Hanbit eBook

Realtime 90

JavaScript Promise

azu 지음 / 주우영 옮김

한빛미디어 Hanbit Media, Inc.

JavaScript
Promise

JavaScript Promise

초판발행 2015년 1월 28일

지은이 azu / **옮긴이** 주우영 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-729-3 15000 / **비매품**

총괄 배용석 / **책임편집** 김창수 / **기획·편집** 김상민
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 김상민
영업 김형진, 김진불, 조유미 / **마케팅** 박상용

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_ azu

웹 브라우저와 자바스크립트의 최신 기술을 익히는 것이 취미인 자바스크립트 개발자로, 자바스크립트의 최신 소식과 토픽을 공유하는 블로그(http://jser.info)를 운영하고 있다.

- 트위터 : https://twitter.com/azu_re
- 깃허브 : https://github.com/azu
- 웹사이트 : http://efcl.info/

역자 소개

옮긴이_ 주우영

NHN Technology Services 프론트엔드개발팀에서 프론트엔드 개발자로 근무하고 있으며, 네이버 모바일 날씨, 네이버 모바일 증권 등 네이버 서비스를 개발하고 있다.

페이스북의 프론트엔드개발그룹(https://www.facebook.com/groups/webfrontend/)에서 주로 활동하고 있으며, WIT 블로그(http://wit.nts-corp.com/)에서 자바스크립트 최신 소식을 공유하고 있다. 저서로는 『네이버는 이렇게 한다! 프런트엔드 개발 시작하기』(위키북스, 2014)가 있다.

- 페이스북 : https://www.facebook.com/coderifleman
- 트위터 : https://twitter.com/
- 슬라이드쉐어 : http://www.slideshare.net/UyeongJu/presentations
- 블로그 : http://blog.coderifleman.com/

저자 서문

최근 몇 년, 웹에서는 다양한 변화가 일어나고 있습니다. HTML5나 ECMAScript6 등 새로운 표준 기술이 잇달아 고안되는 등 웹의 세계는 빠르게 변화하고 있습니다. 브라우저는 그 변화에 대응하기 위해 릴리즈 속도를 높여 새로운 기술을 받아들이고 있습니다.

Promise도 표준으로 제안된 많은 기술의 하나며, ECMAScript6 사양으로 책정되어 이미 많은 브라우저에 구현되어 있습니다. Promise의 기능 자체는 새로운 것이 아니라 이미 있는 것을 표준화했을 뿐입니다. 그래서 라이브러리를 이용해 바로 사용할 수 있고, 이미 사용하고 있을지도 모릅니다.

다음 표준 기술로 제안된 Service Workers와 Streams API 등이 Promise를 기반으로 작성되어 있습니다. Promise는 자바스크립트 비동기 처리의 또 다른 방법으로, 배워두면 다른 API를 배울 때도 도움이 될 것입니다.

이 책이 Promise를 배우는 데 도움이 되었으면 좋겠습니다.

감사합니다.

역자 서문

Promise는 콜백-헬을 위한 도구가 아닙니다. Promise는 잘못 홍보되고 있습니다. 물론 새로운 기술이 특정 세계에 빠르게 흡수되게 하려고 어두운 면보단 밝은 면을 부각하는 것은 어느 정도 이해할 수 있습니다. 하지만 지나친 환상은 자칫 독이 될 수 있고, 오히려 그 기술을 피하게 되는 기피증을 유발할 수 있습니다. 특정 기술이 해결하고자 하는 문제와 전혀 다른 문제에 초점을 맞추게 되면 오히려 코드는 망가질 수 있으며 기술을 안 쓰는 게 더 나을 때도 있습니다. 아깝지 않습니까? 분명 그 기술이 제대로 쓰인다면 많은 이점을 얻을 수 있음에도 불구하고 전혀 다른 문제에서 발버둥 치다 결국 쓸모없는 것으로 치부하게 됩니다.

Promise는 단일 인터페이스, 강력한 에러 처리 메커니즘 등 많은 장점을 가지고 있습니다. 그리고 모든 기술이 그렇듯 단점도 가지고 있습니다. Promise가 해결하고자 하는 문제와 특징을 더 자세히 이해하고 목적과 상황에 맞게 사용해야 합니다. 저는 항상 Promise가 조금 더 본질에서 세상에 알려지길 바라왔습니다. 그리고 그 방법을 고민하던 중 azu의 책을 읽게 됐습니다. 약 70페이지의 짧은 분량으로 Promise의 특징과 철학을 명확하게 설명하고 있었습니다. 저는 이 책을 번역하여 한국의 개발자에게 소개하자고 다짐했습니다.

이 책을 통해 Promise와 조금 더 친해질 수 있길 바라며 목적과 상황에 맞게 사용하여 가독성과 유용성이 높은 코드를 작성할 수 있길 희망합니다.

감사합니다.

2015. 1.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 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 Only - 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 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

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 힘든 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.

차례

1 | Promise란 무엇인가

자바스크립트에서는 비동기 프로그래밍을 위한 하나의 패턴으로 콜백을 사용했다. 하지만 전통적인 콜백 패턴은 비동기 처리 중 발생한 오류를 예외 처리하기 힘들고 여러 개의 비동기 로직을 한꺼번에 처리하는 데도 한계가 있다. 즉, 모든 상황에 해당하지는 않지만 몇몇 비동기적 경우에 그다지 유용한 패턴이 아니다.

이때 비동기 프로그래밍을 위한 또 다른 패턴으로 Promise가 등장했다. Promise는 전통적인 콜백 패턴이 가진 단점을 일부 보완하며 비동기 처리 시점을 명확하게 표현한다. 이 책에서는 Promise의 기본을 잘 이해할 수 있도록 Promise/A+ 사양에 기반을 둔 ECMAScript6를 통해 Promise를 소개한다. 또한, 테스트 작성 시 유의할 점과 Promise가 적합한 경우와 부적합한 경우에 관해서도 다룬다. 이번 장에서는 Promise의 정의와 사용 방법에 대해서 개괄적으로 살펴본다.

1.1 Promise란

가장 먼저 Promise가 무엇인지 알아보자. Promise는 비동기 처리 로직을 추상화한 객체와 그것을 조작하는 방식을 말한다. 이 개념은 E 언어에서 처음 고안됐으며 병렬 및 병행 프로그래밍을 위한 일종의 디자인이다. 이 디자인을 자바스크립트 환경에 가져온 것이 이 책에서 배울 Promise다. 보통 자바스크립트에서는 비동기 처리를 위해 콜백을 사용한다.

```
getAsync("fileA.txt", function(error, result){
    if(error){ // 실패 시
        throw error;
    }
    // 성공 시
```

```
});
    // 실패 시
});
```

Promise를 지원하는 함수는 비동기 처리 로직을 추상화한 promise 객체를 반환한다. 그리고 객체를 변수에 대입하고 성공 시 동작할 함수와 실패 시 동작할 함수를 등록해 사용한다. 함수를 작성하는 방법은 promise 객체의 인터페이스에 의존한다. 즉, promise 객체에서 제공하는 메서드만 사용해야 하므로 전통적인 콜백 패턴처럼 인자가 자유롭게 결정되는 게 아니라 같은 방식으로 통일된다. Promise라고 부르는 하나의 인터페이스를 이용해 다양한 비동기 처리 문제를 해결할 수 있다. 쉽게 말해 복잡한 비동기 처리를 쉽게 패턴화할 수 있다는 뜻이다. 이는 Promise의 역할이며 Promise를 사용하는 많은 이유 중 하나다.

1.2 Promise 살펴보기

이번에는 Promise의 API, 워크플로우, 상태를 간략히 소개하겠다.

1.2.1 API

ES6 Promises 사양에서 정의하는 API는 일반적으로 크게 세 종류다.

Constructor

Promise는 XMLHttpRequest처럼 생성자 함수에 new 연산자를 선언하여 promise 인스턴스 객체(이하 promise 객체)를 생성해 사용한다.

```
var promise = new Promise(function(resolve, reject) {
    // 비동기 처리 작성
    // 처리가 끝나면 resolve 또는 reject를 호출한다.
});
```

Instance Method

new 연산자로 생성한 Promise의 인스턴스 객체에는 성공(resolve) 또는 실패(reject)했을 때 호출될 콜백 함수를 등록할 수 있는 promise.then()이라고 하는 인스턴스의 메서드가 있다.

```
promise.then(onFulfilled, onRejected)
```

성공했을 때는 onFulfilled가 호출되며 실패했을 때는 onRejected가 호출된다. onFulfilled, onRejected는 선택적(optional) 인자이므로 생략할 수 있다.

이처럼 promise.then()으로 성공 또는 실패 시의 동작을 동시에 등록할 수 있다. 만약 오류 처리만 하려면 promise.then(undefined, onRejected)와 같은 의미인 promise.catch(onRejected)를 사용하면 된다.

```
promise.catch(onRejected)
```

Static Method

전역 객체인 Promise에는 Promise.all()이나 Promise.resolve() 같은 정적 메서드가 있다. 이 메서드는 Promise를 다루는 데 필요한 보조 메서드다.

1.2.2 워크플로우

[예제](소스코드: http://jsbin.com/busogo/1/embed?js,console)

```
function asyncFunction() {
   return new Promise(function (resolve, reject) {
      setTimeout(function () {
         resolve('Async Hello world');
      }, 16);
```

```
    });
}

asyncFunction().then(function (value) {
    console.log(value); // 'Async Hello world'
}).catch(function (error) {
    console.log(error);
});
```

asyncFunction()은 Promise 생성자를 new 연산하여 promise 객체를 반환한다. 그리고 비동기 처리가 성공적으로 끝났을 때 호출될 콜백을 promise 객체의 then()으로 등록한다. 실패했을 때 호출될 콜백은 catch()로 등록한다. setTimeout()으로 16ms 이후에 성공하도록 작성했으므로 16ms 이후 then()으로 등록한 콜백이 호출되어 "Async Hello world"라는 메시지가 출력된다.

앞의 경우 catch()로 등록한 콜백은 호출되지 않는다. 하지만 만약 setTimeout()이 존재하지 않는 환경이라면 예외가 발생하므로 catch()로 등록한 콜백 함수가 호출된다. catch()는 promise.then(onFulfilled, onRejected)로 대체해 작성할 수 있다.

```
asyncFunction().then(function (value) {
    console.log(value);
}, function (error) {
    console.log(error);
});
```

1.2.3 상태

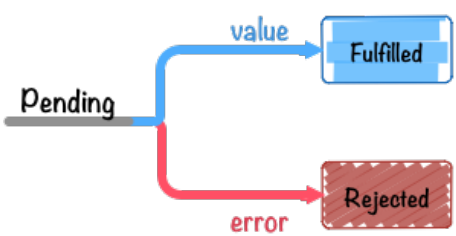
마지막으로 Promise의 상태를 설명한다. Promise 생성자 함수를 new 연산하여

생성한 promise 객체에는 다음과 같은 세 가지 상태State가 존재한다. 왼쪽은 ES6 Promises 사양에서 정한 명칭이며, 오른쪽은 Promises/A+에서 정한 명칭이다.

- “has-resolution” - Fulfilled : 성공(resolve)했을 때의 상태, onFulfilled가 호출된다.
- “has-rejection” - Rejected : 실패(reject)했을 때의 상태, onRejected가 호출된다.
- “unresolved” - Pending : 성공도 실패도 아닌 상태, promise 객체가 생성된 초기 상태를 말한다.

promise 객체의 상태를 직접 조작할 일은 없으므로 이름 자체에 크게 신경쓰지 않아도 된다. 그럼 Promise/A+의 명칭인 Pending, Fulfilled, Rejected를 이용해 상태를 설명하겠다.

[그림 1-1] Promise의 상태



[[PromiseState]]

ECMAScript Language Specification ECMA-262 6th Edition - DRAFT 문서에 따르면 Promise의 상태는 [[PromiseState]]라고 하는 내부 정의에 의해 결정된다. 하지만 [[PromiseState]]에 접근할 수 있는 API가 없어 기본적으로 알 수 없다.

promise 객체는 Pending 상태로 시작해 Fulfilled나 Rejected 상태가 되면 다시는 변화하지 않는다. 그래서 Fulfilled 및 Rejected 상태를 Settled(불변) 상태라고 한다. 즉, Event 리스너와는 다르게 then()으로 등록한 콜백 함수는 한 번만 호출된다. 정리해보면 then()은 promise 객체의 상태가 변화할 때 한번만 호출될 콜백 함수를 등록하는 메서드인 것이다.

> **Settled 상태**
> 성공 또는 실패 했을 때의 상태를 말한다. Pending과 Settled는 서로 대응하는 관계다.

이처럼 Promise 상태의 종류와 변화는 매우 단순하다. 여기까지 세 가지 상태를 모두 살펴봤다. JavaScript Promises - Thinking Sync in an Async Word 슬라이드에 Promise의 상태 변화에 대해서 이해하기 쉽게 설명돼 있으니 참고 바란다.

1.3 Promise 사용하기

마지막으로 Promise 객체를 생성하는 방법부터 사용하는 방법까지 간단하게 소개하겠다.

1.3.1 객체 생성

다음은 promise 객체를 생성하는 과정이다.

1. new Promise(fn)으로 promise 객체를 생성한다.
2. fn에는 비동기 처리를 작성한다.
- 처리 결과가 정상이라면 resolve(결과 값)를 호출한다.
- 처리 결과가 비정상이라면 reject(error 객체)를 호출한다.

그럼 XHR(XMLHttpRequest)을 사용해 데이터를 취득하는 과정을 Promise로 작성한다. 우선, Promise를 사용해 XHR을 감싼다는 느낌으로 getURL()을 작성한다.

[예제](소스 코드: http://jsbin.com/vizoqo/1/embed?js,console)

```
getURL("http://httpbin.org/get").then(function onFulfilled(value){
    console.log(value);
}).catch(function onRejected(error){
    console.error(error);
});
```

getURL()은 XHR 요청에 대한 응답의 상태 코드가 200인 경우에만 성공(resolve)하고, 이외의 경우에는 실패(reject)한다. 성공 시에는 resolve(req.responseText)로 응답 결과를 전달한다. resolve()로 전달할 수 있는 값에 특별한 형식이 있는 건 아니며, 콜백과 마찬가지로 다음 동작에 필요한 값을 전달하면 된다. 이 값은 then()으로 등록한 콜백 함수로 전달된다. Node.js에서는 콜백 함수를 호출할 때 callback(error, response)처럼 첫 번째 인자로 에러 객체를 전달하지만, Promise에서는 그 역할을 resolve()와 reject()로 분담하고 있으므로 resolve()에는 결과 값만 전달한다.

XHR의 onerror 이벤트 핸들러가 호출된 경우는 오류인 경우이므로 reject()를 호출한다. 이때 reject()에 전달하는 값에 주목한다.

reject(new Error(req.statusText))를 보면 알 수 있듯이 Error 객체를 생성하여 전달하고 있다. resolve()와 마찬가지로 reject()에 전달할 수 있는 값에 특별한 형식이 있는 건 아니지만, 일반적으로 실패한 원인을 작성한 Error 객체(또는 Error 객체를 상속한 객체)를 전달하는 것을 원칙으로 한다. 앞의 예제에서는 상태 코드가 200이 아니라면 실패한다고 보고 statusText로 생성한 error 객체를 전달한다. 이 값은 then()의 두 번째 인자로 등록한 콜백 함수나 catch()로 등록한 콜백 함수로 전달된다.

1.3.2 객체 사용

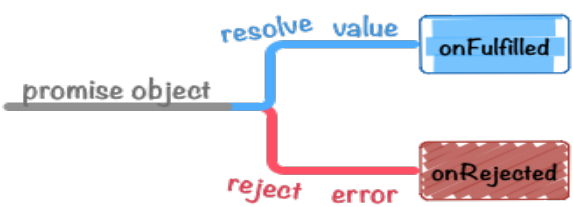
앞에서 작성한 promise 객체를 반환하는 함수, getURL()을 단계별로 사용한다.

```
getURL("http://example.com/"); // promise 객체를 반환한다.
```

앞에서 간단히 소개한 것처럼 promise 객체는 몇 가지 인스턴스 메서드를 가지고 있다. 이 메서드를 이용해 promise 객체 상태가 변화할 때 한 번만 호출될 콜백 함수를 등록할 수 있다. 쉽게 말해 resolve()나 reject()가 호출될 때 실행될 콜백을 promise객체에 등록한다.

- onFulfilled : promise 객체에서 resove()가 호출됐을 때 처리
- onRejected : promise 객체에서 reject()가 호출됐을 때 처리

[그림 1-2] Promise 값의 흐름



우선 XHR 요청이 성공하여 값을 취득할 수 있는 경우를 작성해보자. 요청이 성공했다는 것은 resolve()가 호출됐다는 것이므로 객체의 상태가 Fulfilled로 변경됐다는 것을 의미한다. then()에 콜백 함수를 전달하는 것으로 resolve()가 호출됐을 때 동작을 등록할 수 있다.

```
getURL("http://httpbin.org/get").then(function onFulfilled(value)
    console.log(value);
```

```
});
```

알아보기 쉽게 콜백에 onFulfilled 함수명을 지정했다. getURL() 내에서 resolve(req.responseText)가 호출되어 객체의 상태가 Fulfilled되면 기대하는 값과 함께 onFulfilled()가 호출된다. 하지만 이대로는 오류가 발생했을 때 아무 동작도 하지 않으므로 이번에는 처리 중 문제가 발생했을 때 동작할 코드를 작성한다.

