ES6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。

我们在前端开发术语中经常听到ES6,那么ES6（ECMAScript 6）与JavaScript 是什么关系?

简单来说：ECMAScript 和 JavaScript 的关系是，前者是后者的规格，后者是前者的一种实现（另外的 ECMAScript 方言还有 JScript 和 ActionScript）。日常场合，这两个词是可以互换的。

前端相关工具

如果我们要实现前端的工程化，不是仅仅写几个html,css,js文件就够的，你要考虑到浏览器的兼容性问题，有的老浏览器可能不支持ES6标准，所以会通过其他的工具对其进行降级（高版本语法转成低版本语法）如：Babel

Babel 转码器

Babel 是一个广泛使用的 ES6 转码器，可以将 ES6 代码转为 ES5 代码，从而在老版本的浏览器执行。这意味着，你可以用 ES6 的方式编写程序，又不用担心现有环境是否支持。下面是一个例子。

```
// 转码前
input.map(item => item + 1);

// 转码后
input.map(function (item) {
  return item + 1;
});
```

上面的原始代码用了箭头函数，Babel 将其转为普通函数，就能在不支持箭头函数的 JavaScript 环境执行了。

下面的命令在项目目录中，安装 Babel。

```
$ npm install --save-dev @babel/core   #一般项目脚手架工具会自带，所以不需要单独安装了，此处仅供参考
```

对Babel的介绍就到这里，在项目中我们并不直接操作Babel，而是由脚手架工具帮助我们在渲染脚本的时候自动调用。

ES6新特性

接下来介绍ES6中新加入的javascript语法，其中后面课程提到的部分我们会重点提及，没有涉及到的内容由于篇幅所限，大家可以自行网上查找相关资料

let与const

ES6新增了let和const命令用于声明变量和常量。

其中let与var命令的显著区别分别是，作用域，变量提升和重复声明。

作用域：

```
{
  let sq1 = 'hello';
  var sq2 = 'hello sq';
}
sq1 // ReferenceError: a is not defined.
sq2 // sq
```

上面代码在代码块之中，分别用 `let` 和 `var` 声明了两个变量。然后在代码块之外调用这两个变量，结果 `let` 声明的变量报错，`var` 声明的变量返回了正确的值。这表明，`let` 声明的变量只在它所在的代码块有效。

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要在声明后使用，否则报错。

```
// var 的情况
console.log(foo); // 输出undefined
var foo = 2;

// let 的情况
console.log(bar); // 报错ReferenceError
let bar = 2;
```

上面代码中，变量 `foo` 用 `var` 命令声明，会发生变量提升，即脚本开始运行时，变量 `foo` 已经存在了，但是没有值，所以会输出 `undefined`。变量 `bar` 用 `let` 命令声明，不会发生变量提升。这表示在声明它之前，变量 `bar` 是不存在的，这时如果用到它，就会抛出一个错误。

重复声明

`let` 不允许在相同作用域内，重复声明同一个变量。

```
// 报错
function func() {
  let a = 10;
  var a = 1;
}

// 报错
function func() {
  let a = 10;
  let a = 1;
}
```

因此，不能在函数内部重新声明参数。

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

function func(arg) {
  {
    let arg;
  }
}
func() // 不报错 ，arg定义在对象内部，和函数内部是不同的空间
```

const是声明一个只读的常量，一旦声明，值就不可更改。

```
const name='xiaoming'
name // 'xiaoming'
name='xiaofang' //TypeError: Assignment to constant variable.
```

`const`声明的变量不得改变值，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。

```
const foo;
// SyntaxError: Missing initializer in const declaration
```

上面代码表示，对于`const`来说，只声明不赋值，就会报错。

const与let同时具备相同作用域、变量不可提升和不允许重复声明等特性。

const声明本质是报证变量的值不能改动，以上是变量指向的内存地址所保存的数据无法改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指向实际数据的指针，`const`只能保证这个指针是固定的（即总是指向另一个固定的地址），至于它指向的数据结构是不是可变的，就完全不能控制了。

```
const person = {};

// 为 person 添加一个属性，可以成功
person.name = 'xiaoming';
person.prop // xiaoming

// 将 person 指向另一个对象，就会报错
person = {}; // TypeError: "person" is read-only
```

在实际编码中，指定对象用const居多，原因是要确保对象的唯一性。

变量的解构赋值

ES6增加了一个非常便利的特性，解构赋值，他类似于其他编程语言的解包，用于快速给多个变量进行赋值。ES6中可以用于解构赋值的场景有很多种，我们来介绍最常见的几种。

数组的解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

以前，为变量赋值，只能直接指定值。

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

ES6 允许写成下面这样。

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

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

```
let { foo, bar } = { foo: 'aaa', bar: 'bbb' };
foo // "aaa"
bar // "bbb"
```

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

```
let { bar, foo } = { foo: 'aaa', bar: 'bbb' };
foo // "aaa"
bar // "bbb"

let { baz } = { foo: 'aaa', bar: 'bbb' };
baz // undefined
```

上面代码的第一个例子，等号左边的两个变量的次序，与等号右边两个同名属性的次序不一致，但是对取值完全没有影响。第二个例子的变量没有对应的同名属性，导致取不到值，最后等于`undefined`。

如果解构失败，变量的值等于`undefined`。

