二、ES6 :

1. let、const和var的概念与区别

块作用域由 { } 包括，if语句和 for语句里面的{ }也属于块作用域。

Var、let、const关键字的区别：

* 1. var定义的变量，没有块的概念，可以跨块访问, 不能跨函数访问。
  2. let定义的变量，只能在块作用域里访问，不能跨块访问，也不能跨函数访问。 Let声明的变量只能在let命令所在的代码块内有效。
  3. const声明一个只读的常量（引用了一个值），使用时必须初始化(即必须赋值)，且不能修改常量的值，只能在块作用域里访问，若是数组或对象（引用类型），可修改其中属性，但不能对数组对象重新赋值。

预编译四部曲：

① 创建AO对象 activeobject 也就是函数的执行器上下文

② 创建变量和形参的声明，赋值undefined(初始化)

③ 给形参赋值

④ 函数声明（不包括函数表达式）

变量提升：

var存在创建、初始化提升，赋值不提升；

let只会提升创建过程，不会自动初始化赋值undefined，且不能重复声明已存在的变量。

Const如果用来定义一个基本数据类型的变量，那么值不能改变

如果用来定义一个对象类型，则对象中的属性以及方法等内容可以改变，因为const只是保存的引用对象变量的地址。

2. 变量提升与暂时性死区: let、const都会产生死区。

变量提升： 即变量可以在声明之前使用，值为undefined。

暂时性死区： 当前作用域顶部到该变量声明位置中间的部分，都是该变量的死区，在死区中，禁止访问该变量。由此给出结论，let、const声明的变量存在变量提升， 但是由于死区我们无法在声明前访问这个变量，访问会报错。

3. 变量的结构赋值

解构：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。

数组的解构赋值：

本质上，这种写法属于模式匹配，只要等号两边的模式相同，左边的变量就会被赋予对应的值。如果解构不成功，变量的值就等于undefined。另一种情况是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组，这种情况下，解构依然可以成功。

解构不成功的时候，才会有默认值，没有赋值，没有定义。

对象的解构赋值：变量和属性必须同名，才可以解构赋值，如果解构失败，值是 undefined

默认值跟数组解构的默认值是一样的，解构成功就直接赋值，不走默认值，解构不成功，就是没有值，或者值是 undefined 走默认值；

函数参数等号左边是形参，右边是默认值，函数调用不传参，走函数形参的默认值；函数调用传参，默认值就不走了，走实参

1. 箭头函数及其this问题（匿名函数和箭头函数）

5. Symbol概念及其作用

Symbol()函数接受一个可选参数，可以添加一段文本描述即将创建的Symbol，这段描述不可用于属性访问，但是建议每次创建Symbol时都添加一段描述，便于阅读代码和调试Symbol程序。

Symbol的描述被存储在内部的[[Description]]属性中,只有调用Symbol的toString()方法时才可以读取这个属性.在执行console.log的时候隐式的调用了toString()方法。

Symbol()函数会返回symbol类型的值，该类型具有静态属性和静态方法。它的静态属性会暴露几个内建的成员对象；

它的静态方法会暴露全局的symbol注册，且类似于内建对象类，但作为构造函数来说它并不完整，因为它不支持语法："new Symbol()"。

var sym = Symbol("foo");

每个从Symbol()返回的symbol实例都是唯一的。一个symbol值能作为对象属性的标识符；这是该数据类型仅有的目的。

Symbol()每次都会创建一个新的 symbol类型

var s1 = Symbol('foo'); var s2 = Symbol('foo')

console.log(s1===s2); // false

全局共享symbol体系：

let gs1 = Symbol.for('global\_symbol\_1') //注册一个全局Symbol

let gs2 = Symbol.for('global\_symbol\_1') //获取全局Symbol

console.log(gs1 === gs2); // true

这样一个Symbol在所有window环境下都保持唯一了。

Symbols 在 for...in 迭代中不可枚举。另外，Object.getOwnPropertyNames() （不考虑属性的可枚举性一律返回）和Object.keys()（返回所有可枚举属性名） 都不会返回 symbol 对象的属性。但是能使用 Object.getOwnPropertySymbols() （返回包含所有Symbol自有属性的数组）得到它们，还有反射API 也可以得到全部的属性包括symbol，。利用该特性，可以把一些不需要对外操作和访问的属性使用Symbol来定义。更好的设计我们的数据对象，让“对内操作”和“对外选择性输出”变得更加优雅。

