清风软件测试

前端性能优化方法总结

2018-05-02 01:17 清风软件测试 阅读(6416) 评论(0)

本文章为转载别人的文章，因不知道原博客的地址，再次非常感谢原
无私分享

前端性能优化（一）

前端是庞大的，包括 HTML、 CSS、 Javascript、Image 、Flash等
样的资源。前端优化是复杂的，针对方方面面的资源都有不同的方式。那
么，前端优化的目的是什么？

1. 从用户角度而言，优化能够让页面加载得更快、对用户的操作响应得更及时，能够给用户提供更为友好的体验。

2. 从服务商角度而言，优化能够减少页面请求数、或者减小请求
宽，能够节省可观的资源。

总之，恰当的优化不仅能够改善站点的用户体验并且能够节省相当的资源利用。

前端优化的途径有很多，按粒度大致可以分为两类，第一类是页面级别的优化，例如 HTTP请求数、脚本的无阻塞加载、内联脚本的位置优化等；第二类则是代码级别的优化，例如 Javascript中的DOM 操作优化、CSS选择符优化、图片优化以及 HTML结构优化等等。另外，本着提高投入产出比的目的，后文提到的各种优化策略大致按照投入产出比从大到小的顺序排列。

一、页面级优化

1. 减少 HTTP请求数

这条策略基本上所有前端人都知道，而且也是最重要最有效的。都说要减少 HTTP请求，那请求多了到底会怎么样呢？首先，每个请求都是有成本的，既包含时间成本也包含资源成本。一个完整的请求都需要经过 DNS寻址、与服务器建立连接、发送数据、等待服务器响应、接收数据这样
长”而复杂的过程。时间成本就是用户需要看到或者“感受”到这个资
须要等待这个过程结束的，资源上由于每个请求都需要携带数据，因此每
请求都需要占用带宽。另外，由于浏览器进行并发请求的请求数是有上限的(具体参见此处)，因此请求数多了以后，浏览器需要分批进行请求，因此会增加用户的等待时间，会给用户造成站点速度慢这样一个印象，即使可能用户能看到的第一屏的资源都已经请求完了，但是浏览器的进度条会一直存在。

减少 HTTP请求数的主要途径包括：

(1). 从设计实现层面简化页面

如果你的页面像百度首页一样简单，那么接下来的规则基本上都用不着了。保持页面简洁、减少资源的使用时最直接的。如果不是这样，你的页面需要华丽的皮肤，则继续阅读下面的内容。

(2). 合理设置 HTTP缓存

缓存的力量是强大的，恰当的缓存设置可以大大的减少 HTTP请
啊首页为例，当浏览器没有缓存的时候访问一共会发出 78个请求，共 600多 K数据 (如图 1.1)，而当第二次访问即浏览器已缓存之后访问则仅有 10个请求，共 20多 K数据 (如图 1.2)。(这里需要说明的是，如果直接 F5刷新页面的话效果是不一样的，这种情况下请求数还是一样，不过被缓存资源的请求服务器是 304响应，只有 Header没有Body，可以节省带宽)

怎样才算合理设置？原则很简单，能缓存越多越好，能缓存越久越好。例

About

昵称： 清风软件测试
园龄： 3年1个月
粉丝： 386
关注： 4

+加关注

SEARCH

最新评论

Re:网站系统用的架构演变过程
博主写的很好，求其他博文的阅读密码，谢谢！774797238@qq.com -- whh5353

Re:PageObjects 设计模式
可以提供一下博文阅读密码吗，非常希望可以阅览，或者邮箱给我447895909@qq.com -- elma_hu

Re:Runtime.getRuntime().exec()需要注意的地方
太棒了，用你的方法终于解决了困扰一天半的一个问题。多谢大佬 -- 想成功就得冲

Re:Jmeter接口测试自动化 （三） （数据驱动测试）
很多文章被加密了 请问能提供密码阅读下吗 -- 忧桑的二五仔

Re:PageObjects 设计模式
可以提供一下博客密码吗，非常希望可以阅览，或者邮箱给我826724966@qq.com -- 热爱测试

最新评论

Re:网站系统用的架构演变过程
博主写的很好，求其他博文的阅读密码，谢谢！774797238@qq.com -- whh5353

Re:PageObjects 设计模式
可以提供一下博文阅读密码吗，非常希望可以阅览，或者邮箱给我447895909@qq.com -- elma_hu

Re:Runtime.getRuntime().exec()需要注意的地方
太棒了，用你的方法终于解决了困扰一天半的一个问题。多谢大佬 -- 想成功就得冲

Re:Jmeter接口测试自动化 （三） （数据驱动测试）
很多文章被加密了 请问能提供密码阅读下吗 -- 忧桑的二五仔

Re:PageObjects 设计模式
可以提供一下博客密码吗，非常希望可以阅览，或者邮箱给我826724966@qq.com -- 热爱测试

日历

< 2019年11月 >

日	一	二	三	四	五	六
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23

随笔档案

2019年11月(1)
2019年10月(14)
2019年9月(17)
2019年8月(31)
2019年7月(33)

如，很少变化的图片资源可以直接通过 HTTP Header中的Expires设置一个很长的过期头 ;变化不频繁而又可能会变的资源可以使用 Last-Modifed来做请求验证。尽可能的让资源能够在缓存中待得更久。关于 HTTP缓存的具体设置和原理此处就不再详述了，有兴趣的可以参考下列文章：

HTTP1.1协议中关于缓存策略的描述

Fiddler HTTP Performance中关于缓存的介绍

(3). 资源合并与压缩

如果可以的话，尽可能的将外部的脚本、样式进行合并，多个合为一个。另外， CSS、 Javascript、Image 都可以用相应的工具进行压缩，压缩后往往能省下不少空间。

(4). CSS Sprites

合并 CSS图片，减少请求数的又一个好办法。

(5). Inline Images

使用 data: URL scheme的方式将图片嵌入到页面或 CSS中，如果不考虑资源管理上的问题的话，不失为一个好办法。如果是嵌入页面的话换来的是增大了页面的体积，而且无法利用浏览器缓存。使用在 CSS中的图片则更为理想一些。

(6). Lazy Load Images（自己对这一块的内容还是不了解）

这条策略实际上并不一定能减少 HTTP请求数，但是却能在某些条件下或者页面刚加载时减少 HTTP请求数。对于图片而言，在页面刚加载的时候可以只加载第一屏，当用户继续往后滚屏的时候才加载后续的图片。这样一来，假如用户只对第一屏的内容感兴趣时，那剩余的图片请求就都节省了。有啊首页 曾经的做法是在加载的时候把第一屏之后的图片地址缓存在 Textarca标签中，待用户往下滚屏的时候才 “惰性” 加载。

2. 将外部脚本置底（将脚本内容在页面信息内容加载后再加载）

前文有谈到，浏览器是可以并发请求的，这一特点使得其能够更快的加载资源，然而外链脚本在加载时却会阻塞其他资源，例如在脚本加载完成之前，它后面的图片、样式以及其他脚本都处于阻塞状态，直到脚本加载完成后才会开始加载。如果将脚本放在比较靠前的位置，则会影响整个页面的加载速度从而影响用户体验。解决这一问题的方法有很多，在 这里有比较详细的介绍 (这里是译文和 更详细的例子)，而最简单可依赖的方法就是将脚本尽可能的往后挪，减少对并发下载的影响。

3. 异步执行 inline脚本(其实原理和上面是一样，保证脚本在页面内容后面加载。)

inline脚本对性能的影响与外部脚本相比，是有过之而无不及。首页，与外部脚本一样， inline脚本在执行的时候一样会阻塞并发请求，除此之外，由于浏览器在页面处理方面是单线程的，当 inline脚本在页面渲染之前执行时，页面的渲染工作则会被推迟。简而言之， inline脚本在执行的时候，页面处于空白状态。鉴于以上两点原因，建议将执行时间较长的 inline脚本异步执行，异步的方式有很多种，例如使用 script元素的defer 属性(存在兼容性问题和其他一些问题，例如不能使用 document.write)、使用setTimeout ，此外，在HTML5中引入了 Web Workers的机制，恰恰可以解决此类问题。

4. Lazy Load Javascript（只有在需要加载的时候加载，在一般情况下并不加载信息内容。）

随着 Javascript框架的流行，越来越多的站点也使用起了框架。不过，一个框架往往包括了很多的功能实现，这些功能并不是每一个页面都需要的，如果下载了不需要的脚本则算得上是一种资源浪费 -既浪费了带宽又浪费了执行花费的时间。目前的做法大概有两种，一种是为那些流量特别大的页面专门定制一个专用的 mini版框架，另一种则是 Lazy Load。YUI 则使用了第二种方式，在 YUI的实现中，最初只加载核心模块，其他模块可以等到需要使

用的时候才加载。

5. 将 CSS放在 HEAD中

如果将 CSS放在其他地方比如 BODY中，则浏览器有可能还未下载和解析到 CSS就已经开始渲染页面了，这就导致页面由无 CSS状态跳转到 CSS状态，用户体验比较糟糕。除此之外，有些浏览器会在 CSS下载完成后才开始渲染页面，如果 CSS放在靠下的位置则会导致浏览器将渲染时间推迟。

6. 异步请求 Callback（就是将一些行为样式提取出来，慢慢的加载相应的内容）

在某些页面中可能存在这样一种需求，需要使用 script标签来异步的请求数据。类似：

Javascript:

```
function myCallback(info){
//do something here
}
```

HTML:

cb返回的内容：

```
myCallback('Hello world!');
```

像以上这种方式直接在页面上写 <script>对页面的性能也是有影响的，即增加了页面首次加载的负担，推迟了 DOMLoaded和window.onload 事件的触发时机。如果时效性允许的话，可以考虑在 DOMLoaded事件触发的时候加载，或者使用 setTimeout方式来灵活的控制加载的时机。

7. 减少不必要的 HTTP跳转

对于以目录形式访问的 HTTP链接，很多人都会忽略链接最后是否带’/’，假如你的服务器对此是区别对待的话，那么你也需要注意，这其中很可能隐藏了 301跳转，增加了多余请求。具体参见下图，其中第一个链接是以无’/’结尾的方式访问的，于是服务器有了一次跳转。

8. 避免重复的资源请求

这种情况主要是由于疏忽或页面由多个模块拼接而成，然后每个模块中请求了同样的资源时，会导致资源的重复请求

二、代码级优化

1. Javascript

(1). DOM

DOM操作应该是脚本中最耗性能的一类操作，例如增加、修改、删除DOM元素或者对 DOM集合进行操作。如果脚本中包含了大量的 DOM操作则需要注意以下几点：

a. HTML Collection（HTML收集器，返回的是一个数组内容信息）

在脚本中 document.images、document.forms 、getElementsByTagName()返回的都是 HTMLCollection类型的集合，在平时的使用的时候大多将它作为数组来使用，因为它有 length属性，也可以使用索引访问每一个元素。不过在访问性能上则比数组要差很多，原因是这个集合并不是一个静态的结果，它表示的仅仅是一个特定的查询，每次访问该集合时都会重新执行这个查询从而更新查询结果。所谓的“访问集合”包括读取集合的 length属性、访问集合中的元素。

因此，当你需要遍历 HTML Collection的时候，尽量将它转为数组后再访问，以提高性能。即使不转换为数组，也请尽可能少的访问它，例如在遍历的时候可以将 length属性、成员保存到局部变量后再使用局部变量。

b. Reflow & Repaint

除了上面一点之外， DOM操作还需要考虑浏览器的 Reflow和Repaint，

因为这些都是需要消耗资源的，具体的可以参加以下文章：

如何减少浏览器的repaint和reflow?

Understanding Internet Explorer Rendering Behaviour

Notes on HTML Reflow

(2). 慎用 with

with(obj){ p = 1}; 代码块的行为实际上是修改了代码块中的 执行环境，将 obj 放在了其作用域链的最前端，在 with代码块中访问非局部变量是都是先从 obj 上开始查找，如果没有再依次按作用域链向上查找，因此使用 with相当于增加了作用域链长度。而每次查找作用域链都是要消耗时间的，过长的作用域链会导致查找性能下降。

因此，除非你能肯定在 with代码中只访问 obj中的属性，否则慎用 with，替代的可以使用局部变量缓存需要访问的属性。

(3). 避免使用 eval和 Function

每次 eval 或 Function 构造函数作用于字符串表示的源代码时，脚本引擎都需要将源代码转换成可执行代码。这是很消耗资源的操作 —— 通常比简单的函数调用慢 100倍以上。

eval 函数效率特别低，由于事先无法知晓传给 eval 的字符串中的内容，因此只能有浏览器在运行时解释代码。这对性能影响很大。

Function 构造函数比 eval略好，因为使用此代码不会影响周围代码，但其速度仍很慢。

此外，使用 eval和 Function也不利于Javascript 压缩工具执行压

(4). 减少作用域链查找（这方面设计到一些内容的相关问题）

前文谈到了作用域链查找问题，这一点在循环中是尤其需要注意的问题。如果在循环中需要访问非本作用域下的变量时请在遍历之前用局部变量缓存该变量，并在遍历结束后再重写那个变量，这一点对全局变量尤其重要，因为全局变量处于作用域链的最顶端，访问时的查找次数是最多的。

低效率的写法：

```
// 全局变量
var globalVar = 1;
function myCallback(info){
for( var i = 100000; i--;){
//每次访问 globalVar 都需要查找到作用域链最顶端，本例中需要访问 100000 次
globalVar += i;
}
}
```

更高效的写法：

```
// 全局变量
var globalVar = 1;
function myCallback(info){
//局部变量缓存全局变量
var localVar = globalVar;
for( var i = 100000; i--;){
//访问局部变量是最快的
localVar += i;
}
//本例中只需要访问 2次全局变量
在函数中只需要将 globalVar中内容的值赋给localVar 中区
globalVar = localVar;
}
```


阅读排行榜

此外，要减少作用域链查找还应该减少闭包的使用。

(5). 数据访问

Javascript中的数据访问包括直接量 (字符串、正则表达式)、变量、对象属性以及数组，其中对直接量和局部变量的访问是最快的，对对象属性以及数组的访问需要更大的开销。当出现以下情况时，建议将数据放入局部变量：

a. 对任何对象属性的访问超过 1次

b. 对任何数组成员的访问次数超过 1次

另外，还应当尽可能的减少对对象以及数组深度查找。

(6). 字符串拼接

在 Javascript中使用"+" 号来拼接字符串效率是比较低的，因为每次运行都会开辟新的内存并生成新的字符串变量，然后将拼接结果赋值给新变量。与之相比更为高效的做法是使用数组的 join方法，即将需要拼接的字符串放在数组中最后调用其 join方法得到结果。不过由于使用数组也有一定的开销，因此当需要拼接的字符串较多的时候可以考虑用此方法。

关于 Javascript优化的更详细介绍请参考：

Write Efficient Javascript(PPT)

Efficient JavaScript

2. CSS选择符

在大多数人的观念中，都觉得浏览器对 CSS选择符的解析式从左往右进行的，例如

#toc A { color: #444; }

这样一个选择符，如果是从右往左解析则效率会很高，因为第一个 ID选择基本上就把查找的范围限定了，但实际上浏览器对选择符的解析是从右往左进行的。如上面的选择符，浏览器必须遍历查找每一个 A标签的祖先节点，效率并不像之前想象的那样高。根据浏览器的这一行为特点，在写选择符的时候需要注意很多事项，有人已经一一列举了， 详情参考此处。

3. HTML

对 HTML本身的优化现如今也越来越多的受人关注了，详情可以参见这篇 总结性文章 。

4. Image压缩

图片压缩是个技术活，不过现如今这方面的工具也非常多，压缩之后往往能带来不错的效果，具体的压缩原理以及方法在《 Even Faster Web Sites》第10 章有很详细的介绍，有兴趣的可以去看看。

总结

本文从页面级以及代码级两个粒度对前端优化的各种方式做了一个总结，这些方法基本上都是前端开发人员在开发的过程中可以借鉴和实践的，除此之外，完整的前端优化还应该包括很多其他的途径，例如 CDN、 Gzip、多域名、无 Cookie服务器等等

前端性能优化（二）

一、什么是前端性能优化（what）？

从用户访问资源到资源完整的展现在用户面前的过程中，通过技术手段和优化策略，缩短每个步骤的处理时间从而提升整个资源的访问和呈现速度。

二、为什么要做前端性能优化（why）？

在构建web站点的过程中，任何一个细节都有可能影响网站的访问速度，如果

不了解性能优化知识，很多不利网站访问速度的因素会形成累加，从而严重影响网站的性能，导致网站访问速度变慢，用户体验低下，最终导致用户流失。

三、前端性能优化的原则（rule）

1、不要按照准则照本宣科的做，需要根据实际情况因地制宜；

2、不出bug!

四、从浏览器发起请求到页面能正常浏览都有哪些阶段（process）？

预处理——>DNS解析——>建立连接——>发起请求——>等待响应——>接受数据——>处理元素——>布局渲染

五、性能优化的具体方法（way）

一）内容层面

1、DNS解析优化（DNS缓存、减少DNS查找、keep-alive、适当的主机域名）

2、避免重定向（/还是需要的）

3、切分到多个域名

4、杜绝404

二）网络传输阶段

1、减少传输过程中实体的大小

1）缓存

2）cookie优化

3）文件压缩（Accept-Encoding：g-zip）

2、减少请求的次数

1）文件适当的合并

2）雪碧图

3、异步加载（并发,requirejs）

4、预加载、延后加载、按需加载

三）渲染阶段

1、js放底部，css放顶部

2、减少重绘和回流

3、合理使用Viewport 等meta头部

4、减少dom节点

5、BigPipe

四）脚本执行阶段

1、缓存节点，尽量减少节点的查找

2、减少节点的操作（innerHTML）

3、避免无谓的循环，break、continue、return的适当使用

4、事件委托

六、与性能优化相关的细节的探索

1、缓存

1）Expires Cache-Control Last-Modified ETag If-Modified-Since If-None-Match 这些请求头部在浏览器缓存中分别起什么作用，如何起到缓存的作用?

1.当某一文件在浏览器中第一次被访问的时候，这个时候浏览器是没有缓存的，直接从服务器获取文件，返回给客户端，并且存入浏览器缓存；此时，返回状态码200，并且服务端可以设置响应头部Expires或者Cache-Control，Last-Modified或者ETag。

2.如果设置了Expires或者Cache-Control，那么在指定时间内再次请求该文件时，只要不强制刷新缓存(F5等)，浏览器会直接读取缓存而不再去请求服务器。

3.如果没有设置Expires或者Cache-Control或者过期了，就需要再次请求服务器了，浏览器会发起条件验证，发起请求时在请求头加上If-Modified-Sinse或者If-None-Match，服务器端判断最新的文件是否发生了更新，如果没有，总则返回响应状态码304，并且不带任何响应实体，也就是说，传输到客户端的只有一些相应头部，响应实体是空的，这样就大大减少了传输的体积，浏览器接受到了304响应，就知道了要读取浏览器缓存了。

2）按回车、浏览器刷新按钮、F5、Ctr+F5的区别?

1.按回车，浏览器会判断是否有缓存，并且根据Expires或者Cache-Control判断缓存是否过期，如果没有，就不会发起请求，直接使用缓存。否则就需要像服务器发起请求再验证。

2.浏览器刷新按钮和F5效果相同，不管是否有Expires或者Cache-Control，都会强制去请求服务器，进行再验证，根据If-Modified-Sinse或者If-None-Match判断是否要返回304，如果是，浏览器就会继续使用缓存。

3.按Ctr+F5时，也是不管是否有Expires或者Cache-Control，都会强制去请求服务器，但是并不会进行再验证，服务器会直接把最新的内容返回给浏览器，压根就不考虑缓存的存在或者是否过期。

3）为什么用Last-Modified还不够，要用ETag实体标签验证?

1.有些文档会被周期性的重写，但实际包含的数据是一样的。（尽管内容没有变化，最后修改日期却会发生变化）

2.有些文档可能被修改了，但是修改并不重要，没必要更新缓存。

3.有些服务器无法准确判定页面的最后修改日期。

4.文档在毫秒级间隙发生变化（如实时监控），以秒为颗粒度的Last-Modified就不够用了。

4）post请求拉取大量数据的缓存策略?

场景：

post拉取一个超大的数据，比如通讯录等。

为了避免每次都要请求都要拉取超大数据，我们可以在第一次请求后，把这份超大数据本地存储起来，下一次时，如果判断本地数据没有失效，就直接使用本地数据，而不用服务端传递庞大数据了，这样就在一定程度上缩短了http传递数据的时间了。这里的要点就是判断数据是否失效的机制。流程图不太好画，就用伪码吧。

这里使用了nodejs作为中间过渡层，大概的流程如下：

A：客户端设置headers["cache-flag"]=1给nodejs;

B：nodejs返回大数据Data给客户端

C：nodejs返回heades["cache-md5"] ="xxxx"给客户端

D：客户端本地存储大数据Data

E：客户端本地存储headers.cache-md5的值xxxx

F：客户端设置headers["cache-flag"]=1，并且从本地存储中拿到xxxx设置headers["cache-md5"]="xxxx"给nodejs

G：nodejs返回headers["cache-target"]=1给客户端

H：nodejs返回headers["cache-md5"]="newxxxx"给客户端

I：客户端使用本地存储的Data

具体的解释：

1、客户端：如果要使用缓存机制，在发请求的时候，设置一个请求头headers["cahce-flag"]=1;

2、nodeJs:（每次还是会请求底层服务端拿到数据），判断请求头有没有cache-flag，如果没有直接把从底层服务端拿到的数据返回给客户端；如果有cache-flag标志，再判断有没有headers["cache-md5"]，如果没有，统一直接返回数据，如果有，则把nodejs拿到的数据打一个md5值newxxx"，并且和请求的headers.cache-md5的值xxx相比较，如不相等，则说明过期，直接返回大数据,同时设置响应头，headers["cache-md5"]="newxxx"；如果相等，说明没有过期，则把content-length设置为0，responseText设置为空，同时返回两个头部给客户端，headers["cache-target"] =1;headers["cache-md5"]="xxx"

3、客户端判断每次都会存储大数据data（如果有的话），也会每次存储nodejs返回的cache-md5，并且每次还会把这个cache-md5传递给nodejs，当客户端判断有header["cahce-target"]，如果有，则说明缓存没有失效，则使用本地缓存。

2、DNS解析过程

先说一下DNS的几个基本概念：

一． 根域

就是所谓的“.”，其实我们的网址www.baidu.com在配置当中应该是www.baidu.com.（最后有一点），一般我们在浏览器里输入时会省略后面的点，而这也已经成为了习惯。

根域服务器我们知道有13台，但是这是错误的观点。

根域服务器只是具有13个IP地址，但机器数量却不是13台，因为这些IP地址借助了任播的技术，所以我们可以在全球设立这些IP的镜像站点，你访问到

的这个IP并不是唯一的那台主机。

具体的镜像分布可以参考维基百科。这些主机的内容都是一样的

二． 域的划分

根域下来就是顶级域或者叫一级域，

有两种划分方式，一种互联网刚兴起时的按照行业性质划分的com.，net.等，一种是按国家划分的如cn.，jp.，等。

具体多少你可以自己去查，我们这里不关心。

每个域都会有域名服务器，也叫权威域名服务器。

Baidu.com就是一个顶级域名，而www.baidu.com却不是顶级域名，他是在baidu.com 这个域里的一叫做www的主机。

一级域之后还有二级域，三级域，只要我买了一个顶级域，并且我搭建了自己BIND服务器（或者其他软件搭建的）注册到互联网中，那么我就可以随意在前面多加几个域了（当然长度是有限制的）。

比如a.www.baidu.com，在这个网址中，www.baidu.com变成了一个二级域而不是一台主机，主机名是a。

三． 域名服务器

能提供域名解析的服务器，上面的记录类型可以是A(address)记录，NS记录（name server），MX（mail），CNAME等。

（详解参见博客：域名解析中A记录、CNAME、MX记录、NS记录的区别和联系）

A记录是什么意思呢，就是记录一个IP地址和一个主机名字，比如我这个域名服务器所在的域test.baidu.com，我们知道这是一个二级的域名，然后我在里面有一条A记录,记录了主机为a的IP，查到了就返回给你了。

如果我现在要想baidu.com这个域名服务器查询a.test.baidu.com，那么这个顶级域名服务器就会发现你请求的这个网址在test.baidu.com这个域中，我这里记录了这个二级域的域名服务器test.baidu.com的NS的IP。我返回给你这个地址你再去查主机为a的主机把。

这些域内的域名服务器都称为权威服务器，直接提供DNS查询服务。（这些服务器可不会做递归哦）

四．解析过程

那么我们的DNS是怎么解析一个域名的呢?

1.现在我有一台计算机，通过ISP接入了互联网，那么ISP就会给我分配一个DNS服务器，这个DNS服务器不是权威服务器，而是相当于一个代理的dns解析服务器，他会帮你迭代权威服务器返回的应答，然后把最终查到IP返回给你。

2.现在的我计算机要向这台ISPDNS发起请求查询www.baidu.com这个域名了，(经网友提醒：这里其实准确来说不是ISPDNS，而应该是用户自己电脑网络设置里的DNS，并不一定是ISPDNS。比如也有可能你手工设置了8.8.8.8)

3.ISPDNS拿到请求后，先检查一下自己的缓存中有没有这个地址，有的话就

直接返回。这个时候拿到的ip地址，会被标记为非权威服务器的应答。

4.如果缓存中没有的话，ISPDNS会从配置文件里面读取13个根域名服务器的地址（这些地址是不变的，直接在BIND的配置文件中），

5.然后像其中一台发起请求。

6.根服务器拿到这个请求后，知道他是com.这个顶级域名下的，所以就会返回com域中的NS记录，一般来说是13台主机名和IP。

7.然后ISPDNS向其中一台再次发起请求，com域的服务器发现你这请求是baidu.com这个域的，我一查发现了这个域的NS，那我就返回给你，你再去查。

（目前百度有4台baidu.com的顶级域名服务器）。

8.ISPDNS不厌其烦的再次向baidu.com这个域的权威服务器发起请求，baidu.com收到之后，查了下有www的这台主机，就把这个IP返回给你了，

9.然后ISPDNS拿到了之后，将其返回给了客户端，并且把这个保存在高速缓存中。

下面我们来用 nslookup 这个工具详细来说一下解析步骤：

```
[root@Ceazw crazw]# nslookup www.jsjzx.com
Server:         210.32.32.1
Address:        210.32.32.1#53

Non-authoritative answer:
Name:   www.jsjzx.com
Address: 112.121.162.168
```

从上图我们可以看到:

第一行Server是：DNS服务器的主机名--210.32.32.1

第二行Address是： 它的IP地址--210.32.32.1#53

下面的Name是：解析的URL-- www.jsjzx.com

Address是：解析出来的IP--112.121.162.168

但是也有像百度这样的DNS比较复杂的解析:

```
[root@Ceazw crazw]# nslookup www.baidu.com
Server:         210.32.32.1
Address:        210.32.32.1#53

Non-authoritative answer:
www.baidu.com   canonical name = www.a.shifen.com.
Name:   www.a.shifen.com
Address: 61.135.169.105
Name:   www.a.shifen.com
Address: 61.135.169.125
```

你会发现百度有一个cname = www.a.shifen.com 的别名。

这是怎么一个过程呢?

