本文主要通过以下几个方面说明：

1. **域名解析，**
2. **发起请求**
3. **HTML解析**
4. **CSS解析**
5. **布局**
6. **绘制**

域名解析

在计算机网络中，我们只能通过IP地址访问到具体的主机。我们不能通过域名直接访问。我们的前端的静态资源等，都是存储在服务器上。当输入一个域名的时候，我们首先要做的就是将域名转化成IP地址。在转换的过程中，有以下几个步骤：

1. 首先浏览器会查询自身的缓存中，有没有此条域名的解析，如果有的话，就返回这个解析后的地址。
2. 如果浏览器自身的缓存中，没有找到与此条域名对应的IP地址，那么就会去操作系统中的缓存中查找是否有这条域名的解析。
3. 如果在操作系统中也没有找到的话，那么就需要通过DNS(域名系统)帮助我们解析。
4. 如果浏览器在自身缓存中，以及操作系统的缓存中，并未命中该域名的匹配的话，那么就会查找TCP/IP参数中设置的首选DNS服务器，我们把他叫做本地DNS，如果本地DNS服务器的缓存中命中了该域名，就返回该域名的解析。如果解析不到，那么会根据本地DNS服务器的设置，看是否设置了转发模式，如果设置了转发模式，那么他就会一级一级的去查找，直到找到。如果还没有找到的话，并且这个时候，DNS服务器已经没有启用转发模式，那么就会向根DNS服务器发起查询请求。
5. 当向DNS根服务器发起请求的时候，根服务器会返回当前他所已知的顶层域名服务器，然后，接着向这些顶层域名服务器去发起请求，如果某个顶层域名服务器解析，是属于他所管辖的范畴，那么就会返回他所管辖的二级DNS域名服务器，以此类推，直到找到或者找不到。

我们上面说到了DNS，那么什么是DNS?

DNS的全称是domain name system，也就是域名系统。他工作在应用层，主要作用是帮我们完成域名到IP的转化。他的体系结构是`分布式集群`，顶层为`根服务器`，接下来是`顶层域名服务器`，接下来是`次级域名服务器`。结构大致如下图所示：

