JavaScript Promiseの本

azu

Table of Contents

This book has been released in :

- Chinese: JavaScript Promise迷你#(中文版) [1]
- Korean: ##### eBook JavaScript Promise(###) [2]

These translated is based on Promise Book ver 1.

- JavaScript Promiseの本(v1) [3]

---

[1] http://liubin.github.io/promises-book/
[2] http://www.hanbit.co.kr/store/books/look.php?p_code=E5027975256
[3] https://azu.github.io/promises-book/archives/v1/

はじめに

書籍の目的

この書籍はJavaScript標準仕様のECMAScript Promisesを中心にし、JavaScriptにおけるPromiseについて学ぶことが目的です。

この書籍では、次の3つを目標としています。

- Promiseについて学び、パターンやテストを扱えること
- 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
- ES Promises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こと

この書籍では、先程も述べたようにES Promises、つまりJavaScriptの標準仕様(ECMAScript)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などモダンなブラウザでは、ライブラリを使うこと無く利用できる機能であり、またES Promisesは元がPromises/A+というコミ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から学んでいきます。その中でPromiseの得意/不得意を知り、Promiseを活用したJavaScriptを書け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の基本的な文法や機能をすでに学習している前提です。

次のいずれかの書籍を読んでいれば、十分読み解ける内容だと思います。

- JavaScript: The Good Parts[4]
- JavaScriptパターン[5]
- 初めてのJavaScript 第3版[6]
- JavaScript 第6版[7]
- パーフェクトJavaScript[8]

[4] https://www.oreilly.co.jp/books/9784873113913/
[5] https://www.oreilly.co.jp/books/9784873114880/
[6] https://www.oreilly.co.jp/books/9784873117836/
[7] https://www.oreilly.co.jp/books/9784873115733/
[8] https://gihyo.jp/book/2011/978-4-7741-4813-7

- 改訂新版JavaScript本格入門[9]

この書籍ではECMAScript 2015(ES2015)で追加された構文を利用するため、JavaScriptの基本的な文法や機能に不安がある方は、次の書籍を参照してください。ES2015以降をベースに、JavaScriptの基礎を一から学べる書籍です。この書籍と同一の著者によって書かれており、ウェブでも公開されています。

- JavaScript Primer #jsprimer[10]

または、JavaScriptでウェブアプリケーションを書いた経験や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、どこかで見たことがある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やすいと思います。

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。
  - たとえば、`Promise#then` という表記は、Promiseのインスタンスオブジェクトの `then` 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GitHubから取得できます。

[9] https://gihyo.jp/book/2016/978-4-7741-8411-1
[10] https://jsprimer.net

この書籍は AsciiDoc[11] という形式で書かれています。

- azu/promises-book[12]

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を作成できます。

- Issues · azu/promises-book[13]

また、この書籍についての チャットページ[14] に書いていくのもいいでしょう。

Twitterでのハッシュタグは #Promise本[15] 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[16] も歓迎しています。

Chapter.1 - Promiseとは何か

この章では、JavaScriptにおけるPromiseについて簡単に紹介していきます。

What Is Promise

まずPromiseとはそもそもどのようなものでしょうか？

Promiseは非同期処理を抽象化したオブジェクトとそれを操作する仕組みのことをいいます。詳しくはこれから学んでいくとして、PromiseはJavaScriptで発見された概念ではありません。

最初に発見されたのは E言語[17]におけるもので、並列/並行処理におけるプログラミング言語のデザインの一種です。

[11] http://asciidoctor.org/
[12] https://github.com/azu/promises-book
[13] https://github.com/azu/promises-book/issues?state=open
[14] https://gitter.im/azu/promises-book
[15] https://twitter.com/search?q=%23Promise%E6%9C%AC
[16] https://github.com/azu/promises-book/pulls
[17] http://erights.org/elib/distrib/pipeline.html

このデザインをJavaScriptに持ってきたものが、この書籍で学ぶJavaScript Promiseです。Promiseは、JavaScriptの仕様を決めるECMAScript 2015で導入され動作が定義されています。

一方、JavaScriptにおける非同期処理といえば、コールバックを利用する場合が多いと思います。

コールバックを使った非同期処理の一例

```
getAsync("fileA.txt", (error, result) => { ❶
    if (error) { // 取得失敗時の処理
        throw error;
    }
    // 取得成功の処理
});
```

❶　コールバック関数の引数には(エラーオブジェクト, 結果)が入る

Node.js等JavaScriptでのコールバック関数の第一引数には `Error` オブジェクトを渡すというルールを用いるケースがあります。

このようにコールバックでの非同期処理もルールが統一されていた場合、コールバック関数の書き方が明確になります。しかし、これはあくまでコーディングルールであるため、異なる書き方をしても決して間違いではありません。

Promiseでは、このような非同期に対するオブジェクトとルールを仕様化して、統一的なインターフェースで書くようになっており、それ以外の書き方は出来ないようになっています。

Promiseを使った非同期処理の一例

```
const promise = getAsyncPromise("fileA.txt"); ❶
promise.then((result) => {
    // 取得成功の処理
}).catch((error) => {
    // 取得失敗時の処理
});
```

❶　promiseオブジェクトを返す

非同期処理を抽象化したpromiseオブジェクトというものを用意し、そのpromiseオブジェクトに対して成功時の処理と失敗時の処理の関数を登録するようにして使います。

コールバック関数と比べると何が違うのかを簡単に見ると、非同期処理の書き方がpromiseオブジェクトのインターフェースに沿った書き方に限定されます。

つまり、promiseオブジェクトに用意されているメソッド(ここでは `then` や `catch`)以外は使えないため、コールバックのように引数に何を入れるかが自由に決められるわけではなく、一定のやり方に統一されます。

この、Promiseという統一されたインターフェースがあることで、そのインターフェースにおけるさまざまな非同期処理のパターンを形成することができます。

つまり、複雑な非同期処理等を上手くパターン化できるというのがPromiseの役割であり、Promiseを使う理由の一つであるといえるでしょう。

それでは、実際にJavaScriptでのPromiseについて学んでいきましょう。

Arrow Function ⇒

この書籍のサンプルコードはArrow FunctionなどECMAScript 2015で導入された構文を利用します。

Arrow Functionは、矢印のような `⇒` (イコールと大なり記号)を使い、匿名関数を定義する構文です。関数式と定義方法や使い方は同じです。

```
// 関数式の定義と実行
const fn = function (arg) {
    console.log("通常の関数定義，引数:" + arg)
}
fn("引数");

// Arrow Functionの定義と実行
const arrowFunction = (arg) => {
    console.log("Arrow Functionでの関数定義，引数:" + arg);
}
arrowFunction("引数");
```

通常の関数式と`⇒`という記法を使う以外にも次のような特徴があります。

Arrow Functionには次のような特徴があります。

- 名前を付けることができない(常に匿名関数)
- `this` が静的に決定できる
- `function` キーワードに比べて短く書くことができる
- `new` できない(コンストラクタ関数ではない)

> - `arguments` 変数を参照できない(Rest Parametersを代わりに利用する)
>
> Arrow Functionを使うことで、コールバック関数を関数式に比べて短く簡潔に書くことができます。そのため、この書籍ではArrow Functionをメインに利用しています。
>
> Arrow Functionの詳細は JavaScript Primer[18] を参照してください。
>
> - 関数と宣言 · JavaScript Primer #jsprimer[19]

Promise Overview

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

大きく分けて以下の3種類になります。

Constructor

Promiseは `XMLHttpRequest` のように、コンストラクタ関数である `Promise` からインスタンスとなる promiseオブジェクトを作成して利用します。

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。

```
const promise = new Promise((resolve, reject) => {
    // 非同期の処理
    // 処理が終わったら、resolve または rejectを呼ぶ
});
```

Instance Method

newによって生成されたpromiseオブジェクトにはpromiseの値を resolve(成功) / reject(失敗) した時に呼ばれる コールバック関数を登録するために `promise.then()` というインスタンスメソッドがあります。

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

[18] https://jsprimer.net
[19] https://jsprimer.net/basic/function-declaration/#arrow-function

resolve(成功)した時
　`onFulfilled` が呼ばれる
reject(失敗)した時
　`onRejected` が呼ばれる

`onFulfilled`、`onRejected` どちらもオプショナルな引数となっています。

`promise.then` では成功時と失敗時の処理を同時に登録することができます。また、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(onRejected)` を使うことができます。

```
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。

Promise workflow

以下のようなサンプルコードを見てみましょう。

promise-workflow.js

```
function asyncFunction() {
    ❶
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Async Hello world");
        }, 16);
    });
}
❷
asyncFunction().then((value) => {
    console.log(value); // => 'Async Hello world'
}).catch((error) => {
    console.error(error);
});
```

❶　Promiseコンストラクタを `new` して、promiseオブジェクトを返します
❷　<1>のpromiseオブジェクトに対して `.then` で値が返ってきた時のコールバックを設定します

`asyncFunction` という関数 は promiseオブジェクトを返していて、そのpromiseオブジェクトに対して `then` でresolveし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。

この場合 `catch` のコールバックは呼ばれることはないですが、`setTimeout` が存在しない環境などでは、例外が発生し `catch` で登録したコールバック関数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` を使い、以下のように2つのコールバック関数を設定することでもほぼ同様の動作になります。

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

Promiseの状態

Promiseの処理の流れが少しわかった所で、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。

Fulfilled
: resolve(成功)した時。このとき `onFulfilled` が呼ばれる

Rejected
: reject(失敗)した時。このとき `onRejected` が呼ばれる

Pending
: FulfilledまたはRejected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

これらの状態はES Promisesの仕様で定められている名前です。この状態をプログラムで直接触る方法は用意されていないため、書く際には余り気にしなくても問題ないですが、Promiseについて理解するのに役に立ちます。

この書籍では、Pending、Fulfilled 、Rejected の状態を用いて解説していきます。

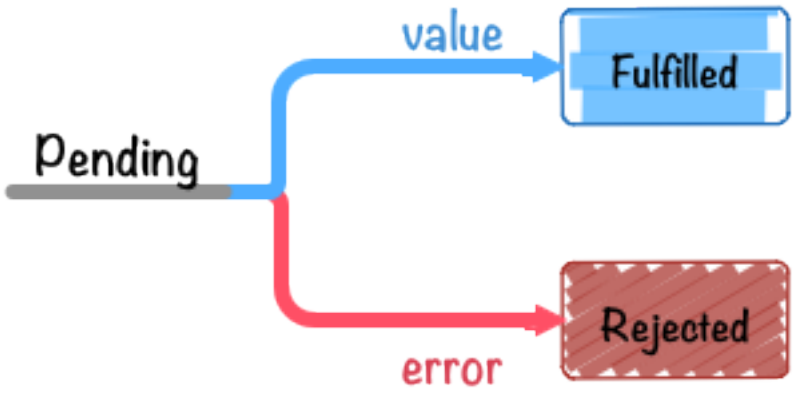


Figure 1. promise states

ES Promisesの仕様 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、すで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回限りということ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(成功) または reject(失敗) した時。

Pending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分かると思います。

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。

JavaScript Promises - Thinking Sync in an Async World // Speaker Deck[20] 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。

[20] https://speakerdeck.com/kerrick/javascript-promises-thinking-sync-in-an-async-world

Promiseの書き方

Promiseの基本的な書き方について解説します。

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. `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(結果の値)` を呼ぶ
    - 処理結果がエラーなら、`reject(Errorオブジェクト)` を呼ぶ

この流れに沿っているものを実際に書いてみましょう。

非同期処理であるXMLHttpRequest(XHR)を使いデータを取得するものをPromiseで書いていきます。

XHRのpromiseオブジェクトを作る

まずは、XHRをPromiseを使って包んだような `fetchURL` という関数を作ります。

xhr-promise.js

```
function fetchURL(URL) {
    return new Promise((resolve, reject) => {
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
```

```
// 実行例
const URL = "https://httpbin.org/get";
fetchURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(function onRejected(error){
    console.error(error);
});
```

この `fetchURL` では、XHRでの取得結果のステータスコードが200以上300未満の場合は `resolve` - つまり取得に成功、それ以外はエラーであるとして `reject` しています。

`resolve(req.responseText)` 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できます)

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` の方を見ていきましょう。

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オブジェクトを作成して渡していることが分かると思います。`reject` に渡す値に制限はありませんが、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2xx以外であるならrejectするとしていたため、`reject` にはstatusTextを入れています。(この値は `then` メソッドの第二引数 or `catch` メソッドで受け取ることができます)

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。

```
fetchURL("https://httpbin.org/get"); // => promiseオブジェクトが返ってくる
```

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メソッドを持っており、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)

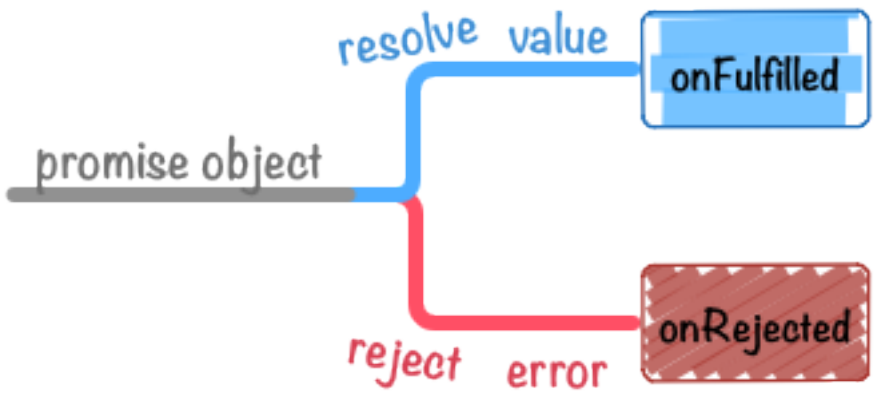


Figure 2. promise value flow

まずは、`fetchURL` で通信が成功して値が取得でき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、resolveされたことにより promiseオブジェクトが Fulfilledの状態になった 時ということですね。

resolveされた時の処理は、`.then` メソッドに呼びたい関数を渡すことで行えます。

```
const URL = "https://httpbin.org/get";
fetchURL(URL).then(function onFulfilled(value){ ❶
    console.log(value);
});
```

❶　分かりやすくするため関数に `onFulfilled` という名前を付けています

fetchURL関数 内で `resolve(req.responseText);` 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、`fetchURL` 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、rejectされたことより promiseオブジェクトが Rejectedの状態になった 時ということですね。

rejectされた時の処理は、`.then` の第二引数 または `.catch` メソッドに呼びたい関数を渡すことで行えます。

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。

```
const URL = "https://httpbin.org/status/500"; ❶
fetchURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(function onRejected(error){ ❷
    console.error(error);
});
```

❶　サーバはステータスコード500のレスポンスを返す
❷　分かりやすくするため関数 `onRejected` という名前を付けています
`fetchURL` の処理中に何らかの理由で例外が起きた場合、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できます。

```
fetchURL(URL).then(onFulfilled, onRejected);❶
```

❶　onFulfilled, onRejected それぞれは先ほどと同じ関数
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れますが、両者の違いについては then or catch? 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` を使った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の基本的な書き方について学びました。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、同様のことはコールバック関数を渡す形でもできるのに対してPromiseで書くメリットについては触れていませんでした。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いきたいと思います。

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。

Promise.resolve

一般に `new Promise()` を使うことで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。

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。

たとえば、`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((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できます。

```
Promise.resolve(42).then((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。

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。

ES 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ことを言います。

`.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を言います。

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

どのようなものがthenableなのかというと、分かりやすい例では jQuery.ajax()[21] の返り値もthenableです。

`jQuery.ajax()` の返り値は jqXHR Object[22] というもので、このオブジェクトは `.then` というメソッドを持っているためです。

```
$.ajax("https://httpbin.org/get");// => `.then` をもつオブジェクト
```

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できます。

promiseオブジェクトにすることができれば、`then` や `catch` といった、ES Promisesがもつ機能をそのまま利用することができるようになります。

thenableをpromiseオブジェクトにする

```
// このサンプルコードはjQueryをロードしている場所でないと動きません
const promise = Promise.resolve($.ajax("https://httpbin.org/get"));// => promiseオブジェクト
promise.then((value) => {
    console.log(value);
});
```

jQueryとthenable

> jQuery.ajax()[23]の返り値も `.then` というメソッドを持った jqXHR Object[24]で、このオブジェクトは Deferred Object[25] のメソッドやプロパティ等を継承しています。
>
> しかし、jQuery 2.x以下では、このDeferred ObjectはPromises/A+やES Promisesに準拠したものではありません。そのため、Deferred

[21] https://api.jquery.com/jQuery.ajax/
[22] http://api.jquery.com/jQuery.ajax/#jqXHR
[23] https://api.jquery.com/jQuery.ajax/
[24] http://api.jquery.com/jQuery.ajax/#jqXHR
[25] http://api.jquery.com/category/deferred-object/

> Objectをpromiseオブジェクトへ変換できたように見えて、一部欠損する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[26] 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ES Promisesとして使えるとは限らないことは知っておくべきでしょう。
>
> - JavaScript Promises: There and back again - HTML5 Rocks[27]
> - You're Missing the Point of Promises[28]
>
> なお、jQuery 3.0からは、Deferred Object[29]や jqXHR Object[30]がPromises/A+準拠へと変更されています。そのため、上記で紹介されている `.then` の挙動が異なる問題は解消されています。
>
> - jQuery 3.0 Final Released! | Official jQuery Blog[31]

`Promise.resolve` は共通の挙動である `then` だけを利用して、さまざまなライブラリ間でのpromiseオブジェクトを相互に変換して使える仕組みを持っていることになります。

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ことからもその挙動が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等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。

> 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。

26 http://api.jquery.com/category/deferred-object/
27 http://www.html5rocks.com/ja/tutorials/es6/promises/#toc-lib-compatibility
28 http://domenic.me/2012/10/14/youre-missing-the-point-of-promises/
29 http://api.jquery.com/category/deferred-object/
30 http://api.jquery.com/jQuery.ajax/#jqXHR
31 https://blog.jquery.com/2016/06/09/jquery-3-0-final-released/

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。

たとえば、`Promise.reject(new Error("エラー"))` 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((resolve, reject) => {
    reject(new Error("エラー"));
});
```

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((error) => {
    console.error(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。

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` で登録した関数が呼ばれるのは、非同期となります。

```
const promise = new Promise((resolve) => {
    console.log("inner promise"); // 1
    resolve(42);
});
promise.then((value) => {
    console.log(value); // 3
});
console.log("outer promise"); // 2
```

上記のコードを実行すると以下の順に呼ばれていることが分かります。