我们用dig工具来跟踪一下把（linux系统自带有）

Dig工具会在本地计算机做迭代，然后记录查询的过程。

```
[root@Ceazw crazw]# dig +trace www.baidu.com

; <<>> DiG 9.8.2rc1-RedHat-9.8.2-0.17.rc1.el6_4.4 <<>> +trace www.baidu.com
;; global options: +cmd
```

第一步是向我这台机器的ISPDNS获取到根域服务区的13个IP和主机名[b-j].root-servers.net.。

```
.                       518400  IN      NS      a.root-servers.net.
.                       518400  IN      NS      j.root-servers.net.
.                       518400  IN      NS      f.root-servers.net.
.                       518400  IN      NS      c.root-servers.net.
.                       518400  IN      NS      e.root-servers.net.
.                       518400  IN      NS      l.root-servers.net.
.                       518400  IN      NS      h.root-servers.net.
.                       518400  IN      NS      k.root-servers.net.
.                       518400  IN      NS      m.root-servers.net.
.                       518400  IN      NS      g.root-servers.net.
.                       518400  IN      NS      d.root-servers.net.
.                       518400  IN      NS      b.root-servers.net.
.                       518400  IN      NS      i.root-servers.net.
;; Received 512 bytes from 210.32.32.1#53(210.32.32.1) in 25 ms
```

第二步是向其中的一台根域服务器（Servername就是末行小括号里面的）发送www.baidu.com的查询请求，他返回了com.顶级域的服务器IP（未显示）和名称，

```
com.                    172800  IN      NS      f.gtld-servers.net.
com.                    172800  IN      NS      k.gtld-servers.net.
com.                    172800  IN      NS      l.gtld-servers.net.
com.                    172800  IN      NS      b.gtld-servers.net.
com.                    172800  IN      NS      a.gtld-servers.net.
com.                    172800  IN      NS      g.gtld-servers.net.
com.                    172800  IN      NS      m.gtld-servers.net.
com.                    172800  IN      NS      c.gtld-servers.net.
com.                    172800  IN      NS      i.gtld-servers.net.
com.                    172800  IN      NS      d.gtld-servers.net.
com.                    172800  IN      NS      j.gtld-servers.net.
com.                    172800  IN      NS      e.gtld-servers.net.
com.                    172800  IN      NS      h.gtld-servers.net.
;; Received 503 bytes from 192.33.4.12#53(192.33.4.12) in 233 ms
```

第三步，便向com.域的一台服务器192.33.4.12请求,www.baidu.com，他返回了baidu.com域的服务器IP（未显示）和名称，百度有四台顶级域的服务器

【此处可以用dig @192.33.4.12 www.baidu.com查看返回的百度顶级域名服务器IP地址】。

```
baidu.com.              172800  IN      NS      dns.baidu.com.
baidu.com.              172800  IN      NS      ns2.baidu.com.
baidu.com.              172800  IN      NS      ns3.baidu.com.
baidu.com.              172800  IN      NS      ns4.baidu.com.
;; Received 167 bytes from 192.5.6.30#53(192.5.6.30) in 499 ms
```

第四步呢，向百度的顶级域服务器（202.108.22.220）请求www.baidu.com，他发现这个www有个别名，而不是一台主机，别名是www.a.shifen.com。

```
www.baidu.com.          1200    IN      CNAME   www.a.shifen.com.
a.shifen.com.           1200    IN      NS      ns2.a.shifen.com.
a.shifen.com.           1200    IN      NS      ns4.a.shifen.com.
a.shifen.com.           1200    IN      NS      ns3.a.shifen.com.
a.shifen.com.           1200    IN      NS      ns1.a.shifen.com.
;; Received 194 bytes from 202.108.22.220#53(202.108.22.220) in 28 ms
```

按照一般的逻辑，当dns请求到别名的时候，查询会终止，而是重新发起查询别名的请求，所以此处应该返回的是www.a.shifen.com而已。

但是为什么返回a.shifen.com的这个域的NS呢?

我们可以尝试下面的这个命令：dig +trace shifen.com 看看有什么结果。。。。。。。。

```
;; global options: +cmd
.                       518400  IN      NS      b.root-servers.net.
.                       518400  IN      NS      m.root-servers.net.
.                       518400  IN      NS      a.root-servers.net.
.                       518400  IN      NS      h.root-servers.net.
.                       518400  IN      NS      f.root-servers.net.
.                       518400  IN      NS      d.root-servers.net.
.                       518400  IN      NS      l.root-servers.net.
.                       518400  IN      NS      i.root-servers.net.
.                       518400  IN      NS      g.root-servers.net.
.                       518400  IN      NS      k.root-servers.net.
.                       518400  IN      NS      e.root-servers.net.
.                       518400  IN      NS      j.root-servers.net.
.                       518400  IN      NS      c.root-servers.net.
;; Received 512 bytes from 210.32.32.1#53(210.32.32.1) in 22 ms

com.                    172800  IN      NS      g.gtld-servers.net.
com.                    172800  IN      NS      f.gtld-servers.net.
com.                    172800  IN      NS      i.gtld-servers.net.
com.                    172800  IN      NS      l.gtld-servers.net.
com.                    172800  IN      NS      c.gtld-servers.net.
com.                    172800  IN      NS      h.gtld-servers.net.
com.                    172800  IN      NS      m.gtld-servers.net.
com.                    172800  IN      NS      e.gtld-servers.net.
com.                    172800  IN      NS      a.gtld-servers.net.
com.                    172800  IN      NS      j.gtld-servers.net.
com.                    172800  IN      NS      b.gtld-servers.net.
com.                    172800  IN      NS      d.gtld-servers.net.
com.                    172800  IN      NS      k.gtld-servers.net.
;; Received 492 bytes from 192.228.79.201#53(192.228.79.201) in 173 ms

shifen.com.             172800  IN      NS      dns.baidu.com.
shifen.com.             172800  IN      NS      ns2.baidu.com.
shifen.com.             172800  IN      NS      ns3.baidu.com.
shifen.com.             172800  IN      NS      ns4.baidu.com.
;; Received 174 bytes from 192.43.172.30#53(192.43.172.30) in 371 ms

www.shifen.com.         300     IN      CNAME   bar.n.shifen.com.
n.shifen.com.           86400   IN      NS      ns3.n.shifen.com.
n.shifen.com.           86400   IN      NS      ns4.n.shifen.com.
n.shifen.com.           86400   IN      NS      ns1.n.shifen.com.
n.shifen.com.           86400   IN      NS      ns2.n.shifen.com.
;; Received 188 bytes from 202.108.22.220#53(202.108.22.220) in 28 ms
```

你会发现第三步时shifen.com这个顶级域的域名服务器和baidu.com这个域的域名服务器是同一台主机（即：dns.baidu.com）！

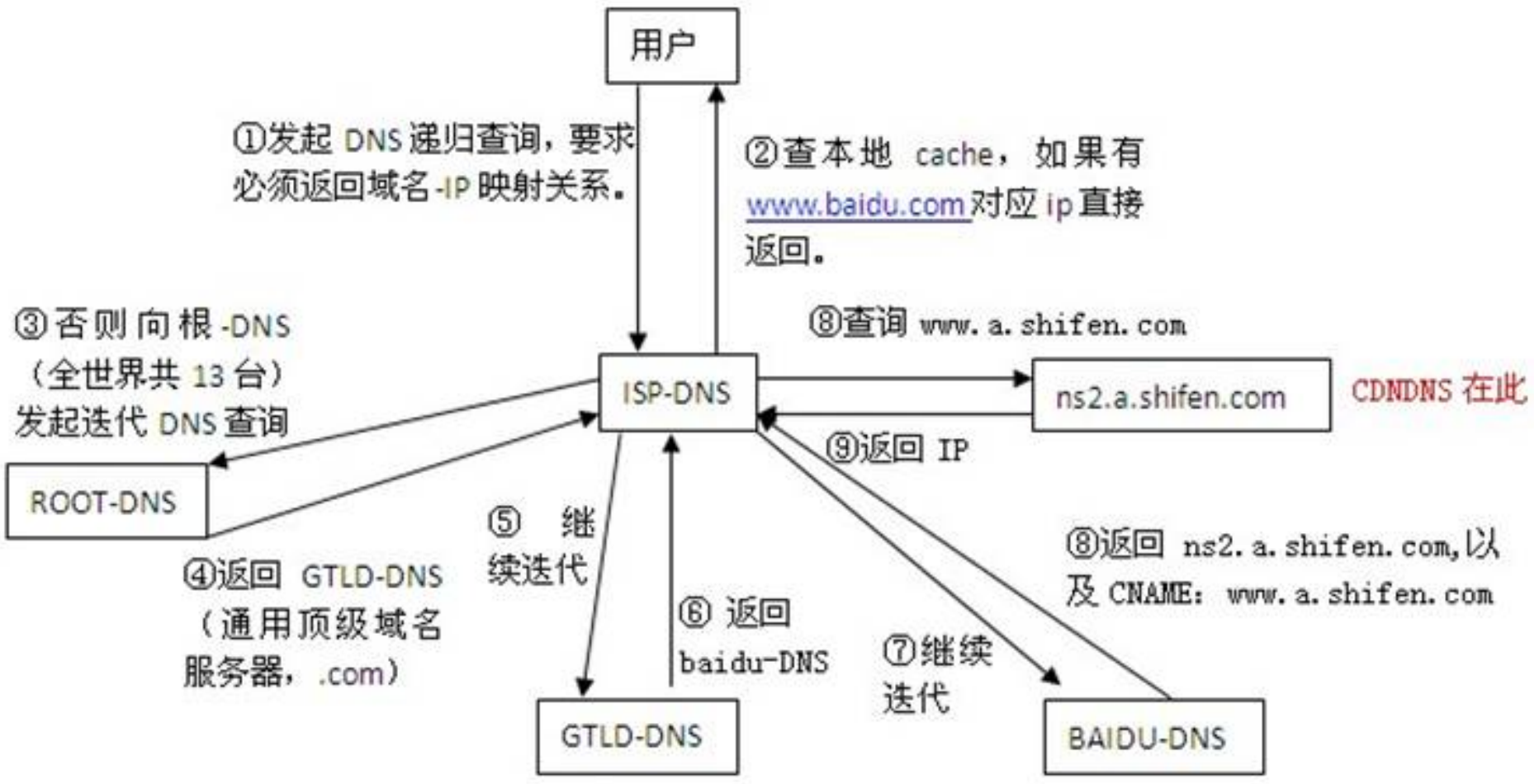
当我拿到www.baidu.com的别名www.a.shifen.com的时候，我本来需要重新到com域查找shifen.com域的NS，但是因为这两个域在同一台NS上，所以直接向本机发起了，

shifen.com域发现请求的www.a.shifen.com是属于a.shifen.com这个域的，

于是就把a.shifen.com的这个NS和IP返回，让我到a.shifen.com这个域的域名服务器上查询www.a.shifen.com。

于是我便从ns X .a.shifen.com中一台拿到了一条A记录，最终的最终也便是www.baidu.com的IP地址了.【此处也可以用dig +trace www.a.shifen.com】跟踪一下

用一个图来说明一下(图中第三步的全世界只有13台是错误的)



以下内容为在虚拟机中搭建local dns服务器得到的实验数据，纠正上述结论

在上面的分析中，我们用dig工具进行了追踪，但是dig没有继续追踪当我们从baidu.com拿到cname和ns2.a.shifen.com的IP之后的事情。

我们就所以然的下结论认为local dns会向ns2.a.shifen.com请求www.a.shifenc.om。

其实这个想法是错误，在自己的本地搭建一个local dns，抓取整个解析过程中是所有包，看看就明白拉。

实际的结果是虽然dns.baidu.com返回了a.shifen.com域的服务器地址和IP，

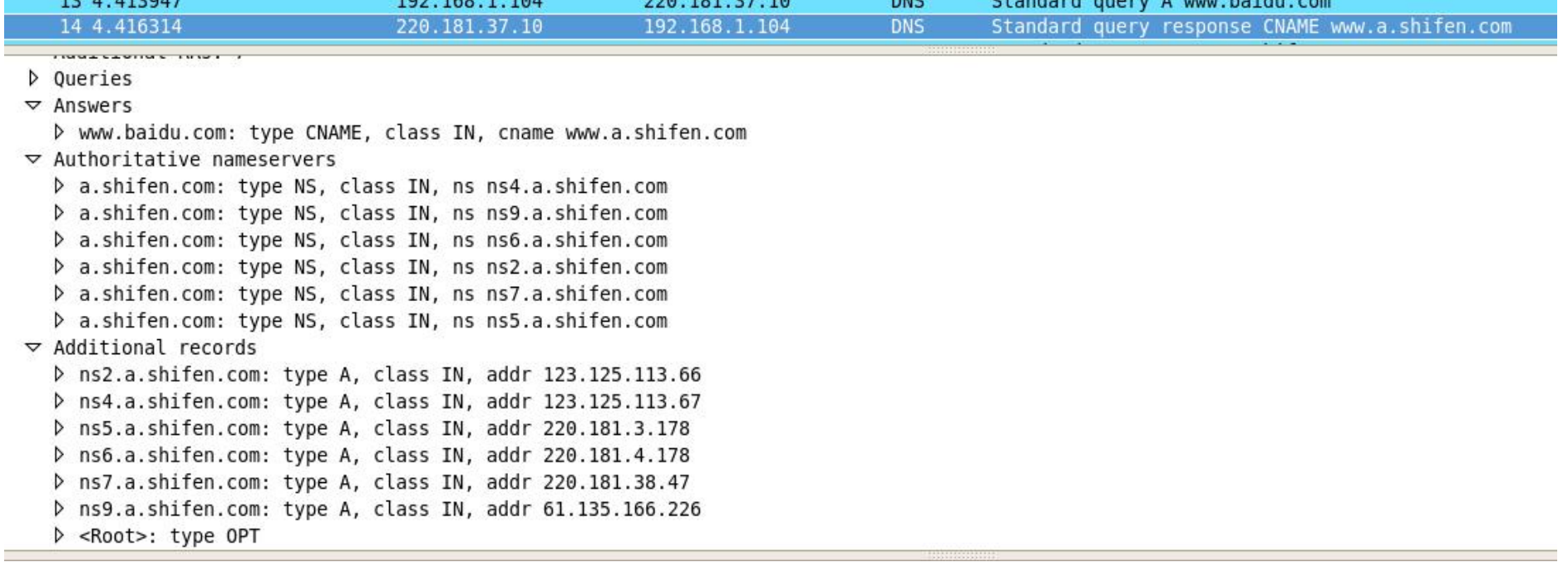
但是local dns并不是直接向上述返回的IP请求www.a.shifen.com，而是再一次去请求com域，得到shifen.com域的服务器（也就是baidu.com的那四台），

然后又请求www.a.shifen.com，返回a.shifen.com的域的服务器，最后才是去请求www.a.shifen.com，

虽然上面已经返回了IP，但是实验的结果就是再走一遍shifen.com域的查询。

Source	Destination	Protocol	Info
192.168.1.103	192.168.1.104	DNS	Standard query A www.baidu.com
192.168.1.104	192.228.79.201	DNS	Standard query A www.baidu.com
192.168.1.104	192.33.4.12	DNS	Standard query NS <Root>
192.228.79.201	192.168.1.104	DNS	Standard query response
192.168.1.104	192.31.80.30	DNS	Standard query A www.baidu.com
192.168.1.104	198.41.0.4	DNS	Standard query NS <Root>
192.31.80.30	192.168.1.104	DNS	Standard query response
192.168.1.104	220.181.37.10	DNS	Standard query A www.baidu.com
220.181.37.10	192.168.1.104	DNS	Standard query response CNAME www.a.shifen.com
192.168.1.104	192.48.79.30	DNS	Standard query A www.a.shifen.com
198.41.0.4	192.168.1.104	DNS	Standard query response NS a.root-servers.net NS c.root-servers.net NS m.root-
192.48.79.30	192.168.1.104	DNS	Standard query response
192.168.1.104	61.135.165.235	DNS	Standard query A www.a.shifen.com
61.135.165.235	192.168.1.104	DNS	Standard query response
192.168.1.104	220.181.38.47	DNS	Standard query A www.a.shifen.com
220.181.38.47	192.168.1.104	DNS	Standard query response A 119.75.218.77 A 119.75.217.56
192.168.1.104	192.168.1.103	DNS	Standard query response CNAME www.a.shifen.com A 119.75.218.77 A 119.75.217.56

上图就是localdns在解析www.baidu.com的抓包全过程。蓝色那条就是在收到cname和响应的a.shifen.com的域名服务器IP地址之后，继续向com域请求shifen.com。



这个图充分说明了返回cname的同时也返回了ns2.a.shifen.com的IP。

因此总结一下便是

①本机向local dns请求www.baidu.com

②local dns向根域请求www.baidu.com，根域返回com.域的服务器IP

③向com.域请求www.baidu.com，com.域返回baidu.com域的服务器IP

④向baidu.com请求www.baidu.com，返回cname www.a.shifen.com和a.shifen.com域的服务器IP

⑤向root域请求www.a.shifen.com

⑥向com.域请求www.a.shife.com

⑦向shifen.com请求

⑧向a.shifen.com域请求

⑨拿到www.a.shifen.com的IP

⑩localdns返回本机www.baidu.com cname www.a.shifen.com 以及www.a.shifen.com的IP

3、HTTP

1）所有常用状态码的含义?

1xx消息

这一类型的状态码，代表请求已被接受，需要继续处理。这类响应是临时响应，只包含状态行和某些可选的响应头信息，并以空行结束。由于HTTP/1.0协议中没有定义任何1xx状态码，所以除非在某些试验条件下，服务器禁止向此类客户端发送1xx响应。 这些状态码代表的响应都是信息性的，标示客户应该采取的其他行动。

100 Continue

客户端应当继续发送请求。这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应。

101 Switching Protocols

服务器已经理解了客户端的请求，并将通过Upgrade消息头通知客户端采用不同的协议来完成这个请求。在发送完这个响应最后的空行后，服务器将会切换到在Upgrade消息头中定义的那些协议。
只有在切换新的协议更有好处的时候才应该采取类似措施。例如，切换到新的HTTP版本比旧版本更有优势，或者切换到一个实时且同步的协议以传送利用此类特性的资源。

102 Processing

由WebDAV（RFC 2518）扩展的状态码，代表处理将被继续执行。

2xx成功

这一类型的状态码，代表请求已成功被服务器接收、理解、并接受。

200 OK

请求已成功，请求所希望的响应头或数据体将随此响应返回。

201 Created

请求已经被实现，而且有一个新的资源已经依据请求的需要而创建，且其URI已经随Location头信息返回。假如需要的资源无法及时创建的话，应当返回’202 Accepted’。

202 Accepted

服务器已接受请求，但尚未处理。正如它可能被拒绝一样，最终该请求可能会也可能不会被执行。在异步操作的场合下，没有比发送这个状态码更方便的做法了。
返回202状态码的响应的目的是允许服务器接受其他过程的请求（例如某个每天只执行一次的基于批处理的操作），而不必让客户端一直保持与服务器的连接直到批处理操作全部完成。在接受请求处理并返回202状态码的响应应当在返回的实体中包含一些指示处理当前状态的信息，以及指向处理状态监视器或状态预测的指针，以便用户能够估计操作是否已经完成。

203 Non-Authoritative Information

服务器已成功处理了请求，但返回的实体头部元信息不是在原始服务器上有效的确定集合，而是来自本地或者第三方的拷贝。当前的信息可能是原始版本的子集或者超集。例如，包含资源的元数据可能导致原始服务器知道元信息的超集。使用此状态码不是必须的，而且只有在响应不使用此状态码便会返回200 OK的情况下才是合适的。

204 No Content

服务器成功处理了请求，但不需要返回任何实体内容，并且希望返回更新了的元信息。响应可能通过实体头部的形式，返回新的或更新后的元信息。如果存在这些头部信息，则应当与所请求的变量相呼应。
如果客户端是浏览器的话，那么用户浏览器应保留发送了该请求的页面，而不产生任何文档视图上的变化，即使按照规范新的或更新后的元信息应当被应用到用户浏览器活动视图中的文档。
由于204响应被禁止包含任何消息体，因此它始终以消息头后的第一个空行结尾。

205 Reset Content

服务器成功处理了请求，且没有返回任何内容。但是与204响应不同，返回此状态码的响应要求请求者重置文档视图。该响应主要是被用于接受用户输入后，立即重置表单，以便用户能够轻松地开始另一次输入。
与204响应一样，该响应也被禁止包含任何消息体，且以消息头后的第一个空行结束。

206 Partial Content

服务器已经成功处理了部分GET请求。类似于FlashGet或者迅雷这类的HTTP下载工具都是使用此类响应实现断点续传或者将一个大文档分解为多个下载

段同时下载。
该请求必须包含Range头信息来指示客户端希望得到的内容范围，并且可能包含If-Range来作为请求条件。
响应必须包含如下的头部域：
Content-Range用以指示本次响应中返回的内容的范围；如果是Content-Type为multipart/byteranges的多段下载，则每一multipart段中都应包含Content-Range域用以指示本段的内容范围。假如响应中包含Content-Length，那么它的数值必须匹配它返回的内容范围的真实字节数。
Date
ETag和 / 或Content-Location，假如同样的请求本应该返回200响应。
Expires, Cache-Control，和 / 或Vary，假如其值可能与之前相同变量的其他响应对应的值不同的话。
假如本响应请求使用了If-Range强缓存验证，那么本次响应不应该包含其他实体头；假如本响应的请求使用了If-Range弱缓存验证，那么本次响应禁止包含其他实体头；这避免了缓存的实体内容和更新了的实体头信息之间的不一致。否则，本响应就应当包含所有本应该返回200响应中应当返回的所有实体头部域。
假如ETag或Last-Modified头部不能精确匹配的话，则客户端缓存应禁止将206响应返回的内容与之前任何缓存过的内容组合在一起。
任何不支持Range以及Content-Range头的缓存都禁止缓存206响应返回的内容。

207 Multi-Status

由WebDAV(RFC 2518)扩展的状态码，代表之后的消息体将是一个XML消息，并且可能依照之前子请求数量的不同，包含一系列独立的响应代码。

3xx重定向

这类状态码代表需要客户端采取进一步的操作才能完成请求。通常，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的Location域中指明。
当且仅当后续的请求所使用的方法是GET或者HEAD时，用户浏览器才可以在没有用户介入的情况下自动提交所需要的后续请求。客户端应当自动监测无限循环重定向（例如：A→B→C→……→A或A→A），因为这会导致服务器和客户端大量不必要的资源消耗。按照HTTP/1.0版规范的建议，浏览器不应自动访问超过5次的重定向。

300 Multiple Choices

被请求的资源有一系列可供选择的回馈信息，每个都有自己特定的地址和浏览器驱动的商议信息。用户或浏览器能够自行选择一个首选的地址进行重定向。
除非这是一个HEAD请求，否则该响应应当包括一个资源特性及地址的列表的实体，以便用户或浏览器从中选择最合适的重定向地址。这个实体的格式由Content-Type定义的格式所决定。浏览器可能根据响应的格式以及浏览器自身能力，自动作出最合适的选择。当然，RFC 2616规范并没有规定这样的自动选择该如何进行。
如果服务器本身已经有了首选的回馈选择，那么在Location中应当指明这个回馈的URI；浏览器可能会将这个Location值作为自动重定向的地址。此外，除非额外指定，否则这个响应也是可缓存的。

301 Moved Permanently

被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个URI之一。如果可能，拥有链接编辑功能的客户端应当自动把请求的地址修改为从服务器反馈回来的地址。除非额外指定，否则这

个响应也是可缓存的。
新的永久性的URI应当在响应的Location域中返回。除非这是一个HEAD请求，否则响应的实体中应当包含指向新的URI的超链接及简短说明。
如果这不是一个GET或者HEAD请求，因此浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。
注意：对于某些使用HTTP/1.0协议的浏览器，当它们发送的POST请求得到了一个301响应的话，接下来的重定向请求将会变成GET方式。

302 Found

请求的资源现在临时从不同的URI响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的。
新的临时性的URI应当在响应的Location域中返回。除非这是一个HEAD请求，否则响应的实体中应当包含指向新的URI的超链接及简短说明。
如果这不是一个GET或者HEAD请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。
注意：虽然RFC 1945和RFC 2068规范不允许客户端在重定向时改变请求的方法，但是很多现存的浏览器将302响应视作为303响应，并且使用GET方式访问在Location中规定的URI，而无视原先请求的方法。状态码303和307被添加了进来，用以明确服务器期待客户端进行何种反应。

303 See Other

对应当前请求的响应可以在另一个URI上被找到，而且客户端应当采用GET的方式访问那个资源。这个方法的存在主要是为了允许由脚本激活的POST请求输出重定向到一个新的资源。这个新的URI不是原始资源的替代引用。同时，303响应禁止被缓存。当然，第二个请求（重定向）可能被缓存。
新的URI应当在响应的Location域中返回。除非这是一个HEAD请求，否则响应的实体中应当包含指向新的URI的超链接及简短说明。
注意：许多HTTP/1.1版以前的浏览器不能正确理解303状态。如果需要考虑与这些浏览器之间的互动，302状态码应该可以胜任，因为大多数的浏览器处理302响应时的方式恰恰就是上述规范要求客户端处理303响应时应当做的。

304 Not Modified

如果客户端发送了一个带条件的GET请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变，则服务器应当返回这个状态码。304响应禁止包含消息体，因此始终以消息头后的第一个空行结尾。
该响应必须包含以下的头信息：
Date，除非这个服务器没有时钟。假如没有时钟的服务器也遵守这些规则，那么代理服务器以及客户端可以自行将Date字段添加到接收到的响应头中去（正如RFC 2068中规定的一样），缓存机制将会正常工作。
ETag和 / 或Content-Location，假如同样的请求本应返回200响应。
Expires, Cache-Control，和 / 或Vary，假如其值可能与之前相同变量的其他响应对应的值不同的话。
假如本响应请求使用了强缓存验证，那么本次响应不应该包含其他实体头；否则（例如，某个带条件的GET请求使用了弱缓存验证），本次响应禁止包含其他实体头；这避免了缓存了的实体内容和更新了的实体头信息之间的不一致。
假如某个304响应指明了当前某个实体没有缓存，那么缓存系统必须忽视这个响应，并且重复发送不包含限制条件的请求。
假如接收到一个要求更新某个缓存条目的304响应，那么缓存系统必须更新整个条目以反映所有在响应中被更新的字段的值。

305 Use Proxy

被请求的资源必须通过指定的代理才能被访问。Location域中将给出指定的代理所在的URI信息，接收者需要重复发送一个单独的请求，通过这个代理才能访问相应资源。只有原始服务器才能创建305响应。
注意：RFC 2068中没有明确305响应是为了重定向一个单独的请求，而且只能被原始服务器创建。忽视这些限制可能导致严重的安全后果。

306 Switch Proxy

在最新版的规范中，306状态码已经不再被使用。

307 Temporary Redirect

请求的资源现在临时从不同的URI响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的。
新的临时性的URI应当在响应的Location域中返回。除非这是一个HEAD请求，否则响应的实体中应当包含指向新的URI的超链接及简短说明。因为部分浏览器不能识别307响应，因此需要添加上述必要信息以便用户能够理解并向新的URI发出访问请求。
如果这不是一个GET或者HEAD请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。

4xx客户端错误

这类的状态码代表了客户端看起来可能发生了错误，妨碍了服务器的处理。除非响应的是一个HEAD请求，否则服务器就应该返回一个解释当前错误状况的实体，以及这是临时的还是永久性的状况。这些状态码适用于任何请求方法。浏览器应当向用户显示任何包含在此类错误响应中的实体内容。
如果错误发生时客户端正在传送数据，那么使用TCP的服务器实现应当仔细确保在关闭客户端与服务器之间的连接之前，客户端已经收到了包含错误信息的数据包。如果客户端在收到错误信息后继续向服务器发送数据，服务器的TCP栈将向客户端发送一个重置数据包，以清除该客户端所有还未识别的输入缓冲，以免这些数据被服务器上的应用程序读取并干扰后者。

400 Bad Request

由于包含语法错误，当前请求无法被服务器理解。除非进行修改，否则客户端不应该重复提交这个请求。

401 Unauthorized

当前请求需要用户验证。该响应必须包含一个适用于被请求资源的WWW-Authenticate信息头用以询问用户信息。客户端可以重复提交一个包含恰当的Authorization头信息的请求。如果当前请求已经包含了Authorization证书，那么401响应代表着服务器验证已经拒绝了那些证书。如果401响应包含了与前一个响应相同的身份验证询问，且浏览器已经至少尝试了一次验证，那么浏览器应当向用户展示响应中包含的实体信息，因为这个实体信息中可能包含了相关诊断信息。参见RFC 2617。

402 Payment Required

该状态码是为了将来可能的需求而预留的。

403 Forbidden

服务器已经理解请求，但是拒绝执行它。与401响应不同的是，身份验证并不能提供任何帮助，而且这个请求也不应该被重复提交。如果这不是一个HEAD请求，而且服务器希望能够讲清楚为何请求不能被执行，那么就应该在实体内描述拒绝的原因。当然服务器也可以返回一个404响应，假如它不希望让客户端获得任何信息。

404 Not Found

请求失败，请求所希望得到的资源未被在服务器上发现。没有信息能够告诉用户这个状况到底是暂时的还是永久的。假如服务器知道情况的话，应当使用410状态码来告知旧资源因为某些内部的配置机制问题，已经永久的不可用，而且没有任何可以跳转的地址。404这个状态码被广泛应用于当服务器不想揭示到底为何请求被拒绝或者没有其他适合的响应可用的情况下。

405 Method Not Allowed

请求行中指定的请求方法不能被用于请求相应的资源。该响应必须返回一个Allow头信息用以表示出当前资源能够接受的请求方法的列表。
鉴于PUT，DELETE方法会对服务器上的资源进行写操作，因而绝大部分的网页服务器都不支持或者在默认配置下不允许上述请求方法，对于此类请求均会返回405错误。