应用场景1：使用Symbol来作为对象属性名(key)；

应用场景2：使用Symbol来替代常量；

应用场景3：使用Symbol定义类的私有属性/方法。

6. Set和Map数据结构

Set: 是一组key的集合，不存储value, 而且key不重复，可自动排重

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()：清除所有成员，没有返回值。

Map: 键/值对的集合。具有极快的查找速度。

属性： 构造函数： 指定创建映射的函数。

Prototype原型： 为映射返回对原型的引用。

size： 返回映射中的元素数。

方法： clear： 从映射中移除所有元素。

delete： 从映射中移除指定的元素。

forEach： 对映射中的每个元素执行指定操作。

get： 返回映射中的指定元素。

has： 如果映射包含指定元素，则返回 true。

set： 添加一个新建元素到映射。

toString： 返回映射的字符串表示形式。

valueOf： 返回指定对象的原始值。

forEach是iterable内置的方法，它接收一个函数，每次迭代就自动回调该函数。

array.forEach(function (element, index, array) {

// element: 指向当前元素的值 // index: 指向当前索引

// array: 指向Array对象本身

});

set.forEach(function (element, sameElement, set) {

// element和sameElement都是元素本身

})

map.forEach(function ( value, key, map) { } );

7. Proxy

Proxy 对象用于定义基本操作的自定义行为（如属性查找，赋值，枚举，函数调用等）。

handler：包含陷阱（traps）的占位符对象。

traps：提供属性访问的方法。这类似于操作系统中捕获器的概念。

target：代理虚拟化的对象。它通常用作代理的存储后端。根据目标验证关于对象不可扩展性或不可配置属性的不变量（保持不变的语义）。

语法：let p = new Proxy(target, handler);

target：用Proxy包装的被代理对象（可以是任何类型的对象，包括原生数组，函数，甚至另一个代理）。

handler：一个对象，其属性是当执行一个操作时定义代理的行为的函数。

当外界每次对 p 进行操作时，就会执行 handler 对象上的一些方法。handler 能代理的一些常用的方法如下：

get：读取 set：修改 has：判断对象是否有该属性 construct：构造函数...

使用 Proxy 的好处是：对象只需关注于核心逻辑，一些非核心的逻辑（如：读取或设置对象的某些属性前记录日志；设置对象的某些属性值前，需要验证；某些属性的访问控制等）可以让 Proxy 来做。从而达到关注点分离，降级对象复杂度的目的。

以下代码：当对象中不存在属性名时，缺省返回数为37。使用了 get。

let handler = {

get: function(target, name){

return name in target ? target[name] : 37;

}

};

let p = new Proxy({}, handler);

p.a = 1;

p.b = undefined;

console.log(p.a, p.b); // 1, undefined

console.log('c' in p, p.c); // false, 37

8. Reflect对象

Reflect 是一个内置的对象，它提供拦截 JavaScript 操作的方法。这些方法与处理器对象的方法相同。Reflect不是一个函数对象，因此它是不可构造的。

与大多数全局对象不同，Reflect不是一个构造函数。不能将其与一个new运算符一起使用，或者将Reflect对象作为一个函数来调用。Reflect的所有属性和方法都是静态的（就像Math对象）。

Reflect.get(target, name, receiver)：此方法查找并返回target的name属性，如果没有，则返回undefined。如果第一个参数不是对象，则Reflect.get则会报错。

如果name属性是读取函数(getter)，则读取函数里的this绑定Reflect.get的receiver参数。

Reflect.set(target, name, value, receiver)：此方法设置target对象的name属性等于value。

如果name参数是一个赋值函数，则赋值函数的this绑定Reflect.set的receiver参数。

var myObject = {

foo: 4,

set bar(value) {

return this.foo = value;

},

};

var myReceiverObject = {

foo: 0,

};