```
let {foo} = {bar: 'baz'};
foo // undefined
```

上面代码中，等号右边的对象没有`foo`属性，所以变量`foo`取不到值，所以等于`undefined`。

对象的扩展

属性的简写

ES6 允许在大括号里面，直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

```
const foo = 'bar';
const baz = {foo};
baz // {foo: "bar"}

// 等同于
const baz = {foo: foo};
```

简单概括：当key与value名称相同，则触发对象的简写形式

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 }
```

由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。

```
let foo = { ...['a', 'b', 'c'] };
foo
// {0: "a", 1: "b", 2: "c"}
```

如果扩展运算符后面是一个空对象，则没有任何效果。

```
{...{}, a: 1}
// { a: 1 }
```

如果扩展运算符后面不是对象，则会自动将其转为对象。

```
// 等同于 {...Object(1)}
{...1} // {}
```

上面代码中，扩展运算符后面是整数`1`，会自动转为数值的包装对象`Number{1}`。由于该对象没有自身属性，所以返回一个空对象。

下面的例子都是类似的道理。

```
// 等同于 {...Object(true)}
{...true} // {}

// 等同于 {...Object(undefined)}
{...undefined} // {}

// 等同于 {...Object(null)}
{...null} // {}
```

但是，如果扩展运算符后面是字符串，它会自动转成一个类似数组的对象，因此返回的不是空对象。

```
{...'hello'}
// {0: "h", 1: "e", 2: "l", 3: "l", 4: "o"}
```

对象的扩展运算符等同于使用`Object.assign()`方法。

```
let aClone = { ...a };
// 等同于
let aClone = Object.assign({}, a);
```

箭头函数

箭头函数可以看成普通函数的简写形式

ES6 允许使用“箭头”（`=>`）定义函数。

```
var f = v => v;

// 等同于
var f = function (v) {
  return v;
};
```

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

```
var f = () => 5;
// 等同于
var f = function () { return 5 };

var sum = (num1, num2) => num1 + num2;
// 等同于
var sum = function(num1, num2) {
  return num1 + num2;
};
```

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用`return`语句返回（如果不需要返回值则不用return）。

```
var sum = (num1, num2) => { return num1 + num2; }
```

由于大括号被解释为代码块，所以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。

```
// 报错
let getTempItem = id => { id: id, name: "Temp" };

// 不报错
let getTempItem = id => ({ id: id, name: "Temp" });
```

箭头函数有几个使用注意点。

（1）箭头函数没有自己的`this`对象（详见下文）。

（2）不可以当作构造函数，也就是说，不可以对箭头函数使用`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。

最重要的是第一点，没有自己的this对象。

```
  const fun1=()=>{
    console.log(this)
  }
  function fun2(){
    console.log(this)
  }
  const obj= {
    fun1,
    fun2
  }
  obj.fun1()   //Window {window: Window, self: Window, document: document, name:
"", location: Location, …}
  obj.fun2()   //{fun1: ƒ, fun2: ƒ}
```

可以看到 fun1中的this并没有指向obj这个对象，fun2中的this指向的是当前的对象

Proxy(代理)

Proxy的本意为代理，即对目标对象的操作必须经过该代理，我们拿到这个代理就可以拦截用户对目标对象的操作行为，从而实现监控操作，Vue3响应式的底层原理就采用了Proxy，下面我们来看下具体使用方法。

```
  const obj = {}
  const p = new Proxy(obj,{
    get: function (){
      console.log('拦截get操作')
    },
    set: function () {
      console.log('拦截set操作')
    }
  })
  console.log(p)
  p.a              //拦截get操作
  p.a = 'haiwen'   //拦截set操作
```

上面代码对一个空对象架设了一层拦截，重定义了属性的读取（`get`）和设置（`set`）行为。我们在操作代理对象p时，实际上调用的是get和set对应的方法。代码海说明了Proxy 实际上重载（overload）了点运算符，即用自己的定义覆盖了语言的原始定义。

注意上述代码想要发生拦截效果，必须操作代理对象p，如果直接操作obj是没有拦截效果的。

```
obj.a='haiwen'
obj.a            //haiwen
```

上面的代码演示了拦截效果，但是把我们真正想要的功能给搞没了，如果我们既想保持拦截效果，又想保持原有功能需要：

```
  const obj = {}
  const p = new Proxy(obj,{
    get: function (target,key,){
      console.log('拦截get操作')
```

```
    return obj.key
  },
  set: function (target,key,value) {
    console.log('拦截set操作')
    obj.key=value
  }
})
console.log(p)
p.a='haiwen'  //拦截set操作  "haiwen"
p.a           //拦截get操作  "haiwen"
```

