                            reject("暂时没有新闻内容");
                        }
                    };
                    */

                    // 修正逻辑
                    xmlHttp.onreadystatechange =
function() {
                        if (xmlHttp.readyState === 4) {
                            if (xmlHttp.status === 200) {
                                //请求成功
                                console.log(xmlHttp.respo
nseText);
                                // 修改状态
                                resolve(xmlHttp.responseT
ext); // 修改 promise 的状态为成功的状态
                            } else {
```

```
                                // 请求失败
                                reject("暂时没有新闻内容");
                            }
                        }
                    };

                    // open 设置请求的方式以及 url
                    // 第三个参数设置为 false，则为同步发，否则默认异步
                    // 基本没有人发同步
                    xmlHttp.open("GET", url);
                    // 发送
                    xmlHttp.send();
                });
                return promise;
            }

            getNews("http://localhost:3000/news?id=2")
                .then(
                    data => {
                        console.log(typeof data); // 使用原生ajax，这里得到的是字符串
                        // 发送请求获取评论内容准备 url
                        let commentsUrl = JSON.parse(data).commentsUrl;
                        let url = `http://localhost:3000${commentsUrl}`;
                        // 发送请求
                        return getNews(url);
                    },
                    error => {
                        console.log(error);
                    }
                )
                .then(
                    data => {
                        console.log(data);
                    },
                    error => {
                        console.log(error);
                    }
```

```
                    );
            </script>
        </body>
    </html>
```



> `promise` 本身其实还是利用回调

---

16. ES6-Symbol属性介绍1

> 老师推荐了阮一峰的ES6入门

`typeof` 检测数据类型，总有有几种?
六种:

- string
- number
- boolean
- undefined
- object（null和array检测出来就是object）
- function

`ES6` 中就是七种，还有一种 `symbol`

`Symbol` 其实是一个函数对象，里面有内置的 `11` 个属性值。
`Symbol` 用来给对象设置唯一的属性。

什么时候用 `Symbol`？

- 比较重要的值
- 标识当前对象身份的值

---

17. ES6-Symbol属性介绍2

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Symbol</title>
    </head>
    <body>
        <!--
          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属性值对应的值是唯一的，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)
            -->

        <script type="text/javascript">
            // Symbol 是新增的原生数据类型
            let symbol = Symbol(); // Symbol 不是构造函数，不能 new
            console.log(symbol); // Symbol()

            let obj = { username: "kobe", age: 39 };
            obj.sex = "男";
            // 不能这么写 obj.symbol = "男";
            // 要使用属性选择器的方式
            obj[symbol] = "hello";
            console.log(obj);
            // for in, for of 不能遍历 symbol 属性
            for (let i in obj) {
                console.log(i);
            }

            let symbol2 = Symbol();
            let symbol3 = Symbol();
            console.log(symbol2 === symbol3); // false
            // 虽然是同一个函数调用产生的，但是不一样
            // 不过 console.log 出来，两个值一模一样，都是 Symbol()
            console.log(symbol2, symbol3);

            // 为了区分可以给 Symbol 函数传入一个唯一标识
            let symbol4 = Symbol("4");
            let symbol5 = Symbol("5");
            console.log(symbol4, symbol5);

            // 可以去定义常量
            const Person_key = Symbol("person_key");
            console.log(Person_key);
        </script>
```