406 Not Acceptable

请求的资源的内容特性无法满足请求头中的条件，因而无法生成响应实体。
除非这是一个HEAD请求，否则该响应就应当返回一个包含可以让用户或者浏览器从中选择最合适的实体特性以及地址列表的实体。实体的格式由Content-Type头中定义的媒体类型决定。浏览器可以根据格式及自身能力自行作出最佳选择。但是，规范中并没有定义任何作出此类自动选择的标准。

407 Proxy Authentication Required

与401响应类似，只不过客户端必须在代理服务器上进行身份验证。代理服务器必须返回一个Proxy-Authenticate用以进行身份询问。客户端可以返回一个Proxy-Authorization信息头用以验证。参见RFC 2617。

408 Request Timeout

请求超时。客户端没有在服务器预备等待的时间内完成一个请求的发送。客户端可以随时再次提交这一请求而无需进行任何更改。

409 Conflict

由于和被请求的资源的当前状态之间存在冲突，请求无法完成。这个代码只允许用在这样的情况下才能被使用：用户被认为能够解决冲突，并且会重新提交新的请求。该响应应当包含足够的信息以便用户发现冲突的源头。
冲突通常发生于对PUT请求的处理中。例如，在采用版本检查的环境下，某次PUT提交的对特定资源的修改请求所附带的版本信息与之前的某个（第三方）请求向冲突，那么此时服务器就应该返回一个409错误，告知用户请求无法完成。此时，响应实体中很可能会包含两个冲突版本之间的差异比较，以便用户重新提交归并以后的新版本。

410 Gone

被请求的资源在服务器上已经不再可用，而且没有任何已知的转发地址。这样的状况应当被认为是永久性的。如果可能，拥有链接编辑功能的客户端应当在获得用户许可后删除所有指向这个地址的引用。如果服务器不知道或者无法确定这个状况是否是永久的，那么就应该使用404状态码。除非额外说明，否则这个响应是可缓存的。
410响应的目的主要是帮助网站管理员维护网站，通知用户该资源已经不再可用，并且服务器拥有者希望所有指向这个资源的远端连接也被删除。这类事件在限时、增值服务中很普遍。同样，410响应也被用于通知客户端在当前服务器站点上，原本属于某个个人的资源已经不再可用。当然，是否需要把所有永久不可用的资源标记为’410 Gone’，以及是否需要保持此标记多长时间，完全取决于服务器拥有者。

411 Length Required

服务器拒绝在没有定义Content-Length头的情况下接受请求。在添加了表明请求消息体长度的有效Content-Length头之后，客户端可以再次提交该请求。

412 Precondition Failed

服务器在验证在请求的头字段中给出先决条件时，没能满足其中的一个或多个。这个状态码允许客户端在获取资源时在请求的元信息（请求头字段数据）中设置先决条件，以此避免该请求方法被应用到其希望的内容以外的资源上。

413 Request Entity Too Large

服务器拒绝处理当前请求，因为该请求提交的实体数据大小超过了服务器愿意或者能够处理的范围。此种情况下，服务器可以关闭连接以免客户端继续发送此请求。
如果这个状况是临时的，服务器应当返回一个Retry-After的响应头，以告知客户端可以在多少时间以后重新尝试。

414 Request-URI Too Long

请求的URI长度超过了服务器能够解释的长度，因此服务器拒绝对该请求提供服务。这比较少见，通常的情况包括：
本应使用POST方法的表单提交变成了GET方法，导致查询字符串（Query String）过长。
重定向URI“黑洞”，例如每次重定向把旧的URI作为新的URI的一部分，导致在若干次重定向后URI超长。
客户端正在尝试利用某些服务器中存在的安全漏洞攻击服务器。这类服务器使用固定长度的缓冲读取或操作请求的URI，当GET后的参数超过某个数值后，可能会产生缓冲区溢出，导致任意代码被执行[1]。没有此类漏洞的服务器，应当返回414状态码。

415 Unsupported Media Type

对于当前请求的方法和所请求的资源，请求中提交的实体并不是服务器中所支持的格式，因此请求被拒绝。

416 Requested Range Not Satisfiable

如果请求中包含了Range请求头，并且Range中指定的任何数据范围都与当前资源的可用范围不重合，同时请求中又没有定义If-Range请求头，那么服务器就应当返回416状态码。
假如Range使用的是字节范围，那么这种情况就是指请求指定的所有数据范围的首字节位置都超过了当前资源的长度。服务器也应当在返回416状态码的同时，包含一个Content-Range实体头，用以指明当前资源的长度。这个响应也被禁止使用multipart/byteranges作为其Content-Type。

417 Expectation Failed

在请求头Expect中指定的预期内容无法被服务器满足，或者这个服务器是一个代理服务器，它有明显的证据证明在当前路由的下一个节点上，Expect的内容无法被满足。

418 I’m a teapot

本操作码是在1998年作为IETF的传统愚人节笑话, 在RFC 2324 超文本咖啡壶控制协议中定义的，并不需要在真实的HTTP服务器中定义。

421 There are too many connections from your internet address

从当前客户端所在的IP地址到服务器的连接数超过了服务器许可的最大范围。通常，这里的IP地址指的是从服务器上看到的客户端地址（比如用户的

网关或者代理服务器地址）。在这种情况下，连接数的计算可能涉及到不止一个终端用户。

422 Unprocessable Entity

请求格式正确，但是由于含有语义错误，无法响应。（RFC 4918 WebDAV）

423 Locked

当前资源被锁定。（RFC 4918 WebDAV）

424 Failed Dependency

由于之前的某个请求发生的错误，导致当前请求失败，例如PROPPATCH。（RFC 4918 WebDAV）

425 Unordered Collection

在WebDav Advanced Collections草案中定义，但是未出现在《WebDAV顺序集协议》（RFC 3658）中。

426 Upgrade Required

客户端应当切换到TLS/1.0。（RFC 2817）

449 Retry With

由微软扩展，代表请求应当在执行完适当的操作后进行重试。

5xx服务器错误

这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。除非这是一个HEAD请求，否则服务器应当包含一个解释当前错误状态以及这个状况是临时的还是永久的解释信息实体。浏览器应当向用户展示任何在当前响应中被包含的实体。
这些状态码适用于任何响应方法。

500 Internal Server Error

服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在服务器的程序码出错时出现。

501 Not Implemented

服务器不支持当前请求所需要的某个功能。当服务器无法识别请求的方法，并且无法支持其对任何资源的请求。

502 Bad Gateway

作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。

503 Service Unavailable

由于临时的服务器维护或者过载，服务器当前无法处理请求。这个状况是临时的，并且将在一段时间以后恢复。如果能够预计延迟时间，那么响应中可以包含一个Retry-After头用以标明这个延迟时间。如果没有给出这个Retry-After信息，那么客户端应当以处理500响应的方式处理它。

504 Gateway Timeout

作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器（URI标识出的服务器，例如HTTP、FTP、LDAP）或者辅助服务器（例如DNS）收到响应。
注意：某些代理服务器在DNS查询超时时会返回400或者500错误。

505 HTTP Version Not Supported

服务器不支持，或者拒绝支持在请求中使用的HTTP版本。这暗示着服务器不能或不愿使用与客户端相同的版本。响应中应当包含一个描述了为何版本不被支持以及服务器支持哪些协议的实体。

506 Variant Also Negotiates

由《透明内容协商协议》（RFC 2295）扩展，代表服务器存在内部配置错误：被请求的协商变元资源被配置为在透明内容协商中使用自己，因此在一个协商处理中不是一个合适的重点。

507 Insufficient Storage

服务器无法存储完成请求所必须的内容。这个状况被认为是临时的。（WebDAV RFC 4918）

509 Bandwidth Limit Exceeded

服务器达到带宽限制。这不是一个官方的状态码，但是仍被广泛使用。

510 Not Extended

获取资源所需要的策略并没有被满足。（RFC 2774）

2）301跳转和302跳转的区别?

一直对http状态码301和302的理解比较模糊，在遇到实际的问题和翻阅各种资料了解后，算是有了一定的理解。这里记录下，希望能有新的认识。大家也共勉。

官方的比较简洁的说明：

301 redirect: 301 代表永久性转移(Permanently Moved)

302 redirect: 302 代表暂时性转移(Temporarily Moved)

ps:这里也顺带记住了两个比较相近的英语单词（permanently、temporarily），嘻哈！

详细来说，301和302状态码都表示重定向，就是说浏览器在拿到服务器返回的这个状态码后会自动跳转到一个新的URL地址，这个地址可以从响应的Location首部中获取（用户看到的效果就是他输入的地址A瞬间变成了另一个地址B）——这是它们的共同点。他们的不同在于。301表示旧地址A的资源已经被永久地移除了（这个资源不可访问了），搜索引擎在抓取新内容的同时也将旧的网址交换为重定向之后的网址；302表示旧地址A的资源还在（仍然可以访问），这个重定向只是临时地从旧地址A跳转到地址B，搜索引擎会抓取新的内容而保存旧的网址。

这里开启傻瓜自问自答模式（自己可能想到的疑问）：

1、什么是重定向啊?

就是地址A跳转到地址B啦。百度百科的解释：重定向(Redirect)就是通过各种方法将各种网络请求重新定个方向转到其它位置（如：网页重定向、域名的重定向、路由选择的变化也是对数据报文经由路径的一种重定向）。

2、可是，为什么要进行重定向啊？什么时候需要重定向呢？

想跳就跳，就跳的漂亮。还是借鉴百度百科：

1）网站调整（如改变网页目录结构）；
2）网页被移到一个新地址；
3）网页扩展名改变(如应用需要把.php改成.Html或.shtml)。

这种情况下，如果不做重定向，则用户收藏夹或搜索引擎数据库中旧地址只能让访问客户得到一个404页面错误信息，访问流量白白丧失；再者某些注册了多个域名的网站，也需要通过重定向让访问这些域名的用户自动跳转到主站点等。

3、那么，什么时候进行301或者302跳转呢？

当一个网站或者网页24—48小时内临时移动到一个新的位置，这时候就要进行302跳转，打个比方说，我有一套房子，但是最近走亲戚去亲戚家住了，过两天我还回来的。而使用301跳转的场景就是之前的网站因为某种原因需要移除掉，然后要到新的地址访问，是永久性的，就比如你的那套房子其实是租的，现在租期到了，你又在另一个地方找到了房子，之前租的房子不住了。

清晰明确而言：
使用301跳转的场景：
1）域名到期不想续费（或者发现了更适合网站的域名），想换个域名。
2）在搜索引擎的搜索结果中出现了不带www的域名，而带www的域名却没有收录，这个时候可以用301重定向来告诉搜索引擎我们目标的域名是哪一个。
3）空间服务器不稳定，换空间的时候。

使用302跳转的场景：
--尽量使用301跳转！

4、为什么尽量要使用301跳转？——网址劫持！

这里摘录百度百科上的解释：

从网址A 做一个302 重定向到网址B 时，主机服务器的隐含意思是网址A随时有可能改主意，重新显示本身的内容或转向其他的地方。大部分的搜索引擎在大部分情况下，当收到302 重定向时，一般只要去抓取目标网址就可以了，也就是说网址B。如果搜索引擎在遇到302 转向时，百分之百的都抓取目标网址B 的话，就不用担心网址URL 劫持了。问题就在于，有的时候搜索引擎，尤其是Google，并不能总是抓取目标网址。比如说，有的时候A 网址很短，但是它做了一个302 重定向到B 网址，而B 网址是一个很长的乱七八糟的URL 网址，甚至还有可能包含一些问号之类的参数。很自然的，A 网址更加用户友好，而B 网址既难看，又不用户友好。这时Google 很有可能会仍然显示网址A。由于搜索引擎排名算法只是程序而不是人，在遇到302 重定向的时候，并不能像人一样的去准确判定哪一个网址更适当，这就造成了网址URL 劫持的可能性。也就是说，一个不道德的人在他自己的网址A 做一个302 重定向到你的网址B，出于某种原因， Google 搜索结果所显示的仍然是网址A，但是所用的网页内容却是你的网址B 上的内容，这种情况就叫做网址URL 劫持。你辛辛苦苦所写的内容就这样被别人偷走了。302 重定向所造成的网址URL 劫持现象，已经存在一段时间了。不过到目前为止，似乎也没有什么更好的解决方法。在正在进行的谷歌大爸爸数据中心转换中，302 重定向问题也是要被解决的目标之一。从一些搜索结果来看，网址劫持现象有所改善，但是并没有完全解决。

我的理解是，从网站A（网站比较烂）上做了一个302跳转到网站B（搜索排名很靠前），这时候有时搜索引擎会使用网站B的内容，但却收录了网站A的地址，这样在不知不觉间，网站B在为网站A作贡献，网站A的排名就靠前

了。

301跳转对查找引擎是一种对照驯良的跳转编制，也是查找引擎能够遭遇的跳转编制，它告诉查找引擎，这个地址弃用了，永远转向一个新地址，可以转移新域名的权重。而302重定向很容易被搜索引擎误认为是利用多个域名指向同一网站，那么你的网站就会被封掉，罪名是“利用重复的内容来干扰Google搜索结果的网站排名”。

自问自答模式先告一段落，这里分享下我在NodeJs中实现跳转的场景：

之前做过一个重构的项目，由于各种原因，我们的网站的登录以及注册部分需要剥离为另一个网站，域名和之前的不同，所以，我们需要保证旧的地址也能重定向到地址中去，我们就在旧的系统的node层中作了一个重定向，代码类似这样：

```
132 router.get(/^(?:\/m)?\/u\/(\w+)(\/)?$/, function (req, res) {
133     var newUrl = "http://ap.hzins.com" + req.url;
134     res.redirect(newUrl);
135 });
```


这里没有设置状态码，发现默认是302跳转，然后我们设置了301状态码，类似这样：

```
132 router.get(/^(?:\/m)?\/u\/(\w+)(\/)?$/, function (req, res) {
133     var newUrl = "http://ap.hzins.com" + req.url;
134     res.redirect(301,newUrl);
```


用fiddle抓包(上面的302调整我就不上图了)，看到效果：

3	301	HTTP	test.[redacted]	/m/u/login	114
4	200	HTTP	[redacted]	/m/u/login	1,408

以上是使用Express，用nodejs原生的代码实现类似这样：

```
res.writeHead(301, {
    "Cache-Control": "max-age=0",
    "Content-Type": "text/html;charset=utf-8",
    "Location": "xxxxxx/login"
});
res.end();
```


3）http头部信息详解?

HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。它于1990年提出，经过几年的使用与发展，得到不断地完善和扩展。目前在WWW中使用的是HTTP/1.0的第六版，HTTP/1.1的规范化工作正在进行之中，而且HTTP-NG(Next Generation of HTTP)的建议已经提出。

HTTP协议的主要特点可概括如下：

1.支持客户/服务器模式。

2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。

3.灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。
4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。
5.无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

一、HTTP协议详解之URL篇

http（超文本传输协议）是一个基于请求与响应模式的、无状态的、应用层的协议，常基于TCP的连接方式，HTTP1.1版本中给出一种持续连接的机制，绝大多数的Web开发，都是构建在HTTP协议之上的Web应用。

HTTP URL (URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息)的格式如下：
http://host[":"port][abs_path]
http表示要通过HTTP协议来定位网络资源；host表示合法的Internet主机域名或者IP地址；port指定一个端口号，为空则使用缺省端口80；abs_path指定请求资源的URI；如果URL中没有给出abs_path，那么当它作为请求URI时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。
eg:
1、输入：www.guet.edu.cn
浏览器自动转换成：http://www.guet.edu.cn/
2、http:192.168.0.116:8080/index.jsp

二、HTTP协议详解之请求篇

http请求由三部分组成，分别是：请求行、消息报头、请求正文

1、请求行以一个方法符号开头，以空格分开，后面跟着请求的URI和协议的版本，格式如下：Method Request-URI HTTP-Version CRLF
其中 Method表示请求方法；Request-URI是一个统一资源标识符；HTTP-Version表示请求的HTTP协议版本；CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

请求方法（所有方法全为大写）有多种，各个方法的解释如下：
GET　请求获取Request-URI所标识的资源
POST　在Request-URI所标识的资源后附加新的数据
HEAD　请求获取由Request-URI所标识的资源的响应消息报头
PUT　请求服务器存储一个资源，并用Request-URI作为其标识
DELETE 请求服务器删除Request-URI所标识的资源
TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断
CONNECT 保留将来使用
OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求
应用举例：
GET方法：在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用GET方法向服务器获取资源，eg:GET /form.html HTTP/1.1 (CRLF)

POST方法要求被请求服务器接受附在请求后面的数据，常用于提交表单。
eg：POST /reg.jsp HTTP/ (CRLF)
Accept:image/gif,image/x-xbit,... (CRLF)

...
HOST:www.guet.edu.cn (CRLF)
Content-Length:22 (CRLF)
Connection:Keep-Alive (CRLF)
Cache-Control:no-cache (CRLF)
(CRLF)　　//该CRLF表示消息报头已经结束，在此之前为消息报头
user=jeffrey&pwd=1234 //此行以下为提交的数据

HEAD方法与GET方法几乎是一样的，对于HEAD请求的回应部分来说，它的HTTP头部中包含的信息与通过GET请求所得到的信息是相同的。利用这个方法，不必传输整个资源内容，就可以得到Request-URI所标识的资源的信息。该方法常用于测试超链接的有效性，是否可以访问，以及最近是否更新。
2、请求报头后述
3、请求正文(略)

三、HTTP协议详解之响应篇

在接收和解释请求消息后，服务器返回一个HTTP响应消息。
HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响应正文
1、状态行格式如下：
HTTP-Version Status-Code Reason-Phrase CRLF
其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。
状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：
1xx：指示信息--表示请求已接收，继续处理
2xx：成功--表示请求已被成功接收、理解、接受
3xx：重定向--要完成请求必须进行更进一步的操作
4xx：客户端错误--请求有语法错误或请求无法实现
5xx：服务器端错误--服务器未能实现合法的请求
常见状态代码、状态描述、说明：
200 OK　　//客户端请求成功
400 Bad Request //客户端请求有语法错误，不能被服务器所理解
401 Unauthorized //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用
403 Forbidden //服务器收到请求，但是拒绝提供服务
404 Not Found //请求资源不存在，eg：输入了错误的URL
500 Internal Server Error //服务器发生不可预期的错误
503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常
eg：HTTP/1.1 200 OK （CRLF）

2、响应报头后述

3、响应正文就是服务器返回的资源的内容

四、HTTP协议详解之消息报头篇

HTTP消息由客户端到服务器的请求和服务器到客户端的响应组成。请求消息和响应消息都是由开始行（对于请求消息，开始行就是请求行，对于响应消息，开始行就是状态行），消息报头（可选），空行（只有CRLF的行），消息正文（可选）组成。

HTTP消息报头包括普通报头、请求报头、响应报头、实体报头。

每一个报头域都是由名字+“：”+空格+值 组成，消息报头域的名字是大小写无关的。

1、普通报头

在普通报头中，有少数报头域用于所有的请求和响应消息，但并不用于被传输的实体，只用于传输的消息。

eg：

Cache-Control 用于指定缓存指令，缓存指令是单向的（响应中出现的缓存指令在请求中未必会出现），且是独立的（一个消息的缓存指令不会影响另一个消息处理的缓存机制），HTTP1.0使用的类似的报头域为Pragma。

请求时的缓存指令包括：no-cache（用于指示请求或响应消息不能缓存）、no-store、max-age、max-stale、min-fresh、only-if-cached;

响应时的缓存指令包括：public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age、s-maxage.

eg：为了指示IE浏览器（客户端）不要缓存页面，服务器端的JSP程序可以编写如下：response.sehHeader("Cache-Control","no-cache");

//response.setHeader("Pragma","no-cache");作用相当于上述代码，通常两者//合用

这句代码将在发送的响应消息中设置普通报头域：Cache-Control:no-cache

Date普通报头域表示消息产生的日期和时间

Connection普通报头域允许发送指定连接的选项。例如指定连接是连续，或者指定“close”选项，通知服务器，在响应完成后，关闭连接

2、请求报头

请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。

常用的请求报头

Accept

Accept请求报头域用于指定客户端接受哪些类型的信息。eg：Accept：image/gif，表明客户端希望接受GIF图象格式的资源；Accept：text/html，表明客户端希望接受html文本。

Accept-Charset

Accept-Charset请求报头域用于指定客户端接受的字符集。eg：Accept-Charset:iso-8859-1,gb2312.如果在请求消息中没有设置这个域，缺省是任何字符集都可以接受。

Accept-Encoding

Accept-Encoding请求报头域类似于Accept，但是它是用于指定可接受的内容编码。eg：Accept-Encoding:gzip.deflate.如果请求消息中没有设置这个域服务器假定客户端对各种内容编码都可以接受。

Accept-Language

Accept-Language请求报头域类似于Accept，但是它是用于指定一种自然语言。eg：Accept-Language:zh-cn.如果请求消息中没有设置这个报头域，服务器假定客户端对各种语言都可以接受。

Authorization

Authorization请求报头域主要用于证明客户端有权查看某个资源。当浏览器访问一个页面时，如果收到服务器的响应代码为401（未授权），可以发送一个包含Authorization请求报头域的请求，要求服务器对其进行验证。

Host（发送请求时，该报头域是必需的）

Host请求报头域主要用于指定被请求资源的Internet主机和端口号，它通常从HTTP URL中提取出来的，eg：

我们在浏览器中输入：http://www.guet.edu.cn/index.html

浏览器发送的请求消息中，就会包含Host请求报头域，如下：

Host：www.guet.edu.cn
此处使用缺省端口号80，若指定了端口号，则变成：
Host：www.guet.edu.cn:指定端口号
User-Agent
我们上网登陆论坛的时候，往往会看到一些欢迎信息，其中列出了你的操作系统的名称和版本，你所使用的浏览器的名称和版本，这往往让很多人感到很神奇，实际上，服务器应用程序就是从User-Agent这个请求报头域中获取到这些信息。User-Agent请求报头域允许客户端将它的操作系统、浏览器和其它属性告诉服务器。不过，这个报头域不是必需的，如果我们自己编写一个浏览器，不使用User-Agent请求报头域，那么服务器端就无法得知我们的信息了。
请求报头举例：
GET /form.html HTTP/1.1 (CRLF)
Accept:image/gif,image/x-xbitmap,image/jpeg,application/x-shockwave-flash,application/vnd.ms-excel,application/vnd.ms-powerpoint,application/msword,*/* (CRLF)
Accept-Language:zh-cn (CRLF)
Accept-Encoding:gzip,deflate (CRLF)
If-Modified-Since:Wed,05 Jan 2007 11:21:25 GMT (CRLF)
If-None-Match:W/"80b1a4c018f3c41:8317" (CRLF)
User-Agent:Mozilla/4.0(compatible;MSIE6.0;Windows NT 5.0) (CRLF)
Host:www.guet.edu.cn (CRLF)
Connection:Keep-Alive (CRLF)
(CRLF)

3、响应报头
响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对Request-URI所标识的资源进行下一步访问的信息。
常用的响应报头
Location
Location响应报头域用于重定向接受者到一个新的位置。Location响应报头域常用在更换域名的时候。
Server
Server响应报头域包含了服务器用来处理请求的软件信息。与User-Agent请求报头域是相对应的。下面是
Server响应报头域的一个例子：
Server：Apache-Coyote/1.1
WWW-Authenticate
WWW-Authenticate响应报头域必须被包含在401（未授权的）响应消息中，客户端收到401响应消息时候，并发送Authorization报头域请求服务器对其进行验证时，服务端响应报头就包含该报头域。
eg：WWW-Authenticate:Basic realm="Basic Auth Test!" //可以看出服务器对请求资源采用的是基本验证机制。

4、实体报头
请求和响应消息都可以传送一个实体。一个实体由实体报头域和实体正文组成，但并不是说实体报头域和实体正文要在一起发送，可以只发送实体报头域。实体报头定义了关于实体正文（eg：有无实体正文）和请求所标识的资源的元信息。
常用的实体报头
Content-Encoding
Content-Encoding实体报头域被用作媒体类型的修饰符，它的值指示了已经被应用到实体正文的附加内容的编码，因而要获得Content-Type报头域中所

引用的媒体类型，必须采用相应的解码机制。Content-Encoding这样用于记录文档的压缩方法，eg：Content-Encoding：gzip

Content-Language

Content-Language实体报头域描述了资源所用的自然语言。没有设置该域则认为实体内容将提供给所有的语言阅读者。eg：Content-Language:da

Content-Length

Content-Length实体报头域用于指明实体正文的长度，以字节方式存储的十进制数字来表示。

Content-Type

Content-Type实体报头域用语指明发送给接收者的实体正文的媒体类型。

eg：

Content-Type:text/html;charset=ISO-8859-1

Content-Type:text/html;charset=GB2312

Last-Modified

Last-Modified实体报头域用于指示资源的最后修改日期和时间。

Expires

Expires实体报头域给出响应过期的日期和时间。为了让代理服务器或浏览器在一段时间以后更新缓存中(再次访问曾访问过的页面时，直接从缓存中加载，缩短响应时间和降低服务器负载)的页面，我们可以使用Expires实体报头域指定页面过期的时间。eg：Expires：Thu，15 Sep 2006 16:23:12 GMT

HTTP1.1的客户端和缓存必须将其他非法的日期格式（包括0）看作已经过期。eg：为了让浏览器不要缓存页面，我们也可以利用Expires实体报头域，设置为0，jsp中程序如下：response.setDateHeader("Expires","0");

五、利用telnet观察http协议的通讯过程

实验目的及原理：

利用MS的telnet工具，通过手动输入http请求信息的方式，向服务器发出请求，服务器接收、解释和接受请求后，会返回一个响应，该响应会在telnet窗口上显示出来，从而从感性上加深对http协议的通讯过程的认识。

实验步骤：

1、打开telnet

1.1 打开telnet

运行-->cmd-->telnet

1.2 打开telnet回显功能

set localecho

2、连接服务器并发送请求

2.1 open www.guet.edu.cn 80 //注意端口号不能省略

HEAD /index.asp HTTP/1.0

Host:www.guet.edu.cn

/*我们可以变换请求方法,请求桂林电子主页内容,输入消息如下*/

open www.guet.edu.cn 80

GET /index.asp HTTP/1.0 //请求资源的内容

Host:www.guet.edu.cn

2.2 open www.sina.com.cn 80 //在命令提示符号下直接输入

telnet www.sina.com.cn 80

HEAD /index.asp HTTP/1.0

Host:www.sina.com.cn

3 实验结果：

3.1 请求信息2.1得到的响应是:

HTTP/1.1 200 OK　　//请求成功
Server: Microsoft-IIS/5.0　　//web服务器
Date: Thu,08 Mar 200707:17:51 GMT
Connection: Keep-Alive
Content-Length: 23330
Content-Type: text/html
Expries: Thu,08 Mar 2007 07:16:51 GMT
Set-Cookie:
ASPSESSIONIDQAQBQQQB=BEJCDGKADEDJKLKKAJEOIMMH; path=/
Cache-control: private

//资源内容省略

3.2 请求信息2.2得到的响应是:

HTTP/1.0 404 Not Found　　//请求失败
Date: Thu, 08 Mar 2007 07:50:50 GMT
Server: Apache/2.0.54 <Unix>
Last-Modified: Thu, 30 Nov 2006 11:35:41 GMT
ETag: "6277a-415-e7c76980"
Accept-Ranges: bytes
X-Powered-By: mod_xlayout_jh/0.0.1vhs.markII.remix
Vary: Accept-Encoding
Content-Type: text/html
X-Cache: MISS from zjm152-78.sina.com.cn
Via: 1.0 zjm152-78.sina.com.cn:80<squid/2.6.STABLES-20061207>
X-Cache: MISS from th-143.sina.com.cn
Connection: close

失去了跟主机的连接

按任意键继续...

