

3장 코드리팩토링과 함수형프로그래밍

프로젝트안의 코드는 모두 아름다워야 합니다. 훌륭한 목수는 장롱뒤쪽에도 좋은 나무를 쓰듯이 겉으로 보이지 않는 코드라도 하나하나 꼼꼼히 짜야 합니다. 어떤 함수를 쓰는지 어떤 로직이 더 효율적인지를 신경쓰고 버그가 생기지 않게 올바르게 객체를 복사하고 변수명을 짓는 법을 통일하고 어떻게 모듈화를 할건지, 그리고 객체지향인지 절차지향적인지 등 프로그래밍 패러다임을 정하고 코드를 짜는 것이 중요합니다. 이러한 것들을 코드리팩토링, 코드최적화라 부르며 이 책에서는 패러다임 중 함수형프로그래밍이라는 패러다임을 공부할 것입니다.

우리 모두 훌륭한 목수가 되봅시다!

화학을 전공한 오펜하이머는 하버드 재학 시절에는 잠자는 것도, 먹는 것도 잊을 정도로 공부에 몰두했다. 아침마다 일어나면 가장 먼저 실험실로 향했고, 배고픔도 잊은 채 매일같이 노력을 쏟았다. 점심시간이 되면 빵 두 장 사이에 땅콩버터와 초콜릿 잼을 발라 먹는 것이 다였다. 서둘러 점심을 먹고 나면 그는 다시 실험에 집중했다. 시간을 아끼고 엄청난 노력을 쏟은 결과 그는 불과 3년 만에 졸업에 필요한 모든 학점을 이수했고, 최우수 성적으로 졸업했다.

- 오펜하이머(Julius Robert Oppenheimer)

03-1 함수성능비교

먼저 똑같은 로직을 구현할 수 있는 함수라도 성능이 좋은 것을 써야 합니다.

정수형변환 : parseInt와 비트연산자

실수형을 정수형으로 변환할 때 보통 parseInt를 씁니다. parseInt(2.5)를 하게 되면 정수형 2가 나오는 것은 자명합니다. 하지만 이 함수 말고도 비트연산자를 사용해서 정수형으로 변환할 수 있습니다. 먼저 비트연산자가 뭔지 보겠습니다.

비트연산자 ~ 와 ~~

```
s = 0000000001000000000000
~s = 11111111101111111111
```

~라는 비트연산자는 숫자 s의 비트를 뒤집어 버리는 비트연산자입니다. 뒤집으면 어떤 수가 될까요? 예를 들어 2는 -3이 됩니다. 즉, $\sim s = -(n + 1)$ 이 되는 셈이죠. ~~를 하면 뭐가 될까요? ~라는 비트를 전체를 뒤집는 비트연산자를 한 번 더 한 것이 되고 바로 소수점을 버리게 됩니다. 이는 자바스크립트의 비트연산자를 통해 적용한 결과는 부호 있는 32비트 정수로 변환된다는 ECMA5 Spec에 따라 그렇게 됩니다.

자 이렇게 두 함수 모두 소수점을 버려서 정수형으로 만들 수 있다는 것을 알았습니다. 그렇다면 성능비교를 하면 어떻게 될까요?

parseInt VS ~~

```
for(let i = 0; i < 100000; i++){
  let a = 123.456789
  b = parseInt(a)
}
```

위의 코드를 비트연산자를 활용한 것으로 바꿔보겠습니다.

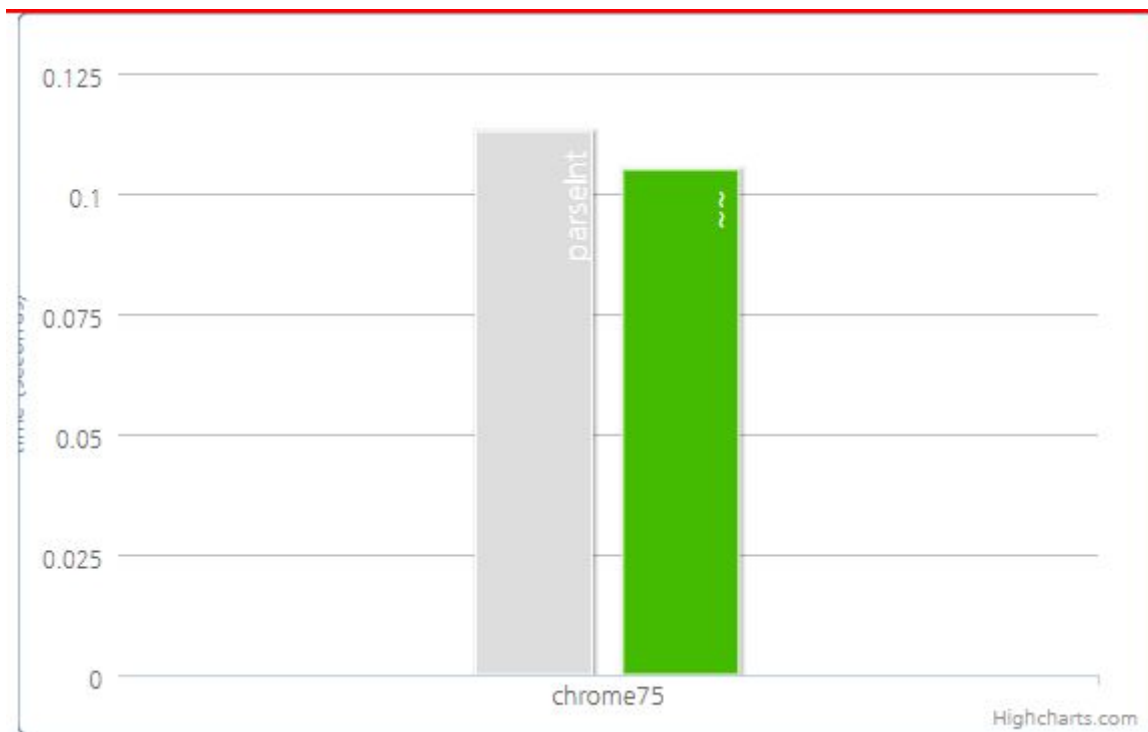
```
for(let i = 0; i < 100000; i++){
  let a = 123.456789
  b = ~~(a)
}
```

바로 위처럼 됩니다. 그렇다면 얼마만큼의 성능적인 장점이 있을까요? JSmatch 라는 네이버에서 만든 비교사이트를 통해 비교해봅시다.

JSmatch

JSmatch는 네이버에서 만든 성능 테스트 사이트입니다. 또한 성능사이트는 jsperf라는 곳도 있습니다. 성능적으로 비교하고 싶은 코드가 있다면 사용해주시면 됩니다.

링크 : <http://jindo.dev.naver.com/jsMatch/index.html>



<< 03-writer-01.PNG >>

이러한 비트 연산자는 실제로도 많이 쓰입니다.

D3.js의 d3-geo-projection

```
// Calculate F(phi|m) where m = k^2 = sin^2 alpha.
// See Abramowitz and Stegun, 17.6.7.
export function ellipticF(phi, m) {
  if (!m) return phi;
  if (m === 1) return log(tan(phi / 2 + quarterPi));
  var a = 1,
      b = sqrt(1 - m),
      c = sqrt(m);
  for (var i = 0; abs(c) > epsilon; i++) {
    if (phi % pi) {
      var dPhi = atan(b * tan(phi) / a);
      if (dPhi < 0) dPhi += pi;
    }
  }
}
```

```

    phi += dPhi + ~~(phi / pi) * pi;
  } else phi += phi;
  c = (a + b) / 2;
  b = sqrt(a * b);
  c = ((a = c) - b) / 2;
}
return phi / (pow(2, i) * a);
}

```

유명한 자바스크립트 시각화 라이브러리인 D3.js의 d3-geo-projection의 한 부분입니다. 이렇듯 정수형 변환을 할 때 ~~를 쓰는 모습을 볼 수 있습니다.

배열포함 확인 : indexOf 와 includes

어떠한 요소가 들어가있음을 확인할 때 어떤 함수를 써야 할까요? 크게 indexOf와 includes가 있습니다. 둘 중에 includes가 성능이 더 좋을 때가 있습니다. 각각의 경우의 수에 따라 다르니 직접 어떠한 로직을 구현할지를 생각하고 테스트해보고 함수를 결정하는 것이 중요합니다.

이번에는 jsperf에서 성능비교를 진행해보겠습니다.

<https://jsperf.com/includesvsindexof-kundol2>

위의 경우는 includes가 더 좋게 나옵니다.

```

//includes
let a = ["홍철", "윤찬", "강우"];
if(a.includes("강우")) return true;
//indexof
let a = [];
if(a.indexOf("강우") !== -1) return true;

```

위의 코드를 비교한 것입니다.

하지만 <https://jsperf.com/includesvsindexof-kundol> 의 경우는 indexOf가 더 좋게 나옵니다.

```

//includes
let a = [];
for(let i = 1; i <= 10000; i++)a.push(i);
if(a.includes(5000)) return true;
//indexof
let a = [];
for(let i = 1; i <= 10000; i++)a.push(i);
if(a.indexOf(5000) !== -1) return true;

```

위의 코드를 비교한 것입니다.

객체선언 : new Object와 {}

객체를 선언할 때는 어떠한 방법을 써야 할까요? 크게 new Object와 {}로 리터럴로 선언하는 방법 두가지가 있습니다. 이또한 비교해보겠습니다.

<https://jsperf.com/object-declaration-kundol/1>

위의 처럼 보듯이 new Object가 좋은 것을 볼 수 있습니다.

```
for(let i = 0; i < 1000; i++){
  let a = {};
}

for(let i = 0; i < 1000; i++){
  let a = new Object();
}
```

비교 코드입니다.

지금까지 간단하게 성능을 좋게 만들어주는 함수, 코드를 보았습니다. 위의 코드처럼 간단하게 성능을 올릴 수 있지만 $O(n^2)$ 알고리즘을 $O(n \log n)$ 알고리즘으로 바꾸는 등 조금 복잡하게 로직을 다시 구성해서 성능을 좋게 만들 수도 있습니다. 하지만 이 결과값이 밀리세컨드 등 미세하게 나타나는 것이라면 그것은 시간낭비일 뿐입니다. 한정된 개발계획에 맞춰 그러한 것들을 고려하는 것이 중요합니다.

03-2 올바르게 객체 복사하기

우리는 정수나 실수 등을 다루지만 보통 객체를 다룰 때가 많습니다. 객체를 다룰 땐 항상 주의를 해야 합니다. 왜냐하면 자바스크립트에서는 보통 정수선언처럼 객체를 할당하게 되면 **참조하는 방법으로 간접복사** 되기 때문에 a를 b에 할당 시킨 뒤, b가 변하면 a도 변화하기 때문입니다. 그래서 자바스크립트에서의 객체는 제대로 복사해야 합니다. 아래의 3가지의 방법이 있습니다.

스프레드 연산자와 Object.assign

먼저 스프레드 연산자와 Object.assign입니다. 이 2가지 방법은 깊이가 1까지만 유효합니다.

```
const a = {"b" : 1}
let b = {...a}
let c = Object.assign({}, a);
c.b = 3;
b.b = 3;
console.log(a)
// { b: 1 }
```

먼저 스프레드 연산자로 복사를 했고 그 다음에는 c에다가 Object.assign으로 복사를 했습니다. 그 이후에 복사한 객체의 b라는 프로퍼티를 임의의값으로 수정했습니다. 이 때 원본 객체는 바뀌었을까요? 바뀌지 않았습니다.

