# ES5基础篇

# JS各种循环

*var* Arr = [1,2,3,4,5,6,7,8,9]

        // for 循环

        for(*var* i=0;i<Arr.length;i++){

            this.$refs.ulliref.innerHTML += `<li>${Arr[i]}</li>`

        }

        // for...in 循环

        for(*var* i in Arr){

            this.$refs.ulliref.innerHTML += `<li>${Arr[i]}</li>`

        }

        // for...of 循环

        for(*var* value of Arr){

            this.$refs.ulliref.innerHTML += `<li>${value}</li>`

        }

        // forEach 循环

        Arr.forEach(*function*(*v*){

            console.log(v);

        });

        // while 循环

*var* total=0;

*var* i=1;

        while(i<=100){

            total += i;

            i++;

        }

        // do…while 循环

*var* total=0;

*var* i=1;

        do{

            total += i;

            i++;

        }

        while(i<=100)

        // swich 循环

*var* day;

        switch (new Date().getDay()) {

        case 0:

            day = "周日";

            break;

        case 1:

            day = "周一";

            break;

        case 2:

            day = "周二";

            break;

        case 3:

            day = "周三";

            break;

        case 4:

            day = "周四";

            break;

        case 5:

            day = "周五";

            break;

        case  6:

            day = "周六";

        }

        this.$refs.ulliref.innerHTML = "今天是" + day;

# JavaScript 异步编程

### 异步的概念

**异步（Asynchronous, async）是与同步（Synchronous, sync）相对的概念。**

在我们学习的传统单线程编程中，程序的运行是同步的**（同步不意味着所有步骤同时运行，而是指步骤在一个控制流序列中按顺序执行）**。而异步的概念则是不保证同步的概念，也就是说，**一个异步过程的执行将不再与原有的序列有顺序关系。**

简单来理解就是：**同步按你的代码顺序执行，异步不按照代码顺序执行，异步的执行效率更高。**

以上是关于异步的概念的解释，接下来我们通俗地解释一下异步：**异步就是从主线程发射一个子线程来完成任务。**

# ES5封装数据请求

"use strict";//严格模式

// /\*

*function* DataOperation(*URL*,*functionSuccess*,*functionFalse*){

    // var URL = URL + '?' + Math.random();//取消缓存

    if(window.*XMLHttpRequest*){

*var* xhr = new *XMLHttpRequest*();

    } else {

*var* xhr = new *ActiveXObject*();

    }

*var* xhr = new *XMLHttpRequest*();

        xhr.open('POST',*URL*,true);

        xhr.send(null);

        xhr.onreadystatechange = *function*(){

            if(xhr.readyState == 4){

                if(xhr.status == 200){

                    functionSuccess(xhr.responseText);

                }else{

                    functionFalse(xhr.status);

                }

            }

        }

    }

# ES6封装数据请求

*const* getJson =*function*(*url*){//创建数据请求函数

                return new Promise((*resolve*,*reject*)*=>*{//在数据请求函数中返回Promise对象

*const* xhr = new XMLHttpRequest();// 在对象中用原生JS封装发送请求

                    xhr.open('GET',*url*)//指定请求方式和接口

                    xhr.onreadystatechange = handler;//封装状态改变函数

                    xhr.responseType = 'json';//指定返回数据的类型

                    xhr.setRequestHeader('Accept','application/json');//设置请求头

                    xhr.send();//发送数据到后台

*function* handler(){//封装状态改变函数

                        if(this.readyState === 4){

                            if(this.status === 200){

*resolve*(this.response)

                            }else{

*reject*(new Error(this.statusText));

                            }

                        }

                    }

                })

            }

            // 调用封装好的函数

*var* url = 'http://jindianwl.com/weixin/index/test\_api.html';

            getJson(url).then((*data*)*=>*{

                console.log(*data*);

            }).catch((*error*)*=>*{

                console.log(*error*);

            })

            // getJson(url).then((data)=>{

            //  console.log(data);

            // },(error)=>{

            //  console.log(error);

            // })

# JS输出

JavaScript 可以通过不同的方式来输出数据：

* 使用 **window.alert()** 弹出警告框。
* 使用 **document.write()** 方法将内容写到 HTML 文档中。
* 使用 **innerHTML** 写入到 HTML 元素。
* 使用 **console.log()** 写入到浏览器的控制台。

