

Table of Contents

Introduction	0
講座で利用する環境を構築する	1
Cloud9というクラウド上の環境を利用する	1.1
Mac/Windows上でNode.jsの環境を利用する	1.2
JavaScriptのわかりづらい概念について学ぶ	2
非同期処理について	2.1
非同期処理の概念について解説	2.1.1
サンプルコードを通じて非同期処理について理解する	2.1.2
非同期処理を考慮しない書き方	2.1.2.1
非同期処理を考慮する形に書き換える	2.1.2.2
非同期処理のコードにありがちな問題解決	2.1.2.3
jQueryのDeferred基本編	2.1.2.4
jQueryのDeferred応用編	2.1.2.5
jQueryのDeferred処理をおさらい	2.1.2.6
thisについて	2.2
アクティベーションオブジェクトとスコープチェーンについて	2.2.1
関数呼び出し時にthisが生成される	2.2.2
thisの呼ばれ方その1：トップレベルのthis	2.2.3
thisの呼ばれ方その2：コンストラクタ内のthis	2.2.4
thisの呼ばれ方その3：何かに所属している時のthis	2.2.5
thisの呼ばれ方その4：外部から書き換えられるthis	2.2.6
WebAPIと連携する処理に対してどのようにテストを書くか	3
Jasmineについて	3.1

このリポジトリについて

テストを書く上で必要な構成要素	3.1.1
Jasmineの特徴	3.1.2
Jasmineの実行	3.1.3
Jasmineを使ったコードを実際に書く	3.2
まずはJasmine自身に慣れてみる	3.2.1
fetch()メソッドが定義されてるというテストを書く	3.2.2
Gistクラスを定義したファイルを作成して最初のテストを成功させる	3.2.3
fetch()メソッドのテストを書く	3.2.4
最初に書いたテストを修正する	3.2.5
beforeEachとspyOnについて	3.2.6
APIからJSONが得られる記述の説明	3.2.7
WebAPI連携のテストを書く	3.3
JSONPlaceholderとは？	3.3.1
実装するクラスの仕様を考える	3.3.2
クラスを定義して最初のテストを書く	3.3.3
投稿情報一覧を取得する処理を実装する	3.3.4
特定の投稿情報を取得する処理を実装する	3.3.5
APIアクセス部分をリファクタリング	3.3.6
投稿を作成する処理を実装する	3.3.7
付録	4
変数について	4.1
if文について	4.2
if文についてさらに詳しく知る	4.2.1
JavaScriptクラス定義をPHPのそれと対比して説明	4.3
基本的なクラス定義の方法	4.3.1
インスタンス化してパブリックな変数をターミナル上に表示	

このリポジトリについて

する	4.3.2
パブリックなメソッドを定義する	4.3.3
プライベートな変数とその変数を得るためのメソッドを定義	4.3.4

このリポジトリについて

[JavaScriptステップアップ勉強会](#)で利用するサンプルコードと説明の時に
利用する資料一式になります

License

The MIT License (MIT) Copyright (c) 2016 Hiroshi Oyamada

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

講座で利用する環境を構築する

ハンズオン形式で講座を進めていきますので、事前に実行環境の準備をお願いしたいと思います。

やっていただく作業は

1. GitHubから必要なファイル一式をダウンロード
2. Cloud9というクラウド上の環境を利用する、もしくは普段からNode.js環境を使ってる方はそちらの環境を利用した環境構築

の作業をお願いします

GitHubから必要なファイル一式を取り込む

ターミナルを起動して開発マシン上の任意のディレクトリで以下コマンドを実行してください。

```
git clone git@github.com:h5y1m141/step-up-javascript.git
```

個々の環境に合わせた作業

JavaScriptの実行環境としてNode.jsを想定しておりますので、以下いづれかの方法で環境を構築しておいてください GitHubから必要なファイルをダウンロードしたら、それぞれの環境に応じて以下の作業を事前にしておいてください。

Node.jsの環境を全く作ったことが無い方

[Cloud9というクラウド上の環境を利用する](#)を参考にして環境を構築してください。

普段からNode.js環境を利用する方

講座をすすめる上で必要なnpmモジュールがインストールできるように package.jsonを準備しておりますので、

```
npm install
```

を実施すればOKです。

[Mac/Windows上でNode.jsの環境を利用する参考に準備をお願いします。](#)

Cloud9というクラウド上の環境を利用する

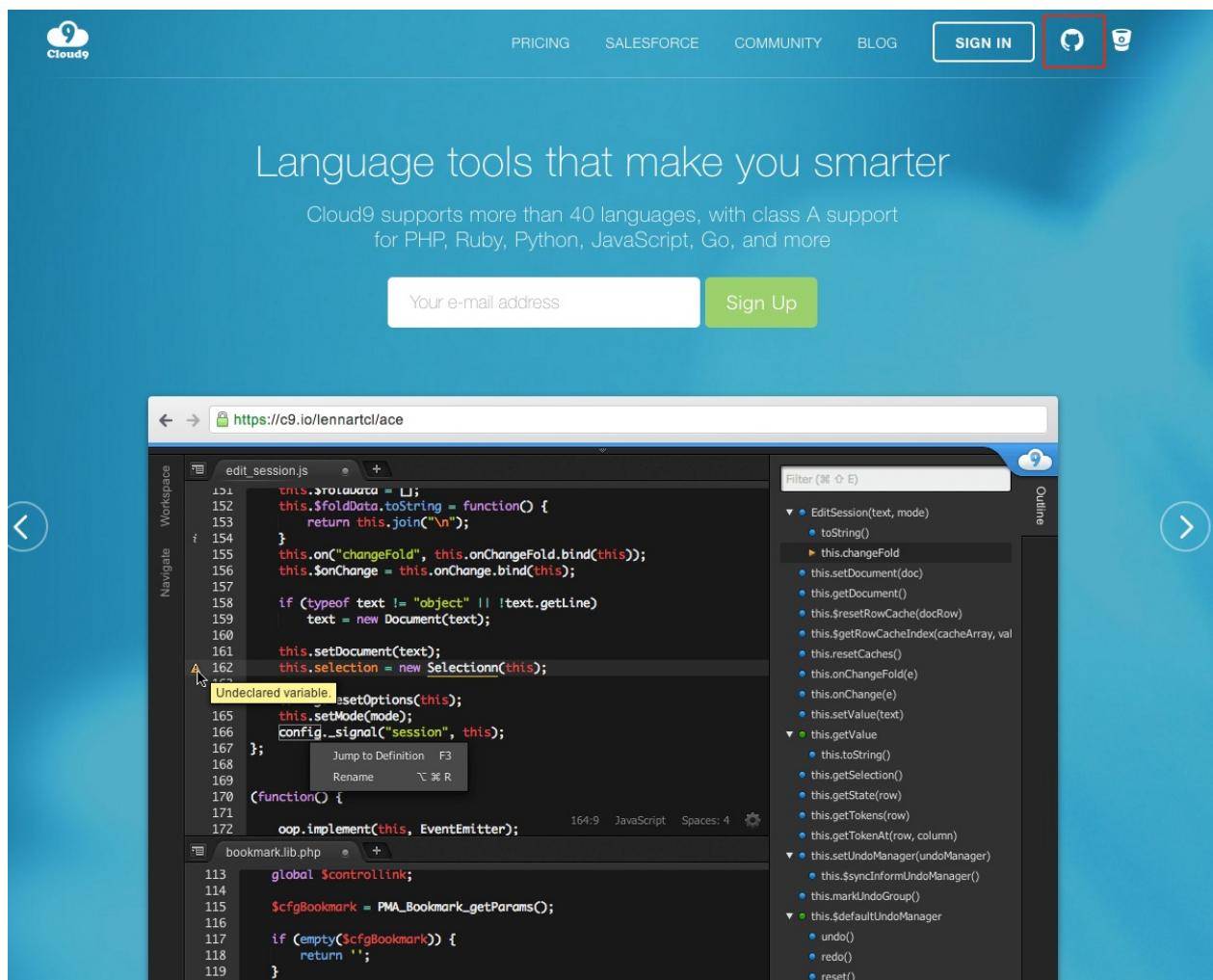
Cloud9はメニューが英語なのでその点で少しあつづらいかもしれません、Node.jsの実行環境だけではなく開発時のエディタ機能も備わってますので、余計な環境構築に時間がとられること無く、開発自体に専念できる環境かと思います。

Cloud9に登録する

以下サイトにアクセスします

<https://c9.io/>

画面右上にあるGitHubアイコンをクリックすることで、GitHubアカウントでの登録も可能で基本的にはそちらでの登録をオススメします。



Cloud9を利用して作業を開始する

アカウントの登録が完了したら、Node.jsの実行環境を作成するためにワークスペース（workspace）を設定します

このリポジトリについて

Create a new workspace

Workspace name Description

Hosted workspace Clone workspace Remote SSH workspace

 Private
This is a workspace for your eyes only

 Public
This will create a workspace for everybody to see

Clone from Git or Mercurial URL (optional)

Choose a template

 HTML5  Node.js  Meteor  PHP, Apache &...  Python  Django

 Ruby  C++  Wordpress  Rails Tutorial  Blank  Harvard Univ...

