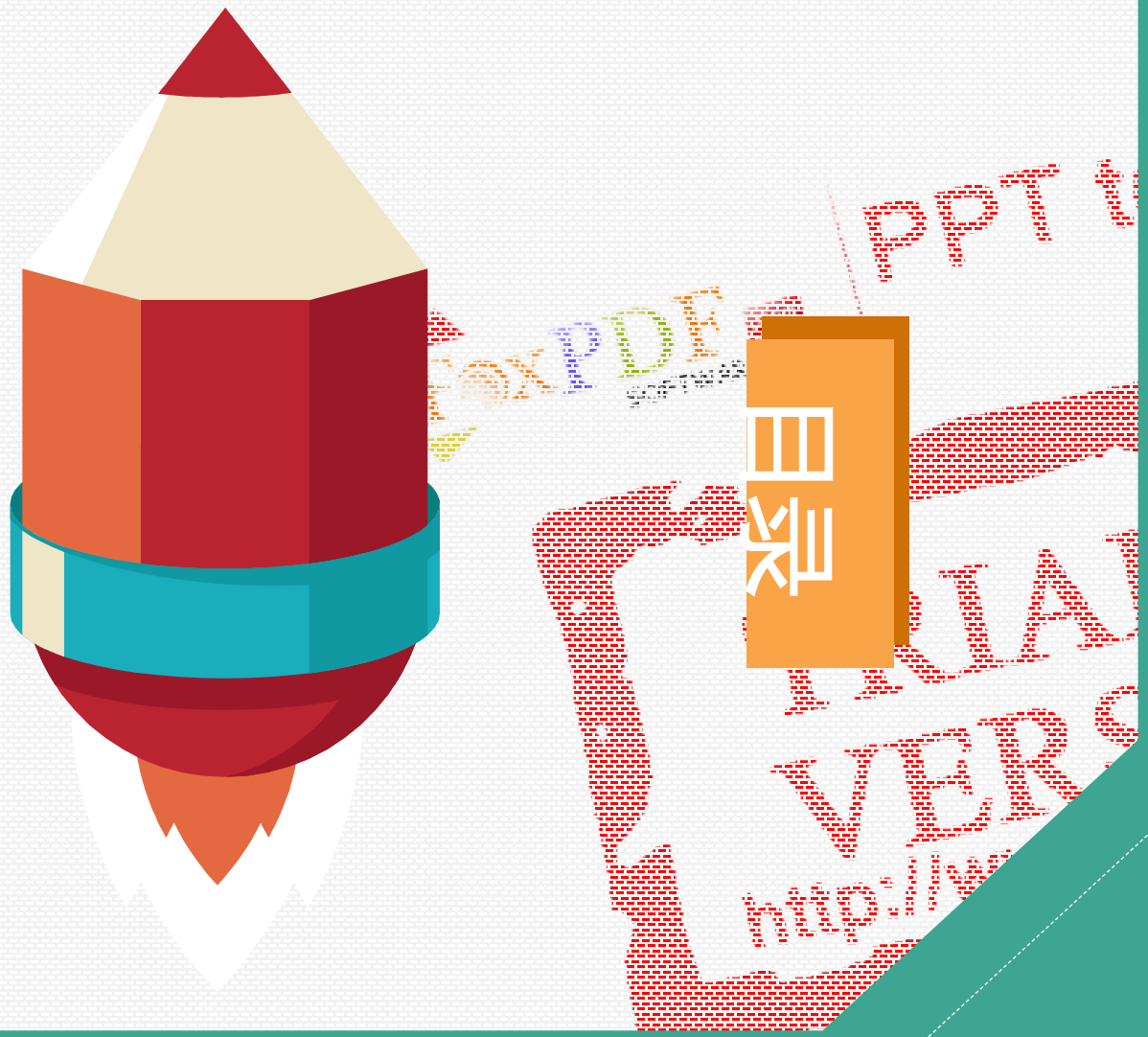

第30课：数组方法拓展
主讲老师：万章



数组方法的拓展

Symbol数据类型

数组的拓展方法

数组的拓展之扩展方法Array.from

Array.from方法用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）

```
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};

// ES5的写法
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']

// ES6的写法
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

上面是一个类似数组的对象，Array.from将它转为真正的数组。