```
inner promise // 1
```

```
outer promise // 2
42            // 3
```

JavaScriptは上から実行されていくため、まず最初に `<1>` が実行されますね。そして次に `resolve(42);` が実行され、この `promise` オブジェクトはこの時点で `42` という値にFulfilledされます。

次に、`promise.then` で `<3>` のコールバック関数を登録しますが、ここがこのコラムの焦点です。

`promise.then` を行う時点でpromiseオブジェクトの状態が決まっているため、プログラム的には同期的にコールバック関数に `42` を渡して呼び出すことはできますね。

しかし、Promiseでは `promise.then` で登録する段階でpromiseの状態が決まっていても、そこで登録したコールバック関数は非同期で呼び出される仕様になっています。

そのため、`<2>` が先に呼び出されて、最後に `<3>` のコールバック関数が呼ばれています。

なぜ、同期的に呼び出せるのにわざわざ非同期的に呼び出しているでしょうか?

同期と非同期の混在の問題

これはPromise以外でも適用できるため、もう少し一般的な問題として考えてみましょう。

この問題はコールバック関数を受け取る関数が、状況によって同期処理になるのか非同期処理になるのかが変わってしまう問題と同じです。

次のような、コールバック関数を受け取り処理する `onReady(fn)` を見てみましょう。

mixed-onready.js

```
function onReady(fn) {
    const readyState = document.readyState;
    if (readyState === "interactive" || readyState === "complete") {
        fn();
    } else {
        window.addEventListener("DOMContentLoaded", fn);
    }
}
onReady(() => {
    console.log("DOM fully loaded and parsed");
});
console.log("==Starting==");
```

mixed-onready.jsではDOMが読み込み済みかどうかで、コールバック関数が同期的か非同期的に呼び出されるのかが異なっています。

onReadyを呼ぶ前にDOMの読み込みが完了している
　同期的にコールバック関数が呼ばれる

onReadyを呼ぶ前にDOMの読み込みが完了していない
　`DOMContentLoaded` のイベントハンドラとしてコールバック関数を設定する

そのため、このコードは配置する場所によって、コンソールに出てくるメッセージの順番が変わってしまいます。

この問題の対処法は、常に非同期で呼び出すように統一することです。

async-onready.js

```
function onReady(fn) {
    const readyState = document.readyState;
    if (readyState === "interactive" || readyState === "complete") {
        setTimeout(fn, 0);
    } else {
        window.addEventListener("DOMContentLoaded", fn);
    }
}
onReady(() => {
    console.log("DOM fully loaded and parsed");
});
console.log("==Starting==");
```

この問題については、Effective JavaScript[32] の 項目67 非同期コールバックを同期的に呼び出してはいけない で紹介されています。

- 非同期コールバックは（たとえデータが即座に利用できても）決して同期的に使ってはならない。
- 非同期コールバックを同期的に呼び出すと、処理の期待されたシーケンスが乱され、コードの実行順序に予期しない変動が生じるかもしれない。
- 非同期コールバックを同期的に呼び出すと、スタックオーバーフローや例外処理の間違いが発生するかもしれない。
- 非同期コールバックを次回に実行されるようスケジューリングするには、`setTimeout` のような非同期APIを使う。

[32] http://effectivejs.com/

— David Herman Effective JavaScript

先ほどの `promise.then` も同様のケースであり、この同期と非同期処理の混在の問題が起きないようにするため、Promiseは常に非同期 で処理されるということが仕様で定められているわけです。

最後に、この `onReady` をPromiseを使って定義すると以下のようになります。

onready-as-promise.js

```
function onReadyPromise() {
    return new Promise((resolve) => {
        const readyState = document.readyState;
        if (readyState === "interactive" || readyState === "complete") {
            resolve();
        } else {
            window.addEventListener("DOMContentLoaded", resolve);
        }
    });
}
onReadyPromise().then(() => {
    console.log("DOM fully loaded and parsed");
});
console.log("==Starting==");
```

Promiseは常に非同期で実行されることが保証されているため、`setTimeout` のような明示的に非同期処理にするためのコードが不要となることが分かります。

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分かるように、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できます。

promiseはメソッドチェーンで繋げて書ける

```
aPromise.then((value) => {
// task A
}).then((value) => {
// task B
}).catch((error) => {
    console.error(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できます。

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。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。

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いきましょう。

> promise chain - すなわちメソッドチェーンが短いことはよいことです。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。

promise-then-catch-flow.js

```
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");
}

const 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。

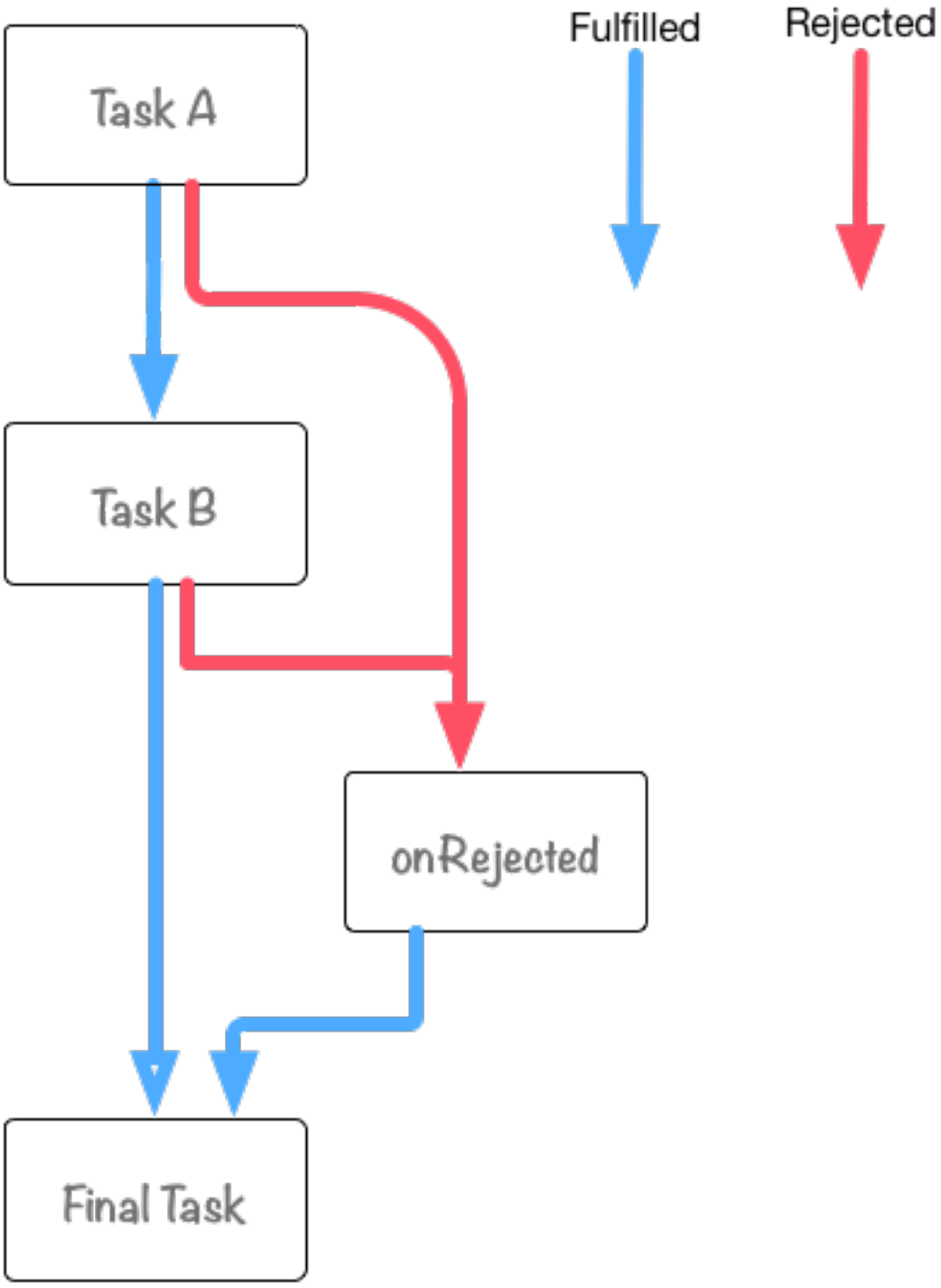


Figure 3. promise-then-catch-flow.jsの図

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、次のような場合に onRejected が呼ばれるということを示しています。

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

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。

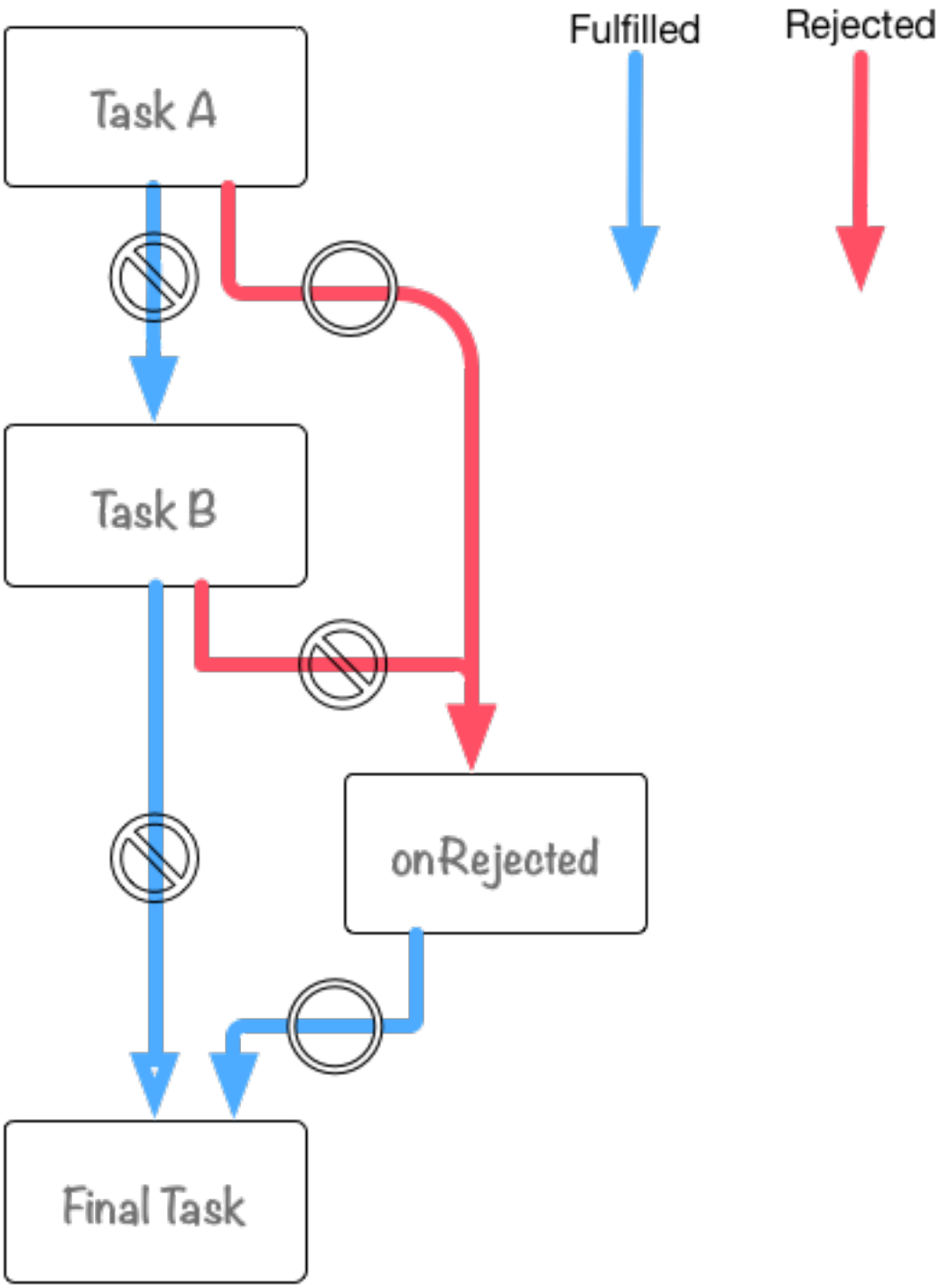


Figure 4. Task Aで例外が発生した時の図

コードにしてみると以下のようになります。

promise-then-taska-throw.js

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

```
function taskB() {
    console.log("Task B");// 呼ばれない
}
function onRejected(error) {
    console.error(error);// => "throw Error @ Task A"
}
function finalTask() {
    console.log("Final Task");
}

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

実行してみると、Task B が呼ばれていないことが分かるでしょう。

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。

promise chainでの値渡し

先ほどの例ではそれぞれのTaskが独立していて、ただ呼ばれているだけでした。

このときに、Task AがTask Bへ値を渡したい時はどうすればよいでしょうか?

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。

promise-then-passing-value.js

```
function doubleUp(value) {
    return value * 2;
}
function increment(value) {
    return value + 1;
}
function output(value) {
```

```
    console.log(value);// => (1 + 1) * 2
}

const promise = Promise.resolve(1);
promise
    .then(increment)
    .then(doubleUp)
    .then(output)
    .catch((error) => {
        // promise chain中にエラーが発生した場合に呼ばれる
        console.error(error);
    });
```

スタートは `Promise.resolve(1);` で、この処理は以下のような流れで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

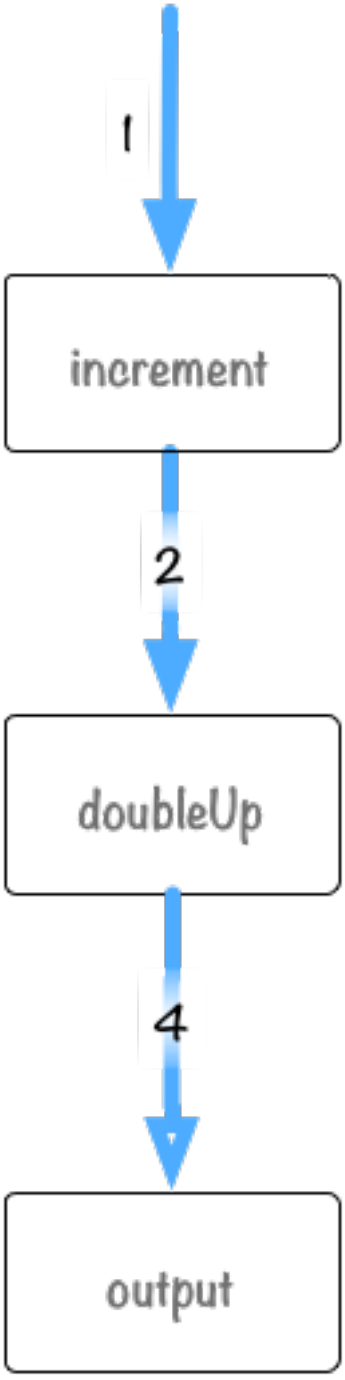


Figure 5. promise-then-passing-value.jsの図

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` することができます。

returnした値は `Promise.resolve(returnされた値);` のように処理されるため、何をreturnしても最終的には新しいpromiseオブジェクトを返します。

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。

つまり、`Promise#then` は単にコールバックとなる関数を登録するだけではなく、受け取った値を変化させて別のpromiseオブジェクトを生成するという機能も持っていることを覚えておくといいでしょう。

Promise#catch

先ほどのPromise#thenについてでも `Promise#catch` はすで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。

次のコードのように `Promise#catch` は `Promise#then` でのエラーハンドリングだけを簡潔に書くためのメソッドです。

Promise#catchとPromise#then

```
Promise.reject(new Error("message")).catch((error) => {
    // エラーハンドリング
});
// Promise#catchは次のPromise#thenと同じ意味
Promise.reject(new Error("message")).then(undefined, (error) => {
    // エラーハンドリング
});
```

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。

IE8以下での問題

このバッジは以下のコードが、polyfill[33] 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。

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[34] を利用しています。

Promise#catchの実行結果

[33] https://github.com/jakearchibald/es6-promise
[34] https://github.com/jakearchibald/es6-promise

```
const promise = Promise.reject(new Error("message"));
promise.catch((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。

これはどういうことかというと、`catch` という単語はECMAScriptにおける 予約語[35] 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できませんでした。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ので、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[36] 、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。

> ECMAScript 5でも予約語は Identifier[37] 、つまり変数名、関数名には利用することができません。`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する書き方も存在します。

ドット表記法[38] 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[39] 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できます。(もちろんpolyfillは必要です)

Promise#catchの識別子エラーの回避

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

35 http://mothereff.in/js-properties#catch
36 http://es5.github.io/#x7.6
37 http://es5.github.io/#x7.6
38 https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Property_Accessors#Dot_notation
39 https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Property_Accessors#Bracket_notation

```
promise["catch"]((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。

Promise#catchではなくPromise#thenを使う

```
const promise = Promise.reject(new Error("message"));
promise.then(undefined, (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。

また多くの圧縮ツールは `promise.catch` を `promise["catch"]` へと置換する処理が組み込まれているため、知らない間に回避できていることも多いかも知れません。

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。

Promise#finally

ECMAScript 2018からpromise chainの最後に処理を実行する `Promise#finally` メソッド追加されました。

`Promise#finally` メソッドは成功時、失敗時どちらの場合でも呼び出すコールバック関数を登録できます。`try…catch…finally` 構文の `finally` 節と同様の役割をもつメソッドです。

次のコードのように、`Promise#finally` メソッドで登録したコールバック関数は、promiseオブジェクトが resolve(成功) / reject(失敗) どちらの場合でも呼ばれます。

finallyのコード例

```
Promise.resolve("成功").finally(() => {
    console.log("成功時に実行される");
});
Promise.reject(new Error("失敗")).finally(() => {
    console.log("失敗時に実行される");
});
```

`finally` メソッドのコールバック関数は引数を受け取らず、どのような値を返してもpromise chainには影響を与えません。また、`finally` メソッドは新しいpromiseオブ

ジェクトを返し、新しいpromiseオブジェクトは呼び出し元のpromiseオブジェクトの状態をそのまま引き継ぎます。

finallyとpromise chain

```
function onFinally() {
    // 成功、失敗どちらでも実行したい処理
}

// `Promise#finally` は新しいpromiseオブジェクトを返す
Promise.resolve(42)
    .finally(onFinally)
    .then((value) => {
        // 呼び出し元のpromiseオブジェクトの状態をそのまま引き継ぐ
        // 呼び出し元のpromiseオブジェクトは `42` で resolveされている
        console.log(value); // 42
    });
```

`Promise#finally` メソッドと同等の表現を `Promise#then` メソッドで書くと次のように書けます。

finallyをthenで表現