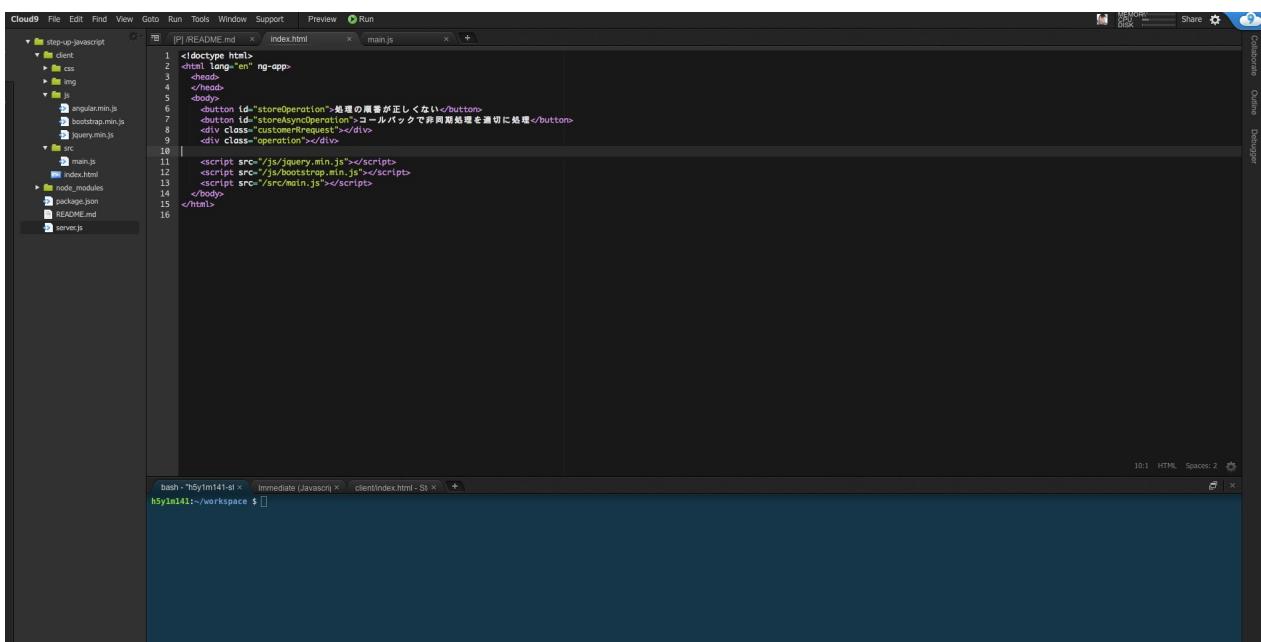
上記スクリーンショットの赤枠でも囲っていますが、基本的には

- Workspace nameを設定
 - 例えば**step-up-javascript**のようにして任意の名前をつけます
- Templateを選択
 - Node.jsのアイコンがあるのでそれを選択

を行った上で、Create workspaceのボタンをクリックすれば環境構築が完了します。

環境が出来上がるまで少し待ちますが、構築が完了すると以下の様な開発時の統合開発環境の画面が表示されます。

このリポジトリについて



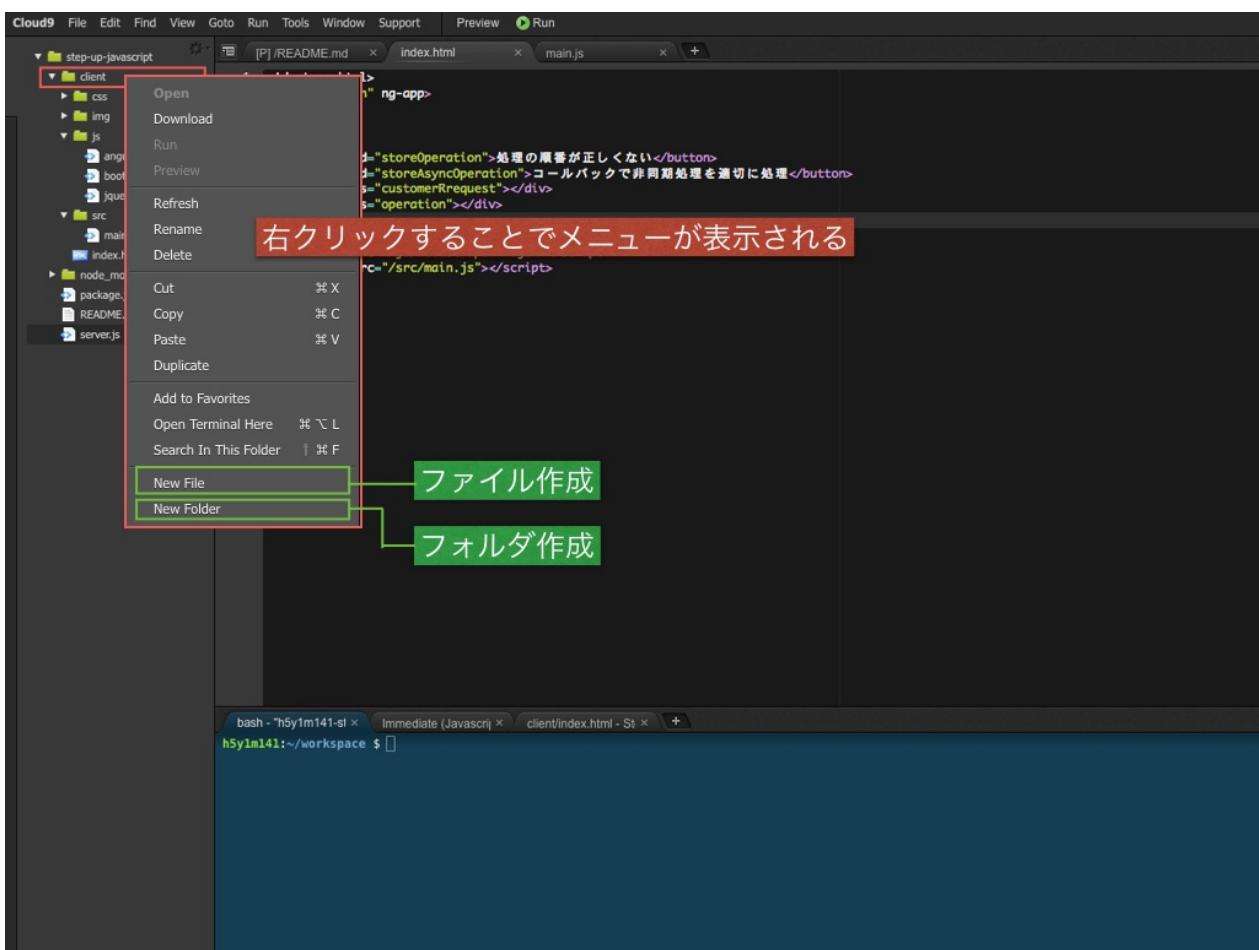
開発時の基本操作

Webブラウザ上で表示・動作確認できるようなサンプルを中心に資料を作っていますが、動作確認などでNode.jsのサーバー機能を利用するケースも出てくるので、Cloud9を利用した開発時に利用しそうな機能について簡単に記載しておきます

