이터레이션 프로토콜

이터레이션 프로토콜

이터레이션 프로토콜은 순회 가능한 데이터 콜렉션(자료구조)을 만들기 위한 규약

ES6 이전에는 순회 가능한 데이터 컬렉션으로 배열, 문자열, 유사배열객체(Array-Like Object), HTMLcollection 등은 통일된 규약 없이 for문, for...in 문, forEach 메서드 등으로 순회

ES6에서부터는 순회 가능한 데이터 콜렉션은 이터레이션 프로토콜을 준수하는 이터러블로 통일 for...of문, 스프레드 문법, 배열 디스트럭처링 할당의 대상

※ data provider와 data consumer

Data provider: 배열, 문자열, 유사배열객체(Array-Like Object), HTMLcollection

Data consumer: for...of문, 스프레드 문법, 배열 디스트럭처링

이터레이션 프로토콜을 만든 이유

- 데이터 공급자 각각의 순회 방식이 존재하면, 데이터 소비자도 각각의 순회 방식 지원해야 함. 비효율적
- 이터레이션 프로토콜은 데이터 공급자가 하나의 순회 방식을 갖도록 규정
- 이터레이션 프로토콜은 데이터 소비자와 데이터 공급자를 연결하는 인터페이스

이터러블 프로토콜과 이터레이터 프로토콜

이터레이션 프로토콜은 이터러블 프로토콜과 이터레이터 프로토콜로 구성

이터러블 프로토콜

1. 이터러블 프로토콜을 준수하는 객체를 이터러블
2. 이터러블은 Symbol.iterator 메서드가 존재
   - Symbol.iterator를 프로퍼티 키로 갖는 메서드를 직접 구현
   - 프로토타입 체인을 통해 상속 받은 Symbol.iterator 존재
3. Symbol.iterator를 호출하면 이터레이터 프로토콜을 준수하는 이터레이터 객체 반환

이터레이터 프로토콜

1. 이터레이터 프로콜을 준수하는 객체를 이터레이터
2. next 메서드 소유
   - next 메서드를 호출하며 이터러블 순회 (next 메서드는 포인터 역할)
   - 이터러블을 순회하며 이터레이터 리절트 객체 반환

---

for 변수선언문 of 이터러블 동작 원리

1. 이터러블은 이터레이터 반환
2. 이터레이터의 next 메서드를 호출하며 이터러블 순회
3. 이터러블 순회하며 이터레이터 리절트 객체 반환
4. 이터레이터 리절트 객체의 value 프로퍼티 값 변수 선언문에 할당
5. 이터레이터 리절트 객체의 done 프로퍼티 값이 true이면 순회 중지

```
for (const item of [1, 2, 3]) {
  console.log(item);
}

// for..of 문의 내부 동작
const iterable = [1,2,3];
const iterator = iterable[Symbol.iterator]();
```

```
for(;;) {
  const res = iterator.next();
  if(res.done) break;
  const item = res.value;
  console.log(item);
}
```

for 변수선언문 in 객체 동작 원리

- 프로토타입 체인 상에 존재하는 모든 프로토타입의 프로퍼티 중에서 프로퍼티 어트리뷰트 [[enumerable]] 값이 true인 프로퍼티 순회하며 열거
- 프로퍼티 키가 Symbol인 경우, 열거하지 않음

이터러블 관련 메서드

이터러블인지 확인하는 함수

```
const isIterable = v => v !== null && typeof v[Symbol.iterator] === 'function';

isIterable([]); // true
isIterable(''); // true
isIterable(new Map()); // true
isIterable(new Set()); // true
isIterable({}); // false
```

이터러블은 for...of, 스프레드 문법, 배열 디스트럭처링 할당의 대상

```
const array = [1,2,3];

console.log(Symbol.iterator in array); // true

(1) for...of
for(const item of array) {
  console.log(item);
}

(2) 스프레드
console.log([...array]);

(3) 배열 디스트럭처링
const [a, ...rest] = array;
console.log(a, rest); // 1 [2, 3]
```

이터러블이 아닌 경우

```
const obj = {a: 1, b: 2};

(1) Symbol.iterator 프로퍼티가 존재하는지
console.log(Symbol.iterator in obj); // false;

(2) for...of
for(const item of obj) {
  console.log(item); // TypeError: obj is not iterable
}

(3) 배열 디스트럭처링
const [a, b] = obj; // TypeError: obj is not iterable
```

2020년 7월 기준, TC39 프로세스의 stage4 단계에 제안되어 있는 스프레드 프로퍼티 제안은 일반 객체에 ***스프레드 문법 사용을 허용****

```
const obj = {a: 1, b: 2}

console.log({...obj}); // {a: 1, b: 2}
```

이터레이터 관련 메서드

```
// 배열은 이터러블
const array = [1, 2, 3];

// 이터러블은 Symbol.iterator 메서드를 호출하면 iterator 반환
const iterator = array[Symbol.iterator]();

// 이터레이터는 next메서드 보유
console.log('next' in iterator);

const array = [1, 2, 3];

const iterator = array[Symbol.iterator]();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

빌트인 이터러블

- 자바스크립트는 이터레이션 프로토콜을 준수하는 객체인 빌트인 이터러블 제공
- 표준 빌트인 객체들 중, 빌트인 이터러블인 객체

