写在开头

- `ES6` **常用但被忽略的方法** 系列文章，整理作者认为一些日常开发可能会用到的一些方法、使用技巧和一些应用场景，细节深入请查看相关内容连接，欢迎补充交流。

相关文章

- ES6常用但被忽略的方法（第一弹解构赋值和数值）
- ES6常用但被忽略的方法（第二弹函数、数组和对象）
- ES6常用但被忽略的方法（第三弹Symbol、Set 和 Map ）
- ES6常用但被忽略的方法（第四弹Proxy和Reflect）
- ES6常用但被忽略的方法（第五弹Promise和Iterator）
- ES6常用但被忽略的方法（第六弹Generator ）
- ES6常用但被忽略的方法（第七弹async）
- ES6常用但被忽略的方法（第八弹Class）
- ES6常用但被忽略的方法（第九弹Module）
- ES6常用但被忽略的方法（第十弹项目开发规范）
- ES6常用但被忽略的方法（第十一弹Decorator）

写在开头

- `ES6` **常用但被忽略的方法** 系列文章，整理作者认为一些日常开发可能会用到的一些方法、使用技巧和一些应用场景，细节深入请查看相关内容连接，欢迎补充交流。

相关文章

- ES6常用但被忽略的方法（第二弹函数、数组和对象）
- ES6常用但被忽略的方法（第三弹Symbol、Set 和 Map ）
- ES6常用但被忽略的方法（第四弹Proxy和Reflect）
- ES6常用但被忽略的方法（第五弹Promise和Iterator）
- ES6常用但被忽略的方法（第六弹Generator ）
- ES6常用但被忽略的方法（第七弹async）
- ES6常用但被忽略的方法（第八弹Class）
- ES6常用但被忽略的方法（第九弹Module）
- ES6常用但被忽略的方法（第十弹项目开发规范）
- ES6常用但被忽略的方法（第十一弹Decorator）
- ES6常用但被忽略的方法（终弹-最新提案）

解构赋值

- ES6-变量的解构赋值

剔除对象不需要的属性的值

```
const obj = {
    name: 'detanx',
    age: 24,
    height: 180
}
// 剔除height
const { height, ...otherObj } = obj;
// otherObj => {name: 'detanx', age: 24}
复制代码
```

剔除数组不需要的项

```
const arr = ['detanx', 24, 180]
// 剔除 第一项
const [ name, ...otherArr ] = arr;
// otherArr => [24, 180]
复制代码
```

设置默认值

- `ES6` 内部使用严格相等运算符（`===`），判断一个位置是否有值。所以，只有当一个数组成员严格等于`undefined`，默认值才会生效。（**面试可能会遇到**）

```
const [x = 1] = [undefined];
// x 1
const [x = 1] = [null];
// x null
复制代码
```

- 可以引用解构赋值的其他变量，但该变量必须已经声明。

```
let [x = 1, y = x] = [];     // x=1; y=1
let [x = 1, y = x] = [2];    // x=2; y=2
let [x = 1, y = x] = [1, 2]; // x=1; y=2
let [x = y, y = 1] = [];     // ReferenceError: y is not defined
复制代码
```

不需要额外变量就可交换两个变量的值

```
let x = 1, y = 2;
[x, y] = [y, x]
// x: 2,y: 1
复制代码
```

获取指定的值

```
// 返回一个数组
function example() {
  return [1, 2, 3];
}
let [a, b, c] = example();

// 返回一个对象
```

```
function example() {
  return {
    foo: 1,
    bar: 2
  };
}
let { foo, bar } = example();
// 导入模块某些方法
import { clone } from '_lodash';
复制代码
```

字符串

- ES6-字符串新增方法

`includes`

- 之前判断目标字符或字符串片段是否存在某个字符串中需要使用`indexOf`来判断，返回的结果是首次出现目标的位置，不存在返回`-1`。实际场景我们并不需要回去目标在字符串中的位置，只需要判断目标是否该字符串中即可，所以我们可以使用`includes`。同时可以接收第二个参数，表示从左到右第几个位置开始

```
'detanx'.includes('tan') // true
'detanx'.includes('tan',3) // false
复制代码
```

查询目标字符在字符串开始或结尾

- `str.startsWith(char[, pos])`查询`char`是否是字符串开头，`pos`代表从左到右哪一个位置算起始位置。

```
'detanx'.startsWith('tan', 2) // true
复制代码
```

- `str.endsWith(char[, len])`查询`char`是否是字符串，`len`代表从左到右包含多少个字符（即长度）。

```
'detanx'.endsWith('tan', 5) // true
复制代码
```

字符串重复多少次`repeat`

- 重复字符串多少次，接收一个参数
    1. 当参数为大于`-1`并且小于`1`或者为`NaN`时作为`0`
    2. 大于`1`且为小数时向下取整
    3. 为`infinity`或者小于`-1`时会报错
    4. 其他类型会转为数字

```
'detanx'.repeat(0) // ""
'detanx'.repeat(-0.7) // ""
'detanx'.repeat(NaN) // ""
'detanx'.repeat(2) // "detanxdetanx"
'detanx'.repeat({}) // ""
'detanx'.repeat([]) // ""
'detanx'.repeat(()=>{}) // ""
复制代码
```

字符串补全

- 从左边补全`padStart`，当字符串长度不够需要指定以什么开头，不传第二参数默认使用空格。

```
// 时间显示小于10补0
const x = 7
if(x < 10) String(x).padStart(2,'0')
// 替换结尾
'04'.padStart(10, 'YYYY-MM-DD'); // 'YYYY-MM-04'
复制代码
```

- 从右边补全`padEnd`，当字符串长度不够需要指定以什么结尾，不传第二参数默认使用空格。

```
'de'.padEnd(6, 'tanx'); // detanx
复制代码
```

去掉头或尾空格

- 除了空格键，这两个方法对字符串头部（或尾部）的`tab`键、换行符等不可见的空白符号也有效。
- 浏览器还部署了额外的两个方法，`trimLeft()`是`trimStart()`的别名，`trimRight()`是`trimEnd()`的别名。

```
// 去掉头部空格
'  detanx  '.trimStart(); // 'detanx  '
'  detanx  '.trimLeft(); // 'detanx  '
// 去掉尾部空格
'  detanx  '.trimEnd(); // '  detanx'
'  detanx  '.trimRight(); // '  detanx'
复制代码
```

数值扩展

- ES6-数值扩展

isFinite

- 用来检查一个数值是否为有限的（`finite`），即不是`Infinity`或者`-Infinity`。其他类型直接返回`false`，包括`NaN`。

```
Number.isFinite(15); // true
Number.isFinite(0.8); // true
Number.isFinite(NaN); // false
Number.isFinite(Infinity); // false
Number.isFinite(-Infinity); // false
Number.isFinite('foo'); // false
Number.isFinite('15'); // false
Number.isFinite(true); // false
复制代码
```

isInteger

- 判断一个数值是否为整数。如果对数据精度的要求较高，不建议使用`Number.isInteger()`判断一个数值是否为整数。
- `JavaScript`采用`IEEE 754`标准，数值存储为`64`位双精度格式，数值精度最多可以达到`53`个二进制位（`1`个隐藏位与`52`个有效位）。如果数值的精度超过这个限度，第54位及后面的位就会被丢弃，这种情况下，`Number.isInteger`可能会误判。

```
Number.isInteger(3.0000000000000002) // true
复制代码
```

- 这个小数的精度达到了小数点后`16`个十进制位，转成二进制位超过了`53`个二进制位，导致最后的那个`2`被丢弃。
- 如果一个数值的绝对值小于`Number.MIN_VALUE（5E-324）`，即小于`JavaScript`能够分辨的最小值，会被自动转为`0`，`Number.isInteger`也会误判。

```
Number.isInteger(5E-324) // false
Number.isInteger(5E-325) // true
复制代码
```

isSafeInteger

- `Number.isSafeInteger()`是用来判断一个整数是否落在`Number.MAX_SAFE_INTEGER`和`Number.MIN_SAFE_INTEGER`这两个常量范围之内。

```
Number.isSafeInteger('a') // false
Number.isSafeInteger(null) // false
Number.isSafeInteger(NaN) // false
Number.isSafeInteger(Infinity) // false
Number.isSafeInteger(-Infinity) // false

Number.isSafeInteger(3) // true
Number.isSafeInteger(1.2) // false
Number.isSafeInteger(9007199254740990) // true
Number.isSafeInteger(9007199254740992) // false

Number.isSafeInteger(Number.MIN_SAFE_INTEGER - 1) // false
Number.isSafeInteger(Number.MIN_SAFE_INTEGER) // true
Number.isSafeInteger(Number.MAX_SAFE_INTEGER) // true
Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1) // false
复制代码
```

- 实际使用这个函数时，需要注意。验证运算结果是否落在安全整数的范围内，不要只验证运算结果，而要同时验证参与运算的每个值。

Math.trunc

- `Math.trunc`方法用于去除一个数的小数部分，返回整数部分。非数值，`Math.trunc`内部使用`Number`方法将其先转为数值。空值和无法截取整数的值，返回`NaN`。

```
Math.trunc(4.9) // 4
Math.trunc(-4.1) // -4
Math.trunc(-0.1234) // -0

Math.trunc('123.456') // 123
Math.trunc(true) //1
Math.trunc(false) // 0
Math.trunc(null) // 0

Math.trunc(NaN);      // NaN
Math.trunc('foo');    // NaN
Math.trunc();         // NaN
Math.trunc(undefined) // NaN
复制代码
```

- 模拟实现

```
Math.trunc = Math.trunc || function(x) {
  return x < 0 ? Math.ceil(x) : Math.floor(x);
};
复制代码
```

Math.sign

- `Math.sign`

  方法用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

  1. 参数为正数，返回+1；
  2. 参数为负数，返回-1；
  3. 参数为 0，返回0；
  4. 参数为-0，返回-0;
  5. 其他值，返回NaN。

```
Math.sign(-5) // -1
Math.sign(5) // +1
Math.sign(0) // +0
Math.sign(-0) // -0
Math.sign(NaN) // NaN
复制代码
```

- 如果参数是非数值，会自动转为数值。对于那些无法转为数值的值，会返回`NaN`。

```
Math.sign('')  // 0
Math.sign(true)  // +1
Math.sign(false)  // 0
Math.sign(null)  // 0
Math.sign('9')  // +1
Math.sign('foo')  // NaN
Math.sign()  // NaN
Math.sign(undefined)  // NaN
复制代码
```

- 模拟实现

```
Math.sign = Math.sign || function(x) {
  x = +x; // convert to a number
  if (x === 0 || isNaN(x)) {
    return x;
  }
  return x > 0 ? 1 : -1;
};
复制代码
```

Math.cbrt

- Math.cbrt()方法用于计算一个数的立方根。

```
Math.cbrt(-1) // -1
Math.cbrt(0)  // 0
Math.cbrt(1)  // 1
Math.cbrt(2)  // 1.2599210498948732
复制代码
```

- 对于非数值，Math.cbrt()方法内部也是先使用Number()方法将其转为数值。

```
Math.cbrt('8') // 2
Math.cbrt('hello') // NaN
复制代码
```

- 模拟实现

```
Math.cbrt = Math.cbrt || function(x) {
  var y = Math.pow(Math.abs(x), 1/3);
  return x < 0 ? -y : y;
};
复制代码
```

Math.hypot

- Math.hypot方法返回所有参数的平方和的平方根。

```
Math.hypot(3, 4);        // 5
Math.hypot(3, 4, 5);     // 7.0710678118654755
Math.hypot();            // 0
Math.hypot(NaN);         // NaN
Math.hypot(3, 4, 'foo'); // NaN
Math.hypot(3, 4, '5');   // 7.0710678118654755
Math.hypot(-3);          // 3
复制代码
```

- 上面代码中，`3` 的平方加上 `4` 的平方，等于 `5` 的平方。
- 如果参数不是数值，`Math.hypot`方法会将其转为数值。只要有一个参数无法转为数值，就会返回`NaN`。

函数扩展

- ES6-函数扩展

函数默认值

1. 函数可以通过定义时给对应的参数赋一个默认值，默认值当函数调用时如果没有传入对应的值便会使用默认值。设置默认值还可以搭配解构赋值一起使用。

```
function detanx(x = 1) { ... }
// 搭配解构赋值
function detanx({name, age = 10} = {name: 'detanx', age: 10}) {...}
// 不完全解构，
function detanx({name, age = 10} = {name: 'detanx'}) {...}
detanx({age: 12}) // undefined 12
复制代码
```

- **不完全解构时，即使后面的对象中有对应的键值也不会被赋值。**

1. 函数有默认值的参数在参数个数中间，并且后面的参数不实用默认值，这时要使用默认值的参数可以传入`undefined`。

```
function detanx(x =  null, y = 6) {
  console.log(x, y);
}
detanx(undefined, null); // null null
复制代码
```

1. 默认值会改变`length`获取函数参数返回的长度。

- 指定了默认值以后，函数的`length`属性，将返回没有指定默认值的参数个数。也就是说，指定了默认值后，`length`属性将失真。

```
(function (a) {}).length // 1
(function (a = 5) {}).length // 0
(function (a, b, c = 5) {}).length // 2
复制代码
```

- **默认值的参数不是尾参数，那么`length`属性也不再计入后面的参数。**

```
(function (a = 0, b, c) {}).length // 0
(function (a, b = 1, c) {}).length // 1
复制代码
```

- 使用扩展运算符表示接收的参数，`length`为`0`.

```
(function(...args) {}).length // 0
复制代码
```

1. 设置默认值后作用域改变

- 设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域（`context`）。

```
var x = 1;
function detanx(x, y = x) {
  console.log(y);
}
f(2) // 2

function detanx(x = x, y = x) {
  console.log(x, y);
}
f(undefined,2) //  ReferenceError: Cannot access 'x' before initialization
复制代码
```

函数默认值应用

1. 指定某一个参数不得省略，如果省略就抛出一个错误。

```
function throwIfMissing() {
  throw new Error('Missing parameter');
}
function foo(mustBeProvided = throwIfMissing()) {
  return mustBeProvided;
}
foo()
// Error: Missing parameter
复制代码
```

- 参数`mustBeProvided`的默认值等于`throwIfMissing`函数的运行结果（注意函数名`throwIfMissing`之后有一对圆括号），这表明参数的默认值不是在定义时执行，而是在运行时执行。如果参数已经赋值，默认值中的函数就不会运行。

严格模式变化

- `ES5`开始，函数内部可以设定为严格模式。`ES2016`做了一点修改，规定只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错。

```
// 报错
function doSomething(a, b = a) {
  'use strict';
  // code
}
// 报错
```

```
const doSomething = function ({a, b}) {
  'use strict';
};
// 报错
const doSomething = (...a) => {
  'use strict';
};
const obj = {
  // 报错
  doSomething({a, b}) {
    'use strict';
  }
};
复制代码
```

- 这种限制有个问题就是在声明时都是先执行参数再执行函数体，只有在进入函数体发现用了严格模式才会报错。
- 规避这种限制的方法。

```
// 第一种是设定全局性的严格模式，这是合法的。
'use strict';
function doSomething(a, b = a) {
  // code
}
// 第二种是把函数包在一个无参数的立即执行函数里面。
const doSomething = (function () {
  'use strict';
  return function(value = 42) {
    return value;
  };
}());
复制代码
```

箭头函数

- 使用注意点
    1. 函数体内的`this`对象，就是定义时所在的对象，而不是使用时所在的对象。
    2. 不可以当作构造函数，也就是说，不可以使用`new`命令，否则会抛出一个错误。
    3. 不可以使用`arguments`对象，该对象在函数体内不存在。如果要用，可以用`rest`参数代替。
    4. 不可以使用`yield`命令，因此箭头函数不能用作`Generator`函数。
- 箭头函数也可以像普通函数一样嵌套使用。
- 简化编写代码，但会降低代码的可读性。

```
// 柯里化
const curry = (fn, arr = []) => (...args) => (
  arg => arg.length === fn.length
    ? fn(...arg)
    : curry(fn, arg)
)([...arr, ...args]);

let curryTest = curry( (a, b) => a + b );
curryTest(1, 2) //返回3
curryTest(1)(2) //返回3
复制代码
```

数组扩展

- ES6-数组扩展

扩展运算符

- 应用

1. 接收不确定参数数量

```
function add(...values) {
  let sum = 0;
  for (var val of values) {
    sum += val;
  }
  return sum;
}
add(2, 5, 3) // 10
复制代码
```

1. 替代`apply`方法
   - 之前在某个函数需要接收数组参数时需要这样写`fn.apply(this, arr)`。

```
// ES5 的写法
const args = [0, 1, 2];
function f(x, y, z) {
  // ...
}
f.apply(null, args);

// ES6的写法
function f(x, y, z) {
  // ...
}
f(...args);
复制代码
```

1. 复制、合并数组、转换字符串为数组

```
const a = [1]
const b = [2];
// 复制数组
const c = [...a]; // [1]
// 合并数组
[...a, ...b, ...c] // [1, 2, 1]
// 转换字符串为数组
[...'detanx'] // ['d', 'e', 't', 'a', 'n', 'x']
复制代码
```

1. 实现了 `Iterator` 接口的对象

- 任何定义了遍历器（`Iterator`）接口的对象（参阅 `Iterator` 一章），都可以用扩展运算符转为真正的数组。

```
Number.prototype[Symbol.iterator] = function*() {
  let i = 0;
  let num = this.valueOf();
  while (i < num) {
    yield i++;
  }
}
console.log([...5]) // [0, 1, 2, 3, 4]
复制代码
```

- 例如函数的`arguments`对象是一个类数组；`querySelectorAll`方法返回的是一个`NodeList`对象。它也是一个类似数组的对象。

```
let nodeList = document.querySelectorAll('div');
let array = [...nodeList];
复制代码
```

Array.from

- `Array.from`的作用和扩展运算符类似，都可以将类数组转为数组，但它还可以接收第二参数，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。

```
// 生成0-99的数组
Array.from(new Array(100), (x, i) => i);
复制代码
```

copyWithin()

- 数组实例的

```
copyWithin()
```

方法，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。

```
Array.prototype.copyWithin(target, start = 0, end = this.length)
复制代码
```

- target（必需）：从该位置开始替换数据。如果为负值，表示倒数。

    - 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。
    - 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。
- **这三个参数都应该是数值，如果不是，会自动转为数值。**

```
// 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}

// 将2号位到数组结束，复制到0号位
let i32a = new Int32Array([1, 2, 3, 4, 5]);
i32a.copyWithin(0, 2);
// Int32Array [3, 4, 5, 4, 5]

// 对于没有部署 TypedArray 的 copyWithin 方法的平台
// 需要采用下面的写法
[].copyWithin.call(new Int32Array([1, 2, 3, 4, 5]), 0, 3, 4);
// Int32Array [4, 2, 3, 4, 5]
复制代码
```

find() 和 findIndex()

- 两个方法都可以接收两个参数，第一个参数是回调函数（可以接收三个参数，分别是当前值，当前位置，当前的原数组），第二个参数用来绑定回调函数的`this`。区别是`find`没有找到返回的是`undefined`，`findIndex`返回的是`-1`。这两个方法都可以发现`NaN`，弥补了数组的`indexOf`方法的不足。

```
[1, 4, -5, 10].find((n) => n > 10) // undefined
[1, 5, 10, 15].findIndex(function(value, index, arr) {
  return value > 15;
}) // -1

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

// 找出NaN
[NaN].indexOf(NaN)
// -1
[NaN].findIndex(y => Object.is(NaN, y))
// 0
复制代码
```

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]
复制代码
```

- 上面代码表明，`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。
- `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。

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

- 上面代码表示，`fill`方法从`1`号位开始，向原数组填充`7`，到`2`号位之前结束。

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]]
复制代码
```

includes

- 某个数组是否包含给定的值，与字符串的`includes`方法类似。与`indexOf`的区别是`includes`直接返回一个布尔值，`indexOf`需要再判断返回值是否为`-1`。当我们需要一个变量是否为多个枚举值之一时就可以使用`includes`。

```
const name = 'detanx';
const nameArr = ['detanx', 'det', 'tanx']
// 使用includes做判断
if(nameArr.indexOf(name) !== -1) { ... }
=>
if(nameArr.includes(name)) { ... }
// 判断name是否是 detanx，tanx，det
if(name === 'detanx' || name === 'det' || name === 'tanx') {...}
=>
nameArr.includes(name);
复制代码
```

数组空位

- 数组的空位指：数组的某一个位置没有任何值（空位不是`undefined`，一个位置的值等于`undefined`，依然是有值的。）。比如，`Array`构造函数返回的数组都是空位。

```
Array(3) // [, , ,]
复制代码
```

1. `forEach()`、`filter()`、`reduce()`、`every()`和`some()`都会跳过空位。

```
// 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
复制代码
```

1. `map()`会跳过空位，但会保留这个值。

```
// map方法
[,'a'].map(x => 1) // [,1]
复制代码
```

1. `join()`和`toString()`会将空位视为`undefined`，而`undefined`和`null`会被处理成空字符串。

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

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

1. `Array.from()`、`...`、`entries()`、`keys()`、`values()`、`find()`和`findIndex()`会将空位处理成`undefined`。

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

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

// entries()
[...[,'a'].entries()] // [[0,undefined], [1,"a"]]

// keys()
[...[,'a'].keys()] // [0,1]

// values()
[...[,'a'].values()] // [undefined,"a"]
```

```
// find()
[,'a'].find(x => true) // undefined

// findIndex()
[,'a'].findIndex(x => true) // 0
```
复制代码

1. copyWithin()、fill()、for...of会将空位视为正常的数组位置。

```
[,'a','b',,].copyWithin(2,0) // [,"a",,"a"]

new Array(3).fill('a') // ["a","a","a"]

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

- **空位的处理规则非常不统一，所以建议避免出现空位。**

对象扩展

- ES6-对象扩展
- ES6-对象新增方法

对象属性枚举

- 对象的每个属性都有一个描述对象（`Descriptor`），用来控制该属性的行为。`Object.getOwnPropertyDescriptor`方法可以获取该属性的描述对象。

```
let obj = { foo: 123 };
Object.getOwnPropertyDescriptor(obj, 'foo')
//  {
//    value: 123,
//    writable: true,
//    enumerable: true,
//    configurable: true
//  }
```
复制代码