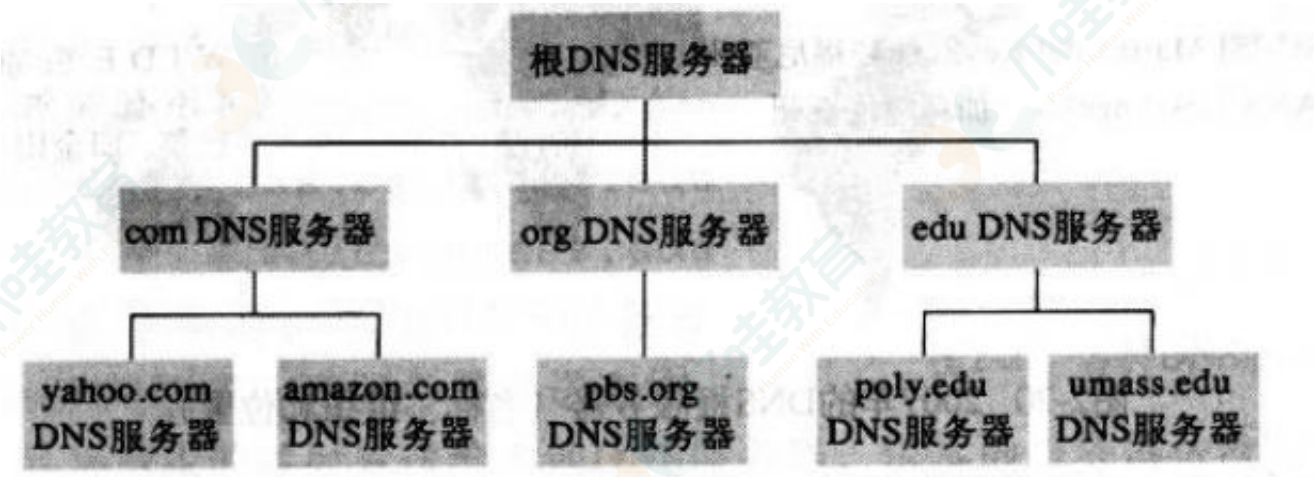


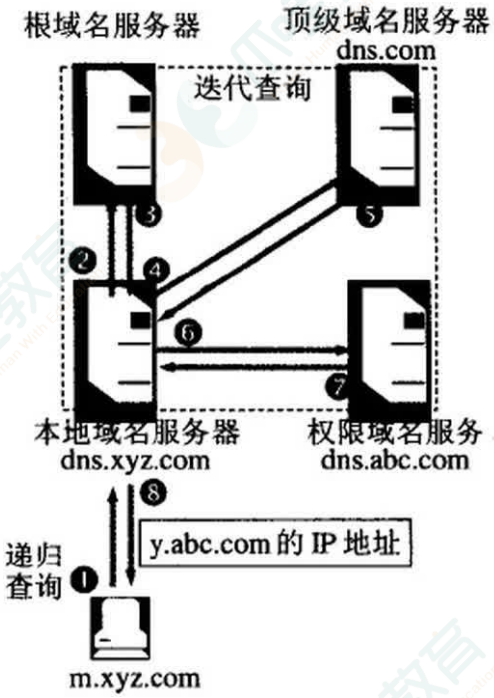
图2-19 DNS服务器的部分层次结构

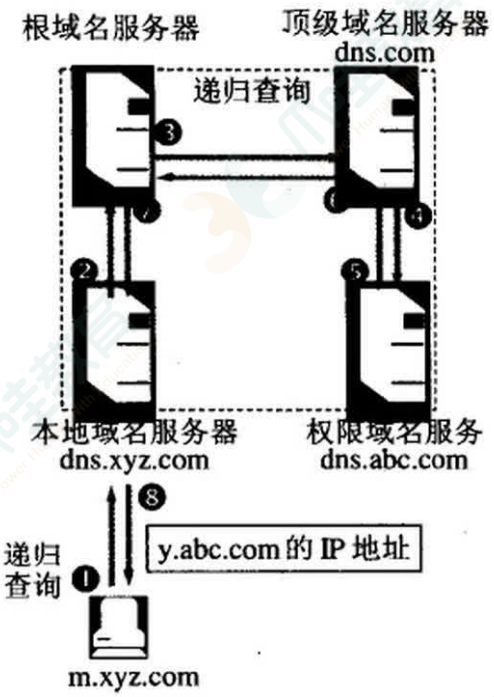
之所以DNS要设计成分布式集群，而不是单站点模式，是因为：如果一个域名服务器挂了，不会导致整个互联网崩掉。查询的时候，也比较快速，如果只有一个服务器，假如这个服务器放在美国，那么距离美国最远的地方，首先DNS客户端向DNS服务器发送的过程，都要走很长一段时间的网络，在加上，如果所有的域名信息否放在同一个服务器上，那么查询速度以及存储都是一个很大的问题。因此就需要设计成分布式集群。

