bytefish Follow

Apr 29, 2020 · 11 min read ★ · Listen

6 Interview Questions That Combine Promise and setTimeout

Thoroughly master this type of interview question.

Photo by Bharat Patil on Unsplash

- There are two task queues in the JS engine: macrotask queue and microtask queue
- The entire script is initially executed as a macro task
- During the execution, the synchronization code executes directly, and the macrotask enters the macro task queue, and the microtask enters the microtask queue
- When the current macro task is completed, the microtask queue is checked and all the microtasks are executed in turn
- Performs rendering of the browser UI thread (You can ignore this in this article)
- If any Web Worker task exists, execute it (You can ignore this in this article)
- Check the macro task queue, and if it is not empty, go back to step 2 and execute the next macro task.

It is worth noting that step 4: when a macrotask is completed, all other microtasks are executed in turn first, and then the next macrotask is executed.

Mircotasks include: `MutationObserver`, `Promise.then()` and `Promise.catch()`, other techniques based on Promise such as the fetch API, V8 garbage collection process, `process.nextTick()` in node environment.

Marcotasks include: initial script, `setTimeout`, `setInterval`, `setImmediate`, I/O, UI rendering.

Well, it doesn't matter if you don't fully understand what's going on here, let's practice with examples.

There are 10 questions: the first 4 are simple Promise questions to help you understand microtasks; the next 6 questions are a mix of Promise and setTimeout.

1.

Let's start with a simple example to explain the microtask.


```
  resolve('success')
});

promise1.then(() => {
  console.log(3);
});

console.log(4);
```

Process Analysis:

- First, the first four lines of this code are executed. The console will print out `1`, and then `promise1` will turn into a `resolved` state.
- Then start executing the `promise1.then(() => {console.log(3);});` snippet. Because `promise1` is now in the resolved state, the `() => {console.log(3);}` will be added to the microtask queue immediately.
- But we know `() => {console.log(3);}` is a microtask, so it is not immediately called.
- Then the last line of code(`console.log(4);`) is executed and `4` is printed in the console.
- At this point, all the synchronized code, the current macrotask, is executed. Then the JavaScript engine checks the queue of microtasks and executes them in turn.
- `() => {console.log(3);}` is then executed and `4` is printed in the console.

Result:


```
> const promise1 = new Promise((resolve, reject) => {
    console.log(1);
    resolve('success')
  });
  promise1.then(() => {
    console.log(3);
  });
  console.log(4);
  1
  4
  3
```

2.

Example:

```
const promise1 = new Promise((resolve, reject) => {
  console.log(1);
});

promise1.then(() => {
  console.log(3);
});

console.log(4);
```

Process Analysis:

This example is very similar to the previous one, except that in this one, `promise1` will always be in a `pending` state, so `() => {console.log(3);}` won't be executed and the console won't print 3.

Result:


```
> const promise1 = new Promise((resolve, reject) => {
    console.log(1);
  });
  promise1.then(() => {
    console.log(3);
  });
  console.log(4);
  1
  4
```

3.

Example:

```
const promise1 = new Promise((resolve, reject) => {
  console.log(1)
  resolve('resolve1')
})

const promise2 = promise1.then(res => {
  console.log(res)
})

console.log('promise1:', promise1);
console.log('promise2:', promise2);
```

Consider carefully the order in which the console prints the results and the state of each promise.

Process Analysis:

- First, the first four lines of code are the same as before, `1` is printed in the console, and the state of `promise1` is `resolved` .
- Then execute `const promise2 = promise1.then(...)` , `res => {console.log(res)}` is added to the microtask queue. At the same time, `promise1.then()` will return a new `pending` promise object.

- Then execute `console.log('promise2:', promise2);` , and the console prints out the string `'promise2'` and the `promise2` in the pending state.
- At this point, all the synchronized code, the current macrotask, is executed. Then the JavaScript engine checks the queue of microtasks and executes them in turn.
- `res => {console.log(res)}` is the only task in the microtask queue, and it will be executed now. And then the console will print `'reslove1'` .

Result:

```
> const promise1 = new Promise((resolve, reject) => {
    console.log(1)
    resolve('resolve1')
  })
  const promise2 = promise1.then(res => {
    console.log(res)
  })
  console.log('promise1:', promise1);
  console.log('promise2:', promise2);
  1
  promise1: ▶ Promise {<resolved>: "resolve1"}
  promise2: ▶ Promise {<pending>}
  resolve1
```

4.

Example:

```
const fn = () => (new Promise((resolve, reject) => {
  console.log(1)
  resolve('success')
}));
```

Process Analysis:

Unlike before, in this example, the behavior of creating a Promise object occurs within the `fn` function. While the `fn` function is a normal synchronization function, there is nothing special about it, and this example is still simple.

Result:

```
> const fn = () => (new Promise((resolve, reject) => {
    console.log(1)
    resolve('success')
  }));
  fn().then(res => {
    console.log(res)
  });
  console.log(2)
  1
  2
  success
```

The previous examples are relatively simple, now the questions will gradually become more complex, are you ready?

5.

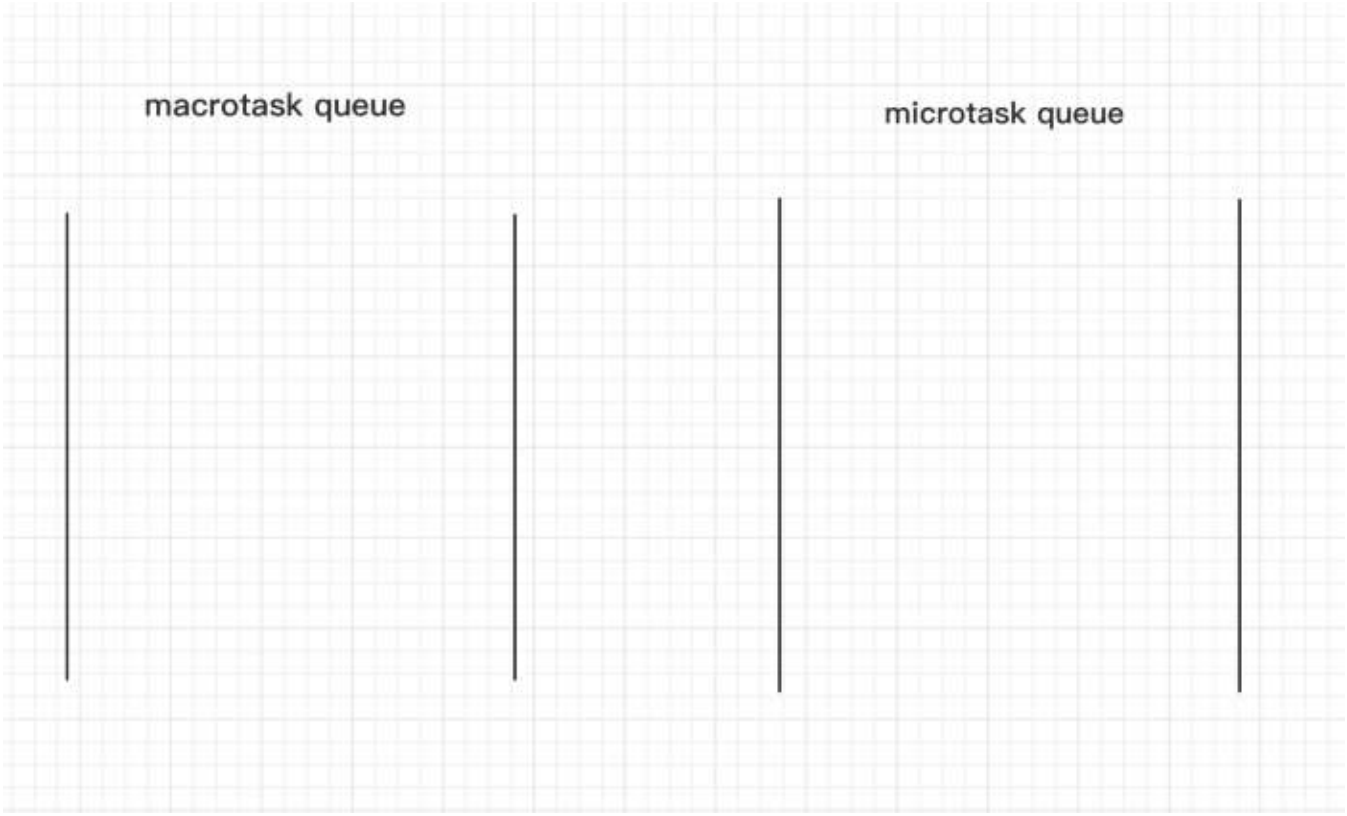
Example:

```
1  console.log('start')
2  setTimeout(() => {
3    console.log('setTimeout')
4  })
5  Promise.resolve().then(() => {
6    console.log('resolve')
7  })
8  console.log('end')
```

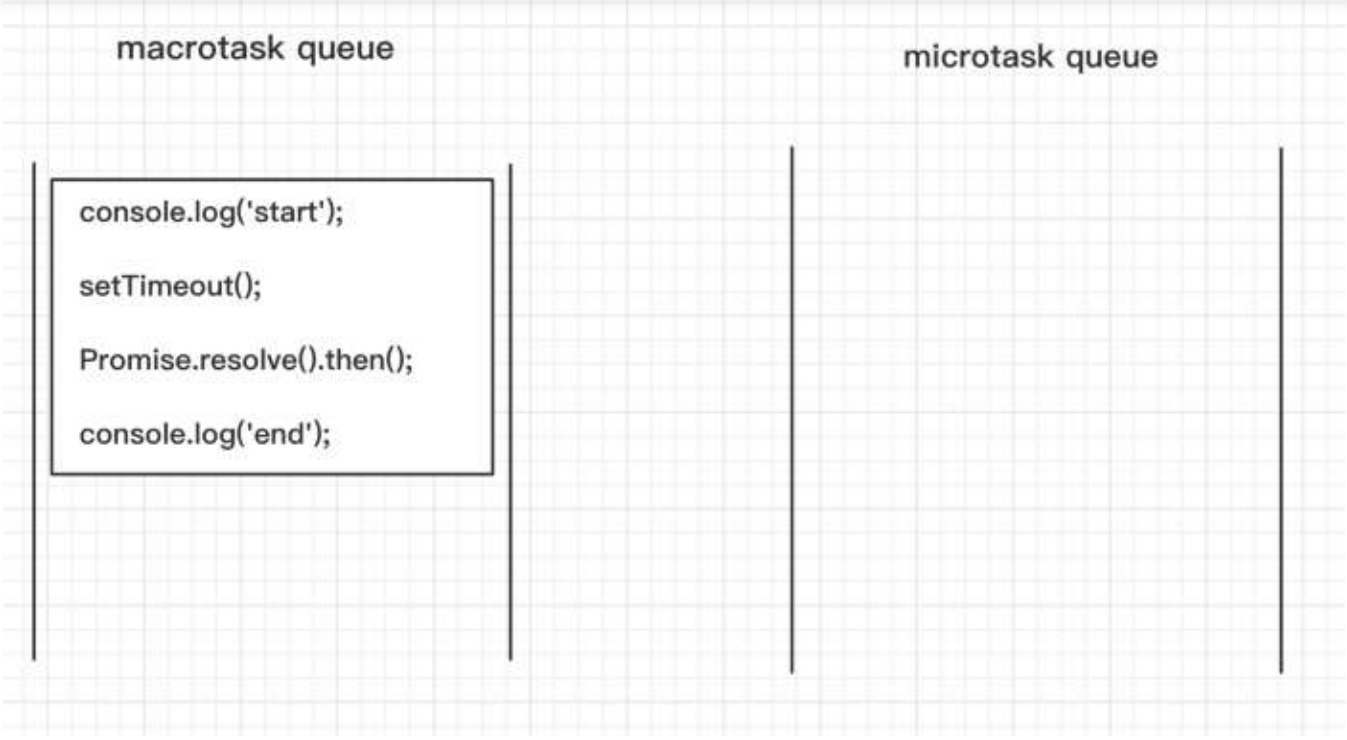
```
setTimeout(() => {
  console.log('setTimeout')
})

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

console.log('end')
```

Process Analysis:

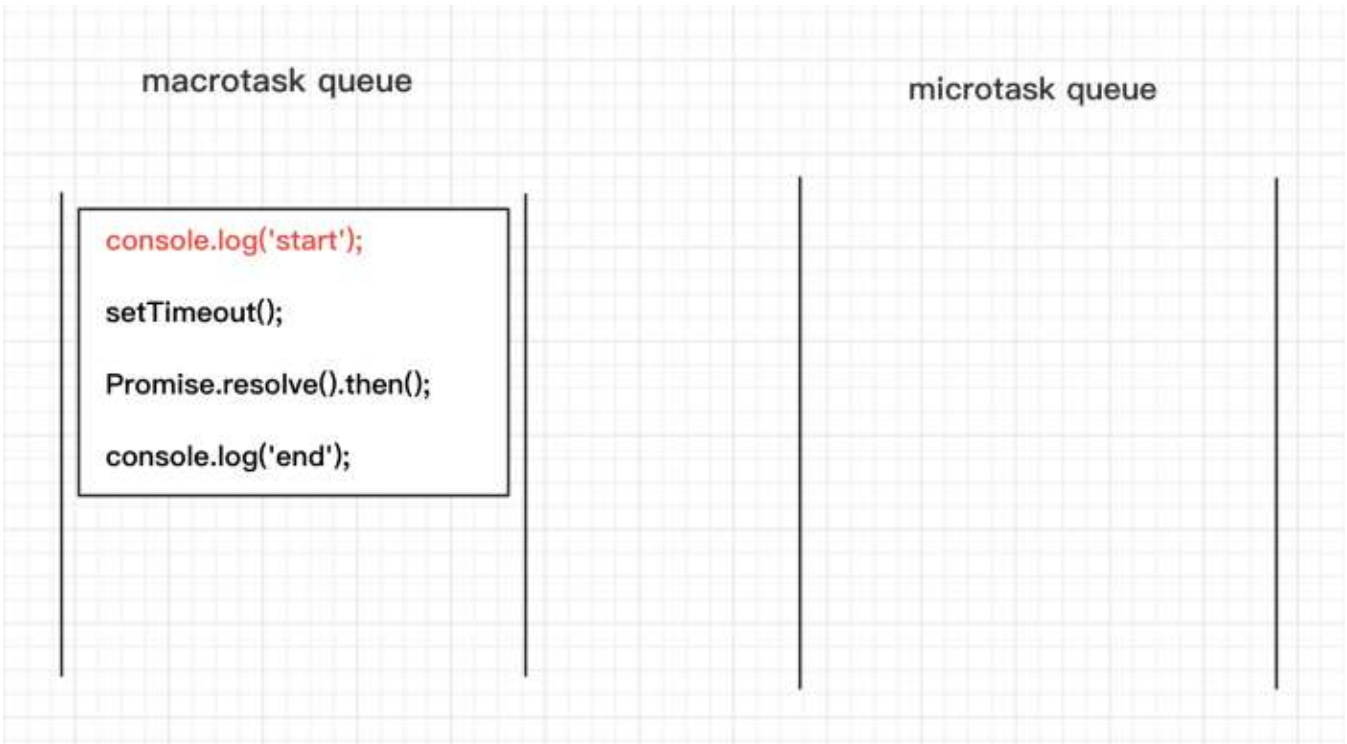
- First, there are two task queues in the JS engine: the macrotask queue and the microtask queue.



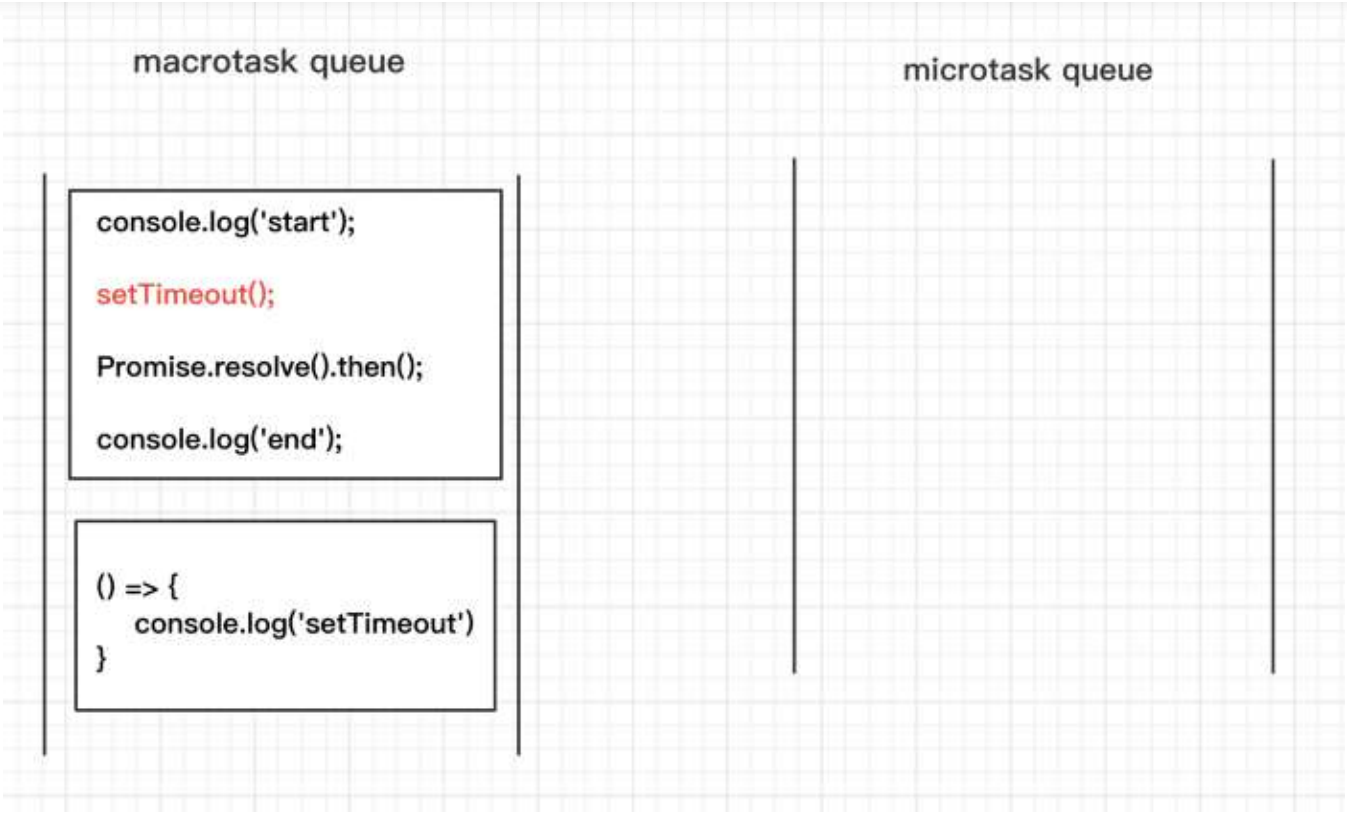
- At the beginning of the program, all the initial code is treated as a macro task, pushed into the macro task queue.



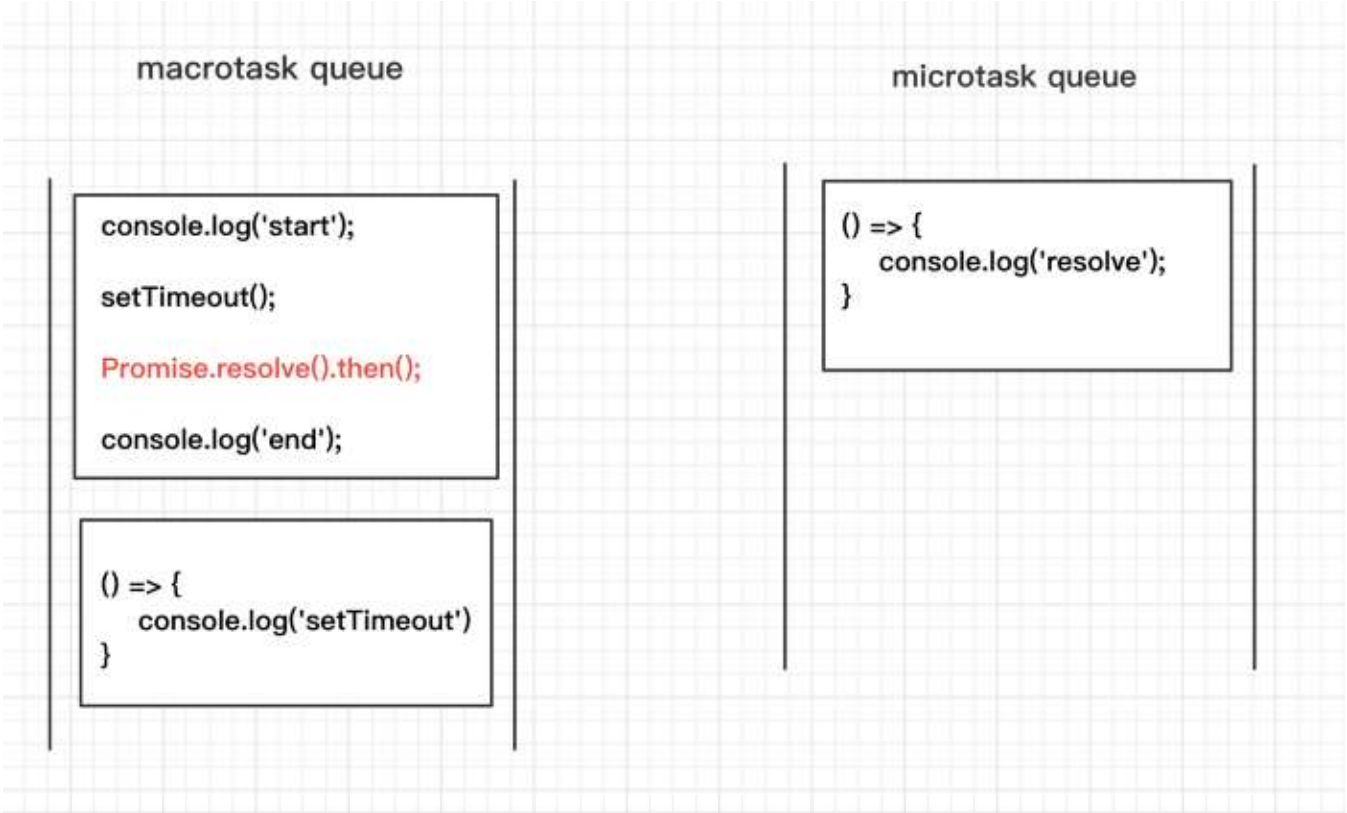
- The first line of code `console.log('start')` is then executed and `'start'` is printed in the console.

