一、概念

- ECMAScript（ES）：是一个国际通用性的标准化脚本语言。
- JavaScript（JS）： ECMAScript、 DOM和BOM组成。
- 简单地理解为： ECMAScript是JavaScript的语言规范， JavaScript是 ECMAScript的实现和扩展。

二、let

- 之前在JavaScript中定义变量，我们用 var 关键字，变量的作用域主要是和函数的定义有关，针对于其他块定义来说是没有作用域的，比如if/for等，这在我们开发中往往会引起一些问题

  1. 存在变量提升问题，降低 js 代码的可阅读性(变量会提升，变量值不会提升，所以输出a=undefined)
  2. 没有块级作用域，容易造成变量污染(var 定义的变量 i 在大括号外依然可以输出结果10)

```
<script>
    //使用var 定义的变量存在变量提升问题
   console.log(a);
   var a = 10;
   console.log(a);
   //使用var定义的变量没有{}作用域的限制
   for(var i = 0; i < 10; i++) {

   }
   //在for循环外依旧可以输出i
   console.log(i)
</script>
```

- let定义变量就不同了，它不存在变量提升问题，只有定义之后才能使用此变量，而且它有{ }作用域(在大括号外面输出 i 也会报错is not defined)

```
<script>
   //使用let 定义的变量不存在变量提升问题
   //报错: Cannot access 'a' before initialization
   console.log(a);
   let a = 10;
   console.log(a);
   //使用var定义的变量有{}作用域的限制
   for(let i = 0; i < 10; i++) {

   }
   //在for循环外输出i报错
   console.log(i);//报错: i is not defined
</script>
```

三、const

- 不存在变量提升问题，只有定义之后才能使用此变量
- const 定义的常量，无法被重新赋值

- 当定义常量的时候，必须定义且初始化，否则报语法错误
- const 定义的常量，也有块级作用域
- 注意： 如果const的是一个对象 ， 对象包含的值是可以被修改的 ， 抽象一点，就是对象所指向的地址不能改变 ， 里面的成员是可以变得

四、模板字符串

模板字符串是增强版的字符串，可以当普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量

```
let person = 'xiaoming';
console.log(person + '是个好学生');
// 等价于
console.log(`${person}是个好学生`);
```

五、对象字面量增强写法

ES6中，对对象字面量进行了很多增强。

```
//我们可以使用Object定义对象，但这种方式不常用，一般直接赋值{}来创建对象，这个{}就称为对象的字面量
// var obj = new Object();
//ES5写法
var id = 1;
var name = "刘备"
var obj = {
    id: id,
    name: name,
    show: function () {
        console.log("hello")
    },
    eat: function () {
        console.log("吃东西")
    }
}
//ES6写法
var obj2 = {
    id, //属性增强写法，内部会将我们变量的key作为对象属性的key，变量的value作为对象属性的value
    name,
    show () { //函数增强写法
        console.log("hello")
    },
    eat () {
        console.log("吃东西")
    }
}
```

六、箭头函数

- 通常函数的定义方法

```
var fn1 = function(a, b) {
    return a + b
}

function fn2(a, b) {
    return a + b
}
```

- 使用ES6箭头函数语法定义函数，将原函数的“function”关键字和函数名都删掉，并使用“=>”连接参数列表和函数体

```
var fn1 = (a, b) => {
    return a + b
}

(a, b) => {
    return a + b
}
```

- 当函数参数只有一个，括号可以省略；但是没有参数时，括号不可以省略。

```
// 无参
var fn1 = function() {}
var fn1 = () => {}

// 单个参数
var fn2 = function(a) {}
var fn2 = a => {}

// 多个参数
var fn3 = function(a, b) {}
var fn3 = (a, b) => {}

// 可变参数
var fn4 = function(a, b, ...args) {}
var fn4 = (a, b, ...args) => {}
```

- 箭头函数相当于匿名函数，并且简化了函数定义。箭头函数有两种格式，一种只包含一个表达式，省略掉了{ ... }和return。还有一种可以包含多条语句，这时候就不能省略{ ... }和return

```
() => return 'hello'

(a, b) => a + b

(a) => {
  a = a + 1
  return a

}
```

