

Hanbit  
RealTime  
121



# You Don't Know JS

# 비동기와 성능

카일 심슨 지음 / 이일웅 옮김



O'REILLY

 한빛미디어  
Hanbit Media, Inc.

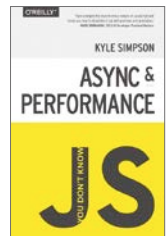


# You Don't Know JS

# 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** 김태현

**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 알려주십시오.

**한빛미디어 홈페이지** [www.hanbit.co.kr](http://www.hanbit.co.kr) / **이메일** [ask@hanbit.co.kr](mailto: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](mailto: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 → 복사후붙여넣기 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 변안 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<sup>jQuery</sup>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<sup>closure</sup>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<sup>callback</sup> 정도는 알고 있을 것이다. 물론 전채 요리만으로는 만족스러운 식사가 될 수 없으니 산해진미로 가득한 다음 코스 프라미스<sup>promise</sup>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<sup>generator</sup>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](http://jakearchibald.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<sup>quirk</sup>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<sup>Brendan Eich</sup><sup>01</sup>의 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<sup>car</sup>”만큼이나 무관하다.

C 스타일의 절차 언어에서 미묘하며 불확실한 스킴<sup>Scheme</sup>/리스프<sup>Lisp</sup> 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 열흘 만에 자바스크립트 언어를 고안했습니다.



른 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(콜백 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<sup>The Tough Part</sup>”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 않으려나.

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 드리고자 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 | 예러/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   | 예러 처리                        | 125 |
| 3.5.1 | 절망의 구덩이                      | 128 |
| 3.5.2 | 잡히지 않은 예러 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<sup>now</sup>’과 ‘나중<sup>later</sup>’ 사이의 간극<sup>gap</sup>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<sup>task</sup> 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<sup>first-class programming language</sup>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<sup>asynchrony</sup>이란 무엇인지, 작동 원리는 무엇인지, 깊이 있는 배경 지식부터 짚아보자.

## 1.1 프로그램 덩이

자바스크립트 프로그램은 js 파일 하나로도 작성할 수 있지만, 보통은 여러 개의 덩이<sup>chunk</sup>, 곧 ‘지금’ 실행 중인 프로그램 덩이 하나 + ‘나중’에 실행할 프로그램 덩이들로 구성된다. 가장 일반적인 프로그램 덩이 단위는 함수<sup>function</sup>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<sup>blocking</sup>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 interaction이 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(타임아웃<sup>timeout</sup>)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 일부분이 아니므로) 그 작동 방법이나  
요건이 명세에 따로 정해져 있지 않지만 ‘호스팅 환경<sup>hosting environment</sup>’ (본 시리즈  
『You Don't Know JS 타입과 문법』 참고)에 추가된 기능이다.

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

특히 console.log(..) 메서드는 브라우저 유형과 상황에 따라 출력할 데이터  
가 마련된 직후에도 콘솔창에 바로 표시되지 않을 수 있다. (자바스크립트뿐만 아니  
라) 많은 프로그램에서 I/O 부분이 가장 느리고 중단<sup>blocking</sup>이 잦기 때문이다. 여려  
분은 이런 일들이 물밑에서 처리되고 있는 줄도 몰랐겠지만 (페이지/UI 관점에서 보  
면) 브라우저가 콘솔 I/O를 백그라운드에서 비동기적으로 처리해야 성능상 유리  
하다.

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<sup>Node.js</sup> 서버)으로 그 영역을 확장해 왔다. 실상 오늘날 로봇에서 전구에 이르기까지 자바스크립트는 거의 모든 유형의 장치에 탑재되어 있다.

그러나 환경은 달라도 “스레드<sup>thread</sup>”는 공통이다. 여러 프로그램 덩이를 시간에 따라 매 순간 한 번씩 엔진을 실행시키는 ‘이벤트 루프<sup>event loop</sup>’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<sup>callback</sup>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<sup>listening</sup>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<sup>iteration</sup>를 ‘틱’<sup>tick</sup>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<sup>interval</sup> 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<sup>async</sup>”와 “병렬<sup>parallel</sup>”은 아무렇게나 섞어 쓰는 경우가 많지만 그 의미는 완전히 다르다. 비동기는 ‘지금’과 ‘나중’ 사이의 간극에 관한 용어고 병렬은 동시에 일어나는 일들과 연관된다.

‘프로세스<sup>process</sup>’와 ‘스레드<sup>thread</sup>’는 가장 많이 쓰는 병렬 컴퓨팅<sup>parallel computing</sup> 도구로, 별개의 프로세서<sup>processor</sup>, 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<sup>parallelism</sup>과 직렬성<sup>serialism</sup>이 나뉜 스레드에서 이벤트 루프를 협동<sup>cooperation</sup>하는 형태로 공존하는 모습이다.

병렬 실행 스레드 인터리빙<sup>interleaving</sup>과 비동기 이벤트 인터리빙은 완전히 다른 수준의 단위<sup>granularity</sup>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 * 2`는 ‘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')

---

결과값은 21이다.

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

자바스크립트는 절대로 스레드 간에 데이터를 공유하는 법이 없으므로 비결정성의 수준<sup>level of nondeterminism</sup>은 문제가 되지 않는다. 하지만 그렇다고 자바스크립트 프로그램이 항상 결정적<sup>deterministic</sup>이란 소리도 아니다. 좀 전의 예제에서도 `foo()`와 `bar()`의 실행 순서에 따라 결과값이 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 마찬가지) 내부의 코드는 원자적<sup>atomic</sup>이다. 즉, 일단 `foo()`가 실행되면 이 함수 전체 코드가 실행되고 나서야 `bar()` 함수로 옮겨간다는 뜻이다. 이를 완전-실행<sup>Run-to-Completion</sup>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<sup>statement</sup> 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<sup>either-first order</sup>으로 실행<sup>01</sup>되므로 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 결과값은 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<sup>1</sup> race condition이라고 표현한다. foo ( )와 bar ( ) 중 누가 먼저 실행되나 내기하는 경합 같다는 의미에서다. 구체적으로는 a와 b의 결과값을 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 시공간<sup>Time window</sup>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<sup>operation-level</sup> 병행성(개별 프로세서의 스레드)과 상반되는 개념의 “프로세스” 수준<sup>process-level</sup>(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 읽어들이는 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 땐 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<sup>gate</sup>’이라고 불러왔다. 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 걸쇠<sup>latch</sup>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 할당한 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’<sup>cooperative concurrency</sup> 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 줄여보면,

---

```

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 덩이 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 조정할 것은 아니므로 res 배열에 어떤 순서로 결과가 저장될지 예측하긴 어렵다. 순서가 중요한 경우라면 앞서 설명한 기법, 또는 2장부터 배우게 될 다양한 기법을 끌어써야 한다.

`setTimeout(...)`은 비동기 스케줄링 꼼수<sup>hack</sup> 중 하나로, “이 함수를 현재 이벤트 루프 큐의 맨 뒤에 붙여주세요”라는 말이다.



`setTimeout(...)`은 엄밀히 말해 원소를 이벤트 루프 큐에 곧바로 삽입하는 게 아니라 타이머가 다음 기회에 이벤트를 삽입한다. 예컨대, 연속 두 번 `setTimeout(...)`을 호출해도 그 순서대로 처리되리란 보장은 없다. 따라서 이벤트 순서를 예측할 수 없는 타이머 표류<sup>timer drift</sup> 등의 다양한 상황이 연출될 수 있다. 노드JS의 `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<sup>job queue</sup>’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, 이젠 ‘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<sup>job loop</sup>’ (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 쉼수와 의도는 비슷하지만, 처리 순서 (나중에, 하지만 가끔씩 빨리)가 더 잘 정의되어 있고 순서가 확실히 보장되는 방향으로 구현되어 있다.

다음은 (쉼수 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<sup>side effect</sup>가 있는 함수 호출(특히 게터 함수<sup>getter function</sup>), ES6 프록시<sup>Proxy</sup> 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<sup>micro-metaphor</sup>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.





1장에서는 자바스크립트 비동기 프로그래밍의 용어와 개념, 그리고 모든 이벤트(비동기 함수 호출)를 관장하는, 단일-스레드(한 번에 하나씩) 방식의 이벤트 루프 큐에 대해 살펴보았다. 또 동시에 실행되는 여러 이벤트 또는 “프로세스”(작업, 함수 호출 등) 연쇄 chain 간의 관계를 설명하는 여러 가지 동시성 패턴에 대해서도 알아보았다.

1장 예제 코드의 모든 함수는 더는 나뉘지 않는 개별적인 실행 단위로, 함수 안의 문은 예측 가능한 순서대로 (컴파일러 상위 수준에서!) 실행되지만 함수 단위의 실행 순서는 이벤트(비동기 함수 호출)에 따라 달라질 수 있다.

어떤 경우든 함수는 ‘콜백callback’ 역할을 한다. 큐에서 대기 중인 코드가 처리되자마자 본 프로그램으로 “되돌아올” 목적지이기 때문이다.

의심할 바 없이 아직 콜백은 자바스크립트에서 비동기성을 표현하고 관리하는 가장 일반적인 기법이자, 사실상 자바스크립트 언어에서 가장 기본적인 비동기 패턴이다.

무수히 많은 정교하고 복잡한 자바스크립트 프로그램들이 콜백으로(물론, 1장에서 살펴본 동시 상호 작용 패턴과 함께) 개발됐다. 자바스크립트의 충실한 비동기 일개미, 콜백 함수는 소임을 잘 해왔다.

그러나 만사가 그렇듯 콜백도 단점이 있기에 더 개선된 비동기 패턴이 나오리란 기대(프라미스)로 많은 개발자가 들떠 있다.<sup>01</sup> 그런데 무엇을, 왜 추상화하는지 모

<sup>01</sup> 역자주\_프라미스(promise)의 사전적 의미가 ‘약속, 기대, 가능성’이란 점을 이용한 저자의 말장난입니다.

르고선 그 결과물을 효과적으로 활용할 수 없을 것이다.

2장에서는 콜백의 실체를 깊이 있게 살펴보고 (뒷장들과 부록 B에서 다룰) 이보다 정교한 비동기 패턴이 나오게 된 계기를 설명한다.

## 2.1 연속성

1장 도입부에 나왔던 비동기 콜백 예제로 돌아가자(포인트만 설명하려고 조금 고쳤다).

---

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

---

//A와 //B는 프로그램 전반부('지금'), //C는 프로그램 후반부('나중')에 각각 해당한다. 전반부 코드가 곧장 실행되면 비결정적indeterminate 시간 동안 중지되고, 언젠가 AJAX 호출이 끝날 때 중지되기 이전 위치로 다시 돌아와서 나머지 후반부 프로그램으로 이어진다.

다시 말해, 콜백 함수는 프로그램의 연속성continuation을 감싼wrapping/캡슐화encapsulation한 장치다.

코드를 더 간단히 하면,

---

```
// A
setTimeout( function(){
    // C
}, 1000 );
// B
```

---

잠깐만 이 프로그램이 어떻게 작동할지 (여러분이 자바스크립트 왕초보를 개인 교습한다

생각하고) 소리 내 말해보기 바란다. 자, 어서 큰 소리로 외쳐보자. 요점을 더 잘 이해하기 위한 연습이라 생각하고!

많은 독자는 아마 이런 식으로 말할 것이다. “A를 실행한 다음 1,000밀리 초 타임아웃<sup>timeout</sup>을 설정하고 타임아웃 시 C를 실행한다.” 어떤가, 비슷하지 않은가?

뭔가 이상하다 싶어 얼른 이렇게 고쳐 말하는 독자도 있으리라. “A를 실행한 다음 1,000밀리 초 타임아웃을 설정하고, B를 실행하고 타임아웃 시 C를 실행한다. “ 먼저 것보다 정확하다. 무슨 차이일까?

후자가 전자보다 더 정확하지만 둘 다 사람의 머릿속에서 돌아가는 코드와 자바스크립트 엔진이 실제로 실행하는 코드를 잘 맞추어 설명했다고 하기엔 부족하다. 바로 이렇게 미묘하고도 엄청난 단절을 피부로 느낄 수 있어야 비동기를 표현/관리하는 콜백의 결점을 납득할 수 있다.

콜백 함수 형태로 단일(또는 많은 프로그램이 그렇듯 수십 개에 이르는) 연속성을 이제 막 소개했을 뿐인데도 우리 두뇌가 생각하는 바와 코드의 작동 방식 간에 상당한 괴리감이 형성되고 그 차이가 점점 더 벌어지면서(잘 알겠지만 한 곳에 국한된 문제가 아니다) 코드는 더 이해/추론하기 어렵고 나중에 디버깅/관리가 힘들어진다.

## 2.2 두뇌는 순차적이다

주변에서 자신이 멀티태스커<sup>multitasker</sup>라고 자부하는 이들을 본 경험이 있을 것이다. 스스로 그렇게 멀티태스커를 지향하는 사람들의 행동은 그냥 웃기려고(시답잖은 ‘머리카락 만지면서 배 두드리기’ 같은 애들 놀이나), 아니면 일상생활(걸어가며 껌 씹기), 심지어는 아주 위험한 짓(운전하며 문자 메시지 보내기)까지 가지가진다.

그들은 진짜 멀티태스커일까? 인간이 정말 동시에 두 개의 의식을 갖고 의도적인 행동을 하면서 정확히 같은 순간에 두 가지 일을 한꺼번에 생각/추론할 수 있을

까? 두뇌 기능이 최고 수준에 이른다면 과연 이러한 병렬 멀티스레딩 처리가 가능할까?

뜻밖에 정답은 '아니오'다. 인간의 두뇌는 그렇게 만들어지지 않았다. 순순히 인정하고 싶진 않았지만(특히 A형 혈액형 보유자들!) 사람은 싱글 태스키<sup>singletasker</sup>에 더 가깝다. 주어진 순간에 기껏해야 한 가지밖에 생각할 수 없다.

인간의 심장 박동, 호흡, 눈 깜빡임 같은 자율 신경, 잠재의식, 무의식까지 논할 생각은 없다. 이런 생체 기능은 생명 유지를 위해 꼭 필요하지만, 사람이 의도적으로 두뇌의 일부분을 애써 할당하지는 않는다. 그래서 고맙게도 우리가 평소 3분에 15번 이상 열심히 SNS 계정을 들여다보는 동시에 두뇌는 백그라운드에서(스레드로) 중요한 작업을 수행할 수 있다.

지금 이 순간 여러분의 머릿속 최전방에는 어떤 이들이 벌어지고 있는가? 나의 경우는 보다시피 이 책을 집필하고 있다. 내가 지금 내 두뇌에서 또 다른 상위 단계의 기능을 꺼내 쓸 수 있을까? 아니라고 본다. 금방 산만해질 테니까. 지금도 마지막 두 단락을 몇 번이나 다시 보고 있으니!

키보드 입력을 하면서 친구/가족과 휴대전화 통화를 할 때처럼 멀티태스킹을 하는 것처럼 보이는 상황도 실은 우리가 아주 재빠른 콘텍스트 교환기<sup>context switcher</sup>처럼 행동하고 있을 뿐이다. 다시 말해 여러 작업 사이를 재빨리 연속적으로 왔다 갔다 하면서 각 작업을 아주 작고 짧은 덩이로 쪼개어 동시에 처리하는 것이다. 이 작업을 아주 빠리하면 외부에서는 마치 여러 작업이 병렬로 실행되고 있는 것처럼 보인다.

그리고 보니 1장에서 배운 (자바스크립트에서 일어나는 일들과 비슷한) 비동기 이벤트의 동시성 같지 않은가? 전혀 생소한 느낌이라면 1장을 다시 읽어보기 바란다!

엄청나게 복잡한 신경학 이론을 맘대로 단순화(남용)하면서까지 대략적이거나 여러분에게 전달하고 싶은 요점은, 인간의 두뇌가 이벤트 루프 큐처럼 작동한다는

사실이다.

지금 내가 두드리고 있는 글자(또는 단어) 하나하나가 단일 비동기 이벤트라고 하면 지금 이 문장 하나를 쓰는 와중에도 내 두뇌는 수십 번도 더 넘게 (감각적으로 그냥 떠오르는 생각들로 인해) 다른 이벤트의 방해 받을지도 모른다.

그럴 때마다 나는 방해 받지 않았고 다른 “프로세스”에 기회를 양보하지도 않았다 (천만다행이다, 안 그랬으면 이 책은 완성되지 못했을 테니). 그래도 가끔 난 내 두뇌가 끊임 없이 여러 가지 콘텍스트(“프로세스”) 사이를 왔다 갔다 하는 느낌이 든다. 자바스크립트 엔진도 영혼이 있다면 그런 기분이었지!?

## 2.2.1 실행 vs 계획

우리 두뇌는 자바스크립트 엔진처럼 단일-스레드 방식의 이벤트 루프 큐처럼 작동한다고 했다.

그런데 꼭 그렇게 단순하지만은 않다. 사람이 여러 가지 작업을 계획하는 방법과 두뇌가 이들을 처리하는 방식 사이엔 엄청난 차이가 있다.

다시 이 글을 쓰고 있는 나 자신을 빗대어보자. 지금 내 머릿속에 떠오르는 대략적인 계획은 내 생각에 내가 명령을 내린 일련의 포인트를 차례대로 즉 따라가면서 계속 글을 쓰는 것이다. 난 집필 중에 갑자기 그만두거나 비선형적 행동을 할 생각 따윈 조금도 없지만 실제로 나의 두뇌는 끊임없이 왔다 갔다 한다.

이렇게 처리 수준의 두뇌에선 비동기 이벤트가 일어나고 있지만 정작 우리 자신은 순차적/동기적인 방향으로 작업을 계획하는 것처럼 보인다. “난 가게로 가서 다음엔 우유를 사고 그다음엔 세탁물을 맡기러 갈 거야.”

상위 수준의 사고(계획)는 형태만으로는 별로 비동기 이벤트처럼 보이지 않는다. 실상 인간이 순전히 이벤트 단위로만 사고하는 경우는 대단히 드물다. 대신할 일을 차례대로(A, 다음 B, 다음 C) 잘 계획한 다음, A가 끝날 때까지 B를 강제로 기다리게 하고, B

가 끝날 때까지 C를 강제로 기다리게 하는 식으로 잠정적인 중단을 한다.

코드를 작성하며 개발자는 일련의 할 일을 계획한다. 좋은 개발자일수록 신중하게 계획한다(예: “난 z에 x값을 할당한 뒤 x에 y값을 할당해야 해.”).

문과 문이 이어진 동기 코드 작성은 꼭 할 일 목록을 적는 것과 비슷하다.

---

```
// (임시 변수 'z'를 통해) 'x', 'y'를 교환한다.
```

```
z = x;
```

```
x = y;
```

```
y = z;
```

---

세 할당문은  $x = y$ 가  $z = x$ 가 끝날 때까지 기다리고  $y = z$ 가  $x = y$ 가 끝날 때까지 기다리는 동기 코드다. 즉, 이들은 어느 한 문이 끝나야 다른 문이 실행되는 특정한 순서로 흘러가도록 짜인다. 이때 다행히 시시콜콜한 비동기 이벤트는 신경 쓰지 않아도 된다(만약 신경 써야 한다면 코드는 금세 견잡을 수 없이 복잡해질 것이다).

자, 이렇게 동기 코드 문은 동기적으로 사고하는 두뇌와 잘 어울리는데 비동기 코드는 어떨까?

(콜백으로) 비동기성을 표현하는 방식이 동기적인 두뇌의 사고 흐름과 잘 맞을 리 없다.

혹시 여러분 중에 오늘 할 일을 이렇게 작성하는 사람이 있는가?

“난 가게에 갈 거지만 도중에 분명히 전화가 올 거야. 그럼 난 ‘안녕, 엄마’라고 인사할 테고 엄마가 이야기를 시작하면 난 가게 주소를 GPS로 조회하겠지. 하지만 로딩 시간은 몇 초 걸리고 난 엄마 목소리가 잘 안 들려서 라디오 볼륨을 줄이는데, 그때 불현듯 재킷을 안 입고 온 게 생각이나, 밖은 얼어 죽겠는데. 그래도 일단 운전을 하며 엄마랑 통화하는데 안전 벨트 경고음이 울리면서 난 얘기하지, ‘엄마 나 지금 안전벨트 매고 있

어요!’ 아, 이제 GPS에 약도가 나온다, 이제.”

이런 식으로 하루에 무슨 일을, 어떤 순서로 계획한다는 게 말도 안 되는 소리처럼 들리겠지만 실제로 사람의 두뇌는 기능적인 수준에서 정확히 이런 식으로 움직인다. 기억하자. 멀티태스킹이 아니라, 그냥 매우 재빠른 컨텍스트 교환이다.

비동기 코드 작성이 어려운 이유(특히 콜백 외에 다른 방법이 마땅찮을 때는) 인간이 비동기 흐름을 생각하고 떠올리는 일 자체가 부자연스럽기 때문이다.

인간은 단계별<sup>step-by-step</sup>로 끊어 생각하는 경향이 있는데, 우리 손에 들려진 도구(콜백)는 동기 → 비동기로 전환된 이후론 단계별로 나타내기가 쉽지 않다.

그래서 콜백으로 비동기 자바스크립트 코드를 정확하게 작성하고 추론하기란 상당히 어렵다. 사람의 두뇌가 그렇게 회전하지 않으니 당연히 어려울 수밖에.



어떤 코드가 깨진 이유를 알려 하지 않는 것보다 애당초 코드가 잘 작동한 이유를 그냥 모르고 지나치는 태도가 더 나쁘다! 이른바 “종이 상자<sup>house of cards</sup>” 정신이다. “잘 돌아가는군. 이유는 모르지만 어쨌거나 잘 돌아가니 됐어. 더는 건드리지 마!” 사르트르는 “만약 지옥이 있다면 그것은 바로 타인이다. Hell is other people.”라는 말을 한 적 있는데 프로그래머 버전으로 바꾸면, “만약 지옥이 있다면 그건 타인의 코드다”. 내 버전으로 바꾸면, “진정한 지옥은 나 자신의 코드를 이해하지 않는 것이다”. 콜백도 그렇게 대충 넘어가는 것 중 하나다.

## 2.2.2 중첩/연쇄된 콜백

```
listen( "click", function handler(evt){
  setTimeout( function request(){
    ajax( "http://some.url.1", function response(text){
      if (text == "hello") {
        handler();
      }
      else if (text == "world") {
        request();
      }
    }
  )
})
```



```
    } );  
    }, 500) ;  
} );
```

---

이런 코드는 여러분들 눈에 익숙할 것이다. 비동기 단계(작업, “프로세스”)를 3개의 함수가 서로 중첩된 형태로 표현했다.

이른바 ‘콜백 지옥<sup>callback hell</sup>’, 또는 (중첩된 코드를 들여쓰기한 옆모습이 꼭 직각 삼각형 같아) ‘운명의 피라미드<sup>pyramid of doom</sup>’라고도 불리는 코드다.

하지만 콜백 지옥은 중첩/들여쓰기와는 무관하고 그보다 훨씬 심각한 문제를 안고 있다. 자세한 내용은 이 장 나머지 부분에서 줄곧 설명한다.

우선, 클릭 이벤트 대기 → 타이머 작동까지 대기 → AJAX 응답을 받을 때까지 대기, 순으로 진행하고 이후 처음부터 되풀이된다.

언뜻 보면 이 코드 자체의 비동기성은 순차적 두뇌 계획<sup>brain planning</sup>과 자연스럽게 잘 조화되는 것처럼 보인다.

‘지금’

---

```
listen( "..", function handler(..){  
    // ..  
} );
```

---

‘나중’

---

```
setTimeout( function request(..){  
    // ..  
}, 500) ;
```

---

더 ‘나중’

---

```
ajax( "..", function response(..){  
    // ..  
} );
```

---

결국(제일 '나중')

---

```
if ( .. ) {  
    // ..  
}  
else ..
```

---

그러나 이런 식의 선형적인 코드 추론에는 몇 가지 문제점이 있다.

첫째, 단순히 순차 실행(1번째 줄 → 2번째 줄 → ...)될 경우는 많은 경우의 수 중 하나에 불과하다. 실제 비동기 자바스크립트 프로그램에는 (어떤 함수에서 다음 함수로 갑자기 실행 흐름이 바뀌면서 머릿속에서 잘 따라가지 않으면 안 될) 갖가지 잡음<sup>noise</sup>이 섞인다. 콜백으로 가득한 코드의 비동기 흐름을 이해하는 일이 아주 불가능하진 않지만 많은 연습/훈련을 반복해도 자연스럽게 쉽게 이해할 만한 일 또한 아니다.

그런데 이 예제에서 뚜렷하지 않은, 더 심각한 오류가 있다. 다른 예제(의사 코드)를 들어보자.

---

```
doA( function(){  
    doB();  
  
    doC( function(){  
        doD();  
    } )  
  
    doE();  
} );  
  
doF();
```

---

노련한 개발자라면 대번에 실행 순서를 정확히 간파할 것이다. 그래도 처음엔 무척 골치가 아프겠지만, 마음을 가다듬고 코드를 잘 따라가면 정답이 눈에 보일 것이다. 자, 이 코드의 실행 순서는 다음과 같다.

- doA ( )
- doF ( )
- doB ( )
- doC ( )
- doE ( )
- doD ( )

여러분의 답안과 같다면 박수를 보낸다!

좋다, 혹자는 내가 일부러 함수를 헷갈리게 이름 지었다고 불평할지 모르겠다. 난 그냥 위에서 아래로 알파벳 순으로 명명했을 뿐이다. 헷갈리지 않게 다시 명명하면,

---

```
doA( function(){
    doC();

    doD( function(){
        doF();
    } )

    doE();
} );

doB();
```

---

실행 순서와 알파벳 순서가 일치하도록 바꾼 코드다. 그런데 이제 익숙해질 만도 하건만  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$  순서로 코드를 따라가는 건 아무래도 불편하게 느껴진다. 여러분도 코드 사이를 이리저리 오가며 눈알을 굴려야 할 테니 피로하지 않은가?

설령 된 대수냐고 할지 몰라도 문제의 소지가 될 만한 위험 요소가 숨어 있다. 무엇일까?

당연히 그러리란 예상을 뒤엎고 만약 doA(..)나 doD(..)가 비동기 코드가 아니라면? 오호, 이렇게 되면 순서가 달라진다. (그때그때 프로그램 상태에 따라 달라지겠지만, 어느 시점에) 두 함수가 동기화되면  $A \rightarrow C \rightarrow D \rightarrow F \rightarrow E \rightarrow B$  순으로 실행된다.

두 손으로 머리를 쥐어뜯고 있을 많은 개발자의 탄식이 멀찌감치 들리는 듯하다.

중첩이 원인일까? 비동기 흐름을 따라가기 어렵게 만드는 주범이 중첩인가? 물론 중첩이 원인 제공을 한 공범자인 건 맞다.

그러나 중첩 없이 이벤트/타임아웃/AJAX 예제를 다시 써보면,

---

```
listen( "click", handler );

function handler() {
    setTimeout( request, 500 );
}

function request(){
    ajax( "http://some.url.1", response );
}

function response(text){
    if (text == "hello") {
        handler();
    }
    else if (text == "world") {
        request();
    }
}
```

---

중첩/들여쓰기로 도배했던 이전 코드보다 알아보기는 훨씬 편하다. 하지만 어쨌

거나 콜백 지옥에 취약한 건 매한가지다. 이유는?

선형적(순차적)으로 이 코드를 추론하자면 한 함수에서 다음 함수로, 또 그다음 함수로, 시퀀스 흐름을 “따라가기” 위해 코드 베이스 전체를 널뛰기해야 한다. 그나마 이 예제는 최고의 경우를 가정한, 단순한 코드다. 짐작하다시피 실무에서 맞닥뜨리는 비동기 자바스크립트 코드는 터무니없게 뒤죽박죽 뒤섞인 경우가 드물지 않아 추론의 어려움은 그 차원이 다르다.

눈여겨볼 부분이 있다. 2, 3, 4단계를 죽 연결하여 연속 실행하고 싶는데 콜백만으로 할 수 있는 일은, 1단계 코드에 2단계 코드를, 3단계 코드에 2단계 코드를, 4단계 코드에 3단계 코드를 하드 코딩해 넣는 정도다. 반드시 2단계가 3단계를 선행한다는 고정 조건이 있을 때 하드 코딩이 나쁜 선택이라고 말할 수만은 없다.

그러나 하드 코딩은 기본적으로 부실한 코드를 양산하기에 단계가 나아가는 도중 엉뚱한 일들이 발생하여 오류가 나는 것까지 대비할 수는 없다. 가령, 2단계에서 실패하면 2단계 재시도는 물론이고 3단계는 아예 시작도 못 하거나 별도의 에러 처리 루틴으로 빠져버린다.

일일이 모든 내용을 단계별 하드 코딩하는 방법도 가능하지만, 십중팔구 다른 단계나 비동기 흐름에서는 재사용할 수 없는, 매우 반복적인 코드 낭비가 초래될 것이다.

물론 우리 두뇌는 순차적인(이것 → 저것 → 그것) 일련의 작업을 계획하는 데 익숙하지만, 두뇌의 이벤트적 특성 덕분에 흐름 제어를 복구<sup>recovery</sup>/재시도<sup>retry</sup>/분기<sup>fork</sup>하는 일은 그리 어렵지 않다. 장을 보러 간 사람이 그날 살 물건들이 적힌 종이를 집에 두고 온 걸 뒤늦게 알아차리더라도 이런 일을 미리 계획하지 않았다고 해서 세상의 종말이 오는 건 아니다. 인간의 두뇌는 이런 순간을 쉽사리 잘 융통한다. 집에 가서 종이를 들고 다시 가게로 향하는 것이다.

그러나 수작업으로 하드 코딩한 콜백은 (하드 코딩한 에러 처리 루틴도 마찬가지로) 대부

분 바람직하지 않다. 만일의 사태와 가능한 경우의 수를 죄다 나열하다간(사전 계획 pre-planning) 코드가 너무 복잡해져 버려 관리/수정이 난처해진다.

바로 이것이 콜백 지옥이다! 중첩/들여쓰기 같은 건 주의를 분산시키는 부수적인 요소일 뿐이다.

여기서 끝나지 않는다. 복수의 연속된 콜백이 서로 연쇄된 상태로 동시 발생하거나, 관문이나 걸쇠 등의 병렬 콜백으로 분기하는 단계 등은 입에 담지도 않았다. 아, OTL... 저자인 나도 머리가 아픈데 여러분은 오죽할까?

사람의 두뇌가 순차적으로 중단을 일으키며 계획하는 방식이 콜백 지향의 비동기 코드와 잘 맞지 않는다는 사실이 이제 개념적으로 이해되는가? 콜백으로 짠 비동기 코드를 동기화해서 따라가자면 두뇌가 비행기를 타고 세계 일주를 해야 할 판이다. 바로 이것이 콜백의 중요한 첫 번째 단점이다.

## 2.3 믿음성 문제

순차적인 두뇌 계획과 콜백식 비동기 자바스크립트 코드 사이의 부조화는 콜백 문제점의 일부에 불과하다. 더 심각한 문제가 있다.

콜백 함수의 개념을 프로그램의 연속(말하자면 후반부)이란 관점에서 다시 보자.

---

```
// A
ajax( "..", function(..){
    // C
} );
// B
```

---

//A와 //B는 자바스크립트 메인 프로그램의 제어를 직접 받으며 '지금' 실행되지만, //C는 다른 프로그램(여기서는 ajax(..) 함수)의 제어 하에 '나중'에 실행된다.

제어권을 주고받는 행위 때문에 프로그램이 항상 탈이 나는 건 아니다.

하지만 문제가 불거지는 주기가 길다고 하여 제어권 교환<sup>control switch</sup>이 별일이 아니라고 지레짐작해선 안 된다. 사실, 제어권 교환이야말로 콜백 중심적 설계 방식의 가장 큰(그리고 발전하기 어려운) 문제점이다. (콜백을 넘겨주는) `ajax(...)`는 개발자가 작성하는, 또는 개발자가 직접 제어할 수 있는 함수가 아니라 서드 파티가 제공한 유틸인 경우가 대부분이다.

내가 작성하는 프로그램인데도 실행 흐름은 서드 파티에 의존해야 하는 이런 상황을 제어의 역전<sup>inversion of control</sup>이라고 한다. 여러분이 짠 코드와 여러분이 원하는 기능이 구현된 서드 파티 유틸 사이에 일종의 구두 계약이 맺어지는 셈이다.

### 2.3.1 다섯 마리 콜백 이야기

제어권 교환이 뭐 그리 대수인지 모호할 수 있다. 무턱대고 믿은 대가가 어떤 것인지 다소 과장된 예를 하나 들어보겠다.

여러분이 고가의 TV를 판매하는 쇼핑몰의 전자상거래<sup>e-commerce</sup> 결제 시스템을 구축하는 담당 개발자라고 하자. 현재 결제 시스템에는 많은 페이지가 예전부터 구축되어 서비스 중이다. 마지막 페이지에서 TV를 구매하려는 고객이 “확인” 버튼을 클릭하면 (분석 추적 솔루션을 만들어 납품하는 모 회사에서 제공한) 서드 파티 함수를 호출하여 구매 정보를 추적할 수 있게 되어 있다.

이 추적 유틸은 아마 성능 문제 때문에 비동기 방식으로 코딩된 것 같은데, 그래서 호출 시 콜백 함수를 같이 넘겨야 한다. 콜백 함수가 시작되면 고객의 신용 카드를 결제하고 감사<sup>thank you</sup> 페이지로 이동하는 코드가 잇따라 실행된다.

다음과 같은 코드다.

---

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard();
```

```
displayThankyouPage();  
} );
```

---

어렵지 않다. 테스트 후 별문제 없으면 운영 서버에 업로드하면 된다. 만사 OK다!

반년이 흘렀고 별 이슈는 없었다. 머릿속에서 코드가 가물가물 흐릿한 와중에 다음 날 아침 커피숍에서 라떼를 마시고 있는데 팀장님께 다급한 호출을 받는다. “큰 일 났구먼. 지금 당장 사무실로 와주게!”

사무실에 달려가 보니 VIP 고객 한 사람이 TV를 구매했는데 똑같은 가격으로 5번이나 연속 결제됐다고 열 받아 이야기한다. 물론 고객 만족팀 담당자가 정식으로 사과하고 환불 처리는 끝난 상태다. 하지만 팀장님은 “제대로 테스트한 거 맞나?” 하며 대체 어떻게 이런 일이 일어날 수 있는지 조사하라고 지시한다.

머리가 흐리멍덩해지고 내가 작성한 코드인데도 기억이 잘 안 난다. 하지만 여러 분은 눈을 부릅뜨고 코드 숲을 헤치며 문제의 근원을 찾기 시작한다.

로그 파일을 캐보니 원인은 한 가지다. 왜 그런 것인진 몰라도 이 추적 유틸이 여러분이 작성한 콜백 함수를 한 번만 호출해야 하는데 무려 다섯 차례나 연달아 불렀다. 업체에서 배포한 매뉴얼을 뒤져봐도 이런 문제는 언급조차 되어있지 않다.

화가 나서 업체에 연락하니 담당자 역시 놀란 음성으로 개발팀에 문의하여 답변하겠노라고 약속한다. 다음날 그들이 확인한 내용이 담긴 장문의 메일을 읽고 곧바로 팀장님께 전달한다.

솔루션 업체가 납품한 코드 중 특정 조건에서 초당 한 번씩 주어진 콜백 함수를 호출하는데 최대 5초 동안 재시도하다가 타임아웃 에러를 내도록 만들어진 테스트 단계의 코드가 포함된 것이 문제를 일으켰다. 절대로 배포 단계에 들어가선 안 될 코드가 실수로 들어간 바람에 업체 측에서도 매우 당황스러웠고 거듭 죄송하다고 사과한다. 나머지 메일 내용은 오류에 대한 상세한 설명이 대부분이고 두 번 다시



같은 일이 반복되는 일은 없을 거라나 뭐라나. 흠.

그다음은?

자초지종을 설명하지만, 팀장님은 이번 사고에 대한 불쾌함을 감추지 않는다. 이제 그 업체 사람 말은 더는 못 믿겠다고 하시는데(그래서 중간에서 곤욕을 치렀다), 팀장님 말씀에 마지 못해 고개를 끄덕이면서도 여러분은 서드 파티에 의존하지 않고 이러한 취약점으로부터 결제 시스템을 보호할 방법은 없을지 골똘히 생각에 잠긴다.

몇 차례 삽질을 거듭하다 여러분은 우선 임시로 다음과 같이 코드를 수정한다. 팀원들은 이제 한시름 놓았다.

---

```
var tracked = false;

analytics.trackPurchase( purchaseData, function(){
  if (!tracked) {
    tracked = true;
    chargeCreditCard();
    displayThankyouPage();
  }
} );
```

---



1장에서도 이와 비슷한 코드를 봤다. 여러 개의 콜백 호출을 동시 처리하려고 걸쇠를 만든 것이다.

그런데 품질 관리팀 엔지니어가 반문한다. “만약 업체 측 함수가 콜백을 한 번도 호출하지 않으면 어떡하죠?” 아차, 그것까진 생각하지 못했다.

개미 굴을 파 내려가는 심정으로 콜백 호출 시 오류가 날 만한 경우의 수를 모두 따져보기로 했다. 분석 유틸이 잘못 작동할 가능성이 있는 경우를 대략 열거하면,

- (추적이 끝나기도 전에) 콜백을 너무 일찍 부른다.

- (아예 호출하지 않거나) 콜백을 너무 늦게 부른다.
- (여러분이 겪은 사고처럼!) 콜백을 너무 적게, 또는 너무 많이 부른다.
- 필요한 환경/파라미터를 정상적으로 콜백에 전달하지 못한다.
- 일어날지 모를 에러/예외를 무시한다.
- ...

다 써놓고 보니 꼭 블랙 리스트 같은데. 모든 경우별로 보완 로직을 구현해 넣는다는 게 얼마나 끔찍한 일인지 서서히 실감하기 시작한다. 그리고 추적 유틸에 넘겨주는 콜백이 전적으로 믿을 만한 지도 의문이다.

이제야 비로소 콜백 지옥의 실체를 몸소 경험하게 된 것이다.

### 2.3.2 남의 코드뿐만 아니라

별일 아닌데 왜 그리 유난스럽게 따지느냐고 반문할 독자도 있을 것이다. 그럼 사람은 아직 진정한 서드 파티 유틸 경험을 해보지 않았거나 버전이 매겨진<sup>versioned</sup> API 또는 자체 호스트 라이브러리만 써봐서 부지불식 중에 실행이 바뀌는 문제를 맞닥뜨려본 일이 없을 것이다.

생각해보자. 여러분은 (자체 코드 베이스 내에서) 이론적으로 제어하는 유틸을 정말 신뢰할 수 있는가?

이렇게도 생각해보자. 개발자는 대부분 예기치 못한 상황을 방지하고 줄이고자 내부 함수 작성 시 입력 파라미터 체크 등 나름의 방어 로직을 짜 넣는다.

입력 파라미터를 굳게 믿는다면,

---

```
function addNumbers(x,y) {
    // + 연산자는 파라미터를 문자열로 강제변환한 뒤
    // 덧붙이는 형태로 오버로딩할 수 있기 때문에
    // 전달되는 값에 따라 항상 안전한 것은 아니다.
    return x + y;
}
```

```
addNumbers( 21, 21 ); // 42
addNumbers( 21, "21" ); // "2121"
```

---

파라미터를 못 믿어 방어 코드를 넣으면,

---

```
function addNumbers(x,y) {
    // 파라미터가 숫자인지 확인한다.
    if (typeof x != "number" || typeof y != "number") {
        throw Error( "파라미터 오류!" );
    }

    // 여기까지 실행됐다면 +는 안전하게 덧셈 연산을 한 셈이다.
    return x + y;
}

addNumbers( 21, 21 ); // 42
addNumbers( 21, "21" ); // Error: "잘못된 파라미터!"
```

---

아니면 안전하면서도 좀 더 친절하게,

---

```
function addNumbers(x,y) {
    // 파라미터가 숫자인지 확인한다.
    x = Number( x );
    y = Number( y );

    // +는 안전한 덧셈 연산을 할 것이다.
    return x + y;
}

addNumbers( 21, 21 ); // 42
addNumbers( 21, "21" ); // 42
```

---

어떻게 구현하든 이런 식의 함수 입력값에 대한 체크/정규화 로직은 지극히 당연하며 우리가 이론적으로 완전히 믿고 쓰는 코드도 예외는 아니다. 어찌 보면 “믿으

라, 하지만 확인하라<sup>trust, but verify</sup>”는 프로그래밍 세계의 지정학적<sup>geopolitical</sup> 원리이기도 부합한다.

그렇다면 완전 외부 코드뿐만 아니라 개발자 본인이 좌지우지할 수 있는 코드에 포함된 비동기 함수 호출에 대해서도 같은 원리를 적용해야 할까? 물론, 마땅히 그래야 한다.

하지만 콜백 자체는 별로 도움이 안 된다. 결국, 매번 비동기적으로 부를 때마다 콜백 함수에 반복적인 관용 코드<sup>boilerplate</sup>/오버헤드<sup>overhead</sup>를 넣는 식으로 손수 필요한 장치를 만들어야 한다.

콜백의 가장 골치 아픈 부분이 바로 이렇게 미쁜 코드를 삽입했음에도 완전히 잘못 들어질 수 있는 제어의 역전 문제다.

(꼭 서드 파티 유틸을 쓰지 않더라도) 제어의 역전으로 빚어진 믿지 못할 코드를 완화할 장치가 없는 상황에서 콜백으로 코딩하면 설령 나중에 버그가 발견되어 난감해지더라도 지금 버그를 심어놓은 것이나 별다를 바 없다. 잠재적인 버그도 버그는 버그니까.

그야말로 지옥이다.

## 2.4 콜백을 구하라

지금까지 살펴본 믿음성 문제를 (전부는 아니더라도) 일부라도 해결하기 위해 기존 디자인을 변형한 콜백 체계가 있다. 콜백 패턴이 스스로 붕괴되는 것을 막기 위해 강구한, 단호하지만 불운한 노력의 일환이다.

예를 들어, 더욱 우아하게 에러를 처리하려고 분할 콜백<sup>split callback</sup>(한쪽은 성공 알림, 다른 쪽은 에러 알림) 기능을 제공하는 API가 있다.

---

```
function success(data) {
```

```

    console.log( data );
}

function failure(err) {
    console.error( err );
}

ajax( "http://some.url.1", success, failure );

```

---

이러한 API 설계에서 에러 처리기 `failure()`는 필수가 아니며, 작성하지 않으면 에러는 조용히 무시된다. 욕!



분할 콜백 디자인은 ES6 프라미스 API가 사용하는 패턴이다. 프라미스는 다음 장에서 아주 자세히 다룬다.

“에러 우선 스타일(`error-first style`)”이라는 콜백 패턴(“노드 스타일(`node style`)”이라고도 불리며 거의 모든 노드JS API에서는 관용어다) 또한 많이 쓴다. 단일 콜백 함수는 에러 객체(오류 발생 시)를 첫 번째 인자로 받는다. 성공 시 이 인자는 빈/`falsy` 객체로 채워지지만(두 번째, 세 번째... 인자는 정상 수신된 데이터(들)이다), 실패 시 `truthy` 또는 에러 객체로 세팅된다(이외에는 아무것도 전달받지 않는다).

---

```

function response(err,data) {
    // 에러인가?
    if (err) {
        console.error( err );
    }
    // 아니면 성공한 것으로 본다.
    else {
        console.log( data );
    }
}

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

```

---

두 경우 모두 다음 몇 가지 사실을 알 수 있다.

우선, 일견 믿음성 문제가 대체로 해결된 것처럼 보이지만 실상은 전혀 그렇지 않다. 원하지 않는 반복적인 호출을 방지하거나 걸러내는 콜백 기능이 전혀 없다. 더구나 이제는 성공/에러 신호를 동시에 받거나 아예 전혀 못 받을 수도 있으므로 상황별로 코딩해야 하는 부담까지 가중됐다.

또 한 가지 놓쳐서는 안 될 점이 표준적인 패턴의 모습을 띠고 있음에도 재사용 불가능한, 장황한 관용 코드라서 실제 애플리케이션 작성 시 매 콜백마다 타이핑해야 한다면 금세 지쳐버릴 것이다.

콜백을 한 번도 호출하지 않으면? 이런 경우가 중요하다면(중요할 가능성이 크다!) 이벤트를 취소하는 타임아웃을 걸어놓아야 한다. 이를테면 (개념 증명<sup>02</sup> proof-of-concept 용도로) 다음과 같은 유틸 함수를 만들 수 있다.

---

```
function timeoutify(fn,delay) {
    var intv = setTimeout( function(){
        intv = null;
        fn( new Error( "타임아웃!" ) );
    }, delay );

    return function() {
        // 타임아웃은 아직인가?
        if (intv) {
            clearTimeout( intv );
            fn.apply( this, arguments );
        }
    };
}
```

---

쓰는 방법은,

---

**02** 역자주 프로그램은 제작할 때 그 프로그램의 개념이 기술적으로 실현 가능한지 여부를 검증하는 행위를 말합니다.

---

```
// "에러 우선 스타일" 콜백 디자인
```

```
function foo(err,data) {  
    if (err) {  
        console.error( err );  
    }  
    else {  
        console.log( data );  
    }  
}  
  
ajax( "http://some.url.1", timeoutify( foo, 500 ) );
```

---

너무 일찍 콜백을 호출해도 문제다. 애플리케이션 관점에서 보면 실제로 어떤 중요한 작업을 마치기 전에 콜백을 부른 것이다. 하지만 더 넓은 시야에서 보면 유틸에 전달한 콜백을 지금(동기적), 또는 나중에(비동기적)에 시작할 주체인 유틸에 문제가 있다.

동기냐 비동기냐에 관한 비결정성은 버그를 추적하기가 거의 항상 곤욕스럽다. 어떤 사람들은 이를 가상의 광기를 유발하는 괴물, 자르고<sup>Zalgo</sup><sup>03</sup>에 비유하기도 한다. “자르고를 풀어주면 안 돼!”<sup>[Don't release Zalgo!]</sup>라는 평범한 구호를 듣고 있노라면 아주 타당한 조건 한 마디가 떠오른다. “이벤트 루프 대기열 바로 다음 차례라고 해도 예측 가능한 비동기 콜백이 될 수 있게 항상 비동기 호출을 해라.”



자르고가 궁금한 독자는 오렌 골랜<sup>Oren Golan</sup>의 “자르고를 풀어주면 안돼!”와 아이작 Z. 슐래터<sup>Isaac Z. Schlueter</sup>의 “비동기성 API 설계”를 참고하자.

다음 예제를 보자.

---

**03** 역사주\_해의 사이트에서 유행하는 합성사진으로 사람/캐릭터 등 기괴하게 변형시킨 이미지를 말하며, 보통 텅 빈 눈과 입 부분에서 피나 검은 액체가 흐르는 사진들이 주류입니다. 이 책에서는 저자의 문화적인 취향에 따라 떠올리기조차 끔찍한 대상을 자르고에 비유하고 있습니다만, 꿈자리가 뒤송송할 수 있으니 굳이 자르고가 뭔지 검색할 필요는 없다고 생각합니다.

---

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..미리 캐시된 URL..", result );
a++;
```

---

콘솔창 결과는 0(동기적 콜백 호출)일까, 1(비동기적 콜백 호출)일까? 조건에 따라 다르다.

자르고의 불가측성<sup>unpredictability</sup>이 얼마나 쉽게 자바스크립트 프로그램을 위험에 빠뜨릴 수 있는지 잘 보여주는 예다. 그래서 바보 같지만 “자르고를 풀어주면 안 돼!”라는 조언은 지극히 당연하고 유익하다. 언제나 비동기로 가자!

주어진 API가 항상 비동기로 작동할지 확신이 없다면? (개념 검증용이지만) 다음 `asyncify( ... )` 같은 유틸을 만들어 쓰면 된다.

---

```
function asyncify(fn) {
    var orig_fn = fn,
        intv = setTimeout( function(){
            intv = null;
            if (fn) fn();
        }, 0 );

    ;

    fn = null;

    return function() {
        // 비동기 차레를 지나갔다는 사실을 나타내기 위해
        // 'intv' 타이머가 기동하기도 전에 너무 빨리 발사
        if (intv) {
            fn = orig_fn.bind.apply(
                orig_fn,
                // 파라미터로 전달된 값들을 커링하면서
                // 감싸미의 'this'에 'bind(...)' 호출 파라미터를 추가한다.
                [this].concat( [].slice.call( arguments ) )
            );
        }
    };
}
```



```

        );
    }
    // 이미 비동기다.
    else {
        // 원본 함수 호출
        orig_fn.apply( this, arguments );
    }
};
}

```

---

사용법은 이렇다.

```

function result(data) {
    console.log( a );
}

var a = 0;

ajax( "..미리 캐시된 URL..", asyncify( result ) );
a++;

```

---

AJAX 요청을 개시한 상태에서 즉각 콜백을 호출하여 귀결하든지, 아니면 데이터를 다른 곳에서 가져오는 터라 나중에 비동기적으로 완료되든지, 어쨌거나 결괏값은 항상 1이 된다. `result( .. )`는 비동기적으로 부를 수밖에 없고, 따라서 `a++`는 `result( .. )` 보다 먼저 실행된다.

오, 다른 믿음성 문제도 “해결됐다”! 하지만 이 정도론 부족하다. 비대해진 관용 코드가 전체 프로젝트를 짓누를 것이다.

콜백은 늘 이런 식이다. 원하는 바를 이룰 수 있게 해주지만 그걸 손에 넣으려면 고생할 각오를 해야 하는데, 코드 추론에 투자해야 할 노력이 종종 능력의 한계를 벗어나기도 한다.

경험자들이라면 언어 자체의 내장 API나 다른 언어에서 지원하는 방법을 갈망해

왔으리라. 드디어 ES6부터 기막힌 해결책이 화려하게 데뷔하는데. 시선 고정!

## 2.5 정리하기

콜백은 자바스크립트에서 비동기성을 표현하는 기본 단위다. 그러나 자바스크립트와 더불어 점점 진화하는 비동기 프로그래밍 환경에서 콜백만으로는 충분치 않다.

첫째, 사람의 두뇌는 순차적, 중단적, 단일-스레드 방식으로 계획하는 데 익숙하지만, 콜백은 비동기 흐름을 비선형적, 비순차적인 방향으로 나타내므로 구현된 코드를 제대로 이해하기가 매우 어렵다. 추론하기 곤란한 코드는 곧 악성 버그를 품은 나쁜 코드로 이어진다.

그래서 비동기성을 좀 더 동기적, 순차적, 중단적인 모습으로, 우리 두뇌가 사고하는 방식과 유사하게 표현할 방법이 필요하다.

둘째, 이 부분이 더 중요한데, 콜백은 프로그램을 진행하기 위해 제어를 역전, 즉 제어권을 다른 파트(종종 여러분 손을 떠나 어찌할 수 없는 서드 파티 유틸)에 암시적으로 넘겨줘야 하므로 골치가 아프다. 이렇게 제어권이 넘어가면서 예상보다 더 자주 콜백을 호출하는 등 여러 가지 믿음성 문제에 봉착하게 된다.

믿음성 문제를 해결하고자 임시 로직을 짜 넣으면 당장은 모면할 순 있지만 생각만큼 구현하기가 쉽지 않은 데다 계속 그렇게 하다 보면 거칠고, 유지 보수가 어려운 코드로 변질된다. 또 나중에 버그가 발견되어 된통 당해보기 전까진 막대한 피해로부터 보호받을 마땅한 방법 또한 없다.

관용 코드를 여기저기 흠뻑리지 않고도 생성한 콜백들을 재사용할 수 있는, 그런 일반적인 해결 방안이 강구되어야 믿음성 문제를 해결할 수 있다.

아무래도 콜백을 능가하는 뭔가가 필요하다. 지금까지 콜백은 임무를 훌륭히 수행해왔지만, 미래의 자바스크립트 환경에서는 더욱 정교하고 역량 있는 비동기 패턴

이 절실하다. 3장 이후부터 바로 그러한 최신 기술을 소개할 것이다.

2장에서는 콜백으로 프로그램의 비동기성을 표현하고 동시성을 다루면 순차성 sequentiality과 믿음성 trustability이 결합되는 중요한 결합이 있음을 확인했다. 문제를 알았으니 이제 해결 방안을 궁리해보자.

먼저 제어의 역전, 허술하게 매달려 있다가 쉽게 소실되는 믿음의 문제부터 해결하자.

프로그램의 연속 실행을 감싼 콜백 함수를 다른 곳(예: 서드 파티 코드)에 전달하면 두 손 모아 별 탈 없이 정상 콜백되길 기도하는 수 밖에 없다.

“지금 하고 있는 일을 다 하고 ‘나중’에 할 일은 여기 있으니 잘 부탁하오.”

그런데 만일 제어의 역전을 되역전시킬 수 있다면? 그러니까... 프로그램의 진행을 다른 파트에 넘겨주지 않고도 개발자가 언제 작업이 끝날지 알 수 있고 그 다음에 무슨 일을 해야 할지 스스로 결정할 수 있다면...?

이러한 체계가 바로 프라미스 promise다.

프라미스는, 개발자와 명세 작성자가 함께 코딩/설계 단계에서 콜백 지옥의 실타래를 풀기 위해 지푸라기 잡는 심정으로 매달리면서 자바스크립트 세상에 혜성처럼 등장했다. 실상 자바스크립트/DOM 플랫폼에 추가된 최신 비동기 API 대부분이 모두 프라미스에 기반을 두고 개발됐다. 자, 슬슬 배움의 욕구가 솟구치지 않는가?



이 장에서는 프라미스 귀결 액션을 종종 “즉시<sup>immediately</sup>”라는 말로 표현한다. 하지만 사실 상 모든 경우에 “즉시”란 잡 큐에서 그렇단 말이지, 엄밀히 말해 동기적인 ‘지금 now’은 아니다.

### 3.1 프라미스란

개발자들은 본능적으로 “코드 먼저 봅시다!” 하며 새로운 기술/패턴을 학습하는 경향이 있다. 잘 모르는 IT 기술을 배울 때 아주 당연한 수순처럼 여긴다.

하지만 단지 API만으로는 그 이면의 추상화<sup>abstraction</sup>까지 파악하기 어렵다. 프라미스가 꼭 그렇다. API 사용법만 재빨리 익혀 사용하는 것과, API가 지향하는 바와 대상이 무엇인지 제대로 알고 쓰는 것이 얼마나 극적으로 달라질 수 있는지 뼈저리게 느끼게 해주는 도구가 바로 프라미스다.

프라미스 코드를 보여주기 전, 나는 여러분에게 프라미스 개념을 충분히 설명하고자 한다. 그래서 여러분이 머릿속에 비동기 흐름과 프라미스 이론을 잘 연결시켜 정리할 수 있도록 안내하련다.

자, 프라미스란 무엇인지 두 가지 비유를 들어보겠다.

#### 3.1.1 미래값

시나리오는 이렇다. 동네 패스트 푸드점에서 나는 치즈 버거 세트를 먹기로 한다. 카운터 점원에게 세트 메뉴를 주문하고 5,500원을 결제한다. 동시에 거래<sup>transaction</sup>가 시작된다.

주문이 밀리면 치즈 버거가 바로 바로 안 나올 때가 많다. 점원은 내게 치즈 버거 대신 주문 번호가 적힌 영수증을 건네준다. 이 주문 번호는 일종의 IOU(차입증서, “당신에게 빚을 졌어요! owe you”)로, 언젠가는 반드시 내게 치즈 버거를 주겠노라는 ‘프라미스(약속)’다.

나는 영수증을 꼭 쥐고 주문 번호를 확인한다. 이 종이 한 장이 나중엔 치즈 버거로 둔갑할 테니 걱정할 필요 없다. 많이 배고프지만!

기다리면서 다른 일을 하면 된다. 친구한테 문자 메시지를 보낸다. “야, 같이 점심 먹을래? 난 치즈 버거 먹을거야.”

나는 아직 받지 못한 미래의 치즈 버거가 눈 앞에 있는 양 사고한다. 내 두뇌는 이미 영수증에 적힌 주문 번호를 치즈 버거의 자리끼움<sup>placeholder</sup>으로 인식하는 모양이다. 이 자리끼움은 시간 독립적<sup>time independent</sup>인 값, 즉 미래값<sup>future value</sup>이다.

결국, 점원이 “113번 주문하신 손님이요!”라고 외치자 나는 영수증을 들고 기뻐서 카운터로 달려간다. 점원에게 영수증을 돌려주고 나는 치즈 버거 세트를 받는다.

다시 말하면, 나의 ‘미래값’이 준비되어 갖고 있던 ‘값-프라이스<sup>value-promise</sup>’를 값 자체와 교환한 셈이다.

한편 이런 상황이 벌어질 수도 있다. 내 주문 번호를 부르기에 달려갔더니만 점원은 나지막이 “죄송합니다, 손님. 확인해보니 오늘은 치즈 버거 재료가 다 떨어졌네요.”라고 하지 않는가! 고객으로서 말문이 막히는 건 잠시뿐, 미래값은 성공 아니면 실패라는 중요한 깨달음을 얻게 된다.

매번 치즈 버거 세트를 주문한 결과는, 결국 언젠가 치즈 버거를 받게 되든지, 아니면 치즈 버거 재고가 없다는 슬픈 소식을 듣고 뭐로 점심을 때워야 하나 절망에 빠지게 되든지, 둘 중 하나다.



코드에선 조금 더 복잡하다. 점원이 주문 번호를 영영 호출하지 않아 마냥 기다리게 될 수도 있기 때문이다. 이 문제는 잠시 후 다시 논의한다.

## 지금값과 나중값

너무 모호하게만 들려서 아직은 가우뚱할 것이다. 더 구체적으로 이야기해보자.

그런데 이런 식으로 프라미스 작동 원리를 소개하기 전에 (이미 여러분도 알고 있는) 미래값을 다루는 방법, 즉 콜백 이야기부터 시작하겠다.

숫자 계산 등 어떤 값을 내는 코드를 짤 때 우리는 그 값이 ‘지금’ 존재하는 구체적인 값이라는, 매우 근원적인 가정을 한다.

---

```
var x, y = 2;
```

```
console.log( x + y ); // NaN ← 'x'는 아직 세팅 전이다
```

---

이럴테면  $x + y$  연산을 할 땐 당연히  $x, y$  모두 이미 세팅된 값이라고 본다. 잠시 후 상술할 용어를 쓰면,  $x, y$  값은 이미 ‘귀결됐다’<sup>resolved</sup>고 할 수 있다.

+ 연산자가 홀로 마술사처럼  $x, y$  값의 상태를 감지하다가 모두 귀결된(준비된) 후 덧셈 연산을 해주리라 기대하는 건 무리다. 상이한 문<sup>statement</sup>들 중 어떤 문은 ‘지금’ 실행되고 다른 문은 ‘나중’에 실행되는 식으로 움직이면 프로그램이 혼돈의 늪에 빠지게 될 것이다.

만약 두 문 중 한쪽(또는 둘 다)이 아직 실행 중이면 둘의 관계는 어떻게 받아들여야 할까? 1번 문이 끝나고 나서 2번 문이 실행되는 조건이면, 1번 문이 ‘지금’ 바로 끝나 만사가 순조롭게 흘러가든지, 아니면 1번 문이 미처 끝나지 않아 2번 문은 결국 실패하거나, 두 가지 경우 중 하나일 것이다.

어쩐지 1장 내용과 비슷하게 들린다고? 아주 좋다!

“ $x$ 랑  $y$ 를 더하도록! 둘 중 하나라도 준비가 덜 됐으면 될 때까지 기다려. 모든 준비가 끝나면 곧바로 더하라고!”  $x + y$  덧셈 연산은 꼭 이런 말을 하는 셈이다.

벌써 여러분의 두뇌는 콜백으로 넘어갔을지도 모르겠다. 다음 코드를 보자.

---

```
function add(getX,getY,cb) {  
  var x, y;
```

```

getX( function(xVal){
    x = xVal;
    // 둘 다 준비됐나?
    if (y != undefined) {
        cb( x + y ); // 더해서 보내
    }
} );
getY( function(yVal){
    y = yVal;
    // 둘 다 준비됐나?
    if (x != undefined) {
        cb( x + y ); // 더해서 보내
    }
} );
}

// 'fetchX()'와 'fetchY()'는 동기/비동기 함수
add( fetchX, fetchY, function(sum){
    console.log( sum ); // 너무 쉽지?
} );

```

---

(점점 멀어지는 호루라기 소리처럼) 코드가 침몰하는 아름다움(아름다움이 결여된 아름다움)을 잠시 감상하기 바란다. 흥한 모습은 어쩔 수 없지만 이런 패턴의 비동기 코드엔 아주 중요한 메시지가 담겨있다.

예제의 *x*, *y*는 모두 미래값으로 취급한다. 그래서 (외부의) `add (..)` 함수 입장에서 *x* 또는 *y*의 값이 (아니면 둘 다) 지금 준비된 상태인지는 관심 밖이다. 다시 말해, ‘지금’과 ‘나중’을 정규화<sup>normalize</sup>한 결과, `add (..)`의 처리 결과를 예측할 수 있게 바뀐 것이다.

시간에 대해 한결같은(‘지금’과 ‘나중’에 걸친 어느 때라도 똑같이 움직이는) `add (..)` 덕분에 비동기 코드가 훨씬 추론하기 편해졌다.

요점만 정리하면, ‘지금’과 ‘나중’을 모두 일관적으로 다루려면 둘 다 ‘나중’으로 만들어 모든 작업을 비동기화하면 된다.



물론, 이런 정교하지 않은 콜백식 `callbacks-based` 접근법은 손봐야 할 부분이 많다. 어떤 값을 언제 손에 넣게 될지 미리 걱정하지 않아도 미래값을 추론할 수 있다는 사실이 얼마나 유용한지, 이제 여러분도 서서히 눈을 뜨기 시작할 것이다.

## 프라미스 값

이번에는  $x + y$  예제를 프라미스 함수로 간단히 나타내보자. 후반부에서 더 자세히 설명할 테니 지금은 헛갈리더라도 걱정하지 말자.<sup>01</sup>

---

```
function add(xPromise,yPromise) {
  // 'Promise.all([ .. ]) '는 프라미스 배열을 인자로 받아
  // 프라미스들이 모두 귀결될 때까지 기다렸다가
  // 새 프라미스를 만들어 반환하는 함수다.
  return Promise.all( [xPromise, yPromise] )

  // 프라미스가 귀결되면 'X'와 'Y' 값을 받아 더한다.
  .then( function(values){
    // 'values'는 앞에서 귀결된 프라미스가
    // 건네준 메시지 배열이다.
    return values[0] + values[1];
  } );
}

// 'fetchX()'와 'fetchY()'는 제각기 값을 가진
// 프라미스를 반환하는데, 지금 또는 나중에 준비된다.
add( fetchX(), fetchY() )

// 두 숫자의 합이 담긴 프라미스를 받는다.
// 이제 반환된 프라미스가 귀결될 때까지 대기하기 위해
// 'then(..)'을 연쇄 호출(chain-call)한다.

