04.24 【讲义】ES6/ESNext规范详解

#2021#

ECMAScript规范发展简介

相信大家都听说过ES6, ES7, ES2015, ES2016, ES2017…等等这些错综复杂的名字.
有的人以版本号描述, 比如ES6, ES7.
有的人以年份描述, 比如ES2015, ES2016.

那么他们之间到底是什么关系? 咱们就要从js的发展看起了.

历史

ECMAScript是由网景的布兰登·艾克开发的一种脚本语言的标准化规范；最初命名为Mocha，后来改名为LiveScript，最后重命名为JavaScript[1]。1995年12月，升阳与网景联合发表了JavaScript[2]。1996年11月，网景公司将JavaScript提交给欧洲计算机制造商协会进行标准化。ECMA-262的第一个版本于1997年6月被Ecma组织采纳。ECMAScript是由ECMA-262标准化的脚本语言的名称。

尽管JavaScript和JScript与ECMAScript兼容，但包含超出ECMAScript的功能[3]。

–来自wikipedia.

总之: 严格来说, ES6是指2015年6月发布的ES2015标准, 但是很多人在谈及ES6的时候, 都会把ES2016 ES2017等标准的内容也带进去.
所以严谨的说, 在谈论ECMAScript标准的时候, 用年份更好一些. 但是也无所谓, 纠结这个没多大意义。

ESNext

前面见到了各种ES6 7, ES2015, 怎么又冒出一个ESNext? ESNext是什么?

其实ESNext是一个泛指, 它永远指向下一个版本. 比如当前最新版本是ES2020, 那么ESNext指的就是2021年6月将要发布的标准.

ES6及以后新增的常用API解析

let 和 const

先来一道经典面试题

```
for(var i=0;i<=3;i++){
    setTimeout(function() {
        console.log(i)
    }, 10);
}
```

分别会输出什么? 为什么? 如何修改可以使其输出0,1,2,3?

```
for(var i = 0; i <=3; i++) {
    (function (i) {
        setTimeout(function () {
            console.log(i);
        }, 10);
    })(i);
}

for(let i=0;i<=3;i++){
    setTimeout(function() {
        console.log(i)
    }, 10);
}
```

原因:
Var定义的变量是全局的, 所以全局只有一个变量i.
setTimeout是异步, 在下一轮事件循环, 等到执行的时候, 去找i变量的引用。所以函数找到了遍历完后的i, 此时它已经变成了4。

1. 而let引入了块级作用域的概念, 创建setTimeout函数时，变量i在作用域内。对于循环的每个迭代，引用的i是i的不同实例。

2. 还存在变量提升的问题

```
console.log(i)
```

```
var i = 1;

console.log(letI)
let letI = 2;
```

3. Const就很简单了, 在let的基础上, 不可被修改.

箭头函数

1. 最大的区别：箭头函数里的this是定义的时候决定的, 普通函数里的this是使用的时候决定的。

```
const teacher = {
    name: 'lubai',
    getName: function() {
        return `${this.name}`
    }
}
console.log(teacher.getName());

const teacher = {
    name: 'lubai',
    getName: () => {
        return `${this.name}`
    }
}
console.log(teacher.getName());
```

2. 简写箭头函数

```
const arrowFn = (value) => Number(value);

console.log(arrowFn('aaa'))
```

3. 注意, 箭头函数不能被用作构造函数

构造函数会干嘛? 改变this指向到新实例出来的对象.
箭头函数会干嘛? this指向是定义的时候决定的.

class

```
class Test {
    _name = '';
    constructor() {
        this.name = 'lubai';
    }

    static getFormatName() {
        return `${this.name} - xixi`;
    }

    get name() {
        return this._name;
    }

    set name(val) {
        console.log('name setter');
        this._name = val;
    }
}

console.log(new Test().name)
console.log(Test.getFormatName())
```

模板字符串

```
const b = 'lubai'
const a  = `${b} - xxxx`;
const c = `我是换行
我换行了!
我又换行了!
```

```
`;
```

面试题来一道. 编写render函数, 实现template render功能.

```
const year = '2021';
const month = '10';
const day = '01';

let template = '${year}-${month}-${day}';
let context = { year, month, day };

const str = render(template)({year,month,day});

console.log(str) // 2021-10-01

function render(template) {
    return function(context) {
        return template.replace(/\$\{(.*?)\}/g, (match, key) => context[key]);
    }
}
```

