# JS进阶

## 闭包问题

得分点： 变量背包、作用域链、局部变量不销毁、函数体外访问函数的内部变量、内存泄漏、内存溢出、形成块级作用域、柯里化、构造函数中定义特权方法、Vue中数据响应式Observer

标准回答：

**一个函数和词法作用域的引用捆绑在一起，这样的组合**就是闭包（closure）。一般就是一个函数A，return其内部的函数B，被return出去的B函数能够在外部访问A函数内部的变量，这时候就形成了一个B函数的变量背包，A函数执行结束后这个变量背包也不会被销毁，并且这个变量背包在A函数外部只能通过B函数访问。

闭包形成的原理：作用域链，当前作用域可以访问上级作用域中的变量

闭包解决的问题：能够让函数作用域中的变量在函数执行结束之后不被销毁，同时也能在函数外部可以访问函数内部的局部变量。

闭包带来的问题：由于垃圾回收器不会将闭包中变量销毁，于是就造成了内存泄露，内存泄露积累多了就容易导致内存溢出。

加分回答： 闭包的应用，能够模仿块级作用域，能够实现柯里化，在构造函数中定义特权方法、Vue中数据响应式Observer中使用闭包等。

### 闭包的应用

封装私有变量、创建模块化的代码、实现柯里化和高阶函数等

## 高阶函数

是一种特殊的函数，它具有以下特点：

1.接受其他函数作为参数：高阶函数可以作为参数传递给其他函数，就像普通的函数一样。

2.返回另一个函数作为结果：除了可以作为参数传递外，高阶函数还可以返回另一个函数作为结果。

## 节流与防抖的原理和应用场景

防抖：当一个动作连续触发，只执行最后一次

节流：限制一个动作在一段时间内只能执行一次

## 解释一下 JavaScript 的同源策略？

概念:同源策略是客户端脚本（尤其是Netscape Navigator2.0，其目的是防止某个文档或脚本从多个不同源装载。

这里的同源策略指的是：协议，域名，端口相同，同源策略是一种安全协议。

指一段脚本只能读取来自同一来源的窗口和文档的属性。

## 函数柯里化

什么叫函数柯里化？其实就是将使用多个参数的函数转换成一系列使用一个参数的函数的技术。还不懂？来举个例子。

function add(a, b, c) {

return a + b + c

}

add(1, 2, 3)

let addCurry = curry(add)

addCurry(1)(2)(3)

现在就是要实现 curry 这个函数，使函数从一次调用传入多个参数变成多次调用每次传一个参数。

function curry(fn) {

let judge = (...args) => {

if (args.length == fn.length) return fn(...args)

return (...arg) => judge(...args, ...arg)

}

return judge

}

## JavaScript事件循环

JavaScript作为一门单线程语言，如何通过**事件循环（Event Loop）和任务队列（Task Queue）**的机制，高效地处理异步任务，保证用户体验的流畅性。

### 事件循环

JavaScript是单线程执行模型，执行的时候将会区分为**主线程和任务队列**。主线程执行完毕，会从任务队列中读取新的任务放入主线程进行执行，这个读取过程是**循环读取**，所以也叫**事件循环**。

任务队列分为宏任务和微任务，同层次，**先执行微任务，再执行宏任务**。

在事件循环中，当主线程执行完当前的同步任务后，会检查事件队列中是否有待处理的事件。如果有，主线程会取出事件并执行对应的回调函数。这个循环的过程被称为事件循环（Event Loop），它由主线程和任务队列两部分组成。主线程负责执行同步任务，而异步任务则通过任务队列进行处理。这种机制保证了异步任务在适当的时机能够插入执行，从而实现了JavaScript的非阻塞异步执行。

#### 事件循环流程如下：

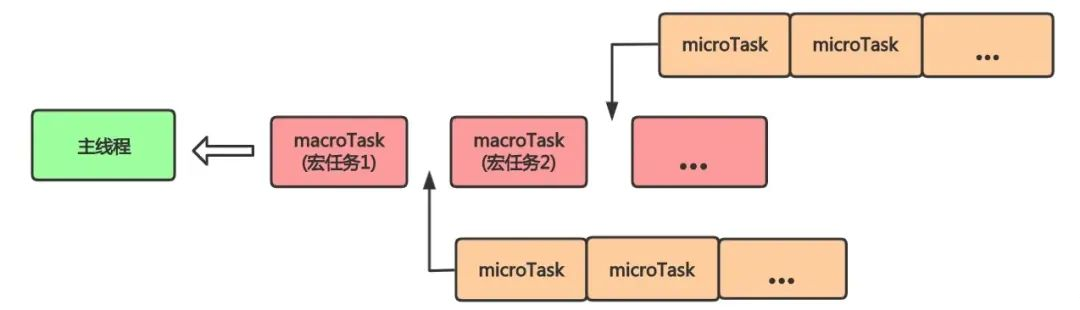
主线程读取JavaScript代码，形成相应的堆和执行栈。

当主线程遇到异步任务时，将其委托给对应的异步进程（如Web API）处理。

异步任务完成后，将相应的回调函数推入任务队列。

主线程执行完同步任务后，检查任务队列，如果有任务，则按照先进先出的原则将任务推入主线程执行。

重复执行以上步骤，形成事件循环。



#### 同步任务

同步任务是按照代码的书写顺序一步一步执行的任务。当主线程执行同步任务时，会阻塞后续的代码执行，直到当前任务执行完成。典型的同步任务包括函数调用、变量赋值、算术运算等。

#### 异步任务

异步任务是在主线程执行的同时，通过回调函数或其他机制委托给其他线程或事件来处理的任务。在执行异步任务时，主线程不会等待任务完成，而是继续执行后续代码。包括：

回调函数 callback

Promise/async await

Generator

事件监听

发布/订阅

计时器

requestAnimationFrame

MutationObserver

process.nextTick

I/O操作

不得不说，异步执行的机制使得 JavaScript 能够更好地处理耗时操作，保持页面的响应性。

### 任务队列

上面我们讨论了同步任务和异步任务的执行过程，接下来我们将进一步探讨任务队列，了解它的最小颗粒度是如何执行的。

在JavaScript中，任务队列是**一个执行异步任务的机制**，由浏览器内部的“任务队列管理器”管理。JavaScript中的任务分为两类：同步任务和异步任务。同步任务会在当前的执行栈中执行，而异步任务会被添加到任务队列中，等待当前的同步任务执行完毕之后再执行。

#### 任务队列类型

任务队列分为宏任务队列（macrotask queue）和微任务队列（microtask queue）两种。JavaScript 引擎遵循事件循环的机制，在执行完当前宏任务后，会检查微任务队列，执行其中的微任务，然后再取下一个宏任务执行。这个过程不断循环，形成事件循环。

