Лекция 1

HTTP. Основы работы браузера. Форматы данных

Что вообще происходит когда вы вбиваете в адресную строку браузера какие-то буквы, а он вам показывает страницу?

URL/DNS

Предположим, что вы ввели адрес (URL, Universal Resource Locator)

```
http://twitter.com
protocol://server_adress:port/path_to_resource/resource_name
```

Разберемся, что есть что. Первая часть адреса — это протокол, в нашем случае — HTTP (Hyper Text Transfer Protocol), который является основным протоколом интернета. Кроме HTTP, браузер может работать с HTTPS, FTP, IRC, почтовыми протоколами.

Дальше идет адрес сервера, но в нашем примере это домен. Домен это такие буквы, которые удобно запоминать людям, от людей вообще много всяких проблем.. в общем нам из букв нужно получить цифры, тут на помощь приходит штука называющаяся DNS (Domain Name System). Грубо говоря эта такая распределенная база данных в которой содержатся сведения о доменных именах в виде ресурсных записей (единица передачи информации в DNS).

Примерный сценарий поиска адреса сервера сайта выглядит так: браузер спрашивает у DNS-клиента операционной системы «куда идти, если хочу twitter почитать?», если вы у себя локально что-нить для этого предприняли, то все на вашей совести, если нет то такой же вопрос задается ближайшему DNS-серверу, если _внезапно_ он ничего вообще

не знает по этому поводу, то обращается к корневому серверу — например, 198.41.0.4. Этот сервер сообщает — «я не знаю, но 204.74.112.1 является ответственным за зону com». Тогда сервер DNS направляет свой запрос к 204.74.112.1, но тот отвечает «я знаю, но 207.142.131.234 знает все про зону twitter.com.» Наконец, тот же запрос отправляется к 207.142.131.234 и получает ответ — IP-адрес, который и передаётся клиенту — браузеру.

Вот, собственно и все, адрес получили, пора установить соединение, нужен порт, он может быть всяким, но если конкретное значение упущено, но для HTTP это 80.

Соединение установили, теперь просим у сервера нужный ресурс, он отвечает.. либо не отвечает (но тут поделать уже нечего). Мы будем оптимистичны, т.е. считаем что ответ получили, в нем может содержаться то, что мы просили, а может и нет. В первом случае там кроме контента ожидаемого еще и мета данные, рассказывающие о том, как вообще все прошло и что делать дальше с полученным ответом; во втором случае там всякая полезная* информация о том, почему то, чего мы хотели не произошло.

HTTP

В начале мы не спроста обратили внимание на то, что первым делом в адресе сайта указан протокол, это типа как язык межсетевого общения, в нашем случае браузера и сервера. Нас он интересует по той причине, что неплохо было бы быть готовым к разным исходам всего, что может произойти и если уж совсем край сообщать об этом пользователю так, чтобы он не нервничал сильно.

Структура

Итак, ходят между клиентом и сервером сообщения парами запрос-ответ. Каждое сообщение имеет следующую структуру:

1. Стартовая строка;
2. Заголовки — характеризуют тело сообщения, параметры передачи и прочие сведения;
3. Тело сообщения — непосредственно данные сообщения.

Стартовая строка определяет тип сообщения. Стартовые строки запроса и ответа отличаются.

Для запроса она выглядит следующим образом

```
METHOD URI HTTP/version
```

Для ответа

```
HTTP/version StatusCode ReasonPhrase
```

METHOD	название метода запроса
URI	определяет путь к запрашиваемому ресурсу

Version	2 разделённых точкой цифр. *Например 1.1*
StatusCode	3 цифры. По статусу определяется дальнейшее содержимое
ReasonPhrase	текстовое короткое пояснение к коду ответа для пользователя

Методы

Вообще, согласно стандарту, у каждого метода есть четкое назначение, но в реальной жизни чаще всего используется GET, POST для пересылки бинарных данных, остальное как глаголы для обращения к REST сервисам. Но, дабы иметь какое-то представление давайте все-таки опишем для чего задумывались хотя бы основные методы HTTP запросов.

GET

Используется для запроса содержимого указанного ресурса. С помощью метода GET можно также начать какой-либо процесс.

POST

Применяется для передачи пользовательских данных заданному ресурсу. С помощью метода POST обычно загружаются файлы на сервер. Основное отсличие от GET в том, что все данные, которые пользователь хочет отправить будут находиться в теле запроса.

PUT

Применяется для загрузки содержимого запроса на указанный в запросе URI. Если по заданному URI не существовало ресурса, то сервер создаёт его и возвращает статус 201 (Created). Если же был изменён ресурс, то сервер возвращает 200 (Ok) или 204 (No Content). Сервер не должен игнорировать некорректные заголовки Content-*, передаваемые клиентом вместе с сообщением. Если какой-то из этих заголовков не может быть распознан или не допустим при текущих условиях, то необходимо вернуть код ошибки 501 (Not Implemented).
Фундаментальное различие методов POST и PUT заключается в понимании предназначений URI ресурсов. Метод POST предполагает, что по указанному URI будет производиться обработка передаваемого клиентом содержимого. Используя PUT, клиент предполагает, что загружаемое содержимое соответствует находящемуся по данному URI ресурсу.

На заметку: важной ремаркой будет то, что сообщения ответов сервера на запросы методами POST/PUT не кэшируются.

DELETE

Удаляет указанный ресурс.

Еще есть такие методы как OPTIONS/UPDATE/PATCH/TRACE но про них, если есть желание, можно прочитать тут.

Статусные коды

Клиент узнаёт по коду ответа о результатах его запроса и определяет, что ему делать дальше. Набор кодов состояния является стандартом, выделены 5 классов кодов состояния.

1xx Informational

В этот класс выделены коды, информирующие о процессе передачи. В HTTP/1.0 сообщения с такими кодами должны игнорироваться. В HTTP/1.1 клиент должен быть готов принять этот класс сообщений как обычный ответ, но ничего отправлять серверу не нужно. Сами сообщения от сервера содержат только стартовую строку ответа и, если требуется, несколько специфичных для ответа полей заголовка. Прокси-серверы подобные сообщения должны отправлять дальше от сервера к клиенту.

2xx Success

Сообщения данного класса информируют о случаях успешного принятия и обработки запроса клиента. В зависимости от статуса сервер может ещё передать заголовки и тело сообщения.