- 描述对象的`enumerable`属性，称为“可枚举性”，如果该属性为`false`，就表示某些操作会忽略当前属性。
- 目前，有四个操作会忽略`enumerable`为`false`的属性。
    1. `for...in`循环：只遍历对象自身的和继承的可枚举的属性。
    2. `Object.keys()`：返回对象自身的所有可枚举的属性的键名。
    3. `JSON.stringify()`：只串行化对象自身的可枚举的属性。
    4. `Object.assign()`： 忽略`enumerable`为`false`的属性，只拷贝对象自身的可枚举的属性。
- **`Symbol`类型的属性只能通过`getOwnPropertySymbols`方法获取。**

链判断运算符

- `ES5` 我们判断一个深层级的对象是否有某一个 `key` 需要一层一层去判断，现在我们可以通过 `?.` 的方式去获取。

```
// es5
// 错误的写法（当某一个key不存在undefined.key就会代码报错）
const  firstName = message.body.user.firstName;

// 正确的写法
const firstName = (message
  && message.body
  && message.body.user
  && message.body.user.firstName) || 'default';

// es6
const firstName = message?.body?.user?.firstName || 'default';
复制代码
```

- 链判断运算符有三种用法。
    1. `obj?.prop` // 对象属性
    2. `obj?.[expr]` // 对象属性
    3. `func?.(...args)` // 函数或对象方法的调用

```
a?.b
// 等同于
a == null ? undefined : a.b

a?.[x]
// 等同于
a == null ? undefined : a[x]

a?.b()
// 等同于
a == null ? undefined : a.b()

a?.()
// 等同于
a == null ? undefined : a()
复制代码
```

- 注意点
    1. 短路机制
    - `?.` 运算符相当于一种短路机制，只要不满足条件，就不再往下执行。

    ```
    // 当name不存在时，后面的就不再执行
    user?.name?.firstName
    复制代码
    ```

    1. delete 运算符

    ```
    delete a?.b
    // 等同于
    a == null ? undefined : delete a.b
    复制代码
    ```

- 上面代码中，如果`a`是`undefined`或`null`，会直接返回`undefined`，而不会进行`delete`运算。

1. 括号的影响

- 如果属性链有圆括号，链判断运算符对圆括号外部没有影响，只对圆括号内部有影响。

```
(a?.b).c
// 等价于
(a == null ? undefined : a.b).c
复制代码
```

1. 报错场合

- 以下写法是禁止的，会报错。

```
// 构造函数
new a?.()
new a?.b()

// 链判断运算符的右侧有模板字符串
a?.`{b}`
a?.b`{c}`

// 链判断运算符的左侧是 super
super?.()
super?.foo

// 链运算符用于赋值运算符左侧
a?.b = c
复制代码
```

1. 右侧不得为十进制数值

- 为了保证兼容以前的代码，允许`foo?.3:0`被解析成`foo ? .3 : 0`，因此规定如果`?.`后面紧跟一个十进制数字，那么`?.`不再被看成是一个完整的运算符，而会按照三元运算符进行处理，也就是说，那个小数点会归属于后面的十进制数字，形成一个小数。

`Null`运算判断符`??`

- 读取对象属性的时候，如果某个属性的值是`null`或`undefined`，有时候需要为它们指定默认值。常见做法是通过`||`运算符指定默认值。
- **`||`运算符当左边为空字符串或者`false`的时候也会设置默认值，`??`运算符只有当左边是`null`或`undefined`才会设置默认值。**

```
// || 运算
const a = '';
b = a || 1;
console.log(b) // 1

// ?? 运算
b = a ?? 1;
console.log(b) // ''
复制代码
```

- **`??`有一个运算优先级问题，它与`&&`和`||`的优先级孰高孰低，所以在一起使用时必须用括号表明优先级，否则会报错。**

Object.is()

- 用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。

```
Object.is('foo', 'foo')
// true
Object.is({}, {})
// false
复制代码
```

- 不同之处只有两个：一是+0不等于-0，二是NaN等于自身。

```
+0 === -0 //true
NaN === NaN // false

Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
复制代码
```

Object.assign

- 用于对象的合并，将源对象（`source`）的所有自身属性（不拷贝继承属性）、自身的`Symbol`值属性和可枚举属性（属性的`enumerable`为`true`），复制到目标对象（`target`）。第一个参数为目标对象，后面的都是源对象。

```
const target = { a: 1, b: 1 };
const source1 = { b: 2, c: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
// 拷贝Symbol值属性
Object.assign({ a: 'b' }, { [Symbol('c')]: 'd' })
// { a: 'b', Symbol(c): 'd' }
复制代码
```

- 传入`null`和`undefined`会报错。

```
Object.assign(undefined) // 报错
Object.assign(null) // 报错
复制代码
```

- 注意点
    1. `Object.assign`方法实行的是浅拷贝，而不是深拷贝。
    2. 遇到同名属性，`Object.assign`的处理方法是替换，而不是添加。
    3. 数组的处理
        - `Object.assign`可以用来处理数组，但是会**把数组视为对象**。

        ```
        Object.assign([1, 2, 3], [4, 5])
        // [4, 5, 3]
        复制代码
        ```

4. 取值函数的处理

- `Object.assign`只能进行值的复制，如果要复制的值是一个取值函数，那么将求值后再复制。

```
const source = {
  get foo() { return 1 }
};
const target = {};
Object.assign(target, source)
// { foo: 1 }
```
复制代码

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

1. Object.keys()

- `ES5`引入了`Object.keys`方法，返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（`enumerable`）属性的键名。我们如果只需要拿对象的键名就可以使用这个放。

```
let obj = { a: 1, b: 2, c: 3 };

for (let key of Object.keys(obj)) {
    console.log(key); // 'a', 'b', 'c'
    console.log(obj[key]) // 1, 2, 3
}
```
复制代码

1. Object.values()

- 同`Object.keys`方法，不同的是一个返回键一个返回值。想通过这个方法拿键名就没有办法了。

```
let obj = { a: 1, b: 2, c: 3 };

for (let value of Object.values(obj)) {
    console.log(value); // 1, 2, 3
}

// 数值的键名
const obj = { 100: 'a', 2: 'b', 7: 'c' };
Object.values(obj)
// ["b", "c", "a"]
```
复制代码

- **属性名为数值的属性，是按照数值大小，从小到大遍历的，因此返回的顺序是`b`、`c`、`a`。**
- **`Object.values`会过滤属性名为`Symbol`值的属性。**

1. Object.entries()

- `Object.entries()`方法返回一个数，除了返回值类型不一样之外，其他的行为和`Object.values`基本一致。

```
for (let [key, value] of Object.entries(obj)) {
    console.log([key, value]); // ['a', 1], ['b', 2], ['c', 3]
}
```
复制代码

- 对于遍历对象我现在常用以下写法

```
Object.entries(obj).forEach(([key, value]) => {
    console.log([key, value]); // ['a', 1], ['b', 2], ['c', 3]
})
复制代码
```

Object.fromEntries()

- `Object.fromEntries()`方法是`Object.entries()`的逆操作，用于将一个键值对数组转为对象。主要目的，是将键值对的数据结构还原为对象。

```
Object.fromEntries([
  ['foo', 'bar'],
  ['baz', 42]
])
// { foo: "bar", baz: 42 }
复制代码
```

- `Map`结构就是一个键值对的数组，所以很适用于`Map`结构转为对象。

```
// 例一
const entries = new Map([
  ['foo', 'bar'],
  ['baz', 42]
]);

Object.fromEntries(entries)
// { foo: "bar", baz: 42 }

// 例二
const map = new Map().set('foo', true).set('bar', false);
Object.fromEntries(map)
// { foo: true, bar: false }
```

Symbol

- ES6-Symbol

特性

1. 唯一性

- 属性名属于`Symbol`类型的，都是独一无二的，可以保证不会与其他属性名产生冲突。即使是两个声明完全一样的也是不相等的。

```
// 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
复制代码
```

1. 不能和其他类型运算

- `Symbol` 值不能与其他类型的值进行运算，会报错。

```
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
复制代码
```

1. 类型转换

- `Symbol` 值可以显式转为字符串和布尔值，但是不能转为数值。

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

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

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

Number(sym) // TypeError
sym + 2 // TypeError
复制代码
```

Symbol的应用

1. 常见的唯一值

- `Symbol` 本身的特性就是任何两个 `Symbol` 类型的值都不相等，所以我们可以在不知到命名变量时，都设置为 `Symbol` 类型。**`Symbol` 类型不能使用new操作符，因为生成的 `Symbol` 是一个原始类型的值，不是对象。**

1. 私有属性

- 由于 `Symbol` 类型的作为属性名时，遍历对象的时候，该属性不会出现在 `for...in`、`for...of` 循环中，也不会被 `Object.keys()`、`Object.getOwnPropertyNames()`、`JSON.stringify()` 返回。需要通过 `Object.getOwnPropertySymbols()` 方法，可以获取指定对象的所有 `Symbol` 属性名。这样就可以把 `Symbol` 类型的属性作为私有属性。
- **新的 `API`，`Reflect.ownKeys()` 方法可以返回所有类型的键名，包括常规键名和 `Symbol` 键名。**

1. 消除魔法字符串

- 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。我们可以把对应的字符串或数值设置成Symbol类型即可。

```
const name = {
    first: Symbol('detanx')
}
function getName(firstName) {
  switch (firstName) {
    case name.first: // 魔术字符串
      ...
  }
}

getName(name.first); // 魔术字符串
复制代码
```

Symbol.for()和Symbol.keyFor()

- `Symbol.for()` 不会每次调用就返回一个新的 `Symbol` 类型的值，而是会先检查给定的key是否已经存在，如果不存在才会新建一个值。即如果传入相同的 `key` 创建的值是相等的。而 `Symbol()` 每次创建的值都不相同，即使 `key` 相同。

```
let s1 = Symbol.for('detanx');
let s2 = Symbol.for('detanx');
s1 === s2 // true

let s1 = Symbol('detanx');
let s2 = Symbol('detanx');
s1 === s2 // false
复制代码
```

- `Symbol.keyFor()` 方法返回一个已登记的 `Symbol` 类型值的 `key`。未登记的 `Symbol` 值，返回 `undefined`。

```
let s1 = Symbol.for("detanx");
Symbol.keyFor(s1) // "detanx"

let s2 = Symbol("detanx");
Symbol.keyFor(s2) // undefined
复制代码
```

- **`Symbol()` 写法没有登记机制，所以每次调用都会返回一个不同的值。**
- **`Symbol.for()` 为 `Symbol` 值登记的名字，是全局环境的，不管有没有在全局环境运行。**

内置的Symbol值

- 内置的Symbol值 想了解的可以自己跳过去看看。

Set 和 Map 数据结构

- ES6-Set 和 Map 数据结构

Set

1. 特性
    - 类似于数组，但是成员的值都是唯一的，没有重复的值。
    - `Set`函数可以接受一个数组（或者具有`iterable`接口的其他数据结构）作为参数，用来初始化。
    - 内部两个数据是否相同基本和`===`类似，区别是在`Set`中`NaN`和`NaN`也是相等的。
2. 应用
    - 数组或字符串去重。

```
[...new Set([1, 1, 2, 3])] // [1, 2, 3]

[...new Set('ababbc')].join('') // "abc"
复制代码
```

    - 某些需要唯一性的数据，例如：用户名，id等。
3. `Set`

    实例的属性和方法
    - `Set`结构的实例有以下属性。

```
Set.prototype.constructor：构造函数，默认就是Set函数。
Set.prototype.size：返回Set实例的成员总数。
复制代码
```

    - 操作方法（用于操作数据

```
Set.prototype.add(value)：添加某个值，返回 Set 结构本身。
Set.prototype.delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。
Set.prototype.has(value)：返回一个布尔值，表示该值是否为Set的成员。
Set.prototype.clear()：清除所有成员，没有返回值。
复制代码
```

    - 遍历方法（用于遍历成员）

```
Set.prototype.keys()：返回键名的遍历器
Set.prototype.values()：返回键值的遍历器
Set.prototype.entries()：返回键值对的遍历器
Set.prototype.forEach()：使用回调函数遍历每个成员
复制代码
```

WeakSet

1. 与`Set`的区别

- `WeakSet`的成员只能是对象（`null`除外），而不能是其他类型的值。

```
const ws = new WeakSet();
ws.add(1) // TypeError: Invalid value used in weak set
ws.add(Symbol())  // TypeError: invalid value used in weak set
ws.add(null)  // TypeError: invalid value used in weak set
复制代码
```

- `WeakSet` 中的对象都是弱引用，即垃圾回收机制不考虑`WeakSet` 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 `WeakSet`之中。

1. 方法。

```
WeakSet.prototype.add(value)：向 WeakSet 实例添加一个新成员。
WeakSet.prototype.delete(value)：清除 WeakSet 实例的指定成员。
WeakSet.prototype.has(value)：返回一个布尔值，表示某个值是否在 WeakSet 实例之中。
复制代码
```

- **`WeakSet` 没有`size`属性，没有办法遍历它的成员。**

Map

- 由于传统的`Object`对象只能使用字符串当作键，所以新增的`Map`结构可以将各种类型的值（包括对象）都可以当作键。

```
const m = new Map();
const o = {p: 'Hello World'};

m.set(o, 'content')
m.get(o) // "content"
复制代码
```

1. 方法

- 相比`Set`的操作方法，`Map`没有`add`方法，新增了`get`方法和`set`方法。遍历方法则是基本是一样的。

```
Map.prototype.get(key) // 读取key对应的键值，如果找不到key，返回undefined。
Map.prototype.has(key) // 返回一个布尔值，表示某个键是否在当前 Map 对象之中。
复制代码
```

1. 转换

   (1)

   ```
   Map
   ```

   和数组

   - 在第二弹中也提到了`Map`和数组之间转换，他们之间互转是很方便的。

   ```
   // Map转为数组
   const myMap = new Map()
     .set(true, 7)
     .set({foo: 3}, ['abc']);
   [...myMap]
   // [ [ true, 7 ], [ { foo: 3 }, [ 'abc' ] ] ]

   // 数组转为 Map。
   new Map([
     [true, 7],
     [{foo: 3}, ['abc']]
   ])
   ```

```
// Map {
//   true => 7,
//   Object {foo: 3} => ['abc']
// }
```
复制代码

(2)

```
Map
```

和对象

- `Map` 的键都是字符串，它可以无损地转为对象。如果有非字符串的键名，那么这个键名会被转成字符串，再作为对象的键名。对象转为 `Map` 可以通过 `Object.entries()`。

```
// Map => Object
function strMapToObj(strMap) {
  let obj = Object.create(null);
  for (let [k,v] of strMap) {
    obj[k] = v;
  }
  return obj;
}

const myMap = new Map()
  .set('yes', true)
  .set('no', false);
strMapToObj(myMap)
// { yes: true, no: false }

// Object => Map
// let obj = {"a":1, "b":2};
let map = new Map(Object.entries(obj));
```
复制代码

2. 应用

- 在存储的数据是键值对的时候，并且键名的类型可能是多种类型是可以使用 `Map` 结构。**与 `java` 中的 `Map` 结构有区别。**

WeakMap

1. 与

```
Map
```

的区别

- `WeakMap` 只接受对象作为键名（`null` 除外），不接受其他类型的值作为键名。

```
const map = new WeakMap();
map.set(1, 2) // TypeError: 1 is not an object!
map.set(Symbol(), 2) // TypeError: Invalid value used as weak map key
map.set(null, 2) // TypeError: Invalid value used as weak map key
map.set(new Number(1), 2) // WeakMap {Number => 2}
```
复制代码

- `WeakMap`的键名所指向的对象，不计入垃圾回收机制。一旦不再需要这两个对象，我们就必须手动删除这个引用，否则垃圾回收机制就不会释放占用的内存。

```
const e1 = document.getElementById('foo');
const e2 = document.getElementById('bar');
const arr = [
  [e1, 'foo 元素'],
  [e2, 'bar 元素'],
];
// 不需要 e1 和 e2 的时候
// 必须手动删除引用
arr [0] = null;
arr [1] = null;
复制代码
```

- **WeakMap 弱引用的只是键名，而不是键值。键值依然是正常引用。**
- 没有遍历操作（即没有`keys()`、`values()`和`entries()`方法），也没有`size`属性。
- 无法清空，即不支持`clear`方法。`WeakMap`只有四个方法可用：`get()`、`set()`、`has()`、`delete()`。

2. 应用

(1)

```
DOM
```

节点作为键名

- `document.getElementById('logo')`是一个`DOM`节点，每当发生`click`事件，就更新一下状态。我们将这个状态作为键值放在`WeakMap`里，对应的键名就是这个节点对象。一旦这个`DOM`节点删除，该状态就会自动消失，不存在内存泄漏风险。

```
let myWeakmap = new WeakMap();

myWeakmap.set(
  document.getElementById('logo'),
  {timesClicked: 0})
;

document.getElementById('logo').addEventListener('click', function() {
  let logoData = myWeakmap.get(document.getElementById('logo'));
  logoData.timesClicked++;
}, false);
复制代码
```

(2) 部署私有属性

- 内部属性是实例的弱引用，所以如果删除实例，它们也就随之消失，不会造成内存泄漏。

```
const _counter = new WeakMap();
const _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
```

```
  dec() {
    let counter = _counter.get(this);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

const c = new Countdown(2, () => console.log('DONE'));

c.dec()
c.dec()
// DONE
```

Proxy

- ES6-Proxy
- `Proxy` 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”（`meta programming`），即对编程语言进行编程。可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。

使用方式

1. 将 `Proxy` 对象，设置到`object.proxy`属性，从而可以在`object`对象上调用。

```
const object = { proxy: new Proxy(target, handler) };
复制代码
```

1. 使用`Proxy`对象

```
const proxy = new Proxy({}, {
  get: function(target, propKey) {
    return 'detanx';
  }
});

obj.name // 'detanx'
复制代码
```

- **同一个拦截器函数，可以设置拦截多个操作。**

拦截操作

- `Proxy`

  支持的拦截操作一览，一共

  `13`

  种。

1. `get(target, propKey, receiver)`：拦截对象属性的读取，比如`proxy.foo`和`proxy['foo']`。
2. `set(target, propKey, value, receiver)`：拦截对象属性的设置，比如`proxy.foo = v`或`proxy['foo'] = v`，返回一个布尔值。
3. `has(target, propKey)`：拦截`propKey in proxy`的操作，返回一个布尔值。
4. `deleteProperty(target, propKey)`：拦截`delete proxy[propKey]`的操作，返回一个布尔值。
5. `ownKeys(target)`：拦截`Object.getOwnPropertyNames(proxy)`、`Object.getOwnPropertySymbols(proxy)`、`Object.keys(proxy)`、`for...in`循环，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而`Object.keys()`的返回结果仅包括目标对象自身的可遍历属性。
6. `getOwnPropertyDescriptor(target, propKey)`：拦截`Object.getOwnPropertyDescriptor(proxy, propKey)`，返回属性的描述对象。
7. `defineProperty(target, propKey, propDesc)`：拦截`Object.defineProperty(proxy, propKey, propDesc）`、`Object.defineProperties(proxy, propDescs)`，返回一个布尔值。
8. `preventExtensions(target)`：拦截`Object.preventExtensions(proxy)`，返回一个布尔值。
9. `getPrototypeOf(target)`：拦截`Object.getPrototypeOf(proxy)`，返回一个对象。
10. `isExtensible(target)`：拦截`Object.isExtensible(proxy)`，返回一个布尔值。
11. `setPrototypeOf(target, proto)`：拦截`Object.setPrototypeOf(proxy, proto)`，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。
12. `apply(target, object, args)`：拦截`Proxy`实例作为函数调用的操作，比如`proxy(...args)`、`proxy.call(object, ...args)`、`proxy.apply(...)`。
13. `construct(target, args)`：拦截`Proxy`实例作为构造函数调用的操作，比如`new proxy(...args)`。

get()

- 拦截某个属性的读取操作，可以接受三个参数，依次为目标对象、属性名和 proxy 实例本身（**严格地说，是操作行为所针对的对象**），其中最后一个参数可选。

```
const proxy = new Proxy({}, {
  get(target, propertyKey [, receiver]) {
    return target[propertyKey];
  }
});
复制代码
```

- 如果一个属性不可配置（`configurable`）且不可写（`writable`），则`Proxy`不能修改该属性，否则通过`Proxy`对象访问该属性会报错。

```
const target = Object.defineProperties({}, {
  foo: {
    value: 123,
    writable: false,
    configurable: false
  },
});

const handler = {
  get(target, propKey) {
    return 'abc';
```

```
  }
};

const proxy = new Proxy(target, handler);

proxy.foo
// TypeError: Invariant check failed
复制代码
```

set()

- `set`方法用来拦截某个属性的赋值操作，可以接受四个参数，依次为目标对象、属性名、属性值和 `Proxy` 实例本身，其中最后一个参数可选。

```
let prxyo = new Proxy({}, {
  set(target, propertyKey, value [, receiver]) {
    return target[propertyKey];
  }
});
复制代码
```

- **如果目标对象自身的某个属性，不可写（`configurable`）且不可写（`writable`），那么`set`方法将不起作用。**

```
const obj = {};
Object.defineProperty(obj, 'foo', {
  value: 'bar',
  writable: false,
});

const handler = {
  set: function(obj, prop, value, receiver) {
    obj[prop] = 'baz';
  }
};

const proxy = new Proxy(obj, handler);
proxy.foo = 'baz';
proxy.foo // "bar"
复制代码
```

- **严格模式下，`set`代理如果没有返回`true`，就会报错。**

```
'use strict';
const handler = {
  set: function(obj, prop, value, receiver) {
    obj[prop] = receiver;
    // 无论有没有下面这一行，都会报错
    return false;
  }
};
const proxy = new Proxy({}, handler);
proxy.foo = 'bar';
// TypeError: 'set' on proxy: trap returned falsish for property 'foo'
复制代码
```

apply()

- `apply`方法拦截函数的调用、`call`和`apply`操作。
- `apply`方法可以接受三个参数，分别是目标对象、目标对象的上下文对象（`this`）和目标对象的参数数组。

```
const proxy = new Proxy( {}, {
  apply (target, ctx, args) {
    return Reflect.apply(...arguments);
  }
});
复制代码
```

- 拦截示例

```
var twice = {
  apply (target, ctx, args) {
    return Reflect.apply(...arguments) * 2;
  }
};
function sum (left, right) {
  return left + right;
};
var proxy = new Proxy(sum, twice);
proxy(1, 2) // 6
proxy.call(null, 5, 6) // 22
proxy.apply(null, [7, 8]) // 30
复制代码
```

- **直接调用`Reflect.apply`方法，也会被拦截。**

