这一次，彻底弄懂 JavaScript 执行机制

2017 年 11 月 21 日

本文的目的就是要保证你彻底弄懂javascript的执行机制，如果读完本文还不懂，可以揍我。

不论你是javascript新手还是老鸟，不论是面试求职，还是日常开发工作，我们经常会遇到这样的情况：给定的几行代码，我们需要知道其输出内容和顺序。因为javascript是一门单线程语言，所以我们可以得出结论：

- javascript是按照语句出现的顺序执行的

看到这里读者要打人了：我难道不知道js是一行一行执行的？还用你说？稍安勿躁，正因为js是一行一行执行的，所以我们以为js都是这样的：

```
let a = '1';
console.log(a);

let b = '2';
console.log(b);
```

然而实际上js是这样的：

```
setTimeout(function(){
    console.log('定时器开始啦')
});

new Promise(function(resolve){
    console.log('马上执行for循环啦');
    for(var i = 0; i < 10000; i++){
```

```
        i == 99 && resolve();
    }
}).then(function(){
    console.log('执行then函数啦')
});

console.log('代码执行结束');
```

依照js是按照语句出现的顺序执行这个理念，我自信的写下输出结果：

```
//"定时器开始啦"
//"马上执行for循环啦"
//"执行then函数啦"
//"代码执行结束"
```

去chrome上验证下，结果完全不对，瞬间懵了，说好的一行一行执行的呢？

我们真的要彻底弄明白javascript的执行机制了。

1.关于javascript

javascript是一门单线程语言，在最新的HTML5中提出了Web-Worker，但

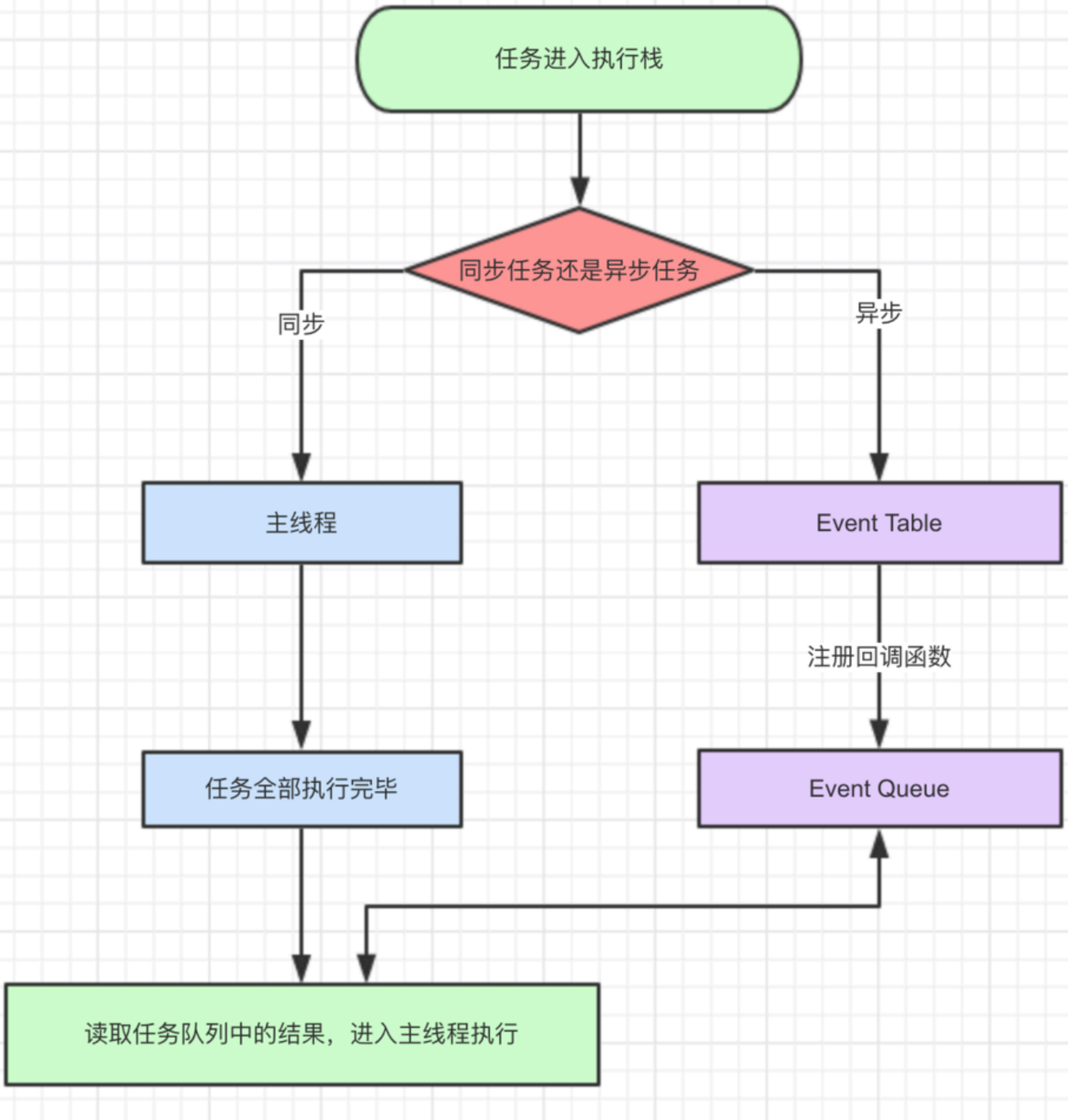
javascript是单线程这一核心仍未改变。所以一切javascript版的"多线程"都是用单线程模拟出来的，一切javascript多线程都是纸老虎！

