

Hanbit
RealTime
121



You Don't Know JS

비동기와 성능

카일 심슨 지음 / 이일웅 옮김



O'REILLY

한빛미디어
Hanbit Media, Inc.

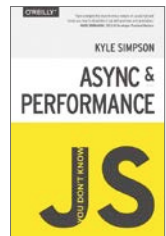


You Don't Know JS

비동기와 성능

카일 심슨 지음 / 이일웅 옮김

이 도서는
ASYNC & PERFORMANCE(O'REILLY)의
번역서입니다



You Don't Know JS 비동기와 성능

초판발행 2015년 12월 18일

지은이 카일 심슨 / **옮김이** 이일웅 / **펴낸이** 김태헌

펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

전화 02-325-5544 / **팩스** 02-336-7124

등록 1999년 6월 24일 제10-1779호

ISBN 978-89-6848-794-1 13000 / **정가** 19,000원

총괄 전태호 / **책임편집** 김창수 / **기획·편집** 김상민

디자인 표지/내지 여동일, 조판 최송실

마케팅 박상용, 송경석 / **영업** 김형진, 김진불, 조유미

이 책에 대한 의견이나 오타자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주세요.

한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

© 2015 Hanbit Media, Inc.

Authorized Korean translation of the English edition of You Don't Know JS: Async & Performance, ISBN 9781491904220 © 2015 Getify Solutions, Inc.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

이 책의 저작권은 오라일리과 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.

1. 국어 표준 맞춤법, 외래어 표기법 및 띄어쓰기 규정을 준수한다.
2. 기술 용어, 제품명 등 고유 명사 형태의 원어는 최대한 한글로 음차하여 옮긴다(예: JavaScript → 자바스크립트). 약자 형태로 축약된 원어나 그 밖에 한글 음차로 옮기는 것보다 원어 그대로 표기하는 편이 독자의 이해에 도움이 된다면 원어를 표기한다(예: PK).
3. 2번에서 한글 음차 시 최초 1회 영문을 위 첨자 형태로 함께 적고, 이후 반복하여 등장할 경우 이를 생략한다. 그러나 이렇게 함께 적은 용어가 다시 나올 경우에도 독자의 이해를 위해 필요할 경우 다시 영문 병기를 한다.
4. 『You Don't Know JS』 시리즈 도서는 대체로 일상생활에서 대화를 나누는 듯한 구어적인 표현이 많은데, 이러한 특성을 한글 번역본에도 최대한 반영하려고 노력하였다.
5. 이미 업계나 기술자들 사이에서 많이 사용되어 외래어처럼 굳어진 용어는, 어차피 이 도서의 대상 독자가 일반인이 아니기 때문에, 굳이 우리말로 번역하지 않고 원어를 그대로 음차한다(예: type → 형 타입, copy and paste → 복사 후 붙여넣기 카피 앤 페이스트). 그리고 저자가 즐겨 쓰는 일부 비표준 용어(예: coercion)는 가장 가까운 한글 용어로 번역하여 그 의미를 최대한 반영하고(예: coercion → 강제변환), 독자가 혼동할 우려가 있는 경우 각주에서 그 이유를 밝힌다.
6. 저자가 기술한 원문이 직역 시 이해가 어렵다고 판단하면 문장 구조를 재배열하거나 관련 문구를 추가하는 식으로 의역을 병행한다. 필요하다면 변환 수준의 번역을 일부 적용한다.
7. 예제 코드의 주석 및 기술적인 내용과는 무관한 상수 문자열은 한글 번역을 하되, 프로그램 로직을 파악하는 데 오히려 방해되거나 코드 가독성을 떨어뜨릴 수 있는 경우 원래의 코드를 유지한다.

순차적/동기적으로 실행되는 프로그램은 사람이 CPU가 되어 코드 진행을 그대로 쫓아가면 되므로 그리 어렵지 않습니다. 사용하는 언어의 문법/구문을 학습하고 눈에 익도록 조금만 훈련을 하고 나면 누구나 실행 흐름을 따라 구현 로직을 분석할 수 있습니다. 반면에 비동기 프로그래밍은 일단 머릿속에 여러 가닥으로 나뉜 실타래를 떠올려야 하는 데다 코드에 기술한 순서와 실제 실행 순서에 엇박자가 생기는 등 처음에는 프로그래머의 상식을 벗어나는 문제들이 많아 편두통을 앓기에 십상입니다. 하지만 기본 원리와 핵심 개념을 이해하고 시대를 앞서간 선배 프로그래머들이 비동기성을 프로그램으로 나타내기 위해 어떤 노력을 했는지 차근차근 알아가면 비동기 프로그래밍의 매력에 흠뻑 빠져들게 될 겁니다.

사실 비동기 프로그래밍의 역사는 꽤 오래된 편이지만 웹 개발 세상에서, 특히 우리나라에서는 2000년대 중후반, '웹 2.0'이란 개념과 함께 AJAX 사용이 널리 보급되면서 많은 주목을 받게 된 것 같습니다. 저 역시 당시 비표준 기술이었지만 인터넷 익스플로러에서 ActiveXObject, XMLHttpRequest 같은 객체를 직접 만지작거리며 서버와 비동기 통신을 수행하는 코드를 작성했던 기억이 납니다. 자바스크립트로 비동기 프로그래밍을 한다고 하면 AJAX 요청을 하여 콜백 함수로 받는 것 이외에 달리 생각할 만한 범용 기술이 없었고, 그나마 그런 코드라도 다룰 줄 아는 개발자를 많이 찾던 시절이 있었지요.

『You Don't Know JS』 시리즈의 결정판이라 할 수 있는 『비동기와 성능(Async & Performance)』은, 현대 자바스크립트가 특히 비동기 스크립트 기술이 그간 얼마나 눈부시게 발전하여 ECMAScript 표준이 되었는지 알아보고, 다음 표준 명세에 등장하게 될 새로운 기술과 흥미로운 주제들까지 한 권에 읽어볼 수 있는 충실한 도서입니다. 아직 프라미스, 제너레이터 등 ES6 이후 등장한 콜백 대체 기술들에 관한 체계적인 안내서가 절대 부족한 상황에서 이 책은 고급 자바스크립트 개발자들의 지적 갈증을 해소하고 실무에서 여러 가지 기술을 검토하시는 분들께 올바른 방향을 제시하는 길라

잡이가 될 것입니다.

모쪼록 부족한 번역이나 많은 개발자 여러분들이 곁에 두고 자주 찾는 번역서가 되길 바라며, 본 시리즈 번역을 의뢰하신 한빛미디어 김창수 팀장님과 스마트미디어팀 여러분, 그리고 주말과 휴일 내내 많은 시간 함께 해주지 못한, 사랑하는 제 아내와 두 딸, 제이와 솔이에게 이 역서를 바칩니다. 언제나 아들에게 변함없는 믿음과 사랑을 보내주신 부모님께 늘 감사드립니다.

2015년, 아침 공기가 문득 쌀쌀해진 가을 무렵에

- 이일웅

저자 소개

지은이_ **카일 심슨** Kyle Simson



텍사스 오스틴 출신의 카일 심슨은 오픈 웹 전도사로, 자바스크립트, HTML5, 실시간 P2P 통신과 웹 성능에 누구 못지않은 열정을 갖고 있다. 안 그랬으면 이미 오래전에 질려버렸을 것이다. 저술가, 워크숍 강사, 기술 연사로, 그리고 오픈 소스 커뮤니티에서도 활약 중이다.

역자 소개

옮긴이_ **이일웅**



10년 넘게 국내, 미국 등지에서 대기업/공공기관 프로젝트를 수행한 웹 개발자이자, 두 딸의 사랑을 한몸에 받고 사는 행복한 딸바보다. 자바 기반의 서버 플랫폼 구축, 데이터 연계, 그리고 다양한 자바스크립트 프레임워크를 응용한 프론트엔드 화면 개발을 주로 담당해 왔다. 시간이 날 땐 피아노를 연주한다.

• 개인 홈페이지: <http://www.bullion.pe.kr>

수년간 내가 근무했던 회사 대표님은 나를 믿고 개발자 면접을 맡기셨다. 자바스크립트 개발자를 채용하는 면접관으로서 제일 먼저...(음, 생각해보니 그 전에 먼저 후보자가 화장실에 다녀왔거나 음료가 필요한지 물어봤다. 편안한 분위기에서 면접을 봐야 할 테니. 개인적인 용무를 마치고 어느 정도 분위기가 좋아졌다 싶으면) 후보자가 자바스크립트를 알고 있는지, 아니면 제이쿼리(jQuery)를 아는 것인지 분별하는 작업에 착수한다.

제이쿼리를 폼하하려는 뜻은 없다. 사실 제이쿼리는 자바스크립트를 잘 몰라도 많은 것들을 할 수 있게 도와주는 도구지, 분명 버그는 아닐 것이다. 하지만 자바스크립트의 성능/관리를 일임할 고급 기술자를 뽑는 포지션이라면, 제이쿼리 같은 라이브러리를 잘 융합시킬 줄 아는, 말하자면 제이쿼리 제작자 정도로 자바스크립트 핵심을 능수능란하게 다룰 수 있어야 한다.

자바스크립트 핵심 스킬을 갖춘 사람인지 구별하기 위해 나는 후보자가 클로저(closure)로 뭘 할 수 있는지(독자들은 이 시리즈 책을 읽었을 것이다), 그리고 이 책의 주제이기도 한 비동기성을 어떻게 최대한 활용할지 등을 물어본다.

