# JavaScript

## JS中的8种数据类型及区别

包括值类型(基本对象类型)和引用类型(复杂对象类型)

**基本类型(值类型)：** Number(数字),String(字符串),Boolean(布尔),Symbol(符号),null(空),undefined(未定义)在内存中占据固定大小，保存在栈内存中

**引用类型(复杂数据类型)：** Object(对象)、Function(函数)。其他还有Array(数组)、Date(日期)、RegExp(正则表达式)、特殊的基本包装类型(String、Number、Boolean) 以及单体内置对象(Global、Math)等 引用类型的值是对象 保存在堆内存中，栈内存存储的是对象的变量标识符以及对象在堆内存中的存储地址。

**使用场景：**

Symbol：使用Symbol来作为对象属性名(key) 利用该特性，把一些不需要对外操作和访问的属性使用Symbol来定义

BigInt：由于在 Number 与 BigInt 之间进行转换会损失精度，因而建议仅在值可能大于253 时使用 BigInt 类型，并且不在两种类型之间进行相互转换。

传送门 ☞# JavaScript 数据类型之 Symbol、BigInt

## JS中的数据类型检测方案

**1.typeof**

```
console.log(typeof 1);              // number
console.log(typeof true);           // boolean
console.log(typeof 'mc');           // string
console.log(typeof Symbol)          // function
console.log(typeof function(){});   // function
console.log(typeof console.log());  // function
console.log(typeof []);             // object
console.log(typeof {});             // object
console.log(typeof null);           // object
console.log(typeof undefined);      // undefined
```

优点：能够快速区分基本数据类型

缺点：不能将Object、Array和Null区分，都返回object

**2.instanceof**

```
console.log(1 instanceof Number);                   // false
console.log(true instanceof Boolean);               // false
console.log('str' instanceof String);               // false
console.log([] instanceof Array);                   // true
```

```
console.log(function(){} instanceof Function);       // true
console.log({} instanceof Object);                   // true
```

优点：能够区分Array、Object和Function，适合用于判断自定义的类实例对象

缺点：Number，Boolean，String基本数据类型不能判断

**3.Object.prototype.toString.call()**

```
var toString = Object.prototype.toString;
console.log(toString.call(1));      //[object Number]
console.log(toString.call(true));   //[object Boolean]
console.log(toString.call('mc'));   //[object String]
console.log(toString.call([]));     //[object Array]
console.log(toString.call({}));     //[object Object]
console.log(toString.call(function(){})); //[object Function]
console.log(toString.call(undefined));  //[object Undefined]
console.log(toString.call(null)); //[object Null]
```

优点：精准判断数据类型

缺点：写法繁琐不容易记，推荐进行封装后使用

**instanceof 的作用**

用于判断一个引用类型是否属于某构造函数；

还可以在继承关系中用来判断一个实例是否属于它的父类型。

**instanceof 和 typeof 的区别：**

- [ ] 、null 都会返回object

为了弥补这一点，instanceof 从原型的角度，来判断某引用属于哪个构造函数，从而判定它的数据类型。

## var && let && const

ES6之前创建变量用的是var,之后创建变量用的是let/const

**三者区别**：

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。
   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。
   const用来定义常量，使用时必须初始化(即必须赋值)，只能在块作用域里访问，且不能修改。
2. var可以先使用，后声明，因为存在变量提升；let必须先声明后使用。
3. var是允许在相同作用域内重复声明同一个变量的，而let与const不允许这一现象。
4. 在全局上下文中，基于let声明的全局变量和全局对象GO（window）没有任何关系；
   var声明的变量会和GO有映射关系；
5. 会产生暂时性死区：

> 暂时性死区是浏览器的bug：检测一个未被声明的变量类型时，不会报错，会返回undefined
> 如：console.log(typeof a) //undefined
> 而：console.log(typeof a)//未声明之前不能使用
> let a

6. let /const/function会把当前所在的大括号(除函数之外)作为一个全新的块级上下文，应用这个机制，在开发项目的时候，遇到循环事件绑定等类似的需求，无需再自己构建闭包来存储，只要基于let的块作用特征即可解决

## 作用域和作用域链

创建函数的时候，已经声明了当前函数的作用域==>当前创建函数所处的上下文。如果是在全局下创建的函数就是`[[scope]]:EC(G)`，函数执行的时候，形成一个全新的私有上下文`EC(FN)`，供字符串代码执行(进栈执行)

