9天VUE集训营

大家好，我是来自路飞学城的小马哥老师。

山东潍坊人

5年

移动app iOS oc H5

Python

Js

- ES6
- Vue基础(指令)+组件化
- Vue全家桶（vue-router+vuex， 单页应用程序SPA MPA(index.html about.html)
- 数据驱动视图
- axios + element-ui + mock.js (nodejs koa eggjs) + mysql/mongodb
- 项目 （ 麻雀虽小 五脏俱全 ） （ 购物车项目）

ECMAScript 6简介

ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布了。它的目标，是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

推荐阮一峰ES6教程

有自己的技术博客

let和const命名

let基本用法-块级作用域

在es6中可以使用let声明变量，用法类似于var

> ⚠️ let声明的变量，只在 `let` 命令所在的代码块内有效

```
1
2 {
3     let a = 10;
4     var b; = 20;
5 }
6 console.log(a); //a is not defined
7 console.log(b); //20
```

不存在变量提升

`var` 命令会发生 `变量提升` 现象，即变量可以在声明之前使用，值为 `undefined` 。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。

为了纠正这种现象，let命令改变了语法行为，它所声明的变量一定在声明后使用，否则报错

```
//var的情况

console.log(c);//输出undefined
var c = 30;



//let的情况
console.log(c);// 报错ReferenceError
let c = 30;
```

不允许重复声明

let 不允许在相同作用域内，重复声明同一个变量

```
let c = 10;

let c = 30;
console.log(c); //报错

function func(arg) {
  let arg; // 报错
}
func('xxx')
```

暂时性死区

了解的一个名词，说的就是 let 和 const 命令声明变量的特征。

在代码块内，使用 let 命令声明变量之前，该变量都是不可用的。这在语法上，称为 暂时性死区 (temporal dead zone，简称 TDZ)

为什么需要块级作用域?

原因一：内层变量可能会覆盖外层变量

```
var a = 10;
function foo(){
    console.log(a);
    if(1===2){
        var a = 'hello 小马哥';
    }
}
foo(); //undefined
```

原因二：用来计数的循环遍历泄露为全局变量

```
var arr = [];
for(var i = 0; i < 10; i++){

    arr[i] = function(){
        return i;
    }
}
console.log(arr[5]());//
```

变量 `i` 只用来控制循环，但是循环结束后，它并没有消失，用于变量提升，泄露成了全局变量。

解决循环计数问题

```
//解决方式一：使用闭包
var arr = []
for(var i = 0; i < 10; i++){

    arr[i] = (function(n){

        return function(){
            return n;
        }
    })(i)
```

```
}
//解决方式二：使用let声明i

var arr = []
for(let i = 0; i < 10; i++){
    arr[i] = function () {
        return i;
    }
}
```

const基本用法-声明只读的常量

这意味着，`const`一旦声明变量，就必须立即初始化，不能留到以后赋值。对于`const`来说，只声明不赋值，就会报错。

```

const a = 10;
a = 20;//报错

const b; //报错
```

与`let`命令相同点

- 块级作用域
- 暂时性死区
- 不可重复声明

`let`和`const`使用建议

> 在默认情况下用**const**,而只有你在知道变量值需要被修改的情况下使用**let**

大家完全放弃掉var

模板字符串

传统的 JavaScript 语言，输出模板通常是这样写的

```
const oBox = document.querySelector('.box');
// 模板字符串
let id = 1,name = '小马哥';

let htmlTel = "<ul><li><p>id:" + id + "</p><p>name:" + name + "</p></li></ul>";
oBox.innerHTML = htmlTel;
```

上面的这种写法相当繁琐不方便,ES6引入了模板字符串解决这个问题

tab键上面的反引号 ``

如何插入变量 ${变量名}

```
let htmlTel = `<ul>
    <li>
    <p>id:${id}</p>
    <p>name:${name}</p>
    </li>
</ul>`;
```

