27 浏览器中的 Event-Loop

更新时间：2020-05-26 14:27:49

> 没有引发任何行动的思想都不是思想，而是梦想。 —— 马丁

同学们，我们现在来到了一个非常有趣的专题——事件循环（英文名Event-Loop）专题。

那些年，你做不对一道 Promise&setTimeout 输出顺序题，以为自己是不懂 Promise；答不出 Node 中 nextTick 和 Promise.resolve 的区别，以为自己是不懂 Node。千算万算，你没有算到自己竟然是输给了事件循环（笑）。

从本节开始，我们会以相对简单、也最最常考的浏览器事件循环机制为切入点，辅以最高频的几道面试题，帮助大家彻底掌握事件循环。

此外，很多同学在备考的过程中，会下意识地忽略 Node 这块考点。确实，如果你不是专业的 Node.js 工程师，很少会有面试官甩出大量 Node 题目来难为你。然而，Event-Loop 不一样，它太重要了，以至于面试官们普遍认为，不管你懂不懂 Node，你都必须懂 Node 中的事件机制。因此，我们在下一节，会着重来扒一扒 Node 的技术架构和事件机制。

从一道面试题说起

```
console.log(1)

setTimeout(function() {
  console.log(2)
})

new Promise(function (resolve) {
  console.log(3)
  resolve()
 }).then(function () {
  console.log(4)
}).then(function() {
  console.log(5)
})

console.log(6)
```

大家先调动自己现有的知识思考一下：上述代码的输出结果是什么？

答案是：1、3、6、4、5、2

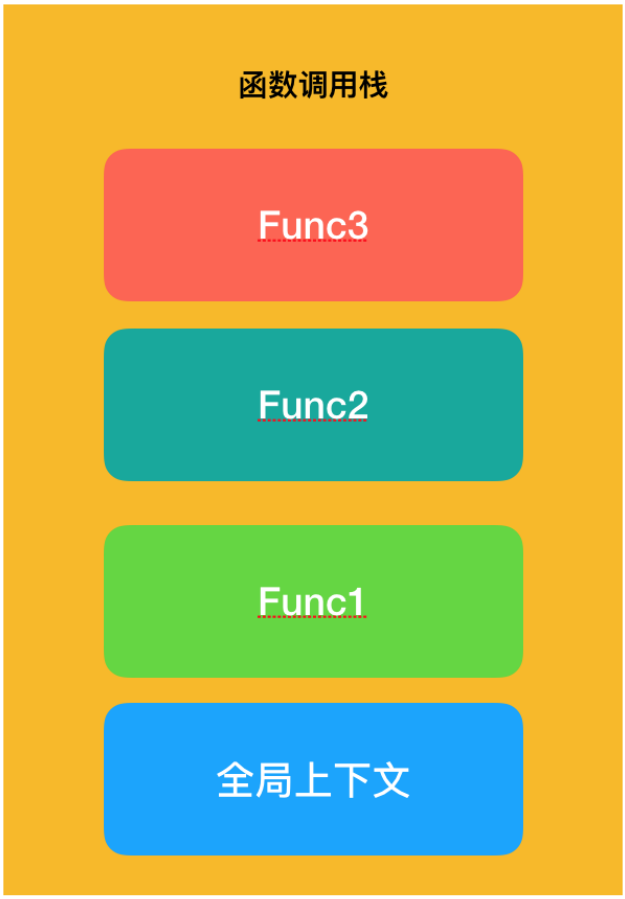
如果你能够准确给出上面的回答、并且说出你的依据，那么恭喜你——你的事件循环基础很扎实，可以直接跳至真题部分开始刷题了；如果你的答案和上面不一致，也不要着急，这个输出顺序是由浏览器的事件循环规则决定的。我们接下来就拿它开刀：