```
function onFinally() {
    // 成功、失敗どちらでも実行したい処理
}

// Promise#finally(onFinally) と同等の表現
promise.then((result) => {
    onFinally();
    return result;
}, (error) => {
    onFinally();
    return Promise.reject(error);
});
```

`Promise#finally` メソッドを使うことで、promise chainで必ず実行したい処理を簡単に書けるようになっています。

次のコードでは、リソースを取得中かどうかを判定するためのフラグを `isLoading` という変数で管理しています。`Promise#finally` メソッドを使い成功失敗どちらにもかかわらず、取得が終わったら `isLoading` は `false` にしています。

finallyのユースケース

```
// リソースを取得中かどうかのフラグ
```

```
let isLoading = false;
function fetchResource(URL) {
    // リソース取得中フラグをONに
    isLoading = true;
    return new Promise((resolve, reject) => {
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    }).finally(() => {
        // リソース取得に成功/失敗どちらの場合も取得中フラグをOFFに
        isLoading = false;
    });
}

console.log("リソースロード開始", isLoading);
fetchResource("https://httpbin.org/get").then((value) => {
    console.log("リソース取得に成功", isLoading);
    console.log(value);
}).catch((error) => {
    console.log("リソース取得に失敗", isLoading);
    console.error(error);
});
console.log("リソースロード中", isLoading);
```

`then` と `catch` メソッドでも実現できますが、`Promise#finally` メソッドを使うことで `isLoading` の代入を一箇所にまとめられます。

コラム: thenは常に新しいpromiseオブジェクトを返す

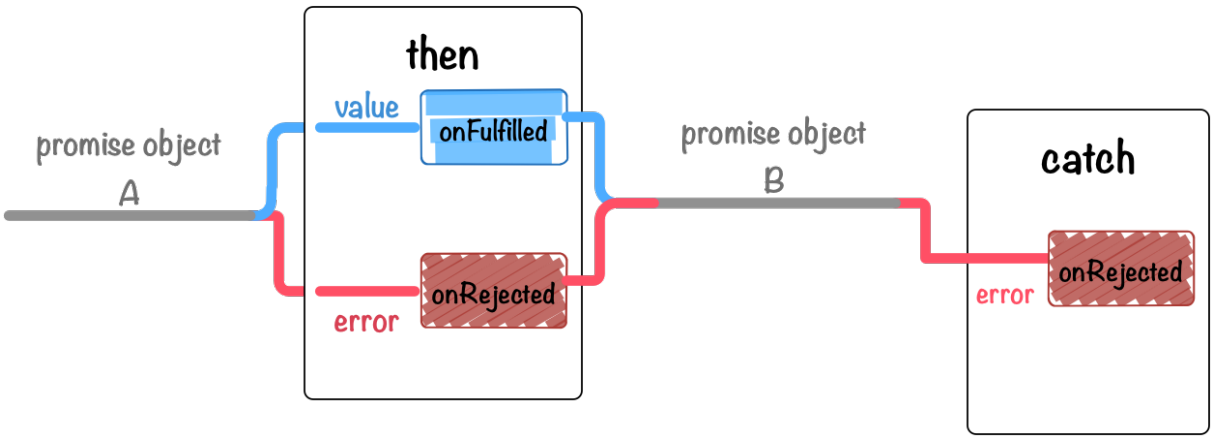
`aPromise.then(…).catch(…)` は一見すると、全て最初の `aPromise` オブジェクトに メソッドチェーンで処理を書いてるように見えます。

しかし、実際には `then` で新しいpromiseオブジェクト、`catch` でも別の新しい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。

```
const aPromise = new Promise((resolve) => {
    resolve(100);
});
const thenPromise = aPromise.then((value) => {
    console.log(value);
});
const catchPromise = thenPromise.catch((error) => {
    console.error(error);
});
console.log(aPromise !== thenPromise); // => true
console.log(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こと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ことに気づくでしょう。

```
// 1: それぞれの `then` は同時に呼び出される
const aPromise = new Promise((resolve) => {
    resolve(100);
});
aPromise.then((value) => {
    return value * 2;
});
aPromise.then((value) => {
    return value * 2;
});
```

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

// vs

// 2: `then` はpromise chain通り順番に呼び出される
const bPromise = new Promise((resolve) => {
    resolve(100);
});
bPromise.then((value) => {
    return value * 2;
}).then((value) => {
    return value * 2;
}).then((value) => {
    console.log("2: " + 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。

✘ `then` の間違った使い方

```
function badAsyncCall() {
    const promise = Promise.resolve();
    promise.then(() => {
        // 何かの処理
        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を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できます。

`then` で作成したオブジェクトを返す

```
function anAsyncCall() {
    const promise = Promise.resolve();
    return promise.then(() => {
        // 何かの処理
        return newVar;
    });
}
```

これらのアンチパターンについて、詳しくは Promise Anti-patterns[40] を参照して下さい。

この挙動はPromise全般に当てはまるため、後に説明する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。

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` で登録でき、`.finally` を使うことで Fulfilled と Rejected どちらの場合でも実行される処理を登録できることを学びました。

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よいですが、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?

たとえば、複数のXHR(非同期処理)が全て終わった後に、何かをしたいという事例を考えてみます。

少し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。

CORSについて

> ブラウザにおけるXHRのリソース取得には、CORS(Cross-Origin Resource Sharing[41])というセキュリティ上の制約が存在します。
>
> このCORSの制約により、ブラウザでは同一ドメインではないリソースを許可なく取得することはできません。そのため、一般的には別サイトのリソースは許可なくXHRでアクセスすることができません。

[40] http://taoofcode.net/promise-anti-patterns/
[41] https://developer.mozilla.org/ja/docs/Web/HTTP/HTTP_access_control

> 次のサンプルでは `https://azu.github.io/promises-book/json/comment.json` という `azu.github.io` ドメイン以下にあるリソースを取得する例が登場します。
>
> `azu.github.io` ドメイン以下のJSONには、別ドメインからの取得が許可する設定がされています。
>
> また、httpbin.org[42] というドメインがリソース取得の例として登場します。こちらも、同一ドメインでなくてもリソースの取得が許可されています。

コールバックで複数の非同期処理

multiple-xhr-callback.js

```
function fetchURLCallback(URL, callback) {
    const req = new XMLHttpRequest();
    req.open("GET", URL, true);
    req.onload = () => {
        if (200 <= req.status && req.status < 300) {
            callback(null, req.responseText);
        } else {
            callback(new Error(req.statusText), req.response);
        }
    };
    req.onerror = () => {
        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 {
            const result = JSON.parse(value);
            callback(null, result);
        } catch (e) {
            callback(e, value);
        }
    }
```

[42] https://httpbin.org/

```
}
// <2> XHRを叩いてリクエスト
const request = {
    comment(callback) {
        return fetchURLCallback("https://azu.github.io/promises-book/json/comment.json",
 jsonParse.bind(null, callback));
    },
    people(callback) {
        return fetchURLCallback("https://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);
    }
    const req = requests.shift();
    req((error, value) => {
        if (error) {
            callback(error, value);
        } else {
            results.push(value);
            allRequest(requests, callback, results);
        }
    });
}

function main(callback) {
    allRequest([request.comment, request.people], callback, []);
}



// 実行例
main(function(error, results){
    if(error){
        console.error(error);
        return;
    }
    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)` のように第一引数にエラー、第二引数にレスポンスを渡す。

`jsonParse` 関数を使うときに `bind` を使うことで、部分適用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);
// は以下のように置き換えるのと殆ど同じ
function bindJSONParse(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` を使って同様のことをしてみたいと思います。

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。

multiple-xhr.js

```
function fetchURL(URL) {
    return new Promise((resolve, reject) => {
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
```

```
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
const request = {
    comment() {
        return fetchURL("https://azu.github.io/promises-book/json/
comment.json").then(JSON.parse);
    },
    people() {
        return fetchURL("https://azu.github.io/promises-book/json/
people.json").then(JSON.parse);
    }
};
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] は記録する初期値を部分適用している
    const pushValue = recordValue.bind(null, []);
    return request.comment()
        .then(pushValue)
        .then(request.people)
        .then(pushValue);
}



// 実行例
main().then((value) => {
    console.log(value);
}).catch((error) => {
    console.error(error);
});
```

コールバックスタイルと比較してみると次のことがわかります。

- `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。

Promiseでは、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が用意されています。

次のセクションではそれらについて学んでいきましょう。

Promise.all

`Promise.all` は promiseオブジェクトの配列を受け取り、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。

先ほどの複数のXHRの結果をまとめて取得する処理は、`Promise.all` を使うとシンプルに書くことができます。

先ほどの例の `fetchURL` はXHRによる通信を抽象化したpromiseオブジェクトを返しています。`Promise.all` に通信を抽象化したpromiseオブジェクトの配列を渡すことで、全ての通信が完了(FulfilledまたはRejected)した時に、次の `.then` を呼び出すことができます。

promise-all-xhr.js

```
function fetchURL(URL) {
    return new Promise((resolve, reject) => {
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
const request = {
    comment() {
        return fetchURL("https://azu.github.io/promises-book/json/
comment.json").then(JSON.parse);
```

```
    },
    people() {
        return fetchURL("https://azu.github.io/promises-book/json/
people.json").then(JSON.parse);
    }
};
function main() {
    return Promise.all([request.comment(), request.people()]);
}



// 実行例
main().then((value) => {
    console.log(value);
}).catch((error) => {
    console.error(error);
});
```

実行方法は 前回のもの と同じですね。`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((results) => {
    console.log(results); // [comment, people]の順番
});
```

`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。

promise-all-timer.js

```
// `delay`ミリ秒後にresolveする
function timerPromisefy(delay) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve(delay);
        }, delay);
    });
}
const startDate = Date.now();
// 全てがresolveされたら終了
Promise.all([
    timerPromisefy(1),
    timerPromisefy(32),
    timerPromisefy(64),
    timerPromisefy(128)
]).then((values) => {
    console.log(Date.now() - startDate + "ms");// 約128ms
    console.log(values); // [1,32,64,128]
});
```

`timerPromisefy` は引数で指定したミリ秒後に、その指定した値でFulfilledとなるpromiseオブジェクトを返してくれます。

`Promise.all` に渡してるのは、それを複数作り配列にしたものですね。

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

この場合は、1, 32, 64, 128 ミリ秒後にそれぞれ `resolve` されます。

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128ms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128msかかることがわかります。

このことから、`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ことがわかります。

> 仮に逐次的に行われていた場合は、1ms待機 → 32ms待機 → 64ms待機 → 128ms待機 となるので、全て完了するまで225ms程度かかる計算になります。

> 実際にPromiseを逐次的に処理したいケースについては第4章のPromiseによる逐次処理を参照して下さい。

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Fulfilled または Rejected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Fulfilled または Rejectedになったら次の処理を実行します。

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

promise-race-timer.js

```
// `delay`ミリ秒後にresolveする
function timerPromisefy(delay) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve(delay);
        }, delay);
    });
}
// 一つでもresolve または reject した時点で終了
Promise.race([
    timerPromisefy(1),
    timerPromisefy(32),
    timerPromisefy(64),
    timerPromisefy(128)
]).then((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Fulfilledとなりますが、一番最初に1msのものがFulfilledとなった時点で、`.then` が呼ばれます。また、`resolve(1)` が呼ばれるため `value` に渡される値も1となります。

最初にFulfilledとなったpromiseオブジェクト以外は、その後呼ばれているのかを見てみましょう。

promise-race-other.js

```
const winnerPromise = new Promise((resolve) => {
    setTimeout(() => {
        console.log("this is winner");
        resolve("this is winner");
    }, 4);
});
const loserPromise = new Promise((resolve) => {
    setTimeout(() => {
        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((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、winner/loserどちらも `setTimeout` の中身が実行されて `console.log` がそれぞれ出力されていることがわかります。

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ことがわかります。

> ES 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いえます。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。

then or catch?

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

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。

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。

then-throw-error.js

```
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);
}



// 実行例
badMain(function(){
    console.log("BAD");
});
goodMain(function(){
    console.log("GOOD");
});
```

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できますが、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ことはできません。

つまり、この場合、`throwError` 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 `throwError` → `onRejected` となるように書かれています。この場合は `throwError` 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ことができます。

`.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。

`.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。

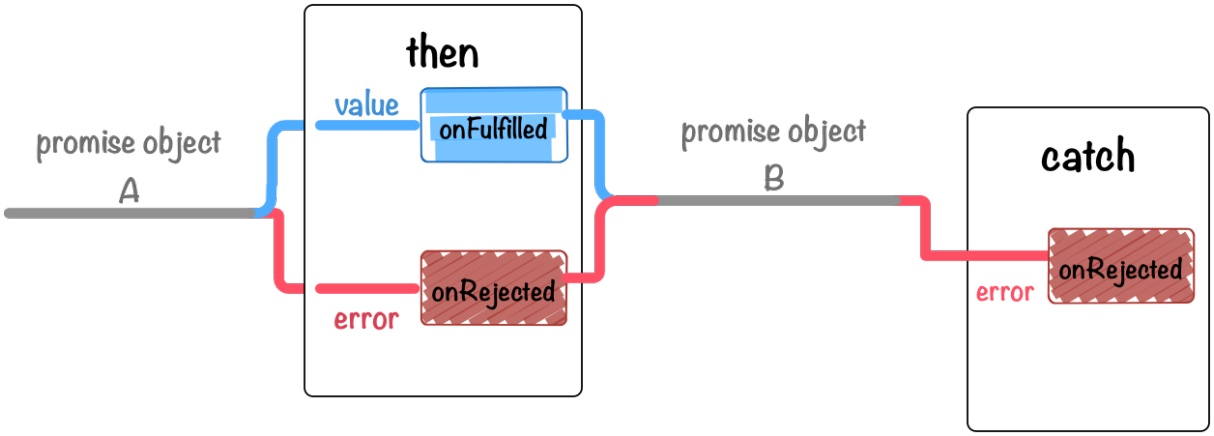


Figure 6. Then Catch flow

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できるのは、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);
```

まとめ

ここでは次のようなこと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` で発生した例外を `.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。

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。

基本的なテスト

ES Promisesのメソッド等についてひととお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できると思います。

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[43]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るため、各自Node.js環境を用意してください。

この書籍中に出てくるサンプルコードはそれぞれテストも書かれています。テストコードは azu/promises-book[44] から参照できます。

Mochaとは

Mochaの公式サイト: https://mochajs.org/

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み合わせて利用します。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を選択した理由は、以下のとおりです。

- 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

[43] https://mochajs.org/
[44] https://github.com/azu/promises-book

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。

この章ではMochaを利用するため、npmを使いMochaをインストールしておく必要があります。

```
$ npm install -g mocha
```

また、アサーション自体はNode.jsに同梱されている `assert` モジュールを使用するので別途インストールは必要ありません。

まずはコールバックスタイルの非同期処理をテストしてみましょう。

コールバックスタイルのテスト

コールバックスタイルの非同期処理をテストする場合、Mochaでは以下のように書くことができます。

basic-test.js

```
const assert = require("assert");
it("should use `done` for test", (done) => {
    setTimeout(() => {
        assert(true);
        done();
    }, 0);
});
```

このテストを `basic-test.js` というファイル名で作成し、先ほどインストールしたMochaでコマンドラインからテストを実行することができます。

```
$ mocha basic-test.js
```

Mochaは `it` の仮引数に `done` のように指定してあげると、`done()` が呼ばれるまでテストの終了を待つことで非同期のテストをサポートしています。

Mochaでの非同期テストは以下のような流れで実行されます。

```
it("should use `done` for test", (done) => {
    ❶
    setTimeout(() => {
        assert(true);
```

```
        done();❷
    }, 0);
});
```

❶　コールバックを使う非同期処理
❷　`done` を呼ぶことでテストが終了する
よく見かける形の書き方ですね。

`done` を使ったPromiseのテスト

次に、同じく `done` を使ったPromiseのテストを書いてみましょう。

```
it("should use `done` for test?", (done) => {
    const promise = Promise.resolve(42);❶
    promise.then((value) => {
        assert(value === 42);
        done();❷
    });
});
```

❶　`Fulfilled` となるpromiseオブジェクトを作成
❷　`done` を呼ぶことでテストの終了を宣言
`Promise.resolve` はpromiseオブジェクトを返しますが、そのpromiseオブジェクトはFulfilledの状態になります。その結果として `.then` で登録したコールバック関数が呼び出されます。

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。

しかし、先ほど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。

意図しない結果となるPromiseのテスト

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

このテストは `assert` が失敗しているため、「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。

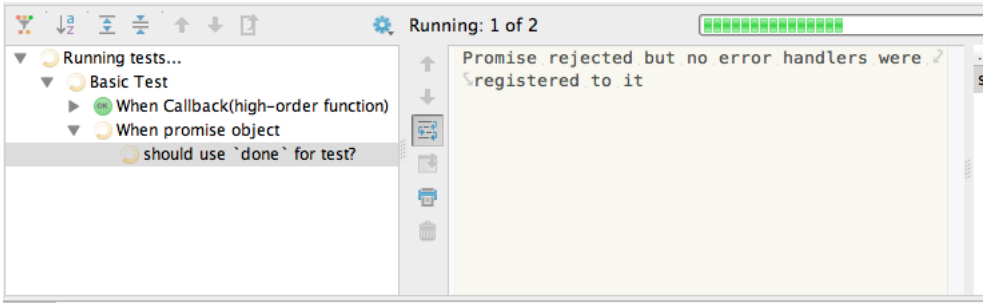


Figure 7. テストが終わることがないためタイムアウトするまでそこで止まる

`assert` が失敗した場合は通常はエラーをthrowし、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届きません。

意図しない結果となるPromiseのテストを改善して、`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。

意図通りにテストが失敗する例

```
it("should use `done` for test?", (done) => {
    const promise = Promise.resolve();
    promise.then((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` が失敗した時のために `.then(done, done);` というものを付ける必要があります。Promiseのテストを書くときにつけ忘れてしまうと終わらないテストができ上がってしまう場合があることに気をつけましょう。

次に、最初にmochaを使う理由に上げた"Promisesのテスト"のサポートがどのような機能であるか学んでいきましょう。

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[45]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のテストの場合はコールバックとして `done()` を呼ぶ代わりに、promiseオブジェクトをreturnすることができると書いてあります。

では、実際にどのように書くかの例を見ていきたいと思います。

mocha-promise-test.js

