JavaScript常见内置类

原始类型的包装类

- **JavaScript的原始类型并非对象类型，所以从理论上来说，它们是没有办法获取属性或者调用方法的**
- 但是，在开发中会看到，我们会经常这样操作：

```
var message = "Hello World"
var words = message.split(" ")
var length = message.length

var num = 2.1314
num = num.toFiexd(2)
```

- **那么，为什么会出现这样奇怪的现象呢？（悖论）**
  - 原始类型是**简单的值**，默认**并不能调用属性和方法**
  - 这是因为JavaScript为了可以**使其可以获取属性和调用方法，对其封装了对应的包装类型**
- **常见的包装类型有：String、Number、Boolean、Symbol、BigInt类型**

包装类型的使用过程

- **默认情况，当我们调用一个原始类型的属性或者方法时，会进行如下操作：**
  - 根据**原始值，创建一个原始类型对应的包装类型对象**
  - **调用对应的属性或者方法，返回一个新的值**
  - **创建的包装类对象被销毁**
  - 通常JavaScript**引擎会进行很多的优化，它可以跳过创建包装类的过程在内部直接完成属性的获取或者方法的调用**
- **我们也可以自己来创建一个包装类的对象：**
  - name1是字面量（literal）的创建方式，name2是new创建对象的方式

```
let name1 = "XiaoYu"
let name2 =new String("XiaoYu")
console.log(typeof name1 === typeof name2)//false
```

- 注意事项：**null、undefined没有任何的方法，也没有对应的"对象包装类**

Number类的补充

- **前面我们已经学习了Number类型，它有一个对应的数字包装类型Number，我们来对它的方法做一些补充**
- **Number属性补充：**
  - Number.MAX_SAFE_INTEGER：JavaScript 中最大的安全整数 (2^53 - 1)
  - Number.MIN_SAFE_INTEGER：JavaScript 中最小的安全整数 -(2^53 - 1)

- **Number实例方法补充：**
  - **方法一：toString(base)**，将数字转成字符串，并且按照base进制进行转化

    ✓ base 的范围可以从 2 到 36，默认情况下是 10

    ✓ 注意：如果是直接对一个数字操作，需要使用..运算符

    ```
    123..toString()//"123"
    ```

  - **方法二：toFixed(digits)**，格式化一个数字，保留digits位的小数，会四舍五入

    digits的范围是0到20（包含）之间
- **Number类方法补充：**
  - **方法一：Number.parseInt(string[, radix])**，将字符串解析成整数，也有对应的全局方法parseInt
  - **方法二：Number. parseFloat(string)**，将字符串解析成浮点数，也有对应的全局方法parseFloat

```
//实现方法
let num = "123.456"
//实现方式1：(两个方法都默认不会四舍五入，如果需要四舍五入的话需要num.toFixed(0))
console.log(Number.parseInt(num))
//实现方式2：因为window对象上也有这个方法，所以可以这样实现
console.log(parseInt(num))

//验证，这里出现了不一致，在Node环境下会报错，在浏览器中打印显示true(正常)
console.log(window.parseInt === Number.parseInt)
```

- **更多Number的知识，可以查看MDN文档：**
  - https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Number
- 类方法和实例方法的区别?

> 1. 定义：
>
> - 实例方法：实例方法是定义在类的原型（prototype）上的方法。当创建一个类的实例时，这些方法将成为实例对象的成员方法。实例方法通常用于实例的行为操作，它们可以访问和修改实例的属性。
> - 类方法：类方法是定义在类本身上的方法，而不是在其原型上。类方法通常用于与类相关的操作，而与实例无关。类方法通常不会直接操作实例的属性。
>
> 2. 调用方式：
>
> - 实例方法：实例方法需要通过类的实例来调用。在实例方法中，关键字`this`引用的是调用该方法的实例对象。

```
class MyClass {
  constructor(name) {
    this.name = name;
  }

  sayHello() {
    console.log(`Hello, ${this.name}!`);
  }
}

let instance = new MyClass('XiaoYu');
instance.sayHello(); // 输出 'Hello, XiaoYu!'
```

- 类方法：类方法可以直接通过类本身调用，而不需要创建实例。在类方法中，关键字`this`引用的是类本身

```
class MyClass {
  static sayHello() {
    console.log('Hello, World!');
  }
}

MyClass.sayHello(); // 输出 'Hello, World!'
```

1. 用途：

- 实例方法：实例方法通常用于实现与特定实例相关的操作，如访问或修改实例的属性。
- 类方法：类方法通常用于实现与类相关的操作，如创建实例、处理类级别的数据等。类方法也可用作工具函数，提供一些与类相关的功能，但与特定实例无关。