.then( function(sum){
  console.log( sum ); // 더 쉽다!
} );
```

---

<sup>01</sup> 역자주\_이 책을 번역하는 현재 개발자 콘솔창에서 프라미스 코드를 실행하려면 최소한 크롬 32.0, 파이어폭스 29.0, 오페라 19, 사파리 7.1 이후에 출시된 브라우저가 필요하며, 국내에서 사용자가 가장 많은 인터넷 익스플로러는 지원되지 않습니다.

이 예제에는 두 계층의 프라미스가 있다.

`fetchX()`와 `fetchY()`를 직접 호출하여 이들의 반환값(프라미스)을 `add(...)`에 전한다. 두 프라미스 속의 원래 값은 지금 또는 나중에 준비되겠지만 시점에 상관없이 각 프라미스가 동일한 결과를 내게끔 정규화한다. 덕분에 미래값 `X`, `Y`는 시간 독립적으로 추론할 수 있다.

두 번째 계층은 `add(...)`가 (`Promise.all([...])`)를 거쳐 만들어 반환한 프라미스로 `then(...)`을 호출하고 대기한다. `add(...)`가 끝나면 덧셈을 마친 미래값이 준비되어 콘솔에 출력되는데, `X`, `Y`의 미래값을 기다리는 로직은 `add(...)` 안에 숨어있다.



`add(...)`에서 `Promise.all([...])`을 호출하여 (`promiseX`와 `promiseY`가 귀결되지만 기다리는) 프라미스를 생성한다. `.then(...)`을 연쇄 호출하면 또 다른 프라미스가 생성되는데, `return values[0] + values[1]` 줄은 즉시 (덧셈 결과값으로) 귀결되므로 (마지막 부분) `add(...)` 호출 끝에서 연쇄된 `then(...)` 호출은 실제로 `Promise.all([...])`이 생성한 첫 번째 프라미스가 아닌, `add(...)`가 반환한 두 번째 프라미스에서 작동한다. 그리고 두 번째 `then(...)` 끝에서 더는 연쇄 호출을 하진 않지만 `then(...)` 역시 프라미스를 하나 더 생성하므로 뒷부분에서 이 프라미스를 사용/감지할 수 있다. 프라미스 연쇄는 잠시 후 뒷부분에서 더 자세히 설명한다.

치즈 버거 세트 주문처럼 프라미스는 이름 `fulfillment` 아닌, 버림 `rejection`으로 귀결될 수 있다.<sup>02</sup> 항상 귀결값을 프로그램이 결정짓는 이름 프라미스와는 다르게 (버림 사유 `rejection reason`라고 하는) 버림값은 프로그램 로직에 따라 직접 세팅되거나 런타임 예외에 의해 암시적으로 생겨나기도 한다.

프라미스 `then(...)` 함수는 이름 함수(예제의 함수)를 첫 번째 인자로, 버림 함수를

<sup>02</sup> 역자주\_프라미스 용어는 아직 공식적인 한글 명칭이 정해진 바 없으나, 본 역서에서는 원서의 `fulfillment`를 프라미스(약속)가 어떤 형태로든 귀결되어 이루어졌다는 의미에서 '이름', `rejection`을 프라미스(약속)가 이루어지지 못하고 버려졌다는 의미에서 '버림'이라는 순 우리말로 순화하여 번역합니다. '성공', '실패'라고 알기 쉽게 번역한 도서도 있습니다만, 원어의 사전적 의미와 미묘한 기술적 콘텍스트를 감안할 때 '이름', '버림' 쪽이 더 가깝다고 봅니다.

두 번째 인자로 각각 넘겨받는다.

---

```
add( fetchX(), fetchY() )
  .then(
    // 이룸 함수
    function(sum) {
      console.log( sum );
    },
    // 버림 함수
    function(err) {
      console.error( err ); // 이런!
    }
  );
```

---

X나 Y 조회 시 문제가 있거나 덧셈 연산이 실패하면 `add( .. )`가 반환하는 프라미스는 버려지고 `then( .. )`의 두 번째 에러 처리 콜백이 이 프라미스에서 버림값을 받는다.

프라미스는 시간 의존적인(*time-dependent*) 상태를 외부로부터 캡슐화(원래 값을 이룰지 버릴지 기다림)하기 때문에 프라미스 자체는 시간 독립적(*time-independent*)이고 그래서 타이밍, 또는 내부 결괏값에 상관없이 예측 가능한 방향으로 구성(조합)할 수 있다.

또한, 프라미스는 일단 귀결된 후에는 상태가 그대로 유지되며(즉, 귀결 시점에 '불변 값'*immutable value*이 된다) 몇 번이든 필요할 때마다 꺼내 쓸 수 있다.



프라미스는 귀결되고 나면 외부적으로 불변 상태이므로 사고로, 또는 악의적으로 변경되는 일은 없으며 안심하고 다른 파트에 전달할 수 있다. 특히 여러 파트가 프라미스 귀결 상태를 바라보고 있을 땐 더욱 그렇다. 어느 파트가 다른 파트가 바라보고 있는 프라미스 귀결에 영향을 줄 수는 없다. '불변성'*immutability* 하면 상아탑에서 통용되는 전문 용어처럼 들리지만 실은 프라미스 체계의 가장 근본적이고 중요한 부분을 차지하는, 가벼이 보아 넘겨선 안 될 핵심이다.

불변성은 프라미스를 제대로 알려면 반드시 이해해야 할, 강력하고 중요한 개념이다. 엄청난 삽질을 하면서 동일한 로직을 콜백을 조합하여 코딩해 놓아도 콜백 조합

을 끊임없이 되풀이해야 하기에 결국 임시변통일 뿐 효과적인 전략이라 할 수 없다.

프래미스는 미래값을 캡슐화하고 조합할 수 있게 해주는 손쉬운 반복 장치다.

### 3.1.2 완료 이벤트

프래미스 각각은 미래값으로서 작동하지만, 프래미스의 귀결은 비동기 작업의 여러 단계를 ‘흐름 제어(시간적으로 이것 다음 저것<sup>this-then-that</sup>)’하기 위한 체계라 볼 수 있다.

어떤 작업을 하려고 `foo(...)` 함수를 부른다 치자. 작업 내용은 구체적으로 알 수 없고 관심을 가질 이유도 없다. 바로 끝날 수도 있지만 다소 시간이 걸릴지도 모른다.

단지 `foo(...)`가 언제 끝나 다음 단계로 넘어갈 수 있을지만 알면 그만이다. 다시 말하면, 다음 단계로 진행할 수 있게끔 `foo(...)`의 완료 상태를 알림 받을 방법이 있으면 좋겠다.

전통적인 자바스크립트 사고 방식에서는 알림 자체를 하나의 이벤트로 보고 리스닝한다. `foo(...)`의 완료 이벤트<sup>completion event</sup>(또는 진행 이벤트<sup>continuation event</sup>)를 리스닝함으로써 알림 요건을 재구성하는 것이다.



‘완료<sup>completion</sup>’, ‘진행<sup>continuation</sup>’ 중 어느 쪽으로 표현할지는 바라보는 관점에 따라 다르다. `foo(...)`에서 벌어진 일에 초점을 둘 것인가, 아니면 `foo(...)` 완료 이후 일어난 일들에 초점을 둘 것인가? 둘 다 모두 정확하고 유용한 관점이다. 이벤트 알림으로 `foo(...)`가 끝났다는 걸 알 수 있고 이는 곧 다음 단계로 나아가도 좋다는 OK 신호다. 사실 이벤트 알림 용도로 넘겨주는 콜백 자체를 앞서 ‘진행’이라는 말로 불렀었다. 완료 이벤트는 `foo(...)` 자체, 즉 현 상태에 더 집중하는 의미가 있는데, 앞으로 이 책에서는 완료 이벤트라 통칭하겠다.

콜백에서의 알림은 작업부(`foo(...)`)에서 넘겨준 콜백을 호출하면 성립된다. 하지만 프래미스에서는 이 관계가 역전되어 `foo(...)`에서 이벤트를 리스닝하고 있다가 알림을 받게 되면 다음으로 진행한다. 다음 의사 코드를 보자.

---

```

foo(x) {
    // 뭔가 시간이 제법 걸리는 일을 시작한다.
}

foo( 42 )

on (foo "완료") {
    // 이제 다음 단계로 갈 수 있다!
}

on (foo "에러") {
    // 어랏, 'foo(..)'에서 뭔가 잘못됐다.
}

```

---

foo(..)를 부른 뒤 2개의 (“완료”, “에러” 이벤트를 각각 리스닝하는) 이벤트 리스너를 설정한다. foo(..)를 호출하여 나올 수 있는 결과는 완료 아니면 에러 뿐이다. 실상 foo(..)는 호출부에서 이벤트를 받아 어떻게 처리할지 알 길이 없으니 아주 멋지게 관심사가 분리(separation of concerns)된다.

이런 의사 코드를 구현하려면 자바스크립트 환경이 흑마술을 부릴 수 있어야 하는데, 이쉽게도 아직까지 그런 환경은 존재하지 않는다(다소 비현실적인 측면도 있다). 자연스런 자바스크립트 코드로 표현하면 다음과 같다.

---

```

function foo(x) {
    // 뭔가 시간이 제법 걸리는 일을 시작한다.

    // 이벤트 구독기를 생성하여 반환한다.

    return listener;
}

var evt = foo( 42 );

evt.on( "completion", function(){
    // 이제 다음 단계로 갈 수 있다!
} );

```

```
evt.on( "failure", function(err){
    // 어랏, 'foo(..)'에서 뭔가 잘못됐다!
} );
```

---

`foo(..)`는 이벤트 구독기를 생성하여 반환하도록 명시되어 있고, 여기에 호출부 코드는 두 이벤트 처리기를 각각 등록한다.

확실히 일반적인 콜백 지향 코드와 정반대인데 의도적으로 그렇게 코딩한 것이다. `foo(..)`에 콜백 함수를 넘겨주는 대신 `foo(..)`가 `evt`라는 이벤트 구독기를 반환하고 여기에 콜백 함수를 넣는다.

하지만 2장에서 콜백은 그 자신이 제어의 역전이라고 하였다. 따라서 콜백 패턴을 뒤집는다는 건 실상 역전을 역전, 곧 제어의 되역전(uninversion of control)이고 당초 우리가 바라던 대로 제어권을 호출부에 되돌려놓게 된다.

이렇게 되면 여러 파트로 나뉘어진 코드가 이벤트를 리스닝하면서 `foo(..)` 완료 시 독립적으로 알림을 받아 이후 단계를 진행할 수 있는, 아주 중요한 이점이 있다.

---

```
var evt = foo( 42 );

// 'bar(..)'는 'foo(..)'의 완료 이벤트를 리스닝한다.
bar( evt );

// 'baz(..)'도 'foo(..)'의 완료 이벤트를 리스닝한다.
baz( evt );
```

---

제어의 비역전 덕분에 더 우아하게 관심사를 분리하여 `bar(..)`, `baz(..)`는 `foo(..)` 호출에 끼어들 이유가 전혀 없다. 마찬가지로 `foo(..)` 역시 `bar(..)`, `baz(..)`가 있거나 말거나, 또는 자신이 완료되기를 누군가 기다리고 있다는 사실을 몰라도 된다.

결국 evt 객체가 분리된 관심사 간의 중립적인 중재자 역할을 수행하는 것이다.

## 프라미스 “이벤트”

짐작했을지 모르지만, 이 evt 이벤트 구독기는 프라미스와 유사하다.

프라미스식으로 앞 예제를 다시 작성하면 `foo(..)`는 프라미스 인스턴스를 생성하여 반환하고 이 프라미스를 `bar(..)`와 `baz(..)`에 전달할 것이다.



리스닝 중인 프라미스 귀걸 “이벤트”는 (예제에선 정확히 이벤트처럼 작동하지만) 엄밀히 말해서 이벤트가 아니고 “완료”나 “에러”라고 하지 않는 게 보통이다. 대신에 `then(..)`을 통한 “then” 이벤트의 등록이며, 더 정확히 말하면 “이름”, “버림” 이벤트를 등록하는 것이다(코드 어디에도 “이름”, “버림”이라 명시적으로 표현하진 않았지만).

다음 코드를 보자.

---

```
function foo(x) {  
    // 뭔가 시간이 제법 걸리는 일을 시작한다.  
  
    // 프라미스를 생성하여 반환한다.  
    return new Promise( function(resolve,reject){  
        // 결과적으로 'resolve(..)','reject(..)'  
        // 중 한쪽을 호출하게 되고 이들은 프라미스의  
        // 귀걸 콜백 함수 역할을 한다.  
    } );  
}  
  
var p = foo( 42 );  
  
bar( p );  
  
baz( p );
```

---



`new Promise( function(...){ .. } )`는 '생성자 노출<sup>07</sup>revealing constructor' 패턴의 전형적인 모습이다. 전달된 `function`은 (`then(...)` 콜백처럼 비동기적인 지연 없이) 즉시 실행되고, `resolve`, `reject`라고 이름붙인 파라미터 2개를 받는다. 이 두 파라미터가 바로 프라미스의 귀결 함수다. `resolve(...)`는 이룸을, `reject(...)`는 버림을 각각 나타낸다.

`bar(...)`, `baz(...)` 내부는 아마 다음과 같은 코드로 채워져 있을 것이다.

---

```
function bar(fooPromise) {  
    // 'foo(...)'의 완료 여부를 리스닝한다.  
    fooPromise.then(  
        function(){  
            // 'foo(...)'는 이제 'bar(...)' 작업을 한다.  
        },  
        function(){  
            // 어랏, 'foo(...)'에서 뭔가 잘못됐다.  
        }  
    );  
}  
  
// 'baz(...)'도 마찬가지로!
```

---

프라미스를 미래값으로 다루었을 때처럼 프라미스 귀결 시 어떤 메시지를 보내야 하는 건 아니다. 이 예제처럼 단지 흐름 제어 신호로 쓰일 수도 있다.

다음과 같은 방법도 있다.

---

```
function bar() {  
    // 'foo(...)'는 확실히 끝났으므로  
    // 'bar(...)' 작업을 한다.  
}
```

---

**03** 역자주\_프라미스 생성자(`new Promise`)가 내부 기능을 코드 상에 드러내기 때문에 '생성자 노출 패턴'이라고 합니다. 그러나 이 내부 기능은 해당 프라미스를 생성하는 코드에만 노출됩니다. 다시 말해, 프라미스를 이룸/버림 시 처리 로직은 프라미스를 생성하는 코드에만 노출되어 있고 이 프라미스를 사용하는 코드에서는 전혀 알 길이 없습니다.



```
function oopsBar() {
    // 어랏, 'foo(...)'에서 뭔가 잘못되어
    // 'bar(...)'는 실행되지 않는다.
}
```

```
// 'baz()'와 'oopsBaz()'도 마찬가지로!
```

```
var p = foo( 42 );

p.then( bar, oopsBar );

p.then( baz, oopsBaz );
```



이전에 프라미스식 코드를 경험한 독자라면 마지막 두 줄, `p.then(...)`; `p.then(...)`을 `p.then(...).then(...)`처럼 연쇄해서 한 줄로 쓰고 싶은 충동을 느낄지 모르지만 완전히 다른 코드이니 주의하길! 지금은 뭐가 다른건지 애매하겠지만 지금까지 살펴본 비동기 패턴과는 전혀 다른 ‘분할splitting/분기forking’라는 패턴이다. 너무 걱정 마시라! 3장 후반부에서 다시 얘기할 것이다.

프라미스 `p`를 `bar(...)`, `baz(...)`에 태워보내는 대신, `bar(...)`, `baz(...)` 두 함수의 실행 이후를 제어하기 위해 프라미스를 이용한다. 앞 예제와 가장 두드러진 차이점은 에러 처리 방식이다.

전자는 `foo(...)`의 이름/버림 여부와 관계없이 무조건 `bar(...)`를 호출하고 `foo(...)` 실행이 실패할 경우엔 자체 로직으로 처리한다. `baz(...)`도 마찬가지다.

후자는 `foo(...)` 성공 시에만 `bar(...)`를 호출하고 그 외엔 `oopsBar(...)` 함수를 호출한다. `baz(...)`도 그렇다. 어느 편이 절대 옳다고 할 수는 없으며, 상황에 따라 적절한 방식을 취사 선택하면 된다.

어쨌거나 `foo(...)`가 반환한 프라미스 `p`로 다음에 벌어질 일을 제어할 수 있다.

또한, 두 예제 모두 동일한 프라미스 `p`에 대해 `then(...)`을 두 번 호출하는 부분에서, 프라미스는 (일단 귀결되면) 똑같은 결과(이름/버림)를 영원히 유지하므로 이후

에 필요하다면 몇번이고 계속 꺼내 쓸 수 있음을 알 수 있다.

(‘지금’이든 ‘나중’이든) p가 언제 귀결되고 나면 다음 단계는 늘 똑같다.

## 3.2 데너블 덕 타이핑

프라미스 세상에서 중요한 문제는 과연 어떤 값이 진짜 프라미스인지 아닌지 어찌 확신할 수 있는가 하는 점이다. 더 직설적으로 말해 정말 프라미스처럼 작동하는 값이 맞을까?

`new Promise(...)` 구문으로 생성된 프라미스는 `p instanceof Promise`로 확인하면 될 것 같다. 하지만 불행히도 여러 가지 이유로 이 정도만으로는 불충분하다.

사실 프라미스 값은 주로 다른 브라우저 창(`iframe` 등)에서 넘겨받는데, 현재 윈도우/프레임에 있는 프라미스와는 동떨어진 그들만의 프라미스이므로 프라미스 인스턴스 체크만으로는 제대로 확인할 수 없다.

게다가 외부 라이브러리/프레임워크 중에는 ES6 프라미스가 아닌, 고유한 방법으로 구현한 프라미스를 사용할 가능성도 있다. 그리고 프라미스 따윈 있지도 않은 구식 브라우저에서 라이브러리 형태로 프라미스를 사용하는 경우도 무시할 수 없다.

이 장 뒷부분에서 프라미스 귀결 과정에 대한 설명을 듣고나면, 사이비 프라미스 값을 찾아내서 흡수하는 기능이 왜 중요한지 더 확실히 이해하게 될 것이다. 지금은 그저 퍼즐 맞추기 이론의 핵심이라고 해두자.

여하튼, 진짜 프라미스는 `then(...)` 메소드를 가진, ‘데너블<sup>thenable</sup>’이라는 객체 또는 함수를 정의하여 판별하는 것으로 규정되었다. 데너블에 해당하는 값은 무조건 프라미스 규격에 맞다고 간주하는 것이다.

어떤 값의 타입을 그 형태(어떤 프로퍼티가 있는가)를 보고 짐작하는 타입 체크<sup>type check</sup>를 일반적인 용어로는 덕 타이핑<sup>duck typing</sup>이라 한다. “오리처럼 보이는 동물

오리 소리(꽹궙)를 낸다면 오리가 분명하다”(본 시리즈 [‘this와 객체 프로토타입’](#) 참고)는 것이다. 이를테면 덕 타이핑 방식으로 데너블 체크를 한다면 다음과 같다.

---

```
if (
  p !== null &&
  (
    typeof p === "object" ||
    typeof p === "function"
  ) &&
  typeof p.then === "function"
) {
  // 데너블로 간주한다!
}
else {
  // 데너블이 아니다!
}
```

---

헉! 보기 흉하게 흩어진 로직은 그렇다 치고 뭔가 심대한 문제가 꿈틀대고 있는 느낌이다.

프라이미스를 하필 `then(...)`이라는 이름의 함수가 정의된 임의의 객체/함수로 이루고 싶지만 이 객체/함수를 프라이미스/데너블로서 다룰 생각은 조금도 없다면? 아쉽게도 그렇게는 안 된다. 엔진이 데너블이라고 자동 인식하여 특별한 규칙을 적용하기 때문이다(이 장 후반부 참고).

`then(...)`이란 함수가 있었다는 사실을 미처 몰랐다고 해도 사정은 달라지지 않는다. 예를 들어,

---

```
var o = { then: function(){} };

// 'v'를 'o'의 '[[Prototype]]'에 연결한다.
var v = Object.create( o );

v.someStuff = "cool";
v.otherStuff = "not so cool";
```

```
v.hasOwnProperty( "then" ); // false
```

---

v는 언뜻 봐도 프라미스/데너블과는 거리가 멀다. 프로퍼티 몇 개 있는 평범한 객체다. 이 개발자는 다른 객체처럼 값을 전달하는 용도로 쓰고자 했을 것이다.

하지만 v는 별개의 객체 o, 그것도 하필이면 then (..) 메소드가 정의된 객체와 [[Prototype]]일 것이라곤 상상도 못했을 것이다. 따라서 데너블 덕 타이핑 감정 결과 v는 데너블로 판정받게 된다. 오, 이런!

직접적으로 의도한 코드만 그런 것이 아니다.

---

```
Object.prototype.then = function(){};
Array.prototype.then = function(){};
```

```
var v1 = { hello: "world" };
var v2 = [ "Hello", "World" ];
```

---

v1, v2 모두 데너블로 인식된다. 다른 회사 개발자가 실수로/악의적으로 then (..)을 Object.prototype, Array.prototype 또는 다른 네이티브 프로토타입<sup>native prototype</sup>에 추가해도 이를 방지하거나 막을 길은 없다. 설상가상으로, then (..) 자리에 콜백 파라미터를 하나도 호출하지 않는 함수가 세팅되면 그런 값으로 귀결된 프라미스는 조용히 무한 루프의 늪에 빠진다. 환장할 노릇이다!

그럴 일이 있겠냐고? 설마라고? 음...

ES6 이전에 이미 커뮤니티를 통해 배포된, 프라미스와는 무관한 유명 라이브러리에 우연히 then (..)이라는 이름의 메소드가 사용됐었다는 사실을 기억할 필요가 있다. 충돌을 피하려고 (제기랄) 나중에 메소드명을 바꾼 라이브러리도 있지만, 도저히 변경이 불가능하여 “프라미스식 코딩과는 호환되지 않습니다”는 불쌍한 간판을 내걸어야 했던 라이브러리도 더러 있었다.

과거에 예약어로 등록되지 않은 단어 중 then을 골라 (발음만으로는 지극히 일반적인 용도의 함수처럼 들리지만) 과거든, 현재든, 미래든, 그리고 실수든 고의로든, 어떤 값도 감히 then (...) 함수를 가지는 일이 없도록 단단히 못박아 두었어야 했었는데, 프라미스 체계의 데너블과 뒤엡키게 되면서 정말로 추적하기 어려운 버그만 양산하는 꼴이 되었다.



프라미스 판별 방법이 데너블 덕 타이핑이 유일한 건 아니다. “브랜딩branding” 또는 “안티-브랜딩anti-branding” 같은 최악의 경우를 가정한 다른 방법도 있다. 하지만 만사가 다 절망적이고 어두운 것만은 아니다. 뒤이어 나오지만 데너블 덕 타이핑이 유용할 때도 있다. 다만 프라미스 아닌 객체를 프라미스로 오판할 경우 데너블 덕 타이핑은 독이 될 수 있음을 유의하자.

### 3.3 프라미스 믿음

지금까지 비동기 코드에서 프라미스를 어떻게 활용할 수 있을지 상이한 관점에서 두 가지 공통점을 살펴보았는데, 프라미스 패턴이 선사하는 가장 중요한 특성인 ‘믿음’을 빼놓을 수는 없을 것 같다.

미래값과 완료 이벤트는 앞서 살펴보았던 코드 패턴에서 확실히 유사한 점이 있지만, 2장 “믿음성 문제”에서 제기한 바 있는 ‘제어의 역전’의 믿음성 문제를 모두 해결할 방안으로 프라미스가 설계된 원리/이유를 속 시원히 설명하기엔 부족한 감이 있다. 그러나 2장에서 처참하게 무너졌던 비동기 코딩의 신뢰를 확실히 회복시킬 만한 중요한 안전 장치가 무엇인지 곧 알게 될 것이다.

콜백만 사용한 코드의 믿음성 문제를 되짚어보자. foo(..)에 콜백을 넘긴 이후 일어날 수 있는 경우는 다음과 같다.

- 너무 일찍 콜백을 호출
- 너무 늦게 콜백을 호출 (또는 전혀 호출하지 않음)
- 너무 적게, 아니면 너무 많이 콜백을 호출

- 필요한 환경/파라미터를 정상적으로 콜백에 전달 못함
- 발생 가능한 에러/예외를 무시함

프라이미스 특성은 이와 같은 모든 일들에 대해 유용하고 되풀이하여 쓸 수 있는 해결책을 제시하게끔 설계됐다.

### 3.3.1 너무 빨리 호출

같은 작업인데도 어떤 때는 동기적으로, 어떤 때는 비동기적으로 끝나 결국 경합 조건에 이르게 되는, 자르고 현상<sup>Zalgo-like effects</sup>(2장 참고)을 일으킬 코드인지 확인하는 문제다.

프라이미스는 `(new Promise(function(resolve){ resolve(42); })` 처럼 바로 이루어져도 프라이미스의 정의 상 동기적으로 볼 수는 없으니 이 문제는 영향받을 일이 없다.

따라서 `then(...)`을 호출하면 프라이미스가 이미 귀결된 이후라 해도 `then(...)`에 건넨 콜백은 항상 비동기적으로만 부른다(자세한 내용은 1장 42 페이지 “잡” 참고).

굳이 `setTimeout(...,0)` 같은 함수는 쓸 필요가 없다. 프라이미스는 자르고를 알아서 예방한다.

### 3.3.2 너무 늦게 호출

방금 전 내용과 비슷하다. 프라이미스 `then(...)`에 등록한 콜백은 새 프라이미스가 생성되면서 `resolve(...)`, `reject(...)` 중 어느 한쪽은 자동 호출하도록 스케줄링된다. 이렇게 스케줄링된 두 콜백은 다음 비동기 시점에 예상대로 실행될 것이다.

동기적인 관찰은 불가능하므로 어떤 동기적인 작업 연쇄가 실제로 예정된 다른

콜백의 실행을 지연시키는 방향으로 움직일 수는 없다. 즉, 프라미스가 귀결되면 `then (..)`에 등록된 콜백들이 그 다음 비동기 기회가 찾아왔을 때 순서대로 실행되며, 어느 한 콜백 내부에서 다른 콜백의 호출에 영향을 주거나 지연시킬 일은 있을 수 없다.

예를 들면,

---

```
p.then( function(){
  p.then( function(){
    console.log( "C" );
  } );
  console.log( "A" );
} );
p.then( function(){
  console.log( "B" );
} );
// A B C
```

---

여기서 프라미스 작동 원리 덕분에 “C”가 툇 끼어들어 “B”를 앞지를 가능성은 없다.

### 프라미스 스케줄링의 기벽

그러나 프라미스 스케줄링에는 묘한 부분이 있다. 별개의 두 프라미스에서 연쇄된 콜백 사이의 상대적인 실행 순서는 장담할 수 없다. 중요한 내용이므로 잠시 짚어 보자.

두 프라미스 `p1`, `p2`가 모두 귀결된 상태라면 `p1.then (..); p2.then (..);`에서 `p1` 콜백이 `p2` 콜백보다 당연히 먼저 실행되어야 할 것 같지만, 꼭 그렇지 않은 애매한 경우가 있다. 다음 코드를 보자.

---

```
var p3 = new Promise( function(resolve,reject){
  resolve( "B" );
} );
```

```

var p1 = new Promise( function(resolve,reject){
    resolve( p3 );
} );

p2 = new Promise( function(resolve,reject){
    resolve( "A" );
} );

p1.then( function(v){
    console.log( v );
} );

p2.then( function(v){
    console.log( v );
} );

// A B <← 아마 B A라고 생각했겠지?

```

---

뒤에서 더 자세히 설명하겠지만 코드를 잘 보면 p1은 즉시값<sup>immediate value</sup><sup>04</sup>으로 귀결되지 않고 다른 프라미스 p3으로 귀결되고 p3은 다시 “B” 값으로 귀결된다. 이때 p3은 p1로, 비동기적으로 풀리므로<sup>unwrap</sup> p1 콜백은 p2 콜백보다 비동기 잡 큐에서 후순위로 밀리게 된다(42 페이지 “잡” 참고).

이런 애매한 문제로 밤을 새지 않으려면 여러 프라미스에 걸친 콜백의 순서/스케줄링에 의존해선 안 된다. 사실 처음부터 다중 콜백의 순서가 문제를 일으키지는 않는 방향으로 코딩하는 편이 바람직하다. 가능하다면 피하는 게 좋다.

### 3.3.3 한번도 콜백을 안 호출

아주 흔한 경우다. 프라미스로 해결할 수 있다.

우선, 프라미스 스스로 (귀결된 이후에) 귀결 사실을 알리지 못하게 막을 방도는 없

---

<sup>04</sup> 역자주 프로그램 소스 코드에서 직접 사용한 값으로, 100, 2.84, ‘a’, “Hello World!” 같은 리터럴 값을 생각하면 됩니다.



다(자바스크립트 에러도 그렇게는 못 한다). 이름/버림 콜백이 프라미스에 모두 등록된 상태라면 프라미스 귀결 시 둘 중 하나는 반드시 부른다.

물론, 콜백 자체에 자바스크립트 에러가 나면 결과가 이상하게 나오겠지만 그래도 어쨌든 콜백은 호출된다. 콜백에서 난 에러가 묻혀서는 안 되므로 잠시 후 콜백에서 발생한 에러를 알릴 처리하는 방법을 살펴보자.

그런데 만일 프라미스 스스로 어느 쪽으로도 귀결되지 않으면? 이럴 상황일지라도 ‘경합<sup>race</sup>’이라는, 상위 수준의 추상화를 이용하면 프라미스로 해결할 수 있다.

---

// 프라미스를 타임아웃시키는 유틸

```
function timeoutPromise(delay) {
  return new Promise( function(resolve,reject){
    setTimeout( function(){
      reject( "타임아웃!" );
    }, delay );
  } );
}
```

// 'foo()'에 타임아웃을 건다.

```
Promise.race( [
  foo(), // 'foo()'를 실행
  timeoutPromise( 3000 ) // 3초를 준다.
] )
.then(
  function(){
    // 'foo(..)'가 제시간에 이루어졌다!
  },
  function(err){
    // 'foo()'가 버려졌거나 제시간에 못 마쳤다.
    // 'err'를 조사하여 원인을 파악한다.
  }
);
```

---

프라미스 타임아웃 패턴에 대한 자세한 내용은 나중에 다시 설명한다.

중요한 건, 프로그램에 `행hang`이 안 걸리게 반드시 `foo()`의 결과를 알려준다는 점이다.

### 3.3.4 너무 가끔, 너무 종종 호출

콜백의 호출 횟수는 당연히 ‘한번’이다. 따라서 “너무 가끔”은 곧 0번 부른다는 뜻이므로 앞 절(“한번도 콜백을 안 호출”) 내용과 같다.

“너무 종종” 호출하는 경우는 간단하다. 프라미스는 정의 상 단 한번만 귀결된다. 어떤 이유로 프라미스 생성 코드가 `resolve(...)`, `reject(...)` 중 하나, 또는 모두를 여러 차례 호출하려고 하면 프라미스는 오직 최초의 귀결만 취하고 이후의 시도는 조용히 무시한다.

프라미스는 딱 한 번만 귀결되기 때문에 `then(...)`에 등록된 콜백 또한 (각각) 한 번씩만 호출된다.

물론, 같은 콜백을 두 번 이상 등록하면(예: `p.then(f)`; `p.then(f)`) 그 횟수만큼 부를 것이다. 응답 함수를 꼭 한번 부른다는 호언장담도 여러분 스스로 자신의 발에 총을 쏘는 행위까지 말리지는 못한다.

### 3.3.5 파라미터/환경 전달 실패

프라미스 귀결값(이름/버림)은 딱 하나뿐이다.

명시적인 값으로 귀결되지 않으면 (자바스크립트가 대개 그렇듯) 그 값은 `undefined`로 세팅된다. 하지만 값이야 어떻든, 지금이든 나중에든, 프라미스는 모든 등록된 (해당 이름 또는 버림) 콜백으로 반드시 전해진다.

여기서 주의할 점이 있다. `resolve(...)`, `reject(...)` 함수를 부를 때 파라미터를 여러 개 넘겨도 두 번째 이후 파라미터는 그대로 무시한다. 앞에서 이야기한 보장 체계를 위반한 것처럼 보이지만, 엄밀히 따지면 프라미스 체계를 잘못 사용한

대가일 뿐이다. 다른 API 오용 사례(예: `resolve(...)`를 여러 차례 호출)로부터도 프라미스가 (약간 혼란스러운 면은 있지만) 일관적으로 작동하도록 보호 장치가 마련되어 있다.

값을 여러 개 넘기고 싶다면 배열이나 객체로 꼭 감싸야 한다.

자바스크립트 함수는 자신이 정의된 스코프의 클로저를 항상 간직하므로(본 시리즈 '[스코프와 클로저](#)' 참고) 클로저를 통해 얼마든지 계속 주변 상태에 접근할 수 있다. 물론 콜백만 사용해도 마찬가지라 프라미스만의 특징점이라고 할 순 없지만 그런 대로 믿을 만한 장치다.

### 3.3.6 에러/예외 삼키기

‘에러/예외 삼키기’는 기본적으로 앞의 요점을 고쳐 쓴 것이다. 어떤 ‘이유’(즉, 여러 메시지)로 프라미스를 버리면 그 값은 버림 콜백(들)으로 전달된다.

하지만 프라미스가 생성 중에, 또는 귀결을 기다리는 도중에 언제라도 `TypeError`, `ReferenceError` 등의 자바스크립트 에러가 나면 예외를 잡아 주어진 프라미스를 강제로 버린다.

예를 들어,

---

```
var p = new Promise( function(resolve,reject){
    foo.bar(); // 'foo'는 정의된 바 없으니 에러가 난다!
    resolve( 42 ); // 실행되지 않는다.
} );

p.then(
    function fulfilled(){
        // 실행되지 않는다.
    },
    function rejected(err){
        // 'foo.bar()' 줄에서 에러가 나므로
        // 'err'는 'TypeError' 예외 객체일 것이다.
```

```
    }  
  );  
};
```

---

foo.bar ( )에서 발생한 자바스크립트 예외는 프라미스 버림 콜백에서 잡아 대응할 수 있다.

에러 아닌 요소는 비동기적이면서 에러가 나면 동기적으로 반응하여, 앞으로 일어날지 모를 문제를 효과적으로 차단할 수 있기 때문에 매우 중요한 세부분이다. 프라미스는 자바스크립트 예외조차도 비동기적 작동으로 바꾸어 경합 조건을 상당히 줄인다.

그런데 프라미스는 이루어졌으나 이를 감지하는 코드(then (..)에 등록된 콜백)에서 자바스크립트 예외가 발생하면 어떤 일이 벌어질까? 에러가 소실될 일은 없겠지만 조금 더 깊이 들어가보면 에러 처리 방식이 다소 독특하다.

---

```
var p = new Promise( function(resolve,reject){  
    resolve( 42 );  
} );  
  
p.then(  
    function fulfilled(msg){  
        foo.bar();  
        console.log( msg ); // 실행되지 않는다.  
    },  
    function rejected(err){  
        // 결단코 실행되지 않을거야 ——;  
    }  
);
```

---

잠깐, 이 예제는 foo.bar ( ) 줄에서 난 에러가 조용히 무시됐던 코드와 비슷하지 않나? 걱정 마시길, 그렇지 않다. 더 깊은 곳에서 발생한 에러를 감지하지 못한 것이다. p.then (..)가 반환한 또 다른 프라미스에서 TypeError 예외가 나면서

버려지게 된다.

그냥 이미 정의되어 있는 에러 처리기를 호출하면 되지 않나? 얼핏 그럴 듯한 논리 같지만 프라미스가 일단 귀결되면 그 값이 불변이라는 근본 원리를 모르고 한 소리다. `p`는 이미 42란 값으로 이루어진 상태여서 나중에 `p`의 귀결 상태를 감지하는 코드에서 에러가 나도 그 상태를 버림으로 바꿀 수는 없다.

원리에도 위배되지만 실제로 그렇게 작동하면 엄청난 혼란을 야기할 수 있다. 가령, 프라미스 `p`에 `then(...)` 콜백이 여러 개 등록된 상태에서 어떤 콜백은 호출하고 어떤 콜백은 호출하지 않을 수 있다면 매우 불분명한 로직이 될 것이다.

### 3.3.7 미더운 프라미스?

마지막으로 프라미스 패턴에 기반을 두고 믿음을 쌓는 문제를 알아보자.

의심할 여지 없이 프라미스는 콜백을 완전히 없애기 위한 장치가 아니다. 단지 프라미스는 콜백을 넘겨주는 위치만 달리할 뿐이다. `foo(...)`에 콜백을 바로 넘기지 않고, `foo(...)`에서 뭔가(외관상 진짜 프라미스)를 반환받아 이 ‘뭔가’에 콜백을 전하는 것이다.

그런데 콜백만 사용할 경우보다 이렇게 하는 편이 더 미답다고 하는 이유는 뭘까? 반환받은 ‘뭔가’가 실제로 미더운 프라미스라고 어떻게 장담할 수 있을까? 이미 믿음을 갖고 있으니 미답다고 하면 너무 불안한 믿음이지 않을까?

이와 같은 의문에 종지부를 찍을 해결책은 이미 프라미스에 구현되어 있다. 프라미스 이야기에서 가장 중요한데 자주 간과되는 세부분이기도 한데, 그 주인공은 바로 ES6 프라미스 구현체에 추가된 `Promise.resolve(...)` 함수다.

즉시값, 또는 프라미스 아닌 `non-Promise` / 데너블 아닌 `non-thenable` 값을 `Promise.resolve(...)`에 건네면 이 값으로 이루어진 프라미스를 손에 넣게 된다. 따라서 다음 예제에서 `p1`과 `p2`는 똑같다.

---

```
var p1 = new Promise( function(resolve,reject){
    resolve( 42 );
} );

var p2 = Promise.resolve( 42 );
```

---

Promise.resolve(...)에 진짜 프라미스가 넘어가도 결과는 마찬가지다.

---

```
var p1 = Promise.resolve( 42 );

var p2 = Promise.resolve( p1 );

p1 === p2; // true
```

---

더욱 중요한 사실은, 프라미스가 아닌 데너블 값을 Promise.resolve(...)에 주면 일단 그 값을 풀어보고 최종적으로 프라미스 아닌 것 같은 구체적인 값이 나올 때까지 계속 풀어본다는 점이다.

데너블을 기억하는지...?

다음 코드를 보자.

---

```
var p = {
    then: function(cb) {
        cb( 42 );
    }
};

// 아주 운이 좋으면 잘 작동하겠지!
p
.then(
    function fulfilled(val){
        console.log( val ); // 42
    },
    function rejected(err){
        // 실행되지 않는다.
```

```
    }  
  );  
};
```

---

p는 데너블이지만 진짜 프라미스는 아니다. 다행히 대부분 잘 들어맞는다. 하지만 코드가 다음과 같이 바뀌면 이야기는 달라진다.

---

```
var p = {  
  then: function(cb,errcb) {  
    cb( 42 );  
    errcb( "사악한 미소" );  
  }  
};  
  
p  
.then(  
  function fulfilled(val){  
    console.log( val ); // 42  
  },  
  
  function rejected(err){  
    // 어랏, 실행되면 안 되는데...  
    console.log( err ); // 사악한 미소  
  }  
);
```

---

p는 데너블이지만 그다지 프라미스처럼 작동하지 않는다. 악의적인 의도가 있는 걸까? 아니면 단지 프라미스 작동 원리가 무시된 까닭일까? 어찌됐건 별 상관은 없다. 이 자체로 미덥지 못하다는 사실이 중요하다.

하지만 어떤 p라도 일단 `Promise.resolve(..)`에 넣으면 정규화하므로 안전한 결과를 기대할 수 있다.

---

```
Promise.resolve( p )  
.then(  
  function fulfilled(val){
```

```

        console.log( val ); // 42
    },
    function rejected(err){
        // 실행되지 않는다.
    }
);

```

---

Promise.resolve(..)는 데너블을 인자로 받아 데너블 아닌 값이 발견될 때까지 풀어봐서 믿을 만한 진짜 순종 프라미스를 즉석에서 내놓는다. 진짜 프라미스 값을 넘기면 도로 내놓으니까 믿음성을 확보하기 위해 Promise.resolve(..)를 거친다 해서 단점이 될 만한 요소가 전혀 없다.

만일 foo(..) 호출 후 반환받은 값이 문제없이 잘 작동하는 프라미스인지 확신은 못해도 데너블인 것만은 확실하다고 치자. Promise.resolve(..)는 연쇄되어 나온, 미더운 프라미스 감싸미<sup>wrapper</sup>를 내어줄 것이다.

---

// 이렇게 하지 않도록!

```

foo( 42 )
    .then( function(v){
        console.log( v );
    } );

```

// 대신 이렇게!

```

Promise.resolve( foo( 42 ) )
    .then( function(v){
        console.log( v );
    } );

```

---



Promise.resolve(..)로 일반 함수의 반환값을 (데너블이든 아니든) 감싸면 함수 호출을 정규화하여 비동기 작업으로 잘 작동하게 할 수 있다는 부수 효과도 있다. 이를테면 foo(42)가 어떤 때는 즉시값을, 어떤 때는 프라미스일 경우 Promise.resolve( foo( 42 ) )로 감싸면 항상 결괏값이 프라미스로 고정된다. 자르고를 미연에 방지하여 더 나은 코드로 개선된 것이다.



### 3.3.8 믿음 형성

이쯤이면 왜 프라미스가 믿음직한지, 견고하고 유지 보수가 용이한 소프트웨어를 제작하는 과정에서 왜 이러한 믿음이 결정적인 역할을 하는지 마음 속에서 충분히 “귀결”되었길 바란다.

믿음 없이 자바스크립트 비동기 코딩이 가능할까? 물론 가능하긴 하다. 20년 가까이 자바스크립트 개발자들은 콜백만으로 비동기 코드를 작성해왔다.

하지만 그간 믿고 써온 근본 체계가 과연 얼마나 믿을 만하고 예측성이 좋은지 의구심을 가지기 시작하면서 여러분도 콜백이 정말 불안하기 짝이 없는 사상누각이란 사실을 곧 깨닫게 될 것이다.

프라미스는 콜백에 ‘믿음’의 의미를 증강시킨 패턴으로 좀 더 타당하고 미더운 방식으로 작동한다. 이제야 콜백의 ‘제어의 역전’을 다시 역전하여 비동기 코드를 온전하게 지켜줄, 믿을 만한 체계(프라미스)에 다시 제어권을 되반환한 셈이다.

## 3.4 연쇄 흐름

앞에서 몇번 암시했던 것처럼 프라미스는 ‘이것-다음-저것’ 식의 단일-단계(single-step) 작업만을 대상으로 만들어진 체계가 아니다. 프라미스는 장난감 블록 같아서 여러 개를 길게 늘어놓으면 일련의 비동기 단계를 나타낼 수 있다.

비결은 프라미스에 내재된 다음 두 가지 작동 방식이다.

- 프라미스에 `then (..)`을 부를 때마다 생성하여 반환하는 새 프라미스를 계속 연쇄할 수 있다.
- `then (..)`의 이름 콜백 함수(첫 번째 파라미터)가 반환한 값은 어떤 값이든 자동으로 (첫 번째 지점에서) 연쇄된 프라미스의 이름으로 세팅된다.

이 두 특성의 의미를 먼저 이해하고 흐름 제어의 비동기 시퀀스를 생성할 때 프라

미스를 어떻게 활용할지 알아보자. 일단 예제 코드를 보자.

---

```
var p = Promise.resolve( 21 );

var p2 = p.then( function(v){
    console.log( v ); // 21

    // 'p2'는 이름 (결괏값 '42')
    return v * 2;
} );

// 'p2'를 연쇄한다.
p2.then( function(v){
    console.log( v ); // 42
} );
```

---

$v * 2$  (42)를 반환하면서 첫 번째 `then(...)` 호출이 만들어준 신생 프라미스 `p2`를 이룬다. `p2`의 `then(...)`을 호출하면 `return v * 2;`에서 이름값을 받는다. 물론, `p2.then(...)`은 다시 새 프라미스를 생성하므로 다른 변수(예: `p3`)에 보관해도 된다.

하지만 도중에 잠깐 임시로 쓸 변수 `p2`(`p3`, `p4`, ...)를 자꾸 선언하는 건 다소 번거롭다. 다행히 변수를 선언하지 않아도 그냥 쉽게 연쇄하면 된다.

---

```
var p = Promise.resolve( 21 );

p
  .then( function(v){
    console.log( v ); // 21

    // 연쇄된 프라미스를 이름 (결괏값은 '42')
    return v * 2;
  } )
  // 프라미스 연쇄
  .then( function(v){
    console.log( v ); // 42
  } );
```

---

비동기 시퀀스 전체적으로 봤을 때 첫 번째 `then(...)`은 1단계, 두 번째 `then(...)`은 2단계에 각각 해당된다. 몇 단계고 필요한 만큼 더 늘릴 수도 있다. 이전 단계 `then(...)`을 연쇄하면 새 프라미스가 자동으로 생성된다.

그런데 뭔가 허전한 감이 있다. 비동기적인 처리 구조 때문에 2단계가 1단계를 기다렸다 실행되게 하고 싶다면? 예제에선 바로 `return` 문을 써서 값을 반환했고 따라서 연쇄된 프라미스가 즉시 이루어졌었다.

프라미스 시퀀스가 각 단계마다 진짜 비동기적으로 작동하게 만드는 핵심은 `Promise.resolve(...)`에 넘긴 값이 어떤 최종값<sup>final value</sup>이 아닌, 프라미스/데너블일 때 `Promise.resolve(...)`의 작동 로직이다. `Promise.resolve(...)`는 진짜 프라미스를 받으면 도로 뱉어내며, 데너블을 받으면 일단 한번 풀어보고 아니면 원하는 값이 나올 때까지 재귀적으로 계속 풀어본다.

이름/버림 처리기에서 데너블/프라미스를 반환해도 마찬가지로 풀어본다.

---

```
var p = Promise.resolve( 21 );

p.then( function(v){
  console.log( v ); // 21

  // 프라미스를 생성하여 반환한다.
  return new Promise( function(resolve,reject){
    // 결괏값 '42'로 이름
    resolve( v * 2 );
  } );
} )
.p.then( function(v){
  console.log( v ); // 42
} );
```

---

42 값을 반환하는 프라미스로 감쌌지만 어차피 연쇄된 프라미스의 귀결 시 다시

풀어보므로 두 번째 `then( .. )`이 받는 값은 여전히 42다. 만약 감싼 프라미스에 비동기성을 부여해도 작동 방식은 동일하다.

---

```
var p = Promise.resolve( 21 );

p.then( function(v){
    console.log( v ); // 21

    // 프라미스를 생성하여 반환한다.
    return new Promise( function(resolve,reject){
        // 비동기성을 부여한다!
        setTimeout( function(){
            // 결괏값 '42'로 이름
            resolve( v * 2 );
        }, 100 );
    } );
} )
.p.then( function(v){
    // 이전 단계에서 100ms 있다가 실행
    console.log( v ); // 42
} );
```

---

정말 강력하지 않은가? 이런 식으로 원하는 개수만큼 비동기 단계로 구성된 시퀀스를 만들어, 필요하다면 각 단계별로 그 다음 단계로 진행을 미룰 수(또는 그 반대로) 있는 것이다.

물론, 예제처럼 단계별로 어떤 값을 꼭 전달해야 할 필요는 없다. 반환값이 명시적이지 않으면 암시적으로 `undefined`로 할당되며 프라미스가 서로 연쇄되는 방식은 변함없다. 따라서 프라미스의 귀결은 그 다음 단계로의 진행을 신호한다고 볼 수 있다.

(귀결 메시지 없이) 지연-프라미스<sup>delay-Promise</sup> 생성을 일반화시켜 단계가 많은 경우에도 재사용할 수 있는 유틸을 작성해보자.

---

```

function delay(time) {
    return new Promise( function(resolve,reject){
        setTimeout( resolve, time );
    } );
}

delay( 100 ) // 1번 단계
.then( function STEP2(){
    console.log( "2번 단계 (100ms 후)" );
    return delay( 200 );
} )
.then( function STEP3(){
    console.log( "3번 단계 (200ms 더 경과 후)" );
} )
.then( function STEP4(){
    console.log( "4번 단계 (다음 작업)" );
    return delay( 50 );
} )
.then( function STEP5(){
    console.log( "5번 단계 (50ms 더 경과 후)" );
} )
...

```

---

delay ( 200 );로 200ms 후 이루어진 프라미스를 생성하고 첫 번째 then ( .. )의 이름 콜백에서 반환하면 두 번째 then ( .. )의 프라미스가 200ms-지연 프라미스를 기다리는 식으로 작동된다.



엄밀히 보면 두 프라미스, 즉 200ms-지연 프라미스와 두 번째 then ( .. )이 연쇄된 원천 프라미스가 교환되는 것이다. 그러나 이 둘의 상태를 프라미스 체계가 자동으로 병합하므로 머릿속에서 이 두 프라미스를 하나라고 보면 알기 쉽다. 이런 관점에서 return delay(200);는 새 프라미스를 만들어 앞에서 연쇄되어 넘겨받은 프라미스를 대체하는 코드다.

하지만 솔직히 전달 메시지가 하나도 없는 지연 시퀀스는 프라미스 흐름 제어를 유용하게 활용한 사례라 볼 순 없다. 더 있음직한 시나리오를 살펴보자.

다음은 타이머 대신 AJAX 요청을 하는 예제다.

---

```
// 'ajax( {url}, {callback} )' 같은 유틸이 있다고 하자.
```

```
// 프라미스-인식형 AJAX
```

```
function request(url) {
    return new Promise( function(resolve,reject){
        // 'ajax(..)' 콜백이 이 프라미스의 'resolve(..)' 함수가 된다.
        ajax( url, resolve );
    } );
}
```

---

ajax(..) 호출의 완료를 가리키는 프라미스를 만드는 request(..) 유틸을 먼저 정의한다.

---

```
request( "http://some.url.1/" )
    .then( function(response1){
        return request( "http://some.url.2/?v=" + response1 );
    } )
    .then( function(response2){
        console.log( response2 );
    } );
```

---



프라미스-인식형(Promise-aware) 비동기 흐름 제어를 프라미스를 사용할 수 없는 (예제에서 콜백 방식의 ajax(..)처럼) 유틸로 처리해야 할 때가 드물지 않다. ES6 프라미스 체계로는 이런 패턴을 자동으로 처리할 수 없지만, 실상 거의 모든 프라미스 라이브러리가 “리프팅(lifting)”, “프라미스화(promisifying)”, 그 밖의 변형된 나름의 방식으로 이 문제를 해결한다. 일단 기법에 대한 내용은 나중에 다시 살펴보자.

request(..)로 첫 번째 URL(http://some.url.1/)을 요청하여 연쇄 1단계를 암시적으로 생성하고 첫 번째 then(..)에서 만들어진 프라미스를 연쇄한다.

response1을 받으면 이 값으로 두 번째 URL을 조합하여 다시 request(..) 요청을 하고, 그리하여 두 번째 request(..) 프라미스가 넘어오고 3단계에서 AJAX

호출이 끝나기만을 기다린다. 결국 response2가 당도하면 화면에 출력한다.

이러한 프라미스 연쇄는 다단계 비동기 시퀀스에서 흐름 제어 뿐만 아니라 단계와 단계 사이에 메시지를 전달하는 채널로도 쓰인다.

프라미스 연쇄의 어느 단계에서 문제가 발생하면...? 에러/예외는 프라미스 단위로 한정되므로 전체 연쇄 어느 곳에서 난 에러라도 모두 잡아 바로 그 지점부터 “리셋<sup>reset</sup>”을 하여 연쇄를 다시 정상 가동시킨다.

---

```
// 1단계
request( "http://some.url.1/" )

// 2단계
.then( function(response1){
    foo.bar(); // 정의되지 않았으니 에러!