3xx Redirection

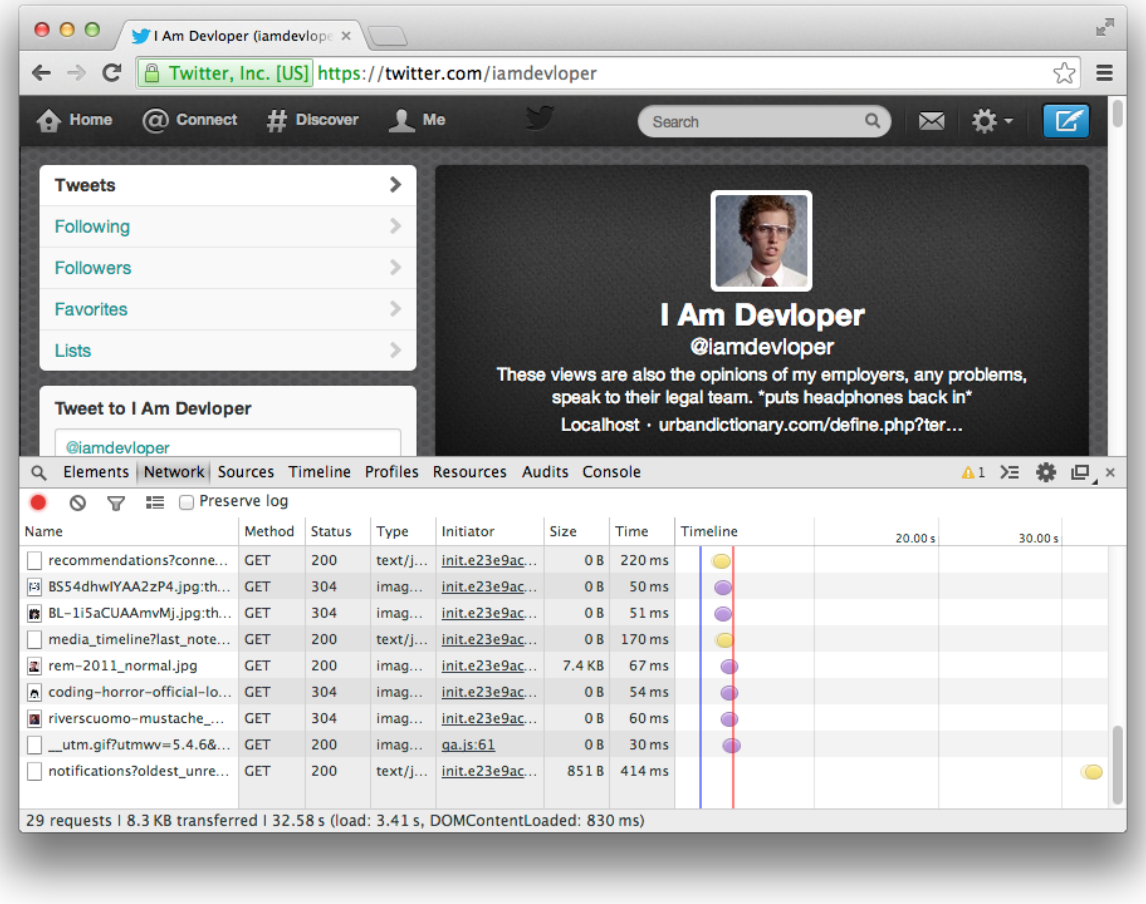
Коды класса 3xx сообщают клиенту что для успешного выполнения операции необходимо сделать другой запрос (как правило по другому URI). Из данного класса пять кодов 301, 302, 303, 305 и 307 относятся непосредственно к перенаправлениям (редирект). Адрес, по которому клиенту следует произвести запрос, сервер указывает в заголовке Location. При этом допускается использование фрагментов в целевом URI.

4xx Client Error

Класс кодов 4xx предназначен для указания ошибок со стороны клиента. При использовании всех методов, кроме HEAD, сервер должен вернуть в теле сообщения гипертекстовое пояснение для пользователя.

5xx Server Error

Коды 5xx выделены под случаи неудачного выполнения операции по вине сервера. Для всех ситуаций, кроме использования метода HEAD, сервер должен включать в тело сообщения объяснение, которое клиент отобразит пользователю.



На картинке можно увидеть 2 типа ответов — 200 и 304, т.е. ресурсы загруженные удачно и ресурсы, загруженные удачно, но из кэша браузера :-)

Заголовки

Заголовки HTTP — это строки в HTTP-сообщении, содержащие разделённую двоеточием пару параметр-значение. Нужны они для того, чтобы клиент и сервер могли рассказать друг-другу какого рода данные они передают, как долго можно хранить эти данные в кэше, в общем всякая мета информация.

Примеры заголовков:

```
Server: Apache/2.2.11 (Win32) PHP/5.3.0
Last-Modified: Sat, 16 Jan 2010 21:16:42 GMT
Content-Type: text/plain; charset=windows-1251
Content-Language: ru
```

Все заголовки разделяются на четыре основных группы:
General Headers — должны включаться в любое сообщение клиента и сервера.

Request Headers — используются только в запросах клиента.
Response Headers — только для ответов от сервера.
Entity Headers — сопровождают каждую сущность сообщения.

Все необходимые для функционирования HTTP заголовки описаны в основных RFC. Если не хватает существующих, то можно вводить свои. Традиционно к именам таких дополнительных заголовков добавляют префикс «X-» для избежания конфликта имён с возможно существующими. Например, как в заголовках X-Powered-By или X-Cache.

На хабре как-то была статья про интересные заголовки.

Тело сообщения

Включается или не включается тело сообщения в сообщение ответа зависит как от метода запроса, так и от кода состояния ответа.

Все ответы на запрос с методом HEAD не должны включать тело сообщения. Никакие ответы с кодами состояния 1xx, 204 (Нет содержимого, No Content), и 304 (Не модифицирован, Not Modified) не должны содержать тела сообщения. Все другие ответы содержат тело сообщения, даже если оно имеет нулевую длину.

Состояние

HTTP не имеет состояния, т.е. между 2мя парами запрос-ответ никакой связи не существует, а знать что происходит/происходило было бы здорово. Для этого у нас есть такие механизмы, как Cookie и Sessions на клиенте и сервере соответственно.

Основы работы браузера

Сегодня на арене по большому счету все 3 игрока: Chrome/Opera/Safari/Я.Браузер, Firefox, IE.

Статистика на январь 2014 по данным w3c:

Chrome	55.7%
Safari	3.9%
Firefox	26.9%
Opera	1.8%
IE	10.2%

Основные функции браузера

Основное предназначение браузера – отображать веб-ресурсы. Для этого на сервер отправляется запрос, а результат выводится в окне браузера. Под ресурсами в основном подразумеваются HTML-документы, однако это также может быть PDF-файл, картинка или иное содержание. Расположение ресурса определяется с помощью URI (унифицированного идентификатора ресурсов).

То, каким образом браузер обрабатывает и отображает HTML-файлы, определено спецификациями HTML и CSS. Они разрабатываются Консорциумом W3C, который внедряет стандарты для Интернета.

Структура верхнего уровня

Пользовательский интерфейс (UI) – включает адресную строку, кнопки "Назад" и "Вперед", меню закладок и т. д. К нему относятся все элементы, кроме окна, в котором отображается запрашиваемая страница.

Механизм браузера (Browser engine) – управляет взаимодействием интерфейса и модуля отображения.

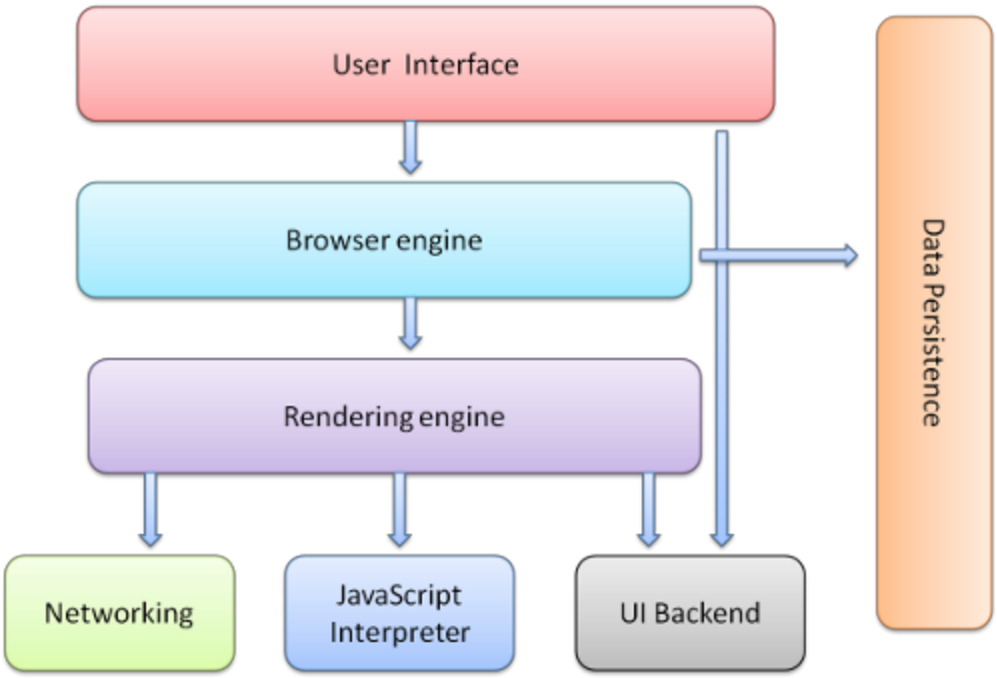
Модуль отображения (Rendering engine) – отвечает за вывод запрошенного содержания на экран. Например, если запрашивается HTML-документ, модуль отображения выполняет синтаксический анализ кода HTML и CSS и выводит результат на экран.