DNS在查询域名解析的过程中，分为：`迭代查询`以及`递归查询`。

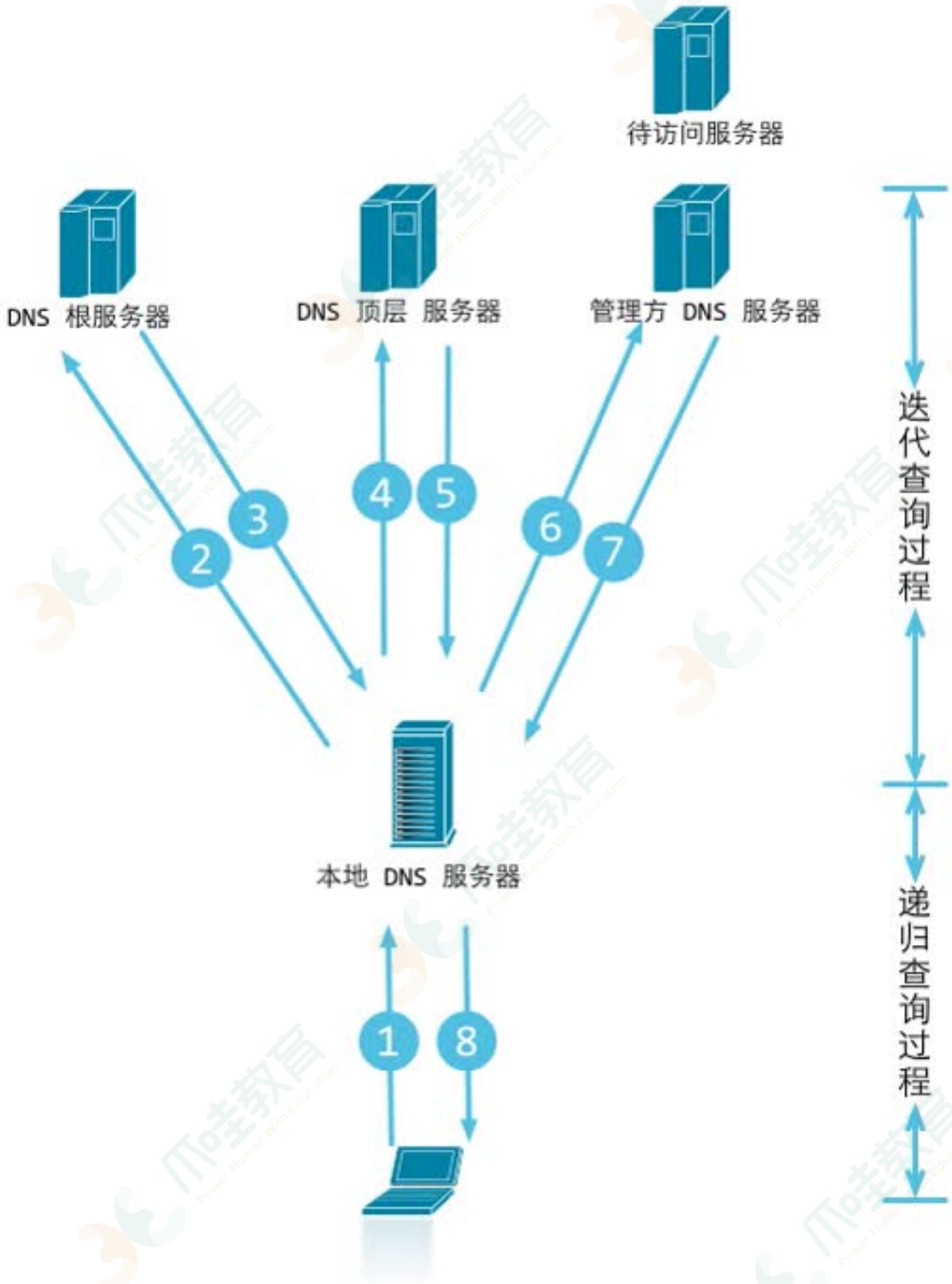
递归查询：当计算机DNS客户端向本地DNS服务器发起查找的过程中，如果没有查找到的话，那么此时，本地DNS服务器充当DNS客户端，向上级DNS服务器，根域名服务器或顶级域名服务器等发起解析。以此类推，到之后我只需要返回给上一级查询的正确域名或者没有查找到。

迭代查询：当计算机DNS客户端向本地DNS服务器发起查找，如果没有查找到，此时本地DNS服务器会返回给计算机DNS客户端我本地DNS服务器的上一级，根域名服务器或顶级域名服务器等，然后计算机DNS在向这些域名服务器发起查询，直到查找到或者未找到，如图所示





那么一般在计算机中，计算机DNS客户端向本地DNS查询的过程为递归查询，本地DNS向上级护着根DNS域名服务器发起查找的过程称为迭代查询。



发起请求：

当域名解析完毕之后，就会发起请求。我们在这里**假定这个域名从来没有被访问过**。那么它会经过以下几个阶段：

如果是第一次请求，那么在请求后，收到的响应中，会有一些关于强弱缓存的字段，比如：

强缓存字段：

`Expires`：Expires 的值是一个 HTTP 日期，在浏览器发起请求的时候，会根据系统时间和 Expires 的值进行比较，如果系统时间超过了 Expires 的值，缓存失效。这个字段会导致一个问题，要是系统时间与服务器时间不一致的时候，就可能出现假性失效，或者出现缓存已经失效了，但是并未去请求最新资源

`Cache-control`： HTTP/1.1 中新增的属性，属性值具有以下几个：

1. max-age：单位是秒，计算方式是距离**第一次**响应头中有该字段的时间。如果时间超过，就要重新发起请求。
2. no-cache：不使用强缓存，每次都需要与服务器校验文件的新鲜度。
3. no-store：不使用任何缓存，每次都要去服务器请求最新的资源。
4. private：专用于个人的缓存。中间代理、cdn不能缓存此响应。
5. public：响应可以被cdn，中间代理所缓存。

