

一、this (2020.5.15)

阅读下面代码，分析执行结果，并说明具体原因。

```
var num = 20;

const obj = {
  num: 10,
  func: (num) => {
    this.num += 5;
    console.log(this.num);

    num += 5;
    console.log(num);
    var num = 30;

    return function () {
      this.num += 4;
      console.log(this.num);

      num += 10;
      console.log(num);
    };
  },
};

obj.func(40)();
```

答案

- 浏览器环境下答案是 25 45 29 40
- node 环境下的答案是 NaN 45 NaN 40

解析

一、浏览器环境

在浏览器下，obj.func 它是一个箭头函数，箭头函数不会影响内部的 this 指向，执行的时候内部的 this 指向的必然是全局对象也就是 window，所以里面的 this.num 其实就是 window 下的 num，所以结果是 25。

obj.func 执行的时候，带入了实参 num=40，函数内的 var num 虽然有变量提升，但是提升的只是声明，在它声明之前，形参内已经声明过了 num，所以这个声明就不会生效，那 num 当中自然保留实参传过来的 40，所以打印出的 num 是 45（这个点是很多人容易出错的地方，尤其需要注意）

obj.func 执行后的返回值是个匿名函数，这个函数内的 this 跟它的执行体有关，这个函数的执行体依然是 window，所以里面的 this.num 还是 window 下的 num，打印结果是 29

obj.func 的返回值匿名函数内没有声明变量 num，所以向它的父级作用域查找，此时父级作用域下的 num 被赋值为 30，所以打印结果是 40

二、Node.js 环境

node 环境下，全局环境中 this 指向的是一个空对象 {}，obj.func 函数再执行的时候，因为它是箭头函数，this 指向 {}，里面没有 num 属性，跟数字相加结果是 NaN

obj.func 执行的时候，带入了实参 num=40，函数内的 var num 虽然有变量提升，但是提升的只是声明，在它声明之前，形参内已经声明过了 num，所以这个声明就不会生效，那 num 当中自然保留实参传过来的 40，所以打印出的 num 是 45（这个点是很多人容易出错的地方，尤其需要注意）

obj.func 执行后的返回值是个匿名函数，这个函数内的 this 跟它的执行体有关，这个函数内的 this 依然是 {}，所以里面的 this.num 还是不存在的，打印结果是 NaN

obj.func 的返回值匿名函数内没有声明变量 num，所以向它的父级作用域查找，此时父级作用域下的 num 被赋值为 30，所以打印结果是 40

全局作用域中 this } web: 指代 window
node: 指代 global / GLOBAL
var 声明变量 } web: 挂在 window 上
node: 挂在 global 上

PS: 1) 在模块中使用 var, this.module.export
2) 在模块中直接声明, 挂在 global 上

此外判断需要查看函数调用环境

非全局型, 勿忘闭包

二 阻塞.被修(2020.5.17)

阅读下面代码，我们只考虑浏览器环境下的输出结果，写出它们结果打印的先后顺序，并分析出原因，小伙伴们，加油哦！

```
1 console.log("AAAA");
2 setTimeout(() => console.log("BBBB"), 1000);
3 const start = new Date();
4 while (new Date() - start < 3000) {}
5 console.log("CCCC");
6 setTimeout(() => console.log("DDDD"), 0);
7 new Promise((resolve, reject) => {
8   console.log("EEEE");
9   foo.bar(100);
10 })
11 .then(() => console.log("FFFF"))
12 .then(() => console.log("GGGG"))
13 .catch(() => console.log("HHHH"));
14 console.log("IIII");
```

此处同步代码执行了3秒

答案：

浏览器下 输出结果的先后顺序是

```
AAAA
CCCC
EEEE
IIII
HHHH
BBBB
DDDD
```

答案解析：这道题考察重点是 js异步执行 宏任务 微任务。

一开始代码执行，输出 AAAA。1

第二行代码开启一个计时器t1(一个称呼)，这是一个异步任务且是宏任务，需要等到1秒后提交。

第四行是个while语句，需要等待3秒后才能执行下面的代码。这里有个问题，就是3秒后上一个计时器t1的提交时间已经过了，但是线程上的任务还没有执行结束，所以暂时不能打印结果，所以它排在宏任务的最前面了。

第五行又输出 cccc

第六行又开启一个计时器t2(称呼)，它提交的时间是0秒(其实每个浏览器有默认最小时间的，暂时忽略)，但是之前的t1任务还没有执行，还在等待，所以t2就排在t1的后面。(t2排在t1后面的原因是while造成的)都还需要等待，因为线程上的任务还没执行完毕。

第七行 new Promise 将执行promise函数，它参数是一个回调函数，这个回调函数内的代码是同步的，它的异步核心在于resolve和reject，同时这个异步任务在任务队列中属于微任务，是优先于宏任务执行的，(不管宏任务有多急，反正我是VIP)。所以先直接打印输出同步代码EEEE。第九行中的代码是个不存在的对象，这个错误要抛给reject这个状态，也就是catch去处理，但是它是异步的且是微任务，只有等到线程上的任务执行完毕，立马执行它，不管宏任务(计时器，ajax等)等待多久了。