```
const assert = require("assert");
describe("Promise Test", () => {
    it("should return a promise object", () => {
        const promise = Promise.resolve(42);
        return promise.then((value) => {
            assert(value === 42);
        });
    });
});
```

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つとなっています。

- `done` そのものを取り除いた
- 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、`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", () => {
    return Promise.resolve().then(() => {
        assert(false);// => テストが失敗する
    });
});
```

[45] https://mochajs.org/#asynchronous-code

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

MochaがPromisesのテストをサポートしました | Web scratch[46] 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。

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、この書き方でよいと思われるかもしれません。しかし、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。

たとえば、以下はある条件だとRejectedなpromiseオブジェクトを返す `mayBeRejected()` のテストコードです。

エラーオブジェクトをテストしたい

```
function mayBeRejected() { ❶
    return Promise.reject(new Error("woo"));
}
it("is bad pattern", () => {
    return mayBeRejected().catch((error) => {
        assert(error.message === "woo");
    });
});
```

❶　この関数が返すpromiseオブジェクトをテストしたい

このテストの目的とは以下のようになっています。

`mayBeRejected()` が返すpromiseオブジェクトがFulfilledとなった場合
　テストを失敗させる

`mayBeRejected()` が返すpromiseオブジェクトがRejectedとなった場合
　`assert` でErrorオブジェクトをチェックする

上記のテストコードでは、Rejectedとなって `onRejected` に登録された関数が呼ばれる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Fulfilledとなった場合に、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() { ❶
```

[46] http://efcl.info/2014/0314/res3708/

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

❶　返されるpromiseオブジェクトはFulfilledとなる
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` の前に `.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() { ❶
    throw new Error("Expected promise to be rejected but it was fulfilled");
}
function mayBeRejected() {
    return Promise.resolve();
}
it("should bad pattern", () => {
    return mayBeRejected().then(failTest).catch((error) => {
        assert(error.message === "woo");
    });
});
```

❶　throwすることでテストを失敗にしたい
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。

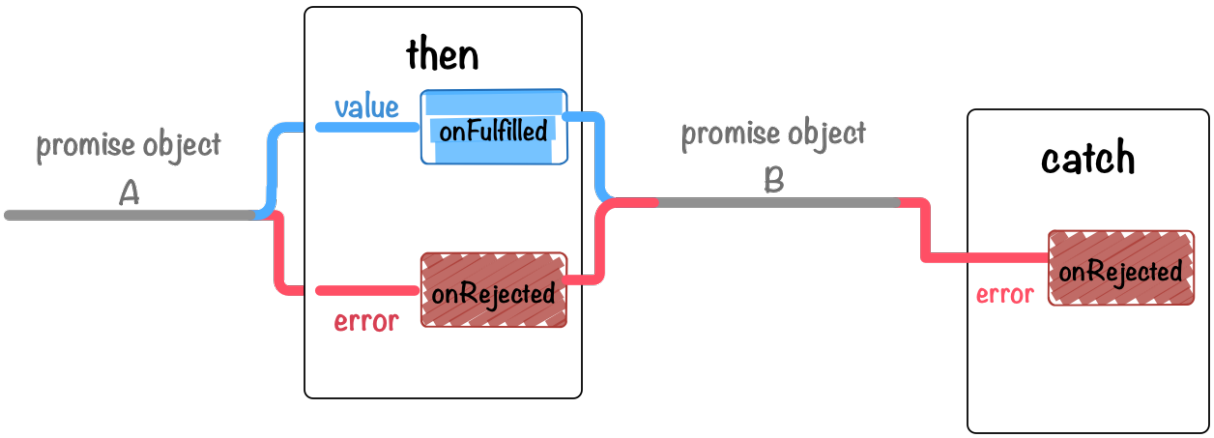


Figure 8. Then Catch flow

`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。

両状態を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のテストを書く場合、どのようにすれば意図せず通ってしまうテストを無くすことができるでしょうか?

一番単純な方法としては、以下のようにそれぞれの状態の場合にどうなるのかをテストコードに書く方法です。

Fulfilledとなった場合
　意図したとおりテストが失敗する

Rejectedとなった場合
　`assert` でテストを行える

つまり、Fulfilled、Rejected 両方の状態について、テストがどうなってほしいかを明示する必要があるわけです。

```
function mayBeRejected() {
    return Promise.resolve();
}
it("catch -> then", () => {
    // Fulfilledとなった場合はテストは失敗する
    return mayBeRejected().then(failTest, (error) => {
        assert(error.message === "woo");
    });
});
```

このように書くことで、Fulfilledとなった場合は失敗するテストコードを書くことができます。

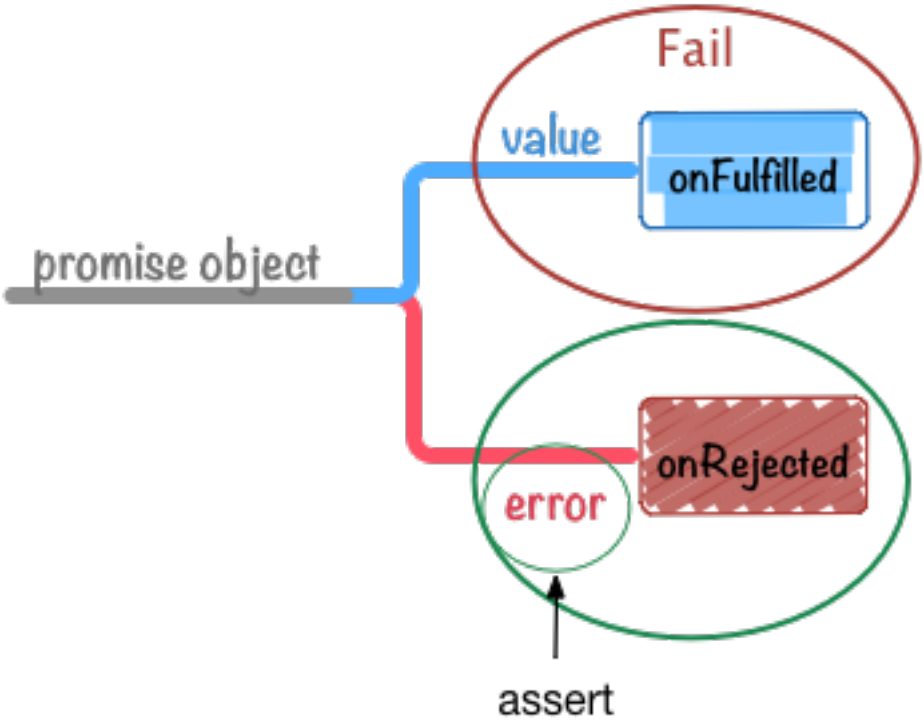


Figure 9. Promise onRejected test

then or catch?のときは、エラーの見逃しを避けるため、`.then(onFulfilled, onRejected)` の第二引数ではなく、`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(failTest, onRejected)` と書くことで、どちらの状態になるのかを明示してテストを書くことができました。

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Fulfilled、Rejectedどちらの状態になるかを明示してテストする必要があります。

しかし、Rejected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, (error) => {
    // assertでerrorをテストする
});
```

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どうするべきかについて見ていきましょう。

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

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

上記のケース(Fail)に該当しなければテストがパスするということですね。

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

先ほどの `.then` を使ったコードはRejectedを期待したテストとなっていますね。

```
promise.then(failTest, (error) => {
    // assertでerrorをテストする
    assert(error instanceof Error);
});
```

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、promiseオブジェクトに期待する状態を明示できるヘルパー関数を定義してみましょう。

ライブラリ化したものが azu/promise-test-helper[47] 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を期待してテストできる `shouldRejected` というヘルパー関数を作ってみたいと思います。

shouldRejected-test.js

```
const assert = require("assert");
function shouldRejected(promise) {
    return {
        "catch": function(fn) {
            return promise.then(() => {
                throw new Error("Expected promise to be rejected but it was fulfilled");
            }, (reason) => {
                fn.call(promise, reason);
            }
            );
        }
    };
}
it("should be rejected", () => {
    const promise = Promise.reject(new Error("human error"));
    return shouldRejected(promise).catch((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。

[47] https://github.com/azu/promise-test-helper

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

`shouldRejected` を使った場合、Fulfilledが呼ばれるとエラーをthrowしてテストが失敗するようになっています。

```
promise.then(failTest, (error) => {
    assert(error.message === "human error");
});
// == ほぼ同様の意味
shouldRejected(promise).catch((error) => {
    assert(error.message === "human error");
});
```

`shouldRejected` のようなヘルパー関数を使うことで、テストコードが少し直感的になりましたね。

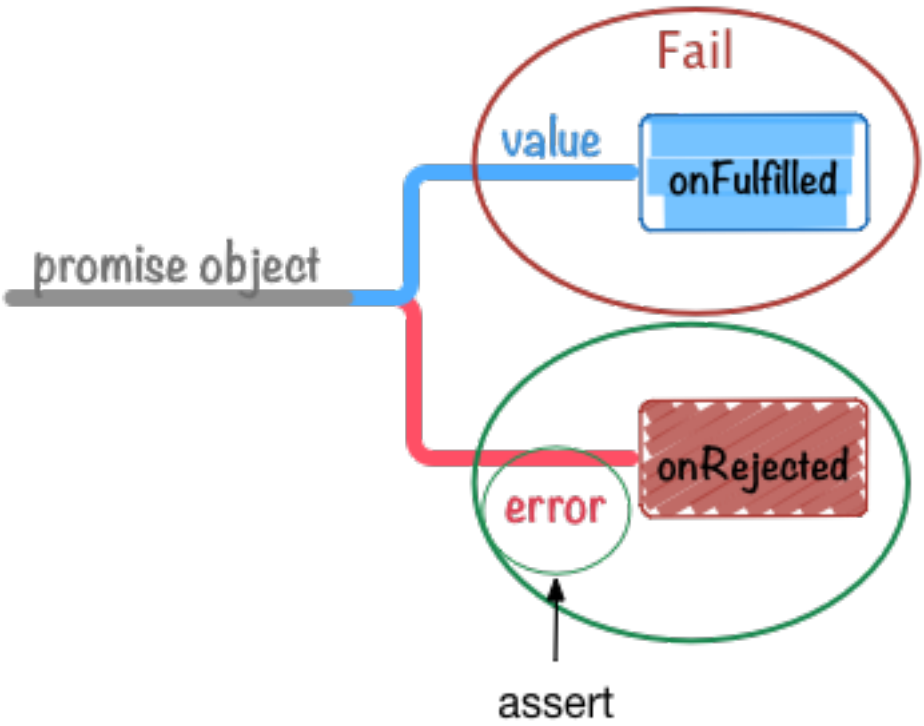


Figure 10. Promise onRejected test

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。

shouldFulfilled-test.js

```
const assert = require("assert");
function shouldFulfilled(promise) {
    return {
        "then": function(fn) {
            return promise.then((value) => {
                fn.call(promise, value);
            }, (reason) => {
                throw reason;
            }
            );
        }
    };
}
it("should be fulfilled", () => {
    const promise = Promise.resolve("value");
    return shouldFulfilled(promise).then((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。

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` はライブラリとして利用できるようになっています。
>
> azu/promise-test-helper[48] からダウンロードすることができます。
>
> また、Node.js 10.0.0から `assert.rejects` と `assert.doesNotReject` というよく似た趣旨のassertionが提供されています。詳細は、Node.jsのAPIドキュメント[49]を参照してください。

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` を使ったテストでは利用しにくいと思います。

[48] https://github.com/azu/promise-test-helper
[49] https://nodejs.org/api/assert.html

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。

たとえば、CoffeeScript[50]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いえます。

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。

Promiseのライブラリ

このセクションでは、ブラウザが実装しているPromiseではなく、サードパーティにより作られた Promise互換のライブラリについて紹介していきたいと思います。

なぜライブラリが必要か?

なぜライブラリが必要か?という疑問に関する多くの答えとしては、その実行環境で「ES Promisesが実装されていないから」というのがまず出てくるでしょう。

Promiseのライブラリを探すときに、一つ目印になる言葉としてPromises/A+互換があります。

Promises/A+というのはES Promisesの前身となったもので、Promiseの `then` について取り決めたコミュニティベースの仕様です。

Promises/A+互換と書かれていた場合は `then` についての動作は互換性があり、多くの場合はそれに加えて `Promise.all` や `catch` 等と同様の機能が実装されています。

しかし、Promises/A+は `Promise#then` についてのみの仕様となっているため、他の機能は実装されていても名前が異なる場合があります。

また、`then` というメソッドに互換性があるということは、Thenableであるということなので、Promise.resolveを使い、ESのPromiseで定められたpromiseオブジェクトに変換することができます。

[50] http://coffeescript.org/

ECMAScriptのPromiseで定められたpromiseオブジェクトというのは、`catch` というメソッドが使えたり、`Promise.all` で扱う際に問題が起こらないということです。

Polyfillとライブラリ

ここでは、大きくわけて2種類のライブラリを紹介したいと思います。

一つはPolyfillと呼ばれる種類のライブラリで、もう一つは、Promises/A+互換に加えて、独自の拡張をもったライブラリです。

Promiseのライブラリは星の数ほどあるので、ここで紹介するのは極々一部です。

Polyfill

Polyfillライブラリは読み込むことで、IEといったPromiseが実装されていないブラウザ等でも、Promiseと同等の機能を同じメソッド名で提供してくれるライブラリのことです。

つまり、Polyfillを読みこめばこの書籍で紹介しているコードは、Promiseがサポートされてない環境でも実行できるようになります。

zloirock/core-js[51]
: ECMAScriptやウェブ標準で定義されている仕様を実装したPolyfillライブラリです。多種多様な機能のPolyfillが含まれており、その一つとしてPromiseのPolyfillが実装されています。Babel[52]のプリセットにも組み込まれています。

jakearchibald/es6-promise[53]
: ES6(ES2015) Promisesと互換性を持ったPolyfillライブラリです。RSVP.js[54]というPromises/A+互換ライブラリがベースとなっており、これのサブセットとしてES6 PromisesのAPIだけが実装されているライブラリです。

taylorhakes/promise-polyfill[55]
: ES Promisesのpolyfillとなることを目的としたライブラリです。実行環境にネイティブのPromiseがある場合はそちらを優先し、上書きしないようにしています。

[51] https://github.com/zloirock/core-js
[52] https://babeljs.io/
[53] https://github.com/jakearchibald/es6-promise
[54] https://github.com/tildeio/rsvp.js
[55] https://github.com/taylorhakes/promise-polyfill

Promise拡張ライブラリ

Promiseを仕様どおりに実装したものに加えて独自のメソッド等を提供してくれるライブラリです。

Promise拡張ライブラリは本当に沢山ありますが、以下の2つの著名なライブラリを紹介します。

kriskowal/q[56]
: `Q` と呼ばれるPromisesやDeferredsを実装したライブラリです。2009年から開発されており、Node.js向けのファイルIOのAPIを提供する Q-IO[57] 等、多くの状況で使える機能が用意されているライブラリです。

petkaantonov/bluebird[58]
: Promise互換に加えて、キャンセルできるPromiseや進行度を取得できるPromise、エラーハンドリングの拡張検出等、多くの拡張を持っており、またパフォーマンスにも気を配った実装がされているライブラリです。

Q と Bluebird どちらのライブラリもブラウザでも動作する他、APIリファレンスが充実しているのも特徴的です。

- API Reference · kriskowal/q Wiki[59]

QのドキュメントにはjQueryがもつDeferredの仕組みとどのように違うのか、移行する場合の対応メソッドについても Coming from jQuery[60] にまとめられています。

- bluebird/API.md at master · petkaantonov/bluebird[61]

BluebirdではPromiseを使った豊富な機能に加えて、エラーが起きた時の対処法や Promiseのアンチパターン[62] について書かれています。

どちらのドキュメントも優れているため、このライブラリを使ってない場合でも読んでおくと参考になることが多いと思います。

[56] https://github.com/kriskowal/q
[57] https://github.com/kriskowal/q-io
[58] https://github.com/petkaantonov/bluebird
[59] https://github.com/kriskowal/q/wiki/API-Reference
[60] https://github.com/kriskowal/q/wiki/Coming-from-jQuery
[61] https://github.com/petkaantonov/bluebird/blob/master/API.md
[62] https://github.com/petkaantonov/bluebird/wiki/Promise-anti-patterns

まとめ

このセクションではPromiseのライブラリとしてPolyfillと拡張ライブラリを紹介しました。

Promiseのライブラリは多種多様であるため、どれを使用するかは好みの問題といえるでしょう。

しかし、PromiseはPromises/A+ または ES Promisesという共通のインターフェースを持っているため、そのライブラリで書かれているコードや独自の拡張などは、他のライブラリを利用している時でも参考になるケースは多いでしょう。

そのようなPromiseの共通の概念を学び、応用できるようになるのがこの書籍の目的の一つです。

Promise.resolveとThenable

第二章の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。

Web Notificationsをthenableにする

Web Notifications[63]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[64]
- Can I use Web Notifications[65]

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。

[63] https://developer.mozilla.org/ja/docs/Web/API/notification
[64] https://developer.mozilla.org/ja/docs/WebAPI/Using_Web_Notifications
[65] https://caniuse.com/notifications

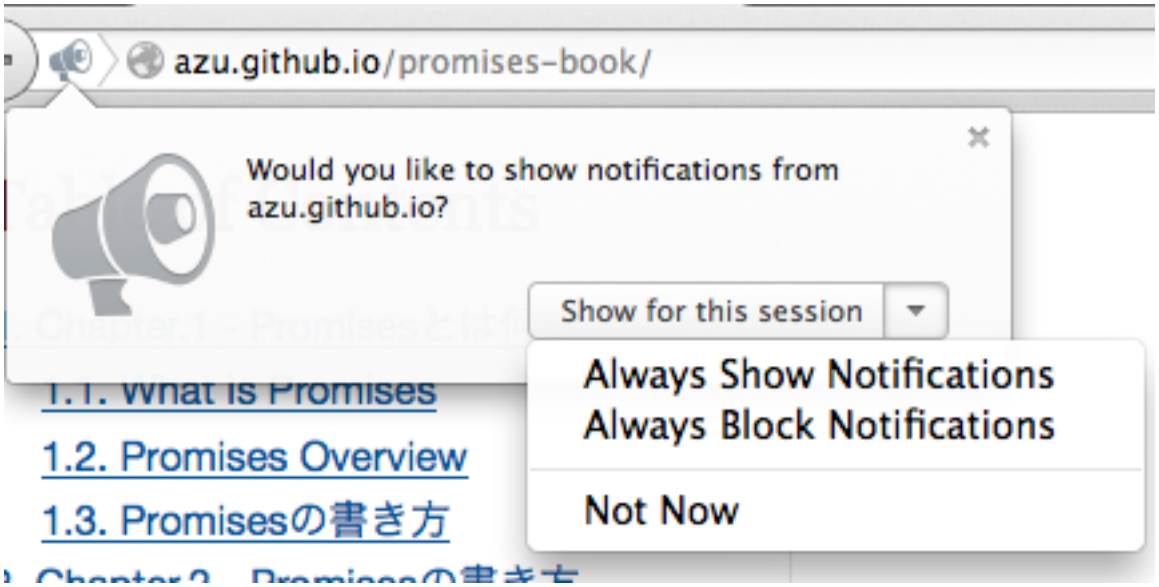


Figure 11. Notificationの許可ダイアログ

この許可ダイアログで選択した結果は、`Notification.permission` に入りますが、値は許可("granted")か不許可("denied")の2種類です。

Notificationのダイアログの選択肢は、Firefoxだと許可、不許可に加えて 永続 か セッション限り の組み合わせがありますが、値自体は同じです。

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結果がコールバック関数の `status` に渡されます。

コールバック関数を受け付けることから分かるように、この許可、不許可は非同期的に行われます。

