[WIP] Promises Book

azu(@azu) [FAMILY Given]

目次

図の一覧

> Working on the book. Contributingは Github[1] から

[1] https://github.com/azu/promises-book

第1章 Chapter.1 - Promisesとは何か

Promisesとは何かを初めて簡単な概要の紹介です。

1.1. What Is Promises

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

1.2. Promises Overview

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

多く分けて以下の3種類になります。

Constructor

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

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

Instance Method

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

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

resolve(解決)された時
`onFulfilled` が呼ばれる

reject(棄却)された時
`onRejected` が呼ばれる

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

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

Static Method

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

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

1.2.1. Promises workflow

以下のようなサンプルコードを見てみましょう。

```
'use strict';
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オブジェクトに足しして `then` でresolveされた時のコールバックを、 `catch` でエラーとなった場合のコールバックを設定しています。

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

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

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

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

1.2.2. Promisesの状態

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

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

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

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

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

見方ですが、 左が ES6Promises[1]で定められている名前で、 右が Promises/A+[2]で登場する状態の名前になっています。

基本的にこの状態をプログラムで直接触る事はないため、名前自体は余り気にしなくても問題ないです。 この文章では、 Promises/A+[3] の Pending、Fulfilled 、Rejected を用いて解説していきます。

[1] http://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects
[2] http://promises-aplus.github.io/promises-spec/
[3] http://promises-aplus.github.io/promises-spec/

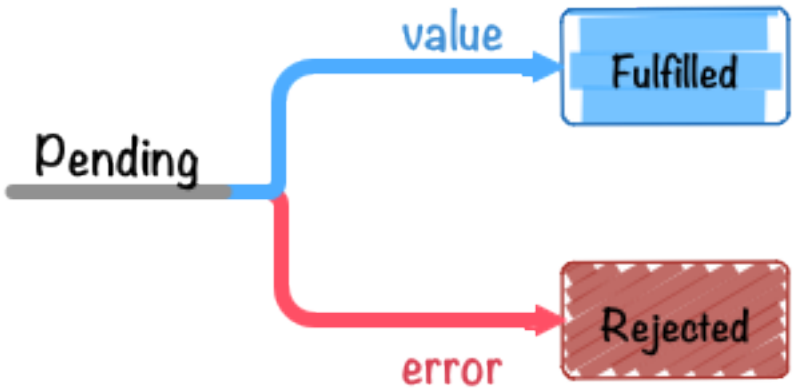

図1.1 promise states

ECMAScript Language Specification ECMA-262 6th Edition - DRAFT[4] 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わかります。

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

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

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

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

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

[4] http://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects

JavaScript Promises - Thinking Sync in an Async World // Speaker Deck[5] というスライドではPromisesの状態遷移について分かりやすく書かれています。

1.3. Promisesの書き方

Promiseの基本的な書き方について解説します。

1.3.1. promiseオブジェクトの作成

promiseオブジェクトを作る流れは以下のようになっています。

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

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

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

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

XHRで200で値が取得できた場合のみ `resolve` として、 それ以外はエラーであるとして `reject` しています。

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

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

次に、 `reject` の方を見て行きましょう。

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

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

`reject` に渡す値はrejectする理由を書いたErrorオブジェクトとなっています。 今回は、ステータスコードが200以外であるならrejectするとしていたため、 `reject` にはstatusTextを渡しています。 （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）

1.3.2. promiseオブジェクトに処理を書く

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