ファイル・フォルダを作る

左側のメニューの任意の箇所を右クリックすること以下の様なメニューが表示されるので、

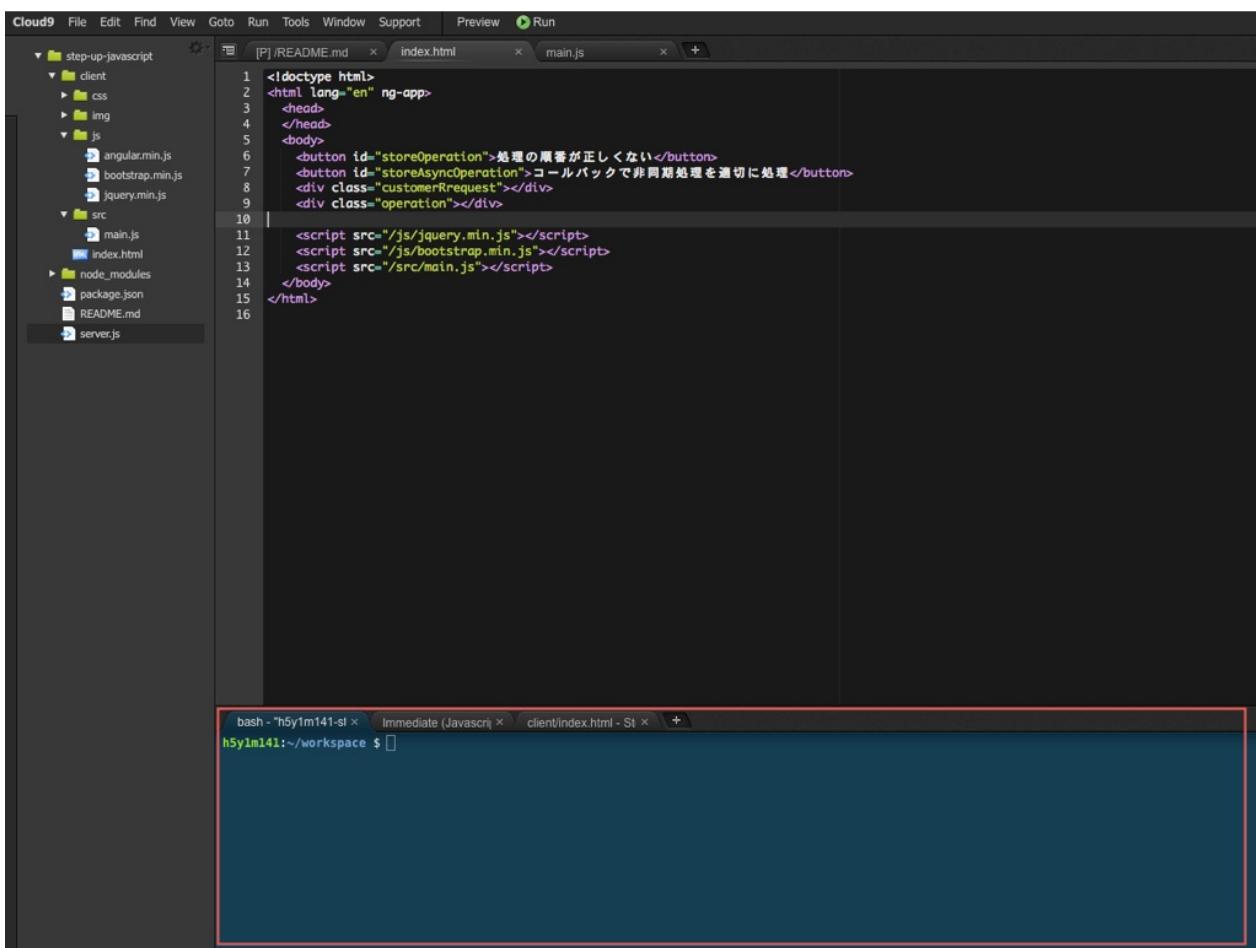
このリポジトリについて



コマンドを入力してサーバー機能を立ち上げる

画面下部にコマンドを入力するためのターミナルが表示されています

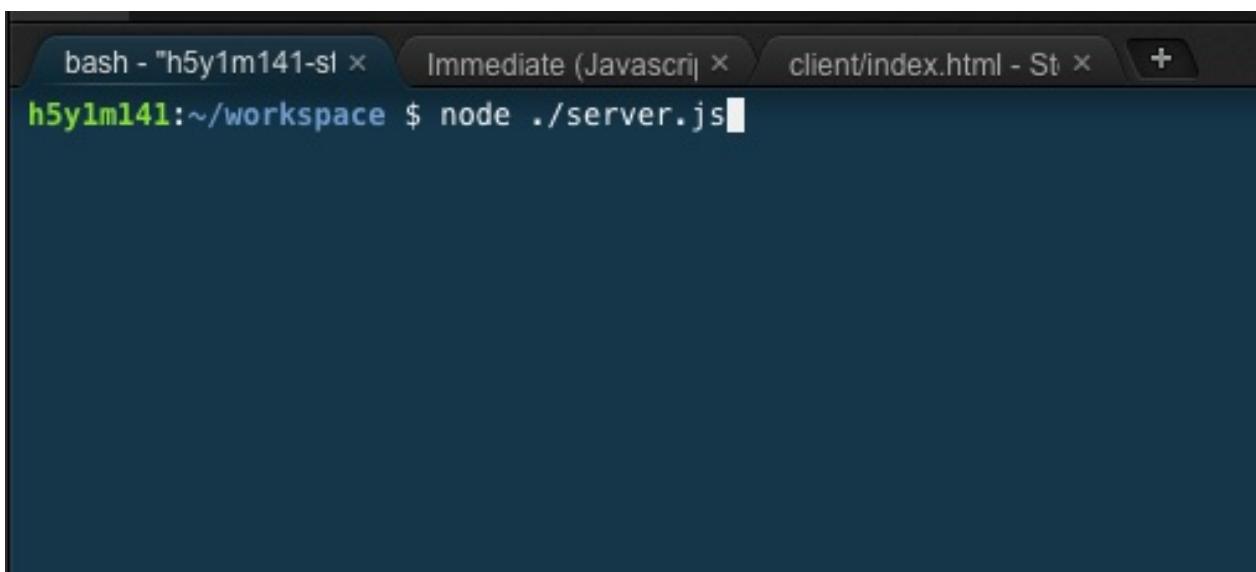
このリポジトリについて



上記画面の赤枠で囲った箇所のタブで**bash**という表示の箇所をクリックすると以下の様にコマンドを入力することができますので

```
node ./server.js
```

と入力をします。



このリポジトリについて

入力完了すると以下の様な表示になります

The screenshot shows the Cloud9 IDE interface. On the left is the workspace sidebar with project files like 'step-up-javascript', 'client', 'js', 'src', and 'node_modules'. The main area has tabs for 'index.html' and 'main.js'. Below is a terminal window with the following log output:

```
node -h5y1m141-s ~ Immediate (JavaScript) client/index.html - St
h5y1m141:~/workspace $ node ./server.js
info  - socket.io started
Chat server listening at 0.0.0.0:8080
```

A context menu is open over the last line of the log, showing options: 'Open', 'Open In Preview', and 'Copy'. A red box highlights this menu.

Chat server listening at 0.0.0.0:8080

という表示の最後の部分の**0.0.0.0:8080**にマウスカーソルを合わせるとクリック出来る状態になりますのでそこをクリックすると、小さいウィンドウが表示されるので**open**を選択すると、browserの別ウィンドウが表示されます。

これを行うことで簡単にWebサーバー機能を利用することが出来て、自分で作ったJavaScriptの動作確認が行えるので基本的には開発時にはこの機能を活用しながら作業することをオススメします。

Mac/Windows上でNode.jsの環境を利用する

普段からNode.js環境を利用してる方の場合には、npmが利用できる状況かと思いますので、以下手順での環境構築が手軽なのでこちらでもOKです。

動作確認環境

以下のとおりです。

- Mac OS X Yosemite (10.10.5)
- Mac OS X El Capitan (10.11.4)
- Node.js
 - v4.2.4

Nodeのバージョンについて

Node.js v4とNode.js v5の違いはNode.js 日本ユーザーグループ代表の方が書いた以下記事が参考になります。

Node.js v4はLTSというリリースしてから2年半サポートするサポートポリシーがついていますが、Node.js v5にはついていません。つまり、Node.js v5は次のバージョンが出たらサポートされなくなります。今のところ、LTSの対象になるのは偶数のバージョンとされています。ただし、偶数のバージョンが必ずLTSというわけではなく、今のところ偶然そういうバージョンになっているというのが正しい状態なので、きちんとLTSかどうかを把握するためには、バージョン番号の他にLTS識別名(ArgonやBoron等の元素名)が付いていることを確認したほうがよいです。

今からでも間に合う！Node.js v4 & v5は何が変わったか？

Node.jsのインストール方法

Macの場合には色々あるかもしれません、私はNode.jsはnodebrewを利用してインストールしてます。

```
nodebrew -v  
nodebrew 0.9.2
```

Usage:

nodebrew help	Show this message
nodebrew install <version>	Download and install a specific version
nodebrew install-binary <version>	Download and install a specific binary
nodebrew uninstall <version>	Uninstall a version
nodebrew use <version>	Use <version>
nodebrew list	List installed versions
nodebrew ls	Alias for `list`
nodebrew ls-remote	List remote versions
nodebrew ls-all	List remote and installed versions
nodebrew alias <key> <version>	Set alias to version
nodebrew unalias <key>	Remove alias
nodebrew clean <version> all	Remove source file
nodebrew selfupdate	Update nodebrew
nodebrew migrate-package <version>	Install global NPM packages
nodebrew exec <version> -- <command>	Execute <command> specifically for this version

Example:

```
# install from binary  
nodebrew install-binary v0.10.22  
  
# use a specific version number  
nodebrew use v0.10.22  
  
# io.js  
nodebrew install-binary io@v1.0.0  
nodebrew use io@v1.0.0
```

環境構築する

いつも通りに

npm install

していただければOKです

参考までに作業時の画面キャプチャを以下貼っておきます

```
last login: Tue Apr 12 12:44:36 on ttys019
- ~ - node
$ cd /Documents/step-up-javascript
$ git clone git@github.com:5ymlm41/step-up-javascript.git
remote: Counting objects: 279, done.
remote: Compressing objects: 100% (213/213), done.
remote: Writing objects: 100% (279/279), done: pack-reused 0
remote: Total 279 (delta 136), reused 184 (delta 47), pack-reused 0
Receiving objects: 100% (279/279), 1.84 MB | 986.00 KB/s, done.
Resolving deltas: 100% (184/184), done.
Checking connectivity... done.
$ cd step-up-javascript
$ npm install
npm WARN package.json step-up-javascript@0.0.0 No repository field.
npm WARN peerDependencies manifest. The peer dependency requirejs@2.1.0 included from karma-requirejs will no longer be automatically installed to fulfill the peerDependency requirement.
npm WARN peerDependencies Longer than expected peerDependencies field.
Your application will need to depend on it explicitly.
npm WARN deprecated graceful-fs@0.8. Please update to graceful-fs@4.0.0 as soon as possible.
npm WARN deprecated lodash@0.9.2: Grunt needs your help! See https://github.com/gruntjs/grunt/issues/1465.
npm WARN deprecated minimatch@0.4.1: minimatch 0.4.1 has a known memory leak that prevents it from
npm WARN deprecated graceful-fs@0.1.3: graceful-fs version 3 and before will fail on newer node releases. Please update to graceful-fs@4.0.0 as soon as possible.
npm WARN deprecated graceful-fs@0.2.3: graceful-fs version 3 and before will fail on newer node releases. Please update to graceful-fs@4.0.0 as soon as possible.
npm WARN deprecated rtmpdump@0.1.2: this package has been reintegrated into npm and is now out of date with respect to npm

> fsevents@1.0.11 install /Users/noyamodo/Documents/step-up-javascript/node_modules/fsevents
> node-pre-gyp install --fallback-to-build

[fsevents] Success: "/Users/noyamodo/Documents/step-up-javascript/node_modules/fsevents/lib/binding/Release/node-v46-darwin-x64/fse.node" is installed via remote

> karma-jasmine-jquery@0.1.1 postinstall /Users/noyamodo/Documents/step-up-javascript/node_modules/karma-jasmine-jquery
> node install.js

underscore@1.8.3 node_modules/underscore
karma-jasmine@0.3.8 node_modules/karma-jasmine
karma-spec-reporter@0.0.22 node_modules/karma-spec-reporter
  └── colors@0.2.2
karma-jasmine-html-reporter@0.1.8 node_modules/karma-jasmine-html-reporter
  └── karma-jasmine@0.2.3
jasmine-core@2.4.1 node_modules/jasmine-core
  └── requirejs@2.2.0 node_modules/requirejs
karma-requirejs@0.2.6 node_modules/karma-requirejs
  └── karma-chrome-launcher@0.2.3 node_modules/karma-chrome-launcher
    ├── fs-access@1.0.0 (null-check@0.1.0)
    └── which@2.2.4 (isexe@1.1.2, is-absolute@0.1.7)
jquery@2.2.3 node_modules/jquery
  └── gulp-webserver@0.9.1 node_modules/gulp-webserver
    ├── isatty@0.0.1
    ├── proxy@0.1.0
    ├── browserify-livereload@0.5.1
    ├── open@0.0.2
    └── watch@0.11.0
      └── node-extensible@1.0.0 (yaml@1.0.0)
        └── thru@3.5.1 (extensible@3.0.0, readable-stream@2.0.3)
    └── browserify@8.4.1 (util-is-merge@0.9.0, parser@1.1.1, debug@2.8.0, finalhandler@0.4.1)
```