문제가 발생했다는 것은 reject()가 호출됐다는 것이므로 객체의 상태가 Rejected가 됐다는 것을 의미한다. reject()가 호출됐을 때의 동작은 then()의 두 번째 인자 또는 catch()에 콜백 함수를 전달하면 된다.

```
getURL("http://httpbin.org/status/500").then(function onFulfilled
    console.log(value);
}).catch(function onRejected(error){
    console.error(error);
});
```

앞의 요청에 대해 서버는 상태 코드 500을 응답한다. resolve()와 마찬가지로 알아보기 쉽도록 콜백에 onRejected 함수명을 지정했다. getURL() 처리 중 어떤 이유로 오류가 발생하거나 명시적으로 reject()가 호출된 경우에 오류의 원인을 담은 error 객체와 함께 catch()로 등록한 콜백 함수가 호출된다. catch()는 promise.then(undefined, onRejected)를 단순히 랩핑한 메서드이므로 다음과 같이 작성해도 무방하다.

```
getURL("http://httpbin.org/status/500").then(onFulfilled, onRejected);
```

개인적으로는 then()과 catch()를 사용하여 onFulfilled와 onRejected를 각

각 따로 등록하는 것을 권장한다. 그 이유에 대해서는 3.3 예외 처리가 되지 않는 onRejected에서 자세히 이야기하겠다.

1.4 정리

이번 장에서는 Promise의 정의부터 API, 워크플로우, 상태에 대해 간단히 살펴봤다. 또 `new Promise()`를 사용해 `promise` 객체를 생성하는 방법과 `then()`이나 `catch()`를 사용해 promise 상태 변화에 따른 동작을 등록하는 방법 등 Promise의 기본적인 사용 방법에 대해서도 간단히 이야기했다. Promise를 사용하는 다양한 방법들은 이번 장에서 설명한 방법을 발전시킨 것이거나 Promise의 정적 메서드를 이용한 것이다. 이번에는 Promise의 매력이나 장점에 대해서 크게 언급하지 않았다. 작성한 예제는 사실 전통적인 콜백 패턴으로 작성해도 문제없다. 다음 장에서는 Promise의 장점인 오류 처리 방식을 전통적인 콜백 기반과 비교한다.

2 | Promise 사용하기

이번 장에서는 Promise의 정적 메서드와 Promise.prototype의 메서드, Thenable에 관한 개념 등 조금 더 다양하고 자세한 내용을 이야기한다. 비동기 처리 시 가장 힘든 부분 중 하나가 바로 오류에 대한 예외를 처리하는 것이다. 이를 조금 더 효율적으로 다룰 수 있도록 하는 Promise의 예외 처리 메커니즘도 이야기하겠다.

2.1 Promise.resolve

Promise의 정적 메서드인 Promise.resolve()를 사용하면 new Promise() 구문을 단축해 표기할 수 있다.

[예제](소스코드: http://jsbin.com/binomo/1/embed?js,console)

```
new Promise(function(resolve){
    resolve(42);
}).then(function(value){
    console.log(value);
});
```

앞의 코드는 객체 초기화 시 resolve(42)하기 때문에 42라는 값과 함께 then()을 이용해 등록한 콜백 함수가 바로 호출된다. Promise.resolve() 역시 Fulfilled 상태인 promise 객체를 반환하므로 다음과 같이 작성할 수 있다.

```
Promise.resolve(42).then(function(value){
    console.log(value);
});
```

때에 따라 Promise.resolve()와 new Promise()는 같은 의미를 가지며 promise 객체를 초기화할 때나 테스트 코드를 작성할 때 활용할 수 있다. 또한, Promise.resolve()는 thenable 객체를 promise 객체로 변환할 수 있다. 이것은 Promise.resolve()의 중요한 특징 중 하나다.

thenable은 ES6 Promises 사양에 정의된 개념이다. then()을 가진 객체 즉, 유사 promise 객체를 의미한다. length 프로퍼티를 갖고 있지만, 배열이 아닌 유사 배열 객체Array-like Object와 같다. Promise.resolve()는 thenable 객체의 then()이 Promise의 then()과 같은 동작을 할 것이라 기대하고 promise 객체로 변환한다. 가장 대표적인 thenable 객체는 jQuery.ajax()가 반환하는 객체다. jQuery.ajax()는 jqXHR을 반환하는데 이 객체는 then()을 갖고 있다.

```
$.ajax('/json/comment.json'); // then() 을 갖는 객체를 반환한다.
```

Promise 객체로 만들면 then()과 catch() 같은 ES6 Promises의 기능을 사용할 수 있게 된다. 다음 예제를 참고한다.

[예제](소스코드: http://jsbin.com/xicezi/1/embed?js,console)

```
var promise = Promise.resolve($.ajax('http://httpbin.org/get')); // promise
객체를 반환한다.

promise.then(function(value){
  console.log(value); // [object Object]
});
```

jQuery와 thenable

jQuery.ajax()가 반환하는 jqXHR Object는 Deferred Object를 상속한다. 하지만 Deferred Object는 Promise/A+나 ES6 Promises 사양을 따른 것이

아니므로 promise 객체로 변환하면 특정 처리 시 문제가 발생할 수 있다. 대개 Deferred Object의 then()이 ES6 Promises 사양에서 정의하는 then()과 다르게 동작하기 때문에 발생하는 문제가 많다. 따라서 then()을 가진 객체일지라도 ES6 Promises 사양을 따른다고 생각해선 안 된다.

- JavaScript Promises: There and back again
- You're Missing the Point of Promises

thenable 객체를 promise 객체로 변환하는 기능은 과거 Promise.cast()라는 메서드로 존재했지만, 지금은 Promise.resolve()를 사용한다. thenable이라는 개념은 Promise를 사용하기 위해서는 알아야 할 중요한 개념이지만 일반적으로 이 기능을 사용해야 할 경우는 많지 않다. thenable과 Promise.resolve()에 관한 구체적인 내용은 5장의 Promise.resolve와 thenable에서 설명하겠다.

2.2 Promise.reject

Promise.reject()도 Promise.resolve()처럼 new Promise()를 단축해 표기할 수 있다. 예를 들어 다음 예제는 Promise.reject(new Error(' 오류 '))와 같다.

[예제](소스코드: http://jsbin.com/nafudi/1/embed?js,console)

```
new Promise(function(resolve, reject){
    reject(new Error('오류'));
}).catch(function(error){
    console.error(error.message); // 오류
});
```

Promise.reject()도 promise 객체를 반환한다. 따라서 에러 객체와 함께 catch()를 이용해 등록한 콜백 함수가 호출된다.

[예제](소스코드: http://jsbin.com/nocabo/1/embed?js,console)

```
Promise.reject(new Error('오류')).catch(function(error){
    console.error(error.message); // 오류
});
```

promise 객체의 상태가 Fulfilled가 아닌 Rejected가 된다는 점이 Promise.resolve()와 다르다. 테스트 코드를 작성하거나 디버깅 또는 일관성을 위해 사용할 수 있다.

2.3 Promise.prototype.then

1장 Promise란 무엇인가?에서 then()과 catch()를 설명할 때 두 메서드를 체인하여 예제 코드를 작성했다. 이처럼 Promise에서는 얼마든지 메서드를 체인하여 코드를 작성할 수 있다.

```
aPromise.then(function taskA(value){
    // task A
}).then(function taskB(vaue){
    // task B
}).catch(function onRejected(error){
    console.log(error);
});
```

메서드를 체인 하여 작성한 콜백 함수는 task A에서 task B 즉, 등록한 순서대로 처리된다. 비동기 처리를 쉽게 다룰 수 있도록 하는 이유 중 하나가 바로 promise 체인이다. 그럼 promise 객체에 메서드를 체인 하여 등록한 함수들이 어떤 흐름으로 호출되는지 알아보자.

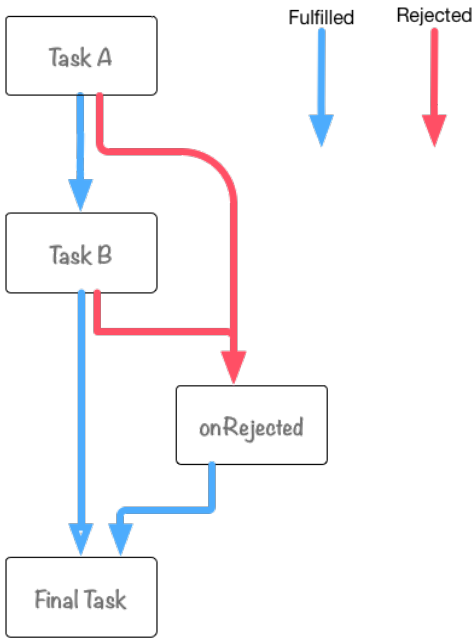
체인은 짧을수록 좋다. 다음 예제는 이해를 돕기 위해 체인을 길게 작성한 것이다.

[예제](소스코드: http://jsbin.com/tipan/1/embed?js,console)

```
function taskA() {
    console.log("Task A");
}

function taskB() {
    console.log("Task B");
}

function onRejected(error) {
    console.log("Catch Error: A or B", error);
}

function finalTask() {
    console.log("Final Task");
}

var promise = Promise.resolve();
promise.then(taskA)
       .then(taskB)
       .catch(onRejected)
       .then(finalTask);

// Task A
// Task B
// Final Task
```

앞 코드의 흐름을 다음 그림과 함께 설명하겠다.

[그림 2-1] Promise의 then(), catch() 흐름



Task A와 Task B에 각각 onRejected를 향한 빨간 선이 있다. 이것은 Task A 또는 Task B를 처리하던 중 오류가 발생하면 Rejected 상태인 promise 객체가 반환되어 catch()로 등록한 onRejected 함수가 호출되는 상황을 나타낸다. Promise에서 추상화하고 있는 로직은 마치 try-catch 되고 있는 것과 같다. 따라서 예외가 발생해도 프로그램은 중단되지 않는다. 하나 주의해야 할 점은 onRejected와 Final Task 뒤에 catch()를 체인하지 않았으므로 onRejected와 Final Task 처리 중 예외가 발생한 경우를 감지할 수 없다는 점이다.

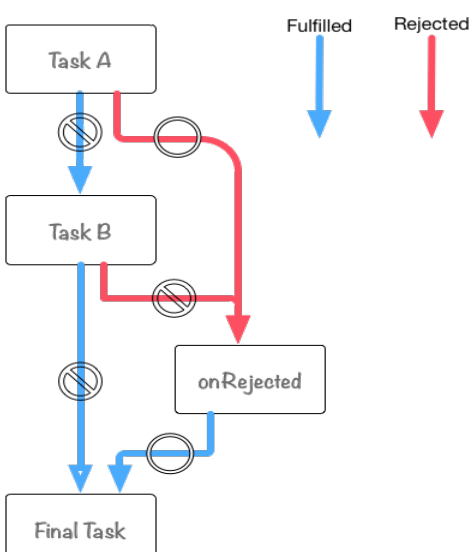
[예제](소스코드: http://jsbin.com/pumeqe/1/edit?js,console)

```
function finalTask() {
    throw new Error('Final Task 처리 중 에러');
```

```
}

var promise = Promise.resolve();
promise.then(taskA)
       .then(taskB)
       .catch(onRejected)
       .then(finalTask);

// 예외가 발생했음을 알 수 없다.
// 개발자 도구에도 출력되지 않는다.
```

이번엔 Task A에서 예외가 발생해 onRejected가 호출되는 경우를 살펴보자. Task A 처리 중 예외가 발생하면 Task A부터 시작해 onRjected, FinalTask 순으로 처리된다.

[그림 2-2] Task A에서 예외가 발생했을 때 흐름



앞의 그림을 코드로 작성하면 다음과 같고, 다음 예제에서 Task B는 호출되지 않는다.

[예제](소스코드: http://jsbin.com/qisoku/1/embed?js,console)

```
function taskA() {
    console.log("Task A");
    throw new Error("throw Error @ Task A");
}

function taskB() {
    console.log("Task B"); // 호출되지 않는다.
}

function onRejected(error) {
    console.error(error.message);
}

function finalTask() {
    console.log("Final Task");
}

var promise = Promise.resolve();
promise.then(taskA)
       .then(taskB)
       .catch(onRejected)
       .then(finalTask);

// Task A
// throw Error @ Task A
// Final Task
```

이해를 돕기 위해 예외 처리 시 throw 구문을 사용했지만, Rejected 상태인

promise 객체를 반환하면 되므로 reject() 사용을 권장한다. 이와 관련된 내용은 5.3 throw 대신 reject 사용에서 자세히 다루도록 하겠다.

다음으로 Task A의 처리 결과 값을 Task B에 어떻게 전달할 수 있는지 알아본다. 방법은 의외로 간단하다. Task A에서 반환한 값이 Task B가 호출될 때 인자로 설정되기 때문이다.

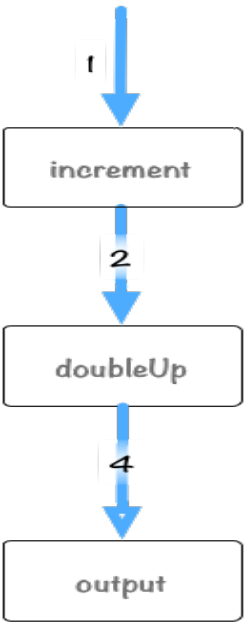
[예제](소스코드: http://jsbin.com/fohima/1/embed?js,console)

```
function doubleUp(value) {
    return value * 2;
}

function increment(value) {
    return value + 1;
}

function output(value) {
    console.log(value); // (1 + 1) * 2
}

var promise = Promise.resolve(1);
promise.then(increment)
       .then(doubleUp)
       .then(output); // 4
```

앞의 코드는 다음과 같은 흐름으로 처리된다.

1. Promise.resolve(1)의 1이 increment()에 전달된다.
2. Increment()는 전달받은 값에 1을 더하고 반환한다.
3. 이 값은 다음 doubleUp()으로 전달된다.
4. 마지막으로 output() 실행되고 메시지가 출력된다.

[그림 2-3] promise 체인에서 값이 전달되는 과정



반환 값으로는 숫자, 문자열, 객체뿐만 아니라 promise 객체도 가능하다. 반환 값은 Promise.resolve(반환 값)처럼 처리되기 때문에 무엇을 반환하더라도 최종적으로는 새로운 promise 객체가 반환된다. 즉, then()은 단순히 콜백 함수를 등록하기만 하는 게 아니라 콜백에서 반환받은 값을 기준으로 새로운 promise 객체를 생성하여 전달하는 기능도 하고 있다. 이 원리에 대한 내용은 3장 새로운 promise 객체를 반환하는 then과 catch에서 흔히 겪는 실수와 함께 자세히 소개하겠다.

이번에는 복수의 XHR 요청의 경우를 예로 여러 개의 promise 객체가 전부 Fulfilled 상태가 됐을 때의 처리를 어떻게 할 수 있는지 알아본다. 가장 먼저 전통적인 콜백 스타일로 작성한다.

[예제](소스코드: http://jsbin.com/qijuta/1/embed?js,console)

```
function getURLCallback(URL, callback) {
    var req = new XMLHttpRequest();

    req.open('GET', URL, true);

    req.onload = function () {
        if (req.status == 200) {
            callback(null, req.responseText);
        } else {
            callback(new Error(req.statusText), req.response);
        }
    };

    req.onerror = function () {
        callback(new Error(req.statusText));
    };

    req.send();
}

// <1> JSON 파싱
function jsonParse(callback, error, value) {
    if (error) {
        callback(error, value);
    } else {
        try {
            var result = JSON.parse(value);
            callback(null, result);
        } catch (e) {
            callback(e, value);
        }
```

```
    }
}

// <2> XHR 요청
var request = {
  infomation: function getInfomation(callback) {
    return getURLCallback('http://httpbin.org/get', jsonParse.bind(null,
callback));
  },
  cookie: function getCookie(callback) {
    return getURLCallback('http://httpbin.org/cookies', jsonParse.bind(null,
callback));
  }
};

// <3> 복수의 XHR 리퀘스트를 실시하고 모두 완료되면 callback을 호출
function allRequest(requests, callback, results) {
    var req = null;

    if (requests.length === 0) {
        return callback(null, results);
    }

    req = requests.shift();

    req(function (error, value) {
        if (error) {
            callback(error, value);
        } else {
            results.push(value);
            allRequest(requests, callback, results);
        }
    });
```

```
}

function main(callback) {
    allRequest([request.infomation, request.cookie], callback, []);
}

// 실행
main(function(error, results){
    if(error){
        return console.error(error);
    }
    console.log(results);
});
```

전통적인 콜백 스타일의 코드를 보면 몇 가지 패턴이 보인다.

- JSON.parse를 바로 사용하면 문제가 될 수 있으므로 랩핑 함수인 jsonParse()를 만들어 사용한다.
- 복수의 XHR을 그대로 요청하면 콜백 중첩이 깊어지므로 여러 개의 요청을 위한 allRequest()를 만들어 사용한다.
- Node.js에서 흔히 볼 수 있는 인터페이스를 차용해 인자를 callback(error, value)처럼 전달한다.

익명 함수의 사용을 줄이기 위해 jsonParse()를 사용할 때 bind를 사용했다. 전통적인 콜백 스타일이라도 목적별로 함수를 잘 분리하면 익명 함수의 사용을 줄일 수 있다. 하지만 다음과 같은 사항이 신경쓰인다.

- 예외 처리를 위한 코드가 반복된다.
- 중첩이 깊어지는 문제를 피하고자 요청을 다루는 별도의 함수가 필요하다.
- 콜백을 자주 사용한다.

이 문제를 then()으로 해결한다. 이해를 돕기위해 then()을 사용하지만, 사실은 복수의 비동기 처리 시 유용한 Promise.all()과 Promise.race()라고 하는 정적 메서드가 있다. 이 메서드를 사용한 처리 방법은 2.5 Promise.all에서 설명하겠다. then()을 사용한 경우는 다음과 같이 작성할 수 있다.

