코어 자바스크립트 - 02. 실행 컨텍스트

yulikim · 2023년 4월 8일

JavaScript

2-1. 실행 컨텍스트란?

스택(stack)

- 출입구가 하나뿐인 깊은 우물 같은 데이터 구조

> **콜 스택(call stack)**
>
> - 여러 함수들을 호출하는 스크립트에서 해당 위치를 추적하는 **인터프리터** (웹 브라우저의 자바스크립트 인터프리터 같은)를 위한 메커니즘
>   - 자바스크립트 엔진이 구동되면서 실행 중인 코드를 추적하는 공간
>   - **함수의 호출을 기록하는 스택(자료구조)**
> - 현재 어떤 함수가 동작하고있는 지, 그 함수 내에서 어떤 함수가 동작하는 지, 다음에 어떤 함수가 호출되어야하는 지 등을 제어

> **인터프리터(interpreter)**
> 고급 언어로 작성된 원시코드 명령어들을 **한번에 한 줄씩 읽어들여서 실행하는 프로그램**

큐(queue)

- 일반적으로 한쪽은 입력만, 다른 한쪽은 출력만을 담당하는 구조

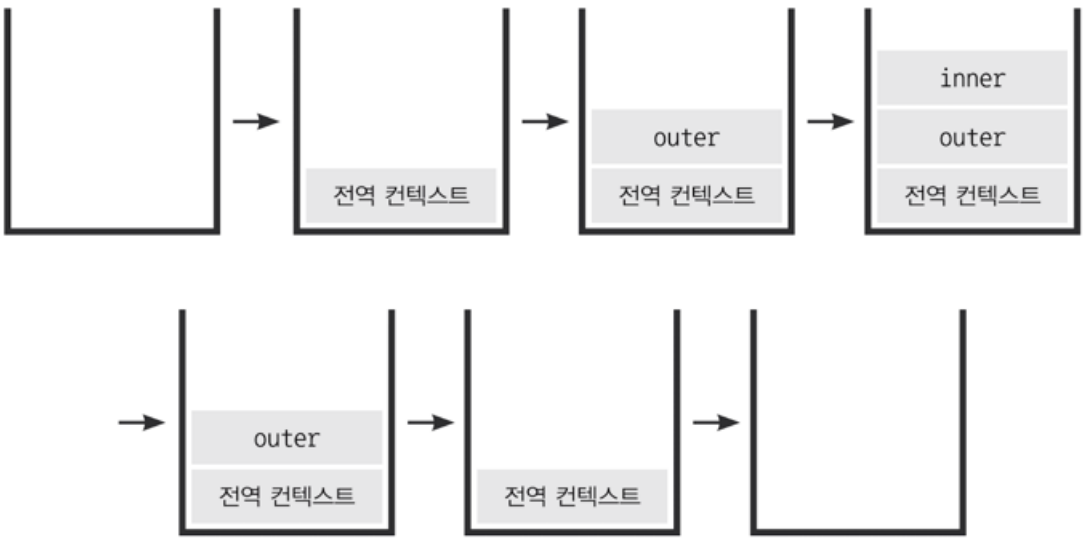
실행 컨텍스트

- 실행할 코드에 제공할 환경 정보들을 모아놓은 객체
- 자바스크립트의 동적 언어로서의 성격을 가장 잘 파악할 수 있는 개념
- **동일한 환경**에 있는 코드들을 실행할 때 필요한 환경 정보들을 모아 컨텍스트를 구성하고,
  이를 콜 스택에 쌓아올렸다가, 가장 위에 쌓여있는 컨텍스트와 관련있는 코드들을 실행하는 방식으로 전체 코드의 환경과 순서를 보장한다.

