第六章：使用计时器制定函数的执行计划

本章内容：

- 函数的延迟执行
- 取消执行计划
- 制定函数的周期性执行计划
- 将函数执行延迟到事件循环的下一轮

如果你熟悉客户端 JavaScript 编程，你可能使用过 setTimeout 和 setInterval 函数，这两个函数允许延时一段时间再运行函数。比如下面的代码， 一旦被加载到 Web 页面，1 秒后会在页面文档后追加“Hello there”：

```
var oneSecond = 1000 * 1; // one second = 1000 x 1 ms
setTimeout(function() {
    document.write('<p>Hello there.</p>');
}, oneSecond);
```

而 setInterval 允许以指定的时间间隔重复执行函数。如果把下面的代码注入到 Web 页面，会导致每秒钟向页面文档后面追加一句“Hello there”：

```
var oneSecond = 1000 * 1; // one second = 1000 x 1 ms
setInterval(function() {
    document.write('<p>Hello there.</p>');
}, oneSecond);
```

因为 Web 早已成为一个用来构建应用程序的平台，而不再是简单的静态页面，所以这种类似的需求日益浮现。这些任务计划函数帮助开发人员实现表单定期验证，延迟远程数据同步，或者那些需要延时反应的 UI 交互。Node 也完整实现了这些方法。在服务器端，你可以用它们来重复或延迟执行很多任务，比如缓存过期，连接池清理，会话过期，轮询等等。

使用 setTimeout 延迟函数执行

setTimeout 可以制定一个在将来某个时间把指定函数运行一次的执行计划，比如：

```
var timeout_ms = 2000; // 2 seconds
var timeout = setTimeout(function() {
    console.log("timed out!");
}, timeout_ms);
```

和客户端 JavaScript 完全一样，setTimeout 接受两个参数，第一个参数是需要被延迟的函数，第二个参数是延迟时间（以毫秒为单位）。

setTimeout 返回一个超时句柄，它是个内部对象，可以用它作为参数调用 clearTimeout 来取消计时器，除此之外这个句柄没有任何作用。

使用 clearTimeout 取消执行计划

一旦获得了超时句柄，就可以用 clearTimeout 来取消函数执行计划，像这样：

```
var timeoutTime = 1000; // one second
var timeout = setTimeout(function() {
    console.log("timed out!");
}, timeoutTime);
clearTimeout(timeout);
```

这个例子里，计时器永远不会被触发，也不会输出”time out!”这几个字。你也可以在将来的任何时间取消执行计划，就像下面的例子：

```
var timeout = setTimeout(function A() {
    console.log("timed out!");
}, 2000);
setTimeout(function B() {
    clearTimeout(timeout);
}, 1000);
```

代码指定了两个延时执行的函数A和B，函数A计划在2秒钟后执行，B计划在1秒钟后执行，因为函数B先执行，而它取消了A的执行计划，因此A永远不会运行。

制定和取消函数的重复执行计划

setInterval 和 setTimeout 类似，但是它会以指定时间为间隔重复执行一个函数。你可以用它来周期性的触发一段程序，来完成一些类似清理，收集，日志，获取数据，轮询等其它需要重复执行的任务。

下面代码每秒会向控制台输出一句“tick”：

```
var period = 1000; // 1 second
setInterval(function() {
    console.log("tick");
}, period);
```

如果你不想让它永远运行下去，可以用clearInterval()取消定时器。

setInterval返回一个执行计划句柄，可以把它用作clearInterval的参数来取消执行计划：

```
var interval = setInterval(function() {
    console.log("tick");
}, 1000);
// …
clearInterval(interval);
```

使用 process.nextTick 将函数执行延迟到事件循环的下一轮

有时候客户端 JavaScript 程序员用 setTimeout(callback,0)将任务延迟一段很短的时间，第二个参数是 0 毫秒，它告诉 JavaScript 运行时，当所有挂起的事件处理完毕后立刻执行这个回调函数。有时候这种技术被用来延迟执行一些并不需要被立刻执行的操作。比如，有时候

需要在用户事件处理完毕后再开始播放动画或者做一些其它的计算。

Node 中，就像“事件循环”的字面意思，事件循环运行在一个处理事件队列的循环里，事件循环工作过程中的每一轮就称为一个 tick。

你可以在事件循环每次开始下一轮（下一个 tick）执行时调用回调函数一次，这也正是 process.nextTick 的原理，而 setTimeout，setTimeout 使用 JavaScript 运行时内部的执行队列，而不是使用事件循环。

通过使用 process.nextTick(callback) ，而不是 setTimeout(callback, 0)，你的回调函数会在队列内的事件处理完毕后立刻执行，它要比 JavaScript 的超时队列快很多（以 CPU 时间来衡量）。

你可以像下面这样，把函数延迟到下一轮事件循环再运行：

```
process.nextTick(function() {
    my_expensive_computation_function();
});
```

