1. 基本语法

1.1 编写位置

```
<!-- 1.编写位置一: 编写在html内部(了解) -->
<a href="#" onclick="alert('百度一下')">百度一下</a>
<a href="javascript: alert('百度一下')">百度一下</a>

<!-- 2.编写位置二: 编写在script元素之内 -->
<a class="google" href="#">Google一下</a>

<script>
  var googleAEl = document.querySelector(".google")
  googleAEl.onclick = function() {
    alert("Google一下")
  }
</script>

<!-- 3.编写位置三: 独立的js文件 -->
<a class="bing" href="#">bing一下</a>
<script src="./js/bing.js"></script>
```

1.2 noscript标签的使用

> 当浏览器禁用script脚本的时候，noscript标签中的内容则会被渲染到界面中

```
<noscript>
  <h1>您的浏览器不支持JavaScript, 请打开或者更换浏览器~</h1>
</noscript>

<script>
  alert("您的浏览器正在运行JavaScript代码")
</script>
```

1.3 js脚本的加载顺序

> 浏览器解析**html**时，是从上往下依次解析，当遇到script脚本时，会阻塞**dom**的解析（该脚本没有加**defer**和**async**属性时），会去下载js脚本，当js脚本下载完并执行完后才会继续构建dom。一般script元素是
>
> 作为body元素的最后子元素的。

1.4 交互方式

> 1. alert
> 2. console.log
> 3. document.write
> 4. prompt

```
<script>
  // 1.交互方式一: alert函数
  alert("Hello World");

  // 2.交互方式二: console.log函数, 将内容输出到控制台中(console)
  // 使用最多的交互方式
  console.log("Hello Coderwhy");

  // 编写的JavaScript代码出错了
  // message.length

  // 3.交互方式三: document.write()
  document.write("Hello Kobe");

  // 4.交互方式四: prompt函数, 作用获取用户输入的内容
  var result = prompt("请输入你的名字: ");
  alert("您刚才输入的内容是:" + result);

</script>
```

1.5 注释

1. 单行注释
2. 多行注释
3. 文档注释

```
<script>
  // 1.单行注释

  // 2.多行注释
  /*
     我是一行注释
     我是另外一行注释
    */

  // 3.文档注释
  /**
     * 和某人打招呼
     * @param {string} name 姓名
     * @param {number} age 年龄
     */
  function sayHello(name, age) {
```

```
  }

  sayHello()
</script>
```

2. 变量和数据结构

2.1 变量

> 变量：变量可以看作存放数据的容器，可以想象成一个盒子。
>
> 补充: **js**中的变量可以存放任意类型的数据（在程序运行中）。变量拥有这种行为时，这种编程语言称为"**动态类型的编程语言**"
>
> 在计算机程序中，数据需要存储到变量当中。计算机程序 = 数据结构 + 算法；数据结构相当于数据，算法相当于处理数据的一些逻辑。

2.2 变量声明

> 变量必须先声明，后使用（但js中是有部分不同的）
>
> 使用var 、let、const 关键字进行声明

```
<script>

  // 定义一个变量
  // 第一步： 变量的声明（告诉js引擎接下来我要定义一个变量）
  // var关键字
  // 第二步: 变量的赋值(使用=赋值即可)
  // var currentTime = "16:00"

  // 其他的写法一:
  // var currentTime;
  // currentTime = "16:02";
  // currentTime = "17:00";

  // 其他的写法二: 同时声明多个变量(不推荐, 阅读性比较差)
  // var name, age, height
  // name = "why"
  // age = 18
  // height = 1.88
  var name = "why", age = 18, height = 1.88;

  // 补充:
  // 1.当我们打印变量时, 实际上是在打印变量中保存的值
  // 2.console.log(参数1, 参数2, 参数3..........)
  console.log(name, age, height);

</script>
```

2.3 变量的命名规则和命名规范

命名规则

> 命名规则是必须要遵守的：表示在声明变量时，我们需要给这个变量（盒子）起一个名字，后续访问的时候可以根据这个变量名来访问。

- 变量名以**数字**、**字母**、**美元符号（$）**、**下划线（_）**组成
- 不能以**数字**开头
- 不能使用**关键字**和**保留字**作为变量名
- 严格区分大小写

命名规范

> 命名规范只是大家普遍认定比较好的约束，建议遵守

- 多个单词使用驼峰标识；
- 赋值 = 两边都加上一个空格；
- 一条语句结束后加上分号; 也有很多人的习惯是不加；
- 变量应该做到见名知意；

```
<script>
  // 规则: 必须遵守

  // 规范: 建议遵守
  // 1.多个单词, 建议使用驼峰标识(小驼峰)
  // 小驼峰(currentTime)/大驼峰(CurrentTime)
  // 2.推荐=的左右两边加上空格
  var currentTime = "16:18"
  var currentPrice = "¥100"

</script>
```

2.4 变量使用注意事项

- 如果一个变量未声明（declaration）就直接使用，那么会报错；
- 如果一个变量有声明，但是没有赋值，那么默认值是undefined
- 如果没有使用var声明变量也可以，但是不推荐（事实上会被添加到window对象上）

```
<script>
  // 注意事项一: 如果一个变量未声明就直接使用, 那么会直接报错
  // var currentAge = age
  // var name = "why"
  // console.log(message) // 浏览器报错

  // 注意事项二: 如果一个变量有声明, 但是没有赋值操作
```

```
  // var info
  // console.log(info) // undefined

  // 注意事项三: 在JavaScript中也可以不使用var在全局声明一个变量(不推荐)
  // 如果不使用var来声明一个变量, 也是可以声明成功的, 并且这个变量会被加入window对象
  address = "广州市"
  console.log(address)

</script>
```

2.5 数据类型

> 什么是数据类型，js中的值都具有特定的类型。不同的数据类型有不同的用途

js中的数据类型

- Number
- String
- Boolean
- Undefined
- Null
- Symbol
- BigInt
- Object

typeof 运算符

> 因为ES的类型系统是松散的，所以需要一种手段来确定任意变量的数据类型。**typeof**运算符就运应而生了

对一个值使用 typeof 操作符会返回下列字符串之一

- "undefined"表示值未定义
- "boolean"表示值为布尔值
- "string"表示值为字符串
- "number"表示值为数值
- "object"表示值为对象(而不是函数)或 null
- "function"表示值为函数
- “symbol”表示值为符号

Number类型

> 数字类型的值，表示整数和小数

- 基本使用

```
// 1.Number类型的基本使用
var age = 18
var height = 1.88
```

- 特殊值

```
// 2.特殊的数值
// Infinity 表示正无穷大，-Infinity表示负无穷大
var num1 = Infinity
var num2 = 1 / 0
console.log(num1, num2)
```

- NaN 不是一个数字

> typeof NaN 为 'number'，isNaN函数用来检测一个值是不是NaN

```
// NaN: not a number(不是一个数字)
var result = 3 * "abc"
console.log(result) // NaN
console.log(isNaN(result)) // true
```

- 进制表示

```
// 3.进制表示
var num3 = 100 // 十进制
// 了解
var num4 = 0x100 // 十六进制
var num5 = 0o100 // 八进制
var num6 = 0b100 // 二进制
console.log(num4, num5, num6) // 256 64 4
```

- 值表示范围

```
// 5.数字可以表示的范围
var max = Number.MAX_VALUE
var min = Number.MIN_VALUE
console.log(max, min) // 1.7976931348623157e+308 5e-324
```

- 运算

> 数字类型可以值可以做各种运算
>
> 例如：+ - * / % **等

String类型

> 在开发中我们经常会有一些文本需要表示，这个时候我们会使用字符串String
>
> 例如：人的姓名：dd。地址：德阳市。简介：认真是一种可怕的力量

- **JavaScript 中的字符串必须被括在引号里，有三种包含字符串的方式。**
  - 单引号 ''
  - 双引号 ""
  - 反引号 ``（es6之后）
- **前后的引号类型必须一致：**
  - 如果在字符串里面本身包括单引号，可以使用双引号；
  - 如果在字符串里面本身包括双引号，可以使用单引号；
- **除了普通的可打印字符以外，一些有特殊功能的字符可以通过转义字符的形式放入字符串中：**
转义字符	表示符号
\'	单引号
\"	双引号
\\	反斜杠
\n	换行符
\r	回车符
\t	制表符
\b	退格符
- **<字符串>本身有的方法和属性**
  - **length**属性：表示字符串的长度
  - 字符串拼接：使用 + 符号

```
// 1.String基本使用
var name = "coderwhy"
var address = "广州市"
var intro = "认真是一种可怕的力量!"

// 2.别的引号的使用
// 单引号
var message1 = 'Hello World'
// 双引号
var message2 = "Hello World"
// 反引号(ES6新增语法)
// ${变量/表达式}
var message3 = `Hello World, ${name}, ${2 + 3}`

// 3.转义字符: 字符串本身中包含引号
var message4 = 'my name is "coderwhy"'
console.log(message4)

var message5 = 'my name \\ \'\' is "coderwhy"'
```

```
console.log(message5)

// 4.<字符串>本身有的方法和属性
var message = "Hello World"
console.log(message.length)

// 字符串操作
var nickname = "coderwhy"
var info = "my name is "
var infoStr = `my name is ${nickname}` // 推荐(babel)
var infoStr2 = info + nickname
console.log(infoStr, infoStr2)
```

Boolean类型

> 布尔类型的值一般用来做逻辑判断使用

- 布尔类型仅包含两个值：**true** 和 **false**
- Boolean（布尔）类型用于表示**真假**
- 布尔（英语：**Boolean**）是计算机科学中的逻辑数据类型，以发明布尔代数的数学家**乔治·布尔**为名

```
// 1.Boolean基本使用
var isLogin = false
var isAdmin = true
```

Undefined类型

> 如果我们声明一个变量，但是没有对其进行初始化时，它默认就是undefined；

```
var message
var info = undefined // 不推荐

console.log(message, info)
```

注意事项：

- 最好在变量**定义的时候进行初始化**，而不只是声明一个变量；
- **不要显式的将一个变量赋值为undefined**
  - 如果变量刚开始什么都没有，我们可以初始化为0、空字符串、null等值；

Object类型

> Object 类型是一个特殊的类型，我们通常把它称为**引用类型或者复杂类型**
>
> 1. 其他的数据类型我们通常称之为 “原始类型”，因为它们的值质保函一个单独的内容（字符串、数字或者其他）；
> 2. Object往往可以表示一组数据，是其他数据的一个集合；

> 3. 在JavaScript中我们可以使用 花括号{} 的方式来表示一个对象；

```
// 1.Object类型的基本使用
// var name = "why"
// var age = 18
// var height = 1.88
var person = {
  name: "why",
  age: 18,
  height: 1.88
}
console.log(person)

// 2.对象类型中某一个属性
console.log(person.name)
```

Null类型

> null类型通常用来表示一个对象为空，所以通常我们在给一个对象进行初始化时，会赋值为null；

- Null 类型同样只有一个值，即特殊值 null。
- null和undefined的关系

> 1. undefined通常只有在一个变量声明但是未初始化时，它的默认值是undefined才会用到；
> 2. 并且我们不推荐直接给一个变量赋值为undefined，所以很少主动来使用；
> 3. null值非常常用，当一个变量准备保存一个对象，但是这个对象不确定时，我们可以先赋值为null；

```
// 3.Null类型
// 3.1. 其他类型的初始化
var age = 0
var num = 0
var message = "" // 空字符串
var isAdmin = false

// Null存在的意义就是对 对象进行初始化的, 并且在转成Boolean类型时, 会转成false
var book = null
console.log(typeof book) // object
```

总结

> JavaScript 中有八种基本的数据类型（前七种为**基本数据类型，也称为原始类型**，而 object 为**复杂数据类型，也称为引用类型**）。

- number 用于任何类型的数字：整数或浮点数。
- string 用于字符串：一个字符串可以包含 0 个或多个字符，所以没有单独的单字符类型。

- boolean 用于 true 和 false。
- undefined 用于未定义的值 —— 只有一个 undefined 值的独立类型。
- object 用于更复杂的数据结构。
- null 用于未知的值 —— 只有一个 null 值的独立类型。

2.6 数据类型转换

在开发中，我们可能会在不同的数据类型之间进行某些操作

- 比如把一个String类型的数字和另外一个Number类型的数字进行运算；
- 比如把一个String类型的文本和另外一个Number类型的数字进行相加；
- 比如把一个String类型或者Number类型的内容，当做一个Boolean类型来进行判断；

也就是在开发中，我们会经常需要对数据类型进行转换

- 大多数情况下，**运算符**和**函数**会自动将赋予它们的值转换为正确的类型，这是一种**隐式转换**；
- 我们也可以，通过**显式**的方式来对数据类型进行转换；

String类型的转换

> 其他类型经常需要转换成字符串类型，比如和字符串拼接在一起或者使用字符串中的方法。

- **隐式转换**
  - 一个字符串和另一个字符串进行+操作；
    - 如果+运算符左右两边有一个是字符串，那么另一边会自动转换成字符串类型进行拼接；
  - 某些函数的执行也会自动将参数转为字符串类型；
    - 比如console.log函数；
- **显式转换**
  - 调用String()函数；
  - 调用toString()方法（面向对象再深入）

```
var num1 = 123
var age = 18
var isAdmin = true

// 1.转换方式一: 隐式转换(用的非常多)
var num1Str = num1 + ""
var ageStr = age + ""
var isAdminStr = isAdmin + ""
console.log(typeof num1Str, typeof ageStr, typeof isAdminStr)

// 2.转换方式一: 显示转换
var num1Str2 = String(num1)
console.log(typeof num1Str2)
```

Number类型的转换

> 其他类型也可能会转成数字类型。

- 隐式转换
  - 在**算数运算**中，通常**会将其他类型转换成数字类型**来进行运算；
    > 比如 "6" / "2"；
    >
    > 但是如果是+运算，并且其中一边有字符串，那么还是按照字符串来连接的；
- 显式转换
  - **使用Number()函数**来进行显式的转换；
- 其他类型转换数字的规则

值	转换后的值
undefined	NaN
null	0
true 和 false	1 and 0
string	去掉首尾空格后的纯数字字符串中含有的数字。如果剩余字符串为空，则转换结果为 0。否则，将会从剩余字符串中“读取”数字。当类型转换出现 error 时返回 NaN。

```
// 方式一: 隐式转换(用的很少)
var num1 = "8"
var num2 = "4"
var result1 = num1 + num2 // 84
console.log(typeof result1) // string

var result2 = num1 * num2
console.log(result2) // 32
```

```
// 方式二: 显示转换(Number())
var result3 = Number(num1) // 8
console.log(typeof result3) // number

// 其他类型转成数字类型的规则:
console.log(Number(undefined)) // NaN
console.log(Number(true)) // 1
console.log(Number(false)) // 0
console.log(Number(null)) // 0
console.log(Number("abc123")) // NaN
console.log(Number("")) // 0
console.log(Number("")) // 0
```

Boolean类型的转换

> 布尔（boolean）类型转换是最简单的，它发生在逻辑运算中，但是也可以通过调用Boolean(value)显式地进行转换。

- 转换规则

值	转换后
0，null，undefined，NaN，""	false
其他值	true

- **注意：包含0的字符串 "0" 是 true**

```
// 方式一: 隐式转换
// 分支语句
var isAdmin = true
var num1 = 123 // true

// 方式二: 显示转换
console.log(Boolean(num1), Boolean(undefined))

// 转换有如下的规则:
// 直观上为空的值, 转成Boolean类型都是false
// 直观上为空的值: 0/""/undefined/null/NaN -> false

// 注意事项
console.log(Boolean("")) // false
console.log(Boolean("0")) // true
```

3. 运算符

> **运算符**就是用来做运算的符号。计算机最基本的操作就是执行运算，执行运算时就需要使用运算符来操作，编程语言按使用场景将运算符分成了以下类型：算术运算符/赋值运算符/关系(比较)运算符/逻辑运算符。

3.1 运算元

> 运算元就是运算符应用的对象，如果一个运算符有**n个运算元**，那么这个运算符称为**n元运算符。**

- 比如说乘法运算 5 * 2，有**两个运算元**；
- 左运算元 5 和右运算元 2；
- 有时候人们也称其为“参数”；

3.2 算数运算符

> 算术运算符用在数学表达式中, 它的使用方式和数学中也是一致的。
>
> 算术运算符是对数据进行计算的符号。

运算符	运算规则	范例	结果
+	加法	2 + 3	5
+	连接字符串	'中' + '国'	'中国'
-	减法	2 - 3	-1
*	乘法	2 * 3	6
/	除法	6 / 2	3
%	取余	5 % 2	1
**	幂（ES7）	2 ** 3	8

- **取余运算符是 %，尽管它看起来很像百分数，但实际并无关联**

```
//  a % b 的结果是 a 整除 b 的余数
console.log(10 % 3) // 1
```

- **求幂运算 a ** b 将 a 提升至 a 的 b 次幂。（ES7中的语法，也叫做ES2016）**

```
// 在数学中我们将其表示为 a 的 b 次方。
console.log(2**3) // 8
console.log(2**4) // 16
```

3.3 赋值运算符

> = 被称之为赋值（ assignments ）运算符。

- **= 是一个运算符，而不是一个有着“魔法”作用的语言结构。**

```
var name; // undefined

// 语句 x = value 将值 value 写入 x 然后返回 x。
name = 'dd' // 'dd'
```

- **链式赋值（Chaining assignments）**

```
let a, b, c;
a = b = c = 2 + 2;
console.log(a, b, c); // 4 4 4
```

  - 链式赋值**从右到左进行计算**；
  - 首先，对最右边的表达式 2 + 2 求值，然后将其赋给左边的变量：c、b 和 a。
  - 最后，所有的变量共享一个值。

3.4 原地修改（Modify-in-place）

> 我们经常需要对一个变量做运算，并将新的结果存储在同一个变量中。

```
// 原写法
var n = 10;
n = n + 5;
n = n * 2;

// 原地修改
var n = 10;
n += 5;
n *= 2;

// 所有算术和位运算符都有简短的“修改并赋值”运算符：/= 和 -= 等。
```

运算符	运算规则	范例	结果
=	赋值	a = 5	5
+=	加后赋值	a = 5, a += 2	7
-=	减后赋值	a = 5, a -= 2	3
*=	乘后赋值	a = 5; a *= 2	10
/=	除后赋值	a = 5; a /= 2	2.5
%=	取模 (余数)后赋值	a = 5; a %= 2	1
**=	幂后赋值	a = 5; a **= 2	25

3.5 自增、自减运算符

> 对一个数进行**加一、减一**是最常见的数学运算符之一。应用于变量上的操作符
>
> ++ 和 --

- 将一个变量进行**自加1**或**自减1**的操作

```
var currentIndex = 5

// 方法一:
// currentIndex = currentIndex + 1

// 方法二:
// currentIndex += 1

// 方法三: 自增
currentIndex++
console.log(currentIndex)

// 自减
// currentIndex -= 1
currentIndex--
console.log(currentIndex)
```

++ 和 -- 的位置

> ++和--运算符可以在变量前面，也可以在变量后面。

相同点：都是将变量进行自加一或者自减1。

不同点：当自增自减表达式在**其他表达式中参与运算时**，**前置形式**返回**变量自增或自减后的值**，**后置形式**返回**原始值**。

```
var currentIndex = 5
// 自己自增或者自减是没有区别
// ++currentIndex
// console.log(currentIndex) // 6
// --currentIndex
// console.log(currentIndex) // 5

// 自增和自减表达式本身又在其他的表达式中，那就有区别
// var result1 = 100 + currentIndex++ // 100 + 5
// console.log(currentIndex)    // 6
// console.log("result1:" + result1)    // 105

var result2 = 100 + ++currentIndex // 100 + 6
console.log(currentIndex) // 6
console.log("result2:" + result2) // 106
```

3.6 比较运算符

> 用来比较大小的运算符

- 比较运算符的结果都是Boolean类型的

运算符	运算规则	范例	结果
==	相等	1 == '1'	true
===	严格相等	1 === '1'	false
!=	不相等	3 != '3'	false
!==	严格不相等	3 !== '3'	true
>	大于	4 > 3	true
<	小于	4 < 3	false
>=	大于等于	4 >= 3	true
<=	小于等于	4 <= 3	false

注意：严格相等和严格不相等运算时，当两边的值的类型不一致时，直接返回false；普通相等和不相等运算时，当两边类型不一致时，首先会将两侧的值先转化为数字，再做比较。

```
var num1 = 20
var num2 = 30

// 1.比较运算符
var isResult = num1 > num2
console.log(isResult)

// 2.==判断
console.log(num1 == num2) // false
console.log(num1 != num2) // true

// 需求: 获取到比较大的那个值
// var result = 0
// if (num1 > num2) {
//   result = num1
// } else {
//   result = num2
// }



var foo1 = 0
var foo2 = ""

// ==运算符, 在类型不相同的情况下, 会将运算元先转成Number的值, 再进行比较(隐式转换)
// null比较特殊: null在进行比较的时候, 应该是会被当成一个对象和原生类型进行比较的
console.log(Number(foo1))
console.log(Number(foo2))
```

```
console.log(foo1 == foo2)

// ===运算符，在类型不相同的情况，直接返回false
console.log(foo1 === foo2)
```

3.7 逻辑运算符

> 逻辑运算符主要有三个，都是用来做逻辑运算，它可以将**多个表达式或者值放到一起**来获取到一个**最终的结果**；
>
> &&（与）、||（或）、！（非）

运算符	运算规则	范例	结果
&&	与：同时为真	false && true	false
||	或：一个为真	false || true	true
!	非：取反	!false	true

```
var chineseScore = 88
var mathScore = 99

// 1.逻辑与: &&, 并且
// 条件1 && 条件2 && 条件3.....
// 所有的条件都为true的时候，最终结果才为true
// 案例: 小明语文考试90分以上，并且数学考试90分以上，才能去游乐场
if (chineseScore > 90 && mathScore > 90) {
    console.log("去游乐场玩~")
}

// 2.逻辑或: ||, 或者
// 条件1 || 条件2 || 条件3....
// 只要有一个条件为true，最终结果就为true
// 案例: 如果有一门成绩大于90，那么可以吃打1小时游戏
if (chineseScore > 90 || mathScore > 90) {
    console.log("打1个小时游戏~")
}

// 3.逻辑非: !, 取反
var isLogin = true
if (!isLogin) {
    console.log("跳转到登录页面")
    console.log("进行登录~")
}

console.log("正常的访问页面")
```

- **逻辑与（&&）的本质**

1. 拿到第一个运算元, 将运算元转成Boolean类型
2. 对运算元的Boolean类型进行判断
    - 如果false, 返回运算元(原始值)
    - 如果true, 查找下一个继续来运算
    - 以此类推
3. 如果查找了所有的都为true, 那么返回最后一个运算元(原始值)

```
// 本质推导一: 逻辑与, 称之为短路与
var chineseScore = 80
var mathScore = 99
if (chineseScore > 90 && mathScore > 90) {}

// 本质推导二: 对一些对象中的方法进行有值判断
var obj = {
    name: "why",
    friend: {
        name: "kobe",
        eating: function() {
            console.log("eat something")
        }
    }
}

// 调用eating函数
// obj.friend.eating()
obj && obj.friend && obj.friend.eating && obj.friend.eating()
```

- **逻辑或（||）的本质**
    1. 先将运算元转成Boolean类型
    2. 对转成的boolean类型进行判断
        - 如果为true, 直接将结果(原始值)返回
        - 如果为false, 进行第二个运算元的判断
        - 以此类推
    3. 如果找到最后, 也没有找到为真值的运算元, 那么返回最后一个运算元

```
// 本质推导一: 之前的多条件是如何进行判断的
var chineseScore = 95
var mathScore = 99
// chineseScore > 90为true, 那么后续的条件都不会进行判断
if (chineseScore > 90 || mathScore > 90) {}

// 本质推导二: 获取第一个有值的结果
var info = "abc"
var obj = {name: "why"}
var message = info || obj || "我是默认值"
console.log(message.length)
```

