说明：这篇文章节选自 John Resig 的《Secrets of the JavaScript Ninja》一书，翻译只是供大家学习交流，翻译不足之处，请斧正。

这篇文章主要从下面几个方面解读计时器：

1. 计时器概述；
2. 计时器速度深度探析；
3. 用计时器处理大量任务；
4. 利用计时器管理动画；
5. 较好的计时器测试；

计时器是一个我们了解很少且经常被滥用的东西，它是 javascript 的特色。实际上，在复杂的应用程序开发中，它能为我们提供很多帮助。计时器提供了一个可以将代码片段异步延时执行的能力，javascript 生来是单线程的（在一定时间范围内仅一部分 js 代码能运行），计时器为我们提供了一种避开这种限制的方法，从而开辟了另一条执行代码的蹊径。

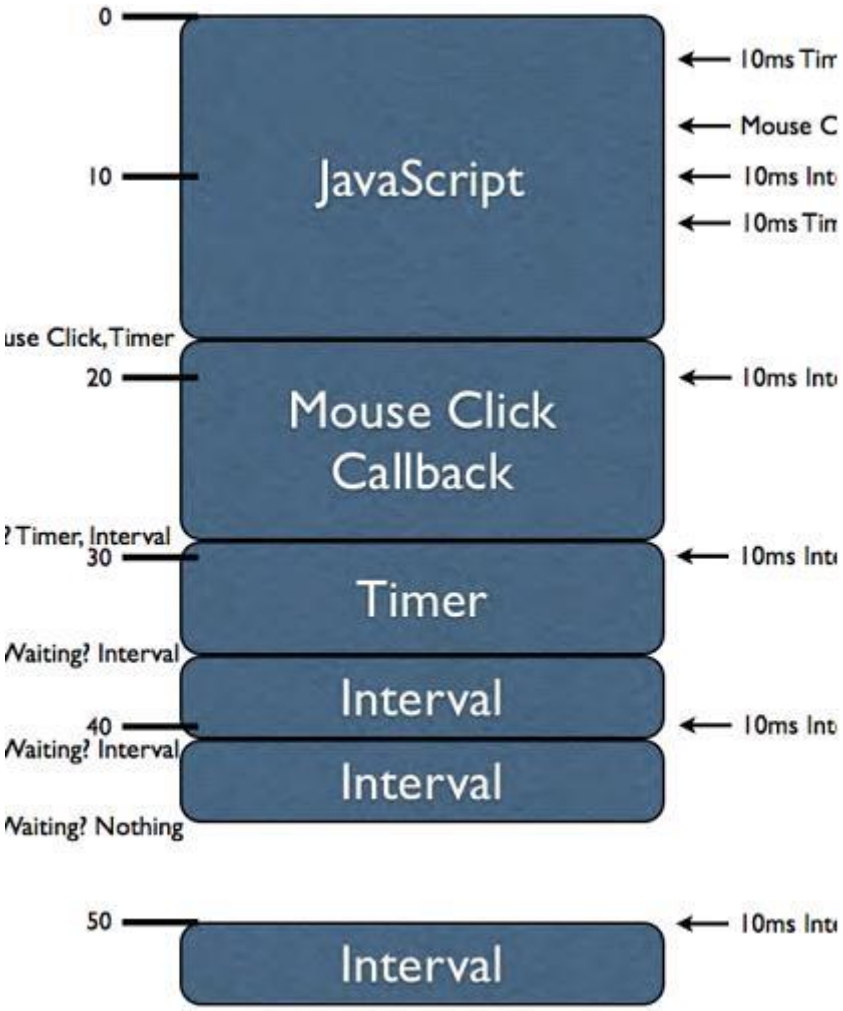
有趣的是，与我们普遍接受的观点相反，计时器并不是 javascript 语言的一部分，而是浏览器引入的方法和对象的一部分。这意味着如果你选择在一个非浏览器的环境运行它，很有可能计时器不存在，你必须使用特定功能推行你自己的版本（如 Rhino 线程）。

