setTimeout函数实现原理

> 链接：https://www.jianshu.com/p/2eb7ec01a5a9

> - JavaScript是单线程执行的，无法同时执行多段代码。
> - 当某一段代码正在执行的时候，所有后续的任务都必须等待，形成一个队列。一旦当前任务执行完毕，再从队列中取出下一个任务，这也常被称为 “阻塞式执行”。所以一次鼠标点击，或是计时器到达时间点，或是Ajax请求完成触发了回调函数，这些事件处理程序或回调函数都不会立即运行，而是立即排队，一旦线程有空闲就执行。
> - 假如当前 JavaScript线程正在执行一段很耗时的代码，此时发生了一次鼠标点击，那么事件处理程序就被阻塞，用户也无法立即看到反馈，事件处理程序会被放入任务队列，直到前面的代码结束以后才会开始执行。
> - 如果代码中设定了一个 setTimeout，那么浏览器便会在合适的时间，将代码插入任务队列，如果这个时间设为 0，就代表立即插入队列，但不是立即执行，仍然要等待前面代码执行完毕。所以 setTimeout 并不能保证执行的时间，是否及时执行取决于 JavaScript 线程是拥挤还是空闲。

也就是说**setTimeout只能保证在指定的时间过后将任务(需要执行的函数)插入队列等候，并不保证这个任务在什么时候执行。**执行javascript的线程会在空闲的时候，自行从队列中取出任务然后执行它。javascript通过这种队列机制，给我们制造一个异步执行的假象。

实例分析

```
console.log(1);
setTimeout('console.log(2)',0);
console.log(3);
```

- 运行结果：1、3、2。
- 解析：setTimeout函数是将`console.log(2)`字符串立即插入任务队列尾部，但不会立即执行。**只有在JS线程中没有任何同步代码要执行的前提下才会执行异步代码。**

```
 var t = true;

  window.setTimeout(function (){
     t = false;
  },1000);

  while (t){}

  alert('end');
```

- 运行结果：程序陷入死循环 t = false;得不到执行 alert('end');不会执行.
- 解析：
  - JS是单线程的，所以会先执行`while(t){}`再alert，但这个循环体是死循环，所以永远不会执行`alert`。

- 为什么不执行 `setTimeout`？是因为JS的工作机制是：当线程中没有执行任何同步代码的前提下才会执行异步代码，`setTimeout`是异步代码，所以`setTimeout`只能等JS空闲才会执行，但死循环是永远不会空闲的，所以`setTimeout`也永远不会得到执行。

```
var start = new Date();

setTimeout(function(){
    var end = new Date();
    console.log("Time elapsed: ", end - start, "ms");
}, 500);

while (new Date - start <= 1000){}
```

- 运行结果：`Time elapsed: 1035 ms` (这里的`1035`不准确 但是一定是大于`1000`的)
- 解析：
  - JS是单线程 `setTimeout`异步代码，其回调函数执行必须需等待主线程运行完毕。
  - 当while循环因为时间差超过`1000ms`跳出循环后，`setTimeout`函数中的回调才得以执行。

```
for(var i=0;i<10;i++){
  setTimeout(function() {
      console.log(i);
  }, 0);
}
```

- 运行结果：输出10个10
- 解析：JS单线程 `setTimeout`异步代码 任务队列
- 问：如何修改可以使上述代码输出

```
0123456789
```

  - 自执行函数 或 使用ES6中的`let`关键字

```
//自执行函数 形成闭包 记忆其被创建时的环境
for(var i=0;i<10;i++){
        setTimeout((function() {
            console.log(i);
        })(), 0);
    }
```

setTimeout(0)函数的作用

现在我们了解了`setTimeout`函数执行的原理，那么它有什么作用呢?

- `setTimeout`函数增加了Javascript函数调用的灵活性，为函数执行顺序的调度提供极大便利。
- 简言之，改变顺序，这正是setTimeout(0)的作用。

使用场景示例：

```
<input type="text" onkeydown="show(this.value)">
<div></div>
<script type="text/javascript">
function show(val) {
  document.getElementsByTagName('div')[0].innerHTML = val;
}
</script>
```

这里绑定了`keydown`事件，意图是当用户在文本框里输入字符时，将输入的内容实时地在`<div>`中显示出来。但是实际效果并非如此，可以发现，每按下一个字符时，`<div>`中只能显示出之前的内容，无法得到当前的字符。修改代码：

```
<input type="text" onkeydown="var self=this; setTimeout(function() {show(self.value)},
0)">
<div></div>
<script type="text/javascript">
function show(val) {
  document.getElementsByTagName('div')[0].innerHTML = val;
}
</script>
```

这段代码使用了`setTimeout(0)`就可以实现需要的效果了。 这里其实涉及2个任务，1个是将键盘输入的字符回写到输入框中，一个是获取文本框的值将其写入`div`中。第一个是浏览器自身的默认行为，一个是我们自己编写的代码。很显然，必须要先让浏览器将字符回写到文本框，然后我们才能获取其内容写到`div`中。改变顺序，这正是`setTimeout(0)`的作用。