```
// NodeList对象
let ps = document.querySelectorAll('p');
Array.from(ps).filter(p => {
  return p.textContent.length > 100;
});

// arguments对象
function foo() {
  var args = Array.from(arguments);
  // ...
}
```

实际应用中，常见的类似数组的对象是 DOM 操作返回的 NodeList 集合，以及函数内部的arguments对象。Array.from都可以将它们转为真正的数组。

数组的拓展之扩展方法Array.from

Array.from方法还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有length属性。因此，任何有length属性的对象，都可以通过Array.from方法转为数组(如果没有length属性,那就返回一个空数组)，而此时扩展运算符就无法转换。

```
> [...{}]
▶Uncaught TypeError: object is not iterable VM521:1
(cannot read property Symbol(Symbol.iterator))
    at <anonymous>:1:1
>
```

```
> Array.from({x:1,y:3})
◀ ▶[]
> Array.from({length:3})
◀ ▶(3) [undefined, undefined, undefined]
> Array.from({})
◀ ▶[]
>
```

数组的拓展之扩展方法Array.from

Array.from还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。

```
Array.from(arrayLike, x => x * x);
// 等同于
Array.from(arrayLike).map(x => x * x);

Array.from([1, 2, 3], (x) => x * x)
// [1, 4, 9]
```

数组的拓展之扩展方法Array.of

Array.of方法用于将一组值，转换为数组。这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array()的行为有差异。

```
Array.of(3, 11, 8) // [3,11,8]
Array.of(3) // [3]
Array.of(3).length // 1
```

```
Array() // []
Array(3) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

Array.of方法使用起来稳定明了，传入的所有参数都是数组的项目.

Array.of总是返回参数值组成的数组。如果没有参数，就返回一个空数组。

上面代码中，Array方法没有参数、一个参数、三个参数时，返回结果都不一样。只有当参数个数不少于 2 个时，Array()才会返回由参数组成的新数组。参数个数只有一个时，实际上是指定数组的长度。

Array.of基本上可以用来替代Array()或new Array()，并且不存在由于参数不同而导致的结果的差异。它的行为非常统一。

数组的拓展之扩展方法copyWithin()

数组实例的copyWithin()方法就是在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，**使用这个方法，会修改当前数组。**

```
[ ].copyWithin(target,start,end)
```

1. target（必需）：从该位置开始替换数据。如果为负值，表示倒数。
2. start（可选）：从该位置开始读取数据，默认为 0。如果为负值，表示从末尾开始计算。
3. end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示从末尾开始计算。

```
// 将3号位复制到0号位
[1, 2, 3, 4, 5].copyWithin(0, 3, 4)
// [4, 2, 3, 4, 5]

// -2相当于3号位，-1相当于4号位
[1, 2, 3, 4, 5].copyWithin(0, -2, -1)
// [4, 2, 3, 4, 5]

// 将3号位复制到0号位
[].copyWithin.call({length: 5, 3: 1}, 0, 3)
// {0: 1, 3: 1, length: 5}
```

数组的拓展之扩展方法find()/findIndex()

数组实例的find方法，用于找出第一个符合条件的数组项目。它的**参数是一个回调函数**，所有数组成员依次执行该回调函数，直到找出**第一个返回值为true的成员，然后返回该成员**。如果没有符合条件的成员，则返回undefined。（有点类似于some和every，只不过这个返回的不是布尔值而是符合条件的数组项目）

```
[1, 5, 10, 15].find(function(value, index, arr) {
  return value > 9;
}) // 10
```

上面代码中，find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。

数组实例的findIndex方法的用法与find方法非常类似，返回**第一个符合条件的数组成员的位置**，如果所有成员都不符合条件，则返回-1。

```
[1, 5, 10, 15].findIndex(function(value, index, arr) {
  return value > 9;
}) // 2
```

