Day 01 (2019-05-08)
1. Browser의 동작

브라우저 구성 요소

항목	설명
사용자 인터페이스	주소표시줄, 이전/다음 버튼, 플러그인, 북마크 기능 등 사용자의 요청에 대한 응답을 표시하는 부분 외의 나머지 인터페이스 부분
브라우저 엔진	사용자 인터페이스와 렌더링 엔진 사이 동작을 제어
렌더링 엔진	사용자 인터페이스에서 요청한 Content를 표시. HTML을 요청하면 응답으로 전송받은 HTML 페이지를 파싱하여 화면에 표시
통신	API 호출, Resource 요청 등 HTTP Request에 사용. 플랫폼 독립적인 인터페이스이며 플랫폼 최하위에서 실행 됨.
UI 백엔드	Dropdown(Select box), Button 등 기본 요소를 그림. OS 별 Browser의 UI 체계를 사용함.

자바스크립트 해석기	자바스크립트 코드를 해석하고 실행	
자료 저장소	쿠키, 폰트, Cache 된 리소스 등 모든 종류의 자원을 하드디스크에 저장하는 공간. HTML5 스펙에 브라우저가 지원하는 Web storage가 이곳에 속함	

렌더링 엔진

렌더링 엔진은 사용자 인터페이스로부터 발생한 요청에 따른 응답을 브라우저에 표시하는 일을 수행한다.
PDF, HTML, XML, JSON, Image 등 다양한 타입의 데이터를 표현할 수 있으며 때로는 파일을 다운로드 하도록 수행할 수 있다.

각 브라우저별 렌더링 엔진의 종류는 아래와 같다.

브라우저	렌더링 엔진
인터넷 익스플로러 (MS)	트라이던트(Trident)
마이크로소프트 엣지(Edge)	최초엔 EdgeHTML이었으나 최근에 Blink로 변경
파이어폭스 (Mozilla)	게코 (Gecko)
크롬 (Google)	27버전 이하 : Webkit
	28버전 이상 : Blink
사파리 (Apple)	웹킷 (Webkit)
오페라 (Opera)	14버전 이하 : Presto
	15버전 이상 : Blink

렌더링 엔진이 요청에 따른 응답을 처리하기 위해 Response의 Payload를 파악하는 방법

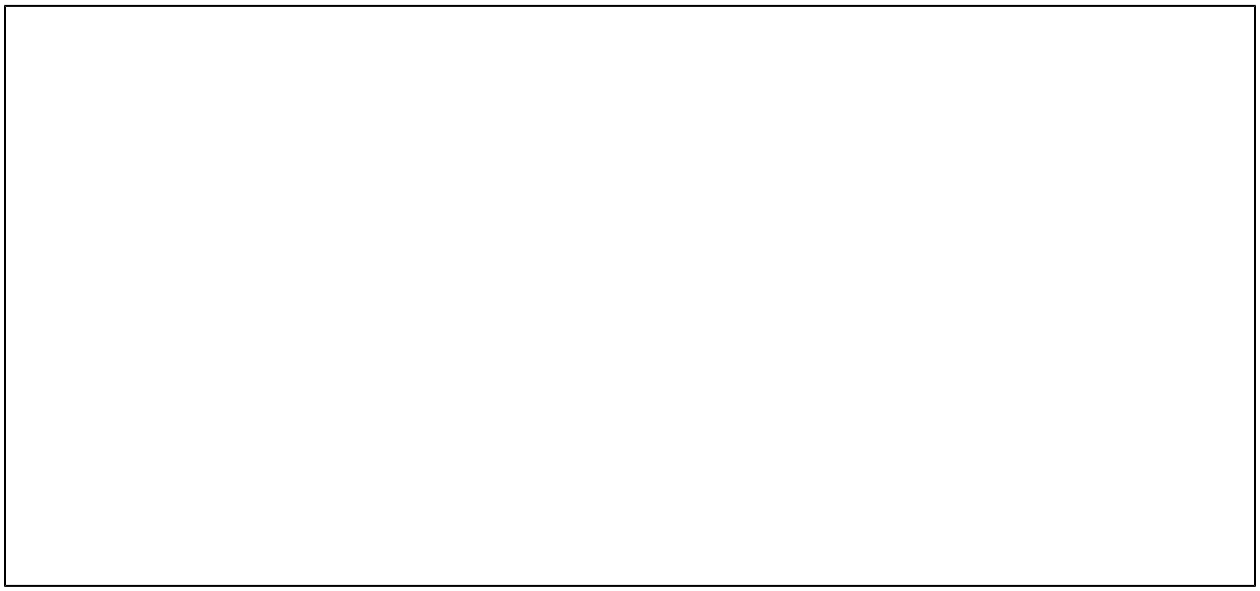
1. 응답 본문을 각각의 파서로 돌려본다?
    - Json Paraser, XML Parser, HTML Parser ...

