JavaScript

- ECMAScript
- DOM
- BOM

一、特点

1. **解释型语言**
2. 类似于 C 和 Java 的语法结构
3. 动态语言
4. **面向对象**

console.log("向浏览器控制台输出");

二、 编写位置

1. 直接写到 .html 文件中（耦合，不常用）

```
<button onclick="alert('js test');">js test</button>

<a href="javascript:alert('js test');">js test</a>
<a href="javascript:;">js test</a> <!-- 将超链接失效 -->

<script>
    /*
    js多行注释
    */
    /*
     *
     */
    // 单行注释
    alert("js test");
</script>
```

2. 写到外部 .js 文件中，通过 script 标签引入（一旦引入文件，该标签不能再写内部语句），可以在不同的页面中同时使用，也可以利用到浏览器的缓存机制（推荐）

```
<script type="text/javascript" src="js/test.js"></script>
```

三、基本语法

1. 严格区分大小写
2. 每一条语句以分号结尾，如果不加，浏览器会自动加上，但可能加错位置，且消耗资源
3. 会忽略多个空格和换行，可以用空格和换行进行格式化代码

四、字面量和变量

1. 字面量/常量，可以直接使用
2. var 声明变量，变量需要初始化才能使用

五、数据类型

1. String 字符串
2. Number 数值（Infinity NaN 也是数值）
3. Boolean 布尔值（true or false）
4. null 空对象
5. undefined 未定义
6. Object 对象（引用数据类型）

注：使用 typeof 检查数据类型，返回数据类型名称的字符串。

> 千万不要使用 JS 进行精确的浮点数运算

六、数据转换

（1）转换成字符串 String

1. 调用 Number、Boolean 和 Object 的 toString() 方法

> null 和 undefined 的对象不能使用 toString 方法

2. 调用 String() 函数

> null 和 undefined 的对象会返回 "null" 和 "undefined"，其他会自动调用 toString() 方法

（2）转换成数值 Number

1. 调用 Number() 函数

> - String --> Number
>   - 纯数字字符串直接转换成数字
>   - 有非数字内容转换为 NaN
>   - 空串/空格串转换为 0
> - Boolean --> Number
>   - true --> 1
>   - false --> 0
> - null --> Number 0
> - undefined --> Number NaN

2. 调用 parseInt() 和 parseFloat() 函数，将字符串中**第一段有效**数字转换为数值

> - 对非 String 类型数据先转换成 String 再操作
> - 为了避免有些浏览器将将八进制和十进制混淆，有时需要加上第二个参数限定进制
> - 由于 JS 中 Number 类型数据包含整数和浮点数等，可用 parseInt() 向下取整

（3）转换成 Boolean

调用 Boolean() 函数。

> - Number --> Boolean 除了 0 和 NaN 其余都转成 true
> - String --> Boolean 除了空串 "" 其余都转成 true
> - null 和 undefined 都会转成 false，对象转换为 true

七、运算

- 与 Java 规则基本相同，NaN 与任何数值相加的结果都为 NaN
- **字符串**与其他数据类型**相加**优先拼串操作，其余的非数据类型进行算术运算优先转换为 Number 然后操作
- 非布尔值的**与或**运算，先将左右两值转换为 Boolean 类型然后操作，并返回**原值**，注意短路与断路规则
- 任何值与 NaN 比较都返回 false，两边都是字符串则比较两边字符 Unicode 编码，一般需要**强转**
- 相等比较会将两边的非数字转换成数字进行比较。undefined 衍生自 null，两者比较是 true。NaN 不与任何值包括自己相等，isNaN() 函数可以判断是否为 NaN
- 全等 === 用来判断两个值是否完全相等，不会进行类型转换，类型不同直接返回 false（!== 同理）

prompt()

弹出带有描述信息的输入框，返回值为输入框内容。

```
var content = prompt("pls input sth");
```

break 和 continue

默认可以跳出当前最近的循环/条件分支，如果后接 label 名称可以跳出整块标记循环

```
outer:
for (var i = 0; i < height; i++) {
    for (var j = 0; j < width; j++) {
        if (j > 10) {
            break outer;
        }
    }
}
```