```
Notification.requestPermission((status) => {
    // statusに"granted" or "denied"が入る
    console.log(status);
});
```

通知を行うまでの流れを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
    - すで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

いくつかのパターンが出ますが、最終的には許可か不許可になるので、以下の2パターンにまとめることができます。

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

不許可時("denied")
　何もしない

この2パターンはどこかで見たことがありますね。そう、PromiseのFulfilled またはRejected となった時の動作で書くことが出来そうな気がします。

resolve(成功)した時 == 許可時("granted")
　`onFulfilled` が呼ばれる

reject(失敗)した時 == 不許可時("denied")
　`onRejected` が呼ばれる

Promiseで書けそうな目処が見えた所で、まずはコールバックスタイルで書いてみましょう。

Web Notification ラッパー

まずは先ほどのWeb Notification APIのラッパー関数をコールバックスタイルで書くと次のように書くことができます。

notification-callback.js

```
function notifyMessage(message, options, callback) {
    if (typeof Notification === "undefined") {
        callback(new Error("doesn't support Notification API"));
        return;
    }
    if (Notification.permission === "granted") {
        const notification = new Notification(message, options);
        callback(null, notification);
    } else {
        Notification.requestPermission((status) => {
            if (Notification.permission !== status) {
                Notification.permission = status;
            }
            if (status === "granted") {
                const notification = new Notification(message, options);
                callback(null, notification);
```

```
            } else {
                callback(new Error("user denied"));
            }
        });
    }
}

// 実行例
// 第二引数は `Notification` に渡すオプションオブジェクト
notifyMessage("Hi!", {}, function (error, notification) {
    if(error){
        console.error(error);
        return;
    }
    console.log(notification);// 通知のオブジェクト
});
```

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。

コールバック関数はエラーとnotificationオブジェクトを受け取る

```
function callback(error, notification) {

}
```

次に、このコールバックスタイルの関数をPromiseとして使える関数を書いてみたいと思います。

Notifications API[66]の最新仕様では、コールバック関数を渡さなかった場合にpromiseオブジェクトを返すようになっています。そのため、ここから先の話は最新の仕様ではもっとシンプルに書ける可能性があります。

しかし、古いNotification APIの仕様では、コールバック関数のみしか扱う方法がありませんでした。ここではコールバック関数のみしか扱えないNotification APIを前提にしています。

Web Notification as Promise

先ほどのコールバックスタイルの `notifyMessage` とは別に、promiseオブジェクトを返す `notifyMessageAsPromise` を定義してみます。

[66] https://notifications.spec.whatwg.org/

notification-as-promise.js

```
function notifyMessage(message, options, callback) {
    if (typeof Notification === "undefined") {
        callback(new Error("doesn't support Notification API"));
        return;
    }
    if (Notification.permission === "granted") {
        const notification = new Notification(message, options);
        callback(null, notification);
    } else {
        Notification.requestPermission((status) => {
            if (Notification.permission !== status) {
                Notification.permission = status;
            }
            if (status === "granted") {
                const notification = new Notification(message, options);
                callback(null, notification);
            } else {
                callback(new Error("user denied"));
            }
        });
    }
}
function notifyMessageAsPromise(message, options) {
    return new Promise((resolve, reject) => {
        notifyMessage(message, options, (error, notification) => {
            if (error) {
                reject(error);
            } else {
                resolve(notification);
            }
        });
    });
}

// 実行例
notifyMessageAsPromise("Hi!").then(function (notification) {
    console.log(notification);// 通知のオブジェクト
}).catch((error) => {
    console.error(error);
});
```

上記の実行例では、許可がある場合 `"Hi!"` という通知が表示されます。

許可されている場合は `.then` が呼ばれ、不許可となった場合は `.catch` が呼ばれます。

ブラウザはWeb Notifications APIの状態をサイトごとに許可状態を記憶できるため、実際には以下の4つのパターンが存在します。

すでに許可されている
`.then` が呼ばれる

許可ダイアログがでて許可された
`.then` が呼ばれる

すでに不許可となっている
`.catch` が呼ばれる

許可ダイアログが出て不許可となった
`.catch` が呼ばれる

つまり、Web Notifications APIをそのまま扱うと、4つのパターンについて書かないといけませんが、それを2パターンにできるラッパーを書くと扱いやすくなります。

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は Promise をサポートしてない環境では使えないという問題があります。

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選択肢があると思います。

Promiseが使える環境を前提とする

- 利用者に `Promise` があることを保証してもらう
- Promiseをサポートしてない環境では動かないことにする

ライブラリ自体に `Promise` の実装を入れてしまう

- ライブラリ自体にPromiseの実装を取り込む
- 例) localForage[67]

コールバックでも `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できるようにする
- Thenableを返せるようにする

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 `Promise` でも使えるようにするということを 実現するのに役立つ概念です。

[67] https://github.com/mozilla/localForage

Web Notifications As Thenable

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。
次はnotification-callback.jsに `thenable` を返すメソッドを追加してみましょう。

notification-thenable.js

```
function notifyMessage(message, options, callback) {
    if (typeof Notification === "undefined") {
        callback(new Error("doesn't support Notification API"));
        return;
    }
    if (Notification.permission === "granted") {
        const notification = new Notification(message, options);
        callback(null, notification);
    } else {
        Notification.requestPermission((status) => {
            if (Notification.permission !== status) {
                Notification.permission = status;
            }
            if (status === "granted") {
                const notification = new Notification(message, options);
                callback(null, notification);
            } else {
                callback(new Error("user denied"));
            }
        });
    }
}
// `thenable` を返す
function notifyMessageAsThenable(message, options) {
    return {
        "then": function(resolve, reject) {
            notifyMessage(message, options, (error, notification) => {
                if (error) {
                    reject(error);
                } else {
                    resolve(notification);
                }
            });
        }
    };
}
// 実行例
Promise.resolve(notifyMessageAsThenable("message")).then(function (notification) {
    console.log(notification);// 通知のオブジェクト
```

```
}).catch((error) => {
    console.error(error);
});
```

notification-thenable.js には `notifyMessageAsThenable` 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 `Promise.resolve(thenable)` を使いpromiseオブジェクトにしてから、Promiseとして利用していることが分かりますね。

```
Promise.resolve(notifyMessageAsThenable("message")).then((notification) => {
    console.log(notification);// 通知のオブジェクト
}).catch((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と比べた時に、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

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スタイルの中間的な実装をすることができました。

まとめ

このセクションではThenableとは何かやThenableを `Promise.resolve(thenable)` を使って、promiseオブジェクトとして利用する方法について学びました。

Callback — Thenable — Promise

Thenableスタイルは、コールバックスタイルとPromiseスタイルの中間的な表現で、ライブラリが公開するAPIとしては中途半端なためあまり見かけることがないと思います。

Thenable自体は `Promise` という機能に依存してはいませんが、Promise以外からの利用方法は特にないため、間接的にはPromiseに依存しています。

また、使うためには利用者が `Promise.resolve(thenable)` について理解している必要があるため、ライブラリの公開APIとしては難しい部分があります。Thenable自体は公開APIより、内部的に使われてるケースが多いでしょう。

> 非同期処理を行うライブラリを書く際には、まずはコールバックスタイルの関数を書いて公開APIとすることをオススメします。
>
> Node.jsのCore moduleがこの方法をとっているように、ライブラリが提供するのは基本となるコールバックスタイル関数としたほうが、利用者がPromiseやGenerator等の好きな方法で実装ができるためです。
>
> 最初からPromiseで利用することを目的としたライブラリや、その機能がPromiseに依存している場合は、promiseオブジェクトを返す関数を公開APIとしても問題ないと思います。

Thenableの使われているところ

では、どのような場面でThenableは使われてるのでしょうか?

恐らく、一番多く使われている所はPromiseのライブラリ間での相互変換でしょう。

たとえば、QライブラリのPromiseのインスタンスであるQ promiseオブジェクトは、ES Promisesのpromiseオブジェクトが持っていないメソッドを持っています。Q promiseオブジェクトには `promise.finally(callback)` や `promise.nodeify(callback)` などのメソッドが用意されてます。

ES PromisesのpromiseオブジェクトをQ promiseオブジェクトに変換するときに使われるのが、まさにこのThenableです。

thenableを使ってQ promiseオブジェクトにする

```
const Q = require("Q");
// このpromiseオブジェクトはESのもの
const promise = new Promise((resolve) => {
    resolve(1);
});
// Q promiseオブジェクトに変換する
Q(promise).then((value) => {
    console.log(value);
```

```
}).finally(() => { ❶
    console.log("finally");
});
```

❶　Q promiseオブジェクトとなったため `finally` が利用できる

最初に作成したpromiseオブジェクトは `then` というメソッドを持っているので、もちろんThenableです。`Q(thenable)` とすることでThenableなオブジェクトをQ promiseオブジェクトへと変換することができます。

これは、`Promise.resolve(thenable)` と同じ仕組みといえるので、もちろん逆も可能です。

このように、Promiseライブラリはそれぞれ独自に拡張したpromiseオブジェクトを持っていますが、Thenableという共通の概念を使うことでライブラリ間(もちろんネイティブPromiseも含めて)で相互にpromiseオブジェクトを変換することができます。

このようにThenableが使われる所の多くはライブラリ内部の実装であるため、あまり目にする機会はないかもしれません。しかしこのThenableはPromiseでも大事な概念であるため知っておくとよいでしょう。

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` 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。

```
const promise = new Promise((resolve, reject) => {
    throw new Error("message");
});
promise.catch((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できます。

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

```
    console.error(error);// => "message"
});
```

`throw` が `reject` に変わったと考えれば、`reject` にはErrorオブジェクトを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。

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、なぜ `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。

たとえば、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。

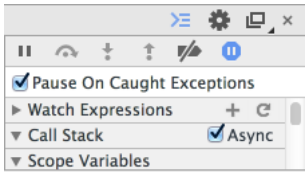


Figure 12. Pause On Caught Exceptions

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。

```
const promise = new Promise((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。

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?

```
const promise = Promise.resolve();
promise.then((value) => {
    setTimeout(() => {
        // 一定時間経って終わらなかったらrejectしたい - 2
```

```
    }, 1000);
    // 時間がかかる処理 - 1
    somethingHardWork();
}).catch((error) => {
    // タイムアウトエラー - 3
});
```

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。

> 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。このときreturnした値が、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ことができます。

このとき、returnしたものがpromiseオブジェクトである場合、そのpromiseオブジェクトの状態によって、次の `then` に登録されたonFulfilledとonRejectedのうち、どちらが呼ばれるかを決めることができます。

```
const promise = Promise.resolve();
promise.then(() => {
    const retPromise = new Promise((resolve, reject) => {
        // resolve or reject で onFulfilled or onRejected どちらを呼ぶか決まる
    });
    return retPromise;❶
}).then(onFulfilled, onRejected);
```

❶ 次に呼び出されるthenのコールバックはpromiseオブジェクトの状態によって決定される

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することができます。

```
const onRejected = console.error.bind(console);
const promise = Promise.resolve();
promise.then(() => {
    const retPromise = new Promise((resolve, reject) => {
```

```
        reject(new Error("this promise is rejected"));
    });
    return retPromise;
}).catch(onRejected);
```

これは、the section called “Promise.reject” を使うことでもっと簡潔に書くことができます。

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

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ことが多いので、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。

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。

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ことがあるかもしれません。有名な所だと jQuery.Deferred[68] や JSDeferred[69] 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[70] のようなDeferredの実装を中心にして話を進めます。

[68] http://api.jquery.com/category/deferred-object/
[69] http://cho45.stfuawsc.com/jsdeferred/
[70] http://api.jquery.com/category/deferred-object/

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

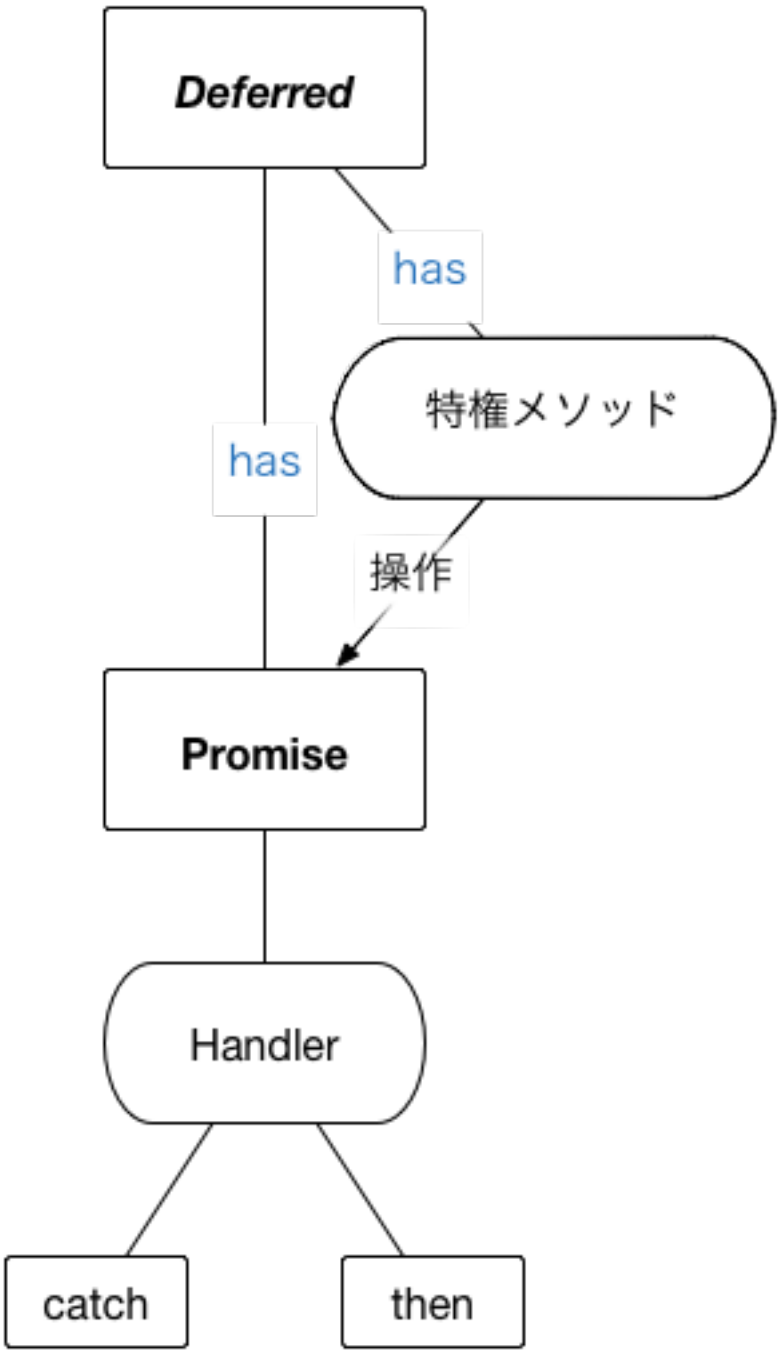


Figure 13. DeferredとPromise

この図を見ると分かりますが、DeferredとPromiseは比べるような関係ではなく、Deferredが Promiseを内蔵しているような関係になっていることが分かります。

jQuery.Deferredの構造を簡略化したものです。Promiseを使わない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クラスを実装してみましょう。

ECMAScript 2015ではクラスを定義する `class` 構文が導入されています。`class` 構文を使ったクラス定義については、次のページを参照してください。

- クラス - JavaScript | MDN[71]
- クラス・JavaScript Primer #jsprimer[72]

Deferred top on Promise

Promiseの上にDeferredクラスを実装した例です。

deferred.js

```
class Deferred {
    constructor() {
        this.promise = new Promise((resolve, reject) => {
            // Arrow Functionを利用しているため、`this`がDeferredのインスタンスを参照する
            this._resolve = resolve;
            this._reject = reject;
        });
    }

    // Deferred#resolveメソッドは、`value`でPromiseインスタンスをresolveする
    resolve(value) {
        this._resolve(value);
    }

    // Deferred#rejectメソッドは、`reason`でPromiseインスタンスをrejectする
    reject(reason) {
        this._reject(reason);
    }
}
```

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

[71] https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Classes
[72] https://jsprimer.net/basic/class/

xhr-deferred.js

```
class Deferred {
    constructor() {
        this.promise = new Promise((resolve, reject) => {
            // Arrow Functionを利用しているため、`this`がDeferredのインスタンスを参照する
            this._resolve = resolve;
            this._reject = reject;
        });
    }

    // Deferred#resolveメソッドは、`value`でPromiseインスタンスをresolveする
    resolve(value) {
        this._resolve(value);
    }

    // Deferred#rejectメソッドは、`reason`でPromiseインスタンスをrejectする
    reject(reason) {
        this._reject(reason);
    }
}
function fetchURL(URL) {
    const deferred = new Deferred();
    const req = new XMLHttpRequest();
    req.open("GET", URL, true);
    req.onload = () => {
        if (200 <= req.status && req.status < 300) {
            deferred.resolve(req.responseText);
        } else {
            deferred.reject(new Error(req.statusText));
        }
    };
    req.onerror = () => {
        deferred.reject(new Error(req.statusText));
    };
    req.send();
    return deferred.promise;
}

// 実行例
const URL = "https://httpbin.org/get";
fetchURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(console.error.bind(console));
```

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ことができません。

通常のPromiseで実装したものと見比べていきたいと思います。

xhr-promise.js

```
function fetchURL(URL) {
    return new Promise((resolve, reject) => {
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}

// 実行例
const URL = "https://httpbin.org/get";
fetchURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(console.error.bind(console));
```

