브라우저의 동작 원리

1. HTML 마크업을 처리하고 DOM 트리를 빌드한다. (**"무엇을"** 그릴지 결정한다.)
2. CSS 마크업을 처리하고 CSSOM 트리를 빌드한다. (**"어떻게"** 그릴지 결정한다.)
3. DOM 및 CSSOM 을 결합하여 렌더링 트리를 형성한다. (**"화면에 그려질 것만"** 결정)
4. 렌더링 트리에서 레이아웃을 실행하여 각 노드의 기하학적 형태를 계산한다. (**"Box-Model"** 을 생성한다.)
5. 개별 노드를 화면에 페인트한다.(or 래스터화)

Reference

- Naver D2 - 브라우저의 작동 원리
- Web fundamentals - Critical-rendering-path
- 브라우저의 Critical path (한글)

뒤로/위로

🔍 NAVER D2 - 브라우저는 어떻게 동작하는가?

2012.05.18

이 글은 이스라엘 개발자 탈리 가르시엘(Tali Garsiel)이 html5rocks.com에 게시한 "How Browsers Work: Behind the scenes of modern web browsers"를 번역한 글입니다. 탈리 가르시엘은 몇 년간 브라우저 내부와 관련된 공개 자료를 확인하고, C++ 코드 수백만 줄 분량의 WebKit이나 Gecko 같은 오픈 소스 렌더링 엔진의 소스 코드를 직접 분석하면서 어떻게 브라우저가 동작하는지 파악했습니다.

소개

브라우저는 아마도 가장 많이 사용하는 소프트웨어일 것이다. 이 글을 통해 브라우저가 어떻게 동작하는지 설명하려고 한다. 이 글을 읽고 나면, 브라우저 주소 창에 naver.com을 입력했을 때 어떤 과정을 거쳐 네이버 페이지가 화면에 보이게 되는지 알게 될 것이다.

이 글에서 설명하는 브라우저

최근에는 인터넷 익스플로러, 파이어폭스, 사파리, 크롬, 오페라 이렇게 다섯 개의 브라우저를 많이 사용하지만 나는 파이어폭스, 크롬, 사파리와 같은 오픈소스 브라우저를 예로 들 것이다. 사파리는 부분적으로 오픈소스이다. StatCounter 브라우저 통계에 의하면 2012년 3월 현재 파이어폭스, 사파리, 크롬의 점유율은 62.57%에 달한다. 오픈소스 브라우저가 시장의 상당 부분을 차지하게 된 것이다.

브라우저의 주요 기능

브라우저의 주요 기능은 사용자가 선택한 자원을 서버에 요청하고 브라우저에 표시하는 것이다. 자원은 보통 HTML 문서지만 PDF나 이미지 또는 다른 형태일 수 있다. 자원의 주소는 URI(Uniform Resource Identifier)에 의해 정해진다.

브라우저는 HTML과 CSS 명세에 따라 HTML 파일을 해석해서 표시하는데 이 명세는 웹 표준화 기구인 W3C(World Wide Web Consortium)에서 정한다. 과거에는 브라우저들이 일부만 이 명세에 따라 구현하고 독자적인 방법으로 확장함으로써 웹 제작자가 심각한 호환성 문제를 겪었지만 최근에는 대부분의 브라우저가 표준 명세를 따른다.

브라우저의 사용자 인터페이스는 서로 닮아 있는데 다음과 같은 요소들이 일반적이다.

- URI를 입력할 수 있는 주소 표시 줄
- 이전 버튼과 다음 버튼
- 북마크
- 새로 고침 버튼과 현재 문서의 로드를 중단할 수 있는 정지 버튼
- 홈 버튼

브라우저의 사용자 인터페이스는 표준 명세가 없음에도 불구하고 수 년간 서로의 장점을 모방하면서 현재에 이르게 되었다. HTML5 명세는 주소 표시줄, 상태 표시줄, 도구 모음과 같은 일반적인 요소를 제외하고 브라우저의 필수 UI를 정의하지 않았다. 물론 파이어폭스의 다운로드 관리자와 같이 브라우저에 특화된 기능도 있다.

브라우저의 기본 구조

브라우저의 주요 구성 요소는 다음과 같다.(1.1)

1. 사용자 인터페이스 - 주소 표시줄, 이전/다음 버튼, 북마크 메뉴 등. 요청한 페이지를 보여주는 창을 제외한 나머지 모든 부분이다.
2. 브라우저 엔진 - 사용자 인터페이스와 렌더링 엔진 사이의 동작을 제어.
3. 렌더링 엔진 - 요청한 콘텐츠를 표시. 예를 들어 HTML을 요청하면 HTML과 CSS를 파싱하여 화면에 표시함.
4. 통신 - HTTP 요청과 같은 네트워크 호출에 사용됨. 이것은 플랫폼 독립적인 인터페이스이고 각 플랫폼 하부에서 실행됨.
5. UI 백엔드 - 콤보 박스와 창 같은 기본적인 장치를 그림. 플랫폼에서 명시하지 않은 일반적인 인터페이스로서, OS 사용자 인터페이스 체계를 사용.
6. 자바스크립트 해석기 - 자바스크립트 코드를 해석하고 실행.
7. 자료 저장소 - 이 부분은 자료를 저장하는 계층이다. 쿠키를 저장하는 것과 같이 모든 종류의 자원을 하드 디스크에 저장할 필요가 있다. HTML5 명세에는 브라우저가 지원하는 '웹 데이터 베이스'가 정의되어 있다.

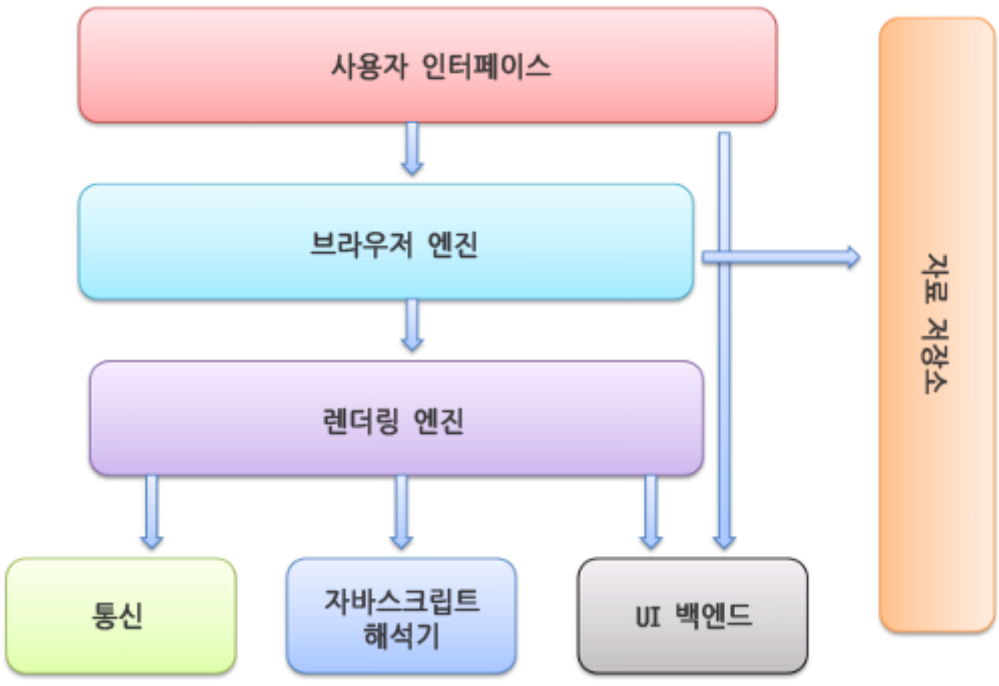


그림 1 브라우저의 주요 구성 요소

크롬은 대부분의 브라우저와 달리 각 탭마다 별도의 렌더링 엔진 인스턴스를 유지하는 것이 주목할만하다. 각 탭은 독립된 프로세스로 처리된다.

렌더링 엔진

렌더링 엔진의 역할은 요청 받은 내용을 브라우저 화면에 표시하는 일이다.

렌더링 엔진은 HTML 및 XML 문서와 이미지를 표시할 수 있다. 물론 플러그인이나 브라우저 확장 기능을 이용해 PDF와 같은 다른 유형도 표시할 수 있다. 그러나 이 장에서는 HTML과 이미지를 CSS로 표시하는 주된 사용 패턴에 초점을 맞출 것이다.

렌더링 엔진들

이 글에서 다루는 브라우저인 파이어폭스와 크롬, 사파리는 두 종류의 렌더링 엔진으로 제작되었다. 파이어폭스는 모질라에서 직접 만든 게코(Gecko) 엔진을 사용하고 사파리와 크롬은 웹킷(Webkit) 엔진을 사용한다.

웹킷은 최초 리눅스 플랫폼에서 동작하기 위해 제작된 오픈소스 엔진인데 애플이 맥과 윈도우즈에서 사파리 브라우저를 지원하기 위해 수정을 가했다. 더 자세한 내용은 webkit.org를 참조한다.

동작 과정

렌더링 엔진은 통신으로부터 요청한 문서의 내용을 얻는 것으로 시작하는데 문서의 내용은 보통 8KB 단위로 전송된다.

다음은 렌더링 엔진의 기본적인 동작 과정이다.

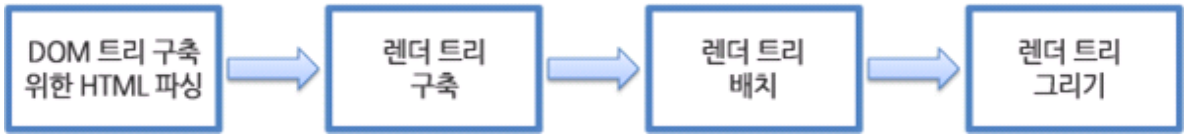


그림 2 렌더링 엔진의 동작 과정

렌더링 엔진은 HTML 문서를 파싱하고 "콘텐츠 트리" 내부에서 태그를 DOM 노드로 변환한다. 그 다음 외부 CSS 파일과 함께 포함된 스타일 요소도 파싱한다. 스타일 정보와 HTML 표시 규칙은 "렌더 트리"라고 부르는 또 다른 트리를 생성한다.

렌더 트리는 색상 또는 면적과 같은 시각적 속성이 있는 사각형을 포함하고 있는데 정해진 순서대로 화면에 표시된다.

렌더 트리 생성이 끝나면 배치가 시작되는데 이것은 각 노드가 화면의 정확한 위치에 표시되는 것을 의미한다. 다음은 UI 백엔드에서 렌더 트리의 각 노드를 가로지르며 형상을 만들어 내는 그리기 과정이다.

일련의 과정들이 점진적으로 진행된다는 것을 아는 것이 중요하다. 렌더링 엔진은 좀 더 나은 사용자 경험을 위해 가능하면 빠르게 내용을 표시하는데 모든 HTML을 파싱할 때까지 기다리지 않고 배치와 그리기 과정을 시작한다. 네트워크로부터 나머지 내용이 전송되기를 기다리는 동시에 받은 내용의 일부를 먼저 화면에 표시하는 것이다.

동작 과정 예

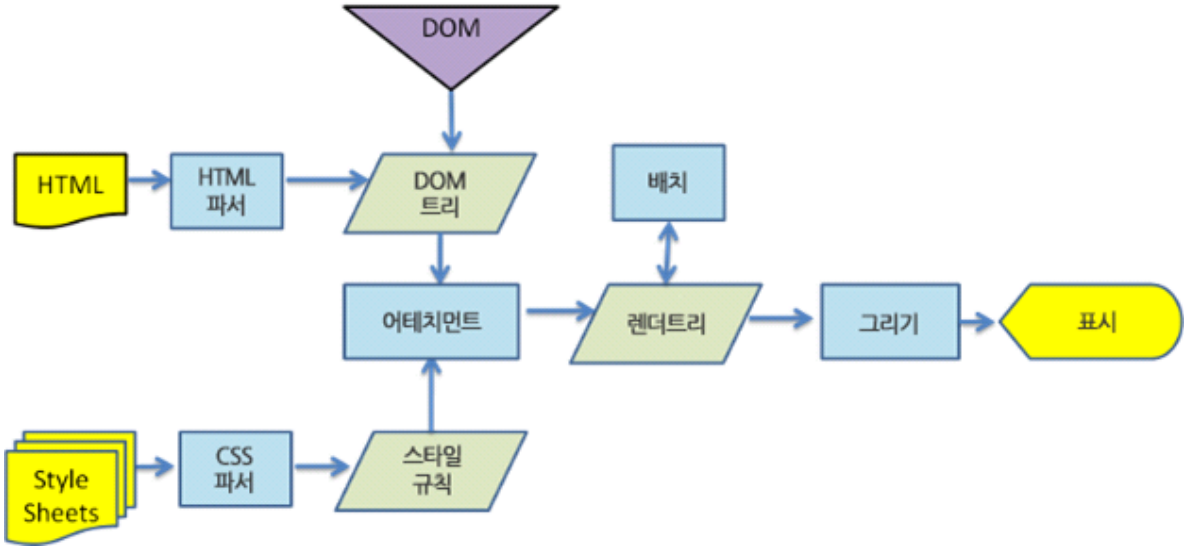


그림 3 웹킷 동작 과정

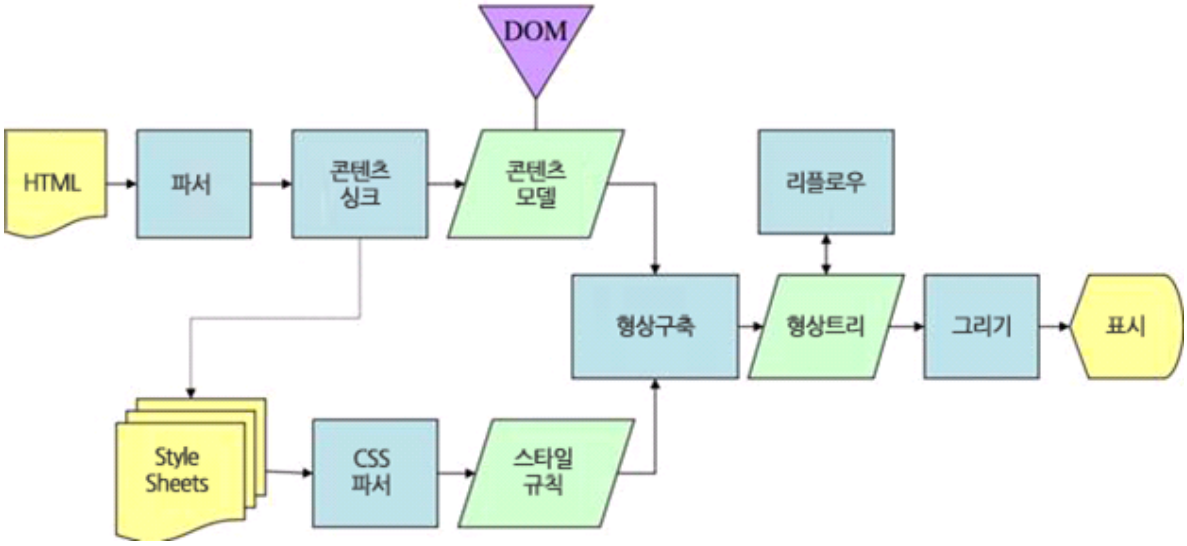


그림 4 모질라의 게코 렌더링 엔진 동작 과정(3.6)

웹킷과 게코가 용어를 약간 다르게 사용하고 있지만 동작 과정은 기본적으로 동일하다는 것을 그림 3과 그림 4에서 알 수 있다.

게코는 시각적으로 처리되는 렌더 트리를 "형상 트리(frame tree)"라고 부르고 각 요소를 형상(frame)이라고 하는데 웹킷은 "렌더 객체(render object)"로 구성되어 있는 "렌더 트리(render tree)"라는 용어를 사용한다. 웹킷은 요소를 배치하는데 "배치(layout)" 라는 용어를 사용하지만 게코는 "리플로(reflow)"라고 부른다. "어태치먼트(attachment)"는 웹킷이 렌더 트리를 생성하기 위해 DOM 노드와 시각 정보를 연결하는 과정이다. 게코는 HTML과 DOM 트리 사이에 "콘텐츠 싱크(content sink)"라고 부르는 과정을 두는데 이는 DOM 요소를 생성하는 공정으로 웹킷과 비교하여 의미있는 차이점이라고 보지는 않는다.

파싱과 DOM 트리 구축

파싱 일반

파싱은 렌더링 엔진에서 매우 중요한 과정이기 때문에 더 자세히 다룰 필요가 있다. 파싱에 대한 간단한 소개로 시작한다.

문서 파싱은 브라우저가 코드를 이해하고 사용할 수 있는 구조로 변환하는 것을 의미한다. 파싱 결과는 보통 문서 구조를 나타내는 노드 트리인데 파싱 트리(parse tree) 또는 문법 트리(syntax tree)라고 부른다.

예를 들면 2+3-1과 같은 표현식은 다음과 같은 트리가 된다.

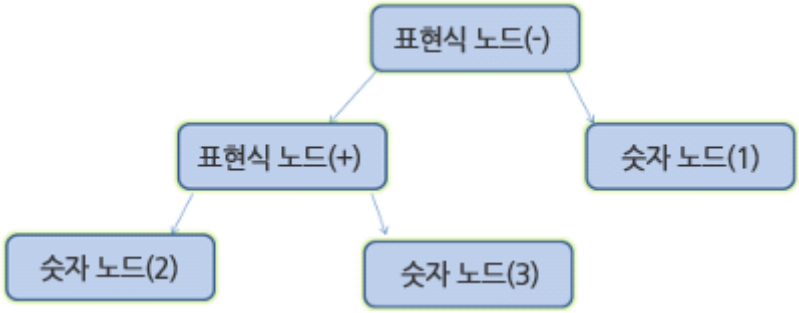


그림 5 수학 표현식을 파싱한 트리 노드

문법

파싱은 문서에 작성된 언어 또는 형식의 규칙에 따르는데 파싱할 수 있는 모든 형식은 정해진 용어와 구문 규칙에 따라야 한다. 이것을 문맥 자유 문법이라고 한다. 인간의 언어는 이런 모습과는 다르기 때문에 기계적으로 파싱이 불가능하다.

파서-어휘 분석기 조합

파싱은 어휘 분석과 구문 분석이라는 두 가지로 구분할 수 있다.

어휘 분석은 자료를 토큰으로 분해하는 과정이다. 토큰은 유효하게 구성된 단위의 집합체로 용어집이라고도 할 수 있는데 인간의 언어로 말하자면 사전에 등장하는 모든 단어에 해당된다.

구문 분석은 언어의 구문 규칙을 적용하는 과정이다.

파서는 보통 두 가지 일을 하는데 자료를 유효한 토큰으로 분해하는 어휘 분석기(토큰 변환기 라고도 부름)가 있고 언어 구문 규칙에 따라 문서 구조를 분석함으로써 파싱 트리를 생성하는 파서가 있다. 어휘 분석기는 공백과 줄 바꿈 같은 의미 없는 문자를 제거한다.

그림 6 문서 소스로부터 파싱 트리를 만드는 과정

파싱 과정은 반복된다. 파서는 보통 어휘 분석기로부터 새 토큰을 받아서 구문 규칙과 일치하는지 확인한다. 규칙에 맞으면 토큰에 해당하는 노드가 파싱 트리에 추가되고 파서는 또 다른 토큰을 요청한다.

규칙에 맞지 않으면 파서는 토큰을 내부적으로 저장하고 토큰과 일치하는 규칙이 발견될 때까지 요청한다. 맞는 규칙이 없는 경우 예외로 처리하는데 이것은 문서가 유효하지 않고 구문 오류를 포함하고 있다는 의미다.

변환

파서 트리는 최종 결과물이 아니다. 파싱은 보통 문서를 다른 양식으로 변환하는데 컴파일이 하나의 예가 된다. 소스 코드를 기계 코드로 만드는 컴파일러는 파싱 트리 생성 후 이를 기계 코드 문서로 변환한다.

그림 7 컴파일 과정

파싱 예

그림 5에서는 수학 표현식을 파싱 트리로 만들어 보았다. 간단한 수학 언어를 정의하고 파싱 과정을 살펴 보자.

어휘: 수학 언어는 정수, 더하기 기호, 빼기 기호를 포함한다.

구문:

1. 언어 구문의 기본적인 요소는 표현식, 항, 연산자이다.
2. 언어에 포함되는 표현식의 수는 제한이 없다.
3. 표현식은 "항" 뒤에 "연산자" 그 뒤에 또 다른 항이 따르는 형태로 정의한다.
4. 연산자는 더하기 토큰 또는 빼기 토큰이다.
5. 정수 토큰 또는 하나의 표현식은 항이다.

입력된 값 2+3-1을 분석해 보자.

규칙에 맞는 첫 번째 부분 문자열은 2이다. 규칙 5번에 따르면 이것은 하나의 항이다. 두 번째로 맞는 것은 2+3 인데 이것은 항 뒤에 연산자와 또 다른 항이 등장한다는 세 번째 규칙과도 일치한다. 입력 값의 마지막 부분까지 진행하면 또 다른 일치를 발견할 수 있다. 2+3은 항과 연산자와 항으로 구성된 하나의 새로운 항이라는 것을 알고 있기 때문에 2+3-1은 하나의 표현식이 된다. 2++은 어떤 규칙과도 맞지 않기 때문에 유효하지 않은 입력이 된다.

어휘와 구문에 대한 공식적인 정의

어휘는 보통 정규 표현식으로 표현한다. 예를 들면 언어는 다음과 같이 정의될 것이다.

```
INTEGER : 0|[1-9][0-9]*
PLUS : +
MINUS : -
```

보시다시피 정수는 정규 표현식으로 정의한다.

구문은 보통 BNF 라고 부르는 형식에 따라 정의한다. 언어는 다음과 같이 정의될 것이다.

```
expression := term operation term
operation := PLUS | MINUS
term := INTEGER | expression
```

문법이 문맥 자유 문법이라면 언어는 정규 파서로 파싱할 수 있다. 문맥 자유 문법을 쉽게 말하면 완전히 BNF로 표현 가능한 문법이다. 공식적인 정의는 위키백과의 문맥 자유 문법을 참조한다.

파서의 종류

파서는 기본적으로 하향식 파서와 상향식 파서가 있다. 하향식 파서는 구문의 상위 구조로부터 일치하는 부분을 찾기 시작하는데 반해 상향식 파서는 낮은 수준에서 점차 높은 수준으로 찾는다.

두 종류의 파서가 예제를 어떻게 파싱하는지 살펴보자.

하향식 파서는 2+3과 같은 표현식에 해당하는 높은 수준의 규칙을 먼저 찾는다. 그 다음 표현식으로 2+3-1을 찾을 것이다. 표현식을 찾는 과정은 일치하는 다른 규칙을 점진적으로 더 찾아내는 방식인데 어쨌거나 가장 높은 수준의 규칙을 먼저 찾는 것으로부터 시작한다.

상향식 파서는 입력 값이 규칙에 맞을 때까지 찾아서 맞는 입력 값을 규칙으로 바꾸는데 이 과정은 입력 값의 끝까지 진행된다. 부분적으로 일치하는 표현식은 파서 스택에 쌓인다.

스택	입력 값
	2+3-1
항	+3-1
항 연산자	3-1
표현식	-1
표현식 연산자	1
표현식	

상향식 파서는 입력 값의 오른쪽으로 이동하면서(입력 값의 처음을 가리키는 포인터가 오른쪽으로 이동하는 것을 상상) 구문 규칙으로 갈수록 남는 것이 점차 감소하기 때문에 이동-감소 파서라고 부른다.

파서 자동 생성

파서를 생성해 줄 수 있는 도구를 파서 생성기라고 한다. 언어에 어휘나 구문 규칙 같은 문법을 부여하면 동작하는 파서를 만들어 준다. 파서를 생성하는 것은 파싱에 대한 깊은 이해를 필요로 하고 수동으로 파서를 최적화하여 생성하는 것은 쉬운 일이 아니기 때문에 파서 생성기는 매우 유용하다.

웹킷은 잘 알려진 두 개의 파서 생성기를 사용한다. 어휘 생성을 위한 플렉스(Flex)와 파서 생성을 위한 바이슨(Bison)이다. 렉스(Lex)와 약(Yacc)이라는 이름과 함께 들어본 적이 있을지도 모르겠다. 플렉스는 토큰의 정규 표현식 정의를 포함하는 파일을 입력 받고 바이슨은 BNF 형식의 언어 구문 규칙을 입력 받는다.

HTML 파서

HTML 파서는 HTML 마크업을 파싱 트리로 변환한다.

HTML 문법 정의

HTML의 어휘와 문법은 W3C에 의해 명세로 정의되어 있다. 현재 버전은 HTML4와 초안 상태로 진행 중인 HTML5 이다.

문맥 자유 문법이 아님

파싱 일반 소개를 통해 알게 된 것처럼 문법은 BNF와 같은 형식을 이용하여 공식적으로 정의할 수 있다.

안타깝게도 모든 전통적인 파서는 HTML에 적용할 수 없다. 그럼에도 불구하여 지금까지 파싱을 설명한 것은 그냥 재미 때문은 아니다. 파싱은 CSS와 자바스크립트를 파싱하는 데 사용된다. HTML은 파서가 요구하는 문맥 자유 문법에 의해 쉽게 정의할 수 없다.

HTML 정의를 위한 공식적인 형식으로 DTD(문서 형식 정의)가 있지만 이것은 문맥 자유 문법이 아니다.

이것은 언뜻 이상하게 보일 수도 있는데 HTML이 XML과 유사하기 때문이다. 사용할 수 있는 XML 파서는 많다. HTML을 XML 형태로 재구성한 XHTML도 있는데 무엇이 큰 차이점일까?

차이점은 HTML이 더 "너그럽다"는 점이다. HTML은 암묵적으로 태그에 대한 생략이 가능하다. 가끔 시작 또는 종료 태그 등을 생략한다. 전반적으로 뻣뻣하고 부담스러운 XML에 반하여 HTML은 "유연한" 문법이다.

이런 작은 차이가 큰 차이를 만들어 낸다. 웹 제작자의 실수를 너그럽게 용서하고 편하게 만들어주는 이것이야 말로 HTML이 인기가 있었던 이유다. 다른 한편으로는 공식적인 문법으로 작성하기 어렵게 만드는 문제가 있다. 정리하자면 HTML은 파싱하기 어렵고 전통적인 구문 분석이 불가능하기 때문에 문맥 자유 문법이 아니라는 것이다. XML 파서로도 파싱하기 쉽지 않다.

HTML DTD

HTML의 정의는 DTD 형식 안에 있는데 SGML 계열 언어의 정의를 이용한 것이다. 이 형식은 허용되는 모든 요소와 그들의 속성 그리고 중첩 구조에 대한 정의를 포함한다. 앞서 말 한대로 HTML DTD는 문맥 자유 문법이 아니다.

DTD는 여러 변종이 있다. 엄격한 형식은 명세만을 따르지만 다른 형식은 낡은 브라우저에서 사용된 마크업을 지원한다. 낡은 마크업을 지원하는 이유는 오래된 콘텐츠에 대한 하위 호환성 때문이다. 현재의 엄격한 형식 DTD는 www.w3.org/TR/html4/strict.dtd 에서 확인할 수 있다.

DOM

"파싱 트리"는 DOM 요소와 속성 노드의 트리로서 출력 트리가 된다. DOM은 문서 객체 모델(Document Object Model)의 준말이다. 이것은 HTML 문서의 객체 표현이고 외부를 향하는 자바스크립트와 같은 HTML 요소의 연결 지점이다. 트리의 최상위 객체는 문서이다.

DOM은 마크업과 1:1의 관계를 맺는다. 예를 들면 이런 마크업이 있다.

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

이것은 아래와 같은 DOM 트리로 변환할 수 있다.

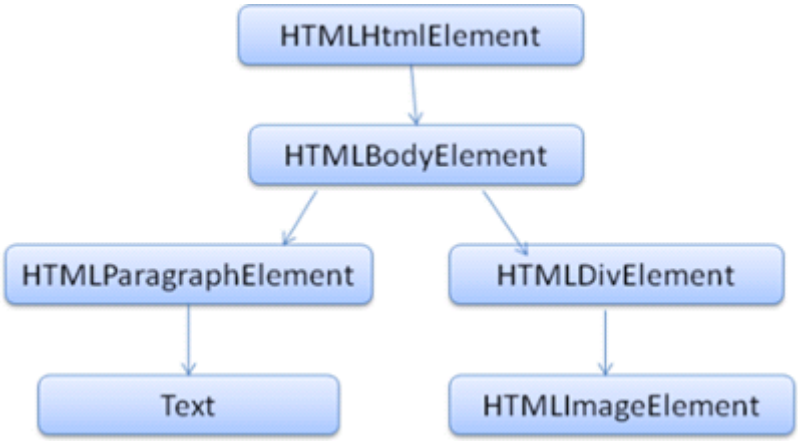


그림 8 예제 마크업의 DOM 트리

HTML과 마찬가지로 DOM은 W3C에 의해 명세(www.w3.org/DOM/DOMTR)가 정해져 있다. 이것은 문서를 다루기 위한 일반적인 명세인데 부분적으로 HTML 요소를 설명하기도 한다. HTML 정의는 www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/idl-definitions.html에서 찾을 수 있다.

트리가 DOM 노드를 포함한다고 말하는 것은 DOM 접점의 하나를 실행하는 요소를 구성한다는 의미다. 브라우저는 내부의 다른 속성들을 이용하여 이를 구체적으로 실행한다.

파싱 알고리즘

앞서 말한대로 HTML은 일반적인 하향식 또는 상향식 파서로 파싱이 안되는데 그 이유는 다음과 같다.

1. 언어의 너그러운 속성.
2. 잘 알려져 있는 HTML 오류에 대한 브라우저의 관용.
3. 변경에 의한 재파싱. 일반적으로 소스는 파싱하는 동안 변하지 않지만 HTML에서 document.write을 포함하고 있는 스크립트 태그는 토큰을 추가할 수 있기 때문에 실제로는 입력 과정에서 파싱이 수정된다.

일반적인 파싱 기술을 사용할 수 없기 때문에 브라우저는 HTML 파싱을 위해 별도의 파서를 생성한다.

파싱 알고리즘은 HTML 자세히 설명되어 있다. 알고리즘은 토큰화와 트리 구축 이렇게 두 단계로 되어 있다.

토큰화는 어휘 분석으로서 입력 값을 토큰으로 파싱한다. HTML에서 토큰은 시작 태그, 종료 태그, 속성 이름과 속성 값이다.

토큰화는 토큰을 인지해서 트리 생성자로 넘기고 다름 토큰을 확인하기 위해 다음 문자를 확인한다. 그리고 입력의 마지막까지 이 과정을 반복한다.

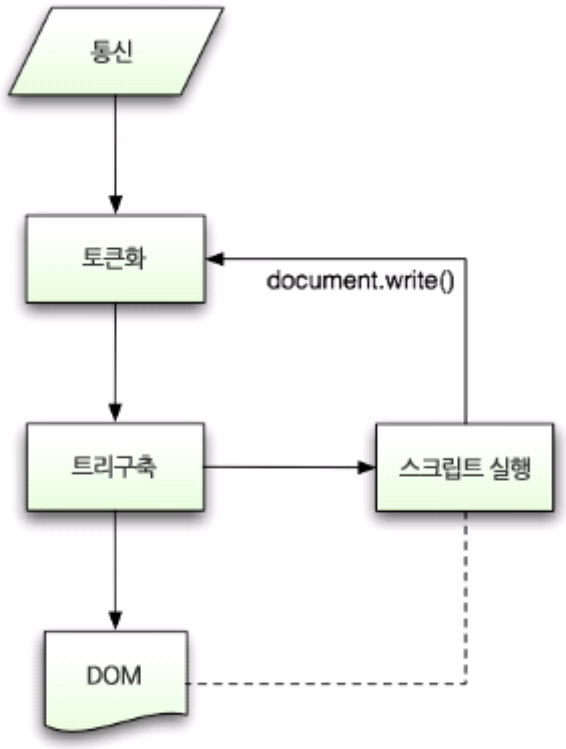


그림 9 HTML 파싱 과정(HTML5 명세에서 가져옴)

토큰화 알고리즘

알고리즘의 결과물은 HTML 토큰이다. 알고리즘은 상태 기계(State Machine)라고 볼 수 있다. 각 상태는 하나 이상의 연속된 문자를 입력받아 이 문자에 따라 다음 상태를 갱신한다. 그러나 결과는 현재의 토큰화 상태와 트리 구축 상태의 영향을 받는데. 이것은 같은 문자를 읽어 들여도 현재 상태에 따라 다음 상태의 결과가 다르게 나온다는 것을 의미한다. 알고리즘은 전체를 설명하기에 너무 복잡하니 원리 이해를 도울만한 간단한 예제를 한번 보자.

다음은 HTML 토큰화를 설명하기 위한 기본적인 예제이다.

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

초기 상태는 "자료 상태" 이다. < 문자를 만나면 상태는 "태그 열림 상태"로 변한다. a 부터 z까지의 문자를 만나면 "시작 태그 토큰"을 생성하고 상태는 "태그 이름 상태"로 변하는데 이 상태는 > 문자를 만날 때까지 유지한다. 각 문자에는 새로운 토큰 이름이 붙는데 이 경우 생성된 토큰은 html 토큰이다.

>문자에 도달하면 현재 토큰이 발행되고 상태는 다시 "자료 상태"로 바뀐다. 태그는 동일한 절차에 따라 처리된다. 지금까지 html 태그와 body 태그를 발행했고 다시 "자료 상태"로 돌아왔다. Hello World의 H 문자를 만나면 문자 토큰이 생성되고 발행될 것이다. 이것은 종료 태그의 < 문자를 만날 때까지 진행된다. Hello World의 각 문자를 위한 문자 토큰을 발행할 것이다.

다시 "태그 열림 상태"가 되었다. / 문자는 종료 태그 토큰을 생성하고 "태그 이름 상태"로 변경 될 것이다. 이 상태는 > 문자를 만날 때까지 유지된다. 그리고 새로운 태그 토큰이 발행되고 다시 "자료 상태"가 된다. 또한 동일하게 처리될 것이다.

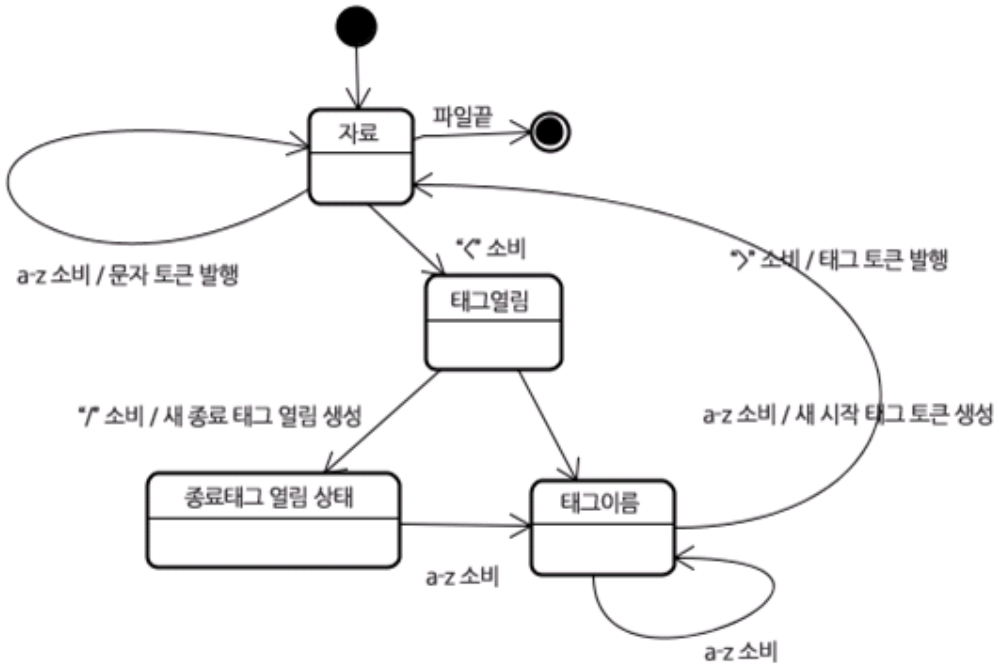


그림 10 입력 예제의 토큰화

트리 구축 알고리즘

파서가 생성되면 문서 객체가 생성된다. 트리 구축이 진행되는 동안 문서 최상단에서는 DOM 트리가 수정되고 요소가 추가된다. 토큰화에 의해 발행된 각 노드는 트리 생성자에 의해 처리된다. 각 토큰을 위한 DOM 요소의 명세는 정의되어 있다. DOM 트리에 요소를 추가하는 것이 아니라면 열린 요소는 스택(임시 버퍼 저장소)에 추가된다. 이 스택은 부정확한 중첩과 종료되지 않은 태그를 교정한다. 알고리즘은 상태 기계라고 설명할 수 있고 상태는 "삽입 모드" 라고 부른다.

아래 입력 예제의 트리 생성 과정을 보자.

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

트리 구축 단계의 입력 값은 토큰화 단계에서 만들어지는 일련의 토큰이다. 받은 html 토큰은 "html 이전" 모드가 되고 토큰은 이 모드에서 처리된다. 이것은 HTMLHtmlElement 요소를 생성하고 문서 객체의 최상단에 추가된다.

상태는 "head 이전" 모드로 바뀌었고 "body" 토큰을 받았다. "head" 토큰이 없더라도 HTMLHeadElement는 묵시적으로 생성되어 트리에 추가될 것이다.

곧이어 "head 안쪽" 모드로 이동했고 다음은 "head 다음" 모드로 간다. body 토큰이 처리 되었고 HTMLBodyElement가 생성되어 추가됐으며 "body 안쪽" 모드가 되었다.

"Hellow world" 문자열의 문자 토큰을 받았다. 첫 번째 토큰이 생성되고 "본문" 노드가 추가되면서 다른 문자들이 그 노드에 추가될 것이다.

body 종료 토큰을 받으면 "body 다음" 모드가 된다. html 종료 태그를 만나면 "body 다음 다음" 모드로 바뀐다. 마지막 파일 토큰을 받으면 파싱을 종료한다.