解构

1. 数组的解构

```
// 基础类型解构
let [a, b, c] = [1, 2, 3]
console.log(a, b, c) // 1, 2, 3

// 对象数组解构
let [a, b, c] = [{name: '1'}, {name: '2'}, {name: '3'}]
console.log(a, b, c) // {name: '1'}, {name: '2'}, {name: '3'}

// ...解构
let [head, ...tail] = [1, 2, 3, 4]
console.log(head, tail) // 1, [2, 3, 4]
```

```
// 嵌套解构
let [a, [b], d] = [1, [2, 3], 4]
console.log(a, b, d) // 1, 2, 4

// 解构不成功为undefined
let [a, b, c] = [1]
console.log(a, b, c) // 1, undefined, undefined

// 解构默认赋值
let [a = 1, b = 2] = [3]
console.log(a, b) // 3, 2
```

2. 对象的结构

```
// 对象属性解构
let { f1, f2 } = { f1: 'test1', f2: 'test2' }
console.log(f1, f2) // test1, test2

// 可以不按照顺序，这是数组解构和对象解构的区别之一
let { f2, f1 } = { f1: 'test1', f2: 'test2' }
console.log(f1, f2) // test1, test2

// 解构对象重命名
let { f1: rename, f2 } = { f1: 'test1', f2: 'test2' }
console.log(rename, f2) // test1, test2

// 嵌套解构
let { f1: {f11}} = { f1: { f11: 'test11', f12: 'test12' } }
console.log(f11) // test11

// 默认值
let { f1 = 'test1', f2: rename = 'test2' } = { f1: 'current1', f2: 'current2'}
console.log(f1, rename) // current1, current2
```

3. 解构的原理是什么?

针对可迭代对象的Iterator接口，通过遍历器按顺序获取对应的值进行赋值.

3.1 那么 Iterator 是什么?

Iterator是一种接口，为各种不一样的数据解构提供统一的访问机制。任何数据解构只要有Iterator接口，就能通过遍历操作，依次按顺序处理数据结构内所有成员。ES6中的for of的语法相当于遍历器，会在遍历数据结构时，自动寻找Iterator接口。

3.2 Iterator有什么用?

- 为各种数据解构提供统一的访问接口
- 使得数据解构能按次序排列处理
- 可以使用ES6最新命令 for of进行遍历

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

const iterator = generateIterator([0, 1, 2])

console.log(iterator.next())
console.log(iterator.next())
console.log(iterator.next())
console.log(iterator.next())
```

3.3 可迭代对象是什么?

可迭代对象是Iterator接口的实现。这是ECMAScript 2015的补充，它不是内置或语法，而仅仅是协议。任何遵循该协议点对象都能成为可迭代对象。可迭代对象得有两个协议：可迭代协议和迭代器协议。

- 可迭代协议：对象必须实现iterator方法。即对象或其原型链上必须有一个名叫Symbol.iterator的属性。该属性的值为无参函数，函数返回迭代器协议。
- 迭代器协议：定义了标准的方式来产生一个有限或无限序列值。其要求必须实现一个next()方法，该方法返回对象有done(boolean)和value属性。

3.4 我们自己来实现一个可以for of遍历的对象?

通过以上可知，自定义数据结构，只要拥有Iterator接口，并将其部署到自己的Symbol.iterator属性上，就可以成为可迭代对象，能被for of循环遍历。

```
const obj = {
    count: 0,
    [Symbol.iterator]: () => {
        return {
            next: () => {
                obj.count++;
                if (obj.count <= 10) {
                    return {
                        value: obj.count,
                        done: false
                    }
                } else {
                    return {
                        value: undefined,
                        done: true
                    }
                }
            }
        }
    }
}

for (const item of obj) {
```

```
    console.log(item)
}
```

或者

```
const iterable = {
    0: 'a',
    1: 'b',
    2: 'c',
    length: 3,
    [Symbol.iterator]: Array.prototype[Symbol.iterator],
};

for (const item of iterable) {
    console.log(item);
}
```

遍历

1. for in

遍历数组时，key为数组下标字符串；遍历对象，key为对象字段名。