초심자라도 비동기 프로그래밍의 전채 요리라 할 만한 콜백(callback) 정도는 알고 있을 것이다. 물론 전채 요리만으로는 만족스러운 식사가 될 수 없으니 산해진미로 가득한 다음 코스 프라미스(promise)로 바로 넘어가자!

프라미스가 처음이라면 지금이 공부할 좋은 타이밍이다. 이제 프라미스는 자바스크립트/DOM에서 비동기 반환값을 반환하는 표준이다. 앞으로 등장할 비동기 DOM API는 꼭 프라미스를 사용할 테니 미리 단단히 준비해두자! 이 추천사를 쓰고 있는 지금도 프라미스는 대부분 주요 브라우저에 탑재된 상태고 조만간 IE에도 실릴 예정이다. 프라미스 코스를 완주했다면 다음 코스인 제너레이터(generator)를 위해 위장을 비워두기 바란다.

제너레이터는 크롬/파이어폭스 안정 버전에 화려한 오픈 행사 없이 안착했다. 솔직히

제너레이터 자체의 흥미로움에 비해 다소 복잡하다 생각했었는데, 프라미스와 결합되는 모습을 지켜본 이후론 생각이 완전히 달라졌다. 가독성과 유지 보수성 측면에서 이제 제너레이터는 필수 도구다.

후식은, 음 이 책의 스포일러가 될 생각은 없지만, 자바스크립트의 미래를 바라볼 전망대에서 할 것이다! 동시성과 비동기성을 여러분 손바닥 안에 쥐락펴락하게 해 줄 막강한 기능들이 기다리고 있다.

자, 이 책을 구매한 여러분의 즐거움을 더 빼앗고 싶지 않다. 어서 책장을 넘기자! 이 추천사를 읽기 전에 이미 한 차례 책거리를 한 독자라면 흔쾌히 비동기 점수 10점을 주겠다! 마땅히 10점 받을 자격 있다!

제이크 아키발드 Jake Archibald

jakearchibald.com, [@jaffathecake](https://twitter.com/jaffathecake)

- 구글 크롬 개발자 애드보킷

이미 눈치챌겠지만, 본 시리즈 제목 일부인 “JS”는 자바스크립트를 폄하할 의도로 쓴 약어가 아니다. 물론 자바스크립트 언어에 숨겨진 기벽^{quirk}이 만인이 비난하는 대상임은 부인할 수 없겠지만!

웹 초창기부터 자바스크립트는 사람들이 대화하듯 웹 콘텐츠를 소비할 수 있게 해준 기반 기술이었다. 마우스 트레일을 깜빡이거나 팝업 알림창을 띄워야 할 수요에서 비롯되어, 20년 가까이 흐른 지금, 자바스크립트는 엄청난 규모로 기술적 역량이 성장하였고, 세계에서 가장 널리 사용되는 소프트웨어 플랫폼이라 불리는, 웹의 심장부를 형성하는 핵심 기술이 되었다.

그러나 프로그래밍 언어로서의 자바스크립트는 끊임없는 비난과 논란의 대상이기도 했는데, 부분적으로 과거로부터 전해 내려온 폐해 탓이기도 하지만 그보다 설계 철학 자체가 문제시되기도 했다. 브렌단 아이크^{Brendan Eich}⁰¹의 표현을 빌자면, ‘자바스크립트’란 이름 자체가 좀 더 성숙하고 나이 많은 형인 ‘자바’ 아래의 ‘바보 같은 꼬마 동생 dumb kid brother’같은 느낌을 준다. 하지만 이름은 정치와 마케팅 사정상 우연히 그렇게 붙여진 것일 뿐, 두 언어는 여러 중요한 부분에서 이질적이다. “자바스크립트”와 “자바”는 “카메라”와 “카^{car}”만큼이나 무관하다.

C 스타일의 절차 언어에서 미묘하며 불확실한 스킴^{Scheme}/리스프^{Lisp} 스타일의 함수형 언어에 이르기까지, 자바스크립트는 서너 개 언어로부터 근본 개념과 구문 체계를 빌려왔기 때문에 꽤 폭넓은 개발자층을 확보하는데 대단히 유리했고, 심지어 프로그래밍 경력이 별로 없는 사람들도 쉽게 배울 수 있었다. “Hello World”를 자바스크립트로 출력하는 코드는 너무 단순해서 출시 당시엔 나름의 매력이 있었고 금방 익숙해졌다.

자바스크립트는 처음 시작하고 실행하기는 가장 쉬운 언어지만, 독특한 기벽 탓에 다

01 역사주_자바스크립트의 창시자. 1995년 넷스케이프 근무 당시 열혈 만에 자바스크립트 언어를 고안했습니다.

큰 언어들에 비해 언어 자체를 완전히 익히고 섭렵한 달인은 찾아보기 매우 드문 편이다. C/C++ 등으로 전체 규모(full-scale)의 프로그램을 작성하려면 언어 자체를 깊이 있게 알고 있어야 가능하지만, 자바스크립트는 언어 전체의 능력 중 일부를 대략 수박 겉핥기 정도만 알고 사용해도 웬만큼 운영 서비스가 가능하다.

언어 깊숙이 뿌리를 내려 자리 잡은, 정교하고 복잡한 개념이 외려(물백 함수를 다른 함수에 인자로 넘기는 것처럼) 겉보기에 단순한 방식으로 사용해도 괜찮게끔 유도하고, 그러다 보니 자바스크립트 개발자는 내부에서 무슨 일들이 벌어지든, 있는 그대로의 언어 자체를 사용하여 개발할 수 있다.


그러나 간단하고 쓰기 쉬운 언어일수록 여러 가지 의미와 복잡하고 세밀한, 다양한 기법들이 결집되어 있기 때문에 꼼꼼하게 학습하지 않으면 제아무리 노련한 개발자 할 지라도 올바르게 이해하지 못한다.

이것이 바로 자바스크립트의 역설이자 아킬레스건이며, 이 책을 읽고 여러분이 넘어야 할 산이다. 다 알지 못해도 사용하는 데 문제가 없다 보니 끝내 자바스크립트를 제대로 이해하지 못하고 넘어가는 경우가 비일비재하다.

목표

자바스크립트의 놀랍거나 불만스런 점들을 마주할 때마다 자신의 블랙리스트에 추가하여 금기시한다면(이런 일에 익숙한 사람들이 더러 있다.), 자바스크립트란 풍성함의 빈 껍데기에 머무르게 될 것이다.

누군가 “좋은 부분(The Good Parts)”이란 유명한 별칭을 달아놓았는데, 부디 독자 여러분들! “좋은 부분”이라기보다는, 차라리 “쉬운 부분”, “안전한 부분”, 또는 “불완전한 부분”이라고 하는 편이 더 정확할 것이다.



『You Don't Know JS』 시리즈는 정반대의 방향으로 접근한다. 자바스크립트의 모든 것, 그 중 특히 “어려운 부분^{The Tough Part}”을 심층적으로 이해하고 학습할 것이다!

나는 자바스크립트 개발자들이 정확히 언어가 어떻게, 그리고 왜 그렇게 작동하는지 알려 하지 않고, “그냥 이 정도면 됐지.” 식으로 이해하고 대충 때우려는 자세를 직접 거론할 것이다. 험한 길을 마주한 상황에서 쉬운 길로 돌아가라는 식의 조언은 절대 하지 않으려 한다.

코드가 일단 잘 돌아가니 이유는 모른 채 그냥 지나치는 건 내 성질에 용납할 수 없다. 여러분도 그래야 한다. 여러분이 나와 함께 험난한 “가시밭길”을 탐험하면서 자바스크립트가 무엇인지, 자바스크립트로 뭘 할 수 있을지 포괄적으로 배우기 바란다. 이런 지식을 확실히 보유하고 있으면, 테크닉, 프레임워크, 금주의 인기 있는 머리글자 따위는 여러분 손바닥 위에서 벗어나지 않을 것이다.

본 시리즈는 자바스크립트에 대해 가장 흔히 오해하고 있거나 잘못 이해하고 있는, 특정한 핵심 언어 요소를 선정하여 아주 깊고 철저하게 파헤친다. 여러분은 이론적으로만 알고 넘어갈 것이 아니라, 실전적으로 “내가 알고 있어야 할” 내용을 분명히 다 알고 간다는 확신을 갖고 책장을 넘기기 바란다.

아마도 지금 여러분이 알고 있는 자바스크립트는 다른 사람들이 불완전한 이해로 구워낸 단편적인 지식을 물려받은 정도일 것이다. 이런 자바스크립트는 진정한 자바스크립트의 그림자에 불과하다. 여러분은 지금 자바스크립트를 제대로 모르지만 본 시리즈를 열독하면 완벽히 알게 될 것이다. 동료, 선후배 여러분들, 포기하지 말고 계속 읽기 바란다. 자바스크립트가 여러분의 두뇌를 기다리고 있다.

정리하기

자바스크립트는 굉장한 언어다. 적당히 아는 건 쉬워도 완전히(충분히) 다 알기는 어렵다. 헛갈리는 부분이 나오면 개발자들은 대부분 자신의 무지를 탓하기 전에 언어 자체를 비난하곤 한다. 본 시리즈는 이런 나쁜 습관을 바로잡고 이제라도 여러분이 자바스크립트를 제대로, 깊이 있게 이해할 수 있도록 도와주는 것을 목표로 한다.