定义：简单来说作用域就是变量与函数的可访问范围，由当前环境与上层环境的一系列变量对象组成
1.全局作用域：代码在程序的任何地方都能被访问，window 对象的内置属性都拥有全局作用域。
2.函数作用域：在固定的代码片段才能被访问

作用：作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。

**作用域链参考链接**一般情况下，变量到 创建该变量 的函数的作用域中取值。但是如果在当前作用域中没有查到，就会向上级作用域去查，直到查到全局作用域，这么一个查找过程形成的链条就叫做作用域链。

## 闭包的两大作用：保存/保护

- **闭包的概念**

  函数执行时形成的私有上下文EC(FN)，正常情况下，代码执行完会出栈后释放;但是特殊情况下，如果当前私有上下文中的某个东西被上下文以外的事物占用了，则上下文不会出栈释放，从而形成不销毁的上下文。 函数执行函数执行过程中，会形成一个全新的私有上下文，可能会被释放，可能不会被释放，不论释放与否，他的作用是：

（1）保护：划分一个独立的代码执行区域，在这个区域中有自己私有变量存储的空间，保护自己的私有变量不受外界干扰（操作自己的私有变量和外界没有关系）；

（2）保存：如果当前上下文不被释放【只要上下文中的某个东西被外部占用即可】，则存储的这些私有变量也不会被释放，可以供其下级上下文中调取使用，相当于把一些值保存起来了；

我们把函数执行形成私有上下文，来保护和保存私有变量机制称为闭包。

> 闭包是指有权访问另一个函数作用域中的变量的函数--《JavaScript高级程序设计》

**稍全面的回答**： 在js中变量的作用域属于函数作用域, 在函数执行完后,作用域就会被清理,内存也会随之被回收, 但是由于闭包函数是建立在函数内部的子函数, 由于其可访问上级作用域,即使上级函数执行完, 作用域也不会随之销毁, 这时的子函数(也就是闭包),便拥有了访问上级作用域中变量的权限,即使上级函数执行完后作用域内的值也不会被销毁。

- **闭包的特性**:
  - 1、内部函数可以访问定义他们外部函数的参数和变量。(作用域链的向上查找，把外围的作用域中的变量值存储在内存中而不是在函数调用完毕后销毁)设计私有的方法和变量，避免全局变量的污染。

1.1.闭包是密闭的容器，，类似于set、map容器，存储数据的

1.2.闭包是一个对象，存放数据的格式为 key-value 形式

- 2、函数嵌套函数
- 3、本质是将函数内部和外部连接起来。优点是可以读取函数内部的变量，让这些变量的值始终保存在内存中，不会在函数被调用之后自动清除

- **闭包形成的条件**：

1. 函数的嵌套
2. 内部函数引用外部函数的局部变量，延长外部函数的变量生命周期

- **闭包的用途**：

1. 模仿块级作用域
2. 保护外部函数的变量 能够访问函数定义时所在的词法作用域(阻止其被回收)
3. 封装私有化变量
4. 创建模块

- **闭包应用场景**

闭包的两个场景，闭包的两大作用：保存/保护。 在开发中, 其实我们随处可见闭包的身影, 大部分前端 JavaScript 代码都是"事件驱动"的,即一个事件绑定的回调方法; 发送ajax请求成功|失败的回调;setTimeout 的延时回调;或者一个函数内部返回另一个匿名函数,这些都是闭包的应用。

- **闭包的优点**：延长局部变量的生命周期
- **闭包缺点**：会导致函数的变量一直保存在内存中，过多的闭包可能会导致内存泄漏

## JS 中 this 的情况

1. 普通函数调用：通过函数名()直接调用：this指向全局对象window（注意let定义的变量不是window属性，只有window.xxx定义的才是。即let a ='aaa'; this.a是undefined）
2. 构造函数调用：函数作为构造函数，用new关键字调用时：this指向新new出的对象
3. 对象函数调用：通过对象.函数名()调用的：this指向这个对象
4. 箭头函数调用：箭头函数里面没有 this ，所以永远是上层作用域this（上下文）
5. apply和call调用：函数体内 this 的指向的是 call/apply 方法第一个参数，若为空默认是指向全局对象 window。
6. 函数作为数组的一个元素，通过数组下标调用的：this指向这个数组
7. 函数作为window内置函数的回调函数调用：this指向window（如setInterval setTimeout 等）