`pragma`：不使用强缓存，需要验证缓存是否新鲜。

强缓存的优先级：pragma > cache-control > expires。

弱缓存字段：

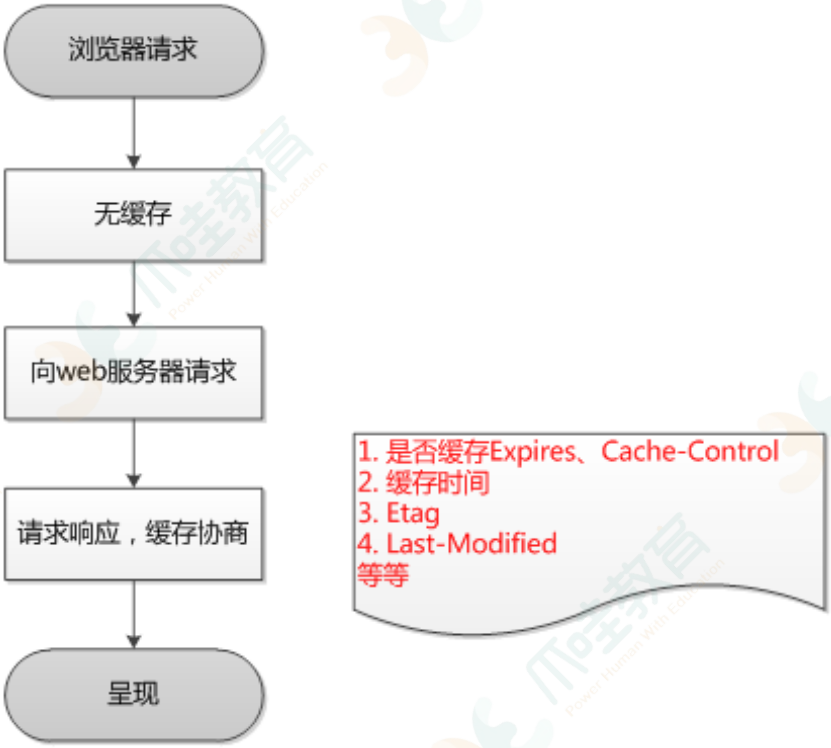
在服务器**第一次响应**请求的时候，有可能有强缓存字段，也有可能没有。也有可能有弱缓存的字段：

1. `last-modified`：最后一次文件修改的时间。单位是秒。浏览器在下一次请求的时候，会在请求头中加上`if-moditied-since`字段，接下来服务器就会去对比时间是否一样，如果一样告知继续使用缓存，如果不一致，返回最新的资源，并且在响应头部更新last-modified字段。这个会存在一个问题：**因为是以秒为单位，假设在1s内服务器先响应了请求，返回了一个时间，接着我在一秒内更新服务器上的资源的，那么这个时候，按照逻辑的话，浏览器下次请求的时候，服务器会 让它继续使用缓存（因为时间一样）。这样就导致了文件更新失败。**
2. `Etag`：为了防止出现last-modified/if-modified-since出现的文件更新失败的情况而引入的。Etag是一个hash串，代表的是一个资源的标识符，当服务端的文件变化的时候，它的 hash码会随之改变。浏览器在下次请求的时候会带上`if-not-match`字段，服务器判断是否与现在的资源的hash串一致，如果一致就让浏览器继续适应缓存，否则就发送最新的资源给浏览器，并在响应头中更新Etag字段。`ETag 又有强弱校验之分，如果 hash 串是以 "W/" 开头的一串字符串，说明此时协商缓存的校验是弱校验的，只有服务器上的文件差异（根据 ETag 计算方式来决定）达到能够触发 hash 值后缀变化的时候，才会真正地请求资源`。

