Reflow 원인과 마크업 최적화 Tip

Reflow 원인과 마크업 최적화 Tip

작성자 정보

- 작성일 : 2011-07-19
- 작성자 : 조규태 (다음서비스 웹표준FT 3파트)
- 문서정보 :

User	Edits	Comments	Watches
조규태	61	0	0

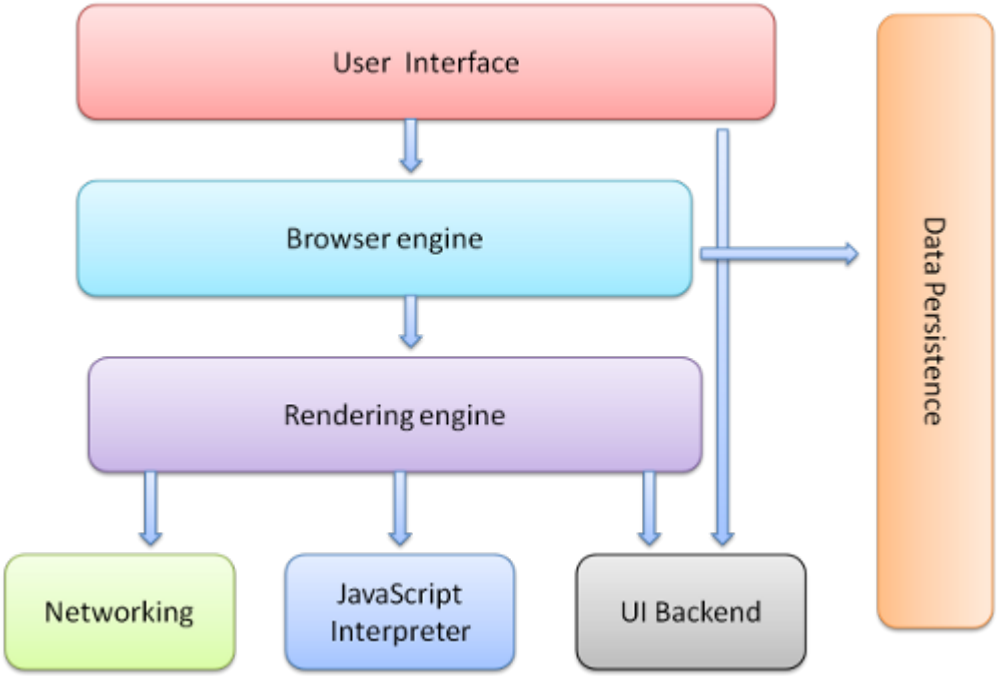
- Table of contents

개요

CSS를 이용한 복잡한 페이지 디자인과 Javascript를 이용한 동적변화가 매우 다양하게 이용되고 있는 상황에서 이에 따른 속도저하 등의 문제점이 발생하고 있다. 이를 원천적으로 해결할 수는 없겠으나 조금이라도 줄일 수 있는 방법들을 찾기 위해 브라우저의 작동원리를 이해해보고, 그에 따른 문제해결 방법을 찾아보고자 한다.

