fx API Reference

함수형 프로그래밍 유틸리티. 지연 평가(Lazy Evaluation)와 Promise 지원.

> ***출처:*** *indongyoo/functional-javascript-01*

핵심 함수

go(...args)

값을 함수 파이프라인에 통과.

```
/**
 * @param {any} initial - 초기값
 * @param {...Function} fns - 순차 적용할 함수들
 * @returns {any} - 최종 결과
 */
fx.go(
    [1, 2, 3, 4, 5],
    fx.filter(n => n % 2 === 0),
    fx.map(n => n * 10),
    console.log  // [20, 40]
);
```

---

pipe(fn, ...fns)

함수 합성.

```
/**
 * @param {Function} fn - 첫 번째 함수
 * @param {...Function} fns - 나머지 함수들
 * @returns {Function} - 합성된 함수
 */
const processData = fx.pipe(
    fx.filter(n => n > 0),
    fx.map(n => n * 2),
    fx.take(3)
);

processData([1, -2, 3, 4, 5]);  // [2, 6, 8]
```

---

curry(fn)

커링 함수 생성.

```
/**
 * @param {Function} fn - 커링할 함수
 * @returns {Function} - 커링된 함수
 */
const add = fx.curry((a, b) => a + b);

add(1, 2);    // 3
add(1)(2);    // 3 (부분 적용)
```

이터러블 함수 (즉시 평가)

map(fn, iter)

모든 요소에 함수 적용.

```
fx.map(n => n * 2, [1, 2, 3]);  // [2, 4, 6]

// 커링
fx.go(
    [1, 2, 3],
    fx.map(n => n * 2)
);
```

---

filter(fn, iter)

조건에 맞는 요소만 필터링.

```
fx.filter(n => n % 2 === 0, [1, 2, 3, 4]);  // [2, 4]
```

---

reduce(fn, acc?, iter)

누적 연산.

```
fx.reduce((a, b) => a + b, 0, [1, 2, 3]);  // 6

// 초기값 생략 - 첫 요소가 초기값
fx.reduce((a, b) => a + b, [1, 2, 3]);  // 6
```

---

each(fn, iter)

각 요소에 함수 실행 (반환값 없음).

```
fx.each(console.log, [1, 2, 3]);
// 1
// 2
// 3
```

---

find(fn, iter)

조건에 맞는 첫 요소 반환.

```
fx.find(user => user.id === 3, users);
```

---

take(n, iter)

처음 n개 요소만 취함.

```
fx.take(3, [1, 2, 3, 4, 5]);  // [1, 2, 3]
```

---

takeAll(iter)

모든 요소를 배열로.

```
fx.takeAll(generator());  // [...모든 값]
```

---

flatten(iter)

1단계 평탄화.

```
fx.flatten([[1, 2], [3, 4]]);  // [1, 2, 3, 4]
```

---

flatMap(fn, iter)

map + flatten.

```
fx.flatMap(x => [x, x * 2], [1, 2]);  // [1, 2, 2, 4]
```

---

range(n)

0부터 n-1까지 배열 생성.

```
fx.range(5);  // [0, 1, 2, 3, 4]
```

지연 평가 함수 (fx.L)

필요할 때만 계산하는 제너레이터 기반 함수들.

L.map(fn, iter)

지연 map.

```
fx.go(
    fx.L.range(10000),  // 0~9999 (지연 생성)
    fx.L.map(n => n * 2),
    fx.L.filter(n => n % 4 === 0),
    fx.take(5)  // 필요한 5개만 계산
);
```

L.filter(fn, iter)

지연 filter.

```
fx.go(
    largeArray,
    fx.L.filter(item => item.active),
    fx.take(10)  // 10개 찾으면 중단
);
```

L.range(n)

지연 range (제너레이터).

```
// 메모리 효율적
for (const i of fx.L.range(1000000)) {
    if (i > 10) break;  // 11개만 생성됨
}
```

L.entries(obj)

객체 entries 지연 생성.

```
fx.go(
    fx.L.entries({ a: 1, b: 2, c: 3 }),
    fx.each(([k, v]) => console.log(k, v))
);
```

L.flatten(iter)

지연 1단계 평탄화.

```
fx.go(
    [[1, 2], [3, 4], [5, 6]],
    fx.L.flatten,
    fx.take(3)  // [1, 2, 3]
);
```

L.deepFlat(iter)

지연 깊은 평탄화.

```
fx.go(
    [[1, [2, 3]], [[4, 5], 6]],
    fx.L.deepFlat,
    fx.takeAll  // [1, 2, 3, 4, 5, 6]
);
```

L.flatMap(fn, iter)

지연 flatMap.

```
fx.go(
    [1, 2, 3],
    fx.L.flatMap(x => [x, x]),
    fx.take(4)  // [1, 1, 2, 2]
);
```

동시성 함수 (fx.C)

Promise 병렬 처리.

C.map(fn, iter)

병렬 map.

```
// 모든 fetch가 동시에 시작
await fx.C.map(id => fetch(`/api/${id}`), [1, 2, 3]);
```

C.filter(fn, iter)

병렬 filter.

```
await fx.C.filter(async item => {
    const result = await validate(item);
    return result.valid;
}, items);
```

C.take(n, iter)

병렬 실행 후 n개 취함.

```
await fx.C.take(3, promises);
```

C.reduce(fn, acc?, iter)

병렬 reduce.

```
await fx.C.reduce(
    async (acc, item) => acc + await fetchValue(item),
    0,
    items
);
```

헬퍼 함수

head(iter)

첫 번째 요소.

```
fx.head([1, 2, 3]);  // 1
```

---

isIterable(a)

이터러블 여부 확인.

```
fx.isIterable([1, 2]);  // true
fx.isIterable('abc');   // true
fx.isIterable(123);     // false
```

---

go1(a, fn)

Promise-aware 단일 함수 적용.

```
fx.go1(Promise.resolve(1), n => n + 1);  // Promise<2>
fx.go1(1, n => n + 1);  // 2
```

사용 패턴

데이터 파이프라인

```
fx.go(
    rawData,
    fx.filter(item => item.active),
    fx.map(item => ({
        id: item.id,
        name: item.name.toUpperCase()
    })),
    fx.take(10)
);
```

지연 평가로 성능 최적화

```
// ❌ 비효율 - 모든 요소 처리
fx.go(
    fx.range(100000),
    fx.map(expensive),
    fx.filter(validate),
    arr => arr.slice(0, 5)
);

// ✅ 효율 - 필요한 만큼만 처리
fx.go(
    fx.L.range(100000),
    fx.L.map(expensive),
    fx.L.filter(validate),
    fx.take(5)
);
```

비동기 처리

```
await fx.go(
    userIds,
    fx.C.map(id => fetchUser(id)),  // 병렬 fetch
    fx.filter(user => user.active),
    fx.map(user => user.name)
);
```

관련 문서

- WKIT_API.md - fx 사용 예제
- GLOBAL_DATA_PUBLISHER_API.md - fx.each 사용