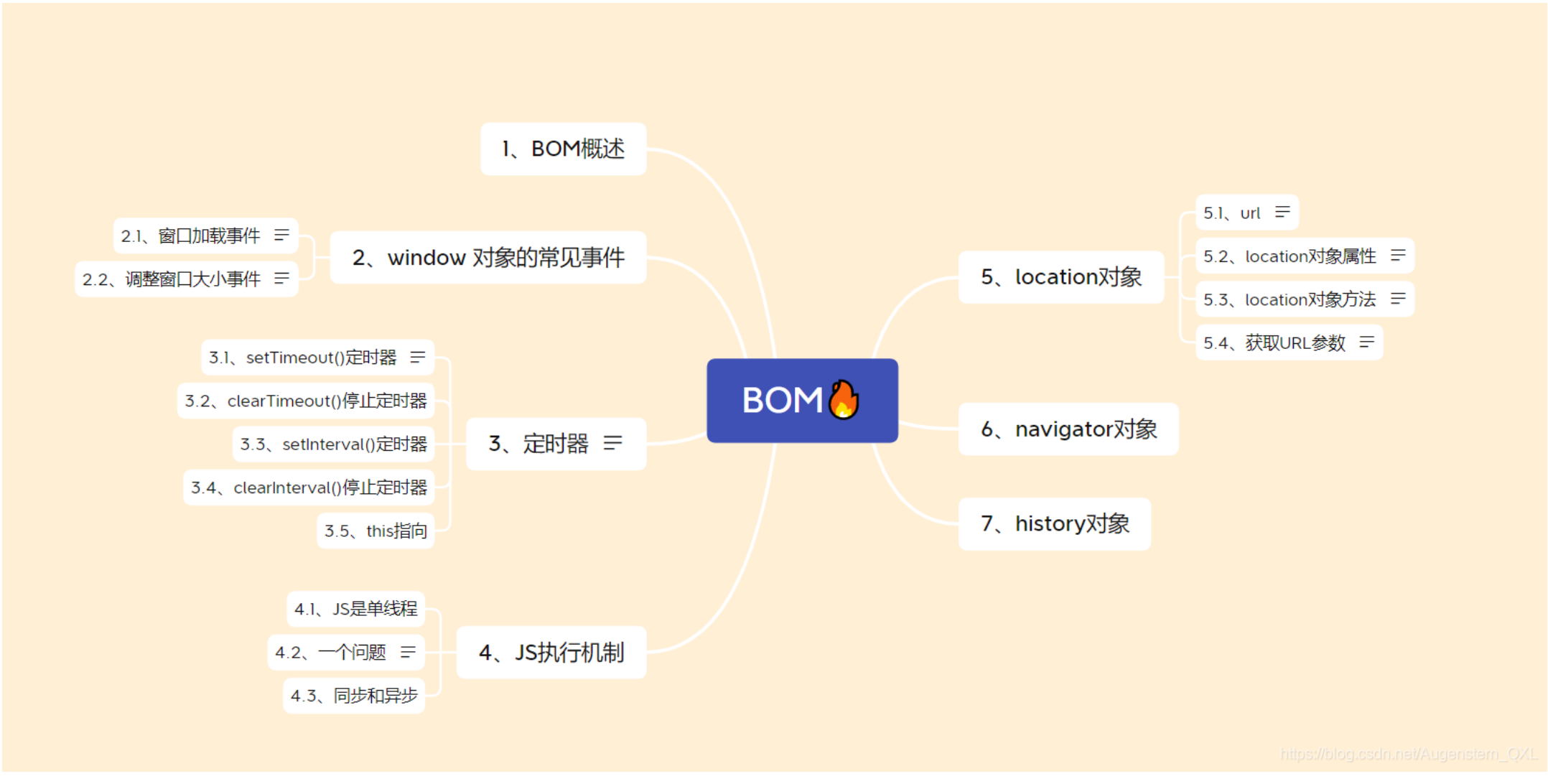
JavaScript进阶班之BOM技术(五)