그렇다면 깊이가 좀 있는 객체는 어떨까요?

```
const a = {
  "b" : 1,
  "c" : {
    "d" : 2
  }
}
let b = {...a}
let c = Object.assign({}, a);
c.c.d = 3;
console.log(a)
//{ b: 1, c: { d: 3 } }
```

깊이 2단계짜리 a를 복사한 c를 수정하니 원본객체 a가 바뀌어버렸습니다. 실무에서 이걸 모르고 지나간다면 생각만해도 끔찍하겠죠? 그렇다면 어떻게 해야 정말 완벽하게 직접복사를 해 원본객체가 수정이 되지 않게 할 수 있을까요?

재귀함수

첫번째는 재귀함수입니다.

```
const copy = o =>{
  let out = {};
  let value, key;
  for(key in o){
    value = o[key];
    out[key] = (typeof value === "object" && value !== null) ? copy(value) : value;
  }
  return out;
}
const a = {
  "b" : 1,
  "c" : {
    "d" : 2
  }
}
let b = copy(a)
b.c.d = 4;
console.log(a)
// { b: 1, c: { d: 2 } }
```

재귀함수 copy가 재귀적으로 작동하면서 해당 value가 object임을 검사함과 동시에 null이 아님을 검사해서 객체가 만약에 또 있다면 또다시 재귀함수가 작동이 되어 새로운 객체에 담아 완전히

복사하는 객체를 이용해 새로운 객체를 만들어 버리는 함수입니다. 이렇게 copy를 하니 복사한 객체 b를 수정해도 a는 절대 변하지 않죠. 참고로 for in에 객체를 넣으면 객체의 키, value를 중심으로 탐색이 가능합니다.

JSON.parse와 JSON.stringify

```
const a = {
  "b" : 1,
  "c" : {
    "d" : 2
  }
}
let b = JSON.parse(JSON.stringify(a))
b.c.d = 4;
console.log(a) // { b: 1, c: { d: 2 } }
```

아예 문자열로 만든 다음에 다시 새로운 객체에 할당하는 방법입니다. 객체 a를 JSON.stringify를 이용해 문자열로 만들어 놓고 다시 JSON.parse를 통해 객체로 만들었습니다. 이 후에 복사한 b를 수정해도 a는 변하지 않습니다.

그렇다면 앞서 설명한 copy(a)와 JSON.parse(JSON.stringify(a)) 중에는 뭘 써야 할까요? 그것은 프로젝트할 때의 객체의 깊이와 객체안에 뭐가 담기는지 파악하고 성능테스트를 진행한 후 결정하는 것이 좋습니다. 실제로 JSmatch를 통해 성능테스트를 한 결과 프로젝트에 쓰이는 객체의 값, 깊이에 따라 JSON.parse가 더 좋을 때가 있고 copy가 더 좋을 때도 있었습니다.

JSON의 기초

지금까지 JSON형태의 객체를 복사하는 방법에 대해 말했는데 혹시 JSON을 모르시는 분들을 위해 JSON의 기초를 준비했습니다. JSON이란 **JavaScript Object Notation**으로써 경량 데이터 교환방식을 뜻합니다. 읽고 쓰기 쉽고 직렬화와 파싱이 쉬운 장점이 있습니다. 객체와 JSON을 헷갈려하시는 분들이 있는데 객체는 자바스크립트의 자료형이고 JSON은 데이터를 표현하는 형식 자체를 뜻합니다. 이 형식은 JSON Object, JSON Array 두가지로 나뉘집니다. 또한 자바스크립트에서는 이 형식을 객체로 변환하는 메서드인 JSON.parse와 문자열로 변환하는 JSON.stringify를 제공합니다.

JSON object

`{"이름" : "큰돌"}` 이러한 구조를 가진 형태를 JSON object라고 부릅니다.

JSON Array

`[{"이름":"큰돌"}, {"이름":"큰돌2"}]`처럼 JSON object가 배열에 담긴 형태를 JSON Array라고 합니다.

JSON의 value 추출하기

JSON 형태의 객체는 object로 먼저 만들어야 key를 통해 value를 참조할 수 있습니다. 예를 들어 `const a = '{"이름": "큰돌"}'`는 문자열입니다. 객체형태가 "라는 따옴표로 감싸져있는 문자열입니다. 이 문자열에서 이름이라는 키를 통해 큰돌이라는 값을 추출해내고 싶다면 먼저 객체로 변환해야 합니다. `JSON.parse(obj)`라는 메소드를 이용합니다. 반대로 문자열을 만들 때는 `JSON.stringify(obj)`라는 메소드를 사용합니다.

```
let a = '{"이름": "큰돌"}'
console.log(typeof a)
console.log(a.이름)
a = JSON.parse(a)
console.log(typeof a)
console.log(a.이름)
/*
    string
    undefined
    object
    큰돌
*/
```

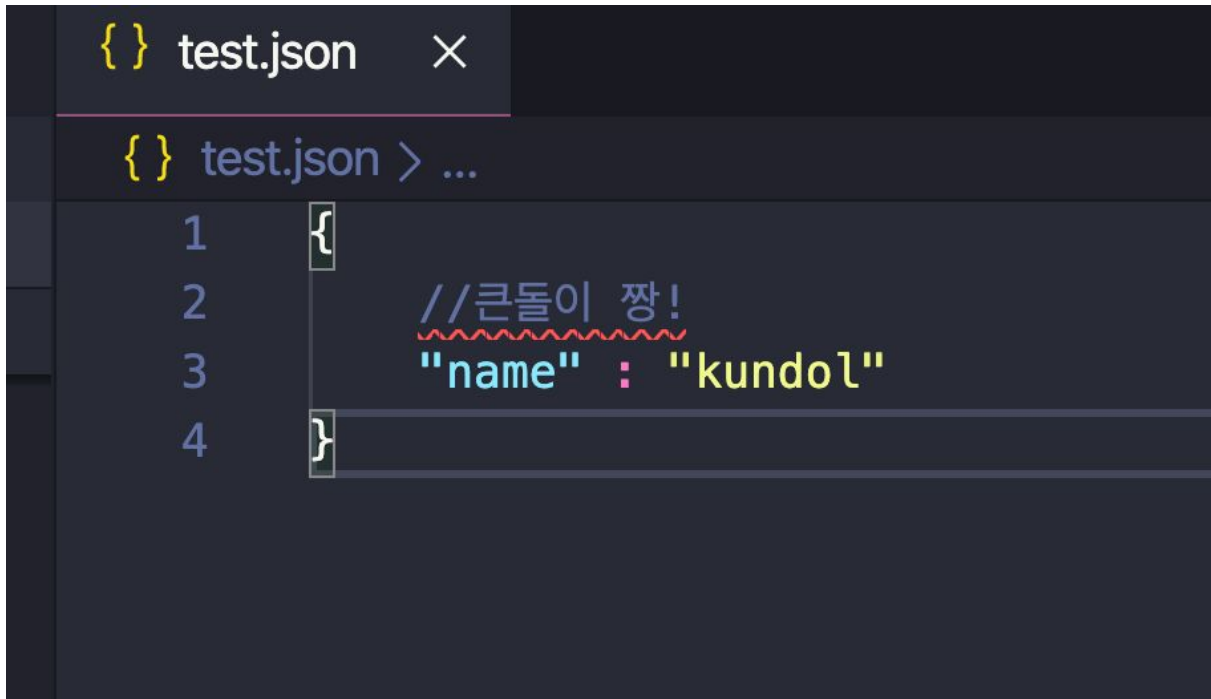
위의 코드를 보면 문자열인 JSON형태의 문자열을 `JSON.parse`을 통해 객체로 만들어서 이름이라는 key를 통해 value를 추출하는 것을 볼 수 있습니다. 이 JSON 형식에는 2가지 규칙이 있습니다.

JSON형식의 규칙

1. JSON에는 주석이 들어갈 수는 있지만 주석이 들어가지 않는 것을 권장합니다.

```
{
    //큰돌이 짱!
    "name" : "kundo1"
}
```


JSON 형식이 맞나 안 맞나를 확인하는 방법은 json파일을 만들어보면 됩니다. test.json이라는 파일을 만들어보겠습니다.



```
{ } test.json ×  
{ } test.json > ...  
1 {  
2 //큰돌이 짱!  
3 "name" : "kundo1"  
4 }
```

<<03-writer-08.png>>

위처럼 test.json을 만들어서 JSON 형태로 만들었지만 주석을 넣어보니 빨간줄이 뜨는 것을 볼 수 있습니다. 맞지 않는 형식이라는 겁니다.

2. JSON의 value로써 허용되는 자료형은 다음과 같습니다. Number, String, Boolean, Array, Object, null 입니다. 이 기본 자료형이 아닌 것들은 직렬화가 되지 않으며 기본값으로 추천하지 않습니다. 참고로 자바스크립트의 기본적인 원시타입은 Boolean, null, undefined, Number, String, Symbol 입니다.

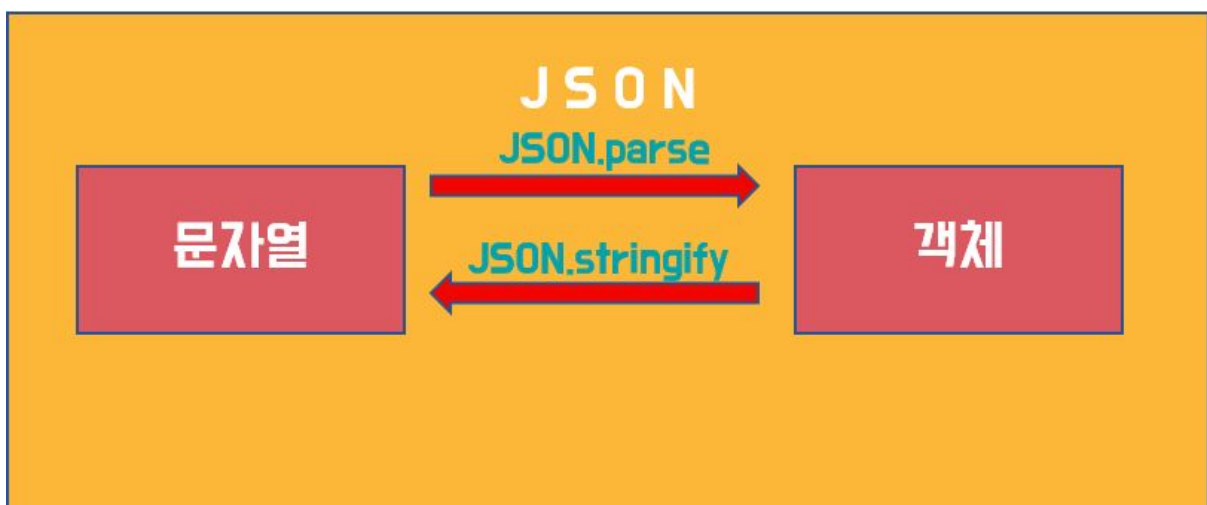
```
{ } test.json ×
{ } test.json
1 {
2   "name" : undefined
3 }
```

<<03-writer-09.png>>

실수를 많이 하시는 부분이 JSON의 기본 초기값을 undefined로 초기화하는 부분입니다. 위에 보시는 것처럼 빨간색을 띄니다. 즉, JSON의 올바른 형식이 아니라는 것입니다. 이렇게 허용되는 자료형을 안했을 경우 직렬화가 되지 않습니다.

