javascript学习

第一章 基础语法

一、基本情况

1、介绍

JavaScript是一门解释性的脚本语言，主要用来给HTML网页增加动态功能。

通常的js是运行在浏览器环境下的，可以帮助我们去控制页面，实现丰富的功能。会有dom和bom的api去操作html文档和浏览器的一些功能。

nodejs是运行在计算机环境下的。语法一样，但是因为环境是计算机，他当然不能操作dom和bom。因为压根就没有，但是他能操作文件，能操作数据库，他其实是一门后端的编程语言。

但是nodejs的出现个js提供了蓬勃的生命力，让js更加强大，比如现在流行的一下编程模式，都需要nodejs的支持。

2、JS解释器

无论是node还是各大浏览器，都需要有解释JS代码的引擎，参考下表浏览器使用的JS解释器

```
- Mozilla       --    Spidermonkey      火狐
- Chrome        --    v8                谷歌
- Safari        --    JavaScriptCore    苹果
- IE、Edge      --    Chakra            ie
- node          --    v8                 nodejs
```

3、js哪里可以执行

（1）放在html标签之中

```
<body>
//中间放页面标签

//放在body的结束标签之前
<script type="text/javascript">
    document.write('<h1>123</h1>')
</script>
</body>
```

（2）引入外部的js

```
<body>
//中间放页面标签

//放在body的结束标签之前
<script src="./index.js"></script>
</body>
```

二、数据类型

> 弱类型自动推断类型

数字（number）

字符串 （string）

布尔型 （boolean）

> null是有值但为空，undefined是只是被申明，未赋值

空（null）

未定义（ undefined）

三、定义变量

> 弱类型，不需要申明这个变量的类型，统一用 var

```
var num = 10;
var money = 1.2;
//字符串单引号和双引号都行，和java对比
var str = 'str';
var str2 = "str2";
var nul = null;
var flag = true;
//压根就没有定义叫undefined
//数组和对象
var arr = [];
var obj = {};

```

因为var有一些弊端，今天就说一个，如果前边定义了一个变量，后边再次定义，就会覆盖，这样会有问题，所有在ES6语法当中新增了let和const两个关键字来定义变量，除此之外还有作用域的问题，有兴趣可以去研究。

```
var num = 3;
var num = 4;
//前边的值会被后边的覆盖
```

//你们不用管

```
//let 和 const 定义的变量不能不覆盖，不能重复定义。
let num = 3;
let num = 4;
//直接会报错
```

```
//cosnt定义的叫常量，定义之后的数据不能被修改
const num = 3;
num = 4;
//直接会报错
```

四、数组（array）

1、定义的方式

（1）使用方法调用

```
var arr = Array();
//Array 是个函数，猜一猜他的返回值就是个空数组
```

（2）使用new关键字

```
var arr = new Array();
//js里函数可以当类使用
```

（3）使用json数组的方式，字面量，个人推荐

```
var arr = [];
```

> 注：js中初始化数组不需要初始化长度

2、赋值的方式

（1）定义之后去赋值

```
arr[0] = 123;
arr[1] = 234;
```

（2）定义的时候赋值

```
//这样当然方便
var arr = [123,234];
```

五、方法

js和java定义方法对比

js对参数要求极其灵活，java极其严格

js不需要声明数据类型，java必须声明数据类型

```
public void getName(){
    System.out.println("123");
}
```

```
function getName(){

}
```

```
//这种方法没有参数没有返回值，右不打印没用让我们看见就毫无意义
function puls(){
    var a = 1 +1;
}
puls()
undefined

//参数是你要处理的数据，
//返回值就是处理的结果
//返回值可以给
function plus(num1,num2){

    return num1 + num2;

}
plus(23,34)
57

//可以使用另一个变量去接收返回值并进行利用
var num = plus(33,44)
num
77

//方法会在遇到return之后停止运行
function plus(num1,num2){
    console.log("before")
    return num1 + num2;
    console.log("after")
}

plus(34,45)
VM688:2 before
79
```

```
return
1终止当前的方法，不在继续往下执行
2、会将return之后的结果，作为方法执行的结果返回出去
返回出去有什么用，别的变量就能拿到这个值，并进行利用
```

参数可以没有，可以有一个或多个

返回值要么没有，要么只有一个

六、 对象（object）

对象是js里最灵活的。

1、定义空对象

（1）使用方法调用

```
var obj = Object();
//Array 是个函数，猜一猜他的返回值就是个空数组
```

（2）使用new关键字

```
var obj = new Object();
//js里函数可以当类使用
```

（3）使用json对象的方式，个人推荐

```
var obj = {};
```

（4）自定义对象类型，有点高级，了解

这一点很是灵活，function定义的函数，既能直接调用，也可以像类一样使用new关键字来生成。也就是函数既可以当做普通函数，也能当做构造函数。

其中要注意，要想给new出来的对象添加属性或方法，必须使用this关键字，不写不行。

命名规范和java一样，首字母大写，驼峰式命名。

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

var user = new User('wusanshui');
console.log(user.name)
```

2、给对象添加属性和方法

（1）定义了对象之后

```
obj.name = 'zhangsan';
obj.age = 18;
onj.eat = function(){
    console.log(" I am eating! ")
}
```

（2）定义类的时候

```
//直接用json对象写一个对象出来
var user = {
    name: 'zhangsan',
    age: 10,
    eat: function(){
        console.log("i am eating! ");
    }
}

```

（3）自定义类的时候

一定要注意和java的区别

```
function User(name){
    this.name = name;
    this.age = 18;
    this.eat = function(){
        console.log("I am eating!")
    }
}

var user = new User('wusanshui');

//new 出来的对象自然而然就拥有这些属性和方法

```

3、获取对象的属性的方法

（1）使用.

```
console.log(user.name);
调用方法
user.eat();
```

（2）使用[]

```
console.log(user['name']);
调用方法
user['eat']();
```

六、判断和循环

和java里的一模一样，简单写一下就行了

1、if语句

如果条件是一个值：

如果是 0 ” null undefined false 都是false

{} [] 非零的数字 字符串 都是真

```
var flag = true;

if(flag){
   alert(true)
}else{
   alert(false)
}
```

2、switch语句

```
var m = 2;
var x;
switch (m) {
  case 0:x="今天是星期日";
  break;
  case 1:x="今天是星期一";
  break;
  case 2:x="今天是星期二";
  break;
}

console.log(x);
```

3、循环数组

```
let cars = [ '兰博基尼','CRV','卡宴',"奔驰是傻逼",'bwm' ];
for (var i=0;i<cars.length;i++)
{
    console.log(cars[i]);
}
```

4、遍历对象属性

注意：获取对象属性的时候可以用. 也可以用[key]