✍JavaScript帝国之行🔥

内容	地址
JavaScript基础大总结(一) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119249534
JavaScript基础之函数与作用域(二) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119250991
JavaScript基础之对象与内置对象(三) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/119250137
JavaScript进阶之DOM技术(四) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115416921
JavaScript进阶之BOM技术(五) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115406408
JavaScript提高之面向对象(六) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115219073
JavaScript提高之ES6(七) 🔥	https://blog.csdn.net/Augenstern_QXL/article/details/115344398

✍目录总览



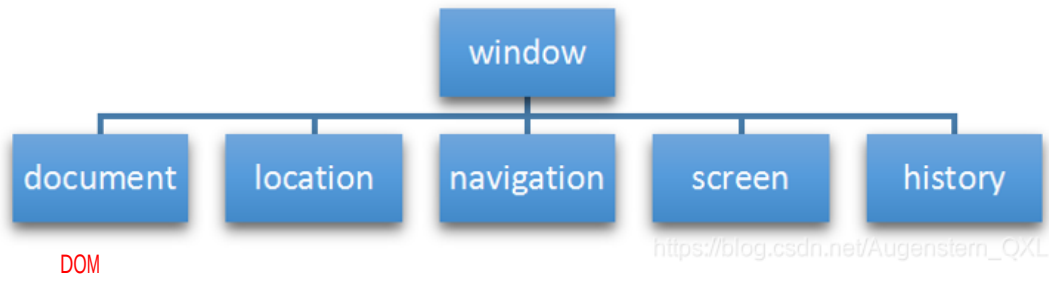
1、BOM概述

- BOM = Browser Object Model 👉浏览器对象模型
- 它提供了独立于内容而与浏览器窗口进行交互的对象，其核心对象是 window
- BOM 由一系列相关的对象构成，并且每个对象都提供了很多方法与属性
- BOM 缺乏标准，JavaScript 语法的标准化组织是 ECMA, DOM 的标准化组织是 W3C, BOM最初是Netscape 浏览器标准的一部分

操作页面元素　　浏览器交互

DOM	BOM
文档对象模型	浏览器对象模型
DOM 就是把 文档 当作一个对象来看待	把 浏览器当作一个对象来看待
DOM 的顶级对象是 document	BOM 的顶级对象是 window
DOM 主要学习的是操作页面元素	BOM 学习的是浏览器窗口交互的一些对象
DOM 是 W3C 标准规范	BOM 是浏览器厂商在各自浏览器上定义的，兼容性较差

1.1、BOM的构成



- BOM 比 DOM 更大。它包含 DOM。
- window 对象是浏览器的顶级对象，它具有双重角色

- 它是 JS 访问浏览器窗口的一个接口
- 它是一个全局对象。定义在全局作用域中的变量、函数都会变成 window 对象的属性和方法
- 在调用的时候可以省略 window，前面学习的对话框都属于 window 对象方法，如 `alert()、prompt()` 等。
- **注意**：window下的一个特殊属性 window.name

```
// 定义在全局作用域中的变量会变成window对象的属性
var num = 10;
console.log(window.num);
// 10

// 定义在全局作用域中的函数会变成window对象的方法
function fn() {
    console.log(11);
}
console.fn();
// 11

var name = 10;  //不要用这个name变量,window下有一个特殊属性window.name
console.log(window.num);
```

2、window 对象的常见事件

2.1、窗口加载事件

`window.onload` 是窗口（页面）加载事件，当文档内容完全加载完成会触发该事件（包括图像，脚本文件，CSS文件等），就调用的处理函数。

```
window.onload = function(){

};

// 或者
window.addEventListener("load",function(){});
```

注意：

- 有了 `window.onload` 就可以把JS代码写到页面元素的上方
- 因为 `onload` 是等页面内容全部加载完毕，再去执行处理函数
- `window.onload` 传统注册事件方式，只能写一次
- 如果有多个，会以最后一个 `window.onload` 为准
- **如果使用addEventListener 则没有限制**

```
document.addEventListener('DOMContentLoaded',function(){})
```