이 책의 예제 코드를 실행하려면 현대적인 자바스크립트 엔진(예: ES6)이 필요하다. 구 엔진(ES6 이전)에서는 코드가 작동하지 않을 수 있다.

이 책의 표기법



팁, 제안은 여기에 적습니다.



일반적인 내용은 여기에 적습니다.



경고나 유의 사항은 여기에 적습니다.



예제 코드 내려받기

보조 자료(예제 코드, 연습 문제 등)는 <https://goo.gl/VR1WNv>에서 내려받을 수 있습니다.

한빛 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾기도 쉽지 않습니다. 또한, 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1 eBook First - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 좀 더 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한, 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

무료로 업데이트되는 전자책 전용 서비스입니다

2 종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3 독자의 편의를 위해 DRM-Free로 제공합니다

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 어려운 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리하고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유 권한을 표시한 문구가 없거나 타인의 소유권함을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 큼니다. 이 경우 저작권법에 따라 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한, 한빛미디어 사이트에서 구매하신 후에는 횡수에 관계없이 내려받을 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오타자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려 드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구매하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

chapter 1 비동기성: 지금과 나중 — 023

- 1.1 프로그램 덩이 — 024
 - 1.1.1 비동기 콘솔 — 027
- 1.2 이벤트 루프 — 028
- 1.3 병렬 스레딩 — 031
 - 1.3.1 완전-실행 — 034
- 1.4 동시성 — 037
 - 1.4.1 비상호작용 — 040
 - 1.4.2 상호작용 — 041
 - 1.4.3 협동 — 046
- 1.5 잡 — 049
- 1.6 문 순서 — 051
- 1.7 정리하기 — 055

chapter 2 콜백 — 057

- 2.1 연속성 — 058
- 2.2 두뇌는 순차적이다 — 059
 - 2.2.1 실행 vs 계획 — 061
 - 2.2.2 중첩/연쇄된 콜백 — 063
- 2.3 믿음성 문제 — 069
 - 2.3.1 다섯 마리 콜백 이야기 — 070
 - 2.3.2 남의 코드뿐만 아니라 — 073
- 2.4 콜백을 구하라 — 075
- 2.5 정리하기 — 081

3.1	프라미스란	084
3.1.1	미래값	084
3.1.2	완료 이벤트	091
3.2	데너블 덕 타이핑	097
3.3	프라미스 믿음	100
3.3.1	너무 빨리 호출	101
3.3.2	너무 늦게 호출	102
3.3.3	한번도 콜백을 안 호출	104
3.3.4	너무 가끔, 너무 종종 호출	105
3.3.5	파라미터/환경 전달 실패	106
3.3.6	에러/예외 삼키기	106
3.3.7	미더운 프라미스?	108
3.3.8	믿음 형성	112
3.4	연쇄 흐름	112
3.4.1	용어 정의: 귀결, 이룸, 버림	121
3.5	에러 처리	125
3.5.1	절망의 구덩이	128
3.5.2	잡히지 않은 에러 처리	130
3.5.3	성공의 구덩이	132
3.6	프라미스 패턴	134
3.6.1	Promise.all([..])	134
3.6.2	Promise.race([..])	136
3.6.3	all([..])/race([..])의 변형	140
3.6.4	동시 순회	141
3.7	프라미스 API 복습	143
3.7.1	new Promise(..) 생성자	143



3.7.2 Promise.resolve(..)와 Promise.reject(..)	144
3.7.3 then(..)과 catch(..)	145
3.7.4 Promise.all([..])과 Promise.race([..])	146
3.8 프라미스 한계	147
3.8.1 시퀀스 에러 처리	147
3.8.2 단일값	149
3.8.3 단일 귀결	153
3.8.4 타성	155
3.8.5 프라미스는 취소 불가	159
3.8.6 프라미스 성능	162
3.9 정리하기	164

chapter 4 제너레이터 — 165

4.1 완전-실행을 타파하다	165
4.1.1 입력과 출력	169
4.1.2 다중 이터레이터	173
4.2 값을 제너레이터링	179
4.2.1 제조기와 이터레이터	179
4.2.2 이터러블	183
4.2.3 제너레이터 이터레이터	185
4.3 제너레이터를 비동기적으로 순회	189
4.3.1 동기적 에러 처리	192
4.4 제너레이터 + 프라미스	195
4.4.1 프라미스-인식형 제너레이터 실행기	198
4.4.2 제너레이터에서의 프라미스 동시성	202
4.5 제너레이터 위임	207

4.5.1 왜 위임을?	210
4.5.2 메시지 위임	210
4.5.3 비동기성을 위임	216
4.5.4 위임 "재귀"	216
4.6 제너레이터 동시성	219
4.7 썬크	224
4.7.1 s/promise/thunk/	229
4.8 ES6 이전 제너레이터	233
4.8.1 수동 변환	234
4.8.2 자동 변환	240
4.9 정리하기	242

chapter 5 프로그램 성능 — 243

5.1 웹 워커	244
5.1.1 워커 환경	247
5.1.2 데이터 전송	248
5.1.3 공유 워커	249
5.1.4 웹 워커 폴리필	251
5.2 SIMD	252
5.3 asm.js	254
5.3.1 asm.js 최적화	255
5.3.2 asm.js 모듈	256
5.4 정리하기	259

chapter 6 **벤치마킹과 튜닝** — 261

6.1	벤치마킹	261
6.1.1	반복	263
6.1.2	Benchmark.js	264
6.2	컨텍스트가 제일	267
6.2.1	엔진 최적화	268
6.3	jsPerf.com	271
6.3.1	정상 테스트	272
6.4	좋은 테스트를 작성하려면	276
6.5	미시성능	277
6.5.1	똑같은 엔진은 없다	282
6.5.2	큰 그림	285
6.6	꼬리 호출 최적화(TCO)	288
6.7	정리하기	291

부록 A **asynquence 라이브러리** — 293

A.1	시퀀스, 추상화 설계	294
A.2	asynquence API	297
A.2.1	단계	297
A.2.2	예러	299
A.2.3	병렬 단계	303
A.2.4	시퀀스 분기	310
A.2.5	시퀀스 병합	311
A.3	값과 예러 시퀀스	312
A.4	프라이미스와 콜백	314

A.5	이터러블 시퀀스	316
A.6	제너레이터 실행하기	317
A.6.1	감싼 제너레이터	318
A.7	정리하기	319

부록 B 고급 비동기 패턴 — 321

B.1	이터러블 시퀀스	321
B.1.1	이터러블 시퀀스 확장	325
B.2	이벤트 반응형	330
B.2.1	ES7 옵저버블	332
B.2.2	반응형 시퀀스	333
B.3	제너레이터 코루틴	338
B.3.1	상태 기계	339
B.4	순차적 프로세스 통신(CSP)	343
B.4.1	메시지 전달	343
B.4.2	CSP 에뮬레이션	346
B.5	정리하기	349

비동기성: 지금과 나중

일정 시간 동안 발생하는 프로그램의 움직임을 어떻게 표현하고 나타낼 것인가? 자바스크립트 같은 프로그래밍 언어에서 가장 중요하면서도 많은 사람의 오해를 사는 문제다.

for 루프가 시작되어 끝날 때까지 (마이크로초에서 밀리초 단위 동안) 벌어지는 일들이 문제가 아니라, 프로그램의 어느 부분은 '지금' 실행되고 다른 부분은 '나중에' 실행되면서 발생하는, 프로그램이 실제로 작동하지 않는 '지금^{now}'과 '나중^{later}' 사이의 간극^{gap}에 관한 문제다.

이러한 간극의 유형은 사용자 입력 대기, 데이터베이스/파일 시스템의 정보 조회, 네트워크를 경유한 데이터 송신 후 응답 대기, 일정한 시간 동안 반복적인 작업^{task} 수행(예: 애니메이션) 등 아주 다양한데, 지금까지 개발된 많은 (자바스크립트) 프로그램들이 나름대로 이 간극을 메우기 위해 애써왔다. 어떤 방법을 동원하든 여러분이 작성한 프로그램도 시간 간극에 의한 상태 변화를 올바르게 다스릴 수 있어야 한다. 지하철에서도 승강장과 전동차 출입문 사이의 간격이 넓어 “내리실 때 조심 하시기 바랍니다”라는 안내 방송이 나오지 않던가?

프로그램에서 '지금'에 해당하는 부분, 그리고 '나중'에 해당하는 부분 사이의 관계가 바로 비동기 프로그램의 핵심이다.

자바스크립트가 태동할 무렵부터 비동기 프로그래밍도 줄곧 함께 해왔음에도 자바스크립트 개발자는 대부분 자신의 프로그램에 비동기 요소를 왜, 어떻게 삽입해

야 하는지 별로 신경 쓰지 않고 비동기 프로그래밍 기법을 제대로 공부하려고 하지도 않았다. 늘 ‘이 정도면 됐지.’ 하며 콜백 함수를 쓰는 정도에 만족했을 테고 아직도 많은 이들이 콜백만 알면 되지 않나, 하는 생각을 하는 게 사실이다.

하지만 예전과 달리 자바스크립트는 (요즘은 브라우저는 물론이고 서버 등 거의 모든 가용 장비에서 실행할 수 있어야 하므로) 일급 프로그래밍 언어^{first-class programming language}로서의 요건을 충족하기 위해 응용 범위와 복잡도가 날로 증가하고 있으며, 이에 따라 비동기 요소를 관리해야 하는 부담과 고통 역시 무시 못 할 정도로 커지고 있기 때문에 다양한 요구 조건을 수용하면서도 합리적으로 접근할 방법이 절실했다.

