JavaScript Promiseの本

azu

Table of Contents

はじめに

書籍の目的

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

この書籍を読むことで学べる事として次の3つを目標としています

- Promiseについて学び、パターンやテストを扱えるようになる事
- Promiseの向き不向きについて学び、何でもPromiseで解決するべきではないと知る事
- ES6 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ようになる事

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

そのため、FirefoxやChromeなど先行実装しているブラウザでは、ライブラリを使うこと無く利用できる機能であり、 またES6 Promisesは元がPromises/A+というコ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を書けるようになることを目的としています。

本書を読むにあたって

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

- JavaScript: The Good Parts[1]
- JavaScriptパターン[2]
- JavaScript 第6版[3]

[1] http://www.oreilly.co.jp/books/9784873113913/
[2] http://www.oreilly.co.jp/books/9784873114880/
[3] http://www.oreilly.co.jp/books/9784873115733/

- パーフェクトJavaScript[4]
- Effective JavaScript[5]

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

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

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

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。
  - 例えば、`Promise#then` という表記は、Promiseのインスタンスオブジェクトの `then` というメソッドを示しています。
- オブジェクトメソッドを object.method という表記で示す。
  - これはJavaScriptの意味そのままで、`Promise.all` なら静的メソッドの事を示しています。

> この部分には文章についての補足が書かれています。

本書のソースコード/ライセンス

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

この書籍は AsciiDoc[6] という形式で書かれています。

[4] http://gihyo.jp/book/2011/978-4-7741-4813-7?ard=1400715177
[5] http://books.shoeisha.co.jp/book/b107881.html
[6] http://asciidoctor.org/

- azu/promises-book[7]

またリポジトリには書籍中に出てくるサンプルコードのテストも含まれています。

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

意見や疑問点

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

- Issues · azu/promises-book[8]

また、この書籍についての チャットページ[9] 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[10] 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[11] 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事をいいます。 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[12]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。

7 https://github.com/azu/promises-book
8 https://github.com/azu/promises-book/issues?state=open
9 https://gitter.im/azu/promises-book
10 https://twitter.com/search?q=%23Promise%E6%9C%AC
11 https://github.com/azu/promises-book/pulls
12 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。

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", function(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

```
var promise = getAsyncPromise("fileA.txt"); ❶
promise.then(function(result){
    // 取得成功の処理
}).catch(function(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様々な非同期処理のパターンを形成することが出来ます。

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。

Promise Overview

ES6 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。

```
var promise = new Promise(function(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)
```

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出来ます。 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)` を使うことが出来ます。

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

Static Method

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

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

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(function (resolve, reject) {
        setTimeout(function () {
            resolve('Async Hello world');
        }, 16);
    });
}
❷
asyncFunction().then(function (value) {
    console.log(value);    // => 'Async Hello world'
}).catch(function (error) {
    console.log(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事はないですが、 `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(function (value) {
    console.log(value);
}, function (error) {
    console.log(error);
});
```

Promiseの状態

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

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

"has-resolution" - Fulfilled
  resolve(成功)した時。この時 `onFulfilled` が呼ばれる

"has-rejection" - Rejected
  reject(失敗)した時。この時 `onRejected` が呼ばれる

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

読み方ですが、 左がES6 Promisesの仕様で定められている名前で、 右がPromises/A+で登場する状態の名前になっています。

基本的にこの状態をプログラムで直接触る事はないため、名前自体は余り気にしなくても問題ないです。 この書籍では、 Promises/A+[13] の Pending、Fulfilled 、Rejected を用いて解説していきます。

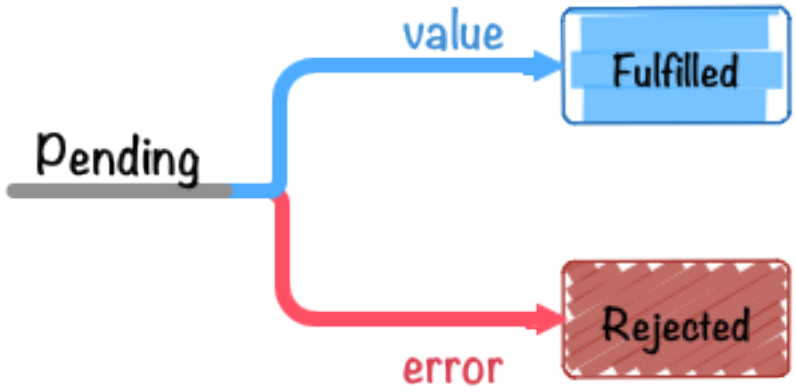


Figure 1. promise states

ECMAScript Language Specification ECMA-262 6th Edition – DRAFT[14] では `[[PromiseStatus]]` という内部定義によって状態が定められています。 `[[PromiseStatus]]` にアクセスするユーザーAPIは用意されていないため、基本的には知る方法はありません。

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

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

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

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

Settled
: resolve(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わかると思います。

13 http://promises-aplus.github.io/promises-spec/
14 http://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects

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[15] 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。

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 `getURL` という関数を作ります。

xhr-promise.js

```
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
```

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

```
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
// 実行例
var URL = "http://httpbin.org/get";
getURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(function onRejected(error){
    console.error(error);
});
```

この `getURL` では、 XHRでの取得結果のステータスコードが200の場合のみ `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出来ます）

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

次に、 `reject` の方を見て行きましょう。

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

エラーの場合は `reject(new Error(req.statusText));` というように、Errorオブジェクトを作成して渡している事がわかると思います。 `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200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出来ます）

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

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

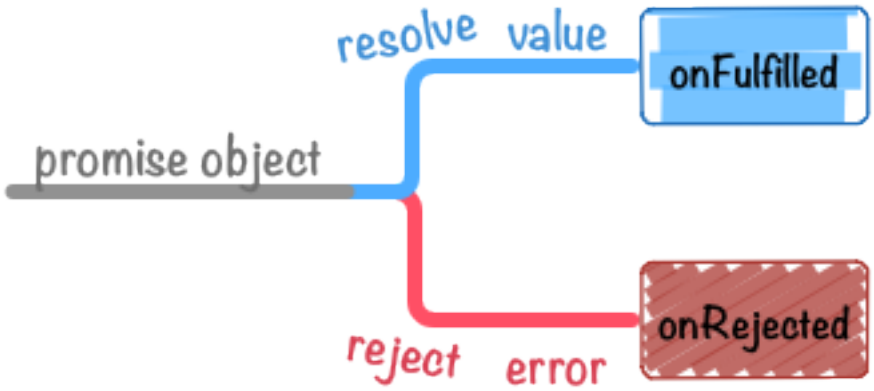


Figure 2. promise value flow

まずは、`getURL` で通信が成功して値が取得出来た場合の処理を書いてみましょう。

この場合の 通信が成功した というのは、 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 時という事ですね。

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。

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

```
getURL(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
getURL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、 `getURL` で何らかの問題があってエラーが起きた場合の処理を書いてみましょう。

この場合の エラーが起きた というのは、 rejectされたことより promiseオブジェクトがRejectedの状態になった 時という事ですね。

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事で行えます。

先ほどのソースにrejectされた場合の処理を追加してみましょう。

```
var URL = "http://httpbin.org/status/500"; ❶
getURL(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
`getURL` 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出来ます。

```
getURL(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とcatchの違い で紹介します。

まとめ

この章では以下のことについて簡単に紹介しました。

- `new Promise` を使った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、同様の事はコールバック関数を渡す形でも出来るのに対してPromiseで書くメリットについては触れていませんでした。 次の章では、Promiseのメリットであるエラーハンドリングの仕組みをコールバックベースの実装と比較しながら見て行きたいと思います。

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

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

new Promiseのショートカット

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

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

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

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

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

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

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

Thenable

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

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

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

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

どういうものがthenableなのかというと、分かりやすい例では jQuery.ajax()[16]の返り値もthenableです。

`jQuery.ajax()` の返り値は jqXHR Object[17] 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('/json/comment.json');// => `.then` を持つオブジェクト
```

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出来ます。

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

16 https://api.jquery.com/jQuery.ajax/
17 http://api.jquery.com/jQuery.ajax/#jqXHR

thenableをpromiseオブジェクトにする

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

jQueryとthenable

jQuery.ajax()[18]の返り値も `.then` というメソッドを持ったjqXHR Object[19]で、 このオブジェクトは Deferred Object[20] のメソッドやプロパティ等を継承しています。

しかし、このDeferred ObjectはPromises/A+やES6 Promisesに準拠したものではないため、 変換できたように見えて一部欠損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[21] の `then` の挙動が違うために発生します。

そのため、 `.then` というメソッドを持っていた場合でも、必ずES6 Promisesとして使えるとは限らない事は知っておくべきでしょう。

- JavaScript Promises: There and back again - HTML5 Rocks[22]
- You're Missing the Point of Promises[23]
- https://twitter.com/hirano_y_aa/status/398851806383452160

`Promise.resolve` は共通の挙動である `then` だけを利用して、 様々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事になります。

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事からもその挙動が想像できるかもしれません。

18 https://api.jquery.com/jQuery.ajax/
19 http://api.jquery.com/jQuery.ajax/#jqXHR
20 http://api.jquery.com/category/deferred-object/
21 http://api.jquery.com/category/deferred-object/
22 http://www.html5rocks.com/ja/tutorials/es6/promises/#toc-lib-compatibility
23 http://domenic.me/2012/10/14/youre-missing-the-point-of-promises/