2.javascript事件循环

既然js是单线程，那就像只有一个窗口的银行，客户需要排队一个一个办理业务，同理js任务也要一个一个顺序执行。如果一个任务耗时过长，那么后一个任务也必须等着。那么问题来了，假如我们想浏览新闻，但是新闻包含的超清图片加载很慢，难道我们的网页要一直卡着直到图片完全显示出来？因此聪明的程序员将任务分为两类：

- 同步任务
- 异步任务

当我们打开网站时，网页的渲染过程就是一大堆同步任务，比如页面骨架和页面元素的渲染。而像加载图片音乐之类占用资源大耗时久的任务，就是异步任务。关于这部分有严格的文字定义，但本文的目的是用最小的学习成本彻底弄懂执行机制，所以我们用导图来说明：



导图要表达的内容用文字来表述的话：

- 同步和异步任务分别进入不同的执行"场所"，同步的进入主线程，异步的进入Event Table并注册函数。
- 当指定的事情完成时，Event Table会将这个函数移入Event Queue。
- 主线程内的任务执行完毕为空，会去Event Queue读取对应的函数，进入主线程执行。
- 上述过程会不断重复，也就是常说的Event Loop(事件循环)。

我们不禁要问了，那怎么知道主线程执行栈为空啊？js引擎存在monitoring process进程，会持续不断的检查主线程执行栈是否为空，一旦为空，就会去Event Queue那里检查是否有等待被调用的函数。

说了这么多文字，不如直接一段代码更直白：

```
let data = [];
```

```
$.ajax({
    url:www.javascript.com,
    data:data,
    success:() => {
        console.log('发送成功!');
    }
})
console.log('代码执行结束');
```

上面是一段简易的ajax请求代码:

- ajax进入Event Table，注册回调函数success。
- 执行console.log('代码执行结束')。
- ajax事件完成，回调函数success进入Event Queue。
- 主线程从Event Queue读取回调函数success并执行。

相信通过上面的文字和代码，你已经对js的执行顺序有了初步了解。接下来我们来研究进阶话题：setTimeout。

3.又爱又恨的setTimeout

大名鼎鼎的setTimeout无需再多言，大家对他的第一印象就是异步可以延时执行，我们经常这么实现延时3秒执行:

```
setTimeout(() => {
    console.log('延时3秒');
},3000)
```

渐渐的setTimeout用的地方多了，问题也出现了，有时候明明写的延时3秒，实际却5，6秒才执行函数，这又咋回事啊?

