

## 第七章 JavaScript新特性

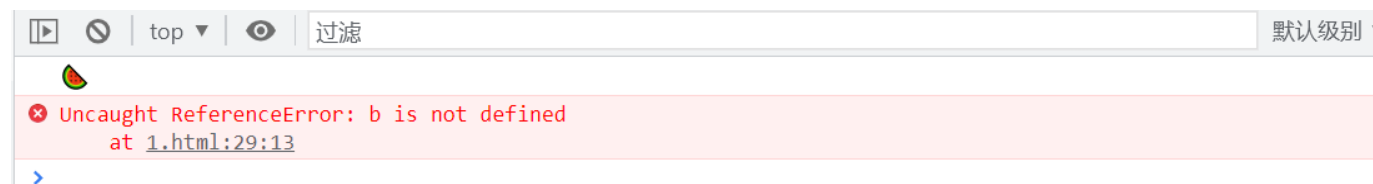
<http://t.csdnimg.cn/ctKlo>

### 7.1、ECMAScript6新特性

ES6 块级作用域 let 首先，什么是作用域？作用域简单讲就是声明一个变量，这个变量的有效范围，在 let 没来之前。js 只有 var 的全局作用域和 函数作用域，ES6 为 js 带来了块级作用域

```
<script>
{
  var a = "🍉";
  let b = "🍌";
}
console.log(a);
console.log(b);

</script>
```



可以看到，我们使用 var 关键字在块中定义了变量 a，其实全局可以访问到，所以说，var 声明的变量是全局的，但我们想让变量就在块中生效，出了块就访问不了了，这时就可以使用 ES6 新增的块级作用域关键字 let 来声明这个变量 a，当我再次访问，报错了，说 a is not defined，a 没有定义

#### 7.1.1、let 关键字

let 关键字用来声明变量，使用 let 声明的变量有几个特点：

- 不允许重复声明
- 块儿级作用域
- 不存在变量提升
- 不影响作用域链

注意：以后声明变量使用 let 就对了

案例演示：创建四个div，单机每一个div让其变色

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title></title>
  <style>
    .item {
      width: 100px;
      height: 50px;
      border: solid 1px rgb(12, 132, 132);
      float: left;
      margin-right: 10px;
    }
  </style>
</head>
<body>
  <div class="item"></div>
  <div class="item"></div>
  <div class="item"></div>
  <div class="item"></div>

  <!-- 在这里写JavaScript代码，因为JavaScript是由上到下执行的 -->
  <script>
    // 获取div元素对象
    let items = document.getElementsByClassName('item');

    // 遍历并绑定事件
    for (let i = 0; i < items.length; i++) {
      items[i].onclick = function () {
        // 以前的做法: this.style.background = "pink";
        items[i].style.background = "pink";
      };
    }
  </script>
</body>
</html>

```

### 7.1.2、const 关键字

const 关键字用来声明常量，const 声明有以下特点：

- 不允许重复声明
- 块级级作用域
- 声明必须赋初始值
- 值不允许修改

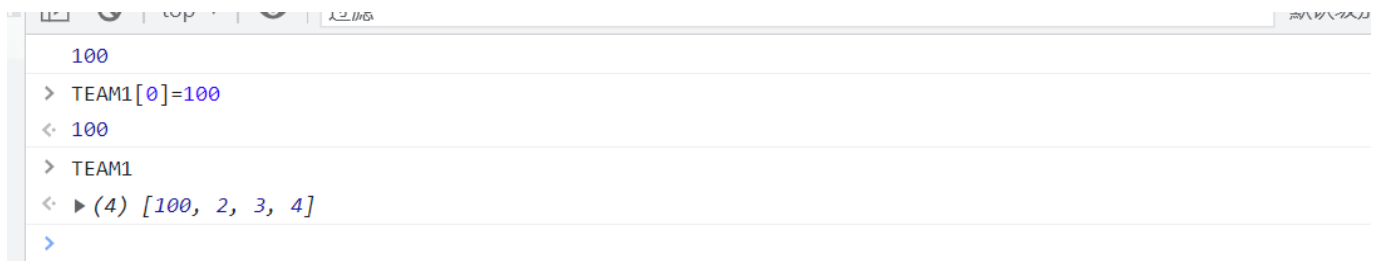
- 标识符一般为大写

注意：声明对象类型使用 `const`，非对象类型声明选择 `let`

```
<script>
  // 声明常量
  const MAX = 100;
  console.log(MAX);

  // 对于数组和对象的元素修改，不算做对常量的修改，不会报错
  const TEAM1 = [1, 2, 3, 4];
  const TEAM2 = [1, 2, 3, 4];
  // 但是不能修改地址指向
  // TEAM2 = TEAM1;

</script>
```



### 7.1.3、变量的解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构赋值。

注意：频繁使用对象方法、数组元素，就可以使用解构赋值形式

#### 数组的解构赋值：

```
<script>
  //数组的解构赋值
  const arr = ["张学友", "刘德华", "黎明", "郭富城"];
  let [zhang, liu, li, guo] = arr;
  console.log(zhang);
  console.log(liu);
  console.log(li);
  console.log(guo);
</script>
```

top	过滤	默认显示
张学友		
刘德华		
黎明		
郭富城		
>		

## 简单对象的解构赋值:

```
<script>
  //对象的解构赋值
  const lin = {
    name: "林志颖",
    tags: ["车手", "歌手", "小旋风", "演员"]
  };
  let {name, tags} = lin;
  console.log(name);
  console.log(tags);
</script>
```

林志颖
▶ Array(4)
> tags
◀ ▶ (4) ['车手', '歌手', '小旋风', '演员']
>

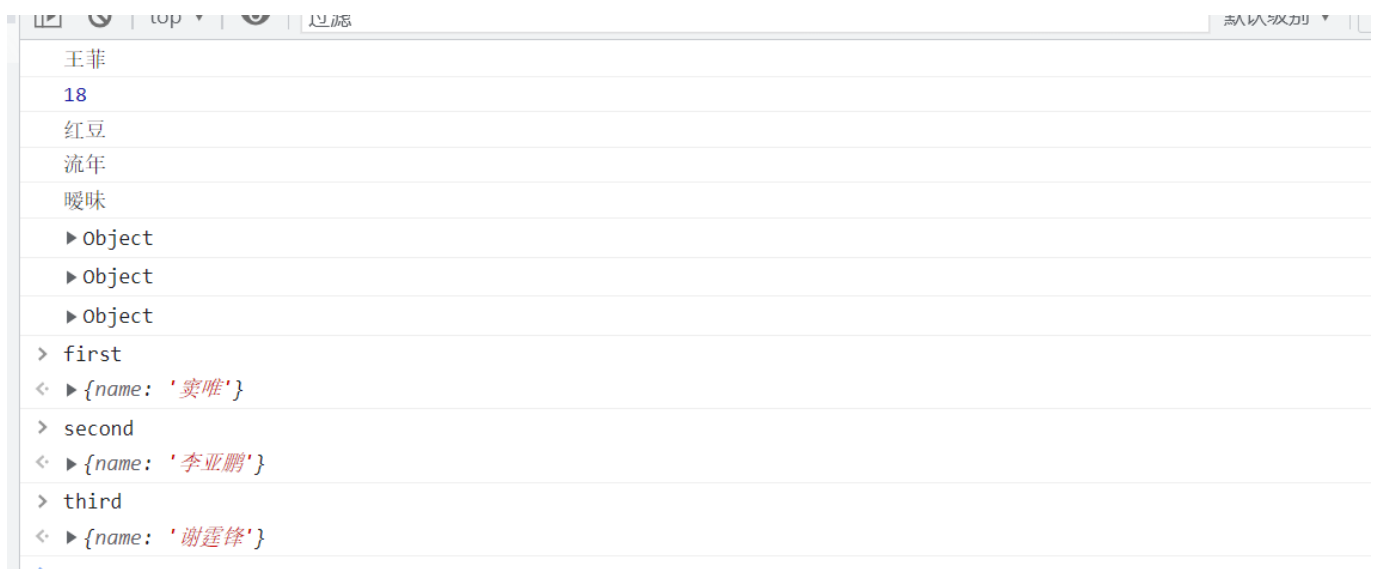
## 复杂对象的解构赋值:

```

<script>
  //复杂对象的解构赋值
  let wangfei = {
    name: "王菲",
    age: 18,
    songs: ["红豆", "流年", "暧昧"],
    history: [
      {name: "窦唯"},
      {name: "李亚鹏"},
      {name: "谢霆锋"}
    ]
  };
  let {name, age, songs: [one, two, three],
    history: [first, second, third]} = wangfei;
  console.log(name);
  console.log(age);
  console.log(one);
  console.log(two);
  console.log(three);
  console.log(first);
  console.log(second);
  console.log(third);

