# 浏览器页面输入url

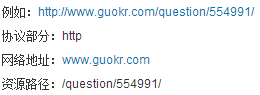
## 发生的过程（步骤）

1. 浏览器查询缓存，如果存在缓存，则直接跳至第9步
2. 接受url，并拆分成协议、网络地址、资源路径。网络地址指示连接网络

上哪一台计算机，可以是域名或者IP地址，可以包括端口号；协议是从

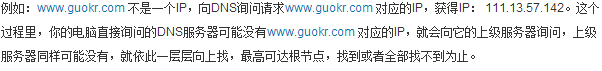
该计算机获取资源的方式（如HTTP，FTP）；资源路径指示从服务器上 获取哪一项资源。

例子：



1. 与缓存进行对比，如果请求的对象在缓存中，则直接进行第9步
2. 浏览器询问操作系统服务器ip
3. 操作系统做dns（域名系统）查询，返回ip地址给浏览器
4. 如果网络地址不是一个IP地址。向操作系统询问，操作系统返回一个IP 地址，操作系统通过DNS将该地址解析成IP地址，IP地址对应着一个 网络上一台计算机，DNS服务器本身也有IP。

例子：



如果地址不包含端口号，根据协议的默认端口确定一个，端口号之于计算

机就像窗口之于银行，一家银行有多个窗口，每隔窗口都有个号码，不同

的窗口可以负责不同的服务，端口只是一个逻辑概念，和计算机硬件没有

关系

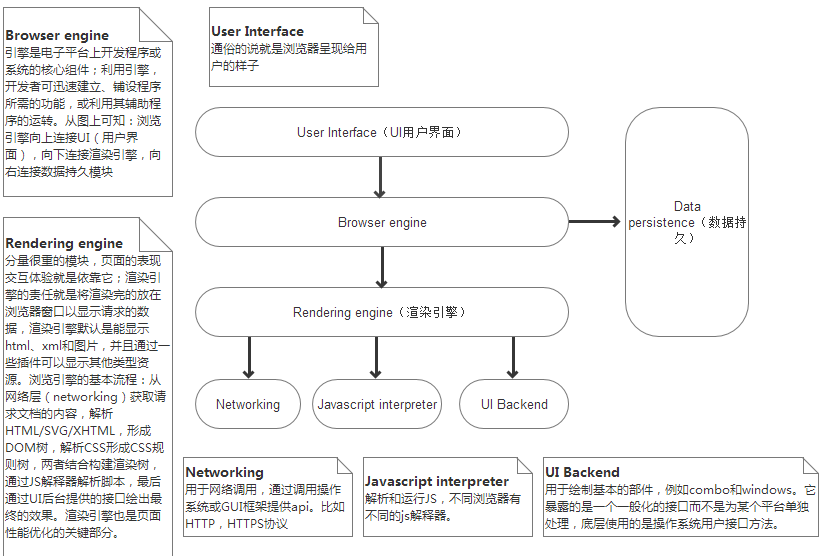
IMG_256

1. 浏览器打开对服务器的tcp连接（tcp连接三次握手，四次分手）
2. 浏览器通过tcp发送http请求（HTTP三次握手）
3. 浏览器接受响应并且可能关掉tcp连接，或者是重新使用连接处理新请求（为另一个连接保留）
4. 浏览器检查响应是否为一个重定向（3XX结果状态码），或者是重新使用连接处理新请求
5. 检查HTTP header里的状态码，做出不同的处理方式，比如：错误（4XX，

5XX）、重定向（3XX）、授权请求（2XX）

1. 如果响应可缓存，将存入缓存
2. 浏览器响应解码
3. 浏览器决定如何处理这些响应（html，音乐，图片）
4. 浏览器展现响应，对未知的mime类型还会弹出下载对话框
5. 浏览器渲染响应，或者不能识别的类型提供下载提示框