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

---

18. ES6-iterator接口机制1

最后一个返回的对象是 `{value: undefined, done: true}`
其余见下节

---

19. ES6-iterator接口机制2

见下节

---

20. ES6-iterator接口机制3

补充

- 我不太明白类似 `let obj = {[xxx]: 666};` 的用法，看下面的图就知道了


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

```
        <meta charset="UTF-8" />
        <title>Iterator遍历器</title>
    </head>
    <body>
        <!--
    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(遍历器对象)，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
      。。。
  -->

        <script type="text/javascript">
            // 模拟指针对象（遍历器对象）

            function myIterator(arr) {
                // iterator 接口
                let nextIndex = 0; // 记录指针的位置
                return {
                    // 遍历器对象
                    next: function() {
```

```
                        return nextIndex < arr.length
                            ? { value: arr[nextIndex++],
done: false }
                            : { value: undefined, done: t
rue };
                    }
                };
            }

            // 准备一个数据
            let arr = [1, 4, 65, "abc"];
            let iteratorObj = myIterator(arr);
            console.log(iteratorObj.next());
            console.log(iteratorObj.next());
            console.log(iteratorObj.next());
            console.log(iteratorObj.next());
            console.log(iteratorObj.next());

            // 将 iterator 接口部署到指定的数据类型上，此时就可
以使用 for of 去遍历它
            // 数组，字符串，arguments，set容器，map容器
            for (let i of arr) {
                console.log(i);
            }

            let str = "你好hello";
            for (let i of str) {
                console.log(i);
            }

            function func() {
                for (let i of arguments) {
                    // arguments 是个伪数组，它没有一般数组的
 forEach 方法
                    console.log(i);
                }
            }
            func(1, 2, 3, "嘿嘿");

            // 对象上没有部署 iterator 接口
            // 下面报错：TypeError: obj is not iterable
```

```
            let obj = { username: "kobe", age: 39 };
            /*
            for (let i of obj) {
                console.log(i);
            }
            */

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

            // 使用三点运算符，解构赋值，默认去调用了iterator接口
            let arr2 = [1, 6];
            let arr3 = [2, 3, 4, 5];
            arr2 = [1, ...arr3, 6];
            console.log(arr2);

            let [a, b] = arr2;
            console.log(a, b); // 1 2
```

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

---

21. ES6-Generator函数简介1

见下节

---

22. ES6-Generator函数简介2

见下节

---

23. ES6-Generator函数简介3

开启服务器

- `cmd`
- `node bin/www`
- `localhost:3000/news`

> 可以下载一个插件叫做前端助手 `FE`

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Generator函数</title>
    </head>
    <body>
        <!--
```

```
        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。
        -->
        <script type="text/javascript" src="./js/jquery-1.10.1.min.js"></script>
        <script type="text/javascript">
            // 小试牛刀
            function* myGenerator() {
                console.log("开始执行");
                let result = yield "hello";
                console.log(result); // 默认得到的是 undefined，但是可以通过 next 来指定
                console.log("暂停后，再次执行");
                yield "generator";
                console.log("遍历完毕");
                return "返回的结果";
            }
```

```
            let MG = myGenerator(); // 返回一个指针对象，它是一个迭代器/遍历器对象
            console.log(MG);
            console.log(MG.next()); // 这个得到一个对象，value 就是 yield 后面的东西
            console.log(MG.next("改变了result"));
            console.log(MG.next());

            // 对象的symbol.iterator属性  指向遍历器对象
            let obj = { username: "kobe", age: 39 };
            // 会报错
            /*
            for (let i of obj) {
                console.log(i);
            }
            */
            // 人为给对象部署一个 iterator 接口
            obj[Symbol.iterator] = function* myTest() {
                yield 1;
                yield 2;
                yield 3;
            };
            for (let i of obj) {
                console.log(i);
            }

            // 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 getNews(data) {
                $.get(url, function(data) {
                    // 发送请求成功时的回调函数
                    console.log(data);
```

```
                    let url = "http://localhost:3000" + d
ata.commentsUrl;
                    SX.next(url);
                });
            }

            // getNews("http://localhost:3000/news?id=
3");