    // 실행되지 않는다.
    return request( "http://some.url.2/?v=" + response1 );
} )

// 3단계
.then(
    function fulfilled(response2){
        // 실행되지 않는다.
    },
    // 에러를 잡기 위한 버림 처리기
    function rejected(err){
        console.log( err );
        // 'foo.bar()'에서 'TypeError' 발생
        return 42;
    }
)

// 4단계
.then( function(msg){
    console.log( msg ); // 42
} );
```

---

2단계에서 에러가 나면 3단계 버림 처리기가 이를 잡아 필요 시 어떤 값(42)을 반환해 다음 4단계의 프라미스가 이루어지게 한다. 이런 식으로 전체 연쇄는 다시 이름 상태로 돌아간다.



이름 처리기가 반환한 프라미스는 풀린 상태라 다음 단계가 지연될 수 있다. 버림 처리기가 반환한 프라미스도 마찬가지라서 만약 3단계에서 `return 42`; 대신 프라미스를 반환한다면 4단계는 지연될 수 있다. `then(...)`의 이름/버림 처리기 내부에서 예외가 발생하면 다음 (연쇄된) 프라미스는 이 예외로 인해 즉시 버려진다.

프라미스의 `then(...)`을 부를 때 이름 처리기만 넘기면 버림 처리기는 기본 처리기로 대체된다.

---

```
var p = new Promise( function(resolve,reject){
    reject( "허격" );
} );

var p2 = p.then(
    function fulfilled(){
        // 실행되지 않는다.
    }
    // 버림 처리기가 생략되거나 함수 아닌 다른 값이 전달되면
    // 다음과 같은 버림 처리기가 있다고 가정하여 처리한다.
    // function(err) {
    //   throw err;
    // }
);
```

---

보다시피 기본 버림 처리기는 단순히 에러를 다시 던지는 역할을 하므로 p2(연쇄된 프라미스)를 던져진 에러와 같은 버림 사유로 강제 폐기한다. 명확하게 정의된 버림 처리기를 만날 때까지 프라미스 연쇄를 타고 계속 에러가 전파되도록 만들어놓은 장치다.





프라이미스의 에러 처리는 유의해야 할 미묘한 점들이 있다. 잠시 후 뒷 절에서 자세히 설명한다.

`then(...)`에 온전한 이름 처리기를 넘기지 않을 경우에도 기본 처리기로 자동 대체된다.

---

```
var p = Promise.resolve( 42 );

p.then(
  // 이름 처리기가 생략되거나 함수 아닌 다른 값이 넘어오면
  // 다음과 같은 이름 처리기가 있다고 가정하여 처리한다.
  // function(v) {
  //   return v;
  // }
  null,
  function rejected(err){
    // 실행되지 않는다.
  }
);
```

---

기본 이름 처리기는 받은 값을 다음 단계(프라이미스)에 그대로 전하기만 한다.



`then(null, function(err){ ... })`처럼 버림만 처리하고 이름은 그냥 통과시킬 때 `catch(function(err){ ... })`로 줄여 쓰는 방법이 있다. `catch(...)`는 다음 절에서 자세히 다룬다.

흐름 제어를 연쇄할 수 있는 프라이미스 고유의 특징을 정리해보자.

- `then(...)`을 호출하면 그 결과 자동으로 새 프라이미스를 생성하여 반환한다.
- 이름/버림 처리기 안에서 어떤 값을 반환하거나 예외를 던지면 이에 따라 (연쇄 가능한) 새 프라이미스가 귀결된다.
- 이름/버림 처리기가 반환한 프라이미스는 풀린 상태로 그 귀결값이 무엇이든 간에 결국 현재의 `then(...)`에서 반환된, 연쇄된 프라이미스의 귀결값이 된다.

흐름 제어 연쇄가 유용한 것은 원래 그렇게 의도한 것이라기보단 여러 프라미스가 함께 구성(조합)되는 방법에 따른 부수 효과라고 보는 편이 가장 정확하다. 이미 수 차례 설명한 바와 같이, 프라미스가 비동기성을 정규화하고 시간 의존적인 값 상태를 캡슐화해준 덕분에 이렇게 유용한 방향으로 연쇄할 수 있게 된 것이다.

순차적 연쇄 표현(이것-다음-이것-다음-이것-...)은 2장에서 밝힌 콜백의 복잡한 난장판에 비하면 확실히 대단한 개선이다. 하지만 적잖은 관용 코드(`then(..)`과 `function(){ .. }`)를 남발해야 하는 문제는 여전하다. 4장에서는 흐름 제어를 제너레이터로 더 멋지게, 순차적으로 표현하는 비법을 소개한다.

### 3.4.1 용어 정의: 귀결, 이룸, 버림

프라미스 동굴 탐사를 떠나기 전에 귀결<sup>resolve</sup>, 이룸<sup>fulfill</sup>, 버림<sup>reject</sup> 등 혼동하기 쉬운 확실히 의미를 알아야 할 용어들을 살펴본다. 먼저 `Promise(..)` 생성자다.

---

```
var p = new Promise( function(X,Y){
    // X()는 이룸
    // Y()는 버림
} );
```

---

콜백 2개(`X, Y`)를 넘기는데, 첫 번째는 ‘보통<sup>usually</sup>’ 프라미스가 이루어졌음을, 두 번째는 ‘항상<sup>always</sup>’ 프라미스가 버려졌음을 표시하는 용도로 쓴다. 그런데 ‘보통’이라는 대체 무슨 뜻일까? 그리고 이 두 파라미터의 정확한 명명에 관하여 어떤 의미가 내포되어 있는 걸까?

어차피 개발자가 입력하는 코드고 엔진이 식별자 이름을 해석해서 모종의 의미를 부여할 리 없을 테니 기술적으로 문제될 건 없다. `foo(..)`든 `bar(..)`든 결국 기능적으로 동등하다. 하지만 어떤 단어를 사용하느냐에 따라 개발자가 코드를 바라보는 관점뿐 아니라 같은 팀 다른 개발자가 프라미스를 사고하는 방식에도 영향

을 끼칠 수 있다. 아무리 잘 조화한 비동기 코드라도 그릇되게 생각하면 스파게티 콜백 코드로 범벅된 예전에 비해 더 나을 리 없다.

따라서 이름을 어떻게 붙일까, 하는 문제가 의외로 중요하다.

두 번째 파라미터는 쉽다. 거의 모든 책에서 `reject(...)`란 이름만으로 무슨 일을 하는 함수인지 정확하게(그리고 유일하게!) 파악할 수 있으므로 `reject`라는 함수명은 탁월한 작명이다. 항시 `reject(...)`란 명칭을 사용하기 바란다.

그런데 첫 번째 파라미터는 조금 애매하다. 다른 프라미스 책에는 `resolve(...)`라고 써어있는 경우가 적잖다. 분명히 “귀결<sup>resolution</sup>”과 연관된 단어로, 많은 책에서(이 책을 포함해서) 프라미스에 최종 값/상태를 지정한다는 의미로 많이 사용한다. 나 역시 이미 벌써 여러 차례 프라미스 이름/버림을 “프라미스를 귀결하다<sup>resolve the Promise</sup>”이란 말로 표현해왔다.

하지만 프라미스 이름 전용으로 이 파라미터를 사용할 의도였다면 `resolve(...)` 보단 `fulfill(...)`이라고 해야 더 정확하지 않을까? 이 질문의 답변을 다음 두 프라미스 API 메소드를 보면서 생각해보자.

---

```
var fulfilledPr = Promise.resolve( 42 );
```

```
var rejectedPr = Promise.reject( "허격" );
```

---

`Promise.resolve(...)`는 주어진 값으로 귀결된 프라미스를 생성한다. 예제의 42는 평범한, 프라미스 아닌, 데너블 아닌 값이므로 프라미스 `fulfilledPr`은 42란 값과 함께 이루어진다. `Promise.reject("허격")`는 “허격”이란 버림 사유로 폐기된 프라미스 `rejectedPr`를 생성한다.

자, 이제 “귀결<sup>resolution</sup>”이란 단어가 (`Promise.resolve(...)`에서 그런 것처럼) 명확하게 ‘결과는 이름 아니면 버림이다’는 맥락으로 쓰일 때 그 의미가 더 분명/정확한 이

유를 설명한다.

---

```
var rejectedTh = {
  then: function(resolved,rejected) {
    rejected( "허격" );
  }
};

var rejectedPr = Promise.resolve( rejectedTh );
```

---

이미 설명했듯이 `Promise.resolve( .. )`는 진짜 프라미스를 넘기면 도로 반환하고 데너블을 넘기면 풀어본다. 다 풀린 데너블의 상태가 버림이면 `Promise.resolve( .. )`가 반환한 프라미스도 실상 버림 상태라는 소리다.

그래서 실제 결과는 이룸, 버림 둘 중 하나이기 때문에 `Promise.resolve( .. )`가 더 적절한 API 메소드명이다.

`Promise( .. )` 생성자의 첫 번째 콜백 파라미터는 (`Promise.resolve( .. )`와 마찬가지로) 데너블, 진짜 프라미스 중 하나를 풀어볼 것이다.

---

```
var rejectedPr = new Promise( function(resolve,reject){
  // 이 프라미스를 버림 프라미스로 귀결시킨다.
  resolve( Promise.reject( "허격" ) );
} );

rejectedPr.then(
  function fulfilled(){
    // 실행되지 않는다.
  },
  function rejected(err){
    console.log( err ); // "허격"
  }
);
```

---

이제 `Promise( .. )` 생성자의 첫 번째 콜백 파라미터 명칭으로 `resolve( .. )`가

적절하다는 느낌을 여러분도 가졌으리라 본다.



좀 전에 언급한 `reject(...)`는 `resolve(...)`와 달리 풀어보지 않는다. `reject(...)`에 프라미스/데너블 값을 넘기면 값 자체는 건드리지 않고 버림 사유로 세팅한다. 그럼 이어서 버림 처리기는 원래 즉시값 말고 `reject(...)`에 전해진 실제 프라미스/데너블을 수신한다.

그럼, `then(...)`에 제공할 콜백명은 (문학적으로나 코딩적으로나) 뭐라고 부르는 게 좋을까? 나라면 `fulfilled(...)`, `rejected(...)`라고 짓고 싶다.

---

```
function fulfilled(msg) {  
    console.log( msg );  
}  
  
function rejected(err) {  
    console.error( err );  
}  
  
p.then(  
    fulfilled,  
    rejected  
);
```

---

`then(...)`의 첫 번째 파라미터는 함수명만 봐도 이름을 처리한다는 의미라서 “귀결”이란 용어의 이중성<sup>duality</sup>은 필요하지 않다. 참고로 ES6 명세에도 두 콜백을 `onFulfilled(...)`, `onRejected(...)`라고 각각 명명했는데 아주 정확한 용어다.

### 3.5 에러 처리

프라미스 버림(의도적으로 `reject(...)`를 호출하거나 사고로 자바스크립트 예외가 발생하여) 이 어떻게 비동기 프로그래밍에서 합리적인 에러 처리를 할 수 있게 해주는지 지난 몇몇 예제를 통해서 알아보았다. 그 과정에서 지나쳤던 몇 가지 상세한 부분을

이 절에서 다시 살펴보자.

동기적인 `try..catch` 구문은 개발자들이 대부분 익숙한 가장 일반적인 에러 처리 형태다. 아쉽게도 `try..catch`는 동기적으로만 사용 가능하므로 비동기 코드 패턴에서는 무용지물이다.

---

```
function foo() {
  setTimeout( function(){
    baz.bar();
  }, 100 );
}

try {
  foo();
  // 나중에 'baz.bar()'에서 전역 에러가 발생한다.
}
catch (err) {
  // 실행되지 않는다.
}
```

---

`try..catch` 구문이 확실히 사용하기 좋지만 비동기 작업에선 작동하지 않는다. 어떤 식으로든 실행 환경의 추가 지원이 꼭 필요한데, 이 문제는 4장 제너레이터에서 다시 다룬다.

콜백 세계에선 에러 처리 패턴에 관한 몇 가지 표준이 있는데, 그 중 '에러-우선 콜백'(`error-first callback`) 스타일을 알아보자.

---

```
function foo(cb) {
  setTimeout( function(){
    try {
      var x = baz.bar();
      cb( null, x ); // 이룸!
    }
    catch (err) {
      cb( err );
    }
  }
}
```

```

    }, 100 );
}

foo( function(err,val){
    if (err) {
        console.error( err ); // 이런... ㅜㅜ
    }
    else {
        console.log( val );
    }
} );

```

---



이 예제에서 `try..catch`는 `baz.bar()` 호출 결과가 즉시 동기적으로 성공/실패한다는 전제 하에서만 작동한다. `baz.bar()` 함수 자체가 비동기로 작동하면 그 내부에서 발생한 에러는 잡을 수 없다.

`foo(...)` 함수에 전달한 콜백은 첫 번째 파라미터 `err`를 통해 에러 신호를 감지할 것이다. `err`가 있으면 에러가 난 거고 없으면 문제가 없었다는 뜻이다.

이런 식으로 비동기적 에러 처리를 구현할 수 있지만 여러 개를 조합하면 문제가 심각해진다. 수준이 제각각인 에러 우선 콜백이 `if` 문이 여기저기 널린 상태로 서로 뒤엉키다보면 결국 콜백 지옥을 피하기 어렵다(2장 참고).

자, 다시 `then(...)`에 넘긴 버림 처리기로 프라미스 에러를 처리하는 문제를 보자. 프라미스는 널리 알려진 ‘에러-우선 콜백’ 대신 ‘분산-콜백<sup>split-callback</sup>’ 스타일로 이름/버림 각각의 콜백을 지정하여 에러 처리를 한다.

---

```

var p = Promise.reject( "허격" );

p.then(
    function fulfilled(){
        // 실행되지 않는다.
    },
    function rejected(err){
        console.log( err ); // "허격"
    }
);

```

```
    }  
  );  
};
```

---

이러한 에러 처리 패턴은 표면적으로는 아주 명쾌한 것 같지만, 프라미스 에러 처리는 미묘한 부분이 숨겨져 있어서 완벽하게 이해하기가 쉽지 않다.

다음 코드를 보자.

---

```
var p = Promise.resolve( 42 );  
  
p.then(  
  function fulfilled(msg){  
    // 숫자에는 문자열 함수를 없으니  
    // 에러를 던질 것이다.  
    console.log( msg.toLowerCase() );  
  },  
  function rejected(err){  
    // 실행되지 않는다.  
  }  
);
```

---

`msg.toLowerCase()`에서 문법 오류가 발생했는데, 에러 처리기는 왜 이 사실을 몰랐을까? 이미 한번 설명했지만 이 에러 처리기의 소속은 프라미스 `p`고 이미 `p`는 42 값으로 이루어진 상태라서 그렇다. `p`는 불변값이므로 에러 알림은 오직 `p.then( .. )`이 반환한 프라미스만이 가능한데 여기서는 이 프라미스를 포착할 방법이 없다.

프라미스 에러 처리가 에러나기 쉬운 이유를 이제는 분명히 이해했으리라 본다. 개발자가 그렇게 의도하고 코딩할 일은 거의 없겠지만 에러가 조용히 묻혀버리기 너무 쉬운 구조다.





프라이스 API를 잘못 사용해서 프라이스 생성 과정에서 에러가 나면 그 결과는 버림 프라이스`rejected Promise`가 아니라, 바로 예외를 던진다. `new Promise(null), Promise.all(), Promise.race(42)` 등 프라이스 생성이 실패하는 대표적인 오용 사례들이 있다. 첫 단추부터 프라이스 API 사용 방법을 오해하여 프라이스를 온전히 만들지 못하면 버림 프라이스는 아무 의미가 없다.

### 3.5.1 절망의 구덩이

“프로그래밍 언어는 대부분 개발자가 사고를 치면 ‘절망의 구덩이’<sup>pit of despair</sup>에 빠져 흑독한 대가를 치르는 방향으로 설계되어 있어서 제대로 실행하려면 정신을 바짝 차리고 열심히 코딩해야 한다.” 수년 전 제프 앳우드<sup>Jeff Atwood<sup>05</sup></sup>가 남긴 말이다. 오히려 프로그램이 예상대로(성공적으로) 처리되도록 ‘성공의 구덩이’<sup>pit of success</sup>를 기본적으로 파놓고 프로그램 실행을 실패하도록 만들어야 한다는 말이다.

프라이스 에러 처리는 분명히 “절망의 구덩이” 방식으로 설계되어 있다. 그래서 기본적으로 에러가 나도 프라이스 상태에 따라 무시할 수 있다고 보기 때문에 개발자가 깜빡 잊고 상태 감지를 하지 않으면 조용히 에러는 (탄식을 머금은 채) 생을 마감한다.

사라진/버려진 프라이스님의 침묵 속에 에러 또한 함께 파묻히는 걸 막으려면 반드시 프라이스 연쇄 끝부분에 `catch(..)`를 써야 한다고 주장하는 개발자들이 있다.

---

```
var p = Promise.resolve( 42 );
```

```
p.then(
  function fulfilled(msg){
    // 숫자에 문자열 함수가 없으니
    // 에러가 날 것이다.
```

---

<sup>05</sup> 역자주\_2008년 조엘 스폴스키(Joel Spolsky)와 스택 오버플로우(Stack Overflow)를 공동 창업한, 유명한 미국의 소프트웨어 개발자이자 사업가입니다. 국내에도 ‘코딩 호러가 들려주는 진짜 소프트웨어 개발 이야기’(위키북스, 2013) 도서 등으로 잘 알려져 있습니다.

```

        console.log( msg.toLowerCase() );
    }
)
.catch( handleErrors );

```

---

버림 처리기를 따로 `then (..)`에 넘기지 않았기에 기본 버림 처리기로 대체되면서 예러는 다음 연쇄 프라미스로 그냥 전파된다. 따라서 결국 `p`로 유입된 예러 및 `p` ‘이후’ 귀결 중 발생한 예러(예제의 `msg.toLowerCase()`) 모두 `handleErrors (..)`로 들어온다.

이걸로 끝난 걸까? 아니다!

만약 `handleErrors (..)` 함수에서 예러가 난다면? 이 예러는 누가 잡을까? 실은 방치된 프라미스가 하나 더 있는데, 바로 `catch (..)`가 반환한 프라미스다. 예제는 이 프라미스에서 예러를 잡지도, 버림 처리기를 등록하지도 않았다.

그렇다고 무작정 연쇄 끝에 `catch (..)`를 하나 더 붙일 순 없다. 이 함수 역시 실패할 가능성이 있기 때문이다. 어쨌든 프라미스 연쇄의 마지막 단계에 방치된 프라미스에서 예러가 나면 잡히지 않고 매달려 있을 가능성은 (점점 낮아지겠지만) 항상 존재한다.

도저히 맞출 수 없는 수수께끼 같다.

### 3.5.2 잡히지 않은 예러 처리

완벽하게 해결하기 쉬운 문제가 아니다. (더 낫다고 얘기를 하는) 다른 접근 방법도 있다.

일부 프라미스 라이브러리는 “전역 미처리 버림<sup>global unhandled rejection</sup>” 처리기 같은 것을 등록하는 메소드를 추가하여 전역 범위로 예러를 던지는 대신 이 메소드가 대신 호출되도록 해놓았다. 그러나 잡히지 않은 예러인지 식별하기 위해 버림 직후 임의의 시간(예: 3초) 동안 타이머를 걸어놓는 식으로 구현한 것이다. 프라미스

가 버려졌으나 타이머 작동 전 등록된 에러 처리기가 없으면 앞으로도 처리기를 등록하지 않겠다는 의사로 간주되어 에러는 잡히지 않는다.

대다수 사용 패턴을 보면 프라미스 버림 시점과 이를 감지하기까지의 지연 시간이 그다지 길지 않기 때문에 이 방법은 실제로 많은 라이브러리에서 잘 통한다. 하지만 여기서 3초라는 시간 자체가 (경험에서 우러나왔던 해도) 대단히 주관적이거나 가끔은 일정 시간 프라미스가 버림 상태로 유지시켜야 할 때도 있기 때문에 잡히지 않은 에러 처리기가 모든 긍정 오류<sup>false positive</sup> (미처리 상태의 잡히지 않은 에러) 발생 시 호출되기를 바랄 사람은 없으리라.

그래서 프라미스 연쇄 끝에 `done (..)`을 붙여 완료 사실을 천명해야 한다고 조언하는 사람들도 있다. `done (..)`은 프라미스를 생성, 반환하는 함수가 아니므로 `done (..)`에 넘긴 콜백이 존재하지도 않는 연쇄된 프라미스에 문제를 알려줄 리만무하다.

그럼 무슨 일이 벌어질까? 잡히지 않은 에러 상황에서 흔히 예상할 수 있는 일들이다. 즉, `done (..)` 버림 처리기 내부에서 에러가 나면 잡히지 않은 전역 에러로 (말하자면 개발자 콘솔창에) 던진다.

---

```
var p = Promise.resolve( 42 );

p.then(
  function fulfilled(msg){
    // 숫자에 문자열 함수가 없으니
    // 에러가 날 것이다.
    console.log( msg.toLowerCase() );
  }
)
.done( null, handleErrors );

// 'handleErrors(..)'에서 예외가 발생하면
// 여기서 전역적으로 던진다.
```

---

끝없는 연쇄나 대충 정한 타임아웃에 비하면 더 매력적인 방법 같다. 하지만 가장 큰 문제점은 ES6 표준에 들어있지 않기 때문에 아무리 그럴싸해 보여도 믿을 만한 보편적인 해결 방안과는 거리가 있다.

그럼 여기서 OTL인가? 그렇지 않다.

브라우저엔 개발자가 작성하는 코드로는 불가능한 고유한 기능이 있다. 브라우저는 언제 어떤 객체가 휴지통으로 직행하여 가비지 콜렉션(garbage collection)될지 정확히 알고 추적할 수 있다. 따라서 브라우저는 프라미스 객체를 추적하면서 언제 가비지를 수거하면 될지 분명히 알고 있으며, 프라미스가 버려지면 그 사유가 논리적인, 잡히지 않은 에러이므로 개발자 콘솔창에 표시해야 할지 여부를 확실하게 결정할 수 있다.



이 책을 쓰는 현재, 크롬, 파이어폭스에선 아직 완벽하지 않지만 이런 식으로 잡히지 않은 버림 기능을 연구 중이다.

그러나 프라미스가 제대로 가비지 콜렉션되지 않으면(코딩 패턴이 뒤죽박죽이다보면 그렇게 되기 쉽다) 브라우저의 가비지 콜렉션 감지 기능은 도처에 널려있는 침묵 속의 버림 프라미스를 파악/진단하는 데 큰 도움이 될 수 없다.

다른 대안은 없을까? 있다.

### 3.5.3 성공의 구덩이

지금부터는 향후 프라미스가 변화할 방향을 제시하기 위한 이론적인 이야기들을 하고자 한다. 나는 프라미스가 분명 지금보단 많이 개선되리라고 생각하며, 이러한 변화가 ES6 프라미스의 웹 호환성을 깨뜨리지 않기 때문에 당장 ES6 다음 버전이라도 기대해볼직하다. 그렇지 않더라도 조금만 성의가 있으면 폴리필/프롤리필해서 쓸 수 있는 방법이 있다.

- 기본적으로 프라미스는 그 다음 잡/이벤트 루프 턴 시점에 에러 처리기가 등록되어 있지 않을 경우 모든 버림을 (개발자 콘솔창에) 알리도록 되어 있다.
- 감지되기 전까지 버림 프라미스의 버림 상태를 계속해서 유지하려면 `defer ( )`를 호출해서 해당 프라미스에 관한 자동 에러 알림 기능을 끈다.

프라미스가 버려지면 엔진은 기본적으로 (조용히 넘어가는 법 없이) 요란하게 개발자 콘솔창에 이 사실을 알린다. 개발자는 암시적으로 버림 전에 에러 처리기를 등록 하든지, 아니면 명시적으로 `defer ( )`를 호출하든지 하여 알림 기능을 끌 수 있다. 어느 쪽이든 긍정 오류를 제어하는 일은 개발자 몫이다.

다음 코드를 보자.

---

```
var p = Promise.reject( "허격" ).defer();

// 'foo(..)'는 프라미스-인식형 함수다.
foo( 42 )
  .then(
    function fulfilled(){
      return p;
    },
    function rejected(err){
      // 'foo(..)' 에러 처리
    }
  );
...

```

---

`p`를 생성할 때 버림 상태를 사용/감지하려면 잠시 대기해야 하므로 `defer ( )`를 호출하는 데, 이렇게 하면 전역 범위로 알림이 발생하지 않는다. `defer ( )`는 계속 연쇄할 목적으로 동일한 프라미스를 단순 반환한다.

`foo ( .. )`가 반환한 프라미스에는 곧바로 에러 처리기가 달리므로 알림 기능은 암시적으로 배제되고 전역 알림 또한 일어나지 않는다.

반면 `then ( .. )`이 반환한 프라미스엔 `defer ( )`나 에러 처리기 같은 것이 달려있

지 않아 (내부적인 귀결 처리 후) 프라미스가 버림되면 잡히지 않은 에러 형태로 개발자 콘솔창에 나타나게 될 것이다.

이러한 설계를 '성공의 구덩이'라고 한다. 대다수의 개발자들이 거의 모든 상황에서 기대하는 바, 모든 에러는 기본적으로 처리 또는 통지된다. 여러분은 처리기를 등록하거나 의도적으로 배제해야 하며 에러 처리를 '나중'에 할 테니 미루겠다는 의사를 밝혀야 한다. 특정한 경우 부가적인 책임을 지겠다고 선택하는 것이다.

이 접근 방식이 유일하게 위험한 경우는, 프라미스를 `defer()` 했으나 실제로 버림을 감지/처리하지 못했을 때다.

그러나 (성공의 구덩이가 기본 작동 방식이지만) 일부러 `defer()`까지 호출하여 절망의 구덩이를 자처해야 하는 상황이라면 자신의 실수로 높에 빠진 개발자를 도와줄 방법은 세상에 없을 듯하다.

나는 아직도 프라미스 에러 처리(ES6 이후)에 희망을 걸어본다. 관계자 여러분들은 상황을 재고하여 대안을 고민해주기 바란다. 그때까지는 여러분도 알아서 구현해 쓰든지(도전적인 과제가 되리라), 똑똑한 프라미스 라이브러리를 찾아 사용하자!



방금 설명한 에러 처리/알림 모델은 내가 만든 `asynquence` 프라미스 추상화 라이브러리(부록 A)에도 구현되어 있으니 참고하자.

### 3.6 프라미스 패턴

이미 앞에서 프라미스 연쇄(이것-다음-이것-다음-저것 식의 흐름 제어)의 시퀀스 패턴을 계속 봐왔는데, 이외에도 프라미스에 기반을 두고 좀 더 추상화한 형태로 구축 가능한 비동기 패턴의 변형이 많다. 이런 패턴들을 잘 활용하면 엄청 복잡하고 난해한 프로그램도 더 추론적이고 관리하기 쉽게 코딩하는 데 유용하다.

그 중 두 패턴은 ES6 프라미스에 바로 구현되어 있으므로 무료이고 다른 패턴 조합에 필요한 블록으로 쓸 수 있다.

### 3.6.1 Promise.all([ .. ])

비동기 시퀀스(프라미스 연쇄)는 주어진 시점에 단 한 개의 비동기 작업만 가능하다 (정확히 1단계 → 2단계 → 3단계 → ...). 그런데 2개 이상의 단계가 동시에(병렬로) 움직일 순 없을까?

복수의 병렬/동시 작업이 끝날 때까지 진행하지 않고 대기하는 (고전 프로그래밍 용어로) 관문이라는 장치가 있다. 어느 쪽이 먼저 끝나든지 모든 작업이 다 끝나야 게이트가 열리고 다음으로 넘어간다.

이런 패턴을 프라미스 API는 `all([ .. ])`에 담았다.

AJAX 2개를 동시에 요청한 뒤 순서에 상관없이 모두 완료될 때까지 기다렸다가 3번째 AJAX 요청을 하는 코드가 있다고 하자.

---

```
// 'request(..)'는 앞에서 정의한 것과 비슷한
// 프라미스-인식형 AJAX 유틸이다.

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.all( [p1,p2] )
  .then( function(msgs){
    // 'p1', 'p2' 둘 다 이루어져
    // 여기에 메시지가 전달된다.
    return request(
      "http://some.url.3/?v=" + msgs.join(",")
    );
  } )
  .then( function(msg){
    console.log( msg );
  } );
```

---

`Promise.all([ .. ])`는 보통 프라미스 인스턴스들이 담긴 배열 하나를 인자로 받고, 호출 결과 반환된 프라미스는 이름 메시지(msg)를 수신한다. 이 메시지는 (이름 순서에 상관없이) 배열에 나열한 순서대로 프라미스들을 통과하면서 얻어진 이름 메시지의 배열이다.



엄밀히 말하면 `Promise.all([..])`에 전달하는 배열은 프라미스, 데너블, 또는 즉시 값 모두 가능하다. 배열값들은 하나씩 `Promise.resolve(...)`를 통과하면서 진짜 프라미스임을 보장하고 즉시값은 해당 값을 지닌 프라미스로 정규화된다. 빈 배열을 넘기면 메인 프라미스는 바로 이루어진다.

`Promise.all([ .. ])`이 반환한 메인 프라미스는 자신의 하위 프라미스들이 모두 이루어져야 이루어질 수 있다. 단 한 개의 프라미스라도 버려지면 `Promise.all([ .. ])` 프라미스 역시 곧바로 버려지며 다른 프라미스 결과도 덩달아 무효가 된다.

프라미스마다 항상 버림/에러 처리기를 붙여넣도록 습관화하자. 특히 `Promise.all([ .. ])`이 내어준 프라미스는 더더욱 잊지말자.

### 3.6.2 `Promise.race([ .. ])`

`Promise.all([ .. ])`은 여러 프라미스를 동시에 편성하여 모두 이루어진다는 전제로 작동하는 데, “결승선을 통과한 최초의 프라미스만” 인정하고 나머지는 무시해야 할 때도 있다.

예전부터 이른바 걸쇠<sup>latch</sup>라는 패턴으로, 프라미스에서는 경합<sup>race</sup>이라고 한다.



“결승선을 통과한 1등만 상을 준다”는 비유로 보면 “경합”라는 단어가 잘 맞지만, 대개 경합 조건이라고 하는 말이 프로그램 버그를 가리키는 경우가 많아 조금 애매한 부분이 있다. 경합 조건과 `Promise.race([..])`를 혼동하지 말자.

`Promise.race([ .. ])` 역시 하나 이상의 프라미스, 데너블, 즉시값이 포함된 배열 인자 1개를 받는다. 하지만 즉시값은 결승선에서 출발하는 것이나 진배없어



만드시 승자가 될 운명이니 사실 상 즉시값과의 경합은 아무런 의미가 없다.

`Promise.all([ .. ])`처럼 `Promise.race([ .. ])` 역시 하나라도 이루어진 프라미스가 있을 경우에 이루어지고, 하나라도 버려지는 프라미스가 있으면 버려진다.



적어도 한 명의 “승자”가 있어야 경합이라 할 수 있으므로 만일 빈 배열을 인자로 넘기면 바로 귀결되지 않고 메인 프라미스 `race([..])`는 결코 귀결되지 않는다. 한 마디로 누워서 침 뱉기다! 이룸이든 버림이든, 아니면 어떤 식으로든 동기적인 에러를 던지게끔 ES6에 명시됐어야 한다. 하지만 아쉽게도 ES6 프라미스보다 먼저 개발된 라이브러리에 우선권이 있다보니 불가피하게 이런 함정이 남아 있다. 혹여 빈 배열을 넘길 생각일랑 꿈도 꾸지 말자.

앞서 동시 AJAX 예제를 이번엔 `p1, p2` 간 경합이라는 맥락에서 다시 살펴보자.

---

```
// 'request(..)'는 앞에서 정의한 것과 비슷한
// 프라미스-인식형 AJAX 유틸이다.
```

```
var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.race( [p1,p2] )
  .then( function(msg){
    // 'p1', 'p2' 중 하나는 경합의 승자가 될 것이다.
    return request(
      "http://some.url.3/?v=" + msg
    );
  } )
  .then( function(msg){
    console.log( msg );
  } );
```

---

프라미스 하나만 승자가 되기에 이룸값은 `Promise.all([ .. ])`처럼 배열이 아닌, 단일 메시지다.

## 타임아웃 경합

Promise.race([ .. ])를 이용하면 프라미스 타임아웃 패턴을 구현할 수 있다.

---

```
// 'foo()'는 프라미스-인식형 함수다.

// 앞서 정의했던 'timeoutPromise(..)'은
// 일정 시간 지연 후 버려진 프라미스를 반환한다.

// 'foo()'에 타임아웃을 건다.
Promise.race( [
    foo(), // 'foo()'를 실행한다.
    timeoutPromise( 3000 ) // 3초를 준다.
] )
.then(
    function(){
        // 'foo(..)'는 제때 이루어졌다!
    },
    function(err){
        // 'foo()'가 버려졌거나 제때 마치지 못했으니
        // 'err'를 조사하여 원인을 찾는다.
    }
);
```

---

위와 같은 타임아웃 패턴은 대부분 잘 작동한다. 몇 가지 미묘한 부분이 없지 않지만, 사실 Promise.race([ .. ])와 Promise.all([ .. ])도 마찬가지다.

## “결론”

한 가지 중요한 질문을 하고싶다. “폐기/무시된 프라미스는 어떻게 되는 걸까?” 성능 관점이 아닌(결국 대부분 가비지 콜렉션 대상이 되겠지만) 작동 관점에서(부수 효과 등)의 질문이다. 프라미스는 취소가 안 되고 (163 페이지 “프라미스는 취소 불가”에서 말했던) 외부적인 불변성<sup>external immutability</sup>에 관한 믿음을 무너뜨리면 안 되기에 그냥 조용히 묻혀버릴 뿐이다.

하지만 앞 예제에서 만약 foo( )가 어떤 자원을 사용하려고 예약한 상태인데 타

임아웃이 먼저 걸려버려 프라미스가 그대로 묻혀버리게 되면 어떨까? 이런 패턴에서 타임아웃 직후 예약된 자원을 선제적으로 다시 놓아주거나, 아니면 이 자원에 있을지 모를 부수 효과를 취소할 방법은 없을까? 그냥 foo ( )에 타임아웃이 걸렸다는 사실만이라도 기록해두고자 한다면?

그래서 일부 개발자들은 finally (..) 같은 콜백을 등록해서 프라미스 귀결 시 항상 호출하는 형태로 필요하다면 뒷정리<sup>cleanup</sup>를 할 수 있을거라 제안한다. 현재 명세에는 없지만 ES7 이후 나올지 모르니 지켜보자.

도입된다면 아마 다음과 같은 모습일 것이다.

---

```
var p = Promise.resolve( 42 );

p.then( something )
  .finally( cleanup )
  .then( another )
  .finally( cleanup );
```

---



많은 프라미스 라이브러리에서 아직 finally (..)는 (연쇄를 이어가기 위해) 새 프라미스를 만들어 반환하도록 구현되어 있다. cleanup (..) 함수가 프라미스를 반환하게 되면 결국 연쇄에 링크되므로 앞서 설명한 미처리 버림 문제로 다시 돌아가게 된다.

공식 명세의 일부가 되기 전까지는 다음과 같은 정적 헬퍼 유틸을 만들어 프라미스 귀결을 (간접 없이) 알아챌 수 있다.

---

```
// 폴리필 안전 체크
if (!Promise.observe) {
  Promise.observe = function(pr,cb) {
    // 'pr'의 귀결을 부수적으로 감지한다.
    pr.then(
      function fulfilled (msg){
        // 비동기 콜백(갑)을 스케줄링한다.

```

```

        Promise.resolve( msg ).then( cb );
    },
    function rejected(err){
        // 비동기 콜백(잡)을 스케줄링한다.
        Promise.resolve( err ).then( cb );
    }
);

// 원래 프라미스를 반환한다.
return pr;
};
}

```

---

방금 전 타임아웃 예제라면 이 유틸을 다음과 같이 쓸 수 있다.

---

```

Promise.race( [
    Promise.observe(
        foo(), // 'foo()'를 실행한다.
        function cleanup(msg){
            // 제 시간에 끝나지 않아도
            // 'foo()' 이후 뒷정리를 한다.
        }
    ),
    timeoutPromise( 3000 ) // 3초를 준다.
] )

```

---

`Promise.observe( ... )`는 다수의 프라미스들이 서로 간섭하지 않고도 완료 여부를 감지할 수 있게 해주는 유틸의 예다. 라이브러리마다 나름의 방식에 따라 구현되어 있다. 어쨌든 어쩌다가 프라미스가 조용히 잊혀지는 일이 분명히 발생하지 않도록 방법을 강구하면 된다.

### 3.6.3 `all([ .. ])/race([ .. ])`의 변형

ES6 프라미스 내장된 `Promise.all([ .. ])`, `Promise.race([ .. ])`을 변형한 패턴 중에 자주 쓰이는 것들이 있다.

- none([ .. ])

all([ .. ])과 비슷하지만 이름/버림이 정반대다. 따라서 모든 프라미스는 버려져야 하며, 버림이 이름값이 되고 이름이 버림값이 된다.

- any([ .. ])

all([ .. ])과 유사하나 버림은 모두 무시하며, (전부가 아닌) 하나만 이루어지면 된다.

- first([ .. ])

any([ .. ])의 경합과 비슷하다. 일단 최초로 프라미스가 이루어지고 난 이후엔 다른 이름/버림은 간단히 무시한다.

- last([ .. ])

first([ .. ])와 거의 같고 최후의 이름 프라미스 하나만 승자가 된다는 것만 다르다.

이런 함수들이 기본 탑재된 프라미스 추상화 라이브러리도 더러 있지만, 여러분 나름대로 race([ .. ]), all([ .. ]) 같은 프라미스 메커니즘을 활용하여 손수 정의해보는 것도 좋다.

가령, first([..])는 다음과 같이 정의할 수 있다.

---

```
// 폴리필 안전 체크
```

```
if (!Promise.first) {
  Promise.first = function(prs) {
    return new Promise( function(resolve,reject){
      // 전체 프라미스를 순회한다.
      prs.forEach( function(pr){
        // 값을 정규화한다.
        Promise.resolve( pr )
        // 어떤 프라미스가 가장 처음 이기더라도
        // 메인 프라미스를 귀결한다.
      })
    })
  }
}
```

```

        .then( resolve );
    } );
} );
};
}

```



`first(...)`를 이렇게 구현하면 모든 프라미스가 버려져도 `first(...)`는 버리지 않고 그냥 `Promise.race([])`처럼 멈춰버린다. 필요하다면 프라미스 각각의 버림을 추적하는 별도 로직을 삽입하고 모든 프라미스가 버려지면 메인 프라미스에 `reject()`를 호출하게 짜면 된다. 여러분의 숙제로 남겨둔다.

### 3.6.4 동시 순회

프라미스 리스트를 죽 순회하면서 각각에 대해 어떤 처리를 하고 싶은 경우가 있다. `forEach(...)`, `map(...)`, `some(...)`, `every(...)` 등으로 동기적 배열에서 했던 방식과 비슷한데, 프라미스별로 처리할 작업이 근본적으로 동기적이라면 이전 예제에서 `forEach(...)`를 썼던 것처럼 이들 함수만 있어도 충분하다.

그러나 처리 작업이 비동기적이거나 동시 실행될 수 있다면 (또는 그렇게 실행되어야 한다면) 많은 라이브러리에서 제공하는 비동기 버전의 유틸을 쓰면 된다.

예를 들어, (프라미스 또는 다른 유형의) 값 배열을 받는 비동기 `map(...)` 유틸과 각 값에 대해 처리할 함수(작업)를 생각해보자. `map(...)`은 각 작업별로 추출된 비동기 이름값이 (매핑 순서대로) 담겨진 배열을 이름값으로 하는 프라미스를 반환한다.

```

if (!Promise.map) {
  Promise.map = function(vals,cb) {
    // 모든 매핑된 프라미스를 기다리는 새 프라미스
    return Promise.all(
      // 참고: 'map(...)'은 값의 배열을 프라미스 배열로 변환한다.
      vals.map( function(val){
        // 'val'이 비동기적으로 매핑된 이후 귀결된 새 프라미스로 'val'을
        대체한다.

```

```

        return new Promise( function(resolve){
            cb( val, resolve );
        } );
    } )
    );
};
}

```

---



map(..)을 이렇게 구현하면 비동기 버림 신호는 줄 수 없지만 매핑시킨 콜백(cb(...))에서 동기적인 예외/에러가 발생하면 Promise.map(..)이 반환한 메인 프라미스는 버려진다.

다음은 map(..)을 (단순값 대신) 프라미스 리스트에 사용한 코드다.

---

```

var p1 = Promise.resolve( 21 );
var p2 = Promise.resolve( 42 );
var p3 = Promise.reject( "허격" );

// 리스트에 있는 값이 프라미스에 있더라도 2를 곱한다.
Promise.map( [p1,p2,p3], function(pr,done){
    // 원소 자체를 확실히 프라미스로 만든다.
    Promise.resolve( pr )
    .then(
        // 'v'로 값을 추출한다.
        function(v){
            // 이루어진 'v'를 새로운 값으로 매핑한다.
            done( v * 2 );
        },
        // 아니면, 프라미스 버림 메시지로 매핑한다.
        done
    );
} )
.then( function(vals){
    console.log( vals ); // [42,84,"허격"]
} );

```

---

## 3.7 프라미스 API 복습

이 장에서 하나씩 풀어보았던 ES6 프라미스 API를 복습하자.



지금부터 설명할 API는 ES6에 내장된 함수에 국한하지만, ES6 이전 브라우저에서도 원래 프라미스를 사용할 수 있도록 명세에 맞게 (단순히 프라미스 라이브러리를 확장한 것뿐만 아니라) 나름대로 Promise와 관련 로직을 정의한 폴리필도 있다. 내가 작성한 "Native Promise Only"가 그런 폴리필 중 하나다.

### 3.7.1 new Promise(..) 생성자

Promise(..) 생성자는 항상 new와 함께 사용하며 동기적으로/즉시 호출할 콜백 함수를 전달해야 한다. 이 함수에는 다시 프라미스를 귀결 처리할 콜백 2개를 넘기는데 resolve(..)와 reject(..)라고 명명하는 것이 보통이다.

---

```
var p = new Promise( function(resolve,reject){
    // 'resolve(..)'는 프라미스를 귀결/이룬다.
    // 'reject(..)'는 프라미스를 버린다.
} );
```

---

reject(..)는 그냥 프라미스를 버리지만, resolve(..)는 넘어온 값을 보고 이름/버림 중 한 가지로 처리한다. 즉시값, 프라미스 아닌/테너블 아닌 값이 resolve(..)에 흘러오면 이 프라미스는 해당 값으로 이루어진다.

반면, resolve(..)에 진짜 프라미스/테너블 값이 전달되면 재귀적으로 풀어보고 결국 그 최종값이 프라미스의 마지막 귀결/상태가 된다.

### 3.7.2 Promise.resolve(..)와 Promise.reject(..)

Promise.reject(..)는 이미 버려진 프라미스를 생성하는 지름길이다. 따라서 두 프라미스는 본질적으로 동등하다.



---

```
var p1 = new Promise( function(resolve,reject){
    reject( "허걱" );
} );

var p2 = Promise.reject( "허걱" );
```

---

`Promise.reject(..)`와 마찬가지로 `Promise.resolve(..)`는 이미 이루어진 프라미스를 생성하는 용도로 쓴다. `Promise.resolve(..)`는 데너블 값을 (별써 수 차례나 반복 설명했듯이) 재귀적으로 풀어보고 그 최종 귀결값(이름/버림)이 결국 반환된 프라미스에 해당된다.

---

```
var fulfilledTh = {
    then: function(cb) { cb( 42 ); }
};

var rejectedTh = {
    then: function(cb,errCb) {
        errCb( "Oops" );
    }
};

var p1 = Promise.resolve( fulfilledTh );
var p2 = Promise.resolve( rejectedTh );

// 'p1'은 이름 프라미스가 될 것이다.
// 'p2'은 버림 프라미스가 될 것이다.
```

---

`Promise.resolve(..)`에 진짜 프라미스를 넣으면 아무 일도 하지 않는다는 점을 기억하기 바란다. 따라서 종류를 모르는 값을 `Promise.resolve(..)`에 넣었는데 우연히 진짜 프라미스더라도 오버헤드는 전혀 없다.

### 3.7.3 `then(..)`과 `catch(..)`

(프라미스 API 명칭공간<sup>namespace</sup>이 아닌) 각 프라미스 인스턴스엔 `then(..)`,

`catch( ... )` 메소드가 들어있고 프라미스에 이름/버림 처리기를 등록할 수 있다. 프라미스가 귀결되면 두 처리기 중 딱 하나만(둘 다는 아니다) 언제나 비동기적으로 호출된다.

`then( ... )`은 하나 또는 2개의 파라미터를 받는데, 첫 번째는 이름 콜백, 두 번째는 버림 콜백이다. 어느 한쪽을 누락하거나 함수가 아닌 값으로 지정하면 각각 기본 콜백으로 대체된다. 기본 이름 콜백은 그냥 메시지를 전달하기만 하고 기본 버림 콜백은 단순히 전달받은 에러 사유를 도로 던진다(전파한다).

`catch( ... )`는 버림 콜백 하나만 받고 이름 콜백은 기본 이름 콜백으로 대체한다. 쉽게 말하면 `then(null, ...)`이나 다름없다.

---

```
p.then( fulfilled );
```

```
p.then( fulfilled, rejected );
```

```
p.catch( rejected ); // 또는 'p.then( null, rejected )'
```

---

`then( ... )`, `catch( ... )`도 새 프라미스를 만들어 반환하므로 프라미스 연쇄 형태로 흐름 제어를 표현할 수 있다. 이름/버림 콜백에서 예외가 발생하면 반환된 프라미스는 버린다. 콜백 반환값이 즉시값, 프라미스 아닌/데너블 아닌 값이면 반환된 프라미스의 이름값으로 지정한다. 이름 처리기가 특정 프라미스나 데너블 값을 반환하면 이 값을 풀어 반환된 프라미스의 귀결값이 된다.

### 3.7.4 Promise.all([ .. ])과 Promise.race([ .. ])

`Promise.all( [ .. ] )`, `Promise.race( [ .. ] )`는 프라미스를 생성하여 반환하는 ES6 프라미스 API의 정적 헬퍼 유틸이다. 프라미스 귀결은 전적으로 전달받은 프라미스 배열에 따라 좌우된다.

`Promise.all([ ... ])`은 주어진 모든 프라미스들이 이루어져야 메인 프라미스도 이루어지고, 단 하나라도 버려지게 되면 메인 반환 프라미스 역시 곧바로 폐기된다(다른 프라미스들의 결과는 그냥 무시한다). 이루어지면 각 프라미스의 이름 값이 담긴 배열을, 버려지면 처음 버려진 프라미스의 버림 사유를 돌려받는다. 전원 도착해야 문을 열 수 있다고 하여 예로부터 관문이라 불린다.

`Promise.race([ ... ])`는 오직 최초로 (이름이든 버림이든) 귀결된 프라미스만 승자가 되고 그 귀결값을 반환할 프라미스의 귀결값으로 삼는다. 옛날부터 결쇠라 불렀던 패턴이다. 결승선을 처음 통과한 일인자만 결쇠를 져히고 문을 열 수 있다. 다음 코드를 보자.

---

```
var p1 = Promise.resolve( 42 );
var p2 = Promise.resolve( "Hello, World" );
var p3 = Promise.reject( "허격" );

Promise.race( [p1,p2,p3] )
  .then( function(msg){
    console.log( msg ); // 42
  } );

Promise.all( [p1,p2,p3] )
  .catch( function(err){
    console.error( err ); // "허격"
  } );

Promise.all( [p1,p2] )
  .then( function(msgs){
    console.log( msgs ); // [42,"Hello, World"]
  } );
```



빈 배열을 `Promise.all([ ... ])`에 주면 바로 이루어지지만, `Promise.race([ ... ])`는 귀결되지 않은 채 프로그램이 멈춘다.

ES6 프라미스 API는 아주 간단하고 직관적이다. 기본적인 비동기 케이스라면 대

부분 바로 이용할 수 있고 콜백 지옥에 빠진 코드를 재편성하여 개선하기 위한 출발점으로서 매우 유용하다.

하지만 애플리케이션을 개발하다 보면 종종 프라미스만으로는 해결하기 어려운, 복잡 미묘한 다양한 문제들이 도사리고 있다. 다음 절에서는 이러한 프라미스의 한계를 극복하기 위해 프라미스 라이브러리를 어떻게 활용할지 이야기한다.

## 3.8 프라미스 한계

이 절에서 얘기할 프라미스의 한계에 대해서는 이미 앞부분에서 언급한 바 있으나 구체적으로 다시 한번 분명히 정리해보겠다.

### 3.8.1 시퀀스 에러 처리

프라미스식 에러 처리 기법은 이 장 도입부에서 살펴보았다. 프라미스의 설계 상 한계 (특히 연쇄 방법) 탓에 프라미스 연쇄에서 에러가 나면 그냥 조용히 묻혀버리기 쉽다.

이외에도 프라미스 에러는 고려해야 할 점이 있다. 프라미스 연쇄는 구성원들을 한데 모아놓은 사슬에 불과하기 때문에 전체 연쇄를 하나의 ‘뭔가’로 가리킬 개체 `entity`가 마땅치 않다. 즉, 일어날지 모를 에러를 밖에서는 감지할 도리가 없다.

에러 처리기가 없는 프라미스 연쇄에서 에러가 나면 나중에 어딘가에서 (어느 단계에서 등록된 버림 처리기에 의해) 감지될 때까지 연쇄를 따라 죽 하위로 전파한다. 그래서 이런 경우라면 연쇄의 마지막 프라미스를 가리키는 레퍼런스(다음 예제 p)만 갖고 있으면 여기에 버림 처리기를 등록하여 전파되어 내려온 에러를 처리할 수 있다.

---

```
// foo(..), STEP2(..), STEP3(..)은  
// 모두 프라미스-인식형 유틸이다.
```

```
var p = foo( 42 )  
.then( STEP2 )  
.then( STEP3 );
```

---

살짝궁 헛갈리지만 p가 가리키는 대상은 이 연쇄의 (foo(42) 호출로 시작된) 첫 번째 프라미스가 아니라 then( STEP3 ) 호출 후 반환된 마지막 프라미스다.

또 프라미스 연쇄는 각 단계에서 자신의 에러를 감지하여 처리할 방법 자체가 없으니 p에 에러 처리기를 달아놓으면 연쇄 어디에서 에러가 나도 이를 받아 처리할 수 있다.

---

```
p.catch( handleErrors );
```

---

하지만 연쇄의 어느 단계에서 나름대로(숨겨진/추상화한 방법으로) 에러 처리를 하면 handleErrors( .. )는 에러를 감지할 방법이 없다. 이게 당초 의도한 바일 수도 있지만(말하자면 “이미 처리된 버림(handled rejection)” 그렇지 않을 가능성도 있다. (“이미 처리된 버림” 에러를) 전혀 통지받을 수 없기 때문에 어떤 유스 케이스(use case)에서는 기능적 제약 사항이 된다.

이것은 try..catch에도 기본적으로 존재하는 한계로, 예외가 잡혀도 그냥 묻혀 버릴 가능성은 얼마든지 있다. 따라서 프라미스만의 한계는 아니지만 어떻게든 해결책을 찾아야 할 문제다.

그러나 아쉽게도 일반적으로 프라미스 연쇄 시퀀스에서 중간 단계를 참조할 레퍼런스가 없기 때문에 정확하게 에러를 슈아낼 에러 처리기를 붙일 수 없다.

### 3.8.2 단일값

프라미스는 정의 상 하나의 이름값, 아니면 하나의 버림 사유를 가진다. 간단한 코드라면 별다른 문제가 없으나 로직이 복잡해지면 발목을 붙들 수 있는 특성이다.

흔히들 메시지를 여러 개 담아둘 값 감싸미(객체나 배열)를 만들면 된다고 하는데, 물론 잘 작동은 하겠지만 프라미스 연쇄의 단계마다 메시지를 감싸고 푸는 일은 무척 불편하고 지루할 것이다.

### 값을 분할

가끔은 이럴 때 2개 이상의 프라미스로 분해해보면 해결의 실마리를 찾을 수 있다.

두 값( $x$ 와  $y$ )을 비동기적으로 만들어내는 유틸 `foo (..)`를 생각해보자.

---

```
function getY(x) {
  return new Promise( function(resolve,reject){
    setTimeout( function(){
      resolve( (3 * x) - 1 );
    }, 100 );
  } );
}
```

```
function foo(bar,baz) {
  var x = bar * baz;

  return getY( x )
  .then( function(y){
    // 두 값을 컨테이너에 넣는다.
    return [x,y];
  } );
}
```

```
foo( 10, 20 )
.then( function(msgs){
  var x = msgs[0];
  var y = msgs[1];

  console.log( x, y ); // 200 599
} );
```

---

먼저,  $x$ ,  $y$ 를 하나의 배열값으로 감싸서 프라미스 하나로 전달할 필요가 없도록

foo(..)가 반환하는 것을 다시 조정하자. 각 값은 자신의 프라미스로 감쌀 수 있다.

---

```
function foo(bar,baz) {
  var x = bar * baz;

  // 두 프라미스를 반환한다.
  return [
    Promise.resolve( x ),
    getY( x )
  ];
}

Promise.all(
  foo( 10, 20 )
)
.then( function(msgs){
  var x = msgs[0];
  var y = msgs[1];

  console.log( x, y );
} );
```

---

한 개의 프라미스를 통해 값 배열보다 프라미스 배열을 넘기는 형태가 정말 더 좋을까? 구문 상으로 그다지 큰 개선이라 할 수 없다.

그러나 후자가 프라미스 설계 사상을 더 잘 반영했다. 이렇게 해야 차후 x, y의 계산을 별도의 함수로 분리해서 리팩토링하기가 더 쉽고, 호출부로 하여금 두 프라미스를 알아서 조정하도록 (예제는 Promise.all([ .. ])을 사용했지만 반드시 그래야 하는 건 아니다) 놔두는 편이 foo(..) 안에 세부 로직을 추상화하는 것보다 더 깔끔하고 유연하다.

## 인자를 풀기/퍼뜨리기

var x = ..와 var y = .. 할당은 불필요한 오버헤드다. 헬퍼 유틸에 기능적인 꿈을 부려보자(출처: 레지날드 브레이스웨이트<sup>Reginald Braithwaite</sup>, 트위터 [@raganwald](#)).

---

```
function spread(fn) {
    return Function.apply.bind( fn, null );
}

Promise.all(
    foo( 10, 20 )
)
.then(
    spread( function(x,y){
        console.log( x, y ); // 200 599
    } )
)

```

---

확실히 나아졌다! 물론 헬퍼를 또 추가하지 않으려면 로직을 안쪽에 넣으면 된다.

---

```
Promise.all(
    foo( 10, 20 )
)
.then( Function.apply.bind(
    function(x,y){
        console.log( x, y ); // 200 599
    },
    null
) );

```

---

이 정도로도 훌륭하지만 ES6부터 **해체 destructing<sup>06</sup>**라는 더 나은 방법을 제공한다.  
다음은 배열 형태로 해체 할당하는 예다.

---

**06** 역자주\_예제를 보면 이해가 더 쉬울 것입니다. 다음 두 할당문은 정확히 같습니다.

```
var [m, d, y] = [3, 14, 1977];
var m = 3, d = 14, y = 1977;
```

함수에서 여러 값을 한꺼번에 반환받을 때 이러한 해체 할당 형식을 사용하면 편리할 때가 많습니다.

```
function now() { return [2, 6, 2013, 8, 0]; }
var [m, d] = now(); // m = 2, d = 6
var [,,year] = now(); // year = 2013
```



---

```
Promise.all(  
  foo( 10, 20 )  
)  
.then( function(msgs){  
  var [x,y] = msgs;  
  
  console.log( x, y ); // 200 599  
} );
```

---

ES6는 파라미터도 배열 해체 형식으로 쓸 수도 있는데 이게 더 낫다.

---

```
Promise.all(  
  foo( 10, 20 )  
)  
.then( function([x,y]){  
  console.log( x, y ); // 200 599  
} );
```

---

이제 ‘프라미스 당 값을 하나만 주시옵소서’ 하는 주기도문을 완성했다! 관용 코드를 최소로 쓰면서 말이다!



ES6 해제 형식에 관한 자세한 내용은 본 시리즈 ‘ES6과 그 이후’를 참고하자.

### 3.8.3 단일 귀결

프라미스가 단 1회만 귀결된다는 점은 프라미스의 가장 중요한 본질이다. 비동기 프로그래밍에서는 대부분 값을 한번만 가져오기 때문에 괜찮다.

그러나 데이터 이벤트/스트림에 더 가까운, 다른 모델과 어울리는 비동기 케이스도 많다. 이러한 유스 케이스에서도 프라미스가 과연 잘 작동할지는 일단 확실하지 않다. 프라미스에 뭔가 중요한 추상화를 덧대지 않고선 다수의 귀결값을 처리

하는 상황에 바로 적용하기는 어려울 것이다.

버튼 클릭 등 실제로 여러 번 발생하는 (이벤트 같은) 자극에 대응하여 일련의 비동기 단계를 진행해야 하는 시나리오가 있다고 하자.

심중팔구 원하는 대로 작동하지 않을 것이다.

---

```
// 'click(..)'은 '"click"' 이벤트를 DOM 요소에 바인딩한다.  
// 'request(..)'는 앞에서 정의한 프라미스-인식형 AJAX 요청이다.
```

```
var p = new Promise( function(resolve,reject){  
    click( "#mybtn", resolve );  
} );  
  
p.then( function(evt){  
    var btnID = evt.currentTarget.id;  
    return request( "http://some.url.1/?id=" + btnID );  
} )  
    .then( function(text){  
        console.log( text );  
    } );
```

---

여기서 정의한 로직은 버튼을 딱 한번만 눌러야 한다는 전제 하에 실행된다. 버튼을 한번 더 누르면 프라미스 `p`는 이미 귀결된 상태이므로 두 번째 `resolve(..)`는 조용히 묻힌다.

따라서 각 이벤트가 발사되면 새 프라미스 연쇄 전체를 생성하는 식으로 기존 체계를 뒤엎을 필요가 있다.

---

```
click( "#mybtn", function(evt){  
    var btnID = evt.currentTarget.id;  
  
    request( "http://some.url.1/?id=" + btnID )  
        .then( function(text){  
            console.log( text );  
        } );  
} );
```

---

이제 버튼 “클릭” 이벤트가 발생할 때마다 전혀 새로운 프라미스 시퀀스가 등장할 것이다.

그러나 이벤트 처리기 안쪽에 전체 프라미스 연쇄를 정의하는 코드가 삽입되어 지저분해지는 건 그렇다 하더라도 설계 관점에서 관심사/능력의 분리<sup>SoC, separation of concerns/capabilities</sup>라는 기본 원리에 위배된다. 여러분도 개발자라면 당연히 이벤트에 ‘응답’하는 코드(프라미스 연쇄)와 이벤트 처리기를 각각 다른 위치에서 정의하려고 할 것이다. 헬퍼 체계 없이는 적잖이 어색하고 불편한 패턴이다.



이러한 한계를 달리 표현하자면, 프라미스 연쇄를 구축할 수 있는 일종의 “옵저버블(observable)”을 구축할 수 있으면 참 좋을 것 같다. (RxJS 처럼) 이미 그러한 추상화를 제공하는 라이브러리가 나와 있지만 구현체가 너무 무거워서 프라미스 본연의 모습은 온데간데없다. 이렇게 덩치만 큰 추상화 라이브러리를 쓸 때는 (프라미스 없이도) 프라미스 설계 사상이 처음부터 의도했던 충분한 믿음성이 보장되는지를 사전에 잘 검토해야 한다. 부록 B “옵저버블” 패턴에서 다시 이야기한다.

### 3.8.4 타성

여러분이 오늘부터 당장 프라미스를 활용하기 시작하려고 할 때 가장 실질적인 걸림돌은 프라미스-인식형이 아닌 기존 코드일 것이다. 콜백이 이미 뿌리깊이 자리 잡은 코드라면 계속 같은 스타일을 유지하는 편이 훨씬 쉬울 테니까.

“지금 운영 중인 콜백식 코드 베이스는 똑똑한 프라미스-인식형 개발자가 작성하지 않으면 계속 현 상태 그대로 (콜백식으로) 남아있게 될 것이다.”

프라미스 체계는 차원이 달라서 전체적인 코드 접근 방식 자체가 다를 수 밖에 없고 아예 근원부터 다른 경우도 더러 있다. 여태껏 개발자들이 버텨 온 옛날 코딩 방식을 조금 흔드는 정도가 아닌 까닭에 변화를 원한다면 굳은 결심이 필요하다.

다음 콜백식 코드를 보자.

---

```
function foo(x,y,cb) {
  ajax(
    "http://some.url.1/?x=" + x + "&y=" + y,
    cb
  );
}

foo( 11, 31, function(err,text) {
  if (err) {
    console.error( err );
  }
  else {
    console.log( text );
  }
} );
```

---

이렇게 생긴 콜백식 코드를 프라미스-인식형 코드로 전환하기 위한 첫 단추는 무엇일까? 개발자 각각의 경험에 따라 다르다. 훈련이 잘 된 개발자일수록 더 당연하게 받아들일 것이다. 하지만 확실히 프라미스는 어떻게 사용해야 할지 겉모습만으로는 도통 알 수가 없다. 모든 경우에 꼭 맞는 정답은 없으니 결국 모든 책임은 개발자 여러분의 몫이다.

앞서 설명했듯이 AJAX도 콜백식이 아닌, 프라미스-인식형 유틸이 있어서 `request(..)`를 부를 수 있어야 한다. 여러분 스스로 유틸을 제작할 수도 있겠지만 모든 콜백식 유틸을 감싸는 프라미스-인식형 감싸미를 일일이 정의해야 하는 부담 때문에 프라미스-인식형 코딩으로 리팩토링할 사기를 떨어뜨리기도 한다.

이런 한계점까지 프라미스가 정답을 주진 않는다. 대다수의 프라미스 라이브러리가 헬퍼를 제공하지만 라이브러리를 안 써도 다음과 같이 직접 헬퍼를 작성하면 된다.

---

```
// 폴리필 안전 체크
if (!Promise.wrap) {
```

```

Promise.wrap = function(fn) {
  return function() {
    var args = [].slice.call( arguments );

    return new Promise( function(resolve,reject){
      fn.apply(
        null,
        args.concat( function(err,v){
          if (err) {
            reject( err );
          }
          else {
            resolve( v );
          }
        } )
      );
    } );
  };
};
}

```

---

음, 그냥 아무렇게나 짤 수 있는 코드는 아닌 것 같다. 하지만 약간 위압감을 느끼게 할 코드일지는 몰라도 생각만큼 그렇게 복잡하진 않다. 예러 우선 스타일의 콜백을 마지막 파라미터로 취하는 함수를 받아 프라미스를 생성, 반환하는 새 함수를 반환하고 콜백을 교체하여 프라미스 이름/버림과 연결시킨다.

`Promise.wrap( ... )` 헬퍼가 어떻게 작동하는지 바로 용례를 보자.

---

```
var request = Promise.wrap( ajax );
```

```
request( "http://some.url.1/" )
  .then( ... )
  ..
```

---

와, 정말 쉽다!

`Promise.wrap(..)`은 프라미스를 직접 만들지 않고 프라미스를 만드는 함수를 만든다. 얼핏 보면 프라미스를 만드는 프라미스 공장처럼 느껴진다. 이런 것들(“프라미스” + “공장”)을 “프라미서리<sup>promisery</sup>”라고 부르면 어떨까 싶다.

콜백식 함수를 프라미스-인식형 함수로 감싸는 것을 “리프팅<sup>lifting</sup>” 내지는 “프라미시파이<sup>promisifying</sup>”이라고 한다. 그러나 “리프트드 함수<sup>lifted function</sup>”를 제외하고 이렇게 만들어진 함수를 지칭할 표준 용어가 마땅치 않아 나는 “프라미서리”라고 부른다. 용어의 느낌이 더 잘 와닿지 않는가?



프라미서리는 그냥 막 지어낸 용어가 아니다. 프라미스를 포함/전달한다는 뜻을 가진, 실제 있는 단어다. 함수가 하는 일과도 정확히 잘 맞는 단어라서 아주 완벽하게 매칭되는 기술 용어다!

따라서, `Promise.wrap ajax`는 `request(..)`로 호출할 `ajax(..)` 프라미서리를 생성하고 이 프라미서리는 AJAX 응답에 대한 프라미스를 만든다.

모든 함수가 진작에 프라미서리였다면 굳이 애써 만들 필요가 없으니 한 단계 더 거치는 것은 약간 낭비다. 하지만 적어도 감싸는 패턴은 (대개) 반복 가능<sup>repeatable</sup>하기 때문에 프라미스 코딩을 보조할 목적으로 예제처럼 `Promise.wrap(..)` 헬퍼에 넣어 쓰면 된다.

앞 예제로 돌아가면 `ajax(..)`, `foo(..)` 모두 프라미서리가 필요하다.

---

```
// 'ajax(..)'에 대해 프라미서리를 만든다.
```

```
var request = Promise.wrap( ajax );
```

```
// 'foo(..)'를 리팩토링한다. 하지만 당장은 다른 부분의 코드와의
```

```
// 호환성을 위해 외부적으로 콜백 기반 체제를 유지한다.
```

```
// 'request(..)'의 프라미스는 오직 내부적으로만 사용한다.
```

```
function foo(x,y,cb) {
  request(
    "http://some.url.1/?x=" + x + "&y=" + y
  )
  .then(
```

```

        function fulfilled(text){
            cb( null, text );
        },
        cb
    );
}

```

```

// 이제 이 예제의 목적인
// 'foo(..)'에 대한 프라미서리를 만든다.
var betterFoo = Promise.wrap( foo );

```

```

// 그리고 프라미서리를 사용한다.
betterFoo( 11, 31 )
.then(
    function fulfilled(text){
        console.log( text );
    },
    function rejected(err){
        console.error( err );
    }
);

```

---

물론, 새 `request( .. )` 프라미서리를 사용하기 위해 `foo( .. )`를 리팩토링했지만 콜백식 코드를 유지하고 이어지는 `betterFoo( .. )` 프라미서리를 쓰지 않고 그냥 `foo( .. )` 자체를 프라미서리로 만드는 방법도 있다. 어떻게 할지는 기존의 코드 베이스와 더불어 `foo( .. )`를 콜백식 코딩과 호환되도록 남겨둬야 할지 결정하면 된다.

다음 코드를 보자.

---

```

// 'request(..)' 프라미서리로 위임되므로
// 'foo(..)'도 이제 프라미서리다.
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

```

```
foo( 11, 31 )
.then( .. )
..
```

---

ES6 프라미스에는 프라미서리를 감싸는 헬퍼 유틸은 기본적으로 제공되지 않지만 많은 라이브러리에 이미 구현되어 있고 필요하다면 직접 만들어 써도 된다. 어쨌든, 크게 공들이지 않고도 해결 가능한 한계점이다(콜백 지옥의 고통에 비한다면야).

### 3.8.5 프라미스는 취소 불가

일단 프라미스를 생성하여 이름/버림 처리기를 등록하면, 도중에 작업 자체를 의미없게 만드는 일이 발생하더라도 외부에서 프라미스 진행을 멈출 방법이 없다.



수많은 프라미스 추상화 라이브러리에서 프라미스 취소 기능을 제공하고 있지만 정말 나쁜 생각이다! 개발자 입장에서선 처음부터 외부에서 프라미스를 취소할 수 있게 만들어졌더라면 좋지 않을까 싶겠지만, 특정한 프라미스의 사용자(consumer)/감지자(observer)가 동일한 프라미스를 바라보는 다른 사용자에게까지 영향을 끼칠 수 있다는 점이 문제다. 이는 미래값의 믿음성(외부적 불변성) 원리에 위배될뿐만 아니라 “원격 작용(action at a distance”<sup>10</sup>이라는 안티패턴(antipattern)을 구현한 것이기도 하다. 프라미스 취소 기능이 꽤나 쓸모있을 것 같겠지만 결국 콜백 지옥과 똑같은 악몽이 재현될 것이다.

다음은 전에 봤었던 프라미스 타임아웃 예제다.

---

```
var p = foo( 42 );

Promise.race( [
  p,
  timeoutPromise( 3000 )
] )
.then(
  doSomething,
```

---

**07** 역자주\_동일한 프로그램 내에서 한 영역이 다른 영역의, 거의 식별하기 불가능한(매우 어려운) 처리 로직 때문에 매우 가변적으로 작동(따라서 예측하기 어려운 방향으로 작동)하는 문제를 말합니다. 이를 방지하려면 전역 변수 사용을 가급적 피하고 데이터를 변경하는 로직은 지역 스코프에 한정하여 구현하는 것이 좋습니다.



