ES6

作为javascript的标准ECMAScript一直在更新中，简称ES，ES6于2015年6月正式发布，又称ES2015，目前主流浏览器对其规定标准的支持已经达到90%以上，当然，通过babel等转义工具，或者加载外部cdn，浏览器完成对ES6的支持

模块化

ES6的目标是使js（javascirpt）可用于编写复杂的大型应用程序，成为企业级开发语言，而作为大型编程语言的共同特点之一，即在于其具有模块开发的概念，ES6中特加入了导入导出模块语法，不过浏览器对模块化语法的支持不高

```
+ 基本语法

通过 import （导入）       export（导出）
```javascript
export var name = 'yiche';
export var age = 8;
import * as school from './yiche.js';
```

- 重命名

```javascript
//导出时重命名
function say(){
    console.log('say');
}
export {say as say2};
//导入时重命名
import {say2 as say3} from './school.js';
```

- 默认导出

一般的导出再导入时，用户需要知道所要加载的变量名或者函数名，为了尽快上手，不去费力阅读文档，可以使用默认导出（export default），允许你在导入时自己命名导出模块中的变量或者函数名，而且可以不用{}来接收，这也是目前多数流行框架默认的导出方式

```javascript
export default class SliderItem extends Component { // 导出
  constructor(props) {
```
```

```
    super(props);
  }
  render() {
    let { count, item } = this.props;
    let width = 100 / count + '%';
    return (
      <li className="slider-item" style={{width: width}}>
        <img src={item.src} alt={item.alt} />
      </li>
    );
  }
}
//导入，其实可以取别的名字
import SliderItem from './SliderItem/SliderItem';
//再比如
import $ from jQuery
```

*ES6*导入导入导出的本质是地址的引用，*commonJs(exports = module.exports= {})*则操作的某个值的拷贝

类

- 继承 先看看ES5类的声明和继承方式，先要有那么一个父类

```
function Dad (x,y){
    this.x = 100;
    this.y = 200;
    this.sleep = function(){
        console.log('zzzzzz');
    }
}
Dad.prototype.eat = function(food){
    return 'I am eating'+food;
}
//原型链继承
function Child(){
}
Child.prototype = new Dad();
Child.prototype.constructor = Child;//构造函数指向自己而不是默认的指向父类，避免继承紊乱
Child.protoytpe.name = 'Son';
//实例
var son = new Child();
```

ES6的类和继承则是模仿了java的写法，基本一样

```
class Child extends Dad{
    constructor(x,y,name){//直接给一个construor方法作为构造方法
        super(x,y);//父类的构造函数，只有调用super方法才能使用this,因为子类没有自己的this.换句话说，this必须再super方法调用之后使用
        this.name = name;
    }
    eat(){
        return this.name+super.eat();
    }
}
//实例
let son = new Child(son);
```

- get / set

也是在模仿java等语句中的getter和setter方法，其根本就是传参则是set设置该属性的值，不传则为获取get该属性的值

```
class Person {
    constructor(){
        this.hobbies = [];
    }
    set hobby(hobby){
        this.hobbies.push(hobby);
    }
    get hobby(){
        return this.hobbies;
    }
}
let person = new Person();
person.hobby = 'basketball';
person.hobby = 'football';
console.log(person.hobby);
```

- 静态方法static

还有啥好说的呢，还是java的声明方式，在类里面添加静态的方法可以使用static这个关键词，静态方法就是不需要实例化类就能使用的方法

```
class Person {
```

```
    static add(a,b){
        return a+b;
    }
}
console.log(Person.add(1,2));
```

作用域

作用域就是一个变量的作用范围。也就是你声明一个变量以后,这个变量可以在什么场合下使用 以前的JavaScript只有全局作用域，还有一个函数作用域

- var 的问题

var没有块级作用域，定义后在当前闭包中都可以访问，如果变量名重复，就会覆盖前面定义的变量，并且也有可能被其他人更改。

```
var a = 'b';
if (true) {
     var a = "a"; // 期望a是某一个值
 }
console.log(a);
```

- var在for循环标记变量共享，一般在循环中使用的i会被共享，其本质上也是由于没有块级作用域造成的

```
for (var i = 0; i < 3; i++) {
     setTimeout(function () {
         alert(i);//333
     }, 0);
 }
```