            function* sendXML() {
                let url = yield getNews("http://localhos
t:3000/news?id=3");
                yield getNews(url);
            }
            // 获取遍历器对象
            let SX = sendXML();
            SX.next();
        </script>
    </body>
</html>
```

其实 `Generator` 最终还是利用了回调函数。

---

24. ES8/ES2017-async函数详解及应用1

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>async函数</title>
    </head>
    <body>
        <!--
        async函数(源自ES2017)
        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
        -->
        <script type="text/javascript" src="./js/jquery-1.10.1.min.js"></script>
        <script type="text/javascript">
            // async 基本使用
            async function foo() {
                // 只用 resolve，不用 reject，可以省略一个小括号
                return new Promise(resolve => {
                    /*
                    setTimeout(function() {
                        resolve();
                    }, 2000)
                    */

                    // 和上面的一模一样，更简洁
                    setTimeout(resolve, 2000);
                });
            }

            async function test() {
                console.log("开始执行", new Date().toTimeString());
                await foo(); // await 等待foo() 执行完毕
                console.log("执行完毕", new Date().toTimeString());
```

```
        }

        // async 函数不用 next，next，一调用就直接执行了
        test();

        // async 里面 await 的返回值
        function test2() {
            return "xxx";
        }
        async function asyncPrint() {
            let result = await test2();
            console.log(result);
        }
        // 如果是普通函数，await 得到的返回值就是函数执行的返回值
        asyncPrint();

        async function asyncPrint2() {
            // Promise 简写方法，生成一个promise对象，并且是成功的状态
            // let result = await Promise.resolve(); // 默认是 undefined
            let result = await Promise.resolve("aaa"); // 这就是 aaa
            console.log(result);
            result = await Promise.reject("失败啊！"); // 这里会异常，异常信息是 失败啊!
            console.log(result);
        }
        asyncPrint2();

        // 获取新闻内容
        async function getNews(url) {
            return new Promise((resolve, reject) => {
                $.ajax({
                    method: "GET",
                    // 同名属性省略不写
                    // 等同于：url: url
                    url,
                    /*
                    success: function(data) {
```

```
                            resolve();
                        },
                        */
                        // 简化版
                        success: data => resolve(data),
                        // error: error => reject()
                        error: function(error) {
                            reject();
                        }
                    });
                });
            }

            async function sendXml() {
                let result = await getNews("http://localh
ost:3000/news?id=7");
                console.log(result);
                result = await getNews("http://localhost:
3000" + result.commentsUrl);
                console.log(result);
            }
            sendXml();
        </script>
    </body>
</html>
```

25. ES8/ES2017-async函数技巧

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>async函数</title>
    </head>
    <body>
        <!--
```

```
        async函数(源自ES2017)
        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
        -->
        <script type="text/javascript" src="./js/jquery-1.10.1.min.js"></script>
        <script type="text/javascript">
            // async 基本使用
            async function foo() {
                // 只用 resolve，不用 reject，可以省略一个小括号
                return new Promise(resolve => {
                    /*
                    setTimeout(function() {
                        resolve();
                    }, 2000)
                    */

                    // 和上面的一模一样，更简洁
                    setTimeout(resolve, 2000);
                });
            }

            async function test() {
                console.log("开始执行", new Date().toTimeString());
                await foo(); // await 等待foo() 执行完毕
```

```
            console.log("执行完毕", new Date().toTimeSt
ring());
        }

        // async 函数不用 next，next，一调用就直接执行了
        test();

        // async 里面 await 的返回值
        function test2() {
            return "xxx";
        }
        async function asyncPrint() {
            let result = await test2();
            console.log(result);
        }
        // 如果是普通函数，await 得到的返回值就是函数执行的返
回值
        asyncPrint();

        async function asyncPrint2() {
            // Promise 简写方法，生成一个promise对象，并且
是成功的状态
            // let result = await Promise.resolve();
 // 默认是 undefined
            let result = await
Promise.resolve("aaa"); // 这就是 aaa
            console.log(result);
            result = await Promise.reject("失败啊！");
// 这里会异常，异常信息是 失败啊!
            console.log(result);
        }
        asyncPrint2();

        // 获取新闻内容
        async function getNews(url) {
            return new Promise((resolve, reject) => {
                $.ajax({
                    method: "GET",
                    // 同名属性省略不写
                    // 等同于：url: url
                    url,
```

```
                        /*
                        success: function(data) {
                            resolve();
                        },
                        */
                        // 简化版
                        success: data => resolve(data),
                        // error: error => reject()
                        error: function(error) {
                            // 这里不用 reject(), 而使用 resolve(), 保证下面代码顺利执行
                            // reject();
                            resolve(false);
                        }
                    });
                });
            }

            async function sendXml() {
                let result = await getNews("http://localhost:3000/news?id=7");
                console.log(result);
                if (!result) {
                    alert("暂时没有新闻推送");
                    return;
                }
                result = await getNews("http://localhost:3000" + result.commentsUrl);
                console.log(result);
            }
            sendXml();
        </script>
    </body>
</html>
```

26. ES6-Class类使用详解

```
<!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">
        /*
        function Person(name, age) {
            this.name = name;
            this.age = age;
        }
        let person = new Person("kobe", 39);
        console.log(person);
        */

        // 定义一个人物的类
        class Person {
            // 类的构造方法
            constructor(name, age) {
                this.name = name;
                this.age = age;
            }