## call/apply/bind 的区别

相同：

1、都是用来改变函数的this对象的指向的。
2、第一个参数都是this要指向的对象。
3、都可以利用后续参数传参。

不同：

apply和call传入的参数列表形式不同。apply 接收 arguments，call接收一串参数列表

```
fn.call(obj, 1, 2);
fn.apply(obj, [1, 2]);
```

bind：语法和call一模一样，区别在于立即执行还是等待执行，bind不兼容IE6~8 bind 主要就是将函数绑定到某个对象，bind()会创建一个函数，返回对应函数便于稍后调用；而apply、call则是立即调用。

总结：基于Function.prototype上的 `apply 、 call 和 bind` 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。`apply`接收参数的是数组，`call`接受参数列表，`` bind方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this指向除了使用new `时会被改变，其他情况下都不会改变。若为空默认是指向全局对象window。

参考：☞ call、apply、bind三者的用法和区别

## 箭头函数的特性

1. 箭头函数没有自己的`this`，会捕获其所在的上下文的this值，作为自己的this值
2. 箭头函数没有`constructor`，是匿名函数，不能作为构造函数，不能通过new 调用；
3. 没有`new.target 属性`。在通过new运算符被初始化的函数或构造方法中，new.target返回一个指向构造方法或函数的引用。在普通的函数调用中，new.target 的值是undefined
4. 箭头函数不绑定`Arguments 对象`。取而代之用rest参数...解决。由于 箭头函数没有自己的this指针，通过 call() 或 apply() 方法调用一个函数时，只能传递参数（不能绑定this），他们的第一个参数会被忽略。（这种现象对于bind方法同样成立）
5. 箭头函数通过 call() 或 apply() 方法调用一个函数时，只传入了一个参数，对 this 并没有影响。
6. 箭头函数没有原型属性 Fn.prototype 值为 undefined
7. 箭头函数不能当做Generator函数,不能使用yield关键字

参考：箭头函数与普通函数的区别

## 原型 && 原型链

**原型关系：**

- 每个 class都有显示原型 prototype
- 每个实例都有隐式原型 `__proto__`
- 实例的 `__proto__` 指向对应 class 的 prototype

**原型：**在 JS 中，每当定义一个对象（函数也是对象）时，对象中都会包含一些预定义的属性。其中每个函数对象都有一个`prototype` 属性，这个属性指向函数的原型对象。

原型链：函数的原型链对象constructor默认指向函数本身，原型对象除了有原型属性外，为了实现继承，还有一个原型链指针__proto__,该指针是指向上一层的原型对象，而上一层的原型对象的结构依然类似。因此可以利用__proto__一直指向Object的原型对象上，而Object原型对象用Object.prototype.__ proto__ = null表示原型链顶端。如此形成了js的原型链继承。同时所有的js对象都有Object的基本防范

**特点：**`JavaScript`对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

## new运算符的实现机制

1. 首先创建了一个新的空对象
2. 设置原型，将对象的原型设置为函数的prototype对象。
3. 让函数的this指向这个对象，执行构造函数的代码（为这个新对象添加属性）
4. 判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引用类型的对象。