2つの `fetchURL` を見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。

- 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
    - Promiseコンストラクタの中で処理が行われていないため、自動的に例外をキャッチしない

逆に以下の部分は同じことをやっています。

- 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。

たとえば、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((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。

```
const 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(Fulfilled)か異常な値(Rejected)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こと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以下を参照するといいでしょう。

- Promise & Deferred objects in JavaScript Pt.1: Theory and Semantics.[73]
- Twisted 入門 — Twisted Intro[74]
- Promise anti patterns · petkaantonov/bluebird Wiki[75]
- Coming from jQuery · kriskowal/q Wiki[76]

[73] http://blog.mediumequalsmessage.com/promise-deferred-objects-in-javascript-pt1-theory-and-semantics
[74] http://skitazaki.appspot.com/translation/twisted-intro-ja/index.html
[75] https://github.com/petkaantonov/bluebird/wiki/Promise-anti-patterns#the-deferred-anti-pattern
[76] https://github.com/kriskowal/q/wiki/Coming-from-jQuery

DeferredはPythonの Twisted[77] 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[78] 、dojo/Deferred[79] 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。

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[80] プロパティを持っているので、これを利用すると簡単にできますが、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。

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いきたいと思います。

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、`setTimeout` 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。

delayPromise.js

```
function delayPromise(ms) {
    return new Promise((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(() => {
    alert("100ms 経ったよ!");
}, 100);
// == ほぼ同様の動作
```

77 https://twistedmatrix.com/trac/
78 http://mochi.github.io/mochikit/doc/html/MochiKit/Async.html
79 http://dojotoolkit.org/reference-guide/1.9/dojo/Deferred.html
80 https://developer.mozilla.org/ja/docs/XMLHttpRequest/Synchronous_and_Asynchronous_Requests

```
delayPromise(100).then(() => {
    alert("100ms 経ったよ!");
});
```

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

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。

```
const winnerPromise = new Promise((resolve) => {
    setTimeout(() => {
        console.log("this is winner");
        resolve("this is winner");
    }, 4);
});
const loserPromise = new Promise((resolve) => {
    setTimeout(() => {
        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((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できます。

simple-timeout-promise.js

```
function delayPromise(ms) {
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}
function timeoutPromise(promise, ms) {
    const timeout = delayPromise(ms).then(() => {
        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できるようになります。

```
function delayPromise(ms) {
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}
function timeoutPromise(promise, ms) {
    const timeout = delayPromise(ms).then(() => {
        throw new Error("Operation timed out after " + ms + " ms");
    });
    return Promise.race([promise, timeout]);
}

// 実行例
var taskPromise = new Promise(function(resolve){
    // 何らかの処理
    var delay = Math.random() * 2000;
    setTimeout(function(){
        resolve(delay + "ms");
    }, delay);
});
timeoutPromise(taskPromise, 1000).then(function(value){
    console.log("taskPromiseが時間内に終わった : " + value);
}).catch((error) => {
    console.log("タイムアウトになってしまった", error);
});
```

タイムアウトになった場合はエラーが呼ばれるようにできましたが、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。

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ことは不可能ですが(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します。

ECMAScript 2015から `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できます。

```
class MyError extends Error {
    // Errorを継承したオブジェクト
}
```

`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。

TimeoutError.js

```
function copyOwnFrom(target, source) {
    Object.getOwnPropertyNames(source).forEach((propName) => {
        Object.defineProperty(target, propName,
            Object.getOwnPropertyDescriptor(source, propName));
    });
    return target;
}
function TimeoutError() {
    const 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。

```
const promise = new Promise(() => {
    throw new TimeoutError("timeout");
});

promise.catch((error) => {
    console.log(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。

> 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[81] で詳しく紹介されています。また、Error - JavaScript | MDN[82] にもErrorオブジェクトについて書かれています。

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 `fetchURL` 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もつオブジェクトを返すようにしています。

delay-race-cancel.js

```
function cancelableXHR(URL) {
    const req = new XMLHttpRequest();
    const promise = new Promise((resolve, reject) => {
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.onabort = function() {
            reject(new Error("this request is aborted"));
        };
        req.send();
    });
    const abort = function() {
```

[81] http://speakingjs.com/es5/ch28.html
[82] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

```
        // 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
    };
}
```

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が時間内に取得できた場合
        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 new 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。

delay-race-cancel-play.js

```
function copyOwnFrom(target, source) {
    Object.getOwnPropertyNames(source).forEach((propName) => {
        Object.defineProperty(target, propName,
            Object.getOwnPropertyDescriptor(source, propName));
    });
    return target;
}
function TimeoutError() {
    const superInstance = Error.apply(null, arguments);
    copyOwnFrom(this, superInstance);
```

```
}
TimeoutError.prototype = Object.create(Error.prototype);
TimeoutError.prototype.constructor = TimeoutError;
function delayPromise(ms) {
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}
function timeoutPromise(promise, ms) {
    const timeout = delayPromise(ms).then(() => {
        return Promise.reject(new TimeoutError("Operation timed out after " + ms + "
 ms"));
    });
    return Promise.race([promise, timeout]);
}

function cancelableXHR(URL) {
    const req = new XMLHttpRequest();
    const promise = new Promise((resolve, reject) => {
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.onabort = function() {
            reject(new Error("this request is aborted"));
        };
        req.send();
    });
    const 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
```

```
    };
}
const object = cancelableXHR("https://httpbin.org/get");
// main
timeoutPromise(object.promise, 1000)
    .then((contents) => {
        console.log("Contents", contents);
    }).
    catch((error) => {
        if (error instanceof TimeoutError) {
            object.abort();
            console.error(error);
            return;
        }
        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。

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。

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に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` でき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、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,ES 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。

cancelableXHR.js

```
"use strict";
const requestMap = {};
function createXHRPromise(URL) {
    const req = new XMLHttpRequest();
    const promise = new Promise((resolve, reject) => {
        req.open("GET", URL, true);
        req.onreadystatechange = function() {
            if (req.readyState === XMLHttpRequest.DONE) {
                delete requestMap[URL];
            }
        };
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.onabort = () => {
            reject(new Error("abort this req"));
        };
        req.send();
    });
    requestMap[URL] = {
        promise: promise,
        request: req
    };
    return promise;
}

function abortPromise(promise) {
    if (typeof promise === "undefined") {
        return;
    }
    let request;
    Object.keys(requestMap).some((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。

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

const xhrPromise = cancelableXHR.createXHRPromise("https://httpbin.org/get");❶
xhrPromise.catch((error) => {
    // abort されたエラーが呼ばれる
});
cancelableXHR.abortPromise(xhrPromise);❷
```

❶ XHRをラップしたpromiseオブジェクトを作成
❷ 1で作成したpromiseオブジェクトのリクエストをキャンセル

まとめ

ここでは以下のこと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ことをより意識していいのかもしれません。

Fetch APIでのキャンセル

XHRの現代的なバージョンである Fetch API[83] では、AbortController[84] というAPIによってリクエストをキャンセルを実現できます。

Fetch APIでは、次のようにリクエストをキャンセルできます。

AbortControllerでのFetchのキャンセル

```
// AbortControllerのインスタンスの作成
const controller = new AbortController()
// キャンセルを通知するための siganl を取得する
const signal = controller.signal
// signal をfetchメソッドの第二引数に渡す
fetch("https://httpbin.org/get", { signal })
.then((result) => {
    // 結果の正常処理
    console.log(result);
})
.catch((error) => {
    if (error.name == "AbortError") {
        // 中断の場合の処理
        console.error("Fetchが中断されました", error);
        return;
    }
    // 中断以外のエラー
    console.error(err);
})
// Fetchをキャンセルする
controller.abort();
```

`AbortController` という今回実装したものと似たような操作メソッドをもつオブジェクトを利用することがわかります。

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。

83 https://developer.mozilla.org/ja/docs/Web/API/Fetch_API
84 https://developer.mozilla.org/en-US/docs/Web/API/AbortController

`Promise.prototype.done` は、ES 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？またなぜ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。

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。

promise-done-example.js

```
if (typeof Promise.prototype.done === "undefined") {
    Promise.prototype.done = function(onFulfilled, onRejected) {
        this.then(onFulfilled, onRejected).catch((error) => {
            setTimeout(() => {
                throw error;
            }, 0);
        });
    };
}
const promise = Promise.resolve();
promise.done(() => {
    JSON.parse("this is not json");
    // => SyntaxError: JSON.parse
});
// => ブラウザの開発ツールのコンソールを開いてみましょう
```

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。

thenを使った場合

```
const promise = Promise.resolve();
promise.then(() => {
    JSON.parse("this is not json");
}).catch((error) => {
    console.error(error);// => "SyntaxError: JSON.parse"
});
```

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の最後におくメソッドというのは分かると思います。

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。

なぜこのような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ているかについては 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。

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。

json-promise.js

```
function JSONPromise(value) {
    return new Promise((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` することができます。

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

```
}

// 実行例
var string = "jsonではない文字列";
JSONPromise(string).then(function (object) {
    console.log(object);
}).catch((error) => {
    // => JSON.parseで例外が発生した時
    console.error(error);
});
```

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。

catchによるエラーハンドリングを忘れてしまった場合

```
const string = "jsonではない文字列";
JSONPromise(string).then((object) => {
    console.log(object);
}); ❶
```

❶　例外が投げられても何も処理されない

`JSON.parse` のような分かりやすい例の場合はまだよいですが、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。

typoによるエラー

```
const string = "{}";
JSONPromise(string).then((object) => {
    conosle.log(object);❶
});
```

❶　conosleというtypoがある

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起きてしまいます。毎回、正しく `catch` の処理を書くことができ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。

このunhandled rejectionが検知しにくい問題はPromiseの実装と実行環境に依存します。

たとえば、Bluebird[85] では、明らかに人間のミスにみえるReferenceErrorの場合などをコンソールにエラーとして表示してくれます。

> "Possibly unhandled ReferenceError. conosle is not defined

また、このunhandled rejectionに関する仕組みが ECMAScript 2016[86] で仕様に追加されています。そのためネイティブPromiseでは、この仕様を活用したGC-based unhandled rejection trackingというものが搭載されているケースが増えています。

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になっています。

Firefox[87] や Chrome[88] のネイティブPromiseでは一部実装されています。

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できるので、`Promise.prototype.done` というPromiseのprototype拡張として実装してみましょう。

promise-prototype-done.js

```
"use strict";
if (typeof Promise.prototype.done === "undefined") {
```

85 https://github.com/petkaantonov/bluebird
86 https://github.com/tc39/ecma262/releases/tag/es2016-draft-20151201
87 https://twitter.com/domenic/status/461154989856264192
88 https://code.google.com/p/v8/issues/detail?id=3093

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

どのようにPromiseの外へ例外を投げているかというと、setTimeoutの中でthrowをすることで、外へそのまま例外を投げられることを利用しています。

setTimeoutのコールバック内での例外

```
try {
    setTimeout(() => {
        throw new Error("error");❶
    }, 0);
} catch (error) {
    console.error(error);
}
```

❶ この例外はキャッチされない

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になります。

- JavaScriptと非同期のエラー処理 - Yahoo! JAPAN Tech Blog[89]

`Promise.prototype.done` をよく見てみると、何も `return` していないことも分かると思います。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。

現在では多くの実行環境で、unhandled rejectionを検知してコンソールに警告を表示するため、`done` が必要な場合は少なくなっています。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るため、ES Promisesの仕様そのものには入らなかったといえるかもしれません。

今回の `Promise.prototype.done` の実装は promisejs.org[90] を参考にしています。

89 http://techblog.yahoo.co.jp/programming/javascript_error/
90 https://www.promisejs.org/

まとめ

このセクションでは、Q[91] や Bluebird[92] や prfun[93] 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` には次の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で問題となるケースは少なくなっています。

また、`done` は値を返さないことでそれ以上Promise chainを繋げること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` を使うこともできます。

ES 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。

そのとき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いえるかもしれません。

> Promises: The Extension Problem (part 4) | getiblog[94] では、Promiseの拡張を書く手法について書かれています。
>
> - `Promise.prototype` を拡張する方法
> - Wrapper/Delegate を使った抽象レイヤーを作る方法
>
> また、Delegateを利用した方法については、Chapter 28. Subclassing Built-ins[95] にて 詳しく解説されています。

[91] https://github.com/kriskowal/q/wiki/API-Reference#promisedoneonfulfilled-onrejected-onprogress
[92] https://github.com/petkaantonov/bluebird
[93] https://github.com/cscott/prfun#promisedone—undefined
[94] http://blog.getify.com/promises-part-4/
[95] http://speakingjs.com/es5/ch28.html

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。

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[96] 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。

fsのメソッドチェーン

以下のような Node.jsのfs[97] 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いえないかもしれません。

fs-method-chain.js

```
"use strict";
const fs = require("fs");
function File() {
    this.lastValue = null;
}
// Static method for File.prototype.read
File.read = function FileRead(filePath) {
    const 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) {
```

96 http://taiju.hatenablog.com/entry/20100307/1267962826
97 http://nodejs.org/api/fs.html

```
    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。

```
const File = require("./fs-method-chain");
const inputFilePath = "input.txt";
const outputFilePath = "output.txt";
File.read(inputFilePath)
    .transform((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。

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。

fs-promise-chain.js

```
"use strict";
const fs = require("fs");
function File() {
    this.promise = Promise.resolve();
}
// Static method for File.prototype.read
File.read = function(filePath) {
    const 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(() => {
        return fs.readFileSync(filePath, "utf-8");
    });
};
File.prototype.transform = function(fn) {
    return this.then(fn);
};
File.prototype.write = function(filePath) {
    return this.then((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。

```
const File = require("./fs-promise-chain");
const inputFilePath = "input.txt";
const outputFilePath = "output.txt";
File.read(inputFilePath)
    .transform((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。

```
const File = require("./fs-promise-chain");
File.read(inputFilePath)
    .transform((content) => {
        return ">>" + content;
```

```
    })
    .write(outputFilePath);
// => 擬似的に以下のような流れに展開できる
promise.then(() => {
    return fs.readFileSync(filePath, "utf-8");
}).then((content) => {
    return ">>" + content;
}).then(() => {
    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、自分で逐次的に処理する機構を実装しなくても問題ないわけです。

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できますが、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` で囲むことで行えます。

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。

fs-promise-chainでのエラーハンドリング

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

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

fs-method-chain.jsに非同期処理を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いえるかもしれません。

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[98] が思い浮かぶと思います。

Stream[99] を使うと、`this.lastValue` のような値を保持する必要がなくなることや大きなファイルの扱いが改善されます。また、Promiseを使った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。

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[100]
- Stream2の基本[101]
- Node-v0.12の新機能について[102]

98 http://nodejs.org/api/stream.html
99 http://nodejs.org/api/stream.html
100 http://jxck.hatenablog.com/entry/20111204/1322966453
101 http://www.slideshare.net/shigeki_ohtsu/stream2-kihon
102 http://www.slideshare.net/shigeki_ohtsu/node-v012tng12

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ES 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[103] などには Promisification[104] という機能が用意されています。また、Node.jsのコアモジュールである `util` モジュールには、util.promisify[105] というAPI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版のメソッドを作成して利用できるようになります。

```
const fs = require("fs");
const util = require("util");
// コールバック版のAPIからPromise版を作成する
const readFile = util.promisify(fs.readFile);

readFile("myfile.js", "utf8").then((contents) => {
    console.log(contents);
}).catch((e) => {
    console.error(e.stack);
});
```

ArrayのPromiseラッパー

先ほどの `util.promisify` 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版とな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` 等のメソッドは配列を返すため、メソッドチェーンが利用しやすい機能です

array-promise-chain.js

103 https://github.com/petkaantonov/bluebird/
104 https://github.com/petkaantonov/bluebird/blob/master/API.md#promisification
105 https://nodejs.org/api/util.html#util_util_promisify_original

```
"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((methodName) => {
    // Don't overwrite
    if (typeof ArrayAsPromise[methodName] !== "undefined") {
        return;
    }
    const arrayMethod = Array.prototype[methodName];
    if (typeof  arrayMethod !== "function") {
        return;
    }
    ArrayAsPromise.prototype[methodName] = function() {
        const that = this;
        const args = arguments;
        this.promise = this.promise.then(() => {
            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。

array-promise-chain-test.js

```
const assert = require("assert");
const ArrayAsPromise = require("../src/promise-chain/array-promise-chain");
```

```
describe("array-promise-chain", () => {
    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", () => {
        it("can method chain", function() {
            const result = this.array.filter(isEven).map(double);
            assert.deepEqual(result, [4, 8]);
        });
    });
    describe("ArrayAsPromise", () => {
        it("can promise chain", function(done) {
            const array = new ArrayAsPromise(this.array);
            array.filter(isEven).map(double).then((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` の実装を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。

先ほどの util.promisify[106] 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

[106] https://nodejs.org/api/util.html#util_util_promisify_original

> `error` が来るというルール（エラーファーストコールバック）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。
>
> また、Node.js 10からは `fs` モジュールに Promise版のAPI[107] が追加されています。

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

ES 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こと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こと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。

Promiseによる逐次処理

第2章のPromise.allでは、複数のpromiseオブジェクトをまとめて処理する方法について学びました。

しかし、`Promise.all` は全ての処理を並行に行うため、Aの処理 が終わったら Bの処理 というような逐次的な処理を扱うことができません。

また、同じ2章のPromiseと配列では、効率的ではないですが、thenを連ねた書き方でそのような逐次処理を行っていました。

このセクションでは、Promiseを使った逐次処理の書き方について学んで行きたいと思います。

[107] https://nodejs.org/api/fs.html#fs_fs_promises_api

ループと逐次処理

thenを連ねた書き方では以下のような書き方でしたね。

```
function fetchURL(URL) {
    return new Promise((resolve, reject) => {
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
const request = {
    comment() {
        return fetchURL("https://azu.github.io/promises-book/json/
comment.json").then(JSON.parse);
    },
    people() {
        return fetchURL("https://azu.github.io/promises-book/json/
people.json").then(JSON.parse);
    }
};
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] は記録する初期値を部分適用している
    const pushValue = recordValue.bind(null, []);
    return request.comment()
        .then(pushValue)
        .then(request.people)
        .then(pushValue);
}
```

```
// 実行例
main().then((value) => {
    console.log(value);
}).catch((error) => {
    console.error(error);
});
```

