**1.关于javaacript**

javascript是一门单线程语言，所以javascript是按语句的执行顺序执行的。

虽然js是单线程，但是我们可以将任务分成两类

1.同步任务：需要执行的任务在主线程上排队，一次执行

2.异步任务：没有立马执行但是需要被执行的任务，放在 任务队列里面，

**2.javascript事件循环**

当我们打开网站的时候，网页的渲染其实是一堆同步任务，不如页面骨架和页面元素的渲染，但是想图片音乐等占用资源大耗时久的任务就是异步任务，

异步执行：

1.所有同步任务偶在主线程上执行，形成一个很执行栈

2.主线程之外，还存在一个任务队列（task queue）只要异步任务有了运行结果，就在“任务队列”之中放置一个事件。

3.一旦“执行栈”中的所有同步任务执行完毕，系统就会读取“任务队列”，看看里面有哪些事件。那些对应的异步任务，就结束等待状态，进入执行栈开始被执行。

4.主线程不断重复以上三步。

**怎么知道主进程为空？**

**js引擎存在monitoring process进程，会持续不断的检查主线程执行栈是否为空，一旦为空，就会去Event Queue那里检查是否有等待被调用的函数。**

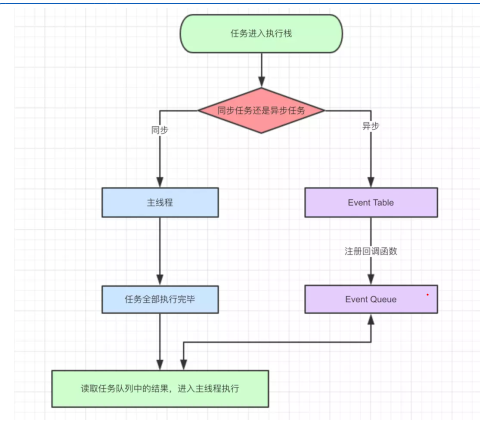
![N]M]\_N{OL(HBZ`B89REKMQ9.png](<https://upload-images.jianshu.io/upload_images/9374643-598ff6c029970ff9.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

同步和异步分别进入到不同的执行场所，同步进入到主线程，异步的进入到event table 并注册函数

当指定事件完成之后，event table 会将函数移入到event queue

当主线程的任务执行完毕之后，会把event queue里面读取对应的函数进入主线程执行

上述过程会不断循环，形成event loop(事件循环)



**setTimeout(() => {
task()
},3000)
sleep(10000000)**

task()进入到event table里面注册计时

然后主线程执行sleep函数，但是非常慢。计时任然在继续

3秒到了。task()进入event queue 但是主线程依旧没有走完

终于过了10000000ms之后主线程走完了，task()进入到主线程

所以可以看出其真实的时间是远远大于3秒的

**4.promise 和 process.nextTick(callback)**

process.nextTick(callback)类似node.js版的"setTimeout"，在事件循环的下一次循环中调用 callback 回调函数。

除了广义的同步任务和异步任务，我们可以分的更加精细一点：

macro-task(宏任务)：包括整体代码script，setTimeout，setInterval

micro-task(微任务)：Promise，process.nextTick

注：每执行一个宏观任务，就执行一次它下面对应的微观任务

不同的任务会进入到不同的event queue。比如setTimeout和setInterval会进入相同的Event Queue。

实例：console.log('1'); setTimeout(function() { console.log('2'); process.nextTick(function() { console.log('3'); }) new Promise(function(resolve) { console.log('4'); resolve(); }).then(function() { console.log('5') }) }) process.nextTick(function() { console.log('6'); }) new Promise(function(resolve) { console.log('7'); resolve(); }).then(function() { console.log('8') }) setTimeout(function() {console.log('9'); process.nextTick(function() { console.log('10'); }) new Promise(function(resolve) { console.log('11'); resolve(); }).then(function() { console.log('12') }) }) // 1,7,6,8,2,4,3,5,9,11,10,12

分析情况：

首先先执行console.log(1) 然后将setTimeout放到宏任务event queue里面 记作 setTimeout 1 ，接着 看到 process.nextTick ，将其放到微任务里面 ，记作 process 1，然后 看到new promise 立即执行输出9 ，将里面的then 放到 微任务里面 记作 then 2, 继续，遇到 setTimeout 放到宏任务里面记作 setTimeout 2 。**目前输出的是：1,7，**

OK, 接下来，开始判断是否有微任务，刚刚放入到微任务event queue的进入到主程序开始执行，process 1 ， then 2 **目前输出的是：6,8**、