```

    handleError
  );

  p.then( function(){
    // 타임아웃이 되어도 여전히 실행된다 ---;;
  } );

```

---

프라미스 p 입장에선 "타임아웃"이 외부 요소이므로 p는 계속 실행되는데, 이는 개발자가 의도한 작동 방식이 아닐 것이다.

귀결 콜백을 직접 작성해보자.

---

```

var OK = true;

var p = foo( 42 );

Promise.race( [
  p,
  timeoutPromise( 3000 )
  .catch( function(err){
    OK = false;
    throw err;
  } )
] )
  .then(
    doSomething,
    handleError
  );

p.then( function(){
  if (OK) {
    // 타임아웃이 없을 때에만 실행된다! ^^
  }
} );

```

---

음, 실행되긴 하지만 최선의 코드와는 거리가 멀다. 보통 이런 코딩은 피하는 게 좋다.

하지만 피할 수 없는 상황에서 이렇게 자구책을 강구하다보면 코드가 지저분해질 수 밖에 없다는 사실로부터 ‘프라미스 취소’는 더 상위의 프라미스 추상화 수준에서 구현해야 할 기능임을 유추할 수 있다. 어떻게든 꼬수를 써서 해결해보려고 하지 말고 프라미스 추상화 라이브러리를 찾아 쓰는 편이 좋겠다.



내가 작성한 `asynquence`(부록 A 참고)에도 이러한 추상화를 제공하며 `시퀀스`에 `abort()` 함수를 사용하면 된다.

사실 프라미스 한 개는 ‘취소’라는 말이 지칭하는 (적어도 그런 뉘앙스를 가진) 흐름 제어 체계라고 볼 수 없다. 프라미스 취소란 말 자체가 어색하게 느껴지는 이유가 바로 이 때문이다.

반면, 프라미스가 모여있는 (내가 “시퀀스”라 부르는) 프라미스 연쇄는 흐름 제어를 나타낸 것이 분명하므로 이 추상화 수준에서 취소 기능을 정의하는 건 타당하다고 본다.

프라미스 하나씩은 취소 불가지만 ‘시퀀스’ 형태라면 프라미스처럼 단일 불변값을 통째로 전달할 일은 없을 테니 취소가 되어야 사리에 맞다.

### 3.8.6 프라미스 성능

프라미스의 독특한 한계는 단순하면서도 복잡하다.

콜백식 비동기 작업 연쇄와 프라미스 연쇄의 움직이는 코드 조각이 얼마나 되는지만 보면 아무래도 프라미스가 처리량이 더 많고 그래서 속도 역시 약간 더 느린 게 사실이다. 하지만 동일한 수준의 믿음성을 보장하기 위해 프라미스에서 간단히 몇 가지 장치만으로 해결했던 것과 콜백에서 임기 응변식 코드를 덕지덕지 발라야 했던 것을 상기하기 바란다.

할 일도 많고 보호해야 할 것도 더 많으니 믿음이 안 가는, 발가벗겨진 콜백보다

프라미스가 상대적으로 느린 건 당연하다. 지극히 단순 명료한 사실이라 이해하기 어렵진 않다.

그렇다면 얼마나 더 느리다는 말인가? 음... 이 질문에 절대적으로 옳은 답변을 하기는 정말 어렵다.

솔직히 이걸 ‘사과냐 오렌지냐<sup>08</sup>’ 같은 잘못된 질문이 아닐까 싶다. 굳이 비교를 하자면 동일한 수준의 보호 장치를 수작업으로 구축한 임시 콜백 시스템을 그 대상으로 삼는 것이 온당하지 않을까?

프라미스에 어떤 성능 상 한계가 분명히 있다면 필요한 믿음성 보장 수단(언제나 모두 갖춰져 있다)을 개별적으로 선택할 수 있게 프라미스가 제공하지 않아서 그런 것이 아니다.

그럼에도 불구하고, 일반적으로 프라미스가 프라미스 아닌, 미덥잖은 콜백 같은 것들보다 (믿음성이 결여돼도 괜찮다고 느껴지는 코드에 쓴다고 할 때) ‘약간이라도 더 느리다’는 사실을 인정한다는 건, 전체 애플리케이션이 될수록 무조건 최고로 빠른 코드에 의해 실행되어야 한다는 전제 하에 프로그램에서 전반적으로 프라미스를 쓰지 말아야 한다는 소리일까?

정상 테스트<sup>sanity check</sup><sup>09</sup>를 해보자. 만약 마땅히 그렇게 짜야 하는 코드였다면 처음부터 구현 언어를 자바스크립트로 택한 행위 자체가 적절했다고 볼 수 있을까? 물론 자바스크립트는 애플리케이션 성능 요건을 충족하기 위해 최적화할 수 있다. 그러나 프라미스의 장점과 혜택을 비추어 보았을 때 사소한 프라미스의 성능 문제에 집착하는 모습이 합리적일까?

프라미스가 ‘모든 것’을 비동기화한다는 논란거리도 있다. 어느 정도 즉시(동기적으

---

<sup>08</sup> 역자주\_본질적으로 비교할 수 없는 대상을 애써 비교하여 어떤 차이점을 찾으려고 할 때 구미권에서는 ‘사과냐 오렌지냐(apples and oranges)’라는 말을 자주 씁니다. 우리 나라 정서에 맞게 바꾸면 ‘짜장면이나 짬뽕이나’ 정도가 되겠군요.

<sup>09</sup> 역자주\_명백한 실수가 없었는지 확인하기 위한 점검을 말합니다.

로 완료된 단계들이 여전히 다음 단계의 잡(1장 참고) 진행을 지연시킨다는 것이다. 즉, 콜백식 시퀀스에 비해 일련의 프라미스 작업이 아주 미미하나마 더 늦게 끝날 가능성이 있다는 소리다.

자, 여기서 한번 생각해보자. 이렇게 사소하지만 성능 상 부정적인 영향을 끼칠 수 있는 단점 한 가지가 이 장에서 또렷하게 밝혀온, 프라미스의 다른 장점을 전부를 합친 것만큼의 가치가 있을까?

나는 프라미스의 성능이 우려가 될 만한 거의 모든 상황에서 사실상 프라미스를 모조리 견어내고 믿음성/조합성이라는 프라미스의 결정적인 혜택을 없애버리는 식의 최적화야말로 진정한 안티패턴<sup>anti-pattern</sup><sup>10</sup>이라고 생각한다.

대신에 코드 베이스 전반에 걸쳐 프라미스 사용을 기본으로 하고 애플리케이션 임계 경로<sup>critical path</sup><sup>11</sup>에 대한 자료를 수집/분석해보자. 프라미스가 정말 병목<sup>bottleneck</sup>인가, 아니면 단지 이론적인 속도 저하인가? 신중하고 책임감 있는 개발자라면 실제로 유의미한 벤치마킹을 수행한 다음 핵심 영역으로 밝혀진 부분에서 프라미스를 숨아낼 것이다.

프라미스가 비록 약간 느린 건 사실이지만, 그 대신 믿음성, 예측성, 조합성과 같은 장점을 고루 누릴 수 있다. 어쩌면 진짜 한계는 프라미스의 성능이 아니라 프라미스의 진면목을 알아보지 못하는 ‘안목 없음’일 것 같다.

<sup>10</sup> 역자주\_안티패턴(anti-pattern)은 소프트웨어 공학 분야 용어이며, 실제 많이 사용되는 패턴이지만 비효율적이거나 비생산적인 패턴을 의미합니다. 안티패턴은 1995년 앤드루 케이니그가 디자인 패턴을 참고하여 처음 사용한 말입니다. (출처: 위키백과)

<sup>11</sup> 역자주\_여러 단계의 과정을 거치는 작업에서 그것을 완성하려면 여러 과정의 경로가 동시에 수행되어야 한다고 할 때, 그중 가장 긴 경로. 즉, 전체 공정 중 시간이 가장 많이 걸리는 경로이다. (출처: IT용어사전, 한국정보통신기술협회)

### 3.9 정리하기

프라미스는 훌륭하다. 맘껏 쓰라. 콜백식 코드에서 줄곧 목의 가시였던 '제어의 역전' 문제를 프라미스가 한방에 해결했다.

프라미스가 콜백을 완전히 없애는 건 아니다. 다만 기존 콜백 코드를 믿을 만한 중계자 역할을 수행하는 유틸을 통해 잘 조정하여 서로 조화롭게 작동할 수 있도록 유도한 것이다.

프라미스 연쇄는 비동기 흐름을 순차적으로 표현하는 더 나은 (완벽하진 않지만) 방법이다. 덕분에 우리 두뇌가 비동기 자바스크립트 코드를 좀 더 효율적으로 계획/관리할 수 있다. 4장에서는 한층 더 개선된 솔루션을 소개할 예정이다!

## 제너레이터

2장에서는 콜백으로 비동기 흐름 제어를 나타내는 방법의 두 가지 결정적 흠을 이야기한 바 있다.

- 콜백식 비동기는 우리가 머릿속으로 계획하는 작업의 단계와 잘 맞지 않는다.
- 콜백은 제어의 역전 문제로 인해 믿음성이 떨어지고 조합하기 어렵다.

3장에서는 프라미스로 믿음성/조합성을 살리면서 제어의 역전을 되역전하는 방법을 살펴보았다.

이 장에선 어떻게 하면 비동기 흐름 제어를 순차적/동기적 모습으로 나타낼 수 있을지 고민해볼 것이다. “흑마술”의 주인공은 바로 ES6 무대에서 데뷔한 ‘제너레이터<sup>generator</sup>’다.

## 4.1 완전-실행을 타파하다

자바스크립트 개발자들이 대부분 당연히 그러리라고 민졌지만, 일단 함수가 실행되기 시작하면 완료될 때까지 계속 실행되며 도중에 다른 코드가 끼어들어 실행되는 법은 없다고 1장에서 말했었다.

그런데 생뚱맞게도 ES6부터 완전-실행 법칙을 따르지 않는, 제너레이터라는 전혀 새로운 종류의 함수가 등장했다.

다음 예제를 보면서 이야기하자.

---

```

var x = 1;

function foo() {
    x++;
    bar(); // <-- 이 줄의 정체는?
    console.log( "x:", x );
}

function bar() {
    x++;
}

foo(); // x: 3

```

---

틀림없이 `bar()`는 `x++`와 `console.log(x)` 사이에서 실행될 것이다. `bar()`가 없으면 당연히 결과도 3이 아닌, 2가 될 것이다.

그런데 이렇게 한번 생각해보자. `bar()`가 없는데도 `x++`, `console.log(x)` 사이에서 `bar()`가 실행될 수 있다면? 이런 일이 일어날 수 있을까?

선점형<sup>preemptive</sup> 멀티스레드<sup>01</sup> 언어라면 일반적으로 두 문 사이의 특정 시점에 정확히 `bar()`가 끼어들어 실행되게 할 수 있지만 자바스크립트는 선점형 언어도, (적어도 지금은) 멀티스레드 언어도 아니다. 하지만 `foo()` 자체가 어떤 코드 부분에서 멈춤 신호를 줄 수만 있다면 이러한 끼어들기<sup>interruption</sup>를 ‘협동적<sup>cooperative</sup>’(동시성) 형태로 나타낼 수 있다.

---

<sup>01</sup> 역자주\_선점(preemption)이란 개념은 컴퓨터 시스템에서 어떤 작업을 실행하다가 나중에 다시 실행할 의도를 갖고 강제로 멈추게 하는 것, 즉 끼어들기(interrupt)를 시키는 행위를 말합니다. 따라서 선점형 멀티스레드 환경에서는 선점 특권을 가진 스케줄러에 의해 스레드 A가 실행되는 도중에도 우선 순위가 더 높은 스레드 B가 준비되면 곧바로 대체될 수 있고 스레드 A는 나중에 다시 실행되는 식으로 작동합니다. 참고로, 이렇게 선점에 의한 실행 흐름의 변경을 컨텍스트 교환(context change)이라고 합니다.



여기서 “협동적”이란 단어를 쓴 이유는, 고전적 동시성 용어(1장 참고)와 연관지어려는 의도도 있지만 코드의 멈춤 위치를 나타내는 ES6 구문이 `yield`(양도)이기 때문이다(바로 다음 예제에 나온다). 즉, 제어권을 친절하게 협력적으로 양도한다는 뜻이다.<sup>17</sup>

협동적 동시성<sup>cooperative concurrency</sup>을 달성한 ES6 코드를 보자.

---

```
var x = 1;

function *foo() {
  x++;
  yield; // 멈추시오!
  console.log( "x:", x );
}

function bar() {
  x++;
}
```

---



다른 자바스크립트 문서/코드에는 제너레이터 선언 코드가 `function *foo()` { .. } 대신 `function* foo()` { .. } 형식으로 기술된 경우가 많다. (\* 위치가 다르다) 기능/구문적으로는 완전히 같고 공백 없이 `function*foo()` { .. }로 써도 마찬가지다. 어디까지나 코딩 스타일 문제지만 `*foo()`로 제너레이터를 참조하는 모양새가 더 좋은 것 같아 개인적으로 난 `function *foo..`를 즐겨쓴다. `foo()`라고만 쓰면 가리키는 대상이 제너레이터인지, 일반 함수인지 다른 사람들은 알 수가 없기 때문이다. 어쨌거나 순전히 개인적 취향에 관한 문제다.

자, 그럼 `*foo()`의 `yield` 지점에서 `bar()`는 어떻게 실행될까?

---

// 이터레이터 'it'를 선언하여 제너레이터를 제어한다.

02 역자주\_협동(cooperation)이란 말은 선점(preemption)의 반대, 즉 비선점(non-preemption)을 의미합니다. 쉽게 풀이하자면, 선점은 협동할 생각이 조금도 없는, 특권을 가진 독재자가 마음대로 실행 중인 작업을 다른 작업으로 갈아치우는 등의 스케줄링을 행하는 것이고, 협동은 작업을 사이에 협동 정신을 발휘하여 시스템 자원을 쓰지 않을 때 다른 작업들이 쓸 수 있도록 제어권을 넘긴다(yield)는 의사를 프로그램에 명시적으로 밝히는 거라고 볼 수 있습니다.



```
var it = foo();

// 'foo()'는 여기서 시작된다!
it.next();
x; // 2
bar();
x; // 3
it.next(); // x: 3
```

---

지금은 여러분 눈에 낯설고 어지러운 코드 일색이겠지만 앞으로 차근차근 살펴볼 예정이니 너무 걱정말길! 색다른 ES6 제너레이터의 체계/구문을 공부하기 전에 그냥 실행 흐름을 죽 따라가보자.

1. `it = foo()` 할당으로 `*foo()` 제너레이터가 실행되는 건 아니다. 제너레이터 실행을 제어할 이터레이터<sup>iterator</sup>만 마련한다. (이터레이터는 잠시 후에 ...)
2. 첫 번째 `it.next()`가 `*foo()` 제너레이터를 시작하고 `*foo()` 첫째 줄의 `x++`가 실행된다.
3. 첫 번째 `it.next()`가 완료되는 `yield`문에서 `*foo()`는 멈춘다. `*foo()`는 실제로 실행 중이지만 멈춰있는 상태다.
4. `x` 값을 출력해보니 2다.
5. `bar()`를 호출하면 `x++`에서 `x` 값은 하나 증가한다.
6. `x` 값을 다시 확인하면 지금은 3이다.
7. 마지막 `it.next()` 호출부에 의해 `*foo()` 제너레이터는 좀 전에 멈췄던 곳에서 재개되어 `console.log(..)` 실행 후 현재 `x` 값 3을 콘솔창에 표시한다.

분명히 `foo()`가 시작되었지만 '완전-실행'되지 않고 `yield` 문에서 잠깐 멈췄다. 나중에 `foo()`를 재개하여 완료하지만 꼭 이렇게 해야 할 필요는 없었다.

제너레이터는 1회 이상 시작/실행을 거듭할 수 있으면서도 반드시 끝까지 실행해야 할 필요는 없는 특별한 함수다. 이런 함수가 뭐가 그리 강력하다는 소린지 지금은 잘 이해가 되지 않겠지만, 이 장의 나머지 부분을 읽으면서 여러분은 ‘비동기 흐름을 제어하는 제너레이터’ 패턴을 완성하기 위한 가장 기초적인 건축 자재로서 제너레이터가 어떻게 활용되는지 목도하게 될 것이다.

### 4.1.1 입력과 출력

다시 한번 말하지만, 제너레이터는 전혀 새로운 처리 모델에 기반을 둔, 특별한 함수다. 그래도 함수는 함수인지라 기본적인 체계, 즉 인자(입력)를 받고 어떤 값을 반환하는(출력) 기능은 일반 함수와 같다.

---

```
function *foo(x,y) {  
    return x * y;  
}  
  
var it = foo( 6, 7 );  
  
var res = it.next();  
  
res.value; // 42
```

---

6, 7을 \*foo (..) 파라미터 x, y로 각각 전달하니 42를 반환한다.

그런데 여타 함수와 제너레이터의 호출 방법이 다른 걸 알 수 있다. foo(6,7)은 당연해 보이는데 제너레이터 \*foo (..)는 일반 함수와는 달라서 이것만으로는 실행되지 않는다.

그래서 제너레이터를 제어하는 ‘이터레이터’ 객체를 만들어 변수 it에 할당하고 it.next() 해야 \*foo (..) 제너레이터가 현재 위치에서 다음 yield, 또는 제너레이터 끝까지 실행할 수 있다.

`next(..)`의 결괏값은 `*foo(..)`가 반환한 값(어떤 값이라도 좋다)을 `value` 프로퍼티에 저장한 객체다. 즉, `yield`는 실행 도중에 제너레이터로부터 값을, 일종의 중간 반환값 형태로 돌려준다.

지금은 왜 이런 간접적인 ‘이터레이터’ 객체를 내세워 제너레이터를 조종하려 하는지 이상하게 느껴질 것이다. 약속하건대 조금만 인내심을 갖자.

## 반복 메시징

인자를 받아 결괏값을 내는 기능 이외에도 제너레이터에는 `yield`와 `next(..)`를 통해 입력/출력 메시지를 주고받는, 강력하고 감탄스러운 기능이 탑재되어 있다.

다음 코드를 보자.

---

```
function *foo(x) {
  var y = x * (yield);
  return y;
}

var it = foo( 6 );

// 'foo(..)'를 시작한다.
it.next();

var res = it.next( 7 );

res.value; // 42
```

---

먼저, 파라미터 `x` 자리에 `6`을 넘기고 `it.next()`를 호출하여 `*foo(..)`를 시작한다.

`*foo(..)`에서 `var y = x ..` 문이 처리될 즈음 `yield` 표현식에서 걸린다. 여기서 `*foo(..)`는 (할당문 중간에서!) 실행을 멈추고 `yield` 표현식에 해당하는 결

값을 달라고 호출부 코드에 요청한다. 그리고 `it.next(7)`을 호출하면 7 값이 `yield` 표현식의 결과값이 되도록 전달한다.

따라서 결과적으로 할당문은 `var y = 6 * 7`이 되고, `return y`하면 `it.next(7)`를 호출한 결과값 42를 반환한다.

여기서 아주 중요하지만 숙련된 자바스크립트 개발자들도 자칫 헷갈리기 쉬운 대목이 있다. 바라보는 관점에 따라서 `yield`문의 개수와 `next(...)` 호출 횟수가 제각각이다. 일반적으로는 `yield` 개수보다 `next(...)` 호출이 하나 더 많다. 예제도 `yield`는 1개, `next(...)` 호출은 2회다.

### 왜 안 맞을까?

첫 번째 `next(...)` 호출이 항상 제너레이터를 가동하여 첫 번째 `yield`까지 실행하기 때문이다. 이후 두 번째 `next(...)` 호출은 첫 번째 멈췄던 `yield` 표현식을 이루고, 세 번째 `next(...)`도 마찬가지로 두 번째 멈췄던 `yield`를 이루는 식으로 진행된다.

### 두 가지 질문

사실 짝이 맞는다, 안 맞는다는 인식은 어떤 코드가 가장 큰 영향을 준다고 생각하는지에 따라 달라진다.

제너레이터 코드만 따로 떼어보자.

---

```
var y = x * (yield);  
return y;
```

---

첫 번째 `yield`의 질문은 기본적으로, “어떤 값을 여기에 삽입해야 하니?” 하는 거다.

답변은 누가 할까? 음, 첫 번째 `next(...)`는 이미 제너레이터를 이 지점까지 실행시켜 왔으니 당연히 답변 불가다. 따라서 첫 번째 `yield`가 던진 질문은 두 번째

next(..)가 대답해야 한다.

두 번째가 첫 번째를? 여기부터 짹짹이 아닌가?

하지만 관점을 확 바꿔보자. 제너레이터가 아닌, 이터레이터 입장에서 말이다.

이터레이터 관점에서 보면 양방향 메시징이 가능하다는 걸 설명할 필요가 있다. 표현식 `yield ..`는 `next(..)` 호출에 대응하여 메시지를 보낼 수 있고 `next(..)`는 멈춘 `yield` 표현식으로 값을 전송할 수 있다. 이해를 돕기 위해 코드를 조금 변형해보겠다.

---

```
function *foo(x) {  
  var y = x * (yield "hello"); // <← 값을 yield한다(준다)!  
  return y;  
}  
  
var it = foo( 6 );  
  
var res = it.next(); // 첫 번째 'next()'에선 아무것도 전달하지 않는다.  
res.value; // "hello"  
  
res = it.next( 7 ); // 기다리고 있는 'yield'에게 '7'을 넘긴다.  
res.value; // 42
```

---

`yield ..`와 `next(..)` 커플은 제너레이터 실행 도중 양방향 메시징 시스템으로 기능한다.

자, 이터레이터 코드만 잘라보자.

---

```
var res = it.next(); // 첫 번째 'next()'에선 아무것도 전달하지 않는다.  
res.value; // "hello"  
  
res = it.next( 7 ); // 기다리고 있는 'yield'에게 '7'을 넘긴다.  
res.value; // 42
```

---



첫 번째 `next()` 호출은 일부러 아무 값도 넘기지 않았다. 멈춘 `yield`만 `next(...)`가 전해준 값을 받을 수 있으므로 첫 번째 `next()`가 넘긴 값을 받을 멈춘 `yield`가 제너레이터 시작부에는 없기 때문이다. 명세에도 나와있고 모든 호환 브라우저에서 첫 번째 `next()`의 인자값은 조용히 무시한다. 굳이 조용히 실패하는 코드를 만들어 혼동을 유발할 필요는 없으니 첫 번째 `next()`는 값 없이 호출하자. 언제나 제너레이터는 인자 없는 `next()`로 시작한다.

(인자 없는) 첫 번째 `next()` 호출의 의미는 기본적으로 “\*foo(...) 제너레이터는 ‘다음에’ 어떤 값을 줄 거니?” 하는 것이다. 대답은 누가? 바로 첫 번째 `yield` “hello” 표현식이다.

이제 알겠는가? 짝이 안 맞는 게 아니다.

대답할 주체가 누구인지에 따라 `yield`와 `next(...)`가 잘 맞기도 하고 안 맞기도 한 것처럼 보인다.

그런데 잠깐! 여전히 `yield` 개수보다 `next()`가 하나 더 많다. 마지막 `it.next(7)` 호출은 다시금 제너레이터가 ‘다음에’ 어떤 값을 내어줄지 묻는 셈인데, 더 이상 이 질문에 대답할 `yield` 문이 하나도 없다. 그럼 누가 대신할까?

바로 `return` 문이다!

제너레이터 끝에 `return` 문이 따로 없으면(제너레이터에서 `return`은 일반 함수처럼 필수가 아니다) `return` 문이 있다고 치고 암시적으로 처리한다. (즉, `return undefined`;)

이제 마지막 `it.next(7)`가 던진 질문에 대답할 수 있게 되었다.

이러한 질의 응답 체계(`yield`, `next(...)` 커플의 양방향 메시징)는 매우 강력하다. 물론 여러분들은 아직도 비동기 흐름 제어와 제너레이터가 당최 무슨 상관인지 감 잡기 어려울 것이다. 기다리시라!

### 4.1.2 다중 이터레이터

구문 사용법만 놓고 보면 이터레이터로 제너레이터를 제어하는 건, 선언된 제너레이터 함수 자체를 제어하는 것처럼 보인다. 그러나 이터레이터를 생성할 때마다 해당 이터레이터가 제어할 제너레이터의 인스턴스 역시 암시적으로 생성된다는 사실을 지나치기 쉽다.

동일한 제너레이터의 인스턴스를 동시에 여러 개 실행할 수 있고 인스턴스끼리 상호 작용도 가능하다.

---

```
function *foo() {
  var x = yield 2;
  z++;
  var y = yield (x * z);
  console.log( x, y, z );
}

var z = 1;

var it1 = foo();
var it2 = foo();

var val1 = it1.next().value; // 2 ← yield 2
var val2 = it2.next().value; // 2 ← yield 2

val1 = it1.next( val2 * 10 ).value; // 40 ← x:20, z:2
val2 = it2.next( val1 * 5 ).value; // 600 ← x:200, z:3

it1.next( val2 / 2 ); // y:300
      // 20 300 3
it2.next( val1 / 4 ); // y:10
      // 200 10 3
```

---



똑같은 제너레이터의 여러 인스턴스를 동시에 실행하는 가장 일반적인 상황은 독립적으로 연결된 자원으로부터 입력값 없이 제너레이터 스스로 값을 생산할 때다. 인스턴스 간 상호 작용이 주가 아니다. 값 생산은 다음 절에서 이야기한다.

처리 로직을 간단히 살펴보자.

1. `*foo()` 인스턴스 2개를 동시에 실행하고 두 `next()` 호출 모두 `yield` 2 지점에서 2 값을 각각 넘겨받는다.
2. `val2 * 10`는 `2 * 10`이고 이 값은 첫 번째 인스턴스 `it1`에 전달되어 `x` 값은 20이 된다. `z` 값은 1에서 2로 증가하고 `20 * 2`을 `yield`하므로 `val1`은 40이 된다.
3. `val1 * 5`은 `40 * 5`이고 이 값은 두 번째 인스턴스 `it2`에 전달되어 `x` 값은 200이 된다. `z` 값은 다시 2에서 3으로 증가하고 `200 * 3`을 `yield`하면 `val2`는 600이 된다.
4. `val2 / 2`는 `600 / 2`이고 `it1`으로 전달되어 `y` 값은 300이 되고 콘솔창엔 `x y z` 값이 20 300 3으로 각각 표시된다.
5. `val1 / 4`는 `40 / 4`이고 `it2`로 전달되어 `y` 값은 10이 되고 콘솔창엔 200 10 3이 표시된다.

머릿속으로 돌려봄직한 재미있는 예제다. 흐름이 확실히 파악되는가?

## 인터리빙

1장 “완전-실행” 예제를 다시 보자.

---

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

function foo() {
  a++;
```



```

    b = b * a;
    a = b + 3;
}

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

```

---

일반 자바스크립트 함수 `foo()`, `bar()` 둘 중 하나는 다른 함수보다 먼저 완전-실행될 것이다. 하지만 `foo()`의 개별 문을 `bar()`에 인터리빙하여 실행하는 건 불가능하다. 따라서 실행 결과는 두 가지만 가능하다.

반면에 제너레이터는 문 사이에서도 인터리빙할 수 있다.

---

```

var a = 1;
var b = 2;

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

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

```

---

`*foo()`, `*bar()`를 제어하는 이터레이터 호출 순서에 따라 이 프로그램의 실행 결과는 제각각일 수 있다. 다시 말해, 동일한 변수를 공유한 상태에서 두 제너레이

터의 이터레이터를 인터리빙함으로써 1장의 (일종의 조작된) 이론적 스레드 결합 조건이 발생하는 환경을 실제로 재현할 수 있다.

먼저, 이터레이터를 제어하는 `step(..)`이란 헬퍼를 만들자.

---

```
function step(gen) {
  var it = gen();
  var last;

  return function() {
    // 어떤 값이 'yield'되어 나오든지
    // 바로 다음에 그냥 반환한다.
    last = it.next( last ).value;
  };
}
```

---

`step(..)`은 먼저 제너레이터를 초기화하고 이터레이터 `it`을 생성한 다음, 호출 시 이터레이터를 한 단계 진행시키는 함수를 반환한다. 그리고 이전 단계에서 `yield`되어 나온 값은 '다음' 단계에 그대로 돌려 보낸다. 이를테면 (`yield` 시점의 값이 무엇이든) `yield 8`은 8, `yield b`는 `b`가 될 것이다.

이제 `*foo()`, `*bar()` 각자의 코드 덩이를 인터리빙하면 어떤 일이 벌어질지 실험해보자. 일단 따분하기 짝이 없는 기본 케이스지만 (1장에서 했던 것처럼) `*foo()`가 `*bar()`보다 먼저 실행을 마치는지 확인한다.

---

```
// 'a'와 'b'를 리셋한다.
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

// 먼저 '*foo()'를 완전-실행한다.
s1();
s1();
```

```
s1();
```

```
// 다음은 '*bar()' 차례다.
```

```
s2();
```

```
s2();
```

```
s2();
```

```
s2();
```

```
console.log( a, b ); // 11 22
```

---

결과는 1장과 마찬가지로 11 22다. 자, 그럼 인터리빙 순서를 뒤섞고 a, b의 최종 값이 어떻게 변하는지 살펴보자.

---

```
// 'a'와 'b'를 리셋한다.
```

```
a = 1;
```

```
b = 2;
```

```
var s1 = step( foo );
```

```
var s2 = step( bar );
```

```
s2(); // b—;
```

```
s2(); // yield 8
```

```
s1(); // a++;
```

```
s2(); // a = 8 + b;
```

```
    // yield 2
```

```
s1(); // b = b * a;
```

```
    // yield b
```

```
s1(); // a = b + 3;
```

```
s2(); // b = a * 2;
```

---

정답을 공개하기 전에 여러분 스스로 a, b 값을 유추할 수 있어야 한다. 킨닝 금지!

---

```
console.log( a, b ); // 12 18
```

---



`s1()`, `s2()` 호출 순서를 재배치하면 몇 가지 결과 조합이 나올지 독자 여러분 스스로 연습삼아 계산해보기 바란다. 항상 `s1()`은 3회, `s2()`는 4회 호출해야 한다는 점 잊지 말자. `next()`와 `yield` 사이의 대응 관계는 이미 설명하였다.

여러분이 이런 정도로 인터리빙을 일으켜 엄청나게 이해하기 어려운 코드를 수고스럽게 작성할 일은 거의 없겠지만, 하나의 훈련이라 생각하고 실습해보면 재미도 있으면서 다중 제너레이터가 동일한 공유 스코프 내에서 동시에 실행되는 원리를 좀 더 확실히 이해할 수 있을 것이다. 이러한 기능이 아주 유익하게 쓰일 데가 있다.

## 4.2 값을 제너레이터링

제너레이터가 값을 만들어내는 용도로 쓸 수 있다는 흥미로운 사실을 배웠다. 비록 이 장에서 얘기할 핵심 주제는 아니지만, 이러한 용도 때문에 제너레이터란 이름이 붙게 되었으므로<sup>03</sup> 기본을 확실하게 짚고 가야 성의없는 저자라는 소리를 안 들을 것 같다.

이 절에서는 화제를 잠시 이터레이터로 전환하여 이터레이터와 제너레이터의 관계, 그리고 제너레이터로 어떻게 값들을 만들어내는지<sup>generate</sup> 둘러본다.

### 4.2.1 제조기와 이터레이터

값이 모두 자신의 앞엿 값과 어떤 관계가 분명히 정의된, 일련의 값들을 생산<sup>produce</sup>하려고 한다. 그렇게 하려면 일단 가장 마지막에 생산한 값을 기억하는 상태성 생산기<sup>stateful producer</sup>가 필요하다.

다음과 같이 함수 클로저로 구현하면 가장 직관적이다(본 시리즈 '[스코프와 클로저](#)' 참고).

---

```
var gimmeSomething = (function(){
```

<sup>03</sup> 역자주\_원어 generator의 동사형 generate는 만들어내다, 발생시키다, 일으키다 등의 뜻을 가진 단어임으로 기본적으로 제너레이터란 이름 자체가 '원가를 만들어내는 장치'라는 의미로부터 유래한 것입니다.

```

var nextVal;

return function(){
  if (nextVal === undefined) {
    nextVal = 1;
  }
  else {
    nextVal = (3 * nextVal) + 6;
  }

  return nextVal;
};
})();

gimmeSomething(); // 1
gimmeSomething(); // 9
gimmeSomething(); // 33
gimmeSomething(); // 105

```

---



여기서 `nextVal` 계산 로직을 더 단순화할 수 있지만 개념적으로 ‘다음’ `gimmeSomething()` 호출이 일어나기 전에 ‘다음 값’(nextVal)을 계산할 일은 없을 것이다. 단순 숫자값보다는 더 영속적(persistent)이고 자원-한정적인(resource-limited) 값을 내는 생산기가 더 일반적이어서 자원 누수가 발생할 우려가 있기 때문이다.

임의의 숫자를 마구 만들어내는 코드는 현실감이 떨어진다. 데이터 소스에서 레코드를 만들어낸다면? 그래도 별반 차이는 없을 듯하다.

사실 이런 작업들은 이터레이터로 해결 가능한, 아주 일반적인 설계 패턴이다. 이터레이터는 생산자로부터 일련의 값들을 받아 하나씩 거치기 위한, 명확한 well-defined 인터페이스다. 대다수의 다른 언어처럼 자바스크립트에서도 이터레이터 인터페이스는 생산기에서 다음 값이 필요할 때마다 `next()`를 호출한다.

다음은 숫자를 만들어내는 생산기에 표준 이터레이터 인터페이스를 적용한 예제다.

---

```

var something = (function(){

```

```

var nextVal;

return {
  // 'for..of' 루프에서 필요하다.
  [Symbol.iterator]: function(){ return this; },

  // 표준 이터레이터 인터페이스 메소드
  next: function(){
    if (nextVal === undefined) {
      nextVal = 1;
    }
    else {
      nextVal = (3 * nextVal) + 6;
    }

    return { done:false, value:nextVal };
  }
};
})();

something.next().value; // 1
something.next().value; // 9
something.next().value; // 33
something.next().value; // 105

```



[Symbol.iterator]: .. 같은 코드가 필요한 이유는 190 페이지 "이터러블"에서 설명한다. [...] 구문은 계산된 프로퍼티명<sup>computed property name</sup>(본 시리즈 '[this와 객체 프 로토타입](#)' 참고)이라 하며 객체 리터럴 정의를 표현식으로 평가하여 그 결과를 프로퍼티명으로 삼는다. 또한 Symbol.iterator는 ES6에서 사전 정의한 특수한 Symbol 값이다(본 시리즈 '[ES6과 그 이후](#)' 참고).

next ( )를 호출하면 프로퍼티가 2개인 객체가 반환된다. done은 이터레이터 완료 상태를 가리키는 불리언 값이고 value는 순회값이다.

ES6는 for..of 루프를 지원하여 표준 이터레이터를 자동으로 기존 루프 구문 형태로 쓸 수 있다.

---

```
for (var v of something) {
  console.log( v );

  // 무한 루프가 되지 않게 하라!
  if (v > 500) {
    break;
  }
}

// 1 9 33 105 321 969
```

---



something 이터레이터는 늘 `done:false`를 반환하므로 `for..of`는 무한 반복한다. 그래서 조건부 `break` 문을 넣은 것이다. 이터레이터가 끝없이 실행되는 건 좋은데 한정된 값들에 대해 실행되다가 마지막에 `done:true`를 반환하는 이터레이터도 있다.

`for..of` 루프는 매번 자동으로 `next()`를 호출하다가(`next()`에 아무 값도 넘기지 않는다) `done:true`를 받으면 그 자리에서 멈춘다. 일습의 데이터를 순회할 때 꽤 편하다.

물론, 이터레이터를 직접 수동 순회하면서 `next()` 호출 후 `done:true` 여부 체크로 중단 시점을 판단할 수도 있다.

---

```
for (
  var ret;
  (ret = something.next()) && !ret.done;
) {
  console.log( ret.value );

  // 무한 루프가 되지 않게 하라!
  if (ret.value > 500) {
    break;
  }
}

// 1 9 33 105 321 969
```

---



수동으로 하면 결코 ES6 `for...of` 루프만큼 우아하다고 할 순 없지만 `next(...)`를 호출할 때 어떤 값들을 넘겨야 할 경우 유용하다.

손수 이터레이터를 제작해도 상관없지만, ES6부터 배열 같은 자바스크립트 내장 자료 구조 대부분에는 기본 이터레이터가 장착되어 있다.

---

```
var a = [1,3,5,7,9];
```

```
for (var v of a) {
  console.log( v );
}
// 1 3 5 7 9
```

---

`for...of` 루프는 `a`의 이터레이터를 추출하여 알아서 `a`의 원소를 순회한다.



일반 객체엔 배열처럼 기본 이터레이터가 없다. ES6 명세에서 왜 의도적으로 누락되었는지 이상하게 생각되었지만 그 이유까지 이 책에서 논하기는 너무 심오한 주제다. 객체 프로퍼티를 (특별한 순서없이) 순회하는 게 목적이라면 `Object.keys(...)`가 반환할 배열로 `for (var k of Object.keys(obj)) { ...}`하면 된다. `for...in`과 달리 `Object.keys(...)`은 `[[Prototype]]` 연쇄에 있는 프로퍼티는 배제한다는 사실만 빼고는, `for...of`, `for...in` 둘 다 객체의 키 값을 순회하는 로직은 엇비슷하다.

## 4.2.2 이터러블

이전 예제 `something` 처럼 `next()` 메소드로 인터페이스하는 객체를 ‘이터레이터 <sup>iterator</sup>’라고 한다. 그러나 순회 가능한 이터레이터를 포괄한 객체, ‘이터러블 <sup>iterable</sup>’이 더 밀접한 용어다.

ES6부터 이터러블은 특별한 ES6 심볼값 `Symbol.iterator`라는 이름을 가진 함수를 지니고 있어야 이 함수를 호출하여 이터러블에서 이터레이터를 가져올 수 있다. 필수 요건은 아니지만 일반적으로 함수를 호출할 때마다 방금 구워낸 따끈따



끈한 새 이터레이터를 내어준다.

앞서 보았던 예제에서 `a`가 바로 이터러블이다. `for..of` 루프는 자동으로 `Symbol.iterator` 함수를 호출하여 이터레이터를 생성한다. 물론, 수동으로 함수를 호출하여 이 함수가 반환한 이터레이터를 사용할 수도 있다.

---

```
var a = [1,3,5,7,9];

var it = a[Symbol.iterator]();

it.next().value; // 1
it.next().value; // 3
it.next().value; // 5
..
```

---

`something` 정의부에 이런 코드가 있던 걸 기억할 것이다.

---

```
[Symbol.iterator]: function(){ return this; }
```

---

약간 헷갈리게 생겼지만 `something` 값(`something` 이터레이터의 인터페이스)을 이터러블로 만드는 코드로, 이제 `something`은 이터러블이면서 동시에 이터레이터다. 그리고 `for..of` 루프에 `something`을 넘긴다.

---

```
for (var v of something) {
    ..
}
```

---

`for..of` 루프 구문은 `something`이 이터러블이라는 전제 하에 `Symbol.iterator` 함수가 있는지 찾아보고 호출한다. 이 함수는 그냥 `return this`하여 스스로를 돌려주기 때문에 `for..of` 루프는 속사정을 알 길이 없다.

## 4.2.3 제너레이터 이터레이터

다시 이터레이터의 맥락에서 제너레이터를 살펴보자. 제너레이터는 일종의 값을 생산하는 공장이며, 이렇게 만들어진 값들은 이터레이터 인터페이스의 `next()`를 호출하여 한번에 하나씩 추출할 수 있다.

따라서 엄밀히 말하면 제너레이터 자체는 이터러블이 아니지만 아주 흡사해서 제너레이터를 실행하면 이터레이터를 돌려받게 된다.

---

```
function *foo(){ .. }
```

```
var it = foo();
```

---

그러므로 무한 수열 생산기 `something`은 다음과 같이 구현할 수 있다.

---

```
function *something() {  
  var nextVal;  
  
  while (true) {  
    if (nextVal === undefined) {  
      nextVal = 1;  
    }  
    else {  
      nextVal = (3 * nextVal) + 6;  
    }  
  
    yield nextVal;  
  }  
}
```

---



보통 자바스크립트 프로그램에서 `while..true` 루프를 이렇게 `break`, `return` 문도 없이 사용하면 자칫 동기적인 무한 루프에 빠지면서 브라우저 UI를 얼어붙게 할 수 있기 때문에 아주 나쁜 코드다. 그러나 제너레이터는 루프 안에 `yield`만 있으면 전혀 염려할 일이 없다. 어차피 제너레이터가 순회할 때마다 멈추면서 메인 프로그램, 또는 이벤트 루프 큐에 바통을 넘겨줄 테니 말이다. 말하자면 “자바스크립트 프로그래밍에 `while..true`를 컴백시킨 주역이 바로 제너레이터다”.

한결 더 깔끔하고 간단하지 않은가? 제너레이터는 `yield`를 만나면 일단 멈추기 때문에 `function *something()`의 상태(스코프)는 그대로 유지된다. 다시 말해, 호출할 때마다 변수 상태값을 보존하기 위해 습관적으로 클로저 구문을 남발할 필요가 없다.

개발자가 직접 이터레이터 인터페이스를 작성할 필요가 없으므로 단순할 뿐만 아니라 사실 내용을 더 분명하게 표현한 추론적인<sup>reason-able</sup> 코드다. 이를테면, `while..true` 루프 하나만 봐도 이 제너레이터의 목적이 무한 실행 중 언제라도 요청을 하면 값을 만들어 내는 것이라는 사실을 금세 알 수 있다.

화려한 새 제너레이터 `*something()`을 `for..of` 루프에 얹어도 기본적으로 작동 방식은 똑같다.

---

```
for (var v of something()) {  
  console.log( v );  
  
  // 무한 루프가 되지 않게 하라!  
  if (v > 500) {  
    break;  
  }  
}
```

---

```
// 1 9 33 105 321 969
```

---

`for (var v of something())`를 눈여겨 보자. 전처럼 `something`을 어떤 값으로 참조한 게 아니라 `*something()` 제너레이터를 호출해서 `for..of` 루프가 사용할 이터레이터를 얻는다.

그런데 제너레이터와 루프 간의 이러한 상호 작용에 관해서 두 가지 의문점이 생긴다.

- `for (var v of something)`처럼 쓰면 어떻게 될까...? 여기서 `something`은 제너레이터지 이터러블이 아니므로 `something()`을 호출해서 `for..of` 루프가 순회할 생산기

를 만들어야 한다.

- `something()`을 호출하면 이터레이터가 만들어지지만 정작 `for..of` 루프가 원하는 건 이터러블 아닌가? 맞다. 제너레이터의 이터레이터에도 `Symbol.iterator` 함수가 있고 (이전에 정의했던 `something` 이터러블처럼) 기본적으로 `return this`한다. 한 마디로, 제너레이터의 이터레이터도 이터러블이다!

## 제너레이터 멈춤

좀 전에 `*something()` 제너레이터의 이터레이터 인터페이스는 루프 내에서 `break`를 호출한 이후엔 영원히 정지<sup>suspended</sup> 상태가 될 것 같아 보인다.

그러나 이 때 알아서 척척 처리해주는 숨겨진 기능이 하나 있다. 일반적으로 `break`, `return`, 또는 잡히지 않은 예외로 인해 `for..of` 루프가 “비정상 완료 *abnormal completion*” (즉, “이르게 종료<sup>early termination</sup>”)되면 제너레이터의 이터레이터를 중지하도록 신호를 준다.



엄밀히는 루프가 정상 완료될 경우에도 `for..of` 루프는 이터레이터에게 신호한다. 제너레이터 관점에서 어차피 자신의 이터레이터가 먼저 끝나야 `for..of` 루프 역시 완료되므로 별 의미는 없다. 그러나 커스텀 이터레이터는 `for..of` 루프를 사용하는 코드로부터 부가적인 신호를 받는 경우가 있다.

`for..of` 루프가 자동으로 전송하는 신호를 수동으로 이터레이터에 보내야 할 경우도 있다. 그럴 때 `return(...)`을 호출한다.

제너레이터가 외부적으로 완료된 다음에도 내부에서 `try..finally` 절을 사용하면 실행할 수 있다. 자원(DB 접속 등)을 정리할 때 유용한 기법이다.

---

```
function *something() {
  try {
    var nextVal;

    while (true) {
      if (nextVal === undefined) {
        nextVal = 1;
      }
    }
  }
}
```

```

    }
    else {
        nextVal = (3 * nextVal) + 6;
    }

    yield nextVal;
}
}
// 정리 코드
finally {
    console.log( "정리 완료!" );
}
}

```

---

for..of 루프에서 break하면 finally 절로 이동할 것이다. 외부에서 return(..) 문을 써서 수동으로 제너레이터의 이터레이터 인스턴스를 멈추게 할 수 있다.

---

```

var it = something();
for (var v of it) {
    console.log( v );

    // 무한 루프가 되지 않게 하라!
    if (v > 500) {
        console.log(
            // 제너레이터의 이터레이터를 마친다.
            it.return( "Hello World" ).value
        );
        // 여기서 'break'문은 필요없다.
    }
}
// 1 9 33 105 321 969
// 정리 완료!
// Hello World

```

---

it.return(..)하면 제너레이터 실행은 즉시 끝나고 finally 절로 옮겨간다.

또 `return(...)`에 전달한 인자값이 반환값이 되어 예제처럼 “Hello World”가 바로 나온다. 제너레이터의 이터레이터는 이미 `done:true`이므로 `break` 문을 넣지 않아도 되며 `for...of` 루프는 다음 순회를 끝으로 막을 내린다.

제너레이터는 이렇게 양면의 거울처럼 ‘생산된 값을 소비’하는 용도로도 쓰이지만 수많은 용도 중 하나에 지나지 않고 솔직히 이 책의 맥락 상 주제로 다룰 만한 내용은 아니다.

어쨌든 여러분이 제너레이터의 작동 원리와 체계를 완전히 이해했으리라 생각하며, 다음 절부터 제너레이터를 비동기 동시성에 적용하는 문제를 집중적으로 다룬다.

### 4.3 제너레이터를 비동기적으로 순회

과연 제너레이터는 비동기 코딩 패턴과 무슨 상관이 있으며, 어떻게 콜백의 문제점을 해결할 수 있다는 말인가? 자, 지금부터 이 중요한 질문의 답을 찾으러 떠나겠다.

3장에서 봤던 콜백식 코드를 다시 보자.

---

```
function foo(x,y,cb) {
  ajax(
    "http://some.url.1/?x=" + x + "&y=" + y,
    cb
  );
}

foo( 11, 31, function(err,text) {
  if (err) {
    console.error( err );
  }
  else {
    console.log( text );
  }
})
```

```
    }  
  } );  
}
```

---

이 코드의 작업 흐름을 제너레이터로 표현하면 다음과 같다.

---

```
function foo(x,y) {  
  ajax(  
    "http://some.url.1/?x=" + x + "&y=" + y,  
    function(err,data){  
      if (err) {  
        // '*main()'으로 에러를 던진다.  
        it.throw( err );  
      }  
      else {  
        // 수신한 'data'로 '*main()'을 재개한다.  
        it.next( data );  
      }  
    }  
  );  
}  
  
function *main() {  
  try {  
    var text = yield foo( 11, 31 );  
    console.log( text );  
  }  
  catch (err) {  
    console.error( err );  
  }  
}  
  
var it = main();  
  
// 모두 시작한다!  
it.next();
```

---

콜백 코드가 더 길고 복잡해 보이더라도 처음부터 지나치게 위축될 필요는 없다.  
제너레이터 코드가 몇 갑절 더 좋다! 내가 설명할 내용이 많은 것 빼고...

우선 이 예제의 핵심은 바로 다음 코드다.

---

```
var text = yield foo( 11, 31 );  
console.log( text );
```

---

어떻게 작동할지 한번 상상해보라. 보통 함수 `foo(...)`를 호출하고 다시 이 함수는 AJAX를 호출하여 `text`를 넘겨받는다. 그런데 `ajax`는 비동기 함수잖!

어떻게 이런 일이...? 1장 앞부분에도 거의 똑같은 코드를 소개한 적 있다.

---

```
var data = ajax( "..url 1.." );  
console.log( data );
```

---

그때 이 코드는 작동하지 않았었다! 이제 차이점이 눈에 들어오는가? 바로 제너레이터에 쓴 `yield`다.

정말 놀라운 마술이다! 겉으로 봐선 중단적/동기적인 코드인데 실제로 전체 프로그램을 중단시키지 않는다! 제너레이터 자신이 알아서 코드를 멈춤/중단하는 것이 바로 비법이다.

일단 `yield foo(11,31)`에서 `foo(11,31)` 호출이 일어나고 반환값은 없으므로(즉 `undefined`), `data`를 요청하기 위해 호출을 했지만 실제로는 `yield undefined`를 수행한 셈이다. 아직은 이 코드가 뭔가 의미있는 일을 하기 위해 `yield`된 값이 필요한 상태는 아니니 괜찮다. 이 부분은 뒷부분에서 다시 정리할 것이다.

여기서 `yield`는 메시지 전달 수단이 아닌, 멈춤/중단을 위한 흐름 제어 수단으로 사용한 것이다. 실제로는 제너레이터가 다시 시작한 이후로 메시지 전달은 단방향으로만 이루어진다.

따라서 `yield`를 만나 멈추면서 제너레이터는 “내가 나중에 어떤 값을 갖고 와서 변수 `text`에 할당해야 하나요?”라고 묻는다. 누구에게 하는 질문일까?



foo(...)를 보자. AJAX 요청이 성공하면 다음 한 줄이 실행된다.

---

```
it.next( data );
```

---

응답 데이터 수신과 동시에 제너레이터는 재개되고 좀 전에 멈췄던 yield 표현식은 이 응답 데이터로 즉시 채워진다. 그 뒤, 제너레이터 코드를 다시 시작하면서 이 값은 지역 변수 text에 할당된다.

정말 쿨하지 않은가?

잠시 눈을 감고 음미해보자. 아무리 봐도 제너레이터 내부의 코드는 (yield 키워드를 제외하고) 동기적으로 작동할 것 같지만 그 속을 들여다보니 foo(...) 안에서는 완전히 비동기적으로 작동 가능하다.

바로 이거다! 비동기성을 우리 두뇌가 허락하는 순차적/동기적 방향으로 표현할 수 없었던, 콜백의 단점을 거의 완벽하게, 그것도 한번에 보완한 솔루션이다.

본질적으로 비동기성을 하나의 구현 상세로 추상화했기 때문에 개발자가 동기/순차적으로 흐름 제어를 추론할 수 있게 된 것이다. “AJAX 요청을 하고 다 끝나면 응답을 출력하라.” 예제에서는 흐름 제어를 두 단계로 나타냈지만 비슷한 식으로 확장하여 여러 단계를 제한없이 표현할 수 있다.



매우 중요한 깨달음이니 피부로 느껴질 때까지 방금 전 세 단락을 반복해서 열독하기 바란다!

### 4.3.1 동기적 에러 처리

그런데 좀 전의 제너레이터 코드는 더 큰 혜택을 우리에게 선사한다. 제너레이터 안쪽에 있는 try..catch 구문에 주목하자.

---

```
try {
  var text = yield foo( 11, 31 );
  console.log( text );
}
catch (err) {
  console.error( err );
}
```

---

이런 코드가 작동은 할까? `foo( .. )`는 비동기 함수여서 3장에서 보았듯이 `try..catch`로 비동기 에러를 잡을 도리가 없다.

`foo( .. )`가 실행을 끝내고 AJAX 응답을 `text`에 할당할 준비를 마칠 때까지 `yield`가 이 할당문을 멈추게 하는 방법은 이미 앞에서 이야기했다. 그런데 놀라운 건 이 `yield`로 제너레이터가 에러를 잡을 수 있게 잠시 멈추게 할 수도 있다는 점이다. 즉, 다음 코드처럼 에러를 제너레이터에 던지는 것이다.

---

```
if (err) {
  // '*main()'으로 에러를 던진다.
  it.throw( err );
}
```

---

이러한 제너레이터의 `yield`-멈춤 기능은 비동기 함수 호출로부터 넘겨받은 값을 동기적인 형태로 `return`하게 해줄 뿐만 아니라 비동기 함수 실행 중 발생한 에러를 동기적으로 `catch`할 수 있게도 해준다.

제너레이터 안쪽으로 에러를 던질 수 있다면 제너레이터 밖으로도 가능할까? 바로 그렇다.

---

```
function *main() {
  var x = yield "Hello World";

  yield x.toLowerCase(); // 예외를 일으킨다!
```

```

}

var it = main();

it.next().value; // Hello World

try {
    it.next( 42 );
}
catch (err) {
    console.error( err ); // TypeError
}

```

---

물론 억지로 예외를 일으키지 않고 throw .. 구문을 이용해서 수동으로 에러를 던져도 된다.

제너레이터에 throw (..)로 던져넣은 해당 에러를 잡는 것도 가능해서 1차적으로는 제너레이터가 에러를 처리하겠지만 그렇지 못할 경우엔 2차적으로 이터레이터 코드가 필히 처리해야 한다.

---

```

function *main() {
    var x = yield "Hello World";

    // 이 코드는 실행되지 않는다.
    console.log( x );
}

var it = main();

it.next();

try {
    // '*main()'은 이 에러를 처리할까? 한번 보자!
    it.throw( "허격" );
}
catch (err) {
    // 아니다, 처리하지 않았다!

```

```
    console.error( err ); // 허걱
}
```

---

비동기 코드에서 난 에러를 동기적인 모양새로 처리할 수 있어서 코드 가독성, 추론성(reason-ability) 면에서 매우 큰 강점이다.