4 .注意事项：1、出现输入错误，则请求不会成功。

2、报头域不分大小写。

3、更深一步了解HTTP协议，可以查看RFC2616，在http://www.letf.org/rfc上找到该文件。

4、开发后台程序必须掌握http协议

六、HTTP协议相关技术补充

1、基础：

高层协议有：文件传输协议FTP、电子邮件传输协议SMTP、域名系统服务DNS、网络新闻传输协议NNTP和HTTP协议等

中介由三种：代理(Proxy)、网关(Gateway)和通道(Tunnel)，一个代理根据URI的绝对格式来接受请求，重写全部或部分消息，通过 URI的标识把已格式化过的请求发送到服务器。网关是一个接收代理，作为一些其它服务器的上层，并且如果必须的话，可以把请求翻译给下层的服务器协议。一 个通道作为不改变消息的两个连接之间的中继点。当通讯需要通过一个中介(例如：防火墙等)或者是中介不能识别消息的内容时，通道经常被使用。

代理(Proxy)：一个中间程序，它可以充当一个服务器，也可以充当一个客

户机，为其它客户机建立请求。请求是通过可能的翻译在内部或经过传递到其它的 服务器中。一个代理在发送请求信息之前，必须解释并且如果可能重写它。代理经常作为通过防火墙的客户机端的门户，代理还可以作为一个帮助应用来通过协议处 理没有被用户代理完成的请求。
网关(Gateway)：一个作为其它服务器中间媒介的服务器。与代理不同的是，网关接受请求就好象对被请求的资源来说它就是源服务器；发出请求的客户机并没有意识到它在同网关打交道。
网关经常作为通过防火墙的服务器端的门户，网关还可以作为一个协议翻译器以便存取那些存储在非HTTP系统中的资源。
　通道(Tunnel)：是作为两个连接中继的中介程序。一旦激活，通道便被认为不属于HTTP通讯，尽管通道可能是被一个HTTP请求初始化的。当被中继 的连接两端关闭时，通道便消失。当一个门户(Portal)必须存在或中介(Intermediary)不能解释中继的通讯时通道被经常使用。

2、协议分析的优势—HTTP分析器检测网络攻击
以模块化的方式对高层协议进行分析处理，将是未来入侵检测的方向。
HTTP及其代理的常用端口80、3128和8080在network部分用port标签进行了规定

3、HTTP协议Content Lenth限制漏洞导致拒绝服务攻击
使用POST方法时，可以设置ContentLenth来定义需要传送的数据长度，例如ContentLenth:999999999，在传送完成前，内 存不会释放，攻击者可以利用这个缺陷，连续向WEB服务器发送垃圾数据直至WEB服务器内存耗尽。这种攻击方法基本不会留下痕迹。
http://www.cnpaf.net/Class/HTTP/0532918532667330.html

4、利用HTTP协议的特性进行拒绝服务攻击的一些构思
服务器端忙于处理攻击者伪造的TCP连接请求而无暇理睬客户的正常请求（毕竟客户端的正常请求比率非常之小），此时从正常客户的角度看来，服务器失去响应，这种情况我们称作：服务器端受到了SYNFlood攻击（SYN洪水攻击）。
而Smurf、TearDrop等是利用ICMP报文来Flood和IP碎片攻击的。本文用“正常连接”的方法来产生拒绝服务攻击。
19端口在早期已经有人用来做Chargen攻击了，即Chargen_Denial_of_Service，但是！他们用的方法是在两台Chargen 服务器之间产生UDP连接，让服务器处理过多信息而DOWN掉，那么，干掉一台WEB服务器的条件就必须有2个：1.有Chargen服务2.有HTTP 服务
方法：攻击者伪造源IP给N台Chargen发送连接请求（Connect），Chargen接收到连接后就会返回每秒72字节的字符流（实际上根据网络实际情况，这个速度更快）给服务器。

5、Http指纹识别技术
　Http指纹识别的原理大致上也是相同的：记录不同服务器对Http协议执行中的微小差别进行识别.Http指纹识别比TCP/IP堆栈指纹识别复杂许 多,理由是定制Http服务器的配置文件、增加插件或组件使得更改Http的响应信息变的很容易,这样使得识别变的困难；然而定制TCP/IP堆栈的行为 需要对核心层进行修改,所以就容易识别.
　　要让服务器返回不同的Banner信息的设置是很简单的,象Apache这样的开放源代码的Http服务器,用户可以在源代码里修改Banner信息,然 后重起Http服务就生效了；对于没有公开源代码的Http服务器比如微软的IIS或者是Netscape,可以在存放Banner信息的Dll文件中修 改,相关的文章有讨论的,这里不再赘述,当然这样的修改的效果还是不错的.另外一种模糊Banner信息的方法

是使用插件。
常用测试请求：
1：HEAD/Http/1.0发送基本的Http请求
2：DELETE/Http/1.0发送那些不被允许的请求,比如Delete请求
3：GET/Http/3.0发送一个非法版本的Http协议请求
4：GET/JUNK/1.0发送一个不正确规格的Http协议请求
Http指纹识别工具Httprint,它通过运用统计学原理,组合模糊的逻辑学技术,能很有效的确定Http服务器的类型.它可以被用来收集和分析不同Http服务器产生的签名。

6、其他：为了提高用户使用浏览器时的性能，现代浏览器还支持并发的访问方式，浏览一个网页时同时建立多个连接，以迅速获得一个网页上的多个图标，这样能更快速完成整个网页的传输。
HTTP1.1中提供了这种持续连接的方式，而下一代HTTP协议：HTTP-NG更增加了有关会话控制、丰富的内容协商等方式的支持，来提供
更高效率的连接。

4）并发连接数、请求数、并发用户数分别是什么意思?

概念

并发连接数-SBC（Simultaneous Browser Connections）

并发连接数指的是客户端向服务器发起请求，并建立了TCP连接。每秒钟服务器链接的总TCP数量，就是并发连接数。

请求数-QPS（Query Per Second）/RPS（Request Per Second）

请求数有2个缩写，可以叫QPS也可以叫RPS。单位是每秒多少请求。Query=查询，也相当于请求。请求数指的是客户端在建立完连接后，向http服务发出GET/POST/HEAD数据包，服务器返回了请求结果后有两种情况：

- http数据包头包含Close字样，关闭本次TCP连接；
- http数据包头包含Keep-Alive字样，本次连接不关闭，可继续通过该连接继续向http服务发送请求，用于减少TCP并发连接数。

服务器性能怎么测?

通常情况下，我们测试的是QPS，也就是每秒请求数。不过为了衡量服务器的总体性能，测试时最好一起测试并发连接数和请求数。

测试原理

- 测试并发连接数采用每个并发1请求，多个并发进行；
- 测试请求数采用多并发、每个并发多个请求进行，总的请求数将会=并发数*单并发请求数，需要注意的是不同的并发和单并发请求数得出来的结果会不同，因此最好测试多次取平均值。

区分请求数意义何在?

大家打开Chrome浏览器，按下F12，切换到Network选项卡，随便打开一个网页，按下F5刷新，将会看到刷刷一堆的请求。这里给出某大牛收集来的不同浏览器产生的单站点并发连接数：

浏览器	HTTP 1.1	HTTP 1.0

IE 6,7	2	4
IE 8	6	6
Firefox 2	2	8
Firefox 3	6	6
Safari 3, 4	4	4
Chrome 1,2	6	?
Chrome 3	4	4
Opera 9.63,10.00alpha	4	4

以Chrome为例，假设服务器设置的是Close(非持久连接)，浏览器打开网页后，首先打开4个并发加载数据，在这些请求完成后关闭4个连接，再打开4个并发连接加载数据。也就是说，并不是这个网页有100个请求就会产生100并发，而是4个并发连接并行。假设服务器设置的是keep-alive(持久连接)，浏览器打开网页后，首先打开4个并发加载数据，在这些请求完成后不关闭连接，而是继续发出请求，节约重新打开连接的时间。【前面红色标出的是keep-alive持久连接和close非持久的区别，持久连接除了Squid(这货用了特殊方法在http 1.0实现持久连接)，只在http 1.1协议中有效！】

主机到底能多少人在线?

看到这里相信你已经知道答案了，这个问题无解，根据网页的内容大小和单网页的请求数和服务器的配置而定，这个数据的浮动值非常大所以无法测量。因此能承诺保证多少用户在线就是坑爹的主机商！

并发用户

并发用户数量，有两种常见的错误观点。一种错误观点是把并发用户数量理解为使用系统的全部用户的数量，理由是这些用户可能同时使用系统；还有一种比较接近正确的观点是把用户在线数量理解为并发用户数量。实际上，在线用户不一定会和其他用户发生并发，例如正在浏览网页的用户，对服务器是没有任何影响的。但是，用户在线数量是统计并发用户数量的主要依据之一。

并发主要是针对服务器而言，是否并发的关键是看用户操作是否对服务器产生了影响。因此，并发用户数量的正确理解为：在同一时刻与服务器进行了交互的在线用户数量。这些用户的最大特征是和服务器产生了交互，这种交互既可以是单向的传输数据，也可以是双向的传送数据。
并发用户数量的统计的方法目前还没有准确的公式，因为不同系统会有不同的并发特点。例如OA系统统计并发用户数量的经验公式为：使用系统用户数量*(5%~20%)。对于这个公式是没有必要拘泥于计算的结果，因为为了保证系统的扩展空间，测试时的并发用户数量要稍微大一些，除非是要测试系统能承载的最大并发用户数量。举例说明：如果一个OA系统的期望用户为1000个，只要测试出系统能支持200个并发用户就可以了。

4、浏览器

1）浏览器加载渲染网页的过程

浏览器的工作机制，一句话概括起来就是：web浏览器与web服务器之间通过HTTP协议进行通信的过程。所以，C/S之间握手的协议就是HTTP协议。浏览器接收完毕开始渲染之前大致过程如下：

从浏览器地址栏的请求链接开始，浏览器通过DNS解析查到域名映射的IP地址，成功之后浏览器端向此IP地址取得连接，成功连接之后，浏览器端将请求头信息 通过HTTP协议向此IP地址所在服务器发起请求，服务器接受到请求之后等待处理，最后向浏览器端发回响应，此时在HTTP协议下，浏览器从服务器接收到 text/html类型的代码，浏览器开始显示此html，并获取其中内嵌资源地址，然后浏览器再发起请求来获取这些资源，并在浏览器的html中显示。

离我们最近并能直接显示一个完整通信过程的工具就是Firebug了，看下图:

其中黄色的tips浮层告诉了我们”colorBox.html”从发起请求到关闭连接整个过程中每个环节的时长（域名解析 -> 建立连接 -> 发起请求 -> 等待响应 -> 接收数据），点击该请求，可以获得HTTP的headers信息，包含响应头信息与请求头信息，如:

```
//响应头信息 HTTP/1.1 304 Not Modified Date: Wed, 02 Mar 2011 08:20:06 GMT Server: Apache/2.2.4 (Win32) PHP/5.2.1 Connection: Keep-Alive Keep-Alive: timeout=5, max=100 Etag: "1e483-1324-a86f5621"
//请求头信息 GET /Docs/eva/api/colorBox.html HTTP/1.1 Host: ued.com User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; zh-CN; rv:1.9.2.13) Gecko/20101203 Firefox/3.6.13 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Language: zh-cn,zh;q=0.5 Accept-Encoding: gzip,deflate Accept-Charset: GB2312,utf-8;q=0.7,*;q=0.7 Keep-Alive: 115 Connection: keep-alive Referer: http://ued.com/Docs/ If-Modified-Since: Thu, 17 Feb 2011 10:14:07 GMT If-None-Match: "1e483-1324-a86f5621" Cache-Control: max-age=0
```

另外，ajax异步请求同样遵循HTTP协议，原理大同小异。

浏览器加载显示html页面内容的顺序

我们经常看到浏览器在加载某个页面时,部分内容先显示出来,又有些内容后显示。那么浏览器加载显示html究竟是按什么顺序进行的呢?

其实浏览器加载显示html的顺序是按下面的顺序进行的:
1、IE下载的顺序是从上到下，渲染的顺序也是从上到下，下载和渲染是同时进行的。
2、在渲染到页面的某一部分时，其上面的所有部分都已经下载完成（并不是说所有相关联的元素都已经下载完）。
3、如果遇到语义解释性的标签嵌入文件（JS脚本，CSS 剑　敲创耸盗E的下载过程会启用单独连接进行下载。
4、并且在下载后进行解析，解析过程中，停止页面所有往下元素的下载。
5、样式表在下载完成后，将和以前下载的所有样式表一起进行解析，解析完成后，将对此前所有元素（含以前已经渲染的）重新进行渲染。
6、JS、CSS中如有重定义，后定义函数将覆盖前定义函数。

Firefox处理下载和渲染顺序大体相同，只是在细微之处有些差别，例如：
iframe的渲染

如果你的网页比较大，希望部分内容先显示出来，粘住浏览者，那么你可以按照上面的规则合理的布局你的网页，达到预期的目的。

JS的加载
不能并行下载和解析（阻塞下载）
当 引用了JS的时候，浏览器发送1个jsrequest就会一直等待该request的返回。因为浏览器需要1个稳定的DOM树结构，而JS中很有可能有代 码直接改变了DOM树结构，比如使用 document.write 或 appendChild,甚至是直接使用的location.href进行跳转，浏览器为了防止出现JS修改DOM树，需要重新构建DOM树的情况，所以 就会阻塞其他的下载和呈现.

为了更清楚的显示页面元素的加载顺序，动手写了一个程序，程序对页面中的每个元素都延迟10秒。

程序的位置在见附件。

首先查看TestHtmlOrder.aspx这个页面，使用HttpWatcher来检测页面元素的加载。

从下面的图中可以看到加载顺序。

IE首先加载了主页面TestHtmlOrder.aspx，

下载了主页面后，页面首先显示的是“红色剑灵”、“蓝色剑灵”几个字，但此时显示的是只是黑色字体，没有样式，因为样式还没有下载下来。

接下来页面中的标签是JS标签，属于嵌入文件，因此IE需要将其下载下来。这有两个文件，虽然IE同时能够和WebServer建立两个链接，但是此时并没有使用两个连接，而是使用一个连接，在下载完成后，接下来才下载另外一个文件。

究其原因，是因为JS包含了语法定义，在第二个文件里面的函数可能用到了第一个文件里面的变量和函数，IE没有办法判断，或者需要很耗时的判断，才 能判断文件下载的先后顺序。而在解释方面，IE对JS文件是下载一个，解释一个(可以执行文件TestJsOrder2.aspx)。如果先下载的是第二 个文件，此时就会发生解释错误。因此需要开发者自己在放置JS文件位置时，按先后顺序放好，IE依次下载进行解释。后面的函数覆盖前面的函数定义

在下载完成后，我们看到helloWorld，helloworld2，开始顺序执行。而此时字体的样式表和图片仍然没有下载下来。

在helloWorld，helloWorld2执行过程时，此时页面停留在函数执行的中断点(alert部分)。此时IE并没有去下载CSS的文件。由此说明JS函数的执行会阻塞IE的下载。

接下来我们看到CSS文件的下载也是使用了一个连接，也是串行下载。其串行下载的原因和JS串行下载原因是一样的。

在两个CSS文件下载过程中，我们看到“红色剑灵”，“蓝色剑灵”依次变为红色和蓝色，两者颜色的转换时间相差在10秒，说明样式文件和JS文件一样是下载完一个解析一个的。

现在转到TestCssOrder.aspx看一下，可以看到 开始时“红色剑灵”，“红色强壮

剑灵”，显示为红色，过了10秒“蓝色剑灵”显示为蓝色，再过10秒，“红色强壮剑灵”字体变粗了，同时“红色强壮剑灵 2”开始出现。在刚开始“红色剑灵”，“红色强壮剑灵”显示红色时，第三个样式还没有下载下来，此时IE使用已经下载到样式对上面的元素渲染了一遍，此时 虽然“红色剑灵”，“红色强壮剑灵”样式定义不同，但是显示效果一样。第三个文件下载后，此时IE又重新对“红色强壮剑灵”渲染了一遍，此时其变为加粗， 以上所有的文件加载并且渲染完成后，开始渲染下面的标签“红色强壮剑灵2”

有一点需要证明：在IE使用样式对标签进行渲染时，是不是停止了其他页面元素的下载?原来我想通过加长渲染时间(利用滤镜，将标签元素数目增大)来检测，不过没有验证成功。只是从JS函数的执行推断CSS的渲染也是如此。

接下来看到的是图片文件下载，此时看到的是两个图片同时开始下载，而且是下载完成后，立即在页面上开始显示，直到所有的图片下载完成。

注：一个测试文件在网络传输上所花费时间的办法。

首先需要明白检测中w ait值的意义：wait = 服务器所花时间 + 网络时间

服务器所花时间我们可以用Thread.Sleep(10000);来让其休息10s，

比如这个：

由此大概可以计算出 10.002-10 = 0.002秒，这就是大概在网络上所花的时间。

2）浏览器工作原理

简介

浏览器可以被认为是使用最广泛的软件，本文将介绍浏览器的工作原理，我们将看到，从你在地址栏输入google.com到你看到google主页过程中都发生了什么。

将讨论的浏览器

今天，有五种主流浏览器——IE、Firefox、Safari、Chrome及Opera。

本文将基于一些开源浏览器的例子——Firefox、 Chrome及Safari，Safari是部分开源的。

根据W3C（World Wide Web Consortium 万维网联盟）的浏览器统计数据，当前（2011年5月），Firefox、Safari及Chrome的市场占有率综合已接近60%。（原文为2009年10月，数据没有太大变化）因此，可以说开源浏览器已经占据了浏览器市场的半壁江山。

浏览器的主要功能

浏览器的主要功能是将用户选择得web资源呈现出来，它需要从服务器请求资源，并将其显示在浏览器窗口中，资源的格式通常是HTML，也包括PDF、image及其他格式。用户用URI（Uniform Resource Identifier 统一资源标识符）来指定所请求资源的位置，在网络一章有更多讨论。

HTML和CSS规范中规定了浏览器解释html文档的方式，由 W3C组织对这些规范进行维护，W3C是负责制定web标准的组织。

HTML规范的最新版本是HTML4(http://www.w3.org/TR/html401/)，HTML5还在制定中（译注：两

年前），最新的CSS规范版本是2（http://www.w3.org/TR/CSS2），CSS3也还正在制定中（译注：同样两年前）。

这些年来，浏览器厂商纷纷开发自己的扩展，对规范的遵循并不完善，这为web开发者带来了严重的兼容性问题。

但是，浏览器的用户界面则差不多，常见的用户界面元素包括：

- 用来输入URI的地址栏
- 前进、后退按钮
- 书签选项
- 用于刷新及暂停当前加载文档的刷新、暂停按钮
- 用于到达主页的主页按钮

奇怪的是，并没有哪个正式公布的规范对用户界面做出规定，这些是多年来各浏览器厂商之间相互模仿和不断改进得结果。

Html5并没有规定浏览器必须具有的UI元素，但列出了一些常用元素，包括地址栏、状态栏及工具栏。还有一些浏览器有自己专有得功能，比如Firefox得下载管理。更多相关内容将在后面讨论用户界面时介绍。

浏览器的主要构成High Level Structure

浏览器的主要组件包括：

1. 用户界面－包括地址栏、后退/前进按钮、书签目录等，也就是你所看到的除了用来显示你所请求页面的主窗口之外的其他部分
2. 浏览器引擎－用来查询及操作渲染引擎的接口
3. 渲染引擎－用来显示请求的内容，例如，如果请求内容为html，它负责解析html及css，并将解析后的结果显示出来
4. 网络－用来完成网络调用，例如http请求，它具有平台无关的接口，可以在不同平台上工作
5. UI后端－用来绘制类似组合选择框及对话框等基本组件，具有不特定于某个平台的通用接口，底层使用操作系统的用户接口
6. JS解释器－用来解释执行JS代码
7. 数据存储－属于持久层，浏览器需要在硬盘中保存类似cookie的各种数据，HTML5定义了web database技术，这是一种轻量级完整的客户端存储技术

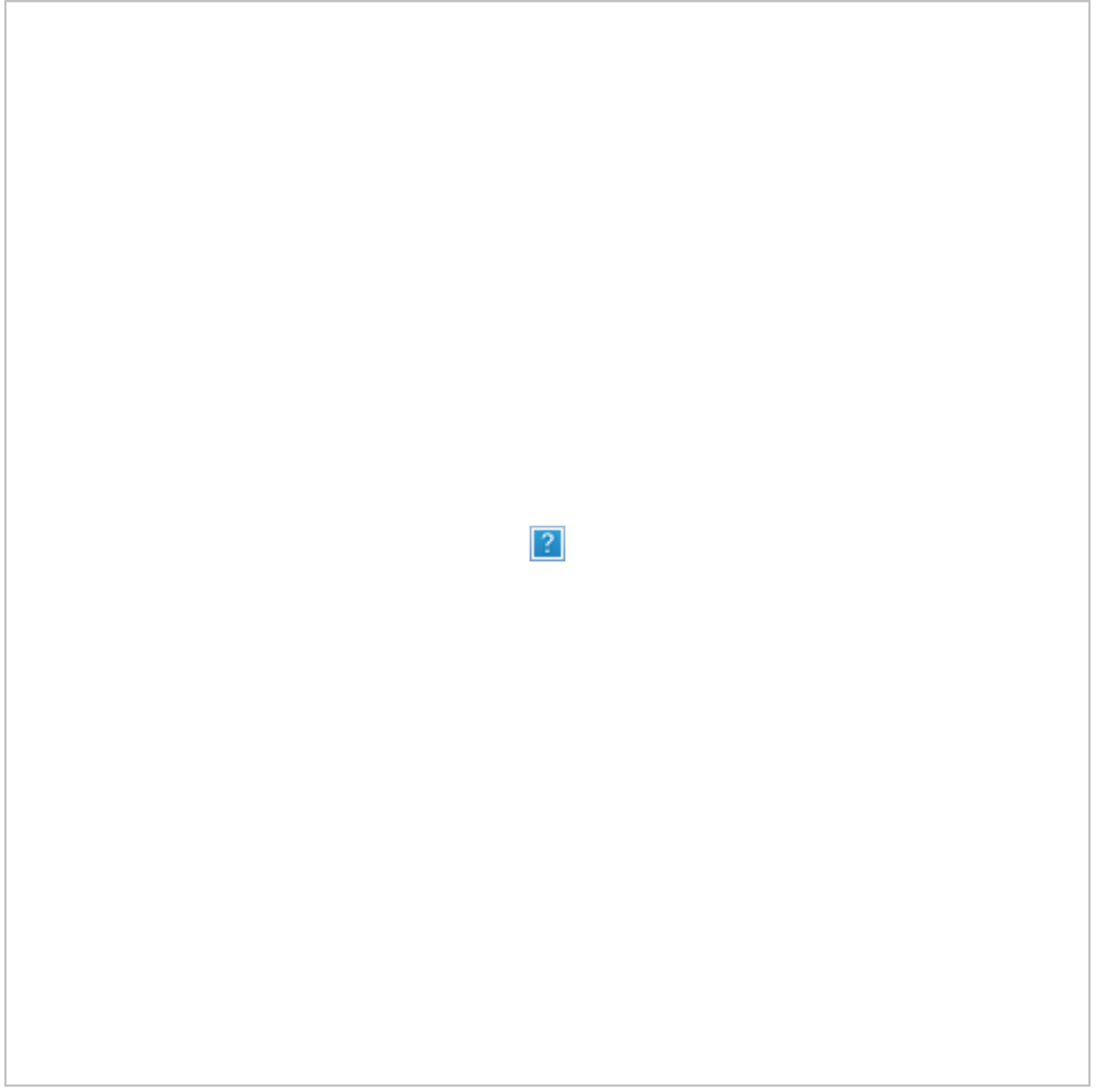


图1：浏览器主要组件

需要注意的是，不同于大部分浏览器，Chrome为每个Tab分配了各自的渲染引擎实例，每个Tab就是一个独立的进程。

对于构成浏览器的这些组件，后面会逐一详细讨论。

组件间的通信 Communication between the components

Firefox和Chrome都开发了一个特殊的通信结构，后面将有专门的一章进行讨论。

渲染引擎 The rendering engine

渲染引擎的职责就是渲染，即在浏览器窗口中显示所请求的内容。

默认情况下，渲染引擎可以显示html、xml文档及图片，它也可以借助插件（一种浏览器扩展）显示其他类型数据，例如使用PDF阅读器插件，可以显示PDF格式，将由专门一章讲解插件及扩展，这里只讨论渲染引擎最主要的用途——显示应用了CSS之后的html及图片。

渲染引擎 Rendering engines

本文所讨论得浏览器——Firefox、Chrome和Safari是基于两种渲染引擎构建的，Firefox使用Geoko——Mozilla自主研发的渲染引擎，Safari和Chrome都使用webkit。

Webkit是一款开源渲染引擎，它本来是为Linux平台研发的，后来由Apple移植到Mac及Windows上，相关内容请参考http://webkit.org。

主流程 The main flow

渲染引擎首先通过网络获得所请求文档的内容，通常以8K分块的方式完成。

下面是渲染引擎在取得内容之后的基本流程：

解析html以构建dom树->构建render树->布局render树->绘制render树

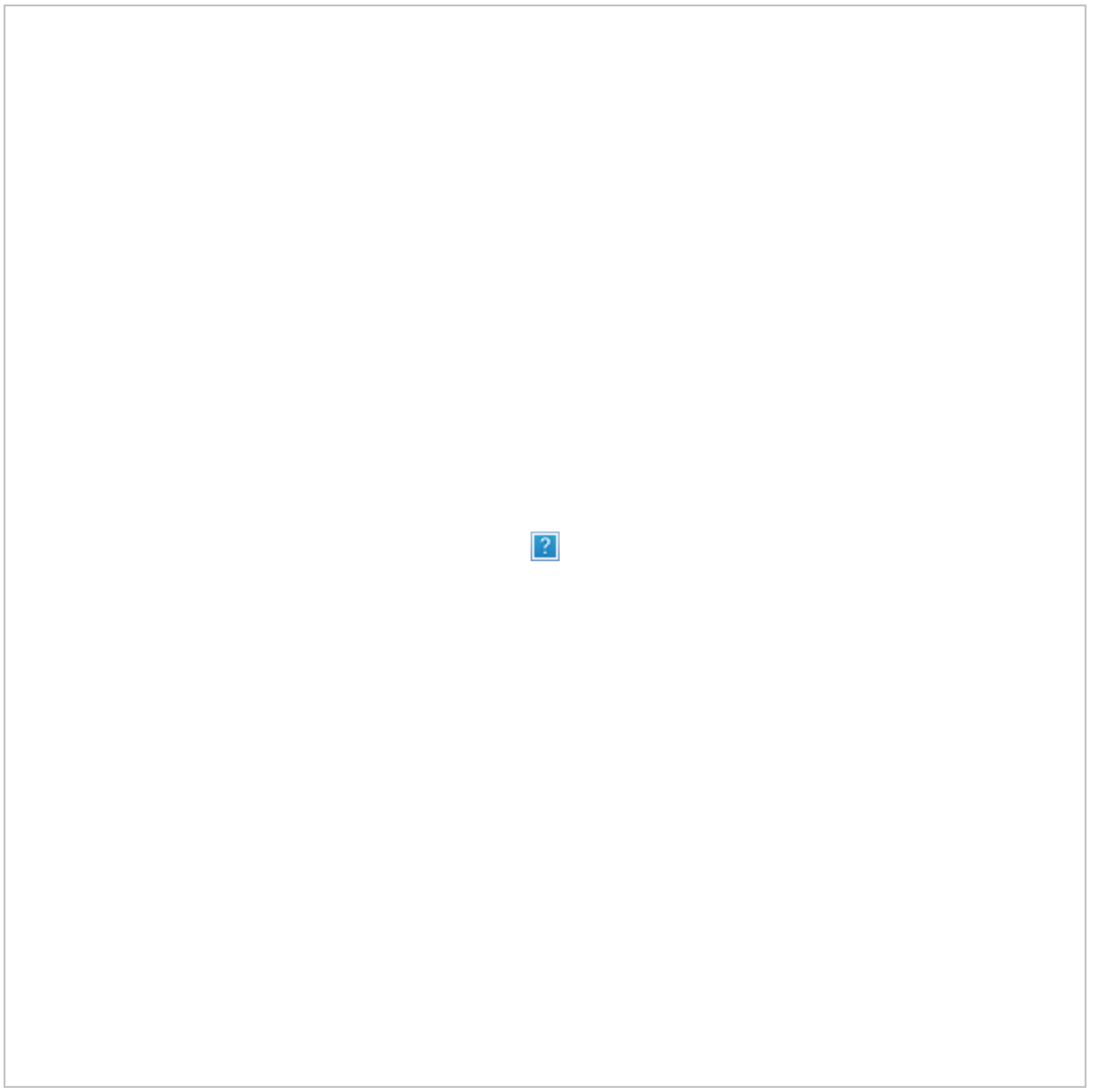


图2：渲染引擎基本流程

渲染引擎开始解析html，并将标签转化为内容树中的dom节点。接着，它解析外部CSS文件及style标签中的样式信息。这些样式信息以及html中的可见性指令将被用来构建另一棵树——render树。

Render树由一些包含有颜色和大小等属性的矩形组成，它们将被按照正确的顺序显示到屏幕上。

Render树构建好了之后，将会执行布局过程，它将确定每个节点在屏幕上的确切坐标。再下一步就是绘制，即遍历render树，并使用UI后端层绘制每个节点。

值得注意的是，这个过程是逐步完成的，为了更好的用户体验，渲染引擎将会尽可能早的将内容呈现到屏幕上，并不会等到所有的html都解析完成之后再去构建和布局render树。它是解析完一部分内容就显示一部分内容，同时，可能还在通过网络下载其余内容。