```
var options = {
    name: 'zhangsan',
    age: 10
}
for(var attr in options){
    console.log(attr)
    //正确
    console.log(options[attr])
    //错误
    console.log(options.attr)
}

```

第二章 常见内置对象

一、Array对象

方法

1. concat（）　表示把几个数组合并成一个数组
2. join（）　设置分隔符连接数组元素为一个字符串
3. pop（）　移除数组最后一个元素
4. shift（）　移除数组中第一个元素
5. slice（start，end）　返回数组中的一段
6. splice（）　可以用来删除，可以用来插入，也可以用来替换
7. push（）　往数组中新添加一个元素，返回最新长度
8. sort（）　对数组进行排序
9. reverse（）　反转数组的顺序
10. toLocaleString()　把数组转换为本地字符串

属性：

1. length　表示取得当前数组长度 （常用）
2. constructor　引用数组对象的构造函数
3. prototype　通过增加属性和方法扩展数组定义

二、Global对象

1. escape（）　对字符串编码
2. eval（）　把字符串解析为JavaScript代码并执行
3. isNaN()　判断一个值是否是NaN
4. parseInt（）　解析一个字符串并返回一个整数
5. parseFloat（）　解析一个字符串并返回一个浮点数
6. number（）　把对象的值转换为数字
7. string（）　把对象的值转换为字符串

三、String对象

1. charAt()　返回指定索引的位置的字符
2. indexOf()　从前向后检索字符串，看是否含有指定字符串
3. lastIndexOf()　从后向前检索字符串，看是否含有指定字符串

4. concat() 连接两个或多个字符串
5. match() 使用正则表达式模式对字符串执行查找，并将包含查找结果最为结果返回
6. replace() 替换一个与正则表达式匹配的子串
7. search() 检索字符串中与正则表达式匹配的子串。如果没有找到匹配，则返回-1。
8. slice（start，end） 根据下表截取子串
9. substring（start，end） 根据下表截取子串
10. split() 根据指定分隔符将字符串分割成多个子串，并返回素组
11. substr(start，length) 根据长度截取字符串 *
12. toUpperCase() 返回一个字符串，该字符串中的所有字母都被转化为大写字母。
13. toLowerCase() 返回一个字符串，该字符串中的所有字母都被转化为小写字母。

四、Math对象

1. ceil() 向上取整。
2. floor() 向下取整。
3. round() 四舍五入。
4. random() 取随机数。

五、Date对象

1. getDate函数: 返回日期的“日”部分，值为1～31。
2. getDay函数: 返回星期，值为0～6，0表示星期日。
3. getHours函数: 返回日期的“小时”部分，值为0～23。
4. getMinutes函数: 返回日期的“分钟”部分，值为0～59。
5. getMonth函数: 返回日期的“月”部分，值为0～11。
6. getSeconds函数: 返回日期的“秒”部分，值为0～59。
7. getTime函数: 返回系统时间。
8. getTimezoneOffset函数: 返回此地区的时差(当地时间与GMT格林威治标准时间的地区时差)，单位为分钟。
9. getYear函数: 返回日期的“年”部分。返回值以1900年为基数，如1999年为99。
10. parse函数: 返回从1970年1月1日零时整算起的毫秒数(当地时间)
11. setDate函数: 设定日期的“日”部分，值为0～31。
12. setHours函数: 设定日期的“小时”部分，值为0～23。
13. setMinutes函数: 设定日期的“分钟”部分，值为0～59。
14. setMonth函数: 设定日期的“月”部分，值为0～11。其中0表示1月，...，11表示12月。
15. setSeconds函数: 设定日期的“秒”部分，值为0～59。
16. setTime函数: 设定时间。时间数值为1970年1月1日零时整算起的毫秒数。
17. setYear函数: 设定日期的“年”部分。
18. toGMTString函数: 转换日期成为字符串，为GMT格林威治标准时间。
19. setLocaleString函数: 转换日期成为字符串，为当地时间。
20. UTC函数: 返回从1970年1月1日零时整算起的毫秒数(GMT)。

第三章 DOM编程

一、概述

在 HTML DOM (Document Object Model) 即文档对象模型中, 每个东西都是 **节点** :

- 文档本身就是一个文档对象
- 所有 HTML 元素都是元素节点
- 所有 HTML 属性都是属性节点
- 元素内的文本是文本节点
- 注释是注释节点，就不用

```
<div class='test1' id='a'>itnanls</div>

div整体是一个元素节点
class='test1' 是一个属性节点
itnanls是个文本节点，注意中间没有东西空字符也是一个文本节点
```

所有的节点都有一个nodeType属性，可以判断节点类型，常用的就是以下

NodeType	Named Constant
1	ELEMENT_NODE 元素节点
2	ATTRIBUTE_NODE 属性节点
3	TEXT_NODE 文本节点

```
//元素节点
var mydiv = document.getElementById("div1")
mydiv.nodeType
1
//属性节点
mydiv.attributes[0].nodeType
2
```

DOM操作其实就是对节点的增删查改

二、元素节点

1、获取元素节点的方法

```
//根据id获取一个元素节点
var div1 = document.getElementById("div1")
//根据标签名获取一堆节点的集合
var li1 = document.getElementsByTagName("li");
//根据class获取一堆元素节点
var div2 = document.getElementsByClassName("content");

//使用css选择器选择第一个匹配的元素节点
var d1 = document.querySelector(".content")
//根据css选择器选择一批能够被匹配到的所有的元素
var d1 = document.querySelectorAll(".content")
```

2、修改元素节点的内容

```
//不渲染html标签，标签会当做文本打印出来
mydiv.innerText = "jiasoushi"
//会将html标签渲染出来
mydiv.innerHTML = "<h1>123</h1>"
```

3、删除一个元素节点

```
//直接把自己干掉
var mydiv = document.getElementById("div1")
mydiv.remove();

//清除内容
mydiv.innerText = “”;

//删除某个子元素节点
//1、找到这个字元素节点
var myul = document.getElementsByTagName('ul')[0];
//2、调用方法干掉，注意这个方法参数一定要是个元素节点
mydiv.removeChild(myul)

var div1 = document.getElementById("div1")
undefined
var child = document.getElementsByTagName("ul")[0]
undefined
child
<ul>•…•</ul>•
div1.removeChild(child );

```

4、新建一个元素节点

```
//创建一个div标签，内存中
var mydiv = document.createElement("div");
//添加进几个属性
mydiv.setAttribute("name","mydiv");
mydiv.setAttribute("class","test");
//获取到我的div
var div1 = document.getElementById("div1");
//将内存中新建的div实实在在的加入到我的div中
div1.append(mydiv)
```

5、获取所有的子节点

- 获取了之后当然就能像操作节点一样操作他了。
- 子节点一般是个集合，其实就是个数组
- 循环遍历可以批量操作
- 不仅仅是子节点集合，任何节点集合都能批量操作

```
//children属性能获取所有的子元素节点，不包括文本节点
mydiv.children
HTMLCollection [ul]

//children属性能获取所有的子元素节点，包括文本节点
mydiv.childNodes
NodeList(3) [text, ul, text]

//子节点也是元素节点，一样可以有子节点
mydiv.children[0].children
```

三、属性节点

1、使用元素节点方法进行增删查改

```
var mydiv = document.getElementById("div1")
//获取这个属性的值
mydiv.getAttribute("name")
//修改，同时可以添加一个属性的值
mydiv.setAttribute("name","cccc")
//删除一个属性
mydiv.removeAttribute("name")
```

2、使用属性节点对象对属性本身进行操作

```
//获取所有的属性节点的集合，是个可以当成数组
mydiv.attributes
//通过下标拿到第二个属性
mydiv.attributes[1]
//拿到属性的name
var attrName = mydiv.attributes[1].name
//拿到属性的值
var attrValue = mydiv.attributes[1].value
//修改这个属性的值
mydiv.attributes[1].value = "aaa"
```

三、常用属性操作

如 id 、class、style

```
var div1 = document.getElementById("div1");
```

```
//获取id的值
div1.id
"div1"
//给元素标签的id赋值
div1.id = "div2"
"div2"