Reflect.set(myObject, 'bar', 1, myReceiverObject);

myObject.foo // 4

myReceiverObject.foo // 1

Reflect.has(obj, name):Reflect.has对应 name in obj 里面的in操作，如果第一个参数不是对象，Reflect.has和in都会报错。

var myObject = { foo: 1, };

'foo' in myObject // 旧写法 true

Reflect.has(myObject, 'foo') // 新写法 true

Reflect.deleteProperty(obj, name):此方法等同于delete obj[name]，用于删除对象属性。该方法返回一个布尔值。如果删除成功或删除的属性不存在，则返回true，如果删除失败，删除的属性依然还在，则返回false。

Reflect.construct(target, args)：此方法等同于new target(...args)，这提供了一种不使用new，来调用构造函数的方法。

Reflect.getPrototypeOf(obj)：此方法用于读取对象的\_\_proto\_\_属性，对应Object.getPrototypeOf(obj)方法。它们的区别是，如果参数不是对象，Object.getPrototypeOf会将参数转化为对象，而Reflect.getPrototypeOf会报错。

Reflect.setPrototypeOf(obj, newProto)：此方法是设置对象的\_\_proto\_\_属性，返回第一个参数对象，对应Object.setPrototypeOf(obj, newProto)。如果第一个参数不是对象，Object.setPrototypeOf会返回第一个参数对象，而Reflect.setPrototypeOf会报错。如果第一个参数是undefined或null，则两个都会报错。

Reflect.apply(func, thisArgs, args)：对一个函数进行调用操作，同时可以传入一个数组作为调用参数。和 Function.prototype.apply() 功能类似。

Reflect.defineProperty(target, propertyKey, attributes)：此方法等同于Object.defineProperty，用来为对象定义属性。未来后者会被逐渐废除。如果第一个参数不是对象，就会抛出错误信息。

Reflect.getOwnPropertyDescriptor(target, propertyKey) ：该方法等同于Object.getOwnPropertyDescriptor，用于得到指定属性的描述对象，将来会代替后者。如果第一个参数不是对象Object.getOwnPropertyDescriptor不报错，返回undefined，而Reflect.getOwnPropertyDescriptor会报错，表示参数非法。

Reflect.isExtensible (target) ：该方法等同于Object.isExtensible，返回一个布尔值，表示当前对象是否可扩展。如果参数不是对象，Object.isExtensible会返回false，因为本来就是不可扩展的，而Reflect.isExtensible则报错。

Reflect.preventExtensions(target)：等同于Object.preventExtensions，用于让一个对象变为不可扩展，返回一个布尔值，表示是否操作成功。如果参数不是对象，Object.preventExtensions在ES5下会报错，ES6下会返回传入的值，Reflect.preventExtensions则会报错。

Reflect.ownKeys (target)：Reflect.ownKeys方法用于返回对象的所有属性，等同于Object.getOwnPropertyNames与Object.getOwnPropertySymbols之和。

作用：

1. 将object对象一些内部的方法，放到Reflect对象上。也就是说，从Reflect对象上可以拿到语言内部的方法。说明现阶段这些方法存在于object和Reflect对象上，未来只会存在于Reflect对象上。
2. object操作对象时出现报错返回false，而reflect会返回false。
3. 让操作对象的编程变为函数式编程
4. 保持和proxy对象的方法一一对应：Reflect对象的方法与Proxy对象的方法一一对应，只要是Proxy对象的方法，就能在Reflect对象上找到对应的方法。

9. Promise（手撕Promise A+规范、Promise.all、Promise相关API和方法）

10. Iterator和for...of（Iterator遍历器的实现）  
11. 循环语法比较及使用场景（for、forEach、for...in、for...of）

12. Generator及其异步方面的应用

13. async函数

async函数对 Generator 函数的改进，体现在以下四点：

（1）内置执行器 （2）更好的语义 （3）更广的适用性 （4）返回值是 Promise

Async函数是一个异步操作函数，本质上，Async函数是Generator函数的语法糖。async函数就是将 Generator 函数的星号（\*）替换成async，将yield替换成await，仅此而已。

async function test(){

const a = await fn1();

const b = await fn2(a);

}

test();

async函数不需要像 Generator 函数，需要调用next方法才能执行，其调用方式与普通函数调用方式一致。

async表示函数里有异步操作，await表示紧跟在后面的表达式需要等待结果

async函数的返回值是 Promise.

