我在学习浏览器的Event Loop时看了大量的文章，那些文章都写的很好，但是往往是每篇文章有那么几个关键的点，很多篇文章凑在一起综合来看，才可以对这些概念有较为深入的理解。

于是，我在看了大量文章之后，想要写这么一篇博客，不采用官方的描述，结合自己的理解以及示例代码，用最通俗的语言表达出来。希望大家可以通过这篇文章，了解到Event Loop到底是一种什么机制。如果在文中出现书写错误的地方，欢迎大家留言一起探讨。

Event Loop是什么？

**event loop是一个执行模型，在不同的地方有不同的实现。浏览器和NodeJS基于不同的技术实现了各自的Event Loop。本文浅谈下浏览器中的event loop。**浏览器的Event Loop是在[html5的规范](https://www.w3.org/TR/html5/webappapis.html#event-loops)中明确定义。

**宏队列和微队列**

**宏队列，macrotask，也叫tasks。** 一些异步任务的回调会依次进入macro task queue，等待后续被调用，这些异步任务包括：

* setTimeout
* setInterval
* setImmediate (Node独有)
* requestAnimationFrame (浏览器独有)
* I/O
* UI rendering (浏览器独有)

微队列，microtask，也叫jobs。 另一些异步任务的回调会依次进入micro task queue，等待后续被调用，这些异步任务包括：

* process.nextTick (Node独有)
* Promise(浏览器常见)
* Object.observe
* MutationObserver

浏览器的Event Loop

## browser-eventloop

## 这张图将浏览器的Event Loop完整的描述了出来，我来讲执行一个JavaScript代码的具体流程：

1. 执行全局Script同步代码，这些同步代码有一些是同步语句，有一些是异步语句（比如setTimeout等）；
2. 全局Script代码执行完毕后，调用栈Stack会清空
3. 从微队列microtask queue中取出位于队首的回调任务，放入调用栈Stack中执行，执行完后microtask queue长度减1；
4. 继续取出位于队首的任务，放入调用栈Stack中执行，以此类推，直到直到把microtask queue中的所有任务都执行完毕。注意，如果在执行microtask的过程中，又产生了microtask，那么会加入到队列的末尾，也会在这个周期被调用执行；
5. microtask queue中的所有任务都执行完毕，此时microtask queue为空队列，调用栈Stack也为空；
6. 取出宏队列macrotask queue中位于队首的任务，放入Stack中执行；
7. 执行完毕后，调用栈Stack为空；
8. 重复第3-7个步骤；
9. 重复第3-7个步骤；
10. ......

**可以看到，这就是浏览器的事件循环Event Loop**

这里归纳3个重点：

1. 宏队列macrotask一次只从队列中取一个任务执行，执行完后就去执行微任务队列中的任务；
2. 微任务队列中所有的任务都会被依次取出来执行，知道microtask queue为空；
3. 图中没有画UI rendering的节点，因为这个是由浏览器自行判断决定的，但是只要执行UI rendering，它的节点是在执行完所有的microtask之后，下一个macrotask之前，紧跟着执行UI render。

好了，概念性的东西就这么多，来看几个示例代码，测试一下你是否掌握了:

console.log(1);

setTimeout(() => {

console.log(2);

Promise.resolve().then(() => {

console.log(3)

});

})

new Promise((resolve, reject) => {

console.log(4)

resolve(5)

}).then((data) => {

console.log(data);

})

setTimeout(() => {

console.log(6);

})

console.log(7);

这里结果会是什么呢？运用上面了解到的知识，先自己做一下试试看。

// 正确答案

1

4

7

5

2

3

6

你答对了吗？

我们来分析一下整个流程：

**Step 1：**

console.log(1)

* Stack Queue: [console]
* Macrotask Queue: []
* Microtask Queue: []

打印结果：

1

**Step 2：**

setTimeout(() => {

// 这个回调函数叫做callback1，setTimeout属于macrotask，所以放到macrotask queue中

console.log(2);

Promise.resolve().then(() => {

console.log(3)

});

});

* Stack Queue: [setTimeout]
* Macrotask Queue: [callback1]
* Microtask Queue: []

打印结果：

4

**Step 3：**

new Promise((resolve, reject) => {

// 注意，这里是同步执行的，如果不太清楚，可以去看一下promise~~

console.log(4)

resolve(5)

}).then((data) => {

// 这个回调函数叫做callback2，promise属于microtask，所以放到microtask queue中

console.log(data);

})

* Stack Queue: [promise]
* Macrotask Queue: [callback1]
* Microtask Queue: [callback2]

打印结果：

1

4

**Step 4：**

setTimeout(() => {

// 这个回调函数叫做callback3，setTimeout属于macrotask，所以放到macrotask queue中

console.log(6);

})

* Stack Queue: [setTimeout]
* Macrotask Queue: [callback1, callback3]
* Microtask Queue: [callback2]

打印结果：

1

4

**Step 5：**

console.log(7)

* Stack Queue: [console]
* Macrotask Queue: [callback1, callback3]
* Microtask Queue: [callback2]

打印结果：

1

4

7

好啦，全局Script代码执行完了，进入下一个步骤，从microtask queue中依次取出任务执行，直到microtask queue队列为空。

**Step 6：**

console.log(data) // 这里data是Promise的决议值5

* Stack Queue: [callback2]
* Macrotask Queue: [callback1, callback3]
* Microtask Queue: []

打印结果：

1

4

7

5

这里microtask queue中只有一个任务，执行完后开始从宏任务队列macrotask queue中取位于队首的任务执行。

**Step 7：**

console.log(2)

* Stack Queue: [callback1]
* Macrotask Queue: [callback3]
* Microtask Queue: []

打印结果：

1

4

7

5

2

取出一个宏任务macrotask执行完毕，然后再去微任务队列microtask queue中依次取出执行

**Step 8：**

console.log(3)

* Stack Queue: [callback4]
* Macrotask Queue: [callback3]
* Microtask Queue: []

打印结果：

1

4

7

5

2

3

微任务队列全部执行完，再去宏任务队列中取第一个任务执

**Step 9：**

console.log(6)

* Stack Queue: [callback3]
* Macrotask Queue: []
* Microtask Queue: []

打印结果：

1

4

7

5

2

3

6

以上，全部执行完后，Stack Queue为空，Macrotask Queue为空，Micro Queue为空

* Stack Queue: []
* Macrotask Queue: []
* Microtask Queue: []

打印结果：

1

4

7

5

2

3

6

因为是第一个例子，所以这里分析的比较详细，大家仔细看一下，接下来我们再来一个例子：

console.log(1);

setTimeout(() => {

console.log(2);

Promise.resolve().then(() => {

console.log(3)

});

});

new Promise((resolve, reject) => {

console.log(4)

resolve(5)

}).then((data) => {

console.log(data);

Promise.resolve().then(() => {

console.log(6)

}).then(() => {

console.log(7)

setTimeout(() => {

console.log(8)

}, 0);

});

})

setTimeout(() => {

console.log(9);

})

console.log(10);

最终输出结果是什么呢？参考前面的例子，好好想一想......

// 正确答案

1

4

10

5

6

7

2

3

9

8

相信大家都答对了，这里的关键在前面已经提过：

在执行微队列microtask queue中任务的时候，如果又产生了microtask，那么会继续添加到队列的末尾，也会在这个周期执行，直到microtask queue为空停止。

注：当然如果你在microtask中不断的产生microtask，那么其他宏任务macrotask就无法执行了，但是这个操作也不是无限的，拿NodeJS中的微任务process.nextTick()来说，它的上限是1000个。浏览器的Event Loop就总结到此。