解构赋值

解构赋值是对赋值运算符的一种扩展。它通常针对数组和对象进行操作。

```
const (a,b) = ();
```

> 优点：代码书写简洁且易读性高

数组解构

在以前，为变量赋值，只能直接指定值

```
let a = 1;
let b = 2;
let c = 3;
```

ES6允许我们这样写:

```
let (a,b,c) = (1,2,3);
```

> 如果解构不成功，变量的值就等于 `undefined`

```
let (foo) = ();
let (bar, foo) = (1);
```

`foo` 的值等于 `undefined`

对象解构

解构可以用于对象

```
let node = {
    type:'identifier',
    name:'foo'
}

let { type, name } = node;
console.log(type,name)//identifier foo
```

对象的解构赋值时，可以对属性忽略和使用剩余运算符

```
let obj = {
    a:{
        name:'张三'
    },
    b:(),
    c:'hello world'
}
//可忽略 忽略b,c属性
let {a} = obj;
//剩余运算符 使用此法将其它属性展开到一个对象中存储
let {a,...res} = obj;
console.log(a,res);
```

默认值

```
let {a,b = 10} = {a:20};
```

函数参数解构赋值

直接看例子

```
function add((x, y)){
  return x + y;
}

add((1, 2)); // 3
```

使用默认值

```
function addCart(n,num=1){
    return n+num;
}
addCart(10,3);//10
addCart(10,20); //30
```

用途

- 交换变量的值

```
let x = 1;
let y = 2;
(x,y) = (y,x);
```

上面代码交换变量 `x` 和 `y` 的值，这样的写法不仅简洁，而且易读，语义非常清晰。

- 从函数返回多个值

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

```
// 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();
```

- 函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来。

```
// 参数是一组有次序的值
function f([x, y, z]) { ... }
f([1, 2, 3]);

// 参数是一组无次序的值
function f({x, y, z}) { ... }
f({z: 3, y: 2, x: 1});
```

- 提取JSON数据

  解构赋值对提取 JSON 对象中的数据，尤其有用

```
let jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
};

let { id, status, data:number } = jsonData;
//对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者
console.log(id, status, number);
// 42, "OK", [867, 5309]
```

- 函数参数的默认值
- 输入模块的指定方法

  加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。

```
const {ajax} = require('xxx')
ajax()
```

函数的扩展

带参数默认值的函数

ES6之前，不能直接为函数的参数指定默认值，只能采用变通的方法

```
function log(x,y){
    y = y || 'world';
    console.log(x,y);
}
log('hello');//hello world
log('hello','china') //hello china
log('hello','')//hello world
```

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

```
function log(x, y = 'World') {
  console.log(x, y);
}

log('Hello') // Hello World
log('Hello', 'China') // Hello China
log('Hello', '') // Hello
```

> ES6 的写法还有两个好处：首先，阅读代码的人，可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；其次，有利于将来的代码优化，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

默认的表达式可以是一个函数

```
function getVal(val) {
    return val + 5;
}
function add2(a, b = getVal(5)) {
    return a + b;
}
console.log(add2(10));
```

小练习

请问下面两种写法有什么区别?

```
// 写法一
function m1({x = 0, y = 0} = {}) {
  return (x, y);
}

// 写法二
function m2({x, y} = { x: 0, y: 0 }) {
  return (x, y);
}
```

上面两种写法都对函数的参数设定了默认值，区别是写法一函数参数的默认值是空对象，但是设置了对象解构赋值的默认值；写法二函数参数的默认值是一个有具体属性的对象，但是没有设置对象解构赋值的默认值。

```
// 函数没有参数的情况
m1() // (0, 0)
m2() // (0, 0)

// x 和 y 都有值的情况
m1({x: 3, y: 8}) // (3, 8)
m2({x: 3, y: 8}) // (3, 8)