지금은 뜬구름 잡는 얘기처럼 들리겠지만 단언컨대 이 책을 다 읽고 난 독자는 비동기 프로그래밍을 섭렵하게 될 것이다. 다양한 최신 비동기 자바스크립트 기술도 다음 장 이후에 계속 살펴볼 예정이다.

먼저 자바스크립트 언어에서 비동기성^{asynchrony}이란 무엇인지, 작동 원리는 무엇인지, 깊이 있는 배경 지식부터 쌓아보자.

1.1 프로그램 덩어리

자바스크립트 프로그램은 js 파일 하나로도 작성할 수 있지만, 보통은 여러 개의 덩어리^{chunk}, 곧 ‘지금’ 실행 중인 프로그램 덩어리 하나 + ‘나중’에 실행할 프로그램 덩어리들로 구성된다. 가장 일반적인 프로그램 덩어리 단위는 함수^{function}다.

‘나중’은 ‘지금’의 직후가 아니다. 자바스크립트 초심자들이 많이 어려워하는 대목인데, 풀이하자면 ‘지금’ 당장 끝낼 수 없는 작업은 비동기적으로 처리되므로 직관적으로도 알 수 있듯이 프로그램을 중단^{blocking}하지 않는다.

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.

```
var data = ajax( "http://some.url.1" );
```

```
console.log( data );  
// 어이쿠! AJAX 결과는 보통 이렇게 'data'에 담지 못한다.
```

표준 AJAX 요청은 동기적으로 작동하지 않아 `ajax(..)` 함수 결과값을 `data` 변수에 할당할 수 없다는 사실 정도는 이미 알고 있을 것이다. `ajax(..)` 함수가 응답을 받을 때까지 흐름을 중단할 수 있다면 `data = ..` 할당문은 문제없이 실행됐을 것이다.

하지만 AJAX는 이렇게 작동하지 않는다. AJAX는 비동기적으로 '지금' 요청하고 '나중'에 결과를 받는다.

'지금'부터 '나중'까지 "기다리는" 가장 간단한(또 사실상 최적의) 방법은 '콜백 함수 callback function'라는 장치를 이용하는 것이다.

```
// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.  
ajax( "http://some.url.1", function myCallbackFunction(data){  
  
    console.log( data ); // 그렇지, 'data' 수신 완료!  
  
} );
```



동기적인 AJAX 요청도 할 수 있지 않느냐고 반문할 독자들도 있을 것이다. 기술적으로는 가능하지만, 브라우저 UI (버튼, 메뉴, 스크롤바 등)를 얼어붙게 할 뿐만 아니라 사용자와의 상호 작용^{user interaction}이 완전히 마비될 수 있으니 혹여라도 그런 일은 하지 말자. 생각만 해도 끔찍하니 아예 떠올리지 말자.

이 말에 이의를 제기하기 전에 콜백 문제를 피하려고 중단적/동기적 AJAX를 사용하는 행위는 정당화할 수 없음을 밝혀둔다.

예를 들어 다음 코드를 보자.

```
function now() {  
    return 21;
```

```
}

function later() {
    answer = answer * 2;
    console.log( "인생의 의미:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // 인생의 의미: 42
```

‘지금’ 실행할 코드와 ‘나중’에 실행할 코드, 두 덩이로 이루어진 프로그램이다. 각 덩이는 그냥 봐도 식별이 가능하지만 엄청 친절하게 설명해보겠다.

‘지금’ 덩이:

```
function now() {
    return 21;
}

function later() { .. }

var answer = now();

setTimeout( later, 1000 );
```

‘나중’ 덩이:

```
answer = answer * 2;
console.log( "인생의 의미:", answer );
```

프로그램이 시작하면 ‘지금’ 덩이는 바로 실행되지만 `setTimeout(...)`은 ‘나중’ 이벤트(타이머^{timeout})를 설정하는 함수이므로 `later()` 함수는 나중(지금보다 1,000밀리초 후)에 실행된다.

코드 조각을 function으로 감싸놓고 이벤트에 반응하여 움직이게 하려면 '나중' 덩이를 코딩하여 프로그램에 '비동기성'을 부여하면 된다.

1.1.1 비동기 콘솔

Console.* 메서드는 (공식적으로 자바스크립트 일부가 아니므로) 그 작동 방법이나 요건이 명세에 따로 정해져 있지 않지만 '호스팅 환경^{hosting environment}' (본 시리즈 [『You Don't Know JS 타입과 문법』](#) 참고)에 추가된 기능이다.

따라서 브라우저와 자바스크립트 실행 환경에 따라 작동 방식이 다르고 종종 혼동을 유발하기도 한다.

특히 `console.log(..)` 메서드는 브라우저 유형과 상황에 따라 출력할 데이터가 마려운 직후에도 콘솔창에 바로 표시되지 않을 수 있다. (자바스크립트뿐만 아니라) 많은 프로그램에서 I/O 부분이 가장 느리고 중단^{blocking}이 잦기 때문이다. 여러분은 이런 일들이 물밑에서 처리되고 있는 줄도 몰랐겠지만 (페이지/UI 관점에서 보면) 브라우저가 콘솔 I/O를 백그라운드에서 비동기적으로 처리해야 성능상 유리하다.

드물긴 하지만 이런 현상은 코드 자체가 아닌 외부에서 관찰되기도 한다.

```
var a = {
  index: 1
};

// 나중에
console.log( a ); // ??

// 더 나중
a.index++;
```

`console.log(..)` 문이 실행될 때 당연히 객체 스냅샷(`{ index: 1 }`)이 콘솔창에

찍힌 다음에 `a.index++`에서 `a` 값이 증가할 것이다.

분명 예상대로 개발자 도구 콘솔창에 `{ index: 1 }`이 표시되겠지만, 간혹 브라우저가 콘솔 I/O를 백그라운드로 전환하는 것이 좋겠다고 결정하면 출력이 지연될 수 있다. 그래서 `a.index++`이 먼저 실행된 후 콘솔창에 객체값이 전달되어 `{ index: 2 }`로 나올 때가 있다.

I/O 지연이 언제 발생하여 엉뚱한 결과가 빚어질지는 상황에 따라 다르므로 정확히 예측하기 어렵다. 하지만 `console.log(...)` 문 이후 변경된 객체의 프로퍼티 값이 콘솔에 표시되는 문제로 디버깅을 할 땐 이러한 I/O 비동기성이 원인으로 작용할 수 있다는 점을 항상 염두에 두어야 한다.



어쩌다 이렇게 흔치 않은 상황에 부딪칠 경우에는 무조건 콘솔창 결과에 의존하지 말고 자바스크립트 디버거의 중단점(breakpoint)을 잘 활용하는 게 최선이다. 객체를 `JSON.stringify(...)` 등의 함수로 문자열 직렬화하여 “스냅샷”을 강제로 떠보는 것이 차선책이라고 할 수 있다.

1.2 이벤트 루프

폭탄선언을 하겠다(놀라지 마시라). 최근까지(ES6) 여러분이 (좀 전에 보았던 타임아웃 같은) 비동기 자바스크립트 코드를 죽 작성해 왔다 해도 실제로 자바스크립트에 비동기란 개념이 있었던 적은 단 한 번도 없었다.

뭐라고!?! 이 양반이 무슨 말이냐고? 사실이다. 자바스크립트 엔진은 요청하면 프로그램을 주어진 시점에 한 덩어리씩 묵묵히 실행할 뿐이다.

“요청한다”... 누가 요청을? 이 부분이 중요하다!

자바스크립트 엔진은 혼자서는 안 되고 반드시 호스팅에서 실행된다(가장 흔한 호스팅 환경이 바로 웹 브라우저다). 지난 수년 동안 (자바스크립트만 그런 건 아니지만) 자바스

크립트는 브라우저를 벗어나 다른 환경(예: 노드JS^{Node.js} 서버)으로 그 영역을 확장해 왔다. 실상 오늘날 로봇에서 전구에 이르기까지 자바스크립트는 거의 모든 유형의 장치에 탑재되어 있다.

그러나 환경은 달라도 “스레드^{thread}”는 공통이다. 여러 프로그램 덩이를 시간에 따라 매 순간 한 번씩 엔진을 실행시키는 ‘이벤트 루프^{event loop}’라는 장치다.

다시 말해, 자바스크립트 엔진은 애당초 시간이란 관념 따윈 없었고 임의의 자바스크립트 코드 조각을 시시각각 주는 대로 받아 처리하는 실행기일 뿐, “이벤트”(자바스크립트 코드 실행)를 스케줄링하는 일은 언제나 엔진을 감싸고 있던 주위 환경의 몫이었다.

예를 들어 어떤 데이터를 서버에서 조회하려고 AJAX 요청을 할 때 함수 형태로 응답 처리 코드(콜백^{callback}이라 한다)를 작성하는 건 마치 자바스크립트 엔진이 호스팅 환경에 이렇게 이야기하는 것과 같다. “여보게, 지금 잠깐 실행을 멈출 테니 네트워크 요청이 다 끝나서 결과 데이터가 만들어지거든 언제라도 이 함수를 다시 불러주시게나!”

이 말을 남기고 브라우저는 네트워크를 통해 열심히 응답을 리스닝^{listening}한다. 뭔가 사용자에게 줄 데이터가 도착하면 콜백 함수를 이벤트 루프에 삽입하여 실행 스케줄링을 한다.

이벤트 루프는 어떻게 만들어졌을까?