# JavaScript 数据类型

* **值类型(基本类型)**：字符串（String）、数字(Number)、布尔(Boolean)、对空（Null）、未定义（Undefined）、Symbol。
* **引用数据类型**：对象(Object)、数组(Array)、函数(Function)。

# 常见的HTML事件

下面是一些常见的HTML事件的列表:

|  |  |
| --- | --- |
| **事件** | **描述** |
| onchange | HTML 元素改变 |
| onclick | 用户点击 HTML 元素 |
| onmouseover | 用户在一个HTML元素上移动鼠标 |
| onmouseout | 用户从一个HTML元素上移开鼠标 |
| onkeydown | 用户按下键盘按键 |
| onload | 浏览器已完成页面的加载 |

# 常见语句

break 语句跳出循环后，会继续执行该循环之后的代码（如果有的话）

**continue 语句**中断循环中的迭代，如果出现了指定的条件，然后继续循环中的下一个迭代

# typeof 操作符

你可以使用 typeof 操作符来检测变量的数据类型。

# null

在 JavaScript 中 null 表示 "什么都没有"。

null是一个只有一个值的特殊类型，表示一个空对象引用。

用 typeof 检测 null 返回是object

你可以设置为 null 来清空对象:

# undefined

在 JavaScript 中, undefined 是一个没有设置值的变量。

typeof 一个没有值的变量会返回 undefined。

# JavaScript 类型转换

Number() 转换为数字， String() 转换为字符串， Boolean() 转化为布尔值

# JavaScript 数据类型

**在 JavaScript 中有 6 种不同的数据类型：**string，number，boolean，object，function，symbol**，3 种对象类型：**Object，Date，Array**，2 个不包含任何值的数据类型：**null，undefined

注意：

* NaN 的数据类型是 number
* 数组(Array)的数据类型是 object
* 日期(Date)的数据类型为 object
* null 的数据类型是 object
* 未定义变量的数据类型为 undefined

如果对象是 JavaScript Array 或 JavaScript Date ，我们就无法通过 **typeof** 来判断他们的类型，因为都是 返回 object。

# constructor 属性

**constructor** 属性返回所有 JavaScript 变量的构造函数。

你可以使用 constructor 属性来查看对象是否为数组 (包含字符串 "Array"):

# JavaScript 类型转换

全局方法 **String()** 可以将数字转换为字符串。

该方法可用于任何类型的数字，字母，变量，表达式：

## 实例

*String*(x)         // 将变量 x 转换为字符串并返回

*String*(123)       // 将数字 123 转换为字符串并返回

*String*(100 + 23)  // 将数字表达式转换为字符串并返回

## Number 方法 toString() 也是有同样的效果。

## 实例

x.toString()

(123).toString()

(100 + 23).toString()

# 将布尔值转换为字符串

全局方法 **String()** 可以将布尔值转换为字符串。

*String*(false)        // 返回 "false"

*String*(true)         // 返回 "true"

Boolean 方法 **toString()** 也有相同的效果。

false.toString()     // 返回 "false"

true.toString()      // 返回 "true"

# 将日期转换为字符串

Date() 返回字符串。

Date()      // 返回 Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)

**全局方法 String() 可以将日期对象转换为字符串。**

String(new Date())      // 返回 Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)

Date 方法 **toString()** 也有相同的效果。

## 实例

obj = new Date()  
obj.toString()   // 返回 Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)

|  |  |
| --- | --- |
| **方法** | **描述** |
| getDate() | 从 Date 对象返回一个月中的某一天 (1 ~ 31)。 |
| getDay() | 从 Date 对象返回一周中的某一天 (0 ~ 6)。 |
| getFullYear() | 从 Date 对象以四位数字返回年份。 |
| getHours() | 返回 Date 对象的小时 (0 ~ 23)。 |
| getMilliseconds() | 返回 Date 对象的毫秒(0 ~ 999)。 |
| getMinutes() | 返回 Date 对象的分钟 (0 ~ 59)。 |
| getMonth() | 从 Date 对象返回月份 (0 ~ 11)。 |
| getSeconds() | 返回 Date 对象的秒数 (0 ~ 59)。 |
| getTime() | 返回 1970 年 1 月 1 日至今的毫秒数。 |

# 将字符串转换为数字

全局方法 **Number()** 可以将字符串转换为数字

字符串包含数字(如 "3.14") 转换为数字.