接收两个参数：

- DOMCountentLoaded事件触发时，仅当DOM加载完成，不包括样式表，图片，flash等等
- 如果页面的图片很多的话, 从用户访问到onload触发可能需要较长的时间
- 交互效果就不能实现，必然影响用户的体验，此时用 `DOMContentLoaded` 事件比较合适。

2.1.1、区别

- `load` 等页面内容全部加载完毕，包括页面dom元素，图片，flash，css等
- `DOMContentLoaded` 是DOM加载完毕，不包含图片 flash css 等就可以执行，加载速度比load更快一些 这个更好

```
<script>
    // window.onload = function() {
    //     var btn = document.querySelector('button');
    //     btn.addEventListener('click', function() {
    //         alert('点击我');
    //     })
    // }
    // window.onload = function() {
    //     alert(22);
    // }

    window.addEventListener('load', function() {
        var btn = document.querySelector('button');
        btn.addEventListener('click', function() {
            alert('点击我');
        })
    })
    window.addEventListener('load', function() {

        alert(22);
    })
    document.addEventListener('DOMContentLoaded', function() {
            alert(33);
        })
        // load 等页面内容全部加载完毕，包含页面dom元素 图片 flash  css 等等
        // DOMContentLoaded 是DOM 加载完毕，不包含图片 falsh css 等就可以执行 加载速度比 load更快一些
</script>
```

2.2、调整窗口大小事件

`window.onresize` 是调整窗口大小加载事件，当触发时就调用的处理函数

```
window.onresize = function() {}

// 或者
window.addEventListener('resize',function(){});
```

- 只要窗口大小发生像素变化，就会触发这个事件
- 我们经常利用这个事件完成响应式布局。 `window.innerWidth` 当前屏幕的宽度

window.innerHeight是高度

```
<body>
    <script>
        window.addEventListener('load', function() {
            var div = document.querySelector('div');
            window.addEventListener('resize', function() {
                console.log(window.innerWidth);

                console.log('变化了');
                if (window.innerWidth <= 800) {
                    div.style.display = 'none';
                } else {
                    div.style.display = 'block';
                }

            })
        })
    </script>
    <div></div>
</body>
```

3、定时器

window 对象给我们提供了两个定时器

- `setTimeout()`
- `setInterval()`

3.1、setTimeout()定时器

`setTimeout()` 方法用于设置一个定时器，该定时器在定时器到期后执行调用函数。

```
window.setTimeout(调用函数,[延迟的毫秒数]);
```

[延迟毫秒数]可以省略，当毫秒数到了，就去调用前面的调用函数
1秒=1000毫秒

注意:

- `window` 可以省略
- 这个调用函数
  - 可以直接写函数
  - 或者写函数名
  - 或者采取字符串 **‘函数名()’**（不推荐）
- 延迟的毫秒数省略默认是0，如果写，必须是毫秒
- 因为定时器可能有很多，所以我们经常给定时器赋值一个标识符
- `setTimeout()` 这个调用函数我们也称为**回调函数** callback
- 普通函数是按照代码顺序直接调用，而这个函数，需要等待事件，事件到了才会去调用这个函数，因此称为回调函数。

```
<body>
    <script>
        // 1. setTimeout
        // 语法规范：   window.setTimeout(调用函数, 延时时间);
        // 1. 这个window在调用的时候可以省略
        // 2. 这个延时时间单位是毫秒 但是可以省略，如果省略默认的是0
        // 3. 这个调用函数可以直接写函数 还可以写 函数名 还有一个写法 '函数名()'
        // 4. 页面中可能有很多的定时器，我们经常给定时器加标识符 （名字）
        // setTimeout(function() {
        //     console.log('时间到了');

        // }, 2000);
        function callback() {
            console.log('爆炸了');

        }
        var timer1 = setTimeout(callback, 3000);
        var timer2 = setTimeout(callback, 5000);
        // setTimeout('callback()', 3000); // 我们不提倡这个写法
    </script>
</body>
```

3.2、clearTimeout()停止定时器

- `clearTimeout()` 方法取消了先前通过调用 `setTimeout()` 建立的定时器

```
window.clearTimeout(timeoutID)
```