</script>

```



#### 7.1.4、模板字符串

模板字符串（template string）是增强版的字符串，用反引号（```）标识，特点：

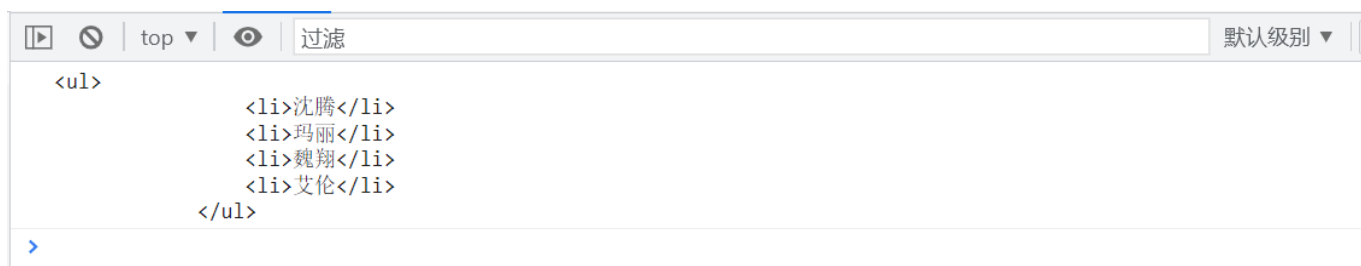
字符串中可以出现换行符 可以使用 `${xxx}` 形式输出变量

注意：当遇到字符串与变量拼接的情况使用模板字符串

## 字符串中可以出现换行符:

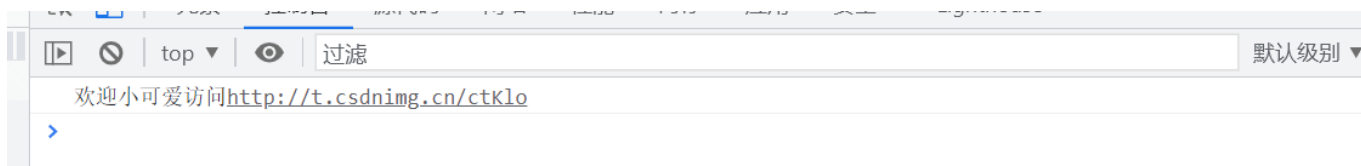
```
<script>
  //定义字符串
  let str = `


    <li>沈腾</li>
    <li>玛丽</li>
    <li>魏翔</li>
    <li>艾伦</li>
  </ul>`;
  console.log(str);
</script>
```



## 变量拼接:

```
<script>
  //变量拼接
  let name = '小可爱';
  let result = `欢迎${name}访问http://t.csdnimg.cn/ctKlo`;
  console.log(result);
</script>
```



### 7.1.5、简化对象写法

ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法，这样的书写更加简洁。

注意：对象简写形式简化了代码，所以以后用简写就对了

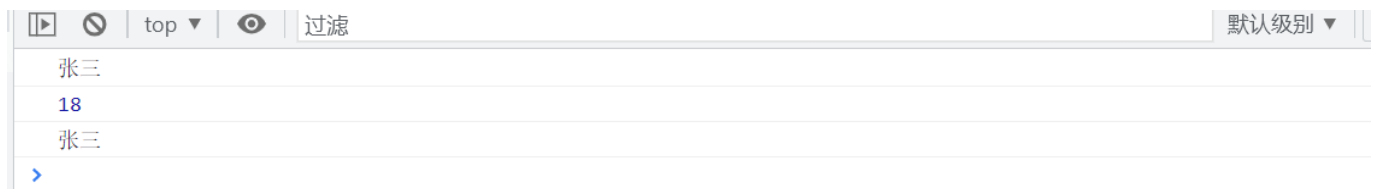
```

<script>
  let name = "张三";
  let age = 18;
  let speak = function () {
    console.log(this.name);
  };

  //属性和方法简写
  let person = {
    name,
    age,
    speak
  };

  console.log(person.name);
  console.log(person.age);
  person.speak();
</script>

```



### 7.1.6、箭头函数

ES6 允许使用「箭头」 (=>) 定义函数，通用写法如下：

```

<script>
  let fn = (arg1, arg2, arg3) => {
    return arg1 + arg2 + arg3;
  }
</script>

```

箭头函数的注意点：

- 如果形参只有一个，则小括号可以省略
- 函数体如果只有一条语句，则花括号可以省略，函数的返回值为该条语句的执行结果
- 箭头函数 this 指向声明时所在作用域下 this 的值，箭头函数不会更改 this 指向，用来指定回调函数会非常合适
- 箭头函数不能作为构造函数实例化

- 不能使用 arguments 实参

省略小括号的情况：

```
<script>
  let fn = num => {
    return num * 10;
  };
</script>
```

省略花括号的情况：

```
<script>
  let fn = score => score * 20;
</script>
```

this 指向声明时所在作用域中 this 的值：

```
<script>
  // this 指向声明时所在作用域中 this 的值
  let fn = () => {
    console.log(this);
  }

  fn();

  let school = {
    name: "张三",
    getName() {
      let subFun = () => {
        console.log(this);
      }
      subFun();
    }
  };
  school.getName();
</script>
```



### 7.1.7、rest 参数

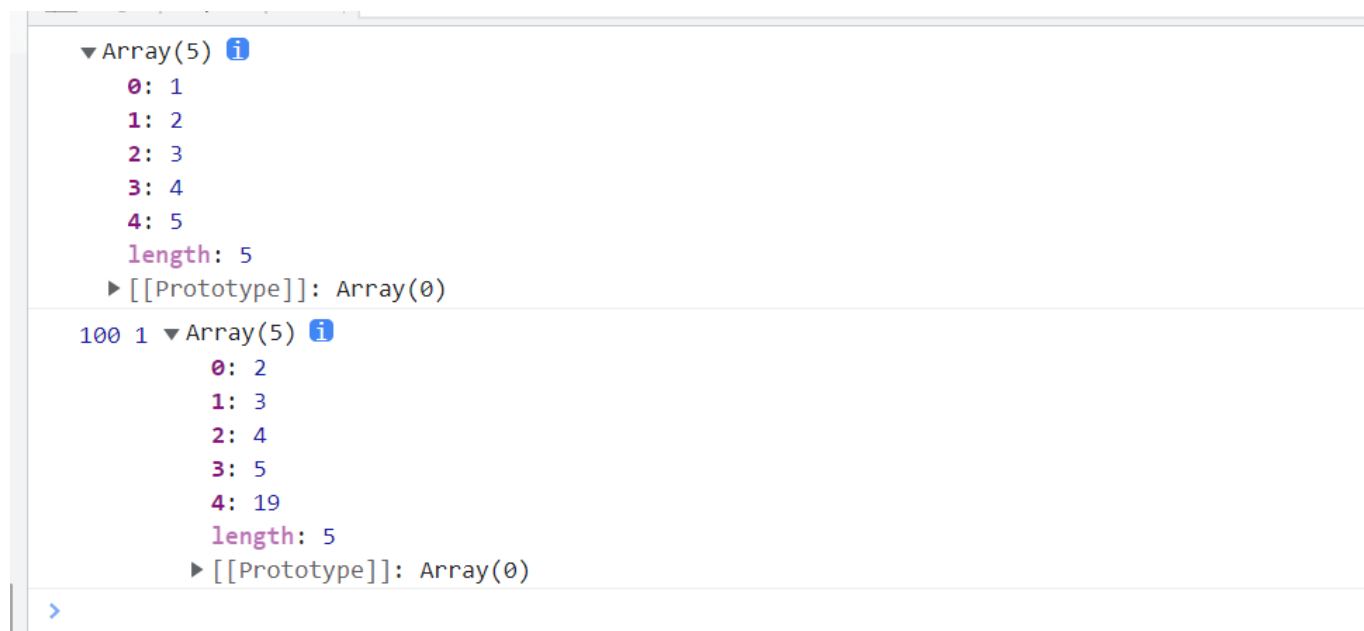


ES6 引入 rest 参数，用于获取函数的实参，用来代替 arguments 参数。

注意：rest 参数非常适合不定个数参数函数的场景

```
<script>
  // 作用与 arguments 类似
  function add(...args) {
    console.log(args);
  }
  add(1, 2, 3, 4, 5);

  // rest 参数必须是最后一个形参
  function minus(a, b, ...args) {
    console.log(a, b, args);
  }
  minus(100, 1, 2, 3, 4, 5, 19);
</script>
```



### 7.1.8、spread 扩展运算符

扩展运算符 (spread) 也是三个点 (...), 它好比 rest 参数的逆运算, 将一个数组转为用逗号分隔的参数序列, 对数组进行解包。

展开数组:

```

<script>
  // 展开数组
  let tfboys = ["德玛西亚之力", "德玛西亚之翼", "德玛西亚皇子"];
  function fn() {
    console.log(arguments);
  }
  fn(...tfboys);
</script>

```

▼ Arguments(3) ⓘ

- 0: "德玛西亚之力"
- 1: "德玛西亚之翼"
- 2: "德玛西亚皇子"
- ▶ callee: *f fn()*
- length: 3
- ▶ symbol(Symbol.iterator): *f values()*
- ▶ [[Prototype]]: Object

展开对象:

```

<script>
  // 展开对象
  let skillOne = {
    q: "致命打击"
  };
  let skillTwo = {
    w: "勇气"
  };
  let skillThree = {
    e: "审判"
  };
  let skillFour = {
    r: "德玛西亚正义"
  };
  let gailun = {...skillOne, ...skillTwo, ...skillThree, ...skillFour};
  console.log(gailun);
</script>

```

▼ Object ⓘ

- e: "审判"
- q: "致命打击"
- r: "德玛西亚正义"
- w: "勇气"
- ▶ [[Prototype]]: Object

## 7.1.9、Symbol类型

### 7.1.9.1、Symbol的使用

ES6 引入了一种新的原始数据类型 Symbol，表示独一无二的值，它是 JavaScript 语言的第七种数据类型，是一种类似于字符串的数据类型，Symbol 特点如下：

- Symbol 的值是唯一的，用来解决命名冲突的问题
- Symbol 值不能与其它数据进行运算
- Symbol 定义的对象属性不能使用 for...in 循环遍历，但是可以使用 Reflect.ownKeys 来获取对象的所有键名
- Symbol()符号永远不会相等，不管符号名是否相同

每个Symbol实例都是唯一的。因此，当你比较两个Symbol实例的时候，将总会返回false;

```

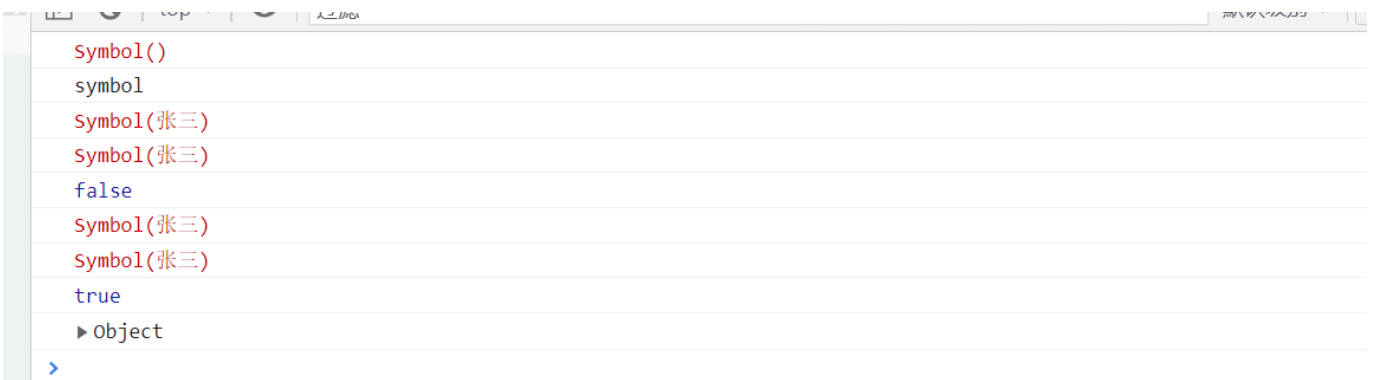
<script>
  //创建 Symbol
  let s1 = Symbol();
  console.log(s1);
  console.log(typeof s1);

  //添加标识的 Symbol
  let s2 = Symbol("张三");
  let s2_2 = Symbol("张三");
  console.log(s2);
  console.log(s2_2);
  console.log(s2 === s2_2);

  //使用 Symbol for 定义
  let s3 = Symbol.for("张三");
  let s3_2 = Symbol.for("张三");
  console.log(s3);
  console.log(s3_2);
  console.log(s3 === s3_2);

  //在方法中使用 Symbol
  let game = {
    name: "狼人杀",
    [Symbol('say')]: function () {
      console.log("我可以发言")
    },
    [Symbol('zibao')]: function () {
      console.log('我可以自爆');
    }
  };
  console.log(game);
</script>

```



注意：遇到唯一性的场景时要想到 Symbol

#### 7.1.9.2、Symbol内置值

除了定义自己使用的 Symbol 值以外，ES6 还提供了 11 个内置的 Symbol 值，指向语言内部使用的方法

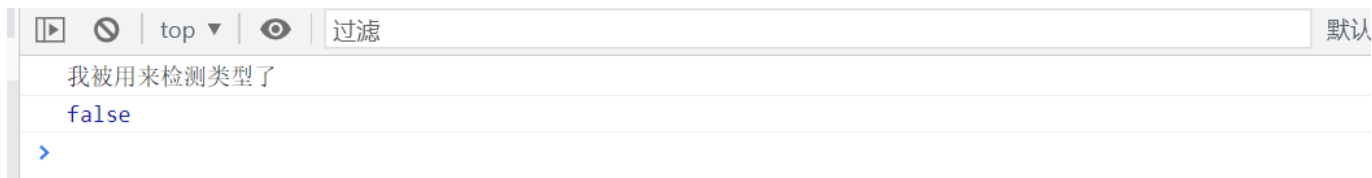
可以称这些方法为魔术方法，因为它们会在特定的场景下自动执行。

内置值	描述
Symbol.hasInstance	当其它对象使用 instanceof 运算符，判断是否为该对象的实例时，会调用这个方法
Symbol.isConcatSpreadable	对象的 Symbol.isConcatSpreadable 属性等于的是一个布尔值，表示该对象用于 Array.prototype.concat()时，是否可以展开
Symbol.species	创建衍生对象时，会使用该属性
Symbol.match	当执行 str.match(myObject) 时，如果该属性存在，会调用它，返回该方法的返回值
Symbol.replace	当该对象被 str.replace(myObject)方法调用时，会返回该方法的返回值
Symbol.search	当该对象被 str.search (myObject)方法调用时，会返回该方法的返回值
Symbol.split	当该对象被 str.split(myObject)方法调用时，会返回该方法的返回值
Symbol.iterator	当对象进行 for...of 循环时，会调用 Symbol.iterator 方法，返回该对象的默认遍历器
Symbol.toPrimitive	当对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值

Symbol. toStringTag	当对象上面调用 toString 方法时，返回该方法的返回值
Symbol. unscopables	当对象指定了使用 with 关键字时，哪些属性会被 with 环境排除

Symbol.hasInstance演示：

```
<script>
  class Person {
    static [Symbol.hasInstance](param) {
      console.log("我被用来检测类型了");
    }
  }
  let o = {};
  console.log(o instanceof Person);
</script>
```



## Symbol.isConcatSpreadable演示:

```
<script>
  const arr1 = [1, 2, 3];
  const arr2 = [4, 5, 6];
  arr2[Symbol.isConcatSpreadable] = true;
  ac1=arr1.concat(arr2);
  console.log(ac1);

  const arr3 = [1, 2, 3];
  const arr4 = [4, 5, 6];
  arr4[Symbol.isConcatSpreadable] = false;
  ac2=arr3.concat(arr4);
  console.log(ac2);
</script>
```

```
> ac1
< ▶ (6) [1, 2, 3, 4, 5, 6]
> ac2
< ▶ (4) [1, 2, 3, Array(3)]
>
```

### 7.1.10、迭代器

遍历器（Iterator）就是一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作。ES6 创造了一种新的遍历命令 for...of 循环，Iterator 接口主要供 for...of 消费，原生具备 iterator 接口的数据：

- Array
- Arguments
- Set
- Map
- String
- TypedArray

- NodeList

注意：需要自定义遍历数据的时候，要想到迭代器

## 工作原理：

1.创建一个指针对象，指向当前数据结构的起始位置 2.第一次调用对象的 next 方法，指针自动指向数据结构的第一个成员 3.接下来不断调用 next 方法，指针一直往后移动，直到指向最后一个成员 4.每调用 next 方法返回一个包含 value 和 done 属性的对象

## 案例演示：遍历数组

```
<script>
  //声明一个数组
  const xiyou = ["唐僧", "孙悟空", "猪八戒", "沙僧"];
  //之前的方法
  for(var i=0;i<xiyou.length;i++){
    console.log(xiyou[i]);
  }
  console.log("=====");

  //使用 for...of 遍历数组
  for (let v of xiyou) {
    console.log(v);
  }
  console.log("=====");

  //获取迭代器对象
  let iterator = xiyou[Symbol.iterator]();
  //调用对象的next方法
  console.log(iterator.next());
  console.log(iterator.next());
  console.log(iterator.next());
  console.log(iterator.next());
  console.log(iterator.next());
</script>
```



### 7.1.11、生成器

生成器函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。

#### 7.1.11.1、生成器函数使用

代码说明：

- \*的位置没有限制
- 生成器函数返回的结果是迭代器对象，调用迭代器对象的 next 方法可以得到 yield 语句后的值
- yield 相当于函数的暂停标记，也可以认为是函数的分隔符，每调用一次 next 方法，执行一段代码
- next 方法可以传递实参，作为 yield 语句的返回值