总结：实例方法和类方法的主要区别在于它们的定义位置、调用方式和用途。实例方法是定义在类的原型上，通过实例对象调用，用于操作实例属性。类方法是定义在类本身上，直接通过类调用，用于执行与类相关的操作。

Math对象

- **在除了Number类可以对数字进行处理之外，JavaScript还提供了一个Math对象**
  - Math是一个**内置对象**（不是一个构造函数），它**拥有一些数学常数属性和数学函数方法**
- **Math常见的属性：**
  - Math.PI：圆周率，约等于 3.14159
- **Math常见的方法：**
  - Math.floor：向下舍入取整
  - Math.ceil：向上舍入取整
  - Math.round：四舍五入取整
  - Math.random：生成0~1的随机数（包含0，不包含1）
  - Math.pow(x, y)：返回x的y次幂

数字	Math.floor	Math.ceil	Math.round
3.1	3	4	3
3.6	3	4	4

String类的补充

String类的补充（一）- 基本使用

- **在开发中，我们经常需要对字符串进行各种各样的操作，String类提供给了我们对应的属性和方法**
- **String常见的属性：**
  - length：获取字符串的长度
- **操作一：访问字符串的字符**
  - 使用方法一：通过字符串的索引 **str[0]**
  - 使用方法二：通过**str.charAt(pos)**方法
  - 它们的区别是**索引的方式没有找到会返回undefined，而charAt没有找到会返回空字符串**
- **字符串的遍历**
  - 方式一：普通for循环
  - 方式二：for..of遍历

```
let name = "XiaoYu"
//方式1:
for (let i = 0; i < name.length ; i++){
  console.log(name[i])
}
//方式2:
for (let item of name){
  console.log(item)
}
```

String类的补充（二） - 修改字符串

- **字符串的不可变性：**
  - 字符串在定义后是**不可以修改**的，所以下面的操作是没有任何意义的

```
let name = "XiaoYu"

name[1] = "A"
console.log(name)//XiaoYu
```

- **所以，在我们改变很多字符串的操作中，都是生成了一个新的字符串**
  - 比如改变字符串大小的两个方法
  - **toLowerCase()**：将所有的字符转成小写

- **toUpperCase()**：将所有的字符转成大写

```
let name = "XiaoYu"

console.log(name.toLowerCase())//xiaoyu
console.log(name.toUpperCase())//XIAOYU
```

String类的补充（三） - 查找字符串

- **在开发中我们经常会在一个字符串中查找或者获取另外一个字符串，String提供了如下方法：**
- **方法一：查找字符串位置**

> 给定一个参数：要搜索的子字符串，搜索整个调用字符串，并返回指定子字符串第一次出现的索引。给定第二个参数：一个数字，该方法将返回指定子字符串在大于或等于指定数字的索引处的第一次出现

  - 从fromIndex开始，查找searchValue的索引
  - 如果没有找到，那么返回-1
  - 有一个相似的方法，叫lastIndexOf，从最后开始查找（用的较少）

```
let name = "XiaoYu"

console.log(name.indexOf("Y",4))//4
console.log(name.indexOf("X",4))//-1
```

- **方法二：是否包含字符串**
  - 从position位置开始查找searchString， 根据情况返回 true 或 false
  - 这是ES6新增的方法(用来判断包含关系)

```
let name = "XiaoYu"

if (name.includes("XiaoYu")){
  console.log("是的，name包含了XiaoYu")
}

//简写
name.includes("XiaoYu") && console.log("包含了")
```

String类的补充（四）- 开头和结尾

- **方法三：以xxx开头**
  - 从position位置开始，判断字符串是否以searchString开头
  - 这是ES6新增的方法，下面的方法也一样
- **方法四：以xxx结尾**
  - 在length长度内，判断字符串是否以searchString结尾

- **方法五：替换字符串**
  - 查找到对应的字符串，并且使用新的字符串进行替代

    这里也可以传入一个正则表达式来查找，也可以传入一个函数来替换

```
let name = "XiaoYu"
//startWith：是否以XXX开头
name.startsWith("Xiao") && console.log("是的，以这个开头")
//endsWith：确定字符串是否以指定字符串的字符结束，并根据需要返回true或false
name.endsWith("Yu") && console.log("是的，以这个结尾")

//replace()方法返回一个新字符串，其中包含被替换的模式的部分或全部匹配。模式可以是字符串或RegExp，替换可以是要为每次匹配调用的字符串或函数。如果pattern是字符串，则只替换第一次出现的字符串。
//原始字符串保持不变
//参数1：查找字符串，参数2：需要替换的字符串
let name2 = name.replace("Yu","Man")
console.log(name2)
```