2. 사용자의 요청에서 힌트를 얻는다?
    - URI의 확장자 (xxx.json, xxx.xml, xxx.html)
3. 서버에서 브라우저에게 알려준다?
    - "응답 데이터는 json 형식입니다"

※ MIME Snipping과 보안
MIME Snipping 혹은 Content Snipping이라고 불리는 기능을 통해 브라우저는 실제 Payload에 대해 올바르지 않은 Content-Type이 응답되었더라도 Payload로 부터 형식을 유추합니다. 예를들어 Content-Type헤더에 application/octet-stream을 내보내면 엑셀이든, PDF든, 이미지든 브라우저에서 알아서 그 형식을 유추하여 다운로드가 처리됩니다. 하지만 편리함을 위해 개발 된 것이나 XSS 공격 등에 악용될 수 있는 취약점이 존재합니다. **x-content-type-options**헤더에 nosniff를 설정하여 응답을 내보내면 브라우저는 MIME Snipping을 처리하지 않습니다. 즉, Server에서 의도한 Content-Type에 따라서만 브라우저가 응답을 처리하도록 강제하는 것입니다. 이러한 방법을 통해 응답 Resource의 유형에 따라 적절히 컨트롤하여 보안 취약점을 예방할 수 있습니다.

렌더링 엔진과 HTML/CSS

렌더링 엔진이 HTML/CSS 사용자 응답을 화면에 그리는 주요 실행 흐름은 (1) Loading → (2) Parsing → (3) Producing a render tree → (4) Layout → (5) Painting의 순서이다



〈웹킷 엔진의 동작 과정〉

(1) Loading

웹 페이지 그리고 웹 페이지에 속한 모든 리소스를 불러오는 과정으로 html, javascript, css, image 등 모든 리소스를 불러오며
브라우저마다 최대 Connection 수가 다르다. 최대 Connection 수를 넘어선 요청들은 Blocking 된다.

(2) Parsing

- HTML Parsing

HTML 문서를 파싱하여 DOM tree를 만들어내는 과정이다. 오류를 허용하며 파싱 도중에 문서 수정이 가능하다.
만약 우리가 작성한 html 파일에 〈head〉 태그가 없는 경우 오류를 허용하여 브라우저에서 자동으로 추가한다. 또한 HTML 파싱 중 Script의 실행으로 HTML의 구조가 바뀔수도 있다.

- CSS Parsing

CSS 파일들을 기반으로 파싱을 진행하며 결과로 Style rule이 도출된다.

(3) Producing a render tree

DOM tree와 Style rules를 통해 Render tree를 생성한다. Render tree는 각 엘리먼트의 색상, 면적 등 시각적 속성 정보를 가지는 랜더 객체로 이루어져있다.