空字符串转换为 0

其他的字符串会转换为 NaN (不是个数字)

*Number*("3.14")    // 返回 3.14

*Number*(" ")       // 返回 0

*Number*("")        // 返回 0

*Number*("99 88")   // 返回 NaN

# 将布尔值转换为数字

全局方法 **Number()** 可将布尔值转换为数字

*Number*(false)     // 返回 0

*Number*(true)      // 返回 1

# 将日期转换为数字

全局方法 **Number()** 可将日期转换为数字

d = new Date();

*Number*(d)          // 返回 1404568027739

日期方法 **getTime()** 也有相同的效果。

d = new Date();

d.getTime()        // 返回 1404568027739

# 自动转换类型

当 JavaScript 尝试操作一个 "错误" 的数据类型时，会自动转换为 "正确" 的数据类型。

以下输出结果不是你所期望的：

5 + null    // 返回 5         null 转换为 0

"5" + null  // 返回"5null"   null 转换为 "null"

"5" + 1     // 返回 "51"      1 转换为 "1"

"5" - 1     // 返回 4         "5" 转换为 5

# 自动转换为字符串

当你尝试输出一个对象或一个变量时 JavaScript 会自动调用变量的

**toString() 方法：**

document.getElementById("demo").innerHTML = myVar;

myVar = {name:"Fjohn"}  // toString 转换为 "[object Object]"

myVar = [1,2,3,4]       // toString 转换为 "1,2,3,4"

myVar = new Date()      // toString 转换为 "Fri Jul 18 2014 09:08:55 GMT+0200"

数字和布尔值也经常相互转换：

myVar = 123             // toString 转换为 "123"

myVar = true            // toString 转换为 "true"

myVar = false           // toString 转换为 "false"

# JS对象（重点）

## 使用对象构造器

## 实例

*function* person(*firstname*,*lastname*,*age*,*eyecolor*)

{

    this.firstname=firstname;

    this.lastname=lastname;

    this.age=age;

    this.eyecolor=eyecolor;

}

# 把方法添加到 JavaScript 对象

方法只不过是附加在对象上的函数。

在构造器函数内部定义对象的方法：

*function* person(*firstname*,*lastname*,*age*,*eyecolor*)

{

    this.firstname=firstname;

    this.lastname=lastname;

    this.age=age;

    this.eyecolor=eyecolor;

    this.changeName=changeName;

*function* changeName(*name*)

    {

        this.lastname=name;

    }

}

changeName() 函数 name 的值赋给 person 的 lastname 属性。

# JavaScript prototype（原型对象）

所有的 JavaScript 对象都会从一个 prototype（原型对象）中继承属性和方法。

## 实例

*function* Person(*first*, *last*, *age*, *eyecolor*) {

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

}

*var* myFather = new Person("John", "Doe", 50, "blue");

*var* myMother = new Person("Sally", "Rally", 48, "green");

我们也知道在一个已存在的对象构造器中是不能添加新的属性的：

要添加一个新的属性需要在在构造器函数中添加：

## 实例

*function* Person(*first*, *last*, *age*, *eyecolor*) {

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

  this.nationality = "English";

}

# prototype 继承

所有的 JavaScript 对象都会从一个 prototype（原型对象）中继承属性和方法：

* Date 对象从 Date.prototype 继承。
* Array 对象从 Array.prototype 继承。
* Person 对象从 Person.prototype 继承。

所有 JavaScript 中的对象都是位于原型链顶端的 Object 的实例。

JavaScript 对象有一个指向一个原型对象的链。当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

Date 对象, Array 对象, 以及 Person 对象从 Object.prototype 继承。

### 添加属性和方法

有的时候我们想要在所有已经存在的对象添加新的属性或方法。另外，有时候我们想要在对象的构造函数中添加属性或方法。使用 prototype 属性就可以给对象的构造函数添加新的属性：

## 实例

*function* Person(*first*, *last*, *age*, *eyecolor*) {

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

}

*Person*.prototype.nationality = "English";

当然我们也可以使用 prototype 属性就可以给对象的构造函数添加新的方法：

## 实例

*function* Person(*first*, *last*, *age*, *eyecolor*) {

  this.firstName = first;

  this.lastName = last;

  this.age = age;

  this.eyeColor = eyecolor;

}

*Person*.prototype.name = *function*() {

  return this.firstName + " " + this.lastName;

};

