[WIP] JavaScript Promiseの本

azu

Version {bookversion}

Table of Contents

ハッシュタグは #Promise本

IMPORTANT	この書籍はまだ作業中です！ Working on the book. Contributingは Github から
IMPORTANT	この書籍の進行状況は Issues · azu/promises-book や以下から確認できます。

はじめに

書籍の目的

この書籍は現在策定中のECMAScript 6 Promisesという仕様を中心にし、JavaScriptにおけるPromiseについて学ぶことを目的とした書籍です。

この書籍を読むことで学べる事として次の3つを目標としてあります。

- Promiseについて学び、パターンやテストを扱えるようになる事
- Promiseの向き不向きについて学び、何でもPromiseで解決するべきではないと知ること
- ES6 Promiseを元に基本をよく学び、より発展した形を自分で形成できるようになること

この書籍では、先程も述べたようにES6 Promises、つまりJavaScriptの標準仕様をベースとしたPromiseについて書かれています。

そのため、FirefoxやChromeなど先行実装しているブラウザでは、ライブラリを使うこと無く利用できる機能であり、またES6 Promisesは元がPromises/A+というコニュニティベースの仕様であるため、多くの実装ライブラリがあります。

ブラウザネイティブの機能 または ライブラリを使うことで今すぐ利用できるPromiseについて基本的なAPIから学び、何が得意で何が不得意なのかを知り、Promiseを活用したJavaScriptを書けるようになることを目的としています。

本書を読むにあたって

この書籍ではJavaScriptの基本的な機能について既に学習していることを前提にしています。

- JavaScript: The Good Parts
- JavaScriptパターン
- JavaScript 第6版 - 十分すぎます
- パーフェクトJavaScript
- Effective JavaScript

のいずれかの書籍を読んだ事があれば十分読み解くことが出来る内容だと思います。

または、JavaScriptでウェブアプリケーションを書いたことがある、Node.jsでコマンドラインアプリやサーバサイドを書いたことがあれば、
どこかで書いたことがあるような内容が出てくるかもしれません。

一部セクションではNode.js環境での話となるため、Node.jsについて軽くでも知っておくとより理解がし易いと思います。

表記法

この書籍では短縮するために幾つかの表記を用いています。

- Promiseに関する用語は用語集を参照する。
  - 大体、初回に出てきた際にはリンクを貼っています。
- インスタンスメソッドを instance#method という表記で示す。
  - 例えば、`Promise#then`
    という表記は、Promiseのインスタンスオブジェクトの`then`というメソッドを示しています。
- オブジェクトメソッドを object.method という表記で示す。
  - これはJavaScriptの意味そのままで、`Promise.all` なら静的メソッドの事を示しています。

> この部分には文章についての補足が書かれています。

本書のソースコード/ライセンス

この書籍に登場するサンプルのソースコード また
その文章のソースコードは全てはGithubから取得することができます。

この書籍は AsciiDoc という形式で書かれています。

- <link href="https://github.com/azu/promises-book">azu/promises-book</link> <img src="https://travis-ci.org/azu/promises-book.svg?branch=master" alt="Build Status">

またリポジトリには書籍中に出てくるサンプルコードのテストも含まれています。

ソースコードのライセンスはMITライセンスで、文章はCC-BY-NCで利用することができます。

意見や疑問点

意見や疑問点がある場合はGithubに直接Issueとして立てる事が出来ます。

- Issues · azu/promises-book

また、Githubアカウントを利用した チャットサービス に書いていくのもよいでしょう。

- <img src="https://badges.gitter.im/azu/promises-book.png" alt="Gitter">

Twitterでのハッシュタグは #Promise本 なので、こちらを利用するのもいいでしょう。

この書籍は読める権利と同時に編集する権利があるため、 Githubで Pull Requests も歓迎しています。

1. Chapter.1 - Promiseとは何か

Promiseとは何かの簡単な概要の紹介です。

1.1. What Is Promise

- プロミスとはどういう概念なのかー
- Promisesはどうやってできた? ⇐ これはコラムとかで良さそう
- 何を目的にしてるー
- どういう時に利用すると便利なのー

1.2. Promises Overview

ES6 Promises で定義されているAPIはそこまで多くはありません。

多く分けて以下の3種類になります。

Constructor

promiseオブジェクトを作成するには、 Promiseコンストラクタを`new`でインスタンス化します。

```
new Promise(function(resolve, reject) {});
```

Instance Method

インスタンスとなるpromiseオブジェクトにはpromiseの値が resolve(解決) / reject(棄却) された時に呼ばれる コールバックを登録するため `promise.then()` というインスタンスメソッドがあります。

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

resolve(解決)された時
`onFulfilled` が呼ばれる

reject(棄却)された時
`onRejected` が呼ばれる

`onFulfilled`、`onRejected` どちらもオプショナルな引数となり、 エラー処理だけを書きたい場合には `promise.then(undefined, onRejected)` と同じ意味である `promise.catch()`を使うことが出来ます。

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

Static Method

`Promise` というグローバルオブジェクトには幾つかの静的なメソッドが存在します。

`Promise.all()` や `Promise.resolve()` などが該当し、Promiseを扱う上での補助メソッドが中心となっています。

1.2.1. Promise workflow

以下のようなサンプルコードを見てみましょう。

Listing 1. promise-workflow.js

```
'use strict';
function asyncFunction() {
    <i class="conum" data-value="1"></i><b>(1)</b>
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve('Async Hello world');
        }, 16);
    });
}
<i class="conum" data-value="2"></i><b>(2)</b>
asyncFunction().then(function (value) {
    console.log(value);    // => 'Async Hello world'
}).catch(function (error) {
    console.log(error);
});
```

`asyncFunction` という関数 は promiseオブジェクトを返していて、そのpromiseオブジェクトに足しして `then` でresolveされた時のコールバックを、 `catch` でエラーとなった場合のコールバックを設定しています。

このpromiseオブジェクトはsetTimeoutで16ms後にresolveされるので、 そのタイミングで `then` のコールバックが呼ばれ `'Async Hello world'` と出力されます。

いまどきの環境では `catch` のコールバックは呼ばれる事はないですが、`setTimout`が存在しない環境などでは、例外が発生し`catch`のコールバックが呼ばれると思います。

もちろん、`promise.then(onFulfilled, onRejected)` というように、 `catch` を使わずに `then` は以下のように2つのコールバックを設定することでもほぼ同様の動作になります。

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

1.2.2. Promiseの状態

Promiseの処理の流れが簡単にわかった所で、少しPromiseの状態について整理したいと思います。

`new Promise` でインスタンス化したpromiseオブジェクトには以下の3つの状態が存在します。

"has-resolution" - Fulfilled
    resolve(解決)された時。この時 `onFulfilled` が呼ばれる

"has-rejection" - Rejected
    reject(棄却)された時。この時 `onRejected` が呼ばれる

"unresolved" - Pending
: resolveまたはrejectではない時。つまりpromiseオブジェクトが作成された初期状態等が該当する

見方ですが、 左が ES6Promisesで定められている名前で、 右が Promises/A+で登場する状態の名前になっています。

基本的にこの状態をプログラムで直接触る事はないため、名前自体は余り気にしなくても問題ないです。 この文章では、 Promises/A+ の Pending、Fulfilled 、Rejected を用いて解説していきます。

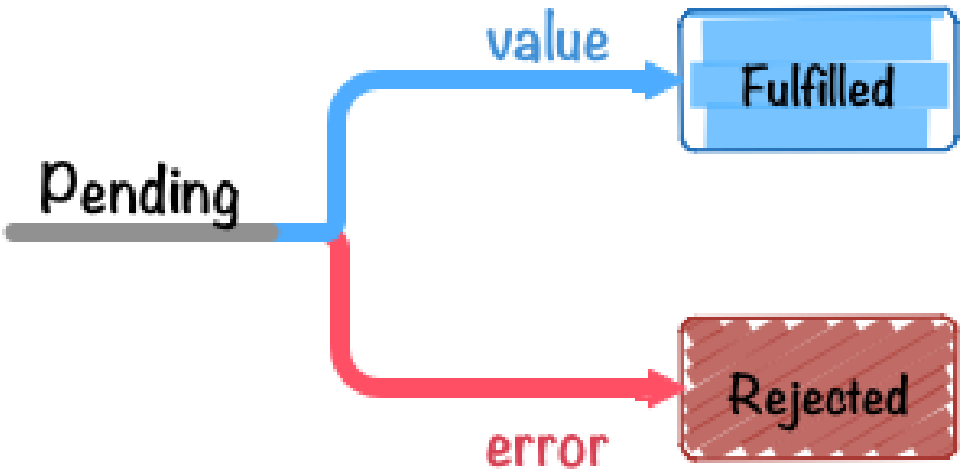


Figure 1. promise states

NOTE

ECMAScript Language Specification ECMA-262 6th Edition – DRAFT では `[[PromiseStatus]]` という内部定義によって状態が定められています。
`[[PromiseStatus]]`
にアクセスするユーザーAPIは用意されていないため、基本的には知る方法はありません。

3つの状態を見たところで、既にこの章で全ての状態が出てきていることが分かります。

promiseオブジェクトの状態は、一度PendingからFulfilledやRejectedになると、
そのpromiseオブジェクトの状態はそれ以降変化することはなくなります。

つまり、PromiseはEvent等とは違い、`.then`
で登録した関数が呼ばれるのは1回限りという事が明確になっています。

また、FulfilledとRejectedのどちらかの状態であることをSettled(不変の)と表現することがあります。

Settled
: resolve(解決) または reject(棄却) された時。

PeddingとSettledが対となる関係であると考えると、Promiseの状態の種類/遷移がシンプルであることがわかると思います。

このpromiseオブジェクトの状態が変化した時に、一度だけ呼ばれる関数を登録するのが `.then` といったメソッドとなるわけです。

NOTE

JavaScript Promises - Thinking Sync in an Async World // Speaker Deck というスライドではPromiseの状態遷移について分かりやすく書かれています。

1.3. Promiseの書き方

Promiseの基本的な書き方について解説します。

1.3.1. promiseオブジェクトの作成

promiseオブジェクトを作る流れは以下のようになっています。

1. `new Promise(fn)` の返り値がpromiseオブジェクト
2. 引数となる関数fnには `resolve` と `reject` が渡る
3. `fn`には非同期等の何らかの処理を書く
    - 処理結果が正常なら、`resolve(▫▫▫▫)` を呼ぶ
    - 処理結果がエラーなら、`reject(Error▫▫▫▫▫▫)` を呼ぶ

実際にXHRでGETをするものをpromiseオブジェクトにしてみましょう。

Listing 2. xhr-promise.js

```
'use strict';
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.response);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
```

XHRでステータスコードが200の場合のみ `resolve` して、 それ以外はエラーであるとして `reject` しています。

`resolve(req.response)` ではレスポンスの内容を引数に入れています。
resolveの引数に入れる値には特に決まりはありませんが、コールバックと同様に次の処理へ渡したい値を入れるといいでしょう。(この値は`then`メソッドで受け取ることが出来ます)

Node.jsをやっている人は、コールバックを書く時に `callback(error, response)` と第一引数にエラーオブジェクトを
入れることがよくあると思いますが、Promiseでは役割がresolve/rejectで分担されているので、
resolveにはresponseの値のみをいれるだけで問題ありません。

次に、`reject` の方を見て行きましょう。

XHRで`onerror`のイベントが呼ばれた場合はもちろんエラーなので`reject`を呼びます。
ここで`reject`に渡している値に注目してみてください。

エラーの場合は `reject(new Error(req.statusText));` というようにErrorオブジェクトとして渡している事がわかると思います。 `reject` には値であれば何でも良いのですが、一般的にErrorオブジェクト(またはErrorオブジェクトを継承したもの)を渡すことになっています。

`reject` に渡す値はrejectする理由を書いたErrorオブジェクトとなっています。
今回は、ステータスコードが200以外であるならrejectするとしていたため、`reject`▫▫statusText▫▫▫▫▫▫▫ (▫▫▫▫`then`▫▫▫▫▫▫▫▫▫ or `catch` メソッドで受け取ることが出来ます)

1.3.2. promiseオブジェクトに処理を書く

先ほどの作成したpromiseオブジェクトを返す関数を実際に使ってみましょう

```
getURL("http://example.com/"); // => promise▫▫▫▫▫▫
```

Promises Overview
でも簡単に紹介したようにpromiseオブジェクトは幾つかインスタンスを持っており、
これを使いpromiseオブジェクトの状態に応じて一度だけ呼ばれるコールバックとなる関数を登録します。

promiseオブジェクトに登録する処理は以下の2種類が主となります

- promiseオブジェクトが resolve された時の処理(onFulfilled)
- promiseオブジェクトが reject された時の処理(onRejected)

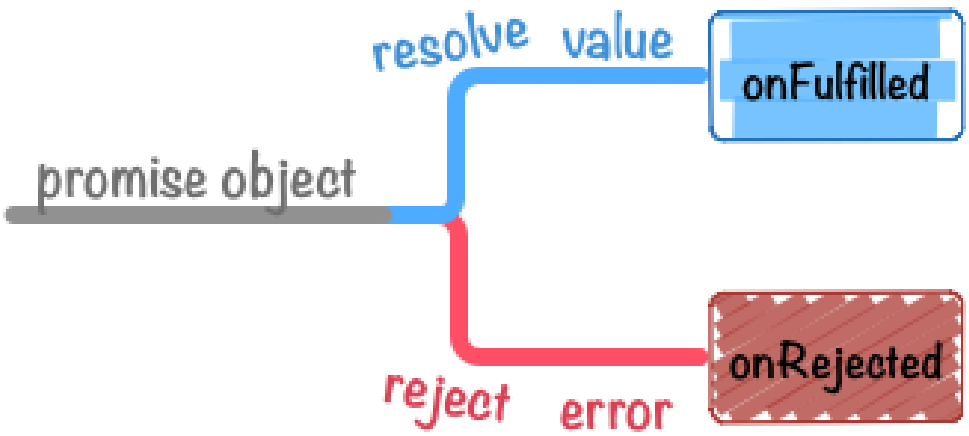


Figure 2. promise value flow

まずは、`getURL`で通信が成功して値が取得出来た場合の処理を書いてみましょう。
この場合の 通信が成功した というのは promiseオブジェクトがresolveされた 時という事ですね。
resolveされた時の処理は、 `.then` メソッドに処理をする関数を渡すことで行えます。

```
var URL = "http://httpbin.org/get";
getURL(URL).then(function onFulfilled(value){ <i class="conum" data-
value="1"></i><b>(1)</b>
    console.log(value);
});
```

getURL関数　内で　resolve(req.response);　によってpromiseオブジェクトが解決されると、値と共に`onFulfilled` 関数が呼ばれます。

このままでは通信エラーが起きた場合などに何も処理がされないため、　今度は、getURLで何らかの問題があってエラーが起きた場合の処理を書いてみましょう。

この場合の エラーが起きた というのは promiseオブジェクトがrejectされた 時という事ですね。

rejectされた時の処理は、.then`□□□□□>>　□□□　<<promise.catch,.catch`メソッドに処理をする関数を渡す事で行えます。

先ほどのソースにrejectされた場合の処理を追加してみましょう。

```
var URL = "http://httpbin.org/status/500"; <i class="conum" data-
value="1"></i><b>(1)</b>
getURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(function onRejected(error){ <i class="conum" data-
value="2"></i><b>(2)</b>
    console.log(error);
});
```

promiseオブジェクトが何らかの理由で例外が起きた時、または明示的にrejectされた場合に、その理由(Errorオブジェクト)と共に .catch の処理が呼ばれます。

.catchは`promise.then(undefined,　onRejected)`のエイリアスであるため、同様の処理は以下のように書くことも出来ます。

```
getURL(URL).then(onFulfilled, onRejected);<i class="conum" data-
value="1"></i><b>(1)</b>
```

基本的には、.catchを使いresolveとrejectそれぞれを別々に処理した方がよいと考えらますが、両者の違いについては thenとcatchの違い で紹介します。