브라우저 렌더링 프로세스의 이해

브라우저의 기본구조

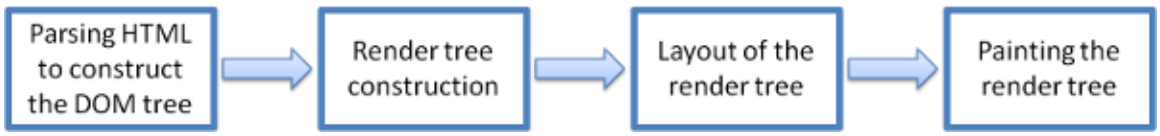


- User Interface - 주소창, 뒤로가기/앞으로가기 버튼, 즐겨찾기 기능등을 포함하며 브라우저 중 웹페이지 표시부분(document)을 제외한 거의 모든 부분에 해당.
- Browser Engine - 렌더링 엔진에 질의를 보내며, 조작하는 인터페이스
- Rendering Engine - 요청된 콘텐츠를 화면에 뿌려주는 기능을 담당함. (전송된 HTML과 CSS 등을 파싱하여 디스플레이)
- Networking - HTTP 리퀘스트와 같은 네트워크 통신기능 수행.
- UI Backend - 브라우처 창의 형태나 셀렉트버튼, 체크박스 등을 표현함. OS의 UI 메소드에 의존함. (XP에서의 셀렉트박스와 윈도우7에서의 셀렉트박스가 다른 것을 생각하면 이해가 쉬움)
- Javascript Interpreter - 자바스크립트 코드의 파싱과 실행에 사용 (유명한 것이 바로 Chrome의 V8 엔진)
- Data Storage - 지속적인 계층(쿠키 등을 위한 저장공간). HTML5에서는 웹DB가 해당됨.

Rendering engine basic flow

브라우저가 네트워크 계층에서 요청된 데이타를 받아오면 렌더링 엔진이 움직이기 시작한다.

다음은 렌더링 엔진의 기본적인 흐름을 도식화 한 것.



1. Parsing HTML : HTML을 파싱하고 DOM Tree를 설계
2. Render Tree
3. Layout : 각각의 노드가 화면내에 위치되어야 할 좌표값 계산 (화면 내 position과 size) 후 배치
4. Paint : *계산되고 지정된 명령에 따라 각각의 노드를 그림

화면구성이 완료된 후 동적인 변화(JS를 통한 DOM 편집, 스타일 수정 등) 발생 시엔?

어떠한 변화가 발생했을 때 브라우저는 최소한의 대응을 하도록 설계되었으며 만일 특정 엘리먼트의 color값에 변화가 발생한다면 오직 해당엘리먼트의 repaint만을 유발한다. 하지만 엘리먼트의 포지션에 변화가 발생했을 경우에는 해당 엘리먼트의 Repaint는 물론 레이아웃까지도 유발(Reflow)한다. html 엘리먼트의 폰트사이즈를 키우는 것과 같은 커다란 변화들은 전체 Render Tree의 Repaint와 Reflow를 유발시킨다.

Reflow? Repaint?

Repaint (or Redraw)

엘리먼트의 스킨에 변화가 발생하지만, 레이아웃에는 영향을 미치지 않을 때 유발된다. (visibility, outline, background-color 등이 포함) Opera에 따르면 Repaint는 해당 행위가 발생하는 순간, 문서내 DOM tree의 다른 노드들의 스킨까지도 검증해야 하므로 비용이 높다고 함.

Reflow

문서 내 노드들의 레이아웃, 포지션을 재계산 후 다시 뿌려주므로 Repaint보다도 더 심각한 퍼포먼스 저하를 유발시키는 프로세스이다. 특정 엘리먼트에 대한 Reflow 발생 시, 페이지에서의 해당 요소는 즉시 Reflow State가 되며 해당 엘리먼트의 자식요소와 부모/조상 요소역시 레이아웃 계산을 진행한다. _(결국은 페이지 전체를 다시 훑는 것이나 마찬가지) _Opera에 의하면, 대부분의 리플로우는 페이지 전체의 렌더링을 다시 일으킨다고 한다.

> " Reflows are very expensive in terms of performance, and is one of the main causes of slow DOM scripts, especially on devices with low processing power, such as phones.In many cases, they are equivalent to laying out the entire page again."
>
> - Reflow는 퍼포먼스 측면에서 매우 고비용을 발생시키는 프로세스이며, 휴대전화와 같은 저성능 디바이스에서는 특히나 더욱 느린 DOM 스크립팅을 발생시키는 주범이다. 많은 경우에서 Reflow는 페이지 전체를 다시한번 레이아웃시키는 결과를 가져온다.

무엇이 Reflow를 유발시키는가?

특정 엘리먼트에 스타일변화가 발생했을 때, 그 개체가 가진 자식요소에 대한 레이아웃 재정리를 위해 Reflow가 실행된다. 설령 그 변화가 그 자식요소 및 페이지에는 아무 영향을 미치지 않을지라도, 기계는 이를 미리 알고있지 못한다. 따라서 작은 변화에도 자식개체는 물론, 페이지 전체에 Reflow가 실행된다. Mozilla에 따르면 다음의 케이스에서 Reflow가 발생한다고 한다.