开启计时器 console.time("test");

停止计时器并输出时间 console.timeEnd("test");

八、对象

只要不是五种基本数据类型都可以看作是对象。对象的属性可以是任意数据类型，包括对象在内。

（1）对象种类

1. 内建对象

- 由 ES 标准中定义的对象，在任何的 ES 实现中都可以使用

  比如：Math、String、Number、Function、Object 等等

2. 宿主对象

- 由 JS 的运行环境提供的对象，多指浏览器提供的对象

  比如：BOM、DOM 等

3. 自定义对象

- 由开发人员自己创建的对象（主要）

```
// 1.创建对象
var obj = new Object();

// 2.向对象中添加/修改属性 name
obj.name = "name";
obj["123"] = 123; // 不推荐使用不规范的命名方式，虽然它也可以正常使用

// 3.读取对象中的属性 name （如果没有初始化，则返回 undefined）
console.log(obj.name);
console.log(obj["123"]);

var a = "123";
console.log(obj[a]);

// 4.删除对象中的属性 name 【delete 关键字】
delete obj.name;
```

推荐使用 obj[xxx] 的形式去操作属性，更加灵活，可以直接传递一个变量。

- 使用 **in** 关键字可以判断对象内是否包含某属性（包括原型对象内），包含则返回 true，否则返回 false：

```
// 语法："属性名" in 对象
if ("name" in obj) {
    console.log("I have a name.")
}
```

- 使用对象的 **hasOwnProperty()** 方法来检查自身是否含有某属性：

```
function MyClass() {

}
MyClass.prototype.name = "原型中的name";
var mc = new MyClass();
mc.age = 22;

console.log("name" in mc); // true
console.log(mc.hasOwnProperty("name")); // false
console.log(mc.hasOwnProperty("age")); // true
```

（2）引用数据类型

- JS 中的变量都是保存到**栈内存**中
  - 基本数据类型的值直接保存到栈内存中
  - 值与值之间相互独立，互不干扰，两值复制为值传递
- 对象保存到**堆内存**中，每新建一个对象，就会在内存中开辟出一个新的空间
  - 而 var obj = new Object(); 的 obj 是变量，保存在栈内存中，值存储的是对象的内存地址（引用）
  - 如果两个变量同时指向一个对象，通过其中一个修改对象的属性，另一个也会受到影响

```
// 新建一个对象
var obj1 = new Object();
obj1["name"] = "luwx";
// 指向同一个对象
var obj2 = obj1;

console.log(obj1.name); // 控制台输出 luwx
console.log(obj2.name); // 控制台输出 luwx

// 通过 obj2 修改 name 的值
obj2.name = "xiehm";

console.log(obj1.name); // 控制台输出 xiehm
console.log(obj2.name); // 控制台输出 xiehm
```

（3）对象字面量

推荐使用对象字面量 “{}” 创建对象，同时初始化属性：

```
var obj = {
    name: "luwx",
    age: 22,
    obj2: {
        name: "xiehm",
        age: 23
    },
    "!@#%!@": "123123"
};
```

> 注意最后一个属性后不要加逗号 “,” 以及大括号结束要加分号 “;”

九、函数

- 函数也是一个对象
- 封装一些代码，需要时调用
- typeof 检查函数对象返回 function

（1）创建函数

实际很少使用构造方法创建函数对象，而使用函数声明。

```
/**
```

```
 * 1.语法:
 * function 函数名(参数) {
 *     语句...
 * }
 */
function fun1() {
    console.log("new function");
}
// 调用
fun1();

/**
 * 2.语法:
 * var 函数名 = function(参数) {
 *     语句...
 * }
 */
var fun2 = function(a, b) {
    return a + b;
};
fun2(); // NaN
var result = fun2(1, 2);
```