JSON 직렬화

직렬화란 외부의 시스템에서도 사용할 수 있도록 byte형태로 데이터를 변환하는 기술을 말하며 JSON 직렬화란 JSON Object를 JSON형태를 가진 문자열로 변환하는 것과 문자열을 JSON Object로 바꾸는 역직렬화를 아우릅니다.



<< 03-writer-02.png >>

JSON직렬화는 `JSON.stringify(obj)`를 통해 문자열로 만들어 구현할 수 있습니다. 다시 말하지만 JSON의 value로 위의 자료형이 아닌 `Number`, `String`, `Boolean`, `Array`, `Object`, `null` 이 아닌 것은 직렬화가 되지 않습니다. 예를 들어 `JSON.stringify({ "이름": undefined })` 이 객체는 직렬화가 되지 않습니다. 따라서 직렬화라는 확장성을 위해 value의 값을 `undefined`로 초기화하는 것을 권장하지 않습니다. `null`이나 `0`, `-999`, JSON의 기본 자료형이며 프로젝트에서 약속된 초기값으로 초기화하는 것을 권장합니다.

참고 파일: MEVN/3장/simple_json.js

```
const obj = {
  "이름" : "큰돌",
  "나이" : 27,
  "나이2" : undefined
}
console.log(obj.이름) //큰돌
const _name = "이름"
console.log(obj[_name]) //큰돌
console.log(obj['이름']) //큰돌
console.log(obj['나이2'])
```

위의 코드를 보면 `obj`라는 JSON형태의 `Object`, JSON `Object`를 정의했습니다. 그리고 `.이름`을 통해 `큰돌`이라는 value를 참조하고 `name`이라는 변수를 선언한 후 `[]`에 담아서 value를 추출해내는 것을 볼 수 있습니다. 이렇게 JSON은 `.`을 통해 해당 value에 접근이 가능합니다. 변수를 통해 value에 접근하는 것은 `[]`를 씁니다. 변수를 쓰지 않는 것은 `.`을 쓰며 변수를 통해 쓰는 것은 `[name]`을 통해 참조하는 것을 권장합니다. 다시 말하자면 `.`을 통해 해당 value에 접근해 참조가 가능합니다. 변수에 문자열을 담아 놓고 그 변수를 통해 value에 접근하는 것은 `[]`를 씁니다. 변수를 쓰지 않는 것은 `.`을 쓰며 변수를 통해 쓰는 것은 `[name]`을 통해 참조하는 것을 권장합니다.

03-4 UX향상을 위한 이미지 레이저로드 구축

예를 들어 10mb짜리 API요청 10개와 100mb짜리 API요청 1개가 있습니다. 무엇이 빠를까요? HTTP에서는 전자가 더 느립니다. 서버요청은 최대한 줄여야 하며 만약 요청이 있더라도 그 용량을 줄이는 것이 중요하며 큰 용량이 여러 개인 경우 지금 사용자에게 필요한 것만 로드해서 효율적으로 줄이는 것 또한 중요합니다.

UX향상 : 레이저로드와 이미지크기 축소

흔히들 UX를 향상시키자고 합니다. UX란 사용자 경험을 말하며 이를 향상시킨다는 것은 홈페이지를 눌렀을 때 빠르게 뜨는 것이 포함이 됩니다. 여러가지 요소들이 있지만 이것은 제일 중요한 요소 중 하나입니다. 사용자가 처음 페이지를 켰을 때 필요한 것은 그렇게 많지 않습니다. 지금 사용자가 보고 있는 영역의 이미지만이 필요할 뿐이고 그 아래의 이미지는 필요하지 않습니다. 그렇기 때문에

사용자가 보고 있는 영역의 이미지만 로드를 하면 될 뿐이죠. 이렇게 추후 나중에 게으르게 이미지를 로드하는 것을 레이저로드, 게으른 로드 라고 합니다. 그렇다면 레이저로드는 어떻게 구현할까요?

1. express를 이용해 서버 구축해보기

먼저 심플한 서버를 구축합니다. node.js로 서버를 구축하는 여러가지 방법이 있지만 그 중 express라는 모듈을 이용해서 서버를 구축해보겠습니다.

```
MEVN/3장/server.js
const express = require('express');
const path = require('path')
const app = express();
const _path = path.join(__dirname, './dist')
app.use('/', express.static(_path))
app.listen(12010, ()=> {
  console.log('lazy 이미지서버 : 12010시작 http://127.0.0.1:12010');
});
```

dist라는 폴더에 있는 정적자원들을 사용자에게 제공하는 서버를 만들었습니다. express.static에 정적자원을 제공할 경로, path를 집어 넣으면 그 경로에 있는 자원들을 사용자들에게 제공하게 됩니다.

▶ 질문 Q. 여기서 dist 폴더가 아닌 public이라는 폴더를 제공한다면 어떤 코드를 수정해야 할까요?

▶ 답 Q. 정답은 express.static에 들어가는 인자에 path.join(__dirname, './public')이 들어가야 합니다.

2. 이미지를 보여주는 index.html과 img_lazy.js파일 준비하기

```
MEVN/3장/dist/index.html
<!DOCTYPE html>
<html lang="ko">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <script src="./img_lazy.js"></script>

  <style>
  ul{
    list-style-type: none;
  }
  .cardWrap > li {
    height: 500px;
    margin-bottom: 100px;
    text-align: center;
  }
  .cardWrap img{
    height: 100%;
```

```

        box-shadow: 0 4px 8px 0 rgba(0,0,0,0.2);
    }
</style>
</head>
<body>
    <ul class="cardWrap">
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
        <li></li>
    </ul>

</body>
</html>

```

dist폴더내에 1. index.html을 만들고 2. img_lazy.js라는 파일을 만들어 둡니다. 트와이스 모모를 좋아하기 때문에 모모이미지 카드들로 만들어진 html파일입니다. 모모 이미지들을 dist/img폴더내에 담아 둡니다. <https://github.com/wngghdcife/MEVN/tree/master/3>장을 통해 모모 이미지들을 쉽게 다운 받으실 수 있습니다. 다른 이미지로 하실분들은 이미지를 다운받아서 img폴더에 넣고 html파일의 img 태그를 수정하시면 됩니다. 자 그렇다면 브라우저로 들어가 <http://127.0.0.1:12010/> 을 입력해서 내 로컬서버를 조회해보도록 하겠습니다.



<< 03-writer-03.PNG >>

모모가 보입니다. 하지만 이렇게 15장의 고화질 이미지를 한꺼번에 로드하는 것은 부하가 큼니다. 이제 이미지레이저로드를 구현해봅시다.

3. 이미지 레이저 로드를 하는 imgLazy.js를 채워보기

우리가 할 것은 dist 폴더에 만들어 놓은 imgLazy.js를 채워놓는 것 뿐입니다.

```
MEVN/3장/dist/imgLazy.js
document.addEventListener("DOMContentLoaded", function() { // --- (1)
  let lazyImages = [].slice.call(document.querySelectorAll("img.lazy")); //
```

```

--- (2)
let active = false; // --- (3)
const lazyLoad = function() {
  if (active === false) {
    active = true;
    setTimeout(()=> { // --- (4)
      lazyImages = lazyImages.map((lazyImage) => {
        if (lazyImage.getBoundingClientRect().top <= window.innerHeight &&
window.getComputedStyle(lazyImage).display !== "none") { // --- (5)
          lazyImage.src = lazyImage.dataset.src; // --- (6)
          lazyImage.classList.remove("lazy");
          return null;
        }else return lazyImage;
      }).filter(image => image);
      if (!lazyImages.length) { // --- (7)
        document.removeEventListener("scroll", lazyLoad);
      }else active = false;
    }, 200);
  }
};
document.addEventListener("scroll", lazyLoad);
});

```

위의 코드를 차근차근 해석해보겠습니다.

- 1) DOMContentLoaded : DOM이 모두 로딩되었을 때의 이벤트입니다. 참고로 DOMContentLoaded 와 비교되는 load이벤트는 js, css, img 등 정적자원들 모두와 DOM이 모두 로딩되었을 때를 나타내며 window.onload = e => {}로 쓰입니다.
- 2) 처음의 lazy라는 클래스가 있는 이미지들을 이용해 배열을 만들고 그걸 이용해 로직을 짜야 합니다. 그냥 document.querySelectorAll("img.lazy")를 하면 유사배열인 NodeList로 배열의 메소드인 filter나 forEach를 쓰지 못하기 때문에 배열로 만들어 주어야 하므로 빈배열에 slice로 얕은복사(shallow copy)를 하고 call을 통해 인자로 document.querySelectorAll("img.lazy") 넣으면 유사배열인 NodeList를 진짜배열로 만들어 filter, forEach 등의 Array.prototype 메서드를 쓸 수 있습니다.
- 3) 이 active라는 플래그로 저화질의 이미지를 버리고 고화질의 이미지를 불러올지 말지를 정하게 됩니다. 이 후 setTimeout을 통해 반복적으로 2번에서 만들어 놓은 저화질리스트들을 업데이트하며 확인하는 것을 반복합니다.
- 4) 스크롤이벤트가 발생했을 때 200밀리세컨드 이후 저화질의 이미지리스트들을 확인합니다.
- 5) 저화질의 이미지리스트들을 하나씩 확인하며 window.innerHeight와 이미지의 lazyImage.getBoundingClientRect().top을 비교해서 내가 지금 보고 있는 화면에 이미지가 들어왔는지를 확인합니다.

- 6) src를 dataset에 설정해 놓은 원래의 고화질의 이미지를 수정해서 불러옵니다. 그리고 lazy라는 클래스를 지우고 object를 null로 바꿔서 리턴합니다. 그 후 filter를 통해 null로 정의된 요소는 배열에서 없어지게 만듭니다.
- 7) 이 후 이렇게 배열의 요소들을 삭제시키고 다 없어졌다면 이 이벤트를 스크롤 이벤트에서 없앱니다.

▶ 참고: slice

slice는 얇은 복사로 빈배열이 아닌 객체가 들어가 있는 배열을 복사하면 원본 객체가 수정이 되면 복사한 객체도 수정이 됩니다. 빈배열을 복사했기 때문에 얇은 복사라도 상관이 없습니다. 단지 배열의 prototype 메소드인 filter 등을 사용할 수 있기 때문에 저렇게 구현했습니다. 또한 객체가 아닌 정수 등이 들어가 있는 배열의 경우에는 복사본을 수정해도 원본에는 변화가 없습니다. 하지만 객체를 다룬다면 앞서 설명한 제대로된 복사로 해야 합니다.

4. 로컬 서버확인해보기

브라우저로 들어가 <http://127.0.0.1:12010/> 을 입력해서 내 로컬서버를 조회해보도록 하겠습니다. 스크롤 할 때 흐릿한 이미지가 분명한 트와이스 이미지로 바뀌면서 구현이 되고, 처음에 15장의 이미지를 동시에 받지 않고 볼 때마다 받는 레이저로드가 완성이 되었습니다. 이 예제는 저화질의 blur이미지 한장이 기본적으로 받아지지만, 실제로는 각각 해당하는 이미지의 저화질이미지를 만들어서 구현하는 것이 일반적이며 15장의 이미지를 모두 로드한다고 해도 저화질이기때문에 데이터소모가 심하지 않습니다.



