云采FED系列教程

浏览器渲染以及JS EVENT LOOP

1.浏览器渲染过程

浏览器如何解析HTML

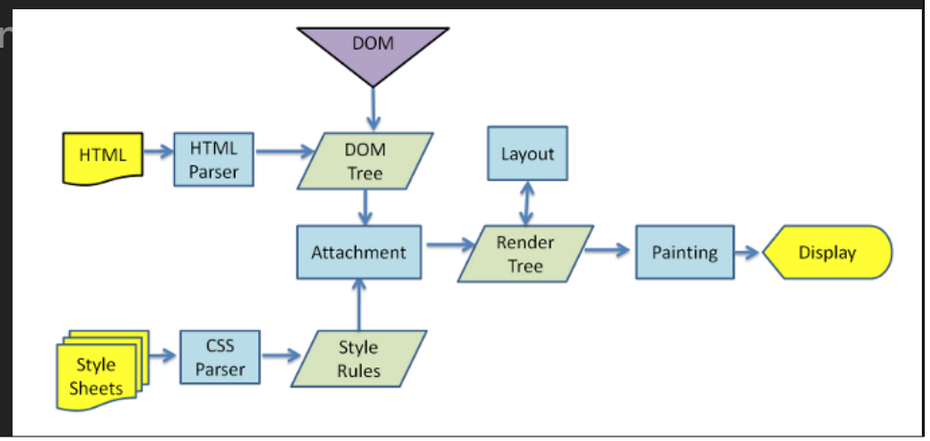
- 1. 浏览器接收到HTML模板文件，开始从上到下依次解析HTML；
- 2. 遇到样式表文件style1.css，这时候浏览器停止解析HTML，接着去请求CSS文件；
- 3. 服务端返回CSS文件，浏览器开始解析CSS；
- 4. 浏览器解析完CSS，继续往下解析HTML，碰到DOM节点，解析DOM；
- 5. 浏览器发现img，向服务器发出请求。此时浏览器不会等到图片下载完，而是继续渲染后面的代码；
- 6. 服务器返回图片文件，由于图片占用了一定面积，影响了页面排布，因此浏览器需要回过头来重新渲染这部分代码；
- 7. 碰到脚本文件，这时停止所有加载和解析，去请求脚本文件，并执行脚本；
- 8. 加载完所有的HTML、CSS、JS后，页面就出现在屏幕上了。

页面渲染过程

- 1.解析html 构建dom树
- 2.构建render树（渲染树，注：不包含display:none的部分）
- 3.布局render树 （layout过程：定位坐标和大小，是否换行，各种position, overflow, z-index属性 ）
- 4.绘制render树（背景->浮动部分->content->padding->border ....）

页面渲染过程

- 1.解析html 构建dom树
- 2.构建render树（渲染树，注：不包含display:none的部分）
- 3.布局render树 （layout过程：定位坐标和大小，是否换行，各种position
- 4.绘制render树 >border ....)



引起浏览器重新LAYOUT

JAVASCRIPT如果动态修改了DOM属性或是CSS属会导致重新LAYOUT

- repaint（重绘）
- reflow（重排）

重绘和重排

	repaint	reflow
触发方式	不改变定位，宽高，只改变元素展示方式	影响了文档内容、结构或者元素定位时
举例	background-color,border-color,visibility等	DOM 操作（如元素增、删、改或者改变元素顺序） 内容的改变，包括 Form 表单中文字的变化 计算或改变 CSS 属性 增加或删除一个样式表 浏览器窗口的操作（改变大小、滚动窗口） 激活伪类（如:hover状态）
重新布局	部分	绝大部分

重排优化

- 1.position 为 absolue 或 fixed 重排只涉及自己的子孙元素
- 2.读取一些element属性的时候，会重新计算并重排
- 3.其他优化不再举例，可以百度搜搜

参考文献：http://www.cnblogs.com/qqqiangqiang/p/5757656.html

重排优化

- 1.position 为 absolue 或 fixed 重排只涉及自己的子孙元素
- 2.读取一些element属性的时候，会重新计算并重排
- 3.其他优化不再举例，可以百度搜搜

参考文献：http://www.cnblogs.com/qqqiangqiang/p/5757656.html

1.offsetTop, offsetLeft, offsetWidth, offsetHeight
2.scrollTop/Left/Width/Height
3.clientTop/Left/Width/Height
4.IE中的 getComputedStyle(), 或 currentStyle

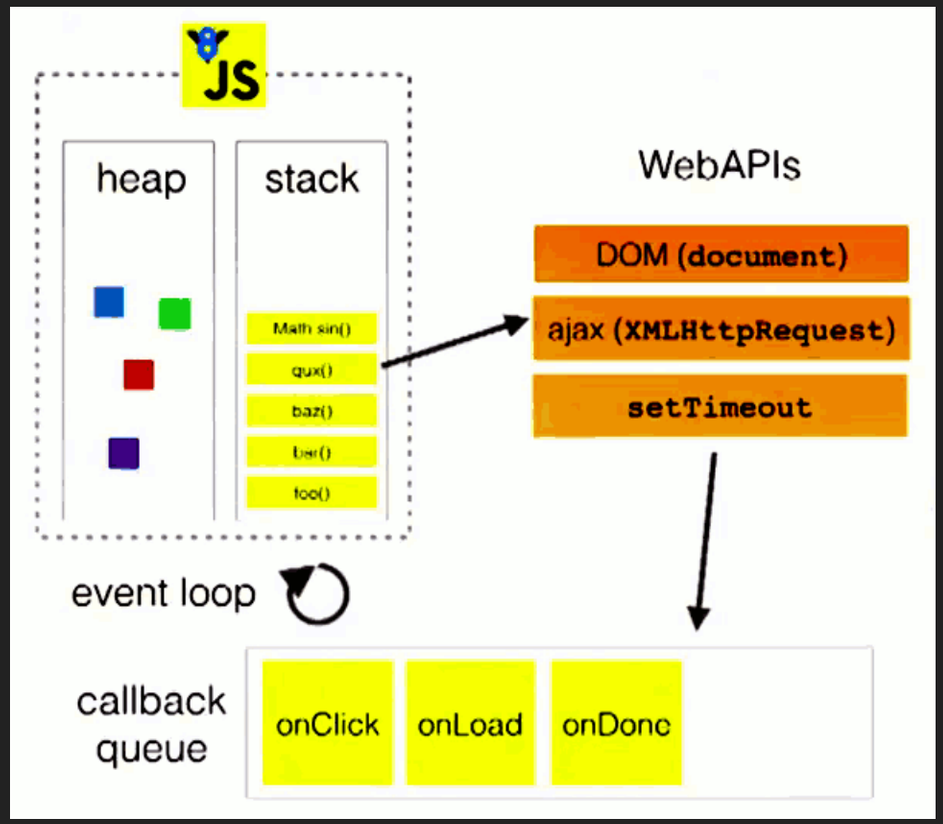
重排优化

- 1.position 为 absolue 或 fixed 重排只涉及自己的子孙元素
- 2.读取一些element属性的时候，会重新计算并重排
- 3.其他优化不再举例，可以百度搜搜

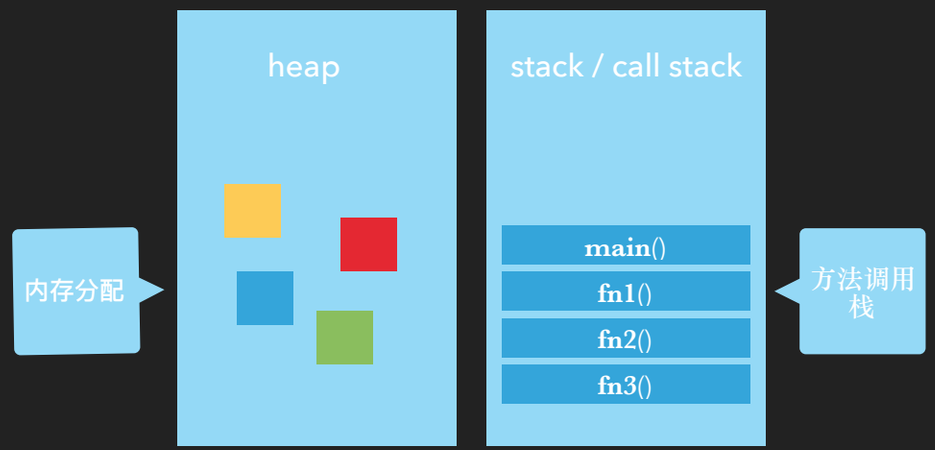
参考文献：http://www.cnblogs.com/qqqiangqiang/p/5757656.html