String类的补充（五） - 获取子字符串

- **方法八：获取子字符串**

方法	选择方式	负值参数
slice(start,end)	**从 start 到 end（不含 end）**	**允许**
substring(start,end)	从 start 到 end（不含 end）	负值代表 0
substr(start,length)，有所限制	从 start 开始获取长为 length 的字符串	允许 start 为负数

- 开发中推荐使用**slice方法**(只需要掌握这种方法)
- slice()方法提取字符串的一部分，并将其作为新字符串返回，而不修改原始字符串

```
let num = "123456789"
console.log(num.slice(0,5))//12345
console.log(num.slice(-1))//9
console.log(num.substring(0,5))//12345
```

String类的补充（六） - 其他方法

- **方法六：拼接字符串**
  - `concat()` 方法将一个或多个字符串与原字符串连接合并，形成一个新的字符串并返回
  - 语法：str.concat(str2, [, ...strN])
  - 在MDN中他强烈建议使用赋值操作符 (en-US)（`+`，`+=`）代替 `concat` 方法
- **方法七：删除首尾空格**

- `trim()` 方法从字符串的两端清除空格，返回一个新的字符串，而不修改原始字符串。此上下文中的空格是指所有的空白字符（空格、tab、不换行空格等）以及所有行终止符字符（如 LF、CR 等）

```
const greeting = '   Hello world!   ';

console.log(greeting);
// Expected output: "   Hello world!   ";

console.log(greeting.trim());
// Expected output: "Hello world!";
```

- **方法九：字符串分割(重点)**
  - `split()` 方法使用指定的分隔符字符串将一个 `String` 对象分割成子字符串数组，以一个指定的分割字串来决定每个拆分的位置。
  - separator：以什么字符串进行分割，也可以是一个正则表达式
  - limit：限制返回片段的数量

```
let name = "Xiao-Yu"
let nameArr = name.split("-")
console.log(nameArr)//[ 'Xiao', 'Yu' ]
//变成数组后再将其拼接回字符串
let nameStr = nameArr.join("--")
console.log(nameStr)//Xiao--Yu
```

- **更多的字符串的补充内容，可以查看MDN的文档：**
  - https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String

认识数组（Array）

- **什么是数组（Array）呢?**
  - 对象允许**存储键值集合**，但是在**某些情况下使用键值对来访问并不方便**
  - 比如说**一系列的商品、用户、英雄**，包括**HTML元素**，我们**如何将它们存储在一起呢?**
  - 这个时候我们需要**一种有序的集合**，里面的元素是**按照某一个顺序来排列**的
  - 这个**有序的集合**，我们可以通过**索引**来获取到它
  - 这个结构就是数组（Array）
- **数组和对象都是一种保存多个数据的数据结构，在后续的数据结构中我们还会学习其他结构**
- **我们可以通过[]来创建一个数组**
  - 数组是一种**特殊的对象类型**

```
let Arr = ["1","3","5"]
```

数组的创建方式

- **创建一个数组的两种语法：**

```
let Arr1 = []
let Arr2 = new Array()
```

- **创建一个数组时，设置数组的长度（很少用）**

```
let Arr1 = new Array(5)
```

- **数组元素从 0 开始编号（索引index）**
  - 一些编程语言允许我们使用负数索引来实现这一点，例如 fruits[-1]
  - JavaScript**并不支持这种写法**
- 数组的基本操作：
  - 访问、修改、增加、删除数组中的元素

数组的基本操作

- **访问数组中的元素：**
  - 通过中括号[]访问
  - arr.at(i)：

    ✓ 如果 i >= 0，则与 arr[i] 完全相同

    ✓ 对于 i 为负数的情况，它则从数组的尾部向前数

    ```
    console.log(arr[0])
    console.log(arr.at(-1))
    ```

- **修改数组中的元素**

```
arr[0] = "小余"
```

- **删除和添加元素虽然也可以通过索引来直接操作，但是开发中很少这样操作**

数组的添加、删除方法（一）

- **在数组的尾端添加或删除元素：**
  - **push** 在末端添加元素
  - **pop** 从末端取出一个元素

```
let nameArr = ["小余","小满","coderwhy"]
nameArr.push("狗洛","康老师")
console.log(nameArr)//[ '小余', '小满', 'coderwhy', '狗洛', '康老师' ]
nameArr.pop()
console.log(nameArr)//[ '小余', '小满', 'coderwhy', '狗洛' ]
nameArr.unshift("首位")
console.log(nameArr)//[ '首位', '小余', '小满', 'coderwhy', '狗洛' ]
nameArr.shift()
console.log(nameArr)//[ '小余', '小满', 'coderwhy', '狗洛' ]
```

