bytefish Follow

Sep 21, 2021 · 4 min read · · Listen

Save

JAVASCRIPT INTERVIEW QUESTIONS

The Execution Order of Asynchronous Functions in the Event Loop

Interview Question

What is the output of the following code?

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

console.log('script start');

setTimeout(function() {
    console.log('setTimeout');
}, 0)

async1();

new Promise(function(resolve) {
    console.log('promise1');
    resolve();
}).then(function() {
    console.log('promise2');
```

Analysis

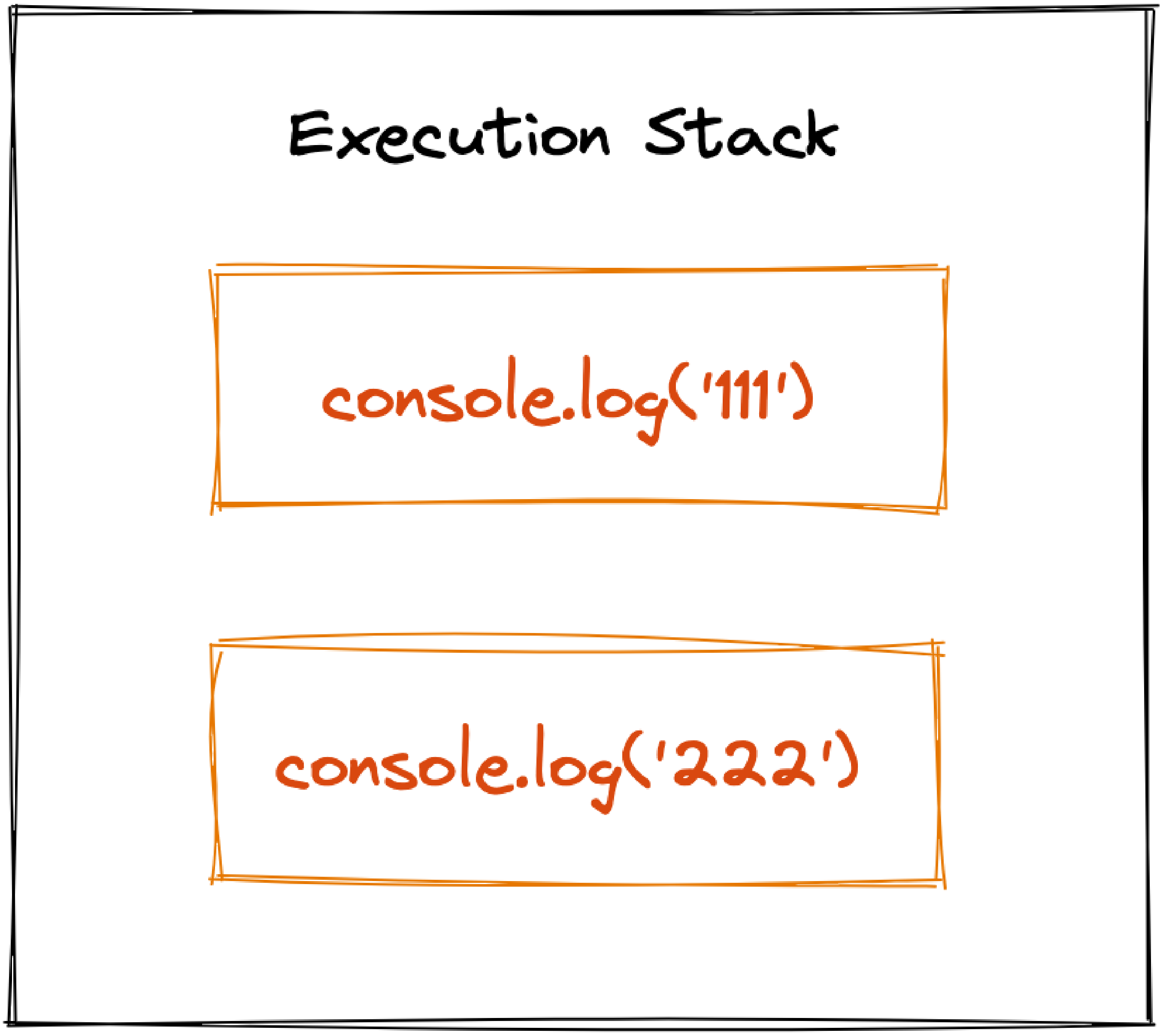
This question is actually examining the execution order of the code. To be precise, it is examining the difference between asynchronous functions such as setTimeout, Promise, and async/await.

Let’s look at the simplest case first:

```
console.log('111')

console.log('222')
```

What is the execution result of this code? Anyone who has studied programming knows that the program will output `111` and `222` in sequence. There is no asynchronous code in this code, so it is very simple.

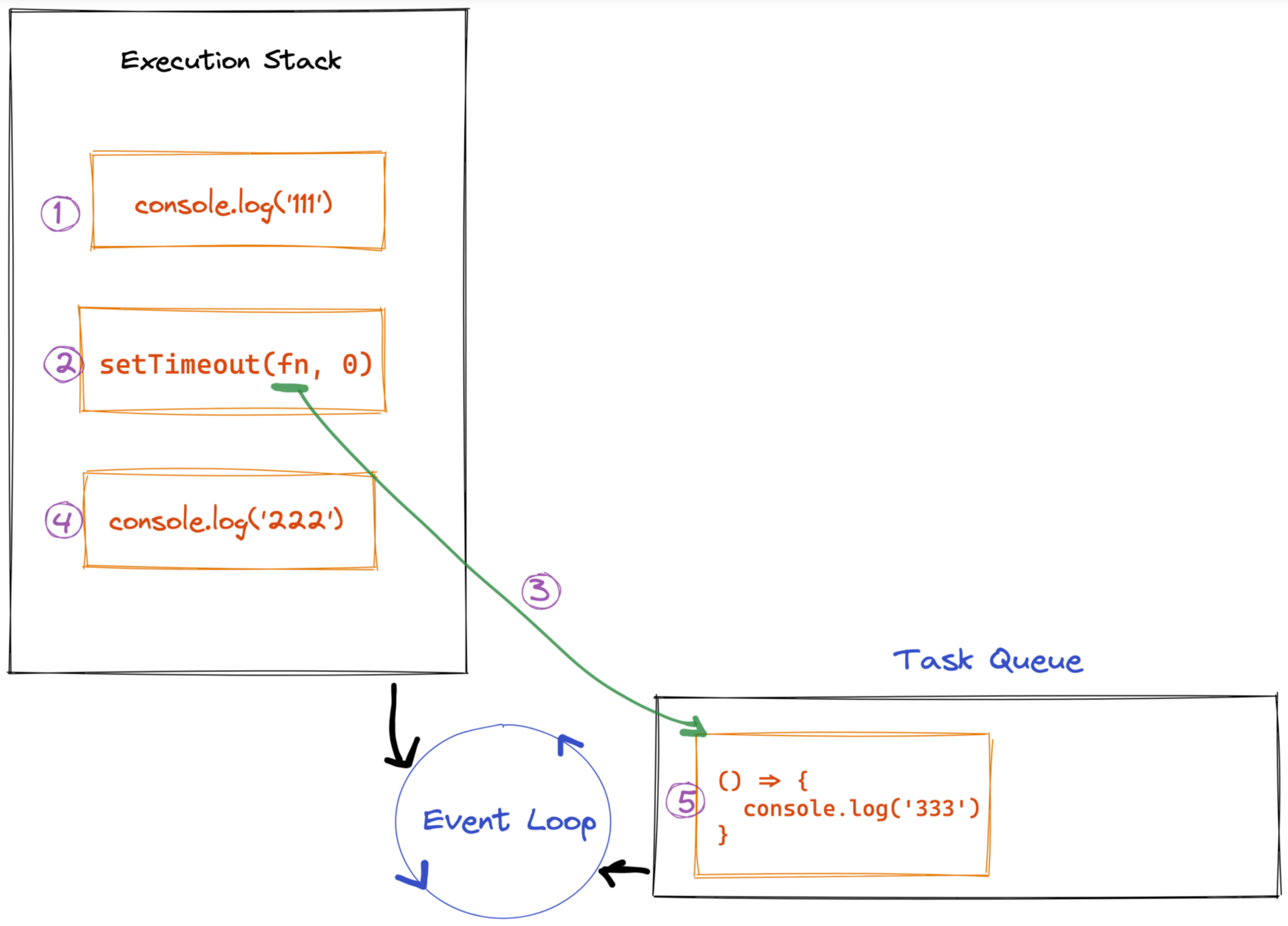


Let's look at another example:

```
console.log('111')

setTimeout(() => {
  console.log('333')
}, 0)

console.log('222')
```



We know that there is a task queue in JavaScript, and all asynchronous callback functions will be put into this task queue first, and then executed when the main thread is free.

In the code above:

- `console.log('111')` will be executed first, so the console will print 111
- Then execute `setTimeout(..., 0)` . Note that in this step, the JS engine executes only the function `setTimeout` , and its callback function will not be executed. Then `setTimeout` will create a timer, the time of this timer is 0, so its callback function will be immediately pushed into the task queue
- Then execute `console.log('222')` , so the console will print 222
- At this time, the current execution stack is free, and the JS engine starts to check the task queue.

There is a little difference between this code and the previous code, that is, Promise and `setTimeout` appear at the same time. In this case, we need to introduce a new concept, that is, microtasks and macrotasks.

I mentioned the task queue before, but this is actually an inaccurate statement. There are actually two different task queues in JavaScript, namely microtask queue and macrotask queue.

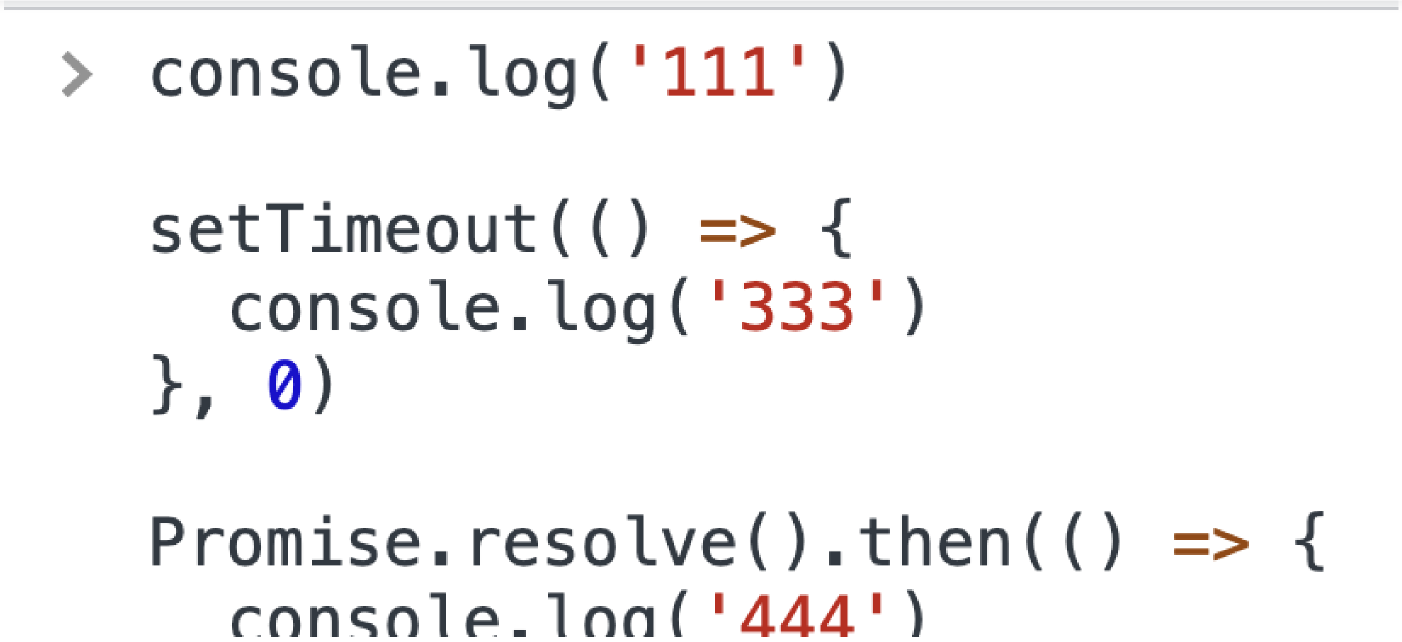
Here is a list of macrotasks and microtasks:

With this understanding, we can draw a more accurate flow chart:

Get unlimited access Open in app

The sequence number in the figure above indicates the order of code execution.

Here is the result:


```
> console.log('111')

  setTimeout(() => {
    console.log('333')
  }, 0)