数组的拓展之扩展方法find()/findIndex()

这两个方法都可以接受第二个参数，用来绑定回调函数里面的this对象。

```
function f(v){
  return v > this.age;
}
let person = {name: 'John', age: 20};
[10, 12, 26, 15].find(f, person);    // 26
```

上面的代码中，find函数接收了第二个参数person对象，回调函数中的this对象指向person对象。

数组的拓展之扩展方法fill()

fill方法使用给定值，填充一个数组。

```
['a', 'b', 'c'].fill(7)
// [7, 7, 7]

new Array(3).fill(7)
// [7, 7, 7]
```

右侧代码表明，fill方法用于空数组的初始化非常方便。
数组中已有的元素，会被全部抹去。

```
['a', 'b', 'c'].fill(7, 1, 2)
// ['a', 7, 'c']
```

fill方法还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置。

注意，如果填充的类型为对象，那么被赋值的是同一个内存地址的对象，而不是深拷贝对象。

```
let arr = new Array(3).fill({name: "Mike"});
arr[0].name = "Ben";
arr
// [{name: "Ben"}, {name: "Ben"}, {name: "Ben"}]

let arr = new Array(3).fill([]);
arr[0].push(5);
arr
// [[5], [5], [5]]
```

数组的拓展之扩展方法entries()，keys() 和 values()

ES6 提供三个新的方法—entries()，keys()和values()—用于遍历数组。可以用for...of循环进行遍历，唯一的区别是keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。()

```
for (let index of ['a', 'b'].keys()) {
    console.log(index);
}
// 0
// 1

for (let elem of ['a', 'b'].values()) {
    console.log(elem);
}
// 'a'
// 'b'

for (let [index, elem] of ['a', 'b'].entries()) {
    console.log(index, elem);
}
  // 0 "a"
  // 1 "b"
```

数组的拓展之扩展方法includes

Array.prototype.includes方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似

```
[1, 2, 3].includes(2)     // true
[1, 2, 3].includes(4)     // false
[1, 2, NaN].includes(NaN) // true
```

该方法的第二个参数表示搜索的起始位置，默认为0。如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从0开始。

数组的indexOf方法与includes方法的对比：

indexOf方法有两个缺点：

一是不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于-1，表达起来不够直观。

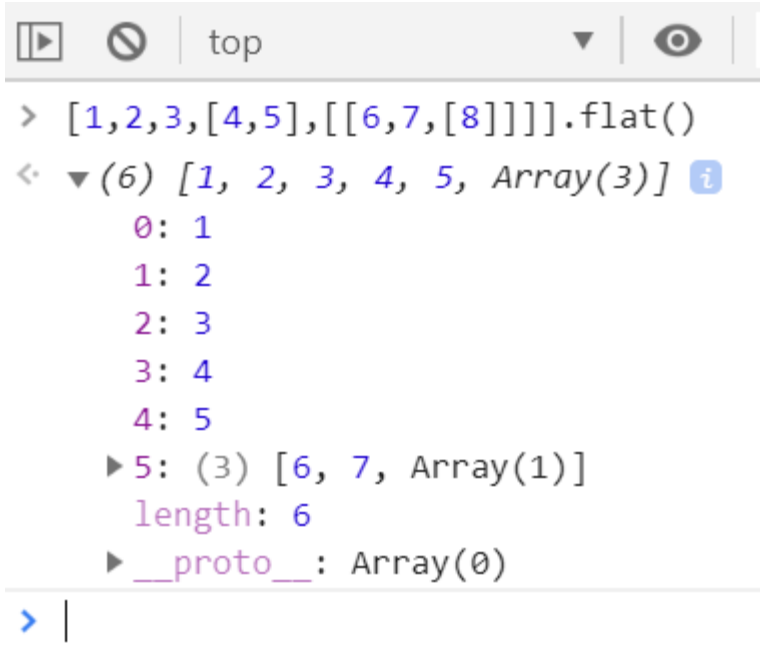
二是，它内部使用严格相等运算符（===）进行判断，这会导致对NaN的误判。

```
[NaN].indexOf(NaN)
// -1
```