```
Reflect.apply(proxy, null, [9, 10]) // 38
复制代码
```

has()

- `has`方法用来拦截`HasProperty`操作（**注意不是`HasOwnProperty`，即`has`方法不判断一个属性是对象自身的属性，还是继承的属性。**），即判断对象是否具有某个属性时，这个方法会生效。典型的操作就是`in`运算符。可以接受两个参数，分别是目标对象、需查询的属性名。
- 用于隐藏某些属性

```
// 隐藏_开头的属性
var handler = {
  has (target, key) {
    if (key[0] === '_') {
      return false;
    }
    return key in target;
  }
};
var target = { _prop: 'foo', prop: 'foo' };
var proxy = new Proxy(target, handler);
'_prop' in proxy // false
复制代码
```

- `has`**拦截对** `for...in`**循环不生效。**

```
let stu1 = {name: '张三', score: 59};
let stu2 = {name: '李四', score: 99};

let handler = {
  has(target, prop) {
    if (prop === 'score' && target[prop] < 60) {
      console.log(`${target.name} 不及格`);
      return false;
    }
    return prop in target;
  }
}

let oproxy1 = new Proxy(stu1, handler);
let oproxy2 = new Proxy(stu2, handler);

'score' in oproxy1
// 张三 不及格
// false

'score' in oproxy2
// true

for (let a in oproxy1) {
  console.log(oproxy1[a]);
}
// 张三
// 59

for (let b in oproxy2) {
  console.log(oproxy2[b]);
}
// 李四
// 99
复制代码
```

construct()

- `construct`方法用于拦截new命令，下面是拦截对象的写法。可以接受三个参数:目标对象、构造函数的参数对象、创造实例对象时，`new`命令作用的构造函数。`construct`**方法返回的必须是一个对象，否则会报错。**

```
var handler = {
  construct (target, args, newTarget) {
    return new target(...args);
  }
};

var p = new Proxy(function () {}, {
  construct: function(target, args) {
    console.log('called: ' + args.join(', '));
    return { value: args[0] * 10 };
  }
});
```

```
(new p(1)).value
// "called: 1"
// 10

var p = new Proxy(function() {}, {
  construct: function(target, argumentsList) {
    return 1;
  }
});

new p() // 报错
// Uncaught TypeError: 'construct' on proxy: trap returned non-object ('1')
复制代码
```

deleteProperty()

- `deleteProperty`方法用于拦截`delete`操作，如果这个方法抛出错误或者返回`false`，当前属性就无法被`delete`命令删除。**目标对象自身的不可配置（`configurable`）的属性，不能被`deleteProperty`方法删除，否则报错。**

```
var handler = {
  deleteProperty (target, key) {
    delete target[key];
    return true;
  }
};

var target = { prop: 'foo' };
var proxy = new Proxy(target, handler);
delete proxy._prop
复制代码
```

defineProperty()

- `defineProperty()`方法拦截了`Object.defineProperty()`操作。**如果目标对象不可扩展（`non-extensible`），则`defineProperty()`不能增加目标对象上不存在的属性，否则会报错。另外，如果目标对象的某个属性不可写（`writable`）或不可配置（`configurable`），则`defineProperty()`方法不得改变这两个设置。**

```
var handler = {
  defineProperty (target, key, descriptor) {
    return false;
  }
};
var target = {};
var proxy = new Proxy(target, handler);
proxy.foo = 'bar' // 不会生效
复制代码
```

getOwnPropertyDescriptor()

- 拦截`Object.getOwnPropertyDescriptor()`，返回一个属性描述对象或者`undefined`。

```
var handler = {
```

```
  getOwnPropertyDescriptor (target, key) {
    if (key[0] === '_') {
      return;
    }
    return Object.getOwnPropertyDescriptor(target, key);
  }
};
var target = { _foo: 'bar', baz: 'tar' };
var proxy = new Proxy(target, handler);
Object.getOwnPropertyDescriptor(proxy, 'wat')
// undefined
Object.getOwnPropertyDescriptor(proxy, '_foo')
// undefined
Object.getOwnPropertyDescriptor(proxy, 'baz')
// { value: 'tar', writable: true, enumerable: true, configurable: true }
```
复制代码

getPrototypeOf()

- 方法主要用来拦截获取对象原型，返回值必须是对象或者`null`，否则报错。
- **如果目标对象不可扩展（`non-extensible`），`getPrototypeOf()`方法必须返回目标对象的原型对象。**
- 拦截下面这些操作:
    1. Object.prototype.**proto**
    2. Object.prototype.isPrototypeOf()
    3. Object.getPrototypeOf()
    4. Reflect.getPrototypeOf()
    5. instanceof

```
var proto = {};
var p = new Proxy({}, {
  getPrototypeOf(target) {
    return proto;
  }
});
Object.getPrototypeOf(p) === proto // true
```
复制代码

isExtensible()、preventExtensions()和setPrototypeOf()

1. 相同点
    - 这三个方法只能返回布尔值，否则会被自动转为布尔值。
2. 不同点
    - `isExtensible()`方法有一个强限制，它的返回值必须与目标对象的`isExtensible`属性保持一致，否则就会抛出错误。

```
var p = new Proxy({}, {
  isExtensible: function(target) {
    console.log("called");
    return true;
  }
});

Object.isExtensible(p)
```

```
// "called"
// true

// 报错
var p = new Proxy({}, {
  isExtensible: function(target) {
    return false;
  }
});

Object.isExtensible(p)
// Uncaught TypeError: 'isExtensible' on proxy: trap result does not reflect
extensibility of proxy target (which is 'true')
```
复制代码

- `preventExtensions()`方法有一个限制，只有目标对象不可扩展时（即`Object.isExtensible(proxy)`为`false`），`proxy.preventExtensions`才能返回`true`，否则会报错。为了防止出现这个问题，通常要在`proxy.preventExtensions()`方法里面，调用一次`Object.preventExtensions()`。

```
var proxy = new Proxy({}, {
  preventExtensions: function(target) {
    return true;
  }
});

Object.preventExtensions(proxy)
// Uncaught TypeError: 'preventExtensions' on proxy: trap returned truish
but the proxy target is extensible

// 纠正后
var proxy = new Proxy({}, {
  preventExtensions: function(target) {
    console.log('called');
    Object.preventExtensions(target);
    return true;
  }
});

Object.preventExtensions(proxy)
// "called"
// Proxy {}
```
复制代码

- `setPrototypeOf()`如果目标对象不可扩展（`non-extensible`），`setPrototypeOf()`方法不得改变目标对象的原型。

```
var handler = {
  setPrototypeOf (target, proto) {
    throw new Error('Changing the prototype is forbidden');
  }
};
var proto = {};
var target = function () {};
var proxy = new Proxy(target, handler);
Object.setPrototypeOf(proxy, proto);
// Error: Changing the prototype is forbidden
复制代码
```

ownKeys()

- `ownKeys()`

  方法用来拦截对象自身属性的读取操作。具体来说，拦截以下操作。

  1. `Object.getOwnPropertyNames()`
  2. `Object.getOwnPropertySymbols()`
  3. `Object.keys()`
  4. `for...in`循环

- 使用`Object.keys()`方法时，有三类属性会被`ownKeys()`方法自动过滤，不会返回。目标对象上不存在的属性、属性名为`Symbol`值、不可遍历（`enumerable`）的属性。
- **使用限制（很重要）**

  1. `ownKeys()`方法返回的数组成员，只能是字符串或`Symbol`值。如果有其他类型的值，或者返回的根本不是数组，就会报错。**

```
var obj = {};

var p = new Proxy(obj, {
  ownKeys: function(target) {
    return [123, true, undefined, null, {}, []];
  }
});

Object.getOwnPropertyNames(p)
// Uncaught TypeError: 123 is not a valid property name
复制代码
```

  1. 如果目标对象自身包含不可配置的属性，则该属性必须被`ownKeys()`方法返回，否则报错。

```
var obj = {};
Object.defineProperty(obj, 'a', {
  configurable: false,
  enumerable: true,
  value: 10 }
);

var p = new Proxy(obj, {
  ownKeys: function(target) {
    return ['b'];
  }
});
```

```
Object.getOwnPropertyNames(p)
// Uncaught TypeError: 'ownKeys' on proxy: trap result did not include 'a'
```
复制代码

1. 目标对象是不可扩展的（`non-extensible`），这时`ownKeys()`方法返回的数组之中，必须包含原对象的所有属性，且不能包含多余的属性，否则报错。

```
var obj = {
  a: 1
};

Object.preventExtensions(obj);

var p = new Proxy(obj, {
  ownKeys: function(target) {
    return ['a', 'b'];
  }
});

Object.getOwnPropertyNames(p)
// Uncaught TypeError: 'ownKeys' on proxy: trap returned extra keys but
proxy target is non-extensible
```
复制代码

Proxy.revocable()

- `Proxy.revocable()`方法返回一个对象，该对象的`proxy`属性是Proxy实例，`revoke`属性是一个函数，可以取消`Proxy`实例。

```
let target = {};
let handler = {};

let {proxy, revoke} = Proxy.revocable(target, handler);

proxy.foo = 123;
proxy.foo // 123

revoke();
proxy.foo // TypeError: Revoked
```
复制代码

- 当执行`revoke`函数之后，再访问`Proxy`实例，就会抛出一个错误。
- `Proxy.revocable()`的一个使用场景是，目标对象不允许直接访问，必须通过代理访问，一旦访问结束，就收回代理权，不允许再次访问。

`this`问题

- 当`proxy`代理后，`this`的指向会发生变化。

```
const target = {
  m: function () {
    console.log(this === proxy);
  }
};
const handler = {};

const proxy = new Proxy(target, handler);

target.m() // false
proxy.m()  // true
```
复制代码

- 一个例子，由于`this`指向的变化，导致`Proxy`无法代理目标对象。

```
const _name = new WeakMap();

class Person {
  constructor(name) {
    _name.set(this, name);
  }
  get name() {
    return _name.get(this);
  }
}

const jane = new Person('Jane');
jane.name // 'Jane'

const proxy = new Proxy(jane, {});
proxy.name // undefined

=> // 增加get拦截
const proxy = new Proxy(jane, {
  get(target, prop) {
    console.log('prop', prop)
    if (prop === 'name') {
      return target.name
    }
    return Reflect.get(target, prop);
  }
});
proxy.name // Jane
```
复制代码

- 有些原生对象的内部属性，只有通过正确的`this`才能拿到，所以`Proxy`也无法代理这些原生对象的属性。这时，`this`绑定原始对象，就可以解决这个问题。

```
const target = new Date();
const handler = {};
const proxy = new Proxy(target, handler);

proxy.getDate();
// TypeError: this is not a Date object.

// 在get中绑定this
```

```
const target = new Date('2015-01-01');
const handler = {
  get(target, prop) {
    if (prop === 'getDate') {
      return target.getDate.bind(target);
    }
    return Reflect.get(target, prop);
  }
};
const proxy = new Proxy(target, handler);

proxy.getDate() //
```
复制代码

vue3使用Proxy替换defineProperty

- Object.defineProperty
    1. `Object.defineProperty`无法监控到数组下标的变化，导致直接通过数组的下标给数组设置值，不能实时响应。经过vue内部处理后可以使用以下几种方法来监听数组`push()`、`pop()`、`shift()`、`unshift()`、`splice()`、`sort()`、`reverse()`，由于只针对了以上八种方法进行了`hack`处理,所以其他数组的属性也是检测不到的，还是具有一定的局限性。
    2. `Object.defineProperty`只能劫持对象的属性,因此我们需要对每个对象的每个属性进行遍历。
- Proxy
    1. 可以劫持整个对象，并返回一个新对象。
    2. 有`13`种劫持操作。

Reflect

- ES6-Reflect

1. 将`Object`对象的一些明显属于语言内部的方法（比如`Object.defineProperty`），放到`Reflect`对象上。
2. 修改某些`Object`方法的返回结果，让其变得更合理。比如，`Object.defineProperty(obj, name, desc)`在无法定义属性时，会抛出一个错误，而`Reflect.defineProperty(obj, name, desc)`则会返回`false`。

```
// 老写法
try {
  Object.defineProperty(target, property, attributes);
  // success
} catch (e) {
  // failure
}

// 新写法
if (Reflect.defineProperty(target, property, attributes)) {
  // success
} else {
  // failure
}
```
复制代码

1. 让`Object`操作都变成函数行为。某些`Object`操作是命令式，比如`name in obj`和`delete obj[name]`，而`Reflect.has(obj, name)`和`Reflect.deleteProperty(obj, name)`让它们变成了函数行为。

```
// 老写法
'assign' in Object // true

// 新写法
Reflect.has(Object, 'assign') // true
复制代码
```

1. `Reflect`对象的方法与`Proxy`对象的方法一一对应，只要是`Proxy`对象的方法，就能在`Reflect`对象上找到对应的方法。**每一个`Proxy`对象的拦截操作（`get`、`delete`、`has`、`...`），内部都调用对应的`Reflect`方法。**

Promise

- ES6-Promise
- 进来看看ES6 Promise最全手写实现
- `Promise`是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。

特点

1. 对象的状态不受外界影响。有三种状态：`pending`（进行中）、`fulfilled`（已成功）和`rejected`（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。
2. **一旦状态改变，就不会再变，任何时候都可以得到这个结果。**`Promise`对象的状态改变，只有两种可能：从`pending`变为`fulfilled`和从`pending`变为`rejected`。

使用

- `Promise`**对象是一个构造函数**，用来生成Promise实例。构造函数接受一个函数作为参数，该函数的两个参数分别是`resolve`和`reject`。`resolve`函数的作用是，将`Promise`对象的状态从“未完成”变为“成功”（即从`pending`变为`resolved`）；`reject`函数的作用是，将`Promise`对象的状态从“未完成”变为“失败”（即从`pending`变为`rejected`）。

```
const promise = new Promise(function(resolve, reject) {
  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});
复制代码
```

- `Promise`实例生成以后，可以用`then`方法分别指定`resolved`状态和`rejected`状态的回调函数。方法可以接受两个回调函数作为参数。第一个回调函数是`Promise`对象的状态变为`resolved`时调用，第二个回调函数是`Promise`对象的状态变为`rejected`时调用。第二个函数是可选的，不一定要提供。

```
promise.then(function(value) {
  // success
}, function(error) {
  // failure
});
复制代码
```

- `Promise` **新建后就会立即执行。** `then`方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行。JavaScript事件循环机制

Promise.resolve()

- 将现有对象转为 `Promise` 对象。
- 不同参数情况：
  1. 参数是一个 `Promise` 实例，直接返回这个实例。
  2. 参数是一个`thenable`对象（`thenable`对象指的是具有then方法的对象），将这个对象转为 `Promise` 对象，然后就立即执行`thenable`对象的`then`方法。

```
let thenable = {
  then: function(resolve, reject) {
    resolve(42);
  }
};

let p1 = Promise.resolve(thenable);
p1.then(function(value) {
  console.log(value);  // 42
});
复制代码
```

  1. 参数不是具有then方法的对象，或根本就不是对象，返回一个新的`Promise` 对象，状态为`resolved`。

```
const p = Promise.resolve('detanx');

p.then(function (s){
  console.log(s)
});
// detanx
复制代码
```

  1. 不带有任何参数，直接返回一个`resolved`状态的 `Promise` 对象。**立即`resolve()`的`Promise`对象，是在本轮“事件循环”（`event loop`）的结束时执行，而不是在下一轮“事件循环”的开始时。**

```
setTimeout(function () {
  console.log('three');
}, 0);

Promise.resolve().then(function () {
  console.log('two');
});
```

```
console.log('one');

// one
// two
// three
```
复制代码

Promise.reject()

- `Promise.reject(reason)`方法也会返回一个新的`Promise`实例，该实例的状态为`rejected`。**`Promise.reject()`方法的参数，会原封不动地作为`reject`的理由，变成后续方法的参数。** 这一点与`Promise.resolve`方法不一致。

```
const thenable = {
  then(resolve, reject) {
    reject('出错了');
  }
};

Promise.reject(thenable)
.catch(e => {
  console.log(e === thenable)
})
// true
```
复制代码

Promise.prototype.catch()

- `Promise.prototype.catch()`方法是`.then(null, rejection)`或`.then(undefined, rejection)`的别名（可以查看手写`Promise`的该方法实现），用于指定发生错误时的回调函数。

```
// 写法一
const promise = new Promise(function(resolve, reject) {
  try {
    throw new Error('test');
  } catch(e) {
    reject(e);
  }
});
promise.catch(function(error) {
  console.log(error);
});

// 写法二
const promise = new Promise(function(resolve, reject) {
  reject(new Error('test'));
});
promise.catch(function(error) {
  console.log(error);
});
```
复制代码

- **如果`Promise`状态已经变成`resolved`，再抛出错误是无效的。**

```
const promise = new Promise(function(resolve, reject) {
  resolve('ok');
  throw new Error('test');
});
promise
  .then(function(value) { console.log(value) })
  .catch(function(error) { console.log(error) });
// ok
```
复制代码

- **`Promise` 对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。**

```
getJSON('/post/1.json').then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
  // some code
}).catch(function(error) {
  // 处理前面三个Promise产生的错误
});
```
复制代码

- 一般不要在`then()`方法里面定义`Reject`状态的回调函数（即then的第二个参数），总是使用`catch`方法。

```
// bad
promise
  .then(function(data) {
    // success
  }, function(err) {
    // error
  });

// good
promise
  .then(function(data) { //cb
    // success
  })
  .catch(function(err) {
    // error
  });
```
复制代码

- **跟传统的`try/catch`代码块不同的是，如果没有使用`catch()`方法指定错误处理的回调函数，`Promise`对象抛出的错误不会传递到外层代码，即不会有任何反应。**

```
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

someAsyncThing().then(function() {
  console.log('everything is great');
});
```

```
setTimeout(() => { console.log(123) }, 2000);
// Uncaught (in promise) ReferenceError: x is not defined
// 123
复制代码
```

Promise.prototype.finally()

- `finally()`**方法用于指定不管** `Promise` **对象最后状态如何，都会执行的操作。** 该方法是 `ES2018` 引入标准的。

```
promise
.then(result => {···})
.catch(error => {···})
.finally(() => {
    console.log('detanx');
});
// 'detanx'
复制代码
```

- `finally`**方法的回调函数不接受任何参数，这意味着没有办法知道，前面的** `Promise` **状态到底是** `fulfilled` **还是** `rejected`**。方法里面的操作，应该是与状态无关的，不依赖于** `Promise`**的执行结果。** `finally`**方法总是会返回原来的值。**

```
// 实现
Promise.prototype.finally = function (callback) {
  let P = this.constructor;
  return this.then(
    value  => P.resolve(callback()).then(() => value),
    reason => P.resolve(callback()).then(() => { throw reason })
  );
};

// resolve 的值是 undefined
Promise.resolve(2).then(() => {}, () => {})

// resolve 的值是 2
Promise.resolve(2).finally(() => {})

// reject 的值是 undefined
Promise.reject(3).then(() => {}, () => {})

// reject 的值是 3
Promise.reject(3).finally(() => {})
复制代码
```

Promise.all()、Promise.race()和Promise.allSettled()

- 相同点
    1. 方法都是用于将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。
    2. `Promise.all()` 所有的实例返回都是 `fulfilled` 和 `Promise.allSettled()` 的返回值都是数组。
- 不同点

1. 有时候，我们不关心异步操作的结果，只关心这些操作有没有结束。这时，`Promise.allSettled()`方法就很有用。如果没有这个方法，想要确保所有操作都结束，就很麻烦。`Promise.all()`方法无法做到这一点。

```
const urls = [ /* ... */ ];
const requests = urls.map(x => fetch(x));

try {
  await Promise.all(requests);
  console.log('所有请求都成功。');
} catch {
  console.log('至少一个请求失败，其他请求可能还没结束。');
}
复制代码
```

1. Promise.all()所有的实例中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值。Promise.allSettled()中的实例fulfilled时，对象有value属性，rejected时有reason属性，对应两种状态的返回值。

```
const resolved = Promise.resolve(42);
const rejected = Promise.reject(-1);

const allSettledPromise = Promise.allSettled([resolved, rejected]);

allSettledPromise.then(function (results) {
  console.log(results);
});
// [
//    { status: 'fulfilled', value: 42 },
//    { status: 'rejected', reason: -1 }
// ]
复制代码
```

1. 如果作为参数的`Promise`实例，自己定义了`catch`方法，那么它一旦被`rejected`，并不会触发`Promise.all()`的`catch`方法。如果实例没有自己的`catch`方法，就会调用`Promise.all()`的`catch`方法。

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result)
.catch(e => e);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
.then(result => result)
=> // 下面的catch移除
.catch(e => e);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// ["hello", Error: 报错了]
=> // 移除p2的catch
```

```
// // Error: 报错了
复制代码
```

1. `Promise.race()` 只要有一个实例率先改变状态，状态就跟着改变。那个率先改变的 `Promise` 实例的返回值，就传递给回调函数。

```
const p = Promise.race([1,
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error('request timeout')), 5000)
  })
]);

p
.then(console.log)
.catch(console.error);
// 1
复制代码
```

Iterator 和 for...of 循环

- ES6-Iterator 和 for...of 循环
- 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（即依次处理该数据结构的所有成员）。**
- `Iterator`

  的作用：

  1. 为各种数据结构，提供一个统一的、简便的访问接口；
  2. 使得数据结构的成员能够按某种次序排列；
  3. 新的遍历命令 `for...of` 循环，`Iterator` 接口主要供 `for...of` 消费。
- 遍历过程：
  1. 创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。
  2. 第一次调用指针对象的 `next` 方法，可以将指针指向数据结构的第一个成员。
  3. 第二次调用指针对象的 `next` 方法，指针就指向数据结构的第二个成员。
  4. 不断调用指针对象的 `next` 方法，直到它指向数据结构的结束位置。
  - 每一次调用 `next` 方法，都会返回数据结构的当前成员的信息（返回一个包含 `value` 和 `done` 两个属性的对象。`value` 属性是当前成员的值，`done` 属性是一个布尔值，表示遍历是否结束。）

```
var it = makeIterator(['a', 'b']);
it.next() // { value: "a", done: false }
it.next() // { value: "b", done: false }
it.next() // { value: undefined, done: true }

function makeIterator(array) {
  var nextIndex = 0;
  return {
    next: function() {
      return nextIndex < array.length ?
        {value: array[nextIndex++], done: false} :
        {value: undefined, done: true};
    }
```

```
  };
}
```
复制代码

- 对于遍历器对象来说，`done: false`和`value: undefined`属性都是可以省略的。

```
// makeIterator函数可以简写成下面的形式
function makeIterator(array) {
  var nextIndex = 0;
  return {
    next: function() {
      return nextIndex < array.length ?
        {value: array[nextIndex++]} :
        {done: true};
    }
  };
}
```
复制代码

- 使用 TypeScript 的写法
  - 遍历器接口（`Iterable`）、指针对象（`Iterator`）和`next`方法返回值的规格可以描述如下。

```
interface Iterable {
  [Symbol.iterator]() : Iterator,
}

interface Iterator {
  next(value?: any) : IterationResult,
}

interface IterationResult {
  value: any,
  done: boolean,
}
```
复制代码

默认 Iterator 接口

- 当使用`for...of`循环遍历某种数据结构时，该循环会自动去寻找`Iterator`接口。默认的`Iterator`接口部署在数据结构的`Symbol.iterator`属性，或者说，一个数据结构只要具有`Symbol.iterator`属性，就可以认为是“可遍历的”（`iterable`）。

```
const obj = {
  [Symbol.iterator] : function () {
    return {
      next: function () {
        return {
          value: 1,
          done: true
        };
      }
    };
  }
};
```
复制代码

- 原生具备`Iterator`接口的数据结构：`Array`、`Map`、`Set`、`String`、`TypedArray`、**函数的`arguments`对象**、`NodeList`**对象**。
- 一个对象如果要具备可被`for...of`循环调用的`Iterator`接口，就必须在`Symbol.iterator`的属性上部署遍历器生成方法（原型链上的对象具有该方法也可）。

```
class RangeIterator {
  constructor(start, stop) {
    this.value = start;
    this.stop = stop;
  }

  [Symbol.iterator]() { return this; }

  next() {
    var value = this.value;
    if (value < this.stop) {
      this.value++;
      return {done: false, value: value};
    }
    return {done: true, value: undefined};
  }
}

function range(start, stop) {
  return new RangeIterator(start, stop);
}

for (var value of range(0, 3)) {
  console.log(value); // 0, 1, 2
}
```
复制代码

- **类似数组的对象调用数组的`Symbol.iterator`可以遍历；普通对象部署数组的`Symbol.iterator`方法，并无效果。**

```
// 类似数组对象
let iterable = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3,
```

```
  [Symbol.iterator]: Array.prototype[Symbol.iterator]
};
for (let item of iterable) {
  console.log(item); // 'a', 'b', 'c'
}
// 普通对象，下标不是数字
let iterable = {
  a: 'a',
  b: 'b',
  c: 'c',
  length: 3,
  [Symbol.iterator]: Array.prototype[Symbol.iterator]
};
for (let item of iterable) {
  console.log(item); // undefined, undefined, undefined
}
复制代码
```

- **如果Symbol.iterator方法对应的不是遍历器生成函数（即会返回一个遍历器对象），解释引擎将会报错。**

```
var obj = {};
obj[Symbol.iterator] = () => 1;
[...obj] // TypeError: [] is not a function
复制代码
```

调用 Iterator 接口的场合

1. 解构赋值
   - 对数组和 `Set` 结构进行解构赋值时，会默认调用`Symbol.iterator`方法。
2. 扩展运算符
   - 扩展运算符（`...`）也会调用默认的 `Iterator` 接口。
3. `yield*`
   - `yield*`后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。
4. 其他
   - `for...of`、`Array.from()`、`Map(), Set()`,`WeakMap()`,`WeakSet()`（比如`new Map([['a',1],['b',2]])`）、`Promise.all()`、`Promise.race()`...