- 윈도우 리사이징
- 폰트의 변화
- 스타일 추가 또는 제거
- 내용 변화 (인풋박스에 텍스트 입력 등..)
- :hover와 같은 CSS Pseudo Class
- 클래스 Attribute의 동적 변화
- JS를 통한 DOM 동적 변화
- 엘리먼트에 대한 offsetWidth / offsetHeight (화면에서 보여지는 좌표) 계산시
- 스타일 Attribute 동적변화

Reflow를 피하거나 그 영향을 최소화하는 방법

1. 클래스 변화에 따른 스타일 변화를 원할 경우, 최대한 DOM 구조 상 끝단에 위치한 노드에 주어라.
2. 인라인 스타일을 최대한 배제하라.
3. 애니메이션이 들어간 엘리먼트는 가급적 position:fixed 또는 position:absolute 로 지정
4. 퀄리티와 퍼포먼스 사이에서 타협하라
5. 테이블 레이아웃을 피하라
6. IE의 경우, CSS에서의 JS표현식을 피하라.
7. JS를 통해 스타일변화를 주어야 할 경우, 가급적 한번에 처리하라.
8. CSS Rules는 필요한 만큼만 정리하라.
9. position:relative 사용 시 주의하자.

1. 클래스 변화에 따른 스타일 변화를 원할 경우, 최대한 DOM 구조 상 끝단에 위치한 노드에 주어라.

클래스 변화로 인한 Reflow는 물론 피할 수 없겠지만, 그 효과는 줄일 수 있다. DOM 트리에서 가급적 말단에 위치한 노드에 클래스 변화를 줄 경우, 이는 리플로우의 행동반경을 전체 페이지가 아닌 일부 노드들로 제한할 수 있다. 따라서 전체 페이지를 감싸는 wrapper에 클래스를 수정하는 행위는 꼭 피해야 한다. 또한 OOCSS 방식을 통해 클래스변화가 발생할 경우, 특정 엘리먼트에 대해 상당히 많은 클래스를

적용시키는 것 같지만, 실제로는 리플로우의 영향을 최소화함으로써 퍼포먼스적인 측면에서 큰 이득이 발생한다.

2. 인라인 스타일을 최대한 배제하라.

DOM은 매우 느린 구조체이다. 게다가 인라인상에 스타일이 주어진 경우, 리플로우는 페이지 전체에 걸쳐 수차례 발생하게 된다. 만일 인라인스타일이 없을 경우, 외부스타일 클래스의 조합으로 단 한번만 리플로우를 발생시킨다.

3. 애니메이션이 들어간 엘리먼트는 가급적 position:fixed 또는 position:absolute 로 지정

일반적으로 JS (특히 jQuery)나 CSS3로 width/height 또는 위치이동을 구현한 애니메이션은 거의 초단위로 상당한 Reflow를 불러일으킨다. 이러한 경우에 해당 개체의 position 속성을 fixed 또는 absoute로 주게 되면 다른 요소들의 레이아웃에 영향을 끼치지 않으므로 페이지 전체의 Reflow 대신 해당 애니메이션요소의 Repaint만을 유발한다. 이것은 비용적인 측면에서 매우 효율적인 방법이다.

4. 퀄리티와 퍼포먼스 사이에서 타협하라

한 time에 1px을 움직이는 애니메이션 A와 한 time에 3px를 움직이는 애니메이션 B가 있다고 할 때, 애니메이션의 계산과 페이지 Reflow 계산이 동시다발적으로 발생함으로써 CPU 퍼포먼스 비용이 발생하는데, A가 B에 비해 더욱 큰 비용이 발생한다. 속도가 빠른 디바이스에서는 둘다 비슷하게 보이지만, 속도가 느린 (휴대전화와 같은) 디바이스에서는 그 차이가 눈에 띌 수 있다.

5. 테이블 레이아웃을 피하라

테이블로 구성된 페이지 레이아웃은 점진적(progressive) 페이지렌더링이 적용되지 않으며, 모두 로드되고 계산된 후에야 화면에 뿌려진다. 더군다나 Mozilla에 따르면 테이블 레이아웃에서는 아주 작은 변화마저도 해당 테이블 전체 모든 노드에 대한 Reflow를 발생시킨다고 한다. 또한 YUI data table 위젯의 개발자인 Jenny Donnelly 에 의하면, 레이아웃 용도가 아닌 데이터표시 용도의 올바른 테이블이라 할지라도 해당 테이블에 table-layout:fixed 속성을 주는 것이 디폴트값인 auto에 비해 성능면에서 더 좋다고 한다.