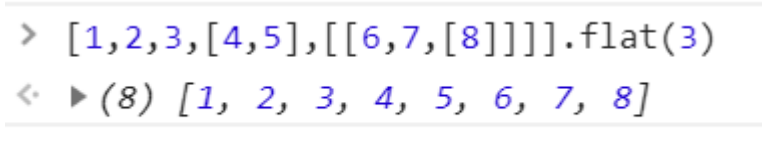
```
[NaN].includes(NaN)
// true
```

数组的拓展之扩展方法flat(),flatMap()

数组的成员有时还是数组，Array.prototype.flat()用于将嵌套的数组“拉平”，变成一维的数组。**该方法返回一个新数组，对原数据没有影响**



flat()默认只会“拉平”一层，如果想要“拉平”多层的嵌套数组，可以将flat()方法的参数写成一个整数，表示想要拉平的层数，默认为1。



“拉平”3层

```
[1, [2, [3]]].flat(Infinity)
// [1, 2, 3]
```

如果不管有多少层嵌套，都要转成一维数组，可以用Infinity关键字作为参数。

```
[1, 2, , 4, 5].flat()
// [1, 2, 4, 5]
```

如果原数组有空位，flat()方法会跳过空位。

数组的拓展之扩展方法flat(),flatMap()

flatMap()方法先对原数组的每个成员执行一个函数（相当于执行Array.prototype.map()），然后对**返回值组成的数组**执行flat()方法。**该方法返回一个新数组，不改变原数组。**

```
// 相当于 [[2, 4], [3, 6], [4, 8]].flat()
[2, 3, 4].flatMap((x) => [x, x * 2])
// [2, 4, 3, 6, 4, 8]
```

flatMap()只能展开一层数组。

```
// 相当于 [[[2]], [[4]], [[6]], [[8]]].flat()
[1, 2, 3, 4].flatMap(x => [[x * 2]])
// [[2], [4], [6], [8]]
```

flatMap()方法的参数是一个遍历函数，该函数可以接受三个参数，分别是数组数组、数组项目的位置（从零开始）、数组。(和map一样一样的)

数组的空位处理

数组的空位指，数组的某一个位置没有任何值。比如，Array构造函数返回的一个只有长的数组，里面自然都是空位。

```
> let a=new Array(3)
< undefined
> a
< ▶ (3) [empty × 3]
> let b=[undefined,undefined,undefined]
< undefined
> 0 in b
< true
> 0 in a
< false
>
```

注意，空位不是undefined，一个位置的值等于undefined，依然是有值的。空位是没有任何值，in运算符可以说明这一点。（其实严格来说，连下标都木得）

```
> [,,,]
< ▶ (3) [empty × 3]
> [1,,2]
< ▶ (3) [1, empty, 2]
>
```

（这种情况下也会产生空位）

数组的空位处理

ES5 对空位的处理，已经很不一致了，大多数情况下会忽略空位。

- forEach(), filter(), reduce(), every() 和some()都会跳过空位。
- map()会跳过空位，但会保留这个值
- join()和toString()会将空位视为undefined，而undefined和null会被处理成空字符串。

```
// forEach方法
[,'a'].forEach((x,i) => console.log(i)); // 1

// filter方法
['a',,'b'].filter(x => true) // ['a','b']

// every方法
[,'a'].every(x => x==='a') // true

// reduce方法
[1,,2].reduce((x,y) => x+y) // 3

// some方法
[,'a'].some(x => x !== 'a') // false

// map方法
[,'a'].map(x => 1) // [,1]

// join方法
[,'a',undefined,null].join('#') // "#a##"

// toString方法
[,'a',undefined,null].toString() // ",a,,"
```