- 如果返回一个对象，需要特别注意，如果是单表达式要返回自定义对象，不写括号会报错，因为和函数体的{ ... }有语法冲突。

  注意，用小括号包含大括号则是对象的定义，而非函数主体

```
x => {key: x} // 报错
x => ({key: x}) // 正确
```

七、ES6解构赋值和扩展运算符

- 解构赋值
  - 左右两边结构必须一样
  - 右边必须有值
  - 声明和赋值不能分开

```
//把数组的三个值赋值给三个变量
let arr = [1,2,3];
let a = arr[0];
let b = arr[1];
let c = arr[2];
//使用解构赋值，与顺序有关
let [a,b,c] = [1,2,3]
console.log(a + "-" + b + "-" + c);
//对象：要求变量必须与对象属性同名(不使用别名的情况)，，和顺序无关
let {id, name, age} = {id: 1, name: "张三", age: 18}
console.log(id + "-" + name + "-" + age);

//复杂结构
let [one, two, three, four] = [{a:5, b: 10}, [1,2,3], 8, 'hello'];
console.log(one + "、" + two  + "、" + three  + "、" + four)
//还可以赋值内层
let [{a, b}, [x, y, z], three, four] = [{a:5, b: 10}, [1,2,3], 8, 'hello'];
console.log(a + "、" + b  + "、" + x + "、" + y  + "、" + z)
```

- 扩展运算符
  - ...三点运算符
  - 展开数组

```
//存在两个数组，想将两个(或多个)数组合并
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6]
//使用扩展运算符实现，相当于把数组展开成用逗号隔开的值
let arr = [...arr1, 7, 8, 9, ...arr2]
console.log(arr);
```

- 默认参数

```
function show(a, b, c) {
    console.log(a + "、" + b + "、" + c);
}
//调用
show(1, 2, 3)
//如果有一个数组，想要将数组的值传递给show()
let arr = [11, 22, 33];
//调用时使用
show(...arr)
//声明时使用
function demo(a, b, ...args) {
    console.log(args)
}
//调用，可以传递任意个数的参数(可变参数)，最终形成数组
demo(1,2,3,4,5,6)
```

八、模块化编程

- 模块化优点：
  - 减少命名冲突、避免引入时的层层依赖、可以提高执行效率
- `import`和`export`是ES6模块中的两个命令。
- `export`命令用于规定模块的对外接口
  - 在es6中一个文件可以默认为一个模块，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
- `import`命令用于输入其他模块提供的功能
  - import命令接受一对{}，里面指定要从其他模块导入的变量名，大括号里面的变量名，必须与被导入模块对外接口的名称相同
- 假设我们有如下几个文件
  - one.js

```
let name = "张三"
let age = 20
function add(x, y) {
    return x + y
}
```

  - two.js

```
let name = "李四"
let age = 20
function multi(x, y) {
    return x * y
}
```

- 主文件main.js

```
let a = 100
```

- 入口html: index.html

```
//引入三个js文件
<script src="one.js" type="module"></script>
<script src="two.js" type="module"></script>
<script src="main.js" type="module"></script>
```

- 运行index.html，可以看到console控制台报错，因为name变量被我们声明了两次，为了避免这种冲突，我们可以给script添加一个属性type="module"，将每个文件声明为一个模块，模块是独立的
- 但是这样我们在main.js中就无法调用其他js中的方法了，因为每个文件是单独的模块，内部的变量和方法不供外部访问

- 解决：
  - 在one.js中通过export将变量或方法暴露出去

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

  - 在main.js中通过import引入(可以按需引入)

```
//如果引入的多个文件中有重名的变量或方法，可以起别名
import {add} from './one.js'
import {add as add1} from './two.js'
console.log(add(10, 20));
```