まとめ

この章では以下のことについて簡単に紹介しました。

- new Promise を使いpromiseオブジェクトの作成
- .then や .catch を使ったpromiseオブジェクトの処理

Promiseの基本的な書き方はこれがベースとなり、
他の多くの処理はこれを発展させたり、用意された静的メソッドを利用したものになります。

ここでは、同様の事はコールバック関数を渡す形でも出来るのに対してPromiseで書くメリットについては触れていませんでしたが、
次の章では、Promiseのメリットであるエラーハンドリングの仕組みをコールバックベースの実装と比

較しながら見て行きたいと思います。

2. Chapter.2 - Promiseの書き方

この章では、Promiseのメソッドの使い方、エラーハンドリングについて学びます。

2.1. Promise.resolve

一般に `new Promise()` を使う事でpromiseオブジェクトを生成しますが、それ以外にもpromiseオブジェクトを生成する方法があります。

ここでは、`Promise.resolve` と `Promise.reject` について学びたいと思います。

2.1.1. new Promiseのショートカット

`Promise.resolve(value)` という静的メソッドは、`new Promise()` のショートカットとなるメソッドです。

例えば、`Promise.resolve(42);` というのは下記のコードのシンタックスシュガーです。

```
new Promise(function(resolve){
    resolve(42);
});
```

結果的にすぐに`resolve(42);`と解決されて、次のthenの`onFulfilled`に設定された関数に`42`という値を渡します。

`Promise.resolve(value);` で返ってくる値も同様にpromiseオブジェクトなので、以下のように続けて `.then` を使った処理を書くことが出来ます。

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

Promise.resolveは`new Promise()`のショートカットとして、promiseオブジェクトの初期化時やテストコードを書く際にも活用できます。

2.1.2. Thenable

もう一つ`Promise.resolve`の大きな特徴として、thenableなオブジェクトをpromiseオブジェクトに変換するという機能があります。

ES6 Promisesには Thenableという概念があり、簡単にいえばpromiseっぽいオブジェクトの事を言います。

`.length`を持っているが配列ではないものを `Array like`というのと同じで、`thenable`の場合は`.then`というメソッドを持ってるオブジェクトのことを言います。

thenableなオブジェクトが持つ`then`は、Promiseの持つ`then`と同じような挙動を期待していて、thenableなオブジェクトが持つ元々の`then`を上手く利用できるようにしつつpromiseオブジェクトに変換するという仕組みです。

どういうものがthenableなのかというと、分かりやすい例では
jQuery.ajax()の返り値もthenableです。
jQuery.ajax()`□□□□□　http://api.jquery.com/jQuery.ajax/#jqXHR[jqXHR　Object]　□□□□□□□□□□□□□□□□.then`というメソッドを持っているためです。

```
$.ajax('/json/comment.json');// => `.then`□□□□□□□□□
```

このthenableなオブジェクトを`Promise.resolve`ではpromiseオブジェクトにすることが出来ます。
promiseオブジェクトにすることができれば、`then`や`catch`といった、ES6 Promisesが持つ機能をそのまま利用することが出来るようになります。

Listing 3. thenableをpromiseオブジェクトにする

```
var promise = Promise.resolve($.ajax('/json/comment.json'));// => promise□□□□□□
promise.then(function(value){
   console.log(value);
});
```

WARNING

jQueryとthenable
jQuery.ajax()の返り値も`.then`というメソッドを持った　jqXHR　Objectで、このオブジェクトは　Deferred　Objectのメソッドやプロパティ等を継承しています。
しかし、このDeferred　ObjectはPromises/A+やES6 Promisesに準拠したものではないため、
変換できたように見えて一部欠損してしまう情報がでてしまうという問題があります。
この問題はjQueryの　Deferred　Objectの`then`の挙動が違うために発生します。
そのため、`.then`というメソッドを持っていた場合でも、必ずES6 Promisesとして使えるとは限らない事は知っておくべきでしょう。

- JavaScript Promises: There and back again - HTML5 Rocks
- You're Missing the Point of Promises
- https://twitter.com/hirano_y_aa/status/398851806383452160

多くの場合は、多種多様なPromiseの実装ライブラリがある中でそれらの違いを意識せず使えるように、
共通の挙動である`then`だけを利用して、他の機能は自分自分のPromiseにあるものを利用できるように変換するという意味合いが強いと思います。
このthenableを変換する機能は、以前は`Promise.cast`という名前であった事からも想像できるかもしれません。
ThenableについてはPromiseを使ったライブラリを書くときなどには知っておくべきですが、
通常の利用だとそこまで使う機会がないものかもしれません。

NOTE

ThenableとPromise.resolveの具体的な例を交えたものは
第4章のPromise.resolveとThenableにて詳しく解説しています。

`Promise.resolve`
を簡単にまとめると、「渡した値でFulfilledされるpromiseオブジェクトを返すメソッド」と考えるのがいいでしょう。

また、Promiseの多くの処理は内部的に`Promise.resolve`のアルゴリズムを使って値をpromiseオブジェクトに変換しています。

2.2. Promise.reject

`Promise.reject(error)`は `Promise.resolve(value)` と同じ静的メソッドで`new Promise()`のショートカットとなるメソッドです。

例えば、`Promise.reject(new Error("▫▫▫"))`というのは下記のコードのシンタックスシュガーです。

```
new Promise(function(resolve,reject){
    reject(new Error("▫▫▫"));
});
```

返り値のpromiseオブジェクトに対して、thenの`onRejected`に設定された関数にエラーオブジェクトが渡ります。

```
Promise.reject(new Error("BOOM!")).catch(function(error){
    console.log(error);
});
```

`Promise.resolve(value)` との違いは resolveではなくrejectが呼ばれるという点で、テストコードやデバッグ、一貫性を保つために利用する機会などがあるかもしれません。

2.3. コラム: Promiseは常に非同期?

`Promise.resolve(value)` 等を使った場合、 promiseオブジェクトがすぐにresolveされるので、 `.then` に登録した関数も同期的に処理が行われるように錯覚してしまいます。

しかし、実際には`.then`に登録した関数が呼ばれるのは、非同期のタイミングとなります。

```
var promise = new Promise(function(resolve){
    console.log("inner promise");<i class="conum" data-value="1"></i><b>(1)</b>
    resolve(42);
});
promise.then(function(value){
    console.log(value); <i class="conum" data-value="3"></i><b>(3)</b>
});
console.log("outer promise");<i class="conum" data-value="2"></i><b>(2)</b>
```

上記のコードは数値の順に呼ばれるため、出力結果は以下のように非同期で呼ばれていることがわかります。

```
inner promise
outer promise
42
```

つまり、Promiseは常に非同期で処理が行われているという事になります。

2.4. Promise#then

先ほどの章でPromiseの基本となるインスタンスメソッドである`then`と`catch`の使い方を説明しました。

その中で`.then().catch()`とメソッドチェーンで繋げて書いていたことからもわかるように、Promiseではいくらでもメソッドチェーンを繋げて処理を書いていくことが出来ます。

Listing 4. promiseはメソッドチェーンで繋げて書ける

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

`then`で登録するコールバック関数をそれぞれtaskというものにした時に、taskA → task Bという流れをPromiseのメソッドチェーンを使って書くことが出来ます。

Promiseのメソッドチェーンだと長いので、今後はpromise chainと呼びますが、このpromise chainがPromiseが非同期処理の流れを書きやすい理由の一つといえるかもしれません。

このセクションでは、`then` を使ったpromise chainの挙動と流れについて学んでいきましょう。

2.4.1. promise chain

第一章の例だと、promise chainは then → catch というシンプルな例でしたが、このpromise chainをもっとつなげた場合に、それぞれのpromiseオブジェクトに登録されたonFulfilledとonRejectedがどのように呼ばれるかを見て行きましょう。

NOTE

promise chain - すなわちメソッドチェーンが短い事は良いことです。
この例では説明のために長いメソッドチェーンを用います。

次のようなpromise chainを見てみましょう。

Listing 5. promise-then-catch-flow.js

```
"use strict";
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);
```

このようなpromise chainをつなげた場合、 それぞれの処理の流れは以下のように図で表せます。

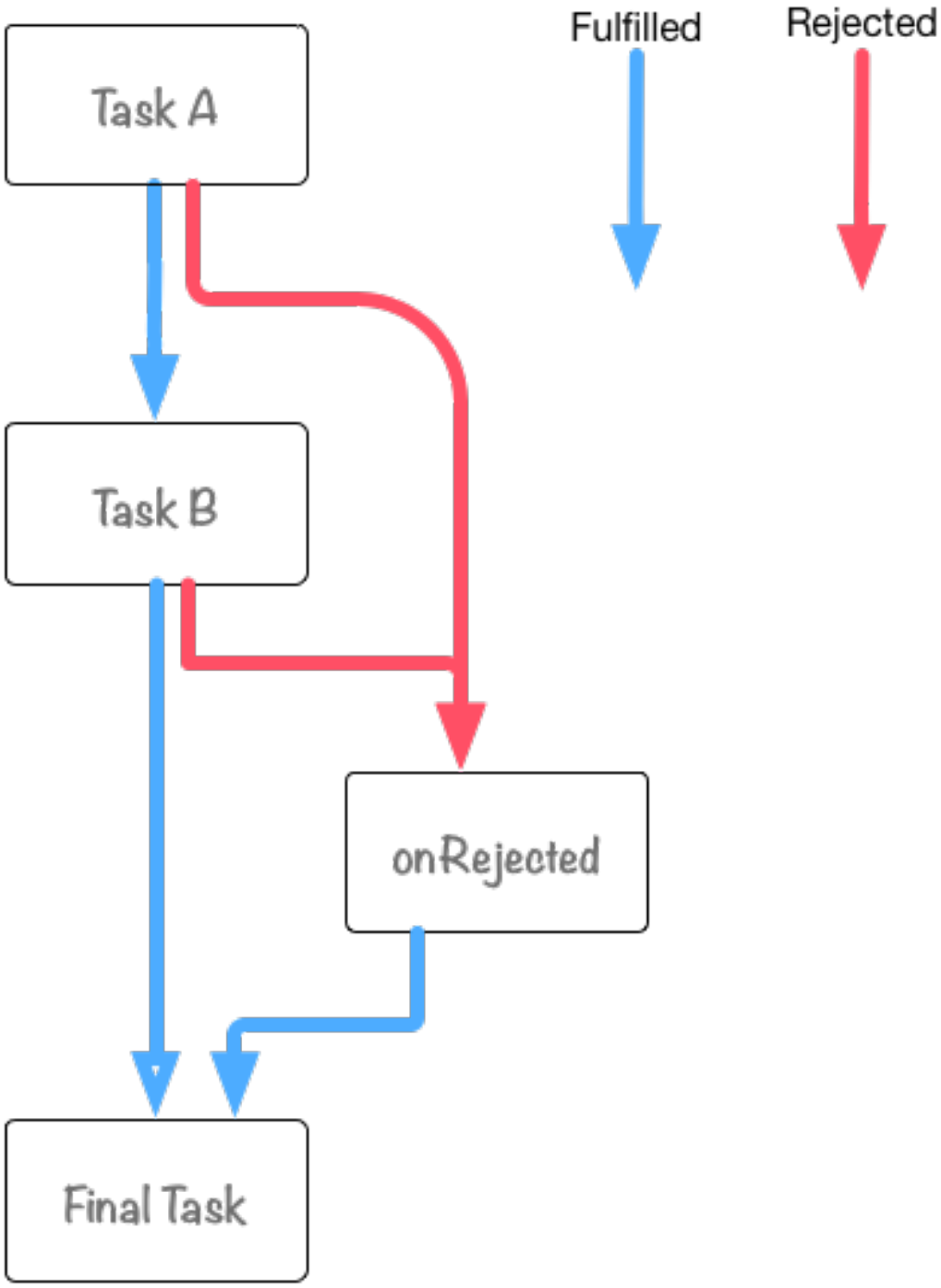


Figure 3. promise-then-catch-flow.jsの図

上記のコードでは`then`は第二引数(onRejected)を使っていないため、
以下のように読み替えても問題ありません。

`then`
: onFulfilledの処理を登録

`catch`
: onRejectedの処理を登録

図の方に注目してもらうと、 Task A と Task B それぞれから onRejected への線が出ていることが分かります。

これは、Task A または Task B の処理にて、以下のどちらかが起きた場合に

- 例外が発生した時
- Rejectedなpromiseオブジェクトがreturnされた時

onRejected が呼ばれるという事を示しています。

第一章でPromiseの処理は常に`try-catch`されているようなものなので、
例外が起きた場合もキャッチして、`catch`で登録された`onRejected`の処理を呼ぶことは学びましたね。

もう一つの Rejectedなpromiseオブジェクトがreturnされた時 については、`throw`を使わずにpromise chain中に`onRejected`を呼ぶ方法です。

これについては、ここでは必要ない内容なので詳しくは、 第4章の throwしないで、rejectしようにて解説しています。

また、onRejected と Final Task には`catch`のpromise chainがこれより後ろにありません。
つまり、この処理中に例外が起きた場合はキャッチすることができないことに気をつけましょう。

もう少し具体的に、Task A → onRejected となる例を見てみます。

Task Aで例外が発生したケース

Task A の処理中に例外が発生した場合、 TaskA → onRejected → FinalTask という流れで処理が行われます。

image::img/promise-taska-rejected-flow.png[promise-taska-rejected-flopush

コードにしてみると以下のようになります。

Listing 6. promise-then-taska-throw.js

```
"use strict";
function taskA() {
    console.log("Task A");
    throw new Error("throw Error @ Task A")
}
function taskB() {
    console.log("Task B");// □□□□□
}
function onRejected(error) {
    console.log(error);// => "throw Error @ Task A"
}
function finalTask() {
    console.log("Final Task");
}

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

実行してみると、Task B が呼ばれていない事がわかるでしょう。

NOTE

例では説明のためにtaskAで`throw`して例外を発生させています。
しかし、実際に明示的にonRejectedを呼びたい場合は、Rejectedなpromiseオブジェクトを返すべきでしょう。 それぞれの違いについては throwしないで、rejectしよう で解説しています。

2.4.2. promise chainでの値渡し

先ほどの例ではそれぞのTaskが独立していて、ただ呼ばれているだけでした。

この時に、Task AがTask Bへ値を渡したい時はどうすれば良いでしょうか?

答えはものすごく単純でTask Aの処理で`return`した値がTask Bが呼ばれるときに引数に設定されます。

実際に例を見てみましょう。

Listing 7. promise-then-passing-value.js