async 注意点：

（1）await 命令后面的 Promise对象结果可能是 rejected，因此需要对此进行捕获，以免错误丢失。

（2）多个 await 的操作，如果互相无逻辑关系，可将多个 await 并发处理：Promise.all

（3）await 只能写在 async 函数中

（4）async 函数可以保留运行堆栈

14. 几种异步方式的比较（回调、setTimeout、Promise、Generator、async）

一、回调函数：

function f1(callback){

　　setTimeout(function () {

　　　　// f1的任务代码

　　 callback();

}, 1000);

}

回调函数的优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度耦合（Coupling），流程会很混乱，而且每个任务只能指定一个回调函数。

二、事件监听：

采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

f1.on('done', f2);

function f1(){

setTimeout(function () {

　　　　// f1的任务代码

　　 f1.trigger('done');

}, 1000);

}

优点：容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以’去耦合’，有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。

三、发布/订阅模式（观察者模式）：

假设存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"一个信号，其他任务可以向信号中心"订阅"这个信号，从而知道什么时候自己可以开始执行。

jQuery.subscribe("done", f2); //f2向"信号中心"jQuery订阅"done"信号

function f1(){

setTimeout(function () {

// f1的任务代码 // f1执行完成后，发布"done"信号，从而引发f2的执行

jQuery.publish("done");

}, 1000);

}

jQuery.unsubscribe("done", f2); // f2完成执行后也可以取消订阅

这种方法的性质与"事件监听"类似，但是明显优于后者。因为我们可以通过查看"消息中心"，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

四、promise：

每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数。

f1().then(f2).then(f3);

优点：回调函数变成了链式写法，程序的流程可以看得很清楚，而且有一整套的配套方法，可以实现许多强大的功能；（前三种没有的好处：）如果一个任务已经完成，再添加回调函数，该回调函数会立即执行。所以不用担心是否错过了某个事件或信号。

五、generator（生成器）函数：

Generator函数，在Javascript中可以创建迭代对象，还有另一个功能：即实现异步。

Generator函数是ES6提供的一种异步编程解决方案，其行为类似于状态机，封装了多个内部状态。

* Generator 函数除了状态机，还是一个遍历器对象生成函数。
* 可暂停函数, yield可暂停，next方法可启动，每次返回的是yield后的表达式结果。
* yield表达式本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield表达式的返回值。

function \* show(){

yield read('1.txt');

yield read('2.txt');

}

六、async / await函数：

实现原理就是就是将 Generator 函数和自动执行器，包装在一个函数里。

async/await本质上还是基于Generator函数，可以说是Generator函数的语法糖，async就相当于之前写的run函数(执行Generator函数的函数),而await就相当于yield，只不过await表达式后面只能跟着Promise对象，如果不是Promise对象的话，会通过Promise.resolve方法使之变成Promise对象。async修饰function,其返回一个Promise对象。await必须放在async修饰的函数里面，就相当于yield只能放在Generator生成器函数里一样。

async function readByAsync(){

try{

let a1 = await read('1.txt');

let a2 = await read('2.txt');

}catch(e){ //TODO handle the exception }

}

特点：

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，这正是它的魔力所在。

Promise 对象、Generator 函数、async 函数比较：

（1）Promise 的 API（then、catch等等）太多，使得实际逻辑展示不清晰

（2）Generator 函数的执行必须依赖执行器

（3）async 函数实现简介，语句清晰

15. class基本语法及继承:

类和模块的内部，默认就是严格模式，所以不需要使用use strict指定运行模式。

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

toString() {

return '(' + this.x + ', ' + this.y + ')';

}

}

var point = new Point（）

Point.prototype.constructor === Point // true

类的数据类型就是函数，类本身就指向构造函数。类的所有方法都定义在类的prototype属性上面。在类的实例上面调用方法，其实就是调用原型上的方法。实例的属性除非显式定义在其本身（即定义在this对象上），否则都是定义在原型上（即定义在class上）。

