# **[for循环里的定时器引发的思考](https://www.cnblogs.com/hahazexia/p/9446585.html)**

在学习js的时候，或者面试的时候，会经常碰到这一道经典题目：

for(var i = 0; i < 5; i++) {

setTimeout(function () {

console.log(i);

});

}

console.log('a');

熟悉这道题目的人立马就可以说出答案：

'a'

5

5

5

5

5

结果是先打印字符串'a'，然后再打印5个数字5。

有人会说这个题目并不难，而且只要你遇到过这个题目，下次再见到基本也不会答错了，但其实这段简单的代码里面包含了很多js知识。

这里就整理总结一下。

**单线程、任务队列以及事件循环（event loop）**

第一次看到这段代码的时候，会给人一种错觉：

1. 会先打印for循环里面的5次i值，然后才会去打印下面的字符串'a'
2. for循环里面的打印结果会是0,1,2,3,4，而不是什么5个5这种奇怪的结果

但是实际运行结果跟我们预期的不一样，原因就是因为这里涉及到了js的运行机制。

单线程

JavaScript语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。

为什么不允许js可以实现多线程？因为如果实现了多线程，一个线程创建了一个div元素，而另外一个线程删除了这个div元素，那么这个时候浏览器应该听谁的？

所以为了避免出现这种互相冲突的操作，js从一开始就是单线程的，这就是它的核心特征。

任务队列

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着。

如果排队是因为计算量大，CPU忙不过来，倒也算了，但是很多时候CPU是闲着的，因为IO设备（输入输出设备）很慢（比如Ajax操作从网络读取数据），不得不等着结果出来，再往下执行。

JavaScript语言的设计者意识到，这时主线程完全可以不管IO设备，挂起处于等待中的任务，先运行排在后面的任务。等到IO设备返回了结果，再回过头，把挂起的任务继续执行下去。

于是，所有任务可以分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程、而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

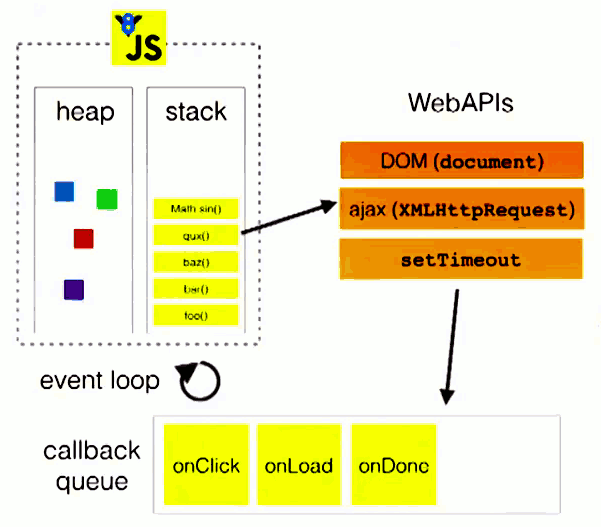
1. 所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。
2. 主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。
3. 一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
4. 主线程不断重复上面的第三步。

只要主线程空了，就会去读取"任务队列"，这就是JavaScript的运行机制。这个过程会不断重复。

"任务队列"中的事件，除了IO设备的事件以外，还包括一些用户产生的事件（比如鼠标点击、页面滚动等等）。只要指定过回调函数，这些事件发生时就会进入"任务队列"，等待主线程读取。

事件循环（event loop）

主线程从"任务队列"中读取事件，这个过程是循环不断的，所以整个的这种运行机制又称为Event Loop（事件循环）。



主线程运行的时候，产生堆（heap）和栈（stack），栈中的代码调用各种外部API，它们在"任务队列"中加入各种事件（click，load，done）。只要栈中的代码执行完毕，主线程就会去读取"任务队列"，依次执行那些事件所对应的回调函数。

**定时器**

在了解了刚才那些知识之后，再回过头来看看这段代码：

for(var i = 0; i < 5; i++) {

setTimeout(function () {

console.log(i);

});

}

console.log('a');

为什么明明定时器的时间设置为了0（setTimeout不写延迟时间参数默认值为0）？定时器却在console.log('a')这句代码运行了之后才运行？

原来在js的任务队列里，除了放置异步操作之外，还会放置定时器事件。

当js代码运行到有定时器的地方的时候，会把定时器操作放在任务队列的尾部，然后跟它说：“你先排队吧，还没有轮到你，因为同步代码还没有执行完。”