先看一个例子:

```
setTimeout(() => {
    task();
},3000)
console.log('执行console');
```

根据前面我们的结论，setTimeout是异步的，应该先执行console.log这个同步任务，所以我们的结论是：

```
//执行console
//task()
```

去验证一下，结果正确！
然后我们修改一下前面的代码：

```
setTimeout(() => {
    task()
},3000)

sleep(10000000)
```

乍一看其实差不多嘛，但我们把这段代码在chrome执行一下，却发现控制台执行task()需要的时间远远超过3秒，说好的延时三秒，为啥现在需要这么长时间啊？

这时候我们需要重新理解setTimeout的定义。我们先说上述代码是怎么执行的：

- task()进入Event Table并注册,计时开始。
- 执行sleep函数，很慢，非常慢，计时仍在继续。
- 3秒到了，计时事件timeout完成，task()进入Event Queue，但是sleep也太慢了吧，还没执行完，只好等着。
- sleep终于执行完了，task()终于从Event Queue进入了主线程执行。

上述的流程走完，我们知道setTimeout这个函数，是经过指定时间后，把要执行的任务(本例中为task())加入到Event Queue中，又因为是单线程任务要一个一个执行，如果前面的任务需要的时间太久，那么只能等着，导致真正的延迟时间远远大于3秒。

我们还经常遇到setTimeout(fn,0)这样的代码，0秒后执行又是什么意思呢？是不是可以立即执行呢？

答案是不会的，setTimeout(fn,0)的含义是，指定某个任务在主线程最早可得的空闲时间执行，意思就是不用再等多少秒了，只要主线程执行栈内的同步任务全部执行完成，栈为空就马上执行。举例说明：

```
//代码1
console.log('先执行这里');
setTimeout(() => {
    console.log('执行啦')
},0);

//代码2
console.log('先执行这里');
setTimeout(() => {
    console.log('执行啦')
},3000);
```

代码1的输出结果是：

```
//先执行这里
//执行啦
```

代码2的输出结果是：

```
//先执行这里
// ... 3s later
// 执行啦
```

关于setTimeout要补充的是，即便主线程为空，0毫秒实际上也是达不到的。根据HTML的标准，最低是4毫秒。有兴趣的同学可以自行了解。

4.又恨又爱的setInterval

上面说完了setTimeout，当然不能错过它的孪生兄弟setInterval。他俩差不多，只不过后者是循环的执行。对于执行顺序来说，setInterval会每隔指定的时间将注册的函数置入Event Queue，如果前面的任务耗时太久，那么同样需要等待。

唯一需要注意的一点是，对于setInterval(fn,ms)来说，我们已经知道不是

每过ms秒会执行一次fn，而是每过ms秒，会有fn进入Event Queue。一旦**setInterval的回调函数fn执行时间超过了延迟时间ms，那么就完全看不出来有时间间隔了。**这句话请读者仔细品味。

5.Promise与process.nextTick(callback)

传统的定时器我们已经研究过了，接着我们探究Promise与process.nextTick(callback)的表现。

Promise的定义和功能本文不再赘述，不了解的读者可以学习一下阮一峰老师的Promise。而process.nextTick(callback)类似node.js版的"setTimeout"，在事件循环的下一次循环中调用 callback 回调函数。

我们进入正题，除了广义的同步任务和异步任务，我们对任务有更精细的定义：

- macro-task(宏任务)：包括整体代码script，setTimeout，setInterval
- micro-task(微任务)：Promise，process.nextTick

不同类型的任务会进入对应的Event Queue，比如setTimeout和setInterval会进入相同的Event Queue。

事件循环的顺序，决定js代码的执行顺序。进入整体代码(宏任务)后，开始第一次循环。接着执行所有的微任务。然后再次从宏任务开始，找到其中一个任务队列执行完毕，再执行所有的微任务。听起来有点绕，我们用文章最开始的一段代码说明：