Сетевые компоненты (Networking) – предназначены для выполнения сетевых вызовов, таких как HTTP-запросы. Их интерфейс не зависит от типа платформы, для каждого из которых есть собственные реализации.

Исполнительная часть пользовательского интерфейса (UI backend) – используется для отрисовки основных виджетов, таких как окна и поля со списками. Ее универсальный интерфейс также не зависит от типа платформы. Исполнительная часть всегда применяет методы пользовательского интерфейса конкретной операционной системы.

Интерпретатор JavaScript (JS enterpreter) – используется для синтаксического анализа и выполнения кода JavaScript.

Хранилище данных (Data persistence) – необходимо для сохраняемости процессов. Браузер сохраняет на жесткий диск данные различных типов, например файлы cookie.



Основные компоненты браузера

Модуль отображения

Большинство браузеров, использует несколько экземпляров модуля отображения, по одному в каждой вкладке, которые представляют собой отдельные процессы.

Как можно догадаться по названию, модуль отображения отвечает за вывод содержания документа на экране браузера.

По умолчанию он способен отображать HTML- и XML-документы, а также картинки. Специальные подключаемые модули (расширения для браузеров) делают возможным отображение другого содержания, например PDF-файлов. Однако эта глава посвящена основным функциям: отображению HTML-документов и картинок, отформатированных с помощью стилей CSS.

В интересующих нас браузерах (Firefox, Chrome и Safari) используются 2.5 модуля отображения. В Firefox применяется Gecko – собственная разработка Mozilla, а в Safari и Я.Браузере — WebKit, в Chrome и Opera используется Blink.

Основная схема работы

Модуль отображения получает содержание запрошенного документа через сетевую компоненту браузера (обычно фрагментами по 8 КБ). Схема дальнейшей работы модуля отображения выглядит приведенным ниже образом.

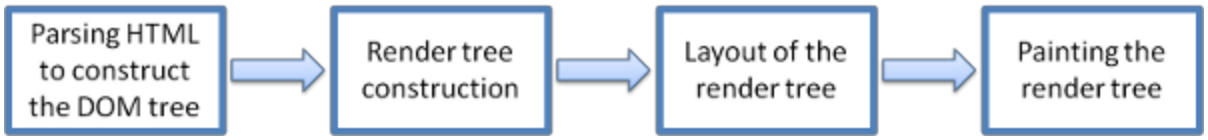


Схема работы модуля отображения

Модуль отображения выполняет синтаксический анализ HTML-документа и построит на его основе DOM, которая представляет из себя дерево содержания (content tree). Информация о стилях извлекается CSS-файлов, элементов style, инлайновых стилей полученные данные будут использованы для построения еще одного дерева – дерева отображения (render tree).

Оно содержит прямоугольники с визуальными атрибутами, такими как цвет и размер. Прямоугольники располагаются в том порядке, в каком они должны быть выведены на экран.

После создания дерева отображения начинается компоновка элементов (layout phase), в ходе которой каждому узлу присваиваются координаты точки на экране, где он должен появиться. Затем выполняется отрисовка (painting phase), при которой узлы дерева отображения последовательно отрисовываются с помощью исполнительной части пользовательского интерфейса (UI backend).

Для удобства пользователя модуль отображения старается вывести содержание на экран как можно скорее, поэтому создание дерева отображения и компоновка могут начаться еще до завершения синтаксического анализа HTML. Одни части документа анализируются и выводятся на экран, в то время как другие только передаются по сети.

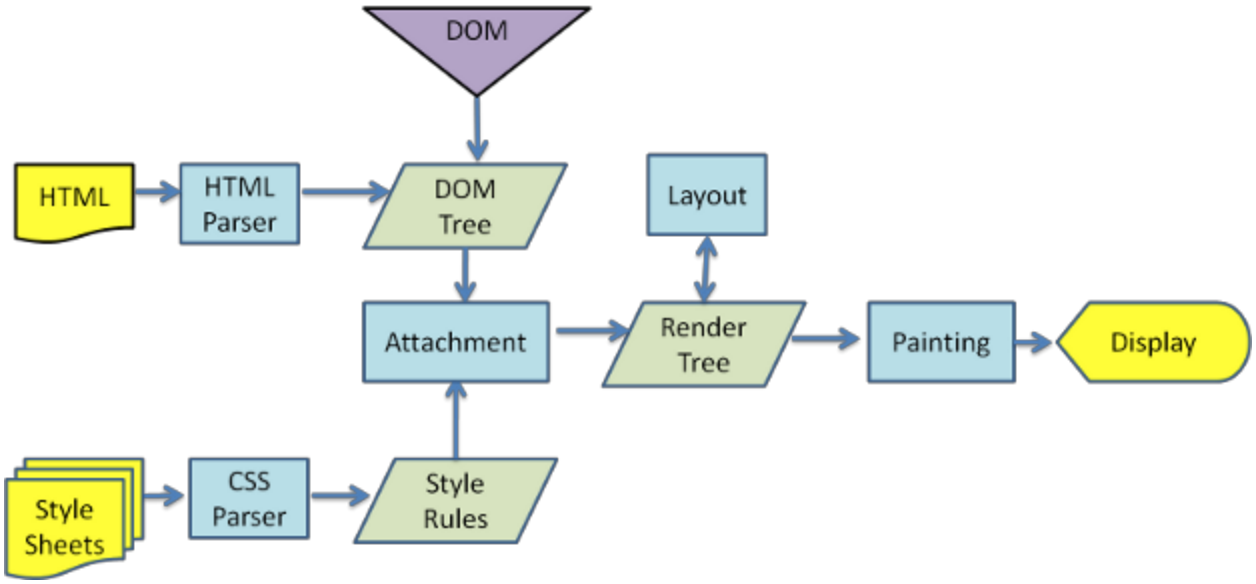


Схема работы модуля отображения

Синтаксический анализатор HTML

Задача синтаксического анализатора HTML – переводить информацию из кода HTML в синтаксическое дерево. Словарь и синтаксис HTML определены в спецификациях W3C. Действующей версией является HTML4, версия HTML5 находится в разработке.

К сожалению, стандартные анализаторы не подходят для языка HTML. HTML невозможно определить с помощью бесконтекстной грамматики, с которой работают синтаксические анализаторы. Это кажется странным, ведь язык HTML не так уж далек от XML, а для XML имеется множество синтаксических анализаторов. Существует даже версия HTML на базе XML (XHTML), так в чем же разница?

Разница в том, что в HTML используется менее строгий подход: если пропущены некоторые теги (например, открывающие или закрывающие), они подставляются автоматически.

Это отличие кажется незначительным только на первый взгляд. С одной стороны, это основная причина популярности HTML: способность языка "прощать" ошибки ощутимо облегчает жизнь разработчику. С другой стороны, из-за этого становится сложно формально определить грамматику. Итак, грамматика HTML не является бесконтекстной, поэтому его анализ нельзя выполнить ни с помощью стандартных анализаторов, ни с помощью анализаторов XML.

DTD в HTML

Определение HTML задается в формате DTD. Этот формат содержит определения всех допустимых элементов, их атрибутов и иерархии.

Существует несколько версий DTD. Строгий формат в точности отвечает спецификации, а остальные также поддерживают разметку, которая использовалась браузерами в прошлом. Это необходимо для обратной совместимости с более старым содержанием. Текущую строгую версию DTD можно загрузить по адресу www.w3.org/TR/html4/strict.dtd.