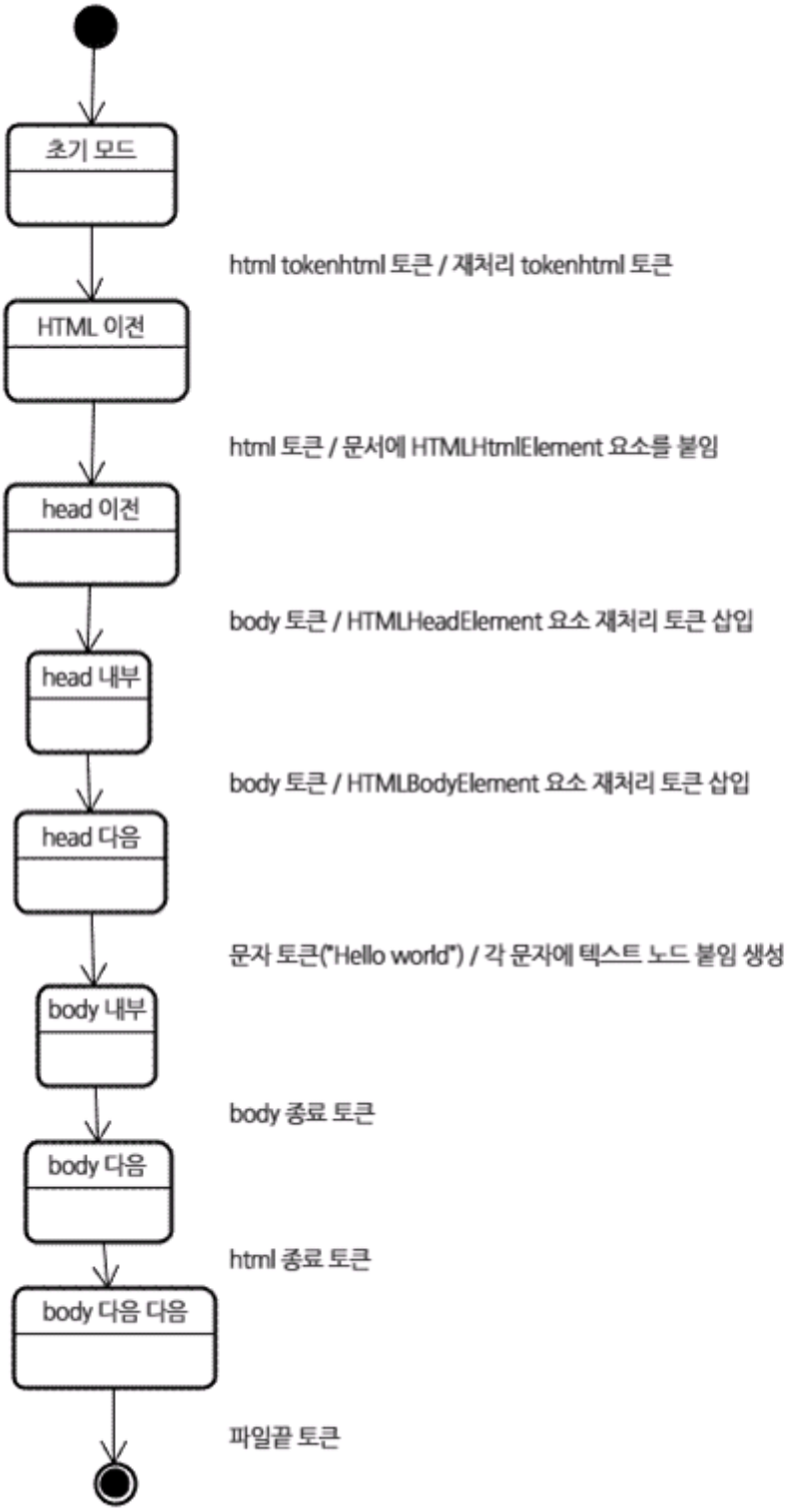


그림 11 예제 html 트리 구축

파싱이 끝난 이후의 동작

이번 단계에서 브라우저는 문서와 상호작용할 수 있게 되고 문서 파싱 이후에 실행되어야 하는 "지연" 모드 스크립트를 파싱하기 시작한다. 문서 상태는 "완료"가 되고 "로드" 이벤트가 발생한다. 보다 자세한 내용은 HTML5의 토큰화 알고리즘과 트리 구축에서 볼 수 있다.

브라우저의 오류 처리

HTML 페이지에서 "유효하지 않은 구문" 이라는 오류를 본 적이 없을 것이다. 이는 브라우저가 모든 오류 구문을 교정하기 때문이다. 아래 오류가 포함된 HTML 예제를 보자.

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

나는 일부러 여러 가지 규칙을 위반 했다. "mytag"는 표준 태그가 아니고 "p" 태그와 "div" 태그는 중첩 오류가 있다. 그러나 브라우저는 투덜거리지 않고 올바르게 표시하는데 이는 파서가 HTML 제작자의 실수를 수정했기 때문이다.

이런 오류 처리 행태는 브라우저에서 꽤나 일반적임에도 불구하고 HTML의 현재 명세가 아니라는 점이 놀라울 뿐이다. 북마크와 이전/다음 버튼처럼 수 년간 브라우저 안에서 구현된 것이다. 잘 알려진 HTML 오류를 많은 사이트에서 발견할 수 있지만 브라우저는 다른 브라우저들이 했던 것처럼 관습적으로 오류를 고치고 있다.

HTML5 명세는 이런 요구 사항 일부를 정의했다. 웹킷은 이것을 HTML 파서 클래스의 시작 부분에 주석으로 잘 요약해 두었다.

파서는 토큰화된 입력 값을 파싱하여 문서를 만들고 문서 트리를 생성한다. 규칙에 맞게 잘 작성된 문서라면 파싱이 수월하겠지만 불행하게도 형식에 맞지 않게 작성된 많은 HTML 문서를 다뤄야 하기 때문에 파서는 오류에 대한 아량이 있어야 한다.

파서는 적어도 다음과 같은 오류를 처리해야 한다.

1. *어떤 태그의 안쪽에 추가하려는 태그가 금지된 것일 때 일단 허용된 태그를 먼저 닫고 금지된 태그는 외부에 추가한다.*
2. *파서가 직접 요소를 추가해서는 안된다. 문서 제작자에 의해 뒤늦게 요소가 추가될 수 있고 생략 가능한 경우도 있다. HTML, HEAD, BODY, TBODY, TR, TD, LI 태그가 이런 경우에 해당한다.*
3. *인라인 요소 안쪽에 블록 요소가 있는 경우 부모 블록 요소를 만날 때까지 모든 인라인 태그를 닫는다.*
4. *이런 방법이 도움이 되지 않으면 태그를 추가하거나 무시할 수 있는 상태가 될 때까지 요소를 닫는다.*

웹킷이 오류를 처리하는 예는 다음과 같다.

대신

어떤 사이트는
 대신
을 사용한다. 인터넷 익스플로러, 파이어폭스와 호환성을 갖기 위해 웹킷은 이것을
 으로 간주한다. 코드는 다음과 같다.

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

오류는 내부적으로 처리하고 사용자에게는 표시하지 않는다.

어긋난 표

어긋난 표는 표 안에 또 다른 표가 th 또는 td 셀 내부에 있지 않은 것을 의미한다. 아래 예제와 같은 경우를 말한다.

```
<table>

    <table>

    <tr><td>inner table</td></tr>

    </table>

    <tr><td>outer table</td></tr>

</table>
```

이런 경우 웹킷은 표의 중첩을 분해하여 형제 요소가 되도록 처리한다.

```
<table>

    <tr><td>outer table</td></tr>

</table>

<table>

    <tr><td>inner table</td></tr>

</table>
```

코드는 다음과 같다.

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

웹킷은 이런 오류를 처리하는데 스택을 사용한다. 안쪽의 표는 바깥쪽 표의 외부로 옮겨져서 형제 요소가 된다.

중첩된 폼 요소

폼 안에 또 다른 폼을 넣은 경우 안쪽의 폼은 무시된다. 코드는 다음과 같다.

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

태그 중첩이 너무 깊을 때

주석에는 이렇게 적혀 있다.

www.liceo.edu.mx 사이트는 약 1,500개 수준의 태그 중첩이 되어 있는 예제인데 모든 요소가 로 되어 있다. 최대 20개의 중첩만 허용하고 나머지는 무시한다.

```
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;
}
```

잘못 닫힌 html 또는 body 태그

주석에는 이렇게 적혀 있다.

깨진 html을 지원한다. 일부 바보 같은 페이지는 문서가 끝나기 전에 body를 닫아버리기 때문에 브라우저는 body 태그를 닫지 않는다. 대신 종료를 위해 end()를 호출한다.

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

웹킷 오류 처리 코드에 예제로 등장하는 것을 원치 않는다면 웹 제작자는 올바른 HTML 작성을 위해 신경 써야 할 것이다.

CSS 파싱

소개 글에서 설명했던 파싱의 개념을 기억하는가? HTML과는 다르게 CSS는 문맥 자유 문법이고 소개 글에서 설명했던 파서 유형을 이용하여 파싱이 가능하다. 실제로 CSS 명세는 CSS 어휘와 문법을 정의하고 있다.

몇 가지 예제를 보자. 어휘 문법은 각 토큰을 위한 정규 표현식으로 정의되어 있다.

omment \/*[^]**+([^/*][^*]**+)*\/*
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"는 클래스 이름처럼 식별자(identifier)를 줄인 것이다. "name"은 요소의 아이디("#"으로 참조하는) 이다.

구문 문법은 BNF로 설명되어 있다.

```
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*] ')' ]
    ;
```

룰셋(ruleset)은 다음과 같은 구조를 나타낸다.

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

div.error와 a.error 는 선택자(selector)이다. 중괄호 안쪽에는 이 룰셋에 적용된 규칙이 포함되어 있다. 이 구조는 공식적으로 다음과 같이 정의되어 있다.

```
Ruleset
    : selector [ ',' S* selector ]*
        '{' S* declaration [ ';' S* declaration ]* '}' S*
    ;
```

룰셋은 쉼표와 공백(S가 공백을 의미함)으로 구분된 하나 또는 여러 개의 선택자라는 것을 의미한다. 룰셋은 중괄호 내부에 하나 또는 세미 콜론으로 구분된 여러 개의 선언을 포함한다. "선언"과 "선택자"는 이어지는 BNF에 정의되어 있다.

웹킷 CSS 파서

웹킷은 CSS 문법 파일로부터 자동으로 파서를 생성하기 위해 플렉스와 바이슨 파서 생성기를 사용한다. 파서 소개에서 언급했던 것처럼 바이슨은 상향식 이동 감소 파서를 생성한다. 파이어폭스는 직접 작성한 하향식 파서를 사용한다. 두 경우 모두 각 CSS 파일은 스타일 시트 객체로 파싱되고 각 객체는 CSS 규칙을 포함한다. CSS 규칙 객체는 선택자와 선언 객체 그리고 CSS 문법과 일치하는 다른 객체를 포함한다.

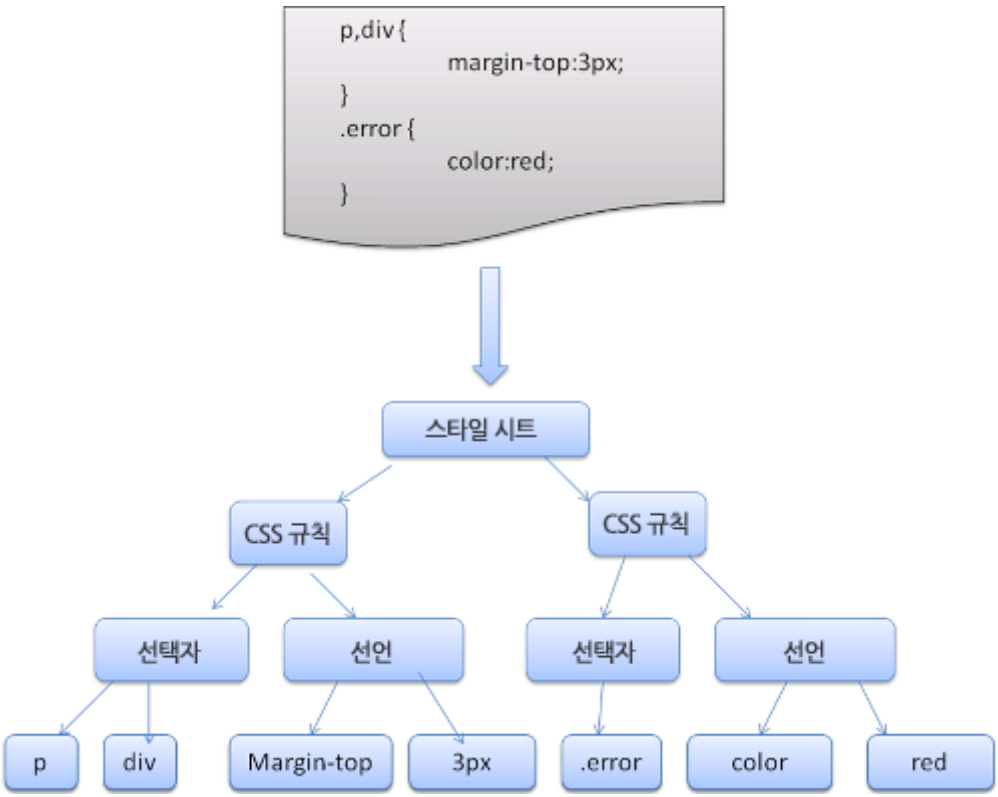


그림 12 CSS 파싱

스크립트와 스타일 시트의 진행 순서

스크립트

웹은 파싱과 실행이 동시에 수행되는 동기화(synchronous) 모델이다. 제작자는 파서가 < script > 태그를 만나면 즉시 파싱하고 실행하기를 기대한다. 스크립트가 실행되는 동안 문서의 파싱은 중단된다. 스크립트가 외부에 있는 경우 우선 네트워크로부터 자원을 가져와야 하는데 이 또한 실시간으로 처리되고 자원을 받을 때까지 파싱은 중단된다. 이 모델은 수 년간 지속됐고 HTML4와 HTML5의 명세에도 정의되어 있다. 제작자는 스크립트를 "지연(defer)"으로 표시할 수 있는데 지연으로 표시하게 되면 문서 파싱은 중단되지 않고 문서 파싱이 완료된 이후에 스크립트가 실행된다. HTML5는 스크립트를 비동기(asynchronous)로 처리하는 속성을 추가했기 때문에 별도의 맥락에 의해 파싱되고 실행된다.

예측 파싱

웹킷과 파이어폭스는 예측 파싱과 같은 최적화를 지원한다. 스크립트를 실행하는 동안 다른 스레드는 네트워크로부터 다른 자원을 찾아 내려받고 문서의 나머지 부분을 파싱한다. 이런 방법은 자원을 병렬로 연결하여 받을 수 있고 전체적인 속도를 개선한다. 참고로 예측 파서는 DOM 트리를 수정하지 않고 메인 파서의 일로 넘긴다. 예측 파서는 외부 스크립트, 외부 스타일 시트와 외부 이미지와 같이 참조된 외부 자원을 파싱할 뿐이다.

스타일 시트

한편 스타일 시트는 다른 모델을 사용한다. 이론적으로 스타일 시트는 DOM 트리를 변경하지 않기 때문에 문서 파싱을 기다리거나 중단할 이유가 없다. 그러나 스크립트가 문서를 파싱하는 동안 스타일 정보를 요청하는 경우라면 문제가 된다. 스타일이 파싱되지 않은 상태라면 스크립트는 잘못된 결과를 내놓기 때문에 많은 문제를 야기한다. 이런 문제는 흔치 않은 것처럼 보이지만 매우 빈번하게 발생한다. 파이어폭스는 아직 로드 중이거나 파싱 중인 스타일 시트가 있는 경우 모든 스크립트의 실행을 중단한다. 한편 웹킷은 로드되지 않은 스타일 시트 가운데 문제가 될만한 속성이 있을 때에만 스크립트를 중단한다.

렌더 트리 구축

DOM 트리가 구축되는 동안 브라우저는 렌더 트리를 구축한다. 표시해야 할 순서와 문서의 시각적인 구성 요소로써 올바른 순서로 내용을 그려낼 수 있도록 하기 위한 목적이 있다.

파이어폭스는 이 구성 요소를 "형상(frames)" 이라고 부르고 웹킷은 "렌더러(renderer)" 또는 "렌더 객체(render object)"라는 용어를 사용한다.

렌더러는 자신과 자식 요소를 어떻게 배치하고 그려내야 하는지 알고 있다.

웹킷 렌더러의 기본 클래스인 RenderObject 클래스는 다음과 같이 정의되어 있다.

```
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
}
```

각 렌더러는 CSS2 명세에 따라 노드의 CSS 박스에 부합하는 사각형을 표시한다. 렌더러는 너비, 높이 그리고 위치와 같은 기하학적 정보를 포함한다.

박스 유형은 노드와 관련된 "display" 스타일 속성의 영향을 받는다(스타일 계산 참고). 여기 보이는 웹킷 코드는 display 속성에 따라 DOM 노드에 어떤 유형의 렌더러를 만들어야 하는지 결정하는 코드이다.

```
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;
        ...
    }
    return o;
}
```

요소 유형 또한 고려해야 하는데 예를 들면 폼 콘트롤과 표는 특별한 구조이다. 요소가 특별한 렌더러를 만들어야 한다면 웹킷은 creatRenderer 메서드를 무시하고 비기하학 정보를 포함하는 스타일 객체를 표시한다.

DOM 트리와 렌더 트리의 관계

렌더러는 DOM 요소에 부합하지만 1:1로 대응하는 관계는 아니다. 예를 들어 "head" 요소와 같은 비시각적 DOM 요소는 렌더 트리에 추가되지 않는다. 또한 display 속성에 "none" 값이 할당된 요소는 트리에 나타나지 않는다(visibility 속성에 "hidden" 값이 할당된 요소는 트리에 나타난다).

여러 개의 시각 객체와 대응하는 DOM 요소도 있는데 이것들은 보통 하나의 사각형으로는 묘사할 수 없는 복잡한 구조다. 예를 들면 "select" 요소는 '표시 영역, 드롭다운 목록, 버튼' 표시를 위한 3개의 렌더러가 있다. 또한 한 줄에 충분히 표시할 수 없는 문자가 여러 줄로 바뀔 때 새 줄은 별도의 렌더러로 추가된다. 여러 렌더러와 대응하는 또 다른 예는 깨진 HTML이다. CSS 명세에 의하면 인라인 박스는 블록 박스만 포함하거나 인라인 박스만을 포함해야 하는데 인라인과 블록 박스가 섞인 경우 인라인 박스를 감싸기 위한 익명의 블록 렌더러가 생성된다.

어떤 렌더 객체는 DOM 노드에 대응하지만 트리의 동일한 위치에 있지 않다. float 처리된 요소 또는 position 속성 값이 absolute로 처리된 요소는 흐름에서 벗어나 트리의 다른 곳에 배치된 상태로 형상이 그려진다. 대신 자리 표시자가 원래 있어야 할 곳에 배치된다.

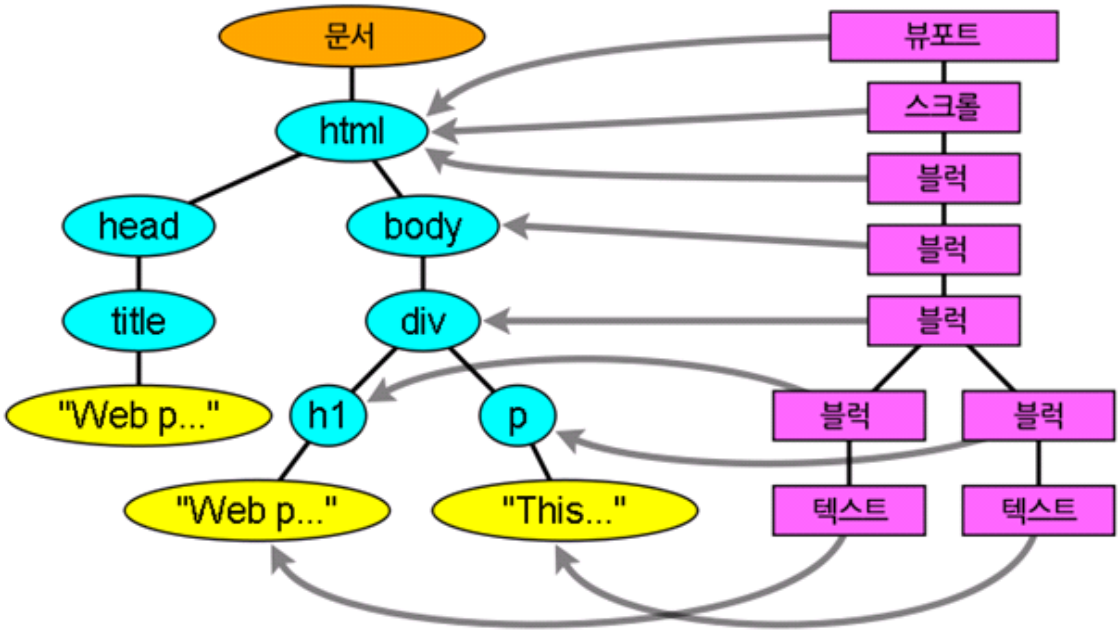


그림 13 렌더 트리와 DOM 트리 대응(3.1). "뷰포트"는 최초의 블록이다. 웹킷에서는 "RenderView" 객체가 이 역할을 한다.

트리를 구축하는 과정

파이어폭스에서 프레젠테이션은 DOM 업데이트를 위한 리스너로 등록된다. 프레젠테이션은 형상 만들기를 FrameConstructor에 위임하고 FrameConstructor는 스타일(스타일 계산 참고)을 결정하고 형상을 만든다.

웹킷에서는 스타일을 결정하고 렌더러를 만드는 과정을 "어태치먼트(attachment)" 라고 부른다. 모든 DOM 노드에는 "attach" 메서드가 있다. 어태치먼트는 동기적인데 DOM 트리에 노드를 추가하면 새 노드의 "attach" 메서드를 호출한다.

html 태그와 body 태그를 처리함으로써 렌더 트리 루트를 구성한다. 루트 렌더 객체는 CSS 명세에서 포함 블록(다른 모든 블록을 포함하는 최상위 블록)이라고 부르는 그것과 일치한다. 파이어폭스는 이것을 ViewPortFrame이라 부르고 웹킷은 RenderView라고 부른다. 이것이 문서가 가리키는 렌더 객체다. 트리의 나머지 부분은 DOM 노드를 추가함으로써 구축된다.

CSS2 처리 모델 명세를 참고.

스타일 계산

렌더 트리를 구축하려면 각 렌더 객체의 시각적 속성에 대한 계산이 필요한데 이것은 각 요소의 스타일 속성을 계산함으로써 처리된다.

스타일은 인라인 스타일 요소와 HTML의 시각적 속성(예를 들면 bgcolor 같은 HTML 속성)과 같은 다양한 형태의 스타일 시트를 포함하는데 HTML의 시각적 속성들은 대응하는 CSS 스타일 속성으로 변환된다.

최초의 스타일 시트는 브라우저가 제공하는 기본 스타일 시트인데 페이지 제작자 또는 사용자도 이를 제공할 수 있다. 브라우저는 사용자가 선호하는 스타일을 정의할 수 있도록 지원하는데 파이어폭스의 경우 "파이어폭스 프로필" 폴더에 있는 스타일 시트를 변경함으로써 사용자 선호 스타일을 정의할 수 있다.

스타일을 계산하는 일에는 다음과 같은 몇 가지 어려움이 따른다.

1. 스타일 데이터는 구성이 매우 광범위한데 수 많은 스타일 속성들을 수용하면서 메모리 문제를 야기할 수 있다.
2. 최적화되어 있지 않다면 각 요소에 할당된 규칙을 찾는 것은 성능 문제를 야기할 수 있다. 각 요소에 할당된 규칙 목록을 전체 규칙으로부터 찾아내는 것은 과중한 일이다. 맞는 규칙을 찾는 과정은 얼핏 보기에는 약속된 방식으로 순탄하게 시작하는 것 같지만 실상 쓸모가 없거나 다른 길을 찾아야만 하는 복잡한 구조가 될 수 있다.
   예를 들어 이런 복합 선택자가 있다.

   div div div div { ... }

   이 선택자는 3번째 자손

   에 규칙을 적용한다는 뜻이다. 규칙을 적용할
   요소를 확인하려면 트리로부터 임의의 줄기를 선택하고 탐색하는 과정에서 규칙에 맞지 않는 줄기를 선택했다면 또 다른 줄기를 선택해야 한다.
3. 규칙을 적용하는 것은 계층 구조를 파악해야 하는 꽤나 복잡한 다단계 규칙을 수반한다.

브라우저가 이 문제를 어떻게 처리하는지 살펴보자.

스타일 정보 공유

웹킷 노드는 스타일 객체(RenderStyle)를 참조하는데 이 객체는 일정 조건 아래 공유할 수 있다. 노드가 형제이거나 또는 사촌일 때 공유하며 다음과 같은 조건일 때 공유할 수 있다.

1. 동일한 마우스 반응 상태를 가진 요소여야 한다. 예를 들어 한 요소가 :hover 상태가 될 수 없는데 다른 요소는 :hover가 될 수 있다면 동일한 마우스 상태가 아니다.
2. 아이디가 없는 요소.
3. 태그 이름이 일치해야 한다.
4. 클래스 속성이 일치해야 한다.
5. 지정된 속성이 일치해야 한다.
6. 링크(link) 상태가 일치해야 한다.
7. 초점(focus) 상태가 일치해야 한다.
8. 문서 전체에서 속성 선택자의 영향을 받는 요소가 없어야 한다. 여기서 영향이라 함은 속성 선택자를 사용한 경우를 말한다(속성 선택자 예 input[type=text]{...})
9. 요소에 인라인 스타일 속성이 없어야 한다(인라인 스타일 예

   ...

).
10. 문서 전체에서 형제 선택자를 사용하지 않아야 한다. 웹 코어는 형제 선택자를 만나면 전역 스위치를 열고 전체 문서의 스타일 공유를 중단한다. 형제 선택자는 + 선택자와 :first-child 그리고 :last-child를 포함한다.

파이어폭스 규칙 트리

파이어폭스는 스타일 계산을 쉽게 처리하기 위해 규칙 트리와 스타일 문맥 트리라고 하는 두 개의 트리를 더 가지고 있다. 웹킷도 스타일 객체를 가지고 있지만 스타일 문맥 트리처럼 저장되지 않고 오직 DOM 노드로 관련 스타일을 처리한다.

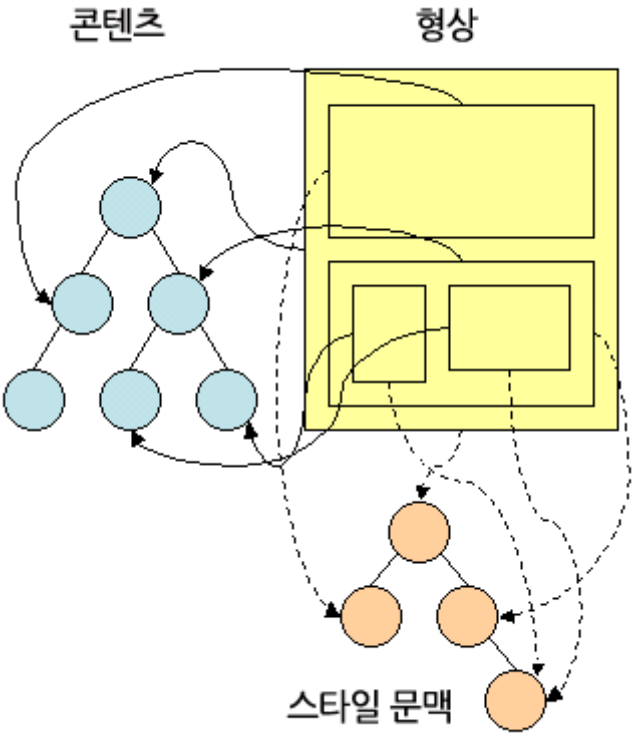
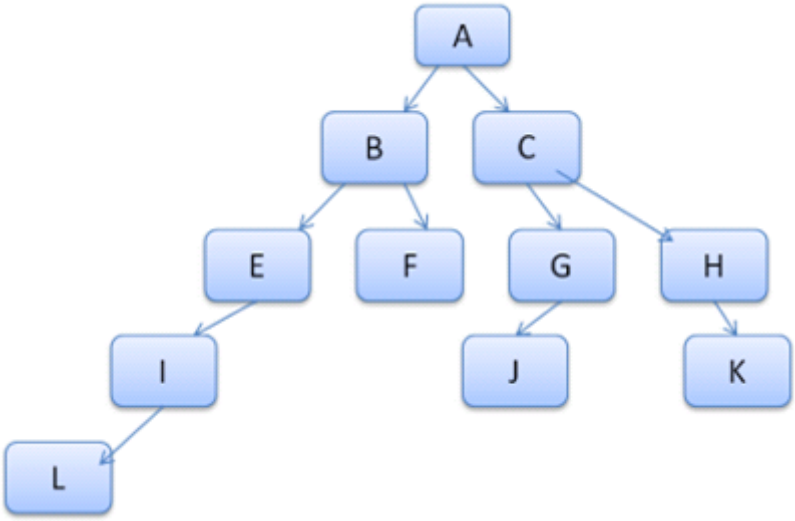


그림 14 파이어폭스 스타일 문맥 트리(2.2)

스타일 문맥에는 최종 값이 저장되어 있다. 값은 올바른 순서 안에서 부합하는 규칙을 적용하고 논리로부터 구체적인 값으로 변환함으로써 계산된다. 예를 들어 논리적인 값이 화면의 백분율(%) 이라면 이 값은 계산에 의해 절대적인 단위(px)로 변환된다. 이런 규칙 트리 아이디어는 정말 현명하다. 노드 사이에서 이 값을 공유함으로써 그것들을 다시 계산하는 일을 방지하기 때문이다.

부합하는 모든 규칙은 트리에 저장하는데 경로의 하위 노드가 높은 우선순위를 갖는다. 규칙 저장은 느리게 처리된다. 트리는 처음부터 모든 노드를 계산하지 않지만 노드 스타일이 계산될 필요가 있을 때 계산된 경로를 트리에 추가한다.

트리 경로를 어휘 목록 속에 있는 단어라고 생각하고 이미 규칙 트리를 계산했다고 가정해 보자.



내용 트리에서 또 다른 요소에 부합하는 규칙이 필요하다고 가정하고 부합하는 규칙이 순서에 따라 B - E - I 라고 치자. 브라우저는 이미 A - B - E - I - L 경로를 계산했기 때문에 트리 안에 이 경로가 있고 할 일이 줄었다.

트리가 작업량을 줄이는 방법을 살펴보자.

구조체로 분리

스타일 문맥은 구조체(structs)로 나뉘는데 선 또는 색상과 같은 종류의 스타일 정보를 포함한다. 구조체의 속성들은 상속되거나 또는 상속되지 않는다. 속성들은 요소에 따라 정해져 있지 않은 한 부모로부터 상속된다. 상속되지 않는 속성들은 "재설정(reset)" 속성이라 부르는데 상속을 받지 않는 것으로 정해져 있다면 기본 값을 사용한다.

트리는 최종으로 계산된 값을 포함하여 전체 구조체를 저장하는 방법으로 도움을 준다. 하위 노드에 구조체를 위한 속성 선언이 없다면 저장된 상위 노드의 구조체 속성을 그대로 받아서 사용하는 것이다.

규칙 트리를 사용하여 스타일 문맥을 계산

어떤 요소의 스타일 문맥을 계산할 때 가장 먼저 규칙 트리의 경로를 계산하거나 또는 이미 존재하는 경로를 사용한다. 그 다음 새로운 스타일 문맥으로 채우기 위해 경로 안에서 규칙을 적용한다. 가장 높은 우선순위(보통 가장 구체적인 선택자)를 가진 경로의 하위 노드에서 시작하여 구조체가 가득 찰 때까지 트리의 상단으로 거슬러 올라간다. 규칙 노드 안에서 구조체를 위한 특별한 선언이 없다면 상당한 최적화를 할 수 있다. 선언이 가득 채워질 때까지 노드 트리의 상위로 찾아 올라가서 간단하게 적용하면 최상의 최적화가 되고 모든 구조체는 공유된다. 이것은 최종 값과 메모리 계산을 절약한다.

선언이 완전하지 않으면 구조체가 채워질 때까지 트리의 상단으로 거슬러 올라간다.

구조체에서 어떤 선언도 발견할 수 없는 경우 구조체는 "상속(inherit)" 타입인데 문맥 트리에서 부모 구조체를 향하면서 성공적으로 구조체를 공유한다. 재설정 구조체라면 기본 값들이 사용될 것이다.

가장 구체적인 노드에 값을 추가하면 실제 값으로 변환하기 위해 약간의 추가적인 계산을 할 필요가 있는데 트리 노드에서 결과를 저장하기 때문에 자식에게도 사용할 수 있다.

같은 트리 노드를 가리키는 형제 요소가 있는 경우 전체 스타일 문맥이 이들 사이에서 공유된다.

이런 HTML이 있다고 가정해 보자.

```
<div class="err" id="div1">
    <p>
    this is a <span class="big"> big error </span>
    this is also a <span class="big"> very big error</span> error
    </p>
</div>
<div class="err" id="div2">another error</div>
```

그리고 다음과 같은 규칙이 있다.

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

좀 단순하게 하기 위해 색상과 여백 이렇게 두 개의 구조체를 채울 필요가 있다고 치자. 색상 구조체는 오직 색상 값만을 포함하고 여백 구조체는 네 개의 면에 대한 값을 포함한다.

결과적으로 규칙 트리는 아래처럼 보일 것이다. 노드는 노드 이름과 노드가 가리키는 규칙의 번호로 표시되어 있다.

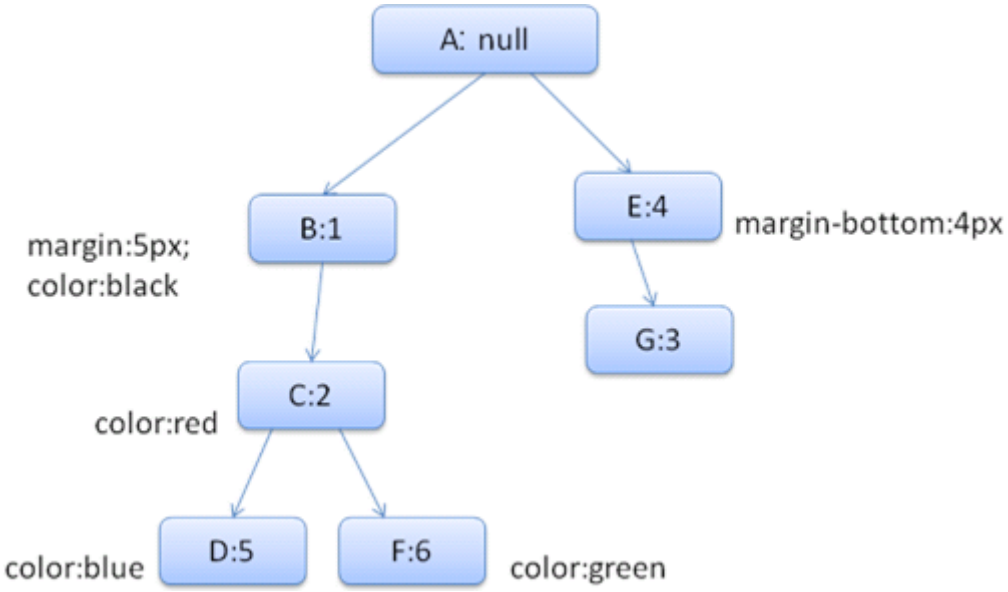


그림 15 규칙 트리

문맥 트리는 아래처럼 보일 것이다. 노드는 노드 이름과 노드가 가리키는 규칙 노드로 표시되어 있다.

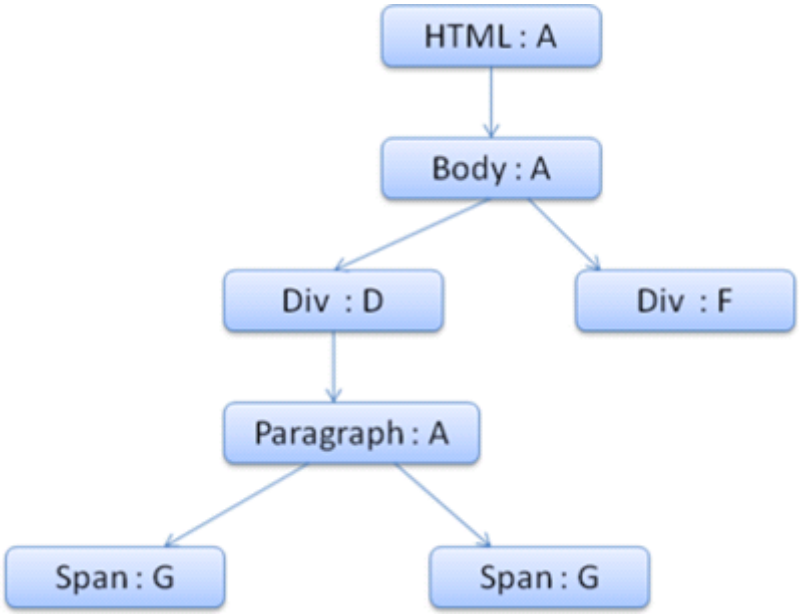


그림 16 문맥 트리

HTML을 파싱하여 두 번째

태그인
에 이르렀다고 가정하자. 이 노드에 필요한 스타일 문맥을 생성하고 스타일 구조체를 채워야 한다.

두 번째

규칙에 맞는 것을 찾으면 1, 2, 6이 되는데 이것은 요소가 사용할 수 있는 트리 경로(규칙 트리의 B:1, C:2)가 이미 존재한다는 것을 의미하고 규칙 6(규칙 트리에서 노드 F:6)에 이르는 또 다른 노드를 문맥 트리에 추가하면 된다. 스타일 문맥을 생성하고 문맥 트리에 추가하면 새로운 스타일 문맥이 규칙 트리의 F:6 노드를 가리킨다.

이제는 스타일 구조체를 채워야 하는데 여백 구조체를 채우는 것으로부터 시작한다. 마지막 규칙 노드(F:6)가 여백 구조체를 포함하지 않기 때문에 이전 노드에 저장된 구조체를 찾을 때까지 위로 거슬러 올라가서 계산된 값을 사용한다. 여백 규칙이 선언된 최상위 노드의 구조체를 규칙 노드 B:1 에서 찾았다.