npm install 完了したら、gulpのweb server機能が動作するか確認

以下コマンドでgulpのweb server機能が動作するか確認してください

```
./node_modules/gulp/bin/gulp.js
```

このリポジトリについて

```
› step-up-javascript git:(develop) ./node_modules/gulp/bin/gulp.js
[16:54:30] Using gulpfile /Documents/step-up-javascript/gulpfile.js
[16:54:30] Starting 'webserver'...
[16:54:30] Webserver started at http://localhost:9000
[16:54:30] Finished 'webserver' after 16 ms
[16:54:30] Starting 'default'...
[16:54:30] Finished 'default' after 7.94 µs
```

JavaScriptのわかりづらい概念について学ぶ

私自身の経験を振り返ると、以下3つの理解が進まないとJavaScriptでのプログラミングの苦手意識が払拭出来ないと思っています。

- 非同期処理の概念
- thisについて
- JavaScriptにおけるクラス定義
 - [Google流 JavaScript におけるクラス定義の実現方法](#)

それぞれの項目についてこれ以降順番に解説していきます。

非同期処理について

JavaScriptを使って、Webブラウザ内での処理についてのJavaScriptでのプログラミングや、サーバーサイドでNode.jsを使ってJavaScriptでプログラミングする場合どちらでも非同期処理の概念を把握してないと、思ったような実装ができずつまづきやすいかと思います。

そのため、比較的身近な状況を例にして、同期処理、非同期処理について解説して、その内容を踏まて、非同期処理について少し掘り下げて解説していくこうと思います。

非同期処理の概念について解説

サブウェイとドトールコーヒーでのオペレーションを例にしながら、同期・非同期処理を対比させながら、非同期処理について説明します。

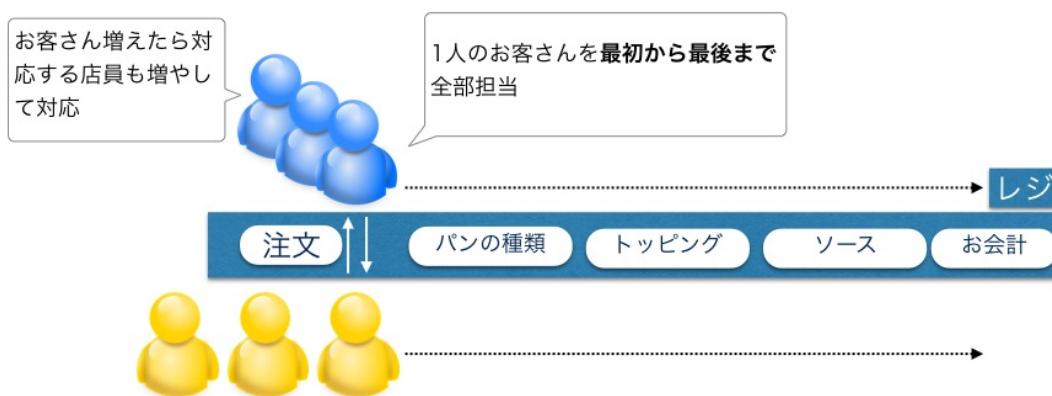
同期処理について

サブウェイに行った場合の商品注文方法によると、お客様は店舗で

- メニューの中からサンドイッチを決める
- パンを選ぶ
- お好みで野菜の量を増減
- ドレッシング・ソースをかける

という流れで店員さんに注文していきます。

注文から最終的なお会計までの流れを図にすると以下のようになります。

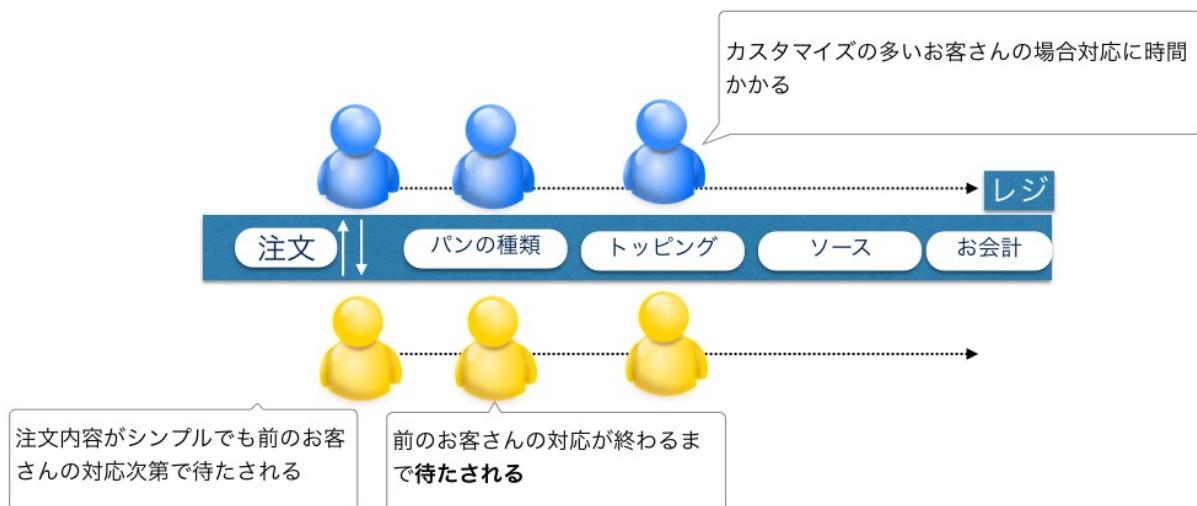


同期的に処理をする場合に、順番に処理がされていくので把握しやすいかと思います。

ただ以下の様な状況の時には問題が生じてしまいます。

- 例えばカスタマイズの多いお客様がいた場合にその対応に時間がかかる
- 上記対応が終わるまで、前のお客さんの対応が終わるまで待たされる
- (飲み物専用のレジがない限り) 飲み物だけのような注文内容がシ

フルな場合にも、そのお客様も前のお客さんの対応が終わるまで待たされる



同期処理の特徴としては以下の様な点かと思います。

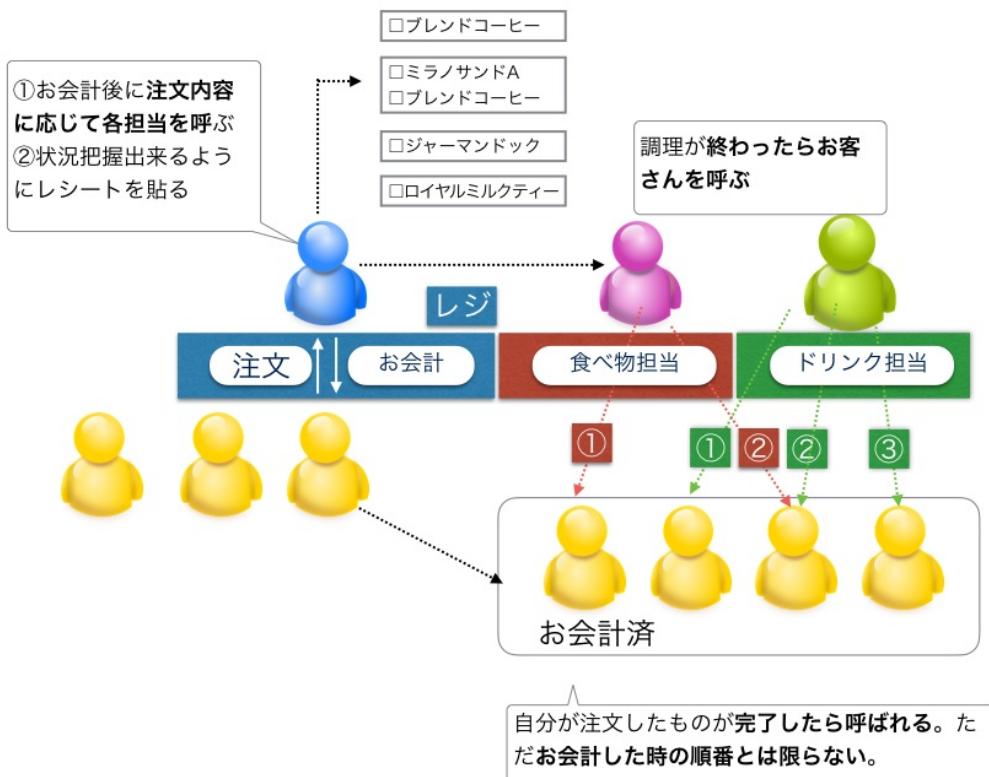
- 順番に処理がされるので全体で見ると処理内容が把握しやすい
- 特定の処理で時間がかかる場合にそれ以降の処理が待たされる