- **在数组的首端添加或删除元素**
  - **shift** 取出队列首端的一个元素，整个数组元素向前前移动
  - **unshift** 在首端添加元素，整个其他数组元素向后移动
- **push/pop 方法运行的比较快，而 shift/unshift 比较慢**

数组的添加、删除方法（二）

- **如果我们希望在中间某个位置添加或者删除元素应该如何操作呢?**
- **arr.splice 方法可以说是处理数组的利器，它可以做所有事情：添加，删除和替换元素**
- **arr.splice的语法结构如下：**

> - `start`（必需）：指定从哪个索引位置开始修改数组。如果`start`为负数，表示从数组末尾开始计算的索引位置。
> - `deleteCount`（可选）：指定要删除的元素数量。如果省略或者大于等于`array.length - start`，则删除从`start`位置开始的所有元素。如果`deleteCount`为0或负数，则不删除任何元素。
> - `item1, item2, ...`（可选）：要添加到数组中的元素，从`start`索引位置开始插入。如果不指定这些参数，则只删除数组元素，不添加。
>
> 返回值：一个包含被删除元素的新数组。如果未删除任何元素，则返回一个空数组。

```
array.splice(start[, deleteCount[, item1[, item2[, ...]]]]);
```

- 示例：

```
let arr = [1, 2, 3, 4, 5];

// 从索引1开始删除2个元素
let deleted = arr.splice(1, 2);
console.log(arr);      // 输出 [1, 4, 5]
console.log(deleted);  // 输出 [2, 3]

// 从索引2开始插入两个元素
arr.splice(2, 0, 6, 7);
console.log(arr); // 输出 [1, 4, 6, 7, 5]
```

```
// 从索引1开始，删除1个元素并插入两个新元素
arr.splice(1, 1, 8, 9);
console.log(arr); // 输出 [1, 8, 9, 6, 7, 5]
```

- **注意：这个方法会修改原数组**

length属性

- **length属性用于获取数组的长度：**
  - 当我们**修改数组**的时候，**length 属性会自动更新**
- **length 属性的另一个有意思的点是它是可写的。**
  - 如果我们**手动增加一个大于默认length的数值**，那么**会增加数组的长度**
  - 但是如果我们**减少它，数组就会被截断**

```
let nameArr = [1,2,3,4,5,6,7,8,9]
//扩展数组
nameArr.length = 12
console.log(nameArr)//[ 1, 2, 3, 4, 5, 6, 7, 8, 9, <3 empty items> ]
//截取数组
nameArr.length = 3
console.log(nameArr)//[ 1, 2, 3 ]
//清空数组
nameArr.length = 0
console.log(nameArr)//[]
```

- **所以，清空数组最简单的方法就是：arr.length = 0;**

数组的遍历

1. **普通for循环遍历**

```
for(let i = 0;i < num.length ; i++){
  console.log(num[i])
}
```

2. **for..in 遍历，获取到索引值**

```
for (let index in num){
  console.log(num[index])
}
```

3. **for..of 遍历，获取到每一个元素**

```
for (let item of num){
  console.log(item)
}
```

数组方法 – slice、cancat、 join

- `slice(start, end)`:
  - 作用：`slice`方法用于从数组中提取一段元素，创建一个新的数组。原数组不会受到影响。
  - 参数：
    - `start`：表示提取范围的起始索引，如果为负数，则表示从数组末尾开始计算。
    - `end`：表示提取范围的结束索引，但不包括该索引对应的元素。如果为负数，则表示从数组末尾开始计算。如果省略此参数，则提取范围一直到数组末尾。

```
const arr = [1, 2, 3, 4, 5];
const newArr = arr.slice(1, 4);
console.log(newArr); // 输出：[2, 3, 4]
```

- `concat(array1, array2, ...)`:
  - 作用：`concat`方法用于连接两个或多个数组，创建一个新的数组。原数组不会受到影响。
  - 参数：可以传递一个或多个数组作为参数，这些数组将按照参数顺序连接到一起。

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinedArr = arr1.concat(arr2);
console.log(combinedArr); // 输出：[1, 2, 3, 4, 5, 6]
```

- `join(separator)`:
  - 作用：`join`方法用于将数组中的所有元素连接成一个字符串。
  - 参数：`separator`是一个可选参数，表示元素之间的分隔符。如果省略此参数，则使用逗号`,`作为分隔符。