浏览器中的 Event-Loop 机制解析

关键角色剖析

在浏览器的事件循环中，首先大家要认清楚 3 个角色：**函数调用栈**、**宏任务（macro-task)队列**和**微任务(micro-task)队列**。

函数调用栈大家都很熟悉了（咱们在开篇第一个大章节就讲过）：当引擎第一次遇到 JS 代码时，会产生一个全局执行上下文并压入调用栈。后面每遇到一个函数调用，就会往栈中压入一个新的函数上下文。JS引擎会执行栈顶的函数，执行完毕后，弹出对应的上下文：



一句话：如果你有一坨需要被执行的逻辑，它首先需要被推入函数调用栈，后续才能被执行。**函数调用栈是个干活的地方，它会真刀真枪地给你执行任务**。

那么宏任务队列、微任务队列又是啥呢？

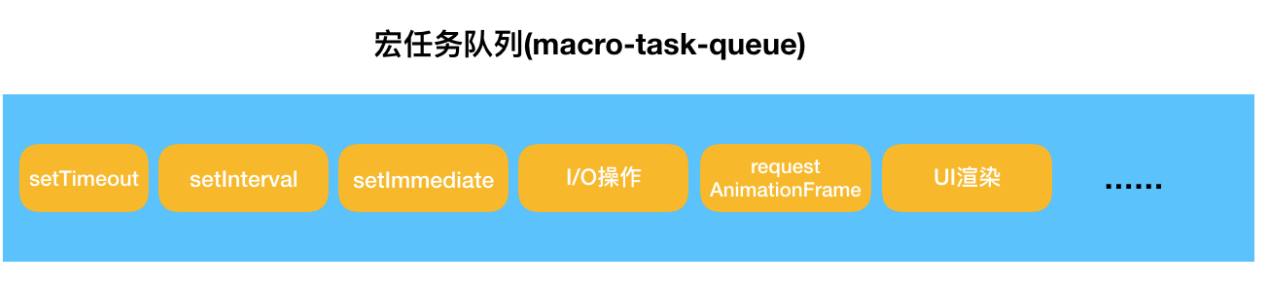
各位知道，JS 的特性就是单线程+异步。在JS中，咱们有一些任务，比如说上面咱们塞进 setTimeout 里那个任务，再比如说你在 Promise 里面塞进 then 里面那个任务——这些任务是异步的，它们不需要立刻被执行，所以它们在刚刚被派发的时候，**并不具备进入调用栈的“资格”**。

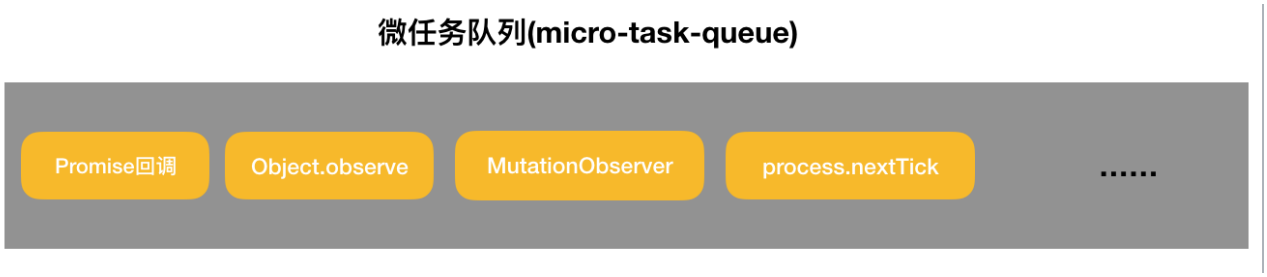
这暂时没资格咋整呢？

排队等呗！

于是这些待执行的任务，按照一定的规则，乖乖排起长队，等待着被推入调用栈的时刻到来——这个队列，就叫做“**任务队列**”。

所谓“宏任务”与“微任务”，是对任务的进一步细分。具体的划分依据如图所示：





常见的 macro-task 比如： setTimeout、setInterval、 setImmediate、 script（整体代码）、I/O 操作等。

常见的 micro-task 比如: process.nextTick、Promise、MutationObserver 等

注意：script（整体代码）它也是一个宏任务；此外，宏任务中的 setImmediate、微任务中的 process.nextTick 这些都是 Node 独有的。