[예제](소스코드: http://jsbin.com/biwomo/1/embed?js,console)

```
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();

        req.open('GET', URL, true);

        req.onload = function () {
            if (req.status == 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };

        req.onerror = function () {
            reject(new Error(req.statusText));
        };

        req.send();
    });
}

var request = {
  infomation: function getComment() {
    return getURL('http://httpbin.org/get').then(JSON.parse);
  },
```

```
  cookie: function getPeople() {
    return getURL('http://httpbin.org/cookies').then(JSON.parse);
  }
};

function main() {
    var pushValue = null;

    function recordValue(results, value) {
        results.push(value);
        return results;
    }

    pushValue = recordValue.bind(null, []);

    return
request.infomation().then(pushValue).then(request.cookie).then(pushValue);
}

main().then(function (value) {
    console.log(value);
}).catch(function(error){
    console.error(error);
});
```

then()을 사용한 코드는 전통적인 콜백 스타일과 다음과 같은 차이가 있다.

- JSON.parse를 바로 사용한다. 부가적인 예외 처리를 하지 않는다.
- main()은 promise 객체를 반환한다.
- 예외 처리는 반환된 promise 객체에 작성한다.

여기까지 then()을 사용하여 복수의 promise를 처리하는 방법에 대해서 알아봤다.

2.4 Promise.prototype.catch

2.3 Promise.prototype.then에서 이미 `catch()`를 사용해 봤다. 다시 말하면 `catch()`는 `promise.then(undefined, onRejected)`의 랩핑 함수다. 즉, `promise` 객체가 `Rejected` 상태가 됐을 때 호출될 콜백 함수를 등록하기 위한 메서드다.

[그림 2-4] Promise polyfill 지원 상태

앞 그림은 polyfill을 사용한 상태에서 다음 예제 코드가 제대로 실행되는지 나타낸 것이다.

> **Polyfill**
>
> polyfill은 특정 기능을 지원하지 않는 브라우저에서도 그 기능을 사용할 수 있도록 하는 라이브러리를 뜻한다.

[예제](소스코드: http://jsbin.com/yamapa/1/embed?js,console)

```
var promise = Promise.reject(new Error("message"));

promise.catch(function (error) {
    console.error(error.message); // message
});
```

앞의 예제는 IE8 이하의 브라우저에서 "식별자를 찾을 수 없습니다."라는 메시지와 함께 Syntax Error가 발생한다. 에러가 발생하는 원인은 catch가 ECMAScript의 예약어기 때문이다. ECMAScript3에서는 예약어를 객체의 프로퍼티명으로 사용할 수 없다. IE8 이하 버전은 ECMAScript3를 따르는 브라우저이기 때문에 promise.catch()을 사용할 수 없다. 현대 브라우저 대부분이 따르고 있는 ECMAScript5는 예약어를 identifierName 즉, 프로퍼티명으로 사용할 수 있다.

ECMAScript5와 예약어

ECMAScript5에서도 예약어는 Identifier 즉, 변수명, 함수명으로 사용할 수 없다. 예를 들어 for를 변수명으로 사용하면 for 문과 구별할 수 없게 된다. 프로퍼티의 경우는 object.for와 for 문을 구별할 수 있기 때문에 문제되지 않는다.

ECMAScript3의 예약어 문제를 회피할 방법이 있다. 점 표기법은 프로퍼티명이 유효한 식별자가 아니면 안 된다. 하지만 대괄호 표기법은 유효한 식별자가 아니라도 사용할 수 있다. 위 코드를 다음과 같이 작성하면 IE8에서도 동작한다. 물론 polyfill이 필요하다.

[예제](소스코드: http://jsbin.com/bipahu/1/embed?js,console)

```
var promise = Promise.reject(new Error("message"));

promise["catch"](function (error) {
    console.error(error.message); // message
});
```

또는 단순하게 catch() 대신 then()을 사용하여 회피할 수 있다.

[예제](소스코드: http://jsbin.com/fabese/1/embed?js,console)

```
var promise = Promise.reject(new Error("message"));
```

```
promise.then(undefined, function (error) {
    console.error(error.message); // message
});
```

catch를 식별자로 사용하면 문제가 될 수 있기 때문에 일부 라이브러리는 caught, fail 등과 같은 이름으로 catch()와 같은 동작을 하는 메서드를 지원한다. 서비스 정책상 IE8 이하를 지원해야 한다면 catch 문제를 조심해야 한다.

2.5 Promise.all

Promise.all()은 promise 객체를 배열로 전달받고 객체의 상태가 모두 Fulfilled 됐을 때 then()으로 등록한 함수를 호출한다. 2.3 Promise.prototype.then에서 작성한 복수의 XHR을 처리하는 비동기 로직은 Promise.all()을 이용하면 간단하게 작성할 수 있다.

[예제](소스코드: http://jsbin.com/perij/1/embed?js,console)

```
var request = {
  infomation: function getComment() {
    return getURL('http://httpbin.org/get').then(JSON.parse);
  },
  cookie: function getPeople() {
    return getURL('http://httpbin.org/cookies').then(JSON.parse);
  }
};

function main() {
    return Promise.all([request.infomation(), request.cookie()]);
}

main().then(function (value) {
    console.log(value);
```

```
}).catch(function(error){
    console.log(error);
});
```

Promise.all()을 사용하여 main()의 로직을 단순화했다. request.information()과 request.people()는 차례대로 실행되는 게 아니라 동시에 실행되며 결과 값의 순서는 Promise.all()에 전달한 배열 순서와 같다.

```
main().then(function(results){
    console.log(results); // [information, cookie]
});
```

Promise.all()에 전달한 promise 객체가 동시에 실행된다는 사실은 타이머를 사용한 예제를 작성해 보면 쉽게 알 수 있다.

[예제](소스코드: http://jsbin.com/hogiye/1/embed?js,console)

```
// delay 밀리세컨트 후 resolve
function timerPromisefy(delay) {
    return new Promise(function (resolve) {
        setTimeout(function () {
            resolve(delay);
        }, delay);
    });
}

var startDate = Date.now();

// 모두 resolve 되면 종료
Promise.all([
    timerPromisefy(1),
```

```
    timerPromisefy(32),
    timerPromisefy(64),
    timerPromisefy(128)
]).then(function (values) {
    console.log(Date.now() - startDate + 'ms'); // 총 128ms
    console.log(values); // [1,32,64,128]
});
```

timerPromisefy()는 인자로 지정한 ms 후 Fulfilled 상태가 되는 promise 객체 생성해 반환한다. 이 함수를 이용해 여러 개의 promise 객체를 배열에 담아 Promise.all()에 전달한다.

```
var promises = [
    timerPromisefy(1),
    timerPromisefy(32),
    timerPromisefy(64),
    timerPromisefy(128)
];
```

앞의 경우엔 1, 32, 64, 128ms 후에 각각 resolve()가 호출된다. 즉, 이 객체의 상태가 모두 Fulfilled되려면 최소한 128ms가 필요하다. 실제로 Promise.all()을 실행해보면 약 128ms가 소요된다. 만약 차례대로 처리된다면 1, 32, 64, 128 밀리 세컨드 순으로 대기하기 때문에 모두 완료되기까지 약 225ms가 소요될 것이다. 여러개의 promise를 차례대로 처리하는 방법은 5.8 Promise를 사용한 순차 처리에서 소개하겠다.

2.6 Promise.race

이번에는 Promise.race()를 살펴본다. 사용 방법은 Promise.all()과 마찬가지

로 promise 객체를 배열로 전달한다. Promise.all()은 전달한 객체의 상태가 모두 Fulfilled될 때까지 기다리지만 Promise.race()는 전달한 객체 중 하나만 완료되어도 다음 동작으로 넘어간다.

[예제](소스코드: http://jsbin.com/fuxeq/1/embed?js,console)

```
function timerPromisefy(delay) {
    return new Promise(function (resolve) {
        setTimeout(function () {
            resolve(delay);
        }, delay);
    });
}

// 1개라도 resolve 또는 reject 되면 실행
Promise.race([
    timerPromisefy(1),
    timerPromisefy(32),
    timerPromisefy(64),
    timerPromisefy(128)
]).then(function (value) {
    console.log(value); // 1
});
```

앞의 코드에서 promise 객체는 약 128ms 후 모두 Fulfilled되지만 객체 중 하나가 1ms 후 Fulfiled된 순간 then()으로 등록한 콜백 함수가 호출된다. 또 resolve(1)가 호출되기 때문에 콜백 함수에 전달되는 값도 1이다. 그럼 최초로 Fulfilled된 객체 이외의 객체도 동작하는지 살펴보자.

[예제](소스코드: http://jsbin.com/vodego/1/embed?js,console)

```
var winnerPromise = new Promise(function(resolve){
```

```
  setTimeout(function(){
    console.log('this is winner');
    resolve('this is winner');
  }, 4);
});

var loserPromise = new Promise(function(resolve){
  setTimeout(function(){
    console.log('this is loser');
    resolve('this is loser');
  }, 1000);
});

// 맨 처음의 promise 객체가 resolve 되면 종료
Promise.race([winnerPromise, loserPromise]).then(function(value){
   console.log(value); // 'this is winner'
});

// this is winner
// this is winner
// this is loser
```

---

this is winner, this is winner가 출력된 후 loserPromise()의 setTimeout()이 실행되어 this is loser가 출력된다. 즉, Promise.race는 첫 번째 Promise 객체가 Fulfilled되더라도 다른 promise 객체를 취소하지 않는다.

ES6 Promise 사양에는 취소라는 개념이 없다. resolve() 또는 reject()를 사용한 상태 변경만 고려한다. 즉, 상태 변경이 있을지 알 수 없는 경우에는 Promise를 사용해선 안 된다. 일부 라이브러리에서는 취소 기능을 제공해주기도 한다.

2.7 정리

여기까지 Fulfilled 또는 Rejected 상태의 promise 객체를 반환하는 Promise.resolve()와 Promise.reject(), promise 인스턴스 메서드인 then()과 catch(), 복수의 promise 객체를 다룰 수 있는 Promise.all()과 Promise.race()를 다양한 예제와 함께 소개했다. 예제를 설명하면서 Promise의 코드 처리 흐름과 예외 처리 메커니즘에 대해서도 이야기했다. 이번 장의 마지막 절에서 언급했듯이 Promise는 상태 변경이 있을지 알 수 없는 경우에는 적합하지 않다. 즉, Promise는 모든 비동기 처리를 해결할 수 있는 은탄환이 아니다. 이점을 명심하도록 하자. 다음 장에서는 Promise를 사용할 때 꼭 알아야 할 특징 몇 가지를 소개한다.

3 | Promise 특징

Promise는 추상화하는 로직이 동기적일지라도 항상 비동기로 처리되며 then()은 매번 새로운 promise 객체를 생성해 반환한다. promise 객체의 상태가 한 번 변화되면 다시 사용할 수 없지만 새로운 객체가 생성돼 반환되므로 체인할 수 있다. 이처럼 Promise에는 잘 알려져 있지 않은 중요한 특징이 있다. 이번 장에서는 Promise를 다루는 데 있어 꼭 알아야 할 특징 몇 가지를 소개하겠다.

3.1 항상 비동기로 처리되는 Promise

Promise.resolve()나 resolve()를 사용하면 promise 객체는 바로 Fulfilled 상태가 되기 때문에 then()으로 등록한 콜백 함수도 동기적으로 호출될 것으로 생각할 수 있다. 하지만 실제로는 then()으로 등록한 콜백 함수는 비동기적으로 호출된다.

[예제](소스코드: http://jsbin.com/cexuxe/1/embed?js,console)

```
var promise = new Promise(function (resolve){
    console.log("inner promise");
    resolve(42);
});

promise.then(function(value){
    console.log(value);
});

console.log("outer promise");

// inner promise
```

```
// outter promise
// 42
```

코드는 위에서부터 해석되므로 inner promise가 먼저 출력된다. 그리고 resolve(42)가 실행되고 promise 객체의 상태는 Fulfilled가 된다. 그다음 promise.then()이 실행되어 콜백 함수가 등록된다. 이 부분이 이번 절에서 설명할 부분이다. promise 객체의 상태가 정해져 있기 때문에 promise.then()이 실행되는 시점에 동기적으로 콜백 함수가 호출되어 42가 출력될 것이라 예상된다. 하지만 ES6 Promises 사양에는 promise 객체의 상태가 이미 정해져 있다 하더라도 비동기로 처리돼야 한다고 정하고 있다. 따라서 outer promise가 먼저 출력된 후 콜백 함수가 호출되어 42가 출력된다.

이처럼 동기적으로 처리 가능한 상황에서도 비동기적으로 처리하는 이유는 동기와 비동기가 혼재되면 여러 가지 문제가 발생하기 때문이다. 이것은 Promise 이외의 상황에서도 자주 발생하는 문제다. 이해를 돕기 위해 Promise가 아닌 일반적인 상황을 예로 설명하겠다.

[예제](소스코드: http://jsbin.com/caroya/1/embed?js,console)

```
function onReady(fn) {
    var readyState = document.readyState;

    if (readyState == 'interactive' || readyState === 'complete') {
        fn();
    } else {
        window.addEventListener('DOMContentLoaded', fn);
    }
}

onReady(function () {
```

```
    console.log('DOM fully loaded and parsed');
});

console.log('==Starting==');
```

onReady()는 DOM Load가 완료된 상황이면 동기적으로 함수를 호출하고, DOM Load가 완료되지 않은 상황이면 DOMContentLoaded 이벤트 핸들러로 함수를 등록하여 비동기적으로 호출한다. 따라서 이 코드는 상황에 따라 콘솔에 출력되는 메시지의 순서가 바뀐다. 즉, 연산의 순서가 바뀌는 것이다. 이 문제는 항상 비동기로 함수가 호출되도록 수정함으로써 해결할 수 있다.

[예제](소스코드: http://jsbin.com/dobuge/1/embed?js,console)

```
function onReady(fn) {
    var readyState = document.readyState;

    if (readyState == 'interactive' || readyState === 'complete') {
        setTimeout(fn, 0);
    } else {
        window.addEventListener('DOMContentLoaded', fn);
    }
}

onReady(function () {
    console.log('DOM fully loaded and parsed');
});

console.log('==Starting==');
```

데이비드 허먼의 『Effective JavaScript』의 아이템 67 비동기 콜백을 절대 동기적으로 호출하지 마라에서 이 문제에 대해 자세히 소개하고 있다.

- 데이터를 즉시 사용할 수 있더라도, 절대로 비동기 콜백을 동기적으로 호출하지 마라.
- 비동기 콜백을 동기적으로 호출하면 기대한 연산의 순서를 방해하고, 예상치 않은 코드의 간섭을 초래할 수 있다.
- 비동기 콜백을 동기적으로 호출하면 스택 오버플로우나 처리되지 않는 예외를 초래할 수 있다.
- 비동기 콜백을 다른 턴에 실행되도록 스케줄링하기 위해 setTimeout 같은 비동기 API를 사용하라.

이처럼 동기와 비동기를 혼재했을 때 발생하는 문제를 예방하기 위해 항상 비동기로 처리하도록 ES6 Promises 사양이 정해진 것이다. 항상 비동기로 처리되기 때문에 명시적으로 비동기 처리를 위한 코드를 추가로 작성할 필요가 없다.

[예제](소스코드: http://jsbin.com/galejo/1/embed?js,console)

```
function onReadyPromise() {
    return new Promise(function (resolve, reject) {
        var readyState = document.readyState;

        if (readyState == 'interactive' || readyState === 'complete') {
            resolve();
        } else {
            window.addEventListener('DOMContentLoaded', resolve);
        }
    });
}

onReadyPromise().then(function () {
    console.log('DOM fully loaded and parsed');
});

console.log('==Starting==');
```

3.2 새로운 promise 객체를 반환하는 then

promise.then(…).catch(…)는 언뜻 모두 최초의 promise 객체에 메서드를 체인하는 것처럼 보인다. 그러나 실제로는 then()과 catch()는 새로운 promise 객체를 생성해 반환한다. 정말 새로운 promise 객체를 생성해 반환하는지 확인해보자.

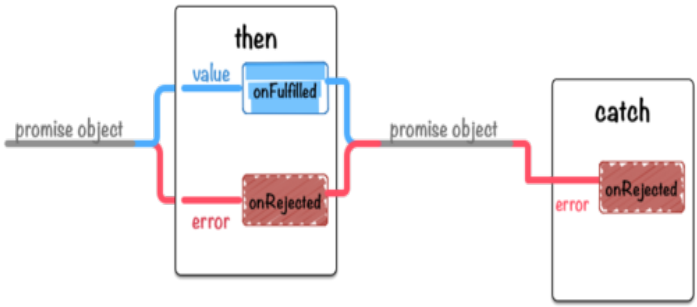
[예제](소스코드: http://jsbin.com/goluxu/1/embed?js,console)

```
var promise = new Promise(function (resolve) {
    resolve(100);
});

var thenPromise = promise.then(function (value) {
    console.log(value);
});

var catchPromise = thenPromise.catch(function (error) {
    console.error(error);
});

console.log(promise === thenPromise); // false
console.log(thenPromise === catchPromise); // false
```

일치 연산자(===)를 이용해 객체를 비교한다. 그리고 서로 다른 객체이기 때문에 false가 출력된다. 간단한 비교 연산으로 then()이나 catch()는 새로운 promise 객체를 반환한다는 사실을 확인할 수 있다.