  Promise.resolve().then(() => {
    console.log('444')
```

Finally, let’s talk about asnyc/await. How to understand this feature? In fact, async/await is syntactic sugar for Promise.

The following two pieces of code are actually equivalent:

```
async function async1() {
```

Therefore, when we are considering the execution flow of the async function, it is enough to convert it into a Promise.

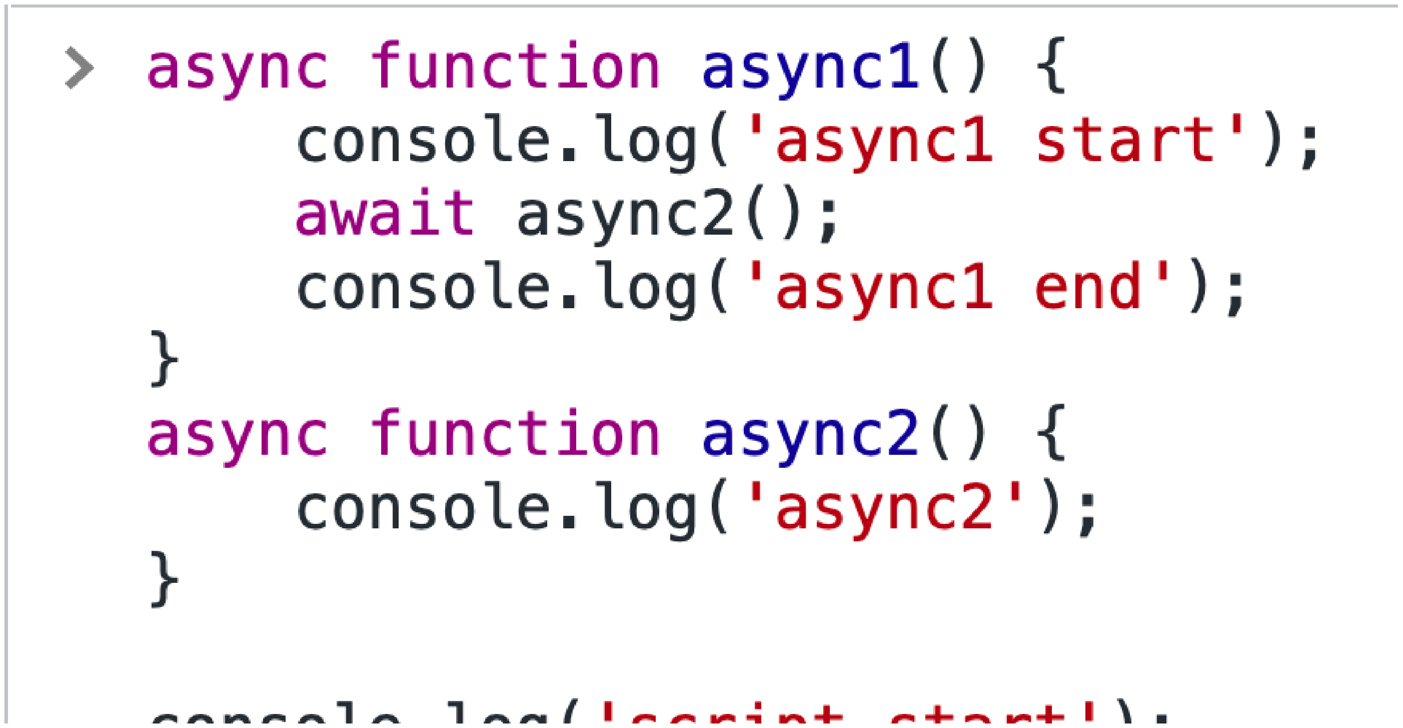
Okay, after explaining these theoretical knowledge, do you know the answer to the interview question at the beginning?

Answer

Let’s go back to this interview question from the beginning. Now you can think about: in this code, which statements are executed synchronously, which statements will be added to the micro task queue, and which statements will be added to the macro task queue?

Here I give an answer. The statements that will be executed synchronously are wrapped in pink boxes, and the statements that will be added to the micro task queue are wrapped in blue boxes, and they will be added to the macro task queue are wrapped in green boxes.