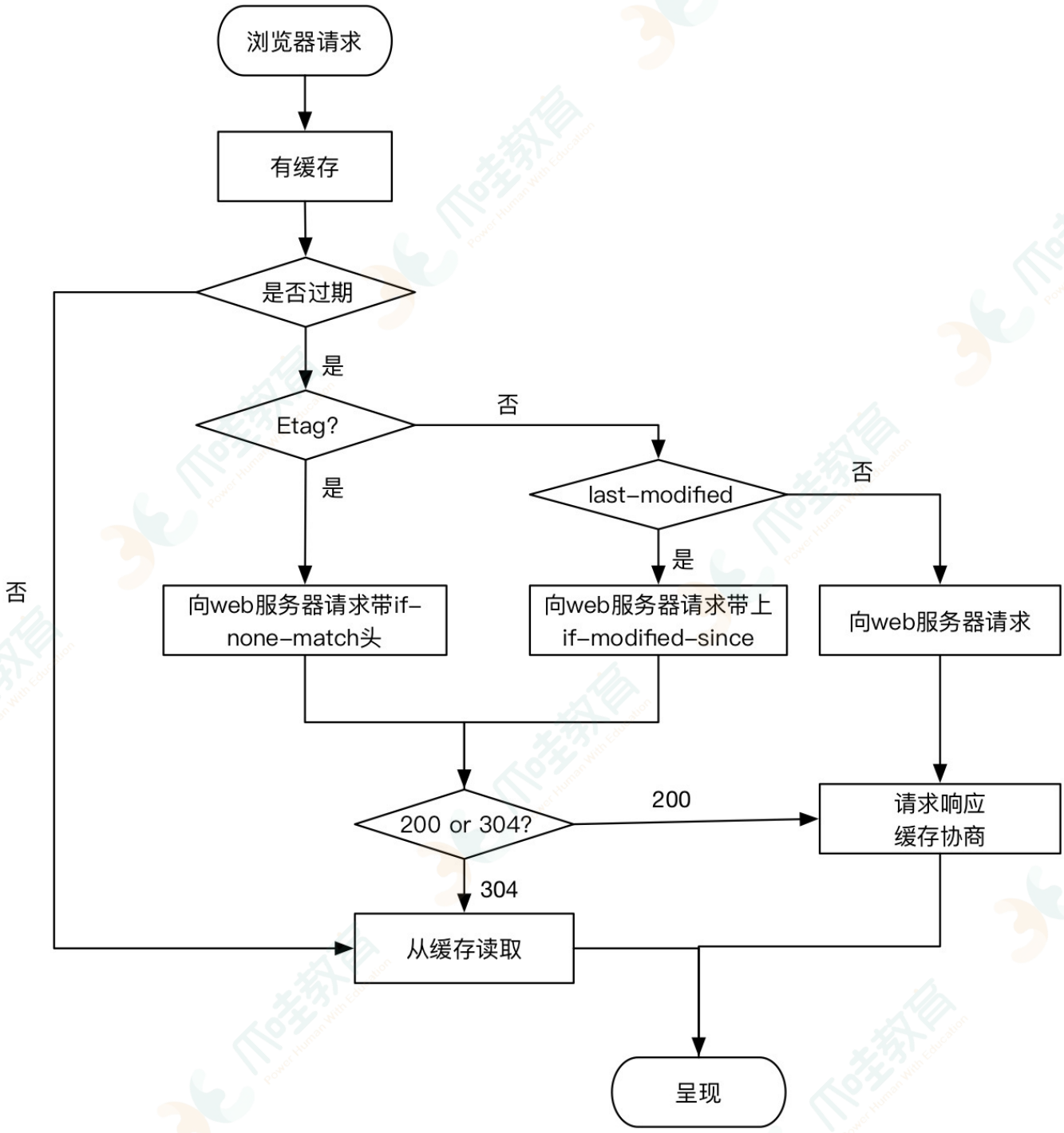
如果对hash比较感兴趣的同学，可以去什么是hash了解下。

用两张张图来总结就是：

浏览器第一次发起请求：


浏览器请求
无缓存
向web服务器请求
请求响应，缓存协商
呈现
1. 是否缓存Expires、Cache-Control
2. 缓存时间
3. Etag
4. Last-Modified
等等