이 구조를 의식하지 않고 개발하면 자신도 모르는 사이 전혀 다른 promise 객체를 다루게 되는 일이 발생한다. 만약 then()이 새로운 객체를 생성하여 반환한다는 것을 알고 있다면 다음 예제가 왜 의도한 대로 동작하지 않는지 알 수 있을 것이다.

[그림 3-1] promise 객체의 생명 주기



[예제](소스코드: http://jsbin.com/siwazo/1/embed?js,console)

```
// 1: then()으로 등록한 함수가 동시에 호출된다.
var aPromise = new Promise(function (resolve) {
    resolve(100);
});

aPromise.then(function (value) {
    return value * 2;
});

aPromise.then(function (value) {
    return value * 2;
});

aPromise.then(function (value) {
    console.log("1: " + value); // => 100
});

// 2: then()으로 등록한 함수가
//    promise 체인의 순서대로 호출된다.
var bPromise = new Promise(function (resolve) {
    resolve(100);
```

```
});

bPromise.then(function (value) {
    return value * 2;
}).then(function (value) {
    return value * 2;
}).then(function (value) {
    console.log("2: " + value); // 100 * 2 * 2
});
```

첫 번째 방식은 promise를 체인으로 연결하지 않았다. 이 경우 then()으로 등록한 각각의 콜백 함수는 동시에 호출되므로 value의 값도 모두 같은 100이다. 두 번째 방식은 promise 체인으로 작성했으므로 resolve → then → then → then 순서대로 실행되며 value 역시 순서대로 연산 후 전달된다. 첫 번째 방식은 안티패턴으로써 다음과 같이 작성하는 경우가 많다.

```
function anAsyncCall() {
    var promise = Promise.resolve();

    promise.then(function() {
        // something do...return newVar;
    });

    return promise;
}
```

앞과 같이 작성하면 then()을 처리 중 어떤 오류가 발생했을 때 감지할 방법이 없다. 또한, 특정 결과값을 반환하더라도 전달받을 수 없다. 이 코드는 then()에 의해 새롭게 생성된 promise 객체를 반환하도록 수정해야 한다. 그래야 promise 체인이 가능하다.

```
function anAsyncCall() {
    var promise = Promise.resolve();

    return promise.then(function() {
        // do something...return newVar;
    });
}
```

이 동작 방식은 Promise 전반에 걸쳐 해당한다. 2장 Promise 사용하기에서 설명한 Promise.all()이나 Promise.race() 역시 새로운 promise 객체를 생성해 반환한다. Promise 안티패턴에 대해 자세히 알고 싶다면 Promise Anti-patterns를 참고한다.

3.3 예외 처리가 되지 않는 onRejected

이 도서에서는 기본적으로 catch()를 사용하여 오류를 처리하도록 예제가 작성돼 있다. 이번엔 then()만 사용했을 때 발생할 수 있는 문제에 관해서 이야기하겠다.

[예제](소스코드: http://jsbin.com/xoniwi/1/embed?js,console)

```
function throwError(value) {
    // 예외 발생
    throw new Error(value);
}

// <1> onRejected가 호출되지 않는다.
function badMain(onRejected) {
    return Promise.resolve(42).then(throwError, onRejected);
}

// <2> 예외 발생시 onRejected가 호출된다.
```

```
function goodMain(onRejected) {
    return Promise.resolve(42).then(throwError).catch(onRejected);
}

// 실행
badMain(function(){
    console.log("BAD");
});

goodMain(function(){
    console.log("GOOD");
});
```

예외 처리가 되지 않는 badMain()과 그 반대의 경우인 goodMain()이 있다. badMain()은 then()의 두 번째 인자로 오류 발생 시 동작할 함수 onRejected()를 등록하고 있다. 하지만 첫 번째 인자로 등록한 throwError() 내에서 오류가 발생해도 onRejected()가 호출되지 않기 때문에 오류를 감지할 수 없다. 반면 goodMain()은 throwError()가 호출된 후 onRejected()가 호출되도록 작성돼 있다. 이 경우는 throwError()에서 오류가 발생하면 다음 체인인 catch()가 호출되므로 예외 처리를 할 수 있다.

onRejected()의 대상은 then()을 이용해 등록한 throwError()가 아니라 이전의 promise 객체에 대한 처리이므로 이와 같은 차이가 발생하는 것이다. 다시 말해 then()이나 catch()는 실행될 때마다 새로운 promise 객체를 생성해 반환한다. 즉, 체인할 때마다 새로운 promise 객체의 상태에 대한 처리를 등록하는 것이다. 따라서 Promise.resolve(42)에 대한 처리로 등록한 onFulfilled에서 발생한 예외는 다음 체인인 catch()에서 감지할 수 있다.

catch()는 then()을 단순히 랩핑한 메서드이므로 다음과 같이 then()을 사용해

서 표현할 수 있다. 하지만 catch()를 이용하는 게 개발자의 의도를 조금 더 명확하게 드러낸다.

```
Promise.resolve(42).then(throwError).then(null, onRejected);
```

3.4 콜백-헬과 무관한 Promise

이번에는 많은 개발자가 오해하고 있는 사실을 이야기한다. 보통 콜백-헬 을 해결할 방법으로 Promise를 소개하는 경우가 3많다. 하지만 엄밀히 말하면 콜백-헬을 해결할 수 없고 일부 완화할 수 있을 뿐이다. 완화(緩和)와 해결(解決)은 전혀 다른 관점이다. 그렇다면 콜백-헬을 완화하는 게 Promise의 목적일까?

역자가 생각할 때 콜백-헬을 완화할 수 있는 가장 큰 이유는 단일 인터페이스와 명확한 비동기 시점 표현, 강력한 에러 처리 메커니즘 때문이다. 하지만 이것은 비단, 콜백-헬 뿐만 아니라 비동기 처리 자체를 손쉽게 다룰 수 있도록 하는 것이므로 콜백-헬을 해결하는 방법으로 여기는건 바람직하지 않다.

이해를 돕기 위해 전통적인 콜백 패턴으로 코드를 작성해 보겠다.

[예제](소스코드: http://jsbin.com/sovipi/1/edit?js,console)

```
/**
 * 커피를 주문한다.
 * @param {string} menu
 * @param {function} callback
 */
function order(menu, callback){
  getGreenBeans(menu, function(err, greenBeans){
    if(err){
      callback(err);
```

```
    }else{
      doRoasting(greenBeans, function(err, blackBeans){
        if(err){
          callback(err);
        }else{
          createCoffee(menu, blackBeans, function(err, coffee){
            callback(err, coffee);
          });
        }
      });
    }
  });
}

order('아메리카노', function(err, coffee){
  if(err){throw err;}
  console.log(coffee + '를 마신다.');
});
```

order()는 getGreenBeans(), doRoasting(), createCoffee() 비동기 함수를 사용한다. 흔히 볼 수 있는 콜백-헬이다. 콜백-헬을 해결하고자 하는 목적을 가지고 Promise를 사용한다면 다음과 같이 코드를 작성하게 된다.

[예제](소스코드: http://jsbin.com/coxone/1/edit?js,console)

```
/**
 * 커피를 주문한다.
 * @param {string} menu
 * @returns {Promise}
 */
function order(menu){
```

```
  return getGreenBeans(menu).then(function(greenBeans){
    return doRoasting(greenBeans);
  }).then(function(backBeans){
    return createCoffee(menu, backBeans);
  });
}

order('아메리카노').then(function(coffee){
  console.log(coffee + '를 마신다.');
}).catch(function(err){
  console.log(err.message);
});
```

이제 order()는 단순히 Promise 객체를 반환한다. 커피전문점에서 커피를 주문하면 전자벨을 주는 것과 비슷하다. 그리고 비동기 로직은 전부 Promise에 추상화된다. 앞의 코드는 중첩되지 않는 형태로 변환했을 뿐 콜백-헬의 문제를 해결했다고 보기 어렵다.

Promise는 미래 어느 시점이 되면 값을 채워주기로 약속한 빈 그릇이며 비동기 처리를 추상화한 추상 컨테이너다. 즉, 통일된 인터페이스로 데이터를 전달할 수 있는 컨테이너로써 장점을 발휘하는 것이다. 중첩을 해결하고자 promise 체인을 길게 연결하는 것은 외형의 느낌만 다를 뿐 콜백-헬과 큰 차이가 없다.

```
function somePromise(){
    return Promise.reject(true).then(function(value){
        // something…
    }).then(function(value){
        // something…
    }).then(function(value){
        // something…
```

```
    }).then(function(value){
        // something…
    }).catch(function(err){
        // something…
    });
}
```

본질적으로 전통적인 콜백 패턴과 Promise 패턴이 해결하고자 하는 문제는 같다. 두 패턴 모두 비동기 처리를 손쉽게 다루기 위함이다. 즉, 비동기 처리를 다루는 방법이 두 가지이지 모든 경우를 Promise로 프로그래밍할 수 있다고 생각하는 것은 옳지 않다. 이벤트 리스너나 Stream처럼 정기적, 지속적으로 비동기 처리가 필요한 경우 Promise를 사용하면 오히려 이상적인 결과를 얻을 수 없다. 또한, 강력한 에러 처리 메커니즘이 오히려 독이 되는 경우도 더러 있다.

깊게 중첩되는 콜백이나 길게 연결되는 Promise 모두 함수 하나의 책임이 거대하다는 뜻이다. 즉, 분석 및 설계가 전혀 이루어지지 않았고 구조가 망가져 있음을 뜻한다. 따라서 콜백이 깊게 중첩되는 경우는 설계를 먼저 의심해봐야 한다. 앞에서 작성한 커피 예제를 다시 살펴보겠다.

손님이 주문order할 때 처리되는 순서를 생각해보자. 먼저 메뉴에 해당하는 생두greenBeans를 가져온다. 그리고 생두를 로스팅Roasting하여 원두blackBeans로 만든다. 마지막으로 원두를 사용해 메뉴에 해당하는 커피를 만들어 전달한다. 이 행위를 분석하면 총 3가지 객체를 추출할 수 있다. 손님이 주문하기 위한 커피전문점coffeehouse, 로스팅을 위한 로스터기roaster, 생두를 보관하는 창고beansStorage다. 먼저 생두 보관 창고 beansStorage 객체를 작성해보자.

```
/**
 * 생두 저장소 객체
 * @namespace
```

```
 */
var beansStorage = {

  /**
   * 메뉴에 해당하는 생두를 반환한다.
   * @param {string} menu
   * @returns {Promise}
   */
  get : function(type){
    return new Promise(function(resolve, reject){
      if(type === '케냐'){
        resolve('생두');
      }else{
        reject('알 수 없는 타입입니다.');
      }
    });
  }
}
```

beansStorage 객체에는 타입에 해당하는 생두를 반환하는 get()만 있지만, 생두를 조작하기 위한 여러 메서드가 추가될 수 있다. 두 번째로 생두를 로스팅하여 원두를 반환하는 로스터기roaster 객체를 작성해보자.

```
/**
 * 로스터기 객체
 * @namespace
 */
var roaster = {

  /**
   * 생두를 로스팅하여 원두를 반환한다.
   * @param {string} greenBeans
```

```
   * @returns {Promise}
   */
  execute : function(beanType){
    return beansStorage.get(beanType).then(function(greenBeans){
      return new Promise(function(resolve, reject){
        if(greenBeans === '생두'){
          resolve('원두');
        }else{
          reject('생두가 아닙니다.');
        }
      });
    });
  }
};
```

roater 객체의 execute()는 beansStorage 객체를 통해 생두를 가져와 로스팅하고 최종적으로 원두를 반환한다. 마지막으로 커피전문점coffeehouse 객체를 작성해보자.

```
/**
 * 커피 하우스 객체
 */
var coffeehouse = {

  /**
   * 커피를 주문한다.
   * @param {string} menu
   * @returns {Promise}
   */
  order : function(menu){
    return roaster.execute('케냐').then(function(blackBeans){
      return new Promise(function(resolve, reject){
```

```
        if(menu === '아메리카노' && blackBeans === '원두'){
          resolve('맛있는 아메리카노');
        }else{
          reject(new Error('알 수 없는 메뉴입니다.'));
        }
      });
    });
  }
};
```

coffeehouse 객체의 order()는 roaster 객체를 통해 원두를 가져와 커피를 만들어 손님에게 전달한다. 이제 이 객체를 사용해서 커피를 주문해보자.

[예제](소스코드: http://jsbin.com/wavixa/1/edit?js,console)

```
coffeehouse.order('아메리카노').then(function(coffee){
  console.log(coffee);
}).catch(function(err){
  console.log(err.message);
});
```

책임 및 관심사별로 나눠 추상화하고 객체를 설계했다. Promise의 진정한 장점을 느끼기 위해서는 이렇게 분석 및 설계를 통한 모듈화가 선행돼야 한다. 깊은 콜백 중첩으로 구조가 이미 망가져 있는 곳에 Promise로 해결하려고 노력하지 말기 바란다. 해결할 수도 없을 뿐더러 문제를 더 복잡하게 만들 수 있다.

3.5 정리

이번 장에서는 Promise는 항상 비동기로 처리된다는 사실과 새로운 promise 객체를 반환하는 then()과 catch(), 예외 처리가 되지 않는 상황 등, Promise를 다루

는 데 필요한 몇 가지 특징에 대해서 알아봤다. 특히 promise.then(onFulfilled, onRejected)일 때 onFulfilled에서 발생한 예외는 onRejected에서 감지할 수 없다는 사실을 기억하자. 간단하지만 지나치기 쉬운 부분이다. Then()과 catch()는 본질에서는 큰 차이가 없지만 분리해서 사용하면 개발자의 의도를 조금 더 명확하게 드러낼 수 있다. 다음 장에서는 Promise를 테스트하는 방법에 관해서 이야기하겠다.

4 | Promise 테스트

여기까지 ES6 Promises에 대해 알아봤으므로 Promise를 사용한 실용적인 코드를 작성할 수 있을 것이다. 이제 Promise를 테스트하는 방법에 대해 고민할 차례다. 이번 장에서 Mocha를 사용해 기본적인 Promise 테스트 작성법에 대해 알아보려고 한다. 테스트 코드는 Node.js 환경을 전제로 한다.

4.1 기본적인 테스트 작성법

어설션 라이브러리로 power-assert를 사용하여 예제를 작성했다. 기능은 Node.js의 assert 모듈과 같으므로 특별히 신경 쓰지 않아도 된다. 가장 먼저 콜백 스타일로 테스트를 작성한다.

```
"use strict";
var assert = require("power-assert");

describe("Basic Test", function(){
    context("When Callback(high-order function)", function(){
        it("should use `done` for test", function(done){
            setTimeout(function(){
                assert(true);
                done();
            }, 0);
        });
    });

    context("When promise object", function(){
        it("should use `done` for test?", function(done){
            var promise = Promise.resolve(1);

            // 이 테스트 코드는 문제가 있다.
```

```
            promise.then(function(value){
                assert(value === 1);
                done();
            });
        });
    });
});
```

Mocha의 it()은 콜백 함수의 매개 변수에 done()을 전달한다. 그리고 done()이 호출될 때까지 테스트 케이스를 종료하지 않는 방식으로 비동기 테스트를 지원한다.

```
it("should use `done` for test", function(done){
    setTimeout(function(){
        assert(true);
        done();
    }, 0);
});
```

done()을 사용하는 콜백 함수를 setTimeout()에 전달한다. 이 함수가 비동기적으로 호출되어 done()이 실행되면 테스트가 완료된다. 흔히 볼 수 있는 비동기 테스트 코드다.

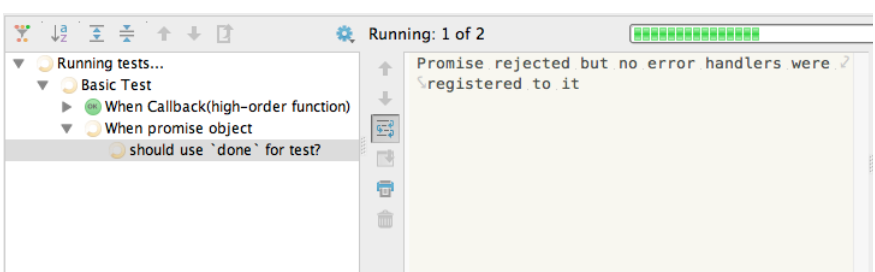
```
it("should use `done` for test?", function(done){
    var promise = Promise.resolve(1);
    promise.then(function (value) {
        assert(value === 1);
        done();
    });
});
```

이번엔 Promise 테스트 코드를 살펴본다. Promise.resolve()는 Fulfilled 상태인 promise 객체를 반환하므로 then()으로 등록한 콜백 함수가 바로 호출된다. 그리고 콜백 함수 내부에서 done()을 실행하는 방식으로 비동기 테스트에 대응하고 있다. 3장 항상 비동기로 처리되는 Promise에서 설명했듯이 promise 객체는 항상 비동기로 처리되므로 테스트 역시 비동기 방식으로 작성해야 한다. 하지만 이 테스트는 어설션이 실패해야 하는 경우 문제가 된다.

```
it("should use `done` for test?", function (done) {
    var promise = Promise.resolve();
    promise.then(function (value) {
        assert(false); // throw AssertionError
        done();
    });
});
```

앞의 코드는 assert()에 false를 전달했기 때문에 테스트가 실패할 것 같지만 실제로는 테스트가 끝나지 않고 타임아웃된다.