```

```
// x 有值，y 无值的情况
m1({x: 3}) // (3, 0)
m2({x: 3}) // (3, undefined)

// x 和 y 都无值的情况
m1({}) // (0, 0);
m2({}) // (undefined, undefined)

m1({z: 3}) // (0, 0)
m2({z: 3}) // (undefined, undefined)
```

rest参数

ES6 引入 rest 参数（形式为`...变量名`），用于获取函数的多余参数，这样就不需要使用`arguments`对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

```
function add(...args) {
 //(2,5,3)
  let sum = 0;

  for (let val of args) {
    sum += val;
  }

  return sum;
}

add(2, 5, 3) // 10
```

上面代码的`add`函数是一个求和函数，利用 rest 参数，可以向该函数传入任意数目的参数。

箭头函数

const fn = (a,b)=>a+b 等价于const fn = function(a+b){return a+b}

ES6允许使用箭头 `=>` 定义函数

```
let f = v=>v;
//等同于
let f = function(v){
    return v;
}
function foo(a,b){return a+b+'10'}===  let foo = (a,b) => a + b+'10'

// 有一个参数
let add = value => value;

// 有两个参数
let add = (value,value2) => value + value2;

let add = (value1,value2)=>{

    return value1 + value2;
}
// 无参数
let fn = () => "hello world";

let doThing = () => {

}
//如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错。
let getId = id => ({id: id,name: 'mjj'}) //注意
let obj = getId(1);
```

箭头函数的作用

- 使表达更加简洁

```
const isEven = n => n % 2 == 0;
const square = n => n * n;
```

- 简化回调函数

```
// 正常函数写法
[1,2,3].map(function (x) {
  return x * x;
});

// 箭头函数写法
[1,2,3].map(x => x * x);
```

使用注意点

- 没有this绑定

```
let PageHandler = {
    id:123,
    init:function(){

 document.addEventListener('click',function(event) {
            this.doSomeThings(event.type);
        },false);
    },
    doSomeThings:function(type){
        console.log(`事件类型:${type},当前id:${this.id}`);
    }
}
PageHandler.init();

//解决this指向问题
let PageHandler = {
    id: 123,
    init: function () {
        // 使用bind来改变内部函数this的指向
```

```
        document.addEventListener('click', function (event) {
            this.doSomeThings(event.type);
        }.bind(this), false);
    },
    doSomeThings: function (type) {
        console.log(`事件类型:${type},当前id:${this.id}`);
    }
}
PageHandler.init();

let PageHandler = {
    id: 123,
    init: function () {
        // 箭头函数没有this的指向，箭头函数内部的this值只能通过查找作用域链来确定
        // 如果箭头函数被一个非箭头函数所包括，那么this的值与该函数的所属对象相等，否则 则是全局的window对象
        document.addEventListener('click', (event) => {
            console.log(this);
            this.doSomeThings(event.type);
        }, false);
    },
    doSomeThings: function (type) {
        console.log(`事件类型:${type},当前id:${this.id}`);
    }
}
PageHandler.init();
```

- 箭头函数中没有arguments对象

```
let getVal = (a,b) => {
    console.log(arguments);
    return a + b;
}
console.log(getVal(1,2)); //arguments is not defined
```

- 箭头函数不能使用new关键字来实例化对象

```
let Person = ()=>{}
let p1 = new Person();// Person is not a constructor
```

对象的扩展

属性的简洁表示法

```
const name = '张三';
const age = 19;
const person = {
    name, //等同于name:name
    age,
    // 方法也可以简写
    sayName() {
        console.log(this.name);
    }
}
person.sayName();
```

这种写法用于函数的返回值，将会非常方便。

```
function getPoint() {
  const x = 1;
  const y = 10;
  return {x, y};
}

getPoint()
// {x:1, y:10}
```

对象扩展运算符

```
const (a, ...b) = (1, 2, 3);
a // 1
b // (2, 3)
```

解构赋值

对象的解构赋值用于从一个对象取值，相当于将目标对象自身的所有可遍历的（enumerable）、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x // 1
y // 2
z // { a: 3, b: 4 }
```

> 解构赋值必须是最后一个参数，否则会报错
>
> ```
> let { ...x, y, z } = obj; // 句法错误
> let { x, ...y, ...z } = obj; // 句法错误
> ```

扩展运算符

对象的扩展运算符（`...`）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