1、计时器是如何工作的

从根本上来说，理解计时器如何工作很重要。通常情况下，计时器的行为并不直观，因为它在一个单独的线程中，让我们从三个函数的测试开始，对于每一个函数我们都有机会构建和控制计时器。

- var id = setTimeout(fn,delay)；启动一个计时器，它将在延迟时间之后调用特定的函数，该函数返回一个唯一的 ID，利用这个 ID 计时器在稍后的时间里被取消；
- var id = setInterval(fn,delay)；与 setTimeout 相似，但它不断的调用函数（每隔一定延迟时间）直到它被取消；
- clearInterval(id)，clearTimeout(id)；接受计时器的 ID（由上述任意一个函数返回）并停止调用计时器。

为了理解计时器内部是如何工作的，有一个很重要的概念需要加以探讨：延迟是无法保证的。既然浏览器中所有 javascript 是在一个单线程中运行的，那么异步事件（如鼠标点击、计时器）在执行中也只有存在开放状态时才运行，下面这张图很好的说明了这个问题：



这张图有很多信息需要消化，充分理解它将使你对异步 js 执行有一个更好的认识，图表是一维的，在垂直方向上是时间（挂钟），以毫秒为单位。蓝色盒子代表代表 js 执行的比例。例如，第一个 javascript 块运行时间大约为 18 秒，鼠标点击大约为 11 秒等等。

既然 javascript 在一定时间内之执行一部分代码（源于单线程的特性），那么这些代码块的每一个就被封锁在其它异步事件执行的进程中。这表明当一个异步事件发生时（如鼠标点击、计时器释放、XMLHttpRequest 请求完成），它将排队等候执行（如何排队在不同浏览器之间是不一样的）。

首先，在第一代码块里，有两个计时器触发：一个是 10ms 的 setTimeout，一个是 10ms 的 setInterval。在第一个代码块真正完成之前，它实际上已经释放了。但是，注意，它不会立即执行（由于单线程的问题，它无能为力），相反，为了能在下一个可行的时间得到执行，那些延时函数被编入队列。

另外，在第一个代码块内，我们看到鼠标点击出现。与这个异步事件（我们永远不知道何时执行动作，这样就可以认为它是不同步的）相关 Javascript 回调函数跟初始的计时器一样不能立即被执行，它排队等候执行。

在 Javascript 最初的代码块执行完毕之后，浏览器会发出疑问：正在等候执行的是什么？在这种情况下，鼠标点击处理器和计时器回调函数同时处于等待之中，然后，浏览器将选择一个(鼠标点击回调函数)并立即执行它，计时器函数将等到下一个可能的时间执行。

注意，当鼠标点击函数处理器执行时，第一个回调函数也在执行，至于计时器，其处理器被编入队列稍后执行。但是，请注意，当 Interval 再次释放时（在计时器处理器执行时），计时器执行的时间将减少。如果你在一大块代码执行期间将所有的 Interval 回调函数编入队列，其结果是一大群 Interval 回调函数会毫无延迟的执行，直到全部完成。而浏览器在队列增大之前只是简单的等到没有 Interval 处理器排队为止（间歇问题）。

事实上，我们看到这样一个情况：Interval 正在执行时，第三个 Interval 函数将释放。这表明一个重要的事实：Interval 对当前正在执行什么漠不关心，它们将不会青红皂白的排队，即使是牺牲回调函数之间的时间也在所不辞。

最后，在第二个 Interval 函数执行完毕之后，我们可以看到没有留下任何 Javascript 引擎执行的东西。也就是说，浏览器在等待一个新的异步事件的出现。Interval 再次释放时，我们在 50ms 处获得它，但这一次，执行起来没有任何障碍，它立即释放。

我们来看一个例子，以便更好的说明 setTimeout 和 setInterval 的差异：

```
1.setTimeout(function(){
2./* Some long block of code... */
3.setTimeout(arguments.callee, 10);
4.}, 10);
5.setInterval(function(){
6./* Some long block of code... */
7.}, 10);
```