2.JS EVENT LOOP

JS运行环境的运行机制



JS引擎（如V8）



call stack

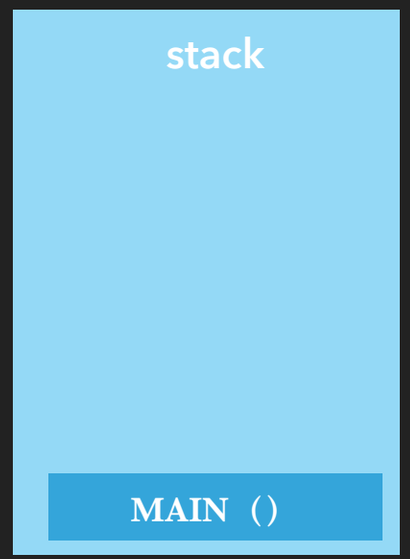
- js单线程
- 单一的call stack
- 一次只做一件事

- 注：浏览器是多进程、多线程。参考文献：https://www.cnblogs.com/hksac/p/6596105.html

文本

call stack

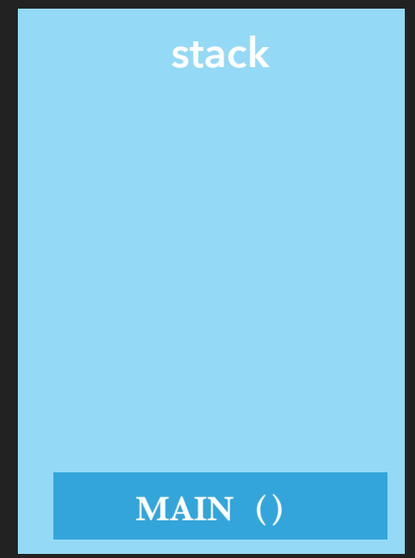
```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```



文本

call stack

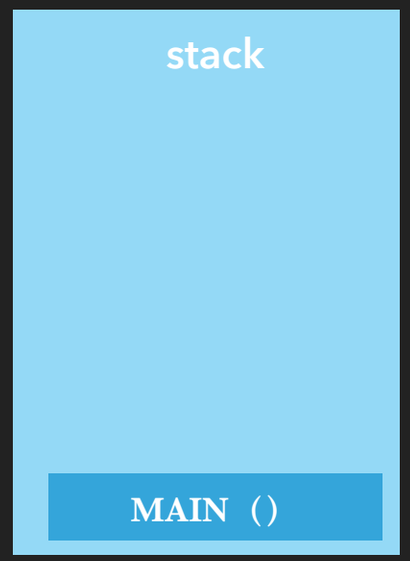
```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```



文本

call stack

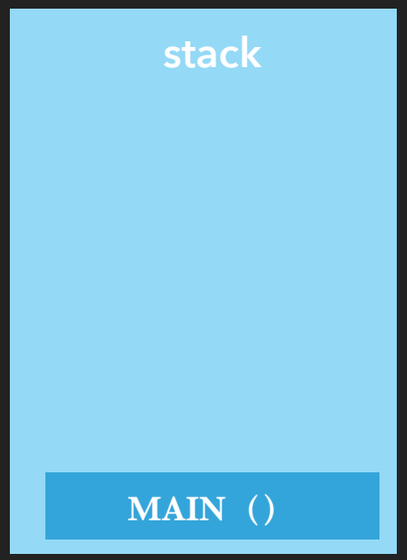
```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```



文本

call stack

```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```



文本

call stack

```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```

文本

call stack

```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```

文本

call stack

```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```

文本

call stack

```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```

文本

call stack

```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```

文本

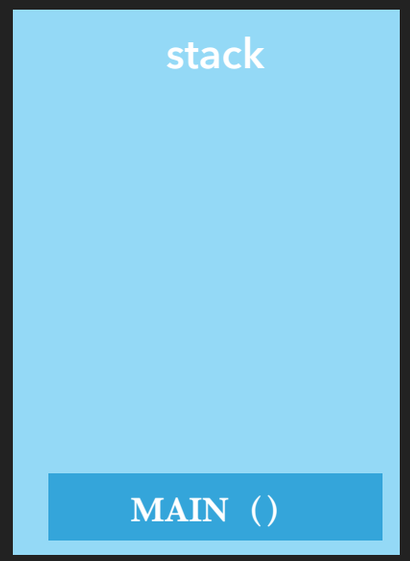
call stack

```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```

文本

call stack

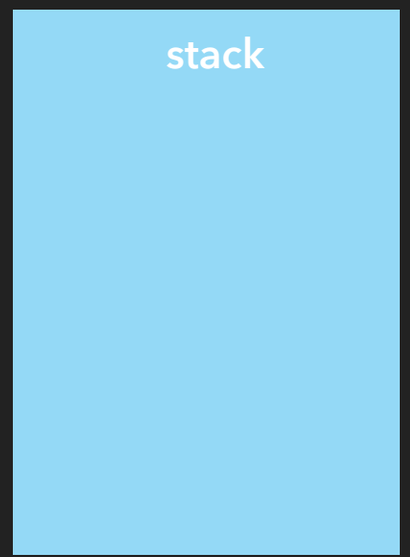
```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```



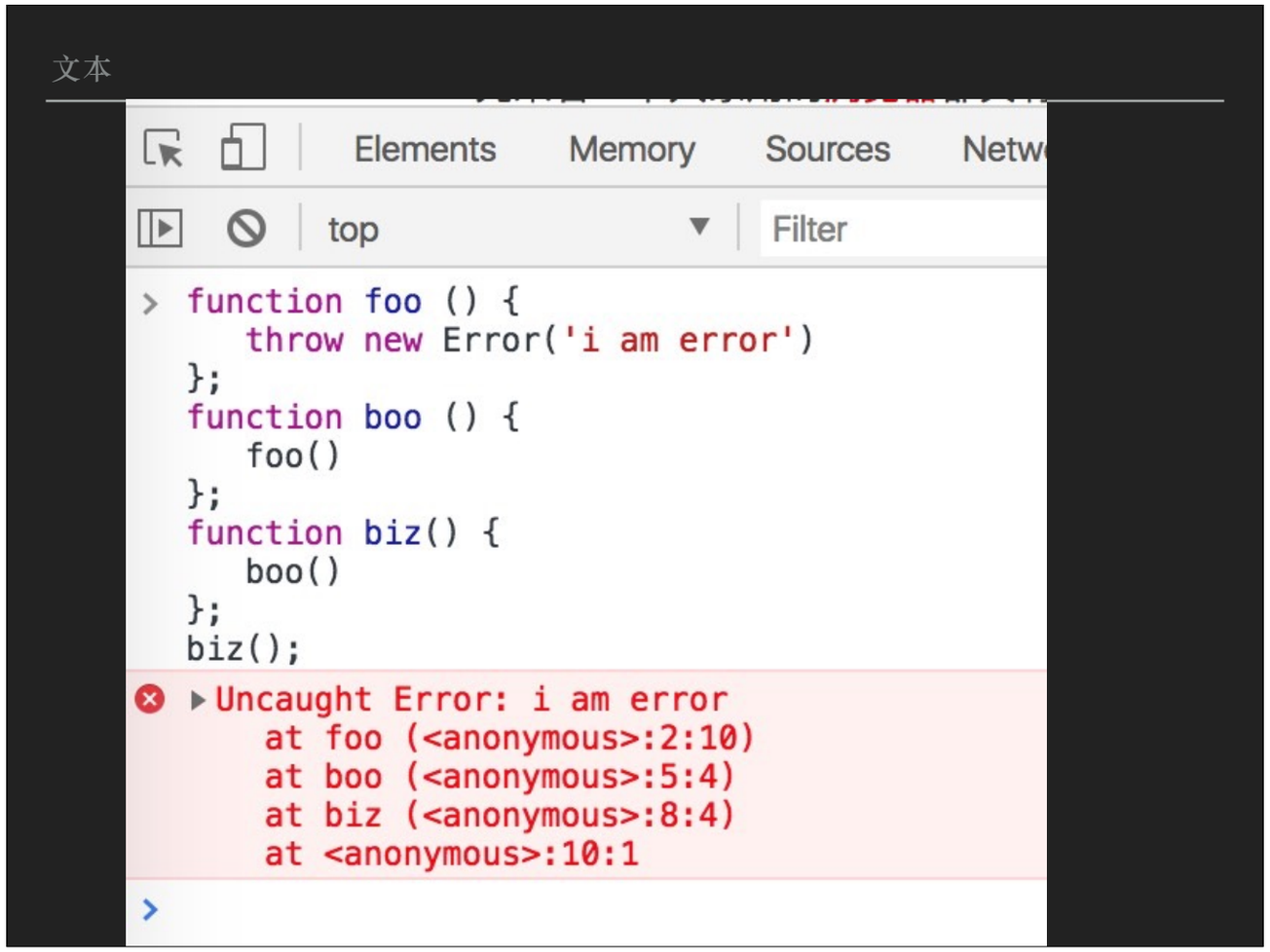
文本

call stack

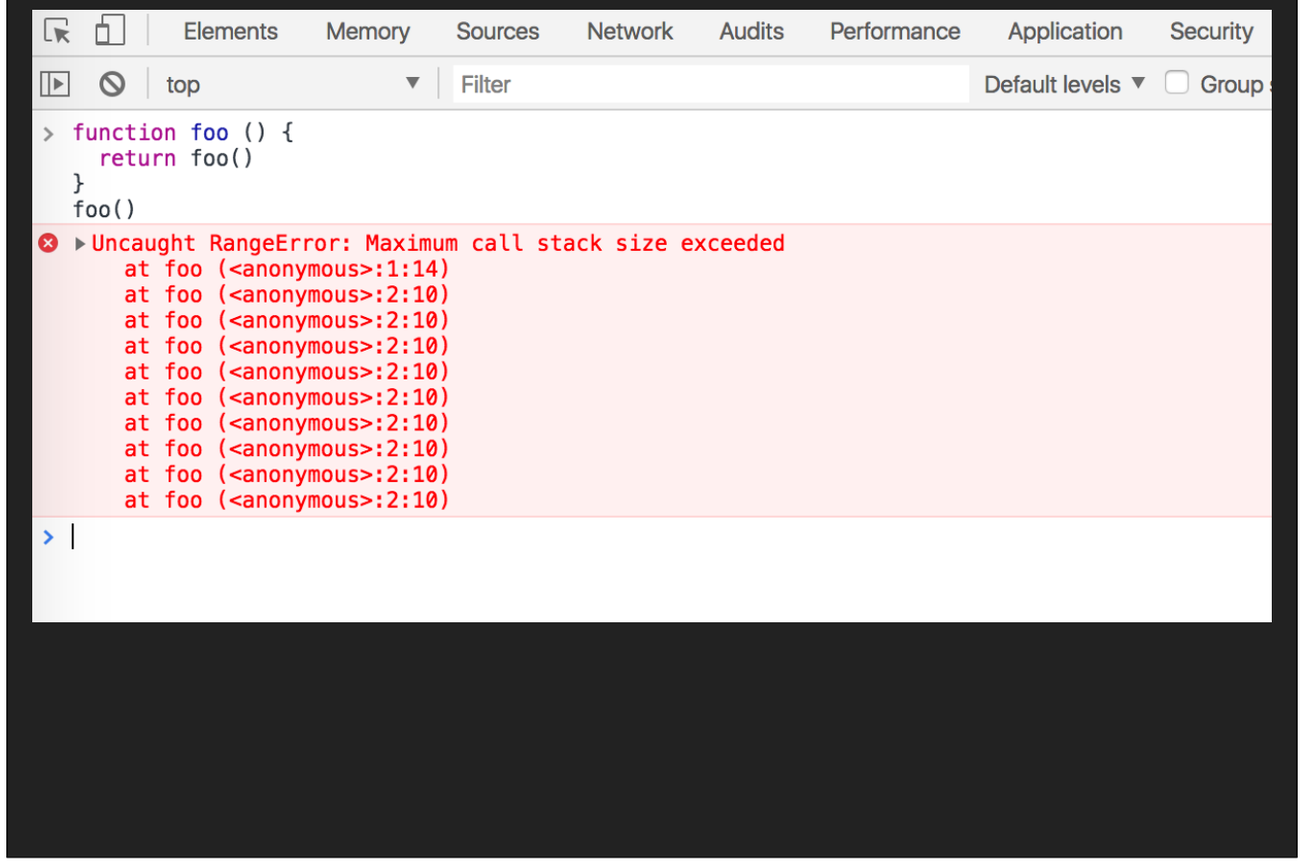
```
function  乘法 (a, b) {
   return a*b
}