- 块级作用域

现在用了let，不仅仅可以通过闭包隔离，增加块级作用域隔离。 块级作用域通俗来讲就是一组大括号定义一个块,使用 let 定义的变量在大括号的外面是访问不到的

```
for (let i = 0; i < 3; i++) {
        console.log("out", i);
        for (let i = 0; i < 2; i++) {
            console.log("in", i);
```

```
        }
}//out 0 in 0 in 1 out 1 in 0 in 1 out 2 in 0 in 1
```

块级作用域中重复定义会报错，并且不存在变量预解释

- 常量 使用const我们可以去声明一个常量，常量一旦赋值就不能再修改了

```
const MY_NAME = 'yiche';
```

解构

解构意思就是分解一个东西的结构,可以用一种类似数组的方式定义N个变量，可以将一个数组中的值按照规则赋值过去。

```
var [name,age] = ['yiche',8];
console.log(name,age);// yiche 8
```

- 嵌套赋值

```
let [x, [y], z] = [1, [2.1, 2.2]];
console.log(x, y, z);// 1 2.1 undefined
let [x, [y,z]] = [1, [2.1, 2.2]];
console.log(x,y,z);//1 2.1 2.2
```

- 省略赋值

```
let [, , x] = [1, 2, 3];
console.log(x);//3
```

- 更加常用的方法，解构对象

```
let obj = {name:'yiche',age:8};
//对象里的name属性的值会交给name这个变量，age的值会交给age这个变量
let {name,age} = obj;
//对象里的name属性的值会交给myname这个变量，age的值会交给myage这个变量
let {name: myname, age: myage} = obj;
console.log(name,age,myname,myage);
```

- 默认值

ES6允许在赋值和传参的时候使用默认值，原理与解构相似

```
let [a = "a", b = "b", c =new Error('C必须指定')] = [1, , 3];
console.log(a, b, c);

function ajax (options) {
    var method = options.method || "get";
    var data = options.data || {};
    //.....
}
function ajax ({method = "get", data}) {
    console.log(arguments);
}
ajax({
    method: "post",
    data: {"name": "yiche"}
});
```

- 模式匹配新语法

模式匹配是基于数据的结构来选择不同行为的手段之一，其方式类似于解构。模式匹配使非常简洁和高度可读的函数式模式成为可能，并且已存在于许多语言之中本提案当前处于stage0阶段，因此不排除会有重大的变更，语法上，使用match，它可以用于对象、数组、字面量，这里就只以对象为例

```
match (obj) {
    { x }: /* 匹配带有属性x的对象 */,
    { x, ... y }: /* 匹配带有属性x的对象，任何额外的属性填充到y中 */,
    { x: [] }: /* 匹配x属性是一个空数组的对象 */,
    { x: 0, y: 0 }: /* 匹配x和y属性值为0的对象 */
}

const matchPoint = point => match (point) {
  { x, y }: [x, y]
}
matchPoint({ x: 5, y: 7 })  // <- [5, 7]
matchPoint({ x: 5, y: 7, z: 9 })   // <- [5, 7]
matchPoint({ x: 3, z: 7 })  // <- Error

const matchPointWithElse = point => match (point) {
  { x, y }: [x, y],
```

```
  else: [0, 0]
}
matchPointWithElse({ x: 3, z: 7 })  // <- [0, 0]

const matchNullPoint = point => match (point) {
  { x: 0, y: 0 }: [x, y]
}
matchNullPoint({ x: 0, y: 0 })  // <- [0, 0]
matchNullPoint({ x: 1, y: 1 })  // <- Error

const isUSD = item => match (item) {
  { options: { currency: 'USD' } }: true,
  else: false
}
isUSD({ value: 19.99, options: { currency: 'USD' } })   // <- true
isUSD({ value: 19.99, options: { currency: 'ARS' } })   // <- false
```

函数

- 默认参数

如上所述，可以给定义的函数接收的参数设置默认的值在执行这个函数的时候，如果不指定函数的参数的值，就会使用参数的这些默认的值

- 展开运算符

```
//强大的把...，放在数组前面可以把一个数组进行展开
let print = function(a,b,c){
    console.log(a,b,c);
}
print([1,2,3]);
print(...[1,2,3]);

// 可以替代apply
let m1 = Math.max.apply(null, [8, 9, 4, 1]);
let m2 = Math.max(...[8, 9, 4, 1]);

// 可以替代concat
var arr1 = [1, 3];
var arr2 = [3, 5];
var arr3 = arr1.concat(arr2);
var arr4 = [...arr1, ...arr2];
console.log(arr3,arr4);

//类数组的转数组
```

```
function max(a,b,c) {
    console.log(Math.max(...arguments));
}
max(1, 3, 4);
```