#### 宏任务和微任务

事件循环由宏任务和在执行宏任务期间产生的所有微任务组成。完成当下的宏任务后，会立刻执行所有在此期间入队的微任务。

这种设计是为了给紧急任务一个插队的机会，否则新入队的任务永远被放在队尾。区分了微任务和宏任务后，本轮循环中的微任务实际上就是在插队，这样微任务中所做的状态修改，在下一轮事件循环中也能得到同步。

常见的**宏任务**有：

script（整体代码）

setTimout

setInterval

setImmediate(node 独有)

requestAnimationFrame(浏览器独有)

IO

UI render（浏览器独有）

常见的**微任务**有：

process.nextTick(node 独有)

Promise.then()

Object.observe

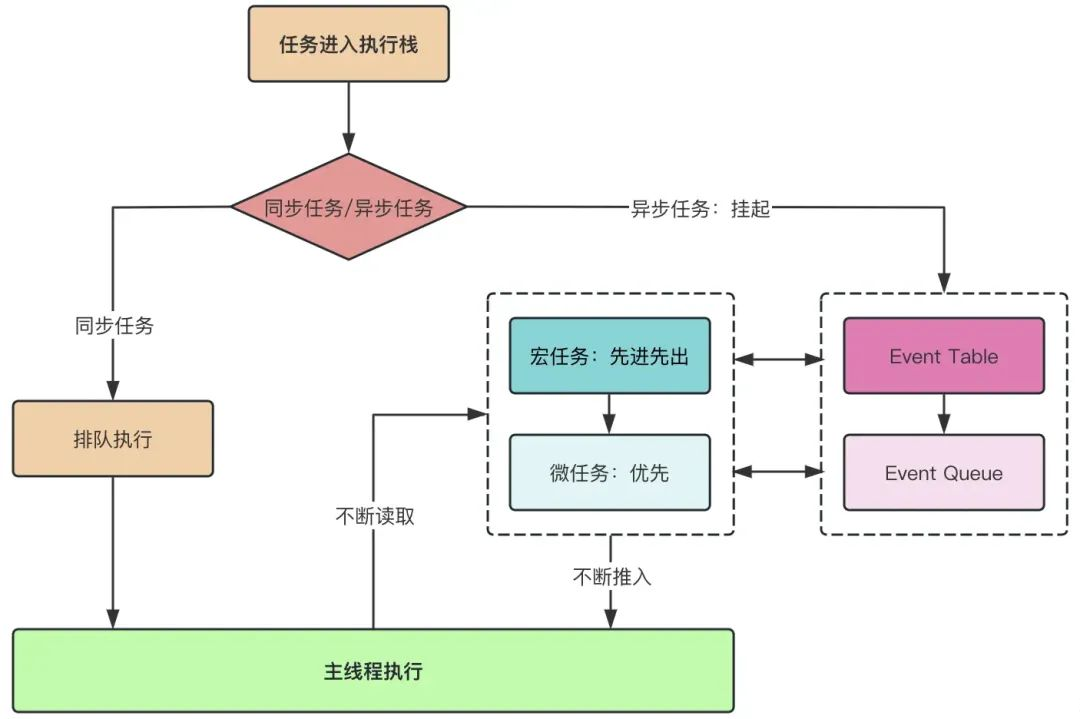
MutationObserver

Generator函数

#### 任务执行过程

首先，必须要明确，在JavaScript中，所有任务都在主线程上执行。任务执行过程分为同步任务和异步任务两个阶段。异步任务的处理经历两个主要阶段：Event Table（事件表）和 Event Queue（事件队列）。

Event Table存储了宏任务的相关信息，包括事件监听和相应的回调函数。当特定类型的事件发生时，对应的回调函数被添加到事件队列中，等待执行。



#### 任务队列的执行流程

同步任务在主线程排队执行，异步任务在事件队列排队等待进入主线程执行。

遇到宏任务则推进宏任务队列，遇到微任务则推进微任务队列。

执行宏任务，执行完毕后检查当前层的微任务并执行。

继续执行下一个宏任务，执行对应层次的微任务，直至全部执行完毕。

这个流程确保了异步任务能够在适当的时机插入执行，保持程序的高效性和响应性。

## requestAnimationFrame、setInterval、setTimeout 功能解析与优劣对比

### requestAnimationFrame

requestAnimationFrame是一种在浏览器中实现动画循环的技术，它通过定时器机制来周期性地调用指定的回调函数，以实现网页动画的效果。与传统的setInterval和setTimeout不同，requestAnimationFrame具有更好的浏览器兼容性、更精确的定时控制和更优秀的性能表现。

使用requestAnimationFrame可以将动画的每一帧绘制操作封装为一个回调函数，并将这个回调函数传递给requestAnimationFrame函数。当浏览器准备进行下一帧绘制时，会自动调用这个回调函数，从而实现了动画的循环。

注意：

* 回调函数的参数是一个时间戳（timestamp），表示当前时间点距离页面加载完成的时间。
* 回调函数的执行时间间隔是不固定的，它由浏览器根据渲染性能和其他因素进行调度。
* 如果在一段时间内没有再次调用requestAnimationFrame，则动画会停止。
* 在浏览器处于非激活状态时，requestAnimationFrame的回调函数不会执行。
* requestAnimationFrame可以用于实现各种类型的动画效果，包括DOM动画、Canvas动画、SVG动画、WebGL动画等。

语法：

requestAnimationFrame(callback);

参数

callback： 下一次重绘之前更新动画帧所调用的函数(即上面所说的回调函数)。该回调函数会被传入DOMHighResTimeStamp参数，该参数与performance.now()的返回值相同，它表示requestAnimationFrame() 开始去执行回调函数的时刻。

返回值

一个 long 整数，请求 ID ，是回调列表中唯一的标识。是个非零值，没别的意义。你可以传这个值给 window.cancelAnimationFrame() 以取消回调函数。

### setTimeout()

setTimeout()方法用来指定某个函数或字符串在指定的毫秒数之后执行。它返回一个整数，表示定时器的编号，这个值可以传递给clearTimeout()用于取消这个函数的执行

如果省略setTimeout的第二个参数，则该参数默认为0

实际上，除了前两个参数，setTimeout()方法还允许添加更多的参数，它们将被传入定时器中的函数中

以下代码中，控制台大概过1000ms即1s后，输出2，而IE9-浏览器只允许setTimeout有两个参数，不支持更多的参数，会在控制台输出NaN

setTimeout(function(a,b){

console.log(a+b);

},1000,1,1);