```
let z = { a: 3, b: 4 };
let n = { ...z };
n // { a: 3, b: 4 }
```

扩展运算符可以用于合并两个对象。

```
let ab = { ...a, ...b };
// 等同于
let ab = Object.assign({}, a, b); //对象的合并
```

```
let obj = {
    name:'xxx',
    age；18，
    fav(){

}
}


const name = obj.name;
const age  = obj.age;
const fav = obj.fav;

const {name,age,fav} = obj;
```

Promise 对象

异步编程模块在前端开发中，显得越来越重要。从最开始的XHR到封装后的Ajax都在试图解决异步编程过程中的问题。随着ES6新标准的到来，处理异步数据流又有了新的解决方案。在传统的ajax请求中，当异步请求之间的数据存在依赖关系的时候，就可能产生不优雅的多层回调，俗称"回调地域"(callback hell)，这却让人望而生畏，Promise的出现让我们告别回调地域，写出更优雅的异步代码。

套娃

pending=>fulfilled

pending=>rejected

回调地狱带来的负面作用有以下几点：

- 代码臃肿。
- 可读性差。
- 耦合度过高，可维护性差。
- 代码复用性差。
- 容易滋生 bug。
- 只能在回调里处理异常。

> 在实践过程中，却发现Promise并不完美，Async/Await是近年来JavaScript添加的最革命性的的特性之一，**Async/Await**提供了一种使得异步代码看起来像同步代码的替代方法。接下来我们介绍这两种处理异步编程的方案。

什么是Promise

> Promise 是异步编程的一种解决方案：
>
> 从语法上讲，Promise是一个对象，通过它可以获取异步操作的消息；
>
> 从本意上讲，它是承诺，承诺它过一段时间会给你一个结果。
>
> promise 有三种状态： **pending(等待态)**， **fulfilled(成功态)**，**rejected(失败态)**；
>
> 状态一旦改变，就不会再变。
>
> 创造promise实例后，它会立即执行。

看段习以为常的代码：

```
// Promise是一个构造函数，自己身上有all,reject,resolve,race方法，原型上有then、catch等方法
let p = new Promise((resolve,reject)=>{
    // 做一些异步操作
    setTimeout(()=>{
    /*  let res = {
            ok:1,
            data:{
                name:"张三"
            }
        } */
        let res = {
            ok:0,
            error:new Error('有错')
        }
        if(res.ok === 1){
            resolve(res.data);
        }else{
            reject(res.error.message)
        }

    }, 1000)
})

```

Promise的状态和值

`Promise` 对象存在以下三种状态

- Pending(进行中)
- Fulfilled(已成功)
- Rejected(已失败)

> 状态只能由 `Pending` 变为 `Fulfilled` 或由 `Pending` 变为 `Rejected` ，且状态改变之后不会在发生变化，会一直保持这个状态。

`Promise` 的值是指状态改变时传递给回调函数的值

上面例子中的参数为resolve和reject，他们都是函数，用他们可以改变Promise的状态和传入的Promise的值

`resolve` 和 `reject`

- `resolve` ：将Promise对象的状态从 `Pending(进行中)` 变为 `Fulfilled(已成功)`
- `reject` ：将Promise对象的状态从 `Pending(进行中)` 变为 `Rejected(已失败)`
- `resolve` 和 `reject` 都可以传入任意类型的值作为实参，表示 `Promise` 对象成功 `（Fulfilled）` 和失败 `（Rejected）` 的值

then方法

```
p.then((data)=>{
    console.log(data);
    return data.ok;
},(error)=>{
    console.log(error)
}).then(data=>{
    console.log(data);
})



p.then().catch()
```

promise的then方法返回一个promise对象，所以可以继续链式调用

上述代码我们可以继续改造，因为上述代码不能传参