- 关于import和export

```
//可以将声明的变量一起导出，导出时可以起别名
//export {name as uname, age, add}
//也可以单个导出
export function add(x, y) {
    return x + y
}
// 导出对象
export const student = {
  name: 'Megan',
  age: 18
}
//我们也可以在导出方法时不给导出的方法起名字，使用export default缺省导出
//注意：一个文件即模块中只能存在一个export default语句，导出一个当前模块的默认对外接口
export default function (args) {
    console.log(args)

}
```

```
//那么我们就需要在导入的时候命名，比如命名为show
import show from './one.js'
show("hello") //调用

//我么可以同时将export default和export导出的内容一起导入
import show, { name, add } from './exportDemo';
```

九、Promise

1、认识Promise

- Promise就是一个类，异步操作容器，里面存在一个异步操作。通过Promise可 以获取该异步操作可能发生的结果，Promise是异步编程的一种解决方案，当有异步操作时，使用Promise对这个异步操作进行封装

2、Promise基本语法

```
//1.使用setTimeout模拟异步操作
// setTimeout(() => {
//     console.log("Hello World!!!!")
// },1000)
// 2.setTimeout是一个异步操作，对于这种异步操作，可以使用Promise进行封装
// 通过new -> 构造函数(参数为函数，执行传入的函数) ，在执行传入的函数时，会传入两个参数
 (resolve,reject) ，
// resolve,reject本身也是函数
new Promise((resolve, reject) => {
    setTimeout(() => {
        //成功的时候调用resolve
        resolve("Hello World!!!!") //异步操作的结果不在这里处理，在这里调用resolve，进入then()处理结
果
        //失败的时候调用reject
        reject("error message") //调用reject，会进入catch()
    },1000)
}).then((data) => {
    console.log(data); //Hello World!!!!
    console.log(data); //Hello World!!!!
}).catch((err) => {
    console.log(err); //error message
})
```

3、Promise三种状态

- 当我们开发中有异步操作时，就可以给异步操作包装一个Promise，异步操作之后会有三种状态
  - pending：等待状态，比如正在进行网络请求，或者定时器没有到时间。
  - fulfilled：满足状态，当我们主动回调了resolve时，就处于该状态，并且会回调.then()
  - rejected：拒绝状态，当我们主动回调了reject时，就处于该状态，并且会回调.catch()

4、Promise链式调用

- 无论是then还是catch都可以返回一个Promise对象。所以，我们的代码其实是可以进行链式调用的：

```
new Promise((resolve, reject) => {
    setTimeout(() => {
        //成功的时候调用resolve
        resolve("Hello World!!!!") //调用resolve，进入then()处理结果
        //失败的时候调用reject
        reject("error message") //调用reject，会进入catch()
    },1000)
}).then((data) => {
    console.log(data); //Hello World!!!!
    console.log(data); //Hello World!!!!
    return new Promise((resolve, reject) => { //在then中继续返回一个Promise对象
        setTimeout(() => {
            resolve("Hello Vuejs")
        },2000)
    })
}).then((data) => {
    console.log(data);
}).catch((err) => {
    console.log(err); //error message
})
```

- 第二种方式：直接通过Promise包装新的数据，将Promise对象返回
  - Promise.resovle()：将数据包装成Promise对象，并且在内部回调resolve()函数
  - Promise.reject()：将数据包装成Promise对象，并且在内部回调reject()函数

```
new Promise((resolve, reject) => {
    setTimeout(() => {
        //成功的时候调用resolve
        resolve("Hello World!!!!") //调用resolve，进入then()处理结果
        //失败的时候调用reject
        reject("error message") //调用reject，会进入catch()
    },1000)
}).then((data) => {
    console.log(data); //Hello World!!!!
    console.log(data); //Hello World!!!!
    /*return new Promise((resolve, reject) => {
            resolve("Hello Vuejs")
        }) 的简写 =>*/
    // return Promise.resolve("Hello Vuejs")
    return Promise.reject("error message")
}).then((data) => {
    console.log(data);
}).catch((err) => {
    console.log(err); //error message
})
```