### setInterval()

setInterval是一个JavaScript函数，用于按照指定的周期（以毫秒计）来调用函数或计算表达式。它能够以相对准确的时间间隔执行代码，并且可以用于创建定时任务或动画。

setInterval的用法与setTimeout完全一致，区别仅仅在于setInterval指定某个任务每隔一段时间就执行一次，也就是无限次的定时执行

HTML5标准规定，setTimeout的最短时间间隔是4毫秒；setInterval的最短间隔时间是10毫秒，也就是说，小于10毫秒的时间间隔会被调整到10毫秒

大多数电脑显示器的刷新频率是60HZ，大概相当于每秒钟重绘60次。因此，最平滑的动画效的最佳循环间隔是1000ms/60，约等于16.6ms

为了节电，对于那些不处于当前窗口的页面，浏览器会将时间间隔扩大到1000毫秒。另外，如果笔记本电脑处于电池供电状态，Chrome和IE10+浏览器，会将时间间隔切换到系统定时器，大约是16.6毫秒

setInterval方法会返回一个ID值，这个ID值可以用于停止该定时器。可以使用clearInterval(id)函数来停止指定的定时器，其中id是setInterval返回的ID值。

setInterval的工作原理是，当指定的时间间隔过去后，它会自动执行一次指定的代码或函数，然后等待下一个时间间隔。这个过程会一直重复，直到clearInterval被调用或窗口被关闭。

需要注意的是，setInterval并不能保证代码或函数严格按照指定的时间间隔执行。如果代码或函数执行的时间过长，或者处理一些异步操作时，可能会导致实际执行的时间间隔与指定的时间间隔有所偏差。此外，如果浏览器或计算机的性能不足，也可能会导致定时器的执行出现延迟。

### 对比

#### 工作原理：

setTimeout：这个函数会将要执行的代码或函数放入事件循环队列中，等待当前代码执行完毕后，再等待指定的时间后执行一次。如果设置了定时器，那么每隔一定时间就会执行一次代码，直到 clearTimeout 被调用或窗口被关闭。

setInterval：与 setTimeout 类似，setInterval 也会将要执行的代码或函数放入事件循环队列中，但它在指定的时间间隔后会一直重复执行，直到 clearInterval 被调用或窗口被关闭。也就是说，setInterval 会不断地调用函数，直到被取消。

requestAnimationFrame：这个函数的工作原理与 setInterval 和 setTimeout 略有不同。它会将回调函数加入到浏览器下一次重绘之前要执行的队列中。这样做的目的是为了确保动画的流畅度，因为浏览器会自动优化这个API，只在浏览器处于激活状态并且页面处于可见状态时才会执行回调函数。此外，requestAnimationFrame 会根据系统的刷新率来自动匹配时间间隔，从而确保每帧动画的间隔时间尽可能地准确。

#### 区别：

**执行时机**：requestAnimationFrame是由浏览器提供的API，它会在浏览器下一次重绘之前执行回调函数。这意味着它能够确保动画的流畅度，并且能够自动匹配系统的刷新率。相比之下，setInterval和setTimeout会在指定的时间间隔后执行回调函数，无论浏览器是否处于激活状态或正在进行其他操作。

**性能优化**：requestAnimationFrame由浏览器自动优化，只在浏览器处于激活状态并且页面处于可见状态时才会执行回调函数。这可以节省CPU、GPU和内存的使用，特别是在移动设备上。相比之下，setInterval和setTimeout不会自动优化，如果页面处于隐藏或不可见状态，它们会继续执行回调函数，这可能会导致资源的浪费。

**回调函数执行时间**：requestAnimationFrame的回调函数会在浏览器下一次重绘之前执行，因此它能够确保回调函数的执行时间相对准确。相比之下，setInterval和setTimeout的回调函数执行时间取决于浏览器事件循环中的队列和执行时间，因此可能会有一定的延迟。

**停止操作**：requestAnimationFrame的回调函数只会在浏览器下一次重绘之前执行一次，因此可以通过清除队列中的回调函数来停止操作。相比之下，setInterval和setTimeout会不断地执行回调函数，直到clearInterval或clearTimeout被调用或关闭页面为止。

**函数节流**：在高频率事件(resize,scroll等)中，为了防止在一个刷新间隔内发生多次函数执行，使用requestAnimationFrame可保证每个刷新间隔内，函数只被执行一次，这样既能保证流畅性，也能更好的节省函数执行的开销。

#### 应用场景

setTimeout：可用于在网页加载后延迟执行某些操作，例如加载页面内容、初始化组件等。也可用于定时触发某些操作，例如定时发送数据、定时检查任务等。

setInterval：常用于需要周期性执行的操作，例如定时更新数据、定时触发事件等。在web端，如果列表需要定时更新，可以使用setInterval来定时获取列表的请求。另外，如果需要在某一特定情况下清除定时任务，可以使用clearInterval来停止定时器。

requestAnimationFrame：主要用于实现流畅的动画效果。它会在浏览器下一次重绘之前执行指定的函数，避免了频繁的重绘导致的性能问题。requestAnimationFrame会自动匹配系统的刷新率，从而确保每帧动画的间隔时间尽可能地准确。在需要反复触发的情况下，使用requestAnimationFrame可以避免连续调用导致的相互干扰。

## 原型链

每个对象（object）都有一个私有属性指向另一个名为原型（prototype）的对象

js分为函数对象和普通对象，每个对象都有\_\_proto\_\_属性，但是只有函数对象才有prototype属性

Object、Function都是js内置的函数, 类似的还有我们常用到的Array、RegExp、Date、Boolean、Number、String

属性\_\_proto\_\_是一个对象，它有两个属性，constructor和\_\_proto\_\_；

原型对象prototype有一个默认的constructor属性，用于记录实例是由哪个构造函数创建；

js原型、原型链默认遵从以下两个准则

* 准则1：原型对象（即Person.prototype）的constructor指向构造函数本身
* 准则2：实例（即per0）的\_\_proto\_\_和原型对象指向同一个地方

## 作用域

js中的作用域说的是变量的可访问性和可见性。也就是说整个程序中哪些部分可以访问这个变量，或者说这个变量都在哪些地方可见。

* 全局作用域
* 函数作用域
* 块级作用域

## 作用域链

当执行一段JavaScript代码（全局代码或函数）时，JavaScript引擎会为其创建一个作用域又称为执行上下文（Execution Context），在页面加载后会首先创建一个全局的作用域，然后每执行一个函数，会建立一个对应的作用域，从而形成了一条**作用域链**。**每个作用域都有一条对应的作用域链，链头是全局作用域，链尾是当前函数作用域。**