```
<script>
  function * gen() {
    /*代码1开始执行*/
    console.log("代码1执行了");
    yield "一只没有耳朵";
    /*代码2开始执行*/
    console.log("代码2执行了");
    yield "一只没有尾巴";
    /*代码3开始执行*/
    console.log("代码3执行了");
    return "真奇怪";
  }

  let iterator = gen();
  console.log(iterator.next());
  console.log(iterator.next());
  console.log(iterator.next());
  console.log("=====");

  //遍历
  for (let v of gen()) {
    console.log(v);
  }
</script>
```

代码1执行了
▼Object ⓘ done: false value: "一只没有耳朵" ▶ [[Prototype]]: Object
代码2执行了
▼Object ⓘ done: false value: "一只没有尾巴" ▶ [[Prototype]]: Object
代码3执行了
▼Object ⓘ done: true value: "真奇怪" ▶ [[Prototype]]: Object
=====
代码1执行了
一只没有耳朵
代码2执行了
一只没有尾巴
代码3执行了
>

### 7.1.11.2、生成器函数参数

```

<script>
  function * gen(arg) {
    console.log(arg);
    let one = yield 111;
    console.log(one);
    let two = yield 222;
    console.log(two);
    let three = yield 333;
    console.log(three);
  }

  //执行获取迭代器对象
  let iterator = gen('AAA');
  console.log(iterator.next());

  //next方法可以传入实参
  console.log(iterator.next('BBB'));
  console.log(iterator.next('CCC'));
  console.log(iterator.next('DDD'));
</script>

```

AAA
▼ Object ⓘ done: false value: 111 ▶ [[Prototype]]: Object
BBB
▼ Object ⓘ done: false value: 222 ▶ [[Prototype]]: Object
CCC
▼ Object ⓘ done: false value: 333 ▶ [[Prototype]]: Object
DDD
▼ Object ⓘ done: true value: undefined ▶ [[Prototype]]: Object
>

### 7.1.11.3、生成器函数实例

案例演示：1s后控制台输出 111，2s后输出 222，3s后输出 333

```

<script>
  function one() {
    setTimeout(() => {
      console.log(111);
      iterator.next();
    }, 1000)
  }

  function two() {
    setTimeout(() => {
      console.log(222);
      iterator.next();
    }, 2000)
  }

  function three() {
    setTimeout(() => {
      console.log(333);
      iterator.next();
    }, 3000)
  }

  function * gen() {
    yield one();
    yield two();
    yield three();
  }

  //调用生成器函数
  let iterator = gen();
  iterator.next();

</script>

```

		top ▼		过滤	默认级别 ▼
111					
222					
333					
>					

案例演示：模拟获取，用户数据，订单数据，商品数据

```

<script>
  function getUsers() {
    setTimeout(() => {
      let data = "用户数据";
      console.log("getUsers()");
      iterator.next(data);
    }, 1000);
  }

  function getOrders() {
    setTimeout(() => {
      let data = "订单数据";
      console.log("getOrders()");
      iterator.next(data);
    }, 1000);
  }

  function getGoods() {
    setTimeout(() => {
      let data = "商品数据";
      console.log("getGoods()");
      iterator.next(data);
    }, 1000);
  }

  function * gen() {
    let users = yield getUsers();
    console.log(users);
    let orders = yield getOrders();
    console.log(orders);
    let goods = yield getGoods();
    console.log(goods);
  }

  //调用生成器函数
  let iterator = gen();
  iterator.next();
</script>

```

		top ▼		过滤	默认级别 ▼
getUsers()					
用户数据					
getOrders()					
订单数据					
getGoods()					
商品数据					
>					

## 7.1.12、Promise

Promise 是 ES6 引入的异步编程的新解决方案，语法上 Promise 是一个构造函数，用来封装异步操作并可以获取其成功或失败的结果。

#### 7.1.12.1、Promise基本使用

```
<script>
//实例化 Promise 对象
const p = new Promise(function (resolve, reject) {
  setTimeout(function () {

    // 成功调用resolve()处理
    let data = "数据读取成功";
    resolve(data);

    // 失败调用reject()处理
    let err = "数据读取失败";
    reject(err);

  }, 1000);
});

//调用 promise 对象的 then 方法
p.then(function (value) {
  console.log(value);
}, function (reason) {
  console.error(reason);
});
</script>
```

#### 7.1.12.2、Promise案例演示

案例演示：

```

<script>
// 接口地址: https://www.baidu.com
const p = new Promise((resolve, reject) => {
  //1. 创建对象
  const xhr = new XMLHttpRequest();
  //2. 初始化
  xhr.open("GET", "https://www.baidu.com");
  //3. 发送
  xhr.send();
  //4. 绑定事件, 处理响应结果
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      //判断响应状态码 200-299
      if (xhr.status >= 200 && xhr.status < 300) {
        //表示成功
        resolve(xhr.response);
      } else {
        //如果失败
        reject(xhr.status);
      }
    }
  }
});

//指定回调
p.then(function (value) {
  console.log(value);
}, function (reason) {
  console.error(reason);
});

</script>

```

### 7.1.12.3、Promise-then方法




调用 then 方法, then 方法的返回结果是 Promise 对象, 对象状态由回调函数的执行结果决定, 如果回调函数中返回的结果是非 promise 类型的属性, 状态为成功, 返回值为对象的成功的值

```

<script>
  //创建 promise 对象
  const p = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("用户数据");
    }, 1000)
  });

  //链式调用+箭头函数
  p.then(value => {
    console.log(value);
    return value;
  }).then(value => {
    console.log(value);
  });
</script>

```

		top ▼		过滤	默认级别 ▼
				用户数据	
				用户数据	
				>	

#### 7.1.12.4、Promise-catch方法




如果只想处理错误状态，我们可以使用 catch 方法

```

<script>
  const p = new Promise((resolve, reject) => {
    setTimeout(() => {
      //设置 p 对象的状态为失败，并设置失败的值
      reject("出错啦!");
    }, 1000);
  });

  p.catch(function (reason) {
    console.error(reason);
  });
</script>

```

		top ▼		过滤	默认级别 ▼
				✖ 出错啦!	
				>	

#### 7.1.13、Set



ES6 提供了新的数据结构 Set（集合）。它类似于数组，但成员的值都是唯一的，集合实现了 iterator 接口，所以可以使用『扩展运算符』和『for...of...』进行遍历，集合的属性和方法：

- size：返回集合的元素个数
- add()：增加一个新元素，返回当前集合
- delete()：删除元素，返回 boolean 值
- has()：检测集合中是否包含某个元素，返回 boolean 值
- clear()：清空集合，返回 undefined

```
<script>
  //创建一个空集合
  let s = new Set();
  //创建一个非空集合
  let s1 = new Set([1, 2, 3, 1, 2, 3]);
  //集合属性与方法
  //返回集合的元素个数
  console.log(s1.size);
  //添加新元素
  console.log(s1.add(4));
  //删除元素
  console.log(s1.delete(1));
  //检测是否存在某个值
  console.log(s1.has(2));
  //清空集合
  //console.log(s1.clear());
</script>
```

#### 7.1.14、Map

ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合。但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。Map 也实现了 iterator 接口，所以可以使用『扩展运算符』和『for...of...』进行遍历。Map 的属性和方法：

- size：返回 Map 的元素个数
- set()：增加一个新元素，返回当前 Map
- get()：返回键名对象的键值
- has()：检测 Map 中是否包含某个元素，返回 boolean 值
- clear()：清空集合，返回 undefined

```
<script>
  //创建一个空 map
  let m = new Map();
  //创建一个非空 map
  let m2 = new Map([
    ["name", "张三"],
    ["gender", "女"]
  ]);
  //属性和方法
  //获取映射元素的个数
  console.log(m2.size);
  //添加映射值
  console.log(m2.set("age", 6));
  //获取映射值
  console.log(m2.get("age"));
  //检测是否有该映射
  console.log(m2.has("age"));
  //清除
  console.log(m2.clear());
</script>
```

#### 7.1.15、class 类

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过 class 关键字，可以定义类。基本上，ES6 的 class 可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的 class 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已，它的一些如下：

- class：声明类
- constructor：定义构造函数初始化
- extends：继承父类
- super：调用父级构造方法
- static：定义静态方法和属性

```
<script>
//父类
class Phone {
    //构造方法
    constructor(brand, color, price) {
        this.brand = brand;
        this.color = color;
        this.price = price;
    }

    //对象方法
    call() {
        console.log("我可以打电话!!!")
    }
}

//子类
class SmartPhone extends Phone {
    constructor(brand, color, price, screen, pixel) {
        super(brand, color, price);
        this.screen = screen;
        this.pixel = pixel;
    }