> **동일한 환경, 하나의 실행 컨텍스트를 구성할 수 있는 방법**
>
> - 전역공간(자동 생성)
> - eval() 함수
> - **함수** (흔히 컨텍스트를 구성하는 방법)

```
// ------------------------(1)
1  var a = 1;
2  function outer() {
3   function inner() {
4     console.log(a); // undefined
5     var a = 3;
6   }
7   inner(); // -------------(2)
8   console.log(a); // 1
9 }
10 outer(); // ---------------(3)
11 console.log(a); // 1
```

그림 2-3 실행 컨텍스트와 콜 스택



1. 처음 자바스크립트 코드를 실행하는 순간**(1)** 전역 컨텍스트가 콜 스택에 담긴다.
2. 전역 컨텍스트와 관련된 코드들을 순차로 진행하다가 **(3)**에서 outer 함수를 호출하면 자바스크립트 엔진은 outer에 대한 환경 정보를 수집해서 outer 실행 컨텍스트를 생성한 후 콜 스택에 담는다.
3. 전역 컨텍스트와 관련된 코드의 실행을 일시중단하고 outer 실행 컨텍스트와 관련된 코드인 outer 함수 내부의 코드들을 순차로 실행한다.
4. 다시 **(2)**에서 inner 함수의 실행 컨텍스트가 콜 스택의 가장 위에 담기면 outer 컨텍스트와 관련된 코드의 실행을 중단하고 inner 함수 내부의 코드를 순서대로 진행한다.
5. inner 함수 내부에서 a 변수에 값 3을 할당하고 나면 inner 함수의 실행이 종료되면서 inner 실행 컨텍스트가 콜 스택에서 제거된다.
6. (2)의 다음 줄부터 이어서 실행하고 a 변수의 값을 출력하고 나면 outer 실행 컨텍스트가 콜 스택에서 제거되어 전역 컨텍스트만 남아 있게 된다.
7. (3)의 다음 줄부터 이어서 실행하고 a 변수의 값을 출력하고 나면 전역 공간에 더이상 실행할 코드가 남아 있지 않아 전역 컨텍스트도 제거되고, 콜 스택에는 아무것도 남지 않은 상태로 종료된다.

- 어떤 실행 컨텍스트가 활성화될 때 자바스크립트 엔진은 해당 컨텍스트에 관련된 코드들을 실행하는 데 필요한 환경 정보들을 수집해서 실행 컨텍스트 객체에 저장한다.
  - **VariableEnvironment**: 현재 컨텍스트 내의 식별자들에 대한 정보 + 외부 환경 정보
    선언 시점의 LexicalEnvironment의 **스냅샷(snapshot)**으로 변경 사항은 반영되지 않음
  - **LexicalEnvironment**: 처음에는 VariableEnvironment와 같지만 변경 사항이 실시간으로 반영됨
  - **ThisBinding**: this 식별자가 바라봐야 할 대상 객체

스냅샷(snapshot)
사진을 찍듯이 특정 시점에 데이터 저장 장치(스토리지)의 파일 시스템을 포착해
별도의 파일이나 이미지로 저장, 보관하는 기술

2-2. VariableEnvironment

- VariableEnvironment에 담기는 내용은 LexicalEnvironment와 같지만 **최초 실행 시의 스냅샷을 유지**한다는 점이 다르다.
- 실행 컨텍스트를 생성할 때 VariableEnvironment에 정보를 먼저 담은 다음, 이를 그대로 복사해서 LexicalEnvironment를 만들고, **이후에는 LexicalEnvironment를 주로 활용**한다.
- VariableEnvironment와 LexicalEnvironment의 내부는 **environmentRecord와 outer-EnvironmentReference로 구성**돼 있다.
- 초기화 과정 중에는 완전히 동일하고 이후 코드 진행에 따라 서로 달라진다.

2-3. LexicalEnvironment

- lexical environment에 대한 한국어 번역은 '어휘적 환경', '정적 환경'이 가장 많은데, '어휘적'은 lexical을 영어사전에 대입해서 치환한 것으로 의미가 와 닿지 않고, '정적'은 수시로 변하는 환경 정보를 의미하는 lexical environment에 대한 적절한 번역이 아니다.
- **'사전적인'**이 더 어울리는 표현
- **컨텍스트를 구성하는 환경 정보들을 사전에서 접하는 느낌**으로 모아놓은 것

2-3-1. environmentRecord와 호이스팅

- environmentRecord에는 현재 컨텍스트와 관련된 코드의 식별자 정보들이 저장된다.

    - 컨텍스트를 구성하는 함수에 지정된 매개변수 식별자, 선언한 함수가 있을 경우 그 함수 자체, var로 선언된 변수의 식별자 등이 식별자에 해당된다.
- 컨텍스트 내부 전체를 처음부터 끝까지 쭉 훑어나가며 **순서대로** 수집한다.