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。

Promise.reject

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

例えば、 `Promise.reject(new Error("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(function(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(function(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。

```
var promise = new Promise(function (resolve){
    console.log("inner promise"); // 1
    resolve(42);
});
promise.then(function(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` を渡して呼び出す事はできますね。

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。

何故、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以外でも適応できるため、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) {
    var readyState = document.readyState;
    if (readyState === 'interactive' || readyState === 'complete') {
        fn();
    } else {
        window.addEventListener('DOMContentLoaded', fn);
    }
}
onReady(function () {
    console.log('DOM fully loaded and parsed');
});
console.log('==Starting==');
```

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が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が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。

この問題の対処法として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) {
    var readyState = document.readyState;
    if (readyState === 'interactive' || readyState === 'complete') {
        setTimeout(fn, 0);
    } else {
        window.addEventListener('DOMContentLoaded', fn);
    }
```

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

この問題については、 Effective JavaScript[24] 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。
>
> — 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事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(function (resolve, reject) {
        var readyState = document.readyState;
        if (readyState === 'interactive' || readyState === 'complete') {
            resolve();
        } else {
            window.addEventListener('DOMContentLoaded', resolve);
        }
    });
}
```

[24] http://effectivejs.com/

```
onReadyPromise().then(function () {
    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わかるように、 Promiseではいくらでもメソッドチェーンを繋げて処理を書いていくことが出来ます。

promiseはメソッドチェーンで繋げて書ける

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

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

Promiseのメソッドチェーンだと長いので、今後はpromise chainと呼びます。 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行きましょう。

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

次のようなpromise chainを見てみましょう。

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");
}

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

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

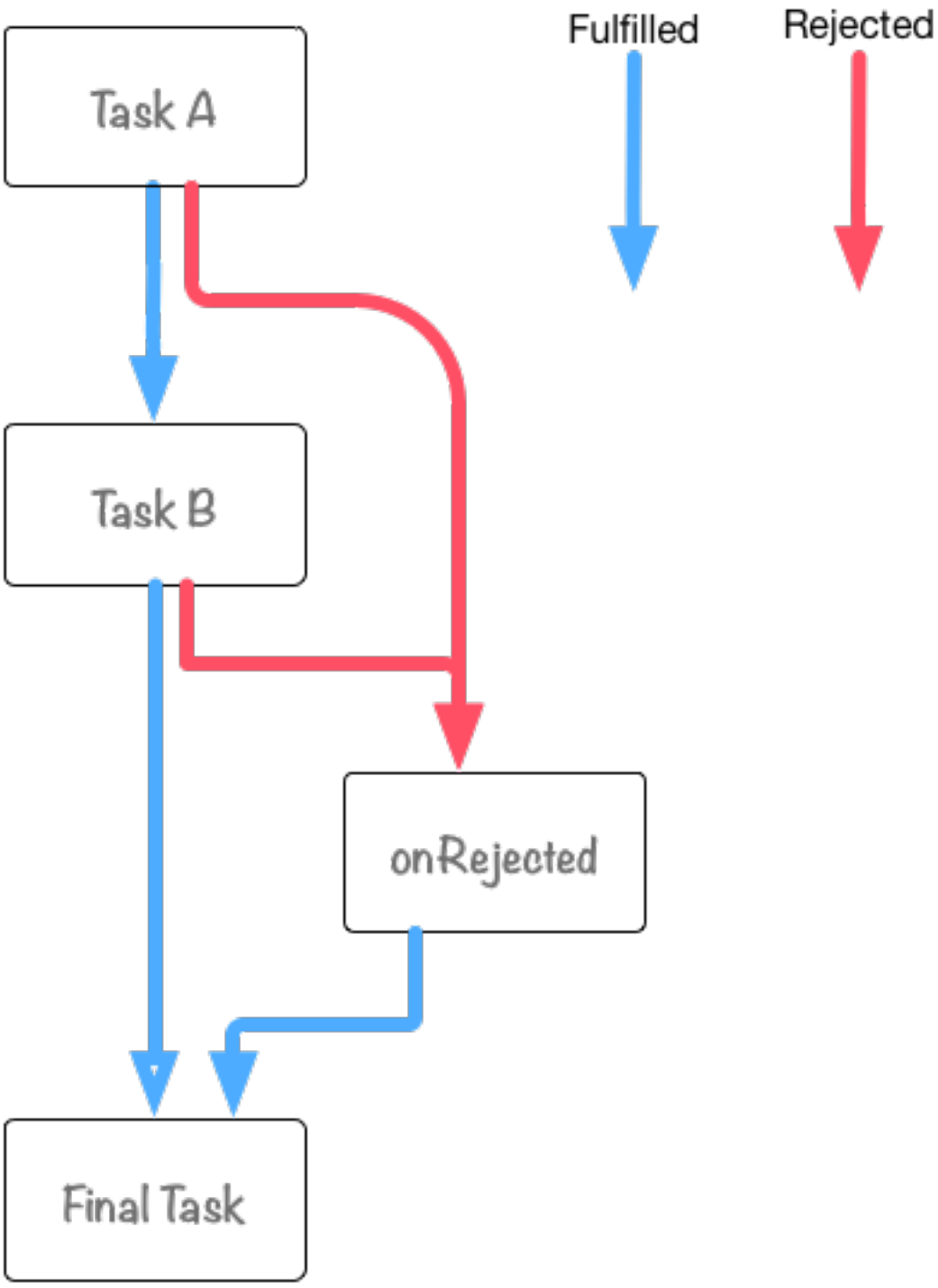


Figure 3. promise-then-catch-flow.jsの図

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

`then`
: onFulfilledの処理を登録

`catch`
: onRejectedの処理を登録

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

これは、Task A または Task B の処理にて、次のような場合に onRejected が呼ばれるという事を示しています。

- 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。

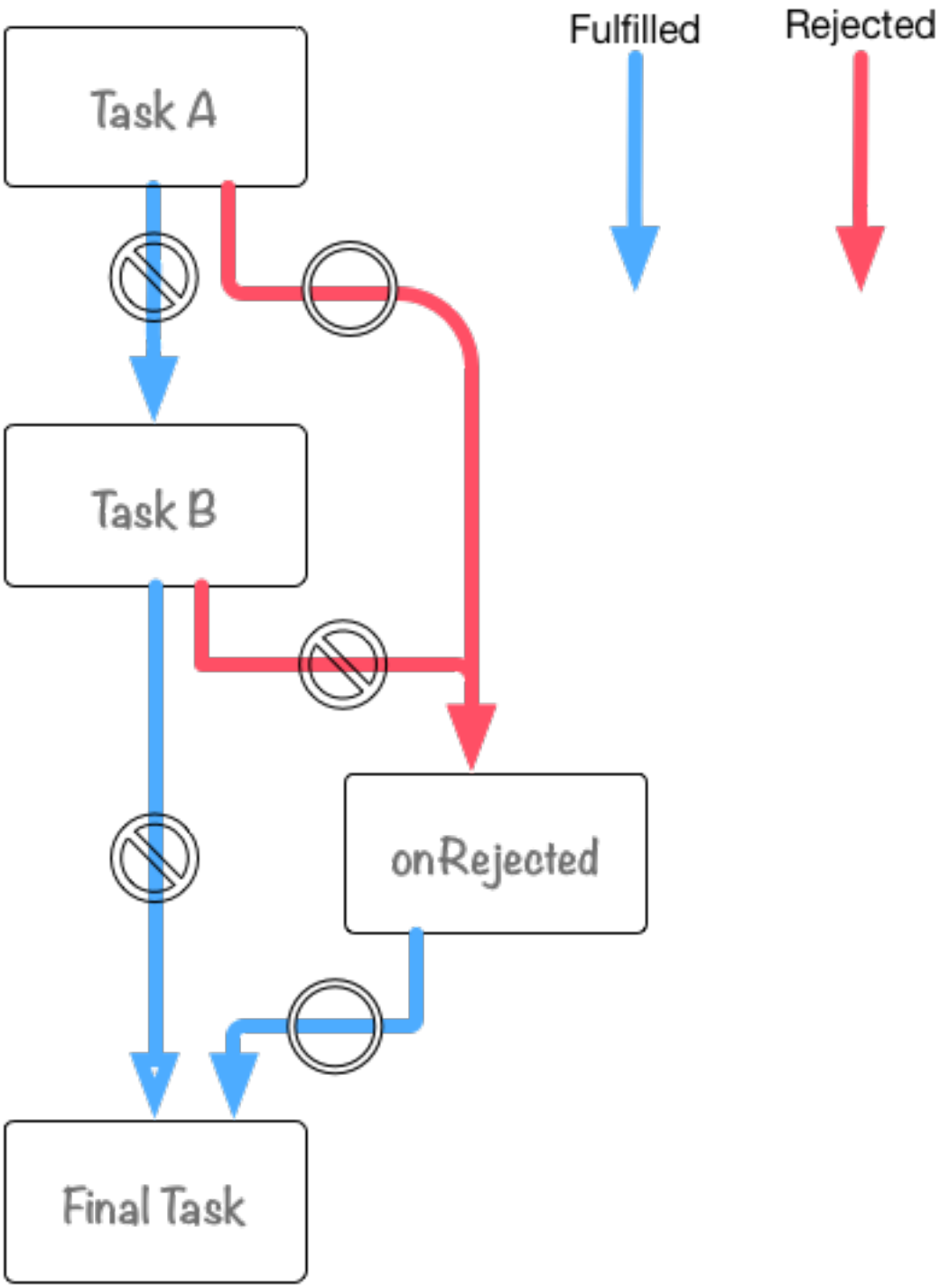


Figure 4. Task Aで例外が発生した時の図

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.log(error);// => "throw Error @ Task A"
}
function finalTask() {
    console.log("Final Task");
}

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

実行してみると、Task B が呼ばれていない事がわかるでしょう。

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。

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

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

実際に例を見てみましょう。

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
}

var promise = Promise.resolve(1);
promise
    .then(increment)
    .then(doubleUp)
    .then(output)
    .catch(function(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

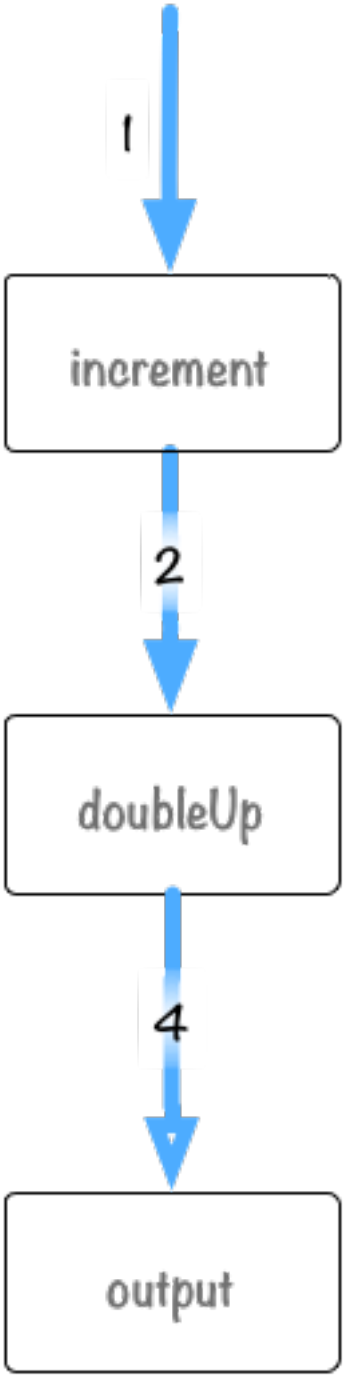


Figure 5. promise-then-passing-value.jsの図

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` することが出来ます。

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` は既に使っていましたね。

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

> 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[25] 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[26] を利用しています。

Promise#catchの実行結果

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

25 https://github.com/jakearchibald/es6-promise
26 https://github.com/jakearchibald/es6-promise