乍一看，这两段代码似乎功能相同，但并非如此。setTimeout 代码在前一个回调函数执行万之后，至少有 10ms 的延迟（最终可能多些，但至少不会少于此），而 setInterval 将每隔 10ms 尝试执行一次回调函数而不管最后一个回调函数何时执行。

这里有很多我们需要了解，让我们回顾一下：

- Javascript 只有单线程，异步事件被迫排队等候执行；
- setTimeout 和 setInterval 在如何执行异步代码方面有根本的区别；
- 如果计时器无法立即执行，它将延时到下一个可能的时间执行（这比预想的延迟时间要长一些）；
- 如果有充分的执行时间，Interval 可能会毫无延迟的来回执行。

所有这些无疑是重要的知识，了解 Javascript 引擎如何工作，特别是有大量异步事件出现时，这使得在构建高级应用代码片段时有一个良好的基础。

2、计时器最小延时时间和可靠性

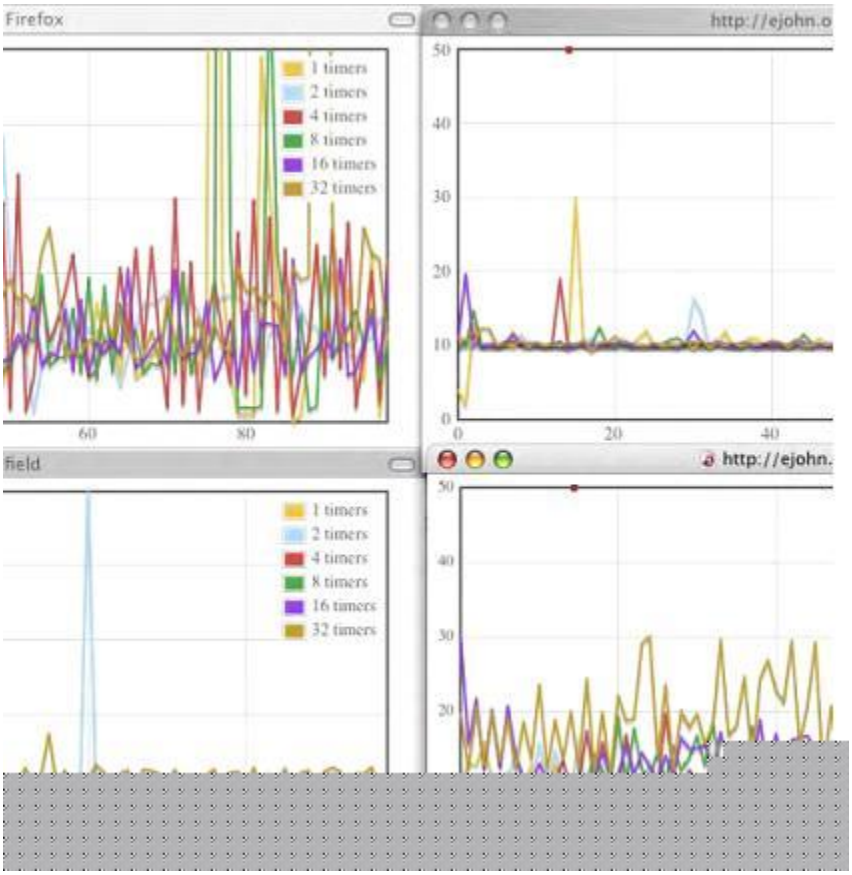
很明显，你可以延迟几秒钟、几分钟、几小时或任何你你想要的时间间隔，但最不明显的是你能选择的最小延时时间。

在一定程度上，浏览器不能为计时器提供良好的解决方案用以精确的处理它们（因为它们自身受操作系统时间的限制）。但是，纵观所有的浏览器，可以很安全的说，最小延时时间大约是 10－15ms。