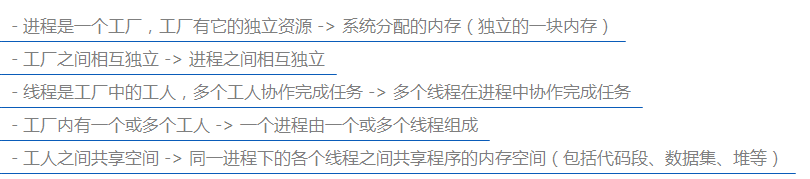
接下来，微任务的event queue 空了，进行下一轮事件，将刚刚放到宏任务的 setTimeout 1 进入到主线程

遇到 console 立即执行， 遇到 process.nextTick 放到微任务 event queue 里面 记作 process1， 接着遇到 new Promise 立即执行， 将 then 放到event queue 里面 记作 then 2，OK，当前宏任务里的任务执行完了，判断是否有微任务，发现有 process1， then 2 两个微任务 ， 一次执行 **目前输出的是：2,4,3,5**、

目前主线程里的任务都执行结束了,又开始第三轮事件循环，同上（字太多，省略。。。。） **目前输出的是：9,11,10,12**、

**注意: 以上所说只能是在浏览器中的执行顺序，**

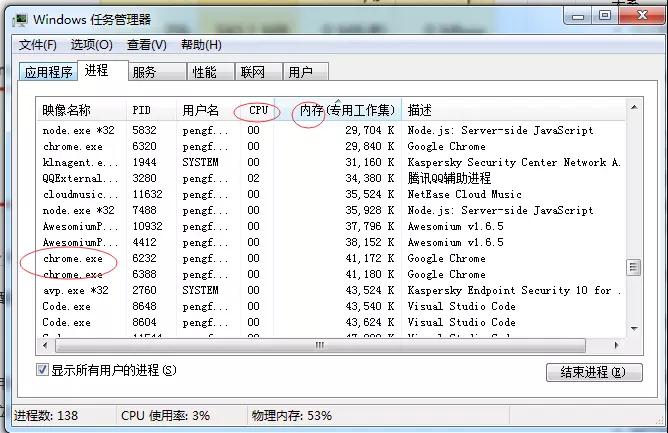
**什么是进程线程？**



进程：是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

线程：程序执行流的最小单元。

解释：平常我们在使用电脑过程中，假如我想登录微信，那么我双击微信图标登录即可。从表面上看，桌面打开了微信界面。实际上，系统也相应的开启了一个进程（一个程序可能会有多个进程，比如qq主进程，qq辅助进程等）。系统会给进程分配相应的cpu资源以及内存资源来保证进程运行。



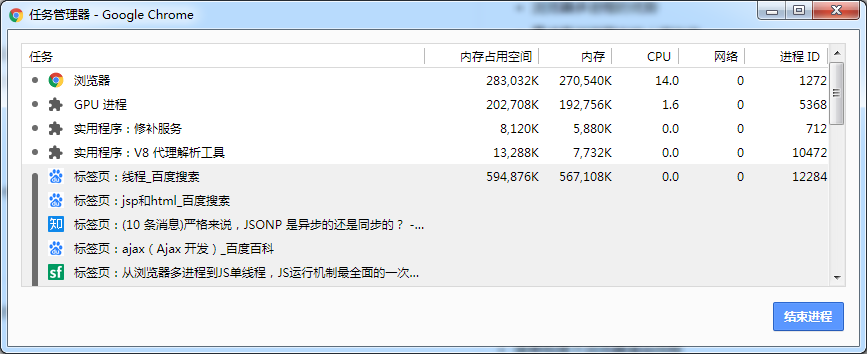
任务管理器：相应任务管理器也会有相对应的进程，以及该进程的cpu占用率、内存占用资源。

实际我们一个程序对应一个进程（这里直接按对应一个来举例），进程也可以细分即线程。比如我通过微信正在聊天，我想看看朋友圈，相应的微信进程会分配朋友圈线程来执行。即进程是种类，线程是种类下的具体列表。进程管理着线程。