<< 03-writer-04.PNG >>

그러나 스크롤 한 후에는 실제 모모 이미지가 뜹니다.



<< 03-writer-05.PNG >>

이미지 레이저로드를 도와주는 라이브러리로 `lozad.js` / `yall.js` 가 있으며 모두 똑같은 원리입니다. 사용자가 보지 않는다면 `placeholder` 이미지 또는 `blur` 이미지를 셋팅하고 사용자가 본다면 실제로 보는 이미지를 대체하는 것입니다. 이외에도 `template`를 사용하는 방법이 있지만 권장하지 않습니다. 또한, 애초에 이미지의 용량을 줄이는 것도 중요합니다. 이미지 용량을 줄이는 사이트로는 <https://compressjpeg.com/> 와 <https://tinypng.com/> 를 추천합니다.

03-5 함수형 프로그래밍의 특징

함수형프로그래밍은 네이버페이, 인프런을 만들었을 때 쓰이기도 한 유명한 프로그래밍 패러다임 중 하나로 작은 순수함수들의 집합으로 최소의 부수효과를 누리는 패러다임입니다. 좀 더 자세히 말하자면 함수들을 작게 쪼개서 블록을 쌓듯이 로직을 구현하고 다형성, 어떠한 인자든 가능하게 하는 것을 높이고 고차함수를 활용하여 재사용성을 높이고 참조투명성, 함수의 출력은 오로지 그 함수에 입력된 값에만 의존성을 가진다는 의미를 통해 부수효과가 줄어들어 유지보수성을 증대시키는 함수형 프로그래밍에 대해서 알아보겠습니다. 이를 통해 단순하고 간결한 흐름 중심의 모델이 되고 서술부와 평가부가 분리되어 지연평가가 가능하며 지금부터 이 함수형프로그래밍의 여러가지 특징들에 대해 살펴보겠습니다. 이 있습니다. 그 특징들을 살펴보겠습니다.

1. 순수 함수 참조 투명성

순수함수란 동일한 입력을 받아서 동일한 출력을 하는 것을 말합니다. 참조투명성의 성질을 가지고 있습니다.

```
const PURE = (a, b)=>{
  return a + b;
}
const not_PURE = (a)=>{
  return a += 10;
}
```

위의 PURE는 순수함수고 아래 not_PURE는 순수함수가 아닙니다. 외부의 인자를 변화시키지 않고 값이 일정해야 합니다. 10을 더하는 함수지만 외부의 a의 값을 증가시켜서 부수효과가 생기게 했으므로 순수함수가 아닙니다. 예를 들어 Math.max는 순수함수입니다. 무조건 들어온 인자들을 토대로 최대값을 반환하기 때문입니다. 즉, 외부의 값들은 상관하지 않고 그저 들어오는 인자값만을 이용해서 결과를 도출해내는 것이 순수함수입니다.

전역변수를 참조할 때의 순수함수

하지만 프로젝트를 하다 보면 전역변수를 쓸 때가 있습니다. 그러나 함수형프로그래밍을 한다면 1) 최대한 지역변수만을 써서 구현해야 합니다. 그리고 2) 인수로 넘기는 값의 본체조차 수정되면 안됩니다. 3) 인자로 넘기는 값이 객체라면 원본객체에 변이를 일으키지 않도록 주의해야 합니다. deep copy를 한 후 변이를 하던가 등의 방법을 써야 합니다.

```
var c = 0;
//받은 인자자체가 변함
function f(){
  return c++;
}
//인자를 변화시키지 않고 +1을 할 수 있다.
```

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

이렇게 인자를 변화시키지 않아야 합니다.

2. 고차 함수

함수가 다른 함수를 인자로 받아서 어떠한 로직을 생성하는 함수를 말합니다. 참고로 이 고차함수는 함수를 값 자체로 다른 함수에 넘길 수 있는 일급 객체여야만 가능하며 여기서 사용하는 자바스크립트라는 언어의 함수는 1급 객체이기 때문에 가능합니다. 1급 객체는 다음과 같은 성질을 만족하는 함수를 뜻합니다.

- 변수나 메서드에 함수를 할당할 수 있습니다.
- 함수안에 함수를 인자로 받을 수 있습니다.
- 함수가 함수를 리턴할 수 있습니다.

map

map은 함수 a를 받는 고차함수입니다. 참고로 이와 같이 map이나 filter는 어떠한 값 [a]를 받아 새로운 []를 리턴하는 것을 모나드 함수라고 합니다. 이외에도 모나드 함수는 불완전한 값을 래핑하는 함수라는 특징을 가지고 있습니다.

```
const a = (a) => {  
  return a + 10;  
}  
const b = [1, 2, 3].map(a)  
console.log(b)
```

위의 코드를 보면 [1, 2, 3]이라는 배열을 map이라는 함수를 통해 어떠한 함수, a를 통해 배열의 값을 바꿔 새로운 배열을 리턴하는 것을 볼 수 있습니다.

함수를 실행하여 리턴하는 _call

```
const _call = (a, b) => a() + b()
```

위의 _call 함수처럼 인자를 받아 함수의 실행시점을 자유자재로 놓을 수 있습니다.

함수를 리턴하는 함수

```
//ES6  
const a = val => () => val;  
//ES5  
function a(val) {  
  return function() {  
    return val;  
  }  
}
```

```

}

var a_lazy = a(12010);
a_lazy();
// 12010

```

위의 코드는 a라는 함수가 함수를 리턴하고 다시 a_lazy에다가 함수를 할당하는 코드입니다. 이렇게 함수를 안에서 다시 리턴하면 실행시점을 지연시킬 수 있습니다.

3. 커링

어떤 함수의 예상되는 여러개의 인자 보다 적은 수의 인자를 받아서 그 함수는 그 인자를 받은 채로 함수를 리턴합니다. 인자가 완전히 채워지지 않은 함수는 실행되지 않고 넘겨지게 되다가 모든 인자가 채워지면 그 때서야 실행되는 기법입니다. 다시 말하자면 함수 하나가 n개의 인자를 받는 과정을 n개의 함수로 쪼개서 한 개의 함수는 한 개의 인자를 받는 기법을 말합니다. 함수들이 각각의 인자를 받고 그 인자가 다 채워지면 그때서야 함수들의 합성이 실행하는 것을 말합니다. 이렇게 지연되는 실행이 가능한 이유는 내부함수가 외부함수의 컨텍스트가 소멸이 되도 외부함수의 실행환경을 참조할 수 있는 클로저 때문이며 내부함수는 파라미터가 완성될 때까지 실행시점을 미룰 수 있습니다.

또한 커링을 이용해 나중에 설명할 무인수프로그래밍도 가능해집니다.

다음은 함수형 프로그래밍을 도와주는 Ramda.js 라이브러리를 이용한 예입니다. 자바스크립트 파일을 만들고 npm init, npm install --save ramda를 해서 패키지 설치 후 node [파일명] 으로 코드를 실행하면 됩니다. R.add의 경우 처음에 인자 1을 넣고 나중에 2를 넣어 인자가 완성이 되었을 때 실행을 할 수 있게 합니다.

```

// ramda를 이용한 예
const R = require('ramda')
const a = R.add(1)
const b = a(2)
console.log(b) //3

```

```

// ramda를 이용한 예
const addFourNumbers = (a, b, c, d) => a + b + c + d;

const curriedAddFourNumbers = R.curry(addFourNumbers);
const f = curriedAddFourNumbers(1, 2);
const g = f(3);
g(4); //=> 10

```

위 코드처럼 어떠한 함수를 만들고 R.curry에 담는다면 그 함수를 토대로 만들어지는 커링함수가 완성됩니다.

커스텀커링

```
// ES6 커링
const k_curry = fn => a => b => fn(a, b);
const f = k_curry((a, b) => a + b);
const f_c = f(1);
const ret = f_c(2);
console.log(ret)
```

```
// ES5 커링
var k_curry = function(fn){
  return function(a){
    return function(b){
      return fn(a, b);
    }
  }
}
var f = k_curry(function(a, b){
  return a + b;
});
var f_c = f(1);
var ret = f_c(2);
console.log(ret)
```

그렇다면 커링을 한번 만들어보겠습니다. ES6, ES5버전으로 각각 만들었습니다. k_curry라는 2개의 인자를 받아 두개의 함수로 쪼개어 인자를 받는 커리함수를 만들었고 그 함수를 이용해서 익명함수인 (a, b) => a + b를 쪼갤 수 있게 되었습니다. 이제 f는 커링이 가능한 더하기함수가 되었습니다.

fxjs내 curry 함수

```
export const curry = f => (a, ..._) => _.length ? f(a, ..._) : (..._) => f(a, ..._);
```

참고로 함수형프로그래밍 라이브러리인 fx.js의 커리는 위의 코드처럼 구현되어있습니다. ..._로 남은 인자를 받아서 만약에 남은 인자가 없다면 함수를 실행합니다. 즉, 인자가 완성 되었다면 **실행**하게 구성되어있습니다.

4. 함수 합성

작은 순수함수들로 만들고 하나의 인자만 받도록 커링을 했다면 이제 함수합성을 할 차례입니다. 합성이라는 것은 함수를 합치는 것을 말합니다. 아래의 코드를 보면 f_1과 f_2가 합쳐져 있는 것을 볼 수 있습니다.

```
//pipe나 compose를 사용하기 전 - 함수합성이 복잡합니다.
const f_1 = map(a => a + 1);
const f_2 = filter(a => a % 2);
```

```
const ret = f_2(f_1([1, 2, 3, 4]))
console.log(ret) // 3, 5
```

하지만 읽기가 너무 불편합니다. 함수합성을 도와주는 메서드인 `compose`와 `pipe`를 보겠습니다.

ramda를 이용한 함수합성 - `compose`

아까 설치한 `ramda` 함수형 라이브러리를 이용한 예를 보겠습니다.

```
// Ramda를 이용한 함수 합성 - compose
const R = require('ramda')
R.compose(Math.abs, R.add(1), R.multiply(2))(-4) // => 7
```

ramda를 이용한 함수합성 - `pipe`

`R.compose`는 위의 코드처럼 오른쪽에서 왼쪽으로 로직이 흘러갑니다. 처음 `-4`에 `2`를 곱하고 `-8`, 하나를 더하고 그 값의 절대값을 구하면 `7`이 됩니다. 위의 `Math.abs`처럼 `ramda`가 아닌 함수도 합성에 쓰일 수 있습니다.

```
// Ramda를 이용한 함수 합성 - pipe
const R = require('ramda')
const f = R.pipe(R.negate, R.inc);
console.log(f(3)) // -2
```

`pipe` 또한 합성이지만 왼쪽에서 오른쪽으로 흘러가는 점이 다릅니다. `compose`보다는 `pipe`를 쓰는 것을 추천드립니다. 왼쪽에서 오른쪽으로 보는 것이 보기 쉽기 때문입니다. 처음 받아온 `3`을 `negate`, `-1`을 곱하고 `inc`를 통해 `1`을 증가시켜 `-2`를 반환합니다.

fx.js를 이용한 함수합성

이제 `fx.js`라는 함수형 라이브러리를 이용한 `pipe` 합성을 보겠습니다. 함수형 라이브러리는 크게 `lodash`, `ramda`, `fxjs`가 있는데 저는 그 중 `fx.js`를 추천합니다. 먼저 `npm install --save fxjs`를 통해 `fx.js` 모듈을 설치합니다. `ramda`와 비슷합니다. `pipe`로 로직을 구현했습니다.

```
const {map, filter, pipe} = require('fxjs')
const f_pipe = pipe(
  map(a => a + 1),
  filter(a => a % 2)
)
const _ret = f_pipe([1, 2, 3, 4])
console.log(_ret) // 3, 5
```

5. 파괴적인 함수를 쓰지 말기. – slice VS splice

이 패러다임에서 중요한 점은 최대한 원본 객체나 인자는 그대로 있어야 한다는 점입니다. splice는 배열을 파괴합니다. 하지만 slice는 파괴를 하지 않고 1을 제외한 나머지 배열 요소들을 추출해낼 수 있습니다. 되도록 함수로 넘어오는 인자(payload) 또는 원본 요소를 파괴하지 않는 함수를 써야 합니다.

```
const a = [1, 2, 3, 4, 5]
let b = a.slice(1)
console.log(a, b)
b = a.splice(1)
console.log(a, b)
/*
[ 1, 2, 3, 4, 5 ] [ 2, 3, 4, 5 ]
[ 1 ] [ 2, 3, 4, 5 ]
*/
```

slice를 써버렸더니 a라는 배열의 원본 배열이 살아있고 splice를 썼더니 a라는 원본배열이 파괴된 것을 볼 수 있습니다. 이러한 함수들을 비순수함수라고 합니다. 비순수함수는 reverse, sort, fill, pop, push, shift, unshift, splice가 있으며 되도록이면 사용하지 않을 것을 권장합니다.