- 逻辑非（！）的补充
    - 逻辑非运算符接受一个参数，并按如下运算：

1. 将操作数转化为布尔类型：true/false；
2. 返回相反的值；

- 两个非运算!! 有时候用来将某个值转化为布尔类型：
  - 第一个非运算将该值转化为布尔类型并取反，第二个非运算再次取反。
  - 最后我们就得到了一个任意值到布尔值的转化。

3.8 条件运算符（也叫三元运算符）

> **条件（三元）运算符**是 JavaScript 唯一使用三个操作数的运算符：一个条件后跟一个问号（`?`），如果条件为真值，则执行冒号（`:`）前的表达式；若条件为假值，则执行最后的表达式。该运算符经常当作 `if...else` 语句的简捷形式来使用
>
> 语法：condition ? exprIfTrue : exprIfFalse
>
> - `condition`
>   - 计算结果用作条件的表达式。
> - `exprIfTrue`
>   - 如果 `condition` 的计算结果为真值（等于或可以转换为 `true` 的值），则执行该表达式。
> - `exprIfFalse`
>   - 如果 `condition` 为假值（等于或可以转换为 `false` 的值）时执行的表达式。
>
> 描述：除了 `false`，可能的假值表达式还有：`null`、`NaN`、`0`、空字符串（`""`）和 `undefined`。如果 `condition` 是其中任何一个，那么条件表达式的结果就是 `exprIfFalse` 表达式执行的结果。

```
// 案例一: 比较两个数字
var num1 = 12*6 + 7*8 + 7**4
var num2 = 67*5 + 24**2

// 三元运算符
var result = num1 > num2 ? num1: num2
console.log(result)

// 案例二: 给变量赋值一个默认值(了解)
var info = {
    name: "why"
}
var obj = info ? info: {}
console.log(obj)

// 案例三: 让用户输入一个年龄, 判断是否成年人
var age = prompt("请输入您的年龄:")
age = Number(age)
// if (age >= 18) {
```

```
//   alert("成年人")
// } else {
//   alert("未成年人")
// }
var message = age >= 18 ? "成年人": "未成年人"
alert(message)
```

3.9 运算符的优先级

> 当一个表达式中有多个运算符时，不同运算符的优先级不一样；**运算符的优先级**决定了表达式中运算执行的先后顺序。优先级高的运算符会作为优先级低的运算符的操作数。MDN

```
var num = 5
var result = 2 + 3 * ++num // 2 + 3 * 6
console.log(result) // 20
```

ps：想要优先运算：添加 **()** 分组运算符，优先级最高

4. 分支语句

> 在程序开发中，程序有三种不同的执行方式：顺序、分支、循环

- 顺序：**从上向下**，顺序执行代码
- 分支：根据**条件判断**，决定执行代码的**分支**
- 循环：让 **特定代码 重复** 执行

```
// 1.顺序执行
var num1 = 10
var num2 = 20

var result = num1 + num2
var result2 = num1 * num2

// 2.分支语句
var isLogin = true
if (isLogin) {
    console.log("访问购物车")
    console.log("访问个人中心")
} else {
    console.log("跳转到登录页面")
}

// 3.循环语句
var i = 0;
while (i < 10) {
    console.log("执行循环语句")
    i++
```

```
}
```

4.1 代码块

> **代码块**是多行执行代码的集合，通过一个**花括号{}**放到了一起。
>
> 在开发中，一行代码很难完成某一个特定的功能，我们就会将这些代码放到一个**代码块**中
>
> **在JavaScript中，我们可以通过流程控制语句来决定如何执行一个代码块：**
>
> - 通常会通过一些关键字来告知js引擎代码要如何被执行;
> - 比如**分支语句**、**循环语句**对应的关键字等;

```
// 一个代码块
{
    var num1 = 10;
    var num2 = 20;
    var result = num1 + num2;
}

// 一个对象
var info = {
    name: "why",
    age: 18
}
```

4.2 分支结构

> - 分支结构的代码就是让我们**根据条件**来决定**代码的执行**
> - 分支结构的语句被称为**判断结构**或者**选择结构**
> - 几乎**所有的编程语言都有分支结构**（C、C++、OC、JavaScript等等）
> - **JavaScript中常见的分支结构有**：**if**分支结构、**switch**分支结构

- if分支结构
  - 单分支结构

    > if(...) 语句计算括号里的条件表达式，如果计算结果是 true，就会执行对应的代码块。

    ```
    // 如果条件成立，那么执行代码块
    // if(条件判断) {
    //     // 执行代码块
    // }


    // 案例一：如果小明考试超过90分，就可以去游乐场
    // 1.定义一个变量来保存小明的分数
    var score = 99
    ```

```
// 2.如果分数超过90分，去游乐场
if (score > 90) {
    console.log("去游乐场玩~")
}


// 案例二：苹果单价5元/斤，如果购买的数量超过5斤，那么立减8元
// 1.定义一些变量保存数据
var price = 5
var weight = 7
var totalPrice = price * weight

// 2.根据购买的重量，决定是否 -8
if (weight > 5) {
    totalPrice -= 8
}

console.log("总价格:", totalPrice)


// 案例三：播放列表中 currentIndex
// ["鼓楼", "理想", "阿刁"]
var currentIndex = 2

// 对currentIndex++完操作之后
currentIndex++
if (currentIndex === 3) {
    currentIndex = 0
}



// 补充一：如果if语句对应的代码块，只有一行代码，那么{}可以省略
// 案例二：苹果单价5元/斤，如果购买的数量超过5斤，那么立减8元
// 定义一些变量保存数据
var price = 5
var weight = 7
var totalPrice = price * weight

// 2.根据购买的重量，决定是否 -8
if (weight > 5) totalPrice -= 8

console.log("总价格:", totalPrice)

// 补充二：if (…) 语句会计算圆括号内的表达式，并将计算结果转换为布尔型（Boolean）
// 转换规则和Boolean函数的规则一致；
// 数字 0、空字符串 “”、null、undefined 和 NaN 都会被转换成 false。
// 其他值被转换为 true，所以它们被称为“真值（truthy）”；
var num = 123 // true
if (num) {
    console.log("num判断的代码执行")
}
```

ps: **如果if语句对应的代码块，只有一行代码，那么{}可以省略**

- 多分支结构

> if..else..
>
> if 语句有时会包含一个可选的 “else” 块
>
> 如果判断条件不成立，就会执行它内部的代码

```
var score = 80
// 条件成立, 专属的代码块
// 条件不成立, 专属的代码块
// if (score > 90) {
//   console.log("去游乐场玩~")
// } else {
//   console.log("哈哈哈哈")
// }

// 案例一: 小明超过90分去游乐场, 否则去上补习班
if (score > 90) {
    console.log("去游乐场玩~")
} else {
    console.log("去上补习班~")
}

// 案例二: 有两个数字, 进行比较, 获取较大的数字
var num1 = 12*6 + 7*8 + 7**4
var num2 = 67*5 + 24**2
console.log(num1, num2)

var result = 0
if (num1 > num2) {
    result = num1
} else {
    result = num2
}
console.log(result)
```

> if..else if..else..
>
> 有时我们需要判断多个条件；我们可以通过使用 else if 子句实现；

```
// 案例: score评级
// 1.获取用户输入的分数
var score = prompt("请输入您的分数:")
score = Number(score)

// 2.判断等级
// 使用if else的方式来实现
// if (score > 90) {
//   alert("您的评级是优秀!")
// } else {
//   if (score > 80) {
//     alert("您的评级是良好!")
//   } else {
//   }
```

```
// }

// edge case
// if (score > 100 || score < 0) {
//   alert("您输入的分数超过了正常范围")
// }

if (score > 90) {
    alert("您的评级是优秀!")
} else if (score > 80) {
    alert("您的评级是良好!")
} else if (score >= 60) {
    alert("您的评级是合格!")
} else {
    alert("不及格!!!")
}
```

- switch分支结构

> 它是通过判断表达式的结果（或者变量）是否**等于**（这里使用 === 进行比较的）case语句的常量，来执行相应的分支体的；
>
> 与if语句不同的是，switch语句只能做值的相等判断（使用全等运算符 ===），而if语句可以做值的范围判断；
>
> switch 语句有至少一个 case 代码块和一个可选的 default 代码块。

```
// 语法
// switch (表达式/变量) {
//   case 常量1:
//     // 语句
// }


// 案例:
// 上一首的按钮: 0
// 播放/暂停的按钮: 1
// 下一首的按钮: 2
// var btnIndex = 100
// if (btnIndex === 0) {
//   console.log("点击了上一首")
// } else if (btnIndex === 1) {
//   console.log("点击了播放/暂停")
// } else if (btnIndex === 2) {
//   console.log("点击了下一首")
// } else {
//   console.log("当前按钮的索引有问题~")
// }

var btnIndex = 0
switch (btnIndex) {
    case 0:
        console.log("点击了上一首")
        break
    case 1:
        console.log("点击了播放/暂停")
        // 默认情况下是有case穿透
```

```
        break
    case 2:
        console.log("点击了下一首停")
        break
    default:
        console.log("当前按钮的索引有问题~")
        break
}
```

- **case穿透问题：**（不会对后续的case **表达式**进行求值）

```
switch (undefined) {
  case console.log(1):
  case console.log(2):
}
// 仅输出 1
```

  - 一条case语句结束后，会自动执行下一个case的语句；
  - 这种现象被称之为case穿透；
- **break关键字**
  - 通过在每个case的代码块后添加break关键字来**解决case穿透**这个问题；
- **这里的相等是严格相等（===）。**

5. 循环语句

> **循环** 是一种重复运行同一代码的方法。
>
> 如果是对某一个列表进行循环操作，我们通常也会称之为 遍历（traversal）或者迭代（iteration）；

js中的三种循环方式

- while循环；
- do...while循环；
- for循环；

5.1 while循环

```
while(循环条件) {
    // 循环代码块
}

// 死循环
while (true) {
    console.log("Hello World")
```

```
    console.log("Hello Coderwhy")
}


// 1.练习一: 打印10次Hello World
// var count = 0
// while (count < 10) {
//   console.log("Hello World:", count)
//   count++ // 10
// }

// 2.练习二: 打印0~99的数字
// var count = 0
// while (count < 100) {
//   console.log(count)
//   count++
// }

// 3.练习三: 计算0~99的数字和
// var count = 0
// var totalCount = 0
// while (count < 100) {
//   totalCount += count
//   count++
// }
// console.log("totalCount:", totalCount)

// 4.练习四: 计算0~99的奇数和
// 如何判断一个数字是奇数还是偶数
// var num = 120
// if (num % 2 !== 0) { // 奇数
//   console.log("num是一个奇数")
// }
// var count = 0
// var totalCount = 0
// while (count < 100) {
//   if (count % 2 !== 0) {
//     totalCount += count
//   }
//   count++
// }

// console.log("所有的奇数和:", totalCount)


// 5.练习五: 计算0~99的偶数和
var count = 0
var totalCount = 0
while (count < 100) {
    if (count % 2 === 0) {
        totalCount += count
    }
    count++
}

console.log("所有的偶数和:", totalCount)
```

```
// 算法优化
var count = 0
var totalCount = 0
while (count < 100) {
    totalCount += count
    count += 2
}

console.log("所有的偶数和:", totalCount)
```

5.2 do...while循环

> do..while循环和while循环非常像，二者经常可以相互替代**(不常用)**
>
> 但是do..while的特点是不管条件成不成立，do循环体都会先执行一次；

```
do {
    // 循环代码块
} while(循环条件);

// 练习一: 打印10次Hello World
var count = 0
do {
    console.log("Hello World")
    count++
} while (count < 10);

// 练习二: 计算0~99的数字和
var count = 0
var totalCount = 0
do {
    totalCount += count
    count++
} while (count < 100);
console.log("totalCount:", totalCount)
```

5.3 for 循环

> for 循环更加复杂，但它是**最常使用的循环形式。**

```
for(begin; condition; step) {
    // 循环代码块
}

for(let i = 0; i < 3; i++) {
    alert(i)
}
```

语句段	例子	执行过程
begin	let i = 0	进入循环时执行一次
condition	i < 3	在每次循环迭代之前检查，如果为false，停止循环
body（循环代码块）	alert(i)	条件为真时，重复运行
step	i++	在每次循环体迭代后执行

说明： begin 执行一次，然后进行迭代：每次检查 condition 后，执行 body 和 step

```
/*
  * 1.首先, 会先执行var count = 0;
  * 2.根据条件执行代码
    * count < 3
    * alert(count) // 0 1 2
    * count++
*/
for (var count = 0; count < 3; count++) {
    alert(count)
}

// 1.打印10次Hello World
// for (var i = 0; i < 10; i++) {
//   console.log("Hello World")
// }

// 2.打印0~99的数字
for (var i = 0; i < 100; i++) {
    console.log(i)
}

// 3.0~99的数字和
var totalCount = 0
for (var i = 0; i < 100; i++) {
    totalCount += i
}
console.log("totalCount:", totalCount)

// 4.0~99的奇数和
var totalCount = 0
for (var i = 0; i < 100; i++) {
    if (i % 2 !== 0) {
        totalCount += i
    }
}
console.log("totalCount:", totalCount)

// 算法优化
var totalCount = 0
for (var i = 1; i < 100; i+=2) {
    totalCount += i
```

```
}
console.log("totalCount:", totalCount)
```

5.4 循环的嵌套

> 循环中的循环体里面继续嵌套循环

```
//   案例一：在屏幕上显示包含很多❤的矩形

// 在屏幕上显示一个❤
// document.write("❤")

// 案例一:
for (var i = 0; i < 9; i++) {
    document.write("<div>")

    for (var m = 0; m < 10; m++) {
        document.write("❤ ")
    }

    document.write("</div>")
}
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤

//   案例二：在屏幕上显示一个三角的❤图像

for (var i = 0; i < 6; i++) {
    document.write("<div>")

    for (var m = 0; m < i+1; m++) {
        document.write("❤ ")
    }

    document.write("</div>")
}
// ❤
// ❤ ❤
// ❤ ❤ ❤
// ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
// ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤ ❤
```

```
//  案例三：在屏幕上显示一个九九乘法表

for (var i = 0; i < 9; i++) {
    document.write("<div>")

    for (var m = 0; m < i+1; m++) {
        var a = m + 1
        var b = i + 1
        var result = (m+1)*(i+1)
        // document.write(`${a}*${b}=${result} `)
        document.write(a + "*" + b + "=" + result + " ")
    }

    document.write("</div>")
}
```

5.5 循环控制

> 在执行循环体的过程中，我们可能需要根据不同的条件来控制循环体的终止（break）或是跳到下一次循环（continue）。

- **break:** 直接跳出循环, 循环结束
  - break 某一条件满足时，退出循环，不再执行后续重复的代码
- **continue:** 跳过本次循环次, 执行下一次循环体
  - continue 指令是 break 的“轻量版”。
  - continue 某一条件满足时，不执行后续重复的代码

```
var names = ["abc", "cba", "nba", "mba", "bba", "aaa", "bbb"]

// 循环遍历数组
// break关键字的使用
// 需求: 遇到nba时, 不再执行后续的迭代
// for (var i = 0; i < 4; i++) {
//   console.log(names[i])
//   if (names[i] === "nba") {
//     break
//   }
// }

// continue关键字的使用: 立刻结束本次循环, 执行下一次循环(step)
// 需求: 不打印nba和cba
for (var i = 0; i < 7; i++) {
    if (names[i] === "nba" || names[i] === "cba") {
        continue
    }
    console.log(names[i])
}
```

6. 函数

> 函数由称为**函数体**的**一系列语句组成。**
>
> 在 JavaScript 中，函数是**头等 (*first-class*)对象，因为它们可以像任何其他对象**一样具有属性和方法。它们与其他对象的区别在于函数可以被调用。简而言之，它们是`Function`对象。
>
> **函数**其实就是**某段代码的封装**，这段代码帮助我们完成某一个功能；
>
> 默认情况下JavaScript引擎或者浏览器会给我们**提供一些已经实现好的函数；**
>
> 我们也可以**编写属于自己的函数；**

6.1 函数的使用步骤

> 1. 定义函数（也称声明函数）
> 2. 使用函数

1. 定义函数

> 定义函数有多种方法：函数声明、函数表达式...
>
> 声明函数的过程是对某些功能的封装过程；

- **函数声明**

```
function name([param[, param[, ... param]]]) { statements }
```

`name`

函数名

> 函数名的命名规则和前面变量名的命名规则是相同的；
>
> 函数名尽量是见名知意
>
> 函数定义完成后，不会执行

`param`

传递给函数的参数的名称

> 这里定义的参数为函数的形参，是用来接受该函数调用时传过来的实参，形参在函数体内当做变量来使用。
>
> **形参（参数 parameter）**：**定义** 函数时，小括号中的参数，是用来接收参数用的，在函数内部 **作为变量使用**
>
> **实参（参数 argument）**：**调用** 函数时，小括号中的参数，是用来把数据传递到 **函数内部** 用的

`statements`

组成函数体的声明语句

> 当函数被调用时，函数体里面声明的语句会依次执行。

```
// 声明一个函数
// 制作好一个工具，但是这个工具默认情况下是没有被使用
function sayHello() {
    console.log("Hello!")
    console.log("how do you do!")
}

// 调用一个函数
sayHello()
// 函数可以在任何你想要使用的时候，进行调用
sayHello()

// 练习一：定义一个函数，打印自己的个人信息
function printInfo() {
    console.log("my name is why")
    console.log("age is 18")
    console.log("height is 1.88")
}

printInfo()
printInfo()

// 练习二：定义一个函数，在内部计算10和20的和
function sum() {
    var num1 = 10
    var num2 = 20
    var result = num1 + num2
    console.log("result:", result)
}
sum()

// name/age/height称之为函数的参数(形参，形式参数，parmaters)
function printInfo(name, age, height) {
    console.log(`my name is ${name}`)
    console.log(`age is ${age}`)
    console.log(`height is ${height}`)
}

// why/18/1.88称之为函数的参数(实参，实际参数，arguments)
printInfo("why", 18, 1.88)
printInfo("kobe", 30, 1.98)

// 另外一个案例也做一个重构
function sum(num1, num2) {
    var result = num1 + num2
    console.log("result:", result)
}

sum(20, 30)
sum(123, 321)
```

```
// 练习一: 和某人打招呼
function sayHello(name) {
    console.log(`Hello ${name}`)
}

sayHello("Kobe")
sayHello("James")
sayHello("Curry")

// 练习二: 给某人唱生日歌
function singBirthdaySong(name) {
    console.log("happy birthday to you")
    console.log("happy birthday to you")
    console.log(`happy birthday to ${name}`)
    console.log("happy birthday to you")
}

singBirthdaySong("Kobe")
singBirthdaySong("Why")
```

- **函数表达式**

```
let function_expression = function [name]([param1[, param2[, ..., paramN]]])
{
   statements
};
```

`name`

函数名称。可被省略，此种情况下的函数是匿名函数（*anonymous*）。**函数名称**只是**函数体中的一个本地变量。**

`paramN`

被传递给函数的一个参数名称。一个函数至多拥有 255 个参数。

`statements`

构成函数体的语句。

```
// 函数的声明(声明语句)
foo()
function foo() {
  console.log("foo函数被执行了~")
}

// 函数的表达式
// console.log(message) // undefined
// var message = "why"
```

```
// console.log(bar)
bar()
var bar = function() {
  console.log("bar函数被执行了~")
}
```

- 函数声明和函数表达式的**异同**
  - 相同点
    - 都是定义一个函数。
    - 都是一个对象，只不过这个对象可以调用。
  - 不同点
    - **语法不同**
      - 函数声明：在主代码流中声明为**单独的语句**的函数。
      - 函数表达式：在一个表达式中或另一个语法结构中创建的函数。
    - **创建时机不同**
      - 函数声明：在函数声明定义函数之前便可被调用。
        - 这是js引擎内部算法的原因
        - 当js引擎**准备**运行脚本时，首先会在脚本中**寻找全局函数声明，并创建这些函数**；
        - 在函数内部使用函数声明的函数，也可以在声明之前调用
      - 函数表达式：函数表达式是在代码**执行到达时**被创建，并且**仅从那一刻起可用**

2. 使用函数

> 通过**函数()**的形式来调用函数
>
> 函数调用时，可以传参，也可以不传；不传参数时，若该函数定义时，有形参定义，那么调用时，对应形参的值为**undefined**

3. 函数返回值

> 当函数执行时，可以通过**return**关键字来返回结果
>
> 程序在函数执行时，一旦遇到**return** 关键字会立即**终止当前函数**
>
> 如果**没有使用return**关键字，函数默认返回**undefined**
>
> 如果使用return关键字，但是后面没有任何值，也返回**undefined**

```
// var result = prompt("请输入一个数字:")
// 1.理解函数的返回值
// function sayHello(name) {
//   console.log(`Hi ${name}`)
```

```
// }

// var foo = sayHello("Kobe")
// console.log("foo:", foo)

// 2.返回值的注意事项
// 注意事项一: 所有的函数, 如果没有写返回值, 那么默认返回undefined
// function foo() {
//   console.log("foo函数被执行~")
// }

// var result = foo()
// console.log("foo的返回值:", result)

// 注意事项二: 我们也可以明确的写上return
// 写上return关键字, 但是后面什么内容都没有的时候, 也是返回undefined
// function bar() {
//   console.log("bar函数被执行~")
//   return
// }
// var result = bar()
// console.log("bar的返回值:", result)

// 注意事项三: 如果在函数执行到return关键字时, 函数会立即停止执行, 退出函数
// function baz() {
//   console.log("Hello Baz")
//   return
//   console.log("Hello World")
//   console.log("Hello Why")
// }

// baz()
```

4. arguments对象

> `arguments` 是一个对应于传递给函数的参数的类数组对象。（有length属性，但不是数组）
>
> `arguments`对象是所有（非箭头）函数中都可用的**局部变量**。你可以使用`arguments`对象在函数中引用函数的参数。此对象包含传递给函数的每个参数，第一个参数在索引 0 处。

```
// 1.arguments的认识
function foo(name, age) {
    console.log("传入的参数", name, age)

    // 在函数中都存在一个变量, 叫arguments
    console.log(arguments)
    // arguments是一个对象
    console.log(typeof arguments)
    // 对象内部包含了所有传入的参数
    // console.log(arguments[0])
    // console.log(arguments[1])
    // console.log(arguments[2])
    // console.log(arguments[3])
```

```
    // 对arguments来进行遍历
    for (var i = 0; i < arguments.length; i++) {
        console.log(arguments[i])
    }
}

foo("why", 18, 1.88, "广州市")


// 2.arguments的案例
function sum() {
    var total = 0
    for (var i = 0; i < arguments.length; i++) {
        var num = arguments[i]
        total += num
    }
    return total
}

console.log(sum(10, 20))
console.log(sum(10, 20, 30))
console.log(sum(10, 20, 30, 40))
```

5. 递归调用

> 一种函数调用自身的操作。递归被用于处理包含有更小的子问题的一类问题。一个递归函数可以接受两个输入参数：一个最终状态（终止递归）或一个递归状态（继续递归）。
>
> 必须要有终止条件，否则会报错。

```
// 下述代码定义了一个函数，其返回运行这段代码的 JavaScript 运行时的最大可用堆栈大小。
const getMaxCallStackSize = (i) => {
    try {
        return getMaxCallStackSize(++i);
    } catch {
        return i;
    }
};

console.log(getMaxCallStackSize(0)); // google: 10449  edge: 10454 firefox: 1513


// 什么是斐波那契数列
// 数列: 1 1 2 3 5 8 13 21 34 55  ... x
// 位置: 1 2 3 4 5 6 7  8  9  10  ... n

// 1.斐波那契的递归实现
function fibonacci(n) {
    if (n === 1 || n === 2) return 1
    return fibonacci(n-1) + fibonacci(n-2)
}
```

```
// 2.斐波那契的for循环实现
function fibonacci(n) {
    // 特殊的情况(前两个数字)
    if (n === 1 || n === 2) return 1

    // for循环的实现
    var n1 = 1
    var n2 = 1
    var result = 0
    for (var i = 3; i <= n; i++) {
        result = n1 + n2
        n1 = n2
        n2 = result
    }
    return result
}

console.log(fibonacci(5))
console.log(fibonacci(10))
console.log(fibonacci(20))
```