注意：process对象是Node为数不多的全局对象之一。

堵塞事件循环

Node 和 JavaScript 的运行时采用的是单线程事件循环，每次循环，运行时通过调用相关回调函数来处理队列内的下个事件。当事件执行完毕，事件循环取得执行结果并处理下个事件，如此反复，直到事件队列为空。如果其中一个回调函数运行时占用了很长时间，事件循环在那期间就不能处理其它挂起的事件，这会让应用程序或服务变得非常慢。

在处理事件时，如果使用了内存敏感或者处理器敏感的函数，会导致事件循环变得缓慢，而且造成大量事件堆积，不能被及时处理，甚至堵塞队列。

看下面堵塞事件循环的例子：

```
process.nextTick(function nextTick1() {
    var a = 0;
    while(true) {
        a ++;
    }
});
process.nextTick(function nextTick2() {
    console.log("next tick");
});
setTimeout(function timeout() {
    console.log("timeout");
}, 1000);
```

这个例子里，nextTick2和timeout函数无论等待多久都没机会运行，因为事件循环被nextTick函数里的无限循环堵塞了，即使timeout函数被计划在1秒钟后执行它也不会运行。

当使用setTimeout时，回调函数会被添加到执行计划队列，而在这个例子里它们甚至不会被添加到队列。这虽然是个极端例子，但是你可以看到，运行一个处理器敏感的任务时可能会堵塞或者拖慢事件循环。

退出事件循环

使用 process.nextTick，可以把一个非关键性的任务推迟到事件循环的下一轮(tick)再执行，这样可以释放事件循环，让它可以继续执行其它挂起的事件。

看下面例子，如果你打算删除一个临时文件，但是又不想让 data 事件的回调函数等待这个 IO 操作，你可以这样延迟它：

```
stream.on("data", function(data) {
    stream.end("my response");
    process.nextTick(function() {
        fs.unlink("/path/to/file");
    });
});
```

使用 setTimeout 替代 setInterval 来确保函数执行的串行性

假设，你打算设计一个叫 my_async_function 的函数，它可以做某些 I/O 操作（比如解析日志文件）的函数，并打算让它周期性执行，你可以用 setInterval 这样实现它：

```
var interval = 1000;
setInterval(function() {
    my_async_function(function() {
        console.log('my_async_function finished!');
    });
},interval);//译者注：前面加粗部分是我添加的，作者应该是笔误遗漏了
```

你必须能确保这些函数不会被同时执行，但是如果使用setinterval你无法保证这一点，假如my_async_function函数运行的时间比interval变量多了一毫秒，它们就会被同时执行，而不是按次序串行执行。

译者注：（下面粗体部分为译者添加，非原书内容）

为了方便理解这部分内容，可以修改下作者的代码，让它可以实际运行：

```
var interval = 1000;
setInterval(function(){
    (function my_async_function(){
        setTimeout(function(){
            console.log("1");
        },5000);
    })();
},interval);
```

运行下这段代码看看，你会发现，等待5秒钟后，“hello ”被每隔1秒输出一次。而我们期望是，当前my_async_function执行完毕（耗费5秒）后，等待1秒再执行下一个my_async_function，每次输出之间应该间隔6秒才对。造成这种结果，是因为*my_async_function*不是串行执行的，而是多个在同时运行。

因此，你需要一种办法来强制使一个my_async_function执行结束到下个my_async_function开始执行之间的间隔时间正好是interval变量指定的时间。你可以这样做：

```
var interval = 1000; // 1 秒

(function schedule() {    //第3行
    setTimeout(function do_it() {
        my_async_function(function() {    //第5行
            console.log('async is done!');
            schedule();
        });
    }, interval);
}());        //第10行
```

前面代码里，声明了一个叫schedule的函数（第3行），并且在声明后立刻调用它（第10行），schedule函数会在1秒（由interval指定）后运行do_it函数。1秒钟过后，第5行的my_async_function函数会被调用，当它执行完毕后，会调用它自己的那个匿名回调函数（第6行），而这个匿名回调函数又会再次重置do_it的执行计划，让它1秒钟后重新执行，这样代码就开始串行地不断循环执行了。

小结

可以用 setTimeout()函数预先设定函数的执行计划，并用 clearTimeout()函数取消它。还可以用 setInterval()周期性的重复执行某个函数，相应的，可以使用 clearInterval()取消这个重复执行计划。

如果因为使用了一个处理器敏感的操作而堵塞了事件循环，那些原计划应该被执行的函数将会被延迟，甚至永远无法执行。所以不要在事件循环内使用 CPU 敏感的操作。还有，你可以使用 process.nextTick()把函数的执行延迟到事件循环的下一轮。

I/O 和 setInterval()一起使用时，你无法保证在任何时间点只有一个挂起的调用，但是，你可以使用递归函数和 setTimeout()函数来回避这个棘手的问题。

译者注：

本文对应原文第二部分第六章：Node Core API Basics：Scheduling the Execution of Functions Using Timers

本系列文章列表和翻译进度，请移步：Node.js 高级编程：用 Javascript 构建可伸缩应用（〇）