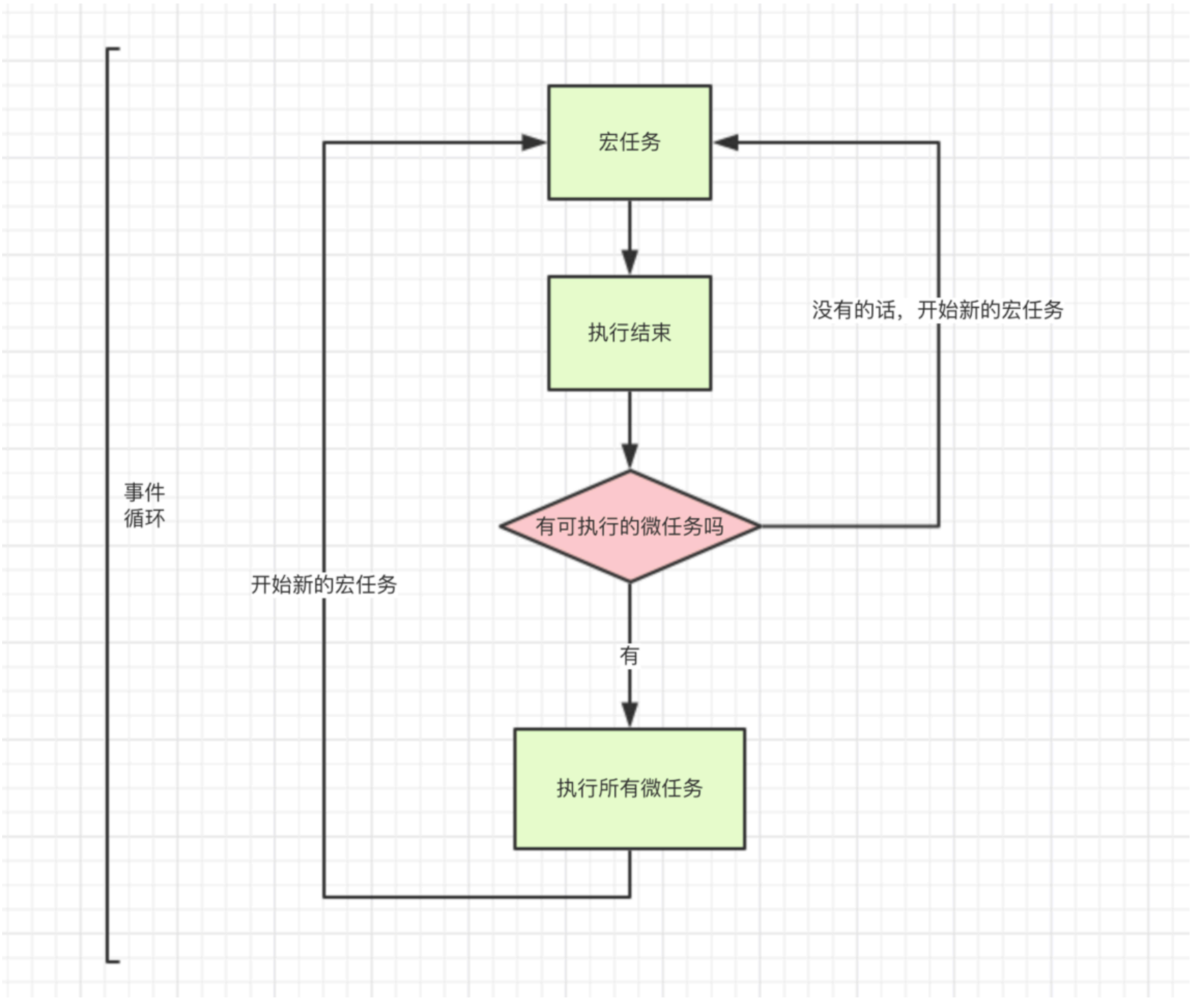
```
setTimeout(function() {
    console.log('setTimeout');
})

new Promise(function(resolve) {
    console.log('promise');
}).then(function() {
    console.log('then');
})

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

- 这段代码作为宏任务，进入主线程。
- 先遇到`setTimeout`，那么将其回调函数注册后分发到宏任务Event Queue。(注册过程与上同，下文不再描述)
- 接下来遇到了`Promise`，`new Promise`立即执行，`then`函数分发到微任务Event Queue。
- 遇到`console.log()`，立即执行。
- 好啦，整体代码script作为第一个宏任务执行结束，看看有哪些微任务？我们发现了`then`在微任务Event Queue里面，执行。
- ok，第一轮事件循环结束了，我们开始第二轮循环，当然要从宏任务Event Queue开始。我们发现了宏任务Event Queue中`setTimeout`对应的回调函数，立即执行。
- 结束。

事件循环，宏任务，微任务的关系如图所示：



我们来分析一段较复杂的代码，看看你是否真的掌握了js的执行机制：