非同期処理について

サブウェイの場合には、注文からお会計までが一連の流れになっていたのに対して、ドトールコーヒーの場合には、

- 注文する
- お会計する
- 注文したものが出来上がったら呼ばれるので商品を受け取る

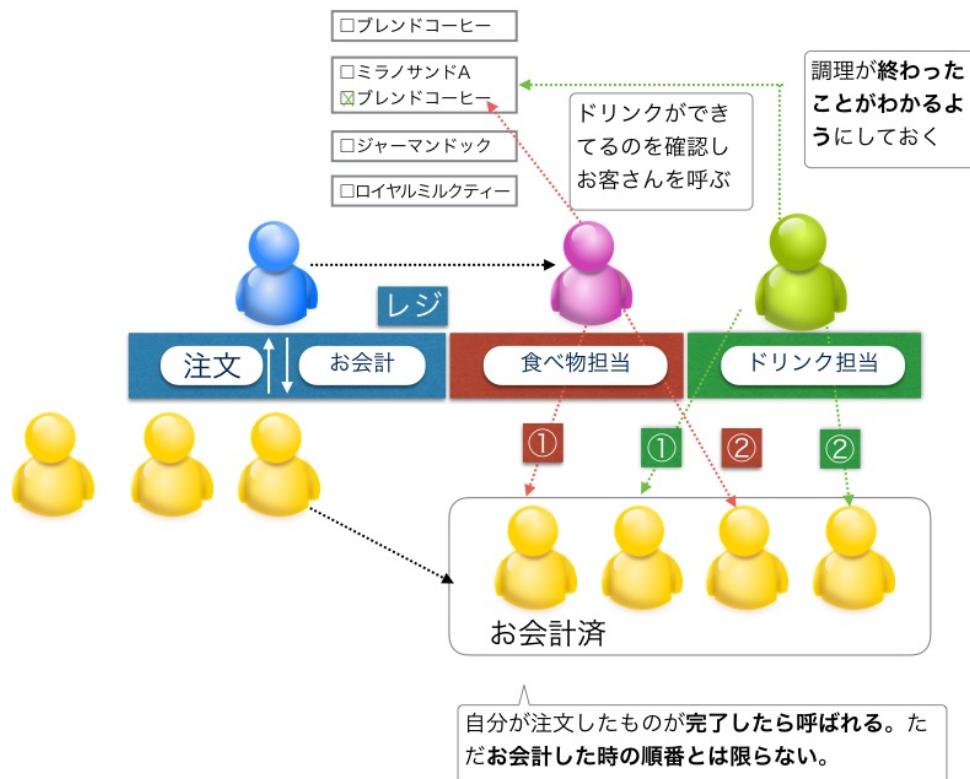
という形なり、注文から実際に商品を受け取るまでの流れを図にすると以下のようになります。



同期処理の場合だと、対応に時間かかるお客様がいるとそれ以降のお客さんは待たされましたが、非同期処理の場合には**処理が終わった順番に呼ばれる**のでそのような問題は生じません。

ただし**処理が一方通行ではないため全体で見ると処理内容が把握しづらくなる可能性が高くなります。**

例えば、上記の図で、飲み物と食べ物がそれぞれ出来上がったタイミングでお客さんを呼ぶようにしてましたが、例えばテイクアウトするお客様の対応の場合にはこのようなオペレーションはできなくなるため、2つが出来上がったタイミングでまとめて呼ばないといけなくなります。



非同期処理の場合には、状態を確認→何らかの処理ということが生じやすくなるのが特徴かと思います。

まとめ

飲食店でのオペレーションを例に同期処理、非同期処理についてそれぞれ説明をしました。

JavaScriptでプログラミングする時には、非同期処理になることが多いので、上記の例だとドトールのような流れになるため、処理が一方通行ではないため全体で見ると処理内容が把握しづらくなる可能性が高くなります。

ただし書き方を工夫することで、JavaScriptでのプログラミングを同期処理のような感覚で書くことは可能なので、次でサンプルコードを例にしながら説明していきます

このリポジトリについて

サンプルコードを通じて非同期処理について理解する

先ほど飲食店でのオペレーションを例に非同期処理の概念を説明したのでその例えをふまえて

- 注文する
- 注文内容を確認する
- お会計の金額を計算する
- 調理を行う
- 調理が完了したのでお客様を呼ぶ

という一連の流れについて、JavaScriptのサンプルコードを示しながら非同期処理について説明をしていきます。

作業用の設定を行う

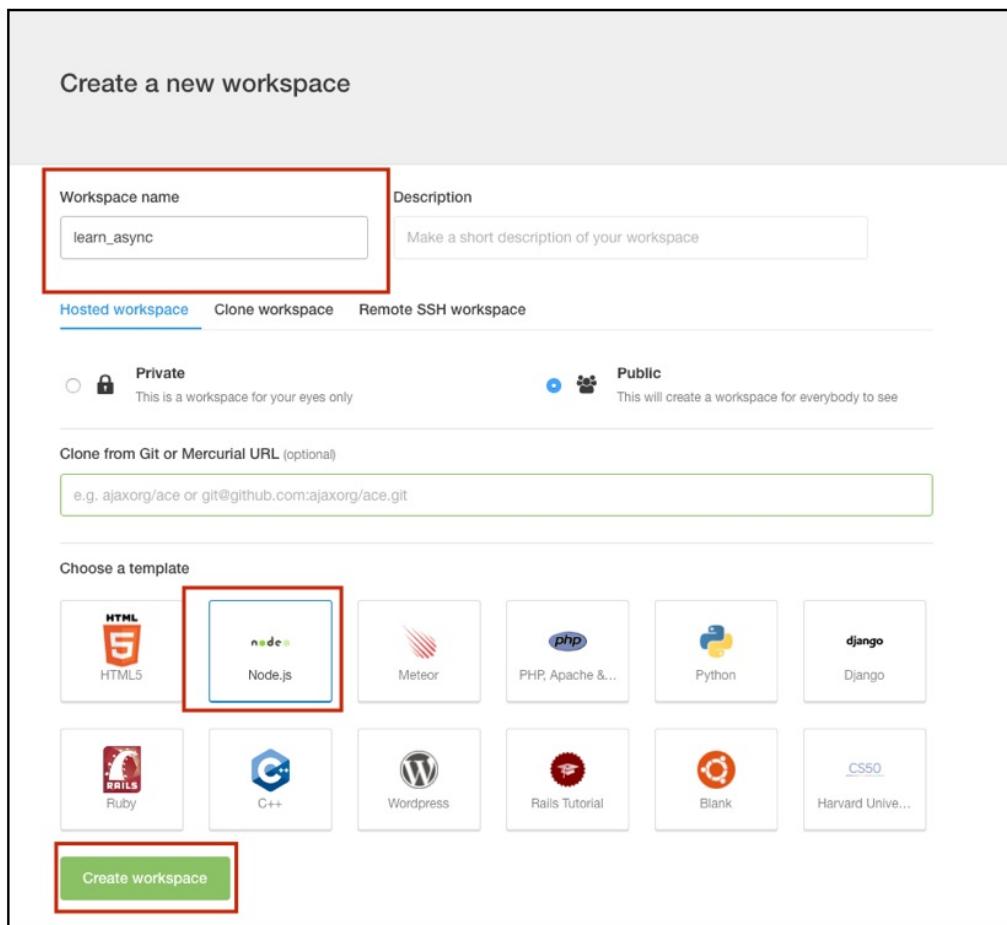
サンプルコードを実行する時に

- ローカルのWebサーバー機能
- jQuery

を念頭に置いて資料を作っていますので各自の環境に応じて以下の設定を行ってください

Cloud9を利用して場合

- ワークスペースの名前はひとまず**learn_async**という名称で作成してください。



ワークスペースの設定が完了すると初期設定段階でjQueryとローカルのWebサーバー機能がそのまま利用できる状態になります。

詳しくは[Cloud9というクラウド上の環境を利用する](#)のページの**コマンドを入力してサーバー機能を立ち上げる**の項目を参照してください

HTMLを修正する

初期段階のHTMLは不要な記述が多いので、clientフォルダの中のindex.htmlを開いて以下の内容に書き換えてください。

```
<html>
  <head></head>
  <body>

    <script src="/js/jquery.min.js"></script>
    <script type="text/javascript" src='/js/main.js'></script>
  </body>
</html>
```

Mac/Windows上でNode.jsの環境を利用する場合

- ローカルのWebサーバー機能
- jQuery

を利用するためには必要なnpmモジュールをインストールします。

```
{
  "name": "learn_async",
  "version": "1.0.0",
  "description": "",
  "author": "",
  "license": "MIT",
  "dependencies": {
    "gulp": "^3.9.0",
    "gulp-webserver": "*",
    "jquery": "^2.1.3"
  }
}
```

```
npm install
```

HTMLを作成する

index.htmlを作成して以下内容を記述します

```
<html>
  <head></head>
  <body>

    <script type="text/javascript" src='/node_modules/jquery/dist/jquery.min.js'></script>
    <script type="text/javascript" src='/js/main.js'></script>
  </body>
</html>
```

Webサーバー機能を利用するための設定

gulp-webserverを利用してWebサーバーを立ち上げるために**gulpfile.js**を作成して以下内容を記述します

```
var gulp = require('gulp');
var webserver = require('gulp-webserver');

gulp.task('webserver', function() {
  gulp.src('.')
    .pipe(webserver({
      livereload: false,
      port: 9000,
      fallback: 'index.html',
      open: true
    }));
});

gulp.task('default', ['webserver']);
```

gulpfile.jsを作成したら、以下コマンドを実行します

```
./node_modules/gulp/bin/gulp.js
```

Webブラウザを開いて以下にアクセス出来るのを確認してください。

このリポジトリについて

<http://localhost:9000/index.html>

非同期処理を考慮しない書き方

HTMLに以下を追加します

index.html

```
<body>
  <button id="storeOperation">処理の順番が正しくない</button>
  <div class="customerRequest"></div>

  <div class="operation"></div>
  <!-- 以下省略 -->
```