循环过程解读

基于对 micro 和 macro 的认知，我们来走一遍完整的事件循环过程。

一个完整的 Event Loop 过程，可以概括为以下阶段：

1. 执行并出队一个 macro-task。注意如果是初始状态：调用栈空。micro 队列空，macro 队列里有且只有一个 script 脚本（整体代码）。这时首先执行并出队的就是 script 脚本；
2. 全局上下文（script 标签）被推入调用栈，同步代码执行。在执行的过程中，通过对一些接口的调用，可以产生新的 macro-task 与 micro-task，它们会分别被推入各自的任务队列里。**这个过程本质上是队列的 macro-task 的执行和出队的过程**；
3. 上一步我们出队的是一个 macro-task，这一步我们处理的是 micro-task。但需要注意的是：当 macro-task 出队时，任务是一个一个执行的；而 micro-task 出队时，任务是一队一队执行的（如下图所示）。因此，我们处理 micro 队列这一步，会逐个执行队列中的任务并把它出队，直到队列被清空；
4. 执行渲染操作，更新界面；
5. 检查是否存在 Web worker 任务，如果有，则对其进行处理。

这里我给大家列出的5步，是相对完整的过程。其实，针对面试，咱们关注第1-3步就足够了。第4步第5步，面试时说了没错，不说也没人会难为你，不必较劲。

真题重做，逐行分析

现在咱们基于对这个过程的理解，重新做一遍开篇那道题：

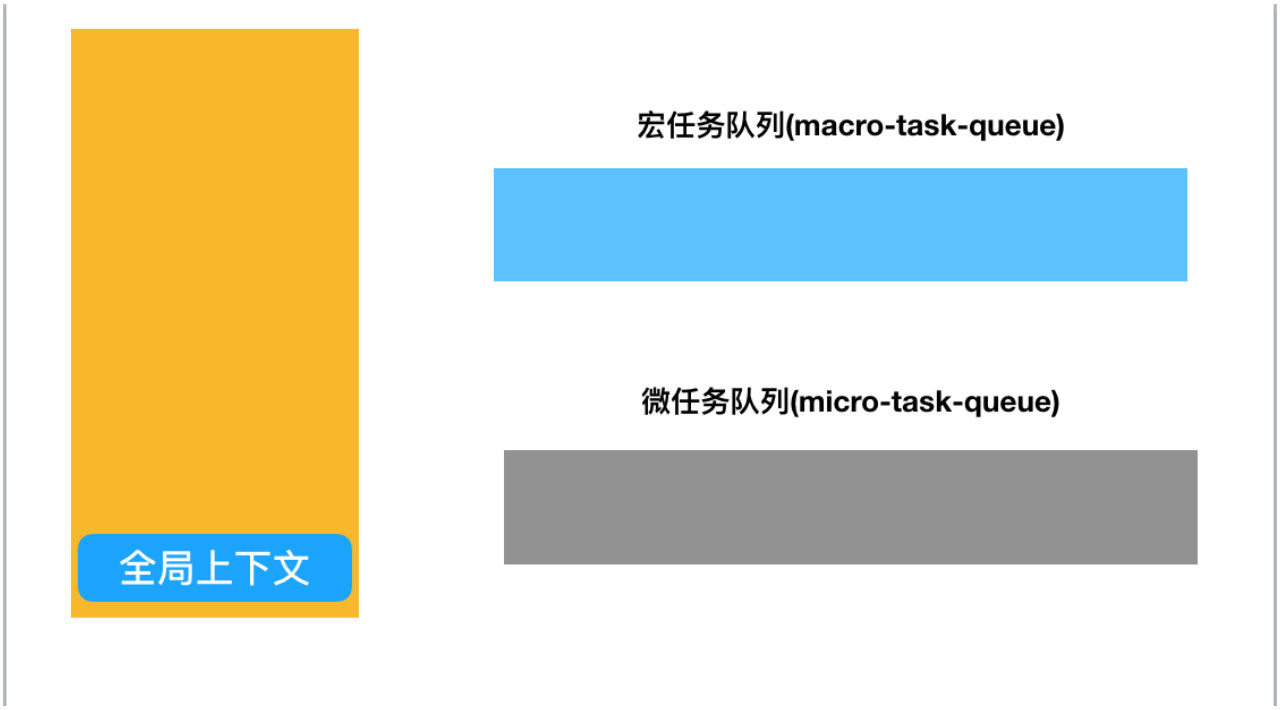
```
console.log(1)

setTimeout(function() {
  console.log(2)
})

new Promise(function (resolve) {
  console.log(3)
  resolve()
 }).then(function () {
  console.log(4)
}).then(function() {
  console.log(5)
})

console.log(6)
```