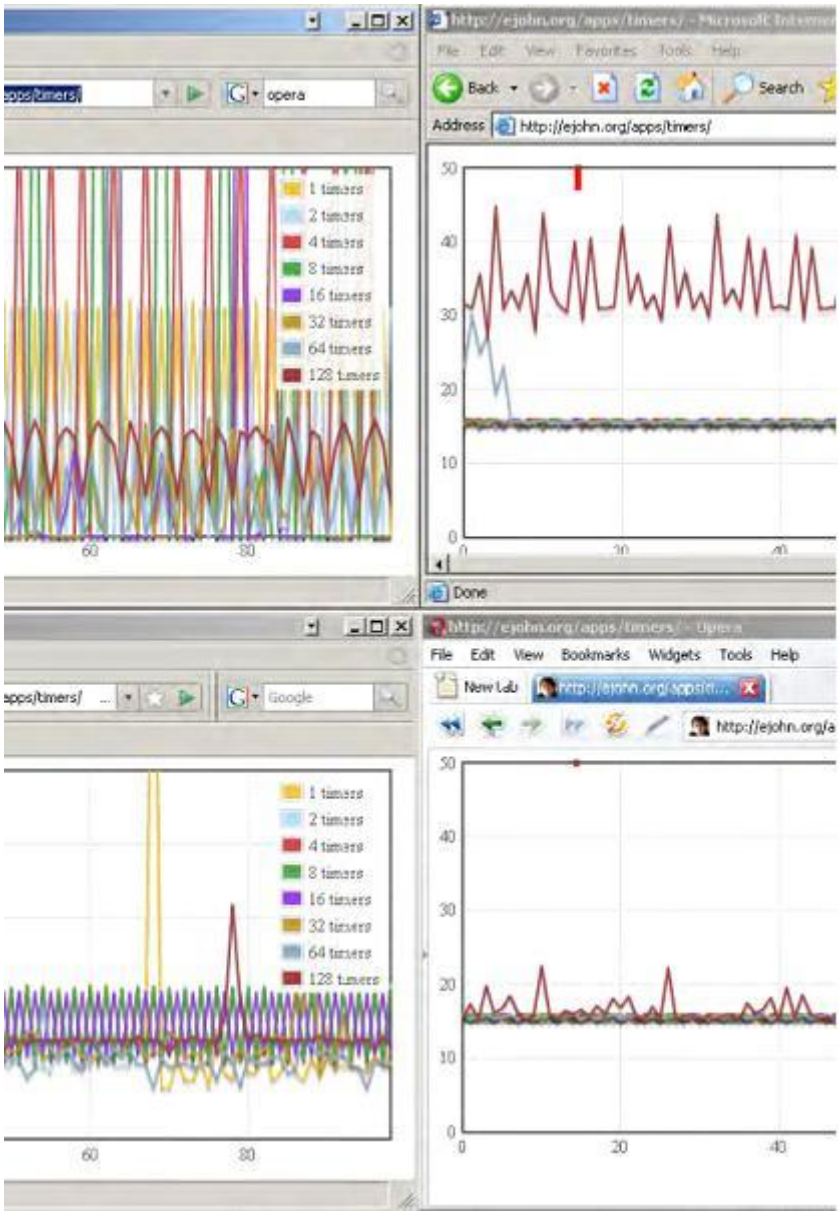
我们可以对跨平台假定的计时器间歇作简单的分析后得出这一结论。例如，如果我们分析延迟时间为 0ms 的 setInterval，我们会发现在大多数浏览器中的最小延迟时间。

在 OS 操作系统下的浏览器中：

从左上角开始，依次为：Firefox 2, Safari 3, Firefox 3, Opera 9



在 Windows 操作系统下得浏览器中：
从左上角开始依次为：Firefox 2, Internet Explorer 6, Firefox 3, Opera 9



上面图表中的线条和数字显示了浏览器同时处理时间间歇的数量，我们可以得出结论：在 OS 上，浏览器的最小延时时间为 10ms，在 windows 上为 15ms。我们可以通过为计时器提供 0（或任何 10ms 以下的任何数值）作为延时时间得到这个值。