(2) 调用函数

- 调用函数时解析器不会检查实参类型，需要注意非法参数控制
- 解析器不会检查实参数量，多余实参不会被使用/赋值，如果传入的实参少于声明的数量，没有对应实参的形参将是 undefined
- 实参可以是**任意数据类型**
- 函数默认返回 undefined

(3) 对象方法

函数作为对象属性。

```
var obj = {
    name: "luwx",
    age: 22,
    say: function(){
        console.log("hello " + name);
    },
    sleep: function(n) {
        console.log("sleep " + n + "s");
    }
}
```

(4) for (...in...) {} 语句遍历对象属性

```
for (var a in obj) {
    console.log(a);
    console.log(obj[a]);
}
```

(5) 作用域

1. **全局作用域**

- 直接编写在 <script> 标签中的 JS 代码
- 在页面打开时创建，页面关闭时销毁
- 全局作用域中有一个全局对象 **window**，它代表一个浏览器窗口，由浏览器创建，可直接使用。在全局作用域中：
    - 创建的**变量**会作为 window 的属性
    - 创建的**函数**会作为 window 的方法（函数和方法即无本质区别）
    - 全局作用域中的变量都是全局变量，页面任意位置都可以访问
        - 被 **var** 标记声明的变量会优先其他代码创建声明（仅限该作用域），然后再在对应行数进行赋值
        - 无论在什么地方，使用 **function fun() {}** 声明的函数，解析器会优先创建，即可以在声明代码之前就调用该函数；而 var fun = function() {} 的方式则只会优先创建 fun 变量而不能创建函数代码，即无法在该代码之前调用 fun 函数

2. **函数作用域（局部）**

- 调用函数时创建函数作用域，执行完毕后销毁函数作用域
- 每调用一次就会创建一个函数作用域，相互独立
- 函数作用域中**可以**访问到全局变量，全局作用域中无法访问函数变量
- 使用变量时，优先寻找自身作用域中的变量，若无法找到则向**上一级**作用域中寻找，直到查找到全局作用域中，若依然无法找到，报 is not defined 错误
- 在函数中，如果局部和全局有同名变量，可用 **window.xxx** 访问全局变量
- 函数作用域中，**var** 和 **函数声明** 的提前声明依然**有效**
- 在函数中，**不使用 var** 声明的变量在**函数执行后**会转换成**全局变量**
- 形参相当于在函数中 var 一个变量但没有赋值

```
// ====== test 1 ======
var n = 2;
function fun1() {
    console.log(n); // 2
    n = 4;
}
fun1();
console.log(n); // 4
// ====== test 2 ======
var n = 2;
function fun2() {
    console.log(n); // undefined
    var n = 4;
    a = 10; // 没有使用 var 声明，转换成全局
}
fun2();
console.log(n); // 2
```

```
console.log(a); // 10
// ====== test 3 ======
var b = 123;
function fun3(b) {
    console.log(b); // 123
    b = 456; // 修改形参，对外部变量无影响
}
fun3(b);
console.log(b); // 123
```

3. **this 关键字**

解析器在调用函数时，每次都会向函数内部传入一个隐含参数 this，this 指向函数执行上下文：

- 以函数形式调用时，this 永远都是 window 对象
- 以对象方法调用时，this 指向调用对象
- 当以构造函数形式调用，this 指向新创建的对象
- 以 call() 和 apply() 调用时，this 指向传入的对象

（6）构造函数

构造函数也是普通的函数，习惯上首字母大写，区别是调用方式的不同，构造函数需要使用 new 关键字调用。执行流程：

1. 立即创建一个新的对象
2. 将新建的对象设置为函数中的 **this**，在构造函数中可以使用 this 引用到新对象
3. 逐行执行构造函数中的代码
4. 将新建的对象作为返回值返回

```
// 构造函数语法：
function Person(name) {
    this.name = name;
}
var luwx = new Person("luwx");
console.log(luwx); // Person {name: "luwx"}
console.log(luwx instanceof Person); // true
console.log(luwx instanceof Object); // true
```