这里所说的 同步代码 就是指下面的console.log('a')。

也就是说，js认为setTimeout是一个异步操作，必须让它排队，它只能在同步代码执行结束后才能执行。

所以这里的原因总结就是这样一句话：

定时器并不是同步的，它会自动插入任务队列，等待当前文件的所有同步代码和当前任务队列里的已有事件全部运行完毕后才能执行。

这就是为什么字符串'a'在5个5之前就打印出来的原因。

那么为什么是5个5呢？为什么不是0,1,2,3,4？

这是因为在所有同步代码执行完毕之后，for循环里的i值早已变成了5，循环已经结束。（注意，for循环的圆括号部分也是同步代码）

这就是为什么打印出来5个5，而不是0,1,2,3,4。

所以这段代码真实的运行情况你可以假想成这样，便于理解：

[IMG_257](https://www.cnblogs.com/hahazexia/p/javascript:void(0);)

for(var i = 0; i < 5; i++) {

}

console.log('a');

setTimeout(function () {

　　console.log(i);

});

setTimeout(function () {

　　console.log(i);

});

setTimeout(function () {

　　console.log(i);

});

setTimeout(function () {

　　console.log(i);

});

setTimeout(function () {

　　console.log(i);

});//先循环，i变成了5，然后打印a，然后再打印5次i  
//这里只是假想，便于理解

[IMG_258](https://www.cnblogs.com/hahazexia/p/javascript:void(0);)

**作用域和闭包**

这道题目还会引申出来另一个问题：

如果想要for循环里的定时器打印出0,1,2,3,4，而不是5个5，该怎么办？

答案是：使用立即执行函数。

[IMG_259](https://www.cnblogs.com/hahazexia/p/javascript:void(0);)

for(var i = 0; i < 5; i++) {

(function(i) {

setTimeout(function () {

console.log(i);

});

})(i)

}

console.log('a');

[IMG_260](https://www.cnblogs.com/hahazexia/p/javascript:void(0);)

打印结果：

'a'

0

1

2

3

4

这又是为什么？

这是因为for循环里定义的i变量其实暴露在全局作用域内，于是5个定时器里的匿名函数它们其实共享了同一个作用域里的同一个变量。

所以如果想要0,1,2,3,4的结果，就要在每次循环的时候，把当前的i值单独存下来，怎么存下当前的循环i值？？

利用闭包的原理，闭包使一个函数可以继续访问它定义时的作用域。而这个新生成的作用域将每一次循环的当前i值单独保存了下来。

[IMG_261](https://www.cnblogs.com/hahazexia/p/javascript:void(0);)

for(var i = 0; i < 5; i++) {

(function(i) {//这个匿名函数生成了闭包的效果，新建了一个作用域，这个作用域接收到每次循环的i值保存了下来，即使循环结束，闭包形成的作用域也不会被销毁

setTimeout(function () {

console.log(i);

});

})(i)

}

[IMG_262](https://www.cnblogs.com/hahazexia/p/javascript:void(0);)

**let关键字、块作用域以及try...catch语句**

如果想实现for循环里的定时器打印出0,1,2,3,4，除了闭包，还可以使用ES6的let关键字。

for(let i = 0; i < 5; i++) {

setTimeout(function () {

console.log(i);

});

}

注意for循环定义i的时候把var换成了let，打印出的结果就是0,1,2,3,4

这是问什么呢？

因为let关键字劫持了for循环的块作用域，产生了类似闭包的效果。并且在for循环中使用let来定义循环变量还会有一个特殊效果：每一次循环都会重新声明变量i，随后的每个循环都会使用上一个循环结束时的值来初始化这个变量i。

let可以实现块作用域的效果，但是它是ES6语法，在低版本语法的时候如何生成块作用域？

答案是：使用try...catch语句。

看下面的效果：

[IMG_263](https://www.cnblogs.com/hahazexia/p/javascript:void(0);)

for(var i = 0; i < 5; i++) {

try {

throw(i)

} catch(j) {

setTimeout(function () {

console.log(j);

});

}

}

//打印结果0,1,2,3,4

[IMG_264](https://www.cnblogs.com/hahazexia/p/javascript:void(0);)

神奇的效果出现了！

这是因为try...catch语句的catch后面的花括号是一个块作用域，和let的效果一样。所以在try语句块里抛出循环变量i，然后在catch的块作用域里接收到传过来的i，就可以将循环变量保存下来，实现类似闭包和let的效果。

好了，这就是关于这道面试题涉及到的知识。