

Web 기초

RESTFUL API

Ajax

자바스크립트 이벤트 루프 등 비동기 함수 과정

생성자메서드에 화살표구문을 사용하면 어떤 차이점

SPA

Promise와 callback

불변객체

호이스팅

클로저

이벤트 캡처링 버블링

call과 apply, bind

크로스도메인 CORS

여러언어 페이지

null undefined 선언되지 않은 변수의 차이점

자바스크립트 원시 / 참조타입 Number

호스트 객체와 내장 객체의 차이점

실행 컨텍스트 Execution Context

자바스크립트 최적화기법

가상요소선택자, 가상클래스 선택자(CSS Pseudo code)

cookie, sessionStorage, localStorage 사이의 차이점

이미지태그 srcset

CSS 선택자 특이성

컨테이닝박스(block)

Resetting과 Normalizing CSS의 차이점

float의 작동법

z-index의 작동법

inline과 inline-block 그리고 block

relative, fixed, absolute 와 static 요소의 차이점

media 속성

콘텐츠 숨기는 방법

attribute와 property의 차이점

CSS 전처리기 SASS

document load 이벤트와 document DOMContentLoaded 이벤트의 차이점

==와 ===의 차이점

DOCTYPE

http2

브라우저의 렌더링 과정

DIP(Dependency Inversion Principle) 의존 역전 원칙

GPU 가속화

리페인트 리플로우 감소

SEO

OOP 개념 / ES5 ES6 Class 차이

AMP

PWA

SVG

AMD와 CommonJS 차이

Vanila JS

JavaScript 코드디버깅

ES6 스펙

함수형 프로그래밍

docker

Redis

MongoDB

Node.js

D3.js

JWT

쿠키와 세션의 차이

this

이벤트위임

지연로딩설명링크

```
.lazy-background {
    background-image: url("hero-placeholder.jpg"); /* Placeholder image */
}

.lazy-background.visible {
    background-image: url("hero.jpg"); /* The final image */
}
```

RTT설명링크

TCP / IP

RESTFUL API

웹에 있는 자원들을 HTTP를 활용하여 효율적으로 얻기위한 아키텍처스타일의 집합입니다.

웹에 있는 자원들은 URI라는 규칙에 따라 정의되어있고 정한 방법에 따라 그 자원들을 얻거나 수정할 수 있다는 것을 말합니다. HTTP 메소드 중하나인 GET / POST 요청으로 자원을 얻거나 수정하는 것을 예를 들 수 있죠. 이 때 자원만을 URL에는 표기 / 메소드만으로 표현 / 동사말고 명사만 / 확장자는 표시하지 않는다 라는 규칙을 지켜줘야 합니다.

특히 자원에는 아래와 같은 규칙을 지켜야합니다.

1. Uniform-Interface : Self-descriptive messages

독립적으로 자원들이 각각 인터페이스를 가져야 한다는 것입니다. 왜 독립적이야 하는가?

- 웹 페이지를 변경했다고 웹 브라우저를 업데이트할 필요는 없다.
- 웹 브라우저를 업데이트했다고 웹 페이지를 변경할 필요도 없다.
- HTTP 명세가 변경되어도 웹은 잘 동작한다.
- HTML 명세가 변경되어도 웹은 잘 동작한다.

즉, 시간이 지나서 클라이언트와 서버가 변경되더라도 언제나 해석 가능하게끔 하는 것입니다. Self-descriptive messages이란 각 자원들의 타입에 대하여 미디어 타입을 이용하고 그 타입에 대해서 IANA에 등록해야 합니다.(하지만 힘듭니다.) HTTP Header에 타입을 명시해주어야 합니다.

각 메시지(자원)들은 MIME types에 맞춰 표현되며 스스로를 표현해야 합니다. 또한 이 데이터가 무엇을 나타내는지 path를 통해 나타내주어야합니다.

```
{
  "path" : "/kundo1"
}
```

MIME타입

1. Uniform-Interface : HATEOAS 구조

URL 에 따라 다른 페이지를 보여줘야 하는 것은 물론이며, 서버는 클라이언트 요청에 따른 URL RESPONSE를 보내야 합니다.

애플리케이션은 하이퍼링크에 따라 전이가 되어야합니다.

```
// send person object with HATEOAS links added
res.json(personObject, [
  { rel: "self", method: "GET", href: 'http://127.0.0.1' },
  { rel: "next", method: "GET", href: 'http://127.0.0.1/people?_start=1' }
])
```

```
{ rel: "create", method: "POST", title: 'Create Person', href: 'http://127.0.0.1' );
```

이렇게 하거나 data위에 링크를 써야 합니다.

```
{
  "link": "http://kundol.net/todos/{id}"
  "data": []
}
```

2. Stateless

이건 HTTP 자체가 Stateless이기 때문에 HTTP를 이용하는 것만으로도 충족이 됩니다. API를 제공해주는 서버는 Session을 그 서버 쪽에 유지하지 않는다는 의미입니다.

3. Cacheable

HTTP 는 원래 웹에서 작동하는 캐싱이 됩니다. 새로고침을 하면 304가 뜨면서 원래 있던 js와 css 이미지등을 불러오는 것을 볼 수 있습니다. 캐시는 네트워크 요청을 줄여주며 이는 UX향상에 도움이 됩니다. 네트워크 요청시 해당되는 자원들을 복사해서 메모리에 저장해두었다가 또 같은 요청시 네트워크요청을 하지 않고 브라우저메모리에 있던 자원을 다시 반환합니다. HTTP 메서드 중 GET에 한정되어있으며 `Cache-Control:max-age=100` 이런식으로 한정된 시간을 정할 수가 있으며 이 캐싱된 데이터가 유효한지를 판단하기 위해 `Last-modified` 그리고 `Etag` 를 씁니다.

`Etag` 는 전달되는 값에 태그를 붙여서 캐싱되는 자원인지를 확인해주는 것입니다. 예를 들어 `Cache-Control:max-age=100` 으로 형성된 자료는 100초가 지나면 응답이 완료 되었기 때문에 다시 똑같은 자료를 가져올 수 있습니다. 이 때 Etag, 디지털 지문을 사용한다면 똑같은 자원은 캐싱되서 요청을 줄일 수 있습니다 node.js에서는 npm을 다운받아 `res.setHeader('ETag', etag(body))` 이렇게 사용됩니다.

'public' 또는 'private'로 설정이 가능합니다. 응답이 'public'으로 표시되면 이와 관련된 HTTP 인증이 구성되어 있고 응답 상태 코드가 정상적으로 캐시할 수 없는 경우에도 캐시가 가능합니다. 대부분의 경우, 명시적 캐싱 정보(예: 'max-age')가 응답이 어떠한 경우든지 캐시가 가능하다고 나타내므로 'public'이 필요하지 않습니다.

반대로, 'private' 응답은 브라우저가 캐시할 수 있습니다. 그러나 일반적으로 이 응답은 단일 사용자를 대상으로 하므로 중간 캐시가 이 응답을 캐시하는 것은 허용되지 않습니다. 예를 들어, 비공개 사용자 정보가 포함된 HTML 페이지는 사용자의 브라우저가 캐시할 수 있지만, CDN은 이 페이지를 캐시할 수 없습니다.

4. Client-Server 구조

클라이언트와 서버가 서로 독립적인 구조를 가져야 합니다. 물론 이는 HTTP를 통해 가능한 구조입니다. 서버에서 HTTP 표준만 지킨다면 웹에서는 그에 따른 화면이 잘 나타나게 됩니다. 서버는 그저 API를 제공하고 그 API에 맞는 비즈니스 로직을 처리하면 됩니다. 마찬가지로 클라이언트에서는 HTTP로 받는 로직만 잘 처리하면 되는 것입니다.

이외에도 Layered System이 있습니다.

Layered System

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. Layers may also enforce security policies.

Ajax

AJAX: Asynchronous Javascript And XML, 에이젝스라 불리는 이 것을 직역하자면, 비동기 자바스크립트 그리고 XML을 뜻합니다. 이건 어떤 특정한 단일 기술을 뜻하는 용어는 아니며 여러가지 기술집합을 의미합니다. 넓은 의미의 AJAX는 웹 클라이언트 측에서 리로드 없이 비동기적으로 콘텐츠를 변경하기 위해 사용하는 모든 기술을 지칭하며, 좁은 의미의 AJAX는 서버측과 비동기적으로 통신하는 기술을 말합니다.

동기적인 방식은 웹사이트에서 1번 어떤 요청을 하면 그 요청을 수행하는 동안 사용자는 아무것도 못하는 것 2번 화면처리를 못함을 말합니다. 그리고 AJAX를 사용하지 않고 동적인 페이지요청이 들어왔을 때 페이지 자체를 HTML 자체를 넘겨주게 되면 중복되는 데이터까지도 전달되어 효율이 떨어집니다. 비동기적인 방식은 1번을 수행하면서 2번을 수행하는 방식을 말합니다. 그렇기 때문에 데이터 요청을 하게 되어도 화면이 계속 유지가 되며, 사용자가 웹사이트를 화면유지된 상태에서 사용할 수 있으므로 좀 더 상호작용적이다라고 말할수 있습니다.

원리와 과정

이 AJAX는 XMLHttpRequest라는 객체를 통해서 수행됩니다. 웹브라우저와 서버간의 다리를 이 XMLHttpRequest 객체가 만들어주고 그 다리를 통해서 저희는 비동기적으로 데이터를 보내고 받을 수 있게 되는 것입니다.

```
const httpRequest = new XMLHttpRequest();
httpRequest.onreadystatechange = function(){
    if(this.readyState == 4 && this.status == 200){
        document.getElementById("app").innerHTML = this.responseText;
    }
}
httpRequest.open("GET", "/getAjax", true);
httpRequest.send();
```

readyState 정리

상태코드	상태	설명
0	UNSENT	Client has been created. open() not called yet.
1	OPENED	open() has been called.
2	HEADERS_RECEIVED	send() has been called, and headers and status are available.
3	LOADING	Downloading; responseText holds partial data.
4	DONE	The operation is complete.

HTTP라이브러리 중 좋은 것은 fetch와 axios입니다. Promise가 리턴됩니다. 공통적으로 미지원 브라우저에 따라 polyfill을 사용해야 하는 것은 동일합니다. 그러나 서비스워커를 이용한 서비스를 할 경우에는 fetch API를 써야 합니다. 서비스워커는 XMLHttpRequest객체 대신 fetch API에서 이용되는 스트림객체를 쓰기 때문입니다.

서비스워커

- 서비스 워커는 브라우저가 백그라운드에서 실행하는 스크립트로, 웹페이지와는 별개로 작동
- 현재 푸시 알림 및 백그라운드 동기화기능 제공
- 네트워크요청을 가로채고 처리하는 기능
- 오프라인 환경을 완벽히 통제할 수 있는 권한을 개발자에게 부여하여 오프라인 환경을 지원

fetch api

```
fetch('http://example.com/movies.json')
  .then(function(response) {
    return response.json();
  })
  .then(function(myJson) {
    console.log(JSON.stringify(myJson));
  });
});
```

Axios

axios는 현재 가장 성공적인 HTTP 클라이언트 라이브러리 중 하나

장점

- request, response 등에 대한 애러 나눠서 처리 가능해서 애러핸들링 쉬움
- validateStatus 를 사용해 status에 관한 핸들링이 편리
- CancelToken을 이용한 요청취소

그러나 Axios는 내부적으로 XMLHttpRequest를 사용하고 있는데 Service Worker등의 웹 최신 기술이 XMLHttpRequest를 지원하지 않으므로, Service Worker를 사용할 예정에 있는 프로젝트에서는 Axios를 사용할 수 없음

Promise

콜백헬은 해당 절차의 내부함수가 다음 내부함수의 절차를 호출하는 IOC, Inversion Of Control과 연속통과스타일, CPS, Continuation Passing Style이라서 패턴적으로도 안 좋다고 합니다. 이를 위해 **Promise** 가 탄생했습니다. 해결하는 것이 아니라 완화시켜줍니다. 라이브러리로 쓰이다가 이제는 자바스크립트 고유기능으로 채택되었습니다.

font-face로딩에도 쓰입니다. DOM API는 프라미스를 사용합니다 `Promise<FontFace> load();`

promise를 선언했을 때의 단계가 pending, 반환했을 때 then으로 넘겼을 때의 단계가 fulfilled상태라 부릅니다. 그리고 나서 성공과 실패에 따라 반환되는 값이 달라지는데 성공된 객체는 resolve메서드가 관리하고 실패한 객체는 reject메서드가 관리하여 error객체를 반환합니다.

```
async1(1)
  .then(async2)
  .then(result =>{
    throw "큰돌의 애러"
    console.log(result) // 4
  }, reason=>{
    console.log(`이 promise는 이 ${reason}으로 종료가 되었습니다.`);
    //reason으로 받지를 못한다.
  })
}
```

then만으로는 rejected됬을 때의 애러만 받을 수 있다. p.then(onFulfilled[, onRejected]); 프로미스의 기본문법입니다. then의 두번째 인자로는 rejected가 됬을 때만의 핸들러를 담고 있습니다.

그럼 어떻게 해야 할까요?

```
async1(1)
  .then(async2)
  .then(result =>{
    console.log(result) // 4
    throw "큰돌의 애러"
  })
  .catch(reason =>{
    console.log(`이 promise는 이 ${reason}으로 종료가 되었습니다.`);
  })
}
```

catch는 resolve를 처리할 수 있습니다.

또한 catch는 체인에서 개별적인 단계의 로직을 짤 때 유용합니다. 프라미스 거부는 거부 콜백(또는 동일하게 기능하는 catch())을 사용하여 다음 then()으로 건너뜁니다. then(func1, func2)를 사용하는 경우 func1와 func2 중에 하나만 호출되며, 둘은 동시에 호출되지 않습니다. 그러나 then(func1).catch(func2)를 사용하는 경우 둘은 체인에서 개별적인 단계이므로 func1이 거부하면 둘 다 호출됩니다.

promise를 구현하는 방법은 new Promise와 Promise.resolve 두개가 있는데 보통 전자를 쓰는 것이 좋습니다. 왜냐하면 전자가 비동기/동기 로직에도 쓸 수 있으며 후자는 비동기적인 로직에만 쓸 수 있기 때문입니다.

병렬적인 요청 : `Promise.all` 모든 요청이 완료되면 넣었던 순서대로 배열형태로 반환됩니다.

async/await 함수를 쓰는 목적은 여러개의 promise들을 "아름답게" 사용할 수 있게 합니다. try catch로 애러를 잡을 수 있으며 디버깅하기가 쉬워지는 장점이 있습니다.

`await`는 `async`안에서만 써야 하고 `async`는 promise의 resolved된 값을 반환합니다.

자바스크립트 이벤트 루프 등 비동기 함수 과정

비동기함수 : WebAPIs 의 함수(DOM, Timer, Ajax)들을 쓰게 되면 브라우저 내 WebAPIs 백그라운드공간내에서 실행해 다 완료가 되는 순서대로 queue에 쌓이고 그 후 이벤트 루프를 통해서 콜스택에 올라가 실행이 됩니다.

화살표구문

화살표 함수 표현식 (arrow function expression) 화살표 함수 표현(arrow function expression)은 function 표현에 비해 구문이 짧고 자신의 `this`, `arguments`, `super` 또는 `new.target`을 바인딩 하지 않습니다. 화살표 함수는 항상 익명입니다. 이 함수 표현은 메소드 함수가 아닌 곳에 가장 적합합니다. 그래서 생성자로서 사용할 수 없습니다. ES6의 화살표 함수는 서브루틴을 정의할 때 일반 함수를 사용하는 것보다 훨씬 낫다. 화살표 함수는 lexical `this`를 가지기 때문입니다.

화살표구문이 할 수 없는 것. 생성자(constructor) / 프로토타입을 이용한 함수 정의 / 객체 메소드 이러한 것들을 하려면 일반 함수를 써야 한다. (일반함수는 dynamic `this`를 가진다)

하지만 생성자 함수는 클래스 정의로 대체 / 서브루틴은 화살표 함수로 대체 / 메소드는 메소드 정의로 대체 할 수 있다.

활용 예)

```
// 파라미터를 입력받아 로직 수행
(param1, param2, .... paramN) => { statements }

// 파라미터를 입력받은 후 값 리턴
(param1, param2, .... paramN) => expression;
(param1, param2, .... paramN) => { return expression; }
```

```
//파라미터가 하나인 경우에는 괄호 생략 가능
(param1) => { statements }

param1 => { statement }

//파라미터가 없는 경우에는 반드시 빈 괄호를 이용
() => { statements }
```

다른 객체지향언어에서 '현재 메소드를 실행하고 있는 인스턴스'를 가리키기 위해 this를 사용하는 것과는 달리, 자바스크립트에서 this는 실행 맥락(execution context)에 따라 달리 해석됩니다. 생성자 함수 내에서는 새롭게 생성하는 객체를, 객체 메소드 내에서는 현재 실행 중인 객체를, 그리고 그 외의 경우에는 함수의 실행 맥락에 의해 결정됩니다. 하지만 이미 언급한 대로, 화살표 함수는 자기 자신의 this가 바인드되지 않도록 설계되어 있습니다. 즉, 함수의 실행 맥락(어떻게 호출되었는가.)과 무관하게, 함수가 정의되는 **lexical scope**에서의 this가 this로서 적용됩니다. 즉, 위의 예제는 다음과 같이 변경될 수 있습니다.

- Lexical this : 화살표 함수는 함수를 선언할 때 this에 바인딩할 객체가 정적으로 결정된다. 동적으로 결정되는 일반 함수(자신을 호출하는 객체를 가리킴)와는 달리 화살표 함수의 this 언제나 상위 스코프의 this를 가리킨다.

```
function Person(){
    this.age = 0;

    setInterval(() => {
        // 여기서의 this는 new Person()으로 생성되는 바로 그 객체이다.
        this.age++;
    }, 1000);
}

var p = new Person();
```

SPA

단일 페이지 애플리케이션(Single Page Application, SPA)은 모던 웹의 패러다임이다. SPA는 기본적으로 단일 페이지로 구성되며 기존의 서버 사이드 렌더링과 비교할 때, 배포가 간단하며 네이티브 앱과 유사한 사용자 경험을 제공할 수 있다는 장점이 있다. link tag를 사용하는 전통적인 웹 방식은 새로운 페이지 요청 시마다 정적 리소스가 다운로드되고 전체 페이지를 다시 렌더링하는 방식을 사용하므로 새로고침이 발생되어 사용성이 좋지 않다. 그리고 변경이 필요없는 부분을 포함하여 전체 페이지를 갱신하므로 비효율적이다. SPA는 기본적으로 웹 애플리케이션에 필요한 모든 정적 리소스를 최초에 한번 다운로드한다. 이후 새로운 페이지 요청 시, 페이지 갱신에 필요한 데이터만을 전달받아 페이지를 갱신하므로 전체적인 트래픽을 감소할 수 있고, 전체 페이지를 다시 렌더링하지 않고 변경되는 부분만을 갱신하므로 새로고침이 발생하지 않아 네이티브 앱과 유사한 사용자 경험을 제공할 수 있다. 모바일의 사용이 증가하고 있는 현 시점에 트

래피의 감소와 속도, 사용성, 반응성의 향상은 매우 중요한 이슈다. SPA의 핵심 가치는 사용자 경험(UX) 향상에 있으며 부가적으로 애플리케이션 속도의 향상도 기대할 수 있어서 모바일 퍼스트(Mobile First) 전략에 부합한다.

단점

초기 구동 속도, SPA는 웹 애플리케이션에 필요한 모든 정적 리소스를 최초에 한번 다운로드하기 때문에 초기 구동 속도가 상대적으로 느린다. 하지만 SPA는 웹페이지보다는 애플리케이션에 적합한 기술이므로 트래픽의 감소와 속도, 사용성, 반응성의 향상 등의 장점을 생각한다면 결정적인 단점이라고 할 수는 없다.

SEO(검색엔진 최적화) 문제, SPA는 서버 렌더링 방식이 아닌 자바스크립트 기반 비동기 모델(클라이언트 렌더링 방식)이다. 따라서 SEO는 언제나 단점으로 부각되어 왔던 이슈다. 하지만 SPA는 정보의 제공을 위한 웹페이지보다는 애플리케이션에 적합한 기술이므로 SEO 이슈는 심각한 문제로 볼 수 없다. Angular 또는 React 등의 SPA 프레임워크는 서버 렌더링을 지원하는 SEO 대응 기술이 이미 존재하고 있어 SEO 대응이 필요한 페이지에 대해서는 선별적 SEO 대응이 가능하다. 즉, **Server side rendering** 를 이용하면 해결된다.

라우팅

라우팅이란 출발지에서 목적지까지의 경로를 결정하는 기능이다. 애플리케이션의 라우팅은 사용자가 태스크를 수행하기 위해 어떤 화면(view)에서 다른 화면으로 화면을 전환하는 내비게이션을 관리하기 위한 기능을 의미한다. 일반적으로 사용자자 요청한 URL 또는 이벤트를 해석하고 새로운 페이지로 전환하기 위한 데이터를 취득하기 위해 서버에 필요 데이터를 요청하고 화면을 전환하는 위한 일련의 행위를 말한다.

AJAX 요청에 의해 서버로부터 데이터를 응답받아 화면을 생성하는 경우, 브라우저의 주소창의 URL은 변경되지 않는다. 이는 사용자의 방문 history를 관리할 수 없음을 의미하며, SEO(검색엔진 최적화) 이슈의 발생 원인이기도 하다. history 관리를 위해서는 각 페이지는 브라우저의 주소창에서 구별할 수 있는 유일한 URL을 소유하여야 한다.

이때 서버는 html로 화면을 표시하는데 부족함이 없는 완전한 리소스를 클라이언트에 응답한다. 이를 서버 렌더링이라 한다. 브라우저는 서버가 응답한 html을 수신하고 렌더링한다. 이때 이전 페이지에서 수신된 html로 전환하는 과정에서 전체 페이지를 다시 렌더링하게 되므로 새로고침이 발생한다.

전통적 링크 방식은 현재 페이지에서 수신된 html로 화면을 전환하는 과정에서 전체 페이지를 새로 렌더링하게 되므로 새로고침이 발생한다. 간단한 웹페이지라면 문제될 것이 없겠지만 복잡한 웹페이지의 경우, 요청마다 중복된 HTML과 CSS, JavaScript를 매번 다운로드해야하므로 속도 저하의 요인이 된다.