```
    console.error(error);
});
```

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

これはどういう事かというと、`catch` という単語はECMAScriptにおける 予約語[27] であることが関係します。

ECMAScript 3では予約語はプロパティの名前に使うことが出来ませんでした。IE8以下はECMAScript 3の実装であるため、`catch` というプロパティを使う `promise.catch()` という書き方が出来ないので、 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[28] 、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。

> ECMAScript5でも予約語は Identifier[29] 、つまり変数名、関数名には利用することが出来ません。 `for` という変数が定義できてしまうと `for` 文との区別ができなくなってしまいます。 プロパティの場合は `object.for` と `for` 文の区別はできるので、少し考えてみると自然な動作ですね。

このECMAScript 3の予約語の問題を回避する書き方も存在します。

ドット表記法[30] 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[31] は有効な識別子ではなくても利用できます。

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

Promise#catchの識別子エラーの回避

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

27 http://mothereff.in/js-properties#catch
28 http://es5.github.io/#x7.6
29 http://es5.github.io/#x7.6
30 https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Property_Accessors#Dot_notation
31 https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Property_Accessors#Bracket_notation

```
    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

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

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

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、知らない間に回避出来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。

コラム: thenは常に新しいpromiseオブジェクトを返す

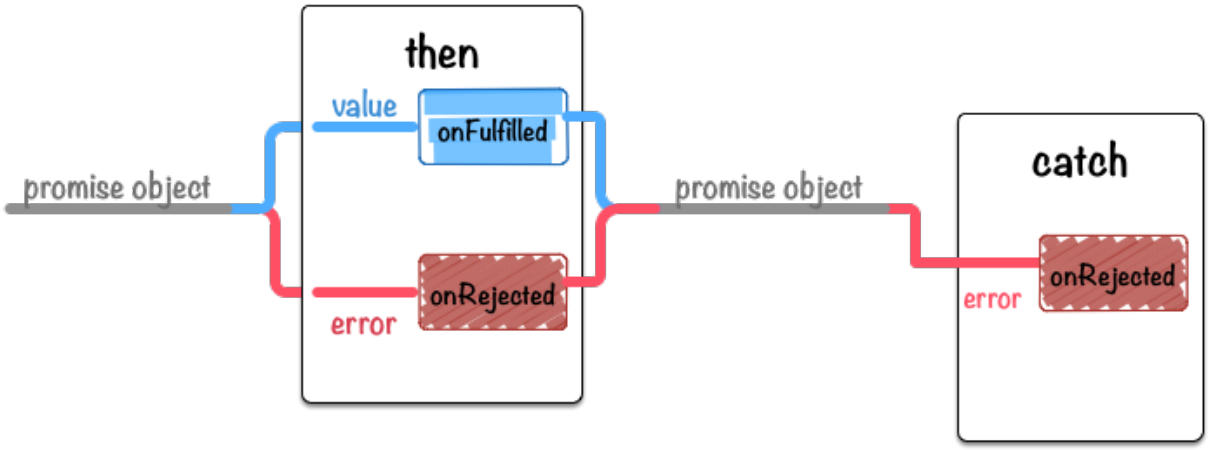
`aPromise.then(…).catch(…)` は一見すると、全て最初の `aPromise` オブジェクトに メソッドチェーンで処理を書いてるように見えます。

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。

```
var aPromise = new Promise(function (resolve) {
    resolve(100);
});
var thenPromise = aPromise.then(function (value) {
    console.log(value);
});
var catchPromise = thenPromise.catch(function (error) {
    console.error(error);
});
console.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事が起こりえると思います。

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

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

// vs

// 2: `then` はpromise chain通り順番に呼び出される
var bPromise = new Promise(function (resolve) {
    resolve(100);
});
bPromise.then(function (value) {
    return value * 2;
```

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

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() {
    var promise = Promise.resolve();
    promise.then(function() {
        // 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出来ます。

`then` で作成したオブジェクトを返す

```
function anAsyncCall() {
    var promise = Promise.resolve();
    return promise.then(function() {
        // 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[32] 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` で登録出来る事を学びました。

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

例えば、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。

コールバックで複数の非同期処理

multiple-xhr-callback.js

```
function getURLCallback(URL, callback) {
    var req = new XMLHttpRequest();
    req.open('GET', URL, true);
    req.onload = function () {
        if (req.status === 200) {
            callback(null, req.responseText);
        } else {
            callback(new Error(req.statusText), req.response);
        }
    };
    req.onerror = function () {
        callback(new Error(req.statusText));
    };
    req.send();
}
// <1> JSONパースを安全に行う
function jsonParse(callback, error, value) {
```

[32] http://taoofcode.net/promise-anti-patterns/

```
    if (error) {
        callback(error, value);
    } else {
        try {
            var result = JSON.parse(value);
            callback(null, result);
        } catch (e) {
            callback(e, value);
        }
    }
}
// <2> XHRを叩いてリクエスト
var request = {
        comment: function getComment(callback) {
            return getURLCallback('http://azu.github.io/promises-book/json/comment.json',
 jsonParse.bind(null, callback));
        },
        people: function getPeople(callback) {
            return getURLCallback('http://azu.github.io/promises-book/json/people.json',
 jsonParse.bind(null, callback));
        }
    };
// <3> 複数のXHRリクエストを行い、全部終わったらcallbackを呼ぶ
function allRequest(requests, callback, results) {
    if (requests.length === 0) {
        return callback(null, results);
    }
    var req = requests.shift();
    req(function (error, value) {
        if (error) {
            callback(error, value);
        } else {
            results.push(value);
            allRequest(requests, callback, results);
        }
    });
}
function main(callback) {
    allRequest([request.comment, request.people], callback, []);
}
// 実行例
main(function(error, results){
    if(error){
        return console.error(error);
    }
    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` を使って同様の事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 getURL(URL) {
    return new Promise(function (resolve, reject) {
```

```
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/
comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/
people.json').then(JSON.parse);
        }
    };
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] は記録する初期値を部分適用している
    var pushValue = recordValue.bind(null, []);
    return request.comment().then(pushValue).then(request.people).then(pushValue);
}
// 実行例
main().then(function (value) {
    console.log(value);
}).catch(function(error){
    console.error(error);
});
```

コールバックスタイルと比較してみると次の事がわかります。

- `JSON.parse` をそのまま使っている
- `main()` はpromiseオブジェクトを返している

・エラーハンドリングは返ってきたpromiseオブジェクトに対して書いている

先ほども述べたように mainの `then` の部分がクドく感じます。

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 `getURL` 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Rejected)した時に、次の `.then` が呼び出すこと事が出来ます。

promise-all-xhr.js

```
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/
comment.json').then(JSON.parse);
```

```
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/
people.json').then(JSON.parse);
        }
    };
function main() {
    return Promise.all([request.comment(), request.people()]);
}
// 実行例
main().then(function (value) {
    console.log(value);
}).catch(function(error){
    console.log(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(function (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(function (resolve) {
        setTimeout(function () {
            resolve(delay);
        }, delay);
    });
}
var startDate = Date.now();
// 全てがresolveされたら終了
Promise.all([
    timerPromisefy(1),
    timerPromisefy(32),
    timerPromisefy(64),
    timerPromisefy(128)
]).then(function (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。

```
var 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かかる事がわかります。

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

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

promise-race-timer.js

```
// `delay`ミリ秒後にresolveする
function timerPromisefy(delay) {
    return new Promise(function (resolve) {
        setTimeout(function () {
            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(function (value) {
    console.log(value);    // => 1
});
```

上記のコードだと、1ms後、32ms後、64ms後、128ms後にそれぞれpromiseオブジェクトが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

