浏览器的javascript引擎

浏览器通过内置的javascript引擎，读取网页中的代码，对其后台进行处理。

概述

javascript代码嵌入方法

（1）直接添加代码块

```
<script>
//some javascript code
</script>
```

（2）加载外部脚本

```
<script src="example.js"></script>
```

如果脚本使用了非英文字符，还应该注明编码。

```
<script charset="utf-8" src="example.js"></script>
```

加载外部脚本和直接添加代码块，这两种方法不能混用。下面的console.log语句直接被忽略。

```
<script charset="utf-8" src="example.js">
    console.log('Hello World!');
</script>
```

（3）行内代码

HTML语言允许在某些元素的事件属性和a元素的href属性中，直接写入javascript

```
<div onclick="alert('Hello')"></div>

<a href="javascript:alert('Hello')"></a>
```

将HTML代码与javascript代码混写在一起，不利于代码管理。

外部脚本的加载和阻塞效应

下载和执行外部javascript代码时，浏览器会暂停页面渲染。

如果某段代码的下载或执行时间特别长，浏览器就会呈现假死状态，失去响应。

将Script标签放在底部就可以避免这种情况。

将脚本文件放在网页底部还有一个好处。在DOM结构生成之前就调用DOM，javascript会报错，放在尾部加载就不会碰到这个问题。

```
<head>
<script>
    console.log(document.body.innerHTML);
</script>
</head>
```

以上代码会报错。

> 了解更多 [浏览器加载和渲染html的顺序

如果是多个Script标签

```
<script src="1.js"></script>
<script src="2.js"></script>
```

会平行下载1.js和2.js,但执行时会先执行1.js,然后执行2.js。

脚本的执行顺序由他们在页面中的出现顺序决定，这是为了保证脚本之间的依赖关系不受到破坏。

解决脚本文件下载阻塞网页渲染问题

加入defer属性

```
<script src="1.js" defer></script>
<script src="2.js" defer></script>
```

加入defer属性之后，浏览器下载脚本文件的时候，不会阻塞页面渲染。下载的脚本在DOMContentLoaded事件触发前执行（即刚刚读取完标签），可以保证脚本的执行顺序。

> 1. 浏览器支持不理想，IE（<=9）,无法保证js执行顺序（2.js在1.js之后执行）。
> 2. 对内置代码，直接添加的代码块，动态生成的Script标签不起作用。

加入async属性

```
<script src="1.js" async></script>
<script src="2.js" async></script>
```

anync属性可以保证脚本下载的同时，浏览器继续渲染。渲染完成后执行脚本。

> 1. 无法保证脚本执行顺序，哪个先下载结束就先执行那个。
> 2. 只支持ie10+
> 3. 同时使用defer和async,defer不起作用

资源文件的下载

对于来自同一个域名的资源，比如脚本文件、样式表文件、图片文件等，浏览器一般最多同时下载六个。如果是来自不同域名的资源，就没有这个限制。可以把静态文件放在不同的域名之下，加快下载速度。

动态嵌入

```
var src=['1.js', '2.js'];
for(var i=0 ; i<src.length;i++){
    var script = document.createElement('script');
    script.src =src[i];
    document.head.appendChild(script);
}
```

> 优点：动态生成的Script标签不会阻塞页面渲染，也就是不会造成浏览器假死。
>
> 缺点：无法保证脚本的执行顺序，哪个文本先下载完成就先执行哪个。

将anync设置为false可以解决执行顺序问题问题

```
var src=['1.js', '2.js'];
for(var i=0 ; i<src.length;i++){
    var script = document.createElement('script');
    script.src =src[i];
    script.async=false;
    document.head.appendChild(script);
}
```