上記で読み込まれるJavaScriptは、js/main.jsというファイルを作つてそこに以下を記述します。

js/main.jsの中身

```
var customerRequestTotal,
customerOrderRequest = {
  food: { jp: 'ミラノサンドA', en: 'milanoA' },
  drink: { jp: 'ブレンドコーヒー', en: 'brendcoffee' }
},
messages = {
  confirm: '<ul><h3>ご注文内容を確認させていただきます</h3>' +
    '<li>' + customerOrderRequest.food.jp + '</li>' +
    '<li>' + customerOrderRequest.drink.jp + '</li>' +
    '</ul><hr />',
  callCustomer: 'ご注文のお客様、おまちどうさまでした！<hr />',
  cookingDone: '-----調理完了！-----<hr />'
};
var calculate = function(){
  var priceList = {
```

このリポジトリについて

```
food: { milanoA: 400 },
drink: { brendCoffee: 200 }
};

customerRequestTotal = priceList.food[customerOrderRequest.food];
priceList.drink[customerOrderRequest.drink.en];
};

var sayTotalPrice = function(){
  $('.operation').append('お会計は' + customerRequestTotal + '円で');
  $('.operation').append('<hr />');
};

var confirmOrderRequest = function(){
  $('.customerRequest').append(messages.confirm);
};

var cookingStaffCallCustomer = function(){
  $('.operation').append(messages.callCustomer);
};

var cookingStaffOperation = function(){ // (1)
  setTimeout(function(){
    $('.operation').append(messages.cookingDone);
  }, 3000);
};

var storeOperation = function(){
  calculate(customerOrderRequest);
  confirmOrderRequest();
  sayTotalPrice();
  cookingStaffOperation(); // (2)
  cookingStaffCallCustomer(); // (3)
};

$('#storeOperation').on('click', storeOperation); // (4)
```

上記のJavaScriptを保存して、HTMLを開くとボタンが1つ表示されるのでそれをクリックすると実行すると以下のようになります。

処理の順番が正しくない

コールバックで非同期処理を適切に処理

ご注文内容を確認させていただきます

- ミラノサンドA
- ブレンドコーヒー

お会計は600円です

ご注文のお客様、おまちどうさまでした！

-----調理完了！-----

先にご注文のお客様、おまちどうさまでした！のメッセージが表示された後に、-----調理完了！-----のメッセージが表示されてしまってます。

処理としては

- ソースコード内の(4)の箇所で、ボタンクリックでstoreOperationが実行される。
- storeOperation内の各メソッドが順番に実行され、cookingStaffOperationとcookingStaffCallCustomerのメソッドが実行される
- JavaScriptは非同期処理なのでcookingStaffOperationの結果を待たずにcookingStaffCallCustomerも実施される
 - cookingStaffOperation内でsetTimeoutを利用して3000ミリ秒（3秒）後に処理が行われる
 - 3秒後に調理完了のメッセージが挿入される
 - cookingStaffCallCustomer内の処理はすぐに完了してしまうので、ご注文のお客様、おまちどうさまでした！のメッセージが先に挿入されてしまう

このリポジトリについて

という流れになるため、本来意図した流れにはならなくなっています。

次で非同期処理を考慮した書き方に書き換えて本来期待される状態に修正します。

非同期処理を考慮する形に書き換える

さきほど非同期処理を考慮しない書き方の時に作ったindex.html、main.jsをそのまま流用しながら一部処理を追加します

HTMLの修正

```
<button id="storeAsyncOperation">コールバックで非同期処理を適切に処理<
```

という記述を追加します。最終的には以下のようになります。

```
<html>
<head></head>
<body>
  <button id="storeOperation">処理の順番が正しくない</button>
  <!-- 以下を追加する -->
  <button id="storeAsyncOperation">コールバックで非同期処理を適切に処理</button>
  <div class="customerRequest"></div>

  <div class="operation"></div>
  <script type="text/javascript" src='/node_modules/jquery/dist/jquery.min.js'></script>
  <script type="text/javascript" src='/js/main.js'></script>
</body>
</html>
```

JavaScriptの修正

続けて、JavaScriptのファイルも以下内容を追加します

```
var cookingStaffAsyncCallCustomer = function(){
    $('.operation').append(messages.callCustomer);
};

var cookingStaffAsyncOperation = function(callback){
    setTimeout(function(_callback){
        $('.operation').append(messages.cookingDone);
        // 調理完了の後にお客さんを呼び出すために以下コールバック関数を呼ぶ
        _callback();
    }, 3000, callback);
};

var storeAsyncOperation = function(){
    calculate(customerOrderRequest);
    confirmOrderRequest();
    sayTotalPrice();
    cookingStaffAsyncOperation(function(){
        cookingStaffAsyncCallCustomer();
    });
};

$('#storeAsyncOperation').on('click', storeAsyncOperation);
```

最終的には以下のようになります。

```
var customerRequestTotal,
customerOrderRequest = {
    food: { jp: 'ミラノサンドA', en: 'milanoA' },
    drink: { jp: 'ブレンドコーヒー', en: 'brendCoffee' }
},
messages = {
    confirm: '<ul><h3>ご注文内容を確認させていただきます</h3>' +
        '<li>' + customerOrderRequest.food.jp + '</li>' +
        '<li>' + customerOrderRequest.drink.jp + '</li>' +
        '</ul><hr />',
    callCustomer: 'ご注文のお客様、おまちどうさまでした！<hr />',
    cookingDone: '-----調理完了！-----<hr />'
};

var calculate = function(){
    var priceList = {
```

このリポジトリについて

```
food: { milanoA: 400 },
drink: { brendCoffee: 200 }
};

customerRequestTotal = priceList.food[customerOrderRequest.food];
priceList.drink[customerOrderRequest.drink.en];
};

var sayTotalPrice = function(){
  $('.operation').append('お会計は' + customerRequestTotal + '円で');
  $('.operation').append('<hr />');
};

var confirmOrderRequest = function(){
  $('.customerRequest').append(messages.confirm);
};

var cookingStaffCallCustomer = function(){
  $('.operation').append(messages.callCustomer);
};

var cookingStaffOperation = function(){
  setTimeout(function(){
    $('.operation').append(messages.cookingDone);
  }, 3000);
};

// ここから追加した内容
var cookingStaffAsyncCallCustomer = function(){
  $('.operation').append(messages.callCustomer);
};

var cookingStaffAsyncOperation = function(callback){
  setTimeout(function(_callback){
    $('.operation').append(messages.cookingDone);
    _callback();
  }, 3000, callback);
};

// ここまで追加した内容

var storeOperation = function(){
  calculate(customerOrderRequest);
  confirmOrderRequest();
};
```

```
sayTotalPrice();
cookingStaffOperation();
cookingStaffCallCustomer();
};

// ここから追加した内容
var storeAsyncOperation = function(){
    calculate(customerOrderRequest);
    confirmOrderRequest();
    sayTotalPrice();
    cookingStaffAsyncOperation(function(){
        cookingStaffAsyncCallCustomer();
    });
};

// ここまで追加した内容

$('#storeOperation').on('click', storeOperation);

// ここから追加した内容
$('#storeAsyncOperation').on('click', storeAsyncOperation);
// ここまで追加した内容
```

上記修正して、保存をしてから画面を開くと**コールバックで非同期処理を適切に処理**というボタンが新たに追加されてるかと思いますのでそちらをクリックして実行すると以下のように調理完了のメッセージの後に、お客様を呼ぶという本来意図した動作になってるかと思います。

処理の順番が正しくない

コールバックで非同期処理を適切に処理

ご注文内容を確認させていただきます

- ミラノサンドA
- ブレンドコーヒー

お会計は600円です

-----調理完了！-----

ご注文のお客様、おまちどうさまでした！

setTimeoutの前後の処理の解説

setTimeoutを使って一定時間経過後に処理を行うようにしていますが、慣れないいうちは頭のなかでここの処理イメージが湧きづらいかと思うので念のため図解しておきます

時間軸とJavaScriptの処理内容の対比



JavaScriptのコード

```
var cookingStaffAsyncOperation = function(callback){  
    setTimeout(function(_callback){  
        $('.operation').append(messages.cookingDone);  
        _callback();  
    }, 3000, callback);  
};  
  
cookingStaffAsyncOperation(function(){  
    cookingStaffAsyncCallCustomer();  
});
```

cookingStaffAsyncOperation
メソッドの引数では引数名を
callbackとしてます

setTimeoutの引数では上記と
区別しやすくするため引数名を
先頭にアンダースコアを付けた
_callbackとしてます

もしも上記のコードのコールバックでもイメージが湧きづらい場合には、一番シンプルな処理を以下記載しておきますので、Webブラウザのconsole画面などで実際に動作確認してみてください

```
var show = function(callback){  
    console.log('show start');  
    callback();  
};  
var message = function(str){  
    console.log(str);  
};  
show(function(){  
    message('showの中身が表示されてからこの値が表示される');  
});  
// show start  
// showの中身が表示されてからこの値が表示される
```


非同期処理のコードにありがちな問題解決

先ほどコールバックで非同期処理を適切に処理するように以下の様なコードを書きました。

```
var storeAsyncOperation = function(){
    calculate(customerOrderRequest);
    confirmOrderRequest();
    sayTotalPrice();
    cookingStaffAsyncOperation(function(){
        cookingStaffAsyncCallCustomer();
    });
};
```

コールバックで次の処理を呼ぶのが少ない現状は良いかもしれません、実際は

```
cookingStaffAsyncOperation(function(){
    cookingStaffAsyncCallCustomer(function(){
        anotherOperation(function(response){
            if(response){
                // 何かの処理
            } else {
                // 何かの処理
            }
        });
    });
});
```