DOM

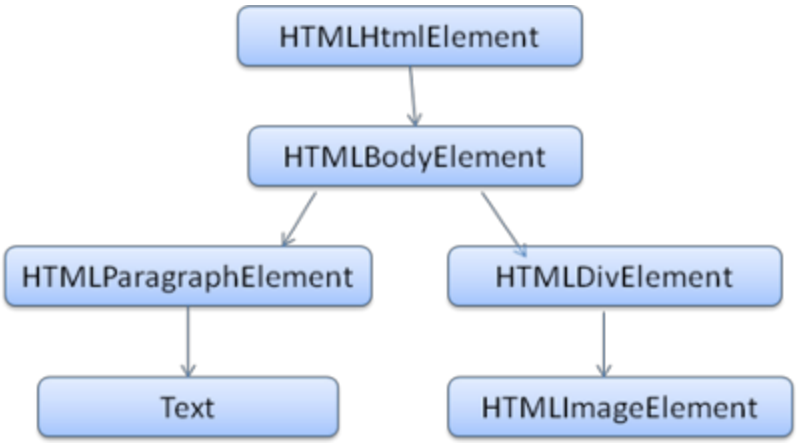
Полученное синтаксическое дерево состоит из элементов DOM и узлов атрибутов. DOM – объектная модель документа (Document Object Model) – служит для представления HTML-документа и интерфейса элементов HTML таким внешним объектам, как код JavaScript. В корне дерева находится объект Document.

Модель DOM практически идентична разметке. Рассмотрим пример разметки:

```
<html>
  <body>
    <p>
      Hello World
    </p>
```

```
    <div> <img src="example.png"/></div>
  </body>
</html>
```

Дерево DOM для этой разметки выглядит так:



Дерево DOM для разметки из примера

Под словами «дерево содержит узлы DOM» подразумевается, что дерево состоит из элементов, которые реализуют один из интерфейсов DOM. В браузерах применяются специфические реализации, обладающие дополнительными атрибутами для внутреннего использования.

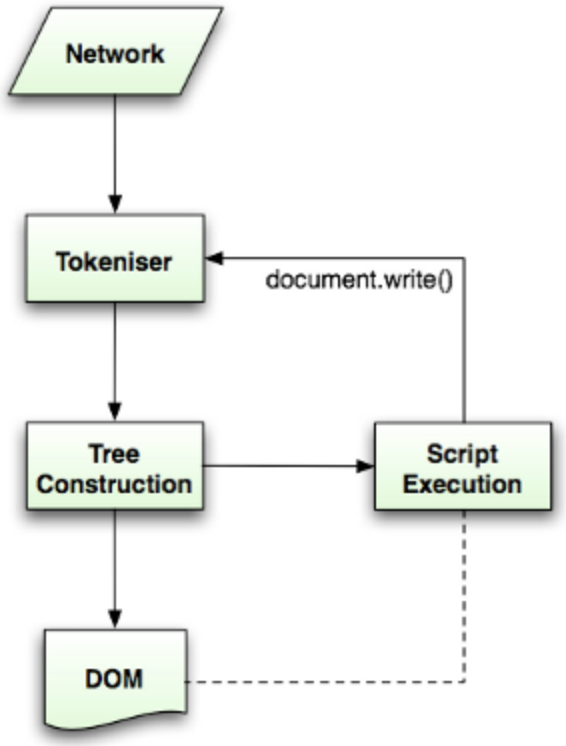
Алгоритм синтаксического анализа

Цикл синтаксического анализа характеризуется возможностью повторного вхождения. Исходный документ обычно не меняется в процессе анализа, однако в случае HTML теги скрипта, содержащие `document.write`, могут добавлять новые токены, поэтому исходный код может меняться.

Алгоритм синтаксического анализа подробно описан в спецификации HTML5. Он состоит из двух этапов: лексического анализа и построения дерева.

В ходе лексического анализа входная последовательность символов разбивается на токены. К токенам HTML относятся открывающие и закрывающие теги, а также названия и значения атрибутов.

Лексический анализатор обнаруживает токен, передает его конструктору деревьев и переходит к следующему символу в поиске дальнейших токенов, и так до окончания входной последовательности.



Этапы синтаксического анализа кода HTML

Алгоритм лексического анализа

Результатом работы алгоритма является токен HTML. Алгоритм выражен в виде автомата с конечным числом состояний. В каждом состоянии обрабатывается один или несколько символов входной последовательности, на основе которых определяется следующее состояние. Оно зависит от этапа лексического анализа и этапа формирования дерева, то есть обработка одного и того же символа может привести к разным результатам (разным состояниям) в зависимости от текущего состояния. Рассмотрим упрощенный пример, который поможет нам лучше понять принцип его работы.

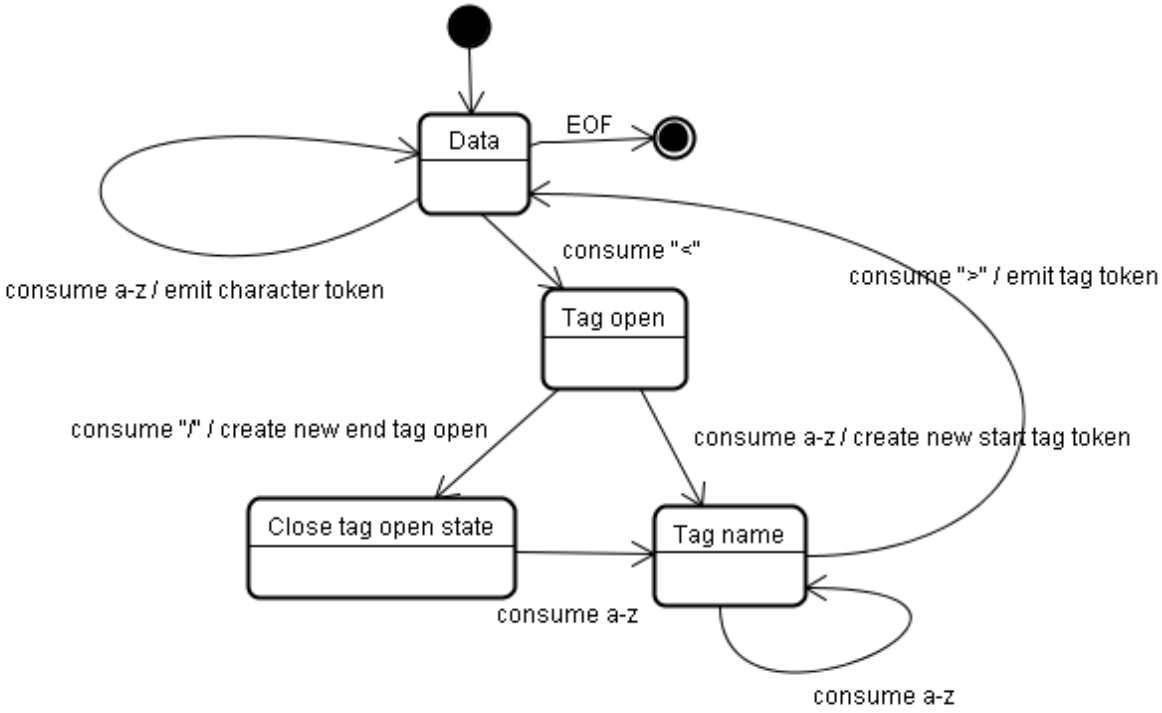
Выполним лексический анализ простого HTML кода:

```
<html>
  <body>
    Hello world
  </body>
</html>
```

Исходное состояние – «данные». Когда анализатор обнаруживает символ `<`, состояние меняется на «открытый тег». Если далее обнаруживается буква (a–z), создается токен открывающего тега, а состояние меняется на «название тега». Оно сохраняется, пока не будет обнаружен символ `>`. Символы по одному добавляются к названию нового токена. В нашем случае получается токен `html`.