아마 다음과 같은 형태로 구현되어 있을 것이다.

```
// 'eventLoop'는 큐(선입, 선출) 역할을 하는 배열이다.
var eventLoop = [ ];
var event;

// "무한" 실행!
while (true) {
```

```

// "틱" 발생
if (eventLoop.length > 0) {

    // 큐에 있는 다음 이벤트 조회
    event = eventLoop.shift();

    // 이제 다음 이벤트를 실행
    try {
        event();
    }
    catch (err) {
        reportError(err);
    }
}
}

```

물론 개념적으로 말도 안 되게 단순화한 의사 코드다. 감을 잡기엔 이 정도로 충분할 것이다.

코드에 while 무한 루프가 있는데 이 루프의 매 순회^{iteration}를 ‘틱tick’이라고 한다. 틱이 발생할 때마다 큐에 적재된 이벤트(콜백 함수)를 꺼내어 실행한다.

setTimeout(..)은 콜백을 이벤트 루프 큐에 넣지 않는다. 헛갈리지 말자. setTimeout(..)은 타이머를 설정하는 함수다. 타이머가 끝나면 환경이 콜백을 이벤트 루프에 삽입한 뒤 틱에서 콜백을 꺼내어 실행한다.

이벤트 루프가 20개의 원소로 가득 차 있을 땐? 일단 콜백은 기다린다. 먼저 앞으로 가려고 새치기하지 않고 암전히 줄 맨 끝에서 대기한다. setTimeout(..) 타이머가 항상 완벽하게 정확한 타이밍으로 작동하지 않는 게 바로 이 때문이다. (대략 말하면) 적어도 지정한 시간^{interval} 이전에 콜백이 실행되지 않을 거란 사실은 보장할 수 있지만, 정확히 언제, 혹은 좀 더 시간이 경과한 이후에 실행될지는 이벤트 루프 큐의 상황에 따라 달라진다.

자바스크립트 프로그램은 수많은 덩이로 잘게 나누어지고 이벤트 루프 큐에서 한

번에 하나씩 차례대로 실행된다. 엄밀히 말해 이 큐엔 개발자가 작성한 프로그램과 직접 상관없는 여타 이벤트들도 중간에 끼어들 가능성도 있다.



ES6부터는 이벤트 루프 큐 관리 방식이 완전히 바뀌기 때문에 앞에서 “최근까지”라는 표현을 썼다. 이벤트 루프 큐는 준 공식적인 기술 요건임에도 ES6에 이르러서야 작동 방식이 정확히 규정되어 이제야 호스팅 환경이 아닌, 자바스크립트 엔진의 관찰이 되었다. 3장 주제 프라미스 도입을 계기로 이러한 변화가 일어났다. 프라미스는 이벤트 루프 큐의 스케줄링을 직접 세밀하게 제어해야 하기 때문이다.

1.3 병렬 스레딩

“비동기^{async}”와 “병렬^{parallel}”은 아무렇게나 섞어 쓰는 경우가 많지만 그 의미는 완전히 다르다. 비동기는 ‘지금’과 ‘나중’ 사이의 간극에 관한 용어고 병렬은 동시에 일어나는 일들과 연관된다.

‘프로세스^{process}’와 ‘스레드^{thread}’는 가장 많이 쓰는 병렬 컴퓨팅^{parallel computing} 도구로, 별개의 프로세서^{processor}, 심지어는 물리적으로 분리된 컴퓨터에서도 독립적으로 (때로는 동시에) 실행되며 여러 스레드는 하나의 프로세스 메모리를 공유한다.

반면 이벤트 루프는 작업 단위로 나누어 차례대로 실행하지만, 공유 메모리에 병렬로 접근하거나 변경할 수는 없다. 병렬성^{parallelism}과 직렬성^{serialism}이 나뉜 스레드에서 이벤트 루프를 협동^{cooperation}하는 형태로 공존하는 모습이다.

병렬 실행 스레드 인터리빙^{interleaving}과 비동기 이벤트 인터리빙은 완전히 다른 수준의 단위^{granularity}에서 일어난다.

예를 들면,

```
function later() {
  answer = answer * 2;
  console.log( "인생의 의미:", answer );
}
```

later() 함수 전체 내용은 이벤트 루프 큐가 하나의 원소로 취급하므로 이 함수를 실행 중인 스레드 입장에선 실제로 여러 상이한 저수준의 작업들이 일어날 수 있다. 예컨대, answer = answer * 2는 '현재 answer 값 조회 → 곱셈 연산 수행 → 곱셈값을 다시 answer 변수에 저장' 순으로 처리한다.

단일-스레드 환경에서는 스레드 간섭은 일어나지 않으므로 스레드 큐에 저수준 작업의 원소들이 쌓여 있어도 별문제 없다. 하지만 하나의 프로그램에서 여러 스레드를 처리하는 병렬 시스템에선 예상치 못했던 일들이 일어날 수 있다.

다음 코드를 보자.

```
var a = 20;

function foo() {
  a = a + 1;
}

function bar() {
  a = a * 2;
}

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

자바스크립트는 단일-스레드로 작동하니까 foo() → bar() 순서로 실행하면 곱셈값은 42지만 반대로 bar() → foo() 순서면 41이 된다.

동일한 데이터를 공유하는 자바스크립트 이벤트의 병렬 실행 문제는 더 복잡하다. foo()와 bar()를 제각기 실행하는 두 스레드의 의사 코드 목록을 써보자. 정확히 똑같은 시점에 두 스레드가 실행되면 어떤 일들이 벌어질까?

스레드 1 (X와 Y는 임시 메모리 공간이다):

foo():

- a. 'a' 값을 'X'에 읽어들인다.
 - b. '1'을 'Y'에 저장한다.
 - c. 'X'와 'Y'를 더하고 그 결과값을 'X'에 저장한다.
 - d. 'X' 값을 'a'에 저장한다.
-

스레드 2(X와 Y는 임시 메모리 공간이다):

bar():

- a. 'a' 값을 'X'에 읽어들인다.
 - b. '2'을 'Y'에 저장한다.
 - c. 'X'와 'Y'를 곱하고 그 결과값을 'X'에 저장한다.
 - d. 'X' 값을 'a'에 저장한다.
-

두 스레드가 진짜 병렬 상태로 실행되면 어떤 문제가 발생할지 이젠 알겠는가? 그렇다, 중간 단계에서 X와 Y라는 메모리 공간을 공유하는 것이 문제다.

다음처럼 진행하면 a의 최종 결과값은 얼마일까?

- 1a ('X'에서 'a' 값을 읽어들인다. ==> '20')
 - 2a ('X'에서 'a' 값을 읽어들인다. ==> '20')
 - 1b ('Y'에 '1'을 저장한다. ==> '1')
 - 2b ('Y'에 '2'을 저장한다. ==> '2')
 - 1c ('X'와 'Y'를 더하고 그 결과값을 'X'에 저장한다. ==> '22')
 - 1d ('a'에 'X' 값을 저장한다. ==> '22')
 - 2c ('X'와 'Y'를 곱하고 그 결과값을 'X'에 저장한다. ==> '44')
 - 2d ('a'에 'X' 값을 저장한다. ==> '44')
-

44다. 하지만 순서가 이렇게 바뀌면?

- 1a ('X'에서 'a' 값을 읽어들인다. ==> '20')
- 2a ('X'에서 'a' 값을 읽어들인다. ==> '20')
- 2b ('Y'에 '2'을 저장한다. ==> '2')
- 1b ('Y'에 '1'을 저장한다. ==> '1')
- 2c ('X'와 'Y'를 곱하고 그 결과값을 'X'에 저장한다. ==> '20')

1c ('X'와 'Y'를 더하고 그 결과값을 'X'에 저장한다. ==> '21')

1d ('a'에 'X' 값을 저장한다. ==> '21')

2d ('a'에 'X' 값을 저장한다. ==> '21')

결괏값은 21이다.

자, 이래서 스레드 프로그래밍^{threaded programming}이 어려운 것이다. 인터럽션/인터리빙 같은 요소가 발생하지 않도록 조치하지 않으면 정말 기가 막힐 정도로 제멋대로 왔다 갔다 하는 결괏값 때문에 편두통에 시달릴지 모른다.

자바스크립트는 절대로 스레드 간에 데이터를 공유하는 법이 없으므로 비결정성의 수준^{level of nondeterminism}은 문제가 되지 않는다. 하지만 그렇다고 자바스크립트 프로그램이 항상 결정적^{deterministic}이란 소리도 아니다. 좀 전의 예제에서도 foo()와 bar()의 실행 순서에 따라 결괏값이 41, 42 사이를 오락가락하지 않았던가?



아직은 그리 마음에 와 닿는 말이 아니겠지만 모든 비결정성이 나쁜 건 아니다. 무관한 경우도 있고 의도적인 경우도 있다. 2장 이후부터 그런 예제 코드를 자주 접하게 될 것이다.

1.3.1 완전-실행

자바스크립트의 작동 모드는 싱글-스레드이므로 foo() (bar()도 마찬가지로) 내부의 코드는 원자적^{atomic}이다. 즉, 일단 foo()가 실행되면 이 함수 전체 코드가 실행되고 나서야 bar() 함수로 옮겨간다는 뜻이다. 이를 완전-실행^{Run-to-Completion}이라 한다.

다음 예제처럼 foo()와 bar() 함수에 코드를 한번 넣어보면 완전-실행의 의미가 분명해진다.