> Object 是所有对象的父类

（7）原型对象（类似继承）

创建的每一个函数，解析器都会向函数中添加一个属性 **prototype**（typeof 返回 Object），即**原型对象。**如果作为普通函数调用该属性没有任何作用，当函数以**构造函数**形式调用时，它所创建的对象中会有一个隐含属性 __proto__，指向该构造函数的原型对象。可以将对象中**共有的内容**统一设置到原型对象中。

```
function MyClass() {}
var mc1 = new MyClass();
var mc2 = new MyClass();
console.log(MyClass.__proto__ == Function.prototype); // true
console.log(mc1.__proto__ == MyClass.prototype); // true
console.log(mc1.__proto__ == mc2.__proto__); // true
```

原型对象相当于一块**公共区域**，当访问对象的一个属性或方法时，会先在对象自身中寻找，如果有则直接使用，否则会去**原型对象中寻找（递归）**，有则使用无则返回 undefined。

创建构造函数时，可以将**对象共有的属性和方法**，统一添加到**构造函数的原型对象**中，这样既可以重复使用，也不需要每次创建对象时都创建重复方法的空间，**节约资源。**例如，修改原型对象中的 toString() 方法比较常见。

> **Object** 是所有对象的原型，故 Object **没有**原型对象，prototype 属性为 null。

（8）函数对象的方法 call() 和 apply()

函数调用的其他方式，如果不传任何参数，与函数名直接加小括号调用相同，如果**第一个参数**传入一个对象 object，则可以将函数内的 this 指向传入的 object（默认函数内的 this 是 window）。

- call() 方法可以将实参在对象之后依次传递
- apply() 方法需要将实参封装到数组中且在对象后传递

```
function fun(a, b) {
    console.log(this);
    console.log(a);
    console.log(b);
}
var obj = new Object();
fun.call(obj, 1, 2);
fun.apply(obj, [1, 2]);
```

（9）arguments

在调用函数时，解析器会传入两个隐含的参数，一个是函数执行上下文 this，另一个是**封装实参的对象** arguments：

- 类数组对象，用 arguments instanceof Array 为 false
- 传入的参数保存在 arguments 中
- 可以获取长度 length、通过索引来操作数据即实参
- 既可以使用形参来操作实参，也可以通过 arguments[i] 来使用实参，能否使用实参与函数是否声明形参无关
- 包含一个属性 callee，对应当前正在执行的函数的对象

十、数组对象

数组中的元素可以是任意数据类型。

（1）创建数组对象

```
var array = new Array();
console.log(array); // "" 空
console.log(typeof array); // object

// 构造方法与字面量
var array = new Array(1, 2, 3);
var array = [];
var array = [1, 2, 3];
```

(2) 向数组中添加元素，读取元素

```
array[0] = 0;
array[1] = 1;
console.log(array[0]); // 0
conosle.log(array[10]); // undefined
```

(3) length 关键字

- 对于连续的数组，使用 length 可以获取到数组的长度（元素个数）
- 对于非连续的数组，可以获取到数组的**已定义最大**索引数 +1（尽量不要创建非连续数组）
- 修改 length 的值，大于原长度，多余部分空出，小于原长度，保留 length 值的索引，多出部分被删除
- 用 length 可以向数组最后一个位置添加元素，array[array.length]
- 通过构造方法创建的数组对象可以初始化 length 值，且该语法与初始化数组值很相似

```
var array = new Array(3);j
console.log(array); // [empty x 3] length = 3 的空数组
var array = new Array(1, 2, 3);
console.log(array); // [1, 2, 3]
```