[그림 4-1] 비동기 테스트 타임 아웃



테스트 프레임워크는 어설션이 실패한 경우, throw를 발생시켜 테스트가 실패했다

고 판단한다. 그러나 Promise의 경우, 추상화한 로직에서 발생한 오류는 모두 자체적인 예외 처리 메커니즘에 따라 처리되기 때문에 테스트 프레임워크에서 오류를 감지할 수 없다. 그럼 어설션이 실패한 경우를 감지할 수 있도록 예제를 수정해 보자.

```
it("should use `done` for test?", function(done){
    var promise = Promise.resolve();

    promise.then(function(value){
        assert(false);
    }).then(done, done);
});
```

then()을 한 번 더 체인 하여 성공 및 실패한 경우 모두 done()이 호출될 수 있도록 수정했다. 어설션이 성공한 경우에는 done(), 실패한 경우에는 done(error)가 실행된다. 이와 같은 방법으로 일반적인 비동기 테스트와 동일하게 동작하는 Promise 테스트를 작성할 수 있다.

Promise 테스트를 작성하는 것은 까다로운 작업이다. 어설션이 실패할 경우를 대비해 then(done, done)을 잊지 않도록 조심해야 한다.

4.2 Promise를 지원하는 Mocha

이번 절에서는 Mocha가 지원하는 Promise 테스트 기능에 대해 알아본다. 공식 사이트에 Asynchronus code에 대해 개요가 작성되어 있다.

Mocha

Mocha는 Node.js와 브라우저, Promise 환경의 테스트를 지원하는 테스트 프레임워크다. BDD, TDD, exports 스타일 줄 하나를 선택할 수 있고 테스트

에 필요한 어설션 메서드를 임의의 라이브러리와 조합해 사용할 수 있다. 즉, Mocha 자체는 테스트 실행을 위한 틀만 제공할 뿐이며 나머지는 개발자가 직접 선택할 수 있다. 세계적으로 인기 있는 테스트 프레임워크다.

(인용구) "Alternately, instead of using the done() callback, you can return a promise. This is useful if the APIs you are testing return promises instead of taking callbacks:"

Promise를 테스트할 때 done()을 호출하는 대신 promise 객체를 반환할 수 있다. 실제로 테스트 코드를 작성해보자.

```
"use strict";
var assert = require("power-assert");

describe("Promise Test", function(){
    it("should return a promise object", function(){
        var promise = Promise.resolve(1);

        return promise.then(function(value){
            assert(value === 1);
        });
    });
});
```

이전 절에서는 Promise를 테스트하기 위해 then(done, done)을 사용했지만, 이번에는 done()을 제거하고 promise 객체를 반환하도록 변경했다. 이제 어설션이 실패하면 테스트 역시 실패한다.

```
it("should be fail", function () {
    return Promise.resolve().then(function () {
```

```
        assert(false); // 테스트가 실패한다.
    });
});
```

Mocha에서 지원하는 기능을 사용하면 then(done, done)처럼 명시적이지 않고 의미가 불분명한 코드를 제거할 수 있다.

4.3 의도하지 않은 테스트 결과

Mocha가 Promise 테스트를 지원하기 때문에 편리하게 테스트 코드를 작성할 수 있을 것 같지만, 또 다른 문제가 존재한다. promise 객체가 Fulfilled된 경우엔 테스트가 실패하고 Rejected된 경우에 테스트가 성공하는 경우를 살펴보자.

```
// 이 함수가 반환하는 promise객체를 테스트
function mayBeRejected(){
    return Promise.reject(new Error("woo"));
}

it("is bad pattern", function(){
    return mayBeRejected().catch(function(error){
        assert(error.message === "woo");
    });
});
```

앞의 테스트 코드는 mayBeRejecte()는 Rejected 상태인 promise 객체를 생성해 반환하므로 catch()로 등록한 콜백 함수가 실행돼 테스트가 성공한다.

```
function mayBeRejected(){
    return Promise.resolve();
}
```

```
it("is bad pattern", function(){
    return mayBeRejected().catch(function(error){
        assert(error.message === "woo");
    });
});
```

이번엔 Fulfilled 상태인 promise 객체를 생성해 반환한다. 이 경우 catch()로 등록한 콜백 함수가 호출되지 않고 테스트가 성공한다. 즉, 의도한 대로 테스트 케이스가 동작하지 않는다. 이 문제를 해결하기 위해서는 catch() 앞에 then()을 추가해 Fulfilled 상태일 때 테스트가 실패하도록 해야 한다.

```
function failTest(){
    throw new Error("Expected promise to be rejected but it was fulfilled");
}

function mayBeRejected(){
    return Promise.resolve();
}

it("should bad pattern", function(){
    return mayBeRejected().then(failTest).catch(function(error){
        assert.deepEqual(error.message === "woo");
    });
});
```

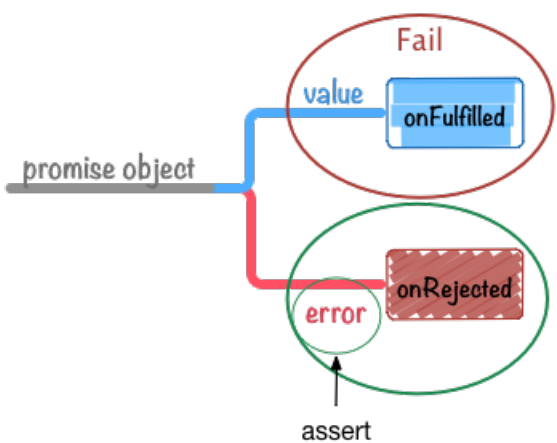
하지만 앞의 방법으로 작성하면 3.3 예외 처리가 되지 않는 onRejected에서 설명했듯이 failTest()에서 오류가 발생한다면 catch()로 등록한 콜백 함수가 호출된다. 따라서 AssertionError가 발생하므로 의도한 상황이라고 볼 수 없다.

이 문제는 then(onFulfilled, onRejected)로 테스트 코드를 수정하여 해결할

수 있다. 이렇게 작성하면 Promise가 실패한 경우에만 어설션으로 테스트를 진행한다. 쉽게 말해 Fulfilled 또는 Rejected 상태에 따라 테스트가 어떻게 진행되길 바라는지 그 의도를 명시할 필요가 있다는 것이다.

```
function mayBeRejected(){
    return Promise.resolve();
}

it("catch -> then", function(){
    // Fulfilled인 경우엔 테스트가 실패한다.
    return mayBeRejected().then(failTest, function(error){
        assert(error.message === "woo");
    });
});
```

[그림 4-2] Promise onRejected 테스트



3.3 예외 처리가 되지 않는 onRejected에서는 오류를 잃어버리는 경우를 방지하고자 then(onFulfilled, onRjected)를 사용하지 말고 then()과 catch()로 적

절히 나눠 작성하도록 권고했다. 하지만 테스트 코드를 작성하는 경우엔 Promise의 강력한 예외 처리 메커니즘이 오히려 테스트를 방해하므로 조금 더 안전하면서 테스트의 의도를 명시적으로 드러낼 수 있는 then(filaTest, onRejected)로 작성하는 편이 좋다.

```
promise.then(failTest, function(error){
    // assert로 error을 테스트
});
```

명시적이긴 하나 다소 직관적이지는 못하다. 다음 절에서 Promise 테스트를 위한 헬퍼 함수를 정의해 조금 더 직관적으로 테스트를 작성하는 방법을 소개하겠다.

4.4 조금 더 직관적으로 테스트 작성

Fulfilled를 기대하는 테스트는 Rejected된 경우와 어설션의 결과가 일치하지 않는 경우 실패한다. Rejected를 기대하는 테스트는 Fulfilled된 경우와 어설션의 결과가 일치하지 않는 경우 실패한다. 그리고 실패 요건에 해당하지 않으면 테스트는 성공한다.

```
promise.then(failTest, function(error){
    // assert로 error를 테스트한다.
    assert(error instanceof Error);
});
```

Promise를 테스트할 때는 Fulfilled와 Rjected 중 어느 상태를 기대하는지, 또 어떤 값을 어설션하는지 직관적으로 드러낼 필요가 있다. 따라서 기대하는 상태를 직관적으로 알 수 있도록 하는 헬퍼 함수를 정의한다.

```
azu/promise-test-helper
```

이번 절에서 작성하는 promise 헬퍼 함수는 이 책의 저자인 azu가 라이브러리로 제공하고 있다.

가장 먼저 Rejected 상태를 기대하는 테스트를 위한 shouldRjected 헬퍼 함수를 정의한다.

```
var assert = require('power-assert');

function shouldRejected(promise){
    return {
        'catch': function(fn){
            return promise.then(function(){
                throw new Error('Expected promise to be rejected but it was
fulfilled');
            }, function (reason){
                fn.call(promise, reason);
            });
        }
    };
}

it('should be rejected', function(){
    var promise = Promise.reject(new Error('human error'));

    return shouldRejected(promise).catch(function(error){
        assert(error.message === 'human error');
    });
});
```

shouldRejected()에 promise 객체를 전달하면 catch()를 가진 리터럴 객체를 반환한다. 이 catch()는 promise의 catch()처럼 사용할 수 있으며 Fulfilled 상태일 때 예외를 발생시킨다. shouldRejected()에 promise 객체를 전달한다는 점을 제외하면 promise를 테스트할 때와 같은 느낌이다.

```
promise.then(failTest, function(error){
    assert(error.message === 'human error');
});

// 위 코드와 같은 테스트 케이스
shouldRejected(promise).catch(function(error){
    assert(error.message === 'human error');
});
```

이처럼 헬퍼 함수를 만들어 사용하면 조금 더 직관적으로 테스트 코드를 작성할 수 있다. 이번엔 Fulfilled 상태를 기대하는 테스트를 위한 shouldFulfilled()를 작성한다.

```
var assert = require('power-assert');

function shouldFulfilled(promise) {
    return {
        'then': function(fn){
            return promise.then(function(value){
                fn.call(promise, value);
            }, function(reason){
                throw reason;
            });
        }
    };
```

```
}

it('should be fulfilled', function(){
    var promise = Promise.resolve('value');

    return shouldFulfilled(promise).then(function(value){
        assert(value === 'value');
    });
});
```

반환하는 객체의 메서드가 catch()에서 then()으로 바뀌었을 뿐 shouldRjected()와 비슷하다.

4.5 정리

Promise의 강력한 예외 처리 메커니즘이 오히려 테스트 작성 시에는 문제가 될 수 있으므로 주의해야 한다. shouldFulfilled()와 shouldRejected()는 실무에서 바로 사용할 수 있다. 하지만 Mocha의 Promise 지원 환경을 전제로 했기 때문에 done()을 사용해야 하는 경우에는 사용할 수 없다. Promise를 지원하는 테스트 프레임워크의 기능을 사용하는 것과 done()을 사용할 것 중 어느 방법이 더 좋은지는 알 수 없다. 예를 들어 SoffeeScript 의 경우에는 암묵적 return 기능이 있기 때문에 done()을 사용하는 것이 오히려 읽기 쉬울 수 있다. 상황에 따라 적절한 방법을 찾아야 한다. 하지만 일관성만큼은 유지할 수 있도록 하자. 이번 장에서는 Promise를 테스트하는 방법을 일반적인 비동기 테스트 코드와 비교해서 알아봤다. 다음 장에서는 Promise를 조금 더 심층적으로 다룬다.

5 | Promise 고급

Promise로 Web Notification API를 어떻게 다룰 수 있는지, throw 대신 reject를 사용해야 하는 이유, Deferred와 Promise의 관계 등 이번 장에서는 지금까지 배운 내용을 응용하거나 조금 더 깊게 다루도록 하겠다. 가장 먼저 서드 파티로 구현된 Promise 호환 라이브러리에 관해서 이야기한다.

5.1 Promise 라이브러리

모든 브라우저에서 ES6 Promises를 구현하고 있지 않기 때문에 라이브러리가 필요하다. 따라서 이번 절에서 크게 두 종류의 라이브러리를 소개하고자 한다. 첫 번째는 Polyfill이고 두 번째는 Promise/A+를 호환하며 독자적으로 확장한 라이브러리다.

Promise 라이브러리를 선택할 때 고려해야 할 사항으로는 Promises/A+ 호환성이 있다. Promises/A+는 ES6 Promises의 전신으로 then()에 대한 스펙을 정의한 사양이다. 보통 Promise/A+를 호환하는 경우 then()의 동작이 같으면서 catch(), Promise.all() 등과 같은 기능도 구현하고 있을 가능성이 높다. 하지만 Promise/A+는 Promise.prototype.then()에 대한 사양만 책정하고 있으므로 다른 기능이 구현돼 있다 하더라도 이름과 동작이 다를 수 있다. 일단 then()의 동작이 같다는 말은 곧 thenable하다는 것이므로 Promise.resolve()를 사용하여 ES6 Promises 사양의 promise 객체로 변환할 수 있다는 뜻이다.

가장 먼저 Polyfill 라이브러리를 소개하겠다. Polyfill은 Promise API를 ES6 Promises의 사양과 같은 인터페이스로 제공하는 라이브러리다. 이 라이브러리를 사용하면 IE10과 같이 Promise를 지원하지 않는 브라우저에서도 Promise를 사용할 수 있다.

- **jakearchibald/es6-promise** : Promise/A+를 호환하는 RSVP.js를 기반으로 개발한 ES6 Promises Polyfill 라이브러리다.
- **yahoo/ypromise** : YUI에서 사용하고 있는 Polyfill 라이브러리다.
- **getify/native-promise-only** : ES6 Promises 사양을 엄격하게 따르는 Polyfill 라이브러리다. 만약 브라우저에 네이티브 Promise가 있다면 그것을 사용한다.

이번에는 Promise 사양을 구현하면서 독자적인 기능도 확장한 라이브러리를 소개하겠다. Promise 라이브러리는 시중에 많이 있지만 가장 인기 있는 라이브러리만 소개한다.

- **kriskowal/q** : Q는 Promises와 Deferreds를 구현한 라이브러리다. 2009년부터 개발됐으며 Node.js 환경을 위한 파일 API를 제공한다. 또한, 다양한 환경에서도 사용할 수 있도록 Q-IO 같은 인터페이스도 제공하고 있다.
- **petkaantonov/bluebird** : Promise 기능은 물론, 취소 기능을 제공하고 진행 정도도 알 수 있으며 에러 핸들링 등 여러 가지 기능을 확장하고 있고, 성능에도 많은 신경을 쓰고 있는 라이브러리다.

Q와 Bluebird 모두 브라우저에서 사용할 수 있다. 그리고 잘 정리된 API 레퍼런스 문서도 제공한다.

- API Reference - kriskowal/q Wiki

Coming from jQuery에는 Q의 Deferred와 jQuery의 Deferred를 비교하고 마이그레이션하는 경우에 대해서 잘 정리돼 있으니 참고 바란다.

- bluebird/API.md at master - petkaantonov/bluebird

Bluebird 문서에는 Promise를 사용한 사용 예제가 풍부하고 에러가 발생했을 때

의 대처법과 안티패턴에 대해서도 소개하고 있다. 두 라이브러리 모두 문서의 수준이 뛰어나기 때문에 사용하지 않더라도 참고할 만한 내용이 많이 있다.

Polyfill과 Promise 라이브러리 모두 Promise/A+ 혹은 ES6 Promises의 공통 인터페이스를 구현하고 있다. 따라서 라이브러리를 사용하더라도 본질적인 개념을 참고해야 하는 경우가 많다.

5.2 Promise.resolve와 Thenable

이번 장은 Web Notification 소개와 Web Notification 랩핑 두 절로 나눠서 Promise.resolve의 큰 특징의 하나인 `thenable` 객체를 `promise` 객체로 변환하여 사용하는 방법을 소개하고 마지막 절에서 `thenable`에 대해 이야기하겠다.

5.2.1 Web Notification 소개

Web Notification은 데스크탑 알람 기능을 제공하는 API다. 더 자세한 정보는 다음 문서를 참고한다.

- Using Web Notifications | MDN
- Can I use Web Notifications

다음과 같이 작성해 알람을 출력할 수 있다.

```
new Notification("Hi!");
```

하지만 알람 하기 위해선 인스턴스를 생성하기 전에 사용자에게 허가를 받아야 한다.

[그림 5-1] 알람 허가 창

알람 허가 창에서 선택한 결과는 Notification.permission으로 알 수 있다. 값은 허가granted 또는 불허denied 두 가지다. 알람 허가 창은 `Notification.requestPermission()`으로 호출할 수 있으며 사용자가 선택한 결과를 콜백 함수의 `status` 매개 변수로 알 수 있다. 콜백 함수라는 점에서 비동기임을 알 수 있다.

[예제](소스코드: http://jsbin.com/haroya/1/embed?js,console)

```
Notification.requestPermission(function (status) {
    // status로 "granted" 또는 "denied"가 전달된다.
    console.log(status);
});
```

사용자가 알람을 허가한 경우, `new Notification()` 구문으로 알람을 할 수 있다. 이미 허가한 경우도 마찬가지다. 만약 허락하지 않으면 아무 동작도 하지 않는다. 이 상황을 다음과 같이 Promise의 `Fulfilled`와 `Rejected`로 매핑시킬 수 있다.

- resolve(성공) == 허가('granted') : `onFulfilled`가 호출된다.
- reject(실패) == 불허('denied') : `onRejected`가 호출된다.

이 경우를 우선 Promise가 아닌 전통적인 콜백 스타일로 작성한다.

5.2.2 Web Notifications 랩핑

Web Notifications API를 전통적인 콜백 스타일로 랩핑한다.