function  平方 (a) {
   return 乘法(a, a)
}

function 打印乘方(n) {
  var square = 平方(n)
  console.log(square)
}
打印乘方(4)
```



文本


Elements
Memory
Sources
top
Filter
function foo () {
throw new Error('i am error')
};
function boo () {
foo()
};
function biz() {
boo()
};
biz();
Uncaught Error: i am error
at foo (<anonymous>:2:10)
at boo (<anonymous>:5:4)
at biz (<anonymous>:8:4)
at <anonymous>:10:1

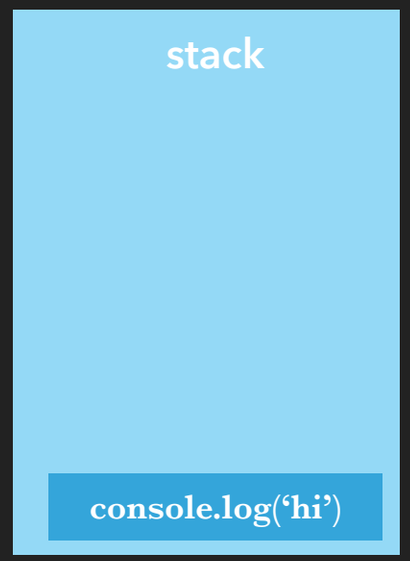
文本


Elements
Memory
Sources
Network
Audits
Performance
Application
Security
top
Filter
Default levels
Group
function foo () {
return foo()
}
foo()
Uncaught RangeError: Maximum call stack size exceeded
at foo (<anonymous>:1:14)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)
at foo (<anonymous>:2:10)

文本

异步

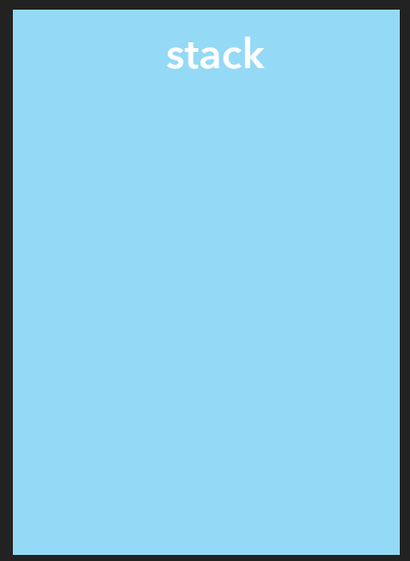
```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



文本

异步

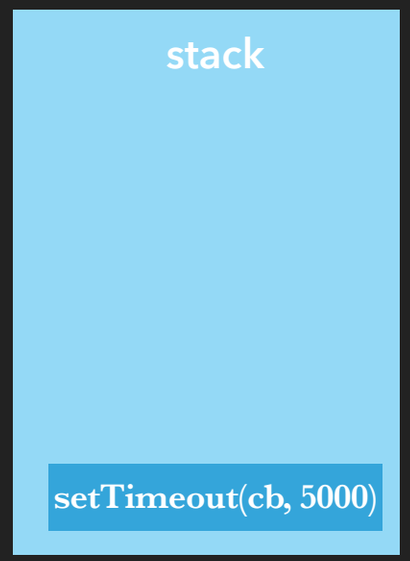
```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



文本

异步

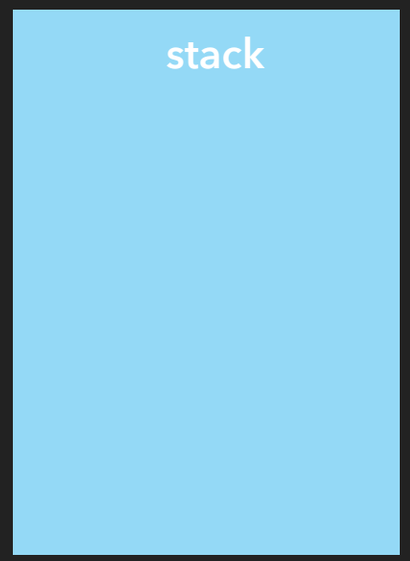
```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



文本

异步

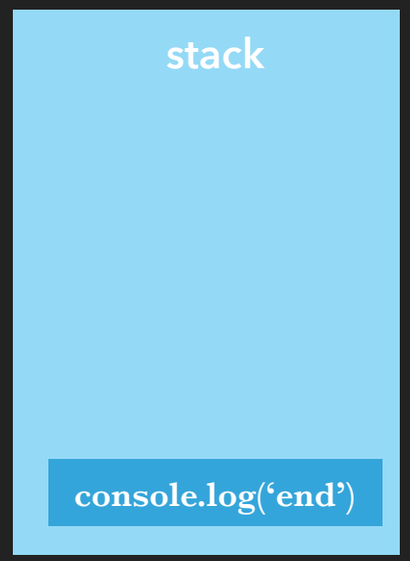
```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



文本

异步

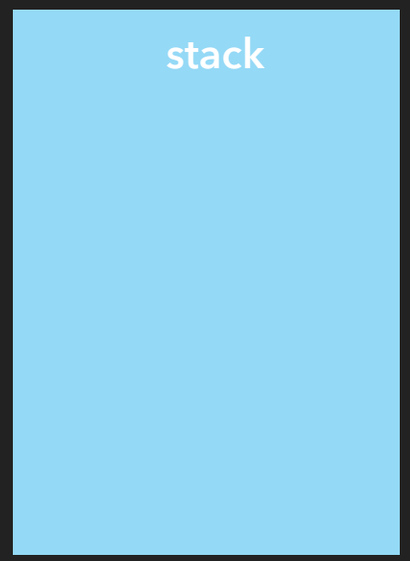
```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