```
let obj = {a: 'test1', b: 'test2'}
for (let key in obj) {
    console.log(key, obj[key])
}
```

缺点：

- for in 不仅会遍历当前对象，还包括原型链上的可枚举属性
- for in 没有break中断
- for in 不适合遍历数组，主要应用为对象

2. for of

可迭代对象（包括 Array，Map，Set，String，TypedArray，arguments对象，NodeList对象）上创建一个迭代循环,调用自定义迭代钩子，并为每个不同属性的值执行语句。

```
let arr = [{age: 1}, {age: 5}, {age: 100}, {age: 34}]
for(let {age} of arr) {
    if (age > 10) {
        break // for of 允许中断
    }
    console.log(age)
}
```

优点：

- for of 仅遍历当前对象
- for of 可与break，continue，return配合

Object

1. Object.keys

该方法返回一个给定对象的自身可枚举属性组成的数组。

```
const obj = { a: 1, b: 2 };
const keys = Object.keys(obj); // [a, b]
```

手写实现一个函数模拟Object.keys?

```
function getObjectKeys(obj) {
    const result = [];
    for (const prop in obj) {
        if (obj.hasOwnProperty(prop)) {
            result.push(prop);
        }
```

```
    }

    return result;
}

console.log(getObjectKeys({
    a: 1,
    b: 2
}))
```

2. Object.values

该方法返回一个给定对象自身的所有可枚举属性值的数组。

```
const obj = { a: 1, b: 2 };
const keys = Object.keys(obj); // [1, 2]
```

手写实现一个函数模拟Object.values?

```
function getObjectValues(obj) {
    const result = [];
    for (const prop in obj) {
        if (obj.hasOwnProperty(prop)) {
            result.push(obj[prop]);
        }
    }

    return result;
}

console.log(getObjectValues({
    a: 1,
    b: 2
}))
```

3. Object.entries

该方法返回一个给定对象自身可枚举属性的键值对数组。

```
const obj = { a: 1, b: 2 };
const keys = Object.entries(obj); // [ [ 'a', 1 ], [ 'b', 2 ] ]
```

手写实现一个函数模拟Object.entries?

```
function getObjectEntries(obj) {
    const result = [];
    for (const prop in obj) {
        if (obj.hasOwnProperty(prop)) {
            result.push([prop, obj[prop]]);
        }
    }

    return result;
}

console.log(getObjectEntries({
    a: 1,
    b: 2
}))
```

4. Object.getOwnPropertyNames

该方法返回一个数组，该数组对元素是 obj自身拥有的枚举或不可枚举属性名称字符串。

看一下这段代码会输出什么?

```
Object.prototype.aa = '1111';
```

```
const testData = {
    a: 1,
    b: 2
}

for (const key in testData) {
    console.log(key);
}

console.log(Object.getOwnPropertyNames(testData))
```

5. Object.getOwnPropertyDescriptor

什么是descriptor? 对象对应的属性描述符, 是一个对象. 包含以下属性:

- configurable。 如果为false，则任何尝试删除目标属性或修改属性特性（writable, configurable, enumerable）的行为将被无效化。所以通常属性都有特性时，可以把configurable设置为true即可。
- writable 是否可写。设置成 false，则任何对该属性改写的操作都无效（但不会报错，严格模式下会报错），默认false。
- enumerable。是否能在for-in循环中遍历出来或在Object.keys中列举出来。

```
const object1 = {};
Object.defineProperty(object1, 'p1', {
  value: 'lubai',
  writable: false
});

object1.p1 = 'not lubai';

console.log(object1.p1);
```

讲到了defineProperty, 那么肯定离不开Proxy.

