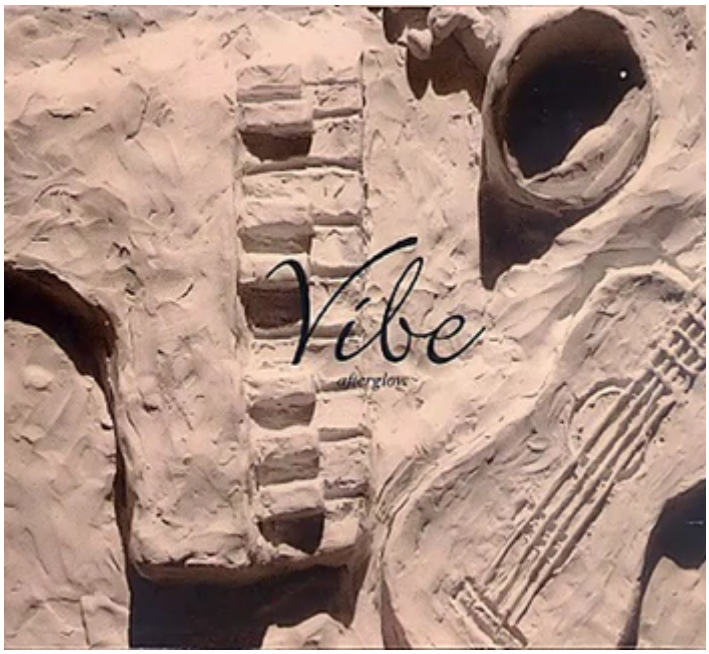
JavaScript 비동기 처리를 위한 promise 이해하기

cyranocoding · 2019년 8월 2일

JavaScript asynchronous callback promise 비동기

dev foundation

목록 보기 5/5



배경지식

JavaScript는 엔진은 **Single Thread**이다. 그래서 동시에 두 가지 작업을 할 수 없다. 그렇다면 여러 작업이 동시에 요청이 될 때 이 전 작업이 마무리 될 때까지 기다려야 하는가? 그렇다.

그래서 JavaScript 엔진은 비동기 처리가 가능하도록 설계되었다.

비동기(Asynchronous)란?

동기(Synchronous)적 처리는 작업을 요청함과 동시에 작업의 결과를 그 자리에서 받을 수 있는 데이터 처리 방식이다. 그 반대로 비동기(Asynchronous)적 처리는 작업을 요청하지만 결과는 그 자리에서 꼭 받지 않아도 되는 데이터 처리 방식이다.