作用域链的作用是用于**解析标识符**，当函数被创建时（不是执行），会将this、arguments、命名参数和该函数中的所有局部变量添加到该当前作用域中，当JavaScript需要查找变量X的时候（这个过程称为变量解析），它首先会从作用域链中的链尾也就是当前作用域进行查找是否有X属性，如果没有找到就顺着作用域链继续查找，直到查找到链头，也就是全局作用域链，仍未找到该变量的话，就认为这段代码的作用域链上不存在x变量，并抛出一个引用错误（ReferenceError）的异常。

* 作用域分为全局作用域和局部作用域。
* 作用域其实就是规定了当前作用域中的变量和函数可被作用的范围。
* 作用域链其实就是规定了变量和函数的查找规则、是当前执行上下文的变量对象以及所有父级执行上下文的变量对象的集合。
* 当查找一个变量时，先从作用域的顶端查找，一直查找到作用域的底端，若查找完仍未找到，抛出错误。

## js作用域的销毁、不立即销毁、不销毁

JavaScript中的函数执行会形成私有的作用域。

（1）作用域的销毁

　　一般情况下，函数执行形成一个私有的作用域，当执行完成后就销毁了->节省内存空间

（2）作用域的不立即销毁

function fn(){

var i=10;

return function(n){

console.log(n+i++);

　　}

}

fn()(15);//->先执行fn,有一个私有的变量i=10,返回一个堆内存地址 xxxfff111,我们发现这个地址还用到了一次,那么当前的这个fn形成的私有作用域（A）就不能立即销毁了,

　　　　　　xxxfff111(15)->输出25,A中的i变为11；当xxxfff111执行完了,发现这个地址没用了,浏览器就把A、xxxfff111都释放了

fn()(20);//->在执行fn的时候一切都从新开始了,和上面的步骤是一样的->输出30

（3）作用域的立即销毁

function fn(){

var i=10;

return function(n){

console.log(n+i++);

}

}

var f=fn();//->fn执行形成一个私有的作用域A,A中有一个私有的变量i=10,A中返回一个地址xxxfff11,被外面的f占用了,那么当前的A就不能销毁了

f(15);//->输出25,让A中的i=11

f(20);//->输出31,让A中的i=12

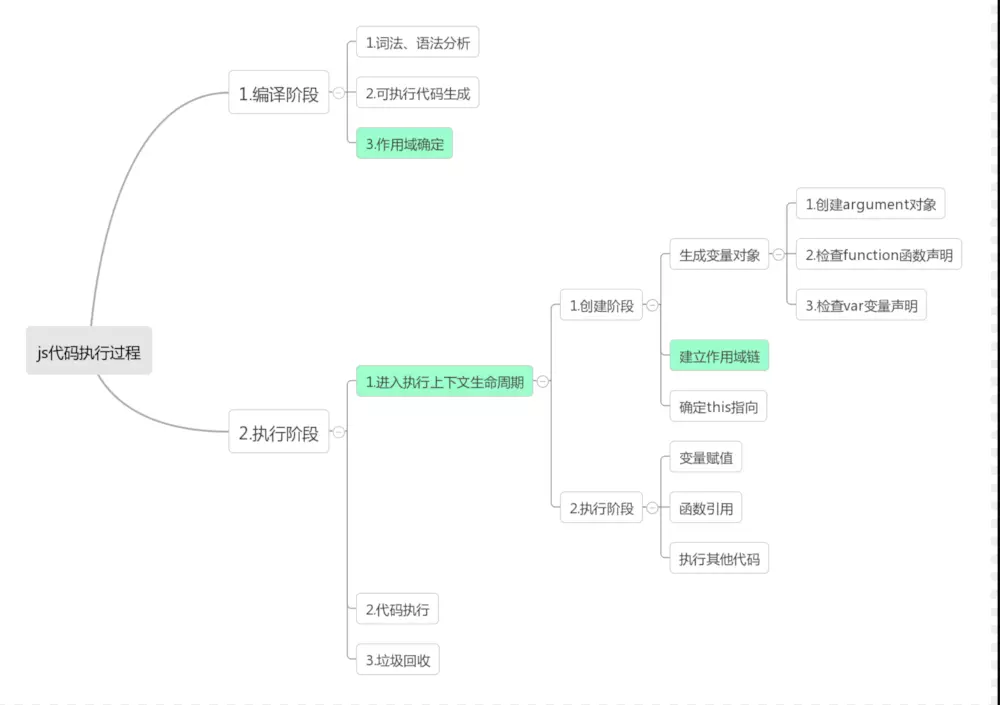
### 几种不销毁常用到的形式:

（1）函数执行,返回一个引用数据类型的值,并且在函数的外面被别人接收了,那么当前函数形成的私有作用域就不在销毁了-->例如上面的案例 var f=fn();

（2）在函数执行的时候,里面的一个小函数的地址赋值给了外面元素的点击事件,那么当前小函数也相当于被外面占用了,大函数执行形成的私有的作用域也不能销毁了

（3）在使用setTimeout实现轮询动画的时候,我们如果move需要传递参数值,那么像下面这样的写法会行成很多的不销毁的作用域,非常的耗性能

## JavaScript 执行过程



JavaScript整个执行过程，分为两个阶段，**代码编译阶段**和**代码执行阶段**

### 一、编译阶段

编译阶段由编译器执行，将代码翻译成可执行代码，这个阶段确定作用域规则。

变量的预编译只作声明，不作初始化，初始化在执行时

function语句定义的函数，不仅声明了函数名，而且函数体也进行了处理

### 二、执行阶段

执行阶段由引擎完成，主要任务是执行可执行代码，执行上下文在这个阶段创建。执行上下文也分为创建阶段和执行阶段。

1、执行上下文

执行上下文可以理解为当前代码的执行环境，它会形成一个作用域。

2、三种类型

a、全局执行上下文：浏览器的全局对象是window对象，this指向这个全局对象

b、函数执行上下文：存在无数个，只有在函数被调用的时候才会被创建，每次调用函数都会创建一个新的执行上下文，函数上下文的变量对象初始化只包括 Arguments 对象。

c、Eval函数执行上下文：指运行在eval函数的代码，很少用而且不建议使用。。

创建阶段：

确定this的值，也被成为 This Binding；

a、全局执行上下文中，this 指向全局对象，在浏览器中this 指向window对象，在nodejs中指向这个文件的module 对象。

b、函数执行上下文中，this的值取决于函数的调用方式。具体有：默认绑定、隐式绑定、显式绑定、new绑定、箭头函数。

需要注意的是同一作用域下存在多个同名函数声明，后面的会替换前面的函数声明。

### 三、执行上下文栈

因为JS引擎创建了很多的执行上下文，所以JS引擎创建了执行上下文栈（Execution context stack，ECS）来管理执行上下文。

当 JavaScript 初始化的时候会向执行上下文栈压入一个全局执行上下文，我们用 globalContext 表示它，并且只有当整个应用程序结束的时候，执行栈才会被清空，所以程序结束之前， 执行栈最底部永远有个 globalContext。