当Script标签指定的外部脚本文件下载和解析完成，会触发一个load事件。可以为这个事件指定回调函数。

```
<script async src="jquery.min.js" onload="console.log('jQuery已加载！')"></script>
```

javascript虚拟机——javascript引擎

JavaScript是一种解释型语言，也就是说，它不需要编译，可以由解释器实时运行。

> 优点：运行和修改方便，刷新页面就可以重新解释。
>
> 缺点：每次调用都要调用解析器，体统开销大，运行速度慢于编译语言。

目前浏览器都将javascript进行了一定程度的编译，生成类似字节码（bytecode）的中间代码，以提高运行速度。

什么是javascript引擎

javascript引擎就是能够“读懂”javascript代码，并给出代码运行结果的一段程序。比如你写了var a=1+1; javascript引擎做的事情就是看懂（解析）这段代码，并且将a的值变为2。

对于静态语言（如java、c++、c）来说,处理上述事情的叫做**编译器（compiler）**。对于javascript这样的动态语言则叫做**解释器（interpret）。编译器是将源代码编译为另一种代码（比如机器码，或者字节码），而解析器是直接解析并将代码运行结果输出。**

现在javascript引擎已经很难界定算是解析器还是编译器了。

> 静态（类型）语言：在运行前编译时检查类型。在写代码时，每声明一个变量必须指定类型。
>
> 动态（类型）语言：在运行期间检查数据类型的语言。用这类语言编程，不会给变量指定类型，而是在赋值时得到数据类型。

javascript引擎和ECMAScript是什么关系？

javascript引擎是一段程序，我们写的javascript代码也是程序，如何让程序去读懂程序呢？这就需要定义规则。ECMAScript 262 就定义了一套完整的标准。标准的javascript引擎就会根据这套文档去实现。

ECMAScript定义了语言的标准，javascript引擎根据它来实现。

javascript引擎与浏览器又有什么关系

javascript引擎是浏览器的组成部分之一。浏览器还要做很多别的事情，比如解析页面、渲染页面、cookie管理、历史记录等等。

不同的浏览器采用不同的javascript引擎。

JIT

早期浏览器内部对javascript的处理过程

1. 读取代码，进行词法分析（Lexical analysis），将代码分解成词元（token）。
2. 对词元进行语法分析（parsing），将代码整理成“语法树”（syntax tree）。
3. 使用“翻译器”（translator），将代码转为字节码（bytecode）。
4. 使用“字节码解释器”（bytecode interpreter），将字节码转为机器码。

逐行解释将字节码转为机器码，是很低效的。

即时编译（Just In Time compiler）:字节码只在运行时编译，用到哪一行就编译到哪一行，并且把编译结果缓存（inline cache）。通常，一个程序被经常用到的，只是其中一小部分代码，有了缓存的编译结果，整个程序的运行速度就会显著提升。

不同的浏览器有不同的编译策略。有的浏览器只编译最经常用到的部分，比如循环的部分；有的浏览器索性省略了字节码的翻译步骤，直接编译成机器码，比如chrome浏览器的V8引擎。

最常见的javascript引擎

- Chakra(Microsoft Internet Explorer)
- Nitro/JavaScript Core (Safari)
- Carakan (Opera)
- SpiderMonkey (Firefox)
- V8 (Chrome, Chromium)

单线程模型

Event Loop

Javascript采用单线程模型，也就是说，所有任务都在一个线程里运行。这意味着一次只能运行一个任务，其他任务都必须在后面排队等待。

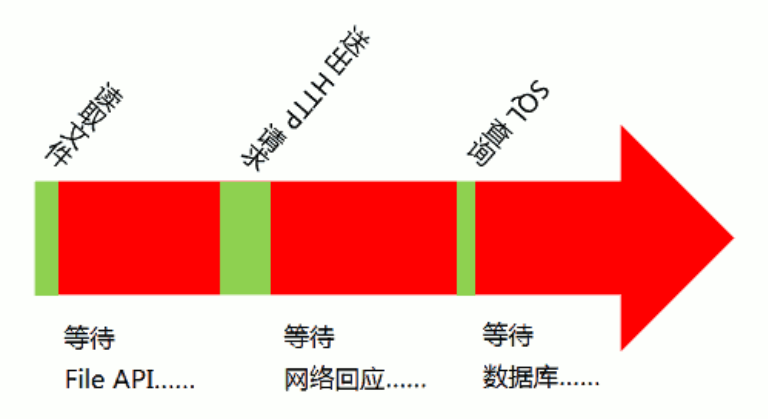
> 问题：新的任务被添加在队列的尾部，只有所有的任务运行结束，才会轮到它执行。如果有一个任务特别耗时，后面的任务都会停在那里等待，造成浏览器失去响应，又称作“假死”。