文本

异步

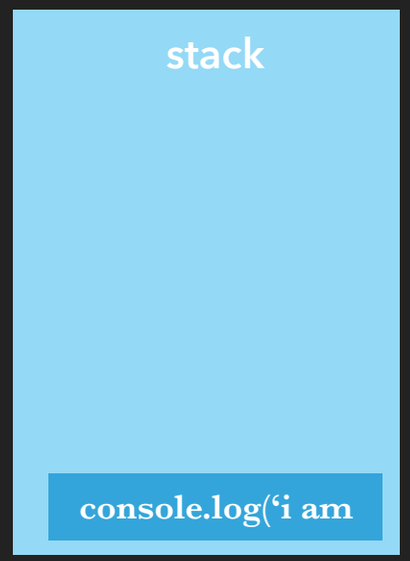
```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



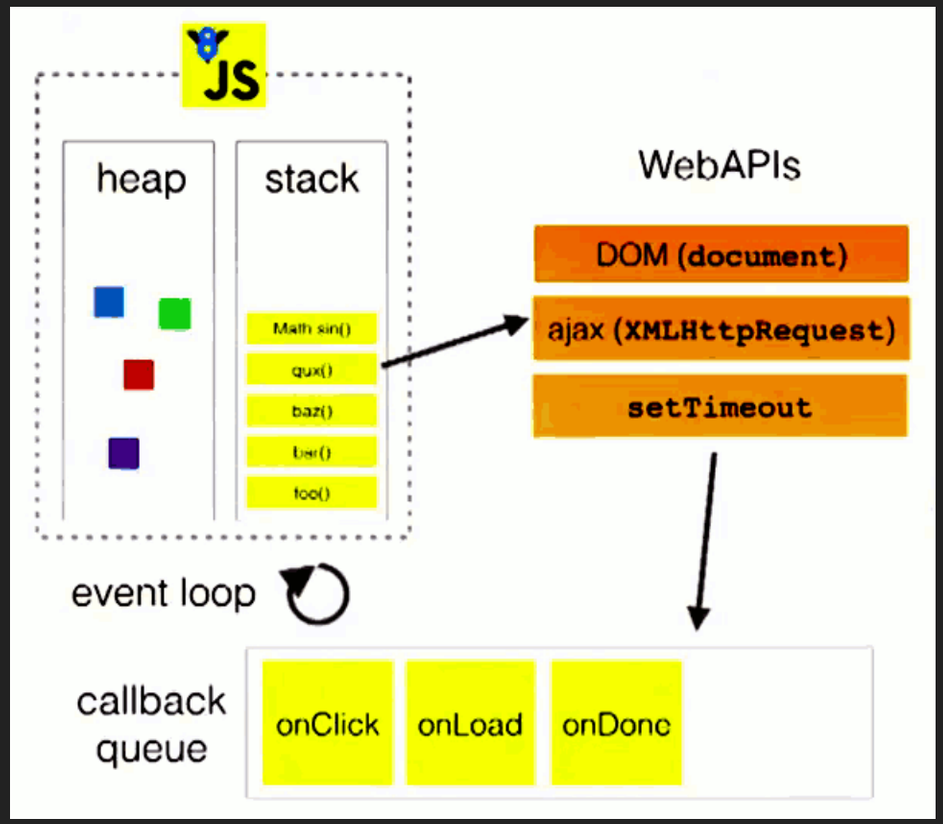
文本

异步

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



JS运行环境的运行机制

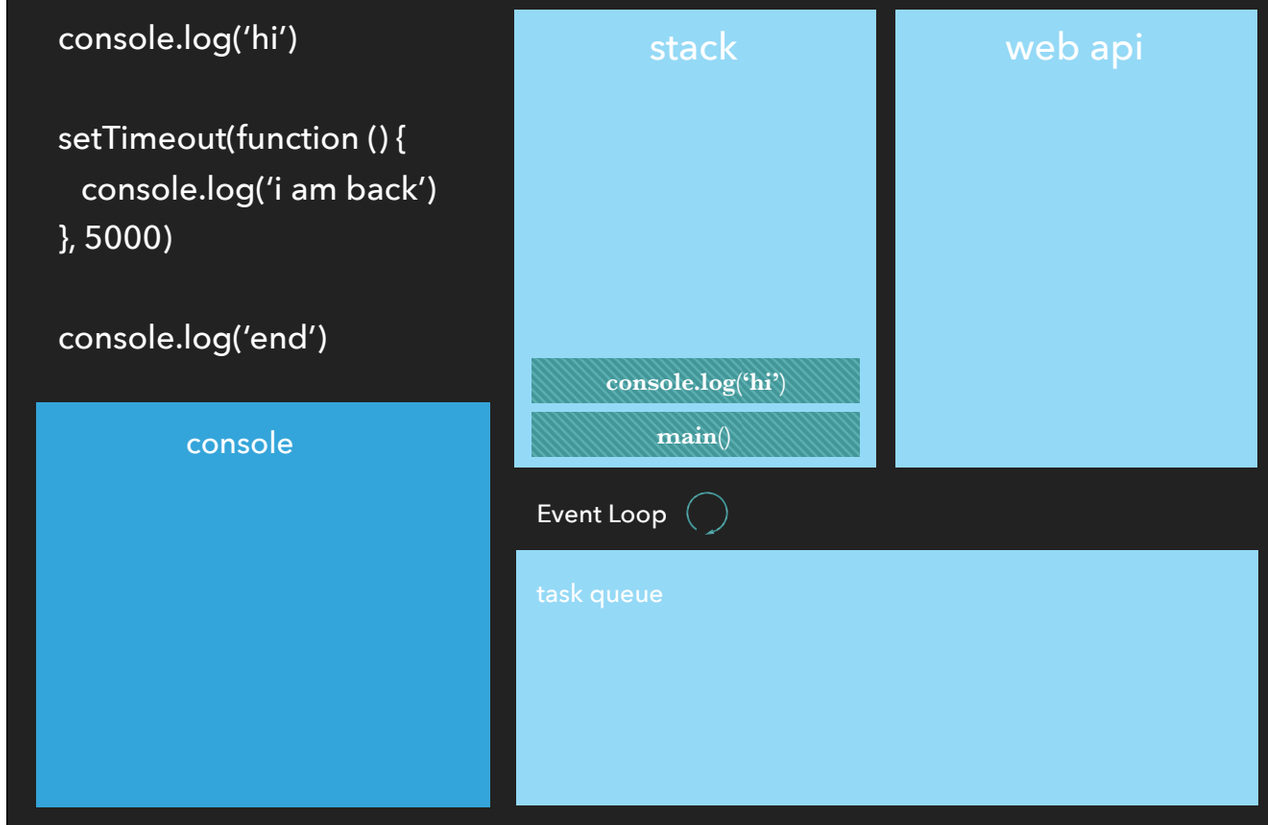


文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```

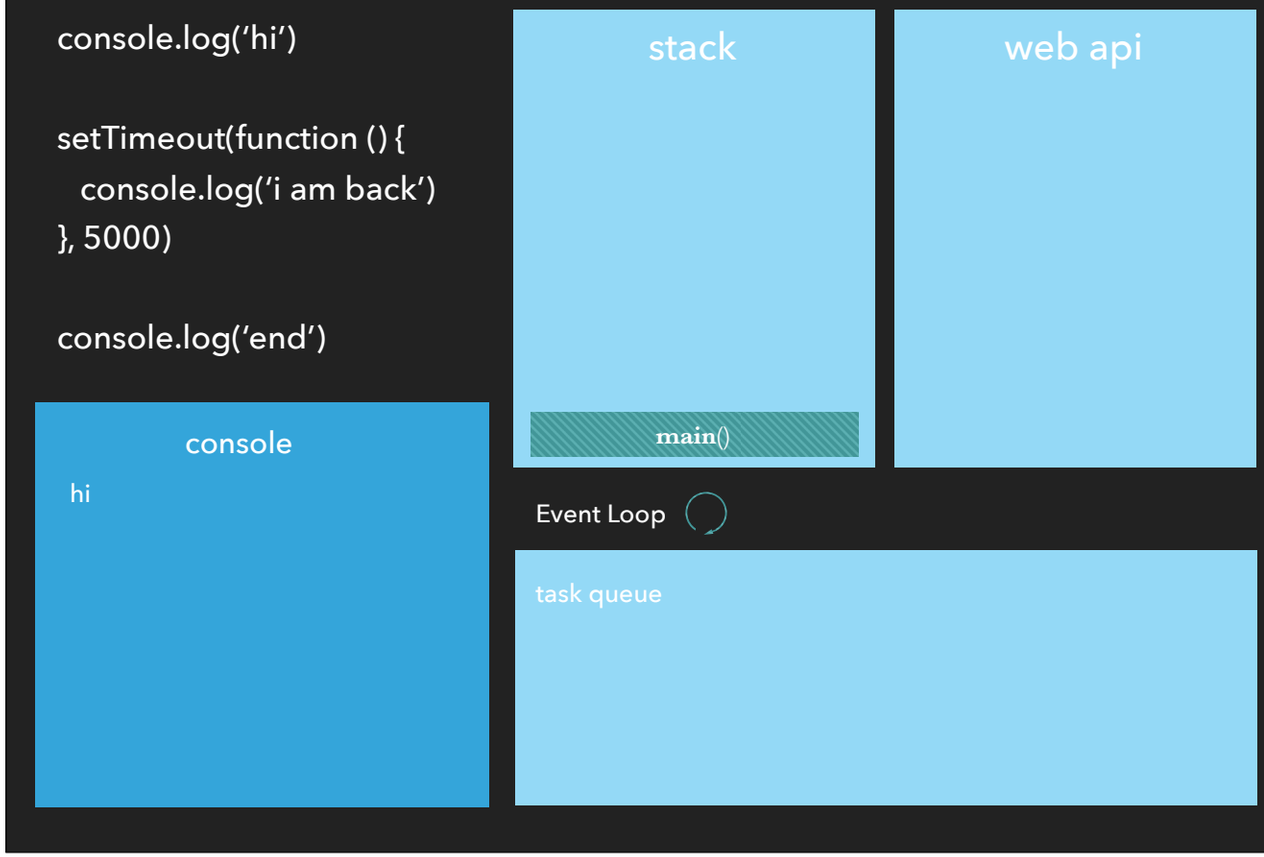
