3.3.3 데이터 변환

데이터가 변경 불가능하다면 애플리케이션에서 어떻게 무언가를 바꿀 수 있을까? 함수형 프로그래밍은 한 데이터를 다른 데이터로 변환하는 것이 전부다. 함수형 프로그래밍은 함수를 사용해 원본을 변경한 복사본을 만들어낸다. 그런 식으로 순수 함수를 사용해 데이터를 변경하면, 코드가 덜 명령형이 되고 그에 따라 복잡도도 감소한다.

어떻게 한 데이터를 변환해서 다른 데이터를 만들어낼 수 있는지 이해하기 위해 특별한 프레임워크가 필요한 것은 아니다. 자바스크립트 언어 안에는 이미 그런 작업에 사용할 수 있는 도구가 들어 있이다. 함수형 자바스크립트를 유창하게 사용하기 위해 통달해야 하는 핵심 함수가 2개 있다. `Array.map`과 `Array.reduce`가 바로 그 두 함수다.

이번 절에서는 이 두 함수와 다른 여러 핵심 함수를 사용해 한 유형의 데이터를 다른 유형으로 어떻게 변경할 수 있는지 살펴볼 것이다.

고등학교 명단이 들어 있는 배열을 생각해보자.

```
const schools = ["Yorktown", "Washington & Lee", "Wakefield"];
```

`Array.join` 함수를 사용하면 콤마(,)로 각 학교를 구분한 문자열을 얻을 수 있다.

```
console.log( schools.join(", ") );

// "Yorktown, Washington & Lee, Wakefield"
```

`Array.join`은 자바스크립트 내장 배열 메서드다. `join`은 배열의 모든 원소를 인자로 받은 구분자delimiter로 연결한 문자열을 반환한다. 원래의 배열은 그대로 남는다. `join`은 단지 배열에 대해 다른 해석을 제공할 뿐이다. 프로그래머가 `join`이 제공하는 추상화를 사용하면 문자열을 실제로 어떻게 만드는지에 대해서는 신경 쓰지 않아도 된다.

용해서는 안 된다. 물론 디버깅 등을 위해 이를 위반하는 경우도 자주 있지만, 실제로(특히 병렬/동시성 프로그래밍에서)는 그런 행위에 의해 함수와 외부 환경과의 상호작용이 바뀔 수도 있음을 명확히 인지하고 있어야 한다. 실제 디버깅을 위해 추가한 코드로 인해서 프로그램의 동작이 바뀌는 경우를 하이젠버그(Heisenbug)라고 부르기도 한다. 이는 물리학에서 불확정성의 원리를 발견한 하이젠베르크(Heisenberg)의 이름을 따서 붙인 명칭이다.

'W'로 시작하는 학교만 들어 있는 새로운 배열을 만들고 싶다면 Array.filter 메서드를 사용하면 된다.

```
const wSchools = schools.filter(school => school[0] === "W");

console.log( wSchools );

// ["Washington & Lee", "Wakefield"]
```

Array.filter는 원본 배열로부터 새로운 배열을 만들어내는 자바스크립트 배열 내장 함수이다. 이 함수는 **술어**predicate를 유일한 인자로 받는다. 술어는 불린Boolean 값, 즉 true나 false를 반환하는 함수를 뜻한다. Array.filter는 배열에 있는 모든 원소를 하나씩 사용해 이 술어를 호출한다. filter는 술어에 배열의 원소를 인자로 전달하며, 술어가 반환하는 값이 true이면 해당 원소를 새 배열에 넣는다. 여기서 Array.filter는 각 학교의 이름이 'W'로 시작하는지 여부를 검사한다.

배열에서 원소를 제거해야 할 필요가 있다면 Array.pop이나 Array.splice보다는 Array.filter를 사용하자. Array.filter는 순수 함수다. 다음 예제에서 cutSchool 함수는 특정 학교의 이름을 제외한 새로운 배열을 반환한다.

```
const cutSchool = (cut, list) =>
  list.filter(school => school !== cut);

console.log(cutSchool("Washington & Lee", schools).join(", "));

// "Yorktown, Wakefield"

console.log(schools.join("\n"));

// Yorktown
// Washington & Lee
// Wakefield
```

cutSchool 함수는 "Washington & Lee"가 들어 있지 않은 새로운 배열을 반환한다. join 함수를 사용해 그 새 배열에 들어 있는 두 학교의 이름을 *로 구분한 문자열을 만든다. cutSchool은 순수 함수다. 학교 목록(배열)과 학교 이름을 인자로 받아서 그 이름에 해당하

는 학교를 제외한 나머지 학교로 이뤄진 새 배열을 반환한다.