전통적 링크 방식의 단점을 보완하기 위해 등장한 것이 AJAX(Asynchronous JavaScript and XML)이다. AJAX는 자바스크립트를 이용해서 비동기적(Asynchronous)으로 서버와 브라우저가 데이터를 교환할 수 있는 통신 방식을 의미한다.

내비게이션이 클릭되면 link tag의 기본 동작을 prevent하고 AJAX를 사용하여 서버에 필요한 리소스를 요청한다. 요청된 리소스가 응답되면 클라이언트에서 웹페이지에 그 내용을 갈아끼워 html을 완성한다.

이를 통해 불필요한 리소스 중복 요청을 방지할 수 있다. 또한 페이지 전체를 새로 렌더링할 필요가 없고 갱신이 필요한 일부만 로드하여 갱신하면 되므로 빠른 퍼포먼스와 부드러운 화면 표시 효과를 기대할 수 있으므로 새로고침이 없는 보다 향상된 사용자 경험을 구현할 수 있다는 장점이 있다.

AJAX는 URL을 변경시키지 않으므로 주소창의 주소가 변경되지 않는다. 이는 브라우저의 뒤로가기, 앞으로가기 등의 history 관리가 동작하지 않음을 의미한다. 물론 코드 상의 history.back(), history.go(n) 등도 동작하지 않는다. 새로고침을 클릭하면 주소창의 주소가 변경되지 않기 때문에 언제나 첫페이지가 다시 로딩된다. 하나의 주소로 동작하는 AJAX 방식은 SEO 이슈에서도 자유로울 수 없다.

AJAX 방식은 불필요한 리소스 중복 요청을 방지할 수 있고, 새로고침이 없는 사용자 경험을 구현할 수 있다는 장점이 있지만 history 관리가 되지 않는 단점이 있다. 이를 보완한 방법이 **Hash** 방식이다.

Hash 방식은 URI의 fragment identifier(#service)의 고유 기능인 앵커(anchor)를 사용한다. fragment identifier는 hash mark 또는 hash라고 부르기도 한다. 위 예제를 살펴보면 link tag(**Service** 등)의 href 어트리뷰트에 hash를 사용하고 있다. 즉, 내비게이션이 클릭되면 hash가 추가된 URI가 주소창에 표시된다. 단, URL이 동일한 상태에서 hash가 변경되면 브라우저는 서버에 어떠한 요청도 하지 않는다. 즉, hash는 변경되어도 서버에 새로운 요청을 보내지 않으며 따라서 페이지가 갱신되지 않는다. hash는 요청을 위한 것이 아니라 fragment identifier(#service)의 고유 기능인 앵커(anchor)로 웹페이지 내부에서 이동을 위한 것이기 때문이다.

또한 hash 방식은 서버에 새로운 요청을 보내지 않으며 따라서 페이지가 갱신되지 않지만 페이지마다 고유의 논리적 URL이 존재하므로 history 관리에 아무런 문제가 없다.

JavaScript의 구현은 아래와 같다.

```
(function () {
    const root = document.querySelector('.app-root');

    function render(data) {
        const json = JSON.parse(data);
        root.innerHTML = `<h1>${json.title}</h1><p>${json.content}</p>`;
    }

    function renderHtml(html) {
        root.innerHTML = html;
    }

    function get(url) {
        return new Promise((resolve, reject) => {
            const req = new XMLHttpRequest();
            req.open('GET', url);
            req.send();

            req.onreadystatechange = function () {
                if (req.readyState === XMLHttpRequest.DONE) {
                    if (req.status === 200) resolve(req.response);
                    else reject(req.statusText);
                }
            }
        })
    }
})()
```

```

    };
  });
}

const routes = {
  '' : function () {
    get('/data/home.json')
      .then(res => render(res));
  },
  'service' : function () {
    get('/data/service.json')
      .then(res => render(res));
  },
  'about' : function () {
    get('/data/about.html')
      .then(res => renderHtml(res));
  },
  otherwise() {
    root.innerHTML = `${location.hash} Not Found`;
  }
};

function router() {
  // url의 hash를 취득
  const hash = location.hash.replace('#', '');
  (routes[hash] || routes.otherwise)();
}

// 내비게이션을 클릭하면 uri의 hash가 변경된다. 주소창의 uri가 변경되므로 history 관리가 가능하다.
// 이때 uri의 hash만 변경되면 서버로 요청을 수행하지 않는다.
// 따라서 uri의 hash가 변경하면 발생하는 이벤트인 hashchange 이벤트를 사용하여 hash의 변경을 감지하
// hash 방식의 단점은 uri에 불필요한 #이 들어간다는 것이다.
window.addEventListener('hashchange', router);

// DOMContentLoaded은 HTML과 script가 로드된 시점에 발생하는 이벤트로 load 이벤트보다 먼저 발생
// 새로고침이 클릭되었을 때, 웹페이지가 처음 로딩되었을 때, 현 페이지(예를들어 localhost:5003/#serv
window.addEventListener('DOMContentLoaded', router);
}());

```

hash 방식은 uri의 hash가 변경하면 발생하는 이벤트인 hashchange 이벤트를 사용하여 hash의 변경을 감지하여 필요한 AJAX 요청을 수행한다.

hash 방식의 단점은 uri에 불필요한 #이 들어간다는 것이다. 일반적으로 hash 방식을 사용할 때 #!을 사용하기도 하는데 이를 해시뱅(Hash-bang)이라고 부른다.

hash 방식은 과도기적 기술이다. HTML5의 Histroy API인 pushState가 모든 브라우저에서 지원이 된다면 해시뱅은 사용하지 않아도 되지만 현재 pushState는 일부의 브라우저(IE 10 이상)에서만 지원이 가능하다.

또 다른 문제는 SEO 이슈이다. 크롤러는 검색엔진이 웹사이트의 콘텐츠를 수집하기 위해 HTTP 1.1과 URL 스펙(RFC-2396같은)을 따른다. 이러한 크롤러는 JavaScript를 실행시키지 않기 때문에 hash 방식으로 만들어진 사이트의 콘텐츠를 수집할 수 없다. 구글은 해시뱅을 일반 URL을 변경시켜 이 문제를 해결한 것으로 알려지지만 다른 검색 엔진은 hash 방식으로 만들어진 사이트의 콘텐츠를 수집할 수 없다. hash 방식의 가장 큰 단점은 SEO 이슈이다. 이를 보완한 방법이 HTML5의 Histroy API인 pushState와 popstate 이벤트를 사용한 PJAX 방식이다. pushState와 popstate은 IE 10 이상에서 동작한다.

pushState의 예

```
container.addEventListener('click', function(e){
  if(e.target != e.currentTarget) {
    e.preventDefault();
    var data = e.target.getAttribute('data-name'),
        url = data + ".php";
    addCurrentClass(data);
    history.pushState(data, null, url);
    updateText(data);
    requestContent(url);
    document.title = "Ghostbuster | " + data;
  }
  e.stopPropagation();
}, false);
```

Promise와 callback

두 패턴 모두 Continuation Passing Style(CPS) 방식으로 Promise 패턴이 Promise 객체를 넘기는 것과 달리 Callback 패턴은 다음 할 일을 계속 Callback함수를 인자로 넘깁니다.

Callback 사용시 Client-side JavaScript에서는 비교적 로직이 적어 Callback 사용이 도움이 될 때도 있지만 business logic을 가진 Server-side 언어로 사용시 코드의 가독성을 떨어트리고, 디버그를 어렵게(hard to debug) 만듭니다.

보통 Callback Hell을 해결할 방법으로 Promise를 소개하는 경우가 많은데 엄밀히 말하면 Callback Hell을 해결할 수 없고 일부를 완화하는 것이다. Callback Hell을 완화할 수 있는 이유는 단일 인터페이스와 명확한 비동기 시점 표현, 강력한 에러 처리 메커니즘 때문이다. 하지만 이것은 Callback Hell 뿐만 아니라 비동기 처리 자체를 손쉽게 다룰 수 있도록 하는 것이므로 Callback Hell 해결하는 방법으로 여기는건 바람직하지 않다. 앞에서 Callback을 Promise 패턴으로

중첩되지 않는 형태로 변환했는데 결국 Promise 체인을 길게 연결 한다면 외형(가독성↑)만 다를 뿐 Callback Hell 문제 해결과는 큰 차이가 없다.

Promise는 미래 어느 시점이 되면 값을 채워주기로 약속한 빈 그릇이며 비동기 처리를 추상화한 추상 컨테이너이다. 즉, 통일된 인터페이스로 데이터를 전달할 수 있는 컨테이너로써 장점을 발휘하는 것이다. 본질적으로 전통적인 Callback과 Promise 두 패턴 모두 해결하고자 하는 문제는 비동기 처리를 손쉽게 다루기 위함이다. 비동기 처리를 다루는 방법이 두 가지이지 모든 경우 Promise로 프로그래밍을 할 수 있다고 생각하면 안 된다. 이벤트 리스너, Stream처럼 정기적, 지속적으로 비동기 처리가 필요한 경우 Promise를 사용하면 오히려 이상적인 결과를 얻을 수 없고 강력한 에러 처리 메커니즘이 독이 되는 경우가 발생한다.

Promise는 비동기적으로 대기(Pending) / 성공(Fulfilled, resolve) / 실패(Rejected, reject)를 다루는 값입니다. ES6+에서는 Promise와 관련하여 `new Promise`, `then`, `catch`, `race`, `Promise.all`, `Promise.resolve`, `Project.reject` 등을 지원합니다. Promise는 보통 소개된 것보다 훨씬 많은 가능성을 지닌 값이며, 자바스크립트에서의 동시성/비동기 프로그래밍을 지탱하는 기반입니다. ES6+에서는 `*/yield`, `async/await`와 함께 사용될 수 있습니다.

자바스크립트에서 비동기 프로그래밍은 매우 중요하며, bluebird, js-csp, co, RxJS 등의 다양한 비동기 해법들이 제시되고 있습니다.

불변객체

Javascript의 원시 타입(primitive data type)은 변경 불가능한 값(immutable value)입니다.

`Boolean` / `null` / `undefined` / `Number` / `String` / `Symbol` (New in ECMAScript 6)

하지만 객체의 경우 주소값이 할당되고 그 주소값은 변하지 않지만 값은 변할 수 있습니다. 그리고 그 값이 바뀌게 되면 본객체의 값도 바뀌게 됩니다. 모든게 다 주소값을 참조형태의 사본으로 할당이 되기 때문입니다. 이를 막기위해서 `object.freeze()` 가 있습니다. `const` 는 원시 타입의 경우에만 막아줍니다.

하지만 `object.freeze()` 도 2단계이상 깊이의 객체를 막아주지 못합니다.

이를 위해 본객체를 완전히 복사해서 새로운 주소값에 할당을 해서 변경하는 것이 필요합니다.

1단계 복사에는 `Object.assign` 가 있습니다. 1단계복사를 `shallow copy` 라고 부르기도 합니다.

`Object.assign` 은 타깃 객체로 소스 객체의 프로퍼티를 복사한다. 이때 소스 객체의 프로퍼티와 동일한 프로퍼티를 가진 타깃 객체의 프로퍼티들은 소스 객체의 프로퍼티로 덮어쓰기된다. 리턴값으로 타깃 객체를 반환한다. ES6에서 추가된 메소드이며 Internet Explorer는 지원하지 않습니다. `Object.assign` 은 아래처럼 합병할때도 쓰입니다.

```
// Copy
const obj = { a: 1 };
const copy = Object.assign({}, obj);
console.log(copy); // { a: 1 }
console.log(obj == copy); // false
```

```
// Merge
const o1 = { a: 1 };
const o2 = { b: 2 };
const o3 = { c: 3 };

const merge1 = Object.assign(o1, o2, o3);

console.log(merge1); // { a: 1, b: 2, c: 3 }
console.log(o1);    // { a: 1, b: 2, c: 3 }, 타겟 객체가 변경된다!

// Merge
const o4 = { a: 1 };
const o5 = { b: 2 };
const o6 = { c: 3 };

const merge2 = Object.assign({}, o4, o5, o6);

console.log(merge2); // { a: 1, b: 2, c: 3 }
console.log(o4);    // { a: 1 }
```

`const o4 = Object.create(o3);` 이렇게 `create`를 써서도 가능합니다. `assign`과 같이 1단계 shallow copy밖에 되지 않습니다. 완전히 복사하려면 `JSON.parse(JSON.stringify(o))` 가 가장 좋으며

아예 내부 객체까지 변경 불가능하게 만들려면 `Deep freeze`라는 재귀함수를 사용해서 각각의 Obj마다 `freeze`를 해주면 됩니다.

```
function deepFreeze(obj) {
  for(let name of Object.keys(obj)){
    const prop = obj[name];
    if (typeof prop === 'object' && prop !== null) {
      deepFreeze(prop);
    }
  }
  return Object.freeze(obj);
}

const user = {
  name: 'Lee',
  address: {
    city: 'Seoul'
  }
}
```

```

};

deepFreeze(user);

user.name = 'Kim'; // 무시된다
user.address.city = 'Busan'; // 무시된다

console.log(user); // { name: 'Lee', address: { city: 'Seoul' } }

```

라이브러리로는 Immutable.js를 이용해서 하면 됩니다.

Object

자바스크립트에서의 Array, Map, WeakMap, Set, WeakSet, Promise 등은 모두 **object** 의 하위 타입입니다.

```

log(typeof {}); // "object"
log(typeof []); // "object"
log(typeof new Map); // "object"
log(typeof new Set); // "object"
log(typeof new Promise(a => a)); // "object"

```

JSON 데이터 타입

JSON 데이터 타입은 다음과 같습니다.

- string
- number
- object
- array
- true, false, null

JSON은 경량의 DATA-교환 형식이면서, 현대 프로그래밍에서 가장 많이 사용되는 DATA-교환 언어입니다. 위 값에 해당하지 않는 모든 값은 JSON으로 직렬화가 불가능합니다. 예를 들면 **undefined** 도 JSON으로 직렬화할 수 없습니다. 또한 Map이나 Set 역시 JSON으로 직렬화할 수 없습니다.

```

JSON.stringify(undefined);
// undefined <--- 이건 JSON String이 된 것이 아니라 함수가 실패해서 undefined가 리턴된 것입니다.

```

```
JSON.stringify({ a: undefined });
// "{}"

JSON.stringify([undefined]);
// [null]
JSON.stringify(new Set([1,2,3]));
// {}
JSON.stringify(new Map([[ 'a' , 1]]));
// {}
```

이처럼 자바스크립트에서는 JSON으로 직렬화할 수 있는 값과 그렇지 않은 값으로, 다시 두 가지로 나누어 데이터를 바라볼 수 있습니다.

Iterable, Iterator

ES6에는 Iterable/Iterator에 대한 프로토콜이 있고, 그것은 ES6에서 매우 중요합니다. 자바스크립트에서 Iterable, Iterator, Symbol.iterator, Generator 등은 자바스크립트의 `for...of` 문이나 전개 연산자 등과 사용됩니다. 이것들은 ES6에서 함수형 프로그래밍을 하는 것에 있어서도 아주 중요한 역할을 합니다. ES6의 String, Array, Map, Set 등은 `for of` 또는 `[...t]`로 배열을 만들 수 있습니다. 아래와 같이 사용 가능하도록 Symbol.iterator 프로퍼티를 가진 객체입니다.

이터레이터와 이터레이블

1. iterable은 `iterable[Symbol.iterator]()`; 메서드를 가져야하고, 결과는 iterator여야합니다.
2. iterator는 `iterator.next()` 메서드를 가져야하고, 결과는 `{ value: someting, done: true/false }` 여야 합니다.

```
const obj = {
  [Symbol.iterator]: function() {
    return {
      cur: 0,
      next: function() {
        if (this.cur > 5) return { value: undefined, done: true }
        return {
          value: ++this.cur,
          done: false
        }
      }
    }
  }
}
```

```

        }
    },
    [Symbol.iterator]: function() {
        return this;
    }
}
};

function *reverseIter(arr) {
    var l = arr.length;
    while (l--) yield arr[l];
}

```

iterator[Symbol.iterator]() == iterator , 그리고 Generator

3. iterator도 `iterator[Symbol.iterator]();` 를 갖도록 하고, `iterator[Symbol.iterator]() == iterator` 라면 더욱 잘 만들어진 iterator라고 할 수 있습니다.

4. 자바스크립트의 내장 iterables의 'iterator'를 리턴하는 메서드'를 실행하여 얻은 iterator는 모두 `iterator[Symbol.iterator]() == iterator` 와 같은 규약을 지키고 있습니다.

5. Generator를 통해 만든 iterator 역시 `iterator[Symbol.iterator]() == iterator` 을 지킵니다.

3, 4, 5를 잘 응용하면 `for...of` 와 잘 동작하는 것을 넘어, 데이터 순회에 대한 매우 재밌고 새로운 해법들을 만들 수 있어 매우 중요합니다.

유사 배열

유사 배열은 `{ 0: 10, 1: 20, length: 2 }` 와 같은 객체를 말합니다. 다음과 같은 유사 배열은 전개 연산자나 `for...of` 와 사용될 수 없습니다.

```

const arrayLike = { 0: 10, 1: 20, length: 2 };
for (const val of arrayLike) log(val);

```

물론 유사 배열을 순회할 수 있게 하는 generator를 만든다면 `for...of` 에서 사용할 수 있겠지만, ES6부터는 iterable이 아닌 유사 배열을 사용하지 않는 방향으로 가야합니다. 자바스크립트 3rd party library의 값들 중에는 아직 iterable/iterator 프로토콜을 따르지 않는 유사 배열도 있지만, 모두 ES6의 프로토콜에 맞게 변경될 것입니다.

대표적인 유사 배열로는 arguments, NodeList 등이 있습니다. 최신 환경에서 해당 값들은 iterable이 되었습니다.

[Symbol.iterator]() 가 구현되어있어, `for...of` 나 전개 연산자와 사용할 수 있습니다.

호이스팅

모든 함수, 변수를 선언 될 때 스코프의 가장 상단으로 가는 것을 말합니다. let, const로 선언해도 일어납니다. 안 일어나는 것이 아닙니다. TDZ로 var처럼 선언 & 초기화 동시발생을 막았을 뿐.

스코프 : 변수, 매개변수의 접근성, 생존기간을 말합니다.

var로 선언하면 선언 & 초기화가 동시에 일어나기 때문에 undefined로 접근할 수 있지만 const / let으로 선언하게 되면 선언 TDZ 초기화로 TDZ가 막기 때문에 Reference Error로 호이스팅이 일어나지만 호이스팅을 막을 수 있습니다.

스코프

지역스코프와 전역 스코프로 나눠진다. 함수밖이나 {}밖은 모두 전역스코프라고 부르며 그 외는 지역스코프라고 부릅니다. 이는 브라우저에만 속한 설명이며 node.js안에서는 다릅니다.

node.js안에서의 전역스코프는 global이라는 변수에 할당해서 씁니다.

지역스코프는 함수 / 블록으로 나눠진다. 함수 레벨 스코프(Function-level scope) 함수의 코드 블록만을 스코프로 인정한다. 따라서 전역 함수 외부에서 생성한 변수는 모두 전역 변수이다.

참고로 for 문의 변수 선언문에서 선언한 변수를 for 문의 코드 블록 외부에서 참조할 수 있다.

변수 호이스팅

변수를 선언하기 이전에 참조할 수 있다.

```
a = 3;
console.log(a);
var a;
//3
console.log(a);
var a = 3;
// undefined
```

함수 호이스팅

자바스크립트는 선언만 호이스팅이 되고 할당, 초기화는 호이스팅이 되지 않는다.

```

var arr = [1, 2, 3, 4]
function a(){
    function b(val){
        i = 10
        console.log(i)
        return val * i
    }
    for(var i = 0; i < arr.length; i++){
        arr[i] = b(arr[i])
    }
    return arr;
}
console.log(a())
//10
//[ 10, 2, 3, 4 ]

```

반복문 안의 루프 카운터 `i`가 호이스팅이 되어 내부함수 `b`의 클로저에 포함이 됩니다. 이 때 `i`를 변경해서 루프카운터까지 변경되게 되어서 꼬이게 됩니다.

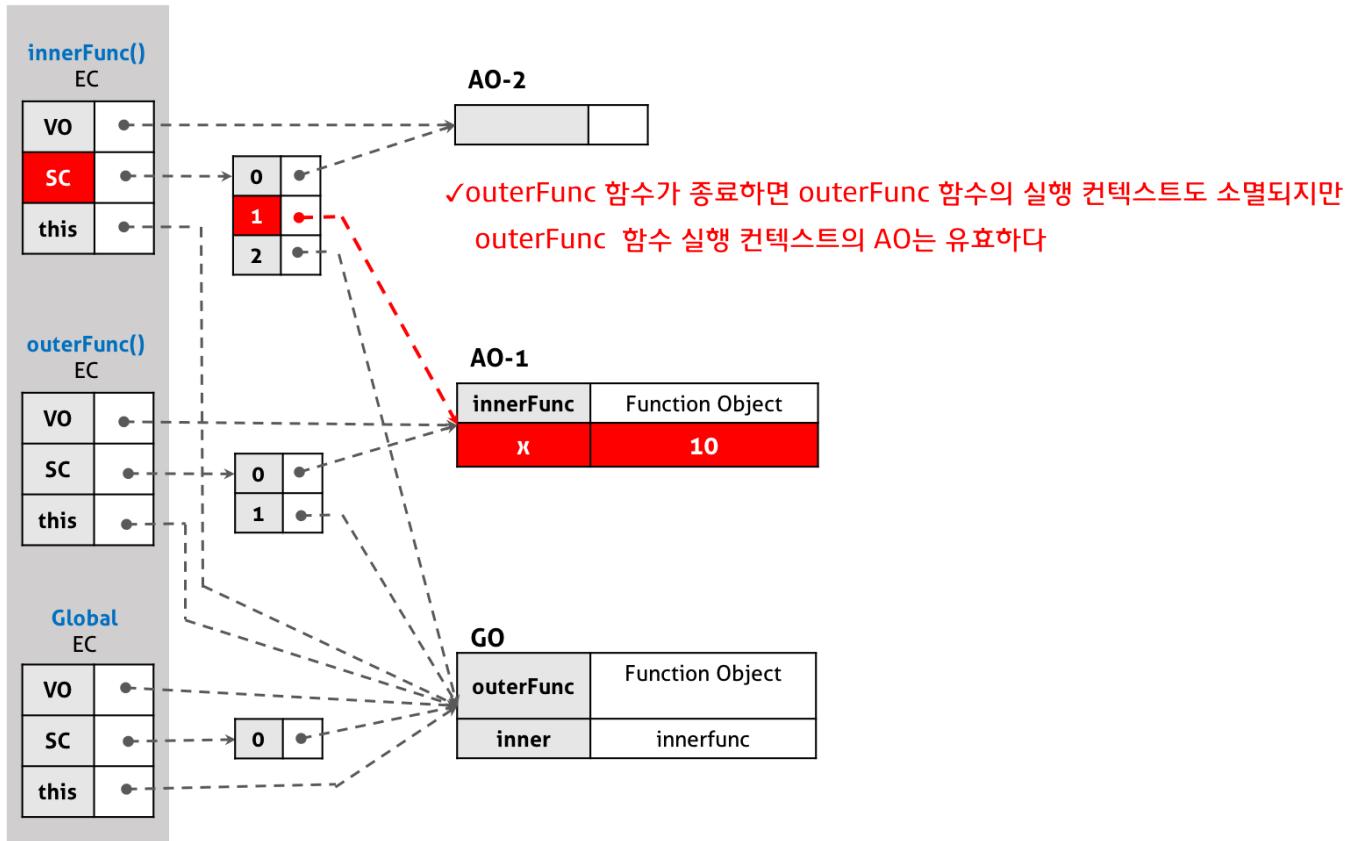
ES6는 위에처럼 함수레벨 스코프에서 일어나는 현상들을 방지하기 위해 `var` 키워드의 단점을 보완하기 위해 `let`과 `const` 키워드를 도입하였습니다. 다시 말하지만 호이스팅은 되지만 블록스코프 / 선언과 초기화를 분리하는 TDZ로 함수레벨스코프 단계에서 일어나는 호이스팅으로 일어나는 버그들을 막는 것입니다.

