**关于JS定时器（setTimeout / setInterval）定时不准问题**

**一、问题说明**

JavaScript中定时器主要有setTimeout和setInterval，但是它们在执行时往往和我们设置的延迟时间有出入。

var id1 = setTimeout(fn, delay); //启动一个单定时器，在延迟后调用指定的函数。该函数返回一个惟一的ID，在以后的时间可以通过该ID取消计时器。

var id2 = setInterval(fn, delay); //类似于setTimeout，但不断调用函数(每次都有延迟)，直到它被取消。

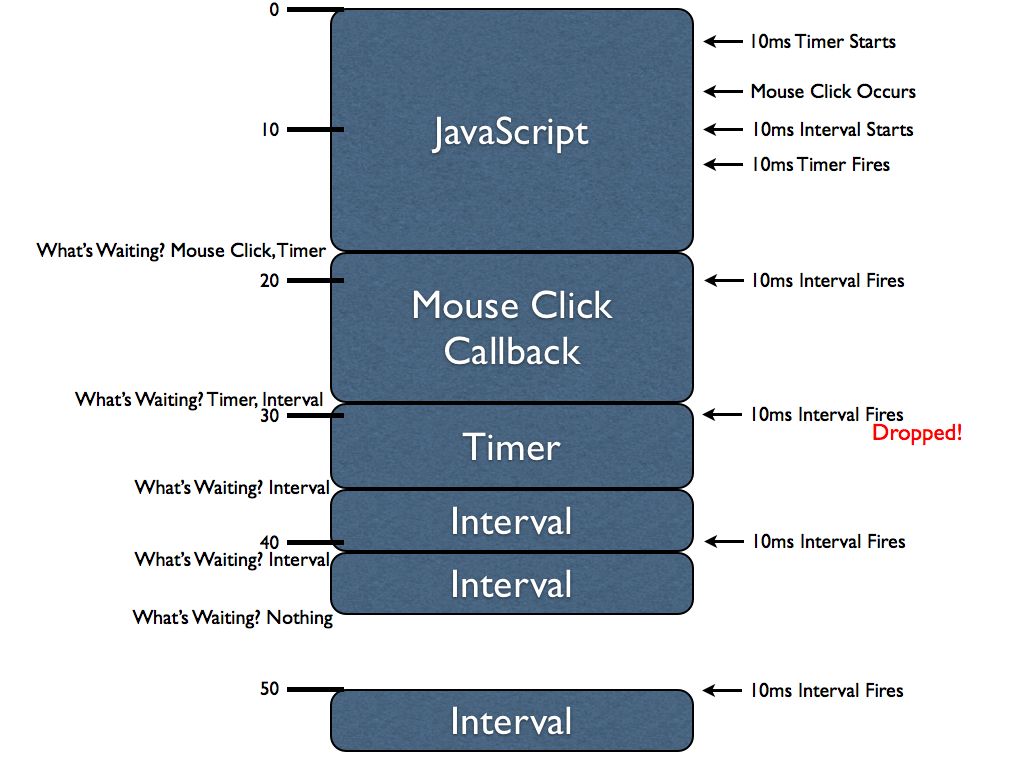
clearInterval (id2), clearTimeout (id1); //接受一个计时器ID(由上述函数返回)并停止计时器回调的发生。

**二、原因分析**

* 浏览器中的所有JavaScript都在单线程上执行，所以异步事件（比如鼠标点击和定时器）仅在线程空闲时才会被调度运行。
* 为了控制要执行的代码， JavaScript 配置了一个任务队列，这些异步事件任务会按照将它们添加到队列的顺序执行。
* 而setTimeout() 的第二个参数（延时时间）只是告诉 JavaScript 再过多长时间把当前任务添加到队列中。如果队列是空的，那么添加的代码会立即执行；如果队列不是空的，那么它就要等前面的代码执行完了以后再执行。

因此**定时器延迟是不能保证的**。

下面是从一篇外文文章摘取的一些解释：



Timers.png

​ 图中有很多信息需要消化，但是完全理解它会让您**更好地了解异步JavaScript执行**是如何工作的。这张图是一维的:垂直方向是(挂钟)时间，单位是毫秒。蓝色框表示正在执行的JavaScript部分。例如，第一个JavaScript块执行大约18ms，鼠标点击块执行大约11ms，以此类推。

​ 由于JavaScript一次只能执行一段代码(由于它的单线程特性)，所以每一段代码都会“阻塞”其他异步事件的进程。这意味着，当异步事件发生时(如鼠标单击、计时器触发或XMLHttpRequest完成)，它将排队等待稍后执行(排队的实际发生方式因浏览器的不同而不同，因此可以认为这是一种简化)。