この書き方だと、`request` の数が増える分 `then` を書かないといけなくなってしまいます。

そこで、処理を配列にまとめて、forループで処理していければ、数が増えた場合も問題無いですね。まずはforループを使って先ほどと同じ処理を書いてみたいと思います。

promise-foreach-xhr.js

```
function fetchURL(URL) {
    return new Promise((resolve, reject) => {
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
const request = {
    comment() {
        return fetchURL("https://azu.github.io/promises-book/json/
comment.json").then(JSON.parse);
    },
    people() {
        return fetchURL("https://azu.github.io/promises-book/json/
people.json").then(JSON.parse);
    }
};
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
```

```
    }

    // [] は記録する初期値を部分適用してる
    const pushValue = recordValue.bind(null, []);
    // promiseオブジェクトを返す関数の配列
    const tasks = [request.comment, request.people];
    let promise = Promise.resolve();// スタート地点
    for (let i = 0; i < tasks.length; i++) {
        const task = tasks[i];
        promise = promise.then(task).then(pushValue);
    }
    return promise;
}



// 実行例
main().then((value) => {
    console.log(value);
}).catch((error) => {
    console.error(error);
});
```

forループで書く場合、コラム: thenは常に新しいpromiseオブジェクトを返すやPromiseとメソッドチェーンで学んだように、Promise#then は新しいpromiseオブジェクトを返しています。

そのため、`promise = promise.then(task).then(pushValue);` というのは `promise` という変数に上書きするというよりは、そのpromiseオブジェクトに処理を追加していくような処理になっています。

しかし、この書き方だと一時変数として `promise` が必要で、処理の内容的にもあまりスッキリしません。

このループの書き方は `Array.prototype.reduce` を使うともっとスマートに書くことができます。

Promise chainとreduce

`Array.prototype.reduce` を使って書き直すと以下のようになります。

promise-reduce-xhr.js

```
function fetchURL(URL) {
    return new Promise((resolve, reject) => {
```

```
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
const request = {
    comment() {
        return fetchURL("https://azu.github.io/promises-book/json/
comment.json").then(JSON.parse);
    },
    people() {
        return fetchURL("https://azu.github.io/promises-book/json/
people.json").then(JSON.parse);
    }
};
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }

    const pushValue = recordValue.bind(null, []);
    const tasks = [request.comment, request.people];
    return tasks.reduce((promise, task) => {
        return promise.then(task).then(pushValue);
    }, Promise.resolve());
}



// 実行例
main().then((value) => {
    console.log(value);
}).catch((error) => {
    console.error(error);
});
```

`main` 以外の処理はforループのものと同様です。

`Array.prototype.reduce` は第二引数に初期値を入れることができます。つまりこの場合、最初の `promise` には `Promise.resolve()` が入り、そのときの `task` は `request.comment` となります。

reduceの中で `return` したものが、次のループで `promise` に入ります。つまり、`then` を使って作成した新たなpromiseオブジェクトを返すことで、forループの場合と同じようにPromise chainを繋げることができます。

`Array.prototype.reduce` については詳しくは以下を参照して下さい。

- Array.prototype.reduce() - JavaScript | MDN[108]
- Array.prototype.reduce Dance[109]

forループと異なる点は、一時変数としての `promise` が不要になることに伴い、`promise = promise.then(task).then(pushValue);` という不格好な書き方がなくなる点が大きな違いだと思います。

`Array.prototype.reduce` とPromiseの逐次処理は相性がよいので覚えておくといいのですが、初めて見た時にどのような動作をするのかがまだ分かりにくいという問題があります。

そこで、処理するTaskとなる関数の配列を受け取って逐次処理を行う `sequenceTasks` というものを作ってみます。

以下のように書くことができれば、`tasks` が順番に処理されていくことが関数名から見て分かるようになります。

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

逐次処理を行う関数を定義する

基本的には、reduceを使ったやり方を関数として切り離せばいいだけですね。

promise-sequence.js

```
function sequenceTasks(tasks) {
```

[108] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce
[109] https://azu.github.io/slide/JSGohan/reduce.html

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

    const pushValue = recordValue.bind(null, []);
    return tasks.reduce((promise, task) => {
        return promise.then(task).then(pushValue);
    }, Promise.resolve());
}
```

一つ注意点として、`Promise.all` 等と違い、引数に受け取るのは関数の配列です。

なぜ、渡すのがpromiseオブジェクトの配列ではないのかというと、promiseオブジェクトを作った段階ですでにXHRが実行されている状態なので、それを逐次処理しても意図とは異なる動作になるためです。

そのため `sequenceTasks` では関数(promiseオブジェクトを返す)の配列を引数に受け取ります。

最後に、`sequenceTasks` を使って最初の例を書き換えると以下のようになります。

promise-sequence-xhr.js

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

    const pushValue = recordValue.bind(null, []);
    return tasks.reduce((promise, task) => {
        return promise.then(task).then(pushValue);
    }, Promise.resolve());
}
function fetchURL(URL) {
    return new Promise((resolve, reject) => {
        const req = new XMLHttpRequest();
        req.open("GET", URL, true);
        req.onload = () => {
            if (200 <= req.status && req.status < 300) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
```

```
        req.onerror = () => {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
const request = {
    comment() {
        return fetchURL("https://azu.github.io/promises-book/json/
comment.json").then(JSON.parse);
    },
    people() {
        return fetchURL("https://azu.github.io/promises-book/json/
people.json").then(JSON.parse);
    }
};
function main() {
    return sequenceTasks([request.comment, request.people]);
}



// 実行例
main().then((value) => {
    console.log(value);
}).catch((error) => {
    console.error(error);
});
```

`main()` の中がかなりスッキリしたことが分かります。

このようにPromiseでは、逐次処理ということをするのに色々な書き方ができると思います。

- thenをその場に並べた書き方
- forループを使った書き方
- reduceを使った書き方
- 逐次処理する関数を分けた書き方

さらに、ここではまだ紹介していませんが、Async Functionを使う方法もあります。

しかし、これはJavaScriptで配列を扱うのにforループや `forEach` 等、色々やり方があるのと本質的には違いはありません。そのため、Promiseを扱う場合も処理をまとめられるところは小さく関数に分けて、実装していくのがいいといえるでしょう。

まとめ

このセクションでは、`Promise.all`とは違い、一つづつ順番に処理したい場合に、Promiseでどのように実装していくかについて学びました。

手続き的な書き方から、逐次処理を行う関数を定義するところまで見ていき、Promiseであっても関数に処理を分けるという基本的なことは変わらないことを示しました。

Promiseで書くとPromise chainを繋げすぎて縦に長い処理を書いてしまうことがあります。

そんな時は基本に振り返り、処理を関数に分けることで全体の見通しを良くすることは大切です。

また、Promiseのコンストラクタ関数や `then` 等は高階関数なので、処理を関数に分けておくと組み合わせが行い易いという副次的な効果もあるため、意識してみるといいかもしれません。

> 高階関数とは引数に関数オブジェクトを受け取る関数のこと

Chapter.5 - Async Function

この章では、ECMAScript 2017で導入されたAsync Function(`async` / `await`)について学んでいきます。

Async Functionとは

Async Functionとは非同期処理を行う関数を定義する構文です。Async Functionは通常の関数とは異なり、必ず `Promise` インスタンスを返す関数を定義する構文です。

Async Functionは次のように関数の前に `async` をつけることで定義できます。この `doAsync` 関数は常に `Promise` インスタンスを返します。

```
async function doAsync() {
    return "値";
}
// doAsync関数はPromiseを返す
doAsync().then((value) => {
    console.log(value); // => "値"
});
```

Async Functionでは `return` した値の代わりに、`Promise.resolve(返り値)` のように返り値をラップした `Promise` インスタンスを返します。そのため、このAsync Functionは次のように書いた場合と同じ意味になります。

```
// 通常の関数でPromiseインスタンスを返している
function doAsync() {
    return Promise.resolve("値");
}
doAsync().then((value) => {
    console.log(value); // => "値"
});
```

またAsync Function内では `await` 式というPromiseの非同期処理が完了するまで待つ構文が利用できます。`await` 式を使うことで非同期処理を同期処理のように扱えるため、Promiseチェーンで実現していた処理の流れを読みやすくかけます。

Async Functionと `await` 式の大まかな動きをイメージするために、まずはPromise APIで書いたものと比較してみます。

ここでは、XHRの現代的なバージョンである Fetch API[110] を使います。Fetch APIは指定したURLのリソースを読み書きでき、デフォルトでES Promisesに対応しています。

次のサンプルコードでは、`https://azu.github.io/promises-book/json/book.json` というURLからJSONデータを取得して、`title` プロパティを取り出す `getBookTitle` 関数を実装していきます。

取得する `https://azu.github.io/promises-book/json/book.json` は次のような内容になっています。

/json/book.json

```
{
    "title": "JavaScript Promiseの本",
    "repository": "https://github.com/azu/promises-book"
}
```

まずは、Fetch APIを使って `fetchBookTitle` 関数で取得したタイトルをコンソールに出力してみます。

`fetch` メソッドはPromiseを返します。このPromiseインスタンスはリクエストのレスポンスを表す `Response` オブジェクトでresolveされます。`Response#json` メソッドもPromiseを

[110] https://developer.mozilla.org/ja/docs/Web/API/Fetch_API

返します。このPromiseインスタンスは取得したリソースをJSONとしてパースしたオブジェクトでresolveされます。

`fetchBookTitle`関数は、次のように`fetch`メソッドで取得したJSONの`title`プロパティでresolveされるPromiseインスタンスを返します。

```
function fetchBookTitle() {
    // Fetch APIは指定URLのリソースを取得しPromiseを返す関数
    return fetch("https://azu.github.io/promises-book/json/book.json").then((res) => {
        return res.json(); // レスポンスをJSON形式としてパースする
    }).then((json) => {
        return json.title; // JSONからtitleプロパティを取り出す
    });
}

function main() {
    // `fetchBookTitle`関数は、取得したJSONの`title`プロパティでresolveされる
    fetchBookTitle().then((title) => {
        console.log(title); // => "JavaScript Promiseの本"
    });
}

main();
```

次は、同様の処理をAsync Functionと`await`式で実装してみます。ここではまだ挙動を理解しなくても問題ありませんが、Promise APIを使っていた場合に比べて、`then`メソッドやコールバック関数がなくなっていることが分かります。

```
// `async`をつけて`fetchBookTitle`関数をAsync Functionとして定義
async function fetchBookTitle() {
    // リクエストしてリソースを取得する
    const res = await fetch("https://azu.github.io/promises-book/json/book.json");
    // レスポンスをJSON形式としてパースする
    const json = await res.json();
    // JSONからtitleプロパティを取り出す
    return json.title;
}

// `async`をつけて`main`関数をAsync Functionとして定義
async function main() {
    // `await`式で`fetchBookTitle`の非同期処理が完了するまで待つ
    // `fetchBookTitle`がresolveした値が返り値になる
    const title = await fetchBookTitle();
    console.log(title); // => "JavaScript Promiseの本"
```

```
}

main();
```

Async FunctionではPromiseの状態が変化するまで待つ`await`式という機能を利用できます。Promiseでは結果を`then`メソッドのコールバック関数で取得していたのが、`await`式の右辺にあるPromiseのresolveされた値が左辺の変数へと代入されます。そのため、Async Functionと`await`式を使うことで非同期処理をまるで同期処理のように書けます。

この章では、このAsync Functionと`await`式について詳しく見ていきます。

重要なこととしてAsync FunctionはPromiseの上に作られた構文です。そのためAsync Functionを理解するには、Promiseを理解する必要があることに注意してください。

<async-function-syntax>
<title>Async Functionの構文</title>
Async Functionは関数の定義に`async`キーワードをつけることで定義できます。JavaScriptの関数定義には関数宣言や関数式、Arrow Function、メソッドの短縮記法などがあります。どの定義方法でも`async`キーワードを前につけるだけでAsync Functionとして定義できます。

```
// 関数宣言のAsync Function版
async function fn1() {}
// 関数式のAsync Function版
const fn2 = async function() {};
// Arrow FunctionのAsync Function版
const fn3 = async() => {};
// メソッドの短縮記法のAsync Function版
const object = {
    async method() {}
};
```

これらのAsync Functionは、次のこと以外は通常の関数と同じ性質を持ちます。

- Async Functionは必ずPromiseを返す
- Async Function内では`await`式が利用できる

Async FunctionはPromiseを返す

Async Functionとして定義した関数は必ず`Promise`インスタンスを返します。返される`Promise`インスタンスの状態は関数の返り値によって異なり、次の3つのケースが考えられます。

1. Async FunctionはPromise以外の値をreturnした場合、その返り値をもつFulfilledなPromiseを返す
2. Async FunctionがPromiseをreturnした場合、その返り値のPromiseをそのまま返す
3. Async Function内で例外が発生した場合は、そのエラーをもつRejectedなPromiseを返す

これらの挙動は`Promise#then`メソッドの返り値とそのコールバック関数が返す値の関係とほぼ同じです。

具体的な例を順番に見ていきます。

まず、Async FunctionはPromise以外の値をreturnした場合、その返り値で解決されるFulfilledなPromiseを返します。これは、返した値が`Promise.resolve`されているのとほとんど同じ感覚です。

```
// `Promise.resolve(undefined)`を返したのと同じ
async function resolveUndefined() {
    // 何も値を返さない場合は`undefined`を返すのと同じ
}
resolveUndefined().then((value) => {
    console.log(value); // => undefined
});
// `Promise.resolve("値")`を返したのと同じ
async function resolveFn() {
    return "値";
}
resolveFn().then((value) => {
    console.log(value); // => "値"
});
```

次に、Async FunctionがPromiseをreturnした場合、その返り値のPromiseをそのまま返します。これは、`Promise#then`メソッドでRejectedなPromiseを返すことで、`throw`文を使わずにPromiseをrejectする方法と同じです。

```
// resolveFnは**Fulfilled**なPromiseインスタンスを返している
// Async Functionは自動的にPromiseを返すので、単に値を返しても同じ
```

```
async function resolveFn() {
    return Promise.resolve("値");
}
resolveFn().then((value) => {
    console.log(value); // => "値"
});

// rejectFnは**Rejected**なPromiseインスタンスを返している
async function rejectFn() {
    return Promise.reject(new Error("エラーメッセージ"));
}
rejectFn().catch((error) => {
    console.log(error.message); // => "エラーメッセージ"
});
```

最後に、Async Function内で例外が発生した場合は、そのエラーをもつRejectedなPromiseを返します。これは、Promise内での処理が自動的に `try…catch` されているのと同じで、Async Functionでも例外が発生した場合は自動的にキャッチされます。

```
// exceptionFnは例外を投げている
async function exceptionFn() {
    throw new Error("例外が発生しました");
    // 例外が発生したため、この行は実行されません
}

// Async Functionで例外が発生するとRejectedなPromiseが返される
exceptionFn().catch((error) => {
    console.log(error.message); // => "例外が発生しました"
});
```

どの場合でも、Async Functionは必ずPromiseを返すことがわかります。このようにAsync Functionを呼び出す側から見れば、Async FunctionはPromiseを返すただの関数と何も変わりません。

</async-function-syntax>
<async-function-await>
<title> `await` 式</title>
Async Functionはasync/awaitとも呼ばれることがあります。この呼ばれ方からも分かるように、Async Functionと `await` 式は共に利用します。

`await` 式はAsync Function内でのみ利用できます。 `await` 式は右辺の `Promise` インスタンスがFulfilledまたはRejectedになるまで、その行（文）で非同期処理の完了を待ちます。そして `Promise` インスタンスの状態が変わると、次の行（文）から処理を再開します。

```
async function asyncMain() {
    // PromiseがFulfilledまたはRejectedとなるまで待つ
    await Promiseインスタンス;
    // Promiseインスタンスの状態が変わったら処理を再開する
}
```

通常の処理の流れでは、非同期処理を実行した場合にその非同期処理の完了を待つことなく、次の行（次の文）を実行します。しかし `await` 式では非同期処理を実行し完了するまで、次の行（次の文）を実行しません。そのため `await` 式を使うことで非同期処理が同期処理のように上から下へと順番に実行するような流れで書けます。

```
// async functionは必ずPromiseを返す
async function doAsync() {
    // 非同期処理
}
async function asyncMain() {
    // doAsyncの非同期処理が完了するまでまつ
    await doAsync();
    // 次の行はdoAsyncの非同期処理が完了されるまで実行されない
    console.log("この行は非同期処理が完了後に実行される");
}
```

`await` 式は式であるため右辺（`Promise` インスタンス）の評価結果を値として返します。この `await` 式の評価方法は評価するPromiseの状態（FulfilledまたはRejected）によって異なります。

`await` 式の右辺のPromiseがFulfilledとなった場合は、resolveされた値が `await` 式の返り値となります。

次のコードでは、`await` 式の右辺にある `Promise` インスタンスは `42` という値でresolveされています。そのため `await` 式の返り値は `42` となり、`value` 変数にもその値が入ります。

```
async function asyncMain() {
    const value = await Promise.resolve(42);
    console.log(value); // => 42
}
asyncMain(); // Promiseインスタンスを返す
```

これはPromiseを使って書くと次のコードと同様の意味となります。`await` 式を使うことでコールバック関数を使わずに非同期処理の流れを表現できていることがわかります。

```
function asyncMain() {
```

```
    return Promise.resolve(42).then((value) => {
        console.log(value); // => 42
    });
}
asyncMain(); // Promiseインスタンスを返す
```

`await`式の右辺のPromiseがRejectedとなった場合は、その場でエラーを`throw`します。また「async-function-syntax」で紹介したように、Async Function内で発生した例外は自動的にキャッチされます。そのため`await`式でPromiseがRejectedとなった場合は、そのAsync FunctionがRejectedなPromiseを返すことになります。

次のコードでは、`await`式の右辺にある`Promise`インスタンスがRejectedの状態になっています。そのため`await`式は エラー を`throw`します。そのエラーを自動的にキャッチするため、`asyncMain`関数はRejectedなPromiseを返します。

```
async function asyncMain() {
    const value = await Promise.reject(new Error("エラーメッセージ"));
    // await式で例外が発生したため、この行は実行されません
}
// Async Functionは自動的に例外をキャッチできる
asyncMain().catch((error) => {
    console.log(error.message); // => "エラーメッセージ"
});
```

`await`式がエラーを`throw`するということは、そのエラーは`try…catch`構文でキャッチできます。通常の非同期処理では完了する前に次の行が実行されてしまうため`try…catch`構文ではエラーをキャッチできませんでした。そのためPromiseでは`catch`メソッドを使いPromise内で発生したエラーをキャッチしていました。（Promise.prototype.doneとは何か?を参照）

