定时器

JavaScript 提供定时执行代码的功能，叫做定时器（timer），主要由`setTimeout()`和`setInterval()`这两个函数来完成。它们向任务队列添加定时任务。

setTimeout()

`setTimeout`函数用来指定某个函数或某段代码，在多少毫秒之后执行。它返回一个整数，表示定时器的编号，以后可以用来取消这个定时器。

```javascript
var timerId = setTimeout(func|code, delay);
```

上面代码中，`setTimeout`函数接受两个参数，第一个参数`func|code`是将要推迟执行的函数名或者一段代码，第二个参数`delay`是推迟执行的毫秒数。

```javascript
console.log(1);
setTimeout('console.log(2)',1000);
console.log(3);
// 1
// 3
// 2
```

上面代码会先输出1和3，然后等待1000毫秒再输出2。注意，`console.log(2)`必须以字符串的形式，作为`setTimeout`的参数。

如果推迟执行的是函数，就直接将函数名，作为`setTimeout`的参数。

```javascript
function f() {
  console.log(2);
}

setTimeout(f, 1000);
```

`setTimeout`的第二个参数如果省略，则默认为0。

```javascript
setTimeout(f)
// 等同于
setTimeout(f, 0)
```

除了前两个参数，`setTimeout`还允许更多的参数。它们将依次传入推迟执行的函数（回调函数）。

```javascript
setTimeout(function (a,b) {
  console.log(a + b);
}, 1000, 1, 1);
```

上面代码中，`setTimeout`共有4个参数。最后那两个参数，将在1000毫秒之后回调函数执行时，作为回调函数的参数。

还有一个需要注意的地方，如果回调函数是对象的方法，那么`setTimeout`使得方法内部的`this`关键字指向全局环境，而不是定义时所在的那个对象。

```javascript
var x = 1;

var obj = {
  x: 2,
  y: function () {
    console.log(this.x);
  }
};

setTimeout(obj.y, 1000) // 1
```

上面代码输出的是1，而不是2。因为当`obj.y`在1000毫秒后运行时，`this`所指向的已经不是`obj`了，而是全局环境。

为了防止出现这个问题，一种解决方法是将`obj.y`放入一个函数。

```javascript
var x = 1;

var obj = {
  x: 2,
  y: function () {
    console.log(this.x);
  }
};

setTimeout(function () {
  obj.y();
}, 1000);
// 2
```

上面代码中，`obj.y`放在一个匿名函数之中，这使得`obj.y`在`obj`的作用域执行，而不是在全局作用域内执行，所以能够显示正确的值。

另一种解决方法是，使用`bind`方法，将`obj.y`这个方法绑定在`obj`上面。

```javascript
var x = 1;

var obj = {
  x: 2,
  y: function () {
    console.log(this.x);
  }
};

setTimeout(obj.y.bind(obj), 1000)
// 2
```

setInterval()

`setInterval`函数的用法与`setTimeout`完全一致，区别仅仅在于`setInterval`指定某个任务每隔一段时间就执行一次，也就是无限次的定时执行。

```javascript
var i = 1
var timer = setInterval(function() {
  console.log(2);
}, 1000)
```

上面代码中，每隔1000毫秒就输出一个2，会无限运行下去，直到关闭当前窗口。

与`setTimeout`一样，除了前两个参数，`setInterval`方法还可以接受更多的参数，它们会传入回调函数。

下面是一个通过`setInterval`方法实现网页动画的例子。

```javascript
var div = document.getElementById('someDiv');
var opacity = 1;
var fader = setInterval(function() {
  opacity -= 0.1;
  if (opacity >= 0) {
    div.style.opacity = opacity;
  } else {
    clearInterval(fader);
  }
}, 100);
```

上面代码每隔100毫秒，设置一次`div`元素的透明度，直至其完全透明为止。

`setInterval`的一个常见用途是实现轮询。下面是一个轮询 URL 的 Hash 值是否发生变化的例子。

```javascript
var hash = window.location.hash;
var hashWatcher = setInterval(function() {
  if (window.location.hash != hash) {
    updatePage();
  }
}, 1000);
```

`setInterval`指定的是“开始执行”之间的间隔，并不考虑每次任务执行本身所消耗的时间。因此实际上，两次执行之间的间隔会小于指定的时间。比如，`setInterval`指定每 100ms 执行一次，每次执行需要 5ms，那么第一次执行结束后95毫秒，第二次执行就会开始。如果某次执行耗时特别长，比如需要105毫秒，那么它结束后，下一次执行就会立即开始。