首先被推入调用栈的是全局上下文，你也可以理解为是 script 脚本作为一个宏任务进入了调用栈，这个动作同时创建了全局上下文；与此同时，宏任务队列被清空，微任务队列暂时还是空的：



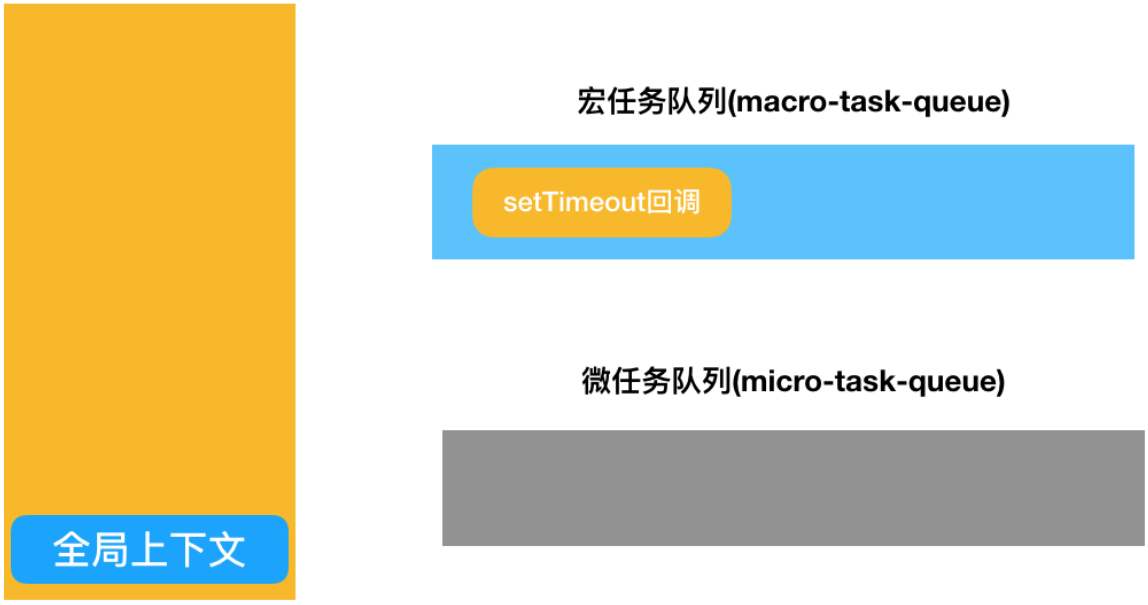
;