```
getURL("http://example.com/"); // => promiseオブジェクト
```

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

promiseオブジェクトに登録する処理は以下の2種類が主となります

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

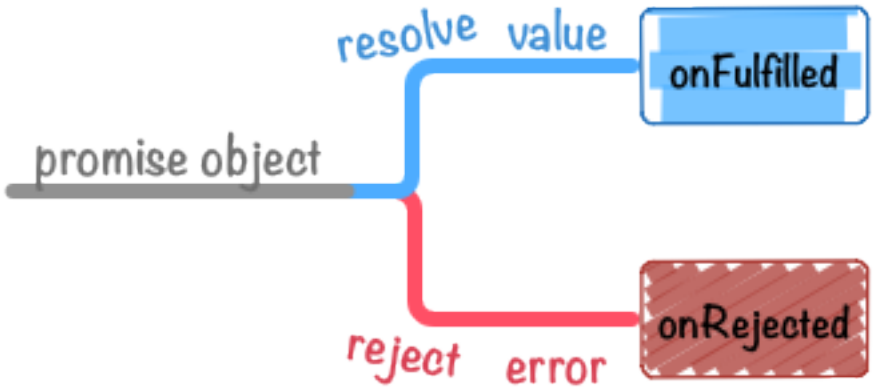

図1.2 promise value flow

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

この場合の 通信が成功した というのは promiseオブジェクトがresolveされた 時という事

resolveされた時の処理は、`.then`メソッドに処理をする関数を渡すことで行えます。

```
var URL = "http://localhost:3000/?status=200&body=text";
getURL(URL).then(function onFulfilled(value){ ❶
    console.log(value);
});
```

❶ 説明しやすくするためコールバック関数に`onFulfilled`という名前を付けています

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

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

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

rejectされた時の処理は、`.then`の第二引数または`.catch`メソッドに処理をする関数を渡す事で行えます。

先ほどのソースに

```
var URL = "http://localhost:3000/?status=500";
getURL(URL).then(function onFulfilled(value){
    console.log(value);
}).catch(function onRejected(error){ ❶
    console.log(error);
});
```

❶　説明しやすくするためエラーが起きた時に呼ばれる関数に `onRejected` という名前を付けています

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

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

```
getURL(URL).then(onFulfilled, onRejected);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ますが、 両者の違いについては thenとcatchの違い で紹介します。

まとめ

この章では以下のことについて簡単に紹介しました。

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

Promisesの基本的な書き方はこれがベースとなり、 他の多くの処理はこれを発展させたり、用意されたStatic Method等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Promisesで書くメリットについては触れていませんでしたが、 次の章では、Promiseのメリットであるエラーハンドリングの仕組みをコールバックベースの実装と比較しながら見て行きたいと思います。

第2章 Chapter.2 - Promisesの書き方

この章では、Promisesのメソッドについて用途や使い方、エラーハンドリングについて学びます。

2.1. promiseと配列

先ほどの章では、promiseオブジェクトがresolve又はrejectされた時の処理は `.then` と `.catch` を 使うことで行える事を学びました。

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オブジェクトに処理を書く場合はどうすればよいでしょうか?

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

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

2.1.1. コールバックで複数の非同期処理

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

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

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

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

このコールバックスタイルでは幾つかの要素が出てきます。

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

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

```
jsonParse.bind(null, callback);
// は以下のように置き換えるのと殆ど同じ
function(error,value){
    jsonParse(callback, error, value);
}
```

コールバックスタイルで書いたものを見ると以下のような点が気になります。

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

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

2.1.2. promise.thenのみで複数の非同期処理

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

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

```
'use strict';
function getURL(URL) {
    return new Promise(function (resolve, reject) {
        var req = new XMLHttpRequest();
        req.open('GET', URL, false);
```

```
        req.onload = function () {
            if (req.status == 200) {
                resolve(req.response);
            } else {
                reject(new Error(req.statusText));
            }
        };
        req.onerror = function () {
            reject(new Error(req.statusText));
        };
        req.send();
    });
}
var request = {
        comment: function getComment() {
            return getURL('http://azu.github.io/promises-book/json/
comment.json').then(JSON.parse);
        },
        people: function getPeople() {
            return getURL('http://azu.github.io/promises-book/json/
people.json').then(JSON.parse);
        }
    };
function main() {
    function recordValue(results, value) {
        results.push(value);
        return results;
    }
    // [] は記録する初期値を部分適応してる
    var pushValue = recordValue.bind(null, []);
    return request.comment().then(pushValue).then(request.people).then(pushValue);
}
```

