async await :: Javascript 의 비동기 동작 원리

#TIL/dev/language/javascript

Javascript는 기본적으로 비동기로 동작한다. 그로 인해 각각의 함수 실행문은 자바스크립트를 실행시키는 엔진의 Queue에 쌓인다. await 등과 같이 명시적으로 동기화하지 않으면 그냥 바로 다음 함수도 실행 Queue에 던져지게 됨.

이는 굉장히 특이한데, 예시를 보면서 살펴보자.

```
onRequest: async () => {
  await ~~~~~~
}

const request = async (): Promise<void> => {
    setIsLoading(true);
    try {
      onRequest();
      onSuccess();
    } catch (e) {
      onFail(e);
    } finally {
      setIsLoading(false);
    }
  };
```

이와 같은 상황에서 처음에 나는 Error try - catch 와 같이 Promise Handling이 이루어지지않을까 생각했는데 아니었다. 주어진 콜백 내부에서 await를 통한 동기로직이 있더라도, 그것은 그 함수 scope 내부에서만 유효하고, 외부로는 전혀 영향을 미치지않는다. 그 이유는 아마 기본적으로 비동기적으로 동작하는 구현원리때문인듯.

아무튼 위와 같은 상황에서 onRequest에는 await와 같은 동기화 로직이 없기 때문에 onRequest를 Message Queue에 Dispatch한 후 onSuccess ➜ Finally로 바로 Queue에 dispatch하는 것.

++

자바스크립트의 동작은 시스템 스레드를 이용한다고 한다. 브라우저나 노드 엔진 등. 자세한 건 나중에 찾아

보면 좋을 듯.