```
const arr = ['hello', 'world'];
const str = arr.join(' ');
console.log(str); // 输出："hello world"
```

数组方法 – 查找元素

- `arr.indexOf(element, fromIndex)`:
  - 作用：`indexOf`方法用于在数组中查找给定元素的第一个匹配项的索引。如果没有找到匹配项，则返回-1。
  - 参数：
    - `element`：要在数组中查找的元素。
    - `fromIndex`：可选参数，表示开始查找的索引位置。如果为负数，则从数组末尾开始计算。默认值为0。

```
const arr = [1, 2, 3, 2, 4];
const index = arr.indexOf(2);
console.log(index); // 输出：1
```

- `arr.includes(element, fromIndex)`：
  - 作用：`includes`方法用于检查数组中是否包含给定的元素。如果包含，则返回`true`，否则返回`false`。
  - 参数：
    - `element`：要在数组中查找的元素。
    - `fromIndex`：可选参数，表示开始查找的索引位置。如果为负数，则从数组末尾开始计算。默认值为0。

```
const arr = [1, 2, 3, 4, 5];
const hasElement = arr.includes(3);
console.log(hasElement); // 输出：true
```

- `find(callback)`：(高阶函数)
  - 作用：`find`方法用于在数组中查找第一个满足条件的元素。如果没有找到满足条件的元素，则返回`undefined`。
  - 参数：`callback`是一个函数，用于测试数组中的每个元素。当回调函数返回`true`时，查找停止，并返回当前元素。

```
const arr = [1, 2, 3, 4, 5];
const found = arr.find(element => element > 2);
console.log(found); // 输出：3
```

- `findIndex(callback)`：(高阶函数)
  - 作用：`findIndex`方法用于在数组中查找第一个满足条件的元素的索引。如果没有找到满足条件的元素，则返回-1。
  - 参数：`callback`是一个函数，用于测试数组中的每个元素。当回调函数返回`true`时，查找停止，并返回当前元素的索引。

```
const arr = [1, 2, 3, 4, 5];
const foundIndex = arr.findIndex(element => element > 2);
console.log(foundIndex); // 输出：2
```

Find与forEach内部实现

```
// 定义一个 find 高阶函数，它接受一个数组 arr 和一个回调函数 callback 作为参数
function find(arr, callback) {
  // 遍历数组中的每个元素
  for (let i = 0; i < arr.length; i++) {
    // 使用 callback 函数测试当前元素是否满足条件
    const isFound = callback(arr[i], i, arr);
```

```
    // 如果回调函数返回 true，则表示找到了满足条件的元素，返回该元素
    if (isFound) {
      return arr[i];
    }
  }

  // 如果遍历完整个数组都没有找到满足条件的元素，则返回 undefined
  return undefined;
}

// 示例：
const arr = [1, 2, 3, 4, 5];
const found = find(arr, element => element > 2);
console.log(found); // 输出：3
```

```
// 定义一个 forEach 函数，它接受一个数组 arr 和一个回调函数 callback 作为参数
function forEach(arr, callback) {
  // 遍历数组中的每个元素
  for (let i = 0; i < arr.length; i++) {
    // 使用 callback 函数处理当前元素，传递元素、索引和数组本身作为参数
    callback(arr[i], i, arr);
  }
  // forEach 函数没有返回值
}

// 示例：
const arr = [1, 2, 3, 4, 5];
forEach(arr, (element, index) => {
  console.log(`Element at index ${index} is ${element}`);
});
/*
输出：
Element at index 0 is 1
Element at index 1 is 2
Element at index 2 is 3
Element at index 3 is 4
Element at index 4 is 5
*/
```

> 放到原型链的做法讲解在JavaScript高级笔记中

```
// 在 Array 的原型链上添加 forEach 方法，接受一个回调函数 callback 作为参数
Array.prototype.forEachCustom = function(callback) {
  // 遍历数组中的每个元素
  for (let i = 0; i < this.length; i++) {
    // 使用 callback 函数处理当前元素，传递元素、索引和数组本身作为参数
    callback(this[i], i, this);
  }
  // forEach 函数没有返回值
}
```

```
// 示例：
const arr = [1, 2, 3, 4, 5];
arr.forEachCustom((element, index) => {
  console.log(`Element at index ${index} is ${element}`);
});
/*
输出：
Element at index 0 is 1
Element at index 1 is 2
Element at index 2 is 3
Element at index 3 is 4
Element at index 4 is 5
*/

//在这个重构后的代码中，我们将forEach函数添加到了Array.prototype上，并将其命名为forEachCustom。这样，所有的数组实例都可以直接调用forEachCustom方法。注意，这里我们使用this关键字来引用调用该方法的数组实例。
```

数组的排序 – sort/reverse

1. `sort()`：这个方法用于对数组元素进行排序。默认情况下，它将数组元素转换为字符串，然后根据字符串的Unicode码点值进行升序排列。这可能会导致数字数组的排序结果不如预期。