> **참고**
> 전역 실행 컨텍스트는 변수 객체를 생성하는 대신 자바스크립트 구동 환경이 별도로 제공하는 객체, 즉, 전역 객체(global object)를 활용한다. 전역 객체에는 브라우저의 window, Node.js의 global 객체 등이 있다. 이는 자바스크립트 내장 객체(native object)가 아닌 호스트 객체(host object)로 분류된다.

호이스팅 규칙

> **호이스팅(hoisting)**
> 변수의 선언과 초기화를 분리한 후, **선언(식별자)만 코드의 최상단으로 끌어올린 것**으로 간주하는 가상의 개념 (실제로 끌어올리지는 않음)
> 인터프리터가 **변수와 함수의 메모리 공간을 선언 전에 미리 할당하는 것**을 의미한다.
> **var**로 선언한 변수의 경우 호이스팅 시 **undefined로 변수를 초기화**한다.
> 반면 **let과 const**로 선언한 변수의 경우 호이스팅 시 **변수를 초기화하지 않는다.**

매개변수와 변수에 대한 호이스팅 - 원본 코드

```
function a (x) {    // 수집 대상 1(매개변수)
  console.log(x);   // (1)
  var x;            // 수집 대상 2(변수 선언)
  console.log(x);   // (2)
  var x = 2;        // 수집 대상 3(변수 선언)
  console.log(x);   // (3)
}
a(1);
```

위의 코드는 다음과 같이 해석될 수 있다.

매개변수를 변수 선언/할당과 같다고 간주해서 변환한 상태

```
function a () {
  var x = 1;        // 수집 대상 1(매개변수 선언)
  console.log(x);   // (1)
  var x;            // 수집 대상 2(변수 선언)
  console.log(x);   // (2)
  var x = 2;        // 수집 대상 3(변수 선언)
  console.log(x);   // (3)
}
a();
```

- environmentRecord는 **현재 실행될 컨텍스트의 대상 코드 내에 어떤 식별자들이 있는지에만 관심**이 있고, 각 식별자에 어떤 값이 할당될 것인지는 관심이 없다.
- 따라서 변수를 호이스팅할 때 변수명만 끌어올리고 할당 과정은 원래 자리에 그대로 남겨둔다.

매개변수와 변수에 대한 호이스팅 - 호이스팅을 마친 상태

```
1  function a () {
2    var x;           // 수집 대상 1의 변수 선언 부분
3    var x;           // 수집 대상 2의 변수 선언 부분
```

```
4    var x;            // 수집 대상 3의 변수 선언 부분
5
6    x = 1;            // 수집 대상 1의 할당 부분
7    console.log(x);   // (1)
8    console.log(x);   // (2)
9    x = 2;            // 수집 대상 3의 할당 부분
10   console.log(x);   // (3)
11 }
12 a();
```

주소	…	1002	1003	1004	1005	…
데이터			이름: x 값: @5004			
주소	…	5002	5003	5004	5005	…
데이터				1		

- 2번째 줄: 변수 x를 선언한다. 이때 메모리에서는 저장할 공간을 미리 확보하고, 확보한 공간의 주솟값을 변수 x에 연결해둔다.
- 3번째 줄과 4번째 줄: 다시 변수 x를 선언한다. 이미 선언된 변수 x가 있으므로 무시한다.
- 6번째 줄: x에 1을 할당하라고 한다. 우선 숫자 1을 별도의 메모리에 담고, x와 연결된 메모리 공간에 숫자 1을 가리키는 주솟값을 입력한다.
- 7번째 줄과 8번째 줄: 각 x를 출력하라고 한다. **(1), (2) 모두 1이 출력**된다.

주소	…	1002	1003	1004	1005	…
데이터			이름: x 값: @5005			
주소	…	5002	5003	5004	5005	…
데이터				1	2	

- 9번째 줄: x에 2를 할당하라고 한다. 숫자 2를 별도의 메모리에 담고, 그 주솟값을 든 채로 x와 연결된 메모리 공간으로 간다. 숫자 1을 가리키는 주솟값이 들어있었는데, 이것을 2의 주솟값으로 대치한다. 이제 변수 x는 숫자 2를 가리키게 된다.
- 10번째 줄: **(3)에서는 2가 출력**되고, 함수 내부의 모든 코드가 실행됐으므로 실행 컨텍스트가 콜 스택에서 제거된다.

