

# JavaScript的EventLoop

## 1. 区分进程和线程

官方描述：一个进程是一个工厂，不同的工厂之间相互独立，一个工厂有它的独立资源，线程是工厂中的工人，工厂内至少有一个或多个工人，人协作完成任务，工人之间共享空间。

官方术语：

- 进程是cpu资源分配的最小单位（是能拥有资源和独立运行的最小单位）。
- 线程是cpu调度的最小单位（线程是建立在进程的基础上的一次程序运行单位，线程是进程中的一个实体）。
- 一个进程中可以有多个线程，不同进程之间也可以通信，不过代价较大。
- 单线程与多线程，都是指在一个进程内的单和多（所以核心还是得属于一个进程才行）。

## 2. 浏览器的进程和线程

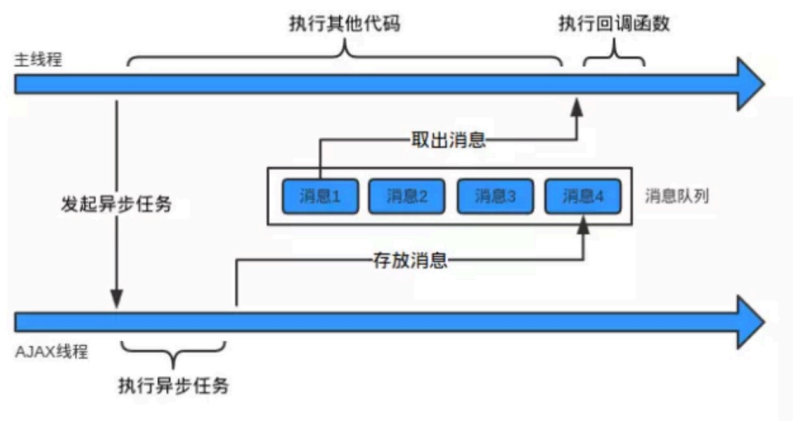
- 浏览器是多进程的（每开启一个网页至少会开一个进程，开启多个空白网页，默认认为是一个进程）
- 浏览器之所以能够运行，是因为系统给它的进程分配了资源（cpu、内存）
- 简单点理解，每打开一个Tab页，就相当于创建了一个独立的浏览器进程。
- 每个Tab页进程中又有多个线程：（1）javascript引擎线程（2）界面渲染线程（3）浏览器事件触发线程（4）HTTP请求线程

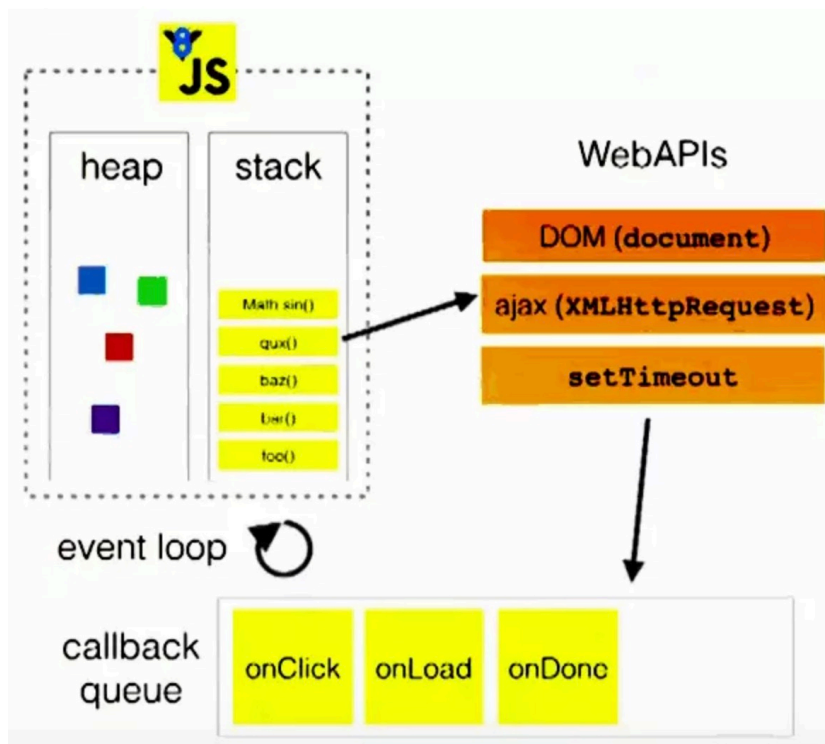
## 3. Javascript 为什么是单线程？

- 浏览器是用户进行操作触发的**事件驱动（Event driven）**，浏览器中很多行为是**异步（Async）**的，会创建事件并放入执行队列中。
- 假设JS有两个线程，一个线程在一个节点上添加内容，另一个线程又在这个节点上删除内容，这是浏览器应该以哪个线程为准？浏览器必须给发生事件的时间点排序。
- JS引擎是单线程处理它的任务队列，你可以理解成就是普通函数和回调函数构成的队列。当异步事件发生时，如mouse click, a timer fir XMLHttpRequest completing，将他们放入执行队列，等待当前代码执行完成。
- 单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等。