6. IE의 경우, CSS에서의 JS표현식을 피하라.

소개된지 오래된 규칙이지만 매우 효과적인 규칙이다. 이 CSS 표현식의 비용이 매우 높은 이유는, 문서 전체 또는 문서중 일부가 Reflow될 때마다 표현식이 다시계산되기 때문이다. 이는 결국.. 애니메이션과 같은 변화에 의해 리플로우가 발생했을 때, 경우에 따라 1초당 수천, 수만번의 표현식 계산이 진행될 수 있다는 것을 의미한다. 때문에 CSS표현식은 반드시 피해야한다.

7. JS를 통해 스타일변화를 주어야 할 경우, 가급적 한번에 처리하라

특정 요소에 스타일변화를 주어야 할 경우 다음과 같이 시도해볼 수 있다.

```
var toChange = document.getElementById('elem');
toChange.style.background = '#333';
toChange.style.color = '#fff';
toChange.style.border = '1px solid #ccc';
```

이러한 접근은 여러번 중복된 Reflow와 Repaint를 유발시킨다.

때문에 위와 같은 방법보다는 다음과 같은 방법으로, 단 한번의 변화만을 발생시키는 것이 더욱 효과적이다.

```
/* CSS */
#elem { border:1px solid #000; color:#000; background:#ddd; }
.highlight { border-color:#00f; color:#fff; background:#333; }
/* js */
document.getElementById('elem').className = 'highlight';
```

8. CSS 하위선택자는 필요한 만큼만 정리하라.

Reflow 자체보다도, Reflow가 유발시키는 CSS Recalculation 에 필요한 내용이다. CSS의 Rule 매칭 프로세스는, 가장 우측의 핵심 선택자에서 좌측으로 흐른다. 이 과정은 더이상 매치시킬 Rule이 없거나 잘못된 Rule이 튀어나올 때까지 계속 매칭시키며 진행된다. 만일 해당 CSS의 특별성(specialty)이 확보되는 선에서, 가급적 딱 필요한만큼만 사용한다면 퍼포먼스 측면에서의 극적인 향상이 발생하게 된다. (즉, 룰이 적을수록 비용절감) 설령 .btn_more라는 클래스가 list_service 내에 쓰이는 유일한 요소일 경우, 아래와 같은 두가지 예가 발생할 수 있다.

첫번째 예

```
.section_service .list_service li .box_name .btn_more {display:block;width:100px;height:30px;}
```

두번째 예

```
.section_service .list_service .btn_more {display:block;width:100px;height:30px;}
```

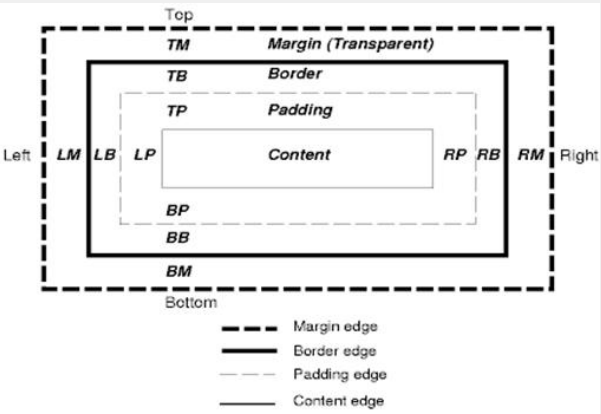
가정 상 둘 다 .btn_more의 specialty가 유효한 CSS임에도 불구하고, 첫번째 예처럼 쓰는 경우는 "코드 가독성"과 같은 이유에서일 것이다. 유지보수의 측면에서 보자면 물론 가독성도 중요한 부분이나, 위의 첫번째 예와 같이 5단계에 걸쳐 필요이상의 규칙들을 작성해놓을 경우 퍼포먼스 하락을 가져올 수 있다. 더군다나 이러한 CSS 코드들이 5~10라인이 아닌, 500~1000라인쯤 될 경우 퍼포먼스에 상당한 영향을 미치게 된다. 때문에 두번째 예와 같이 딱 필요한 선에서 핵심만을 짚는 CSS Rule 선언이 필요하며, 코드 가독성을 위해서라면 차라리 해당 분류 묶음에 CSS 주석처리를 하는 것이 효과적이다. (하위선택자는 가급적 적을수록 좋다)

9. position:relative 사용 시 주의하자.