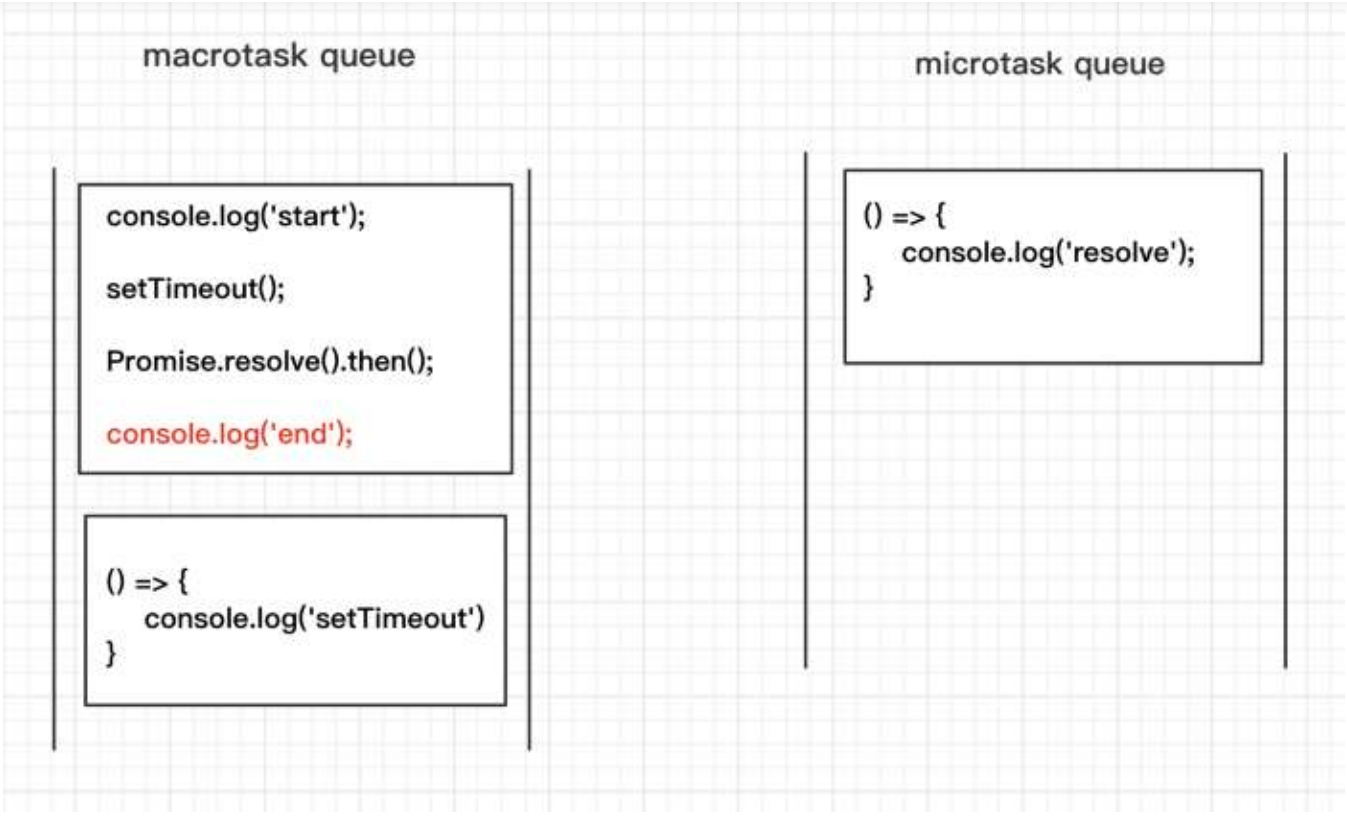


- Then the `setTimeout(...)` is a timer with a wait time of 0, which will be executed immediately. As we mentioned at the beginning of this article, `setTimeout` is a

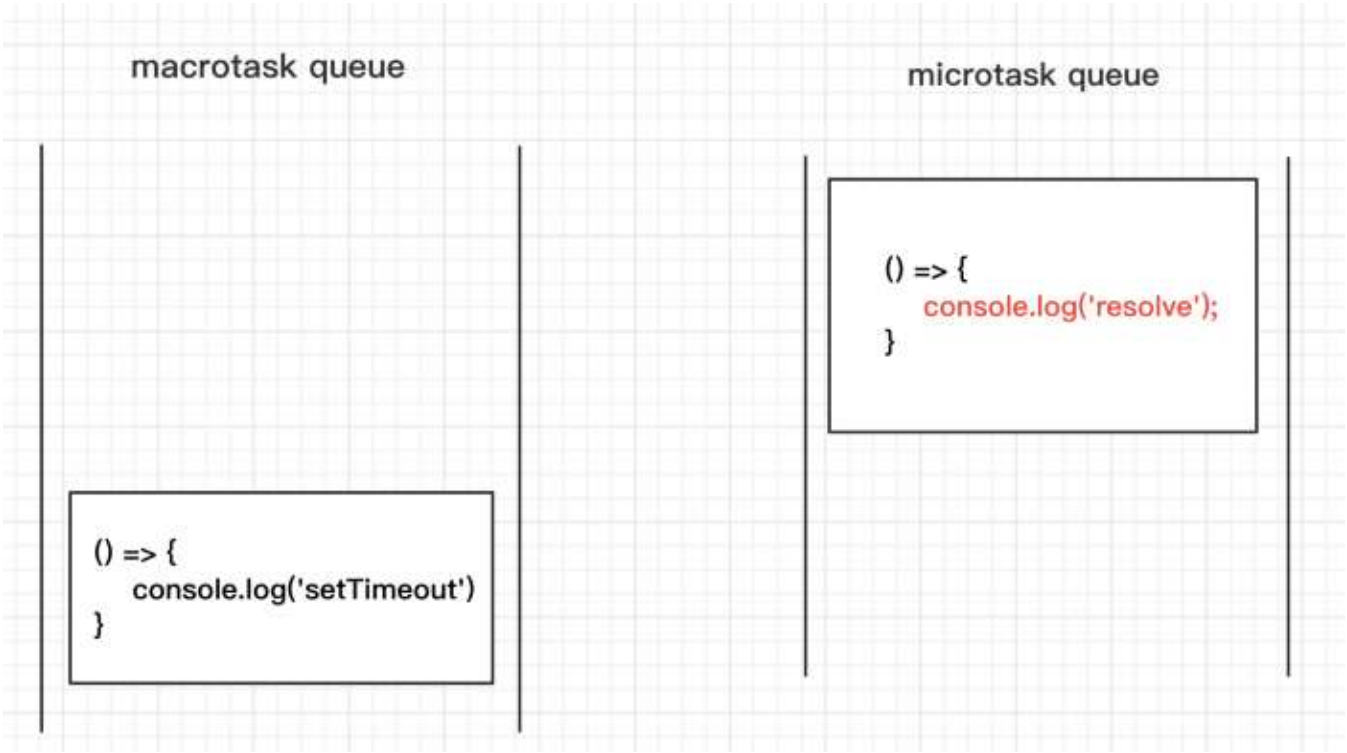


- Then it start executing `Promise.resolve().then(…)` , and `() => {console.log('resolve')}` gets pushed into the queue of microtasks.

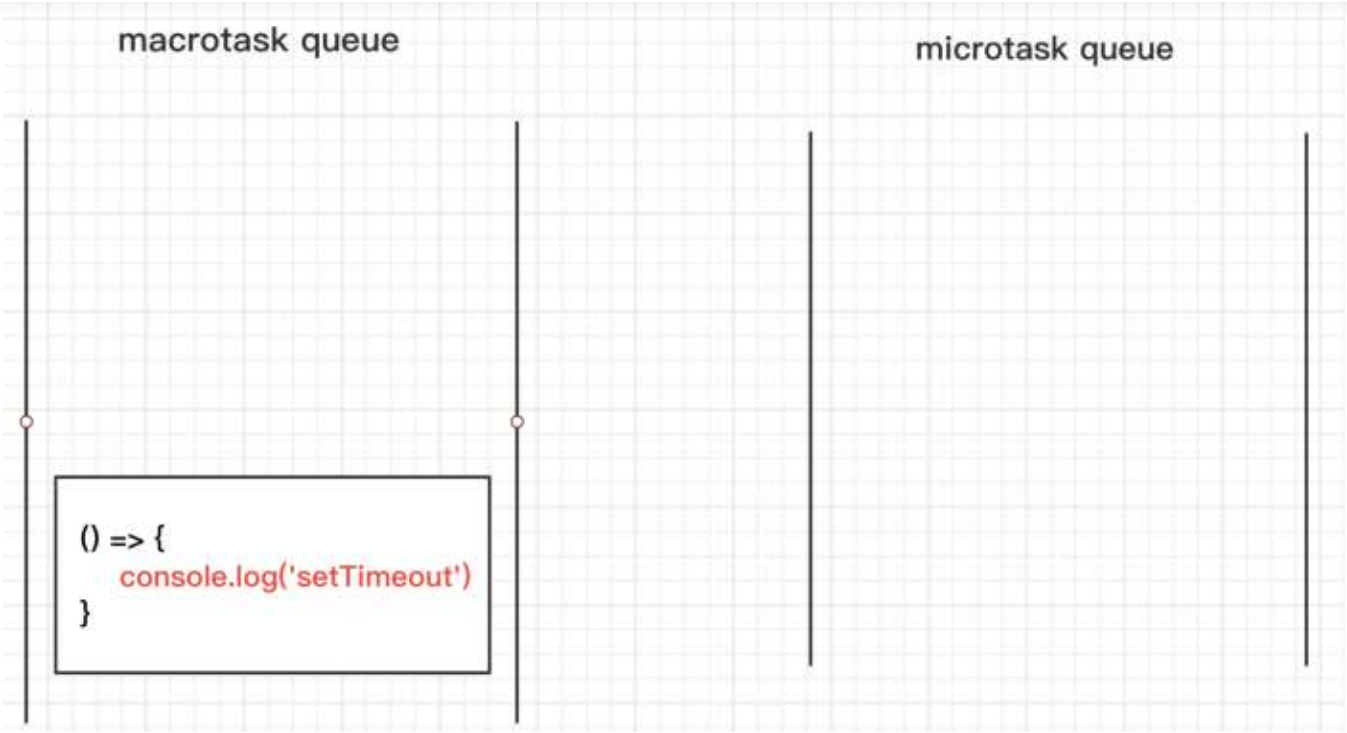




- When a macro task completes, the JS engine checks the queue of microtasks first and executes all the microtasks in turn.



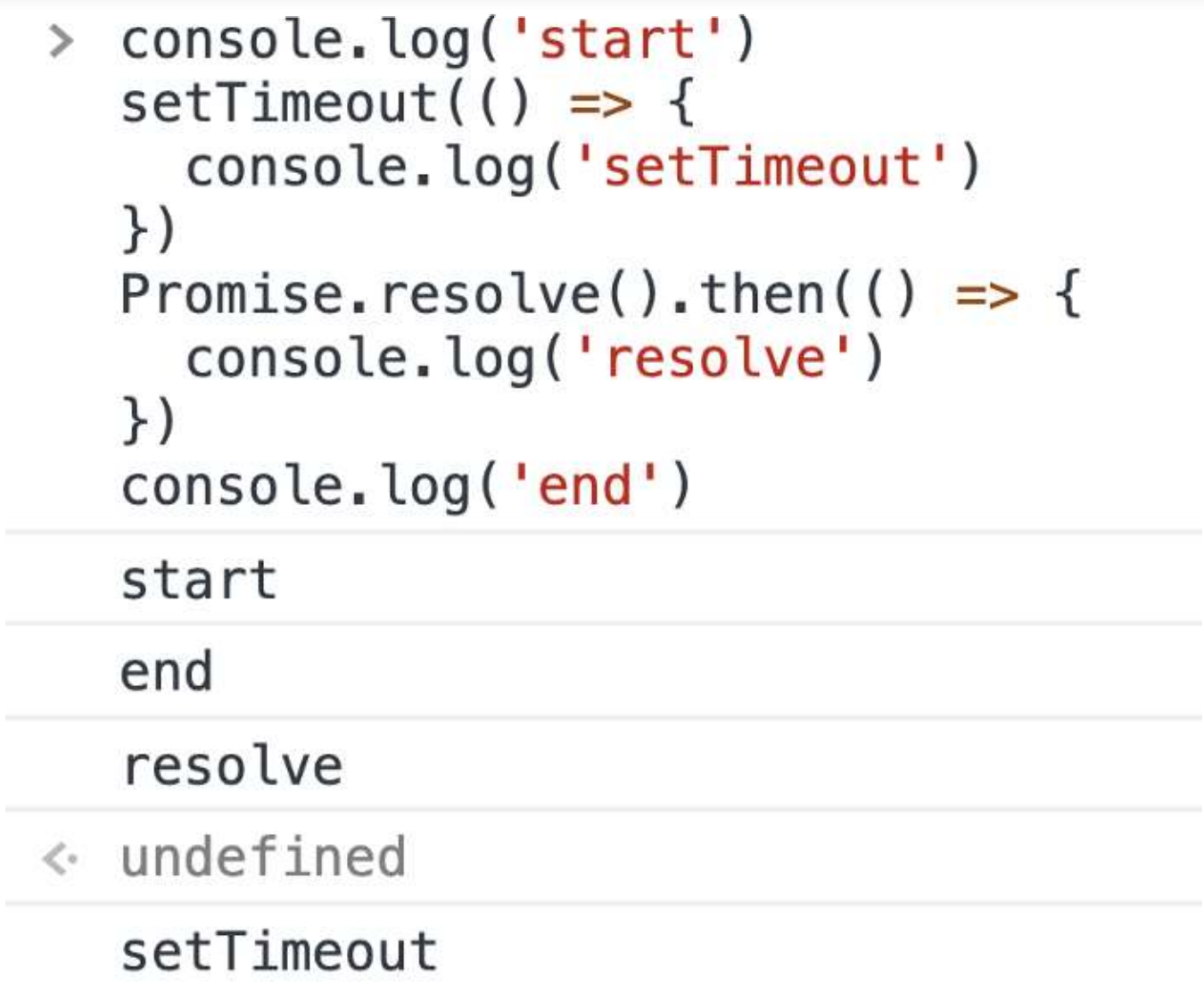
- When the microtask queue is empty, the JS engine checks the macro task queue



It is worth emphasizing that while `setTimeout(...)` is executed earlier than `Promise.resolve().then(...)`, the callback function for `setTimeout(...)` is still executed later because `setTimeout` is a macro task. This is where the novice makes the most mistakes.

Okay, so that's how the sample code above runs. I hope my sketch will help you.

Result:



6.

Example:

```
1  const promise = new Promise((resolve, reject) => {
2    console.log(1);
3    setTimeout(() => {
4      console.log("timerStart");
5      resolve("success");
6      console.log("timerEnd");
7    }, 0);
8    console.log(2);
9  });
10 promise.then((res) => {
11   console.log(res);
```

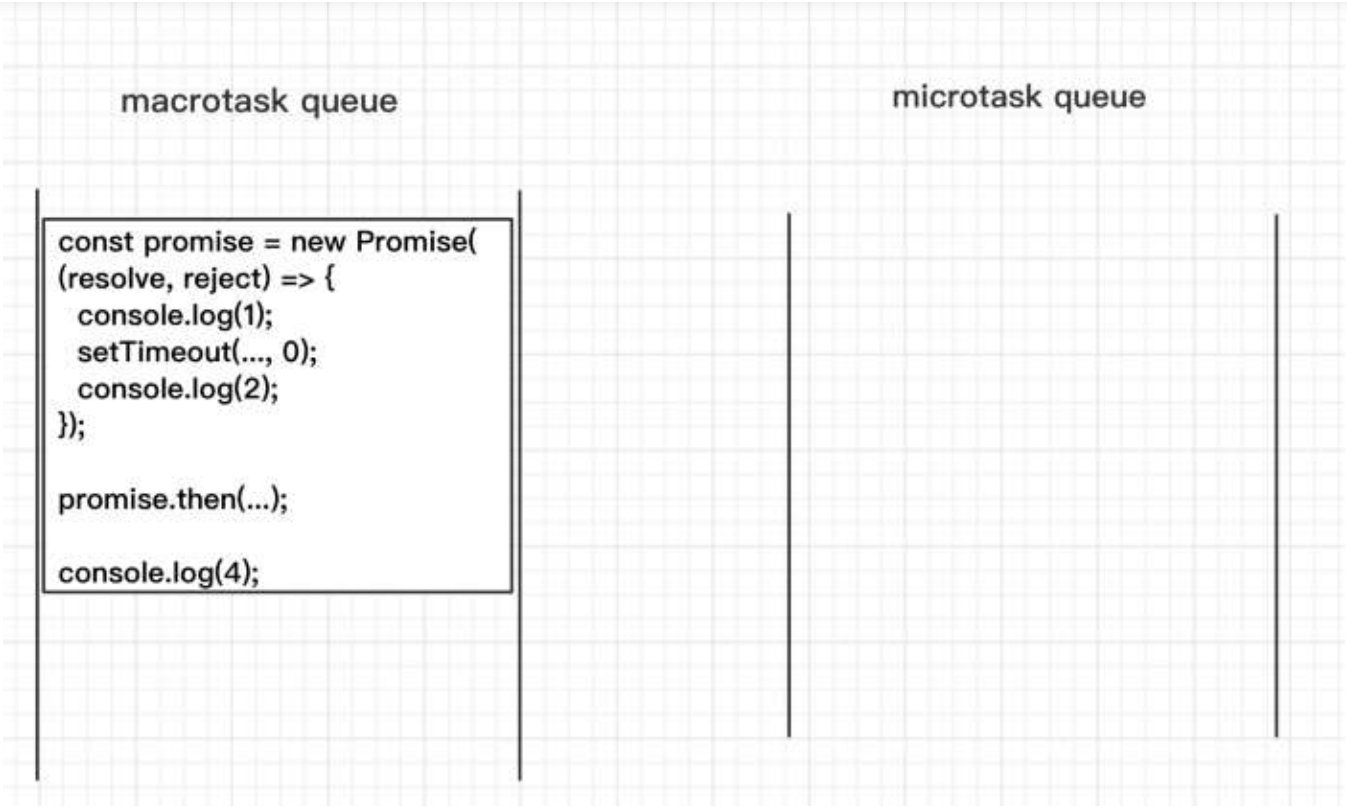
```
const promise = new Promise((resolve, reject) => {
  console.log(1);
  setTimeout(() => {
    console.log("timerStart");
    resolve("success");
    console.log("timerEnd");
  }, 0);
  console.log(2);
});

promise.then((res) => {
  console.log(res);
});

console.log(4);
```

Process Analysis:

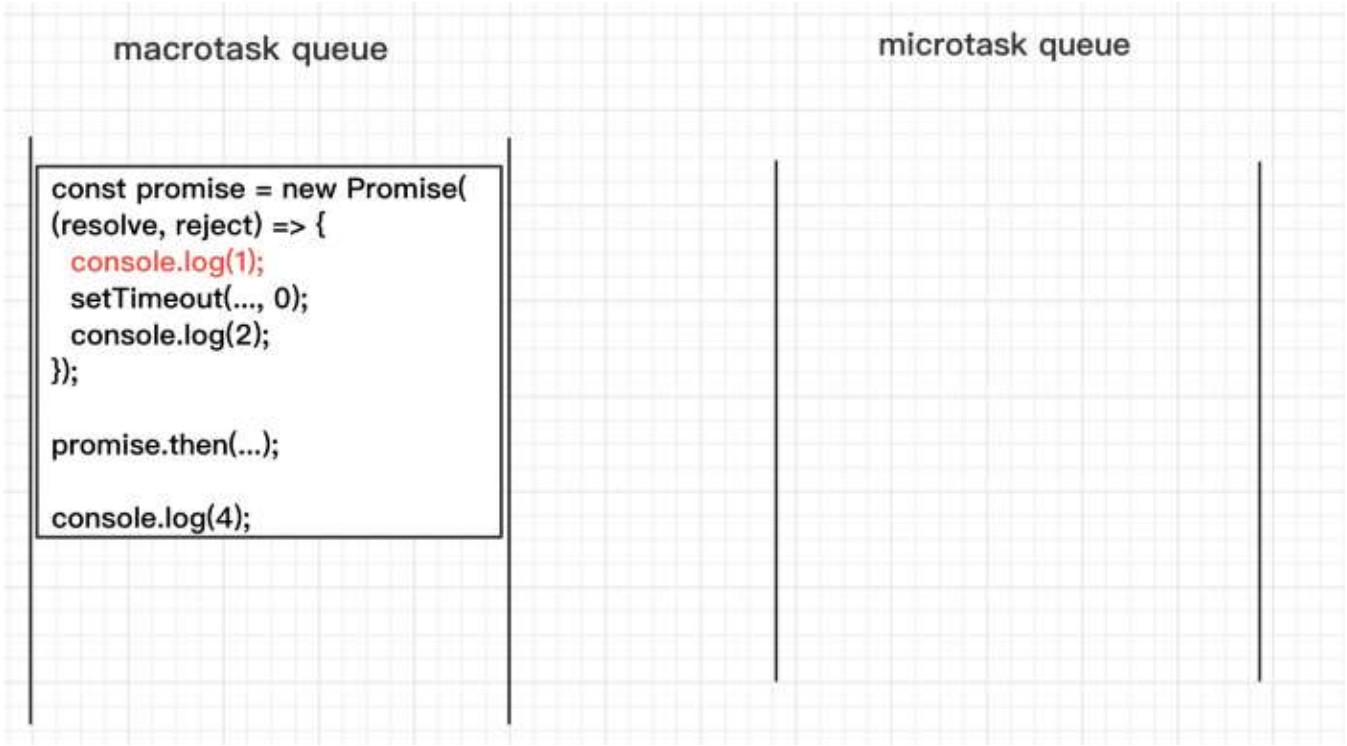
- First of all, we temporarily ignore those callback functions and simplify the code:

```
const promise = new Promise((resolve, reject) => {
  console.log(1);
  setTimeout(..., 0);
  console.log(2);
});

promise.then(...);

console.log(4);
```

