### let

\* 作用:定义变量

\* 注意点:

> \* let定义过的变量不能再定义,var定义过的变量可以重新定义

> \* let定义的变量不会声明提升,var定义的变量会发生声明提升

> \* let定义的变量在let命令所在的代码块内有效,var定义的变量在函数范围内有效

### const

\* 作用:定义常量(一旦定义,值就不能改变)

\* 注意点:

> \* 定义常量的时候必须赋初值

> \* 常量不可以再次赋值

> \* const定义的常量保证的是常量的地址不可以改变

>> \* 对于简单数据来说值就保存在变量指向的那个内存地址,因此不可以再赋值

>> \* 对于复杂数据来说常量指向数据保存的地址,const保证的是这个地址不变,但地址指向的数据是可以修改的

> \* const定义的常量在const命令所在的代码块内有效

> \* const定义的常量不会声明提升

### 结构解析

\* 作用:从数组或者对象中提取值然后对变量进行赋值

\* 数组的结构解析:按照对应位置,把等号右边数组中的值赋给等号左边数组中的变量,数组可以嵌套

> \* 注意点

>> \* 如果解构不成功,变量的值就等于undefined

>> \* 如果等号的右边不是数组,会报错

>> \* 可以在等号右边数组中设置默认值,当等号左边数组中对应位置的值是undefined时默认值才会生效。

\* 对象的结构解析:按照对应属性名,把等号右边对象中的属性的值赋给等号左边对象中的同名属性

> \* 注意点

>> \* 等号左边对象中的属性的值是真正被赋值的变量名,如果属性的值省略,属性的值和属性的名称相同

### 模版字符

语法:`字符 ${javascript表达式}`

### 扩展运算

\* 扩展运算符三个点...

\* 数组的扩展运算就是将一个数组转为用逗号分隔的参数序列

\* 数组的扩展运算的应用

> \* 函数调用时使用把数组转换为参数列表

```javascript

function add(n1,n2){

return n1+n2;

}

let arr = [11,22];

add(...arr);

```

> \* 函数定义时使用把参数列表转换为数组

```javascript

function add(...arr){

let result = 0;

for(let i = 0;i<arr.length;i++){

result += arr[i]

}

return result;

}

add(11,22);

```

> \* 求数组的最大数

```javascript

let arr = [11,22,44,88,77];

//ES5

Math.max.apply(null,arr)

//ES6 用扩展运算

Math.max(...arr)

```

> \* 深拷贝数组

```javascript

let arr1 = [11,22,44,88,77];

//ES5

let arr2 = arr1.concat();

//ES6 用扩展运算

let arr2 = [...arr1];

```

\* 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

\* 对象的扩展运算符的应用

> \* 拷贝对象(浅拷贝)

```javascript

let obj1 = {name:'Tom',age:18};

let obj2 = {...ojb1};

```

> 如果对象中有复杂数据类型,拷贝的是数据的地址

### 遍历器

\* 遍历器(Iterator)是一种接口,实现了该接口的数据结构就可以完成遍历操作

\* 遍历器接口主要供for...of循环来使用,也就是说实现了遍历器接口的数据类型都可以用for...of循环来遍历数据

### Set

\* 类似于数组,但值唯一的数据结构

\* Set对象的size属性返回Set实例的值的总数

\* Set对象的操作方法

> \* add(value):添加某个值,返回 Set 结构本身

> \* delete(value):删除某个值,返回一个布尔值,表示删除是否成功

> \* has(value):返回一个布尔值,表示该值是否为Set的成员

> \* clear():清除所有值,没有返回值

\* Set对象的遍历方法

> \* keys():返回键名的遍历器

> \* values():返回键值的遍历器

> \* entries():返回键值对的遍历器

> \* forEach(function(value,key){}):使用回调函数遍历

\* 注意点:

> \* Set只有值没有键,所以key和value一样

> \* Set的遍历顺序就是插入顺序

### Map

\* 类似于对象,但键的范围不限于字符串,各种类型的值(包括对象)都可以当作键的数据结构

\* Map对象的size属性返回Map实例的键值对的总数

\* Map对象的操作方法

> \* set(key,value):设置键名key对应的键值为value，然后返回整个 Map 结构,如果key已经有值，则键值会被更新，否则就新生成该键。

> \* get(key):读取key对应的键值，如果找不到key，返回undefined。

> \* delete(key):delete方法删除某个键，返回true。如果删除失败，返回false。

> \* has(key):has方法返回一个布尔值，表示某个键是否在当前 Map 对象之中。

> \* clear():清除所有键值,没有返回值

\* Map对象的遍历方法

> \* keys():返回键名的遍历器

> \* values():返回键值的遍历器

> \* entries():返回键值对的遍历器

> \* forEach(function(value,key){}):使用回调函数遍历

\* 注意点:

> \* Map的遍历顺序就是插入顺序

### Symbol

\* Symbol是一种新的数据类型,表示独一无二的值

\* Symbol值通过Symbol函数生成

\* Symbol函数可以接受一个字符串作为参数,表示对Symbol实例的描述

\* Symbol的主要作用是用来表示对象的属性名

> \* Symbol值作为对象属性名时,不能用点运算符,在对象的内部页不可以

### class类

\* class类是ES5面向对象的语法糖

\* 举例

```javascript

class Person{

constructor(name,age){

this.name = name;

this.age = age;

}

show(){

console.log('I am '+this.name+' and I am '+this.age+' years old');

}

showName(){

console.log('I am '+this.name);

}

showAge(){

console.log('I am '+this.age+' years old');

}

static sayHi(){

console.log('Hi...')

}

}

class Student extends Person{

constructor(name,age,major){

super(name,age);

this.major = major;

}

show(){

super.show();

console.log('I am from '+this.major)

}

showMajor(){

console.log('I am from '+this.major)

}

}

```

\* 定义注意点

> \* 用class定义的类也是函数

> \* 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,如果没有显式定义,系统默认添加一个空的constructor方法

> \* constructor方法默认返回实例对象

> \* 在class 中定义的方法实际上是给原型对象上添加方法

> \* 类不存在变量提升

> \* 静态方法前面添加static关键字,调用时用 类名.方法名 的形式调用

\* 继承注意点

> \* 父类的静态方法,可以被子类继承

> \* 通过extends关键字实现继承

> \* 子类必须在constructor方法中调用super方法,该super方法实际上是调用父类的constructo

> \* 子类如果没有定义constructor方法,系统默认添加一个空的constructor方法,并且在这个方法会被默认添加一个空的super方法

> \* 在子类的构造函数中,只有调用super之后,才可以使用this关键字

### 函数参数默认值

\* 直接在参数定义的时候赋值来为为函数的参数设置默认值

```javascript

function show2(x,y="world"){

console.log(x,y);

}

show2("hello");//hello world

```

\* 参数默认值不是传值的,而是每次都重新计算默认值表达式的值

```javascript

let x = 1;

function show(i=x+1){

console.log(i);

}

show();//2

x = 2;

show();//3

```

\* 结构解析和默认参数结合

```javascript

function show2(url,{method="GET",body="name=Tom"} = {}){

console.log(method,body);

}

show2();//GET name=Tom

show2('http://kuazhu.com');//GET name=Tom

```

\* 默认值参数应该是参数的尾部参数,否则不能省略

### rest参数

\* rest参数就是在函数定义时使用扩展运算(...变量名)把参数列表转换为数组

### 箭头函数

\* 箭头函数就是使用"箭头"(=>)来定义函数

\* 箭头函数的基本用法

```javascript

let fn = (x,y)=>{

console.log(x,y);

return x + y;

}

```

\* 如果函数体只有返回值,函数体的花括号和return语句可以省略,简写:

```javascript

let fn = (x,y)=> x + y

```

\* 如果只有一个参数,圆括号可以省略,简写:

```javascript

let fn = x =>{

console.log(x);

return x + 10;

}