При обнаружении символа `>` токен считается готовым и анализатор возвращается в состояние «данные». Тег `<body>` обрабатывается точно так же. Таким образом, анализатор уже сгенерировал теги `html` и `body` и вернулся в состояние «данные». Обнаружение буквы H во фразе Hello world ведет к генерации токена символа. То же происходит с остальными буквами, пока анализатор не дойдет до символа < в теге </body>. Для каждого символа фразы Hello world создается свой токен.

Затем анализатор снова возвращается в состояние "открытый тег". Обнаружение символа / ведет к созданию токена закрывающего тега и переходу в состояние "название тега". Оно сохраняется, пока не будет обнаружен символ `>`. В этот момент генерируется токен нового тега, а анализатор снова возвращается в состояние «данные». Последовательность символов `</html>` обрабатывается, как описано выше.



Лексический анализ входной последовательности символов

Алгоритм построения дерева

При создании синтаксического анализатора формируется объект Document. На этапе построения дерево DOM, в корне которого находится этот объект, изменяется и к нему добавляются новые элементы. Каждый узел, генерируемый лексическим анализатором, обрабатывается конструктором деревьев. Для каждого токена создается свой элемент DOM, определенный спецификацией. Элементы добавляются не только в дерево DOM, но и в стек открытых элементов, который служит для исправления неправильно

вложенных или незакрытых тегов. Алгоритм также выражается в виде автомата с конечным числом состояний, которые называются «способами включения» (insertion mode).

Рассмотрим этапы создания дерева для следующего фрагмента кода:

```
<html>
  <body>
    Hello world
  </body>
</html>
```

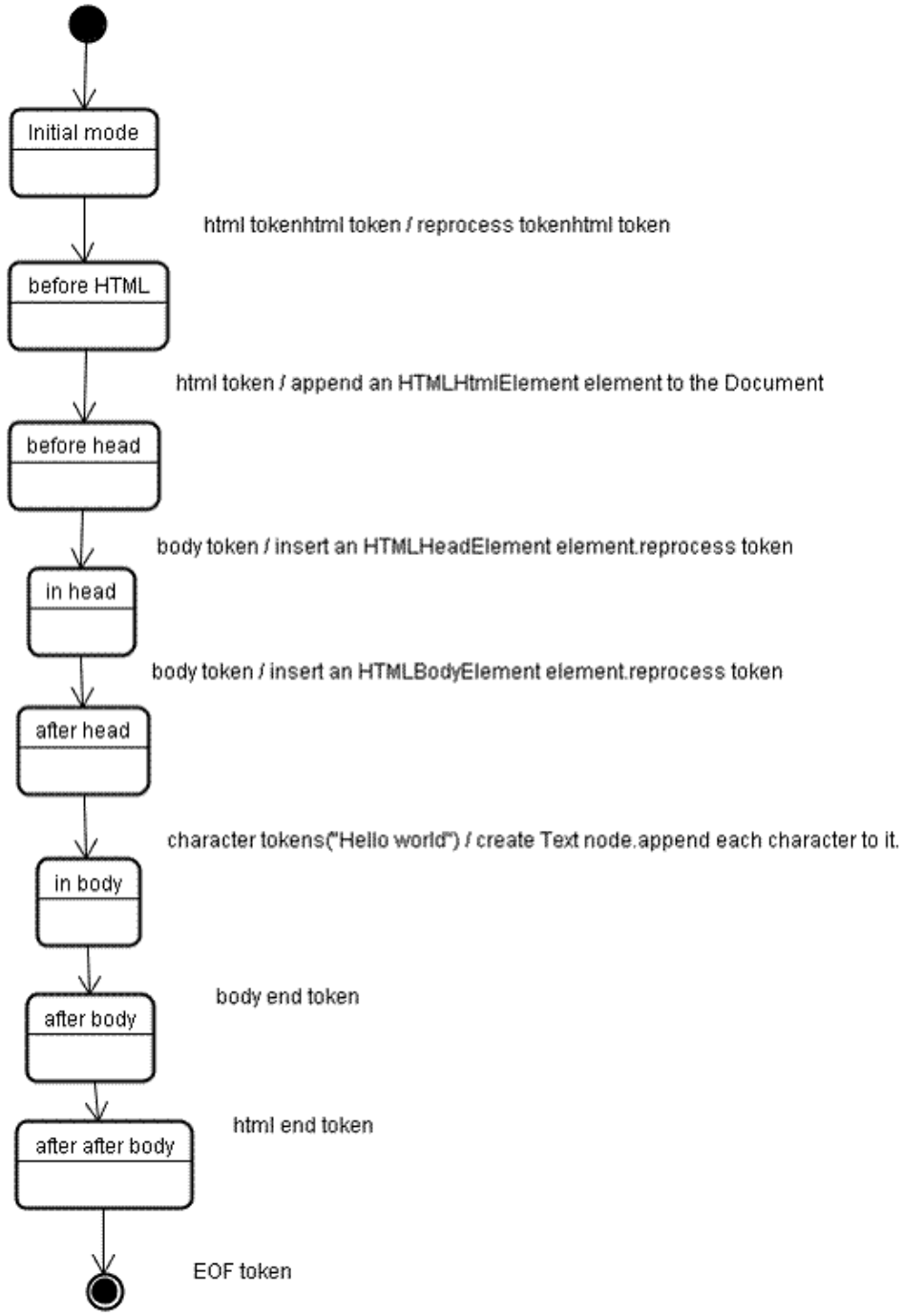
В начале этапа построения дерева у нас есть последовательность токенов, полученная в результате лексического анализа. Первое состояние называется исходным. При получении токена html состояние меняется на «до html», после чего происходит повторная обработка токена в этом состоянии. В результате создается элемент HTMLHtmlElement, который добавляется к корневому объекту Document.

Состояние меняется на «до head». Анализатор обнаруживает токен body. Хотя в нашем коде нет тега head, элемент HTMLHeadElement будет автоматически создан и добавлен в дерево.

Состояние меняется на «внутри head», затем на «после head». Токен body обрабатывается еще раз, создается элемент HTMLBodyElement, который добавляется в дерево, и состояние меняется на «внутри body».

Теперь пришла очередь токенов строки Hello world. Обнаружение первого из них ведет к созданию и вставке узла Text, к которому затем добавляются остальные символы.

При получении закрывающего токена body состояние меняется на «после body». Когда анализатор доходит до закрывающего тега html, состояние меняется на «после после body». При получении токена конца файла анализ завершается.



Построение дерева для кода HTML из примера

Действия после синтаксического анализа

На этом этапе браузер помечает документ как интерактивный и начинает анализ отложенных скриптов, которые необходимо выполнить после завершения анализа документа. Состояние документа затем меняется на «готово», и вызывается событие load.

Обработка ошибок браузерами

На странице HTML вы никогда не увидите ошибку «Недопустимый синтаксис».

Рассмотрим вот такой код HTML:

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

В этом коротком фрагменте я нарушено множество правил (mytag не является стандартным тегом, теги p и div вложены неверно и т. д.), однако браузер не испытывает никаких проблем и отображает содержание корректно. Большая часть кода синтаксического анализатора служит для исправления ошибок разработчиков.

В браузерах используются очень похожие механизмы обработки ошибок, но, как ни странно, они не описаны в текущей спецификации HTML. Как и закладки или кнопки навигации, они просто появились в результате многолетней эволюции браузеров. Существуют недопустимые конструкции HTML, которые довольно часто встречаются на сайтах, и разные браузеры исправляют их похожими способами.

Синтаксический анализатор обрабатывает входящие токены и создает дерево документа. Если документ написан без ошибок, выполняется его стандартный анализ.