```
console.log('1');

setTimeout(function() {
    console.log('2');
    process.nextTick(function() {
        console.log('3');
    })
    new Promise(function(resolve) {
        console.log('4');
        resolve();
    }).then(function() {
        console.log('5')
    })
})
process.nextTick(function() {
    console.log('6');
})
new Promise(function(resolve) {
    console.log('7');
    resolve();
}).then(function() {
    console.log('8')
})

setTimeout(function() {
    console.log('9');
    process.nextTick(function() {
        console.log('10');
    })
    new Promise(function(resolve) {
        console.log('11');
        resolve();
    }).then(function() {
        console.log('12')
    })
})
```

第一轮事件循环流程分析如下：

- 整体script作为第一个宏任务进入主线程，遇到`console.log`，输出1。
- 遇到`setTimeout`，其回调函数被分发到宏任务Event Queue中。我们暂且记为`setTimeout1`。

- 遇到`process.nextTick()`，其回调函数被分发到微任务Event Queue中。我们记为`process1`。
- 遇到`Promise`，`new Promise`直接执行，输出7。`then`被分发到微任务Event Queue中。我们记为`then1`。
- 又遇到了`setTimeout`，其回调函数被分发到宏任务Event Queue中，我们记为`setTimeout2`。

宏任务Event Queue	微任务Event Queue
setTimeout1	process1
setTimeout2	then1

- 上表是第一轮事件循环宏任务结束时各Event Queue的情况，此时已经输出了1和7。
- 我们发现了`process1`和`then1`两个微任务。
- 执行`process1`,输出6。
- 执行`then1`，输出8。

好了，第一轮事件循环正式结束，这一轮的结果是输出1，7，6，8。那么第二轮时间循环从`setTimeout1`宏任务开始：

- 首先输出2。接下来遇到了`process.nextTick()`，同样将其分发到微任务Event Queue中，记为`process2`。`new Promise`立即执行输出4，`then`也分发到微任务Event Queue中，记为`then2`。

宏任务Event Queue	微任务Event Queue
setTimeout2	process2
	then2

- 第二轮事件循环宏任务结束，我们发现有`process2`和`then2`两个微任务可以执行。
- 输出3。
- 输出5。
- 第二轮事件循环结束，第二轮输出2，4，3，5。

- 第三轮事件循环开始，此时只剩setTimeout2了，执行。
- 直接输出9。
- 将`process.nextTick()`分发到微任务Event Queue中。记为`process3`。
- 直接执行`new Promise`，输出11。
- 将`then`分发到微任务Event Queue中，记为`then3`。

宏任务Event Queue	微任务Event Queue
	process3
	then3

- 第三轮事件循环宏任务执行结束，执行两个微任务`process3`和`then3`。
- 输出10。
- 输出12。
- 第三轮事件循环结束，第三轮输出9，11，10，12。

整段代码，共进行了三次事件循环，完整的输出为1，7，6，8，2，4，3，5，9，11，10，12。
(请注意，node环境下的事件监听依赖libuv与前端环境不完全相同，输出顺序可能会有误差)

6.写在最后

(1)js的异步

我们从最开头就说javascript是一门单线程语言，不管是什么新框架新语法糖实现的所谓异步，其实都是用同步的方法去模拟的，牢牢把握住单线程这点非常重要。

(2)事件循环Event Loop

事件循环是js实现异步的一种方法，也是js的执行机制。

(3)javascript的执行和运行

执行和运行有很大的区别，javascript在不同的环境下，比如node，浏览器，Ringo等等，执行方式是不同的。而运行大多指javascript解析引擎，是统一

的。

(4)setImmediate

微任务和宏任务还有很多种类，比如`setImmediate`等等，执行都是有共同点的，有兴趣的同学可以自行了解。

(5)最后的最后

- javascript是一门单线程语言
- Event Loop是javascript的执行机制

牢牢把握两个基本点，以认真学习javascript为中心，早日实现成为前端高手的伟大梦想！

- 联系邮箱：ssssyoki@foxmail.com
- 联系微信：the-UK