6.2 js中的函数是一等公民

> js中的函数是一等公民（头等函数）。
>
> 一种编程语言要称其函数为头等函数,通常需要满足以下几个条件:
>
> 1. 函数可以赋值给变量
> 2. 函数可以作为参数传递给其他函数
> 3. 函数可以作为其他函数的返回值
> 4. 函数可以存储在数据结构中
> 5. 函数可以在运行时创建
>
> JavaScript满足所有这些条件,因此其函数被称为头等函数。这种特性使得JavaScript在函数式编程范式中非常强大,能够实现高阶函数、闭包等高级概念,增加了语言的表达能力和灵活性。

```
// 1. 函数可以赋值给变量:
const greet = function(name) {
  console.log(`Hello, ${name}!`);
};

// 2. 函数可以作为参数传递给其他函数
function executeFunction(fn, param) {
  fn(param);
}

executeFunction(greet, "Alice");

// 3. 函数可以作为其他函数的返回值
function createMultiplier(factor) {
```

```
  return function(number) {
    return number * factor;
  };
}

const double = createMultiplier(2);
console.log(double(5)); // 输出: 10

// 4. 函数可以存储在数据结构中
const functionArray = [
  function(x) { return x * 2; },
  function(x) { return x + 3; }
];

// 5. 函数可以在运行时创建
const dynamicFunction = new Function("a", "b", "return a + b");
```

6.3 函数式编程

> ***使用函数来作为头等公民使用函数, 这种编程方式(范式)称为函数式编程.***
>
> 函数式编程是一种编程范式，它将计算过程视为数学函数的求值，并避免改变状态和可变数据。在JavaScript中，函数式编程的核心概念包括：
>
> 1. 纯函数：给定相同的输入，总是返回相同的输出，且没有副作用。
> 2. 不可变性：一旦创建，数据就不应被修改。
> 3. 函数组合：将多个简单函数组合成复杂函数。
> 4. 高阶函数：接受函数作为参数或返回函数的函数。
> 5. 声明式编程：描述要做什么，而不是如何做。

1. 纯函数

> 纯函数总是为相同的输入返回相同的输出,并且没有副作用。

```
// 纯函数
function add(a, b) {
  return a + b;
}

// 非纯函数 (有副作用)
let total = 0;
function addToTotal(value) {
  total += value;
  return total;
}
```

2. 不可变性

> 不可变性意味着创建后的数据不应被修改。在JavaScript中,我们可以使用const声明和扩展运算符来实现不可变性。

```
const originalArray = [1, 2, 3];

// 不改变原数组,而是创建新数组
const newArray = [...originalArray, 4];

console.log(originalArray); // [1, 2, 3]
console.log(newArray);      // [1, 2, 3, 4]
```

3. 函数组合

> 函数组合是将多个简单函数组合成一个更复杂的函数。

```
const compose = (f, g) => x => f(g(x));

const addOne = x => x + 1;
const double = x => x * 2;

const addOneThenDouble = compose(double, addOne);

console.log(addOneThenDouble(3)); // 8
```

4. 高阶函数

> 高阶函数是那些接受函数作为参数或返回函数的函数。

```
// 接受函数作为参数
function applyOperation(x, y, operation) {
  return operation(x, y);
}

const result = applyOperation(5, 3, (a, b) => a + b);
console.log(result); // 8

// 返回函数
function multiplier(factor) {
  return function(number) {
    return number * factor;
  }
}

const double = multiplier(2);
console.log(double(4)); // 8
```

5. 声明式编程

> 声明式编程关注的是描述我们想要的结果,而不是如何得到结果的具体步骤。

```
// 命令式 (how)
const numbers = [1, 2, 3, 4, 5];
let sum = 0;
for (let i = 0; i < numbers.length; i++) {
  sum += numbers[i];
}

// 声明式 (what)
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
```

6. 柯里化

> 柯里化是将一个接受多个参数的函数转换为一系列使用一个参数的函数。

```
// 普通函数
function add(x, y, z) {
  return x + y + z;
}

// 柯里化后的函数
const curriedAdd = x => y => z => x + y + z;

console.log(add(1, 2, 3));          // 6
console.log(curriedAdd(1)(2)(3));   // 6
```

应用场景：

> 1. 事件处理和参数预设
> 2. 表单验证
> 3. API请求封装
> 4. 样式组件的属性设置
> 5. 条件渲染优化

```
// 1. 柯里化可以帮助我们创建可重用的事件处理器，同时预设一些参数。
const handleClick = (message) => (event) => {
    console.log(message, event.target.id);
};

// 使用
document.getElementById('button1').addEventListener('click',
handleClick('Button 1 clicked:'));
document.getElementById('button2').addEventListener('click',
handleClick('Button 2 clicked:'));


// 2. 柯里化可以用来创建可复用的验证函数
const validate = (regex) => (message) => (value) => {
    if (!regex.test(value)) {
        return message;
    }
};
```

```
const validateEmail = validate(/^[^\s@]+@[^\s@]+\.[^\s@]+$/)('Invalid email address');
const validatePhone = validate(/^\d{10}$/)('Invalid phone number');

// 使用
function validateForm(email, phone) {
    const emailError = validateEmail(email);
    const phoneError = validatePhone(phone);

    if (emailError) console.log(emailError);
    if (phoneError) console.log(phoneError);
}

validateForm('test@example.com', '1234567890');


// 3. 柯里化可以用来创建更灵活的API请求函数
const apiRequest = (baseUrl) => (endpoint) => (method) => (data) => {
    return fetch(`${baseUrl}${endpoint}`, {
        method,
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(data),
    }).then(response => response.json());
};

const myApi = apiRequest('https://api.example.com');
const getUsersApi = myApi('/users');
const createUser = getUsersApi('POST');
const updateUser = getUsersApi('PUT');

// 使用
createUser({ name: 'John Doe', email: 'john@example.com' })
    .then(response => console.log(response));

updateUser({ id: 1, name: 'Jane Doe' })
    .then(response => console.log(response));


// 4. 在使用CSS-in-JS库（如styled-components）时，柯里化可以用来创建可复用的样式设置函数
import styled from 'styled-components';

const setFlex = (direction) => (justify) => (align) => `
  display: flex;
  flex-direction: ${direction};
  justify-content: ${justify};
  align-items: ${align};
`;

const Container = styled.div`
  ${setFlex('column')('center')('flex-start')}
  width: 100%;
  height: 100vh;
`;
```

```
const Row = styled.div`
  ${setFlex('row')('space-between')('center')}
  width: 100%;
  padding: 20px;
`;

// 使用
function App() {
    return (
        <Container>
        <Row>
        <p>Item 1</p>
        <p>Item 2</p>
        </Row>
        </Container>
    );
}


// 5. 柯里化可以用来创建更灵活的条件渲染函数
const when = (predicate) => (whenTrue) => (whenFalse) => (value) =>
predicate(value) ? whenTrue(value) : whenFalse(value);

const isNotEmpty = (x) => x !== '';
const isLongEnough = (x) => x.length >= 5;

const renderInput = when(isNotEmpty)(
    when(isLongEnough)
    ((x) => <p style={{color: 'green'}}>{x} is valid</p>)
((x) => <p style={{color: 'orange'}}>{x} is too short</p>)
)
(() => <p style={{color: 'red'}}>Please enter a value</p>);

// 使用
function Form() {
    const [value, setValue] = useState('');

    return (
        <div>
        <input value={value} onChange={(e) => setValue(e.target.value)} />
{renderInput(value)}
</div>
);
}
```

1. 好处

- 可预测性：纯函数和不可变数据使得代码行为更加可预测。
- 可测试性：纯函数易于单元测试。
- 并发：不可变数据和无副作用的函数使并发编程更安全。
- 模块化：通过函数组合，可以构建复杂的逻辑。
- 可读性：声明式代码通常更容易理解。
- 可维护性：函数式代码通常更容易维护和重构。

2. 如何学习

1. 理解基础概念

- 学习纯函数、不可变性、高阶函数等核心概念。
- 理解闭包、柯里化、组合等函数式技术。

2. 实践常用方法

- 熟练使用 `map`, `filter`, `reduce` 等数组方法。
- 学习 `Promise` 和异步编程的函数式方法。

3. 学习函数式库

- 尝试使用 Lodash、Ramda 等函数式工具库。
- 了解 RxJS 等响应式编程库。

4. 编码练习

- 重构现有代码，应用函数式原则
- 解决编程问题时尝试使用函数式方法

5. 深入学习资源

- 阅读《JavaScript函数式编程指南》等书籍
- 观看在线课程和视频教程
- 参与开源项目，学习实际应用

6. 渐进式采用

- 在小型项目中逐步引入函数式概念
- 逐步增加函数式编程在日常工作中的应用

> 学习函数式编程是一个gradual的过程。建议您从基础概念开始，逐步在实际项目中应用，并不断深化理解。随着实践的增加，您会发现函数式编程能够帮助您写出更加简洁、可维护的代码。

3. 应用场景（后台管理系统）

```
// 1. 数据处理和转换
// 后台管理系统经常需要处理和转换大量数据。使用函数式编程的方法可以使这些操作更加清晰和可组合。

// 假设我们有一个用户列表，需要进行筛选、转换和排序
const users = [
    { id: 1, name: 'Alice', age: 30, role: 'admin' },
    { id: 2, name: 'Bob', age: 25, role: 'user' },
    { id: 3, name: 'Charlie', age: 35, role: 'user' },
    // ...更多用户
];

// 使用函数式方法处理数据
const processUsers = (users) => {
    return users
        .filter(user => user.age >= 18)
        .map(user => ({
        ...user,
        displayName: `${user.name} (${user.role})`
    }))
        .sort((a, b) => a.age - b.age);
};

const processedUsers = processUsers(users);


// 2. 表单验证
// 表单验证是后台管理系统的常见需求。使用函数式编程可以创建可组合的验证函数。

// 创建可组合的验证函数
const required = field => value =>
value ? null : `${field} is required`;

const minLength = (field, min) => value =>
value && value.length >= min ? null : `${field} must be at least ${min}
characters`;

const isEmail = field => value =>
/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value) ? null : `${field} must be a valid
email`;

// 组合验证函数
const validateUser = (user) => {
```

```
    const nameValidation = [required('Name'), minLength('Name', 2)];
    const emailValidation = [required('Email'), isEmail('Email')];

    return {
        name: nameValidation.map(validate =>
validate(user.name)).filter(Boolean),
        email: emailValidation.map(validate =>
validate(user.email)).filter(Boolean)
    };
};

// 使用
const user = { name: 'A', email: 'invalid-email' };
const errors = validateUser(user);
console.log(errors);


// 3. 状态管理
// 使用函数式编程的原则可以简化状态管理，特别是在使用 Redux 这样的库时。

// Action creators
const setUsers = users => ({ type: 'SET_USERS', payload: users });
const addUser = user => ({ type: 'ADD_USER', payload: user });
const updateUser = user => ({ type: 'UPDATE_USER', payload: user });
const deleteUser = userId => ({ type: 'DELETE_USER', payload: userId });

// Reducer
const initialState = { users: [] };

const userReducer = (state = initialState, action) => {
    switch (action.type) {
        case 'SET_USERS':
            return { ...state, users: action.payload };
        case 'ADD_USER':
            return { ...state, users: [...state.users, action.payload] };
        case 'UPDATE_USER':
            return {
                ...state,
                users: state.users.map(user =>
                                    user.id === action.payload.id ?
action.payload : user
                                   )
            };
        case 'DELETE_USER':
            return {
                ...state,
                users: state.users.filter(user => user.id !== action.payload)
            };
        default:
            return state;
    }
};


// 4. API 请求封装
// 使用函数式编程可以创建更加灵活和可组合的 API 请求函数。
```

```
const apiRequest = method => endpoint => data =>
fetch(`https://api.example.com${endpoint}`, {
    method,
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(data)
}).then(response => response.json());

const get = apiRequest('GET');
const post = apiRequest('POST');
const put = apiRequest('PUT');
const del = apiRequest('DELETE');

// 使用
const getUsers = get('/users');
const createUser = post('/users');
const updateUser = put('/users');
const deleteUser = del('/users');

// 示例
getUsers().then(users => console.log(users));
createUser({ name: 'New User', email: 'new@example.com' })
    .then(newUser => console.log(newUser));


// 5. 权限控制
// 函数式编程可以用于创建灵活的权限控制系统。

const hasPermission = (requiredPermission) => (user) =>
user.permissions.includes(requiredPermission);

const withPermission = (permission) => (Component) => (props) => {
    if (hasPermission(permission)(props.user)) {
        return <Component {...props} />;
    } else {
        return <div>You don't have permission to view this content.</div>;
    }
};

// 使用
const AdminPanel = withPermission('ADMIN')(({ user }) => (
    <div>Welcome, Admin {user.name}!</div>
                                            ));

// 渲染
// const user = { name: 'Alice', permissions: ['USER', 'ADMIN'] };
// <AdminPanel user={user} />


// 6. 日志和错误处理
// 使用函数式编程可以创建可组合的日志和错误处理函数。

const logError = (error) => {
    console.error('An error occurred:', error);
    // 可以在这里添加更多的错误处理逻辑，比如发送到错误追踪服务
};
```

```
const withErrorHandling = (fn) => (...args) => {
    try {
        return fn(...args);
    } catch (error) {
        logError(error);
        throw error;
    }
};

// 使用
const riskyOperation = withErrorHandling(() => {
    // 一些可能抛出错误的操作
    throw new Error('Something went wrong');
});

try {
    riskyOperation();
} catch (error) {
    console.log('Error caught in the UI layer');
}
```

6.4 IIFE（立即调用函数表达式）

> 是一个在定义后立即被调用的JavaScript函数。

语法

```
(function() {
    // 函数体
})();

// 或者

(() => {
    // 函数体
})();
```

应用场景

```
// 1. 创建私有作用域
const counter = (function() {
    let count = 0;
    return {
        increment: function() {
            count++;
            return count;
```

```
        },
        decrement: function() {
            count--;
            return count;
        },
        getCount: function() {
            return count;
        }
    };
})();

console.log(counter.getCount()); // 0
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1


// 2. 避免全局命名空间污染
(function($) {
    // 这里的 $ 是 jQuery，不会与其他库冲突
    $(document).ready(function() {
        // ...
    });
})(jQuery);


// 3. 模块化代码
const myModule = (function() {
    const privateVariable = 'Hello World';
    function privateMethod() {
        console.log(privateVariable);
    }

    return {
        publicMethod: function() {
            privateMethod();
        }
    };
})();

myModule.publicMethod(); // 输出: Hello World


// 4. 业务逻辑应用
// var btnEls = document.querySelectorAll(".btn")
// for (var i = 0; i < btnEls.length; i++) {
//   var btn = btnEls[i];
//   btn.onclick = function() {
//     console.log(`按钮${i+1}发生了点击`)
//   }
// }

// 使用立即执行函数
var btnEls = document.querySelectorAll(".btn")
for (var i = 0; i < btnEls.length; i++) {
    var btn = btnEls[i];
```

```
    (function(m) {
        btn.onclick = function() {
            console.log(`按钮${m+1}发生了点击`)
        }
    })(i)
}

console.log(i)
```

6.5 回调函数

> 回调函数是作为参数传递给另一个函数，并在某个事件发生或某个任务完成后被调用的函数。

语法

```
function doSomething(callback) {
    // 做一些事情
    callback();
}

doSomething(function() {
    console.log('回调函数被调用');
});
```

应用场景

```
// 1. 异步操作
function fetchData(callback) {
    setTimeout(() => {
        const data = { id: 1, name: 'John' };
        callback(data);
    }, 1000);
}

fetchData((data) => {
    console.log('Data received:', data);
});


// 2. 事件处理
document.getElementById('myButton').addEventListener('click', function() {
    console.log('Button clicked!');
});


// 3. 高阶函数
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(function(num) {
```

```
    return num * 2;
});

console.log(doubledNumbers); // [2, 4, 6, 8, 10]


// 4. 错误处理
function divideNumbers(a, b, callback) {
    if (b === 0) {
        callback(new Error('Cannot divide by zero'));
        return;
    }
    callback(null, a / b);
}

divideNumbers(10, 2, (error, result) => {
    if (error) {
        console.error('Error:', error.message);
    } else {
        console.log('Result:', result);
    }
});


// 5. 迭代器和生成器
function* numberGenerator() {
    yield 1;
    yield 2;
    yield 3;
}

const gen = numberGenerator();

function handleNext(result) {
    if (!result.done) {
        console.log(result.value);
        gen.next().then(handleNext);
    }
}

gen.next().then(handleNext);
```

6.6 js中的参数传递

> 在JavaScript中，参数传递的方式取决于传递的数据类型。

1. 基本数据类型：值传递
    - 包括 number, string, boolean, null, undefined, Symbol
    - 函数接收的是值的拷贝
    - 在函数内部对参数的修改不会影响外部变量

2. 对象类型：共享传递
    - 包括普通对象、数组、函数等
    - 函数接收的是对象引用的拷贝
    - 可以通过引用修改对象的属性，这会影响到原始对象
    - 但是，重新赋值函数内部对应形参对象不会影响原始对象
3. 数组的参数传递（按共享传递的特例）：
    - 数组作为特殊的对象，遵循对象的传递规则。
    - 可以修改数组内容（如push、pop等），会影响原始数组。
    - 重新赋值整个数组不会影响原始数组。

```
// 1. 基本类型：值传递
function changeNumber(num) {
    num = 100;
    console.log("Inside function:", num);
}

let x = 10;
changeNumber(x);
console.log("Outside function:", x);
// 输出:
// Inside function: 100
// Outside function: 10

// 2. 对象：按共享传递
function changeObject(obj) {
    // 修改属性：会影响原对象
    obj.name = "Changed";
    console.log("Inside function (after modifying):", obj);

    // 重新赋值：不影响原对象
    obj = { name: "New Object" };
    console.log("Inside function (after reassigning):", obj);
}

let person = { name: "Original" };
changeObject(person);
console.log("Outside function:", person);
// 输出:
// Inside function (after modifying): { name: "Changed" }
// Inside function (after reassigning): { name: "New Object" }
// Outside function: { name: "Changed" }

// 3. 数组：按共享传递（特殊的对象）
function changeArray(arr) {
    // 修改数组：会影响原数组
    arr.push(4);
    console.log("Inside function (after push):", arr);

    // 重新赋值：不影响原数组
    arr = [5, 6, 7];
    console.log("Inside function (after reassigning):", arr);
}
```

```
let numbers = [1, 2, 3];
changeArray(numbers);
console.log("Outside function:", numbers);
// 输出:
// Inside function (after push): [1, 2, 3, 4]
// Inside function (after reassigning): [5, 6, 7]
// Outside function: [1, 2, 3, 4]
```

按共享传递

> 1. **对象的引用被传递**： 当你将一个对象传递给函数时，实际上传递的是该对象的引用（内存地址）。
> 2. **可以修改对象的内容**： 在函数内部，你可以通过这个引用修改对象的属性，这些修改会反映到原始对象上。
> 3. **不能改变引用本身**： 虽然你可以修改对象的内容，但是你不能改变引用本身指向的对象。如果你在函数内部给参数赋予一个新的对象，这只会改变局部变量，而不会影响原始引用。

```
// 示例1：修改对象属性
function modifyObject(obj) {
    obj.property = "modified";
}

let myObj = { property: "original" };
modifyObject(myObj);
console.log(myObj.property); // 输出: "modified"

// 示例2：重新赋值整个对象
function reassignObject(obj) {
    obj = { property: "new object" };
}

let anotherObj = { property: "original" };
reassignObject(anotherObj);
console.log(anotherObj.property); // 输出: "original"

// 示例3：数组操作
function modifyArray(arr) {
    arr.push(4);          // 修改原数组
    arr = [5, 6, 7];      // 创建新数组并赋值给局部变量arr
}

let myArray = [1, 2, 3];
modifyArray(myArray);
console.log(myArray); // 输出: [1, 2, 3, 4]
```

6.7 函数中的this指向

> 在 JavaScript 中，`this` 是一个特殊的关键字，它在函数内部使用，指向当前函数执行的上下文对象。`this` 是函数执行时自动创建的一个内部对象，它代表函数运行时的上下文。简单来说，`this` 指向的是"谁在调用当前函数"。

`this` 的值取决于函数的调用方式，主要有以下几种情况：

- 默认绑定

  > 在普通函数调用（函数直接调用）中，`this` 指向全局对象（非严格模式）或 `undefined`（严格模式）。

  ```
  function showThis() {
      console.log(this);
  }
  showThis(); // 在浏览器中输出 window 对象，或在严格模式下输出 undefined
  ```

- 隐式绑定

  > 当函数作为对象的方法调用时，`this` 指向调用该方法的对象。

  ```
  const obj = {
      name: 'MyObject',
      showThis: function() {
          console.log(this.name);
      }
  };
  obj.showThis(); // 输出 "MyObject"
  ```

- 显式绑定

  > 使用 `call()`、`apply()` 或 `bind()` 方法可以明确指定 `this` 的值

  ```
  function greet() {
      console.log(`Hello, ${this.name}`);
  }
  const user = { name: 'Alice' };
  greet.call(user); // 输出 "Hello, Alice"
  ```

- new 绑定

  > 当函数用作构造函数时，`this` 指向新创建的对象实例。

  ```
  function Person(name) {
      this.name = name;
  }
  const person = new Person('Bob');
  console.log(person.name); // 输出 "Bob"
  ```

- 箭头函数中的 `this`

> 箭头函数不创建自己的 `this` 上下文，而是继承外围作用域的 `this` 值

```
const obj = {
    name: 'MyObject',
    regularFunc: function() {
        setTimeout(function() {
            console.log(this.name); // undefined，因为 this 指向全局对象
        }, 100);
    },
    arrowFunc: function() {
        setTimeout(() => {
            console.log(this.name); // "MyObject"，因为箭头函数继承了外部的 this
        }, 100);
    }
};
```