​ 首先，在JavaScript的第一个块中，启动了两个计时器:一个10ms的setTimeout和一个10ms的setInterval。由于计时器是在哪里和什么时候启动的，它实际上在我们实际完成第一个代码块之前触发。但是请注意，它不会立即执行(由于线程的原因，它无法这样做)。相反，被延迟的函数被排队，以便在下一个可用的时刻执行。

​ 此外，在第一个JavaScript块中，我们看到鼠标单击发生。与此异步事件相关联的JavaScript回调(我们永远不知道用户何时会执行某个动作，因此它被认为是异步的)无法立即执行，因此，就像初始计时器一样，它被排队等待稍后执行。

​ 在JavaScript的初始块完成执行后，浏览器会立即问一个问题:等待执行的是什么?在本例中，鼠标单击处理程序和计时器回调都在等待。然后浏览器选择一个(鼠标点击回调)并立即执行它。计时器将等待到下一个可能的时间，以便执行。

​ 注意，当鼠标单击处理程序执行时，第一个interval回调将执行。与计时器一样，它的处理程序排队等待稍后执行。但是，请注意，当interval再次触发时(当计时器处理程序正在执行时)，此时该处理程序的执行将被删除。如果你想在一个大的代码块执行的时候将所有的interval回调队列起来，那么结果将是一堆在完成时没有延迟的interval执行。相反，浏览器倾向于简单地等待，直到没有更多的间隔处理程序排队(针对所讨论的间隔)。

​ 实际上，我们可以看到，当第三个interval回调被触发时，interval本身正在执行。这向我们展示了一个重要的事实:interval并不关心当前执行的是什么，它们将不加区别地排队，即使这意味着回调之间的时间间隔将被牺牲。

​ 最后，在第二个interval回调执行完成后，我们可以看到JavaScript引擎没有任何东西可以执行了。这意味着浏览器现在等待一个新的异步事件发生。当interval再次触发时，我们会在50ms处得到这个值。但是这一次，没有任何东西阻碍它的执行，因此它立即触发。

**三、解决方案**

1. **动态计算时差 （*仅针对循环定时，只起修正作用* ）**
   * 在定时器开始前和运行时动态获取当前时间，在设置下一次定时时长时，在期望值基础上减去当前时延，以获得相对精准的定时运行效果。
   * 此方法仅能消除setInterval()长时间运行造成的误差累计，但无法消除单个定时器执行延迟问题。

var count = count2 = 0;

var runTime,runTime2;

var startTime,startTime2 = performance.now();//获取当前时间

//普通任务-对比

setInterval(function(){

runTime2 = performance.now();

++count2;

console.log("普通任务",count2 + ' --- 延时：' + (runTime2 - (startTime2 + count2 \* 1000)) + ' 毫秒');

}, 1000);

//动态计算时长

function func(){

runTime = performance.now();

++count;

let time = (runTime - (startTime + count \* 1000));

console.log("优化任务",count2 + ' --- 延时：' + time +' 毫秒');

//动态修正定时时间

t = setTimeout(func,1000 - time);

}

startTime = performance.now();

var t = setTimeout(func , 1000);

//耗时任务

setInterval(function(){

let i = 0;

while(++i < 100000000);

}, 0);

效果：

| **https://upload-images.jianshu.io/upload_images/23791825-e4469561e820ff98.png?imageMogr2/auto-orient/strip|imageView2/2/w/375**  **图1** | **https://upload-images.jianshu.io/upload_images/23791825-57590cb5fff2caaf.png?imageMogr2/auto-orient/strip|imageView2/2/w/375**  **图2** |
| --- | --- |

上图中由于我中途切换了浏览器窗口，导致setInterval任务执行时间往后推移了很多，而修正后版本能够将定时器在拉回原轨道。

**额外说明**：  
​ 在查阅网上资料时，有很多文章说：setInterval一直执行会出现误差累计的问题，但是我在用谷歌浏览器测试的时候并没有发现这问题。  
上述代码中普通任务在正常（保持前台）运行时，延时基本保持在100ms上下波动。



图3

1. **使用 Web Worker**

Web Worker 的作用，就是**为 JavaScript 创造多线程环境**，允许主线程创建 Worker 线程，将一些任务分配给后者运行。在主线程运行的同时，Worker 线程在后台运行，两者互不干扰。等到 Worker 线程完成计算任务，再把结果返回给主线程。这样的好处是，一些计算密集型或高延迟的任务，被 Worker 线程负担了，主线程（通常负责 UI 交互）就会很流畅，不会被阻塞或拖慢。

测试代码如下：

<!-- index.html -->

<html>

<meta charset="utf-8">

<body>

<script type="text/javascript">

var count = 0;

var runTime;

//performance.now()相对Date.now()精度更高，并且不会受系统程序堵塞的影响。

//API：https://developer.mozilla.org/zh-CN/docs/Web/API/Performance/now

var startTime = performance.now(); //获取当前时间

//普通任务-对比测试

setInterval(function(){

runTime = performance.now();