```
// const obj = {};
// let val = undefined;
```

```
// Object.defineProperty(obj, 'a', {
//     set: function (value) {
//         console.log(`${value} - xxxx`);
//         val = value;
//     },
//     get: function () {
//         return val;
//     },
//     configurable: true,
// })

// obj.a = 111;

// console.log(obj.a)

const obj = new Proxy({}, {
    get: function (target, propKey, receiver) {
        console.log(`getting ${propKey}`);
        return target[propKey];
    },
    set: function (target, propKey, value, receiver) {
        console.log(`setting ${propKey}`);
        return Reflect.set(target, propKey, value, receiver);
    }
});

obj.something = 1;
console.log(obj.something);
```

Reflect又是个什么东西?

- 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。现阶段，某些方法同时在Object和Reflect对象上部署，未来的新方法将只部署在Reflect对象上。也就是说，从Reflect对象上可以拿到语言内部的方法
- 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。
- 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。这就让Proxy对象可以方便地调用对应的Reflect方法，完成默认行为，作为修改行为的基础。也就是说，不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

但是要注意, 通过defineProperty设置writable为false的对象, 就不能用Proxy了

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


const proxy = new Proxy(target, {
    get(target, propKey) {
        return 'abc';
    }
});

proxy.foo
```

6. Object.create()

Object.create()方法创建一个新的对象，并以方法的第一个参数作为新对象的**proto**属性的值(根据已有的对象作为原型，创建新的对象。)
Object.create()方法还有第二个可选参数，是一个对象，对象的每个属性都会作为新对象的自身属性，对象的属性值以descriptor（Object.getOwnPropertyDescriptor(obj, 'key')）的形式出现，且enumerable默认为false

```
const person = {
    isHuman: false,
    printIntroduction: function () {
        console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);
    }
```

```
};
const me = Object.create(person);

me.name = "lubai";
me.isHuman = true;
me.printIntroduction();

console.log(person);

const myObject = Object.create(null)
```

传入第二个参数是怎么操作的呢?

```
function Person(name, sex) {
    this.name = name;
    this.sex = sex;
}

const b = Object.create(Person.prototype, {
    name: {
        value: 'coco',
        writable: true,
        configurable: true,
        enumerable: true,
    },
    sex: {
        enumerable: true,
        get: function () {
            return 'hello sex'
        },
        set: function (val) {
            console.log('set value:' + val)
        }
    }
})

console.log(b.name)
```

```
console.log(b.sex)
```

那么Object.create(null)的意义是什么呢? 平时创建一个对象Object.create({}) 或者 直接声明一个{} 不就够了?

Object.create(null)创建一个对象，但这个对象的原型链为null，即Fn.prototype = null

```
const b = Object.create(null) // 返回纯{}对象，无prototype

b // {}
b.__proto__ // undefined
b.toString() // throw error
```

所以当你要创建一个非常干净的对象, 没有任何原型链上的属性, 那么就使用 Object.create(null). For in 遍历的时候也不需要考虑原型链属性了.

1. Object.assign

浅拷贝, 类似于 { ...a, ...b };

```
function shallowClone(source) {
    const target = {};
    for (const i in source) {
        if (source.hasOwnProperty(i)) {
            target[i] = source[i];
        }
    }

    return target;
}

const a = {
    b: 1,
    c: {
        d: 111
```

```
    }
}

const b = shallowClone(a);



b.b = 2222;

b.c.d = 333;

console.log(b)
console.log(a)
```

8. Object.is

```
const a = {
    name: 1
};
const b = a;
console.log(Object.is(a, b))

console.log(Object.is({}, {}))
```

Promise

大部分promise都讲过了, 大概再来复习一下promise.all吧

```
function PromiseAll(promiseArray) {
    return new Promise(function (resolve, reject) {
        //判断参数类型
        if (!Array.isArray(promiseArray)) {
            return reject(new TypeError('arguments muse be an array'))
        }
        let counter = 0;
        let promiseNum = promiseArray.length;
        let resolvedArray = [];
```

```
        for (let i = 0; i < promiseNum; i++) {
            // 3. 这里为什么要用Promise.resolve?
            Promise.resolve(promiseArray[i]).then((value) => {
                counter++;
                resolvedArray[i] = value; // 2. 这里直接Push, 而不是用索引赋值, 有问题吗
                if (counter == promiseNum) { // 1. 这里如果不计算counter++, 直接判断resolvedArr.length === promiseNum,  会有问题吗?
                    // 4. 如果不在.then里面, 而在外层判断, 可以吗?
                    resolve(resolvedArray)
                }
            }).catch(e => reject(e));
        }
    })
}

// 测试
const pro1 = new Promise((res, rej) => {
    setTimeout(() => {
        res('1')
    }, 1000)
})
const pro2 = new Promise((res, rej) => {
    setTimeout(() => {
        res('2')
    }, 2000)
})
const pro3 = new Promise((res, rej) => {
    setTimeout(() => {
        res('3')
    }, 3000)
})

const proAll = PromiseAll([pro1, pro2, pro3])
    .then(res =>
        console.log(res) // 3秒之后打印 ["1", "2", "3"]
    )
    .catch((e) => {
        console.log(e)
```

```
    })
```