색상 구조체 정의에는 저장된 구조체를 사용할 수 없다. 색상은 이미 하나의 속성 값을 가지고 있기 때문에 다른 값을 채우기 위해 규칙 트리 상단으로 거슬러 올라갈 필요가 없다. 최종 값을 계산하고 계산된 값(문자열에서 RGB 등으로 변환된)을 이 노드에 저장할 것이다.

두 번째 요소는 보다 수월하게 진행된다. 맞는 규칙을 찾다 보면 이전 span과 같이 규칙 트리의 G:3를 가리킨다는 결론에 이르는데 동일한 노드를 가리키는 형제가 있기 때문에 전체 스타일 문맥을 공유하고 이전 span의 문맥을 취하면 된다.

부모로부터 상속된 규칙을 포함하고 있는 구조체의 저장은 문맥 트리에서 처리된다. 색상 속성은 실제로 상속된다. 그러나 파이어폭스는 재설정으로 처리해서 규칙 트리에 저장한다.

예를 들어 문단 요소에 글꼴을 위한 규칙을 추가한다면.

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

문맥 트리에서 div의 자식인 p 요소는 그 부모의 동일 글꼴 구조체를 공유할 수 있다. p 요소에 지정된 규칙이 없는 경우라도 마찬가지다.

규칙 트리가 없는 웹킷은 선언이 일치하는 규칙이 4번 탐색된다. 우선 중요하지 않은 상위 속성(display와 같은 속성이 의존하기 때문)이 적용되고, 그 다음 중요한 상위 속성이 적용된다. 그리고 나서 중요하지 않은 일반 속성이 적용되고 마지막으로 중요한 일반 속성이 적용된다. 이것은 여러 번 나타나는 속성들이 정확한 다단계 순서에 따라 결정된다는 것을 의미하고 가장 마지막 값이 적용된다.

요약하면 스타일 객체는 전체 또는 일부를 공유함으로써 1번과 3번 문제를 해결한다. 파이어폭스 규칙 트리는 올바른 순서에 따라 속성을 적용하는 것을 돕는다.

쉬운 선택을 위한 규칙 다루기

스타일 규칙을 위한 몇 가지 소스가 있다.

CSS 규칙을 외부 스타일 시트에서 선언하거나 style 요소에서 선언

```
p {color:blue}
```

인라인 스타일 속성

```
<p style="color:blue"></p>
```

HTML의 시각적 속성(이것들은 CSS 규칙으로 변환됨)

```
<p bgcolor="blue"></p>
```

마지막 두 가지 스타일은 자신이 스타일 속성을 가지고 있거나 HTML 속성을 이용하여 연결할 수 있기 때문에 요소에 쉽게 연결된다.

위에서 언급한 문제 2번에 따라 CSS 규칙을 연결하는 것은 까다로울 수 있는데 이 문제를 해결하려면 쉽게 접근할 수 있도록 규칙을 교묘하게 처리해야 한다.

스타일 시트를 파싱한 후 규칙은 선택자에 따라 여러 해시맵 중 하나에 추가된다. 아이디, 클래스 이름, 태그 이름을 사용한 맵이 있고 이런 분류에 맞지 않는 것을 위한 일반적인 맵이 있다. 선택자가 아이디인 경우 규칙은 아이디 맵에 추가되고 선택자가 클래스인 경우 규칙은 클래스 맵에 추가된다.

이런 처리 작업을 통해 규칙을 찾는 일은 훨씬 쉬워진다. 맵에서 특정 요소와 관련 있는 규칙을 추출할 수 있기 때문에 모든 선언을 찾아 볼 필요가 없다. 이러한 최적화는 찾아야 할 규칙의 95% 이상을 제거하기 때문에 규칙을 찾는 동안 모든 선언을 고려할 필요가 없다.

다음 스타일 규칙 예제를 살펴보자.

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

첫 번째 규칙은 클래스 맵에 추가된다. 두 번째는 아이디 맵에 추가되고 세 번째는 태그 맵에 추가된다.

위 스타일과 관련된 HTML 코드는 다음과 같다.

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

우선 p 요소의 규칙을 찾아보자. 클래스 맵은 발견된 "p.error"를 위한 규칙 하부의 "error" 키를 찾았다. div 요소는 아이디 맵(키는 아이디)과 태그 맵에 관련 규칙이 있다. 그러므로 이제 남은 작업은 키를 사용하여 추출한 규칙 중에 실제로 일치하는 규칙을 찾는 것이다.

예를 들어 div에 해당하는 다음과 같은 또 다른 규칙이 있다고 가정하자.

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

이 예제는 여전히 태그 맵에서 규칙을 추출할 것이다. 가장 우측에 있는 선택자가 키이기 때문이다. 그러나 앞서 작성한 div 요소와는 일치하지 않는다. 상위에 table이 없기 때문이다.

웹킷과 파이어폭스 모두 이런 방식으로 처리하고 있다.

다단계 순서에 따라 규칙 적용하기

스타일 객체는 모든 CSS 속성을 포함하고 있는데 어떤 규칙과도 일치하지 않는 일부 속성은 부모 요소의 스타일 객체로부터 상속 받는다. 그 외 다른 속성들은 기본 값으로 설정된다.

문제는 하나 이상의 속성이 정의될 때 시작되고 다단계 순서가 이 문제를 해결하게 된다.

스타일 시트 다단계 순서

스타일 속성 선언은 여러 스타일 시트에서 나타날 수 있고 하나의 스타일 시트 안에서도 여러 번 나타날 수 있는데 이것은 규칙을 적용하는 순서가 매우 중요하다는 것을 의미한다. 이것을 "다단계(cascade)" 순서라고 한다. CSS2 명세에 따르면 다단계 순서는 다음과 같다(우선 순위가 낮은 것에서 높은 순서임).

1. 브라우저 선언 (browser declarations)
2. 사용자 일반 선언 (user normal declarations)
3. 저작자 일반 선언 (author normal declarations)
4. 저작자 중요 선언 (author important declarations)
5. 사용자 중요 선언 (user important declarations)

브라우저 선언의 중요도가 가장 낮으며 사용자가 저작자의 선언을 덮어 쓸 수 있는 것은 선언이 중요하다고 표시한 경우뿐이다. 같은 순서 안에서 동일한 속성 선언은 특정성(specificity)에 의해 정렬이 되고 이 순서는 곧 특정성이 된다. HTML 시각 속성은 CSS 속성 선언으로 변환되고 변환된 속성들은 저작자 일반 선언 규칙으로 간주된다.

특정성

선택자 특정성은 CSS2 명세에 다음과 같이 정의되어 있다.

- 선택자 없이 'style' 속성이 선언된 것이면 1을 센다. 그렇지 않으면 0을 센다. (=a)
- 선택자에 포함된 아이디 선택자 개수를 센다. (=b)
- 선택자에 포함된 속성 선택자(클래스 선택자와 속성 선택자)와 가상 클래스 선택자의 숫자를 센다. (=c)
- 선택자에 포함된 요소 선택자와 가상 요소 선택자의 숫자를 센다. (=d)

네 개의 연결된 숫자 a-b-c-d (큰 진법의 숫자)를 연결하면 특정성의 값이 된다.

사용할 진법은 분류 중에 가장 높은 숫자에 의해서 정의된다. 예를 들어 a=14이면 16진수를 사용할 수 있다. 흔치는 않겠지만 a=17과 같은 경우라면 17진법이 필요할 것이다. 17진법을 사용해야 하는 경우는 html body div div p ... (선택자에 17개의 태그를 사용하는 경우로 흔치 않음)와 같이 선택자를 사용하는 경우에 발생할 수 있다.

다음과 같은 몇 가지 예제를 참고하기 바란다.

```
*{} /* 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 */
```

규칙 정렬

맞는 규칙을 찾으면 다단계 규칙에 따라 정렬된다. 웹킷은 목록이 적으면 버블 정렬을 사용하고 목록이 많을 때는 병합 정렬을 사용한다. 웹킷은 규칙에 ">" 연산자를 덮어쓰는 방식으로 정렬을 실행한다.

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

점진적 처리

웹킷은 @import를 포함한 최상위 수준의 스타일 시트가 로드되었는지 표시하기 위해 플래그를 사용한다. DOM 노드와 시각정보를 연결하는 과정(attaching)에서 스타일이 완전히 로드되지 않았다면 문서에 자리 표시자를 사용하고 스타일 시트가 로드됐을 때 다시 계산한다.

배치

렌더러가 생성되어 트리에 추가될 때 크기와 위치 정보는 없는데 이런 값을 계산하는 것을 배치 또는 리플로라고 부른다.

HTML은 흐름 기반의 배치 모델을 사용하는데 이것은 보통 단일 경로를 통해 크기와 위치 정보를 계산할 수 있다는 것을 의미한다. 일반적으로 "흐름 속"에서 나중에 등장하는 요소는 앞서 등장한 요소의 위치와 크기에 영향을 미치지 않기 때문에 배치는 왼쪽에서 오른쪽으로 또는 위에서 아래로 흐른다. 단, 표는 크기와 위치를 계산하기 위해 하나 이상의 경로를 필요로 하기 때문에 예외가 된다 (3.5).

좌표계는 기준점으로부터 상대적으로 위치를 결정하는데 좌단(X축)과 상단(Y축) 좌표를 사용한다.

배치는 반복되며 HTML 문서의 요소에 해당하는 최상위 렌더러에서 시작한다. 배치는 프레임 계층의 일부 또는 전부를 통해 반복되고 각 렌더러에 필요한 크기와 위치 정보를 계산한다.

최상위 렌더러의 위치는 0,0 이고 브라우저 창의 보이는 영역에 해당하는 뷰포트 만큼의 면적을 갖는다.

모든 렌더러는 "배치" 또는 "리플로" 메서드를 갖는데 각 렌더러는 배치해야 할 자식의 배치 메소드를 불러온다.

더티 비트 체제

소소한 변경 때문에 전체를 다시 배치하지 않기 위해 브라우저는 "더티 비트" 체제를 사용한다. 렌더러는 다시 배치할 필요가 있는 변경 요소 또는 추가된 것과 그 자식을 "더티"라고 표시한다.

"더티"와 "자식이 더티" 이렇게 두 가지 플래그가 있다. 자식이 더터하다는 것은 본인은 괜찮지만 자식 가운데 적어도 하나를 다시 배치할 필요가 있다는 의미다.

전역 배치와 점증 배치

배치는 렌더러 트리 전체에서 일어날 수 있는데 이것을 "전역" 배치라 하고 다음과 같은 경우에 발생한다.

1. 글꼴 크기 변경과 같이 모든 렌더러에 영향을 주는 전역 스타일 변경.
2. 화면 크기 변경에 의한 결과.

배치는 더티 렌더러가 배치되는 경우에만 점증되는데 추가적인 배치가 필요하기 때문에 약간의 손실이 발생할 수 있다.

점증 배치는 렌더러가 더티일 때 비동기적으로 일어난다. 예를 들면 네트워크로부터 추가 내용을 받아서 DOM 트리에 더해진 다음 새로운 렌더러가 렌더 트리에 붙을 때이다.

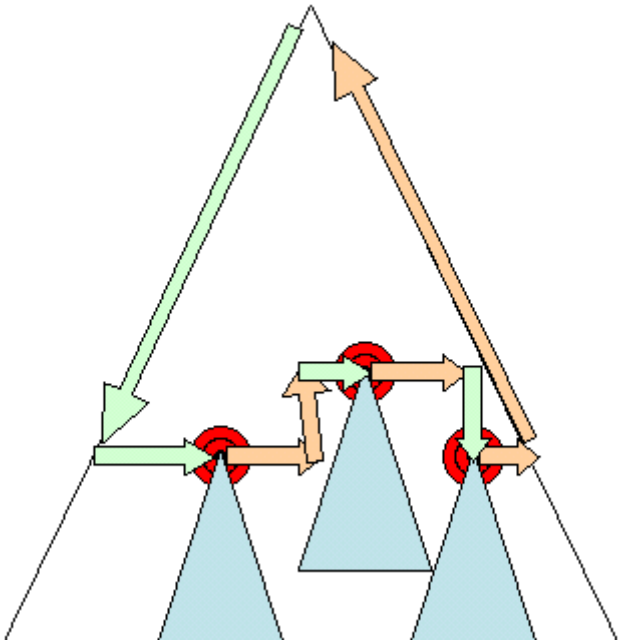


그림 17 점증 배치 - 오직 더티 렌더러와 그 자식만 배치된다(3.6).

비동기 배치와 동기 배치

점증 배치는 비동기로 실행된다. 파이어폭스는 점증 배치를 위해 "리플로 명령"을 쌓아 놓고 스케줄러는 이 명령을 한꺼번에 실행한다. 웹킷도 점증 배치를 실행하는 타이머가 있는데 트리를 탐색하여 "더티" 렌더러를 배치한다.

"offsetHeight" 같은 스타일 정보를 요청하는 스크립트는 동기적으로 점증 배치를 실행한다.

전역 배치는 보통 동기적으로 실행된다.

때때로 배치는 스크롤 위치 변화와 같은 일부 속성들 때문에 초기 배치 이후 콜백으로 실행된다.

최적화

배치가 "크기 변경" 또는 렌더러 위치 변화 때문에 실행되는 경우 렌더러의 크기는 다시 계산하지 않고 캐시로부터 가져온다.

어떤 경우는 하위 트리만 수정이 되고 최상위로부터 배치가 시작되지 않는 경우도 있다. 이런 경우는 입력 필드에 텍스트를 입력하는 경우와 같이 변화 범위가 한정적이어서 주변에 영향을 미치지 않을 때 발생한다. 만약 입력 필드 바깥쪽에 텍스트가 입력되는 경우라면 배치는 최상단으로부터 시작될 것이다.

배치 과정

배치는 보통 다음과 같은 형태로 진행된다.

1. 부모 렌더러가 자신의 너비를 결정.
2. 부모가 자식을 검토.
    1. 자식 렌더러를 배치(자식의 x와 y를 설정)
    2. (부모와 자식이 더티하거나 전역 배치 상태이거나 또는 다른 이유로) 필요하다면 자식 배치를 호출하여 자식의 높이를 계산한다.
3. 부모는 자식의 누적된 높이와 여백, 패딩을 사용하여 자신의 높이를 설정한다. 이 값은 부모 렌더러의 부모가 사용하게 된다.
4. 더티 비트 플래그를 제거한다.

파이어폭스는 "상태" 객체(nsHTMLReflowState)를 배치("리플로"를 의미)를 위한 매개 변수로 사용하는데 상태는 부모의 너비를 포함한다.

파이어폭스 배치의 결과는 "매트릭스" 객체(nsHTMLReflowMatrics)인데 높이가 계산된 렌더러를 포함한다.

너비 계산

렌더러의 너비는 포함하는 블록의 너비, 그리고 렌더러의 너비와 여백, 테두리를 이용하여 계산된다.

예를 들어 다음은 div 요소의 너비를 보자.

```
<div style="width:30%"></div>
```

웹킷은 다음(RenderBox 클래스의 calcWidth 메서드)과 같이 계산할 것이다.

- 컨테이너의 너비는 컨테이너 availableWidth와 0 사이의 최대값이다. 이 경우 availableWidth는 다음과 같이 계산된 contentWidth이다.

  clientWidth() - paddingLeft() - paddingRight()

  clientWidth와 clientHeight는 객체의 테두리와 스크롤바를 제외한 내부 영역을 의미한다.
- 요소의 너비는 "width" 스타일 속성의 값이다. 이 컨테이너 너비의 백분률 값은 절대 값으로 변환될 것이다.
- 좌우측 테두리와 패딩 값이 추가된다.

여기까지 "미리 획득한 너비"의 계산이었다. 이제는 최소 너비와 최대 너비를 계산해야 한다.

미리 획득한 너비가 최대 너비보다 크면 최대 너비가 사용된다. 미리 획득한 너비가 최소 너비(깨지지 않는 가장 작은 단위)보다 작으면 최소 너비가 사용된다.

배치할 필요가 있지만 너비가 고정된 경우 값은 캐시에 저장된다.

줄 바꿈

렌더러가 배치되는 동안 줄을 바꿀 필요가 있을 때 배치는 중단되고 줄 바꿀 필요가 있음을 부모에게 전달한다. 부모는 추가 렌더러를 생성하고 배치를 호출한다.

그리기

그리기 단계에서는 화면에 내용을 표시하기 위한 렌더 트리가 탐색되고 렌더러의 "paint" 메서드가 호출된다. 그리기는 UI 기반의 구성 요소를 사용한다.

전역과 점증

그리기는 배치와 마찬가지로 전역 또는 점증 방식으로 수행된다. 점증 그리기에서 일부 렌더러는 전체 트리에 영향을 주지 않는 방식으로 변경된다. 변경된 렌더러는 화면 위의 사각형을 무효화 하는데 OS는 이것을 "더티 영역"으로 보고 "paint" 이벤트를 발생시킨다. OS는 몇 개의 영역을 하나로 합치는 방법으로 효과적으로 처리한다. 크롬은 렌더러가 별도의 처리 과정이기 때문에 조금 더 복잡하다. 크롬은 OS의 동작을 어느 정도 모방한다. 프레젠테이션은 이런 이벤트에 귀 기울기고 렌더 최상위로 메시지를 전달한다. 그러면 트리는 적절한 렌더러에 이를 때까지 탐색되고 스스로(보통 자식과 함께) 다시 그려진다.

그리기 순서

CSS 2는 그리기 과정의 순서를정의했다. 이것은 실제로 요소가 stacking contexts에 쌓이는 순서다. 스택은 뒤에서 앞으로 그려지기 때문에 이 순서는 그리기에 영향을 미친다. 블록 렌더러가 쌓이는 순서는 다음과 같다.

1. 배경 색
2. 배경 이미지
3. 테두리
4. 자식
5. 아웃라인

파이어폭스 표시 목록

파이어폭스는 렌더 트리를 검토하고 그려진 사각형을 위한 표시 목록을 구성한다. 목록은 올바른 그리기 순서(배경, 테두리, 기타.....)에 따라 사각형을 위한 적절한 렌더러를 포함한다. 이런 방법으로 트리는 여러 번 리페인팅을 실행하는 대신 한 번만 탐색하면서 배경 색, 배경 이미지, 테두리 그리고 나머지 순으로 그려낸다.

파이어폭스는 다른 불투명 요소 뒤에 완전히 가려진 요소는 추가하지 않는 방법으로 최적화를 진행한다.

웹킷 사각형 저장소

리페인팅 전에 웹킷은 기존의 사각형을 비트맵으로 저장하여 새로운 사각형과 비교하고 차이가 있는 부분만 다시 그린다.

동적 변경

브라우저는 변경에 대해 가능한 한 최소한의 동작으로 반응하려고 노력한다. 그렇기 때문에 요소의 색깔이 바뀌면 해당 요소의 리페인팅만 발생한다. 요소의 위치가 바뀌면 요소와 자식 그리고 형제의 리페인팅과 재배치가 발생한다. DOM 노드를 추가하면 노드의 리페인팅과 재 배치가 발생한다. "html" 요소의 글꼴 크기를 변경하는 것과 같은 큰 변경은 캐시를 무효화하고 트리 전체의 배치와 리페인팅이 발생한다.

렌더링 엔진의 스레드

렌더링 엔진은 통신을 제외한 거의 모든 경우에 단일 스레드로 동작한다. 파이어폭스와 사파리의 경우 렌더링 엔진의 스레드는 브라우저의 주요한 스레드에 해당한다. 크롬에서는 이것이 탭 프로세스의 주요 스레드이다.

통신은 몇 개의 병렬 스레드에 의해 진행될 수 있는데 병렬 연결의 수는 보통 2개에서 6개로 제한된다 (예를 들면 파이어폭스 3은 6개를 사용).

이벤트 순환

브라우저의 주요 스레드는 이벤트 순환으로 처리 과정을 유지하기 위해 무한 순환된다. 배치와 그리기 같은 이벤트를 위해 대기하고 이벤트를 처리한다. 아래는 주요 이벤트 순환을 위한 파이어폭스 코드이다.

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

CSS2 시각 모델

캔버스

CSS2 명세는 캔버스를 "서식 구조가 표현되는 공간" 이라고 설명한다. 브라우저가 내용을 그리는 공간인 것이다. 캔버스 공간 각각의 면적은 무한하지만 브라우저는 뷰포트의 크기를 기초로 초기 너비를 결정한다.

CSS2 명세에 따르면 캔버스는 기본적으로 투명하기 때문에 다른 캔버스와 겹치는 경우 비쳐 보이고, 투명하지 않을 경우에는 브라우저에서 정의한 색이 지정된다.

CSS 박스 모델

CSS 박스 모델은 문서 트리에 있는 요소를 위해 생성되고 시각적 서식 모델에 따라 배치된 사각형 박스를 설명한다.

각 박스는 콘텐츠 영역(문자, 이미지 등)과 선택적인 패딩과 테두리, 여백이 있다.

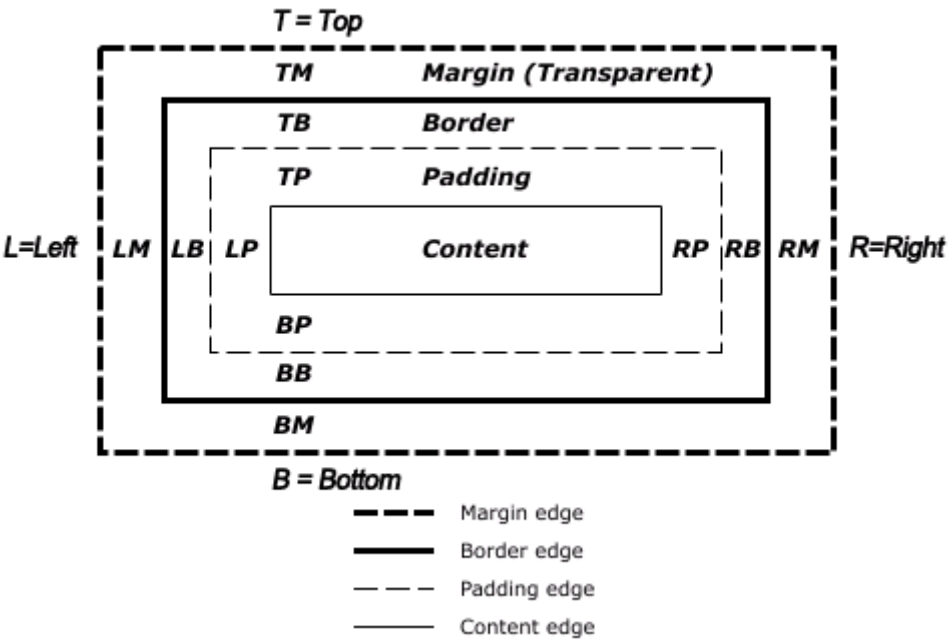


그림 18 CSS2 박스 모델

각 노드는 이런 상자를 0에서 n개 생성한다.

모든 요소는 만들어질 박스의 유형을 결정하는 "display" 속성을 갖는데 이 속성의 유형은 다음과 같다.

- block - 블록 상자를 만든다.

- inline - 하나 또는 그 이상의 인라인 상자를 만든다.
- none - 박스를 만들지 않는다.

기본 값은 인라인이지만 브라우저의 스타일 시트는 다른 기본 값을 설정한다. 예를 들면 "div" 요소의 display 속성에 대한 기본 값은 block 이다.

브라우저의 기본 스타일 시트 예제는 [www][45][.][45][w][45][3.][45][org][45][/][45][TR][45][/][45][CSS][45][2/][45][sample][45][.][45][html][45]에서 찾을 수 있다.

위치 결정 방법

위치를 결정하는 방법은 다음과 같은 세 가지다.

1. Normal - 객체는 문서 안의 자리에 따라 위치가 결정된다. 이것은 렌더 트리에서 객체의 자리가 DOM 트리의 자리와 같고 박스 유형과 면적에 따라 배치됨을 의미한다.
2. Float - 객체는 우선 일반적인 흐름에 따라 배치된 다음 왼쪽이나 오른쪽으로 흘러 이동한다.
3. Absolute - 객체는 DOM 트리 자리와는 다른 렌더 트리에 놓인다.

위치는 "position" 속성과 "float" 속성에 의해 결정된다.

- static과 relative로 설정하면 일반적인 흐름에 따라 위치가 결정된다.
- absolute와 fixed로 설정하면 절대적인 위치가 된다.

position 속성을 정의하지 않으면 static이 기본 값이 되며 일반적인 흐름에 따라 위치가 결정된다.
static 아닌 다른 속성 값(relatice, absolute, fixed)을 사용하면 top, bottom, left, right 속성으로 위치를 결정할 수 있다.

박스가 배치되는 방법은 다음과 같은 방법으로 결정된다.

- 박스 유형(display, inline ...)
- 박스 크기(width, height ...)
- 위치 결정 방법(position, float)
- 추가적인 정보 - 이미지 크기와 화면 크기 등

박스 유형

블록 박스: 브라우저 창에서 사각형 블록을 형성한다.

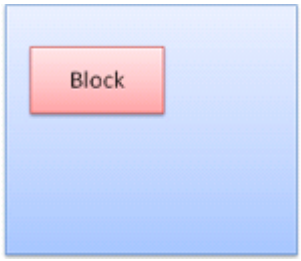


그림 19 블록 박스

인라인 박스: 블록이 되지 않고 블록 내부에 포함된다.

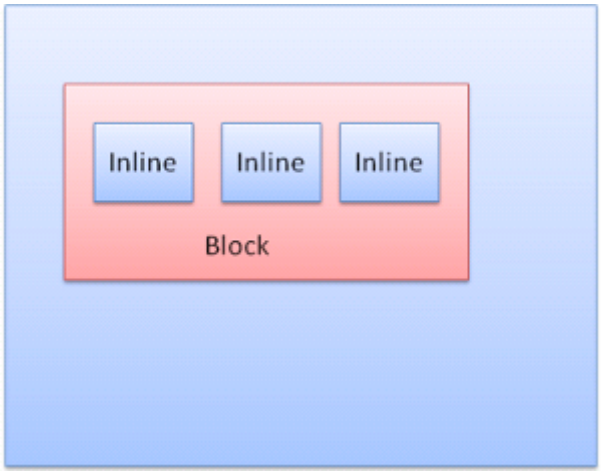


그림 20 인라인 박스

블록은 다른 블록 아래 수직으로 배치되고 인라인은 수평으로 배치된다.

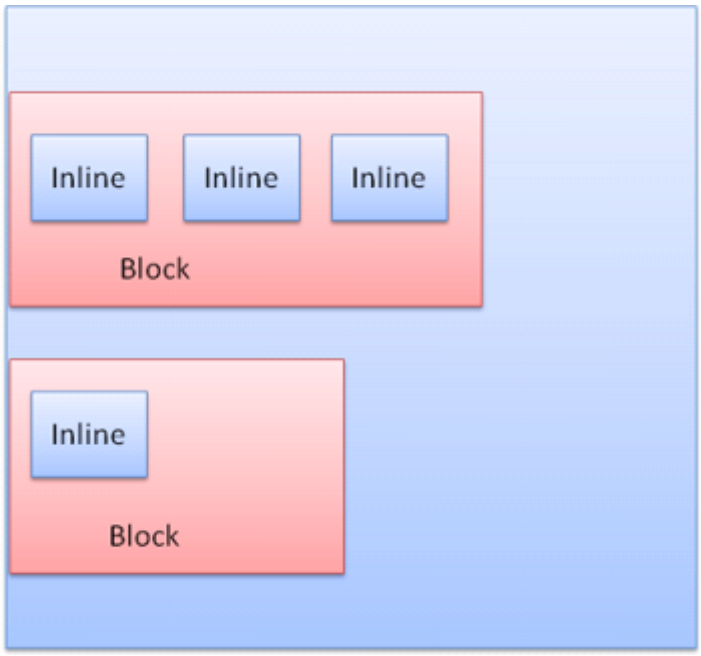


그림 21 블록과 인라인 배치

인라인 박스는 라인 또는 "라인 박스" 안쪽에 놓인다. 라인은 적어도 가장 큰 박스만큼 크지만 "baseline" 정렬일 때 더 커질 수 있다. 이것은 요소의 하단이 다른 상자의 하단이 아닌 곳에 배치된 경우를 의미한다. 포함하는 너비가 충분하지 않으면 인라인은 몇 줄의 라인으로 배치되는데 이것은 보통 문단 안에서 발생한다.

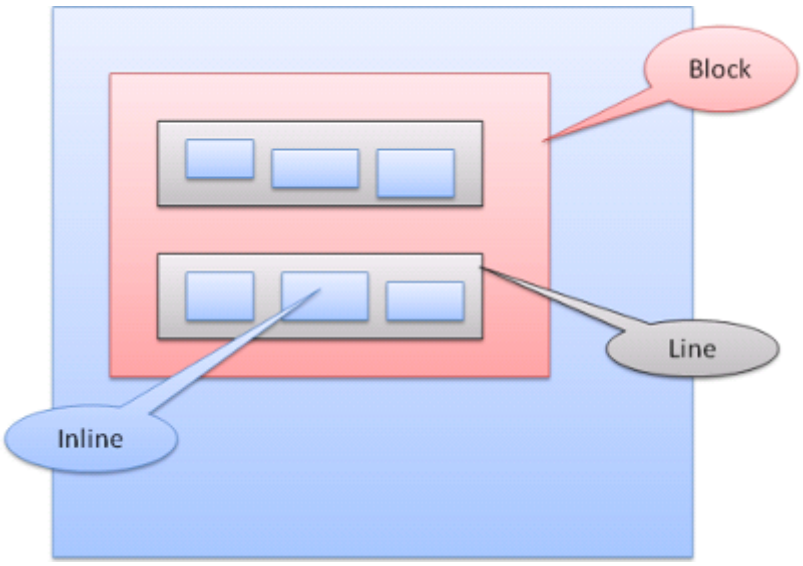


그림 22 라인

위치 잡기

상대적인 위치

상대적인 위치 잡기는 일반적인 흐름에 따라 위치를 결정한 다음 필요한 만큼 이동한다.

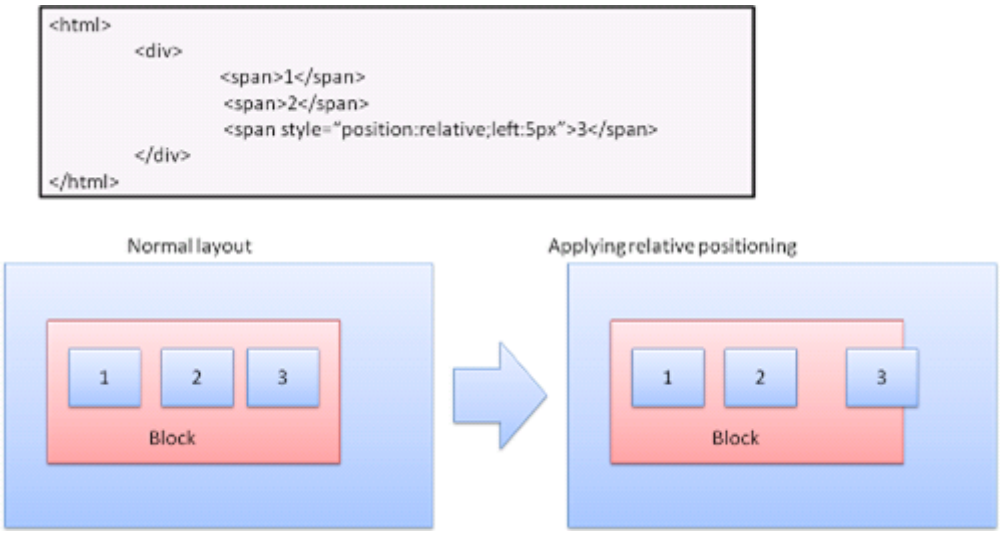


그림 23 상대적인 위치 잡기

플로트

플로트 박스는 라인의 왼쪽 또는 오른쪽으로 이동한다. 흥미로운 점은 다른 박스가 이 주변을 흐른다는 것이다.

HTML을 다음과 같이 작성하면.

```
<p>
<img src="http://helloworld.naver.com/image.gif" alt="image.gif" width="100"
height="100" style="float:right">
Lorem ipsum dolor sit amet, consectetuer...
</p>
```

아래와 같이 보일 것이다.

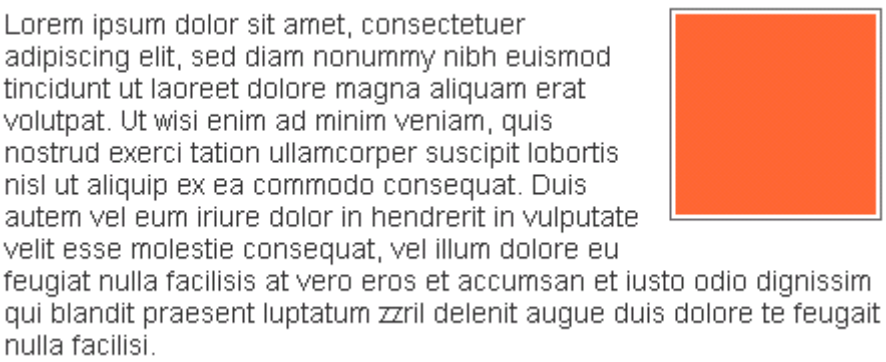


그림 24 플로트

절대적인(absolute) 위치와 고정된(fixed) 위치

절대와 고정 배치는 일반적인 흐름과 무관하게 결정되고, 일반적인 흐름에 관여하지 않으며, 면적은 부모에 따라 상대적이다. 고정인 경우 뷰포트로부터 위치를 결정한다.

```
<html>
        <div>
                <span>1</span>
                <span>2</span>
                <span style="position:fixed;top:5px,left:5px">3</span>
        </div>
</html>
```

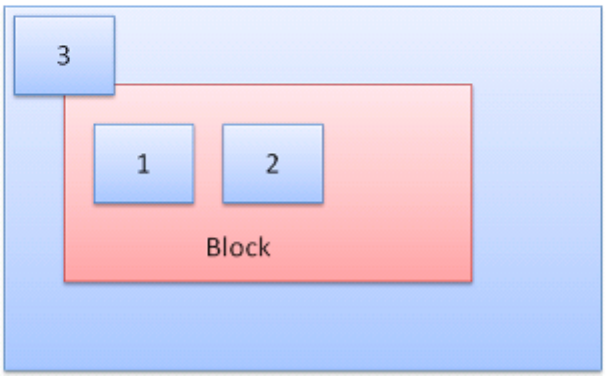


그림 25 고정된 위치 잡기

참고 - 고정된 박스는 문서가 스크롤되어도 따라 움직이지 않는다.

층 표현

이것은 CSS의 z-index 속성에 의해 명시된다. 층은 박스의 3차원 표현이고 "z 축"을 따라 위치를 정한다.

박스는 (stacking contexts라고 부르는) 스택으로 구분된다. 각 스택에서 뒤쪽 요소가 먼저 그려지고 앞쪽 요소는 사용자에게 가까운 쪽으로 나중에 그려진다. 가장 앞쪽에 위치한 요소는 겹치는 이전 요소를 가린다.

스택은 z-index 속성에 따라 순서를 결정한다. z-index 속성이 있는 박스는 지역 스택(local stack)을 형성한다. 뷰포트는 바깥쪽의 스택(outer stack)이다.

다음 예제를 보자.

```
<style type="text/css">
div {
    position: absolute;
    left: 2in;
    top: 2in;
}
</style>
<p>
    <div style="z-index:3;background-color:red;width:1in;height:1in"></div>
    <div style="z-index:1;background-color:green;width:2in;height:2in"></div>
</p>
```

이 코드는 다음과 같이 보일 것이다.

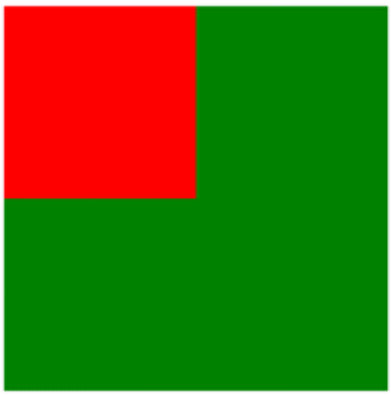


그림 26 고정 위치 잡기

붉은색 박스가 초록색 박스보다 마크업에서 먼저 나오기 때문에 일반적인 흐름이라면 먼저 그려져야 하지만 z-index 속성이 높기 때문에 더 앞쪽에 표시된다.

모든 내용 출처:

https://d2.naver.com/helloworld/59361

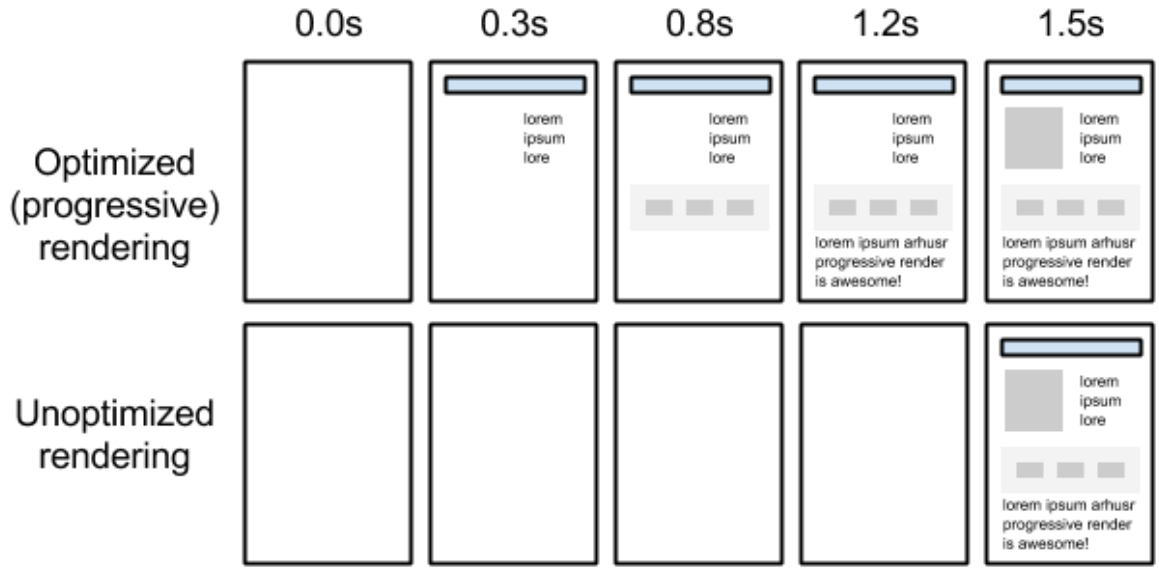
🔍Web fundamentals - Critical-rendering-path

주요 렌더링 경로bookmark_border

*주요 렌더링 경로 최적화*란 현재 사용자 작업과 관련된 콘텐츠 표시의 우선순위를 지정하는 것을 말합니다.