[예제](소스코드: http://jsbin.com/yugigo/1/embed?js,console)

```
function notifyMessage(message, options, callback){
    var notification = null;

    if (Notification && Notification.permission === 'granted'){
        notification = new Notification(message, options);
        callback(null, notification);
    }else if (Notification.requestPermission){
        Notification.requestPermission(function(status){
            if(Notification.permission !== status){
                Notification.permission = status;
            }
            if(status === 'granted'){
                notification = new Notification(message, options);
                callback(null, notification);
            }else{
                callback(new Error('user denied'));
            }
        });
    }else{
        callback(new Error('doesn\'t support Notification API'));
    }
}

notifyMessage("Hi!", {}, function(error, notification){
    if(error){
        return console.error(error);
    }
```

```
    console.log(notification); // notification 객체
});
```

사용자가 허가하면 `notifyMessage()`는 알람을 실행하고 콜백 함수에 `notification` 객체가 전달된다. 만약 허가하지 않으면 `error` 객체가 전달된다.

```
function callback(error, notification){
    // do something...
}
```

다음으로 전통적인 콜백 스타일로 작성한 함수를 Promise로 재작성한다.

[예제](소스코드: http://jsbin.com/duneri/1/embed?js,console)

```
function notifyMessage(message, options, callback){
    var notification = null;

    if(Notification && Notification.permission === 'granted'){
        notification = new Notification(message, options);
        callback(null, notification);
    }else if(Notification.requestPermission){
        Notification.requestPermission(function(status){
            if(Notification.permission !== status){
                Notification.permission = status;
            }

            if(status === 'granted'){
                notification = new Notification(message, options);
                callback(null, notification);
            }else{
                callback(new Error('user denied'));
```

```
            }
        });
    }else{
        callback(new Error('doesn\'t support Notification API'));
    }
}

function notifyMessageAsPromise(message, options){
    return new Promise(function(resolve, reject){
        notifyMessage(message, options, function(error, notification){
            if(error){
                reject(error);
            }else{
                resolve(notification);
            }
        });
    });
}

notifyMessageAsPromise("Hi!").then(function(notification){
    console.log(notification); // notification 객체
}).catch(function(error){
    console.error(error);
});
```

promise 객체를 반환하는 notifyMessageAsPromise()를 작성했다. 알람이 허가된 경우 'HI!'라고 출력하고 then()으로 등록한 콜백 함수가 호출된다. 허가하지 않으면 catch()가 호출된다. 브라우저는 Web Notifications API의 허가 상태를 사이트마다 기억하므로 실제로는 이미 허가한 경우, 허가 창을 이용해 허가한 경우, 이미 허가하지 않은 경우, 허가 창을 이용해 허가하지 않은 경우 이렇게 4가지

패턴이 존재한다고 볼 수 있다. 하지만 이를 최대한 단순화하여 두 가지 패턴으로 다룰 수 있도록 하는 게 사용하기 좋다.

`notifyMessageAsPromise()`는 Promise를 지원하지 않는 환경에서는 사용할 수 없다. 이 문제는 다음과 같은 방법으로 대응할 수 있다.

- Promise를 사용할 수 있는 환경만 지원한다.
- localForage처럼 라이브러리 자체에서 Promise를 구현한다.
- 콜백으로 사용할지 `promise` 객체화해서 사용할지 사용자가 선택할 수 있도록 `thenable` 객체를 반환한다.

이번에는 `thenable` 객체를 반환하도록 코드를 수정한다.

[예제](소스코드: http://jsbin.com/sepoyi/1/embed?js,console)

```
function notifyMessage(message, options, callback){
    var notification = null;

    if(Notification && Notification.permission === 'granted'){
        notification = new Notification(message, options);
        callback(null, notification);
    }else if (Notification.requestPermission){
        Notification.requestPermission(function(status){
            if(Notification.permission !== status){
                Notification.permission = status;
            }

            if(status === 'granted'){
                notification = new Notification(message, options);
                callback(null, notification);
```

```
            }else{
                callback(new Error('user denied'));
            }
        });
    }else{
        callback(new Error('doesn\'t support Notification API'));
    }
}

// thenable 객체를 반환한다.
function notifyMessageAsThenable(message, options){
    return {
        'then' : function(resolve, reject){
            notifyMessage(message, options, function(error, notification){
                if(error){
                    reject(error);
                }else{
                    resolve(notification);
                }
            });
        }
    };
}


Promise.resolve(notifyMessageAsThenable("message")).then(function(notificati
on){
    console.log(notification); // notification 객체
}).catch(function(error){
    console.error(error);
});
```

notifyMessageAsThenable()를 작성했다. 이 함수가 반환하는 리터럴 객체는 then()을 가지고 있다. 그리고 then()의 매개 변수에는 정상적으로 처리됐을 때 호출할 resolve()와 비정상적으로 처리됐을 때 호출할 reject()가 전달된다고 가정한다. notifyMessageAsThenable()의 then()은 notifyMessageAsPromise()와 같다. 이 thenable 객체를 Promise.resolve(thenable)로 작성해 promise 객체로 변환해 사용한다.

```
Promise.resolve(notifyMessageAsThenable("message")).then(function(notificati
on){
    console.log(notification);
}).catch(function(error){
    console.error(error);
});
```

notifyMessageAsThenable()과 notifyMessageAsPromise()는 사용법이 비슷하지만 차이점이 있다.

- notifyMessageAsThenable()은 Promise 코드가 없다.
- 사용자가 promise 객체를 사용하고자 한다면 promise 객체로 변환해야 한다.

5.2.3 Thenable에 대해

thenable은 Promise 기능에 직접 의존하지 않지만 인터페이스는 Promise 사양을 따르고 있으므로 간접적으로 Promise를 의존하고 있다고 볼 수 있다. 이는 콜백과 Promise, 둘 다 지원하는 방법이다. 하지만 라이브러리의 API로써는 어중간한 방법이기 때문에 자주 볼 수 있는 방식은 아니다. thenable 객체를 사용하기 위해선 사용자가 then()이나 Promise.resolve(thenable)에 대한 이해가 필요하므로 공개 API보단 내부적으로만 사용하는 경우가 많다.

Node.js의 Core module처럼 비동기 API는 우선 전통적인 콜백 스타일로 작성하는 게 좋다. 기본 API를 콜백 스타일로 작성하면 사용자가 Promise나 Generator 등 취향에 맞는 방법으로 구현할 수 있기 때문이다. 처음부터 Promise를 이용할 것을 목적으로 한 라이브러리나, 기능이 Promise를 의존하고 있는 경우에는 promise 객체를 반환하는 함수를 공개 API로 제공해도 문제가 되진 않는다.

thenable은 Promise 라이브러리간 상호 변환 시 가장 많이 사용한다. 예를 들어 Q 라이브러리는 ES6 Promises에서 정의하고 있지 않은 finally(), nodeify() 등 많은 기능을 구현하고 있다. ES6 Promises의 promise 객체를 Q 객체로 변환할 수 있다.

```
var Q = require("Q");

// 이 promise 객체는 ES6 Promises의 인스턴스이다.
var promise = new Promise(function(resolve){
    resolve(1);
});

// Q promise 객체로 변환
Q(promise).then(function(value){
    console.log(value);
}).finally(function(){
    console.log("finally");
});
```

ES6 Promises의 promise 객체는 then()을 가진 thenable한 객체다. 이를 Q 객체로 변환했기 때문에 finally()를 사용할 수 있다. 물론 반대로 변환하는 것도 가능하다. 이처럼 Promise 라이브러리는 각자 독자적인 방법으로 확장하지만 thenable이라는 공통 개념이 있기 때문에 라이브러리 간(네이티브 Promise 포함) 객

체를 변환할 수 있다. thenable은 보통 라이브러리 내부에서만 사용하기 때문에 자주 볼 수 없지만 Promise에서 꼭 알아야 할 개념이다.

5.3 throw 대신 reject 사용

Promise에서 추상화하고 있는 로직은 기본적으로 try-catch되는 것과 같으므로 처리 중 throw가 발생해도 프로그램은 종료되지 않고 promise 객체의 상태가 Rejected된다.

[예제](소스코드: http://jsbin.com/pugor/1/embed?js,console)

```
var promise = new Promise(function(resolve, reject){
    throw new Error("message");
});

promise.catch(function(error){
    console.error(error.message); // message
});
```

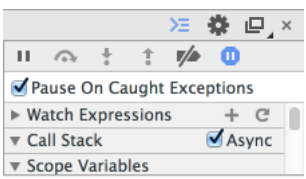
따라서 앞과 같이 작성해도 되지만 promise 객체의 상태를 Rejected하고자 할 때는 reject()를 사용하는 것이 일반적이다.

[예제](소스코드: http://jsbin.com/fiqoq/1/embed?js,console)

```
var promise = new Promise(function(resolve, reject){
    reject(new Error("message"));
});

promise.catch(function(error){
    console.error(error.message); // message
})
```

throw를 reject()로 바꿔 작성한다는 특징을 보면 reject()에 error 객체를 전달하는 것이 자연스럽게 느껴진다. 왜 throw가 아니라 reject()를 사용하는 게 좋을까, 그 이유 중 하나는 의도한 예외인지 아니면 예기치 않은 오류인지 구별하기 위함이다. 예를 들어 Chrome 개발자 도구는 예외가 발생했을 때 디버거가 자동으로 그 위치를 중단break하는 기능이 있다.

[그림 5-2] Pause On Caught Exceptions



이때 Promise에서 직접 throw한다면 디버거는 중단된다. 이는 디버깅의 목적과 관계없는 곳에서 중단될 수 있기 때문에 옳지 않다.

Promise 생성자에 전달하는 콜백에는 reject()가 인자로 전달되므로 throw를 사용하지 않고 간단히 promise 객체의 상태를 Rejected할 수 있다. 하지만 다음과 같이 then()에서 reject()를 사용하고 싶은 경우엔 어떻게 할까?

```
var promise = Promise.resolve();

promise.then(function(value){
    setTimeout(function(){
        // 처리가 1초 이상 지나면 reject(2)
    }, 1000);

    // 시간이 걸리는 처리 수행(1)

    somethingHardWork();
}).catch(function(error){
    // 타임 아웃 에러 처리(3)
});
```

위는 간단히 타임아웃을 처리하는 예제다. 이때 then() 안에서 reject()를 사용하고 싶지만 콜백 함수에 전달되는 것은 이전 promise 객체에서 반환한 결과 값이다. Promise를 사용한 타임아웃 처리에 대해서는 5.5 Promise.race를 사용한 타임아웃과 XHR 취소에서 자세히 소개한다. 일단 이번 절에서는 then()에서 reject()를 사용하는 방법에 대해서 알아본다. 잠시 then()의 동작에 대해 생각해보자.

then()으로 등록한 콜백 함수는 값을 반환할 수 있다. 그리고 반환한 값은 다음으로 then()이나 catch()로 등록한 콜백 함수에 전달된다. 이때 반환할 수 있는 값은 원시 타입뿐만 아니라 promise 객체도 가능하다. 반환한 값이 promise 객체인 경우, 객체의 상태에 따라 어느 함수가 호출될지 결정된다.

```
var promise = Promise.resolve();

promise.then(function(){
    var retPromise = new Promise(function(resolve, reject){
        // resolve 또는 reject 호출
    });

    return retPromise;
}).then(onFulfilled, onRejected);
```

retPromise의 상태가 Rejected되면 onRejected 함수가 호출된다. 따라서 throw를 사용하지 않아도 then()에서 reject()를 사용할 수 있다.

[예제](소스코드: http://jsbin.com/rixaze/1/embed?js,console)

```
var promise = Promise.resolve();

promise.then(function(){
    var retPromise = new Promise(function(resolve, reject){
```

```
        reject(new Error("this promise is rejected"));
    });

    return retPromise;
}).catch(function(error){
    console.log(error.message);
});
```

앞의 예제는 Promise.reject()를 사용함으로써 더 간결하게 표현할 수 있다.

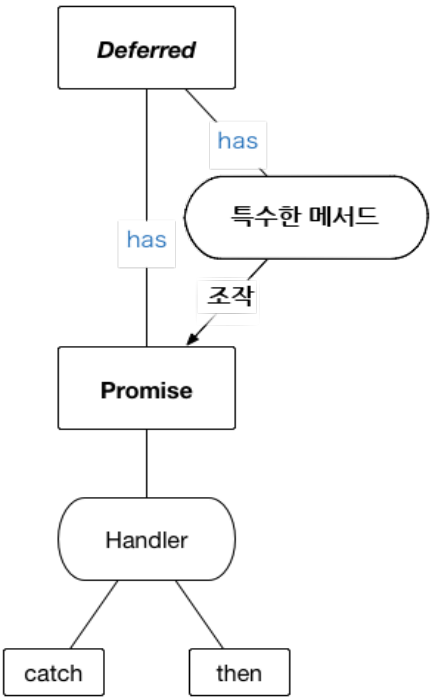
[예제](소스코드: http://jsbin.com/semehe/1/embed?js,console)

```
var promise = Promise.resolve();

promise.then(function(){
    return Promise.reject(new Error("this promise is rejected"));
}).catch(function(error){
    console.log(error.message);
});
```

5.4 Deferred와 Promise

Deferred는 Python의 Twisted라는 프레임워크에서 처음 정의한 개념이다. 자바스크립트에는 MochiKit.Async와 dojo/Deferred 등의 라이브러리가 그 개념을 차용하면서 알려졌고, 현재는 jQuery.Deferred와 JSDeferred가 가장 유명하다. Promise를 사용한 코드에서 Deferred를 본 적이 있을 것이다. Deferred는 Promise처럼 공통의 사양이 존재하지 않는다. 각자 비슷한 문제를 해결하기 위해 구현한 것을 Deferred라고 부르고 있다. 이번 절에서는 jQuery.Deferred와 비슷한 형태를 직접 만들어 가면서 Deferred와 Promise의 관계에 대해 알아보겠다.

[그림 5-3] Deferred와 Promise의 관계도



Deferred는 Promise를 랩핑하고 있으며 Promise의 상태를 조작할 수 있는 특수한 메서드가 구현돼 있다. 이는 jQuery.Deferred의 구조를 간략화한 것으로 Promise를 랩핑하지 않는 Deferred도 있음을 명심하길 바란다. 그럼 Promise를 랩핑하는 Deferred를 직접 구현해보겠다.

```
function Deferred() {
    this.promise = new Promise(function (resolve, reject) {
        this._resolve = resolve;
        this._reject = reject;
    }.bind(this));
}
```

```
Deferred.prototype.resolve = function (value) {
    this._resolve.call(this.promise, value);
};

Deferred.prototype.reject = function (reason) {
    this._reject.call(this.promise, reason);
};
```

1.3 Promise 사용하기에서 작성한 getURL()을 Deferred를 사용하여 재작성한다.

[예제](소스코드: http://jsbin.com/qocaza/1/embed?js,console)

```
function getURL(URL){
    var deferred = new Deferred(),
        req = new XMLHttpRequest();

    req.open('GET', URL, true);
    req.onload = function(){

        if(req.status == 200){
            deferred.resolve(req.responseText);
        }else{
            deferred.reject(new Error(req.statusText));
        }
    };

    req.onerror = function(){
        deferred.reject(new Error(req.statusText));
    };

    req.send();

    return deferred.promise;
}
```

```
var URL = "http://httpbin.org/get";

getURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(function(error){
    console.log(error.message);
});
```

Promise의 상태를 조작할 수 있는 메서드란 promise 객체의 상태를 Fulfilled 또는 Rejected할 수 있는 메서드를 말한다. 보통은 Promise 생성자 함수에 전달되는 resolve(), reject()를 사용해야 했다. 그럼 Promise로 구현한 getURL()과 비교해보자.

[예제](소스코드: http://jsbin.com/vazolu/2/embed?js,console)

```
function getURL(URL){
    return new Promise(function(resolve, reject){
        var req = new XMLHttpRequest();

        req.open('GET', URL, true);

        req.onload = function(){
            if(req.status == 200){
                resolve(req.responseText);
            }else{
                reject(new Error(req.statusText));
            }
        };

        req.onerror = function(){
            reject(new Error(req.statusText));
        };
```

```
        req.send();
    });
}
```

Deferred는 Promise를 랩핑하고 있어 큰 맥락은 같으며 특수한 메서드를 이용해 직접 상태 흐름을 제어할 수 있다. Promise로 구현한 getURL()을 보면 비동기 로직을 new Promise()로 감싸고 있다. Deferred는 감싸지 않아도 되기 때문에 결과적으로 함수의 중첩이 한 단계 줄어드는 효과가 있다. 하지만 처리 중 예상치 못한 오류가 발생하면 핸들링하지 못한다.

Promise로 구현한 getURL()은 promise 객체를 반환할 뿐 아무 작업도 하지 않는다. 전체적인 비동기 로직은 Promise에서 추상화되며 resolve(), reject()를 호출할 시점도 알기 쉽다.

```
new Promise(function(resolve, reject){
    // 이 안에서 resolve()나 reject()를 호출한다.
});
```

Deferred는 함수형으로 문제를 해결하는 것이 아니라 deferred 객체를 생성하고 임의의 시점에서 resolve(), reject()를 호출하는 방법으로 문제를 해결한다.

```
var deferred = new Deferred();
    // 임의의 시점에서 resolve(), reject()를 호출한다.
```

여기까지 Deferred를 간단하게 구현해봄으로써 Promise와의 차이점을 알 수 있었다. Promise는 비동기 로직과 상태를 모두 추상화한 객체인 것에 반해 Deferred는 상태만 추상화한다.

정리하면 Promise는 미래 어느 시점에 정상적 혹은 비정상적으로 처리가 완료될 것을 예약한 객체이고 Deferred는 아직 처리가 끝나지 않았다는 상태를 가리키는 객체다. 그리고 처리가 끝났을 때 Promise를 이용해 결과를 취득할 수 있도록 하는 구조다. Deferred에 대한 더 자세한 내용은 아래 문서를 참고한다.