(4) 常用数组方法

1. **push()**：该方法可以向数组末尾添加一个或多个元素，并返回新的数组长度
2. **pop()**：改方法可以删除并返回数组的最后一个元素
3. **unshift()**：向数组的开头添加一个或更多元素，并返回新的数组长度
4. **shift()**：删除并返回数组的第一个元素
5. **slice(start, end)**：从已知数组返回一个新的**数组**，包含从 start 到 end（不含）的元素，不影响原数组
    - start 必需。规定从何处开始选取。如果是**负数**，那么它规定从数组**尾部开始**算起的位置。也就是说，-1 指最后一个元素，-2 指倒数第二个元素，以此类推。
    - end 可选。规定从何处结束选取。该参数是数组片断结束处的数组下标（不含）。如果没有指定该参数，那么切分的数组包含从 start 到数组结束的所有元素。如果这个参数是**负数**，那么它规定的是从数组**尾部开始**算起的元素。
6. **splice(index, howmany, item...)**：从已知数组中**添加/删除**元素，然后以数组形式返回**被删除**的元素，会**影响原数组**
    - index 必需。整数，规定添加/删除元素的位置，使用**负数**可从数组结尾处规定位置。
    - howmany 必需。要删除的元素数量。如果设置为 0，则不会删除元素。
    - item1, ..., itemX 可选。向数组 index 位置添加的新元素（相当于替换原来删除的元素）。

> 如果第二个参数设为 0，且 item 有值，则可以向数组中间特定位置插入元素。

(5) forEach 方法（兼容 IE8.0 及以上）

- 需要一个函数作为参数作为回调函数
- 数组中有几个元素，回调函数就执行几次，每次执行时，解析器会将遍历到的元素作为实参传入，最多传入 3 个参数：

1. 遍历到的元素 value
2. 索引 index
3. 数组对象 array

```
var array = [];
// 匿名函数
array.forEach(function(value, index, object){
    console.log(value, index, object);
})
// 分开
function callback(value, index, object) {

}
array.forEach(callback);
```

(6) 其他方法

1. concat()：连接两个或更多的数组，并返回结果
2. join()：把数组的所有元素放入一个**字符串**并返回。元素通过指定的分隔符进行分隔，默认为逗号“,”
3. reverse()：颠倒数组中元素的顺序（影响原数组）
4. sort()：对数组的元素进行排序（影响原数组），默认通过 Unicode 编码进行排序，对数字进行排序一般需要重写回调函数，函数传入两个参数返回大于零的值则交换两个元素，小于等于零则不变

```
var array = [2, 3, 11, 5];
array.sort(); // [11, 2, 3, 5]
function mySort(a, b) {
    return a - b; // 升序
    // return b - a; // 降序
}
array.sort(mySort); // [2, 3, 5, 11]
```

5. toString()：与无参的 join() 相同
6. Array.isArray()：检查是否是数组对象

十一、常用工具类

(1) Date 对象

获取当前/指定时间的对象，封装了许多关于时间的方法。时间与系统所在时区有关。

```
var date = new Date();
var date2 = new Date("4/1/1996 00:00:00"); // MM/dd/yyyy HH:mm:ss

console.log(date); // 输出当前系统所在时区的时间

conosle.log(Date.now()); // now() 方法：返回时间戳（时区相关）
```

(2) Math 工具类

封装了许多关于数学计算的方法，直接 Math.function() 就可以使用：

1. ceil(x) 对数进行向上取整
2. floor(x) 对数进行向下取整
3. round(x) 把数四舍五入为最接近的整数
4. abs(x) 返回数的绝对值
5. max(x,y) 返回 x 和 y 中的最高值
6. min(x,y) 返回 x 和 y 中的最低值
7. pow(x,y) 返回 x 的 y 次幂
8. random() 返回 0 ~ 1 之间的随机数

```
// 生成 x~y 之间的随机数
var betweenXY = Math.round(Math.random() * (y - x) + x);
```

> - max 返回一系列参数（可以多于两个）中最大的值。如果没有参数，则返回 -Infinity。如果有某个参数为 NaN，或是不能转换成数字的非数字值，则返回 NaN。
> - min 返回一系列参数（可以多于两个）中最小的值。如果没有参数，则返回 Infinity。如果有某个参数为 NaN，或是不能转换成数字的非数字值，则返回 NaN。

十二、正则表达式

（1）对象

RegExp 对象表示正则表达式，它是对字符串执行模式匹配的强大工具。

```
// var reg = new RegExp("正则表达式", "匹配模式");
```