再来写一个Promise.allSeettled, 需要返回所有promise的状态和结果

```
function PromiseAllSettled(promiseArray) {
    return new Promise(function (resolve, reject) {
        //判断参数类型
        if (!Array.isArray(promiseArray)) {
            return reject(new TypeError('arguments muse be an array'))
        }
        let counter = 0;
        const promiseNum = promiseArray.length;
        const resolvedArray = [];
        for (let i = 0; i < promiseNum; i++) {
            Promise.resolve(promiseArray[i])
                .then((value) => {
                    resolvedArray[i] = {
                        status: 'fulfilled',
                        value
                    };

                })
                .catch(reason => {
                    resolvedArray[i] = {
                        status: 'rejected',
                        reason
                    };
                })
                .finally(() => {
                    counter++;
                    if (counter == promiseNum) {
                        resolve(resolvedArray)
                    }
                })
        }
    })
}
```

数组

1. Array.flat

flat() 方法会按照一个可指定的深度递归遍历数组，并将所有元素与遍历到的子数组中的元素合并为一个新数组返回

```
const arr1 = [1, 2, [3, 4]];
arr1.flat();
// [1, 2, 3, 4]

const arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat();
// [1, 2, 3, 4, [5, 6]]

const arr3 = [1, 2, [3, 4, [5, 6]]];
arr3.flat(2);
// [1, 2, 3, 4, 5, 6]

//使用 Infinity，可展开任意深度的嵌套数组
const arr4 = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]];
arr4.flat(Infinity);
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

如何模拟实现Array.flat?

```
// 使用 reduce、concat 和递归展开无限多层嵌套的数组
const arr1 = [1, 2, 3, [1, 2, 3, 4, [2, 3, 4]]];

function flatDeep(arr, d = 1) {
    if (d > 0) {
        return arr.reduce((res, val) => {
            if (Array.isArray(val)) {
```

```
                res = res.concat(flatDeep(val, d - 1))
            } else {
                res = res.concat(val);
            }
            return res;
        }, [])
    } else {
        return arr.slice()
    }
};

console.log(flatDeep(arr1, Infinity))
// [1, 2, 3, 1, 2, 3, 4, 2, 3, 4]
```

如果不考虑深度,咱们直接给他无限打平

```
function flatten(arr) {
    let res = [];
    let length = arr.length;
    for (let i = 0; i < length; i++) {
        if (Object.prototype.toString.call(arr[i]) === '[object Array]') {
            res = res.concat(flatten(arr[i]))
        } else {
            res.push(arr[i])
        }
    }
    return res
}

// 如果数组元素都是Number类型
function flatten(arr) {
  return arr.toString().split(',').map(item => +item)
}

function flatten(arr){
    while(arr.some(item=>Array.isArray(item))){
        arr = [].concat(...arr);
```

```
    }
    return arr;
}
```

2. Array.includes

includes() 方法用来判断一个数组是否包含一个指定的值，根据情况，如果包含则返回 true，否则返回false。

```
const array1 = [1, 2, 3];

console.log(array1.includes(2));

const pets = ['cat', 'dog', 'bat'];

console.log(pets.includes('cat'));
```

其实它有两个参数, 只不过我们平时只使用一个.

- valueToFind
  需要查找的元素值。

- fromIndex 可选
  从fromIndex 索引处开始查找 valueToFind。如果为负值，则按升序从 array.length + fromIndex 的索引开始搜 （即使从末尾开始往前跳 fromIndex 的绝对值个索引，然后往后搜寻）。默认为 0。

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

// fromIndex 大于等于数组长度
var arr = ['a', 'b', 'c'];
```

```
arr.includes('c', 3);   // false
arr.includes('c', 100); // false

// 计算出的索引小于 0
var arr = ['a', 'b', 'c'];

arr.includes('a', -100); // true
arr.includes('b', -100); // true
arr.includes('c', -100); // true
```