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

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

コールバックスタイルと比較してみると次の事がわかります。

- `JSON.parse` をそのまま使っている

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

先ほども述べたように mainの `then` の部分がクドく感じます。

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

2.2. Promise.all

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

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

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

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

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

実行方法は 前回のもの と同じで以下のようにして実行出来ます。

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

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

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

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

というように処理を書いた場合は、`request.comment()` と `request.people()` は同時に実行されますが、　それぞれのpromiseの結果(resolve,rejectで渡される値)は、`Promise.all`に渡した配列の順番となります。

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

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

```
'use strict';
// 配列の中身をそれぞれpromiseオブジェクトにした配列を返す
function promisedMapping(ary) {
```

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

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

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

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

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

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

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

逐次的に実行した場合は、 ??? で紹介したような、 `.then` を重ねていくような書き方が必要になる。

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

□ Promise.reduce 的な機能の紹介

2.3. 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渡します。

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

Promise.allの時と同じタイマーを使った例を見てみましょう

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

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

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

```
'use strict';
var winnerPromise = new Promise(function (resolve) {
        setTimeout(function () {
            console.log('this is winner');
            resolve('this is winner');
```

```
        }, 4);
    });
var loserPromise = new Promise(function (resolve) {
        setTimeout(function () {
            console.log('this is loser');
            resolve('this is loser');
        }, 1000);
    });
// 一番最初のものがresolveされた時点で終了
Promise.race([winnerPromise, loserPromise]).then(function (value) {
    console.log(value);    // => 1
});
```

先ほどのコードに `console.log` をそれぞれ追加しただけの内容です。

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

つまり、`Promise.race`では、 一番最初のpromiseがresolveしても他の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。

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

2.4. Promise.resolve

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

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

2.4.1. new Promiseのショートカット

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

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

```
new Promise(function(resolve){
```

```
    resolve(42);
});
```

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

`Promise.resolve(value)` の返す値も同様にpromiseオブジェクトなので、 以下のように `.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);
});
```

2.4.2. ThenableとPromise.resolve

Promise.resolveは `new Promise()` のショートカットとして使う事が出来るのは、 テストコードを書く際にも活用できますが、もう一つ大きな特徴を持っています。

ES6 PromisesにはThenableという概念があり、 thenableとは簡単にいえばpromiseっぽいオブジェクトの事を言います。 `.length` を持つけど配列じゃないものをArray likeとかというのと同じような話となります。

Thenableの場合は `.then` というメソッドを持ってるオブジェクトのことを言います。

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。

Web Notificationsをthenableにする

Web Notifications[1]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[2]
- Can I use Web Notifications[3]

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。

[1] https://developer.mozilla.org/ja/docs/Web/API/notification
[2] https://developer.mozilla.org/ja/docs/WebAPI/Using_Web_Notifications
[3] http://caniuse.com/notifications