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



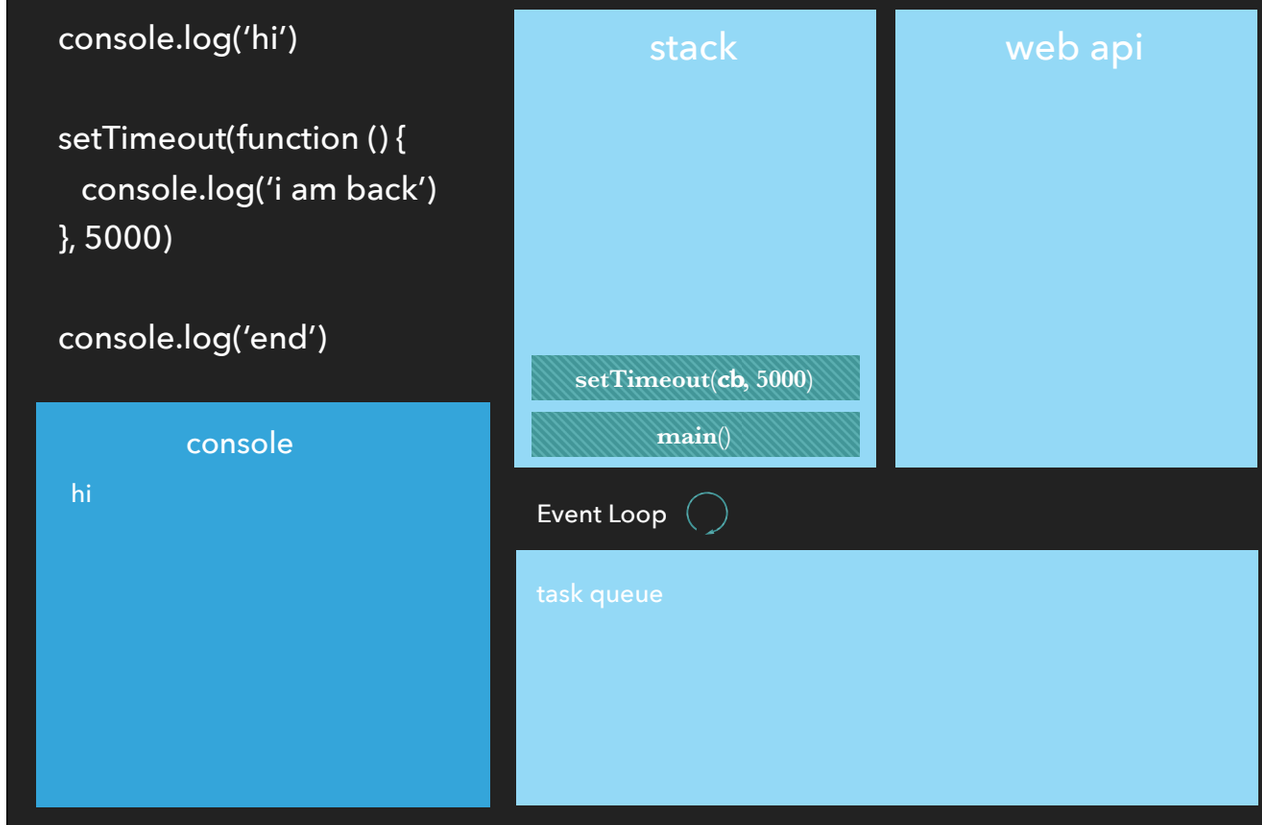
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



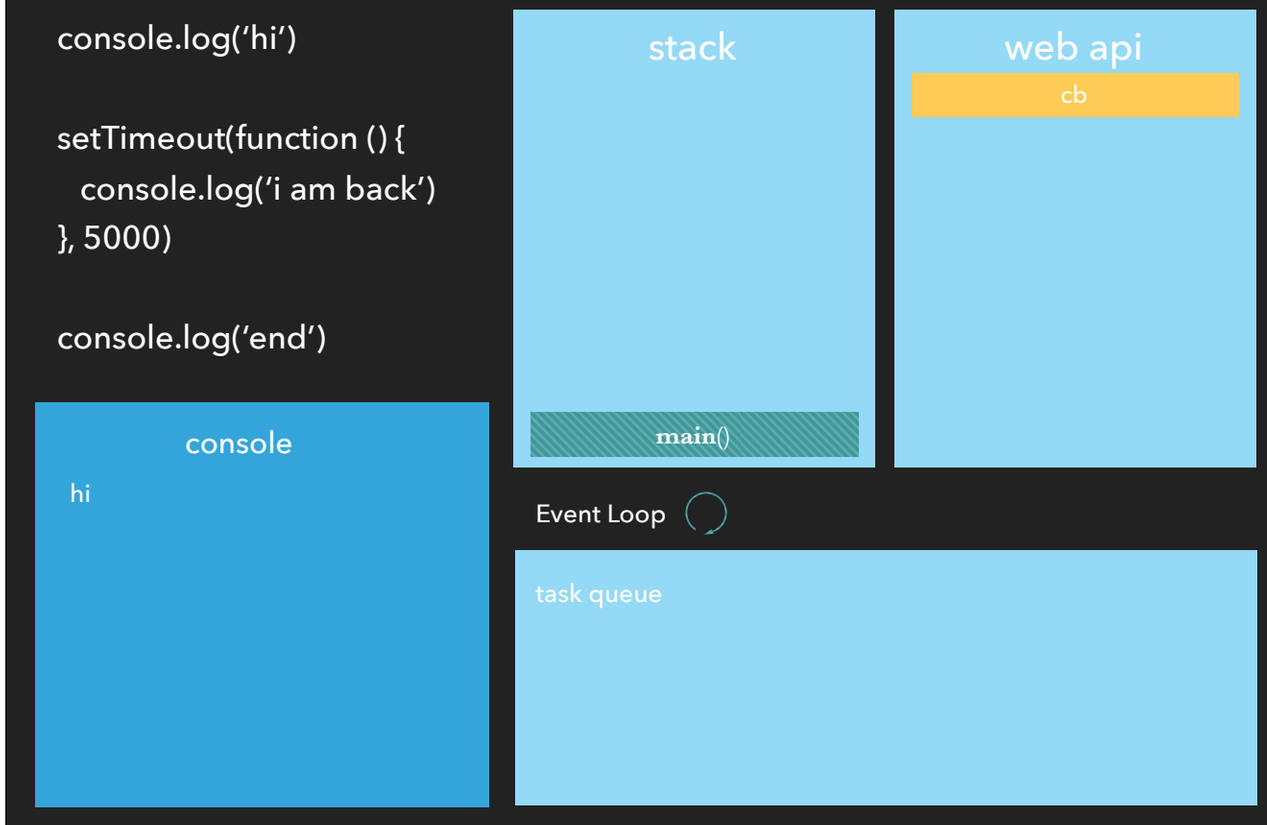
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



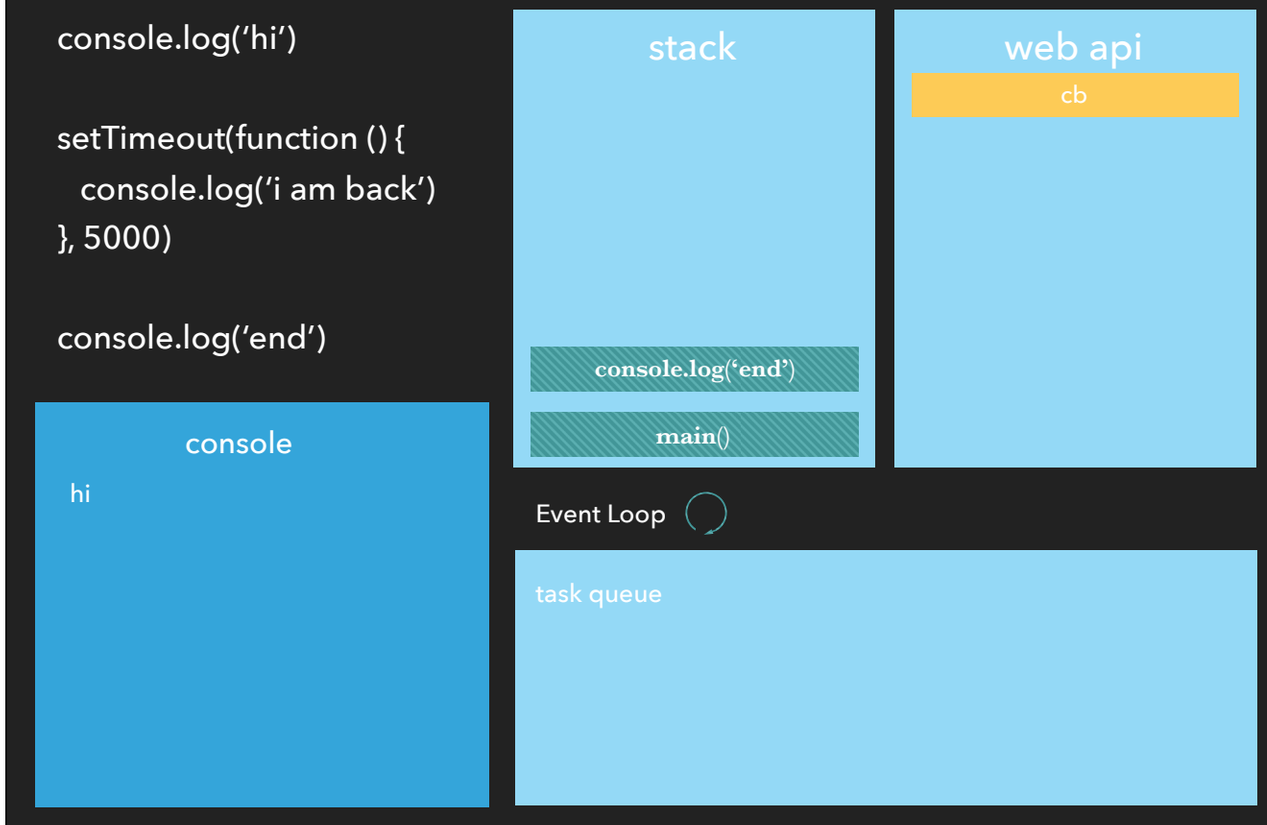
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