```
"use strict";
"use strict";
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);
```

スタートは`Promise.resolve(1);`で、この処理は以下のような流れてpromise chainが処理されていきます。

1. `Promise.resolve(1);` から 1 が `increment`に渡される
2. `increment`では渡された値に+1した値を`return`している
3. この値(2)が次の`doubleUp`に渡される
4. 最後に`output`が出力する

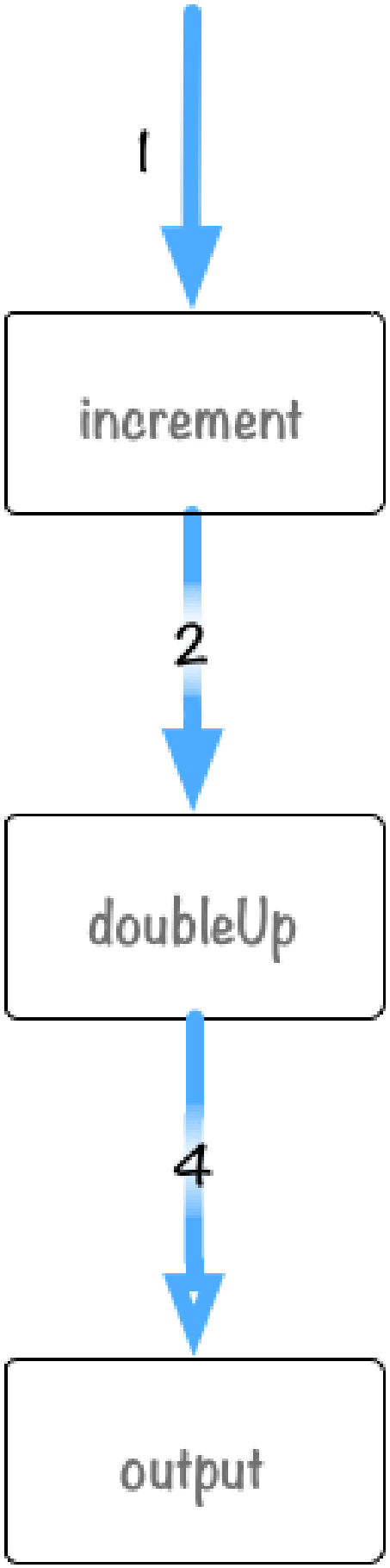


Figure 4. promise-then-passing-value.jsの図

この`return`する値は数字や文字列だけではなく、
オブジェクトやpromiseオブジェクトも`return`することが出来ます。

returnした値は `Promise.resolve(return□□□□);` のような処理されるため、
何をreturnしても最終的にはpromiseオブジェクトが返されるとおぼえておくとよいでしょう。

NOTE

これについて詳しくは thenは常に新しいpromiseオブジェクトを返す にて、
よくある間違いと共に紹介しています。

2.5. Promise#catch

先ほどのPromise#thenについてでも`Promise#catch`は既に使っていましたね。

改めて説明するとPromise#catchは`promise.then(undefined, onRejected);`のエイリアスとなるメソッドです。
つまり、promiseオブジェクトがRejectedとなった時に呼ばれる関数を登録するためのメソッドです。

NOTE

Promise#thenとPromise#catchの使い分けについては、 then or catch?で紹介しています。

2.5.1. IE8以下での問題

<img src="https://ci.testling.com/azu/promise-catch-error.png" alt="Build Status">

このバッジは以下のコードが、 polyfill
を用いた状態でそれぞれのブラウザで正しく実行できているかを示したものです。

NOTE

polyfillとはその機能が実装されていないブラウザでも、その機能が使えるようにするライブラリのことです。 この例では jakearchibald/es6-promise を利用しています。

Listing 8. Promise#catchの実行結果

```
var promise = Promise.reject(new Error("message"));
promise.catch(function (error) {
    console.error(error);
});
```

このコードをそれぞれのブラウザで実行させると、IE8以下では実行する段階で 識別子がありませんというSyntax Errorになってしまいます。

これはどういう事かというと、`catch`という単語はECMAScriptにおける 予約語
であることが関係します。

ECMAScript 3では予約語はプロパティの名前に使うことが出来ませんでした。 IE8以下はECMAScript 3の実装であるため、`catch`というプロパティを使う`promise.catch()`という書き方が出来ないため、識別子がありませんというエラーを起こしてしまう訳です。

一方、現在のブラウザが実装済みであるECMAScript 5以降では、 予約語を IdentifierName

、つまりプロパティ名に利用することが可能となっています。

NOTE

ECMAScript5でも予約語は Identifier 、つまり変数名、関数名には利用することが出来ません。 `for` という変数が定義できてしまうと `for` 文との区別ができなくなってしまいます。 プロパティの場合は `object.for` と `for`文の区別はできるので、少し考えてみると自然な動作ですね。

このECMAScript 3の予約語の問題を回避した書き方も存在します。

ドット表記法 はプロパティ名が有効な識別子(ECMAScript 3の場合は予約語が使えない)でないといけませんが、 ブラケット表記法 は有効な識別子ではなくても利用できます。

つまり、先ほどのコードは以下のように書き換えれば、IE8以下でも実行することが出来ます。(もちろんpolyfillは必要です)

Listing 9. Promise#catchの識別子エラーの回避

```
var promise = Promise.reject(new Error("message"));
promise["catch"](function (error) {
    console.error(error);
});
```

もしくは単純に`catch`を使わずに、`then`を使うことでも回避できます。

Listing 10. Promise#catchではなくPromise#thenを使う

```
var promise = Promise.reject(new Error("message"));
promise.then(undefined, function (error) {
    console.error(error);
});
```

`catch`という識別子が問題となっているため、ライブラリによっては`caught`等の名前が違うだけのメソッドを用意しているケースがあります。

サポートブラウザにIE8以下を含める時は、この`catch`の問題に気をつけるといいでしょう。

2.6. コラム: thenは常に新しいpromiseオブジェクトを返す

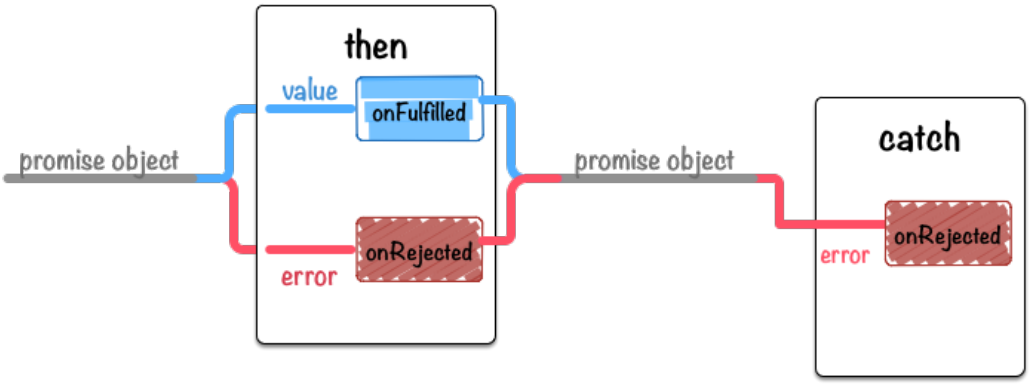
`aPromise.then(...).catch(...)` は一見すると、全て最初の`aPromise`オブジェクトにメソッドチェーンで処理を書いてるように見えます。

しかし、実際には`then`で別のpromiseオブジェクト、`catch`でも別のpromiseオブジェクトを作成して返しています。

本当に新しいpromiseオブジェクトを返しているのか確認してみましょう。

```
var aPromise = new Promise(function (resolve) {
    resolve(100);
});
var thenPromise = aPromise.then(function (value) {
    console.log(value);
});
var catchPromise = thenPromise.catch(function (error) {
    console.error(error);
});
console.info(aPromise !== thenPromise); // => true
console.info(thenPromise !== catchPromise);// => true
```

=== 厳密比較演算子によって比較するとそれぞれが別々のオブジェクトなので、本当に`then`や`catch`は別のpromiseオブジェクトを返していることが分かりました。



この挙動はPromise全般に当てはまるため、`Promise.all`や`Promise.race`も
引数で受け取ったものとは別のpromiseオブジェクトを作って返しています。

この仕組みはPromiseを拡張する時は意識しないと、いつのまにか触ってるpromiseオブジェクトが別のものであったという事が起こりえると思います。

また、`then`は新しいオブジェクトを作って返すということがわかっていれば、
次の`then`の使い方では意味が異なる事に気づくでしょう。

```
<i class="conum" data-value="1"></i><b>(1)</b>
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(value); // => 100
})

// vs

<i class="conum" data-value="2"></i><b>(2)</b>
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(value); // => 100 * 2 * 2
});
```

1のpromiseをメソッドチェーン的に繋げない書き方はあまりすべきではありませんが、
このような書き方をした場合、それぞれの`then`はほぼ同時に呼ばれ、また`value`に渡る値も全て同じ`100`となります。

2はメソッドチェーン的につなげて書くことにより、resolve → then → then → then と書いた順番にキチンと実行され、
それぞれの`value`に渡る値は、一つ前のpromiseオブジェクトで`return`された値が渡ってくるようになります。

1の書き方により発生するアンチパターンとしては以下のようなものが有名です

Listing 11. ✘ `then`の間違った使い方

```
function anAsyncCall() {
    var promise = Promise.resolve();
    promise.then(function() {
        // □□□□□
        return newVar;
    });
    return promise;
}
```

このように書いてしまうと、`promise.then`
の中で例外が発生するとその例外を取得する方法がなくなり、
また、何かの値を返していてもそれを受け取る方法が無くなってしまいます。

これは`promise.then`によって新たに作られたpromiseオブジェクトを返すようにすることで、
2のようにpromise chainがつなげるようにするべきなので、次のように修正することが出来ます。

Listing 12. `then`で作成したオブジェクトを返す

```
function anAsyncCall() {
    var promise = Promise.resolve();
    return promise.then(function() {
        // □□□□□
        return newVar;
    });
}
```

これらのアンチパターンについて、詳しくは Promise Anti-patterns を参照して下さい。

2.7. Promiseと配列

ここまでで、promiseオブジェクトが FulFilled または Rejected となった時の処理は `.then` と `.catch` で登録出来る事を学びました。

一つのpromiseオブジェクトなら、そのpromiseオブジェクトに対して処理を書けば良いですが、
複数のpromiseオブジェクトが全てFulFilledとなった時の処理を書く場合はどうすればよいでしょうか？

例えば、A→B→C
という感じで複数のXHR(非同期処理)を行った後に、何かをしたいという事例を考えてみます。

ちょっとイメージしにくいので、
まずは、通常のコールバックスタイルを使って複数のXHRを行う以下のようなコードを見てみます。

2.7.1. コールバックで複数の非同期処理

Listing 13. multiple-xhr-callback.js

```
'use strict';
function getURLCallback(URL, callback) {
    var req = new XMLHttpRequest();
    req.open('GET', URL, true);
    req.onload = function () {
        if (req.status == 200) {
            callback(null, req.response);
        } else {
            callback(new Error(req.statusText), req.response);
        }
    };
    req.onerror = function () {
        callback(new Error(req.statusText));
    };
```

```
    req.send();
}
// <1> JSONパースを安全に行う
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を叩いてリクエスト
var request = {
        comment: function getComment(callback) {
            return getURLCallback('http://azu.github.io/promises-
book/json/comment.json', jsonParse.bind(null, callback));
        },
        people: function getPeople(callback) {
            return getURLCallback('http://azu.github.io/promises-
book/json/people.json', jsonParse.bind(null, callback));
        }
    };
// <3> 複数の XHRリクエストを行い、全部終わったら    callbackを呼ぶ
function allRequest(requests, callback, results) {
    if (requests.length === 0) {
        return callback(null, results);
    }
    var 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.comment, request.people], callback, []);
}
```

上記のコードを実際に実行して、XHRで取得した結果を得るには次のようになると思います。

```
main(function(error, results){
    console.log(results);
});
```

このコールバックスタイルでは幾つかの要素が出てきます。

- `JSON.parse` をそのまま使うと例外となるケースがあるためラップした`jsonParse`関数を使う
- 複数のXHRをそのまま書くとネストが深くなるため、`allRequest`というrequest関数を実行するものを利用する
- コールバック関数には `callback(error,value)` というNode.jsでよく見られる引数を渡す

`jsonParse` 関数を使うときに `bind` を使うことで、部分適応を使って無名関数を減らすようにしています。
(コールバックスタイルでも関数の処理などをちゃんと分離すれば、無名関数の使用も減らせると思います)

```
jsonParse.bind(null, callback);
// ▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
function(error,value){
    jsonParse(callback, error, value);
}
```

コールバックスタイルで書いたものを見ると以下のような点が気になります。

- 明示的な例外のハンドリングが必要
- ネストを深くしないために、requestを扱う関数が必要
- コールバックがたくさんでてくる

次は、`Promise#then` を使って同様の事をしてみたいと思います。

2.7.2. Promise#thenのみで複数の非同期処理

先に述べておきますが、`Promise.all` というこのような処理に適切なものがあるため、 ワザと`.then`の部分をクドく書いています。

`.then` を使った場合は、コールバックスタイルと完全に同等というわけではないですが以下のように書けると思います。

```
'use strict';
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.response);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-
book/json/comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-
book/json/people.json').then(JSON.parse);
        }
    };
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] □□□□□□□□□□□□□□□□
    var pushValue = recordValue.bind(null, []);
    return request.comment().then(pushValue).then(request.people).then(pushValue);
}
```

上記のコードを実際に実行して、XHRで取得した結果を得るには次のようになると思います。

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

コールバックスタイルと比較してみると次の事がわかります。

- `JSON.parse` をそのまま使っている

- `main()` はpromiseオブジェクトを返している
- エラーハンドリングは返ってきたpromiseオブジェクトに対して書いている

先ほども述べたように mainの `then` の部分がクドく感じます。

このような複数の非同期処理をまとめて扱う `Promise.all` と `Promise.race` という静的メソッドについて 学んでいきましょう。

2.8. Promise.all

先ほどの複数のXHRの結果をまとめたものを取得する処理は、`Promise.all` を使うと次のように書くことが出来ます。

`Promise.all` は配列を受け取り、その配列に入っているpromiseオブジェクトが全てresolveされた時に、次の`.then`を呼び出します。

下記の例では、promiseオブジェクトはXHRによる通信を抽象化したオブジェクトといえるので、全ての通信が完了(resolveまたはreject)された時に、次の`.then`が呼び出されます。

Listing 14. promise-all-xhr.js

```
'use strict';
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.response);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-
book/json/comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-
book/json/people.json').then(JSON.parse);
        }
    };
function main() {
    return Promise.all([request.comment(), request.people()]);
}
```

実行方法は 前回のもの と同じで以下のようにして実行出来ます。

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

`Promise.all` を使うことで以下のような違いがあることがわかります。

- mainの処理がスッキリしている
- Promise.all は promiseオブジェクトの配列を扱っている

```
Promise.all([request.comment(), request.people()]);
```

というように処理を書いた場合は、`request.comment()` と `request.people()`

は同時に実行されますが、 それぞれのpromiseの結果(resolve,rejectで渡される値)は、`Promise.all`に渡した配列の順番となります。

つまり、この場合に次の`.then`に渡される結果の配列は [comment, people]の順番になることが保証されています。

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

`Promise.all` に渡したpromiseオブジェクトが同時に実行されてるのは、次のようなタイマーを使った例を見てみると分かりやすいです。

Listing 15. promise-all-timer.js

```
'use strict';
// promise
function promisedMapping(ary) {
    function timerPromisefy(value) {
        return new Promise(function (resolve) {
            setTimeout(function () {
                resolve(value);    // => return
            }, value);
        });
    }
    return ary.map(timerPromisefy);
}
var promisedMap = promisedMapping([1, 2, 4, 8, 16, 32]);
var startDate = Date.now();
Promise.all(promisedMap).then(function (values) {
    console.log(Date.now() - startDate + 'ms');
    // 32ms
    console.log(values);    // [1, 2, 4, 8, 16, 32]
});
```

`promisedMapping` 数値の配列を渡すと、数値をそのまま`setTimeout`に設定したpromiseオブジェクトの配列を返す関数です。

```
promisedMapping([1, 2, 4, 8, 16, 32]);
```

この場合は、1,2,4,8,16,32
ms後にそれぞれresolveされるpromiseオブジェクトの配列を作って返します。

つまり、このpromiseオブジェクトの配列がすべてresolveされるには最低でも32msかかることがわかります。 実際に`Promise.all` で処理してみると 約32msかかってる事がわかると思います。

この事から、`Promise.all` が一つづつ順番にやるわけではなく、渡されたpromiseオブジェクトの配列を並列に実行してるという事がわかると思います。

NOTE

仮に逐次的に行われていた場合は、 1ms待機 → 2ms待機 → 4ms待機 → … → 32ms待機 となるので、 全て完了するまで64ms程度かかる計算になる

TIP

逐次的に実行した場合は、 xhr-promise.js で紹介したような`.then`を重ねていくような書き方が必要になります

多くのライブラリでは、同様の機能をするメソッドが用意されているが、以下のような感じで書くことが出来る

- Promise.reduce 的な機能の紹介

2.9. Promise.race

`Promise.all`と同様に複数のpromiseオブジェクトを扱う`Promise.race`を見てみましょう。

使い方はPromise.allと同様で、promiseオブジェクトの配列を引数に渡します。

`Promise.all`は、渡した全てのpromiseが解決状態になるまで次の処理を待ちましたが、`Promise.race`は、どれか一つでもpromiseが解決状態になったら次の処理を実行します。

Promise.allの時と同じく、タイマーを使った`Promise.race`の例を見てみましょう

Listing 16. promise-race-timer.js

