CH.04 콜백 함수

3조 허 동, 최낙훈

목차

1. 콜백 함수란?

다른 코드의 인자로 넘겨주는 함수

> 콜백 함수를 넘겨 받은 코드는 콜백 함수를 필요에 따라 실행

함수 X를 호출하면서 '특정 조건일 때 콜백 함수 Y를 실행해라'는 요청을 전송

> 콜백 함수는 다른 코드(함수 또는 메서드)에게 인자를 넘겨줌

> 이 때 콜백 함수는 제어권을 함께 위임함

2. 제어권

예제 4-1 SetInterval

```
1  + var count = 0;
2  + var timer = setInterval(function() {
3  +    console.log(count);
4  +    if (++count > 4) clearInterval(timer);
5  + }, 300);
```

1번째 줄 : count 변수 선언 및 0 할당
2번째 줄 : timer 변수 선언 및 setInterval 실행 결과 할당

SetInterval 의 구조

```
* setInterval의 구조
var intervalID = scope.setInterval(func, delay[, param1, param2, ...]);

scope - Window 객체 또는 Worker의 인스턴스가 들어올 수 있음
* 두 객체 모두 setInterval 메서드 제공
* 일반적인 브라우저 환경에서는 window 생략

매개변수 func, delay
*  반드시 포함해야함
* func 는 함수 delay는 ms 단위의 숫자

나머지 param1, param2, ...
* func 함수를 실행할 때 사용하는 매개변수

setInterval 를 실행하면 반복적으로 실행되는 내용 자체를 특정할 수 있는
고유 ID 값이 반환됨
* 이를 변수에 담는 이유는 반복 실행되는 중간에 종료(clearInterval)하기 위함
```

2. 제어권

예제 4-2 SetInterval

```
1  + var count = 0;
2  + var cbFunc = function() {
3  +   console.log(count);
4  +   if (++count > 4) clearInterval(timer);
5  + };
6  + var timer = setInterval(cbFunc, 300);
7  +
8  + // -- 실행 결과 --
9  + // 0  (0.3초)
10 + // 1  (0.6초)
11 + // 2  (0.9초)
12 + // 3  (1.2초)
13 + // 4  (1.5초)
```

6번째 줄
timer 변수 : setInterval 의 ID 값 저장
setInterval의 첫번재 인자 cbFunc 함수(콜백함수) 는 0.3초 마다 자동으로 실행됨

cbFunc 함수 내부에서는 count값을 출력하고 count에 1을 더한 다음 4보다 크면
반복 실행을 종료함

cbFunc 실행 방식과 제어권

* cbFunc 실행 방식과 제어권

cbFunc(); -> 호출 주체 : 사용자 제어권 : 사용자
setInterval(cbFunc, 300); -> 호출 주체 : setInterval 제어권 : setInterval

cbFunc 함수를 다른 코드(setInterval)의 인자로 넘겨주면서 제어권 또한 넘겨줌
setInterval의 자체 판단에 따라 적절한 시점에 콜백 함수를 실행

콜백함수의 제어권을 넘겨받은 코드는 콜백 함수 호출 시점 제어권을 가짐

2. 제어권

예제 4-3 Array.prototype.map

```
1번째 줄 : newArr 변수 선언, 우항의 결과 할당
우항은 배열 [ 10, 20, 30 ]에 map 메서드를 호출함
```

```
1  var newArr = [10, 20, 30].map(function(currentValue, index) {
2      console.log(currentValue, index);
3      return currentValue + 5;
4    });
5    console.log(newArr);
6
7    // -- 실행 결과 --
8    // 10 0
9    // 20 1
10   // 30 2
11   // [15, 25, 35]
```

```
* map 메서드의 동작 방식(Array.prototype이 배열)
Array.prototype.map(callback[ , thisArg])
여기서 callback : function(currentValue, index, array)

map 메서드의 첫번째 인자 : callback 함수
두번째 인자 : callback 함수 내부에서 this 로 인식할 대상(생략 시 전역객체)

map 메서드는 메서드의 대상이 되는 배열의 모든 요소를 처음부터 끝까지
하나씩 꺼내어 콜백 함수를 반복 호출하고, 콜백 함수의 실행 결과들을 모아
새로운 배열을 만든다
```

```
배열 [10, 20, 30]의 각 요소를 처음부터 하나씩 꺼내어 콜백함수 실행
첫번째 실행 : currenValue = 10,  index = 0, return 15
두번째 실행 : currenValue = 20,  index = 1, return 25
세번째 실행 : currenValue = 30,  index = 2, return 35
```