为了确保两次执行之间有固定的间隔，可以不用`setInterval`，而是每次执行结束后，使用`setTimeout`指定下一次执行的具体时间。

```javascript
var i = 1;
var timer = setTimeout(function f() {
  // ...
  timer = setTimeout(f, 2000);
}, 2000);
```

上面代码可以确保，下一次执行总是在本次执行结束之后的2000毫秒开始。

clearTimeout()，clearInterval()

`setTimeout`和`setInterval`函数，都返回一个整数值，表示计数器编号。将该整数传入`clearTimeout`和`clearInterval`函数，就可以取消对应的定时器。

```javascript
var id1 = setTimeout(f, 1000);
var id2 = setInterval(f, 1000);

clearTimeout(id1);
clearInterval(id2);
```

上面代码中，回调函数`f`不会再执行了，因为两个定时器都被取消了。

`setTimeout`和`setInterval`返回的整数值是连续的，也就是说，第二个`setTimeout`方法返回的整数值，将比第一个的整数值大1。

```javascript
```

```
function f() {}
setTimeout(f, 1000) // 10
setTimeout(f, 1000) // 11
setTimeout(f, 1000) // 12
```

上面代码中，连续调用三次`setTimeout`，返回值都比上一次大了1。

利用这一点，可以写一个函数，取消当前所有的`setTimeout`定时器。

```javascript
(function() {
  // 每轮事件循环检查一次
  var gid = setInterval(clearAllTimeouts, 0);

  function clearAllTimeouts() {
    var id = setTimeout(function() {}, 0);
    while (id > 0) {
      if (id !== gid) {
        clearTimeout(id);
      }
      id--;
    }
  }
})();
```

上面代码中，先调用`setTimeout`，得到一个计算器编号，然后把编号比它小的计数器全部取消。

实例：debounce 函数

有时，我们不希望回调函数被频繁调用。比如，用户填入网页输入框的内容，希望通过 Ajax 方法传回服务器，jQuery 的写法如下。

```javascript
$('textarea').on('keydown', ajaxAction);
```

这样写有一个很大的缺点，就是如果用户连续击键，就会连续触发`keydown`事件，造成大量的 Ajax 通信。这是不必要的，而且很可能产生性能问题。正确的做法应该是，设置一个门槛值，表示两次 Ajax 通信的最小间隔时间。如果在间隔时间内，发生新的`keydown`事件，则不触发 Ajax 通信，并且重新开始计时。如果过了指定时间，没有发生新的`keydown`事件，再将数据发送出去。

这种做法叫做 debounce（防抖动）。假定两次 Ajax 通信的间隔不得小于2500毫秒，上面的代码可以改写成下面这样。

```javascript
$('textarea').on('keydown', debounce(ajaxAction, 2500));
```

```
function debounce(fn, delay){
  var timer = null; // 声明计时器
  return function() {
    var context = this;
    var args = arguments;
    clearTimeout(timer);
    timer = setTimeout(function () {
      fn.apply(context, args);
    }, delay);
  };
}
```

上面代码中，只要在2500毫秒之内，用户再次击键，就会取消上一次的定时器，然后再新建一个定时器。这样就保证了回调函数之间的调用间隔，至少是2500毫秒。

运行机制

`setTimeout`和`setInterval`的运行机制，是将指定的代码移出本轮事件循环，等到下一轮事件循环，再检查是否到了指定时间。如果到了，就执行对应的代码；如果不到，就继续等待。

这意味着，`setTimeout`和`setInterval`指定的回调函数，必须等到本轮事件循环的所有同步任务都执行完，才会开始执行。由于前面的任务到底需要多少时间执行完，是不确定的，所以没有办法保证，`setTimeout`和`setInterval`指定的任务，一定会按照预定时间执行。

```javascript
setTimeout(someTask, 100);
veryLongTask();
```

上面代码的`setTimeout`，指定100毫秒以后运行一个任务。但是，如果后面的`veryLongTask`函数（同步任务）运行时间非常长，过了100毫秒还无法结束，那么被推迟运行的`someTask`就只有等着，等到`veryLongTask`运行结束，才轮到它执行。