    //子类方法
    photo() {
        console.log("我可以拍照!!");
    }

    playGame() {
        console.log("我可以玩游戏!!");
    }

    //方法重写
    call() {
        console.log("我可以进行视频通话!!");
    }

    //静态方法
    static run() {
        console.log("我可以运行程序")
    }

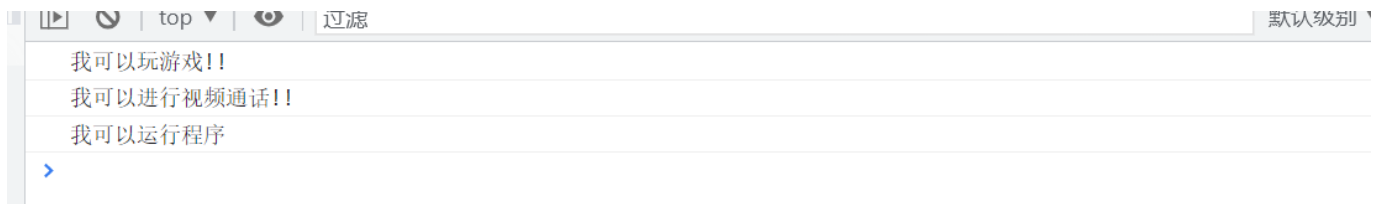
    static connect() {
        console.log("我可以建立连接")
    }
}

//实例化对象
const Nokia = new Phone("诺基亚", "灰色", 230);
```

```
const iPhone6s = new SmartPhone("苹果", "白色", 6088,
                                "4.7inch", "500w");

//调用子类方法
iPhone6s.playGame();
//调用重写方法
iPhone6s.call();
//调用静态方法
SmartPhone.run();

</script>
```

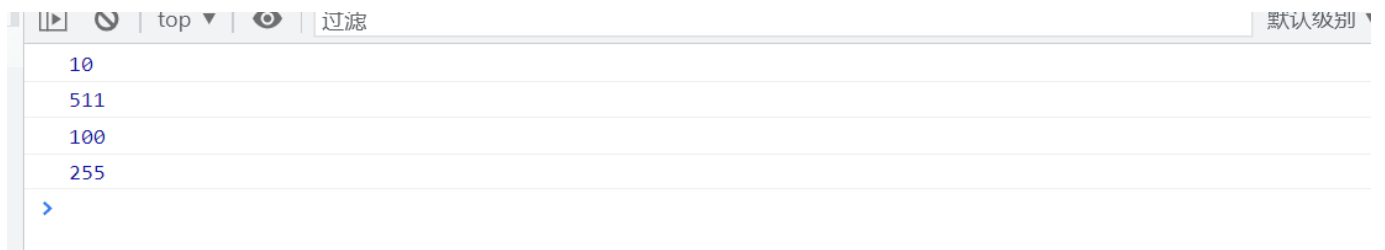


## 7.1.16、数值扩展

### 7.1.16.1、二进制和八进制

ES6 新增了二进制和八进制的表示方法

```
<script>
let b = 0b1010//二进制
let o = 0o777;//八进制
let d = 100;//十进制
let x = 0xff;//十六进制
console.log(b);
console.log(o);
console.log(d);
console.log(x);
</script>
```



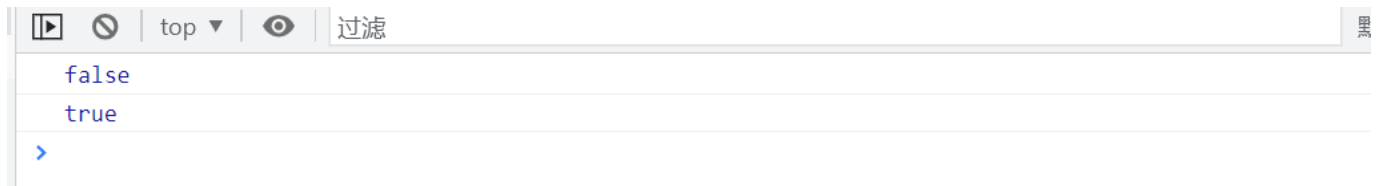
### 7.1.16.2、Number.EPSILON

Number.EPSILON：它是 JavaScript 表示的最小精度，EPSILON 属性的值接近于 2.2204460492503130808472633361816E-16

```

<script>
  function equal(a, b) {
    if (Math.abs(a - b) < Number.EPSILON) {
      return true;
    } else {
      return false;
    }
  }
  console.log(0.1 + 0.2 === 0.3);
  console.log(equal(0.1 + 0.2, 0.3));
</script>

```



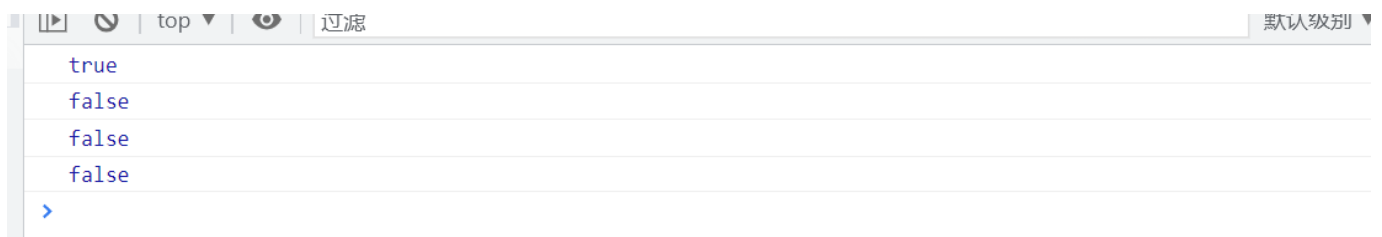
### 7.1.16.3、Number.isFinite

Number.isFinite：检测一个数值是否为有限数

```

<script>
  console.log(Number.isFinite(100));
  console.log(Number.isFinite(100 / 0));
  console.log(Number.isFinite(Infinity));
  console.log(Number.isFinite(-Infinity));
</script>

```



### 7.1.16.4、Number.isNaN

Number.isNaN：检测一个数值是否为 NaN

```

<script>
  console.log(Number.isNaN(123));
</script>

```



#### 7.1.16.5、Number.parseInt

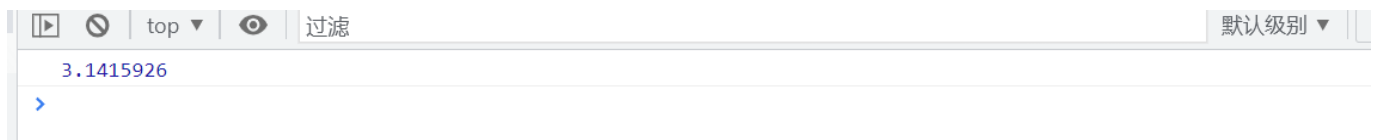
Number.parseInt：将一个字符串转换为整数

```
<script>
  console.log(Number.parseInt("123abc"));
</script>
```

#### 7.1.16.6、Number.parseFloat

Number.parseFloat：将一个字符串转换为浮点数

```
<script>
  console.log(Number.parseFloat("3.1415926神奇"));
</script>
```



#### 7.1.16.7、Number.isInteger

Number.isInteger：判断一个数是否为整数

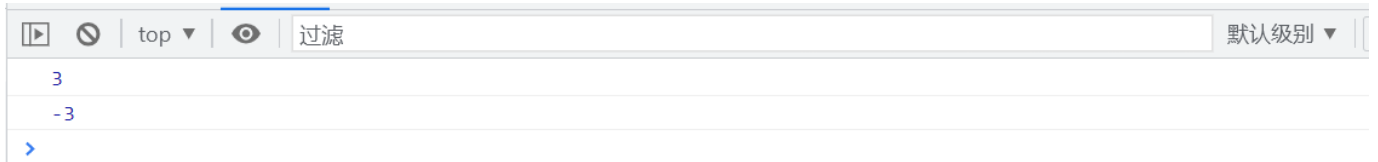
```
<script>
  console.log(Number.isInteger(5));
  console.log(Number.isInteger(2.5));
</script>
```



#### 7.1.16.8、Math.trunc

Math.trunc：将数字的小数部分抹掉

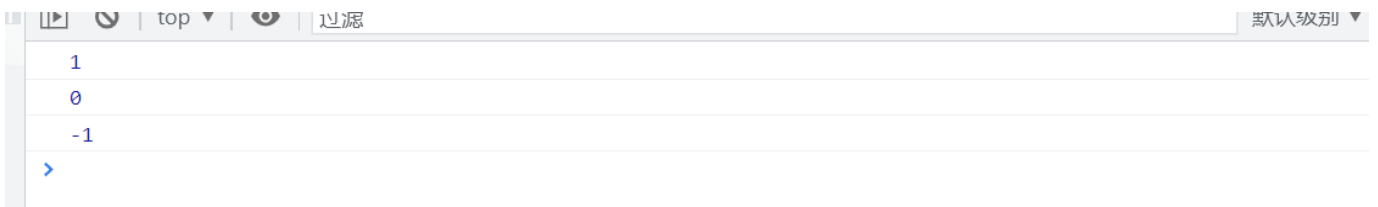
```
<script>
  console.log(Math.trunc(3.5));
  console.log(Math.trunc(-3.5));
</script>
```



#### 7.1.16.9、Math.sign

Math.sign: 判断一个数到底为正数、负数、还是零