# JavaScript Window - 浏览器对象模型

浏览器对象模型 (BOM) 使 JavaScript 有能力与浏览器"对话"。

## 浏览器对象模型 (BOM)

## Window 尺寸

有三种方法能够确定浏览器窗口的尺寸。

对于Internet Explorer、Chrome、Firefox、Opera 以及 Safari：

•   window.innerHeight - 浏览器窗口的内部高度(包括滚动条)

•   window.innerWidth - 浏览器窗口的内部宽度(包括滚动条)

对于 Internet Explorer 8、7、6、5：

•   document.documentElement.clientHeight

•   document.documentElement.clientWidth

或者

•   document.body.clientHeight

•   document.body.clientWidth

## 其他 Window 方法

一些其他方法：

        •   window.open() - 打开新窗口

        •   window.close() - 关闭当前窗口

        •   window.moveTo() - 移动当前窗口

        •   window.resizeTo() - 调整当前窗口的尺寸

# JavaScript Window Screen

window.screen 对象包含有关用户屏幕的信息。

## Window Screen

**window.screen**对象在编写时可以不使用 window 这个前缀。

一些属性：

•   screen.availWidth - 可用的屏幕宽度

•   screen.availHeight - 可用的屏幕高度

# JavaScript Window Location

window.location 对象用于获得当前页面的地址 (URL)，并把浏览器重定向到新的页面。

## Window Location

**window.location** 对象在编写时可不使用 window 这个前缀

一些实例:

•   location.hostname 返回 web 主机的域名

•   location.pathname 返回当前页面的路径和文件名

•   location.port 返回 web 主机的端口 （80 或 443）

•   location.protocol 返回所使用的 web 协议（http: 或 https:）

## Window Location Href

location.href 属性返回当前页面的 URL。

## Window Location Pathname

location.pathname 属性返回 URL 的路径名。

# JavaScript Window History

window.history 对象包含浏览器的历史。

## Window History

**window.history**对象在编写时可不使用 window 这个前缀。

为了保护用户隐私，对 JavaScript 访问该对象的方法做出了限制。

一些方法：

•   history.back() - 与在浏览器点击后退按钮相同

•   history.forward() - 与在浏览器中点击向前按钮相同

## Window history.back()

history.back() 方法加载历史列表中的前一个 URL。

Window history.forward()

history forward() 方法加载历史列表中的下一个 URL。

# JavaScript 弹窗

可以在 JavaScript 中创建三种消息框：警告框、确认框、提示框。

## 警告框

警告框经常用于确保用户可以得到某些信息。

当警告框出现后，用户需要点击确定按钮才能继续进行操作。

### 语法

window.alert("sometext");

**window.alert()** 方法可以不带上window对象，直接使用**alert()**方法。

## 确认框

确认框通常用于验证是否接受用户操作。

当确认卡弹出时，用户可以点击 "确认" 或者 "取消" 来确定用户操作。

当你点击 "确认", 确认框返回 true， 如果点击 "取消", 确认框返回 false。

### 语法

window.confirm("sometext");

**window.confirm()** 方法可以不带上window对象，直接使用**confirm()**方法。

## 提示框

提示框经常用于提示用户在进入页面前输入某个值。

当提示框出现后，用户需要输入某个值，然后点击确认或取消按钮才能继续操纵。

如果用户点击确认，那么返回值为输入的值。如果用户点击取消，那么返回值为 null。

### 语法

window.prompt("sometext","defaultvalue");

**window.prompt()** 方法可以不带上window对象，直接使用**prompt()**方法

## 换行

弹窗使用 反斜杠 + "n"(\n) 来设置换行。

## 实例

alert("Hello\nHow are you?");

## JavaScript 计时事件

通过使用 JavaScript，我们有能力做到在一个设定的时间间隔之后来执行代码，而不是在函数被调用后立即执行。我们称之为计时事件。

在 JavaScritp 中使用计时事件是很容易的，两个关键方法是:

•   setInterval() - 间隔指定的毫秒数不停地执行指定的代码。

•   setTimeout() - 在指定的毫秒数后执行指定代码。

**注意:** setInterval() 和 setTimeout() 是 HTML DOM Window对象的两个方法。

## setInterval() 方法

setInterval() 间隔指定的毫秒数不停地执行指定的代码

### 语法

window.setInterval("javascript function",milliseconds);