2. 제어권

예제 4-4 인자의 순서를 임의로 바꾸어 사용한 경우

```
+ var newArr2 = [10, 20, 30].map(function(index, currentValue) {
+     console.log(index, currentValue);
+     return currentValue + 5;
+   });
+   console.log(newArr2);
+
+   // -- 실행 결과 --
+   // 10 0
+   // 20 1
+   // 30 2
+   // [5, 6, 7]
```

제이쿼리(jQuery)의 메서드들은 첫번재 인자에 index, 두번째 인자에 currentValue

map의 인자를 제이쿼리와 같이 순서를 바꾸어 사용할 경우 예제와 같이 오류가 발생

따라서 정의된 규칙에 따라 함수를 작성해야한다.
* map 메서드에 정의된 규칙에는 콜백 함수의 인자로 넘어올 값들 및 순서도 포함되어 있다.

콜백 함수의 제어권을 넘겨받은 코드는 콜백 함수를 호출할 대 인자에 어떤 값들을 어떤 순서로 넘길 것인지에 대한 제어권을 가짐

2. 제어권

콜백 함수 또한 함수이기 때문에 기본적으로 this 는 전역객체를 참조
* 제어권을 넘겨받을 코드에서 콜백 함수에 별도로 this를 지정할 경우에는 그 대상을 참조한다.

메서드 구현의 핵심 -> call/apply

예제 4-5 콜백 함수에 별도의 this 지정하기

```
1  Array.prototype.map = function(callback, thisArg) {
2      var mappedArr = [];
3      for (var i = 0; i < this.length; i++) {
4        var mappedValue = callback.call(thisArg || window, this[i], i, this);
5        mappedArr[i] = mappedValue;
6      }
7      return mappedArr;
8    };
```

4번째 줄 : thisArg에 값이 있을 경우 thisArg 값을, 없을 경우 전역객체를 this에 지정

첫번째 인자 : map 메서드의 this가 배열을 가리킬 것이기 때문에 배열의 i번재 요소 값에 할당
두번째 인자 : index i 값 지정
세번째 인자 : 배열 자체를 지정
결과값이 mappedValue에 담겨 mapperArr의 i번째 인자에 할당됨

* this에 다른 값이 담기는 이유
제어권을 넘겨받을 코드에서 call/apply 메서드의 첫번째 인자에 콜백 함수 내부에서 this가 될 대상을 명시적으로 바인딩하기 때문

2. 제어권

예제 4-6 콜백 함수 내부에서의 this

```
1  setTimeout(function() {
2    console.log(this);
3    }, 300); // (1) Window { ... }
4
5    [1, 2, 3, 4, 5].forEach(function(x) {
6      console.log(this); // (2) Window { ... }
7    });
8
9    document.body.innerHTML += '<button id="a">클릭</button>';
10   document.body.querySelector('#a').addEventListener(
11     'click',
12     function(e) {
13       console.log(this, e); // (3) <button id="a">클릭</button>
14     } // MouseEvent { isTrusted: true, ... }
15   );
```

1번째 : setTimeout은 내부에서 콜백 함수를 호출할 때 call 메서드의 첫번째 인자에 전역객체를 넘기기 때문에 콜백 함수 내부에서의 this가 전역객체를 가리킴

2번째 : forEach는 별도의 인자로 this를 받지만 this를 별도로 명시하지 않았기 때문에 콜백 함수 내부의 this는 전역객체를 가리킨다

3번째 : addEventListener는 내부에서 콜백함수를 호출할 때 call 메서드의 첫번째 인자에 addEventListner 메서드의 this를 그대로 넘기도록 정의되어 있기 때문에 콜백 함수 내부에서의 this가 addEventListner를 호출한 주체인 HTML 엘리먼트가 된다

3. 콜백 함수는 함수다

예제 4-7 함수로서 호출되는 객체의 메서드

```
1  var obj = {
2      vals: [1, 2, 3],
3      logValues: function(v, i) {
4        console.log(this, v, i);
5      }
6    };
7    obj.logValues(1, 2); // { vals: [1, 2, 3], logValues: f } 1 2
8    [4, 5, 6].forEach(obj.logValues); // Window { ... } 4 0
9    // Window { ... } 5 1
10   // Window { ... } 6 2
```