(4) Layout

Render tree를 left-to-right, top-to-bottom의 순서로 탐색하며 랜더 객체에게 화면에 배치 될 위치와 크기를 부여한다

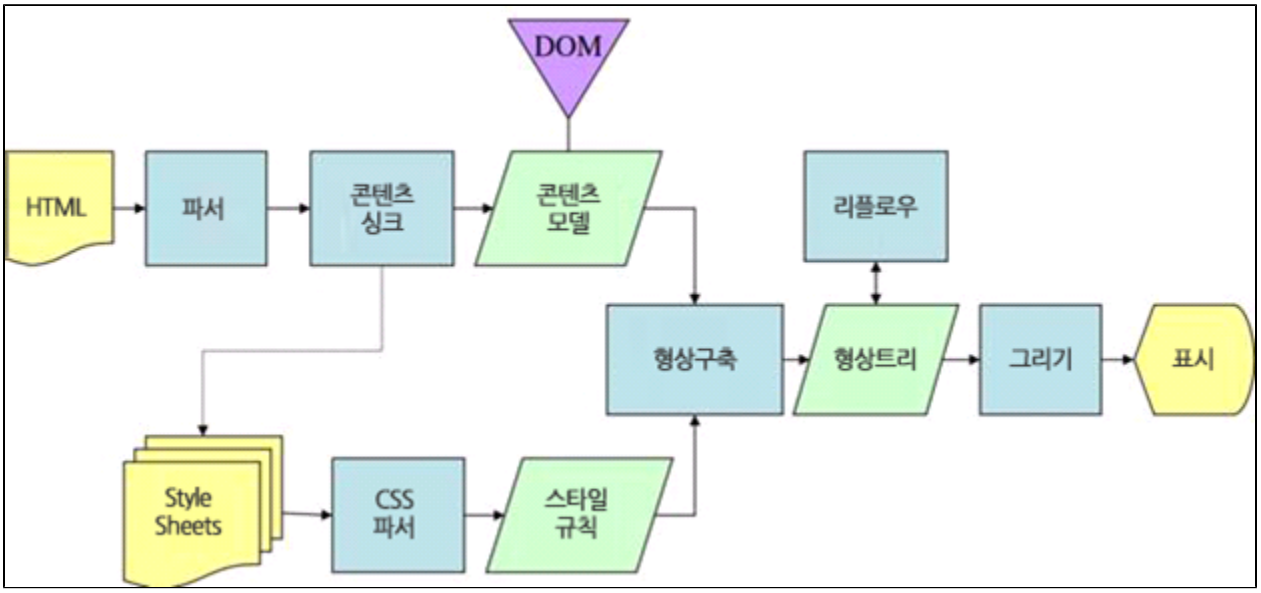
%, em 등의 상대적인 값들이 모드 절대 적인 값 (px)로 변환된다

---

(5) Painting

실제 각 렌더 객체를 그리는(배치하는) 단계로 랜더 트리를 탐색하며 각 객체의 paint()를 실행한다.
보통의 순서는 Background-color, Background-image, border, children, outline의 순서이다

아래는 게코 엔진이 위 과정과 비슷한 순서로 렌더링 하는 과정을 표현한 것이다.