## EventLoop 事件循环

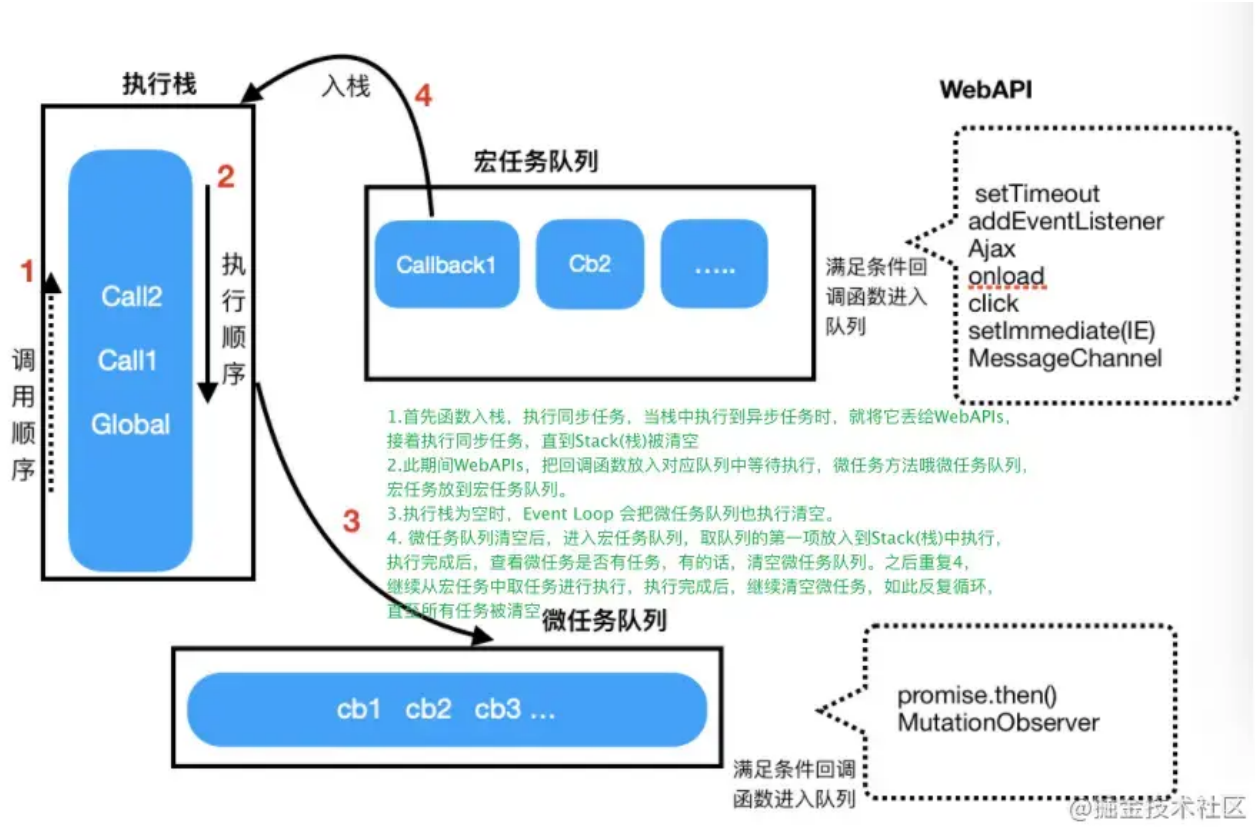
JS是单线程的，为了防止一个函数执行时间过长阻塞后面的代码，所以会先将同步代码压入执行栈中，依次执行，将异步代码推入异步队列，异步队列又分为宏任务队列和微任务队列，因为宏任务队列的执行时间较长，所以微任务队列要优先于宏任务队列。微任务队列的代表就是，Promise.then，MutationObserver，宏任务的话就是setImmediate setTimeout setInterval

JS运行的环境。一般为浏览器或者Node。 在浏览器环境中，有JS 引擎线程和渲染线程，且两个线程互斥。Node环境中，只有JS 线程。 不同环境执行机制有差异，不同任务进入不同Event Queue队列。 当主程结束，先执行准备好微任务，然后再执行准备好的宏任务，一个轮询结束。

**浏览器中的事件环（Event Loop）**

事件环的运行机制是，先会执行栈中的内容，栈中的内容执行后执行微任务，微任务清空后再执行宏任务，先取出一个宏任务，再去执行微任务，然后在取宏任务清微任务这样不停的循环。

- eventLoop 是由JS的宿主环境（浏览器）来实现的；
- 事件循环可以简单的描述为以下四个步骤:
    1. 函数入栈，当Stack中执行到异步任务的时候，就将他丢给WebAPIs,接着执行同步任务,直到Stack为空；
    2. 此期间WebAPIs完成这个事件，把回调函数放入队列中等待执行（微任务放到微任务队列，宏任务放到宏任务队列）
    3. 执行栈为空时，Event Loop把微任务队列执行清空；
    4. 微任务队列清空后，进入宏任务队列，取队列的第一项任务放入Stack(栈）中执行，执行完成后，查看微任务队列是否有任务，有的话，清空微任务队列。重复4，继续从宏任务中取任务执行，执行完成之后，继续清空微任务，如此反复循环，直至清空所有的任务。