当执行一个函数的时候，就会创建一个执行上下文，并且压入执行上下文栈，当函数执行完毕的时候，就会将函数的执行上下文从栈中弹出。

## 深拷贝&浅拷贝&赋值

### 1.深拷贝

深拷贝是将一个对象从内存中完整的拷贝一份出来，从堆内存中开辟一个新的区域存放新对象（新旧对象不共享同一块内存），且修改新对象不会影响原对象（`深拷贝采用了在堆内存中申请新的空间来存储数据，这样每个对象可以避免指针悬挂`）

### 2.浅拷贝

如果属性是基本类型，拷贝的就是基本类型的值。如果属性是引用类型，那么拷贝的就是内存地址（新旧对象共享同一块内存），所以如果其中一个对象改变了这个地址，就会影响到另一个对象（`只是拷贝了指针，使得两个指针指向同一个地址`）

### 3.赋值与浅拷贝的区别

1. 把一个对象赋值给一个新的变量的时候，赋的其实是该对象在栈中的地址，而不是栈中的数据。也就是这两个对象指的是同一个储存空间，不论哪个对象发生改变，其实都是改变储存空间里的内容。因此，两个对象是联动的。

2. 浅拷贝是按位拷贝对象的，他会创建一个新的对象，这个对象有着原始对象属性值的一份精确拷贝。

* 如果属性是基本类型，拷贝的就是基本类型的值。
* 如果属性是内存地址（引用类型），拷贝的就是内存地址，因此如果一个对象改变了这个地址，就会影响到另一个对象。
* 即默认拷贝构造函数只是对对象进行浅拷贝复制（逐个成员依次拷贝），即只复制对象空间而不复制资源。

### 4.浅拷贝的实现【当拷贝对象只有一层的时候，是深拷贝】

* 展开运算符...
* Object.assign()
* Array.prototype.concat()
* Array.prototype.slice()

### 5.深拷贝的实现

* JSON.parse(JSON.stringify())
* 手写递归方法：(递归方法实现深度克隆原理：遍历对象、数组直到里边都是基本数据类型，然后再去复制，就是深度拷贝)
* jQuery的extend方法实现深拷贝
* 函数库lodash的\_.cloneDeep方法

### 6.JSON.parse(JSON.stringify())有什么缺陷？

数据里面有特殊符号、正则这种符号。json.parse进行拷贝可能会出问题

## 怎样解决深拷贝中的循环拷贝

**使用weakmap**：解决循环引用问题，我们可以额外开辟一个存储空间，来存储当前对象和拷贝对象的对应关系这个存储空间，需要可以存储 key-value 形式的数据，且 key 可以是一个引用类型，我们可以选择 weakMap 这种数据结构:

**可以使用 set**：发现相同的对象直接赋值，也可以用 Map

## 跨标签通信的应用场景

1共享登录状态：当用户在一个标签页中登录后，其他打开的标签页需要及时获取到登录状态，以保持一致的用户体验。在这种情况下，可以使用浏览器的localStorage或sessionStorage来存储登录状态，并通过监听storage事件来实现不同标签页之间的状态同步。

2实时通知和消息推送：如果用户在一个标签页上收到了新消息或通知，可以通过跨标签页通信将该消息或通知传递给其他标签页。一种常见的处理方式是使用浏览器的localStorage或IndexedDB来存储未读消息或通知，然后通过监听storage事件或定时轮询来检查新消息或通知的变化。

3跨标签页数据共享：有时候需要在不同的标签页之间共享一些数据，例如购物车数据、选项设置等。这可以通过在localStorage或IndexedDB中存储数据，并借助storage事件或定时轮询来实现数据的同步更新。

4标签页之间的导航同步：当用户在一个标签页中进行导航操作（例如点击链接或提交表单）时，其他标签页可能也需要跟随导航到相应的页面。这可以通过在标签页之间发送消息或共享状态来实现导航的同步。

## 操作系统中，有以下几种通信方式

1基于管道的通信：

管道是一种半双工的通信机制，可用于同一父进程与其子进程之间通信，或者用于同一计算机上的不同进程之间通信。

命名管道提供了进程间进行双向通信的能力。可以被多个进程打开和使用。其中一个进程将数据写入管道，而另一个进程则可以从管道中读取这些数据。命名管道通常用于在不相关的进程之间传递数据，比如客户端和服务器之间的通信。

匿名管道是一种用于单向通信的机制，仅用于具有父子关系的进程之间。它只能在创建时通过操作系统提供的机制进行传递。匿名管道在创建时自动建立，并且只能用于具有亲缘关系的进程之间的通信。其中一个进程将数据写入管道的写端，而另一个进程则从管道的读端读取这些数据。

2消息队列：消息队列允许进程通过将消息放入队列中来进行通信。进程可以从队列中接收消息，实现异步通信。消息队列适用于不需要直接的点对点连接的场景，而且可以在不同计算机之间通信。

3共享内存：共享内存允许多个进程访问同一块物理内存区域，从而实现高效的数据共享。进程可以在共享内存中读写数据，而不需要显式的数据传输操作。

4套接字Socket：套接字通信是一种在计算机网络中实现进程间通信的方式。它基于网络协议栈，使用 TCP 或 UDP 等传输层协议，在不同的主机之间进行数据传输和通信。

5Remote Procedure Call：RPC 允许一个进程通过网络请求调用另一个进程中的函数，就像调用本地函数一样。远程过程调用隐藏了底层通信细节，使得进程间通信更加方便。

6信号（Signal）：信号通信是一种在操作系统中实现进程间通信的机制。它允许一个进程向另一个进程发送信号，用于通知、中断或请求处理等目的。它是一种异步事件，当某个事件发生时，操作系统会向进程发送相应的信号。进程可以事先注册信号处理函数来捕获并处理这些信号