次のコードでは、`await`式で発生した例外を`try…catch`構文でキャッチしています。

```
async function asyncMain() {
    // await式のエラーはtry...catchできる
    try {
        // `await`式で評価した右辺のPromiseがRejectedとなったため、例外がthrowされる
        const value = await Promise.reject(new Error("エラーメッセージ"));
        // await式で例外が発生したため、この行は実行されません
    } catch (error) {
        console.log(error.message); // => "エラーメッセージ"
    }
}
asyncMain().then(() => {
```

```
    console.log("この行は実行されます");
}, (error) => {
    // すでにtry...catchされているため、この行は実行されません
});
```

このように`await`式を使うことで、`try…catch`構文のように非同期処理を同期処理と同じ構文を使って扱えます。またコードの見た目も同期処理と同じように、その行（その文）の処理が完了するまで次の行を評価しないという分かりやすい形になるのは大きな利点です。

</async-function-await>
<promise-chain-to-async-function>
<title>Async Functionと配列</title>
Promiseと配列のように、配列を元にした複数の非同期処理を扱う場合のAsync Functionについて見ていきます。

例として、複数のリソースを順番に取得する処理をPromiseで書いていきます。

まずは、Promiseを使った非同期処理を行う関数として、リソースを擬似的に取得する`dummyFetch`という関数を実装していきます。`dummyFetch`関数は擬似的にデータ取得する関数で、1000ミリ秒未満のランダムなタイミングでレスポンスを返す非同期的な処理です。パスが`/resource`から始まるリソースを取得した場合は、そのレスポンスをもったResolved状態のPromiseオブジェクトを返します。それ以外の場合は、リソースの取得に失敗したとしてRejected状態のPromiseオブジェクトを返します。

```
/**
 * 1000ミリ秒未満のランダムなタイミングでレスポンスを擬似的にデータ取得する関数
 * 指定した`path`にデータがある場合、成功として**Resolved**状態のPromiseオブジェクトを返す
 * 指定した`path`にデータがない場合、失敗として**Rejected**状態のPromiseオブジェクトを返す
 */
function dummyFetch(path) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (path.startsWith("/resource")) {
                resolve({ body: `Response body of ${path}` });
            } else {
                reject(new Error("NOT FOUND"));
            }
        }, 1000 * Math.random());
    });
}
// `then`メソッドで成功時と失敗時に呼ばれるコールバック関数を登録
```

```
// /resource/A のリソースは存在するので成功しonFulfilledが呼ばれる
dummyFetch("/resource/A").then((response) => {
    console.log(response); // => { body: "Response body of /resource/A" }
}, (error) => {
    // この行は実行されません
});
// /not_found のリソースは存在しないのでonRejectedが呼ばれる
dummyFetch("/not_found").then((response) => {
    // この行は実行されません
}, (error) => {
    console.log(error.message); // => "NOT FOUND"
});
```

この `dymmyFetch` 関数を使い、複数のリソースを順番に取得する `fetchResources` 関数を実装していきます。`fetchResources` 関数は、配列で複数のリソースへのパスを受け取り、取得したリソースの中身(`body`)を配列として返すことにします。

まずは、Promise APIのみを使って `fetchResources` 関数を実装してみましょう。Promise APIでは、`Array#reduce` メソッドを使った逐次処理を実装することで、複数の非同期処理を順番に実行できます。(詳細はPromiseによる逐次処理を参照)

```
function dummyFetch(path) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (path.startsWith("/resource")) {
                resolve({ body: `Response body of ${path}` });
            } else {
                reject(new Error("NOT FOUND"));
            }
        }, 1000 * Math.random());
    });
}
// 複数のリソースを取得し、レスポンスボディの配列を返す
function fetchResources(resources) {
    const results = [];
    // 配列をpromise chainにして順番に処理する
    return resources.reduce((promise, resource) => {
        return promise.then(() => {
            return dummyFetch(resource).then((response) => {
                results.push(response.body);
                return results;
            });
        });
    }, Promise.resolve());
}
```

```
const resources = ["/resource/A", "/resource/B"];
// リソースを取得して出力する
fetchResources(resources).then((results) => {
    console.log(results); // => ["Response body of /resource/A", "Response body of /
resource/B"]
});
```

次に、先ほどと同じ処理をする `fetchResources` をAsync Functionと `await` 式で書いてみます。Async Functionとして定義した `fetchResources` 関数では、forループの中で `await` 式を使うことで複数の非同期処理を順番に実行できます。

次のコードでは、リソースのパスをforループで順番に、`dummyFetch` 関数を使ってリソースの中身を取得しています。forループによる反復処理も `await` 式で `dummyFetch` 関数の完了を待っているため、その非同期処理が終ってから次の反復処理を行います。すべてのforループの処理が終わると、`fetchResources` 関数が返したPromiseオブジェクトが変数 `results` が参照する値でresolveされます。

```
function dummyFetch(path) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (path.startsWith("/resource")) {
                resolve({ body: `Response body of ${path}` });
            } else {
                reject(new Error("NOT FOUND"));
            }
        }, 1000 * Math.random());
    });
}
// 複数のリソースを取得し、レスポンスボディの配列を返す
async function fetchResources(resources) {
    const results = [];
    for (let i = 0; i < resources.length; i++) {
        const resource = resources[i];
        const response = await dummyFetch(resource);
        results.push(response.body);
    }
    return results;
}
const resources = ["/resource/A", "/resource/B"];
// リソースを取得して出力する
fetchResources(resources).then((results) => {
    console.log(results); // => ["Response body of /resource/A", "Response body of /
resource/B"]
});
```

Promise APIのみで `fetchResources` 関数書いた場合は、コールバックの中で処理するためややこしい見た目になりがちです。一方で、Async Functionと `await` 式を使った場合は、非同期処理での取得と追加を順番に行うだけとなりネストがなく見た目はシンプルです。

<await-in-async-function>
<title> `await` 式はAsync Functionの中でのみ利用可能</title>
先ほどの `fetchResources` 関数ではforループを利用していました。このとき、配列の反復処理に `Array#forEach` メソッドを利用したくなるかもしれません。

しかし、次のようにforループを `Array#forEach` に変更するだけでは、構文エラー（Syntax Error）となってしまいます。

```
async function fetchResources(resources) {
    const results = [];
    // Syntax Errorとなる例
    resources.forEach(function(resources) {
        const resource = resources[i];
        // Async Functionではないスコープで`await`式を利用しているためSyntax Errorとなる
        const response = await dummyFetch(resource);
        results.push(response.body);
    });
    return results;
}
```

これは、 `await` 式はAsync Functionの直下でのみ利用できるからです。

Async Functionではない通常の関数で `await` 式を使うとSyntax Errorとなります。これは間違った `await` 式の使い方を防止するための仕様です。

```
function main(){
    // Syntax Error
    await Promise.resolve();
}
```

Async Function内で `await` 式を使って処理を待っている間も、関数の外側では通常とおり処理が進みます。次のコードを実行してみると、Async Function内で `await` しても、Async Function外の処理は停止していないことがわかります。

```
async function asyncMain() {
    // 中でawaitしても、Async Functionの外側の処理まで止まるわけではない
    await new Promise((resolve) => {
        setTimeout(resolve, 16);
```

```
    });
}
console.log("1. asyncMain関数を呼び出します");
// Async Functionは外から見れば単なるPromiseを返す関数
asyncMain().then(() => {
    console.log("3. asyncMain関数が完了しました");
});
// Async Functionの外側の処理はそのまま進む
console.log("2. asyncMain関数外では、次の行が同期的に呼び出される");
```

このように`await`式で非同期処理を一時停止しても、Async Function外の処理が停止するわけではありません。Async Function外の処理も停止できてしまうと、JavaScriptでは基本的にメインスレッドで多くの処理をするため、UIを含めた他の処理が止まってしまいます。これが`await`式がAsync Functionの外で利用できない理由の一つです。

`await`式はAsync Functionの中でのみ利用可能なため、コールバック関数もAsync Functionとして定義しないと`await`式が利用できないことに注意してください。

そのため、`fetchResources`関数の`Array#forEach`メソッドのコールバック関数に対して、`async`キーワードをつけることで構文エラーは発生しなくなります。この場合は、コールバック関数がAsync Functionとなるため、コールバック関数内で`await`式が利用できます。しかし、コールバック関数をAsync Functionに修正するだけでは、`fetchResources`関数が意図した結果を返しません。

次のように`Array#forEach`メソッドのコールバック関数をAsync Functionにしてみます。このコードを実行してみると、`fetchResources`関数の返したPromiseの結果は空の配列となり、意図した結果にならないことが分かります。

```
function dummyFetch(path) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (path.startsWith("/resource")) {
                resolve({ body: `Response body of ${path}` });
            } else {
                reject(new Error("NOT FOUND"));
            }
        }, 1000 * Math.random());
    });
}
// リソースを順番に取得する
async function fetchResources(resources) {
    const results = [];
    resources.forEach(async(resource) => {
        const response = await dummyFetch(resource);
```

```
        results.push(response.body);
    });
    return results;
}
const resources = ["/resource/A", "/resource/B"];
// リソースを取得して出力する
fetchResources(resources).then((results) => {
    // resultsは空になってしまう
    console.log(results); // => []
});
```

forEachメソッドのコールバック関数としてAsync Functionを渡し、コールバック関数中でawait式を利用して非同期処理の完了を待っています。しかし、この非同期処理の完了を待つのはコールバック関数Async Functionの中だけで、外側ではfetchResources関数の処理が進んでいます。そのため、コールバック関数でresultsに結果を追加する前に、fetchResources関数はその時点の変数resultsの値でresolveしてしまいます。

次のようにfetchResources関数にコンソール出力を入れてみると動作が分かりやすいでしょう。forEachメソッドのコールバック関数が完了するのは、fetchResources関数の呼び出しがすべて終わった後になります。そのためfetchResources関数はその時点の変数resultsの値である空の配列でresolveします。

```
function dummyFetch(path) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (path.startsWith("/resource")) {
                resolve({ body: `Response body of ${path}` });
            } else {
                reject(new Error("NOT FOUND"));
            }
        }, 1000 * Math.random());
    });
}
// リソースを順番に取得する
async function fetchResources(resources) {
    const results = [];
    console.log("1. fetchResourcesを開始");
    resources.forEach(async(resource) => {
        console.log(`2. ${resource}の取得開始`);
        const response = await dummyFetch(resource);
        console.log(`3. ${resource}の取得完了`);
        results.push(response.body);
    });
    console.log("4. fetchResourcesを終了");
```

```
    return results;
}
const resources = ["/resource/A", "/resource/B"];
// リソースを取得して出力する
fetchResources(resources).then((results) => {
    console.log(results); // => []
});
```

この問題を解決する方法として、先ほどのようにコールバック関数を使わずにforループを使う方法があります。また、リソースを順番が重要ではない場合は、`Promise.all`メソッドを使い、複数の非同期処理を1つのPromiseとしてまとめる方法があります。

</await-in-async-function>
<relationship-promise-async-function>
<title>PromiseとAsync Functionを組み合わせる</title>
Async Functionと`await`式でも非同期処理を同期処理のような見た目で書けます。しかし、非同期処理は必ずしも順番に処理することが重要ではない場合があります。その際に、forループと`await`式で書くと複数の非同期処理を順番に行ってしまい無駄な待ち時間を作ってしまうコードになってしまいます。

先ほど`fetchResources`関数ではリソースAを取得し終わってから、リソースBを取得していました。このとき、取得順が変わっても問題無い場合は、リソースAとリソースBを同時に取得する方が効率的です。

`Promise.all`メソッドを使い、リソースAとリソースBを取得する非同期処理を1つの`Promise`インスタンスにまとめることができます。`await`式が評価するのは`Promise`インスタンスであるため、`await`式は`Promise.all`メソッドなど`Promise`インスタンスを返す処理と組み合わせて利用できます。

そのため、先ほど`fetchResources`関数でリソースを同時に取得する場合は、次のように書けます。`Promise.all`メソッドは複数のPromiseを配列で受け取り、それを1つのPromiseとしてまとめたものを返す関数です。`Promise.all`メソッドの返す`Promise`インスタンスを`await`することで、非同期処理の結果を配列としてまとめて取得できます。

```
function dummyFetch(path) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (path.startsWith("/resource")) {
                resolve({ body: `Response body of ${path}` });
            } else {
                reject(new Error("NOT FOUND"));
            }
```

```
        }, 1000 * Math.random());
    });
}

// 複数のリソースを取得しレスポンスボディの配列を返す
async function fetchResources(resources) {
    // リソースをまとめて取得する
    const promises = resources.map((resource) => {
        return dummyFetch(resource);
    });
    // すべてのリソースが取得できるまで待つ
    // Promise.allは [ResponseA, ResponseB] のように結果が配列となる
    const responses = await Promise.all(promises);
    // 取得した結果からレスポンスのボディだけを取り出す
    return responses.map((response) => {
        return response.body;
    });
}
const resources = ["/resource/A", "/resource/B"];
// リソースを取得して出力する
fetchResources(resources).then((results) => {
    console.log(results); // => ["Response body of /resource/A", "Response body of /
resource/B"]
});
```

このようにAsync Functionや`await`式は既存のPromise APIと組み合わせて利用できます。Async Functionも内部的にPromiseの仕組みを利用した構文です。そのため、Async FunctionとPromiseのAPIを組み合わせて考えることは重要です。

</relationship-promise-async-function>
</promise-chain-to-async-function>

Promises API Reference

Promise#then

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

thenコード例

```
const promise = new Promise((resolve, reject) => {
    resolve("thenに渡す値");
});
```

```
promise.then((value) => {
    console.log(value);
}, (error) => {
    console.error(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。

Promise#catch

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

catchのコード例

```
const promise = new Promise((resolve, reject) => {
    resolve("thenに渡す値");
});
promise.then((value) => {
    console.log(value);
}).catch((error) => {
    console.error(error);
});
```

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

Promise#finally

```
promise.finally(onFinally);
```

finallyのコード例

```
let isLoading = true;
```

```
fetch("https://httpbin.org/get").then((response) => {
    if (response.ok) {
        return response.json();
    }
    throw new TypeError("正しくデータを取得できなかった");
  })
  .then((json) => {
      console.log("リクエストが成功した", json);
   })
  .catch((error) => {
      console.error("リクエストが失敗した", error);
   })
  .finally(() => {
      // 成功、失敗どちらの場合も必ず呼ばれる処理
      isLoading = false;
  });
```

Promise chainが成功、失敗どちらの場合でも呼ばれるハンドラを登録し、新しいpromiseオブジェクトを作成して返す。返したpromiseオブジェクトは、finallyの呼び出し元となったpromiseオブジェクトの状態を引き継ぐ。

Promise.resolve

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

Promise.resolveのコード例

```
const taskName = "task 1";
asyncTask(taskName).then((value) => {
    console.log(value);
}).catch((error) => {
    console.error(error);
});
function asyncTask(name) {
    return Promise.resolve(name).then((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` をも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

Promise.reject

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

Promise.rejectのコード例

```
const 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.reject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。

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

Promise.all

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

Promise.allのコード例

```
const p1 = Promise.resolve(1);
const p2 = Promise.resolve(2);
const p3 = Promise.resolve(3);
Promise.all([p1, p2, p3]).then((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。

Promise.race

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

Promise.raceのコード例

```
const p1 = Promise.resolve(1);
const p2 = Promise.resolve(2);
const p3 = Promise.resolve(3);
Promise.race([p1, p2, p3]).then((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。

用語集

Promises
: プロミスという仕様そのもの

promiseオブジェクト
: プロミスオブジェクト、`Promise` のインスタンスオブジェクトのこと

ES Promises
: ECMAScriptの仕様[111] を明示的に示す場合にprefixとして ES をつける

[111] https://tc39.es/ecma262/

Promises/A+
: Promises/A+[112]のこと。ES Promisesの前身となったコミュニティベースの仕様であり、ES 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こと。`.then`というメソッドをもつオブジェクト。

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。

参考サイト

w3ctag/promises-guide[113]（日本語訳）[114]
: Promisesのガイド - 概念的な説明はここから得たものが多い

domenic/promises-unwrapping[115]
: ES Promisesの仕様の元となったリポジトリ - issueを検索して得た経緯や情報も多い

ECMAScript 2015 Language Specification – ECMA-262 6th Edition[116]
: ECMAScript 2015におけるPromiseの仕様書 - PromiseはES2015で導入された

ECMAScript® Language Specification[117]
: ECMAScriptの仕様書 - 最新の仕様

JavaScript Promises: There and back again - HTML5 Rocks[118]
: Promisesについての記事 - 完成度がとても高くサンプルコードやリファレンス等を参考にした

Node.jsにPromiseが再びやって来た！ - ぼちぼち日記[119]
: Node.jsとPromiseの記事 - thenableについて参考にした

Exploring JS: JavaScript books for programmers[120]
: ECMAScript全般について詳しく書かれている書籍

112 http://promises-aplus.github.io/promises-spec/
113 https://github.com/w3ctag/promises-guide
114 https://triple-underscore.github.io/promises-guide-ja.html
115 https://github.com/domenic/promises-unwrapping
116 http://www.ecma-international.org/ecma-262/6.0/index.html#sec-promise-objects
117 https://tc39.es/ecma262/
118 http://www.html5rocks.com/ja/tutorials/es6/promises/
119 http://d.hatena.ne.jp/jovi0608/20140319/1395199285
120 https://exploringjs.com/

著者について

azu[121] (Twitter : @azu_re[122])

ブラウザ、JavaScriptの最新技術を常に追いかけている。

目的を手段にしてしまうことを得意としている(この書籍もその結果できた)。

Web Scratch[123] や JSer.info[124] といったサイトを運営している。

著者へのメッセージ/おまけ

以下の おまけ.pdf[125] では、この書籍を書き始めた理由や、どのように書いていったか、テストなどについて書かれています。

- JavaScript Promiseの本のおまけ[126]

Gumroadから無料 または 好きな値段でダウンロードすることができます。

ダウンロードする際に作者へのメッセージも書けるので、メッセージを残すついでにダウンロードして行ってください。

問題の指摘などがありましたら、GitHubやGitterに書いてくださると解決できます。

- Issues · azu/promises-book[127]
- azu/promises-book - Gitter[128]

121 https://github.com/azu/
122 https://twitter.com/azu_re
123 https://efcl.info/
124 https://jser.info/
125 https://gumroad.com/l/javascript-promise
126 https://gumroad.com/l/javascript-promise
127 https://github.com/azu/promises-book/issues?state=open
128 https://gitter.im/azu/promises-book