遍历器对象的 return()和throw()

- 遍历器对象除了具有`next`方法，还可以具有`return`方法和`throw`方法。如果你自己写遍历器对象生成函数，那么`next`方法是必须部署的，`return`方法和`throw`方法是否部署是可选的。
- `for...of`循环提前退出（通常是因为出错，或者有`break`语句），就会调用`return`方法。如果一个对象在完成遍历前，需要清理或释放资源，就可以部署`return`方法。

```
function readLinesSync(file) {
  return {
    [Symbol.iterator]() {
      return {
        next() {
          return { done: false };
        },
```

```
        return() {
          file.close();
          return { done: true };
        }
      };
    },
  };
}
// 触发return情况一
for (let line of readLinesSync(fileName)) {
  console.log(line);
  break;
}

// 触发return情况二
for (let line of readLinesSync(fileName)) {
  console.log(line);
  throw new Error();
}
复制代码
```

- `throw`方法主要是配合 `Generator` 函数使用，下一弹会提到。

for...of 循环

- **作为遍历所有数据结构的统一的方法。**
- 和其他遍历对比
    1. `for...of`会正确识别 `32` 位 `UTF-16` 字符。有着同`for...in`一样的简洁语法，但是没有`for...in`那些缺点。不同于`forEach`方法，它可以与`break`、`continue`和`return`配合使用。

    ```
    for (let x of 'a\uD83D\uDC0A') {
      console.log(x);
    }
    // 'a'
    // '\uD83D\uDC0A'
    复制代码
    ```

    1. `forEach`无法中途跳出`forEach`循环，`break`命令或`return`命令都不能奏效。
    2. `for...in`循环缺点，一是数组的键名是数字，但是`for...in`循环是以字符串作为键名“`0`”、“`1`”、“`2`”等等。二是`for...in`循环不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键。最后某些情况下，`for...in`循环会以任意顺序遍历键名。`for...in`**循环主要是为遍历对象而设计的，不适用于遍历数组。**
- 其他类型的数据可以使用其他技巧转为有

    ```
    Iterator
    ```

    接口的数据结构再使用

    ```
    for...of
    ```

    进行遍历。

    1. 对象

- 使用`Object.keys`方法将对象的键名生成一个数组，然后遍历这个数组。

```
for (var key of Object.keys(someObject)) {
  console.log(key + ': ' + someObject[key]);
}
复制代码
```

- 使用`Generator`函数将对象重新包装一下。

```
function* entries(obj) {
  for (let key of Object.keys(obj)) {
    yield [key, obj[key]];
  }
}

for (let [key, value] of entries(obj)) {
  console.log(key, '->', value);
}
// a -> 1
// b -> 2
// c -> 3
复制代码
```

1. 类似数组的对象但不具有`Iterator`接口
- 使用`Array.from`方法将其转为数组。

```
let arrayLike = { length: 2, 0: 'a', 1: 'b' };
// 报错
for (let x of arrayLike) {
  console.log(x);
}
// 正确
for (let x of Array.from(arrayLike)) {
  console.log(x);
}
```

Generator

- ES6-Generator

基本概念

- `Generator`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。语法上，`Generator`函数是一个状态机，封装了多个内部状态。**执行`Generator`函数会返回一个遍历器对象，可以依次遍历`Generator`函数内部的每一个状态。**

1. 特征
    1. `function`关键字与函数名之间有一个星号；
    2. 函数体内部使用`yield`表达式，定义不同的内部状态。

```
function* detanxGenerator() {
  yield 'detanx';
  return 'ending';
}

const dg = detanxGenerator();
```
复制代码

2. 调用

```
dg.next() // { value: 'detanx', done: false }
dg.next() // { value: 'ending', done: true }
dg.next() // { value: undefined, done: true }
```
复制代码

   1. 第一次调用，`Generator` 函数开始执行，直到遇到第一个 `yield` 表达式为止。`next` 方法返回一个对象，它的 `value` 属性就是当前 `yield` 表达式的值 `hello`，`done` 属性的值 `false`，表示遍历还没有结束。
   2. 第二次调用，`Generator` 函数从上次yield表达式停下的地方，一直执行到 `return` 语句（如果没有 `return` 语句，就执行到函数结束）。`done` 属性的值 `true`，表示遍历已经结束。
   3. 第三次调用，此时 `Generator` 函数已经运行完毕，`next` 方法返回对象的 `value` 属性为 `undefined`，`done` 属性为 `true`。以后再调用 `next` 方法，返回的都是这个值。
3. 写法
   - `function` 关键字与函数名之间的 `*` 未规定，所以有很多写法，我们写得时候最好还是使用第一种，即 `*` 紧跟着 `function` 关键字后面，`*` 后面再加一个空格。

```
function* foo(x, y) { ··· }
function*foo(x, y) { ··· }
function *foo(x, y) { ··· }
function * foo(x, y) { ··· }
```
复制代码

yield 表达式

- `Generator` 函数返回的遍历器对象，只有调用 `next` 方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。`yield` 表达式就是暂停标志。

1. `next` 方法的运行逻辑
   （1）遇到 `yield` 表达式，就暂停执行后面的操作，并将紧跟在 `yield` 后面的那个表达式的值，作为返回的对象的 `value` 属性值。
   （2）下一次调用 `next` 方法时，再继续往下执行，直到遇到下一个 `yield` 表达式。
   （3）如果没有再遇到新的 `yield` 表达式，就一直运行到函数结束，直到 `return` 语句为止，并将 `return` 语句后面的表达式的值，作为返回的对象的 `value` 属性值。
   （4）如果该函数没有 `return` 语句，则返回的对象的 `value` 属性值为 `undefined`。
2. `yield` 表达式后面的表达式，只有当调用 `next` 方法、内部指针指向该语句时才会执行。下面的 `123 + 456` 不会立即求值，只有当执行 `next` 到对应的 `yield` 表达式才会求值。

```
function* gen() {
  yield  123 + 456;
}
```
复制代码

1. `yield`表达式只能用在`Generator`函数里面，用在其他地方都会报错。

```
(function (){
  yield 1;
})()
// SyntaxError: Unexpected number
复制代码
```

1. `yield`表达式如果用在另一个表达式之中，必须放在圆括号里面。用作函数参数或放在赋值表达式的右边，可以不加括号。

```
function* demo() {
  console.log('Hello' + yield); // SyntaxError
  console.log('Hello' + yield 123); // SyntaxError

  console.log('Hello' + (yield)); // OK
  console.log('Hello' + (yield 123)); // OK
}
// 参数和表达式右边
function* demo() {
  foo(yield 'a', yield 'b'); // OK
  let input = yield; // OK
}
复制代码
```

与Iterator接口和for...of 循环的关系

1. 与`Iterator`接口的关系

- 上一弹ES6常用但被忽略的方法（第五弹Promise和Iterator）说过，任意一个对象的`Symbol.iterator`方法，等于该对象的遍历器生成函数，调用该函数会返回该对象的一个遍历器对象。
- `Generator`函数就是遍历器生成函数，可以把`Generator`赋值给对象的`Symbol.iterator`属性，从而使得该对象具有`Iterator`接口。

```
var myIterable = {};
myIterable[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};
[...myIterable] // [1, 2, 3]
复制代码
```

- `Generator`函数执行后，返回一个遍历器对象。该对象本身也具有`Symbol.iterator`属性，执行后返回自身。

```
function* gen(){
  // some code
}
var g = gen();
g[Symbol.iterator]() === g
// true
复制代码
```

1. `for...of` 循环

- **`for...of` 循环可以自动遍历 `Generator` 函数运行时生成的 `Iterator` 对象，且此时不再需要调用 `next` 方法。*一旦 `next` 方法的返回对象的 `done` 属性为 `true`，`for...of` 循环就会中止，且不包含该返回对象。***

```
function* numbers() {
  yield 1;
  yield 2;
  return 3;
}

for (let v of numbers()) {
  console.log(v);
}
// 1 2
复制代码
```

1. 除了 `for...of` 循环

- 扩展运算符（`...`）、解构赋值和 `Array.from` 方法内部调用的，都是遍历器接口。它们都可以将 `Generator` 函数返回的 `Iterator` 对象，作为参数。

```
// 扩展运算符
[...numbers()] // [1, 2]

// Array.from 方法
Array.from(numbers()) // [1, 2]

// 解构赋值
let [x, y] = numbers();
x // 1
y // 2
复制代码
```

next 方法的参数

- `yield` 表达式本身没有返回值，或者说总是返回 `undefined`。`next` 方法可以带一个参数，该参数就会被当作**上一个 `yield` 表达式的返回值。**

```
function* foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z);
}

var a = foo(5);
a.next() // Object{value:6, done:false}
a.next() // Object{value:NaN, done:false}
a.next() // Object{value:NaN, done:true}

var b = foo(5);
b.next() // { value:6, done:false }
b.next(12) // { value:8, done:false }
b.next(13) // { value:42, done:true }
复制代码
```

- 上面代码中，第二次运行`next`方法的时候不带参数，导致`y`的值等于`2 * undefined`（即`NaN`），除以`3`以后还是`NaN`，因此返回对象的`value`属性也等于`NaN`。第三次运行`next`方法的时候不带参数，所以`z`等于`undefined`，返回对象的`value`属性等于`5 + NaN + undefined`，即`NaN`。
- 如果向`next`方法提供参数，返回结果就完全不一样了。上面代码第一次调用b的`next`方法时，返回`x+1`的值`6`；第二次调用`next`方法，将上一次`yield`表达式的值设为`12`，因此`y`等于`24`，返回`y / 3`的值`8`；第三次调用`next`方法，将上一次`yield`表达式的值设为`13`，因此`z`等于`13`，这时`x`等于`5`，`y`等于`24`，所以`return`语句的值等于`42`。
- **由于`next`方法的参数表示上一个`yield`表达式的返回值，所以在第一次使用`next`方法时，传递参数是无效的。**

next()、throw()、return()

共同点

- 它们的作用都是让`Generator`函数恢复执行，并且使用不同的语句替换`yield`表达式。

1. `next()`是将上一个`yield`表达式替换成一个值。

```
const g = function* (x, y) {
  let result = yield x + y;
  return result;
};

const gen = g(1, 2);
gen.next(); // Object {value: 3, done: false}

gen.next(1); // Object {value: 1, done: true}
// 相当于将 let result = yield x + y
// 替换成 let result = 1;
复制代码
```

1. `throw()`是将`yield`表达式替换成一个`throw`语句。

```
gen.throw(new Error('出错了')); // Uncaught Error: 出错了
// 相当于将 let result = yield x + y
// 替换成 let result = throw(new Error('出错了'));
复制代码
```

1. `return()`是将`yield`表达式替换成一个`return`语句。

```
gen.return(2); // Object {value: 2, done: true}
// 相当于将 let result = yield x + y
// 替换成 let result = return 2;
复制代码
```

不同点

1. `Generator.prototype.throw()`

    - `Generator`函数返回的遍历器对象，`throw`方法可以在函数体外抛出错误，然后在`Generator`函数体内捕获。**`throw`方法可以接受一个参数，该参数会被`catch`语句接收，建议抛出`Error`对象的实例。**

```
var g = function* () {
  try {
    yield;
  } catch (e) {
    console.log(e);
  }
};

var i = g();
i.next();
i.throw(new Error('出错了! '));
// Error: 出错了! (…)
复制代码
```

- 不要混淆遍历器对象的`throw`方法和全局的`throw`命令。上面代码的错误，是用遍历器对象的`throw`方法抛出的，而不是用`throw`命令抛出的。**后者只能被函数体外的`catch`语句捕获。**
- **如果`Generator`函数内部没有部署`try...catch`代码块，那么`throw`方法抛出的错误，将被外部`try...catch`代码块捕获。**

```
var g = function* () {
  while (true) {
    yield;
    console.log('内部捕获', e);
  }
};

var i = g();
i.next();

try {
  i.throw('a');
  i.throw('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 外部捕获 a
复制代码
```

- **如果`Generator`函数内部和外部，都没有部署`try...catch`代码块，那么程序将报错，直接中断执行。**

```
var gen = function* gen(){
  yield console.log('hello');
  yield console.log('world');
}

var g = gen();
g.next();
g.throw();
// hello
// Uncaught undefined
复制代码
```

- **`throw`方法抛出的错误要被内部捕获，前提是必须至少执行过一次`next`方法。**
  `g.throw(1)`执行时，`next`方法一次都没有执行过。这时，抛出的错误不会被内部捕获，而是直接在外部抛出，导致程序出错。

```
function* gen() {
  try {
    yield 1;
  } catch (e) {
    console.log('内部捕获');
  }
}

var g = gen();
g.throw(1);
// Uncaught 1
```
复制代码

- **`Generator`函数体外抛出的错误，可以在函数体内捕获；反过来，`Generator`函数体内抛出的错误，也可以被函数体外的`catch`捕获。**

```
function* foo() {
  var x = yield 3;
  var y = x.toUpperCase();
  yield y;
}
var it = foo();
it.next(); // { value:3, done:false }
try {
  it.next(42);
} catch (err) {
  console.log(err);
}
```
复制代码

- 数值是没有toUpperCase方法的，所以会抛出一个 TypeError 错误，被函数体外的catch捕获。

2. `Generator.prototype.return()`

- `Generator`函数返回的遍历器对象，`return`方法可以返回给定的值，并且终结遍历`Generator`函数。`return`方法调用时，不提供参数，则返回值的`value`属性为`undefined`。

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

var g = gen();

g.next()        // { value: 1, done: false }
g.return() // { value: undefined, done: true }
```
复制代码

- 如果 `Generator` 函数内部有 `try...finally` 代码块，且正在执行 `try` 代码块，那么 `return` 方法会导致立刻进入 `finally` 代码块，执行完以后，整个函数才会结束。

```
function* numbers () {
  yield 1;
  try {
    yield 2;
    yield 3;
  } finally {
    yield 4;
    yield 5;
  }
  yield 6;
}
var g = numbers();
g.next() // { value: 1, done: false }
g.next() // { value: 2, done: false }
g.return(7) // { value: 4, done: false }
g.next() // { value: 5, done: false }
g.next() // { value: 7, done: true }
复制代码
```

`yield*` 表达式

- **`yield\*` 表达式用来在一个 `Generator` 函数里面执行另一个 `Generator` 函数。**

```
function* foo() {
  yield 'a';
  yield 'b';
}
function* bar() {
  yield 'x';
  yield* foo();
  yield 'y';
}
for (let v of bar()){
  console.log(v);
}
// "x"
// "a"
// "b"
// "y"
复制代码
```

- 如果 `yield*` 后面跟着一个数组，由于数组原生支持遍历器，因此就会遍历数组成员。**`yield` 命令后面如果不加星号，返回的是整个数组，加了星号就表示返回的是数组的遍历器对象。**

```
function* gen(){
  yield* ["a", "b", "c"];
}
gen().next() // { value:"a", done:false }
复制代码
```

- 任何数据结构只要有 `Iterator` 接口，就可以被 `yield*` 遍历。

```
let read = (function* () {
  yield 'hello';
  yield* 'hello';
})();

read.next().value // "hello"
read.next().value // "h"
复制代码
```

- 如果被代理的 `Generator` 函数有 `return` 语句，那么就可以向代理它的 `Generator` 函数返回数据。下面例子中函数 `foo` 的 `return` 语句，向函数 `bar` 提供了返回值。

```
function* foo() {
  yield 2;
  yield 3;
  return "foo";
}

function* bar() {
  yield 1;
  var v = yield* foo();
  console.log("v: " + v);
  yield 4;
}

var it = bar();

it.next()
// {value: 1, done: false}
it.next()
// {value: 2, done: false}
it.next()
// {value: 3, done: false}
it.next();
// "v: foo"
// {value: 4, done: false}
it.next()
// {value: undefined, done: true}
复制代码
```

作为对象属性的 Generator 函数写法

```
let obj = {
  * myGeneratorMethod() {
    ...
  }
};
// 等价
let obj = {
  myGeneratorMethod: function* () { // *位置可以在function关键字和括号之间任意位置
    // ...
  }
};
复制代码
```

Generator 函数的this

```
function* g() {}
g.prototype.hello = function () {
  return 'hi!';
};
let obj = g();
obj instanceof g // true
obj.hello() // 'hi!'
复制代码
```

- `Generator` 函数`g`返回的遍历器`obj`，是`g`的实例，而且继承了`g.prototype`。但是，把`g`当作普通的构造函数，并不会生效，因为`g`返回的总是遍历器对象，而不是`this`对象。
- **

  ```
  Generator
  ```

  函数也不能跟

  ```
  new
  ```

  命令一起用，会报错。

  ```
  function* F() {
    yield this.x = 2;
    yield this.y = 3;
  }

  new F()
  // TypeError: F is not a constructor
  复制代码
  ```

  - 变通方法。首先，生成一个空对象，使用`call`方法绑定 `Generator` 函数内部的`this`。构造函数调用以后，这个空对象就是 `Generator` 函数的实例对象。

  ```
  function* F() {
    this.a = 1;
    yield this.b = 2;
    yield this.c = 3;
  }
  var obj = {};
  var f = F.call(obj);

  f.next();  // Object {value: 2, done: false}
  f.next();  // Object {value: 3, done: false}
  f.next();  // Object {value: undefined, done: true}

  obj.a // 1
  obj.b // 2
  obj.c // 3
  复制代码
  ```

- 上面执行的是遍历器对象`f`，但是生成的对象实例是`obj`，将`obj`换成`F.prototype`。再将F改成构造函数，就可以对它执行`new`命令了。

```
function* gen() {
  this.a = 1;
  yield this.b = 2;
  yield this.c = 3;
}
function F() {
  return gen.call(gen.prototype);
}

var f = new F();
f.next();  // Object {value: 2, done: false}
f.next();  // Object {value: 3, done: false}
f.next();  // Object {value: undefined, done: true}

f.a // 1
f.b // 2
f.c // 3
复制代码
```

应用

1. 取出嵌套数组的所有成员

```
function* iterTree(tree) {
  if (Array.isArray(tree)) {
    for(let i=0; i < tree.length; i++) {
      yield* iterTree(tree[i]);
    }
  } else {
    yield tree;
  }
}

const tree = [ 'a', ['b', 'c'], ['d', 'e'] ];

for(let x of iterTree(tree)) {
  console.log(x);
}
// a
// b
// c
// d
// e
复制代码
```

2. 遍历完全二叉树。

```
// 下面是二叉树的构造函数，
// 三个参数分别是左树、当前节点和右树
function Tree(left, label, right) {
  this.left = left;
  this.label = label;
  this.right = right;
```

```
}

// 下面是中序（inorder）遍历函数。
// 由于返回的是一个遍历器，所以要用generator函数。
// 函数体内采用递归算法，所以左树和右树要用yield*遍历
function* inorder(t) {
  if (t) {
    yield* inorder(t.left);
    yield t.label;
    yield* inorder(t.right);
  }
}

// 下面生成二叉树
function make(array) {
  // 判断是否为叶节点
  if (array.length == 1) return new Tree(null, array[0], null);
  return new Tree(make(array[0]), array[1], make(array[2]));
}
let tree = make([[['a'], 'b', ['c']], 'd', [['e'], 'f', ['g']]]);