但有一个例外，IE 为 setInterval 提供的延时时间不能为 0（即使 setTimeout 能欣然的接受）。当 setInterval 的延时时间为 0 时，它会转变成 setTimeout（仅执行一次回调函数），而我们可以通过为其提供 1ms 的延迟来解决这个问题。由于所有浏览器都能自动向上舍入任何低于最小延时时间的值，所以用 1ms 与有效的使用 0ms 一样安全，或更安全（既然 IE 浏览器现在能工作）。

从这些表中我们可以得到其它信息。最重要的是加强了我们以前所了解到的：浏览器不能保证你所指定的精确的时间间歇。像 Firefox 2，Opera 9（OS）在提供可靠的执行率方面有一定的难度。很多与浏览器如何处理 Javascript 的垃圾回收有关（Firefox 3 在 Javascript 的执行上作了显著的改善，其垃圾回收在这些结果中立竿见影）。

因此，浏览器可以提供非常小的延迟时间，但其精确度得不到保证，那么在使用计时器时，你需要考虑你的应用程序（如果 10ms 和 15ms 有差异，你应该重新思考你应用程序代码的结构）。

说明：本文第一小节—*计时器是如何工作的* 在 musicduwei 的 blog 上也有翻译，读者可点击查看 JavaScript 的单线程性质以及定时器的工作原理
转载地址：http://www.denisdeng.com/?p=396

1、昂贵计算的处理

在复杂 Javascript 应用程序开发中，最复杂的可能是用户界面的单线程特性。而 Javascript 在处理用户交互时最好的状况是反应迟钝，最糟糕的情况是无响应而导致浏览器挂起（在 Javascript 执行时，页面中所有的更新操作暂停）。源于这一事实，将所有复杂操作（任何多于 100ms 的计算）减小到可管理的程度就势在必行。另外，如果运行了至少 5 秒钟还没有停止，一些浏览器（如 Firefox 、Opera）将产生一个提示框警告用户脚本无相应。

这显然是不可取的，产生一个无响应的界面并不好。但是，几乎可以肯定的是，当你需要处理大量数据时就会出现这种情况（如处理数以千计的 DOM 元素会导致这种情况出现）。

此时，计时器就显得尤为有用。由于计时器能有效的暂停 Javascript 代码的执行，它也能阻止浏览器将执行的代码挂起（只要个别代码还不足以使浏览器挂起）。想到这一点，我们可以将正常的、密集的、循环计算纳入到非阻塞的计算之中，让我们看看下面这个例子，这种类型的计算是必需的。

一个长时运行的任务：

```
1.<table><tbody></tbody></table>
01.// Normal, intensive, operation
02.var table = document.getElementsByTagName("tbody");
03.for (  var i = 0; i < 2000; i++ ) {
04.var tr = document.createElement("tr");
05.for (  var t = 0; t < 6; t++ ){
06.var td = document.createElement("td");
07.td.appendChild( document.createTextNode("" + t));
08.tr.appendChild( td );
09.}
10.table.appendChild( tr );
11.}
12.}
```

在这个例子中，我们创建了总数为 26000 个 DOM 节点，并将数字填入表中，这太昂贵了，很有可能将浏览器挂起以阻止用户正常的交互。我们可以将计时器引入其中，从而得到与众不同，也许更好的结果。

用计时器将耗时较长的任务拆分开来

```
1.<table><tbody></tbody></table>
01.var table = document.getElementsByTagName("tbody");
02.var i = 0, max = 1999;
03.setTimeout(function(){
```

```
04.for (  var step = i + 500; i < step; i++ ) {
05.var tr = document.createElement("tr");
06.for (  var t = 0; t < 6; t++ ){
07.var td = document.createElement("td");
08.td.appendChild( document.createTextNode("" + t));
09.tr.appendChild( td );
10.}
11.}
12.table.appendChild( tr );
13.}
14.if ( i < max )
15.setTimeout( arguments.callee, 0 );
16.}, 0);
```

在我们修改的例子中，我们将密集的计算分成四部分，每个创建 6500 个节点。这些计算不太可能中断浏览器正常的流。最糟糕的情况也只是这些数字可能随时调整（例如，使其在 250-500 之间变化，这样我们的每一个单元将产生 3500DOM 个节点）。但是，给人印象最深的是我们如何改变我们的代码以适应新的异步系统。我们需要做更多的工作以确保元素的数字正确生成（该循环不会永无休止）。这些代码与我们初始的很相似。注意，我们使用闭包维持代码片段间的迭代状态，毫无疑问，不使用闭包，此代码将更为复杂。