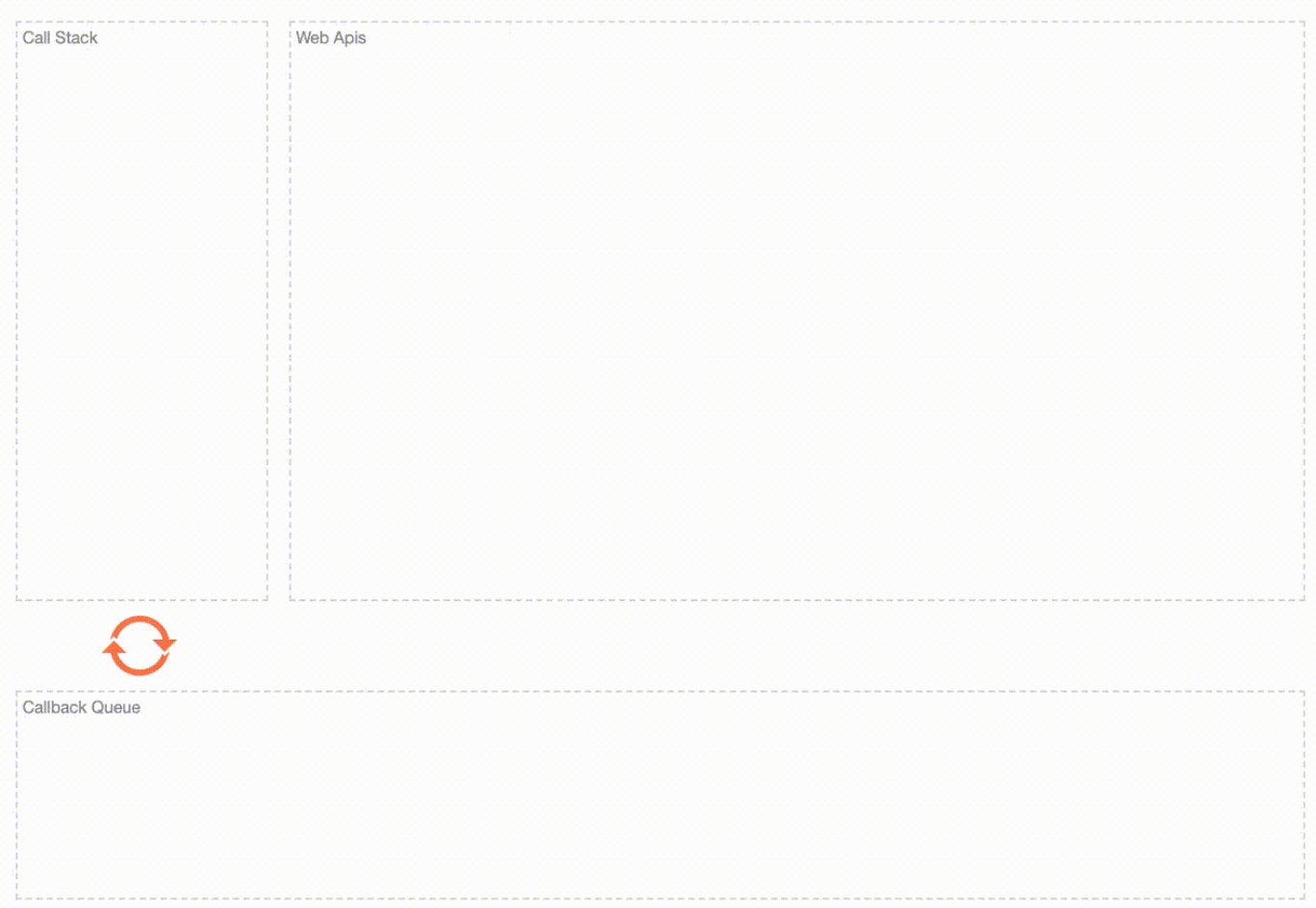
> 카페에서 커피를 주문할 때, 앞사람이 주문을 하고 주문한 커피를 다 제공한 다음, 다음 사람의 주문을 받는다면 **동기적 처리**라고 볼 수 있다. 반대로 모든 사람의 주문을 한꺼번에 받고 커피가 완성되는 대로 사람들에게 커피를 제공한다면 **비동기적 처리**이다.

다음은 아래의 코드를 처리하는 JavaScript 엔진의 모습을 도식화 한 것이다. Event Loop이 작업의 처리를 관리하고 있는데, 이에 관해서는 자세히 이야기할 기회가 있을 것이다.

```
console.log("first");

setTimeout(function setTimeout() {
    console.log("async");
}, 1000);

console.log("second");
```



setTimeout은 몇 초 뒤에 실행하고자 하는 function을 미리 설정하는 메소드이다. 요청한 즉시 결과를 얻어야 하는 것이 아니기 때문에 비동기적 처리의 대표적인 예이다.

위의 코드를 V8(JavaScript Engine) 다음의 순서로 처리하고 있다.

```
1) 첫번째 'first'를 콘솔에 log한다.

2) 1초 뒤에 'async'를 콘솔에 log하는 함수를 실행하기로 약속한다.

3) 약속한 함수는 Web API에서 기억하고 있다.

4) 'second'를 콘솔에 log한다.

… 1초 뒤

5) Web API는 Callback Queue에 'async'를 콘솔에 log하는 함수를 보낸다.

6) EventLoop은 Callback Queue에 들어온 순서대로 Call Stack으로 보내 실행이 되도록 한다.
```

간단 용어 설명

Call Stack : 작업이 바로 실행되고 다 실행되면 작업이 빠지는 공간이다.

CallBack Queue : 비동기 작업이 대기하고 있다가 Call Stack의 작업이 비어있으면 Event Loop의 명령에 따라 차례로 Call Stack으로 들어가는 공간이다.(비동기 작업들이 줄서는 곳)

EventLoop : 비동기/동기 작업의 순서를 관리하는 역할

Promise

JavaScript에서 Promise는 **비동기적으로 실행하는 작업의 결과(성공 or 실패)**를 나타내는 **객체**이다. 여기서 주목해야 하는 점은 `객체` 라는 것인데, 비동기의 결과를 객체화 시킨다는 점이 Promise의 가장 큰 장점이자 특징이 된다.

Promise 생성자

생성자 함수와 동일하게 new로 Promise 객체를 만들 수 있다. 이 때 인자로는 Executor가 들어가는 데 `Executor` 는 `resolve` 와 `reject` 라는 두 개의 함수를 매개변수로 받는 실행함수이다. `Executor` 는 비동기 작업을 시작하고 모든 작업을 끝낸 후, 해당 작업이 성공적으로 이행이 되었으면 `resolve` 함수를 호출하고, 중간에 오류가 발생한 경우 `reject` 함수를 호출한다.