注意:

- `window` 可以省略
- 里面的参数就是定时器的标识符

```
<body>
    <button>点击停止定时器</button>
    <script>
        var btn = document.querySelector('button');
        var timer = setTimeout(function() {
            console.log('爆炸了');
        }, 5000);
        btn.addEventListener('click', function() {
            clearTimeout(timer);
        })
    </script>
</body>
```

3.3、setInterval()定时器

- `setInterval()` 方法重复调用一个函数，每隔这个时间，就去调用一次回调函数

setTimeout：里面的回调函数调用几次就重复几次
setinterval：自己重复调用，隔一段[间隔毫秒数]就调用一次

```
window.setInterval(回调函数,[间隔的毫秒数]);
```

- `window` 可以省略
- 这个回调函数:
  - 可以直接写函数
  - 或者写函数名
  - 或者采取字符 '函数名()'
- 第一次执行也是间隔毫秒数之后执行，之后每隔毫秒数就执行一次

```
<body>
    <script>
        // 1. setInterval
        // 语法规范:   window.setInterval(调用函数, 延时时间);
        setInterval(function() {
            console.log('继续输出');

        }, 1000);
        // 2. setTimeout   延时时间到了，就去调用这个回调函数，只调用一次 就结束了这个定时器
        // 3. setInterval   每隔这个延时时间，就去调用这个回调函数，会调用很多次，重复调用这个函数
    </script>
</body>
```

3.4、clearInterval()停止定时器

- `clearInterval ( )` 方法取消了先前通过调用 `setInterval()` 建立的定时器

注意:

- `window` 可以省略
- 里面的参数就是定时器的标识符

```
<body>
    <button class="begin">开启定时器</button>
    <button class="stop">停止定时器</button>
    <script>
        var begin = document.querySelector('.begin');
        var stop = document.querySelector('.stop');
        var timer = null; // 全局变量   null是一个空对象
        begin.addEventListener('click', function() {
            timer = setInterval(function() {
                console.log('ni hao ma');

            }, 1000);
        })
        stop.addEventListener('click', function() {
            clearInterval(timer);
        })
    </script>
</body>
```

3.5、this指向

- `this` 的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定 `this` 到底指向谁

现阶段，我们先了解一下几个this指向

- 全局作用域或者普通函数中 `this` 指向全局对象 `window` (注意定时器里面的this指向window)
- 方法调用中谁调用 `this` 指向谁
- 构造函数中 `this` 指向构造函数实例

this指向的是执行环境的作用域对象

```
<body>
    <button>点击</button>
    <script>
```

```
        // this 指向问题 一般情况下this的最终指向的是那个调用它的对象

        // 1. 全局作用域或者普通函数中this指向全局对象window ( 注意定时器里面的this指向window)
        console.log(this);

        function fn() {
            console.log(this);

        }
        window.fn();
        window.setTimeout(function() {
            console.log(this);

        }, 1000);
        // 2. 方法调用中谁调用this指向谁
        var o = {
            sayHi: function() {
                console.log(this); // this指向的是 o 这个对象

            }
        }
        o.sayHi();
        var btn = document.querySelector('button');
        // btn.onclick = function() {
        //     console.log(this); // this指向的是btn这个按钮对象

        // }
        btn.addEventListener('click', function() {
                console.log(this); // this指向的是btn这个按钮对象

            })
            // 3. 构造函数中this指向构造函数的实例
        function Fun() {
            console.log(this); // this 指向的是fun 实例对象

        }
        var fun = new Fun();
    </script>
</body>
```

4、JS执行机制

4.1、JS是单线程

- JavaScript 语言的一大特点就是**单线程**，也就是说，**同一个时间只能做一件事**。这是因为 Javascript 这门脚本语言诞生的使命所致——JavaScript 是为处理页面中用户的交互，以及操作 DOM 而诞生的。比如我们对某个 DOM 元素进行添加和删除操作，不能同时进行。 应该先进行添加，之后再删除。
- 单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。这样所导致的问题是： 如果 JS 执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的感觉。