〈게코 엔진의 동작 과정〉

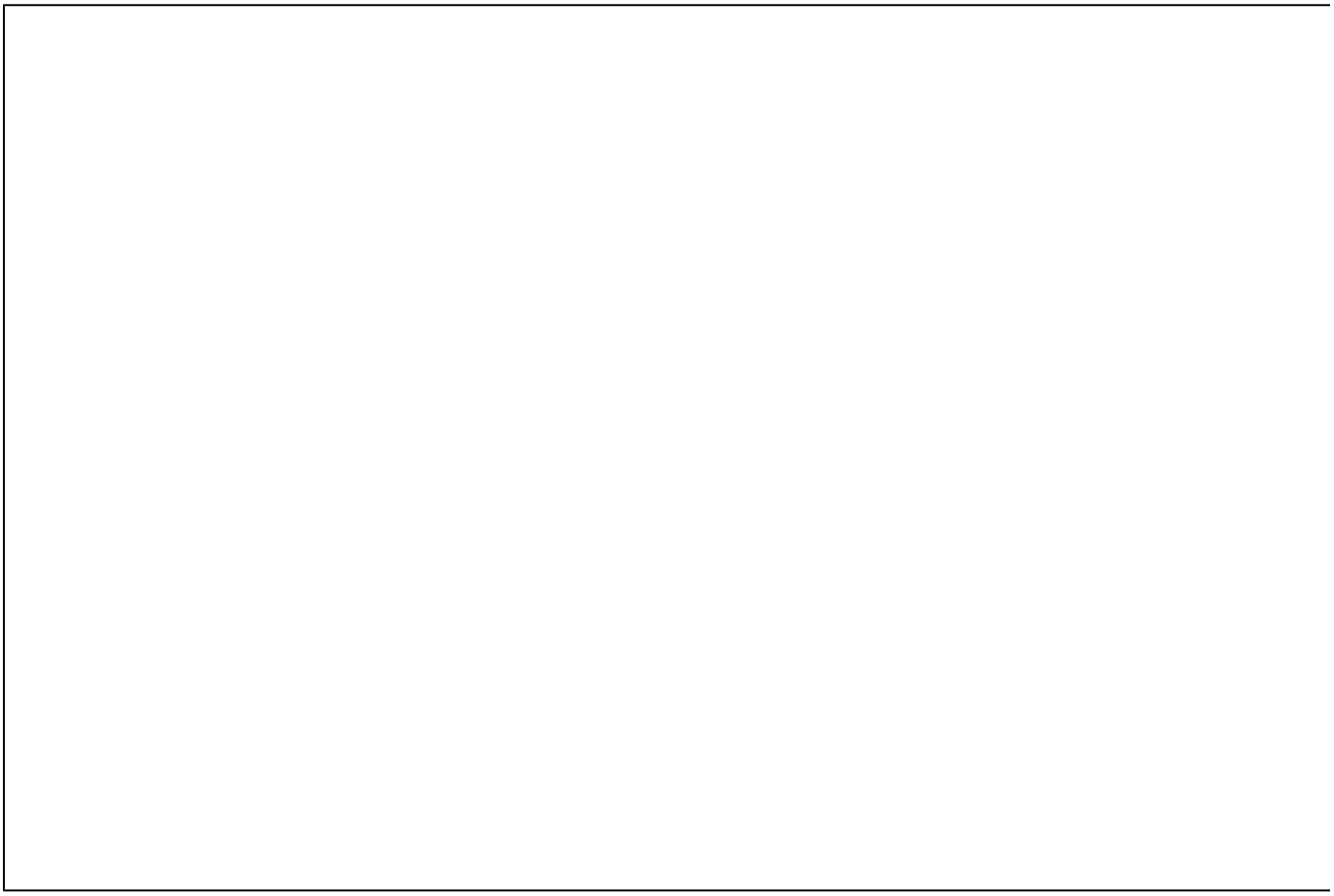
2. Vanilla JS와 Virtual DOM

DOM의 조작(Manipulation)

사용자 화면은 DOM(Document Object Model)의 조작과 관련이 있다. 하지만 앞서의 복잡한 단계로인해 DOM은 느리다는 평가를 매우 많이 받고 있다.

Rendering 속도가 빠르기로 소문난 React를 비롯하여 Modern Javascript Framework들은 Virtual DOM이라는 기술을 내세워 효율적인 렌더링 성능을 약속한다.