**window.setInterval()** 方法可以不使用 window 前缀，直接使用函数 **setInterval()**。

setInterval() 第一个参数是函数（function）。

第二个参数间隔的毫秒数

**注意:** 1000 毫秒是一秒。

## 实例

*var* myVar=setInterval(*function*(){myTimer()},1000);

*function* myTimer(){

*var* d=new Date();

*var* t=d.toLocaleTimeString();

    document.getElementById("demo").innerHTML=t;

}

## 如何停止执行?

clearInterval() 方法用于停止 setInterval() 方法执行的函数代码。

### 语法

window.clearInterval(intervalVariable)

**window.clearInterval()** 方法可以不使用window前缀

直接使用函数**clearInterval()**

要使用 clearInterval() 方法, 在创建计时方法时你必须使用全局变量：

myVar=setInterval("javascript function",milliseconds);

然后你可以使用 clearInterval() 方法来停止执行。

## 实例

*var* myVar=setInterval(*function*(){myTimer()},1000);

*function* myTimer(){

*var* d=new Date();

*var* t=d.toLocaleTimeString();

    document.getElementById("demo").innerHTML=t;

}

*function* myStopFunction(){

    clearInterval(myVar);

}

## setTimeout() 方法

### 语法

myVar= window.setTimeout("javascript function", milliseconds);

setTimeout() 方法会返回某个值。在上面的语句中，值被储存在名为 myVar 的变量中。假如你希望取消这个 setTimeout()，你可以使用这个变量名来指定它。

setTimeout() 的第一个参数是含有 JavaScript 语句的字符串。这个语句可能诸如 "alert('5 seconds!')"，或者对函数的调用，诸如 alertMsg。

第二个参数指示从当前起多少毫秒后执行第一个参数。

提示：1000 毫秒等于一秒。

## 实例

等待3秒，然后弹出 "Hello":

setTimeout(*function*(){

    alert("Hello")

},3000);

# JavaScript Cookie

Cookie 用于存储 web 页面的用户信息。

## 什么是 Cookie？

Cookie 是一些数据, 存储于你电脑上的文本文件中。

当 web 服务器向浏览器发送 web 页面时，在连接关闭后，服务端不会记录用户的信息。

Cookie 的作用就是用于解决 "如何记录客户端的用户信息":

* 当用户访问 web 页面时，他的名字可以记录在 cookie 中。
* 在用户下一次访问该页面时，可以在 cookie 中读取用户访问记录。

Cookie 以名/值对形式存储，如下所示:

username=John Doe

当浏览器从服务器上请求 web 页面时， 属于该页面的 cookie 会被添加到该请求中。服务端通过这种方式来获取用户的信息。

## 使用 JavaScript 创建Cookie

JavaScript 可以使用 **document.cookie** 属性来创建 、读取、及删除 cookie。

JavaScript 中，创建 cookie 如下所示：

document.cookie="username=John Doe";

您还可以为 cookie 添加一个过期时间（以 UTC 或 GMT 时间）。默认情况下，cookie 在浏览器关闭时删除：

document.cookie="username=John Doe; expires=Thu, 18 Dec 2043 12:00:00 GMT";

您可以使用 path 参数告诉浏览器 cookie 的路径。默认情况下，cookie 属于当前页面

document.cookie="username=John Doe; expires=Thu, 18 Dec 2043 12:00:00 GMT; path=/";

## 使用 JavaScript 读取 Cookie

在 JavaScript 中, 可以使用以下代码来读取 cookie：

*var* x = document.cookie;

document.cookie 将以字符串的方式返回所有的 cookie，类型格式： cookie1=value; cookie2=value; cookie3=value;

## 使用 JavaScript 修改 Cookie

在 JavaScript 中，修改 cookie 类似于创建 cookie，如下所示：

document.cookie="username=John Smith; expires=Thu, 18 Dec 2043 12:00:00 GMT; path=/";

旧的 cookie 将被覆盖。

## 使用 JavaScript 删除 Cookie

删除 cookie 非常简单。您只需要设置 expires 参数为以前的时间即可，如下所示，设置为 Thu, 01 Jan 1970 00:00:00 GMT:

document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 GMT";

注意，当您删除时不必指定 cookie 的值。

## Cookie 字符串

document.cookie 属性看起来像一个普通的文本字符串，其实它不是。

即使您在 document.cookie 中写入一个完整的 cookie 字符串, 当您重新读取该 cookie 信息时，cookie 信息是以名/值对的形式展示的。