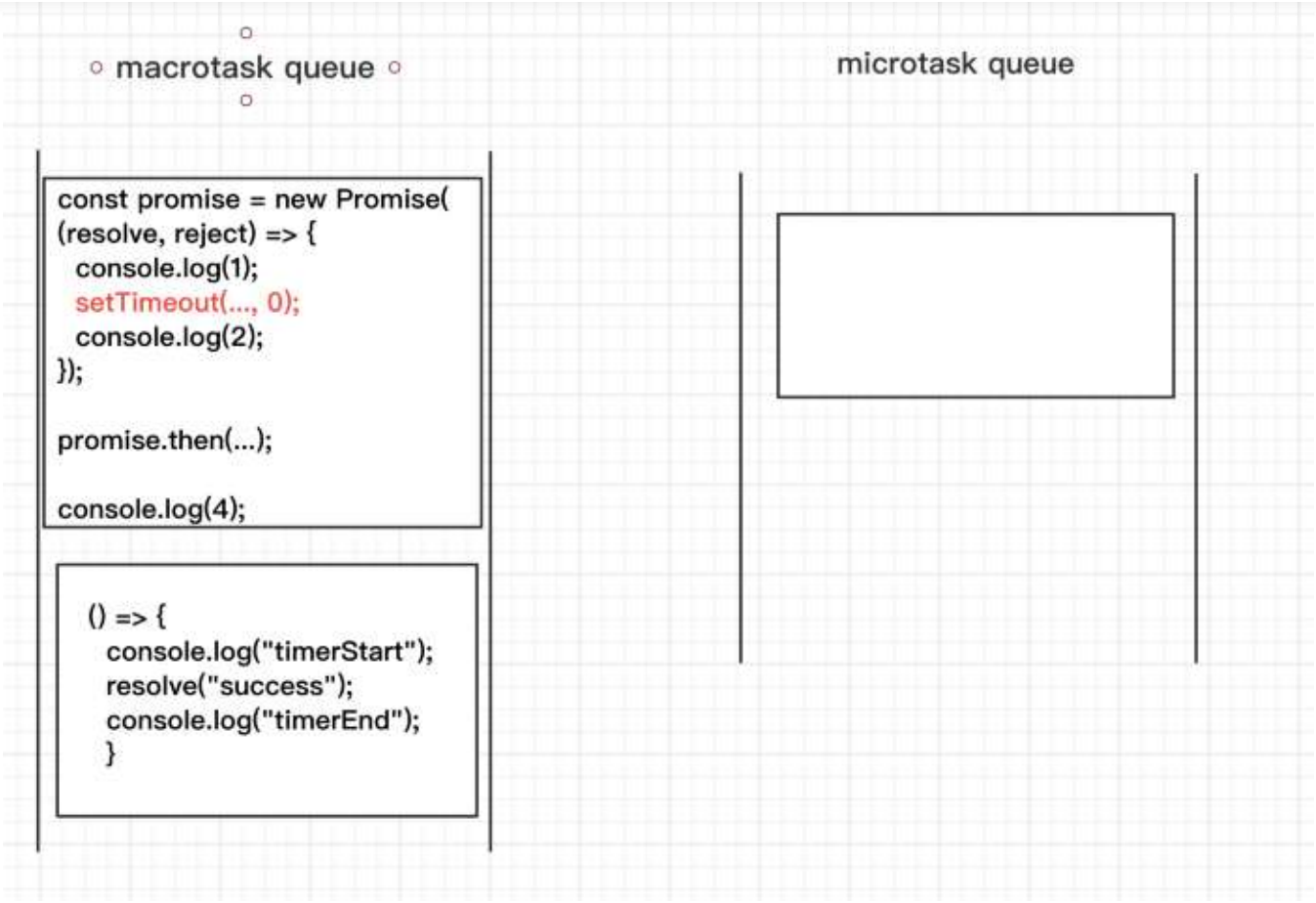
- Then we draw the picture as before. At first all the code can be thought of as a macro task.

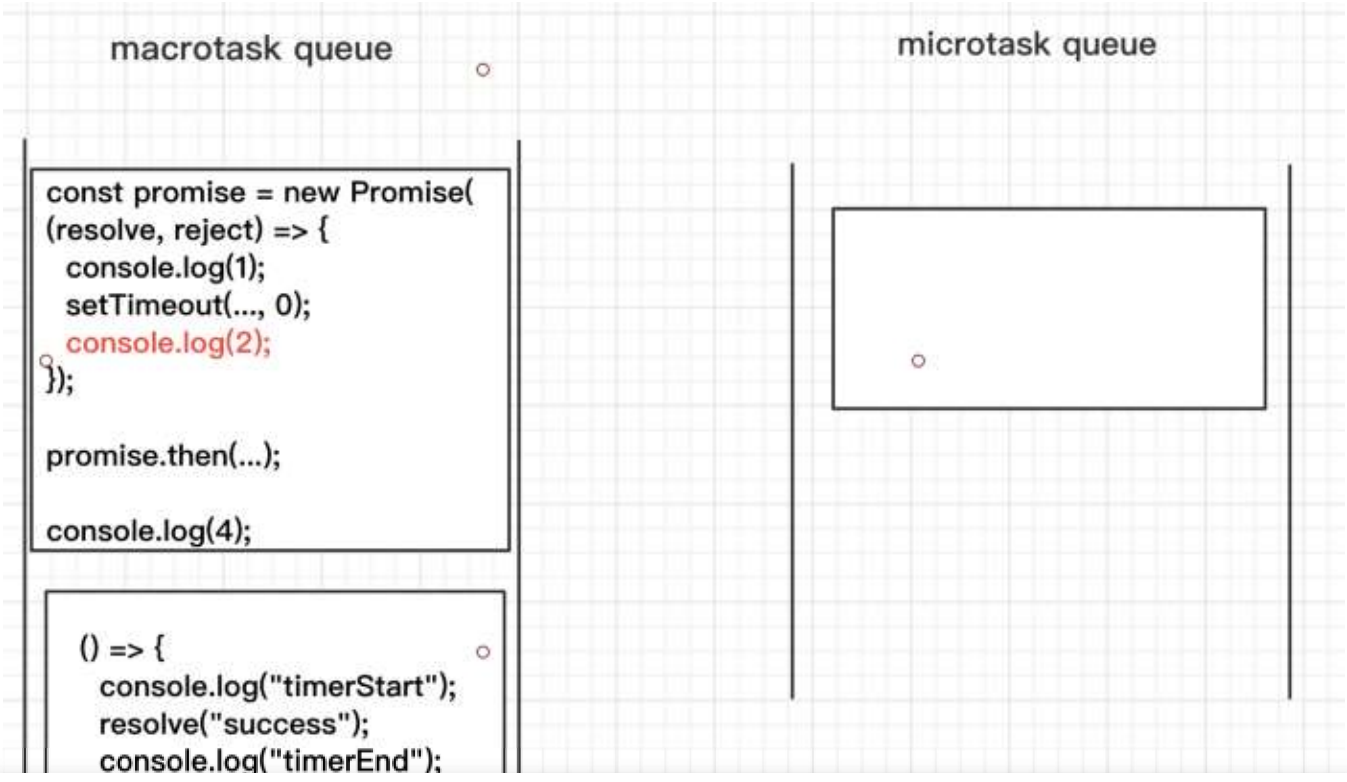
- Then start executing `new Promise(...)`, and then go inside the executor and executing `console.log(1)`.



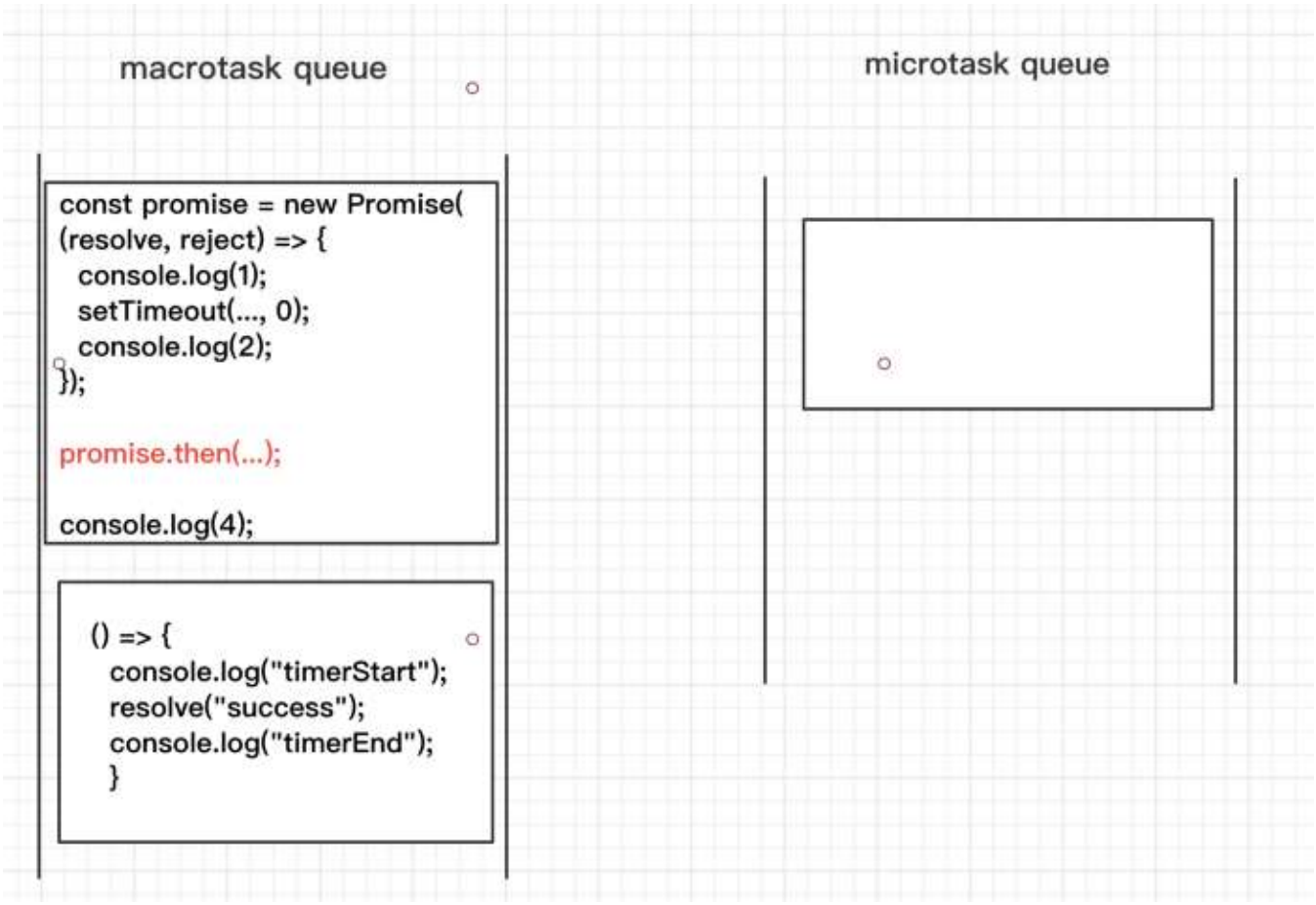
- Then start executing ... The timer finish immediately and its



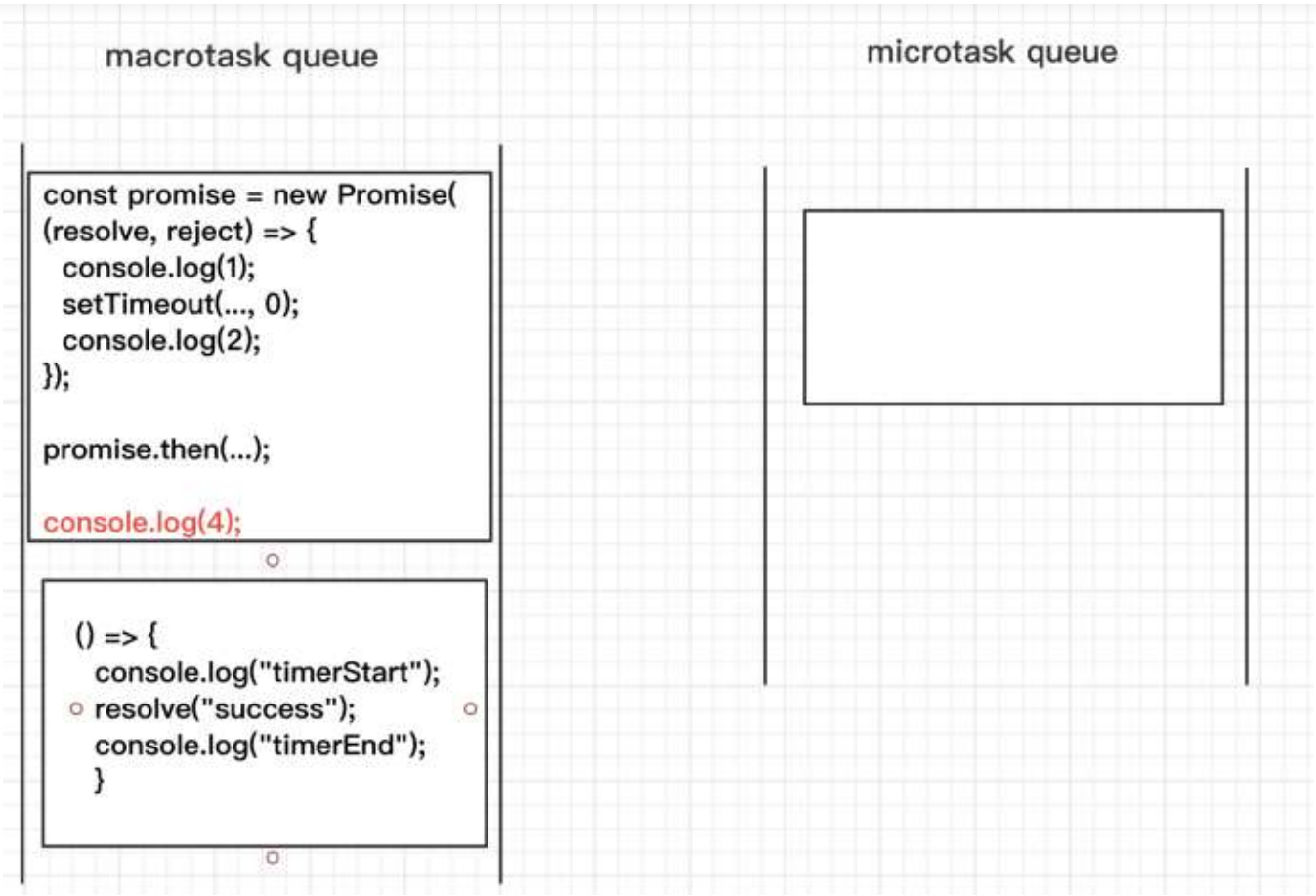
- Then start executing `console.log(2)` .



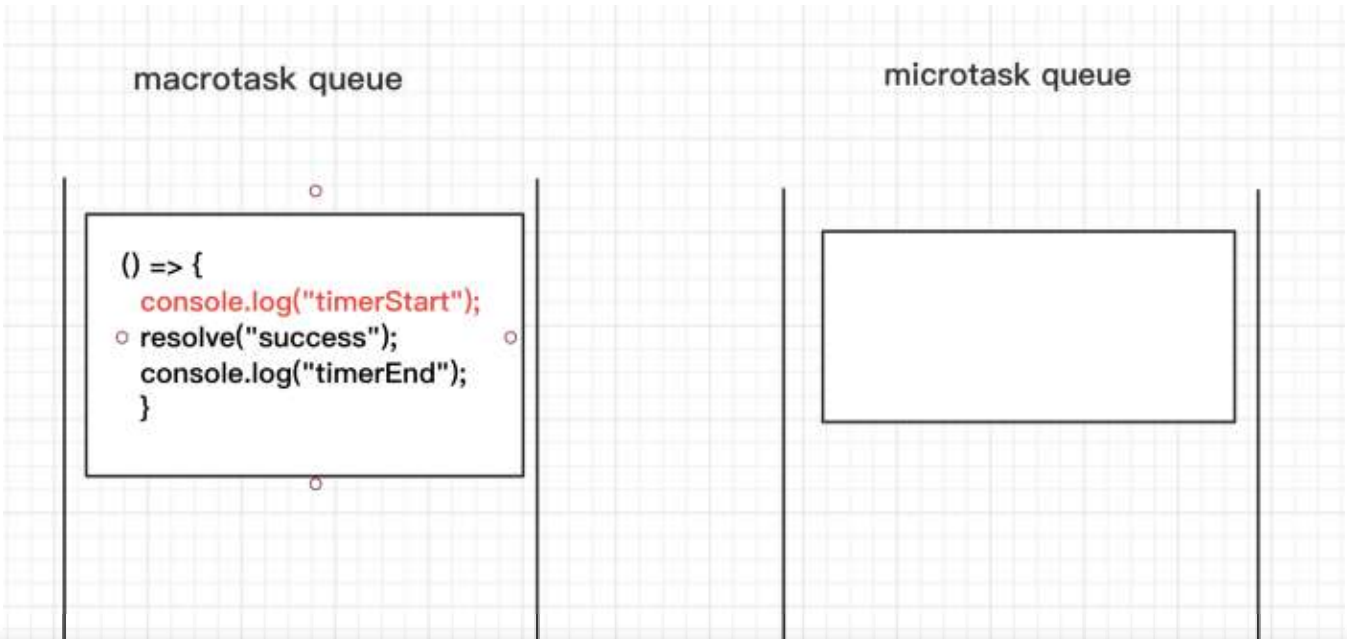
- Now start executing `promise.then(…)` . Because the promise object is still in the pending state, its callback function is not yet pressed into the queue of microtasks. That is, the microtask queue is currently still empty.



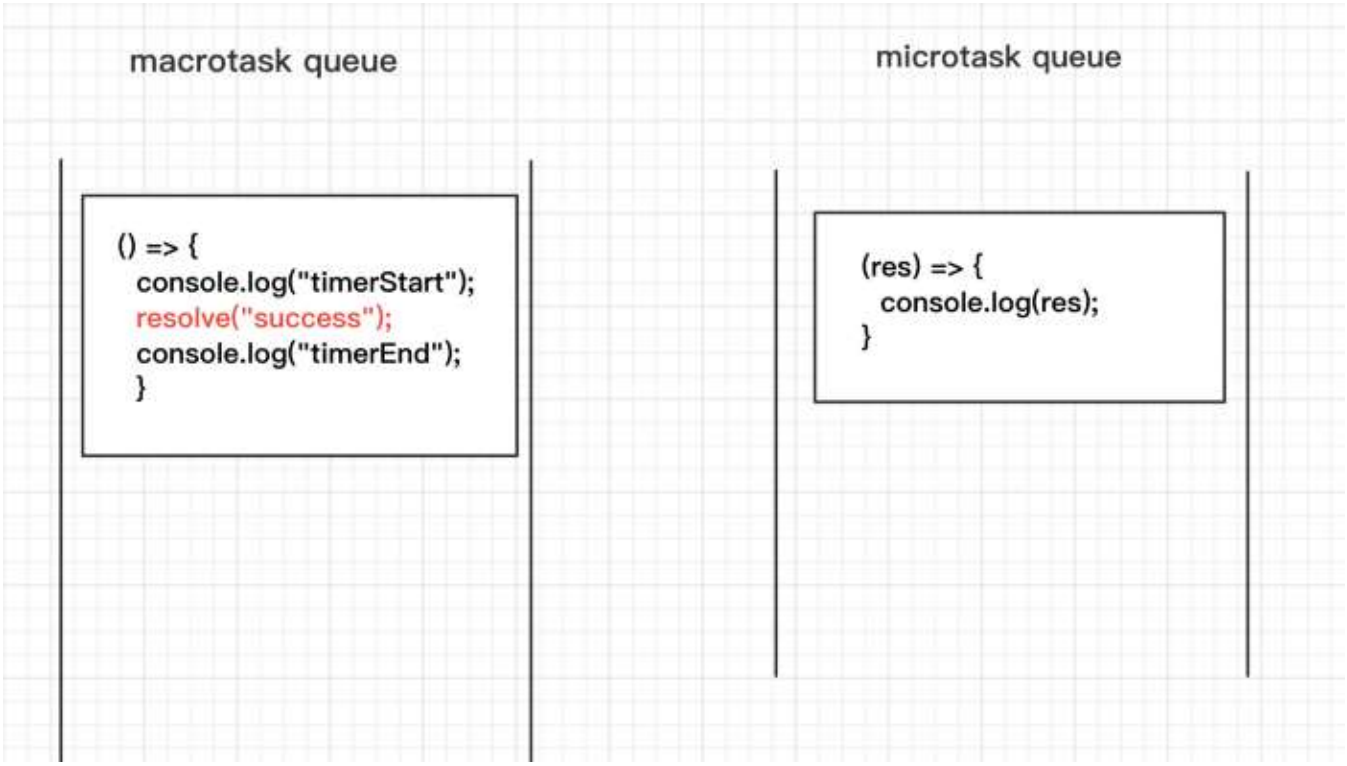
- Then start executing `console.log(4)` .