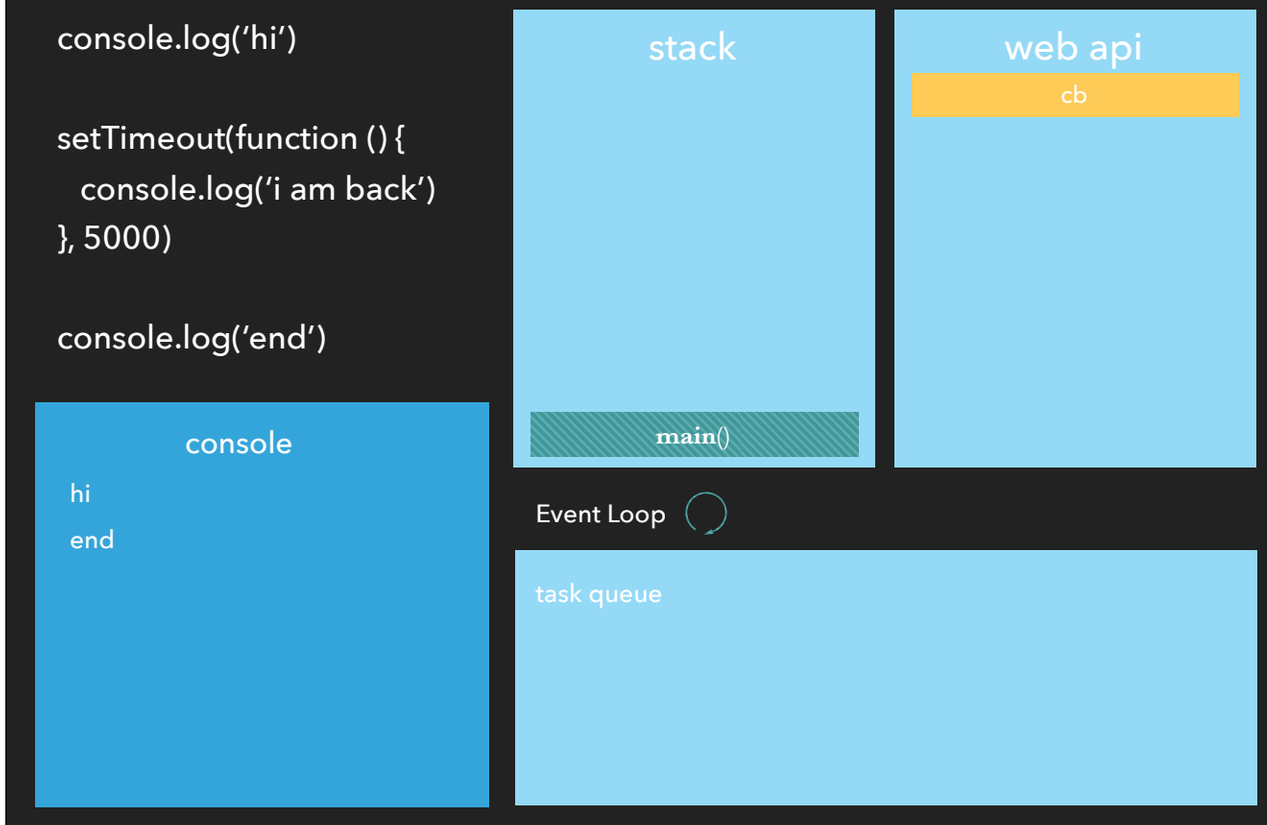
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



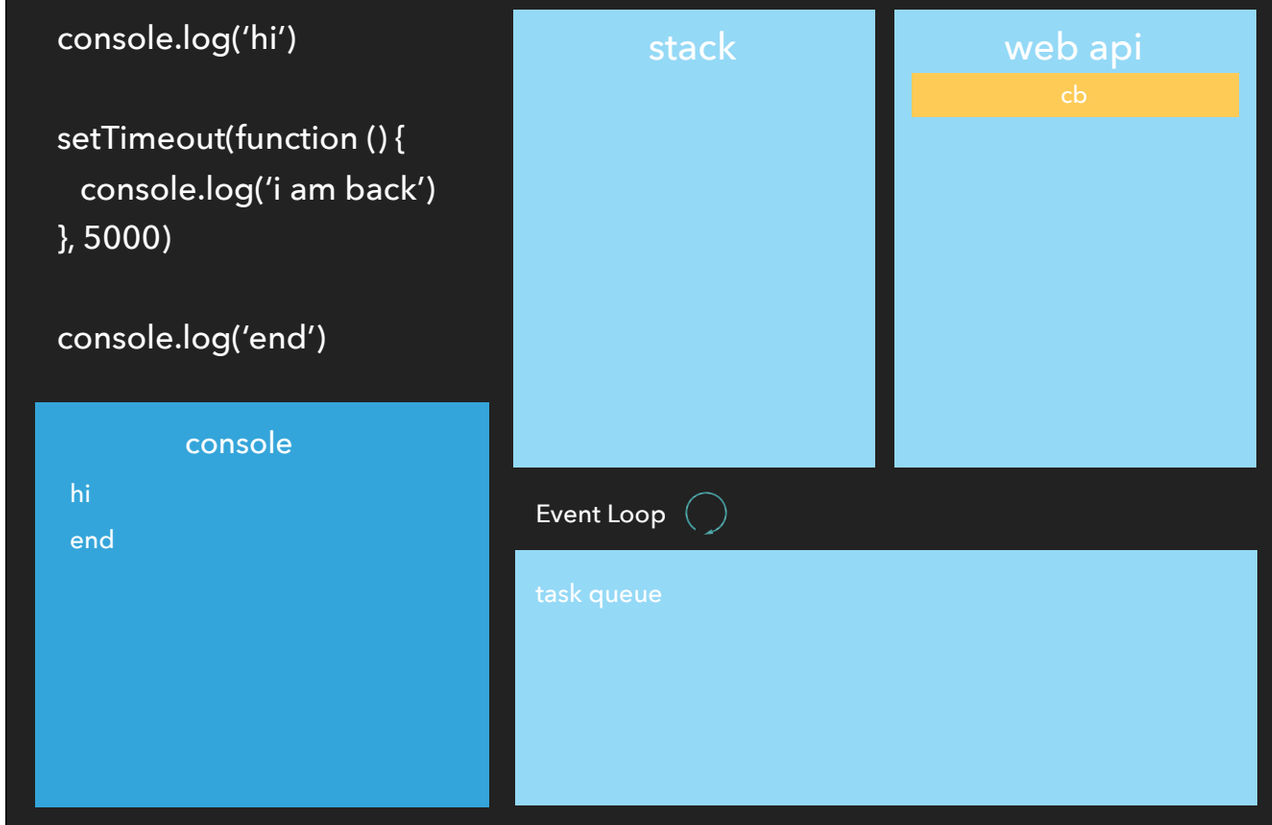
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



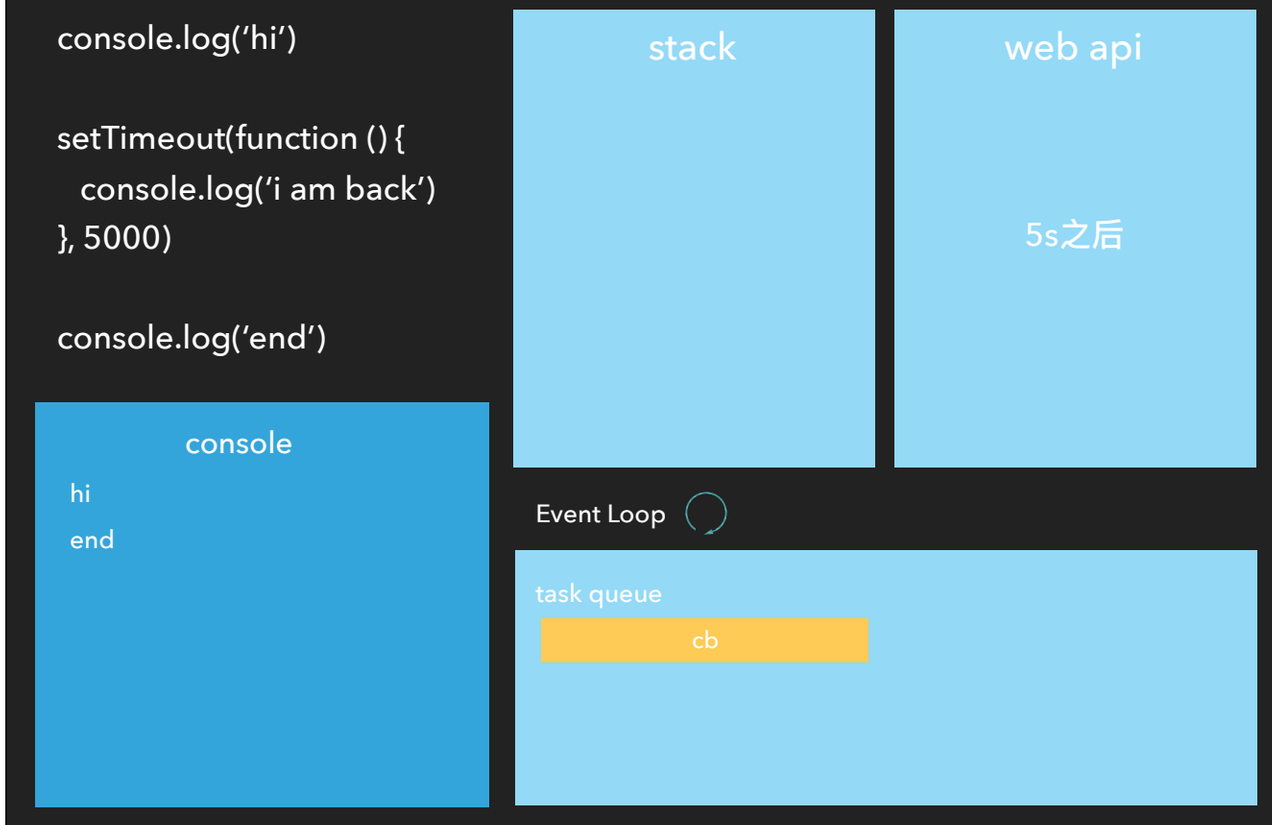
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



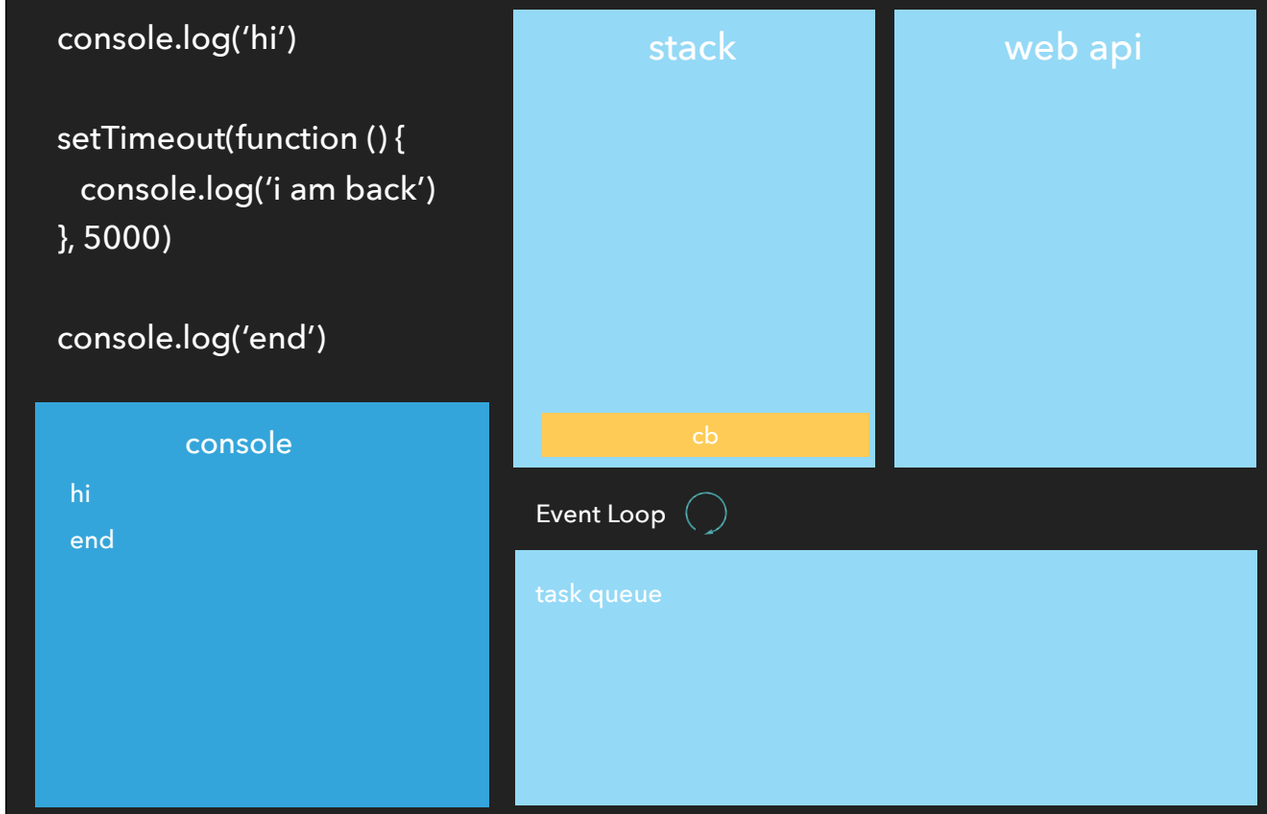