```
var winnerPromise = new Promise(function (resolve) {
        setTimeout(function () {
            console.log('this is winner');
            resolve('this is winner');
        }, 4);
    });
var loserPromise = new Promise(function (resolve) {
        setTimeout(function () {
            console.log('this is loser');
            resolve('this is loser');
        }, 1000);
    });
// 一番最初のものがresolveされた時点で終了
Promise.race([winnerPromise, loserPromise]).then(function (value) {
    console.log(value);    // => 'this is winner'
});
```

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

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

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

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

then or catch?

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

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

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

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出来ますが、 そのエラー処理は第一引数の `onFulfilled` で指定した関数内で起きたエラーをキャッチする事は出来ません。

つまり、この場合、 `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事が出来ます。

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

`.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。

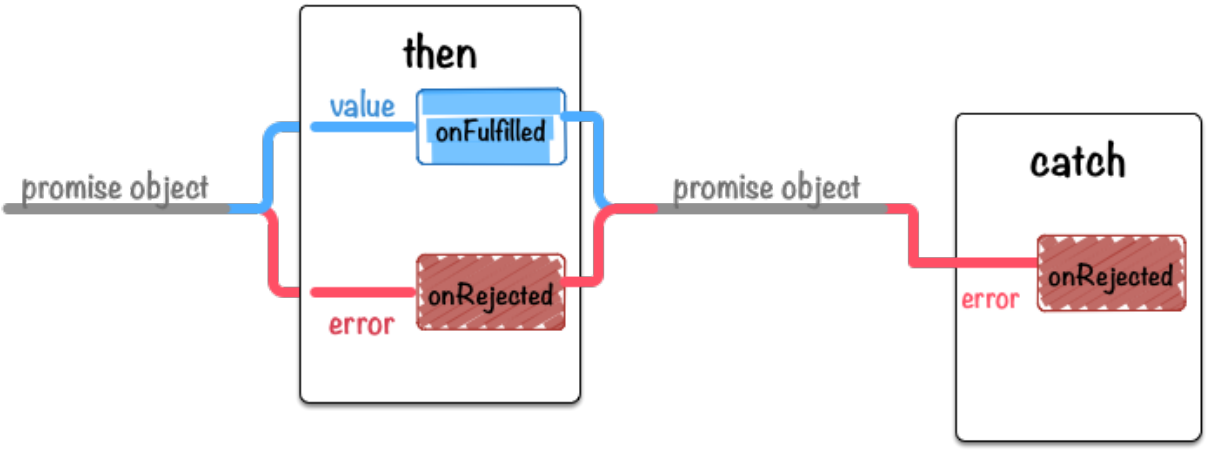


Figure 6. Then Catch flow

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出来るのは、次のchainで書かれた `catch` となります。

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

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

まとめ

ここでは次のような事について学びました。

1. `promise.then(onFulfilled, onRejected)` において
   - `onFulfilled` で例外がおきても、この `onRejected` はキャッチできない
2. `promise.then(onFulfilled).catch(onRejected)` とした場合
   - `then` 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

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

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

ここではまず、 Mocha[33]を使った基本的なPromiseのテストの書き方について学んでいきましょう。

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

この書籍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[34] から参照できます。

Mochaとは

ここでは、 Mocha[35]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を選んだ理由としては、以下の点で選択しました。

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

33 http://mochajs.org/
34 https://github.com/azu/promises-book
35 http://mochajs.org/

・"Promiseのテスト"をサポートしている

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

また、アサーションライブラリには、 power-assert[36]を利用しますが、 アサーション自体はNode.jsの `assert` モジュールと全く同じであるため、今回はあまり気にしなくても問題ありません。

まずは、コールバック関数のテストと同じような形でテストを書いてみましょう。

コールバックスタイルのテスト

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

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

basic-test.js

```
"use strict";
var assert = require("power-assert");
describe("Basic Test", function () {
    context("When Callback(high-order function)", function () {
        it("should use `done` for test", function (done) {
            setTimeout(function () {
                assert(true);
                done();
            }, 0);
        });
    });
    context("When promise object", function () {
        it("should use `done` for test?", function (done) {
            var promise = Promise.resolve(1);
            // このテストコードはある欠陥があります
            promise.then(function (value) {
                assert(value === 1);
                done();
            });
        });
    });
});
```

[36] https://github.com/twada/power-assert

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

Mochaでの非同期テストは以下のような流れになっています。

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

❶　非同期処理のコールバックを指定
❷　`done` を呼ぶことでテストの終了を宣言
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、Promiseのテストの方を見てみましょう。

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

間違ったテストの書き方

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

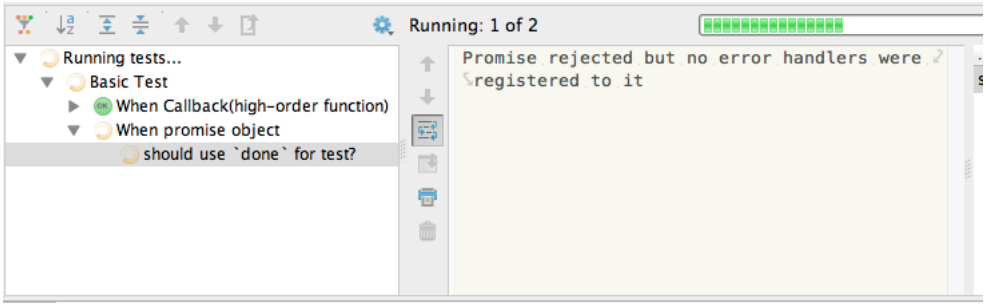


Figure 7. テストが終わることがないためタイムアウトするまでそこで止まる

`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throwされません。

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

ちゃんとテストが失敗する例

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

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

`assert` がパスした場合は単純に `done()` が呼ばれ、`assert` が失敗した場合は `done(error)` が呼ばれます。

これでようやくコールバックスタイルのテストと同等のPromiseのテストを書くことができました。

しかし、`assert` 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出来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"をサポートしているという事がどういう機能なのかを学んでいきましょう。

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[37] 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することが出来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行きたいと思います。

mocha-promise-test.js

```
"use strict";
var assert = require("power-assert");
describe("Promise Test", function () {
    it("should return a promise object", function () {
        var promise = Promise.resolve(1);
        return promise.then(function (value) {
            assert(value === 1);
        });
    });
});
```

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

[37] http://mochajs.org/#asynchronous-code

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", function () {
    return Promise.resolve().then(function () {
        assert(false);// => テストが失敗する
    });
});
```

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出来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[38]
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、この書き方で良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。

例えば、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", function () {
    return mayBeRejected().catch(function (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

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

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
    return Promise.resolve();
}
it("is bad pattern", function () {
    return mayBeRejected().catch(function (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", function () {
    return mayBeRejected().then(failTest).catch(function (error) {
        assert.deepEqual(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。

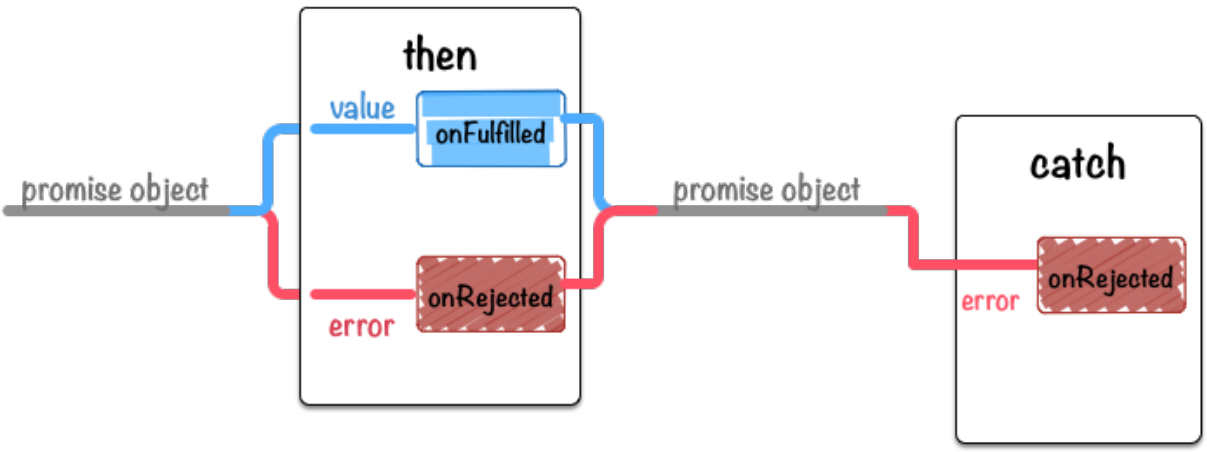


Figure 8. Then Catch flow

`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事が出来るでしょうか?