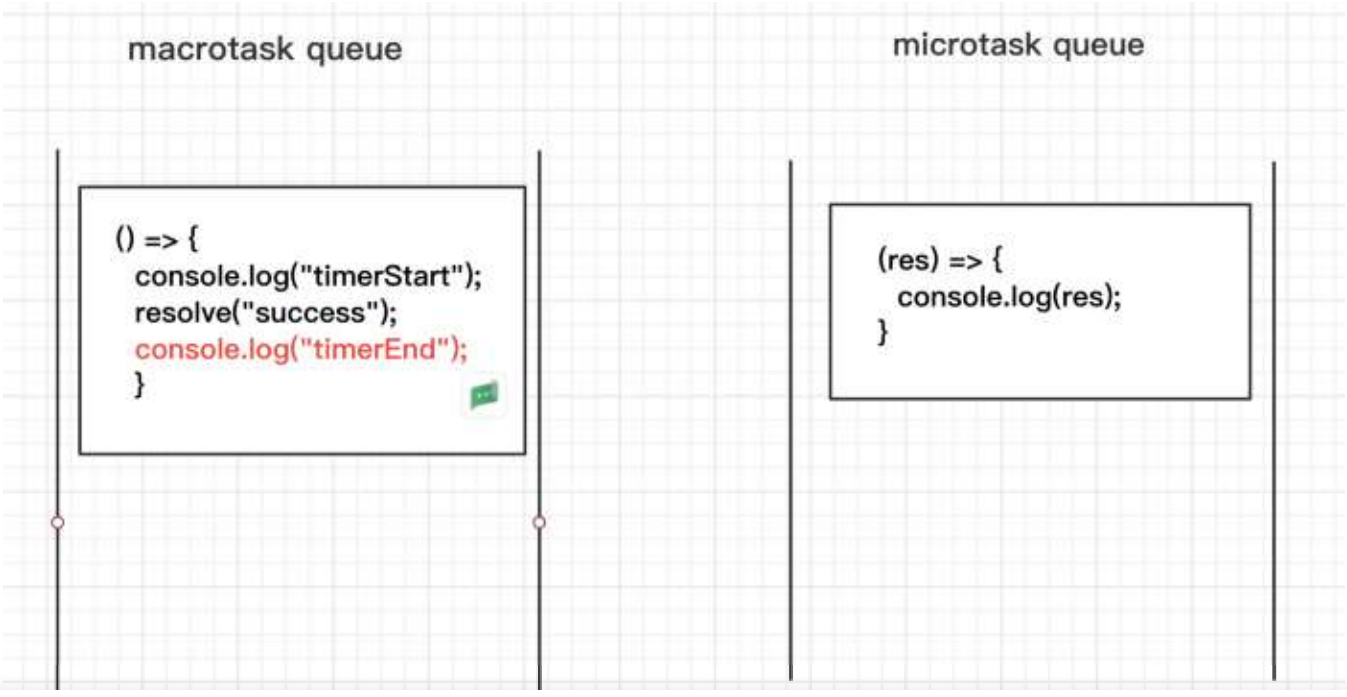
- At this point, the first macro task ends, and the microtask queue is still empty, so the JS engine starts the next macro task.
- Then start executing `console.log('timerStart')` .

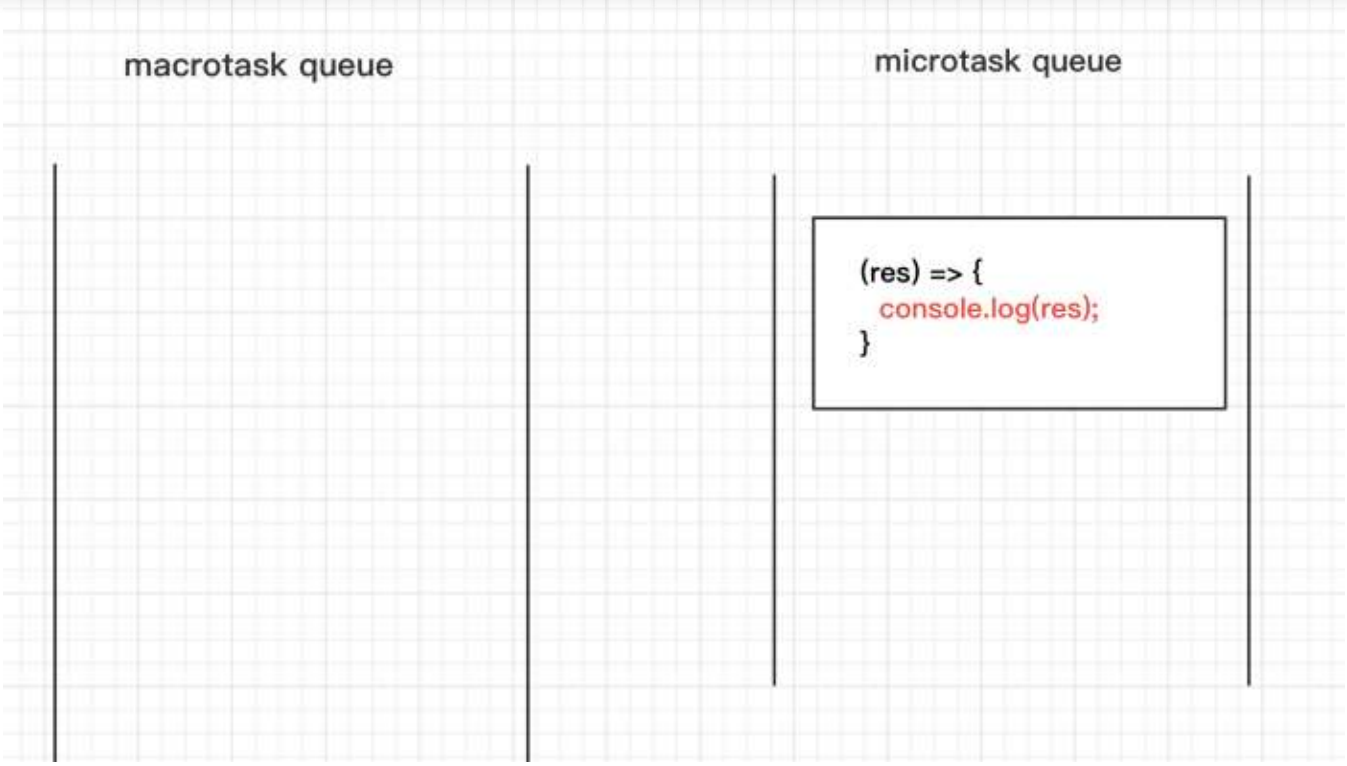


- Now that the `resolve()` function is executed, the state of the promise will be `resolved` and the callback function of `promise.then(…)` is pushed into the queue of microtasks.



- Then start executing `console.log('timerEnd')` .





Result:

```
> const promise = new Promise((resolve, reject) => {
    console.log(1);
    setTimeout(() => {
      console.log("timerStart");
      resolve("success");
      console.log("timerEnd");
    }, 0);
    console.log(2);
  });
  promise.then((res) => {
    console.log(res);
  });
  console.log(4);
  1
  2
  4
< undefined
  timerStart
```

7.

Example:

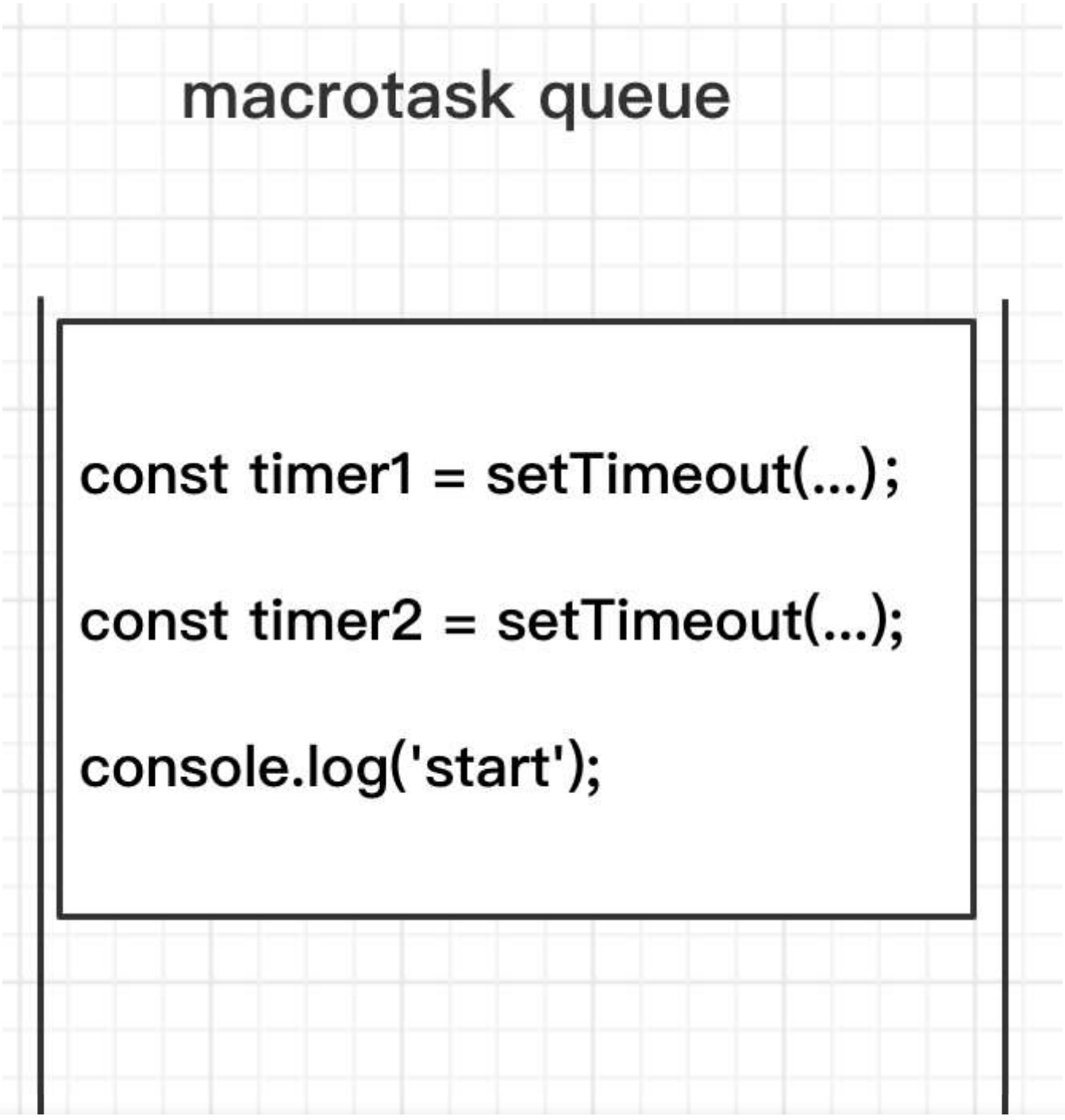
```
const timer1 = setTimeout(() => {
  console.log('timer1');

const timer3 = setTimeout(() => {
    console.log('timer3')
```

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

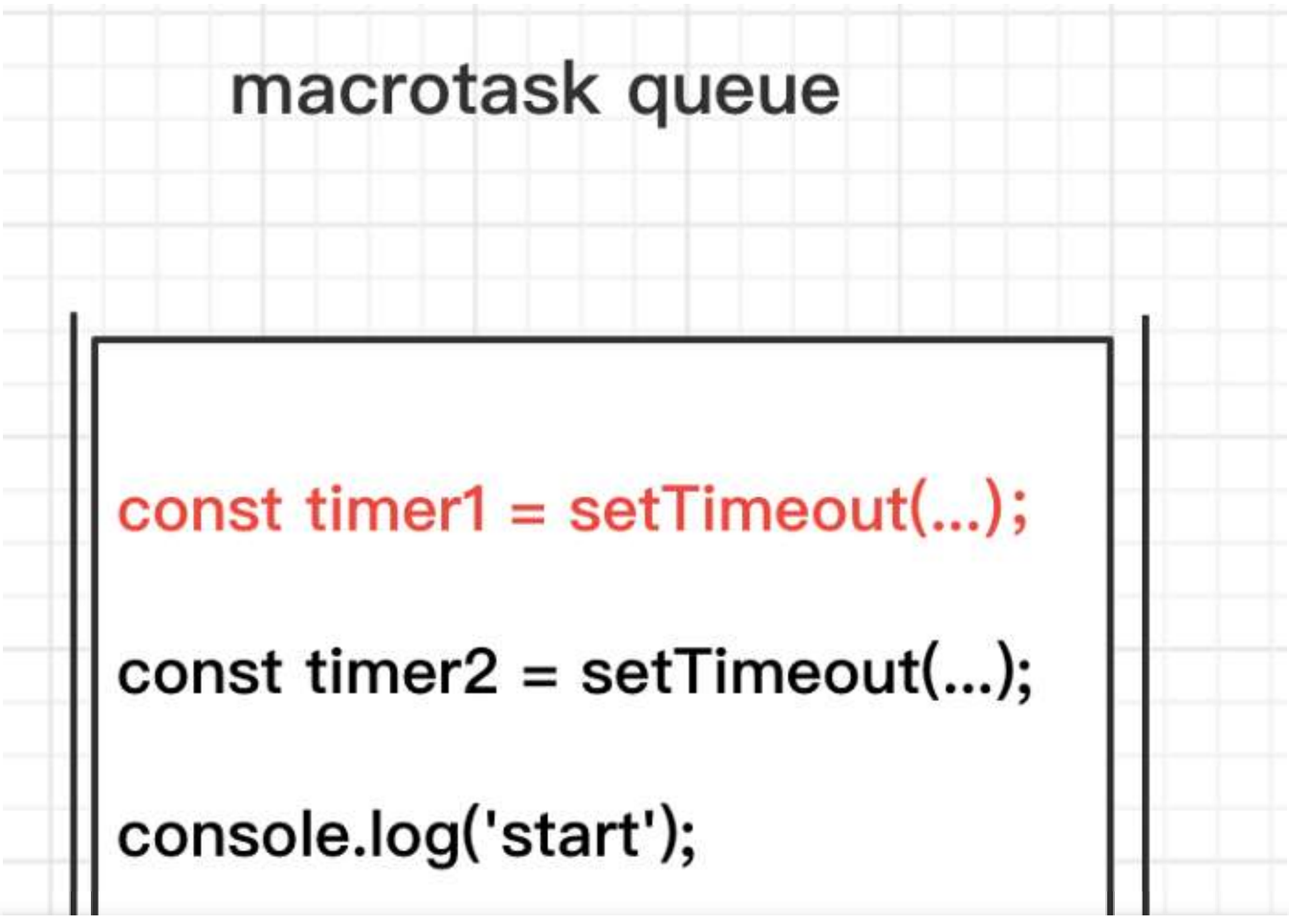
Process Analysis:

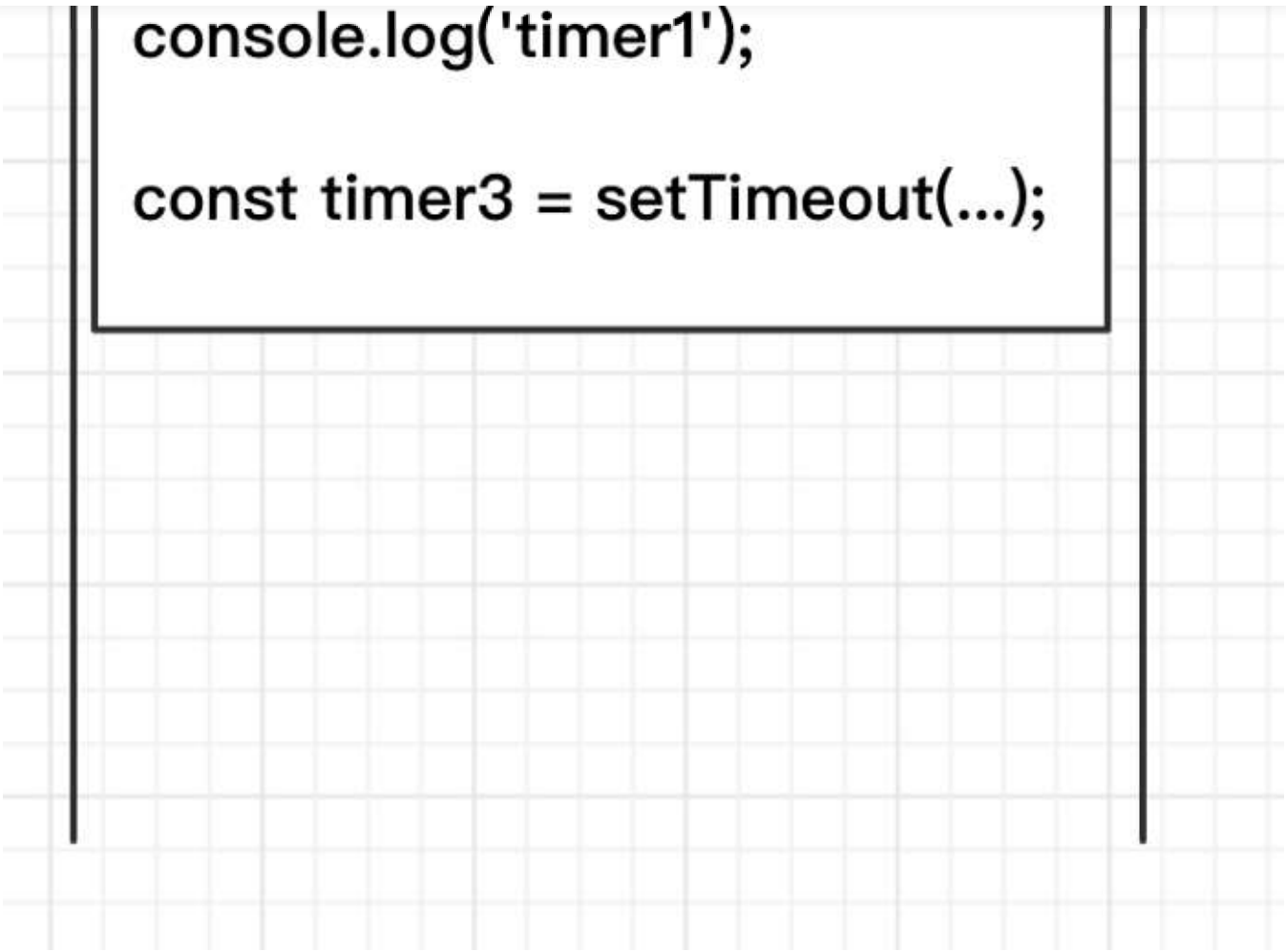
There are three `setTimeout` function in this example, so the program accumulates three additional macro tasks.

- First, let's draw the initial macro task queue.

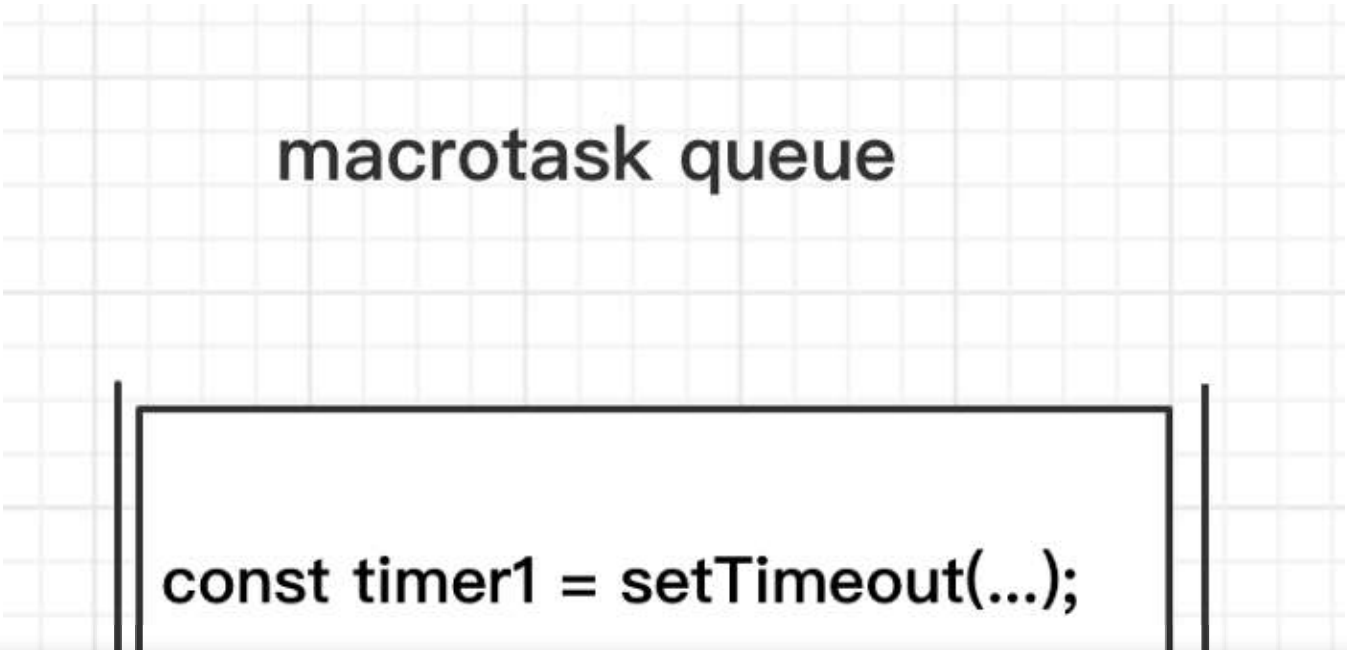


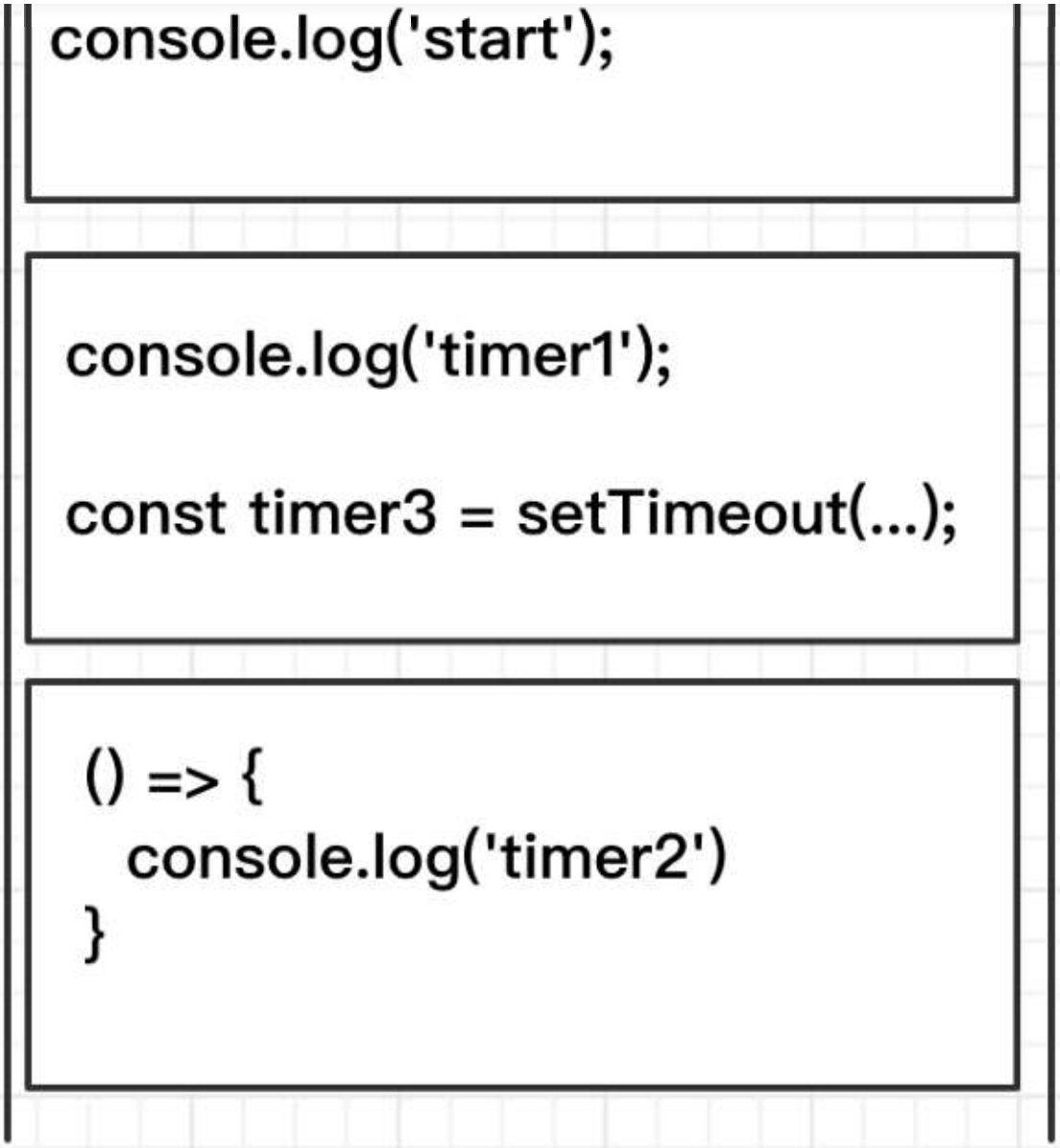
- Then start executing the `setTimeout(...)` corresponding to `timer1` . Meanwhile, a new macro task is created.