- Promise & Deferred objects in JavaScript Pt.1: Theory and Semantics
- Twisted 입문
- Promise anti patterns

5.5 Promise.race를 사용한 타임아웃과 XHR 취소

이번 절에서는 `Promise.race()`를 사용해 타임아웃 기능을 구현하고, `XHR`을 취소하는 방법에 대해 알아본다. `XHR`에는 `timeout` 속성이 있지만 여러 개의 `XHR`을 동시에 호출하는 경우나 다른 기능 구현 시에도 응용할 수 있도록 차근차근 소개한다.

5.5.1 타임아웃 구현

우선 타임아웃을 Promise로 어떻게 구현할 수 있는지 알아본다. 타임아웃은 일정 시간이 지났을 때의 처리를 말한다. 따라서 `setTimeout()`을 이용하면 된다.

```
function delayPromise(ms){
    return new Promise(function(resolve){
        setTimeout(resolve, ms);
    });
}
```

`delayPromise()`는 지정한 밀리 세컨드 후 `resolve()`를 호출하는 `promise` 객체를 반환한다. `setTimeout()`을 직접 사용한 코드와 비교해보자.

```
setTimeout(function(){
    alert("100ms가 지났다.");
}, 100);

delayPromise(100).then(function(){
    alert("100ms가 지났다.");
});
```

이번에는 Promise.race()를 사용해 타임아웃을 구현한다. Promise.race()는 2.6 Promise.race에서 소개했듯이 promise 객체 중 어느 하나라도 해결되면 다음 동작으로 넘어가는 정적 메서드다. 이 원리를 이용해 일반적인 promise 객체와 delayPromise()의 객체를 동시에 실행시키는 방식으로 타임아웃을 구현할 수 있다.

```
function timeoutPromise(promise, ms){
    var timeout = delayPromise(ms).then(function(){
        throw new Error('Operation timed out after ' + ms + ' ms');
    });

    return Promise.race([promise, timeout]);
}
```

timeoutPromise()는 promise 객체와 타임아웃 시간을 전달받아 delayPromise()의 객체와 동시에 실행시켜 가장 먼저 완료된 promise 객체를 반환한다. 이 함수는 다음과 같이 사용할 수 있다.

[예제](소스코드: http://jsbin.com/cutira/1/embed?js,console)

```
var taskPromise = new Promise(function(resolve){
    // 어떤 처리
    var delay = Math.random() * 2000;
```

```
    setTimeout(function(){
        resolve(delay + "ms");
    }, delay);
});

timeoutPromise(taskPromise, 1000).then(function(value){
    console.log("taskPromise가 시간 내 완료 : " + value);
}).catch(function(error){
    console.log("타임 아웃", error.message);
});
```

타임아웃이 된 경우엔 에러를 출력하도록 했지만, 이 방식은 일반적인 오류와 타임아웃을 구별하지 못한다. 따라서 두 오류를 구별할 수 있도록 Error 객체를 상속하여 TimeoutError을 정의한다.

Error 객체

Error 객체는 ECMAScript의 Built-in(빌트-인, 내장) 객체다. ECMAScript5에서는 Error 객체의 특성(스택 트레이스)을 온전히 상속받지 못한다. 하지만 ECMAScript6에서는 class 구문을 사용해 Error 객체를 상속받아 내부적으로도 정확히 동작하는 또 다른 Error 객체를 만들 수 있다.

error instanceof

TimeoutError 구문이 유효하도록 TimeoutError 객체를 정의하면 다음과 같다.

```
function copyOwnFrom(target, source){
    Object.getOwnPropertyNames(source).forEach(function(propName){
        Object.defineProperty(target, propName,
Object.getOwnPropertyDescriptor(source, propName));
    });
```

```
    return target;
}

function TimeoutError(){
    var superInstance = Error.apply(null, arguments);
    copyOwnFrom(this, superInstance);
}

TimeoutError.prototype = Object.create(Error.prototype);
TimeoutError.prototype.constructor = TimeoutError;
```

TimeoutError 생성자 함수를 정의하고 prototype이 Error.prototype을 상속하도록 구현했다. Error 객체와 마찬가지로 throw하거나 reject()에서 사용할 수 있다.

```
var promise = new Promise(function(){
    throw new TimeoutError("timeout");
});

promise.catch(function(error){
    console.log(error instanceof TimeoutError); // true
});
```

이제 TimeoutError 객체를 사용해 일반적인 오류와 타임아웃을 쉽게 구별할 수 있다. timeoutPromise()를 수정한다.

```
function timeoutPromise(promise, ms) {
    var timeout = delayPromise(ms).then(function () {
            return Promise.reject(new TimeoutError('Operation timed out after '
+ ms + ' ms'));
```

```
        });

    return Promise.race([promise, timeout]);
}
```

예제 코드를 작성하는 데 사용한 빌트-인 객체 상속 방법은 Speaking Javascript의 Chapter 28. Subclassing Built-ins에 자세히 소개되어 있다.

5.5.2 타임아웃과 XHR 취소

여기까지 잘 이해했다면 Promise를 사용한 XHR 취소 기능도 구현할 수 있을 것이다. XHR 취소는 XMLHttpRequest 객체의 abort() 메서드를 호출하면 된다. 따라서 abort() 메서드를 클라이언트가 호출할 수 있도록 getURL()을 수정한다.

```
function cancelableXHR(URL) {
    var req = new XMLHttpRequest(),
        promise = new Promise(function(resolve, reject){
            req.open('GET', URL, true);

            req.onload = function(){
                if(req.status == 200){
                    resolve(req.responseText);
                }else{
                    reject(new Error(req.statusText));
                }
            };

            req.onerror = function(){
                reject(new Error(req.statusText));
            };

            req.onabort = function(){
```

```
            reject(new Error('abort this request'));
        };

        req.send();
    });

    var abort = function(){
        // 이전 request가 진행 중이라면 중단한다.
        //
https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest/
Using_XMLHttpRequest
        if(req.readyState !== XMLHttpRequest.UNSENT){
            req.abort();
        }
    };

    return {
        promise: promise,
        abort: abort
    };
}
```

이것으로 취소 기능을 위한 모든 요소(TimeoutError, cancelableXHR 등)를 구현했다. 이제는 흐름에 따라 적당한 로직을 작성하면 된다. 대략적인 흐름을 살펴보면 다음과 같다.

- cancelableXHR()을 사용해 promise 객체와 abort()를 반환받는다.
- timeoutPromise()를 사용해 promise 객체가 타임아웃되는지 Promise.race()로 판단한다.
- XHR 요청이 시간 내 완료된 경우엔 then()으로 등록한 콜백 함수가 실행된다.

- 타임아웃된 경우에는 catch()로 등록한 콜백 함수가 실행되고 error 객체가 전달된다. 안전성을 위해 TimeoutError 객체인지 판단 후 abort()를 실행한다.

[예제](소스코드: http://jsbin.com/kukowa/1/embed?js,console)

```
var object = cancelableXHR('http://httpbin.org/get');

timeoutPromise(object.promise, 1000).then(function (contents) {
    console.log('Contents', contents);
}).catch(function (error) {
    if (error instanceof TimeoutError) {
        object.abort();
        return console.log(error.message);
    }

    console.log('XHR Error :', error.message);
});
```

cancelableXHR()은 promise 객체와 abort()를 포함하는 리터럴 객체를 반환하는데 이는 조금 의아할 수 있다. 하나의 함수는 하나의 값을 반환하는 것이 좋지만 cancelableXHR() 내에서 생성한 req 객체는 사용자가 참조할 수 없으므로 필요한 메서드만 공개해 사용할 수 있도록 해야 했다. 반환하는 promise 객체에 abort() 기능을 확장할 수 있지만 어떤 형식이든 메서드까지 확장하면 복잡해질 수밖에 없다.

어찌 됐든 하나의 함수에서 모든 처리를 담당하는 것은 좋은 방법이 아니므로 promise 객체를 반환하는 함수와 진행 중인 요청을 abort()하는 함수를 분리해 구현한다. 이를 모듈로 만들면 확장성과 가독성, 실용적인 면에서도 유용할 것이다. 모듈을 작성하는 방법으로는 AMD, CommonJS, ES6 modules 등 여러 가지가 있지만 여기에서는 Node.js의 모듈로 작성한다.

```
"use strict";

var requestMap = {};

function createXHRPromise(URL) {
    var req = new XMLHttpRequest(),
        promise = new Promise(function (resolve, reject) {
            req.open('GET', URL, true);

            req.onreadystatechange = function () {
                if (req.readyState === XMLHttpRequest.DONE) {
                    delete requestMap[URL];
                }
            };

            req.onload = function () {
                if (req.status == 200) {
                    resolve(req.responseText);
                } else {
                    reject(new Error(req.statusText));
                }
            };

            req.onerror = function () {
                reject(new Error(req.statusText));
            };

            req.onabort = function () {
                reject(new Error('abort this req'));
            };

            req.send();
```

```
    });

    requestMap[URL] = {
        promise: promise,
        request: req
    };

    return promise;
}

function abortPromise(promise) {
    var request;

    if (typeof promise === "undefined") {
        return false;
    }

    Object.keys(requestMap).some(function (URL) {
        if (requestMap[URL].promise === promise) {
            request = requestMap[URL].request;
            return true;
        }
    });

    if (request != null && request.readyState !== XMLHttpRequest.UNSENT) {
        request.abort();
    }
}

module.exports.createXHRPromise = createXHRPromise;
module.exports.abortPromise = abortPromise;
```

createXHRPromise()에서 XHR 요청을 하는 promise 객체를 생성하고, 요청을 취소하고 싶은 경우엔 abortPromise()에 promise 객체를 전달하는 방식으로 사용할 수 있다.

```
var cancelableXHR = require("./cancelableXHR");

// XHR 요청을 하는 promise 객체를 생성한다.
var xhrPromise = cancelableXHR.createXHRPromise('http://httpbin.org/get');

xhrPromise.catch(function (error) {
    // abort된 에러
});

// promise 객체의 request 취소
cancelableXHR.abortPromise(xhrPromise);
```

Promise는 흐름을 제어하는 힘이 뛰어나기 때문에 그 장점을 최대한 살리기 위해서는 하나의 함수에 모든 처리를 구현하기 보다 작은 단위로 나누는 등, 자바스크립트에서 널리 알려진 안티패턴이나 구현 규칙을 더욱 더 의식해야 한다.

5.6 Promise.prototype.done

Promise를 구현한 많은 라이브러리에는 Promise.prototype.done이 구현돼 있다. Promise 라이브러리를 사용해본 적 있는 개발자는 then() 대신 done()을 사용해 본 적 있을 것이다. 사용방법은 then()과 같지만 promise 객체를 반환하지 않는다. 이번 절에서는 Promise.prototype.done에 관해 알아본다.

Promise.prototype.done은 ES6 Promises나 Promises/A+ 사양에 없는 기능이므로 직접 구현하거나 라이브러리를 사용해야 한다. 구현 방법은 뒤에서 설명하겠다. 먼저 then()을 사용한 경우와 done()을 사용한 경우를 비교해보자.

[done 예제](소스코드: http://jsbin.com/koqako/1/embed?js,console)

```
var donePromise = Promise.resolve();
donePromise.done(function () {
    JSON.parse('this is not json'); // SyntaxError: JSON.parse
});
```

[then 예제](소스코드: http://jsbin.com/pemasu/1/embed?js,console)

```
var thenPromise = Promise.resolve();
thenPromise.then(function () {
    JSON.parse("this is not json");
}).catch(function (error) {
    console.error(error.message); // SyntaxError: JSON.parse
});
```

두 코드를 실행해 보면 다음과 같은 사실을 알 수 있다.

- done()은 promise 객체를 반환하지 않으므로 then()이나 catch()를 체인할 수 없다. 따라서 promise 체인 마지막에 사용한다.
- done()에서 발생한 오류는 일반적인 방법으로 처리된다. 즉, Promise에서 처리하지 않는다.

왜 Promise의 기능과 상반되는 메서드를 많은 라이브러리에서 구현하고 있는지 생각해 볼 필요가 있다. 가장 큰 이유는 Promise에서 오류가 발생했을 때 생길 수 있는 문제 때문이다.

Promise에는 강력한 예외 처리 메커니즘이 있지만, 이 메커니즘이 의도하지 않은 오류를 더욱 복잡하게 만드는 경향이 있다. 이와 비슷한 문제를 5.3 throw 대신 reject 사용에서 소개한 바 있다. 다음 예제 코드를 보자.

```
function JSONPromise(value){
    return new Promise(function(resolve){
        resolve(JSON.parse(value));
    });
}
```

JSONPromise()는 전달된 값을 JSON.parse를 이용해 파싱하고 promise 객체를 반환한다. 만약 파싱 중 오류가 발생하면 catch()로 이 오류에 대응할 수 있다.

[예제](소스코드: http://jsbin.com/cirida/1/embed?js,console)

```
var string = "json이 아닌 일반 문자열";

JSONPromise(string).then(function(object){
    console.log(object);
}).catch(function(error){
    // JSON.parse에서 에러가 발생한다.
    console.error(error.message);
});
catch()를 사용해 제대로 사용한다면 문제가 되지 않는다. 하지만 이를 잊어버리면
어디에서 오류가 발생했는지 알 수 없게 된다.
var string = "json이 아닌 일반 문자열";

JSONPromise(string).then(function (object) {
    console.log(object);
});

// 오류가 발생했는지 알 수 없다.
```

앞의 예제와 같이 오류를 알아보기 쉬운 경우는 큰 문제가 아니지만, 문법 오류 같은 경우에는 문제가 심각해진다.

```
var string = "{}";

JSONPromise(string).then(function (object) {
   conosle.log(object);
});
```

console을 conosole이라고 잘못 작성했다. 이 코드를 실행하면 다음과 같은 오류가 발생해야 한다.

```
ReferenceError: conosle is not defined
```

하지만 Promise에서는 오류 처리되지 않는다. 매번 올바르게 catch()를 작성한다면 상관없지만 구현에 따라 오류를 감지하기 힘들 수 있다. 이렇게 처리되지 않는 오류를 unhandled rejection라고 한다. Rejected 됐을 때의 처리를 등록하지 않은 상태를 뜻한다. Unhandled rejection 문제는 Promise 구현에 따라 달라진다. 예를 들어 ypromise는 unhandled rejection일 때 발생한 오류는 개발자 도구의 콘솔에 출력한다.

(인용구) Promise rejected but no error handlers were registered to it

Bluebird 역시 의도하지 않은 ReferenceError는 개발자 도구의 콘솔에 출력한다.

(인용구) Possibly unhandled ReferenceError. conosle is not defined

네이티브 Promise 역시 이 문제를 대체하기 위해 GC-based unhandled rejection tracking이라는 기능을 구현하고 있다. 이 기능은 GC에서 회수하는 promise 객체가 unhandled rejection 객체라면 오류를 출력하는 방식으로 동작한다. Firefox, Chrome 등 일부 브라우저에서 구현돼 있다.

Promise의 done()은 오류가 일반적으로 처리되도록 하는 방법으로 unhandled rejection 문제를 회피한다. Promise.prototype을 확장하여 Promise.prototype.done을 구현해보자.

```
"use strict";

if(typeof Promise.prototype.done === "undefined"){
    Promise.prototype.done = function(onFulfilled, onRejected){
        this.then(onFulfilled, onRejected).catch(function(error){
            setTimeout(function(){
                throw error;
            }, 0);
        });
    };
}
```

setTimeout() 내에서 throw하여 일반적인 오류로 처리되도록 한다. 이렇게 작성한 오류는 Promise의 예외 처리 메커니즘으로 처리되지 않는다. 비동기 callback 내에서 에러를 감지하지 못하는 원인은 "자바스크립트와 비동기 오류 처리" 문서에 설명돼 있다.

done()은 아무것도 반환하지 않는다. 즉, 여기에서 promise 체인을 종료하고 에러가 났을 경우는 일반적인 방법으로 처리된다 라고 말하는 것과 같다. 만약 라이브러리나 브라우저가 unhandled rejection 문제에 제대로 대응하고 있다면 done()은 필요하지 않을 수 있다. 하지만 일관성을 유지할 목적으로 사용하기도 한다. 특별한 이유 없이 그냥 지나치면 애써 작성한 비동기 처리가 복잡해질 수 있기 때문에 일관성을 유지한다는 것은 Promise에서 중요한 부분이다.

Promise.prototype을 확장하여 간단한 방법으로 구현할 수 있기 때문에 ES6

Promises 사양에 채택되지 않았다. ES6 Promises 사양에 정의되어 있는 기능은 많지 않다. 따라서 개발자 스스로 확장하거나 이미 확장한 라이브러리를 이용한다. Promise를 확장하는 방법은 Promises:The Extension Problem(part 4)에 자세히 정리돼 있다. 또한, 이번 장에서 구현한 `Promise.prototype.done`은 promisejs.org를 참고했다.

5.7 Promise와 메서드 체인

Promise는 `then()` 또는 `catch()`를 체인 할 수 있다. 이는 DOM이나 jQuery 등에서 볼 수 있는 메서드 체인과 비슷한 기능이다. 일반적으로 메서드 체인은 `this`를 반환하여 메서드를 이어서 작성할 수 있도록 한다. 하지만 Promise는 항상 새로운 `promise` 객체를 반환하므로 외형적인 모습만 일반적인 메서드 체인과 같다. 이번 절에서는 일반적인 메서드 체인 방식으로 예제를 작성해보고, 인터페이스를 유지하면서 Promise 체인으로 구현하는 방법과 Promisification 개념에 관해 알아보겠다.