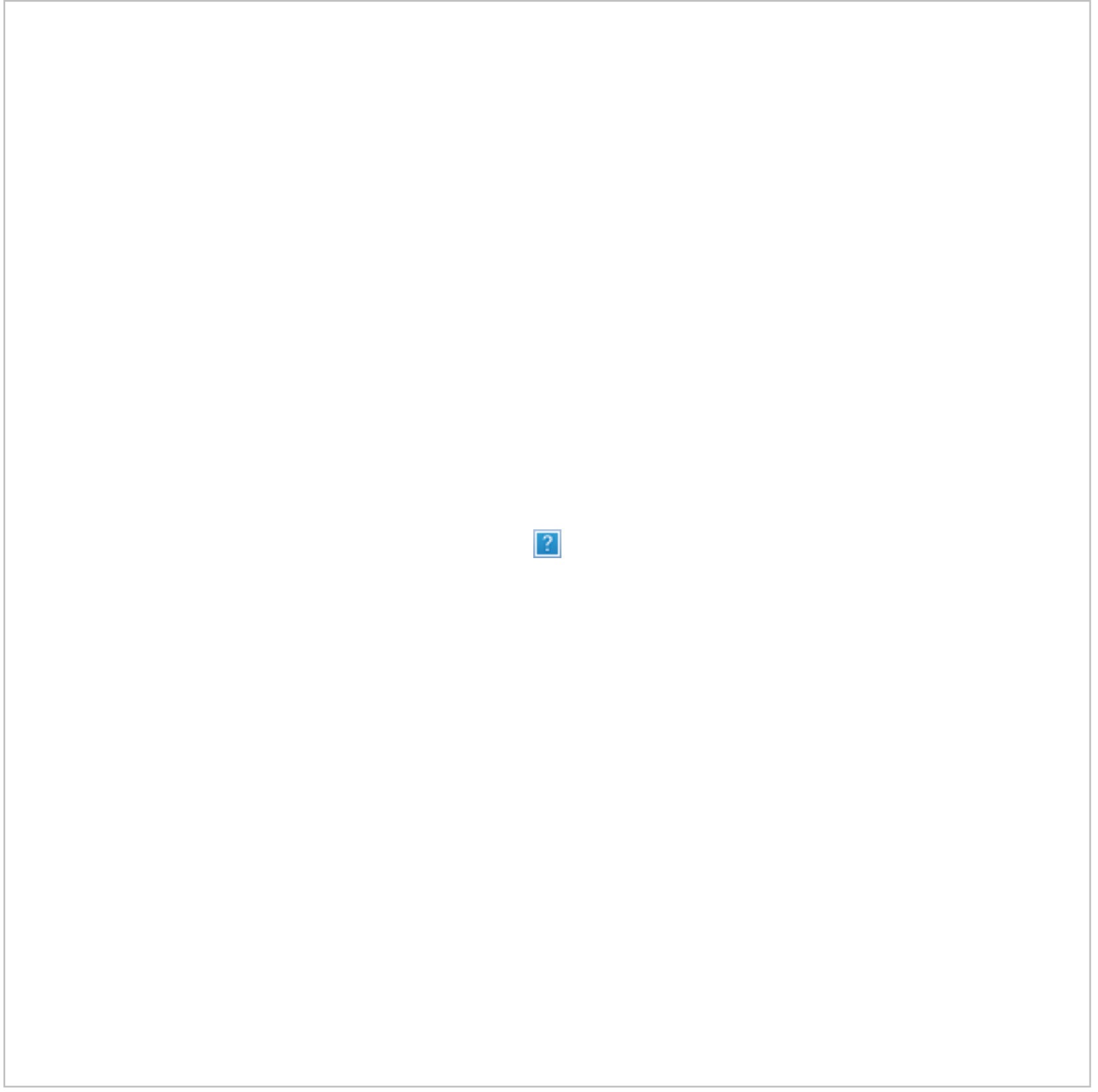


图3：webkit主流程

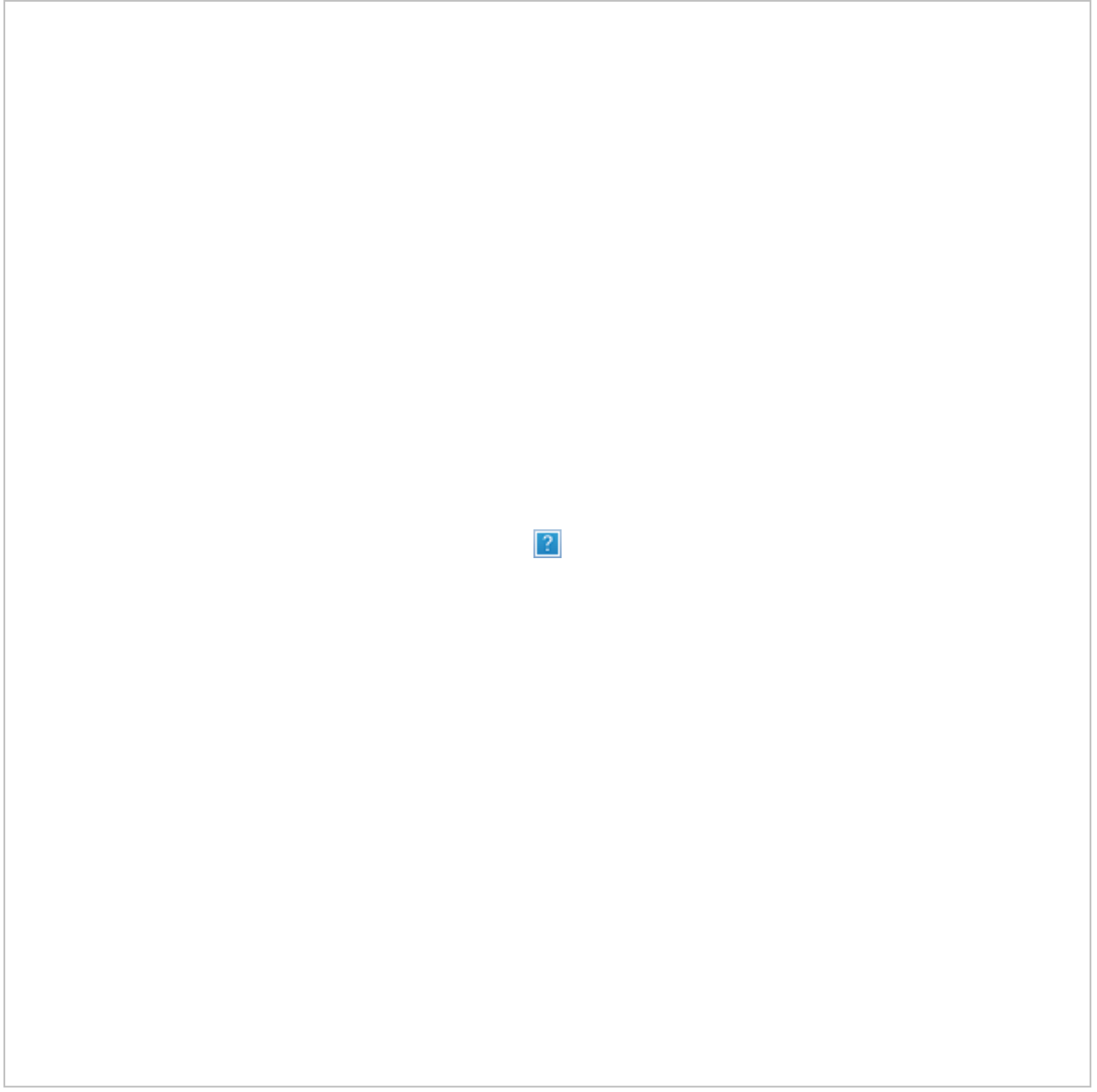


图4：Mozilla的Geoko 渲染引擎主流程

从图3和4中可以看出，尽管webkit和Gecko使用的术语稍有不同，他们的主要流程基本相同。Gecko称可见的格式化元素组成的树为frame树，每个元素都是一个frame，webkit则使用render树这个名词来命名由渲染对象组成的树。Webkit中元素的定位称为布局，而Gecko中称为回流。Webkit称利用dom节点及样式信息去构建render树的过程为attachment，Gecko在html和dom树之间附加了一层，这层称为内容接收器，相当制造dom元素的工厂。下面将讨论流程中的各个阶段。

解析 Parsing－general

既然解析是渲染引擎中一个非常重要的过程，我们将稍微深入的研究它。首先简要介绍一下解析。

解析一个文档即将其转换为具有一定意义的结构——编码可以理解和使用的东西。解析的结果通常是表达文档结构的节点树，称为解析树或语法树。

例如，解析“2＋3－1”这个表达式，可能返回这样一棵树。

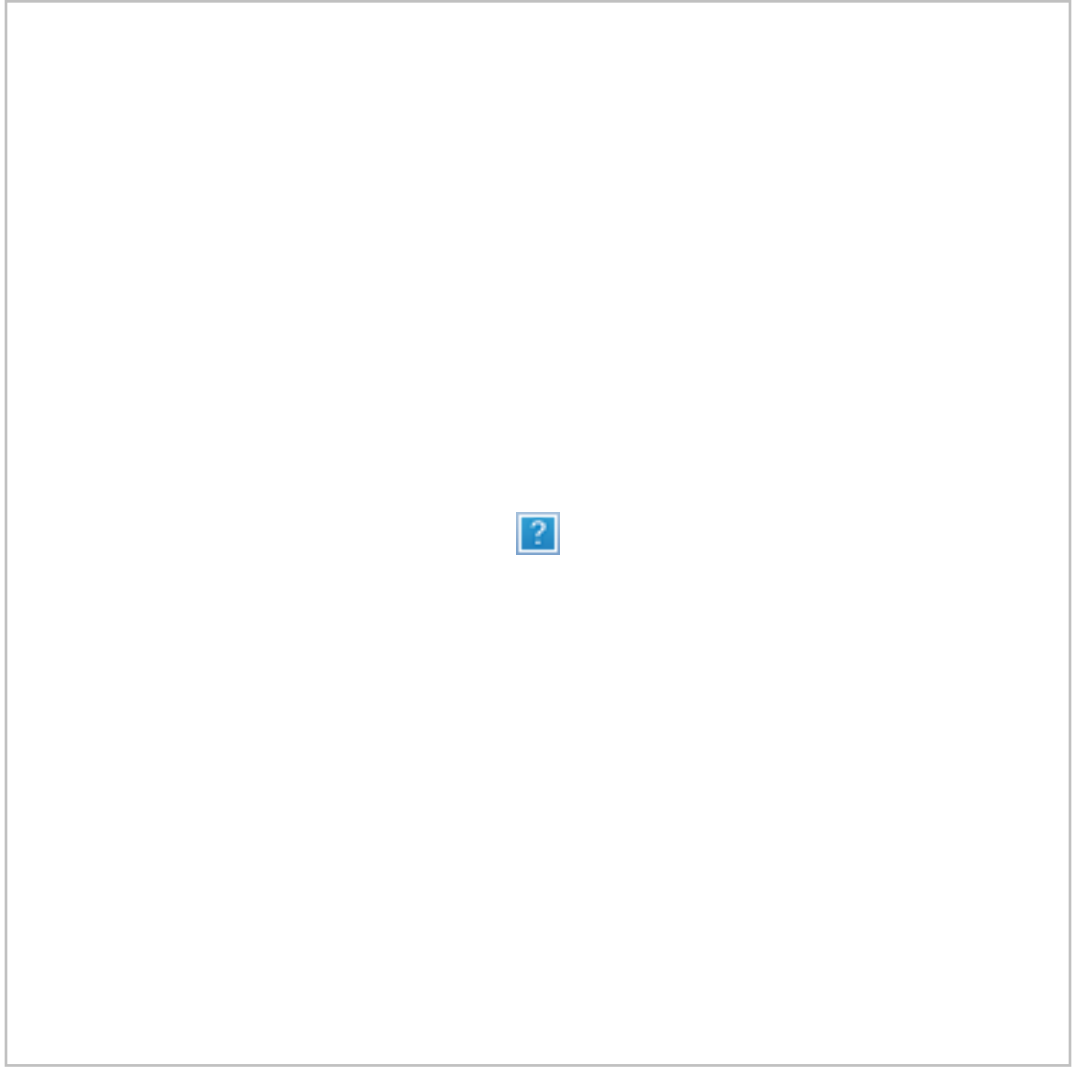


图5：数学表达式树节点

文法 Grammars

解析基于文档依据的语法规则——文档的语言或格式。每种可被解析的格式必须具有由词汇及语法规则组成的特定的文法，称为上下文无关文法。人类语言不具有这一特性，因此不能被一般的解析技术所解析。

解析器－词法分析器 Parser－Lexer combination

解析可以分为两个子过程——语法分析及词法分析

词法分析就是将输入分解为符号，符号是语言的词汇表——基本有效单元的集合。对于人类语言来说，它相当于我们字典中出现的所有单词。

语法分析指对语言应用语法规则。

解析器一般将工作分配给两个组件——词法分析器（有时也叫分词器）负责将输入分解为合法的符号，解析器则根据语言的语法规则分析文档结构，从而构建解析树，词法分析器知道怎么跳过空白和换行之类的无关字符。

图6：从源文档到解析树

解析过程是迭代的，解析器从词法分析器处取道一个新的符号，并试着用这个符号匹配一条语法规则，如果匹配了一条规则，这个符号对应的节点将被添加到解析树上，然后解析器请求另一个符号。如果没有匹配到规则，解析器将在内部保存该符号，并从词法分析器取下一个符号，直到所有内部保存的符号能够匹配一项语法规则。如果最终没有找到匹配的规则，解析器将抛出一个异常，这意味着文档无效或是包含语法错误。

转换 Translation

很多时候，解析树并不是最终结果。解析一般在转换中使用——将输入文档转换为另一种格式。编译就是个例子，编译器在将一段源码编译为机器码的时候，先将源码解析为解析树，然后将该树转换为一个机器码文档。

图7：编译流程

解析实例 Parsing example

图5中，我们从一个数学表达式构建了一个解析树，这里定义一个简单的数学语言来看下解析过程。

词汇表：我们的语言包括整数、加号及减号。

语法：

1. 该语言的语法基本单元包括表达式、term及操作符
2. 该语言可以包括多个表达式
3. 一个表达式定义为两个term通过一个操作符连接
4. 操作符可以是加号或减号
5. term可以是一个整数或一个表达式

现在来分析一下“2＋3－1”这个输入

第一个匹配规则的子字符串是“2”，根据规则5，它是一个term，第二个匹配的是“2＋3”，它符合第2条规则——一个操作符连接两个term，下一次匹配发生在输入的结束处。“2＋3－1”是一个表达式，因为我们已经知道“2＋3”是一个term，所以我们有了一个term紧跟着一个操作符及另一个term。“2＋＋”将不会匹配任何规则，因此是一个无效输入。

词汇表及语法的定义

词汇表通常利用正则表达式来定义。

例如上面的语言可以定义为：

INTEGER：0｜［1－9］［0－9］＊

PLUS：＋

MINUS：－

正如看到的，这里用正则表达式定义整数。

语法通常用BNF格式定义，我们的语言可以定义为：

expression :＝ term operation term

operation := PLUS | MINUS

term := INTEGER | expression

如果一个语言的文法是上下文无关的，则它可以用正则解析器来解析。对上下文无关文法的一个直观的定义是，该文法可以用BNF来完整的表达。可查看http://en.wikipedia.org/wiki/Context-free_grammar。

解析器类型 Types of parsers

有两种基本的解析器——自顶向下解析及自底向上解析。比较直观的解释是，自顶向下解析，查看语法的最高层结构并试着匹配其中一个；自底向上解析则从输入开始，逐步将其转换为语法规则，从底层规则开始直到匹配高层规则。

来看一下这两种解析器如何解析上面的例子：

自顶向下解析器从最高层规则开始——它先识别出“2＋3“，将其视为一个表达式，然后识别出”2＋3－1“为一个表达式（识别表达式的过程中匹配了其他规则，但出发点是最高层规则）。

自底向上解析会扫描输入直到匹配了一条规则，然后用该规则取代匹配的输入，直到解析完所有输入。部分匹配的表达式被放置在解析堆栈中。

Stack	Input
	2 + 3 – 1
term	+ 3 - 1
term operation	3 – 1
expression	- 1
expression operation	1
expression	

自底向上解析器称为shift reduce 解析器，因为输入向右移动（想象一个指针首先指向输入开始处，并向右移动），并逐渐简化为语法规则。

自动化解析 Generating parsers automatically

解析器生成器这个工具可以自动生成解析器，只需要指定语言的文法——词汇表及语法规则，它就可以生成一个解析器。创建一个解析器需要对解析有深入的理解，而且手动的创建一个由较好性能的解析器并不容易，所以解析生成器很有用。Webkit使用两个知名的解析生成器——用于创建语法分析器的Flex及创建解析器的Bison（你可能接触过Lex和Yacc）。Flex的输入是一个包含了符号定义的正则表达式，Bison的输入是用BNF格式表示的语法规则。

HTML解析器 HTML Parser

HTML解析器的工作是将html标识解析为解析树。

HTML文法定义 The HTML grammar definition

W3C组织制定规范定义了HTML的词汇表和语法。

非上下文无关文法 Not a context free grammar

正如在解析简介中提到的，上下文无关文法的语法可以用类似BNF的格式来定义。

不幸的是，所有的传统解析方式都不适用于html（当然我提出它们并不只是因为好玩，它们将用来解析css和js），html不能简单的用解析所需的上下文无关文法来定义。

Html 有一个正式的格式定义——DTD（Document Type Definition 文档类型定义）——但它并不是上下文无关文法，html更接近于xml，现在有很多可用的xml解析器，html有个xml的变体——xhtml，它们间的不同在于，html更宽容，它允许忽略一些特定标签，有时可以省略开始或结束标签。总的来说，它是一种soft语法，不像xml呆板、固执。

显然，这个看起来很小的差异却带来了很大的不同。一方面，这是html流行的原因——它的宽容使web开发人员的工作更加轻松，但另一方面，这也使很难去写一个格式化的文法。所以，html的解析并不简单，它既不能用传统的解析器解析，也不能用xml解析器解析。

HTML DTD

Html适用DTD格式进行定义，这一格式是用于定义SGML家族的语言，包括了对所有允许元素及它们的属性和层次关系的定义。正如前面提到的，htmlDTD并没有生成一种上下文无关文法。

DTD有一些变种，标准模式只遵守规范，而其他模式则包含了对浏览器过去所使用标签的支持，这么做是为了兼容以前内容。最新的标准DTD在http://www.w3.org/TR/html4/strict.dtd

DOM

输出的树，也就是解析树，是由DOM元素及属性节点组成的。DOM是文档对象模型的缩写，它是html文档的对象表示，作为html元素的外部接口供js等调用。

树的根是“document”对象。

DOM和标签基本是一一对应的关系，例如，如下的标签：

```
<html>
  <body>
    <p>
      Hello DOM
    </p>
    <div><img src=”example.png” /></div>
  </body>
</html>
```

将会被转换为下面的DOM树：

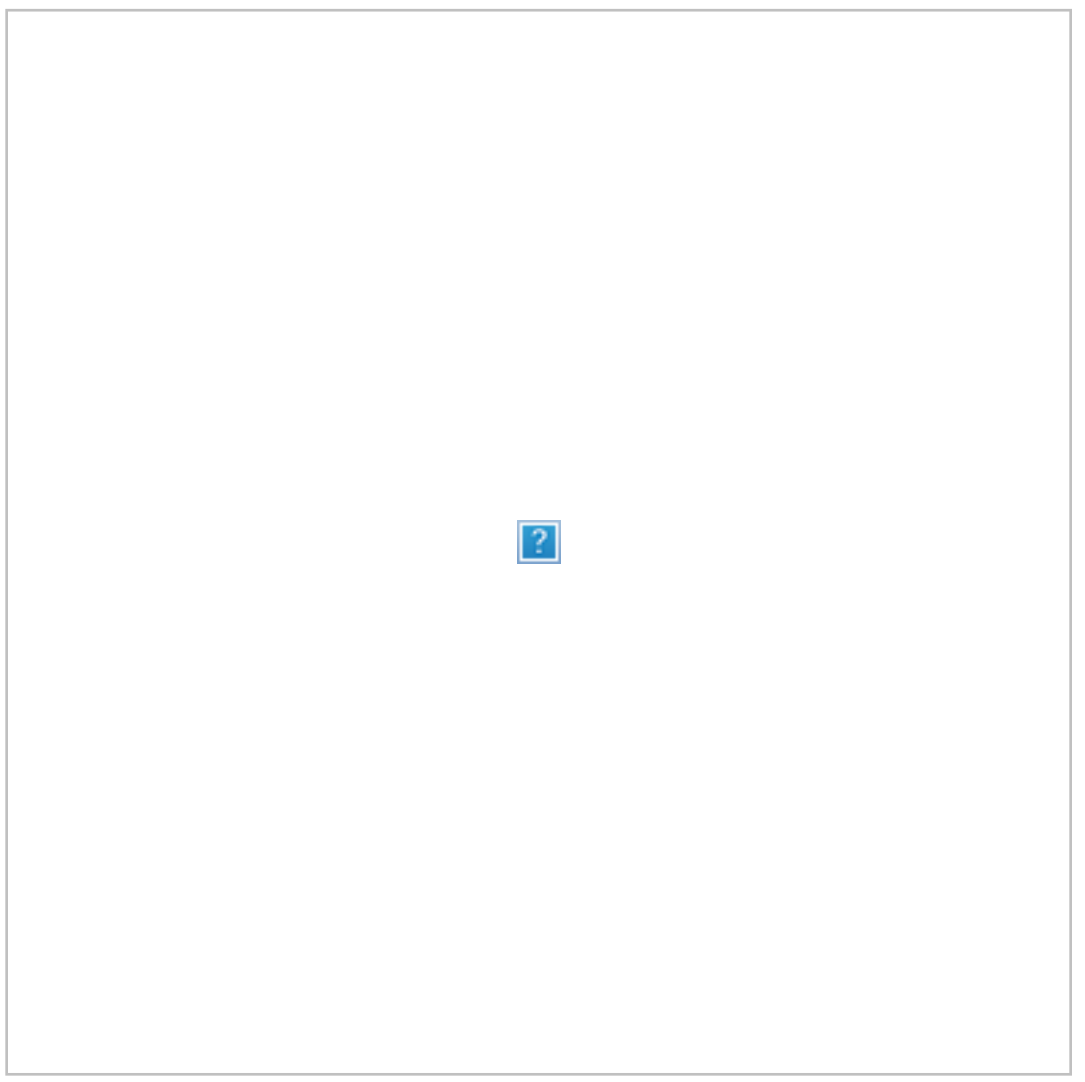


图8：示例标签对应的DOM树

和html一样，DOM的规范也是由W3C组织制定的。访问

http://www.w3.org/DOM/DOMTR，这是使用文档的一般规范。一个模型描述一种特定的html元素，可以在http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/idl-definitions.htm 查看html定义。

这里所谓的树包含了DOM节点是说树是由实现了DOM接口的元素构建而成的，浏览器使用已被浏览器内部使用的其他属性的具体实现。

解析算法 The parsing algorithm

正如前面章节中讨论的，hmtl不能被一般的自顶向下或自底向上的解析器所解析。

原因是：

1. 这门语言本身的宽容特性

2. 浏览器对一些常见的非法html有容错机制

3. 解析过程是往复的，通常源码不会在解析过程中发生改变，但在html中，脚本标签包含的“document.write ”可能添加标签，这说明在解析过程中实际上修改了输入

不能使用正则解析技术，浏览器为html定制了专属的解析器。

Html5规范中描述了这个解析算法，算法包括两个阶段——符号化及构建树。

符号化是词法分析的过程，将输入解析为符号，html的符号包括开始标签、结束标签、属性名及属性值。

符号识别器识别出符号后，将其传递给树构建器，并读取下一个字符，以识别下一个符号，这样直到处理完所有输入。

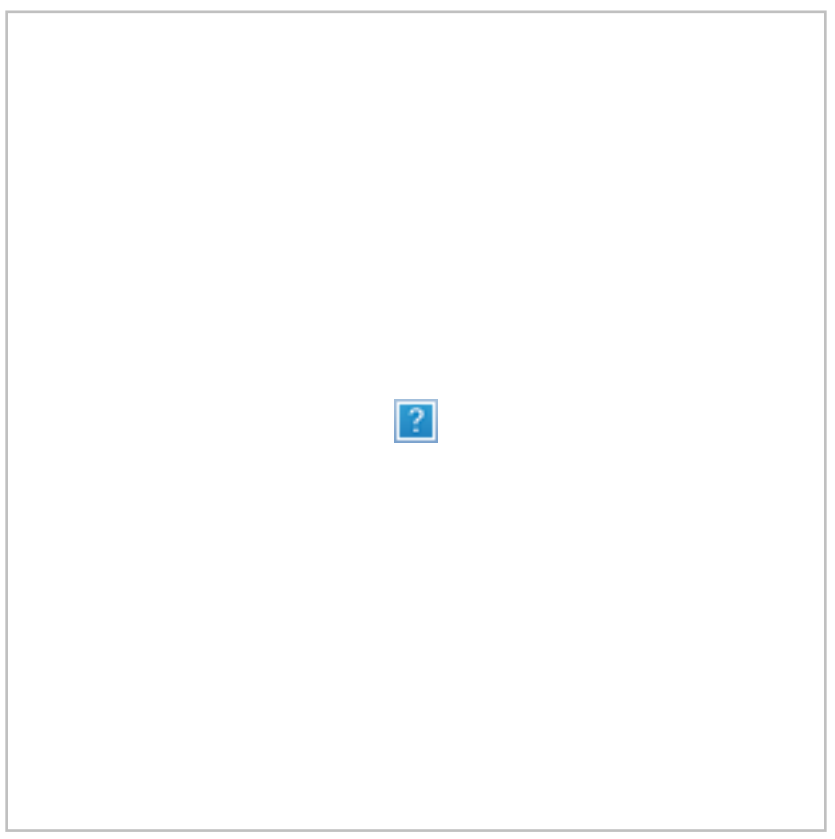


图9：HTML解析流程

符号识别算法 The tokenization algorithm

算法输出html符号，该算法用状态机表示。每次读取输入流中的一个或多个字符，并根据这些字符转移到下一个状态，当前的符号状态及构建树状态共同影响结果，这意味着，读取同样的字符，可能因为当前状态的不同，得到不同的结果以进入下一个正确的状态。

这个算法很复杂，这里用一个简单的例子来解释这个原理。

基本示例——符号化下面的html：

```
<html>
    <body>
        Helloworld
```

```
    </body>

</html>
```

初始状态为“Data State”，当遇到“<”字符，状态变为“Tag open state”，读取一个a－z的字符将产生一个开始标签符号，状态相应变为“Tag name state”，一直保持这个状态直到读取到“>”，每个字符都附加到这个符号名上，例子中创建的是一个html符号。

当读取到“>”，当前的符号就完成了，此时，状态回到“Data state”，“<body>”重复这一处理过程。到这里，html和body标签都识别出来了。现在，回到“Data state”，读取“Hello world”中的字符“H”将创建并识别出一个字符符号，这里会为“Hello world”中的每个字符生成一个字符符号。

这样直到遇到“</body>”中的“<”。现在，又回到了“Tag open state”，读取下一个字符“/”将创建一个闭合标签符号，并且状态转移到“Tag name state”，还是保持这一状态，直到遇到“>”。然后，产生一个新的标签符号并回到“Data state”。后面的“</html>”将和“</body>”一样处理。