4.2、一个问题

以下代码执行的结果是什么?

```
console.log(1);
setTimeout(function() {
    console.log(3);
},1000);
console.log(2);
```

那么以下代码执行的结果又是什么?

```
console.log(1);
setTimeout(function() {
    console.log(3);
},0);
console.log(2);
```

4.3、同步和异步

- 为了解决这个问题，利用多核 CPU 的计算能力，HTML5 提出 Web Worker 标准，允许 JavaScript 脚本创建多个线程
- 于是，JS 中出现了同步和异步。
- 同步:
  - 前一个任务结束后再执行后一个任务
- 异步:
  - 在做这件事的同时，你还可以去处理其他事情

> 同步任务

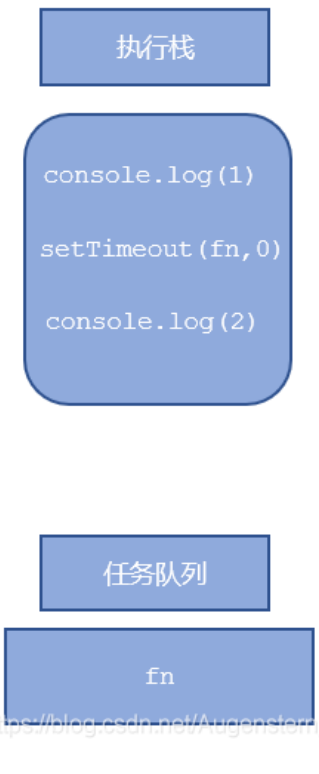
- 同步任务都在主线程上执行，形成一个 执行栈

> 异步任务

- JS中的异步是通过回调函数实现的
- 异步任务有以下三种类型
  - 普通事件，如 `click` , `resize` 等
  - 资源加载，如 `load` , `error` 等

- 定时器，包括 setInterval , setTimeout 等

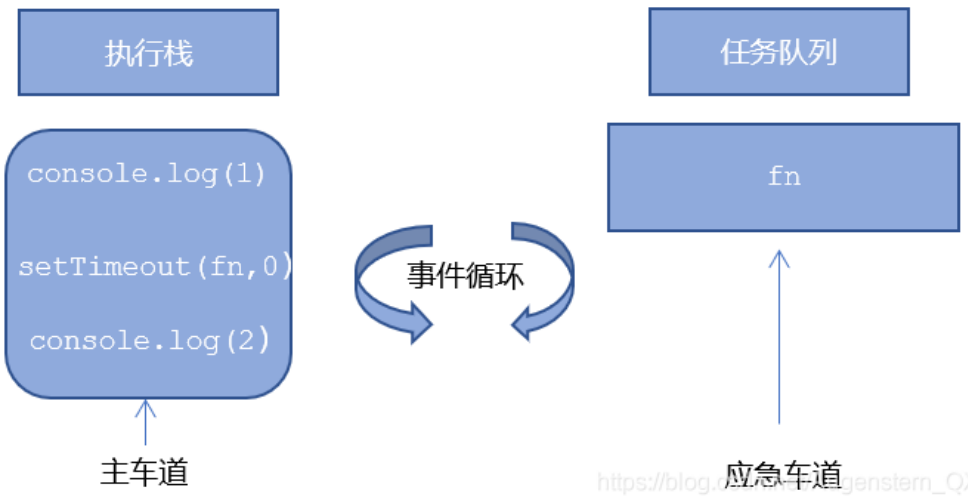
- 异步任务相关回调函数添加到任务队列中



JS执行机制

先同后异

1. 先执行执行栈中的同步任务
2. 异步任务(回调函数)放入任务队列中
3. 一旦执行栈中的所有同步任务执行完毕，系统就会按次序读取任务队列中的异步任务，于是被读取的异步任务结束等待状态，进入执行栈，开始执行