```
<script>
  console.log(Math.sign(100));
  console.log(Math.sign(0));
  console.log(Math.sign(-20000));
</script>
```



#### 7.1.17、对象扩展

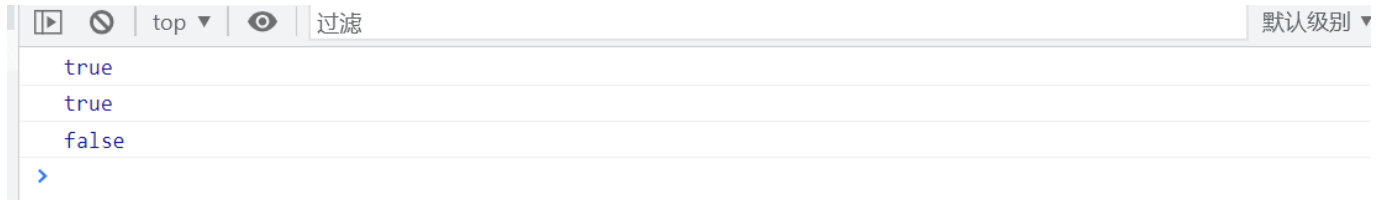
ES6 新增了一些 Object 对象的方法，例如：

- Object.is: 比较两个值是否严格相等，与『===』行为基本一致（+0 与 NaN）
- Object.assign: 对象的合并，将源对象的所有可枚举属性，复制到目标对象
- \_proto\_、setPrototypeOf、setPrototypeOf可以直接设置对象的原型

##### 7.1.17.1、Object.is

Object.is: 判断两个值是否完全相等

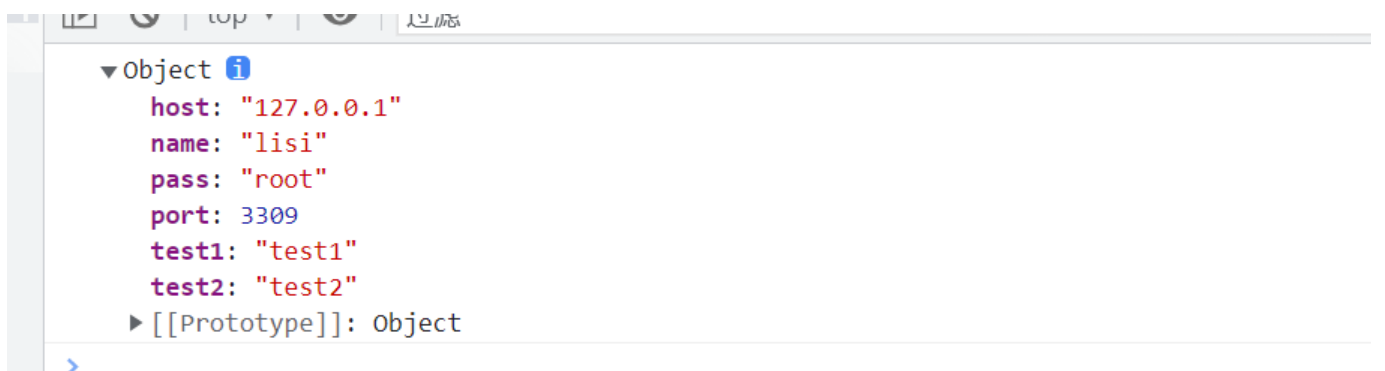
```
<script>
  console.log(Object.is(120, 120));// ===
  console.log(Object.is(NaN, NaN));// ===
  console.log(NaN === NaN);// ===
</script>
```



#### 7.1.17.2、Object.assign

Object.assign: 对象的合并，后边的对象会把前边对象的相同属性和方法覆盖，没有的属性和方法会合并

```
<script>
  const config1 = {
    host: "localhost",
    port: 3306,
    name: "zhangsan",
    pass: "root",
    test1: "test1"
  };
  const config2 = {
    host: "127.0.0.1",
    port: 3309,
    name: "lisi",
    pass: "root",
    test2: "test2"
  }
  console.log(Object.assign(config1, config2));
</script>
```

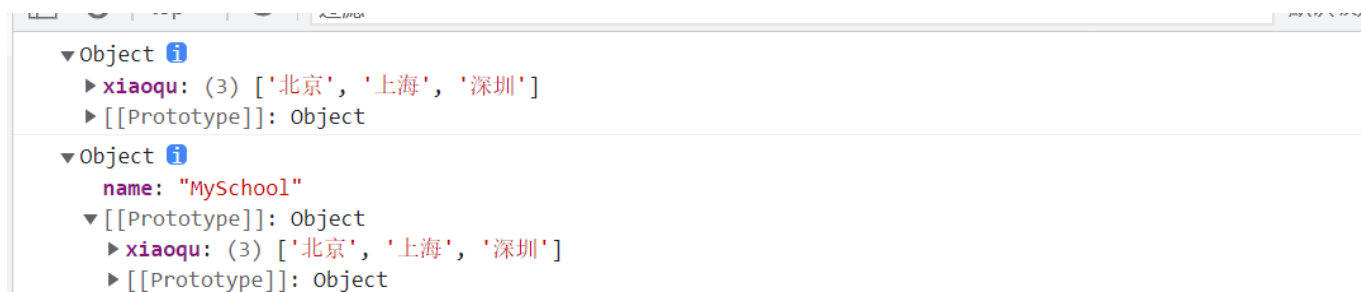




### 7.1.17.3、设置原型对象

- Object.setPrototypeOf: 设置原型对象
- Object.getPrototypeOf: 获取原型对象

```
<script>
  const school = {
    name: "MySchool"
  };
  const cities = {
    xiaoqu: ["北京", "上海", "深圳"]
  };
  Object.setPrototypeOf(school, cities);
  console.log(Object.getPrototypeOf(school));
  console.log(school);
</script>
```



### 7.1.18、模块化

模块化是指将一个大的程序文件，拆分成许多小的文件，然后将小文件组合起来。

#### 7.1.18.1、模块化的好处

- 防止命名冲突
- 代码复用
- 高维护性

#### 7.1.18.2、模块化的产品

- CommonJS => NodeJS、Browserify
- AMD => requireJS
- CMD => seaJS

### 7.1.18.3、模块化的语法

模块功能主要由两个命令构成：export 和 import。

- export 命令用于规定模块的对外接口
- import 命令用于输入其它模块提供的功能

### 7.1.18.4、模块化的暴露

- m1.js

```
<script>
  //方式一：分别暴露
  export let school = "华北理工大学";
  export function study() {
    console.log("我们要学习！");
  }
</script>
```

- m2.js

```
<script>
  //方式二：统一暴露
  let school = "华北理工大学";

  function findJob() {
    console.log("我们要找工作！");
  }

  export {school, findJob};
</script>
```

- m3.js

```
<script>
  //方式三：默认暴露
  export default {
    school: "华北理工大学",
    change: function () {
      console.log("我们要改变自己!");
    }
  }
</script>
```

#### 7.1.18.5、模块化的导入

index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>

<!-- 在这里写JavaScript代码，因为JavaScript是由上到下执行的 -->
<script type="module">
  // 引入 m1.js 模块内容
  import * as m1 from "./m1.js";
  // 引入 m2.js 模块内容
  import * as m2 from "./m2.js";
  // 引入 m3.js 模块内容
  import * as m3 from "./m3.js";

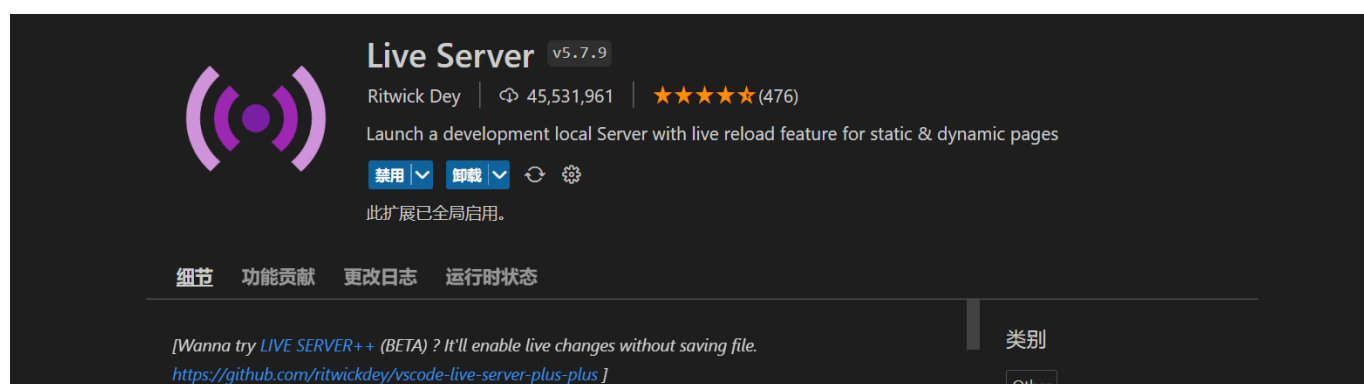
  m1.study();
  m2.findJob();
  m3.default.change();
</script>
</body>
</html>
```