## JavaScript 如何实现跨标签页通信

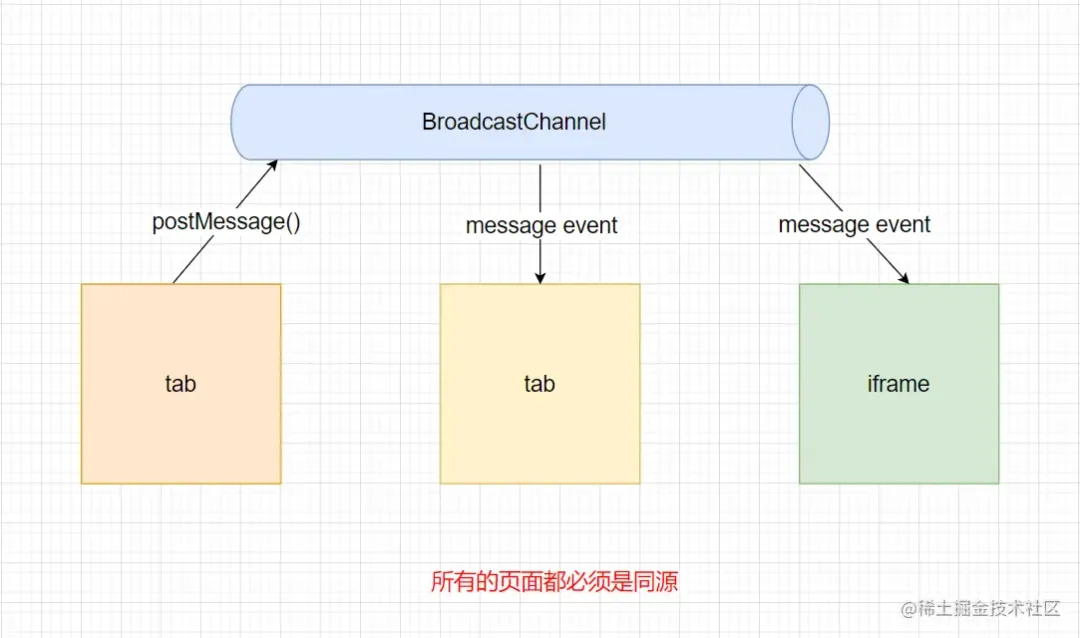
JavaScript 跨标签通信允许不同的浏览器标签页之间进行数据传递和通信，为构建更复杂和协同的 Web 应用程序提供了一种机制。通过适当的技术和策略，我们可以实现有效的跨标签通信，以满足不同应用场景的需求。

### 1.BroadcastChannel

BroadcastChannel 通信的方式原理就是一个命名管道。它允许让指定的同源下浏览器不同的窗口来订阅它。

每个 BroadcastChannel 对象都需要使用一个唯一的名称来标识通道，这个名称在同一域名下的不同页面之间必须是唯一的。它允许同一域名下的不同页面之间进行通信。

通过 postMessage 方法，一个页面可以将消息发送到频道中，而其他页面则可以监听 message 事件来接收这些消息。通过这种方式是短线了一种实时通信的机制，可以在不同的页面之间传递信息，实现页面间的即时交流。如下图所示：



### 2.Service Worker

Service Worker 它是一种服务工作线程，是一种在浏览器背后运行的脚本，用于处理网络请求和缓存等任务。它是一种在浏览器与网络之间的中间层，允许开发者拦截和控制页面发出的网络请求，以及管理缓存，从而实现离线访问、性能优化和推送通知等功能。

它在浏览器背后独立运行与网页分开，这意味着即使用户关闭了网页，Service Worker 仍然可以运行。可以用于实现推送通知功能。它可以注册为推送消息的接收者，当服务器有新的通知要发送时，Service Worker 可以显示通知给用户，即使网页没有打开。

### 3.localStorage

在 Web Storage 中,每一次将一个值存储到本地存储时,都会触发一个 storage 事件,由事件监听器发送给回调函数的事件对象

### 4.SharedWorker

SharedWorker 是一种在 Web 浏览器中使用的 Web API，它允许不同的浏览上下文,如不同的浏览器标签页之间共享数据和执行代码。它可以用于在多个浏览上下文之间建立通信通道，以便它们可以共享信息和协同工作。

与普通的 Worker 不同，SharedWorker 可以在多个浏览上下文中实例化，而不仅限于一个单独的浏览器标签页或框架。这使得多个浏览上下文可以共享同一个后台线程，从而更有效地共享数据和资源，而不必在每个标签页或框架中都创建一个独立的工作线程。

### 5.IndexDB

IndexedDB 是一种在浏览器中用于存储和管理大量结构化数据的 Web API。它提供了一种持久性存储解决方案，允许 Web 应用程序在客户端存储数据，以便在不同会话、页面加载或浏览器关闭之间保留数据。

与传统的 cookie 或 localStorage 等存储方式不同，IndexedDB 更适合存储复杂的、结构化的数据，例如对象、数组、键值对等。这使得它特别适用于应用程序需要存储大量数据、执行高级查询或支持离线工作的情况。

### 6.cookie

### 7.postMessage

window.postMessage() 方法可以安全地实现跨源通信。通常，对于两个不同页面的脚本，只有同源时，这两个脚本才能相互通信。

## 谈谈前端路由的实现原理【hash＆history】

### 路由

路由一词最早来自服务器，和前端没有关系。当你想要从服务器中读取某个盘的文件，这个文件的路径就是路由。也就是说「路由是服务器端用来描述路径的，或者是说url和文件的映射关系」

后来因为前端的SPA单页应用，前端也借鉴了路由这个概念。浏览器的url变了需要映射到页面的某个组件，url变了需要展示某个组件。/home和Home.vue，/about和About.vue就是一一映射的关系。「前端借鉴路由的称呼来描述url和组件的映射关系」。这个时候你就想起来router中index.js文件中，一个path对应一个component，也就是一个路径对应一个组件

1.如何修改url还不引起页面的刷新

2.如何知道url变化了

若是能解决这两个问题就可以实现前端路由了。

### 哈希Hash

哈希是一种值，按照某种规则生成的一串值，用来代表一个唯一的文件，文件名后加一个哈希值，可以看到文件是否被修改过。

在浏览器中也有hash这个概念，url中接一个#，#后的值就是哈希值，按道理url变了，页面一定会刷新，但是哈希是个特例，放个哈希值就是不会刷新页面，这样，我们就解决了第一个问题，修改url不引起页面的刷新

「在浏览器url后加个哈希值，哈希值的变更不会引起浏览器页面的刷新」

<body>

<ul>

<li><a href="#/home">首页</a></li>

<li><a href="#/about">关于</a></li>

</ul>

<div id="routeView">

<!-- 放一个代码片段 点击首页首页代码片段生效，反之关于生效-->

</div>

<script>

const routes = [{

path: '#/home',

component: '首页内容'

},

{

path: '#/about',

component: '关于页面内容'