빠른 웹 환경을 제공하려면 브라우저가 많은 작업을 수행해야 합니다. 이러한 작업 대부분은 웹 개발자에게 숨겨져 있습니다. 즉, 개발자가 마크업을 작성하면 그저 멋진 페이지가 화면에 표시될 뿐이죠. 그렇다면 브라우저가 HTML, CSS 및 자바스크립트를 사용하여 화면에 렌더링된 픽셀로 변환하는 과정은 정확히 어떻게 될까요?

성능을 최적화하려면 HTML, CSS 및 자바스크립트 바이트를 수신한 후 렌더링된 픽셀로 변환하기 위해 필요한 처리까지, 그 사이에 포함된 중간 단계에서 어떠한 일이 일어나는지를 파악하기만 하면 됩니다. 이러한 단계가 바로 **주요 렌더링 경로**입니다.



주요 렌더링 경로를 최적화하면 최초 페이지 렌더링에 걸리는 시간을 상당히 단축시킬 수 있습니다. 또한, 주요 렌더링 경로에 대한 이해를 토대로 뛰어난 성능의 대화형 애플리케이션을 빌드할 수도 있습니다. 대화형 업데이트 프로세스도 이와 동일합니다. 연속 루프에서 실행되며 이상적인 속도는 초당 60프레임입니다. 그러나 먼저 브라우저에서 간단한 페이지를 표시하는 방법을 살펴봅시다.

객체 모델 생성bookmark_border

브라우저가 페이지를 렌더링하려면 먼저 DOM 및 CSSOM 트리를 생성해야 합니다. 따라서 HTML 및 CSS를 가능한 한 빨리 브라우저에 제공해야 합니다.

TL;DR

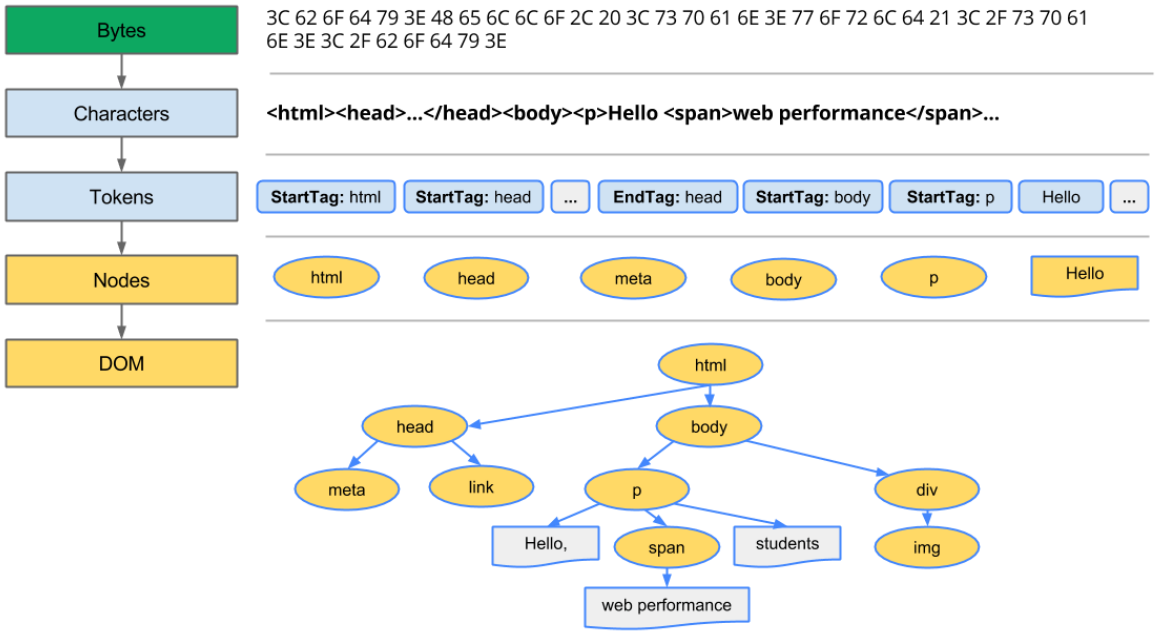
- 바이트 → 문자 → 토큰 → 노드 → 객체 모델.
- HTML 마크업은 DOM(Document Object Model)으로 변환되고, CSS 마크업은 CSSOM(CSS Object Model)으로 변환됩니다.
- DOM 및 CSSOM은 서로 독립적인 데이터 구조입니다.
- Chrome DevTools Timeline을 사용하면 DOM 및 CSSOM의 생성 및 처리 비용을 수집하고 점검할 수 있습니다.

DOM(Document Object Model)

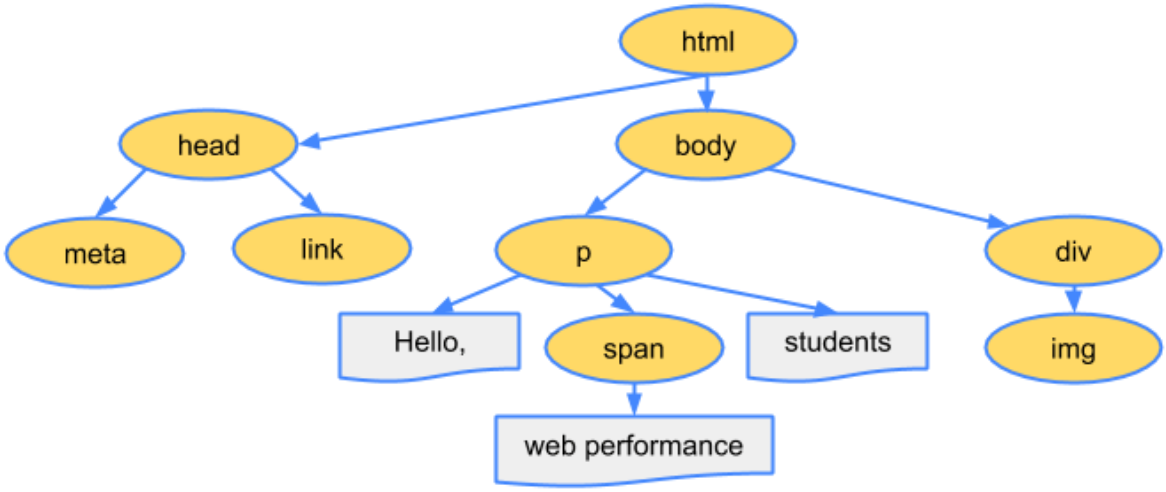
```
 <!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

체험해 보기

가장 단순한 경우인 몇몇 텍스트와 하나의 이미지만 포함하는 일반 HTML 페이지부터 살펴보도록 하겠습니다. 브라우저가 이 페이지를 어떻게 처리하나요?



1. **변환:** 브라우저가 HTML의 원시 바이트를 디스크나 네트워크에서 읽어와서, 해당 파일에 대해 지정된 인코딩(예: UTF-8)에 따라 개별 문자로 변환합니다.
2. **토큰화:** 브라우저가 문자열을 W3C HTML5 표준에 지정된 고유 토큰으로 변환합니다(예: '', '' 및 꺽쇠괄호로 묶인 기타 문자열). 각 토큰은 특별한 의미와 고유한 규칙을 가집니다.
3. **렉싱:** 방출된 토큰은 해당 속성 및 규칙을 정의하는 '객체'로 변환됩니다.
4. **DOM 생성:** 마지막으로, HTML 마크업이 여러 태그(일부 태그는 다른 태그 안에 포함되어 있음) 간의 관계를 정의하기 때문에 생성된 객체는 트리 데이터 구조 내에 연결됩니다. 이 트리 데이터 구조에는 원래 마크업에 정의된 상위-하위 관계도 포함됩니다. 즉, *HTML* 객체는 *body* 객체의 상위이고, *body* 는 *paragraph* 객체의 상위인 식입니다.



이 전체 프로세스의 최종 출력이 바로 이 간단한 페이지의 DOM(Document Object Model)이며, 브라우저는 이후 모든 페이지 처리에 이 DOM을 사용합니다.

브라우저는 HTML 마크업을 처리할 때마다 위의 모든 단계를 수행합니다. 즉, 바이트를 문자로 변환하고, 토큰을 식별한 후 노드로 변환하고 DOM 트리를 빌드합니다. 이 전체 프로세스를 완료하려면 시간이 약간 걸릴 수 있으며, 특히 처리해야 할 HTML이 많은 경우 그렇습니다.

참고: 여기서는 Chrome DevTools에 대한 기본적인 사항, 즉 네트워크 워터폴(waterfall)을 캡처하거나 타임라인을 기록하는 방법에 대해 알고 있다고 가정합니다. DevTools에 대해 한 번 더 간단하게 되짚어 보려면 Chrome DevTools 문서를 확인하고, DevTools를 처음 사용하는 경우에는 Codeschool의 Discover DevTools 과정을 학습할 것을 권장합니다.

Chrome DevTools를 열고 페이지가 로드되는 동안 타임라인을 기록하면 이 단계를 수행하는 데 소요된 실제 시간을 확인할 수 있습니다. 위 예시에서는 HTML 조각을 DOM 트리로 변환하는 데 약 5ms 정도 걸립니다. 큰 페이지의 경우 이 프로세스가 훨씬 더 오래 걸릴 수 있습니다. 매끄러운 애니메이션을 만드는 경우, 브라우저가 대량의 HTML을 처리해야 한다면 쉽게 병목 현상이 발생할 수 있습니다.

DOM 트리는 문서 마크업의 속성 및 관계를 포함하지만 요소가 렌더링될 때 어떻게 표시될지에 대해서는 알려주지 않습니다. 이것은 CSSOM의 책임입니다.

CSSOM(CSS Object Model)

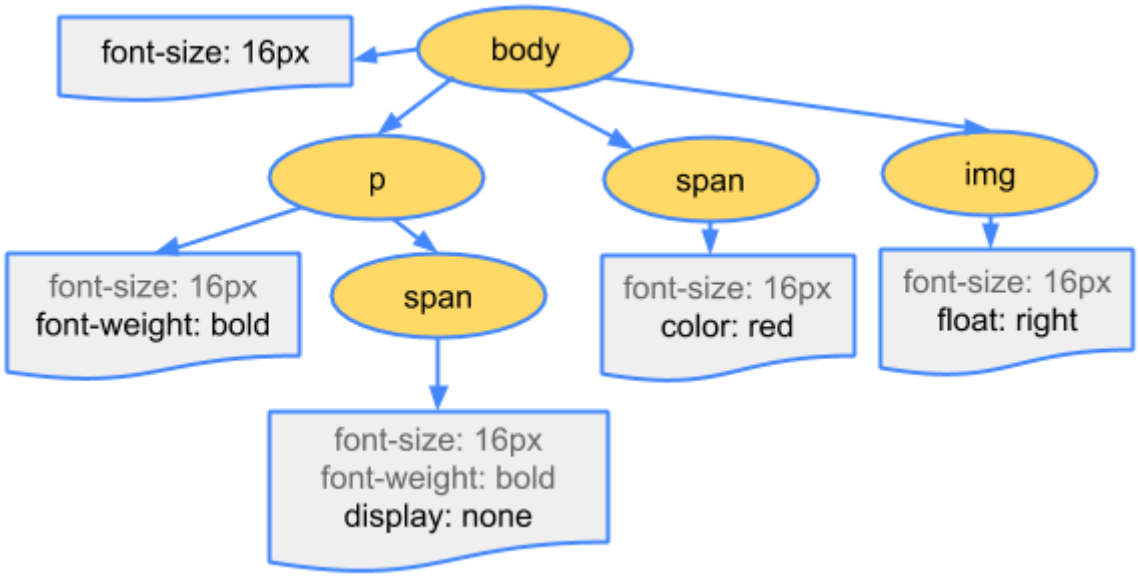
브라우저는 단순한 페이지의 DOM을 생성하는 동안 외부 CSS 스타일시트인 style.css를 참조하는 문서의 헤드 섹션에서 링크 태그를 접합니다. 페이지를 렌더링하는 데 이 리소스가 필요할 것이라고 판단한 브라우저는 이 리소스에 대한 요청을 즉시 발송하고 요청의 결과로 다음 콘텐츠가 반환됩니다.

```
 body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
img { float: right }
```

HTML 마크업 내에 직접(인라인) 스타일을 선언할 수도 있지만 CSS를 HTML과 별도로 유지하면 콘텐츠와 디자인을 별도의 항목으로 처리할 수 있습니다. 즉, 디자이너는 CSS를 처리하고, 개발자는 HTML에만 집중할 수 있습니다.

HTML과 마찬가지로, 수신된 CSS 규칙을 브라우저가 이해하고 처리할 수 있는 형식으로 변환해야 합니다. 따라서 HTML 대신 CSS에 대해 HTML 프로세스를 반복합니다.

CSS 바이트가 문자로 변환된 후 차례로 토큰과 노드로 변환되고 마지막으로 'CSS Object Model'(CSSOM)이라는 트리 구조에 링크됩니다.

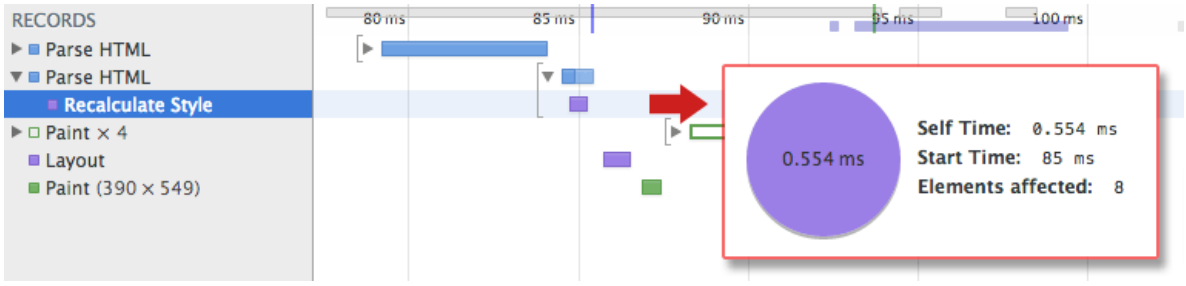


CSSOM이 트리 구조를 가지는 이유는 무엇일까요? 페이지에 있는 객체의 최종 스타일을 계산할 때 브라우저는 해당 노드에 적용 가능한 가장 일반적인 규칙(예: body 요소의 하위인 경우 모든 body 스타일 적용)으로 시작한 후 더욱 구체적인 규칙을 적용하는 방식으로, 즉 '하향식'으로 규칙을 적용하는 방식으로 계산된 스타일을 재귀적으로 세분화합니다.

더욱 구체화하기 위해 위에 나와 있는 CSSOM 트리를 살펴봅시다. body 요소 내에 있는 *span* 태그 안에 포함된 모든 텍스트의 글꼴 크기는 16픽셀이고 색상은 빨간색입니다. font-size 지시문은 body에서 span으로 하향식으로 적용되기 때문입니다. 하지만 span 태그가 단락(p) 태그의 하위인 경우 해당 콘텐츠는 표시되지 않습니다.

또한, 위의 트리는 완전한 CSSOM 트리가 아니고 스타일시트에서 재정의하도록 결정한 스타일만 표시한다는 점에 유의하세요. 모든 브라우저는 '사용자 에이전트 스타일'이라고 하는 기본 스타일 집합, 즉 개발자가 고유한 스타일을 제공하지 않을 경우 표시되는 스타일을 제공합니다. 개발자가 작성하는 스타일은 이러한 기본 스타일(예: 기본 IE 스타일)을 간단하게 재정의합니다.

CSS 처리에 시간이 얼마나 걸리는지 알기 위해, DevTools에서 타임라인을 기록하고 'Recalculate Style' 이벤트를 찾을 수 있습니다. DOM 파싱과 달리, 타임라인에 'Parse CSS' 항목이 별도로 표시되지 않으며, 대신 파싱 및 CSSOM 트리 생성과 계산된 스타일의 재귀적 계산이 이 단일 이벤트에서 캡처됩니다.



작은 스타일시트를 처리하는 데 0.6ms 미만이 걸리며, 페이지에 있는 8개 요소에 영향을 미칩니다. 많지는 않지만 비용이 전혀 안 드는 것은 아니죠. 그런데 8개 요소는 어디서 왔을까요? CSSOM 및 DOM은 서로 독립적인 데이터 구조입니다. 알고보니 브라우저에서 숨겨진 중요한 단계가 있습니다. 다음 섹션에서 DOM 및 CSSOM을 함께 연결하는 렌더링 트리에 대해 살펴보도록 하겠습니다.

렌더링 트리 생성, 레이아웃 및 페인트 bookmark_border

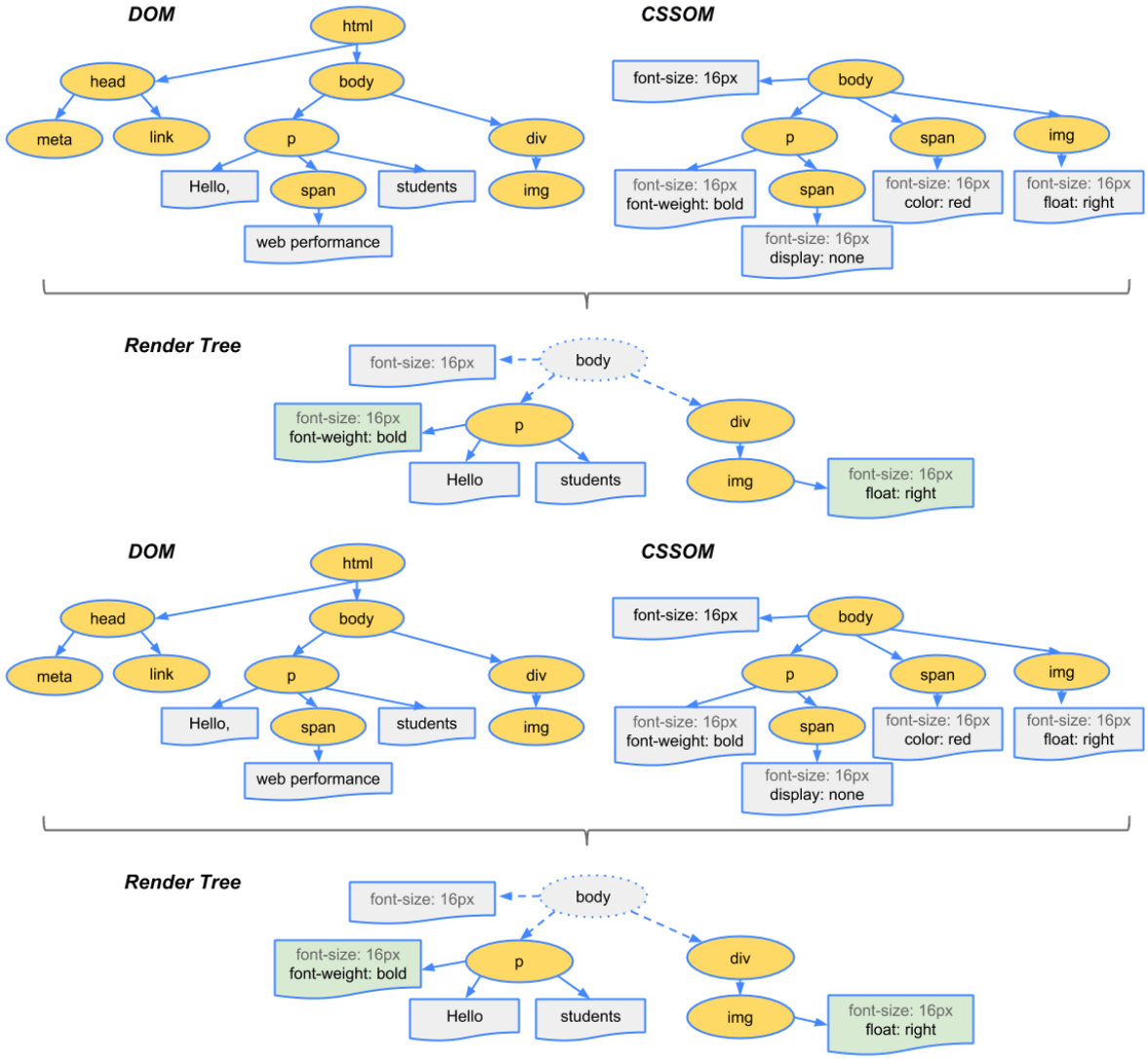
CSSOM 및 DOM 트리는 결합하여 렌더링 트리를 형성합니다. 이 렌더링 트리는 표시되는 각 요소의 레이아웃을 계산하는 데 사용되고 픽셀을 화면에 렌더링하는 페인트 프로세스에 대한 입력으로 처리됩니다. 최적의 렌더링 성능을 얻기 위해서는 이러한 단계 각각을 최적화하는 것이 중요합니다.

객체 모델을 생성하는 방법을 설명한 이전 섹션에서 우리는 HTML 및 CSS 입력을 기반으로 DOM 및 CSSOM 트리를 빌드했습니다. 하지만, 이들 모두 문서의 각기 다른 측면을 캡처하는 서로 독립적인 객체입니다. 하나는 콘텐츠를 설명하고, 다른 하나는 문서에 적용되어야 하는 스타일 규칙을 설명합니다. 이 두 가지를 병합하여 브라우저가 화면에 픽셀을 렌더링하도록 하려면 어떻게 해야 할까요?

TL;DR

- DOM 및 CSSOM 트리는 결합되어 렌더링 트리를 형성합니다.
- 렌더링 트리에는 페이지를 렌더링하는 데 필요한 노드만 포함됩니다.
- 레이아웃은 각 객체의 정확한 위치 및 크기를 계산합니다.
- 마지막 단계는 최종 렌더링 트리에서 수행되는 페인트이며, 픽셀을 화면에 렌더링합니다.

먼저, 브라우저가 DOM 및 CSSOM을 '렌더링 트리'에 결합합니다. 이 트리는 페이지에 표시되는 모든 DOM 콘텐츠와 각 노드에 대한 모든 CSSOM 스타일 정보를 캡처합니다.



렌더링 트리를 생성하려면 브라우저가 대략적으로 다음 작업을 수행합니다.

1. DOM 트리의 루트에서 시작하여 표시되는 노드 각각을 트래버스합니다.
    - 일부 노드는 표시되지 않으며(예: 스크립트 태그, 메타 태그 등), 렌더링된 출력에 반영되지 않으므로 생략됩니다.
    - 일부 노드는 CSS를 통해 숨겨지며 렌더링 트리에서도 생략됩니다. 예를 들어,---위의 예시에서---span 노드의 경우 'display: none' 속성을 설정하는 명시적 규칙이 있기 때문에 렌더링 트

리에서 누락됩니다.
2. 표시된 각 노드에 대해 적절하게 일치하는 CSSOM 규칙을 찾아 적용합니다.
3. 표시된 노드를 콘텐츠 및 계산된 스타일과 함께 내보냅니다.

참고: 간단한 여담으로, `visibility: hidden`은 `display: none`과 다릅니다. 전자는 요소를 보이지 않게 만들지만, 이 요소는 여전히 레이아웃에서 공간을 차지합니다(즉, 비어 있는 상자로 렌더링됨). 반면, 후자(`display: none`)는 요소가 보이지 않으며 레이아웃에 포함되지도 않도록 렌더링 트리에서 요소를 완전히 제거합니다.

최종 출력은 화면에 표시되는 모든 노드의 콘텐츠 및 스타일 정보를 모두 포함하는 렌더링 트리입니다. **렌더링 트리가 생성되었으므로 '레이아웃' 단계로 진행할 수 있습니다.**

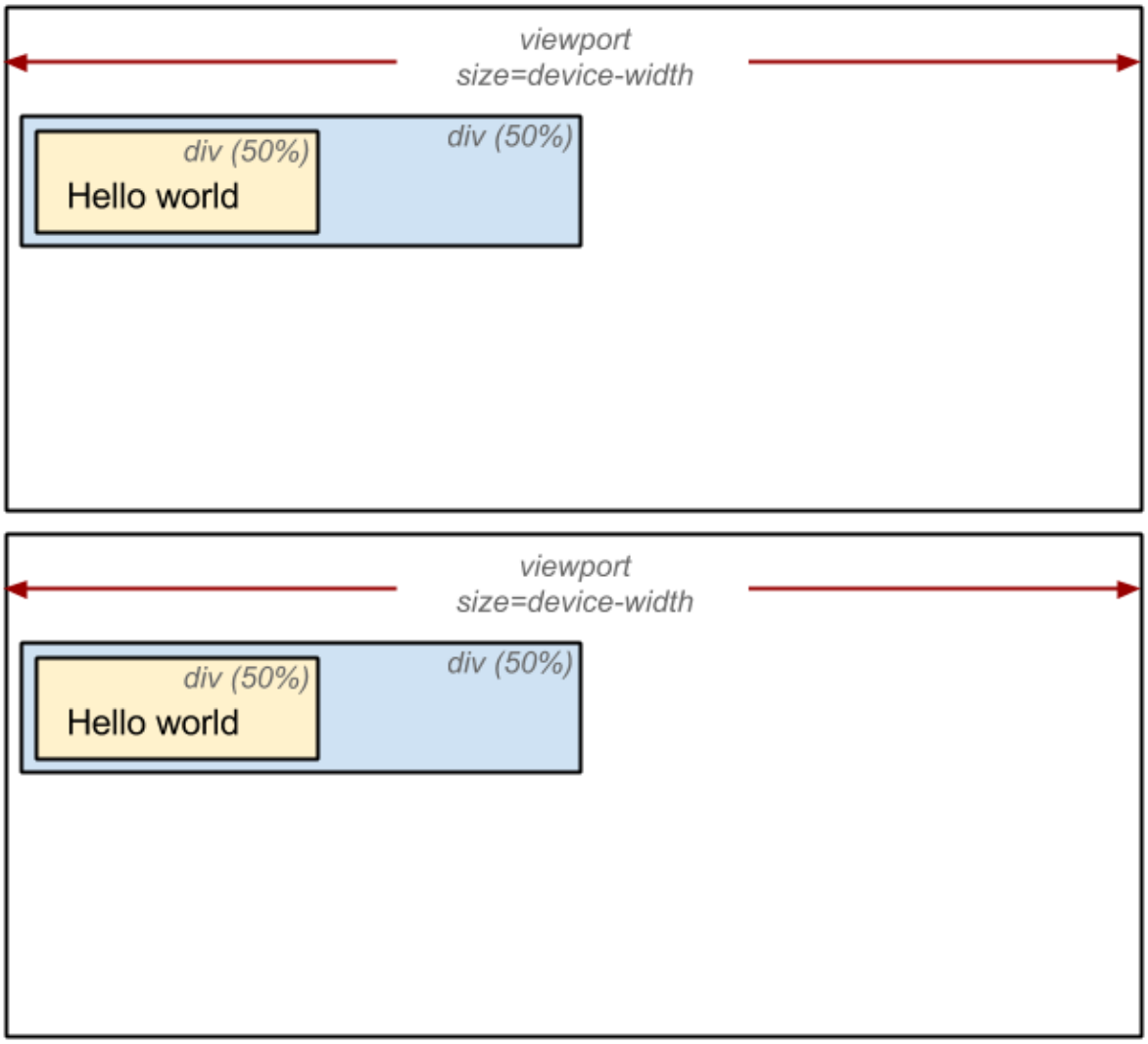
지금까지 표시할 노드와 해당 노드의 계산된 스타일을 계산했습니다. 하지만 기기의 뷰포트 내에서 이러한 노드의 정확한 위치와 크기를 계산하지는 않았습니다.---이것이 바로 '레이아웃' 단계이며, 경우에 따라 '리플로우'라고도 합니다.

페이지에서 각 객체의 정확한 크기와 위치를 파악하기 위해 브라우저는 렌더링 트리의 루트에서 시작하여 렌더링 트리를 트래버스합니다. 간단한 실습 예시를 살펴보도록 하겠습니다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Critial Path: Hello world!</title>
  </head>
  <body>
    <div style="width: 50%">
      <div style="width: 50%">Hello world!</div>
    </div>
  </body>
</html>
```

체험해 보기

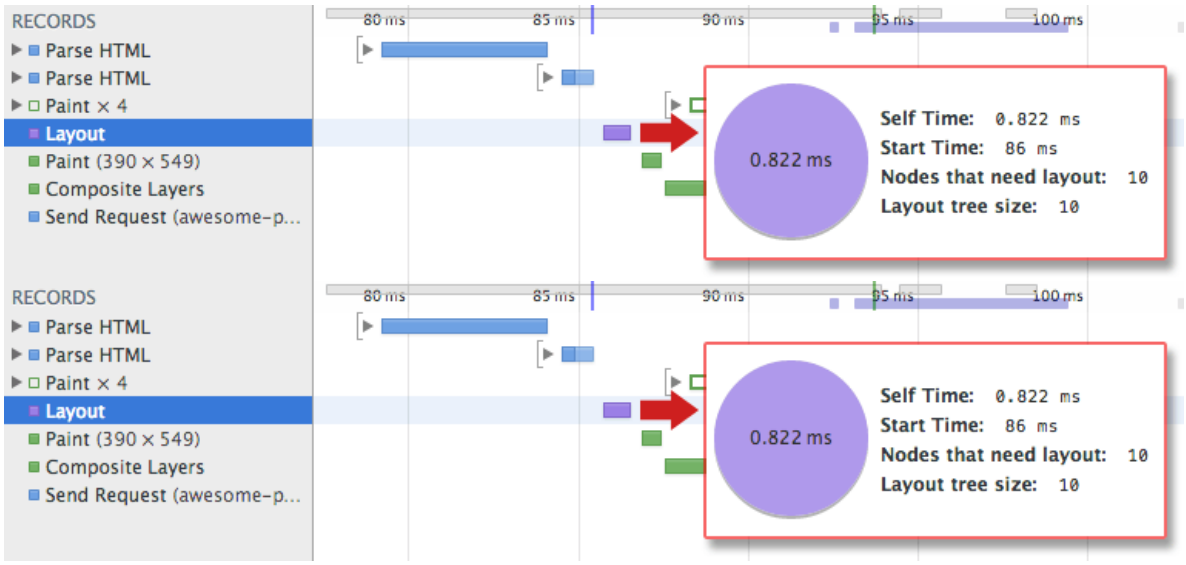
위 페이지의 본문에는 두 가지 중첩된 div가 포함되어 있습니다. 첫 번째(상위) div는 노드의 표시 크기를 뷰포트 너비의 50%로 설정하며,---상위 div에 포함된---두 번째 div는 해당 너비를 상위 항목 너비의 50%(즉, 뷰포트 너비의 25%)로 설정합니다.



레이아웃 프로세스에서는 뷰포트 내에서 각 요소의 정확한 위치와 크기를 정확하게 캡처하는 '상자 모델'이 출력됩니다. 모든 상대적인 측정값은 화면에서 절대적인 픽셀로 변환됩니다.

마지막으로, 이제 표시되는 노드와 해당 노드의 계산된 스타일 및 기하학적 형태에 대해 파악했으므로, 렌더링 트리의 각 노드를 화면의 실제 픽셀로 변환하는 마지막 단계로 이러한 정보를 전달할 수 있습니다. 이 단계를 흔히 '페인팅' 또는 '래스터화'라고 합니다.

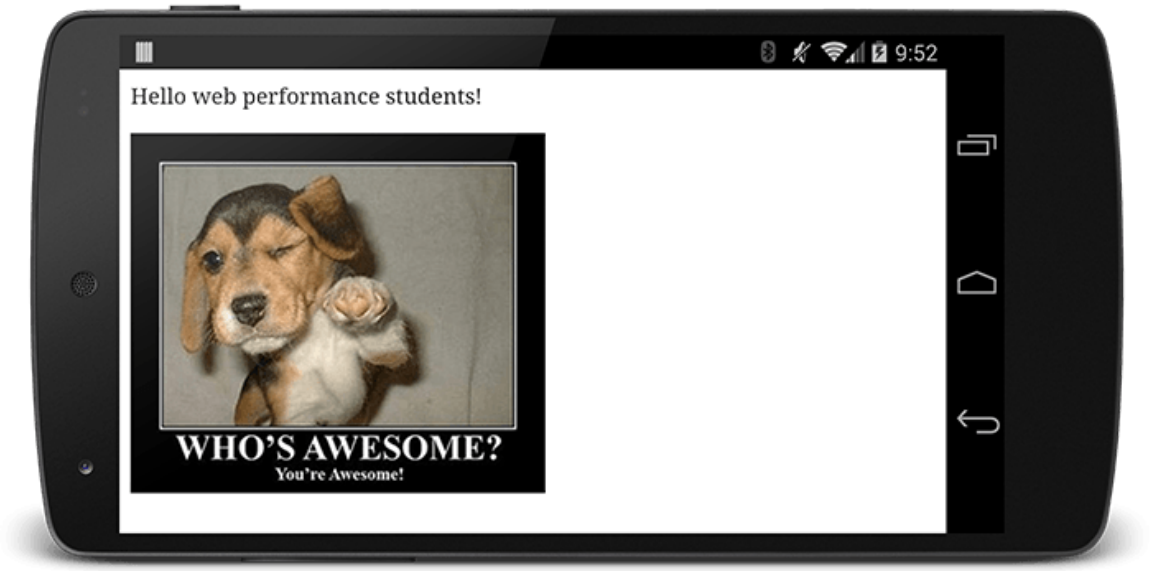
이 경우 브라우저가 처리해야 할 작업이 상당히 많으므로 시간이 약간 걸릴 수 있습니다. 그러나 Chrome DevTools는 위에 설명된 세 단계 모두에 대해 몇 가지 정보를 제공할 수 있습니다. 원래 'hello world' 예시의 레이아웃 단계를 검토해 보도록 하겠습니다.

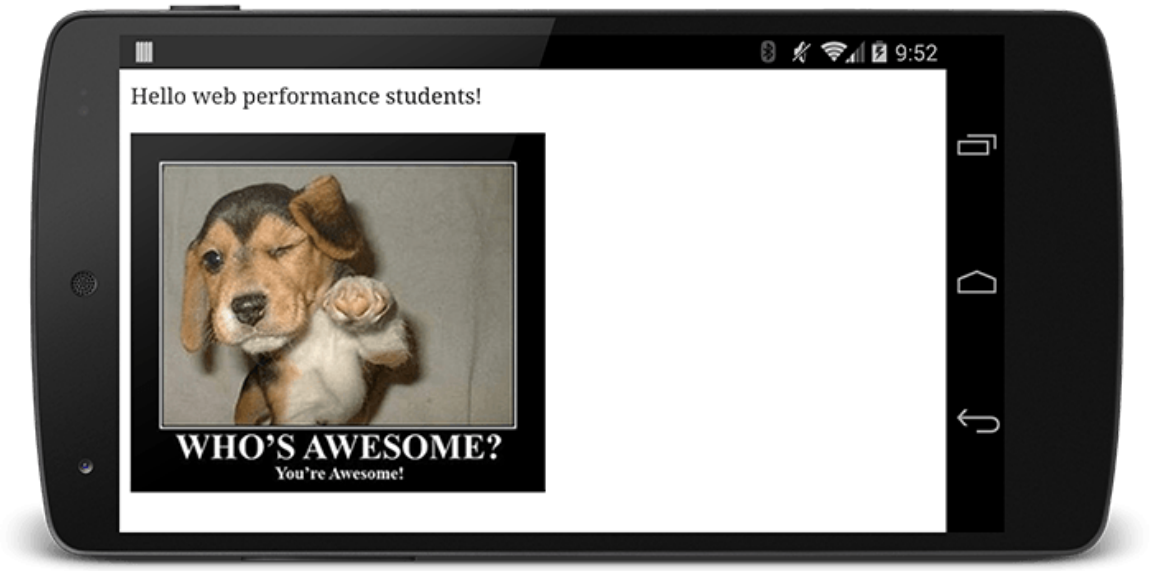


- 'Layout' 이벤트는 타임라인에서 렌더링 트리 생성, 위치 및 크기 계산을 캡처합니다.
- 레이아웃이 완료될 때 브라우저가 'Paint Setup' 및 'Paint' 이벤트를 발생시킵니다. 이러한 작업은 렌더링 트리를 화면의 픽셀로 변환합니다.

렌더링 트리 생성, 레이아웃 및 페인트 작업을 수행하는 데 필요한 시간은 문서의 크기, 적용된 스타일 및 실행 중인 기기에 따라 달라집니다. 즉, 문서가 클수록 브라우저가 수행해야 하는 작업도 더 많아지며, 스타일이 복잡할수록 페인팅에 걸리는 시간도 늘어납니다. 예를 들어, 단색은 페인트하는 데 시간과 작업이 적게 필요한 반면, 그림자 효과는 계산하고 렌더링하는 데 시간과 작업이 더 필요합니다.

페이지가 드디어 뷰포트에 표시됩니다.





다음은 브라우저의 단계를 빠르게 되짚어 보겠습니다.

1. HTML 마크업을 처리하고 DOM 트리를 빌드합니다.
2. CSS 마크업을 처리하고 CSSOM 트리를 빌드합니다.
3. DOM 및 CSSOM을 결합하여 렌더링 트리를 형성합니다.
4. 렌더링 트리에서 레이아웃을 실행하여 각 노드의 기하학적 형태를 계산합니다.
5. 개별 노드를 화면에 페인트합니다.

여기에 표시된 데모 페이지는 간단해 보일 수 있지만, 이 페이지에도 꽤 많은 작업이 필요합니다. DOM 또는 CSSOM이 수정된 경우, 화면에 다시 렌더링할 필요가 있는 픽셀을 파악하려면 이 프로세스를 다시 반복해야 합니다.

***주요 렌더링 경로를 최적화하는 작업* 은 위 단계에서 1단계~5단계를 수행할 때 걸린 총 시간을 최소화하는 프로세스입니다.** 이렇게 하면 콘텐츠를 가능한 한 빨리 화면에 렌더링할 수 있으며, 초기 렌더링 후 화면 업데이트 사이의 시간을 줄여 줍니다. 따라서 대화형 콘텐츠의 새로고침 속도를 높일 수 있습니다.