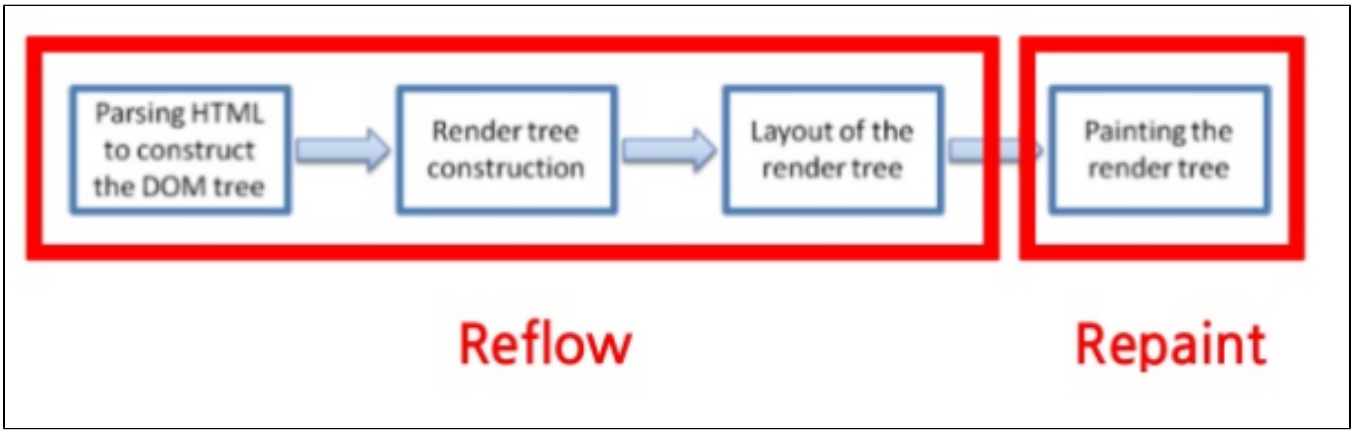
그렇다면 Vanilla JS(**최종적인 DOM 조작 API**)를 통해 DOM을 조작하는 것 보다 Virtual DOM이 빠를까? 빠르다면 얼마나 빠를까?