```
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。

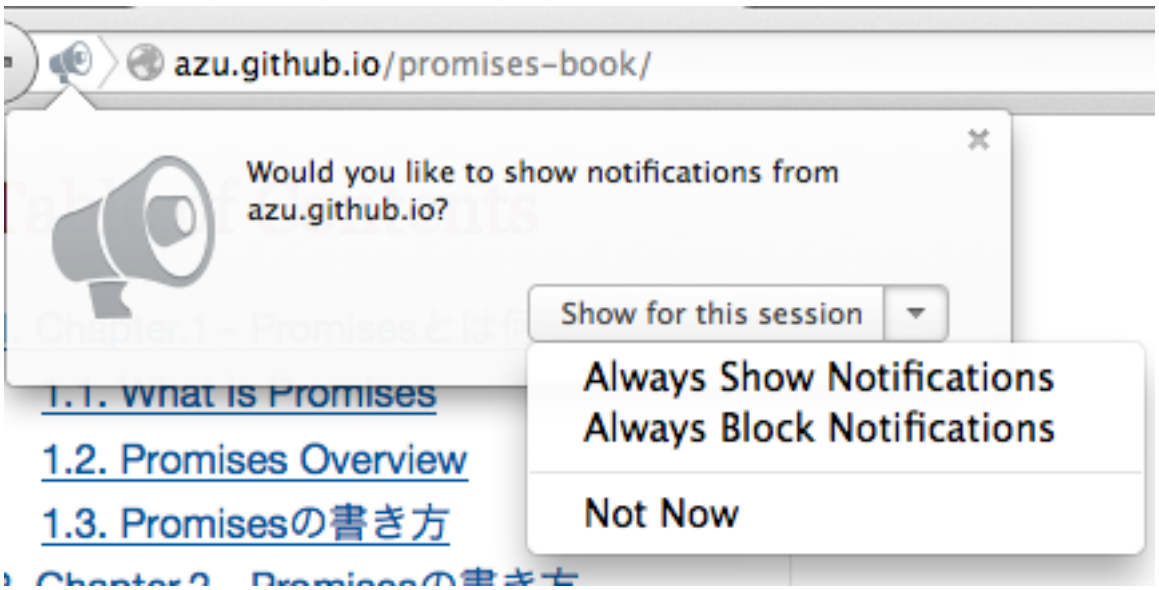

図2.1 Notificationの許可ダイアログ

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

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

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

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

不許可時("denied")
何もしない

まとめると以下のようになります。

- ユーザーに通知の許可を受け付ける非同期処理がある
- 許可がある場合は`new Notification`で通知を表示できる

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

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

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

Web Notification as Promise

notification-as-promise.js

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

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

```
notifyMessage("Hi!").then(function (notification) {
    console.log(notification);// 通知のオブジェクト
```

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

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

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

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

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

- `Promise`があることを前提とする
  - 利用者に`Promise`があることを保証してもらう
- ライブラリ自体に`Promise`の実装を入れてしまう
  - 例）localForage[4]
- コールバックでも使う事ができ、`Promise`でも使えるようにする
  - 利用者がどちらを使うかを選択出来るようにする

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

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

2.4.3. Thenableでコールバックと両立する

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

notification-callback.js

```
'use strict';
function notifyMessage(message, options, callback) {
```

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

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

これを利用する場合は以下のような感じになります。

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

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

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

thenableを返すメソッドを追加する

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

notification-thenable.js

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

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

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

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

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

```
Promise.resolve(notifyMessage.thenable("message")).then(function (notification) {
    console.log(notification);// 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とは次のような違いがあります。

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

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

jQueryとthenable

jQuery.ajax()[5]の返り値も`.then`というメソッドを持ったオブジェクト(Deferred Object[6])です。

しかし、このDeferred ObjectはPromises/A+やES6 Promisesに則したものではないため、`Promise.resolve`では上手く扱えない場合があります。

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。

- You're Missing the Point of Promises[7]

[5] https://api.jquery.com/jQuery.ajax/
[6] http://api.jquery.com/category/deferred-object/
[7] http://domenic.me/2012/10/14/youre-missing-the-point-of-promises/

> - https://twitter.com/hirano_y_aa/status/398851806383452160

2.5. 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.log(error);
});
```

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

2.6. コラム: Promiseは常に非同期?

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

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

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

```
    console.log(value); ❷
});
console.log("outer promise");❸
```

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

```
inner promise
outer promise
42
```

つまり、Promisesは常に非同期で処理が行われているという事になります。

2.7. then or catch?

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

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

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

2.7.1. エラー処理ができないonRejected

次のようなコードを見ていきます。

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

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

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

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

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

`.then` や `.catch` によるchainは、それより前のpromiseオブジェクトに対しての処理を行なわれるため、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。