浏览器是多进程：

实际浏览器也会有自己的进程。

我们在浏览器打开每打开一个tab页，就相当于创建了一个独立的浏览器tab进程。类似windows的任务管理器，我们可以打开谷歌的进程管理器。



**注意：在这里浏览器应该也有自己的优化机制，有时候打开多个tab页后，可以在Chrome任务管理器中看到，有些进程被合并了，譬如打开多个空白标签页后，会发现多个空白标签页被合并成了一个进程。**

**（所以每一个Tab标签对应一个进程并不一定是绝对的）**

**1.浏览器都包含哪些进程（**我们在浏览器打开每打开一个tab页，就相当于创建了一个独立的浏览器tab进程**）**

Browser进程(即上篇文章截图里面的浏览器进程)：浏览器的主进程（负责协调、主控），只有一个。主要作用：

负责浏览器界面显示，与用户交互。如前进，后退等

负责各个页面的管理，创建和销毁其他进程

将渲染（Renderer）进程得到的内存中的Bitmap（位图），绘制到用户界面上

网络资源的管理，下载等

第三方插件进程：每种类型的插件对应一个进程，仅当使用该插件时才创建

GPU进程：最多一个，用于3D绘制等

浏览器渲染进程（即通常所说的浏览器内核）（Renderer进程，内部是多线程的）：主要作用为页面渲染，脚本执行，事件处理等

**浏览器内核定义：浏览器最重要或者说核心的部分是“Rendering Engine”，可大概译为“渲染引擎”，不过我们一般习惯将之称为“浏览器内核”**

**2.浏览器多进程的优势**

相比于单进程浏览器，多进程有如下优点：

避免单个page crash影响整个浏览器

避免第三方插件crash影响整个浏览器

多进程充分利用多核优势

方便使用沙盒模型隔离插件等进程，提高浏览器稳定性

简单点理解：如果浏览器是单进程，那么某个Tab页崩溃了，就影响了整个浏览器，体验有多差；同理如果插件崩溃了也会影响整个浏览器；而且多进程还有其它的诸多优势。当然，多进程，内存等资源消耗也会更大，有点空间换时间的意思。

对于普通的前端操作来说，最终要的是渲染进程。因为页面的渲染，JS的执行，事件的触发，都在这个进程内进行的。接下来重点分析这个进程。

注：进程一般是多线程的

**1.渲染进程包括哪些线程**

GUI渲染线程

负责渲染浏览器界面，解析HTML，CSS，构建DOM树和RenderObject树，布局和绘制等。

当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时，该线程就会执行

注意，**GUI渲染线程与JS引擎线程是互斥的**，当JS引擎执行时GUI线程会被挂起（相当于被冻结了），GUI更新会被保存在一个队列中**等到JS引擎空闲时**立即被执行。

JS引擎线程(单线程)

也称为JS内核，负责处理Javascript脚本程序。（例如常常听到的谷歌浏览器的V8引擎，新版火狐的JaegerMonkey引擎等）

JS引擎线程负责解析Javascript脚本，运行代码。

JS引擎一直等待着**任务队列**中任务的到来，然后加以处理，一个Tab页（renderer进程）中无论什么时候都只有一个JS线程在运行JS程序

同样注意，**GUI渲染线程与JS引擎线程是互斥的**，所以如果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。3事件触发线程

3 事件触发线程

归属于渲染进程而不是JS引擎，用来控制**事件轮询**（可以理解，JS引擎自己都忙不过来，需要浏览器另开线程协助）

当JS引擎执行代码块如鼠标点击、AJAX异步请求等，会将对应任务添加到事件触发线程中

当对应的事件符合触发条件被触发时，该线程会把事件添加到待处理**任务队列**的队尾，等待JS引擎的处理

注意，由于JS的单线程关系，所以这些待处理队列中的事件都得排队等待JS引擎处理（当JS引擎空闲时才会去执行）

4 定时触发器线程

定时器setInterval与setTimeout所在线程

浏览器定时计数器并不是由JavaScript引擎计数的,（因为JavaScript引擎是单线程的, 如果任务队列处于阻塞线程状态就会影响记计时的准确）

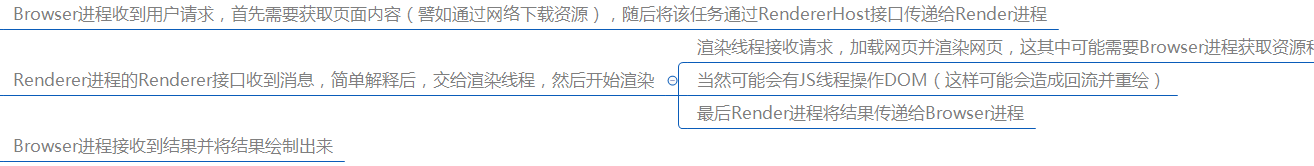
因此通过单独线程来计时并触发定时（计时完毕后，添加到事件队列中，等待JS引擎空闲后执行）

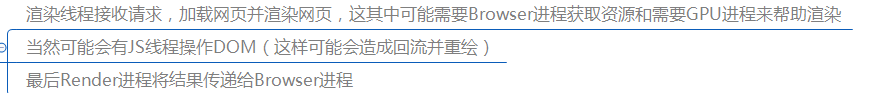
注意，W3C在HTML标准中规定，规定要求setTimeout中低于4ms的时间间隔算为4ms。