匹配模式："i" 忽略大小写、"g" 全局匹配。

使用字面量创建正则表达式：

```
// var reg = /正则表达式/匹配模式;
var reg = /abcd/i;
```

一般用构造函数创建正则表达式，更加灵活，可传入变量。

（2）使用

调用 test() 方法：

```
var reg = new RegExp("a");
console.log(reg.test("abc")); // true
```

（3）正则语法

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。
(red|blue|green)	查找任何指定的选项。

(4) 支持正则表达式的 String 方法

方法	描述	FF	IE
search	检索与正则表达式相匹配的值。（只匹配首次出现）	1	4
match	找到一个或多个正则表达式的匹配。	1	4
replace(regexp/substr, replacement)	替换与正则表达式匹配的子串。（不影响原字符串）	1	4
split	把字符串分割为字符串数组。（只全局匹配）	1	4

- 默认情况下，match 方法只会找到第一个符合要求的内容，找到以后停止检索，如果设置正则表达式匹配模式为 “g”，则可以匹配所有符合的内容，返回字符串数组（即使只查询到一个）。

```
var str = "1a2b3c4d5e6f7A8B9C";
var result = str.match(/[a-z]/gi);
console.log(result); // ["a", "b", "c", "d", "e", "f", "A", "B", "C"]
```

- replace 方法，第二个参数用空串代替，可以删除匹配到的字符。返回一个新的字符串。

(5) 量词

- 说明正则表达式匹配出现几次的字符
- 只对**前一个**内容起作用

量词	描述
n+	匹配任何包含至少一个 n 的字符串。
n*	匹配任何包含零个或多个 n 的字符串。
n?	匹配任何包含零个或一个 n 的字符串。
n{X}	匹配包含 X 个 n 的序列的字符串。
n{X,Y}	匹配包含 X 至 Y 个 n 的序列的字符串。
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。

（6）元字符

拥有特殊含义的字符。特别是对于“点”，需要转义，且字面量和构造函数创建的正则表达式转义写法有区别，构造函数需要将反斜杠再次转义。

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 NUL 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。

（7）正则表达式常用举例

1. 手机号：15550686159

   1.1 以 1 开头；

   1.2 第二位 3-9 之间；

   1.3 三位以后任意数字；

   1.4 总共 11 位纯数字。

   解：^1　[3-9]　[0-9]　{9}$

```
var phoneRegex = /^1[3-9][0-9]{9}$/;
```

2. 邮箱：weixin.lu@hand-china.com

   2.1 任意字母数字下划线（不建议太短）；

   2.2 [可选] 点或任意字母数字；

   2.3 “@”；

2.4 任意字母数字 + 点 + 任意字母（2-5 位）；

2.5 [可选] 点 + 任意字母（2-5 位）。

解：^\w{3,} (\.\w+)* @ [A-z0-9]+ (\.[A-z]{2,5}){1,2}$

```
var mailRegex = /^\w{3,}(\.\w+)*@[A-z0-9]+(\.[A-z]{2,5}){1,2}$/;
```

3. 去除头尾空格：

```
var str = "      ad  min    ";
str = str.replace(/^\s*|\s*$/g, "");
```

十三、DOM 文档对象模型

（1）什么是 DOM?

DOM 是 W3C（万维网联盟）的标准。

DOM 定义了访问 HTML 和 XML 文档的标准："W3C 文档对象模型 （DOM） 是中立于平台和语言的接口，它允许程序和脚本动态地访问和更新文档的内容、结构和样式。"

W3C DOM 标准被分为 3 个不同的部分：

- 核心 DOM - 针对任何结构化文档的标准模型
- XML DOM - 针对 XML 文档的标准模型
- HTML DOM - 针对 HTML 文档的标准模型