6. Point – free, 무인수 프로그래밍

아까 말했던 Point-free, 무인수 프로그래밍입니다. 인자에 신경쓰지 않고 자유롭게 구현하는 것을 뜻하는 이 무인수프로그래밍은 실행되는 함수 안에 인자에 관한, 인자에 의한 값들이 고정되어 들어가는 것이 아닌 함수합성을 설정할 때 인자를 유동적으로 설정할 수 있다는 것입니다.

```
const {map, filter, pipe, values, go} = require('fxjs')
const L = require('fxjs')
const 친구들 = [
  {
    "name" : "연제호",
    "do" : "군인"
  }, {
    "name" : "박종선",
    "do" : "회장"
  }, {
    "name" : "윤성용",
    "do" : "개발자"
  }, {
    "name" : "박동영",
    "do" : "개발자"
  }
]

const t = 친구들.map(e => e.do === "개발자" ? e : null).filter(e => e)

const prop = key => obj => obj[key]
```



```

const propEqual = value => key => obj => prop(key)(obj) === value
console.log(t)
const t2 = go(
  친구들,
  L.filter(propEqual('개발자')('do')),
  L.takeAll
)
console.log(t2)
/*
[ { name: '윤성웅', do: '개발자' }, { name: '박동영', do: '개발자' } ]
[ { name: '윤성웅', do: '개발자' }, { name: '박동영', do: '개발자' } ]
*/

```

위의 코드를 보면서 설명하겠습니다. 친구들 중 개발자인 요소들을 뽑는다면 어떻게 해야 할까요?

1. 기본적으로 map을 통해 관련 함수를 집어 넣습니다.
2. 그 함수는 바로 다음과 같습니다. do라는 key의 해당하는 value 가 “개발자”라면 원래의 요소를 반환하고 아니라면 null을 반환하는 함수입니다.
3. 이 후에 filter를 통해 null이라면 boolean값으로 false가 되므로 null값에 해당하는 요소는 filter로 걸러지게 됩니다. 이렇게 하면 다음과 같이 do가 개발자인 요소를 꼬집어낼 수 있습니다.

하지만 이와 같은 함수합성은 문제점이 있습니다. 이러한 로직이 만약 반복된다면 우리는 t라는 함수를 계속해서 새로 만들어야 합니다. e.do === “음악가”, e.나이 === 27 등 꾸준하고 멍청하게 만들어야 하죠. 이를 무인수 프로그래밍으로 해결할 수 있습니다. prop와 propEqual이라는 함수를 통해서 key와 value에 해당하는 값을 통해 필요한 요소를 추출하는 함수를 만듭니다. 그 후 go, pipe의 즉시실행함수 버전을 통해 함수 합성을 합니다. 친구들, L.filter를 통해 그 안에 미리 만들어 놓은 propEqual를 집어 넣어서 해당 로직을 짤 수 있습니다. 이렇게 만들어 놓으면 추후 똑같은 로직이 발생되어도 함수를 설정할 때의 인자만을 바꾸면 되는 것입니다. 즉, 새로운 함수를 만들 필요가 없이 인자값만을 수정하면 되게 유연하게 설정하는 것입니다.

7. 모나드

모나드는 컨텍스트 레벨을 값 레벨로 끌어들여 추상화를 한 후 이를 통해 로직 구성의 유연성에 도움을 주고 함수합성을 안전하게 하는데 도움을 줄 수 있습니다. 컨텍스트란 어떤 루틴의 실행에 영향을 줄 수 있는 레지스터, 콜스택, 객체의 현재 상황 등에 해당하는 상태를 말하는데 이를 하나의 값으로 추상화시키는 것을 모나드라고 합니다. 또한 개념적으로 함수형프로그래밍에서는 반드시 참조투명성, 입력과 출력이 동일하게 1 : 1 매칭이 되어야 하므로 모나드를 씁니다. 예를 들어 특정값을 예상했는데 애러가 나타난다면 1 : 다수의 매칭이 되므로 이 특성을 유지하기 위해 안전하게 모나드를 쓰는 것입니다. 모나드는 함수합성에 많이 쓰이는데 예를 들어 f라는 함수가 있고 g라는 함수가 있고 f(g()) 이런식으로 합성을 한다고 했을 때 예상치 못한 값을 집어넣게 되면 오류가 발생하거나 예상치 못한 결과가 나올 수 있습니다. 이를 방지하고 안전하게 컨테이너 박스안에서 연산을 실행할 수 있게 도와주는 개념이 바로 모나드입니다.

모나드는 아이덴티티 모나드, 퓨처 모나드 등이 있으며 이 중 여기서 퓨처모나드인 Promise를 중점으로 알아봅니다. 이 퓨처모나드로 null, undefined 애러처리, I/O 관련 불완전한 상태로 일어나는 로직들을 안전하게 처리할 수 있습니다.

퓨처모나드, Promise

퓨처모나드인 Promise 모나드는 래핑하기 위해 쓰입니다. 비동기를 처리하거나 null값이 발생할 수 있는 로직에 대해서 어떠한 값의 결과가 불확실하게 2가지형태 이상으로 나타날 수 있습니다. 이럴 때 컨테이너로 래핑하여 안전하게 연산을 하는 것을 말합니다. 또한 함수형프로그래밍의 특징을 지키기 위해서도 존재합니다. Promise는 퓨처모나드이며 미래의 값을 담을 수 있다는 뜻을 가지고 있습니다. 그렇다면, 미래의 값은 뭘까요? 바로 I/O모델적인 상황입니다. 데이터베이스에서 값을 가져오거나 네트워크를 통해서 어떠한 API서버로부터 값을 가져온다고 했을 때 그 값은 미래의 값이 됩니다. 지금상황으로써는 그 값이 어떠한 결과인지 모르고 단지 우리가 짜놓은 스키마 등에 따른 예상만 가능하기 때문이죠. 이러한 미래의 값을 기다리는 비동기적 상황에서 성공과 실패라는 두가지의 값으로 놓고 미래를 다루는 하나의 컨테이너 박스가 Promise입니다. 뭔가 멋져 보이죠? 2장에서 Promise를 통해 불완전한 비동기의 결과값을 콜백, 어떤 함수가 어떠한 시점에서 다시 호출되는 것이 또는 컨텍스트가 아닌 하나의 값으로 로직을 수행하는 것을 볼 수 있었습니다. 모나드라 불리는 것은 모나드의 조건을 충족시키기 때문에 모나드로 불리는 것이며 Promise는 아래의 모나드 조건을 모두 충족시키기 때문에 모나드입니다.

1. 타입을 인자로 받아 모나드화 된 타입을 반환할 수 있어야 합니다. Promise는 순수한 값을 Promise로 래핑해서 반환할 수 있습니다. 프로미스로 감싸서 리턴하면 Promise<pending>이라고 나오게 되는 것처럼 말이죠.
2. unit함수가 있어야 합니다. 모나드화 된 타입에서 순수한 값으로 변환할 수 있어야 합니다. Promise는 resolve 메서드를 통해 순수한 값을 꼬집어 낼 수 있습니다.
3. bind함수가 있어야 합니다. 모나드가 중복되어 감싸진 상태라도 꺼낼 수 있는 방법이 있어야 합니다. Promise는 중복되더라도 .then메서드를 통해 순수한 값을 꼬집어 낼 수 있습니다. 이를 flatten되었다라고도 합니다.

안전한 함수합성, Promise를 이용한 예(1)

```
const g = JSON.parse;
const f = k => k.temp;
const fg = x => new Promise((resolve, reject)=> resolve(x)).then(g).then(f)
const log = x => console.log(x)

fg('{"temp":36.5}').catch(_ => 'JSON PARSE is not working').then(log)
```

함수 g와 함수 f를 준비합니다. {"temp":36.5}라는 문자열을 g를 이용해서 JSON Object로 변환합니다. 이 때 Promise를 이용해서 오류가 있다면 then(f)가 아닌 catch로 넘어가서 애러를 처리하게 합니다.

안전한 함수합성, Promise를 이용한 예(2)

```
const log = console.log
const users = [{name : '큰돌'}, {name : '제호'}, {name : '우람'}, {name : '다빈'}]
const getUserByName = (name) => users.find(u => u.name === name) ||
Promise.reject("객체에 없습니다.")
const g = getUserByName;
const f = ({name}) => `${name}이가 춤을 춥니다.
Const fg = x => new Promise((resolve, reject)=> resolve(x)).then(g).then(f)
fg("큰돌1").catch(_ => _).then(log)
// 객체에 없습니다.
fg("큰돌").catch(_ => _).then(log)
// 큰돌이가 춤을 춥니다.
/*
큰돌이가 춤을 춥니다.
객체에 없습니다.
*/
```

아까 보았던 코드와 같은 구조이지만 다른 예제입니다. getUserByName의 경우 원하지 않는 결과값이 나올 경우 Promise.reject를 통해 다음 로직은 실행하지 않은 바로 catch로 rejected상태의 Promise를 넘겨서 종료시켜버립니다.

8. 제너레이터와 이터레이터

2장에서 설명을 했습니다. 지연평가와 코루틴을 주의깊게 봐주세요.

9. 클로저

클로저는 독특한 함수체제를 멋지게 활용할 수 있습니다.

- 프라이빗 변수를 모방
- 가상의 블록 스코프 변수를 생성

클로저란 외부함수의 실행컨텍스트가 소멸해도 [[scope]]프로퍼티가 가리키는 외부함수의 실행환경은 소멸하지 않고 참조할 수 있는 것을 말합니다. 스코프체인이 형성될 때 [[scope]]로 참조할 수 있는데 이를 이용해 참조할 수 있는 것을 말합니다. 잘 이해가 되지 않는다면 이 글 아래의 EC 형성부분을 보주시면 됩니다. 클로저의 예는 다음과 같습니다.