```
const arr = [10, 5, 8, 1, 7];
arr.sort();
console.log(arr); // 输出：[1, 10, 5, 7, 8]（按字符串Unicode码点值排序）
```

- 为了按照自定义的规则进行排序，你可以向`sort()`方法传递一个比较函数。这个比较函数接受两个参数，通常表示为`a`和`b`。如果`a`应该排在`b`之前，比较函数应返回负数；如果`a`应该排在`b`之后，比较函数应返回正数；如果`a`和`b`相等，则返回0。例如，以下代码将按升序排列数字数组：
  - 如果 `compareFn(a, b)` 大于 0，b 会被排列到 a 之前。
  - 如果 `compareFn(a, b)` 小于 0，那么 a 会被排列到 b 之前；

```
const arr = [10, 5, 8, 1, 7];
arr.sort((a, b) => a - b);
console.log(arr); // 输出：[1, 5, 7, 8, 10]（按数字升序排列）
```

2. `reverse()`：这个方法用于反转数组元素的顺序。它不需要任何参数，直接在原数组上进行操作。例如：

```
const arr = [1, 2, 3, 4, 5];
arr.reverse();
console.log(arr); // 输出：[5, 4, 3, 2, 1]（数组元素反转）
```

- 注意，`reverse()`方法会直接修改原数组，而不是创建一个新的反转后的数组。如果你需要保留原数组，请先使用`()`方法创建一个副本，然后对副本进行操作。例如：

```
const arr = [1, 2, 3, 4, 5];
const reversed = arr.slice().reverse();
console.log(reversed); // 输出：[5, 4, 3, 2, 1]（反转后的副本）
console.log(arr); // 输出：[1, 2, 3, 4, 5]（原数组保持不变）
```

数组的其他高阶方法

- **arr.forEach**
  - 遍历数组的每个元素，并对每个元素执行回调函数。它不返回任何值。回调函数接受三个参数：当前元素、当前索引和数组本身

```
const arr = [1, 2, 3, 4, 5];
arr.forEach((element, index) => {
  console.log(`Element at index ${index} is ${element}`);
});
```

- **arr.map**
  - 遍历数组的每个元素，并对每个元素执行回调函数。它返回一个新数组，其中的元素是原数组元素经过回调函数处理后的结果。回调函数接受三个参数：当前元素、当前索引和数组本身

```
const arr = [1, 2, 3, 4, 5];
const doubled = arr.map(element => element * 2);
console.log(doubled); // 输出：[2, 4, 6, 8, 10]
```

- **arr.filter**
  - 遍历数组的每个元素，并对每个元素执行回调函数。它返回一个新数组，包含满足回调函数条件（回调函数返回`true`）的元素。回调函数接受三个参数：当前元素、当前索引和数组本身

```
const arr = [1, 2, 3, 4, 5];
const even = arr.filter(element => element % 2 === 0);
console.log(even); // 输出：[2, 4]
```

- **arr.reduce**
  - 遍历数组的每个元素，并对每个元素执行回调函数。它返回一个累计值，通常用于对数组元素执行某种累加操作。回调函数接受四个参数：累计值、当前元素、当前索引和数组本身。还需提供一个初始值作为累计值的起始值

```
const arr = [1, 2, 3, 4, 5];
const sum = arr.reduce((accumulator, element) => accumulator + element, 0);
console.log(sum); // 输出：15
```

时间的表示方式

- **关于《时间》，有很多话题可以讨论：**
  - 比如物理学有**《时间简史：从大爆炸到黑洞》**，讲述的是关于**宇宙的起源、命运**

    - 比如文学上有**《记念刘和珍君》**：时间永是流驶，街市依旧太平
    - 比如音乐上有**《时间都去哪儿了》**：时间都去哪儿了，还没好好感受年轻就老了
- **我们先来了解一下时间表示的基本概念：**
- 最初，人们是通过观察**太阳的位置**来决定时间的，但是这种方式有一个最大的弊端就是**不同区域位置大家使用的时间**是不一致的
    - 相互之间没有办法通过一个统一的时间来沟通、交流
- 之后，人们开始制定的标准时间是**英国伦敦的皇家格林威治（Greenwich）天文台的标准时间**（刚好在本初子午线经过的地方），这个时间也称之为GMT（Greenwich Mean Time）
    - 其他时区根据标准时间来确定自己的时间，往东的时区（GMT+hh:mm），往西的时区（GMT+hh:mm）
- 但是，根据公转有一定的误差，也会造成GMT的时间会造成一定的误差，于是就提出了根据原子钟计算的标准时间UTC（Coordinated Universal Time）
- **目前GMT依然在使用，主要表示的是某个时区中的时间，而UTC是标准的时间**

时区对比图

创建Date对象

- **在JavaScript中我们使用Date来表示和处理时间**
    - **Date的构造函数**有如下用法：

1. 无参数构造函数：创建一个表示当前时间的`Date`对象

