创建一个匿名函数并立即执行

(**function** (x) { **return** x \* x }) (3);

定义函数

在JavaScript中，定义函数的方式如下：

**function** abs(x) {

**if** (x >= 0) {

**return** x;

} **else** {

**return** -x;

}

}

上述abs()函数的定义如下：

* function指出这是一个函数定义；
* abs是函数的名称；
* (x)括号内列出函数的参数，多个参数以,分隔；
* { ... }之间的代码是函数体，可以包含若干语句，甚至可以没有任何语句。

请注意，函数体内部的语句在执行时，一旦执行到return时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有return语句，函数执行完毕后也会返回结果，只是结果为undefined。

由于JavaScript的函数也是一个对象，上述定义的abs()函数实际上是一个函数对象，而函数名abs可以视为指向该函数的变量。

因此，第二种定义函数的方式如下：

**var** abs = **function** (x) {

**if** (x >= 0) {

**return** x;

} **else** {

**return** -x;

}

};

在这种方式下，function (x) { ... }是一个匿名函数，它没有函数名。但是，这个匿名函数赋值给了变量abs，所以，通过变量abs就可以调用该函数。

上述两种定义完全等价，注意第二种方式按照完整语法需要在函数体末尾加一个;，表示赋值语句结束。

### arguments

JavaScript还有一个免费赠送的关键字arguments，它只在函数内部起作用，并且永远指向当前函数的调用者传入的所有参数。arguments类似Array但它不是一个Array：

function foo(x) {

console.log('x = ' + x); // 10

for (var i=0; i<arguments.length; i++) {

console.log('arg ' + i + ' = ' + arguments[i]); // 10, 20, 30

}

}

利用arguments，你可以获得调用者传入的所有参数。也就是说，即使函数不定义任何参数，还是可以拿到参数的值：

**function** abs() {

**if** (arguments.length === 0) {

**return** 0;

}

**var** x = arguments[0];

**return** x >= 0 ? x : -x;

}

abs(); *// 0*

abs(10); *// 10*

abs(-9); *// 9*

实际上arguments最常用于判断传入参数的个数。你可能会看到这样的写法：

*// foo(a[, b], c)*

*// 接收2~3个参数，b是可选参数，如果只传2个参数，b默认为null：*

**function** foo(a, b, c) {

**if** (arguments.length === 2) {

*// 实际拿到的参数是a和b，c为undefined*

c = b; *// 把b赋给c*

b = null; *// b变为默认值*

}

*// ...*

}

要把中间的参数b变为“可选”参数，就只能通过arguments判断，然后重新调整参数并赋值。

**apply、call 的区别**

对于 apply、call 二者而言，作用完全一样，只是接受参数的方式不太一样。例如，有一个函数定义如下：

|  |  |
| --- | --- |
| 1  2  3 | var func = function(arg1, arg2) {    }; |

就可以通过如下方式来调用：

|  |  |
| --- | --- |
| 1  2 | func.call(this, arg1, arg2);  func.apply(this, [arg1, arg2]) |

其中 this 是你想指定的上下文，他可以是任何一个 JavaScript 对象(JavaScript 中一切皆对象)，call 需要把参数按顺序传递进去，而 apply 则是把参数放在数组里。

JavaScript 中，某个函数的参数数量是不固定的，因此要说适用条件的话，当你的参数是明确知道数量时用 call 。

而不确定的时候用 apply，然后把参数 push 进数组传递进去。当参数数量不确定时，函数内部也可以通过 arguments 这个伪数组来遍历所有的参数。

由于JavaScript的函数可以嵌套，此时，内部函数可以访问外部函数定义的变量，反过来则不行：

这说明JavaScript的函数在查找变量时从自身函数定义开始，从“内”向“外”查找。如果内部函数定义了与外部函数重名的变量，则内部函数的变量将“屏蔽”外部函数的变量。

### 全局作用域

不在任何函数内定义的变量就具有全局作用域。实际上，JavaScript默认有一个全局对象window，全局作用域的变量实际上被绑定到window的一个属性：

'use strict';

**var** course = 'Learn JavaScript';

alert(course); *// 'Learn JavaScript'*

alert(window.course); *// 'Learn JavaScript'*

因此，直接访问全局变量course和访问window.course是完全一样的。

你可能猜到了，由于函数定义有两种方式，以变量方式var foo = function () {}定义的函数实际上也是一个全局变量，因此，顶层函数的定义也被视为一个全局变量，并绑定到window对象：

这说明JavaScript实际上只有一个全局作用域。任何变量（函数也视为变量），如果没有在当前函数作用域中找到，就会继续往上查找，最后如果在全局作用域中也没有找到，则报ReferenceError错误。这说明JavaScript实际上只有一个全局作用域。任何变量（函数也视为变量），如果没有在当前函数作用域中找到，就会继续往上查找，最后如果在全局作用域中也没有找到，则报ReferenceError错误。窗体顶端这说明JavaScript实际上只有一个全局作用域。任何变量（函数也视为变量），如果没有在当前函数作用域中找到，就会继续往上查找，最后如果在全局作用域中也没有找到，则报ReferenceError错误。