//获取class属性
div1.className
"content aaa"
//为class属性赋值
div1.className = 'content'
"content"
div1.className
"content"

//直接修改行内样式
div1.style = 'background: black'
"background: black"
```

第四章 BOM编程

一、概述

BOM是浏览器对象模型。

BOM提供了独立于内容 而与浏览器窗口进行交互的对象;

BOM的核心对象是window;

BOM由一系列相关的对象构成，并且每个对象都提供了很多方法与属性;

打开浏览器，F12打开调试窗口，console里输入window，就能看到这个对象。里边有很多的方法和属性，能够帮助我们查看和浏览器相关的一些内容，比如浏览器的版本啦（navigator）、浏览的历时记录啦（history）、网站的地址信息啦（location），和屏幕相关的内容啦（带screende）等等，自己可以浏览一下即可。

二、常用方法

> 回调函数

```
//js函数非常灵活，定义了参数传什么都行
var callback = function(fun){
    console.log(fun)
}
callback(1)
callback("123")
callback( {name:'zhangnan'} )
callback( [1,2,3] )

//实际上传什么，就要把这个参数当成什么来用
```

```
//要是传个方法就要在方法里找个合适的地方调用一下
var callback = function(fun){
    console.log(fun)
}

var test = function(fun){
    console.log("before");
    fun();
    console.log("after");
}

//你知道需要传方法却传了一个数字，更定不能调用，就会报错
test(1)
VM1038:2 before
VM1038:3 Uncaught TypeError: fun is not a function
    at test (<anonymous>:3:5)
    at <anonymous>:1:1
test @ VM1038:3
(anonymous) @ VM1060:1


var callback = function(){
    console.log("I am callback!")
}
test(callback);

//结果
VM1038:2 before
VM1151:2 I am callback!
VM1038:4 after

//callback就是个方法的名字
var callback = function(data){
    console.log("I am callback!"+data)
}

var test = function(fun){
    console.log("before");
    fun();
    console.log("after");
}

var test = function(fun){
    console.log("before");
    var i = 10;
    fun(i);
    console.log("after");
}
//可以直接传名字
test(callback)
VM1296:2 before
VM1255:2 I am callback!10
VM1296:5 after

//也能直接传个方法体进去
test( function(data){
    console.log("I am callback!"+data)
} )
```

```
VM1296:2 before
VM1363:2 I am callback!10
VM1296:5 after

//直接调用方法体也行
(function(){
    console.log(123)
})()
VM1427:2 123

var a  = function(){
    console.log(123)
}
//拿名字调用也行
a()
VM1452:2 123
```

1、setTimeout

```
//一次性定时器，会在多少毫秒后执行这个函数
//里边的是匿名函数，也叫回调函数（就是等过了两秒后回过头来再调用这个函数）
//返回值是个定时器，这个方法是在未来去执行某个函数
var timer = setTimeout( function(){
    console.log(123)
},2000 )

//如果时间未到，不想让他执行了，就需要取消这个定时器
clearTimeout(timer)
```

2、setInterval

```
//周期性定时器，会每隔多少毫秒后执行这个函数
//里边的是匿名函数，也叫回调函数（就是等过了两秒后回过头来再调用这个函数）
//返回值是个定时器，这个方法是在未来去执行某个函数
var timer = setInterval( function(){
    console.log(123)
},2000 )

//如果时间未到，或者中途不想让他执行了，就需要取消这个定时器
clearInterval(timer)
```

3、浏览器自带小型数据库

为每一个网址提供两个小型数据库

```
//localStorage只要不人为删除，会浏览器被删除数据会一直在
//增加或修改一个
window.localStorage.setItem("name","lucy")
```

```
//获取
window.localStorage.getItem("name")
//删除一个
window.localStorage.removeItem("name")
//清空
window.localStorage.clear()

//sessionStorage网页被关闭就没有了
//增加或修改一个
window.sessionStorage.setItem("name","lucy")
//获取
window.sessionStorage.getItem("name")
//删除一个
window.sessionStorage.removeItem("name")
//清空
window.sessionStorage.clear()
```

4、弹窗其实没求用

```
//弹个提示窗口，调试也不要用，不优雅
alert(1)

//弹出确认框
//点击确定就是true 点击否就是false
var flag = confirm("您确定要退出吗?")
console.log(flag)

//弹出信息框
//输入信息后点击确定返回填的内容，点击取消返回none
var message = prompt("请填写您的手机号！");
console.log(message);
VM3797:2 1373838438

var message = window.prompt("请输入名字：")
undefined
message
"张楠"
var message = window.prompt("请输入名字：")
undefined
message
""
var message = window.prompt("请输入名字：")
undefined
message
null
var message = window.prompt("请输入名字：","liankun")
```

5、history

```
//回退
history.go(-1)
//向前
history.go(1)
//回退
history.back()
//向前
window.history.forward()
```

6、navigator

这个属性提供了一写浏览器的属性，比如浏览器类型，版本之类的信息。

5、一点注意

在浏览器模型中，调用的方法或属性其实是属于window对象的

你在最外层定义一个方法或者一个变量其实是赋给了window对象

在浏览器模型中，调用window的方法可以省略window．也可以不省略，如下：

```
window.localStorage.setItem("name","lucy")
localStorage.setItem("name","lucy")
```

浏览器编程中，全局的变量，就是直接在最外边定义变量的时候尽量避开name，应为window有name属性，你再定义就覆盖了人家的了，当然在方法里，对象中可以随便使用。

第五章 事件

事件是为了让我们更好的去和页面进行交互。

事件一般是定义在元素节点上的，所以我们一般称之为给元素节点绑定一个事件。

一 、定义事件

（1）addEventListener

最常用，大神都是这么写。

康永亮说兼容性问题，大家可以自行查阅资料。

```
var div1 = document.getElementById("div1");
div1.addEventListener('click',function(){
    console.log("click");
})
```

（2）onclick

一定要知道，能少用就少用把。

```
var div1 = document.getElementById("div1");
div1.onclick = function(){
    console.log("click")
}
```

（3）标签内使用

也常用

```
<div class="content aaa" onclick="test(123)" id="div1"  name="bbb"></div>

<script type="text/javascript">
    function test(){
        console.log("test");
    }
</script>
```

二、清除事件

（1）方式一

```
div1.onclick = null
或者
div1.onclick = false
```

（2）方式二

使用此方法，必须将回调函数定义在外边，不能使用匿名回调

```
var callback = function(){
    console.log("click")
}
var div1 = document.getElementById("div1");
//添加一个事件
div1.addEventListener('click',callback)
//移除一个事件
div1.removeEventListener('click',callback)
```

三、事件分类

(1) 鼠标事件

```
（常用）
onclick：点击某个对象时触发
ondblclick：双击某个对象时触发
onmouseover：鼠标移入某个元素时触发
onmouseout：鼠标移出某个元素时触发

（知道，不用）
onmouseenter：鼠标进入某个元素时触发
onmouseleave：鼠标离开某个元素时触发
onmousedown：鼠标按下时触发
onmouseup：鼠标抬起时触发
onmousemove：鼠标被移动时触发
onwheel：鼠标滚轮滚动时触发
oncontextmenu：点击鼠标右键时触发
```

(2) 键盘事件

```
键盘事件
onkeydown：键盘的键按下时触发
onkeyup：键盘的键放开时触发
onkeypress：按下或按住键盘键时触发
```

(3) 表单事件