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
　意図した通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", function () {
    // FulFilledとなった場合はテストは失敗する
    return mayBeRejected().then(failTest, function (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。

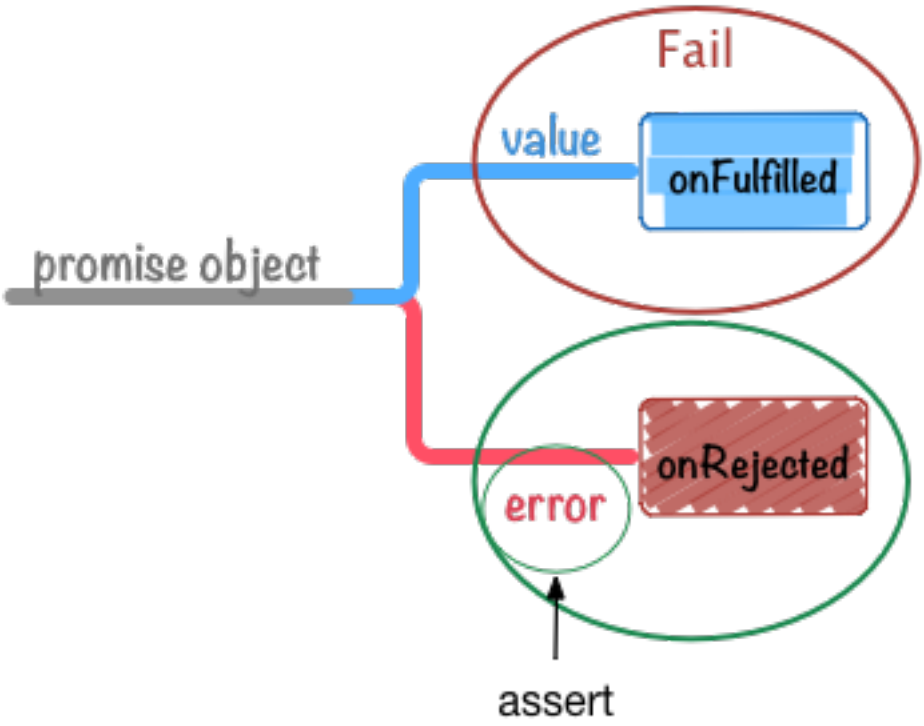


Figure 9. Promise onRejected test

then or catch?の時は、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, function(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行きましょう。

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事ですね。

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, function(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[39] にありますが、 今回はその場で簡単に定義して進めます。

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

shouldRejected-test.js

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

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

```
        assert(error.message === 'human error');
    });
});
```

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

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

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

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

`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, function(error){
    assert(error.message === 'human error');
});
// == ほぼ同様の意味
shouldRejected(promise).catch(function (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。

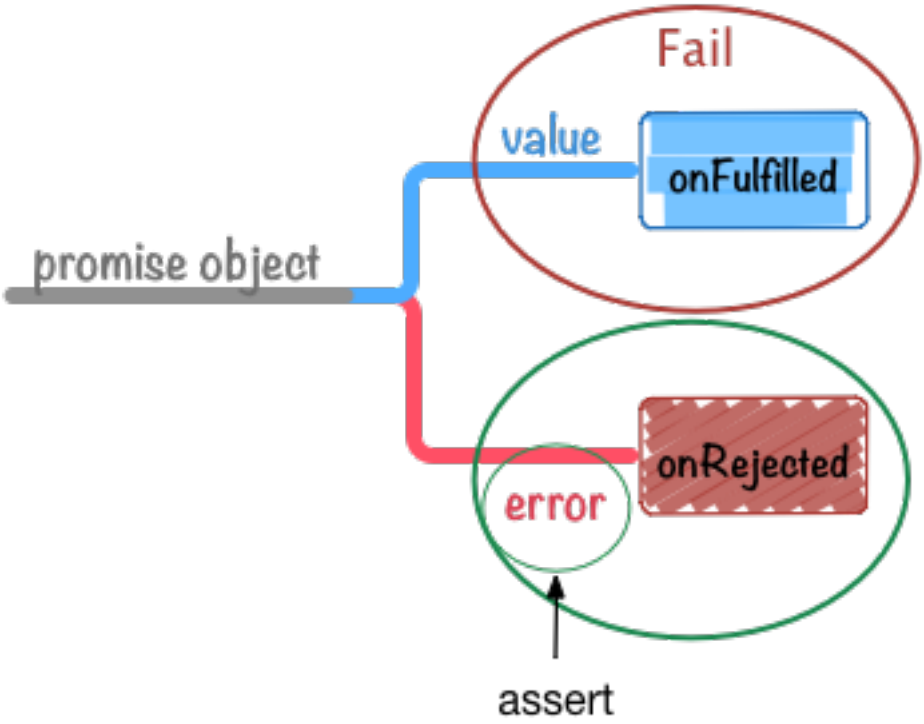


Figure 10. Promise onRejected test

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

```
var assert = require('power-assert');
function shouldFulfilled(promise) {
    return {
        'then': function (fn) {
            return promise.then(function (value) {
                fn.call(promise, value);
            }, function (reason) {
                throw reason;
            });
        }
    };
}
it('should be fulfilled', function () {
    var promise = Promise.resolve('value');
    return shouldFulfilled(promise).then(function (value) {
        assert(value === 'value');
    });
});
```

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

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[40] からダウンロードすることが出来ます。

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

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

例えば、 CoffeeScript[41]でテストを書いたりすると、 CoffeeScriptには暗黙のreturnがあるので、`done` を使ったほうが分かりやすいかもしれません。

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

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。

40 https://github.com/azu/promise-test-helper
41 http://coffeescript.org/

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「ES6 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ES6 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事は、Thenableであるということなので、 `Promise.resolve`を使い、ES6のPromiseで定められたpromiseオブジェクトに変換することが出来ます。

> ES6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事で、IE10等まだ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出来るようになります。

jakearchibald/es6-promise[42]
: ES6 Promisesと互換性を持ったPolyfillライブラリです。 RSVP.js[43] 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。

yahoo/ypromise[44]
: YUI[45] の一部としても利用されているES6 Promisesと互換性を持ったPolyfillライブラリです。 この書籍ではこのPolyfillを読み込み、サンプルコードを動かしています。

getify/native-promise-only[46]
: ES6 Promisesのpolyfillとなることを目的としたライブラリです。 ES6 Promisesの仕様に厳密に沿うように作られており、仕様にない機能は入れないようになっています。 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ます。

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[47]
: `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[48] 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[49]
: Promise互換に加えて、キャンセル出来るPromiseや進行度を取得出来る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。

42 https://github.com/jakearchibald/es6-promise
43 https://github.com/tildeio/rsvp.js
44 https://github.com/yahoo/ypromise
45 http://yuilibrary.com/
46 https://github.com/getify/native-promise-only/
47 https://github.com/kriskowal/q
48 https://github.com/kriskowal/q-io
49 https://github.com/petkaantonov/bluebird

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[50]

Qのドキュメントにはjqueryが持つ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[51] にまとめられています。

・bluebird/API.md at master · petkaantonov/bluebird[52]

BluebirdではPromiseを使った豊富な実装例に加えて、エラーが起きた時の対処法やPromiseのアンチパターン[53] 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事が多いと思います。

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 ES6 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。

50 https://github.com/kriskowal/q/wiki/API-Reference
51 https://github.com/kriskowal/q/wiki/Coming-from-jQuery
52 https://github.com/petkaantonov/bluebird/blob/master/API.md
53 https://github.com/petkaantonov/bluebird/wiki/Promise-anti-patterns

Web Notificationsをthenableにする

Web Notifications[54]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[55]
- Can I use Web Notifications[56]

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。

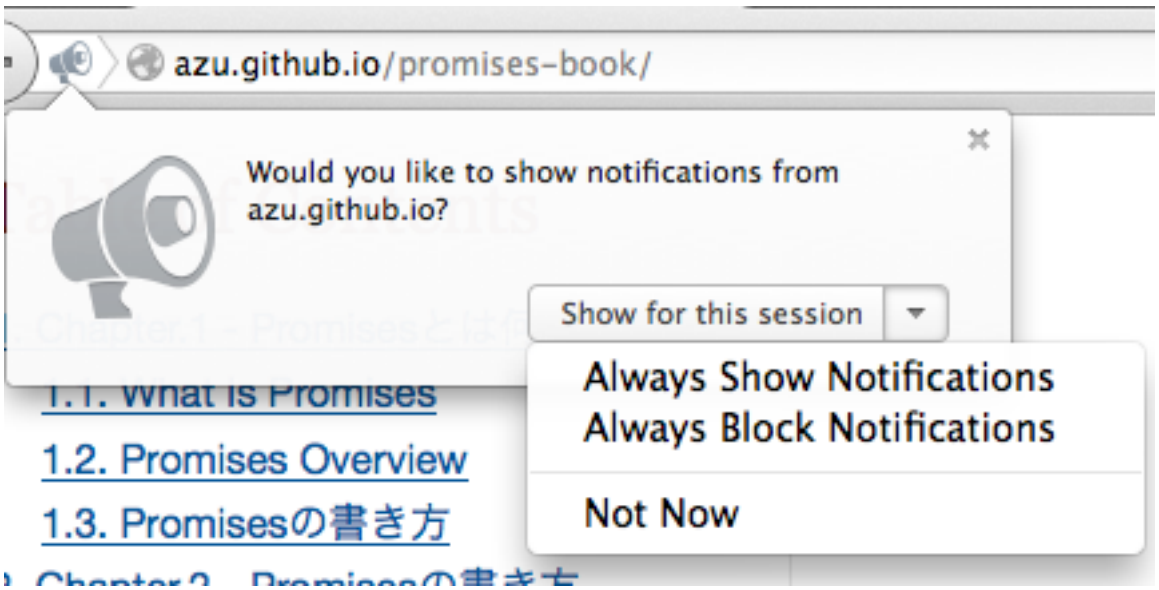


Figure 11. Notificationの許可ダイアログ

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。

54 https://developer.mozilla.org/ja/docs/Web/API/notification
55 https://developer.mozilla.org/ja/docs/WebAPI/Using_Web_Notifications
56 http://caniuse.com/notifications

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(function (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
    - 既に許可済みのケース
    - その場で許可を貰うケース
- 許可がない場合は何もしない

いくつか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出来ます。

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出来ます。

notification-callback.js

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

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。

notification-as-promise.js

```
function notifyMessage(message, options, callback) {
    if (Notification && Notification.permission === 'granted') {
        var notification = new Notification(message, options);
        callback(null, notification);
    } else if (Notification.requestPermission) {
        Notification.requestPermission(function (status) {
            if (Notification.permission !== status) {
                Notification.permission = status;
            }
            if (status === 'granted') {
                var notification = new Notification(message, options);
                callback(null, notification);
            } else {
                callback(new Error('user denied'));
            }
        });
    } else {
        callback(new Error('doesn\'t support Notification API'));
    }
}
function notifyMessageAsPromise(message, options) {
    return new Promise(function (resolve, reject) {
        notifyMessage(message, options, function (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(function(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。
>
> 既に許可されている
> `.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
>
> 既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[57]

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

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出来るようにする
- 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事を 実現するのに役立つ概念です。

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

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

notification-thenable.jsにはnotification-as-promise.jsと比べた時に、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。

例えば、 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、 ES6 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。

ES6 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

```
var Q = require("Q");
// このpromiseオブジェクトはES6のもの
var promise = new Promise(function(resolve){
    resolve(1);
});
// Q promiseオブジェクトに変換する
Q(promise).then(function(value){
    console.log(value);
}).finally(function(){ 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出来ます。

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出来ます。

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` をしてもプログラムは終了しません。

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

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

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

先ほどのコードは以下のように書くことが出来ます。

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

`throw` が `reject` に変わったと考えれば、`reject` にはErrorオブジェクト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、 何故 `throw` ではなく `reject` した方がいいのでしょうか?