            // 类的一般方法
            // 这个方法在实例对象的 __proto__ 上
            // 相当于: Person.prototye.showName = function
 () {...};
            showName() {
                console.log("调用父类的方法");
                console.log(this.name, this.age);
            }
```

```
            // 如何实现 Person.test = function () {console.
log("test");};
            // (这样添加的方法实例是不能使用的)
            // 类的静态方法
            static test = function() {
                console.log("类的静态方法");
            };
            // 简化写法
            /*
            static test() {
                console.log("简化类的静态方法写法");
            }
            */
            static attr = "类的静态属性";
        }

        let person = new Person("kobe", 39);
        console.log(person);
        person.showName();

        console.log(Person.attr);
        Person.test();

        // 子类
        // 如何继承? 子类的原型=父类的实例
        // 等同于 ChildPerson.prototype = new Person();
        class StarPerson extends Person {
            constructor(name, age, salary) {
                super(name, age); // 调用父类的构造方法, 记得
传入相应参数
                this.salary = salary;
            }
            // 重写父类方法
            showName() {
                console.log("调用子类的方法");
                console.log(this.name, this.age, this.sal
ary);
            }
        }
```

```
            let p1 = new StarPerson("wade", 36, 1000000);
            console.log(p1);
            // p1.__proto__.__proto__ 里面有 showName() 方法
            p1.showName();
        </script>
    </html>
```

27. ES6-字符串、数组的扩展

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>01_字符串扩展</title>
    </head>
    <body>
        <!--
            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
        -->
        <script type="text/javascript">
            let str = "asdals;dkjf";
            console.log(str.includes("t")); // false
            console.log(str.includes("a")); // true

            console.log(str.startsWith("a")); // true
            console.log(str.startsWith("asda")); // true

            console.log(str.endsWith("f")); // true

            console.log(str.repeat(5));
        </script>
    </body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>02_数值扩展</title>
    </head>
    <body>
        <!--
            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
        -->
        <script type="text/javascript">
            console.log(0b1010);
            console.log(0o56);

            console.log(Number.isFinite(123)); // true
            console.log(Number.isFinite(Infinity)); // fa
lse

            console.log(Number.isNaN(NaN)); // true
            // 这个居然是true
            console.log(Number.isInteger(0.0)); // true
            console.log(Number.parseInt("123abc")); // 12
3

            console.log(Math.trunc(3.67)); // 3
        </script>
    </body>
</html>
```

28. 数组方法的扩展

```
<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 />

        <!--
        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 t
rue}) : 找出第一个满足条件返回true的元素下标
        -->
        <script type="text/javascript">
            let btns = document.getElementsByTagName("but
ton");
            // btns 是一个伪数组，没有数组的 forEach 方法
            // 下面报错：TypeError: btns.forEach is not a f
unction
            /*
            btns.forEach(function(item, index, array) {
                console.log(item);
            });
            */

            Array.from(btns).forEach((item, index) => {
                console.log(item);
            });
```

```
            let arr = Array.of(1, "lala", true);
            console.log(arr);

            let arr2 = [2, 3, 4, 2, 5, 7, 3, 6, 5];
            // 和 map 一样，都是声明式的方法
            let result = arr2.find(function(item, index,
 array) {
                return item > 4;
            });
            console.log(result);

            let result2 = arr2.findIndex(function(item, i
ndex, array) {
                return item > 4;
            });
            console.log(result2);
        </script>
    </body>
</html>
```

29. ES6-对象方法的扩展

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>04_对象扩展</title>
    </head>
    <body>
        <!--
        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. 直接操作 __proto__ 属性
        let obj2 = {};
```

```
        obj2.__proto__ = obj1;
        -->

        <script type="text/javascript">
            console.log(0 === -0); // true
            console.log(0 == -0); // true
            console.log(NaN == NaN); // false
            console.log(Object.is(0, -0)); // false
            console.log(Object.is(NaN, NaN)); // true
            // Object.is 是以字符串的形式来判断的

            let obj = {};
            let obj1 = { username: "anverson", age: 42 };
            let obj2 = { sex: "男", age: 30 }; // 30 覆盖了
 42
            Object.assign(obj, obj1, obj2);
            console.log(obj);