为了解决这个问题，javascript采用了**Event Loop**机制。

运行以后的程序叫做“进程”（process）,一般情况下，一个进程一次只能执行一个任务。如果有很多任务需要执行，解决方案一共有以下三种。

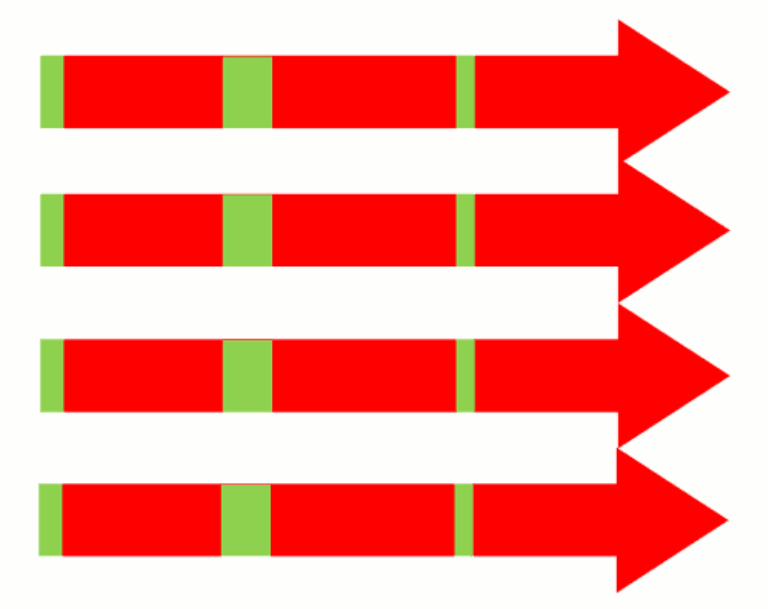
1. 排队。一个进程一次只能执行一个任务，只好等待前面的任务执行完毕，再执行后面的任务。 **2. 新建进程。**使用fork命令，为每个任务新建一个进程 **3. 新建线程。**因为进程太耗资源，所以如今的程序往往允许一个进程包含多个线程，由线程去完成任务。

如果某个任务很耗时，比如涉及很多I/O操作，那么线程大概如下



上图的绿色部分是程序的运行时间，红色部分是等待时间。由于I/O操作很慢，所以大部分运行时间都在空等I/O操作返回结果。这就是**“同步模式”**（synchronous I/O）。