上面代码的get多了两个参数，target可以理解为目标对象，key可以理解为要操作的属性名

同理，set中的target，key也是如此，至于value当然就是所赋的值

Proxy的应用场景

如果我们把以上代码的打印语句替换成dom操作，如：

```
<h1 id="demo">hello</h1>
<script>
  const obj = {}
  const p = new Proxy(obj,{
    get: function (target,key,){
      console.log('拦截get操作')
      return obj.key
    },
    set: function (target,key,value) {
      console.log('拦截set操作')
      document.getElementById('demo').innerText=value
      obj.key=value
    }
  })
  console.log(p)

</script>
```


此时操作代理对象，不用刷新浏览器就发现页面元素发生变化

```
p.a='haiwen'
```

这个就是Vue3实现响应式操作的底层原理

Reflect（反射）

Reflect是配合Proxy而推出的另一个新的API，常常和Proxy一起使用，用于动态设置对象属性，比如前面我们设置对象属性采用的是点语法，实际这样做会出现一些意想不到的bug，如key传递的非字符串。因此使用Reflect操作帮助我们很好的处理这些问题。

```
const obj = {}
const p = new Proxy(obj,{
  get: function (target,key,){
    console.log('拦截get操作')
    return Reflect.get(target,key)
  },
  set: function (target,key,value) {
    console.log('拦截set操作')
    document.getElementById('demo').innerText=value
    Reflect.set(target,key,value)
  }
})
console.log(p)
```

Promise对象

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。

基本用法

```
//创建promise实例对象
const promise = new Promise(function(resolve, reject) {
  console.log('创建promise实例')    //先执行
  setTimeout(function () {
    document.getElementById('demo').innerText='haiwen'
  },3000) //三秒后修改文本--异步

  resolve(
    //成功后要干的事情
    ()=>'success'
  )
  reject(
    //失败后要干的事情
    ()=>'failed'
  )

});
promise.then(function (){
  console.log('执行promise')    //等所有同步任务执行完才执行
})

console.log('hi')              //同步任务--按顺序执行
```

模块的导入

ES6之前，js没有自身的模块体系，即无法使用类似import 语句来导入模块，ES6推出了export和import指令分别用于导出和导入。

先看ES6的模块引用模式

A模块 1个js文件可以看成1个模块，

```
//demo1.js
export const day1='2020-8-11'
```

B模块

```
//index.js
import {day1} from './demo1.js'

console.log(day1)
```

测试网页 index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>demo</title>
</head>
<body>
  <script type="module" src="./index.js"></script>
</body>
</html>
```

在服务器环境下打开浏览器（open live server），查看控制台输出 '2020-8-11'

export命令

若想其他模块使用当前模块的属性或方法，需要显示导出模块中的属性或方法

export命令可以输出变量，函数和类

```
export const day1='2020-8-11' //输出变量
export function add(a,b){return a+b }  //输出函数
```

除了这种写法，还可以写成

```
const day1='2020-8-11' //输出变量
function add(a,b){return a+b }  //输出函数

export {day1,add}
```

export输出的变量和函数就对应本来的名字，但是也可以用as别名

```
const day1='2020-8-11' //输出变量
function add(a,b){return a+b }  //输出函数

export {day1 as prop1,add as prop2}
```

但是需要注意，如果单独export变量或方法需要用大括号包起来

```
export day1; //报错
export {day1};  //正确
```

import命令

有了export导出当前模块相应的属性或方法后 其他js模块就可以import加载这个模块，并使用导出的方法。

以下案例为同一个js文件：index.js

```
import {day1,add} from './demo1.js'

console.log(day1)
console.log(add(11,11))
```

同导出一样，导入也可以使用别名

```
import {day1 as prop1,add as prop2} from './demo1.js'

console.log(prop1)
console.log(prop2(11,11))
```

`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。路径必须以`./`或`../`或`../`

注意，`import`命令具有提升效果，会提升到整个模块的头部，首先执行。

```
console.log(prop1)
console.log(prop2(11,11))

import {day1 as prop1,add as prop2} from './demo1.js'
```

上面的代码不会报错，因为`import`的执行早于console.log的调用。这种行为的本质是，`import`命令是编译阶段执行的，在代码运行之前。

export default命令

相比较export导出时需要指定变量或函数名，export default支持直接导出匿名函数，导入的时候指定任意一个名字就可以

demo.js

```
export default function(a,b){return a+b }
```

index.js

```
import test from './demo1.js'

console.log(test(11,33))
```

我们注意到，使用`export default`时，对应的`import`语句不需要使用大括号。

`export default`命令用于指定模块的默认输出。显然，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。所以，import命令后面才不用加大括号，因为只可能唯一对应`export default`命令。

网页发送Ajax请求

使用框架开发页面时，通常不是把页面数据写死，而是通过和服务端数据交互，动态的渲染到页面上面。

那就不得不涉及到在网页上发起http请求的方法了。

Vue是没有提供对应的功能的，所以我们需要自己实现。

现在使用的http请求技术基本指的是Ajax请求（异步请求，同步会阻塞页面渲染造成页面卡顿）

除了使用原生的xmlrequest和ES6的fetch技术，我们还可以用现成的库axios

axios中文文档|axios中文网 | axios (axios-js.com)

xcode插件

主题插件：Luke Dark Theme