// 遍历二叉树
var result = [];
for (let node of inorder(tree)) {
  result.push(node);
}

result
// ['a', 'b', 'c', 'd', 'e', 'f', 'g']
复制代码
```

3. `Ajax`

   的异步操作

   - 通过 `Generator` 函数部署 `Ajax` 操作，可以用同步的方式表达。**注意，`makeAjaxCall` 函数中的 `next` 方法，必须加上 `response` 参数，因为 `yield` 表达式，本身是没有值的，总是等于 `undefined`。**

```
function* main() {
  var result = yield request("http://some.url");
  var resp = JSON.parse(result);
    console.log(resp.value);
}

function request(url) {
  makeAjaxCall(url, function(response){
    it.next(response);
  });
}

var it = main();
it.next();
复制代码
```

4. 逐行读取文本文件。

```
function* numbers() {
  let file = new FileReader("numbers.txt");
  try {
    while(!file.eof) {
      yield parseInt(file.readLine(), 10);
    }
  } finally {
    file.close();
  }
}
```
复制代码

5. 控制流管理

- 如果有一个多步操作非常耗时，采用回调函数，可能会写成下面这样。

```
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        // Do something with value4
      });
    });
  });
});
```
复制代码

- 利用`for...of`循环会自动依次执行`yield`命令的特性，提供一种更一般的控制流管理的方法。

```
let steps = [step1Func, step2Func, step3Func];

function* iterateSteps(steps){
  for (var i=0; i< steps.length; i++){
    var step = steps[i];
    yield step();
  }
}
```
复制代码

6. 部署

```
Iterator
```

接口

- 利用`Generator`函数，可以在任意对象上部署`Iterator`接口。

```
function* iterEntries(obj) {
  let keys = Object.keys(obj);
  for (let i=0; i < keys.length; i++) {
    let key = keys[i];
    yield [key, obj[key]];
  }
}
```

```
let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {
  console.log(key, value);
}

// foo 3
// bar 7
```
复制代码

7. 作为数据结构
    - Generator 可以看作是数据结构，更确切地说，可以看作是一个数组结构。

```
function* doStuff() {
  yield fs.readFile.bind(null, 'hello.txt');
  yield fs.readFile.bind(null, 'world.txt');
  yield fs.readFile.bind(null, 'and-such.txt');
}

for (task of doStuff()) {
  // task是一个函数，可以像回调函数那样使用它
}
```
复制代码

8. 异步应用
    - ES6-Generator 函数的异步应用

async 函数

- ES6-async 函数

介绍

- **`async` 函数是 `Generator` 函数的语法糖。**

```
const fs = require('fs');

const readFile = function (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function(error, data) {
      if (error) return reject(error);
      resolve(data);
    });
  });
};
// Generator函数写法
const gen = function* () {
  const f1 = yield readFile('/etc/fstab');
  const f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
// async 函数写法
const asyncReadFile = async function () {
```

```
  const f1 = await readFile('/etc/fstab');
  const f2 = await readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
复制代码
```

- `async`

  函数对

  `Generator`

  函数的改进:

  1. `Generator` 函数的执行必须靠执行器，所以才有了 `co` 模块（基于 `ES6` 的 `Generator` 和 `yield` ，让我们能用同步的形式编写异步代码），而 `async` 函数自带执行器。 **（`async` 函数的执行，与普通函数一模一样，只要一行。）**

  ```
  asyncReadFile();
  复制代码
  ```

  - `Generator` 函数，需要调用 `next` 方法，或者用 `co` 模块，才能真正执行，得到最后结果。

  ```
  const gg = gen();
  gg.next()
  复制代码
  ```

  1. 更好的语义。 `async` 和 `await` ，比起 `*` 和 `yield` ，语义更清楚。 `async` 表示函数里有异步操作， `await` 表示紧跟在后面的表达式需要等待结果。
  2. 更广的适用性。 `co` 模块约定， `yield` 命令后面只能是 `Thunk` 函数或 `Promise` 对象，而 `async` 函数的 `await` 命令后面，可以是 `Promise` 对象和原始类型的值（数值、字符串和布尔值，但这时会自动转成立即 `resolved` 的 `Promise` 对象）。
  3. 返回值是 `Promise` 。 `async` 函数的返回值是 `Promise` 对象，这比 `Generator` 函数的返回值是 `Iterator` 对象方便很多。

使用

1. 基操
   - `async` 函数返回一个 `Promise` 对象，可以使用 `then` 方法添加回调函数。 **`async` 函数内部 `return` 语句返回的值，会成为 `then` 方法回调函数的参数。** `async` 函数内部抛出错误，会导致返回的 `Promise` 对象变为 `reject` 状态。抛出的错误对象会被 `catch` 方法回调函数接收到。

```
async function getName() {
  return 'detanx';
}
getName().then(val => console.log(val))
// "detanx"

// 抛出错误
async function f() {
  throw new Error('出错了');
```

```
}

f().then(
  v => console.log(v),
  e => console.log(e)
)
// Error: 出错了
复制代码
```

2. 使用方式

```
// 函数声明
async function foo() {}

// 函数表达式
const foo = async function () {};

// 对象的方法
let obj = { async foo() {} };
obj.foo().then(...)

// 箭头函数
const foo = async () => {};

// Class 的方法
class Storage {
  constructor() {
    this.cachePromise = caches.open('avatars');
  }

  async getAvatar(name) {
    const cache = await this.cachePromise;
    return cache.match(`/avatars/${name}.jpg`);
  }
}

const storage = new Storage();
storage.getAvatar('jake').then(…);
复制代码
```

3. 状态变化

- `async`函数返回的 `Promise` 对象，**必须等到内部所有`await`命令后面的 `Promise` 对象执行完，才会发生状态改变，除非遇到`return`语句或者抛出错误。**

4. `await`

命令

- 正常情况下，`await`命令后面是一个 `Promise` 对象，返回该对象的结果。如果不是 `Promise` 对象，就直接返回对应的值。

```
async function getName() {
  return 'detanx';
}
getName().then(val => console.log(val))
// "detanx"
复制代码
```

- 另一种情况是，`await`命令后面是一个`thenable`对象（定义了`then`方法的对象），那么`await`会将其等同于`Promise`对象。

```
class Sleep {
  constructor(timeout) {
    this.timeout = timeout;
  }
  then(resolve, reject) {
    const startTime = Date.now();
    setTimeout(
      () => resolve(Date.now() - startTime),
      this.timeout
    );
  }
}

(async () => {
  const sleepTime = await new Sleep(1000);
  console.log(sleepTime);
})();
// 1000
复制代码
```

- 任何一个`await`语句后面的`Promise`对象变为`reject`状态，那么整个`async`函数都会中断执行。

```
async function f() {
  await Promise.reject('出错了');
  await Promise.resolve('hello world'); // 不会执行
}
复制代码
```

  - 前一个异步操作失败，也不要中断后面的异步操作。

  1. `await`放在`try...catch`里面

```
async function f() {
  try {
    await Promise.reject('出错了');
  } catch(e) {
  }
  return await Promise.resolve('detanx');
}

f()
.then(v => console.log(v))
// detanx
复制代码
```

1. `await`后面的`Promise`对象再跟一个`catch`方法，处理前面可能出现的错误。

```
async function f() {
  await Promise.reject('出错了')
    .catch(e => console.log(e));
  return await Promise.resolve('detanx');
}

f()
.then(v => console.log(v))
// 出错了
// detanx
复制代码
```

5. 使用注意点

1. `await`命令后面的`Promise`对象，运行结果可能是`rejected`，所以最好把`await`命令放在`try...catch`代码块中。
2. 多个`await`命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

```
// 写法一
let [foo, bar] = await Promise.all([getFoo(), getBar()]);

// 写法二
let fooPromise = getFoo();
let barPromise = getBar();
let foo = await fooPromise;
let bar = await barPromise;
复制代码
```

1. `await`命令只能用在`async`函数之中，如果用在普通函数，就会报错。正确的写法是采用`for`循环或者使用数组的`reduce`方法。希望多个请求并发执行，可以使用`Promise.all`或者`Promise.allSettled`方法。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];

  // 报错，await的上一级函数不是async函数
  docs.forEach(function (doc) {
    await db.post(doc);
  });
  => for循环
  for (let doc of docs) {
    await db.post(doc);
  }
  => 数组的reduce方法
  await docs.reduce(async (_, doc) => {
    await _;
    await db.post(doc);
  }, undefined);
}
复制代码
```

1. `async`函数可以保留运行堆栈。

```
const a = () => {
  b().then(() => c());
};
复制代码
```

- 上面代码中，函数 a 内部运行了一个异步任务 b()。当 b() 运行的时候，函数 a() 不会中断，而是继续执行。等到 b() 运行结束，可能 a() 早就运行结束，b() 所在的上下文环境已经消失。如果 b() 或 c() 报错，错误堆栈将不包括 a()。
- 现在将这个例子改成 async 函数。

```
const a = async () => {
  await b();
  c();
};
复制代码
```

- 上面代码中，b() 运行的时候，a() 是暂停执行，上下文环境都保存着。一旦 b() 或 c() 报错，错误堆栈将包括 a()。

实现

- async 函数的实现原理，就是将 Generator 函数和自动执行器，包装在一个函数里。

```
async function fn(args) {
  // ...
}
// 等同于
function fn(args) {
  return spawn(function* () {
    // ...
  });
}
复制代码
```

- 所有的 async 函数都可以写成上面的第二种形式，其中的 spawn 函数就是自动执行器。下面 spawn 函数的实现，基本就是前文自动执行器的翻版。

```
function spawn(genF) {
  return new Promise(function(resolve, reject) {
    const gen = genF();
    function step(nextF) {
      let next;
      try {
        next = nextF();
      } catch(e) {
        return reject(e);
      }
      if(next.done) {
        return resolve(next.value);
      }
      Promise.resolve(next.value).then(function(v) {
        step(function() { return gen.next(v); });
      }, function(e) {
        step(function() { return gen.throw(e); });
```

```
      });
    }
    step(function() { return gen.next(undefined); });
  });
}
```
复制代码

异步处理方法的比较

- 传统方法，`ES6` 诞生以前，异步编程的方法，大概有下面四种。
    1. 回调函数
    2. 事件监听
    3. 发布/订阅
    4. `Promise` 对象
- 通过一个例子，主要来看 `Promise`、`Generator` 函数与`async` 函数的比较，其他的不太熟悉的可以自己去查相关的资料。
- 假定某个 `DOM` 元素上面，部署了一系列的动画，前一个动画结束，才能开始后一个。如果当中有一个动画出错，就不再往下执行，返回上一个成功执行的动画的返回值。
    1. `Promise` 的写法

```
function chainAnimationsPromise(elem, animations) {
  // 变量ret用来保存上一个动画的返回值
  let ret = null;
  // 新建一个空的Promise
  let p = Promise.resolve();

  // 使用then方法，添加所有动画
  for(let anim of animations) {
    p = p.then(function(val) {
      ret = val;
      return anim(elem);
    });
  }
  // 返回一个部署了错误捕捉机制的Promise
  return p.catch(function(e) {
    /* 忽略错误，继续执行 */
  }).then(function() {
    return ret;
  });
}
```
复制代码

    - 一眼看上去，代码完全都是 `Promise` 的 `API`（`then`、`catch`等等），操作本身的语义反而不容易看出来。
    1. `Generator` 函数的写法。

```
function chainAnimationsGenerator(elem, animations) {

  return spawn(function*() {
    let ret = null;
    try {
      for(let anim of animations) {
        ret = yield anim(elem);
      }
```

```
    } catch(e) {
      /* 忽略错误，继续执行 */
    }
    return ret;
  });
}
```
复制代码

- 语义比 `Promise` 写法更清晰，用户定义的操作全部都出现在`spawn`函数的内部。问题在于，必须有一个任务运行器，自动执行 `Generator` 函数，上面代码的`spawn`函数就是自动执行器，它返回一个 `Promise` 对象，而且必须保证`yield`语句后面的表达式，必须返回一个 `Promise`。

1. `async` 函数的写法。

```
async function chainAnimationsAsync(elem, animations) {
  let ret = null;
  try {
    for(let anim of animations) {
      ret = await anim(elem);
    }
  } catch(e) {
    /* 忽略错误，继续执行 */
  }
  return ret;
}
```
复制代码

- `Async` 函数的实现最简洁，最符合语义，几乎没有语义不相关的代码。

- **所有的异步处理方法存在即合理，没有那个最好，只有最合适，在处理不同的实际情况时，我们选择最适合的处理方法即可。**

实例

- 实际开发中，经常遇到一组异步操作，需要按照顺序完成。比如，依次远程读取一组

```
URL
```

，然后按照读取的顺序输出结果。

1. `Promise` 的写法。

```
function logInOrder(urls) {
  // 远程读取所有URL
  const textPromises = urls.map(url => {
    return fetch(url).then(response => response.text());
  });

  // 按次序输出
  textPromises.reduce((chain, textPromise) => {
    return chain.then(() => textPromise)
      .then(text => console.log(text));
  }, Promise.resolve());
}
```
复制代码

- 上面代码使用`fetch`方法，同时远程读取一组`URL`。每个`fetch`操作都返回一个`Promise`对象，放入`textPromises`数组。然后，`reduce`方法依次处理每个`Promise`对象，然后使用`then`，将所有`Promise`对象连起来，因此就可以依次输出结果。

1. `async`函数实现。

```
async function logInOrder(urls) {
  // 并发读取远程URL
  const textPromises = urls.map(async url => {
    const response = await fetch(url);
    return response.text();
  });

  // 按次序输出
  for (const textPromise of textPromises) {
    console.log(await textPromise);
  }
}
复制代码
```

- 上面代码中，虽然`map`方法的参数是`async`函数，但它是并发执行的，因为只有`async`函数内部是继发执行，外部不受影响。后面的`for..of`循环内部使用了`await`，因此实现了按顺序输出。

顶级`await`

- 现在

```
await
```

命令只能出现在

```
async
```

函数内部，否则都会报错。有一个

语法提案

（目前提案处于

```
Status: Stage 3
```

），允许在模块的顶层独立使用

```
await
```

命令，使得上面那行代码不会报错了。

这个提案的目的，是借用`await`解决模块异步加载的问题。

1. 模块`awaiting.js`的输出值`output`，取决于异步操作。

```
// awaiting.js
let output;
(async function1 main() {
  const dynamic = await import(someMission);
  const data = await fetch(url);
  output = someProcess(dynamic.default, data);
})();
export { output };
复制代码
```

1. 加载`awaiting.js`模块

```
// usage.js
import { output } from "./awaiting.js";

function outputPlusValue(value) { return output + value }
console.log(outputPlusValue(100));
setTimeout(() => console.log(outputPlusValue(100), 1000);
复制代码
```

- `outputPlusValue()`的执行结果，完全取决于执行的时间。如果`awaiting.js`里面的异步操作没执行完，加载进来的`output`的值就是`undefined`。

- 目前的解决方法，就是让原始模块输出一个

```
Promise
```

对象，从这个

```
Promise
```

对象判断异步操作有没有结束。

```
// usage.js
import promise, { output } from "./awaiting.js";

function outputPlusValue(value) { return output + value }
promise.then(() => {
  console.log(outputPlusValue(100));
  setTimeout(() => console.log(outputPlusValue(100), 1000);
});
复制代码
```

- 上面代码中，将`awaiting.js`对象的输出，放在`promise.then()`里面，这样就能保证异步操作完成以后，才去读取`output`。
- 这种写法比较麻烦，等于要求模块的使用者遵守一个额外的使用协议，按照特殊的方法使用这个模块。一旦你忘了要用`Promise`加载，只使用正常的加载方法，依赖这个模块的代码就可能出错。而且，如果上面的`usage.js`又有对外的输出，等于这个依赖链的所有模块都要使用`Promise`加载。

- 顶层的await命令，它保证只有异步操作完成，模块才会输出值。

```
// awaiting.js
const dynamic = import(someMission);
const data = fetch(url);
export const output = someProcess((await dynamic).default, await data);

// usage.js
import { output } from "./awaiting.js";
function outputPlusValue(value) { return output + value }

console.log(outputPlusValue(100));
setTimeout(() => console.log(outputPlusValue(100), 1000);
```
复制代码

  - 上面代码中，两个异步操作在输出的时候，都加上了`await`命令。只有等到异步操作完成，这个模块才会输出值。

- 顶层

```
await
```

的一些使用场景。

```
// import() 方法加载
const strings = await import(`/i18n/${navigator.language}`);

// 数据库操作
const connection = await dbConnector();

// 依赖回滚
let jQuery;
try {
  jQuery = await import('https://cdn-a.com/jQuery');
} catch {
  jQuery = await import('https://cdn-b.com/jQuery');
}
```
复制代码

- 如果加载多个包含顶层`await`命令的模块，加载命令是同步执行的。

```
// x.js
console.log("X1");
await new Promise(r => setTimeout(r, 1000));
console.log("X2");

// y.js
console.log("Y");

// z.js
import "./x.js";
import "./y.js";
console.log("Z");
```
复制代码

  - 上面代码有三个模块，最后的`z.js`加载`x.js`和`y.js`，打印结果是`X1`、`Y`、`X2`、`Z`。这说明，`z.js`并没有等待`x.js`加载完成，再去加载`y.js`。

- 顶层的`await`命令有点像，交出代码的执行权给其他的模块加载，等异步操作完成后，再拿回执行权，继续向下执行。

Class 函数

- ES6-Class

简介

- 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。

```
// 传统写法
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype.toString = function () {
  return '(' + this.x + ', ' + this.y + ')';
};
var p = new Point(1, 2);

// class 写法
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}
复制代码
```

- **定义类的方法的时候，前面不需要加上`function`这个关键字，直接把函数定义放进去了就可以了。方法之间不需要逗号分隔，加了会报错。** 类的所有方法都定义在类的`prototype`属性上面。

```
class Point {
  constructor() {...}
  toString() {...}
  toValue() {...}
}
// 等同于
Point.prototype = {
  constructor() {},
  toString() {},
  toValue() {},
};

typeof Point // "function"
Point === Point.prototype.constructor // true
复制代码
```

- 类的内部所有定义的方法，都是不可枚举的（non-enumerable）。这与 ES5 的行为不一致。

```
// es6
```

```
class Point {
  constructor(x, y) {...}
  toString() {...}
}

Object.keys(Point.prototype)
// []
Object.getOwnPropertyNames(Point.prototype)
// ["constructor","toString"]

// es5
var Point = function (x, y) {...};
Point.prototype.toString = function() {...};

Object.keys(Point.prototype)
// ["toString"]
Object.getOwnPropertyNames(Point.prototype)
// ["constructor","toString"]
```
复制代码

constructor 方法

- `constructor`方法是类的默认方法，通过`new`命令生成对象实例时，自动调用该方法。**一个类必须有`constructor`方法，如果没有显式定义，一个空的`constructor`方法会被默认添加。**
- `constructor`方法默认返回实例对象（即`this`），完全可以指定返回另外一个对象。

```
class Foo {
  constructor() {
    return Object.create(Object);
  }
}

new Foo() instanceof Foo // false
new Foo() instanceof Object // true
```
复制代码

取值函数（getter）和存值函数（setter）

- 与`ES5`一样，在"类"的内部可以使用`get`和`set`关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。**存值函数和取值函数是设置在属性的`Descriptor`对象上的。**

```
class MyClass {
  constructor() {...}
  get prop() {
    return 'getter';
  }
  set prop(value) {
    console.log('setter: '+value);
  }
}

let inst = new MyClass();
inst.prop = 123; // setter: 123
inst.prop; // 'getter'

const descriptor = Object.getOwnPropertyDescriptor(
```

```
  MyClass.prototype, "prop"
);

"get" in descriptor  // true
"set" in descriptor  // true
```
复制代码

表达式

1. 属性表达式

```
let methodName = 'getArea';

class Square {
  constructor(length) {...}
  [methodName]() {...}
}
```
复制代码

2. Class 表达式

```
const MyClass = class Me {
  getClassName() {
    return Me.name;
  }
};
```
复制代码

   - Class 表达式，可以写出立即执行的 Class。

```
let person = new class {
  constructor(name) {
    this.name = name;
  }
  sayName() {
    console.log(this.name);
  }
}('detanx');
person.sayName(); // "detanx"
```
复制代码

注意点

1. 严格模式
   - 类和模块的内部，默认就是严格模式，所以不需要使用`use strict`指定运行模式。
2. 不存在提升
   - 使用在前，定义在后，这样会报错。

```
new Foo(); // ReferenceError
class Foo {}
```
复制代码

3. `name`属性

- 由于本质上，`ES6`的类只是`ES5`的构造函数的一层包装，所以函数的许多特性都被`Class`继承，包括`name`属性。

```
class Point {}
Point.name // "Point"
复制代码
```

- `name`属性总是返回紧跟在`class`关键字后面的类名。

4. `Generator`方法
   - 如果某个方法之前加上星号（`*`），就表示该方法是一个`Generator`函数。
5. `this`的指向
   - 类的方法内部如果含有`this`，它默认指向类的实例。一旦单独使用该方法，很可能报错。

```
class Logger {
  printName(name = 'there') {
    this.print(`Hello ${name}`);
  }
  print(text) {
    console.log(text);
  }
}

const logger = new Logger();
const { printName } = logger;
printName(); // TypeError: Cannot read property 'print' of undefined
复制代码
```

静态方法

- 类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，**加上`static`关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。**

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}
Foo.classMethod() // 'hello'
var foo = new Foo();
foo.classMethod()
// TypeError: foo.classMethod is not a function
复制代码
```

- **如果静态方法包含`this`关键字，这个`this`指的是类，而不是实例。**
- **父类的静态方法，可以被子类继承。**

```
class Foo {
  static classMethod() {
    return 'detanx';
  }
}
class Bar extends Foo {
}
Bar.classMethod() // 'detanx'
复制代码
```

- **静态方法也是可以从`super`对象上调用的。**

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}
class Bar extends Foo {
  static classMethod() {
    return super.classMethod() + ', detanx';
  }
}
Bar.classMethod() // "hello, detanx"
复制代码
```

实例属性的新写法

- 实例属性除了定义在`constructor()`方法里面的this上面，也可以定义在类的最顶层。

```
class IncreasingCounter {
  _count = 0
  => // 或写在constructor
  // constructor() {
  //  this._count = 0;
  // }
  get value() {
    console.log('Getting the current value!');
    return this._count;
  }
  increment() {
    this._count++;
  }
}
复制代码
```

静态属性

- 静态属性指的是`Class`本身的属性，即`Class.propName`，而不是定义在实例对象（`this`）上的属性。

```
class Foo { }
Foo.prop = 1;
Foo.prop // 1
复制代码
```