```
'use strict';
// 配列の中身をそれぞれ promiseオブジェクトにした配列を返す
function promisedMapping(ary) {
    function timerPromisefy(value) {
        return new Promise(function (resolve) {
            setTimeout(function () {
                resolve(value);    // => returnする値
            }, value);
        });
    }
    return ary.map(timerPromisefy);
}
var promisedMap = promisedMapping([1, 32, 64, 128]);
// 一番最初のものが resolveされた時点で終了
Promise.race(promisedMap).then(function (value) {
    console.log(value);    // => 1
});
```

上記のコードだと、1ms後、32ms後、64ms後、128ms後にそれぞれpromiseオブジェクトが解決されますが、 一番最初に1msのものがresolveされた時点で、`.then`が呼ばれ、`resolve(1)`なので`value`に渡される値も1となります。

最初に解決したpromiseオブジェクト以外は、その時点で呼ばれるのかを見てみましょう

Listing 17. promise-race-other.js

```
'use strict';
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);
    });
// □□□□□□□□resolve□□□□□□□□
Promise.race([winnerPromise, loserPromise]).then(function (value) {
    console.log(value);    // => 'this is winner'
});
```

先ほどのコードに `console.log` をそれぞれ追加しただけの内容となっています。

実行してみると、winnter/loser　どちらの`setTimeout`　の中身が実行されて　`console.log` がそれぞれ出力されている事がわかります。

つまり、`Promise.race`では、
一番最初のpromiseオブジェクトがFulfilledとなっても、他のpromiseがキャンセルされるわけでは無いという事がわかります。

NOTE

ES6 Promisesの仕様には、キャンセルという概念はありません。 必ず、resolve or rejectによる状態の解決が起こることが前提となっています。　つまり、状態が固定されてしまうかもしれない処理には不向きであると言えます。
ライブラリによってはキャンセルを行う仕組みが用意されている場合があります。

2.10. then or catch?

前の章で `.catch` は `promise.then(undefined, onRejected)` であるという事を紹介しました。

この書籍では基本的には、`.catch`を使い　`.then` とは分けてエラーハンドリングを書くようにしています。

ここでは、`.then`でまとめて指定した場合と、どのような違いがでるかについて学んでいきましょう。

2.10.1. エラー処理ができないonRejected

次のようなコードを見ていきます。

Listing 18. then-throw-error.js

```
'use strict';
function throwError(value) {
    // 例外を投げる
    throw new Error(value);
}
// <1> onRejectedが呼ばれることはない
function badMain(onRejected) {
    return Promise.resolve(42).then(throwError, onRejected);
}
// <2> onRejectedが例外発生時に呼ばれる
function goodMain(onRejected) {
    return Promise.resolve(42).then(throwError).catch(onRejected);
}
```

このコード例では、(必ずしも悪いわけではないですが)良くないパターンの `badMain` とちゃんとエラーハンドリングが行える `goodMain` があります。

`badMain` がなぜ良くないかというと、`.then`の第二引数にはエラー処理を書くことが出来ますが、そのエラー処理は第一引数の`onFulfilled`で指定した関数内で起きたエラーをキャッチする事は出来ません。

つまり、この場合、`theError` でエラーがおきても、`onRejected` に指定した関数は呼ばれることなく、 どこでエラーが発生したのかわからなくなってしまいます。

それに対して、 `goodMain` は `theError`→`onRejected`となるように書かれています。 この場合は `theError` でエラーが発生しても、次のchainである`.catch`が呼ばれるため、エラーハンドリングを行う事が出来ます。

`.then`のonRejectedが扱う処理は、その(またはそれ以前の)promiseオブジェクトに対してであって、`.then`に書かれたonFulfilledは対象ではないためこのような違いが生まれます。

NOTE

`.then`や`.catch`はその場で新しいpromiseオブジェクトを作って返します。
Promiseではchainする度に異なるpromiseオブジェクトに対して処理を書くようになっています。

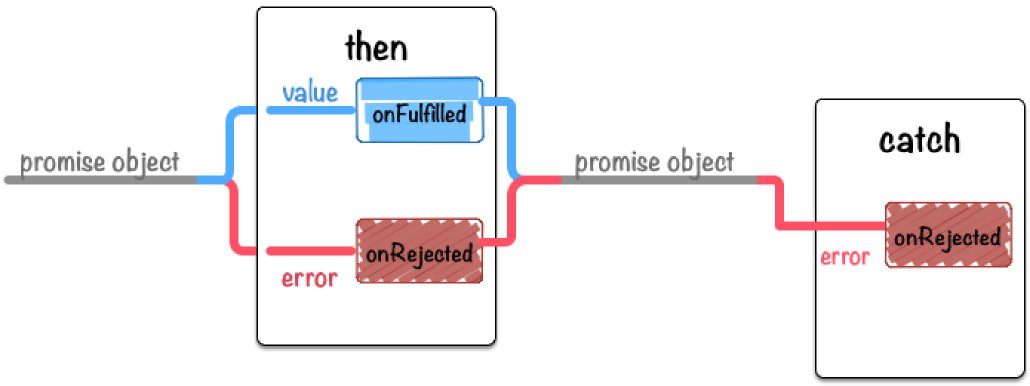


Figure 5. Then Catch flow

この場合の `then` は `Promise.resolve(42)` に対する処理となり、`onFulfilled`で例外が発生しても、同じ`then`で指定された`onRejected`はキャッチすることはありま

せん。

この`then`で発生した例外をキャッチ出来るのは、次のchainで書かれた`catch`となります。

もちろん`.catch`は`.then`のエイリアスなので、下記のように`.then`を使っても問題はありませんが、`.catch`を使ったほうが意図が明確で分かりやすいでしょう。

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

2.10.2. まとめ

ここでは次のような事について学びました。

1. `promise.then(onFulfilled, onRejected)` において
   - `onFulfilled` で例外がおきても、この`onRejected`はキャッチできない
2. `promise.then(onFulfilled).catch(onRejected)`とした場合
   - `then`□□□□□□□□.catch`でキャッチできる
3. `.then`と`.catch`に本質的な意味の違いはない
   - 使い分けると意図が明確になる

`badMain`のような書き方をすると、意図とは異なりエラーハンドリングができないケースが存在することは覚えておきましょう。

3. Chapter.3 - Promiseのテスト

この章ではPromiseのテストの書き方について学んで行きます。

3.1. 基本的なテスト

ES6 Promisesのメソッド等についてひと通り学ぶことができたため、実際にPromiseを使った処理を書いていくことは出来ると思います。

そうした時に、次にどうすればいいのか悩むのがPromiseのテストの書き方です。

ここではまず、 Mochaを使った基本的なPromiseのテストの書き方について学んでいきましょう。

またこの章でのテストコードはNode.js環境で実行することを前提としています。

- この書籍のソースコードへのリンク

3.1.1. Mochaとは

ここでは、 Mocha自体については詳しく解説しませんが、MochaはNode.js製のテストフレームワークツールです。

MochaはBDD,TDD,exportsのどれかのスタイルを選択でき、テストに使うアサーションメソッドも任意のライブラリと組わせて利用します。
つまり、Mocha自体はテスト実行時の枠だけを提供しており、他は利用者が選択するというものになっています。

Mochaを選んだ理由としては、以下の点で選択しました。

- 著名なテストフレームワークであること
- Node.jsとブラウザ どちらのテストもサポートしている
- "Promiseのテスト"をサポートしている

最後の_"Promiseのテスト"をサポートしている_とはどういうことなのかについては後ほど解説します。

また、アサーションライブラリには、power-assertを利用しますが、アサーション自体はNode.jsの`assert`モジュールと全く同じであるため、今回はあまり気にしなくても問題ありません。

まずは、コールバック関数のテストと同じような形でテストを書いてみましょう。

3.1.2. コールバックスタイルのテスト

コラム: Promiseは常に非同期?で確認したように、Promiseでは`then`で登録した関数が呼ばれるタイミングは常に非同期となります。

まずはコールバックスタイルと同じようにPromiseのテストを書いてみましょう。

Listing 19. basic-test.js

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

Mochaは`it`の仮引数に`done`という感じで指定してあげると、
`done()`が呼ばれるまでテストケースが終了しなくなることで非同期のテストをサポートしています。

次のコールバックスタイルのテストは以下のような流れになっています。

```
it("should use `done` for test", function (done) {
    <i class="conum" data-value="1"></i><b>(1)</b>
    setTimeout(function () {
        assert(true);
        done();<i class="conum" data-value="2"></i><b>(2)</b>
    }, 0);
});
```

よく見かける形の書き方ですね。

3.1.3. `done`を使ったPromiseのテスト

次に、Promiseのテストの方を見てみましょう。

```
it("should use `done` for test?", function (done) {
    var promise = Promise.resolve(1);<i class="conum" data-
value="1"></i><b>(1)</b>
    promise.then(function (value) {
        assert(value === 1);
        done();<i class="conum" data-value="2"></i><b>(2)</b>
    });
});
```

`Promise.resolve` はpromiseオブジェクトを返し、 そのpromiseオブジェクトはresolveされます。

コラム: Promiseは常に非同期? でも出てきたように、promiseオブジェクトは常に非同期で処理されるため、テストも非同期に対応した書き方が必要となります。

これで、Promiseのテストもできてるように見えますが、
上記のテストコードでは`assert`が失敗した場合に問題が発生します。

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

`assert`が失敗してる例、この場合「テストは失敗する」と思うかもしれませんが、
実際にはテストが終わることがなくタイムアウトします。

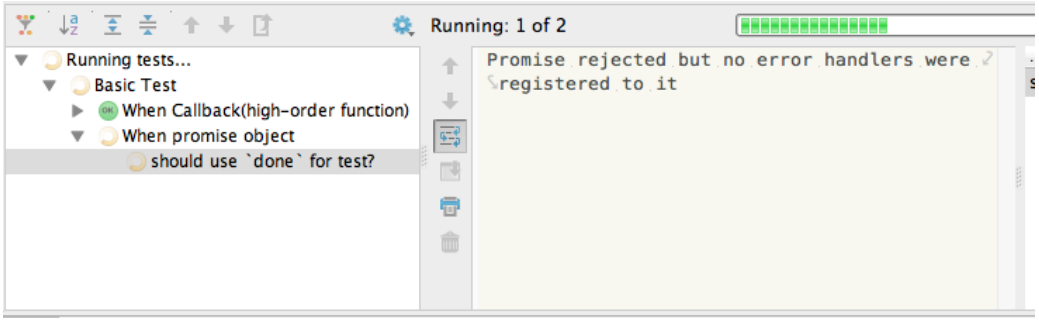


Figure 6. promise test timeout

`assert`が失敗した場合は通常はエラーをthrowするため、
テストフレームワークがそれをキャッチすることで、テストが失敗したと判断します。

しかし、Promiseの場合は`.then`の中で行われた処理でエラーが発生しても、
Promiseがそれをキャッチしてしまい、テストフレームワークまでエラーがthrowされません。

`assert`が失敗してる例を改善して、
`assert`が失敗した場合にちゃんとテストが失敗となるようにしてみましょう。

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

ちゃんとテストが失敗する例では、必ず`done`が呼ばれるようにするため、 最後に`.then(done, done);`を追加しています。

`assert`がパスした場合は単純に`done()`が呼ばれ、`assert`が失敗した場合は`done(error)`が呼ばれます。 これでようやく
コールバックスタイルのテストと同等のPromiseのテストを書くことができました。

しかし、`assert`□□□□□□□□□□.then(done, done);`というものを付ける必要があります。
毎回やるにはつけ忘れてしまうこともあるため、あまりテストしやすいとは言えないかもしれません。

次に、最初にmochaを使う理由に上げた"Promisesのテスト"をサポートしているという事がどういう機能なのかを学んでいきましょう。

3.2. MochaのPromiseサポート

Mochaがサポートしてるpromiseのテストとは何かについて学んでいきましょう。

公式サイトの Asynchronous codeにもその概要が書かれています。

> 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:

Promiseのテストの場合は`done()`の代わりに、promiseオブジェクトをreturnすることでできると書

いてあります。

実際にどういう風に書くかの例を見て行きたいと思います。

Listing 20. mocha-promise-test.js

```
"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);
        });
    });
});
```

先ほどの`done`を使った例をMochaのPromiseテストの形式に変更しました。

変更点としては以下の2箇所です

- done そのものを取り除いた
- テストしたい`assert`が登録されてるpromiseオブジェクトを返すようにした

この書き方をした場合は、`assert`が失敗した場合はもちろんテストが失敗します。

```
it("should be fail", function () {
    return Promise.resolve().then(function () {
        assert(false);// => ▫▫▫▫▫▫▫▫▫
    });
});
```

これにより`.then(done,
done);`というような本質的にはテストに関係ない記述を省くことが出来るようになりました。

NOTE

MochaがPromisesのテストをサポートしました | Web scratch という記事でもMochaのPromiseサポートについて書かれています。

3.2.1. 意図しないテスト結果

MochaがPromiseのテストをサポートしているため、これでよいと思われるかもしれませんが、
この書き方にも意図しない結果になる例外が存在します。

例えば、以下はある条件だとrejectされるコードがあり、
そのエラーメッセージをテストしたいという目的のコードを簡略化したものです。

このテストの目的

`mayBeRejected()`がresolveした場合
    テストを失敗させる

`mayBeRejected()`がrejectした場合
    `assert`でErrorオブジェクトをチェックする

```
function mayBeRejected(){ <i class="conum" data-value="1"></i><b>(1)</b>
    return Promise.reject(new Error("woo"));
}
it("is bad pattern", function () {
    return mayBeRejected().catch(function (error) {
        assert(error.message === "woo");
    });
});
```

この場合は、`Promise.reject`は`onRejected`に登録された関数を呼ぶため、 テストはパスしますね。
このテストで問題になるのは`mayBeRejected()`で返されたpromiseオブジェクトが
resolveされた場合に、必ずテストがパスしてしまうという問題が発生します。

```
function mayBeRejected(){ <i class="conum" data-value="1"></i><b>(1)</b>
    return Promise.resolve();
}
it("is bad pattern", function () {
    return mayBeRejected().catch(function (error) {
        assert(error.message === "woo");
    });
});
```

この場合、`catch`で登録した`onRejected`の関数はそもそも呼ばれないため、
`assert`がひとつも呼ばれることなくテストが必ずパスしてしまいます。
これを解消しようとして、`.catch`▫▫▫.then`を入れて、
`.then`が呼ばれたらテストを失敗にしたいと考えるかもしれません。

```
function failTest() { <i class="conum" data-value="1"></i><b>(1)</b>
    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");
    });
});
```

しかし、この書き方だとthen or catch?で紹介したように、
`failTest`で投げられたエラーが`catch`されてしまいます。

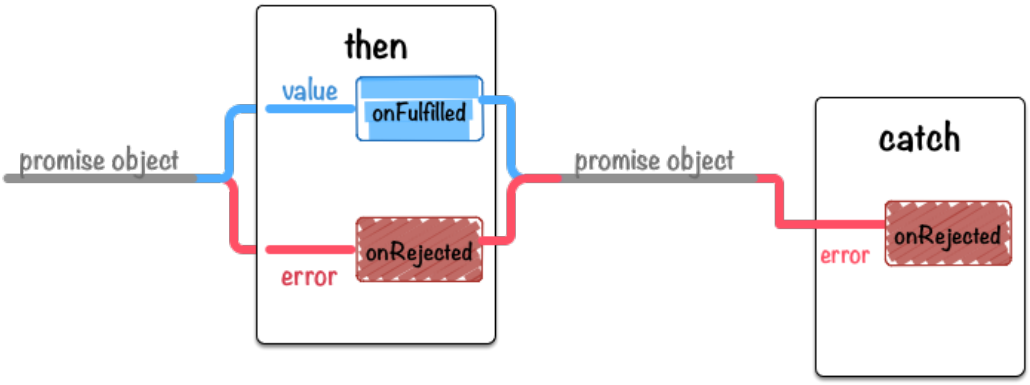


Figure 7. Then Catch flow

`then` → `catch` となり、`catch`に渡ってくるErrorオブジェクトは`AssertionError`となり、意図したものとは違うものが渡ってきてしまいます。

つまり、onRejectedになることだけを期待して書かれたテストは、onFulfilledの状態になってしまうと 常にテストがパスしてしまうという問題を持っていることが分かります。

3.2.2. 両状態の明示して意図しないテストを改善

上記のエラーオブジェクトをテストしたい場合は、どうすればよいでしょうか?

先ほどとは逆に `catch` → `then` とした場合は、以下のように意図した挙動になります。

resolveした場合
　意図した通りテストが失敗する

rejectした場合
　`assert`でテストを行える