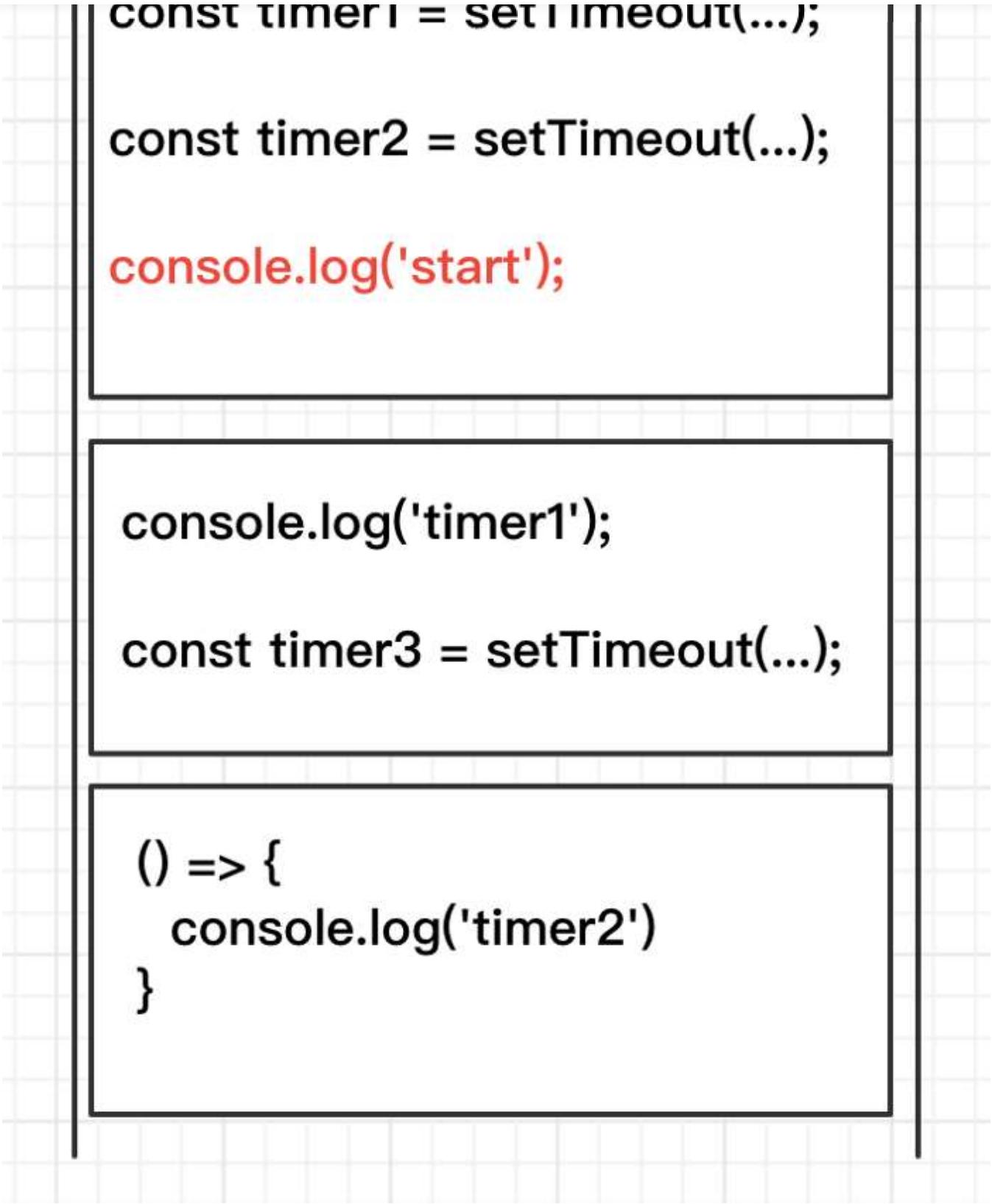


- Then start executing the `setTimeout` corresponding to `timer2` . Meanwhile, another new macro task is created.

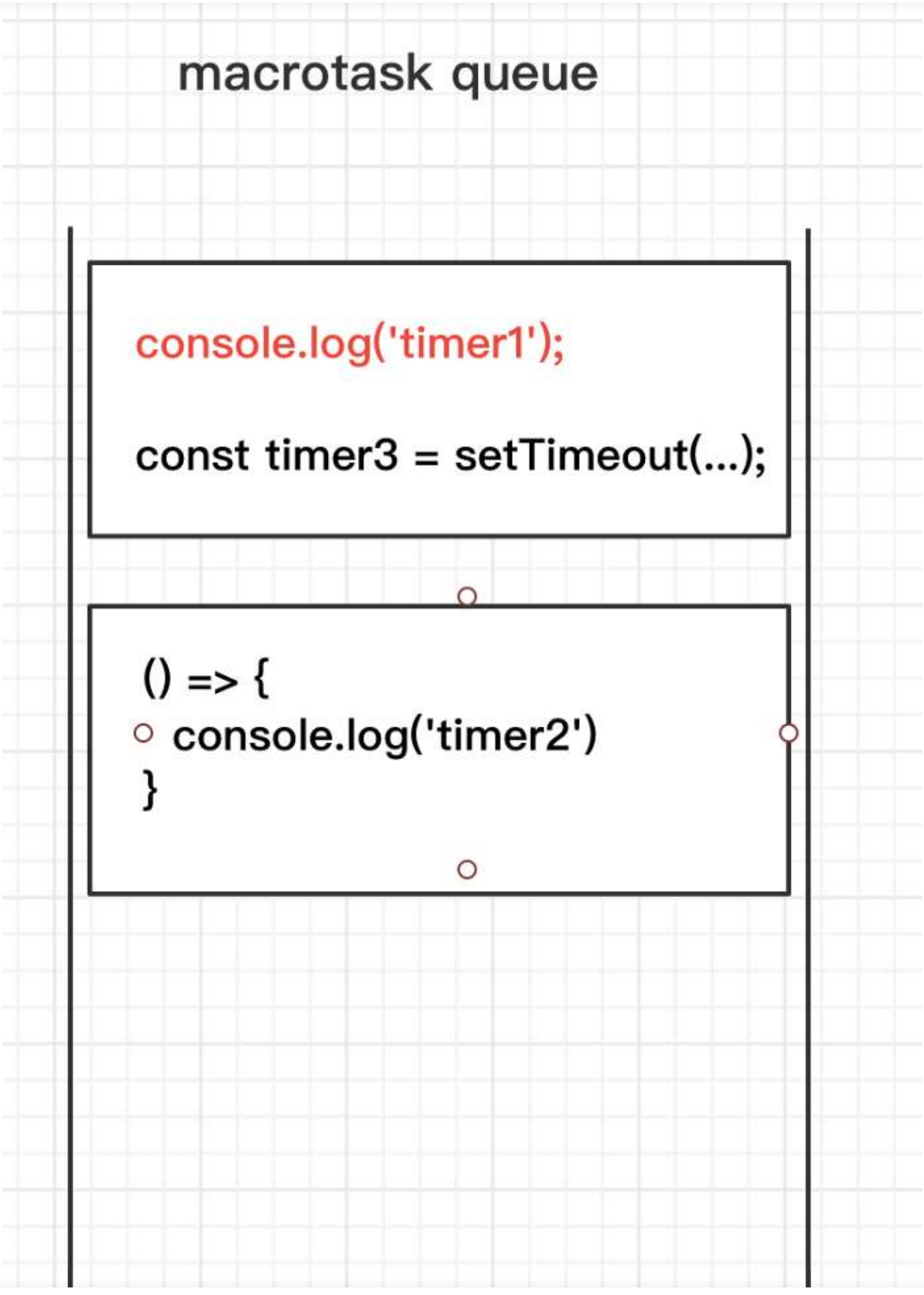




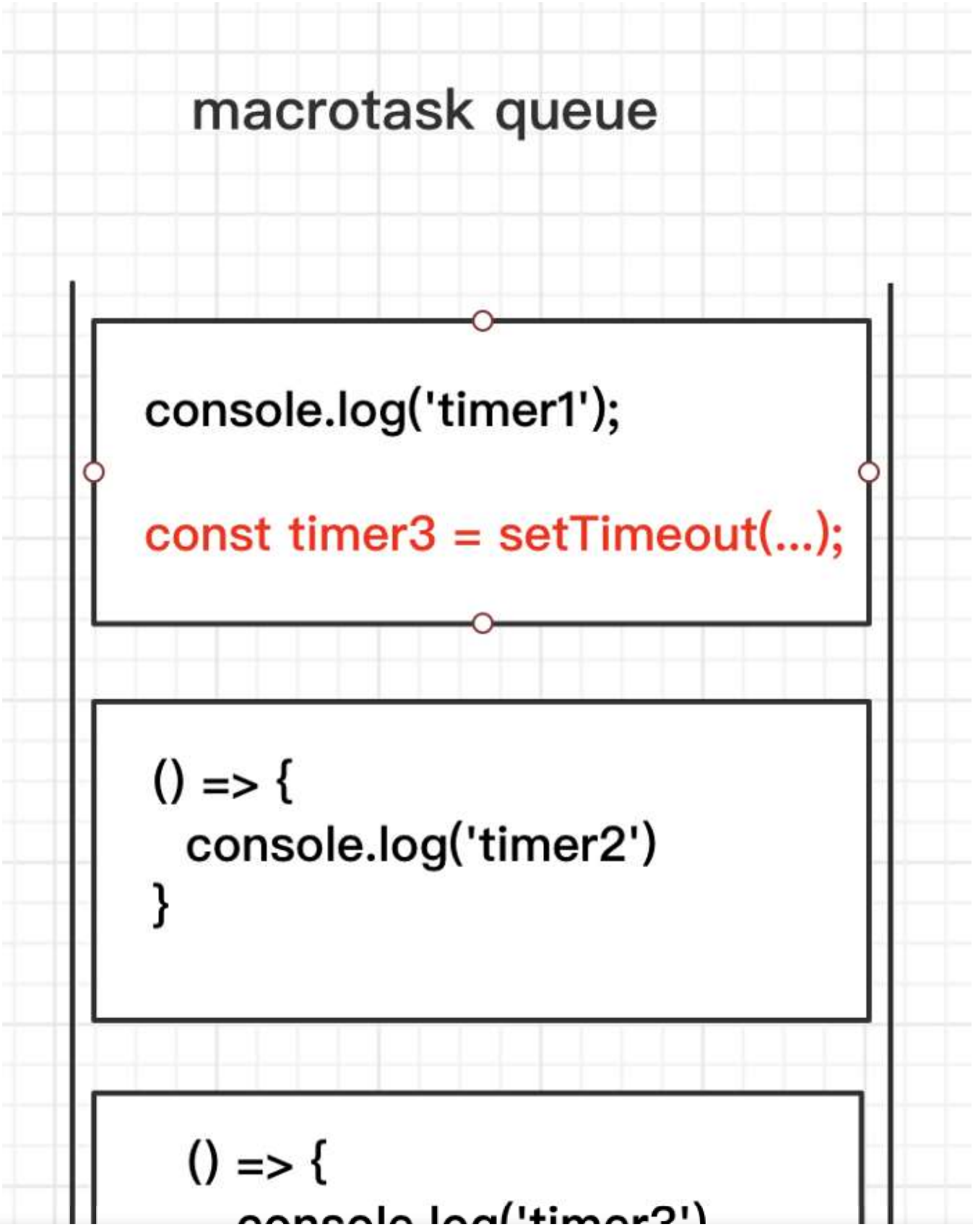
- Okay, now we have three macro tasks, no microtasks.
- Then



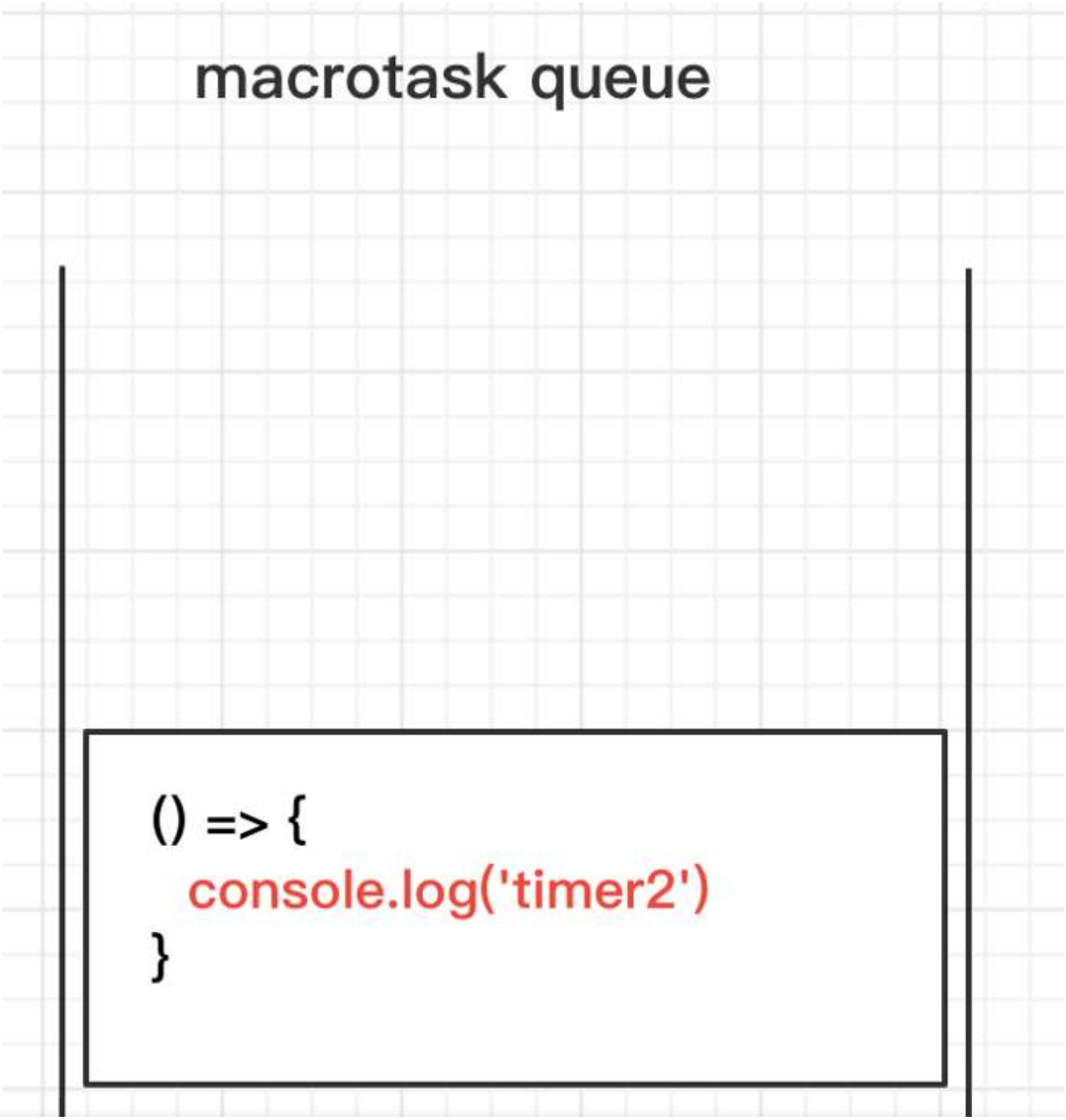
- Now that the first macro task and its execution are complete, and the microtask queue is still empty, the JS engine will start executing the next macro task.
- `console.log('timer1')` is executed.

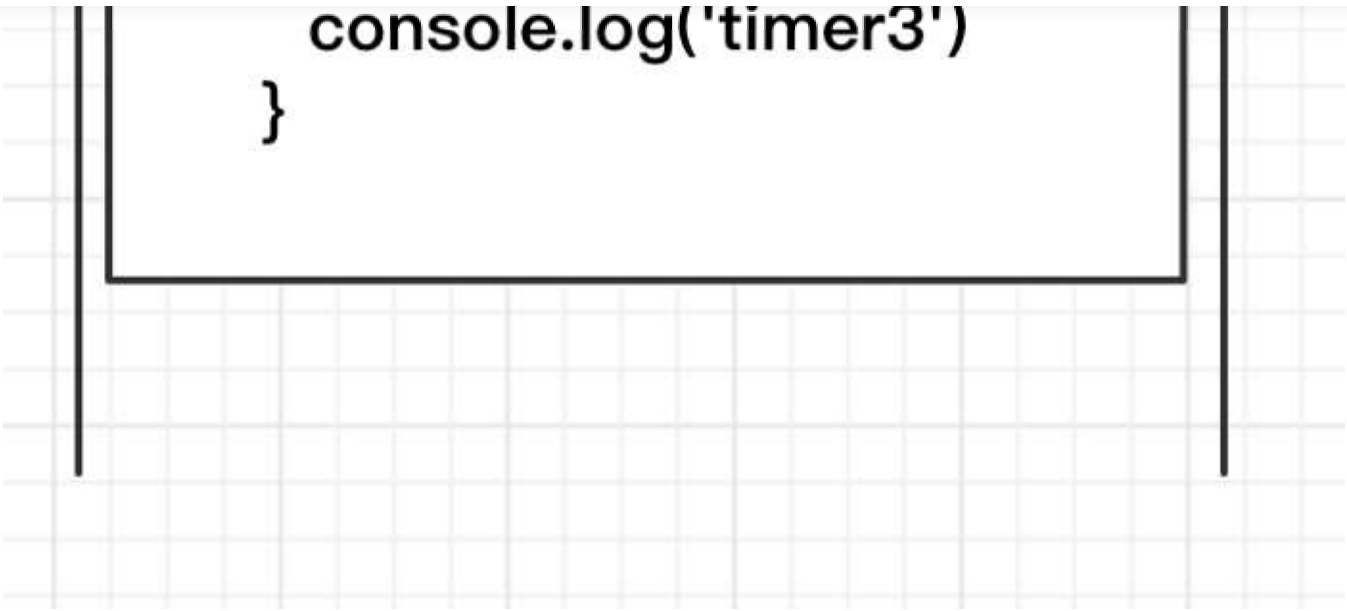


- Then start executing the `setTimeout(...)` corresponding to `timer3` . A new macro task is created.