}

]

window.addEventListener('DOMContentLoaded', onHashChange)

window.addEventListener('hashchange', onHashChange)

const routeView = document.getElementById('routeView')

console.log("🚀 ~ routeView:", routeView)

function onHashChange() {

console.log(location)

routes.forEach((item, index) => {

if (item.path === location.hash) {

routeView.innerHTML = item.component

}

})

}

</script>

</body>

其实这就是vue-router中两种模式之一哈希模式，哈希模式就是这样是实现的。

### 修改地址栏 history

1.a标签

2.浏览器前进后退

3.window.location

以上方式导致url变更都会触发hashchange事件。

history用得更多，二者没有本质区别，仅仅是因为哈希模式的url多了个#很丑，所以用的少

history自带的方法pushState

它可以修改url且不引起页面的刷新

浏览器中有个会话历史栈，它可以维护你的访问路径，有了这个你返回就可以按照栈的顺序进行前进回退。

pushState提到了popState，他是靠popState监听url的改变的，并且仅当浏览器前进后退时生效

<body>

<ul>

<li><a href="/home">首页</a></li>

<li><a href="/about">关于</a></li>

</ul>

<div id="routeView">

</div>

<script>

const routes = [{

path: '/home',

component: '首页内容'

},

{

path: '/about',

component: '<h1>关于页面内容</h1>'

}

]

const routeView = document.getElementById('routeView')

window.addEventListener('DOMContentLoaded', onLoad)

window.addEventListener('popstate', onPopState)

function onLoad() {

const links = document.querySelectorAll('li a') // 获取所有的li下的a标签

// console.log(links)

links.forEach((a) => {

// 禁用a标签的默认跳转行为

a.addEventListener('click', (e) => {

console.log(e)

// 阻止a的跳转行为

e.preventDefault()

// 核心方法 a.getAttribute('href')获取a标签下的href属性

history.pushState(null, '', a.getAttribute('href'))

// 映射对应的dom

onPopState()

})

})

}

function onPopState() {

console.log(location.pathname)

routes.forEach((item) => {

if (item.path === location.pathname) {

routeView.innerHTML = item.component

}

})

}

</script>

</body>

## 模块化

### 1.什么是模块化

* 将一个复杂的程序，依据一定的规则(规范)封装成一个或多个块(文件), 并进行组合在一起
* 块的内部数据与实现是私有的, 只是向外部暴露一些接口(方法)与外部其它模块通信

模块化是一种将系统分离成独立功能部分的方法，可将系统分割成独立的功能部分，严格定义模块接口、模块间具有透明性。

### 2.模块化的进化过程

#### a.无模块时代

在ajax还未提出之前，js还只是用来在网页上进行表单校验、提交，对DOM渲染操作。

缺点：

全局变量污染

函数命名冲突

文件依赖顺序

#### b.模块雏形时代

2006年，ajax的概念被提出，前端拥有了主动向服务端发送请求并操作返回数据的能力，传统的网页向“富客户端”发展，出现了简单的功能对象封装。

#### c.namespace模式

优点: 减少了全局变量，解决命名冲突

缺点: 数据不安全(外部可以直接修改模块内部的数据)，模块名称会暴露在全局，存在命名冲突，依赖顺序问题。

#### d.自执行匿名函数（闭包）模式

优点：变量、方法全局隐藏，模块私有化

缺点：模块名称会暴露在全局，存在命名冲突，依赖顺序问题。

#### 面临的问题

从以上的尝试中，可以归纳出js模块化需要解决那些问题：

如何安全的包装一个模块的代码？（不污染模块外的任何代码）

如何唯一标识一个模块？

如何优雅的把模块的API暴漏出去？（不能增加全局变量）

如何方便的使用所依赖的模块？

### 3.模块化的规范

#### 1、CommonJS

2009年Nodejs发布，采用 CommonJS 模块规范。

特点：

每个文件都是一个模块实例，代码运行在模块作用域，不会污染全局作用域。

文件内通过require对象引入指定模块，通过exports对象来向往暴漏API，文件内定义的变量、函数，都是私有的，对其他文件不可见。

每个模块加载一次之后就会被缓存。

所有文件加载均是同步完成，加载的顺序，按照其在代码中出现的顺序。

模块输出的是一个值的拷贝，模块内部的变化不会影响该值。

缺点：模块同步加载，资源消耗和等待时间，适用于服务器编程

#### 2、AMD/RequireJS

Commonjs局限性很明显：

基于Node原生api在服务端可以实现模块同步加载，但是仅仅局限于服务端，客户端如果同步加载依赖的话时间消耗非常大，所以需要一个在客户端上基于Commonjs但是对于加载模块做改进的方案，于是AMD规范诞生了。

AMD是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到所有依赖加载完成之后（依赖前置），这个回调函数才会运行。

RequireJS是一个工具库，主要用于客户端的模块管理。它的模块管理遵守AMD规范，RequireJS的基本思想是，通过define方法将代码定义为模块，通过require方法实现代码的模块加载。

特点：浏览器直接运行无需编译，异步加载，依赖关系清晰。

#### 3、CMD/SeaJS

CMD规范专门用于浏览器端，同样是受到Commonjs的启发，国内（阿里）诞生了一个CMD（Common Module Definition）规范。该规范借鉴了Commonjs的规范与AMD规范，在两者基础上做了改进。

与AMD相比非常类似，CMD规范（2011）具有以下特点：

define定义模块，require加载模块，exports暴露变量。

不同于AMD的依赖前置，CMD推崇依赖就近（需要的时候再加载）

推崇api功能单一，一个模块干一件事。

SeaJs是CMD规范的实现，跟RequireJs类似，CMD是SeaJs推广过程中诞生的规范。CMD借鉴了很多AMD和Commonjs优点。

#### AMD、CMD区别

AMD 推崇依赖前置

CMD 推崇依赖就近

### 4.ES6

2015年，ES6规范中，终于将模块化纳入JavaScript标准，从此js模块化被ECMA官方扶正，也是后来js的标准。

ES6中的模块化在CommonJS的基础上有所不同，关键字有import，export，default，as，from。

### CommonJS和ES6区别

CommonJS 模块输出的是一个值的拷贝，即原来模块中的值改变不会影响已经加载的该值。

ES6 模块输出的是值的只读引用，模块内值改变，引用也改变。

CommonJS 模块是运行时加载，加载的是整个模块，即将所有的接口全部加载进来。

ES6 模块是编译时输出接口，可以单独加载其中的某个接口。

## 前端模块化commonJS,CMD,AMD,UMD,es6 module

### 总结

CommonJS规范主要用于服务端编程，加载模块是同步的，不适合在浏览器环境，存在阻塞加载，浏览器资源是异步加载的，因此有了AMD、CMD解决方案。

AMD规范在浏览器环境中异步加载模块，而且可以并行加载多个模块。

CMD规范与AMD规范很相似，都用于浏览器编程，依赖就近，代码更简单。

ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

### 1、Commonjs

用于nodejs环境，它使用 require 引用和加载模块，exports 定义和导出模块，module 标识模块。有人试图将其引入浏览器环境，由于node服务器环境上文件总是位于服务器上，加载速度较快，因此使用同步加载即可；但浏览器需要从服务端下载文件，所以说如果用CommonJS的require方式加载模块，需要等代码模块下载完毕，并运行之后才能得到所需要的API。如果我们在某个代码模块里使用CommonJS的方法require了一个模块，而这个模块需要通过http请求从服务器去取，如果网速很慢，而CommonJS又是同步的，所以将阻塞后面代码的执行，从而阻塞浏览器渲染页面，使得页面出现假死状态。因此产生了异步加载，主要是AMD和CMD