3. Array.find

find() 方法返回数组中满足提供的测试函数的第一个元素的值。否则返回 undefined。

callback

在数组每一项上执行的函数，接收 3 个参数：

- element

  当前遍历到的元素。
- index可选

  当前遍历到的索引。
- array可选

  数组本身。

```
const test = [
    {name: 'lubai', age: 11 },
    {name: 'xxx', age: 100 },
    {name: 'nnn', age: 50}
];

function findLubai(teacher) {
    return teacher.name === 'lubai';
}

console.log(test.find(findLubai));
```

4. Array.from

4.1 Array.from() 方法从一个类似数组或可迭代对象创建一个新的，浅拷贝的数组实例。

- arrayLike
  想要转换成数组的伪数组对象或可迭代对象。
- mapFn 可选
  如果指定了该参数，新数组中的每个元素会执行该回调函数。

4.2 Array.from() 可以通过以下方式来创建数组对象：

- 伪数组对象（拥有一个 length 属性和若干索引属性的任意对象）
- 可迭代对象（可以获取对象中的元素,如 Map和 Set 等）

```
console.log(Array.from('foo'));

console.log(Array.from([1, 2, 3], x => x + x));

const set = new Set(['foo', 'bar', 'baz', 'foo']);
Array.from(set);
// [ "foo", "bar", "baz" ]

const map = new Map([[1, 2], [2, 4], [4, 8]]);
Array.from(map);
// [[1, 2], [2, 4], [4, 8]]

const mapper = new Map([['1', 'a'], ['2', 'b']]);
Array.from(mapper.values());
// ['a', 'b'];

Array.from(mapper.keys());
// ['1', '2'];
```

所以数组去重我们可以怎么做?

```
function unique (arr) {
  return Array.from(new Set(arr))
```

```
  // return [...new Set(arr)]
}
const test = [1,1,'true','true',true,true,15,15,false,false,
undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a'];
console.log(unique(test));


function unique(arr) {
    const map = new Map();
    const array = []; // 数组用于返回结果
    for (let i = 0; i < arr.length; i++) {
        if (!map.has(arr[i])) { // 如果有该key值
            array.push(arr[i]);
            map.set(arr[i], true);
        }
    }
    return array;
}

function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    const array = [];
    for (let i = 0; i < arr.length; i++) {
        if (!array.includes(arr[i])) { //includes 检测数组是否有某个值
            array.push(arr[i]);
        }
    }
    return array
}
```

5. Array.of

Array.of() 方法创建一个具有可变数量参数的新数组实例，而不考虑参数的数量或类型。

```
Array.of(7);       // [7]
Array.of(1, 2, 3); // [1, 2, 3]
```

那怎么去模拟实现它呢?

```
Array.of = function() {
    return Array.prototype.slice.call(arguments);
};
```

async await yeild

之前promise的课上见过了, 就不细讲了.

babel编译工具链的使用

https://astexplorer.net/ 查看AST.

Babel 是一个工具链，主要用于将 ECMAScript 2015+ 版本的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中

Babel提供了插件化的功能, 一切功能都可以以插件来实现. 方便使用和弃用.

抽象语法书 AST

这个处理过程中的每一步都涉及到创建或是操作抽象语法树，亦称 AST。

这样一段代码, 会被转换成什么呢?

```
function square(n) {
  return n * n;
}
```

大概是这样的

```
{
  type: "FunctionDeclaration",
```

```
  id: {
    type: "Identifier",
    name: "square"
  },
  params: [{
    type: "Identifier",
    name: "n"
  }],
  body: {
    type: "BlockStatement",
    body: [{
      type: "ReturnStatement",
      argument: {
        type: "BinaryExpression",
        operator: "*",
        left: {
          type: "Identifier",
          name: "n"
        },
        right: {
          type: "Identifier",
          name: "n"
        }
      }
    }]
  }
}
```

每一层都有相同的结构

```
{
  type: "FunctionDeclaration",
  id: {...},
  params: [...],
  body: {...}
}
{
```

```
  type: "Identifier",
  name: ...
}
{
  type: "BinaryExpression",
  operator: ...,
  left: {...},
  right: {...}
}
```

这样的每一层结构也被叫做 节点（Node）。 一个 AST 可以由单一的节点或是成百上千个节点构成。 它们组合在一起可以描述用于静态分析的程序语法。

字符串形式的 type 字段表示节点的类型（如： “FunctionDeclaration”，”Identifier”，或 “BinaryExpression”）。 每一种类型的节点定义了一些附加属性用来进一步描述该节点类型。

Babel 还为每个节点额外生成了一些属性，用于描述该节点在原始代码中的位置。比如 start end

Babel 的处理步骤

Babel 的三个主要处理步骤分别是： 解析（parse），转换（transform），生成（generate）。.