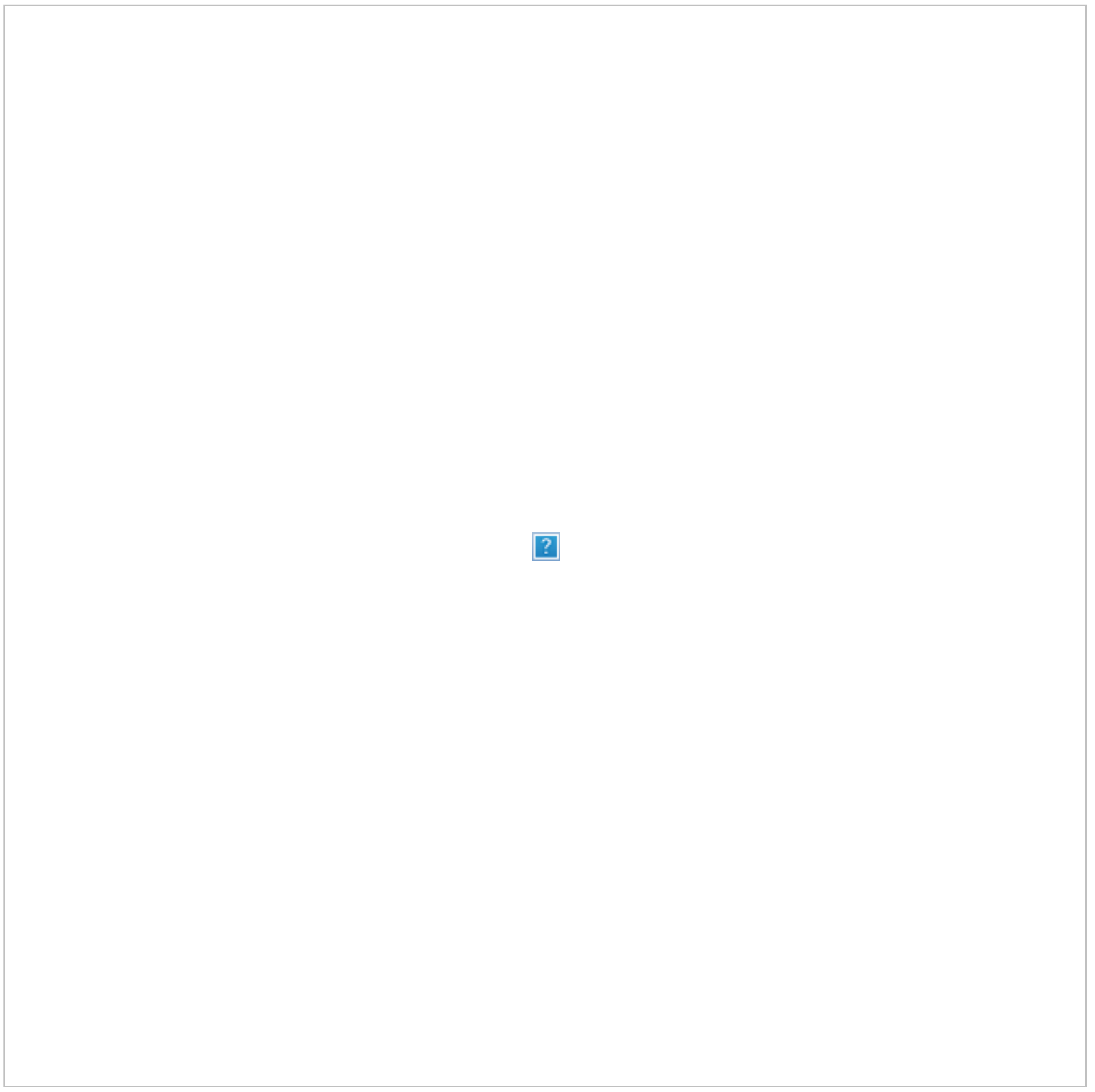


图10：符号化示例输入

树的构建算法 Tree construction algorithm

在树的构建阶段，将修改以Document为根的DOM树，将元素附加到树上。每个由符号识别器识别生成的节点将会被树构造器进行处理，规范中定义了每个符号相对应的Dom元素，对应的Dom元素将会被创建。这些元素除了会被添加到Dom树上，还将被添加到开放元素堆栈中。这个堆栈用来纠正嵌套的未匹配和未闭合标签，这个算法也是用状态机来描述，所有的状态采用插入模式。

来看一下示例中树的创建过程：

```
<html>
    <body>
        Helloworld
    </body>
</html>
```

构建树这一阶段的输入是符号识别阶段生成的符号序列。

首先是“initial mode”，接收到html符号后将转换为“before html”模式，在这个模式中对这个符号进行再处理。此时，创建了一个HTMLHtmlElement元素，并将其附加到根Document对象上。

状态此时变为“before head”，接收到body符号时，即使这里没有head符号，也将自动创建一个HTMLHeadElement元素并附加到树上。

现在，转到“in head”模式，然后是“after head”。到这里，body符号会被再次处理，将创建一个HTMLBodyElement并插入到树中，同时，转移到“in body”模式。

然后，接收到字符串“Hello world”的字符符号，第一个字符将导致创建并插入一个text节点，其他字符将附加到该节点。

接收到body结束符号时，转移到“afterbody”模式，接着接收到html结束符号，这个符号意味着转移到了“after after body”模式，当接收到文件结束符时，整个解析过程结束。

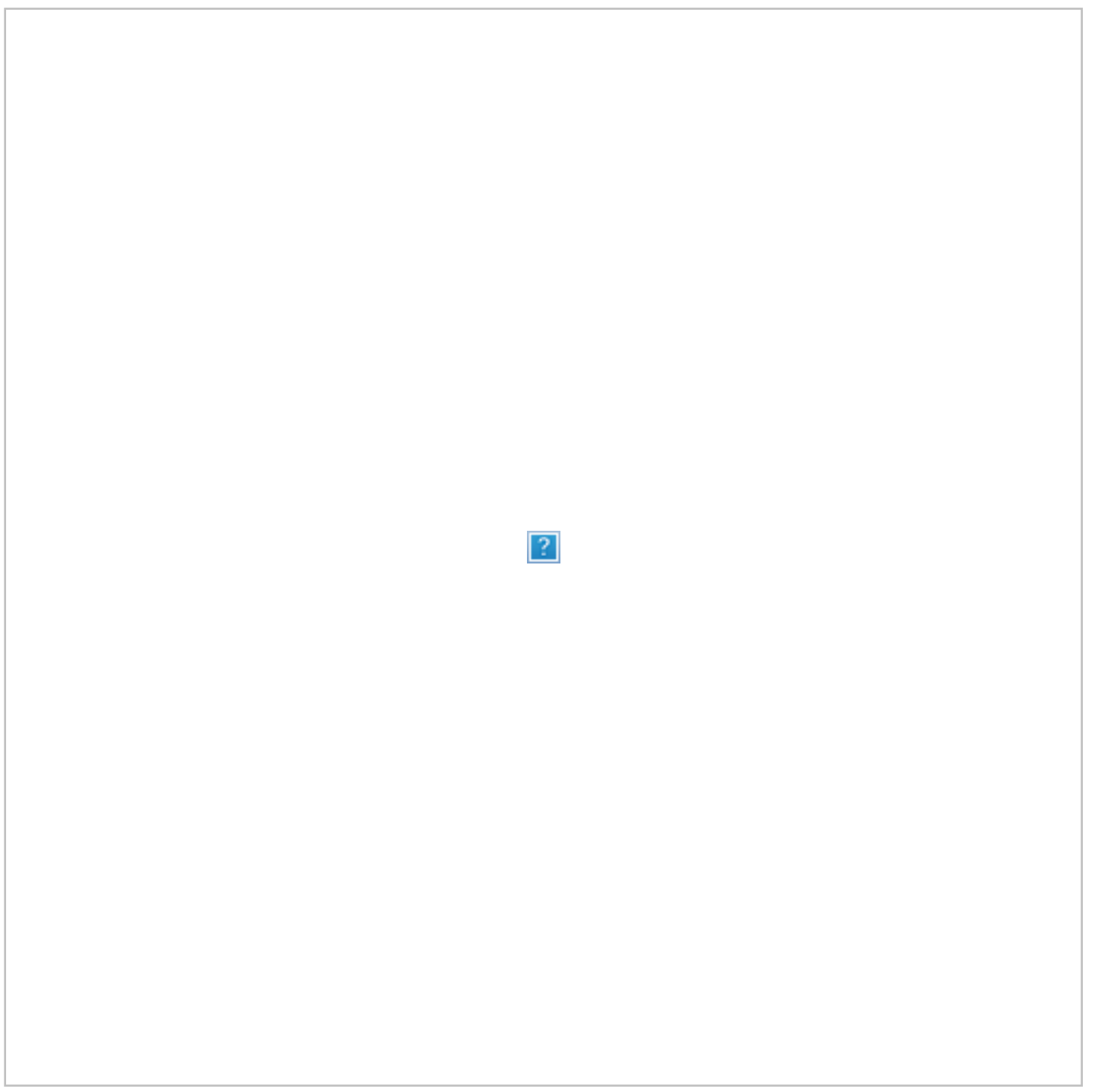


图11：示例html树的构建过程

解析结束时的处理 Action when the parsing is finished

在这个阶段，浏览器将文档标记为可交互的，并开始解析处于延时模式中的脚本——这些脚本在文档解析后执行。

文档状态将被设置为完成，同时触发一个load事件。

Html5规范中有符号化及构建树的完整算法(http://www.w3.org/TR/html5/syntax.html#html-parser)。

浏览器容错 Browsers error tolerance

你从来不会在一个html页面上看到“无效语法”这样的错误，浏览器修复了无效内容并继续工作。

以下面这段html为例：

```
<html>
 <mytag>
 </mytag>
 <div>
  <p>
 </div>
    Really lousy HTML
  </p>
</html>
```

这段html违反了很多规则（mytag不是合法的标签，p及div错误的嵌套等等），但是浏览器仍然可以没有任何怨言的继续显示，它在解析的过程中修复了html作者的错误。

浏览器都具有错误处理的能力，但是，另人惊讶的是，这并不是html最新规范的内容，就像书签及前进后退按钮一样，它只是浏览器长期发展的结果。一些比较知名的非法html结构，在许多站点中出现过，浏览器都试着以一种和其他浏览器一致的方式去修复。

Html5规范定义了这方面的需求，webkit在html解析类开始部分的注释中做了很好的总结。

解析器将符号化的输入解析为文档并创建文档，但不幸的是，我们必须处理很多没有很好格式化的html文档，至少要小心下面几种错误情况。

1. 在未闭合的标签中添加明确禁止的元素。这种情况下，应该先将前一标签闭合

2. 不能直接添加元素。有些人在写文档的时候会忘了中间一些标签（或者中间标签是可选的），比如HTML HEAD BODY TR TD LI等

3. 想在一个行内元素中添加块状元素。关闭所有的行内元素，直到下一个更高的块状元素

4. 如果这些都不行，就闭合当前标签直到可以添加该元素。

下面来看一些webkit容错的例子：

</br>替代

一些网站使用</br>替代
，为了兼容IE和Firefox，webkit将其看作
。

代码：

```
if (t->isCloseTag(brTag) &&m_document->inCompatMode()) {
    reportError(MalformedBRError);
    t->beginTag = true;
```

}

Note－这里的错误处理在内部进行，用户看不到。

迷路的表格

这指一个表格嵌套在另一个表格中，但不在它的某个单元格内。

比如下面这个例子：

```
<table>
    <table>
       <tr><td>innertable</td></tr>
      </table>
    <tr><td>outertable</td></tr>
</table>
```

webkit将会将嵌套的表格变为两个兄弟表格：

```
<table>
    <tr><td>outertable</td></tr>
</table>
<table>
    <tr><td>innertable</td></tr>
 </table>
```

代码：

```
if (m_inStrayTableContent && localName ==tableTag)
      popBlock(tableTag);
```

webkit使用堆栈存放当前的元素内容，它将从外部表格的堆栈中弹出内部的表格，则它们变为了兄弟表格。

嵌套的表单元素

用户将一个表单嵌套到另一个表单中，则第二个表单将被忽略。

代码：

```
if (!m_currentFormElement) {
      m_currentFormElement = new HTMLFormElement(formTag,
m_document);
}
```

太深的标签继承

www.liceo.edu.mx是一个由嵌套层次的站点的例子，最多只允许20个相同类型的标签嵌套，多出来的将被忽略。

代码：

```
bool HTMLParser::allowNestedRedundantTag(const
AtomicString&tagName)
{


unsigned i = 0;
```

```
for (HTMLStackElem* curr = m_blockStack;
        i< cMaxRedundantTagDepth && curr && curr->tagName ==tagName;
    curr =curr->next, i++) { }
return i != cMaxRedundantTagDepth;
}
```

放错了地方的html、body闭合标签

又一次不言自明。

支持不完整的html。我们从来不闭合body，因为一些愚蠢的网页总是在还未真正结束时就闭合它。我们依赖调用end方法去执行关闭的处理。

代码：

```
if (t->tagName == htmlTag || t->tagName ==bodyTag )
      return;
```

所以，web开发者要小心了，除非你想成为webkit容错代码的范例，否则还是写格式良好的html吧。

CSS解析 CSS parsing

还记得简介中提到的解析的概念吗，不同于html，css属于上下文无关文法，可以用前面所描述的解析器来解析。Css规范定义了css的词法及语法文法。

看一些例子：

每个符号都由正则表达式定义了词法文法（词汇表）：

```
comment      ///*[^*]*/*+([^/*][^*]*/*+)*//
num     [0-9]+|[0-9]*"."[0-9]+
nonascii [/200-/377]
nmstart      [_a-z]|{nonascii}|{escape}
nmchar       [_a-z0-9-]|{nonascii}|{escape}
name    {nmchar}+
ident       {nmstart}{nmchar}*
```

“ident”是识别器的缩写，相当于一个class名，“name”是一个元素id（用“#”引用）。

语法用BNF进行描述：

```
ruleset
  :selector [ ',' S* selector ]*
    '{' S*declaration [ ';' S* declaration ]* '}' S*
  ;
selector
  :simple_selector [ combinator selector | S+ [ combinator selector ] ]
  ;
simple_selector
  :element_name [ HASH | class | attrib | pseudo ]*
  | [ HASH| class | attrib | pseudo ]+
```

;

class

: '.'IDENT

;

element_name

: IDENT |'*'

;

attrib

: '[' S*IDENT S* [['=' | INCLUDES | DASHMATCH] S*

[IDENT| STRING] S*] ']'

;

pseudo

: ':' [IDENT | FUNCTION S* [IDENT S*] ')']

;

说明：一个规则集合有这样的结构

```
div.error , a.error {
    color:red;
    font-weight:bold;
}
```

div.error和a.error时选择器，大括号中的内容包含了这条规则集合中的规则，
这个结构在下面的定义中正式的定义了：

ruleset

:selector [',' S* selector]*

'{' S*declaration [';' S* declaration]* '}' S*

;

这说明，一个规则集合具有一个或是可选个数的多个选择器，这些选择器以逗号和空格（S表示空格）进行分隔。每个规则集合包含大括号及大括号中的一条或多条以分号隔开的声明。声明和选择器在后面进行定义。

Webkit CSS 解析器 Webkit CSS parser

Webkit使用Flex和Bison解析生成器从CSS语法文件中自动生成解析器。回忆一下解析器的介绍，Bison创建一个自底向上的解析器，Firefox使用自顶向下解析器。它们都是将每个css文件解析为样式表对象，每个对象包含css规则，css规则对象包含选择器和声明对象，以及其他一些符合css语法的对象。

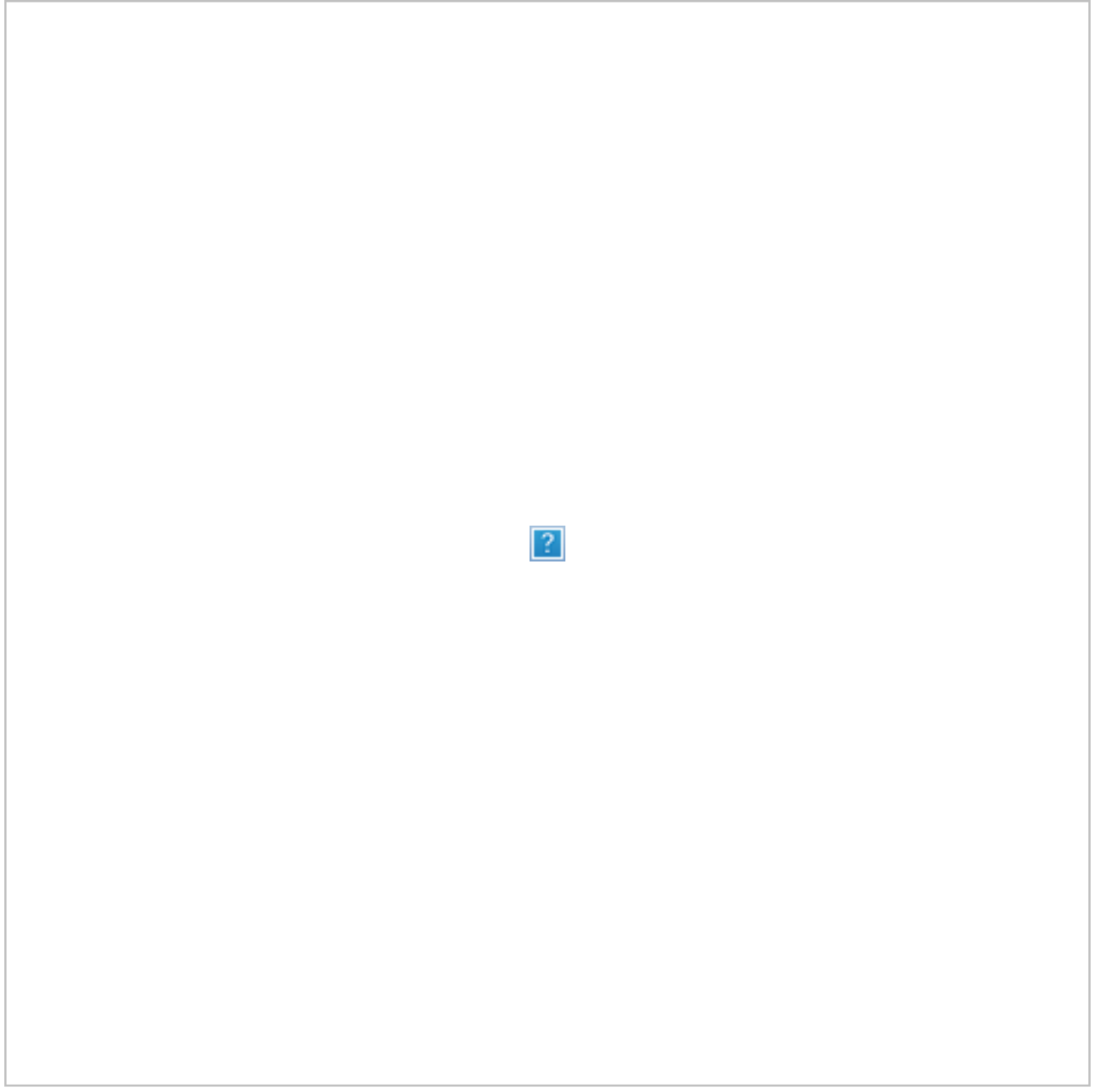


图12：解析css

脚本解析 Parsing scripts

本章将介绍JavaScript。

处理脚本及样式表的顺序 Theorder of processing scripts and style sheets

脚本

web的模式是同步的，开发者希望解析到一个script标签时立即解析执行脚本，并阻塞文档的解析直到脚本执行完。如果脚本是外引的，则网络必须先请求到这个资源——这个过程也是同步的，会阻塞文档的解析直到资源被请求到。这个模式保持了很多年，并且在html4及html5中都特别指定了。开发者可以将脚本标识为defer，以使其不阻塞文档解析，并在文档解析结束后执行。Html5增加了标记脚本为异步的选项，以使脚本的解析执行使用另一个线程。

预解析Speculative parsing

Webkit和Firefox都做了这个优化，当执行脚本时，另一个线程解析剩下的文档，并加载后面需要通过网络加载的资源。这种方式可以使资源并行加载从而使整体速度更快。需要注意的是，预解析并不改变Dom树，它将这个工作留给主解析过程，自己只解析外部资源的引用，比如外部脚本、样式表及图片。

样式表 Style sheets

样式表采用另一种不同的模式。理论上，既然样式表不改变Dom树，也就没有必要停下文档的解析等待它们，然而，存在一个问题，脚本可能在文档的解析过程中请求样式信息，如果样式还没有加载和解析，脚本将得到错误的值，显然这将会导致很多问题，这看起来是个边缘情况，但确实很常见。Firefox在存在样式表还在加载和解析时阻塞所有的脚本，而chrome只在当脚本试图访问某些可能被未加载的样式表所影响的特定的样式属性时才阻塞这些脚本。

渲染树的构造 Render tree construction

当Dom树构建完成时，浏览器开始构建另一棵树——渲染树。渲染树由元素显示序列中的可见元素组成，它是文档的可视化表示，构建这棵树是为了以正确的顺序绘制文档内容。

Firefox将渲染树中的元素称为frames，webkit则用renderer或渲染对象来描述这些元素。

一个渲染对象直到怎么布局及绘制自己及它的children。

RenderObject是Webkit的渲染对象基类，它的定义如下：

```
class RenderObject{
    virtualvoid layout();
    virtualvoid paint(PaintInfo);
    virtualvoid rect repaintRect();
    Node*node;  //the DOM node
    RenderStyle*style;  // the computed style
    RenderLayer*containgLayer; //the containing z-index layer
}
```

每个渲染对象用一个和该节点的css盒模型相对应的矩形区域来表示，正如css2所描述的那样，它包含诸如宽、高和位置之类的几何信息。盒模型的类型受该节点相关的display样式属性的影响（参考样式计算章节）。下面的webkit代码说明了如何根据display属性决定某个节点创建何种类型的渲染对象。

```
RenderObject* RenderObject::createObject(Node*node, RenderStyle* style)
{
  Document* doc = node->document();
  RenderArena* arena = doc->renderArena();
   ...
  RenderObject* o = 0;
switch(style->display()) {
  case NONE:
     break;
  case INLINE:
     o = new (arena) RenderInline(node);
     break;
  case BLOCK:
     o = new (arena) RenderBlock(node);
     break;
  case INLINE_BLOCK:
     o = new (arena) RenderBlock(node);
     break;
  case LIST_ITEM:
```

```
        o = new (arena) RenderListItem(node);
        break;
      ...
    }

    returno;
}
```

元素的类型也需要考虑，例如，表单控件和表格带有特殊的框架。

在webkit中，如果一个元素想创建一个特殊的渲染对象，它需要复写“createRenderer”方法，使渲染对象指向不包含几何信息的样式对象。

渲染树和Dom树的关系 The render tree relation to the DOMtree

渲染对象和Dom元素相对应，但这种对应关系不是一对一的，不可见的Dom元素不会被插入渲染树，例如head元素。另外，display属性为none的元素也不会在渲染树中出现（visibility属性为hidden的元素将出现在渲染树中）。

还有一些Dom元素对应几个可见对象，它们一般是一些具有复杂结构的元素，无法用一个矩形来描述。例如，select元素有三个渲染对象——一个显示区域、一个下拉列表及一个按钮。同样，当文本因为宽度不够而折行时，新行将作为额外的渲染元素被添加。另一个多个渲染对象的例子是不规范的html，根据css规范，一个行内元素只能仅包含行内元素或仅包含块状元素，在存在混合内容时，将会创建匿名的块状渲染对象包裹住行内元素。

一些渲染对象和所对应的Dom节点不在树上相同的位置，例如，浮动和绝对定位的元素在文本流之外，在两棵树上的位置不同，渲染树上标识出真实的结构，并用一个占位结构标识出它们原来的位置。

图12：渲染树及对应的Dom树

创建树的流程 The flow of constructing the tree

Firefox中，表述为一个监听Dom更新的监听器，将frame的创建委派给Frame Constructor，这个构建器计算样式（参看样式计算）并创建一个frame。

Webkit中，计算样式并生成渲染对象的过程称为attachment，每个Dom节点有一个attach方法，attachment的过程是同步的，调用新节点的attach方法将节点插入到Dom树中。

处理html和body标签将构建渲染树的根，这个根渲染对象对应被css规范称为containing block的元素——包含了其他所有块元素的顶级块元素。它的大小就是viewport——浏览器窗口的显示区域，Firefox称它为viewPortFrame，webkit称为RenderView，这个就是文档所指向的渲染对象，树中其他的部分都将作为一个插入的Dom节点被创建。

样式计算 Style Computation

创建渲染树需要计算出每个渲染对象的可视属性，这可以通过计算每个元素的样式属性得到。

样式包括各种来源的样式表，行内样式元素及html中的可视化属性（例如bgcolor），可视化属性转化为css样式属性。

样式表来源于浏览器默认样式表，及页面作者和用户提供的样式表——有些样式是浏览器用户提供的（浏览器允许用户定义喜欢的样式，例如，在Firefox

中，可以通过在Firefox Profile目录下放置样式表实现）。

计算样式的一些困难：

1. 样式数据是非常大的结构，保存大量的样式属性会带来内存问题

2. 如果不进行优化，找到每个元素匹配的规则会导致性能问题，为每个元素查找匹配的规则都需要遍历整个规则表，这个过程有很大的工作量。选择符可能有复杂的结构，匹配过程如果沿着一条开始看似正确，后来却被证明是无用的路径，则必须去尝试另一条路径。

例如，下面这个复杂选择符

div div div div {...}

这意味着规则应用到三个div的后代div元素，选择树上一条特定的路径去检查，这可能需要遍历节点树，最后却发现它只是两个div的后代，并不使用该规则，然后则需要沿着另一条路径去尝试

3. 应用规则涉及非常复杂的级联，它们定义了规则的层次

我们来看一下浏览器如何处理这些问题：

共享样式数据

webkit节点引用样式对象（渲染样式），某些情况下，这些对象可以被节点间共享，这些节点需要是兄弟或是表兄弟节点，并且：

1. 这些元素必须处于相同的鼠标状态（比如不能一个处于hover，而另一个不是）

2. 不能有元素具有id

3. 标签名必须匹配

4. class属性必须匹配

5. 对应的属性必须相同

6. 链接状态必须匹配

7. 焦点状态必须匹配

8. 不能有元素被属性选择器影响

9. 元素不能有行内样式属性

10. 不能有生效的兄弟选择器，webcore在任何兄弟选择器相遇时只是简单的抛出一个全局转换，并且在它们显示时使整个文档的样式共享失效，这些包括＋选择器和类似:first-child和:last-child这样的选择器。

Firefox规则树 Firefox rule tree

Firefox用两个树用来简化样式计算－规则树和样式上下文树，webkit也有样式对象，但它们并没有存储在类似样式上下文树这样的树中，只是由Dom节点指向其相关的样式。