```
function timeout(ms) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('hello world')
        }, ms);
    })
}
timeout(1000).then((value) => {
    console.log(value);
})
```

then方法的规则

- then 方法下一次的输入需要上一次的输出
- 如果一个promise执行完后 返回的还是一个promise，会把这个promise 的执行结果，传递给下一次 then 中
- 如果 then 中返回的不是Promise对象而是一个普通值，则会将这个结果作为下次then的成功的结果
- 如果当前 then 中失败了 会走下一个 then 的失败
- 如果返回的是undefined 不管当前是成功还是失败 都会走下一次的成功
- catch是错误没有处理的情况下才会走
- then 中不写方法则值会穿透，传入下一个 then 中

Promise封装XHR对象

```
const getJSON = function (url) {
    return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        xhr.open('GET', url);
        xhr.onreadystatechange = handler;
        xhr.responseType = 'json';
        xhr.setRequestHeader('Accept', 'application/json');
        xhr.send();
        function handler() {
            console.log(this.readyState);
```

```
            if (this.readyState !== 4) {
                return;
            }
            if (this.status === 200) {
                resolve(this.response);
            } else {
                reject(new Error(this.statusText));
            }
        }
    })
}
getJSON('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976')
    .then((res) => {
    console.log(res);

}, function (error) {
    console.error(error);
})

//then方法的链式调用
getJSON('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976')
    .then((res)=>{
    return res.HeWeather6;
}).then((HeWeather6)=>{
    console.log(HeWeather6);
})
```

catch方法

`catch(err=>{})` 方法等价于 `then(null,err=>{})`

```
getJSON('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976')
    .then((json) => {
    console.log(json);
}).then(null,err=>{
    console.log(err);
})
//等价于
getJSON('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976')
    .then((json) => {
    console.log(json);
}).catch(err=>{
    console.log(err);
})
```

resolve()

`resolve()` 方法将现有对象转换成Promise对象，该实例的状态为fulfilled

```
let p = Promise.resolve('foo');
//等价于 new Promise(resolve=>resolve('foo'));
p.then((val)=>{
    console.log(val);
})
```

reject()

`reject()` 方法返回一个新的Promise实例，该实例的状态为rejected

```
let p2 = Promise.reject(new Error('出错了'));
//等价于 let p2 = new Promise((resolve,reject)=>reject(new Error('出错了')));
p2.catch(err => {
    console.log(err);
})
p2.then(null,err=>{})

```

all()方法

all()方法提供了并行执行异步操作的能力，并且再所有异步操作执行完后才执行回调

试想一个页面聊天系统，我们需要从两个不同的URL分别获得用户的的个人信息和好友列表，这两个任务是可以并行执行的，用Promise.all实现如下

```
let meInfoPro = new Promise( (resolve, reject)=> {
    setTimeout(resolve, 500, 'P1');
});
let youInfoPro = new Promise( (resolve, reject)=> {
    setTimeout(resolve, 600, 'P2');
});
// 同时执行p1和p2，并在它们都完成后执行then:
Promise.all((meInfoPro, youInfoPro)).then( (results)=> {
    console.log(results); // 获得一个Array: ('P1', 'P2')
});
```

race()方法

有些时候，多个异步任务是为了容错。比如，同时向两个URL读取用户的个人信息，只需要获得先返回的结果即可。这种情况下，用Promise.race()实现：

```
let meInfoPro1 = new Promise( (resolve, reject)=> {
    setTimeout(resolve, 500, 'P1');
});
let meInfoPro2 = new Promise( (resolve, reject)=> {
    setTimeout(resolve, 600, 'P2');
});
Promise.race([meInfoPro1, meInfoPro2]).then((result)=> {
    console.log(result); // P1
});
```

> **Promise.all**接受一个**promise**对象的数组，待全部完成之后，统一执行**success**;
>
> **Promise.race**接受一个包含多个**promise**对象的数组，只要有一个完成，就执行**success**

举个更具体的例子，加深对race()方法的理解

当我们请求某个图片资源，会导致时间过长，给用户反馈

用race给某个异步请求设置超时时间，并且在超时后执行相应的操作