此时再来看我们刚才的问题：

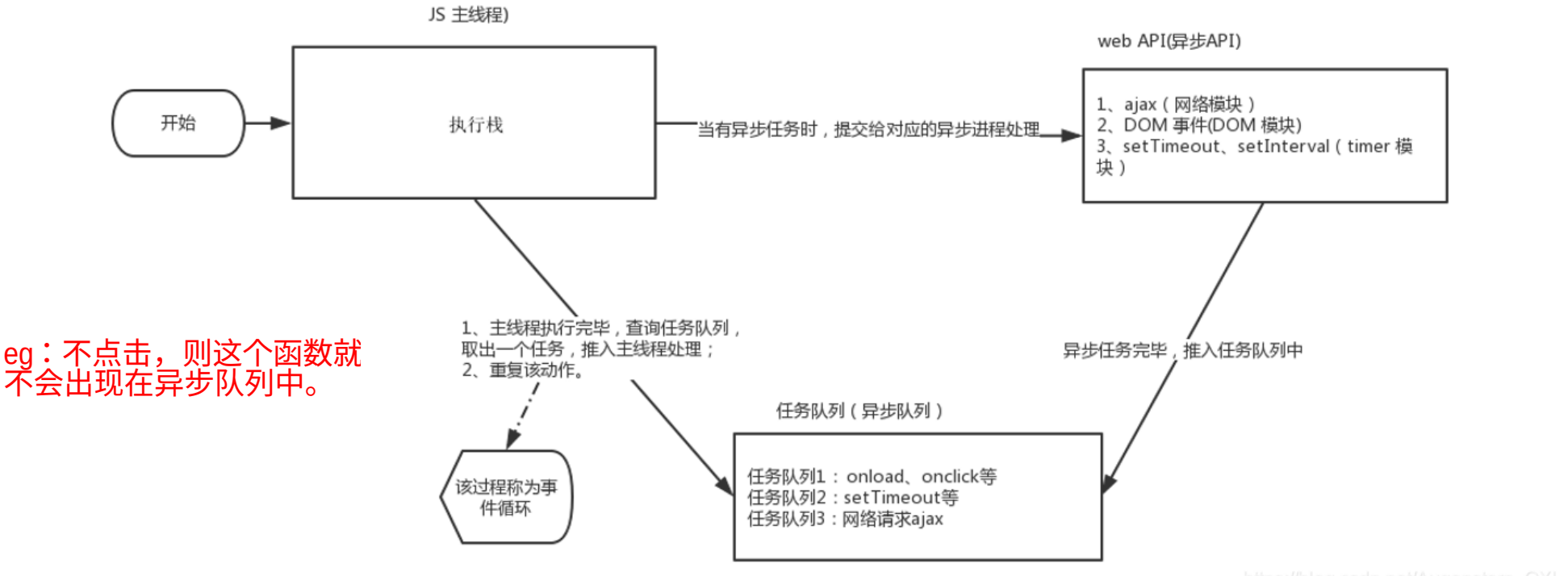
```
console.log(1);
setTimeout(function() {
    console.log(3);
},1000);
console.log(2);
```

- 执行的结果和顺序为 1、2、3

```
console.log(1);
setTimeout(function() {
    console.log(3);
},0);
console.log(2);
```

- 执行的结果和顺序为 1、2、3

```
// 3. 第三个问题
console.log(1);
document.onclick = function() {
    console.log('click');
}
console.log(2);
setTimeout(function() {
    console.log(3)
}, 3000)
```



同步任务放在执行栈中执行，异步任务由异步进程处理放到任务队列中，执行栈中的任务执行完毕会去任务队列中查看是否有异步任务执行，由于主线程不断的重复获得任务、执行任务、再获取任务、再执行，所以这种机制被称为事件循环（ event loop）。

5、location对象

- window 对象给我们提供了一个 `location` 属性用于获取或者设置窗体的url，并且可以解析url。因为这个属性返回的是一个对象，所以我们将这个属性也称为 location 对象。

5.1、url

==统一资源定位符（uniform resouce locator）==是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的 URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

url 的一般语法格式为：

```
protocol://host[:port]/path/[?query]#fragment

http://www.itcast.cn/index.html?name=andy&age=18#link
```