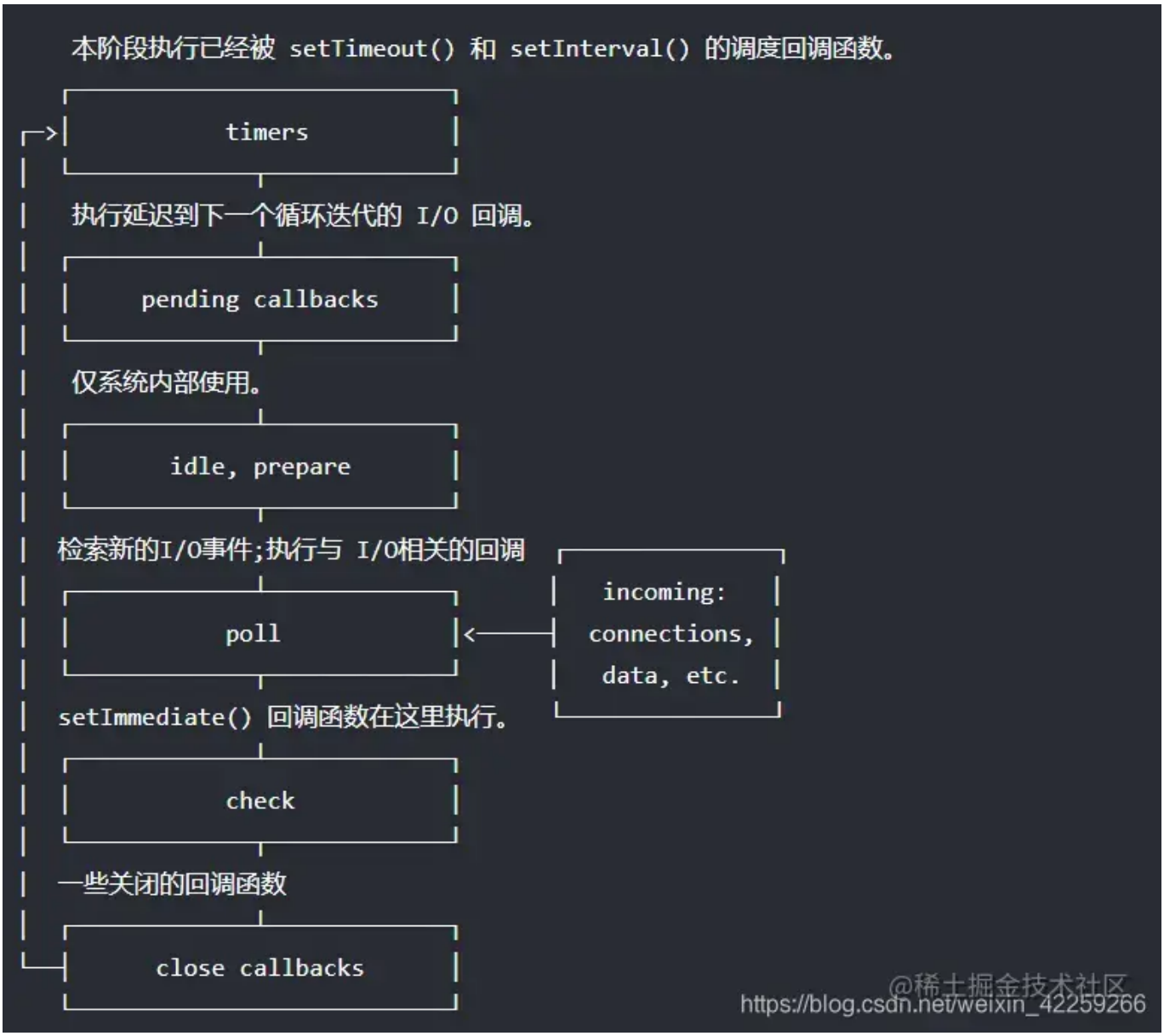



- 浏览器中的任务源(task):
  - 宏任务(macrotask)：
    宿主环境提供的，比如浏览器
    ajax、setTimeout、setInterval、setTmmediate(只兼容ie)、script、requestAnimationFrame、messageChannel、UI渲染、一些浏览器api
  - 微任务(microtask)：
    语言本身提供的，比如promise.then
    then、queueMicrotask(基于then)、mutationObserver(浏览器提供)、messageChannel 、mutationObersve

传送门 ☞ # 宏任务和微任务

**Node 环境中的事件环（Event Loop）**

Node是基于V8引擎的运行在服务端的JavaScript运行环境，在处理高并发、I/O密集(文件操作、网络操作、数据库操作等)场景有明显的优势。虽然用到也是V8引擎，但由于服务目的和环境不同，导致了它的API与原生JS有些区别，其Event Loop还要处理一些I/O，比如新的网络连接等，所以Node的Event Loop(事件环机制)与浏览器的是不太一样。




执行顺序如下：

- `timers`: 计时器，执行setTimeout和setInterval的回调
- `pending callbacks`: 执行延迟到下一个循环迭代的 I/O 回调
- `idle, prepare`: 队列的移动，仅系统内部使用
- `poll轮询`: 检索新的 I/O 事件;执行与 I/O 相关的回调。事实上除了其他几个阶段处理的事情，其他几乎所有的异步都在这个阶段处理。
- `check`: 执行`setImmediate`回调，setImmediate在这里执行
- `close callbacks`: 执行`close`事件的`callback`，一些关闭的回调函数，如：socket.on('close', ...)