这说明JavaScript实际上只有一个全局作用域。任何变量（函数也视为变量），如果没有在当前函数作用域中找到，就会继续往上查找，最后如果在全局作用域中也没有找到，则报ReferenceError错误。

### 名字空间

全局变量会绑定到window上，不同的JavaScript文件如果使用了相同的全局变量，或者定义了相同名字的顶层函数，都会造成命名冲突，并且很难被发现。

减少冲突的一个方法是把自己的所有变量和函数全部绑定到一个全局变量中。例如：

*// 唯一的全局变量MYAPP:*

**var** MYAPP = {};

*// 其他变量:*

MYAPP.name = 'myapp';

MYAPP.version = 1.0;

*// 其他函数:*

MYAPP.foo = **function** () {

**return** 'foo';

};

把自己的代码全部放入唯一的名字空间MYAPP中，会大大减少全局变量冲突的可能。

许多著名的JavaScript库都是这么干的：jQuery，YUI，underscore等等。

### 局部作用域

由于JavaScript的变量作用域实际上是函数内部，我们在for循环等语句块中是无法定义具有局部作用域的变量的：

'use strict';

**function** foo() {

**for** (**var** i=0; i<100; i++) {

*//*

}

i += 100; *// 仍然可以引用变量i*

}

为了解决块级作用域，ES6引入了新的关键字let，用let替代var可以申明一个块级作用域的变量：

'use strict';

**function** foo() {

**var** sum = 0;

**for** (**let** i=0; i<100; i++) {

sum += i;

}

*// SyntaxError:*

i += 1;

}

### 常量

由于var和let申明的是变量，如果要申明一个常量，在ES6之前是不行的，我们通常用全部大写的变量来表示“这是一个常量，不要修改它的值”：

**var** PI = 3.14;

ES6标准引入了新的关键字const来定义常量，const与let都具有块级作用域：

'use strict';

**const** PI = 3.14;

PI = 3; *// 某些浏览器不报错，但是无效果！*

PI; *// 3.14*

### 解构赋值

从ES6开始，JavaScript引入了解构赋值，可以同时对一组变量进行赋值。

什么是解构赋值？我们先看看传统的做法，如何把一个数组的元素分别赋值给几个变量：

**var** **array** = ['hello', 'JavaScript', 'ES6'];

**var** x = **array**[0];

**var** y = **array**[1];

**var** z = **array**[2];

现在，在ES6中，可以使用解构赋值，直接对多个变量同时赋值：

窗体顶端

'use strict';

// 如果浏览器支持解构赋值就不会报错:

var [x, y, z] = ['hello', 'JavaScript', 'ES6'];

窗体底端

注意，对数组元素进行解构赋值时，多个变量要用[...]括起来。

如果数组本身还有嵌套，也可以通过下面的形式进行解构赋值，注意嵌套层次和位置要保持一致：

**let** [x, [y, z]] = ['hello', ['JavaScript', 'ES6']];

x; *// 'hello'*

y; *// 'JavaScript'*

z; *// 'ES6'*

解构赋值还可以忽略某些元素：

**let** [, , z] = ['hello', 'JavaScript', 'ES6']; *// 忽略前两个元素，只对z赋值第三个元素*

z; *// 'ES6'*

如果需要从一个对象中取出若干属性，也可以使用解构赋值，便于快速获取对象的指定属性：

窗体顶端

'use strict';

var person = {

name: '小明',

age: 20,

gender: 'male',

passport: 'G-12345678',

school: 'No.4 middle school'

};

var {name, age, passport} = person;

窗体底端

对一个对象进行解构赋值时，同样可以直接对嵌套的对象属性进行赋值，只要保证对应的层次是一致的：

**var** person = {

name: '小明',

age: 20,

gender: 'male',

passport: 'G-12345678',

school: 'No.4 middle school',

address: {

city: 'Beijing',

street: 'No.1 Road',

zipcode: '100001'

}

};

**var** {name, address: {city, zip}} = person;

name; *// '小明'*

city; *// 'Beijing'*

zip; *// undefined, 因为属性名是zipcode而不是zip*

*// 注意: address不是变量，而是为了让city和zip获得嵌套的address对象的属性:*

address; *// Uncaught ReferenceError: address is not defined*

使用解构赋值对对象属性进行赋值时，如果对应的属性不存在，变量将被赋值为undefined，这和引用一个不存在的属性获得undefined是一致的。如果要使用的变量名和属性名不一致，可以用下面的语法获取：

**var** person = {

name: '小明',

age: 20,

gender: 'male',

passport: 'G-12345678',

school: 'No.4 middle school'

};

*// 把passport属性赋值给变量id:*

**let** {name, passport:id} = person;

name; *// '小明'*

id; *// 'G-12345678'*

*// 注意: passport不是变量，而是为了让变量id获得passport属性:*

passport; *// Uncaught ReferenceError: passport is not defined*

解构赋值还可以使用默认值，这样就避免了不存在的属性返回undefined的问题：

**var** person = {

name: '小明',

age: 20,

gender: 'male',

passport: 'G-12345678'