```

\* 箭头函数中的this是在定义时确定的(定义时属于谁),而不是在运行时确定(谁调用)的

```javascript

function Timer(id){

this.id = id;

setTimeout(function(){

console.log(this,this.id);//window 99

},100)

}

var id = 99;

let t = new Timer(100);

```

> \* function 定义的函数,this是在运行时确定的,运行时函数属于谁this就是谁,上面例子中的延迟函数会在100毫秒后由window对象来调用,所以里面的this是window对象

> \* function 定义的函数,可以通过call/bind/apply来改变函数的内部this

```javascript

function Timer(id){

this.id = id;

setTimeout(()=>{

console.log(this,this.id);//Timer {id: 100} 100

},100);

}

var id = 99;

let t = new Timer(100);

```

> \* 箭头函数导致this总是指向函数定义生效时所在的对象

> \* 上面例子中延迟函数中匿名函数在定义的时候是属于Timer的实例对象,所以里面的this就是Timer对象

> \* 用let定义的函数不会声明提升

### Promise的概念

\* Promise 是一种异步编程解决方案

\* 将异步操作以同步操作的流程表达出来,避免了层层嵌套的回调函数

\* Promise对象提供统一的接口,使得控制异步操作更加容易

### Promise的特点

\* Promise有三种状态

> \* pending（进行中）

> \* fulfilled（已成功）

> \* rejected（已失败）

\* 只有异步操作的结果,可以决定当前是哪一种状态,任何其他操作都无法改变这个状态

\* 一旦状态改变,就不会再变

\* Promise对象的状态改变，只有两种可能：

>\* 从pending变为fulfilled

>\* 从pending变为rejected

### Promise的使用

\* Promise构造函数接受一个函数作为参数,该函数的两个参数分别是resolve和reject

\* resolve函数在异步操作成功时(从pending变为fulfilled)调用,并将异步操作的结果作为参数传递出去

\* reject在异步操作失败时(从pending变为rejected)调用,并将异步操作报出的错误作为参数传递出去

\* Promise实例生成以后,可以用then方法分别指定resolved状态和rejected状态的回调函数,rejected状态的回调函数可以省略

\* Promise 新建后就会立即执行

### Promise对象的方法

\* then(resolved状态的回调,rejected状态的回调)

> \* then方法返回的是一个新的Promise实例

\* catch(回调)

> \* 如果异步操作抛出错误,状态就会变为rejected,当then中没有指定rejected状态的回调时就会调用catch方法指定的回调函数

> \* 如果运行中抛出错误,也会被catch方法捕获

\* finally(回调)

> \* 不管Promise对象最后状态如何,都会执行的操作

> \* finally方法的回调函数不接受任何参数

### Promise静态方法

\* Promise.resolve()

> \* 如果参数是Promise实例,那么Promise.resolve将不做任何修改、原封不动地返回这个实例

> \* 如果参数是一个具有then方法的对象,resolve方法会将这个对象转为 Promise 对象然后就立即执行该对象的then方法

> \* 如果参数参数不是具有then方法的对象,或根本就不是对象,返回一个新的 Promise 对象,状态为resolved

> \* 不带有任何参数,直接返回一个resolved状态的 Promise 对象

\* Promise.reject()

> \* 返回一个新的Promise实例,该实例的状态为rejected,参数会原封不动地作为reject的理由

\* Promise.all()

> \* 接受一个Promise实例数组作为参数,返回一个新Promise实例

> \* 当数组中的所有Promise实例的状态都变成fulfilled时,Promise实例的状态才会变成fulfilled,此时,返回值组成一个数组传递给新Promise实例的回调函数

> \* 当数组中有一个Promise实例被rejected,新Promise实例的状态就变成rejected,此时第一个被reject的实例的返回值会传递给新Promise实例的回调函数

\* Promise.race()

> \* 接受一个Promise实例数组作为参数,返回执行最快的那个promise对象