5、Promise实例提供的方法

1. Promise.all(iterator),其中iterator必须是个可迭代的对象，比如 Array 或者 String，返回一个Promise实例
   传入参数中任意一个promise返回失败，则整体返回失败，返回的错误信息为第一个失败的promise结果

```
Promise.all([
    new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("result1")
        },1000)
    }),
    new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("result2")
        },3000)
    }),
]).then((res) => { //这个res是将两次请求结果封装为数组
    console.log(res)
})
```

十、原型和原型链

1、名称对应

- prototype：原型
- __proto__：原型链(链接点，根据__proto__找原型，由__proto__形成的链条)

2、从属关系

- prototype → 函数的一个属性，实际就是一个对象
- __proto__ → 对象Object的一个属性，也是一个对象
- 对象的__proto__保存着该对象的构造函数的prototype

```
//定义构造函数
function Person(name) {}
console.log(Person.prototype)
//创建实例
let person = new Person();
console.log(person.__proto__)
console.log(Person.prototype===person.__proto__);//true
```

- __proto__是对象Object的一个属性，prototype 也是一个对象，所以它也有一个__proto__属性

```
console.log(Person.prototype.__proto__)
console.log(Person.prototype.__proto__ === Object.prototype) //true，非构造函数实例化出来的对象的__proto__都指向Object.prototype
console.log(Object.prototype.__proto__ ) //null，原型链上最后一个就是指向Object.prototype
```

3、原型与原型链继承

- 以代码解释：

```
function Person(name) {
    this.a = 1;
}
Person.prototype.b = 2;
let person = new Person();
//1.如果我在构造函数中添加一个属性a，那么打印实例，肯定能看到a
//2.Person.prototype是一个对象，添加一个属性b,打印实例，能在person的__proto__的属性中看到b
//3.Object.prototype添加一个属性，
Object.prototype.c = 3;
console.log(person)
```

关系：

```
/**
  * person {
  *     a: 1,
  *     __proto__: Person.prototype = {
  *         b: 2,
  *         __proto__: Object.prototype = {
  *             c: 3,
  *             __proto__: null
  *         }
  *     }
  * }
  */
```

- 总结：原型链就是以一个对象为基准，以__proto__为链接，一直到Object.prototype为止的一条链。为什么叫原型链，实际是以对象原型为节点连起来的一条链

- ```
  console.log(person.a) //a
  console.log(person.b) //b
  console.log(person.c) //c
  //以上输出说明，实例能不能访问这些值，跟是不是构造它的函数所有是没有关系的，
  // 并不是说构造函数构造了这个实例，这个实例就只能继承构造函数
  //person能够访问到b，是因为person的__proto__保存了Person.prototype，而Person.prototype中有b，
  //如果我在自己身上没有找到b这个属性，那么就到Person.prototype身上找，这叫原型继承
  //同样，找c时，自身及Person.prototype身上都没有找到，就到Object.prototype身上找
  //即沿着__proto__找我原型链上的任意一个原型属性，只要找到就用，找不到就沿着链继续找
  ```

4、Function和Object

- 它俩既是函数，又是对象
  ```

```
//Function也有自己的prototype 和 __proto__
console.log(Function.prototype === Function.__proto__) //true,底层规定好了的
//定义对象
var obj = {};
//相当于
var obj = new Object();//这里Object是一个function，一个function是由Function构造而来
console.log(Object.__proto__)
console.log(Object.__proto__ === Function.prototype)//true
//导出一个等式
console.log(Object.__proto__ === Function.__proto__)//true
```

5、判断属性是否存在的方法

- 判断自身是否有某个属性：hasOwnProperty()

```
console.log(person.hasOwnProperty('a')) //true
console.log(person.hasOwnProperty('b')) //false
console.log(person.hasOwnProperty('c')) //false
```

- 判断链上是否有某个属性

```
console.log('a' in person); //true
console.log('b' in person); //true
console.log('c' in person); //true
```

6、constructor

```
//person的__proto__下有一个constructor，等于构造函数
console.log(person.constructor == Person) //true
//这个构造函数可以修改
function Person1() {
    this.d = 5
}
person.constructor = Person1;
console.log(person.constructor) //Person1
```