如果采用多线程，同时运行多个任务，很可能像下面这样。

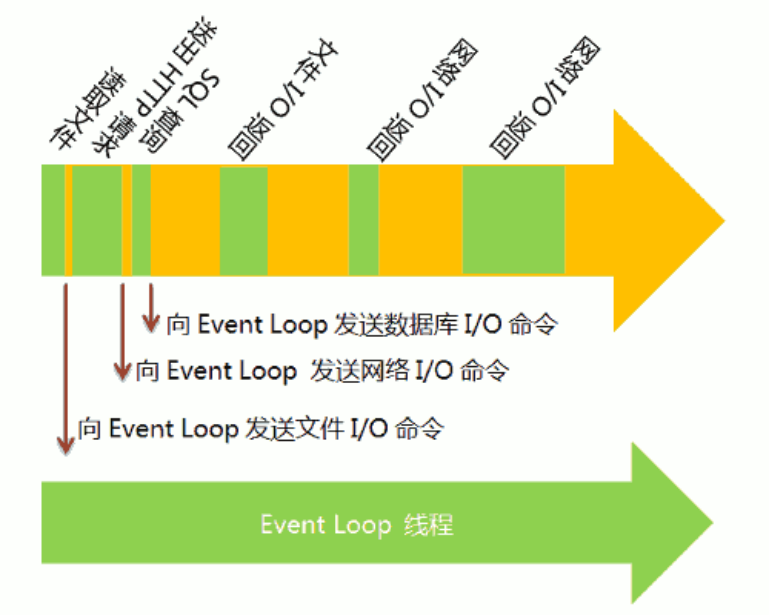


多线程不仅占用多倍的系统资源，也闲置了多倍的资源。

Event Loop就是为了解决这个问题而提出的。

Event Loop 是一个程序结构，用于等待和发送消息事件。

简单说，就是在程序中设置两个线程：一个负责本身程序的运行，称为“主线程”；另一个负责主线程与其他进程（主要是各种I/O操作）的通信，被称为“Event Loop线程”（消息线程）。



绿色部分表示运行时间，橙色部分表示空闲时间。每当遇到I/O时，主线程就让Event Loop线程去通知I/O程序，然后接着往后运行，所以不存在红色的等待时间。等到I/O程序完成操作，Event Loop线程再把结果返回主线程。主线程就调用事先设定的回调函数，完成整个任务。

这种方式被称为“**异步模式**”（asynchronous I/O）。

这正是javascript语言的运行方式。单线程模型虽然对JavaScript构成了很大的限制，但也因此使它具备了其他语言不具备的优势。如果部署得好，JavaScript程序是不会出现堵塞的，这就是为什么node.js平台可以用很少的资源，应付大流量访问的原因。

> 更多相关
>
> 1. socket阻塞与非阻塞，同步与异步
> 2. Node.js异步式I/O与事件式编程
> 3. 进程与线程及其区别 概念版
> 4. 进程和线程的区别 c#版

间歇调用和超时调用

javascript是单线程语言，但它允许通过设置超时值和间歇时间值来调度代码在特定的时刻执行。

超时调用——setTimeout()

参数：

- 要执行的代码 —— 一个包含javascript代码的字符串或函数
- 以毫秒表示的时间

```
//不建议传递字符串，因为传递字符串可能会导致性能损失
setTimeout("alert('Hello World')",1000);

//推荐调用方式
setTimeout(function(){
    alert("hello world");
},1000)
```

第二个参数表示等待时长，但是经过该时间的代码不一定会执行。

javascript是一个单线程解释器，一定时间内只能执行一段代码。为了要控制要执行的代码，就有一个javascript任务队列。这些任务会按照添加顺序执行。setTimeout()的第二个参数告诉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，会马上被执行；如果队列不是空的，就要等前面的代码执行完了以后再执行。

调用setTimeout()之后会返回一个数值ID，表示超时调用。这个超时调用ID是计划执行代码的唯一标识符，可以通过它来取消超时调用。

```
var timeoutId=setTimeout(function(){
    alert("Hello world");
},1000);

clearTimeout(timeoutId);
```

> setTimeout()方法是window对象的，因此都是在全局作用域中执行的。

间歇调用——setInterval()

按照指定时间间隔重复执行代码，直至间歇调用被取消或者页面被卸载。其他类似setTimeout();

```
var num = 0;
var max = 10;
var intervalId = null;
function incrementNumber() {
    num++;
    if (num == max) {
        clearInterval(intervalId);
        alert(”Done”);
    }
}
intervalId = setInterval(incrementNumber, 500);
```

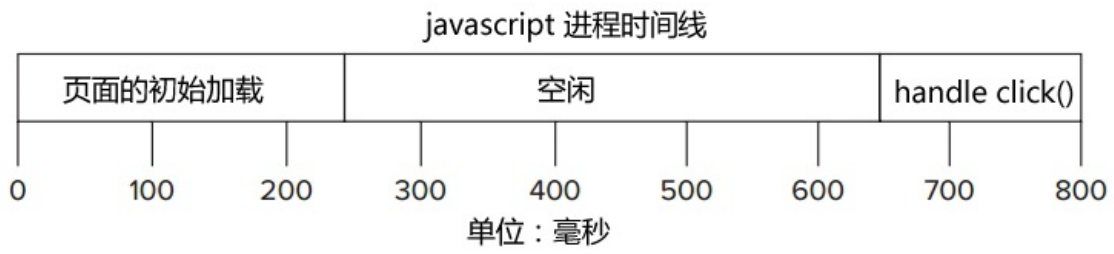
可以使用setTimeout（）模拟

```
var num=0;
var max=10;

function incrementNumber(){
    num++;
    if(num<max){
        setTimeout(incrementNumber,500);
    }else{
        alert("done");
    }
}

setTimeout(incrementNumber,500);
```