```
var a = 1;  
var b = 2;
```

```
function foo() {
  a++;
  b = b * a;
  a = b + 3;
}
```

```
function bar() {
  b--;
  a = 8 + b;
  b = a * 2;
}
```

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.

```
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

foo ()와 bar ()는 상대의 실행을 방해할 수 없으므로 이 프로그램의 결괏값은 먼저 실행되는 함수가 좌우한다. 만약 문^{statement} 단위로도 스레딩이 일어나면 문별 인터리빙이 발생하여 경우의 수는 기하급수적으로 늘어난다!

덩이 1은 ('지금' 실행 중인) 동기 코드, 덩이 2와 3은 ('나중'에 실행될) 비동기 코드로, 일정 시간 차이를 두고 실행된다.

덩이 1:

```
var a = 1;
var b = 2;
```

덩이 2 (foo ()):

```
a++;
b = b * a;
a = b + 3;
```

덩이 3 (bar ()):

```
b--;  
a = 8 + b;  
b = a * 2;
```

덩이 2와 덩이 3은 선발순^{either-first order}으로 실행⁰¹되므로 결과는 다음 둘 중 하나다.

결과 1:

```
var a = 1;  
var b = 2;  
  
// foo()  
a++;  
b = b * a;  
a = b + 3;  
  
// bar()  
b--;  
a = 8 + b;  
b = a * 2;  
  
a; // 11  
b; // 22
```

결과 2:

```
var a = 1;  
var b = 2;  
  
// bar()  
b--;  
a = 8 + b;
```

01 역자주_둘 중 어느 한쪽이 먼저 출발(실행)한 순서대로 실행된다는 뜻입니다. 즉, 먼저 출발한 쪽이 실행을 마칠 때까지 다른 한쪽은 마냥 기다려야 합니다.

```
b = a * 2;
```

```
// foo()
```

```
a++;
```

```
b = b * a;
```

```
a = b + 3;
```

```
a; // 183
```

```
b; // 180
```

똑같은 코드인데 결괏값은 두 가지이므로 이 프로그램은 비결정적이다. 그러나 여기서 비결정성은 함수(이벤트)의 순서에 따른 것이지, 스레드처럼 문의 순서(표현식의 처리 순서) 수준까지는 아니다. 즉, 스레드보다는 결정적이라고 할 수 있다.

자바스크립트에서는 함수 순서에 따른 비결정성을 흔히 경합 조건^{race condition}이라고 표현한다. foo ()와 bar () 중 누가 먼저 실행되나 내기하는 경합 같다는 의미에서다. 구체적으로는 a와 b의 결괏값을 예측할 수 없으므로 경합 조건이 맞다.



자바스크립트 함수가 완전-실행되지 않는다면 가능한 결과의 가짓수는 엄청나게 늘어날 것이다. 실제로 ES6부터 그런 함수가 등장하는데(4장 참고) 일단 지금은 걱정하지 마시길 나중에 다시 설명한다.

1.4 동시성

사용자가 스크롤바를 아래로 내리면 계속 (소셜 네트워크의 뉴스 피드 등에서) 갱신된 상태 리스트가 화면에 표시되는 웹 페이지를 만들고자 한다. 이런 기능은 (적어도) 2개의 분리된 “프로세스”를 동시에(같은 시구간^{Time window}에서, 반드시 동일한 순간이 아니어도 상관없다) 실행할 수 있어야 제대로 기능을 구현할 수 있다.



엄밀히 말하면 컴퓨터 과학 교과서에 나오는 운영 체제 수준의 프로세스와 구별하기 위해 “프로세스” 양쪽에 큰따옴표를 붙였다. 여기서 말하는 프로세스란 논리적으로 연결된 순차적인 일련의 작업을 나타내는 가상 프로세스, 또는 작업을 의미한다. “작업” 대신 “프로세스”를 택한 건 여기서 설명하려는 개념과 잘 맞기 때문이다.

첫 번째 “프로세스”는 사용자가 페이지를 스크롤바로 내리는 순간 발생하는 onscroll 이벤트에 반응한다(AJAX 요청 후 더 많은 데이터를 가져온다). 두 번째 “프로세스”는 AJAX 응답을 받는다(그리고 페이지에 데이터를 표시한다).

성미 급한 사용자가 아주 빨리 스크롤바를 내리면 처음 수신된 응답을 처리하는 도중 2개 이상의 onscroll 이벤트가 발생하기에 십상이고 onscroll 이벤트와 AJAX 요청 이벤트가 아주 빠르게 발생하며 인터리빙 된다.

동시성은 복수의 “프로세스”가 동일한 시간 동안 동시에 실행됨을 의미하며, 각 프로세스 작업들이 병렬로(별개의 프로세서/코어에서 동일한 시점에) 처리되는지 여부와는 관계없다. 동시성은 처리 수준^{operation-level} 병행성(개별 프로세서의 스레드)과 상반되는 개념의 “프로세스” 수준^{process-level}(작업 수준)의 병행성이라 할 수 있다.



나중에 다시 언급하지만, 동시성에는 이러한 “프로세스”들이 상호 작용한다는 개념도 포함되어 있다.

주어진 시구간(사용자가 스크롤하는 2, 3초 정도의 시간) 동안 독립적인 각 “프로세스”를 이벤트/처리 목록으로 시각화해보자.

“프로세스” 1 (onscroll 이벤트):

```
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
```

“프로세스” 2 (AJAX 응답 이벤트):

```
response 1
response 2
response 3
response 4
response 5
response 6
response 7
```

onscroll 이벤트와 AJAX 응답 이벤트는 동시에 발생할 수 있다. 예를 들어, 시간에 따른 이벤트를 나열해보면,

```
onscroll, request 1
onscroll, request 2   response 1
onscroll, request 3   response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6   response 4
onscroll, request 7
response 6
response 5
response 7
```

하지만 이벤트 루프 개념을 다시 곱씹어보면 자바스크립트는 한 번에 하나의 이벤트만 처리하므로 onscroll, request 2든 response 1이든 둘 중 어느 한쪽이 먼저 실행되고 정확히 같은 시각에 실행되는 일은 결코 있을 수 없다. 학교 구내식당에 한꺼번에 학생들이 몰려들어 아수라장이 되어도 결국 한 줄로 서서 배식을 받을 수밖에 없는 현실과 비슷하다.

이벤트 루프 큐에서 이벤트들은 어떻게 인터리빙 될까?

```
onscroll, request 1  <← 프로세스 1 시작
onscroll, request 2
response 1           <← 프로세스 2 시작
onscroll, request 3
response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
response 4
onscroll, request 7  <← 프로세스 1 종료
response 6
response 5
response 7           <← 프로세스 2 종료
```

“프로세스” 1과 “프로세스” 2는 동시에(작업 수준의 병행성) 실행되지만, 이들을 구성하는 이벤트들은 이벤트 루프 큐에서 차례대로 실행된다.

그런데 response 6과 response 5는 어떻게 순서가 뒤바뀐 걸까?

단일-스레드 이벤트 루프는 동시성을 나타내는 하나의 표현 방식이다(다른 방식은 차후 또 설명한다).

1.4.1 비상호작용

어떤 프로그램 내에서 복수의 “프로세스”가 단계/이벤트를 동시에 인터리빙 할 때 이들 프로세스 사이에 연관된 작업이 없다면 사실 프로세스 간 상호 작용은 의미가 없다. 프로세스 간 상호 작용이 일어나지 않는다면 비결정성은 완벽하게 수용 가능하다.

예를 들어,

```
var res = {};  
  
function foo(results) {  
    res.foo = results;  
}  
  
function bar(results) {  
    res.bar = results;  
}  
  
// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.  
ajax( "http://some.url.1", foo );  
ajax( "http://some.url.2", bar );
```

2개의 동시 “프로세스” foo()와 bar() 중 누가 먼저 실행될지 알 수는 없지만 적어도 서로에게 아무런 영향을 끼치지 않고 개별 작동하니 실행 순서는 문제를 삼을 필요가 없다.

순서와 상관없이 언제나 정확히 작동하므로 경합 조건에 따른 버그는 아니다.

1.4.2 상호작용

동시 “프로세스”들은 필요할 때 스코프나 DOM을 통해 간접적으로 상호 작용을 한다. 이때 이미 한번 살펴봤던 것처럼 경합 조건이 발생하지 않도록 잘 조율해주어야 한다.

다음 코드는 암묵적인 순서 때문에 두 개의 동시 “프로세스”가 가끔 깨지는 예다.

```
var res = [];  
  
function response(data) {  
    res.push( data );  
}  
  
// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.  
ajax( "http://some.url.1", response );
```

```
ajax( "http://some.url.2", response );
```

두 동시 “프로세스” 모두 AJAX 응답 처리를 하는 response () 함수를 호출하는 터라 선발 순으로 처리된다.

아마도 프로그램 개발자는 “http://some.url.1” 결과는 res[0]에, “http://some.url.2” 결과는 res[1]에 담고 싶었을 것이다. 이 의도대로 실행될 때도 있지만, 어느 쪽 URL 응답이 먼저 도착할지에 따라 결과는 뒤집힐 수 있다.



이런 상황에서 선부른 가정을 하는 사람들이 많은데 유의해야 한다. 보통 개발자들은 처리하는 작업의 성격을 보고 (예컨대, 한쪽은 데이터베이스를 조회하고 다른 한쪽은 정적 파일을 읽어드리는 작업이라면) “http://some.url.2”의 응답이 “http://some.url.1”보다 항상 느릴 거라는 식으로 확신한다. 하지만 요청 서버가 동일하고 이 서버가 특정 순서로 응답하도록 고정을 해놓아도 브라우저에 정말 어떤 순서로 응답이 도착할지는 아무도 장담할 수 없다.