```
const currentDate = new Date();
console.log(currentDate);
```

2. 通过时间戳构造：创建一个表示从1970年1月1日00:00:00 UTC开始经过指定毫秒数的时间的`Date`对象

```
const timestamp = 1629204392000; // 时间戳，单位毫秒
const dateFromTimestamp = new Date(timestamp);
console.log(dateFromTimestamp);
```

3. 通过年、月构造：创建一个表示指定年份和月份的`Date`对象。年份和月份是必需的参数，其余参数可选

```
const year = 2021;
const month = 7; // 月份从0开始，0表示一月，1表示二月，依此类推
const dateFromYearMonth = new Date(year, month);
console.log(dateFromYearMonth);
```

4. 通过年、月、日构造：创建一个表示指定年份、月份和日期的`Date`对象

```
const year = 2021;
const month = 7;
const day = 18;
const dateFromYearMonthDay = new Date(year, month, day);
console.log(dateFromYearMonthDay);
```

5. 通过年、月、日、小时构造：创建一个表示指定年份、月份、日期和小时的`Date`对象

```
const year = 2021;
const month = 7;
const day = 18;
const hour = 14;
const dateFromYearMonthDayHour = new Date(year, month, day, hour);
console.log(dateFromYearMonthDayHour);
```

6. 通过年、月、日、小时、分钟构造：创建一个表示指定年份、月份、日期、小时和分钟的`Date`对象

```
const year = 2021;
const month = 7;
const day = 18;
const hour = 14;
const minute = 30;
const dateFromYearMonthDayHourMinute = new Date(year, month, day, hour,
minute);
console.log(dateFromYearMonthDayHourMinute);
```

7. 通过年、月、日、小时、分钟、秒构造：创建一个表示指定年份、月份、日期、小时、分钟和秒的`Date`对象

```
const year = 2021;
const month = 7;
const day = 18;
const hour = 14;
const minute = 30;
const second = 45;
const dateFromYearMonthDayHourMinuteSecond = new Date(year, month, day,
hour, minute, second);
console.log(dateFromYearMonthDayHourMinuteSecond);
```

8. 通过年、月、日、小时、分钟、秒、毫秒构造：创建一个表示指定年份、月份、日期、小时、分钟、秒和毫秒的`` `Date`对象

```
const year = 2021;
const month = 7;
const day = 18;
const hour = 14;
const minute = 30;
const second = 45;
const millisecond = 500;
const dateFromYearMonthDayHourMinuteSecondMillisecond = new Date(year, month, day, hour, minute, second, millisecond);
console.log(dateFromYearMonthDayHourMinuteSecondMillisecond);
```

9. 通过日期字符串构造：创建一个表示指定日期字符串的`Date`对象。字符串的格式可以是"IETF-compliant RFC 2822 timestamps"或者"version of ISO8601"

```
const dateString = '2021-08-18T14:30:45.500Z';
const dateFromString = new Date(dateString);
console.log(dateFromString);
```

dateString时间的表示方式

- **日期的表示方式有两种：RFC 2822 标准 或者 ISO 8601 标准**
- **默认打印的时间格式是RFC 2822标准的：**

```
new Date()
Mon Apr 24 2023 03:05:48 GMT+0800 (中国标准时间)
```

- **我们也可以将其转化成ISO 8601标准的:**

```
new Date().toISOString()//会受到VPN的影响
2023-04-23T19:11:58.988Z
```

字母	含义
YYYY	年份，0000~9999
MM	月份，01~12
DD	日，01~31
T	分隔日期和时间，没有特殊含义，可以省略
HH	小时，00~24
mm	分钟，00~59
ss	秒。00~59
.sss	毫秒
Z	时区

Date获取信息的方法

- **我们可以从Date对象中获取各种详细的信息：**
  - getFullYear()：获取年份（4 位数）
  - getMonth()：获取月份，从 0 到 11
  - getDate()：获取当月的具体日期，从 1 到 31（方法名字有点迷）
  - getHours()：获取小时
  - getMinutes()：获取分钟
  - getSeconds()：获取秒钟
  - getMilliseconds()：获取毫秒
- **获取某周中的星期几：**
  - getDay()：获取一周中的第几天，从 0（星期日）到 6（星期六）

```
// 创建一个表示当前时间的Date对象
const currentDate = new Date();