페이지를 새로 열거나 Reflow가 발생되어 CSS Calculation이 발생할 경우, Box model Calculation → Normal Flow 의 순서로 계산이 진행된다. (Normal flow는 Layout 또는 Reflow라 불리는 과정에 속하는 일부임.) 일반적인 경우, 엘리먼트 들은 margin, border, padding, content(width,height) 등 Box model을 먼저 계산한 후 Normal flow 상태의 레이아웃에 배치된다. (다른말로 선형적 배치)

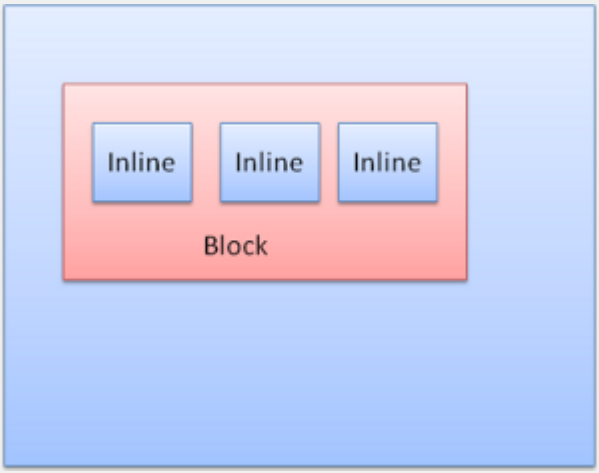
1) Box model Calculation에 의한 계산

아래 이미지와 같이, 화면내 배치가 아닌 각 엘리먼트 자체의 Metrics 계산을 우선 진행한다.



2) Normal flow에 의해 선형적으로 배치된 상태

Box model 계산 후, 마크업 순서에 따라 화면 내 배치를 실행한다. (단, position:absolute 또는 fixed일 경우 Normal flow를 거치지 않고 Out of flow 즉, 곧바로 Positioning을 진행한다.)



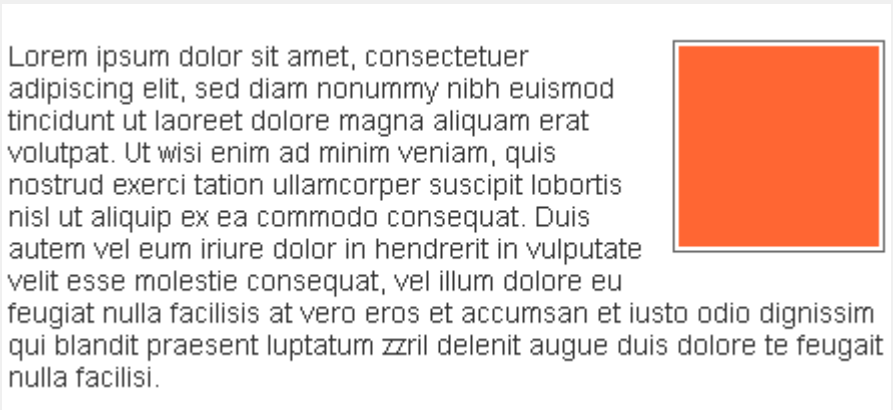
3) Normal Flow 이후

Float냐 Position이냐에 따라 Positioning 과정이 한번 더 일어나는데 각 케이스별 시나리오는 다음과 같다.

→ case 1. Float 속성을 가진 요소

Normal flow 이후 별도의 Positioning 계산은 없으며, 왼쪽 또는 오른쪽으로 자신이 갈수있는 한 끝까지 이동한다.