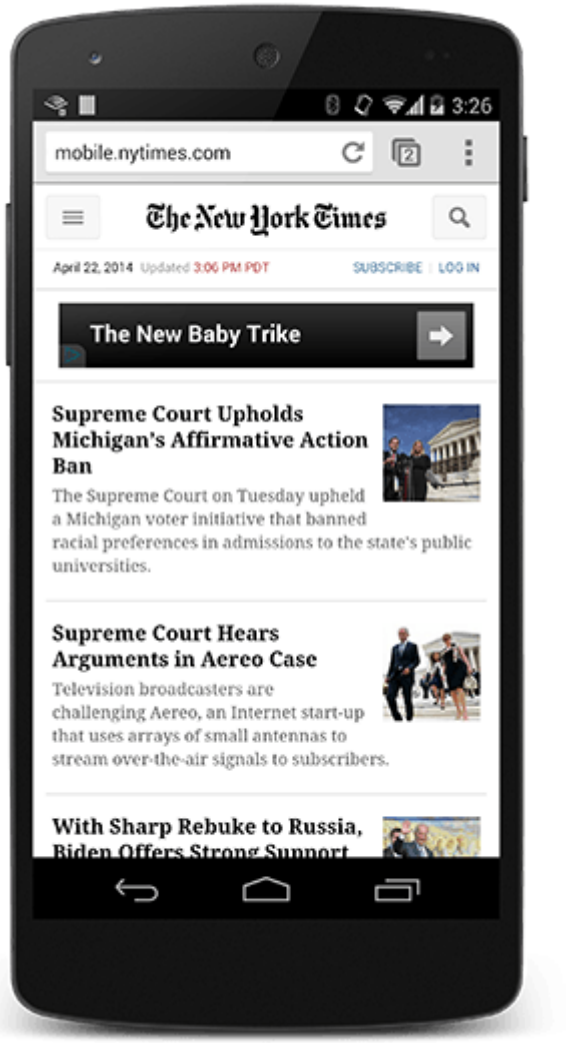
렌더링 차단 CSSbookmark_border

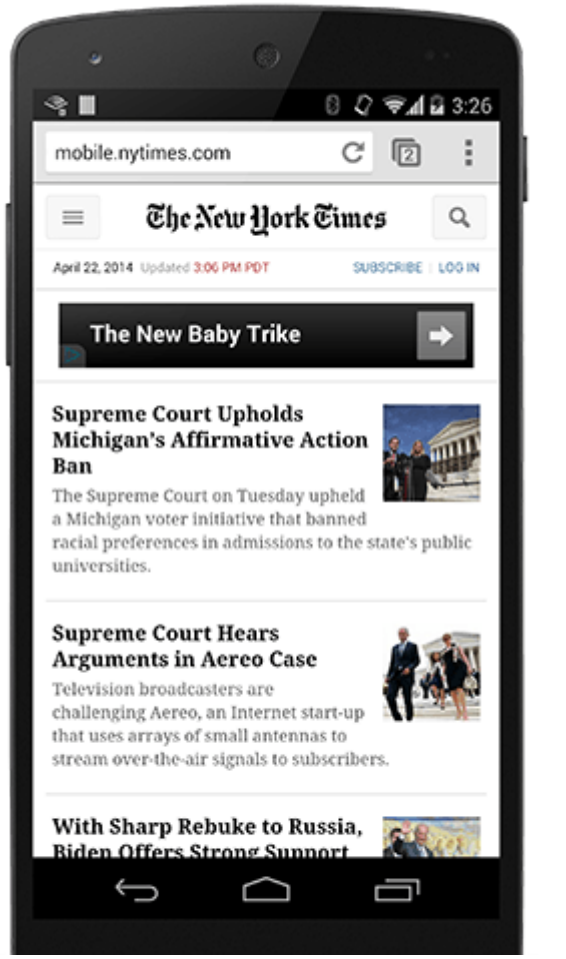
기본적으로, CSS는 렌더링 차단 리소스로 취급됩니다. 즉, CSSOM이 생성될 때까지 브라우저는 처리되는 모든 콘텐츠를 렌더링하지 않습니다. CSS를 간단하게 유지하고 가능한 한 빨리 제공하고, 미디어 유형과 미디어 쿼리를 사용하여 렌더링의 차단을 해제해야 합니다.

렌더링 트리 생성에서 우리는 렌더링 트리를 생성하는 데 DOM 및 CSSOM이 둘다 필요하다는 점을 확인했습니다. 이것은 성능에 중요한 영향을 미칩니다. **HTML 및 CSS는 둘다 렌더링 차단 리소스입니다.** HTML의 경우 DOM이 없으면 렌더링할 것이 없기 때문에 명확하지만, CSS 요구사항은 다소 불명확할 수 있습니다. CSS에서 렌더링을 차단하지 않고 일반 페이지를 렌더링하려고 하면 어떠한 일이 벌어질까요?

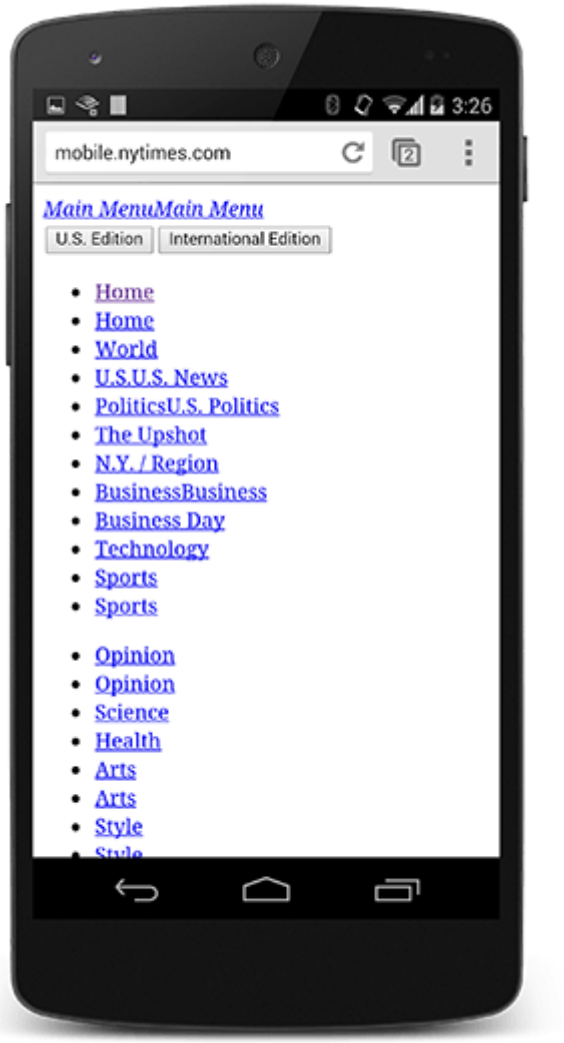
TL;DR

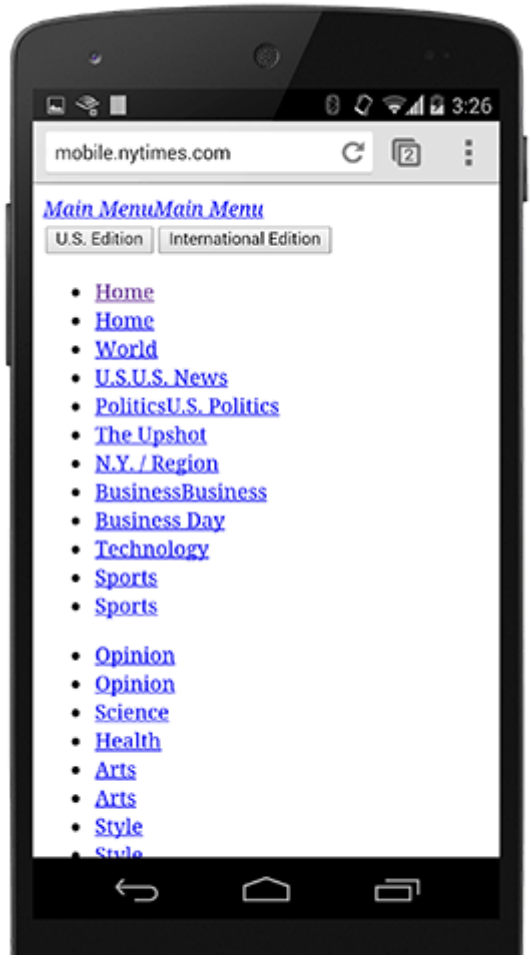
- 기본적으로, CSS는 렌더링 차단 리소스로 취급됩니다.
- 미디어 유형과 미디어 쿼리를 통해 일부 CSS 리소스를 렌더링을 비차단 리소스로 표시할 수 있습니다.
- 브라우저는 차단 동작이든 비차단 동작이든 관계없이 모든 CSS 리소스를 다운로드합니다.





The New York Times(CSS 사용)





The New York Times(CSS 미사용)(FOUC)

위의 예에서는 CSS가 사용 가능해질 때까지 렌더링이 차단되는 이유를 설명하기 위해 CSS가 있는 NYTimes 웹사이트와 CSS가 없는 NYTimes 웹사이트를 보여줍니다.---CSS가 없는 페이지는 상대적으로 사용성이 떨어집니다. 오른쪽의 환경은 흔히 'Flash of Unstyled Content'(FOUC)로 불립니다. 브라우저는 DOM과 CSSOM을 모두 사용할 수 있게 될 때까지 렌더링을 차단합니다.

> ***CSS는 렌더링 차단 리소스입니다. 최초 렌더링에 걸리는 시간을 최적화하려면 클라이언트에 최대한 빠르게 다운로드되어야 합니다.***

하지만 특정한 조건, 예를 들어 페이지가 인쇄될 때나 대형 모니터에 출력하는 경우에만 사용되는 몇 가지 CSS 스타일이 있다면 어떻게 될까요? 이러한 리소스에서 렌더링을 차단할 필요가 없었다면 좋았을 것입니다.

CSS '미디어 유형'과 '미디어 쿼리'를 사용하면 이러한 사용 사례를 해결할 수 있습니다.

```
<link href="style.css" rel="stylesheet">
<link href="print.css" rel="stylesheet" media="print">
<link href="other.css" rel="stylesheet" media="(min-width: 40em)">
```

미디어 쿼리는 하나의 미디어 유형과 특정 미디어 기능의 조건을 확인하는 0개 이상의 식으로 구성됩니다. 예를 들어, 우리의 첫 번째 스타일시트 선언은 미디어 유형이나 미디어 쿼리를 제공하지 않으며, 따라서 모든 경우에 적용됩니다. 다시 말해서, 항상 렌더링을 차단합니다. 반면에, 두 번째 스타일시트 선언은 콘텐츠가 인쇄될 때만 적용됩니다.---아마도 여러분은 레이아웃을 다시 정렬하거나 글꼴을 변경하는 등의 기능을 원할 것입니다. 따라서 이 스타일시트 선언은 처음에 로드될 때 페이지 렌더링을 차단할 필요가 없습니다. 마지막으로, 마지막 스타일시트 선언은 브라우저가 실행하는 '미디어 쿼리'를 제공합니다. 조건이 일치하면 스타일시트가 다운로드되고 처리될 때까지 브라우저가 렌더링을 차단합니다.

미디어 쿼리를 사용하면 우리가 특정한 사용 사례(예: 표시 또는 인쇄)와 동적인 조건(예: 화면 방향 변경, 크기 조정 이벤트 등)에 맞게 프레젠테이션을 조정할 수 있습니다. **스타일시트 자산을 선언할 때 미디어 유형과 미디어 쿼리에 세심한 주의를 기울여야 합니다. 이러한 요소들은 주요 렌더링 경로의 성능에 큰 영향을 미칩니다.**

몇 가지 실습 예시를 살펴봅시다.

```
<link href="style.css"    rel="stylesheet">
<link href="style.css"    rel="stylesheet" media="all">
<link href="portrait.css" rel="stylesheet" media="orientation:portrait">
<link href="print.css"    rel="stylesheet" media="print">
```

- 첫 번째 선언은 렌더링을 차단하고 모든 조건에서 일치합니다.
- 두 번째 선언도 렌더링을 차단합니다. 'all'이 기본 유형이므로 특정 유형을 지정하지 않을 경우 암묵적으로 'all'로 설정됩니다. 따라서 첫 번째와 두 번째 선언은 사실상 똑같습니다.
- 세 번째 선언은 페이지가 로드될 때 평가되는 동적 미디어 쿼리를 가집니다. portrait.css의 렌더링 차단 여부는 페이지가 로드되는 중에 기기의 방향에 따라 달라질 수 있습니다.
- 마지막 선언은 페이지가 인쇄될 때만 적용됩니다. 따라서 페이지가 브라우저에서 처음 로드될 때는 렌더링이 차단되지 않습니다.

마지막으로, '렌더링 차단'은 브라우저가 해당 리소스에 대해 페이지의 초기 렌더링을 보류해야 하는지 여부만 나타냅니다. 어느 경우든지, 비차단 리소스의 우선순위가 낮더라도 브라우저가 여전히 CSS 자산을 다운로드합니다.

자바스크립트로 상호작용 추가 bookmark_border

자바스크립트를 사용하면 콘텐츠, 스타일 지정, 사용자 상호작용에 대한 응답 등 페이지의 거의 모든 측면을 수정할 수 있습니다. 하지만, 자바스크립트는 DOM 생성을 차단하고 페이지가 렌더링될 때 지연시킬 수도 있습니다. 최적의 성능을 제공하려면 자바스크립트를 비동기로 설정하고 주요 렌더링 경로에서 불필요한 자바스크립트를 제거하세요.

TL;DR

- 자바스크립트는 DOM 및 CSSOM을 쿼리하고 수정할 수 있습니다.
- 자바스크립트 실행은 CSSOM을 차단합니다.
- 자바스크립트는 명시적으로 비동기로 선언되지 않은 경우 DOM 생성을 차단합니다.

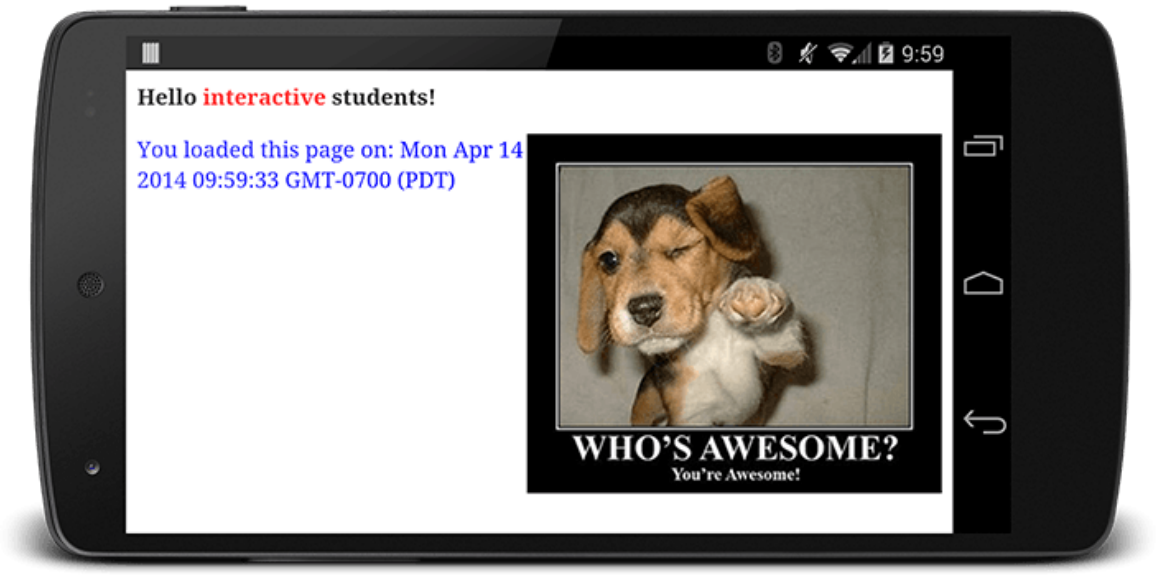
자바스크립트는 브라우저에서 실행되고 페이지 동작 방식에 대한 거의 모든 측면을 변경할 수 있게 하는 동적 언어입니다. DOM 트리에서 요소를 추가하고 제거하여 콘텐츠를 수정하거나, 각 요소의 CSSOM 속성을 수정하거나, 사용자 입력을 처리하는 등의 많은 작업을 수행할 수 있습니다. 이 과정을 보여주기 위해 간단한 인라인 스크립트를 사용하여 이전의 'Hello World' 예시를 확장시켜 보겠습니다.

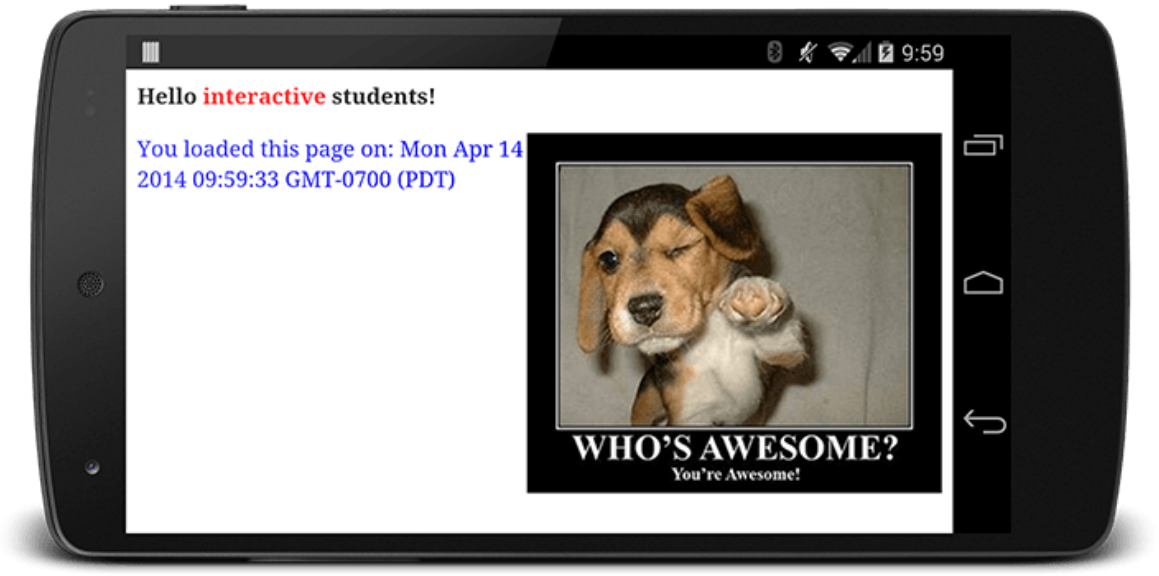
```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script>
      var span = document.getElementsByTagName('span')[0];
      span.textContent = 'interactive'; // change DOM text content
      span.style.display = 'inline';  // change CSSOM property
      // create a new element, style it, and append it to the DOM
      var loadTime = document.createElement('div');
      loadTime.textContent = 'You loaded this page on: ' + new Date();
      loadTime.style.color = 'blue';
      document.body.appendChild(loadTime);
    </script>
  </body>
</html>
```

체험해 보기

- 자바스크립트를 사용하면 DOM에 접근하고 숨겨진 span 노드에 대한 참조를 가져올 수 있습니다. 이러한 숨겨진 노드는 렌더링 트리에 표시되지 않지만, DOM에는 여전히 존재합니다. 그런 다음 참조를 가져오면 (.textContent를 통해) 해당 텍스트를 변경할 수 있으며, 계산된 디스플레이 스타일 속성을 'none'에서 'inline'으로 재정의할 수도 있습니다. 이제 페이지에 '**Hello interactive students!**'가 표시됩니다.
- 자바스크립트를 사용하면 DOM에서 새로운 요소를 생성, 추가, 제거하고 이 요소의 스타일을 지정할 수 있습니다. 기술적으로 볼 때, 전체 페이지는 요소를 하나씩 생성하고 이 요소의 스타일을 지정하는 하나의 커다란 자바스크립트 파일일 수 있습니다. 이 파일도 작동하기는 하지만 실제로는

HTML 및 CSS를 이용하는 것이 훨씬 더 쉽습니다. 자바스크립트 함수의 두 번째 부분에서 새로운 div 요소를 생성하고, 해당 텍스트 콘텐츠를 설정하고, 스타일을 지정하고, 본문에 추가합니다.





이와 함께 기존 DOM 노드의 콘텐츠와 CSS 스타일을 수정하고 완전히 새로운 노드를 문서에 추가했습니다. 이 페이지는 어떠한 디자인 상도 수상하지는 않겠지만 자바스크립트가 제공하는 강력한 기능과 유연성을 보여줍니다.

그러나 자바스크립트는 성능이 뛰어난 반면, 페이지의 렌더링 방식과 시기에 있어 많은 제한이 있습니다.

먼저, 위의 예시에서 인라인 스크립트가 페이지의 맨 아래 부근에 있는 것을 확인할 수 있습니다. 그 이유는 무엇일까요? 여러분 스스로 해보는 것이 좋겠지만, 만약 *span* 요소 위로 스크립트를 이동하면 스크립트가 실패하고 문서에서 *span* 요소에 대한 참조를 찾을 수 없다고 불평할 것입니다. 예를 들어 *getElementsByTagName('span')* 은 *null* 을 반환합니다. 이는 스크립트가 문서에 삽입된 정확한 지점에서 실행된다는 중요한 속성을 보여줍니다. HTML 파서는 스크립트 태그를 만나면 DOM 생성 프로세스를 중지하고 자바스크립트 엔진에 제어 권한을 넘깁니다. 자바스크립트 엔진의 실행이 완료될 후 브라우저가 중지했던 시점부터 DOM 생성을 재개합니다.

다시 말해서, 요소가 아직 처리되지 않았기 때문에 스크립트 블록이 페이지의 뒷부분에서 어떠한 요소도 찾을 수 없습니다. 즉, **인라인 스크립트를 실행하면 DOM 생성이 차단되고, 이로 인해 초기 렌더링도 지연되게 됩니다.**

예시 페이지를 통해 스크립트에 대해 소개할 또 다른 미묘한 속성은 스크립트가 DOM뿐만 아니라 CSSOM 속성도 읽고 수정할 수 있다는 점입니다. 실제로, 이는 예시에서 span 요소의 표시 속성을 none에서 inline으로 변경할 때 우리가 수행한 작업입니다. 최종 결과는 어떻게 될까요? 이제 경합 조건이 생성되었습니다.

스크립트를 실행하려는 경우 브라우저가 CSSOM을 다운로드하고 빌드하는 작업을 완료하지 않았으면 어떻게 될까요? 답은 간단하지만 성능에는 그다지 좋지 않습니다. **브라우저가 CSSOM을 다운로드하고 생성하는 작업을 완료할 때까지 스크립트 실행 및 DOM 생성을 지연시킵니다.**

간단히 말해서, 자바스크립트에서는 DOM, CSSOM 및 자바스크립트 실행 간에 여러 가지 새로운 종속성을 도입합니다. 이 때문에 브라우저가 화면에서 페이지를 처리하고 렌더링할 때 상당한 지연이 발생할 수 있습니다.

- 문서에서 스크립트의 위치는 중요합니다.
- 브라우저가 스크립트 태그를 만나면 이 스크립트가 실행 종료될 때까지 DOM 생성이 일시 중지됩니다.
- 자바스크립트는 DOM 및 CSSOM을 쿼리하고 수정할 수 있습니다.
- 자바스크립트 실행은 CSSOM이 준비될 때까지 일시 중지됩니다.

일반적으로 '주요 렌더링 경로 최적화'란 HTML, CSS 및 자바스크립트 간의 종속성 그래프를 이해하고 최적화하는 것을 말합니다.

파서 차단 대 비동기 자바스크립트

기본적으로, 자바스크립트 실행은 '파서를 차단'합니다. 브라우저가 문서 내에서 스크립트를 만나면 DOM 생성을 중지시키고, 자바스크립트 런타임에 제어 권한을 넘겨 스크립트가 실행되도록 한 후 DOM 생성을 계속합니다. 우리는 앞의 예시를 통해 인라인 스크립트에서 이러한 동작이 수행되는 것을 확인했습니다. 실제로, 실행을 지연시킬 추가적인 코드를 작성하지 않는 한 인라인 스크립트는 항상 파서를 차단합니다.

스크립트 태그를 통해 포함된 스크립트는 어떨까요? 이전 예시의 코드를 별도의 파일로 추출해 보도록 하겠습니다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script External</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js"></script>
  </body>
</html>
```

app.js

```
var span = document.getElementsByTagName('span')[0];
span.textContent = 'interactive'; // change DOM text content
span.style.display = 'inline';  // change CSSOM property
// create a new element, style it, and append it to the DOM
var loadTime = document.createElement('div');
loadTime.textContent = 'You loaded this page on: ' + new Date();
loadTime.style.color = 'blue';
document.body.appendChild(loadTime);
```

체험해 보기

저희가 태그를 사용하든 인라인 자바스크립트 스니펫을 사용하든 간에 여러분은 이 둘이 동일한 방식으로 동작할 것으로 기대합니다. 두 경우 모두 브라우저가 일시 중지하고 스크립트를 실행해야만 문서의 나머지 부분을 처리할 수 있습니다. 하지만, **외부 자바스크립트 파일의 경우 브라우저가 일시 중지하고 디스크, 캐시 또는 원격 서버에서 스크립트를 가져올 때까지 기다려야 합니다. 이로 인해 주요 렌더링 경로에 수십~수천 밀리초의 지연이 추가로 발생할 수 있습니다.**

기본적으로 모든 자바스크립트는 파서를 차단합니다. 브라우저는 스크립트가 페이지에서 무엇을 수행할지 모르기 때문에, 브라우저는 최악의 시나리오를 가정하고 파서를 차단합니다. 스크립트가 참조되는 바로 그 지점에서 이 스크립트를 실행할 필요가 없음을 브라우저에 신호로 알려준다면, 브라우저가 계속해서 DOM을 구성할 수 있고 준비가 끝난 후에 스크립트를 실행할 수 있습니다(예: 파일을 캐시나 원격 서버에서 가져온 후에 스크립트를 실행).

이를 위해 저희는 이 스트립트를 *async* 로 표시합니다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path: Script Async</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js" async></script>
  </body>
</html>
```

주요 렌더링 경로 측정bookmark_border

모든 탄탄한 성능 전략은 훌륭한 측정과 계측을 바탕으로 합니다. 측정할 수 없는 사항을 최적화할 수 없습니다. 이 문서에서는 다른 접근방식의 CRP 성능 측정에 대해 설명합니다.

- Lighthouse 접근방식에서는 일련의 자동화된 테스트를 페이지에 실행한 다음, 이 페이지의 CRP 성능에 대한 보고서를 생성합니다. 이 접근방식은 브라우저에 로드된 특정 페이지의 CRP 성능을 쉽고 빠르게 측정해 주며, 신속하게 테스트를 수행하고 반복하여 성능을 개선해 줍니다.
- Navigation Timing API 접근방식에서는 RUM(Real User Monitoring) 지표를 캡처합니다. 이름에서 알 수 있듯이, 이 지표는 실제 사용자의 사이트 상호작용으로부터 캡처되며, 다양한 기기와 네트워크 조건에서 사용자가 경험하는 실제 CRP 성능을 정확하게 보여줍니다.

일반적인 좋은 접근방식은 Lighthouse를 사용하여 CRP 최적화의 명확한 기회를 파악하는 것입니다. 그런 다음, Navigation Timing API로 코드를 작성하여 앱의 실제 성능을 모니터링합니다.

Lighthouse로 페이지 감사

Lighthouse는 웹 앱 감사 도구이며 해당 페이지에 대해 일련의 테스트를 수행한 다음, 이 페이지의 결과를 통합된 보고서로 표시해줍니다. Lighthouse를 Chrome 확장 프로그램이나 NPM 모듈로서 실행할 수 있으며, 이는 Lighthouse와 지속적 통합 시스템을 통합하는 데 유용합니다.

시작하려면 Lighthouse로 웹 앱 페이지 감사를 참조하세요.

Lighthouse를 Chrome 확장 프로그램으로 실행하는 경우, 페이지의 CRP 결과는 아래 스크린샷과 같이 보입니다.

- **Performance:** Critical Request Chains 11

  Longest request chain (shorter is better): ***2***

  Longest chain duration (shorter is better): ***5875.38ms***

  Longest chain transfer size (smaller is better): ***114.83KB***

  Initial navigation

  - *chrome-devtools/ (developers.google.com)*
    - */css (fonts.googleapis.com) -* ***989.35ms, 73.38KB***
    - *css/devsite-googler-buttons.css (developers.google.com) -* ***1147.65ms, 71.46KB***
    - *jsi18n/ (developers.google.com) -* ***1176.68ms, 71.64KB***
    - *css/devsite-google-blue.css (developers.google.com) -* ***3066.29ms, 86.42KB***
    - *2.2.0/jquery.min.js (ajax.googleapis.com) -* ***3073.02ms, 100.50KB***
    - *images/devicemode.png (developers.google.com) -* ***4152.05ms, 99.75KB***
    - *images/elements.png (developers.google.com) -* ***5031.06ms, 120.76KB***
    - *v18/2fcrYFNaTjcS6g4U3t-Y5UEw0lE80llgEseQY3FEmqw.woff2 (fonts.gstatic.cor*
    - *js/jquery_ui-bundle.js (developers.google.com) -* ***5334.16ms, 134.75KB***
    - *js/script_foot.js (developers.google.com) -* ***5611.67ms, 157.08KB***
    - *js/script_foot_closure.js (developers.google.com) -* ***5875.38ms, 185.17KB***

- **Performance:** Critical Request Chains 11

  Longest request chain (shorter is better): ***2***

  Longest chain duration (shorter is better): ***5875.38ms***

  Longest chain transfer size (smaller is better): ***114.83KB***

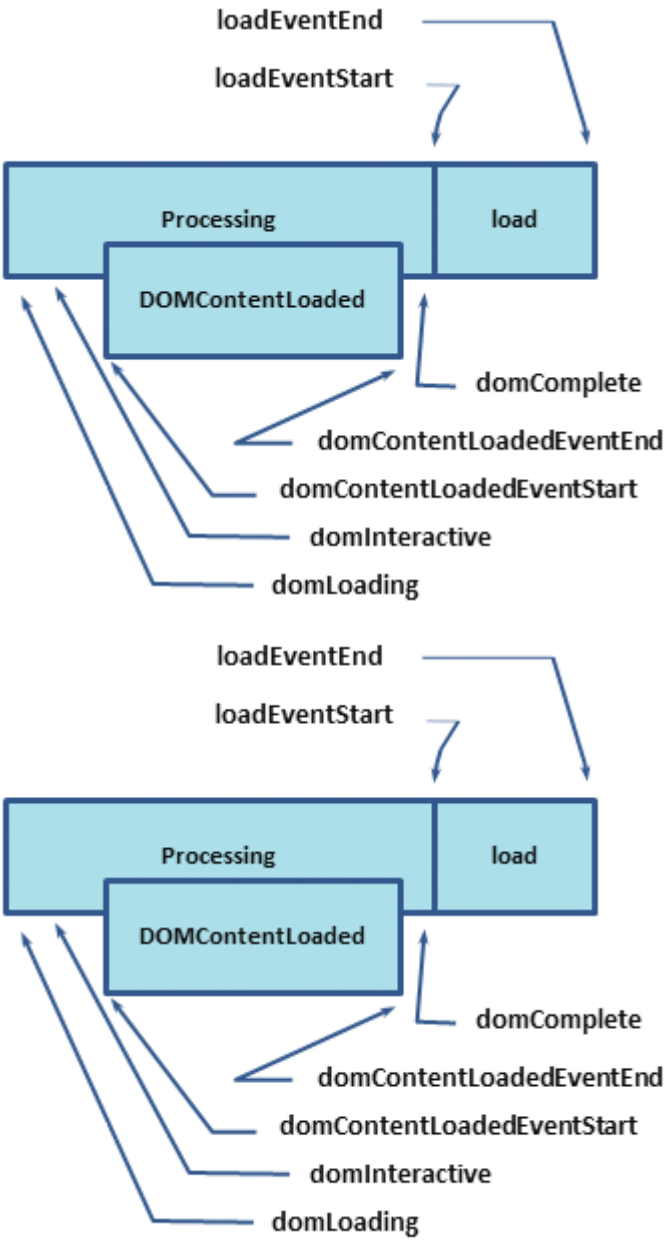
  Initial navigation

  - *chrome-devtools/ (developers.google.com)*
    - */css (fonts.googleapis.com) -* ***989.35ms, 73.38KB***
    - *css/devsite-googler-buttons.css (developers.google.com) -* ***1147.65ms, 71.46KB***
    - *jsi18n/ (developers.google.com) -* ***1176.68ms, 71.64KB***
    - *css/devsite-google-blue.css (developers.google.com) -* ***3066.29ms, 86.42KB***
    - *2.2.0/jquery.min.js (ajax.googleapis.com) -* ***3073.02ms, 100.50KB***
    - *images/devicemode.png (developers.google.com) -* ***4152.05ms, 99.75KB***
    - *images/elements.png (developers.google.com) -* ***5031.06ms, 120.76KB***
    - *v18/2fcrYFNaTjcS6g4U3t-Y5UEw0lE80llgEseQY3FEmqw.woff2 (fonts.gstatic.cor*
    - *js/jquery_ui-bundle.js (developers.google.com) -* ***5334.16ms, 134.75KB***
    - *js/script_foot.js (developers.google.com) -* ***5611.67ms, 157.08KB***
    - *js/script_foot_closure.js (developers.google.com) -* ***5875.38ms, 185.17KB***

이 감사의 결과에 대한 자세한 내용은 주요 요청 체인을 참조하세요.

Navigation Timing API로 코드 작성

Navigation Timing API와 기타 여러 브라우저 이벤트를 조합해서 사용하여 임의 페이지의 실제 CRP 성능을 캡처하고 기록할 수 있습니다.



위 다이어그램의 각 레이블은 로드되는 각각의 모든 페이지에 대해 브라우저가 추적하는 고해상도 타임스탬프에 해당합니다. 사실상, 이 특정 경우에서는 다양한 모든 타임스탬프 중 일부만 보여줍니다. 지금은 모든 네트워크 관련 타임스탬프를 건너뛰지만 이후 과정에서 이에 대해 다시 살펴볼 것입니다.

그렇다면 이러한 타임스탬프가 의미하는 바는 무엇일까요?

- `domLoading`: 전체 프로세스의 시작 타임스탬프입니다. 브라우저가 처음 수신한 HTML 문서 바이트의 파싱을 시작하려고 합니다.
- `domInteractive`: 브라우저가 파싱을 완료한 시점을 표시합니다. 모든 HTML 및 DOM 생성 작업이 완료되었습니다.
- `domContentLoaded`

  : DOM이 준비되고 자바스크립트 실행을 차단하는 스타일시트가 없는 시점을 표시합니다. 즉, 이제 (잠재적으로) 렌더링 트리를 생성할 수 있습니다.
  - 많은 자바스크립트 프레임워크가 자체 로직을 실행하기 전에 이 이벤트를 기다립니다. 이러한 이유로 브라우저는 `EventStart` 및 `EventEnd` 타임스탬프를 캡처합니다. 이를 통해 이 실행이 얼마나 오래 걸렸는지 추적할 수 있습니다.
- `domComplete`: 이름이 의미하는 바와 같이, 모든 처리가 완료되고 페이지의 모든 리소스(이미지 등) 다운로드가 완료되었습니다(예: 로딩 스피너가 회전을 멈춤).

- `loadEvent`: 각 페이지 로드의 최종 단계로, 브라우저가 추가 애플리케이션 로직을 트리거할 수 있는 `onload` 이벤트를 발생시킵니다.

HTML 사양은 이벤트가 발생하는 시기, 충족해야 하는 조건 등 각 이벤트에 대한 특정 조건을 규정합니다. 여기서는 주요 렌더링 경로와 관련된 몇 가지 주요 마일스톤을 중점적으로 살펴보겠습니다.

