1.2 ES6 新语法

1.2.1 let

let 声明的变量只在 let 命令所在的代码块内有效。

```
{
  let a = 0;
  var b = 1;
}
a  // ReferenceError: a is not defined
b  // 1
```

let 声明不存在变量提升，在声明变量之前，变量不存在会报错。

```
console.log(a);  //ReferenceError: a is not defined
let a = "apple";

console.log(b);  //undefined
var b = "banana";
```

综合示例：

```
for (var i = 0; i < 10; i++) {
  setTimeout(function(){
    console.log(i);
  }, 10)
}
// 输出十个 10
for (let j = 0; j < 10; j++) {
  setTimeout(function(){
    console.log(j);
  }, 10)
}
// 输出 0123456789
```

说明：

- 变量 i 是用 var 声明的，在全局范围内有效，所以全局中只有一个变量 i, 每次循环时，setTimeout 定时器里面的 i 指的是全局变量 i，而循环里的十个 setTimeout 是在循环结束后才执行，所以此时的 i 都是 10。
- 变量 j 是用 let 声明的，当前的 j 只在本轮循环中有效，每次循环的 j 其实都是一个新的变量，所以 setTimeout 定时器里面的 j 其实是不同的变量，即最后输出 12345。（若每次循环的变量 j 都是重新声明的，如何知道前一个循环的值？这是因为 JavaScript 引擎内部会记住前一个循环的值）。

> PS：**setTimeOut() 是一个异步函数，当JS遇到异步函数的时候，会把异步函数插入到队列中等待**。也就是所谓的插队。而setTimeOut 就是一个异步函数。所以当JS检测到setTimeOut()的时候，会把setTimeOut()插入到队列中，然后继续执行后面的代码，也就是接下来的循环。

1.2.2 const

const 声明一个只读的常量，声明时必须进行初始化，常量的值不能改变。

示例：

```
const PI = "3.1415926";
```

1.2.3 扩展运算符

扩展运算符（ spread ）是三个点（...），主要用于函数调用，它能够将一个数组转为用逗号分隔的参数序列，也可以用于数组的合并、对象合并、解构赋值等。

示例：

```
//数组合并
const color = ['red', 'yellow'] ;
const colors = [...color, 'green', 'pink'] ;
console.log(colors); //[red, yellow, green, pink]
//对象合并
const chars = { fist: 'a', second: 'b'};
const newChars = { ...chars, third: 'c' } ;
console.log(newChars); //{ "fist": "a", "second": "b", "third": "c"}
```

1.2.4 解构赋值

解构赋值是对赋值运算符的扩展。他是一种针对数组或者对象进行模式匹配，然后对其中的变量进行赋值。在代码书写上简洁且易读，语义更加清晰明了；也方便了复杂对象中数据字段获取。

在解构中，有下面两部分参与：

- 解构的源，解构赋值表达式的右边部分。
- 解构的目标，解构赋值表达式的左边部分。

基本

```
let [a, b, c] = [1, 2, 3];

let arr = [1, 2, 3]
let a = arr[0]
let b = arr[1]
let c = arr[2]
// a = 1
// b = 2
// c = 3
```

可嵌套

```
let [a, [[b], c]] = [1, [[2], 3]];
// a = 1
// b = 2
// c = 3
```

可忽略

```
let [a, , b] = [1, 2, 3];
// a = 1
// b = 3
```

不完全解构

```
let [a = 1, b] = []; // a = 1, b = undefined
```

扩展（剩余）运算符

```
let [a, ...b] = [1, 2, 3];
//a = 1
//b = [2, 3]
```

字符串等

在数组的解构中，解构的目标若为可遍历对象，皆可进行解构赋值。可遍历对象即实现 Iterator 接口的数据。

```
let [a, b, c, d, e] = 'hello';
// a = 'h'
// b = 'e'
// c = 'l'
// d = 'l'
// e = 'o'
```

解构默认值

```
let [a = 2] = [undefined]; // a = 2
```

当解构模式有匹配结果，且匹配结果是 undefined 时，会触发默认值作为返回结果。

```
let [a = 3, b = a] = [];     // a = 3, b = 3
let [a = 3, b = a] = [1];    // a = 1, b = 1
let [a = 3, b = a] = [1, 2]; // a = 1, b = 2
```