7. 作用域

> 作用域表示定义了变量的可见范围。
>
> 作用域（Scope）：定义了变量的可见性和可访问性的代码区域。
>
> 全局作用域（Global Scope）：整个程序中都可以访问的最外层作用域。
>
> 函数作用域（Function Scope）：在函数内部定义的作用域，只在该函数内部可见。
>
> 块级作用域（Block Scope）：在代码块（如if语句、for循环）内部定义的作用域。
>
> 词法作用域（Lexical Scope）：也称为静态作用域，指的是作用域是由代码中函数声明的位置来决定的，而不是由函数调用的位置决定。

```
// 1.作用域的理解:message在哪一个范围内可以被使用, 称之为message的作用域(scope)
// 全局变量: 全局作用域
var message = "Hello World"
if (true) {
    console.log(message)
}
function foo() {
    console.log("在foo中访问", message)
}
foo()

// 2.ES5之前是没有块级作用域(var定义的变量是没有块级作用域)
{
    var count = 100
    console.log("在代码块中访问count:", count)
}
console.log("在代码块外面访问count:", count)
// for循环的代码块也是没有自己的作用域
```

```
for (var i = 0; i < 3; i++) {
    var foo = "foo"
    }
console.log("for循环外面访问foo:", foo)
console.log("for循环外面访问i:", i) // 3

// 3.ES5之前函数代码块是会形成自己的作用域
// 意味着在函数内部定义的变量外面是访问不到的
function test() {
    var bar = "bar"
    }

test()
// console.log("test函数外面访问bar:", bar)

// 函数有自己的作用域: 函数内部定义的变量只有函数内部能访问到
function sayHello() {
    var nickname = "kobe"
    console.log("sayHello函数的内部:", nickname)

    function hi() {
        console.log("hi function~")
        console.log("在hi函数中访问nickname:", nickname)
    }
    hi()
}
sayHello()
// console.log("sayHello外面访问nickname:", nickname)
```

```
// 1.全局变量(global variable): 在全局(script元素中)定义一个变量, 那么这个变量是可以在定义之后的任何范围内被访问到的, 那么这个变量就称之为是一个全局变量.
var message = "Hello World"

// 在函数中访问message
function sayHello() {
    // 外部变量(outer variable): 在函数内部去访问函数之外的变量, 访问的变量称之为外部变量
    console.log("sayHello中访问message:", message)

    // 2.局部变量(local variable): 在函数内部定义的变量, 只有在函数内部才能进行访问, 称之为局部变量
    var nickname = "coderwhy"

    function hi() {
        console.log("hi function~")
        // message也是一个外部变量
        console.log("hi中访问message:", message)
        // nickname也是一个外部变量
        console.log("hi中访问nickname:", nickname)
    }
    hi()
}
```

```
sayHello()
```

```
// 变量的访问顺序

// var message = "Hello World"

function sayHello() {
    // var message = "Hello Coderwhy"

    function hi() {
        // var message = "Hi Kobe"
        console.log(message)
    }
    hi()
}

sayHello()
```

外部变量和局部变量的概念：

- 定义在函数内部的变量，被称之为**局部变量**（Local Variables）。
- 定义在函数外部的变量，被称之为**外部变量**（Outer Variables）。

全局变量：

- 在函数之外声明的变量（在script中声明的），称之为全局变量。
- 全局变量**在任何函数中都是可见的。**
- 通过**var声明的全局变量会在window对象**上添加一个属性（了解）；

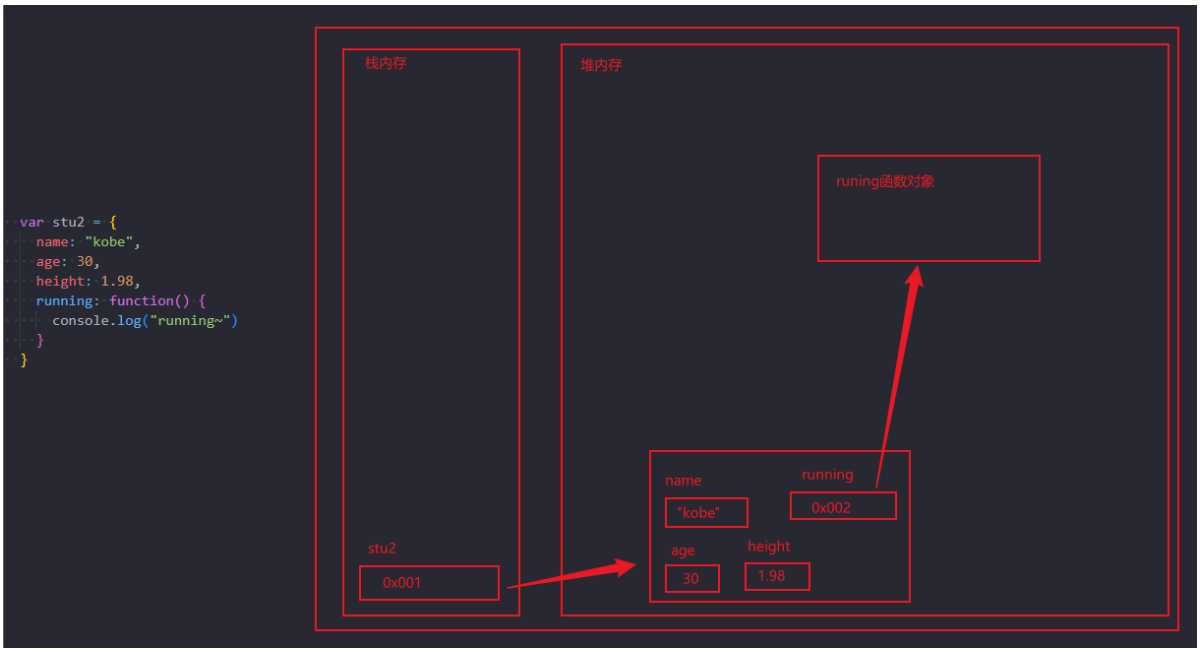
ps：

- 查找变量
  - 通过作用域链进行查找；若找不到，js引擎报错。
- 查找对象的属性
  - 通过对象的原型链进行查找；若找不到，返回**undefined**
- 上述规则都是**就近原则**

8. 对象

> 对象是包含相关数据和/或功能的集合。它们通常由多个变量和函数组成，这些变量和函数称为对象的属性和方法。

> 对象是存在堆内存中的，对象中的非引用属性，存在该对象的内存空间中，如果是引用类型的属性，其值是存在堆内存中的另外一块单独的内存中。依次类推



1. 创建对象

- 对象字面量

> 这是最简单和最常用的方法
>
> 优点：简洁，适合创建单个对象。

```
//  两个术语: 函数/方法
//      函数(function): 如果在JavaScript代码中通过function默认定义一个结构, 称之为是函数.
//      方法(method): 如果将一个函数放到对象中, 作为对象的一个属性, 那么将这个函数称之为方法.

// key: 字符串类型也可以是symbol类型, 但是在定义对象的属性名时, 大部分情况下引号都是可以省略的
var person = {
    // key: value
    name: "why",
    age: 18,
    height: 1.88,
    "my friend": {
        name: "kobe",
        age: 30
    },
    run: function() {
        console.log("running")
    },
    eat: function() {
        console.log("eat foods")
    },
    study: function() {
```

```
        console.log("studying")
    }
}
```

- 构造函数

> 使用构造函数可以创建多个具有相同结构的对象
>
> 优点：可以创建多个类似的对象，支持初始化。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        console.log(`Hello, I'm ${this.name}`);
    };
}

const person1 = new Person("Bob", 25);
const person2 = new Person("Charlie", 35);
```

- Object.create() 方法

> 基于现有对象创建新对象
>
> 优点：可以直接指定对象的原型。

```
const personProto = {
    greet: function() {
        console.log("Hello!");
    }
};

const person = Object.create(personProto);
person.name = "John";
person.age = 30;
```

- 类（ES6+）

> ES6引入了类语法，使对象创建更接近传统的面向对象语言
>
> 优点：语法清晰，支持继承，方法在原型上共享。

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log(`Hello, I'm ${this.name}`);
    }
}

const person = new Person("Eve", 28);
```

- 工厂函数

> 工厂函数是返回对象的函数
>
> 优点：可以封装复杂的创建逻辑，不使用 `new` 关键字。

```
function createPerson(name, age) {
    return {
        name,
        age,
        greet() {
            console.log(`Hello, I'm ${this.name}`);
        }
    };
}

const person = createPerson("Frank", 33);
```

- 单例模式

> 只需要一个对象实例时
>
> 优点：确保只有一个实例，全局访问点。

```
const Singleton = (function() {
    let instance;

    function createInstance() {
        return {
            name: "Singleton",
            method() {
                console.log("Method called");
            }
        };
    }

    return {
        getInstance: function() {
            if (!instance) {
                instance = createInstance();
            }
            return instance;
        }
    };
})();

const object1 = Singleton.getInstance();
const object2 = Singleton.getInstance();
console.log(object1 === object2); // true
```

- Object.assign()

> 用于合并多个对象的属性
>
> 优点：可以快速组合多个对象的属性。

```
const baseObject = { a: 1, b: 2 };
const newObject = Object.assign({}, baseObject, { c: 3 });
console.log(newObject); // { a: 1, b: 2, c: 3 }
```

- 展开运算符（ES6+）

> 类似于 `Object.assign()`，但语法更简洁
>
> 语法简洁，易读

```
const baseObject = { a: 1, b: 2 };
const newObject = { ...baseObject, c: 3 };
console.log(newObject); // { a: 1, b: 2, c: 3 }
```

每种方法都有其适用场景

- 对象字面量适合简单对象。
- 构造函数和类适合创建多个类似对象。
- `Object.create()` 适合需要特定原型的对象。
- 工厂函数适合需要封装创建逻辑的场景。
- 单例模式适合整个应用只需一个实例的对象。
- `Object.assign()` 和展开运算符适合对象合并和浅拷贝。

2. 操作对象

1. 访问对象属性

- 点符号

```
console.log(person.name); // "John"
```

- 括号符号

```
console.log(person["age"]); // 30

// 方括号 + 变量方式访问
const myAge = "age";
console.log(person[myAge]); // 30
```

2. 修改对象属性

```
person.age = 31;
person["name"] = "John Doe";
```

3. 添加新属性

```
person.job = "Developer";
```

4. 删除属性

```
delete person.job;
```

5. 方法

> 当函数作为对象的属性值时，就说这个函数是这个对象的方法。
>
> **方法是作为对象属性的函数。**

```
const person = {
    name: "John",
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};

person.greet(); // 输出: "Hello, my name is John"
```

6. getter和setter

```
const person = {
    firstName: "John",
    lastName: "Doe",
    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    },
    set fullName(name) {
        const parts = name.split(' ');
        this.firstName = parts[0];
        this.lastName = parts[1];
    }
};

console.log(person.fullName); // "John Doe"
person.fullName = "Jane Smith";
console.log(person.firstName); // "Jane"
```

7. Object.keys(), Object.values(), Object.entries()

```
const person = { name: "John", age: 30 };

console.log(Object.keys(person)); // ["name", "age"]
console.log(Object.values(person)); // ["John", 30]
console.log(Object.entries(person)); // [["name", "John"], ["age", 30]]
```

8. 对象解构

```
const { name, age } = person;
console.log(name); // "John"
console.log(age);  // 30
```

9. 展开运算符

```
const person1 = { name: "John", age: 30 };
const person2 = { ...person1, job: "Developer" };
console.log(person2); // { name: "John", age: 30, job: "Developer" }
```

3. 遍历对象

```
// 下述案例所要用到的对象
const person = {
    name: "John",
    age: 30,
    job: "Developer",
    city: "New York"
};
```

1. for...in循环

> `for...in` 循环遍历对象的所有可枚举属性，包括原型链上的属性。

```
for (let key in person) {
  if (person.hasOwnProperty(key)) {
    console.log(key + ": " + person[key]);
  }
}

// 注意：使用 hasOwnProperty() 方法可以确保只遍历对象自身的属性，而不是原型链上的属性。
```

2. Object.keys()

> `Object.keys()` 返回一个由对象的自身可枚举属性组成的数组。

```
Object.keys(person).forEach(key => {
  console.log(key + ": " + person[key]);
});
```

3. Object.values()

> `Object.values()` 返回一个由对象的自身可枚举属性值组成的数组。

```
Object.values(person).forEach(value => {
  console.log(value);
});
```

4. Object.entries()

> `Object.entries()` 返回一个由对象的自身可枚举属性的键值对数组组成的数组。

```
Object.entries(person).forEach(([key, value]) => {
  console.log(key + ": " + value);
});
```

5. Object.getOwnPropertyNames()

> `Object.getOwnPropertyNames()` 返回一个由对象的所有自身属性的名称组成的数组，包括不可枚举属性。

```
Object.getOwnPropertyNames(person).forEach(key => {
  console.log(key + ": " + person[key]);
});
```

6. Reflect.ownKeys()

> `Reflect.ownKeys()` 返回一个由对象的所有自身属性的键组成的数组，包括符号属性和不可枚举属性。

```
Reflect.ownKeys(person).forEach(key => {
  console.log(key + ": " + person[key]);
});
```

7. Object.getOwnPropertyDescriptors()

> 这个方法返回对象的所有自身属性描述符。虽然它不是直接用于遍历，但可以用来获取所有属性，包括 getter 和 setter。

```
const descriptors = Object.getOwnPropertyDescriptors(person);
for (let key in descriptors) {
  console.log(key + ": " + person[key]);
}
```

8. 比较和建议

for...in: 最古老的方法，但需要注意处理原型链属性。

Object.keys(): 常用且简洁，只遍历自身可枚举属性。

Object.values(): 当只需要值而不需要键时很有用。

Object.entries(): 同时需要键和值时很方便，特别是在需要对对象进行转换时。

Object.getOwnPropertyNames(): 需要包括不可枚举属性时使用。

Reflect.ownKeys(): 最全面，包括符号属性，但使用较少。

Object.getOwnPropertyDescriptors(): 需要属性的完整描述符时使用。

性能考虑

> 在大多数情况下，这些方法的性能差异不大。但如果处理大型对象或需要频繁遍历，可以考虑使用 `for...in` 或 `Object.keys()` 来获得更好的性能。

实际应用示例

- 创建对象的浅拷贝

```
function shallowCopy(obj) {
  const copy = {};
  Object.keys(obj).forEach(key => {
    copy[key] = obj[key];
  });
  return copy;
}

const personCopy = shallowCopy(person);
console.log(personCopy);
```

- 过滤对象属性

```
function filterObject(obj, predicate) {
  return Object.entries(obj).reduce((acc, [key, value]) => {
    if (predicate(value, key)) {
      acc[key] = value;
    }
    return acc;
  }, {});
}

const adult = filterObject(person, (value, key) => key === 'age' ? value >= 18 : true);
console.log(adult);
```

9. 可枚举属性

> 对象自身的可枚举属性是指（同时满足以下两个条件）：
>
> 1. 直接定义在对象上的属性（不是从原型链继承的）
> 2. 其内部 `enumerable` 标志设置为 `true` 的属性
>
> 这些属性在使用某些方法（如 `for...in` 循环或 `Object.keys()`）时会被包括在内。

属性特性

每个对象属性都有一些特性（也称为属性描述符），包括：

- `value`：属性的值
- `writable`：是否可以修改属性的值
- `enumerable`：是否可枚举
- `configurable`：是否可以删除该属性或修改其特性

可枚举 vs 不可枚举

1. **可枚举属性**：
   - 在 `for...in` 循环中会被遍历
   - 会被 `Object.keys()` 方法返回
   - 会被 `JSON.stringify()` 序列化
2. **不可枚举属性**：
   - 不会在 `for...in` 循环中被遍历
   - 不会被 `Object.keys()` 方法返回
   - 不会被 `JSON.stringify()` 序列化（除非显式指定）

示例

```
const person = {
    name: "John",
    age: 30
};

// 添加一个不可枚举属性
Object.defineProperty(person, 'ssn', {
    value: '123-45-6789',
    enumerable: false
});

console.log(Object.keys(person)); // ['name', 'age']
console.log(person.ssn); // '123-45-6789'

for (let key in person) {
    console.log(key); // 只输出 'name' 和 'age'
}

console.log(JSON.stringify(person)); // {"name":"John","age":30}

// 在这个例子中，name 和 age 是可枚举属性，而 ssn 是不可枚举属性。
```

检查属性是否可枚举

> 可以使用 `propertyIsEnumerable()` 方法来检查一个属性是否可枚举

```
console.log(person.propertyIsEnumerable('name')); // true
console.log(person.propertyIsEnumerable('ssn'));  // false
```

获取所有属性（包括不可枚举属性）

> 获取对象的所有属性，包括不可枚举的，可以使用 `Object.getOwnPropertyNames()`

```
console.log(Object.getOwnPropertyNames(person)); // ['name', 'age', 'ssn']
```

实际应用

1. **隐藏内部实现细节**： 不可枚举属性通常用于存储不希望被外部轻易访问或修改的内部数据。
2. **避免意外遍历**： 在某些情况下，你可能不希望某些属性在遍历对象时被包括进来，使用不可枚举属性可以避免这种情况。
3. **与 API 设计相关**： 在设计库或框架时，可以使用不可枚举属性来存储内部状态或元数据，而不影响公共接口。

创建不可枚举属性

```
// 创建属性时设置其为不可枚举
const obj = {};
Object.defineProperty(obj, 'hiddenProp', {
  value: 'I am hidden',
  enumerable: false
});

// 修改现有属性的可枚举性
Object.defineProperty(person, 'age', {
  value: person.age,
  enumerable: false
});
```

4. 全局对象

> 浏览器中存在一个全局对象object -> window
>
> 1. 查找变量时, 最终会找到window头上
> 2. 将一些浏览器全局提供给我们的变量/函数/对象, 放在window对象上面
> 3. 使用var定义的变量会被默认添加到window上面，当做window对象的属性，通过函数声明的方式创建的函数会被当做window对象的方法。

```
console.log(window)

// 使用var定义变量
var message = "Hello World"

function foo() {
    // 自己的作用域
    // abc()
    // alert("Hello World")
    console.log(window.console === console)

    // 创建一个对象
    // var obj = new Object()
    console.log(window.Object === Object)

    // DOM
    console.log(document)

    // window.message
    console.log(window.message)
}
foo()
```

9. 包装类

> JavaScript中的基本类型（也称为原始类型）包括：字符串（String）、数字（Number）、布尔值（Boolean）、Undefined、Null、Symbol（ES6引入）和BigInt（ES2020引入）。其中，**String、Number、Boolean、Symbol和BigInt**都有对应的包装类。

工作原理

> 当你在基本类型值上调用方法时，JavaScript会临时创建一个对应的包装对象，让你能够访问这些方法。操作完成后，这个临时对象会被丢弃。

```
let str = "Hello";
console.log(str.toUpperCase()); // "HELLO"

// 内部大致相当于：
// let temp = new String(str);
// console.log(temp.toUpperCase());
// temp = null;
```

1. string包装类

> String对象是对字符串的包装。

常用方法和属性

- 访问和检索
  - `charAt(index)`：返回指定位置的字符
  - `charCodeAt(index)`：返回指定位置字符的Unicode编码
  - `codePointAt(pos)`：返回使用UTF-16编码的给定位置的值
  - `indexOf(searchValue[, fromIndex])`：返回指定值首次出现的位置
  - `lastIndexOf(searchValue[, fromIndex])`：返回指定值最后出现的位置

```
let str = "Hello, World!";

// charAt(index) - 返回指定位置的字符
console.log(str.charAt(7)); // 'W'

// charCodeAt(index) - 返回指定位置字符的Unicode值
console.log(str.charCodeAt(0)); // 72 ('H'的Unicode值)

// codePointAt(pos) - 返回使用UTF-16编码的给定位置的值
console.log(str.codePointAt(0)); // 72 (对于基本拉丁字符，与charCodeAt相同)

// indexOf(searchValue[, fromIndex]) - 返回指定值首次出现的位置
console.log(str.indexOf('o')); // 4
console.log(str.indexOf('o', 5)); // 7 (从索引5开始搜索)

// lastIndexOf(searchValue[, fromIndex]) - 返回指定值最后出现的位置
console.log(str.lastIndexOf('o')); // 7
```

```
console.log(str.lastIndexOf('o', 5)); // 4 (从索引5开始向后搜索)
```

- 查找和匹配
  - `includes(searchString[, position])`：判断是否包含指定字符串
  - `startsWith(searchString[, position])`：判断是否以指定字符串开头
  - `endsWith(searchString[, length])`：判断是否以指定字符串结尾
  - `match(regexp)`：与正则表达式进行匹配
  - `matchAll(regexp)`：返回所有匹配的迭代器
  - `search(regexp)`：搜索匹配正则表达式的子串

```
let str = "Hello, World!";

// includes(searchString[, position]) - 判断是否包含指定字符串
console.log(str.includes('World')); // true
console.log(str.includes('World', 8)); // false (从索引8开始搜索)

// startsWith(searchString[, position]) - 判断是否以指定字符串开头
console.log(str.startsWith('Hello')); // true
console.log(str.startsWith('World', 7)); // true (从索引7开始考虑'World'作为开头)

// endsWith(searchString[, length]) - 判断是否以指定字符串结尾
console.log(str.endsWith('World!')); // true
console.log(str.endsWith('Hello', 5)); // true (只考虑前5个字符)

// match(regexp) - 与正则表达式进行匹配
console.log(str.match(/o/g)); // ['o', 'o']

// matchAll(regexp) - 返回所有匹配的迭代器
let matches = [...str.matchAll(/o/g)];
console.log(matches.map(m => m.index)); // [4, 7]

// search(regexp) - 搜索匹配正则表达式的子串
console.log(str.search(/World/)); // 7
```

- 提取和分割
  - `slice(beginIndex[, endIndex])`：提取字符串的一部分
  - `substring(indexStart[, indexEnd])`：返回指定两个下标之间的字符
  - `substr(start[, length])`：从指定位置开始提取指定长度的字符串（已废弃）
  - `split([separator[, limit]])`：将字符串分割为数组

```
let str = "Hello, World!";

// slice(beginIndex[, endIndex]) - 提取字符串的一部分
console.log(str.slice(7)); // "World!"
console.log(str.slice(0, 5)); // "Hello"

// substring(indexStart[, indexEnd]) - 返回指定两个下标之间的字符
console.log(str.substring(7)); // "World!"
console.log(str.substring(0, 5)); // "Hello"

// split([separator[, limit]]) - 将字符串分割为数组
console.log(str.split(', ')); // ["Hello", "World!"]
console.log(str.split('', 5)); // ["H", "e", "l", "l", "o"]
```

- 修改和替换
  - `replace(searchFor, replaceWith)`: 替换匹配的子串
  - `replaceAll(searchFor, replaceWith)`: 替换所有匹配的子串
  - `toLowerCase()`: 转换为小写
  - `toUpperCase()`: 转换为大写
  - `toLocaleLowerCase([locale, ...locales])`: 根据区域设置转换为小写
  - `toLocaleUpperCase([locale, ...locales])`: 根据区域设置转换为大写
  - `trim()`: 删除两端空白字符
  - `trimStart()` / `trimLeft()`: 删除开头空白字符
  - `trimEnd()` / `trimRight()`: 删除结尾空白字符
  - `padStart(targetLength [, padString])`: 在开头填充字符串
  - `padEnd(targetLength [, padString])`: 在结尾填充字符串
  - `repeat(count)`: 重复字符串指定次数

```
let str = "Hello, World!";

// replace(searchFor, replaceWith) - 替换匹配的子串
console.log(str.replace('World', 'JavaScript')); // "Hello, JavaScript!"

// replaceAll(searchFor, replaceWith) - 替换所有匹配的子串
console.log("Hello World, World!".replaceAll('World', 'JavaScript')); //
"Hello JavaScript, JavaScript!"

// toLowerCase() - 转换为小写
console.log(str.toLowerCase()); // "hello, world!"

// toUpperCase() - 转换为大写
console.log(str.toUpperCase()); // "HELLO, WORLD!"

// toLocaleLowerCase([locale, ...locales]) - 根据区域设置转换为小写
console.log(str.toLocaleLowerCase('tr')); // "hello, world!" (土耳其语区
域)
```

```
// toLocaleUpperCase([locale, ...locales]) - 根据区域设置转换为大写
console.log(str.toLocaleUpperCase('tr')); // "HELLO, WORLD!" (土耳其语区域)

// trim() - 删除两端空白字符
console.log("  Hello, World!  ".trim()); // "Hello, World!"

// trimStart() / trimLeft() - 删除开头空白字符
console.log("  Hello, World!  ".trimStart()); // "Hello, World!  "

// trimEnd() / trimRight() - 删除结尾空白字符
console.log("  Hello, World!  ".trimEnd()); // "  Hello, World!"

// padStart(targetLength [, padString]) - 在开头填充字符串
console.log('5'.padStart(3, '0')); // "005"

// padEnd(targetLength [, padString]) - 在结尾填充字符串
console.log('5'.padEnd(3, '0')); // "500"

// repeat(count) - 重复字符串指定次数
console.log('Ha'.repeat(3)); // "HaHaHa"
```