如果您设置了新的 cookie，旧的 cookie 不会被覆盖。 新 cookie 将添加到 document.cookie 中，所以如果您重新读取document.cookie，您将获得如下所示的数据：

cookie1=value; cookie2=value;

      如果您需要查找一个指定 cookie 值，您必须创建一个JavaScript 函数在 cookie 字符串中查找 cookie 值。

## JavaScript Cookie 实例

在以下实例中，我们将创建 cookie 来存储访问者名称。

首先，访问者访问 web 页面, 他将被要求填写自己的名字。该名字会存储在 cookie 中。

访问者下一次访问页面时，他会看到一个欢迎的消息。

在这个实例中我们会创建 3 个 JavaScript 函数:

1. 设置 cookie 值的函数
2. 获取 cookie 值的函数
3. 检测 cookie 值的函数

## 设置 cookie 值的函数

首先，我们创建一个函数用于存储访问者的名字：

*function* setCookie(*cname*,*cvalue*,*exdays*){

*var* d = new Date();

  d.setTime(d.getTime()+(exdays\*24\*60\*60\*1000));

*var* expires = "expires="+d.toGMTString();

  document.cookie = cname + "=" + cvalue + "; " + expires;

}

**函数解析：**

以上的函数参数中，cookie 的名称为 cname，cookie 的值为 cvalue，并设置了 cookie 的过期时间 expires。

该函数设置了 cookie 名、cookie 值、cookie过期时间。

## 获取 cookie 值的函数

然后，我们创建一个函数用于返回指定 cookie 的值：

*function* getCookie(*cname*){

*var* name = cname + "=";

*var* ca = document.cookie.split(';');

  for(*var* i=0; i<ca.length; i++)

  {

*var* c = ca[i].trim();

    if (c.indexOf(name)==0) return c.substring(name.length,c.length);

  }

  return "";

}

**函数解析：**

cookie 名的参数为 cname。

创建一个文本变量用于检索指定 cookie :cname + "="。

使用分号来分割 document.cookie 字符串，并将分割后的字符串数组赋值给 ca (ca = document.cookie.split(';'))。

循环 ca 数组 (i=0;i<ca.length;i++)，然后读取数组中的每个值，并去除前后空格 (c=ca[i].trim())。

如果找到 cookie(c.indexOf(name) == 0)，返回 cookie 的值 (c.substring(name.length,c.length)。

如果没有找到 cookie, 返回 ""。

## 检测 cookie 值的函数

最后，我们可以创建一个检测 cookie 是否创建的函数。

如果设置了 cookie，将显示一个问候信息。

如果没有设置 cookie，将会显示一个弹窗用于询问访问者的名字，并调用 setCookie 函数将访问者的名字存储 365 天：

*function* checkCookie()

{

*var* username=getCookie("username");

  if (username!="")

  {

    alert("Welcome again " + username);

  }

  else

  {

    username = prompt("Please enter your name:","");

    if (username!="" && username!=null)

    {

      setCookie("username",username,365);

    }

  }

}

## 完整实例

## 实例

*function* setCookie(*cname*,*cvalue*,*exdays*){

*var* d = new Date();

    d.setTime(d.getTime()+(exdays\*24\*60\*60\*1000));

*var* expires = "expires="+d.toGMTString();

    document.cookie = cname+"="+cvalue+"; "+expires;

}

*function* getCookie(*cname*){

*var* name = cname + "=";

*var* ca = document.cookie.split(';');

    for(*var* i=0; i<ca.length; i++) {

*var* c = ca[i].trim();

        if (c.indexOf(name)==0) {

return c.substring(name.length,c.length);

}

    }

    return "";

}

*function* checkCookie(){

*var* user=getCookie("username");

    if (user!=""){

        alert("欢迎 " + user + " 再次访问");

    }

    else {

        user = prompt("请输入你的名字:","");

        if (user!="" && user!=null){

            setCookie("username",user,30);

        }

    }

}

# 操作数组

数组有很多方法

1. **Array.map()**

此方法是将数组中的每个元素调用一个提供的函数，结果作为一个新的数组返回，并没有改变原来的数组

*let* arr = [1, 2, 3, 4, 5]