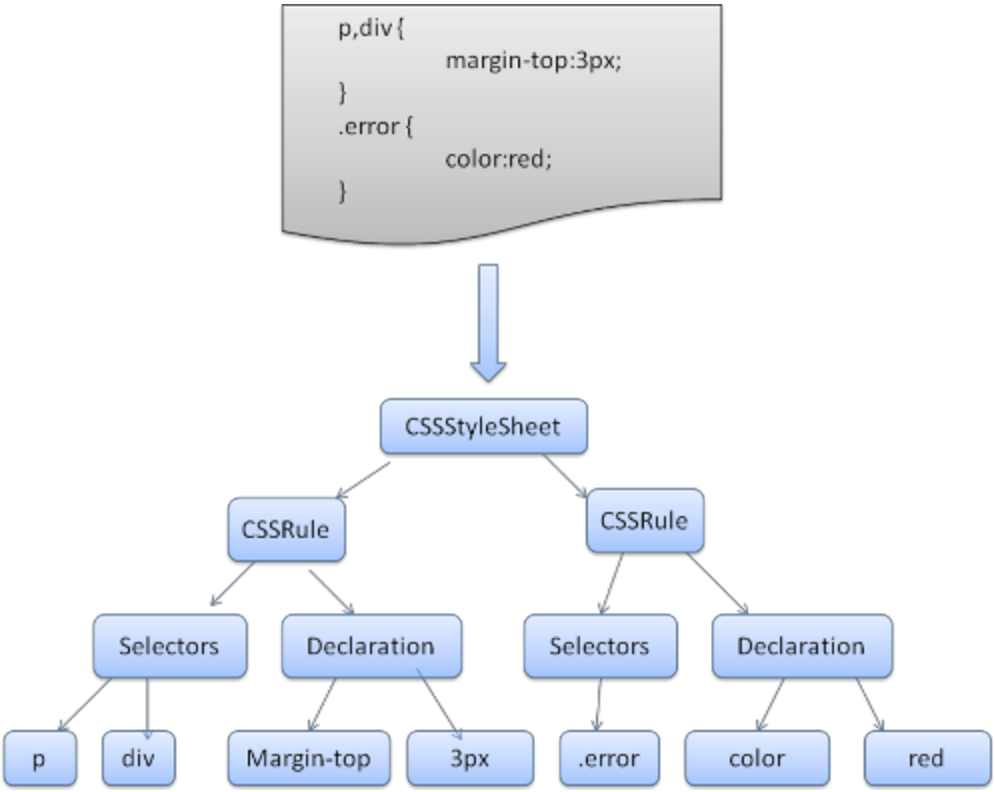
К сожалению, многие документы HTML содержат ошибки, и синтаксический анализатор должен быть к ним готов.

Ниже приведены наиболее «стандартные» ошибки разметки:

- **Использование добавляемого элемента явно запрещено одним из внешних тегов.** В этом случае необходимо закрыть все теги, кроме того, который запрещает использование данного элемента, и добавить этот элемент в самом конце.
- **Элемент нельзя добавить напрямую.** Возможно, автор документа забыл вставить тег между элементами (или такой тег необязателен). Это касается тегов HTML, HEAD, BODY, TBODY, TR, TD, LI (надеюсь, я ничего не забыла).
- **Блочный элемент добавлен внутрь строчного.** Необходимо закрыть все строчные элементы вплоть до следующего в иерархии блочного элемента. Если это не помогает, необходимо закрывать элементы, пока не появится возможность добавить нужный элемент или проигнорировать тег.

Синтаксический анализатор CSS

Во время синтаксического анализа файл CSS разбирается на объекты StyleSheet, содержащие правила CSS. Объект правил CSS содержит селектор и объявление, а также другие объекты, характерные для грамматики CSS.



Синтаксический анализ CSS

Порядок обработки скриптов и таблиц стилей

Скрипты

Веб-документы придерживаются синхронной модели. Предполагается, что скрипты будут анализироваться и исполняться сразу же, как только анализатор обнаружит тег `<script>`. Синтаксический анализ документа откладывается до завершения выполнения скрипта. Если речь идет о внешнем скрипте, сначала необходимо запросить сетевые ресурсы. Это также делается синхронно, а анализ откладывается до получения ресурсов. Такая модель использовалась много лет и даже занесена в спецификации HTML 4 и 5. Разработчик мог пометить скрипт тегом defer, чтобы синтаксический анализ документа можно было выполнять до завершения выполнения скрипта. В HTML5 появилась возможность пометить скрипт как асинхронный (asynchronous), чтобы он анализировался и выполнялся в другом потоке.

Ориентировочный синтаксический анализ

Этот механизм оптимизации используется и в WebKit, и в Firefox. При выполнении скриптов остальные части документа анализируются в другом потоке, чтобы оценить необходимые ресурсы и загрузить их из сети. Таким образом, ресурсы загружаются в параллельных потоках, что повышает общую скорость обработки. Обратите внимание: ориентировочный анализатор не изменяет дерево DOM (это работа основного анализатора), а лишь обрабатывает ссылки на внешние ресурсы, такие как внешние скрипты, таблицы стилей и картинки.

Таблицы стилей

Таблицы стилей основаны на другой модели. Так как они не вносят изменений в дерево DOM, теоретически останавливать анализ документа, чтобы дождаться их обработки, бессмысленно. Однако скрипты могут запрашивать данные о стилях на этапе синтаксического анализа документа. Если стиль еще не загружен и не проанализирован, скрипт может получить неверную информацию. Разумеется, это повлекло бы за собой целый ряд проблем. Если Firefox обнаруживает таблицу стилей, которая еще не загружена и не проанализирована, то все скрипты останавливаются. В WebKit они останавливаются только в случае, если пытаются извлечь свойства стилей, которые могут быть определены в незагруженных таблицах.

Построение дерева отображения

Во время построения дерева DOM браузер создает еще одну структуру – дерево отображения. В нем визуальные элементы размещаются в том порядке, в каком их необходимо вывести на экран. Это визуальное представление документа.

В Firefox элемент дерева отображения называется "фреймом" (frame). В WebKit используется термин "объект отображения" (render object).

Каждый объект отображения представляет собой прямоугольную область, он содержит геометрические данные, такие как ширина, высота и положение. Объекты отображения указывают на объекты style, содержащие негеометрическую информацию.

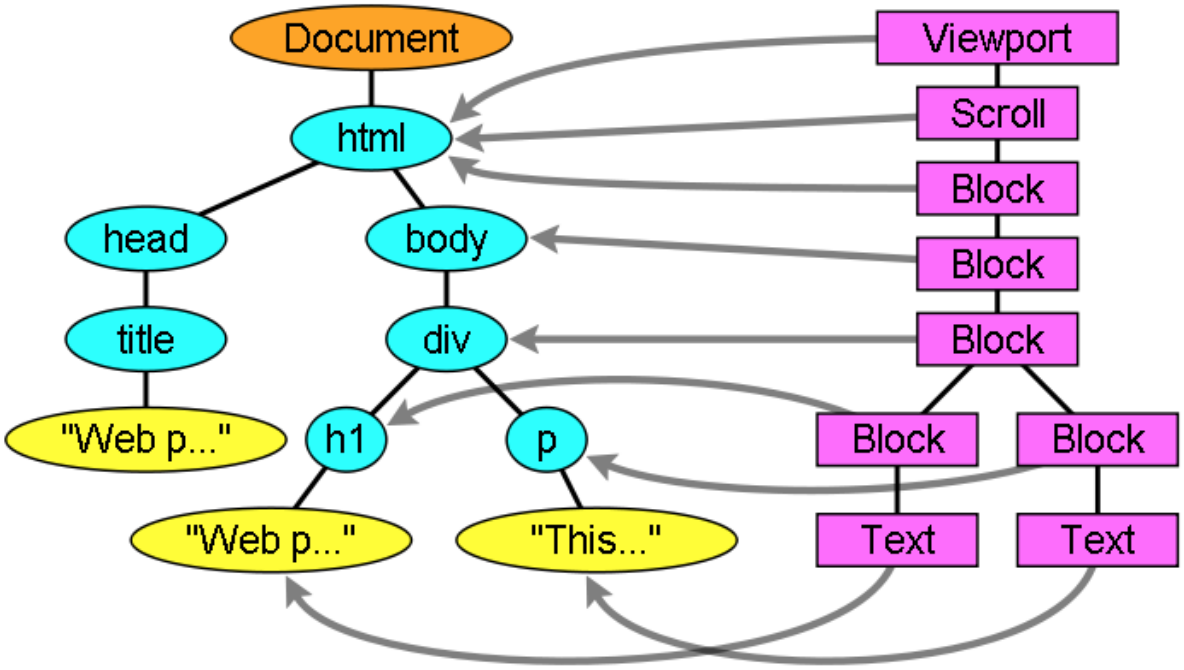
Как дерево отображения связано с деревом DOM