콜백 함수로 어떤 객체의 메서드를 전달하더라도 그 메서드는 메서드가 아닌 함수로서 호출된다

obj 객체의 logValues : 메서드로 정의됨

7번째 줄에서 메서드로서 호출(this : obj)

8번째 줄에서 forEach 함수의 콜백함수로 logValues(메서드)가 호출
obj.logValues가 가리키는 함수만을 전달
-> 메서드로서 호출되지 않는 한 obj와 연관이 없어짐

forEach에 의해 콜백이 함수로서 호출 + 별도의 this 지정 x
-> 함수 내부에서의 this 는 전역객체

4. 콜백 함수 내부의 this에 다른 값 바인딩하기

예제 4-8 콜백 함수 내부의 this에 다른 값 바인딩 –전통적인 방법

```
1   var obj1 = {
2       name: 'obj1',
3       func: function() {
4         var self = this;
5         return function() {
6           console.log(self.name);
7         };
8       },
9     };
10    var callback = obj1.func();
11    setTimeout(callback, 1000);
```

앞선 예제 4-7 을 통해 콜백 함수로서 사용된 메서드는 함수만을 전달하므로
this 가 전역 객체를 가리킨다
-> 객체의 메서드를 콜백 함수로 전달하면 해당 객체를 this로 바라볼 수 없다
-> 그렇다면 어떻게 콜백 함수 내부에서 this 가 객체를 바라보게 할 수 있을가?

this 를 다른 변수에 담아 콜백 함수로 활용할 함수에서 변수를 사용하고 이를
클로저로 만드는 방식 사용 (전통적인 방식)

obj1.func 메서드 내부에서 this를 self 변수에 담고 익명 함수를 선언과 동시에 반환
10번째 줄 : obj.func를 호출 -> 선언한 내부함수가 return되어 callback 변수에 전달
11번째 줄 : 1초 뒤 콜백함수 callback이 실행되며 obj1 출력
-> 매우 번거로움

4. 콜백 함수 내부의 this에 다른 값 바인딩하기

예제 4-9 콜백 함수 내부에서 this를 사용하지 않은 경우

```
1  var obj1 = {
2      name: 'obj1',
3      func: function() {
4        console.log(obj1.name);
5      },
6    };
7    setTimeout(obj1.func, 1000);
```

예제 4-8에서 this 를 사용하지 않은 결과이다
-> 간결, 직관적
그러나 다양한 상황에서 재사용할 수 없다

4. 콜백 함수 내부의 this에 다른 값 바인딩하기

예제 4-10 func 함수의 재활용

```
1  var obj1 = {
2      name: 'obj1',
3      func: function() {
4        var self = this;
5        return function(){
6          console.log(self.name);
7        } ;
8      }
9    };
10   var obj2 = {
11     name: 'obj2',
12     func: obj1.func
13   };
14   var callback2 = obj2.func();
15   setTimeout(callback2, 1500);
16
17   var obj3 = { name: 'obj3' };
18   var callback3 = obj1.func.call(obj3);
19   setTimeout(callback3, 2000);
```

14, 15번째 줄
callback2에는 obj1의 func를 복사한 obj2의 func를 실행결과를 담아 콜백으로 사용

18, 19번째 줄
callback3에는 obj1의 func를 실행하면서 this를 obj3로 지정하고 콜백으로 사용

-> 번거롭지만 this를 우회적으로 활용
(다양한 상황에서 원하는 객체를 바라보는 콜백 함수 제작 가능)

예제 4-9sms 처음부터 객체를 obj1으로 명시하였기 때문에 다른 객체를 바라볼 수 없다

4. 콜백 함수 내부의 this에 다른 값 바인딩하기

예제 4-11 bind 메서드를 활용하여 this를 바인딩하기

```
1   var obj1 = {
2       name: 'obj1',
3       func: function() {
4         console.log(this.name);
5       }
6     };
7     setTimeout(obj1.func.bind(obj1), 1000);
8
9     var obj2 = { name: 'obj2' };
10    setTimeout(obj1.func.bind(obj2), 1500);
```

ES5에서 등장한 bind 메서드를 활용하면 전통적인 방식의 아쉬움을 보완하며 콜백 함수 내부의 this 에 다른 값을 바인딩 할 수 있다.