## setTimeout、Promise、Async/Await 的区别

1. setTimeout

   settimeout的回调函数放到宏任务队列里，等到执行栈清空以后执行。

2. Promise

   Promise本身是**同步的立即执行函数**， 当在executor中执行resolve或者reject的时候, 此时是异步操作，会先执行then/catch等，当主栈完成后，才会去调用resolve/reject中存放的方法执行。

```
console.log('script start')
let promise1 = new Promise(function (resolve) {
    console.log('promise1')
    resolve()
    console.log('promise1 end')
}).then(function () {
    console.log('promise2')
})
setTimeout(function(){
    console.log('settimeout')
})
console.log('script end')
// 输出顺序: script start->promise1->promise1 end->script end->promise2->settimeout
```

3. async/await

async 函数返回一个 Promise 对象，当函数执行的时候，一旦遇到 await 就会先返回，等到触发的异步操作完成，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 async 函数体。

```
async function async1(){
   console.log('async1 start');
    await async2();
    console.log('async1 end')
}
async function async2(){
    console.log('async2')
}

console.log('script start');
async1();
console.log('script end')

// 输出顺序: script start->async1 start->async2->script end->async1 end
```

传送门 ☞ # JavaScript Promise 专题

## Async/Await 如何通过同步的方式实现异步

Async/Await就是一个**自执行**的generate函数。利用generate函数的特性把异步的代码写成"同步"的形式,第一个请求的返回值作为后面一个请求的参数,其中每一个参数都是一个promise对象.

## 介绍节流防抖原理、区别以及应用

节流：事件触发后，规定时间内，事件处理函数不能再次被调用。也就是说在规定的时间内，函数只能被调用一次，且是最先被触发调用的那次。

防抖：多次触发事件，事件处理函数只能执行一次，并且是在触发操作结束时执行。也就是说，当一个事件被触发准备执行事件函数前，会等待一定的时间（这时间是码农自己去定义的，比如 1 秒），如果没有再次被触

发，那么就执行，如果被触发了，那就本次作废，重新从新触发的时间开始计算，并再次等待 1 秒，直到能最终执行！

使用场景：
节流：滚动加载更多、搜索框搜的索联想功能、高频点击、表单重复提交.....
防抖：搜索框搜索输入，并在输入完以后自动搜索、手机号，邮箱验证输入检测、窗口大小 resize 变化后，再重新渲染。

```
/**
 * 节流函数 一个函数执行一次后，只有大于设定的执行周期才会执行第二次。有个需要频繁触发的函数，出于优化性能的角度，在规定时间内，只让函数触发的第一次生效，后面的不生效。
 * @param fn要被节流的函数
 * @param delay规定的时间
 */
function throttle(fn, delay) {
    //记录上一次函数触发的时间
    var lastTime = 0;
    return function(){
        //记录当前函数触发的时间
        var nowTime = Date.now();
        if(nowTime - lastTime > delay){
            //修正this指向问题
            fn.call(this);
            //同步执行结束时间
            lastTime = nowTime;
        }
    }
}

document.onscroll = throttle(function () {
    console.log('scllor事件被触发了' + Date.now());
}, 200);

/**
 * 防抖函数  一个需要频繁触发的函数，在规定时间内，只让最后一次生效，前面的不生效
 * @param fn要被节流的函数
 * @param delay规定的时间
 */
function debounce(fn, delay) {
    //记录上一次的延时器
    var timer = null;
    return function () {
       //清除上一次的演示器
        clearTimeout(timer);
        //重新设置新的延时器
        timer = setTimeout(function(){
            //修正this指向问题
            fn.apply(this);
        }, delay);
    }
}
document.getElementById('btn').onclick = debounce(function () {
```

```
    console.log('按钮被点击了' + Date.now());
}, 1000);
```