というように処理が入れ子状（ネストの深い）コードになってしまることが多いかと思います。 こういうコードにならないようにするためにアプローチを、次で説明していきます。

このリポジトリについて

jQueryのDeferred基本編

jQueryのDeferredという機能を活用することで処理が入れ子状にならないようになります。

jQueryのDeferredについてはYahooデベロッパーネットワークの記事にわかりやすくまとめが書かれてるのでそこから引用します。

非同期処理を連結する際、コールバック地獄から解放される（直列処理、並列処理が可能） エラー処理をうまく記述できる一連の非同期処理を関数化して再利用しやすくできる

[爆速でわかるjQuery.Deferred超入門](#)より引用

実装した非同期処理をjQueryのDeferredを利用した形に書き換えることで、

```
cookingStaffAsyncOperation()  
  .done(cookingStaffAsyncCallCustomer)  
  .done(anotherOperation)
```

というような処理が入れ子状にならないような書き方をすることができます。

DeferredとPromiseの用語の整理

[爆速でわかるjQuery.Deferred超入門](#)の概念図、ならびに、説明がとてもわかりやすいのでそちらを引用しながら用語の整理をしておきます

DeferredとPromise

- 2つともjQuery.Deferredが生成するオブジェクト
- DeferredはPromiseを内包
 - Deferredオブジェクトを生成時に自動的にPromiseオブジェクトも

生成される

- DeferredとPromiseは常に1対1で作成される
- Promiseオブジェクトは概念的には以下3つを持っている
 - 状態
 - 状態がresolvedになった時に実行されるコールバック(.done)
 - 状態がrejectedになった時に実行されるコールバック(.fail)



- `$.Deferred()`でDeferredオブジェクトを生成
- オブジェクト生成時に自動的にPromiseオブジェクトも生成

Deferredオブジェクトと非同期処理

まず概念図をYahooデベロッパネットワークの記事から引用します。



上記の概念図で非同期に処理されるメソッドをDeferred対応させると書いてますが、対応前のコードと対応後のコードを比較するとイメージが湧くかと思うので以下記載します

対応前のコード

```
var operation = function(callback){ // (1)
    setTimeout(function(_callback){
        $('.operation').append(messages.cookingDone);
        _callback(); // (2)
    }, 3000, callback);
};
```

Deferred対応させたコード

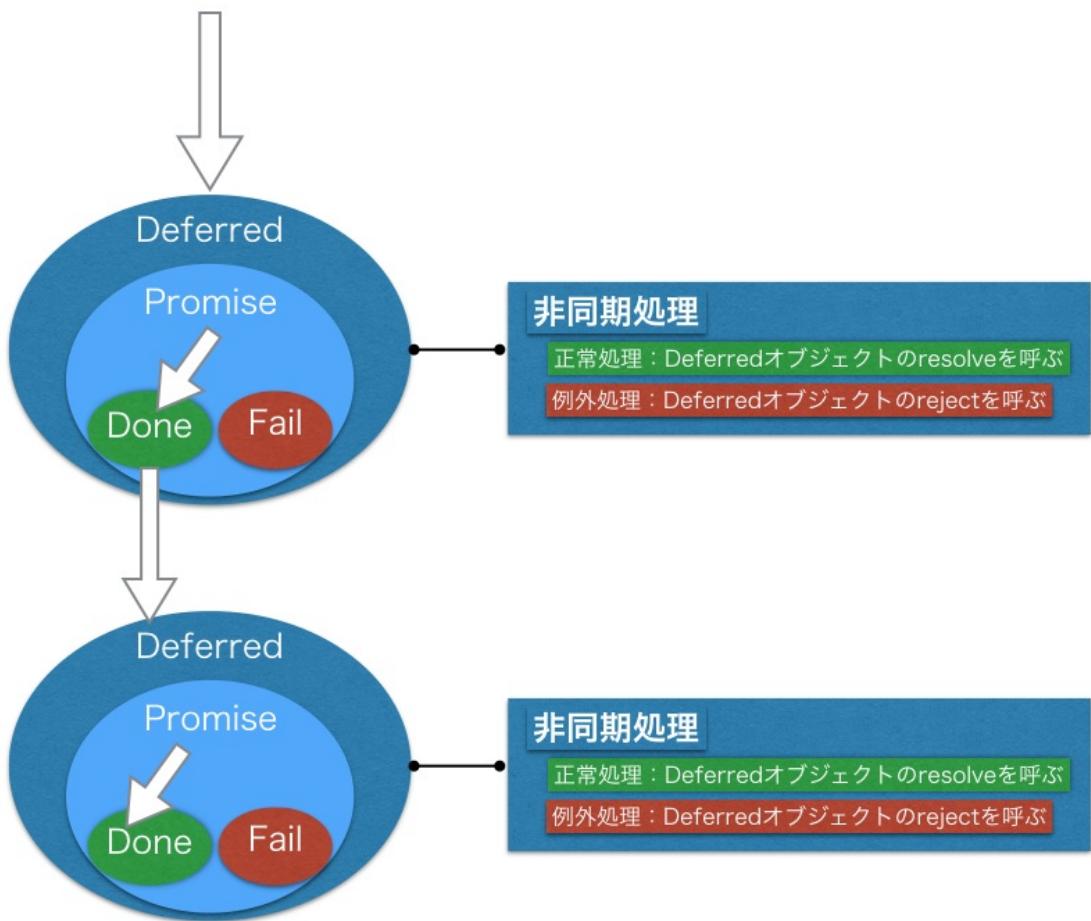
```
var operation = function(){ // (3)
    var deferred = new $.Deferred; // (4)
    setTimeout(function(){
        $('.operation').append(messages.cookingDone);
        deferred.resolve(); // (5)
    }, 3000);
    return deferred.promise(); // (6)
};
```

対応前では引数にcallbackという名前で関数を取り処理が終了したら
_callback()でコールバック関数を呼び出してました

対応後は、引数にコールバック関数を取らなくなり、その代わりに、メソッド内でjQuery.Deferredオブジェクトを生成しつつ必要な処理を行っています。

Deferred対応させたコードの流れ

Deferred対応させたメソッドが2つあった場合に以下のように処理が実行されます。



非同期処理とDeferredのオブジェクトが紐付いてることで、そこのオブジェクトの状態に応じて次の処理が呼ばれるという流れになります。

ここまで説明を踏まえてjQueryのDeferredを利用した形に書き換える

main.jsに以下を追加します

```
var cookingStaffDeferredOperation = function(){
    var deferred = new $.Deferred;
    setTimeout(function(){
        $('.operation').append(messages.cookingDone);
        deferred.resolve();
    }, 3000);
    return deferred.promise();
};

var dringStaffDeferredOperation = function(){
    var deferred = new $.Deferred;
    setTimeout(function(){
        $('.operation').append(messages.dringDone);
        deferred.resolve();
    }, 1000);
    return deferred.promise();
};

var storeDeferredOperation = function(){
    calculate(customerOrderRequest);
    confirmOrderRequest();
    sayTotalPrice();
    cookingStaffDeferredOperation()
        .done(dringStaffDeferredOperation)
        .done(cookingStaffAsyncCallCustomer);
};

$('#storeDeferredOperation').on('click', storeDeferredOperation);
```

index.htmlに

```
<button id="storeDeferredOperation">Deferredを利用する</button>
```

という内容を追加して画面上で動作確認をしてみてください。

まとめ

このリポジトリについて

jQueryのDeferredの機能を活用することで処理が入れ子状にならないようにすることが出来ることを紹介しました。

今回のようなサンプルだとjQueryのDeferredの機能の有り難みがそれほど感じられないかもしれません、もう少し実践的なサンプルを次で説明します。

jQueryのDeferred応用編

jQueryのDeferredで処理が入れ子状にならないようにすることが出来るこ
とを先ほど説明しましたが、今度は

- A、B、Cという処理がある
- **AとBが完了したらCを実行する**というようにある処理を連結してそれ
が完了したら最後の処理を行う

というような場合にjQueryのDeferredの機能を利用することで見通しの良
いコードで書けるようになるので最後にその例についても取り上げます

最初に飲食店でのオペレーションを例に非同期処理の概念を説明してるので、その例えをふまえて

- 以下2つの作業を同時に行う
 - 食べ物の調理を行う
 - 飲み物の準備を行う
- 上記2つが完了したらお客様を呼び出す

という状況を表現するサンプルコードを考えてみたいと思います。

ある処理を連結させてそれが完了したら最後の 処理を行うためにjQuery.whenを利用する

ある処理を連結させるという部分でjQueryのwhenを以下のようにして利用
します

```
$.when(処理A, 処理B)
  .done(function(){
    処理Aと処理Bが完了したら呼ばれる処理C
  });
}
```

実際のコード

先程実装したコードがそれぞれ以下の様なイメージで実装します。

- 食べ物の調理を行う
 - cookingStaffDeferredOperation
- 飲み物の準備を行う
 - dringStaffDeferredOperation

cookingStaffDeferredOperation、dringStaffDeferredOperationはそれぞれjQueryのDeferredの機能を利用して実装しています。