- `domInteractive`는 DOM이 준비된 시점을 표시합니다.
- `domContentLoaded`

  는 일반적으로

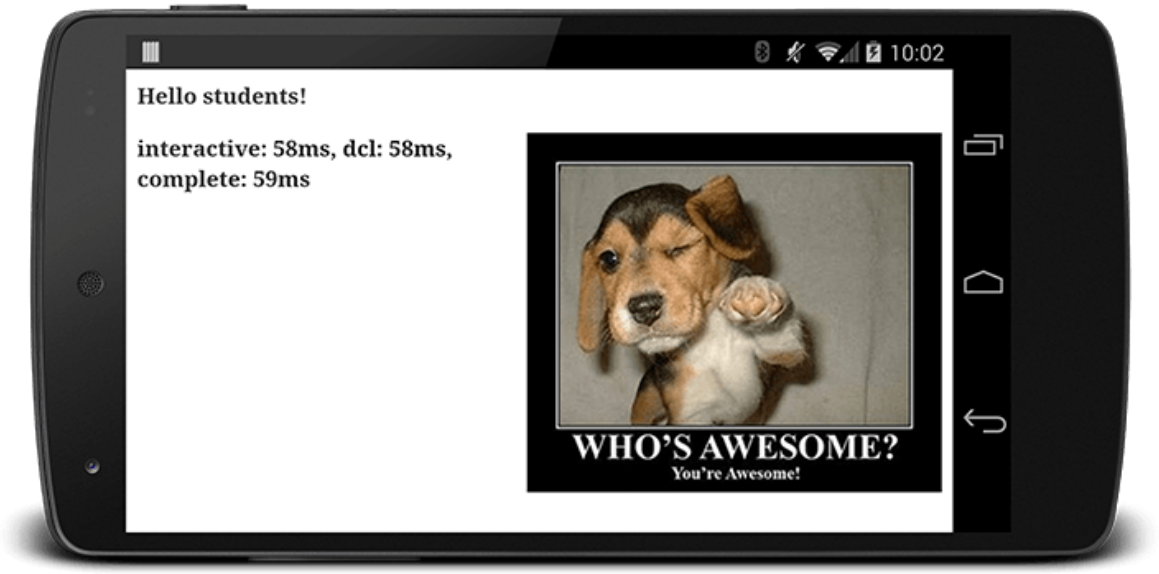
  DOM 및 CSSOM이 모두 준비

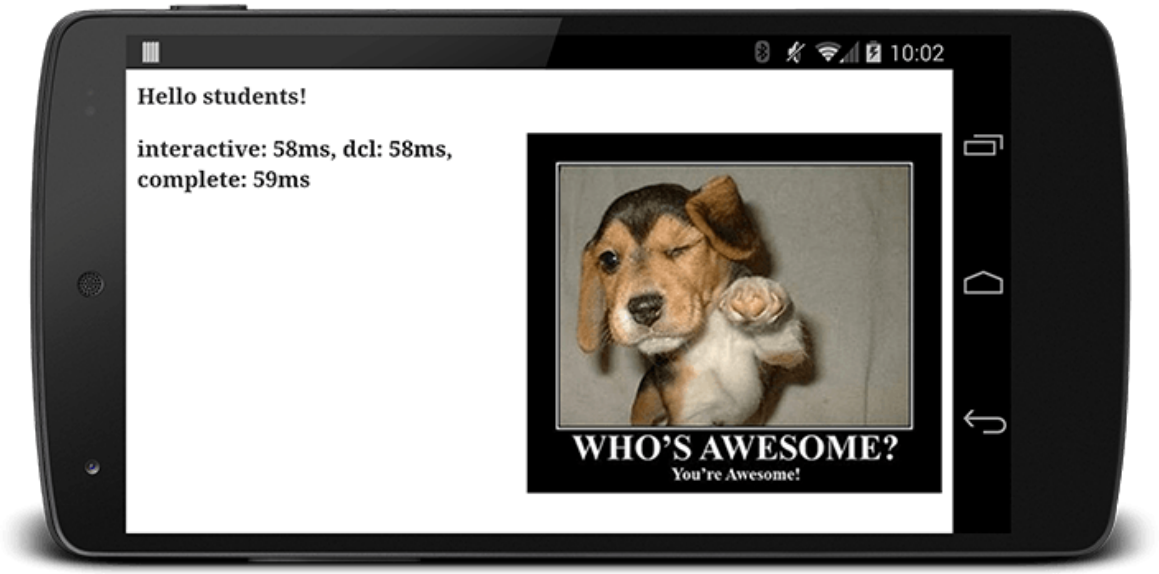
  된 시점을 표시합니다.
  - 파서 차단 자바스크립트가 없으면 `domInteractive` 직후에 `DOMContentLoaded`가 발생할 것입니다.
- `domComplete`는 페이지 및 해당 하위 리소스가 모두 준비된 시점을 표시합니다.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Critical Path: Measure</title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <script>
      function measureCRP() {
        var t = window.performance.timing,
          interactive = t.domInteractive - t.domLoading,
          dcl = t.domContentLoadedEventStart - t.domLoading,
          complete = t.domComplete - t.domLoading;
        var stats = document.createElement('p');
        stats.textContent = 'interactive: ' + interactive + 'ms, ' +
            'dcl: ' + dcl + 'ms, complete: ' + complete + 'ms';
        document.body.appendChild(stats);
      }
    </script>
  </head>
  <body onload="measureCRP()">
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

체험해 보기

위 예시는 처음 보면 조금 복잡해 보일 수 있지만 사실은 매우 간단합니다. Navigation Timing API는 모든 관련 타임스탬프를 캡처하고 개발자 코드는 단순히 `onload` 이벤트가 발생하기를 기다립니다. `onload` 이벤트는 `domInteractive`, `domContentLoaded` 및 `domComplete` 이후에 발생한다는 사실을 상기하세요. 또한 이 API는 다양한 타임스탬프 간의 차이를 계산합니다.





모든 것이 말한대로 이루어졌으면, 이제 측정할 몇 가지 특정한 마일스톤과 이러한 측정 결과를 출력하기 위한 간단한 함수가 만들어졌을 것입니다. 페이지에 이러한 메트릭을 출력하는 대신 분석 서버로 전송하도록 코드를 수정할 수도 있습니다(Google 애널리틱스에서는 이 작업을 자동으로 수행함). 이는 페이지의 성능을 감시하고 몇 가지 최적화 작업을 통해 성능을 높일 수 있는 페이지를 식별하기 위한 훌륭한 방법입니다.

DevTools 소개

이 문서에서는 CRP 개념을 설명하기 위해 Chrome DevTools Network 패널을 사용하기도 하지만 현재는 DevTools가 CRP 분석에 잘 맞지 않습니다. 그 이유는 DevTools에는 주요 리소스를 분리하기 위한 내장 메커니즘이 없기 때문입니다. 이러한 리소스를 식별하도록 도우려면 Lighthouse 감사를 실행하세요.

주요 렌더링 경로 성능 분석 bookmark_border

주요 렌더링 경로 성능 병목 현상을 식별하고 해결하려면 흔히 있는 함정들에 대해 잘 파악하고 있어야 합니다. 실습 과정을 통해 페이지를 최적화하는 데 도움이 되는 일반적인 성능 패턴을 알아보도록 하겠습니다.

주요 렌더링 경로를 최적화하게 되면 브라우저가 가능한 한 빨리 페이지를 그릴 수 있습니다. 더 빠른 페이지는 더 높은 몰입도, 더 많은 페이지 조회 수 및 전환율 향상으로 이어집니다. 방문자가 빈 화면에서 보내는 시간을 최소화하기 위해 어떤 리소스를 어떤 순서로 로드할지 최적화해야 합니다.

이 프로세스를 설명하는 데 도움이 되도록 가능한 가장 간단한 사례부터 시작하고 점차적으로 페이지를 확대하여 추가 리소스, 스타일 및 애플리케이션 로직을 포함해 보도록 하겠습니다. 이 프로세스에서 우리는 각 사례를 최적화하고 어떤 부분이 잘못될 수 있는지도 살펴볼 것입니다.

지금까지 우리는 처리할 리소스(CSS, JS 또는 HTML 파일)가 준비가 되면 브라우저에 어떤 일이 일어나는지에 대해서만 초점을 맞췄습니다. 캐시나 네트워크에서 리소스를 가져오는 데 걸리는 시간을 무시했습니다. 우리는 다음 사항을 가정할 것입니다.

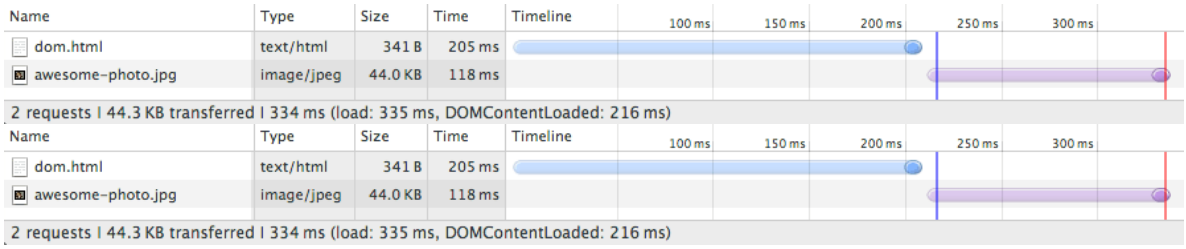
- 서버에 대한 네트워크 왕복 시간(전파 지연 시간)은 100ms입니다.
- 서버 응답 시간은 HTML 문서의 경우 100ms이고 기타 모든 파일의 경우 10ms입니다.

Hello World 체험

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Critical Path: No Style</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

체험해 보기

기본적인 HTML 마크업과 하나의 이미지로 시작해 보겠습니다. CSS 또는 자바스크립트는 포함하지 않습니다. Chrome DevTools에서 네트워크 타임라인을 열고 결과로 나타나는 리소스 워터폴(waterfall)을 검토합니다.



참고: 이 문서에서는 CRP 개념을 설명하기 위해 DevTools를 사용하지만 현재는 DevTools가 CRP 분석에 잘 맞지 않습니다. 자세한 내용은 DevTools 소개를 참조하세요.

예상대로, HTML 파일의 다운로드 시간은 약 200ms가 걸렸습니다. 파란색 선의 투명한 부분은 브라우저가 응답 바이트를 수신하지 않고 네트워크에서 대기하는 시간을 나타내는 반면, 진한 부분은 첫 응답 바이트가 수신된 후에 다운로드가 완료된 시간을 보여줍니다. HTML 다운로드 크기는 매우 작기 때문에 (4K 미만) 전체 파일을 가져오기 위해서는 한 번의 왕복만 필요합니다. 따라서 HTML 문서를 가져오려면 약 200ms가 걸립니다. 이 시간 중 절반은 네트워크에서 대기하는 데 사용되고 절반은 서버 응답에서 대기하는 데 사용됩니다.

HTML 콘텐츠를 사용할 수 있게 되면 브라우저가 바이트를 파싱하고 토큰으로 변환한 후 DOM 트리를 빌드합니다. DevTools는 개발자가 편하게 볼 수 있도록 DOMContentLoaded 이벤트 시간(216ms)을 맨 아래에 표시합니다. 이는 파란색 수직선에 해당합니다. HTML 다운로드 완료 시점과 파란색 수직선(DOMContentLoaded) 사이의 격차는 브라우저가 DOM 트리를 빌드하는 데 소요된 시간을 나타냅니다. 이 경우에는 몇 밀리초만 걸립니다.

'awesome photo'가 `domContentLoaded` 이벤트를 차단하지 않았다는 점에 주목하세요. 이는 페이지의 각 자산을 기다릴 필요 없이 렌더링 트리를 생성하고 페이지를 그릴 수 있음을 의미합니다. **일부 리소스는 페이지를 신속하게 처음 그리는 데 중요하지는 않습니다**. 실제로, 우리가 주요 렌더링 경로에 대해 이야기할 때 일반적으로 HTML 마크업, CSS 및 자바스크립트에 대해 언급합니다. 이미지는 페이지의 초기 렌더링을 차단하지 않습니다. 물론 이미지를 가능한 한 빨리 그리도록 해야 합니다.

하지만 `load` 이벤트(`onload`라고도 불림)는 이미지에서 차단됩니다. DevTools는 335ms에 `onload` 이벤트를 보고합니다. `onload` 이벤트는 페이지에 필요한 **모든 리소스**가 다운로드되고 처리되는 시점을 표시한다는 것을 기억하세요. 이 시점에서 로딩 스피너가 브라우저에서 회전을 멈출 수 있습니다(워터폴에서 빨간색 수직선).

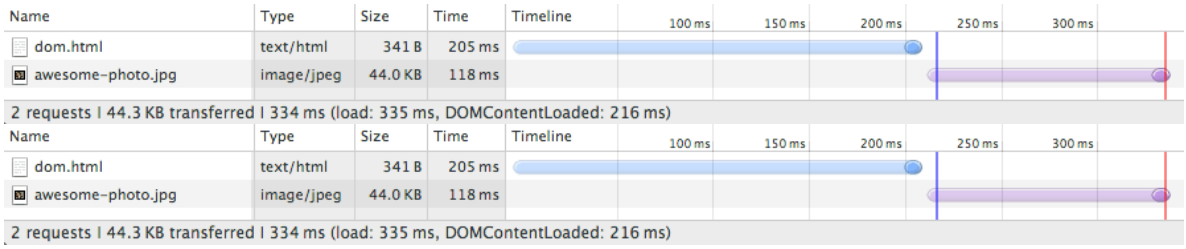
자바스크립트 및 CSS를 함께 추가

'Hello World 체험' 페이지는 단순해 보이지만 자세히 들여다보면 많은 일들이 벌어지고 있습니다. 실제로는 HTML 외에 다른 것도 필요합니다. 아마 페이지에 일부 상호작용을 추가하기 위해 CSS 스타일시트와 하나 이상의 스크립트가 필요할 것입니다. 이 두 가지를 모두 추가해보고 어떤 일이 일어나는지 확인해봅시다.

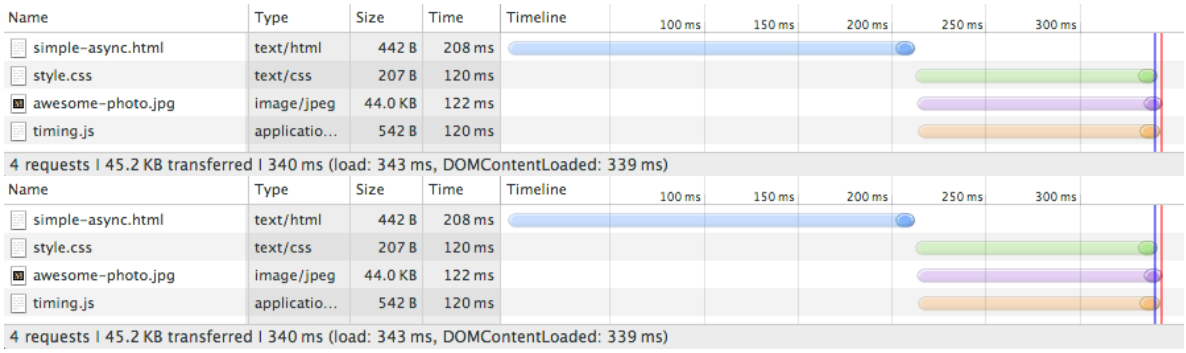
```
<!DOCTYPE html>
<html>
  <head>
    <title>Critical Path: Measure Script</title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
  </head>
  <body onload="measureCRP()">
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="timing.js"></script>
  </body>
</html>
```

체험해 보기

자바스크립트 및 CSS 추가 전:



자바스크립트 및 CSS 추가 후:



외부 CSS 및 자바스크립트 파일을 추가하면 위의 워터폴에 두 개의 요청이 더 추가됩니다. 두 개 모두 브라우저가 거의 같은 시간에 발송합니다. 하지만, `domContentLoaded`와 `onload` **이벤트 간에 훨씬 작은 시간 차이가 있습니다.**

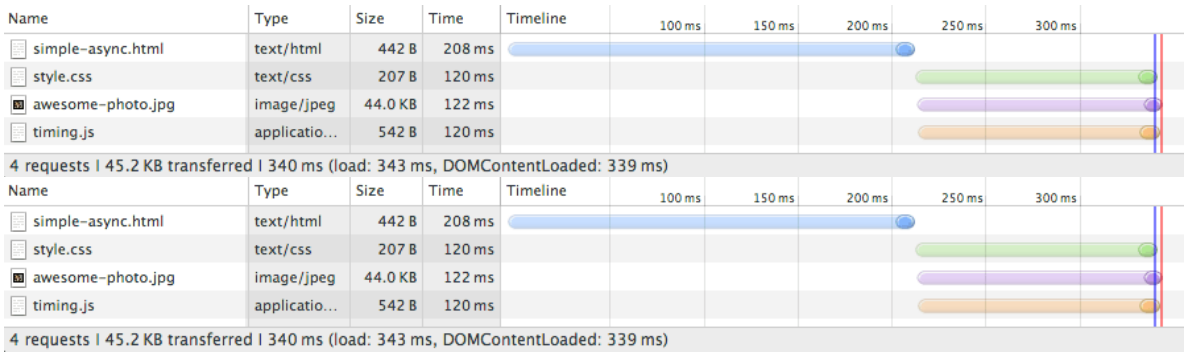
무슨 일이 일어난 걸까요?

- 일반 HTML 예시와 달리 CSSOM을 생성하기 위해 CSS 파일도 가져오고 파싱해야 하며, 렌더링 트리를 빌드하기 위해 DOM과 CSSOM이 모두 필요합니다.
- 페이지에는 또한 파서 차단 자바스크립트 파일이 포함되기 때문에, CSS 파일이 다운로드되어 파싱될 때까지 `domContentLoaded` 이벤트가 차단됩니다. 자바스크립트가 CSSOM을 쿼리할 수도 있기 때문에, 자바스크립트를 실행하기 전에 CSS 파일이 다운로드될 때까지 차단해야 합니다.

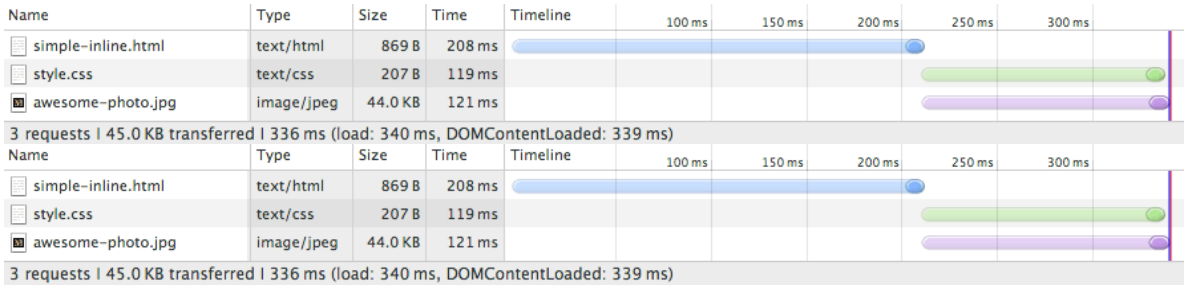
외부 스크립트를 인라인 스크립트로 바꾸면 어떻게 될까요? 스크립트가 페이지에 바로 인라인 처리되더라도, CSSOM이 생성될 때까지는 브라우저가 이 스크립트를 실행할 수 없습니다. 간단히 말해서, 인라인 자바스크립트도 파서를 차단합니다.

CSS를 차단하더라도 스크립트를 인라인 처리하면 페이지 렌더링 속도가 빨라질까요? 같이 살펴보고 어떠한 일이 일어나는지 확인해봅시다.

외부 자바스크립트:



인라인 자바스크립트:



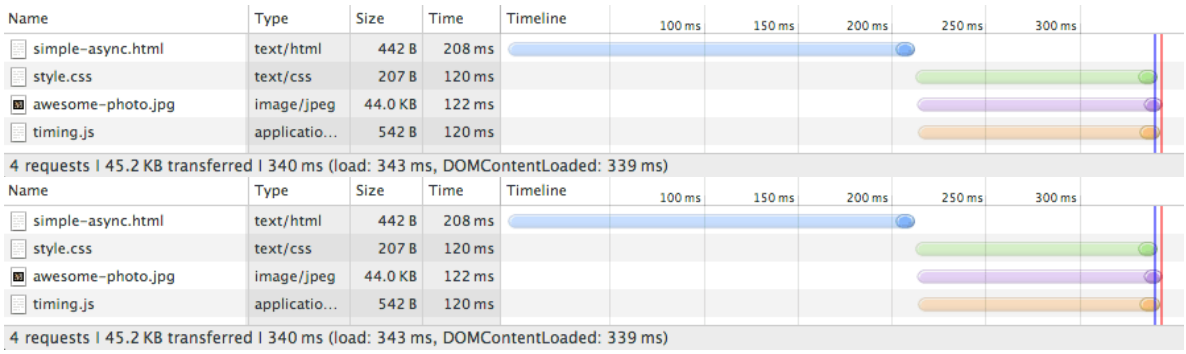
요청을 하나 더 적게 수행하지만 `onload` 및 `domContentLoaded` 시간은 거의 같습니다. 그 이유는 무엇일까요? 그 이유는 자바스크립트가 인라인이든 외부이든 상관이 없기 때문입니다. 브라우저는 스크립트 태그를 만나자마자 스크립트 실행을 차단하고 CSSOM이 생성될 때까지 대기합니다. 또한 첫 번째 예시에서 브라우저가 CSS 및 자바스크립트를 동시에 다운로드하고 거의 동일한 시간에 다운로드를 완료합니다. 이 경우 자바스크립트 코드를 인라인 처리해도 그다지 도움이 되지 못합니다. 그러나 페이지 렌더링 속도를 높일 수 있는 여러 가지 전략이 있습니다.

첫째, 모든 인라인 스크립트가 파서를 차단하지만 외부 스크립트의 경우 'async' 키워드를 추가하여 파서의 차단을 해제할 수 있다는 점을 기억하세요. 인라인 처리를 취소하고 다음과 같이 해봅시다.

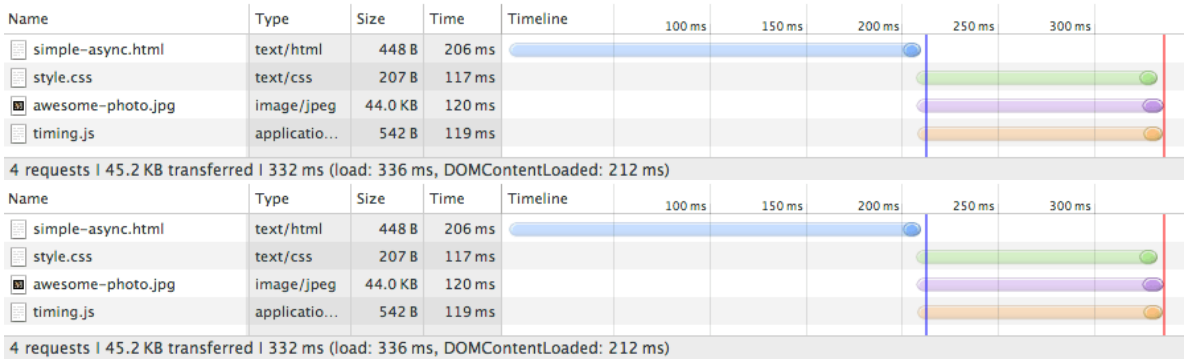
```
<!DOCTYPE html>
<html>
  <head>
    <title>Critical Path: Measure Async</title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
  </head>
  <body onload="measureCRP()">
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script async src="timing.js"></script>
  </body>
</html>
```

체험해 보기

파서 차단(외부) 자바스크립트:



비동기(외부) 자바스크립트:



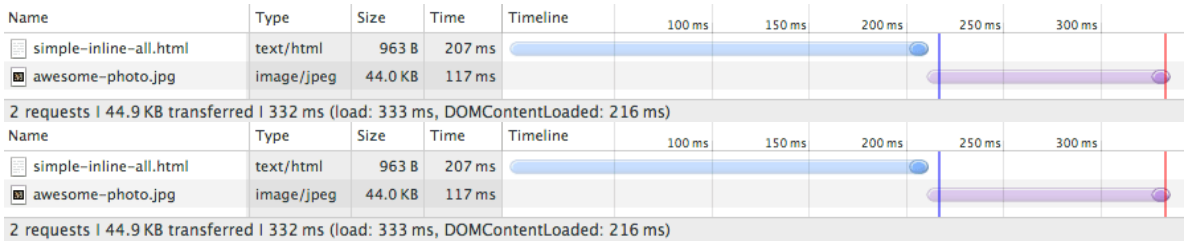
훨씬 낫네요! `domContentLoaded` 이벤트는 HTML이 파싱된 후 바로 실행됩니다. 브라우저가 자바스크립트를 차단하지 않는다는 것을 알고 있고 다른 파서 차단 스크립트가 없으므로 CSSOM 생성 또한 동시에 처리될 수 있습니다.

또는 CSS와 자바스크립트를 모두 인라인 처리할 수도 있습니다.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Critical Path: Measure Inlined</title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <style>
      p { font-weight: bold }
      span { color: red }
```

```
      p span { display: none }
      img { float: right }
    </style>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script>
      var span = document.getElementsByTagName('span')[0];
      span.textContent = 'interactive'; // change DOM text content
      span.style.display = 'inline';  // change CSSOM property
      // create a new element, style it, and append it to the DOM
      var loadTime = document.createElement('div');
      loadTime.textContent = 'You loaded this page on: ' + new Date();
      loadTime.style.color = 'blue';
      document.body.appendChild(loadTime);
    </script>
  </body>
</html>
```

체험해 보기



`domContentLoaded` 시간은 이전의 예시와 거의 비슷합니다. 자바스크립트를 비동기로 표시하는 대신 CSS와 JS를 모두 페이지 내에 인라인으로 추가했습니다. 이로 인해 HTML 페이지가 더 커지지만, 장점은 페이지 안에 필요한 모든 요소가 있기 때문에 브라우저가 외부 리소스를 가져올 때까지 기다릴 필요가 없다는 점입니다.

이처럼, 아주 단순한 페이지더라도 주요 렌더링 경로를 최적화하는 것은 사소한 문제가 아닙니다. 서로 다른 리소스 간의 의존성 그래프를 파악해야 하며, 어떤 리소스가 '중요'한지 식별해야 하고, 이러한 리소스를 페이지에 포함할 방법에 대한 다양한 전략 중에서 선택해야 합니다. 이 문제를 해결할 수 있는 방법이 한 가지만 있는 것은 아닙니다. 각 페이지가 서로 다르기 때문에 자신만의 유사한 프로세스에 따라 최적의 전략을 찾아야 합니다.

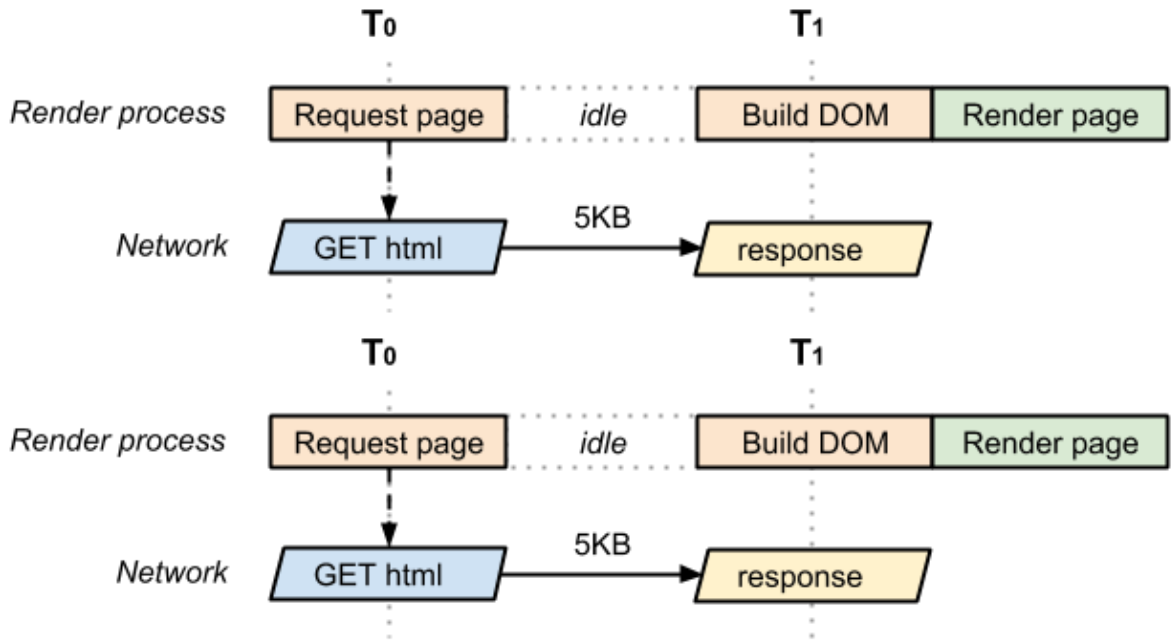
이제 위의 과정에서 몇 가지 일반적인 성능 패턴을 찾을 수 있는지 살펴봅시다.

성능 패턴

가장 간단한 페이지는 CSS, 자바스크립트 및 기타 유형의 리소스 없이 HTML 마크업으로만 이루어져 있습니다. 이 페이지를 렌더링하려면 브라우저가 요청을 시작하고, HTML 문서가 도착할 때까지 기다리고, 해당 문서를 파싱하고, DOM을 빌드한 후 최종적으로 화면에 렌더링해야 합니다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Critical Path: No Style</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

체험해 보기

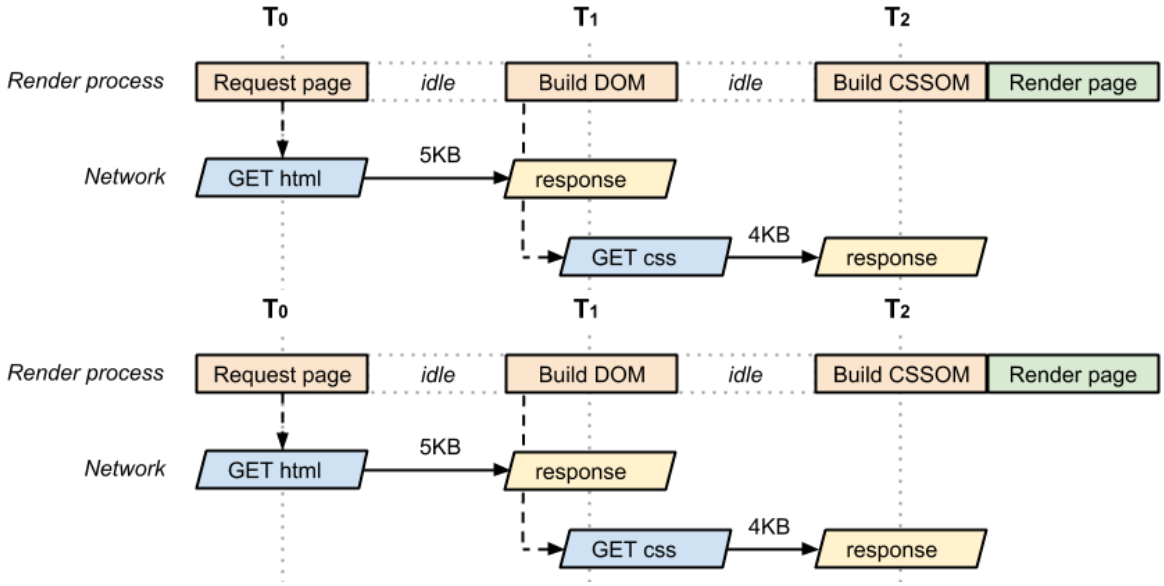


T0와 T1 사이의 시간은 네트워크 및 서버 처리 시간을 나타냅니다. 최상의 경우(HTML 파일이 작을 경우) 한 번의 네트워크 왕복만으로 전체 문서를 가져옵니다. TCP 전송 프로토콜의 작동 방식으로 인해 큰 파일은 더 많은 왕복이 필요할 수 있습니다. **결과적으로, 최상의 경우 위 페이지는 (최소) 1회 왕복의 주요 렌더링 경로를 갖게 됩니다.**

이제 외부 CSS 파일이 추가된 동일한 페이지를 살펴보도록 하겠습니다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

체험해 보기

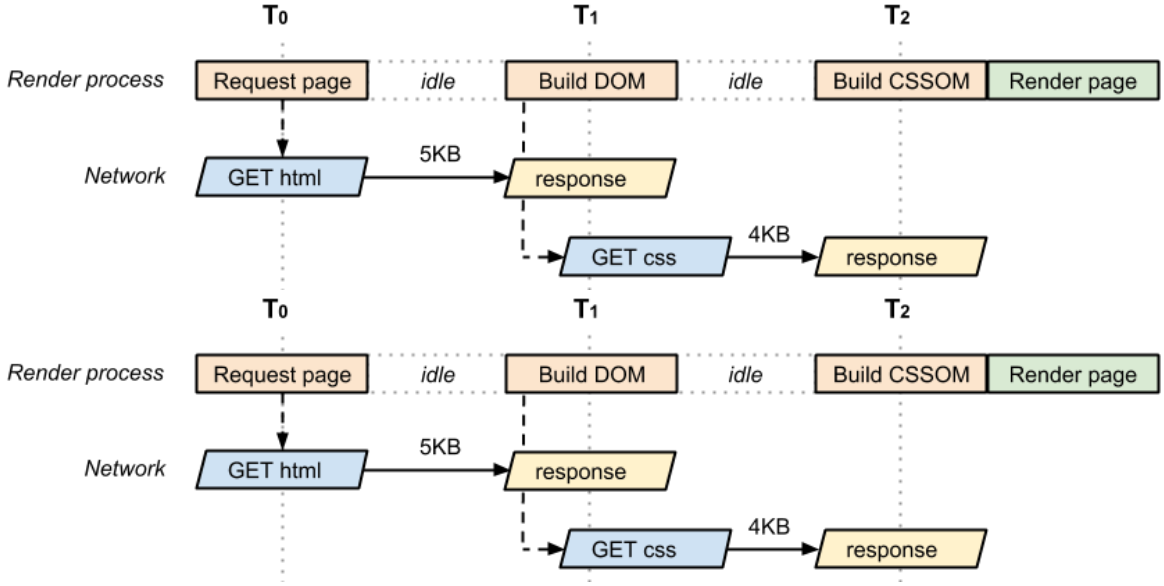


다시 한 번, HTML 문서를 가져오기 위한 네트워크 왕복을 발생시킵니다. 그러면 가져온 마크업이 CSS 파일도 필요하다고 알려줍니다. 즉, 브라우저가 화면에 페이지를 렌더링하기 전에 서버로 돌아가서 CSS를 가져와야 합니다. **따라서, 이 페이지는 표시되기 전에 최소 두 번의 왕복이 발생합니다.** 다시 말하자면 CSS 파일은 여러 번의 왕복이 필요할 수 있으므로 '최소'라는 표현을 썼습니다.

주요 렌더링 경로를 설명하기 위해 우리가 사용할 용어에 대한 정의를 살펴보겠습니다.

- **주요 리소스:**페이지의 초기 렌더링을 차단할 수 있는 리소스입니다.
- **주요 경로 길이:**왕복 횟수, 또는 모든 주요 리소스를 가져오는 데 필요한 총 시간입니다.
- **주요 바이트:**페이지의 최초 렌더링에 필요한 총 바이트 수로, 모든 주요 리소스에 대한 전송 파일 크기의 합계입니다. 단일 주요 리소스(HTML 문서)가 포함된 단일 HTML 페이지로 구성된 첫 번째 예시에서 주요 경로 길이는 한 번의 네트워크 왕복과 같으며(파일이 작다고 가정했을 때) 총 주요 바이트 수는 HTML 문서의 전송 크기입니다.

이제, 위에 나오는 HTML + CSS 예시의 주요 경로 특성과 비교해봅시다.



- **2**개의 주요 리소스
- **2**번 이상의 왕복(최소 주요 경로 길이)
- **9**KB의 주요 바이트

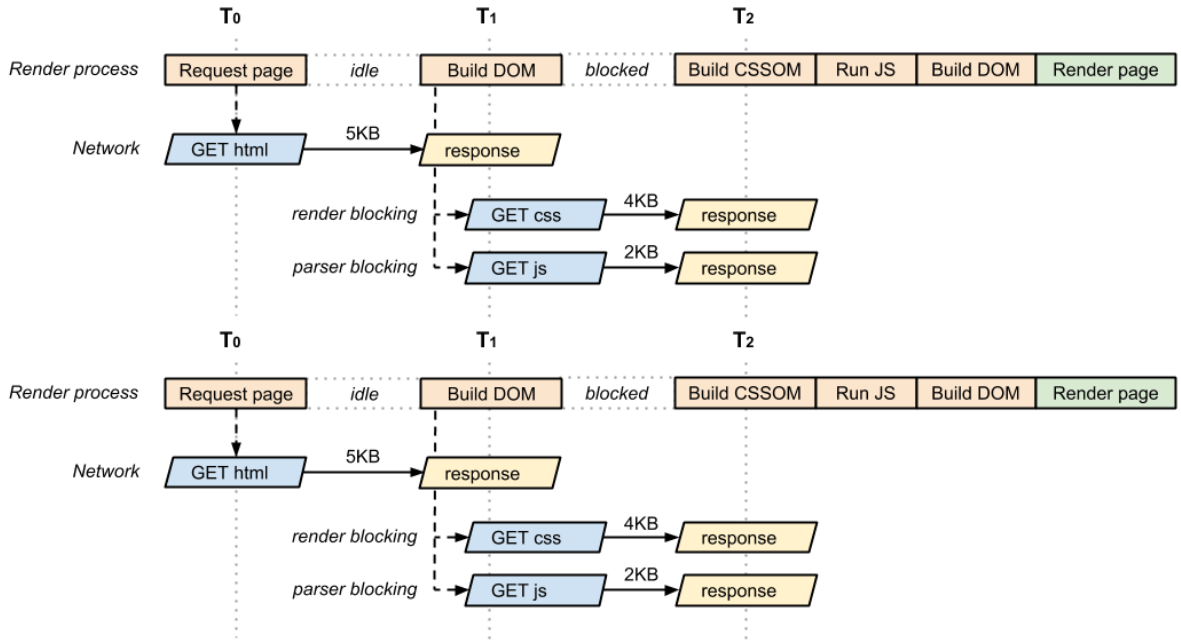
렌더링 트리를 생성하기 위해서는 HTML과 CSS가 모두 필요합니다. 따라서 HTML과 CSS는 모두 주요 리소스입니다. CSS 가져오기는 브라우저가 HTML 문서를 가져온 후에만 수행됩니다. 따라서 주요 경로 길이는 최소 2번의 왕복입니다. 두 리소스 크기의 합은 총 9KB의 주요 바이트입니다.

이제 여기에 자바스크립트 파일을 하나 더 추가해봅시다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js"></script>
  </body>
</html>
```

체험해 보기

우리는 `app.js`를 추가했습니다. 이것은 페이지에서 외부 자바스크립트 자산이자 파서 차단 리소스(즉, 주요 리소스)입니다. 더 안 좋은 경우, 자바스크립트 파일을 실행하기 위해 작업을 차단하고 CSSOM이 처리될 때까지 기다려야 합니다. 자바스크립트가 CSSOM을 처리할 수 있기 때문에 `style.css`가 다운로드되고 CSSOM이 생성될 때까지 브라우저가 일시 중지된다는 것을 기억하세요.



하지만 실제로는 이 페이지의 '네트워크 워터폴'을 보면 CSS와 자바스크립트 요청이 모두 거의 같은 시간에 시작된다는 것을 알 수 있습니다. 브라우저가 HTML을 가져오고 두 리소스를 검색한 후 두 요청을 모두 실행합니다. 결과적으로, 위 페이지는 다음 주요 경로 특성을 갖습니다.