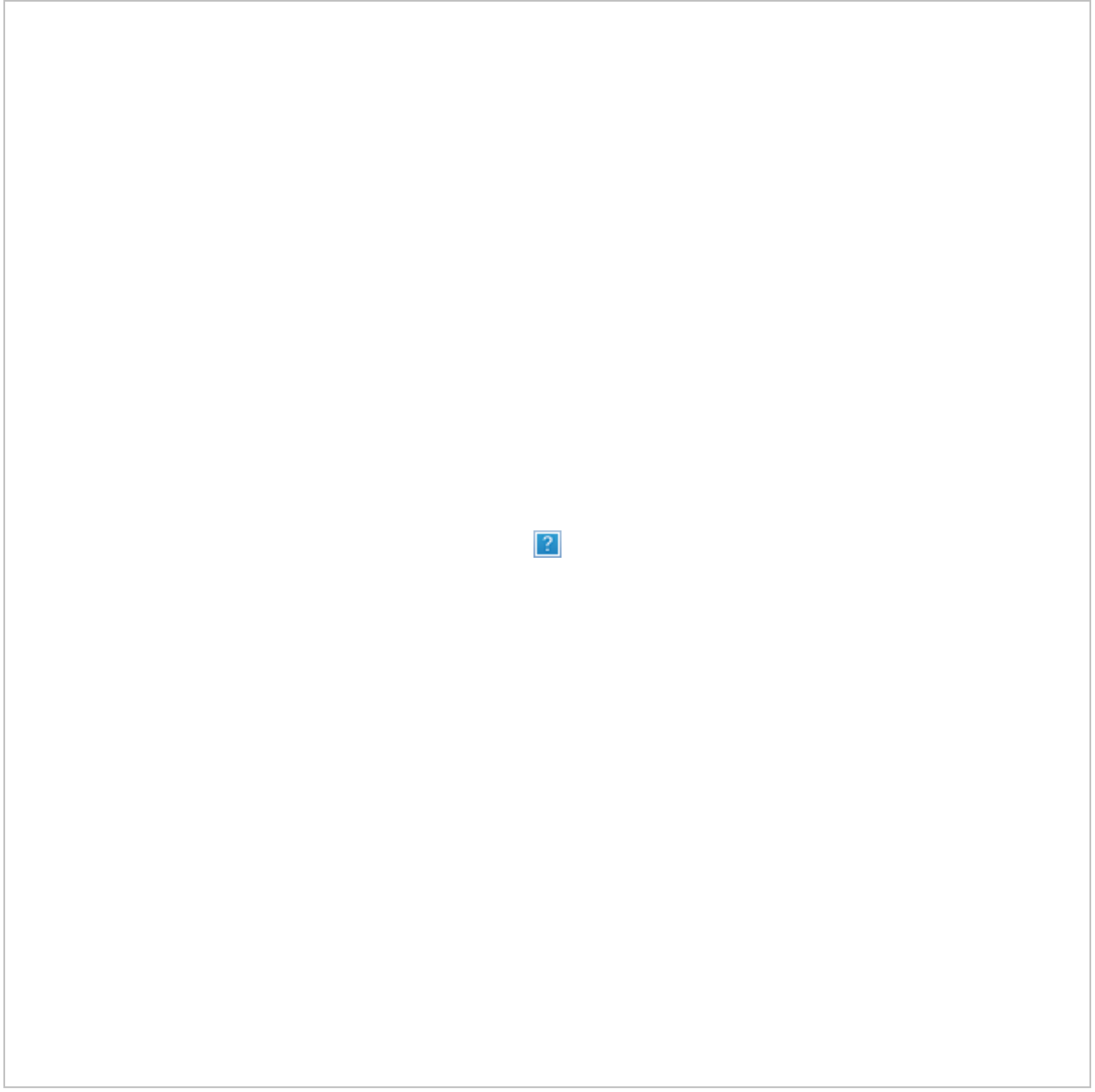
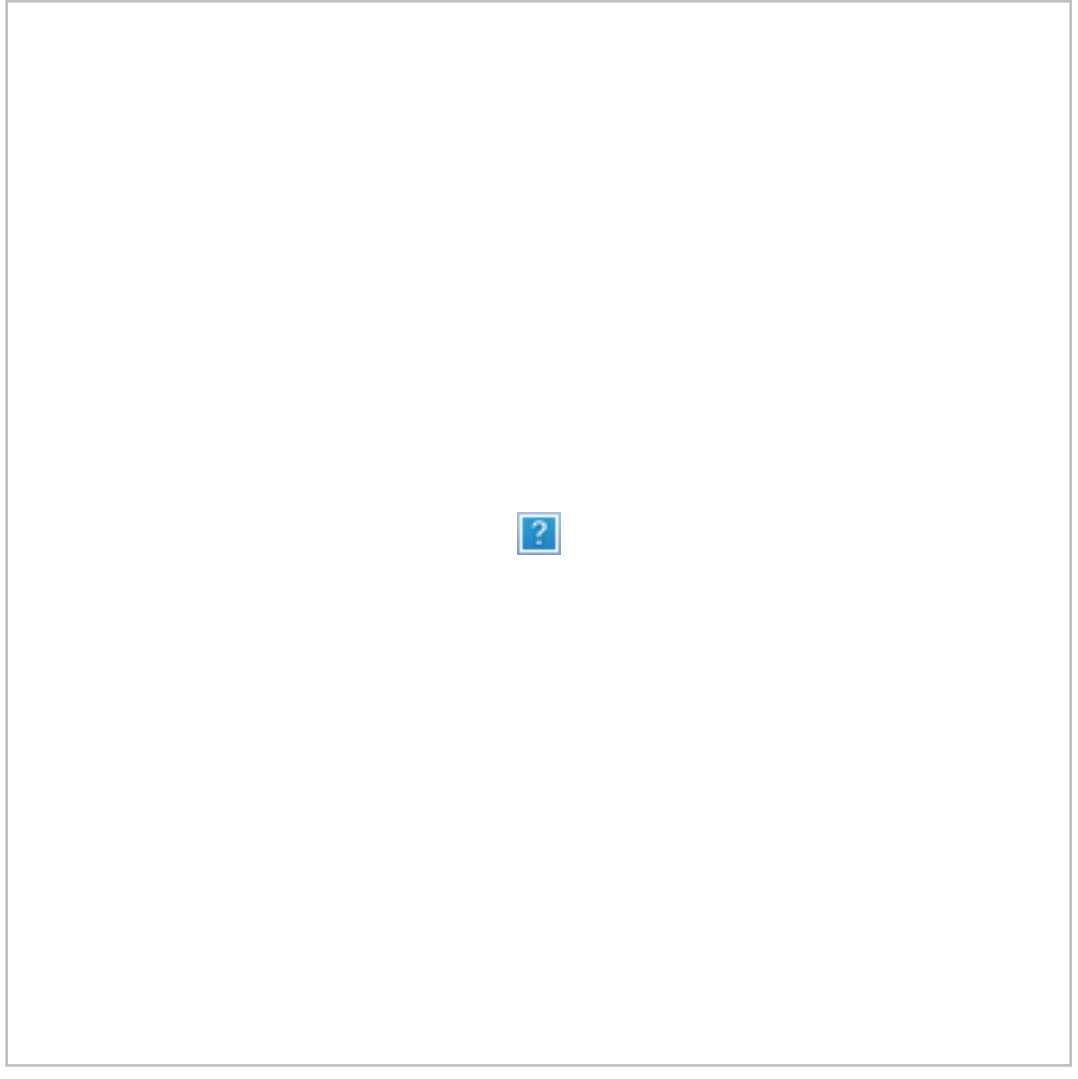


图14：Firefox样式上下文树

样式上下文包含最终值，这些值是通过以正确顺序应用所有匹配的规则，并将它们由逻辑值转换为具体的值，例如，如果逻辑值为屏幕的百分比，则通过计算将其转化为绝对单位。样式树的使用确实很巧妙，它使得在节点中共享的这些值不需要被多次计算，同时也节省了存储空间。

所有匹配的规则都存储在规则树中，一条路径中的底层节点拥有最高的优先级，这棵树包含了所找到的所有规则匹配的路径（译注：可以取巧理解为每条路径对应一个节点，路径上包含了该节点所匹配的所有规则）。规则树并不是一开始就为所有节点进行计算，而是在某个节点需要计算样式时，才进行相应的计算并将计算后的路径添加到树中。

我们将树上的路径看成辞典中的单词，假如已经计算出了如下的规则树：



假如需要为内容树中的另一个节点匹配规则，现在知道匹配的规则（以正确的顺序）为B-E-I，因为我们已经计算出了路径A-B-E-I-L，所以树上已经存在了这条路径，剩下的工作就很少了。

现在来看一下树如何保存。

结构化

样式上下文按结构划分，这些结构包括类似border或color这样的特定分类的样式信息。一个结构中的所有特性不是继承的就是非继承的，对继承的特性，除非元素自身有定义，否则就从它的parent继承。非继承的特性（称为reset特性）如果没有定义，则使用默认的值。

样式上下文树缓存完整的结构（包括计算后的值），这样，如果底层节点没有为一个结构提供定义，则使用上层节点缓存的结构。

使用规则树计算样式上下文

当为一个特定的元素计算样式时，首先计算出规则树中的一条路径，或是使用已经存在的一条，然后使用路径中的规则去填充新的样式上下文，从样式的底层节点开始，它具有最高优先级（通常是最特定的选择器），遍历规则树，直到填满结构。如果在那个规则节点没有定义所需的结构规则，则沿着路径向上，直到找到该结构规则。

如果最终没有找到该结构的任何规则定义，那么如果这个结构是继承型的，则找到其在内容树中的parent的结构，这种情况下，我们也成功的共享了结构；如果这个结构是reset型的，则使用默认的值。

如果特定的节点添加了值，那么需要做一些额外的计算以将其转换为实际值，然后在树上的节点缓存该值，使它的children可以使用。

当一个元素和它的一个兄弟元素指向同一个树节点时，完整的样式上下文可以被它们共享。

来看一个例子：假设有下面这段html

```
<html>
 <body>
  <div class="err"id="div1">
   <p>this is a
    <spanclass="big"> big error </span>
```

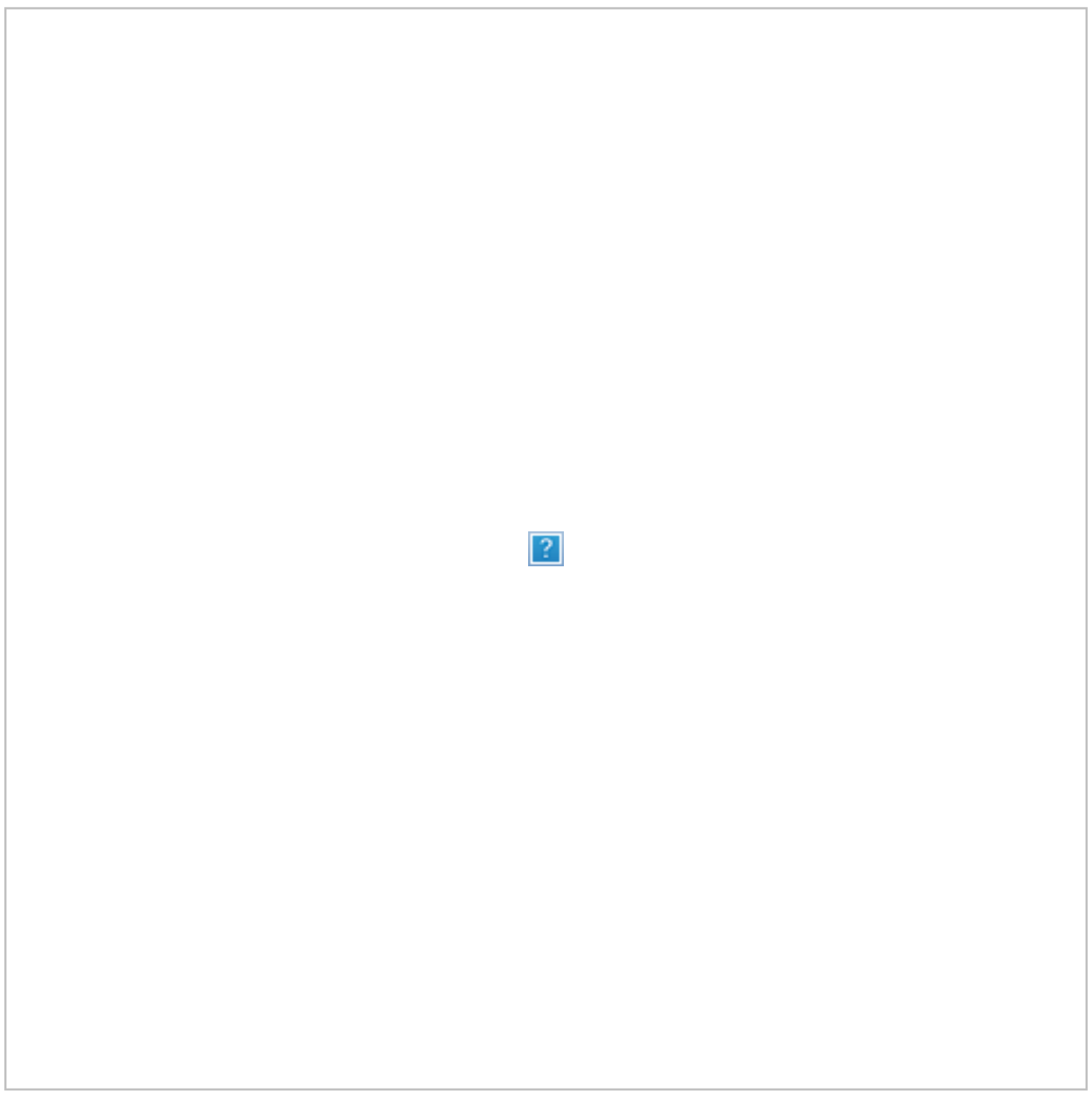
```
        this is also a
      <spanclass="big"> very  big  error</span>
        error
     </p>
    </div>
    <divclass="err" id="div2">another error</div>
  </body>
</html>
```

以及下面这些规则

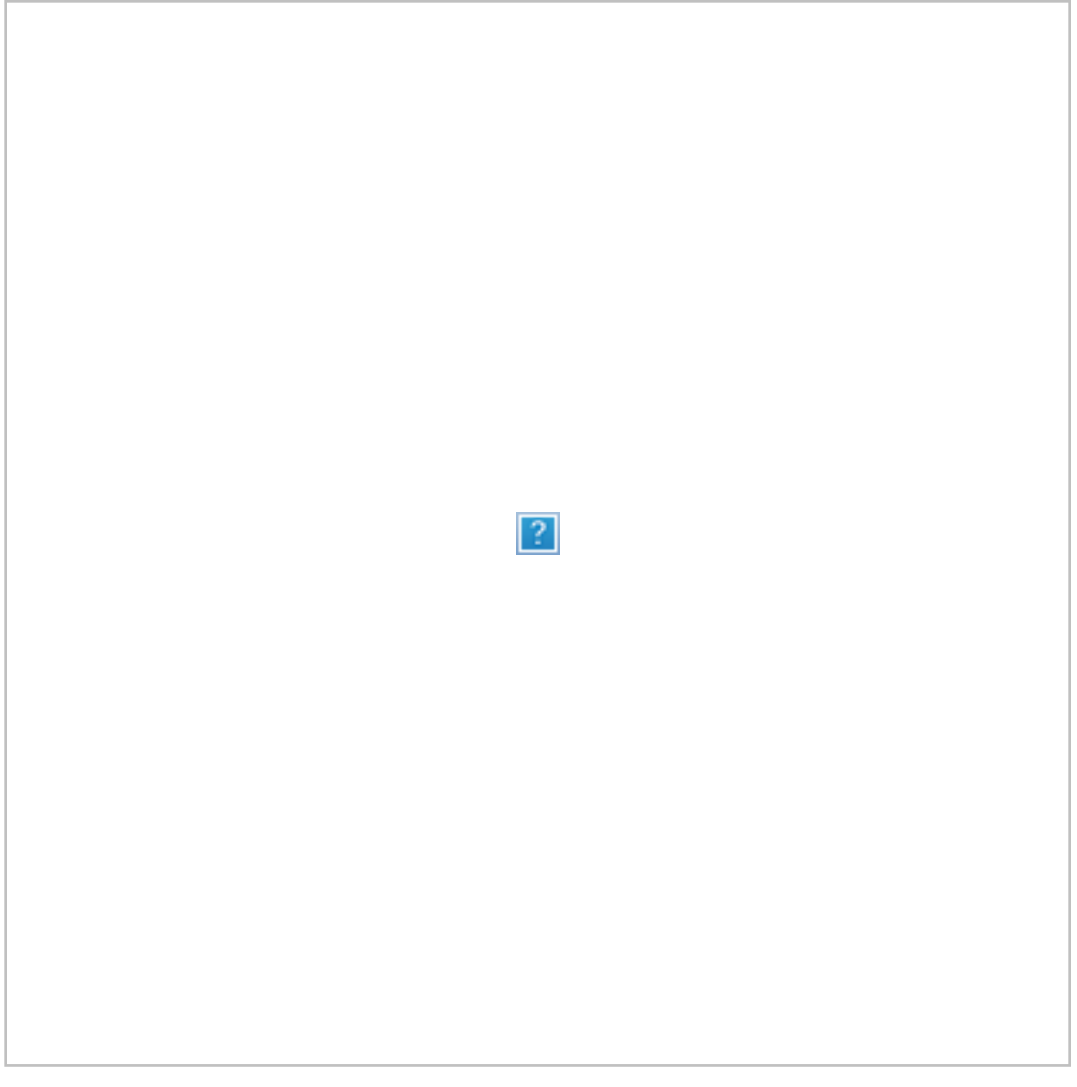
1. div{margin:5px;color:black}
2. .err{color:red}
3. .big{margin-top:3px}
4. divspan {margin-bottom:4px}
5. #div1{color:blue}
6. #div2{color:green}

简化下问题，我们只填充两个结构——color和margin，color结构只包含一个成员－颜色，margin结构包含四边。

生成的规则树如下（节点名：指向的规则）



上下文树如下（节点名：指向的规则节点）



假设我们解析html，遇到第二个div标签，我们需要为这个节点创建样式上下文，并填充它的样式结构。

我们进行规则匹配，找到这个div匹配的规则为1、2、6，我们发现规则树上已经存在了一条我们可以使用的路径1、2，我们只需为规则6新增一个节点添加到下面（就是规则树中的F）。

然后创建一个样式上下文并将其放到上下文树中，新的样式上下文将指向规则树中的节点F。

现在我们需要填充这个样式上下文，先从填充margin结构开始，既然最后一个规则节点没有添加margin结构，沿着路径向上，直到找到缓存的前面插入节点计算出的结构，我们发现B是最近的指定margin值的节点。因为已经有了color结构的定义，所以不能使用缓存的结构，既然color只有一个属性，也就不需要沿着路径向上填充其他属性。计算出最终值（将字符串转换为RGB等），并缓存计算后的结构。

第二个span元素更简单，进行规则匹配后发现它指向规则G，和前一个span一样，既然有兄弟节点指向同一个节点，就可以共享完整的样式上下文，只需指向前一个span的上下文。

因为结构中包含继承自parent的规则，上下文树做了缓存（color特性是继承来的，但Firefox将其视为reset并在规则树中缓存）。

例如，如果我们为一个paragraph的文字添加规则：

p {font-family:Verdana;fontsize:10px;font-weight:bold}

那么这个p在内容树中的子节点div，会共享和它parent一样的font结构，这种情况发生在没有为这个div指定font规则时。

Webkit中，并没有规则树，匹配的声明会被遍历四次，先是应用非important的高优先级属性（之所以先应用这些属性，是因为其他的依赖于它们－比如display），其次是高优先级important的，接着是一般优先级非important的，最后是一般优先级important的规则。这样，出现多次的属性将被按照正确的级联顺序进行处理，最后一个生效。

总结一下，共享样式对象（结构中完整或部分内容）解决了问题1和3，Firefox的规则树帮助以正确的顺序应用规则。

对规则进行处理以简化匹配过程

样式规则有几个来源：

- 外部样式表或style标签内的css规则
- 行内样式属性
- html可视化属性（映射为相应的样式规则）

后面两个很容易匹配到元素，因为它们所拥有的样式属性和html属性可以将元素作为key进行映射。

就像前面问题2所提到的，css的规则匹配可能很狡猾，为了解决这个问题，可以先对规则进行处理，以使其更容易被访问。

解析完样式表之后，规则会根据选择符添加一些hash映射，映射可以是根据id、class、标签名或是任何不属于这些分类的综合映射。如果选择符为id，规则将被添加到id映射，如果是class，则被添加到class映射，等等。

这个处理是匹配规则更容易，不需要查看每个声明，我们能从映射中找到一个元素的相关规则，这个优化使在进行规则匹配时减少了95＋%的工作量。

来看下面的样式规则：

p.error {color:red}

#messageDiv {height:50px}

div {margin:5px}

第一条规则将被插入class映射，第二条插入id映射，第三条是标签映射。

下面这个html片段：

an erroroccurred

this is amessage

我们首先找到p元素对应的规则，class映射将包含一个“error”的key，找到p.error的规则，div在id映射和标签映射中都有相关的规则，剩下的工作就是找出这些由key对应的规则中哪些确实是正确匹配的。

例如，如果div的规则是

table div {margin:5px}

这也是标签映射产生的，因为key是最右边的选择符，但它并不匹配这里的div元素，因为这里的div没有table祖先。

Webkit和Firefox都会做这个处理。

以正确的级联顺序应用规则

样式对象拥有对应所有可见属性的属性，如果特性没有被任何匹配的规则所定义，那么一些特性可以从parent的样式对象中继承，另外一些使用默认值。

这个问题的产生是因为存在不止一处的定义，这里用级联顺序解决这个问题。

样式表的级联顺序

一个样式属性的声明可能在几个样式表中出现，或是在一个样式表中出现多次，因此，应用规则的顺序至关重要，这个顺序就是级联顺序。根据css2的规范，级联顺序为（从低到高）：

1. 浏览器声明
2. 用户声明
3. 作者的一般声明

4. 作者的important声明

5. 用户important声明

浏览器声明是最不重要的，用户只有在声明被标记为important时才会覆盖作者的声明。具有同等级别的声明将根据specifity以及它们被定义时的顺序进行排序。Html可视化属性将被转换为匹配的css声明，它们被视为最低优先级的作者规则。

Specifity

Css2规范中定义的选择符specifity如下：

· 如果声明来自style属性，而不是一个选择器的规则，则计1，否则计0（＝a）

· 计算选择器中id属性的数量（＝b）

· 计算选择器中class及伪类的数量（＝c）

· 计算选择器中元素名及伪元素的数量（＝d）

连接a－b－c－d四个数量（用一个大基数的计算系统）将得到specifity。这里使用的基数由分类中最高的基数定义。例如，如果a为14，可以使用16进制。不同情况下，a为17时，则需要使用阿拉伯数字17作为基数，这种情况可能在这个选择符时发生html body div div ...（选择符中有17个标签，一般不太可能）。

一些例子：

```
*           {}  /* a=0 b=0 c=0 d=0 ->specificity = 0,0,0,0 */
 li          {} /* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,1 */
 li:first-line {}  /* a=0 b=0 c=0 d=2 -> specificity =0,0,0,2 */
 ul li        {} /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
 ulol+li      {}  /* a=0 b=0 c=0 d=3 -> specificity =0,0,0,3 */
 h1 +*[rel=up]{}  /* a=0 b=0 c=1 d=1 ->specificity = 0,0,1,1 */
 ul ol li.red  {}  /*a=0 b=0 c=1 d=3 -> specificity = 0,0,1,3 */
 li.red.level {}  /* a=0 b=0 c=2 d=1 ->specificity = 0,0,2,1 */
 #x34y        {} /* a=0 b=1 c=0 d=0 -> specificity = 0,1,0,0 */
 style=""         /* a=1 b=0 c=0 d=0 -> specificity= 1,0,0,0 */
```

规则排序

规则匹配后，需要根据级联顺序对规则进行排序，webkit先将小列表用冒泡排序，再将它们合并为一个大列表，webkit通过为规则复写“>”操作来执行排序：

```
static bool operator >(CSSRuleData& r1,CSSRuleData& r2)
{
   intspec1 = r1.selector()->specificity();
   intspec2 = r2.selector()->specificity();
   return(spec1 == spec2) : r1.position() > r2.position() : spec1 > spec2;
}
```

逐步处理 Gradual process

webkit使用一个标志位标识所有顶层样式表都已加载，如果在attch时样式没

有完全加载，则放置占位符，并在文档中标记，一旦样式表完成加载就重新进行计算。

布局 Layout

当渲染对象被创建并添加到树中，它们并没有位置和大小，计算这些值的过程称为layout或reflow。

Html使用基于流的布局模型，意味着大部分时间，可以以单一的途径进行几何计算。流中靠后的元素并不会影响前面元素的几何特性，所以布局可以在文档中从右向左、自上而下的进行。也存在一些例外，比如html tables。

坐标系统相对于根frame，使用top和left坐标。

布局是一个递归的过程，由根渲染对象开始，它对应html文档元素，布局继续递归的通过一些或所有的frame层级，为每个需要几何信息的渲染对象进行计算。

根渲染对象的位置是0,0，它的大小是viewport－浏览器窗口的可见部分。

所有的渲染对象都有一个layout或reflow方法，每个渲染对象调用需要布局的children的layout方法。

Dirty bit 系统

为了不因为每个小变化都全部重新布局，浏览器使用一个dirty bit系统，一个渲染对象发生了变化或是被添加了，就标记它及它的children为dirty－需要layout。存在两个标识－dirty及children are dirty，children are dirty说明即使这个渲染对象可能没问题，但它至少有一个child需要layout。

全局和增量 layout

当layout在整棵渲染树触发时，称为全局layout，这可能在下面这些情况下发生：

1. 一个全局的样式改变影响所有的渲染对象，比如字号的改变
2. 窗口resize

layout也可以是增量的，这样只有标志为dirty的渲染对象会重新布局（也将导致一些额外的布局）。增量 layout会在渲染对象dirty时异步触发，例如，当网络接收到新的内容并添加到Dom树后，新的渲染对象会添加到渲染树中。

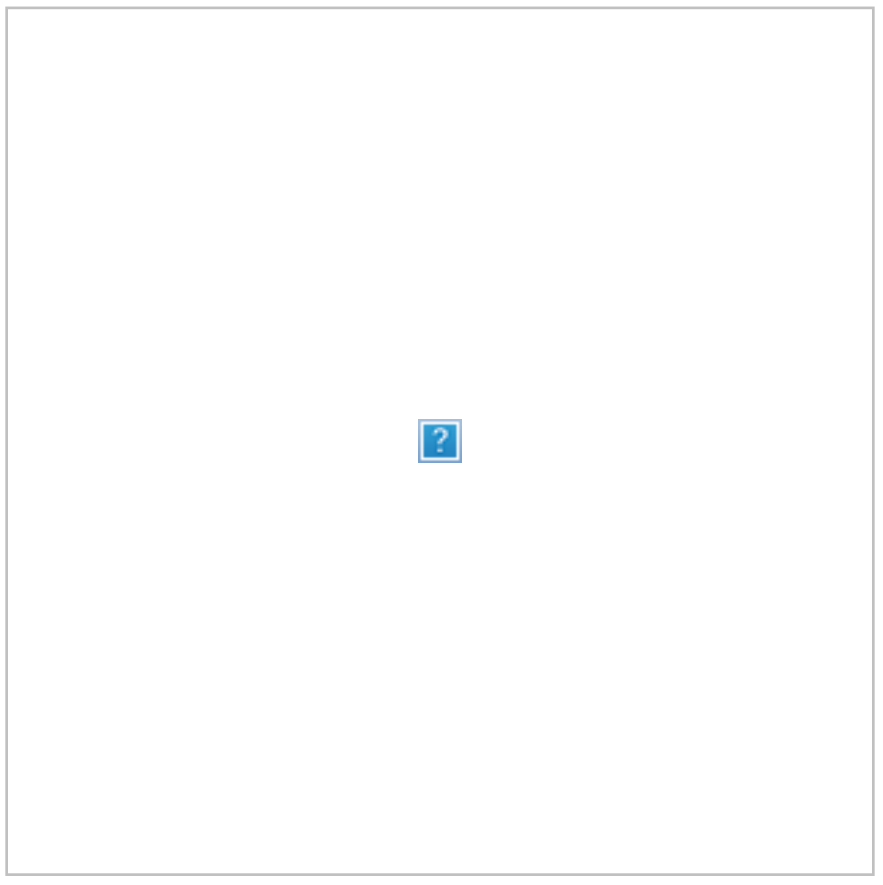


图20：增量 layout

异步和同步layout

增量layout的过程是异步的，Firefox为增量layout生成了reflow队列，以及一

个调度执行这些批处理命令。Webkit也有一个计时器用来执行增量layout－遍历树，为dirty状态的渲染对象重新布局。

另外，当脚本请求样式信息时，例如“offsetHeight”，会同步的触发增量布局。

全局的layout一般都是同步触发。

有些时候，layout会被作为一个初始layout之后的回调，比如滑动条的滑动。

优化

当一个layout因为resize或是渲染位置改变（并不是大小改变）而触发时，渲染对象的大小将会从缓存中读取，而不会重新计算。

一般情况下，如果只有子树发生改变，则layout并不从根开始。这种情况发生在，变化发生在元素自身并且不影响它周围元素，例如，将文本插入文本域（否则，每次击键都将触发从根开始的重排）。

layout过程

layout一般有下面这几个部分：

1. parent渲染对象决定它的宽度

2. parent渲染对象读取childiren，并：

1. 放置child渲染对象（设置它的x和y）

2. 在需要时（它们当前为dirty或是处于全局layout或者其他原因）调用child渲染对象的layout，这将计算child的高度

3. parent渲染对象使用child渲染对象的累积高度，以及margin和padding的高度来设置自己的高度－这将被parent渲染对象的parent使用

4. 将dirty标识设置为false

Firefox使用一个“state”对象（nsHTMLReflowState）做为参数去布局（firefox称为reflow），state包含parent的宽度及其他内容。

Firefox布局的输出是一个“metrics”对象（nsHTMLReflowMetrics）。它包括渲染对象计算出的高度。

宽度计算

渲染对象的宽度使用容器的宽度、渲染对象样式中的宽度及margin、border进行计算。例如，下面这个div的宽度：

webkit中宽度的计算过程是（RenderBox类的calcWidth方法）：

· 容器的宽度是容器的可用宽度和0中的最大值，这里的可用宽度为：contentWidth=clientWidth()-paddingLeft()-paddingRight()，clientWidth和clientHeight代表一个对象内部的不包括border和滑动条的大小

· 元素的宽度指样式属性width的值，它可以通过计算容器的百分比得到一个绝对值

· 加上水平方向上的border和padding

到这里是最佳宽度的计算过程，现在计算宽度的最大值和最小值，如果最佳宽度大于最大宽度则使用最大宽度，如果小于最小宽度则使用最小宽度。最后缓存这个值，当需要layout但宽度未改变时使用。