* CommonJS 是一种用于服务器端的模块化规范，最初由Node.js采用并推广。
* 使用require()函数导入模块，使用module.exports导出模块。
* 同步加载模块，适用于服务器端的模块加载。
* 不能在浏览器端直接使用，需要通过工具转换成浏览器可用的代码。

### 2、AMD

用于浏览器环境，依赖前置，提前执行。主要有requirejs，它主要有两个接口：define 和 require。define 是模块开发者关注的方法，而 require 则是模块使用者关注的方法。

优点：1、首先，尽早执行依赖可以尽早发现错误。

2、尽早执行依赖通常可以带来更好的用户体验。

缺点：1、容易产生浪费。有些模块加载后未使用或者使用较晚

1. 开发成本高，代码的阅读和书写比较困难，模块定义方式的语义不顺畅；不符合通用的模块化思维方式，是一种妥协的实现。

* AMD 是一种用于浏览器端的模块化规范，最著名的实现是RequireJS。
* 使用define()函数定义模块，使用require()函数异步加载模块。
* 异步加载模块，适用于浏览器端的模块加载，可以提高页面加载速度。
* 适用于浏览器端的异步加载，可以并行加载多个模块。

### 3、CMD：

用于浏览器环境，依赖就近，延迟执行。主要实现为sea.js

CMD是SeaJS在推广过程中生产的对模块定义的规范，在Web浏览器端的模块加载器中，SeaJS与RequireJS并称，SeaJS作者为阿里的玉伯。CMD规范专门用于浏览器端，模块的加载是异步的，模块使用时才会加载执行。CMD规范整合了CommonJS和AMD规范的特点。在 Sea.js 中，所有 JavaScript 模块都遵循 CMD模块定义规范。

优点：依赖就近，延迟执行 可以很容易在 Node.js 中运行；

缺点：依赖 SPM 打包，模块的加载逻辑偏重

* CMD 也是一种用于浏览器端的模块化规范，最著名的实现是SeaJS。
* 使用define()函数定义模块，使用require()函数加载模块。
* 与AMD类似，但CMD更加倾向于依赖就近，需要使用时再加载。
* 适用于浏览器端的模块加载，更加符合国内开发者的使用习惯。

### 4、UMD

nodejs和浏览器环境都支持，是common和AMD的统一，能够兼容他们，这样我们在开发时，就不需要再去考虑依赖模块所遵循的规范了，UMD的出现就是为了解决这个问题。(没有兼容CMD吧)

* UMD是一种兼容多种模块规范的解决方案，可以在CommonJS、AMD和全局变量环境下使用。
* 通过判断当前环境，选择使用CommonJS、AMD或全局变量的方式导出模块。

### 5、ES6 Module

设计思想是尽量的静态化，使得编译时就能确定模块的依赖关系，以及导入和导出的变量，也就是所谓的"编译时加载"。主要语法是import和export

* ES6 模块是 ECMAScript 6 标准中新增的模块化规范，已成为 JavaScript 的官方模块化标准。
* 使用import关键字导入模块，使用export关键字导出模块。
* 支持静态分析，可以进行更好的优化。
* 可以在浏览器端和服务器端直接使用，不需要额外的转换工具。

### 6、对比

common引入的是一个拷贝，而es6module引入的是引用，可以实时变化，比如文件中有一个settimeout，延时后再调用可获得最新值，但是commonjs不能获取最新值。

commonjs或AMD，都是运行时加载，在编译阶段只判断语法层面对不对，如果引入了不存在的变量等只有在执行时才会发现。

ES6 可以因为是一边编译一边执行，在编译阶段会检查引入变量是否存在、引入模块是否存在等，问题更早暴露。

### 编译时和运行时知识点扩展

编译时

编译：将源代码翻译成机器能识别的语言（二进制）。计算机还是那个计算机，他至今还是只能看懂01,但随着时间的推移和人类的进步，逐渐出现了很多高级语言，高级语言之所以高级就在于我们可以用一种简单的方式比如if elese 来实现我们的代码逻辑，但最终在执行的时候计算机还是没办法识别的，这个时候就需要一个处理过程"编译"，将大家平时写的if else 翻译成机器可以看懂的语言。那负责编译的这部分，我们一般称之为 编译器。

那编译时就会做一些简单的翻译工作,比如检查你有没有粗心写错啥关键字了啊。会进行词法分析,语法分析之类的。就像个老师检查学生的作文中有没有错别字和病句一样.如果发现啥错误编译器就告诉你。开始编译时,如果有errors或者warning信息,都是编译器检查出来的。所谓这时的错误就叫编译时错误,这个过程中做的类型检查也就叫编译时类型检查,或静态类型检查(所谓静态嘛就是没把真把代码放内存中运行起来,而只是把代码当作文本来扫描下)。

运行时

所谓运行时就是代码跑起来了。被装载到内存中去了。(你的代码保存在磁盘上没装入内存之前是个文件。只有跑到内存中才变成活的)。而运行时类型检查就与前面讲的编译时类型检查(或者静态类型检查)不一样。不是简单的扫描代码.而是在内存中做些操作,做些判断以确定我们的程序是否存在错误。

## AST抽象语法树

概念

AST 全称为 Abstract Syntax Tree，译为抽象语法树。在 JavaScript 中，任何一个对象（变量、函数、表达式等）都可以转化为一个抽象语法树的形式。抽象语法树本质就是一个树形结构的对象。

用途

* 常用各类转义、编译的插件中。比如最典型的 ES6 转换为 ES6 工具 、JSX 语法转换为 JavaScript 语法。即 babel 模块。
* 代码语法的检查，比如代码规范工具 ESLint 模块。
* 各类 JS/CSS/HTML 压缩工具。
* 代码的格式化、高亮。
* 代码错误提示。
* 代码自动补全。