注：DOM 是 Document Object Model（文档对象模型）的缩写。

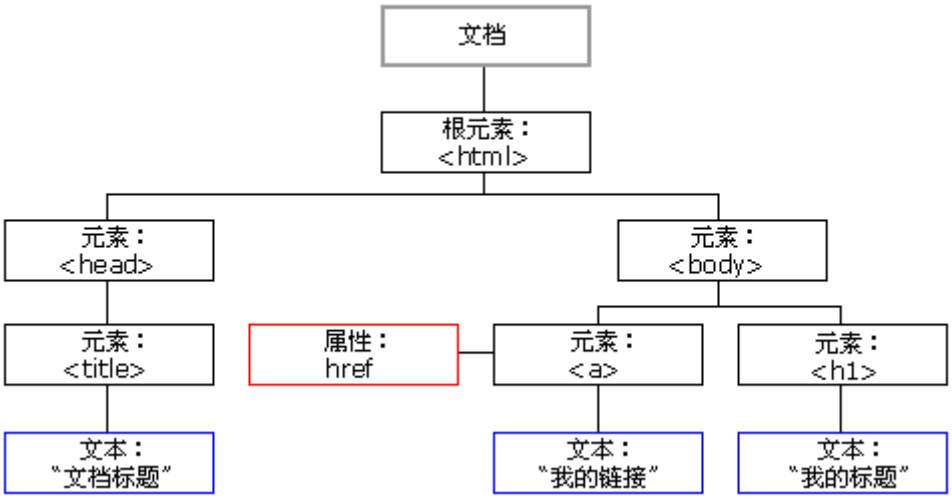
（2）DOM 节点

根据 W3C 的 HTML DOM 标准，HTML 文档中的所有内容都是节点：

- 整个文档是一个文档节点
- 每个 HTML 元素是元素节点
- HTML 元素内的文本是文本节点
- 每个 HTML 属性是属性节点
- 注释是注释节点

（3）HTML DOM 节点树

HTML DOM 将 HTML 文档视作树结构。这种结构被称为节点树：



通过 HTML DOM，树中的所有节点均可通过 JavaScript 进行访问。所有 HTML 元素（节点）均可被修改，也可以创建或删除节点。

（4）事件

用户与文档或浏览器窗口中发生的一些交互行为。

（5）节点属性

1. innerHTML：当前节点内所包含的 HTML 文档（String）
2. innerText：当前节点内所包含的文本（String）
3. nodeName：节点名称，且始终包含 HTML 元素的大写字母标签名
4. nodeValue：节点的值，例如**文本节点就是文本本身**
5. nodeType：节点类型，只读

属性	**nodeName**	**nodeType**	**nodeValue**
文档节点	#document	9	null
元素节点	标签名	1	null
属性节点	属性名	2	属性值
文本节点	#text	3	**文本内容**

（6）获取文档元素节点

建议在 window.onload 方法或文档最后的 script 标签中获取文档元素节点，保证在页面加载完成后获取到正确的元素节点。通过 document 及其子对象调用：

1. getElementById()：通过 **id** 属性获取**一个**元素节点对象
2. getElementsByTagName()：通过**标签名**获取**一组**元素节点对象
3. getElementsByName()：通过 **name** 属性获取**一组**元素节点对象

读取节点属性，一般直接使用点 + 属性名，特殊的 class 属性需要使用 className。

（7）获取元素节点的子节点

通过第（5）获取到的具体元素节点对象调用方法/访问属性：

1. getElementsByTagName()：返回当前节点的指定标签名的后代节点
2. childNodes：包含当前节点的所有子节点（不同内核的浏览器结果不一样，可能获取到换行作为一个子节点，也就是说节点数可能比后代标签多）
3. firstChild：包含当前节点的第一个子节点（可能取空格/换行 #text）
4. lastChild：包含当前节点的最后一个子节点（可能取空格/换行 #text）
5. children：包含当前节点的所有元素子节点（IE8 以下不兼容）

（8）获取父节点和兄弟节点

通过第（5）获取到的具体元素节点对象访问属性：

1. parentNode：当前节点的父节点
2. previousSibling：当前节点的前一个兄弟节点（可能取空格/换行 #text）
3. previousElementSibling：当前节点的前一个兄弟元素（IE8 以下不兼容）
4. nextSibling：当前节点的后一个兄弟节点（可能取空格/换行 #text）