테스트 테스트



`let` 키워드로 선언된 변수는 선언 단계와 초기화 단계가 분리되어 진행된다. 즉, 스코프에 변수를 등록(선언단계)하지만 초기화 단계는 변수 선언문에 도달했을 때 이루어진다. 초기화 이전에 변수에 접근하려고 하면 참조 에러(ReferenceError)가 발생한다. 이는 변수가 아직 초기화되지 않았기 때문이다. 다시 말하면 변수를 위한 메모리 공간이 아직 확보되지 않았기 때문이다. 따라서 스코프의 시작 지점부터 초기화 시작 지점까지는 변수를 참조할 수 없다. 스코프의 시작 지점부터 초기화 시작 지점까지의 구간을 ‘일시적 사각지대(Temporal Dead Zone; TDZ)’라고 부른다.

클로저



스코프는 함수를 호출할 때가 아니라 함수를 어디에 선언하였는지에 따라 결정됩니다. 즉, "렉시컬한 환경"으로 설정이 됩니다. 이 때문에 함수가 다른 함수 내부에서 정의되었을 때 내부함수는 외부함수의 변수에 접근 가능하지만 외부함수는 내부함수의 변수에 접근 불가한 "렉시컬 스코핑"이 발생되게 됩니다.

클로저는 함수와 그 함수가 선언된 렉시컬 환경의 조합이며 외부 함수가 반환된 후에도 외부 함수의 변수 범위 체인에 접근할 수 있는 함수입니다.

클로저는 각자의 환경을 가진다. 이 환경을 기억하기 위해서는 당연히 메모리가 소모될 것이다. 클로저를 생성해놓고 참조를 제거하지 않는 것은 C++에서 동적 할당으로 객체를 생성해놓고 delete를 사용하지 않는 것과 비슷하다. 클로저를 통해 내부 변수를 참조하는 동안에는 내부 변수가 차지하는 메모리를 GC가 회수하지 않는다. 따라서 클로저 사용이 끝나면 참조를 제거하는 것이 좋다.

```
// 함수를 인자로 전달받고 함수를 반환하는 고차 함수
// 이 함수가 반환하는 함수는 클로저로서 카운트 상태를 유지하기 위한 자유 변수 counter를 기억한다.
function makeCounter(predicate) {
    // 카운트 상태를 유지하기 위한 자유 변수
    var counter = 0;
    // 클로저를 반환
    return function () {
        counter = predicate(counter);
        return counter;
    };
}
```

```

}
```

```

// 보조 함수
function increase(n) {
    return ++n;
}

```

```

// 보조 함수
function decrease(n) {
    return --n;
}

```

```

// 함수로 함수를 생성한다.
// makeCounter 함수는 보조 함수를 인자로 전달받아 함수를 반환한다
const increaser = makeCounter(increase);
console.log(increaser()); // 1
console.log(increaser()); // 2

```

```

increaser = null // 로 할당을 해제하면 된다.
// increaser 함수와는 별개의 독립된 렉시컬 환경을 갖기 때문에 카운터 상태가 연동하지 않는다.
const decreaser = makeCounter(decrease);
console.log(decreaser()); // -1
console.log(decreaser()); // -2

```

이벤트 캡처링 버블링

이벤트 버블링 - 하위 엘리먼트에서 상위 엘리먼트로 이벤트가 전파되는 특성

```

<body>
    <div class="one">
        <div class="two">
            <div class="three">
                </div>
            </div>
        </div>
    </body>
    <script>

        var divs = document.querySelectorAll('div');

```

```

divs.forEach(function(div) {
    div.addEventListener('click', logEvent);
});

function logEvent(event) {
    console.log(event.currentTarget.className);
}

</script>

```

이벤트 캡처링 - 버블링의 반대방향으로 진행

```

<body>
    <div class="one">
        <div class="two">
            <div class="three">
            </div>
        </div>
    </div>
</body>

<script>

var divs = document.querySelectorAll('div');
divs.forEach(function(div) {
    div.addEventListener('click', logEvent, {
        // 캡처링으로 설정. default는 false이므로 버블링으로 동작한다.
        capture: true
    });
});

function logEvent(event) {
    console.log(event.currentTarget.className);
}
</script>

```

call 과 apply , bind

.call 과 .apply 는 모두 함수를 호출하는데 사용되며 첫 번째 매개 변수는 함수 내에서 this 의 값으로 사용됩니다. 그러나 .call 은 쉼표로 구분된 인수를 두 번째 인수로 취하고 .apply 는 인수의 배열을 두 번째 인수로 취합니다.

니다. `call` 은 `C : Comma` 로 구분되며 `apply` 는 인수 배열인 `A : arguments` 라고 기억하면 쉽습니다.

js

```
function add(a, b) {
  return a + b;
}

console.log(add.call(null, 1, 2)); // 3
console.log(add.apply(null, [1, 2])); // 3
```

call

```
function greet() {
  var reply = [this.animal, 'typically sleep between', this.sleepDuration].join(' ')
  console.log(reply);
}

var obj = {
  animal: 'cats', sleepDuration: '12 and 16 hours'
};

greet.call(obj); // cats typically sleep between 12 and 16 hours
```

`greet`함수의 `this`는 `obj`를 가리키게 됩니다. 원하는 함수에 인자로 넘긴 `this`가 바인딩 된 새로운 함수를 리턴한다

유사배열의 경우 `Array.prototype.slice.call(arguments)` 이렇게 `slice`라는 함수를 빌려서 실행할 수도 있습니다.

bind

`this`의 값을 바꿀 수 있는 마지막 함수는 `bind`이다. `bind`를 사용하면 함수의 `this` 값을 영구히 바꿀 수 있습니다.

`slice`의 default parameters, `slice`는 인자가 아무것도 없으면 실행되는 문맥의 `this`를 배열로 바꿔 직접복사를 해서 반환합니다. 이를 `call`과 `bind`를 이용해서 기본값을 만들어줄 수 있습니다. `list`라는 함수에 `bind`를 통해 `this`에 아무것도 선언하지않고 기본값으로 37을 넣었습니다. `list`라는 함수는 이제 기본값 37을 갖는 유사배열의 메소드인 `slice`함수가 됩니다. 이로써 그냥 호출하게 되면 37이라는 배열이 반환되고 1, 2, 3을 추가해서 호출하면 37, 1, 2, 3 이렇게 기본값을 갖는 배열이 리턴되게 됩니다.

```

function list() {
    return Array.prototype.slice.call(arguments);
}

var list1 = list(1, 2, 3);
var leadingThirtysevenList = list.bind(undefined, 37);
var list2 = leadingThirtysevenList(); // [37]
var list3 = leadingThirtysevenList(1, 2, 3); // [37, 1, 2, 3]

```

또한, setTimeout의 callback의 this는 함수호출패턴에 의해 setTimeout이 호출되는 위치인 window 또는 global(Node.js)입니다. 따라서 setTimeout의 callback으로 오브젝트의 메소드를 주고, 그 메소드 내에서 오브젝트의 인스턴스를 this로 참조하려면, bind가 꼭 필요하다. (화살표 함수로 실행될 수 있다)

크로스도메인 CORS

웹 개발시 주요한 이슈중 하나로, 웹 개발을 하다보면 어떤 경로던 이 이슈를 마주하게 된다. 동일 출처 정책(same-origin-policy)은 하나의 웹 페이지에서 다른 도메인 서버에 요청하는 것을 제한하는 것이다. 제한하는 이유는 간단한데, 내가 네이버라고 가정해보자. 누군가 다른 포탈 서비스를 만들고, 네이버에서 검색한 결과만 가져온다면 문제가 되지 않을까? 때문에 보통의 브라우저에서는 외부 도메인으로의 Ajax로 요청을 보낼 때, cors를 체킹한다.

서버 : 헤더로 해결한다.

```
response.setHeader("Access-Control-Allow-Origin", "*");
```

- 는 모든 도메인에 대해 허용하겠다는 의미. 즉 어떤 웹사이트라도 이 서버에 접근하여 AJAX 요청하여 결과를 가져갈 수 있도록 허용하겠다는 의미. 만약 보안 이슈가 있어서 특정 도메인만 허용해야 한다면 * 대신 특정 도메인만을 지정할 수 있음.

서버 : cors라는 npm을 사용한다.

```

var express = require('express')
var cors = require('cors')
var app = express()

app.use(cors())

app.get('/products/:id', function (req, res, next) {
  res.json({msg: 'This is CORS-enabled for all origins!'})
})

```

```
app.listen(80, function () {
  console.log('CORS-enabled web server listening on port 80')
})
```

클라이언트 : JSONP (JSON with Padding)

`<script/>` 태그는 **same-origin-policy (SOP)** 정책에 속하지 않는다는 사실을 근거로, 서로 다른 도메인간의 javascript 호출을 한다.

```
var script = document.createElement('script');
script.src = '//kingbbode.com/jsonp?callback=parseResponse';
document.getElementsByTagName('head')[0].appendChild(script);
function parseResponse(data){
  //callback method
}
```

이와 유사한 것은 바로 Queing제거이다. `script > css > http` 요청을 처리하기 때문에 처음에 무조건 받아야 할 `http` 요청이 있다면 `http`요청으로 하지 않고 스크립트 로드 방식으로 하면 된다. 또한 `http`요청은 매번 TCP연결을 해야 한다. 예를 들어 `php`로 받는다면 `echo 'var a = '` 이런식으로 처리를 해서 로드하면 된다.

<https://developers.google.com/web/tools/chrome-devtools/network/understanding-resource-timing>

여러언어 페이지

HTTP 요청을 서버에 보내면, 대개 요청하는 유저 에이전트가 `Accept-Language` 헤더와 같은 기본 언어 설정에 대한 정보를 보냅니다. 그 다음 서버는 이 정보를 사용하여 해당 언어가 제공 가능한 경우, 해당 언어 버전의 문서를 반환할 수 있습니다. 반환된 HTML 문서는 `<html lang="en">...</html>` 과 같이 `<html>` 태그에 `lang` 속성을 선언해야 합니다. 백엔드에서, HTML 마크업은 YML 또는 JSON 형식으로 저장된 특정 언어에 대한 `i18n` placeholder와 내용을 포함합니다. 그 다음 서버는 일반적으로 백엔드 프레임워크의 도움을 받아 특정 언어로 HTML 페이지를 동적 생성합니다.

null undefined 선언되지 않은 변수의 차이점

선언되지 않은 변수 변수는 이전에 `var`, `let`, `const` 를 사용하여 생성되지 않은 식별자에 값을 할당할 때 생성됩니다. **선언되지 않은 변수** 는 현재 범위 외부에서 전역으로 정의됩니다. strict 모드에서는 **선언되지 않은 변수** 에 할당하려고 할 때 `ReferenceError` 가 throw 됩니다. **선언되지 않은 변수** 는 전역 변수처럼 좋지 않은 것입니다. 그들은 모두 피하세요! 이들을 검사하기 위해 사용할 때 `try` / `catch` 블록에 감싸십시오.

```

function foo() {
  x = 1; // strict 모드에서 ReferenceError를 발생시킵니다.
}

foo();
console.log(x); // 1

```

`undefined` 변수는 선언되었지만 값이 할당되지 않은 변수입니다. 이것은 `undefined` 타입입니다. 함수가 실행 결과에 따라 값을 반환하지 않으면 변수에 할당되며, 변수가 `undefined` 값을 갖습니다. 이것을 검사하기 위해, 엄격한 (`==`) 연산자 또는 `typeof` 에 `undefined` 문자열을 사용하여 비교하십시오. 확인을 위해 추상 동등 연산자 (`==`)를 사용해서는 안되며, 이는 값이 `null` 이면 `true` 를 반환합니다.

```

var foo;
console.log(foo); // undefined
console.log(foo === undefined); // true
console.log(typeof foo === 'undefined'); // true

console.log(foo == null); // true. 옳지 않습니다. 확인하는 데 사용하지 마세요.

function bar() {}
var baz = bar();
console.log(baz); // undefined

```

`null` 인 변수는 `null` 값에 명시적으로 할당될 것입니다. 그것은 값을 나타내지 않으며 명시적으로 할당된다는 점에서 `undefined` 와 다릅니다. `null` 을 체크하기 위해서 단순히 완전 항등 연산자(`==`)를 사용하여 비교하면 됩니다. 위와 같이, 추상 동등 연산자 (`==`)를 사용해서는 안되며, 값이 `undefined` 이면 `true` 를 반환합니다.

```

var foo = null;
console.log(foo === null); // true

console.log(foo == undefined); // true. 옳지 않습니다. 확인하는 데 사용하지 마세요.

```

개인적 습관으로, 저는 변수를 선언하지 않거나 할당하지 않은 상태로 두지 않습니다. 아직 사용하지 않으려는 경우, 선언한 후에 명시적으로 `null` 을 할당할 것입니다.

자바스크립트 원시 / 참조타입 Number

원시 : Number, String, Boolean, Null, Undefined, Symbol Symbol의 예

Number

자바스크립트에서의 수 표현 자바스크립트에서는 정수와 실수를 따로 구분하지 않고, 모든 수를 실수 하나로만 표현합니다. 자바스크립트에서 모든 숫자는 IEEE 754 국제 표준에서 정의한 64비트 부동 소수점 수로 저장됩니다. 64비트 부동 소수점 수(double precision floating point numbers)는 메모리에 다음과 같은 형태로 저장됩니다.

0 ~ 51 비트	52 ~ 62 비트	63 비트
총 52비트의 가수 부분	총 11비트의 지수 부분	총 1비트의 부호 부분

이러한 64비트 부동 소수점 수의 정밀도는 정수부는 15자리까지, 소수부는 17자리까지만 유효합니다. 부동소수점은 $a * 2^n$ 형식으로 표현되는 수이며 2의 지수 형태로 연산되기 때문에 정확도가 떨어집니다.

다음 예제는 64비트 부동 소수점 수의 정밀도를 알아보는 예제입니다.

```
var x = 999999999999999; // 15자리의 정수부
var y = 999999999999999; // 16자리의 정수부
var z = 0.1 + 0.2
x; // 999999999999999
y; // 1000000000000000
z; // 0.3000000000000004
```

위의 예제에서 변수 z의 값을 살펴보면 오차가 발생했음을 알 수 있습니다. 이렇게 부동 소수점 수를 가지고 수행하는 산술 연산의 결괏값은 언제나 오차가 발생할 가능성을 가지고 있습니다. 이것은 자바스크립트만의 문제가 아닌 부동 소수점 수를 가지고 실수를 표현하는 모든 프로그래밍 언어에서의 문제점입니다. 자바스크립트에서는 이러한 오차를 없애기 위해 정수의 형태로 먼저 변환하여 계산을 수행하고, 다시 실수의 형태로 재변환하는 방법을 사용합니다.

```
var z = (0.2 * 10 + 0.1 * 10) / 10; // 0.3
```

호스트 객체와 내장 객체의 차이점

내장 객체는 ECMAScript 사양에 정의된 JavaScript 언어의 일부인 객체입니다. (예: `String`, `Math`, `RegExp`, `Object`, `Function` 등) 호스트 객체는 `window`, `XMLHttpRequest` 등과 같이 런타임 환경 (브라우저 또는 노드)에 의해 제공됩니다.

실행 컨텍스트 Execution Context

코드 실행에 필요한 정보들을 물리적인 객체로 관리하는 것이 EC입니다. 이 EC들이 Call Stack에 쌓여서 순차적으로 실행이 됩니다.

초기에는 Global Object인 빌트인 객체(Math, String, Array 등)와 BOM, DOM이 있지만 추후 EC들이 쌓여서 실행 됩니다. EC는 Variable Object, this, 스코프체인으로 구성됩니다.

Variable Object

- Activation Object : 함수선언(표현식은제외), Arguments, 변수를 포함합니다.
- Global Object : 전역변수들을 포함합니다.

Variable Object가 형성되는 과정

- (Function Code인 경우) 매개변수(parameter)가 Variable Object의 프로퍼티로, 인수(argument)가 값으로 설정된다.
- 대상 코드 내의 함수 선언(함수 표현식 제외)을 대상으로 함수명이 Variable Object의 프로퍼티로, 생성된 함수 객체가 값으로 설정된다.(함수 호이스팅)
- 대상 코드 내의 변수 선언을 대상으로 변수명이 Variable Object의 프로퍼티로, undefined가 값으로 설정된다.(변수 호이스팅)

this

함수 호출 패턴에 따라 달라집니다.

스코프체인

GO, 함수의 스코프 주소를 차례대로 담은 리스트입니다. 순차적으로 탐색하며 해당 주소, 즉 [[scope]]로 참조가 가능합니다. 현재 실행 컨텍스트의 활성 객체(AO)를 선두로 하여 순차적으로 상위 컨텍스트의 활성 객체(AO)를 가리키며 마지막 리스트는 전역 객체(GO)를 가리킵니다. 이렇게 변수를 찾는 과정을 스코프체이닝이라고 합니다. 참고로 프로퍼티를 찾는 것을 프로토타입체이닝이라고 합니다.

자바스크립트 최적화기법

- **<script>** - HTML 파싱이 중단되고, 스크립트를 즉시 가져오고 실행되며, 스크립트 실행 후 HTML 파싱이 다시 시작됩니다.
- **<script async>** - 이 스크립트는 HTML 파싱과 병렬적으로 가져오며, 가능할 때 즉시 실행됩니다(아마 HTML 파싱이 끝나기 전). 스크립트가 페이지의 다른 스크립트들과 독립적인 경우 **async** 를 사용하세요. 예) analytics.
- **<script defer>** - 이 스크립트는 HTML 파싱과 병렬적으로 가져오지만, 페이지 파싱이 끝나면 실행됩니다. 이것이 여러개 있는 경우, 각 스크립트는 페이지에 등장한 순서대로 실행됩니다. 스크립트가 완전히 파싱된 DOM에 의존되는 경우 **defer** 속성은 스크립트를 실행하기 전에 HTML이 완전히 파싱되도록 하는데 유용합니다. **<body>**

의 끝부분에 일반 `<script>` 를 두는 것과 별 차이가 없습니다. `defer` 스크립트는 `document.write` 를 포함하면 안됩니다.

이 외 CSS 최적화로는 다음과 같다.

- 압축 / 중첩된 태그x / short-hand / 선별적 하드웨어 가속 `transform:translateZ(0);`
- lazy 로딩: 이미지, 스크립트, CSS 파일들이 lazy 로드 되어서 현 페이지의 응답시간을 향상시킴

가상요소선택자, 가상클래스 선택자(CSS Pseudo code)

가상요소 선택자

CSS `Pseudo-element` 는 Selector 에 추가된 키워드로, 선택한 요소의 특정한 부분을 스타일링 할 수 있습니다. 마크업을 수정하지 않고 (`:before`, `:after`) 텍스트 데코레이션을 위해 사용하거나 (`:first-line`, `:first-letter`) 또는 마크업에 요소를 추가할 수 있습니다. (`content: ...` 와 결합) 툴팁의 삼각형 화살표는 `:before` 와 `:after` 를 사용해서 추가적인 HTML 요소를 사용하지 않고 CSS 스타일만으로 삼각형을 그립니다.

가상클래스 선택자

`:visited`, `:hover` 어떤 등 클래스가 존재하지 않지만 어떤 행위에 대해서 가상적인 클래스를 만들 때 사용됩니다.

cookie , sessionStorage , localStorage 사이의 차이점

위 세 가지 기술은 모두 클라이언트 측에서 값을 저장하는 key-value 저장소 매커니즘이다. 모두 문자열로만 값을 저장할 수 있습니다.

	cookie	localStorage	sessionStorage
생성자	클라이언트나 서버. 서버는 <code>Set-Cookie</code> 헤더를 사용할 수 있습니다	클라이언트	클라이언트
만료	수동으로 설정	영구적	탭을 닫을 때
브라우저 세션 전체에서 지속	만료 설정 여부에 따라 다름	O	X
모든 HTTP 요청과 함께 서버로 보냄	쿠키는 <code>Cookie</code> 헤더를 통해 자동 전송됨	X	X
용량 (도메인당)	4kb	5MB	5MB
접근성	모든 윈도우	모든 윈도우	같은 탭

이미지태그 `srcset`

기기의 디스플레이 너비에 따라 다른 이미지를 사용자에게 제공하려는 경우 `srcset` 속성을 사용합니다 - 레티나 디스플레이를 통해 장치에 고품질 이미지를 제공하여 사용자 경험을 향상시키고, 저해상도 이미지를 저사양 기기에 제공하여 성능을 높이고 데이터 낭비를 줄입니다. (왜냐하면 더 큰 이미지를 제공하는 것은 눈에 보일 정도의 차이가 없기 때문). 예를 들면: `` 는 클라이언트의 해상도에 따라 브라우저에 small, medium, large .jpg 그래픽을 표시하도록 지시합니다. 첫 번째 값은 이미지 이름이고 두 번째 값은 픽셀 단위의 이미지 너비입니다. 320px 너비의 경우, 다음과 같은 계산을 따릅니다

- $500 / 320 = 1.5625$
- $1000 / 320 = 3.125$
- $2000 / 320 = 6.25$

클라이언트의 해상도가 1x 일 경우, 1.5625가 가장 가깝고 `small.jpg` 에 해당하는 `500w` 가 브라우저에 의해 선택됩니다.