1. 解析

解析步骤接收代码并输出 AST。

- 词法分析

词法分析阶段把字符串形式的代码转换为 令牌（tokens） 流。.

你可以把令牌看作是一个扁平的语法片段数组：

n * n

转换成tokens是这样的

[

{ type: { … }, value: “n”, start: 0, end: 1, loc: { … } },
{ type: { … }, value: “*”, start: 2, end: 3, loc: { … } },
{ type: { … }, value: “n”, start: 4, end: 5, loc: { … } },
…
]

- 语法分析
  语法分析阶段会把一个令牌流转换成 AST 的形式。 这个阶段会使用令牌中的信息把它们转换成一个 AST 的表述结构，这样更易于后续的操作。

2. 转换
   转换步骤接收 AST 并对其进行遍历，在此过程中对节点进行添加、更新及移除等操作。 这是 Babel 或是其他编译器中最复杂的过程 同时也是插件将要介入工作的部分。

3. 生成

   代码生成步骤把最终（经过一系列转换之后）的 AST 转换成字符串形式的代码，同时还会创建源码映射（source maps）。.

   代码生成其实很简单：深度优先遍历整个 AST，然后构建可以表示转换后代码的字符串。

简单写一个babel插件

1. 一个插件就是一个函数

```
export default function(babel) {
}
// babel里我们主要用到types属性

export default function({ types: t }) {
}
```

Babel Types模块拥有每一个单一类型节点的定义，包括节点包含哪些属性，什么是合法值，如何构建节点、遍历节点，以及节点的别名等信息。

单一节点类型的定义形式如下：

```
defineType("BinaryExpression", {
  builder: ["operator", "left", "right"],
  fields: {
    operator: {
      validate: assertValueType("string")
    },
    left: {
      validate: assertNodeType("Expression")
    },
    right: {
      validate: assertNodeType("Expression")
    }
  },
  visitor: ["left", "right"],
  aliases: ["Binary", "Expression"]
});
```

2. 返回一个对象

Visitor 属性是这个插件的主要访问者。visitor中的每个函数都接收两个参数 state 和 path.

```
export default function({ types: t }) {
  return {
    visitor: {
    }
  };
};
```

AST 通常会有许多节点，那么节点直接如何相互关联呢？ 我们可以使用一个可操作和访问的巨大可变对象表示节点之间的关联关系，或者也可以用Paths（路径）来简化这件事情。.

Path 是表示两个节点之间连接的对象。

将子节点 Identifier 表示为一个路径（Path）的话，看起来是这样的：

```
{
  "parent": {
    "type": "FunctionDeclaration",
    "id": {...},
    ....
  },
  "node": {
    "type": "Identifier",
    "name": "square"
  }
}
```

3. 创建plugin.js

yarn add @babel/core
yarn add babel-template

```
const template = require('babel-template');

const temp = template("var b = 1")

module.exports = function ({
    types: t
}) {
    // 插件内容
    return {
        visitor: {
            // 接收两个参数path, state
            VariableDeclaration(path, state) {
                // 找到AST节点
                const node = path.node;
                // 判断节点类型 是否是变量节点, 申明方式是const
                if (t.isVariableDeclaration(node, {
                        kind: "const"
                    })) {
                    // 将const 声明编译为let
```

```
                    node.kind = "let";
                    // var b = 1 的AST节点
                    const insertNode = temp();
                    // 插入一行代码var b = 1
                    path.insertBefore(insertNode);
                }
            }
        }
    }
}
```

4. 使用插件

babel.js

```
const myPlugin = require('./plugin')
const babel = require('@babel/core');
const content = 'const name = lubai';
// 通过你编写的插件输出的代码
const {
    code
} = babel.transform(content, {
    plugins: [
        myPlugin
    ]
});

console.log(code);
```