再看一个`setInterval`的例子。

```javascript
setInterval(function () {
  console.log(2);
}, 1000);

sleep(3000);

function sleep(ms) {
  var start = Date.now();
  while ((Date.now() - start) < ms) {
  }
}
```

```

上面代码中，`setInterval`要求每隔1000毫秒，就输出一个2。但是，紧接着的`sleep`语句需要3000毫秒才能完成，那么`setInterval`就必须推迟到3000毫秒之后才开始生效。注意，生效后`setInterval`不会产生累积效应，即不会一下子输出三个2，而是只会输出一个2。

## setTimeout(f, 0)

### 含义

`setTimeout`的作用是将代码推迟到指定时间执行，如果指定时间为`0`，即`setTimeout(f, 0)`，那么会立刻执行吗?

答案是不会。因为上一节说过，必须要等到当前脚本的同步任务，全部处理完以后，才会执行`setTimeout`指定的回调函数`f`。也就是说，`setTimeout(f, 0)`会在下一轮事件循环一开始就执行。

```javascript
setTimeout(function () {
  console.log(1);
}, 0);
console.log(2);
// 2
// 1
```

上面代码先输出`2`，再输出`1`。因为`2`是同步任务，在本轮事件循环执行，而`1`是下一轮事件循环执行。

总之，`setTimeout(f, 0)`这种写法的目的是，尽可能早地执行`f`，但是并不能保证立刻就执行`f`。

实际上，`setTimeout(f, 0)`不会真的在0毫秒之后运行，不同的浏览器有不同的实现。以 Edge 浏览器为例，会等到4毫秒之后运行。如果电脑正在使用电池供电，会等到16毫秒之后运行；如果网页不在当前 Tab 页，会推迟到1000毫秒（1秒）之后运行。这样是为了节省系统资源。

应用

`setTimeout(f, 0)`有几个非常重要的用途。它的一大应用是，可以调整事件的发生顺序。比如，网页开发中，某个事件先发生在子元素，然后冒泡到父元素，即子元素的事件回调函数，会早于父元素的事件回调函数触发。如果，想让父元素的事件回调函数先发生，就要用到`setTimeout(f, 0)`。

```javascript
// HTML 代码如下
// <input type="button" id="myButton" value="click">

var input = document.getElementById('myButton');

```
input.onclick = function A() {
  setTimeout(function B() {
    input.value +=' input';
  }, 0)
};

document.body.onclick = function C() {
  input.value += ' body'
};
```

上面代码在点击按钮后，先触发回调函数`A`，然后触发函数`C`。函数`A`中，`setTimeout`将函数`B`推迟到下一轮事件循环执行，这样就起到了，先触发父元素的回调函数`C`的目的了。

另一个应用是，用户自定义的回调函数，通常在浏览器的默认动作之前触发。比如，用户在输入框输入文本，`keypress`事件会在浏览器接收文本之前触发。因此，下面的回调函数是达不到目的的。

```javascript
// HTML 代码如下
// <input type="text" id="input-box">

document.getElementById('input-box').onkeypress = function (event) {
  this.value = this.value.toUpperCase();
}
```

上面代码想在用户每次输入文本后，立即将字符转为大写。但是实际上，它只能将本次输入前的字符转为大写，因为浏览器此时还没接收到新的文本，所以`this.value`取不到最新输入的那个字符。只有用`setTimeout`改写，上面的代码才能发挥作用。

```javascript
document.getElementById('input-box').onkeypress = function() {
  var self = this;
  setTimeout(function() {
    self.value = self.value.toUpperCase();
  }, 0);
}
```

上面代码将代码放入`setTimeout`之中，就能使得它在浏览器接收到文本之后触发。

由于`setTimeout(f, 0)`实际上意味着，将任务放到浏览器最早可得的空闲时段执行，所以那些计算量大、耗时长的任务，常常会被放到几个小部分，分别放到`setTimeout(f, 0)`里面执行。

```javascript
var div = document.getElementsByTagName('div')[0];
```

```
// 写法一
for (var i = 0xA00000; i < 0xFFFFFF; i++) {
  div.style.backgroundColor = '#' + i.toString(16);
}

// 写法二
var timer;
var i=0x100000;

function func() {
  timer = setTimeout(func, 0);
  div.style.backgroundColor = '#' + i.toString(16);
  if (i++ == 0xFFFFFF) clearTimeout(timer);
}

timer = setTimeout(func, 0);
```

上面代码有两种写法，都是改变一个网页元素的背景色。写法一会造成浏览器“堵塞”，因为 JavaScript 执行速度远高于 DOM，会造成大量 DOM 操作“堆积”，而写法二就不会，这就是`setTimeout(f, 0)`的好处。

另一个使用这种技巧的例子是代码高亮的处理。如果代码块很大，一次性处理，可能会对性能造成很大的压力，那么将其分成一个个小块，一次处理一块，比如写成`setTimeout(highlightNext, 50)`的样子，性能压力就会减轻。