## 4.4 제너레이터 + 프라미스

제너레이터를 비동기적으로 순회할 수 있다는 사실만으로도 순차적 추론성(sequential reason-ability) 측면에서 콜백 범벅인 스파게티 코드에 비해 엄청난 발전이다.

하지만 정말 중요한 것이 하나 빠졌다. 바로 프라미스의 믿음성과 조합성이다.

염려 붙들어 매시라! ES6의 백미가 바로 제너레이터(동기적 형태의 비동기 코드)와 프라미스(믿음성과 조합성)의 만남이다.

둘을 어떻게?

3장의 프라미스식 AJAX 예제를 다시 보자.

---

```
function foo(x,y) {
    return request(
        "http://some.url.1/?x=" + x + "&y=" + y
    );
}

foo( 11, 31 )
.then(
    function(text){
        console.log( text );
    },
    function(err){
        console.error( err );
    }
);
```

---

이전 제너레이터식 AJAX 예제에서 `foo(..)`는 아무것도 반환하지 않았고 (undefined 반환) 이터레이터 제어 코드는 `yield`된 값에 관심이 없었다.

하지만 이 예제에서 프라미스-인식형 함수 `foo(..)`는 AJAX 호출 이후 프라미스를 반환한다. 따라서 `foo(..)`로 프라미스를 생성하고 제너레이터에서 `yield`해서 이터레이터 제어 코드가 이 프라미스를 받게 하면 뭔가 이루어질 것 같다.

그런데 이터레이터는 프라미스로 무슨 일을 할까?

이터레이터는 프라미스가 귀결(이름/버림)되기를 리스닝하고 있다가 제너레이터를 이름 메시지로 재개하든지, 아니면 제너레이터로 버림 사유로 채워진 에러를 던진다.

중요한 내용이라 다시 말하겠다. 프라미스, 제너레이터를 최대한 활용하는 가장 자연스러운 방법은 우선 프라미스를 `yield`한 다음 이 프라미스로 제너레이터의 이터레이터를 제어하는 것이다.

자, 한번 시도해보자! 먼저, 프라미스-인식형 `foo(..)`와 `*main()` 제너레이터를 한데 모으자.

---

```
function foo(x,y) {
  return request(
    "http://some.url.1/?x=" + x + "&y=" + y
  );
}

function *main() {
  try {
    var text = yield foo( 11, 31 );
    console.log( text );
  }
  catch (err) {
    console.error( err );
  }
}
```

---

이 리팩토링 과정에서 `*main()` 코드는 전혀 변경되지 않았다는 사실이 가장 주목할 부분이다! 제너레이터 이면에서 값을 `yield`하고 내보내는 건 구현 상세로 직이므로 어떤 일들이 일어나는지 관심가질 필요가 없다.

그나저나 `*main()`은 어떻게 실행할까? `yield`된 프라미스를 받아 귀결 시 제너레이터가 다시 움직이게끔 하려면 아직 공사가 덜 됐다. 일단 수동으로 해보자.

---

```
var it = main();

var p = it.next().value;

// 'p' 프라미스가 귀결될 때까지 기다린다.
p.then(
  function(text){
    it.next( text );
  },
  function(err){
    it.throw( err );
  }
);
```

---

이러고 보니 그리 힘든 일이 아니다.

이 코드는 좀 전에 예러 우선 콜백 방식으로 제너레이터를 수동으로 제어했던 예제와 비슷하다. `if (err) { it.throw..` 대신 프라미스는 이미 이름(성공)과 버림(실패)을 분기해주지만 이터레이터를 제어하는 부분은 똑같다.

자, 이제 앞에서 얼버무리고 넘어갔던 몇 가지 중요한 세부분을 설명한다.

`*main()`에는 프라미스를 인식하는 단계가 하나만 있다는 사실을 여러분과 나눈 알고 있었고 이 사실을 십분 활용했다는 점이 가장 중요한 포인트다. 아주 많은 단계가 있더라도 제너레이터를 프라미스로 구동해야 한다면? 당연히 제너레이터별로 프라미스 연쇄를 수동으로 작성하고 싶지는 않을 것이다. 순회 제어를 반복하

면서 (루프를 돌리면서) 프라미스가 나올 때마다 일단 귀결되길 기다렸다가 진행할 수만 있다면 금상첨화일 것 같다.

그런데 `it.next(...)` 도중 제너레이터에서 (고의든 실수든) 에러를 나면? 그냥 종료해야 할까, 아니면 에러를 잡아 돌려주어야 할까? 마찬가지로 제너레이터에 프라미스 버림을 `it.throw(...)` 했는데 처리되지 않고 곧바로 다시 튀어나오게 하려면 어떻게 해야 할까?

#### 4.4.1 프라미스-인식형 제너레이터 실행기

아마 책장을 넘길수록 여러분의 마음도 간절해질 것이다. “아, 이런 일을 대신 해 줄 유틸이 있으면 참 좋겠는데...” 당연하다. 이렇게 중요한 패턴을 잘못 코딩하는 (아니면 계속 에러가 나서 결국 포기하는) 일이 있어서는 안 될 테니, 지금까지 설명한 로직으로 프라미스를 `yield`하는 제너레이터를 실행하도록 잘 설계된 유틸을 찾아 쓰는 것이 최선이다.

몇몇 프라미스 추상화 라이브러리에서 그런 유틸을 제공하는데, 내가 작성한 `asynquence` 라이브러리(부록 A 참고)도 그 중 하나다.

하지만 처음부터 남의 유틸을 가져다 쓸 수는 없으니 `run(...)`을 호출할 수 있는 독자적인 스탠드얼론<sup>standalone</sup> 유틸을 작성해보자.

---

// 코드 개량에 큰 도움을 주신 벤자민 그루엔바움(깃허브 @benjaminr)씨에게 감사드립니다!

```
function run(gen) {
  var args = [].slice.call( arguments, 1), it;

  // 현재 콘텍스트에서 제너레이터를 초기화한다.
  it = gen.apply( this, args );

  // 제너레이터 완료를 의미하는 프라미스를 반환한다.
  return Promise.resolve()
    .then( function handleNext(value){
```

```

// 다음 yield된 값까지 실행한다.
var next = it.next( value );

return (function handleResult(next){
    // 제너레이터 실행이 끝났다면,
    if (next.done) {
        return next.value;
    }
    // 아니면 계속 실행한다.
    else {
        return Promise.resolve( next.value )
            .then(
                // 성공 시 귀결값을 제너레이터로 반환하면서
                // 비동기 루프를 재개한다.
                handleNext,

                // 'value'가 버림 프라미스면
                // 제너레이터 자신이 에러를 처리하게끔
                // 거꾸로 에러를 전파한다.
                function handleErr(err) {
                    return Promise.resolve(
                        it.throw( err )
                    )
                        .then( handleResult );
                }
            );
    }
})(next);
} );
}

```

보다시피 손수 작성하기에 코드가 꽤 복잡한데다 매번 제너레이터를 쓸 때마다 이런 코드를 반복 작성하고 싶지는 않다. 그래서 유틸/라이브러리 헬퍼가 필요한 것이다. 하지만 다만 몇 분이라도 시간을 들여 이 코드를 면밀히 분석해보고 제너레이터 + 프라미스를 어떻게 함께 다루는지 공부하기 바란다.

`run ( .. )`은 AJAX 예제의 `*main ( )`에서 다음과 같이 사용한다.



---

```
function *main() {  
    // ..  
}  
  
run( main );
```

---

이걸로 끝이다! `run( .. )` 내부 로직에 따라 주어진 제너레이터를 비동기적으로 완료될 때까지 알아서 진행한다.



`run( .. )`은 제너레이터가 완료되면 바로 귀결되는 프라미스를 반환하거나, 이 프라미스를 제너레이터가 처리하지 않으면 잡하지 않은 예러를 수신한다. 이런 기능은 여기서 밝히지 않았는데 이 장 뒷부분에서 다시 설명한다.

### ES7: `async`와 `await`?

제너레이터가 프라미스를 `yield`하고 나중에 이 프라미스가 제너레이터의 이터레이터를 제어하여 끝까지 진행하는, 이런 패턴은 매우 강력하고 쓸모가 있다. 라이브러리 유틸 헬퍼(`run( .. )`)를 쓰지 않고 이러한 기능을 구현할 수 있으면 얼마나 좋을까?

조만간 좋은 소식이 있을 듯하다. 아직은 초기 단계에 불과하지만 이 글을 쓰고 있는 현재 다음 버전의 명세 ES7에 추가 구문을 넣어달라는 의견이 제안되어 강력한 지지를 받고 있다. 세부적으로 어떻게 구현될지 장담할 순 없지만 대략 다음과 같은 형태가 될 것 같다.

---

```
function foo(x,y) {  
    return request(  
        "http://some.url.1/?x=" + x + "&y=" + y  
    );  
}  
  
async function main() {  
    try {  
        var text = await foo( 11, 31 );
```

```

        console.log( text );
    }
    catch (err) {
        console.error( err );
    }
}

main();

```

---

`main()`을 실행했던 `run(...)` 호출부는 없고(라이브러리 유틸 따위는 필요없다) 그냥 일반 함수를 호출하고 있다. 또 `main()`은 이제 제너레이터 함수 아닌, `async`라는 새로운 유형의 함수로 선언하고 프라미스를 `yield`하지 않고 프라미스가 귀결되기를 `await`(대기)한다.

`async function` 하면 프라미스를 `await`할 경우 해야 할 일, 즉 프라미스가 귀결될 때까지 (꼭 제너레이터처럼) 이 함수를 멈추게 할 거란 사실을 자동으로 인식한다. 예제에서 따로 설명하진 않았지만 `main()` 같은 비동기 함수를 호출하면 함수가 완료될 때마다 귀결되는 프라미스를 알아서 반환한다.



`async / await`는 C# 경험자라면 구문 생김새가 똑같아서 눈에 익을 것이다.

본질적으로 이 제안은 방금 전 살펴보았던 패턴을 동기적 형태의 흐름 제어 코드와 프라미스를 결합하는, 하나의 구문 체계로 정리한 것이다. 두 세상의 결합은 앞서 개략적으로 서술했던 콜백의 모든 주요 문제점을 실질적으로 해결할 최선의 방법이다.

벌써 ES7 비슷한 제안이 등장했고 활발한 기술 교류가 이루어지고 있는 것만 봐도 이러한 비동기 패턴의 영향력은 점점 더 확대될 게 뻔하다.

## 4.4.2 제너레이터에서의 프라미스 동시성

지금까지는 한 단계의 프라미스 + 제너레이터 비동기 흐름을 설명했으나 실제 비동기 코드는 보통 여러 단계로 구성된다.

신중하게 판단하지 않으면 동기적 형태의 제너레이터만 있어도 비동기적 동시성을 구현하기 충분하다는 자기 만족을 하며 결국 비최적<sup>suboptimal</sup> 성능 패턴에 만족하게 된다. 조금 더 시간을 들여 더 나은 옵션은 없는지 찾아보는 게 좋겠다.

서로 다른 두 데이터 소스에 각각 데이터를 요청 후 수신한 응답을 조합해서 다시 세 번째 요청을 전송하고 그렇게 하여 받은 마지막 응답을 출력하는 프로그램을 작성한다고 하자. 이와 비슷한 시나리오는 이미 3장 프라미스에서 제시한 적 있는데 이번에는 제너레이터 맥락에서 재고해보자.

일단 육감적으로 다음 코드가 머릿속에 떠오를 것이다.

---

```
function *foo() {
  var r1 = yield request( "http://some.url.1" );
  var r2 = yield request( "http://some.url.2" );

  var r3 = yield request(
    "http://some.url.3?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// 앞서 정의한 'run(...)' 유틸을 이용한다.
run( foo );
```

---

이 코드도 잘 실행은 되지만 주어진 시나리오를 감안하면 최적은 아니다. 왜 그럴까?

여기서 r1, r2은 동시 실행이 가능하지만(성능상 그래야 하지만) 실제로는 순차 실행

된다. 즉, `http://some.url.1` 요청이 완료된 다음에야 `http://some.url.2` URL에 AJAX로 데이터를 요청하게 된다. 두 요청은 서로 독립적이어서 가급적 동시 실행을 하는 편이 성능 상 유리하다.

그럼 제너레이터와 `yield`로 뭘 어떻게 해야 한단 말인가? `yield`는 기껏해야 코드 한 곳에서 멈추게 할 수는 있지만 동시에 두 곳에서 멈추게 하는 건 불가능하다.

가장 무난하면서 효과적인 해결책은 전체 비동기 흐름의 기반을 프라미스에 두는, 좀 더 구체적으로는 시간 독립적 형태로 상태 관리가 가능한 프라미스 본연의 능력에 맡기는 것이다(3장 80 페이지 “미래값” 참고).

가장 간단하게는,

---

```
function *foo() {
  // 두 요청을 "병렬" 실행한다.
  var p1 = request( "http://some.url.1" );
  var p2 = request( "http://some.url.2" );

  // 두 프라미스가 모두 귀결될 때까지 기다린다.
  var r1 = yield p1;
  var r2 = yield p2;

  var r3 = yield request(
    "http://some.url.3/?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// 앞서 정의한 'run(..)' 유틸을 이용한다.
run( foo );
```

---

이전 버전과 무슨 차이가 있을까? `yield` 위치를 잘 보기 바란다. `p1`, `p2`는 동시에 (“병렬로”) AJAX 요청을 하는 프라미스고 프라미스는 한번 귀결되면 그 귀결 상태

를 계속 유지하므로 어느 요청이 먼저 완료되어도 상관없다.

그 다음 잇따른 yield문 2개로 프라미스 귀결값을 (각각 r1, r2으로) 받을 때까지 대기한다. p1이 먼저 귀결되면 yield p1을 먼저 재개한 뒤 yield p2 재개를 기다린다. p2가 먼저 귀결되면 요청받을 때까지 곳곳이 귀결값을 들고 있지만 yield p1은 p1이 귀결될 때까지 우선 보류된다.

어떻게 흘러가든 p1, p2 모두 동시 실행되고 순서에 상관없이 둘 다 끝나면 그제서야 r3 = yield request... AJAX 요청으로 넘어간다.

어디서 많이 본 흐름 제어 모델같은데... 3장에서 Promise.all([ ... ]) 유틸로 구현했던 관문 패턴과 기본적으로 동일한 형태다. 따라서 흐름 제어를 이렇게 나타낼 수도 있다.

---

```
function *foo() {
  // 두 요청을 "병렬" 실행하고
  // 두 프라미스 모두 귀결될 때까지 기다린다
  var results = yield Promise.all( [
    request( "http://some.url.1" ),
    request( "http://some.url.2" )
  ] );

  var r1 = results[0];
  var r2 = results[1];

  var r3 = yield request(
    "http://some.url.3?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// 앞서 정의한 'run(..)' 유틸을 이용한다.
run( foo );
```

---



3장에서 말했듯이 `var r1 = .. var r2 = ..`를 `var [r1,r2] = results` 같은 ES6 해체 할당문으로 표기할 수 있다.

결론적으로 프라미스의 모든 동시성 능력을 제너레이터 + 프라미스 방식에서 마음껏 이용할 수 있다. 따라서 ‘이것-다음-저것’ 형태의 순차적 비동기 흐름 제어 이상의 뭔가가 필요하다 싶을 때 일단 프라미스가 최선이다.

### 프라미스 숨김

코딩 스타일 문제지만 제너레이터 내부에 얼마만큼의 프라미스 로직을 집어넣을지 신중히 판단하기 바란다. 지금까지 설명한 제너레이터로 비동기성을 나타내는 방법은, 단순 + 순차적 + 동기적인 형태의 코드를 작성하면서도 가능한 한 비동기 관련 세부분은 감추고자 하는 의도가 지배적이다.

예컨대, 다음과 같이 작성한 코드가 더 깔끔하다.

---

```
// 제너레이터 아닌, 일반 함수다.
function bar(url1,url2) {
  return Promise.all( [
    request( url1 ),
    request( url2 )
  ] );
}

function *foo() {
  // 프라미스형 동시성 관련 세부분은 감춘다.
  // 'bar(..)' 내부
  var results = yield bar(
    "http://some.url.1",
    "http://some.url.2"
  );

  var r1 = results[0];
  var r2 = results[1];
}
```

```

var r3 = yield request(
  "http://some.url.3/?v=" + r1 + "," + r2
);

console.log( r3 );
}

// 앞서 정의한 'run(..)' 유틸을 이용한다.
run( foo );

```

---

\*foo ( )가 내부적으로 bar ( )에게 어떤 결과를 요청한 뒤 yield로 기다리게 하면 코드가 훨씬 더 깔끔해진다. 그 과정에서 프라미스 조합을 이용하는 Promise.all ( [ .. ] )의 내부에서 벌어지는 일들은 신경 쓸 필요가 없다.

비동기성, 사실상 프라미스를 하나의 구현 상세로 보는 셈이다.

프라미스 로직을 어떤 함수 안에 감추고 이 함수를 제너레이터에서 그냥 호출하는 식으로 짜면 흐름 제어를 정교하게 다루어야 할 때 특히 유용하다. 예를 들면,

---

```

function bar() {
  Promise.all( [
    baz( .. )
    .then( .. ),
    Promise.race( [ .. ] )
  ] )
  .then( .. )
}

```

---

이렇게 생긴 필수 로직을 제너레이터 안으로 몽땅 밀어넣는 건 일단 제너레이터를 사용하는 이유를 무의미하게 만든다. 세부 로직은 반드시 제너레이터 코드로부터 멀찌감치 떨어뜨리고 의도적으로 추상화시켜 놓아야 더 상위 수준의 작업을 표현하기가 편하다.

기능 좋고 성능 좋은 코드를 작성하는 것도 좋지만 항상 되도록 추론성과 유지 보

수성이 나온 코드를 작성하려고 노력해야 한다.



간결함이라는 대가로 복잡도는 증가하기 마련이므로 프로그래밍에서 추상화가 꼭 유익하다고 말할 수는 없다. 개인적으로 나는 제너레이터 + 프라미스 형태의 비동기 코드라면 추상화가 큰 도움이 된다고 본다. 그러나 어디까지나 충고는 충고일 뿐 여러분 각자가 처한 상황에 집중하고 팀원들과 상의하여 현명한 판단을 내리기 바란다.

## 4.5 제너레이터 위임

제너레이터 내부에서 일반 함수를 호출하는 식으로 (비동기 프라미스 흐름 같은) 구현 상세를 추상화시켜 로직을 감추는 기법이 유용한 이유를 앞서 살펴보았다. 그런데 일반 함수를 이용하면 일반 함수 규칙에 맞게끔 작동하기 때문에 제너레이터처럼 yield에서 멈추게 할 수 없는 점이 가장 큰 단점이다.

다음과 같이 어떤 제너레이터를 다른 제너레이터에서 `run(..)` 헬퍼를 이용하여 호출하는 경우를 떠올리면 무슨 말인지 이해가 될 것이다.

---

```
function *foo() {
  var r2 = yield request( "http://some.url.2" );
  var r3 = yield request( "http://some.url.3/?v=" + r2 );

  return r3;
}

function *bar() {
  var r1 = yield request( "http://some.url.1" );

  // 'run(..)'를 통해 '*foo()'에 "위임한다".
  var r3 = yield run( foo );

  console.log( r3 );
}

run( bar );
```

---



`*bar()`에서 `run(..)` 유틸로 `*foo()`를 실행하고 있다. 이 제너레이터의 실행이 완료되면(또는 예러가 나면) 귀결된 프라미스를 `run(..)`이 반환할 테니 `run(..)` 인스턴스를 통해 다른 `run(..)`에서 나온 프라미스를 `yield`해서 내보내면 자동으로 `*foo()`가 끝날 때까지 `*bar()`를 멈추게 할 거란 계산이다.

그러나 `*foo()` 호출을 `*bar()` 안으로 합하고자 한다면 `yield-위임`<sup>yield-delegation</sup>이라는 더 좋은 방법이 있다. `yield * __` 형태(중간에 \*이 더 있다)의 특수 구문이다. 방금 전 예제에 적용하기 전에 더 간단한 코드를 먼저 보자.

---

```
function *foo() {
  console.log( "'*foo()' 시작" );
  yield 3;
  yield 4;
  console.log( "'*foo()' 끝" );
}

function *bar() {
  yield 1;
  yield 2;
  yield *foo(); // 'yield'-위임!
  yield 5;
}

var it = bar();

it.next().value; // 1
it.next().value; // 2
it.next().value; // '*foo()' 시작
                // 3
it.next().value; // 4
it.next().value; // '*foo()' 끝
                // 5
```

---



내가 `function* foo()`보다 `function *foo()` .. 형태를 더 선호하는 이유를 기억할 것이다. 여기서도 나는 (다른 책 저자들과 달리) `yield* foo()` 대신 `yield *foo()`이 더 좋아보인다. \*를 어디에 붙일지는 순수한 코딩 스타일 문제고 각자 알아서 하면 될 일이긴 하지만 아무리 봐도 내 방식이 더 나은 것 같다.

`yield *foo()` 위임은 어떻게 작동하는 걸까?

우선, 보다시피 `foo()`를 호출하면 이터레이터가 생성되고 `yield *`는 이터레이터 인스턴스에 관한 제어권을 또 다른 이터레이터 `*foo()`에게 위임한다(넘긴다).

그래서 두 번째 `it.next()` 호출까지는 `*bar()`를 제어하지만, 세 번째 호출은 `*foo()`를 시작하고 `*bar()` 대신 `*foo()`를 제어한다. 즉, `*bar()`가 자신의 순회 권한을 `*foo()`에게 위임했다고 볼 수 있기 때문에 위임이라는 말이 붙은 것이다.

`it` 이터레이터로 전체 `*foo()` 이터레이터를 훑고나면 자동으로 제어권은 `*bar()`로 넘어온다.

이전의 3개 순차 AJAX 요청 예제에 위임을 적용하면,

---

```
function *foo() {
  var r2 = yield request( "http://some.url.2" );
  var r3 = yield request( "http://some.url.3/?v=" + r2 );
  return r3;
}

function *bar() {
  var r1 = yield request( "http://some.url.1" );

  // 'yield*'를 통해 '*foo()'에 "위임한다".
  var r3 = yield *foo();

  console.log( r3 );
}

run( bar );
```

---

이 절을 시작하며 보았던 예제와 비교하면, `yield run(foo)` 대신 `yield *foo()` 를 썼다는 점만 다르다.



`yield *`가 내어주는 건 이터레이터 제어권이 아니라 제너레이터 제어권이 아니다. `*foo()` 제너레이터를 시작할 때 이미 이터레이터에 `yield`-위임을 하기 때문이다. `yield`-위임은 실제로 모든 '이터러블'에 가능하다. 가령, `yield *[1,2,3]`은 `[1,2,3]` 배열이 원래 소유한 기본 이터레이터를 쓴다.

### 4.5.1 왜 위임을?

`yield`-위임을 하는 목적은 주로 코드를 조직화하고 그렇게 해서 일반 함수 호출과 맞추기 위함이다.

`foo()`, `bar()`라는 메소드가 있고 `bar()`가 `foo()`를 호출하는 2개의 모듈이 있다고 하자. 이렇게 별개의 함수로 나누는 이유는 일반적으로 개별적인 함수 단위로 프로그램을 호출하는 편이 낫기 때문이다. 예를 들어, `foo()`를 혼자 실행할 수도 있고 `bar()`가 `foo()`를 호출하는 경우의 수도 있다.

이와 정확히 동일한 이유로 제너레이터 역시 분리 배치하는 편이 프로그램 가독성, 유지 보수성, 디버깅 측면에서 유리하다. 이런 점에서 `yield *`는 `*bar()` 내부에서 `*foo()`의 실행 단계를 수동으로 순회할 때 사용하는 단축 구문<sup>syntactic shortcut</sup>이다.

`*foo()`에 비동기적으로 움직이는 단계가 여럿 있을 경우 일일이 수동으로 하려면 엄청나게 복잡하므로 `run(...)` 같은 유틸이 절실해진다. 이미 보았듯이 `yield *foo()` 하면 (`run(foo)` 같은) `run(...)` 유틸의 하위 인스턴스는 더 이상 필요 없다.

### 4.5.2 메시지 위임

놀랍게도 `yield`-위임은 이터레이터뿐 아니라 양방향 메시징에도 쓰인다. 다음

코드에서 yield-위임을 통해 메시지가 오가는 흐름을 잘 따라가보자.

---

```
function *foo() {
  console.log( '*foo()' 내부:", yield "B" );

  console.log( '*foo()' 내부:", yield "C" );

  return "D";
}

function *bar() {
  console.log( '*bar()' 내부:", yield "A" );

  // 'yield'-위임!
  console.log( '*bar()' 내부:", yield *foo() );

  console.log( '*bar()' 내부:", yield "E" );

  return "F";
}

var it = bar();

console.log( "외부:", it.next().value );
// 외부: A

console.log( "외부:", it.next( 1 ).value );
// '*bar()' 내부: 1
// 외부: B

console.log( "외부:", it.next( 2 ).value );
// '*foo()' 내부: 2
// 외부: C

console.log( "외부:", it.next( 3 ).value );
// '*foo()' 내부: 3
// '*bar()' 내부: D
// 외부: E

console.log( "외부:", it.next( 4 ).value );
```

```
// '*bar()' 내부: 4  
// 외부: F
```

---

it.next(3) 호출 이후 처리 단계에 집중하자.

1. 대기 중인 \*foo()의 yield “C” 표현식으로 3이 (\*bar()의 yield-위임을 거쳐) 전해진다.
2. \*foo()에서 return “D” 해도 이 값이 \*foo() 외부로 빠져나와 it.next(3) 호출 후에 그대로 돌아오진 않는다.
3. 대신 “D”는 \*bar()에서 기다리고 있는 yield \*foo() 표현식의 결과값으로 반환된다. 즉, 이 yield-위임 표현식은 모든 \*foo() 실행이 끝날 때까지 마냥 기다린다. 그래서 \*bar() 내부에선 “D”가 출력할 최종값이다.
4. \*bar() 내부에서 yield “E”를 호출하고 “E”는 it.next(3) 호출의 결과값으로 yield되어 나온다.

외부 이터레이터(it) 입장에서는 원래 제너레이터나 위임 제너레이터나 전혀 다른 점이 없다.

yield-위임은 제너레이터가 아닌, 일반 ‘이터러블’에도 다음과 같이 쓸 수 있다.

---

```
function *bar() {  
  console.log( "'*bar()' 내부:", yield "A" );  
  
  // 제너레이터 아닌 객체에 'yield'-위임을 한다!  
  console.log( "'*bar()' 내부:", yield *[ "B", "C", "D" ] );  
  
  console.log( "'*bar()' 내부:", yield "E" );  
  
  return "F";  
}  
  
var it = bar();
```

```

console.log( "외부:", it.next().value );
// 외부: A

console.log( "외부:", it.next( 1 ).value );
// '*bar()' 내부: 1
// 외부: B

console.log( "외부:", it.next( 2 ).value );
// 외부: C

console.log( "외부:", it.next( 3 ).value );
// 외부: D

console.log( "외부:", it.next( 4 ).value );
// '*bar()' 내부: undefined
// 외부: E

console.log( "외부:", it.next( 5 ).value );
// '*bar()' 내부: 5
// 외부: F

```

---

이 예제와 이전 예제에서 메시지를 받고/알리는 위치가 어떻게 다른지 꼭 확인하기 바란다.

가장 주목할 부분은, 기본 배열 이터레이터는 `next( .. )` 호출을 거쳐 보내진 메시지는 전혀 관심이 없어서 2, 3, 4 같은 값들은 그냥 무시한다. 또, 이터레이터는 (앞서 사용했던 `*foo( )`와는 달리) 명시적인 반환값이 없으므로 `yield *` 표현식은 실행이 끝나면 `undefined`를 받는다.

예외도 위임된다!

명쾌하게 양방향 메시징을 수행하는 `yield`-위임은 에러/예외도 같은 식으로 양방향 배달을 해준다.

---

```

function *foo() {
  try {

```

```

        yield "B";
    }
    catch (err) {
        console.log( "'*foo()'에서 붙잡힌 에러:", err );
    }

    yield "C";

    throw "D";
}

function *bar() {
    yield "A";

    try {
        yield *foo();
    }
    catch (err) {
        console.log( "'*bar()'에서 붙잡힌 에러:", err );
    }

    yield "E";

    yield *baz();

    // 아래 코드는 실행되지 않는다!
    yield "G";
}

function *baz() {
    throw "F";
}

var it = bar();

console.log( "외부:", it.next().value );
// 외부: A

console.log( "외부:", it.next( 1 ).value );
// 외부: B

```

```

console.log( "외부:", it.throw( 2 ).value );
// '*foo()'에서 붙잡힌 에러: 2
// 외부: C

console.log( "외부:", it.next( 3 ).value );
// '*bar()'에서 붙잡힌 에러: D
// 외부: E

try {
    console.log( "외부:", it.next( 4 ).value );
}
catch (err) {
    console.log( "외부에서 붙잡힌 에러:", err );
}
// 외부에서 붙잡힌 에러: F

```

---

몇 가지 특기할 부분만 요약한다.

1. `it.throw(2)` 하면 에러 메시지 2를 `*bar()`에 전하고 이 메시지는 다시 `*foo()`로 위임되어 우아하게 에러를 잡아 처리한다. 이후 `yield "C"`는 `it.throw(2)` 호출의 결과값 "C"를 보낸다.
2. 그리고 `*foo() → *bar()` 방향으로 전파되어 던져진 "D"는 `*bar()`가 잡아 역시 우아하게 처리한다. 그런 다음 `yield "E"`는 `it.next(3)` 호출의 결과값 "E"를 보낸다.
3. 다음, `*baz()`에서 발생한 예외는 (외부에서 붙잡았지만) `*bar()`에서 잡히지 않는다. 따라서 `*baz()`, `*bar()` 모두 완료 상태가 된다. 이 코드 밑으로 연달아 `next(..)`를 호출해도 `undefined`를 반환할 뿐 "G" 값을 받을 방법은 없다.



### 4.5.3 비동기성을 위임

다시 한번 다중 순차 AJAX 요청 + yield-위임 예제로 돌아가자.

---

```
function *foo() {
  var r2 = yield request( "http://some.url.2" );
  var r3 = yield request( "http://some.url.3?v=" + r2 );

  return r3;
}

function *bar() {
  var r1 = yield request( "http://some.url.1" );

  var r3 = yield *foo();

  console.log( r3 );
}

run( bar );
```

---

\*bar ( )에서 yield run(foo) 하지 않고 그냥 yield \*foo ( ) 했다.

이전 버전에선 \*foo ( )의 return r3 값을 \*bar ( ) 내부의 지역 변수 r3으로 실어보내기 위해 (run(..)이 제어하는) 프라미스 체계를 이용했었다. 이제 이 값은 \*yield 체계를 거쳐 직접 반환된다.

이 밖의 로직은 거의 동일하다.

### 4.5.4 위임 “재귀”

yield-위임은 위임 단계가 아주 많아도 잘 따라간다. 그래서 비동기형 제너레이터의 재귀, 즉 스스로에게 yield-위임하는 제너레이터를 작성할 때에도 쓸 수 있다.

---

```
function *foo(val) {
  if (val > 1) {
```

```

// 제너레이터 재귀
val = yield *foo( val - 1 );
}

return yield request( "http://some.url?v=" + val );
}

function *bar() {
  var r1 = yield *foo( 3 );
  console.log( r1 );
}

run( bar );

```

---



`run(...)`을 `run( foo, 3 )`으로 호출해도 된다. 제너레이터 초기화 시 전달할 추가 파라미터를 지원하기 때문이다. 하지만 나는 `yield *`의 유연성을 강조하고자 파라미터 없는 `*bar()`를 썼다.

실행 단계별로 살펴보자. 자세히 파고들면 얹히고 설켜있어 복잡할 수 있으니 조금 긴장하자.

1. `run( bar )` 하여 `*bar()` 제너레이터를 시작한다.
2. `foo( 3 )`로 `*foo(...)`의 이터레이터를 생성하고 `val`에 3을 넘긴다.
3. > 1 이므로 `foo( 2 )`는 또 다른 이터레이터를 생성하고 `val`에 2를 넘긴다.
4. > 1 이므로 `foo( 1 )` 역시 이터레이터를 추가로 생성한 후 `val`에 1을 넘긴다.
5. 1 > 1은 `false`이므로 처음 `request(...)`를 `v = 1`으로 AJAX 호출하고 해당 프라미스를 반환한다.
6. 이 프라미스가 `yield`되어 나오면 `*foo( 2 )` 제너레이터 인스턴스로 돌아간다.
7. `yield *`는 이 프라미스를 다시 `*foo( 3 )` 제너레이터 인스턴스로 보낸다.

또 다른 `yield *`는 이 프라미스를 `*bar()` 제너레이터 인스턴스로 보내고, 다시 또 다른 `yield *`는 `run(...)` 유틸로 이 프라미스를 보내기를 반복하여 해당 프라미스(최초의 AJAX 요청)가 귀결되길 기다린다.

8. 프라미스가 귀결되면 이름 메시지가 전송되어 `*bar()`를 재개하고 `yield *`를 통해 `*foo(3)` 인스턴스로 전달한다. 그리고 `yield *`를 통해 `*foo(2)` 제너레이터 인스턴스로 전해지고, 이후에 `yield *`를 통해 `*foo(3)` 제너레이터 인스턴스에서 대기 중인 일반 `yield`문에 이른다.
9. 첫 번째 호출의 AJAX 응답은 이제 바로 `*foo(3)` 제너레이터 인스턴스로부터 넘겨받고 이 값을 다시 `*foo(2)` 인스턴스의 `yield *` 표현식 결과로 전송, 지역 변수 `val`에 할당한다.
10. `*foo(2)` 내부에서 `request(...)`로 두 번째 AJAX 요청이 실행되면 그 프라미스는 `*foo(1)` 인스턴스로 `yield`되어 돌아오고, 그 다음 `yield *`가 `run(...)`까지 내내 달려 전달된다. (다시 단계 7) 프라미스 귀결 시 두 번째 AJAX 응답은 왔던 길을 죽 따라 `*foo(2)` 제너레이터 인스턴스로 돌아가고 지역 변수 `val`에 할당된다.
11. 마지막으로 `request(...)`로 세 번째 AJAX 요청을 하고 프라미스가 `run(...)`으로 간 뒤 해당 귀결값은 다시 되돌아가 반환되어 결국 `*bar()`에서 대기 중인 `yield *` 표현식의 품에 안긴다.

휴! 멘탈 붕괴 직전이라고? 두세 번 반복하여 읽어보고 밖에 나가 간식과 음료로 머리를 식히자!

## 4.6 제너레이터 동시성

1, 4장을 앞부분에서도 말했듯이 동시 실행 중인 두 “프로세스”는 협동적으로 각자의 작업을 인터리빙할 수 있고, 그래서 많은 경우 아주 강력한 비동기 표현식을 구사할 수 있다.

솔직히, 지금까지 보았던 다중 제너레이터의 동시성 인터리빙에 관한 예제들만 봐서는 이해하기 어려운 부분이 있지만 나는 그러한 능력 자체가 힘을 발휘하게 될 상황이 있다고 몇 차례 넌지시 언급했었다.

1장에서 상이한 2개의 동시 AJAX 응답 처리기를 데이터 통신 과정에서 경합 조건이 발생하지 않게끔 잘 조정했던 시나리오가 기억나는가? AJAX 응답을 `res` 배열에 다음과 같이 구분해 넣었다.

---

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

---

다중 제너레이터로 구현하면 어떤 모습일까?

---

// 'request(..)'는 프라미스-인식형 유틸이다.

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

```
function *reqData(url) {
  res.push(
    yield request( url )
  );
}
```

---



여기서 `*reqData(...)` 제너레이터 인스턴스를 2개 사용하지만, 2개의 상이한 제너레이터의 인스턴스 1개를 실행하는 것과 차이는 없다. 어느 쪽으로 하든 로직은 같다. 상이한 제너레이터 2개를 조정하는 문제는 잠시 후 설명한다.

수동으로 `res[0]`, `res[1]` 할당할 필요 없이 `res.push(...)`가 적절히 기대/의도한 순서대로 값을 넣도록 조정한다. 따라서 표현 로직이 조금 더 깔끔해진 느낌이다.

하지만 이런 상호 작용을 어떻게 잘 맞출까? 우선 그냥 프라미스로, 수동으로 해보자.

---

```
var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next();
var p2 = it2.next();

p1
.then( function(data){
    it1.next( data );
    return p2;
} )
.then( function(data){
    it2.next( data );
} );
```

---

두 `*reqData(...)` 인스턴스가 모두 AJAX 요청 후 `yield`에서 멈춘다. 그 후 `p1`이 귀결되면 첫 번째 인스턴스를 재개하고 그 다음 `p2`가 귀결되면 두 번째 인스턴스를 재개한다. 이런 식으로 `res[0]`엔 첫 번째 응답이, `res[1]`엔 두 번째 응답이 확실히 저장되도록 프라미스를 조정한다.

그런데 솔직히 이렇게 하면 손이 많이 가고 실제로 제너레이터 자체를 조정하는 게 아니라서 진가를 발휘했다고 보기 어렵다. 다른 방법을 궁리해보자.

---

// 'request(...)'는 프라미스-인식형 유틸이다.

```

var res = [];

function *reqData(url) {
    var data = yield request( url );

    // 제어권 넘김
    yield;

    res.push( data );
}

var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it.next();
var p2 = it.next();

p1.then( function(data){
    it1.next( data );
} );

p2.then( function(data){
    it2.next( data );
} );

Promise.all( [p1,p2] )
.then( function(){
    it1.next();
    it2.next();
} );

```

---

그렇다, 여전히 수동적인 요소는 있지만 조금 발전된 형태다. 지금은 두 `*reqData( .. )` 인스턴스가 정말 동시에, (적어도 앞부분은) 독립적으로 실행된다.

이전 예제에서는 첫 번째 인스턴스가 완전히 끝난 다음에야 두 번째 인스턴스에 데이터를 넘겼었다. 그러나 이제 두 인스턴스 모두 각자의 응답이 돌아오자마자 데이터를 수신하고 각 인스턴스는 제어권을 넘길 요량으로 한번 더 `yield`한다.

그런 다음, `Promise.all([ ... ])` 처리기에서 어떤 순서로 이들을 재개할지 결정한다.

이러한 방식이 재사용 가능한 유틸 측면에서 더 쉬운 형태인지 확실하진 않다. 더 개선할 수 있을 것 같다. 다음 `runAll(...)`이란 유틸을 한번 생각해보자.

---

// 'request(..)'는 프라미스-인식형 유틸이다.

```
var res = [];  
  
runAll(  
  function*(){  
    var p1 = request( "http://some.url.1" );  
  
    // 제어권 넘김  
    yield;  
  
    res.push( yield p1 );  
  },  
  function*(){  
    var p2 = request( "http://some.url.2" );  
  
    // 제어권 넘김  
    yield;  
  
    res.push( yield p2 );  
  }  
);
```

---



`runAll(...)`의 전체 코드는 지면상 너무 길고 앞에서 구현한 `run(...)`의 확장판이라 다 신지 않았다. 숙제라 생각하고 `runAll(...)`처럼 작동하는 코드를 `run(...)`에서 각자 개량해보기 바란다. 그리고 내가 만든 `asynquence` 라이브러리(부록 A 참고)에도 이와 비슷한 기능을 가진 `runner(...)` 유틸이 있다.

`runAll(...)`의 내부 처리 로직은 다음과 같다.

첫 번째 제너레이터는 “http://some.url.1”에서 첫 번째 AJAX 응답 프라미스를 얻고 다시 `runAll(..)` 유틸로 제어권을 넘겨준다.

두 번째 제너레이터 역시 마찬가지로 “http://some.url.2” 실행 후 `runAll(..)`에 제어권을 도로 갖다준다.

1. 첫 번째 제너레이터 재개 후 자신의 프라미스 `p1`를 `yield`한다. 이 때 `runAll(..)` 유틸이 하는 일은 프라미스 귀결을 기다렸다가 동일한 제너레이터를 재개한다(제어권 넘김은 없다!)는 점에서 앞의 `run(..)`과 다를 바 없다. `p1`이 귀결되면 `runAll(..)`은 첫 번째 제너레이터를 귀결값으로 재개 후 `res[0]`에 해당 값을 할당한다. 첫 번째 제너레이터가 끝나면 암시적으로 제어권 넘김이 이루어진다.
2. 두 번째 제너레이터가 재개되고 `p2` 프라미스를 `yield`한 후 귀결될 때까지 대기한다. 귀결되면 `runAll(..)`은 이 값으로 두 번째 제너레이터를 재개 하고 `res[1]`을 세팅한다.

예제에선 `res`라는 외부 변수를 상이한 두 AJAX 응답 결과를 담아둘 용기로 사용했다. 동시성 조정이 있기에 가능한 일이다.

`runAll(..)`을 확장하여 다중 제너레이터 인스턴스가 공유할 수 있는 내부 변수 공간(다음 예제의 `data`라는 빈 객체)을 제공하면 훨씬 유용한 유틸이 될 것이다. 이 변수는 프라미스 아닌 값을 받아 `yield`해서 다음 제너레이터로 넘길 수 있다.

다음 예제를 보자.

---

// 'request(..)'는 프라미스-인식형 유틸이다.

```
runAll(  
  function*(data){  
    data.res = [];
```



```

// 제어권(그리고 메시지) 전달
var url1 = yield "http://some.url.2";

var p1 = request( url1 ); // "http://some.url.1"

// 제어권 넘김
yield;

data.res.push( yield p1 );
},
function*(data){
// 제어권(그리고 메시지) 전달
var url2 = yield "http://some.url.1";

var p2 = request( url2 ); // "http://some.url.2"

// 제어권 넘김
yield;

data.res.push( yield p2 );

});

```

---

이렇게 고치니 두 제너레이터가 단순한 제어권 조정뿐 아니라 data.res, yield 된 메시지를 통해 url1, url2 두 값을 교류하면서 실질적인 통신을 서로 하게 된다. 실로 막강한 기능이다!

이 로직은 부록 B에서 다룰, 순차적 프로세스 통신<sup>CSP, Communicating Sequential Processes</sup>이라는 더 정교한 비동기 기법의 개념적 근거 역할을 한다.

## 4.7 씹크

나는 제너레이터에서 프라미스를 yield하고 run(...) 같은 헬퍼를 통해 이 프라미스가 제너레이터를 다시 시작하게 만드는 방법이 제너레이터로 비동기성을 관

리하는 최선이라는 전제 하에 설명을 이어왔다.

그런데 꽤 많은 사람들이 알고 있는 패턴이 또 하나 있다. 완벽을 추구하는 여러분을 위해 이 패턴까지 간단히 살펴보겠다.

일반 컴퓨터 과학에 썻크<sup>think</sup>라는, 자바스크립트 이전에 등장한 개념이 있다. 역사적 유래까지 굳이 열거할 필요는 없을 것 같고 한정된 의미에서 썻크를 표현하자면, 다른 함수를 호출할 운명을 가진, 파라미터 없는 함수다.

다시 말하면, 어떤 함수 정의부를 (필요한 파라미터도 같이) 또 다른 함수 호출부로 감싸 실행을 지연시키는 것이고, 여기서 감싼 함수가 바로 썻크다. 따라서 나중에 썻크를 실행하면 결국 원래 함수를 호출하는 것이나 다를 바 없다.

예를 들면,

---

```
function foo(x,y) {  
    return x + y;  
}  
  
function fooThink() {  
    return foo( 3, 4 );  
}  
  
// 나중에  
  
console.log( fooThink() ); // 7
```

---

보다시피 동기적 썻크는 상당히 직관적이다. 그렇다면 비동기 썻크는 어떨까? 콜백을 수신하는 기능까지 포함하여 한정된 썻크의 범위를 확장하면 된다.

다음 코드를 보자.

---

```
function foo(x,y,cb) {  
    setTimeout( function(){
```

```

        ( x + y );
    , 1000 );
}

function fooThunk(cb) {
    foo( 3, 4, cb );
}

// 나중에

fooThunk( function(sum){
    console.log( sum ); // 7
} );

```

---

3, 4(각각 x, y 값) 값이 이미 지정되어 foo( ..)에 전달할 채비를 마쳤으므로 fooThunk( ..)는 cb( ..) 파라미터에 해당하는 함수만 넣어주면 된다. 썩크는 그냥 끈기있게 자신의 임무 완수에 필요한 마지막 퍼즐 조각, 즉 콜백을 기다린다. 그러나 일일이 수동으로 썩크를 코딩하고 싶지는 않을 것이다. 포장 업무를 대행할 유틸을 작성하자.

---

```

function thunkify(fn) {
    ar args = [].slice.call( arguments, 1 );
    eturn function(cb) {
        args( cb );
        return fn.apply( null, args );
    };
}

var fooThunk = thunkify( foo, 3, 4 );

// 나중에

fooThunk( function(sum) {
    console.log( sum ); // 7
} );

```

---



여기서 원본 함수(foo(...))의 시그니처<sup>signature</sup> 가장 마지막 위치에 콜백이, 다른 파라미터는 그 앞에 있다고 가정한다. 보편적으로 통용되는, 비동기 자바스크립트 함수에 관한 일종의 표준이다. 이름을 붙인다면 “콜백-나중 스타일<sup>callback-last style</sup>” 정도다. 어떤 이유에서건 “콜백-우선 스타일<sup>callback-first style</sup>” 시그니처로 처리할 경우는 args.push(...) 대신 args.unshift(...)를 써서 유틸을 하나 더 만들면 된다.

예제에서 thunkify(...)는 foo(...) 함수의 레퍼런스와 필요한 파라미터를 입력받고 썩크 자신(fooThunk(...))을 도로 반환한다. 하지만 자바스크립트에서 썩크를 이용하는 표준적인 방법은 아니다.

보통 thunkify(...)로 썩크 자신을 생성하기보단 (너무 복잡하지 않다면) 썩크를 만드는 함수를 생성한다.

무슨 소린지...?

다음 코드를 보자.

---

```
function thunkify(fn) {
  return function() {
    var args = [].slice.call( arguments );
    return function(cb) {
      args.push( cb );
      return fn.apply( null, args );
    };
  };
}
```

---

가장 두드러진 차이점은 return function() { .. } 부분이고 사용하는 방법도 다 음과 같이 달라진다.

---

```
var whatIsThis = thunkify( foo );

var fooThunk = whatIsThis( 3, 4 );
```

```
// 나중에
fooThunk( function(sum) {
    console.log( sum ); // 7
} );
```

---

whatIsThis 프로퍼티를 뭐라고 부르는 게 좋을지가 제일 난감하다. 썩크가 아닌, 썩크를 만드는 것으로, “썩크”를 생산하는 “공장” 같은 존재다. 이런 객체를 명명하기 위한 표준화된 체계는 아직 없는 것 같다.

그래서 내 생각에 “썩커리<sup>thunkory</sup>” (“썩크<sup>thunk</sup>” + “팩토리<sup>factory</sup>”)가 어떨까 싶다. `thunkify(..)`는 썩커리를 만들고 다시 썩커리는 썩크를 만든다. 이미 내가 3장에서 제안했던 “프라미서리”와 잘 매치된다.

---

```
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );
```

```
// 나중에

fooThunk1( function(sum) {
    console.log( sum ); // 7
} );

fooThunk2( function(sum) {
    console.log( sum ); // 11
} );
```

---



`foo(...)` 실행 예제의 콜백 유형은 “에러-우선 스타일”이 아니다. 물론, “에러-우선 스타일”을 훨씬 더 많이 쓴다. `foo(...)`에서 어떤 문법 에러가 날 것 같으면 에러-우선 콜백으로 얼마든지 변경할 수도 있다. 어떤 콜백 스타일을 전제로 하든지 이후의 `thunkify(...)` 체계는 아무래도 좋다. 다만 사용법이 `fooThunk1(function(err,sum){...}` 식으로 달라진다.

썸커리 메소드를 노출하는 건 (앞서 `thinkify(...)`가 중간 단계를 숨겼던 것에 비하면) 쓸데없이 복잡할 수도 있다. 하지만 일반적으로 기존 API 메소드를 감싸기 위해 썸커리를 프로그램 앞부분에 위치하면 썸크가 필요한 시점에 이 썸커리를 호출/전달할 수 있어서 제법 유용하다. 두 단계를 분명히 구분해서 더 깔끔하게 기능 분리 가능하다.

예를 들면,

---

```
// 이렇게 하는 게 더 깔끔하다.
```

```
var fooThinkory = thinkify( foo );
```

```
var fooThink1 = fooThinkory( 3, 4 );
```

```
var fooThink2 = fooThinkory( 5, 6 );
```

```
// 다음 코드와 비교해보자.
```

```
var fooThink1 = thinkify( foo, 3, 4 );
```

```
var fooThink2 = thinkify( foo, 5, 6 );
```

---

썸커리를 어떻게 취급하든 두 썸크 `fooThink1(...)`, `fooThink2(...)`의 사용법은 변함없다.

## 4.7.1 s/promise/thunk/

그런데 이런 얘기들이 제너레이터와 무슨 상관일까?

썸크와 프라미스는 작동 개념이 동등하지 않아 직접적인 상호 호환성은 없다. 있는 그대로의 썸크와 비교하면 프라미스가 훨씬 더 능력 좋고 미덥다.

그러나 또 다른 의미에서 둘 다 어떤 값을 요청하여 비동기적 응답을 받는 점은 동일하다.

3장에서 함수를 프라미스화<sup>promisify</sup>하는, `Promise.wrap(...)`라는 유틸을 소개했었는데 이 역시 `promisify(...)`라고 할 수 있다! 이 프라미스 감싸미 유틸은 직

접 프라미스를 생성하는 대신, 나중에 프라미스를 생성할 프라미서리를 만들어낸다. 좀 전의 썬커리 - 썬크의 관계와 정확히 같다.

이전의 `foo(...)` 예제를 “에러-우선 스타일” 콜백을 가정하여 약간 바꿔보자.

---

```
function foo(x,y,cb) {
  setTimeout( function(){
    // 'cb(...)'는 "에러-우선 스타일"이다.
    cb( null, x + y );
  }, 1000 );
}
```

---

자, `thunkify(...)`와 `promisify(...)` (3장의 `Promise.wrap(...)`)의 사용법을 비교해보라.

---

```
// 대칭적이다: 질의자(question asker)를 생성
var fooThunkory = thunkify( foo );
var fooPromisory = promisify( foo );
```

```
// 대칭적이다: 질문(question)을 던진다.
var fooThunk = fooThunkory( 3, 4 );
var fooPromise = fooPromisory( 3, 4 );
```

```
// 썬크 답변을 받는다.
fooThunk( function(err,sum){
  if (err) {
    console.error( err );
  }
  else {
    console.log( sum ); // 7
  }
} );
```

```
// 프라미스 답변을 받는다.
fooPromise
.then(
  function(sum){
```

```

    console.log( sum ); // 7
  },
  function(err){
    console.error( err );
  }
);

```

---

썹커리, 프라미서리 모두 기본적으로 (어떤 값을) 질의하고, fooThunk 썹크, fooPromise 프라미스는 이 질문에 대해 각각 미래값을 나타낸다. 이 정도만으로도 명백한 대칭 관계다.

이런 관점으로 보면 제너레이터는 비동기성 프라미스 대신, 비동기성 썹크를 yield해도 된다. run( .. ) 유틸을 (앞에서도 그랬듯이) 더 똑똑하게 고쳐주면 yield된 프라미스 + yield된 썹크에도 콜백을 걸 수 있다.

```

function *foo() {
  var val = yield request( "http://some.url.1" );
  console.log( val );
}

run( foo );

```

---

이렇게하면 여기서 request( .. )는 프라미스를 반환하는 프라미서리, 또는 썹크를 반환하는 썹커리일 수 있다. 제너레이터 내부 코드 입장에서 어떻게 구현됐는지 신경 쓸 필요가 없다. 정말 막강하지 않은가?

request( .. )는 둘 중 하나다.

```

// 프라미서리 'request(..)' (3장 참고)
var request = Promise.wrap( ajax );

// vs.

// 썹커리 'request(..)'

```



```
var request = thunkify( ajax );
```

---

결국, `run (..)` 유틸이 썩크를 인지할 수 있게 패치를 하자면 다음 로직을 추가해야 한다.

---

```
// ..
// 썩크를 돌려받았는가?
else if (typeof next.value == "function") {
  return new Promise( function(resolve,reject){
    // 에러-우선 콜백으로 썩크를 부른다.
    next.value( function(err,msg) {
      if (err) {
        reject( err );
      }
      else {
        resolve( msg );
      }
    } );
  } )
  .then(
    handleNext,
    function handleErr(err) {
      return Promise.resolve(
        it.throw( err )
      )
      .then( handleResult );
    }
  );
}
```

---

이제 우리가 만든 제너레이터는 프라미서리를 호출하여 프라미스를 `yield`하거나 썩커리를 호출해서 썩크를 `yield`할 테고, 어느 쪽이든 `run (..)`은 이 값을 받아 기다렸다가 완료 시 제너레이터를 재개하는 용도로 사용할 것이다.

대칭 관계라 두 접근 방식은 일치하는 것처럼 보인다. 하지만 제너레이터를 진행할

미래값을 나타내는 프라미스, 썬크의 관점에서만 그렇다는 사실을 강조하고 싶다. 좀 더 넓게 보면 썬크 자체는 프라미스의 믿음성/조합성은 거의 보장하지 못한다. 썬크를 특정한 제너레이터의 비동기 패턴에서 프라미스 대응으로 쓸 수는 있겠지만 프라미스가 제공하는 제반 혜택(3장 참고)을 생각하면 이상적인 해결책은 아니다.

선택할 수 있다면 `yield th` 보단 `yield pr` 쪽을 쓰자. `run(...)` 유틸은 어떤 종류의 값이든 다 다룰 수 있으니 문제될 것이 없다.



`asynquence` 라이브러리(부록 A 참고)의 `runner(...)` 유틸은 프라미스, 썬크, `asynquence` 시퀀스의 `yield`까지 모두 처리한다.

## 4.8 ES6 이전 제너레이터

이제는 여러분도 제너레이터가 비동기 프로그래밍 공구함에서 얼마나 큰 비중을 차지하는 연장인지 충분히 깨달았을 것이다. 그러나 제너레이터는 ES6 이후에 나온 신생 구문이므로 (신생 API인) 프라미스처럼 단순 폴리필은 불가능하다. ES6 이전 브라우저를 완전히 배제하는 사치를 누리기 어려운 상황에서 어떻게든 제너레이터를 브라우저에서 쓸 방법은 없을까?

다행히 ES6에서 새로 확장된 구문 중 트랜스파일러(`transpiler`(트랜스`trans` + 컴파일러`compiler`)라는 도구 덕분에 어떤 ES6 구문을 (물론 겉모습은 흉하지만) 이에 상응하는 ES6 이전 코드로 변환할 수 있다. 제너레이터 코드 역시 ES5 이하 환경에서 똑같이 작동하게끔 트랜스파일이 가능하다.

그런데 방법은? 음, `yield` “마술”은 확실히 트랜스파일이 만만찮아 보인다. 눈치 빠른 독자라면 앞서 클로저식 ‘이터레이터’를 언급할 때 내가 던지시 해법을 제시했던 걸 기억할 것이다.

## 4.8.1 수동 변환

먼저 제너레이터를 수동으로 트랜스파일하는 방법을 보자. 실제로 직접 해보아야 나중에 산 지식이 될 테니 숙제라고 불평하지 않기 바란다.

---

// 'request(..)'는 프라미스-인식형 유틸이다.

```
function *foo(url) {
  try {
    console.log( "요청 중:", url );
    var val = yield request( url );
    console.log( val );
  }
  catch (err) {
    console.log( "에러:", err );
    return false;
  }
}

var it = foo( "http://some.url.1" );
```

---

일단 호출 가능한 일반 함수 `foo()`가 필요하고 이 함수는 이터레이터를 반환해야 한다는 걸 알 수 있다. 자, 제너레이터 아닌 객체를 제너레이터로 탈바꿈하는 코드의 윤곽을 잡아보자.

---

```
function foo(url) {

  // ..

  // 이터레이터를 만들어 반환한다.
  return {
    next: function(v) {
      // ..
    },
    throw: function(e) {
      // ..
    }
  }
}
```

```

    };
}

var it = foo( "http://some.url.1" );

```

---

제너레이터는 자신의 스코프/상태를 지연시키는 “마술사”인데 함수 클로저로도 비슷하게 흉내낼 수 있다. (본 시리즈 '[스코프와 클로저](#)' 참고) 방법을 설명하기 위해 먼저 상태값을 기준으로 제너레이터 코드를 각각 나누어 살펴보자.

---

// 'request(..)'는 프라미스-인식형 유틸이다.

```

function *foo(url) {

    // 상태 1
    try {
        console.log( "요청 중:", url );
        var TMP1 = request( url );

        // 상태 2
        var val = yield TMP1;
        console.log( val );
    }
    catch (err) {
        // 상태 3
        console.log( "에러:", err );
        return false;
    }
}

```

---



더 정확히 설명하려고 `val = yield request..` 문을 임시 변수 TMP1으로 이등 분했다. `request(..)`는 상태 1에서 실행되고 상태 2에서 그 완료값이 `val`에 할당 된다. 중간 단계를 나타내는 TMP1은 나중에 제너레이터 아닌 코드로 변환할 때 제거될 예정이다.

이 코드는 상태 1에서 시작하여 `request(..)`가 성공하면 상태 2, 실패하면 상

태 3으로 넘어간다. 이런 식으로 상태를 추가해서 yield 단계를 얼마든지 늘릴 수 있다.

자, 다시 트랜스파일된 제너레이터로 돌아가자. 먼저 클로저에서 상태 추적용으로 사용할 변수 state를 정의한다.

---

```
function foo(url) {  
    // 제너레이터 상태를 관리  
    var state;  
  
    // ..  
}
```

---

이제 클로저 내부에 switch문을 써서 상태에 따라 처리하는, 내부 함수 process(..)를 정의한다.

---

// 'request(..)'는 프라미스-인식형 유틸이다.

```
function foo(url) {  
    // 제너레이터 상태를 관리  
    var state;  
  
    // 제너레이터 스코프 변수 선언  
    var val;  
  
    function process(v) {  
        switch (state) {  
            case 1:  
                console.log( "요청 중:", url );  
                return request( url );  
            case 2:  
                val = v;  
                console.log( val );  
                return;  
            case 3:  
                var err = v;  
                console.log( "에러:", err );
```

```

        return false;
    }
}

// ..
}

```

---

제너레이터의 모든 상태는 이 switch문의 case로 나타내고 상태가 바뀔 때마다 `process (..)`를 호출한다. 작동 로직은 좀 있다가 다시 설명한다.

`process (..)`를 여러 번 호출해도 제너레이터 스코프의 변수(`val`)값이 유지되도록 `process (..)` 바깥쪽 `var` 선언부 위치로 옮긴다. 하지만 블록 스코프 변수 `err`는 상태 3에서만 필요하므로 그냥 놔둔다.

상태 1에서 `yield resolve (..)` 대신 `return resolve (..)` 했다. 상태 2 끝에 명시적 `return`이 없으므로 그냥 `return`한다. (`return undefined` 하는 것과 같다) 상태 3 끝에는 `return false`가 있으니 그대로 놔둔다.

이제 이터레이터 함수를 코딩하여 `process (..)`를 올바르게 호출해보자.