함수형 프로그래밍에 꼭 필요한 다른 함수로는 Array.map이 있다. Array.map은 술어가 아니라 변환 함수를 인자로 받는다. Array.map은 그 함수를 배열의 모든 원소에 적용해서 반환받은 값으로 이뤄진 새 배열을 반환한다.

```
const highSchools = schools.map(school => `${school} High School`);

console.log(highSchools.join("\n"));

// Yorktown High School
// Washington & Lee High School
// Wakefield High School

console.log(schools.join("\n"));

// Yorktown
// Washington & Lee
// Wakefield
```

이 경우 map 함수는 각 학교 이름 뒤에 'High School'을 추가한다. 이때 원본 schools 배열은 아무 변화가 없다.

이 마지막 예제에서는 문자열의 배열에서 문자열의 배열을 만들었다. map 함수는 객체, 값, 배열, 다른 함수 등 모든 자바스크립트 타입의 값으로 이뤄진 배열을 만들 수 있다. 다음 예제는 학교가 담겨있는 객체의 배열을 반환하는 map 함수를 보여준다.

```
const highSchools = schools.map(school => ({ name: school }));

console.log( highSchools );

// [
// { name: "Yorktown" },
// { name: "Washington & Lee" },
// { name: "Wakefield" }
// ]
```

이 예제는 문자열을 포함하는 배열로부터 객체를 포함하는 배열을 만든다.

배열의 원소중 하나만을 변경하는 순수 함수가 필요할 때도 map을 사용할 수 있다. 다음 예제는 원본 schools 배열을 변경하지 않으면서 'Stratford'라는 이름의 학교를 'HB Woodlawn'으로 바꾼다.

```
let schools = [
  { name: "Yorktown"},
  { name: "Stratford" },
  { name: "Washington & Lee"},
  { name: "Wakefield"}
];

let updatedSchools = editName("Stratford", "HB Woodlawn", schools);

console.log( updatedSchools[1] ); // { name: "HB Woodlawn" }
console.log( schools[1] );        // { name: "Stratford" }
```

schools 배열은 객체의 배열이다. updatedSchools 변수는 editName 함수에 대상 학교 이름, 그 학교의 새 이름, 그리고 schools 배열을 넘겨서 받은 결과를 저장한다. editName 함수는 원본 배열은 그대로 둔채로 학교 이름이 바뀐 새 배열을 반환한다.

```
const editName = (oldName, name, arr) =>
  arr.map(item => {
    if (item.name === oldName) {
      return {
        ...item,
        name
      };
    } else {
      return item;
    }
  });
```

editName는 map 함수를 사용해서 원본 배열로부터 새로운 객체로 이뤄진 배열을 만든다. editName이 Array.map에 전달하는 화살표 함수는 배열의 원소를 item 파라미터로 받으며 파라미터의 이름과 oldName이 같은지 비교해서 이름이 같은 경우에는 새 이름(name)을 객체에 넣어서 반환하고 이름이 다르면 예전 원소를 그대로 반환한다.

editName 함수를 한 줄로 쓸 수도 있다. 다음은 if/else 문장 대신 3항 연산자(? :)를 사용해 editName 함수를 짧게 쓴 코드다.

```
const editName = (oldName, name, arr) =>
  arr.map(item => (item.name === oldName ? { ...item, name } : item));
```