与原来的技术相比，使用该技术有一个明显的变化。浏览器的长时挂起被 4 个视觉化的页面更新替代。虽然浏览器尝试着尽可能快得去执行这些代码片段，它也在计时器的每一步操作之后渲染 DOM 变化（就好像大量的更新）。大多数情况下，用户觉察不到此种类型的更新，但记住它们曾经发生过很重要。

有一种情况会使该技术能专门为我服务——我构建的计算大学生日程排列的应用程序。起初，该应用程序是典型的 CGI（客户与服务器交流，服务器计算出日程表之后将其返回）。但我对它作了改变，讲所有的日程计算放到客户端，这是日程计算的视图：

这些计算的代价相当昂贵（需要遍历数以千计的排列以找到正确的答案）。将日程计算分割成实实在在的单元使这一问题得到了解决（用已经完成的那部分更新用户界面）。最后，用户提交的是快速、反应灵敏、可用性较高的用户界面。

如此有用的技术常常令人惊奇。你会发现它经常被用于长时运行的程序之中，就像测试箱（我们在这章末尾讨论它）。更为重要的是，这种技术向我们显示了解决浏览器环境的限制是多么的轻而易举，同时也为用户提供了丰富的经验。

2、中央计时器控件

在使用计时器时，出现了一个问题——在处理大量计时器时如何管理它们。在动画处理中你尝试去同步处理大量属性时就尤其重要，你需要一种方法去管理它们。

你的计时器有一个核心的控制将赋予你很大的权力和灵活性，即：

- 在某个时刻每个页面仅一个计时器运行；
- 你可以随时暂停和继续计时器；
- 移除回调函数不过是小菜一碟。

你也必需认识到，增加同步计时器的数量有可能增加浏览器垃圾回收出现的可能性。一般来说，浏览器在搜寻并尝试绑定任何零碎的东西（删除未使用的变量、对象等）。因为它们通常在正常的 Javascript 引擎管理之外（通过其它线程），这时定时器的问题就尤为严重。一些浏览器能处理这种情况而另外一些浏览器会导致长时的垃圾回收循环出现。你也许会注意到这种问题——当你在一个浏览器中看到的是漂亮、平滑的动画，而在另一个浏览器中动画是走走停停完成的。减少计时器同步应用的数量对此大有裨益（这也是现代浏览器引入类似中央计时器控件构件的原因）。

让我们来看一个例子，我们有多个函数，这些函数分别对单个属性产生动画效果，但它们被一个单独的计时器函数管理。

用计时器队列控制多重动画

```
1.<div id="box" style="position:absolute;">Hello!</div>
01.var timers = {
02.timerID: 0,
03.timers: [],
04.start:  function(){
05.if (  this.timerID )
06.return;
07.(function(){
08.for (  var i = 0; i < timers.timers.length; i++ )
09.if (
10.timers.timers[i]() ===  false ) {
11.timers.timers.splice(i, 1); i--; }
12.timers.timerID = setTimeout( arguments.callee, 0 ); })();
13.},
14.stop:  function(){
15.clearTimeout(  this.timerID );
16.this.timerID = 0;
17.},
18.add:  function(fn){
19.this.timers.push( fn );  this.start();
20.}
21.};
22.var box = document.getElementsById("box"), left = 0, top = 20;
23.timers.add(function(){
24.box.style.left = left +  "px";  if ( ++left > 50 )
25.return false;
26.});
27.timers.add(function(){
28.box.style.top = top +  "px";
29.top += 2;
30.if ( top > 120 )
31.return false;
32.});
```

在这我们创建了一个中央控制结构。我们可以添加新的计时器回调函数，并可停止和开始它们的执行。此外，回调函数有能力在任何时候删除自己，只需简单的返回“false”即可（这比通常的 clearTimeout 模式更容易），让我们逐一分析代码看看它是如何工作的。

