[JS-Deep-Dive] 46장- 제너레이터와 async/await

Status	완료

제너레이터와 async/await

제너레이터

> 코드 블록의 실행을 필요한 시점에 일시중지 했다가 필요한 시점에 재개 할수 있는 특수한 함수

제너레이터와 일반 함수의 차이

제너레이터	일반 함수

▫함수 호출자가 함수 실행을 일시중지시키거나 재개 할수 있음 (`제어가능`)

▫함수 호출자와 양방향으로 함수의 상태를 주고 받을 수 있다.

▫제너레이터 함수를 호출 함수 코드를 실행하는것이 아니라 이터러블 하면서 동시에 이터레이터인 제너레이터 객체를 반환

▫함수 호출자는 next 메소드를 이용해 yield 표현식까지 함수를 실행시켜 제너레이터 객체가 관리하는 상태를 꺼내올수있고, 인수를 전달해서 제너레이터 객체에 상태를 밀어 넣을수있다.

> ▫제어권이 함수에게 넘어가고 함수 코드를 한번에 실행
> ▫매개변수를 통해 함수 외부에서 값을 주입받고 함수코드를 일괄실행하여 결과값을 함수 외부로 반환
> ▫함수 코드를 일괄 실행하고 값을 반환

▼ 이터러블 이면서 동시에 이터레이터 의 의미

> 자바스크립트에서 **반복 가능한 데이터 구조**를 다룰 때 자주 등장하는 개념, 이들은 서로 밀접하게 관련되어 있지만, 서로 다른 역할을 함

1. 이터러블 (Iterable)

- 반복가능한 객체 , 즉 데이터를 하나씩 순차적으로 접근할수 있는 객체
- for of 문을 사용하여 이터러블 객체를 순회할수 있음.
- ex) **배열**, **문자열**, **Map**, **Set**
- 이터러블 객체는 반드시 `Symbol.iterator` 메서드를 가지고 있어야 함 이 메서드는 **이터레이터 객체**를 반환

2. 이터레이터 (Iterator)

- **이터러블 객체를 순차적으로 순회할 수 있는 객체**
- 이터레이터는 `next()` 메서드를 제공하며, 이 메서드를 호출할 때마다 이터러블 객체에서 **다음 값을 반환 , 다음값이 없으면** `done` 이 `true` 인 결과를 반환
- 이터레이터 객체는 `{ value, done }` 형태의 객체를 반환합니다.

- `value`: 현재 항목의 값.
- `done`: 순회가 끝났는지를 나타내는 Boolean 값. `done: true`는 더 이상 값이 없다는 뜻

제너레이터 함수의 정의

> ▫제너레이터 함수는 function 키워드로 선언하고, 하나이상의 yield 표현식을 포함함
>
> ▫제너레이터는 화살표 함수로 정의할수 없다
>
> ▫new연산자와함께 생성자 함수로 호출할수 없다.

- 제너레이터 함수는 함수가 호출될 때 즉시 실행되지 않고, yield 를 만날 때마다 실행이 멈추고, 값을 반환하거나 외부로 값을 전달할 수 있음.
- next() 메서드를 호출하면 중단된 위치에서 다시 실행이 시작

```
function* count(num) { //애스터 리스크를 function 과 함수 이름 사이
    let count = 1;
    while (count <= num) {
        yield count; //yield: 제너레이터 함수 내에서 값을 반환할 때
        count++;
    }
}

const test = count(3);

console.log(test.next()); // { value: 1, done: false }
//next(): 제너레이터 함수의 실행을 한 단계 진행시킬 때 사용, 두가지 값 반
console.log(test.next()); // { value: 2, done: false }
console.log(test.next()); // { value: 3, done: false }
console.log(test.next()); // { value: undefined, done: true }
//value: yield로 반환된 값.
//done: 제너레이터 함수가 끝났는지 여부를 나타내는 Boolean 값.
```

▼ yield 표현식?

- `yield` 표현식은 **제너레이터(generator)** 함수 내에서 사용되는 특별한 키워드로, 함수의 실행을 일시 중단하고 값을 반환하거나, 다시 시작할 때 어떤 값을 전달할 수 있게 함 . `yield` 는 **제너레이터 함수**에서만 사용할 수 있음.
- 제너레이터 함수는 함수가 호출될 때 즉시 실행되지 않고, yield 를 만날 때마다 실행이 멈추고, 값을 반환하거나 외부로 값을 전달할 수 있음.
- next() 메서드를 호출하면 중단된 위치에서 다시 실행이 시작

제너레이터 객체

▫제너레이터 함수는 이터러블이면서 동시에 이터레이터한 제너레이터 객체를 생성해 반환한다.

▫제너레이터 객체는 next 메소드를 가지는 이터레이터 이므로 심볼, 이터레이터 메서드를 호출해서 별도로 이터레이터를 생성할 필요가 없다.

▫제너레이터 객체는 이터레이터에는 없는 return, throw 메소드를 갖는다.