```
常用
onfocus：元素获得焦点时触发
onblur：元素失去焦点时触发
onchange：元素内容改变时触发
oninput：元素获取用户输入时触发
```

(4) （文档、浏览器）对象事件

```
会用到
onload：元素加载完时触发

一辈子也用不了几次，知不知道都行
onresize：浏览器窗口大小改变时触发
onabort：图形的加载被中断时触发
onerror：当加载文档或者图片时（没找到）发生的错误时触发
onscroll：文档滚动时触发
onpageshow：用户访问页面时触发
onunload：用户退出页面时触发
```

四、事件冒泡和捕获

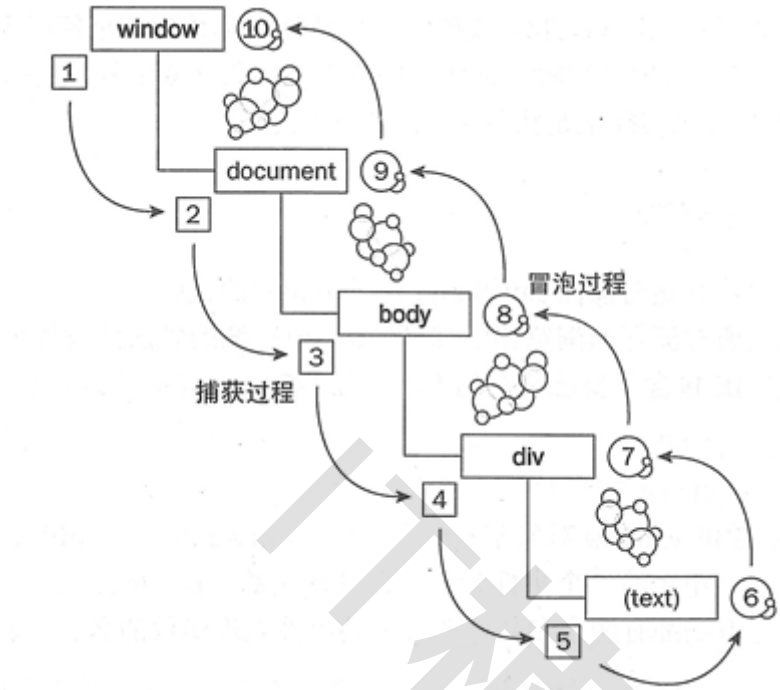
1、事件捕获

捕获型事件(event capturing)：事件从最不精确的对象(document 对象)开始触发，然后到最精确(也可以在窗口级别捕获事件，不过必须由开发人员特别指定)

2、事件冒泡

冒泡型事件：事件按照从最特定的事件目标到最不特定的事件目标(document对象)的顺序触发。

3、捕获和冒泡过程图



事件是先进行捕获，后进行冒泡！

addEventListener的第三个参数如果是false表示在冒泡阶段处理回调函数，如果是true表示在捕获阶段处理回调函数。

怎么阻止事件冒泡？event.stopPropagation();

```
ul.addEventListener('click',function(event){
            console.log('ul')
            event.stopPropagation();
        },true)
```

第六章 语法深入

一、回调函数

在js里参数可以直接传方法。

重要的事情说三遍，在java里不行，在java里不，行在java里不行。

```
//这个函数负责从后天获取数据
var request = function(callback){
    console.log("从服务器获取数据！")
    var data = 123;
    callback(data);
}
//回调通常是我们写的，等参数被请求完成后执行的函数
var callb = function(data){
    console.log("我从服务器获取了数据："+data);
    document.getElementById("div1").innerHTML = data;
}

request(callb);

结果：
VM500:2 从服务器获取数据！
VM679:2 我从服务器获取了数据：123

//使用之中回调函数，屏蔽了难以实现的延时操作，我们只需要关系几秒之后发生的事情（传入的回调方法）就行了
setTimeout(function(){
    console.log(123)
},1000)

//模仿setTimeOut写一下
var mySetTimeOut = function(callback,delay){
       console.log("已经过了"+delay+"毫秒");
       callback();
}
//写一个回调假装实现
mySetTimeOut(function(){
    console.log(123)
},2000)

结果：
VM561:2 已经过了2000毫秒
VM636:2 123
```

二、方法默认传入的形参

1、arguments

方法会将调用时传入的所有参数封装成一个类数组。

js对传参要求的非常灵活，基本上就是想怎么传就怎么传。

所以最重要的一点就是，怎要合适的利用参数。

命名是个数字，就不要当成对象用，命名是个字符串就不要当成数组用。

```
function test(){
    for( let i = 0 ; i < arguments.length ; i++ ){
        console.log(arguments[i])
```

```
    }
}

test(1,'2342',34,456,678,789,null);

VM1546:3 1
VM1546:3 2342
VM1546:3 34
VM1546:3 456
VM1546:3 678
VM1546:3 789
VM1546:3 null
```

```
function test1(){
    for( var i = 0 ; i < arguments.length ; i++  ){
        document.getElementById(arguments[i]).style = 'background:blue'
    }

}

test1()
test1('div1','div2')
```

2、this

this总是指向调用这个方法的实例对象。

在浏览器中，直接定义一个方法，其实是定义在了window这个对象之中，所以直接调用方法其实是window.方法名()，因为在window环境下，所以window通常不用写。所有如果直接调用这个方法，this会指向window。

```
function test(){
    console.log(this)
}

test()
和
window.test()
一样
结果:
VM1620:2 Window {parent: Window, opener: null, top: Window, length: 0,
frames: Window, …}
```

```
function User(name){
    this.name = name;
    this.print = function(){
        console.log(this)
    }
}

let user = new User("李兴");

user.name
"李兴"
```

```
12
13 //使用user调用print this就是这个user
14 user.print()
15 VM1786:4 User {name: "李兴", print: ƒ}
```

```
1  let dog = {
2              name: 'teddy',
3              say: function (a,b) {
4                  var that = this;
5                  console.log(a,b)
6                  setTimeout(function(){
7                      console.log('my name is ' + that.name)
8                  }, 1000);
9              }
10         }
11
12 dog.say();
13 // 这里say方法调用时的this指向调用say的dog对象，
14 // 而setTimeout方法调用时是由window对象负责调用，
15 // 所以setTimeoue的this指向window。
16 // 如果要在setTimeout内使用dog对象需要在外边进行保存
```

this指向的改变

使用call、apply、bind可以改变this的指向

1、第一个参数都是新的this对象

2、从第二个参数开始，需要传递say方法的实参，

3、call是以多个参数的方式传递，而apply是以数组形式传递

4、bind不能直接执行方法，而是返回一个方法，需要另行执行

```
1 dog.say.call({name: '刘奇'},12,23)
2 dog.say.apply({name: '刘奇'},[23,45])
3
4 var fn = dog.say.bind({name: '刘奇'})
5 fn()
```

三、作用域

全局作用域只有一个，每个函数又都有作用域（环境）。

- 编译器运行时会将变量定义在所在作用域
- 使用变量时会从当前作用域开始向上查找变量

- 作用域就像攀亲亲一样，晚辈总是可以向上辈要些东西

1、作用域链

作用域链只向上查找，找到全局window即终止，应该尽量不要在全局作用域中添加变量。

函数被执行后其环境变量将从内存中删除。下面函数在每次执行后将删除函数内部的total变量。

```
function count() {
  let total = 0;
}
count();
```

函数每次调用都会创建一个新作用域

```
let site = 'itnanls';

function a() {
  let hd = 'zn.com';

  function b() {
      let cms = 'itnanls.cn';
      console.log(hd);
      console.log(site);
  }
  b();
}
a();
```

如果子函数被使用时父级环境将被保留