따라서 경합 조건을 해결하려면 상호 작용의 순서를 잘 조정해야 한다.

```
var res = [];
```

```
function response(data) {  
    if (data.url == "http://some.url.1") {  
        res[0] = data;  
    }  
    else if (data.url == "http://some.url.2") {  
        res[1] = data;  
    }  
}
```

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.

```
ajax( "http://some.url.1", response );  
ajax( "http://some.url.2", response );
```

이제 어느 쪽 AJAX 응답이 먼저 오더라도 data.url(서버에서 반환한다.)을 보고

res 배열의 어느 슬롯에 응답 데이터를 저장할지 결정할 수 있다. 즉, res[0]엔 “http://some.url.1”이, res[1]엔 “http://some.url.2”의 응답 결과가 항상 저장될 것이다. 간단한 조정만으로 경합 조건에 의한 비결정성을 해소한 사례다.

한꺼번에 여러 함수를 호출하는 형태로 공유 DOM을 통해 상호 작용하는 경우도 마찬가지다. 이를테면, <div> 내용을 업데이트하는 함수와 <div>의 속성/스타일을 수정(말하자면 DOM 요소의 내용이 채워진 이후에 화면에 보여주는 식으로)하는 함수가 있다고 하자. 내용이 텅 채워진 DOM 요소를 화면에 보여주고 싶지는 않을 테니 세심하게 조정할 필요가 있다.

조정이 잘 안 되면 동시성이 항상(가끔만 그런 게 아니라) 문제 되는 경우가 있다. 다음 코드를 보자.

```
var a, b;

function foo(x) {
  a = x * 2;
  baz();
}

function bar(y) {
  b = y * 2;
  baz();
}

function baz() {
  console.log(a + b);
}

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

foo()나 bar() 중 어느 쪽이 먼저 실행되더라도 baz() 함수는 처음에 항상 너

무 빨리 부른다(a나 b 중 하나는 여전히 undefined인 상태다). 그러나 두 번째 실행할 때 a, b 모두 값이 존재하니 제대로 작동한다.

해결 방법은 여러 가지인데 간단히 하면,

```
var a, b;

function foo(x) {
  a = x * 2;
  if (a && b) {
    baz();
  }
}

function bar(y) {
  b = y * 2;
  if (a && b) {
    baz();
  }
}

function baz() {
  console.log( a + b );
}

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

if (a && b) 조건으로 baz () 호출을 에두른 형태를 예전부터 ‘관문^{gate}’이라고 불려왔다. a와 b 중 누가 일찍 도착할지 알 수는 없지만, 관문은 반드시 둘 다 도착한 다음에야 열린다(즉, baz () 함수를 부른다).

이러한 동시적 상호 작용 조건은 또 있다. 경합이라 부르는 경우도 있지만, 더 정확히는 결쇠^{latch}라는 용어가 맞고 “선착순 한 명만 이기는” 형태다. 비결정성을 수용하는 조건으로 결승선을 통과한 오직 한 명의 승자만 뽑는 “달리기 시합”을 명

시적으로 선언하는 것이다.

잘못된 코드를 먼저 보자.

```
var a;

function foo(x) {
  a = x * 2;
  baz();
}

function bar(x) {
  a = x / 2;
  baz();
}

function baz() {
  console.log( a );
}

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

(foo(), bar()) 둘 중 하나는 나중에 실행된 함수가 다른 함수가 할당된 값을 덮어쓸 뿐만 아니라 baz()를 한 번 더 호출하게 되는 (의도하지 않은) 코드다.

결쇠로 조정하면 간단히 선착순으로 바꿀 수 있다.

```
var a;

function foo(x) {
  if (!a) {
    a = x * 2;
    baz();
  }
}
```

```
function bar(x) {
  if (!a) {
    a = x / 2;
    baz();
  }
}

function baz() {
  console.log( a );
}
```

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.

```
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

foo(), bar() 둘 중 첫 번째 실행된 함수가 if (!a) 조건을 통과하고 두 번째 (늦게 실행된) 함수 호출은 무시한다. 2등에게 상은 없다!



예제에서는 간단히 설명하기 위해 전역 변수를 사용했지만, 꼭 전역 변수를 써야 할 이유는 없다. 함수가 (스코프 내에서) 변수에 접근할 수만 있으면 의도한 대로 작동할 것이다. 어휘 스코프 변수(본 시리즈 『You Don't Know JS 스코프와 클로저』 참고)나 예제처럼 전역 변수에 의존하는 건 동시성 문제 해결에 전혀 바람직하지 않다. 2장부터 이런 관점에서 좀 더 깔끔하게 해결할 방법을 찾아볼 것이다.

1.4.3 협동

‘협동적 동시성^{cooperative concurrency}’ 역시 동시성을 조정하는 다른 방안으로, 스코프에서 값을 공유하는 식의 상호 작용(그렇게 할 순 있지만)엔 별 관심이 없다. 협동적 동시성은 실행 시간이 오래 걸리는 “프로세스”를 여러 단계/배치로 쪼개어 다른 동시 “프로세스”가 각자 작업을 이벤트 루프 큐에 인터리빙 할 수 있게 해주는 게 목표다.

예를 들어 아주 긴 리스트를 받아 값을 변환하는 AJAX 응답 처리기가 있다고 하자. Array#map(..)를 사용하여 코드를 들여보면,

```

var res = [];

// AJAX 호출 결과 'response(..)'는 배열을 받는다.
function response(data) {
    // 기존 'res' 배열에 추가한다.
    res = res.concat(
        // 배열의 원소를 하나씩 변환한다.
        // 원래 값을 2배로 늘린다.
        data.map( function(val){
            return val * 2;
        } )
    );
}

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );

```

처음 “http://some.url.1” 호출 결과가 넘어오면 전체 리스트는 바로 res에 매핑된다. 몇천 개 정도의 레코드라면 별문제 아니지만, 개수가 천만 개 정도에 이르면 처리 시간이 제법 걸린다(고급 사양의 노트북 PC라면 수 초 정도 걸릴 테고 모바일 기기에선 더 오래 걸릴 수 있다).

이 “프로세스” 실행 중에 페이지는 그대로 멈춰버린다. response (..) 함수의 실행, UI 업데이트는 물론이고, 심지어 스크롤링, 타이핑, 버튼 클릭 등의 사용자 이벤트도 먹통이다. 대단히 난감할 것이다.

따라서 이벤트 루프 큐를 독점하지 않는, 보다 친화적이고 협동적인 동시 시스템이 되려면 각 결과를 비동기 배치로 처리하고 이벤트 루프에서 대기 중인 다른 이벤트들과 함께 실행되게끔 해야 한다.

아주 간단한 예를 들겠다.

```

var res = [];

```



```

// 'response(..)'는 AJAX 호출 결과로 배열을 받는다.
function response(data) {
    // 한번에 1,000개씩 실행하자.
    var chunk = data.splice( 0, 1000 );

    // 기존 'res' 배열에 추가한다.
    res = res.concat(
        // 배열의 원소를 하나씩 변환한다.
        // 'chunk'값에 2를 곱한다.
        chunk.map( function(val){
            return val * 2;
        } )
    );

    // 아직도 처리할 프로세스가 남아 있나?
    if (data.length > 0) {
        // 다음 배치를 비동기 스케줄링한다.
        setTimeout( function(){
            response( data );
        }, 0 );
    }
}

// ajax(..)는 라이브러리에 있는 임의의 AJAX 함수다.
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );

```

데이터 집합을 최대 1,000개 원소를 가진 덩어리 단위로 처리했다. 이렇게 하면 더 많은 후속 “프로세스”를 처리해야 하지만, 각 “프로세스” 처리 시간은 단축되므로 이벤트 루프 큐에 인터리빙이 가능하고 응답성이 좋은(성능 요건을 충족하는) 사이트/앱을 만들 수 있다.

물론 이렇게 나뉜 “프로세스”들의 실행 순서까지 조정하는 것은 아니므로 res 배열에 어떤 순서로 결과가 저장될지 예측하기 어렵다. 순서가 중요한 경우라면 앞서 설명한 기법, 또는 2장부터 배우게 될 다양한 기법을 끌어써야 한다.

setTimeout(...0)은 비동기 스케줄링 꼼수^{hack} 중 하나로, “이 함수를 현재 이벤트 루프 큐의 맨 뒤에 붙여주세요”라는 말이다.



setTimeout(...0)은 엄밀히 말해 원소를 이벤트 루프 큐에 곧바로 삽입하는 게 아니라 타이머가 다음 기회에 이벤트를 삽입한다. 예컨대, 연속 두 번 setTimeout(...0)을 호출해도 그 순서대로 처리되리란 보장은 없다. 따라서 이벤트 순서를 예측할 수 없는 타이머 표류^{timer drift} 등의 다양한 상황이 연출될 수 있다. NodeJS의 process.nextTick(...)도 사용은 편하지만(그리고 보통 성능이 좋지만) 모든 환경에서 비동기 이벤트 순서를 고정할 직접적인 방법은 (적어도 아직은) 없다. 다음 절에서 이 문제를 더 자세히 살펴보자.

1.5 잡

‘잡 큐^{job queue}’는 ES6부터 이벤트 루프 큐에 새롭게 도입된 개념이다. 주로 프라미스(3장 참고)의 비동기 작동에서 가장 많이 보게 될 것이다.