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

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

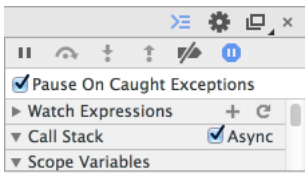


Figure 12. Pause On Caught Exceptions

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。

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

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

thenでもrejectする

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

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

```
var promise = Promise.resolve();
promise.then(function (value) {
    setTimeout(function () {
        // 一定時間経って終わらなかったらrejectしたい - 2
    }, 1000);
    // 時間がかかる処理 - 1
    somethingHardWork();
}).catch(function (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。 この時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事が出来ます。

この時、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出来ます。

```
var promise = Promise.resolve();
promise.then(function () {
    var retPromise = new Promise(function (resolve, reject) {
        // resolve or reject で onFulfilled or onRejected どちらを呼ぶか決まる
    });
    return retPromise;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することが出来ます。

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

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

```
var onRejected = console.error.bind(console);
var promise = Promise.resolve();
promise.then(function () {
    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事が多いので、 覚えておくといいでしょう。

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

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事があるかもしれません。 有名な所だと jQuery.Deferred[58] や JSDeferred[59] 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[60] のようなDeferredの実装を中心にして話を進めます。

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

58 http://api.jquery.com/category/deferred-object/
59 http://cho45.stfuawsc.com/jsdeferred/
60 http://api.jquery.com/category/deferred-object/

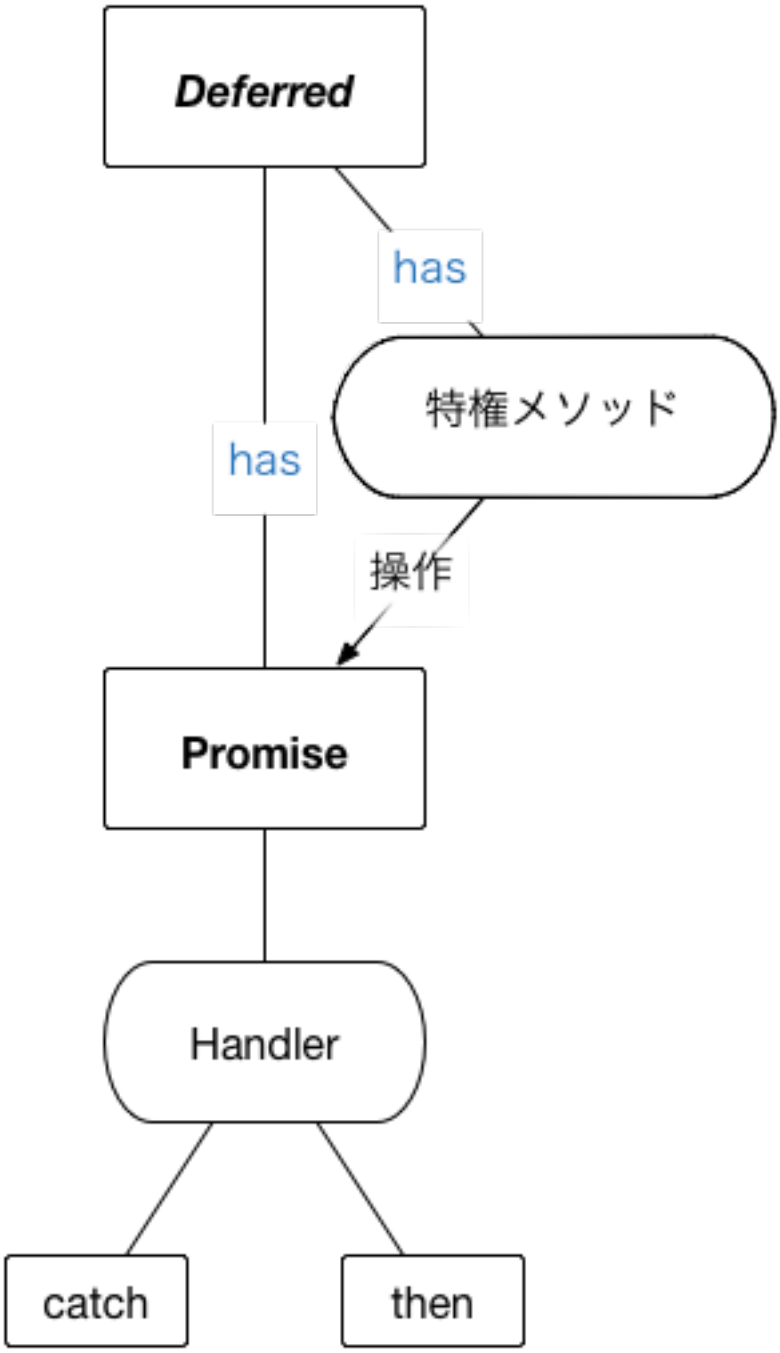


Figure 13. DeferredとPromise

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。もちろんPromiseを持たないDeferredの実装もあります。

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

Deferred top on Promise

Promiseの上にDeferredを実装した例です。

deferred.js

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

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

xhr-deferred.js

```
function Deferred() {
    this.promise = new Promise(function (resolve, reject) {
        this._resolve = resolve;
        this._reject = reject;
    }.bind(this));
}
Deferred.prototype.resolve = function (value) {
    this._resolve.call(this.promise, value);
};
Deferred.prototype.reject = function (reason) {
    this._reject.call(this.promise, reason);
};
function getURL(URL) {
    var deferred = new Deferred();
    var req = new XMLHttpRequest();
    req.open('GET', URL, true);
    req.onload = function () {
        if (req.status === 200) {
            deferred.resolve(req.responseText);
        } else {
            deferred.reject(new Error(req.statusText));
        }
```

```
    };
    req.onerror = function () {
        deferred.reject(new Error(req.statusText));
    };
    req.send();
    return deferred.promise;
}
// 実行例
var URL = "http://httpbin.org/get";
getURL(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事が出来ません。

Promiseで実装したものと見比べていきたいと思います。

xhr-promise.js

```
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
// 実行例
var URL = "http://httpbin.org/get";
getURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(console.error.bind(console));
```

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

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

逆に以下の部分は同じ事をやっています。

- 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。

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

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

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

```
var deferred = new Deferred();

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

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

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

言い換えると、 Promiseはこの値は将来的に正常な値(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事を表すオブジェクトで、 処理が終わった時の結果を取得する機構(Promise)に加えて処理を進める機構をもったものといえるかもしれません。

より詳しくDeferredについて知りたい人は以下を参照するといいでしょう。

- Promise & Deferred objects in JavaScript Pt.1: Theory and Semantics.[61]
- Twisted 入門 — Twisted Intro[62]
- Promise anti patterns · petkaantonov/bluebird Wiki[63]
- Coming from jQuery · kriskowal/q Wiki[64]

> DeferredはPythonの Twisted[65] 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[66] 、 dojo/Deferred[67] 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[68] プロパティを持っているので、 これを利用すると簡単に出来ますが、複数のXHRを束ねたタイムアウトや他の機能でも応用が効くため、 分かりやすい非同期処理であるXHRにおけるタイムアウトによるキャンセルを例にしています。

Promiseで一定時間待つ

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

タイムアウトというのは一定時間経ったら何かするという処理なので、 `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。

61 http://blog.mediumequalsmessage.com/promise-deferred-objects-in-javascript-pt1-theory-and-semantics
62 http://skitazaki.appspot.com/translation/twisted-intro-ja/index.html
63 https://github.com/petkaantonov/bluebird/wiki/Promise-anti-patterns#the-deferred-anti-pattern
64 https://github.com/kriskowal/q/wiki/Coming-from-jQuery
65 https://twistedmatrix.com/trac/
66 http://mochi.github.io/mochikit/doc/html/MochiKit/Async.html
67 http://dojotoolkit.org/reference-guide/1.9/dojo/Deferred.html
68 https://developer.mozilla.org/ja/docs/XMLHttpRequest/Synchronous_and_Asynchronous_Requests

delayPromise.js

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

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

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

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

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。

```
var winnerPromise = new Promise(function (resolve) {
        setTimeout(function () {
            console.log('this is winner');
            resolve('this is winner');
        }, 4);
    });
var loserPromise = new Promise(function (resolve) {
        setTimeout(function () {
            console.log('this is loser');
            resolve('this is loser');
        }, 1000);
    });
// 一番最初のものがresolveされた時点で終了
Promise.race([winnerPromise, loserPromise]).then(function (value) {
    console.log(value);    // => 'this is winner'
```

```
});
```

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

simple-timeout-promise.js

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

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

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