방금 본 예에서는 Array.map에 전달한 변환 함수의 파라미터가 1개뿐이었다. 하지만 실제로 Array.map은 각 원소의 인덱스를 변환 함수의 두 번째 인자로 넘겨준다. 다음 예를 보자.

```
const editNth = (n, name, arr) =>
  arr.map((item, i) => (i === n ? { ...item, name }  : item ));

let updateSchools2 = editNth(2,"Mansfield",schools);

console.log( updateSchools2[2] ); // { name: "Mansfield" }
console.log( schools[2] );        // { name: "Washington & Lee" }
```

editNth에서는 원소와 인덱스를 넘겨주는 Array.map 기능을 사용했다. 변환 함수에서 인덱스 i가 변경 대상 n과 같은 인덱스라면 item에 새 이름을 넣은 객체를 만들어서 반환하고 인덱스가 같지 않으면 원래의 item을 그대로 반환한다. 이 예제는 약간 억지스러운 느낌이 있지만, 실전에서는 필요에 따라 원소만 받는 화살표 함수나 원소와 인덱스를 함께 받는 화살표 함수 중 필요한 쪽을 적절히 선택하면 된다.

객체를 배열로 변환하고 싶을 때는 Array.map과 Object.keys를 함께 사용하면 된다. Object.keys는 어떤 객체의 키로 이뤄진 배열을 반환하는 메서드다.

schools 객체를 학교의 배열로 바꾸고 싶다고 치자.

```
const schools = {
  "Yorktown": 10,
  "Washington & Lee": 2,
  "Wakefield": 5
};

const schoolArray = Object.keys(schools).map(key => ({
  name: key,
  wins: schools[key]
```

```
})
);

console.log(schoolArray);
// [ { name: 'Yorktown', wins: 10 },
//   { name: 'Washington & Lee', wins: 2 },
//   { name: 'Wakefield', wins: 5 } ]
```

이 예제에서 Object.keys는 학교 이름의 배열을 반환한다. 그 배열에 map을 사용해서 길이가 같은 새로운 배열을 만든다. 새로 만들어진 객체의 name 프로퍼티의 값으로는 학교 이름(schools의 키)을 설정하고 wins 프로퍼티의 값으로는 schools에서 그 학교 이름에 해당하는 값을 설정한다.

지금까지는 Array.map과 Array.filter를 사용한 배열 변환을 살펴봤다. 또 Object.keys와 Array.map을 사용해 객체를 배열로 바꿀 수 있음을 알았다. 함수형 프로그래밍에 필요한 마지막 도구는 배열을 기본 타입의 값이나 다른 객체로 변환하는 기능이다.

reduce와 reduceRight 함수를 사용하면 객체를 수, 문자열, 불린 값, 객체, 심지어 함수와 같은 값으로 변환할 수 있다.

수로 이뤄진 배열에서 최댓값을 찾을 필요가 있다고 하자. 배열을 하나의 수로 변환해야 하므로 reduce를 사용할 수 있다.

```
const ages = [21,18,42,40,64,63,34];

const maxAge = ages.reduce((max, age) => {
  console.log(`${age} > ${max} = ${age > max}`);
  if (age > max) {
    return age;
  } else {
    return max;
  }
}, 0);

console.log('maxAge', maxAge);

// 21 > 0 = true
// 18 > 21 = false
// 42 > 21 = true
// 40 > 42 = false
```

```
// 64 > 42 = true
// 63 > 64 = false
// 34 > 64 = false
// maxAge 64
```

ages 배열을 한 값으로 축약reduce했다. 그 값은 최댓값인 64이다. reduce 함수는 변환 함수와 초깃값을 인자로 받는다. 여기서 초깃값은 0이고 처음에 그 값으로 최댓값 max를 설정한다. 변환 함수는 객체의 모든 원소에 대해 한 번씩 호출된다. 처음 변환 함수가 호출될 때는 age가 배열의 첫 번째 원소인 21이고 max는 초깃값인 0이다. 변환 함수는 0과 21중 더 큰 값인 21을 반환한다. 이 반환값인 21이 다음 이터레이션 시 max 값이 된다. 각 이터레이션마다 매번 age와 max를 비교해서 더 큰 값을 반환한다. 마지막으로 배열의 마지막 원소를 직전 변환 함수가 반환한 max와 비교해서 더 큰 값을 최종 값으로 반환한다.