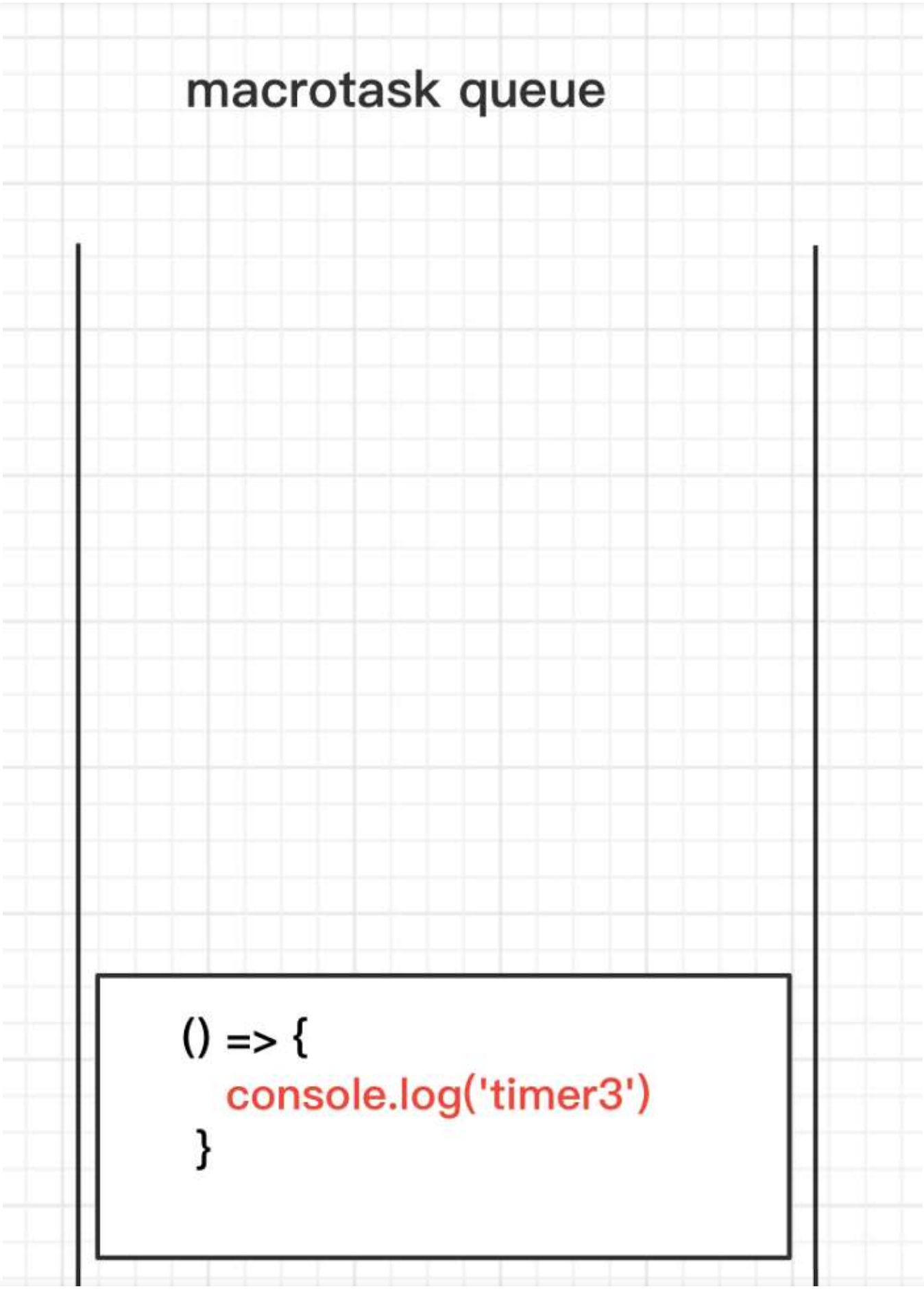


- Then





- Then



Result:

```
> const timer1 = setTimeout(() => {
    console.log('timer1');
    const timer3 = setTimeout(() => {
      console.log('timer3')
    }, 0)
  }, 0)
  const timer2 = setTimeout(() => {
    console.log('timer2')
  }, 0)
  console.log('start')

  start
< undefined
  timer1
  timer2
  timer3
```

8.

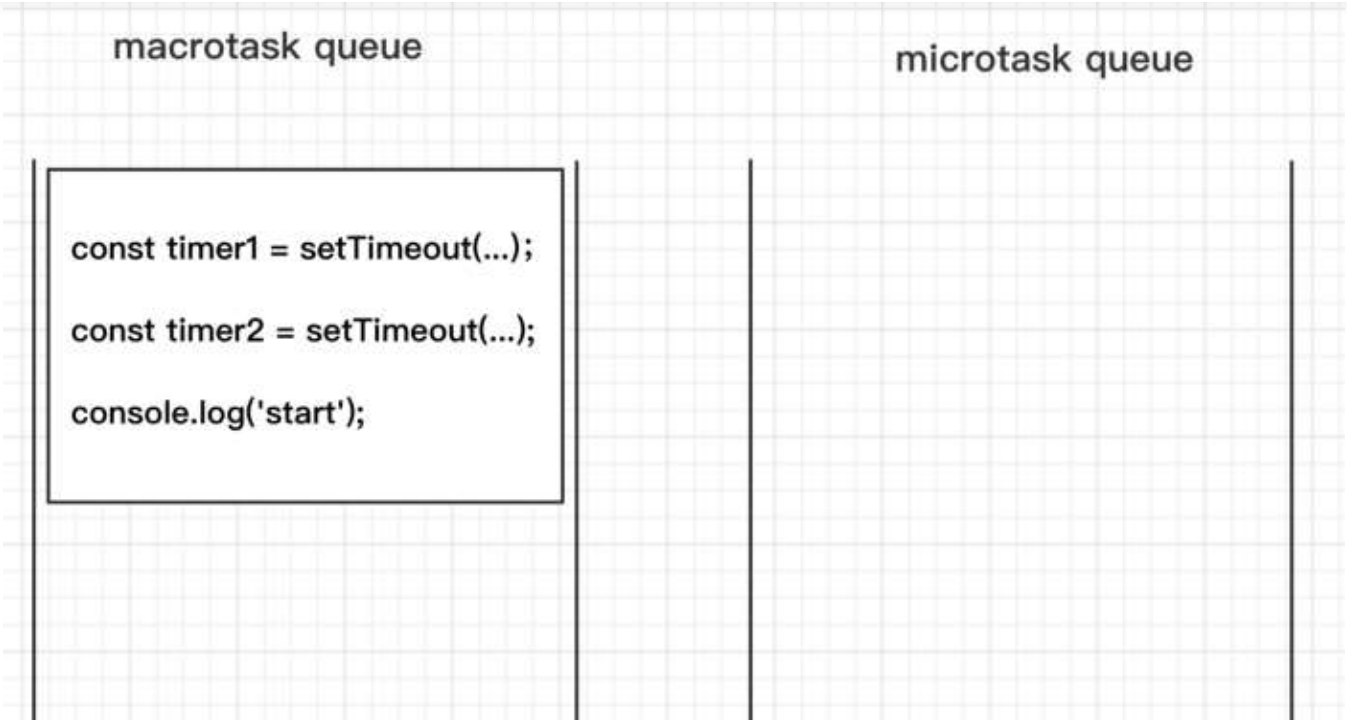
Example:

```
const timer1 = setTimeout(() => {
  console.log('timer1');
  const promise1 = Promise.resolve().then(() => {
    console.log('promise1')
  })
}, 0)

const timer2 = setTimeout(() => {
  console.log('timer2')
}, 0)

console.log('start')
```

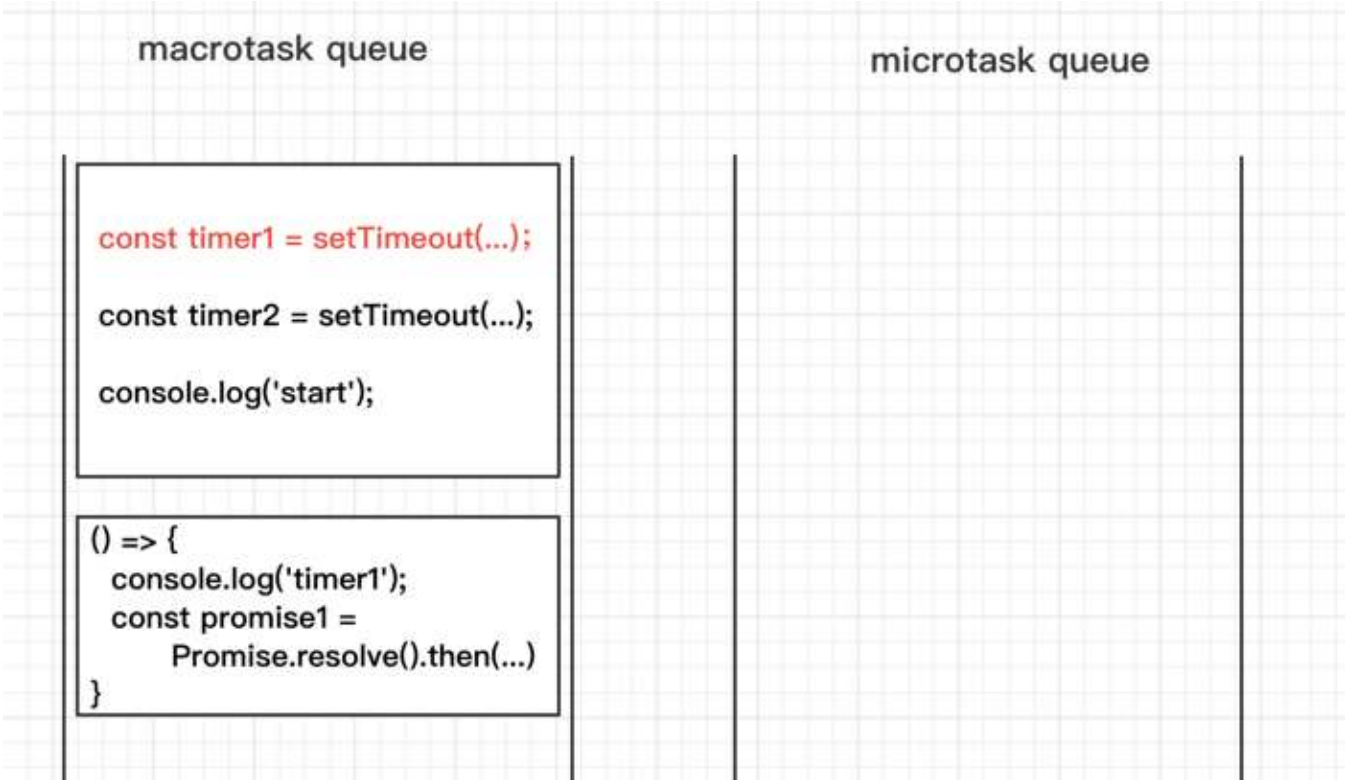
Process Analysis:

This example is similar to the previous one, except that we replaced one of the `setTimeout` with a `Promise.then`. Because `setTimeout` is a macro task and `Promise.then` is a microtask, and microtasks take precedence over macro tasks, the order of the output from the console is not the same.

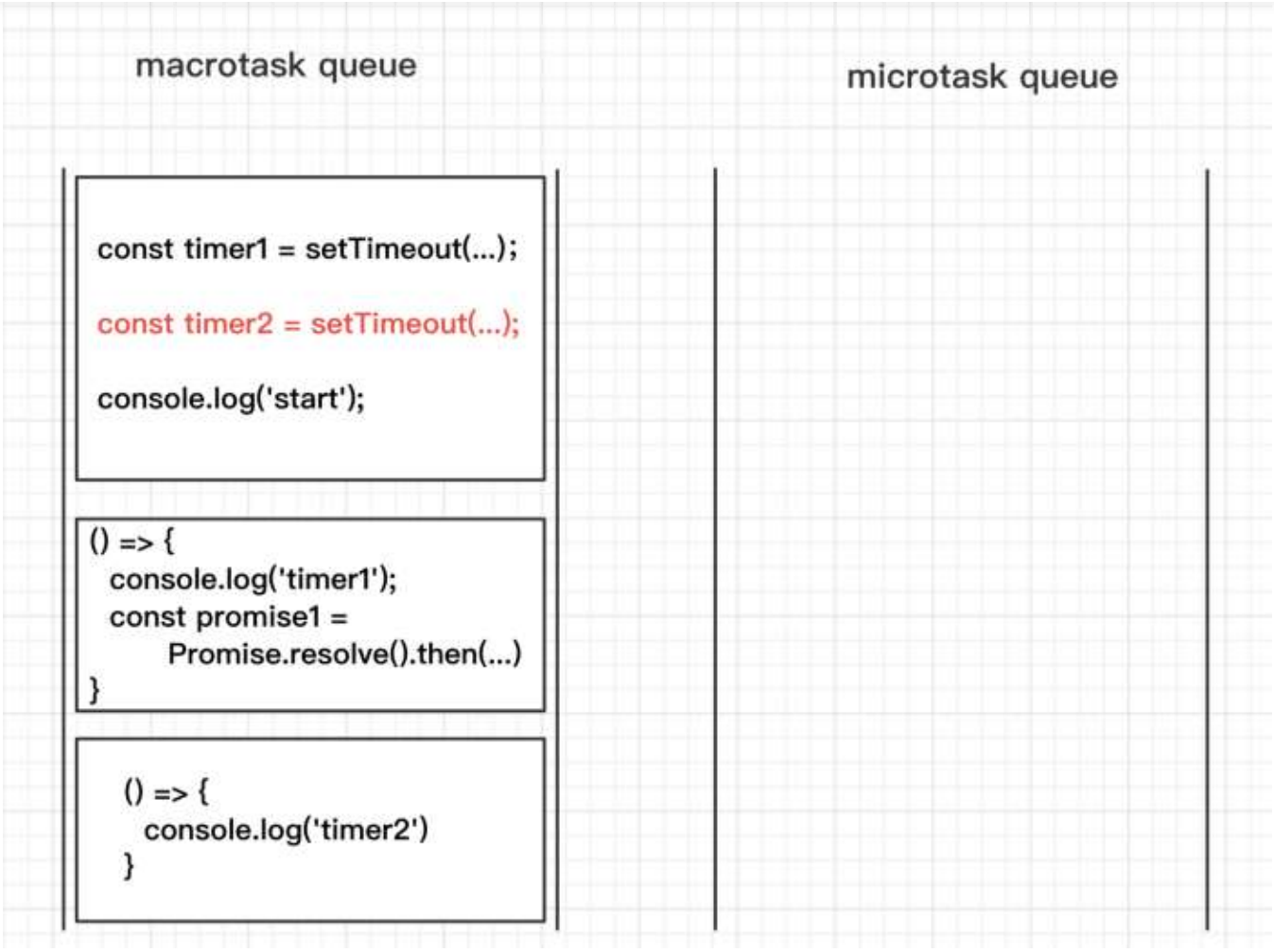
- First, let's draw the initial task queues

