异步加载

**一.DOM树**

**（1）**首先形成DOM树(深度优先：一条枝干解析到头再看另一条)

document

html

head body

… …

**（2）**src=“xxx”把节点挂到树上，是解析完毕，不是下载完成，是与DOM树形成是异步发生的，也就是解析完毕，画面并没有加载完毕<img src=“xxx”></img>

**（3）**DOM树生成以后等着CSS树形成，对应完事

**（4）**CSS树形成以后和DOM树拼在一起形成randerTree，然后渲染引擎开始绘制画面

**DOM树被JS间接改变，randerTree就重新生成，形成reflow重排，效率低**

1.reflow回流:**1.**DOM节点的删除和添加

**2.**DOM节点的高度变化，位置变化，

**3.**display none变成display block

**4.**offsetWidth计算DOM节点的位置

2.repaint重绘，基于CSS，重绘是部分

reflow是会影响父级以及周边元素，repaint不影响其他元素，自身的改变，比如颜色

3.回流必定会发生重绘，重绘不一定会引发回流。

**二.异步加载JS**

JS同步加载的缺点：加载工具方法没必要阻塞文档，JS加载会影响页面效率，一旦网速不好，那么整个网站将等待JS加载而不进行后续渲染等工作。

有些工具方法需要按需加载，用到了加载用不到不加载啊

JS一般是同步的可以阻断加载线，不能与css和html同时，因为JS修改他俩

**异步加载的方法**

**（1）defer** 异步加载，(与css和html文件同时加载，互不影响)要等到dom文档全部解析完成（dom树生成完）才会被执行，**只有ie9以下能用**，也可以将代码写到内部。

**<script type="text/javascript" src="js.js" defer="defer"></script>**

**（2）aysnc** W3C标准方法,异步加载，加载完就能执行，ayanc只能加载外部脚本，不能把js代码写在script标签里

**<script type="text/javascript" src="js.js" aysnc="aysnc"></script>**

**（3）创建script,插入到dom中，记载完毕后callBack！！！最重要，可以按需加载**

<script type="text/javascript">

var script=document.createElement(“script”);

script.type="text/javascript"

script.src="js.js"

document.head.appendChild(script);放到页面里才会执行js.js

</script>

判断上面的代码体加载完了没有

在除了IE的浏览器里

用script.onload=function(){

js里函数执行

}

在IE浏览器里用script上的一个属性叫做readyState 状态码开始存的是loading加载完成值变成loaded||complete

再根据事件onreadystatechange去监听

scirpt. onreadystatechange=function ( ){

if(script.readyState==”complete”||”loaded”)

函数执行

}

**随用随取**

方法在js文件里，用到了在下载js文件执行里面的方法

function loadScript(url, callbackFn) {url是目标js文件，callBackFn是执行的函数（不是写在目标js文件里）

var script = document.createElement("script");

script.type = "text/javascript"

if (script.readyState) {//如果是ie

script.onreadyStatechange = function ( ) {

if (script.readyState == complete || script.readyState == loaded) {

callbackFn( );

}

}

}

else {

script.onload = function () {//如果不是ie

callbackFn( );

}

}

script.src = url;

document.head.appendChild(script);

}

**三.时间线，记录浏览器出生时起**

（1）**创建Document对象，开始解析web页面。**解析html元素和他们的文本内容后添加Element对象和Text对象节点到文档中。这个**阶段document.readyState=“loading”**

（2）遇到link外部css，创建线程加载，并继续解析文档。

（3）遇到script外部js，并且没有设置async和defer，浏览器加载，并阻塞，等待js加载完成并执行该脚本，然后继续解析文档。

（4）遇到script外部js，并且设有async，defer，浏览器创建线程加载，并继续解析文档，对于async属性的脚本，脚本加载完成后立即执行。（异步禁止使用document.write( )会清空文档）

（5）遇到img等，先正常解析dom结构，然后浏览器异步加载src里面的内容，并继续解析文档。

（6）当文档解析完成，domTree建立完，**document.readyState=“interactive”**。用readystatechange事件去监听可以得到，document. onreadystatechange=function(){

console.log(document.readyState)}

（7）文档解析完成后，所有设置有defer的脚本会按照**顺序**执行。（注意与async的不同，但同样禁止使用document.write（））

（8）document对象触发**DOMContentLoaded**事件dom解析完执行（只有addEventListener有）,这也标志着程序执行从同步脚本执行阶段,转化为事件驱动阶段。

（9）当所有async的脚本加载完成并执行后,img等加载完成后**document.readyState=“complete”,window对象触发load事件**。

（10）从此，以异步想要方式处理用户输入，网络事件等。

**四.浏览器的渲染模式**

(1)浏览器采用流式布局模型

(2)浏览器会把HTML解析成DOM，把CSS解析成CSSOM，DOM和CSSOM合并就产生了渲染树（Render Tree）

(3)有了RenderTree，我们就知道了所有节点的样式，然后计算他们在页面上的大小和位置，最后把节点绘制到页面上。

(4)由于浏览器使用流式布局，对Render Tree的计算通常只需要遍历一次就可以完成，但table及其内部元素除外，他们可能需要多次计算，通常要花3倍于同等元素的时间，这也是为什么要避免使用table布局的原因之一。

**五.浏览器优化**

应该避免频繁使用以下属性，它们都会强制渲染刷新队列

offsetTop,offsetLeft,offsetWidth,offsetHeight

scrollTop,scrollLeft,scrollWidth,scrollHeight

clientTop,clientLeft,clientWidth,clientHeight

width,height

getComputedStyle()

getBoundingClientRect()