함수 선언의 호이스팅 - 원본 코드

```
function a () {
  console.log(b);       // (1)
  var b = 'bbb';        // 수집 대상 1(변수 선언)
  console.log(b);       // (2)
  function b () { };    // 수집 대상2(함수 선언)
  console.log(b);       // (3)
}
a();
```

- a 함수를 실행하는 순간 a 함수의 실행 컨텍스트가 생성된다.
- 이때 변수명과 함수 선언의 정보를 위로 끌어올린다.
- 변수는 선언부와 할당부를 나누어 선언부만 끌어올리는 반면 함수 선언은 함수 전체를 끌어올린다.

함수 선언의 호이스팅 - 호이스팅을 마친 상태

```
function a () {
  var b;                // 수집 대상 1. 변수는 선언부만 끌어올린다.
  function b() { };     // 수집 대상 2. 함수 선언은 전체를 끌어올린다.

  console.log(b);       // (1)
  b = 'bbb';            // 변수의 할당부는 원래 자리에 남겨둔다.
  console.log(b);       // (2)
  console.log(b);       // (3)
}
a();
```

- 호이스팅이 끝난 상태에서의 함수 선언문은 함수명으로 선언한 변수에 함수를 할당한 것처럼 여길 수 있다.

함수 선언문을 함수 표현식으로 바꾼 코드

```
function a () {
  var b;
  var b = function () { };  // ← 바뀐 부분

  console.log(b);           // (1)
  b = 'bbb';
  console.log(b);           // (2)
  console.log(b);           // (3)
}
a();
```

- 2번째 줄: 변수 b를 선언한다. 이때 메모리에서는 저장할 공간을 미리 확보하고, 확보한 공간의 주솟값을 변수 b에 연결해둔다.
- 3번째 줄: 이미 선언된 변수 b가 있으므로 선언 과정은 무시한다. 함수는 별도의 메모리에 담기고, 그 함수가 저장된 주솟값을 b와 연결된 공간에 저장한다. 이제 변수 b는 함수를 가리키게 된다.
- 5번째 줄: 변수 b에 할당된 **함수 b를 출력한다.(1)**
- 6번째 줄: 변수 b에 'bbb'를 할당하라고 한다. b와 연결된 메모리 공간에는 함수가 저장된 주솟값이 담겨있었는데 이걸 문자열 'bbb'가 담긴 주솟값으로 대치한다. 이제 변수 b는 문자열 'bbb'를 가리킨다.
- 7번째 줄과 8번째 줄: **(2)와 (3) 모두 'bbb'가 출력**된다. 함수 내부의 모든 코드가 실행됐으므로 실행 컨텍스트가 콜 스택에서 제거된다.

함수 선언문과 함수 표현식

- **함수 선언문(function declaration)**
  - function 정의부만 존재하고 별도의 할당 명령이 없는 것을 의미한다.
  - 반드시 함수명이 정의되어 있어야 한다.
- **함수 표현식(function expression)**
  - 정의한 function을 별도의 변수에 할당하는 것을 말한다.
  - 함수명이 없어도 된다.
  - 함수명을 정의한 함수 표현식을 **'기명 함수 표현식'**, 정의하지 않은 것을 **'익명 함수 표현식'**이라고 부르는데 **일반적으로 함수 표현식은 익명 함수 표현식**을 말한다.

함수를 정의하는 세 가지 방식

```
function a () { /* ... */ }       // 함수 선언문, 함수명 a가 곧 변수명
a(); // 실행 OK

var b = function () { /* ... */ } // (익명) 함수 표현식, 변수명 b가 곧 함수명
b(); // 실행 OK

var c = function d () { /* ... */ } // 기명 함수 표현식 변수명은 c, 함수명은 d
c(); // 실행 OK
d(); // Error
```

> **주의사항**
> 기명 함수 표현식은 외부에서는 함수명으로 함수를 호출할 수 없다.
> 함수명은 오직 함수 내부에서만 접근할 수 있다.