해상도가 레티나 (2x)인 경우 브라우저는 최소값에서 가장 위로 가까운 해상도를 사용합니다. `500w` (1.5625)는 1보다 크고 이미지가 보기 좋지 않을 수 있기 때문에 선택하지 않는다는 것을 의미합니다. 그래서 브라우저는 계산 결과 비율값이 2에 가까운 `1000w` (3.125) 이미지를 선택합니다.

`srcset` 는 데스크탑 디스플레이처럼 거대한 이미지를 필요로하지 않기 때문에 화면 장치를 좁히는 작은 이미지 파일을 제공하고자 하는 문제를 해결합니다.

`srcset` 는 화면이 작은 기기에서 데스크탑 디스플레이처럼 큰 이미지가 필요하지 않기 때문에 작은 이미지 파일을 제공하는 문제를 해결합니다 — 또한 선택적으로 고해상도/저해상도 화면에 다른 해상도 이미지를 제공할 수도 있습니다. 저는 레티나 디스플레이를 다루기 위해 고해상도 그래픽을 사용하는 경향이 있습니다. 가장 좋은 방법은 `@media only screen and (min-device-pixel-ratio : 2) {...}` 와 같은 미디어 쿼리를 사용하고 `background-image` 를 변경하는 것입니다.

아이콘의 경우 해상도에 관계없이 매우 선명하게 렌더링하므로 가능하면 `svg` 및 아이콘 글꼴을 사용하도록 선택합니다.

또 다른 방법으로는 `window.devicePixelRatio` 값을 확인한 후에 `img src` 특성을 더 높은 해상도 버전으로 대체하는 JavaScript를 사용하는 것이다.

CSS 선택자 특이성

브라우저는 CSS 규칙의 특수성에 따라 요소에 표시할 스타일을 결정합니다. 브라우저는 이미 특정 요소와 일치하는 규칙을 결정했다고 가정합니다. 일치하는 규칙들 가운데, 다음에 기초하여 각 규칙에 대해 특수성, 네개의 쉼표로 구분된 값, `a`, `b`, `c`, `d` 가 계산됩니다.

1. `a` 는 인라인 스타일이 사용되고 있는지 여부입니다. 속성 선언이 요소에서 인라인 스타일이면 '`a`'는 1이고, 그렇지 않으면 0입니다.
2. `b` 는 ID 셀렉터의 수입니다.
3. `c` 는 클래스, 속성 및 가상 클래스 선택자의 수입니다.

4. **d** 는 태그 및 유사 요소 선택자의 수입니다.

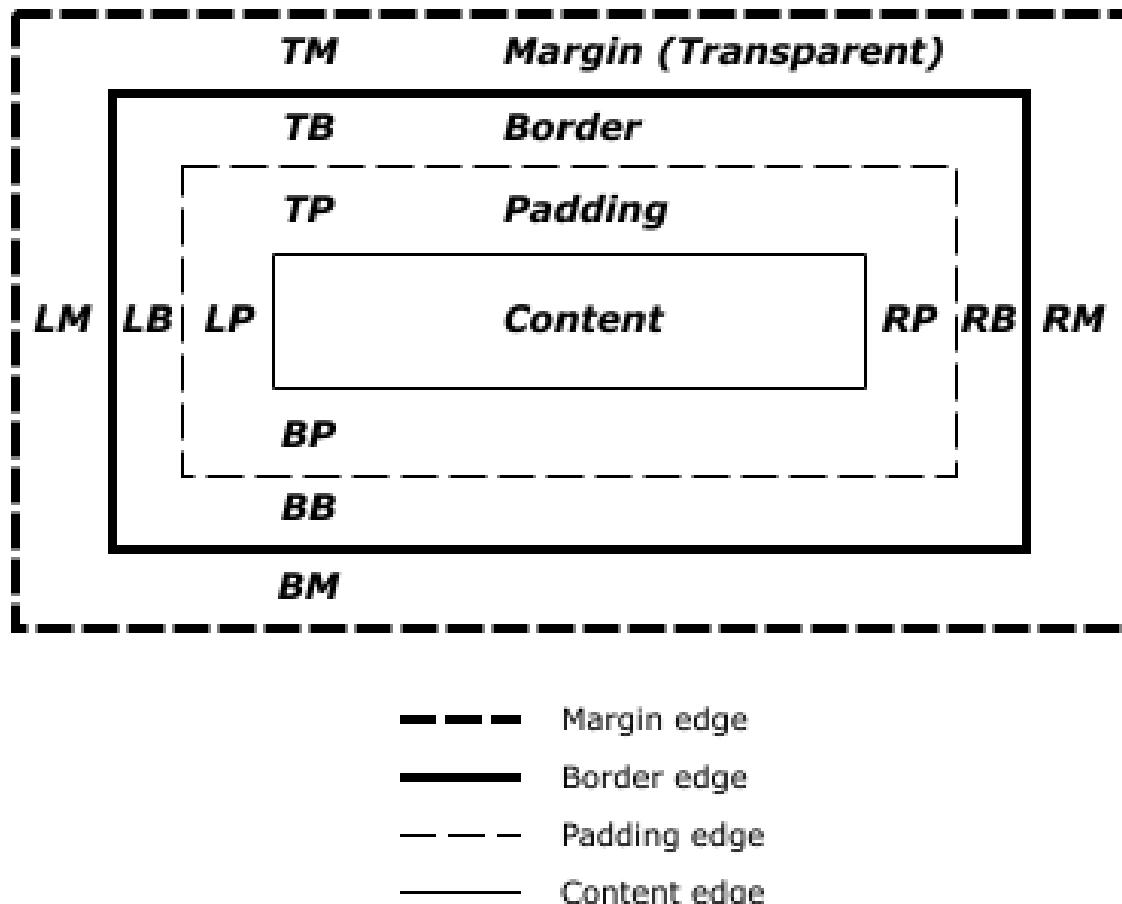
결과적인 특정성은 점수가 아니라, 컬럼마다 비교할 수 있는 가치들의 행렬입니다. 선택자를 비교하여 가장 높은 특이성을 갖는 항목을 결정할 때 왼쪽에서 오른쪽으로 보고 각 열의 가장 높은 값을 비교하세요. 따라서 **b** 열의 값은 **c** 와 **d** 열에 있는 값을 무시합니다. 따라서 **0, 1, 0, 0** 의 특이성은 **0, 0, 10, 10** 중 하나보다 큽니다.

동등한 특이성의 경우: 최신 규칙은 중요한 규칙입니다. 스타일 시트에 동일한 규칙을 두 번 작성한 경우(내부나 외부에 관계 없이) 스타일 시트의 하위 규칙이 스타일 될 요소에 더 가까우므로 더 구체적으로 적용됩니다.

필자는 필요하다면 쉽게 재정의할 수 있도록 낮은 특정성 규칙들을 작성할 것입니다. CSS UI 컴포넌트 라이브러리 코드를 작성할 때 특이성을 높이거나 **!important** 를 사용하기 위해 라이브러리 사용자가 지나치게 복잡한 CSS 규칙을 사용하지 않고도 이를 무시할 수 있도록 특이성이 낮은 것이 중요합니다.

컨테이닝박스(block)

요소들의 크기와 위치는 컨테이닝 블록(containing block)의 영향을 받곤 합니다. 대부분의 경우 어떤 요소의 컨테이닝 블록은 그 요소에 가장 가까운 블록 레벨 조상의 콘텐츠 영역이나, 항상 그런 것은 아닙니다. 이 글에서는 요소의 컨테이닝 블록을 결정하는 요인을 살펴보겠습니다.



많은 개발자들은 요소의 컨테이닝 블록이 언제나 부모 요소의 콘텐츠 영역이라고 생각하지만, 사실 꼭 그렇지는 않습니다. 어떤 항목이 컨테이닝 블록을 결정짓나 알아보겠습니다.

컨테이닝 블록의 효과

컨테이닝 블록을 결정하는 요인에 뭐가 있는지 알아보기 전에, 애초에 컨테이닝 블록이 무슨 상관인지 알아두는게 유용하겠습니다. 요소의 크기와 위치는 컨테이닝 블록의 영향을 자주 받습니다. 백분율 값을 사용한 width, height, padding, margin 속성의 값과 절대적 위치(absolute나 fixed 등)로 설정된 요소의 오프셋 속성 값은 자신의 컨테이닝 블록으로부터 계산됩니다.

컨테이닝 블록 식별 컨테이닝 블록의 식별 과정은 position 속성에 따라 완전히 달라집니다. position 속성이 static이나 relative면, 컨테이닝 블록은 블록 컨테이너(inline-block, block, list-item 등의 요소) 또는 서식 문맥을 형성하는 요소(table, flex, grid, 아니면 블록 컨테이너 자기 자신)의 콘텐츠 영역입니다. position 속성이 absolute인 경우, 컨테이닝 블록은 position 값이 static이 아니고(fixed, absolute, relative, sticky) 가장 가까운 조상의 내부 여백 영역입니다. position 속성이 fixed인 경우, 컨테이닝 블록은 뷰포트나 (페이지로 나뉘는 매체에선) 페이지 영역입니다. position 속성이 absolute나 fixed인 경우, 다음 조건 중 하나를 만족하는 가장 가까운 조상의 내부 여백 영역이 컨테이닝 블록이 될 수도 있습니다. height, top, bottom 속성은 컨테이닝 블록의 height를 사용해 백분율을 계산합니다. 컨테이닝 블록의 height가 콘텐츠의 크기에 따라 달라질 수 있고, 컨테이닝 블록의 position이 relative거나 static이면 계산값은 0이 됩니다. width, left, right, padding, margin 속성은 컨테이닝 블록의 width를 사용해 백분율을 계산합니다. position이 fixed라면 컨테이닝 블록은 초기 컨테이닝 블록(화면 매체에서는 뷰포트)입니다. `width: 50%; /* == (50vw - (세로 스크롤바 너비)) */`

"Resetting"과 "Normalizing" CSS의 차이점

- **Resetting** - Resetting은 요소의 모든 기본 브라우저 스타일을 제거하기 위한 것입니다. 예: `margin`, `padding`, `font-size`는 같은 값으로 재설정됩니다. 일반적인 타이포그래피 요소에 대한 스타일을 재 선언해야 합니다.
- **Normalizing** - Normalizing는 모든 것을 "정리"하는 것이 아니라 유용한 기본 스타일을 보존합니다. 또한 일반적인 브라우저 종속성에 대한 버그를 수정합니다.

필자는 나만의 스타일링을 많이 해야 하고 보존할 기본 스타일링이 필요하지 않도록 매우 맞춤화되었거나 자유로운 사이트 디자인을 가지고 있을 때 리셋을 선택합니다.

`float` 의 작동법

Float은 CSS 위치 지정 속성입니다. Float 된 요소는 페이지의 흐름의 일부로 남아 있으며 페이지의 흐름에서 제거되는 'position : absolute' 요소와 달리 다른 요소 (예: 플로팅 요소 주위로 텍스트가 흐르게 됨)의 위치 지정에 영향을 줍니다.

CSS `clear` 속성은 `left` / `right` / `both` float 엘리먼트 아래에 위치하도록 사용될 수 있습니다.

부모 요소에 float 된 요소만 있으면 그 요소로 반영되는 높이는 무효가 됩니다. 컨테이너의 플로팅된 요소 다음에 있지만 컨테이너가 닫히기 전에 float 를 clear 하면 해결할 수 있습니다.

이 때 가상선택자를 사용해서 DOM태그를 줄이면 좋습니다.

css

```
.parent::after {
    content: ' ';
    display: block;
    clear: both;
}
```

양자택일로, 부모 요소에 `overflow : auto` 또는 `overflow : hidden` 속성을 주어서 자식 요소 내부에 새로운 블록 포맷 컨텍스트를 설정하고 자식을 포함하도록 확장해도 됩니다.

z-index 의 작동법

CSS 의 `z-index` 속성은 요소의 겹치는 요소의 순서를 제어합니다. `z-index` 는 `static` 이 아닌 `position` 및 `relative` 값을 갖는 요소에만 영향을 줍니다.

`z-index` 값이 없으면 DOM에 나타나는 순서대로 요소가 쌓이게 됩니다 (동일한 레이어에서 가장 낮은 레이어의 맨 위에 나타납니다). 정적이지 않은(non-static) 위치 지정 요소 (및 해당 하위 요소)는 HTML 레이어 구조와 상관없이 기본 정적 위치 지정을 사용하여 항상 요소 위에 나타납니다.

쌓임 맥락(stacking context)은 레이어 집합을 포함하는 요소입니다. 쌓임 맥락(stacking context) 지역 내에서 자식의 `z-index` 값은 문서 루트가 아닌 해당 요소를 기준으로 설정됩니다. 해당 컨텍스트 외부의 레이어 – 즉 로컬 쌓임 맥락의 형제 요소 – 그 사이의 레이어에 어울릴 수 없습니다. 요소 B 가 요소 A 의 상단에 위치하는 경우, 요소 A 의 하위 요소 C 는 요소 C 가 요소 B 보다 `z-index` 가 더 높은 경우에도 요소 B 보다 높을 수 없습니다.

각각의 쌓임 맥락은 자체적으로 포함되어 있습니다 - 요소의 내용이 쌓인 후에는 전체 요소를 쌓임 맥락의 쌓인 순서로 고려합니다. 소수의 CSS 속성이 `opacity` 가 1 보다 작고 `filter` 가 `none` 이 아니며 `transform` 이 `none` 이 아닌 새롭게 쌓임 맥락(stacking context)을 트리거합니다. <https://tiffanybbrown.com/2015/09/css-stacking-contexts-wtf/index.html>

inline 과 **inline-block** 그리고 **block**

	block	inline-block	inline
크기	부모 컨테이너의 너비 를 채웁니다.	내용에 따라 달라집니다.	내용에 따라 달라집니다.

	block	inline-block	inline
위치	새 줄에서 시작하고 그 옆에 HTML 요소를 허용하지 않습니다 (<code>float</code> 을 추가할 때를 제외하고).	다른 콘텐츠와 함께 흐르고 다른 요소는 옆에 있는 것을 허용합니다.	F 다른 콘텐츠와 함께 흐르고 다른 요소는 옆에 있는 것을 허용합니다.
<code>width</code> , <code>height</code> 지정 가능 여부	가능	가능	불가능. 설정되면 무시됩니다.
<code>vertical-align</code> 정렬 가능 여부	불가능	가능	불가능
margin 및 padding	모든 방향에서 가능.	모든 방향에서 가능.	수평방향만 가능. 세로방향을 지정하면 레이아웃에 영향을 주지 않습니다. <code>border</code> 와 <code>padding</code> 이 콘텐츠 주위에 시각적으로 나타나는 경우에도 수직영역은 <code>line-height</code> 에 의존합니다.
Float	-	-	수직 margin 과 padding 을 설정할 수 있는 block 엘리먼트와 같습니다.
### <code>relative</code> , <code>fixed</code> , <code>absolute</code> 와 <code>static</code> 요소의 차이점			

위치가 정해진 요소는 계산된 `position` 속성이 `relative` , `absolute` , `fixed` 또는 `sticky` 인 요소입니다.

- `static` - 기본 위치. 요소는 평소와 같이 페이지에 위치합니다. `top` , `right` , `bottom` , `left` 및 `z-index` 속성은 적용되지 않습니다.
- `relative` - 요소의 위치는 레이아웃을 변경하지 않고 자체에 상대적으로 조정됩니다. (따라서 배치되지 않은 요소의 간격을 남겨 둡니다.)
- `absolute` - 요소는 페이지의 평소 위치에서 제거되고 가장 가까운 위치에 `relative` 부모 블록이 있는 경우 지정된 위치에 배치됩니다. 그렇지 않으면 최상위 블록과 관련됩니다. `absolute` 로 배치된 박스는 여백을 가질 수 있으며 다른 여백과 충돌하지 않습니다. 이 요소는 다른 요소의 위치에 영향을 주지 않습니다.
- `fixed` - 요소는 페이지의 평소 위치에서 제거되고 뷰포트를 기준으로 지정된 위치에 배치되며 스크롤 할 때 이동하지 않습니다.

- **sticky** - 스티키 포지셔닝은 **relative** 와 **fixed** 의 하이브리드입니다. 요소는 지정된 임계 값을 넘을 때까지 **상대적** 위치로 처리되며, 특정 지점에서 **고정된** 위치로 처리됩니다.

media 속성

예, @ media 속성을 screen 포함하여 4 가지 종류가 있습니다 :

- all - 모든 미디어 기기 장치
- print - 프린터
- speech - 화면을 크게 읽는 스크린리더
- screen - 컴퓨터 스크린, 태블릿, 스마트 폰 등 **print** 미디어의 사용 예제 :

css

```
@media print {
  body {
    color: black;
  }
}
```

콘텐츠 숨기는 방법

- **display :none**
- **visibility: hidden** . 그러나 요소는 아직 페이지의 흐름에 여전히 공간을 차지하고 있습니다.
- **width: 0; height: 0** . 요소가 화면의 어떤 공간도 차지하지 않도록하십시오. 결과적으로 보이지 않습니다.
- **position: absolute; left: -99999px** . 화면 외부에 배치합니다.
- **text-indent: -9999px** . 이것은 **block** 인 엘리먼트 내의 텍스트에서만 작동합니다.

웹접근성

- 메타 데이터. 예를 들어, Schema.org, RDF 및 JSON-LD 를 사용합니다.
- WAI-ARIA. 웹 페이지의 접근 가능성을 높이는 방법을 지정하는 W3C 기술 사양입니다.

WAI-ARIA

role 속성을 사용하여 객체(article, alert, slider와 같은 것들)의 일반(general) 타입을 정의할 수 있습니다. 이 외에도 ARIA 속성을 추가로 사용하여 서식에 관한 설명이나 상태바(progressbar)의 현재 값을 제공하는 등 유용한 프로퍼티들을 제공할 수 있습니다.

attribute와 property의 차이점

attribute 속성은 HTML 마크업에 정의되지만 property 속성은 DOM에 정의됩니다. DOM이 변경될 때 같이 변화되는 것을 Property라고 합니다. 차이점을 설명하기 위해 HTML에 이 텍스트 필드가 있다고 세요. `<input type="text" value="Hello">`.

js

```
const input = document.querySelector('input');
console.log(input.getAttribute('value')); // Hello
console.log(input.value); // Hello
```

그러나 텍스트 필드에 "World!"를 추가하면 이렇게 될것입니다.

js

```
console.log(input.getAttribute('value')); // Hello
console.log(input.value); // Hello World!
```

CSS 전처리기| SASS

장점:

- CSS 의 유지보수성 향상됩니다.
- 중첩된 선택자를 작성하기 쉽습니다.
- 일관된 스타일링 설정을 위한 변수사용.

단점:

- 전처리기를 위한 도구가 필요합니다. 다시 컴파일하는 시간이 느릴 수 있습니다.

document load 이벤트와 document DOMContentLoaded 이벤트의 차이점

- DOMContentLoaded: DOM트리 구축 그러나 `img` 및 스타일시트는 로드 미완료
- load : 모든 리소스다운완료

`==` 와 `===` 의 차이점

`==` 는 추상 동등 연산자이고 `===` 는 완전 동등 연산자입니다. `==` 연산자는 타입 변환이 필요한 경우 타입 변환을 한 후에 동등한지 비교할 것입니다. `===` 연산자는 타입 변환을 하지 않으므로 두 값이 같은 타입이 아닌 경우 `==` 는 단순히 `false` 를 반환합니다. `==` 를 사용하면 다음과 같은 무서운 일이 발생할 수 있습니다.

```
1 == '1'; // true
1 == [1]; // true
1 == true; // true
0 == ''; // true
0 == '0'; // true
0 == false; // true
```

저의 조언은 편의상 `null` 과 `undefined` 를 비교할 때를 제외하고, `==` 연산자를 절대 사용하지 않는 것입니다. `a == null` 은 `a` 가 `null` 또는 `undefined` 이면 `true` 를 반환합니다.

```
var a = null;
console.log(a == null); // true
console.log(a == undefined); // true
```

DOCTYPE

DOCTYPE은 **document type**의 약어입니다. **DOCTYPE**은 항상 **DTD(Document Type Definition)**와 관련되며 **DTD**는 특정 문서가 어떻게 구성되어야 하는지 정의합니다(예시: `button` 은 `span` 을 포함할 수 있지만, `div` 는 그럴 수 없다.)

HTML은 한 종류가 아니라, 여러 종류의 HTML이 있습니다. HTML 4.01 Strict, HTML 4.01 Transitional, XHTML 1.0 등입니다. 이러한 유형에 따라서 같은 코딩을 하더라도 html 파일 실행시에 다른 결과 화면으로 나타나며, Doctype 선언했을 경우와 선언하지 않았을 경우, 브라우저에 따라서 다르게 출력이 됩니다.

만약 DOCTYPE 선언하지 않는다면 브라우저는 현재 페이지가 어떠한 HTML 버전을 사용하고 있는지 인식할 수 없기 때문에 호환모드(quirks mode)로 변환을 해서 rendering(화면구현)하게 됩니다.

Quirks Mode

Quirks mode는 오래된 웹 브라우저들을 위해 디자인된 웹 페이지의 하위 호환성을 유지하기 위해 W3C나 IETF의 표준을 엄격히 준수하는 Standards Mode를 대신하여 사용되는 웹 브라우저의 기술을 나타낸다. 같은 코드라도 웹 브라우저마다 서로 해석을 달리 하기 때문에, 전혀 다른 결과물을 보여주게 된다.

HTML5 표준에 대한 DOCTYPE 선언은 `<!DOCTYPE html>` 입니다.

http2

http1.1의 단점 : HTTP/1.1는 기본적으로 Connection당 하나의 요청을 처리 하도록 설계 되어있다. 그래서 위 그림과 같이 동시전송이 불가능하고 요청과 응답이 순차적으로 이루어 지게된다. 그렇다 보니 HTTP문서안에 포함된 다수의 리소스 (Images, CSS, Script)를 처리하려면 요청할 리소스 개수에 비례해서 Latency(대기 시간)는 길어지게 된다.