```
function mayBeRejected() {
    return Promise.resolve();
}
it("catch -> then", function () {
    return mayBeRejected().catch(function (error) {
        assert(error.message === "woo");
    }).then(failTest);<i class="conum" data-value="1"></i><b>(1)</b>
});
```

このコードをよく見てみると、`.then(onFulfilled, onRejected)`の一つにまとめられることに気付きます。

```
function mayBeRejected() {
    return Promise.resolve();
}
it("catch -> then", function () {
    return mayBeRejected().then(failTest, function (error) {
        assert(error.message === "woo");
    });
});
```

つまり、onFulfilled、onRejected 両方の状態についてどうなるかを明示する必要があるわけです。

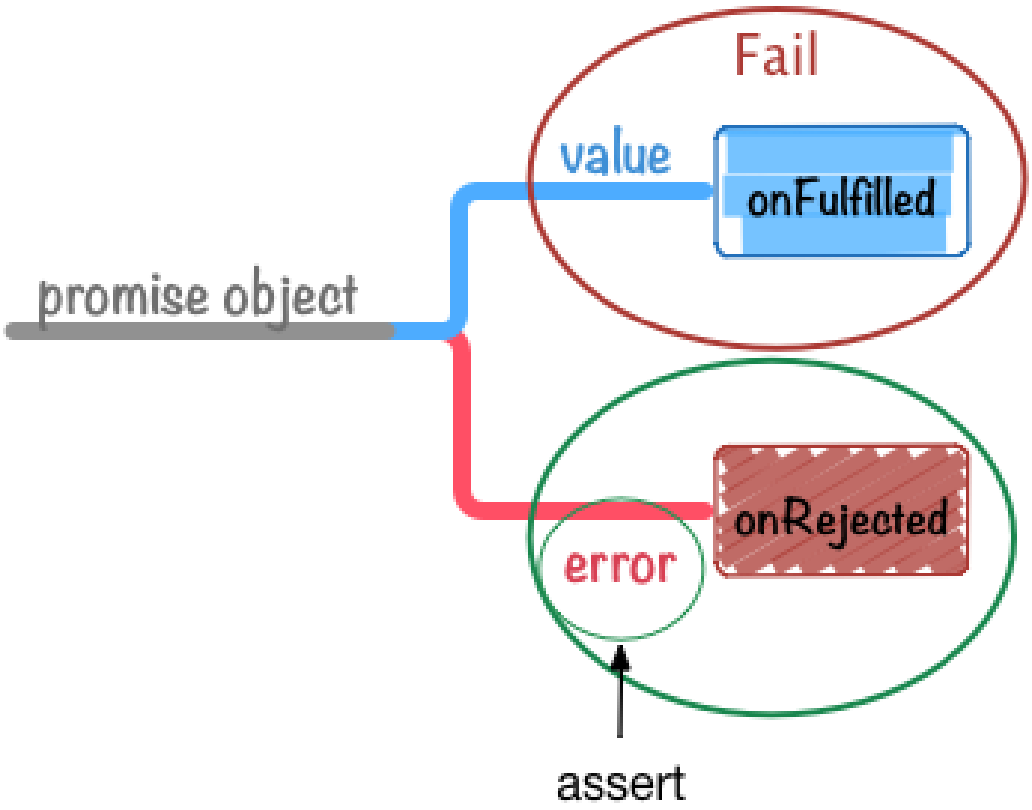


Figure 8. Promise onRejected test

then or catch?の時は、エラーの見逃しを避けるため、`.then(onFulfilled, onRejected)`□□□□□□□□□□`then` → `catch`と分けることを推奨していました。

しかし、テストの場合はPromiseの強力なエラーハンドリングが逆にテストの邪魔をしてしまいます。そのため`.then(onFulfilled, onRejected)`というように指定事でより簡潔にテストを書くことが出来ました。

3.2.3. まとめ

MochaのPromiseサポートについてと意図しない挙動となる場合について紹介しました。

- 通常のコードは`then` → `catch`と分けた方がよい
  - エラーハンドリングのため。then or catch?を参照
- テストコードは`then`にまとめた方がよい?
  - アサーションエラーがテストフレームワークに届くようにするため。

`.then(onFulfilled, onRejected)`を使うことで、promiseオブジェクトがonFulfilled、onRejectedどちらの状況になることを明示してテストすることが出来ます。

しかし、onRejectedのテストであることを明示するために、以下のように書くのはあまり直感的ではないと思います。

```
promise.then(failTest, function(error){
    // assert▫error▫▫▫▫▫▫
}
```

次は、Promiseのテストを手助けするヘルパー関数を定義して、
もう少し分かりやすいテストを書くにはするべきかについて見て行きましょう。

3.3. 意図したテストを書くには

ここでいう意図したテストとは以下のような定義で進めます。

あるpromiseオブジェクトをテスト対象として

- onFulfilledされることを期待したテストを書いた時
  - onRejectedされた場合はFail
  - assertionの結果が一致しなかった場合はFail
- onRejectedされることを期待したテストを書いた時
  - onFulfilledされた場合はFail
  - assertionの結果が一致しなかった場合はFail

つまり、ひとつのテストケースにおいて以下のことを書く必要があります。

- Fulfilled or Rejected どちらを期待するか
- assertionで渡された値のチェック

以下のコードはRejectedを期待したテストとなっていますね。

```
promise.then(failTest, function(error){
    // assert▫error▫▫▫▫▫▫
    assert(error insteadof Error);
}
```

3.3.1. どちらの状態になるかを明示する

意図したテストにするためには、promiseの状態がFulfilled or Rejected
どちらの状態になって欲しいかを明示する必要があります。

しかし、`.then`だと引数は省略可能なので、テストが落ちる条件を入れ忘れる可能性もあります。
そこで、状態を明示できるヘルパー関数を定義してみましょう。

NOTE

ライブラリ化したものがazu/promise-test-helperにありますが、
今回はその場で簡単に定義して進めます。

まずは、先ほどの`.then`の例を元にonRejectedを期待してテスト出来る
`shouldRejected`というヘルパー関数を作ってみたいと思います。

Listing 21. shouldRejected-test.js

```
'use strict';
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`にpromiseオブジェクトを渡すと、`catch`というメソッドをもつオブジェクトを返します。

この`catch`にはonRejectedで書くものと全く同じ使い方ができるので、
`catch`の中にassertionによるテストを書けるようになっています。

`shouldRejected`で囲む以外は通常のpromiseの処理と似た感じになるので以下のようになります。

1. `shouldRejected` にテスト対象のpromiseオブジェクトを渡す
2. 返ってきたオブジェクトの`catch`メソッドでonRejectedの処理を書く
3. onRejectedにassertionによるテストを書く

内部で`then`を使った場合と同様に、onFulfilledが呼ばれた場合はエラーをthrowしてテストが失敗する用になっています。

```
promise.then(failTest, function(error){
    // assert□error□□□□□□
}
// => □□□□□□□□□□□□□□□
return promise.then(function () {
    throw new Error('Expected promise to be rejected but it was fulfilled');
}, function (reason) {
    fn.call(promise, reason);
});
```

`shouldRejected`のようなヘルパー関数を使うことで、
promiseオブジェクトがFulfilledになった場合はテストが失敗するためテストが意図したものとなります。

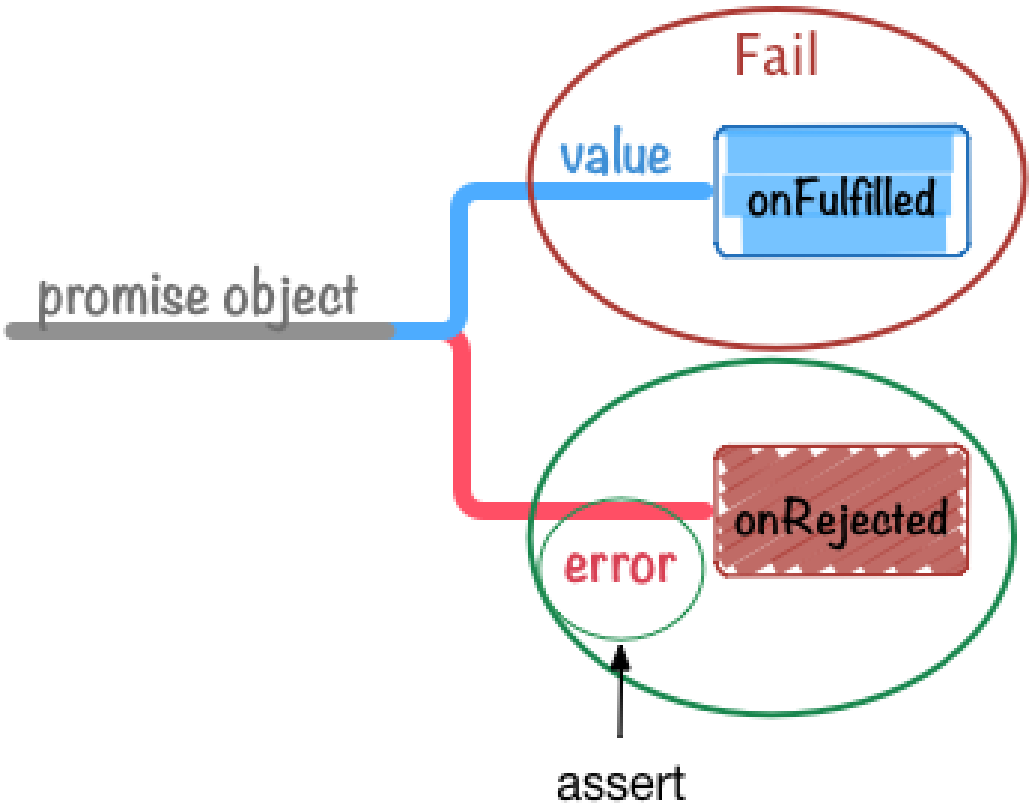


Figure 9. Promise onRejected test

同様に、promiseオブジェクトがFulfilledになることを期待する`shouldFulfilled`も書いてみましょう。

Listing 22. shouldFulfilled-test.js

```
'use strict';
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');
    });
});
```

shouldRejected-

test.jsと基本は同じで、返すオブジェクトの`catch`が`then`になって中身が逆転しただけですね。

3.3.2. まとめ

Promiseで意図したテストを書くためにはどうするか、またそれを補助するヘルパー関数について学びました。

今回書いた`shouldFulfilled`と`shouldRejected`をライブラリ化したものは azu/promise-test-helper にあります。

また、今回のヘルパー関数はMochaのPromiseサポートを前提とした書き方なので、
`done`を使ったテストは利用しにくいと思います。

テストフレームワークのPromiseサポートを使うか、`done`のようにコールバックスタイルのテストを使うかは、 人それぞれのスタイルの問題であるためそこまではっきりした優劣はないと思います。

例えば、 CoffeeScriptでテストを書いたりすると、CoffeeScriptには暗黙のreturnがあるので、`done`を使ったほうが分かりやすいかもしれません。

Promiseのテストは普通に非同期関数のテスト以上に落とし穴があるため、
どのスタイルを取るかは自由ですが、一貫性を持った書き方をすることが大切だと言えます。

4. Chapter.4 - Advanced

この章では、これまでに学んだことの応用や発展した内容について学んでいきます。

4.1. Promise.resolveとThenable

Promise.resolveにて、`Promise.resolve`の大きな特徴の一つとしてthenableなオブジェクトを変換する機能について紹介しました。

このセクションでは、thenableなオブジェクトからpromiseオブジェクトに変換してどのように利用するかについて学びたいと思います。

4.1.1. Web Notificationsをthenableにする

Web Notificationsという デスクトップ通知を行うAPIを例に考えてみます。

Web Notifications APIについて詳しくは以下を参照して下さい。

- Web Notifications の使用 - WebAPI | MDN
- Can I use Web Notifications

Web Notifications APIについて簡単に解説すると、以下のように`new Notification`をすることで通知メッセージが表示できます。

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

しかし、通知を行うためには、`new Notification`をする前にユーザーに許可を取る必要があります。

Notificationのダイアログの選択肢はFirefoxだと永続かセッション限り等で4種類ありますが、
最終的に`Notification.permission`に入ってくる値は許可("granted")か不許可("denied")の2種類です。

許可ダイアログは`Notification.requestPermission`を実行すると表示され、
ユーザーが選択した内容が`status`に渡されます。

```
Notification.requestPermission(function (status) {
    // status␣"granted" or "denied"␣␣␣
});
```

許可時("granted")
　　`new Notification`で通知を作成

不許可時("denied")
　　何もしない

まとめると以下のようになります。

- ユーザーに通知の許可を受け付ける非同期処理がある
- 許可がある場合は`new Notification`で通知を表示できる
  - 既に許可済みのケース
  - その場で許可を貰うケース
- 許可がない場合は何もしない

いくつか許可のパターンが出ますが、シンプルにまとめると
許可がある場合は`onFulfilled`、許可がない場合は`onRejected`と書くことができると思います。

いきなりこれを`thenable`にするのは分かりにくいので、まずは今まで学んだPromiseを使って
promiseオブジェクトを返すラッパー関数を書いてみましょう。

4.1.2. Web Notification as Promise

Listing 23. notification-as-promise.js

```
'use strict';
function notifyMessageAsPromise(message, options) {
    return new Promise(function (resolve, reject) {
        if (Notification && Notification.permission === 'granted') {
            var notification = new Notification(message, options);
            resolve(notification);
        } else if (Notification) {
            Notification.requestPermission(function (status) {
                if (Notification.permission !== status) {
                    Notification.permission = status;
                }
                if (status === 'granted') {
                    var notification = new Notification(message, options);
                    return resolve(notification);
                } else {
                    reject(new Error('user denied'));
                }
            });
        } else {
            reject(new Error('doesn\'t support Notification API'));
        }
    });
}
```

これを使うと`"Hi!"`というメッセージを通知したい場合以下のように書くことが出来ます。

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

許可あるor許可された場合は`.then`が呼ばれ、ユーザーが許可しなかった場合は`.catch`が呼ばれます。

上記のnotification-as-promise.jsは、とても便利そうですが実際に使うときに以下の問題点があります。

- Promiseをサポートしてない(orグローバルに`Promise`のshimがない)環境では使えない

notification-as-promise.jsのようなPromiseスタイルで使えるライブラリを作る場合、
ライブラリ作成者には以下のような選択肢があると思います。

- `Promise`があることを前提とする
  - 利用者に`Promise`があることを保証してもらう
- ライブラリ自体に`Promise`の実装を入れてしまう
  - 例) localForage
- コールバックでも使う事ができ、`Promise`でも使えるようにする

- 利用者がどちらを使うかを選択出来るようにする

notification-as-promise.jsは`Promise`があることを前提としたような書き方です。

本題に戻りThenableはここでいう"コールバックでも使う事ができ、`Promise`でも使えるようにする"という事を 実現するのに役立つ概念です。

4.2. Thenableでコールバックと両立する

まずは先程のWeb Notification APIのラッパー関数をコールバックスタイルで書いてみましょう。

Listing 24. notification-callback.js

```
'use strict';
function notifyMessage(message, options, callback) {
    if (Notification && Notification.permission === 'granted') {
        var 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') {
                var notification = new Notification(message, options);
                callback(null, notification);
            } else {
                callback(new Error('user denied'));
            }
        });
    } else {
        callback(new Error('doesn\'t support Notification API'));
    }
}
```