## 4. EventLoop（事件循环）

当一个异步事件发生的时候，它就进入事件队列。浏览器有一个内部大消息循环，**Event Loop（事件循环）**，会轮询大的事件队列并处理事件。如，浏览器当前正在忙于处理onclick事件，这时另外一个事件onSize发生了，这个异步事件就被放入事件队列等待处理，只有前面的处理完毕了才会执行这个事件。setTimeout也是一样，当调用的时候，JS引擎会启动定时器timer，大约xxms以后执行xxx，当定时器时间到，就把该事件队列等待处理（浏览器不忙的时候才会真正执行）。





- **stack**为自动分配的内存空间，它由系统自动释放；而**heap**则是动态分配的内存，大小不定也不会自动释放。
- **基本类型**：存放在栈内存中的简单数据段，数据大小确定，内存空间大小可以分配。5种基本数据类型有 **Undefined**、**Null**、**Boolean**、**Number** 和 **String**，它们是直接按值存放的，所以可以直接访问。
- **引用类型**：存放在堆内存中的对象，变量实际保存的是一个指针，这个指针指向另一个位置。每个空间大小不一样，要根据情况进行分配。

1. 执行栈执行主线程任务，当有 **操作dom**，**ajax交互**，**使用定时器**异步操作的时候，这些任务会被移入到 callback queue 任务队列中
2. 当主线程任务执行完毕为空时，会读取callback queue队列中的函数，进入主线程执行
3. 上述过程会不断重复，也就是常说的Event Loop(事件循环)。

## 5. macro task与micro task

在一个事件循环中，异步事件返回结果后会被放到一个任务队列中。然而，根据这个异步事件的类型，这个事件实际上会被对应的宏任务队列：任务队列中去，当执行栈为空的时候，主线程会首先查看微任务中的事件，如果微任务不是空的那么执行微任务中的事件，如果没有在宏任务中！面的一个事件。把对应的回调加入当前执行栈...如此反复，进入循环。

- macro-task(宏任务)： (1) setTimeout (2) setInterval (3) setImmediate
- micro-task(微任务)： (1) Promise (2) process.nextTick

代码一

```

1 function test1() {
2   console.log(1);
3 }
4
5 // T1-1
6 setTimeout(test1, 1000);
7 // (立即) 无输出
8 // (1s ->) 1
9
10
```

```

11 // T1-2
12 // setTimeout(test1(), 1000);
13 // (立即) 1
14 // (1s ->) 无输出
15
16
17 // T1-3
18 // setTimeout(console.log(1.1), 1000);
19 // (立即) 1.1
20 // (1s ->) 无输出
21
22
23 // ADD
24 // T1-4
25 // setTimeout('console.log(1.2)', 1000);
26 // (立即) 无输出
27 // (1s ->) 1.2

```

#### 代码二

```

1 function test2(value) {
2     value = value || 'default';
3     console.log(value);
4 }
5
6 // T2-1
7 setTimeout(test2, 1000, 2.1);
8 // (立即) 无输出
9 // (1s ->) 2.1
10
11
12 // T2-2
13 // setTimeout(test2(), 1000, 2.2);
14 // (立即) default
15 // (1s ->) 无输出
16
17
18 // T2-3
19 // setTimeout(test2(2.3), 1000, 2.31);
20 // (立即) 2.3
21 // (1s ->) 无输出

```

#### 代码三

```

1 function test3(value) {
2     value = value || 'default';
3     console.log(value);
4     return test3;
5 }
6
7 // T3-1
8 setTimeout(test3, 1000, 3.1);
9 // (立即) 无输出
10 // (1s ->) 3.1
11
12

```

```

13 // T3-2
14 // setTimeout(test3(), 1000, 3.2);
15 // (立即) default
16 // (1s ->) 3.2
17
18
19 // T3-3
20 // setTimeout(test3(3.3), 1000, 3.31);
21 // (立即) 3.3
22 // (1s ->) 3.31

```

#### 代码四

```

1 // T4-1
2 for (var i = 0; i < 5; i++) {
3     console.log(i);
4 }
5 // (立即) 0 1 2 3 4
6
7
8 // T4-2
9 // for (var i = 0; i < 5; i++) {
10 //     setTimeout(function () {
11 //         console.log(i);
12 //     }, 1000 * i);
13 // }
14 // (立即) 无输出
15 // (0s ->) 5
16 // (1s ->) 5
17 // (2s ->) 5
18 // (3s ->) 5
19 // (4s ->) 5
20
21
22 // T4-3
23 // for (var i = 0; i < 5; i++) {
24 //     setTimeout(function (i) {
25 //         console.log(i);
26 //     }, 1000 * i);
27 // }
28 // (立即) 无输出
29 // (0s ->) undefined
30 // (1s ->) undefined
31 // (2s ->) undefined
32 // (3s ->) undefined
33 // (4s ->) undefined
34
35
36 // ADD
37 // T4-3-2
38 // for (var i = 0; i < 5; i++) {
39 //     setTimeout(function (i) {
40 //         console.log(i);
41 //     }, 1000 * i, 999);
42 // }