浏览器发起第二次请求：



解析HTML文档

在请求到资源之后，浏览器需要解析HTML，生成dom树，cssRule树。结合之后形成render树，之后再渲染。

浏览器在解析HTML的时候，主要所做的事情是两个：词法分析和语法分析。

词法分析：

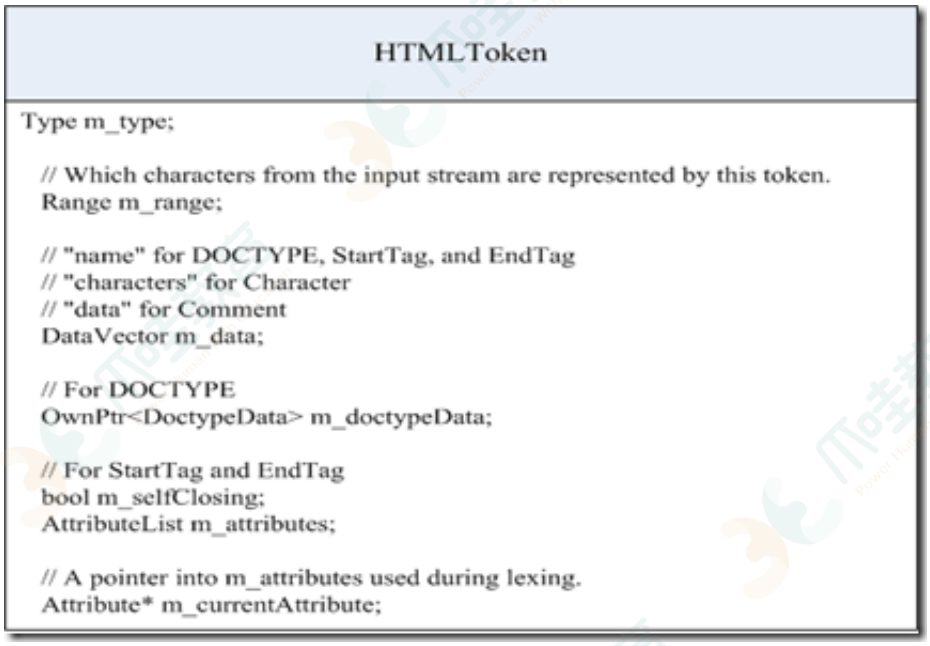
所谓的词法分析就是将**一大段字符串转根据规则解析成一个个最小有意义的单元，之后再根据这个最小意义单元的相应数据生成一个token**。

词法分析阶段采用的算法是：**标记化算法**（将html从左到右依次的读入字符，内部使用状态机来断言当前的状态，根据语法规则匹配出可以分解的htmlToken，最后将这个htmlToken提供给语法分析阶段）

HTML中最小有意义单元的种类有：标签开始，标签结束，注释，文本，属性，CDATA 节点。

类型	描述
<xx	"开始标签"的开始
/>	"开始标签"的结束
name='byeL'	属性
	结束标签
我是文本	文本节点
	注释
	CDATA 节点
attr="xxxx"	属性

在讲具体的词法分析之前，我们需要先了解下，我们分词之后需要生成的数据结构长什么样。生成的具体的数据结构如下图所示：



具体解释为：

```
Type m_type;    //类型 取值有DOCTYPE, StartTag, EndTag, Character, Comment

   Range m_range; // Always starts at zero.   //在字节流中的偏移

   int m_baseOffset;

   // "name" for DOCTYPE, StartTag, and EndTag
```

```
    // "characters" for Character

    // "data" for Comment

    DataVector m_data;  //数据

    // For DOCTYPE

    OwnPtr<DoctypeData> m_doctypeData; //文档类型

    // For StartTag and EndTag

    bool m_selfClosing;  //是否自封闭

    AttributeList m_attributes;   //属性列表

    // A pointer into m_attributes used during lexing.

Attribute* m_currentAttribute;   //当前属性
```

我们以下面为例详细说明：

```
/*
  <a href="w3c.org">w3c</a>
  1.初始状态为DataState
  2.读取到,进入"<",进入TagOpenState状态,
  3.读取到"a"，进入到TagNameState状态，并且初始化一个HtmlToken,类型为StartTag,
  4.读取到空格，进入BeforeAttributeNameState状态，将之前的TagNameState阶段存储的名称存储起到
HtmlToken的name中。
  5.碰到"h"，进入到AttributeNameState状态,
  6.继续读取"r", "e", "f" 直到读取到"=", 进入到BeforeAttributeValue状态
  7.继续读取，碰到""",进入AttributeValueDoubleQuotedState状态。
  8.继续读取，碰到："w", "3"......"r", "g",保持状态，提取属性值。
  9.读取到""",  进入AfterAttributeValueState状态。
  10.碰到">",进入到DataState阶段。
  以此类推：
  最终会生成三个htmlToken，分别是：
*/
{
    m_type: 'StartTag',
    m_attributes: [{
     href: 'w3c.org',
   }]
    m_data:'a',
    m_selfClosing: false,
},
```

```
{
    m_type: 'character',
    m_data: 'w3c',
    m_attributes: [],
    m_selfClosing: false,

},
  {
    m_type: 'EndTag',
    m_attributes: [{
     href: 'w3c.org',
   }]
    m_data:'a',
    m_selfClosing: false,
},
```