5.7.1 File System API 메서드 체인

Node.js의 File System API 을 메서드 체인 하여 사용할 수 있도록 구현한다. 이해를 돕기 위해 선택한 것이므로 실용적인 면에서는 유용한 예제가 아니다.

```
"use strict";

var fs = require("fs");

function File(){
    this.lastValue = null;
}

// Static method for File.prototype.read
File.read = function FileRead(filePath){
```

```
    var file = new File();
    return file.read(filePath);
};

File.prototype.read = function(filePath){
    this.lastValue = fs.readFileSync(filePath, "utf-8");
    return this;
};

File.prototype.transform = function(fn){
    this.lastValue = fn.call(this, this.lastValue);
    return this;
};

File.prototype.write = function(filePath){
    this.lastValue = fs.writeFileSync(filePath, this.lastValue);
    return this;
};

module.exports = File;
```

앞의 모듈을 다음과 같이 read(), transform(), write() 순으로 체인할 수 있다.

```
var File = require("./fs-method-chain"),
    inputFilePath = "input.txt",
    outputFilePath = "output.txt";

File.read(inputFilePath)
    .transform(function(content){
        return ">>" + content;
    })
    .write(outputFilePath);
```

fransform()은 전달받은 값을 변경하는 메서드로 앞 예제의 경우에는 read()에서 읽어드린 내용 앞에 >>라는 문자열을 추가한다. 이 인터페이스를 그대로 유지하면서 Promise로 처리되도록 구현한다.

```
"use strict";

var fs = require("fs");

function File() {
    this.promise = Promise.resolve();
}

// Static method for File.prototype.read
File.read = function(filePath){
    var file = new File();
    return file.read(filePath);
};

File.prototype.then = function(onFulfilled, onRejected){
    this.promise = this.promise.then(onFulfilled, onRejected);
    return this;
};

File.prototype["catch"] = function(onRejected){
    this.promise = this.promise.catch(onRejected);
    return this;
};

File.prototype.read = function(filePath){
    return this.then(function(){
        return fs.readFileSync(filePath, "utf-8");
    });
```

```
};

File.prototype.transform = function(fn){
   return this.then(fn);
};

File.prototype.write = function(filePath){
   return this.then(function(data){
      return fs.writeFileSync(filePath, data)
   });
};

module.exports = File;
```

promise 객체를 랩핑하여 then()과 catch()를 구현했지만 외형적인 인터페이스와 사용방법은 같다. 따라서 기존에 작성했던 실행 코드도 잘 동작한다.

```
var File = require("./fs-promise-chain");

File.read(inputFilePath)
   .transform(function (content) {
      return ">>" + content;
   })
   .write(outputFilePath);

// 위는 아래와 흐름이 같다.
promise.then(function read(){
      return fs.readFileSync(filePath, "utf-8");
   }).then(function transform(content) {
       return ">>" + content;
   }).then(function write(){
```

```
        return fs.writeFileSync(filePath, data);
    });
```

File.prototype.then은 this.promise.then()이 반환한 새로운 promise 객체를 this.promise에 대입한다. this.promise = this.promise.then()을 보면 얼핏 새로운 객체로 덮어쓰는 것처럼 보이기 때문에 promise 체인이 끊어질 것 같지만 실제로는 promise = addPromiseChain(promise, fn)과 같은 느낌으로 기존 promise 객체에 새로운 로직을 추상화한 promise 객체를 생성해 반환하기 때문에 문제가 되지 않는다.

두 방법은 동기와 비동기라는 큰 차이점이 있다. 일반적인 메서드 체인 방식도 큐와 같은 처리를 구현하면 비동기적으로 구현할 수 있지만 코드가 복잡해지므로 단순하게 동기적으로 구현했다. Promise는 항상 비동기로 처리하므로 작성한 예제 코드 역시 비동기로 처리된다.

두 번째 차이점은 오류 핸들링에 있다. 일반적인 메서드 체인 방식은 코드 전체를 try-catch로 감싸 오류를 처리해야 한다. Promise 체인 방식은 catch()를 이용해 오류를 처리할 수 있다.

```
File.read(inputFilePath)
    .transform(function (content) {
        return ">>" + content;
    })
    .write(outputFilePath)
    .catch(function(error){
        console.error(error);
    });
```

일반적인 메서드 체인 방식에서 비동기 처리를 구현하게 되면 오류 핸들링이 큰 문

제가 되기 때문에 비동기 처리가 필요한 경우는 Promise를 사용하는 편이 좋다.

Node.js에 익숙한 개발자는 메서드 체인과 비동기 처리를 보면서 Stream을 생각했을 것이다. Stream을 사용하면 this.lastValue와 같은 값을 유지할 필요가 없고 무엇보다 큰 파일을 취급할 때 유용하다. 또 Promise를 사용했을 때 보다 고속으로 처리될 가능성이 높다.

```
//stream을 이용한 read→transform→write 처리
readableStream.pipe(transformStream).pipe(writableStream);
```

즉, 비동기 처리에 항상 Promise가 최적일 것이라는 가정은 좋지 않다. 목적과 상황에 맞게 적절한 방법을 찾아야 한다.

5.7.2 Promisification

자바스크립트는 동적으로 메서드를 정의할 수 있으므로 런타임 상에서 기존 코드에 promise 메서드를 정의해 사용할 수 있다. 이런 기능을 Promisification이라고 한다. 물론 동적으로 인터페이스가 결정되는 건 좋지 않지만 때때로 문제 해결, 효율성 등의 이유로 사용된다. Promisification은 ES6 Promises 사양에는 없는 개념이지만 유명한 Promise 라이브러리인 Bluebird나 Q 등에는 구현돼 있다. 이 기능을 이용하면 다음과 같이 동적으로 promise 메서드를 추가해 사용할 수 있다.

```
var fs = Promise.promisifyAll(require("fs"));

fs.readFileAsync("myfile.js", "utf8").then(function(contents){
   console.log(contents);
}).catch(function(e){
   console.error(e.stack);
});
```

앞의 예제만으로는 이해하기 어려울 수 있느니 직접 네이티브 Array에 promise 메서드를 추가하는 객체를 정의하겠다. 자바스크립트에는 DOM, String 등 메서드 체인이 가능한 객체가 있다. Array도 그 중 하나다. Array의 map, filter 등의 메서드는 배열을 반환하기 때문에 메서드를 체인할 수 있다.

```
"use strict";

function ArrayAsPromise(array){
    this.array = array;
    this.promise = Promise.resolve();
}

ArrayAsPromise.prototype.then = function(onFulfilled, onRejected){
    this.promise = this.promise.then(onFulfilled, onRejected);
    return this;
};
ArrayAsPromise.prototype["catch"] = function(onRejected){
    this.promise = this.promise.catch(onRejected);
    return this;
};

Object.getOwnPropertyNames(Array.prototype).forEach(function(methodName){
    var arrayMethod = null;

    // Don't overwrite
    if(typeof ArrayAsPromise[methodName] !== "undefined"){
        return null;
    }

    arrayMethod = Array.prototype[methodName];
    if(typeof arrayMethod !== "function"){
```

```
        return null;
    }

    ArrayAsPromise.prototype[methodName] = function(){
        var that = this,
            args = arguments;

        this.promise = this.promise.then(function(){
            that.array = Array.prototype[methodName].apply(that.array, args);
            return that.array;
        });

        return this;
    };
});

module.exports = ArrayAsPromise;
module.exports.array = function newArrayAsPromise(array){
    return new ArrayAsPromise(array);
};
```

네이티브 Array를 사용한 경우와 ArrayAsPromise()를 사용한 경우를 테스트한다.

```
"use strict";

var assert = require("power-assert"),
    ArrayAsPromise = require("./array-promise-chain");

describe("array-promise-chain", function(){
    function isEven(value){
        return value % 2 === 0;
    }
```

```
    function double(value){
        return value * 2;
    }

    beforeEach(function(){
        this.array = [1, 2, 3, 4, 5];
    });

    describe("Native array", function(){
        it("can method chain", function(){
            var result = this.array.filter(isEven).map(double);
            assert.deepEqual(result, [4, 8]);
        });
    });

    describe("ArrayAsPromise", function(){
        it("can promise chain", function(done){
            var array = new ArrayAsPromise(this.array);
            array.filter(isEven).map(double).then(function(value){
                assert.deepEqual(value, [4, 8]);
            }).then(done, done);
        });
    });
});
```

네이티브 Array는 동기적으로, ArrayAsPromise()는 비동기적으로 처리된다. ArrayAsPromise()를 사용한 경우에도 Array의 메서드를 사용하고 있는데, ArrayAsPromise()의 코드를 보면 알 수 있듯이 Array.prototype의 메서드를 모두 구현한다. Array.prototype에는 array.indexOf와 같이 배열을 반환하지 않는 메서드도 있기 때문에 모든 메서드를 체인할 순 없다.

Node.js의 Core 모듈을 Promisification하는 경우에는 function(error, result){} 인터페이스를 이용해 자동으로 Promise로 랩핑한 객체를 생성한다. 이처럼 통일된 인터페이스로 구현한 API는 조금 더 유용하게 Promisification할 수 있다. 이렇듯 API의 규칙성을 의식하면 또 다른 좋은 방법을 발견할 수 있을 것이다.

5.8 Promise를 이용한 순차 처리

2.5 Promise.all에서 여러 개의 promise 객체를 한꺼번에 처리하는 방법을 소개했다. 하지만 Promise.all()은 모든 객체를 동시에 처리하므로 차례대로 처리를 할 수 없다. 이번 절에서는 promise 객체를 순차로 처리하는 방법에 관해서 알아본다.

2.3 Promise.prototype.then에서 복수의 XHR 요청을 설명하기 위해 작성한 예제가 promise 객체를 차례대로 처리한다. 하지만 XHR 요청 수가 늘어나는 만큼 then() 역시 추가로 작성해야 하므로 별로 유용한 방법은 아니다. 이때 promise 객체를 배열로 만들어 for 루프로 하나씩 처리하면 XHR 요청 수가 늘어나는 경우를 신경 쓰지 않아도 된다.

[예제](소스코드: http://jsbin.com/zewaha/1/embed?js,console)

```
var request = {
  infomation: function getComment(){
    return getURL('http://httpbin.org/get').then(JSON.parse);
  },
  cookie: function getPeople(){
    return getURL('http://httpbin.org/cookies').then(JSON.parse);
  }
};

function main(){
    function recordValue(results, value){
```

```
        results.push(value);
        return results;
    }

    // 처리가 끝났을때 값을 []에 push
    var pushValue = recordValue.bind(null, []);

    // promise 객체를 반환하는 함수 배열
    var tasks = [request.infomation, request.cookie],
        promise = Promise.resolve();

    // 처리가 시작되는 지점
    for(var i = 0; i < tasks.length; i++){
        var task = tasks[i];
        promise = promise.then(task).then(pushValue);
    }

    return promise;
}

main().then(function(value){
    console.log(value);
}).catch(function(error){
    console.error(error);
});
```

---

for 루프를 살펴보겠다. promise의 then()은 항상 새로운 promise 객체를 반환한다. 따라서 promise = promise.then(task).then(pushValue)는 promise 변수에 덮어쓴다기보다 기존 객체에 처리를 추가해나간다고 보면 된다. 하지만 이 방법은 임시 변수로 사용할 promise 객체가 필요하고 흐름도 매끄럽지 않다. 이 루프는 Array.prototype.reduce를 사용하면 조금 더 훌륭히 작성할 수 있다.

[예제](소스코드: http://jsbin.com/hixare/1/embed?js,console)

```
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }

    var pushValue = recordValue.bind(null, []),
        tasks = [request.infomation, request.cookie];

    return tasks.reduce(function (promise, task) {
        return promise.then(task).then(pushValue);
    }, Promise.resolve());
}
```

for 루프 예제에서 main()만 변경됐다. Array.prototype.reduce는 두 번째 매개변수에 초기값을 지정할 수 있다. 즉, 최초의 promise로 Promise.resolve()를 지정하고, 첫 번째 task로 request.comment가 실행한다. Reduce에서 반환한 객체가 다음 루프에서 promise로 전달된다. 따라서 promise 체인을 할 수 있다. for 루프와 다르게 임시 변수가 필요 없어졌으므로 promise = promise.then(task).then(pushValue) 같은 혼란스러운 코드가 제거됐다.

Array.prototype.reduce와 Promise 조합은 순차적 처리 구현 시 유용하게 사용된다. 하지만 처음 코드를 읽을 때 어떤 식으로 동작하는지 알기 어렵다. 따라서 promise 객체를 반환하는 함수를 배열로 전달받아 차례대로 처리하는 sequenceTasks()를 작성한다.

```
var pushValue = null;
function sequenceTasks(tasks){
```

```
    function recordValue(results, value){
        results.push(value);
        return results;
    }

    pushValue = recordValue.bind(null, []);

    return tasks.reduce(function(promise, task){
        return promise.then(task).then(pushValue);
    }, Promise.resolve());
}

sequenceTasks([request.comment, request.people]);
```

이제 함수명만 봐도 차례대로 처리된다는 사실을 알 수 있다. `Promise.all()`과 달리 인자로 전달하는 것은 `promise` 객체를 반환하는 함수의 배열이라는 점에 주목하자. `Promise` 객체를 배열로 전달하면 이미 객체에 추상화된 로직이 실행되기 때문에 의도와 다르게 동작한다.

5.9 정리

이번 장에서는 Promise의 라이브러리, 타임아웃 구현 등 Promise에 대한 다양한 주제에 관해서 이야기했다. 예제를 보면 Promise를 사용하더라도 목적별로 함수를 나눈다는 사실을 알 수 있다. 대개 Promise로 코드를 작성하면 Promise 체인을 과도하게 연결하여 작성하는 경향이 있다. 이때 전체적인 로직을 알아보기 쉽게 함수를 나누는 게 좋다. Promise 생성자 함수나 `then()`은 고차 함수이므로 목적별로 나눈 함수를 조합하기 쉽다. 또한, Promise는 모든 비동기 처리의 해답이 될 수 없다. 예를 들어 여러 번 콜백을 호출하는 Event 같은 경우에는 오히려 부적합하다. 어떤 비동기든 Promise로 해결하기보다 현재 상황에 Promise가 맞는지를 먼저 생각해보길 바란다.

부록 | Promise API

부록.1 Promise.prototype.then

```
promise.then(onFulfilled, onRejected);
```

예)

```
var promise = new Promise(function(resolve, reject){
    resolve("이 값은 then()에 전달된다.");
});
promise.then(function(value){
    console.log(value);
}, function(error){
    console.error(error);
});
```

promise 객체에 대한 onFulfilled와 onRejected 콜백 함수를 정의하고 새로운 promise 객체를 반환한다. 이 함수는 promise 객체가 완료resolve 또는 실패reject될 때 각각 호출된다. 처리 로직에서 반환한 값은 새로운 promise 객체의 onFulfilled 콜백 함수에 전달한다. 처리 로직에서 오류가 발생하면 error 객체를 새로운 promise 객체의 onRejected 콜백 함수에 전달한다.

부록.2 Promise.prototype.catch

```
promise.catch(onRejected);
```

예)

```
var promise = new Promise(function(resolve, reject){
    reject(new Error('error 객체는 catch()에 전달된다.'));
});
promise.then(function(value){
    console.log(value);
}).catch(function(error){
    console.error(error);
});
```

부록.3 Promise.resolve

```
Promise.resolve(promise);
Promise.resolve(thenable);
Promise.resolve(object);
```

예)

```
var taskName = "task 1";

asyncTask(taskName).then(function(value){
    console.log(value);
}).catch(function(error){
    console.error(error);
});

function asyncTask(name){
    return Promise.resolve(name).then(function(value){
        return "Done! "+ value;
    });
```

```
}
```

Fulfilled 상태의 promise 객체를 반환한다. 전달받은 값은 then()으로 등록한 콜백 함수에 전달된다. 어떤 경우에도 promise 객체를 반환하지만 크게 세 가지 케이스로 나눌 수 있다.

- promise 객체를 전달받은 경우엔 전달받은 객체를 반환한다.
- thenable 객체를 전달받은 경우엔 promise 객체로 변환해서 반환한다.
- 다른 값(객체나 null 포함)을 전달받은 경우엔 새로운 promise 객체를 생성해 반환한다.

부록.4 Promise.reject

```
Promise.reject(object)
```

예)

```
var failureStub = sinon.stub(xhr, "request").returns(Promise.reject(new
Error("bad!")));
```

Rejected 상태의 promise 객체를 생성해 반환한다. reject()에는 error 객체를 전달해야 한다. 전달받은 값은 catch()로 등록한 콜백 함수에 전달된다. resolve()와는 달리 항상 새로운 객체를 생성한다.

```
var r = Promise.reject(new Error("error"));
console.log(r === Promise.reject(r)); // false
```

부록.5 Promise.all

```
Promise.all(promiseArray);
```

예)

```
var p1 = Promise.resolve(1),
    p2 = Promise.resolve(2),
    p3 = Promise.resolve(3);
Promise.all([p1, p2, p3]).then(function (results) {
    console.log(results); // [1, 2, 3]
});
```

새로운 promise 객체를 생성해 반환한다. 배열로 전달한 promise 객체의 처리가 모두 완료됐을 때 새로운 promise 객체도 완료된다. 객체 중 하나라도 실패하면 새로운 객체도 실패한다. 배열의 값은 Promise.resolve()로 다뤄지기 때문에 promise 객체 이외의 값이 존재해도 정상적으로 동작한다.

부록.6 Promise.race

```
Promise.race(promiseArray);
```

예)

```
var p1 = Promise.resolve(1),
    p2 = Promise.resolve(2),
    p3 = Promise.resolve(3);
Promise.race([p1, p2, p3]).then(function (value) {
    console.log(value);  // 1
```

```
});
```

새로운 promise 객체를 생성해 반환한다. 배열로 전달한 promise 객체 중 하나만 완료되면 새로운 promise 객체도 완료된다.