++count;

console.log("普通任务",count + ' --- 普通任务延时：' + (runTime - (startTime + 1000))+' 毫秒');

startTime = performance.now();

}, 1000);

//耗时任务

setInterval(function(){

let i = 0;

while(i++ < 100000000);

}, 0);

// worker 解决方案

let worker = new Worker('worker.js');

</script>

</body>

</html>

// worker.js

var count = 0;

var runTime;

var startTime = performance.now();

setInterval(function(){

runTime = performance.now();

++count;

console.log("worker任务",count + ' --- 延时：' + (runTime - (startTime + 1000))+' 毫秒');

startTime = performance.now();

}, 1000);

效果：

| **https://upload-images.jianshu.io/upload_images/23791825-ce71dd1e36cd02dc.png?imageMogr2/auto-orient/strip|imageView2/2/w/453**  **图4** | **https://upload-images.jianshu.io/upload_images/23791825-bf7451eed090ff6e.png?imageMogr2/auto-orient/strip|imageView2/2/w/453**  **图5** |
| --- | --- |

可以看到使用worker后，时延能够控制在3ms以内，效果很好。而且worker任务不会受到浏览器后台运行的影响。

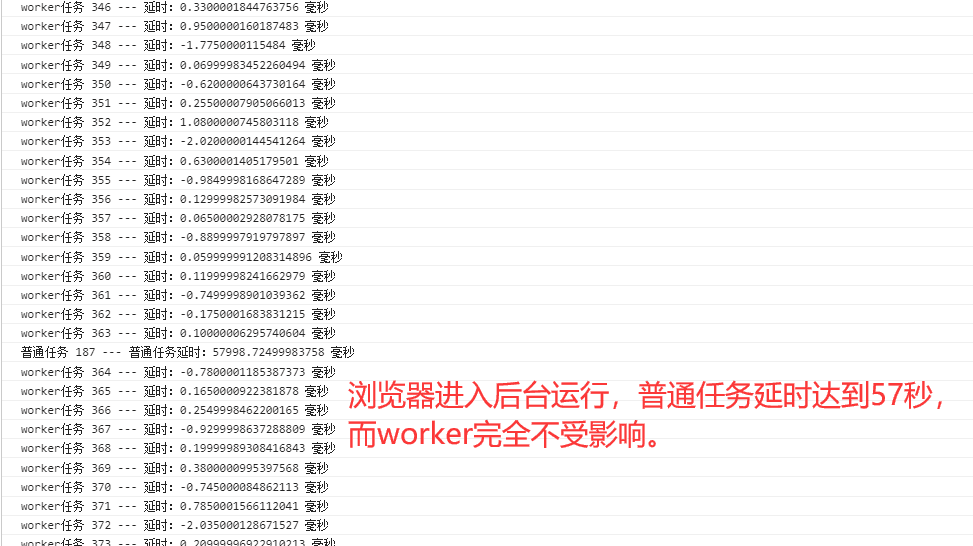


图6

***但是Web Worker 有以下几个使用注意点：***

（1）**同源限制**

分配给 Worker 线程运行的脚本文件，必须与主线程的脚本文件同源。

（2）**DOM 限制**

Worker 线程所在的全局对象，与主线程不一样，无法读取主线程所在网页的 DOM 对象，也无法使用document、window、parent这些对象。但是，Worker 线程可以navigator对象和location对象。

（3）**通信联系**

Worker 线程和主线程不在同一个上下文环境，它们不能直接通信，必须通过消息完成。

（4）**脚本限制**

Worker 线程不能执行alert()方法和confirm()方法，但可以使用 XMLHttpRequest 对象发出 AJAX 请求。

（5）**文件限制**

Worker 线程无法读取本地文件，即不能打开本机的文件系统（file://），它所加载的脚本，必须来自网络。

**总结**： 目前没发现能完全消除定时误差的方法，相对来说Web Worker是个很不错的解决方案。

参考文章：  
[http://ejohn.org/blog/how-javascript-timers-work/](https://links.jianshu.com/go?to=http%3A%2F%2Fejohn.org%2Fblog%2Fhow-javascript-timers-work%2F)  
[https://blog.csdn.net/qq\_41494464/article/details/99944633](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fqq_41494464%2Farticle%2Fdetails%2F99944633)  
[http://www.ruanyifeng.com/blog/2018/07/web-worker.html](https://links.jianshu.com/go?to=http%3A%2F%2Fwww.ruanyifeng.com%2Fblog%2F2018%2F07%2Fweb-worker.html)  
[https://www.cnblogs.com/7qin/p/10225220.html](https://links.jianshu.com/go?to=https%3A%2F%2Fwww.cnblogs.com%2F7qin%2Fp%2F10225220.html)