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



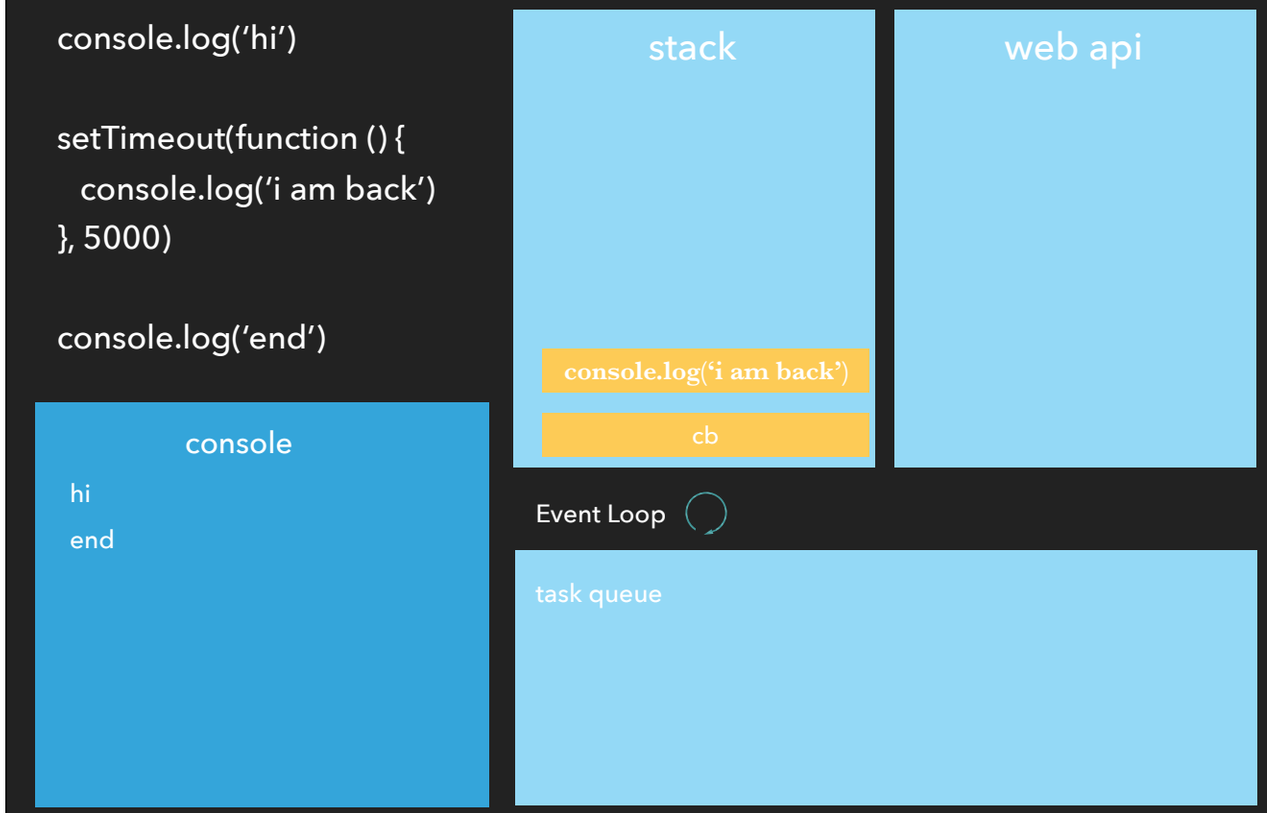
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



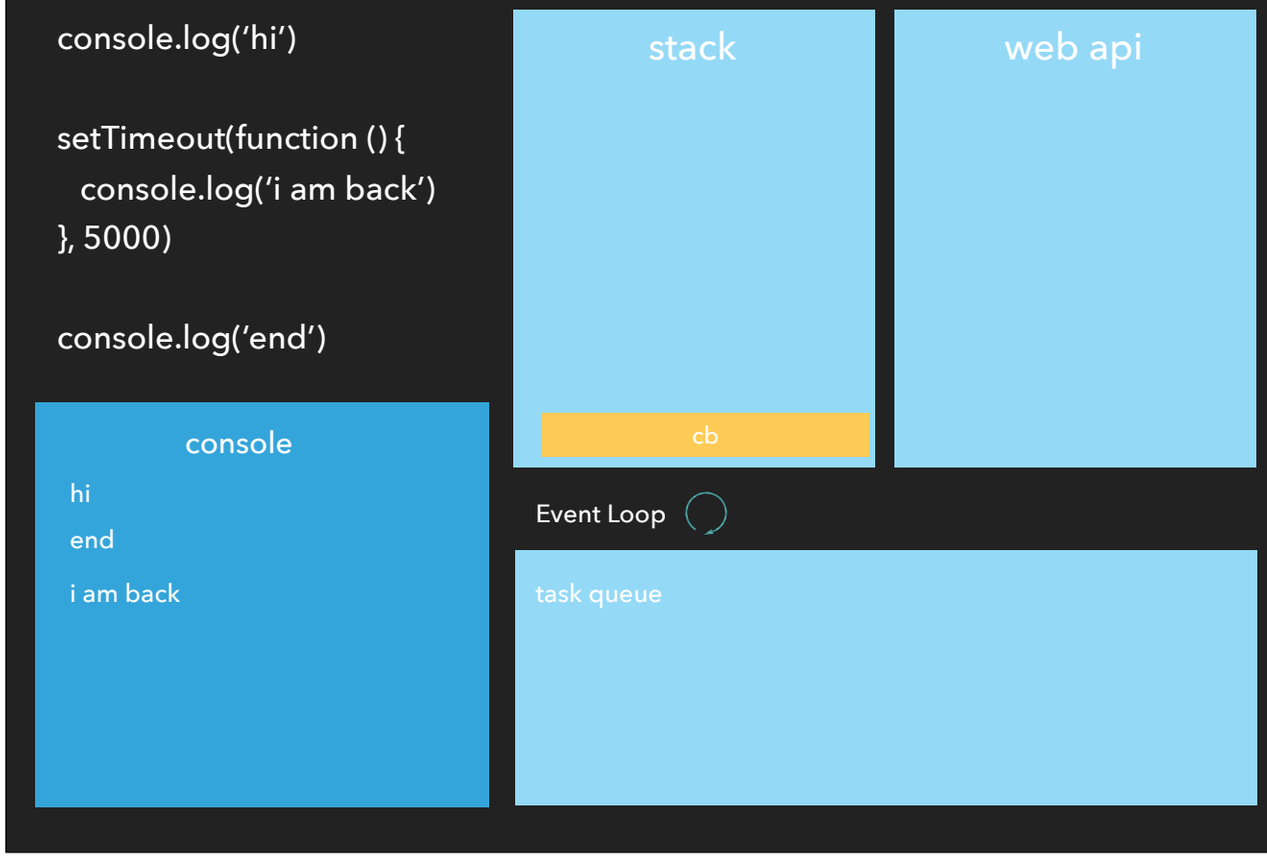
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



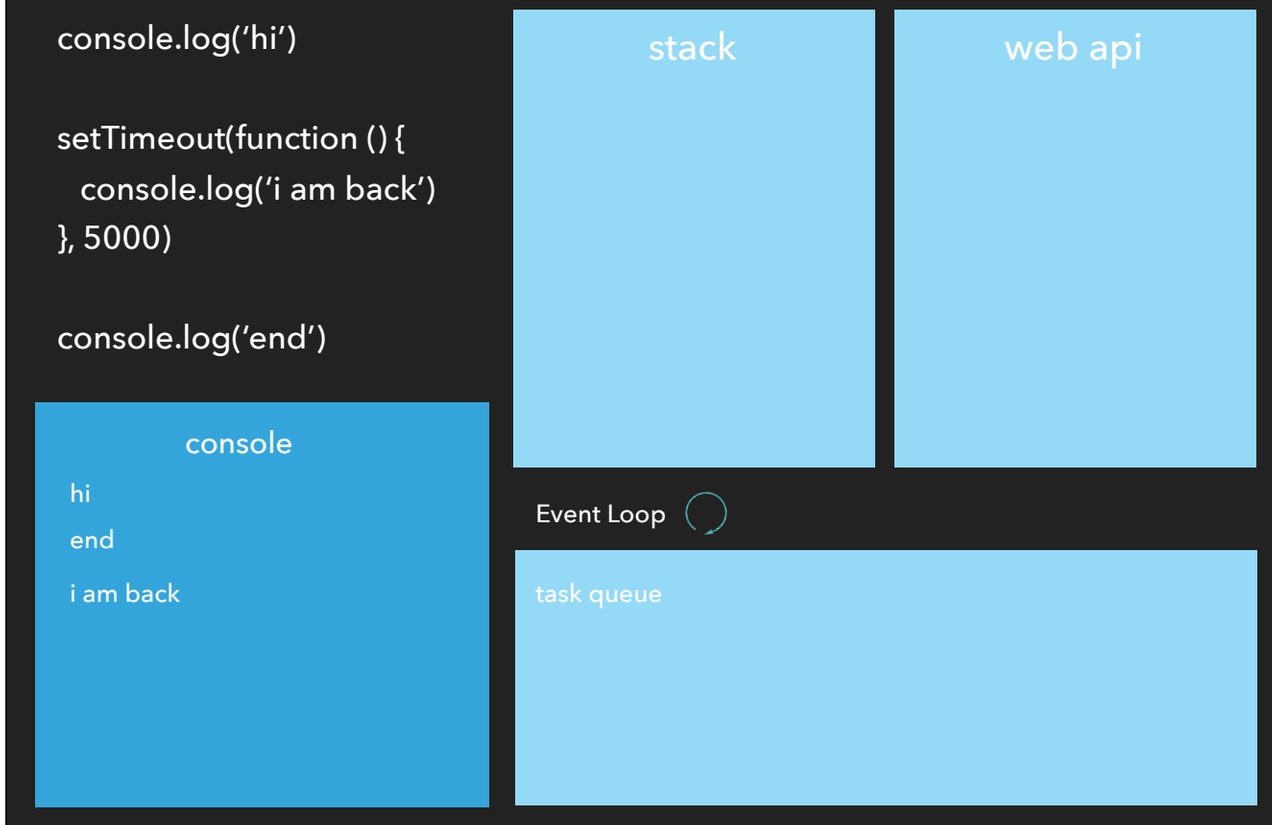
文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```



文本