```
function delayPromise(ms) {
    return new Promise(function (resolve) {
        setTimeout(resolve, ms);
    });
}
function timeoutPromise(promise, ms) {
    var timeout = delayPromise(ms).then(function () {
            throw new Error('Operation timed out after ' + ms + ' ms');
        });
    return Promise.race([promise, timeout]);
}
// 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(function(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出来ましたが、 このままでは通常のエラーとタイムアウトのエラーの区別がつかなくなってしまいます。

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

カスタムErrorオブジェクト

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

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

> ECMAScript6では `class` 構文を使うことで内部的にも正確に継承を行うことが出来ます。
>
> ```
> class MyError extends Error{
>     // Error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(function (propName) {
        Object.defineProperty(target, propName, Object.getOwnPropertyDescriptor(source,
 propName));
    });
    return target;
}
function TimeoutError() {
    var superInstance = Error.apply(null, arguments);
    copyOwnFrom(this, superInstance);
}
TimeoutError.prototype = Object.create(Error.prototype);
```

```
TimeoutError.prototype.constructor = TimeoutError;
```

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

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

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

promise.catch(function(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[69] で詳しく紹介されています。 また、 Error - JavaScript | MDN[70] 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 `getURL` を少し拡張して、 XHRを包んだpromiseオブジェクトと共にそのXHRを中止するメソッドを持つオブジェクトを返すようにしています。

delay-race-cancel.js

```
function cancelableXHR(URL) {
    var req = new XMLHttpRequest();
```

69 http://speakingjs.com/es5/ch28.html
70 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

```
    var promise = new Promise(function (resolve, reject) {
            req.open('GET', URL, true);
            req.onload = function () {
                if (req.status === 200) {
                    resolve(req.responseText);
                } else {
                    reject(new Error(req.statusText));
                }
            };
            req.onerror = function () {
                reject(new Error(req.statusText));
            };
            req.onabort = function () {
                reject(new Error('abort this request'));
            };
            req.send();
        });
    var abort = function () {
        // 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が時間内に取得出来た場合
        a. 通常のpromiseと同様に `then` で中身を取得する
    - タイムアウトとなった場合は
        a. `throw TimeoutError` されるので `catch` する

b. catchしたエラーオブジェクトが `TimeoutError` のものだったら `abort` を呼び出してXHRをキャンセルする

これらの要素を全てまとめると次のように書けます。

delay-race-cancel-play.js

```
function copyOwnFrom(target, source) {
    Object.getOwnPropertyNames(source).forEach(function (propName) {
        Object.defineProperty(target, propName, Object.getOwnPropertyDescriptor(source,
 propName));
    });
    return target;
}
function TimeoutError() {
    var superInstance = Error.apply(null, arguments);
    copyOwnFrom(this, superInstance);
}
TimeoutError.prototype = Object.create(Error.prototype);
TimeoutError.prototype.constructor = TimeoutError;
function delayPromise(ms) {
    return new Promise(function (resolve) {
        setTimeout(resolve, ms);
    });
}
function timeoutPromise(promise, ms) {
    var timeout = delayPromise(ms).then(function () {
            return Promise.reject(new TimeoutError('Operation timed out after ' + ms + '
 ms'));
        });
    return Promise.race([promise, timeout]);
}
function cancelableXHR(URL) {
    var req = new XMLHttpRequest();
    var promise = new Promise(function (resolve, reject) {
            req.open('GET', URL, true);
            req.onload = function () {
                if (req.status === 200) {
                    resolve(req.responseText);
                } else {
                    reject(new Error(req.statusText));
                }
            };
            req.onerror = function () {
                reject(new Error(req.statusText));
            };
```

```
            req.onabort = function () {
                reject(new Error('abort this request'));
            };
            req.send();
        });
    var abort = function () {
        // 既にrequestが止まってなければabortする
        // https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest/
Using_XMLHttpRequest
        if (req.readyState !== XMLHttpRequest.UNSENT) {
            req.abort();
        }
    };
    return {
        promise: promise,
        abort: abort
    };
}
var object = cancelableXHR('http://httpbin.org/get');
// main
timeoutPromise(object.promise, 1000).then(function (contents) {
    console.log('Contents', contents);
}).catch(function (error) {
    if (error instanceof TimeoutError) {
        object.abort();
        return console.log(error);
    }
    console.log('XHR Error :', error);
});
```

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

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` 出来るようにするという手段もあると思いますが、 promiseオブジェクトは値を抽象化したオブジェクトであるため、何でも操作用のメソッドをつけていくと複雑になってしまうかもしれません。

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

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

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

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

cancelableXHR.js

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

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

function abortPromise(promise) {
    if (typeof promise === "undefined") {
        return;
    }
    var request;
    Object.keys(requestMap).some(function (URL) {
        if (requestMap[URL].promise === promise) {
            request = requestMap[URL].request;
            return true;
        }
    });
    if (request != null && request.readyState !== XMLHttpRequest.UNSENT) {
        request.abort();
    }
}
module.exports.createXHRPromise = createXHRPromise;
module.exports.abortPromise = abortPromise;
```

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

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

var xhrPromise = cancelableXHR.createXHRPromise('http://httpbin.org/get');❶
xhrPromise.catch(function (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

ここでは以下の事について学びました。

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

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

Promise.prototype.done とは何か?

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

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

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

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

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(function (error) {
            setTimeout(function () {
                throw error;
            }, 0);
        });
    };
}
var promise = Promise.resolve();
promise.done(function () {
    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

```
var promise = Promise.resolve();
promise.then(function () {
    JSON.parse("this is not json");
}).catch(function (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わかると思います。

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

何故このようなPromiseの機能とは相反するメソッドが、多くのライブラリで実装され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(function (resolve) {
        resolve(JSON.parse(value));
    });
}
```

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

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

```
function JSONPromise(value) {
    return new Promise(function (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(function(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

```
var string = "jsonではない文字列";
JSONPromise(string).then(function (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良いですが、 メソッドをtypoしたことによるSyntax Errorなどはより深刻な問題となりやすいです。

typoによるエラー

```
var string = "{}";
```

```
JSONPromise(string).then(function (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出来る場合は何も問題ありませんが、 Promiseの実装によってはこのようなミスが検知しにくくなるケースがあることを知っておくべきでしょう。

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

このunhandled rejectionが検知しにくい問題はPromiseの実装に依存します。 例えば、 ypromise[71] はunhandled rejectionがある場合は、その事をコンソールに表示します。

```
Promise rejected but no error handlers were
registered to it
```

また、 Bluebird[72] の場合も、 明らかに人間のミスにみえるReferenceErrorの場合などはそのままコンソールにエラーを表示してくれます。

```
"Possibly unhandled ReferenceError. conosle is
not defined
```

ネイティブのPromiseの場合も同様にこの問題への対処としてGC-based unhandled rejection trackingというものが 搭載されつつあります。

これはpromiseオブジェクトがガーベッジコレクションによって回収されるときに、 それがunhandled rejectionであるなら、エラー表示をするという仕組みがベースとなっているようです。

71 https://github.com/yahoo/ypromise
72 https://github.com/petkaantonov/bluebird

> Firefox[73] や Chrome[74] 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出来るので、 `Promise.prototype.done` というPromiseのprototype拡張として実装してみましょう。

promise-prototype-done.js

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

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(function callback() {
        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

[73] https://twitter.com/domenic/status/461154989856264192
[74] https://code.google.com/p/v8/issues/detail?id=3093

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[75]

`Promise.prototype.done` をよく見てみると、何も `return` していないこともわかると思います。 つまり、`done` は「ここでPromise chainは終了して、例外が起きた場合はそのままpromiseの外へ投げ直す」という処理になっています。

実装や環境がしっかり対応していれば、unhandled rejectionの検知はできるため、必ずしも `done` が必要というわけではなく、 また今回の`Promise.prototype.done`のように、`done` は既存のPromiseの上に実装することができるため、 ES6 Promisesの仕様そのものには入らなかったと言えるかもしれません。

今回の `Promise.prototype.done` の実装は promisejs.org[76] を参考にしています。

まとめ

このセクションでは、 Q[77] や Bluebird[78] や prfun[79] 等 多くのPromiseライブラリで実装されている `done` の基礎的な実装と、`then` とはどのような違いがあるかについて学びました。

`done` には2つの側面があることがわかりました。

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

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

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

75 http://techblog.yahoo.co.jp/programming/javascript_error/
76 https://www.promisejs.org/
77 https://github.com/kriskowal/q/wiki/API-Reference#promisedoneonfulfilled-onrejected-onprogress
78 https://github.com/petkaantonov/bluebird
79 https://github.com/cscott/prfun#promisedone—undefined

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

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

> Promises: The Extension Problem (part 4) | getiblog[80] 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[81] にて 詳しく解説されています。

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[82] 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[83]モジュールを例にしてみたいと思います。

80 http://blog.getify.com/promises-part-4/
81 http://speakingjs.com/es5/ch28.html
82 http://taiju.hatenablog.com/entry/20100307/1267962826
83 http://nodejs.org/api/fs.html

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

fs-method-chain.js

```
"use strict";
var fs = require("fs");
function File() {
    this.lastValue = null;
}
// Static method for File.prototype.read
File.read = function FileRead(filePath) {
    var file = new File();
    return file.read(filePath);
};
File.prototype.read = function (filePath) {
    this.lastValue = fs.readFileSync(filePath, "utf-8");
    return this;
};
File.prototype.transform = function (fn) {
    this.lastValue = fn.call(this, this.lastValue);
    return this;
};
File.prototype.write = function (filePath) {
    this.lastValue = fs.writeFileSync(filePath, this.lastValue);
    return this;
};
module.exports = File;
```