(즉, Box model → Normal flow → Floating)

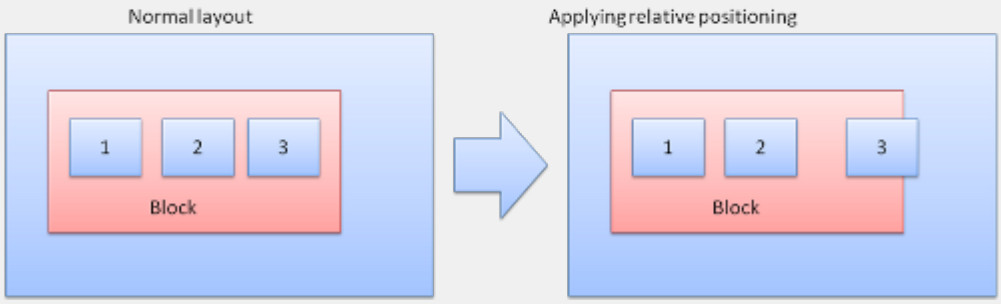


→ caes 2. position:relative;와 함께 top,left 등 위치값을 가진 요소

Normal flow 상태에서 한번 더 Positioning 프로세스를 거치게 된다.

(Box model → Normal flow → Positioning)

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

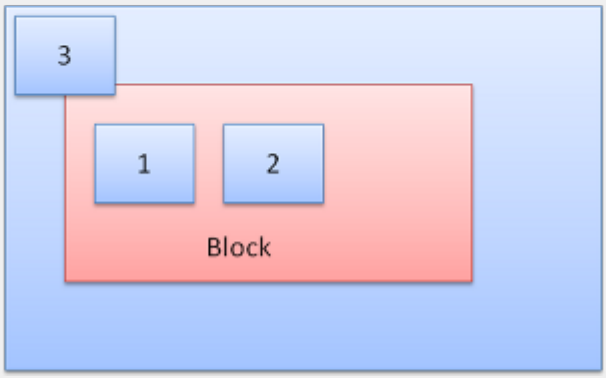


→ case 3. position:absolute 또는 fixed를 가진 요소

Box model 계산 후 Normal flow 과정을 거치지 않고 바로 자신의 위치에 박히게 된다. (Out of flow)

(Box model → Positioning)

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



위에서 확인할 수 있듯, position:relative가 오히려 position:absolute 또는 float 속성보다 더 큰 비용을 가진다. (Box model → Normal flow → Positioning 의 3단계를 모두 거치므로) 때문에 UL 또는 OL과 같은 목록에서 상당수 반복되는 LI 요소에 position:relative 와 top,left 속성등을 주는 경우, 퍼포먼스 하락이 발생할 가능성이 크다.

확인해볼 수 있는 URL : http://211.233.30.20/~bjs/reflow.html

관련 참조자료 및 인용자료

How browsers work - behind the scenes of modern web browsers (by Tali Garsiel)

http://taligarsiel.com/Projects/howbrowserswork1.htm

Reflows & Repaints : CSS Performance making your Javascript slow? (by Nicole Sullivan)

http://www.stubbornella.org/content/2009/03/27/reflows-repaints-css-performance-making-your-javascript-slow/

Efficient Javascript (by Mark Wilton-jones, Opera)

http://dev.opera.com/articles/view/efficient-javascript/

Pegs, Holes And Reflow (by Robert O'Callahan, Mozilla)

http://weblogs.mozillazine.org/roc/archives/2007/11/pegs_holes_and.html

Notes on HTML Reflow (by Chris Waterson, Mozilla)

http://www-archive.mozilla.org/newlayout/doc/reflow.html

Gecko:Reflow Refactoring (Mozilla Wiki)

https://wiki.mozilla.org/Gecko:Reflow_Refactoring

Writing Efficient CSS for use in the Mozilla UI (by David Hyatt)

https://developer.mozilla.org/en/Writing_Efficient_CSS

WebCore Rendering I - The Basics (by David Hyatt)

http://www.webkit.org/blog/114/webcore-rendering-i-the-basics/

CSS Positioning and Layout (by Jennifer Kyrnin)

http://webdesign.about.com/od/beginningcss/p/aacss9layout.htm

The CSS Box Model (by Jennifer Kyrnin)

http://webdesign.about.com/od/beginningcss/p/aacss6box.htm