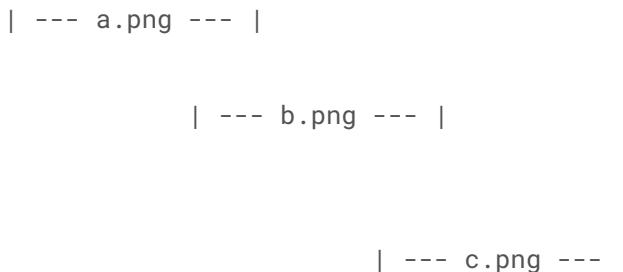
HOL (Head Of Line) Blocking - 특정 응답의 지연

Web환경에서 HOLB는 실제로 두 종류가 존재한다.

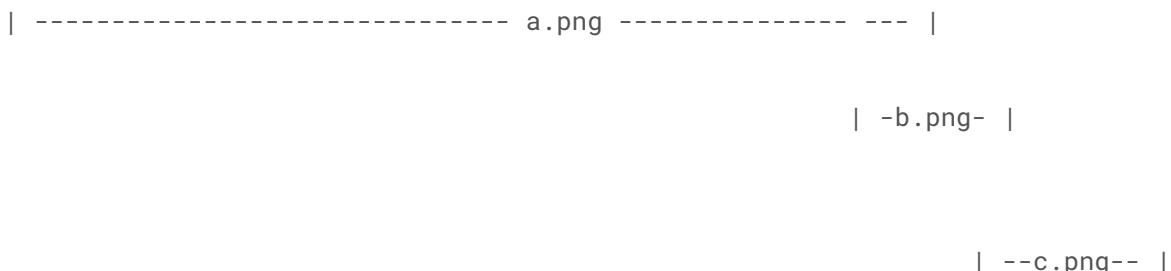
- HTTP의 HOL Blocking
- TCP의 HOL Blocking

우리의 관심은 HTTP의 HOLB이고 이에 대해 알아보자 HTTP/1.1의 connection당 하나의 요청처리를 개선할 수 있는 기법중 pipelining이 존재하는데 이것은 하나의 Connection을 통해서 다수개의 파일을 요청/응답 받을 수 있는 기법을 말하는데 이 기법을 통해서 어느정도의 성능 향상을 꾀 할 수 있으나 큰 문제점이 하나 있다.

하나의 TCP연결에서 3개의 이미지(a.png, b.png, c.png)를 얻을려고 하는경우 HTTP의 요청순서는 다음 그림과 같다.



순서대로 첫번째 이미지를 요청하고 응답받고 다음 이미지를 요청하게 되는데 만약 첫번째 이미지를 요청하고 응답이 지연되면 아래 그림과 같이 두,세번째 이미지는 당연히 첫번째 이미지의 응답처리가 완료되기 전까지 대기하게 되며 이와 같은 현상을 HTTP의 Head of Line Blocking 이라 부르며 파이프 라이닝의 큰 문제점 중 하나이다.



- RTT(Round Trip Time) 증가 앞서 말한것처럼 http/1.1의 경우 일반적으로 하나의 connection에 하나의 요청을 처리 한다. 이렇다 보니 매 요청별로 connection을 만들게 되고 TCP상에서 동작하는 HTTP의 특성상 3-way Handshake 가 반복적으로 일어나고 또한 불필요한 RTT증가와 네트워크 지연을 초래하여 성능을 저하 시키게 된다.

- 무거운 Header 구조 (특히 Cookie) http/1.1의 헤더에는 많은 메타정보들이 저장되어져 있다. 사용자가 방문한 웹 페이지는 다수의 http요청이 발생하게 되는데 이 경우 매 요청시마다 중복된 헤더값을 전송하게 되며(별도의 domain sharding을 하지 않았을 경우) 또한 해당 domain에 설정된 cookie정보도 매 요청시마다 헤더에 포함되어 전송되며 어쩔땐 요청을 통해서 전송하려는 값보다 헤더 값이 더 큰경우도 비일비재 하다.

이를 극복하기 위한 방법

- Image Spriting
- Domain Sharding(동시에 한 도메인으로부터 6개의 데이터를 동시에 받는 성질을 이용해)
- Minify CSS/Javascript
- Data URI Scheme: HTML문서내 이미지 리소스를 Base64로 인코딩된 이미지 데이터로 직접 기술하는 방식이고 이를 통해 요청 수를 줄이기도 한다.
- Load Faster : 스타일시트를 HTML 문서 상위에 배치 / 스크립트를 HTML문서 하단에 배치

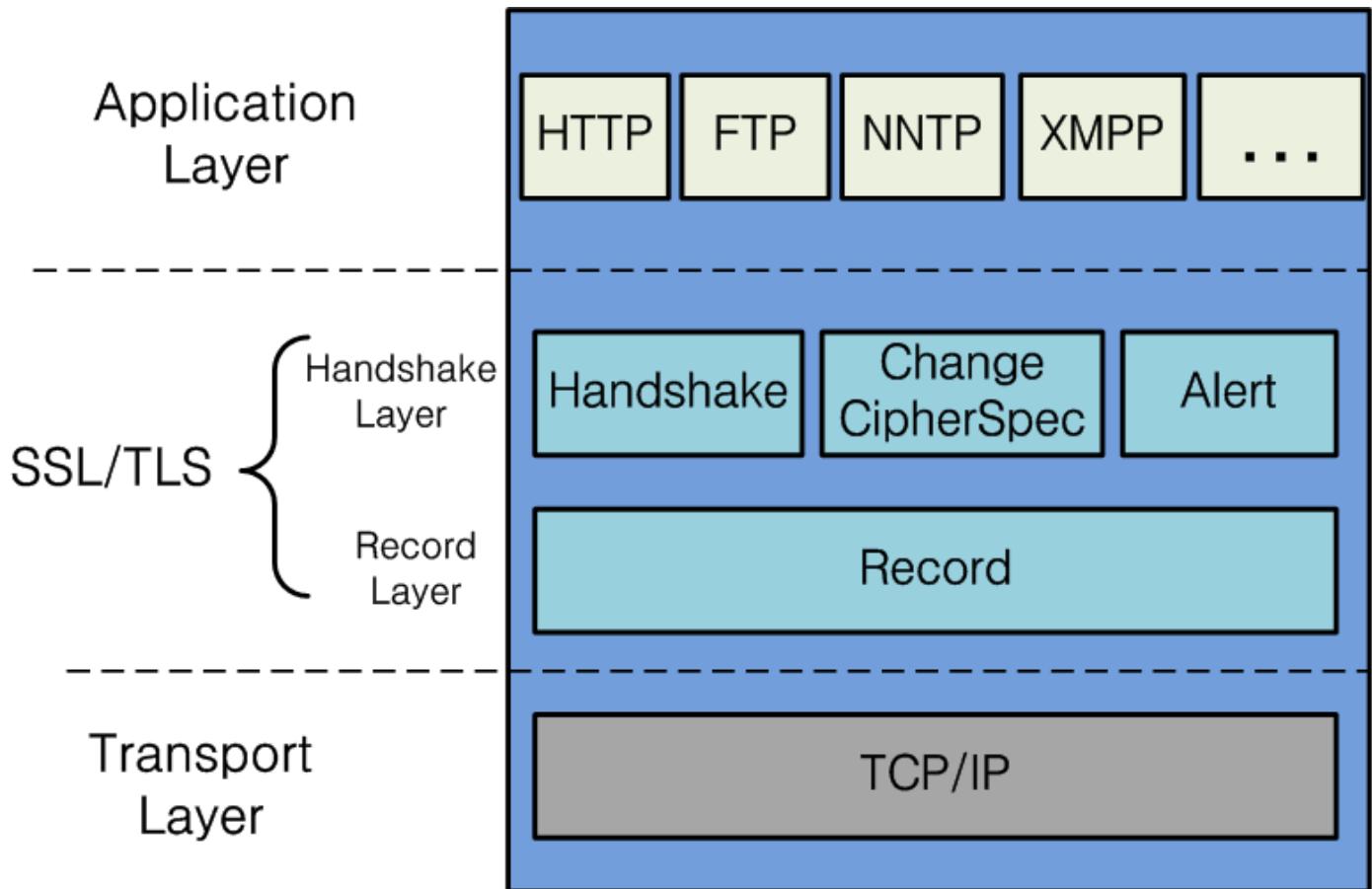
그러다.. SPDY가 나오고 HTTP/2가 나오게 되었다.

HTTP/2

HTTP/2에서는 새 바이너리 프레이밍 계층을 도입합니다. 이 계층은 이전의 HTTP/1.x 서버 및 클라이언트와 호환되지 않으며 따라서 주 프로토콜 버전은 HTTP/2로 올라갑니다.

TLS

TLS는 SSL, Secure Sockets Layer의 새로운 이름, 이 프로토콜이 그저 인터넷 소켓이 아닌 양방향의 바이트 스트림에서 작동하는 것을 위함입니다. HTTPS는 SSL을 의미하는 S를 가진 HTTP이다. TLS(Transport Layer Security)는 인터넷 상에서 통신할 때 주고받는 데이터를 보호하기 위한 표준화된 암호화 프로토콜입니다. TLS는 넷스케이프사에 의해 개발된 SSL(Secure Socket Layer) 3.0 버전을 기반으로 하며, 현재는 TLS버전 1.3이 최종 버전입니다. TLS는 전송계층(Transport Layer)의 암호화 방식이기 때문에 HTTP뿐만 아니라 FTP, XMPP등 응용 계층(Application Layer)프로토콜의 종류에 상관없이 사용할 수 있다는 장점이 있으며 기본적 으로 인증(Authentication), 암호화(Encryption), 무결성(Integrity)을 지원합니다.



SSL에서 TLS로 이름이 변경된 지 오래됐지만 아직도 사람들은 TLS대신 SSL이라는 표현을 더 많이 사용하고 있으며 실제로 SSL/TLS의 오픈소스 구현체 프로젝트의 명칭은 아직도 OpenSSL이기도 합니다. 또한 SSL/TLS의 가장 주된 적용 대상이 HTTPS다 보니 SSL/TLS를 HTTPS와 혼용하는 경우도 많습니다. 이 문서에서는 SSL이라고 할 경우 SSL 프로토콜, TLS는 TLS 프로토콜을 의미하며, 보안이 적용된 HTTP는 HTTPS로 지칭하겠습니다.

SSL/TLS를 사용하면 중간자 공격과 Packet Spoofing을 통한 도감청을 막을 수 있으며 통신하는 상대방이 맞는지 인증할 수 있습니다.

TLS의 작동 방식

인터넷을 통해 안전하게 통신하려면 암호화가 필요하다. 만일 데이터가 암호화되지 않으면 누구나 패킷을 들여다 보고 기밀 정보를 읽을 수 있다. 가장 안전한 암호화 방법은 ‘비대칭 암호화’다. 제대로 작동하려면 2개의 암호 키(대개 엄청나게 큰 숫자로 이루어졌다)가 필요하다. 하나는 공용 키이고 다른 하나는 개인 키다.

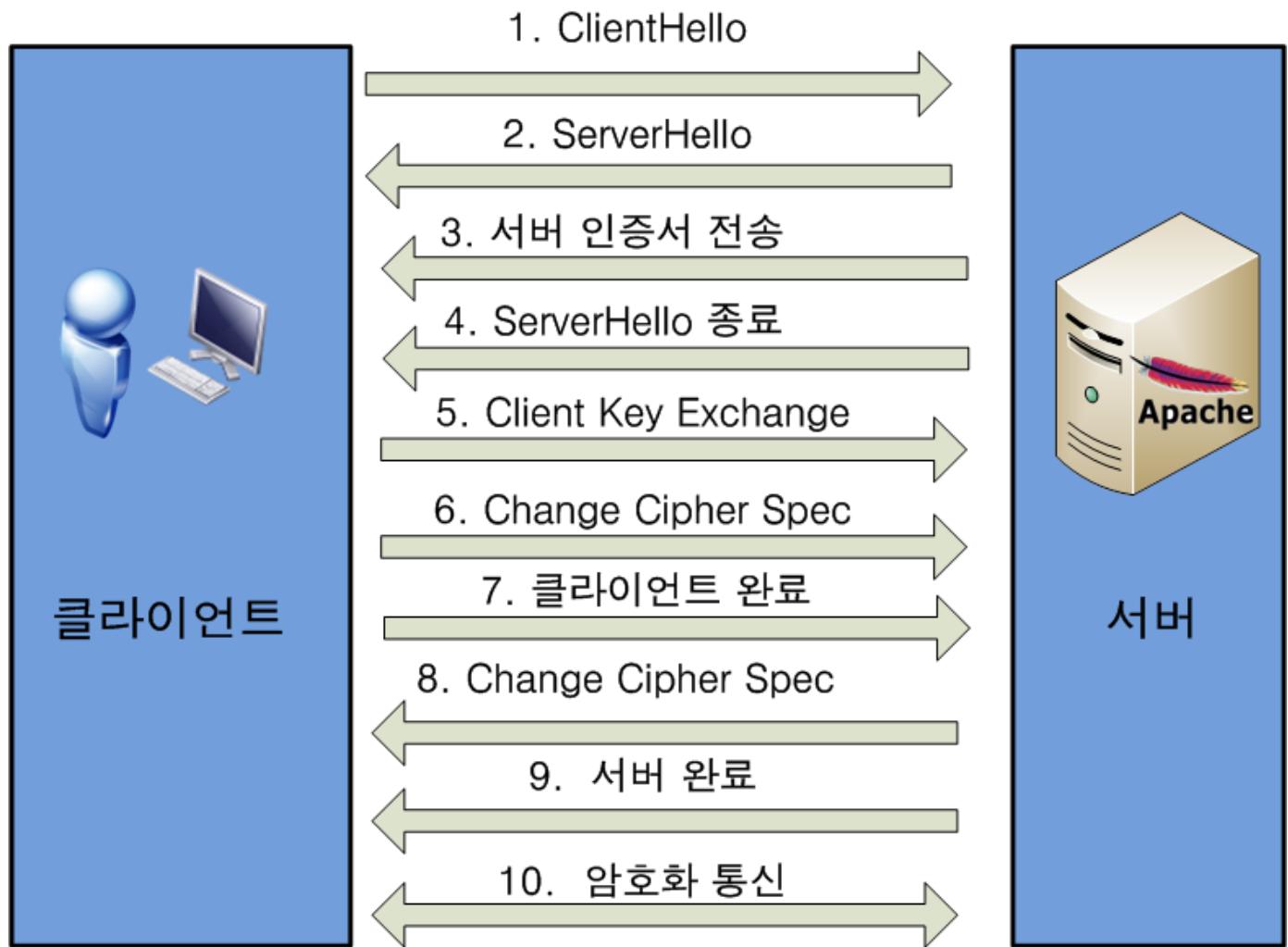
여기서 관련된 수학 개념은 복잡한데 간단히 말하면 공용 키는 데이터 ‘암호화’에 사용되고 ‘복호화’하에는 개인 키가 필요하다. 2개의 키는 무작위 시도로는 역엔지니어링하기 어려운 복잡한 수학 공식에 의해 서로 연결돼 있다. 비유하자면 공용 키는 전면에 넣을 수 있는 구멍이 있는 잠겨진 우편함에 대한 정보이고 개인 키는 그 우편함을 열 수 있는 키라고 생각하면 된다. 우편함의 장소를 아는 사람이라면 누구나 메시지를 안에 넣을 수 있지만 누군가 그 메시지를 읽으려면 개인 키가 필요하다.

비대칭 암호화에는 이와 같은 어려운 문제가 수반되기 때문에 컴퓨팅 자원이 많이 소요된다. 통신 세션에서 모든 정보를 암호화하면 감당이 안돼 컴퓨터와 연결이 서서히 중단될 정도이다. TLS는 이 문제를 해결하기 위해 통신 세션이 맨 처음 시작할 때만 비대칭 암호화를 사용한다. 그 이후부터는 패킷 암호화에 서버와 클라이언트가 사용할 하나의 ‘세션 키’에

합의하기 위해 양쪽이 나누는 대화를 암호화하는 것이다. 공유된 키를 사용하는 암호화를 ‘대칭 암호화’라고 한다. 비대칭 암호화에 비해 컴퓨터 자원 소모가 덜하다. 해당 세션 키는 비대칭 암호 작성 방식을 이용해 설정되었기 때문에 그렇지 않은 경우에 비해 통신 세션 전체가 훨씬 더 안전하다.

TLS HandShake

SSL/TLS 세션은 다음 핸드셰이크 과정을 거친 후에 구축됩니다.



- 클라이언트와 서버는 헬로 메시지로 기본적인 정보를 송수신 (1, 2)
- 서버는 서버가 사용하는 SSL/TLS 인증서를 전달 (3, 4)
- 클라이언트는 암호화 통신에 사용할 대칭키를 생성하고 사이를 서버에 전달(5). 이 과정을 키 교환(Key Exchange)라고 하며 디피-헬만 키 교환(Diffie–Hellman key exchange) 또는 RSA 를 많이 사용.
- 클라이언트는 암호화 통신에 사용 가능한 암호 알고리즘과 해시 알고리즘 목록을 서버에 전달. (6, 7)
- 서버도 알고리즘 목록을 교환후 핸드셰이크가 종료되며 이제 클라이언트와 서버는 암호화 통신에 필요한 대칭키를 서로 보유.(8, 9)

위 과정이 끝나면 SSL 세션이 구축되며 실제 암호화 통신을 시작할 수 있습니다.

Multiplexed Streams

한출처당 하나의 연결, 즉, 한 커넥션으로 동시에 여러개의 메세지를 주고 받을 있으며, 응답은 순서에 상관없이 stream으로 주고 받는다. HTTP/1.1의 Connection Keep-Alive, Pipelining의 개선이라 보면 된다. 연결 수가 적으면 값비싼 TLS 핸드셰이크가 줄어들고, 세션 재사용이 더 향상되며, 필요한 클라이언트 및 서버 리소스가 감소합니다.

Stream Prioritization

예를 들면 클라이언트가 요청한 HTML문서안에 CSS파일 1개와 Image파일 2개가 존재하고 이를 클라이언트가 각각 요청하고 난 후 Image파일보다 CSS파일의 수신이 늦어지는 경우 브라우저의 렌더링이 늦어지는 문제가 발생하는데 HTTP/2의 경우 리소스간 의존관계(우선순위)를 설정하여 이런 문제를 해결하고 있다.

Server Push

서버는 클라이언트의 요청에 대해 요청하지도 않은 리소스를 마음대로 보내줄 수도 있다.

모든 서버 푸시 스트림은 PUSH_PROMISE 프레임을 통해 시작되며, 이 프레임은 설명된 리소스를 클라이언트에 푸시하라는 신호를 서버 인텐트에 보냅니다. 이 프레임은 푸시된 리소스를 요청하는 응답 데이터보다 먼저 전달되어야 합니다. 이러한 전달 순서는 매우 중요합니다. 리소스에 대해 중복 요청이 생성되는 것을 막기 위해 클라이언트는 서버가 어떤 리소스를 푸시할지를 알아야 합니다. 이러한 요구사항을 충족시키는 가장 단순한 전략은 약속했던 리소스의 HTTP 헤더만 포함된 모든 PUSH_PROMISE 프레임을 상위 요소의 응답(즉, DATA 프레임)보다 먼저 전송하는 것입니다.

클라이언트가 PUSH_PROMISE 프레임을 수신한 후에 (RST_STREAM 프레임을 통해) 해당 스트림을 거부할 수 있는 옵션이 있습니다. (예를 들어, 리소스가 이미 캐시에 있기 때문에 이러한 상황이 발생할 수 있습니다) 이것은 HTTP/1.x에 비해 개선된 중요한 기능입니다. 반대로 리소스 인라인 처리 사용은 HTTP/1.x에서 인기 있는 '최적화' 방법으로, '강제 푸시'와 동일합니다. 클라이언트는 인라인 처리된 리소스를 개별적으로 옵트아웃하거나 취소하거나 처리할 수 없습니다.

옵트아웃(Opt-out)은 당사자가 자신의 데이터 수집을 허용하지 않는다고 명시할 때 정보수집이 금지되는 제도이다.

HTTP/2에서는 클라이언트가 서버 푸시의 사용 방식을 완벽하게 제어합니다. 클라이언트는 동시에 푸시되는 스트림의 수를 제한할 수 있고, 스트림이 최초로 열릴 때 푸시되는 데이터의 크기를 제어하는 초기 흐름 제어 창을 조정할 수 있으며, 서버 푸시를 완전히 비활성화할 수도 있습니다. 이러한 기본은 HTTP/2 연결 시작 시에 SETTINGS 프레임을 통해 전달되며 언제든지 업데이트될 수 있습니다.

Header Compression

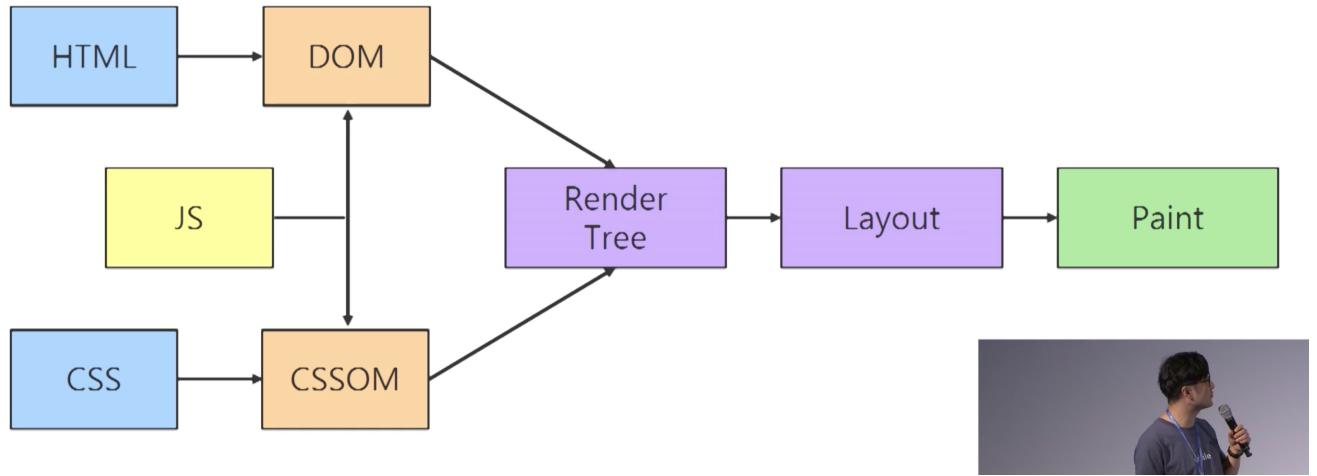
HTTP/2는 Header 정보를 압축하기 위해 Header Table과 Huffman Encoding 기법을 사용하여 처리하는데 이를 HPACK 압축방식이라 부르며 별도의 명세서(RFC 7531)로 관리하고 있다. HTTP/1.x의 경우 두개의 요청 Header에 중복 값이 존재해도 그냥 중복 전송한다. 하지만 HTTP/2에선 Header에 중복값이 존재하는 경우 Static/Dynamic Header Table 개념을 사용하여 중복 Header를 검출하고 중복된 Header는 index값만 전송하고 중복되지 않은 Header정보의 값은 Huffman Encoding 기법으로 인코딩 처리 하여 전송한다.