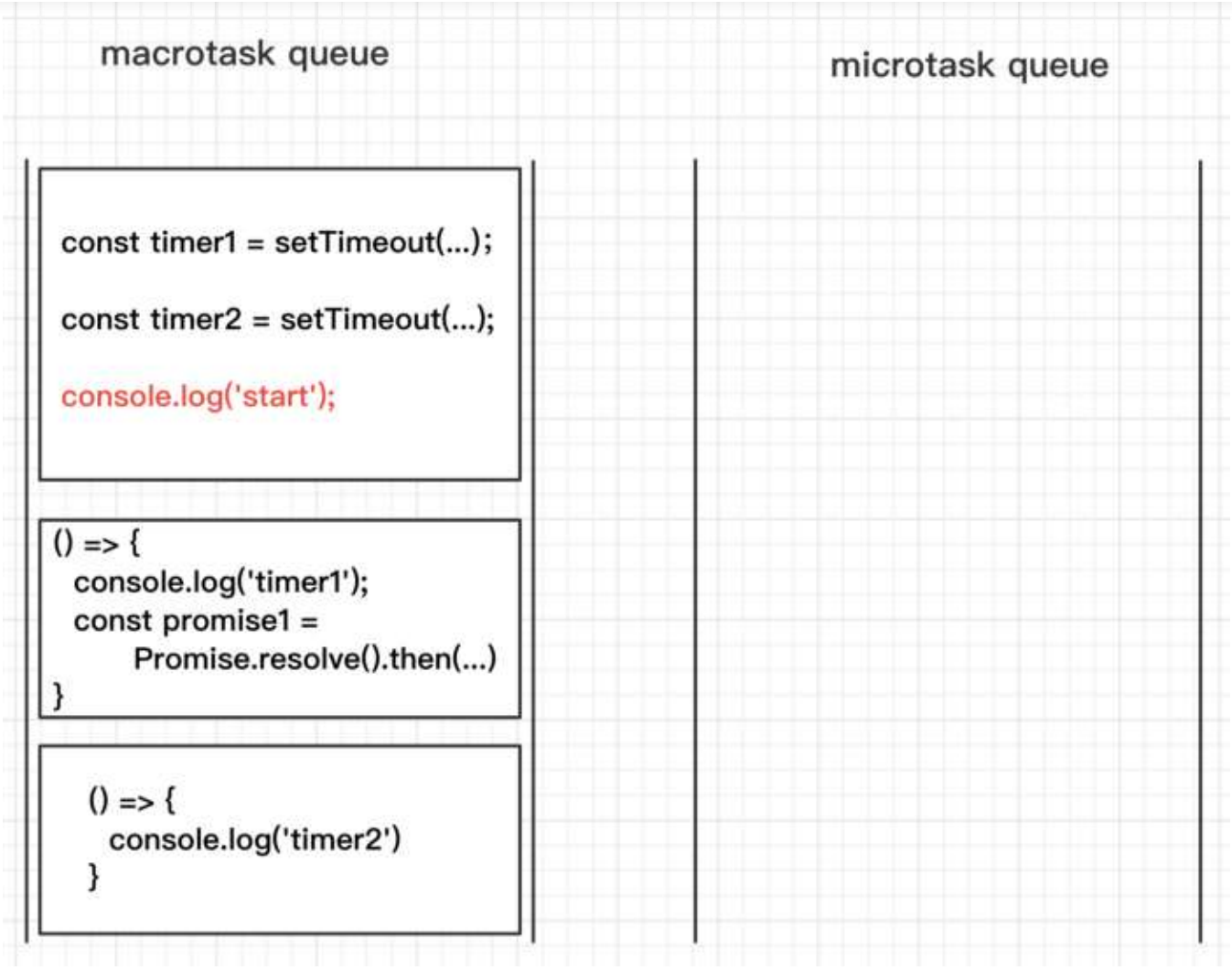
- Then



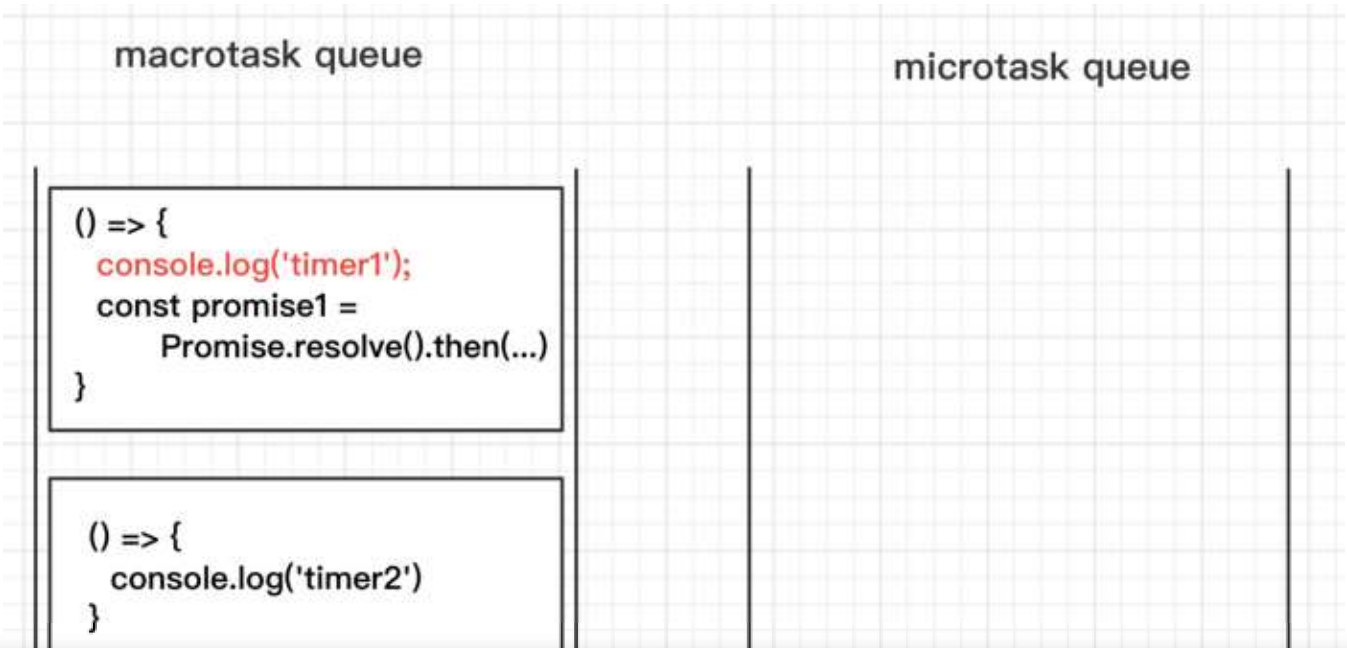
- Then



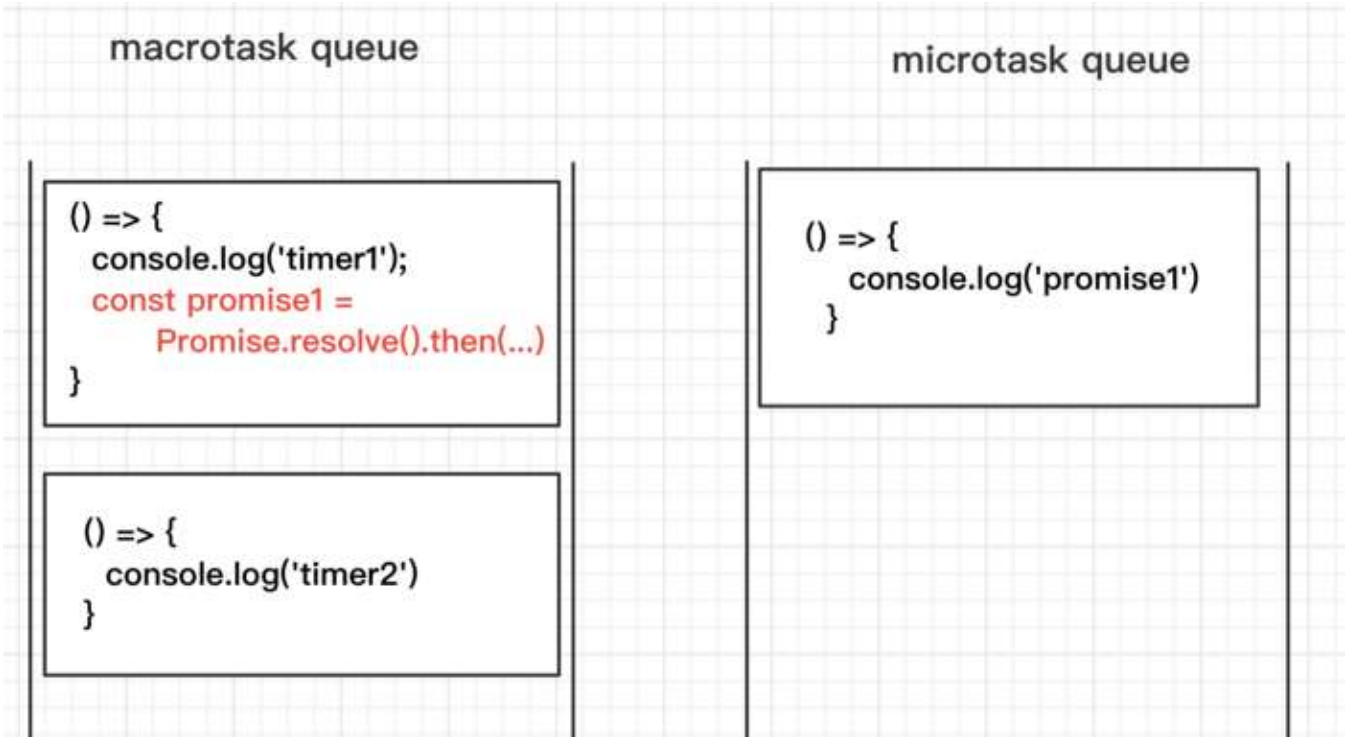
- Then



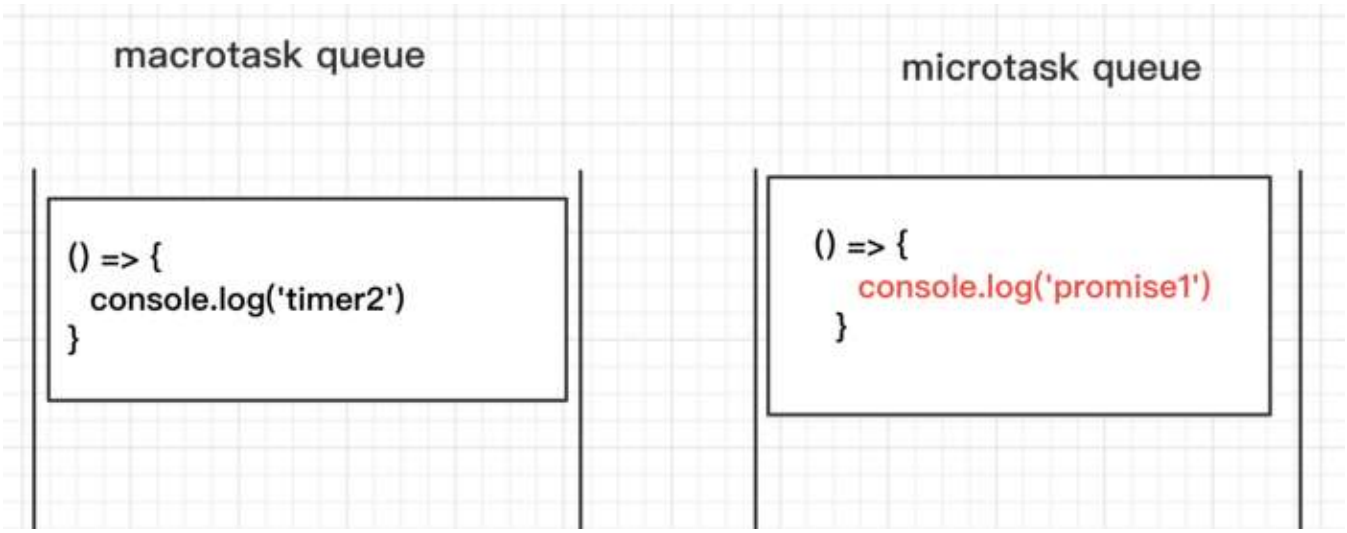
- Then



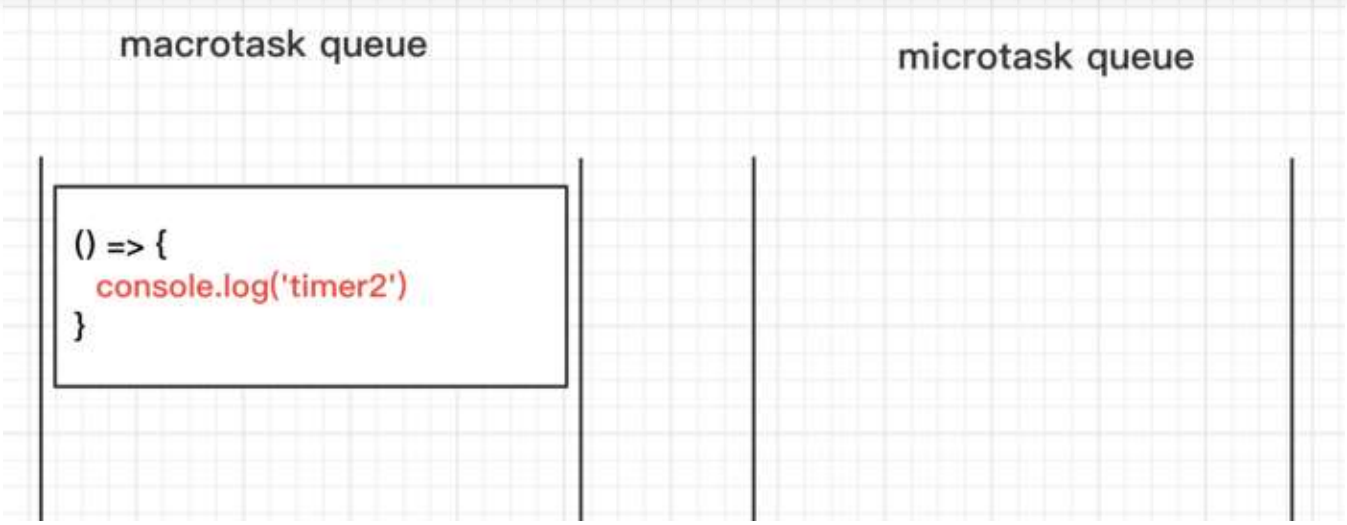
- Then



- Notice at this point that `Promise.then()` is creating a microtask. Its callback function is executed by the JS engine before the next macro task.



- Then the end.



Result:

```
> const timer1 = setTimeout(() => {
    console.log('timer1');
    const promise1 = Promise.resolve().then(() => {
      console.log('promise1')
    })
  }, 0)
  const timer2 = setTimeout(() => {
    console.log('timer2')
  }, 0)
  console.log('start')
  start
<· undefined
  timer1
  promise1
  timer2
```

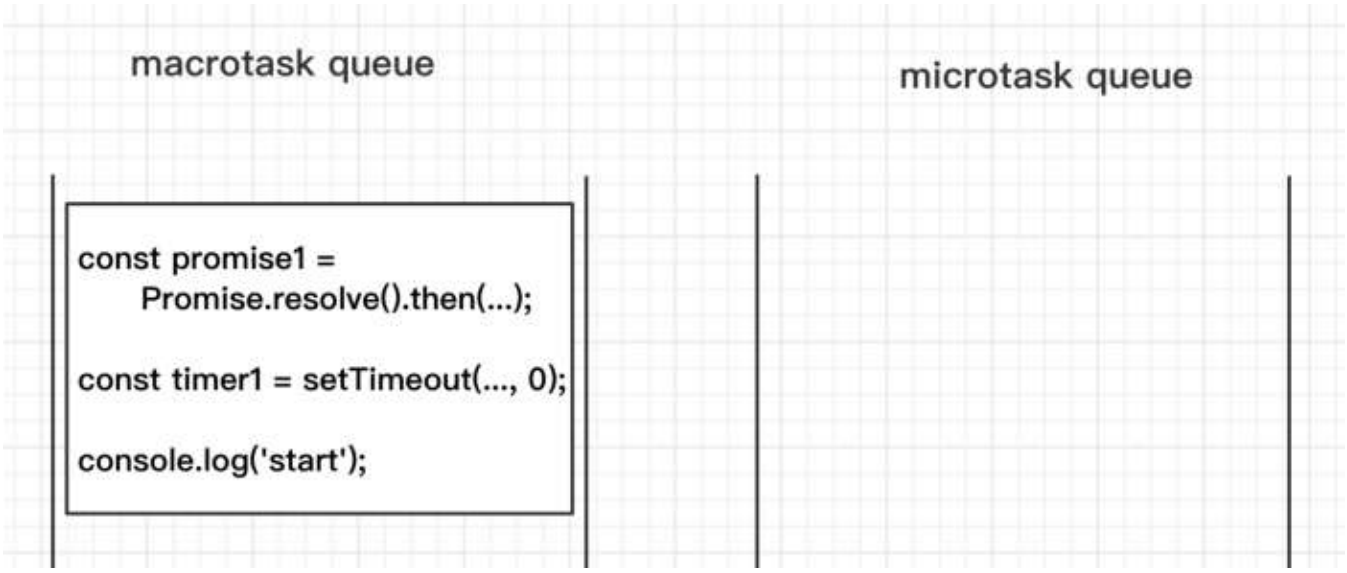
9.

Example:

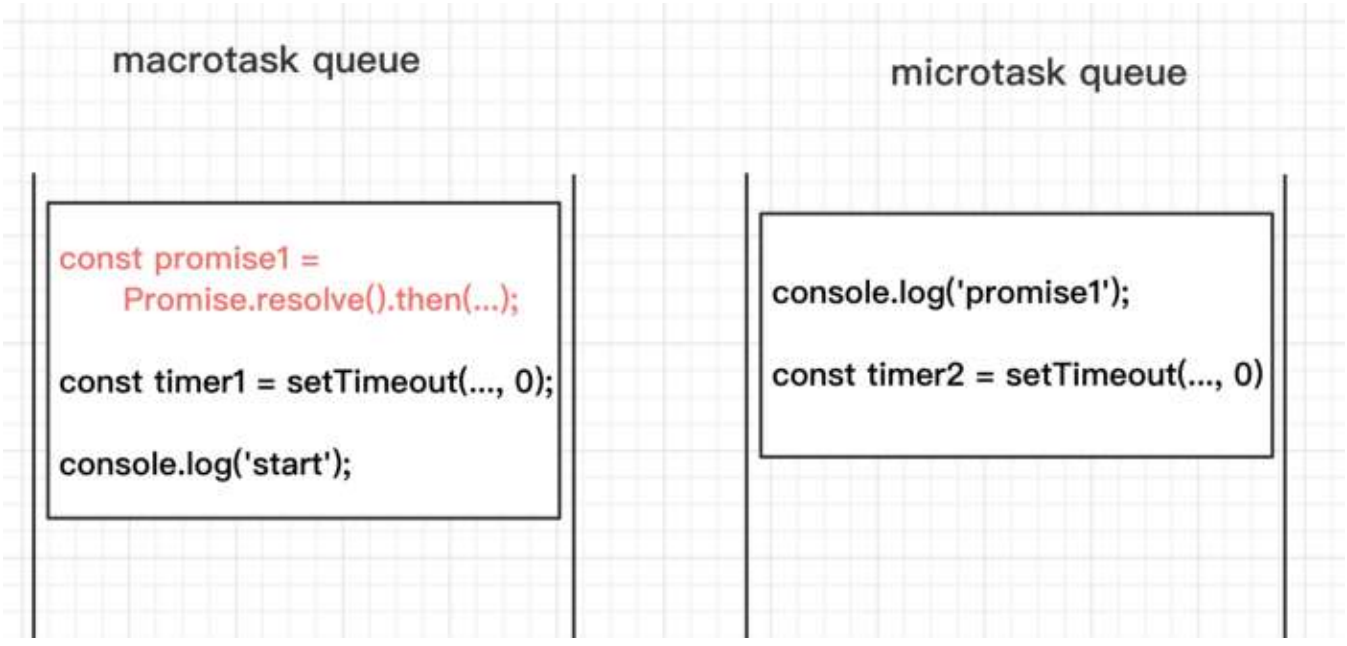
```
const promise1 = Promise.resolve().then(() => {
  console.log('promise1');
  const timer2 = setTimeout(() => {
    console.log('timer2')
  }, 0)
});

const timer1 = setTimeout(() => {
  console.log('timer1')
  const promise2 = Promise.resolve().then(() => {
    console.log('promise2')
  })
}, 0)

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


start drawing diagrams with me, it's easy to find the right answer.

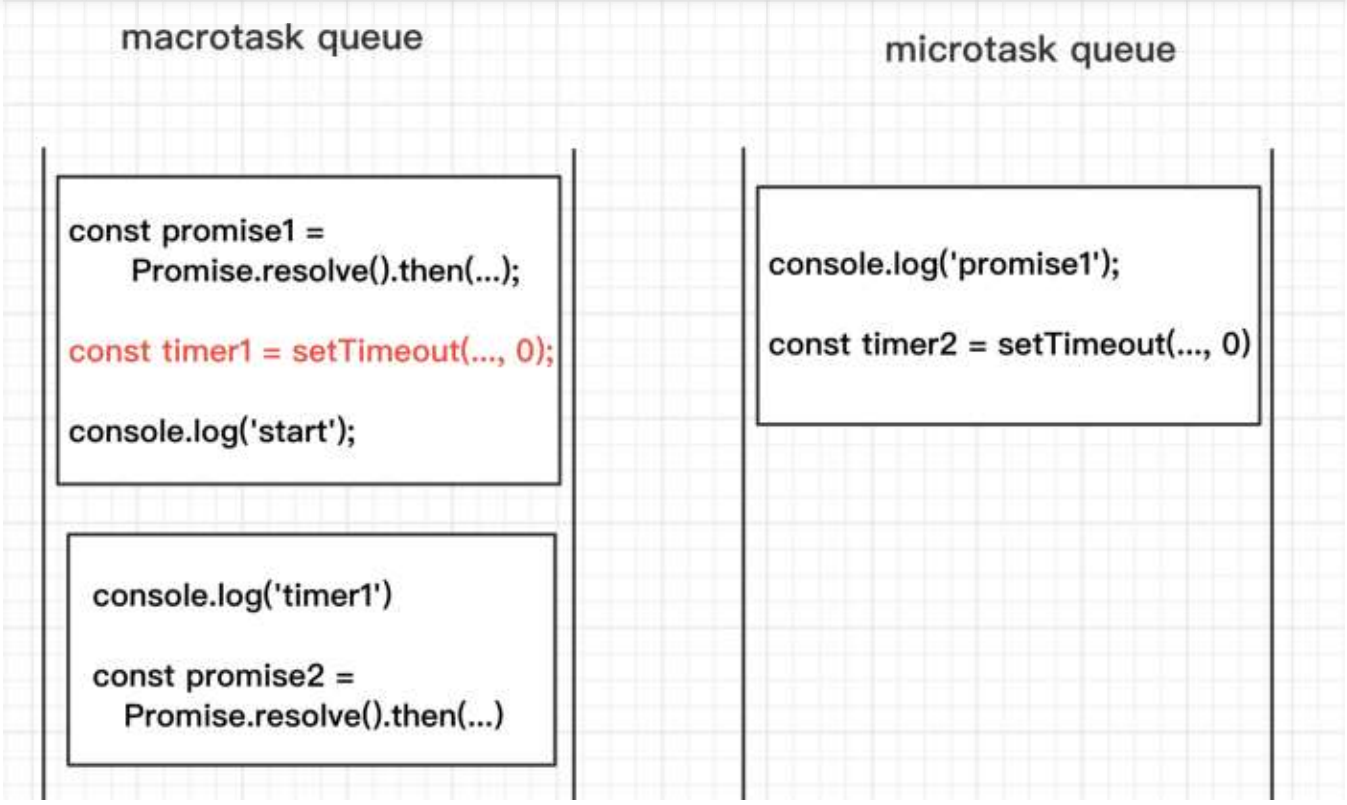
- First, let's draw the initial macro task queue.



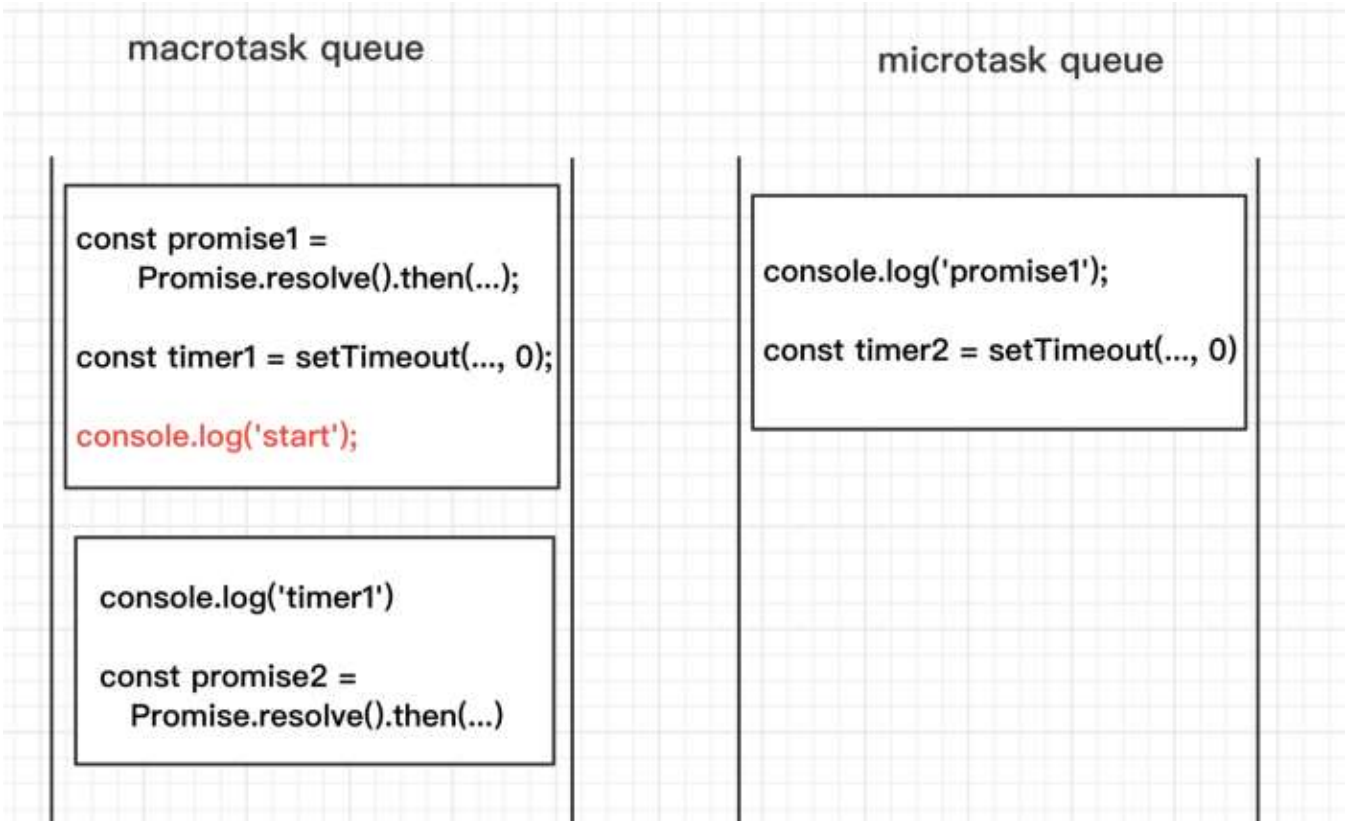
- The first piece of code is then executed, and a microtask is created.

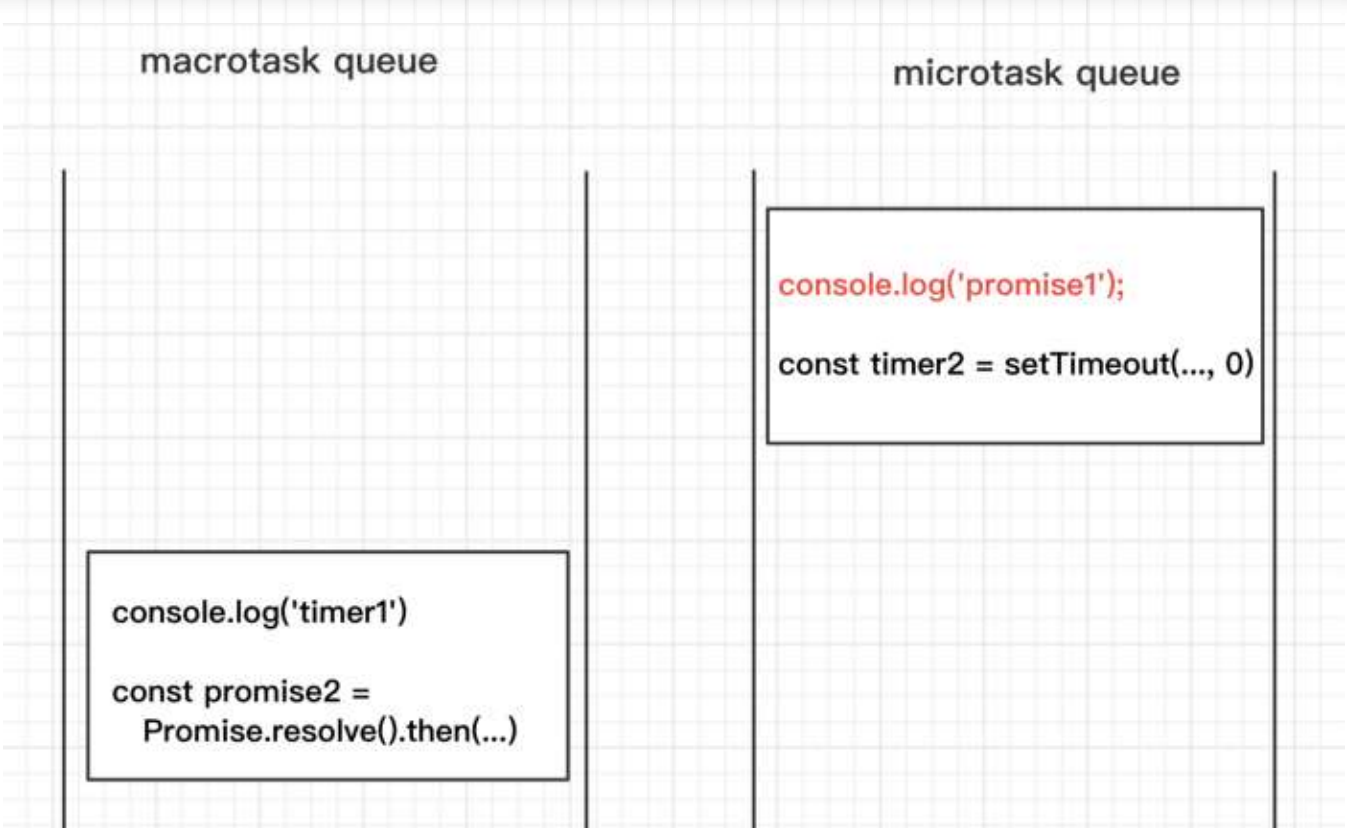


- The second piece of code is then executed, and a macro task is created

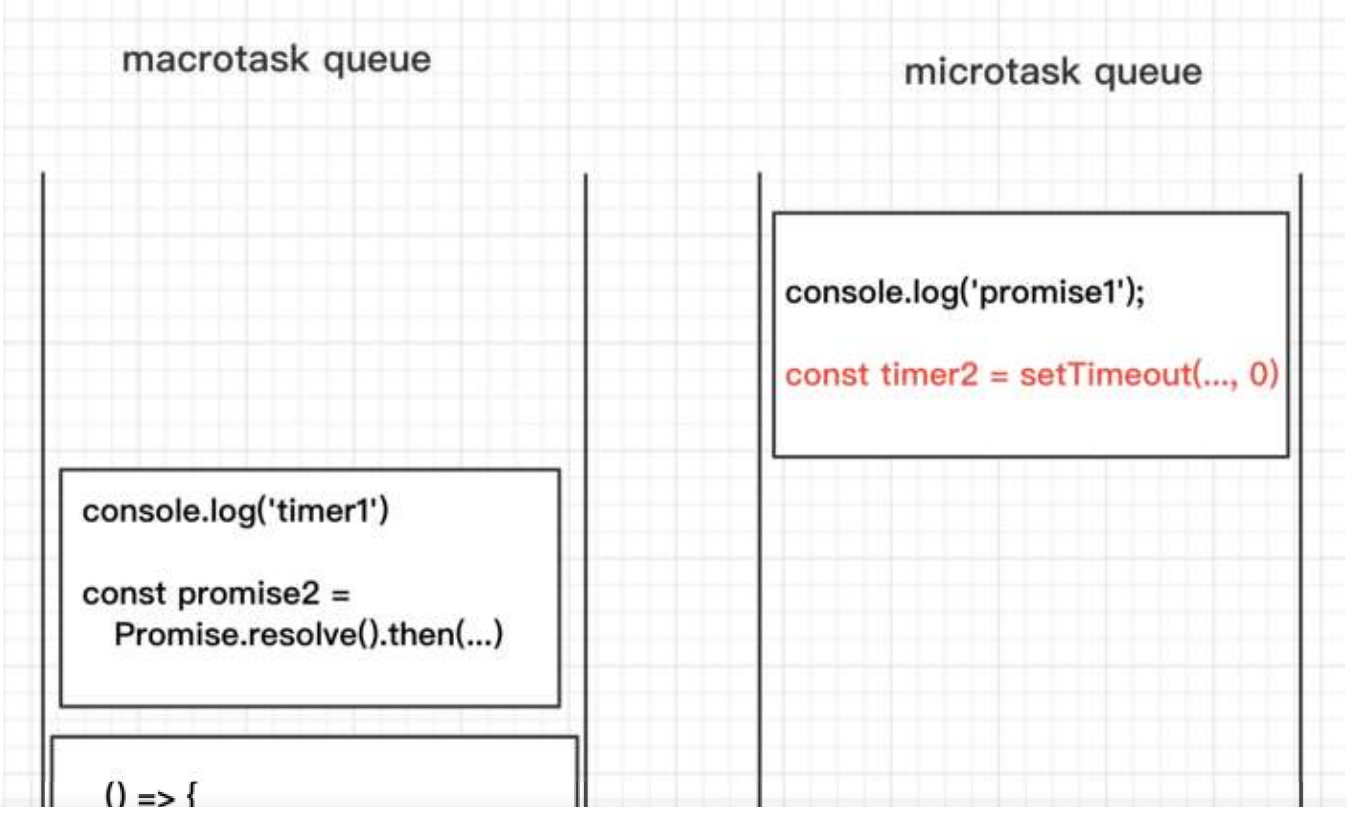


- Then

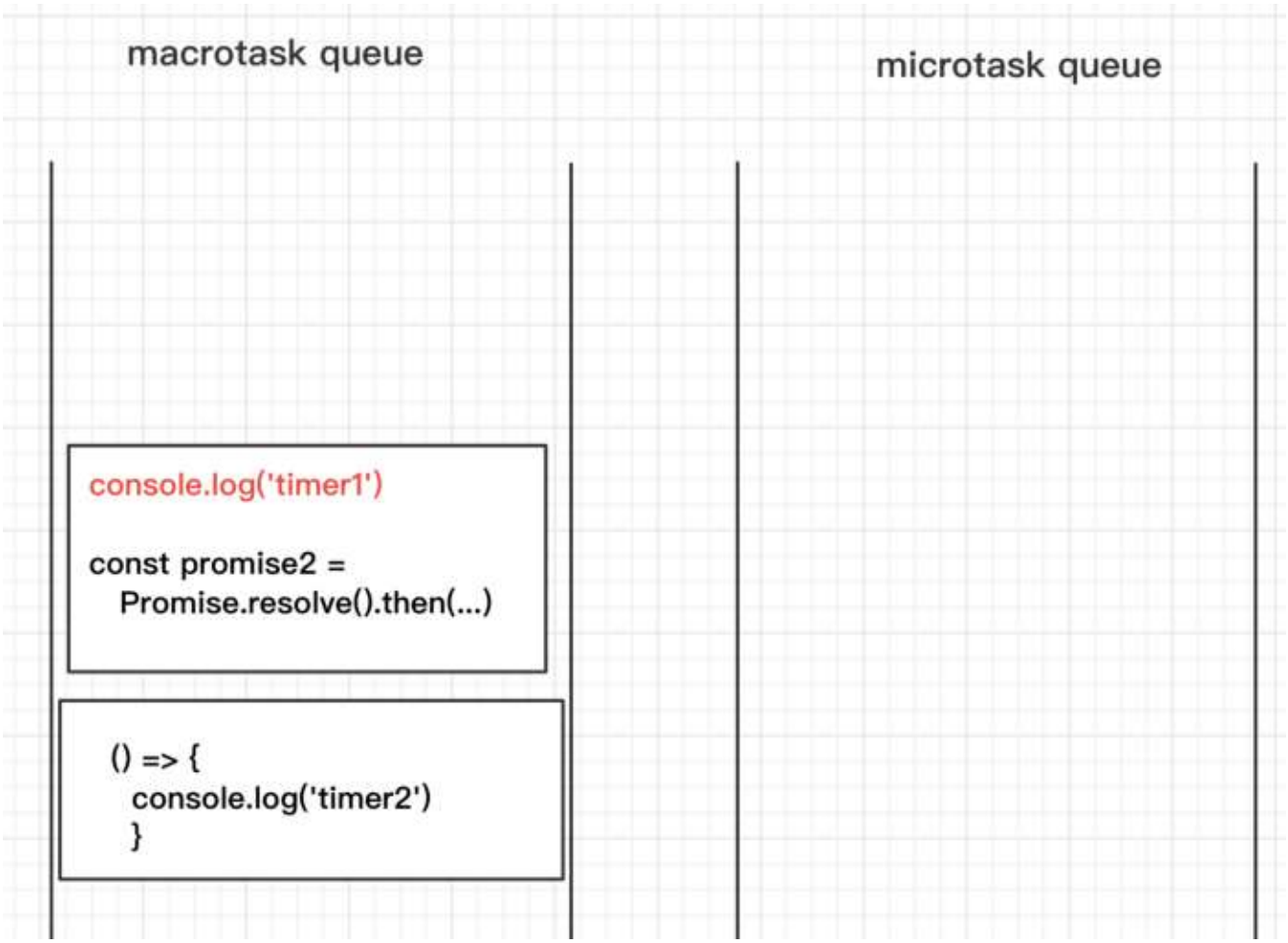




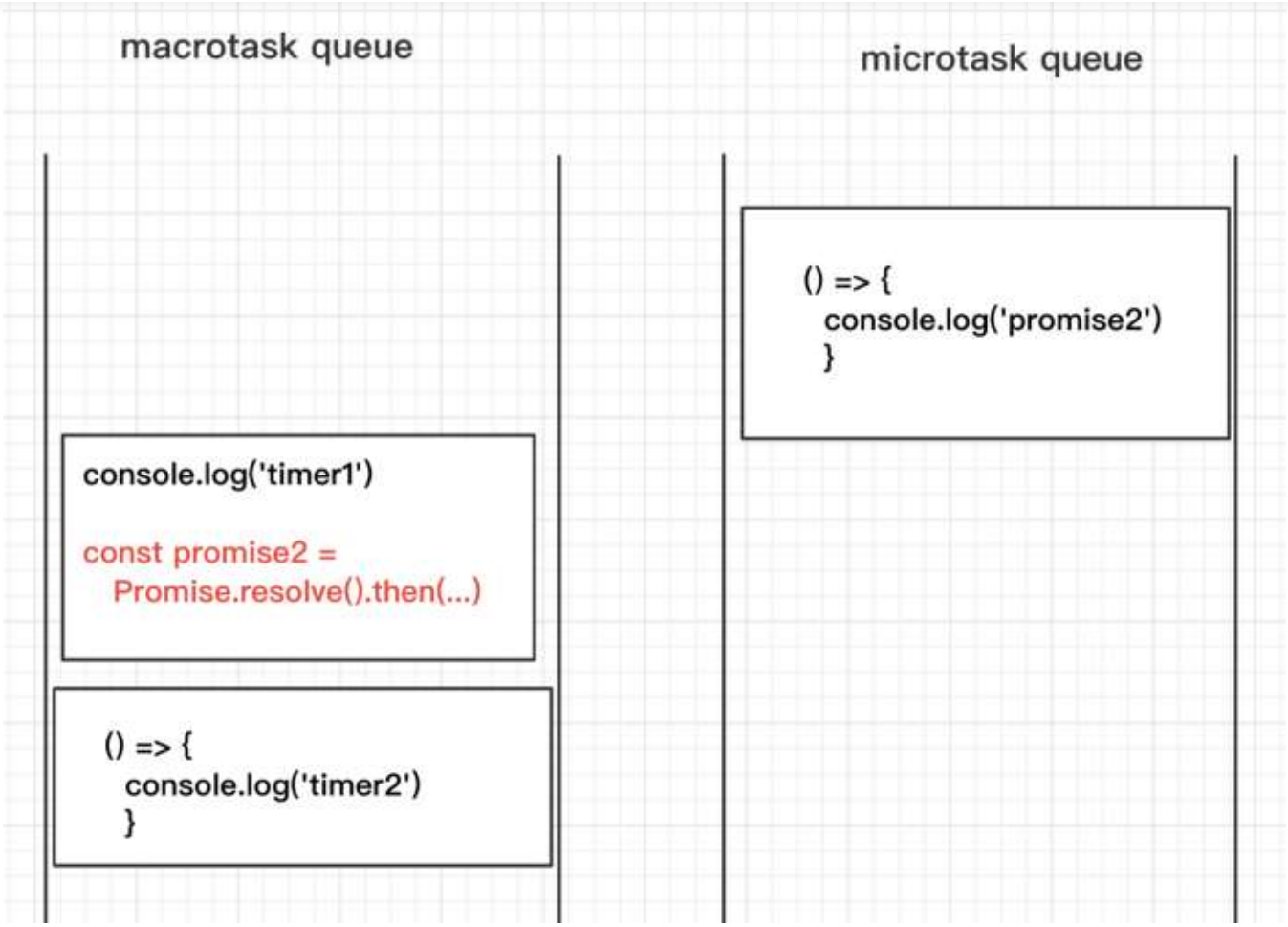
- Then, start executing `setTimeout(...)` relate to timer2 and create a new macro task



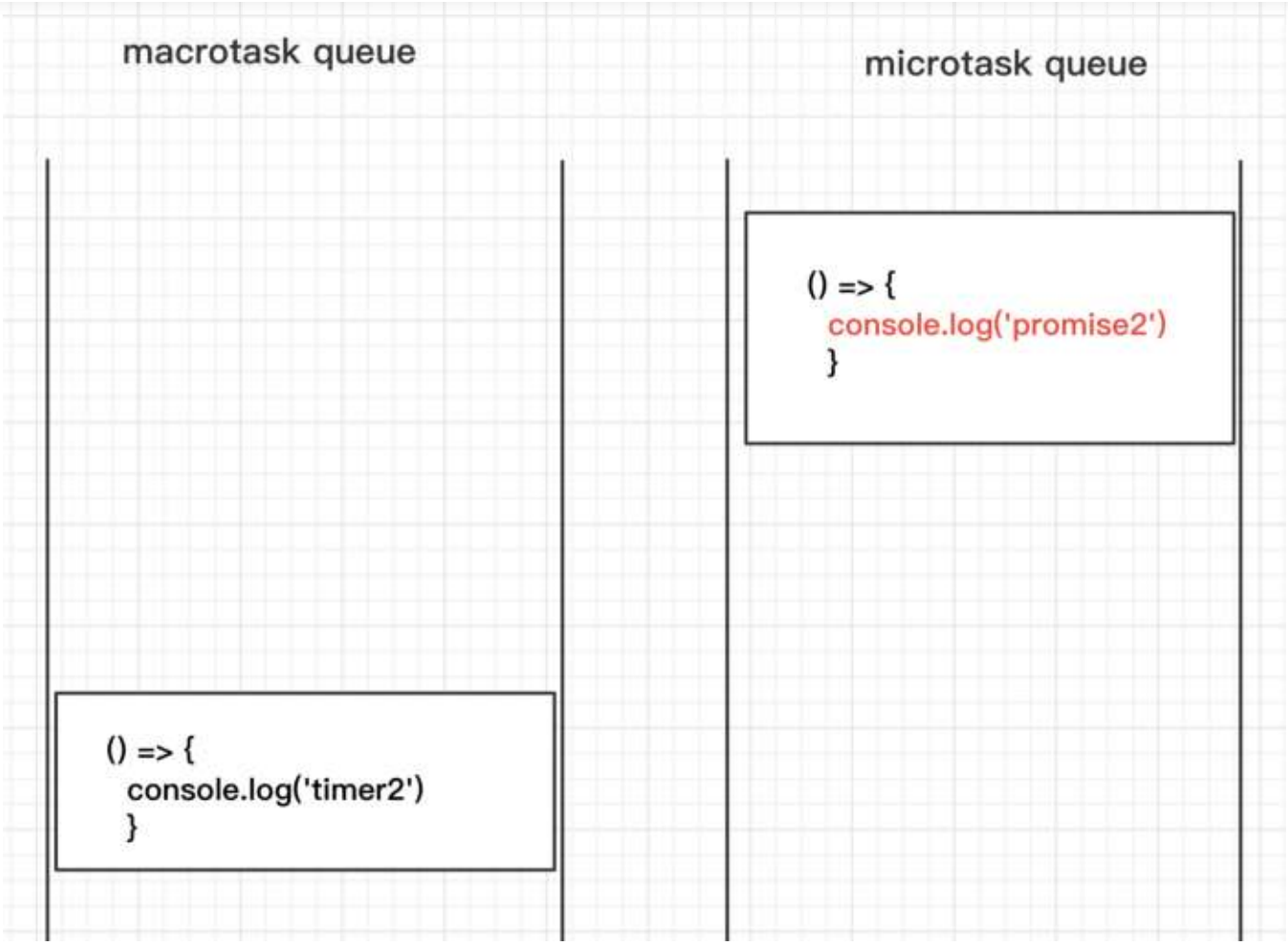
- The current microtask queue is cleared and the next macro task is started.



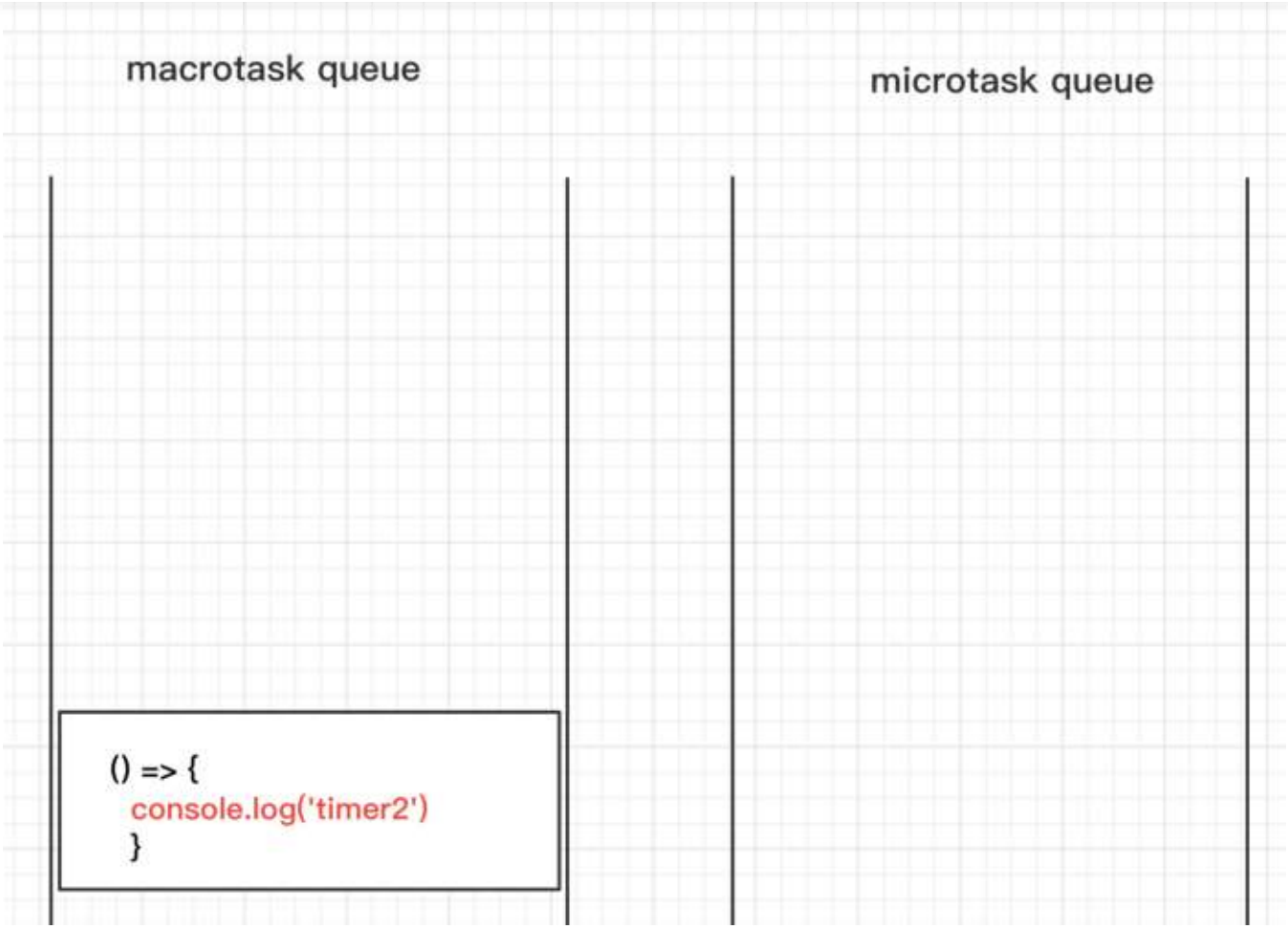
- Then, another microtask is created.



- The current macro task is completed. The JS engine checks the microtask queue again, finds that the queue is not empty, and starts prioritizing the tasks in the microtask queue.



- Finally



Result:

```
> const promise1 = Promise.resolve().then(() => {
    console.log('promise1');
    const timer2 = setTimeout(() => {
      console.log('timer2')
    }, 0)
  });
  const timer1 = setTimeout(() => {
    console.log('timer1')
    const promise2 = Promise.resolve().then(() => {
      console.log('promise2')
    })
  }, 0)
  console.log('start');
```

```
start
promise1
<· undefined
timer1
promise2
timer2
```

10.

Example:

```
const promise1 = new Promise((resolve, reject) => {
  const timer1 = setTimeout(() => {
    resolve('success')
  }, 1000)
})

const promise2 = promise1.then(() => {
  throw new Error('error!!!')
})

console.log('promise1', promise1)
console.log('promise2', promise2)

const timer2 = setTimeout(() => {
  console.log('promise1', promise1);
  console.log('promise2', promise2);
}, 2000)
```

Process Analysis:

- First, it created `promise1` through `new Promise(…)`, which is in the pending state. A timer with a delay of 1 second is also created.
- Then execute `const promise2 = promise1.then(...)`, because `promise1` is currently in a `Pending` state, so the callback function of `promise1.then()` won't be added to the microtask queue yet.


- Then execute `console.log('promise2', promise2)` . At this point, the `promise2` is still `Pending` .
- Then execute `const timer2 = setTimeout(…)` . A timer with a delay of 2 seconds is also created.
- After 1000 milliseconds, `timer1` was completed. then `resolve('success')` is then executed, and `promise1` becomes `resolved` .
- `promise1.then(...)` 's callback function is called, and `throw new Error('error!!!')` was executed. An error was throw and `promise2` became `reject` .
- After another 1000 milliseconds, `timer2` was completed. `() => {console.log('promise1', promise1); console.log('promise2', promise2);}` is executed.

Result:


```
> const promise1 = new Promise((resolve, reject) => {
    const timer1 = setTimeout(() => {
      resolve('success')
    }, 1000)
  })
  const promise2 = promise1.then(() => {
    throw new Error('error!!!')
  })
  console.log('promise1', promise1)
  console.log('promise2', promise2)
  const timer2 = setTimeout(() => {
    console.log('promise1', promise1);
    console.log('promise2', promise2);
  }, 2000)
```

```
promise1 ▶ Promise {<pending>}
promise2 ▶ Promise {<pending>}
<· undefined
✕ ▶ Uncaught (in promise) Error: error!!!
      at <anonymous>:7:9
promise1 ▶ Promise {<resolved>: "success"}
promise2 ▶ Promise {<rejected>: Error: error!!!
      at <anonymous>:7:9}
```