```
function hd() {
  let n = 1;
  return function() {
    let b = 1;
    return function() {
      console.log(++n);
      console.log(++b);
    };
  };
}
let a = hd()();
a(); //2,2
a(); //3,3
```

2、块作用域

使用 `let/const` 可以将变量声明在块作用域中（放在新的环境中，而不是全局中）

```
{
    let a = 9;
}
console.log(a); //ReferenceError: a is not defined
if (true) {
    var i = 1;
}
console.log(i);//1
```

也可以通过下面的定时器函数来体验

```
for (let i = 0; i < 10; i++) {
  setTimeout(() => {
    console.log(i);
  }, 500);
}
```

在 `for` 循环中使用`let/const` 会在每一次迭代中重新生成不同的变量

```
let arr = [];
for (let i = 0; i < 10; i++) {
    arr.push(() => i));
}
console.log(arr[3]()); //3 如果使用var声明将是10
```

在没有`let/const` 的历史中使用以下方式产生作用域

```
//自行构建闭包
var arr = [];
for (var i = 0; i < 10; i++) {
  (function (a) {
      arr.push(()=>a);
  })(i);
}
console.log(arr[3]()); //3
```


四、闭包使用

闭包指子函数可以访问外部作用域变量的函数特性，即使在子函数作用域外也可以访问。如果没有闭包那么在处理事件绑定，异步请求时都会变得困难。

- JS中的所有函数都是闭包
- 闭包一般在子函数本身作用域以外执行，即延伸作用域

一、基本示例

前面在讲作用域时已经在使用闭包特性了，下面再次重温一下闭包。

```
function hd() {
  let name = '欣知';
  return function () {
    return name;
  }
}
let xzcms = hd();
console.log(xzcms()); //欣知
```

二、使用闭包做计数器

计时器中使用闭包来获取独有变量

```
var adder = (function(start) {
            return function() {
                return start++;
            }
        })(10);

console.log(adder());
console.log(adder());
console.log(adder());
adder = null;
```

三、使用闭包做缓存

```
var plus = (function () {
            var cache = {};

            return function () {
                var key = [].join.call(arguments);
                if (key in cache) {
                    console.log('走了缓存')
                    return cache[key]
                }

                console.log('又计算了一次')
                var sum = 0;
                for (var i = 0; i < arguments.length; i++) {
                    sum += arguments[i]
                }
                cache[key] = sum;

                return sum;
            }
        })();

plus(1,2,3,4);
plus(1,2,3,4);
plus(1,2,3,4);
```

四、闭包问题

内存泄漏

闭包特性中上级作用域会为函数保存数据，从而造成的如下所示的内存泄漏问题

```
<body>
  <div desc="zn">在线学习</div>
  <div desc="xzcms">开源产品</div>
</body>
<script>
  let divs = document.querySelectorAll("div");
  divs.forEach(function(item) {
    item.addEventListener("click", function() {
      console.log(item.getAttribute("desc"));
    });
  });
</script>
```

下面通过清除不需要的数据解决内存泄漏问题

```
let divs = document.querySelectorAll("div");
divs.forEach(function(item) {
  let desc = item.getAttribute("desc");
  item.addEventListener("click", function() {
    console.log(desc);
  });
  item = null;
});
```

第七章 原型

一、原型对象

每个对象都有一个原型`prototype`对象，通过函数创建的对象也将拥有这个原型对象。原型是一个指向对象的指针。

- 可以将原型理解为对象的父亲，对象从原型对象继承来属性
- 原型就是对象除了是某个对象的父母外没有什么特别之处
- 所有函数的原型默认是`Object`的实例，所以可以使用`toString/toValues/isPrototypeOf`等方法的原因
- 使用原型对象为多个对象共享属性或方法
- 如果对象本身不存在属性或方法将到原型上查找
- 使用原型可以解决，通过构建函数创建对象时复制多个函数造成的内存占用问题
- 原型包含`constructor`属性，指向构造函数
- 对象包含`__proto__`指向他的原型对象
- 函数有一个prototype属性，当函数作为构造函数时，new出来的对象的`__proto__`指向prototype

下例使用的就是数组原型对象的`concat`方法完成的连接操作

```
let zn = ["a"];
console.log(zn.concat("b"));
console.log(zn);
```

默认情况下创建的对象都有原型

```
let zn = { name: "楠哥" };
console.log(zn);
```

以下x、y的原型都为元对象Object，即JS中的根对象

```
let x = {};
let y = {};
console.log(Object.getPrototypeOf(x) == Object.getPrototypeOf(y)); //true
```

我们也可以创建一个极简对象（纯数据字典对象）没有原型（原型为null）

```
let zn = { name: 3 };
console.log(zn.hasOwnProperty("name"));

let ng = Object.create(null, {
  name: {
    value: "楠哥"
  }
});
console.log(ng.hasOwnProperty("name")); //Error

//Object.keys是静态方法，不是原型方法所以是可以使用的
console.log(Object.keys(ng));
```

函数拥有多个原型，`prototype` 用于实例对象使用，`__proto__`用于函数对象使用

```
function User() {}
User.__proto__.view = function() {
  console.log("User function view method");
};
User.view();

User.prototype.show = function() {
  console.log("楠哥");
};
let zn = new User();
zn.show();
console.log(User.prototype == zn.__proto__);
```

下面是原型关系分析，与方法继承的示例

```
let zn = new Object();
zn.name = "楠哥";
Object.prototype.show = function() {
  console.log("hodunren.com");
};
zn.show();

function User() {}
let ng = new User();
ng.show();
User.show();
```

下面是使用构造函数创建对象的原型体现

- 构造函数拥有原型
- 创建对象时构造函数把原型赋予对象

```
function User() {}
let ng = new User();
console.log(ng.__proto__ == User.prototype);
```

下面使用数组会产生多级继承继原型链

```
let zn = [];
console.log(zn);
console.log(zn.__proto__ == Array.prototype);

let str = "";
console.log(str.__proto__ == String.prototype);
```

下面使用 `setPrototypeOf` 与 `getPrototypeOf` 获取与设置原型

```
let zn = {};
let parent = { name: "parent" };
Object.setPrototypeOf(zn, parent);
console.log(zn);
console.log(Object.getPrototypeOf(zn));
```

使用自定义构造函数创建的对象的原型体现

```
function User() {}
let zn = new User();
console.log(zn);
```

constructor存在于prototype原型中，用于指向构建函数的引用。

```
function zn() {
  this.show = function() {
    return "show method";
  };
}
const obj = new zn(); //true
console.log(obj instanceof zn);

const obj2 = new obj.constructor();
console.dir(obj2.show()); //show method
```

使用对象的 `constructor` 创建对象

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

function createByObject(obj, ...args) {
  const constructor = Object.getPrototypeOf(obj).constructor;
  return new constructor(...args);
}

let zn = new User("楠哥");
let ng = createByObject(zn, "楠哥", 12);
console.log(ng);
```

二、原型链

通过引用类型的原型，继承另一个引用类型的属性与方法，这也是实现继承的步骤。

使用`Object.setPrototypeOf` 可设置对象的原型，下面的示例中继承关系为 obj>zn>cms。

`Object.getPrototypeOf` 用于获取一个对象的原型。

```
let obj = {
  name: "楠哥"
};
let zn = {
  web: "xinzhi"
};
let cms = {
  soft: "zncms"
};
//让obj继承zn，即设置obj的原型为zn
Object.setPrototypeOf(obj, zn);
Object.setPrototypeOf(zn, cms);
console.log(obj.web);
console.log(Object.getPrototypeOf(zn) == cms); //true
```