5. 콜백 지옥과 비동기 제어

콜백 지옥 : 콜백 함수를 익명 함수로 전달하는 과정이 반복되어 코드의 들여쓰기 수준이 감당하기 힘든 정도로 깊어지는 현당
-> 이벤트 처리, 서버 통신과 같은 비동기적 작업을 수행하기 위해 자주 등장
-> 가독성이 떨어지고 코드 수정이 어려움

* 비동기 / 동기
동기적인 코드 : 현재 실행 중인 코드가 완료된 후 다음 코드를 실행하는 방식
비동기적인 코드 : 현재 실행 중인 코드의 완료 여부와 무관하게 다음 코드 실행

CPU의 계산에 의해 즉시 처리가 가능한 코드는 주로 '동기적인 코드'
-> 계산식이 복잡하여 CPU 처리 시간이 길더라도 동기적인 코드

setTimeout : 사용자의 요청에 의해 특정 시간이 경과되기 전까지 함수의 실행 보류
addEventListener : 사용자의 직접적인 개입이 있을 때 함수를 실행하도록 대기
XMLHttpRequest : 웹브라우저 자체가 아닌 별도의 대상에 무언가를 요청하고 그에 대한 응답이 있을 때 비로소 함수 실행하도록 대기
+ 별도의 요청, 실행 대기, 보류 등과 관련된 코드
-> '비동기적인 코드'

5. 콜백 지옥과 비동기 제어

예제 4-12 콜백 지옥 예시

```
setTimeout(
    function(name) {
      var coffeeList = name;
      console.log(coffeeList);

      setTimeout(
        function(name) {
          coffeeList += ', ' + name;
          console.log(coffeeList);

          setTimeout(
            function(name) {
              coffeeList += ', ' + name;
              console.log(coffeeList);

              setTimeout(
                function(name) {
                  coffeeList += ', ' + name;
                  console.log(coffeeList);
                },
                500,
                '카페라떼'
              );
            },
            500,
            '카페모카'
          );
        },
        500,
        '아메리카노'
      );
    },
    500,
    '에스프레소'
);
```

예제 4-12는 비동기적인 코드로 인해 콜백 지옥에 빠진 예시이다
목적을 달성하였으나 들여쓰기 수준이 과도하고 '아래에서 위로' 값이 출력되어 어색하다
-> 가독성 문제, 어색함

5. 콜백 지옥과 비동기 제어

```
1  var coffeeList = '';
2
3  var addEspresso = function(name) {
4    coffeeList = name;
5    console.log(coffeeList);
6    setTimeout(addAmericano, 500, '아메리카노');
7  };
8  var addAmericano = function(name) {
9    coffeeList += ', ' + name;
10   console.log(coffeeList);
11   setTimeout(addMocha, 500, '카페모카');
12 };
13 var addMocha = function(name) {
14   coffeeList += ', ' + name;
15   console.log(coffeeList);
16   setTimeout(addLatte, 500, '카페라떼');
17 };
18 var addLatte = function(name) {
19   coffeeList += ', ' + name;
20   console.log(coffeeList);
21 };
22
23 setTimeout(addEspresso, 500, '에스프레소');
```

예제 4-13 기명함수로 변환 (해결법)

- 코드의 가독성을 높이고 어색함을 해결

- 변수가 최상단으로 끌어올려져 외부에 노출되었지만 전체를 즉시 실행 함수로 감싸면 해결 가능

- 일회성 함수에 전부 변수를 할당하여야 한다

- 코드명을 따라 가다가 헷갈릴 소지가 존재

-> ES6 : Promise, Genrator ES2017 : async , await 도입
-> 비동기적인 일련의 작업을 동기적, 동기적인 것 처럼 보이게 처리하는 장치

5. 콜백 지옥과 비동기 제어

예제 4-14

비동기 작업의 동기적 표현 Promise

```
new Promise(function(resolve) {
    setTimeout(function() {
      var name = '에스프레소';
      console.log(name);
      resolve(name);
    }, 500);
  })
    .then(function(prevName) {
      return new Promise(function(resolve) {
        setTimeout(function() {
          var name = prevName + ', 아메리카노';
          console.log(name);
          resolve(name);
        }, 500);
      });
    })
    .then(function(prevName) {
      return new Promise(function(resolve) {
        setTimeout(function() {
          var name = prevName + ', 카페모카';
          console.log(name);
          resolve(name);
        }, 500);
      });
    })
    .then(function(prevName) {
      return new Promise(function(resolve) {
        setTimeout(function() {
          var name = prevName + ', 카페라떼';
          console.log(name);
          resolve(name);
        }, 500);
      });
    });
```