};

*// 如果person对象没有single属性，默认赋值为true:*

**var** {name, single=true} = person;

name; *// '小明'*

single; *// true*

有些时候，如果变量已经被声明了，再次赋值的时候，正确的写法也会报语法错误：

*// 声明变量:*

**var** x, y;

*// 解构赋值:*

{x, y} = { name: '小明', x: 100, y: 200};

*// 语法错误: Uncaught SyntaxError: Unexpected token =*

这是因为JavaScript引擎把{开头的语句当作了块处理，于是=不再合法。解决方法是用小括号括起来：

({x, y} = { name: '小明', x: 100, y: 200});

### 使用场景

解构赋值在很多时候可以大大简化代码。例如，交换两个变量x和y的值，可以这么写，不再需要临时变量：

**var** x=1, y=2;

[x, y] = [y, x]

快速获取当前页面的域名和路径：

**var** {hostname:domain, pathname:path} = location;

如果一个函数接收一个对象作为参数，那么，可以使用解构直接把对象的属性绑定到变量中。例如，下面的函数可以快速创建一个Date对象：

**function** buildDate({year, month, day, hour=0, minute=0, second=0}) {

**return** **new** Date(year + '-' + month + '-' + day + ' ' + hour + ':' + minute + ':' + second);

}

它的方便之处在于传入的对象只需要year、month和day这三个属性：

buildDate({ year: 2017, month: 1, day: 1 });

// Sun Jan 01 2017 00:00:00 GMT+0800 (CST)

也可以传入hour、minute和second属性：

buildDate({ year: 2017, month: 1, day: 1, hour: 20, minute: 15 });

// Sun Jan 01 2017 20:15:00 GMT+0800 (CST)

使用解构赋值可以减少代码量，但是，需要在支持ES6解构赋值特性的现代浏览器中才能正常运行。目前支持解构赋值的浏览器包括Chrome，Firefox，Edge等。

方法

在一个对象中绑定函数，称为这个对象的方法。

在JavaScript中，对象的定义是这样的：

**var** xiaoming = {

name: '小明',

birth: 1990

};

但是，如果我们给xiaoming绑定一个函数，就可以做更多的事情。比如，写个age()方法，返回xiaoming的年龄：

**var** xiaoming = {

name: '小明',

birth: 1990,

age: **function** () {

**var** y = **new** Date().getFullYear();

**return** y - **this**.birth;

}

};

xiaoming.age; *// function xiaoming.age()*

xiaoming.age(); *// 今年调用是25,明年调用就变成26了*

绑定到对象上的函数称为方法，和普通函数也没啥区别，但是它在内部使用了一个this关键字，这个东东是什么？

在一个方法内部，this是一个特殊变量，它始终指向当前对象，也就是xiaoming这个变量。所以，this.birth可以拿到xiaoming的birth属性。

让我们拆开写：

**function** getAge() {

**var** y = **new** Date().getFullYear();

**return** y - **this**.birth;

}

**var** xiaoming = {

name: '小明',

birth: 1990,

age: getAge

};

xiaoming.age(); *// 25, 正常结果*

getAge(); *// NaN*

单独调用函数getAge()怎么返回了NaN？请注意，我们已经进入到了JavaScript的一个大坑里。

JavaScript的函数内部如果调用了this，那么这个this到底指向谁？

答案是，视情况而定！

如果以对象的方法形式调用，比如xiaoming.age()，该函数的this指向被调用的对象，也就是xiaoming，这是符合我们预期的。

如果单独调用函数，比如getAge()，此时，该函数的this指向全局对象，也就是window。

如果这么写：

**var** fn = xiaoming.age; *// 先拿到xiaoming的age函数*

fn(); *// NaN*

也是不行的！要保证this指向正确，必须用obj.xxx()的形式调用！

**var** xiaoming = {

name: '小明',

birth: 1990,

age: **function** () {

**function** getAgeFromBirth() {

**var** y = **new** Date().getFullYear();

**return** y - **this**.birth;

}

**return** getAgeFromBirth();

}

};

xiaoming.age(); *// Uncaught TypeError: Cannot read property 'birth' of undefined*

结果又报错了！原因是this指针只在age方法的函数内指向xiaoming，在函数内部定义的函数，this又指向undefined了！（在非strict模式下，它重新指向全局对象window！）

修复的办法也不是没有，我们用一个that变量首先捕获this：

'use strict';

**var** xiaoming = {

name: '小明',

birth: 1990,

age: **function** () {

**var** that = **this**; *// 在方法内部一开始就捕获this*

**function** getAgeFromBirth() {

**var** y = **new** Date().getFullYear();

**return** y - that.birth; *// 用that而不是this*

}

**return** getAgeFromBirth();

}

};

xiaoming.age(); *// 25*

用var that = this;，你就可以放心地在方法内部定义其他函数，而不是把所有语句都堆到一个方法中。