在实验时，总是出现Access to script。。。。。

经过查阅资料可以用如下方案解决：

去 vs code 下载Live Server插件即可完美解决：



并且

注意需要切换：



默认打开的是[http://127.0.0.1:5500/...](http://127.0.0.1:5500/)

#### 7.1.18.6、解构赋值形式

index.html

```

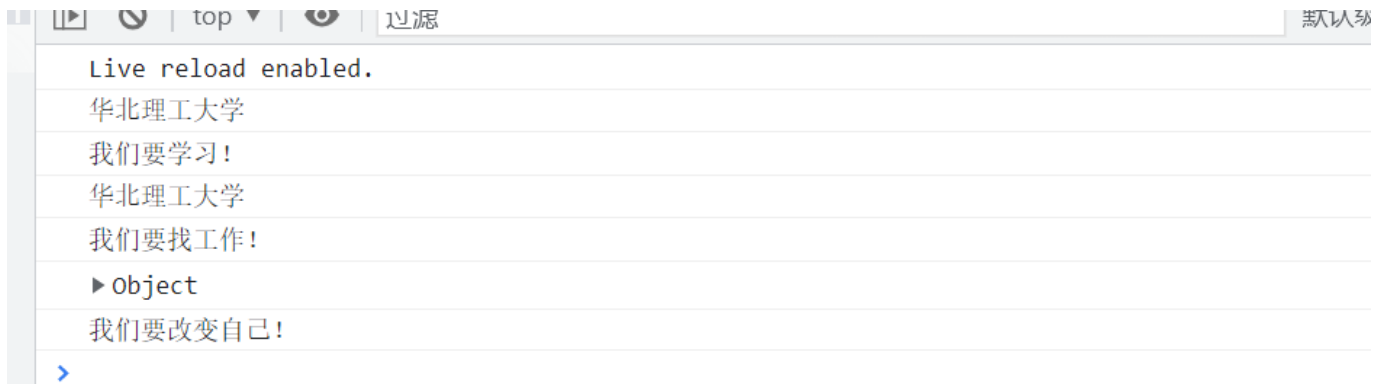
<script type="module">
  // 引入 m1.js 模块内容
  import {school, study} from "./m1.js";
  // 引入 m2.js 模块内容
  import {school as s, findJob} from "./m2.js";
  // 引入 m3.js 模块内容
  import {default as m3} from "./m3.js";

  console.log(school);
  study();

  console.log(s);
  findJob();

  console.log(m3);
  m3.change();
</script>

```



注意：针对默认暴露还可以直接 `import m3 from "./m3.js"`

### 7.1.19、浅拷贝和深拷贝

如何区分深拷贝与浅拷贝，简单点来说，就是假设B复制了A，当修改A时，看B是否会发生变化，如果B也跟着变了，说明这是浅拷贝；如果B没变，那就是深拷贝，深拷贝与浅拷贝的概念只存在于引用数据类型。

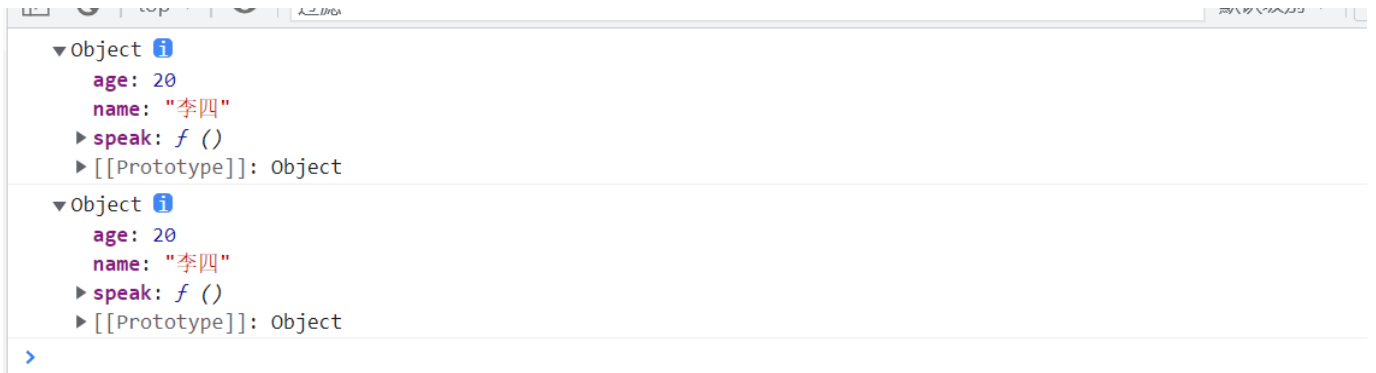
#### 7.1.19.1、浅拷贝

```
<script>
  var obj1 = {
    name: "张三",
    age: 20,
    speak: function () {
      console.log("我是" + this.name);
    }
  };

  var obj2 = obj1;

  // 当修改obj2的属性和方法的时候，obj1相应的属性和方法也会改变
  obj2.name = "李四";
  console.log(obj1);
  console.log(obj2);

</script>
```



### 7.1.19.2、深拷贝

#### 7.1.19.2.1、自带的

Array: `slice()`、`concat()`、`Array.from()`、... 操作符：只能实现一维数组的深拷贝

**`slice()`方法演示：**

```
<script >
  var arr1 = [1, 2, 3, 4];
  var arr2 = arr1.slice();
  arr2[0] = 200;
  console.log(arr1);
  console.log(arr2);
</script>
```

```
> arr1
< ▶ (4) [1, 2, 3, 4]
> arr2
< ▶ (4) [200, 2, 3, 4]
>
```

### concat()方法演示:

```
<script >
  var arr1 = [1, 2, 3, 4];
  var arr2 = arr1.concat();
  arr2[0] = 200;
  console.log(arr1);
  console.log(arr2);
</script>
```

```
> arr1
< ▶ (4) [1, 2, 3, 4]
> arr2
< ▶ (4) [200, 2, 3, 4]
>
```

### Array.from()方法演示:

```
<script >
  var arr1 = [1, 2, 3, 4];
  var arr2 = Array.from(arr1);
  arr2[0] = 200;
  console.log(arr1);
  console.log(arr2);
</script>
```

```
> arr1
< ▶ (4) [1, 2, 3, 4]
> arr2
< (4) [200, 2, 3, 4]
```

### ... 操作符演示:

```
<script >
  var arr1 = [1, 2, 3, 4];
  var arr2 = [...arr1];
  arr2[0] = 200;
  console.log(arr1);
  console.log(arr2);
</script>
```

```
> arr1
< ▶ (4) [1, 2, 3, 4]
> arr2
< ▶ (4) [200, 2, 3, 4]
>
```

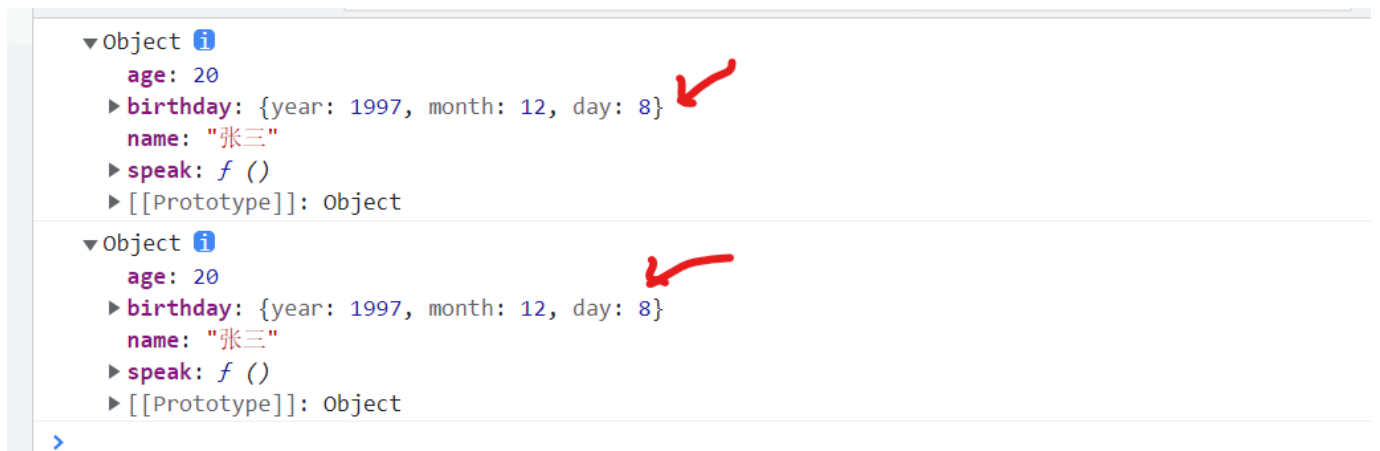
```
<script >
  var obj1 = {
    name: "张三",
    age: 20,
    birthday: {
      year: 1997,
      month: 12,
      day: 5
    },
    speak: function () {
      console.log("我是" + this.name);
    }
  };

  var obj2 = {
    ...obj1
  };
};
```

```
// 当修改obj2的属性和方法的时候，obj1相应的属性和方法不会改变
obj2.birthday.day=8;
console.log(obj1);
console.log(obj2);
```

```
</script>
```