全局代码开始执行，跑通了第一个console：

```
console.log(1)
```

此时输出1。

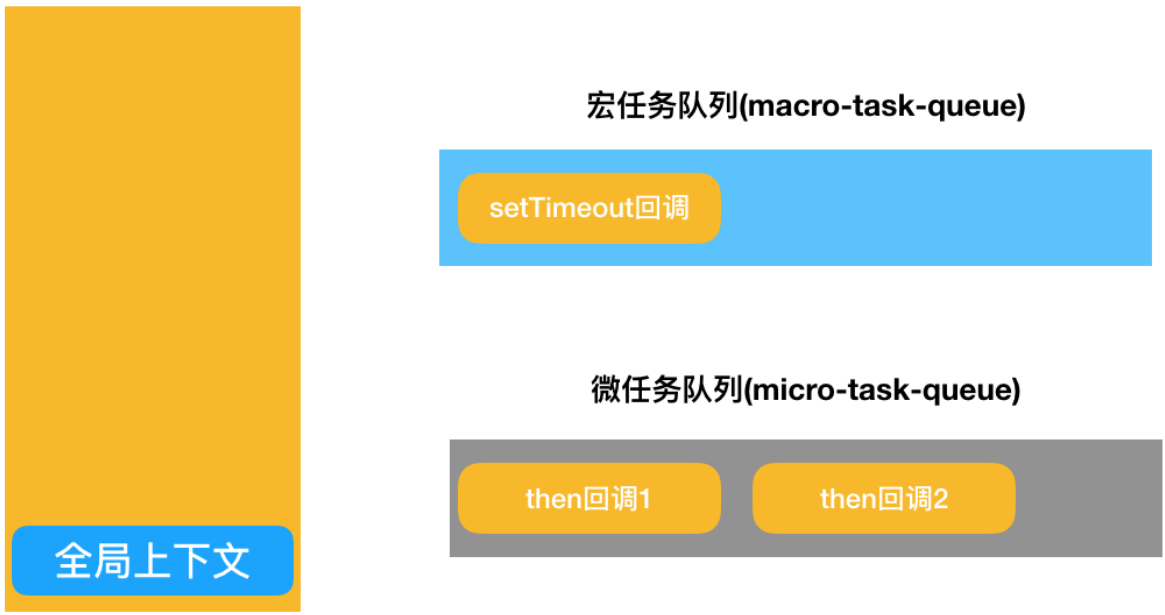
接下来，执行到 setTimeout 这句，一个宏任务被派发了，宏任务队列里多了一个小兄弟：



再往下走，遇到了一个 new Promise。大家知道，Promise 构造函数中函数体的代码都是立即执行的，所以这部分逻辑执行了：

```
console.log(3)
resolve()
```

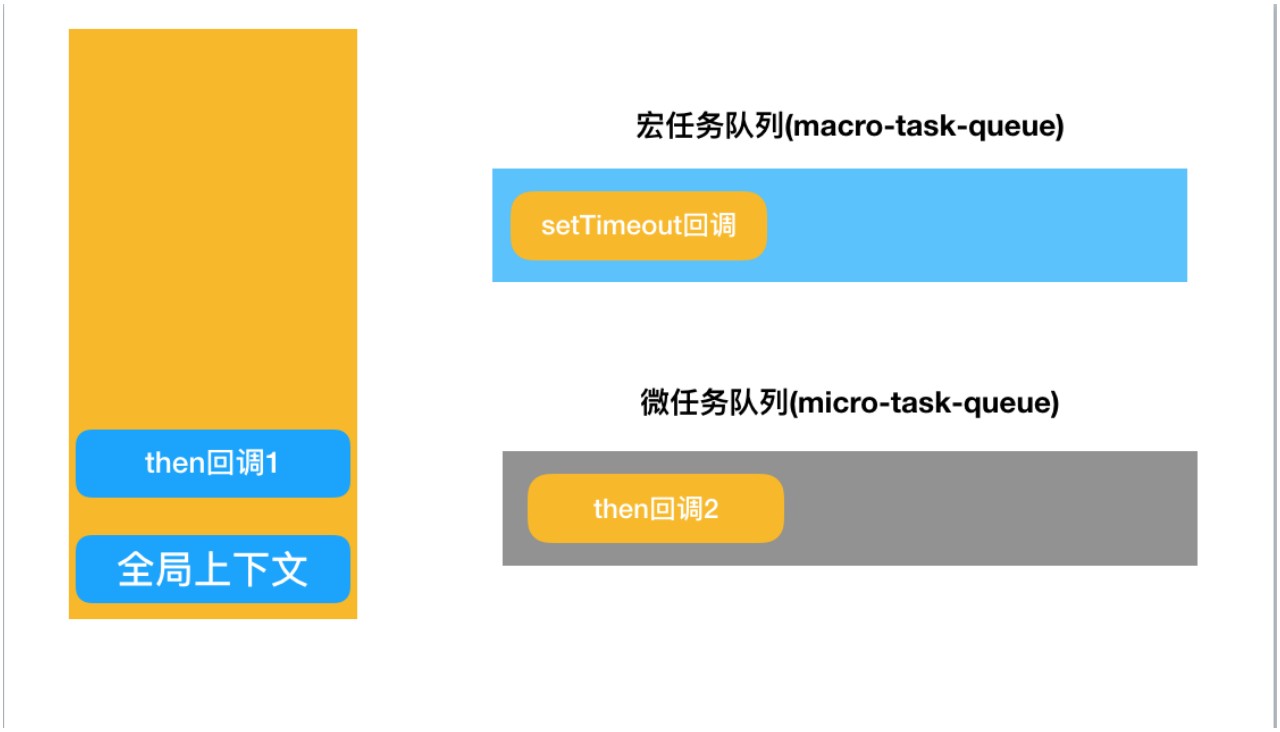
第一步输出了3，第二步敲定了 Promise 的状态为 Fullfilled，成功把 then 方法中对应的两个任务依次推入了微任务队列：