            let obj3 = {};
            let obj4 = { money: 5000 };
            obj3.__proto__ = obj4;
            console.log(obj3);
            console.log(obj3.money);
        </script>
    </body>
</html>
```

30. ES6-深度克隆1

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>对象的深度克隆</title>
    </head>
    <body>
```

```
        <!--
        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
        -->
        <script type="text/javascript">
            let str = "abcd";
            let str2 = str;
            console.log(str2);
            str2 = "";
            console.log(str); // abcd

            let bool1 = true;
            let bool2 = bool1;
            bool2 = false;
            console.log(bool1); // true
```

```
            let obj = { username: "kobe", age: 39 };
            let obj1 = obj;
            console.log(obj1);
            obj1.username = "wade";
            console.log(obj.username); // wade

            // 拷贝数组、对象没有生成新的数据，而是复制了一份引用
            let arr = [1, 4, { username: "kobe", age: 39
}];
            let arr2 = arr;
            arr2[0] = "abcd";
            console.log(arr, arr2);

            /*
                拷贝数据：
                    基本数据类型：
                        拷贝会生成一份新的数据，修改拷贝以后的数
据不会影响原数据
                    对象、数组
                        拷贝后不会生成新的数据，而是拷贝引用。修
改拷贝以后的数据会影响原来的数据。
                拷贝数据的方法：
                    1. 直接赋值给一个变量  // 浅拷贝
                    2. Object.assign()  // 浅拷贝
                    3. Array.prototype.concat()  // 浅拷贝
                    4. Array.prototype.slice()  // 浅拷贝
                    5. JSON.parse(JSON.stringify())  // 深
拷贝（深度克隆），拷贝的数据里不能有函数，处理不了。
                浅拷贝（针对对象和数组来说的，因为基本数据类型根本
没有深浅之分）：
                    特点：拷贝的是引用，修改拷贝以后的数据会影响
原数据，使得原数据不安全
                深拷贝（深度克隆）：
                    特点：拷贝的时候生成新数据，修改拷贝以后的数
据不会影响原数据
            */

            let obj2 = { username: "kobe" };
            let obj3 = Object.assign(obj2);
            console.log(obj2);
```

```
            obj3.username = "wade";
            console.log(obj2);

            let arr3 = [1, 3, { username: "kobe" }];
            let testArr = [2, 4];
            // let arr4 = arr3.concat(testArr);
            let arr4 = arr3.concat(); // 不传参数，则复制了一
份
            console.log(arr4);
            arr4[1] = "abcd";
            console.log(arr3);
            arr4[2].username = "改变了";
            console.log(arr3);

            let arr5 = arr3.slice();
            arr5[2].username = "arr5";
            console.log(arr3);

            let arr6 = JSON.parse(JSON.stringify(arr3));
            console.log(arr6);
            arr6[2].username = "改变不了";
            console.log(arr3);
        </script>
    </body>
</html>
```

31. ES6-深度克隆2-自己实现深度克隆

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-widt
h, initial-scale=1.0" />
        <meta http-equiv="X-UA-Compatible" content="ie=ed
ge" />
        <title>Document</title>
```

```
    </head>
    <body>
        <script>
            /*
                思考：如何实现深度拷贝（克隆）？
                拷贝的数据里有对象、数组才会发生浅拷贝
                所以拷贝的数据里不能有对象、数组，如果有，需要继续遍历对象、数组，拿到其中每一项的值，
                直到拿到的都是基本数据类型，然后再去赋值，此时就是深拷贝（深度克隆）。
            */

            // 知识点储备
            /*
                如何判断数据类型：
                1. typeof返回的数据类型：string, number ,boolean, undefined, object, function, symbol
                2. Object.prototype.toString.call(obj)
                // call 的特点是指定完 this 立即调用
                // 通过 Object 原型找 toString 方法是因为它是最原始的，没有人修改过它
            */
            let result = "abcd";
            console.log(Object.prototype.toString.call(result)); // [object String]
            result = null;
            console.log(Object.prototype.toString.call(result)); // [object Null]
            result = [];
            console.log(Object.prototype.toString.call(result)); // [object Array]
            result = {};
            console.log(Object.prototype.toString.call(result)); // [object Object]
            result = function(data) {
                console.log(data);
            };
            console.log(Object.prototype.toString.call(result)); // [object Function]
            result = () => {};
```