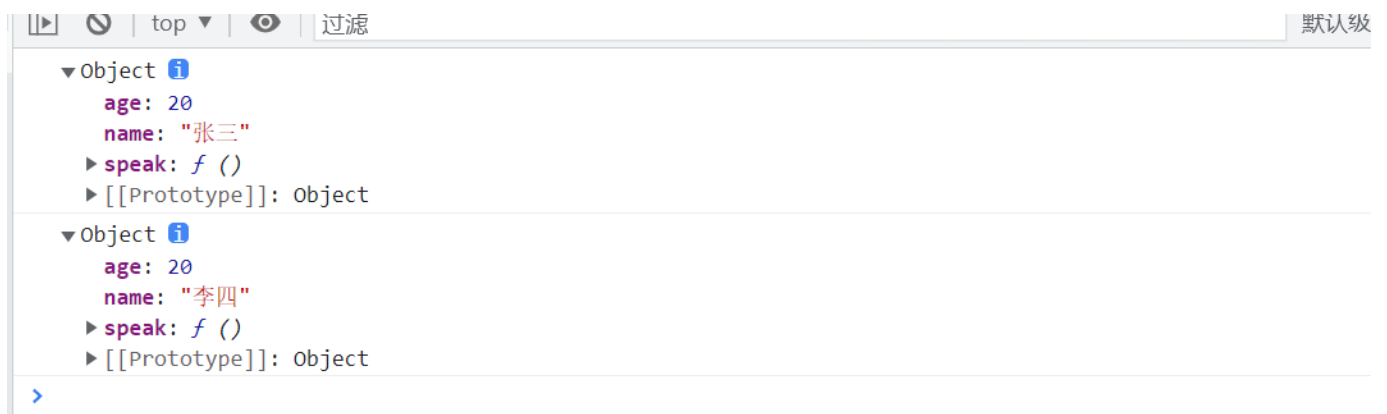
Object: Object.assign()、... 操作符：只能实现一维对象的深拷贝

Object.assign()方法演示：

```
<script >
  var obj1 = {
    name: "张三",
    age: 20,
    speak: function () {
      console.log("我是" + this.name);
    }
  };

  var obj2 = Object.assign({}, obj1);

  // 当修改obj2的属性和方法的时候，obj1相应的属性和方法不会改变
  obj2.name = "李四";
  console.log(obj1);
  console.log(obj2);
</script>
```



... 操作符演示：

```

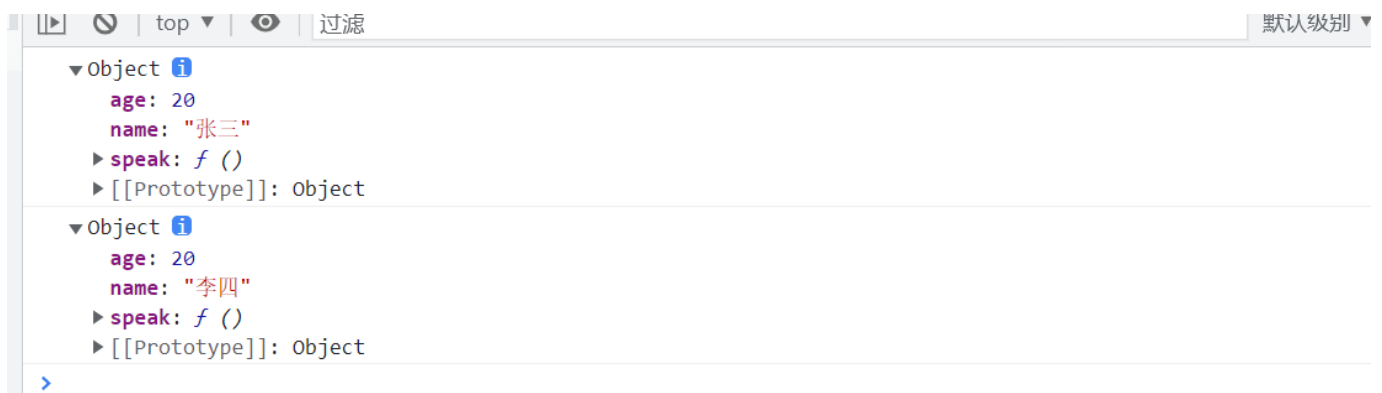
<script >
  var obj1 = {
    name: "张三",
    age: 20,
    speak: function () {
      console.log("我是" + this.name);
    }
  };

  var obj2 = {
    ...obj1
  };

  // 当修改obj2的属性和方法的时候，obj1相应的属性和方法不会改变
  obj2.name = "李四";
  console.log(obj1);
  console.log(obj2);

</script>

```



JSON.parse(JSON.stringify(obj)): 可实现多维对象的深拷贝，但会忽略 undefined、任意的函数、Symbol 值

```

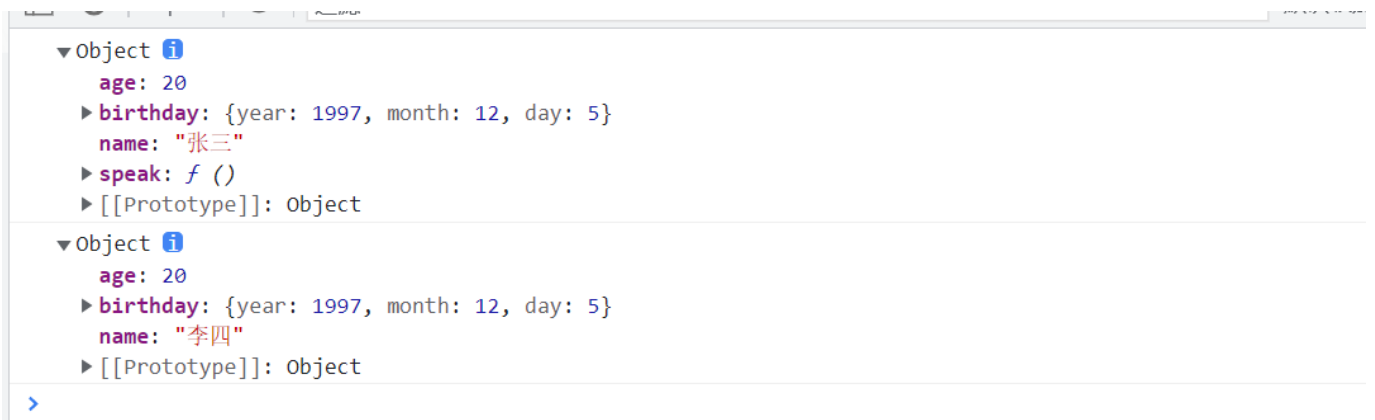
<script >
  var obj1 = {
    name: "张三",
    age: 20,
    birthday: {
      year: 1997,
      month: 12,
      day: 5
    },
    speak: function () {
      console.log("我是" + this.name);
    }
  };

  var obj2 = JSON.parse(JSON.stringify(obj1));

  // 当修改obj2的属性和方法的时候，obj1相应的属性和方法不会改变
  obj2.name = "李四";
  console.log(obj1);
  console.log(obj2);

</script>

```



注意：进行JSON.stringify()序列化的过程中，undefined、任意的函数以及 symbol 值，在序列化过程中会被忽略（出现在非数组对象的属性值中时）或者被转换成 null（出现在数组中时），由上面可知，JS 提供的自有方法并不能彻底解决Array、Object的深拷贝问题，因此我们应该自己实现。

#### 7.1.19.2.2、通用版

```

<script >
    var obj1 = {
        name: "张三",
        age: 20,
        birthday: {
            year: 1997,
            month: 12,
            day: 5
        },
        speak: function () {
            console.log("我是" + this.name);
        }
    };

    var obj2 = deepClone(obj1);

    // 当修改obj2的属性和方法的时候，obj1相应的属性和方法不会改变
    obj2.name = "李四";
    console.log(obj1);
    console.log(obj2);

    /**
     * 深拷贝通用方法
     * @param obj    需要拷贝的对象
     * @param has
     * @returns {any|RegExp|Date}
     */
    function deepClone(obj, has = new WeakMap()) {
        // 类型检查
        if (obj == null) return obj;
        if (obj instanceof Date) return obj;
        if (obj instanceof RegExp) return obj;
        if (!(typeof obj == "object")) return obj;

        // 构造对象
        const newObj = new obj.constructor;

        // 防止自引用导致的死循环
        if (has.get(obj)) return has.get(obj);
        has.set(obj, newObj);

        // 循环遍历属性及方法
        for (let key in obj) {
            if (obj.hasOwnProperty(key)) {
                newObj[key] = deepClone(obj[key]);
            }
        }

        // 返回对象
        return newObj;
    }

```

&lt;/script&gt;