브라우저의 렌더링 과정

- 과정
- vSync
- 리플로우 / 리페인트 최소화

과정

1.1 Summary of browsers main flows

DEVIEW
2018

HTML >> DOM토큰(이 과정을 HTML파싱) >> DOM트리 >> CSS규칙(파싱된 CSS결과인 CSSOM)에 따라 Render트리생성(display:none제거 / font-size 등 상속 스타일 부모에만 위치하게설계) >> Layout설정(좌표 설정, 보통 부모를 기준으로 설정됨 / Global Layout이 변경될 때는 브라우저의 사이즈가 증가하거나 폰트사이즈를 증가시키면 변경된다.) >> paint(한 픽셀 한 픽셀 인쇄하는 듯 칠해지게 된다.)

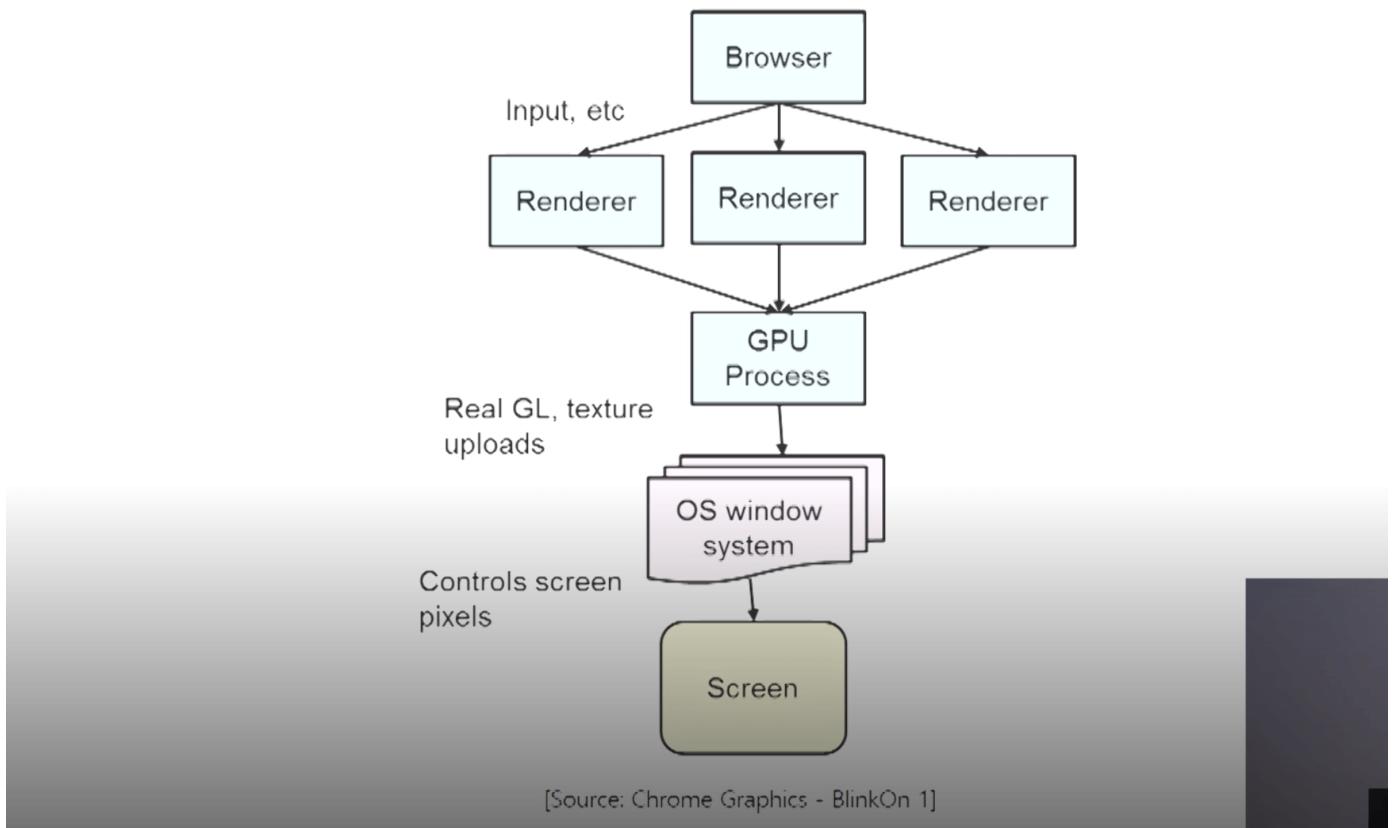
렌더트리는 DOMTree와 1 : 1 관계가 아니다. 화면에 보여줄 부분만 렌더링을 하기 위해 트리로 만드는 것.

최신 브라우저는 JS > Recalc Style > Layout > Update Layer Tree > Paint > Composite 의 과정으로 화면 생성됩니다.

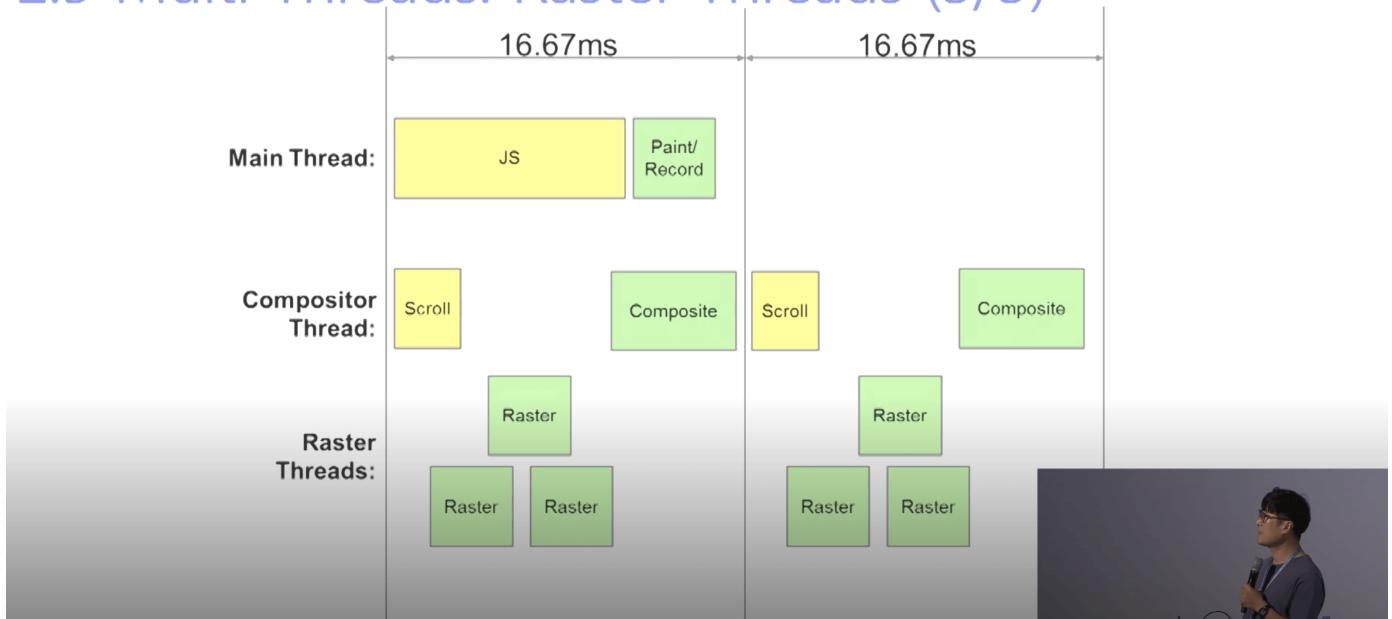
이 과정을 vSync 안에서 즉, 16.6ms안에 끝내야 합니다.

예전 브라우저는 이 모든 과정을 메인쓰레드 즉, 싱글코어밖에 사용하지 못하는 구조이였지만 최신브라우저는 이를 분할했습니다.

Multi-process architecture



2.5 Multi Threads: Raster Threads (3/3)

DEVVIEW
2018

- Compositor Thread : scrolling, animation, zoomIn/ out : 단독으로 호출 가능 - Raster Thread : draw line을 직접 수행한다.
- Paint : 레이어 별로 색을 칠한다.
- Composite : 레이어를 합쳐 bitmap으로 만듬

가장 끝단을 수정하는 것이 제일 좋으며 각 단계에서는 이 점을 고려하면 최적화가 됩니다.

- Layout : width, height, font(1000개 이하의 DOM이 효율적)

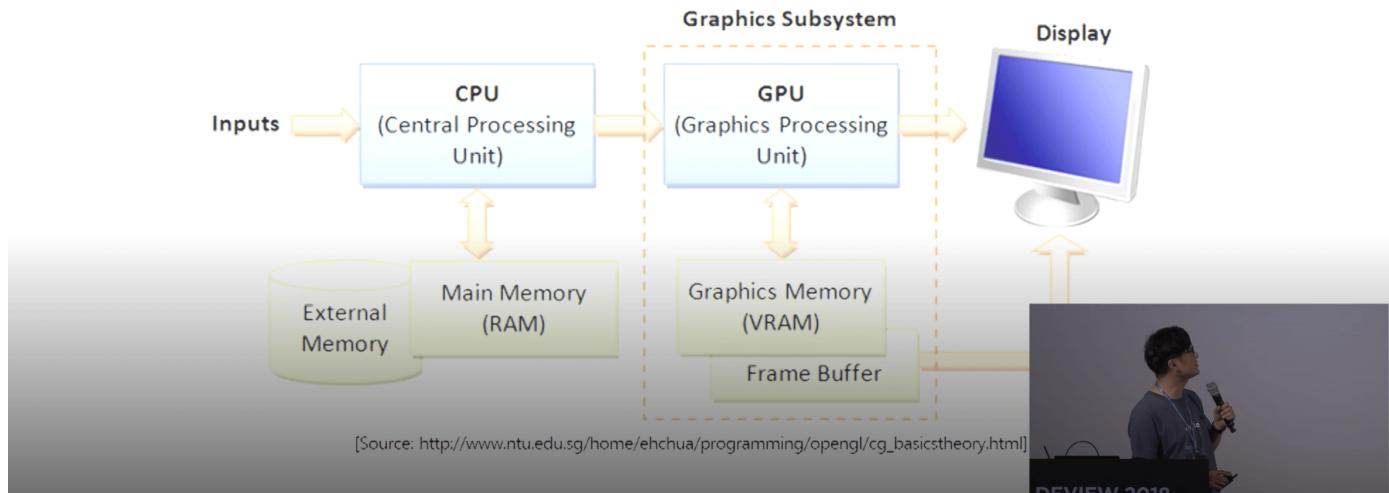
- Paint : color, background (GPU Rsterization를 이용하면 더 빠름. `view-port content="width=device-width"` 를 사용하면 됨.)
- Composite : opacity, transform()(레이어는 30개 이하의 레이어가 효율적입니다.)

vSync

DEVIEW
2018

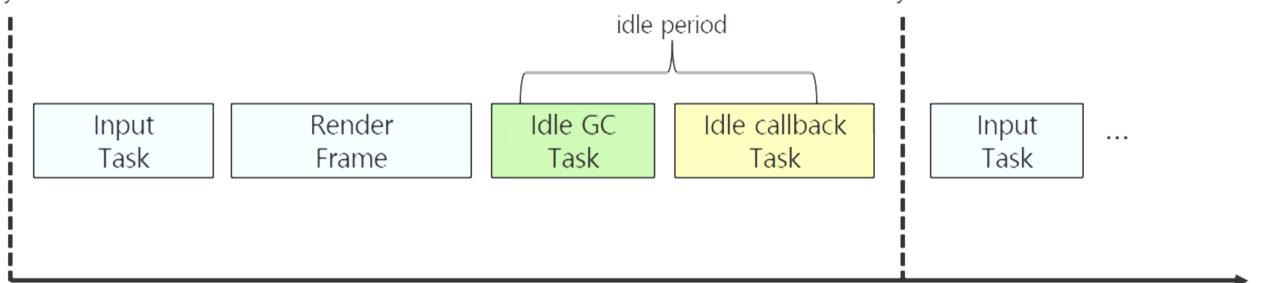
2.1 VSync Overview (1/2)

“60hz의 의미는 모니터가 16.6ms 단위로 Frame Buffer의 내용을 Fetch!”

DEVIEW
2018

3.4 VSync based browser processing

VSync



VSync



자연스럽게 화면을 바꿀 수 있는 타이밍을 뜻합니다.

리플로우 / 리페인트 최소화

노드를 추가하거나 스타일을 변경하면 리플로우와 리페인트가 일어나게 된다. 이를 통해 부라우저 렌더링이 통째로 일어나게 되고 많은 cost를 가져오게 됩니다. 이를 최소화하는 것이 중요합니다.

```
var bstyle = document.body.style; // cache

bstyle.padding = "20px"; // reflow, repaint
bstyle.border = "10px solid red"; // another reflow and a repaint

bstyle.color = "blue"; // repaint only, no dimensions changed
bstyle.backgroundColor = "#fad"; // repaint

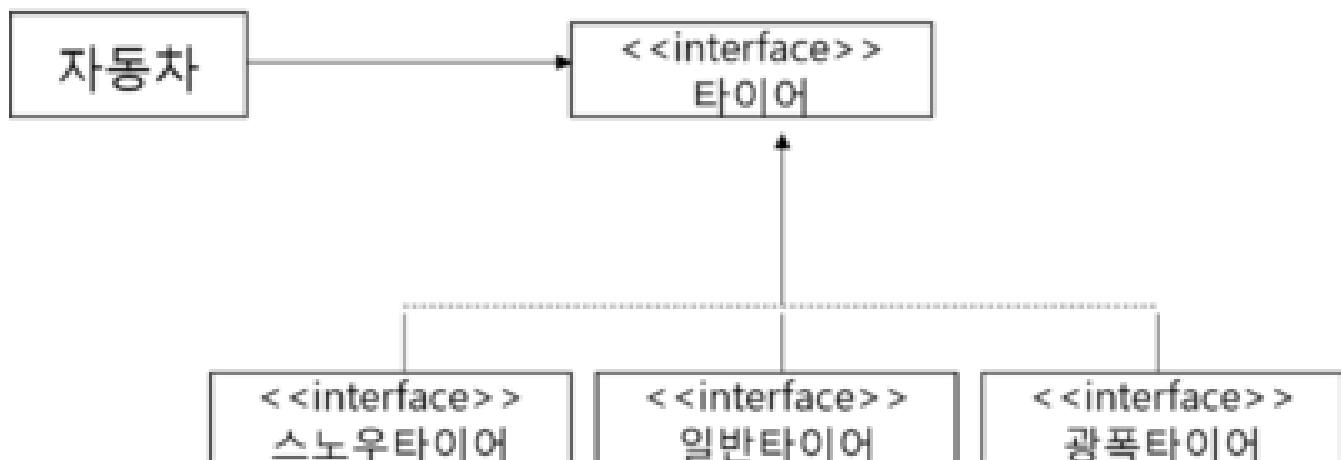
bstyle.fontSize = "2em"; // reflow, repaint

// new DOM element - reflow, repaint
document.body.appendChild(document.createTextNode('dude!'));
```

아래의 요소에 요청을 하기만 하는 스크립트를 작성해도 변경대기열큐에 쌓이게 됩니다. 그리고 변경하는 스크립트가 있으면 flush가 발생하면서 대기열큐에 있던 것이 사라지면서 리플로우/리페인트가 일어나게 됩니다.

```
offsetTop, offsetLeft, offsetWidth, offsetHeight
scrollTop
clientTop
getComputedStyle(), or currentStyle in IE
```

DIP(Dependency Inversion Principle) 의존 역전 원칙



그림과 같이 자동차가 구체적인 타이어들(스노우타이어, 일반타이어, 광폭타이어)이 아닌 추상화된 타이어 인터페이스에만 의존하게 함으로써 스노우타이어에서 일반타이어로, 또는 다른 구체적인 타이어로 변경돼도 자동차는 이제 그 영향을 받지 않는 형태로 구성된다. 자동차는 자신보다 변하기 쉬운 스노우타이어에 의존하던 관계를 중간에 추상화된 타이어 인터페이스를 추가해 두고 의존 관계를 역전시키고 있다. 이처럼 자신보다 변하기 쉬운 것에 의존하던 것을 추상화된 인터페이스나 상위 클래스를 두어 변하기 쉬운 것의 변화에 영향받지 않게 하는 것이 의존 역전 원칙이다. 상위 클래스일 수록, 인터페이스일수록, 추상 클래스일수록 변하지 않을 가능성이 높기에 하위 클래스나 구체 클래스가 아닌 상위클래스, 인터페이스, 추상 클래스를 통해 의존하라는 것이 바로 의존 역전 원칙이다.

상위 계층(정책 결정)이 하위 계층(세부 사항)에 의존하는 전통적인 의존 관계를 반전(역전)시킴으로써 상위 계층이 하위 계층의 구현으로부터 독립되게 할 수 있다. 이 원칙은 다음과 같은 내용을 담고 있다.

- 상위 모듈은 하위 모듈에 의존해서는 안된다. 상위 모듈과 하위 모듈 모두 추상화에 의존해야 한다.
- 추상화는 세부 사항에 의존해서는 안된다. 세부사항이 추상화에 의존해야 한다.

이는 생성자 주입 또는 속성 주입으로 이루어진다. 생성자주입의 경우 예제코드는 다음과 같다.

```
public class Response {

    private final Converter converter;

    public Response(Converter converter) {
        this.converter = converter;
    }
}
```

결국 나쁜 디자인에서 벗어나 좋은 디자인으로 변경하는 하나의 방법을 뜻한다. 나쁜디자인이란?

- 모든 변경마다 많은 다른 부분에 영향을 미쳐 변경 자체가 어렵다. (Rigidity)
- 변경 작업을 할때 예상치 못한 다른 부분이 망가진다. (Fragility)
- 현재의 어플리케이션에서 분리할 수 없기 때문에 다른 어플리케이션에서 재사용 하기가 매우 어렵다. (Immobility)

즉, DIP 패턴으로 작성하게 된다면 의존성 파라미터를 생성자에 작성하지않아도 되기 때문에 코드수가 줄어들고 유지보수성이 증대됩니다.

DI & IOC

둘은 같은 의미로 사용되는데,

객체 자체가 아닌 Framework에 의해 객체의 의존성이 주입되는 설계 패턴이다. 외부에서 넘겨주는 무언가가 IOC 컨테이너에 객체를 (외부)생성 후 이 외부 객체들을 객체에 주입(DI)하는 것이다. 외부에서 넘겨주는 역할을 하는 것을 컨테이너, 컴포넌트, 모듈이라고 부른다. 이는 생성자를 통해 전달 할 수 있고, 멤버 변수를 통해 전달 할 수도 있습니다.

- 오버로딩(Overloading) : 같은 이름의 메소드를 여러 개 가지면서 매개변수의 유형과 개수가 다르도록 하는 기술
- 오버라이딩(Overriding) : 상위 클래스가 가지고 있는 메소드를 하위 클래스가 재정의 해서 상요한다.

GPU 가속화

<https://d2.naver.com/helloworld/2061385>

리페인트 리플로우 감소

<https://wit.nts-corp.com/2017/06/05/4571>

- 또한, 노드의 position 값을 초기에 적용하지 않았더라도 애니메이션 시작 시 값을 변경(fixed, absolute)하고 종료 시 다시 원복 시키는 방법을 사용해도 무관 합니다.

렌더링 트리를 구성하는데 사용된 입력 정보의 어떤 변경도 리플로우와 리페인트를 발생시킨다. 구체적으로는 다음과 같은 경우이다.

- DOM 노드의 추가, 삭제 및 업데이트
- display: none(리플로우와 리페인트) 또는 visibility: hidden(위치 변경은 일어나지 않기 때문에, 리페인트만) 등과 같은 DOM 노드 숨김
- 페이지상에서 DOM 노드의 위치 이동 및 애니메이션
- 스타일 속성의 약간의 변화를 위해 스타일 시트 추가
- 윈도우 크기를 변경하거나 폰트 크기 변경, 그리고 스크롤 등 사용자의 액션 최적화 : cssText 및 클래스
`el.style.cssText += " left: " + left + "px; top: " + top + "px;"`; 애니메이션이 들어간 노드는 가급적 position:fixed 또는 position:absolute로 지정하여 전체 노드에서 분리 시킨다. 클래스 변화에 따른 스타일 변경 시, 최대한 DOM 구조 상 끝단에 위치한 노드에..!일부 노드로 제한 시킬 수 있다.

SEO

<http://blog.airbridge.io/google-seo-guide-part-two/>

OOP 개념 / ES5 ES6 Class 차이

<https://poiemaweb.com/js-object-oriented-programming> <https://goodgid.github.io/What-is-OOP/>
https://developer.mozilla.org/ko/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

먼저 각각의 예를 살펴 보겠습니다.

```
// ES5 함수 생성자
function Person(name) {
    this.name = name;
}

// ES6 클래스
class Person {
    constructor(name) {
        this.name = name;
    }
}
```

간단한 생성자의 경우에는 매우 유사합니다.

생성자의 주요 차이점은 상속을 사용할 때 발생합니다. `Person` 의 하위 클래스이면서 `studentId` 필드를 추가로 가지고 있는 `Student` 클래스를 만들고자 한다면, 이것이 우리가 위에 추가해서 해야 할 일입니다.

```
// ES5 함수 생성자
function Student(name, studentId) {
    // 수퍼 클래스의 생성자를 호출하여 수퍼 클래스에서 상속된 멤버를 초기화합니다.
    Person.call(this, name);

    // 서브 클래스의 멤버를 초기화합니다.
    this.studentId = studentId;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

// ES6 클래스
class Student extends Person {
    constructor(name, studentId) {
        super(name);
        this.studentId = studentId;
    }
}
```

ES5에서 상속을 사용하는 것이 훨씬 더 상세하며 ES6 버전은 이해하고 기억하기가 더 쉽습니다.