- 连接和合并
  - `concat(str1, str2, ...)`: 连接两个或多个字符串

```
// concat(str1, str2, ...) - 连接两个或多个字符串
let str1 = "Hello";
let str2 = "World";
console.log(str1.concat(", ", str2, "!")); // "Hello, World!"
```

- 比较和排序
  - `localeCompare(compareString[, locales[, options]])`: 比较字符串

```
// localeCompare(compareString[, locales[, options]]) - 比较字符串
console.log('a'.localeCompare('b')); // -1
console.log('b'.localeCompare('a')); // 1
console.log('a'.localeCompare('a')); // 0
```

- 转换和规范化
  - `toString()`: 返回字符串对象的字符串形式
  - `valueOf()`: 返回字符串对象的原始值
  - `normalize([form])`: 将字符串正规化为指定的Unicode形式

```
// toString() - 返回字符串对象的字符串形式
let strObj = new String("Hello");
console.log(strObj.toString()); // "Hello"

// valueOf() - 返回字符串对象的原始值
console.log(strObj.valueOf()); // "Hello"

// normalize([form]) - 将字符串正规化为指定的Unicode形式
let str = '\u0041\u006d\u00e9\u006c\u0069\u0065';
console.log(str.normalize('NFC')); // "Amélie"
```

- 静态方法（String构造函数的方法）
  - `String.fromCharCode(num1[, ...[, numN]])`：从UTF-16代码单元创建字符串
  - `String.fromCodePoint(num1[, ...[, numN]])`：从代码点创建字符串
  - `String.raw(callSite, ...substitutions)`：获取模板字符串的原始字符串

```
// String.fromCharCode(num1[, ...[, numN]]) - 从UTF-16代码单元创建字符串
console.log(String.fromCharCode(72, 101, 108, 108, 111)); // "Hello"

// String.fromCodePoint(num1[, ...[, numN]]) - 从代码点创建字符串
console.log(String.fromCodePoint(72, 101, 108, 108, 111)); // "Hello"

// String.raw(callSite, ...substitutions) - 获取模板字符串的原始字符串
console.log(String.raw`Hello\nWorld`); // "Hello\nWorld" (反斜杠不被解释)
```

- 属性
  - `length`：返回字符串的长度

```
// length - 返回字符串的长度
let str = "Hello, World!";
console.log(str.length); // 13
```

2. Number 包装类

> Number对象是对数字的包装。

常用方法和属性

- `toFixed(digits)`：格式化为指定小数位数的字符串
- `toPrecision(precision)`：格式化为指定精度的字符串
- `toString(radix)`：转换为字符串，可指定基数
- `parseInt(string, radix)`：将字符串解析为整数（静态方法）
- `parseFloat(string)`：将字符串解析为浮点数（静态方法）
- `isNaN(value)`：检查是否为NaN（静态方法）

- `isFinite(value)`：检查是否为有限数（静态方法）

```
// Number 类型的方法和属性

// 1. 数值转换方法
let num = 123.456;

// toString() - 将数字转换为字符串
console.log(num.toString()); // "123.456"
console.log(num.toString(2)); // "1111011.011101001011110001101010l" (2进制)

// toFixed() - 格式化小数位数
console.log(num.toFixed(2)); // "123.46"

// toPrecision() - 格式化数字的精度
console.log(num.toPrecision(4)); // "123.5"

// toExponential() - 转换为指数表示法
console.log(num.toExponential(2)); // "1.23e+2"

// 2. 数值检查方法
// isFinite() - 检查是否为有限数
console.log(Number.isFinite(num)); // true
console.log(Number.isFinite(Infinity)); // false

// isInteger() - 检查是否为整数
console.log(Number.isInteger(123)); // true
console.log(Number.isInteger(123.45)); // false

// isNaN() - 检查是否为NaN
console.log(Number.isNaN(NaN)); // true
console.log(Number.isNaN(123)); // false

// 3. 数学常量
console.log(Number.EPSILON); // 最小精度
console.log(Number.MAX_SAFE_INTEGER); // JavaScript 中最大的安全整数
console.log(Number.MIN_SAFE_INTEGER); // JavaScript 中最小的安全整数
console.log(Number.MAX_VALUE); // 可表示的最大正数
console.log(Number.MIN_VALUE); // 可表示的最小正数
console.log(Number.POSITIVE_INFINITY); // 正无穷大
console.log(Number.NEGATIVE_INFINITY); // 负无穷大

// 4. 解析方法
// parseInt() - 将字符串解析为整数
console.log(Number.parseInt("123")); // 123
console.log(Number.parseInt("123.45")); // 123
console.log(Number.parseInt("123", 16)); // 291 (16进制)

// parseFloat() - 将字符串解析为浮点数
console.log(Number.parseFloat("123.45")); // 123.45
console.log(Number.parseFloat("123.45.67")); // 123.45

// 5. 符号相关方法
// Math.sign() - 返回数字的符号
console.log(Math.sign(5)); // 1
```

```
console.log(Math.sign(-5)); // -1
console.log(Math.sign(0)); // 0

// 6. 比较方法
let num1 = 123, num2 = 456;

// Math.max() - 返回最大值
console.log(Math.max(num1, num2)); // 456

// Math.min() - 返回最小值
console.log(Math.min(num1, num2)); // 123
```

3. Boolean 包装类

> Boolean对象是对布尔原始类型的包装。

常用方法

- `toString()`: 转换为字符串 "true" 或 "false"
- `valueOf()`: 返回原始布尔值

```
let bool = new Boolean(true);
console.log(bool.toString()); // "true"
console.log(bool.valueOf()); // true
```

4. Symbol（ES6+）

> Symbol是ES6引入的新的原始数据类型，用于创建唯一的标识符。

常用方法和属性

- `Symbol.for(key)`: 搜索现有的symbol，如果找到则返回它，否则创建一个新的symbol
- `Symbol.keyFor(sym)`: 获取全局symbol的key
- `description`: 返回symbol的可选描述

```
let sym1 = Symbol("my symbol");
let sym2 = Symbol.for("shared symbol");
console.log(sym1.description); // "my symbol"
console.log(Symbol.keyFor(sym2)); // "shared symbol"
```

5. BigInt（ES11+）

> BigInt是用于表示任意精度整数的数据类型。

常用方法

- `toString(radix)`：转换为字符串，可指定基数
- `valueOf()`：返回BigInt的原始值

```
let bigInt = 1234567890123456789012345678901234567890n;
console.log(bigInt.toString()); // "1234567890123456789012345678901234567890"
console.log(bigInt.toString(16)); // "3b9ac9ff8e7ce5a45c9c7c25c21a3d5fn"
```

注意事项

1. 虽然可以使用 `new String()`，`new Number()`，或 `new Boolean()` 创建包装对象，但通常不推荐这样做，因为它可能导致意外的行为。
2. `null` 和 `undefined` 没有对应的包装类。
3. 使用包装类的方法不会改变原始值，而是返回一个新值。
4. ES6+引入的 `Symbol` 和 `BigInt` 类型有一些特殊性，它们的行为可能与其他原始类型略有不同。

10. 内置类

JavaScript内置类

JavaScript中的主要内置类包括:

1. Object
2. Array
3. String
4. Number
5. Boolean
6. Function
7. Date
8. RegExp
9. Error
10. Math

下面详细列出每个类的主要属性和方法:

1. Object

属性:

- `constructor`: 返回创建此对象的构造函数

方法:

- `hasOwnProperty(prop)`: 检查对象是否具有指定的属性
- `isPrototypeOf(object)`: 检查当前对象是否在另一个对象的原型链中
- `propertyIsEnumerable(prop)`: 检查指定属性是否可枚举
- `toString()`: 返回对象的字符串表示
- `valueOf()`: 返回对象的原始值

```
// Object 类的方法和属性

// 创建一个示例对象
let exampleObj = {
    name: "测试对象",
    value: 42
};

// 1. Object 构造函数方法

// Object.assign() - 将一个或多个源对象的可枚举属性复制到目标对象
let targetObj = { a: 1 };
let sourceObj = { b: 2 };
console.log(Object.assign(targetObj, sourceObj)); // { a: 1, b: 2 }

// Object.create() - 使用指定的原型对象和属性创建一个新对象
let prototypeObj = { greet() { console.log("你好!"); } };
let newObj = Object.create(prototypeObj);
newObj.greet(); // 输出: 你好!

// Object.defineProperty() - 在对象上定义新属性或修改现有属性
Object.defineProperty(exampleObj, 'readOnlyProp', {
    value: 'This is read-only',
    writable: false
});
console.log(exampleObj.readOnlyProp); // This is read-only
exampleObj.readOnlyProp = "尝试修改"; // 在严格模式下会抛出错误
console.log(exampleObj.readOnlyProp); // 仍然是: This is read-only

// Object.defineProperties() - 在对象上定义新的或修改现有的多个属性
Object.defineProperties(exampleObj, {
    prop1: { value: 42, writable: true },
    prop2: { value: '你好', writable: false }
});
console.log(exampleObj.prop1, exampleObj.prop2); // 42 '你好'

// Object.entries() - 返回对象自身可枚举属性的键值对数组
console.log(Object.entries(exampleObj)); // [['name', '测试对象'], ['value', 42],
...]
```

```
// Object.freeze() - 冻结对象，防止添加新属性和修改现有属性
Object.freeze(exampleObj);
exampleObj.newProp = "新属性"; // 在严格模式下会抛出错误
console.log(exampleObj.newProp); // undefined

// Object.fromEntries() - 将键值对列表转换为对象
let entries = [['name', 'Alice'], ['age', 30]];
let objFromEntries = Object.fromEntries(entries);
console.log(objFromEntries); // { name: 'Alice', age: 30 }

// Object.getOwnPropertyDescriptor() - 返回对象指定属性的属性描述符
console.log(Object.getOwnPropertyDescriptor(exampleObj, 'name'));
// { value: '测试对象', writable: true, enumerable: true, configurable: true }

// Object.getOwnPropertyDescriptors() - 返回对象所有自有属性的属性描述符
console.log(Object.getOwnPropertyDescriptors(exampleObj));

// Object.getOwnPropertyNames() - 返回对象所有自有属性的名称数组
console.log(Object.getOwnPropertyNames(exampleObj)); // ['name', 'value',
'readOnlyProp', 'prop1', 'prop2']

// Object.getOwnPropertySymbols() - 返回对象所有自有 Symbol 属性的数组
let sym = Symbol('testSymbol');
exampleObj[sym] = 'Symbol value';
console.log(Object.getOwnPropertySymbols(exampleObj)); // [Symbol(testSymbol)]

// Object.getPrototypeOf() - 返回指定对象的原型
console.log(Object.getPrototypeOf(exampleObj) === Object.prototype); // true

// Object.is() - 判断两个值是否相同
console.log(Object.is(NaN, NaN)); // true
console.log(Object.is(0, -0)); // false

// Object.isExtensible() - 判断对象是否可扩展
console.log(Object.isExtensible(exampleObj)); // false (因为之前被冻结了)

// Object.isFrozen() - 判断对象是否被冻结
console.log(Object.isFrozen(exampleObj)); // true

// Object.isSealed() - 判断对象是否被密封
console.log(Object.isSealed(exampleObj)); // true

// Object.keys() - 返回对象自身可枚举属性的键名数组
console.log(Object.keys(exampleObj)); // ['name', 'value', 'prop1', 'prop2']

// Object.preventExtensions() - 防止对象被扩展
let extensibleObj = { a: 1 };
Object.preventExtensions(extensibleObj);
extensibleObj.b = 2; // 在严格模式下会抛出错误
console.log(extensibleObj.b); // undefined

// Object.seal() - 密封对象，防止添加新属性并将所有现有属性标记为不可配置
let sealedObj = { x: 42 };
Object.seal(sealedObj);
sealedObj.y = 100; // 在严格模式下会抛出错误
delete sealedObj.x; // 在严格模式下会抛出错误
```

```
console.log(sealedObj); // { x: 42 }

// Object.setPrototypeOf() - 设置对象的原型
let protoObj = { protoMethod() { console.log("我是原型方法"); } };
let objWithProto = {};
Object.setPrototypeOf(objWithProto, protoObj);
objWithProto.protoMethod(); // 输出: 我是原型方法

// Object.values() - 返回对象自身可枚举属性的值的数组
console.log(Object.values(exampleObj)); // ['测试对象', 42, 42, '你好']

// 2. Object 实例方法

// hasOwnProperty() - 检查对象是否具有指定的属性
console.log(exampleObj.hasOwnProperty('name')); // true

// isPrototypeOf() - 检查一个对象是否存在于另一个对象的原型链中
console.log(Object.prototype.isPrototypeOf(exampleObj)); // true

// propertyIsEnumerable() - 判断指定属性是否可枚举
console.log(exampleObj.propertyIsEnumerable('name')); // true

// toString() - 返回对象的字符串表示
console.log(exampleObj.toString()); // [object Object]

// valueOf() - 返回指定对象的原始值
console.log(exampleObj.valueOf()); // 返回对象本身

// 3. 其他重要属性

// constructor - 指向创建此对象的构造函数
console.log(exampleObj.constructor === Object); // true

// __proto__ - 指向对象的原型（不推荐直接使用）
console.log(exampleObj.__proto__ === Object.prototype); // true
```

2. Array

属性:

- `length`: 数组的长度

方法:

- `push()`, `pop()`: 在数组末尾添加/删除元素
- `unshift()`, `shift()`: 在数组开头添加/删除元素
- `splice()`: 在指定位置添加或删除元素
- `slice()`: 返回数组的一部分
- `concat()`: 合并数组
- `join()`: 将数组元素连接成字符串
- `forEach()`, `map()`, `filter()`, `reduce()`: 用于遍历和处理数组的高阶函数

3. String

属性:

- `length`: 字符串的长度

方法:

- `charAt()`, `charCodeAt()`: 返回指定位置的字符或其Unicode值
- `substring()`, `slice()`: 提取字符串的一部分
- `indexOf()`, `lastIndexOf()`: 查找子字符串
- `toUpperCase()`, `toLowerCase()`: 大小写转换
- `trim()`: 去除字符串两端的空白
- `split()`: 将字符串分割成数组
- `replace()`: 替换字符串中的内容

4. Number

属性:

- `MAX_VALUE`, `MIN_VALUE`: 最大和最小可表示的数
- `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`: 正无穷和负无穷
- `NaN`: 非数值

方法:

- `toFixed()`: 格式化小数
- `toPrecision()`: 格式化数字的精度
- `toString()`: 将数字转换为字符串

5. Boolean

方法:

- `toString()`: 返回布尔值的字符串表示
- `valueOf()`: 返回布尔值的原始值

6. Function

属性:

- `length`: 函数期望的参数个数
- `name`: 函数的名称

方法:

- `call()`, `apply()`: 以指定的this值和参数调用函数
- `bind()`: 创建一个新函数，绑定指定的this值和部分参数

7. Date

方法:

- `getFullYear()`, `getMonth()`, `getDate()`: 获取年、月、日
- `getHours()`, `getMinutes()`, `getSeconds()`: 获取时、分、秒
- `setFullYear()`, `setMonth()`, `setDate()`: 设置年、月、日
- `toString()`, `toLocaleString()`: 返回日期的字符串表示

```
// 创建Date对象的方式
// 1.没有传入任何的参数, 获取到当前时间
var date1 = new Date()
console.log(date1)

// 2.传入参数: 时间字符串
var date2 = new Date("2022-08-08")
console.log(date2)

// 3.传入具体的年月日时分秒毫秒
var date3 = new Date(2033, 10, 10, 9, 8, 7, 333)
console.log(date3)

// 4.传入一个Unix时间戳
// 1s -> 1000ms
var date4 = new Date(10004343433)
console.log(date4)

var date = new Date()

console.log(date)
console.log(date.toDateString())
console.log(date.toISOString())


var date = new Date()

console.log(date)
console.log(date.toISOString())

// 1.获取想要的时间信息
var year = date.getFullYear()
var month = date.getMonth() + 1
var day = date.getDate()
var hour = date.getHours()
var minute = date.getMinutes()
var second = date.getSeconds()
console.log(year, month, day, hour, minute, second)
console.log(`${year}/${month}/${day} ${hour}:${minute}:${second}`)

var weekday = date.getDay() // 一周中的第几天
console.log(weekday)


// 2.也可以给date设置时间(了解)
```

```
date.setFullYear(2033)
// 自动校验
date.setDate(32)
console.log(date)


// Date对象, 转成时间戳
var date = new Date()
var date2 = new Date("2033-03-03")

// 方法一: 当前时间的时间戳
var timestamp1 = Date.now()
console.log(timestamp1)

// 方法二/三将一个date对象转成时间戳
var timestamp2 = date.getTime()
var timestamp3 = date2.valueOf()
console.log(timestamp2, timestamp3)

// 方法四: 了解
console.log(+date)


// 计算这个操作所花费的时间
var startTime = Date.now()
for (var i = 0; i < 100000; i++) {
    console.log(i)
}
var endTime = Date.now()
console.log("执行100000次for循环的打印所消耗的时间:", endTime - startTime);


// 封装一个简单函数
function testPerformance(fn) {
    var startTime = Date.now()
    fn()
    var endTime = Date.now()
    }

// Date.parse(str) 方法可以从一个字符串中读取日期，并且输出对应的Unix时间戳。
// Date.parse(str)
//  作用等同于 new Date(dateString).getTime() 操作；
//  需要符合 RFC2822 或 ISO 8601 日期格式的字符串；
//      比如YYYY-MM-DDTHH:mm:ss.sssZ
//  其他格式也许也支持，但结果不能保证一定正常；
//  如果输入的格式不能被解析，那么会返回NaN；
var time1 = Date.parse("2024-07-07T16:00:00.000Z")
console.log(time1)
```

8. RegExp

属性:

- `global`, `ignoreCase`, `multiline`: 正则表达式的标志

方法:

- `test()`: 测试字符串是否匹配正则表达式
- `exec()`: 在字符串中执行匹配搜索

9. Error

属性:

- `name`: 错误名称
- `message`: 错误信息

方法:

- `toString()`: 返回错误的字符串表示

10. Math (静态对象)

属性:

- `PI`, `E`: 数学常量

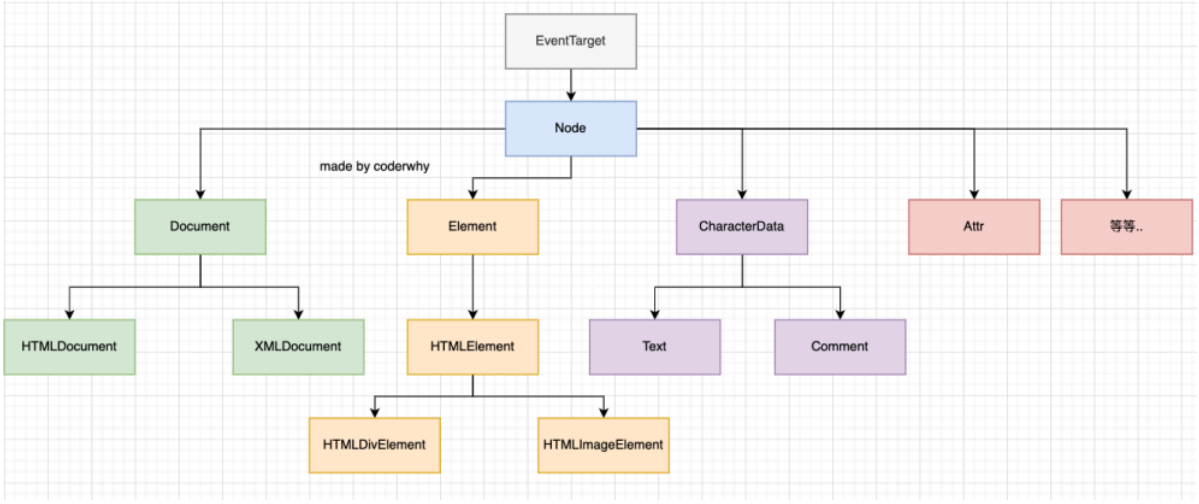
方法:

- `abs()`, `max()`, `min()`: 绝对值、最大值、最小值
- `round()`, `floor()`, `ceil()`: 四舍五入、向下取整、向上取整
- `random()`: 生成随机数
- `sin()`, `cos()`, `tan()`: 三角函数

11. DOM

> DOM全称为：文档对象模型（Document Object Model），将页面的所有内容表示为可以修改的对象。是浏览器给我们提供操作页面的API。DOM把HTML文档表示为一个树状结构，其中每个节点代表文档的一部分，比如元素、属性、文本等。

1. DOM类型继承图

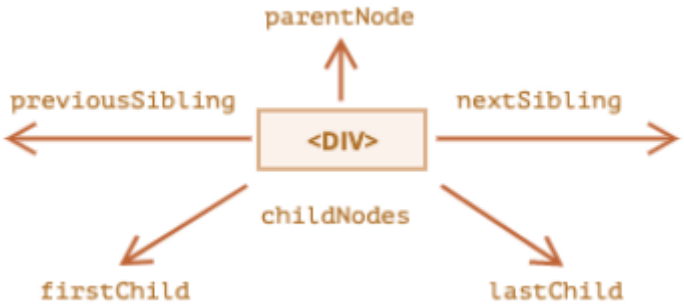


2. document对象

- Document节点表示整个载入的网页，它的实例是全局的**document对象**
  - 对DOM的所有操作都是**从document对象开始**的
  - 它是DOM的入口点，可以从**document**开始去访问任何节点元素
- 对于最顶层的html、head、body元素，我们可以直接在document对象中获取到：
  - html元素：`<html>` = document.documentElement
  - body元素：`<body>` = document.body
  - head元素：`<head>` = document.head
  - 文档声明：`<!DOCTYPE html>` = document.doctype

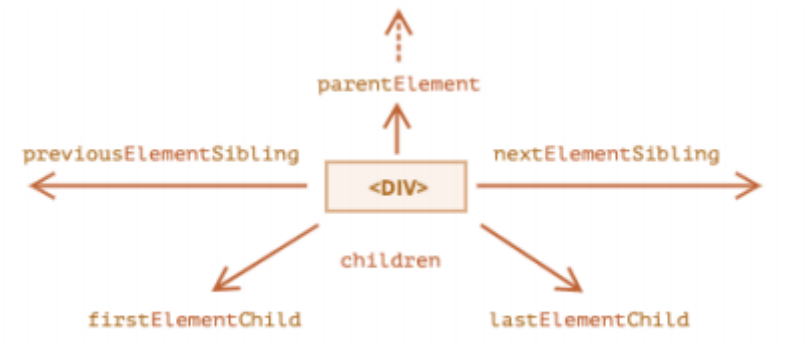
3. 节点之间的导航

- 如果我们获取到**一个节点（Node）**后，可以根据**这个节点去获取其他的节点**，我们称之为**节点之间的导航。**
- **节点之间存在如下的关系：**
  - 父节点：`parentNode`
  - 前兄弟节点：`previousSibling`
  - 后兄弟节点：`nextSibling`
  - 子节点：`childNodes`
  - 第一个子节点：`firstChild`
  - 最后一个子节点：`lastChild`



4. 元素之间的导航

- 如果我们获取到一个**元素（Element）**后，可以根据**这个元素去获取其他的元素**，我们称之为**元素之间的导航。**
- 元素之间存在如下的关系：
  - 父元素：`parentElement`
  - 前兄弟元素节点：`previousElementSibling`
  - 后兄弟元素节点：`nextElementSibling`
  - 子元素节点：`children`
  - 第一个元素子节点：`firstElementChild`
  - 最后一个元素子节点：`lastElementChild`