一般认为，使用超时调用来模拟间歇调用是一种最佳模式。

高级定时器

可以把javascript想象成在时间线上运行的。

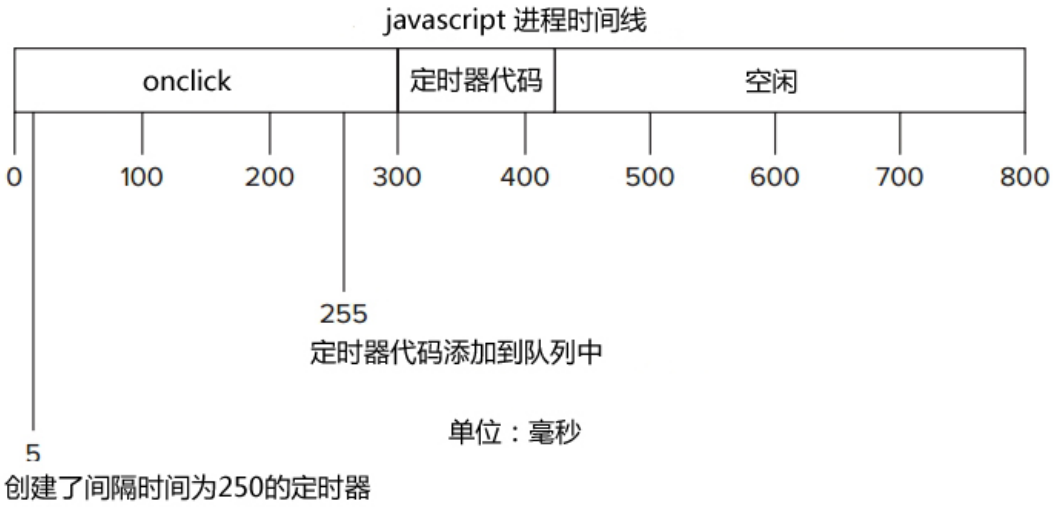


除了主javascript执行程序外，还有一个需要在进程下一次空闲时执行的代码队列。（详情见Event Loop）.

```
var btn = document.getElementById(“my-btn”);
    btn.onclick = function(){
    setTimeout(function(){
        document.getElementById(“message”).style.visibility = “visible”;
    }, 250);
```

```
    //other code
};
```

在这里给按钮设置了一个事件处理程序。事件处理程序设置了一个250ms后调用的定时器。首先将onclick事件处理程序加入队列。执行后设定计时器，250ms后定时器中的代码被添加到队列中。如果前面的onclick事件执行了300ms,那么定时请代码至少要在300ms后执行。