AMP

구글 AMP는 기존의 모바일웹보다 빠르게 로딩 되도록 만들어진 페이지

PWA

PWA는 최고의 웹과 최고의 앱을 결합한 경험이다. 브라우저를 통해 처음 방문한 사용자에게 유용하며, 설치가 필요하지 않다. 사용자가 PWA와 관계를 점진적으로 형성할수록 성능이 더욱 강력해 질 것이다. 느린 네트워크에서도 빠르게 로드되고, 관련된 푸시 알림을 전송한다. 모바일 앱처럼 전체 화면으로 로드되고, 홈 화면에 아이콘이 있다.

SVG

SVG는 벡터를 기반으로 점, 선 등을 그린다.

벡터란? 벡터Vector 방식은 비트맵과는 반대로 표현되는 그래픽의 형태shape들이 수학적 공식으로 이루어져 있습니다. 다시말해 벡터 방식의 그래픽은 고정된 비트맵을 가지고 있는 것이 아니라, 수학적 공식으로 이루어진 오브젝트들이 그 때그때 디스플레이에 비트맵화 되어 스크린에 표시됩니다. 여기에서 좀 더 높은 퀄리티의 이미지를 만들기 위해 안티-앨리어싱anti-aliasing이라는 기술이 사용됩니다.

이와 상반되는 것이 비트맵Bitmap입니다. 비트맵은 ‘비트의 지도map of bits’란 듯으로, 각 픽셀에 저장된 일련의 비트 정보 집합입니다. 활용할 때는 **object** 요소는 HTML 문서 내에 직접 내장(inline)시키지 않고 SVG를 조작하는 경우에 가장 좋은 방법입니다.

SVG를 최대한 활용하려면 **object** 을 사용하세요. 또는 HTTP 요청을 저장하기 위해 인라인으로 사용할 수 있지만 이미지는 캐시되지 않는 문제가 있습니다. 이미지처럼 동일하게 SVG를 사용하려면 또는 background-image를 사용하는 것이 좋습니다.

AMD와 CommonJS 차이

두 가지 모두 ES2015 가 등장할 때까지 JavaScript 에 기본적으로 존재하지 않는 모듈 시스템을 구현하는 방법입니다. CommonJS 는 동기식인 반면 AMD (Asynchronous Module Definition - 비동기식 모듈 정의)는 분명히 비동기식입니다. CommonJS 는 서버-사이드 개발을 염두에 두고 설계되었으며 AMD 는 모듈의 비동기 로딩을 지원하므로 브라우저 용으로 더 많이 사용됩니다.

AMD 은 구문이 매우 장황하고 CommonJS 은 다른 언어로 된 import 문을 작성하는 스타일에 더 가깝습니다. 대부분의 경우 AMD 를 필요로 하지 않습니다. 모든 JavaScript 를 연결된 하나의 번들 파일로 제공하면 비동기 로딩 속성의 이점을 누릴 수 없기 때문입니다. 또한 CommonJS 구문은 모듈 작성의 노드 스타일에 가깝고 클라이언트-사이드와 서버-사이드 JavaScript 개발 사이를 전환할 때 문맥 전환 오버 헤드가 적습니다.

ES2015 모듈이 동기식 및 비동기식 로딩을 모두 지원하는 것이 반가운 것은 마침내 하나의 접근 방식만 고수할 수 있다 는 점입니다. 브라우저와 노드에서 완전히 작동되지는 않았지만 언제나 트랜스파일러를 사용하여 코드를 변환할 수 있습

니다.

Vanilla JS

순수 JavaScript, 가볍고 빠른데다가 호환성이 좋다 프로그램세계에서는 부가기능을 제외한 기본적인 기능만 구현한 상태를 부르는 어원이기도 하다

<https://github.com/tastejs/todomvc/tree/master/examples/vanilla-es6>

```
var s = document.getElementById('thing').style;
s.opacity = 1;
(function fade(){(s.opacity-=.1)<0?s.display="none":setTimeout(fade,40)}())();

var r = new XMLHttpRequest();
r.open("POST", "path/to/api", true);
r.onreadystatechange = function () {
  if (r.readyState != 4 || r.status != 200) return;
  alert("Success: " + r.responseText);
};
r.send("banana=yellow");
```

JavaScript 코드디버깅

- [Chrome Devtools](#)
- `debugger` statement
- Good old `console.log` debugging

ES6 스펙

- 기본 매개 변수 (Default Parameters)
- 템플릿 리터럴 (Template Literals)
- 멀티 라인 문자열 (Multi-line Strings)
- 비구조화 할당 (Destructuring Assignment)

ES5에서는 구조화된 데이터를 변수로 받기 위해 아래와 같이 처리해야 했다.

```
// browser
var data = $('body').data(), // data has properties house and mouse
    house = data.house,
    mouse = data.mouse

// Node.js
var jsonMiddleware = require('body-parser').json

var body = req.body, // body has username and password
    username = body.username,
    password = body.password
```

하지만 ES6에서는 비구조화 할당을 사용해 아래와 같이 처리할 수 있다.

```
var {house, mouse} = $('body').data() // we'll get house and mouse variables

var {jsonMiddleware} = require('body-parser')

var {username, password} = req.body
```

주의할 점은 var로 할당하려는 변수명과 구조화된 데이터의 property명이 같아야 한다. 또한 구조화된 데이터가 아니라 배열의 경우 {} 대신 []를 사용해서 위와 유사하게 사용할 수 있다.

```
var [col1, col2] = $('.column'),
    [line1, line2, line3, , line5] = file.split('\n')
```

- 향상된 객체 리터럴 (Enhanced Object Literals)

ES5에서는 아래와 같이 JSON을 사용해서 객체 리터럴을 만들 수 있었다.

```
var serviceBase = {port: 3000, url: 'azat.co'},
    getAccounts = function(){return [1,2,3]}

var accountServiceES5 = {
    port: serviceBase.port,
    url: serviceBase.url,
```

```

getAccounts: getAccounts,
toString: function() {
    return JSON.stringify(this.valueOf())
},
getUrl: function() {return "http://" + this.url + ':' + this.port},
valueOf_1_2_3: getAccounts()
}

```

위 예시와 달리 serviceBase를 확장하길 원한다면 Object.create로 프로토타입화하여 상속 받을 수 있다.

```

var accountServiceES5ObjectCreate = {
    getAccounts: getAccounts,
    toString: function() {
        return JSON.stringify(this.valueOf())
    },
    getUrl: function() {return "http://" + this.url + ':' + this.port},
    valueOf_1_2_3: getAccounts()
}

```

accountServiceES5ObjectCreate.proto = Object.create(serviceBase) accountServiceES5ObjectCreate와 accountServiceES5는 동일하게 사용할 수 있으나 다른 구조를 가진다. accountServiceES5ObjectCreate는 accountServiceES5와 다르게 proto에 port와 url 속성을 가진 객체를 담고 있다.

ES6에서는 아래와 같이 처리할 수 있다.

```

var serviceBase = {port: 3000, url: 'azat.co'},
    getAccounts = function(){return [1,2,3]}
var accountService = {
    __proto__: serviceBase,
    getAccounts,
    toString() {
        return JSON.stringify((super.valueOf()))
    },
    getUrl() {return "http://" + this.url + ':' + this.port},
    [ 'valueOf_' + getAccounts().join('_') ]: getAccounts()
};

```

위 예시에 대해 ES5와의 차이를 요약하면 아래와 같다.

`__proto__` 속성을 사용해서 바로 프로토타입을 설정할 수 있다. `getAccounts: getAccounts`, 대신 `getAccounts` 를 사용할 수 있다 (변수명으로 속성 이름을 지정). [`'valueOf_'` + `getAccounts().join('_')`] 와 같이 동적으로 속성 이름을 정의할 수 있다.

- 화살표 함수 (Arrow Functions)
- Promises
- 블록 범위 생성자 Let 및 Const (Block-Spaced Constructs Let and Const)
- 클래스 (Classes)
- 모듈 (Modules)

await는 ES2017이며 Map + Set + WeakMap + WeakSet, Generators, For..Of 가 있다.

Generators

이터러블/이터레이터 프로토콜

- 이터러블: 이터레이터를 리턴하는 `[Symbol.iterator]()` 를 가진 값
- 이터레이터: `{ value, done }` 객체를 리턴하는 `next()` 를 가진 값
- 이터러블/이터레이터 프로토콜: 이러한 것들을 for...of, 전개 연산자 등과 함께 동작하도록한 규약

이터레이터만들기

```
const obj = {
  [Symbol.iterator]: function() {
    return {
      cur: 0,
      next: function() {
        if (this.cur > 5) return { value: undefined, done: true }
        return {
          value: ++this.cur,
          done: false
        }
      },
      [Symbol.iterator]: function() {
        return this;
      }
    }
  }
};
```

유사 배열

유사 배열은 `{ 0: 10, 1: 20, length: 2 }` 와 같은 객체를 말합니다. 다음과 같은 유사 배열은 전개 연산자나 `for...of` 와 사용될 수 없습니다.

```
const arrayLike = { 0: 10, 1: 20, length: 2 };
for (const val of arrayLike) log(val);
```

물론 유사 배열을 순회할 수 있게 하는 generator를 만든다면 `for...of` 에서 사용할 수 있겠지만, ES6부터는 iterable이 아닌 유사 배열을 사용하지 않는 방향으로 가야합니다. 자바스크립트 3rd party library의 값들 중에는 아직 iterable/iterator 프로토콜을 따르지 않는 유사 배열도 있지만, 모두 ES6의 프로토콜에 맞게 변경될 것입니다.

대표적인 유사 배열로는 arguments, NodeList 등이 있습니다. 최신 환경에서 해당 값들은 iterable이 되었습니다. `[Symbol.iterator]()` 가 구현되어있어, `for...of` 나 전개 연산자와 사용할 수 있습니다.

함수형 프로그래밍

부수효과를 방지하고 상태변이를 감소하기 위해 데이터의 제어흐름과 연산을 추상하는 것 원하는 결과를 얻기 위한 비즈니스 로직이 담겨 있는 고수준의 연산을 일련의 단계들로 체이닝하는, 간결한 흐름 중심의 모델을 선호합니다. 함수의 서술부와 평가부를 분리하는 함수 합성의 미학이자 표현식이 호출 될 때까지 느긋하게 미루다가 평가하는 사상입니다.

외부에서 관찰 가능한 부수효과가 제거된 불변 프로그램을 작성하기 위해 순수함수를 선언적으로 평가하는 것을 말합니다.

부수효과가 없다는 것: 함수의 출력은 오로지 그 함수에 입력된 값에만 의존성을 가진다는 의미이다.

순수함수

외부의 인자를 변화시키지 않고 최대한 지역변수만을 써서 구현되는 함수입니다. 인수로 넘기는 값의 본체조차 수정되면 안됩니다. 즉, 함수인수로 넘길 때 객체에 변이를 일으키지 않도록 주의를 하며 deep copy를 한 후 변이를 하던가 등의 방법을 써야 합니다.

클로저

클로저는 독특한 함수체제를 멋지게 활용할 수 있습니다.

- 프라이빗 변수를 모방
- 서버 측 비동기 호출
- 가상의 블록 스코프 변수를 생성

모나드

값은 컨테이너화 하는 행위는 함수형 프로그래밍의 기본 디자인패턴 연산을 컨테이너에 매핑하고 체이닝 가능한 표현식 또는 계산일뿐이라서 마치 공장 컨베이어 벨트처럼 순서대로 흘려 보내면서 단계별로 부가 처리를 할 수 있게 배열할 수 있습니다.

```
fg('{"k: 10"}).catch(_ => '미안...').then(log);
```

이 때.. chain 패턴이나 `_compose`, `Promise` 등의 일종의 파이프라인 혹은 모나드 등에서 아쉬운점은 함수 모음의 첫번째 함수를 제외하고는 인자를 하나만 받을 수 있다는 점입니다.

애러처리

예외를 던지는 행위는 참조 투명성 원리에 위배됩니다. 함수 호출 범위를 벗어나 전체 시스템에 영향을 미치는 부수 효과를 일으킵니다 비지역성 원리위배, 지역스택과 환경에 벗어납니다. 이는 모나드를 통해 해결합니다.

참조투명성

순수함수와 비슷한 개념, 같은 입력에 같은 결과를 반환해야 한다. 즉, 동일입력을 동일결과에 매핑해야 합니다. 수학적인 형태로 프로그램을 해아릴 수 있다. 외부변수에 종속되지 않고, 동일한 입력을 받았을 때 동일한 결과를 내면 참조 투명한 함수

```
var c = 0;
function f(){
    return c++;
}

const f = c => c + 1
```

테스트용이 / 전체로직 파악 용이합니다. `_chain` 은 `value()`에 시동을 걸면 전체 함수 순차열을 몽땅 실행하게 한다.

파이프 & 체이닝

- 체이닝 : 단단한 결합, 제한된 표현성
- 파이프 : 느슨한 결합, 유연성
- 파이라이닝은 함수를 연결하는 또 다른 기법입니다.

[나중에]

docker

<https://subicura.com/2017/01/19/docker-guide-for-beginners-1.html>

Redis

Redis는 데이터베이스의 여러 솔루션 중 하나로 메모리를 사용하는 키, 밸류 형식의 데이터베이스이다. Redis의 최고 장점은 메모리를 사용하기 때문에 매우 빠른 속도를 자랑한다. 일반적인 하드 디스크에 비하여 상대적으로 엄청나게 빠를 수도 있다. 다만 메모리라는 제약으로 공간이 크지 않고 키, 밸류(Key, Value) 형식의 입출력 방식이기에 복잡한 데이터베이스에 적합하지 않다. 위 내용만으로도 장점과 단점이 너무 극명하다. 빠르나 효과적인 주 데이터베이스로 사용하기에 다소 어려워보이는 점이다. 또한 공간이 많아질수록 비용이 급격히 올라가므로 이를 고려한다면 Redis는 빛좋은 개살구일 수 있다... 하지만! 대부분 Redis를 보조 데이터베이스 역할로 그 가치를 최대한 끌어내어 활용하고 있다. 그 중 Redis에 가장 적합한 것이 바로 데이터베이스 캐싱(Caching)이다.

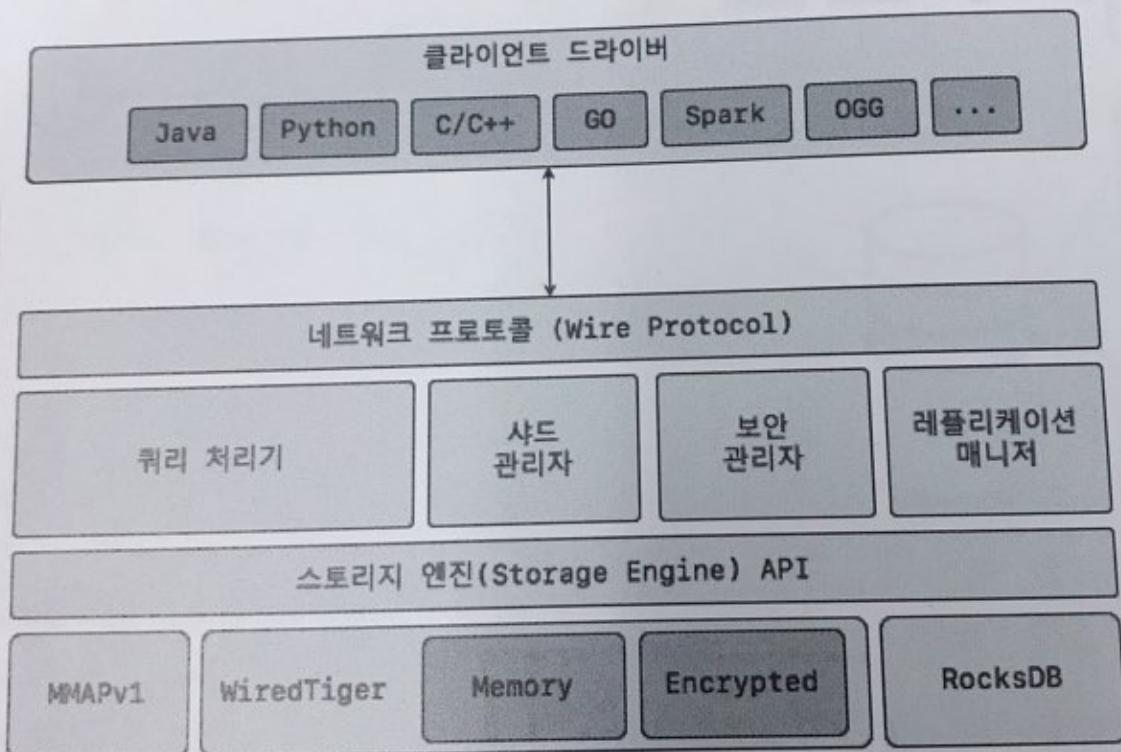
MongoDB

대용량INSERT에 대한 요건이 커지면서 NOSQL DBMS가 도입 시작되었고 방대한 양의 데이터를 충분히 빠른 속도로 처리할 수 있는 데이터베이스입니다. 먼저 NoSQL, Not Only SQL이며 관계형 DB가 아닌 유동적인 DB라는 특징이 있습니다. NoSQL은 데이터의 일관성을 약간 포기한 대신 여러 대의 컴퓨터에 데이터를 분산하여 저장하는 것(Scale-out : 수평적 확장)을 목표로 등장하였습니다. NoSQL의 등장으로 작고 값싼 장비 여러 대로 대량의 데이터와 컴퓨팅 부하를 처리하는 것이 가능하게 되었고 유연성, 확장성 등이 RDBMS에 비해 좋습니다.

22 Real MongoDB

2.1 플러그인 스토리지 엔진

스토리지 엔진은 사용자의 데이터를 디스크와 메모리에 저장하고 읽어오는 역할을 담당하는 MongoDB 서버를 구성하는 컴포넌트를 도식화한 것이다.

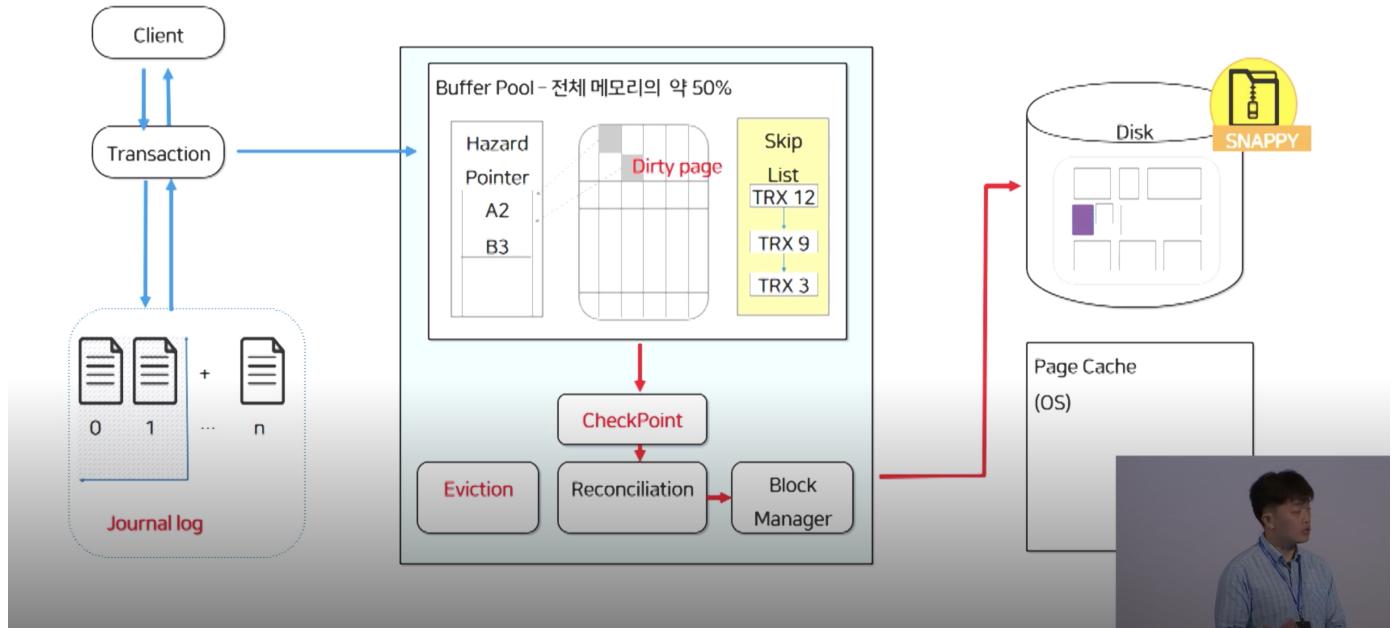


〈그림 2-1〉 MongoDB 아키텍처

그림 2-1에서 가장 하위에 위치한 “MMAPv1”과 “WiredTiger” 등을 스토리지 엔진으로 부른다. MongoDB는 MySQL과 같은 관계형 데이터베이스와 같은 방식으로 데이터를 디스크에 영구적으로 기록하거나 다시 읽어오는 역할을 담당한다. MongoDB는 MySQL과 같은 방식으로 데이터를 디스크에 영구적으로 기록하거나 다시 읽어오는 역할을 담당한다. MongoDB는 MySQL과 같은 방식으로 데이터를 디스크에 영구적으로 기록하거나 다시 읽어오는 역할을 담당한다. MongoDB는 MySQL과 같은 방식으로 데이터를 디스크에 영구적으로 기록하거나 다시 읽어오는 역할을 담당한다. 하지만 MongoDB는 MySQL과 같은 방식으로 데이터를 디스크에 영구적으로 기록하거나 다시 읽어오는 역할을 담당한다.

스토리지엔진 : 디스크에서 데이터를 어떻게 가져오고 어떻게 최적으로 저장할 것인지 결정하는 부분이 스토리지 엔진이다

3.6 & 7 eviction & Checkpoint



서버안에 MongoDB 자체가 전체 메모리의 메모리 버퍼를 전체의 50%를 씁니다. 때문에 한 서버안에 프라이머리 / 세컨더리 두개 절대 놓지 않습니다.

메모리가 부족할 경우 **eviction** 이 잘되지 않고 메모리버퍼를 아예 지우는 작업을 하고 심할 경우 데몬이 죽게 됩니다.