클로저의 예(1)

```
const test = (function(){
  let count = 0;
  return {
    increase(){
      count++;
    },
    decrease(){
      count--;
    },
    getValue(){
      console.log(count);
    }
  }
})();
test.increase();
test.increase();
test.increase();
test.getValue();
```

자바스크립트에는 private 변수는 존재하지 않습니다. 하지만 즉시실행함수와 클로저를 이용하면 private 변수를 구현할 수 있습니다. 위의 코드는 count라는 private 변수를 만듭니다. test라는 함수는 getValue를 통해서만 count가 어떠한 값인지 확인 할 수 있습니다.

클로저의 예(2)

또한 counter라는 변수가 어떠한 행위에 이어서 실행될 수 있게 만듭니다. counter라는 변수는 내부함수에서 참조할 수 있는 클로저로 남아있기에 계속해서 더하기가 가능합니다.

```
const add = (function () {
  let counter = 0;
  return function () {
    counter += 1;
    return counter;
  }
})();
console.log([add(), add(), add()]) //[1, 2, 3]
```

D3js에서의 클로저

실제로 이 클로저는 유명한 라이브러리인 D3.js에서도 쓰입니다. offset이라는 private 변수를 만드는 모습입니다.

```
function center(scale) {
  var offset = Math.max(0, scale.bandwidth() - 1) / 2; // Adjust for 0.5px offset.
```

```

if (scale.round()) offset = Math.round(offset);
return function(d) {
    return +scale(d) + offset;
};
}

```

자바스크립트 실행컨텍스트, EC

자바스크립트(JavaScript)는 코드 실행에 필요한 정보들을 물리적인 객체인 EC, 실행컨텍스트(Execution Context)를 통해서 관리합니다. 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 1. 스코프체인, 2. Variable Object, 3. this로 구성이 되며 1. 스코프체인생성과 초기화 2. Variable Object 형성 3. this value 결정이 순차적으로 결정됩니다.

1. 스코프체인생성과 초기화

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. 이 때 AO나 GO모두 아직 값은 정해지지 않았고 빈 껍데기인 상태입니다. 좀 더 자세히 말하자면, 자신의 실행환경, 자신을 포함하는 외부실행환경, 전역객체를 순차적으로 가리키며 리스트를 만들게 됩니다. 이렇게 변수를 찾는 과정을 스코프체이닝이라고 합니다. 이 스코프체인이 형성될 때 [[scope]]로 참조할 수 있는데 이를 이용해 참조할 수 있는 것을 클로저라 합니다.

2. Variable Object 형성

변수들에 대해 2가지, AO, GO의 프로퍼티들이 채워지게 됩니다. 여기서 AO란 Activation Object를 가리키며 함수선언(표현식은제외), Arguments, 변수를 포함하고 GO는 Global Object, 전역변수들을 포함하게 됩니다. 이 때 함수인 경우 매개변수가 property로 그 값인 argument가 값으로 설정됩니다. 대상 코드 내의 함수 선언(함수 표현식 제외)을 대상으로 함수명이 Variable Object의 프로퍼티로, 생성된 함수 객체가 값으로 설정됩니다. 그리고 이 함수는 실행할 수 있습니다.(함수 호이스팅) 대상 코드 내의 var로 이루어진 변수 선언을 대상으로 변수명이 Variable Object의 프로퍼티로, undefined가 값으로 초기화됩니다. (변수 호이스팅)

3. this 의 결정

this는 함수 호출 패턴에 따라 달라지거나 화살표함수의 경우 lexical scope를 참조합니다. 화살표함수의 this와 그 외의 this로 나뉘집니다.

화살표구문에서의 this

화살표구문에서는 lexical this, 바인딩할 객체가 정적으로 정해지며 함수가 호출될 때 동적으로 변환되지 않고 정해집니다. 참고로 아래의 있는 예제들은 모두 크롬 브라우저를 열고 f12를 눌러서 chrome devTools를 연다음 console창에 이 코드를 친 후 [Enter]을 치면 테스트를 할 수 있습니다.

```
// 간단한 this
const a = () =>{
  console.log(this);
}
a() // window
function b(){
  console.log(this);
}
b() //window
```

a라는 함수가 실행하는데 this는 무엇을 가리킬까요? 이것은 화살표함수던 화살표함수가 아니던 똑같습니다. 바로 window 전역객체를 가리키게 됩니다. 왜냐하면 화살표함수일 때 this가 정해져있지 않고 위로 올라가며 스코프체이닝을 합니다. 그 결과가 window가 되고 화살표함수가 아닌 경우에는 호출되었을 때의 this가 window객체 내에서 실행되기 때문에 this는 모두 window를 가리키게 됩니다.

setTimeout

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

  setTimeout(() => {
    this.age++;
    console.log(this, this.age)
  }, 1000);
}
var p = new Person(); // Person {age: 1} 1
```

화살표 함수를 쓰면 정적으로 선언이 될 때 상위스코프인 Person이라는 클래스를 가리키게 됩니다. 그래서 이 this.age는 증가하게 됩니다. 그러나 화살표함수를 쓰지 않으면 함수호출패턴에 의해서 this를 가리키게 됩니다.

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

  setTimeout(function(){
    this.age++;
    console.log(this, this.age) // Window NaN
  }, 1000);
}
var p = new Person();
```

올바르게 this.age가 증가할 것 같지만 NaN이 나오게 됩니다. 화살표함수를 쓰지 않으니 이 this는 함수호출패턴을 통해 가리키게 되고 즉, 함수호출로 인해 setTimeout이라는 함수는 window객체에서 나온 메소드이므로 this는 window객체를 가리키게 되며 window에 age라는 값은 없기 때문에 Not a Number, NaN이 나오게 됩니다. 그렇다면 함수호출패턴이란 무엇이며 화살표함수를 쓰지 않는다면 이 패턴을 따른다는데 어떻게 이 패턴안에서 this는 어떻게 결정이 되는 것일까요? 먼저 간단한 예를 들어보겠습니다.

```
function add(c, d, fn) {
    return fn(c + d);
}
let user = {
    a:2,
    b:3,
    add() {
        console.log(this) //user 객체
        add(this.a, this.b, function(total){
            console.log(this) // window
        })
    }
}
user.add();
```

위의 user이라는 객체의 method인 add를 호출하는데 이 때 호출할 때의 console.log를 찍으면 당연히 user객체를 가리킵니다. 하지만 add로 넘어가서 실행되는 함수는 function add(c, d, fn){.. 으로 넘어가서 실행되므로 그 때의 this인 window를 가리킵니다. 이렇게 함수가 어디서 호출되느냐가 this를 판가름하게 됩니다. 이외에도 this의 함수호출패턴은 아래처럼 3가지가 있습니다.

생성자 함수에서의 this

```
function Person(){
    this.value = 'kundol',
    this.printThis = function() {
        console.log(this);
    }
}

var p = new Person();
var print = p.printThis;
p.printThis(); // -> Person {value: "kundol", printThis: f}
print();
// -> Window {stop: f, open: f, alert: f, ...}(브라우저)
// -> Object[global](Node.js)
```

function Person은 class함수와 비슷한 생성자함수인데요. 이렇게 생성자함수안에서의 this는 또 다릅니다. print라는 변수에 메서드함수를 담아 print()로 호출하는 것과 new Person이란 클래스로 만든 객체에서 호출하는 메소드는 다릅니다. 즉, 어디서 호출하느냐에 따라 다른 것입니다. p.printThis의

경우 p라는 new Person이란 클래스로 만든 객체에서 호출하기 때문에 p를 가리키게 됩니다. 하지만 그저 print()로 호출하게 된다면 호출되는 환경에서의 영향을 받습니다. 브라우저의 경우 전역개체인 window를 참조하게 되고 Node.js의 경우 global Object가 됩니다.

올바른 생성자 함수

this를 설명하려 생성자 함수를 들었지만 사실 저는 이렇게 this를 이용하며 생성자 함수를 만드는 것을 추천하지 않습니다.

```
function Person(){
  let age = 0;
  function up(){
    return ++age
  }
  function down(){
    return --age;
  }
  return Object.freeze({
    up, down
  })
}
const a = new Person();
const ret = a.up();
console.log(ret)
const ret2 = a.up();
console.log(ret2)
/*
1
2
*/
```

위 코드를 보면 age라는 변수는 객체의 private 속성을 가지며 up, down 메서드로만 접근이 가능합니다. 객체의 인터페이스는 오직 그 객체의 메서드 하나여야만 합니다.

더 깊게 알아보는 상식! global Object, 브라우저와 Node.js

자바스크립트는 실행하는 환경안에서 global 객체가 달라집니다. 예를들어 브라우저안에서 자바스크립트를 실행한다면 top-level 즉, 최상위 global객체는 다름아닌 window가 됩니다. 그래서 name = 1 이렇게 해도 window.name에 저장이 되는 것을 볼 수 있습니다. 왜냐하면 변수를 선언했는데 var이나 const, let로 선언을 안해주니 이게 대체 어떤 스코프라는 거야? 하면서 계속해서 위의 객체들을 찾게 되고 결국 브라우저의 최상위 객체인 window 객체에 바인딩 되고 맙니다. 그래서 name = 1이라고 선언해도 window.name 으로 접근해서 1이라는 값을 받아낼 수 있게 됩니다. 하지만 Node.js에서의 global은 module그 자체입니다. 예를 들어 a.js라는 파일을 만들고 그 안에서 name = 1

이런식으로 선언하면 그 모듈인 a.js 안에서의 전역변수가 됩니다. 하지만 Node.js에서도 global이라는 것은 존재합니다. 바로 말그대로

```
global.test = "foo";
console.log(test); // "foo"
```

라는 식인데요. 브라우저의 window.test 와 동일한 개념입니다. 하지만 Node.js에서 test = "foo"를 하면 global이 아닌 모듈 그 자체인 a.js의 전역변수가 됩니다. 굳이 Node.js 에서 전역변수를 쓰자면 global을 쓴다는 것입니다. 하지만 이는 추천하지 않습니다. 헷갈릴 수 있는 부분이 브라우저의 global과 Node.js의 global인데 브라우저에서의 global은 단순 상위객체인 window입니다. 이 window라는 객체안에 setTimeout 등의 함수가 있는 것입니다. 하지만 Node.js에서의 global은 global objects에서의 global일 뿐입니다. global이라는 말은 단순 모듈안의 전역변수라고도 명칭할 수도 있습니다. 예를 들어 __dirname 이나 __filename 처럼 내가 어디에다가 선언을 안했는데도 불구하고 바로 자바스크립트로 만든 모듈안에서 쓸 수 있죠? global 객체 처럼 말이죠. 이렇게 Node.js안에서 모든 모듈 예를 들어 내가 만든 a.js, b.js 따위의 파일들이 있을 때 모든 파일들에서 접근이 가능하고 쓸 수 있는 객체를 Global Object 라고 부릅니다.