由于空位的处理规则非常不统一，所以建议避免出现空位

数组的空位处理

ES6 则是明确将空位转为undefined

Array.from方法会将数组的空位，转为undefined，也就是说，这个方法不会忽略空位。

```
Array.from(['a',,'b'])
// [ "a", undefined, "b" ]
```

扩展运算符（...）也会将空位转为undefined。

```
[...['a',,'b']]
// [ "a", undefined, "b" ]
```

copyWithin()会连空位一起拷贝。

```
> [,'a','b',1].copyWithin(2,0) ;
< ▶ (4) [empty, "a", empty, "a"]
>
```

for...of循环也会遍历空位。

```
let arr = [, ,];
for (let i of arr) {
  console.log(1);
}
// 1
// 1
```

entries()、keys()、values()、find()和findIndex()会将空位处理成undefined。

symbol数据类型

symbol数据类型

ES5 的对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（混合 模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，保证每个属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是 ES6 引入Symbol的原因。

```
let o={x:1,y:2,z:3};

//我们此时得到了一个全局作用域下的对象o
//现在我们想给它加一个名为x的方法方便后期使用

o.x=function (){//加上该方法之后，原来对象里面的x就完全被覆盖了
    console.log("hello");
}
```

symbol数据类型

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，前六种是：undefined、null、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）。

Symbol 值通过Symbol函数生成。

```
> let s=Symbol()
< undefined
> s
< Symbol()
> typeof s
< "symbol"
>
```

这就是说，对象的属性名现在可以有两种类型：

1. 一种是原来就有的字符串；
2. 一种就是新增的 Symbol 类型。

凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

注意，Symbol函数前**不能使用new命令**，否则会报错。这是因为生成的 Symbol 是一个原始类型的值，不是对象。基本上，它是一种类似于字符串的数据类型。

symbol数据类型

Symbol函数可以接受一个字符串(如果传入的是其他类型的数据，那就先转化为字符串)作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。

```
> let s1=Symbol("hello")
<· undefined
> let s2=Symbol("world")
<· undefined
> s1
<· Symbol(hello)
> s2
<· Symbol(world)
> s2.toString()
<· "Symbol(world)"
> s1.toString()
<· "Symbol(hello)"
>
```

上面代码中，s1和s2是两个 Symbol 值。如果不加参数，它们在控制台的输出都是Symbol()，不利于区分。有了参数以后，就等于为它们加上了描述，输出的时候就能够分清，到底是哪一个值。

symbol数据类型

如果传入Symbol函数的是其他类型的数据,那就先转化为字符串(注意当传入的是undefined时，就不会有任何反应，这道理和简单，对于函数来说，执行时没有传入的参数,它得到的就是undefined,你传一个undefined就没啥区别了)

```
// 没有参数的情况
let s1 = Symbol();
let s2 = Symbol();

s1 === s2 // false

// 有参数的情况
let s1 = Symbol('foo');
let s2 = Symbol('foo');

s1 === s2 // false
```

注意，Symbol函数的**参数只是表示对当前 Symbol 值的描述**，因此相同参数的Symbol函数的返回值是不相等的。

s1和s2都是Symbol函数的返回值，而且参数相同，但是它们是不相等的。

```
> let s1=Symbol({x:1});
< undefined
> s1
< Symbol([object Object])
> let s2=Symbol([1,2,3,4]);
< undefined
> s2
< Symbol(1,2,3,4)
> let s3=Symbol(undefined);
< undefined
> s3
< Symbol()
> let s4=Symbol(true);
< undefined
> s4
< Symbol(true)
> let s5=Symbol(null);
< undefined
> s5
< Symbol(null)
> let s6=Symbol(4);
< undefined
> s6
< Symbol(4)
> let s7=Symbol("hello");
< undefined
> s7
< Symbol(hello)
>
```

symbol数据类型

```
let sym = Symbol('My symbol');

"your symbol is " + sym
// TypeError: can't convert symbol to string
`your symbol is ${sym}`
// TypeError: can't convert symbol to string
```