- **eviction** : 메모리버퍼에서 필요없는 데이터를 삭제하는 작업
- **체크포인트** : 메모리 버퍼와 디스크사이의 데이터 불일치를 해소하기 위해 메모리에서 디스크로 data 동기화를 하는 작업

특징

1. 그리고 한개의 DATA 구조는 key - value 형태로 이루어져 있으며 _id라는 고유한 아이디를 가집니다.
2. 스키마 없이 데이터모델을 구현하지 않은채 그냥 유동적으로 데이터를 삽입할 수 있습니다.
3. 스키마 : 데이터베이스를 구성하는 속성, 관계 등 데이터 값들이 갖는 type들을 명시 해놓은 것을 말합니다.
4. 장점 : 다양한 서비스로부터 데이터를 쌓을 때 유동적으로 쌓을 수 있음
5. 단점 : RDBMS인 경우에는 int, char[14]인 경우에는 18바이트 그러나 몽고디비는 한 도큐먼트로 칼럼이름또한 바이트에 추가가 되기 때문에 바이트가 더 커집니다. 즉, 공간적인 소모가 많음.
6. min, max, aggregate, mapReduce 등 강력한 함수로 데이터를 뽑아내고 조합하는 등의 활동을 할 수 있습니다. data들을 모아서 최대값을 뽑아내거나 최소값 등등의 행위를 할 수 있습니다.
7. ReplicaSet을 이용해 이중화가 가능합니다.
8. BSON, Binary Json 형태로 저장합니다. json 형태가 아닙니다.
9. geoSpatial 인덱스를 써서 2Dimension 좌표 search에 활용이 가능합니다. 이것을 인덱싱이 자유롭다고 합니다.
10. \$lookup을 통한 collection조인이 가능
11. 쿼리정렬시 애러 발생할경우 다음과 같이 해결할 수 있다.

- 정렬 작업에 인덱스를 활용
 - 메모리 공간을 크게 놓는다.
 - find명령어 대신 aggregate명령을 사용하고 allowDiskUse옵션을 true로 설정하는 것
12. TDE를 통해 보안 적용가능(범위검색에 경우 좋음) 응용프로그램으로 보안이 가능(bcrypt)하지만 범위검색의 경우 수행이 불가함.
13. 3.4이상 wired Tiger엔진 사용
- LSM_Tree(로그 기반 병합트리)를 이용 읽기 성능을 포기하고 그만큼의 저장 성능을 향상시킨 솔루션 / 느린 읽기 성능을 보완하기 위해 블룸 필터를 사용
14. Background Index생성시 DB 자체에 Lock이 걸림
15. Mongoose는 빠르지가 않아서 권장하지 않으며 Node.js 2.x 드라이버 권장하며 3.x는 커넥션 문제가 발생됩니다.
16. 기본적으로 JSON도큐먼트를 인자로 사용
17. 도큐먼트에서 _id라는 필드가 자동으로 그 도큐먼트의 프라이머리 키로 선정된다. 나머지의 인덱스는 세컨더리 키로 선정된다. JSON 값을 문자열로 그대로 저장하는 것이 아니라 문자열 기반의 JSON텍스트를 BSON, Binary JSON형태로 저장한다.
18. 색인환경에서 index 설정시 각 색인환경안에서만 유니크하다.
19. 인덱스의 기본 정렬은 항상 오름차순으로 구현돼 있지만 이 인덱스를 읽는 방향에 따라서 오름차순 또는 내림차순의 효과를 얻을 수 있으며 인덱스에서 각 필드의 정렬을 오름차순, 내림차순으로 결정할 수 있으며 대부분의 인덱스는 B-Tree를 적용한다.(인덱스 정렬은 B-tree에서만 가능)
20. 디스크에 데이터를 저장하는 가장 기본단위를 페이지 또는 블록이라고 한다.

Hbase와 비교

조회조건에 따라서 세컨더리 인덱싱이 필요하는데 Hbase는 그걸 하지 못하며 하둡에 올라가서 물리적으로 분리가 되어 있으나 MongoDB는 색드 단위로 논리적으로 분리가 되어있음 wired tiger로 높은 수준의 ACID를 할 수 있다. 또 한..REST API를 사용하니까 편리하게 JSON을 제공할 플랫폼이 필요! 이를 위한 MongoDB

색인이란?

데이터베이스 서버를 확장하려면 데이터베이스의 데이터가 여러 서버로 분산될 수 있게 미리 응용 프로그램을 설계하고 개발하는 것을 말합니다.

- 스케일업 : 단일서버 증가
- 스케일아웃 : 여러대의 서버로 확장

ACID

ACID(원자성, 일관성, 독립성, 지속성)는 데이터베이스 트랜잭션이 안전하게 수행되는 것을 보장하는 특성 집합이다. 데이터베이스에서 데이터에 대한 하나의 논리적 실행단계를 트랜잭션이라고 한다.

- 원자성(Atomicity)은 트래잭션과 관련된 일들이 모두 수행되었는지 아니면 모두 실행이 안되었는지를 보장하는 능력이다. 자금 이체는 성공할 수도 실패할 수도 있지만 원자성은 중간 단계까지 실행되고 실패하는 일은 없도록 하는 것이다.

- 일관성(Consistency)은 트랜잭션이 실행을 성공적으로 완료하면 언제나 일관성 있는 데이터베이스 상태로 유지하는 것을 의미한다. 시스템이 가지고 있는 고정요소는 트랜잭션 전과 후가 동일해야 한다.
- 독립성(Isolation)은 트랜잭션을 수행 시 다른 트랜잭션의 연산 작업이 끼어들지 못하도록 보장하는 것을 의미한다. 이것은 트랜잭션 밖에 있는 어떤 연산도 중간 단계의 데이터를 볼 수 없음(참조를 하지 못한다)을 의미한다.
- 지속성(Durability)은 성공적으로 수행된 트랜잭션은 영원히 반영되야 함을 의미한다. 시스템 문제가 발생되도 유지되어 함을 의미한다. 전형적으로 모드 트랜잭션은 로그로 남고 시스템 장애 발생 전 상태로 되도릴 수 있다. 트랜잭션은 로그에 모든 것이 저장된 후에만 commit 상태로 간주될 수 있다.

DR

Disaster Recovery 서비스는 재해 복구 시스템입니다. IDC별, 전산실별, 지점별 등 지역적으로 분리된 서버들에 대해 무정지 서비스 가능하게 합니다. 데이터의 이중화 구성 가능합니다. 갑작스런 네트워크의 단절, 네트워크 노드의 불안정, 장비의 다운, 정전 등으로 인한 문제를 해결합니다. auto failover도 같은 뜻이다.

쿼리튜닝

쿼리를 튜닝한다는 것은 처리에 꼭 필요한 데이터만 읽도록 쿼리를 개선하는 것을 말합니다. 인덱스 선택을 얼마나 많이 하느냐 / 얼마나 많은 도큐먼트를 읽느냐에 따라 잘하느냐를 판단할 수 있습니다.

Node.js

Chrome V8 Javascript 엔진으로 빌드 된 비동기 이벤트 주도 Javascript 런타임, 확장성 있는 네트워크 애플리케이션을 만들 수 있도록 설계되며 이벤트 기반, 논 블로킹 I/O 모델을 사용해 가볍고 효율적인 "서버사이즈 플랫폼, 코드를 실행할 수 있는 Javascript 런타임" 즉, 자바스크립트 프로그램이 실행되고 있을 때 존재하는 곳을 말합니다.

D3.js

시각화 자바스크립트 라이브러리 SVG, Canvas and HTM DOM manipulation

JWT

JSON Web Token (JWT) 은 웹표준 (RFC 7519) 으로서 두 개체에서 JSON 객체를 사용하여 가볍고 자가수용적인 (self-contained) 방식으로 정보를 안전성 있게 전달해줍니다.

수많은 프로그래밍 언어에서 지원됩니다 JWT 는 C, Java, Python, C++, R, C#, PHP, JavaScript, Ruby, Go, Swift 등 대부분의 주류 프로그래밍 언어에서 지원됩니다.

자가 수용적 (self-contained) 입니다 JWT 는 필요한 모든 정보를 자체적으로 지니고 있습니다. JWT 시스템에서 발급된 토큰은, 토큰에 대한 기본정보, 전달 할 정보 (로그인시스템에서는 유저 정보를 나타내겠죠?) 그리고 토큰이 검증됐다는 것을 증명해주는 signature 를 포함하고있습니다.

쉽게 전달 될 수 있습니다 JWT 는 자가수용적이므로, 두 개체 사이에서 손쉽게 전달 될 수 있습니다. 웹서버의 경우 HTTP의 헤더에 넣어서 전달 할 수도 있고, URL 의 파라미터로 전달 할 수도 있습니다.

회원 인증: JWT 를 사용하는 가장 흔한 시나리오 입니다. 유저가 로그인을 하면, 서버는 유저의 정보에 기반한 토큰을 발급하여 유저에게 전달해줍니다. 그 후, 유저가 서버에 요청을 할 때마다 JWT를 포함하여 전달합니다. 서버가 클라이언트에게서 요청을 받을 때마다, 해당 토큰이 유효하고 인증됐는지 검증을 하고, 유저가 요청한 작업에 권한이 있는지 확인하여 작업을 처리합니다. 서버측에서는 유저의 세션을 유지 할 필요가 없습니다. 즉 유저가 로그인되어 있는지 안되어 있는지 신경 쓸 필요가 없고, 유저가 요청을 했을 때 토큰만 확인하면 되니, 세션 관리가 필요 없어서 서버 자원을 많이 아낄 수 있죠. **정보 교류:** JWT는 두 개체 사이에서 안정성 있게 정보를 교환하기에 좋은 방법입니다. 그 이유는, 정보가 sign 이 되어 있기 때문에 정보를 보낸이가 바꿔진 않았는지, 또 정보가 도중에 조작되지는 않았는지 검증할 수 있습니다.

헤더 / 내용 / 서명으로 이루어져 있다.

쿠키와 세션의 차이

쿠키

쿠키는 클라이언트(브라우저) 로컬에 저장되는 키와 값이 들어있는 작은 데이터 파일이다. 사용자 인증이 유효한 시간을 명시할 수 있으며, 유효 시간이 정해지면 브라우저가 종료되어도 인증이 유지된다는 특징이 있습니다. 쿠키는 클라이언트의 상태 정보를 로컬에 저장했다가 참조한다. 클라이언트에 300개까지 쿠키저장 가능, 하나의 도메인당 20개의 값만 가질 수 있음, 하나의 쿠키값은 4KB까지저장 Response Header에 Set-Cookie 속성을 사용하면 클라이언트에 쿠키를 만들 수 있다. 쿠키는 사용자가 따로 요청하지 않아도 브라우저가 Request시에 Request Header를 넣어서 자동으로 서버에 전송한다.

세션 (Session)

세션은 쿠키를 기반하고 있지만, 사용자 정보 파일을 브라우저에 저장하는 쿠키와 달리 세션은 서버 측에서 관리합니다. 서버에서는 클라이언트를 구분하기 위해 세션 ID를 부여하며 웹 브라우저가 서버에 접속해서 브라우저를 종료할 때까지 인증상태를 유지합니다. 물론 접속 시간에 제한을 두어 일정 시간 응답이 없다면 정보가 유지되지 않게 설정이 가능 합니다. 사용자에 대한 정보를 서버에 두기 때문에 쿠키보다 보안에 좋지만, 사용자가 많아질수록 서버 메모리를 많이 차지하게 됩니다.

즉 동접자 수가 많은 웹 사이트인 경우 서버에 과부하를 주게 되므로 성능 저하의 요인이 됩니다. 클라이언트가 Request 를 보내면, 해당 서버의 엔진이 클라이언트에게 유일한 ID를 부여하는 데 이것이 세션ID다.

세션의 동작 방식

1. 클라이언트가 서버에 접속 시 세션 ID를 발급
2. 클라이언트는 세션 ID에 대해 쿠키를 사용해서 저장 (이 때 쿠키 이름은 JSESSIONID이다.)
3. 클라이언트가 서버에 다시 접속 시 이 쿠키를 이용해서 세션 ID 값을 서버에 전달

세션의 특징

1. 각 클라이언트에게 고유 ID를 부여
2. 세션 ID로 클라이언트를 구분해서 클라이언트의 요구에 맞는 서비스를 제공
3. 보안 면에서 쿠키보다 우수
4. 사용자가 많아질수록 서버 메모리를 많이 차지하게 됨

세션의 사용 예

로그인과 같이 보안상 중요한 작업을 수행할 때 사용

차이

쿠키와 세션은 비슷한 역할을 하며, 동작원리도 비슷합니다 그 이유는 세션도 결국 쿠키를 사용하기 때문입니다. 그런데 가장 큰 차이점은 사용자의 기록 정보가 저장되는 위치입니다. 때문에 쿠키는 서버의 자원을 전혀 사용하지 않으며, 세션은 서버의 자원을 사용합니다. 보안 면에서 세션이 더 우수하며, 요청 속도는 쿠키가 세션보다 더 빠릅니다. 그 이유는 세션은 서버의 처리가 필요하기 때문입니다.

- 보안, 쿠키는 클라이언트 로컬에 저장되기 때문에 변질되거나 request에서 스나이핑 당할 우려가 있어서 보안에 취약하지만 세션은 쿠키를 이용해서 sessionid 만 저장하고 그것으로 구분해서 서버에서 처리하기 때문에 비교적 보안성이 좋다.
- 라이프 사이클, 쿠키도 만료시간이 있지만 파일로 저장되기 때문에 브라우저를 종료해도 계속해서 정보가 남아 있을 수 있다. 또한 만료기간을 넉넉하게 잡아두면 쿠키삭제를 할 때 까지 유지될 수도 있다.
- 반면에 세션도 만료시간을 정할 수 있지만 브라우저가 종료되면 만료시간에 상관없이 삭제된다.
- 속도, 쿠키에 정보가 있기 때문에 서버에 요청시 속도가 빠르고 세션은 정보가 서버에 있기 때문에 처리가 요구되어 비교적 느린 속도를 낸다.

세션을 주로 사용하면 좋은데 왜 쿠키를 사용할까?

세션은 서버의 자원을 사용하기 때문에 무분별하게 만들다보면 서버의 메모리가 감당할 수 없어질 수가 있고 속도가 느려질 수 있기 때문이다.

쿠키, 세션은 캐시와 엄연히 다르다.

캐시는 이미지나 css, js파일 등이 사용자의 브라우저에 저장이 되는 것이다. 이를 이용해 자원이 아껴지는 것 한번 캐시에 저장되면 브라우저를 참고하기 때문에 서버에서 변경이 되어도 사용자는 변경되지 않게 보일 수 있는데 이런 부분을 캐시를 지워주거나 서버에서 클라이언트로 응답을 보낼 때 header에 캐시 만료시간을 명시하는 방법 등을 이용할 수 있다.

세션은 사용자의 수 만큼 서버 메모리를 차지하기 때문에 요즘은 이런 문제들을 보완한 토큰 기반의 인증방식을 사용하는 추세입니다. 그 중 JWT(JSON Web Token)라는 것이 있습니다.

this

<https://codeburst.io/the-simple-rules-to-this-in-javascript-35d97f31bde3>

1. 함수를 호출할 때 `new` 키워드를 사용하는 경우 함수 내부에 있는 `this` 는 완전히 새로운 객체입니다.
2. `apply` , `call` , `bind` 가 함수의 호출 / 작성에 사용되는 경우 함수 내의 `this` 는 인수로 전달된 객체입니다.
3. `obj.method()` 와 같이 함수를 메서드로 호출하는 경우 `this` 는 함수가 프로퍼티인 객체입니다.

```
var obj = {
  value: 'hi',
  printThis: function() {
    console.log(this);
  }
};
var print = obj.printThis;
obj.printThis(); // -> {value: "hi", printThis: f}
print(); // -> Window {stop: f, open: f, alert: f, ...}
```

이렇게 함수호출패턴에 따라 달라지게 됩니다.

4. 함수가 자유함수로 호출되는 경우 즉 위의 조건 없이 호출되는 경우 `this` 는 전역 객체입니다. 브라우저에서는 `window` 객체입니다. 엄격 모드(`'use strict'`)일 경우 `this` 는 전역 객체 대신 `undefined` 가 됩니다.
5. 위의 규칙 중 다수가 적용되면 더 높은 규칙이 승리하고 `this` 값을 설정합니다.
6. 함수가 ES2015 화살표 함수인 경우 위의 모든 규칙을 무시하고 생성된 시점에서 주변 스코프의 `this` 값을 받습니다.

```
setTimeout(function(){
  this.a
}, 100);
```

에서 `this.a` 는 `window`의 `a` 를 가리키게 된다. 왜냐면 함수가 호출되는 위치 즉, `window object`로부터 나온 `setTimeout` 함수이기 때문입니다.

이벤트위임

애플리케이션을 제작할 때 사용자가 페이지 요소를 조작할 수 있도록 페이지의 버튼, 텍스트, 이미지 등에 이벤트를 붙여야 할 때가 있습니다. 예를 들어, 면접에서 면접관이 todo 리스트 애플리케이션 제작에 대해 질문하는 경우 “해당 리스트의 아이템에 대해서 사용자가 클릭할 때 이벤트가 일어나도록 구현하라” 라고 하면서 아래와 같은 샘플을 주겠죠.

```
<ul id="todo-app">
  <li class="item">Walk the dog</li>
  <li class="item">Pay bills</li>
  <li class="item">Make dinner</li>
  <li class="item">Code for one hour</li>
</ul>
```

그렇다면 아마 대부분이 아래와 같이 구현할 것입니다.

```
document.addEventListener('DOMContentLoaded', function() {

  let app = document.getElementById('todo-app');
  let items = app.getElementsByClassName('item');

  // 각 아이템에 이벤트 리스너를 등록합니다.
  for (let item of items) {
    item.addEventListener('click', function() {
      alert('you clicked on item: ' + item.innerHTML);
    });
  }
});
```

위 코드는 제대로 동작하지만 문제점은 리스트의 아이템 각각에 이벤트를 붙이고 있는 것입니다. 아이템 요소가 위와 같아 4개일 때는 상관이 없지만 만약 10,000 개라면 어떻게 될까요? 위 함수는 10,000 개의 분리된 이벤트 리스너를 생성하고 그걸 각각 DOM에 등록할 것입니다. 이는 매우 비효율적이죠.

이런 면접에서는 먼저 면접관에게 사용자가 최대로 입력할 수 있는 요소의 개수를 물어보는 것이 좋습니다. 최대 갯수가 10개가 넘지 않는다면, 위 코드는 문제가 없을 테니까요. 하지만 만약 사용자가 입력할 수 있는 아이템 수가 무한개라면 더 효율적인 해결책을 찾아야 합니다.

아이템 갯수마다 이벤트 리스너를 생성, 등록 하는 것보다는 모든 아이템 리스트에 대해서 한 개의 이벤트 리스너를 생성하여 전체 영역에 등록하는 것이 훨씬 효율적이죠. 그렇게 하면 사용자가 해당 아이템을 선택했을 때 이벤트 리스너가 해당 아이템에 대해서 이벤트를 발생시킵니다. 이를 우리는 이벤트 위임이라고 합니다. 각각의 이벤트 핸들러를 붙이는 방식보다 훨씬 효율적이죠. 아래 코드는 위의 이벤트 위임을 구현한 것입니다.

```
document.addEventListener('DOMContentLoaded', function() {
```

```

let app = document.getElementById('todo-app');

// 리스트 아이템의 전체 영역에 이벤트 리스너를 등록합니다.
app.addEventListener('click', function(e) {
  if (e.target && e.target.nodeName === 'LI') {
    let item = e.target;
    alert('you clicked on item: ' + item.innerHTML);
  }
});

});

});

```

마크업이미지

- GIF파일 - 다양한 색상의 이미지에는 적절x
- PNG-8보다는 PNG-24가 다양한 색상을 가진 이미지에 적절
- JPEG파일이 파일용량 / 이미지까지지 않음
- but, 단순색상 이미지는 PNG-8이 최고
- 반투명 이미지는 PNG-24가 최고

이벤트 정지 함수정리

event prevent에 관한 함수 정리

	stop bubbling	prevent default action	prevent "same element" event handlers
return false	Yes	Yes	No
preventDefault	No	Yes	No
stopPropagation	Yes	No	No
stopImmediatePropagation	Yes	No	Yes

Git

git은 형상관리 툴이라고 보면 되는데 말그대로 프로젝트의 시작과 끝, 개발과 배포를 담당하는 녀석입니다. 독립적으로 이 프로젝트를 a방향으로 나아가게 한다음에 pull request라는 것을 보내서 승인이 되면 프로젝트에 a방향이 더해지거나 그쪽으로 방향이 틀어지게 됩니다. 독립적이니까 프로그램의 자유성도 보장된다. 그리고 유연성있게 프로젝트를 진행 할 수 있으며 브랜치라는 가지를 이용해서 각각 하고 싶은 주제를 기반으로 코드를 발전시켜서 이 주제는 이런 브랜치로 이렇게 수정되었구나라는 것을 쉽게 알 수 있으며 각각의 브랜치마다의 모든 코드를 저장하는것이 아닌 다른 부분들만

저장해서 공간을 차지하는 것도 낫습니다. 우리가 항상 버전이라는 것을 관리한다고 하는데 버전은 이전상태와 다른 특정한 형태를 지칭하며 git은 이 다른 부분만 저장해서 저장 효율도 높였습니다.

git은 각 커밋내용에 기반한 ID로 체크섬을 구현합니다.

- 체크섬은 전송된 데이터확인에 쓰이는 일종의 디지털 지문입니다. 그래서 정확한 데이터를 보존할 수 있습니다.

Git 명령어

- `git add` : git이라는 녀석에 내가 설정한 파일을 관리하라는 뜻으로 수정/추가/삭제한 파일들의 목록들의 스냅샷을 찍어 기록에 올립니다.
- `git commit` : add의 다음 단계이자 확정된 수정본을 뜻합니다.
- `git clone [url] [저장되는폴더명]` : 다른 사람의 repository를 가져와서 저장합니다.
- `git checkout -b kundol` : kundol이라는 브랜치를 만들고 그 브랜치로 이동합니다.
- `git push origin master` : origin은 git으로 호스팅하는 하나의 공유사본을 뜻합니다. 바꿀 수 있지만 보통 origin으로 사용됩니다.
- `git pull origin master` : 작업하는 브랜치를 최신 master의 내용으로 바꿉니다.
- `git log --pretty=oneline` : git에 대한 로그 확인할 수 있습니다.
- `git diff` : 병합할 때 중복되는 작업부분에 대해서 차이점을 잘 말해줍니다.
- `git commit --amend` : commit할 때 메세지를 정했는데 맘에 안들 때가 있습니다. 이를 통해 수정할 수 있습니다. 실행시키면 vi편집기가 뜹니다. a를 눌러서 insert모드로 바꾸고 마지막에 esc를 누른 후 wq를 눌러 저장시켜 줍니다.