---

```

function foo(url) {
    // 제너레이터 상태를 관리
    var state;

    // 제너레이터 스코프 변수 선언
    var val;

    function process(v) {
        switch (state) {
            case 1:
                console.log( "요청 중:", url );
                return request( url );
            case 2:
                val = v;
                console.log( val );
                return;

```

```

        case 3:
            var err = v;
            console.log( "에러:", err );
            return false;
        }
    }

    // 이터레이터를 만들어 반환한다.
    return {
        next: function(v) {
            // 초기 상태
            if (!state) {
                state = 1;
                return {
                    done: false,
                    value: process()
                };
            }
            // yield 재개가 성공
            else if (state == 1) {
                state = 2;
                return {
                    done: true,
                    value: process( v )
                };
            }
            // 제너레이터는 이미 완료
            else {
                return {
                    done: true,
                    value: undefined
                };
            }
        },
        "throw": function(e) {
            // 명시적인 에러 처리는 상태 1에만 해당된다.
            if (state == 1) {
                state = 3;
                return {
                    done: true,
                    value: process( e )
                };
            }
        }
    };

```

```

        };
    }
    // 이밖의 예러는 처리되지 않으니
    // 그냥 곧바로 되던진다.
    else {
        throw e;
    }
}
};
}

```

---

프로그램 실행 흐름을 살펴보자.

1. 이터레이터 `next()`를 처음 호출하면 제너레이터는 초기화되지 않은 상태 → 상태 1로 바뀌고 `process()`를 호출하여 상태 1을 처리한다. `request(..)`의 반환값, 즉 AJAX 응답에 해당하는 프라미스는 `next()` 호출의 `value` 프로퍼티로 돌려준다.
2. AJAX 요청이 성공하면 `next()`를 두 번째 호출하여 AJAX 응답값을 보내고 상태 2가 된다. AJAX 응답값을 파라미터로 `process(..)`를 재호출하고 `next(..)` 호출의 `value` 프로퍼티는 `undefined`가 될 것이다.
3. 반면, AJAX 요청이 실패하면 에러 객체와 함께 `throw(..)`가 호출되고 상태 1 → (상태 2 대신) 상태 3으로 변경된다. 이번에는 에러값을 파라미터로 `process(..)` 한다. 이 케이스의 반환값 `false`는 `throw(..)` 호출의 `value` 프로퍼티 값이 된다.

보다시피 외부에서 오직 이터레이터만 갖고 일반 함수 `foo(..)`가 `*foo(..)` 제너레이터와 진배없이 거의 똑같이 작동한다. ES6 제너레이터로 하여금 ES6-이전 호환성을 갖게 효과적으로 트랜스파일한 셈이다.

수동으로(`var it = foo(".."); it.next(..)` 등으로) 제너레이터를 인스턴스화하



고 이터레이터를 제어할 수도 있다. `run(foo, "...")`처럼 이미 앞에서 정의했던 `run(...)` 유틸에 넘겨주면 더 좋다.

## 4.8.2 자동 변환

ES6 제너레이터를 ES6-이전과 호환되는 제너레이터로 수동 변환하는 과정을 실습하면서 제너레이터가 개념적으로 어떻게 작동하는지 엿볼 수 있었다. 그런데 사실 변환 과정이 아주 복잡할 뿐더러 같은 코드의 다른 제너레이터에 이식성도 상당히 떨어진다. 무엇보다 일일이 수동 변환한다는 건 비현실적이고 또 제너레이터의 모든 혜택들이 무용지물이 되어버린다.

하지만 다행히도 앞에서 도출했던 동일한 로직으로 ES6 제너레이터를 자동 변환하는 툴이 이미 나와있다. 엄청 부담스러운 작업을 대행할뿐만 아니라 내가 대략 지나갔던 몇몇 세부분까지 깔끔하게 처리해준다.

그 중 똑똑한 페이스북 개발자들이 만든 리제너레이터<sup>regenerator</sup>(<https://facebook.github.io/regenerator>)라는 툴이 있다.

다음은 (이 글을 쓰는 현재) 리제너레이터로 제너레이터를 트랜스파일한 코드다.

---

```
// 'request(..)'는 프라미스-인식형 유틸이다.
```

```
var foo = regeneratorRuntime.mark(function foo(url) {
  var val;

  return regeneratorRuntime.wrap(function foo$($context$1$0) {
    while (1) switch ($context$1$0.prev = $context$1$0.next) {
      case 0:
        $context$1$0.prev = 0;
        console.log( "요청 중:", url );
        $context$1$0.next = 4;
        return request( url );
      case 4:
        val = $context$1$0.sent;
```

```

        console.log( val );
        context$1$0.next = 12;
        break;
    case 8:
        context$1$0.prev = 8;
        context$1$0.t0 = context$1$0.catch(0);
        console.log("에러:", context$1$0.t0);
        return context$1$0.abrupt("return", false);
    case 12:
    case "end":
        return context$1$0.stop();
    }
}, foo, this, [[0, 8]]);
});

```

---

switch/case 문을 쓴 것이나 콜로저에서 val 변수를 도출한 로직까지 수동 변환과 흡사하다.

한 가지 흠이라면, 리제너레이터로 트랜스파일하려면 일반적인 제너레이터/이터레이터 관리에 필요한 재사용 가능 로직을 담고 있는, `regeneratorRuntime` 헬퍼가 필수라는 점이다. 대다수 관용 코드가 내 버전과는 겉모습이 다르지만, 그래도 `context$1$0.next = 4`에서 보드시피 다음 제너레이터 상태를 추적하는 등 개념은 그대로다.

요점은 제너레이터가 ES6+ 환경에서만 쓸모있는 장치는 아니라는 사실이다. 여러분도 개념을 알고 나면 작성 중인 코드에 얼마든지 널리 적용할 수 있고 구 버전 환경과의 호환성이 문제라면 툴을 써서 코드를 변환하면 된다.

프라미스 API 폴리필을 갖다쓰는 것보다 해야될 일이 약간 더 많지만 노력의 대가는 충분하다. 추론/실용적이면서 동기/순차적 형태로 비동기 흐름 제어를 나타내는데 제너레이터만한 도구는 없다.

한번 제너레이터의 매력에 꽂히고 나면 다시는 비동기 콜백의 스파게티 코드로 돌

아갈 생각은 꿈도 꾸지 못할 것이다.

## 4.9 정리하기

제너레이터는 ES6부터 도입된 새로운 유형의 함수로서, 일반 함수처럼 완전-실행하지 않고 실행 도중 (상태 정보를 그대로 간직한 채) 멈출 수도 있고 멈춘 지점에서 나중에 다시 시작할 수도 있다.

멈춤/재개가 번갈아 일어나므로 제너레이터는 선점적이라기 보다 협동적인 툴이다. `yield` 키워드를 이용하여 스스로 멈출 수 있고 이 제너레이터를 제어하는 이터레이터는 `next(...)`를 호출하여 제너레이터를 다시 시작할 수 있다.

`yield / next(...)` 이중성은 제어 장치뿐만 아니라 양방향 메시징 체계로도 실질적인 활용이 가능하다. `yield ..` 표현식은 일단 멈추고 어떤 값을 기다리게 하고 `next(...)` 호출은 이렇게 멈춘 `yield` 표현식에 값(아니면 암시적으로 `undefined`)을 전해준다.

비동기 흐름 제어와 연관된 제너레이터의 핵심은 제너레이터 내부 코드가 동기/순차적 형태로 일련의 작업 단계를 자연스럽게 표현할 수 있는 능력이다. 그 비결은 바로 `yield` 키워드 뒷편에 숨겨진 잠재적인 비동기성이다. 즉, 제너레이터의 ‘이터레이터’가 제어하는 코드로 비동기성을 옮겨놓은 것이다.

결론적으로, 제너레이터는 비동기 코드의 순차/동기/중단적 패턴을 유지함으로써 개발자들이 훨씬 더 코드를 자연스럽게 이해할 수 있게 하고 콜백식 비동기 코드의 치명적인 단점 두 가지를 해결한 일등 공신이다.

# 프로그램 성능

지금까지 비동기 패턴을 조금이라도 더 효과적으로 활용할 수 있는 방안을 이야기해왔는데, 그렇다면 자바스크립트에서 비동기성이 그렇게 중요한 이유는 뭘까? 바로 ‘성능<sup>performance</sup>’ 때문이다.

가령, 2개의 독립적인 AJAX 요청이 다 끝나야만 다음 단계로 진행하는 코드가 있을 때, 이 둘의 상호 작용은 ‘순차<sup>serial</sup>’, ‘동시<sup>concurrent</sup>’ 두 가지 방법으로 모델링할 수 있다.

즉, 첫 번째 요청을 하고 완료되면 두 번째 요청을 시작하거나(순차), 프라미스/제너레이터처럼 두 요청을 병렬 전송한 뒤 둘 다 관문을 통과할 때까지 잠시 대기 후 진행하는(동시) 두 가지다.

두말 할 것 없이 후자의 성능이 전자에 비해 훨씬 우수하다. 사용자 경험<sup>user experience</sup> 측면에서도 더 낫다.

설사 프로그램 전체 실행 시간이 같다 하더라도 비동기성(인터리빙된 동시성)은 체감 성능을 높여준다. 모든 경우를 통틀어 사용자가 피부로 느끼는 성능이야말로 실제 측정 가능한 성능만큼 (그 이상은 아니겠지만) 중요하다.

5장에서는 로컬 영역의 비동기 패턴보다 넓은 시야에서 프로그램 수준의 성능을 고찰한다.



a++, ++a 중 어느 쪽이 더 빠를까, 하는 미시성능microperformance 문제는 6장에서 다룬다.

## 5.1 웹 워커

(브라우저/UI를 느려지게 할 만한) 처리 집약적processing-intensive 작업을 메인 스레드에서 실행하고 싶지 않은 상황에서 자바스크립트가 알아서 멀티스레드로 돌아갔으면 하는 생각, 누구나 한번쯤 해봤을 것이다.

1장에서 자바스크립트는 단일-스레드로만 돌아간다고 설명했고 여전히 그 말은 진리다. 하지만 단일-스레드가 프로그램 실행을 구조화할 수 있는 유일한 방법은 아니다.

여러분이 짠 프로그램을 두 조각으로 나누어 한 쪽은 메인 UI 스레드에서, 다른 한 쪽은 전혀 별개의 스레드에서 실행한다고 하자.

이런 아키텍처로 프로그램 개발을 한다면 어떤 이슈가 있을까?

우선, 별개의 스레드에서 실행한다는 말이 두 번째 스레드 상에서 처리 시간이 긴 프로세스가 메인 프로그램 스레드를 중단시키는 일이 없도록 (다중 CPU/코어 시스템에서) 병렬 실행한다는 의미일까? 만약 그렇지 않다면 “가상 스레딩virtual threading”은 앞 장에서 자바스크립트로 비동기 동시성을 구현한 것보다 그다지 나을 게 없다.

그리고 두 프로그램 조각이 동일한 공유 스코프/자원에 접근할 수 있을까? 만약 그렇다면 (자바, C++ 등의) 여타 멀티스레드 프로그래밍 언어에서 협동적cooperative / 선점적preemptive 잠금locking (뮤텍스mutexes 등) 등을 다루면서 수반되는 갖가지 이슈들을 떠안고 가야 하는데 추가적인 부담이 만만치 않아 가벼이 볼 일은 아니다.

그럼 두 프로그램 조각이 스코프/자원을 공유할 수 없으면 서로 어떻게 통신할 수

있을까?

HTML5가 태동할 무렵부터 이와 같은 의미있는 질문들이 웹 워커Web Worker라는 웹 플랫폼 특성으로 결실을 맺는다. 하지만 자바스크립트 언어 자체와는 관계 없는, 브라우저(호스트 환경) 특성인데다 사실 자바스크립트는 ‘현재까지’ 스레드 실행을 지원하지 않는다.

브라우저 같은 환경은 다수의 자바스크립트 엔진 인스턴스를 쉽게 내어줄 수 있고 인스턴스마다 개별 스레드를 배정하여 실행할 수도 있다. 이러한 프로그램의 독립적인 스레드 조각을 (웹) 워커Worker라고 하며, 프로그램을 덩이로 나누어 병렬 실행하는 ‘작업 병행성task parallelism’을 추구한다.

자바스크립트 메인 프로그램(또는 다른 워커)에서 워커는 다음과 같이 인스턴스화한다.

---

```
var w1 = new Worker( "http://some.url.1/mycoolworker.js" );
```

---

워커로 읽어들이 자바스크립트 파일(HTML 페이지 말고)의 URL을 지정하면 브라우저는 이 파일을 별도의 스레드에서 독립적인 프로그램으로 실행한다.



이렇게 URL로 생성한 워커를 전용 워커dedicated worker라 부른다. 하나의 (이진) 값에 저장한 인라인 파일이므로 외부 파일 URL 대신 (또 다른 HTML5 특성인) Blob URL<sup>16</sup>로 인라인 워커inline worker를 생성하는 방법도 있다. Blob은 이 책의 주제가 아니므로 생략한다.

워커는 같은 워커끼리, 심지어는 메인 프로그램과도 스코프/자원 공유를 안 하지만, (그래서 자칫 스레드 프로그래밍의 판도라 상자가 열릴 뻔 했다) 기본적인 이벤트 메시지 체계를 바탕으로 서로 연결한다.

---

**01** 역자주\_Blob은 한 마디로 데이터 시퀀스의 바이트 덩어리를 가리키는 레퍼런스로 HTML5에 추가된 객체입니다. Blob URL은 일반적인 파일을 브라우저에서 file://... 식으로 참조하는 것처럼 blob://... 형태로 Blob 객체를 가리키는 URL입니다. 자세한 내용은 HTML5 참고 서적 및 [MDN 웹사이트](#)를 참고하시기 바랍니다.

워커 객체 w1은 이벤트 리스너 + 트리거로, 워커가 보낸 이벤트를 구독하고 워커에 이벤트를 보낸다.

이벤트(사실, 고정된 “메시지” 이벤트) 리스닝 코드를 보자.

---

```
w1.addEventListener( "메시지", function(evt){  
    // evt.data  
} );
```

---

워커로 “메시지” 이벤트를 보낼 때에는,

---

```
w1.postMessage( "아주 재미난 얘기" );
```

---

워커 내부 메시징 구조 역시 완전히 대칭적<sup>symmetrical</sup>이다.

---

```
// "mycoolworker.js"  
  
addEventListener( "메시지", function(evt){  
    // evt.data  
} );  
  
postMessage( "정말 쿨한 대답" );
```

---

전용 워커는 자신을 만든 프로그램과 1:1 관계라는 사실을 기억하자. 따라서, “메시지” 이벤트는 오직 이 (워커 또는 메인 페이지에서 온) 1:1 관계에서 비롯되었으니 굳이 구별할 이유가 없다.

대개 워커는 메인 페이지 애플리케이션에서 만들지만, 필요 시 자신의 자식 워커(서브워커<sup>subworker</sup>)(들)를 인스턴스화할 수 있다. 세부 로직은 이른바 “마스터<sup>master</sup>” 워커에게 넘겨서 작업 전체를 단계별 처리할 다른 워커들을 생성하도록 위임하는 편이 유용한 경우가 있다. 아쉽게도 이 글을 집필하는 현재 크롬에서 (파이어폭스와는

달리) 서버워커는 지원되지 않는다.

워커를 낳은 프로그램은 워커 객체를 (이전 예제 w1 처럼) `terminate()` 해서 곧바로 제거한다. 갑자기 워커 스레드를 종료하면 미처 작업을 완료하거나 자원 정리할 기회가 없다. 마치 페이지를 죽이려고 브라우저 탭을 닫는 격이다.

브라우저에서 다수의 페이지가 동일한 파일 URL로부터 워커를 생성하려고 하면 (또는 같은 페이지에 탭이 여러 개 있으면) 각 페이지는 완전히 별개의 워커로 움직인다. 워커를 공유하는 방법은 잠시 후 설명한다.



이렇게 말하면 악의적으로 은밀하게 잠입한 자바스크립트 프로그램이 워커를 수백개 찍어내고 각자 스레드를 돌려 손쉽게 서비스 거부 공격(denial-of-service attack)을 시도할 수 있을 것 같다. 워커는 반드시 개별 스레드가 되지만 100% 완벽히 장담할 순 없다. 실제로 몇 개의 스레드/CPU/코어를 생성할지는 시스템이 결정할 몫이다. 적어도 가용한 한도 내에서 CPU/코어를 최대한 사용하지 않겠냐고 추측하는 사람들이 많지만 그 숫자가 정확히 어느 정도일지 예측/단언할 방법은 없다. 가장 방어적으로 말한다면 메인 UI 스레드 빼고 적어도 하나의 스레드가 더 있다고 보면 될 듯싶다.

### 5.1.1 워커 환경

워커 내부에서는 메인 프로그램의 자원에 접근할 수 없다. 전역 변수는 물론이고 페이지 DOM 등 여타 자원도 접근 불허다. 전혀 다른 스레드니 당연하다.

하지만 워커는 네트워크 작업(AJAX, 웹소켓), 타이머 설정이 가능하며, `navigator`, `location`, `JSON`, `applicationCache` 등 중요한 전역 변수/특성을 자체 복사하여 접근할 수 있다.

워커에 추가 자바스크립트를 읽어들이려면 `importScripts(...)`를 쓴다.

---

```
// 워커 내부에서
importScripts( "foo.js", "bar.js" );
```

---



스크립트는 동기적으로 읽기 때문에 `importScripts(...)`를 호출하면 해당 파일(들)을 완전히 읽고 실행할 때까지 나머지 워커 코드는 실행 중지된다.



워커에서 `<canvas>` API를 쓸 수 있게 하는 문제는 현재 논의 중이다. 캔버스를 트랜스퍼러블(`Transferables`)과 결합하여(259 페이지 “데이터 전송” 참고) 워커로 더 정교한 오프-스레드(`off-thread`) 그래픽 처리를 할 수 있으면 고성능 게임(WebGL) 및 이와 유사한 애플리케이션 구현 시 큰 도움이 될 것이다. 아직은 지원은 브라우저가 하나도 없지만 조만간 현실이 될 것 같다.

웹 워커의 주요 용도는 다음과 같다.

- 처리 집약적 수학 계산
- 대용량 데이터 세트 정렬
- 데이터 작업(압축, 오디오 분석, 이미지 픽셀 변환 등)
- 트래픽 높은 네트워크 통신

### 5.1.2 데이터 전송

웹 워커의 네 가지 주요 용도에는 공통점이 있다. 이벤트 체계를 바탕으로 스레드 간 장벽을 넘어 대량의 데이터가 양방향 전송되어야 하는 요건이다.

초기에는 전체 데이터를 문자열 값으로 직렬화하는 방법뿐이었다. 양방향 직렬화로 인해 속도가 떨어지는 것도 문제지만, 데이터 복사 과정에서 메모리 사용량이 엄청나게 늘어나는(그만큼 가비지 콜렉션량도 늘어나는) 단점이 있었다.

다행히 지금은 더 나은 방법들이 나왔다.

어떤 객체를 전달하면 수신측에서는 ‘구조화된 복제 알고리즘(`structured clone algorithm`)’으로 객체를 복사/복제한다. 이 알고리즘은 상당히 정교해서 `circular reference` 환형 참조 객체도 복제할 수 있다. 객체 ↔ 문자열 변환 비용은 어쩔 수 없지만 이 알고리즘을 쓰면 메모리에 사본을 둘 수 있다(IE10+ 및 주요 브라우저에서 지원된다).

데이터 세트가 방대한 규모라면 '트랜스퍼러블 객체'를 고려하는 것이 좋다. 데이터 자체는 그대로 두고 객체의 소유권만 전송하는 방식이다. 어떤 객체가 워커로 변신하고 나면 원래 위치에서는 텅 빈, 접근할 수 없는 객체이기 때문에 공유 스코프에서 스프레드 프로그래밍의 위험 요소를 제거할 수 있다. 물론 소유권 전송은 양방향 모두 가능하다.

트랜스퍼러블 객체에 옵션을 주고 할 일은 거의 없다. [Transferable 인터페이스](#)를 구현한 자료 구조는 자동으로 그렇게 전송되기 때문이다(파이어폭스, 크롬에서 지원된다).

Uint8Array(본 시리즈 'ES6과 그 이후' 참고) 같은 타입화<sup>typed</sup> 배열이 트랜스퍼러블의 한 예다. 트랜스퍼러블 객체는 `postMessage( ... )`로 전송한다.

---

```
// 예를 들어 'foo'는 'Uint8Array' 타입이다.
```

```
postMessage( foo.buffer, [ foo.buffer ] );
```

---

첫 번째 파라미터는 원시 버퍼<sup>raw buffer</sup>, 두 번째 파라미터는 전송 대상 목록이다.

트랜스퍼러블 객체를 지원하지 않는 브라우저에서는 어쩔 수 없이 구조화된 복제<sup>structured cloning</sup>를 해야 하는데, 특성이 아주 파괴되지는 않지만 성능 저하는 피할 수 없다.

### 5.1.3 공유 워커

같은 페이지(공통 기능)에 탭을 여러 개 읽어들이는 웹 사이트/앱에서는 마땅히 전용 워커가 중복되는 걸 최대한 방지하여 시스템 자원 점유율을 낮춰야 한다. 가장 흔한 제약 자원은 네트워크 소켓 접속으로, 브라우저가 단일 호스트에 동시 접속할 수 있는 개수를 제한한다. 물론, 단일 클라이언트의 다중 접속 제한은 서버 자원 요건을 완화시키는데 도움이 된다.

이런 경우라면 웹 사이트/앱의 페이지 인스턴스가 모두 공유할 수 있는, 하나의 중앙 워커를 두는 것이 좋다.

SharedWorker가 바로 중앙 워커이며 생성 방법은 다음과 같다(파이어폭스, 크롬만 지원한다).

---

```
var w1 = new SharedWorker( "http://some.url.1/mycoolworker.js" );
```

---

공유 워커는 다수의 프로그램 인스턴스/페이지와 연결 가능한 관계로 메시지 출체가 어느 프로그램인지 파악할 방법이 필요하다. 그래서 ‘포트port’라는 고유 식별자가 있다(네트워크 소켓 포트와 비슷하다). 호출 프로그램은 워커의 포트 객체를 통하여 통신한다.

---

```
w1.port.addEventListener( "message", handleMessages );
```

```
// ..
```

```
w1.port.postMessage( "something cool" );
```

---

포트 연결은 반드시 다음과 같이 초기화해야 한다.

---

```
w1.port.start();
```

---

공유 워커 내부에서는 “connect”라는 이벤트를 처리해야 하는데, 이 이벤트는 특정 연결에 관한 포트 객체를 제공한다. 다중 접속을 떼어놓는 가장 편리한 방법은 다음 코드처럼 포트에 클로저(본 시리즈 [‘스코프와 클로저’](#) 참고)를 이용하여 “connect” 이벤트 처리기에 정의된 접속에 대해 이벤트 리스닝/전송을 하는 것이다.

---

```
// 공유 워커 내부에서
```

```

addEventListener( "connect", function(evt){
    // 이 접속에 할당된 포트
    var port = evt.ports[0];

    port.addEventListener( "message", function(evt){
        // ..

        port.postMessage( .. );

        // ..
    } );

    // 포트 접속을 초기화한다.
    port.start();
} );

```

---

이 부분만 차이가 있을 뿐, 공유 워커와 전용 워커의 기능과 의도는 같다.



포트 접속이 끊겨도 다른 포트 접속이 살아있으면 공유 워커는 지속되지만 전용 워커는 자신을 초기화한 프로그램이 종료되면 자취를 감춘다.

### 5.1.4 웹 워커 폴리필

자바스크립트 프로그램의 병렬 실행 시 웹 워커의 성능은 상당히 매력적이지만 실행 환경이 비호환 구형 브라우저인 경우도 감안해야 한다. 워커는 API이고 구문은 아니라서 어느 정도 폴리필이 가능하다.

워커를 지원하지 않는 브라우저는 성능 상 멀티스레딩을 흉내조차 낼 수 없다. 흔히 Iframe으로 적당히 병렬 실행 환경을 모방하려 하지만 실제로 모든 현대 브라우저는 메인 페이지와 동일한 스레드에서 실행하므로 역부족이다.

1장에서 자세히 설명했듯이 자바스크립트 비동기성(병행성이 아니다)의 근원은 이벤트 루프 큐이므로 타이머(setTimeout(..) 등)로 워커를 강제 조작하여 비동기

성을 주는 방법이 있다. 그런 다음 워커 API를 폴리필하면 된다. [Modernizr 깃허브 페이지](#)에 가보면 몇 가지 폴리필이 게시되어 있는데, 솔직히 별로 좋아 보이지 않는다.

그래서 내가 직접 [워커 폴리필](#)을 만들었다. 누추하나마 정확한 양방향 메시징, “onerror” 처리 지원 등 간단한 워커로 쓰기에 충분하다. 필요하다면 `terminate()`나 조작된 공유 워커 같은 기능을 추가 확장할 수 있다.



동기적 종단을 모방하는 건 불가능하므로 이 폴리필을 사용하려면 `importScripts(...)`는 포기해야 한다. (AJAX 로딩이 끝나는대로) 워커 코드를 파싱, 변환해서 프레임리스-인식형 인터페이스가 가미된 `importScripts(...)` 폴리필의 비동기 형태를 다시 만들도록 처리하는 것도 방법이다.

## 5.2 SIMD

SIMD(단일 명령, 다중 데이터Single Instruction, Multiple Data)는 ‘데이터 병행성data parallelism’을 나타내는 형식으로, 웹 워커의 ‘작업 병행성task parallelism’과는 대조적인 개념이다. SIMD의 관심사는 프로그램 로직 덩이들을 병렬 실행하는 게 아니라 여러 데이터 비트를 병렬 처리하는 일이다.

SIMD는 스레드 병행성을 제공하지 않는다. 대신 현대 CPU는 숫자 “벡터vector”(타입화 배열을 생각하면 된다)와 모든 숫자에 병렬 연산이 가능한 명령어 세트(명령어 수준의 병행성을 만드는 저수준low-level 작업을)을 이용하여 SIMD 기능을 제공한다.

자바스크립트에 SIMD를 가져오려는 노력은 (이 글을 쓰는 현재) 인텔의 [모하매드 하그히갓](#)Mohammad Haghghat이 파이어폭스/크롬 개발팀과 함께 주도하고 있다. 아직 초기 단계에 불과하지만 SIMD는 자바스크립트 다음 표준(아마도 ES7)에 포함될 것 같다.

SIMD 자바스크립트는 단축 벡터 타입과 API를 자바스크립트 코드의 일부분

처럼 사용하는데, 이러한 SIMD 구동 시스템은 처리할 작업들을 CPU 동등체 equivalents로 직접 매핑한다. 병렬화하지 않은 작업은 SIMD 아닌non-SIME 시스템에 “십shim”하여 대체한다.

데이터 집약적data-intensive 애플리케이션(신호 분석, 그래픽 행렬 계산 등)은 보통 수학 계산을 병렬로 처리하므로 뚜렷한 성능 향상을 기대할 수 있다!

이 글을 쓰는 시점에 제안된 SIMD API 초기 형태는 다음과 같다.

---

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );
```

```
var v3 = SIMD.int32x4( 10, 101, 1001, 10001 );
var v4 = SIMD.int32x4( 10, 20, 30, 40 );
```

```
SIMD.float32x4.mul( v1, v2 );
// [ 6.597339, 67.2, 138.89, 299.97 ]
SIMD.int32x4.add( v3, v4 );
// [ 20, 121, 1031, 10041 ]
```

---

32비트 부동 소수점 숫자와 32비트 정수, 상이한 두 가지 벡터 데이터 타입이 보인다. 이들 벡터 크기가 정확히 32비트 요소 4개이므로 요즘 거의 모든 CPU에서 가용한 SIMD 벡터 사이즈(128비트)와 딱 맞는다. 언젠가는 x8(또는 그 이상!) API 버전도 나올 것이다.

mul(), add() 이외에도 sub(), div(), abs(), neg(), sqrt(), reciprocal(), reciprocalSqrt() (산술), shuffle() (벡터 원소 재배열), and(), or(), xor(), not() (논리), equal(), greaterThan(), lessThan() (비교), shiftLeft(), shiftRightLogical(), shiftRightArithmetic() (시프트), fromFloat32x4(), fromInt32x4() (변환) 등 많은 연산들이 포함될 성싶다.



현재 공식 SIMD "프롤리필<sup>prollyfill</sup>"(앞으로 이렇게 나올 거라고 기대하고 바라는 미래 지향적 폴리필) ([https://github.com/tc39/ecmascript\\_simd](https://github.com/tc39/ecmascript_simd))이 나와있는데, 이 절에서 설명한 것보다 훨씬 더 많은 SIMD 기능이 들어있다.

### 5.3 asm.js

asm.js는 자바스크립트 언어에서 고도로 최적화 가능한 부분 집합을 말한다. 최적화하기 어려운, 특정한 체계와 패턴(가비지 콜렉션, 강제변환 등)을 방지한 asm.js 식 코드는 자바스크립트 엔진이 인식하여 아주 공격적으로 저수준 최적화를 하는 등 특별히 조치를 한다.

이 장에서 언급한 프로그램 성능 체계와는 완전히 다른 asm.js는 사실 자바스크립트 언어 명세로 채택되어야 할 필요는 없다. 이미 **asm.js 명세**가 나와 있는데 자바스크립트 엔진의 요건을 정리했다기 보다는 대부분 최적화 문제에 관한 수렴된 의견을 추적하기 위한 용도로 작성됐다.

새로운 구문이 제안된 상태도 아니다. 다만 asm.js 규칙과 부합하는 현행 표준 자바스크립트 구문을 인식하고 이에 따라 엔진이 어떻게 자체 최적화 할지 방향을 제시하는 정도다.

프로그램에서 asm.js를 정확히 어떤 방법으로 활성화할지는 브라우저 업체 간의 견이 엇갈려왔다. 초창기 asm.js 베타 버전은 "use asm"; 지시자<sup>pragma</sup>(엄격 모드 지시자 "use strict"와 비슷하다)를 쓰게 하여 자바스크립트 엔진의 asm.js 최적화 여부를 경고/암시했다. asm.js은 말하자면 휴리스틱<sup>heuristics</sup><sup>02</sup> 같은 것으로, 개발자가 굳이 뭘 추가하지 않아도 자동으로 엔진이 인식할 수 있어야 한다고 주장하는 이들도 있다. 즉, 예전 프로그램도 특별히 뭘가 해주지 않아도 asm.js 최적화

---

<sup>02</sup> 역자주\_휴리스틱은 종합적, 합리적, 체계적인 판단을 내릴 만큼 정보가 충분하지 않거나 굳이 그럴 만한 필요가 없는 상황에서 한 마디로 '대충 어림짐작으로' 신속하게 내리는 행위를 말합니다.

를 통해 개선이 가능하다는 뜻이다.

### 5.3.1 asm.js 최적화

asm.js 최적화를 이해하려면 먼저 타입과 강제변환(본 시리즈 '타입과 문법' 참고)을 알고 있어야 한다. 자바스크립트 엔진이 다양한 연산 도중 상이한 타입의 변수값들을 추적하면서 필요 시 타입 간 강제변환을 처리하는 상황에서 프로그램 최적화를 방해하는 잡다한 요소가 너무 많다.



이 책에서 설명을 위해 asm.js식 코드를 사용하지만 여러분이 현장에서 직접 이렇게 코딩할 일은 거의 없다. asm.js는 [엠프스크립트](#) Emscripten 같은 다른 툴에서 컴파일 타겟 용으로 더 많이 쓰인다. asm.js 코드를 손수 작성한다고 문제될 건 없지만, 매우 저 수준의 코딩인데다 관리하는데 시간이 많이 들고 에러가 나기 쉬워 좋은 생각 같진 않다. 물론 어쩔 수 없이 자신의 코드를 약간 수정해서 asm.js 최적화를 해야 할 경우도 더러 있다.

asm.js-인식형 자바스크립트 엔진이 변수/연산 타입을 추론할 수 있게 힌트를 주고 강제변환 추적 단계를 건너뛰도록 하는 몇 가지 수법이 있다.

예를 들면,

---

```
var a = 42;
```

```
// ..
```

```
var b = a;
```

---

b = a 할당 결과 변수 타입은 천차만별일 수 있으니 다음과 같이 바꿔쓴다.

---

```
var a = 42;
```

```
// ..
```

```
var b = a | 0;
```

---



값 자체는 아무런 영향도 없지만 32비트 정수 타입이 되도록 강제하려고 0을 | (이진 OR) 연산한다. 이 코드는 일반 자바스크립트 엔진에서도 잘 실행되지만 asm.js-인식형 엔진에서 실행하면 반드시 b를 32비트 정수로 취급해야 한다고 알려주고 강제변환 추적을 건너뛸 수 있다.

비슷하게 두 변수의 덧셈 연산도 요건에 (부동 소수점 연산 대신) 맞게 형태를 제한할 수 있다.

---

$$(a + b) \mid 0$$

---

전체 표현식의 평가 결과가 32비트 정수로 자동 변환되기 때문에 이를 토대로 asm.js-인식형 자바스크립트 엔진은 +가 32비트 정수의 덧셈 연산이란 사실을 미리 추론할 수 있다.

### 5.3.2 asm.js 모듈

메모리 할당, 가비지 콜렉션, 스코프 접근은 늘 자바스크립트 성능 문제의 논란거리였다. 그 해결 방안으로 asm.js에서는 더 정형화한 asm.js ‘모듈’(ES6 모듈과 헛갈리지 말자. 본 시리즈 ‘ES6과 그 이후’ 참고)을 선언한다.

asm.js 모듈은 단순히 어휘 스코프<sup>lexical scope</sup>를 통해 전역 객체를 쓰는 대신 필요한 심볼을 가져오기 위해 엄격하게 규정된 명칭공간(표준 라이브러리라는 의미에서 명세에는 stdlib라고 나와있다)을 분명히 전달한다. window 객체는 asm.js 모듈이 의도한 바와 잘 맞는 기본 stdlib 객체지만 더 제한된 객체를 만들(만들어야 할) 수도 있다.

또, 힙<sup>heap</sup>(메모리에 예약된 공간을 나타내는 기술 용어로, 메모리를 추가하거나 과거에 점유한 메모리를 해제하지 않고도 변수가 사용할 수 있는 메모리 영역)을 반드시 선언하고 전달해서

asm.js 모듈이 메모리 천(memory churn<sup>03</sup>)을 일으키지 않고 사전 예약된 공간을 사용할 수 있게 한다.

힙은 타입화한 ArrayBuffer와 비슷하다.

---

```
var heap = new ArrayBuffer( 0x10000 ); // 64k 힙
```

---

asm.js 모듈은 메모리 할당, 가비지 콜렉션 같은 불이익을 받지 않고도 미리 예약된 64k 이진 공간을 버퍼 삼아 값을 저장/조회할 수 있다. 예를 들어, 모듈 내에 다음 64비트 부동 소수점 값들을 힙 버퍼에 담을 수 있다.

---

```
var arr = new Float64Array( heap );
```

---

자, 그럼 asm.js식 모듈을 간략히 한번 작성해보면서 지금까지 설명한 퍼즐 조각을 한데 맞춰보자. 어떤 범위를 표시하는 시작(x), 끝(y) 정수를 인자로 받아 범위 내 인접값을 모두 곱하고 평균값을 계산하는 foo(..) 함수를 생각해 보자.

---

```
function fooASM(stdlib,foreign,heap) {
    "use asm";

    var arr = new stdlib.Int32Array( heap );

    function foo(x,y) {
        x = x | 0;
        y = y | 0;

        var i = 0;
        var p = 0;
        var sum = 0;
        var count = ((y|0) - (x|0)) | 0;
```

---

**03** 역자주\_가비지 콜렉터(GC)가 메모리 상의 객체 생성/제거를 빈번하게 일으키는 현상을 말합니다. 애플리케이션 실행 중 메모리 천이 자주 발생하면 그만큼 애플리케이션 실행 시간을 많이 점유하게 되므로 성능 저하가 유발됩니다. 천(churn)은 우리말로 '휘젓기', '흔들기'란 뜻입니다.

```

// 내부 인접값을 모두 곱한다.
for (i = x | 0;
     (i | 0) < (y | 0);
     p = (p + 8) | 0, i = (i + 1) | 0
) {
    // 결과를 저장한다.
    arr[ p >> 3 ] = (i * (i + 1)) | 0;
}

// 모든 중간 단계의 값을 평균 낸다.
for (i = 0, p = 0;
     (i | 0) < (count | 0);
     p = (p + 8) | 0, i = (i + 1) | 0
) {
    sum = (sum + arr[ p >> 3 ]) | 0;
}

return +(sum / count);
}

return {
    foo: foo
};
}

```

```

var heap = new ArrayBuffer( 0x1000 );
var foo = fooASM( window, null, heap ).foo;

foo( 10, 20 ); // 233

```



교육용으로 대충 작성한 코드라 asm.js를 타겟으로 컴파일 툴이 생성한 코드와는 당연히 다르지만 asm.js 코드의 전형적인 특징, 특히 타입 암시와 임시 변수 저장용으로 힙 버퍼를 사용한 모습을 엿볼 수 있다.

fooASM( .. )을 호출하면 힙을 할당 후 asm.js 모듈을 시작한다. 그 결과 무제한 호출 가능한 foo( .. ) 함수를 반환하고 asm.js-인식형 자바스크립트 엔진은 foo( .. ) 호출을 특별한 방법으로 최적화한다. 이 예제 코드는 자바스크립트 표준을 완벽히 준수하기 때문에 asm.js 아닌 엔진에서도 (별다른 최적화 없이도)

잘 실행된다는 걸 알아두자.

asm.js 코드에 최적화 능력을 부여하는 제약의 본질 상 그 사용 영역 또한 한정되는 건 어쩔 수 없다. asm.js은 모든 자바스크립트 프로그램에서 쓸 수 있는 최적화 도구라기 보다는, 게임 그래픽 처리 특유의 집중적인 수학 연산 등 특수 작업에 최적화된 도구다.

## 5.4 정리하기

1 ~ 4장에서는 비동기 코딩 패턴 덕분에 더 성능 좋은 코드를 작성할 수 있고 이는 매우 중요한 개선 포인트라는 전제를 했었다. 하지만 비동기 특성 또한 근본적으로는 단일 이벤트 루프 스레드에 묶여 있기 때문에 한계가 있다.

그래서 이 장에서는 프로그램 수준에서 성능을 개선할 수 있는 체계를 세 가지 다루었다.

첫째, 웹 워커는 비동기 이벤트를 이용하여 스레드 간에 메시지를 교환하면서 자바스크립트 파일(프로그램)을 개별 스레드 단위로 실행하게 해준다. 메인 UI 스레드의 응답성을 높이면서도 소요 시간이 길거나 자원을 집중적으로 소모하는 작업을 다른 스레드로 분산시킬 수 있는 장점이 있다.

둘째, SIMD은 CPU 수준의 병렬 수학 연산을 대량 데이터의 수치 연산 같은 고성능 병렬 데이터 연산에 특화된 자바스크립트 API로 연결짓는 기법이다.

셋째, asm.js는 (가비지 콜렉션, 강제변환 등) 최적화하기 어려운 영역을 피해서 자바스크립트 엔진이 이런 부류의 코드를 자동 인식하여 공격적인 최적화를 하도록 유도하는 자바스크립트의 부분 집합이다. asm.js는 직접 작성할 수 있지만 어셈블리 언어 코딩만큼이나 매우 지리하고 실수하기 쉽기 때문에 요점은 C/C++를 자바스크립트로 트랜스파일하는 엠스크립트처럼 다른 고도로 최적화된 프로그래밍

언어에서 크로스 컴파일할 대상으로 한다는 것이다.

이 장에서 확실하게 다루지는 않았지만 (자료 구조 API 내부에 그냥 감춘 게 아니라) 직접 스레딩을 근사화한 기법을 비롯하여 자바스크립트를 대상으로 한 아주 초기 단계의 논의들 중에는 더 근본적인 아이디어들도 있다. 명시적이든 암시적이든 앞으로 더 많은 병행성이 자바스크립트의 일부로 굳혀지면서 조만간 자바스크립트 언어도 프로그램 수준에서 더욱 최적화된 성능을 발휘하게 되리라고 본다.

# 벤치마킹과 튜닝

1 ~ 4장에서는 코딩 패턴(비동기성과 동시성)으로서의 성능을, 그리고 5장에선 거시적인 프로그램 아키텍처 관점에서의 성능을 이야기하였다. 6장은 표현식/문 하나 하나에 초점을 두고 미시적 수준의 성능 문제를 다룬다.

일반적으로 개발자들은 다양한 코딩 옵션을 분석/테스트해보고 어느 쪽이 더 빠른지 결정하는 일에 관심이 많다. (실제로 어떤 이는 중독자에 가까운 경지에 이른 경우도 본 적 있다)

이와 관련된 몇 가지 이슈를 차근차근 살펴볼 텐데, 나는 자바스크립트 엔진이 ++a를 a++보다 빨리 실행한다, 하는 미시적인 성능 튜닝<sup>micro-performance tuning</sup>에는 집착하지 않을 것이다. 그보다 일단 자바스크립트 성능에 있어서 중요한 부분과 그렇지 않은 부분을 가려내고 그 차이점이 무엇인지 밝히는 일에 초점을 두었다.

자, 먼저 자바스크립트 성능을 가장 정확하고 믿음성 있게 테스트하는 방법부터 알아보자. 너무나 많은 오해와 미신이 그릇된 지식 근간으로 굳어진 현실은 어쩔 수 없지만 지금이라도 명쾌하게 옥석을 분별하고 머릿속을 정리하자.

## 6.1 벤치마킹

마음 속에서 오해와 미신을 내쫓을 시간이다. 장담컨대 여러분을 포함해서 개발자 대부분은 속도(실행 시간) 벤치마킹을 할 때 대개 이렇게 한다.

---

```
var start = (new Date()).getTime(); // 또는 'Date.now()'

// 어떤 작업을 수행한다.

var end = (new Date()).getTime();

console.log( "소요 시간:", (end - start) );
```

---

대략 내 말이 맞다면 손들고 자수하자. 하하, 그럴 줄 알았다. 이 방법은 완전히 틀렸다. 언짢아 하진 마시길, 다들 그랬으니까.

대관절 시간을 계측해서 무슨 정보를 얻겠다는 말인가? 자바스크립트 성능 벤치마킹은 주어진 작업 실행 시간 동안 측정한 결과를 보고 무엇을 알 수 있는지, 무엇을 알 수 없는지부터 파악하는 게 기본이다.

만약 소요 시간이 0으로 측정됐다면 1ms도 안 되는 시간 동안 실행이 끝났다고 단정하기 쉽지만 정말 부정확하기 짝이 없는 판단이다. 정확도 자체가 1ms 보다 떨어지고 더 큼지막한 단위로 타이머를 업데이트하는 플랫폼도 있으니 말이다. 예컨대, 윈도우 구버전(그리고 IE)의 정확도는 15ms여서 이보다 짧은 시간은 모조리 0으로 표시한다!

계다가 결과 수치야 어떻든 소요 시간만으로는 작업을 딱 한 번 실행할 때 그만큼 시간이 걸린다는 정도만 알 수 있다. 늘 그 속도로 돌아가리란 보장이 없다. 엔진/시스템이 그때그때 어떤 식으로 간섭을 일으킬지 예측할 수 없고 더 빨리 실행하는 경우도 더러 있다.

소요 시간이 4로 나왔다고 하자. 정확히 4ms 걸렸단 말인가? 아니다. start, end 타임스탬프를 얻는 과정에서도 지연이 발생 가능해서 실제로는 4ms 이하일 것이다.

더 큰 문제는, 테스트 환경이 지나치게 낙관적이라는 사실을 모른다는 점이다. 독

립적인 테스트 케이스는 자바스크립트 엔진이 나름대로 공리 끝에 최적화할 가능성이 높지만, 실제 프로그램은 최적화 정도가 떨어지거나 아예 불가능할 때도 있어 테스트했을 때보다 더 느려지는 게 보통이다.

자, 그럼 우리는 무엇을 알 수 있을까? 아쉽게도 이러한 현실적 한계로 인해 알 수 있는 게 별로 없다. 사정이 이러하니 자신없게 쓴 보고서가 효과적인 의사 결정 자료로 쓰일 리 만무하다. 이런 벤치마킹은 일단 가치가 없고 잘못된 확신을 심어줄 수 있어서 위험하다. 여러 가지 조건을 제대로 따져보지도 않은 채 다른 사람들이 잘못 판단하여 곤경에 빠질지도 모른다.

### 6.1.1 반복

“좋아, 그럼 테스트 루프를 여러 번 돌리면 되겠군.” 혹시 여러분의 생각도...? 테스트를 100회 반복한 결과가 137ms라면 100으로 나누어 작업 당 소요 시간은 1.37ms라고 단정할 수 있을까?

그렇지 않다.

단순 평균치만으로는 전체 애플리케이션의 성능에 관하여 어떤 결론을 내리기 어렵다. 100회 반복 테스트를 한다 해도 (높든 낮든) 이상점<sup>01</sup>이 2개만 있어도 평균은 왜곡될 수 있고, 여기서 도출된 결론을 반복 적용하면 왜곡된 사실은 더욱 더 부풀려지고 만다.

반복 횟수를 고정하는 대신 일정 시간 테스트를 반복하는 방법도 있다. 좀 더 미더운 결과가 나올지는 모르겠지만, 여기서 ‘일정 시간’의 기준은 어떻게 정할까? 1회 실행 시간의 몇 배수 정도면 되지 않겠느냐 생각하겠지만... 틀렸다.

오류 빈도를 최소화하려면 사용 중인 타이머의 정확도에 따라 반복 테스트 시간을 결정해야 한다. 부정확한 타이머일수록 테스트 시간을 늘려 가능한 한 에러를 줄

---

01 역자주\_다른 측정치보다 지나치게 동떨어진 값을 말합니다.



여야 한다. 15ms 타이머로는 정확히 벤치마킹하기 힘들다. 이 타이머로 불확실성(에러율)을 1% 이하로 낮추려면 적어도 750ms 동안은 테스트해야 한다. 1ms 타이머는 50ms만 해도 동일한 수준의 믿음성을 확보할 수 있다.

하지만 단지 샘플일 뿐이다. 왜곡을 확실히 상쇄하려면 평균 낼 샘플이 아주 많아야 한다. 또 최악의 샘플은 얼마나 느린지, 최고의 샘플은 얼마나 빠른지, 그리고 이 최악/최고 간 격차는 얼마인지 파악해야 한다. 어느 것이 얼마나 빠르다, 하는 수치뿐 아니라 이 수치를 어느 정도 신뢰할 수 있을지 판단의 근거가 되는 정량적 데이터도 필요하다.

이처럼 여러 가지 상이한 기법들을 조합하여 모든 가용한 접근 방식 간의 균형점을 찾아야 한다.

최소한 충족해야 할 기준이 이 정도다. 여러분이 지금까지 훑어본 내용보다 진지하게 성능 벤치마킹을 고민한 적이 없다면 음... 벤치마킹을 전혀 모르는 것이나 다름없다.

### 6.1.2 Benchmark.js

유의미한, 믿음성 있는 벤치마킹은 반드시 통계적으로 검증된 지침에 근거한다. 이 책에서 통계학을 다룰 생각은 없으니 표준 편차<sup>standard deviation</sup>, 분산<sup>variance</sup>, 오차 한계<sup>margin of error</sup> 같은 용어는 따로 설명하지 않는다. 하지만 지금껏 이런 용어를 한번도 들어본 적 없는 사람이라면(학부 시절 통계학 수업을 들었던 나 역시 가끔 가물거울 할 때가 있다) 실무에서 벤치마킹 로직을 운용할 자격이 없다.

다행히 존-데이빗 달튼<sup>John-David Dalton</sup>과 마티아스 바이넨스<sup>Mathias Bynens</sup>처럼 개념을 통달한 똑똑한 사람들이 **Benchmark.js**라는, 통계학적으로 검증된 벤치마킹 도구를 세상에 내놓았다. 내가 하고 싶은 말은 "그냥 속편하게 이 툴을 가져다 쓰라"는 것이다.

Benchmark.js 기술 문서를 이 책에 재탕하진 않겠다. API 문서가 잘 정리되어 있으니 참고하자. 구현 상세와 방법론에 관한 좋은 읽을거리(<http://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks>, <http://monsur.hossa.in/2012/12/11/benchmarkjs.html>)도 있으니 일독을 권한다.

다음 예제를 보면 Benchmark.js로 신속하게 성능 테스트 하는 방법을 알 수 있다.

---

```
function foo() {
  // 테스트할 작업(들)
}

var bench = new Benchmark(
  "foo test", // 테스트 명
  foo, // 테스트할 함수 (내용만)
  {
    // .. // 추가 옵션 (문서 참조)
  }
);

bench.hz; // 초당 작업 개수
bench.stats.moe; // 한계 에러
bench.stats.variance; // 분산
// ..
```

---

수박 겉핥기로 예시한 내용 외에 Benchmark.js는 공부할 분량이 제법 많다. 요는, 자바스크립트 코드의 올바른, 미더운, 유효한 성능 벤치마킹에 필요한 제반 설정의 복잡함을 Benchmark.js가 고맙게도 대신 처리해준다는 점이다. 이 라이브러리를 코드 테스트/벤치마킹에 잘 활용하기 바란다.

예제는 하나의 작업 X를 테스트하고 있지만 또 다른 작업 Y와 비교하고 싶을 때가 많다. 이럴 때 상이한 두 테스트를 묶어 ‘스위트<sup>suite</sup> (Benchmark.js의 조직화 기능)’로 설정하고 서로 진검 승부를 펼칠 수 있게 명석을 알아주면 된다. 그 후 집계된

수치를 비교해서 X, Y 중 어느 쪽이 더 빠른지 판단한다.

Benchmark.js는 브라우저 환경의 자바스크립트 테스트에 주로 쓰지만, 브라우저 이외의 환경(노드JS 등)에서도 사용할 수 있다.

향후 Benchmark.js는 개발 또는 QA 환경에서 애플리케이션 자바스크립트 코드의 임계 경로 부분에 대한, 자동화 성능 회귀 테스트<sup>test02</sup> automated performance regression 도구로 많이 쓰일 것 같다. 배포 전 단위 테스트 스위트를 실행하는 것처럼 이전에 수행한 벤치마크와 비교하여 애플리케이션 성능이 점점 개선/악화되고 있는지 모니터링할 수 있다.

## 설정/정리

방금 전 예제 주석을 보면 “추가 옵션”을 { .. } 객체에 넣는데 이 자리에 ‘설정 및 정리’ 코드를 지정한다.

테스트 케이스를 실행하기 이전/이후에 호출할 함수를 각각 정의한다.

설정/정리는 테스트가 반복될 때마다 실행되는 코드가 아니라는 점을 유의하자. 바깥쪽 루프(사이클을 반복)와 안쪽 루프(테스트를 반복)가 따로 있다고 생각하면 이해가 빠르다. 여기서 설정/정리 코드는 안쪽 루프가 아닌, 바깥쪽 루프(사이클)의 반복 시작/끝 부분에서 실행된다.

왜 이런 얘기를 꺼낸 걸까? 다음과 같은 테스트 케이스가 있다고 치자.

---

```
a = a + "w";  
b = a.charAt( 1 );
```

---

테스트 설정은 이렇게 한다.

---

**02** 역자주\_중요한 모듈의 소스 코드를 변경했을 때 오히려 변경 전에 비해 성능에 부정적인 영향은 없는지 확인하기 위한 테스트를 말합니다.