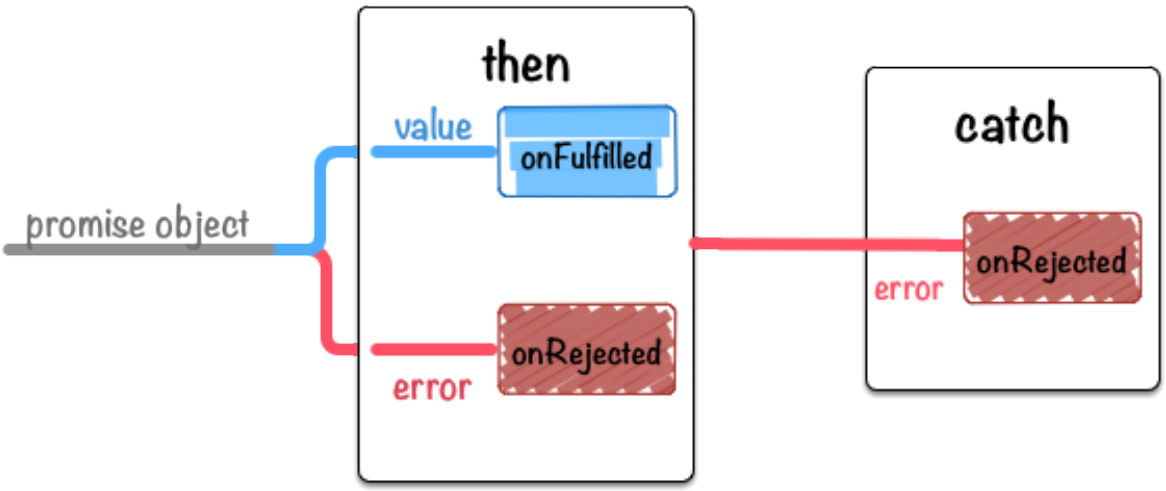

図2.2 Then Catch flow

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

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

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

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

2.7.2. まとめ

ここでは次のような事について学びました。

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

第3章 Chapter.3 - Promisesのテスト

この章ではPromisesのテストの書き方について学んで行きます。

3.1. 基本的なテスト

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

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

ここではまず、 Mocha[1]を使った基本的なPromiseのテストの書き方について学んでいきましょう。

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

- □ この書籍のソースコードへのリンク

3.1.1. Mochaとは

ここでは、 Mocha[2]自体については詳しく解説しませんが、 MochaはNode.js製のテストフレームワークツールです。

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

Mochaを選んだ理由としては、以下の点で選択しました。

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

最後の"Promises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。

[1] http://visionmedia.github.io/mocha/
[2] http://visionmedia.github.io/mocha/

また、アサーションライブラリには、 power-assert[3]を利用しますが、 アサーション自体はNode.jsの `assert` モジュールと全く同じであるため、今回はあまり気にしなくても問題ありません。

まずは、コールバック関数のテストと同じような形でテストを書いてみましょう。

3.1.2. コールバックスタイルのテスト

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

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

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

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

次のコールバックスタイルのテストは以下のような流れになっています。

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

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

❶❶ 非同期処理のコールバックを指定
❸❷ `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。

3.1.3. `done`を使ったPromiseのテスト

次に、Promiseのテストの方を見てみましょう。

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

❶ `onFulfilled`を呼ぶ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、 そのpromiseオブジェクトはresolveされます。

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

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

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

```
});
```

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

図3.1 promise test timeout

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

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

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

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

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

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

Promiseのテストは普通のコールバックのテストと少し違いますが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。 毎回やるにはつけ忘れてしまうこともあるため、あまりテストしやすいとは言えないかもしれません。

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

3.2. MochaのPromises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[4] にもその概要が書かれています。

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

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

実際にどういう風に書くかの例を見て行きたいと思います。

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);
        });
    });
    function throwError(value) {
        throw new Error(value);
    }

    function mayBeRejected() {
        return Promise.reject(new Error("woo"));
    }

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

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。

[4] http://visionmedia.github.io/mocha/#asynchronous-code

変更点としては以下の2箇所です

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

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

```
it("should be fail", function () {
    return Promise.resolve().then(function () {
        assert(false);// => 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に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[5] という記事でも MochaのPromiseサポートについて書かれています。

3.2.1. `catch`によるテスト

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

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

このテストの目的

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

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

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

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