함수 선언문과 함수 표현식 - 원본 코드

```
console.log(sum(1, 2));
console.log(multiply(3, 4));

function sum (a, b) {               // 함수 선언문 sum
  return a + b;
}

var multiply = function (a, b) {    // 함수 표현식 multiply
  return a * b;
}
```

함수 선언문과 함수 표현식 - 호이스팅을 마친 상태

```
var sum = function sum (a, b) { // 함수 선언문은 전체를 호이스팅한다.
  return a + b;
};
var multiply;                   // 변수는 선언부만 끌어올린다.

console.log(sum(1, 2));
console.log(multiply(3, 4));

multiply = function (a, b) { // 변수의 할당부는 원래 자리에 남겨둔다.
  return a * b;
};
```

- 함수 선언문은 전체를 호이스팅한 반면 함수 표현식은 변수 선언부만 호이스팅했다.
- 함수도 하나의 값으로 취급할 수 있다.
- **함수를 다른 변수에 값으로써 '할당'**한 것이 함수 표현식이다.

- 1번째 줄: 메모리 공간을 확보하고 확보된 공간의 주솟값을 변수 sum에 연결한다.
- 4번째 줄: 또 다른 메모리 공간을 확보하고 그 공간의 주솟값을 변수 multiply에 연결한다.
- 1번째 줄(다시): sum 함수를 또 다른 메모리 공간에 저장하고, 그 주솟값을 앞서 선언한 변수 sum의 공간에 할당한다. 이로써 변수 sum은 함수 sum을 바라보는 상태가 된다.
- 5번째 줄: sum을 실행한다. 정상적으로 실행되어 3이 출력된다.
- 6번째 줄: 현재 multiply에는 값이 할당되어 있지 않다. 비어있는 대상을 함수로 여겨 실행하라고 명령한 것이다. 따라서 'multiply is not a function'이라는 에러 메시지가 출력된다. 뒤의 8번째 줄은 6번째 줄의 에러로 인해 실행되지 않은 채 런타임이 종료된다.

2-3-2. 스코프, 스코프 체인, outerEnvironmentReference

- **스코프 (scope)**
  식별자에 대한 유효범위

> ES5까지의 자바스크립트는 특이하게도 전역공간을 제외하면 **오직 함수에 의해서만 스코프가 생성**된다. ES6부터는 블록에 의해서도 스코프 경계가 발생하게 함으로써 다른 언어와 훨씬 비슷해졌다. 이러한 블록은 var로 선언한 변수에 대해서는 작용하지 않고, 새로 생긴 let, const, class, strict mode에서의 함수 선언 등에 대해서만 범위로서의 역할을 수행한다. 둘을 구분하기 위해 **함수 스코프, 블록 스코프**라는 용어를 사용한다.

- **스코프 체인(scope chain)**
  '식별자의 유효범위'를 안에서부터 바깥으로 차례로 검색해나가는 것
  이를 가능하게 하는 것이 LexicalEnvironment의 두 번째 수집 자료인 outerEnvironmentReference

스코프 체인

- outerEnvironmentReference는 **현재 호출된 함수가 선언될 당시**의 LexicalEnvironment를 참조한다.
- '선언하다'라는 행위가 실제로 일어날 수 있는 시점은 콜 스택 상에서 어떤 실행 컨텍스트가 활성화된 상태일 뿐이다.
- 어떤 함수를 선언(정의)하는 행위 자체도 하나의 코드이고, 모든 코드는 실행 컨텍스트가 활성화 상태일 때 실행되기 때문이다.

> **예시)**
> A 함수 내부에 B 함수를 선언하고 다시 B 함수 내부에 C 함수를 선언한 경우, 함수 C의 outerEnvironmentReference는 함수 B의 LexicalEnvironment를 참조한다. 함수 B의 LexicalEnvironment에 있는 outerEnvironmentReference는 다시 함수 B가 선언되던 때(A)의 LexicalEnvironment를 참조한다.