Объекты отображения соответствуют элементам DOM, но не идентичны им. Невизуальные элементы DOM не включаются в дерево отображения (примером может служить элемент head). Кроме того, в дерево не включаются элементы, у которых для свойства display задан атрибут none (элементы с атрибутом hidden включаются). Существуют и такие элементы DOM, которым соответствует сразу несколько визуальных объектов. Обычно это элементы со сложной структурой, которые невозможно описать одним-единственным прямоугольником. Например, элементу select соответствуют три визуальных объекта: один для области отображения, другой для раскрывающегося списка, третий для кнопки. Кроме того, если текст не вмещается на одну строку и

разбивается на фрагменты, новые строки добавляются как самостоятельные объекты отображения.

Еще одним примером, где используется несколько объектов отображения, является некорректно написанный код HTML. Согласно спецификации CSS, строчный элемент может содержать либо только блочные, либо только строчные элементы. Если же содержание смешанное, то в качестве оболочки для строчных объектов создаются анонимные блочные объекты.

Некоторым объектам отображения соответствует узел DOM, но их положения в дереве не совпадают. Плавающие элементы и элементы с абсолютными координатами исключаются из общего процесса, помещаются в отдельную часть дерева и затем отображаются в стандартном фрейме, хотя на самом деле должны отображаться во фрейме-заполнителе.



Дерево отображения и соответствующее ему дерево DOM. Viewport (область просмотра) – это главный контейнер. В WebKit он представлен объектом RenderView.

Процесс построения дерева

В Firefox визуальное представление регистрируется как слушатель обновлений DOM. Создание фреймов делегируется конструктору FrameConstructor, который определяет стили (см. Вычисление стилей) и создает фрейм.

В WebKit процесс определения стиля и создания объекта отображения называется совмещением (attachment). Каждый узел DOM имеет метод attach. Совмещение

выполняется синхронно; при добавлении нового узла в дерево DOM для него вызывается метод attach.

В результате обработки тегов html и body создается корневой объект дерева отображения. В спецификации CSS он называется контейнером – блоком верхнего уровня, в котором содержатся все остальные блоки. Его размеры формируют область просмотра, то есть часть окна браузера, в которой будет показано содержание. Это объект отображения, на который указывает документ. Остальное дерево строится посредством добавления в него узлов DOM.

Вычисление стилей

Чтобы построить дерево отображения, необходимо рассчитать визуальные свойства каждого объекта. Для этого вычисляются свойства стиля каждого элемента.

Стиль определяется различными таблицами стилей, строчными элементами style и визуальными свойствами в документе HTML (такими как bgcolor). Последние переводятся в свойства CSS.

С вычислением стилей связаны некоторые сложности:

- Данные стилей содержат множество свойств и бывают очень объемны, что может вести к проблемам с памятью.
- Поиск подходящих правил для каждого элемента может замедлить работу, если код не оптимизирован. Если подставлять к каждому элементу все правила по очереди, это заметно отразится на производительности. Селекторы могут иметь сложную структуру, поэтому даже если определенная последовательность правил сначала покажется подходящей, в ходе анализа может оказаться, что это не так, и придется пробовать другой вариант.
- Применение правил подразумевает определение иерархии для достаточно сложных перекрывающихся правил.

Для решения поставленных задач браузеры используют различные механизмы. Для проблем 1 и 3 браузеры на базе Webkit совместно используют информацию о стилях, а Firefox использует дополнительные структуры, такие как дерево стилей и контексты стилей.

Классификация правил для упрощения сопоставления

Как мы уже знаем, правила стилей извлекаются из нескольких источников (внешние таблицы, инлайн стили, визуальные атрибуты).

Последние два источника легко сопоставить с элементом, так как он содержит атрибуты объекта style и при сопоставлении атрибутов HTML может служить ключом. Но в случаях с внешними таблицами стилей, сопоставление правил CSS не так однозначно. Чтобы упростить задачу, правила классифицируются.

После синтаксического анализа таблицы стилей каждое правило добавляется в одну или в несколько хэш-карт в зависимости от селектора. Существуют карты, организованные по идентификатору, названию класса или тега, а также общие карты для всех остальных случаев. Если селектором является идентификатор, правило добавляется в карту идентификаторов, если класс, то в карту классов и т. д.
Такая классификация упрощает поиск подходящих правил. Нам не приходится проверять все объявления: достаточно извлечь из карты подходящие правила. Такая классификации позволяет сразу отбросить более 95% правил, что ускоряет и упрощает процесс сопоставления.

Рассмотрим пример со следующими правилами стилей:

```
p.error {color:red}
#messageDiv {height:50px}
div {margin:5px}
```

Первое правило будет помещено в карту классов, второе – в карту идентификаторов, а третье – в карту тегов.

Рассмотрим следующий код HTML:

```
<p class="error">an error occurred </p>
<div id=" messageDiv">this is a message</div>
```

Сначала найдем правила для элемента p. В карте классов содержится ключ error, по которому находим правило p.error. Правила, соответствующие элементу div, содержатся в карте идентификаторов (по ключу id) и в карте тегов. Осталось только определить, какие из правил, найденных по ключам, являются подходящими.

Предположим, правило для элемента div таково:

```
table div {margin:5px}
```

Мы в любом случае извлекли бы его из карты тегов, так как ключом является крайний правый селектор, однако оно не подошло бы для этого элемента div, потому что для него не существует родительской таблицы.

Применение правил в порядке приоритета

Свойства объекта style отвечают всем визуальным атрибутам (всем атрибутам CSS, но на более универсальном уровне). Если свойство не определяется ни одним из подходящих правил, в некоторых случаях оно может быть унаследовано от родительского объекта style. В других случаях используется значение по умолчанию.

Сложности начинаются, если существует более одного определения, и тогда, чтобы разрешить конфликт, требуется установить порядок приоритета.

Порядок приоритета таблиц стилей

Объявление свойства объекта style может содержаться сразу в нескольких таблицах стилей, иногда по нескольку раз в одной таблице. В таком случае очень важно установить верный порядок применения правил. Такой порядок называется каскадным. В спецификации CSS2 указан следующий порядок приоритета (по возрастанию).

1. Объявления браузера
2. Обычные объявления пользователя
3. Обычные объявления автора
4. Важные объявления автора
5. Важные объявления пользователя

Объявления браузера имеют самый низкий приоритет, а объявления пользователя важнее объявлений автора, только если имеют пометку !important. Объявления с одинаковым приоритетом сортируются по степени специфичности, а затем по порядку, в котором были определены. Визуальные атрибуты HTML переводятся в соответствующие объявления CSS и обрабатываются как правила автора с низким приоритетом.

Специфичность

Специфичность селектора определена в спецификации CSS2 описанным ниже образом.

- Если объявление содержится в атрибуте style, а не в правиле с селектором, выбирается значение 1, в противном случае – 0 (= a)
- Количество атрибутов ID внутри селектора (= b)
- Количество других атрибутов и псевдоклассов внутри селектора (= c)
- Количество названий элементов и псевдоэлементов внутри селектора (= d)

Объединение этих значений в последовательность a-b-c-d (в системе счисления с большим основанием) и определяет специфичность.
Основание системы счисления определяется самым большим числом в любой из категорий.
Например, если a=14, можно использовать шестнадцатеричную систему. Если a=17 (что маловероятно), потребуется система счисления по основанию 17. Такая ситуация может возникнуть, если имеется селектор такого типа: html body div div p... Но вряд ли внутри селектора будет 17 тегов.