```

```
43 // (立即) 无输出
44 // (0s ->) 0
45 // (1s ->) 0
46 // (2s ->) 0
47 // (3s ->) 0
48 // (4s ->) 0
49
50
51 // T4-4
52 // for (var i = 0; i < 5; i++) {
53 //     (function (i) {
54 //         setTimeout(function () {
55 //             console.log(i);
56 //         }, i * 1000);
57 //     })(i);
58 // }
59 // (立即) 无输出
60 // (0s ->) 0
61 // (1s ->) 1
62 // (2s ->) 2
63 // (3s ->) 3
64 // (4s ->) 4
65
66
67 // T4-5
68 // for (var i = 0; i < 5; i++) {
69 //     (function () {
70 //         setTimeout(function () {
71 //             console.log(i);
72 //         }, i * 1000);
73 //     })(i);
74 // }
75 // (立即) 无输出
76 // (0s ->) 5
77 // (1s ->) 5
78 // (2s ->) 5
79 // (3s ->) 5
80 // (4s ->) 5
81
82
83 // ADD
84 // T4-5-2
85 // for (var i = 0; i < 5; i++) {
86 //     (function () {
87 //         setTimeout(function () {
88 //             console.log(i);
89 //         }, i * 1000);
90 //     })();
91 // }
92 // (立即) 无输出
93 // (0s ->) 5
94 // (1s ->) 5
95 // (2s ->) 5
96 // (3s ->) 5
```

```

97 // (4s ->) 5
98
99
100 // T4-6
101 // for (var i = 0; i < 5; i++) {
102 //     setTimeout((function (i) {
103 //         console.log(i);
104 //     })(i), i * 1000);
105 // }
106 // (立即) 0 1 2 3 4
107 // (0s ->) 无输出
108 // (1s ->) 无输出
109 // (2s ->) 无输出
110 // (3s ->) 无输出
111 // (4s ->) 无输出
112
113
114 // ADD
115 // T4-6-2
116 // for (var i = 0; i < 5; i++) {
117 //     setTimeout((function () {
118 //         console.log(i);
119 //     })(), i * 1000);
120 // }
121 // (立即) 0 1 2 3 4
122 // (0s ->) 无输出
123 // (1s ->) 无输出
124 // (2s ->) 无输出
125 // (3s ->) 无输出
126 // (4s ->) 无输出
127
128
129 // ADD
130 // T4-6-3
131 // for (var i = 0; i < 5; i++) {
132 //     setTimeout((function () {
133 //         console.log(i);
134 //     })(), i * 1000);
135 // }
136 // (立即) 0 1 2 3 4
137 // (0s ->) 无输出
138 // (1s ->) 无输出
139 // (2s ->) 无输出
140 // (3s ->) 无输出
141 // (4s ->) 无输出

```

#### 代码五

```

1 function someLoop() {
2     var tag = true;
3     var temp = 0;
4
5     // 异步
6     setTimeout(function () {
7         tag = !tag;

```

```

8     console.log(tag)
9   }, 1000);
10
11   // 同步
12   while (tag) {
13     temp++;
14   }
15 }
16
17 someLoop();

```

#### 代码六

```

1  const times = 10000; // for阻塞耗时 < 200ms
2  // const times = 50000; // 200ms < for阻塞耗时 < 300ms:
3  // const times = 100000; // for阻塞耗时 > 300ms
4
5  console.log(1);
6
7  setTimeout(function () {
8    console.log(2);
9  }, 300);
10
11 setTimeout(function () {
12   console.log(3)
13 }, 400);
14
15 var start = new Date();
16 for (var i = 0; i < times; i++) {
17   console.log(4);
18 }
19 var end = new Date();
20 console.log('阻塞耗时: ' + Number(end - start) + '毫秒');
21
22 setTimeout(function () {
23   console.log(5);
24 }, 100);
25
26 // for阻塞耗时 < 200ms
27 // 1
28 // 4 4 4 ``
29 // 5
30 // 2
31 // 3
32
33 // 200ms < for阻塞耗时 < 300ms:
34 // 1
35 // 4 4 4 4 4 ``
36 // 2
37 // 5
38 // 3
39
40 // for阻塞耗时 > 300ms
41 // 1
42 // 4 4 4 4 4 ``

```

```
43 // 2
44 // 3
45 // 5
```

#### 代码七

```
1  setTimeout(() => {
2    console.log(1)
3  }, 0);
4
5  new Promise((resolve) => {
6    console.log(2);
7    for (let i = 0; i < 10000; i++) {
8      if (i === 9999) {
9        resolve();
10       console.log(i);
11     }
12   }
13   console.log(3);
14 }).then(() => {
15   console.log(4);
16 });
17
18 console.log(5);
19 // 2 9999 3 5 4 1
```