제너레이터와 메소드의 동작

- next : 제너레이터 함수의 yield 표현식까지 코드 블록을 실행, `{ value: yield된 값 , done: false }` 된 값을 반환 , 인수를 전달 할수있다
- return : `{ value: 인수로 전달받은 값, done: true }` 의 이터레이터 리절트 객체를 반환한다.
- throw : 인수로 전달받은 에러를 발생, `{ value: undefined, done: true }` 를 값으로 갖는 이터레이터 리절트 객체를 반환

제너레이터의 일시 중지와 재개

```
function* count(max) {
    let count = 1;
    while (count <= max) {
        yield count;
        count++;
    }
```

```
}

const test = count(3);

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

- 제너레이터 함수를 호출시 코드블록이 생성되는것이 아니라 제너레이터 객체를 반환

1. 제너레이터 함수안에 yield 키워드 사용
2. next () 호출
    - 이때 함수의 제어권이 호출자로 양도
    - { value: yield된 값, done: boolean } 객체 반환
    - 전달한 인수는 yield 표현식을 할당받는 변수에 할당됨.
3. 다음 next() 호출
    - 일시중지된 코드부터 실행을 재개 하기 시작하여 다음 yield 표현식 까지 실행되고 다시 일시 중지
    - 호출할 때마다 이터러블 객체에서 다음 값을 반환 , 다음값이 없으면 done **이** true 인 결과를 반환

제너레이터의 활용

- 기존보다 간단한 방식의 이터러블의 구현
    - 기존에 이터러블을 구현하기 위해서는 symbol.iterator 메서드를 구현하고, 이 메서드가 **이터레이터 객체**를 반환해야 하는 이터레이션 프로토콜을 지켜야 했다
- 비동기 처리
    - 제너레이터 함수에서 yield 를 사용해 promise 가 완료되기를 대기하고, next() 가 호출되면 그 다음 값을 반환
    - 즉, 후속처리 메소드없이 비동기 처리결과를 반환하도록 구현가능하다.

```
//제너레이터 생성
function test() {
  return (function* () { //1. test 는 제너레이터를 반환
    console.log('첫번째')
    // 첫 번째 Promise 완료 대기
    const result1 = yield new Promise((resolve) =>
      setTimeout(() => resolve("Result 1"), 1000)
    );
    console.log("두번째:", result1);

    // 두 번째 Promise 완료 대기
    const result2 = yield new Promise((resolve) =>
      setTimeout(() => resolve("Result 2"), 1000)
    );
    console.log("세번째:", result2);

    return "끝";
  })();
}

{value: yield 값 , done: true or false}

// 제너레이터 실행
async function run(generator) {
//2. run 함수는 제너레이터를 실행하면서 yield에서 반환된 Promise를
  const iterator = generator();
  let result = iterator.next();
  //3. 각 Promise가 완료시 결과를 next() 메서드에 전달,다음 단계를

  while (!result.done) {
    const promise = result.value;
    result = iterator.next(await promise); // Promise 결과를
  }

  return result.value; // 제너레이터 종료 시 최종 반환값
}

// 실행
```

```
run(test)

.then((finalResult) =>
  console.log("네번째:", finalResult)
  //4. 마지막 단계에서는 제너레이터가 종료되며 최종 결과("Done")를 빈

);

//결과
첫번째
두번째 :  Result 1
세번째 : Result 2
네번째 : 끝
```

async / await

> ▫es8 부터는 async/await이 도입되어 제너레이터 보다 간단하고 가독성좋게 비동기 처리를 구현할수 있다.

async

- await 키워드는 반드시 async 함수 내부에서 사용해야함
- async 함수는 키워드를 사용해 정의
- async 함수가 명시적으로 프로미스를 반환하지 않더라도 이 함수는 암묵적으로 반환 값을 resolve 하는 프로미스를 반환한다.

await

- 프로미스 settled한 상태가 될때까지 대기하다가 상태가 되면 프로미스가 resolve한 처리 결과를 반환한다. await 키워드는 반드시 프로미스 앞에서 사용해야한다.

- await 키워드는 프로미스가 settled한 상태가 될때 까지 대기함. 따라서 fetch 함수가 수행한 요청에 대해 서버의 응답이 도착해서 반환한 프로미스가 settled 상태가 될때까지 대기, settled 상태가 되면 프로미스가 resolve 한 처리 결과가 변수에 할당된다.

async/ await 에서의 에러처리

- try...catch 문을 사용해서 에러를 처리, 콜백함수를 전달받는 비동기 함수와 달리 promise를 반환 하는 비동기 함수는 명시적으로 호출할수 있기 때문에 호출자가 명확

- async 함수 내 에서 catch 문을 사용하지 않으면 발생한 에러를 reject 하는 promise를 반환하기 때문에, 함수내부가 아니라 함수 호출 후 후속처리 메소드를 사용해서 에러를 catch 할 수도있다.