类的内部所有定义的方法，都是不可枚举的。

类的方法内部如果含有this，它默认指向类的实例。

注意⚠：

1、类不存在变量提升

2、类必须使用new调用，否则会报错。而构造函数可以用new或者不使用new。

加new执行构造函数时：函数内部的this指向当前实例对象

不加new执行构造函数时：非严格模式下函数内部的this指向window对象，严格模式为undefined。返回值是构造函数的执行结果；

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，一个空的constructor方法会被默认添加。

super关键字：必须使用继承才会用到super关键字

①super作为函数调用时，代表父类的构造函数。

②super作为对象时，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

如果子类调用constructor，那么子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。

Class A{

constructor(name) {

this.name = name;

}

}

Class B extends A{

constructor(){

super();

}

}

16. 模块加载方案比较（CommonJS和ES6的Module）  
17. ES6模块加载与CommonJS加载的原理

callee和caller是arguments对象的两个内部属性。

callee返回正在执行的函数本身的引用。callee是arguments的一个属性，这个属性是一个指针，指向拥有这个arguments对象的函数。可利用该属性完成递归。可以降低耦合性： return num\*factorial(num-1);

caller；这个属性中保存着调用当前函数的函数引用，如果是全局作用于中调用当前的函数就返回null。

arguments.callee在严格模式下会导致错误；不能为caller赋值，否则报错；

18.项目难点：图片上传

<input type="file" id="imgUp" onchange="preImg(this.id,'imgPre');" multiple="multiple" accept="image/\*" />

accept:表示控件允许上传的文件格式 multiple表示可以上传多个文件

获得选中图片的url有两种方式：

1、FileReader：

用来把文件读入内存，并且读取文件中的数据。FileReader接口提供了一个异步API，使用该API可以在浏览器主线程中异步访问文件系统，读取文件中的数据。

var reads = new FileReader(); reads.readAsDataURL( element.files[0] );

reads.onload = function (e) { // onload表示成功读取

document.getElementById('img').src = this.result; //图片预览显示

};

2、createObjectURL：

URL.createObjectURL() 静态方法会创建一个 DOMString，其中包含一个表示参数中给出的对象的URL。这个 URL 的生命周期和创建它的窗口中的 document 绑定。这个新的URL 对象表示指定的 File 对象或 Blob 对象。

url = window.URL.createObjectURL(element.files.item(0));

区别：

1. 通过FileReader.readAsDataURL(file)可以获取一段data:base64的字符串
2. FileReader.readAsDataURL是异步执行（过一段时间）
3. FileReader.readAsDataURL则返回包含很多字符的base64，并会比blob url消耗更多内存，但是在不用的时候会自动从内存中清除（通过垃圾回收机制）
4. 通过URL.createObjectURL(blob)可以获取当前文件的一个内存URL
5. createObjectURL是同步执行（立即的）
6. createObjectURL返回一段带hash的url，并且一直存储在内存中，直到document触发了unload事件（例如：document close）或者执行revokeObjectURL来释放。

使用createObjectURL可以节省性能并更快速，只不过需要在不使用的情况下手动释放内存。如果不太在意设备性能问题，并想获取图片的base64，则推荐使用FileReader.readAsDataURL;

将图片url传给后台数据使用FormData；调用它的 append() 方法来添加字段

var formdata = new FormData();

formdata.append("username", "Groucho");

// 通过append()方法在数据末尾追加key为username值为Groucho的数据

formdata.get("username "); // 获取key为username的第一个值

formdata.getAll("username "); //获取key为username的所有值，返回值为数组

formdata.set("name","laoli"); // 通过set(key, value)来设置或修改数据

formdata.has("username") // 通过has(key)来判断是否存在对应的key值

formdata.delete("name"); //通过delete(key)可以删除数据

//构造一个包含Form表单数据的FormData对象，需要在创建FormData对象时指定表单元素。

var formdata=new FormData(document.getElementById("advForm"));

FormData的作用：1、异步上传文件

2、将form表单元素的name与value进行组合，实现表单数据的序列化，从而减少表单元素的拼接，提高工作效率。

问题：

1、解决input file上传图片无法上传相同的图片 如果是相同图片onChange事件只会触发一次

办法：onChange事件代码最后清除元素的value

document.querySelector('#uploader').value = null 或者 this.value = null;