아쉽게도 잡 큐는 아직 공개 API조차 마련되지 않은 터라 구체적인 실례를 들어 설명하긴 곤란하다. 그래서 1장에선 개념 정도만 간단히 소개하고 3장에서 프라미스의 비동기 작동을 배우면서 스케줄링 및 처리 방법을 이해하도록 하자.

‘잡 큐는 이벤트 루프 큐에서 매 틱의 끝자락에 매달려 있는 큐’라고 생각하면 가장 알기 쉽다. 이벤트 루프 틱 도중 발생 가능한, 비동기 특성이 내재된 액션으로 인해 전혀 새로운 이벤트가 이벤트 루프 큐에 추가되는 게 아니라, 현재 틱의 잡 큐 끝 부분에 원소(잡)가 추가된다.

마치 이런 말을 하는 것과 같다. “자, 이진 ‘나중’에 처리할 작업인데, 다른 어떤 작업들보다 우선하여 바로 처리해주게나.”

비유하자면 이벤트 루프 큐는 테마파크에서 롤러코스터를 타고나서 한 번 더 타고 싶으면 다시 대기열 맨 끝에서 기다리는 것이고, 잡 큐는 롤러코스터에서 내린 직후 대기열 맨 앞에서 곧바로 다시 타는 것이다.

잡은 같은 큐 끝에 더 많은 잡을 추가할 수 있는 구조이기 때문에 이론적으로는 ‘잡 루프(job loop)’(계속 다른 잡을 추가하는 잡)가 무한 반복되면서 프로그램이 다음 이벤트 루프 틱으로 이동할 기력을 결국 상실할 수도 있다. 개념적으로는 프로그램에서 실행 시간이 긴 코드, 또는 무한 루프(while(true) .. 같은)를 표현한 것과 비슷하다.

잡은 기본적으로 setTimeout(. . 0) 같은 썬수와 의도는 비슷하지만, 처리 순서(나중에, 하지만 가급적 빨리)가 더 잘 정의되어 있고 순서가 확실히 보장되는 방향으로 구현되어 있다.

다음은 (썬수 없이 직접) 잡 스케줄링을 하는 schedule(. .)이라는 API다.

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

setTimeout( function(){
    console.log( "B" );
}, 0 );

// 이론적인 "잡 API"
schedule( function(){
    console.log( "C" );

    schedule( function(){
        console.log( "D" );
    } );
} );
```

A B C D일 것 같지만, 실행 결과는 A C D B다. 잡은 ‘현재’ 이벤트 루프 틱의 끝에서 시작하지만, 타이머는 (가능하다면) ‘다음’ 이벤트 루프 틱에서 실행하도록 스케줄링하기 때문이다.

3장에서 잡 기반의 프라미스 비동기 작동을 다룰 예정이다. 지금은 잡과 이벤트 루프 사이의 관계만 분명히 이해하고 책장을 넘기자.

1.6 문 순서

자바스크립트 엔진은 반드시 프로그램에 표현된 문의 순서대로 실행하지 않는다. 뭘 소리인가 싶을 텐데 간략히 살펴보겠다.

자, 그 전에 한 가지만 분명히 하자. 프로그래밍 언어의 규칙/문법(본 시리즈 『[You Don't Know JS 타입과 문법](#)』 참고)에는 프로그램 관점에서 문의 순서에 대해 매우 미덥고 예측 가능한 작동 방식이 기술되어 있으므로 여기서 내가 말하려는 내용은 여러분이 작성한 자바스크립트 프로그램에서 육안으로는 확인할 수 없다.



만일 컴파일러가 (지금부터 내가 말하려는) 문의 순서를 재정렬하는 광경을 여러분이 목격한다면 이는 명백한 명세 위반이며 (엔진 개발자에게 정정해달라고 전화해 요구해야 할) 자바스크립트 엔진 버그다. 하지만 여러분이 작성한 코드를 엔진이 이상하게, 이해할 수 없는 방식으로 실행한다면 여러분의 코드에 (아마도 경합 조건 같은) 버그가 숨어있을 가능성이 크다. 따라서 먼저 작성한 코드를 차근차근 뜯어보기 바란다. 자바스크립트 디버거의 중단점 기능을 이용하면 줄 단위로 단계별 확인이 가능해서 디버깅 시 아주 유용하다.

```
var a, b;
```

```
a = 10;
```

```
b = 30;
```

```
a = a + 1;
```

```
b = b + 1;
```

```
console.log( a + b ); // 42
```

이 코드는 (앞서 예시한, 아주 드문 콘솔의 비동기 I/O 경우를 제외하고) 비동기적인 요소가 없어서 당연히 위 → 아래 방향으로 한 줄 씩 실행될 것처럼 보인다.

그러나 자바스크립트 엔진은 이 코드를 컴파일(그렇다, 자바스크립트는 컴파일 언어다 - 본 시리즈 『[You don't Know JS 스코프와 클로저](#)』 참고)한 뒤, 문 순서를 (안전하게) 재정

렬하면서 실행 시간을 줄일 여지는 없는지 확인한다. 이런 과정은 개발자가 들여다볼 수 없으므로 전혀 문제 될 일은 없다.

가령, 엔진은 이렇게 실행하면 더 빠르다고 판단할 수 있다.

```
var a, b;

a = 10;
a++;

b = 30;
b++;

console.log( a + b ); // 42
```

아니면 이렇게,

```
var a, b;

a = 11;
b = 31;

console.log( a + b ); // 42
```

심지어는,

```
// 'a'와 'b'는 더 이상 쓰지 않으므로
// 할당값을 그냥 쓰기만 할 변수는 필요없다!
console.log( 42 ); // 42
```

어떤 경우라도 자바스크립트 엔진은 컴파일 과정에서 최종 결과가 뒤바뀌지 않도록 안전하게 최적화한다.

그러나 안전하지 않아 최적화하면 안 되는 경우도 있다(물론, 아예 최적화하지 않는 건

아니다).

```
var a, b;
```

```
a = 10;
```

```
b = 30;
```

```
console.log( a * b ); // 300
```

```
a = a + 1;
```

```
b = b + 1;
```

```
console.log( a + b ); // 42
```

부수 효과(side effect)가 있는 함수 호출(특히 게터 함수getter function), ES6 프록시Proxy 객체 (본 시리즈 『You Don't Know JS ES6와 그 이후』 참고) 등 컴파일러의 순서 조정으로 인해 현저한 부수 효과가 발생할 수 있다(따라서 순서 조정을 하면 안 된다).

```
function foo() {  
  console.log( b );  
  return 1;  
}
```

```
var a, b, c;
```

```
// ES5.1 게터 리터럴 구문
```

```
c = {  
  get bar() {  
    console.log( a );  
    return 1;  
  }  
};
```

```
a = 10;
```

```
b = 30;
```

```
a += foo(); // 30
```

```
b += c.bar; // 11
```

```
console.log( a + b ); // 42
```

만약 `console.log(...)`(부수 효과의 예시를 위해 확인하기 쉬운 코드로 고른 것이다)가 없으면 자바스크립트 엔진은 내키는 대로(내킬지 안 내킬지는 아무도 모른다) 다음처럼 코드 순서를 바꿀 것이다.

```
// ...
```

```
a = 10 + foo();
```

```
b = 30 + c.bar;
```

```
// ...
```

다행히 자바스크립트 언어는 컴파일러가 임의로 문 순서를 변경함으로 인해 너무 뻔한 곤경에 처할 일은 없지만, 소스 코드 순서(위 → 아래)와 컴파일 후 실행 순서는 사실상 아무 관련이 없다는 사실을 기억하기 바란다.

컴파일러의 문 순서는 동시성과 상호 작용을 미세하게 비유^{micro-metaphor}한 것이다. 자, 일반적인 개념은 이 정도만 알아도 자바스크립트의 비동기 코드 흐름을 이해하는 데 큰 어려움은 없을 것이다.

1.7 정리하기

자바스크립트 프로그램은 (사실상) 언제나 2개 이상의 덩이로 쪼개지며 이벤트 응답으로 첫 번째 덩이는 '지금', 다음 덩이는 '나중'에 실행된다. 한 덩이씩 실행되도 모든 덩이가 프로그램의 스코프/상태에 똑같이 접근할 수 있으므로 상태 변화는 차례대로 반영된다.

실행할 이벤트가 있으면 이벤트 루프는 큐를 다 비울 때까지 실행한다. 이벤트 루프를 한 차례 순회하는 것을 턱이라 한다. 이벤트 큐에 UI, IO, 타이머는 이벤트 큐에 이벤트를 넣는다.

언제나 한 번에 정확히 한 개의 이벤트만 큐에서 꺼내 처리한다. 이벤트 실행 도중, 하나 또는 그 이상의 후속 이벤트를 직/간접적으로 일으킬 수 있다.

동시성은 복수의 이벤트들이 연쇄적으로 시간에 따라 인터리빙 되면서, 고수준의 관점에서 볼 때 (실제로는 특정 시점에 1개 이벤트만 처리되고 있지만) 꼭 동시에 실행되는 것처럼 보인다.

(OS의 시스템 프로세스와는 달리) 동시 “프로세스”들은 어떤 형태로든 서로 영향을 미치는 작업을 조정하여 실행 순서를 보장하거나 경합 조건을 예방하는 등 조치를 해야 한다. 이 “프로세스” 자체를 더 작은 덩이로 잘게 나누어 다른 “프로세스”에 인터리빙 되는 형태의 협동 또한 가능하다.