起初，所有回调函数连同当前计时器的 ID（timers.ID）被存储在一个中央数组中（timers.timers）。核心内容在 start()函数内部，在这里我们需要确认得是已没有计时器在运行，如果是那样，就开始我们的中央计时器。

计时器包含一个循环，它遍历所有的回调函数，并按顺序执行它们，它还检查回调函数返回的值，如果是“false”，将从执行中移除。事实证明，这是计时器管理非常简单的方式。

有一点非常重要，用这种方式组织计时器会确保回调函数总是按顺序执行，那样，正常的计时器总是得不到保证（浏览器一个接一个的选择执行）。

定时器的这种组织方式对于大型应用程序或任何形式的真正的 Javascript 动画至关重要，当我们讨论如何创建跨浏览器动画时，有一种解决方案肯定有助于将来任何形式的应用开发。

3、异步测试

另外一种中央计时器控件能派上用场的情况是在你打算做异步测试的时候。这里的问题是当我们需要对没有立即完成的计算执行测试时（如计时器内的一些行为或 XMLHttpRequest）。我们需要将测试包分解，这样就会完全异步工作。

例如，在一个正常的测试包中，我们可以很容易的运行这些测试。但是，一旦在我们的测试中引入这种需求，我们需要分解它们并单独处理。我们可以用下面的代码达到我们期望的效果。

简单的异步测试包

```
01.(function(){
02.var queue = [], timer;
03.this.test =  function(fn){
04.queue.push( fn );
05.resume();
06.};
07.this.pause =  function(){
08.clearInterval( timer );
09.timer = 0;
10.};
11.this.resume =  function(){
12.if ( !timer )
13.return;
14.timer = setInterval(function(){
15.if ( queue.length )
16.queue.shift()();
17.else
18.pause();
19.}, 1);
20.};
21.})();
22.test(function(){
23.pause();
24.setTimeout(function(){
25.assert(  true,  "First test completed");
26.resume();
27.}, 100);
```

```
28.});
29.test(function(){
30.pause();
31.setTimeout(function(){
32.assert(  true,  "Second test completed");
33.resume();
34.}, 200);
35.});
```

最重要的一个方面是，传递给 test()的每一个函数至多包含一个异步测试。其异步性是通过使用 pause()和 resume()函数来定义的，这些函数在异步事件前后调用。实际上，上面的代码只不过是保持异步行为的一种手段——其包含的函数以特定的顺序执行（它不完全用于本测试包，但在这非常有用）。

让我们看看使这种行为成为可能的代码。函数的大部分功能包含在 resume()函数中，它的行为与我们前面例子中的 start()方法相似，但它主要用来处理队列数据，其主要目的是取出一个函数并执行它。如果有一个在等待，它将完全停止运行。最重要的方面是，既然队列处理代码完全是异步的，他就能保证在我们调用 pause()函数之后尝试执行。

这一段简短的代码使我们的测试以完全异步的方式执行，但仍维持着 test()函数执行的顺序（如果结果具有毁坏性且影响其它测试，这就非常关键）。谢天谢地，我们可以看到，使用最有效的计时器，给一个存在的测试包增加可靠的异步测试并不需要所有的开销。

4、总结

了解 Javascript 函数如何工作启发了我们：这些看似简单的特征在其执行时相当复杂。但是，考虑到它的复杂性，我们会更加深入的研究它们。很明显，计时器最终在复杂的应用程序中特别有用（计算昂贵的代码、动画、异步测试包）。但由于其易用性（特别是增加了闭包），它们往往易于掌握，即使是在最复杂的情况下。

附文章中提到的 assert()函数：（该函数是基于 jQuery 创建的）

```
1.function assert(pass, msg){
2.var type = pass ?  "PASS":  "FAIL";
3.jQuery("#results").append("
4.<li  class=""+ type +  ""><strong>" + type + "</strong>  " + msg +
"</li>
5.");
6.}
```

转载地址：http://www.denisdeng.com/?p=409