JS执行上下文：JS执行代码的时候的一个运行环境，或者说是一个作用域。

三种不同运行环境：

全局级别：默认的代码运行环境，一旦代码被载入，引擎最先进入

函数级别：当执行一个函数时，运行函数体中的代码。（无限调用函数就有无限多个）

Eval级别：在Eval函数内运行的代码。

执行代码前JS所做的准备工作：

1、变量、**函数表达式**——变量声明，默认赋值为undefined

2、this——赋值

3、函数声明——赋值

在执行代码之前，把将要用到的所有的变量都事先拿出来，有的直接赋值，有点先用undefined点个空

全局作用域的上下文准备工作

将全局变量设为undefined

将函数表达式的值设为undefined

为函数声明直接赋值

将window对象赋给this

函数作用域的上下文准备工作

确定自由变量的作用域

为函数的参数和arguments对象赋值

将局部变量的值设为undifined

将函数表达式的值设为undifined

为函数声明直接赋值

JS执行代码的时候，会进入不同的执行上下文，这些不同的执行上下文就构成了一个执行上下文栈。

在浏览器中，JS引擎的工作方式是单线程的，在某一时刻只有唯一的一个事件是被激活处理的，其他事件被放入队列中，等待被处理。

综合上面所述，形成一种压栈方式运行执行上下文。

|  |  |
| --- | --- |
| 正在运行 | 被调用的函数1的代码 |
|  | 被调用的函数2的代码 |
|  | 被调用的函数3的代码 |
| 总是在最底层 | 全局 |

var a = 10,

fn,

bar = function(x){

var b = 5;

fn(x+b);

};

fn = function(y){

var c = 3;

console.log(y+c);

}

//定位AA

bar(10);

执行代码前，全局上下文

|  |  |
| --- | --- |
| 全局上下文 | |
| a | undefined |
| fn | undefined |
| bar | undefined |
| this | window |

向下执行代码，执行到AA前，上下文环境中的变量发生变化

|  |  |
| --- | --- |
| 全局上下文 | |
| a | 10 |
| fn | function |
| bar | function |
| This | window |

向下执行代码，调用bar函数

跳转到bar函数内部，执行代码之前，会创建一个新的执行上下文环境

|  |  |
| --- | --- |
| bar函数－执行上下文 | |
| b | undefined |
| x | 10 |
| arguments | [10] |
| This | window |

继续执行代码

|  |  |
| --- | --- |
| Bar函数－执行上下文 | |
| b | 5 |
| x | 10 |
| fn | 调用 |
| this | window |

调用fn函数，跳转到fn函数内部，执行代码之前，会创建一个新的执行上下文环境

|  |  |
| --- | --- |
| fnr函数－执行上下文 | |
| c | undefined |
| y | 15 |
| this | ?? |

继续执行代码

|  |  |
| --- | --- |
| fn函数－执行上下文 | |
| c | 5 |
| y | 15 |
| this | window |

如下压栈方式执行，红色为当前活动状态

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 全局上下文环境 | bar上下文环境 | fn上下文环境 | bar上下文环境 | 全局上下文环境 |
| bar上下文环境 |
| 全局上下文环境 | 全局上下文环境 | 全局上下文环境 |

创建执行上下文的过程

每当调用一个函数时，一个新的执行上下文就会被创建出来，步骤分两阶段

1、建立阶段（发生在调用函数时，但是在没有执行函数体内的代码之前）

建立变量、函数、arguments对象、参数

建立作用域链

确定this的值

2、代码执行阶段

变量赋值、函数引用、执行其他代码

console.log(a);

//报错，变量未定义

var a = 10;

console.log(a);

//正常

var a;

console.log(a); //undefined

console.log(a);＝＝右内容

var a = 10;

//undefined

函数先被声明并赋值了,指向了函数

var a;

a = function a(){

this.user = "pschen";

}

console.log(a);

function a(){

this.user = “pschen”;

}

//可以事先打印出函数主体

a();

var a = function(){

通常把这种函数赋值给变量的形式叫做函数表达式

函数声明时的赋值行为是在函数创建的时候进行

函数表达式的赋值行为是在执行赋值给变量的语句时进行

console.log(“”);

}

//undefined

var a = 1;

function b(){

var a

console.log(a); //1

a = 5;

}

b();

var a = 1;

function b(){

console.log(a); //1

a = 5;

}

b();

var a = 1;

function b() {

var a;

a = function a(){}

console.log(typeof a);

a = 120;

console.log(typeof a);

return;

function a() {}

}

b();

console.log (a);

执行之前预编译，先声明函数并赋值，就把a变成了函数内部变量

var a = 1;

function b() {

a = 120;

return;

function a() {}

}

b();

console.log (a); //1

最后输出为1；函数b内部的变量，在外部无法直接访问，最后取全局变量

变量的赋值是指向还是其他，要看他的数据类型是什么

原始类型：代表不同的值

引用类型：就是指向 不同的变量名指向相同的值

同名变量的字符串是直接复制一份，变量代表不同的值

var a = 1;

function b() {

a = 120; **//声明变量没有使用var，a还是一个全局变量**

return;

}

b();

console.log(a); //120