三、原型检测

instanceof 检测构造函数的 `prototype` 属性是否出现在某个实例对象的原型链上

```
function A() {}
function B() {}
function C() {}

const c = new C();
B.prototype = c;
const b = new B();
A.prototype = b;
const a = new A();

console.dir(a instanceof A); //true
console.dir(a instanceof B); //true
console.dir(a instanceof C); //true
console.dir(b instanceof C); //true
console.dir(c instanceof B); //false
```

使用`isPrototypeOf`检测一个对象是否是另一个对象的原型链中

```
const a = {};
const b = {};
const c = {};

Object.setPrototypeOf(a, b);
Object.setPrototypeOf(b, c);

console.log(b.isPrototypeOf(a)); //true
console.log(c.isPrototypeOf(a)); //true
console.log(c.isPrototypeOf(b)); //true
```

四、属性遍历

使用`in` 检测原型链上是否存在属性，使用 `hasOwnProperty` 只检测当前对象

```
let a = { url: "xinzhi" };
let b = { name: "楠哥" };
Object.setPrototypeOf(a, b);
console.log("name" in a);
console.log(a.hasOwnProperty("name"));
console.log(a.hasOwnProperty("url"));
```

使用 `for/in` 遍历时同时会遍历原型上的属性如下例

```
let zn = { name: "楠哥" };
let ng = Object.create(zn, {
  url: {
    value: "xinzhi.com",
    enumerable: true
  }
});
for (const key in ng) {
  console.log(key);
}
```

`hasOwnProperty` 方法判断对象是否存在属性，而不会查找原型。所以如果只想遍历对象属性使用以下代码

```
let zn = { name: "楠哥" };
let ng = Object.create(zn, {
  url: {
    value: "xinzhi.com",
    enumerable: true
  }
});
for (const key in ng) {
  if (ng.hasOwnProperty(key)) {
    console.log(key);
  }
}
```

五、借用原型

使用 `call` 或 `apply` 可以借用其他原型方法完成功能。

下面的ng对象不能使用`max`方法，但可以借用 zn 对象的原型方法

```
let zn = {
  data: [1, 2, 3, 4, 5]
};
Object.setPrototypeOf(zn, {
  max: function() {
    return this.data.sort((a, b) => b - a)[0];
  }
});
console.log(zn.max());

let ng = {
  lessons: { js: 100, php: 78, node: 78, linux: 125 },
  get data() {
    return Object.values(this.lessons);
  }
};
console.log(zn.__proto__.max.apply(ng));
```

上例中如果方法可以传参，那就可以不在 `ng` 对象中定义 `getter` 方法了

```
let zn = {
  data: [1, 2, 3, 4, 5]
```

```
};
Object.setPrototypeOf(zn, {
  max: function(data) {
    return data.sort((a, b) => b - a)[0];
  }
});
console.log(zn.max(zn.data));

let ng = {
  lessons: { js: 100, php: 78, node: 78, linux: 125 }
};
console.log(zn.__proto__.max.call(ng, Object.values(ng.lessons)));
```

因为 `Math.max` 就是获取最大值的方法，所以代码可以再次优化

```
let zn = {
  data: [1, 2, 3, 4, 5]
};
console.log(Math.max.apply(null, Object.values(zn.data)));

let ng = {
  lessons: { js: 100, php: 78, node: 78, linux: 125 }
};
console.log(Math.max.apply(ng, Object.values(ng.lessons)));
```

下面是获取设置了 `class` 属性的按钮，但DOM节点不能直接使用数组的`filter` 等方法，但借用数组的原型方法就可以操作了。

```
<body>
  <button message="楠哥" class="red">楠哥</button>
  <button message="zncms">zncms</button>
</body>
<script>
  let btns = document.querySelectorAll("button");
  btns = Array.prototype.filter.call(btns, item => {
    return item.hasAttribute("class");
  });
</script>
```

##

六、prototype

函数也是对象也有原型，函数有 `prototype` 属性指向他的原型

为构造函数设置的原型指，当使用构造函数创建对象时把这个原型赋予给这个对象

```
function User(name) {
  this.name = name;
}
User.prototype = {
  show() {
    return this.name;
  }
};
let ng = new User("楠哥");
console.log(ng.show());
```

函数默认`prototype`指包含一个属性`constructor`的对象，`constructor`指向当前构造函数

```
function User(name) {
  this.name = name;
}
let ng = new User("楠哥");
console.log(ng);
console.log(User.prototype.constructor == User); //true
console.log(ng.__proto__ == User.prototype); //true

let lisi = new ng.constructor("李四");
console.log(lisi.__proto__ == ng.__proto__); //true
```

原型中保存引用类型会造成对象共享属性，所以一般只会在原型中定义方法。

```
function User() {}
User.prototype = {
  lessons: ["JS", "VUE"]
};
const lisi = new User();
const wangwu = new User();

lisi.lessons.push("CSS");

console.log(lisi.lessons); //["JS", "VUE", "CSS"]
console.log(wangwu.lessons); //["JS", "VUE", "CSS"]
```

为Object原型对象添加方法，将影响所有函数

```
<body>
  <button onclick="this.hide()">楠哥</button>
</body>
<script>
  Object.prototype.hide = function() {
    this.style.display = "none";
  };
</script>
```

了解了原型后可以为系统对象添加方法，比如为字符串添加了一截断函数。

- 不能将系统对象的原型直接赋值

```
String.prototype.truncate = function (len = 5) {
    return this.length <= len ? this : this.substr(0, len) + '...';
}
console.log('楠哥每天不断视频教程'.truncate(3)); //楠哥...
```

七、Object.create

使用`Object.create`创建一个新对象时使用现有对象做为新对象的原型对象

使用`Object.create` 设置对象原型

```
let user = {
  show() {
    return this.name;
  }
};

let zn = Object.create(user);
zn.name = "楠哥";
console.log(zn.show());
```

强以在设置时使用第二个参数设置新对象的属性

```
let user = {
  show() {
    return this.name;
  }
};
let zn = Object.create(user, {
  name: {
    value: "楠哥"
  }
});
console.log(zn);
```

八、__proto__

在实例化对象上存在 **proto** 记录了原型，所以可以通过对象访问到原型的属性或方法。

- `__proto__` 不是对象属性，理解为`prototype` 的 `getter/setter` 实现，他是一个非标准定义
- `__proto__` 内部使用`getter/setter` 控制值，所以只允许对象或null
- 建议使用 `Object.setPrototypeOf` 与`Object.getProttoeypOf` 替代 `__proto__`

下面修改对象的 `__proto__` 是不会成功的，因为`_proto__` 内部使用`getter/setter` 控制值，所以只允许对象或null

```
let ng = {};
ng.__proto__ = "楠哥";
console.log(ng);
```

下面定义的`__proto__` 就会成功，因为这是一个极简对象，没有原型对象所以不会影响`__proto__`赋值。

```
let zn = Object.create(null);
zn.__proto__ = "楠哥";
console.log(zn); //{__proto__: "楠哥"}
```