说明：

- a 与 b 匹配结果为 undefined ，触发默认值：**a = 3; b = a =3**
- a 正常解构赋值，匹配结果：a = 1，b 匹配结果 undefined ，触发默认值：**b = a =1**

- a 与 b 正常解构赋值，匹配结果：**a = 1，b = 2**

1.2.5对象解构

JS对象也可以解构，例如

```
let user = {name:"david",age:20}

let {name,age} = user //解构user对象，注意变量名必须和属性名一致

console.log(name,age)
```

1.2.6 模板字符串

模板字符串相当于加强版的字符串，用反引号 `,除了作为普通字符串，还可以用来定义多行字符串，还可以在字符串中加入变量和表达式。

基本用法

```
let string1 =  `Hey,
can you stop angry now?`;
console.log(string1);
// Hey,
// can you stop angry now?
```

变量和表达式

```
let name = "Mike";
let age = 27;
let info = `My Name is ${name},I am ${age+1} years old next year.`
console.log(info);
// My Name is Mike,I am 28 years old next year.
```

1.4 ES6 函数

1.4.1 函数参数默认值

声明函数参数时可以给参数赋初值。

```
function fn(name,age=17){
 console.log(name+","+age);
}
fn("Amy",18);  // Amy,18
fn("Amy","");  // Amy,
fn("Amy");     // Amy,17
```

1.4.2 rest参数

ES6新增了Rest 参数用来接收函数的多余参数，组成一个数组，放在形参的最后。

示例：

```
function nums(a,b,...rest){
  console.log(a);
  console.log(b);
  console.log(rest);
}

nums(1,2,3,4,5);        //1,2,Array [3,4,5]
nums(19);    //19, undefined, Array[]
```

1.4.3 箭头函数

使用箭头函数不需要function关键字来创建函数，省略return关键字，**继承当前上下文的 this 关键字**

看下面代码（ES6）

```
(response,message) => { ..... } // 一个参数括号可以省略
```

相当于ES5代码

```
function(response,message){ ..... }
```

基本用法：

```
var f = v => v;
//等价于
var f = function(v){
 return v;
}
f(1);  //1
```

当箭头函数要返回对象的时候，为了区分于代码块，要用 **()** 将对象包裹起来

```
// 报错
var f = (id,name) => {id: id, name: name};
f(6,2);  // SyntaxError: Unexpected token :

// 不报错
var f = (id,name) => ({id: id, name: name});
var f = (id,name) => { return {id: id, name: name}};
f(6,2);  // {id: 6, name: 2}
```

1.5 ES6 对象

1.5.1 属性简化写法

ES6允许对象的属性直接写变量，这时候属性名是变量名，属性值是变量值。

```
const age = 12;
const name = "Amy";
const person = {age, name};
person    //{age: 12, name: "Amy"}
//等同于
const person = {age: age, name: name}
```

ES5我们对于对象都是以键值对的形式书写，是有可能出现键值对重名的。

```
function people(name, age) {
    return {
        name: name,
        age: age
    };
}
```

以上代码可以简写为

```
function people(name, age) {
    return {
        name, age
    };
}
```

1.5.2 方法名简化写法

```
const person = {
  sayHi(){
    console.log("Hi");
  }
}
person.sayHi();  //"Hi"
//等同于
const person = {
  sayHi:function(){
    console.log("Hi");
  }
}
person.sayHi();//"Hi"
```

2.1 Module模块

2.1.1 基本用法

ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单。ES6 模块的设计思想，是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。ES6 模块不是对象，而是通过`export`命令显式指定输出的代码，再通过`import`命令输入。

示例：userInfo.js

```
//userInfo.js作为一个模块封装了用户的信息
var username = "zhangsan";
var password = "123";
```

模块导入导出各种类型的变量，**如字符串，数值，函数，类。**

- 导出的函数声明与类声明必须要有名称（export default 命令另外考虑）。
- 不仅能导出声明还能导出引用（例如函数）。
- export 命令可以出现在模块的任何位置，但必需处于模块顶层。
- import 命令会提升到整个模块的头部，首先执行。