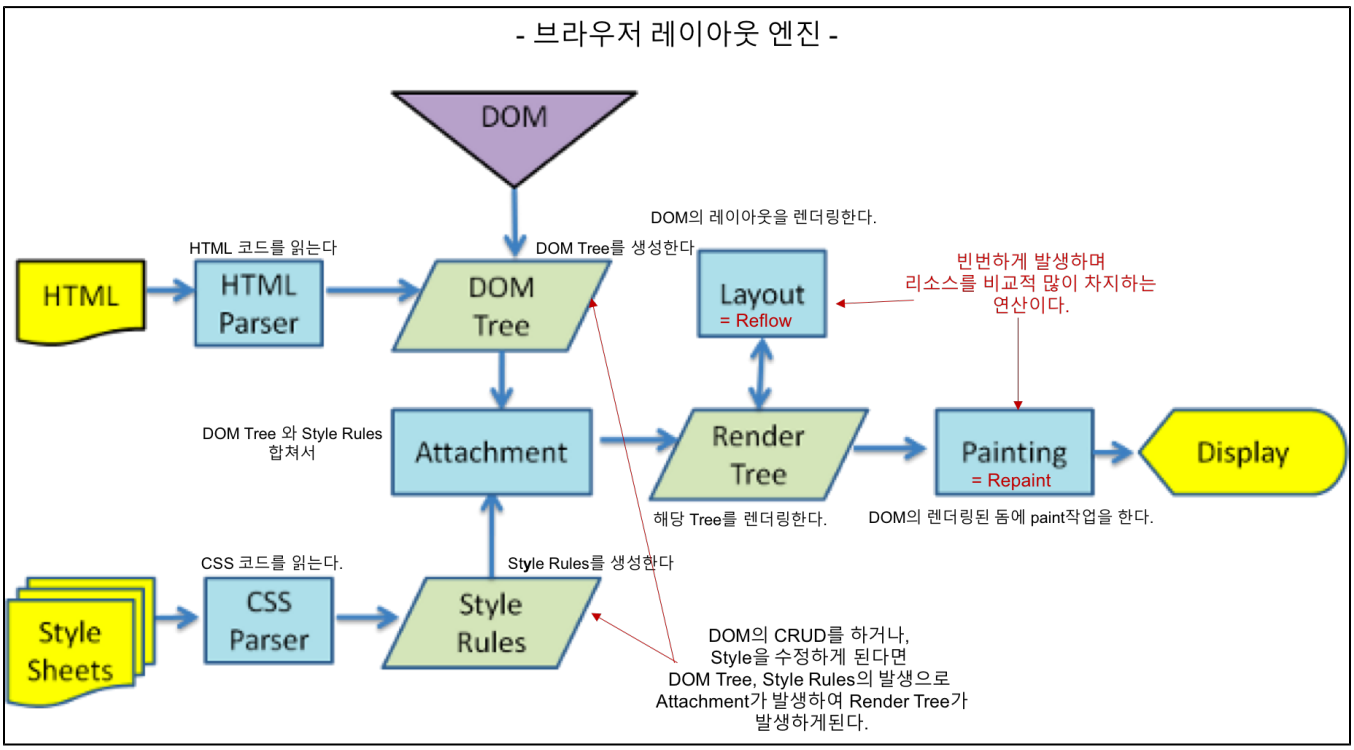


〈렌더링 성능 비교 표 (http://vuejs.kr/jekyll/update/2017/01/02/about-vuejs-korea)〉

실제 비교 표를 확인하면 Vanilla JS가 가장 빠른 성능을 보인다. 그렇다면 왜 많은 개발자들은 Front-end framework를 더 선호하고 사용할까?

Reflow와 Repaint





Reflow (Layout)

Render tree의 전체 또는 일부를 재확인하고 노드의 크기를 다시 계산해야 할 때 발생

적어도 최소한 한번은 발생함 (최초 페이지가 그려질 때)

→ 발생 트리거

- DOM 엘리먼트 추가, 제거 또는 변경
- CSS 스타일 추가, 제거 또는 변경
    - CSS 스타일을 직접 변경하거나, 클래스를 추가함으로써 레이아웃이 변경될 수 있다. 엘리먼트의 길이를 변경하면, DOM 트리에 있는 다른 노드에 영향을 줄 수 있다.
- CSS3 애니메이션과 트랜지션

        - 애니메이션의 모든 프레임에서 리플로우가 발생한다.
    - offsetWidth 와 offsetHeight 의 사용
        - offsetWidth 와 offsetHeight 속성을 읽으면, 초기 리플로우가 트리거되어 수치가 계산된다.
    - 유저 행동
        - 유저 인터랙션으로 발생하는 hover 효과, 필트에 텍스트 입력, 창 크기 조정, 글꼴 크기 변경, 스타일시트 또는 글꼴 전환등을 활성화하여 리플로우를 트리거할 수 있다.