下面通过改变对象的 `__proto__` 原型对象来实现继承，继承可以实现多层,

```
let zn = {
  name: "楠哥"
};
let xinzhi = {
  show() {
    return this.name;
  }
};
let ng = {
  handle() {
    return `用户: ${this.name}`;
  }
};
xinzhi.__proto__ = ng;
zn.__proto__ = xinzhi;
console.log(zn.show());
console.log(zn.handle());
console.log(zn);
```

构造函数中的 `__proto__` 使用

```
function User(name, age) {
  this.name = name;
  this.age = age;
}
User.prototype.show = function () {
    return `姓名:${this.name}，年龄:${this.age}`;
};
let lisi = new User('李四', 12);
let xiaoming = new User('小明', 32);
console.log(lisi.__proto__ == User.prototype); //true
```

可以使用 `__proto__` 或 `Object.setPrototypeOf` 设置对象的原型，使用`Object.getProttoeypOf` 获取对象原型。

```
function Person() {
  this.getName = function() {
    return this.name;
  };
}
function User(name, age) {
  this.name = name;
  this.age = age;
}
let lisi = new User("李四", 12);
Object.setPrototypeOf(lisi, new Person());
console.log(lisi.getName()); //李四
```

对象设置属性，只是修改对象属性并不会修改原型属性，使用`hasOwnProperty`判断对象本身是否含有属性并不会检测原型。

```
function User() {}
const lisi = new User();
const wangwu = new User();

lisi.name = "小明";
console.log(lisi.name);
console.log(lisi.hasOwnProperty("name"));

//修改原型属性后
lisi.__proto__.name = "张三";
console.log(wangwu.name);

//删除对象属性后
delete lisi.name;
console.log(lisi.hasOwnProperty("name"));
console.log(lisi.name);
```

使用`in`会检测原型与对象，而`hasOwnProperty`只检测对象，所以结合后可判断属性是否在原型中

```
function User() {
}
User.prototype.name = "楠哥";
const lisi = new User();
//in会在原型中检测
console.log("name" in lisi);
//hasOwnProperty 检测对象属性
console.log(lisi.hasOwnProperty("name"));
```

使用建议

通过前介绍我们知道可以使用多种方式设置原型，下面是按时间顺序的排列

1. `prototype`构造函数的原型属性
2. `Object.create`创建对象时指定原型
3. `__proto__`声明自定义的非标准属性设置原型，解决之前通过`Object.create`定义原型，而没提供获取方法
4. `Object.setPrototypeOf`设置对象原型

这几种方式都可以管理原型，一般以我个人情况来讲使用`prototype`更改构造函数原型，使用`Object.setPrototypeOf`与`Object.getPrototypeOf`获取或设置原型。

第八章 正则表达式

一、什么是正则表达式

正则表达式是由一个字符序列形成的搜索模式，搜索模式可用于文本搜索和文本替换以及文本检测。

二、创建正则表达式

JS正则表达式的创建有两种方式： `new RegExp()` 和 直接字面量。

```
var re=new RegExp ();
//RegExp 是JS中的类，同Array类似。然而这个创建方法没有指定表达式内容
re=new RegExp ("a");
//最简单的正则表达式，将匹配字母a
re=new RegExp ("a","i");
//重载的构造函数，其第二个参数指定将不区分大小写
```

其中，对于第二个参数，为可选参数，常用的有：

- `g`：全文查找；
- `i`：不区分大小写；

然而更为常见的正则表达式创建法则是：字面量的声明方式。 即：

```
var re=/a/i;
//其作用同：re=new RegExp ("a","i")，而且更常用。
```

三、正则表达式本身的方法

在 JavaScript 中，RegExp 对象是一个预定义了属性和方法的正则表达式对象。

- `test()` 方法
  `test()` 方法用于检测一个字符串是否匹配某个模式，如果字符串中含有匹配的文本，则返回 `true`，否则返回 `false`。

```
//以下实例用于搜索字符串中的字符 "e"：
var patt = /e/;
patt.test("The best things in life are free!");
字符串中含有 "e"，所以该实例输出为：
true
```

```
以上两行代码可以合并为一行：
/e/.test("The best things in life are free!")
```

- `exec()` 方法
  `exec()` 方法用于检索字符串中的正则表达式的匹配。
  该函数返回一个数组，其中存放匹配的结果。如果未找到匹配，则返回值为 `null`。
  以下实例用于搜索字符串中的字母 `e`：

```
/e/.exec("The best things in life are free!");
字符串中含有 "e"，所以该实例输出为：
e
```

比较常用的就是正则表达式的`test`方法了，因为大多只需要知道：某个字符串是否匹配某正则表达式，是则`True`，否则为`False`。

四、字符串对象中与正则表达式有关的方法

在 JavaScript 中，正则表达式通常用于两个字符串方法：`search()`和 `replace()`。

- ```
  search()
  ```

  方法

  1. 检索与正则表达式相匹配的子字符串，并返回子串的起始位置。

```
//使用正则表达式搜索 "Runoob" 字符串，且不区分大小写：
var str = "Visit Runoob!";
var n = str.search(/Runoob/i);
//输出结果为：
6
```

1. 用于检索字符串中指定的子字符串。
   `search`方法可使用字符串作为参数。字符串参数会转换为正则表达式：

```
//检索字符串中 "Runoob" 的子串：
var str = "Visit Runoob!";
var n = str.search("Runoob");
```

- `replace()` 方法
  `replace()` 方法将接收字符串作为参数：

```
var str="Visit W3CSchool!";
var n=str.replace("W3CSchool","Runoob");
console.log(n);
//结果Visit Runoob!
console.log(str);
//结果Visit W3CSchool!
```

五、常用语法分析

5.1 ()、[]、{} 的区别

先看个例子
校验字符串是否全由8位数字组成
```

```
function isStudentNo(str) {
    var reg=/^[0-9]{8}$/;   /*定义验证表达式*/
    return reg.test(str);     /*进行验证*/
}
```

`[]`是定义匹配的字符范围。`[0-9]`表示查找任何从 0 至 9 的数字。

`{}` 一般用来表示匹配的长度。`{8}`表示位数为8位。

`()` 的作用是提取匹配的字符串。表达式中有几个`()`就会得到几个相应的匹配字符串。比如`(\s+)`表示连续空格的字符串。

5.2 ^ 和 $

`^` 匹配一个字符串的开头，比如 (`^a`) 就是匹配以字母`a`开头的字符串

`$` 匹配一个字符串的结尾,比如 (`b$`) 就是匹配以字母`b`结尾的字符串