2.1.2 export 命令

模块功能主要由两个命令构成：`export`和`import`。`export`命令用于规定模块的对外接口，`import`命令用于输入其他模块提供的功能。

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。下面是一个 JS 文件，里面使用`export`命令输出变量。

示例：userInfo.js

```
var username = "zhangsan";
var password = "123";
//export输出变量username,password
export {username, password};
//输出函数multiply
export function multiply(x, y) {
  return x * y;
};
```

导出数据可以使用别名：

```
// export.js
const arr = ['html', 'css', 'js']
export { arr as technologies }

// import.js
import { technologies } from './export.js';
```

2.1.3 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。

示例：

```
import {username,password} from './userInfo';

function showInfo() {
  console.log(username + ' ' + password);
}
```

2.1.4 使用模块(Module)引用

在前面操作中，import的变量必须与export输出的变量名称相同，这个规则导致我们必须知道输出的变量名称是什么才能顺利import，这就增加了我们阅读文档的时间，不利于快速上手。export default的出现就是为了让用户不用知道模块内定义的变量就能加载模块，它为模块指定了默认输出。所以优先推荐使用这个方式导出和导入。

注意：export default命令在一个模块中只能使用一次。

export

```
let user = {
    userName : "david",
    password : "123",
    showUser : function () {
        return this.userName + " | " + this.password;
    }
}
export default user;
```

import

```
import user1 from "./user.js";
console.log(user1.showUser());
```

可以看到import后面的user1没有加花括号{}，并且名称和export.js中输出的变量名称不同，这样我们在引用第三方模块时就无需知晓内部export的变量名称是什么。

html页面中需要设置`type = “module”`属性来控制。

```
<script type="module">
  import user from "./user.js";
  console.log(user.userName)
</script>
```

2.1.5 import()

import命令做不到node的require的**动态加载**功能，因此ES2020提案引入import()函数，支持动态加载模块。

```
import(spec) //参数spec代表加载的模块的位置
```

import命令能接受什么参数import()函数就能接受什么参数，区别只是后者是动态加载。

后续开发中用到的import()的应用最常见的就是Vue的路由懒加载了。因为当打包构建应用时，JavaScript包会变得非常大，影响页面加载。这时候我们将不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件。写法如下：

```
// index.js
{
  path: 'home',
  component: () => import('@/views/page/home'),
  name: 'home',
  meta: {
    title: '首页'
  },
}
```

它代表的就是当访问home这个路由时，才加载home组件，其他路由均使用此方法，就能大大减小js包体积，加快页面加载速度。

总结一下：

- import()函数可以用在任何地方；
- import()函数与所加载的模块没有任何静态连接关系；
- import()函数类似于Node的require方法，但它是异步加载。

2.3.6 总结

1. ES6新增export和import两个命令实现模块加载；
2. ES6模块自动采用严格模式，而不管你有没有在模块头部加上'use strict;'；
3. export命令可以输出变量、函数和类；
4. 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，引用的时候就是重命名的那个名称；
5. export命令规定的是对外的接口，因此必须与模块内部的变量、函数或类建议一一对应关系；
6. export命令输出的接口与其对应的值是动态绑定关系，意思是通过此接口可以获取模块内部实时的值；
7. import和export命令只能在模块的顶层，不能在代码块之中（比如在函数或者if语句之中）；
8. import()函数在路由懒加载、条件加载中最能体现价值；
9. 比较一下在Vue中，import各个写法的含义：

```
import '@/style/index.scss';  // 执行所加载的模块，但是不输入任何值

import HelloWorld from "@/components/HelloWorld.vue";  // 引用组件（export default）

import Vue from 'vue';  // 引用Vue模块（export default）

import { uploadOss as alias } from '@/utils/upload';  //引用uploadOss并重命名为alias
（export）

import { queryList, addUser } from '@/api/home.js';  //引用home.js中的变量或函数或类
（export）

import * as alias from 'vue-router';   // 加载整个模块（export），alias.xxx

import _, { each, forEach } from 'lodash';   //引入默认方法和其他接口
```