```
var a = "x";
```

이렇게 보면 매번 테스트 반복 시 a가 “x” 값으로 시작한다고 착각하기 쉽다.

분명히 아니다! 테스트 사이클을 주기로 a는 “x” 값으로 시작하고 실제로 1 자리의 “w” 문자에만 접근하고 + “w” 덧붙이기가 이어지면서 a값은 점점 불어날 것이다.

DOM에서 자식 요소 덧붙이기 등의 작업을 할 때 부수 효과<sup>side effect</sup>가 일어나는 경우도 여러분을 골탕먹이는 상황이다. 매번 부모 요소가 빈 상태로 시작하리라 상상하지만, 실제로 이미 잡다한 요소들이 붙어버린 후라 테스트 결과에 지대한 영향을 끼치게 된다.

## 6.2 컨텍스트가 제일

X, Y 작업 간 성능 비교 등 특정한 성능 벤치마킹을 할 때 컨텍스트를 살피는 걸 잊으면 안 된다. 테스트 결과, X가 Y보다 빠른 것처럼 나왔다고 해서 실제로 “X가 Y보다 빠르다”고 결론 내릴 수 없다.

가령, 측정 결과 초당 X는 10,000,000개, Y는 8,000,000개의 연산을 각각 수행했다고 하자. 수치만 봐서는 분명히 Y가 X보다 20% 더 느리지만, 여러분의 생각처럼 그렇게 단정지어 말할 만큼 확실하지는 않다.

테스트 결과를 좀 더 면밀하게 관찰하라. 1초에 10,000,000개면 ms당 10,000개,  $\mu$ s당 10개고, 연산 한 개에 0.1 $\mu$ s, 또는 100ns가 걸리는 셈이다. 사람 눈동자가 100ms(100ns 보다 백만배 느리다)보다 빨리 움직이는 물체를 식별할 수 없을 정도니 100ns이 얼마나 짧은 시간인가!

최근 연구 결과에 따르면 두뇌의 처리 속도가 이론적으로는 13ms(100ms 보다는 8배 빠른 수치)까지 가능하다고 하는데 X는 이보다도 125,000배 더 빠른 것이다.

X는 진짜 엄청, 엄청나게 빠르다.

그러나 초당 2,000,000개라는 X, Y의 차이가 더 중요하다. X가 100ns, Y가 80ns 걸리면 그 시간차는 20ns인데, 이는 기껏해야 두뇌가 인식할 수 있는 시차의  $1 / 650,000$  정도에 불과하다.

결론은? 이런 성능 차이는 아무 의미도 없다!

잠깐... 그럼 만약 같은 작업을 오랫동안 연속하여 실행하면? 응당 성능 차이가 두드러지지 않을까?

좋다, 한번 스스로에게 물어보라. X 실행을 계속 반복하면서 언젠가는 두뇌가 인지할 정도에 이르게 될거란 실낱같은 희망으로 650,000회 실행할 가능성은 얼마나 될까? 말이 되게 하려면 5,000,000 ~ 10,000,000회 정도 뽀박하게 돌려야 할 것이다.

독자들 중 컴퓨터 과학자가 있다면 불가능한 얘기는 아니라고 주장할지 몰라도, 실제로 이게 얼마나 가능/불가능한 애긴지 정상 테스트를 해야 한다는 현실주의자들의 목소리가 더 설득력이 있을 게다. 그렇지 않은 경우도 물론 드물게 있겠지만 대부분은 맞다.

(‘+++ 대 x++’ 신화처럼) 미세 연산에 관한 벤치마킹 결과는 대부분 X가 Y보다 성능이 더 우수하다는, 말갈지 않은 결론을 뒷받침하는 가짜 근거일 뿐이다.

### 6.2.1 엔진 최적화

고립 테스트<sup>isolated test</sup> 결과, X가 Y보다 10ms 빠르게 나왔다고 단순히 X는 Y보다 항상 빠를 것이라고 넘겨짚는 건 위험한 발상이다. 사실 성능은 다루기가 아주 까다롭다.

예를 들어, 다음과 같이 미세 성능<sup>microperformance</sup>을 테스트한다고 치자.

---

```
var twelve = "12";
var foo = "foo";

// 테스트 1
var X1 = parseInt( twelve );
var X2 = parseInt( foo );

// 테스트 2
var Y1 = Number( twelve );
var Y2 = Number( foo );
```

---

`Number(..)`와는 다르게 `parseInt(..)`가 어떻게 작동할지 아는 사람은 `parseInt(..)`가 인자가 `foo`인 경우 필경 하는 일이 더 많거나, 아니면 어차피 두 함수 모두 첫 번째 문자, `f`에서 멈출 테니 작업량은 엇비슷할 거라 추측할 것이다.

정답은 어느 쪽일까? 난 솔직히 잘 모르겠지만 어차피 어느 쪽이 정답이든 상관없다. 여러분이 테스트하면 결과는 어떻게 나올까? 순전히 가설에 지나지 않을 뿐이고, 실제로 난 이런 테스트 해본 적도 없고 해보고 싶지도 않다.

좋다, `X`, `Y`를 테스트한 결과, 통계적으로 일치했다고 치자. 그럼 여러분은 `f` 문자에 대한 자신의 직관을 확신할 수 있을까? 아니다.

내 가설에 따르면, 자바스크립트 엔진은 두 테스트 모두 `twelve`, `foo`를 한 곳에서만 사용하므로 이 값들을 인라인<sup>inline</sup>하기로 결정할 것이다.<sup>03</sup> 결국 엔진은 `Number( "12" )`를 그냥 `12`로 대체해도 상관없겠구나 인지한 뒤, `parseInt(..)`에도 같은 결론을 내리거나, 또는 다르게 판단할 것이다.

아니면 엔진 고유의 데드코드<sup>dead-code</sup> 제거<sup>04</sup> 휴리스틱이 개입되어 `X`, `Y`가 실은

---

<sup>03</sup> 역자주 `C/C++` 프로그래밍의 '매크로(macro)와 인라인(inline)'을 생각하면 이해가 빠릅니다. 매크로와 인라인의 정의와 컴파일러가 이들을 처리하는 작동 원리는 구글링해보시면 좋은 자료가 많으니 이 책을 읽는 김에 반드시 참고하시기 바랍니다.

<sup>04</sup> 역자주 데드코드 제거는 컴파일러의 최적화 옵션에 의해 수행되는 로직으로, 사용하지 않는 계산 또는 달을 수 없는 코드를 생략하는 것을 말합니다.

사용하지 않는 변수라 선언 자체가 무의미하고 두 테스트에서 아무 일도 하지 않는 백수임을 알게 될 것이다.

지금까지는 테스트 하나만 두고 대략 추정해본 것이다. 현대 자바스크립트 엔진은 우리가 상상의 나래를 펼쳐 짐작할 수준과 비교조차 안 될 정도로 어마어마하게 복잡하다. 단시간 내에, 특정 입력 제약 조건에서 어느 코드 조각이 어떻게 작동하는지 별의 별 꼼수를 동원해 추적한다.

어떤 고정된 입력을 받으면 정해진 방향으로 엔진을 최적화하도록 설계했는데, 실제 프로그램은 다양한 입력을 받아 전혀 다른 식의 최적화를 하기로(아니면 아예 안 하기로) 엔진이 결정을 내린다면? 또는 어떤 벤치마킹 유틸로 특정 코드를 몇 만번 반복 실행할 때에는 엔진이 최적화에 개입했는데, 실제 프로그램에선 대략 100번 정도밖에 안 돌아갔다면 이 때에도 엔진은 최적화를 할 가치가 있다고 결정할까?

지금까지 가정한 모든 최적화는 내가 설정한 제약 조건에서 일어날 수도 있지만 더 복잡한 프로그램에서는 엔진이 (여러 가지 이유에서) 안 하기로 결단을 내릴지 모른다. 아니면 반대로 자잘한 코드 따위는 신경도 쓰지 않던 엔진이 다소 복잡한 프로그램이 시스템 자원을 많이 거덜낸 상황이라면 아주 공격적으로 최적화를 하기 위해 에너지를 쏟아부을 가능성도 있다.

내가 하고 싶은 말은 여러분들과 나는 저 깊숙한 곳에서 무슨 일이 벌어질지 도저히 알 수 없다는 점이다. 추측/추정만 무성할 뿐 확실한 결정을 내릴 만한 구체적인 증거는 아니다.

그럼 유용한 테스트는 꿈도 꾸지 말아야 하나? 그건 아니다!

‘실제 코드’ 아닌 테스트만으로 ‘실제 결과’를 알 수는 없다는 말이다. 현실적으로 가능하다면 진짜 실제와 가장 가까운 환경에서 여러분이 생각하기에 정말 중요하고 의미있는 코드를 테스트하라. 그래야만 본질에 가까운 결과를 얻게 될 가능성이 조금이라도 높아질 것이다.

‘++x 대 x++’ 따위의 미세한 벤치마킹은 생각할 가치도 없으니 그냥 그런가보다 하고 지나치는 게 상책이다.

## 6.3 jsPerf.com

Benchmark.js는 자바스크립트 실행 환경에 구애받지 않고 코드 성능을 테스트 할 수 있는 유용한 도구지만, 신뢰할 수 있는 결론을 원한다면 다양한 환경(테스크 톱 브라우저, 모바일 디바이스 등)에서 테스트한 결과를 취합해야 한다는 사실은 아무리 강조해도 지나치지 않다.

예컨대, 고성능 데스크톱의 크롬과 스마트폰의 크롬 모바일 성능이 차이나는 건 당연하다. 또한 100% 배터리 충전이 끝난 스마트폰과 2% 밖에 남지 않아 디바이스 차원에서 무선/프로세서 전원을 차단하기 시작한 스마트폰이 동일한 성능을 보일 리 만무하다.

다수의 환경에서 “X가 Y보다 빠르다” 하고 합리적인 결론을 내리려면 가급적 다양한 실제 환경에서 테스트를 실시해야 한다. 크롬에서 실행하니 X가 Y보다 빠르다고 해서 다른 브라우저도 다 그럴 거라 넘겨짚는 행위는 금물이다. 또한 실 사용자를 대상으로 멀티 브라우저 테스트한 결과를 상호 참조cross-reference할 필요성도 있다.

jsPerf가 바로 이런 목적으로 개발된 멋진 웹 사이트로, 접속 가능한 공개 URL에 대하여 앞서 소개한 Benchmark.js 라이브러리를 이용하여 통계적으로 정확하고 믿음성 있는 테스트를 대행한다.

테스트를 할 때마다 결과를 수집/저장하고 다른 사람도 볼 수 있도록 한 페이지에 누적 테스트 결과를 그래프로 보여주기도 한다.

jsPerf에서 테스트를 새로 만들면 일단 테스트 케이스 2개로 시작하고 원하면 케



이스를 더 추가할 수 있다. 테스트 사이클 앞부분에서 실행할 설정 코드, 끝부분에서 실행할 정리 코드를 지정할 수도 있다.



(정백전이 아닌 어느 한쪽만 벤치마킹하기 위해) 테스트 케이스를 하나만 실행하려면 일단 두 번째 테스트 입력 박스에 처음 생성할 때의 자리까운placeholder 텍스트를 써넣고 테스트 편집 후 두 번째 테스트를 비워두면 삭제된다. 물론 테스트 케이스는 나중에 언제고 추가할 수 있다.

초기 페이지 설정(라이브러리 가져오기, 유틸 헬퍼 함수 정의, 변수 선언 등)도 가능하며, 필요 시 설정/정리 로직을 정의하는 옵션도 지원한다.

### 6.3.1 정상 테스트

jsPerf는 멋진 도구지만 잘 들여다보면 지금까지 이 장에서 설명해온 이유 때문에 결함이 있거나 믿을 수 없는 테스트가 상당히 많다는 사실을 알 수 있다.

다음 코드를 보자.

---

// 케이스 1

```
var x = [];  
for (var i=0; i<10; i++) {  
    x[i] = "x";  
}
```

// 케이스 2

```
var x = [];  
for (var i=0; i<10; i++) {  
    x[x.length] = "x";  
}
```

// 케이스 3

```
var x = [];  
for (var i=0; i<10; i++) {  
    x.push( "x" );  
}
```

---

이런 테스트 시나리오가 있을 때 잘 따져봐야 할 항목들을 정리한다.

- Benchmark.js가 필요 시 반복을 대신한다는 사실을 까맣게 잊은 채 개발자가 테스트 케이스에 루프를 직접 넣는 경우가 비일비재하다. 결국 for 루프는 완전히 쓸데없는 잡음으로 남게 될 가능성이 매우 높다.
- 테스트 케이스마다 x를 선언/초기화했는데 사실 불필요한 코드다. 설정 코드에 x = [] 하면 테스트 케이스가 아닌, 테스트 사이클 시작부에서 한번만 실행됐을 것이다. 즉, for 루프에서 의도한 크기 10을 넘어 x 값은 계속 불어나게 될 것이다.

그럼 여기서 테스트 작성자의 의도는 자바스크립트 엔진이 아주 작은 배열(크기 10)에 대해 어떻게 작동하는지만 테스트하려 했던 걸까? 그럴 수도 있겠지만 정말 그랬다면 미묘한 내부 구현 상세에 초점을 제대로 못 맞춘 건 아닌지 따져봐야 한다.

그게 아니라면, 이 테스트는 실제로 배열이 점점 더 커질 거란 가정을 깔고 있는 걸까? 덩치 큰 배열을 자바스크립트 엔진이 처리하는 로직이 실제 용도와 견주어보았을 때 의미있고 정확하다고 할 수 있을까?

- 그럼 이 테스트는 x.length, x.push(..)가 배열 x에 원소를 추가할 때 성능에 얼마나 영향을 끼치는 알아내는 게 목적인가? 그렇다, 유효한 테스트 항목인 것 같다. 그런데 코드를 보면 push(..)는 함수 호출이므로 당연히 [...] 접근보다 느릴 것이다. 아마 틀림없이 케이스 1, 2가 케이스 3보다는 빠르리라.

‘사과냐 오렌지냐’ 오류는 다음 예제도 마찬가지다.

---

// 케이스 1

```
var x = ["민준", "서연", "지후", "민서", "현우"];  
x.sort();
```

```
// 케이스 2
var x = ["민준", "서연", "지후", "민서", "현우"];
x.sort( function mySort(a,b){
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
} );
```

---

보아하니 mySort(..)라는 커스텀 비교기<sup>comparator</sup>가 기본 내장 비교기보다 얼마나 더 느린지 알아보는 테스트인 듯싶다. 하지만 mySort(..)가 인라인 함수 표현식인 까닭에 부적절한/가짜 테스트가 되어버렸다. 그래서 케이스 2는 커스텀 자바스크립트 함수뿐만 아니라 테스트 반복 시 새 함수 표현식도 함께 테스트한다.

인라인 함수 표현식을 따로 함수 선언으로 빼내어 테스트해보면 인라인 함수 표현식을 생성할 때보다 2 ~ 20% 더 느려진다는 걸 알 수 있다. 놀랍지 않은가?

인라인 함수 표현식의 생성 비용을 확인할 의도가 아니었다면 mySort(..) 선언부를 페이지 설정부에 넣고(테스트 설정부가 아니다. 테스트 사이클마다 쓸데없이 재선언을 하기 때문이다) 테스트 케이스에서 x.sort(mySort)로 명칭에 의한 참조를 하는 편이 더 개선된/적절한 테스트다.

‘사과나 오렌지냐’ 식 테스트 케이스에서 모호하게 작업을 회피하거나 가외의 작업을 붙이는 것도 문제다.

---

```
// 케이스 1
var x = [12,-14,0,3,18,0,2.9];
x.sort();
```

```
// 케이스 2
var x = [12,-14,0,3,18,0,2.9];
x.sort( function mySort(a,b){
    return a - b;
} );
```

---

아까 말했던 인라인 함수 표현식의 함정은 둘째 치고, 케이스 2의 mySort(...)는 인자가 문자열이면 당연히 실패하지만 여기서는 숫자밖에 없어서 잘 작동한다. 케이스 1은 에러가 나진 않지만 실제로 전혀 다른 방향으로 실행되고 결과 또한 다르다! 두 테스트 케이스의 결과가 다르니 당연히 전체 테스트 역시 무용지물이다!

그런데 결과가 다른 것보다 주목할 부분은, 내장 sort(...) 비교기는 mySort(...)에선 하지 않는 일(비교할 값을 문자열로 강제변환한 뒤 사전적 비교<sup>lexicographic comparison</sup>를 하는 등)를 더 수행한다는 사실이다. 그래서 케이스 2 결과(의미 상 더 정확하다)는 [-14, 0, 0, 2.9, 3, 12, 18]인 반면, 케이스 1 결과는 [-14, 0, 0, 12, 18, 2.9, 3]다.

케이스별로 실제 하는 일이 상이하니 잘못된 테스트고 그 결과 또한 모두 거짓말이다.

비슷한 함정이지만 더욱 미묘한 예제를 보자.

---

// 케이스 1

```
var x = false;
var y = x ? 1 : 2;
```

// 케이스 2

```
var x;
var y = x ? 1 : 2;
```

---

아직 불리언 값이 아닌(본 시리즈 '타입과 문법') 표현식 x를 ? : 연산자로 강제변환 시 성능에 미치는 영향을 알아보는 것이 테스트 목적이다. 케이스 2 강제변환 과정에서 가윗일을 하는 것으로 밝혀지면 OK다.

그럼 여기서 무엇이 미묘한 문제일까? 케이스 1은 x값을 세팅하지만 케이스 2는 아무 값도 세팅하지 않기 때문에 실제로 케이스 2가 하지 않는 일을 케이스 1은 하고 있다. 아무리 사소하다고 해도 잠재적인 왜곡을 제거하려면 다음과 같이 수정해야 맞다.

---

```
// 케이스 1
```

```
var x = false;
```

```
var y = x ? 1 : 2;
```

```
// 케이스 2
```

```
var x = undefined;
```

```
var y = x ? 1 : 2;
```

---

둘 다 할당문이 있으니 이제 당초 테스트하려 했던 바(x 강제변환의 성능 영향 평가)를 정확하게 분리하여 테스트할 수 있다.

## 6.4 좋은 테스트를 작성하려면

자, 이제 여러분에게 전하고 싶은 메시지를 정리하련다.

좋은 테스트를 작성하고 싶다면 두 테스트 케이스 사이의 차이점은 무엇인지, 그리고 그 차이점은 ‘의도적인지<sup>intentional</sup> 비의도적인지<sup>unintentional</sup>’ 철저히 분석하고 고민해야 한다.

의도적인 차이는 아무 문제가 없는 반면, 비의도적인 차이는 쉽게 발생하고 테스트 결과를 왜곡한다. 이렇게 테스트 결과가 예기치 않게 비틀리는 현상을 정말, 정말로 주의해서 방지해야 한다. 그리고 처음에 여러분이 의도한 차이가 다른 사람들 눈에는 명확하지 않아 테스트 자체를 부정확한 것으로 의심(또는 믿음)하게 만들 수 있다. 해결 방법은?

더 나은, 더 명확한 테스트를 작성하는 것이다. 정확한 테스트 의도가 무엇인지 미묘한 세부분까지 (jsPerf.com의 “Description” 필드와 코드 코멘트에) 시간을 들여 문서화하라. 의도적인 차이는 확실하게 공표하자. 나중에 다른 사람들, 심지어 여러분 스스로도 테스트 결과를 왜곡시킬 비의도적인 차이를 식별하는데 큰 도움이 된다.

테스트와 크게 상관없는 부분은 페이지나 테스트 설정부에 미리 선언하는 형태로 따로 빼내어 테스트에 포함시키지 않는다.

사소한 코드 조각에 집착하여 컨텍스트를 벗어나 한정적인 벤치마킹을 하기 보다 더 넓은 범위의 (물론 연관된) 컨텍스트로 포괄하면 더 나은 테스트/벤치마크를 할 수 있다. 이렇게 테스트하면 실행은 대개 더 느려지겠지만 컨텍스트와 더 밀접하게 연관된 차이점을 포착할 수 있을 것이다.

## 6.5 미시성능

5장에서 지나친 집착은 금물이라고 부정적인 언급을 했었던 미시성능 관련 이슈들을 잠시 살펴보고 가자.

성능 벤치마킹을 할 때는 우선 자신이 작성한 코드와 실제로 엔진이 실행하는 코드가 같지 않을 수도 있다는 사실을 당연하게 받아들여야 한다. 1장에서 컴파일러가 문의 위치를 재배치한다고 이야기한 바 있는데, 순서뿐 아니라 개발자의 코드와 전혀 내용이 다른 코드를 실행하는 경우도 있다.

다음 코드 조각을 보자.

---

```
var foo = 41;

(function(){
  (function(){
    (function(baz){
      var bar = foo + baz;
      // ..
    })(1);
  })();
})();
```

---

가장 안쪽 함수에서 foo 레퍼런스는 3단계 스코프 룩업을 하여 값을 참조할 것처

럼 보인다. 본 시리즈 ‘스코프와 클로저’에서 어휘 스코프의 작동 원리를 설명했는데, 컴파일러는 보통 록업을 캐시하기 때문에 다른 스코프에서 foo를 참조하더라도 실제로 추가 비용은 발생하지 않는다.

하지만 좀 더 깊이 생각해보자. foo가 참조할 코드가 사실 한 군데 밖에 없고 그나마 그 값이 상수 41이라는 사실을 컴파일러가 알고 있다면...?

컴파일러가 foo 변수를 아예 없애고 그 자리를 상수로 대체하지 않을까?

---

```
(function(){
  (function(){
    (function(baz){
      var bar = 41 + baz;
      // ..
    })(1);
  })();
})();
```



마찬가지로 baz 변수도 컴파일러가 바꿔치기 할 수 있다.

지금까지 여러분이 코딩한 자바스크립트 코드가 꼭 그렇게 기재해야 한다는 ‘요건’이라기보다 엔진이 무슨 일을 해야 할지 살짝 ‘귀뜸’해준 것에 불과하다는 사실을 알고 나면 사소한 구문 하나하나에 집착했던 자신의 지난 모습이 얼마나 부질 없는 행동이었는지 깨닫게 되리라.

다음 예제를 보자.

---

```
function factorial(n) {
  if (n < 2) return 1;
  return n * factorial( n - 1 );
}

factorial( 5 ); // 120
```

---

아, 예전에 많이 유행했던 고풍스런 팩토리얼 알고리즘이다! 자바스크립트 엔진이 이 코드를 있는 그대로 읽어 실행하리라 믿어 의심치 않는 독자도 있을 것 같다. 솔직히 장담은 못하겠지만 정말 그럴 가능성도 없지 않다.

실제로 이 예제를 C 언어로 코딩하여 고급 최적화 옵션을 주고 컴파일하면 컴파일러는 `factorial(5)` 호출을 상수값 120으로 대체하고 함수 정의부와 호출 코드를 통째로 날려버린다!

더욱이 어떤 엔진은 소스 코드의 재귀가 루프 하나로 더 쉽게(즉, 최적으로) 표현된다고 판단하면 폴림 재귀<sup>05</sup> `unrolling recursion`하는 버릇이 있다. 따라서 좀 전의 팩토리얼 재귀 코드는 임의로 자바스크립트 엔진이 다음과 같이 해석할 가능성이 있다.

---

```
function factorial(n) {
  if (n < 2) return 1;

  var res = 1;
  for (var i=n; i>1; i--) {
    res *= i;
  }
  return res;
}
```

```
factorial( 5 ); // 120
```

---

문득 `n * factorial( n - 1)`와 `n *= factorial(--n)` 중 어느 쪽이 더 빠른지 궁금하다. 직접 성능 벤치마킹을 해서 확인할 수도 있지만, 더 넓은 시야에서 보자면 자바스크립트 엔진이 언제라도 이처럼 폴림 재귀를 단행할 수 있기 때문에 소스 코드를 있는 그대로 실행하지 않을지도 모른다는 사실은 간과하기 쉽다.

`--n` 대 `n--` 문제는 `n--`이 이론적으로 어셈블리 수준에서 작업량이 더 적어 최적화에 유리하다고 알려져 있다.

---

<sup>05</sup> 역자주\_재귀 코드를 재귀 아닌 루프문으로 풀어버리는 것입니다.



이런 식의 강박증은 현대 자바스크립트에서는 헛소리에 불과하며 그냥 엔진이 알아서 처리하도록 놔두어야 할 문제다. 프로그래머는 가장 ‘논리에 맞게’ 코드를 작성하면 된다. for 루프가 포함된 다음 세 코드를 비교해보자.

---

// 옵션 1

```
for (var i=0; i<10; i++) {  
    console.log( i );  
}
```

// 옵션 2

```
for (var i=0; i<10; ++i) {  
    console.log( i );  
}
```

// 옵션 3

```
for (var i=-1; ++i<10; ) {  
    console.log( i );  
}
```

---

옵션 3은 for 루프에 전치 증가 연산자 ++i를 써서 i가 -1부터 시작하는 등 괜시리 혼란스럽기만 하다. 옵션 2, 3이 옵션 1보다 조금이라도 더 성능이 좋을거란 확신에 가득차 있는 이들에겐 미안하지만 개똥철학이다! 실제 옵션 1, 2의 성능 차이는 무의미하다.

자바스크립트 엔진이 ++를 발견하여 같은 로직으로 이와 동등한 ++i로 바꿀 가능성은 얼마든지 있기 때문에 둘 중 어느 것을 쓸까 고민하는 건 100% 시간 낭비다.

다음 코드 역시 어리석게도 미시성능에 집착한 사례다.

---

```
var x = [ .. ];
```

// 옵션 1

```
for (var i=0; i < x.length; i++) {  
    // ..  
}
```

```
// 옵션 2
for (var i=0, len = x.length; i < len; i++) {
    // ..
}
```

---

배열 x의 크기가 순회 도중 변경될 일은 없으니 변수 len에 캐시해두면 매번 x.length에서 프로퍼티를 찾느라 소모되는 비용을 아낄 수 있지 않겠냐는 주장이다.

그러나 막상 len에 캐시하는 경우(옵션 2)와 그냥 x.length를 사용한 경우(옵션 1) 성능을 벤치마킹하면 이론은 그럴싸하지만 실제 통계 수치는 전혀 상관이 없다.

오히려 v8 같은 일부 엔진에선 미리 배열 길이를 캐시하면 엔진이 자체 해결하도록 놔두는 것보다 성능이 떨어진다. 성능 최적화에 관해서라면 여러분이 자바스크립트 엔진을 능가할 수는 없으니 감히 꿈도 꾸지 말지어다.

### 6.5.1 똑같은 엔진은 없다

상이한 브라우저에 탑재된 각 자바스크립트 엔진은 모두 “스펙을 준수”하지만 코드 처리 방식은 제각각이다. 304 페이지 “꼬리 호출 최적화(TCO)”에서 설명할 ES6 “꼬리 호출 최적화”를 제외하면 자바스크립트 명세가 명시한 성능 요건 같은 건 없다.

작업에 따라 어느 정도 성능 가감은 불가피하지만 어떤 작업을 최적화 대상으로 삼을지는 엔진이 주관적으로 판단할 문제다. 모든 브라우저에서 언제나 더 빨리 실행되는 접근 방식은 찾아보기 어렵다.

자바스크립트 개발자 커뮤니티, 특히 노드JS 일을 하는 전문가들을 중심으로 v8 자바스크립트 엔진의 특정한 내부 구현 로직을 상세히 분석하고 v8 작동 원리를 최대한 활용할 수 있는 방향으로 자바스크립트를 코딩하는 문제가 연구되어 왔

다. 열정적인 그들 덕분에 실제로 놀랄 만한 수준의 성능 개선을 해왔고 기대 효과 역시 매우 높은 편이다.

v8에서 종종 언급되는 사례 몇 가지를 추려본다.

- 함수 간에 arguments 변수를 전달하지 말라. 누수 탓에 함수 구현체가 느려진다.
- try...catch 구문은 함수에서 떼어내라. 브라우저는 try..catch 구문이 포함된 함수를 최적화하려고 전력질주하므로 이 구문을 함수에 남겨두면 주변 코드는 최적화하고 반최적화<sup>deoptimization</sup>를 하여 불이익을 받을 소지가 있다.

하지만 팁 하나하나에 집중하지 말고 일반적인 관점에서 v8 전용 최적화 방식을 정상 테스트 해보자.

진정 여러분은 특정 자바스크립트 엔진에서만 실행할 코드를 개발하고 있는가? 여러분이 짠 코드가 ‘지금’은 노드JS에 딱 맞게 개발됐다 해도 ‘언젠가’ v8 엔진이 사용되리란 믿음을 앞으로도 계속 가져갈 수 있을까? 지금부터 수 년 후 언젠가 노드JS 아닌 서버형 자바스크립트 플랫폼에서 여러분이 작성했던 코드를 실행할 일이 결단코 없을까? 이전 버전에서는 최적화시켰던 방식이 새 엔진에서는 아주 느리게 실행된다면?

아니면 줄곧 v8에서만 운영해온 코드가 어느 순간부터 엔진의 일부 작동 방식이 변경되면서 이전에 빨랐던 코드가 느려지거나 반대로 느렸던 코드가 빨라지거나 한다면?

이런 시나리오는 단지 이론에 그치지 않는다. 예로부터 배열의 여러 문자열 원소 값을 하나로 붙일 땐 일일이 +로 추가하는 것보다 배열 join("") 메소드를 쓰는 게 더 빠르다고 구전되어 왔다. 그렇게 전해내려온 역사적 원인은 미묘한데, 어쨌

거나 문자열 값이 메모리에 저장/관리되는 구현 세부분과 연관이 있다.

결과적으로 당시 업계에 널리 보급된 최선의 지침은 언제나 `join(...)`을 애용하  
란 거였고 대부분 개발자들은 그대로 따랐다.

그러던 중 어느 시점부터 자바스크립트 엔진은 문자열을 다루는 내부 로직, 특히  
+로 붙이는 부분의 로직을 최적화하기 시작했다. 엔진 개발자들은 더 많이 알려진  
`join(...)` 자체의 속도는 떨어뜨리지 않으면서도 + 사용에 더 심혈을 기울인 것  
이다.



대개 일반적으로 통용되는 사용법에 근거한 특정 접근 방식을 표준화/최적화하는 작  
업을 종종 (비유적으로) “소가 낸 길을 닦는다(paving the cowpath)”고 한다.

새로운 문자열 붙이기 방법이 일단 정착되고 나니 불행히도 `join(...)`을 애용했  
던 모든 코드는 비최적화(suboptimal)되었다.<sup>06</sup>

비슷한 예가 또 있다. 한때 오페라 브라우저는 원시값을 감싸는 객체의 박싱(boxing/  
언박싱(unboxing)<sup>07</sup> (본 시리즈 ‘타입과 문법’ 참고) 처리 방법이 다른 브라우저와 달랐다.  
그래서 당시 오페라 측은 개발자들에게 `length` 같은 프로퍼티나 `charAt(...)`  
메소드로 문자열 접근 시 원시값 대신 String 객체를 사용할 것을 권고했었다.  
오페라 최적화 로직은 문자열 원시값에만 해당하고 객체에 상응하는 감싸미와는  
무관했기 때문에 당대의 다른 주요 브라우저와는 정반대로 이 권고 사항을 따르는  
게 최선이었다.

지금은 그러한 위험 요소가 없을 것 같지만 내 생각엔 현재 코드에도 남아있다. 그러

<sup>06</sup> 역자주\_최적화(1등) 아닌, 2등, 3등, ...의 최적화 상태를 말합니다. 즉, 진짜 최적화가 가능성에도 불구하고 차  
선택에 머무는 것입니다.

<sup>07</sup> 역자주\_자바스크립트에서 원시값은 스택(stack) 영역에, 객체는 힙(heap)에 저장됩니다. 박싱(boxing)은 원  
시값을 객체 타입으로 변환하여 스택에 있는 데이터를 힙으로 복사하는 행위를, 언박싱(unboxing)은 그 반대  
과정을 말합니다.

므로 순전히 엔진 구현 상세에 입각해서 (특정 엔진에만 해당하는 상세라면) 자바스크립트 코드를 광범위하게 성능 최적화하는 일은 대단히 조심스러울 수밖에 없다.

그 반대로 조심스럽긴 마찬가지다. 특정 엔진에서 만족스러운 성능이 나오지 않는다고 작성한 코드를 굳이 뜯어고칠 필요는 없다.

역사적으로 IE는 항상 이런 논란의 중심에 있었다. 동시대의 다른 주요 브라우저에서는 별 문제없던 성능 이슈로 고군분투하던 구버전 IE는 참으로 별별 시나리오를 다 연출했다. 방금 전 살펴봤던 문자열 붙이기 문제는 IE6, IE7 시절 진짜 큰 골칫거리였고 +보다 `join(...)`을 쓰는 편이 성능면에서 정말 유리했었다.

그러나 단 하나의 브라우저에서 불거진 성능 문제를 보고 다른 브라우저에서는 비최적 접근 방식을 사용하라고 권장하는 태도는 문제가 있다. 심지어 여러분 고객들이 대부분 사용하는 브라우저의 시장 점유율<sup>08</sup>이 매우 크다고 해도 일단 개발자는 올바른 코드를 작성하고 나중에 브라우저 업데이트 시 성능 개선이 이루어지길 기대하는 편이 현실적이다.

“임시로 넣은 꼼수보다 더 영속적인 건 없다.” 지금 어떤 성능 문제를 황급히 해결하고자 여러분이 짜넣은 코드가 브라우저 자체 성능 버그보다 훨씬 더 오래 존속될지도 모른다.

브라우저가 5년에 한번씩 업데이트되던 시절에는 결정하기가 당혹스러웠을 것이다. 하지만 지금은 전반적으로 브라우저의 업데이트 주기가 (모바일 세상은 아직도 더 더지만) 훨씬 짧아졌고 웹 기능을 더욱 더 최적화하기 위해 브라우저 간 무한 경쟁이 벌어지고 있다.

혹시라도 여러분이 다른 브라우저엔 없는 성능 버그를 특정 브라우저에서 발견하면 가능한 모든 수단을 다 동원해서 해당 업체에 정식으로 문제를 제기하기 바란

---

<sup>08</sup> 역사주\_2015년 2월 발표된 한국인터넷진흥원(KISA) 조사 결과에 의하면 국내 PC 웹 브라우저는 IE가 87.5%, 모바일은 안드로이드 브라우저가 73.61%를 차지하고 있습니다.

다. 대다수 브라우저가 이런 의도로 공개 버그 추적 시스템을 구비해놓았다.



내 생각엔 브라우저 성능 문제는 그냥 짜증나거나 성가신 정도 말고 정말 눈에 확 띄는 현상일 때에만 해결 방안을 고민할 것을 권한다. 그리고 나라면 어떤 성능 목표를 적용하기 전에 다른 브라우저에서 알려진 부작용이 있진 않은지 아주 꼼꼼히 체크할 것이다.

## 6.5.2 큰 그림

여러분은 미묘한 미세성능에 사로잡히지 말고 최적화의 큰 그림을 볼 수 있어야 한다.

큰 그림인지 아닌지는 어떻게 구분할까? 임계 경로에서 실행되는 코드인지 먼저 확인하라. 임계 경로에 있지 않은 코드면 최적화해도 별 효과가 없을 가능성이 높다.

“너무 이른 최적화<sup>premature optimization</sup>다!” 하는 말을 들어본 적 있는가? 도날드 커누스<sup>Donald Knuth<sup>09</sup></sup>의 유명한 어록(“너무 이른 최적화는 만병의 근원이다.<sup>premature optimization is the root of all evil.</sup>”)에서 나온 말이다. 많은 개발자들이 이 말을 인용하면서 대부분의 최적화가 “시기상조”이고 쓸데없는 짓이라고 한다. 진실은 늘 그렇듯 그렇게 단순하지 않은 않다. 커누스가 한 말을 전문 인용한다.

프로그래머들은 자신이 짠 코드 중 비임계<sup>noncritical</sup> 경로의 실행 속도를 신경쓰고 걱정하느라 많은 시간을 낭비하는 경향이 있는데 디버깅/유지 보수 관점에서 이런 식으로 효율을 높여보고자 하는 행위는 매우 부정적인 영향을 끼친다. 97%의 자잘한 효율은 잊어버리라. 너무 이른 최적화는 만병의 근원이다. 하지만 3% 임계 경로에 대해서는 최적화를 미루면 안 된다.

- Computing Surveys 6  
(1974년 12월)

<sup>09</sup> 역자주\_ 미국의 저명한 컴퓨터 과학자이며, 현재 스탠퍼드 대학교 명예교수입니다. 컴퓨터 과학 분야의 가장 권위있는 도서, 컴퓨터 프로그래밍의 예술(The Art of Computer Programming)의 저자입니다.

결국 그가 한 말은 “비임계 경로 최적화는 만병의 근원이다.”라고 살짝 바꾸어도 같은 뜻이다. 임계 경로에 있는 코드를 밝혀내서 이를 최적화해야 하는 것이 포인트다!

내가 한 마디 덧붙이자면, 임계 경로 최적화는 비록 성과가 보잘 것 없아해도 투자한 시간이 낭비되는 법이 없지만 비임계 경로 최적화는 그 결과가 아무리 좋아도 정당화하기 어렵다.

여러분이 작성한 코드가 계속 임계 경로에서 반복, 또 반복 실행되는 “뜨거운” 코드 조각이거나, 애니메이션 루프 또는 CSS 스타일 업데이트처럼 사용자들이 보는 중요한 UX라면 나름대로 의미있는 최적화를 해보려는 노력을 아끼지 말자.

예컨대, 임계 경로에 위치한, ‘문자열 → 숫자’로 강제변환하는 애니메이션 루프를 생각해보자. 다음 구현 옵션(본 시리즈 ‘타입과 문법’ 참고) 중 어느 것이 가장 빠를까?

---

```
var x = "42"; // 숫자 '42'가 필요하다.
```

```
// 옵션 1: 자동으로 암시적 강제변환이 일어나게 한다.
```

```
var y = x / 2;
```

```
// 옵션 2: 'parseInt(...)'를 쓴다.
```

```
var y = parseInt( x, 0 ) / 2;
```

```
// 옵션 3: 'Number(...)'를 쓴다.
```

```
var y = Number( x ) / 2;
```

```
// 옵션 4: '+' 단항 연산자를 쓴다.
```

```
var y = +x / 2;
```

```
// 옵션 5: '|' 단항 연산자를 쓴다.
```

```
var y = (x | 0) / 2;
```

---



옵션들 간의 미세한 성능 차이를 알아보고 싶은 독자는 직접 테스트를 작성하여 한번 실행해보라. 연습 문제로 남긴다.

“어느 하나는 다른 것들과 다르겠지” 하는 생각으로 5개 옵션을 죽 살펴보면 먼저 `parseInt(...)`가 하는 일은 강제변환이 아닌, 문자열 파싱이므로 작업량이 더 많고 느릴 수 밖에 없다. 일단 제외!

물론, `x`가 “42px”(CSS 스타일 코드)처럼 파싱이 필요한 값이면 실제로는 `parseInt(...)`가 가장 타당한 옵션이 될 것이다!

`Number(...)` 역시 함수 호출이다. 작동 관점에서 보면 + 단항 연산자를 쓴 옵션 4와 정확히 같지만, 함수 실행에 따른 절차가 더 많아서 실제로 약간 느려진다. 물론, 자바스크립트 엔진이 작동 방식의 대칭성을 자동 인지하여 `Number(...)` 코드를 (+x 식으로) 인라인 처리할 가능성도 있다.

하지만 기억하시길! ‘+x 대 x | 0’ 성능 비교에 집착하는 건 어리석은 시간 낭비다. 미세성능 이슈니까 그냥 넘어가고 프로그램 가독성을 해치는 일은 하지 말자.

프로그램 임계 경로에서 성능이 매우 중요한 건 맞지만 그렇다고 유일무이한 요소는 아니다. 성능이 얼추 비슷한 옵션들이 있다면 가독성은 그 다음으로 중요한 관심사로 다루어야 한다.

## 6.6 꼬리 호출 최적화(TCO)

ES6부터는 특정한 성능 요건이 한 가지 추가되었다. 함수 호출과 관련된 특수한 형태의 최적화, 즉 ‘꼬리 호출 최적화(TCO)<sup>tail call optimization</sup>’에 관한 내용이다.<sup>10</sup>

<sup>10</sup> 역자주\_꼬리 호출 최적화를 비롯한 알고리즘에 대해 기초 지식이 부족한 독자 여러분은 ‘뇌를 자극하는 알고리즘(박상현 저, 한빛미디어 2009)’을 먼저 읽어보시기 바랍니다.



꼬리 호출은 함수 호출부가 다른 함수의 “꼬리” 부분에 있고, 호출이 끝나면 (결괏값 반환을 제외하고) 더 이상 수행할 작업을 남기지 않는 방식이다.

꼬리 호출과 관련하여 설정 예제를 보자.

---

```
function foo(x) {
  return x;
}

function bar(y) {
  return foo( y + 1 ); // 꼬리 호출
}

function baz() {
  return 1 + bar( 40 ); // 꼬리 호출 아님
}

baz(); // 42
```

---

`foo(y + 1)`가 바로 꼬리 호출이다. `foo(..)`가 끝나면 `bar(..)`도 함께 끝나고 `foo(..)` 호출의 결괏값을 반환하면 되기 때문이다. 그러나 `bar(40)`은 완료 후 그 결괏값을 `baz()`가 반환하기 전 1을 더해야 하므로 꼬리 호출이 아니다.

쉽게 말해서 새 함수를 호출하려면 스택 프레임<sup>stack frame</sup>이라는 호출 스택을 쌓기 위해 별도의 메모리 할당이 필요하다. 이 예제만 해도 `baz()`, `bar(..)`, `foo(..)` 모두 한번 호출할 때마다 스택 프레임이 소모된다.

그러나 TCO 능력을 갖춘 엔진은 `foo(y + 1)`가 꼬리 위치<sup>tail position</sup>에서 호출된다는 사실을 알고 있어서 기본적으로 `bar(..)`가 끝난 뒤 `foo(..)`를 부를 때 새로운 스택 프레임을 생성하지 않고 기존 `bar(..)`의 스택 프레임을 재사용한다. 속도도 빠르지만 메모리도 덜 쓰는 일석이조의 효과가 있다.

짧은 코드에선 대수롭지 않은 최적화일지 모르지만 수백 ~ 수천 스택 프레임이 소

요되는 재귀라면 현저한 효과를 기대할 수 있다. TCO 기능 덕택에 모든 호출을 스택 프레임 하나로 처리할 수 있다!

자바스크립트 세계에서 재귀는 난제다. TCO가 없는 세상이라면 메모리가 바닥 나서 프로그램이 몇기 전 엔진은 임의로(나름대로) 재귀 스택 깊이를 제한할 수밖에 없기 때문이다. TCO가 있으면 메모리를 더 쓸 일이 없으니 꼬리 위치 호출에 해당하는 재귀 함수는 일단 무제한 실행이 가능하다.

`factorial(..)` 재귀 예제를 TCO 식으로 다시 작성해보자.

---

```
function factorial(n) {  
  function fact(n,res) {  
    if (n < 2) return res;  
  
    return fact( n - 1, n * res );  
  }  
  
  return fact( n, 1 );  
}  
  
factorial( 5 ); // 120
```

---

고친 `factorial(..)` 역시 재귀적으로 작동하지만 안쪽의 두 `fact(..)` 호출이 꼬리 위치에 있으므로 TCO로 최적화할 수 있다.



TCO는 진짜 꼬리 호출일 때에만 효과가 있다는 점을 유의하자. 꼬리 호출 없는 재귀 함수는 여전히 일반적인 스택 프레임 할당에 의존할 수 밖에 없고 엔진은 임의로 재귀 스택에 제한을 가할 것이다. 웬만한 재귀 함수는 `factorial(..)`처럼 재작성할 수 있지만 세부적인 로직은 잘 보고 구현해야 한다.

엔진 개발자에게 일임하지 않고 ES6 명세로 TCO 구현 요건을 규정한 건 TCO가 없을 경우 실제로 자바스크립트 개발자들이 호출 스택 한계를 노심초사한 나머지 재귀 알고리즘을 안 쓰려고 하는 경향이 있기 때문이다.

TCO 기능이 빠진 엔진은 어쨌든 성능이 더 떨어질 수 밖에 없으므로 ES6 명세서에 ‘요건’으로 박아넣은 것도 무리는 아닌 듯싶다. 하지만 TCO가 없으면 실제로 무용지물이 되어버리는 프로그램도 있기 때문에 단지 숨겨진 구현 상세라기 보다는 언어 상의 중요한 기능으로 봐야할 것 같다.

ES6가 보증하는 기능이니 이제 자바스크립트 개발자들은 모든 ES6+ 호환 브라우저에서 안심하고 이 최적화에 의지할 수 있게 되었다. 자바스크립트 성능을 위하여 건배!

## 6.7 정리하기

코드 조각을 효과적으로 벤치마킹하려면, 특히 같은 코드에 대해 여러 가지 옵션이 있고 그 중 어느 것이 더 빠르니 결정해야 하는 상황이라면 일단 자세한 내막을 살펴야 한다.

여러분이 통계적으로 옳다고 여기는 벤치마킹 로직을 들이대지 말고 골치아픈 일은 Benchmark.js가 대신 하도록 위탁하라. 이 때 겉으로 보기엔 합리적인 것 같지만 심각한 결함이 내재된 테스트를 작성하기 쉬우니 각별히 주의하자. 사소한 차이가 완전히 믿을 수 없는 테스트 결과로 왜곡시키는 일이 다반사다.

하드웨어/디바이스에 따른 편향<sup>bias</sup>을 방지하려면 되도록 다양한 환경에서 많은 테스트 결과를 수집하는 것이 관건이다. jsPerf.com은 테스트 실행 후 측정된 성능 결과를 수집하는 아주 훌륭한 웹 사이트다.

‘x++이냐 ++x이냐’ 식의 대수롭지 않은 미세성능의 세부면에 집착하는 성능 테스트가 아직도 많다. 좋은 테스트를 작성하려면 임계 경로의 최적화 같은, 넓은 시야에서 고찰한 주제에 집중해야 하며 엔진마다 제각각인 구현 상세의 늪에 빠지지 않도록 조심해야 한다.

꼬리 호출 최적화(TCO)는 ES6부터 필수 최적화 구현 항목으로 자리잡았고 자바 스크립트에서 다른 방법으로 불가능한 일부 재귀 패턴을 가능하게 한다. TCO를 이용하면 실행할 다른 함수의 꼬리 부분에서 리소스를 추가하지 않고도 함수를 호출할 수 있으며, 엔진 입장에서는 더 이상 재귀 알고리즘에 쓸 호출 스택 깊이를 제한할 필요가 없다.



# 부록 A

## asynquence 라이브러리

1, 2장에서 비동기 프로그래밍의 일반적인 패턴과 콜백을 응용한 해법에 대해 자세히 설명했다. 이어서 3, 4장에서는 제한된 기능의 콜백에 비해 더 견고하고 미더우면서 추론적인 비동기성 기반을 제공하는 프라미스, 제너레이터를 이야기했다.

그러면서 내가 만든 비동기 라이브러리 asynquence(“async(비동기)” + “sequence(시퀀스)” = “asynquence”)(<https://github.com/getify/asynquence>)를 몇 차례 언급했었는데, 이제 이 라이브러리의 작동 원리를 간략히 살펴보고 특유의 설계 개념이 중요한/유용한 이유를 설명한다.

부록 B에서는 고급 비동기 패턴을 몇 가지 소개할 예정이다. 하지만 그 전에 그러한 패턴을 잘 활용할 수 있도록 도와줄 라이브러가 필요한데, 나는 asynquence를 이용하여 이들을 표현하고자 한다. 일단 어떻게 생긴 라이브러리인지 먼저 알아야 하니 조금만 시간을 들여 학습하자.

물론, asynquence가 유일한 옵션은 아니고 비동기 코딩을 지원하는 다른 괜찮은 라이브러리도 많다. 하지만 asynquence는 최선의 패턴을 모두 한 라이브러리에 집약시켜 독특한 관점에서 바라보게 해주며, 무엇보다 (비동기) 시퀀스라는, 하나의 근본적인 추상화를 기반으로 한다.

정교한 자바스크립트 프로그램일수록 다양한 비동기 패턴이 한데 어우러지기 마련인데, 나의 전제는 대개 이런 문제를 개발자가 직접 고민해서 해결 방안을 찾아

내야 한다는 점이다. `asyncquence`는 비동기성을 각기 다른 관점에서 바라본 상이한 비동기 라이브러리를 여럿 쓰는 대신, 핵심 라이브러리 하나만 사용법을 익혀 배포할 수 있도록 다수의 비동기 라이브러리를 변형된 시퀀스 단계로 통합한다.

`asyncquence`로 비동기 흐름 제어 프로그램을 짜면 프라미스다운 의미를 쉽게 부여할 수 있어 학습할 가치는 충분하다고 본다. `asyncquence`를 이렇게 부록으로 나눈 이유도 그 때문이다.

먼저 `asyncquence`의 내부 설계 원리를 살펴보고 이어서 예제 코드와 함께 API 사용법을 알아보자.

## A.1 시퀀스, 추상화 설계

`asyncquence`는 일련의 동기적/비동기적 작업 단계들을 집합적으로 ‘시퀀스 *sequence*’라고 간주한다. 즉, 시퀀스는 작업을 담는 그릇이고 작업을 마치는데 필요한 (비동기적인) 개별 단계들로 구성된다. `asyncquence` 학습의 출발점은 이와 같은 근본적인 추상화를 이해하는 것이다.

시퀀스 각 단계는 내부에서 프라미스가 제어한다. 즉, 시퀀스에 단계를 붙일 때마다 암시적으로 프라미스가 생성되어 시퀀스 끝에 추가된다. 프라미스의 정의 덕분에 시퀀스 단계 하나 하나는 동기적으로 끝나도 반드시 비동기적으로 진행된다.

또한 시퀀스 단계의 진행은 항상 선형적(1단계 → 2단계 → ...)이다. 기존 시퀀스에서 새로운 시퀀스가 분기할 수도 있다. 즉, 메인 시퀀스 진행 중 어느 지점에서 갈라져 나오는 것이다. 또 시퀀스는 특정 위치에서 한 시퀀스를 다른 시퀀스에 병합하는 등 다양한 방법으로 합치는 것도 가능하다.

시퀀스는 프라미스 연쇄와 비슷하지만 프라미스 연쇄는 전체 연쇄 참조를 그러칠

“손잡이”가 없다. 프라미스 연쇄에서는 현재 단계, 그리고 이를 대기하는 다른 단계만 참조할 수 있다. 연쇄의 첫 번째 프라미스를 가리킬 방도가 없으면 전체 연쇄 참조 또한 불가능하다.

전체 시퀀스를 집합적으로 참조할 수 있는 레퍼런스가 필요할 때가 종종 있다. 특히, 시퀀스를 중지/취소하는 경우가 그렇다. 3장에서 배웠듯이 프라미스는 외부 불변성<sup>external immutability</sup>이라는 설계 원리와 배치되기 때문에 절대로 취소할 수 없다.

시퀀스는 불변값 개념이 필요한 미래값을 담는 그릇으로 전달되는 형태가 아니라서 불변성 설계 원리 같은 건 없다. 시퀀스는 중지/취소가 가능하도록 적절한 수준으로 추상화한 것이다. `asynquence` 시퀀스는 언제든지 멈출 수 있고 일단 중지되면 그 자리에 못박혀 더 이상 나아가지 않는다.

흐름 제어 장치로서 프라미스 연쇄보다 시퀀스가 더 나은 추상화인 이유는,

첫째, 프라미스 연쇄는 단조로운 수작업 공정이라(프로그램 전반에 걸쳐 프라미스를 생성/연쇄하는 일은 단조로운 편이라 지루해진다) 정작 필요한 곳에 개발자가 프라미스를 사용하려고 할 때 역효과가 날 수 있다.

추상화는 관용 코드와 지리한 작업을 줄이기 위한 장치라는 점에서, 시퀀스 추상화는 훌륭한 솔루션이라 할 만하다. 프라미스는 개별 단계에 집중하고 연쇄를 계속 이어간다는 전제가 없는 반면, 시퀀스는 더 많은 단계가 무제한 추가될거라 가정한다.

추상화의 복잡도가 줄면 (`race([...])` 및 `all([...])` 이상의) 고계<sup>higher-order</sup> 프라미스 패턴에서 특히 강력해진다.

예를 들면, 결과를 반드시 성공 처리하는(의도했던 성공 귀결이든, 아니면 에러를 붙잡아 에러가 아니라고 긍정적 신호를 보내든 간에) `try...catch`와 비슷한 개념으로 시퀀스 도



중의 단계를 표현하고 싶을 때가 있다. 아니면, 성공할 때까지 몇번이고 같은 단계를 반복 시도하는 `retry/until` 루프처럼 나타내야 할 경우도 생긴다.

이런 유형의 추상화는 프라미스 원시값만 갖고 나타내기엔 비중이 있는데다, 기존 프라미스 연쇄 중간에서 그렇게 하는 건 별로 예쁘지 않다. 그러나 의도한 바를 하나의 시퀀스로 추상화하고 단계는 프라미스로 감싸면 세부에 얽매이지 않고 합리적인 흐름 제어를 생각할 수 있는 자유를 얻게 될 것이다.

둘째, 더 중요한 문제로서, 비동기 흐름 제어를 하나의 시퀀스 단계로 바라보면 단계마다 어떤 형태의 비동기성이 연관되는지 그 자세한 내용을 추상화할 수 있다. 뚜껑 안에서 프라미스는 항상 단계를 제어하고 있지만 뚜껑 밖에서는 해당 단계가 진행 콜백(continuation callback (기본 단순 형태), 진짜 프라미스, 아니면 완전-실행 제너레이터처럼 생겼을지도 모른다. 어떤 그림인지 그려지는가?

셋째, 시퀀스는 이벤트식, 스트림식, 반응형 코딩 같은 이질적인 사고 체계에 쉽게 녹일 수 있다. `asynquence`는 내가 반응형 시퀀스(reactive sequences (잠시 후 다룬다))라 부르는, `RxJS`(반응형 확장판(Reactive Extensions))의 '반응형 옵저버블(reactive observable)'을 변형한 패턴을 제공하므로 반복적 이벤트 발생 시 매번 새 시퀀스 인스턴스를 생성할 수 있다. 프라미스는 일회용이라 반복적인 비동기성을 표현하기가 적절치 않다.

다른 대안이라면, 내가 이터러블 시퀀스(iterable sequences)라 명명한 패턴에서 귀결/제어 능력을 역전시키는 사고 방식이다. 각 단계가 내부적으로 자신의 완료 상태를 제어하는 (그래서 시퀀스를 진행하는) 대신 외부 이터레이터가 진행 과정을 통제하도록 시퀀스를 역전시켜 이터레이터가 `next(...)`하면 이터러블 시퀀스의 각 단계는 그냥 반응하는 것이다.

지금까지 언급한 변형에 대해서는 부록 A의 나머지 부분에서 설명할 테니 너무 많은 것들이 한꺼번에 쏟아져나와 주눅들진 말자.

요컨대, 시퀀스가 복잡한 비동기성을 다루는데 있어서 프라미스(프라미스 연쇄), 제

너레이터보다 강력하고 합리적인 추상화며, `asyncquence`는 더 이해하기 쉽고 즐거운 비동기 프로그래밍을 위한, 적절한 수준의 추상화를 제공할 목적으로 고안되었다.

## A.2 `asyncquence` API

우선 `ASQ(...)` 함수로 시퀀스(`asyncquence` 인스턴스)를 생성하자. 파라미터 없이 `ASQ()`를 호출하면 빈 초기 시퀀스가 만들어지는데, 하나 또는 그 이상의 값/함수를 `ASQ(...)`에 전달하면 시퀀스 단계를 해당 인자로 초기화한 시퀀스가 생성된다.



이 책에서는 코드 예시를 위해 브라우저 전역 범위의 최상위 `asyncquence` 식별자를 `ASQ`로 표기한다. 모듈 시스템(브라우저나 서버)을 통해 `asyncquence`를 포함/사용할 생각이라면 여러분이 좋아하는 심볼로 바꿔도 된다. `asyncquence`는 전혀 개의치 않는다!

부록의 API 메소드 대부분은 `asyncquence` 코어로 구현되어 있지만 선택적인 “이바지<sup>contrib</sup>”<sup>01</sup> 플러그인 패키지를 통해 제공되는 것들도 있다. 내장 메소드인지 플러그인 메소드인지는 [asyncquence 문서](#)를 참고하자.

### A.2.1 단계

시퀀스의 일반 단계를 나타내는 함수는 첫 번째 파라미터 자리에 진행 콜백, 두 번째 파라미터 이후에는 이전 단계가 전달한 메시지를 넣고 호출한다. 단계가 완료되면 진행 콜백을 호출하고 처음에 전달한 인자를 시퀀스 다음 단계로 흘려보낸다.

일반 단계는 `then(...)` 함수를 호출하여 시퀀스에 추가한다(실은 `ASQ(...)` 호출과 정확히 같다).

<sup>01</sup> 역자주\_이바지(`contrib`) 패키지만 자체로는 무료지만 무료 아님(`non-free`) 패키지에 의존하고 있는 패키지를 말합니다.

---

```

ASQ(
  // 1 단계
  function(done){
    setTimeout( function(){
      done( "Hello" );
    }, 100 );
  },
  // 2 단계
  function(done,greeting) {
    setTimeout( function(){
      done( greeting + " World" );
    }, 100 );
  }
)
// 3 단계
.then( function(done,msg){
  setTimeout( function(){
    done( msg.toUpperCase() );
  }, 100 );
} )
// 4 단계
.then( function(done,msg){
  console.log( msg ); // Hello World
} );

```

---



이 예제에서 `then(...)`은 내장 프라미스 API 함수와 이름은 같지만 전혀 다른 함수다. 원하는 만큼 함수/값을 인자로 전달할 수 있고 각자 개별 단계로 간주된다. 이름/버림 콜백 같은 건 없다.

한 프라미스를 다음 프라미스와 연쇄하고 `then(...)` 이름 처리기에서 프라미스를 반환하는 프라미스 체계와는 달리 `asyncquence`는 진행 콜백(나는 늘 `done()`이라고 하지만 뭐라 불러도 상관없다)을 호출하면 끝이고 필요 시 완료 메시지를 인자로 전달할 수 있다.

`then(...)`으로 정의한 단계는 모두 비동기적이라고 본다. 동기적 단계는 `done(...)`을 바로 호출하거나, 더 단순하게는 `val(...)` 헬퍼를 쓰면 된다.

---

```

// 1 단계 (동기)
ASQ( function(done){
    done( "Hello" ); // 수동으로 동기화한다.
} )
// 2 단계 (동기)
.val( function(greeting){
    return greeting + " World";
} )
// 3 단계 (비동기)
.then( function(done,msg){
    setTimeout( function(){
        done( msg.toUpperCase() );
    }, 100 );
} )
// 4 단계 (동기)
.val( function(msg){
    console.log( msg );
} );

```

---

보다시피 `val (..)`로 단계를 시작하면 나머지 로직은 개발자 몫으로 보고 진행 콜백을 따로 받지 않기 때문에 결과적으로 파라미터 목록이 깔끔해진다. 다음 단계로 메시지를 보내려면 그냥 `return` 한다.

`val (..)`은 로깅처럼 동기적인 값을 처리하는 작업에 유용한, “값 전용<sup>value-only</sup>” 동기 단계를 표현하는 함수라고 생각하면 된다.

## A.2.2 예러

프라미스와 `asynquence`의 가장 큰 차이점은 예러 처리다.

프라미스는 연쇄의 각 프라미스(단계)가 독립적으로 예러를 갖고 그 다음 단계에서 예러를 처리할(또는 처리하지 않을) 수 있다. 전체 연쇄(시퀀스)가 아닌, 개별 프라미스에 초점을 두기 때문이다.

시퀀스 어딘가에서 발생한 예러는 대개 복구되지 않으므로 그 다음 단계에서 불필

요하게 고민하지 말고 그냥 건너뛰는 게 맞다고 본다. 따라서 시퀀스는 어떤 단계에서 에러가 발생하면 전체 시퀀스를 에러 모드로 바꾸고 나머지 일반 단계는 실패 무시한다.

에러를 복구할 수 있는 단계가 있어야 한다면 (`try...catch`와 비슷한 `try(...)`나 `until(...)`(성공할 때까지, 또는 수동으로 `break()`할 때까지 계속 단계를 재시도하는 루프) 같은 API 메소드를 쓰면 된다. `asynquence`는 프라미스 `then(...)`, `catch(...)`와 동일한 `pThen(...)`, `pCatch(...)` 메소드를 지원하므로 시퀀스 도중 국지적인 에러 처리 또한 가능하다.

이렇듯 옵션은 두 가지 있지만, 내 경험 상 `asynquence` 기본 옵션만으로도 충분하다. 프라미스는 에러가 났을 때 모든 단계를 무시하려고 연쇄 참조를 할 때 버림 처리기가 등록된 단계가 하나라도 있으면 안 되므로 각별히 신경을 써야 한다. 그렇지 않으면 에러 처리 중 증발하여 시퀀스는 아무 일 없다는 듯 (의도했던 바와 달리) 그대로 진행할 가능성이 있기 때문이다. 이런 식으로 무뎌지게 흘러가면 제대로, 미답게 처리하기 어렵다.

`asynquence`에는 `or(...)`라는 시퀀스 메소드(`onerror(...)`라고도 한다)가 있어서 시퀀스 에러 알림 처리기를 등록할 수 있다. 이 메소드는 시퀀스 어디에서든 호출 가능하며 등록 개수에 제한은 없다. 서로 다른 여러 컨슈머가 시퀀스를 리스닝하면서 실패 여부를 쉽게 확인할 수 있게 해준다. 말하자면 일종의 에러 이벤트 처리기 같은 것이다.

자바스크립트 예외는 프라미스처럼 시퀀스 에러가 되거나, 혹은 프로그램 코드로 시퀀스 에러를 알릴 수도 있다.

---

```
var sq = ASQ( function(done){
  setTimeout( function(){
    // 시퀀스에 에러를 알린다.
    done.fail( "허격" );
```

```

    }, 100 );
  } )
  .then( function(done){
    // 여기는 실행되지 않는다.
  } )
  .or( function(err){
    console.log( err ); // 허걱
  } )
  .then( function(done){
    // 여기도 실행되지 않는다.
  } );

// 나중에

sq.or( function(err){
  console.log( err ); // 허걱
} );

```

---

미처리 예외(unhandled exception)에 관한 기본 로직 역시 asynquence와 프라미스 사이의 특징할 만한 차이점이다. 3장에서 자세히 살펴보았던 것처럼, 버림 처리기가 등록되지 않은 상태에서 버려진 프라미스는 그냥 조용히 에러를 손에 쥐고 있으므로(다른 말로, 삼켜버리므로) 연쇄의 마지막은 반드시 `catch (..)`로 끝나야 한다.

asynquence는 정반대다.

등록된 에러 처리기가 하나도 없을 때 시퀀스 에러가 나면 콘솔에 에러가 표시된다. 즉, 미처리 버림이 묵살되거나 소실되지 않도록 필히 알려준다.

이러한 알림 기능은 시퀀스에 에러 처리기를 등록하면 바로 해제되어 쓸데없는 노이즈를 방지할 수 있다.

처리기를 미처 등록하기도 전에 생성한 시퀀스가 에러 상태에 빠지는 경우도 생각해야 한다. 드물기는 하지만 가끔 그럴 때가 있다.

시퀀스에 `defer ( )`를 호출하면 에러 알림 대상에서 해당 시퀀스 인스턴스를 배제

할 수 있다. 나중에 어차피 반드시 처리할 예러라면 이렇게 예러 알림 기능을 꺼두는 게 현명할 것이다.

---

```
var sq1 = ASQ( function(done){
    doesnt.Exist(); // 콘솔에 예외를 던질 것이다.
} );

var sq2 = ASQ( function(done){
    doesnt.Exist(); // 시퀀스 예러만 던질 것이다.
} )
// 예러 알림 기능을 끈다.
.defer();

setTimeout( function(){
    sq1.or( function(err){
        console.log( err ); // ReferenceError
    } );

    sq2.or( function(err){
        console.log( err ); // ReferenceError
    } );
}, 100 );

// ReferenceError (sq1에서)
```

---

이와 같은 예러 처리 로직은 실패의 구덩이가 아닌, 성공의 구덩이라는 점에서 프라미스보다 낫다.



시퀀스 A가 시퀀스 B로 흘러든(포함된) 이후 시퀀스 A의 예러 알림을 끄면 시퀀스 B의 예러 알림은 어떻게 될지도 반드시 고려해야 한다.

### A.2.3 병렬 단계

한 가지 (비동기) 작업만 수행하는 시퀀스 단계만 있는 건 아니다. 여러 단계를 (동시에) 병렬로 처리해야 하는 단계도 있다. 여러 하위 단계를 동시 처리하는 시퀀

스 단계를 `gate(..)`라고 하며 (`all(..)`이란 별칭도 있다) 내장 프라미스 메소드 `Promise.all([..])`에 해당한다.

`gate(..)`의 모든 단계를 무사히 마치면 성공 메시지는 다음 시퀀스 단계로 넘어간다. 한 단계라도 에러가 나면 전체 시퀀스는 에러 상태가 된다.

다음 코드를 보자.

---

```
ASQ( function(done){
    setTimeout( done, 100 );
} )
.gate(
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    function(done){
        setTimeout( function(){
            done( "World", "!" );
        }, 100 );
    }
)
.val( function(msg1,msg2){
    console.log( msg1 ); // Hello
    console.log( msg2 ); // [ "World", "!" ]
} );
```

---

다음 프라미스 코드와 비교해보자.

---

```
new Promise( function(resolve,reject){
    setTimeout( resolve, 100 );
} )
.then( function(){
    return Promise.all( [
        new Promise( function(resolve,reject){
            setTimeout( function(){
```



```

        resolve( "Hello" );
    }, 100 );
} ),
new Promise( function(resolve,reject){
    setTimeout( function(){
        // 참고: 여기서 [ ] 배열이 필요하다.
        resolve( [ "World", "!" ] );
    }, 100 );
} )
] );
} )
.then( function(msgs){
    console.log( msgs[0] ); // Hello
    console.log( msgs[1] ); // [ "World", "!" ]
} );

```

---

똑같은 비동기 흐름 제어를 나타낸 코드임에도 프라미스는 관용 코드가 더 많이 들어있다. asynquence API와 추상화 로직이 프라미스보다 더 깔삼한 느낌을 주는 대목이다. 복잡한 비동기성일수록 더 큰 개선 효과를 기대할 수 있다.

## 단계 변형

이바지 플러그인 중에는 제법 쓸 만한 asynquence의 `gate( ... )` 단계 변형이 있다.

- `any( ... )`는 한 조각<sup>segment</sup>만 메인 시퀀스에서 계속 남아 진행한다는 점을 제외하고 `gate( ... )`와 같다.
- `first( ... )`는 한 조각이라도 성공하면 메인 시퀀스를 진행한다(다른 조각들 후속 결과는 무세)는 점만 다르고 `any( ... )`와 같다.
- (`Promise.race( [ ... ] )`에 해당하는) `race( ... )`는 한 조각이라도 (성공이든 실패든) 완료하면 바로 메인 시퀀스를 진행한다는 점만 다를 뿐 `first( ... )`와 같다.
- `last( ... )`는 성공한 마지막 한 조각만 메인 시퀀스에 메시지를 전송한다. 나머지는 `any( ... )`와 같다.
- `none( ... )`은 `gate( ... )`의 역으로, 모든 조각들이 실패해야만 메인 시퀀스를 진행한다 (여러 메시지(들)는 모두 성공 메시지(들)로 바뀐다. 그 반대 역시 마찬가지다).

이해를 돕기 위해 몇 가지 헬퍼를 정의한다.

---

```
function success1(done) {
  setTimeout( function(){
    done( 1 );
  }, 100 );
}
```

```
function success2(done) {
  setTimeout( function(){
    done( 2 );
  }, 100 );
}
```

```
function failure3(done) {
  setTimeout( function(){
    done.fail( 3 );
  }, 100 );
}
```

```
function output(msg) {
  console.log( msg );
}
```

---

이제 `gate( ... )` 단계 변형의 예를 들어 보자.

---

```
ASQ().race(
  failure3,
  success1
)
.or( output ); // 3

ASQ().any(
  success1,
  failure3,
  success2
)
.val( function(){
  var args = [].slice.call( arguments );
```

```

    console.log(
      args // [ 1, undefined, 2 ]
    );
  } );

```

```

ASQ().first(
  failure3,
  success1,
  success2
)
.val( output ); // 1

```

```

ASQ().last(
  failure3,
  success1,
  success2
)
.val( output ); // 2
  ASQ().none(
    failure3
  )
.val( output ) // 3
  .none(
    failure3
    success1
  )
.or( output ); // 1

```

---

비동기적으로 배열 원소를 다른 값으로 매핑하는 `map(..)` 역시 또 다른 단계 변형으로, 매핑이 다 끝나야만 해당 단계를 진행한다. `map(..)`은 `gate(..)`와 매우 비슷한데, 따로따로 함수를 지정하지 않고 배열에서 초기 값을 취하는 점만 다르다. 각 배열값에 대해 콜백 함수 하나만 정의하면 되기 때문이다.

---

```

function double(x,done) {
  setTimeout( function(){
    done( x * 2 );
  }, 100 );
}

```

```
}  
  
ASQ().map( [1,2,3], double )  
.val( output ); // [2,4,6]
```

---

map(...)은 이전 단계에서 넘어온 메시지에서 파라미터(배열 또는 콜백)를 받기도 한다.

---

```
function plusOne(x,done) {  
    setTimeout( function(){  
        done( x + 1 );  
    }, 100 );  
}  
  
ASQ( [1,2,3] )  
.map( double ) // 메시지 '[1,2,3]'가 들어간다.  
.map( plusOne ) // 메시지 '[2,4,6]'가 들어간다.  
.val( output ); // [3,5,7]
```

---

waterfall(...)도 단계 변형의 일종인데, gate(...)의 메시지 취합과 then(...)의 순차 처리 로직을 혼합한 함수다.

1단계가 먼저 실행되고 1단계 성공 메시지가 2단계로 넘어가 1, 2단계 성공 메시지가 모두 3단계로 흘러들면 그 다음 이들 세 성공 메시지가 4단계로 이동... 이런 식으로 마치 메시지를 취합해서 “폭포수”처럼 흘러보내는 것처럼 작동한다.

다음 코드를 보자.

---

```
function double(done) {  
    var args = [].slice.call( arguments, 1 );  
    console.log( args );  
  
    setTimeout( function(){  
        done( args[args.length - 1] * 2 );  
    });  
}
```

```

    }, 100 );
}

ASQ( 3 )
.waterfall(
    double, // [ 3 ]
    double, // [ 6 ]
    double, // [ 6, 12 ]
    double // [ 6, 12, 24 ]
)
.val( function(){
    var args = [].slice.call( arguments );
    console.log( args ); // [ 6, 12, 24, 48 ]
} );

```

---

“폭포수” 어딘가에서 에러가 나면 전체 시퀀스는 즉시 에러 상태가 된다.

## 에러 허용

단계에서 발생한 에러로 인해 전체 시퀀스가 에러 상태로 빠지게 하고 싶지 않을 경우를 위해 `asynquence`는 두 가지 단계 변형을 지원한다.

`try( .. )`는 성공 시 시퀀스를 정상 진행하지만, 실패 시 에러 메시지를 `{ catch: .. }` 형식의 성공 메시지로 탈바꿈한다.