Line breaking

当一个渲染对象在布局过程中需要折行时，则暂停并告诉它的parent它需要折

行，parent将创建额外的渲染对象并调用它们的layout。

绘制 Painting

绘制阶段，遍历渲染树并调用渲染对象的paint方法将它们的内容显示在屏幕上，绘制使用UI基础组件，这在UI的章节有更多的介绍。

全局和增量

和布局一样，绘制也可以是全局的－绘制完整的树－或增量的。在增量的绘制过程中，一些渲染对象以不影响整棵树的方式改变，改变的渲染对象使其在屏幕上的矩形区域失效，这将导致操作系统将其看作dirty区域，并产生一个paint事件，操作系统很巧妙的处理这个过程，并将多个区域合并为一个。Chrome中，这个过程更复杂些，因为渲染对象在不同的进程中，而不是在主进程中。Chrome在一定程度上模拟操作系统的行为，表现为监听事件并派发消息给渲染根，在树中查找到相关的渲染对象，重绘这个对象（往往还包括它的children）。

绘制顺序

css2定义了绘制过程的顺序－http://www.w3.org/TR/CSS21/zindex.html。这个就是元素压入堆栈的顺序，这个顺序影响着绘制，堆栈从后向前进行绘制。

一个块渲染对象的堆栈顺序是：

1. 背景色
2. 背景图
3. border
4. children
5. outline

Firefox显示列表

Firefox读取渲染树并为绘制的矩形创建一个显示列表，该列表以正确的绘制顺序包含这个矩形相关的渲染对象。

用这样的方法，可以使重绘时只需查找一次树，而不需要多次查找——绘制所有的背景、所有的图片、所有的border等等。

Firefox优化了这个过程，它不添加会被隐藏的元素，比如元素完全在其他不透明元素下面。

Webkit矩形存储

重绘前，webkit将旧的矩形保存为位图，然后只绘制新旧矩形的差集。

动态变化

浏览器总是试着以最小的动作响应一个变化，所以一个元素颜色的变化将只导致该元素的重绘，元素位置的变化将大致元素的布局和重绘，添加一个Dom节点，也会大致这个元素的布局和重绘。一些主要的变化，比如增加html元素的字号，将会导致缓存失效，从而引起整数的布局和重绘。

渲染引擎的线程

渲染引擎是单线程的，除了网络操作以外，几乎所有的事情都在单一的线程中处理，在Firefox和Safari中，这是浏览器的主线程，Chrome中这是tab的主线程。

网络操作由几个并行线程执行，并行连接的个数是受限的（通常是2－6个）。

事件循环

浏览器主线程是一个事件循环，它被设计为无限循环以保持执行过程的可用，等待事件（例如layout和paint事件）并执行它们。下面是Firefox的主要事件循环代码。

```
while (!mExiting)
  NS_ProcessNextEvent(thread);
```

CSS2 可视模型 CSS2 visual module

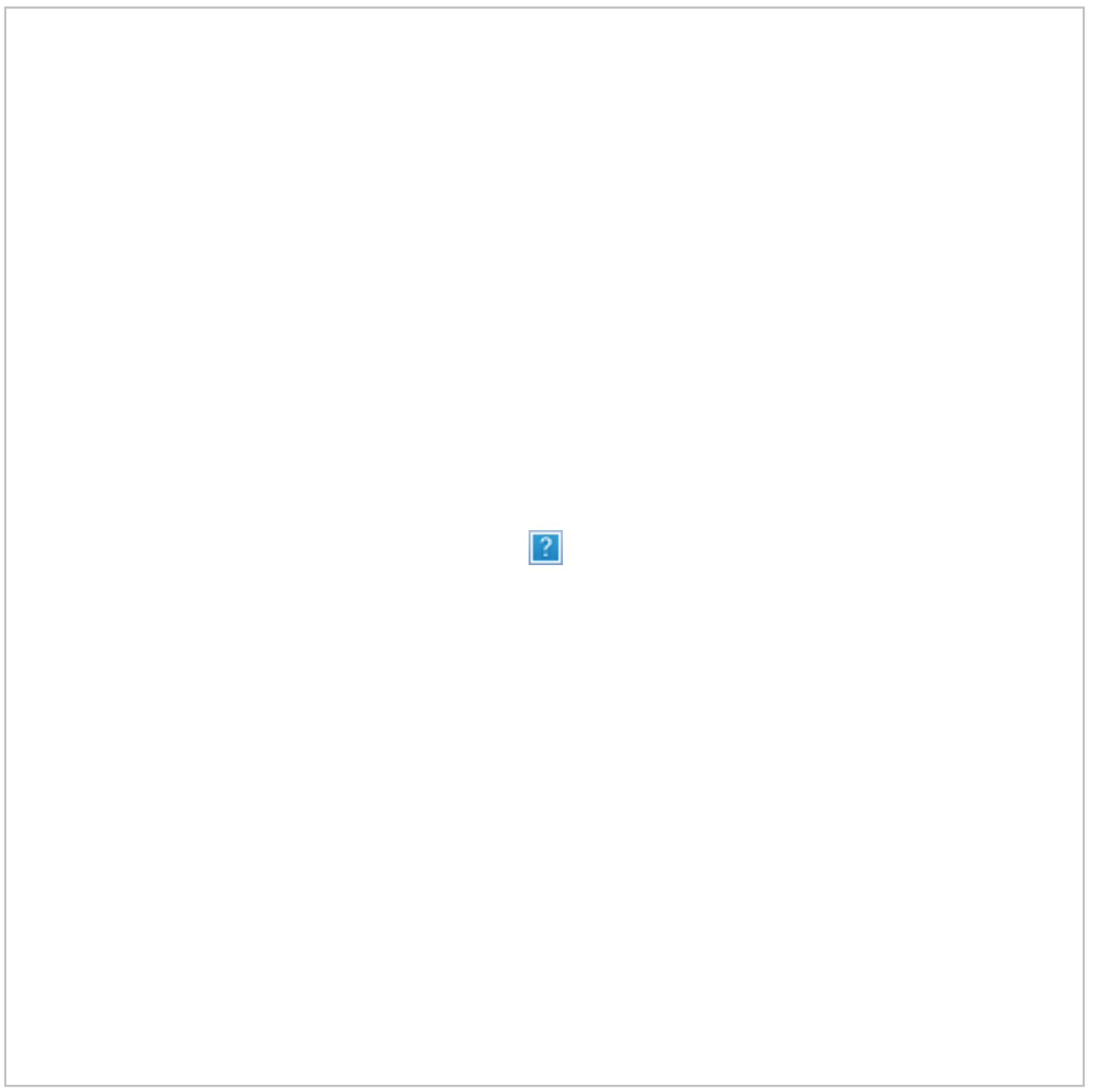
画布The Canvas

根据CSS2规范，术语canvas用来描述格式化的结构所渲染的空间——浏览器绘制内容的地方。画布对每个维度空间都是无限大的，但浏览器基于viewport的大小选择了一个初始宽度。

根据http://www.w3.org/TR/CSS2/zindex.html的定义，画布如果是包含在其他画布内则是透明的，否则浏览器会指定一个颜色。

CSS盒模型

CSS盒模型描述了矩形盒，这些矩形盒是为文档树中的元素生成的，并根据可视的格式化模型进行布局。每个box包括内容区域（如图片、文本等）及可选的四周padding、border和margin区域。



每个节点生成0－n个这样的box。

所有的元素都有一个display属性，用来决定它们生成box的类型，例如：

block－生成块状box

inline－生成一个或多个行内box

none－不生成box

默认的是inline，但浏览器样式表设置了其他默认值，例如，div元素默认为block。可以访问http://www.w3.org/TR/CSS2/sample.html查看更多的默认样式表示例。

定位策略 Position scheme

这里有三种策略：

1. normal－对象根据它在文档的中位置定位，这意味着它在渲染树和在Dom树中位置一致，并根据它的盒模型和大小进行布局
2. float－对象先像普通流一样布局，然后尽可能的向左或是向右移动
3. absolute－对象在渲染树中的位置和Dom树中位置无关

static和relative是normal，absolute和fixed属于absolute。

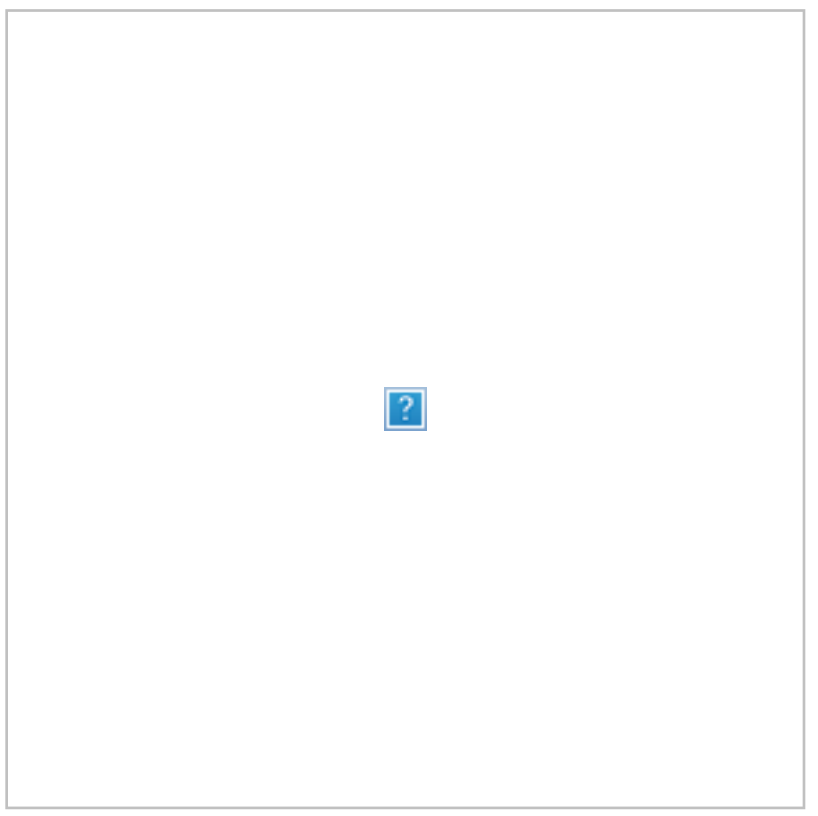
在static定位中，不定义位置而使用默认的位置。其他策略中，作者指定位置——top、bottom、left、right。

Box布局的方式由这几项决定：box的类型、box的大小、定位策略及扩展信息（比如图片大小和屏幕尺寸）。

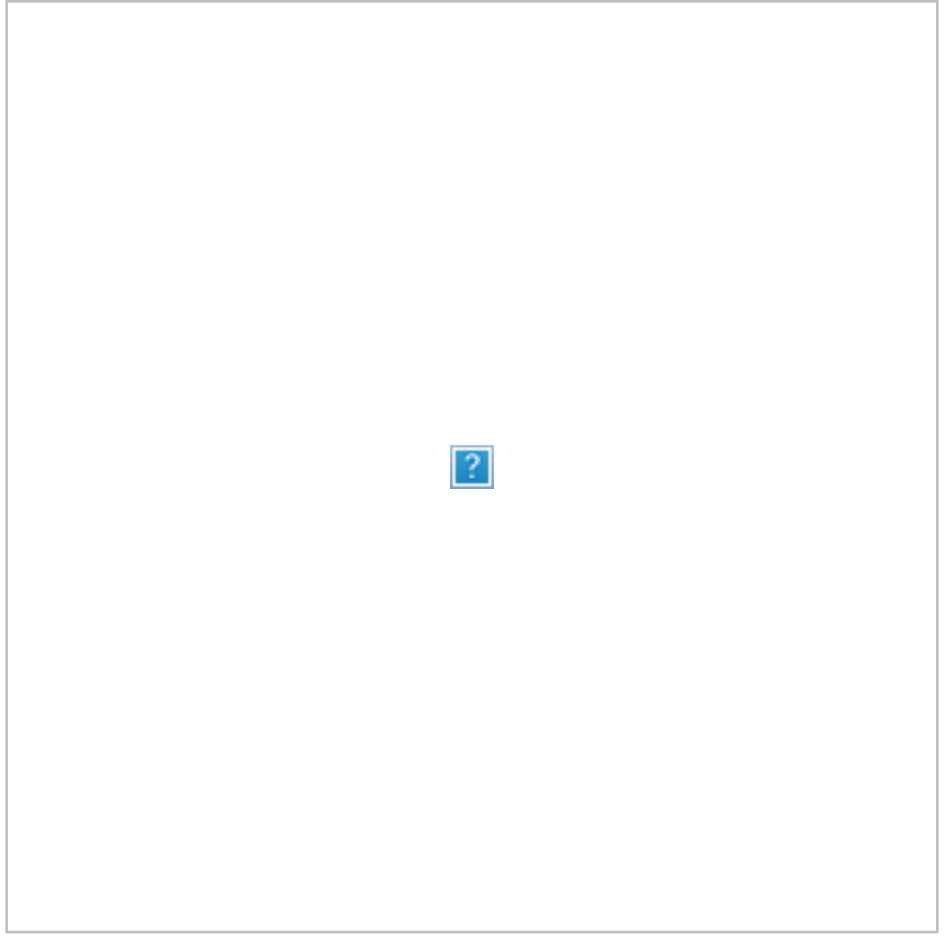
Box类型

Block box：构成一个块，即在浏览器窗口上有自己的矩形

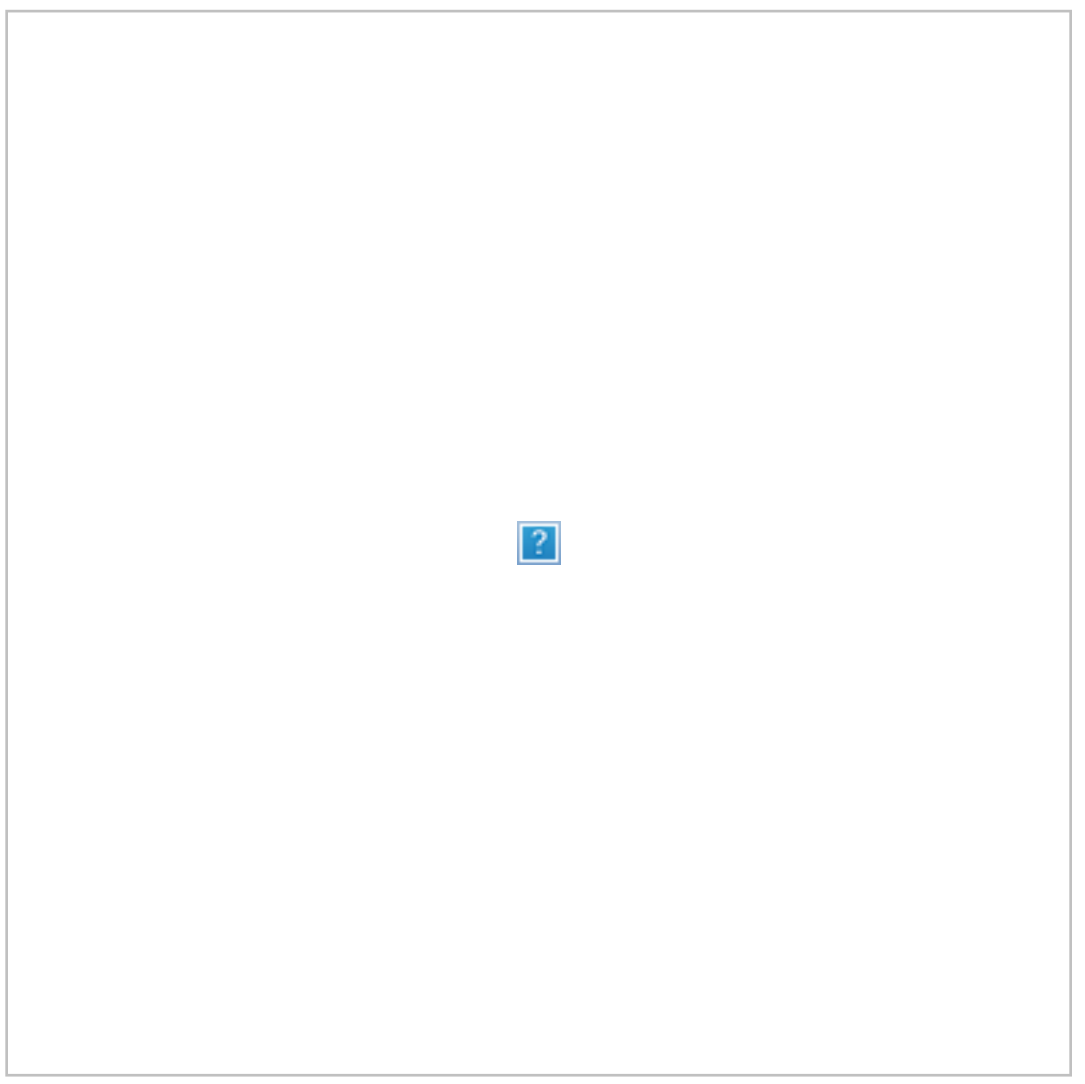
Inline box：并没有自己的块状区域，但包含在一个块状区域内



block一个挨着一个垂直格式化，inline则在水平方向上格式化。



Inline盒模型放置在行内或是line box中，每行至少和最高的box一样高，当box以baseline对齐时——即一个元素的底部和另一个box上除底部以外的某点对齐，行高可以比最高的box高。当容器宽度不够时，行内元素将被放到多行中，这在一个p元素中经常发生。



定位 Position

Relative

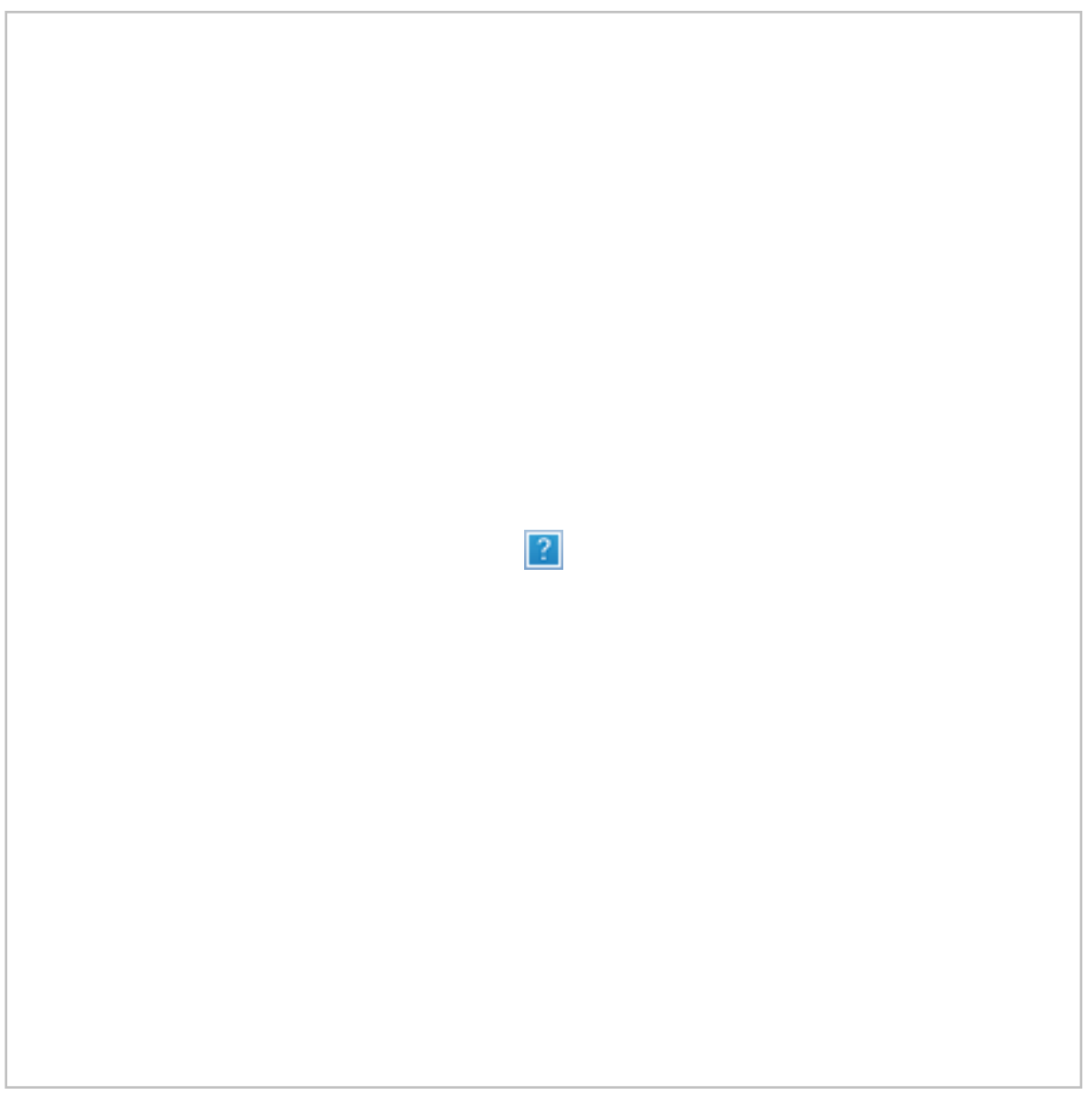
相对定位——先按照一般的定位，然后按所要求的差值移动。

Floats

一个浮动的box移动到一行的最左边或是最右边，其余的box围绕在它周围。
下面这段html：

<img style="float:right"src="images/image.gif"
width="100"height="100">Lorem ipsum dolor sit amet, consectetuer...

将显示为：



Absolute和Fixed

这种情况下的布局完全不顾普通的文档流，元素不属于文档流的一部分，大

小取决于容器。Fixed时，容器为viewport（可视区域）。

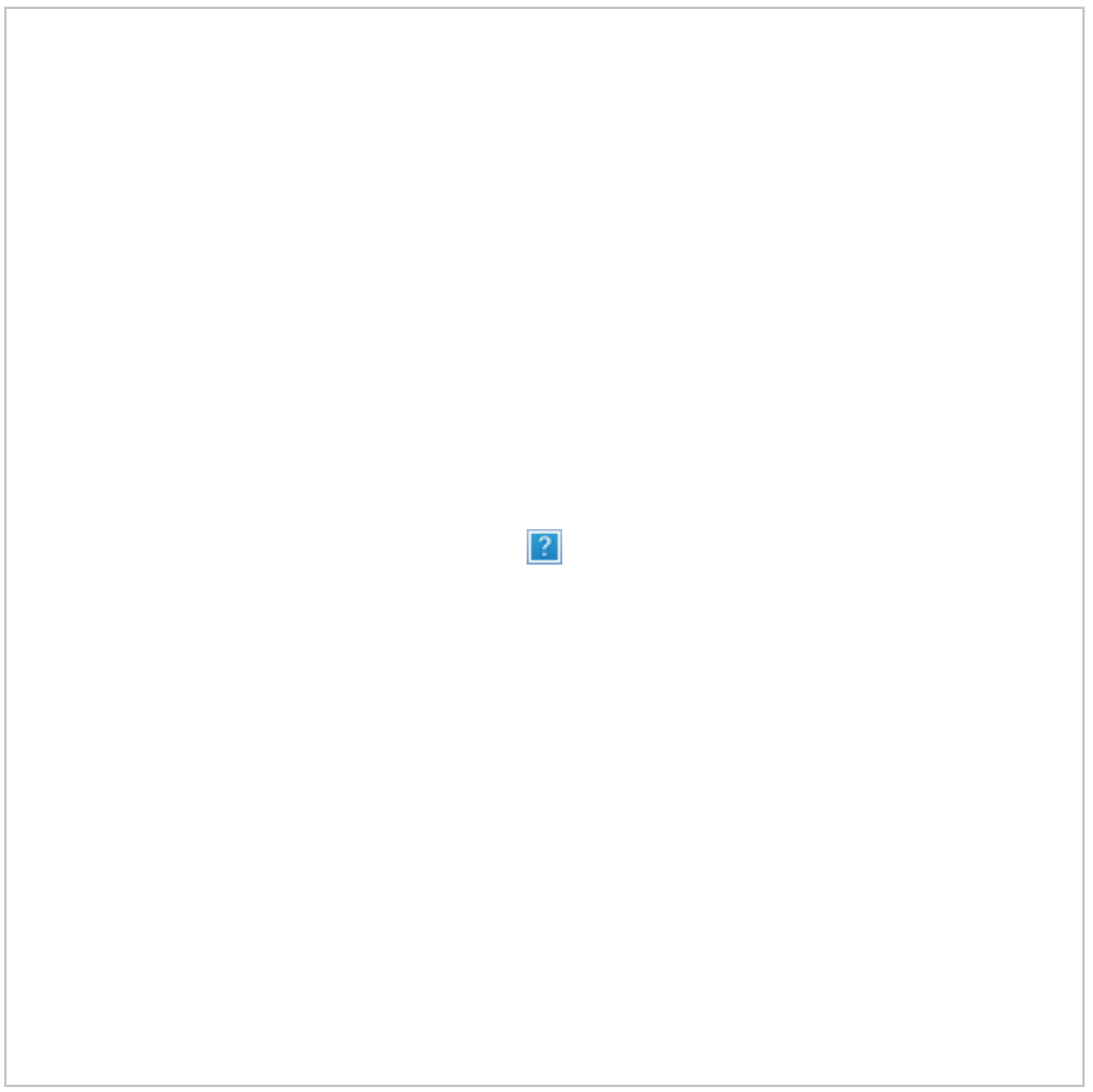


图17：fixed

注意－fixed即使在文档流滚动时也不会移动。

Layered representation

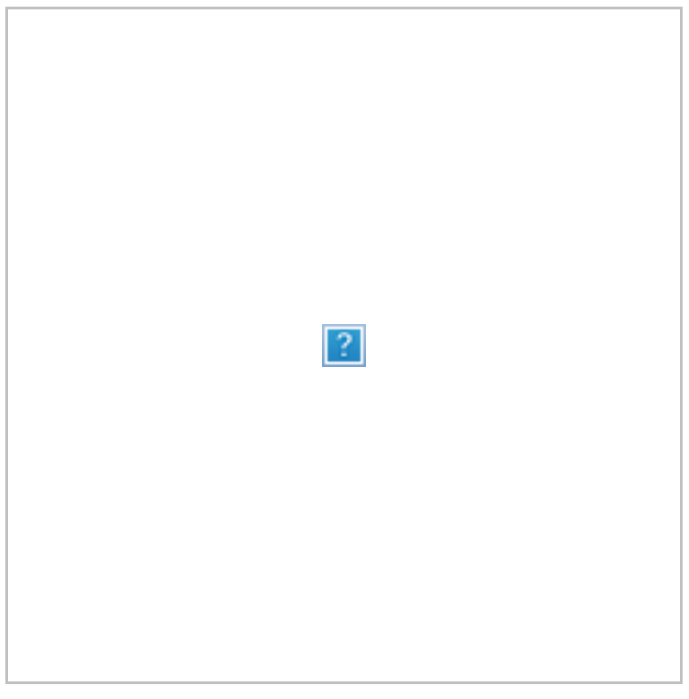
这个由CSS属性中的z-index指定，表示盒模型的第三个大小，即在z轴上的位置。Box分发到堆栈中（称为堆栈上下文），每个堆栈中靠后的元素将被较早绘制，栈顶靠前的元素离用户最近，当发生交叠时，将隐藏靠后的元素。堆栈根据z-index属性排序，拥有z-index属性的box形成了一个局部堆栈，viewport有外部堆栈，例如：

```
<STYLE type="text/css">
    div {
     position: absolute;
     left: 2in;
     top: 2in;
    }
  </STYLE>


<P>
      <DIV
          style="z-index:3; width: 1in; height: 1in; ">
      </DIV>
      <DIV
          style="z-index:1;width: 2in; height: 2in;">
      </DIV>
```

</p>

结果是：



虽然绿色div排在红色div后面，可能在正常流中也已经被绘制在后面，但z-index有更高优先级，所以在根box的堆栈中更靠前。

清风软件测试
关注 - 4
粉丝 - 386
+加关注

« 上一篇： 前端性能优化归纳总结篇！！！
» 下一篇： Web前端性能优化——如何提高页面加载速度

分类 软件测试，性能测试

标签 web前端性能优化，性能优化，前端性能优化

· CPU市场已然天翻地覆！AMD稳稳占据78%
· 宇宙的一切，都源自这个“扭曲”的过程
· 苹果宣布拿出25亿美元应对美国加州住房危机
» 更多新闻...

历史上的今天：
2018-05-02 前端性能优化归纳总结篇！！！
2018-05-02 httpclient接口测试完整用例以及获取信息的方法
2018-05-02 移动 H5（PC Web）前端性能优化指南