객체리터럴로 만든 객체에서의 this

```
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, ...}
```

print()의 경우 window라는 전역객체를 가리키게 됩니다. obj.printThis()의 경우 올바르게 obj를 가리키는 것을 볼 수 있습니다. 즉 어떠한 함수를 어떻게 호출되는가에 따라서 this가 결정되게 됩니다. 객체의 메소드로 호출되게 된다면 그 this는 객체를 가리키게 되고 그저 변수로 할당되어서 호출이 된다면 호출될 때의 this는 그 실행맥락에서의 상위 스코프인 전역스코프를 가리키게 됩니다(브라우저에서는 window를 가리키게 되며 Node.js에서는 global 객체를 가리키게 됩니다).

call과 apply, bind로 정해지는 this

.call과 .apply는 모두 함수를 호출하는데 사용되며 첫 번째 매개 변수는 함수 내에서 this의 값으로 사용됩니다. 그러나 두 함수의 다른점은 .call은 두번째 인자부터 쉼표로 구분된 인자들을 인자로 취하고 .apply는 인자로써 배열을 두 번째 인자로 취합니다. 참고로 call을 쓰는 생성자함수는 화살표함수로 만들면 안됩니다. 생성자함수를 정의할 때 정적으로 this가 정해져 window또는 global(Node.js)를 가리키게 되어 생성자함수의 구실을 못하게 됩니다.

```
// call & apply
function a(c, d){
    const b = [this.name, this.adjective, '이가 춤을 춥니다.', c, d].join(' ')
    return b;
}

const 큰돌 = {
    name: '큰돌', adjective: '아주 이쁘게'
};
console.log(a.apply(큰돌, ["하지만 혼자", "외롭게 말이죠"]));
console.log(a.call(큰돌, "하지만 혼자", "외롭게 말이죠"));
/*
큰돌 아주 이쁘게 이가 춤을 춥니다. 하지만 혼자 외롭게 말이죠
큰돌 아주 이쁘게 이가 춤을 춥니다. 하지만 혼자 외롭게 말이죠
*/
```

a라는 함수를 호출할 때 큰돌이라는 객체가 this가 되어 호출하게 만들었습니다. this라는 것이 큰돌을 가리키게 때문에 this.name에 큰돌이 되고 this.adjective는 아주이쁘게 라는 값이 나오게 됩니다. 또한, apply는 배열로써 인자를 넘기고 call은 각각 넘기는 것을 볼 수 있습니다.

이벤트함수에서의 this

event listener로 this를 받게 되면 그 event를 받은 target을 가리키게 됩니다.

03-6 함수형 프로그래밍의 응용 예제

위에서는 함수형 프로그래밍의 기본에 대해서 배웠습니다. 그렇다면 어떻게 하면 이 함수형 프로그래밍을 쓸 수 있을까요? 그렇다면 좀 더 응용한 예제를 살펴보도록 하겠습니다.

배열안에서의 차례대로 작동하는 비동기 로직

먼저 `npm install fxjs`로 `fxjs` 라이브러리를 설치합니다. 보통 `map`이라는 함수를 쓰고 그 안에 비동기 로직을 담아 `async`와 `await`를 쓰면 pending 6개가 찍혔다가 동시에 실행이 될 것입니다. 그만큼 `map` 함수 안에서 비동기 로직을 차례대로 구현하기란 어렵습니다. 위의 코드는 함수형 프로그래밍을 도와주는 라이브러리인 `fx.js`를 이용해 `delay`라는 특정 비동기 함수가 순차적으로 실행되는 예제입니다. 차근차근 보겠습니다. 이렇게 하면 지연평가로 인해 `L.map(a + 100) > L.map(delay)`으로 이어지며 순차적인 비동기적 `map`이 구현되게 됩니다.

```
const FxJS = require("fxjs");
const _ = require("fxjs/Strict");
const L = require("fxjs/Lazy");
const C = require("fxjs/Concurrency");
const log = a => {
  return console.log(`${new Date()} : ${a}`)
}

const delay = (val)=>{ // --- (1)
  return new Promise((resolve, reject)=>{
    setTimeout(() => {
      resolve(val)
    }, 1000);
  })
}

async function test_fp() {
  const list = [1, 3, 5, 6, 7, 9];
  return await _.go(// --- (2)
    list,
    L.map(a => a + 100), // --- (3)
    L.map(delay), // --- (4)
    _.each(log) // --- (5)
  )
}

test_fp().then((ret)=> console.log(ret));
/*
Fri Aug 23 2019 15:13:31 GMT+0900 (GMT+09:00) : 101
Fri Aug 23 2019 15:13:32 GMT+0900 (GMT+09:00) : 103
Fri Aug 23 2019 15:13:33 GMT+0900 (GMT+09:00) : 105
Fri Aug 23 2019 15:13:34 GMT+0900 (GMT+09:00) : 106
Fri Aug 23 2019 15:13:35 GMT+0900 (GMT+09:00) : 107
Fri Aug 23 2019 15:13:36 GMT+0900 (GMT+09:00) : 109
[ 101, 103, 105, 106, 107, 109 ]
*/
```

```
*/
```

- ① 1초동안의 지연효과를 가져올 함수입니다. Promise로 래핑했습니다.
- ② 즉시실행pipe함수입니다. 이것으로 함수합성을 합니다.
- ③ map을 통해 각각 원소에 100을 더합니다.
- ④ map을 이용해 배열의 요소마다 각각 delay를 시킵니다.
- ⑤ Each를 통해 각각 로그를 찍게 만듭니다.

reduce를 통한 집계

```
const FxJS = require("fxjs");
const _ = require("fxjs/Strict");
const L = require("fxjs/Lazy");
const C = require("fxjs/Concurrency");
const users = [
  { id: 1, name: "홍철", age: 22 },
  { id: 2, name: "서영", age: 25 },
  { id: 3, name: "종선", age: 31 },
  { id: 4, name: "제호", age: 27 }
];
const f = (info, user) => {
  const group = user.age - user.age % 10;
  info.count[group] = (info.count[group] || 0) + 1;
  info.total[group] = (info.total[group] || 0) + user.age;
  return info;
}

const ret = _.reduce(f, { count: {}, total: {} }, users)
console.log(ret)
/*
{ count: { '20': 3, '30': 1 }, total: { '20': 74, '30': 31 } }
*/
```

이외에도 reduce를 통해 함수형 프로그래밍을 통하면 쉽게 집계된 데이터를 뽑아낼 수 있습니다. 위의 코드는 20대 10대 별로 그룹화해서 몇 명이 있는지와 나이의 합계를 계산합니다.

지금까지 함수형 프로그래밍을 설명하면서 ramda, lodash, fx.js를 이용한 예제를 설명했습니다. 저는 이 3가지의 함수형 프로그래밍을 도와주는 라이브러리 중 유인동개발자님이 만든 fx.js와 lodash를 추천합니다. 함수형 프로그래밍을 조금 더 알고 싶다면 인프런의 유인동의 함수형프로그래밍 강의를 보기를 추천드립니다.

03-7 프로젝트에서의 CSS와 자바스크립트

프로젝트에서 가장 중요한 것은 뭘까요? 요구사항입니다. 그 다음 중요한 건 바로 맞춤입니다. 큰 프로젝트에서는 여러 사람들이 투입되는데 각자 개인적인 성향의 코드 스타일이 있습니다. 그러한 코드 스타일이 뒤섞여 짬뽕이 되면 맛이 없겠죠? 프로젝트 전 반드시 코드 스타일을 어느정도는 맞추는 것이 중요합니다. css에서는 대표적으로 eslint가 있습니다. 이것을 이용해서 할수도 있지만 직접 스타일을 정해놓고 할 수도 있습니다.

0은 px을 쓰지 않는다. - css

```
article{
  margin:0;
  padding:0;
}
```

short hand를 사용한다. - css

```
/* x */
article{
  margin-top:10px;
  margin-bottom:10px;
}
/* o */
article{
  margin:10px 0px;
}
```

CSS SPRITE 이미지를 쓰자 - css

서버요청을 줄이기 위해 반복되는 이미지는 한장의 이미지를 요청하고 background속성을 이용해서 이미지를 나타낸다.

하위 선택자는 좀 더 깔끔하게 한다. - css

library book article보다는 library article이 좋습니다.

repaint 와 reflow를 최소화 한다 - css

애니메이션을 할 때는 transform, filter, opacity를 사용해야 합니다. 그냥 top, left가 움직이게끔 한다면 repaint와 reflow가 일어나는데 저 속성을 걸게 되면 그 요소가 자체적으로 레이어를 가지게 되며 UI thread에서 composite layer로 GPU에 업로드되어 repaint와 reflow가 일어나지 않습니다.

스타일을 재사용한다. - css

제일 중요합니다. 많은 컴포넌트들 사이에서 재사용할 클래스를 미리 지정해 놓고 사용하는 것이 중요합니다. 예를 들어 A컴포넌트에서 `display : flex`를 사용하는데 B 컴포넌트에서도 `display : flex`를 사용한다면 좋지 않습니다. 부모 컴포넌트에서 `display : flex`를 정의한 후 재사용하는 것이 좋습니다.

이름 짓는 규칙을 정합니다. - js

보통 프로젝트를 하면 컴포넌트단위로 프로젝트를 하게 되는데 그 컴포넌트를 이용해서 이름을 지으면 좋습니다. 예를 들어 App.vue에 들어가는 클래스는 App-main이런식으로 짓기를 결정해 놓으면 나중에 찾을 때도 정말 좋습니다.

자바스크립트의 맞춤

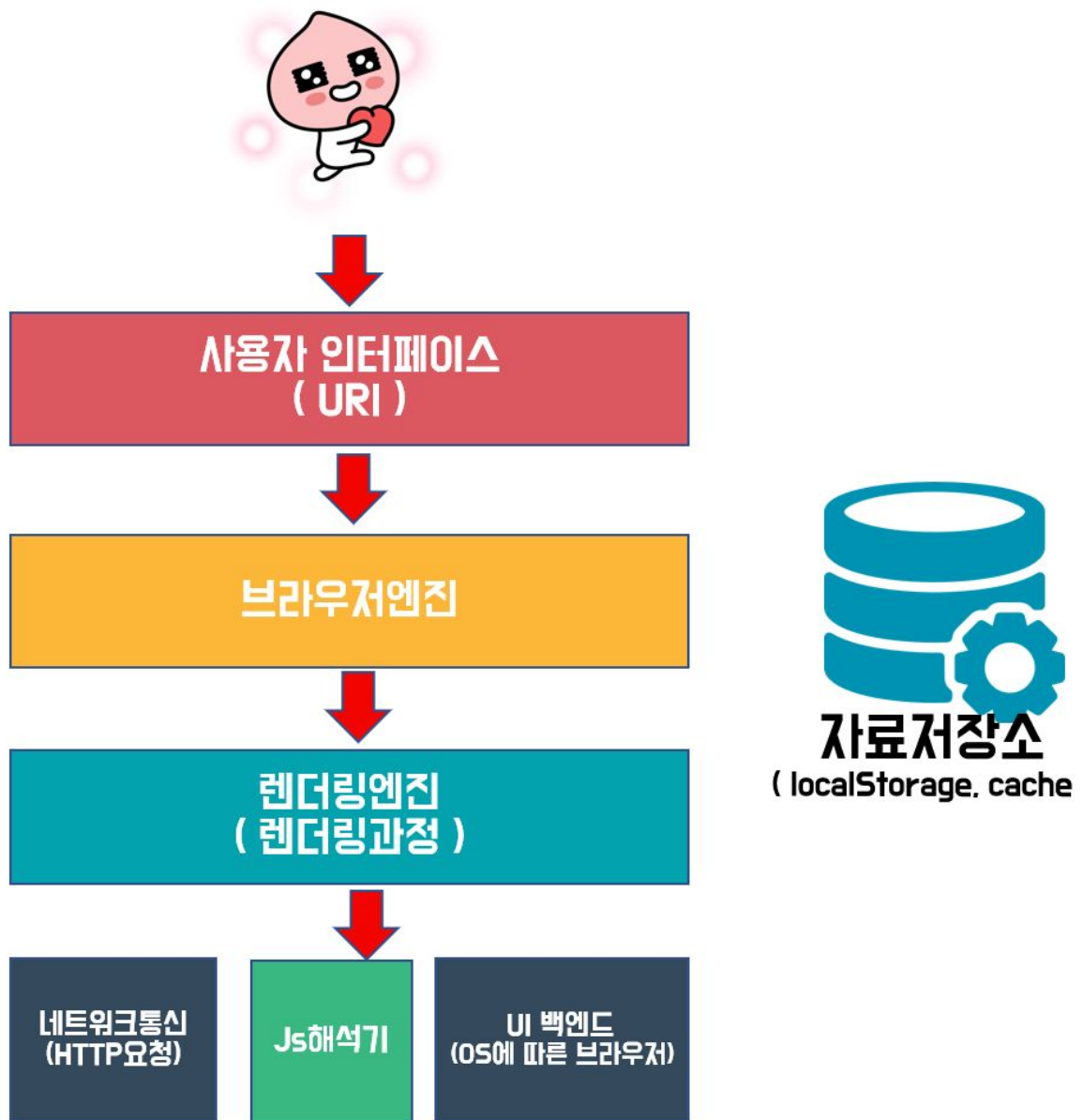
1. 함수, 변수명은 카멜표기법을 사용합니다. 팻줄표기법(underbar 표기법)도 있지만 저는 이것을 추천합니다.
2. 간단하더라도 `if { }` 중괄호를 사용해야 한다.
3. 문자열은 무조건 템플릿문자열을 사용해야 합니다.
4. 한 컴포넌트, 모듈의 길이는 200줄 이하로 작성되어야 한다.
5. 배열순회의 경우 `for-in` 루프를 쓰지 않습니다. 프로토타입 체인까지 검색하므로 느립니다. 하지만 객체 속성순회일 경우에는 씁니다.
6. 정규표현식은 간단하게 씁니다. 예를 들어 `ble|blue` 대신 `blu?e`를 씁니다.
7. 세미콜론은 쓰거나 쓰지 않습니다. 유명한 오픈소스 2개를 예를 들자면 `D3.js`의 경우 쓰고, `nightmare.js`에서는 쓰지 않습니다.