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

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

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

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";
var fs = require("fs");
function File() {
    this.promise = Promise.resolve();
}
// Static method for File.prototype.read
File.read = function (filePath) {
    var file = new File();
    return file.read(filePath);
};

File.prototype.then = function (onFulfilled, onRejected) {
    this.promise = this.promise.then(onFulfilled, onRejected);
    return this;
};
File.prototype["catch"] = function (onRejected) {
    this.promise = this.promise.catch(onRejected);
    return this;
};
File.prototype.read = function (filePath) {
    return this.then(function () {
        return fs.readFileSync(filePath, "utf-8");
    });
};
File.prototype.transform = function (fn) {
    return this.then(fn);
};
File.prototype.write = function (filePath) {
    return this.then(function (data) {
        return fs.writeFileSync(filePath, data)
    });
};
module.exports = File;
```

内部に持ってるpromiseオブジェクトに対するエイリアスとして `then` と `catch` を持たせていますが、それ以外のインターフェースは全く同じ使い方となっています。

そのため、先ほどのコードで `require` するモジュールを変更しただけで動作します。

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

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

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

```
var File = require("./fs-promise-chain");
File.read(inputFilePath)
    .transform(function (content) {
        return ">>" + content;
    })
    .write(outputFilePath);
// => 擬似的に以下のような流れに展開できる
promise.then(function read(){
        return fs.readFileSync(filePath, "utf-8");
    }).then(function transform(content) {
         return ">>" + content;
    }).then(function write(){
        return fs.writeFileSync(filePath, data);
    });
```

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

イメージとしては `promise = addPromiseChain(promise, fn);` のような感じになっていて、 既存のpromiseオブジェクトに対して新たな処理を追加したpromiseオブジェクトを作って返すため、 自分で逐次的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出来ますが、複雑になるため今回は単純な同期的なメソッドチェーンにしました。

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

エラーハンドリング

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

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

fs-promise-chainでのエラーハンドリング

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

fs-method-chain.jsに非同期処理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言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[84] が思い浮かぶと思います。

Stream[85] を使うと、 `this.lastValue` のような値を保持する必要がなくなる事や大きなファイルの扱いが改善されます。 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[86]
- Stream2の基本[87]
- Node-v0.12の新機能について[88]

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）

84 http://nodejs.org/api/stream.html
85 http://nodejs.org/api/stream.html
86 http://jxck.hatenablog.com/entry/20111204/1322966453
87 http://www.slideshare.net/shigeki_ohtsu/stream2-kihon
88 http://www.slideshare.net/shigeki_ohtsu/node-v012tng12

そのような仕組みはES6 Promisesにはありませんが、 著名なサードパーティのPromise実装である bluebird[89] などには Promisification[90] という機能が用意されています。

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

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

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

ArrayのPromiseラッパー

先ほどの Promisification[91] 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

```
"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;
};
```

89 https://github.com/petkaantonov/bluebird/
90 https://github.com/petkaantonov/bluebird/blob/master/API.md#promisification
91 https://github.com/petkaantonov/bluebird/blob/master/API.md#promisification

```
Object.getOwnPropertyNames(Array.prototype).forEach(function (methodName) {
    // Don't overwrite
    if (typeof ArrayAsPromise[methodName] !== "undefined") {
        return;
    }
    var arrayMethod = Array.prototype[methodName];
    if (typeof  arrayMethod !== "function") {
        return;
    }
    ArrayAsPromise.prototype[methodName] = function () {
        var that = this;
        var args = arguments;
        this.promise = this.promise.then(function () {
            that.array = Array.prototype[methodName].apply(that.array, args);
            return that.array;
        });
        return this;
    };
});

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

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

array-promise-chain-test.js

```
"use strict";
var assert = require("power-assert");
var ArrayAsPromise = require("../src/promise-chain/array-promise-chain");
describe("array-promise-chain", function () {
    function isEven(value) {
        return value % 2 === 0;
    }

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

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

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

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

`ArrayAsPromise` の実装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 Promisification[92] は Node.jsのCoreモジュールの非同期処理には `function(error,result){}` というように第一引数に `error` が来るというルールを利用して、 自動的にPromiseでラップしたメソッドを生成しています

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

92 https://github.com/petkaantonov/bluebird/blob/master/API.md#promisification

ES6 PromisesはCoreとなる機能しか用意されていません。 そのため、自分でPromiseを使った既存の機能のラッパー的な実装をする事があるかもしれません。

しかし、何度もコールバックを呼ぶEventのような処理がPromiseには不向きなように、 Promiseが常に最適な非同期処理という訳ではありません。

その機能にPromiseを使うのが最適なのかを考える事はこの書籍の目的でもあるため、 何でもPromiseにするというわけではなく、その目的にPromiseが合うのかどうかを考えてみるのもいいと思います。

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出来ません。

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。

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

```
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/
comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/
people.json').then(JSON.parse);
        }
    };
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] は記録する初期値を部分適用している
    var pushValue = recordValue.bind(null, []);
    return request.comment().then(pushValue).then(request.people).then(pushValue);
}
// 実行例
main().then(function (value) {
    console.log(value);
}).catch(function(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 getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
```

```
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/
comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/
people.json').then(JSON.parse);
        }
    };
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] は記録する初期値を部分適用してる
    var pushValue = recordValue.bind(null, []);
    // promiseオブジェクトを返す関数の配列
    var tasks = [request.comment, request.people];
    var promise = Promise.resolve();
    // スタート地点
    for (var i = 0; i < tasks.length; i++) {
        var task = tasks[i];
        promise = promise.then(task).then(pushValue);
    }
    return promise;
}
// 実行例
main().then(function (value) {
    console.log(value);
}).catch(function(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出来ます。

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 getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/
comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/
people.json').then(JSON.parse);
        }
    };
function main() {
    function recordValue(results, value) {
```

```
        results.push(value);
        return results;
    }
    var pushValue = recordValue.bind(null, []);
    var tasks = [request.comment, request.people];
    return tasks.reduce(function (promise, task) {
        return promise.then(task).then(pushValue);
    }, Promise.resolve());
}
// 実行例
main().then(function (value) {
    console.log(value);
}).catch(function(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を繋げることが出来ます。

`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[93]
- azu / Array.prototype.reduce Dance - Glide[94]

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良いので覚えておくといいのですが、 初めて見た時にどういう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。

93 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce
94 http://glide.so/azu/6919649

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事が関数名から見てわかるようになります。

```
var 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) {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    var pushValue = recordValue.bind(null, []);
    return tasks.reduce(function (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。

何故、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既に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;
    }
```

```
    var pushValue = recordValue.bind(null, []);
    return tasks.reduce(function (promise, task) {
        return promise.then(task).then(pushValue);
    }, Promise.resolve());
}
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, true);
        req.onload = function () {
            if (req.status === 200) {
                resolve(req.responseText);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/
comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/
people.json').then(JSON.parse);
        }
    };
function main() {
    return sequenceTasks([request.comment, request.people]);
}
// 実行例
main().then(function (value) {
    console.log(value);
}).catch(function(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、逐次処理という事をするのに色々な書き方が出来ると思います。

- 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

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言えるでしょう。

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事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

Promises API Reference

Promise#then

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

thenコード例

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

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

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

Promise.resolve

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

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

Promise.resolveのコード例

```
var taskName = "task 1"
asyncTask(taskName).then(function (value) {
    console.log(value);
}).catch(function (error) {
    console.error(error);
});
function asyncTask(name){
    return Promise.resolve(name).then(function(value){
        return "Done! "+ value;
    });
}
```

受け取った値に応じたpromiseオブジェクトを返す。

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

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

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

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

Promise.reject

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

Promise.rejectのコード例

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

受け取った値でrejectされた新たなpromiseオブジェクトを返す。

Promise.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。

```
var 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

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

新たなpromiseオブジェクトを作成して返す。

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

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

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

Promise.race

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

Promise.raceのコード例

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

新たなpromiseオブジェクトを作成して返す。

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

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事

ES6 Promises
: ECMAScript 6th Edition[95] を明示的に示す場合にprefixとして ES6 をつける

Promises/A+
: Promises/A+[96]の事。 ES6 Promisesの前身となったコミュニティベースの仕様であり、ES6 Promisesとは多くの部分が共通している。

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

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

参考サイト

w3ctag/promises-guide[97]
: 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[98]
: ES6 Promisesの仕様リポジトリ - issueを検索して得た経緯や情報も多い

ECMAScript Language Specification ECMA-262 6th Edition – DRAFT[99]
: ES6 Promisesの仕様書 - 仕様書として参照する場合はこちらを優先した

[95] http://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects
[96] http://promises-aplus.github.io/promises-spec/
[97] https://github.com/w3ctag/promises-guide
[98] https://github.com/domenic/promises-unwrapping
[99] http://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects

JavaScript Promises: There and back again - HTML5 Rocks[100]
: 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[101]
: Node.jsとPromiseの記事 - thenableについて参考にした

著者について

azu[102] (Twitter : @azu_re[103])

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[104] や JSer.info[105] 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[106] 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。

- Gumroad ¥0

  JavaScript Promiseの本のおまけ[107]

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出来ます。

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出来ます。

100 http://www.html5rocks.com/ja/tutorials/es6/promises/
101 http://d.hatena.ne.jp/jovi0608/20140319/1395199285
102 https://github.com/azu/
103 https://twitter.com/azu_re
104 http://efcl.info/
105 http://jser.info/
106 https://gumroad.com/l/javascript-promise
107 https://gumroad.com/l/javascript-promise

- Issues · azu/promises-book[108]
- azu/promises-book - Gitter[109]

[108] https://github.com/azu/promises-book/issues?state=open
[109] https://gitter.im/azu/promises-book