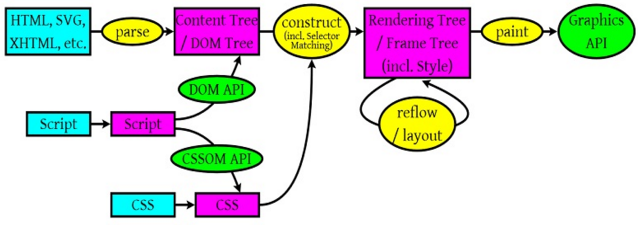
## 看图说话（解释）



## 浏览器的渲染原理

<http://coolshell.cn/articles/9666.html>

1. **浏览器工作流程**



从图中可以看出：

1. 浏览器会解析三个东西：
2. 解析HTML/SVG/XHTML，事实上，Webkit有三个C++的类对应这三类文

档，解析这三类文档会产生一个DOM Tree

1. CSS，解析CSS会产生CSS规则树，CSS Rule Tree
2. javaScript，脚本，只要是通过DOM API和CSSOM API 来操作DOM Tree

和 CSS Rule Tree

1. 解析上面三个东西完成后，浏览器会通过DOM Tree 和 CSS Rule Tree 来构造Rendering Tree。注意：
2. Rendering Tree （渲染树）并不等同于 DOM Tree ，因为一些像 Header 或

display：none的东西就没有必要放在渲染树中

1. CSS Rule Tree主要是为了完成匹配并把CSS Rule附加在Rendering Tree

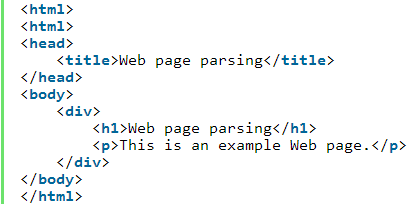
上的每一个Element（即DOM结点或者Frame）。

1. 计算每个Frame（即每个Element）的位置，此过程叫layout（布局）和reflow

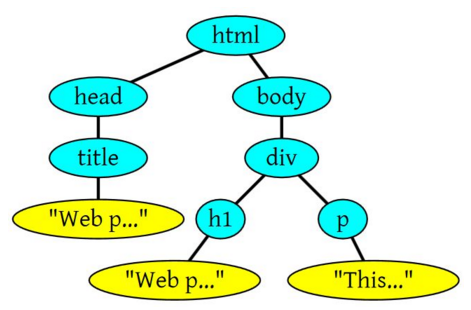
过程

1. 最后通过调用操作系统Native（原生） GUI的API绘制
2. **具体解析例子（以火狐为例）**
3. DOM 解析

* HTML代码：

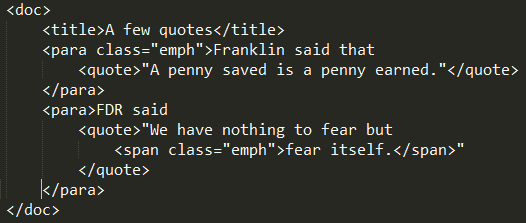


* HTML解析为DOM Tree ：

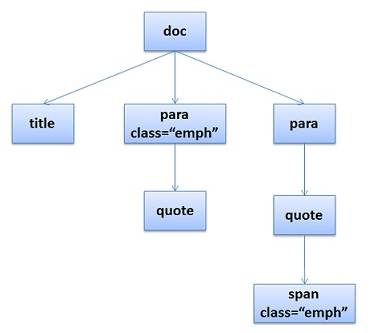


1. CSS解析

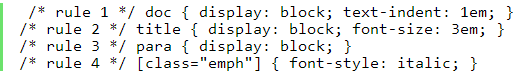
* HTML代码：



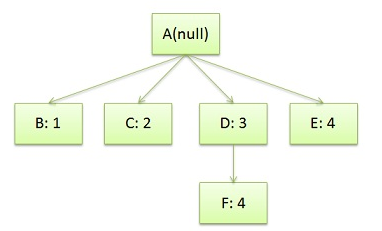
* DOM Tree：



* CSS代码：



* **CSS Rule Tree：**

****

**由上几图结合可知：**

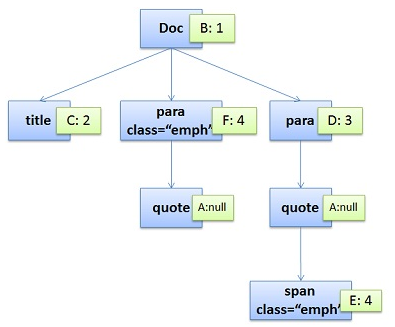
* **图中的第4条规则出现了两次，一次是独立的，一次是在规则3的子结点，**

**因此，可以知道，建立CSS Rule Tree是需要参照DOM Tree**

**来建立的，CSS 匹配DOM Tree 主要是从右到左解析CSS的Selector，**

**CSS匹配HTML元素是一个相当复杂和有性能问题的事情，所以，你会在很 多网页看到，DOM树要小，CSS尽量用id和class，千万不要过度层叠下去。**

* **通过这两个树，可以得到一个叫Style Context Tree （把CSS Rule结点Attach到DOM Tree上）**



* CSS引擎查找样式表，对每条规则都按照从右到左的顺序去匹配。例如：
* 

CSS选择符是从右到左进行匹配的。了解这方面的知识后，我们知道这个之前看似高效地规则实际开销相当高，浏览器必须遍历页面上每个li元素并确定其父元素的id是否为nav。