- 剩余运算符

```
//剩余操作符可以把其余的参数的值都放到一个叫b的数组里面
let rest = function(a,...b){
    console.log(a,b);
}
rest(1,2,3);
```

- 解构参数

```
let destruct = function({name,age}){
    console.log(name,age);
}
destruct({name:'yiche',age:100});
```

- 函数名

ES6 为函数添加了name属性，可查看该函数的name值

```
let a = ()=>{}
console.log(a.name)// 'a'
let b = function c(){}
console.log(b.name)// 'c'
```

- 箭头函数

箭头函数简化了函数的的定义方式，一般以 "=>" 操作符左边为输入的参数，而右边则是进行的操作以及返回的值inputs=>output 箭头函数非常指的展开讲下，现在已经是主流写法 [1,2,3].forEach(val => console.log(val));); 这里展开讲下

数组

- 改

```
```

```
fill(item，start,end),用item来填充数组中的值，方便用于数组的初始化，单一参数时表示全部替换，同slice()类似，可以接受第二个第三个参数，不同的是，后面表示替换从何开始,从何结束。
    [1,1,1].fill(7,1,2)=>[1,7,1]
copyWithin(target,start,end),见名知意，复制数组中的元素[start,end)从target开始复制替换
    [1,2,3,4,5].copyWithin(0,3,4)=>[4, 2, 3, 4, 5]
```

- 查

```
//查看全部
entries()/keys()/values(),用于ES6的数组遍历，均返回遍历器对象，可用for..of循环遍历，区别entries遍历键值对，keys遍历键，values遍历值。
//点查看
find()/findindex(),均为找出第一个符合条件的数组成员，参数为回掉函数，所有数组成员依次执行，找出返回true,区别在于,没有满足函数的数组成员时，find()将返回undefinde，findindex()将返回-1;
    [1,3,4,5,-2].find(function(value,index,arr){
        return value <0;
    })//-2
```

- Array类的两个扩展方法

```
Array.from()，将对象转为真正的数组，类数组何迭代对象均可使用。
    let arrayLike = {'0','a','1':'b'};
    let arr = Array.from(arrayLike);=>['a','b'];

Array.of(),将一组值转换为数组，其主要为弥补Array()构造时，再参数不足2位默认位数组长度的不足，
    Array(3)=>[, , ,]
    Array.of(3)=>[3];
```

对象

- 对象字面量

键值相同，可省略键

```
let name = 'yiche';
let age = 8;
let getName = function(){
```

```
    console.log(this.name);
}
let person = {
    name,
    age,
    getName
}
```

- 比较 is()

相当于绝对比较===，区别有亮点

```
console.log(Object.is(NaN,NaN));
console.log(+0,-0)
```

- Object.assign

把多个对象的属性复制到一个对象中,第一个参数是复制的对象,从第二个参数开始往后,都是复制的源对象

```
let nameObj = {name:'yiche'};
let ageObj = {age:8};
let obj = {};
Object.assign(obj,nameObj,ageObj);
console.log(obj);
```

需要注意的是，assign只是浅复制，这一点要谨慎

Set 和 Map

集合类似于数组，略有不同 Set的成员值是唯一的，没有重复值,可以用来数组去重

```
let foo = (array) => Array.from(new Set(array))
```

Map则可以使用非字符串作为键名，比如你甚至可以用undefined

```
var books = new Map();
books.set('js',{name:'js'});//向map中添加元素
books.set('html',{name:'html'});
```

```
console.log(books.size);//查看集合中的元素
console.log(books.get('js'));//通过key获取值
books.delete('js');//执照key删除元素
console.log(books.has('js'));//判断map中有没有key
books.forEach((value, key) => { //forEach可以迭代map
    console.log( key + ' = ' + value);
});
books.clear();//清空map
```

Promise

异步操作时往往需要回调函数来执行异步成功或者失败后的操作，在需要多个操作的时候，会导致多个回调函数嵌套，导致代码不够直观，就是常说的回调地狱
Promise本意是承诺，在程序中的意思就是承诺我过一段时间后会给你一个结果。什么时候会用到过一段时间？答案是异步操作，异步是指可能比较长时间才有结果的才做，例如网络请求、读取本地文件等

- promise的三种状态

1. Pending Promise对象实例创建时候的初始状态
2. Fulfilled 可以理解为成功的状态
3. Rejected 可以理解为失败的状态 then 方法就是用来指定Promise对象的状态改变时确定执行的操作，resolve 时执行第一个函数（onFulfilled），reject时执行第二个函数（onRejected）