```
function requestImg(imgSrc) {
    return new Promise((resolve, reject) => {
        var img = new Image();
        img.onload = function () {
            resolve(img);
        }
        img.src = imgSrc;
    });
}
//延时函数，用于给请求计时
function timeout() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
```

```
14              reject('图片请求超时');
15          }, 5000);
16      });
17  }
18  Promise.race([requestImg('images/2.png'),
    timeout()]).then((data) => {
19      console.log(data);
20  }).catch((err) => {
21      console.log(err);
22  });
```

async 函数

异步操作是JavaScript编程的麻烦事，很多人认为async函数是异步编程的解决方案

Async/await介绍

- async/await是写异步代码的新方式，优于回调函数和Promise。
- async/await是基于Promise实现的，它不能用于普通的回调函数。
- async/await与Promise一样，是非阻塞的。
- async/await使得异步代码看起来像同步代码，再也没有回调函数。但是改变不了JS单线程、异步的本质。(异步代码同步化)

Async/await的使用规则

- 凡是在前面添加了**async**的函数在执行后都会自动返回一个**Promise**对象

```
async function req() {
    const res = await axios.get('xxxx');
    const r =  await axios.post('bbbb',res.data);
}

let result = req()
console.log(result)  //即便代码里test函数什么都没返回，我们依然打出了Promise对象
```

- **await必须在async函数里使用，不能单独使用**

```
async test() {
    let result = await Promise.resolve('success')
    console.log(result)
}
test()//success
```

- **await后面需要跟Promise对象，不然就没有意义，而且await后面的Promise对象不必写then，因为await的作用之一就是获取后面Promise对象成功状态传递出来的参数。**

```
function fn() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('success')
        })
    })
}

async test() {
    let result = await fn() //因为fn会返回一个Promise对象
    console.log(result)    //这里会打出Promise成功后传递过来的'success'
}

test()
```

Async/Await的用法

- 使用await，函数必须用async标识
- await后面跟的是一个Promise实例

```
function loadImg(src) {
    const promise = new Promise(function (resolve, reject) {
        const img = document.createElement('img')
        img.onload = function () {
            resolve(img)
        }
        img.onerror = function () {
            reject('图片加载失败')
        }
        img.src = src
    })
    return promise
}
const src1 = 'https://hcdn1.luffycity.com/static/frontend/index/banner@2x_1574647618.8112254.png'
const src2 = 'https://hcdn2.luffycity.com/media/frontend/index/%E7%94%BB%E6%9D%BF.png'
const load = async function () {
    const result1 = await loadImg(src1)
    console.log(result1)
    const result2 = await loadImg(src2)
    console.log(result2)
}
load()
```

当函数执行的时候，一旦遇到 **await** 就会先返回，等到触发的异步操作完成，再接着执行函数体内后面的语句。

async/await的错误处理

关于错误处理，如规则三所说，await可以直接获取到后面Promise成功状态传递的参数，但是却捕捉不到失败状态。在这里，我们通过给包裹await的async函数添加then/catch方法来解决，因为根据规则一，async函数本身就会返回一个Promise对象。

```
const load = async function () {
    try{
        const result1 = await loadImg(src1)
        console.log(result1)
        const result2 = await loadImg(src2)
        console.log(result2)
    }catch(err){
        console.log(err);
    }
}
load()
```

为什么Async/Await更好?

Async/Await较Promise有诸多好处，以下介绍其中三种优势：

- 简洁

  使用Async/Await明显节约了不少代码。我们不需要写.then，不需要写匿名函数处理Promise的resolve值，也不需要定义多余的data变量，还避免了嵌套代码。

- 中间值

在前端编程中，我们偶尔会遇到这样一个场景：我们需要发送多个请求，而后面请求的发送总是需要依赖上一个请求返回的数据。对于这个问题，我们既可以用的Promise的链式调用来解决，也可以用async/await来解决，然而后者会更简洁些