```

ASQ()
.try( success1 )
.val( output ) // 1
.try( failure3 )
.val( output ) // { catch: 3 }
.or( function(err){
    // 이 부분은 실행되지 않는다.
} );

```

---

`until( .. )`는 재시도 루프를 설정한 뒤 단계를 시도해보고 실패하면 다음 이벤트 루프 틱에서 해당 단계를 다시 시도한다.

재시도 루프는 무한 루프라 중간에 빠져나가려면 완료 트리거에서 `break()` 플래그를 호출해서 메인 시퀀스를 에러 상태로 보내면 된다.

---

```
var count = 0;

ASQ( 3 )
.until( double )
.val( output ) // 6
.until( function(done){
    count++;

    setTimeout( function(){
        if (count < 5) {
            done.fail();
        }
        else {
            // 'until(..)' 재시도 루프를 빠져나간다.
            done.break( "허격" );
        }
    }, 100 );
} )
.or( output ); // 허격
```

---

## 프래미스식 단계

`then(..)/catch(..)` 같은 프래미스식으로 시퀀스를 꾸미고 싶다면 `pThen/pCatch` 플러그인이 있다.

---

```
ASQ( 21 )
.pThen( function(msg){
    return msg * 2;
} )
.pThen( output ) // 42
.pThen( function(){
    // 예외를 던진다.
    doesnt.Exist();
} )
```

```

.pCatch( function(err){
    // 예외를 잡는다. (버림)
    console.log( err ); // ReferenceError
} )
.val( function(){
    // 이전 예외를 'pCatch(..)'가 잡았기 때문에
    // 메인 시퀀스는 성공 상태로 되돌아간다.
} );

```

---

pThen ( ..)/pCatch ( ..)는 원래 시퀀스에서 끌려고 만들었지만 일반 프라미스 연쇄와 똑같이 작동한다. 따라서 pThen ( ..)에 전달한 이름 처리기에서 진짜 프라미스, 또는 asynquence 시퀀스로 귀결된다.

## A.2.4 시퀀스 분기

한 프라미스에 여러 then ( ..) 처리기를 덧붙여 흐름 제어를 분기할 수 있는 건 프라미스의 매력 중 하나다.

```

var p = Promise.resolve( 21 );

// 분기 1 ('p'에서)
p.then( function(msg){
    return msg * 2;
} )
.then( function(msg){
    console.log( msg ); // 42
} )

// 분기 2 ('p'에서)
p.then( function(msg){
    console.log( msg ); // 21
} );

```

---

asynquence도 fork ( )로 손쉽게 동일한 “분기<sup>forking</sup>”를 할 수 있다.

---

```
var sq = ASQ(..).then(..).then(..);
```

```
var sq2 = sq.fork();
```

```
// 분기 1
sq.then(..)..;
```

```
// 분기 2
sq2.then(..)..;
```

---

## A.2.5 시퀀스 병합

seq (..) 인스턴스 메소드는 fork ()의 정반대로, 두 시퀀스를 어느 한쪽으로 병합한다.

---

```
var sq = ASQ( function(done){
    setTimeout( function(){
        done( "Hello World" );
    }, 200 );
} );

ASQ( function(done){
    setTimeout( done, 100 );
} )
// 'sq' 시퀀스를 합한다.
.seq( sq )
.val( function(msg){
    console.log( msg ); // Hello World
} )
```

---

예제처럼 시퀀스를 직접 받거나 함수를 받는다. 함수를 인자로 넘기면 호출 시 이 함수가 시퀀스를 반환하므로 결국 다음 코드처럼 된다.

---

```
// ..
.seq( function(){
    return sq;
} )
```



```
} )  
// ..
```

---

대신에 이 단계는 `pipe(...)`로 마무리될 것이다.

---

```
// ..  
.then( function(done){  
    // 'sq'를 진행 콜백 'done'으로 보낸다(pipe).  
    sq.pipe( done );  
} )  
// ..
```

---

시퀀스가 병합되면 성공 메시지 스트림, 에러 스트림 모두 전송된다.



이전 노트에서 메모했듯이 `pipe(...)`로 수동화하든, `seq(...)`로 자동화하든 소스 시퀀스에서 에러 알림을 끌 수 있지만 타겟 시퀀스의 에러 알림 상태에는 영향을 끼치지 못한다.

## A.3 값과 에러 시퀀스

시퀀스 단계가 일반 값일 때에는 그 단계의 완료 메시지로 매핑된다.

---

```
var sq = ASQ( 42 );  
  
sq.val( function(msg){  
    console.log( msg ); // 42  
} );
```

---

자동으로 에러가 나는 시퀀스를 만들려면,

---

```
var sq = ASQ.failed( "허격" );  
  
ASQ()
```

```

.seq( sq )
.val( function(msg){
    // 실행되지 않는다.
} )
.or( function(err){
    console.log( err ); // 허걱
} );

```

---

자동으로 지연된 값/에러 시퀀스를 생성해야 할 때도 있다. 이럴 때 `after`, `failAfter` 플러그인을 쓴다.

---

```

var sq1 = ASQ.after( 100, "Hello", "World" );
var sq2 = ASQ.failAfter( 100, "허걱" );

sq1.val( function(msg1,msg2){
    console.log( msg1, msg2 ); // Hello World
} );

sq2.or( function(err){
    console.log( err ); // 허걱
} );

```

---

`after(..)`로 시퀀스 중간에 지연을 삽입할 수도 있다.

---

```

ASQ( 42 )
// 시퀀스에 지연을 넣는다.
.after( 100 )
.val( function(msg){
    console.log( msg ); // 42
} );

```

---

## A.4 프라미스와 콜백

나는 `asynquence` 시퀀스가 순수 프라미스보다 더 많은 가치를 제공하며, 전반

적으로 동일한 추상화 수준에서 개발자의 입맛에 맞는 강력한 기능을 갖고 있다고 생각한다. 하지만 `asynquence`가 아닌 코드와 `asynquence`를 통합하는 일이 관건이다.

프라미스는 `promise( .. )` 인스턴스 메소드로 어렵지 않게 시퀀스에 포함할 수 있다.

---

```
var p = Promise.resolve( 42 );

ASQ()
  .promise( p ) // function(){ return p; }도 가능하다.
  .val( function(msg){
    console.log( msg ); // 42
  } );
```

---

반대로, 특정 단계에서 프라미스를 시퀀스로부터 분기/분리하려면 `toPromise` 플러그인을 쓴다.

---

```
var sq = ASQ.after( 100, "Hello World" );

sq.toPromise()
// 이제 sq는 표준적인 프라미스 연쇄다.
.then( function(msg){
  return msg.toUpperCase();
} )
.then( function(msg){
  console.log( msg ); // Hello World
} );
```

---

콜백을 사용하는 시스템에 맞추어 `asynquence`를 사용할 수 있게 해주는 헬퍼도 있다. 시퀀스에서 “에러-우선 스타일” 콜백을 자동 생성하여 콜백 지향 유틸에 꽂아 넣으려면 `errfcb`를 사용한다.

---

```

var sq = ASQ( function(done){
    // 참고: "에러-우선 스타일" 콜백이어야 한다.
    someAsyncFuncWithCB( 1, 2, done.errfcb )
} )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );

// 참고: "에러-우선 스타일" 콜백이어야 한다.
anotherAsyncFuncWithCB( 1, 2, sq.errfcb() );

```

---

asynquence에도 3장 “프라미서리”, 4장 “쟁커리”에 필적하는 `ASQ.wrap(...)` 라는 시퀀스 감싸미 유틸이 있다.

---

```

var coolUtility = ASQ.wrap( someAsyncFuncWithCB );

coolUtility( 1, 2 )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );

```

---



확실하게 (그리고 재미로!) 하려면 예제의 `coolUtility`처럼 `ASQ.wrap(...)` 로 만든 시퀀스-생산 함수의 이름을 다르게 붙여주자. 내 생각엔 “시퀀리<sup>sequence</sup>” (“sequence(시퀀스)” + “factory(팩토리)”)가 적당할 듯싶다.

## A.5 이터러블 시퀀스

단계별로 알아서 책임지고 일을 끝내고 그에 따라 시퀀스를 진행한다는 것이 시퀀

스의 기본 철학이다. 프라미스도 마찬가지다.

그러나 아쉽게도 이따금 프라미스/단계를 외부에서 제어해야 할 경우 ‘기능 추출 capability extraction’로 이어질 우려가 있다.

다음 프라미스 예제를 보자.

---

```
var domready = new Promise( function(resolve,reject){
    // 지역적으로 다른 코드에 속해 있기 때문에
    // 여기에 이 코드를 넣을 일은 없다.
    document.addEventListener( "DOMContentLoaded", resolve );
} );

// ..

domready.then( function(){
    // DOM 준비 완료!
} );
```

---

프라미스에서 “기능 추출” 안티패턴은 다음과 같은 형태다.

---

```
var ready;

var domready = new Promise( function(resolve,reject){
    // 'resolve()' 기능을 추출한다.
    ready = resolve;
} );

// ..

domready.then( function(){
    // DOM 준비 완료!
} );

// ..

document.addEventListener( "DOMContentLoaded", ready );
```

---



이런 코드 색은 내 나는 안티패턴을 즐겨쓰는 개발자들이 있는데 도무지 그 이유는 모르겠다.

asynquence엔 이터러블 시퀀스<sup>iterable sequence</sup>라는 역전된 시퀀스 타입이 있어서 제어 기능을 외부에 둘 수 있다(다음 domready 같이 쓰면 재미가 쏠쏠하다).

---

// 참고: 여기서 'domready'는 시퀀스를 제어하는 이터레이터다.

```
var domready = ASQ.iterable();
```

```
// ..
```

```
domready.val( function(){  
    // DOM 준비 완료  
} );
```

```
// ..
```

```
document.addEventListener( "DOMContentLoaded", domready.next );
```

---

이터러블 시퀀스의 용도는 더 있는데 부록 B에서 다시 설명한다.

## A.6 제너레이터 실행하기

yield된 프라미스를 리스닝하고 제너레이터를 비동기적으로 재개하면서 제너레이터를 완전-실행하는 `run( .. )`이란 유틸을 4장에서 설명했다. asynquence에도 이와 비슷한 `runner( .. )`가 있다.

사용법을 알아보기 전에 먼저 다음 헬퍼를 정의한다.

---

```
function doublePr(x) {  
    return new Promise( function(resolve,reject){  
        setTimeout( function(){  
            resolve( x * 2 );  
        } );  
    } );  
}
```

```

    }, 100 );
  } );
}

function doubleSeq(x) {
  return ASQ( function(done){
    setTimeout( function(){
      done( x * 2)
    }, 100 );
  } );
}

```

---

이제 시퀀스 중간에 `runner (..)`를 하나의 단계처럼 사용한다.

---

```

ASQ( 10, 11 )
.runner( function*(token){
  var x = token.messages[0] + token.messages[1];

  // 진짜 프라미스를 yield한다.
  x = yield doublePr( x );

  // 시퀀스를 yield한다.
  x = yield doubleSeq( x );

  return x;
} )
.val( function(msg){
  console.log( msg ); // 84
} );

```

---

## A.6.1 감싼 제너레이터

`ASQ.wrap (..)`은 자체-패키지<sup>self-packaged</sup> 제너레이터(주어진 제너레이터를 실행하고 완료 시 시퀀스를 반환하는 일반 함수)를 생성한다.

---

```

var foo = ASQ.wrap( function*(token){

```

```

var x = token.messages[0] + token.messages[1];

// 진짜 프라미스를 yield한다.
x = yield doublePr( x );

// 시퀀스를 yield한다.
x = yield doubleSeq( x );

return x;
}, { gen: true } );

// ..

foo( 8, 9 )
.val( function(msg){
    console.log( msg ); // 68
} );

```

---

runner( .. )에는 이 밖에도 쿨한 기능이 많은데 다음 장에서 다시 이야기한다.

## A.7 정리하기

asynquence는 프라미스 기반의, 일련의 (비동기) 단계를 시퀀스로 단순하게 추상화한 유틸로, 기능은 그대로 유지한 채 다양한 비동기 패턴을 훨씬 쉽게 사용할 수 있다.

asynquence 핵심 API와 관련 플러그인 중에는 여기에 설명한 내용 말고도 유용한 것들이 많은데, 관심있는 독자는 직접 찾아보고 사용해보기 바란다.

asynquence의 본질과 기본 사상을 알아보았다. 핵심은 시퀀스가 단계들로 구성된다는 것이고, 이 단계는 프라미스의 수많은 변형 중 하나이거나 제너레이터가 될 수도 있다. 선택은 여러분의 몫이다. 작업에 가장 잘 맞다고 판단되는 비동기 흐름 제어 로직을 마음껏 엮어보라. 더 이상 뭔가 색다른 비동기 패턴을 위해 라이



브러리를 바꿀 필요는 없다.

asynquence 코드가 잘 맞는 것 같다면 서둘러 라이브러리를 습득하기 바란다. 학습 시간이 그리 오래 걸리지 않는다!

아직도 작동 원리나 이유가 혼란스러운 독자는 책장을 넘기지 말고 부록 A의 예제들을 다시 한번 잘 살펴서 직접 asynquence 실습을 해보자. 부록 B에서 asynquence는 더욱 강력한 고급 비동기 패턴으로 진화한다.

## 부록 B

# 고급 비동기 패턴

부록 A는 주로 프라미스 및 제너레이터에 기반을 둔, 시퀀스 지향 비동기 흐름 제어 라이브러리인 `asynquence`를 소개했다.

이번 장에서는 `asynquence` 본연의 기능과 이해를 바탕으로 몇몇 고급 비동기 패턴을 살펴보고, 별도로 많은 라이브러리를 가져다 쓰지 않고도 이러한 정교한 비동기 기법을 `asynquence`로 쉽게 버무려 쓰는 조리법을 알아본다.

### B.1 이터러블 시퀀스

앞 장의 `asynquence` 이터러블 시퀀스를 조금 더 자세히 알아보자.

다음 코드를 기억할 것이다.

---

```
var domready = ASQ.iterable();

// ..

domready.val( function(){
    // DOM 준비 완료
} );

// ..

document.addEventListener( "DOMContentLoaded", domready.next );
```

---

자, 다단계 시퀀스를 이터러블 시퀀스로 표시해보자.

---

```
var steps = ASQ.iterable();
```

```
steps
  .then( function STEP1(x){
    return x * 2;
  } )
  .steps( function STEP2(x){
    return x + 3;
  } )
  .steps( function STEP3(x){
    return x * 4;
  } );
```

```
steps.next( 8 ).value; // 16
steps.next( 16 ).value; // 19
steps.next( 19 ).value; // 76
steps.next().done; // true
```

---

보다시피 이터러블 시퀀스는 표준 이터레이터라서 제너레이터(또는 다른 이터러블)처럼 ES6 for...of 루프로 순회할 수 있다.

---

```
var steps = ASQ.iterable();
```

```
steps
  .then( function STEP1(){ return 2; } )
  .then( function STEP2(){ return 4; } )
  .then( function STEP3(){ return 6; } )
  .then( function STEP4(){ return 8; } )
  .then( function STEP5(){ return 10; } );
```

```
for (var v of steps) {
  console.log( v );
}
// 2 4 6 8 10
```

---

부록 A에서 봤던 이벤트 트리거링 기능 말고도 이터러블 시퀀스는 사실 상 제너레이터/프라이미스 연쇄를 대신하면서도 훨씬 더 유연하다는 점에서 흥미롭다.

다중 AJAX 요청 예제(3, 4장에서 프라이미스 연쇄와 제너레이터를 각각 배우면서 봤던 예제다)를 이터러블 시퀀스로 표현해보자.

---

// 시퀀스-인식형 AJAX

```
var request = ASQ.wrap( ajax );

ASQ( "http://some.url.1" )
  .runner(
    ASQ.iterable()

    .then( function STEP1(token){
      var url = token.messages[0];
      return request( url );
    } )

    .then( function STEP2(resp){
      return ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
      );
    } )

    .then( function STEP3(r1,r2){ return r1 + r2; } )
  )
  .val( function(msg){
    console.log( msg );
  } );
```

---

이터러블 시퀀스는 프라이미스 연쇄와 정말 비슷한 일련의 순차적인 (동기 또는 비동기) 단계를 나타낸다. 다시 말해, 단순 중첩된 콜백보다는 훨씬 깔끔하지만 제너레이터의 yield식 순차적 구문만큼은 아니다.

그런데 이터러블 시퀀스를 `ASQ#runner( .. )`에 전달하면 마치 제너레이터인 양

완전-실행한다. 이터러블 시퀀스를 기본적으로 제너레이터처럼 작동시킬 수 있다는 사실은 두 가지 관점에서 중요하다.

첫째, 이터러블 시퀀스는 ES6 제너레이터의 특정 부분 집합에 해당하는 ES6 이전 버전의 동등체<sup>equivalent</sup>다. 따라서 직접 작성해(서 어디서건 실행해)도 되고, 아니면 ES6 제너레이터를 작성한 뒤 이터러블 시퀀스(아니면 여기서는 프라미스 연쇄)로 트랜스파일/변환할 수도 있다.

비동기-완전-실행 제너레이터를 프라미스 연쇄의 간편 구문<sup>syntactic sugar</sup><sup>01</sup> 정도로 바라보는 건 이들이 근본적으로 동일한 구조임을 상기시킨다는 점에서 중요하다.

방금 전 예제는 다음과 같이 `asynquence`로 코딩해도 된다.

---

```
ASQ( "http://some.url.1" )
.seq( /*1단계*/ request )
.seq( function STEP2(resp){
  return ASQ().gate(
    request( "http://some.url.2/?v=" + resp ),
    request( "http://some.url.3/?v=" + resp )
  );
} )
.val( function STEP3(r1,r2){ return r1 + r2; } )
.val( function(msg){
  console.log( msg );
} );
```

---

2단계는 이렇게 나타낸다.

---

```
.gate(
  function STEP2a(done,resp) {
    request( "http://some.url.2/?v=" + resp )
    .pipe( done );
  },
```

---

<sup>01</sup> 역자주\_더 읽기 편하고 표현하기 쉬운, 프로그래밍 언어의 구문을 말합니다. 마치 사람에게 설탕(sugar) 같은 단맛을 느끼게 해주는 독특한 언어의 스타일이라 생각하면 됩니다.

```
function STEP2b(done,resp) {
  request( "http://some.url.3/?v=" + resp )
    .pipe( done );
}
)
```

---

그럼, 어차피 더 간단하고 평면적인 asynquence 연쇄로도 더 잘 할 수 있는데 굳이 ASQ#runner ( .. ) 단계에서 흐름 제어를 이터러블 시퀀스로 나타내는 이유는 뭘까?

이터러블 시퀀스만이 할 수 있는 중요한 기능이 있기 때문이다. 시선 고정!

### B.1.1 이터러블 시퀀스 확장

기본적인 제너레이터, asynquence 시퀀스, 프라미스 연쇄는 모두 ‘이르게 평가’<sup>eagerly evaluated</sup> 된다. 그래서 초기에 표현한 흐름 제어가 고정된 상태로 계속 이어진다.

하지만 이터러블 시퀀스는 ‘뒤늦게 평가’<sup>lazily evaluated</sup> 되므로 실행 도중에도 필요 시 시퀀스에 단계를 추가하여 확장할 수 있다.



이터러블 시퀀스 끝에만 덧붙일 수 있고 중간에 끼워넣을 수는 없다.

먼저 간단한 (동기적) 예제를 보면서 어떤 기능인지 훑어보자.

---

```
function double(x) {
  x *= 2;

  // 계속 확장해야 하는가?
  if (x < 500) {
    isq.then( double );
  }

  return x;
}
```

```

}

// 단일-단계 이터러블 시퀀스를 만든다.
var isq = ASQ.iterable().then( double );

for (var v = 10, ret;
    (ret = isq.next( v )) && !ret.done;
) {
    v = ret.value;
    console.log( v );
}

```

---

이터러블 시퀀스는 오직 한 단계를 정의(isq.then(double))하는 것으로 시작하지만, 이 시퀀스는 특정 조건( $x < 500$ ) 하에서 꾸준히 자기 자신을 확장한다. asynquence 시퀀스나 프라미스 연쇄로도 이와 유사한 기능을 구현할 수는 있지만 한참 모자란다.

이 예제는 사소한 로직이라 제너레이터에서 while 루프를 써서 나타낼 수도 있다. 자, 그럼 더 복잡 미묘한 케이스를 보자.

예컨대, AJAX 요청 후 반환된 응답 내용을 보니 더 많은 데이터가 필요해서 조건부로 이터러블 시퀀스 단계를 늘려 추가 요청을 한다고 하자. 아니면 조건부로 AJAX 처리 코드 끝에 값-형식화<sup>value-formatting</sup> 단계를 넣을 수도 있다.

다음 코드를 보자.

---

```

var steps = ASQ.iterable()

.then( function STEP1(token){
    var url = token.messages[0].url;

    // 형식화 단계가 추가되었는가?
    if (token.messages[0].format) {
        steps.then( token.messages[0].format );
    }
}

```

```

    return request( url );
} )

.then( function STEP2(resp){
    // AJAX 요청을 하나 더 시퀀스에 추가할까?
    if (/x1/.test( resp )) {
        steps.then( function STEP5(text){
            return request(
                "http://some.url.4/?v=" + text
            );
        } );
    }

    return ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );
} )

.then( function STEP3(r1,r2){ return r1 + r2; } );

```

---

보다시피 두 곳에서 `steps.then( .. )`으로 조건에 따라 단계를 확장하고 있다. 이터러블 시퀀스 `steps`를 실행하려면 `ASQ#runner( .. )`로 `asynquence` 시퀀스(여기서 `main`)와 함께 메인 프로그램 흐름에 편입시키면 된다.

---

```

var main = ASQ( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
    }
} )
.runner( steps )
.val( function(msg){
    console.log( msg );
} );

```

---

이터러블 시퀀스 `steps`의 유연함(조건부 작동)을 제너레이터로 표현할 수는 없을



까? 음, 못할 건 없지만 로직을 적잖이 불편하게 비틀어야 한다.

---

```
function *steps(token) {
  // 1단계
  var resp = yield request( token.messages[0].url );

  // 2단계
  var rvals = yield ASQ().gate(
    request( "http://some.url.2/?v=" + resp ),
    request( "http://some.url.3/?v=" + resp )
  );

  // 3단계
  var text = rvals[0] + rvals[1];

  // 4단계
  // 형식화 단계가 추가되었는가?
  if (token.messages[0].format) {
    text = yield token.messages[0].format( text );
  }

  // 5단계
  // AJAX 요청을 하나 더 시퀀스에 추가할까?
  if (/foobar/.test( resp )) {
    text = yield request(
      "http://some.url.4/?v=" + text
    );
  }

  return text;
}
```

// 참고: '\*steps()'은 이전에 'steps'에서 했던 것과 동일한 'ASQ' sequence로 실행 가능하다.

---

이미 밝혀진, 순차/동기적인 제너레이터 구문의 장점을 덮어놓고 보면, 확장 가능한 이터러블 시퀀스 단계의 역동성을 흉내내기 위해 \*steps() 제너레이터 폼에서 단계 로직을 다시 정렬할 수 밖에 없다.

똑같은 기능을 프라미스나 시퀀스로는 어떻게 구현할까? 대략 다음 코드와 같을 것이다.

---

```
var steps = something( .. )
.then( .. )
.then( function(..){
    // ..

    // 연쇄를 확장한다.
    steps = steps.then( .. );

    // ..
})
.then( .. );
```

---

미묘하나 꼭 알고 넘어가야 할 중요한 문제다. 이번에는 `asynquence` 대신 프라미스로, `steps` 프라미스 연쇄를 메인 프로그램 흐름에 연결해보자.

---

```
var main = Promise.resolve( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
    }
} )
.then( function(..){
    return steps; // 힌트!
} )
.val( function(msg){
    console.log( msg );
} );
```

---

무엇이 문제인지 알겠는가? 자세히 보라!

시퀀스 단계 정렬 시 경합 조건이 발생한다. `steps`는 `return steps`하는 시점에 정렬 상태에 따라 원래 정의한 프라미스 연쇄를 참조할 수도 있고, 아니면 `steps`

= steps.then (...)로 확장된 프라미스 연쇄를 가리킬 가능성도 있다.

결과적으로 두 가지 경우 중 하나다.

- steps가 아직 원래 프라미스 연쇄인 상태에서 steps = steps.then(...)로 나중에 “확장” 되면, 연쇄 끝부분에 위치한 이 확장 프라미스는 이미 steps 연쇄의 일부이므로 메인 흐름으로 간주하지 않는다. 아쉽지만 이것은 ‘이른 평가’의 한계다.
- steps가 이미 확장 프라미스 연쇄라면 확장 프라미스는 메인 흐름의 일부로서 예상대로 작동한다.

결합 조건을 허용할 수 없다는 분명한 사실 말고도 첫 번째 경우는 프라미스 연쇄에 있는 ‘이른 평가’의 문제점을 잘 보여준다. 반면에 이터러블 시퀀스는 ‘늦게 평가’되니 이런 문제 없이 확장이 쉽다.

동적인 흐름 제어가 절실한 상황이라면 이터러블 시퀀스는 더욱 진가를 발휘할 것이다.



이터러블 시퀀스를 더 알고 싶은 독자는 [asyncquence 사이트](#)를 참고하자.

## B.2 이벤트 반응형

3장에서 알기 쉽게 설명했듯이 프라미스는 매우 강력한 비동기 도구다. 하지만 한 가지 약점이, 단 한번만 귀결되는 프라미스의 본질 탓에 이벤트 스트림을 처리하기 어렵다. 솔직히 말하면 평범한 asyncquence 시퀀스도 똑같은 약점을 갖고 있다.

특정 이벤트가 발생할 때마다 일련의 단계들을 시작해야 한다고 치자. 프라미스/시퀀스 하나만으로는 모든 이벤트 발생을 일일이 표현할 수 없으므로 다음과 같이 이벤트가 발생할 때마다 완전히 새로운 프라미스 연쇄(또는 시퀀스)를 생성해야 한다.

---

```
listener.on( "foobar", function(data){

    // 이벤트를 처리할 새로운 프라미스 연쇄를 생성한다.
    new Promise( function(resolve,reject){
        // ..
    } )
    .then( .. )
    .then( .. );

} );
```

---

이 정도로 기본 기능 구현은 가능하겠지만 처음부터 의도한 로직을 잘 표현했다고 보긴 어렵다. 여기서는 이벤트 리스닝과 이벤트에 대한 반응, 두 가지 개발 기능이 뭉통그려져 있으므로 관심사의 분리 원칙에 따라 나누는 편이 좋겠다.

눈치빠른 독자라면 이 문제가 2장에서 콜백을 설명하며 제시했던 문제와 어느 정도 상응한다는 사실을 알아차렸을 것이다. 이것도 제어의 역전 문제다.

다음처럼 되역전하면 어떨까?

---

```
var observable = listener.on( "foobar" );

// 나중에
observable
    .then( .. )
    .then( .. );

// 어딘가에서
observable
    .then( .. )
    .then( .. );
```

---

물론 엄밀히 말해 `observable` 값이 프라미스는 아니지만 마치 프라미스를 감지하듯 감지<sup>observe</sup>할 수 있으니 밀접한 연관 관계가 있다. 실제로 `observable` 값은 여러 차례

감지가 가능하며, 해당 이벤트("foobar")가 발생할 때마다 알림을 보낸다.



방금 전 내가 설명한 패턴이 반응형 프로그래밍(RP) [Reactive Programming](#)을 배경으로 한 개념 및 동기를 엄청나게 단순화한 것으로, 몇몇 뛰어난 프로젝트와 언어에서 구현 및 해석된 바 있다. 함수-반응형 프로그래밍(FRP)은 RP의 변형으로 함수형 프로그래밍 기법<sup>34</sup>(불변성, 참조 무결성 등)을 데이터 스트림에 적용한 것을 일컫는다. "반응형"이란 말 자체가 이벤트에 반응하여 기능을 시간에 대해 넓게 펼친다는 의미다. 관심있는 독자는 마이크로소프트에서 제작한 ["Reactive Extensions"](#) (자바스크립트 버전은 "RxJS")<sup>35</sup>라는 환상적인 라이브러리를 살펴 보면서 "Reactive Observables"를 공부하길 권한다. 내가 이 책에서 제시한 것보다 훨씬 더 정교하고 강력한 툴이고, [안드레 스탈츠\(Andre Staltz\)](#)가 [구체적인 예제와 함께 프로그램적으로 RP를 구현하는 방법을 탁월한 필체로 친절하게 설명해냈다](#).

## B.2.1 ES7 옵저버블

이 글을 쓰는 즈음, ["옵저버블Observable"](#)이라는 새로운 데이터 타입이 ES7 초기 제안 목록에 포함되어 있다. 이 책에서 설명한 내용과 원리는 비슷하지만 아무래도 더 복잡하고 정교하다.

옵저버블은, 스트림에서 이벤트를 "구독<sup>subscribe</sup>"하는 방법을 각 이벤트에 대해 `next(...)` 메소드를 호출할 제너레이터(실은 이터레이터가 관련한다)에게 전달하는 개념이다.

다음 코드를 보자.

---

```
// 'someEventStream'은 마우스 클릭과 비슷한 이벤트 스트림이다.
var observer = new Observer( someEventStream, function*(){
    while (var evt = yield) {
        console.log( evt );
    }
} );
```

---

02 역자주\_함수형 프로그래밍이 처음인 독자는 '코드의 재사용과 높은 수준의 테스팅을 원한다면: 함수형 길들이기(조슈아 백필드, 한빛미디어 2015)'를 참고하시기 바랍니다.

03 역자주\_아쉽게도 이 웹 페이지는 현재 운영되고 있지 않다.

전달한 제너레이터는 다음 이벤트를 기다리며 while 루프 멈춤을 yield한다. 제너레이터 인스턴스에 부착된 이터레이터는 someEventStream에 새 이벤트가 게시될 때마다 next (..)를 호출하고 그 결과 이벤트 데이터는 evt 데이터로 제너레이터/이터레이터를 재개한다.

이벤트 구독에 관여하는 파트는 제너레이터가 아닌, 이터레이터다. 따라서 개념만 놓고 보면 ASQ.iterable ( ) 이터러블 시퀀스는 물론이고 여느 이터러블 객체를 다 전달할 수 있다.

흥미롭게도 특정 스트림 유형에서 옵저버블을 쉽게 생성할 수 있게 해주는 (DOM 이벤트에 특화된 fromEvent (..) 같은) 어댑터가 이미 나와있다. ES7 초기 제안으로 이어진 fromEvent (..)의 구현 코드를 들여다 보면 다음 절의 주제인 ASQ.react (..)와 놀라울 만치 흡사하다는 사실을 알게 된다.

물론, 아직은 초기 제안일 뿐이니 차후 외관이나 로직은 많이 변경될 수 있다. 그런데 서로 다른 언어, 서로 다른 라이브러리에서 비슷한 개념이 공통적으로 태동하는 모습이 참으로 재미지다!

## B.2.2 반응형 시퀀스

정말 간략하게나마 여러분의 관심과 학습 동기를 북돋우기 위해 옵저버블(그리고 F/RP) 이야기를 했다. 이제 내가 “반응형 시퀀스<sup>Reactive Sequence</sup>”라고 부르는, “반응형 옵저버블<sup>Reactive Observables</sup>”의 부분 집합을 적용하는 문제를 설명한다.

먼저, react (..)라는 asynquence 플러그인 유틸로 옵저버블을 어떻게 생성하는지 보자.

---

```
var observable = ASQ.react( function setup(next){
  listener.on( "foobar", next );
} );
```

---

그 다음, 이 옵저버블에 "반응하는<sup>react</sup>"(보통 F/RP에서는 "구독한다<sup>subscribing</sup>"고 표현한다) 시퀀스를 정의한다.

---

```
observable
.seq( .. )
.then( .. )
.val( .. );
```

---

이렇게 해서 옵저버블을 연쇄한 시퀀스 정의가 끝났다. 정말 쉽지 않은가?

F/RP에서 이벤트 스트림은 대개 `scan( .. )`, `map( .. )`, `reduce( .. )` 등의 함수형 변환 세트를 통해, 반응형 시퀀스에서 각 이벤트는 새 시퀀스 인스턴스를 통해 흘러간다. 구체적인 예시를 들어보겠다.

---

```
ASQ.react( function setup(next){
    document.getElementById( "mybtn" )
    .addEventListener( "click", next, false );
} )
.seq( function(evt){
    var btnID = evt.target.id;
    return request(
        "http://some.url.1/?id=" + btnID
    );
} )
.val( function(text){
    console.log( text );
} );
```

---

(`next( .. )` 호출로) 이벤트 트리거를 작동시키려고 하나, 또는 그 이상의 이벤트 처리기를 할당하는 코드가 바로 반응형 시퀀스의 “반응형” 부분이다.

반응형 시퀀스의 “시퀀스”에 해당하는 부분은 앞서 보았던 시퀀스와 정확히 같다. 각 단계는 진행 콜백, 프라미스, 제너레이터 등 이치에 맞는 비동기 기법이면 모두

가능하다.

한번 반응형 시퀀스를 설정하면 이벤트가 끊이지 않는 한 시퀀스의 인스턴스를 계속 초기화할 것이다. 반응형 시퀀스를 멈추고 싶을 때 `stop()`을 호출한다.

반응형 시퀀스가 `stop()`되면 이벤트 처리기(들)도 같이 해제해야 하므로 정리기 `teardown handler`를 등록한다.

---

```
var sq = ASQ.react( function setup(next,registerTeardown){
    var btn = document.getElementById( "mybtn" );

    btn.addEventListener( "click", next, false );

    // 'sq.stop()'을 호출하면 실행된다.
    registerTeardown( function(){
        btn.removeEventListener( "click", next, false );
    } );
} )
.seq( .. )
.then( .. )
.val( .. );

// 나중에
sq.stop();
```

---



`setup(...)` 처리기에 있는 `this` 바인딩 레퍼런스는 반응형 시퀀스 `sq`를 가리키므로 `this` 레퍼런스를 이용하면 반응형 시퀀스 정의에 추가하거나 `stop()` 같은 메소드를 호출하는 등의 작업이 가능하다.

노드JS 세상에서 반응형 시퀀스로 유입된 HTTP 요청을 처리하는 예제를 보자.

---

```
var server = http.createServer();
server.listen(8000);

// 반응형 옵저버
var request = ASQ.react( function setup(next,registerTeardown){
```



```

server.addListener( "request", next );
server.addListener( "close", this.stop );

registerTeardown( function(){
    server.removeListener( "request", next );
    server.removeListener( "close", request.stop );
} );
});

```

// 요청에 반응한다.

```

request
.seq( pullFromDatabase )
.val( function(data,res){
    res.end( data );
} );

```

// 노드 정리

```

process.on( "SIGINT", request.stop );

```

---

next( .. ) 트리거는 onStream( .. ), unStream( .. )으로 노드 스트림에 쉽게 변환할 수 있다.

---

```

ASQ.react( function setup(next){
    var fstream = fs.createReadStream( "/some/file" );

    // 스트림의 "데이터" 이벤트를 'next(..)'로 흘려보낸다.
    next.onStream( fstream );

    // 스트림의 끝을 리스닝한다.
    fstream.on( "end", function(){
        next.unStream( fstream );
    } );
} )
.seq( .. )
.then( .. )
.val( .. );

```

---

시퀀스를 조합하여 반응형 시퀀스 스트림을 여러 개 구성하는 방법도 있다.

```
var sq1 = ASQ.react( .. ).seq( .. ).then( .. );
var sq2 = ASQ.react( .. ).seq( .. ).then( .. );

var sq3 = ASQ.react(...)
    .gate(
        sq1,
        sq2
    )
    .then( .. );
```

요컨대, `ASQ.react( .. )`는 F/RP 개념을 가볍게 각색한 것으로, 이벤트 스트림을 시퀀스에 연결할 수 있게 해주는 장치라 “반응형 시퀀스”라는 이름이 붙었다. 반응형 시퀀스는 기본적인 반응형 용도로는 충분하다.



다음 예제도 참고하자.

`ASQ.react( .. )`로 UI 상태를 다루는 예제

- <http://jsbin.com/rozipaki/6/edit?js,output>

`ASQ.react( .. )`로 HTTP 요청/응답 스트림을 처리하는 예제

- <https://gist.github.com/getify/bba5ec0de9d6047b720e>

## B.3 제너레이터 코루틴

4장을 정독했다면 여러분에게 이미 ES6 제너레이터는 친숙한 존재가 되었을 것이다. 이 절에서는 “제너레이터 동시성”이란 주제를 다시 끄집어내어 더 깊숙이 살펴보고자 한다.

`runAll( .. )`은 여러 제너레이터를 인자로 받아 동시 실행하는 유틸이었고, 이 제너레이터들이 선택적으로 서로 메시지를 주고받으며 한 제너레이터에서 다른 제너레이터로 제어권을 협동적으로 `yield`하게 해주는 역할을 수행했다.

runAll( .. )과 유사한 개념을 구현한, 부록 A의 ASQ#runner( .. )는 하나의 제너레이터뿐만 아니라 여러 제너레이터를 동시에 완전-실행할 수 있다.

4장에서 나왔던 동시 AJAX 시나리오를 어떻게 구현할 수 있을지 생각해보자.

---

```
ASQ(
  "http://some.url.2"
)
.runner(
  function*(token){
    // 제어권을 넘겨줌
    yield token;

    var url1 = token.messages[0]; // "http://some.url.1"

    // 처음부터 다시 시작하기 위해 메시지를 비운다.
    token.messages = [];

    var p1 = request( url1 );

    // 제어권을 넘겨줌
    yield token;

    token.messages.push( yield p1 );
  },
  function*(token){
    var url2 = token.messages[0]; // "http://some.url.2"

    // 메시지를 전달 후 제어권을 넘겨줌
    token.messages[0] = "http://some.url.1";
    yield token;

    var p2 = request( url2 );

    // 제어권을 넘겨줌
    yield token;

    token.messages.push( yield p2 );

    // 결과를 시퀀스 다음 단계로 전달한다.
```

```

        return token.messages;
    }
)
.val( function(res){
    // 'res[0]'는 "http://some.url.1"의 결과다.
    // 'res[1]'는 "http://some.url.2"의 결과다.
} );

```

---

ASQ#runner (..)와 runAll (..)은 몇 가지 주요한 차이점이 있다.

- 각 제너레이터(코루틴<sup>coroutine</sup>)는 token이란 인자를 받는다. token은 제어권을 다음 코루틴으로 확실히 이전할 때 yield하는 특별한 값이다.
- token.messages는 이전 시퀀스 단계로부터 전달된 메시지를 담은 배열이다. 코루틴 간에 메시지 공유 목적으로 사용되는 자료 구조이기도 하다.
- 프라미스(또는 시퀀스) 값을 yield한다고 제어권이 넘어가는 건 아니지만 해당 값이 준비될 때까지 코루틴 처리를 멈춘다.
- 코루틴 처리 결과 마지막으로 반환, 또는 yield된 값은 시퀀스 다음 단계로 이동한다.

ASQ#runner (..) 기본 기능을 바탕으로 다른 용도에 맞게 헬퍼를 만들어 쉽게 엮어 쓸 수 있다.

### B.3.1 상태 기계

상태 기계<sup>state machine</sup>는 대부분의 프로그래머들에게 낯익은 주제다. 치장을 도와줄 간단한 유틸만 있으면 표현하기 쉬운 상태 기계 처리기를 만들 수 있다.

일단 대강의 모습을 스케치하자. 이름은 state (..)고 상태값 및 이 상태를 처리할 제너레이터를 각각 인자로 받는다. state (..)는 ASQ#runner (..)에 전달할 어댑터 제너레이터<sup>adapter generator</sup>를 생성/반환하는 복잡한 일들을 대행한다.

다음 코드를 보자.

---

```

function state(val,handler) {
    // 이 상태에 대한 코루틴 처리기를 생성한다.
    return function*(token) {
        // 상태 전이 처리기
        function transition(to) {
            token.messages[0] = to;
        }

        // 초기 상태를 설정한다. (아직 설정되지 않았다면)
        if (token.messages.length < 1) {
            token.messages[0] = val;
        }

        // 최종 상태(false)에 이를 때까지 계속한다.
        while (token.messages[0] !== false) {
            // 현재 상태가 이 처리기에 해당하는가?
            if (token.messages[0] === val) {
                // 상태 처리기를 위임한다.
                yield *handler( transition );
            }

            // 다른 상태 처리기로 제어권을 넘길까?
            if (token.messages[0] !== false) {
                yield token;
            }
        }
    };
}

```

---

코드를 뜯어보면 `state (..)`가 토큰을 취하는 제너레이터를 반환하고 상태 기계가 최종 상태(여기선 임의로 `false` 값을 택했다)에 이를 때까지 실행할 `while` 루프를 설정하고 있다. 바로 `ASQ#runner (..)`에 넘기려는 제너레이터 형태다!

`token.messages[0]` 슬롯은 상태 기계의 현재 상태를 추적하기 위한 공간으로 예약했는데, 이런 식으로 이전 시퀀스 단계로부터 초기 상태를 값으로 심어놓는 것도 가능하다.

이어서 ASQ#runner( ..)에서 state( ..) 헬퍼를 어떻게 사용하는지 알아보자.

---

```
var prevState;
```

```
ASQ(
  /* 선택적: 초기 상태값 */
  2
)
// 상태 기계를 실행한다.
// 전이: 2 -> 3 -> 1 -> 3 -> false
.runner(
  // 상태 '1' 처리기
  state( 1, function *stateOne(transition){
    console.log( "상태 1" );

    prevState = 1;
    yield transition( 3 ); // 상태 '3'으로 간다.
  } ),

  // 상태 '2' 처리기
  state( 2, function *stateTwo(transition){
    console.log( "상태 2" );

    prevState = 2;
    yield transition( 3 ); // 상태 '3'으로 간다.
  } ),

  // 상태 '3' 처리기
  state( 3, function *stateThree(transition){
    console.log( "상태 3" );

    if (prevState === 2) {
      prevState = 3;
      yield transition( 1 ); // 상태 '1'로 간다.
    }
    // 다 끝났다!
    else {
      yield "여기까지입니다!";

      prevState = 3;
      yield transition( false ); // 마지막 상태
    }
  }
);
```

```

    } )
  )
  // 상태 기계가 완료되면 다음 코드를 실행한다.
  .val( function(msg){
    console.log( msg ); // 여기까지입니다!
  } );

```

---

\*stateOne(...), \*stateTwo(...), \*stateThree(...) 세 제너레이터는 각자 상태값이 입력될 때마다 다시 시작하며 다른 값으로 transition(...)할 때 비로소 끝난다. 바로 이 부분이 중요하다. 예제 코드엔 나와 있지 않지만 이들 상태 제너레이터 처리기는 프라미스/시퀀스/甁크를 yield하여 비동기적으로 멈출 수 있다.

state(...) 헬퍼가 만들어 실제로 ASQ#runner(...)에 전달한, 내부의 제너레이터들은 숨겨진 상태로 상태 기계가 살아있을 동안 죽 동시 실행되며, 각자 협동심을 발휘하여 제어권을 다음으로 yield한다.



ASQ#runner(...) 제너레이터를 이용하여 협동적 동시성을 구현한, 좀 더 자세한 예제를 원한다면 "[핑퐁\(ping pong\)](#)" 예제를 살펴보기 바란다.

## B.4 순차적 프로세스 통신(CSP)

순차적 프로세스 통신(CSP)Communicating Sequential Processes은 1978년 C. A. R. 호어 Hoare가 발표한 논문 주제로, 1985년에 같은 제목으로 책도 출간되었다. CSP는 여러 동시 “프로세스”가 처리 중 상호 작용(통신)을 하는 문제에 관한 정규 방법론이다.

이미 1장에서 설명한 동시 “프로세스”를 여러분이 기억하고 있다는 전제 하에 CSP 탐험을 떠날 작정이다.

컴퓨터 과학사의 다른 위대한 개념들처럼 CSP도 프로세스 대수학process algebra으

로 불리는 학문적 형식주의에 찌들어 있다. 기호만 무성한 대수학 공식이 여러분에게 실질적인 의미로 다가오긴 어려울 테니 다른 방법으로 CSP를 알아보고자 한다.

이 책에서는 호어 교수의 CSP 정규 이론 및 논증, 그 밖의 명문들은 제외하고, 상아탑을 벗어난 관점에서 가능한 한 직관적으로 이해할 수 있게 CSP 아이디어를 간략히 설명하겠다.

### B.4.1 메시지 전달

CSP의 핵심 사상은 독립 프로세스 간의 모든 통신/상호 작용이 반드시 정규 메시지 채널을 통해 전달되어야 한다는 것이다. 여러분의 예상과는 반대로 CSP 메시지 전달은, 송신자 처리와 수신자 처리가 상호 메시지를 주고 받을 수 있는 상태에서 이루어지는 동기적인 액션이다.

어떻게 자바스크립트에서 이러한 동기적 메시징을 비동기 프로그래밍과 연관지을 수 있을까?

사실 내부는 동기적 또는 (대부분) 비동기적이지만 동기적인 형태의 액션을 어떻게 ES6 제너레이터로 만들어내는지 그 원리를 잘 떠올려보면 둘 간의 관계는 명확해진다.

다시 말해, 근본적인 시스템의 비동기성을 간직한 채 동시 실행 중인 여러 제너레이터의 상호 동기적 메시징이 가능하다. 각 제너레이터 코드가 비동기 액션이 재개되길 기다리면서 멈추기(차단되기)에 있음직한 일이다.

작동 과정을 구체적으로 살펴보자.

“A” 제너레이터(“프로세스”)가 “B” 제너레이터에게 메시지를 보내려 한다고 치자. 우선 “A”는 “B”에 보낼 메시지를 yield한다. (“A”를 멈춘다) “B”가 준비되고 메시지를 받으면 “A”를 재개한다(차단 해제한다).



“A” 제너레이터가 “B” 제너레이터로부터 메시지를 받는 상황도 마찬가지다. “A”가 “B”에서 올 메시지를 받기 위해 자신의 요청을 `yield`하면 (“A”를 멈추면), “B”가 메시지를 전송할 때 “A”는 이 메시지를 받아 재개한다.

CSP 메시지 전달 이론의 표현 방법으로는 클로저스크립트 <sup>ClosureScript</sup>의 `core.async` 라이브러리와 <sup>04</sup>`go` 언어 등이 유명하다. 이들은 채널 `channel`이라는 프로세스 간에 공개된 통로에서 서로 통신하는 체계를 구현했다.



채널이란 용어는 하나 이상의 값을 한번에 채널 버퍼로 보낼 때 몇 가지 모드가 있기 때문에 부분적으로 사용된다. 우리가 스트림이라고 생각하는 것과 대체로 비슷하다고 보면 된다. 더 이상 깊숙이 파고들진 않겠지만 매우 강력한 데이터 스트림 관리 기법이랄 할 수 있다.

가장 간단한 CSP 개념을 보면, “A”, “B” 간 채널에는, 값을 수신하기 위해 차단하는 `take(..)`라는 메소드와 값을 송신하기 위해 차단하는 `put(..)`이라는 메소드가 있다.

다음과 같은 모습이다.

---

```
var ch = channel();

function *foo() {
  var msg = yield take( ch );

  console.log( msg );
}

function *bar() {
  yield put( ch, "Hello World" );

  console.log( "메시지 보냈어요!" );
}
```

---

<sup>04</sup> `역자주` 구글에서 2009년도에 내놓은 프로그래밍 언어로 C와 대체로 비슷한 형태이며, 빠른 컴파일, 병행성 (concurrency) 지원 등의 특징이 있습니다.

```
run( foo );
run( bar );
// Hello World
// "메시지 보냈어요!"
```

---

이와 같은 구조적, 동기적(인 모습의) 메시지 전달 상호 작용과, ASQ#runner (..)가 token.messages 배열과 협동적 yield를 통해 제공하는 비공식적, 비구조적인 메시지 공유 체계를 잘 비교해보자. 사실상, yield put (..)은 제어권 이전을 위해 값을 보내고 실행을 멈추는 기능의 작업 하나인 반면, 이전 예제들은 여러 개의 개별 단계로 나누어졌다.

그리고 CSP는 정말 명시적으로 제어권을 넘기지 말고 나름대로 동시 루틴을 설계하여 채널에서 어떤 값을 받거나 채널을 통해 어떤 값을 보내거나 하는 일을 막아야 한다고 강조한다. 메시지 송수신 차단을 통하여 코루틴 간의 작동 순서를 조정할 수 있다는 뜻이다.



주의할 점! 이 패턴은 매우 강력한 대신 처음부터 익숙해지려고 하다가 두뇌에 찰과상을 입울지도 모른다. 동시성을 조정하는 전혀 새로운 사고 방식에 먼저 연습과 훈련을 통해 어느 정도 익숙해질 필요가 있다.

자바스크립트에 CSP 풍미를 훌륭하게 잘 녹여낸 라이브러리들이 있어 소개한다. 가장 유명한 것은 제임스 롱<sup>James Long</sup>이 저작한 `js-csp`고 아주 자세한 설명도 곁들여져 있다. 그리고 데이빗 놀렌<sup>David Nolen</sup>이 ‘클로저스크립트의 고 스타일 core.async CSP를 자바스크립트 제너레이터에 맞추기’ 문제를 고심하여 쓴 청산유수 같은 글들도 빼놓을 수 없다.

## B.4.2 CSP 에뮬레이션

지금까지 부록에서 줄곧 asynquence 라이브러리 위주로 비동기 패턴 이야기를 해온 터라 ASQ#runner(..) 제너레이터 위에 에뮬레이션 층<sup>emulation layer</sup>을 하

나 그냥 없으면 CSP API와 작동 원리를 거의 완벽하게 포팅할 수 있지 않을까 하는 생각이 든다. 바로 이 에뮬레이션 층이 asynquence의 “asynquence-contrib” 패키지에서 선택적으로 제공하는 기능 중 하나다.

앞서 나왔던 `state (..)` 헬퍼와 마찬가지로 `ASQ.csp.go (..)`는 제너레이터(고/core.async 용어로는 고루틴<sup>goroutine</sup>이라고 한다)를 받아 `ASQ#runner (..)`과 함께 사용할 수 있도록 맞춘 새로운 제너레이터를 반환한다.

token 대신 고루틴은 모든 고루틴이 실행 중 공유할, 처음 생성된 채널(ch)을 받는다. `ASQ.csp.chan (..)`으로 채널을 여러 개 생성할 수도 있다(종종 그렇게 하는 게 도움이 된다).

CSP에서는 프라미스/시퀀스/썬크 완료를 대기하는 행위를 차단하기보다는 모든 비동기성을 채널 메시지의 차단이라는 관점에서 바라본다.

따라서 `request (..)`가 반환한 프라미스를 `yield`하는 대신, `request (..)`는 값을 `take (..)`한 원천 채널을 반환한다. 즉, 이러한 프라미스/시퀀스의 맥락/용도에서 보면 단일값 채널이 대략 동등하다고 볼 수 있다.

우선 `request (..)`를 채널-인식형 버전으로 바꿔보자.

---

```
function request(url) {
  var ch = ASQ.csp.channel();
  ajax( url ).then( function(content){
    // 'putAsync(..)'는 'put(..)'을 제너레이터 외부에서
    // 사용 가능하게 바꾼 버전으로 작업이 끝나면 프라미스를 반환한다.
    // 예제에서는 이 프라미스를 사용하진 않았지만,
    // 값이 'take(..)'되는 시점에 알람이 필요할 땐
    // 사용 가능하다.
    ASQ.csp.putAsync( ch, content );
  } );
  return ch;
}
```

---

3장 “프라미서리”는 프라미스를 생산하는 유틸, 4장 “썬커리”는 썬크를 만들어 내는 유틸이라 하였고, 부록 A에서는 시퀀스 생산 유틸을 “시퀀리”라고 명명했었다.

마찬가지로 채널 생산 유틸도 비슷한 명칭이 필요할 테니 “채노리<sup>chanory</sup>” (“채널 channel” + “팩토리<sup>factory</sup>”)라고 부르자. 3장 `Promise.wrap(..)/promisify(..)`, 4장 `thunkify(..)`, 그리고 부록 A `ASQ.wrap(..)`에 상응하는 `channelify(..)` 유틸은 독자 여러분이 직접 연습 삼아 작성해보기 바란다.

자, 동시 AJAX 예제를 `asynquence` 맛 나는 CSP를 이용해 작성하자.

---

```
ASQ()
.runner(
  ASQ.csp.go( function*(ch){
    yield ASQ.csp.put( ch, "http://some.url.2" );

    var url1 = yield ASQ.csp.take( ch );
    // "http://some.url.1"

    var res1 = yield ASQ.csp.take( request( url1 ) );

    yield ASQ.csp.put( ch, res1 );
  } ),
  ASQ.csp.go( function*(ch){
    var url2 = yield ASQ.csp.take( ch );
    // "http://some.url.2"

    yield ASQ.csp.put( ch, "http://some.url.1" );

    var res2 = yield ASQ.csp.take( request( url2 ) );
    var res1 = yield ASQ.csp.take( ch );

    // 결과를 시퀀스 다음 단계로 넘긴다.
    ch.buffer_size = 2;
    ASQ.csp.put( ch, res1 );
    ASQ.csp.put( ch, res2 );
  } )
)
.val( function(res1,res2){
```

```
// 'res1'는 "http://some.url.1"의 응답이다.  
// 'res2'는 "http://some.url.2"의 응답이다.  
} );
```

---

두 고루틴 간에 URL 문자열을 교환하기 위한 메시지 전달 방법은 지극히 간단하다. 첫 번째 고루틴이 첫 번째 URL에 AJAX 요청을 하면 그 응답은 `ch` 채널로 향한다. 두 번째 고루틴이 두 번째 URL에 AJAX 요청을 하면 `ch` 채널에서 첫 번째 응답 `res1`을 추출한다. 바로 이 때 `res1`, `res2` 두 응답은 완료되어 준비를 마친다.

고루틴 실행이 종료될 때 `ch` 채널에 값이 남아있으면 시퀀스 다음 단계로 모두 넘어간다. 따라서 마지막 고루틴에서 받은 메시지를 배포하려면 `ch`에 `put(..)`하면 된다. 이 마지막 `put(..)`들이 차단되는 걸 방지하려면 `ch`의 `buffer_size`를 2(기본값: 0)로 세팅하여 버퍼 모드로 바꾸어야 한다.



[gist](#)에 가보면 `asynquence` 및 `CSP` 예제를 더 감상할 수 있다.

## B.5 정리하기

프라미스, 제너레이터는 정교하고 기능이 풍부한 비동기성을 구축하는데 필요한 기초 건축 자재라 할 수 있다.

`asynquence`에는 이터러블 시퀀스, 반응형 시퀀스("오퍼버블"), 동시적 코루틴, `CSP` 고루틴 구현에 꼭 필요한 유틸을 고루 갖추어져 있다.

이러한 패턴들은 진행 콜백, 프라미스의 기능과 함께 상이한 비동기 기능의 강력한 혼합체, 즉 시퀀스라는 깔끔한 비동기 흐름 제어 추상화에 통합한 기능을 `asynquence`에 부여한다.