2、上传多张图片时，需要拼接file数组：

方式1：

$.each(getImgFiles(), function(i, file){

formData.append('files', file);

});

方式2：

$.each(getImgFiles(), function(i, file){

formData.append('files[]', file);

});

3、如果使用jquery封装的ajax提交表单数据则需要注意以下两点：

processData : false, // 告诉jQuery不要去处理发送的数据

contentType : false, // 告诉jQuery不要去设置Content-Type请求头

# 展开运算符（…）

展开运算符（spread operator）允许一个表达式在某处展开。展开运算符在多个参数（用于函数调用）或多个元素（用于数组字面量）或者多个变量（用于解构赋值）的地方可以使用。

展开运算符不能用在对象当中，因为目前展开运算符只能在可遍历对象（iterables）可用。iterables的实现是依靠[Symbol.iterator]函数，而目前只有**Array,Set,String**内置[Symbol.iterator]方法，而Object尚未内置该方法，因此无法使用展开运算符。

## 数组字面量中使用展开运算符

var arr1=['a','b','c'];

var arr2=[...arr1,'d','e']; //['a','b','c','d','e']

var arr2=['d','e'];

arr1.push(...arr2); //['a','b','c','d','e']

## 用于解构赋值

展开运算符在解构赋值中的作用跟之前的作用看上去是相反的，将多个数组项组合成了一个新数组。不过要注意：解构赋值中展开运算符只能用在最后。

let [arg1,arg2,...arg3] = [1, 2, 3, 4];

arg1 //1

arg2 //2

arg3 //['3','4']

## 类数组对象变成数组

var list=document.getElementsByTagName('div');

var arr=[..list];

list是类数组对象，而我们通过使用展开运算符使之变成了数组。

# 模板字符串

用反引号(`)标识，它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

如果在模板字符串中需要使用反引号，则前面需要用反斜杠转义。

var greeting = `\`Yo\` World!`; // `Yo` World!

**模板字符串中嵌入变量，要将变量名写在${}之中。**大括号内可以放入任意的JavaScript表达式，可以进行运算，以及引入对象属性：

var x = 1, y = 2;

`${x} + ${y \* 2} = ${x + y \* 2}`; *// "1 + 4 = 5"*

var obj = {x: 1, y: 2};

`${obj.x + obj.y}` *// "3"*

模板字符串之中还可以调用函数：

function func() {

return 'Hello';

}

`${func()} World`; *// "Hello World"*

如果需要引用模板字符串本身，在需要时执行，可以像下面这样写：

1. *// 写法一*
2. var str = 'return ' + '`Hello ${name}!`';
3. var func = new Function('name', str);
4. func('Amy'); *// "Hello Amy!"*
5. *// 写法二*
6. var str = '(name) => `Hello ${name}!`';
7. var func = eval.call(null, str);
8. func('Amy'); *// "Hello Amy!"*

## 标签模板

标签使您可以用函数解析模板字符串。标签函数的第一个参数包含一个字符串值的数组。其余的参数与表达式相关。最后，你的函数可以返回处理好的的字符串，用于该标签的函数的名称可以被命名为任何名字。

1. var a = 5, b = 10;
2. function tag(strings, ...values) {
3. console.log(strings[0]); *// "Hello "*
4. console.log(strings[1]); *// " world"*
5. console.log(strings[2]); *// ""*
6. console.log(values[0]); *// 15*
7. console.log(values[1]); *// 50*
8. return "Anything";
9. }
10. tag`Hello ${a + b} world ${a \* b}`; *// Anything*

tag函数的第一个参数是一个数组，该数组的成员是模板字符串中那些没有变量替换的部分，也就是说，变量替换只发生在数组的第一个成员与第二个成员之间、第二个成员与第三个成员之间，以此类推。tag函数的其他参数，都是模板字符串各个变量被替换的值。上例中，模板字符串含有两个变量，因此tag会接收到value1和value2两个参数。

## String对象的raw（）方法

String.raw方法用来充当模板字符串的处理函数，返回一个除表达式和变量会被替换，其它都保持原样的字符串。

var str = String.raw`Hi\n${2+3}!`; // "Hi\n5!"

str.length; // 6

str.split('').join(','); // "H,i,\,n,5,!"