Symbol 值不能与其他类型的值进行运算，会报错。

```
let sym = Symbol('My symbol');

String(sym) // 'Symbol(My symbol)'
sym.toString() // 'Symbol(My symbol)'
```

但是，Symbol 值可以显式转为字符串。

```
let sym = Symbol();
Boolean(sym) // true
!sym  // false

if (sym) {
  // ...
}

Number(sym) // TypeError
sym + 2 // TypeError
```

另外，Symbol 值也可以转为布尔值，但是不能转为数值。

symbol数据类型之描述字符

创建 Symbol 的时候，可以添加一个描述。

```
const sym = Symbol('foo');

String(sym) // "Symbol(foo)"
sym.toString() // "Symbol(foo)"
```

左侧代码中，sym的描述就是字符串foo。

但是，读取这个描述需要将 Symbol 显式转为字符串，即下面的写法。

上面的用法不是很方便。ES2019 提供了一个实例属性description，直接返回 Symbol 的描述。

```
> let sym= Symbol("万章")
<- undefined
> sym.description
<- "万章"
>
```

symbol数据类型之属性名称的应用

由于每一个 Symbol 值都是不相等的，这意味着 Symbol 值可以作为标识符，用于对象的属性名，就能保证不会出现同名的属性

```
let mySymbol = Symbol();

// 第一种写法
let a = {};
a[mySymbol] = 'Hello!';

// 第二种写法
let a = {
  [mySymbol]: 'Hello!'
};

// 第三种写法
let a = {};
Object.defineProperty(a, mySymbol, { value: 'Hello!' });

// 以上写法都得到同样结果
a[mySymbol] // "Hello!"
```

```
> const mySymbol = Symbol();
  const a = {};

  a.mySymbol = 'Hello!';
< "Hello!"
> a.mySymbol
< "Hello!"
> a
< ▼ {mySymbol: "Hello!"} i
    mySymbol: "Hello!"
  ▶ __proto__: Object
> a[mySymbol]
< undefined
> a['mySymbol']
< "Hello!"
```

注意，Symbol 值作为对象属性名时，不能用点运算符。因为点运算符后面总是字符串，所以不会读取mySymbol作为标识名所指代的那个值

symbol数据类型之不等常量定义

类型还可以用于定义一组常量，保证这组常量的值都是不相等的。

```
const COLOR_RED    = Symbol();
const COLOR_GREEN  = Symbol();

function getComplement(color) {
  switch (color) {
    case COLOR_RED:
      return COLOR_GREEN;
    case COLOR_GREEN:
      return COLOR_RED;
    default:
      throw new Error('Undefined color');
    }
}
```

常量使用 Symbol 值最大的好处，就是其他任何值都不可能有相同的值了，因此可以保证上面的switch语句会按设计的方式工作。(有点密码加密的效果, 一次定义, 永久都是独一无二的)

symbol数据类型之魔术字符串

魔术字符串指的是，在代码之中多次出现、与代码形成强耦合的某一个具体的字符串或者数值。风格良好的代码，应该尽量消除魔术字符串，改由含义清晰的变量代替。

```
oImgBox.addEventListener("click",function(ev){
    switch(ev.target.parentNode.classList[0]){
        case "list1":
            prePic();
            break;
        case "list3":
            nextPic();
            break;
    }
})
```

这一块的代码和这两个字符串紧紧的耦合在一起

```
let goalEle={
    prePicEle:"list1",
    nextPicEle:"list3"
}

oImgBox.addEventListener("click",function(ev){
    switch(ev.target.parentNode.classList[0]){
        case goalEle.prePicEle:
            prePic();
            break;
        case goalEle.nextPicEle:
            nextPic();
            break;
    }
})
```

常用的消除魔术字符串的方法，就是把它写成一个变量。

仔细分析就可以发现goalEle里面每个属性的值等于哪个值并不重要，只要确保不会跟其他属性的值冲突即可。因此，这里就很适合改用 Symbol 值。