* 

这种效率是差到极点的做法，因为\*通配符将匹配所有元素，所以浏览器必须去遍历每一个元素，这样的计算次数可能是上万次！

* 

在页面中一个指定的ID只能对应一个元素，所以没有必要添加额外的限定符，而且这会使它更低效。同时也不要用具体的标签限定类选择符，而是要根据实际的情况对类名进行扩展。例如把ul.nav改成.main\_nav更好。

* 

对于这样的选择器，之前也写过，最后自己也数不过来有多少后代选择器了，何不用一个类来关联最后的标签元素，如.extra\_navitem，这样只需要匹配class为extra\_navitem的元素，效率明显提升了

* 在css书写过程中，性能提升的几个方案：
* 避免使用通配规则 如 \*{} 计算次数惊人！只对需要用到的元素进行选择
* 尽量少的去对标签进行选择，而是用class 如：#nav li{},可以为li加上nav\_item的类名，如下选择.nav\_item{}
* 不要去用标签限定ID或者类选择符   如：ul#nav,应该简化为#nav
* 尽量少的去使用后代选择器，降低选择器的权重值  后代选择器的开销是最高的，尽量将选择器的深度降到最低，最高不要超过三层，更多的使用类来关联每一个标签元素
* 考虑继承 了解哪些属性是可以通过继承而来的，然后避免对这些属性重复指定规则

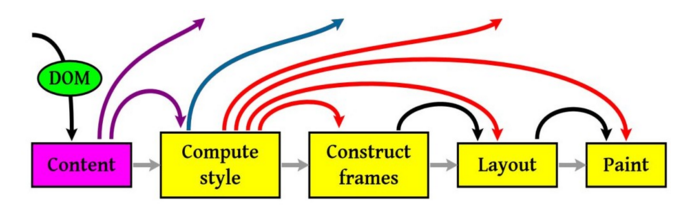
1. **渲染**

* 渲染流程（黄色的四个步骤）：

1. 计算css样式
2. 构建Render Tree
3. Layout--定位坐标和大小，是否换行，各种position，overflow，z-index

属性and so on

1. 正式开画



**注意：**流程中有很多连接线，这表示了js动态修改DOM属性或者css属性会导

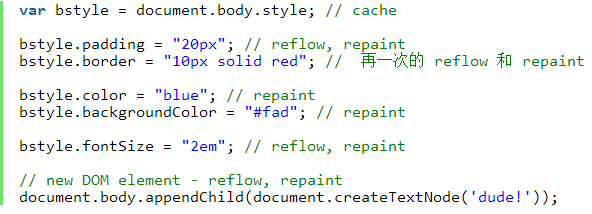
致重新layout，有些改变不会，比如指向天上的箭头，如，修改后的css rule没

有被匹配到。

* Repaint--屏幕的一部分要重画，比如，某个css背景色变了，但是元素的几何尺寸没有变
* Reflow--意味着元件的几何尺寸变了，需要重新验证并计算Render Tree，是Render Tree 的一部分或者全部分发生了变化（或者Layout），HTML使用的是flow based layout（流式布局），因此，如果某元件的几何尺寸发生了变化，需要重新布局（reflow），reflow会从<html>的root Element开始递归往下，一次计算所有结点的几何尺寸和位置，在reflow中，可能会增加一些frame（Element），比如一个文本字符串必须被包装起来。
* Reflow的成本比Repaint的成本高得多的多。DOM Tree里的每个结点都会有reflow方法，一个结点的reflow很有可能导致子结点，甚至父点以及同级结点的reflow。在一些高性能的电脑上也许还没什么，但是如果reflow发生在手机上，那么这个过程是非常痛苦和耗电的。
* 比如：
  + 当你增加、删除、修改DOM结点时，会导致Reflow或Repaint
  + 当你移动DOM的位置，或是搞个动画的时候。
  + 当你修改CSS样式的时候。
  + 当你Resize窗口的时候（移动端没有这个问题），或是滚动的时候。
  + 当你修改网页的默认字体时