解析的详细过程可以插件WEBKIT中的HTML词法解析。

在词法分析的过程中，语法分析也会同步开始进行。

语法分析：

语法分析的作用是根据词法分析阶段生成的htmlToken，将其转化成一颗树状结构，也就是我们所说的dom树。

在将这些分好的词转化成dom树的时候，我们需要用到一种数据结构：栈。

在开始之前，先向栈顶压入根元素，等到解析完成之后，这个根元素就是最后的dom树。

当解析完生成一个词的时候，就会将他入栈，有以下几种操作的可能：

1. 如果是一个开始节点的话，那么直接入栈。不做任何操作
2. 如果前一个是文本节点，并且本次入栈的也是文本节点的话，会将最后入栈的文本节点与前一个文本节点进行合并。先把它添加到当前栈顶元素的子节点数组中，然后入栈。
3. 如果是注释节点，那么直接添加到当前栈顶元素的自己诶单数组中。
4. 如果是属性的话，直接添加到当前栈顶元素的属性中。
5. 遇到一个结束节点，就向前找到第一个与之匹配的开始节点，并且出栈。

举个例子：

```
<div>
  <p>
  1234   45678   789
  </p>
</div>
```

```
// 上面具有最小意义的字符单元有：<div,  >,  <p ,  >, 1234, 45678, 789, </p>, </div>;
// 经过分词之后，会生成如下数据结构：
```

```
  {
    m_type: 'StartTag',
    m_attributes: [{

   }]
  m_data:'div',
  m_selfClosing: false,
}
// 在每生成一个词的时候，就需要将其入栈分析，分析步骤如下：
// 部分逻辑代码如下所示（js模拟实现）：
class HTMLDocument {
  constructor () {
    this.isDocument = true
    this.childNodes = []
  }
}
class Node {}
class Element extends Node {
  constructor (token) {
    super(token)
    for (const key in token) {
      this[key] = token[key]
    }
    this.childNodes = []
  }
  [Symbol.toStringTag] () {
    return `Element<${this.name}>`
  }
}
class Text extends Node {
  constructor (value) {
    super(value)
    this.value = value || ''
  }
}
function HTMLSyntaticalParser () {
  const stack = [new HTMLDocument] // 先实例化一个栈，并且往栈里压入一个根元素，当执行完毕的时候，栈顶就是一个完整的dom树

  this.receiveInput = function (token) {
    if (typeof token === 'string') { // 如果是文本类型的话，
      if (getTop(stack) instanceof Text) { // 并且栈顶也是文本节点的话
        getTop(stack).value += token // 合并
      } else { // 否则将其加入到栈顶的孩子节点
        let t = new Text(token)
        getTop(stack).childNodes.push(t)
        stack.push(t)
      }
```

```
    } else if (getTop(stack) instanceof Text) { // 如果不是文本节点，并且前一个是文本节点的话，需要将其出栈
      stack.pop()
    }

    if (token instanceof StartTagToken) { // 如果是开始节点，先将其假如到栈顶元素的孩子节点中，在将其插入栈顶
      let e = new Element(token)
      getTop(stack).childNodes.push(e)
      return stack.push(e)
    }
    if (token instanceof EndTagToken) { // 如果是结束节点的话，那么不入栈，并且出栈一个元素，这个元素肯定是与他匹配的（文档结构正确的前提下）
      return stack.pop()
    }
  }

  this.getOutput = () => stack[0]
}

function getTop (stack) {
  return stack[stack.length - 1]
}
```