```
^`还有另个一个作用就是取反，比如`[^xyz]`表示匹配的字符串不包含`xyz
```

需要注意的是：如果`^`出现在`[]`中一般表示取反，而出现在其他地方则是匹配字符串的开头。

5.3 \d \s \w .

`\d` 匹配一个非负整数， 等价于 `[0-9]`;

`\s` 匹配一个空白字符;

`\w` 匹配一个英文字母或数字，等价于`[0-9a-zA-Z]`;

`.` 匹配除换行符以外的任意字符，等价于`[^\n]`。

5.4 * + ?

`*`表示匹配前面元素0次或多次，比如`(\s*)`就是匹配0个或多个空格;

`+` 表示匹配前面元素1次或多次，比如`(\d+)`就是匹配由至少1个整数组成的字符串;

`?`表示匹配前面元素0次或1次，相当于`{0,1}`，比如`(\w?)` 就是匹配最多由1个字母或数字组成的字符串 。

六、语法大全

修饰符
修饰符用于执行区分大小写和全局匹配:

修饰符	描述
i	执行对大小写不敏感的匹配。
g	执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
m	执行多行匹配。

方括号
方括号用于查找某个范围内的字符:

表达式	描述	
`[abc]`	查找方括号之间的任何字符。	
`[^abc]`	查找任何不在方括号之间的字符。	
`[0-9]`	查找任何从 0 至 9 的数字。	
`[a-z]`	查找任何从小写 a 到小写 z 的字符。	
`[A-Z]`	查找任何从大写 A 到大写 Z 的字符。	
`[A-z]`	查找任何从大写 A 到小写 z 的字符。	
`[adgk]`	查找给定集合内的任何字符。	
`[^adgk]`	查找给定集合外的任何字符。	

元字符

元字符（`Metacharacter`）是拥有特殊含义的字符：

元字符	描述
`.`	**查找单个字符，除了换行和行结束符。**
`\w`	**查找单词字符。**
`\W`	**查找非单词字符。**
`\d`	**查找数字。**
`\D`	**查找非数字字符。**
`\s`	**查找空白字符。**
`\S`	**查找非空白字符。**
`\b`	匹配单词边界。
`\B`	匹配非单词边界。
`\0`	查找 NULL 字符。
`\n`	查找换行符。
`\f`	查找换页符。
`\r`	查找回车符。
`\t`	查找制表符。
`\v`	查找垂直制表符。
`\xxx`	查找以八进制数 xxx 规定的字符。
`\xdd`	查找以十六进制数 dd 规定的字符。
`\uxxxx`	查找以十六进制数 xxxx 规定的 Unicode 字符。

量词

量词	描述
`n+`	匹配任何包含至少一个 `n`的字符串。例如，`/a+/` 匹配 `candy`中的 `a`，`caaaaaaandy` 中所有的`a`。
`n*`	匹配任何包含零个或多个 `n` 的字符串。例如，`/bo*/`匹配 `A ghost booooed` 中的 `boooo`，`A bird warbled` 中的 `b`，但是不匹配 `A goat grunted`。
`n?`	匹配任何包含零个或一个 `n` 的字符串。例如，`/e?le?/` 匹配 `angel` 中的 `el`，`angle` 中的 `le`。
`n{X}`	匹配包含`X` 个`n`的序列的字符串。例如，`/a{2}/`不匹配 `candy,` 中的 `a`，但是匹配 `caandy,` 中的两个 `a`，且匹配 `caaandy.` 中的前两个 `a`。
`n{X,}`	`X` 是一个正整数。前面的模式 `n` 连续出现至少`X` 次时匹配。例如，`/a{2,}/`不匹配 `candy` 中的 `a`，但是匹配 `caandy` 和 `caaaaaaandy.` 中所有的 `a`。
`n{X,Y}`	`X`和 `Y` 为正整数。前面的模式`n` 连续出现至少 `X`次，至多 `Y`次时匹配。例如，`/a{1,3}/` 不匹配 `cndy`，匹配 `candy,` 中的 `a`，`caandy,` 中的两个 `a`，匹配 `caaaaaaandy` 中的前面三个 `a`。注意，当匹配 `caaaaaaandy` 时，即使原始字符串拥有更多的 `a`，匹配项也是 `aaa`。
`n{X,}`	匹配包含至少 `X` 个`n` 的序列的字符串。
`n$`	匹配任何结尾为 `n`的字符串。
`^n`	匹配任何开头为`n`的字符串。
`?=n`	匹配任何其后紧接指定字符串`n` 的字符串。
`?!n`	匹配任何其后没有紧接指定字符串 `n` 的字符串。

七、常用的js正则表达式

```
1.验证用户名和密码：”^[a-zA-Z]\w{5,15}$”

2.验证电话号码：（”^\d{3,4}-\d{7,8}$”）

eg：021-68686868 0511-6868686；

3.验证手机号码：”^1[3|4|5|7|8][0-9]\d{8}$”；

4.验证身份证号（15位或18位数字）：”\d{14}[[0-9],0-9xX]”；

5.验证Email地址：(“^\w+([-+.]\w+)@\w+([-.]\w+).\w+([-.]\w+)*$”)；

6.只能输入由数字和26个英文字母组成的字符串：(“^[A-Za-z0-9]+$”) ；

7.整数或者小数：^[0-9]+([.]{0,1}[0-9]+){0,1}$

8.只能输入数字：”^[0-9]*$”。

9.只能输入n位的数字：”^\d{n}$”。

```

10.只能输入至少n位的数字：”^\d{n,}$”。

11.只能输入m~n位的数字：”^\d{m,n}$”。

12.只能输入零和非零开头的数字：”^(0|[1-9][0-9]*)$”。

13.只能输入有两位小数的正实数：”^[0-9]+(.[0-9]{2})?$”。

14.只能输入有1~3位小数的正实数：”^[0-9]+(.[0-9]{1,3})?$”。

15.只能输入非零的正整数：”^+?[1-9][0-9]*$”。

16.只能输入非零的负整数：”^-[1-9][]0-9”*$。

17.只能输入长度为3的字符：”^.{3}$”。

18.只能输入由26个英文字母组成的字符串：”^[A-Za-z]+$”。

19.只能输入由26个大写英文字母组成的字符串：”^[A-Z]+$”。

20.只能输入由26个小写英文字母组成的字符串：”^[a-z]+$”。

21.验证是否含有%&',;=?$\”等字符：”[%&',;=?$\x22]+”。

22.只能输入汉字：”^[\u4e00-\u9fa5]{0,}$”。

23.验证URL：”^http://([\w-]+.)+[\w-]+(/[\w-./?%&=]*)?$”。

24.验证一年的12个月：”^(0?[1-9]|1[0-2])$”正确格式为：”01”～”09”和”10”～”12”。

25.验证一个月的31天：”^((0?[1-9])|((1|2)[0-9])|30|31)$”正确格式为：”01”～”09”、”10”～”29”和“30”~“31”。

26.获取日期正则表达式：[\d{4](file:////d%7B4)}[年|-|.]\d{\1-\12}[月|-|.]\d{\1-\31}日?

评注：可用来匹配大多数年月日信息。

27.匹配双字节字符(包括汉字在内)：[^\x00-\xff]

评注：可以用来计算字符串的长度（一个双字节字符长度计2，ASCII字符计1）

28.匹配空白行的正则表达式：\n\s*\r

评注：可以用来删除空白行

29.匹配HTML标记的正则表达式：<(\S?)[^>]>.?</>|<.? />

评注：网上流传的版本太糟糕，上面这个也仅仅能匹配部分，对于复杂的嵌套标记依旧无能为力

30.匹配首尾空白字符的正则表达式：^\s|\s$

评注：可以用来删除行首行尾的空白字符(包括空格、制表符、换页符等等)，非常有用的表达式

31.匹配网址URL的正则表达式：[a-zA-z]+://[^\s]*

评注：网上流传的版本功能很有限，上面这个基本可以满足需求

32.匹配帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)：^[a-zA-Z][a-zA-Z0-9_]{4,15}$

评注：表单验证时很实用

33.匹配腾讯QQ号：[1-9][0-9]{4,}

评注：腾讯QQ号从10 000 开始

34.匹配中国邮政编码：[1-9]\d{5}(?!\d)

评注：中国邮政编码为6位数字

35.匹配ip地址：((2[0-4]\d|25[0-5]|[01]?\d\d?).){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)。