你真的了解Javascript执行顺序么?

原创：石头 TSG前端技术分享 1周前

前言

大家都知道，Javascript是单线程、顺序执行的，通过事件循环来处理异步。而且稍有开发经验的同学也知道，利用 `setTimeout`、 `setInterval` 以及 `Promise` 可以延时代码的执行。如果在Node.js中，大家会用 `process.nextTick` 来让代码在下一个周期执行；或者在Vue中，会利用 `Vue.nextTick` 保证DOM全部更新完毕后再执行回调函数。但是，如果他们都放在一起呢？执行顺序又会是怎么样的?

来个样例

聪明的你知道下面代码的执行顺序么?

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

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

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

看到这个问题，有经验的同学会脱口而出：太简单了，会输出如下内容：

```
script start
script end
setTimeout
```

因为 `setTimeout` 会加入到队列，延时执行。的确没错。那我们再看看下面的例子呢?

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

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

Promise.resolve().then(() => {
    console.log('promise1')
}).then(() => {
    console.log('promise2')
})

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

呃，这个问题好像难住了一部分同学，因为他们会有这样的想法：

> setTimeout和 Promise到底谁先执行呢？听说 Promise是异步的，但是 setTimeout也是异步的，而且延时为0， 这可怎么好？

想不明白？没关系，先执行下看看结果：

```
script start
script end
promise1
promise2
setTimeout
```

结果是不是蛮有意思的？为啥 Promise会先执行呢？我尝试着解释下，如果解释的不对，希望各位大牛多多指导。

大家知道，Javascript是基于事件循环(event loop)来处理事件的，用户的一些操作会放到事件队列里面，Javascript引擎会在合适的时候执行队列里面的操作。注意我们这里用到了“合适的时候“这个限定词，是因为Javascript单线程的，如果某段Javascript执行时间过长，那么它会阻塞主线程的执行。所以 setTimeout也并不说是一定会精确的执行。

在Javascript引擎里面，队列还分为 Task队列（也有人叫做 MacroTask）和 MicroTask队列， MicroTask会优先于 Task执行。比如常见的点击事件、 setImmediate、 setTimeout、 MessageChannel等会放入 Task队列，但是 Promise以及 MutationObserver会放到 Microtask队列。同时，

Javascript引擎在执行 Microtask 队列的时候，如果期间又加入了新的 Microtask，则该 Microtask 会加入到之前的 Microtask 队列的尾部，保证 Microtask 先于 Task 队列执行。

这样，大家就清楚了为啥 Promise 先执行吧，因为它是一个 Microtask 呀！优先级高，真是没办法 :-)。大家也许会问，优先级高，会高到什么程度呢？我们可以简单量度下：

```
const checkDuration = () => {
      const start = Date.now()
      let setTimeoutDuration = 0
      let promiseDuration = 0

      setTimeout(() => {
            setTimeoutDuration = Date.now() - start
      }, 0)

      Promise.resolve().then(() => {
            promiseDuration = Date.now() - start
      })

      setTimeout(() => {
            console.log(`setTimeout耗时: ${setTimeoutDuration}`)
            console.log(`Promise耗时: ${promiseDuration}`)
      }, 100)
}

checkDuration()
```

我在Chrome的console里面执行多次，会输出：

```
setTimeout耗时: 1
Promise耗时: 0

setTimeout耗时: 4
Promise耗时: 1
```

当然，如果这个结果不是固定的，测试多次， setTimeout执行大慨在4ms左右， Promise大慨在1ms左右。哈哈，其实就快了3ms，前端同学为了争取这3ms真是不懈努力而且煞费苦心呀，不过真的为他们爱专研的态度点赞！！！

值得说明的是， Vue中 Vue . nextTick也利用了该原理来保证在下次DOM更新循环结束之后执行延迟回调。如Vue 2.5.2里面就有这样的代码逻辑：

```
var microTimerFunc;
var macroTimerFunc;
var useMacroTask = false;

if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  macroTimerFunc = function () {
    setImmediate(flushCallbacks);
  };
} else if (typeof MessageChannel !== 'undefined' && (
  isNative(MessageChannel) ||
  // PhantomJS
  MessageChannel.toString() === '[object MessageChannelConstructor]'
)) {
  var channel = new MessageChannel();
  var port = channel.port2;
  channel.port1.onmessage = flushCallbacks;
  macroTimerFunc = function () {
    port.postMessage(1);
  };
} else {
  /* istanbul ignore next */
  macroTimerFunc = function () {
    setTimeout(flushCallbacks, 0);
  };
}

// Determine MicroTask defer implementation.
/* istanbul ignore next, $flow-disable-line */
if (typeof Promise !== 'undefined' && isNative(Promise)) {
  var p = Promise.resolve();
  microTimerFunc = function () {
```

```
        p.then(flushCallbacks);
        // in problematic UIWebViews, Promise.then doesn't completely break, but
        // it can get stuck in a weird state where callbacks are pushed into the
        // microtask queue but the queue isn't being flushed, until the browser
        // needs to do some other work, e.g. handle a timer. Therefore we can
        // "force" the microtask queue to be flushed by adding an empty timer.
        if (isIOS) { setTimeout(noop); }
    };
} else {
    // fallback to macro
    microTimerFunc = macroTimerFunc;
}
```

在Vue，用 MacroTask就是我们上文说的 Task。可见执行的时机是:

Task（MacroTask）队列中： setImmediate > MessageChannel > setTimeout MicroTask队列中： 直接用了 Promise，新版本中弃用了MutationObserver，因为其兼容性不好

扯了这么多，大家应该知道原因了吧?

再来个样例

为了巩固大家对 MicroTask的列举，我们再看一个例子

```
<div class="outer">
    <div class="inner"></div>
</div>

// 获取DOM
const outer = document.querySelector('.outer')
const inner = document.querySelector('.inner')

// 利用MuationObserver监听DOM的变化
new MutationObserver( () => {
    console.log('mutate')
```

```
}).observe(outer, {
    attributes: true
});

// 事件处理
const onClick = () => {
    console.log('click')

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

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

    outer.setAttribute('data-random', Math.random())
}

// 事件绑定
inner.addEventListener('click', onClick)
outer.addEventListener('click', onClick)
```

如果我们点击 inneer区域， 输出内容为什么呢？ 如果你理解了上文的内容，就会知道输出结果为：

```
click
promise
mutate
click
promise
mutate
timeout
timeout
```

好，今天就分享在这里。下篇文章，我们聊聊Node.js里面的事件。比如上文我们还没提到 `setImmediate` 呢？这个东西只在IE里面支持，但是在Node.js里面是支持的，而且Node.js里面还有一个 `Process.nextTick`。下次我们再聊聊。

参考资料

1. 什么是微任务与宏任务
2. Vue.nextTick源码阅读
3. Vue 中如何使用 MutationObserver 做批量处理?
4. Node.js Event Loop 的理解 Timers，process.nextTick()
5. what-is-the-event-loop
6. Process.nextTick 和 setImmediate 的区别?

阅读原文