## 1事件驱动

所谓的**事件驱动**，就是将**一切抽象为事件**。

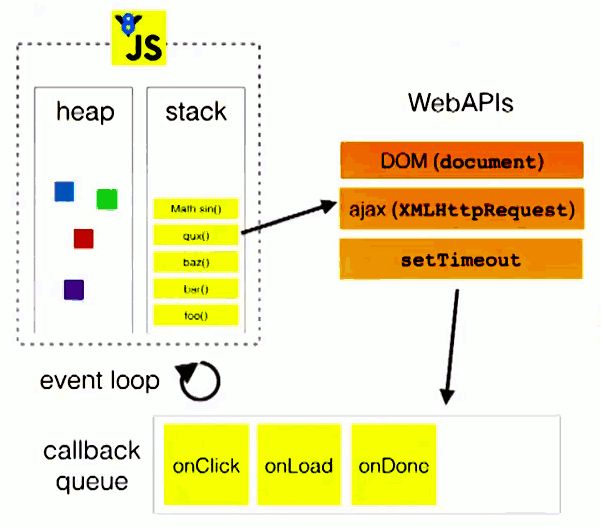
IO操作完成是一个事件，用户的鼠标点击是事件，Ajax 完成了是一个事件，一个图片加载完成是一个事件。

“任务队列”中的事件，除了IO设备的事件以外，还包括一些用户产生的事件（比如鼠标点击、页面滚动等等）。只要指定过回调函数，这些事件发生时就会进入”任务队列”，等待主线程读取。

**事件驱动**的实现过程主要靠事件循环完成。进程启动后就进入主循环。主循环的过程就是不停的从事件队列里读取事件。如果事件有关联的handle(也就是注册的callback)，就执行handle。一个事件并不一定有callback。

如下图，主线程运行的时候，产生堆（heap）和栈（stack），栈中的代码调用各种外部API，它们在”任务队列”中加入各种事件（click，load，done）。只要栈中的代码执行完毕，主线程就会去读取”任务队列”，依次执行那些事件所对应的回调函数。

执行栈中的代码，总是在读取”任务队列”之前执行。



## 2事件循环 Event Loop

JS 是单线程的，同一时间，只能做同一件事，为了协调事件、用户交互、脚本、UI渲染和网络处理等行为，防止主线程阻塞，Event Loop方案应运而生，事件循环是JS实现异步的方式，也是JS的执行机制。

## 3任务队列 Task Queue

事物循环通过任务队列的机制进行协调，一个任务列表（task queue) 包含多个任务（task)，每个任务都有一个任务源（task source)，源自同一个任务源的 task 必须放到同一个任务队列，遵循先进先出的原则。

task 任务源举例：

* DOM操作任务源：用来操作相应的DOM，例如一个元素以非阻塞的方式插入文档
* 用户交互任务源：对用户的交互做出反应，例如键盘鼠标的输入，响应用户操作的事件（如click）
* 网络任务源：用来响应网络活动
* History traversal任务源：当调用 history.back()等类似API时，将任务插进队列

task 任务源非常宽泛，比如 ajax 的 onload，click 事件，基本上我们经常绑定的各种事件都是task任务源，还有数据库操作（IndexedDB ），需要注意的是setTimeout、setInterval、setImmediate也是task任务源。

## 4JS的异步运行机制

1. 所有任务都在主线程上执行，形成一个执行栈（execution context stack）。
2. 主线程之外，还存在一个”任务队列”（task queue）。系统把**异步任务**放到**”任务队列”**之中，然后继续执行后续的任务。
3. 一旦”执行栈”中的所有任务执行完毕，系统就会读取”任务队列”。如果这个时候，异步任务已经结束了等待状态，就会从”任务队列”进入执行栈，恢复执行。
4. 主线程不断重复上面的第三步。

## 5Tick

事件循环中，每一次循环称为 tick, 每一次tick的任务如下：

选择最先进入队列的任务 task，如果有则执行

检查是否存在 Microtask，如果存在则不停的执行，直至清空 microtask 队列

更新render

重复以上步骤：宏任务 -> 所有微任务 -> 宏任务

## 6宏任务 macrotask 与 微任务 microtask

异步任务分为 宏任务task（macrotask） 与 微任务 microtask，不同的API注册的任务会依次进入自身对应的队列中，然后等待 Event Loop 将它们依次压入执行栈中执行

task 宏任务: script代码执行、setTimeout、setInterval、I/O、UI交互事件、postMessage、requestAnimationFrame、MessageChannel、setImmediate(Node.js环境）；

microtask 微任务: Promise.then、MutaionObserver、process.nextTick(Node.js环境）。