5. 表格元素的导航

- **table**元素支持以下这些属性：
  - **table.rows** - `<tr>` 元素的集合
  - **table.caption/tHead/tFoot** - 引用元素 `<caption> <thead> <tfoot>`
  - **table.tBodies** - `<tbody>` 元素的集合
- **thead，tfoot，tbody**元素提供了**rows**属性：
  - tbody.rows - 表格内部 `<tr>` 元素的集合
- **tr**：
  - **tr.cells** - 在给定 **tr** 中的 **td** 和 **th** 单元格的集合
  - **tr.sectionRowIndex** - 给定的 `<tr>` 在封闭的 `<thead>/<tbody>/<tfoot>` 中的位置（索引）

- **tr.rowIndex** - 在整个表格中 `<tr>` 的编号（包括表格的所有行）
- **td 和 th**
  - **td.cellIndex** - 在封闭的 `<tr>` 中单元格的编号

6. 获取元素的方法

- **DOM为我们提供了获取元素的方法**

方法名	搜索方式	可以在元素上调用	实时的
querySelector	CSS-selector	✔	-
querySelectorAll	CSS-selector	✔	-
getElementById	id	-	-
getElementsByName	name	-	✔
getElementsByTagName	tag or '*'	✔	✔
getElementsByClassName	class	✔	✔

7. 节点共有的属性

- **nodeType**属性
  - nodeType属性提供了一种获取**节点类型**的方法
  - 它有一个**数值型值**（numberic value）
- 常见的节点类型有：

常量	值	描述
Node.ELEMENT_NODE	1	一个 元素 节点，例如 `<p>` 和 `<div>`。
Node.TEXT_NODE	3	**Element** 或者 **Attr** 中实际的 文字
Node.COMMENT_NODE	8	一个 **Comment** 节点。
Node.DOCUMENT_NODE	9	一个 **Document** 节点
Node.DOCUMENT_TYPE_NODE	10	描述文档类型的 DocumentType 节点。例如 `<!DOCTYPE html>` 就是用于 HTML5 的。

- **nodeName**属性：获取node节点的名字
- **tagName**属性：获取元素的标签名词

- **tagName和 nodeName之间有什么不同呢?**

    - tagName 属性仅适用于 Element 节点
    - nodeName 是为任意 Node 定义的
        - 对于元素，它的意义与 tagName 相同，所以使用哪一个都是可以的
        - 对于其他节点类型（text，comment 等），它拥有一个对应节点类型的字符串

- **innerHTML**属性
    - 将元素中的 HTML 获取为字符串形式；
    - 设置元素中的内容；
- **outerHTML** 属性
    - 包含了元素的完整 HTML
    - 与innerHTML 加上元素本身一样的效果；
- **textContent** 属性
    - 仅仅获取元素中的文本内容；
- **innerHTML和textContent的区别：**
    - 使用 innerHTML，我们将其“作为 HTML”插入，带有所有 HTML 标签。
    - 使用 textContent，我们将其“作为文本”插入，所有符号（symbol）均按字面意义处理。

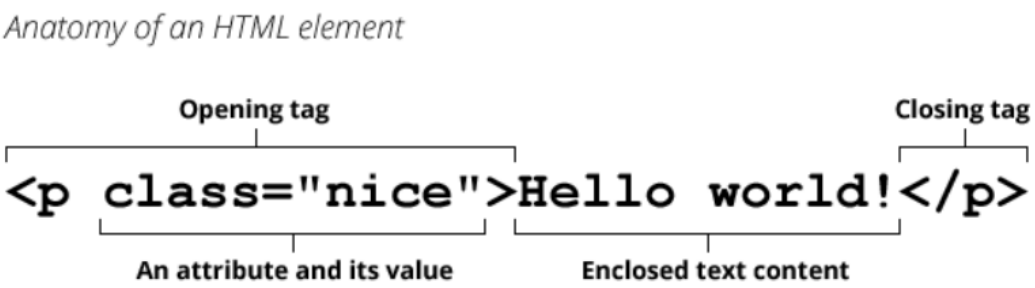
- **nodeValue**属性
    - 用于获取非元素节点的文本内容

- **hidden**属性：也是一个全局属性，可以用于设置元素隐藏（原理：display: none;）

- **DOM元素**的其他属性
    - **value**
        - `<input>`，`<select>` 和 `<textarea>`（HTMLInputElement，HTMLSelectElement……）的 value。
    - **href**
        - `<a href="...">`（HTMLAnchorElement）的 href。
    - **id**
        - 所有元素（HTMLElement）的 “id” 特性（attribute）的值。

8. 元素的属性（property）和特性（attribute）

- 我们知道，一个元素除了有**开始标签、结束标签、内容**之外，还有很多的**属性（attribute）**

- 

- 浏览器在解析HTML元素时，会将**对应的attribute**也创建出来放到**对应的元素对象**上
  - 比如**id、class就是全局的attribute**，会有对应的**id、class属性**；
  - 比如**href属性**是针对**a元素**的，**type、value属性**是针对**input元素**的；
- **属性attribute的分类：**
  - **标准的attribute**：某些attribute属性是标准的，比如id、class、href、type、value等；
  - **非标准的attribute**：某些attribute属性是自定义的，比如abc、age、height等；
  - ```

        hehe

    ```

- **对于所有的attribute访问都支持如下的方法**
  - **elem.hasAttribute(name)** — 检查特性是否存在。
  - **elem.getAttribute(name)** — 获取这个特性值。
  - **elem.setAttribute(name, value)** — 设置这个特性值。
  - **elem.removeAttribute(name)** — 移除这个特性。
  - **attributes**：attr对象的集合，具有name、value属性；
  - ```
    for(var attr of boxEl.attributes) {
        console.log(attr.name, attr.value)
    }
    console.log(boxEl.hasAttribute("age"))
    console.log(boxEl.getAttribute("name"))
    boxEl.setAttribute("name", "kobe")
    boxEl.removeAttribute("abc")
    ```

- **attribute具备以下特征：**
  - 它们的**名字是大小写不敏感**的（id 与 ID 相同）。
  - 它们的**值总是字符串类型**的。

- 对于**标准的attribute**，会在DOM对象上创建**与其对应的property属性**
  - ```
    console.log(boxEl.id, boxEl.className) // box main
    console.log(boxEl.abc, boxEl.age, boxEl.height) // undefined...
    ```
    ```

- **在大多数情况下，它们是相互作用的**
  - 改变**property**，通过**attribute**获取的值，会随着改变；
  - 通过**attribute**操作修改，**property**的值会随着改变；
    - 但是input的value修改只能通过attribute的方法；
- **除非特别情况，大多数情况下，设置、获取attribute，推荐使用property的方式：**
  - 这是因为它默认情况下是有类型的；
  - ```
    toggleBtn.onclick = function() {
        checkBoxInput.cheked = !checkBoxInput.checked
    }
    ```

- **HTML5的data-*自定义属性**
  - 它们可以在dataset属性中获取到的：
  - ```
    <div class="box" data-name="why" data-age="18"></div>
    <script>
        var boxEl = document.querySelector(".box")
        console.log(boxEl.dataset.name)
        console.log(boxEl.dataset.age)
    </script>
    ```

9. js动态修改样式

- **有时候我们会通过JavaScript来动态修改样式，这个时候我们有两个选择：**
  - 选择一：在CSS中编写好对应的样式，**动态的添加class；**
  - 选择二：**动态的修改style属性；**
- **开发中如何选择呢？**
  - 在大多数情况下，如果可以动态修改class完成某个功能，更**推荐使用动态class；**
  - 如果对于某些情况，无法通过动态修改class（比如精准修改某个css属性的值），那么就可以**修改style属性；**
- **元素的className和classList**
  - 元素的class attribute，对应的property并非叫class，而是**className**：
    - 这是因为JavaScript早期是不允许使用class这种关键字来作为对象的属性，所以DOM规范使用了className；
    - 虽然现在JavaScript已经没有这样的限制，但是并不推荐，并且依然在使用className这个名称；
  - 我们可以对className进行赋值，它会替换整个类中的字符串。
    - ```
      var boxEl = document.querySelector(".box")
      boxEl.className = "abc why"
      ```
  - 如果我们需要添加或者移除单个的class，那么可以使用classList属性。
      ```

- elem.classList 是一个特殊的对象：
  - elem.classList.add (class)：添加一个类
  - elem.classList.remove(class)：添加/移除类。
  - elem.classList.toggle(class)：如果类不存在就添加类，存在就移除它。
  - elem.classList.contains(class)：检查给定类，返回 true/false。
- classList是**可迭代对象**，可以通过**for of**进行遍历。

- **元素的style属性**
  - **如果需要单独修改某一个CSS属性，那么可以通过style来操作：**
    - 对于多词（multi-word）属性，使用驼峰式 camelCase
    - ```
      boxEl.style.width = "100px"
      boxEl.style.height = "50px"
      boxEl.style.backgroundColor = "red"
      ```
  - 如果我们将值设置为**空字符串**，那么会使用**CSS的默认样式**：
    - ```
      boxEl.style.display = ""
      ```
  - 多个样式的写法，我们需要使用cssText属性；
    - 不推荐这种用法，因为它会替换整个字符串
    - ```
      boxEl.style.cssText = `width: 100px; height: 100px; background-color: red;`
      ```

10. 元素style的读取 - getComputedStyle

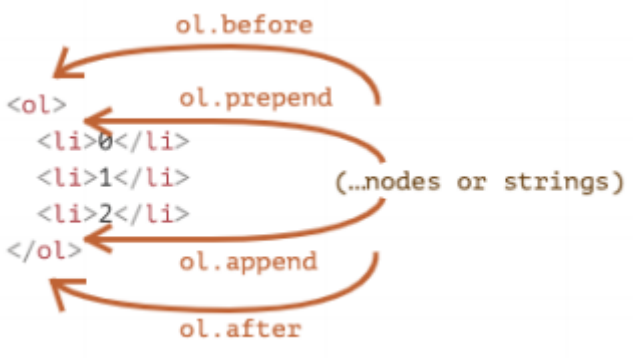
- **如果我们需要读取样式**
  - 对于**内联样式**，是可以通过**style.***的方式读取到的
  - 对于**style、css文件中**的样式，是**读取不到**的
- 这个时候，我们可以通过**getComputedStyle**的全局函数来实现
  - ```
    console.log(getComputedStyle(boxEl).width)
    console.log(getComputedStyle(boxEl).height)
    console.log(getComputedStyle(boxEl).backgroundColor)
    ```

11. 创建元素

- document.createElement(tag)

12. 插入元素

- 插入元素的方式如下：
  - **node.append(...nodes or strings)** —— 在 node 末尾 插入节点或字符串，
  - **node.prepend(...nodes or strings)** —— 在 node 开头 插入节点或字符串，
  - **node.before(...nodes or strings)** —— 在 node 前面 插入节点或字符串，
  - **node.after(...nodes or strings)** —— 在 node 后面 插入节点或字符串，
  - **node.replaceWith(...nodes or strings)** —— 将 node 替换为给定的节点或字符串。



13. 移除和克隆元素

- **移除元素我们可以调用元素本身的remove方法**

```
boxEl.remove()
```

- **如果我们想要复制一个现有的元素，可以通过cloneNode方法**
  - 可以传入一个**Boolean类型**的值，来决定是否是**深度克隆**；
  - 深度克隆会克隆对应元素的子元素，否则不会；

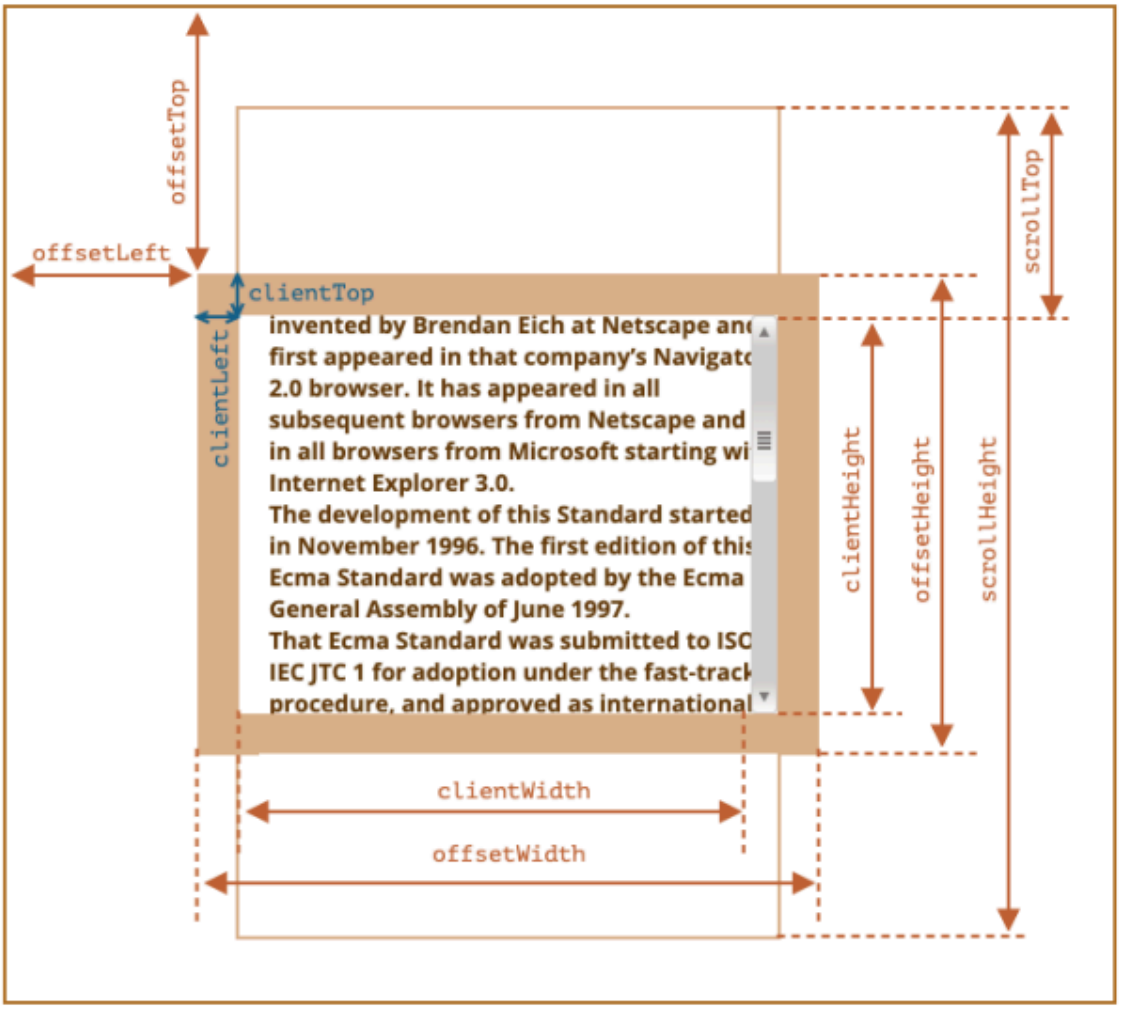
```
var cloneBoxEl = boxEl.cloneNode(true)
document.body.append(cloneBoxEl)
```

14. 旧的元素操作方法

- parentElem.appendChild(node)
  - 在parentElem的父元素最后位置添加一个子元素
- parentElem.insertBefore(node, nextSibling)
  - 在parentElem的nextSibling前面插入一个子元素
- parentElem.replaceChild(node, oldChild)
  - 在parentElem中，新元素替换之前的oldChild元素
- parentElem.removeChild(node)
  - 在parentElem中，移除某一个元素；

15. 元素的大小、滚动

- clientWidth：contentWith+padding（不包含滚动条）
- clientHeight：contentHeight+padding
- clientTop：border-top的宽度
- clientLeft：border-left的宽度
- offsetWidth：元素完整的宽度
- offsetHeight：元素完整的高度
- offsetLeft：距离父元素的x
- offsetHeight：距离父元素的y
- scrollHeight：整个可滚动的区域高度
- scrollTop：滚动部分的高度



16. window的大小、滚动

- **window的width和height**
  - innerWidth、innerHeight：获取window窗口的宽度和高度（包含滚动条）
  - outerWidth、outerHeight：获取window窗口的整个宽度和高度（包括调试工具、工具栏）
  - documentElement.clientHeight、documentElement.clientWidth：获取html的宽度和高度（不包含滚动条）

- **window的滚动位置**
  - scrollX：X轴滚动的位置（别名pageXOffset）
  - scrollY：Y轴滚动的位置（别名pageYOffset）
- **也有提供对应的滚动方法**
  - 方法 scrollBy(x,y)： 将页面滚动至 相对于当前位置的 (x, y) 位置；
  - 方法 scrollTo(pageX,pageY) 将页面滚动至 绝对坐标；

17. 事件（Event）

- **Web页面需要经常和用户之间进行交互，而交互的过程中我们可能想要捕捉这个交互的过程：**
  - 比如用户点击了某个按钮、用户在输入框里面输入了某个文本、用户鼠标经过了某个位置；
  - 浏览器需要搭建一条JavaScript代码和事件之间的桥梁；
  - 当某个事件发生时，让JavaScript可以相应（执行某个函数），所以我们需要针对事件编写处理程序（handler）；
- **事件监听的三种方式**
  1. 在script中直接监听（很少使用）；
  2. DOM属性，通过元素的on来监听事件；
  3. 通过EventTarget中的addEventListener来监听；

```
<!-- 直接在html中编写JavaScript代码(了解) -->
<button onclick="console.log('按钮1发生了点击~');">按钮1</button>
<script>
    // 2. onclick属性
    function handleClick01() {
        console.log("按钮2发生了点击~")
    }
    btn2El.onclick = handleClick01

    // 3. addEventListener(推荐)
    btn3El.addEventListener("click", function() {
        console.log("第一个btn3的事件监听~")
    })
    btn3El.addEventListener("click", function() {
        console.log("第二个btn3的事件监听~")
    })
    btn3El.addEventListener("click", function() {
        console.log("第三个btn3的事件监听~")
    })
</script>
```

18. 常见事件列表

- **鼠标事件**
  - click —— 当鼠标点击一个元素时（触摸屏设备会在点击时生成）。
  - mouseover / mouseout —— 当鼠标指针移入/离开一个元素时。
  - mousedown / mouseup —— 当在元素上按下/释放鼠标按钮时。
  - mousemove —— 当鼠标移动时。
- **键盘事件**
  - keydown 和 keyup —— 当按下和松开一个按键时。
- **表单（form）元素事件**
  - submit —— 当访问者提交了一个`<form>`时
  - focus —— 当访问者聚焦于一个元素时，例如聚焦于一个`<input>`
- **Document 事件**
  - DOMContentLoaded —— 当 HTML 的加载和处理均完成，DOM 被完全构建完成时。
- **CSS 事件**
  - transitionend —— 当一个 CSS 动画完成时

19. 认识事件流

- **事实上对于事件有一个概念叫做事件流，为什么会产生事件流呢?**
  - 我们可以想到一个问题：当我们在浏览器上对着一个元素点击时，你点击的不仅仅是这个元素本身；
  - 这是因为我们的HTML元素是存在父子元素叠加层级的；
  - 比如一个span元素是放在div元素上的，div元素是放在body元素上的，body元素是放在html元素上的；

```
<div class="box">
    <span></span>
</div>

<script>

    // 1.获取元素
    var spanEl = document.querySelector("span")
    var divEl = document.querySelector("div")
    var bodyEl = document.body


    // 2.绑定点击事件
    // spanEl.onclick = function() {
    //   console.log("span元素发生了点击~")
    // }
    // divEl.onclick = function() {
    //   console.log("div元素发生了点击~")
    // }
    // bodyEl.onclick = function() {
```

```
    //   console.log("body元素发生了点击~")
    // }

    // 默认情况下是事件冒泡
    spanEl.addEventListener("click", function() {
        console.log("span元素发生了点击~冒泡")
    })
    divEl.addEventListener("click", function() {
        console.log("div元素发生了点击~冒泡")
    })
    bodyEl.addEventListener("click", function() {
        console.log("body元素发生了点击~冒泡")
    })

    // 设置希望监听事件捕获的过程
    spanEl.addEventListener("click", function() {
        console.log("span元素发生了点击~捕获")
    }, true)
    divEl.addEventListener("click", function() {
        console.log("div元素发生了点击~捕获")
    }, true)
    bodyEl.addEventListener("click", function() {
        console.log("body元素发生了点击~捕获")
    }, true)

</script>
```

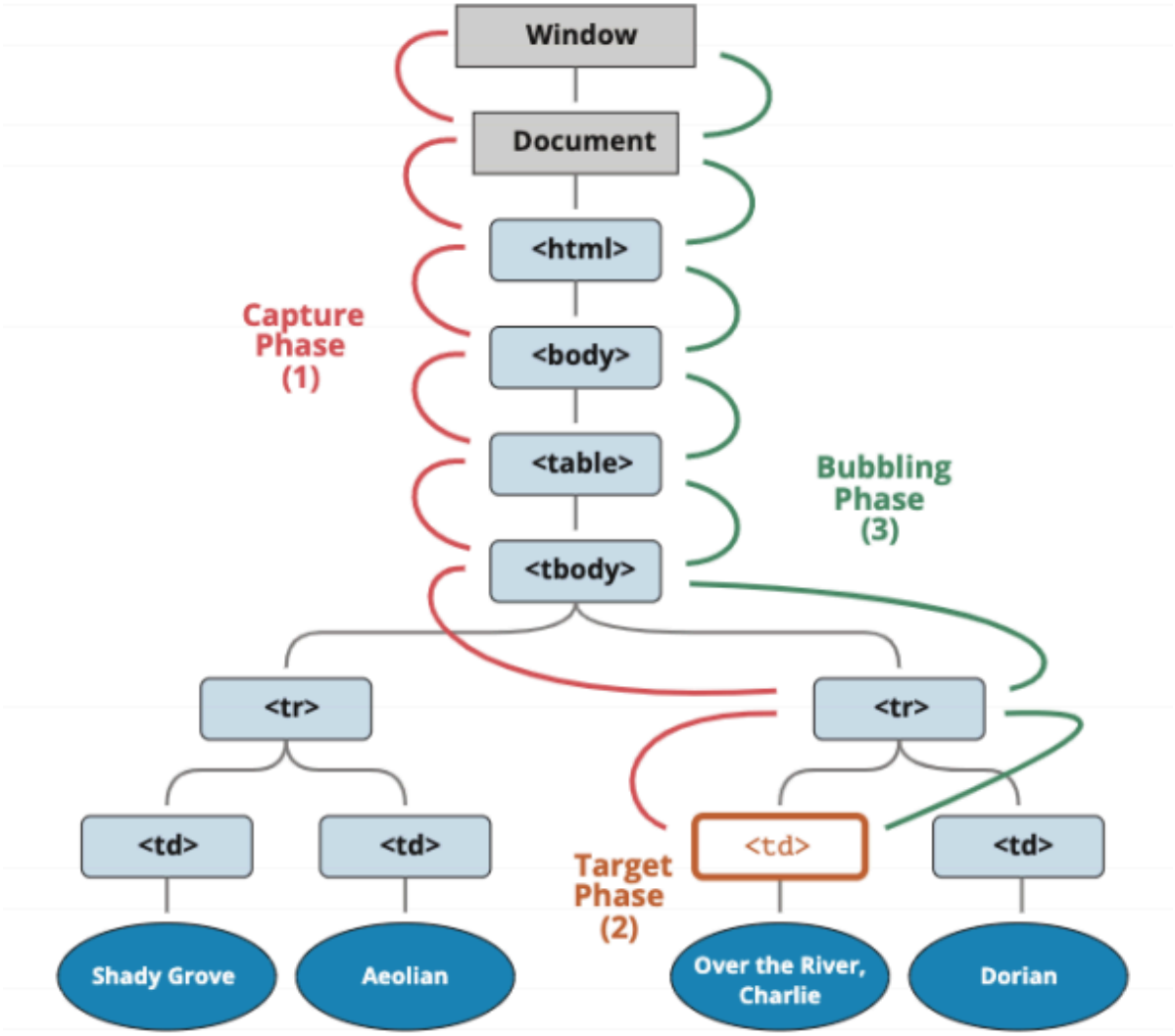
20. 事件冒泡和事件捕获

- 我们会发现默认情况下事件是**从最内层的span向外依次传递的顺序**，这个顺序我们称之为**事件冒泡（Event Bubble）**；
- 事实上，还有另外一种监听事件流的方式就是**从外层到内层（body -> span）**，这种称之为**事件捕获（Event Capture）**；
- 为什么会产生两种不同的处理流呢?
    - 这是因为早期浏览器开发时，不管是IE还是Netscape公司都发现了这个问题;
    - 但是他们采用了完全相反的事件流来对事件进行了传递;
    - **IE**采用了**事件冒泡**的方式，**Netscape**采用了**事件捕获**的方式;