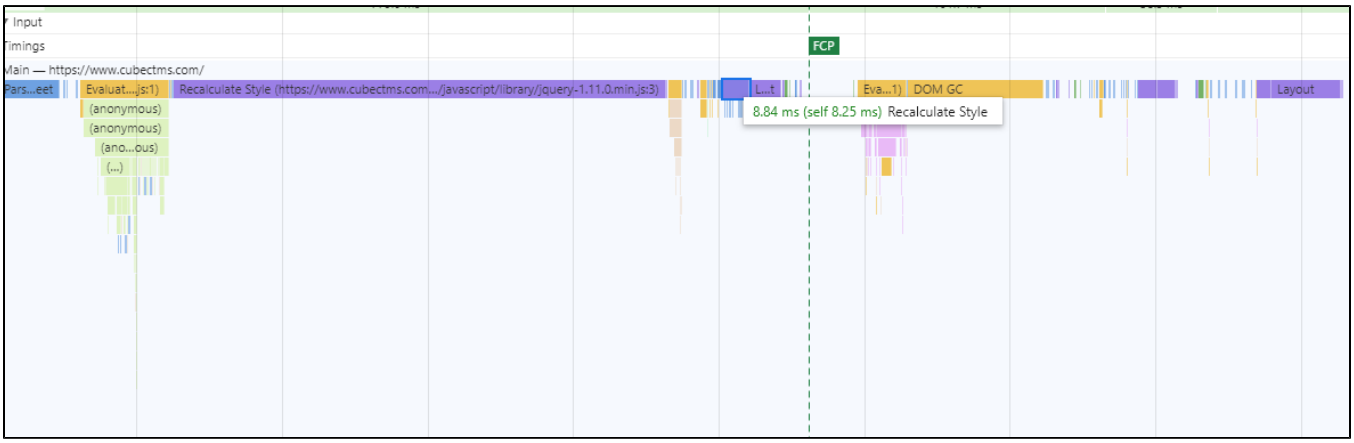
Repaint

화면이 다시 그려져야 하는 경우 발생 (Update)

→ 발생 트리거

- DOM Element의 Board 색상, Background 색상 변경
- Font 색상 변경
- opacity 값 변경

Profiling을 통해 확인 (CSS를 다운로드 받는 족족 Recalculate Style → Reflow(Layout) 발생)



Javascript Code로 확인하는 Reflow, Repaint

```
var bodyElement = document.body;
bodyElement.padding = "20px"; //reflow, repaint
bodyElement.border = "10px solid red"; // reflow, repaint

bodyElement.color = "blue"; //repaint
bodyElement.fontSize = "2em"; //reflow
```

하지만 브라우저가 바보는 아님.

Browser는 reflow, repaint를 최대한 최적화한다. 즉, 발생이 자주하더라도 한꺼번에 모아서 처리하는 lazy 연산 처리가 되어있다.

하지만 Browser의 최적화를 무시하는 경우가 존재함. (강제적으로 Reflow를 발생시킴)

- offsetTop, offsetLeft, offsetWidth, offsetHeight
- scrollTop, scrollLeft, scrollWidth, scrillHeight
- clientTop, clientLeft, clientWidth, clientHeight

Virtual DOM

Virtual DOM 이란?

Virtual DOM은 DOM의 조작을 빨리하는 것이 아닌 DOM 조작에서 발생하는 repaint, reflow 횟수를 줄이는 것으로
핵심 컨셉은 "가상의" DOM에서 변경 사항을 반영하여 처리하고 최종적인 위치에서 DOM에 딱 한번만 적용하자이다.

즉, Vritual DOM 이라는 가상의 DOM을 똑같이 메모리에 올려두고 React, Angular, Vue 등에서 제공하는 API에 따라 가상의 DOM을 조작한다. 그 후 실제 변경이 있는 부분만 추려내어(DIFF) DOM에 반영하는 것이다.

Virtual DOM이 없더라도...

	직접적으로 style 객체에 접근	

cssText 를 활용	function collect() {
	var elem = document.getElementById('container');
	
	elem.style.backgroundColor = 'red';
	elem.style.width = '200px';
	elem.style.height = '200px';
	
	return false;
	}	

	직접 DOM 노드에 접근	Document Fragment (노드조각)을 활용

노드 조각 혹은 노드 사본을 활용	function notReflow() {
	
	var elem = document.getElementById('container');
	
	for (var i = 0; i < 10; i++) {
	var a = document.createElement('a');
	a.href = '#';
	a.appendChild(document.createTextNode('test' + i));
	elem.appendChild(a);
	}
	
	return false;
	}
		function notReflow() {
		
		var frag = document.create
		
		for (var i = 0; i < 10; i++) {
		var a = document.create
		a.href = '#';
		a.appendChild(documen
		frag.appendChild(a);
		}
		
		document.getElementById(
		
		return false;
		}

참고자료

웹 프론트 엔드의 숨겨진 성능 비용(Reflow, Repaint)

Naver D2 - 브라우저는 어떻게 동작하는가?

Virtual DOM의 등장 배경