간단한 예제 :

1초 뒤에 랜덤한 숫자가 5이상이면 성공하고, 5보다 작으면 실패하는 경우. Promise 객체 생성 방법

```
var timeAttack = new Promise(function (resolve, reject) {
  setTimeout(function () {
    var ran = Math.random() * 10;
    if (ran >= 5) {
      resolve(ran);
    } else {
      reject();
    }
  }, 1000);
});
```

이 경우에 `timeAttack` 이라는 Promise 객체는 3가지 상태를 가진다.

- 대기(pending) : 아직 실행되지 않은 초기 상태
- 이행(fulfilled) : 작업이 성공적으로 완료됨.
- 거부(rejected) : 작업이 실패함.

작업이 성공적으로 이행이 되었거나, 실패 했을 때, 어떠한 작업을 해야 하는데 이 작업은 then 메소드에 의해 실행된다. 이는 callback함수를 실행한 것과 같은 효과를 낸다. **then 메소드는 promise객체에 붙여서 사용한다.**

```
timeAttack.then(function (num) {
  console.log(num + 'complete');
}, function () {
  console.log('error');
});
```

`promise.then(successCallback, failureCallback)` 이러한 방식으로 콜백을 실행할 수 있다.

then() Method

then 메소드는 **promise객체를 리턴**하고 두 개의 콜백 함수를 인수로 받는다. 사용 형태는 다음과 같다.

```
promise.then(successCallback, failureCallback)

promise.then(function (value) {
  //성공했을 때 실행
}, function (reason) {
  //실패했을 때 실행
});
```

위에 적은 `successCallback` 은 promise가 **성공(fulfilled)했을 때를 위한 콜백 함수**이고, `failureCallback` 은 **실패(rejected)했을 때를 위한 콜백 함수**이다.

chaining

then 메소드는 promise 객체를 리턴하고 인수로 받은 콜백 함수들의 리턴 값을 이어 받는다. 따라서 chaining이 가능하다. 아래의 예제처럼.

```
var promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve(1);
  }, 1000);
});

promise.then(function (num) {
  console.log(num + 'complete'); /// 1complete
  return num + 1; /// return = 2
}).then(function (value) {
  console.log(value) // 2
});
```

resolve에 인자로 들어간 숫자 1이 첫번째 then() 메소드를 거치면서 +1이 되고 두번째 then() 메소드에서는 2가 되었다. 이처럼 promise를 리턴하는 then의 특성으로 계속해서 체이닝 패턴이 사용 가능하고, 값을 조작할 수 있다.

Callback Hell

Promise 이전의 비동기 처리는 Callback함수를 설정하는 방식으로 이루어졌다. 비동기가 완료되는 시점에 실행이 되는 callback함수로 완료를 인지하고 그 다음을 처리한 것이다. 그렇게 하다보니 비동기 처리를 연속적으로 해야하는 경우 Callback함수에 Callback함수가 들어가고 거기에 또 Callback함수가 들어가는 이상하고 요상한 코드가 되어 버렸다. 이것을 이른바 **Callback Hell 이라고 한다.**

```
a(function (resultA) {
  b(resultA, function (resultB) {
    c(resultB, function (resultC) {
      d(resultC, function (resultD) {
        //OMG.....
      });
    });
  });
});
```

Promise를 이용하여 이러한 Callback Hell을 말끔히 탈출할 수 있는 것은 아니지만 Callback을 함수로 바로 넘겨받지 않고 객체에 이어서 사용할 수 있게 되면서 훨씬 보기 쉬워졌다.

```
promise.then(function(a){

}).then(function(b){

}).then(function(c){

}).then(function(d){

});
```

promise는 비동기 처리에 있어서 객체의 개념을 도입했다는 점이 가장 큰 특징이다. 그래서 자바스크립트에서 객체가 사용할 수 있는 메소드를 활용할 수 있는데, (like map...?) 본 포스팅에서는 생략하였다. 다음에 기회가 된다면 포스팅할수도... 사실 promise를 활용하여 hacker-news 페이지 따라하기를 포스팅 주제로 생각했었는데, 이 것도 나중에 기회가 된다면 하도록 하겠다.

비동기 처리에서 많이 사용하는 Async/Await에 대한 내용도 좋은 주제가 될 것 같다. 공부할게 많네..

By Cyrano on Aug 2, 2019.

박한준

개발자로 한걸음 한걸음 가고 있어요.

이전 포스트

JavaScript에서의 OOP, Inheritance와 Prototype Chain과 Class 에 대한 개념 정...

0개의 댓글

댓글을 작성하세요

댓글 작성