再往下走，就走到了全局代码的最后一句：

```
console.log(6)
```

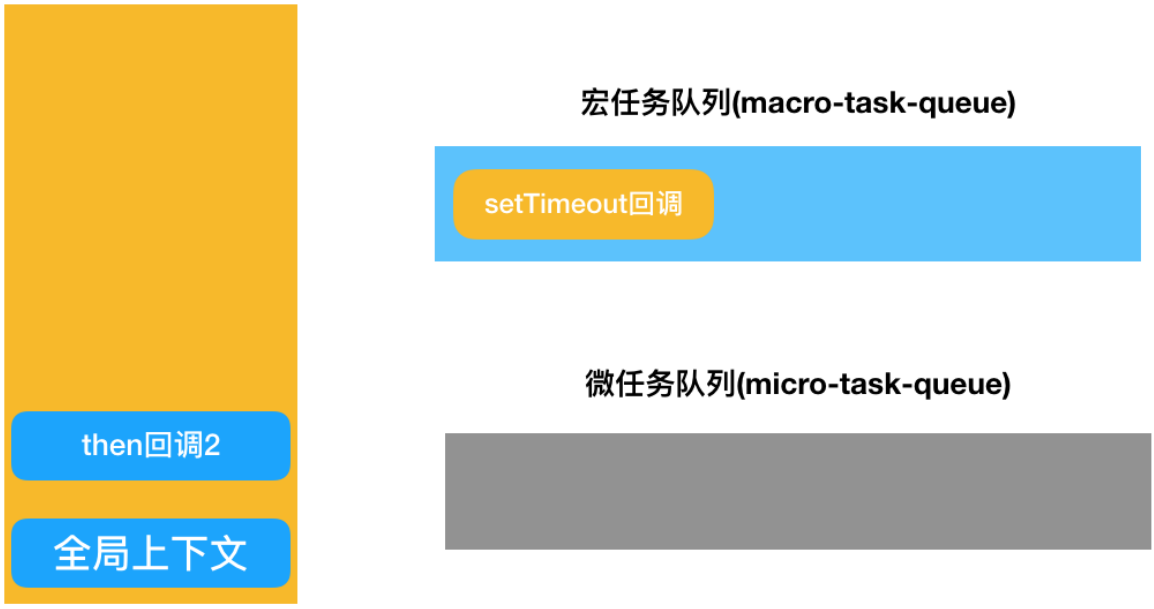
这一步输出了6，script脚本中的同步代码就执行完了。不过大家注意，全局上下文并不会因此消失——它与页面本身共存亡。接下来，咱们就开始往调用栈里推异步任务了。本着“一个 macro，一队micro”的原则，咱们现在需要处理的是微任务队列里的所有任务：