```
const makeRequest = () => {
  return promise1()
    .then(value1 => {
      return promise2(value1)
        .then(value2 => {
          return promise3(value1, value2)
        })
    })
}
```

使用async/await的话，代码会变得异常简单和直观

```
const makeRequest = async () => {
  const value1 = await promise1()
  const value2 = await promise2(value1)
  return promise3(value1, value2)
}
```

- 提高可读性

下面示例中，需要获取数据，然后根据返回数据决定是直接返回，还是继续获取更多的数据。

```
const makeRequest = () => {
  return getJSON()
    .then(data => {
      if (data.needsAnotherRequest) {
        return makeAnotherRequest(data)
          .then(moreData => {
```

```
          console.log(moreData)
          return moreData
        })
    } else {
      console.log(data)
      return data
    }
  })
}
```

代码嵌套（6层）可读性较差，它们传达的意思只是需要将最终结果传递到最外层的Promise。使用async/await编写可以大大地提高可读性:

```
const makeRequest = async () => {
  const data = await getJSON()
  if (data.needsAnotherRequest) {
    const moreData = await makeAnotherRequest(data);
    console.log(moreData)
    return moreData
  } else {
    console.log(data)
    return data
  }
}
```

Class的基本用法

简介

JavaScript语言中，生成实例对象的传统方法是通过构造函数

```
function Person(name,age) {
    this.name = name;
    this.age = age;
}
Person.prototype.sayName  = function() {
    return this.sayName;
}
let p = new Person('小马哥',18);
console.log(p);
```

上面这种写法跟传统的面向对象语言（比如 C++ 和 Java）差异很大，很容易让新学习这门语言的程序员感到困惑

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过 `class` 关键字，可以定义类。

基本上，ES6 的 `class` 可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的 `class` 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用 ES6 的 `class` 改写，就是下面这样

```
class Person {
    // 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,如果没有定义,会被默认添加
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    //等同于Person.prototype = function sayName(){}
    sayName(){
        return this.name;
    }
}
console.log(Person===Person.prototype.constructor)
```

> 类的方法内部如果含有 this，它默认指向类的实例

Module 模块化

概述

历史上，JavaScript 一直没有模块（module）体系，无法将一个大程序拆分成互相依赖的小文件，再用简单的方法拼装起来。其他语言都有这项功能，比如 Ruby 的 require、Python 的 import，甚至就连 CSS 都有 @import，但是 JavaScript 在这方面的支持都没有，这对开发大型的、复杂的项目形成了巨大障碍。

在 ES6 之前，社区制定了一些模块加载方案，最主要的有 CommonJS 和 AMD 两种。前者用于服务器，后者用于浏览器。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，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

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。CommonJS 和 AMD 模块，都只能在运行时确定这些东西。比如，CommonJS 模块就是对象，输入时必须查找对象属性。

```
let a = 1;
let b = 2;
setTimeout(()=>{
    console.log(3)
},2000);
console.log(a);

```

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

```
//module/index.js
export const name = 'zhangsan ';
export const age = 18;
export const color = 'red ';
export const sayName = function() {
    console.log(fristName);
}

//也可以这样
const name = 'zhangsan ';
const age = 18;
const color = 'red ';
const sayName = function() {
    console.log(fristName);
}
export {name,age,color,sayName}

```

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。

```
//main.js
import {name,age,color,sayName,fn} from
'./modules/index.js';
```

如果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 * as obj from './modules/index.js';
console.log(obj);
```

export default 命令

使用 `export default` 命令为模块指定默认输出

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

//或者写成
function foo() {
  console.log('foo');
}

export default foo;
```

在其它模块加载该模块时，`import` 命令可以为该匿名函数指定任意名字

```
//import-default.js
import customName from './export-default.js'
customNmae();//foo
```

如果想在一条import语句中，同时输入默认方法和其他接口，可以写成下面这样

```
import customName,{add} from 'export-default.js'
```

对应上面 `export` 语句如下

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

`export default` 也可以用来输出类。

```
// MyClass.js
export default class Person{ ... }

// main.js
import Person from 'MyClass';
let o = new Person();
```