```
        assert(error.message === "woo");
    });
});
```

❶　この関数が返すpromiseオブジェクトをテストしたい
この場合は、`Promise.reject`は`onRejected`に登録された関数を呼ぶため、　テストはパスしますね。

このテストで問題になるのは`mayBeRejected()`で返されたpromiseオブジェクトがresolveされた場合に、必ずテストがパスしてしまうという問題が発生します。

```
function mayBeRejected(){ 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resolveする
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 throwError(value) { ❶
    throw new Error(value);
}
function mayBeRejected(){
    return Promise.resolve();
}
it("should bad pattern", function () {
    return mayBeRejected().then(throwError).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、　`throwError`で投げられたエラーが`catch`されてしまいます。

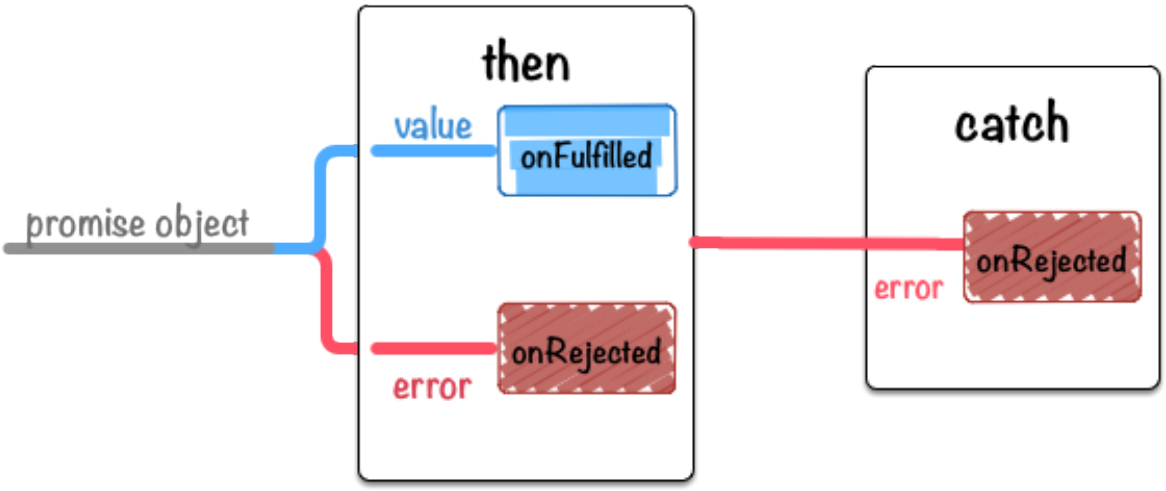

図3.2 Then Catch flow

`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。

3.2.2. `catch`によるテストの改善

上記のエラーをテストしたい場合は、どうすればよいでしょうか?

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

resolveした場合
　意図した通りテストが失敗する

rejectした場合
　`assert`でテストを行える

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

❶　resolveされた場合はテストは失敗する

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

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

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(onFulfilled, onRejected)` というように指定すると、より簡潔にテストを書くことが出来ます。

3.2.3. まとめ

- 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
  - エラーがMochaに届くようにエラーハンドリングは行わない

第4章 Promises API Reference

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

promiseオブジェクトがresolve 又は rejectされた時に呼ばれるメソッド

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

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

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

```
Promise.resolve(thenable)
Promise.resolve(value)
```

```
Promise.rejct(value)
```

第5章 用語集

Promises
: プロミスという仕様そのもの

Promise
: プロミスオブジェクト、インスタンス

ES6 Promises
: ECMAScript 6th Edition[1] を明示的に示す場合にprefixとして ES6 をつける

Promises/A+
: Promises/A+[2]の事。 ES6 Promisesのベースとなったコミュニティベースの仕様であり、ES6 Promisesとは多くの部分が共通している。

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

[1] http://people.mozilla.org/%7Ejorendorff/es6-draft.html#sec-operations-on-promise-objects
[2] http://promises-aplus.github.io/promises-spec/