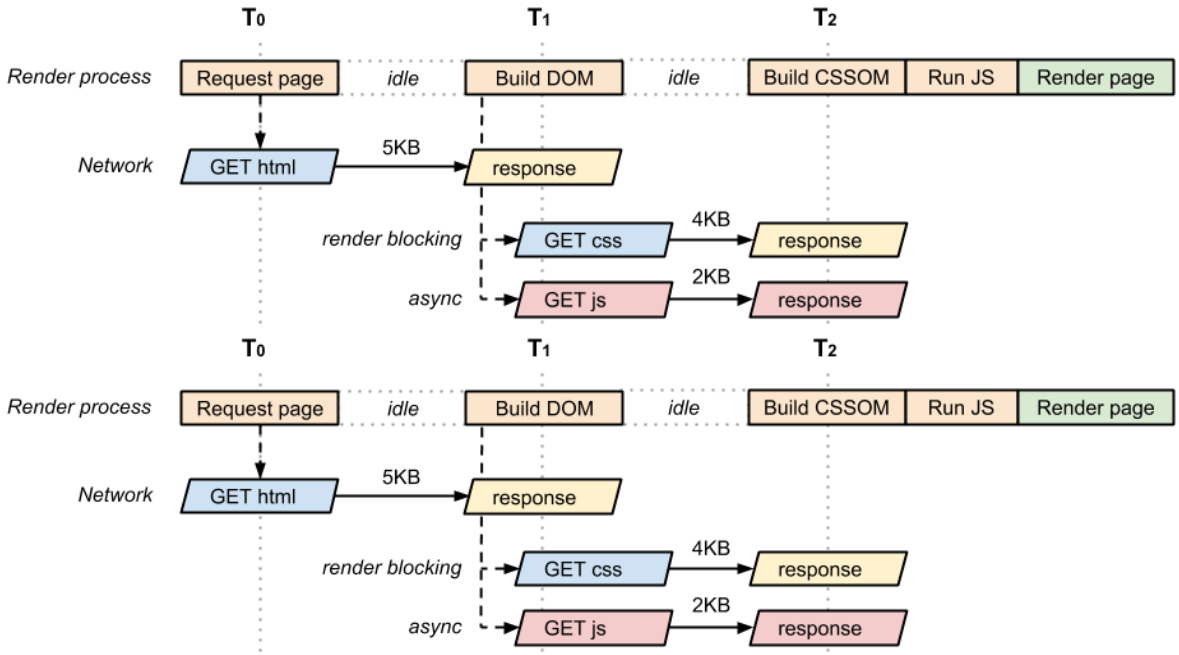
- **3**개의 주요 리소스
- **2**번 이상의 왕복(최소 주요 경로 길이)
- **11**KB의 주요 바이트

이제 총 11KB의 주요 바이트에 해당하는 3개의 주요 리소스를 갖게 되었습니다. 하지만 주요 경로 길이는 여전히 2번 왕복입니다. 그 이유는 CSS와 자바스크립트를 동시에 전송할 수 있기 때문입니다. **주요 렌더링 경로 특성을 파악하면 주요 리소스를 식별할 수 있으며 브라우저가 이에 대한 가져오기 작업을 예약하는 방식을 이해할 수 있습니다.** 다른 예시도 살펴보도록 하겠습니다.

우리는 Google 사이트 개발자와 얘기를 나눈 후 페이지에 포함한 자바스크립트를 차단할 필요가 없다는 사실을 알게 되었습니다. 몇 가지 분석 방법과 페이지 렌더링을 차단할 필요가 없는 다른 코드가 있습니다. 즉, 'async' 속성을 스크립트 태그에 추가하여 파서 차단을 해제할 수 있습니다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js" async></script>
  </body>
</html>
```

체험해 보기



비동기 스트립트는 여러 가지 이점이 있습니다.

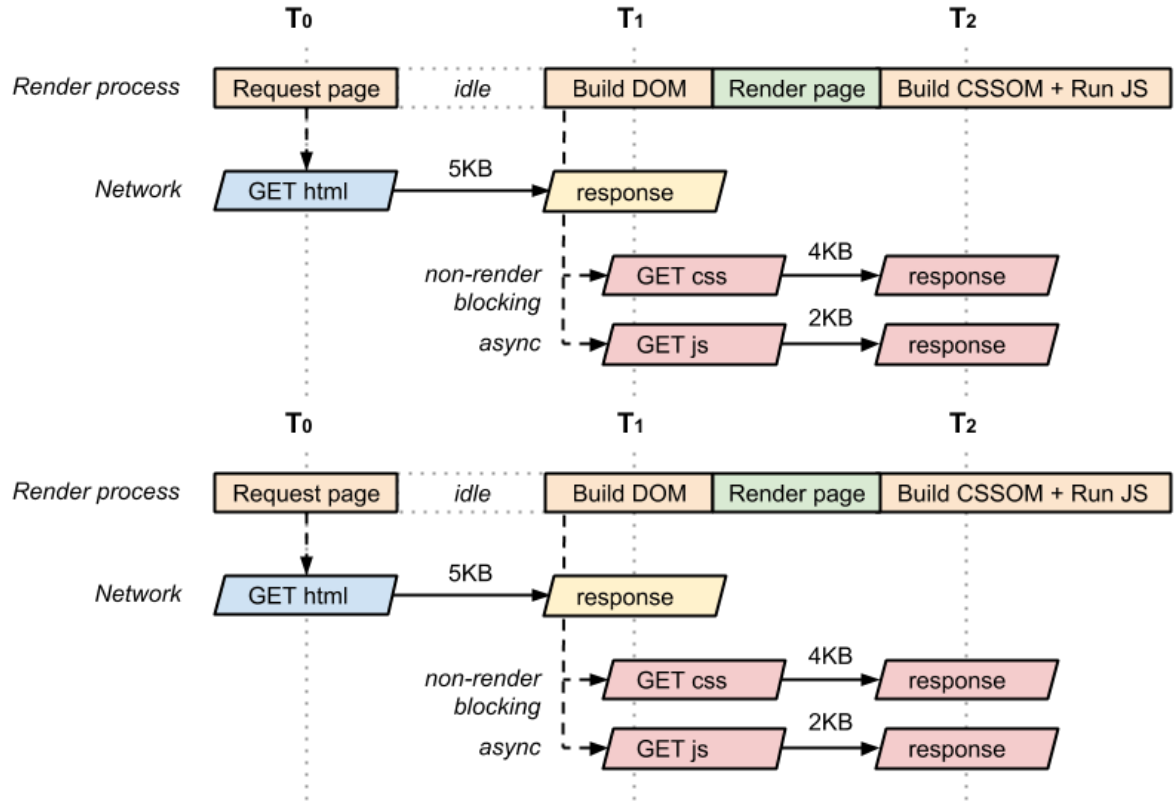
- 스크립트가 더 이상 파서를 차단하지 않고 주요 렌더링 경로에 포함되지 않습니다.
- 주요 스크립트가 없기 때문에 CSS가 `domContentLoaded` 이벤트를 차단할 필요가 없습니다.
- `domContentLoaded` 이벤트가 빨리 실행될수록 다른 애플리케이션 로직도 빨리 실행될 수 있습니다.

그 결과, 최적화된 페이지가 이제 다시 2개의 주요 리소스(HTML 및 CSS), 최소 2번 왕복의 주요 경로 길이, 총 9KB의 주요 바이트를 갖게 됩니다.

마지막으로, CSS 스타일시트가 인쇄에만 필요하다면 어떻게 보일까요?

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet" media="print">
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js" async></script>
  </body>
</html>
```

체험해 보기



style.css 리소스는 인쇄에만 사용되기 때문에 브라우저가 페이지를 렌더링하기 위해 차단할 필요가 없습니다. 따라서, DOM 생성이 완료되자마자 브라우저가 페이지를 렌더링하는 데 충분한 정보를 갖게 됩니다. 그 결과, 이 페이지는 하나의 주요 리소스(HTML 문서)만 가지며, 최소 주요 렌더링 경로 길이는 1회 왕복이 됩니다.

주요 렌더링 경로 최적화bookmark_border

최초 렌더링 시 최대한 빠르게 렌더링하려면 다음 세 가지 변수를 최소화해야 합니다.

- 주요 리소스의 수.
- 주요 경로 길이.
- 주요 바이트의 수.

주요 리소스는 페이지의 초기 렌더링을 차단할 수 있는 리소스입니다. 이러한 리소스가 적을수록 브라우저, CPU 및 기타 리소스의 작업이 줄어듭니다.

마찬가지로, 주요 경로 길이는 주요 리소스와 해당 바이트 크기 간의 종속성 그래프를 나타내는 기능입니다. 일부 리소스 다운로드는 이전 리소스가 처리된 후에만 시작될 수 있으며, 리소스가 클수록 다운로드하는 데 걸리는 왕복 수가 더 많아집니다.

마지막으로, 브라우저에서 다운로드해야 하는 주요 바이트 수가 적을수록 신속하게 콘텐츠를 처리하여 화면에 렌더링할 수 있습니다. 바이트 수를 줄이기 위해 리소스를 제거하거나 중요하지 않은 것으로 만들어 리소스 수를 줄일 수 있으며, 각각의 리소스를 압축하고 최적화하여 전송 크기를 최소화할 수도 있습니다.

주요 렌더링 경로를 최적화하기 위한 일반적인 단계는 다음과 같습니다.

1. 주요 경로(리소스 수, 바이트 수, 길이)를 분석하고 파악합니다.
2. 주요 리소스를 제거하거나 이에 대한 다운로드를 연기하거나 비동기로 표시하는 등의 방법으로 주요 리소스 수를 최소화합니다.
3. 주요 바이트 수를 최적화하여 다운로드 시간(왕복 수)을 단축합니다.
4. 나머지 주요 리소스가 로드되는 순서를 최적화합니다. 주요 경로 길이를 단축하려면 가능한 한 빨리 모든 주요 자산을 다운로드합니다.

PageSpeed 규칙 및 권장 사항 bookmark_border

이 가이드에서는 PageSpeed Insights 규칙에 대해 살펴보고, 주요 렌더링 경로를 최적화할 때 주의해야 할 사항과 그 이유에 대해 알아봅니다.

렌더링 차단 자바스크립트 및 CSS 제거

최초 렌더링을 가장 빠르게 수행하려면 페이지의 주요 리소스 수를 최소화하거나 (가능한 경우) 제거하고, 다운로드되는 주요 바이트 수를 최소화하고, 주요 경로 길이를 최적화해야 합니다.

자바스크립트 사용 최적화

자바스크립트 리소스는 `async`로 표시하거나 특별한 자바스크립트 스니펫을 추가하지 않은 경우 기본적으로 파서를 차단합니다. 파서 차단 자바스크립트는 CSSOM이 처리될 때까지 브라우저를 기다리게 하고 DOM 생성을 일시 중지합니다. 이는 최초 렌더링에 상당한 지연을 일으킬 수 있습니다.

비동기 자바스크립트 리소스 선호

비동기 리소스는 문서 파서의 차단을 해제하고, 브라우저가 스크립트를 실행하기 전에 CSSOM을 차단하지 않도록 합니다. 대개, 스크립트가 `async` 속성을 사용할 수 있다면 이는 해당 스크립트가 최초 렌더링에 필수적이지 않음을 의미합니다. 따라서, 초기 렌더링 후 스크립트의 비동기 로드를 고려해 보세요.

동기식 서버 호출 금지

`navigator.sendBeacon()` 메서드를 사용하여 `unload` 핸들러의 XMLHttpRequests에서 전송하는 데이터를 제한합니다. 많은 브라우저에서 이러한 요청이 동기식으로 처리되어야 하므로 페이지 전환이 때로는 현저하게 느려질 수 있습니다. 다음 코드는 `navigator.sendBeacon()`을 사용하여 `unload` 핸들러 대신 `pagehide` 핸들러에서 데이터를 서버로 보내는 방법을 보여줍니다.

```
<script>
  function() {
    window.addEventListener('pagehide', logData, false);
    function logData() {
      navigator.sendBeacon(
        'https://putsreq.herokuapp.com/Dt7t2QzUkG18aDTMMcop',
        'Sent by a beacon!');
    }
  }();
</script>
```

새 `fetch()` 메서드를 사용하면 데이터를 비동기식으로 쉽게 요청할 수 있습니다. 이 메서드는 아직 모든 경우에 사용할 수 없으므로 사용하기 전에 기능 검색을 통해 이 메서드가 지원되는지 테스트해야 합니다. 이 메서드는 여러 이벤트 핸들러를 사용하지 않고 Promise로 응답을 처리합니다.
XMLHttpRequest에 대한 응답과 달리, fetch 응답은 Chrome 43부터 지원되는 스트림 객체입니다. 이는 `json()` 호출도 Promise를 반환함을 의미합니다.

```
<script>
fetch('./api/some.json')
  .then(
    function(response) {
      if (response.status !== 200) {
        console.log('Looks like there was a problem. Status Code: ' +
response.status);
        return;
      }
      // Examine the text in the response
      response.json().then(function(data) {
        console.log(data);
      });
    }
  )
  .catch(function(err) {
    console.log('Fetch Error :-S', err);
  });
</script>
```

`fetch()` 메서드는 POST 요청을 처리할 수도 있습니다.

```
<script>
fetch(url, {
  method: 'post',
  headers: {
    "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
  },
  body: 'foo=bar&lorem=ipsum'
}).then(function() { // Aditional code });
</script>
```

자바스크립트 파싱 지연

브라우저가 페이지를 렌더링하는 데 수행해야 할 작업을 최소화하려면, 초기 렌더링을 위해 표시되는 콘텐츠를 생성하는 데 중요하지 않은 모든 비필수 스크립트를 지연시키세요.

장기적으로 실행되는 자바스크립트 피하기

실행 시간이 긴 자바스크립트는 브라우저가 DOM 및 CSSOM을 생성하고 페이지를 렌더링하는 것을 차단합니다. 따라서, 최초 렌더링에 필수적이지 않은 초기화 로직과 기능을 나중으로 지연시켜야 합니다. 장기 초기화 작업 시퀀스를 실행해야 할 경우, 여러 단계로 나누어 브라우저가 이러한 단계 사이에 다른 이벤트를 처리할 수 있도록 해야 합니다.

CSS 사용 최적화

CSS는 렌더링 트리를 생성하는 데 필요하며 자바스크립트가 초기 페이지 생성 시 CSS를 차단하는 경우가 많습니다. 비필수적인 CSS를 주요하지 않은 것으로 표시하고(예: 인쇄 및 기타 미디어 쿼리), 주요 CSS의 양과 이를 제공하는 시간을 가능한 한 작도록 해야 합니다.

CSS를 문서 헤드에 넣기

브라우저에서 `` 태그를 검색하고 해당 CSS에 대한 요청을 최대한 빨리 발송할 수 있도록, 모든 CSS 리소스를 가능한 한 HTML 문서의 앞쪽에 지정하세요.

CSS 가져오기 피하기

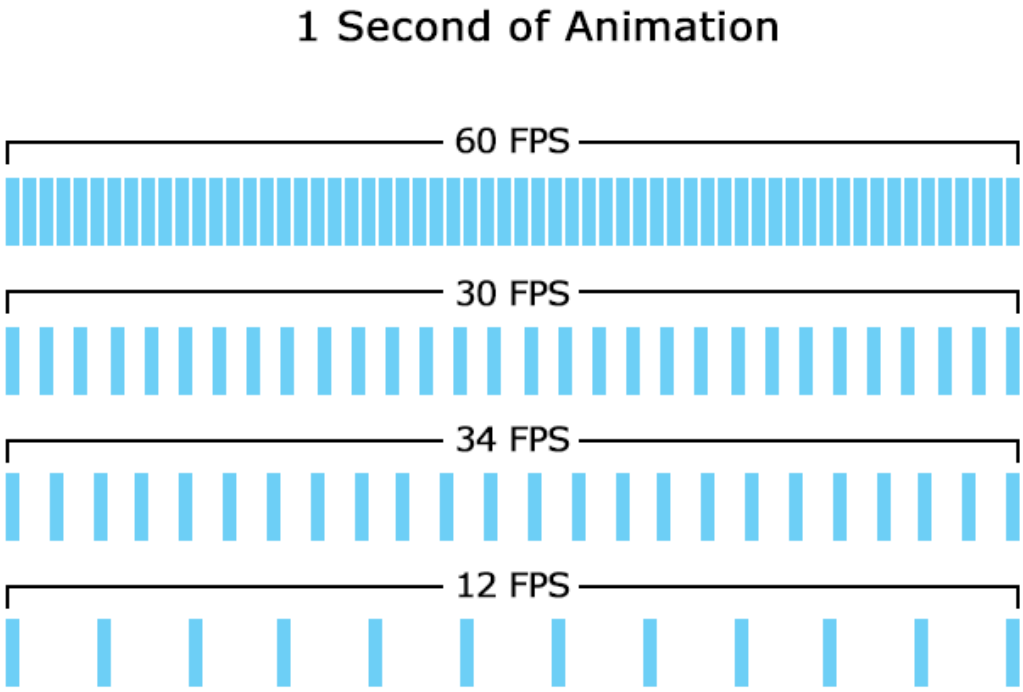
CSS 가져오기(`@import`) 지시문을 사용하면 하나의 스타일시트에서 다른 스타일 시트 파일의 규칙을 가져올 수 있습니다. 하지만, 이러한 지시문은 주요 경로에 대한 추가 왕복을 유도하므로 사용을 피하세요. 가져온 CSS 리소스는 `@import` 규칙을 가진 CSS 스타일시트가 수신되고 파싱된 후에만 검색됩니다.

렌더링 차단 CSS를 인라인 처리

최상의 성능을 위해 주요 CSS를 HTML 문서에 직접 인라인으로 추가하는 것을 고려해야 합니다. 이 경우 주요 경로에 발생하는 추가적인 왕복이 제거되고, (제대로 처리된 경우) HTML만 차단 리소스인 '1회 왕복' 주요 경로 길이를 전달할 수 있습니다.

###

🎙브라우저는 웹페이지를 어떻게 그리나요? - Critical Rendering Path



https://stackoverflow.com/questions/21202555/what-is-60fps-in-web-application

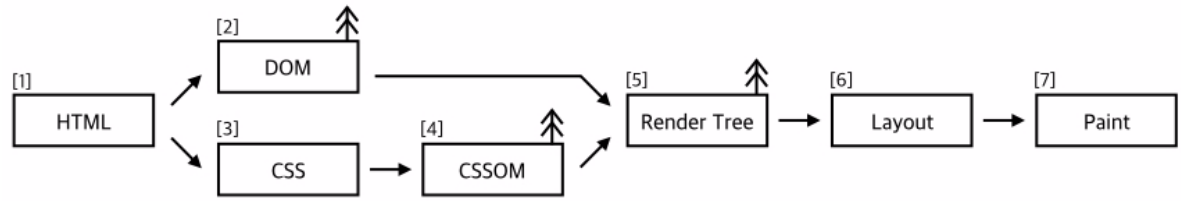
요즈음의 일반적인 스크린은 1초에 화면을 60번 그린다고 합니다. 그러니까 60fps(frame per second)인 것인데, 이 때문에 브라우저 역시 **60fps**를 유지해야 웹페이지가 매끄럽게 보일 수 있습니다. 실제 기기의 스크린이 업데이트되는 속도에 맞춰야 하니까요.

만약 브라우저가 1초에 60프레임을 그려내지 못하게 되면, 웹페이지에서 스크롤을 내린다거나 드래그해서 뭔가를 움직인다거나 할 때 버벅버벅거리는 현상이 생기게 됩니다. (이러한 현상을 **쟁크Jank**라고 합니다.) 혹은 애초에 페이지 자체가 뜨는 데까지 하세월이 걸릴 수도 있습니다. 어느 쪽이든 썩 즐거운 상태는 아니겠죠. 그럴 때 **최적화(Optimization)**가 필요합니다.

그런데, 뭘 최적화해야 하죠??

다짜고짜 최적화라니, 대체 뭘 어디에 최적화 시켜야 하는 것이죠??
브라우저가 1초에 60프레임을 그릴 수 있으려면, 1개의 프레임을 그릴 때 약 0.016초(16ms)를 사용해야 합니다. 이보다 오래 걸린다면 60번 그리기 전에 1초가 끝나버리겠죠....sad... 그러니 페이지가 빠르게 뜨고, 버벅거리지 않도록 하려면 먼저 브라우저가 그 페이지의 하나의 프레임을 어떻게, 얼마동안 그리는지 알아보아야 합니다. 그리고 그 과정에서 시간이 오래걸리는 단계를 좀 더 효율적으로 바꿔 최대한 한 프레임을 빠르게 그려낼 수 있도록 만들면, 매끈한 웹페이지 완성! ~~(...아라고 정리하는데 밥아저씨가 생각나는 이유는 뭘까요..)~~

브라우저가 하나의 화면을 그려내는 이 과정을 **중요 렌더링 경로(Critical Rendering Path)**라고 부릅니다. 우리가 일상적으로 접하는 주소창에 url을 입력하고, 엔터키를 치면 브라우저는 해당 서버에 요청(request)을 보내게 됩니다. 서버에서는 응답(response)으로 HTML 데이터를 내려주는데, 이 HTML 데이터를 실제 우리가 보는 화면으로 그리기까지 브라우저는 다음 단계를 거쳐 작업을 진행합니다. 이 과정의 각 단계가 최대한 효율적으로 이루어지도록 만드는 것을 보통 최적화라고 부릅니다.

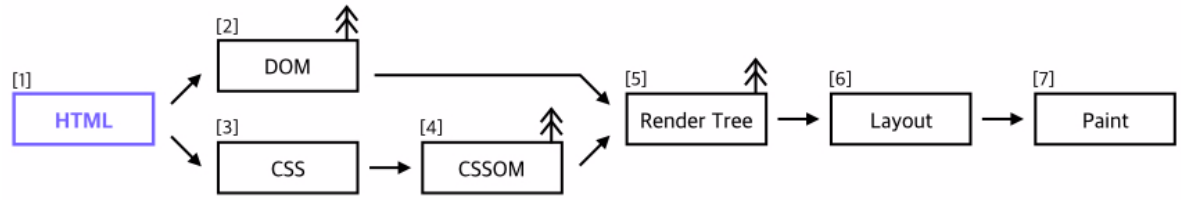


Critical Rendering Path

1 서버에서 응답으로 받은 HTML 데이터를 파싱한다.
2 HTML을 파싱한 결과로 DOM Tree를 만든다.
3 파싱하는 중 CSS 파일 링크를 만나면 CSS 파일을 요청해서 받아온다.
4 CSS 파일을 읽어서 CSSOM(CSS Object Model)을 만든다.
5 DOM Tree와 CSSOM이 모두 만들어지면 이 둘을 사용해 Render Tree를 만든다.
6 Render Tree에 있는 각각의 노드들이 화면의 어디에 어떻게 위치할 지를 계산하는 Layout과정을 거쳐서,
7 화면에 실제 픽셀을 Paint한다.

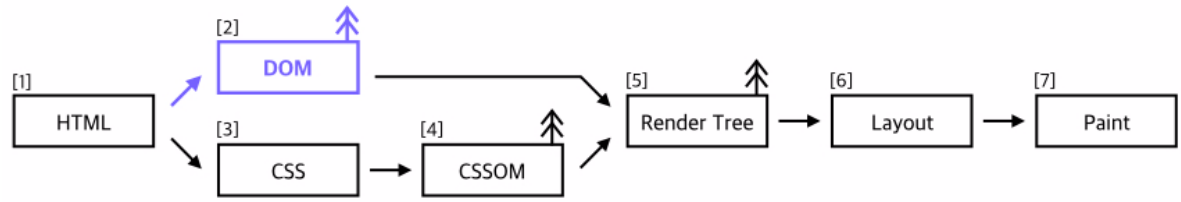
각 단계를 좀 더 살펴보자면,

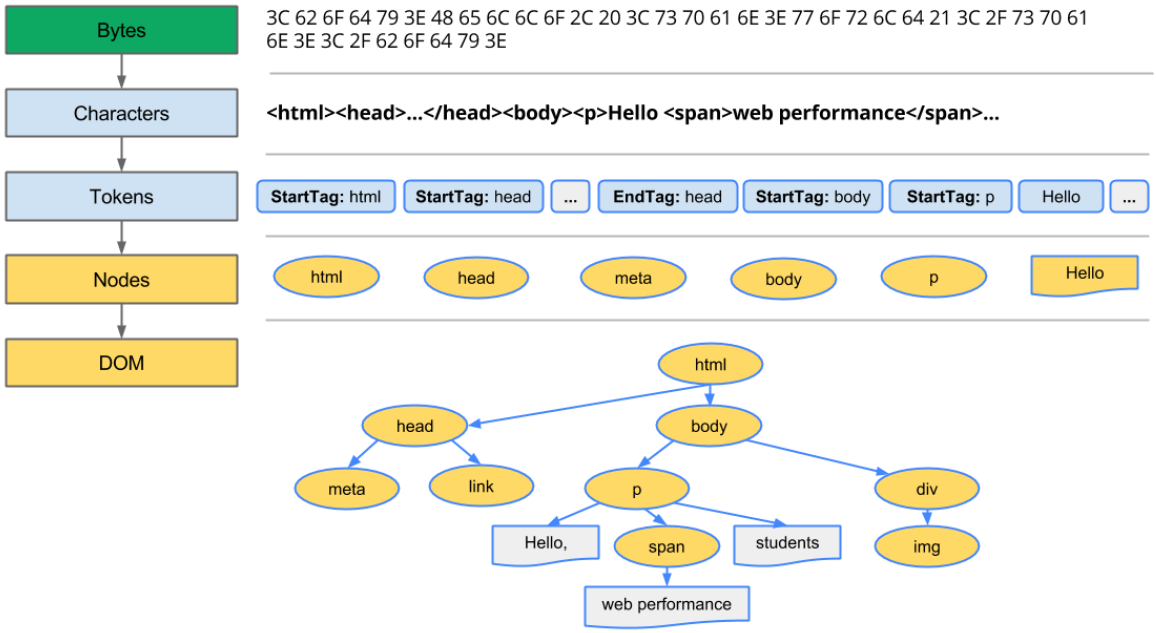
서버에서 응답으로 받은 HTML 데이터를 파싱한다



앞서 언급했듯이, 브라우저 주소창에 url을 입력하고 엔터키를 치면 브라우저는 해당 서버에 요청을 보내게 됩니다. 요청에 대한 응답으로는 위와 같은 형태의 HTML문서를 받아오게 되고, 이걸 하나하나 **파싱parsing**하기 시작하면서 브라우저가 데이터를 화면에 그리는 과정이 시작됩니다.
- 미디어 파일을 만나면 추가로 요청을 보내서 받아옵니다.
- JavaScript 파일을 만나면 해당 파일을 받아와서 실행할 때까지 파싱이 멈춥니다.

HTML에서 DOM Tree로

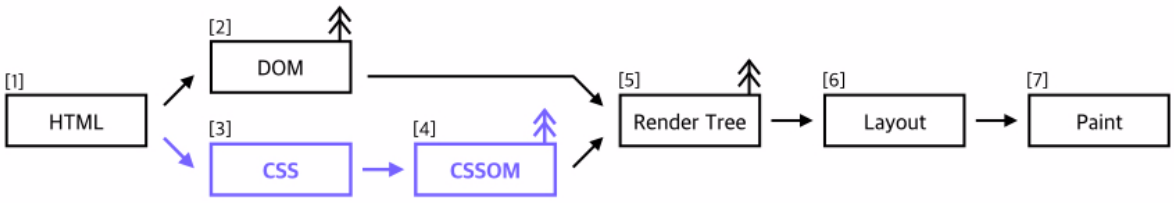




https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model?hl=ko

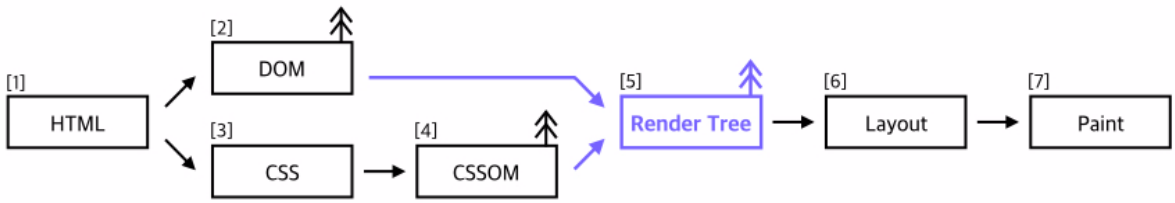
브라우저는 읽어들인 HTML 바이트 데이터를, 해당 파일에 지정된 인코딩(ex.UTF-8)에 따라 문자열로 바꾸게 됩니다.
바꾼 문자열을 다시 읽어서, HTML표준에 따라 문자열을 토큰Token으로 변환합니다. 이미지에서와 같이 이 과정에서 은 StartTag: html 로, 은 EndTag: html 로 변환됩니다.
이렇게 만들어진 토큰들을 다시 노드로 바꾸는 과정을 거칩니다. StartTag: html 이 들어왔으면 html노드를 만들고 EndTag:html 을 만나기 전까지 들어오는 토큰들은 html노드의 자식 노드로 넣는 식으로 변환이 이루어지기 때문에, 과정이 끝나면 Tree모양의 **DOM(Document Object Model)**이 완성되게 됩니다.

CSS에서 CSSOM으로



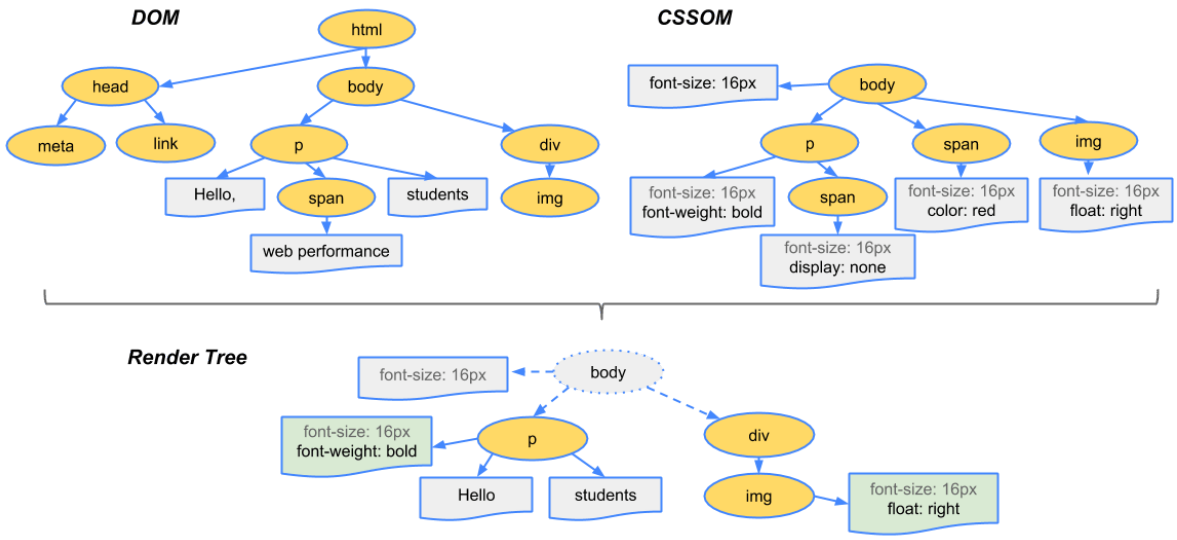
HTML을 파싱하다가 CSS링크를 만나면, CSS파일을 요청해서 받아오게 됩니다.
받아온 CSS파일은 HTML을 파싱한 것과 유사한 과정을 거쳐서 역시 Tree형태의 **CSSOM**으로 만들어집니다. CSS 파싱은 CSS 특성상 자식 노드들이 부모 노드의 특성을 계속해서 이어받는(cascading) 규칙이 추가된다는 것을 빼고는 HTML파싱과 동일하게 이루어집니다.
이렇게 CSSOM을 구성하는 것이 끝나야, 비로소 이후의 Rendering 과정을 시작할 수 있습니다. (그래서 CSS는 rendering의 blocking 요소라고 합니다.)

DOM(Content) + CSSOM(Style) = Render Tree



CSSOM을 모두 만들었으면, DOM과 CSSOM를 합쳐서 Render Tree를 만듭니다. Render Tree는 DOM Tree에 있는 것들 중에서 화면에 실제로 **'보이는'** 친구들만으로 이루어집니다. 만약 CSS에서 display: none 으로 설정하였다면, 그 노드(와 그 자식 노드 전부)는 Render Tree에 추가되지 않는 것이죠. 마찬가지로 화면에 보이지 않는 태그 안의 내용들도 Render Tree에는 추가되지 않습니다.
 그래서 아래 이미지의 Render Tree에는 태그와, display속성이 none인

 태그 하위의 태그가 사라진 것을 확인할 수 있습니다.



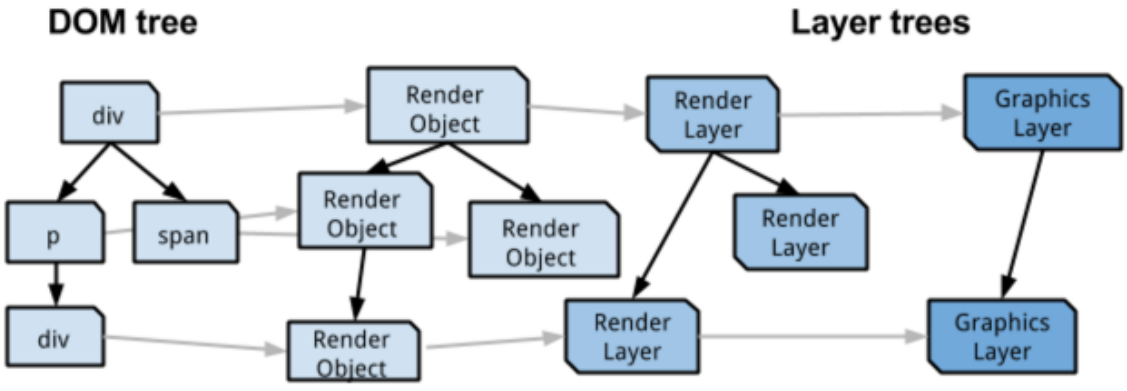
https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction?hl=ko

 DOM과 CSSOM을 참조해서 Render Tree를 만드는 과정샷이 궁금하다면 Udacity강의의 일부인 아래 영상을 참고!

The Render Tree - Website Performance Optimization

Render Object에서 Render Layer로

Render Tree에는 사실 여러 가지가 포함되어 있습니다. Render Object Tree, Render Layer Tree 등등을 합쳐서 화면을 그리는 데에 필요한 모-든 정보를 가지고 있는 Render Tree가 완성됩니다. ('등등'에는 Render Style Tree, InlineBox Tree같은 것들도 있습니다.)



https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

Render Object Tree가 위에서 말했듯이 DOM Tree의 노드 중에서 화면에 보이는 것들만으로 이루어지는 트리입니다. block, inline, image, text, table같은 요소들이 Render Object가 됩니다. DOM Tree에서

는 Render Object Tree에 Block element로, 은 Inline element로 옮겨지는 것이정

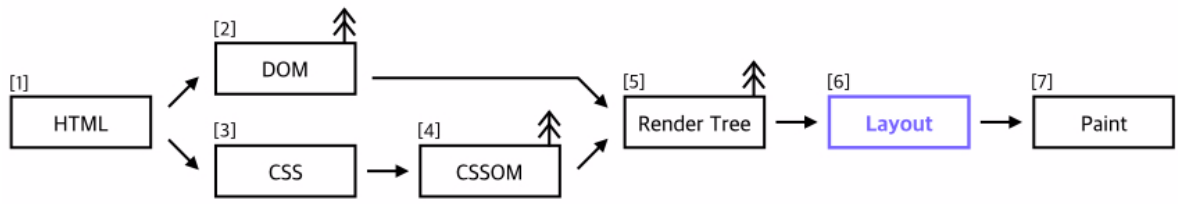
Render Object의 속성에 따라 필요한 경우 **Render Layer**가 만들어집니다. 그리고 이 Render Layer중에서 GPU에서 처리되는 부분이 있으면 다시 **Graphic Layer**로 분리됩니다. 대표적으로는 다음과 같은 속성들이 쓰였을 때 Graphic Layer가 만들어지게 됩니다. ('하드웨어 가속'을 사용할 수 있게 되어 성능을 좋게 한다고 할 때가 요때입니다. 자세한 내용은 여기로)
- CSS 3D Transform(translate3d, preserve-3d 등)이나 perspective 속성이 적용된 경우
-

또는

만약 이런거 저런거 하나도 없이

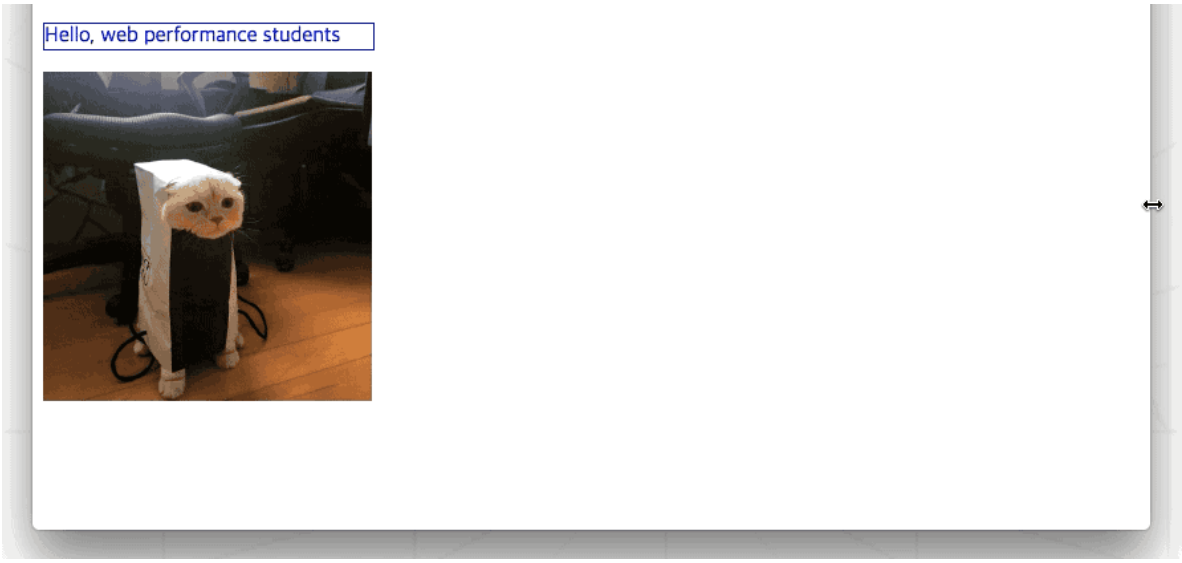
하나에 width정도 속성만 있다고 하면 레이어는 기본으로 만들어지는 하나만 사용하게 됩니다.

Layout(reflow)



화면에 보이는 노드들만을 가지고 있는 Render Tree가 다 만들어지면, 이제 Render Tree에 있는 각각의 노드들이 화면의 어디에 위치할 지를 계산하는 Layout과정을 거칩니다. CSSOM에서 가져온 스타일 정보들로 이미 얘가 어떻게 생겨야 한다는 것은 모두 알고 있지만, 그래서 현재 보이는 뷰포트를 기준으로 실제로 놓으려면 얘가 어디에 가야하는 지는 계산을 또 해야하는 거죠. 여기에서 CSS box model이 쓰이며, position(relative, absolute, fixed..), width, height 등등 틀과 위치에 관련된 부분들이 계산됩니다.