首先登场的是 then 中注册的第一个回调，这个回调会输出4：

```
function () {
  console.log(4)
}
```

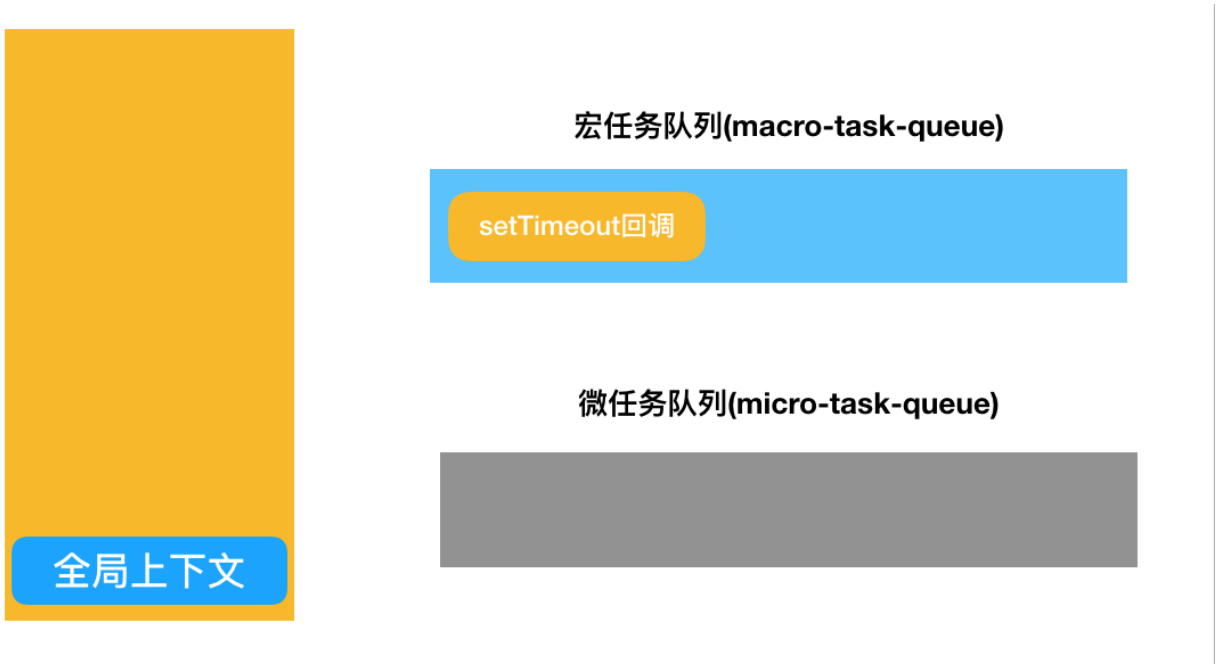
接着处理第二个回调：



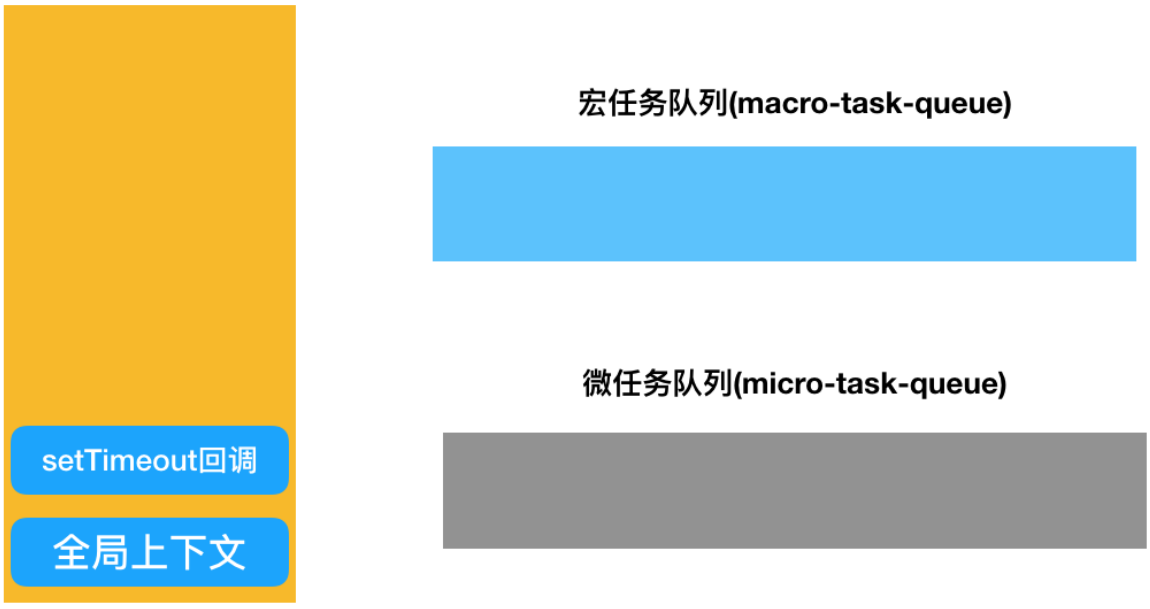
这个回调会输出5：

```
function () {
  console.log(5)
}
```