- 现在有一个 提案（目前处于`stage 3`） 提供了类的静态属性，写法是在实例属性的前面，加上`static`关键字。

```
// 新写法
class Foo {
  static prop = 1;
}
复制代码
```

私有方法和私有属性

1. 现有的解决方案
    - 私有方法和私有属性，是只能在类的内部访问的方法和属性，外部不能访问。这是常见需求，有利于代码的封装，但 `ES6` 不提供，只能通过变通方法模拟实现。
    1. 做法是在命名上加以区别。
        - 开发时约定以什么开头的属性或方法为私有属性。例如以 `_` 开头或者 `$` 开头的为私有。
    2. 将私有方法移出模块

        ```
        class Widget {
          foo (baz) {
            bar.call(this, baz);
          }
        }
        function bar(baz) {
          return this.snaf = baz;
        }
        复制代码
        ```

    3. 利用

        ```
        Symbol
        ```

        值的唯一性，将私有方法的名字命名为一个

        ```
        Symbol
        ```

        值。

        ES6常用但被忽略的方法（第三弹Symbol、Set 和 Map ）之Symbol的应用

        ```
        const bar = Symbol('bar');
        const snaf = Symbol('snaf');

        export default class myClass{
          // 公有方法
          foo(baz) {
            this[bar](baz);
          }
          // 私有方法
          [bar](baz) {
            return this[snaf] = baz;
          }
        ```

```
};
复制代码
```

- 也不是绝对不行，`Reflect.ownKeys()`依然可以拿到它们。

```
const inst = new myClass();
Reflect.ownKeys(myClass.prototype)
// [ 'constructor', 'foo', Symbol(bar) ]
复制代码
```

2. 私有属性的提案

- 目前，有一个 提案（`Stage 3`），为`class`加了私有属性和方法。方法是在属性名和方法之前，使用`#`表示。私有属性也可以设置`getter`和`setter`方法。

```
class Counter {
  #xValue = 0;
  constructor() {
    super();
    // ...
  }
  get #x() { return #xValue; }
  set #x(value) {
    this.#xValue = value;
  }
}
复制代码
```

- 私有属性和私有方法前面，也可以加上`static`关键字，表示这是一个静态的私有属性或私有方法。

`new.target`属性

- `new`是从构造函数生成实例对象的命令。`ES6`为`new`命令引入了一个`new.target`属性，该属性一般用在构造函数之中，返回`new`命令作用于的那个构造函数。**如果构造函数不是通过`new`命令或`Reflect.construct()`调用的**，`new.target`会返回`undefined`，因此这个属性可以用来确定构造函数是怎么调用的。**`Class`内部调用`new.target`，返回当前`Class`。**

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    this.length = length;
    this.width = width;
  }
}
var obj = new Rectangle(3, 4); // 输出 true
复制代码
```

- **子类继承父类时，`new.target`会返回子类。** 利用这个特点，可以写出不能独立使用、必须继承后才能使用的类。

```
class Shape {
  constructor() {
    if (new.target === Shape) {
```

```
      throw new Error('本类不能实例化');
    }
  }
}
class Rectangle extends Shape {
  constructor(length, width) {
    super();
  }
}
var x = new Shape();  // 报错
var y = new Rectangle(3, 4);  // 正确
复制代码
```

继承

- ES6-Class 继承

注意点

1. 子类继承父类需要先调用

   ```
   super
   ```

   方法。

   ```
   class Point { /* ... */ }
   class ColorPoint extends Point {
     constructor() {
     }
   }
   let cp = new ColorPoint(); // ReferenceError
   复制代码
   ```

   - `ColorPoint`继承了父类`Point`，但是它的构造函数没有调用`super`方法，导致新建实例时报错。
2. 在子类的构造函数中，只有调用

   ```
   super
   ```

   之后，才可以使用

   ```
   this
   ```

   关键字，否则会报错。

```
class Point {
  constructor(x) {
    this.x = x;
  }
}
class ColorPoint extends Point {
  constructor(x, color) {
    this.color = color; // ReferenceError
    super(x);
    this.color = color; // 正确
  }
}
复制代码
```

Object.getPrototypeOf()

- `Object.getPrototypeOf`方法可以用来从子类上获取父类。可以使用这个方法判断，一个类是否继承了另一个类。

```
Object.getPrototypeOf(ColorPoint) === Point // true
复制代码
```

super 关键字

- `super`这个关键字，既可以当作函数使用，也可以当作对象使用。在这两种情况下，它的用法完全不同。

1. `super`

   作为函数调用时，代表父类的构造函数。

   ```
   ES6
   ```

   要求，子类的构造函数必须执行一次

   ```
   super
   ```

   函数。否则

   ```
   JavaScript
   ```

   引擎会报错。

```
class A {
  constructor() {
    console.log(new.target.name);
  }
}
class B extends A {
  constructor() {
    super();
  }
}
new A() // A
new B() // B
复制代码
```

- `super` 虽然代表了父类 `A` 的构造函数，但是返回的是子类 `B` 的实例，`super()` 相当于 `A.prototype.constructor.call(this)`，`super()` 内部的 `this` 指向的是 `B`。
- **作为函数时，`super()` 只能用在子类的构造函数之中，用在其他地方就会报错。**

```
class A {}
class B extends A {
  m() {
    super(); // 报错
  }
}
复制代码
```

2. `super`

作为对象时

- **普通方法中，指向父类的原型对象；静态方法（带 `static` 前缀的方法）中，指向父类。**

```
class A {
  p() {
    return 2;
  }
}
class B extends A {
  constructor() {
    super();
    console.log(super.p()); // 2
  }
}

let b = new B();
复制代码
```

- `super` 指向父类的原型对象，所以定义在父类实例上的方法或属性，是无法通过 `super` 调用的。如果属性定义在父类的原型对象上，`super` 就可以取到。

```
class A {
  constructor() {
    this.p = 2;
  }
```

```
}
class B extends A {
  get m() {
    return super.p;
  }
}
let b = new B();
b.m // undefined

// 定义到原型上
class A {}
A.prototype.x = 2;

class B extends A {
  constructor() {
    super();
    console.log(super.x) // 2
  }
}
let b = new B();
```
复制代码

- **在子类的静态方法中通过`super`调用父类的方法时，方法内部的`this`指向当前的子类，而不是子类的实例。**

```
class A {
  constructor() {
    this.x = 1;
  }
  static print() {
    console.log(this.x);
  }
}

class B extends A {
  constructor() {
    super();
    this.x = 2;
  }
  static m() {
    super.print();
  }
}

B.x = 3;
B.m() // 3
```
复制代码

- **使用`super`的时候，必须显式指定是作为函数、还是作为对象使用，否则会报错。**

```
class A {}

class B extends A {
  constructor() {
    super();
    console.log(super); // 报错
  }
}
复制代码
```

- **由于对象总是继承其他对象的，所以可以在任意一个对象中，使用`super`关键字。**

```
var obj = {
  toString() {
    return "MyObject: " + super.toString();
  }
};

obj.toString(); // MyObject: [object Object]
复制代码
```

类的`prototype`属性和`__proto__`属性

- 大多数浏览器的

  ```
  ES5
  ```

  实现之中，每一个对象都有

  ```
  __proto__
  ```

  属性，指向对应的构造函数的

  ```
  prototype
  ```

  属性。存在两条继承链。

  1. 子类的`__proto__`属性，表示构造函数的继承，总是指向父类。
  2. 子类`prototype`属性的`__proto__`属性，表示方法的继承，总是指向父类的`prototype`属性。

  ```
  class A {}
  class B extends A {}

  B.__proto__ === A // true
  B.prototype.__proto__ === A.prototype // true
  复制代码
  ```

- 类的继承模式实现

```
class A {}
class B {}

// B 的实例继承 A 的实例
Object.setPrototypeOf(B.prototype, A.prototype);
// B 继承 A 的静态属性
Object.setPrototypeOf(B, A);

const b = new B();
```
复制代码

- `Object.setPrototypeOf`方法的实现。

```
Object.setPrototypeOf = function (obj, proto) {
  obj.__proto__ = proto;
  return obj;
}
```
复制代码

- 类实例的`__proto__`属性的`__proto__`属性，指向父类实例的`__proto__`属性。子类的原型的原型，是父类的原型。

```
var p1 = new Point(2, 3);
var p2 = new ColorPoint(2, 3, 'red');

p2.__proto__ === p1.__proto__ // false
p2.__proto__.__proto__ === p1.__proto__ // true
```
复制代码

原生构造函数的继承

- 原生构造函数是指语言内置的构造函数，通常用来生成数据结构。`ECMAScript`的原生构造函数大致有：`Boolean()`、`Number()`、`String()`、`Array()`、`Date()`、`Function()`、`RegExp()`、`Error()`、`Object()`...
- `ES6`可以自定义原生数据结构（比如`Array`、`String`等）的子类，这是`ES5`无法做到的。例如实现一个自己的带其他功能的数组类。

```
class VersionedArray extends Array {
  constructor() {
    super();
    this.history = [[]];
  }
  commit() {
    this.history.push(this.slice());
  }
  revert() {
    this.splice(0, this.length, ...this.history[this.history.length - 1]);
  }
}

var x = new VersionedArray();

x.push(1);
x.push(2);
```

```
x // [1, 2]
x.history // [[]]

x.commit();
x.history // [[], [1, 2]]

x.push(3);
x // [1, 2, 3]
x.history // [[], [1, 2]]

x.revert();
x // [1, 2]
```
复制代码

- 继承`Object`的子类，有一个行为差异。

```
class NewObj extends Object{
  constructor(){
    super(...arguments);
  }
}
var o = new NewObj({attr: true});
o.attr === true  // false
```
复制代码

- 上面代码中，`NewObj`继承了`Object`，但是无法通过`super`方法向父类`Object`传参。这是因为`ES6`改变了`Object`构造函数的行为，一旦发现`Object`方法**不是通过`new Object()`这种形式调用，`ES6`规定`Object`构造函数会忽略参数。**

`Mixin`模式的实现

- `Mixin`指的是多个对象合成一个新的对象，新对象具有各个组成成员的接口。使用的时候，只要继承这个类即可。

```
function mix(...mixins) {
  class Mix {
    constructor() {
      for (let mixin of mixins) {
        copyProperties(this, new mixin()); // 拷贝实例属性
      }
    }
  }
  for (let mixin of mixins) {
    copyProperties(Mix, mixin); // 拷贝静态属性
    copyProperties(Mix.prototype, mixin.prototype); // 拷贝原型属性
  }
  return Mix;
}

function copyProperties(target, source) {
  for (let key of Reflect.ownKeys(source)) {
    if ( key !== 'constructor'
      && key !== 'prototype'
      && key !== 'name'
    ) {
      let desc = Object.getOwnPropertyDescriptor(source, key);
```

```
      Object.defineProperty(target, key, desc);
    }
  }
}

// 使用
class DistributedEdit extends mix(Loggable, Serializable) {
  // ...
}
```

Module

- ES6-Module
- `CommonJS` 和 `AMD` 模块，都只能在运行时确定这些东西。 `ES6` 可以在编译时就完成模块加载，效率要比 `CommonJS` 模块的加载方式高，这种加载称为“编译时加载”或者静态加载。
- 优势
    1. 能进一步拓宽 `JavaScript` 的语法，比如引入宏（`macro`）和类型检验（`type system`）这些只能靠静态分析实现的功能。
    2. 不再需要 `UMD` 模块格式。
    3. 将来浏览器的新 `API` 就能用模块格式提供，不再必须做成全局变量或者 `navigator` 对象的属性。
    4. 不再需要对象作为命名空间（比如 `Math` 对象），未来这些功能可以通过模块提供。

严格模式

- `ES6` 的模块自动采用严格模式，不管你有没有在模块头部加上 `"use strict"`。
- 限制：
    1. 变量必须声明后再使用
    2. 函数的参数不能有同名属性，否则报错
    3. 不能使用 `with` 语句
    4. 不能对只读属性赋值，否则报错
    5. 不能使用前缀 `0` 表示八进制数，否则报错
    6. 不能删除不可删除的属性，否则报错
    7. 不能删除变量 `delete prop`，会报错，只能删除属性 `delete global[prop]`
    8. `eval` 不会在它的外层作用域引入变量
    9. `eval` 和 `arguments` 不能被重新赋值
    10. `arguments` 不会自动反映函数参数的变化
    11. 不能使用 `arguments.callee` 和 `arguments.caller`
    12. 禁止 `this` 指向全局对象
    13. 不能使用 `fn.caller` 和 `fn.arguments` 获取函数调用的堆栈
    14. 增加了保留字（比如 `protected`、`static` 和 `interface`）
- **尤其需要注意this的限制。 `ES6` 模块之中，顶层的 `this` 指向 `undefined`，即不应该在顶层代码使用 `this`。**

export 命令

- `export` 命令用于规定模块的对外接口。
- 一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用 `export` 关键字输出该变量。除了输出变量，还可以输出函数或类（`class`）。

```
```

```
// index.js
export const name = 'detanx';
export const year = 1995;
export function multiply(x, y) {
  return x * y;
};

// 写法二
const name = 'detanx';
const year = 1995;
function multiply(x, y) {
  return x * y;
};
export { name, year, multiply }
复制代码
```

- `export`输出的变量就是本来的名字，但是可以使用`as`关键字重命名。重命名后，可以用不同的名字输出多次。

```
function v1() { ... }
function v2() { ... }

export {
  v1 as streamV1,
  v2 as streamV2,
  v2 as streamLatestVersion
};
复制代码
```

- `export`命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。

```
// 报错
export 1;

var m = 1;
export m;

// 正确
export var m = 1;

var m = 1;
export {m};

var n = 1;
export {n as m};
复制代码
```

- **`export`命令可以出现在模块的任何位置，只要处于模块顶层就可以。**
- **`export \*`命令会忽略模块的`default`方法。**

```
// 整体输出
export * from 'my_module';
复制代码
```

import 命令

- 使用`export`命令定义了模块的对外接口以后，其他`JS`文件就可以通过`import`命令加载这个模块。想为输入的变量重新取一个名字，`import`命令要使用`as`关键字，将输入的变量重命名。

```
import { name, year } from './index.js';
import { name as username } from './profile.js';
复制代码
```

- **`import`命令输入的变量都是只读的，因为它的本质是输入接口。** 也就是说，不允许在加载模块的脚本里面，改写接口。**如果`a`是一个对象，改写`a`的属性是允许的。**

```
import {a} from './xxx.js'
a = {}; // Syntax Error : 'a' is read-only;
a.foo = 'hello'; // 合法操作
复制代码
```

- `import`后面的`from`指定模块文件的位置，可以是相对路径，也可以是绝对路径，`.js`后缀可以省略。如果只是模块名，不带有路径，那么必须有配置文件（例如使用`webpack`配置路径），告诉`JavaScript`引擎该模块的位置。

```
import {myMethod} from 'util';
复制代码
```

- **`import`命令具有提升效果，会提升到整个模块的头部，首先执行。**

```
foo(); // 不会报错
import { foo } from 'my_module';
复制代码
```

- `import`是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。

```
// 报错
import { 'f' + 'oo' } from 'my_module';

// 报错
let module = 'my_module';
import { foo } from module;

// 报错
if (x === 1) {
  import { foo } from 'module1';
} else {
  import { foo } from 'module2';
}
复制代码
```

- **多次重复执行同一句`import`语句，那么只会执行一次，而不会执行多次。**

```
import 'lodash';
import 'lodash'; // 只会执行一次

import { foo } from 'my_module';
import { bar } from 'my_module';

// 等同于
import { foo, bar } from 'my_module';
复制代码
```

模块的整体加载

- 除了指定加载某个输出值，还可以使用整体加载，即用星号（`*`）指定一个对象，所有输出值都加载在这个对象上面。

```
import * as user from './index.js';
user.name; // 'detanx'
user.year; // 1995
复制代码
```

export default 命令

- `export default`命令，为模块指定默认输出。其他模块加载该模块时，`import`命令（`import`命令后面，不使用大括号）可以为该匿名函数指定任意名字。

```
// export-default.js
export default function () {
  console.log('detanx');
}

// import-default.js
import customName from './export-default';
customName(); // 'detanx'
复制代码
```

- **使用`export default`时，对应的`import`语句不需要使用大括号；使用`export`，对应的`import`语句需要使用大括号。** 一个模块只能有一个默认输出，因此`export default`命令只能使用一次。

```
export default function crc32() {  ...}
import crc32 from 'crc32';

export function crc32() { ... };
import { crc32 } from 'crc32';
复制代码
```

export 与 import 的复合写法

- **如果在一个模块之中，先输入后输出同一个模块，**`import`语句可以与`export`语句写在一起。**写成一行以后，**`foo`**和**`bar`**实际上并没有被导入当前模块，只是相当于对外转发了这两个接口，导致当前模块不能直接使用**`foo`**和**`bar`**。**

```
export { foo, bar } from 'my_module';

// 可以简单理解为
import { foo, bar } from 'my_module';
export { foo, bar };
复制代码
```

- 模块的接口改名和整体输出，也可以采用这种写法。

```
// 接口改名
export { foo as myFoo } from 'my_module';

// 整体输出
export * from 'my_module';
复制代码
```

- 默认接口的写法如下。

```
export { default } from 'foo';
复制代码
```

- 具名接口改为默认接口的写法如下。

```
export { es6 as default } from './someModule';

// 等同于
import { es6 } from './someModule';
export default es6;
复制代码
```

- 同样地，默认接口也可以改名为具名接口。

```
export { default as es6 } from './someModule';
ES2020 之前，有一种import语句，没有对应的复合写法。

import * as someIdentifier from "someModule";
复制代码
```

- `ES2020`补上了这个写法。

```
export * as ns from "mod";

// 等同于
import * as ns from "mod";
export {ns};
复制代码
```

应用

1. 公共模块
    - 例如项目有很多的公共方法放到一个`constant`的文件，我们需要什么就加载什么。

```
// constants.js 模块
export const A = 1;
export const B = 3;
export const C = 4;

// use.js
import {A, B} from './constants';
```
复制代码

2. import()

- `import`命令会被`JavaScript`引擎静态分析，先于模块内的其他语句执行（`import`命令叫做“连接”`binding`其实更合适）。所以我们只能在最顶层去使用。`ES2020`引入`import()`函数，支持动态加载模块。
- `import()`返回一个`Promise`对象。

```
const main = document.querySelector('main');

import(`./section-modules/${someVariable}.js`)
  .then(module => {
    module.loadPageInto(main);
  })
  .catch(err => {
    main.textContent = err.message;
  });
```
复制代码

- `import()`函数可以用在任何地方，不仅仅是模块，非模块的脚本也可以使用。它是运行时执行，也就是说，什么时候运行到这一句，就会加载指定的模块。另外，`import()`函数与所加载的模块没有静态连接关系，这点也是与import语句不相同。`import()`类似于`Node`的`require`方法，区别主要是前者是异步加载，后者是同步加载。
- **适用场景按需加载、条件加载、动态的模块路径。**

3. 注意点

- `import()`加载模块成功以后，这个模块会作为一个对象，当作`then`方法的参数。因此，可以使用对象解构赋值的语法，获取输出接口。

```
import('./myModule.js')
.then(({export1, export2}) => {
  // ...·
});
```
复制代码

- 上面代码中，`export1`和`export2`都是`myModule.js`的输出接口，可以解构获得。
- 如果模块有`default`输出接口，可以用参数直接获得。

```
import('./myModule.js')
.then(myModule => {
  console.log(myModule.default);
});
```
复制代码

- 上面的代码也可以使用具名输入的形式。

```
import('./myModule.js')
.then(({default: theDefault}) => {
  console.log(theDefault);
});
复制代码
```

- 如果想同时加载多个模块，可以采用下面的写法。

```
Promise.all([
  import('./module1.js'),
  import('./module2.js'),
  import('./module3.js'),
])
.then(([module1, module2, module3]) => {
   ...
});
复制代码
```

- `import()` 也可以用在 `async` 函数之中。

```
async function main() {
  const myModule = await import('./myModule.js');
  const {export1, export2} = await import('./myModule.js');
  const [module1, module2, module3] =
    await Promise.all([
      import('./module1.js'),
      import('./module2.js'),
      import('./module3.js'),
    ]);
}
main();
复制代码
```

Module 加载实现

- ES6-Module 加载实现

简介

1. 传统加载
   - 默认情况下，浏览器是同步加载 `JavaScript` 脚本，即渲染引擎遇到 `<script>` 标签就会停下来，等到执行完脚本，再继续向下渲染。为了解决 `<script>` 标签打开 `defer` 或 `async` 属性，脚本就会异步加载。
   - `defer` 与 `async` 的区别是：`defer` 要等到整个页面在内存中正常渲染结束（`DOM` 结构完全生成，以及其他脚本执行完成），才会执行；`async` 一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话，`defer` 是“渲染完再执行”，`async` 是“下载完就执行”。另外，如果有多个 `defer` 脚本，会按照它们在页面出现的顺序加载，而多个 `async` 脚本是不能保证加载顺序的。
2. 加载规则
   - 浏览器加载 `ES6` 模块，也使用 `<script>` 标签，但是要加入 `type="module"` 属性。等同于打开了 `<script>` 标签的 `defer` 属性。

```
<script type="module" src="./foo.js"></script>

<!-- 等同于 -->
<script type="module" src="./foo.js" defer></script>
复制代码
```

- 对于外部的模块脚本，有几点需要注意。
  1. 代码是在模块作用域之中运行，而不是在全局作用域运行。模块内部的顶层变量，外部不可见。
  2. 模块脚本自动采用严格模式，不管有没有声明`"use strict"`。
  3. 模块之中，可以使用`import`命令加载其他模块（`.js`后缀不可省略，需要提供绝对`URL`或相对`URL`），也可以使用`export`命令输出对外接口。
  4. 模块之中，顶层的`this`关键字返回`undefined`，而不是指向`window`。也就是说，在模块顶层使用`this`关键字，是无意义的。
  5. 同一个模块如果加载多次，将只执行一次。

```
import utils from 'https://example.com/js/utils.js';
const x = 1;

console.log(x === window.x); //false
console.log(this === undefined); // true
复制代码
```

- 利用顶层的`this`等于`undefined`这个语法点，可以侦测当前代码是否在`ES6`模块之中。