// 使用getFullYear()获取年份（4位数）
const currentYear = currentDate.getFullYear();
console.log('Year:', currentYear);

// 使用getMonth()获取月份，从0到11
const currentMonth = currentDate.getMonth();
console.log('Month (0-11):', currentMonth);

// 使用getDate()获取当月的具体日期，从1到31
const currentDay = currentDate.getDate();
console.log('Date:', currentDay);

// 使用getHours()获取小时
const currentHours = currentDate.getHours();
console.log('Hours:', currentHours);

// 使用getMinutes()获取分钟
const currentMinutes = currentDate.getMinutes();
console.log('Minutes:', currentMinutes);

// 使用getSeconds()获取秒钟
const currentSeconds = currentDate.getSeconds();
console.log('Seconds:', currentSeconds);

// 使用getMilliseconds()获取毫秒
const currentMilliseconds = currentDate.getMilliseconds();
console.log('Milliseconds:', currentMilliseconds);
```

Date设置信息的方法

- **Date也有对应的设置方法：**
  - setFullYear(year, [month], [date])
  - setMonth(month, [date])
  - setDate(date)
  - setHours(hour, [min], [sec], [ms])
  - setMinutes(min, [sec], [ms])
  - setSeconds(sec, [ms])
  - setMilliseconds(ms)
  - setTime(milliseconds)

```
// 创建一个表示当前时间的Date对象
const currentDate = new Date();
console.log('Original Date:', currentDate);

// 使用setFullYear(year, [month], [date])设置年份、月份和日期
currentDate.setFullYear(2025, 4, 15);
console.log('Date after setFullYear:', currentDate);

// 使用setMonth(month, [date])设置月份和日期
currentDate.setMonth(9, 10);
console.log('Date after setMonth:', currentDate);

// 使用setDate(date)设置日期
currentDate.setDate(20);
console.log('Date after setDate:', currentDate);

// 使用setHours(hour, [min], [sec], [ms])设置小时、分钟、秒钟和毫秒
currentDate.setHours(12, 30, 45, 500);
console.log('Date after setHours:', currentDate);

// 使用setMinutes(min, [sec], [ms])设置分钟、秒钟和毫秒
currentDate.setMinutes(45, 30, 250);
console.log('Date after setMinutes:', currentDate);

// 使用setSeconds(sec, [ms])设置秒钟和毫秒
currentDate.setSeconds(55, 750);
console.log('Date after setSeconds:', currentDate);

// 使用setMilliseconds(ms)设置毫秒
currentDate.setMilliseconds(999);
console.log('Date after setMilliseconds:', currentDate);

// 使用setTime(milliseconds)设置距离1970年1月1日00:00:00的毫秒数
currentDate.setTime(1609459200000);
console.log('Date after setTime:', currentDate);
```

Date获取Unix时间戳

- **Unix 时间戳：它是一个整数值，表示自1970年1月1日00:00:00 UTC以来的毫秒数**
- **在JavaScript中，我们有多种方法可以获取这个时间戳：**
  - 方式一：new Date().getTime()
  - 方式二：new Date().valueOf()
  - 方式三：+new Date()
  - 方式四：Date.now()
- **获取到Unix时间戳之后，我们可以利用它来测试代码的性能：**

> 其实就是获取当前的时间，进行相减得到中间的运行时间

```
let StartTime = Date.now()

for (let i = 0; i < 10000 ;i++){
  console.log(i)
}

let endTime = Date.now()

console.log(`一共运行了${endTime - StartTime}毫秒`)
```

Date.parse方法

> 补充一个内容，前面忘记写了，像这种方法都叫做类方法，因为这是来自Class中的方法(函数)
>
> - 类方法（Class method）是在类定义中的一个函数，用于操作类或实例对象的属性和行为。在JavaScript中，类方法通常通过关键字 `class` 定义的类中声明。类方法与实例方法相区别，实例方法仅在类的实例化对象上可用，而类方法可以在类本身或其实例上调用

```
class MyClass {
  static myClassMethod() {
    console.log('This is a class method.');
  }
}
```

> - 类方法可以通过类名直接调用，而无需创建实例对象：

```
MyClass.myClassMethod(); // 输出 "This is a class method."
```

> - 类方法通常用于执行与特定实例无关的操作，例如工具函数、计数器、创建实例等。在某些编程语言中，类方法也称为静态方法（static method）。请注意，尽管在其他编程语言中，类方法和静态方法可能有所不同，但在JavaScript中，它们是相同的概念

- **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**

```
let time1 = Date.parse("2023-05-04T08:08:08.666Z")
console.log(time1)//1683187688666
```