Here is the execution result of the code:



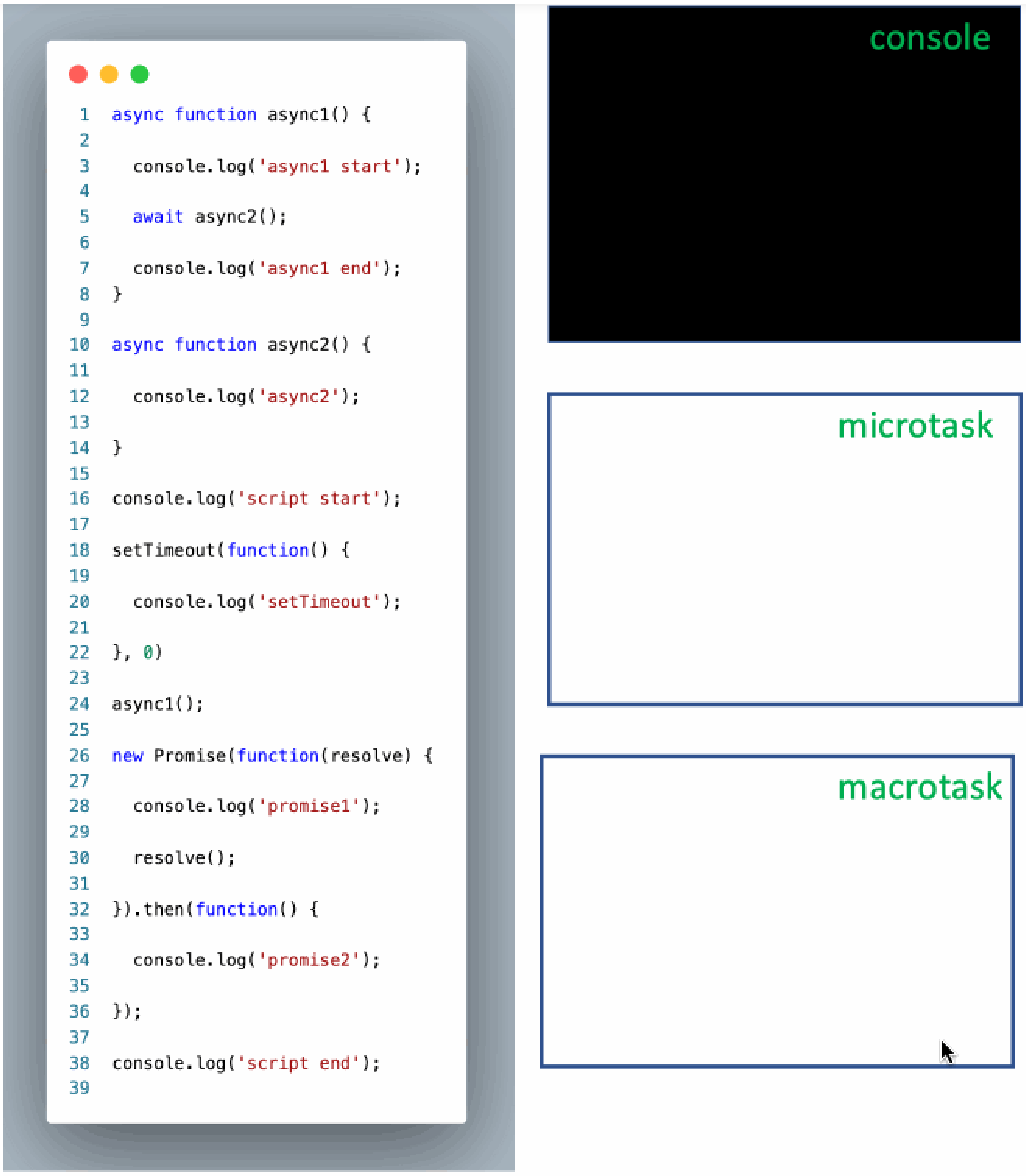
```
async function async1() {
        console.log('async1 start');
        await async2();
        console.log('async1 end');
}
async function async2() {
        console.log('async2');
}

console.log('script start');
```

Animation

In order to help you better understand the execution flow of the above program, I made the following animation. In this animation, the red box represents the code currently being executed.

Get unlimited access Open in app



```
async function async1() {

  console.log('async1 start');

  await async2();

  console.log('async1 end');
}

async function async2() {

  console.log('async2');

}

console.log('script start');

setTimeout(function() {

  console.log('setTimeout');

}, 0)

async1();

new Promise(function(resolve) {

  console.log('promise1');

  resolve();

}).then(function() {

  console.log('promise2');

});

console.log('script end');
```