```
const isNotModuleScript = this !== undefined;
复制代码
```

ES6 模块与 CommonJS 模块的差异

- 讨论`Node.js`加载`ES6`模块之前，必须了解`ES6`模块与`CommonJS`模块完全不同。
- 它们有两个重大差异。
  1. `CommonJS`模块输出的是一个值的拷贝，`ES6`模块输出的是值的引用。
  2. `CommonJS`模块是运行时加载，`ES6`模块是编译时输出接口。（因为`CommonJS`加载的是一个对象（即`module.exports`属性），该对象只有在脚本运行完才会生成。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。）
- `CommonJS`模块输出的是值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。除非写成一个函数，才能得到内部变动后的值。

```
// lib.js
var counter = 3;
function incCounter() {
  counter++;
}
module.exports = {
  counter: counter,
  incCounter: incCounter,
};

// main.js
var mod = require('./lib');
```

```
console.log(mod.counter);  // 3
mod.incCounter();
console.log(mod.counter); // 3

// 写成函数
// lib.js
var counter = 3;
function incCounter() {
  counter++;
}
module.exports = {
  get counter() {
    return counter
  },
  incCounter: incCounter,
};

$ node main.js
3
4
```
复制代码

- `ES6` **模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。**

```
// lib.js
export let counter = 3;
export function incCounter() {
  counter++;
}

// main.js
import { counter, incCounter } from './lib';
console.log(counter); // 3
incCounter();
console.log(counter); // 4
```
复制代码

- `ES6` 输入的模块变量，只是一个“符号连接”，所以这个变量是只读的，对它进行重新赋值会报错。

```
// lib.js
export let obj = {};

// main.js
import { obj } from './lib';

obj.prop = 123; // OK
obj = {}; // TypeError
```
复制代码

- `export`通过接口，输出的是同一个值。不同的脚本加载这个接口，得到的都是同样的实例。

```
// mod.js
function C() {
  this.sum = 0;
  this.add = function () {
    this.sum += 1;
  };
  this.show = function () {
    console.log(this.sum);
  };
}

export let c = new C();
复制代码
```

- 上面的脚本`mod.js`，输出的是一个`C`的实例。不同的脚本加载这个模块，得到的都是同一个实例。

```
// x.js
import {c} from './mod';
c.add();

// y.js
import {c} from './mod';
c.show();

// main.js
import './x';
import './y';
复制代码
```

- 现在执行`main.js`，输出的是`1`。

```
$ babel-node main.js
1
复制代码
```

- 证明了`x.js`和`y.js`加载的都是`C`的同一个实例。

Node.js 加载

- `Node.js`要求`ES6`模块采用`.mjs`后缀文件名。`Node.js`遇到`.mjs`文件，就认为它是`ES6`模块，默认启用严格模式，不必在每个模块文件顶部指定`"use strict"`。 如果不希望将后缀名改成`.mjs`，可以在项目的`package.json`文件中，指定`type`字段为`module`。

```
{
   "type": "module"
}
复制代码
```

- 这时还要使用`CommonJS`模块，那么需要将`CommonJS`脚本的后缀名都改成`.cjs`。如果没有`type`字段，或者`type`字段为`commonjs`，则`.js`脚本会被解释成`CommonJS`模块。
- 总结：**`.mjs`文件总是以`ES6`模块加载，`.cjs`文件总是以`CommonJS`模块加载，`.js`文件的加载取决于`package.json`里面`type`字段的设置。**

- **注意，`ES6`模块与`CommonJS`模块尽量不要混用。`require`命令不能加载`.mjs`文件，会报错，只有`import`命令才可以加载`.mjs`文件。反过来，`.mjs`文件里面也不能使用`require`命令，必须使用`import`。**
- Node.js 加载 主要是介绍`ES6`模块和`CommonJS`相互之间的支持，有兴趣的可以自己去看看。

循环加载

- “循环加载”（`circular dependency`）指的是，`a`脚本的执行依赖`b`脚本，而`b`脚本的执行又依赖`a`脚本。“循环加载”表示存在强耦合，如果处理不好，还可能导致递归加载，使得程序无法执行，因此应该避免出现，但很难避免尤其是特别复杂的项目。

文件结构

- 在日常项目开发中，大部分的开发人员是不需要去关注整个项目的结构目录的，因为项目的负责人，或者其他的架构人员已经把整个项目文件结构弄好了。开发人员只需要在自己负责的模块目录去编写对应的模块即可。那如果需要你去搭建一个项目，你应该怎么去设计文件结构呢?

`react`项目为例

- 项目名称为`detanx-react`。

入口

- `detanx-react/index.js`文件，项目的入口文件。

配置目录

- `detanx-react/config`文件夹，用来存放项目的一些配置文件，例如我们使用`webpack`的打包配置以及其他比如将`react-router`、`react-dom`等通过gulp打包单独的包等配置。

打包分析目录

- `detanx-react/analyz`文件夹，用来存放通过`webpack`打包后，分析打包过程的文件。

生产目录

- `detanx-react/build`或者`detanx-react/dist`文件夹，用来存放项目开发完成后，需要部署上线的所有生产文件。

数据目录

- `detanx-react/mock`文件夹，在项目开发时，后端开发的进度可能没有前端快，导致接口无法调试，所以我们可以自己配置`mock`数据进行开发和调试。这个文件夹就用来存放所有的`mock`数据。

公共目录

- `detanx-react/public`文件夹，打包会需要基本的`html`模版，内嵌的一些比如`css`、`favicon`等我们就可以放到这个文件夹下面。

引用目录

- ```
  detanx-react/src
  ```

  文件夹，这个文件夹是我们项目开发变化最多的，我们写的大多数代码都在这个文件夹下面，针对不同项目，它下面的结构也不一样。

  1. `detanx-react/src/components`文件夹，存放项目开发的公共组件。
  ```

2. `detanx-react/src/hooks`文件夹，存放开发过程中封装的一些新的hook。
3. `detanx-react/src/request`文件夹，存放项目请求的封装。
4. `detanx-react/src/router`文件夹，存放项目的路由相关设置，配置也可以写到`detanx-react/config`文件夹下。
5. ```
   detanx-react/src/view
   ```

   文件夹，存放项目开发的所有页面内容，根据每个模块划分。
   - 每个模块中又可以分为`index`文件（模块入口）、`components`文件夹（存放模块的抽离组件）等。
6. `detanx-react/src/common`文件夹，存放项目的公共内容，例如：公共的方法（判断空、数据类型等）、常量（会多个（`2`个及以上）不同模块用到的常量）文件等。

静态目录

- `detanx-react/static`文件夹，这个文件夹也可以放在`detanx-react/src`下面，用于存放一些图片、字体、`CSS`（`LESS`、`SASS`）等文件。

其他

- 根据自己的项目需求在任意目录下可以适当添加其他文件夹。

  

命名

- 命名部分主要包括了函数命名、变量命名、模块命名、`class`命名（`CSS`命名）。开发项目时，尤其在多人合作的时候，如果没有一个好的命名规范并且也不喜欢写代码注释。当你离职或者其他原因需要项目交接时，别人可能就是在遍看代码边问候你的家人了，可能还会打电话问你，浪费彼此很多时间。

函数命名

- 命名方式可以用大驼峰、小驼峰。一般建议使用小驼峰。
- 我们在写一个函数的时候，我们首先应该确定该函数的一个功能，比如判断一个值的类型、通过请求获取一个某个表格的数据、显示或隐藏某个页面等。所以根据不同的功能我们可以用不同的单词开始，例如：
  1. `can` 判断是否可执行某个动作（`canGetUserName`）
  2. `has` 判断是否含有某个值（`hasUserName`）
  3. `is` 判断是否为某个值（`isEmpty`）
  4. `get` 获取某个值（`getUserName`）
  5. `set` 设置某个值（`setUserName`）
  6. `load` 加载某些数据（`loadUserInfo`）
- 这几个肯定不可能完整的覆盖项目中所有的场景，当我们函数命名不能直接看出在做什么的时候，我们可以通过给函数添加注释来说明函数的作用。
- **每个函数应该都是一个独立的功能，降低函数之间的耦合。**
- **每个函数的大小不宜超过`200`行，超过应该尝试抽离部分逻辑为一个新的函数。实在无法抽离应该在复杂逻辑部分添加适当的注释。**
- **私有方法可以在项目开发的时候，开发人员之间做一个规范，例如使用`_`或`$`开头的函数为私有。**

变量命名
   ```

- 在项目开发的时候，很多开发人员为了省事，就对一个变量随便取名。例如：可能只是用来接收一个函数的返回值，就取个`a`，反正下面只会用到一次；再者一个是某个计数变量就直接一个`i`或者其他字母。当你这个模块全身这种变量时，别人看你的代码就是一个灾难。
- 普通变量我们可以使用var或者let去声明，在命名一个变量时，我们应该通过变量的类型或者它的使用场景去命名，尽量做到明确语义。例如：
    1. 布尔类型：`isShowModal`（是否显示弹窗）
    2. 数组类型：`userLists`（用户列表）
    3. 对象类型：`userInfoObj`（用户信息对象，虽然我们一般知道`userInfo`就是一个对象，我这里只是表示一个如果看不出来的命名，我们可以在后面加一个标识）
    4. `Symbol`类型：`userNameSym`（`Symbol`类型的用户名）
    5. 等等
- **变量命名必须以字母、下划线`_`或者`$`为开头，通常开发时约定`_`或者`$`开头的为私有变量。**
- 变量命名时应该尽量保证命名中不出现数字，例如：`list1`、`list2`等等这种变量名称。
- **常量应该全部大些并且每个单词以下划线连接（`WARNING_DUATION_TIME`：警告显示时间）。**

模块命名

- 以`react`模块为例，我们每个模块的命名应该使用大驼峰（`UserConfig`），并且模块名称应该语义化，表示模块的内容，例如（`UserConfig`）表示用户的配置模块。

`class`命名（`CSS`命名）

- ```
  CSS
  ```

  命名我们一般是使用

  ```
  BEM
  ```

  （块（

  ```
  block
  ```

  ）、元素（

  ```
  element
  ```

  ）、修饰符（

  ```
  modifier
  ```

  ））命名规范。

  ```
  .block {}
  .block__element {}
  .block--modifier {}
  复制代码
  ```

    - `block`代表了更高级别的抽象或组件。
    - `block__element`代表`.block`的后代，用于形成一个完整的`.block`的整体。
    - `block--modifier`代表`.block`的不同状态或不同版本。
  ```

- 使用两个连字符和下划线而不是一个，是为了让你自己的块可以用单个连字符来界定。

```
.sub-block__element {}
.sub-block--modifier {}
复制代码
```

- 优缺点：
  1. 优点：可以获得更多的描述和更加清晰的结构，从其名字可以知道某个标记的含义。
  2. 缺点：当嵌套太深，`class`名称会特别长，我们可以通过其他约定解决，例如：缩减层级等。
- 使用
  1. 时机：需要明确关联性的模块关系时。
  2. 处理：通过 `LESS/SASS` 等预处理器语言来编写 `CSS`。

代码规范

- ES6-编程风格
- 在开发中，如果没有使用一些代码规范的插件，多个开发人员写得代码就几个风格，所以协同开发时，我们需要对代码的规范进行约定，例如`js`通过`eslint`去约束，`ts`可以通过`tslint`约束，我们只需要一个人去配置相应的约束的配置文件。
- 但有些代码的编写这些约束也做不到，只是实现的好坏而已。下面举一些代码上的规范（不一定是约束无法做到的）。

ESLint 的使用示例

- `ESLint` 是一个语法规则和代码风格的检查工具，可以用来保证写出语法正确、风格统一的代码。
- 安装 `ESLint`。

```
$ npm i -g eslint
复制代码
```

- 安装 `Airbnb` 语法规则，以及 `import`、`a11y`、`react` 插件。

```
$ npm i -g eslint-config-airbnb
$ npm i -g eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-react
复制代码
```

- 在项目的根目录下新建一个`.eslintrc`文件，配置 `ESLint`。

```
{
  "extends": "eslint-config-airbnb"
}
复制代码
```

- `vscode`中安装`eslint`插件即可边写边检查。

引号

- 引号分为单引号（`'`）和双引号（`"`）以及ES6新增的模版字符串使用`表示，可能在开发时大部分都是随便去使用这个引号，怎么输入着方便怎么来。一般情况下，`.js`、`.ts`、`.jsx`、`.tsx`、`.vue`等可以写`js`的文件中，一般是使用单引号（`''`），在`html`中（标签的属性等）使用双引号（`""`）。`可以代替以前的连续字符串连接操作。

```
export default class extends PureComponent {
    constructor() {
        this.state = {
          value: 'detanx',
          list: null
        }
      }
    render() {
        const { value, list } = this.state
        return (<>
            <div className="detanx">{value}</div>
        </>)
    }
}

// ``使用
let user = 'my name is'
const NAME = 'detanx'; // 简单示例一下

// es5
user = user + ' ' + NAME;

// es6
let user = `${user} ${NAME}`
复制代码
```

变量

- 变量命名规范在上面说过了，这里说一下变量的使用。我们现在编写代码应该尽量的使用`let`和`const`去声明变量，而不是用`var`。
- **所有变量都应该先声明再使用。**
- **超过`2`处使用的某个值，我们应该使用`const`在当前使用模块的最顶层去声明，多个模块使用的应该在文件的公共模块的常量文件中声明。**
- **声明一个变量应该给它赋一个初值。**

```
// bad
let i
for(i = 0; i < 10; i ++) { ... }

// good
let i = 0;
for(; i < 10; i ++) { ... }

for (let i = 0; i < 10; i ++) { ... }
复制代码
```

- 使用`const`连续声明多个值时，可以使用数组解构赋值的方式。

```
// bad
var a = 1, b = 2, c = 3;

// good
const a = 1;
const b = 2;
const c = 3;

// best
const [a, b, c] = [1, 2, 3];
```
复制代码

解构赋值

- 可以使用解构赋值的地方优先使用解构赋值。例如上面的多个

  ```
  const
  ```

  声明。

  1. 数组成员对变量赋值时，优先使用解构赋值。

  ```
  const arr = [1, 2, 3, 4];

  // bad
  const first = arr[0];
  const second = arr[1];

  // good
  const [first, second] = arr;
  ```
  复制代码

  1. 函数的参数如果是对象的成员，优先使用解构赋值。

  ```
  // bad
  function getFullName(user) {
    const firstName = user.firstName;
    const lastName = user.lastName;
  }

  // good
  function getFullName(obj) {
    const { firstName, lastName } = obj;
  }

  // best
  function getFullName({ firstName, lastName }) {
  }
  ```
  复制代码

  1. 如果函数返回多个值，优先使用对象的解构赋值，而不是数组的解构赋值。这样便于以后添加返回值，以及更改返回值的顺序。

```
// bad
function processInput(input) {
  return [left, right, top, bottom];
}

// good
function processInput(input) {
  return { left, right, top, bottom };
}

const { left, right } = processInput(input);
```
复制代码

对象

- 单行定义的对象，最后一个成员不以逗号结尾。多行定义的对象，最后一个成员以逗号结尾。

```
// bad
const a = { k1: v1, k2: v2, };
const b = {
  k1: v1,
  k2: v2
};

// good
const a = { k1: v1, k2: v2 };
const b = {
  k1: v1,
  k2: v2,
};
```
复制代码

- 对象尽量静态化，一旦定义，就不得随意添加新的属性。如果添加属性不可避免，要使用`Object.assign`方法。

```
// bad
const a = {};
a.x = 3;

// if reshape unavoidable
const a = {};
Object.assign(a, { x: 3 });

// good
const a = { x: null };
a.x = 3;
```
复制代码

- 如果对象的属性名是动态的，可以在创造对象的时候，使用属性表达式定义。

```
// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
```

```
obj[getKey('enabled')] = true;

// good
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```
复制代码

- 上面代码中，对象`obj`的最后一个属性名，需要计算得到。这时最好采用属性表达式，在新建`obj`的时候，将该属性与其他属性定义在一起。这样一来，所有属性就在一个地方定义了。
- 对象的属性和方法，尽量采用简洁表达法，这样易于描述和书写。

```
var ref = 'some value';
// bad
const atom = {
  ref: ref,
  value: 1,
  addValue: function (value) {
    return atom.value + value;
  },
};
// good
const atom = {
  ref,
  value: 1,
  addValue(value) {
    return atom.value + value;
  },
};
```
复制代码

数组

使用扩展运算符（`...`）拷贝数组。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;
for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}
// good
const itemsCopy = [...items];
```
复制代码

- 使用`Array.from`方法，将类似数组的对象转为数组。

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```
复制代码

函数

- 匿名函数当作参数的场合，尽量用箭头函数代替。

```
// 立即执行函数
(() => {
  console.log('Welcome to the Internet.');
})();
复制代码
```

- 所有配置项都应该集中在一个对象，放在最后一个参数，**布尔值不可以直接作为参数。**

```
// bad
function divide(a, b, option = false ) { ... }
// good
function divide(a, b, { option = false } = {}) { ... }
复制代码
```

- 不要在函数体内使用 `arguments` 变量，使用 `rest` 运算符（`...`）代替。因为 `rest` 运算符显式表明你想要获取参数，而且 `arguments` 是一个类似数组的对象，而 `rest` 运算符可以提供一个真正的数组。

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}
// good
function concatenateAll(...args) {
  return args.join('');
}
复制代码
```

- 使用默认值语法设置函数参数的默认值。

```
// bad
function handleThings(opts) {
  opts = opts || {};
}
// good
function handleThings(opts = {}) { ... }
复制代码
```

Map 结构

- **注意区分 `Object` 和 `Map`，只有模拟现实世界的实体对象时，才使用 `Object`。如果只是需要 `key: value` 的数据结构，使用 `Map` 结构。因为 `Map` 有内建的遍历机制。**

```
let map = new Map(arr);
for (let key of map.keys()) { console.log(key); }

for (let value of map.values()) { console.log(value); }

for (let item of map.entries()) { console.log(item[0], item[1]); }
复制代码
```

模块

- 使用`import`取代`require`。

```
// bad
const moduleA = require('moduleA');
const func1 = moduleA.func1;
const func2 = moduleA.func2;

// good
import { func1, func2 } from 'moduleA';
复制代码
```

- 使用`export`取代`module.exports`。

```
// commonJS的写法
const React = require('react');
const Breadcrumbs = React.createClass({
  render() {
    return <nav />;
  }
});
module.exports = Breadcrumbs;

// ES6的写法
import React from 'react';
class Breadcrumbs extends React.Component {
  render() {
    return <nav />;
  }
};
export default Breadcrumbs;
复制代码
```

- 如果模块只有一个输出值，就使用`export default`，如果模块有多个输出值，就使用`export`，`export default`与普通的`export`不要同时使用。
- 不要在模块输入中使用通配符。因为这样可以确保你的模块之中，有一个默认输出（`export default`）。

```
// bad
import * as myObject from './importModule';

// good
import myObject from './importModule';
复制代码
```

- **模块默认输出一个函数，函数名的首字母应该小写。默认输出一个对象，对象名的首字母应该大写。**

```
function makeStyleGuide() { ... }
export default makeStyleGuide;

const StyleGuide = {
  es6: {
  }
};
export default StyleGuide;
```
复制代码

关注下面的标签，发现更多相似文章

Decorator

- ES6-Decorator

环境配置

- 由于装饰器目前处于`stage 2`，所以无法在浏览器中直接使用，我们需要通过`babel`编译为`es5`去使用。
- 我们如果只想简单去使用这样一个语法去做一个了解，我们可以简单配置一下，过程如下：
    1. 全局安装`@babel/cli`和`@babel/core`。

```
npm install -g @babel/core @babel/cli
```
复制代码

    1. 项目目录下初始化一个`package.json`，并安装`decorator`的相关`babel`插件`@babel/plugin-proposal-class-properties`和`@babel/plugin-proposal-decorators`。

```
npm init -y // 初始化package.json
npm install --save-dev @babel/plugin-proposal-class-properties
npm install --save-dev @babel/plugin-proposal-decorators
```
复制代码

    1. 在项目根目录下创建`.babelrc`文件，并添加一下代码。

```
{
  "presets": [],
  "plugins": [
    [
      "@babel/plugin-proposal-decorators",
      {
        "legacy": true
      }
    ],
    [
      "@babel/plugin-proposal-class-properties",
      {
        "loose": true
      }
    ]
  ]
}
```

复制代码

1. 在`package.json`的`scripts`中添加编译命令。

```
// package.json
// babel inputfile -w（实时编译）-o（输出）outputfile
scripts: {
    build: 'babel es6.js -w -o es5.js'
}
复制代码
```

1. 在`html`文件中引入编译后的文件。

```
// index.html
<DOCTYPE html>
<html>
...
<body>
...
<script src='./es5.js'></script>
</body>
</html>
复制代码
```

介绍

- `Decorator`**提案经过了大幅修改，目前还没有定案，不知道语法会不会再变。** 装饰器（`Decorator`）是一种与类（`class`）相关的语法，用来注释或修改类和类方法，写成`@ +`函数名。它可以放在类和类方法的定义前面。

```
@frozen class Foo {
  @configurable(false)
  @enumerable(true)
  method() {}

  @throttle(500)
  expensiveMethod() {}
}
复制代码
```

使用

类的装饰

- 装饰器可以装饰整个类。装饰器是一个对类进行处理的函数。装饰器函数的第一个参数，就是所要装饰的目标类。

```
@decorator
class A {}

// 等同于
class A {}
A = decorator(A) || A;
复制代码
```

- 一个参数不够用，可以在装饰器外面再封装一层函数。

```
function testable(isTestable) {
  return function(target) {
    target.isTestable = isTestable;
  }
}

@testable(true)
class MyTestableClass {}
MyTestableClass.isTestable // true

@testable(false)
class MyClass {}
MyClass.isTestable // false
复制代码
```

- **装饰器对类的行为的改变，是代码编译时发生的，而不是在运行时。这意味着，装饰器能在编译阶段运行代码。也就是说，装饰器本质就是编译时执行的函数。**
- 想添加实例属性，可以通过目标类的`prototype`对象操作。

```
function testable(target) {
  target.prototype.isTestable = true;
}

@testable
class MyTestableClass {}

let obj = new MyTestableClass();
obj.isTestable // true
复制代码
```

- 装饰器函数`testable`是在目标类的`prototype`对象上添加属性，因此就可以在实例上调用。

方法的装饰

- 装饰器不仅可以装饰类，还可以装饰类的属性。例如：装饰器`readonly`用来装饰“类”的`getName`方法。装饰器函数`readonly`一共可以接受三个参数，`target`（装饰的对象）、`name`（装饰的属性名称），`descriptor`：属性的描述。

```
class Person {
  @readonly
  name() { return `${this.first} ${this.last}` }
}
function readonly(target, name, descriptor){
  console.log(target, name, descriptor)
  // descriptor对象原来的值如下
  // {
  //   value: specifiedFunction,
  //   enumerable: false,
  //   configurable: true,
  //   writable: true
  // };
  descriptor.writable = false; // 设置不可写
  return descriptor;
}
```

```
// 重新getNameAge不生效
Person.prototype.getName = function(name) {
  console.log(name)
}

const per = new Person()
console.log('getName', per.getName('detanx'))
复制代码
```

- 如果同一个方法有多个装饰器，会像剥洋葱一样，先从外到内进入，然后由内向外执行。

```
function dec(id){
  console.log('evaluated', id);
  return (target, property, descriptor) => console.log('executed', id);
}

class Example {
    @dec(1)
    @dec(2)
    method(){}
}
// evaluated 1
// evaluated 2
// executed 2
// executed 1
复制代码
```

为什么装饰器不能用于函数?