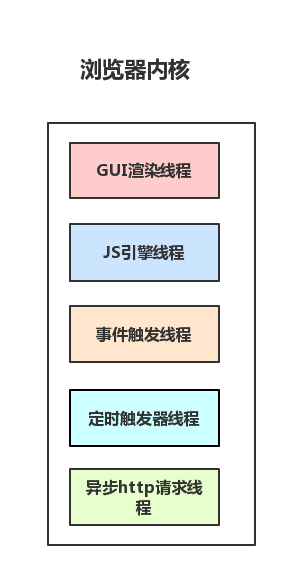
5.异步http请求线程

用于处理请求[XMLHttpRequest](https://link.jianshu.com?t=http%3A%2F%2Fwww.w3school.com.cn%2Fxml%2Fxml_http.asp)，在连接后是通过浏览器新开一个线程请求。如ajax，是浏览器新开一个http线程

将检测到状态变更（如ajax返回结果）时，如果设置有回调函数，异步线程就产生状态变更事件，将这个回调再放入js引擎线程的事件队列中。再由JavaScript引擎执行。







**3.js引擎是单线程的**

我们知道js是单线程的。也就是说，同一个时间只能做一件事。那么，为什么JavaScript不能有多个线程呢？这样能提高效率啊。

参考阮一峰大神的文章[js事件轮询（Event Loop）](https://link.jianshu.com?t=http%3A%2F%2Fwww.ruanyifeng.com%2Fblog%2F2014%2F10%2Fevent-loop.html)

JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？

所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

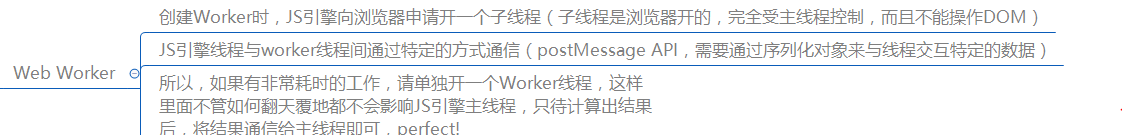
为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

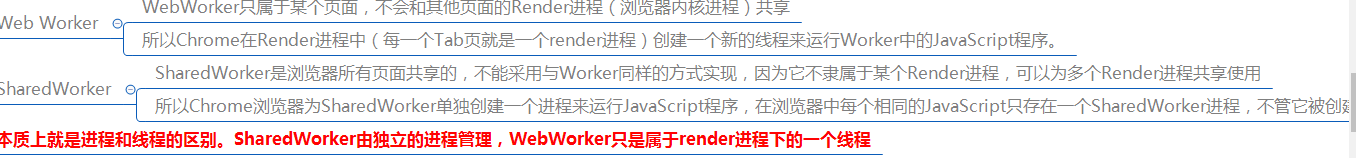
**webworker：**

Web Worker 的作用，就是为 JavaScript 创造多线程环境，允许主线程创建 Worker 线程，将一些任务分配给后者运行。在主线程运行的同时，Worker 线程在后台运行，两者互不干扰。等到 Worker 线程完成计算任务，再把结果返回给主线程。这样的好处是，一些计算密集型或高延迟的任务，被 Worker 线程负担了，主线程（通常负责 UI 交互）就会很流畅，不会被阻塞或拖慢。

Worker 线程一旦新建成功，就会始终运行，不会被主线程上的活动（比如用户点击按钮、提交表单）打断。这样有利于随时响应主线程的通信。但是，这也造成了 Worker 比较耗费资源，不应该过度使用，而且一旦使用完毕，就应该关闭。

<http://www.ruanyifeng.com/blog/2018/07/web-worker.html>（webworker的使用）





**2.渲染进程中的线程之间的关系**

GUI渲染线程与JS引擎线程互斥

由于JavaScript是可操纵DOM的，如果在修改这些元素属性同时渲染界面（即JS线程和GUI线程同时运行），那么渲染线程前后获得的元素数据就可能不一致了。

因此为了防止渲染出现不可预期的结果，浏览器设置GUI渲染线程与JS引擎为互斥的关系，当JS引擎执行时GUI线程会被挂起，

GUI更新则会被保存在一个队列中等到JS引擎线程空闲时立即被执行。

JS阻塞页面加载

从上述的互斥关系，可以推导出，JS如果执行时间过长就会阻塞页面。

譬如，假设JS引擎正在进行巨量的计算，所以JS引擎很可能很久很久后才能空闲，所以导致页面渲染加载阻塞。这就牵扯到script标签在html中的存放位置。

**3.js引擎是单线程的**

我们知道js是单线程的。也就是说，同一个时间只能做一件事。那么，为什么JavaScript不能有多个线程呢？这样能提高效率啊。

参考阮一峰大神的文章[js事件轮询（Event Loop）](https://link.jianshu.com?t=http%3A%2F%2Fwww.ruanyifeng.com%2Fblog%2F2014%2F10%2Fevent-loop.html)

JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？

所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

为了利用多核CPU的计算能力，HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

**4.js事件轮询**

作者：YINdevelop

链接：https://www.jianshu.com/p/05606b0b4eb1

來源：简书

简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。