**注**：display:none会触发reflow，而visibility:hidden只会触发repaint，因为没有发现位置变化。

* Reflow的原因：
  + Initial。网页初始化的时候。
  + Incremental（增量型）。一些Javascript在操作DOM Tree时。
  + Resize。其些元件的尺寸变了。
  + StyleChange。如果CSS的属性发生变化了。
  + Dirty。几个Incremental的reflow发生在同一个frame的子树上。
* 比如：

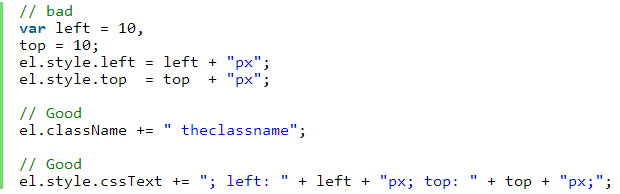


**注：** 当然，浏览器是聪明的，它不会像上面那样，你每改一次样式，它就reflow或repaint一次。一般来说，浏览器会把这样的操作积攒一批，然后做一次reflow，这又叫异步reflow或增量异步reflow。但是有些情况浏览器是不会这么做的，比如：resize窗口，改变了页面默认的字体，等。对于这些操作，浏览器会马上进行reflow。

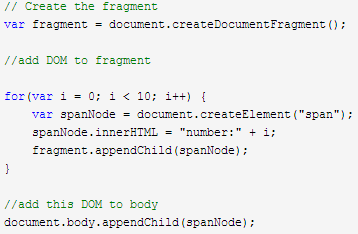
* 特殊情况：
* offsetTop, offsetLeft, offsetWidth, offsetHeight
* scrollTop/Left/Width/Height
* clientTop/Left/Width/Height
* IE中的 getComputedStyle(), 或 currentStyle

**注：**当请求上面的DOM值时，那么浏览器需要返回最新的值，而这样一样会flush出去一些样式的改变，从而造成频繁的reflow/repaint。

* 减少reflow或repaint
* 不要一条一条地修改DOM的样式。与其这样，还不如预先定义好css的class，然后修改DOM的className。



* 把DOM离线后修改，如：
  + - 使用documentFragment 对象在**内存里**操作DOM
* DocumentFragment对象表示邻接结点和它们的子树，
* DocumentFragment 接口表示文档的一部分（或一段）。更确切地说，它表示一个或多个邻接的 [Document 节点](http://www.w3school.com.cn/xmldom/dom_document.asp" \o "XML DOM - Document 对象)和它们的所有子孙节点。
* DocumentFragment 节点**不属于文档树**，继承的 parentNode 属性总是 null。
* 不过它有一种特殊的行为，该行为使得它非常有用，即当请求把一个 DocumentFragment 节点插入文档树时，**插入的不是 DocumentFragment 自身，而是它的所有子孙节点。这使得 DocumentFragment 成了有用的占位符，暂时存放那些一次插入文档的节点。它还有利于实现文档的剪切、复制和粘贴操作。**
* 可以用 Document.createDocumentFragment() 方法创建新的空 DocumentFragment 节点。也可以用 [Range.extractContents() 方法](http://www.w3school.com.cn/xmldom/met_range_extractcontents.asp" \o "XML DOM extractContents() 方法) 或 [Range.cloneContents() 方法](http://www.w3school.com.cn/xmldom/met_range_clonecontents.asp" \o "XML DOM cloneContents() 方法) 获取包含现有文档的片段的 DocumentFragment 节点。
* 例子：



* + - 先把DOM给**display:none(有一次reflow)**，然后你想怎么改就怎么改。比如修改100次，然后再把他显示出来。
    - **clone一个DOM结点到内存里**，然后想怎么改就怎么改，改完后，和在线的那个的交换一下
* **不要把DOM结点的属性值放在一个循环里当成循环里的变量。**不然这会导致大量地读写这个结点的属性。
* **尽可能的修改层级比较低的DOM。**当然，改变层级比较底的DOM有可能会造成大面积的reflow，但是也可能影响范围很小。
* **为动画的HTML元件使用fixed或absolute的position**，那么修改他们的CSS是不会reflow的。
* **千万不要使用table布局。**因为可能很小的一个小改动会造成整个table的重新布局。