执行完一套代码后，javascript进程返回一段很短的时间，这样页面上的其他处理就可以进行了。

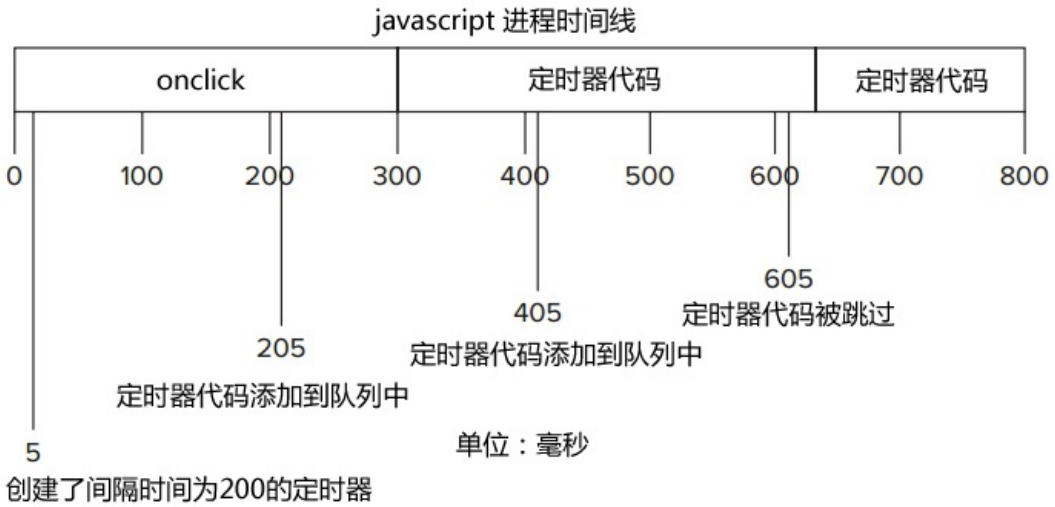
重复的定时器

setInterval()的问题在于，定时器代码可能在代码再次被添加到列队之前还没有完成执行，结果导致定时器代码连续运行好几次，而之间没有任何停顿。

javascript引擎的解决方案：仅当没有该定时器的任何其他代码实例时，才将定时器代码添加到队列中

导致的问题：

1. 某些间隔会被跳过。
2. 多个定时器的代码执行之间的间隔可能会比预期的小



解决方案：使用链式setTimeout()

```
setTimeout(function(){
    //处理中
    setTimeout(arguments.callee, interval);
}, interval);
```

> callee： 返回正被执行的 Function 对象，也就是所指定的 Function 对象的正文。callee 属性是 arguments 对象的一个成员，它表示对函数对象

> 本身的引用，这有利于匿名函数的递归或者保证函数的封装性

数组分块（array chunking）

运行在浏览器中的javascript都被分配了一个确定的数量的资源。为了防止恶意的web程序员把用户的计算机搞挂，javascript需要的内存大小和处理时间都被限制了。

其中一个限制是长时间运行脚本的限制，如果代码运行超过特定的时间或者特定语句数量就不让他执行。如果达到了这个限制就回弹出一个错误对话框，告诉用户某个脚本会用过长时间执行，询问时继续执行还是停止他

造成问题原因

1. 过长的、过深嵌套的函数调用
2. 进行大量处理的循环

后者较为容易解决，长时间运行的循环一般遵循如下模式

```
for(var i=0;i<data.length;i++){
    prosess(data[i]);
}
```

此模式的问题在于data.length数量是未知的。如果process（）要花100ms,data.length=10的话，脚本就需要运行1秒钟才能完成。length越大，脚本运行时间越长，用户无法与用户交互的时间越长。

我们可以用定时器分割这个循环，来解决这个问题。这种技术叫做数组分块（array chucking）技术。基本思路是为要处理的项目创建一个队列，然后使用定时器取出下一个要处理的项目进行处理，接着再设定另一个定时器。

```
setTimeout(function(){
    //取出一个条目并处理
    var item =array.shift();
    pocess(item);

    //若还有条目，再设置另一个定时器
    if(array.length>0){
        setTimeout(arguments.callee,100);
    }
},100);
```

实现数组分块，可以使用下面的函数：

```
function chunk(array,process,context){
    setTimeout(function(){
        var item=array.shift();
        process.call(context,item);
        if(array.length>0){
            setTimeout(arguments.callee,100);
        }
    },100);
}
```

chunk()方法接受三个参数：要处理项目的数组，用于处理项目的函数，以及可选的运行该函数的环境。通过call()调用函数，可以设置一个合适的执行环境。

函数的使用：

```
var data = [12,123,1234,453,436,23,23,5,4123,45,346,5634,2234,345,342];
    function printValue(item){
        var div = document.getElementById("myDiv");
        div.innerHTML += item + "<br>";
    }
chunk(data, printValue);
```

需要注意的是，传递给chunk()的数组是用作一个队列的，因此当处理数据的同时，数组中的条目也在改变。如果想保持原数组不变，则应该传一个副本给chunk();

chunk(data.contat(),printValue);

> 全面理解javascript的caller,callee,call,apply概念(修改版)