무조건 위의 사항들을 지키라는 것이 아닙니다. 위의 사항들은 하나의 예시입니다. 맘에 드는 부분들만 지켜서 짜도 되고 ESLint문법이나 클린코드를 참고해도 좋고 아예 PR(Pull Request)를 할 때 ESLint를 지키지 않으면 아예 못하게 만들 수도 있습니다. 참고로 Mocha.js같은 오픈소스를 보면 여러 사람들의 스타일을 맞추기 위해 Eslint를 적용하고 있습니다. 중요한 것은 프로젝트에 코드스타일을 정해놓고 개발에 임해야 한다는 점입니다. 그러면 더 아름다운 개발문화, 코드를 만들 수 있을 것입니다.

<더 깊게 알아보는 상식!>

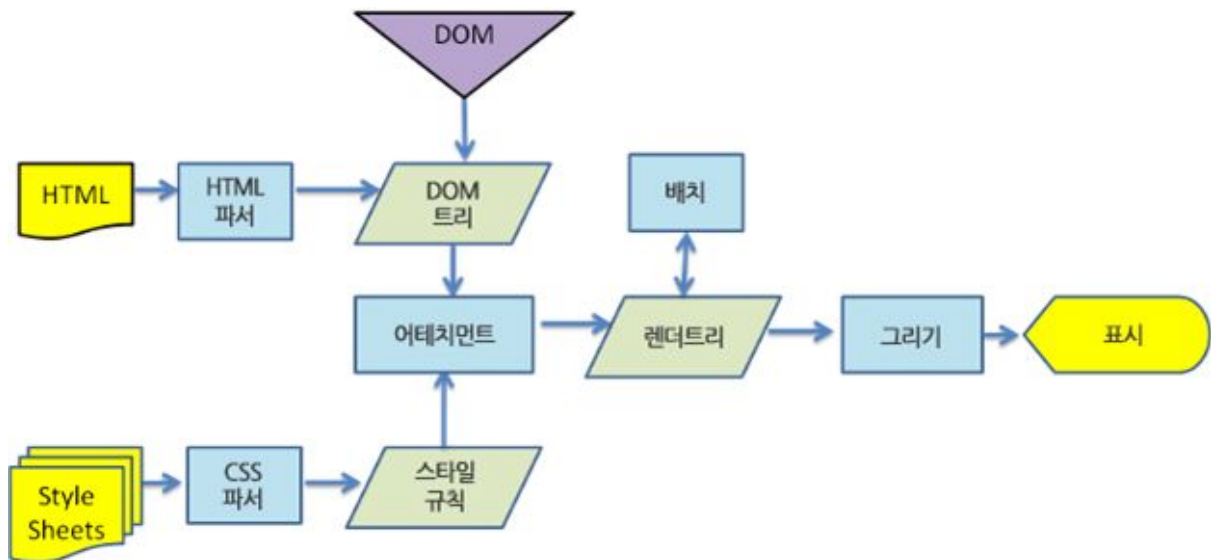
화면의 구성(브라우저 렌더링)과 reflow와 repaint

브라우저 주소 창에 `www.naver.com`을 입력했을 때 어떤 과정을 거쳐 네이버메인 페이지가 보여지게 되는 것일까요? 브라우저는 아래와 같은 요소로 이루어져 있습니다.



<< 03-writer-06.png >>

사용자인터페이스는 URI로 받은 요청을 처리하고 브라우저엔진은 사용자인테페이스와 렌더링 엔진을 이어주는 다리 역할을 하며 자료저장소와도 데이터를 공유합니다. 자료저장소는 localStroage, cache 등의 자료를 저장하는 것을 담당합니다. 렌더링 엔진은 아래 설명할 렌더링 과정을 담당하고 네트워크통신, 자바스크립트 해석기, UI 백엔드로 화면을 이루는 HTML, AJAX, JS등이 합쳐져 만들어지게 됩니다. 이렇게 험난한 과정으로 화면이 형성되는데 이 중 핵심이 되는 렌더링엔진을 보도록 하겠습니다. 브라우저 내의 렌더링엔진이 미리 설정한 html, js 등의 내용을 토대로 브라우저 화면에 표시하는 일을 합니다. 이 렌더링엔진은 파이어폭스는 게코엔진, 사파리와 크롬은 웹킷이란 엔진을 사용합니다. 또한 IE는 Trident를 씁니다. 여기서는 크롬, 웹킷을 기준으로 설명합니다.



<< 03-writer-07.png >>

크롬의 경우 위와 같이 HTML과 CSS가 다 따로 따로 일어나다 DOM트리를 구축하고 렌더트리를 구축하는 것을 볼 수 있습니다. 이 과정을 좀 더 자세히 보겠습니다.

1. DOM 트리 구축 : 하나의 html페이지는 div, span 등 각각의 요소들을 가지고 있습니다. 그러한 요소들 하나하나 Node객체로 설정되고 이것들이 트리형태로 저장이 됩니다. 이를 DOM트리라고 합니다. 예를 들어 div > span, span 이라는 요소가 있다면 div라는 부모노드 밑에 span이라는 자식노드가 2개 생기는 것입니다.
2. 각각의 Node는 CSS파서에 의해 정해진 스타일 규칙이 있습니다. 이 span.color = "red"라고 하면 이 노드의 색깔은 빨간색이다 등을 말하는 것이죠. 이 규칙에 따라 CSSOM이 만들어지고 미리 만들어 놓은 DOM트리내에 있는 노드와 함께 Render Object가 생성되며 이를 모아 병렬적인 렌더트리가 생성 됩니다. 이 때 display:none이 포함된 노드는 지워지고 font-size 등 상속 스타일 부모에만 위치하게설계하는 등의 최적화를 거쳐서 렌더트리를 통해 렌더레이어가 완성됩니다. 참고로 그렇기 때문에 렌더레이어와 렌더오브젝트는 1:1대응이 아니게 됩니다. display:none 으로 사라질 수도 있기 때문입니다. 하지만 DOM트리와 노드는 1:1 대응이 됩니다. 이렇게 렌더트리가 생성된 후 그 후 렌더레이어에 올려지게 됩니다. 하지만 이 때 GPU에서 처리되는 부분(CSS3D / video & canvas / filter / animation / transform : translateZ(0) 등)이 있으면 이 요소들이 각각 강제적으로 Graphic Layer로 분리됩니다. 참고로 visibility:hidden은 display:none과 다르게 보이지는 않지만 비어있는 영역으로 자리를 차지하는게 다릅니다.
3. 렌더레이어를 대상으로 Layout설정(좌표 설정, 보통 부모를 기준으로 설정됨 / Global Layout이 변경될 때는 브라우저의 사이즈가 증가 하거나 폰트사이즈를 증가시키면 변경됩니다)

4. 렌더레이어를 대상으로 paint(한 픽셀 한 픽셀 인쇄하는 듯 칠해지게 됩니다) 레스터화라고도 합니다.
5. composite layer , 이 레이어들은 각각 독립적으로 비트맵으로 출력이 되어 각각 색깔을 가지게 됩니다. 이제 합칠 준비만이 남은 것이죠. 이 후 GPU에 텍스처로써 업로드되어서 각각의 레이어들이 합쳐지게 됩니다. 이 때 렌더레이어와 그래픽레이어들이 합쳐지게 되며 화면이 됩니다.

reflow와 repaint

이렇듯 화면이 구성될 때 렌더링레이어와 그래픽레이어로 분리되며 구성이 되는데 렌더레이어에 포함이 된 요소 중 레이아웃의 변화가 생기면 layout부터 시작해서 다시 설정하는 것이 reflow, 화면의 색깔 등의 변화가 일어나면 repaint가 일어나게 됩니다. reflow의 경우 다음과 같은 요소를 수정하면 일어나게 되며 height, width, scrollHeight, scrollLeft, scrollTop, scrollWidth, scrollTo(), padding, top, left, repaint의 경우 backgroundColor, color 등 색깔에 관한 요소들을 수정하면 일어나게 됩니다. 이 reflow와 repaint가 많이 일어나면 버벅거리는 현상이 일어날 수도 있는데 많이 변경되는 사항에 경우 reflow과 repaint가 일어나지 않도록 그래픽 레이어에 담아서 최적화할 수 있습니다. 예를 들어,

```
@keyframes swim {
  from {
    top:0px;
  }
  to {
    top:200px;
  }
}
```

이것 보다는 transform을 이용해서 y축을 움직여야 합니다. 왜냐하면 요소가 style="transform:translateZ(0);"를 통해 스타일에 CSS 3D속성인 translateZ를 걸게 되면 해당요소는 하드웨어 가속대상이 되게 되며 그 요소만 그래픽 레이어로 분할이 됩니다. 그리고 그 대상이 움직이거나 그럴 때 repaint와 reflow가 일어나지 않게 됩니다. 그러나 무조건 이렇게 GPU 하드웨어 가속에 걸게 되었을 때 성능이 낮은 기기에서는 성능저하를 가져올 수 있습니다. 따라서 하드웨어 가속을 적용하는 요소의 크기를 줄이고 화면에서 몇개 단위로 구성하는 것이 좋으며 기기에 따라 선별적인 하드웨어 가속을 해야 합니다. 보통 30개의 그래픽 레이어가 좋다고 합니다.

</더 깊게 알아보는 상식!>