- **装饰器只能用于类和类的方法，不能用于函数，因为存在函数提升。**

```
var counter = 0;
var add = function () {
  counter++;
};

@add
function foo() {
}
复制代码
```

- 上面的代码，意图是执行后`counter`等于`1`，但是实际上结果是`counter`等于`0`。因为函数提升，使得实际执行的代码是下面这样。

```
@add
function foo() {
}

var counter;
var add;

counter = 0;

add = function () {
  counter++;
};
```
复制代码

- 另一个例子。

```
var readOnly = require("some-decorator");

@readOnly
function foo() {
}
```
复制代码

- 上面代码也有问题，因为实际执行是下面这样。

```
var readOnly;

@readOnly
function foo() {
}

readOnly = require("some-decorator");
```
复制代码

- **如果一定要装饰函数，可以采用高阶函数的形式直接执行。**

```
function doSomething(name) {
  console.log('Hello, ' + name);
}

function loggingDecorator(wrapped) {
  return function() {
    console.log('Starting');
    const result = wrapped.apply(this, arguments);
    console.log('Finished');
    return result;
  }
}

const wrapped = loggingDecorator(doSomething);
```
复制代码

core-decorators.js

- `core-decorators.js`是一个第三方模块，提供了几个常见的装饰器，通过它可以更好地理解装饰器。

@autobind

- `autobind`装饰器使得方法中的`this`对象，绑定原始对象。

```
import { autobind } from 'core-decorators';
class Person {
  @autobind
  getPerson() {
    return this;
  }
}

let person = new Person();
let getPerson = person.getPerson;

getPerson() === person;
// true
复制代码
```

@readonly

- `readonly`装饰器使得属性或方法不可写。

```
import { readonly } from 'core-decorators';
class Meal {
  @readonly
  entree = 'steak';
}

var dinner = new Meal();
dinner.entree = 'salmon';
// Cannot assign to read only property 'entree' of [object Object]
复制代码
```

@override

- `override`装饰器检查子类的方法，是否正确覆盖了父类的同名方法，如果不正确会报错。

```
import { override } from 'core-decorators';
class Parent {
  speak(first, second) {}
}

class Child extends Parent {
  @override
  speak() {}
  // SyntaxError: Child#speak() does not properly override Parent#speak(first,
second)
}

// or
class Child extends Parent {
  @override
  speaks() {}
```

```
  // SyntaxError: No descriptor matching Child#speaks() was found on the
prototype chain.
  //   Did you mean "speak"?
}
复制代码
```

@deprecate (别名@deprecated)

- `deprecate`或`deprecated`装饰器在控制台显示一条警告，表示该方法将废除。

```
import { deprecate } from 'core-decorators';
class Person {
  @deprecate
  facepalm() {}

  @deprecate('We stopped facepalming')
  facepalmHard() {}

  @deprecate('We stopped facepalming', { url:
'http://knowyourmeme.com/memes/facepalm' })
  facepalmHarder() {}
}

let person = new Person();

person.facepalm();
// DEPRECATION Person#facepalm: This function will be removed in future
versions.

person.facepalmHard();
// DEPRECATION Person#facepalmHard: We stopped facepalming

person.facepalmHarder();
// DEPRECATION Person#facepalmHarder: We stopped facepalming
// See http://knowyourmeme.com/memes/facepalm for more details.
复制代码
```

@suppressWarnings

- `suppressWarnings`装饰器抑制`deprecated`装饰器导致的`console.warn()`调用。但是，异步代码发出的调用除外。

```
import { suppressWarnings } from 'core-decorators';
class Person {
  @deprecated
  facepalm() {}

  @suppressWarnings
  facepalmWithoutWarning() {
    this.facepalm();
  }
}
let person = new Person();
person.facepalmWithoutWarning();
// no warning is logged
复制代码
```

Mixin

- 在装饰器的基础上，可以实现`Mixin`模式。`Mixin`模式，就是对象继承的一种替代方案，中文译为“混入”（`mix in`），意为在一个对象之中混入另外一个对象的方法。
- 部署一个通用脚本`mixins.js`，将`Mixin`写成一个装饰器。

函数实现

```
export function mixins(...list) {
  return function (target) {
    Object.assign(target.prototype, ...list);
  };
}
复制代码
```

- 使用`mixins`这个装饰器，为类“混入”各种方法。

```
import { mixins } from './mixins';

const Foo = {
  foo() { console.log('foo') }
};

@mixins(Foo)
class MyClass {}

let obj = new MyClass();
obj.foo() // "foo"
复制代码
```

类实现

- 函数写法会改写`MyClass`类的`prototype`对象，如果不喜欢这一点，也可以通过类的继承实现`Mixin`。

```
class MyClass extends MyBaseClass {
  /* ... */
}
复制代码
```

- `MyClass`继承了`MyBaseClass`。如果我们想在`MyClass`里面“混入”一个`foo`方法，一个办法是在`MyClass`和`MyBaseClass`之间插入一个混入类，这个类具有`foo`方法，并且继承了`MyBaseClass`的所有方法，然后`MyClass`再继承这个类。

```
let MyMixin = (superclass) => class extends superclass {
  foo() {
    console.log('foo from MyMixin');
  }
};
复制代码
```

- `MyMixin`是一个混入类生成器，接受`superclass`作为参数，然后返回一个继承`superclass`的子类，该子类包含一个`foo`方法。接着，目标类再去继承这个混入类，就达到了“混入”`foo`方法的目的。如果需要“混入”多个方法，就生成多个混入类。

```
class MyClass extends MyMixin(MyBaseClass) {
  /* ... */
}

let c = new MyClass();
c.foo(); // "foo from MyMixin"

// 混入多个
class MyClass extends Mixin1(Mixin2(MyBaseClass)) {
  /* ... */
}
复制代码
```

- 这种写法的一个好处，是可以调用`super`，因此可以避免在“混入”过程中覆盖父类的同名方法。

```
let Mixin1 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin1');
    if (super.foo) super.foo();
  }
};

let Mixin2 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin2');
    if (super.foo) super.foo();
  }
};

class S {
  foo() {
    console.log('foo from S');
  }
}

class C extends Mixin1(Mixin2(S)) {
  foo() {
    console.log('foo from C');
    super.foo();
  }
}
复制代码
```

- 码中，每一次混入发生时，都调用了父类的`super.foo`方法，导致父类的同名方法没有被覆盖，行为被保留了下来。

```
new C().foo()
// foo from C
// foo from Mixin1
// foo from Mixin2
// foo from S
复制代码
```

Trait

- `Trait` 也是一种装饰器，效果与 `Mixin` 类似，但是提供更多功能，比如防止同名方法的冲突、排除混入某些方法、为混入的方法起别名等等。
- 采用 `traits-decorator` 这个第三方模块作为例子。这个模块提供的 `traits` 装饰器，不仅可以接受对象，还可以接受 `ES6` 类作为参数。

```
import { traits } from 'traits-decorator';
class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') }
};

@traits(TFoo, TBar)
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
obj.bar() // bar
复制代码
```

- 上面代码中，通过 `traits` 装饰器，在 `MyClass` 类上面“混入”了 `TFoo` 类的 `foo` 方法和 `TBar` 对象的 `bar` 方法。
- `Trait` 不允许“混入”同名方法。

```
import { traits } from 'traits-decorator';
class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
};

@traits(TFoo, TBar)
class MyClass { }
// 报错
// throw new Error('Method named: ' + methodName + ' is defined twice.');
//        ^
// Error: Method named: foo is defined twice.
复制代码
```

- 上面代码中，`TFoo` 和 `TBar` 都有 `foo` 方法，结果 `traits` 装饰器报错。
- 一种解决方法是排除 `TBar` 的 `foo` 方法。

```
import { traits, excludes } from 'traits-decorator';
class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
```

```
};

@traits(TFoo, TBar::excludes('foo'))
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
obj.bar() // bar
复制代码
```

- 上面代码使用绑定运算符（`::`）在`TBar`上排除`foo`方法，混入时就不会报错了。
- 另一种方法是为`TBar`的`foo`方法起一个别名。

```
import { traits, alias } from 'traits-decorator';
class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
};

@traits(TFoo, TBar::alias({foo: 'aliasFoo'}))
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
obj.aliasFoo() // foo
obj.bar() // bar
复制代码
```

- 上面代码为`TBar`的`foo`方法起了别名`aliasFoo`，于是`MyClass`也可以混入`TBar`的`foo`方法了。
- `alias`和`excludes`方法，可以结合起来使用。

```
@traits(TBar::excludes('foo','bar')::alias({baz:'tBar'}))
class MyClass {}
复制代码
```

- 上面代码排除了`TBar`的`foo`方法和`bar`方法，为`baz`方法起了别名`tBaz`。
- `as`方法则为上面的代码提供了另一种写法。

```
@traits(TBar::as({excludes:['foo', 'bar'], alias: {baz: 'exampleBaz'}}))
class MyClass {}
复制代码
```

关注下面的标签，发现更多相似文章

- ES6-最新提案

do表达式

- 在`do`表达式 提案 之前，块级作用域是一个语句，将多个操作封装在一起，没有返回值。`do`表达式使得块级作用域可以变为表达式，也就是说可以返回值，**返回内部最后执行的表达式的值。**

```
// x是最后 t * t + 1 的值
let x = do {
  let t = f();
  t * t + 1;
};

// 不同情况执行不同函数
let x = do {
  if (foo()) { f() }
  else if (bar()) { g() }
  else { h() }
};
复制代码
```

throw表达式

- `JavaScript`语法规定`throw`是一个命令，用来抛出错误，不能用于表达式之中。

```
// 报错
console.log(throw new Error());
复制代码
```

- 现在有个 提案，`throw`可以直接用于表达式中。

```
// 参数的默认值
function save(filename = throw new TypeError("Argument required")) {
}

// 箭头函数的返回值
lint(ast, {
  with: () => throw new Error("avoid using 'with' statements.")
});

// 条件表达式
let typeof x === number ? x : throw new Error("Is NaN");

// 逻辑表达式
let val = value || throw new Error("Invalid value");
复制代码
```

- 语法上，`throw`表达式里面的throw不再是一个命令，而是一个运算符。为了避免与`throw`命令混淆，**规定`throw`出现在行首，一律解释为`throw`语句，而不是`throw`表达式。**

函数的部分执行

- 多参数的函数有时需要绑定其中的一个或多个参数，然后返回一个新函数。

```
function add(x, y) { return x + y; }
function add7(x) { return x + 7; }

// bind 方法
const add7 = add.bind(null, 7);

// 箭头函数
const add7 = x => add(x, 7);
复制代码
```

- 现在有一个 提案，使得绑定参数并返回一个新函数更加容易。这叫做函数的部分执行（`partial application`）。`?`是单个参数的占位符，`...`是多个参数的占位符。

```
f(x, ?)
f(x, ...)
f(?, x)
f(..., x)
f(?, x, ?)
f(..., x, ...)
复制代码
```

- **`?`和`...`只能出现在函数的调用之中，并且会返回一个新函数。**

```
const g = f(?, 1, ...);
// 等同于
const g = (x, ...y) => f(x, 1, ...y);
复制代码
```

- 函数的部分执行，也可以用于对象的方法。

```
let obj = {
  f(x, y) { return x + y; },
};

const g = obj.f(?, 3);
g(1) // 4
复制代码
```

注意点

1. 函数的部分执行是基于原函数的。如果原函数发生变化，部分执行生成的新函数也会立即反映这种变化。

```
let f = (x, y) => x + y;
const g = f(?, 3);
g(1); // 4

// 替换函数 f
f = (x, y) => x * y;
g(1); // 3
复制代码
```

1. 如果预先提供的那个值是一个表达式，那么这个表达式并不会在定义时求值，**而是在每次调用时求值。**

```
let a = 3;
const f = (x, y) => x + y;
const g = f(?, a);
g(1); // 4

// 改变 a 的值
a = 10;
g(1); // 11
复制代码
```

1. 如果新函数的参数多于占位符的数量，那么多余的参数将被忽略。

```
const f = (x, ...y) => [x, ...y];
const g = f(?, 1);
g(2, 3, 4); // [2, 1]
复制代码
```

1. **`...`只会被采集一次，如果函数的部分执行使用了多个`...`，那么每个`...`的值都将相同。**

```
const f = (...x) => x;
const g = f(..., 9, ...);
g(1, 2, 3); // [1, 2, 3, 9, 1, 2, 3]
复制代码
```

管道运算符

- 提案，`JavaScript`的管道是一个运算符，写作`|>`。它的左边是一个表达式，右边是一个函数。管道运算符把左边表达式的值，传入右边的函数进行求值。

```
x |> f
// 等同于
f(x)
复制代码
```

- **管道运算符只能传递一个值，这意味着它右边的函数必须是一个单参数函数。如果是多参数函数，就必须进行柯里化，改成单参数的版本。**

```
function double (x) { return x + x; }
function add (x, y) { return x + y; }

let person = { score: 25 };
person.score
  |> double
  |> (_ => add(7, _))
// 57
复制代码
```

- 管道运算符对于`await`函数也适用。

```
x |> await f
// 等同于
await f(x)

const userAge = userId |> await fetchUserById |> getAgeFromUser;
// 等同于
const userAge = getAgeFromUser(await fetchUserById(userId));
复制代码
```

数值分隔符

- 欧美语言中，较长的数值允许每三位添加一个分隔符（通常是一个逗号），增加数值的可读性。比如，`1000`可以写作`1,000`。
- 现在有一个提案，允许`JavaScript`的数值使用下划线（`_`）作为分隔符。

```
let budget = 1_000_000_000_000;
budget === 10 ** 12 // true
复制代码
```

- `JavaScript`的数值分隔符没有指定间隔的位数，小数和科学计数法也可以使用数值分隔符。

```
123_00 === 12_300 // true

12345_00 === 123_4500 // true
12345_00 === 1_234_500 // true

// 小数
0.000_001
// 科学计数法
1e10_000
复制代码
```

- 注意点。

    1. 不能在数值的最前面（`leading`）或最后面（`trailing`）。
    2. 不能两个或两个以上的分隔符连在一起。
    3. 小数点的前后不能有分隔符。
    4. 科学计数法里面，表示指数的`e`或`E`前后不能有分隔符。

    ```
    // 全部报错
    3_.141
    3._141
    1_e12
    1e_12
    123__456
    _1464301
    1464301_
    复制代码
    ```

- 除了十进制，其他进制的数值也可以使用分隔符。

```
// 二进制
0b1010_0001_1000_0101
// 十六进制
0xA0_B0_C0
复制代码
```

- **分隔符不能紧跟着进制的前缀**`0b`、`0B`、`0o`、`0O`、`0x`、`0X`。

```
// 报错
0_b111111000
0b_111111000
复制代码
```

- `Number()`、`parseInt()`、`parseFloat()`三个将字符串转成数值的函数，不支持数值分隔符。主要原因是提案的设计者认为，数值分隔符主要是为了编码时书写数值的方便，而不是为了处理外部输入的数据。

```
Number('123_456') // NaN
parseInt('123_456') // 123
复制代码
```

Math.signbit()

- `Math.sign()`用来判断一个值的正负，但是如果参数是`-0`，它会返回`-0`。实际编程中，判断一个值是`+0`还是`-0`非常麻烦，因为它们是相等的。

```
Math.sign(-0) // -0
+0 === -0 // true
复制代码
```

- 有一个提案，引入了`Math.signbit()`方法判断一个数是否设置了符号位。

```
Math.signbit(2) //false
Math.signbit(-2) //true
Math.signbit(0) //false
Math.signbit(-0) //true
复制代码
```

- 该方法的算法如下。
    1. 如果参数是`NaN`，返回`false`。
    2. 如果参数是`-0`，返回`true`。
    3. 如果参数是负值，返回`true`。
    4. 其他情况返回`false`。

双冒号运算符

- 箭头函数可以绑定`this`对象，大大减少了显式绑定`this`对象的写法（`call`、`apply`、`bind`），箭头函数并不适用于所有场合，所以现在有一个 提案，提出了“函数绑定”（`function bind`）运算符，用来取代`call`、`apply`、`bind`调用。函数绑定运算符是并排的两个冒号（`::`），双冒号左边是一个对象，右边是一个函数。

```
foo::bar;
// 等同于
bar.bind(foo);

foo::bar(...arguments);
// 等同于
bar.apply(foo, arguments);
```
复制代码

- 如果双冒号左边为空，右边是一个对象的方法，则等于将该方法绑定在该对象上面。

```
var method = obj::obj.foo;
// 等同于
var method = ::obj.foo;

let log = ::console.log;
// 等同于
var log = console.log.bind(console);
```
复制代码

- 如果双冒号运算符的运算结果，还是一个对象，就可以采用链式写法。

```
import { map, takeWhile, forEach } from "iterlib";

getPlayers()
::map(x => x.character())
::takeWhile(x => x.strength > 100)
::forEach(x => console.log(x));
```
复制代码

Realm API

- `Realm API` 提案 提供沙箱功能（`sandbox`），允许隔离代码，防止那些被隔离的代码拿到全局对象。提供一个`Realm()`构造函数，用来生成一个 `Realm` 对象。该对象的`global`属性指向一个新的顶层对象，这个顶层对象跟原始的顶层对象类似。`Realm` **顶层对象与原始顶层对象是两个对象。**

```
const globalOne = window;
const globalTwo = new Realm().global;
let a1 = globalOne.evaluate('[1,2,3]');
let a2 = globalTwo.evaluate('[1,2,3]');
a1.prototype === a2.prototype; // false
a1 instanceof globalTwo.Array; // false
a2 instanceof globalOne.Array; // false
```
复制代码

- `Realm` **沙箱里面只能运行** `ECMAScript` **语法提供的** `API`**，不能运行宿主环境提供的** `API`**。** 例如`console`不是语法标准，是宿主环境提供的。

```
globalTwo.evaluate('console.log(1)')
// throw an error: console is undefined

// 解决方法
globalTwo.console = globalOne.console;
复制代码
```

- `Realm()`构造函数可以接受一个参数对象，该参数对象的`intrinsics`属性可以指定`Realm`沙箱继承原始顶层对象的方法。

```
const r1 = new Realm();
r1.global === this;
r1.global.JSON === JSON; // false

const r2 = new Realm({ intrinsics: 'inherit' });
r2.global === this; // false
r2.global.JSON === JSON; // true
复制代码
```

#!命令

- `Unix`的命令行脚本都支持#!命令，又称为`Shebang`或`Hashbang`。这个命令放在脚本的第一行，用来指定脚本的执行器。

```
// Bash 脚本的第一行。
#!/bin/sh

// Python 脚本的第一行。
#!/usr/bin/env python
复制代码
```

- 现在有一个 提案，为`JavaScript`脚本引入了`#!`命令，写在脚本文件或者模块文件的第一行。

```
// 写在脚本文件第一行
#!/usr/bin/env node
'use strict';
console.log(1);

// 写在模块文件第一行
#!/usr/bin/env node
export {};
console.log(1);
复制代码
```

- 有了这一行以后，`Unix`命令行就可以直接执行脚本。

```
# 以前执行脚本的方式
$ node hello.js

# hashbang 的方式
$ ./hello.js
复制代码
```

- 对于`JavaScript`引擎来说，会把`#!`理解成注释，忽略掉这一行。

import.meta

- 开发者使用一个模块时，有时需要知道模板本身的一些信息（比如模块的路径）。现在有一个 提案，为 `import` 命令添加了一个元属性 `import.meta`，返回当前模块的元信息。
- `import.meta` **只能在模块内部使用，如果在模块外部使用会报错。**
- 这个属性返回一个对象，该对象的各种属性就是当前运行的脚本的元信息。具体包含哪些属性，标准没有规定，由各个运行环境自行决定。一般来说，`import.meta` 至少会有下面两个属性。

1. `import.meta.url`

- `import.meta.url` 返回当前模块的 `URL` 路径。举例来说，当前模块主文件的路径是 `https://foo.com/main.js`，`import.meta.url` 就返回这个路径。如果模块里面还有一个数据文件 `data.txt`，那么就可以用下面的代码，获取这个数据文件的路径。

```
new URL('data.txt', import.meta.url)
复制代码
```

- **`Node.js` 环境中，`import.meta.url` 返回的总是本地路径，即是 `file:URL` 协议的字符串，比如 `file:///home/user/foo.js`。**

1. `import.meta.scriptElement`

- `import.meta.scriptElement` 是浏览器特有的元属性，返回加载模块的那个 `<script>` 元素，相当于 `document.currentScript` 属性。

```
// HTML 代码为
// <script type="module" src="my-module.js" data-foo="abc"></script>

// my-module.js 内部执行下面的代码
import.meta.scriptElement.dataset.foo
// "abc"
复制代码
```

总结

- 在最新的提案中，有写特性在我看来还是非常有用的。例如
    1. `throw` 表达式，我们之前不能在块级作用域中使用，提案通过后我们可以直接在表达式中去做一些异常的处理。
    2. 管道运算符（`|>`），我们在连续调用的时候可以使用管道运算符直接写在一行，逻辑也更加的清楚。
    3. 双冒号运算符（`::`），之前我们绑定 `this` 要使用 `call`、`apply`、`bind` 或者使用箭头函数，但是有些情况箭头函数又会存在限制，所以双冒号运算符可以简介写法还可以处理箭头函数之外的特殊情况。
    4. 其他的特新都有各自的用处，可能我们在开发时有些情况没有遇到，存在即合理。
- `ES6`（`ES2016` 后统称）的特性越来越多，我们需要不断去熟悉新的特新有哪些，哪些是我们在开发中会运用到的，可以提高我们的开发效率，每次看完都会有不一样的收获，大家也可以隔一段时间去回顾一下，加深印象。
- `ES6` 常用但被忽略的方法系列文章到这儿就完结了，撒花🎉🎉🎉，感谢大家的支持。