Примеры вычисления специфичности:

```
*             {}  /* 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 */
ul ol+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 */
```

Сортировка правил

После сопоставления правил они сортируются согласно приоритету. В WebKit для коротких списков используется сортировка простыми обменами, а для длинных – сортировка слиянием.

Многоэтапное применение правил

В WebKit используется специальный флаг, который указывает, загружены ли все таблицы стилей верхнего уровня (включая @imports). Если совмещение уже началось, а таблица стилей еще не загружена целиком, используются плейсхолдеры, а в документе появляются соответствующие пометки. После завершения загрузки таблицы плейсхолдеры пересчитываются.

Компоновка

Когда только что созданный объект отображения включается в дерево, он не имеет ни размера, ни положения. Расчет этих значений называется компоновкой (layout или reflow).

В HTML используется поточная модель компоновки, то есть в большинстве случае геометрические данные можно рассчитать за один проход. Элементы, встречающиеся в потоке позднее, не влияют на геометрию уже обработанных элементов. Существуют исключения: например, для компоновки таблиц HTML может потребоваться более одного цикла.

Система координат рассчитывается на основе корневого фрейма. Используются верхняя и левая координаты.

Компоновка выполняется в несколько циклов. Она начинается с корневого объекта отображения, соответствующего элементу <html> в HTML-документе. Затем обрабатывается иерархия фреймов (или отдельные ее части), и геометрическая информация рассчитывается для объектов отображения, которым она необходима.

Корневой объект отображения имеет координаты (0; 0), а его размеры соответствуют области просмотра (видимой части окна браузера). Любой объект отображения может при необходимости вызвать метод layout или reflow для своих дочерних элементов.

Система «грязных битов»

Чтобы не выполнять перекомпоновку при каждом изменении, браузеры используют так называемую систему «грязных битов». Измененный объект отображения и его дочерние элементы помечаются как «грязные», то есть требующие перекомпоновки.

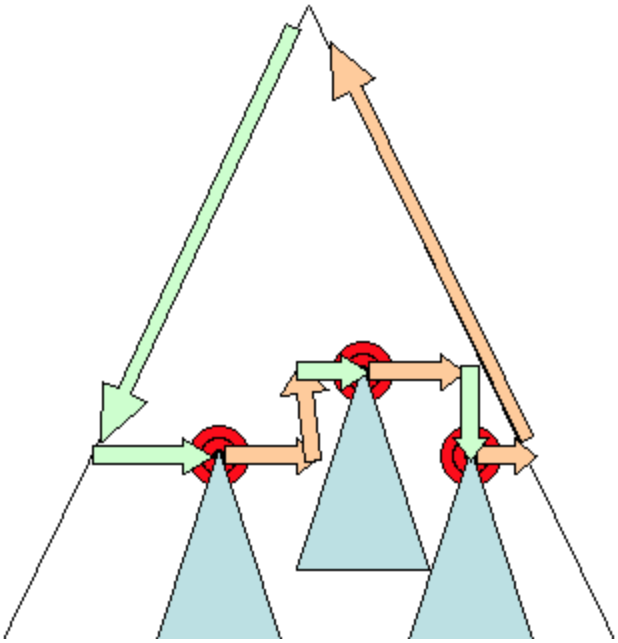
Используется два флага: dirty и children are dirty. Флаг children are dirty означает, что перекомпоновка требуется не самому объекту отображения, а одному или нескольким из его дочерних объектов.

Глобальная и инкрементная компоновка

Если компоновка выполняется для всего дерева отображения, она называется глобальной. Ее могут вызывать перечисленные ниже события.

- Глобальное изменение стиля, который используется во всех объектах отображения, например изменение шрифта.
- Изменение размеров экрана.

При инкрементной компоновке изменяются только «грязные» объекты отображения (при этом может потребоваться перекомпоновка некоторых других объектов).
Инкрементная компоновка выполняется асинхронно и начинается при обнаружении «грязных» объектов отображения. Пример: после получения содержания из сети и его добавления в дерево DOM в дереве отображения появляется новый объект.



Инкрементная компоновка, при которой обрабатываются только "грязные" объекты отображения и их дочерние элементы

Синхронная и асинхронная компоновка

Инкрементная компоновка выполняется асинхронно. В Firefox команды инкрементной компоновки помещаются в очередь, а затем планировщик вызывает их все вместе. В WebKit выполнение инкрементной компоновки также откладывается, чтобы обработать целое дерево за один цикл и перекомпоновать все "грязные" объекты отображения.

Скрипты, запрашивающие данные о стилях, такие как offsetHeight, могут привести к синхронному выполнению инкрементной компоновки.
Глобальная компоновка обычно выполняется синхронно.
Иногда компоновка выполняется в обратном вызове после исходной компоновки, потому что меняются значения некоторых атрибутов, таких как положение прокрутки.

Оптимизация

Если компоновка вызвана событием resize или изменением положения (но не размера) объекта отображения, размеры объекта извлекаются из кэша и не рассчитываются заново.
Если меняется только часть дерева, перекомпоновка всего дерева не выполняется. Это происходит, если изменение носит локальный характер и не влияет на окружающие объекты, например при вводе текста в текстовые поля (в остальных случаях ввод каждого символа вызывает перекомпоновку всего дерева).

Процесс компоновки

Компоновка обычно выполняется по описанной ниже схеме.

1. Родительский объект отображения определяет собственную ширину.
2. Родительский объект отображения обрабатывает дочерние элементы:
    a. определяет положение дочернего объекта отображения (задает его координаты x и y);
    b. вызывает компоновку дочернего элемента (если он помечен как "грязный", если выполняется глобальная перекомпоновка и т. д.), в результате чего рассчитывается его высота.
3. На основе суммарной высоты дочерних элементов, а также высоты полей и отступов рассчитывается высота родительского объекта отображения: она требуется его собственному родительскому объекту.
4. Снимаются флаги «грязных» битов.

Отрисовка

На этапе отрисовки для каждого объекта отображения по очереди вызывается метод paint и их содержание выводится на экран. Для отрисовки используется компонент инфраструктуры пользовательского интерфейса.

Глобальная и инкрементная отрисовка

При глобальной отрисовке все дерево отрисовывается целиком, а при инкрементной – только отдельные объекты отображения, не влияющие на остальные части дерева.
Измененный объект отображения помечает свой прямоугольник как недействительный. Операционная система расценивает его как "грязную" область и вызывает событие paint. Области при этом объединяются, чтобы отрисовку можно было выполнить сразу для всех. В браузере Chrome отрисовка выполняется несколько сложнее, так как объект отображения находится вне главного процесса: Chrome в некоторой степени имитирует

поведение операционной системы. Компонент визуального представления прослушивает эти события и делегирует сообщение корневому объекту отображения. Все объекты дерева по очереди проверяются, пока не будет найден нужный. Затем выполняется отрисовка его самого и, как правило, его дочерних элементов.

Порядок отрисовки

Порядок отрисовки определен в спецификации CSS2. Фактически он соответствует порядку помещения элементов в контексты стеков. Порядок отрисовки играет важную роль, так как стеки отрисовываются задом наперед. Порядок добавления блочных объектов в стек таков:

1. Цвет фона
2. Фоновое изображение
3. Рамка
4. Дочерние объекты
5. Внешние границы

Динамические изменения

При наступлении изменений браузеры стараются не выполнять лишних операций. Например, при изменении цвета одного элемента остальные не отрисовываются заново. При изменении положения элемента выполняется повторная компоновка и отрисовка его самого, его дочерних элементов и, возможно, других объектов того же уровня. При добавлении узла DOM выполняется его повторная компоновка и отрисовка. Серьезные изменения, такие как увеличение размера шрифта элемента html, ведут к очистке кэша и повторной компоновке и отрисовке целого дерева.