```
Array, Array.prototype[Symbol.iterator]
String, String.prototype[Symbol.iterator]
Map, Map.prototype[Symbol.iterator]
Set, Set.prototype[Symbol.iterator]
TypedArray, TypedArray.prototype[Symbol.iterator]
arguments, arguments[Symbol.iterator]
DOM Collection,
NodeList.prototype[Symbol.iterator]
HTMLCollection.prototype[Symbol.iterator]
```

이터러블과 유사배열객체

유사배열객체는 (1) 인덱스로 접근, (2) length 프로퍼티가 존재하는 객체

```
const arrayLike = {
  0: 1,
  1: 2,
  2: 3,
  length: 3
}

// for문으로 순회 가능함.
for(let i = 0; i < arrayLike.length; i++) {
  console.log(arrayLike[i]) // 1 2 3
}
```

유사배열객체에는 Symbol.iterator 메서드가 없으므로, for of문으로 순회 불가

```
for(const item of arrayLike) {
  console.log(item) // TypeError: arrayLike is not iterable
}
```

유사배열객체 !== 이터러블

arrayLike는 유사배열객체이지만 이터러블은 아님

arguments, nodeList, HTMLcollection은 유사배열객체이면서 이터러블

Array.from을 통해서 유사배열객체 또는 이터러블을 배열로 변환

```
const arrayLike = {
  0: 1,
  1: 2,
  2: 3,
  length: 3
}

const arr = Array.from(arrayLike);
console.log(arr); [1, 2, 3]
```

NodeList와 HTMLcollection 비교

1. NodeList는 childNodes와 querySelector평가된 값이 해당.

- childNodes는 DOM의 변화가 실시간으로 반영되는 **Live collection**
- HTMLCollection은 DOM의 변화가 실시간으로 반영되지 않는 **Static collection**

```
const staticNList = document.querySelectorAll('div');
const dynamicNList = document.body.childNodes;

console.log(dynamicNList);
> NodeList(33) [text, script, text, ul#nav-access, text, comment, text, header#main-header.header-main, ...]

console.log(staticNList);
> NodeList(52) [div.nav-toolbox-wrapper, div#nav-tech-submenu.submenu.js-submenu, div.submenu-column, div#nav-learn-submenu.submenu.js-submenu, ...]


// DOM 변경
const div = document.createElement('div');
document.body.appendChild(div);

console.log(dynamicNList);
> NodeList(34) [text, script, text, ul#nav-access, text, comment, text, header#main-header.header-main, ...]

console.log(staticNList);
> NodeList(52) [div.nav-toolbox-wrapper, div#nav-tech-submenu.submenu.js-submenu, div.submenu-column, div#nav-learn-submenu.submenu.js-submenu, ...]
```

2. HTMLCollection의 예로는 children 프로퍼티 해당

사용자 정의 이터러블

```
const fibonacci = {
    [Symbol.iterator]() {
        let [pre, cur] = [0, 1];
        const max = 10;
        return {
            next() {
                [pre, cur] = [cur, pre + cur];
                return { value: cur, done: cur >= max };

            }
        }
    }
}

for (const num of fibonacci) {
    console.log(num);
}
```

이터러블을 생성하는 함수

```
const fibonacciFunc = function(max) {
    let [pre, cur] = [0, 1];

    return {
        [Symbol.iterator]() {
            return {
                next() {
                    [pre, cur] = [cur, pre + cur];
                    return { value: cur, done: cur >= max };

                }
            }
        }
    }
}

for (const num of fibonacciFunc(10)) {
    console.log(num);
}
```

이터러블이면서 이터레이터인 객체

위 함수는 이터레이터 함수를 생성하려면 별도로 호출해야 함

```
const iterable = fibonacciFunc(5);
const iterator = iterable[Symbol.iterator]();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: 5, done: true }
```

이러한 불편을 없애기 위해 아래와 같이 구현

```
const fibonacciFunc = function (max) {
  let [pre, cur] = [0, 1];

  return {
    [Symbol.iterator]() { return this; }
    next() {
      [pre, cur] = [cur, pre + cur];
      return { value: cur, done: cur >= max }
    }
  }
}

let iter = fibonacciFunc(10);

for(const num of iter) {
  console.log(num);
}


iter = fibonacciFunc(10);

console.log(iter.next()); // { value: 1, done: false }
console.log(iter.next()); // { value: 2, done: false }
console.log(iter.next()); // { value: 3, done: false }
console.log(iter.next()); // { value: 5, done: false }
console.log(iter.next()); // { value: 8, done: false }
console.log(iter.next()); // { value: 13, done: true }
```

무한 이터러블과 지연 평가

무한 이터러블을 생성하는 함수를 통해, 무한 수열을 아래와 같이 구현

```
const fibonacciFunc = function (max) {
  let [pre, cur] = [0, 1];

  return {
    [Symbol.iterator]() { return this; }
    next() {
      [pre, cur] = [cur, pre + cur];
      return { value: cur, done: cur >= max }
    }
  }
}

for(const num of fibonacciFunc()) {
  if(num > 10000) break;
  console.log(num); // 1 2 3 5 8 ... 4181 6765
}

const [f1, f2, f3] = fibonacciFunc();
console.log(f1, f2, f3); // 1 2 3
```

배열이나 문자열은 모든 데이터를 미리 메모리에 확보한 다음 데이터 공급

이터러블은 **데이터가 필요한 시점이 돼서야 비로소 데이터를 생성(지연평가, lazy evaluation)**

즉, for...of문, 배열 디스터럭처링 할당 등이 실행되기 전까지는 데이터 생성 지연

이를 통해서,

1__ 빠른 속도

2__ 메모리 소비 없이 무한도 표현 가능