これを利用する場合は以下のような感じになります。

```
// □□□□□`Notification`□□□□□□□□□□□□□□□
notifyMessage("message", {}, function (error, notification) {
    if(error){
        console.error(error);
        return;
    }
    console.log(notification);// □□□□□□□□□
});
```

コールバックスタイルでは、許可がない場合は`error`に値が入り、
許可がある場合は通知が行われて`notification`に値が入ってくるという感じにしました。

```
function (error, notification){}
```

4.2.1. thenableを返すメソッドを追加する

thenableというのは`.then`というメソッドを持ってるオブジェクトのことを言いましたね。
次にnotification-callback.jsに`thenable`を返すメソッドを追加してみましょう。

Listing 25. notification-thenable.js

```
'use strict';
function notifyMessage(message, options, callback) {
    if (Notification && Notification.permission === 'granted') {
        var 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') {
                var notification = new Notification(message, options);
                callback(null, notification);
            } else {
                callback(new Error('user denied'));
            }
        });
    } else {
        callback(new Error('doesn\'t support Notification API'));
    }
}
// `thenable` ▫▫▫
notifyMessage.thenable = function (message, options) {
    return {
        'then': function (resolve, reject) {
            notifyMessage(message, options, function (error, notification) {
                if (error) {
                    reject(error);
                } else {
                    resolve(notification);
                }
            });
        }
    };
};
```

notification-thenable.js
には`notifyMessage.thenable`をというそのままのメソッドを追加してみました。
返すオブジェクトには`then`というメソッドがあります。

`then`メソッドの仮引数には`new Promise(function (resolve, reject){}`と同じように、解決した時に呼ぶ`resolve`と、棄却した時に呼ぶ`reject`が渡ります。

`then`メソッドがやっている中身はnotification-as-promise.jsの`notifyMessageAsPromise`と同じですね。

この`thenable`を使う場合は以下のように`Promise.resolve(thenable)`を使ってpromiseオブジェクトとして利用できます。

```
Promise.resolve(notifyMessage.thenable("message")).then(function (notification) {
    console.log(notification);// □□□□□□□□□
}).catch(function(error){
    console.error(error);
});
```

Thenableを使ったnotification-thenable.jsとPromiseに依存したnotification-as-promise.jsは、非常に似た使い方ができることがわかります。

notification-thenable.jsにはnotification-as-promise.jsとは次のような違いがあります。

- ライブラリ側に`Promise`実装そのものはでてこない
  - 利用者が`Promise.resolve(thenable)`を使い`Promise`の実装を与える
- Promiseとして使う時に`Promise.resolve(thenable)`と一枚挟む必要がある
- コールバックスタイル(`notifyMessage()`)でも利用できる

thenableオブジェクトを利用することで、
既存のコールバックスタイルとPromiseの親和性を高めることができる事が分かります。

4.3. throwしないで、rejectしよう

Promiseコンストラクタや、`then`で実行される関数は基本的に、
`try…catch`で囲まれてるような状態なので、その中で`throw`をしてもプログラムは終了しません。

Promiseの中で`throw`による例外が発生した場合は自動的に`try…catch`され、そのpromiseオブジェクトはRejectedとなります。

```
var promise = new Promise(function(resolve ,reject){
    throw new Error("message");
});
promise.catch(function(error){
    console.error(error);// => "message"
})
```

このように書いても動作的には問題ありませんが、promiseオブジェクトの状態をRejectedにしたい場合は `reject`という与えられた関数を呼び出すのが一般的です。

先ほどのコードは以下のように書くことが出来ます。

```
var promise = new Promise(function(resolve ,reject){
    reject(new Error("message"));
});
promise.catch(function(error){
    console.error(error);// => "message"
})
```

`throw`が`reject`に変わったと考えれば、`reject`にはErrorオブジェクト渡すべきであるということが分かりやすいかもしれません。

4.3.1. なぜrejectした方がいいのか

そもそも、promiseオブジェクトの状態をRejectedにしたい場合に、
何故`throw`ではなく`reject`した方がいいのでしょうか?

ひとつは`throw`が意図したものか、それとも本当に例外なのか区別が難しくなってしまうことにあります。

例えば、Chrome等の開発者ツールには例外が発生した時に、
デバッガーが自動でbreakする機能が用意されています。

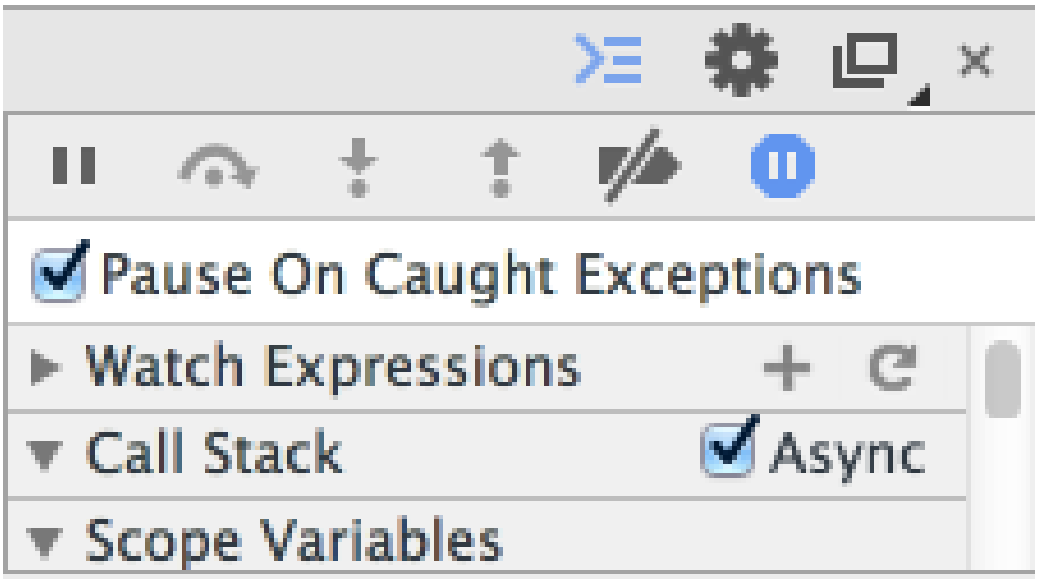


Figure 11. Pause On Caught Exceptions

この機能を有効にしていた場合、以下のように`throw`するとbreakしてしまいます。

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

本来デバッグとは関係ない場所でbreakしてしまうため、
Promiseの中で`throw`している箇所があると、この機能が殆ど使い物にならなくなってしまうでしょう。

4.3.2. thenでもrejectする

Promiseコンストラクタの中では`reject`という関数そのものがあるので、
`throw`を使わないでpromiseオブジェクトをRejectedにするのは簡単でした。

では、次のような`then`の中でrejectしたい場合はどうすればいいでしょうか?

```
var promise = Promise.resolve();
promise.then(function (value) {
    setTimeout(function () {
        // 一定時間経って終わらなかったら　rejectしたい　<i class="conum" data-value="2"></i><b>(2)</b>
    }, 1000);
    // 時間がかかる処理　<i class="conum" data-value="1"></i><b>(1)</b>
    somethingHardWork();
}).catch(function (error) {
    // タイムアウトエラー　<i class="conum" data-value="3"></i><b>(3)</b>
});
```

いわゆるタイムアウト処理ですが、`then`の中で`reject`を呼びたいと思った場合に、
コールバック関数に渡ってくるのは一つ前のpromiseオブジェクトの返した値だけなので困ってしまいます。

> NOTE
>
> Promiseを使ったタイムアウト処理の実装については
> Promise.raceとdelayによるXHRのキャンセル にて詳しく解説しています。

ここで少し`then`の挙動について思い出してみましょう。

`then`に登録するコールバック関数では値を`return`することができます。
この時returnした値は次のpromiseオブジェクト、つまり次の`then`や`catch`のコールバックに渡されます。

また、returnするものはプリミティブな値に限らずオブジェクト、そしてpromiseオブジェクトも返す事が出来ます。

以下のように書いた場合、`then`
に登録するコールバック関数で返すpromiseオブジェクトの状態によって、
次の`then`に登録されたonFulfilled、onRejectedどちらが呼ばれるかを決めることが出来ます。

```
"use strict";
var promise = Promise.resolve();
promise.then(function () {
    var retPromise = new Promise(function (resolve, reject) {
        // resolve or reject で onFulfilled or onRejected どちらを呼ぶか決まる
    });
    return retPromise;<i class="conum" data-value="1"></i><b>(1)</b>
}).then(onFulfilled, onRejected);
```

つまり、この`retPromise`がRejectedになった場合は、`onRejected`が呼び出されるので、
`throw`を使わなくても`then`の中でrejectすることが出来ます。

```
"use strict";
var onRejected = console.error.bind(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(onRejected);
```

これは、Promise.reject を使うことでもっと簡潔に書くことが出来ます。

```
"use strict";
var onRejected = console.error.bind(console);
var promise = Promise.resolve();
promise.then(function () {
    return Promise.reject(new Error("this promise is rejected"));
}).catch(onRejected);
```

4.3.3. まとめ

このセクションでは、以下のことについて学びました。

- `throw`ではなくて`reject`した方が安全
- `then`の中でもrejectする方法

中々使いどころが多くはないかもしれませんが、安易に`throw`してしまうよりはいい事が多いので、覚えておくといいでしょう。

これを利用した具体的な例としては、Promise.raceとdelayによるXHRのキャンセルで解説しています。

4.4. DeferredとPromise

このセクションではDeferredとPromiseの関係について簡潔に学んでいきます。

4.4.1. Deferredとは何か

Deferredという単語はPromiseと同じコンテキストで聞いた事があるかもしれません。　有名な所だとjQuery.Deferred や JSDeferred 等があげられるでしょう。

DeferredはPromiseと違い、共通の仕様があるわけではなく、各ライブラリがそのような目的の実装をそう呼んでいます。

今回は jQuery.Deferred を中心にして話を進めます。

4.4.2. DeferredとPromiseの関係

DeferredとPromiseの関係を簡単に書くと以下のようになります。

- Deferred は Promiseを持っている

- Deferred は Promiseの状態を操作する特権的なメソッドを持っている

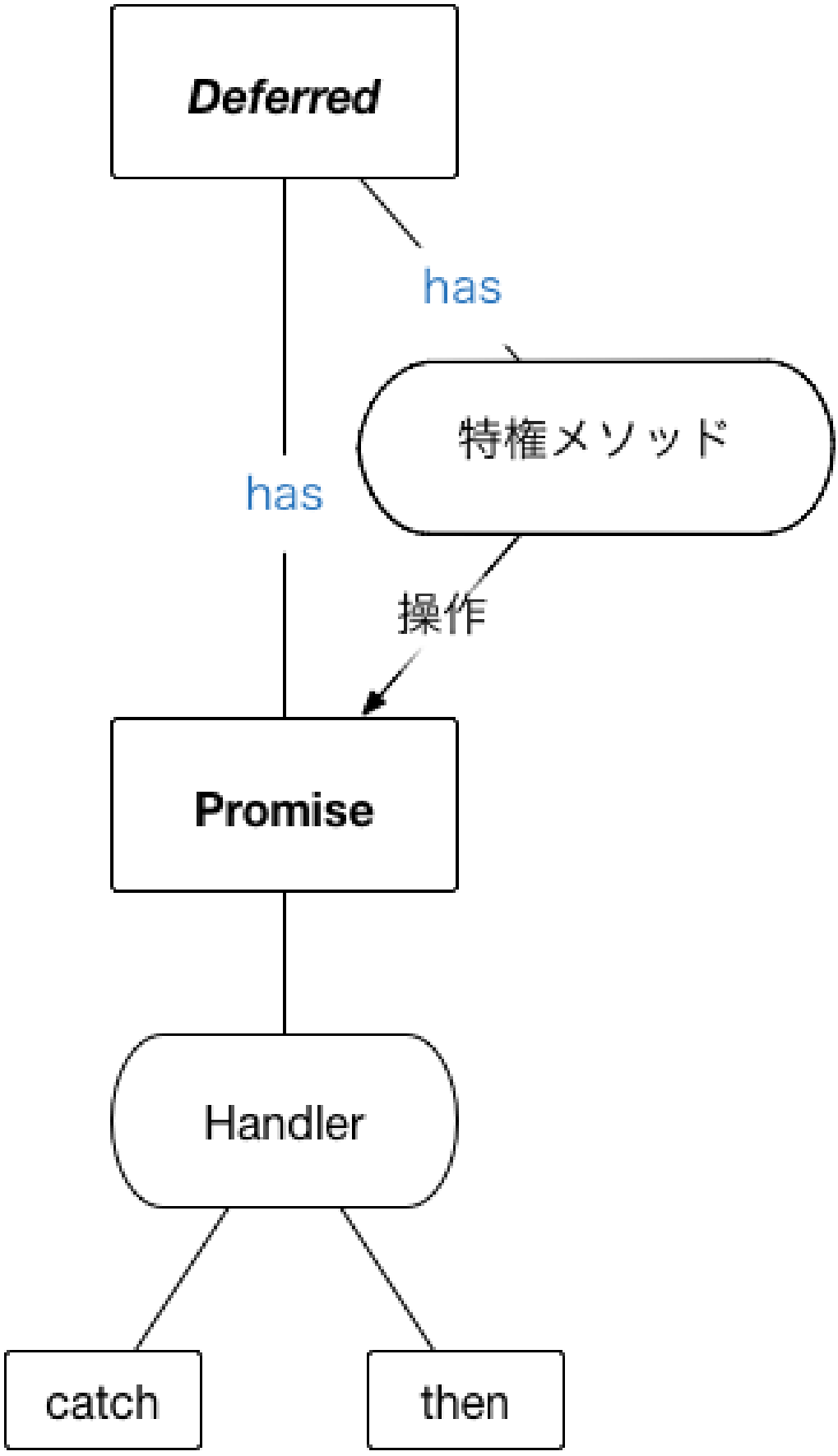


Figure 12. DeferredとPromise

この関係を見れば分かると思いますが、DeferredとPromiseは比べるような関係ではなく、DeferredがPromiseの上に成り立っていて、DeferredがPromiseを操作するメソッドを持っています。

NOTE

jQuery.Deferredの構造を簡略化したものです。もちろんPromiseを持たないDeferredの実装もあります。

図だけだと分かりにくいので、実際にPromiseを使ってDeferredを実装してみましょう。

4.4.3. Deferred top on Promise

Promiseの上にDeferredを実装した例です。

Listing 26. deferred.js

```
'use strict';
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);
};
```

以前Promiseを使って実装した`getURL`をこのDeferredで実装しなおしてみます。

Listing 27. xhr-deferred.js

```
'use strict';
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);
};
function getURL(URL) {
    var deferred = new Deferred();
    var req = new XMLHttpRequest();
    req.open('GET', URL, true);
    req.onload = function () {
        if (req.status == 200) {
            deferred.resolve(req.response);
        } else {
            deferred.reject(new Error(req.statusText));
        }
    };
    req.onerror = function () {
        deferred.reject(new Error(req.statusText));
    };
    req.send();
    return deferred.promise;
}
```

Promiseの状態を操作する特権的なメソッドというのは、
promiseオブジェクトの状態をresolve、rejectすることができるメソッドで、
通常のPromiseだとコンストラクタで渡した関数の中でしか操作する事が出来ません。

Promiseで実装したものと見比べていきたいと思います。

Listing 28. xhr-promise.js

```
'use strict';
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.response);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
```

2つの`getURL`を見比べて見ると以下のような違いがある事が分かります。

- Deferred の場合は全体がPromiseで囲まれていない
  - 関数で囲んでないため、1段ネストが減っている
  - 逆にPromiseでのエラーハンドリングは行われていない