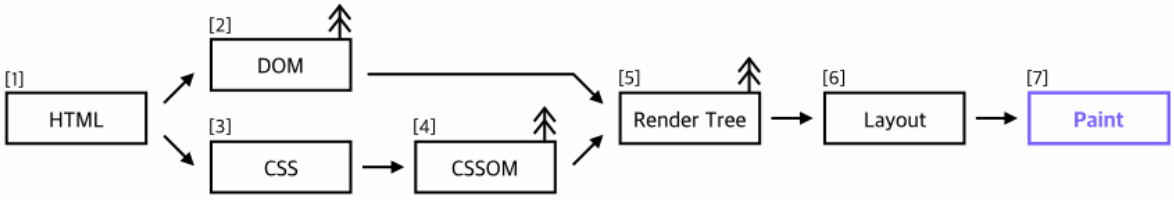
만약 width: 50% 로 되어있는데 브라우저를 리사이즈한다고 하면, 보이는 요소들은 변함이 없으니 Render Tree는 그대로인 상태에서, layout단계만 다시 거쳐 위치를 계산해서 그리게 됩니다.



요런 경우, 같은 렌더 트리를 가지고 layout(과 이후 paint)단계부터 다시 거칩니다

이렇게 화면에 보이는 요소 각각이 어디에 어떻게 위치할 지를 정해주는 과정을 Webkit에서는 layout으로, Gecko에서는 reflow로 부르고 있습니다.

Paint(repaint)



그리고 드디어 Render Tree의 각 노드들을 실제로 화면에 그리게 됩니다..! visibility, outline, background-color같이 정말로 눈에 보이는 픽셀들이 여기에서 그려집니다.
만약 Render Layer가 2개 이상이라면 각각의 Layer를 paint한 뒤 하나의 이미지로 Composite하는 과정을 추가로 거친 뒤에 실제로 화면에 그려지게 됩니다.
(Gecko에서 reflow를 거쳐서 화면에 paint되기까지를 보여주는 영상이 있어서 추가로 붙여둡니다.)

Gecko Reflow Visualization - mozilla.org

참고자료

아래 영상&아티클을 참고하였습니다.

차근차근 알려주기로는 Udacity 강의가 제일..! (한글 자막도 제공!)
Udacity - Website Performance Optimization

그 다음으로는 이 아저씨 발표. (썸네일보다 착하게 생겼어요)

Ryan Seddon: So how does the browser actually render a website | JSConf EU 2015

이건 얼마 못 알아들었지만 Webkit 커미터인 구글러...

Rendering in WebKit

영상 말고 아티클로는,
일단 구글에서 마치 가이드처럼 제공하고 있는 웹 성능 문서.
https://developers.google.com/web/fundamentals/performance/critical-rendering-path/?hl=ko
+a

https://afasterweb.com/2015/08/29/what-the-jank/
https://www.html5rocks.com/ko/tutorials/speed/layers/
http://d2.naver.com/helloworld/59361
https://github.com/nhnent/fe.javascript/wiki/Reflow%EC%99%80-Repaint
http://www.mimul.com/pebble/default/2013/07/07/1373183724195.html
http://donggov.tistory.com/56
http://d2.naver.com/helloworld/2061385
https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome
그리고...끝판왕스러운 마지막 아티클
https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/

모든 내용 출처

https://m.post.naver.com/viewer/postView.nhn?volumeNo=8431285&memberNo=34176766

Accelerated Rendering in Chrome

소개

대다수의 웹 개발자들에게 웹 페이지의 일반적인 모델은 DOM입니다. 렌더링은 일반적으로 이러한 페이지의 표현 형태를 화면상의 이미지로 변환하는 모호한 과정입니다. 모던 브라우저들은 최근 몇년동안 다음과 같은 그래픽 카드의 장점을 취하기 위해 렌더링 동작의 방식을 변경했습니다. 이는 일반적으로 "하드웨어 가속"이라고 모호한 형태로 참조됩니다. 일반 웹 페이지(예를 들어 Canvas2D나 WebGL이 아닌)에 대해 얘기할 때 이 용어는 정말 무엇을 뜻하는 것일까요? 이 글은 크롬에서 웹 컨텐츠의 하드웨어 가속 렌더링을 뒷바침하는 기본 모델에 대해 설명합니다.

몇가지 중요한 전달사항

여기서 우리는 웹킷(WebKit)에 대해 이야기하고 있으며 보다 정확히는 웹킷의 크로미움(Chromium) 포팅에 대해 이야기하고 있습니다. 이 글은 웹 플랫폼 기능이 아니라 크롬의 자세한 구현 사항에 대해 다루고 있습니다. 웹 플랫폼과 표준은 자세한 구현 사항의 수준을 나타내지 않으므로 이글을 다른 브라우저들에 적용하는 것에 대한 어떠한 보장도 할 수 없습니다만 그럼에도 불구하고 내부에 대한 지식은 디버깅의 향상 및 성능 튜닝에 있어 유용할 것입니다.

또한 이 글 전체가 굉장히 빠르게 변경되고 있는 크롬 렌더링 구조의 핵심 부분을 논의하고 있음을 주의하시기 바랍니다. 이 글은 변경되지 않을 것 같은 것들에 대해서만 다루려고 노력하고 있습니다만 6개월 뒤에도 여전히 모두가 적용 가능하다고 보장할 수는 없습니다.

크롬이 현재 얼마간의 기간동안 하드웨어-가속의 경로와 예전의 소프트웨어 경로와 같은 2개의 다른 렌더링 경로를 가지고 있음을 이해하는 것은 매우 중요합니다. 현재 쓰고 있는 모든 페이지들은 윈도우즈, 크롬OS 그리고 안드로이드용 크롬 상에서 하드웨어 가속 경로로 처리되고 있습니다. 맥과 리눅스 상에서는 그들 컨텐츠 일부를 위해 합성(Composition)이 필요한 페이지들만 가속 경로로 처리됩니다만 (무엇이 합성을 필요로 하는지에 대한 더 자세한 내용은 아래에서 볼 수 있습니다.) 맥과 리눅스에서도 곧 모든 페이지들이 하드웨어 가속 경로를 따라 처리될 것입니다.

최종적으로 렌더링 엔진을 살펴보는 중이며 성능 상의 큰 영향을 주는 기능을 확인하는 중입니다. 사이트의 성능을 개선하려는 노력은 레이어 모델을 이해하는데 도움을 주지만 이는 또한 다음과 같이 자기 무덤을 파는 일이기도 합니다. 레이어(Layers)는 유용한 구성물입니다만 지나친 레이어의 생성은 그래픽 스택의 전반적인 오버헤드를 발생시키기도 합니다. 주의하시기 바랍니다!

DOM에서 스크린으로

레이어(Layers)의 소개

일단 페이지가 로딩되고 파싱되고 나면 많은 웹 개발자에게 친숙한 DOM 구조로써 브라우저에서 표현됩니다. 그러나 페이지를 렌더링할 때 브라우저는 개발자에게 직접 보여지지 않는 중간 표현의 연속 과정을 가지게 됩니다. 이 구조들 중 가장 중요한 것은 레이어입니다.

크롬에서는 실제로 DOM의 서브트리와 대응되는 렌더레이어(RenderLayer)와 렌더레이어의 서브트리와 대응되는 그래픽스레이어(GraphicsLayer)과 같은 몇가지 다른 형식의 레이어들이 존재합니다. 후자가 이 글에서 우리에게 가장 흥미로운 부분입니다. 왜냐하면 그래픽스레이어는 텍스쳐로써 GPU에 업로드되는 것이기 때문입니다. 이 글에서는 그냥 "레이어(Layer)"라고 호칭하는 것은 이 그래픽스레이어를 뜻합니다.

다음과 같이 GPU 용어들에 대해 빠르게 보도록 하겠습니다. 텍스처(Texture)는 주 기억장치(예를 들어 RAM)에서 비디오 메모리(예를 들어 GPU 상의 VRAM)로 이동하는 비트맵 이미지 같은 것이라고 생각하시기 바랍니다. 일단 GPU 상에 올라가면 그것을 메쉬 기하 구조와 매핑할 수 있습니다. -- 비디오게임이나 CAD 프로그램에서 이 기법은 스켈레탈 3D 모델들에 "스킨(Skin)"을 주기 위해 사용됩니다. 크롬은 웹페이지 컨텐츠의 덩어리들을 GPU 상에 올리는데 텍스쳐를 사용합니다. 텍스쳐들은 그들을 정말 단순한 사각형 메쉬에 적용함으로써 다른 위치나 변환(Transformation)을 저렴한 비용으로 매핑할 수 있습니다. 이것이 3D CSS이 동작하는 방식이며 빠른 스크롤에도 훌륭합니다. 그러나 이 두가지는 뒤에서 더 알아보도록 하겠습니다.

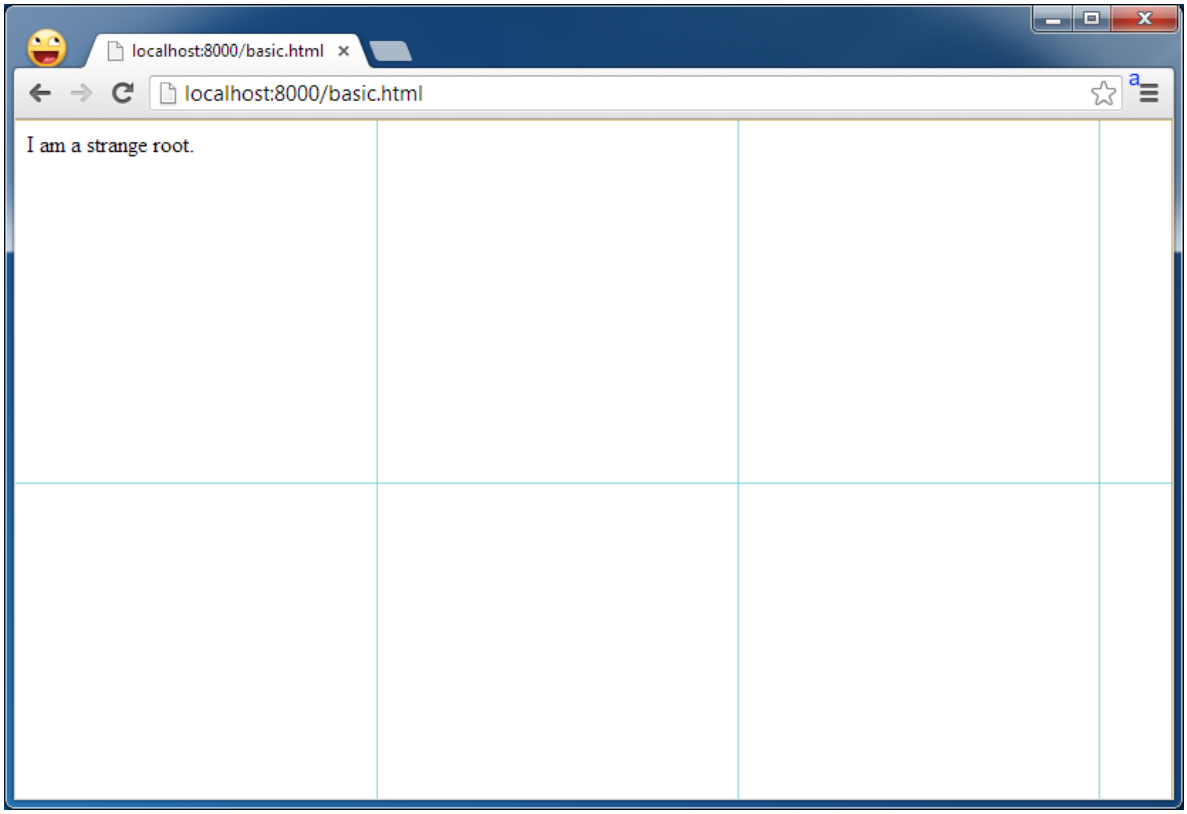
역주: 3D 그래픽스는 일반적으로 어떠한 형태와 그 구조를 정의하기 위한 메쉬(Mesh, Wireframe 형태로 이것을 본 적이 있을 것입니다.)와 그 메쉬의 표면에 출력될 수 있는 일종의 이미지 소스인 텍스쳐(Texture, 이것이 메쉬의 표면에 그려지게 됩니다.)를 가지게 됩니다. 일단 텍스쳐가 GPU의 비디오 메모리 상에 업로드된 뒤에는 이를 출력하는 방식이 변경되더라도 대부분의 경우 이미지의 새로운 업로드없이 약간의 출력 옵션의 조정만으로도 변경된 출력이 가능하며 이 출력 또한 하드웨어 가속에 의해 매우 빠르게 처리됩니다. 따라서 변경이 일어날 경우 웹 컨텐츠를 매번 직접 출력해야 하는 경우보다 훨씬 우수한 성능을 보여줄 수 있습니다.

레이어의 개념을 실제로 보여줄 2가지 예제를 보도록 하겠습니다.

크롬에서 레이어를 학습할 때 매우 유용한 도구는 개발자도구 내 설정(즉, 작은 cog 아이콘)에서 "rendering" 항목 밑의 "show composited layer borders" 플래그입니다. 이는 레이어를 매우 간단한 하일라이트를 화면에 표시합니다. 이것을 켜보시기 바랍니다. 이 글을 쓰고 있는 현재 시점에서 아래의 스크린샷들과 예제들은 모두 최신의 크롬 카나리, 크롬 27버전부터 동작합니다.

Figure 1: 단일-레이어로 구성된 페이지. (새창으로 열기)

```
<!doctype html>
<html>
<body>
  <div>I am a strange root.</div>
</body>
</html>
```

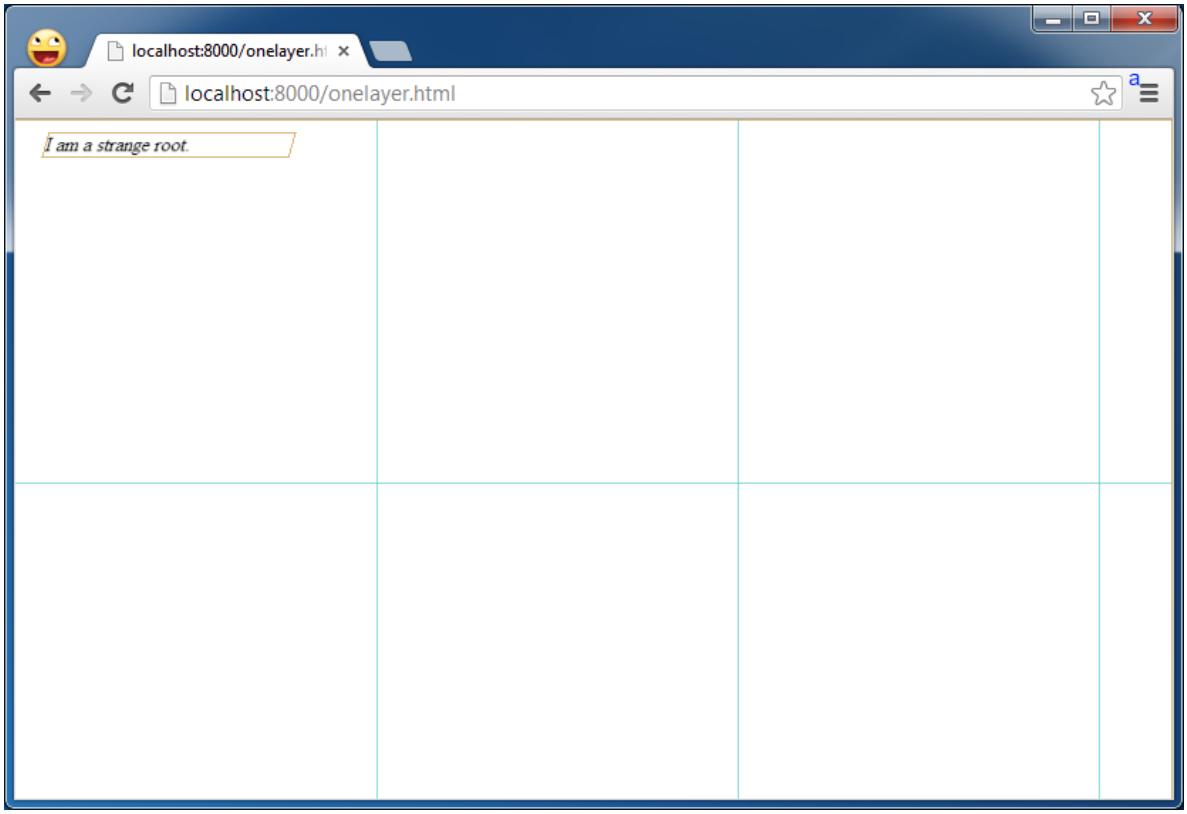


페이지의 기반 레이어 주변의 테두리를 렌더링하는 합성(Composited)된 레이어의 스크린샷

이 페이지는 단 하나의 레이어를 가지고 있습니다. 파란색 그리드는 크롬에서 커다란 레이어의 일부를 GPU로 한번에 업로드하기 위해 사용되는 레이어의 서브-유닛(Sub-unit)으로 간주할 수 있는 타일들을 표현합니다. 이 글에서 정말 중요한 것은 아닙니다.

Figure 2: 레이어의 엘리먼트(새창으로 열기)

```
<!doctype html>
<html>
<body>
  <div style="transform: rotateY(30deg) rotateX(-30deg); width: 200px;">
    I am a strange root.
  </div>
</body>
</html>
```



레이어의 회전으로 렌더링된 테두리의 스크린샷

엘리먼트가 자체의 레이어를 가지고 있을 때

를 회전하는 3D CSS 속성 설정에 의해 다음과 같이 어떻게 보여지는지 확인할 수 있습니다. 이 뷰에서 레이어를 표지하는 오렌지색 테두리에 주의를 기울이시기 바랍니다.

레이어의 생성 기준

레이어 이외에 가지는 것은 무엇일까요? 경험적으로 크롬은 오랜동안 지속적으로 진화해왔지만 현재는 다음과 같은 모든 사항에 대해 레이어를 생성합니다.

- 3D 혹은 시점(Perspective) 변환(Transform) CSS 속성
- 비디오 디코딩 가속을 사용하는

엘리먼트

- 3D(WebGL) 컨텍스트나 가속되는 2D 컨텍스트를 가진

- (예를 들어 Flash)와 같이 포함된 플러그인
- 투명도(Opacity)의 CSS 애니메이션이나 애니메이션되는 변환(Transform)을 사용하는 엘리먼트

- 가속되는 CSS 필터를 가진 엘리먼트
- 합성 레이어(Compositing layer)를 자손으로 가지는 엘리먼트 (즉, 스스로의 레이어를 가지는 자식 엘리먼트를 가진 엘리먼트)
- 형제(Sibling)가 합성 레이어와 낮은 z-index를 가지는 엘리먼트 (즉, 합성 레이어의 상단에 렌더링 되는 엘리먼트)

현실적인 영향 요소: 애니메이션

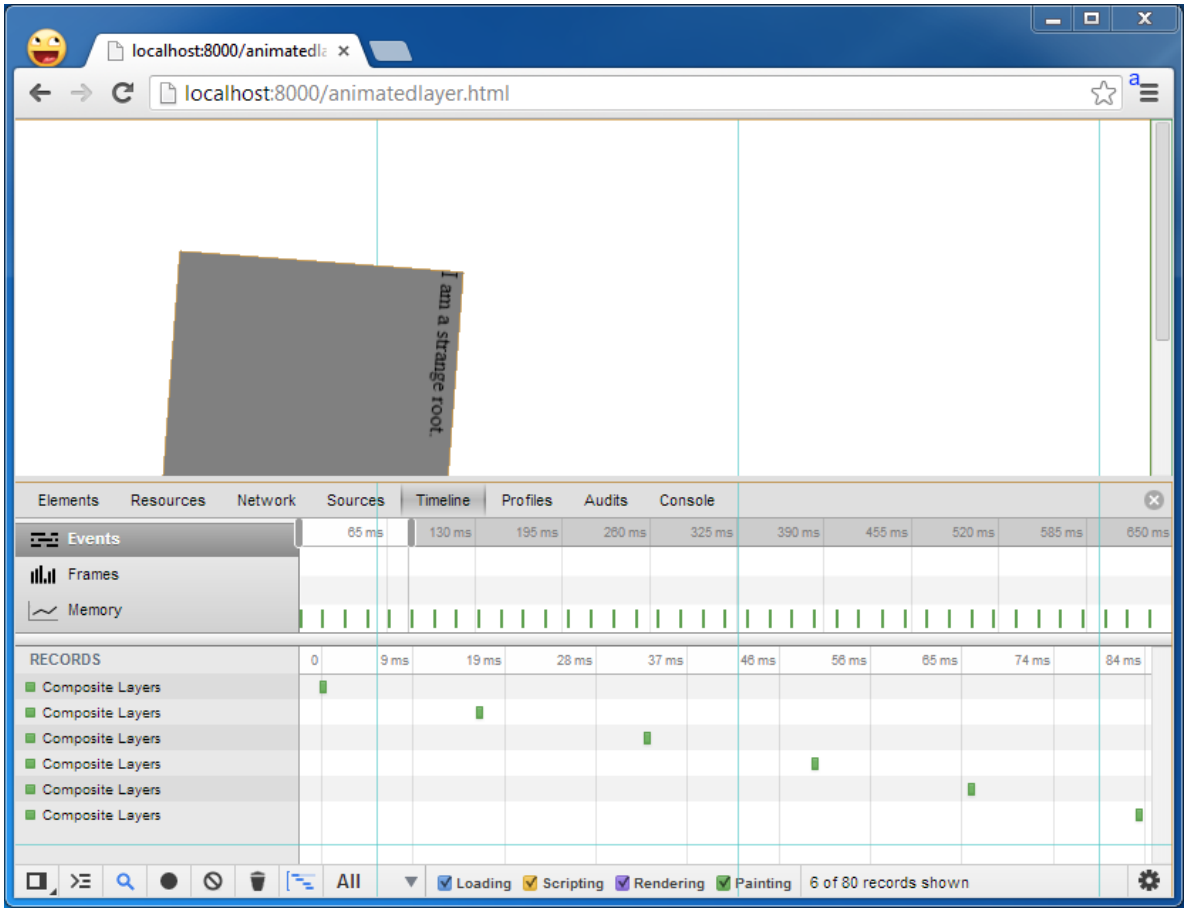
애니메이션을 매우 유용하게 만드는 레이어로 다시 옮겨가보도록 하겠습니다.

Figure 3: 애니메이션된 레이어 (새창에서 열기)

```
<!doctype html>
<html>
<head>
  <style>
  div {
    animation-duration: 5s;
    animation-name: slide;
    animation-iteration-count: infinite;
    animation-direction: alternate;
    width: 200px;
    height: 200px;
    margin: 100px;
    background-color: gray;
  }
  @keyframes slide {
    from {
      transform: rotate(0deg);
    }
    to {
      transform: rotate(120deg);
    }
  }
  </style>
</head>
<body>
  <div>I am a strange root.</div>
</body>
</html>
```

이전에도 말햇듯이, 레이어는 정적인 웹 컨텐츠를 움직이는데 실로 유용합니다. 기본적인 경우, 크롬은 레이어의 컨텐츠를 GPU에 텍스쳐로 업로드하기 전에 소프트웨어 비트맵으로 출력합니다. 만약 앞으로도 컨텐츠가 변경되지 않는다면 다시 그려야할 필요가 없습니다. 이는 다음과 같은 좋은 점들이 있습니다. 재출력(Repaint)는 자바스크립트의 실행과 같은 다른 것들에 사용될 수 있는 시간을 소모할 수 있으며 만약 출력에 긴 시간이 걸린다면 애니메이션이 벌벌 떨리거나 딜레이되는 현상을 초래할 수 있습니다

에를 들어 이관점에서 다음과 같은 개발자도구의 타임라인을 보도록 하겠습니다. 레이어가 앞뒤로 회전하는 동안 아무런 출력 동작도 일어나지 않습니다.



애니메이션 중의 개발자도구 타임라인의 스크린샷

무효한! 재출력(Repainting)

그러나 만약 레이어가 변경된다면 이는 반드시 재출력(Repaint)되어야 합니다.

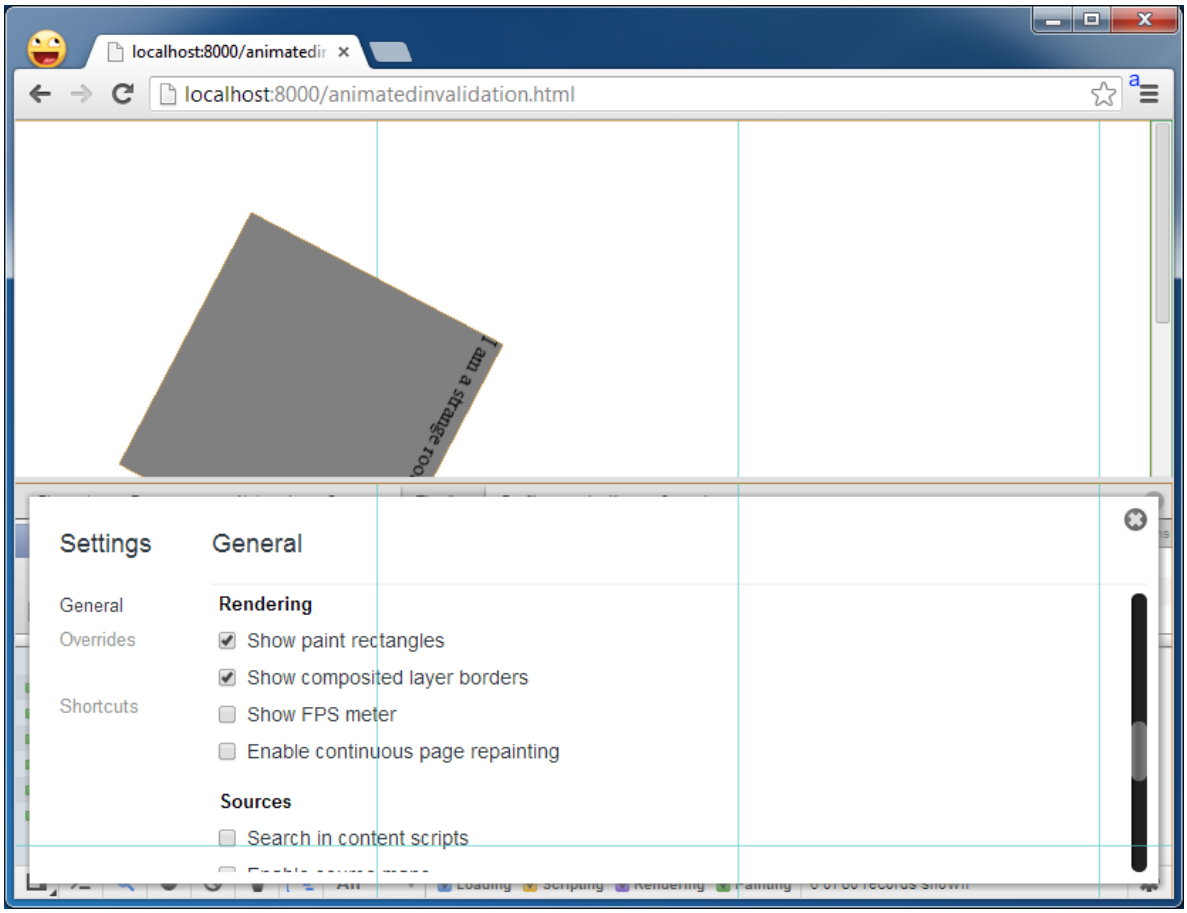
Figure 4: 레이어 다시 그리기(Repaints) (새창에서 열기)

```
<!doctype html>
<html>
<head>
  <style>
  div {
    animation-duration: 5s;
    animation-name: slide;
    animation-iteration-count: infinite;
    animation-direction: alternate;
    width: 200px;
    height: 200px;
    margin: 100px;
    background-color: gray;
  }
  @keyframes slide {
    from {
      transform: rotate(0deg);
    }
    to {
      transform: rotate(120deg);
    }
  }
  </style>
</head>
```

```
<body>
  <div id="foo">I am a strange root.</div>
  <input id="paint" type="button" value="repaint">
  <script>
    var w = 200;
    document.getElementById('paint').onclick = function() {
      document.getElementById('foo').style.width = (w++) + 'px';
    }
  </script>
</body>
</html>
```

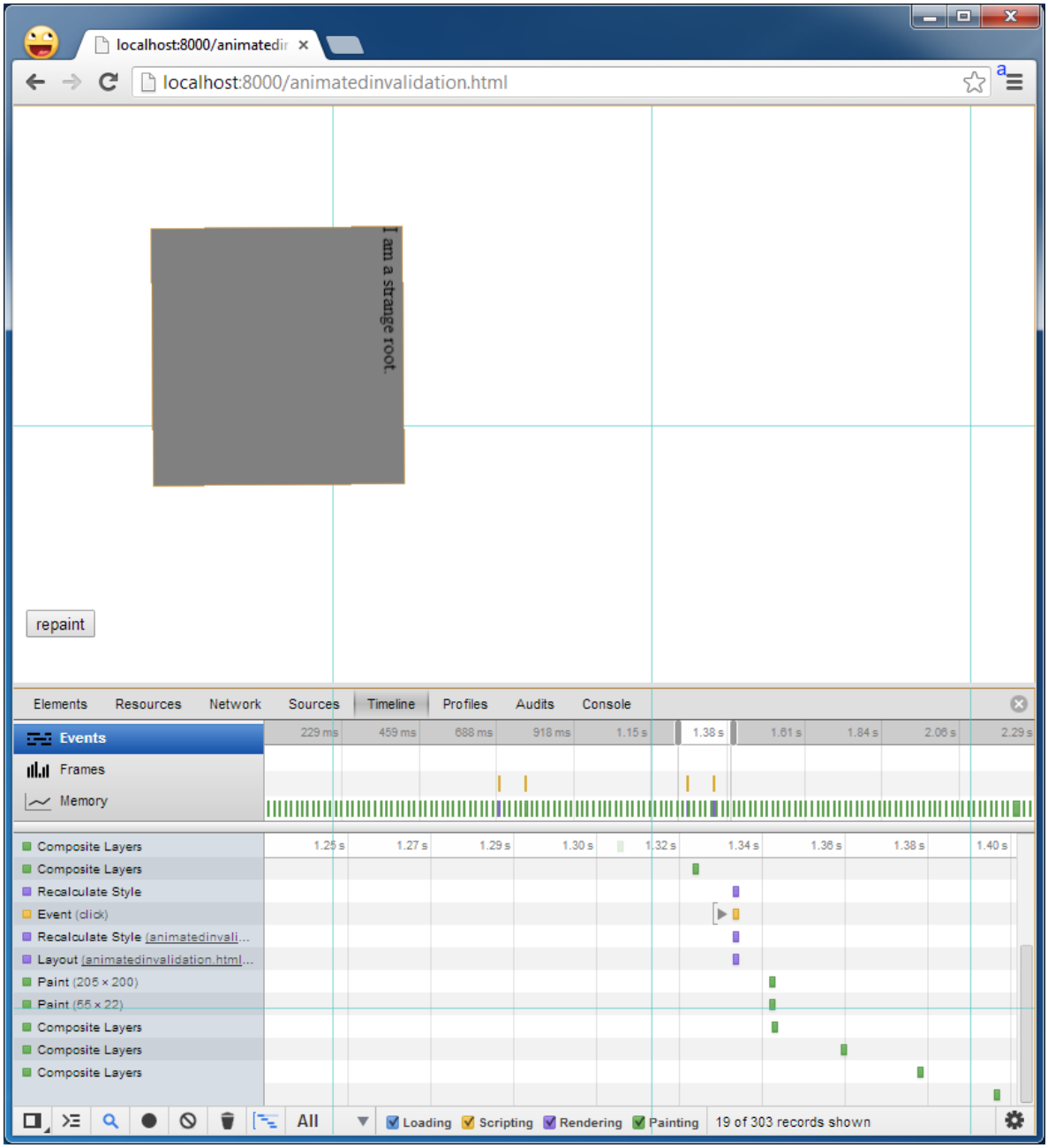
Input 엘리먼트는 클릭이 될 때마다 회전되는 엘리먼트를 폭을 1px씩 넓게 합니다. 이는 재배치(Relayout)와 이 경우 전체 레이어에 해당하는 전체 엘리먼트의 재출력을 발생합니다.

무엇이 출력되는지를 볼 수 있는 좋은 방법은 개발자도구 설정의 "Rendering" 항목 밑의 "show paint rects"입니다. 이를 켠 뒤에 버튼을 클릭할 때마다 애니메이션되는 엘리먼트와 버튼이 둘다 빨간색으로 반짝이는 것을 주의하여 보시기 바랍니다.



show paint rect 화면

출력(paint) 이벤트 역시 개발자도구의 타임라인에 나타납니다. 날카로운 눈을 가진 독자들은 그곳에 다음과 같은 2개의 출력(Paint) 이벤트가 있다는 사실을 알아챌 것입니다. 하나는 레이어에 대한 것이고 다른 하나는 (버튼이) 눌린 상태로(혹은 부터) 변경되었을 때 재출력되는 버튼 그 자체에 대한 것입니다.



개발자도구 타임라인의 레이어 재출력(Repainting) 스크린샷

크롬은 항상 전체 레이어에 대한 재출력을 필요로하지는 않는다는 점에 주의하여야 하며 DOM이 무효화(Invalid)된 DOM의 일부만 재출력함으로써 영리해지려 합니다. 이 경우 수정된 DOM 엘리먼트는 전체 레이어의 크기입니다. 그러나 다른 대부분의 경우 여러개의 DOM 엘리먼트들이 하나의 레이어 안에 있을 것입니다.

그 다음으로 뻔한 질문은 무엇이 무효화(Invalidation)을 발생하고 재출력(Repaint)를 강제로 하게 하는가입니다. 무효화(Invalidation)을 발생할 수 있는 극단적인 많은 경우있기 때문에 완전하게 답하기는 곤란합니다. 가장 일반적인 발생은 CSS 스타일의 조작이나 레이아웃-재정렬(Relayout)의 발생으로 인해 DOM이 갱신되어야 하는 경우입니다. Tony Gentilcore가 작성한 무엇이 레이아웃-재정렬(relayout)을 발생하는가에 대한 훌륭한 블로그 포스트와 Stoyan Stefanov가 작성한 페인팅을 다룬 글은 보다 자세한 사항을 담고 있습니다. (그러나 이 훌륭한 합성(Compositing)과 관련된 것이 아닌 그냥 출력(Painting)에 대해서만 다루는 것으로 끝납니다.)

만약 그것이 작업 중인 무엇인가에 영향을 주는지를 이해할 수 있는 가장 좋은 방법은 원하지 않는 재출력 여부를 확인할 수 있고 재정렬/재출력 직전에 DOM의 갱신 필요 여부를 증명하기 위한 개발자도구의 타임라인과 출력 사각형 보기(Show Paint Rect) 도구를 사용하는 것입니다. 만약 출력이 불가피하지만 이유없이 길게 보인다면 개발자도구의 연속적인 페인팅 모드에 대한 Eberhard Gräther의 글을 확인해 보시기 바랍니다.

종합: DOM에서 스크린으로

그럼 크롬은 어떻게 DOM을 스크린 이미지로 변경할까요? 개념적으로는 다음과 같습니다.

1. DOM을 얻고 그것들을 레이어들로 분리합니다.
2. 이 레이어들 각각을 독립적인 소프트웨어 비트맵으로 출력합니다.
3. 그것들을 GPU에 텍스쳐로써 업로드합니다.
4. 다양한 레이어를 최종 스크린 이미지로 함께 합성합니다.

이것이 크롬이 처음에 웹페이지의 프레임을 생성할 때 일어나는 모든 것입니다. 그러나 그 이후로부터 발생하는 프레임들에 대해서는 다음과 같이 몇가지 손쉬운 방법을 사용할 수 있습니다.

- 정확한 CSS 속성이 변경되면 아무것도 재출력할 필요가 없습니다. 크롬은 그저 이미 GPU에 텍스쳐로 저장되어 있지만 다르게 합성될 속성들(예를 들어 다른 위치에서 다른 투명도 등)을 가진 기존 레이어들을 다시 합성할 수 있습니다.
- 만약 레이어의 일부가 무효화되면 그는 다시 출력되고 업로드됩니다. 만약 컨텐츠가 똑같이 남아 있는데 합성 속성들이 변경(예를 들어 이동하거나 투명도가 변하는 등)되면 크롬은 GPU의 것을 그대로 두고 새로운 프레임을 생성하기 위해 다시 합성을 수행할 수도 있습니다.

역주: '정확한 CSS 속성이 변경될 경우 재출력될 필요가 없다.'는 뜻은 CSS의 속성 변화가 재출력의 회피를 보장한다는 뜻이 아닙니다. 여기서의 정확한 CSS 속성은 합성(Composition)의 발생 조건만을 가지는 CSS 속성들을 뜻합니다. 이러한 대표적인 속성은 trasform의 translate(), rotate(), scale()과 opacity 등이 있습니다. 자세한 사항은 Paul Lewis와 Paul Irish가 쓴 고성능 애니메이션을 참조하시기 바랍니다.

이제 확실하게 말하자면 레이어-기반의 합성 모델(Layer-based compositing model)은 렌더링 성능에 깊은 영향을 가지고 있습니다. 합성(Compositing)은 아무것도 출력(Paint)이 필요하지 않을 때 비교적 저렴하므로 레이어의 재출력을 피하는 것은 렌더링 성능을 디버깅하고자 할 때 전체적인 목표가 됩니다. 요령있는 개발자들은 위의 합성을 발생하는 것들의 리스트를 보고 강제로 레이어의 생성하는 것이 쉽게 가능함을 깨달을 것입니다. 그러나 다음과 같이 그는 공짜가 아니기 때문에 맹목적으로 그들을 생성하기만 하는 것을 주의해야 합니다. (모바일 장비에서는 특히 제한된) 시스템과 GPU의 메모리를 잡아먹고 지나치게 많은 생성은 가시성 추적을 유지하는 또다른 로직 상의 부하를 발생합니다. 많은 레이어는 또한 그들이 크거나 이전에 있지 않던 곳에서 많이 겹치게될 때 픽셀화(Rasterizing)에 소모되는 시간을 실제로 증가시키고 때로는 "과도한 출력(Overdraw)"이라 언급되는 무언가로 이어집니다. 그러므로 이 지식을 현명하게 사용하여야 합니다!

이제 전부 끝났습니다. 레이어 모델의 실제적인 영향에 대한 더 많은 글들을 보시기 바랍니다.

모든 내용 출처: https://www.html5rocks.com/ko/tutorials/speed/layers/