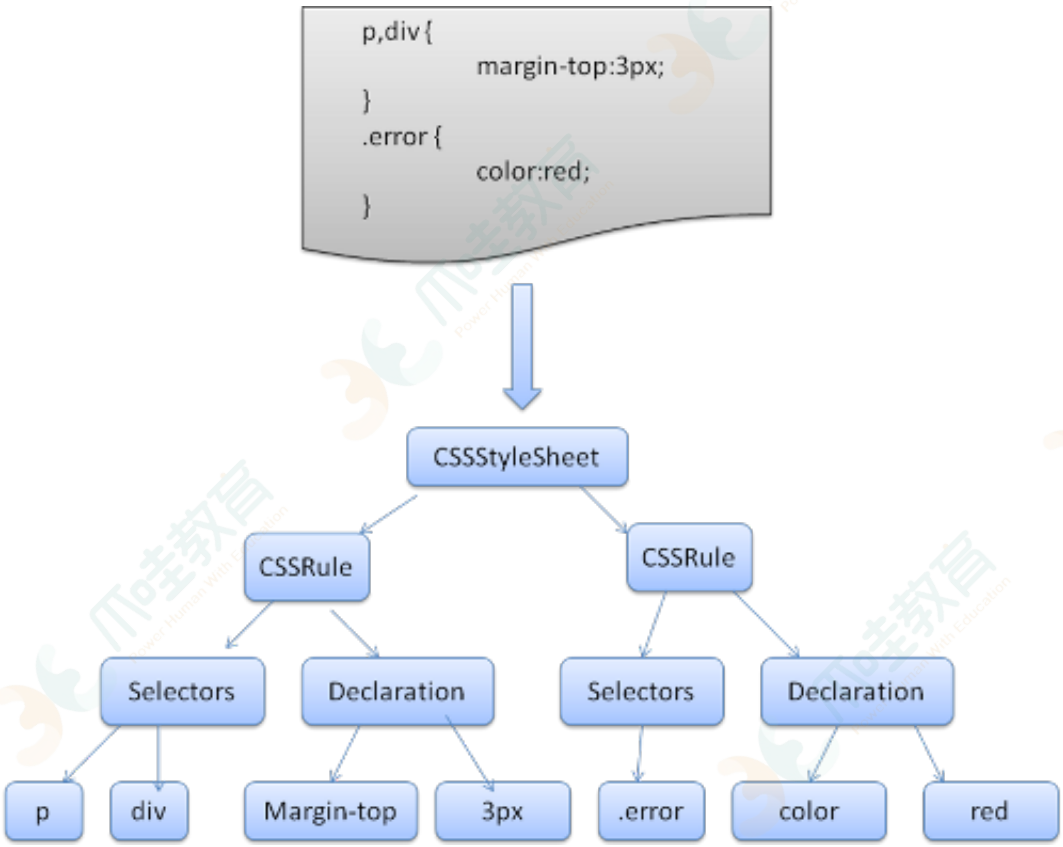
经过上面的几个步骤，那就就会将html文档转化为一个dom树。

需要注意的是：

如果存在JS脚本，那么在JS脚本解析的过程中，会将渲染线程挂起。原因详见：https://juejin.cn/post/6961379369436741663/

css解析

在生成dom树的时候，也会解析css，这两个是并行执行的，一旦存在css样式（包括但不限于行内样式，外部样式引入等），就会根据语法规范进行解析和标记。解析完成后，会生成一个stylesheet对象，这个对象里面包含着解析好的css规则，css规则是由选择器和声明对象组成。



例如：

```
.btn-style {
  font-size: 12px;
  background-color: yellow;
}
```

以上的css被解析之后，会生成：

选择器名称	属性	值
.btn-style	font-size	12px
.btn-style	background-color	yellow

render树的生成

等到css的rule树与dom树都解析完毕之后，那么就会根据这两个树生成最终的render树。

render树的生成，就是遍历当前生成的dom树，根据当前的dom树的子节点信息以及对应的css规则，最终生成一个或多个render子节点。

在webkit中，所有的reder子节点都继承与RenderObject，在RenderObject中，有重绘与重排的具体方法声明虚方法。以及dom节点，style样式信息等。

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

一个render子节点中，如果style样式中，有具体的大小设置，例如：`width:12px;`，那么在布局的时候就会直接使用这个具体的宽度。当没有定义节点宽度或者定义的宽度为百分比的时候，例如：`width:50%`；那么就需要在布局的时候对其大小进行计算。

需要注意的是：

render树的节点并不等同的dom树的节点，因为有些节点的display为none，那么在生成render树的时候，就不会将其加入到render树种。还有例如“select”元素有 3 个render树子节点：一个用于显示区域，一个用于下拉列表框，还有一个用于按钮。

布局阶段：

遍历render树，根据render节点的类型，确定元素的大小以及位置。

绘制阶段

在绘制阶段，系统会遍历render树，并调render树的子节点的“paint”方法，将render树的子节点的内容显示在屏幕上。绘制工作是使用用户界面基础组件完成的。

CSS2 规范定义了绘制流程的顺序。绘制的顺序其实就是元素进入堆栈样式上下文的顺序。这些堆栈会从后往前绘制，因此这样的顺序会影响绘制。块呈现器的堆栈顺序如下：

1. 背景颜色
2. 背景图片
3. 边框
4. 子代
5. 轮廓