```
console.log('hi')

setTimeout(function () {
  console.log('i am back')
}, 5000)

console.log('end')
```

console

hi

end

i am back

stack

web api

Event Loop

task queue

当且仅当stack为空时，才会执行task queue里的代码

文本

macrotask 和microtask

- macrotask
  - setTimeout
  - setInterval
  - setImmediate
  - I/O（键盘、网络）
  - UI rendering
- microtask
  - process.nextTick
  - promise
  - Object.observe
  - MutationObserver

EVENT LOOP的循环过程：

- 1.在tasks队列中选择最老的一个task,用户代理可以选择任何task队列，如果没有可选的任务，则跳到下边的microtasks步骤。
- 2.将上边选择的task设置为正在运行的task。
- 3.Run: 运行被选择的task。
- 4.将event loop的currently running task变为null。
- 5.从task队列里移除前边运行的task。
- 6.Microtasks: 执行microtasks任务检查点。（也就是执行microtasks队列里的任务）
- 7.更新渲染（Update the rendering）...
- 8.返回到第一步。

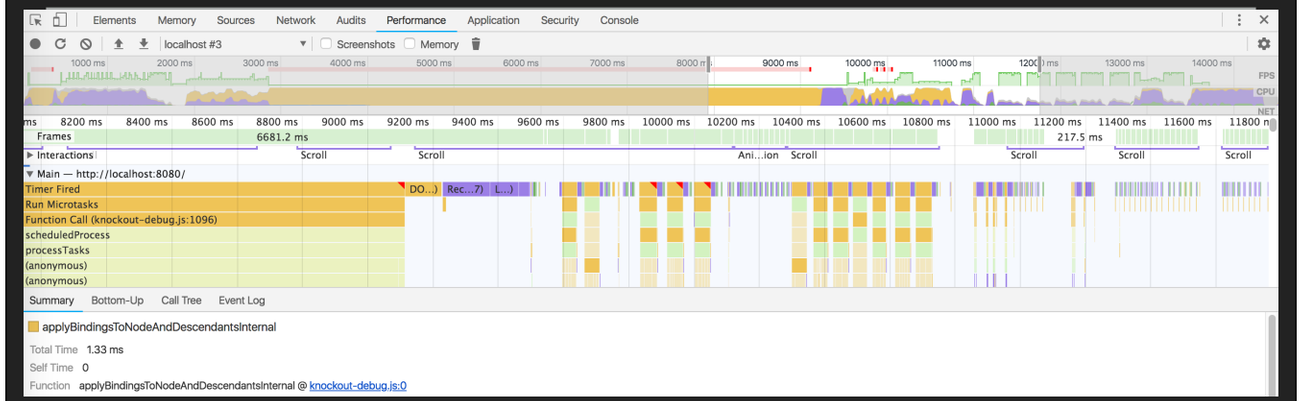
文本

概况

event loop会不断循环的去取tasks队列的中最老的一个任务推入栈中执行，并在当次循环里依次执行并清空microtask队列里的任务。

执行完microtask队列里的任务，有可能会渲染更新。（浏览器很聪明，在一帧以内的多次dom变动浏览器不会立即响应，而是会积攒变动以最高60HZ的频率更新视图）

文本



- 黄色部分是脚本运行，紫色部分是更新render树、计算布局，深绿色部分是绘制。

文本

看具体例子

- 参考文章：
- https://v.youku.com/v_show/id_XODA0MDYyNTcy.html
- https://juejin.im/entry/59082301a22b9d0065f1a186
- https://www.jianshu.com/p/1ee6c21f6efa