21. 事件捕获和冒泡的过程

- **如果我们都监听，那么会按照如下顺序来执行**
- **捕获阶段（Capturing phase）**：
    - 事件（从 Window）向下走近元素。
- **目标阶段（Target phase）**：
    - 事件到达目标元素。
- **冒泡阶段（Bubbling phase）**：
    - 事件从元素上开始冒泡。

- 事实上，我们可以通过event对象来获取当前的阶段：
  - eventPhase
- 开发中通常会使用**事件冒泡**，所以事件捕获了解即可



22. 事件对象

- **当一个事件发生时，就会有和这个事件相关的很多信息：**
  - 比如事件的类型是什么，你点击的是哪一个元素，点击的位置是哪里等等相关的信息；
  - 那么这些信息会被封装到一个Event对象中，这个对象由浏览器创建，称之为event对象；
  - 该对象给我们提供了想要的一些属性，以及可以通过该对象进行某些操作；
- 如何获取这个event对象？
  - **event对象**会在传入的**事件处理（event handler）函数回调时，被系统传入**
  - 我们可以在回调函数中拿到这个event对象；

```
spanEl.onclick = function(event) {
    console.log("事件对象:", event)
}

spanEl.addEventListener("click", function(event) {
    console.log("事件对象:", event)
})
```

- **event对象常见的属性**
  - type：事件的类型；
  - target：当前事件发生的元素；
  - currentTarget：当前处理事件的元素；
  - eventPhase：事件所处的阶段；
  - offsetX、offsetY：事件发生在元素内的位置；
  - clientX、clientY：事件发生在客户端内的位置；
  - pageX、pageY：事件发生在客户端相对于document的位置；
  - screenX、screenY：事件发生相对于屏幕的位置；
- **event对象常见的方法**
  - preventDefault：取消事件的默认行为；
  - stopPropagation：阻止事件的进一步传递（冒泡或者捕获都可以阻止）；

23. 事件处理中的this

- 在函数中，我们也可以通过this来获取当前的发生元素：

```
boxEl.addEventListener("click", function(event) {
    console.log(this === event.target)
})
```

- 这是因为在浏览器内部，调用**event handler是绑定到当前的target上的**

24. EventTarget类

- **我们会发现，所有的节点、元素都继承自EventTarget**
  - 事实上Window也继承自EventTarget；

- **那么这个EventTarget是什么呢?**
  - EventTarget是一个**DOM接口**，主要用于**添加、删除、派发Event事件**；
- **EventTarget常见的方法**
  - **addEventListener**：注册某个事件类型以及事件处理函数；
  - **removeEventListener**：移除某个事件类型以及事件处理函数；

- **dispatchEvent**：派发某个事件类型到EventTarget上；

25. 事件委托（event delegation）

- 事件冒泡在某种情况下可以帮助我们实现强大的事件处理模式 –**事件委托模式**（也是一种设计模式）
- 那么这个模式是怎么样的呢？
  - 因为**当子元素被点击**时，父元素可以**通过冒泡可以监听到子元素的点击**；
  - 并且**可以通过event.target获取到当前监听的元素**；
- 案例：一个ul中存放多个li，点击某一个li会变成红色
  - 方案一：监听**每一个li的点击，并且做出相应；**
  - 方案二：在**ul中监听点击**，并且**通过event.target拿到对应的li进行处理；**
    - 因为这种方案并不需要遍历后给每一个li上添加事件监听，所以它更加高效；

```
var listEl = document.querySelector(".list")
var currentActive = null
listEl.addEventListener("click", function(event) {
    if(currentActive) currentActive.classList.remove("active")
    event.target.classList.add("active")
    currentActive = event.target
})
```

26. 事件委托的标记

- 某些事件委托可能需要对具体的子组件进行区分，这个时候我们可以使用*data-*对其进行标记：
- 比如多个按钮的点击，区分点击了哪一个按钮：

```
<div class="box">
    <button data-action="search">搜索~</button>
    <button data-action="new">新建~</button>
    <button data-action="remove">移除~</button>
    <button>1111</button>
</div>

<script>

    var boxEl = document.querySelector(".box")
    boxEl.onclick = function(event) {
        var btnEl = event.target
        var action = btnEl.dataset.action
        switch (action) {
            case "remove":
                console.log("点击了移除按钮")
                break
            case "new":
                console.log("点击了新建按钮")
                break
```

```
            case "search":
                console.log("点击了搜索按钮")
                break
            default:
                console.log("点击了其他")
        }
    }
</script>
```

27. 常见的鼠标事件

- 接下来我们来看一下常见的鼠标事件（不仅仅是鼠标设备，也包括模拟鼠标的设备，比如手机、平板电脑）
- 常见的鼠标事件：

属性	描述
click	当用户点击某个对象时调用的事件句柄。
contextmenu	在用户点击鼠标右键打开上下文菜单时触发
dblclick	当用户双击某个对象时调用的事件句柄
mousedown	鼠标按钮被按下。
mouseup	鼠标按键被松开。
mouseover	鼠标移到某元素之上。（支持冒泡）
mouseout	鼠标从某元素移开。（支持冒泡）
mouseenter	当鼠标指针移动到元素上时触发。（不支持冒泡）
mouseleave	当鼠标指针移出元素时触发。（不支持冒泡）
mousemove	鼠标被移动。

- **mouseenter和mouseleave**
  - 不支持冒泡
  - 进入子元素依然属于在该元素内，没有任何反应
- **mouseover和mouseout**
  - 支持冒泡
  - 进入元素的子元素时
    - 先调用父元素的mouseout
    - 再调用子元素的mouseover
    - 因为支持冒泡，所以会将mouseover传递到父元素中；

28. 常见的键盘事件

- **常见的键盘事件**

属性	描述
onkeydown	某个键盘按键被按下。
onkeypress	某个键盘按键被按下。
onkeyup	某个键盘按键被松开。

- **事件的执行顺序是 onkeydown、onkeypress、onkeyup**
  - down事件先发生；
  - press发生在文本被输入；
  - up发生在文本输入完成；
- **我们可以通过key和code来区分按下的键：**
  - code：“按键代码”（"KeyA"，"ArrowLeft" 等），特定于键盘上按键的物理位置。
  - key：字符（"A"，"a" 等），对于非字符（non-character）的按键，通常具有与 code 相同的值。）

29. 常见表单事件

属性	描述
onchange	该事件在表单元素的内容改变时触发(`<input>`, `<keygen>`, `<select>`, `<textarea>`)
oninput	元素获取用户输入时触发
onfocus	元素获取焦点时触发
onblur	元素失去焦点时触发
onreset	表单重置时触发
onsubmit	表单提交时触发

30. 文档加载事件

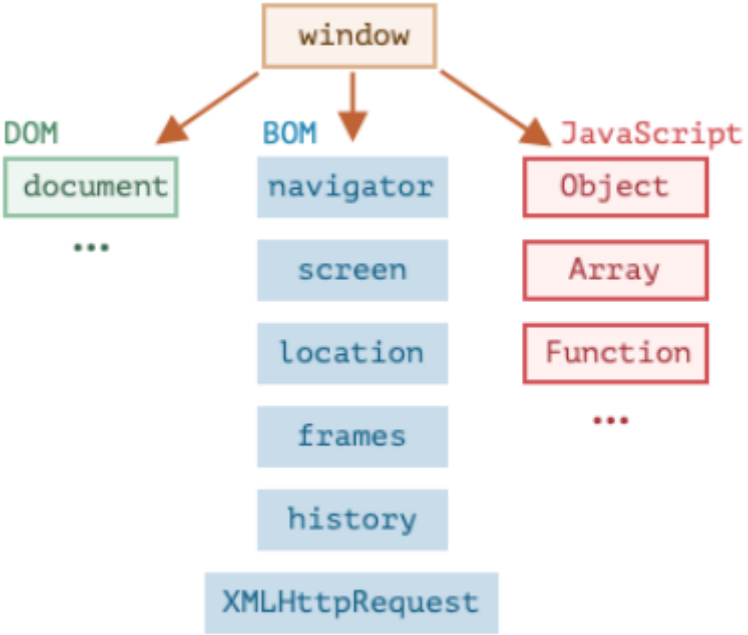
- **DOMContentLoaded**：浏览器已完全加载 HTML，并构建了 DOM 树，但像 `<img>` 和样式表之类的外部资源可能尚未加载完成。
- **load**：浏览器不仅加载完成了 HTML，还加载完成了所有外部资源：图片，样式等。

```
<script>

    // 注册事件监听
    window.addEventListener("DOMContentLoaded", function() {
```

```
        // 1.这里可以操作box, box已经加载完毕
        // var boxEl = document.querySelector(".box")
        // boxEl.style.backgroundColor = "orange"
        // console.log("HTML内容加载完毕")

        // 2.获取img对应的图片的宽度和高度
        var imgEl = document.querySelector("img")
        console.log("图片的宽度和高度:", imgEl.offsetWidth, imgEl.offsetHeight)
    })

    window.onload = function() {
        console.log("文档中所有资源都加载完毕")
        // var imgEl = document.querySelector("img")
        // console.log("图片的宽度和高度:", imgEl.offsetWidth,
imgEl.offsetHeight)
    }

    window.onresize = function() {
        console.log("创建大小发生改变时")
    }

</script>

<div class="box">
    <p>哈哈哈啊</p>
</div>
<a href="#">百度一下</a>
<img src="../images/kobe01.jpg" alt="">
```

12. BOM：浏览器对象模型（Browser Object Model）

> 简称 BOM，由浏览器提供的用于处理文档（document）之外的所有内容的其他对象；
>
> 比如navigator、location、history等对象；



- JavaScript有一个非常重要的运行环境就是浏览器
  - 而且浏览器本身又作为一个应用程序需要对其本身进行操作；
  - 所以通常浏览器会有对应的对象模型（BOM，Browser Object Model）；
  - 我们可以将BOM看成是连接JavaScript脚本与浏览器窗口的桥梁；
- BOM主要包括一下的对象模型：
  - window：包括全局属性、方法，控制浏览器窗口相关的属性、方法；
  - location：浏览器连接到的对象的位置（URL）；
  - history：操作浏览器的历史；
  - navigator：用户代理（浏览器）的状态和标识（很少用到）；
  - screen：屏幕窗口信息（很少用到）；

1. window对象

- **window对象在浏览器中可以从两个视角来看待：**
  - 视角一：全局对象。
    - 我们知道ECMAScript其实是有一个全局对象的，这个全局对象在**Node中是global**；
    - 在浏览器中就是**window对象；**
  - 视角二：浏览器窗口对象。
    - 作为浏览器窗口时，提供了对浏览器操作的相关的API；
- **当然，这两个视角存在大量重叠的地方，所以不需要刻意去区分它们：**
  - 事实上对于浏览器和Node中全局对象名称不一样的情况，目前已经指定了对应的标准，称之为globalThis，并且大多数现代浏览器都支持它；
  - 放在window对象上的所有属性都可以被访问；
  - 使用var定义的变量会被添加到window对象中；
  - window默认给我们提供了全局的函数和类：setTimeout、Math、Date、Object等；
- **事实上window对象上肩负的重担是非常大的：**
  - 第一：包含大量的属性，localStorage、console、location、history、screenX、scrollX等等（大概60+个属性）；
  - 第二：包含大量的方法，alert、close、scrollTo、open等等（大概40+个方法）；
  - 第三：包含大量的事件，focus、blur、load、hashchange等等（大概30+个事件）；
  - 第四：包含从EventTarget继承过来的方法，addEventListener、removeEventListener、dispatchEvent方法；
- **那么这些大量的属性、方法、事件在哪里查看呢?**
  - MDN文档：https://developer.mozilla.org/zh-CN/docs/Web/API/Window
- **查看MDN文档时，我们会发现有很多不同的符号，这里我解释一下是什么意思：**
  - 删除符号：表示这个API已经废弃，不推荐继续使用了；
  - 点踩符号：表示这个API不属于W3C规范，某些浏览器有实现（所以兼容性的问题）；
  - 实验符号：该API是实验性特性，以后可能会修改，并且存在兼容性问题；

`window` 对象是浏览器中的全局对象，表示浏览器窗口，提供了大量的属性和方法，方便开发者操作浏览器、文档和与用户交互。以下是一些常见的属性和方法。

一、常见属性

1. `window.document`
   - 描述：指向当前页面的DOM对象，常用于操作HTML文档结构。
   - 示例：

```
console.log(window.document.title); // 获取页面标题
```

2. `window.innerWidth` & `window.innerHeight`
   - 描述：表示浏览器窗口的内容区域（不包括工具栏和滚动条）的宽度和高度，单位为像素。
   - 示例：

```
console.log(window.innerWidth, window.innerHeight); // 输出窗口宽度和高度
```

3. `window.outerWidth` & `window.outerHeight`
   - 描述：浏览器窗口的整体宽度和高度，包含工具栏、菜单栏等。
   - 示例：

```
console.log(window.outerWidth, window.outerHeight); // 输出浏览器整体的宽高
```

4. `window.location`
   - 描述：用于获取或设置当前页面的URL信息。
   - 示例：

```
console.log(window.location.href); // 获取当前页面URL
window.location.href = "https://example.com"; // 跳转到指定URL
```

5. `window.navigator`
   - 描述：提供有关浏览器的信息，如用户代理、平台等。
   - 示例：

```
console.log(window.navigator.userAgent); // 获取用户代理字符串
```

6. `window.history`
   - 描述：允许你操作浏览器的会话历史记录。
   - 示例：

```
window.history.back(); // 回到上一页
window.history.forward(); // 前进到下一页
```

7. `window.localStorage` & `window.sessionStorage`
   - 描述：提供在客户端存储数据的方式，`localStorage` 是持久的，`sessionStorage` 是会话的。

- 示例：

```
window.localStorage.setItem('key', 'value'); // 存储数据到localStorage
console.log(window.localStorage.getItem('key')); // 获取存储的数据
```

8. `window.screen`
   - 描述：包含有关用户显示屏的信息，如分辨率、颜色深度等。
   - 示例：

```
console.log(window.screen.width, window.screen.height); // 获取屏幕分辨率
```

9. `window.scrollX` & `window.scrollY`
   - 描述：获取页面的水平和垂直滚动偏移量。
   - 示例：

```
console.log(window.scrollX, window.scrollY); // 输出页面的滚动位置
```

10. `window.console`
    - 描述：提供了对开发者控制台的访问接口，用于调试和输出信息。
    - 示例：

```
window.console.log('Hello, World!'); // 输出信息到控制台
```

二、常见方法

1. `window.alert()`
   - 描述：弹出一个带有消息的警告框。
   - 示例：

```
window.alert('This is an alert!');
```

2. `window.confirm()`
   - 描述：弹出一个带有确认和取消按钮的对话框，返回布尔值。
   - 示例：

```
const result = window.confirm('Are you sure?');
console.log(result); // 如果点击确定，返回true；否则返回false
```

3. `window.prompt()`
   - 描述：弹出一个带有输入框的对话框，允许用户输入值，返回输入的内容。
   - 示例：

```
const name = window.prompt('What is your name?');
console.log(name); // 输出用户输入的内容
```

4. `window.open()`

- 描述：打开一个新的浏览器窗口或标签页。
- 示例：

```
window.open('https://example.com', '_blank'); // 在新窗口中打开指定URL
```

5. `window.close()`
   - 描述：关闭当前的浏览器窗口（仅对由`window.open()`打开的窗口有效）。
   - 示例：

```
window.close(); // 关闭当前窗口
```

6. `window.setTimeout()`
   - 描述：在指定的延迟时间后执行代码，仅执行一次。
   - 示例：

```
window.setTimeout(() => {
  console.log('This message appears after 2 seconds');
}, 2000); // 2秒后执行
```

7. `window.setInterval()`
   - 描述：每隔一段时间重复执行代码，直到手动停止。
   - 示例：

```
const intervalId = window.setInterval(() => {
  console.log('This message appears every 3 seconds');
}, 3000); // 每3秒执行一次
```

8. `window.clearTimeout()` & `window.clearInterval()`
   - 描述：分别用于停止由`setTimeout()`或`setInterval()`启动的计时器。
   - 示例：

```
const timeoutId = window.setTimeout(() => {
  console.log('This will not appear');
}, 5000);
window.clearTimeout(timeoutId); // 停止计时器，代码不会执行
```

9. `window.scrollTo()`
   - 描述：将窗口滚动到指定的坐标位置。
   - 示例：

```
window.scrollTo(0, 500); // 滚动到页面顶部500像素处
```

10. `window.addEventListener()`
    - 描述：为窗口对象注册事件监听器。
    - 示例：

```
window.addEventListener('resize', () => {
  console.log('Window resized');
}); // 当窗口大小变化时执行代码
```

`window` 对象是 JavaScript 中的全局对象，表示浏览器的窗口或框架。`window` 上有许多常用的事件可以响应用户或浏览器的行为。以下是一些常见的 `window` 事件：

三、常见事件

1. **load**
   - **描述**：当整个页面（包括所有依赖的资源，如图像、样式表等）加载完毕时触发。
   - **用法**：

```
window.addEventListener('load', function() {
    console.log('页面完全加载');
});
```

2. **DOMContentLoaded**
   - **描述**：当 HTML 文档被完全加载和解析时触发，不需要等待样式表、图像等外部资源加载完毕。
   - **用法**：

```
document.addEventListener('DOMContentLoaded', function() {
    console.log('DOM 内容加载完成');
});
```

3. **resize**
   - **描述**：当浏览器窗口大小改变时触发。
   - **用法**：

```
window.addEventListener('resize', function() {
    console.log('窗口大小改变');
});
```

4. **scroll**
   - **描述**：当页面滚动时触发。
   - **用法**：

```
window.addEventListener('scroll', function() {
    console.log('页面正在滚动');
});
```

5. **unload**
   - **描述**：当页面或文档从浏览器窗口卸载时触发（通常用于页面关闭或导航离开）。
   - **用法**：

```
window.addEventListener('unload', function() {
    console.log('页面将被卸载');
});
```

6. **beforeunload**
   - **描述**：当用户试图离开页面时触发，可以用来显示确认对话框，提示用户保存未完成的工作。
   - **用法**：

```
window.addEventListener('beforeunload', function (event) {
    event.preventDefault();
    event.returnValue = '';  // 显示确认对话框
});
```

7. **error**
   - **描述**：当页面中的 JavaScript 发生错误时触发。
   - **用法**：

```
window.addEventListener('error', function(event) {
    console.error('发生错误:', event.message);
});
```

8. **focus**
   - **描述**：当浏览器窗口或框架获得焦点时触发。
   - **用法**：

```
window.addEventListener('focus', function() {
    console.log('窗口获得焦点');
});
```

9. **blur**
   - **描述**：当浏览器窗口或框架失去焦点时触发。
   - **用法**：

```
window.addEventListener('blur', function() {
    console.log('窗口失去焦点');
});
```

10. **hashchange**
    - **描述**：当 URL 的哈希部分（即 `#` 之后的部分）发生变化时触发。
    - **用法**：

```
window.addEventListener('hashchange', function() {
    console.log('URL 哈希变化:', location.hash);
});
```

11. **popstate**
    - **描述**：当浏览器历史记录中的活动条目发生变化时触发（例如点击回退按钮）。

- **用法**:

```
window.addEventListener('popstate', function(event) {
    console.log('浏览器历史记录发生变化', event.state);
});
```

12. **online**
    - **描述**：当浏览器重新连接到网络时触发。
    - **用法**:

```
window.addEventListener('online', function() {
    console.log('网络连接恢复');
});
```

13. **offline**
    - **描述**：当浏览器失去网络连接时触发。
    - **用法**:

```
window.addEventListener('offline', function() {
    console.log('网络连接断开');
});
```

14. **storage**
    - **描述**：当 `localStorage` 或 `sessionStorage` 发生改变时触发（但只有在不同页面之间变化时才会触发）。
    - **用法**:

```
window.addEventListener('storage', function(event) {
    console.log('存储数据发生变化:', event.key, event.newValue);
});
```

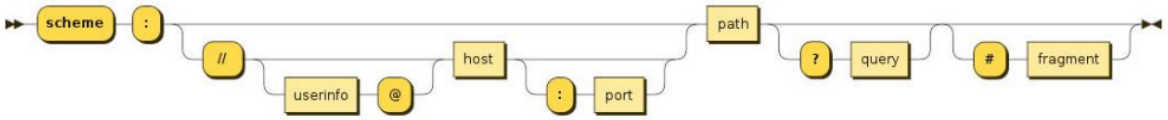
这些是一些常见的 `window` 事件，可以用于处理页面加载、大小变化、滚动、历史记录变化、网络状态变化等。通过监听这些事件，开发者可以创建更加互动和响应式的 Web 应用。

总结

`window`对象提供了与浏览器相关的大量接口，允许开发者控制窗口、访问文档、存储数据、处理事件等。熟悉这些常见属性和方法，可以帮助更好地开发网页应用。

2. location对象

> `location` 对象是 JavaScript 中的全局对象，代表当前文档的 URL，并提供了处理和修改 URL 的方法和属性。`location` 对象常用于重定向、导航和获取 URL 相关信息。location其实是URL的一个抽象实现；以下是 `location` 对象常见的属性和方法：



一、常见属性

1. `location.href`
   - **描述**：获取或设置整个 URL。
   - **用法**：

     ```
     console.log(location.href);  // 获取当前 URL
     location.href = 'https://www.example.com';  // 跳转到新的 URL
     ```

2. `location.protocol`
   - **描述**：返回当前 URL 的协议部分（包括 `:`），例如 `http:` 或 `https:`。
   - **用法**：

     ```
     console.log(location.protocol);  // 输出: "https:"
     ```

3. `location.host`
   - **描述**：返回当前 URL 的主机名和端口号，例如 `www.example.com:8080`。
   - **用法**：

     ```
     console.log(location.host);  // 输出: "www.example.com:8080"
     ```

4. `location.hostname`
   - **描述**：返回当前 URL 的主机名，不包含端口号，例如 `www.example.com`。
   - **用法**：

     ```
     console.log(location.hostname);  // 输出: "www.example.com"
     ```

5. `location.port`
   - **描述**：返回当前 URL 的端口号。如果 URL 没有指定端口号，则返回空字符串。
   - **用法**：

     ```
     console.log(location.port);  // 输出: "8080" 或 ""
     ```

6. `location.pathname`
   - **描述**：返回当前 URL 的路径部分（从主机名之后开始，包含 `/`），例如 `/path/to/page`。
   - **用法**：

     ```
     console.log(location.pathname);  // 输出: "/path/to/page"
     ```

7. `location.search`

- **描述**：返回 URL 中的查询字符串部分，包括 `?`，例如 `?id=123&name=John`。
- **用法**：

```
console.log(location.search);  // 输出: "?id=123&name=John"
```

8. `location.hash`
   - **描述**：返回 URL 的锚部分（即 `#` 之后的部分），例如 `#section1`。
   - **用法**：

```
console.log(location.hash);  // 输出: "#section1"
location.hash = '#newSection';  // 改变 URL 锚
```