앞의 함수에서 console.log를 제거하고 if/else를 짧게 변경하면 다음 코드를 사용해 배열의 최댓값을 계산할 수 있다.

```
const max = ages.reduce((max, value) => (value > max ? value : max), 0);
```

NOTE_ Array.reduceRight

Array.reduceRight는 Array.reduce와 같은 방식으로 동작한다. 다만 Array.reduceRight는 배열의 첫 번째 원소부터가 아니라 맨 마지막 원소부터 축약을 시작한다는 점이 다르다.[12]

때로 배열을 객체로 변환해야 할 때가 있다. 다음 예제는 reduce를 사용해 값이 들어 있는 배열을 해시로 변환한다.

```
const colors = [
 {
    id: 'xekare',
    title: "과격한 빨강",
    rating: 3
```

12 옮긴이_ 기호로 표시하자면 배열 arr = [a1, ..., an]과 이항 함수 f(v,item)이 있을 때 arr.reduce(f,initVal)는 f(...f(f(initVal,a1),a2)...,an)이다. reduceRight는 적용 방향이 reduce와 반대다. arr.reduceRight(f,initVal)은 g(...g(g(initVal,an),an-1)...,a1)이 된다. 다른 언어의 reduceLeft/reduceRight와 달리 자바스크립트 reduce와 reduceRight는 전달받은 람다의 첫 번째 항에 초깃값과 이전 결괏값을 계속 누적시킨다는 사실에 유의하자.

```
  },
  {
    id: 'jbwsof',
    title: "큰 파랑",
    rating: 2
  },
  {
    id: 'prigbj',
    title: "회색곰 회색",
    rating: 5
  },
  {
    id: 'ryhbhsl',
    title: "바나나",
    rating: 1
  }
];

const hashColors = colors.reduce(
  (hash, {id, title, rating}) => {
    hash[id] = {title, rating};
    return hash;
  },
  {}
);

console.log(hashColors);

//{ 'xekare': { title: '과격한 빨강', rating: 3 },
//  'jbwsof': { title: '큰 파랑', rating: 2 },
//  'prigbj': { title: '회색곰 회색', rating: 5 },
//  'ryhbhsl': { title: '바나나', rating: 1 } }
```

이 예제에서 reduce에 전달한 두 번째 인자는 빈 객체다. 이 빈 객체가 바로 해시에 대한 초깃값이다(즉 처음 해시에는 아무 것도 안 들어 있이다). 각 이터레이션마다 변환 함수는 각괄호([]) 연산을 사용해 해시에 새로운 키를 추가한다. 이때 배열의 각 원소에 있는 id 필드의 값을 키 값으로 사용한다. Array.reduce를 이런 방식으로 사용하면 배열을 한 값으로 축약할 수 있다. 이 예제에서는 객체가 바로 그 이다.

심지어 reduce를 사용해 배열을 전혀 다른 배열로 만들 수도 있다. reduce를 통해 같은 값이 여럿 들어 있은 배열을 서로 다른 값이 한 번씩만 들어 있은 배열로 다음과 같이 바꿀 수 있다.

```
const colors = ["red", "red", "green", "blue", "green"];

const uniqueColors = colors.reduce(
  (unique, color) =>
   distinct.indexOf(color) !== -1 ? unique : [...unique, color],
  []
);

console.log(distinctColors);

// ["red", "green", "blue"]
```

이 예제에서는 colors 배열을 서로 다른 값으로 이뤄진 배열로 만든다. reduce 함수에게 전달한 두 번째 인자는 빈 배열이다. 이 배열은 unique의 최초 값이다. unique 배열에 어떤 색이 이미 들어 있이지 않다면 그 색을 추가한다. 이미 unique에 그 색이 들어 있다면 아무 일도 하지 않고 다음으로 넘어가기 위해 현재의 unique를 그대로 반환한다.

map과 reduce는 함수형 프로그래머가 주로 사용하는 무기이며 자바스크립트도 예외가 아니다. 여러분이 더 자바스크립트를 유창하게 구사하는 엔지니어가 되고 싶다면 반드시 이 두 함수에 통달해야 한다. 한 데이터 집합에서 다른 데이터 집합을 만들어내는 능력은 꼭 필요한 기술이며 프로그래밍 패러다임과 관계없이 유용하다.