第十四行，这是线程上的最后一个任务，打印输出 IIII

我们先找出线程上的同步代码，将结果依次排列出来：AAAA CCCC EEEE IIII

然后我们再找出所有异步任务中的微任务 把结果打印出来 HHHH

最后我们再找出异步中的所有宏任务，这里t1排在前面t2排在后面(这个原因是while造成的)，输出结果顺序是 BBBB DDDD

所以综上 结果是 AAAA CCCC EEEE IIII HHHH BBBB DDDD

三 异步与微任务 (2020.5.18)

20200518面试题

阅读下面代码，我们只考虑浏览器环境下的输出结果，写出它们结果打印的先后顺序，并分析出原因，小伙伴们，加油哦！

```
1 async function async1() {  
2   console.log("AAAA");  
3   async2();  
4   console.log("BBBB");  
5 }  
6 async function async2() {  
7   console.log("CCCC");  
8 }  
9 console.log("DDDD");  
10 setTimeout(function () {  
11   console.log("FFFF");  
12 }, 0);  
13 async1();  
14 new Promise(function (resolve) {  
15   console.log("GGGG");  
16   resolve();  
17 }).then(function () {  
18   console.log("HHHH");  
19 });  
20 console.log("IIII");
```

答案：

浏览器下 输出结果的先后顺序是

```
DDDD  
AAAA  
CCCC  
BBBB  
GGGG  
IIII  
HHHH  
FFFF
```

答案解析：这道题考察重点是 js 异步执行 宏任务 微任务。

这道题的坑就在于 async 中如果没有 await，那么它就是一个纯同步函数。

这道题的起始代码在第9行，输出 DDDD

第10行计时器开启一个异步任务 t1（一个称呼），这个任务且为宏任务。

第13行函数 async1 执行，这个函数内没有 await 所以它其实就是一个纯同步函数，打印输出 AAAA。

在 async1 中执行 async2 函数，因为 async2 的内部也没有 await，所以它也是个纯同步函数，打印输出 CCCC

紧接着打印输出 BBBB。

第14行 new Promise 执行里面的代码也是同步的，所以打印输出 GGGG，resolve() 调用的时候开启一个异步任务 t2（一个称呼），且这个任务 t2 是微任务，它的执行交给 then() 中的第一个回调函数执行，且优先级高于宏任务（t1）执行。

第20行打印输出 IIII，此时线程上的同步任务全部执行结束。

在执行任务队列中的异步任务时，微任务优先于宏任务执行，所以先执行微任务 t2 打印输出 HHHH，然后执行宏任务 t1 打印输出 FFFF

所以综上 结果输出是 DDDD AAAA CCCC BBBB GGGG IIII HHHH FFFF

被 async 修饰的函数被调用时：

1 函数内语句同步执行，直到遇到 await

2 遇到 await 后，函数内：暂停

函数外：执行下一个同步代码

3 返回值被 Promise 自动包裹

函数内：有 await 时，延迟返回

函数外：无 await 时，直接返回

三. Async/Await (2020.5.19)

1. 问题1

```
async function t1() {  
  let a = await "lagou";  
  console.log(a);  
}  
t1()
```

1) 当 await 后接非 promise 时, 直接返回代码块结果

2. 问题2

```
async function t2() {  
  let a = await new Promise((resolve)  
=> {});  
  console.log(a);  
}  
t2()
```

2) 当 await 后 promise 未决议时, 后续代码会被挂起

3. 问题3

```
async function t3() {  
  let a = await new Promise((resolve)  
=> {  
    resolve();  
  });  
  console.log(a);  
}  
t3()
```

4. 问题4

```
async function t4() {  
  let a = await new Promise((resolve)  
=> {  
    resolve("hello");  
  });  
  console.log(a);  
}  
t4()
```

5. 问题5

```
async function t5() {  
  let a = await new Promise((resolve)  
=> {  
    resolve("hello");  
  }).then(() => {  
    return "lala";  
  });  
  console.log(a);  
}  
t5()
```

5) 当 await 后接 promise 链时, 会一直等到调用链结束, 使用最后一次结果

6. 问题6

```
async function t6() {  
  let a = await fn().then((res) => {  
    return res;  
  });  
  console.log(a);  
}  
async function fn() {  
  await new Promise((resolve) => {  
    resolve("lagou")  
  });  
}  
t6()
```

6) 被 async 修饰的函数, 返回函数会被 promise 自动包装

7. 问题7

```
async function t7() {  
  let a = await fn().then((res) => {  
    return res;  
  });  
  console.log(a);  
}  
async function fn() {  
  await new Promise((resolve) => {  
    resolve("lagou")  
  });  
  return "lala"  
}  
t7()
```

7) 所有子函数后调用均未使用 await, 区别只在于是否会阻塞当前调用环境, 其内部执行并不受影响。