9. `location.origin`
   - **描述**：返回当前页面的来源，包括协议、主机名和端口号，例如 `https://www.example.com:8080`。
   - **用法**：

```
console.log(location.origin);  // 输出: "https://www.example.com:8080"
```

二、常见方法

1. `location.assign(url)`
   - **描述**：加载指定的 URL，相当于设置 `location.href`，并将当前页面加入到浏览器的历史记录中。
   - **用法**：

```
location.assign('https://www.example.com');  // 导航到新的 URL
```

2. `location.reload(force)`
   - **描述**：重新加载当前页面。可选的 `force` 参数为 `true` 时，强制从服务器重新加载页面（忽略缓存）。
   - **用法**：

```
location.reload();  // 重新加载当前页面
location.reload(true);  // 强制从服务器重新加载页面
```

3. `location.replace(url)`
   - **描述**：用新的 URL 替换当前页面，但不保留历史记录，因此不能通过浏览器的“后退”按钮返回之前的页面。
   - **用法**：

```
location.replace('https://www.example.com');  // 替换当前页面
```

4. `location.toString()`
   - **描述**：返回当前 URL 的字符串形式。通常等同于 `location.href`。

- **用法**：

```
console.log(location.toString());  // 输出当前 URL
```

示例

假设当前页面的 URL 是 `https://www.example.com:8080/path/to/page?id=123&name=John#section1`，我们可以获取以下信息：

```
console.log(location.href);       // 输出: "https://www.example.com:8080/path/to/page?id=123&name=John#section1"
console.log(location.protocol);   // 输出: "https:"
console.log(location.host);       // 输出: "www.example.com:8080"
console.log(location.hostname);   // 输出: "www.example.com"
console.log(location.port);       // 输出: "8080"
console.log(location.pathname);   // 输出: "/path/to/page"
console.log(location.search);     // 输出: "?id=123&name=John"
console.log(location.hash);       // 输出: "#section1"
console.log(location.origin);     // 输出: "https://www.example.com:8080"
```

通过 `location` 对象，开发者可以轻松操作和获取当前页面的 URL 信息，并可以通过相关方法来控制页面的导航、刷新和重定向。

三、URI和URL

> URI（统一资源标识符，**Uniform Resource Identifier**）和 URL（统一资源定位符，**Uniform Resource Locator**）是网络资源标识和访问的重要概念。它们常常会混用，但实际上存在一些细微的区别。

1. **URI 和 URL 的定义**
   - **URI（Uniform Resource Identifier，统一资源标识符）**：
     - URI 是一个通用的概念，指代用来唯一标识资源的字符串。URI 可以表示资源的名字、位置，甚至其他描述。它包括两类：URL 和 URN（统一资源名称）。
     - **URI 的作用**是标识某一个资源，而不一定要提供它的访问方式。

- **URL（Uniform Resource Locator，统一资源定位符）**：
  - URL 是 URI 的一个子集，除了标识资源外，它还具体定义了访问该资源的方式（比如协议、主机名、端口号等）。URL 主要是用来“定位”资源。
  - **URL 的作用**是标识资源，并且提供访问资源的方法。

2. **区别与联系**
   - **URI 是一个广义的概念**，它可以表示任何资源的标识，包括但不限于网络资源。URI 包含 URL 和 URN，两者是 URI 的子集。
   - **URL 是 URI 的子集**，它不仅标识了资源，还提供了获取该资源的方式（协议 + 位置）。
   - **URN（Uniform Resource Name，统一资源名称）**是 URI 的另一子集，它只用来标识资源而不涉及具体的访问方法。URN 主要用于给资源命名，而不涉及资源的物理位置。

3. **URL 和 URI 的关系图**

```
URI
 ├── URL （标识 + 访问方式）
 └── URN （只标识资源）
```

4. **示例**
   - **URL 示例**:
     - `https://www.example.com/index.html`
       - 这是一个 URL，因为它不仅标识了资源 `/index.html`，还包括了如何访问它：通过 `https` 协议从 `www.example.com` 主机获取资源。
   - **URN 示例**:
     - `urn:isbn:0451450523`
       - 这是一个 URN，因为它通过 ISBN 标识了一本书，但并没有说明如何通过网络访问这本书。

- **URI 示例**（包括 URL 和 URN）：
  - `https://www.example.com/index.html`（URL 是 URI 的一种）
  - `urn:isbn:0451450523`（URN 也是 URI）

5. **URI 和 URL 的技术解析**
   - **URI 结构**:
     一个完整的 URI 通常由以下部分组成：

     ```
     [scheme]://[userinfo]@[host]:[port]/[path]?[query]#[fragment]
     ```

     - **scheme**: 协议，比如 `http`、`https`、`ftp`。
     - **userinfo**: 可选，通常用于用户名和密码认证，比如 `user:pass`。
     - **host**: 主机名或 IP 地址，比如 `www.example.com` 或 `192.168.1.1`。
     - **port**: 可选，网络端口号，比如 `:80`、`:443`。
     - **path**: 资源的路径，比如 `/index.html`。
     - **query**: 可选，查询参数，比如 `?id=123&name=John`。
     - **fragment**: 可选，文档中的片段标识符，比如 `#section1`。
   - **URL 结构**:
     URL 通常会有完整的 scheme 和 host 信息，用于定义资源位置和访问方式。例如：

     ```
     https://www.example.com/path/to/resource?id=123#fragment
     ```

6. **总结**
   - **URI** 是统一资源标识符，是一个更广泛的概念，用于标识资源。所有的 URL 和 URN 都是 URI。
   - **URL** 是 URI 的一个子集，它不仅标识了资源，还提供了获取该资源的方式（比如使用 `http` 或 `ftp` 进行访问）。

因此，所有的 URL 都是 URI，但并非所有的 URI 都是 URL。

四、URLSearchParams

> `URLSearchParams` 是用于操作 URL 查询参数的接口，帮助我们轻松解析、添加、删除和修改 URL 中 `?` 后的查询字符串（键值对）。

简单示例：

给定 URL：`https://example.com/?name=John&age=30`，查询字符串为 `?name=John&age=30`，使用 `URLSearchParams` 可以操作这个部分。

常用 API：

1. `get(name)`：获取指定键的值。

```
const params = new URLSearchParams('name=John&age=30');
console.log(params.get('name'));  // 输出 'John'
```

2. `set(name, value)`：设置键的值（会覆盖已有值）。

```
params.set('name', 'Jane');
console.log(params.toString());  // 输出 'name=Jane&age=30'
```

3. `append(name, value)`：追加键值对（不覆盖原有值）。

```
params.append('color', 'blue');
console.log(params.toString());  // 输出 'name=Jane&age=30&color=blue'
```

4. `has(name)`：检查是否存在某个键。

```
console.log(params.has('age'));  // 输出 true
```

5. `delete(name)`：删除指定键及其值。

```
params.delete('age');
console.log(params.toString());  // 输出 'name=Jane'
```

6. `toString()`：将参数序列化为字符串。

```
console.log(params.toString());  // 输出 'name=Jane'
```

3. history对象

> `history` 对象是浏览器提供的接口，用于操作浏览器的历史记录。它包含了一系列属性和方法，可以让你通过编程的方式前进、后退或者跳转到指定的页面。

一、常见属性：

1. `length`
   - **描述**：返回当前浏览器会话中的历史记录数量。
   - **示例**：

     ```
     console.log(history.length);  // 输出历史记录的数量
     ```

2. `state`
   - **描述**：返回当前历史记录条目的状态对象。
   - **示例**：

     ```
     console.log(history.state);  // 输出当前的状态对象
     ```

二、常见方法：

1. `back()`
   - **描述**：让浏览器回退到上一个历史记录条目，等效于用户点击了浏览器的后退按钮。
   - **示例**：

     ```
     history.back();  // 返回上一页
     ```

2. `forward()`
   - **描述**：让浏览器前进到下一个历史记录条目，等效于用户点击了浏览器的前进按钮。
   - **示例**：

     ```
     history.forward();  // 前进到下一页
     ```

3. `go(n)`
   - **描述**：跳转到浏览器历史记录中的某个特定页面。参数 `n` 表示跳转的相对位置：
     - `n = -1` 表示后退一页。
     - `n = 1` 表示前进一页。
   - **示例**：

     ```
     history.go(-1);  // 后退一页
     history.go(1);   // 前进一页
     history.go(0);   // 重新加载当前页
     ```

4. `pushState(state, title, url)`
   - **描述**：将一个新的状态添加到历史记录栈中，不会触发页面刷新。可以用来实现单页应用程序的前端路由。
   - **参数**：
     - `state`：与新历史记录条目相关联的状态对象。
     - `title`：标题（目前大部分浏览器忽略该参数）。

- `url`：新的 URL。

- **示例**：

```
history.pushState({ page: 1 }, 'Title', '/page1');
```

5. `replaceState(state, title, url)`
   - **描述**：修改当前历史记录条目的状态，不会添加新的历史记录条目，也不会触发页面刷新。
   - **参数**：
     - `state`：新的状态对象。
     - `title`：标题。
     - `url`：新的 URL。
   - **示例**：

```
history.replaceState({ page: 2 }, 'Title', '/page2');
```

三、总结：

- `history.length`：获取历史记录条目数。
- `history.state`：获取当前条目的状态对象。
- `history.back()`：返回上一页。
- `history.forward()`：前进到下一页。
- `history.go(n)`：跳转到相对位置的页面。
- `history.pushState()`：添加新的历史记录条目。
- `history.replaceState()`：替换当前的历史记录条目。

这些方法和属性是前端开发中操作浏览器历史记录的常见工具，尤其在开发单页应用（SPA）时非常有用。

4. navigator对象

> `navigator` 对象表示用户代理（通常是浏览器）的状态和身份，并提供与浏览器环境相关的信息。通过 `navigator` 对象，开发者可以获取设备信息、检测网络状态、访问用户的地理位置等。

属性/方法	说　明
locks	返回暴露 Web Locks API 的 LockManager 对象
mediaCapabilities	返回暴露 Media Capabilities API 的 MediaCapabilities 对象
mediaDevices	返回可用的媒体设备
maxTouchPoints	返回设备触摸屏支持的最大触点数
mimeTypes	返回浏览器中注册的 MIME 类型数组
onLine	返回布尔值，表示浏览器是否联网
oscpu	返回浏览器运行设备的操作系统和（或）CPU
permissions	返回暴露 Permissions API 的 Permissions 对象
platform	返回浏览器运行的系统平台
plugins	返回浏览器安装的插件数组。在 IE 中，这个数组包含页面中所有<embed>元素
product	返回产品名称（通常是"Gecko"）
productSub	返回产品的额外信息（通常是 Gecko 的版本）
registerProtocolHandler()	将一个网站注册为特定协议的处理程序
requestMediaKeySystemAccess()	返回一个期约，解决为 MediaKeySystemAccess 对象
sendBeacon()	异步传输一些小数据
serviceWorker	返回用来与 ServiceWorker 实例交互的 ServiceWorkerContainer
share()	返回当前平台的原生共享机制
storage	返回暴露 Storage API 的 StorageManager 对象
userAgent	返回浏览器的用户代理字符串
vendor	返回浏览器的厂商名称
vendorSub	返回浏览器厂商的更多信息
vibrate()	触发设备振动
webdriver	返回浏览器当前是否被自动化程序控制

一、常见属性：

1. `navigator.userAgent`
   - **描述**：返回浏览器的用户代理字符串，提供关于浏览器和操作系统的相关信息。
   - **示例**：

     ```
     console.log(navigator.userAgent);  // 输出用户代理字符串，如 "Mozilla/5.0..."
     ```

2. `navigator.language`
   - **描述**：返回浏览器的默认语言（例如 "en-US" 表示美式英语）。
   - **示例**：

     ```
     console.log(navigator.language);  // 输出当前语言，如 "en-US"
     ```

3. `navigator.platform`
   - **描述**：返回浏览器正在运行的系统平台（例如 "Win32" 表示 Windows 操作系统）。
   - **示例**：

     ```
     console.log(navigator.platform);  // 输出系统平台，如 "Win32"
     ```

4. `navigator.onLine`
   - **描述**：返回一个布尔值，表示当前设备是否连接到网络。
   - **示例**：

```
console.log(navigator.onLine);  // 输出 true 或 false
```

5. `navigator.cookieEnabled`
   - **描述**：返回一个布尔值，表示浏览器是否启用了 Cookie。
   - **示例**：

```
console.log(navigator.cookieEnabled);  // 输出 true 或 false
```

6. `navigator.geolocation`
   - **描述**：返回 `Geolocation` 对象，用于获取设备的地理位置信息。
   - **示例**：

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(function(position) {
    console.log(position.coords.latitude, position.coords.longitude);
  });
}
```

二、常见方法：

1. `navigator.geolocation.getCurrentPosition(successCallback, errorCallback)`
   - **描述**：获取设备的当前地理位置，返回纬度和经度信息。
   - **参数**：
     - `successCallback`：成功时调用的回调函数。
     - `errorCallback`：失败时调用的回调函数。
   - **示例**：

```
navigator.geolocation.getCurrentPosition(function(position) {
  console.log('Latitude: ' + position.coords.latitude);
  console.log('Longitude: ' + position.coords.longitude);
}, function(error) {
  console.log('Error occurred: ' + error.message);
});
```

2. `navigator.vibrate(pattern)`
   - **描述**：让设备振动。常用于移动设备。
   - **参数**：接受一个数组或整数，表示振动的模式和时间（以毫秒为单位）。
   - **示例**：

```
navigator.vibrate(1000);  // 设备振动1秒
navigator.vibrate([200, 100, 200]);  // 振动200ms，停100ms，再振动200ms
```

3. `navigator.sendBeacon(url, data)`
   - **描述**：用于向服务器异步发送数据，适用于发送小量数据且不需要立即处理响应的场景。
   - **参数**：
     - `url`：要发送数据的服务器地址。

        - `data`：要发送的数据，可以是字符串、Blob 或 ArrayBuffer。
    - **示例**：

```
navigator.sendBeacon('/track', JSON.stringify({ event: 'pageUnload' }));
```

4. `navigator.clipboard.writeText(text)`
    - **描述**：将文本写入系统剪贴板。
    - **参数**：`text` 是要复制的文本内容。
    - **示例**：

```
navigator.clipboard.writeText('Hello World').then(function() {
  console.log('Text copied to clipboard');
});
```

5. `navigator.clipboard.readText()`
    - **描述**：从系统剪贴板中读取文本。
    - **示例**：

```
navigator.clipboard.readText().then(function(text) {
  console.log('Clipboard content: ', text);
});
```

总结：

- **属性**：`userAgent`、`language`、`platform`、`onLine`、`cookieEnabled`、`geolocation`。
- **方法**：`geolocation.getCurrentPosition()`、`vibrate()`、`sendBeacon()`、`clipboard.writeText()`、`clipboard.readText()`。

`navigator` 对象为开发者提供了与用户设备和浏览器相关的有用信息和功能，在网页开发中广泛用于设备检测、网络状态管理和位置服务等场景。

5. screen对象

> `screen` 对象用于提供有关用户屏幕的信息，通常用于优化网页的布局和设计，以适应不同的屏幕大小。

一、常见属性：

1. `screen.width`
    - **描述**：返回屏幕的宽度（以像素为单位）。
    - **示例**：

```
console.log(screen.width);  // 输出屏幕宽度
```

2. `screen.height`
    - **描述**：返回屏幕的高度（以像素为单位）。
    - **示例**：

```
console.log(screen.height);  // 输出屏幕高度
```

3. `screen.availWidth`
   - **描述**：返回屏幕可用宽度，排除了操作系统任务栏或其他界面占用的部分。
   - **示例**：

```
console.log(screen.availWidth);  // 输出可用的屏幕宽度
```

4. `screen.availHeight`
   - **描述**：返回屏幕可用高度，排除了操作系统任务栏或其他界面占用的部分。
   - **示例**：

```
console.log(screen.availHeight);  // 输出可用的屏幕高度
```

5. `screen.colorDepth`
   - **描述**：返回屏幕显示的颜色深度（单位：位），即屏幕每像素使用的位数，通常是 24（即 16,777,216 色）。
   - **示例**：

```
console.log(screen.colorDepth);  // 输出颜色深度，例如24
```

6. `screen.pixelDepth`
   - **描述**：返回屏幕的像素深度，通常与 `colorDepth` 相同。
   - **示例**：

```
console.log(screen.pixelDepth);  // 输出像素深度
```

二、常见方法：

`screen` 对象没有常用的内置方法，主要是通过其属性来获取与屏幕相关的信息。

三、实用场景示例：

1. **适配布局**
   - 根据屏幕宽高，动态调整网页布局：

```
if (screen.width < 1024) {
  console.log("小屏设备，调整布局");
} else {
  console.log("大屏设备");
}
```

2. **判断可用屏幕空间**
   - 检查是否有足够的可用屏幕空间：

```
if (screen.availHeight < 800) {
  console.log("可用屏幕空间较小，可能被任务栏占用");
}
```

3. **颜色深度优化**
    - 根据屏幕的颜色深度，调整网页显示质量：

```
if (screen.colorDepth < 24) {
  console.log("颜色深度较低，使用低质量图片");
}
```

总结：

- **属性**：
    - `width` / `height`：屏幕的总宽度和高度。
    - `availWidth` / `availHeight`：可用屏幕的宽度和高度。
    - `colorDepth` / `pixelDepth`：屏幕的颜色深度和像素深度。
- **方法**：`screen` 对象没有常用方法，主要用于获取屏幕信息。

这些属性在适应不同屏幕设备的布局设计中十分重要，尤其是在响应式设计和跨设备开发时会经常用到。

13. JSON

> **JSON**（JavaScript Object Notation）是一种轻量级的数据交换格式，易于人阅读和编写，同时也便于机器解析和生成。尽管 JSON 的名字带有 JavaScript，但它是与语言无关的格式，现已成为数据交换的标准格式之一，广泛应用于各种编程语言中。

一、JSON 的用途：

1. **数据交换**：客户端与服务器之间的数据传输，如 RESTful API、WebSocket、AJAX 请求等，常使用 JSON 格式。
2. **配置文件**：许多应用程序使用 JSON 作为配置文件格式，例如 Node.js 中的 `package.json` 文件。
3. **数据存储**：某些数据库（如 NoSQL 数据库 MongoDB）采用 JSON 或类似的 BSON 格式来存储数据。
4. **跨平台通信**：JSON 是前后端分离架构中常用的通信格式，在不同语言之间进行数据传递时，JSON 是非常便捷的选择。

二、JSON 的语法：

- **对象**：由键值对组成，键是字符串，值可以是字符串、数值、布尔值、数组、对象或 `null`，键值对之间用逗号分隔，整体用 `{}` 包裹。

```
{
  "name": "John",
  "age": 30,
  "isStudent": false,
  "courses": ["Math", "Science"],
  "address": {
    "city": "New York",
    "zip": "10001"
  }
}
```

- **数组**：JSON 中的数组是值的有序集合，元素之间用逗号分隔，整体用 `[]` 包裹。

```
["apple", "banana", "cherry"]
```

- **基本数据类型**：
  - **字符串**：必须用双引号包围。

```
"example"
```

  - **数值**：整数或浮点数，直接书写即可。

```
42
```

  - **布尔值**：`true` 或 `false`。

```
true
```

  - `null`：表示空值。

```
null
```

三、JSON 常用 API：

在 JavaScript 中，处理 JSON 数据的主要 API 是 `JSON.parse()` 和 `JSON.stringify()`。

1. `JSON.parse()`
   - **作用**：将 JSON 格式的字符串解析为 JavaScript 对象。
   - **语法**：

```
JSON.parse(text);
```

   - **示例**：

```
const jsonString = '{"name":"John", "age":30}';
const obj = JSON.parse(jsonString);
console.log(obj.name);  // 输出 "John"
```

2. `JSON.stringify()`
   - **作用**：将 JavaScript 对象转换为 JSON 格式的字符串。
   - **语法**：

```
JSON.stringify(value[, replacer[, space]]);
```

   - **参数**：
     - `value`：要转换为 JSON 字符串的 JavaScript 值（对象、数组等）。
     - `replacer`：可选，函数或数组，用于筛选或转换对象中的值。
     - `space`：可选，用于格式化输出（如添加缩进），通常是数值或字符串。

- **示例**：

```
const obj = { name: "John", age: 30 };
const jsonString = JSON.stringify(obj);
console.log(jsonString);  // 输出 '{"name":"John","age":30}'
```

3. `JSON.stringify()` **使用** `space` **参数进行格式化**：
   - **示例**：

```
const obj = { name: "John", age: 30 };
const jsonString = JSON.stringify(obj, null, 2);  // 使用 2 空格缩进
console.log(jsonString);
/* 输出:
{
  "name": "John",
  "age": 30
}
*/
```

4. **处理 JSON 数组**：
   - **解析 JSON 数组**：

```
const jsonString = '["apple", "banana", "cherry"]';
const arr = JSON.parse(jsonString);
console.log(arr[1]);  // 输出 "banana"
```

   - **字符串化 JavaScript 数组**：

```
const arr = ["apple", "banana", "cherry"];
const jsonString = JSON.stringify(arr);
console.log(jsonString);  // 输出 '["apple","banana","cherry"]'
```

总结：

- **JSON** 是一种轻量级的数据交换格式，适合传递数据和配置。
- **语法**：包括对象 `{}`、数组 `[]`、字符串、数值、布尔值和 `null`。
- **常用 API**：
  - `JSON.parse()`：将 JSON 字符串解析为 JavaScript 对象。
  - `JSON.stringify()`：将 JavaScript 对象转换为 JSON 字符串。

14、浏览器中的storage

> 浏览器的 `Storage` 是用于在客户端（浏览器）中存储数据的机制，它允许 web 应用程序在用户的设备上存储数据，以便在页面刷新、关闭或重新打开时，数据仍然能够被持久化或临时存储。`Storage` 包含两个主要的 API：`localStorage` 和 `sessionStorage`，它们在功能上有些不同。

1. `localStorage`

`localStorage` 用于存储持久化的数据，数据没有过期时间，除非手动删除，否则会一直保存在用户的浏览器中，即使浏览器关闭或计算机重启，数据仍然存在。

特点：

- 数据存储在浏览器中，持久化保存。
- 不限制页面或会话的生命周期，可以跨会话、跨标签页读取。
- 数据容量一般限制在 5MB 左右，不同浏览器略有不同。
- 只能存储字符串类型的数据，如果是对象类型需要通过 JSON 序列化。

常用 API：

```
// 设置数据
localStorage.setItem('key', 'value');

// 获取数据
const value = localStorage.getItem('key');

// 删除数据
localStorage.removeItem('key');

// 清空所有数据
localStorage.clear();
```

2. `sessionStorage`

`sessionStorage` 用于在一个会话（session）内存储数据。当浏览器标签页关闭时，数据会被清除。它的数据仅在页面会话期间有效，不会在页面之间共享，主要用于在当前标签页中临时保存数据。

特点：

- 数据在会话结束（关闭标签页或浏览器）时清除。
- 只在当前页面或标签页中有效，不会在不同的标签页之间共享数据。
- 数据容量一般与 `localStorage` 类似，约 5MB。
- 只能存储字符串类型的数据，同样需要序列化对象。

常用 API：

```
// 设置数据
sessionStorage.setItem('key', 'value');

// 获取数据
const value = sessionStorage.getItem('key');

// 删除数据
sessionStorage.removeItem('key');

// 清空所有数据
sessionStorage.clear();
```

区别总结：

- `localStorage`：数据持久化，除非手动删除，否则不会过期。
- `sessionStorage`：数据只在当前会话中有效，会话结束后自动删除。

这两种 `Storage` 方法都属于 `HTML5` 标准的一部分，用于在客户端存储数据，避免频繁的网络请求，优化性能，增强用户体验。