ES6의 Promise를 이용한 방식

new 연산자와 함께 호출한 Promise의 인자로 넘겨주는 콜백 함수는 호출할 때
바로 실행되지만 그 내부에 resolve 또는 reject 함수를 호출하는 구문이 있을 경우
둘 중 하나가 실행되기 전까지 다음(then) 또는 오류 구문(catch)로 넘어가지 않음

따라서 비동기 작업이 완료될때 다음 함수로 넘어가는 방법으로
비동기 작업의 동기적 표현이 가능해짐

5. 콜백 지옥과 비동기 제어

예제 4-15 비동기 작업의 동기적 표현 Promise

```
var addCoffee = function(name) {
    return function(prevName) {
      return new Promise(function(resolve) {
        setTimeout(function() {
          var newName = prevName ? prevName + ', ' + name : name;
          console.log(newName);
          resolve(newName);
        }, 500);
      });
    };
  };
  addCoffee('에스프레소')()
    .then(addCoffee('아메리카노'))
    .then(addCoffee('카페모카'))
    .then(addCoffee('카페라떼'));
```

예제 4-14의 반복적인 내용을 함수화하여 짧게 표현함
2, 3번째 줄에 클로저 등장

5. 콜백 지옥과 비동기 제어

예제 4-16 비동기 작업의 동기적 표현 Generator

```
1  var addCoffee = function(prevName, name) {
2      setTimeout(function() {
3        coffeeMaker.next(prevName ? prevName + ', ' + name : name);
4      }, 500);
5    };
6    var coffeeGenerator = function*() {
7      var espresso = yield addCoffee('', '에스프레소');
8      console.log(espresso);
9      var americano = yield addCoffee(espresso, '아메리카노');
10     console.log(americano);
11     var mocha = yield addCoffee(americano, '카페모카');
12     console.log(mocha);
13     var latte = yield addCoffee(mocha, '카페라떼');
14     console.log(latte);
15   };
16   var coffeeMaker = coffeeGenerator();
17   coffeeMaker.next();
```

ES6의 Generator 이용

6번째 줄 function* 가 Generator 함수 실행 시 Iterator 반환
Iterator는 next라는 메서드를 지님
next 메서드를 호출하면 Generator 함수 내부에서 가장 먼저 등장하는
yield에서 함수 실행을 정지
이후 다시 next 메서드를 호출하면 멈췄던 부분부터 시작해서 그 다음 등장하는
yield에서 함수 실행을 정지

즉, 비동기 작업이 완료되는 시점마다 next메서드를 호출해준다면 Generator 함수
내부의 소스가 위에서 아래로 순차적으로 진행됨
Generator 함수 실행 시

5. 콜백 지옥과 비동기 제어

예제 4-17 비동기 작업의 동기적 표현 Promise + Async / await

```
var addCoffee = function(name) {
    return new Promise(function(resolve) {
      setTimeout(function() {
        resolve(name);
      }, 500);
    });
  };
  var coffeeMaker = async function() {
    var coffeeList = '';
    var _addCoffee = async function(name) {
      coffeeList += (coffeeList ? ',' : '') + (await addCoffee(name));
    };
    await _addCoffee('에스프레소');
    console.log(coffeeList);
    await _addCoffee('아메리카노');
    console.log(coffeeList);
    await _addCoffee('카페모카');
    console.log(coffeeList);
    await _addCoffee('카페라떼');
    console.log(coffeeList);
  };
  coffeeMaker();
```

ES2017 에서는 가독성이 뛰어나면서 작성법도 간단한 새로운 기능을 추가
-> async / await

비동기 작업을 수행하고자 하는 함수 앞에 async를 표기하고
함수 내부에서 실질적인 비동기 작업이 필요한 위치에 await를 표기

-> 뒤에 내용을 Promise로 자동 전환하고 그 내용이 resolve 된 이후에 다음으로 진행

즉 Promise의 then과 유사한 효과를 얻을 수 있다