如此一来，微任务队列就被清空了：



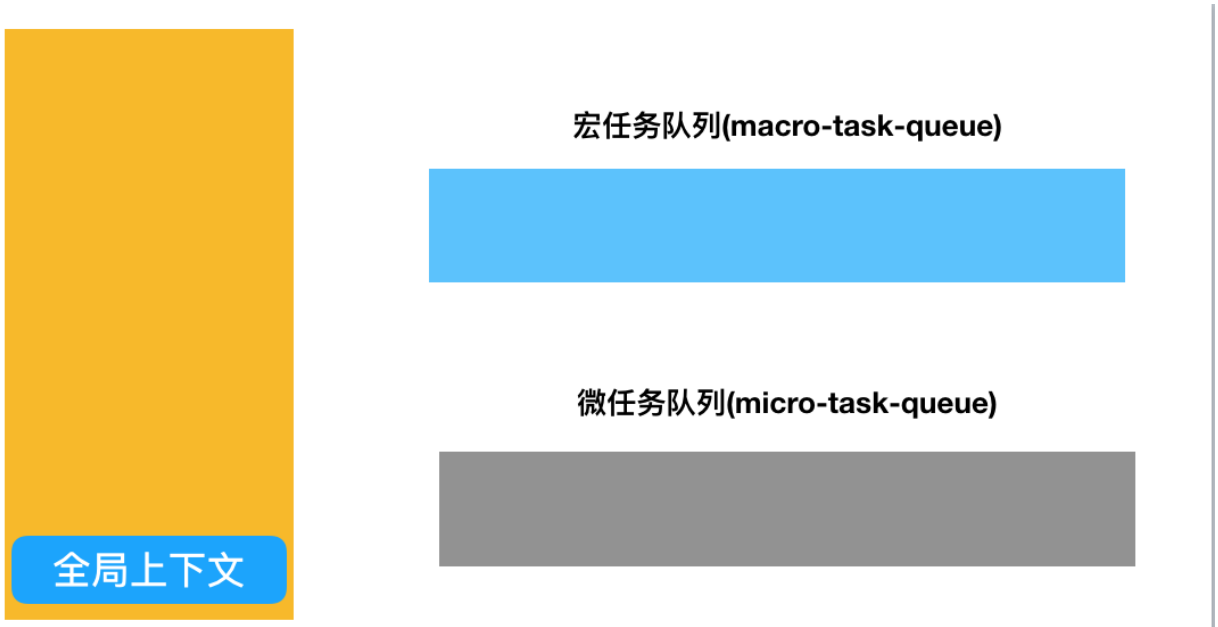
我们重新把目光放在宏任务队列上，将其队列头部的一个任务入栈：



对应的回调执行，输出2：

```
function() {
  console.log(2)
}
```

执行完毕后，我们就结束了所有任务的处理，两个任务队列都空掉了：



此时，只剩下一个全局上下文，待你关闭标签页后，它也会跟着被销毁。

}

26 命题难点深入挖掘—重绘与重排

28：灵魂拷问：Node中的Event-Loop与浏览器有何不同？