- outerEnvironmentReference는 **연결리스트(linked list) 형태**를 띤다.
- '선언 시점의 LexicalEnvironment'를 계속 찾아 올라가면 **마지막엔 전역 컨텍스트의 LexicalEnvironment**가 있다.
- 각 outerEnvironmentReference는 오직 자신이 선언된 시점의 LexicalEnvironment만 참조하고 있으므로 가장 가까운 요소부터 차례대로만 접근할 수 있고 **다른 순서로 접근하는 것은 불가능**하다.
- 이런 구조적 특성 덕분에 여러 스코프에서 동일한 식별자를 선언한 경우에는 **무조건 스코프 체인 상에서 가장 먼저 발견된 식별자에만 접근 가능**하게 된다.

```
1  var a = 1;
2  var outer = function () {
3   var inner = function () {
4     console.log(a);
5     var a = 3;
6   };
7   inner();
8   console.log(a);
9 };
10 outer();
11 console.log(a);
```

<table>
<tr><td colspan="3">[0] 전역 컨텍스트 활성화 – LexicalEnvironment, VariableEnvironment, thisBinding</td></tr>
<tr><td rowspan="8">전역 컨텍스트
<table><tr><td colspan="2">L.E</td></tr><tr><td>e</td><td>a, outer</td></tr></table>(this: 전역 객체)</td><td colspan="2">[1], [2] a에 1, outer에 함수 할당</td></tr>
<tr><td colspan="2">[10] outer 함수 호출, 전역 컨텍스트 비활성화
[2] outer 실행 컨텍스트 활성화</td></tr>
<tr><td rowspan="5">outer 컨텍스트
<table><tr><td colspan="2">L.E</td></tr><tr><td>e</td><td>inner</td></tr><tr><td>o</td><td>GLOBAL L.E</td></tr></table>(this: 전역 객체)</td><td colspan="2">[3] inner에 함수 할당</td></tr>
<tr><td colspan="2">[7] inner 함수 호출, 함수 outer 실행 컨텍스트 비활성화
[3] inner 실행 컨텍스트 재활성화</td></tr>
<tr><td>inner 컨텍스트
<table><tr><td colspan="2">L.E</td></tr><tr><td>e</td><td>a</td></tr><tr><td>o</td><td>outer L.E</td></tr></table>(this: 전역 객체)</td><td>[4] inner의 L.E에서 a 탐색
→ undefined 출력

[5] a에 3 할당</td></tr>
<tr><td colspan="2">[6] inner 함수 종료, inner 실행 컨텍스트 제거
[7] outer 실행 컨텍스트 재활성화</td></tr>
<tr><td colspan="2">[8] outer의 L.E에서 a 탐색
→ GLOBAL의 L.E에서 a 탐색 → 1 출력</td></tr>
<tr><td colspan="2">[9] outer 함수 종료, outer 실행 컨텍스트 제거
[10] 전역 컨텍스트 재활성화</td></tr>
<tr><td colspan="2">[11] GLOBAL의 L.E에서 a 탐색 → 1 출력</td></tr>
<tr><td colspan="3">[ ] 전체 종료, 전역 컨텍스트 제거</td></tr>
</table>

- 전체 윤곽을 왼쪽에서 오른쪽으로 바라보면 '전역 컨텍스트 → outer 컨텍스트 → inner 컨텍스트' 순으로 점차 규모가 작아지는 반면 스코프 체인을 타고 접근 가능한 변수의 수는 늘어난다.

변수 은닉화(variable shadowing)

- inner 함수 내부에서 a에 접근하려고 하면 스코프 체인 상의 첫 번째 인자인 inner 스코프의 LexicalEnvironment부터 검색한다.
- a 식별자가 존재하므로 스코프 체인 검색을 더 진행하지 않고 즉시 inner LexicalEnvironment 상의 a를 반환한다.
- 즉, inner 함수 내부에서 a 변수를 선언했기 때문에 전역 공간에서 선언한 동일한 이름의 a 변수에는 접근할 수 없다.

전역변수와 지역변수

- 전역변수(global variable): 전역 공간에서 선언한 변수
- 지역변수(local variable): 함수나 블록 내부에서 선언한 변수

2-4. this

- 실행 컨텍스트의 thisBinding에는 this로 지정된 객체가 저장된다.
- 함수를 호출하는 방법에 따라 this에 저장되는 대상이 다르다.
- 실행 컨텍스트 활성화 당시에 this가 지정되지 않은 경우 this에는 전역 객체가 저장된다.