逆に以下の部分は同じ事やっています。

- 全体的な処理の流れ
  - `resolve`、`reject`を呼ぶタイミング
- 関数はpromiseオブジェクトを返す

このDeferredはPromiseを持っているため、大きな流れは同じですが、
Deferredには特権的なメソッドを持っていることや自分で流れを制御する裁量が大きいことが分かります。

例えば、Promiseの場合はコンストラクタの中に処理を書くことが通例なので、
`resolve`、`reject`を呼ぶタイミングが大体みて分かります。

```
new Promise(function (resolve, reject){
    // この中に解決する処理を書く
}
```

一方Deferredの場合は、関数的なまとまりはないのでdeferredオブジェクトを作ったところから、
任意のタイミングで`resolve`、`reject`を呼ぶ感じになります。

```
var deferred = new Deferred();

// どこかのタイミングで  deferred.resolve or deferred.rejectを呼ぶ
```

このように小さなDeferredの実装ですがPromiseとの違いが出ていることが分かります。

これは、Promiseが値を抽象化したオブジェクトなのに対して、
Deferredはまだ処理が終わってないという状態や操作を抽象化したオブジェクトである違いがでているのかもしれません。

言い換えると、
Promiseはこの値は将来的に正常な値(onFulfilled)か異常な値(onRejected)が入るというものを予約したオブジェクトなのに対して、

Deferredはまだ処理が終わってないという事を表すオブジェクトで、
処理が終わった時の結果を取得する機構(Promise)に加えて処理を進める機構をもったものといえるかもしれません。

より詳しくDeferredについて知りたい人は、jQuery.DeferredやDeferredの元となったTwisted、また以下を参照するといいでしょう。

- Promise & Deferred objects in JavaScript Pt.1: Theory and Semantics.
- Twisted 入門 — Twisted Intro
- Promise anti patterns · petkaantonov/bluebird Wiki
- Coming from jQuery · kriskowal/q Wiki

NOTE

DeferredはPythonの Twisted というフレームワークが最初に定義した概念です。
JavaScriptへは MochiKit.Async 、 dojo/Deferred 等のライブラリがその概念を持ってきたと言われています。

4.5. Promise.raceとdelayによるXHRのキャンセル

このセクションでは2章で紹介したPromise.raceのユースケースとして、
Promise.raceを使ったタイムアウトの実装を学んでいきます。

もちろんXHRは timeout プロパティを持っているので、
これを利用すると簡単に出来ますが、複数のXHRを束ねたタイムアウトや他の機能でも応用が効くため、 分かりやすい非同期処理であるXHRにおけるタイムアウトによるキャンセルを例にしています。

4.5.1. Promiseで一定時間待つ

まずはタイムアウトをPromiseでどう実現するかを見て行きたいと思います。

タイムアウトというのは一定時間経ったら何かするという処理なので、`setTimtout`を使えばいいことが分かりますね。

まずは単純に`setTimeout`をPromiseでラップした関数を作ってみましょう。

Listing 29. delayPromise.js

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

`delayPromise(ms)` は引数で指定したミリ秒後にonFulfilledを呼ぶpromiseオブジェクトを返すので、通常の`setTimeout`を直接使ったものと比較すると以下のように書けるだけの違いです。

```
setTimeout(function () {
    alert("100ms 経ったよ !");
}, 100);
// == ほぼ同様の動作
delayPromise(100).then(function () {
    alert("100ms 経ったよ !");
});
```

ここではpromiseオブジェクトであるという事が重要になってくるので覚えておいて下さい。

4.5.2. Promise.raceでタイムアウト

`Promise.race`について簡単に振り返ると、
以下のようにどれか一つでもpromiseオブジェクトが解決状態になったら次の処理を実行する静的メソッドでした。

```
'use strict';
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);
    });
// 一番最初のものが resolveされた時点で終了
Promise.race([winnerPromise, loserPromise]).then(function (value) {
    console.log(value);    // => 'this is winner'
});
```

先ほどのdelayPromiseと別のpromiseオブジェクトを、
`Promise.race`によって競争させることで簡単にタイムアウトが実装出来ます。

Listing 30. simple-timeout-promise.js

```
'use strict';
function delayPromise(ms) {
    return new Promise(function (resolve) {
        setTimeout(resolve, ms);
    });
}
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(比較対象の promise, ms)` はタイムアウト処理を入れたいpromiseオブジェクトとタイムアウトの時間を受け取り、`Promise.race`により競争させたpromiseオブジェクトを返します。

`timeoutPromise` を使うことで以下のようにタイムアウト処理を書くことが出来るようになります。

```
var taskPromise = new Promise(function(resolve){
    // 何らかの処理
    var result = "...";
    resolve(result);
});
timeoutPromise(taskPromise, 1000).then(function(value){
    // taskPromiseが時間内に終わった
}).catch(function(error){
    // タイムアウトになってしまった
});
```

タイムアウトになった場合はエラーが呼ばれるように出来ましたが、このままでは通常のエラーとタイムアウトのエラーの区別がつかなくなってしまいます。

この`Error`オブジェクトの区別をしやすくするため、
`Error`オブジェクトのサブクラスとして`TimeoutError`を定義したいと思います。

4.5.3. カスタムErrorオブジェクト

`Error` オブジェクトはECMAScriptのビルトインオブジェクトです。

ECMAScript5では完璧に`Error`を継承したものを作る事は不可能ですが(スタックトレース周り等)、今回は通常のErrorとは区別を付けたいという目的なので、それを満たせる`TimeoutError`オブジェクトを作成出来ます。

NOTE

ECMAScript6では`class`構文を使うことで内部的にも正確に継承を行うことが出来ます。

```
class MyError extends Error{
    // Error□□□□□□□□□□□
}
```

`error instanceof TimeoutError` というように利用できる`TimeoutError`を定義すると以下のようになります。

Listing 31. TimeoutError.js

```
'use strict';
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`というコンストラクタ関数を定義して、このコンストラクタにErrorをprototype継承させています。

使い方は通常の`Error`オブジェクトと同じで以下のように`throw`するなどして利用できます。

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

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

この`TimeoutError`を使えば、タイムアウトによるErrorオブジェクトなのか、他の原因のErrorオブジェクトなのかが容易に判定できるようになります。

NOTE

今回紹介したビルトインオブジェクトを継承したオブジェクトの作成方法については Chapter 28. Subclassing Built-ins で詳しく紹介されています。 また、 Error - JavaScript | MDN にもErrorオブジェクトについて書かれています。

4.5.4. タイムアウトによるXHRのキャンセル

ここまでくれば、どのようにPromiseを使ったXHRのキャンセルを実装するか見えてくるかもしれません。

XHRのキャンセル自体は`XMLHttpRequest`オブジェクトの `abort()` メソッドを呼ぶだけなので難しくないですね。

`abort()` メソッドを外から呼べるようにするために、今までのセクションにもでてきた `getURL`を少し拡張して、
XHRを包んだpromiseオブジェクトと共にそのXHRを中止するメソッドを持つオブジェクトを返すようにしています。

Listing 32. delay-race-cancel.js

```
'use strict';
function cancelableXHR(URL) {
    var req = new XMLHttpRequest();
    var promise = new Promise(function (resolve, reject) {
            req.open('GET', URL, true);
            req.onload = function () {
                if (req.status == 200) {
                    resolve(req.response);
                } 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 () {
        // □□request□□□□□□□□□abort□□
        //
https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest
        if (req.readyState !== XMLHttpRequest.UNSENT) {
            req.abort();
        }
    };
    return {
        promise: promise,
        abort: abort
    };
}
```

これで必要な要素は揃ったので後は、Promiseを使った処理のフローに並べていくだけです。

大まかな流れとしては以下のようになります。

1. `cancelableXHR`を使いXHRのpromiseオブジェクトと中止を呼び出すメソッドを取得する
2. `timeoutPromise`を使いXHRのpromiseとタイムアウト用のpromiseを`Promise.race`で競争させる
    - XHRが時間内に取得出来た場合
        a. 通常のpromiseと同様に`then`で中身を取得する
    - タイムアウトとなった場合は
        a. `throw TimeoutError`されるので`catch`する
        b. catchしたエラーオブジェクトが`TimeoutError`のものだったら`abort`を呼び出してXHRをキャンセルする

これらの要素を全てまとめると次のように書けます。

Listing 33. delay-race-cancel-play.js

```
'use strict';
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;
function delayPromise(ms) {
    return new Promise(function (resolve) {
        setTimeout(resolve, ms);
    });
}
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]);
}
function cancelableXHR(URL) {
    var req = new XMLHttpRequest();
    var promise = new Promise(function (resolve, reject) {
            req.open('GET', URL, true);
            req.onload = function () {
                if (req.status == 200) {
                    resolve(req.response);
                } 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 () {
        // 既にrequestが止まってなければ  abortする
        //
https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest
        if (req.readyState !== XMLHttpRequest.UNSENT) {
            req.abort();
        }
    };
    return {
        promise: promise,
        abort: abort
    };
}
var object = cancelableXHR('http://httpbin.org/get');
// main
timeoutPromise(object.promise, 1000).then(function (contents) {
    console.log('Contents', contents);
}).catch(function (error) {
    if (error instanceof TimeoutError) {
        object.abort();
        return console.log(error);
    }
    console.log('XHR Error :', error);
});
```

これで、一定時間後に解決されるpromiseオブジェクトを使ったタイムアウト処理が実現できました。

NOTE

通常の開発の場合は繰り返し使えるように、それぞれファイルに分割して定義しておくといいですね。

4.5.5. promiseと操作メソッド

先ほどの`cancelableXHR`はpromiseオブジェクトと操作のメソッドが
一緒になったオブジェクトを返すようにしていたため少し分かりくかったかもしれません。

一つの関数は一つの値(promiseオブジェクト)を返すほうが見通しがいいと思いますが、
`cancelableXHR`の中で生成した`req`は外から参照できないので、特定のメソッド(先ほどのケースは`abort`)からは触れるようにする必要があります。

返すpromiseオブジェクト自体を拡張して`abort`出来るようにするという手段もあると思いますが、

promiseオブジェクトは値を抽象化したオブジェクトであるため、何でも操作用のメソッドをつけていくと複雑になってしまうかもしれません。

一つの関数で全てやろうとしてるのがそもそも良くないので、
ひとつの関数で何でもやるのは止めて、以下のように関数に分離していくというのが妥当な気がします。

- XHRを行うpromiseオブジェクトを返す
- promiseオブジェクトを渡したら該当するXHRを止める

これらの処理をまとめたモジュールを作れば今後の拡張がしやすいですし、
一つの関数がやることも小さくて済むので見通しも良くなると思います。

モジュールの作り方は色々作法(AMD,CommonJS,ES6 module etc..)があるのでここでは、先ほどの`cancelableXHR`をNode.jsのモジュールとして作りなおしてみます。

Listing 34. cancelableXHR.js

```
"use strict";
var requestMap = {};
function createXHRPromise(URL) {
    var req = new XMLHttpRequest();
    var 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.response);
            } 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) {
```

```
    if (typeof promise === "undefined") {
        return;
    }
    var request;
    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;
```

使い方もシンプルに`createXHRPromise`でXHRのpromiseオブジェクトを作成して、
そのXHRを`abort`したい場合は`abortPromise(promise)`にpromiseオブジェクトを渡すという感じで利用できるようになります。

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

var xhrPromise = cancelableXHR.createXHRPromise('http://httpbin.org/get');<i
class="conum" data-value="1"></i><b>(1)</b>
xhrPromise.catch(function (error) {
    // abort □□□□□□□□□□□
});
cancelableXHR.abortPromise(xhrPromise);<i class="conum" data-
value="2"></i><b>(2)</b>
```

4.5.6. まとめ

ここでは以下の事について学びました。

- 一定時間後に解決されるdelayPromise
- delayPromiseとPromise.raceを使ったタイムアウトの実装
- XHRのpromiseのリクエストのキャンセル
- モジュール化によるpromiseオブジェクトと操作の分離

Promiseは処理のフローを制御する力に優れているため、
それを最大限活かすためには一つの関数でやり過ぎないで処理を小さく分けること等、
今までのJavaScriptで言われているような事をより意識していいのかもしれません。

4.6. Promise.prototype.done とは何か?

既存のPromise実装ライブラリを利用したことがある人は、`then`の代わりに使う`done`というメソッドを見たことがあるかもしれません。

それらのライブラリでは `Promise.prototype.done` というような実装が存在し、
使い方は`then`と同じですが、promiseオブジェクトを返さないようになっています。

`Promise.prototype.done` は、ES6 PromisesやPromises/A+の仕様には
存在していない記述ですが、多くのライブラリが実装しています。

このセクションでは、`Promise.prototype.done`とは何か?
また何故このようなメソッドが多くのライブラリで実装されているかについて学んでいきましょう。

4.6.1. doneを使ったコード例

実際にdoneを使ったコードを見てみると`done`の挙動が分かりやすいと思います。

Listing 35. promise-done-example.js

```
'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);
        });
    };
}
var promise = Promise.resolve();
promise.done(function () {
    JSON.parse('this is not json');    // => SyntaxError: JSON.parse: unexpected
keyword at line 1 column 1 of the JSON data
});
```

最初に述べたように、`Promise.prototype.done`は仕様としては存在しないため、
利用する際は実装されているライブラリを使うか自分で実装する必要があります。

実装については後で解説しますが、まずは`then`を使った場合と`done`を使ったものを比較してみます
。

Listing 36. thenを使った場合

```
var promise = Promise.resolve();
promise.then(function () {
    JSON.parse("this is not json");
}).catch(function (error) {
    console.error(error);// => "SyntaxError: JSON.parse: unexpected keyword at
line 1 column 1 of the JSON data"
});
```

比べて見ると以下のような違いがあることが分かります。

- `done` はpromiseオブジェクトを返さない
  - つまり、doneの後に`catch`等のメソッドチェーンはできない
- `done` の中で発生したエラーはそのまま外に例外として投げられる

- つまり、Promiseによるエラーハンドリングが行われない

done はpromiseオブジェクトを返していないので、Promise chainの最後になるメソッドというのはわかると思います。

また、Promiseには強力なエラーハンドリング機能があると紹介していましたが、done の中ではそのエラーハンドリングをワザと突き抜けて例外を出すようになっています。

何故このようなPromiseの機能とは相反するメソッドが、多くのライブラリで実装されいるかについては 次のようなPromiseの失敗例を見ていくと分かるかもしれません。

4.6.2. 沈黙したエラー

Promiseには強力なエラーハンドリング機能がありますが、（デバッグツールが上手く働かない場合に）この機能がヒューマンエラーをより複雑なものにしてしまう一面があります。

これは、then or catch?でも同様の内容が出てきたことを覚えているかもしれません。

次のような、promiseオブジェクトを返す関数を考えてみましょう。

Listing 37. json-promise.js

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

渡された値を`JSON.parse`してpromiseオブジェクトを返す関数ですね。

以下のように使うことができ、`JSON.parse`はパースに失敗すると例外を投げるので、
それを`catch`することが出来ます。

```
var string = "json□□□□□□□";
JSONPromise(string).then(function (object) {
    console.log(object);
}).catch(function(error){
    // => JSON.parse□□□□□□□□□
});
```

ちゃんと`catch`していれば何も問題がないのですが、その処理を忘れてしまうというミスをした時にどこでエラーが発生してるのかわからなくなるというヒューマンエラーを助長させる面があります。

Listing 38. catchによるエラーハンドリングを忘れてしまった場合

```
var string = "json□□□□□□□";
JSONPromise(string).then(function (object) {
    console.log(object);
}); <i class="conum" data-value="1"></i><b>(1)</b>
```

`JSON.parse`のような分かりやすい例の場合はまだ良いですが、
メソッドをtypoしたことによるSyntax Errorなどはより深刻な問題となりやすいです。

Listing 39. typoによるエラー