*let* newArr = arr.map(*x* *=>* x\*2)

    arr= [1, 2, 3, 4, 5]   //原数组保持不变

    newArr = [2, 4, 6, 8, 10] //返回新数组

1. **Array.forEach()**

此方法是将数组中的每个元素执行传进提供的函数，没有返回值，注意和map方法区分

*let* arr = [1, 2, 3, 4, 5]

    num.forEach(*x* *=>* x\*2)

    arr = [1, 2, 3, 4, 5]  // 数组改变,注意和map区分

1. Array.filter()

此方法是将所有元素进行判断，将满足条件的元素作为一个新的数组返回

|  |
| --- |
|  |

*let* arr = [1, 2, 3, 4, 5]

*const* isBigEnough = *value* *=>* value >= 3

*let* newArr = arr.filter(isBigEnough )

    newNum = [3, 4, 5]   //满足条件的元素返回为一个新的数组

1. Array.every()

此方法是将所有元素进行判断返回一个布尔值，如果所有元素都满足判断条件，则返回true，否则为false：

*let* arr = [1, 2, 3, 4, 5]

*const* isLessThan4 = *value* *=>* value < 4

*const* isLessThan6 => *value* *=>* value < 6

    arr.every(isLessThan4 ) //false

    arr.every(isLessThan6 ) //true

1. Array.some()

此方法是将所有元素进行判断返回一个布尔值，如果存在元素都满足判断条件，则返回true，若所有元素都不满足判断条件，则返回false：

*let* arr= [1, 2, 3, 4, 5]

*const* isLessThan4 = *value* *=>* value < 4

*const* isLessThan6 = *value* *=>* value > 6

    arr.some(isLessThan4 ) //true

    arr.some(isLessThan6 ) //false

1. Array.reduce()

此方法是所有元素调用返回函数，返回值为最后结果,传入的值必须是函数类型：

*let* arr = [1, 2, 3, 4, 5]

*const* add = (*a*, *b*) *=>* a + b

*let* sum = arr.reduce(add)

   sum = 15  //相当于累加的效果

与之相对应的还有一个 Array.reduceRight() 方法，区别是这个是从右向左操作的

1. Array.push()

此方法是在数组的后面添加新加元素，此方法改变了数组的长度：

1. Array.pop()

此方法在数组后面删除最后一个元素，并返回数组，此方法改变了数组的长度：

*let* arr = [1, 2, 3, 4, 5]

    arr.pop()

    console.log(arr) //[1, 2, 3, 4]

    console.log(arr.length) //4

1. Array.shift()

此方法在数组后面删除第一个元素，并返回数组，此方法改变了数组的长度：

*let* arr = [1, 2, 3, 4, 5]

    arr.shift()

    console.log(arr) //[2, 3, 4, 5]

    console.log(arr.length) //4

1. Array.unshift()

此方法是将一个或多个元素添加到数组的开头，并返回新数组的长度：

*let* arr = [1, 2, 3, 4, 5]

    arr.unshift(6, 7)

    console.log(arr) //[6, 7, 1, 2, 3, 4, 5]

    console.log(arr.length) //7

1. Array.isArray()

判断一个对象是不是数组，返回的是布尔值

1. Array.concat()

此方法是一个可以将多个数组拼接成一个数组：

*let* arr1 = [1, 2, 3]

    arr2 = [4, 5]

*let* arr = arr1.concat(arr2)

    console.log(arr)//[1, 2, 3, 4, 5]

1. Array.toString()

此方法将数组转化为字符串：

*let* arr = [1, 2, 3, 4, 5];

*let* str = arr.toString()

   console.log(str)// 1,2,3,4,5

1. Array.join()

此方法也是将数组转化为字符串：

*let* arr = [1, 2, 3, 4, 5];

*let* str1 = arr.toString()

*let* str2 = arr.toString(',')

*let* str3 = arr.toString('##')

   console.log(str1)// 12345

   console.log(str2)// 1,2,3,4,5

   console.log(str3)// 1##2##3##4##5

通过例子可以看出和toString的区别，可以设置元素之间的间隔~

1. Array.splice(开始位置， 删除的个数，元素)万能方法，可以实现增删改：

*let* arr = [1, 2, 3, 4, 5];

*let* arr1 = arr.splice(2, 0 'haha')

*let* arr2 = arr.splice(2, 3)

*let* arr1 = arr.splice(2, 1 'haha')

    console.log(arr1) //[1, 2, 'haha', 3, 4, 5]新增一个元素

    console.log(arr2) //[1, 2] 删除三个元素

    console.log(arr3) //[1, 2, 'haha', 4, 5] 替换一个元素