异步操作-定时器

1 setTimeout

```
var timerId = setTimeout(callback|code, delay);
clearTimeout(timerId);
```

`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 `clearTimeout()` 来取消这个定时器。

例如，下面的代码会先输出 1 和 3，然后在 1000ms 后输出 2。

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

setTimeout 的第二个参数如果省略，则默认为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）。

使用时可能出现的问题：

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。

例如，下面的代码会输出 5， 而不是 2。

```
var x = 5;

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

setTimeout(obj.y, 1000);
```

解决办法一：

将 `obj.y` 放入一个函数。

```
var x = 1;

var obj = {
  x: 2,
  y: function () {
```

```
    console.log(this.x);
  }
};

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

解决办法二：

使用 bind 方法，将 `obj.y` 这个方法绑定在 obj 上面。

```
var x = 1;

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

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

setTimeout 应用：防抖动

假定两次 Ajax 通信的间隔不得小于2500毫秒，代码可以写成下面这样：

```
$('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);
  };
}
```

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。

2 setInterval

```
var timerId = setInterval(callback|code, time);
clearInterval(timerId);
```

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。

常见用途：轮询

例如，下面是一个轮询 URL 的 Hash 值是否发生变化的例子。

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

不能严格保证时间间隔

setInterval 并不能严格保证两次任务之间的时间间隔。

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。

解决办法：使用 setTimeout

如下面的代码可以确保，下一次执行总是在本次执行结束之后的2000毫秒开始。

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

3 定时器的运行机制

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。

例如

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

上面代码的setTimeout，将回调函数 someTask 移出本轮事件循环，并等待 100毫秒以后执行；如果本轮事件循环中的veryLongTask函数（同步任务）运行时间超过100毫秒还没有结束，那回调函数 someTask 也只能等着veryLongTask函数结束。

再看一个setInterval的例子

```

```

```
console.time('ss');
setInterval(function () {
  console.log(2);
}, 1000);

sleep(3000);
console.timeEnd('ss');
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。但是在第一次输出2时，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。

4 setTimeout(f,0)

setTimeout(f, 0)不会立即执行，而是会在下一轮事件循环一开始就执行。

例如：

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

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。

应用：

1. 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)。

```
// 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。

再比如，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。

想要在用户输入文本时，自动转换为大写，使用下面的代码是达不到目的的，因为keypress事件会在浏览器接收文本之前触发。

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

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

上面的代码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。

解决办法：

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