```
var string = "{}";
JSONPromise(string).then(function (object) {
    conosle.log(object);<i class="conum" data-value="1"></i><b>(1)</b>
});
```

この場合は、`console`を`conosle`とtypoしているため、以下のようなエラーが発生するはずです。

> ReferenceError: conosle is not defined

しかし、Promiseではtry-catchされるため、エラーが握りつぶされてしまうという現象が発生しやすくなります。
毎回、正しく`catch`の処理を書くことが出来る場合は何も問題ありませんが、
Promiseの実装によってはこのようなミスが検知しにくくなるケースがあることを知っておくべきでしょう。

このようなエラーの握りつぶしはunhandled rejectionと言われることがあります。
Rejectedされた時の処理がないというそのままの意味ですね。

NOTE

このunhandled rejectionが検知しにくい問題はPromiseの実装に依存します。
例えば、ypromiseはunhandled rejectionがある場合は、その事をコンソールに表示します。

> Promise rejected but no error handlers were registered to it

また、Bluebirdの場合も、明らかに人間のミスにみえるReferenceErrorの場合などはそのままコンソールにエラーを表示してくれます。

> "Possibly unhandled ReferenceError. conosle is not defined

ネイティブのPromiseの場合も同様にこの問題への対処としてGC-based unhandled rejection trackingというものが 搭載されつつあります。
これはpromiseオブジェクトがガーベッジコレクションによって回収されるときに、それがunhandled rejectionであるなら、エラー表示をするという仕組みがベースとなっているようです。
Firefox や Chrome のネイティブPromiseでは一部実装されています。

4.6.3. doneの実装

Promiseにおける `done` は先程のエラーの握りつぶしを避けるにはどうするかという方法論として、そもそもエラーハンドリングをしなければいい という豪快な解決方法を提供するメソッドです。

`done`はPromiseの上に実装することが出来るので、
`Promise.prototype.done`というPromiseのprototype拡張として実装してみましょう。

Listing 40. promise-prototype-done.js

```
"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の中でthrowをすることで、外へそのまま例外を投げることを利用しています。

Listing 41. setTimeoutのコールバック内ではの例外

```
try{
    setTimeout(function callback() {
        throw new Error("error");<i class="conum" data-value="1"></i><b>(1)</b>
    }, 0);
}catch(error){
    console.error(error);
}
```

NOTE

なぜ非同期の`callback`内での例外をキャッチ出来ないのかは以下が参考なります。

- JavaScriptと非同期のエラー処理 - Yahoo! JAPAN Tech Blog

`Promise.prototype.done` をよく見てみると、何も`return`していないこともわかると思います。
つまり、`done`は「ここでPromise
chainは終了して、例外が起きた場合はそのままpromiseの外へ投げ直す」という処理になっています。
実装や環境がしっかり対応していれば、unhandled
rejectionの検知はできるため、必ずしも`done`が必要というわけではなく、
また今回の`Promise.prototype.done`のように、`done`は既存のPromiseの上に実装することができため、 ES6 Promisesの仕様そのものには入らなかったと言えるかもしれません。

今回の`Promise.prototype.done`の実装は promisejs.org を参考にしています。

4.6.4. まとめ

このセクションでは、Q や Bluebird や prfun 等多くのPromiseライブラリで実装されている`done`の基礎的な実装と、`then`とはどのような違いがあるかについて学びました。

`done`には2つの側面があることがわかりました。

- `done`の中で起きたエラーは外へ例外として投げ直す
- Promise chain を終了するという宣言

then or catch? と同様にPromiseにより沈黙してしまったエラーについては、デバッグツールやライブラリの改善等で殆どのケースでは問題ではなくなるかもしれません。

また、`done`は値を返さない事でそれ以上Promise chainを繋げる事ができなくなるため、そのような統一感を持たせるという用途で`done`を使うことも出来ます。

ES6 Promises では根本に用意されてる機能はあまり多くありません。
そのため、自ら拡張したり、拡張したライブラリ等を利用するケースが多いと思います。

その時でも何でもやり過ぎると、せっかく非同期処理をPromiseでまとめても複雑化してしまう場合があるため、
統一感を持たせるというのは抽象的なオブジェクトであるPromiseにおいては大事な部分と言えるかもしれません。

4.7. Promiseとメソッドチェーン

Promiseは`then`や`catch`等のメソッドを繋げて書いていきます。
これはDOMやjQuery等でよくみられるメソッドチェーンとよく似ています。

一般的なメソッドチェーンは`this`を返すことで、メソッドを繋げて書けるようになっています。

NOTE	メソッドチェーンの作り方については メソッドチェーンの作り方 - あと味 などを参照するといいでしょう。

一方、Promiseは毎回新しいpromiseオブジェクトを返すようになっていますが、
一般的なメソッドチェーンと見た目は全く同じです。

このセクションでは、一般的なメソッドチェーンで書かれたものを
インターフェースはそのままで内部的にはPromiseで処理されるようにする方法について学んでいきたいと思います。

4.7.1. fsのメソッドチェーン

以下のようなNode.jsの fsモジュールを例にしてみたいと思います。

また、今回の例は見た目のわかりやすさを重視しているため、
現実的にはあまり有用なケースとは言えないかもしれません。

Listing 42. fs-method-chain.js

```
"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;
```

このモジュールは以下のようにread → transform → writeという流れをメソッドチェーンで表現することができます。

```
var File = require("./fs-method-chain");
var inputFilePath = "input.txt",
    outputFilePath = "output.txt";
File.read(inputFilePath)
    .transform(function (content) {
        return ">>" + content;
    })
    .write(outputFilePath);
```

`transform` は引数で受け取った値を変更する関数を渡して処理するメソッドです。
この場合は、readで読み込んだ内容の先頭に`>>`という文字列を追加しているだけです。

4.7.2. Promiseによるfsのメソッドチェーン

次に先ほどのメソッドチェーンをインターフェースはそのまま維持して
内部的にPromiseを使った処理にしてみたいと思います。

Listing 43. fs-promise-chain.js

```
"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;
```

内部に持ってるpromiseオブジェクトに対するエイリアスとして
`then`と`catch`を持たせていますが、それ以外のインターフェースは全く同じで使い方となっています
。
そのため、先ほどのコードで`require`するモジュールを変更しただけで動作します。

```
var File = require("./fs-promise-chain");
var inputFilePath = "input.txt",
    outputFilePath = "output.txt";
File.read(inputFilePath)
    .transform(function (content) {
        return ">>" + content;
    })
    .write(outputFilePath);
```

`File.prototype.then` というメソッドは、`this.promise.then` が返す新しいpromiseオブジェクトを`this.promise`に対して代入しています。

これはどういうことなのかというと、以下のように擬似的に展開してみると分かりやすいでしょう。

```
var File = require("./fs-promise-chain");
File.read(inputFilePath)
    .transform(function (content) {
        return ">>" + content;
    })
    .write(outputFilePath);
// => ▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫▫
promise.then(function read(){
        return fs.readFileSync(filePath, "utf-8");
    }).then(function transform(content) {
         return ">>" + content;
    }).then(function write(){
        return fs.writeFileSync(filePath, data);
    });
```

`promise = promise.then(...)` という書き方は一見すると、上書きしているようにみえるため、それまでのpromiseのchainが途切れてしまうと思うかもしれません。

イメージとしては `promise = addPromiseChain(promise, fn);` のような感じになっていて、既存のpromiseオブジェクトに対して新たな処理を追加したpromiseオブジェクトを作って返すため、自分で逐次的処理する機構を実装しなくても問題ないわけです。

4.7.3. 両者の違い

同期と非同期

fs-method-chain.jsとPromise版の違いを見ていくと、
そもそも両者には同期的、非同期的という大きな違いがあります。

fs-method-chain.js のようなメソッドチェーンでもキュー等の処理を実装すれば、非同期的なほぼ同様のメソッドチェーンを実装出来ますが、複雑になるため今回は単純な同期的なメソッドチェーンにしました。

Promise版はコラム: Promiseは常に非同期?で紹介したように常に非同期処理となるため、promiseを使ったメソッドチェーンも非同期となっています。

エラーハンドリング

fs-method-chain.jsにはエラーハンドリングの処理は入っていないですが、
同期処理であるため全体を`try-catch`で囲む事で行えます。

Promise版 では内部で利用するpromiseオブジェクトの`then`と`catch`へのエイリアスを用意してあるため、通常のpromiseと同じように`catch`によってエラーハンドリングが行えます。

Listing 44. fs-promise-chainでのエラーハンドリング

```
var File = require("./fs-promise-chain");
File.read(inputFilePath)
    .transform(function (content) {
        return ">>" + content;
    })
    .write(outputFilePath)
    .catch(function(error){
        console.error(error);
    });
```

fs-method-chain.jsに非同期処理が加えたものを自力で実装する場合、
エラーハンドリングが大きな問題となるため、非同期処理にしたい時は
Promiseを使うと比較的簡単に実装できると言えるかもしれません。

4.7.4. Promise以外での非同期処理

このメソッドチェーンと非同期処理を見てNode.jsに慣れている方は Stream
が思い浮かぶと思います。

Stream を使うと、
`this.lastValue`のような値を保持する必要がなくなる事や大きなファイルの扱いが改善したり、
上記の例に比べるとより高速に処理できる可能性が高いと思います。

Listing 45. streamによるread→transform→write

```
readableStream.pipe(transformStream).pipe(writableStream);
```

そのため、非同期処理には常にPromiseが最適という訳ではなく、
目的と状況にあった実装をしていくことを考えていくべきでしょう。

> Node.jsのStreamはEventをベースにしている技術

Node.jsのStreamについて詳しくは以下を参照して下さい。

- Node.js の Stream API で「データの流れ」を扱う方法 - Block Rockin' Codes
- Stream2の基本
- Node-v0.12の新機能について

4.7.5. Promiseラッパー

話を戻してfs-method-chain.jsとPromise版の両者を比べると、
内部的にもかなり似ていて、同期版のものがそのまま非同期版でも使えるような気がします。

JavaScriptでは動的にメソッドを定義することもできるため、
自動的にPromise版を生成できないかということを考えると思います。
(もちろん静的に定義する方が扱いやすいですが)

そのような仕組みはES6 Promisesにはありませんが、 著名なサードパーティのPromise実装である
bluebird などには Promisification という機能が用意されています。

これを利用すると以下のように、その場でpromise版のメソッドを追加して利用できるようになります。

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

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

ArrayのPromiseラッパー

先ほどの Promisification が何をやっているのか少しイメージしにくいので、次のようなネイティブ`Array`のPromise版を使えるメソッドを動的に定義する例を考えてみましょう。

JavaScriptにはネイティブにもDOMやString等メソッドチェーンが行える機能が多くあります。`Array`もその一つで`map`や`filter`等の高階関数を受け取って処理はメソッドチェーンが利用しやすい機能です。

Listing 46. array-promise-chain.js

```
"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) {
    // Don't overwrite
    if (typeof ArrayAsPromise[methodName] !== "undefined") {
        return;
    }
    var arrayMethod = Array.prototype[methodName];
    if (typeof  arrayMethod !== "function") {
        return;
    }
    ArrayAsPromise.prototype[methodName] = function () {
        var that = this;
        var 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);
};
```

ネイティブのArrayと`ArrayAsPromise`を使った場合の違いは
上記のコードのテストを見てみるのが分かりやすいでしょう。

Listing 47. array-promise-chain-test.js

```
"use strict";
var assert = require("power-assert");
var ArrayAsPromise = require("../src/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);
        });
    });
});
```

`ArrayAsPromise` でもArrayのメソッドを利用できているのが分かります。
先ほどと同じように、ネイティブのArrayは同期処理で、`ArrayAsPromise`
は非同期処理という違いがあります。

`ArrayAsPromise`
の実装見て気づくと思いますが、`Array.prototype`のメソッドを全て実装しています。
しかし、`array.indexOf`など`Array.prototype`には配列を返さないものもあるため、全てをメソッドチェーンにするのは不自然なケースがあると思います。

ここで大事なのが、同じ値を受けるインターフェースを持っているAPIはこのような手段でPromise版のAPIを自動的に作成できるという点です。
このようなAPIの規則性を意識してみるとまた違った使い方が見つかるかもしれません。

NOTE

先ほどの Promisification は Node.jsのCoreモジュールの非同期処理には
`function(error,result){}`
というように第一引数に`error`が来るというルールを利用して、
自動的にPromiseでラップしたメソッドを生成しています

4.7.6. まとめ

このセクションでは以下のことについて学びました。

- Promise版のメソッドチェーンの実装
- Promiseが常に非同期の最善の手段ではない
- Promisification
- 統一的なインターフェースの再利用

ES6 PromisesはCoreとなる機能しか用意されていません。
そのため、自分でPromiseを使った既存の機能のラッパー的な実装をする事があるかもしれません。
しかし、何度もコールバックを呼ぶEventのような処理がPromiseには不向きなように、
Promiseが常に最適な非同期処理という訳ではありません。
その機能にPromiseを使うのが最適なのかを考える事はこの書籍の目的でもあるため、
何でもPromiseにするというわけではなく、その目的にPromiseが合うのかどうかを考えてみるのもいいと思います。

5. Promises API Reference

5.1. Promise#then

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

Listing 48. thenコード例

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

promiseオブジェクトに対してonFulfilledとonRejectedのハンドラを定義し、
新たなpromiseオブジェクトを作成して返す。
このハンドラはpromiseがresolve または rejectされた時にそれぞれ呼ばれる。

- 定義されたハンドラ内で返した値は、新たなpromiseオブジェクトのonFulfilledに対して渡される。
- 定義されたハンドラ内で例外が発生した場合は、新たなpromiseオブジェクトのonRejectedに対して渡される。

5.2. Promise#catch

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

Listing 49. catchのコード例

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

`promise.then(undefined, onRejected)` と同等の意味を持つシンタックスシュガー。

5.3. Promise.resolve

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

Listing 50. Promise.resolveのコード例

```
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;
    });
}
```

受け取った値に応じたpromiseオブジェクトを返す。

どの場合でもpromiseオブジェクトを返すが、大きく分けて以下の3種類となる。

promiseオブジェクトを受け取った場合
　受け取ったpromiseオブジェクトをそのまま返す

thenableなオブジェクトを受け取った場合
　`then`を持つオブジェクトを新たなpromiseオブジェクトにして返す

その他の値(オブジェクトやnull等も含む)を受け取った場合
　その値でresolveされる新たなpromiseオブジェクトを作り返す

5.4. Promise.reject

```
Promise.rejct(object)
```

Listing 51. Promise.rejectのコード例

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

受け取った値でrejectされた新たなpromiseオブジェクトを返す。

Promise.rejctに渡す値は`Error`オブジェクトとすべきである。

また、Promise.resolveとは異なり、promiseオブジェクトを渡した場合も常に新たなpromiseオブジェクトを作成する。

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

5.5. Promise.all

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

Listing 52. Promise.allのコード例

```
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]
});
```

新たなpromiseオブジェクトを作成して返す。

渡されたpromiseオブジェクトの配列が全てresolveされた時に、
新たなpromiseオブジェクトはその値でresolveされる。

どれかの値がrejectされた場合は、その時点で新たなpromiseオブジェクトはrejectされる。

渡された配列の値はそれぞれ`Promise.resolve`にラップされるため、
promiseオブジェクト以外が混在している場合も扱える。

5.6. Promise.race

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

Listing 53. Promise.raceのコード例

```
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
});
```

新たなpromiseオブジェクトを作成して返す。

渡されたpromiseオブジェクトの配列のうち、一番最初にresolve または rejectされたpromiseにより、新たなpromiseオブジェクトはその値でresolve または rejectされる。

6. 用語集

Promises
: プロミスという仕様そのもの

Promise
: プロミスオブジェクト、インスタンス

ES6 Promises
: ECMAScript 6th Edition を明示的に示す場合にprefixとして ES6 をつける

Promises/A+
: Promises/A+の事。ES6 Promisesのベースとなったコミュニティベースの仕様であり、ES6 Promisesとは多くの部分が共通している。

Thenable
: Promiseライクなオブジェクトの事。`.then`というメソッドを持つオブジェクト。

promise chain
: promiseオブジェクトを`then`や`catch`のメソッドチェーンでつなげたもの。
この用語は書籍中のものであり、ES6 Promisesで定められた用語ではありません。