```
            console.log(Object.prototype.toString.call(re
sult)); // [object Function]

            console.log(typeof Object.prototype.toString.
call(result)); // string

            console.log(Object.prototype.toString.call(re
sult).slice(8, -1)); // Function

            // 知识点储备
            // for in 循环一个对象，枚举出来的是属性名（键）
            // 对于数组枚举出来的是下标
            let obj = { username: "kobe", age: 39 };
            for (let i in obj) {
                console.log(i); // 找到的是键
            }
            let arr = [1, 3, "abc"];
            for (let i in arr) {
                console.log(i); // 找到的是下标
            }

            // 定义检测数据类型的功能
            function checkType(target) {
                return Object.prototype.toString.call(tar
get).slice(8, -1);
            }
            console.log(checkType(result)); // Function

            // 实现深度克隆 --> 针对对象和数组
            function clone(data) {
                // 判断拷贝的数据类型
                let result, // 初始化变量 result，代表最终克
隆出来的数据
                    targetType = checkType(data);
                if (targetType === "Object") {
                    result = {};
                } else if (targetType === "Array") {
                    result = [];
                } else {
                    return data;
                }
```

```
                // 遍历数据
                for (let i in data) {
                    // 获取遍历数据结构的每一项值（同时适用于数组和对象）
                    let value = data[i];
                    // 判断每一项值是否存在对象或数组，如果存在，需要递归
                    if (checkType(value) === "Object" || checkType(value) === "Array") {
                        // 继续遍历获取到的 value 值
                        result[i] = clone(value);
                    } else {
                        // 获取到的value值是基本数据类型或者函数
                        result[i] = value;
                    }
                }

                return result;
            }

            let arr1 = [1, 2, { username: "kobe", age: 39 }, [3, 4]];
            let arr2 = clone(arr1);
            console.log(arr2);
            arr2[2].username = "xiaoming";
            arr2[3][1] = "hahaha";
            console.log(arr1, arr2);
        </script>
    </body>
</html>
```

32. ES6-Set容器和Map容器详解

```
<!DOCTYPE html>
```

```
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>05_Set和Map数据结构</title>
    </head>
    <body>
        <!--
        1. Set容器 : 无序不可重复的多个value的集合体
            * Set()
            * Set(array)
            * add(value)
            * delete(value)
            * has(value)
            * clear()
            * size
        2. Map容器 : 无序的 key不重复的多个key-value的集合体
            * Map()
            * Map(array)  // 这个是二维数组
            * set(key, value)//添加
            * get(key)
            * delete(key)
            * has(key)
            * clear()
            * size
        -->

        <script type="text/javascript">
            // 团子注：Set容器类似 Python 里面的集合
            let set = new Set([1, 1, 2, 2, 3, 5, 8]);
            console.log(set);

            set.add(7);
            // set.size 等同于数组的 length
            console.log(set.size, set);
            console.log(set.has(8), set.has(666)); // tru
e false

            set.clear();
            console.log(set.size);

            // 团子注：Map就是映射，对应于 Python 里面的字典
```

```
            let map = new Map([["username", "aaa"]]);
            console.log(map);
            console.log(map.size);
            map.set("age", 36);
            map.set(78, "haha");
            console.log(map);
            map.delete(78);
            console.log(map);
        </script>
    </body>
</html>
```

---

33. ES6-for-of循环

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>06_for_of循环</title>
    </head>
    <body>
        <!--
            for(let value of target){}循环遍历
            // for of 之所以能遍历，是因为被遍历对象实现了 iterator 接口
            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（arguments）
        -->

        <button>按钮1</button>
        <button>按钮2</button>
        <button>按钮3</button>
```

```
        <script type="text/javascript">
            let set = new Set([1, 2, 4, 5, 5, 6]);
            for (let i of set) {
                console.log(i);
            }

            // Set 的作用：为数组去重
            let arr = [1, 2, 2, 3];
            let arr1 = arr;
            arr = [];
            let set1 = new Set(arr1);
            for (let i of set1) {
                arr.push(i);
            }
            console.log(arr);
        </script>
    </body>
</html>
```

34. ES7-方法介绍

```
<!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);
```

```
            // ES6 当中字符串有 includes, ES7 中数组也有了
            let arr = [1, 2, "abc"];
            console.log(arr.includes("a")); // false
        </script>
    </body>
</html>
```

完成于 `2019.9.3`