组成	说明
protocol	通信协议 常用的http,ftp,maito等
host	主机(域名) www.itheima.com
port	端口号，可选
path	路径 由零或多个 '/' 符号隔开的字符串
query	参数 以键值对的形式，通过 & 符号分隔开来
fragment	片段 # 后面内容 常见于链接 锚点

5.2、location对象属性

location对象属性	返回值
location.href	获取或者设置整个URL
location.host	返回主机（域名）www.baidu.com
location.port	返回端口号，如果未写返回空字符串
location.pathname	返回路径
location.search	返回参数
location.hash	返回片段 #后面内容常见于链接 锚点

重点记住： `href` 和 `search`

需求：5s之后跳转页面

```
<body>
    <button>点击</button>
    <div></div>
    <script>
        var btn = document.querySelector('button');
        var div = document.querySelector('div');
        var timer = 5;
        setInterval(function() {
            if (timer == 0) {
                location.href = 'http://www.itcast.cn';
            } else {
                div.innerHTML = '您将在' + timer + '秒钟之后跳转到首页';
                timer--;
            }

        }, 1000);
    </script>
</body>
```

5.3、location对象方法

location对象方法	返回值
location.assign()	跟href一样，可以跳转页面（也称为重定向页面）记录历史页面，可以后退
location.replace()	替换当前页面，因为不记录历史，所以不能后退页面
location.reload()	重新加载页面，相当于刷新按钮或者 f5 ，如果参数为true 强制刷新 ctrl+f5

```
<body>
    <button>点击</button>
    <script>
        var btn = document.querySelector('button');
        btn.addEventListener('click', function() {
            // 记录浏览历史，所以可以实现后退功能
            // location.assign('http://www.itcast.cn');
            // 不记录浏览历史，所以不可以实现后退功能
            // location.replace('http://www.itcast.cn');
            location.reload(true);
        })
    </script>
</body>
```

5.4、获取URL参数

我们简单写一个登录框，点击登录跳转到 index.html

```
<body>
    <form action="index.html">
        用户名： <input type="text" name="uname">
        <input type="submit" value="登录">
    </form>
</body>
```

接下来我们写 index.html

？！

```
<body>
    <div></div>
    <script>
        console.log(location.search); // ?uname=andy
        // 1.先去掉?  substr('起始的位置', 截取几个字符);
        var params = location.search.substr(1); // uname=andy
        console.log(params);
        // 2. 利用=把字符串分割为数组 split('=');
        var arr = params.split('=');
        console.log(arr); // ["uname", "ANDY"]
        var div = document.querySelector('div');
        // 3.把数据写入div中
        div.innerHTML = arr[1] + '欢迎您';
    </script>
</body>
```

这样我们就能获取到路径上的URL参数

6、navigator对象

- navigator 对象包含有关浏览器的信息，它有很多属性
- 我们常用的是 `userAgent` ,该属性可以返回由客户机发送服务器的 `user-agent` 头部的值

下面前端代码可以判断用户是用哪个终端打开页面的，如果是用 PC 打开的，我们就跳转到 PC 端的页面，如果是用手机打开的，就跳转到手机端页面

```
if((navigator.userAgent.match(/(phone|pad|pod|iPhone|iPod|ios|iPad|Android|Mobile|BlackBerry|IEMobile|MQQBrowser|JUC|Fennec|wOSBrowser|BrowserNG|WebOS|Symbian|Windows Phone)/i))) {
    window.location.href = "";     //手机
 } else {
    window.location.href = "";     //电脑
 }
```

7、history对象

- window 对象给我们提供了一个 history 对象，与浏览器历史记录进行交互
- 该对象包含用户（在浏览器窗口中）访问过的 URL。

history对象方法	作用
back()	可以后退功能
forward()	前进功能
go(参数)	前进后退功能，参数如果是 1 前进1个页面 如果是 -1 后退1个页面

```
<body>
    <a href="list.html">点击我去往列表页</a>
    <button>前进</button>
    <script>
        var btn = document.querySelector('button');
        btn.addEventListener('click', function() {
            // history.forward();
            history.go(1);
        })

```

```
    </script>
</body>
```