そのためこちらのコードは修正の必要がありません。

この2つのメソッドを呼び出す時にjQueryのwhenを利用して以下のようにします。

```
var mergeOperations = function(){
    $.when(cookingStaffDeferredOperation(), dringStaffDeferredOper
        .done(function(){
            cookingStaffAsyncCallCustomer();
        });
};
```

index.htmlに

```
<button id="mergeOperations">whenを利用する</button>
```

という内容を追加して画面上で動作確認をしてみてください。

上記修正後に実行すると

- 飲み物の準備完了と調理完了のメッセージが表示される
- 上記2つが完了したらお客様を呼び出すためのメッセージが表示される

という形になったかと思います。

ここまで処理のサンプルコード全体

index.html

```
<html>
<head>
</head>
<body>
  <button id="storeOperation">処理の順番が正しくない</button>
  <button id="storeAsyncOperation">コールバックで非同期処理を適切に処理</button>
  <button id="storeDeferredOperation">Deferredを利用する</button>
  <button id="mergeOperations">whenを利用する</button>
  <div class="customerRequest"></div>

  <div class="operation"></div>
  <script type="text/javascript" src='/node_modules/jquery/dist/jquery.js'></script>
  <script type="text/javascript" src='src/main.js'></script>
</body>
</html>
```

main.js

```
var customerRequestTotal,
    customerOrderRequest = {
      food: { jp: 'ミラノサンドA', en: 'milanoA' },
      drink: { jp: 'ブレンドコーヒー', en: 'brendCoffee' }
    },
    messages = {
      confirm: '<ul><h3>ご注文内容を確認させていただきます</h3>' +
        '<li>' + customerOrderRequest.food.jp + '</li>' +
        '<li>' + customerOrderRequest.drink.jp + '</li>' +
        '</ul><hr />',
```

このリポジトリについて

```
callCustomer: 'ご注文のお客様、おまちどうさまでした！<hr />',
cookingDone: '-----調理完了！-----<hr />',
dringDone: '-----飲み物の準備完了！-----<hr />'
};

var calculate = function(){
  var priceList = {
    food: { milanoA: 400 },
    drink: { brendCoffee: 200 }
  };
  customerRequestTotal = priceList.food[customerOrderRequest.food];
  priceList.drink[customerOrderRequest.drink.en];
};

var sayTotalPrice = function(){
  $('.operation').append('お会計は' + customerRequestTotal + '円で');
  $('.operation').append('<hr />');
};

var confirmOrderRequest = function(){
  $('.customerRequest').append(messages.confirm);
};

var cookingStaffCallCustomer = function(){
  $('.operation').append(messages.callCustomer);
};

var cookingStaffOperation = function(){
  setTimeout(function(){
    $('.operation').append(messages.cookingDone);
  }, 3000);
};

var cookingStaffAsyncCallCustomer = function(){
  $('.operation').append(messages.callCustomer);
};

var cookingStaffAsyncOperation = function(callback){
  setTimeout(function(_callback){
    $('.operation').append(messages.cookingDone);
    // 調理完了の後にお客さんを呼び出すために以下コールバック関数を呼ぶ
    _callback();
  }, 3000, callback);
};

var storeOperation = function(){
```

```
calculate(customerOrderRequest);
confirmOrderRequest();
sayTotalPrice();
cookingStaffOperation();
cookingStaffCallCustomer();
};

var storeAsyncOperation = function(){
    calculate(customerOrderRequest);
    confirmOrderRequest();
    sayTotalPrice();
    cookingStaffAsyncOperation(function(){
        cookingStaffAsyncCallCustomer();
    });
};

var cookingStaffDeferredOperation = function(){
    var deferred = new $.Deferred;
    setTimeout(function(){
        $('.operation').append(messages.cookingDone);
        deferred.resolve();
    }, 3000);
    return deferred.promise();
};

var dringStaffDeferredOperation = function(){
    var deferred = new $.Deferred;
    setTimeout(function(){
        $('.operation').append(messages.dringDone);
        deferred.resolve();
    }, 1000);
    return deferred.promise();
};

var storeDeferredOperation = function(){
    calculate(customerOrderRequest);
    confirmOrderRequest();
    sayTotalPrice();
    cookingStaffDeferredOperation()
        .done(dringStaffDeferredOperation)
        .done(cookingStaffAsyncCallCustomer());
};
```

このリポジトリについて

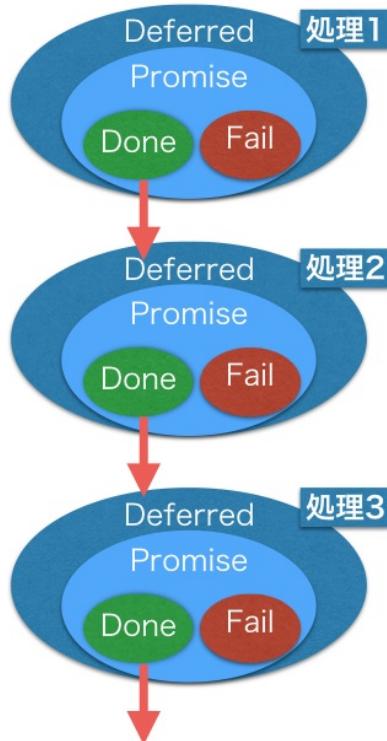
```
var mergeOperations = function(){
  $.when(cookingStaffDeferredOperation(), dringStaffDeferredOper
    .done(function(){
      cookingStaffAsyncCallCustomer();
    });
};

$('#storeOperation').on('click', storeOperation);
$('#storeAsyncOperation').on('click', storeAsyncOperation);
$('#storeDeferredOperation').on('click', storeDeferredOperation);
$('#mergeOperations').on('click', mergeOperations);
```

jQueryのDeferred処理をおさらい

jQueryのDeferredを利用してある処理を意図した順番に行う場合にはこういうイメージになるかと思います。

ある処理を意図した順番に行う



JavaScriptのコードだと以下の様な形になるかと思います。

```
operation1()
  .done(operation2())
  .done(operation3())
  .done(finalOperation());
```

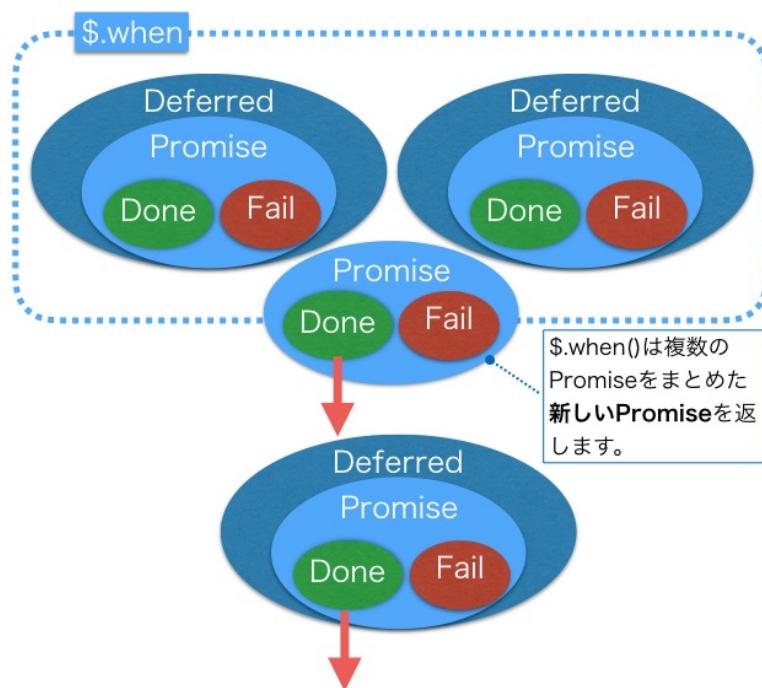
もしも処理の順序を処理 2 → 処理 3 → 処理 1 に変更したいという仕様変更があった場合でもjQueryのDeferred化の対応をしたメソッドとして定義しておくことで

```
operation2()  
    .done(operation3())  
    .done(operation1())  
    .done(finalOperation());
```

という形ですぐに対応できます。

ある処理を結合（並列）してそれが終わったら次を呼ぶ処理というのもjQueryのDeferred化の対応をしたメソッドとして定義しつつ\$.whenを利用することで対応できます。

ある処理を結合（並列）してそれが終わったら次を呼ぶ



JavaScriptのコードだと以下の様な形になるかと思います。

```
$.when(operation1(), operation2())  
    .done(operation3())  
    .done(finalOperation());
```

当然のことながら、ある処理を結合（並列）をやめて、直列的に処理したいという場合には\$.whenの処理を変更するだけですみます。

全体のまとめ

非同期処理は、処理が一方通行にならないため全体としてみた場合に把握しづらいという特徴があります。

上記点をふまえてコールバック関数を利用することで、処理が意図した順番で呼ばれるようなコードをまずは書いてもらいましたが、コールバック関数で処理をしていくと以下の様なネストが深い（よく言われるコールバック地獄）メンテナンスしづらいコードになってしまいます。

```
cookingStaffAsyncOperation(function(){
    cookingStaffAsyncCallCustomer(function(){
        anotherOperation(function(response){
            if(response){
                // 何かの処理
            } else {
                // 何かの処理
            }
        });
    });
});
```

こういった問題を解決する方法の1つとしてjQueryのDeferredを利用することで対応することが出来ることを紹介しました。

jQueryのDeferredは[結局jQuery.Deferredの何が嬉しいのか分からない、という人向けの小話](#)の記事を参考に情報まとめておりますので、こちらの記事もチェックすると理解が深まるかと思います。

thisについて

JavaScriptでのプログラミングをする上でthisの理解というのは欠かせないものの1つだと思います。

thisの理解をする上で以下の概念の理解がポイントになるかと思っています。

- アクティベーションオブジェクトとスコープチェーン
 - 関数呼び出し時に作成される目に見えない変数オブジェクトがあり、それはアクティベーションオブジェクト（もしくはCallオブジェクト）と呼ばれる
- 関数呼び出し時にthisが生成される
 - 呼び出し時にthisが自動的に生成される
- thisの呼ばれ方は4パターン
 - トップレベルのthis
 - コンストラクタ内のthis
 - 何かに所属している時のthis
 - thisを外部から書き換えられる場合
 - function#apply
 - function#call

サンプルコードを示しつつ、実際の挙動についてGoogle ChromeのDevelopetToolsを使って確認してもらうことを通じて理解の手助けとなるような解説をしていこうと思います。

最初にプロジェクトの設定を行う

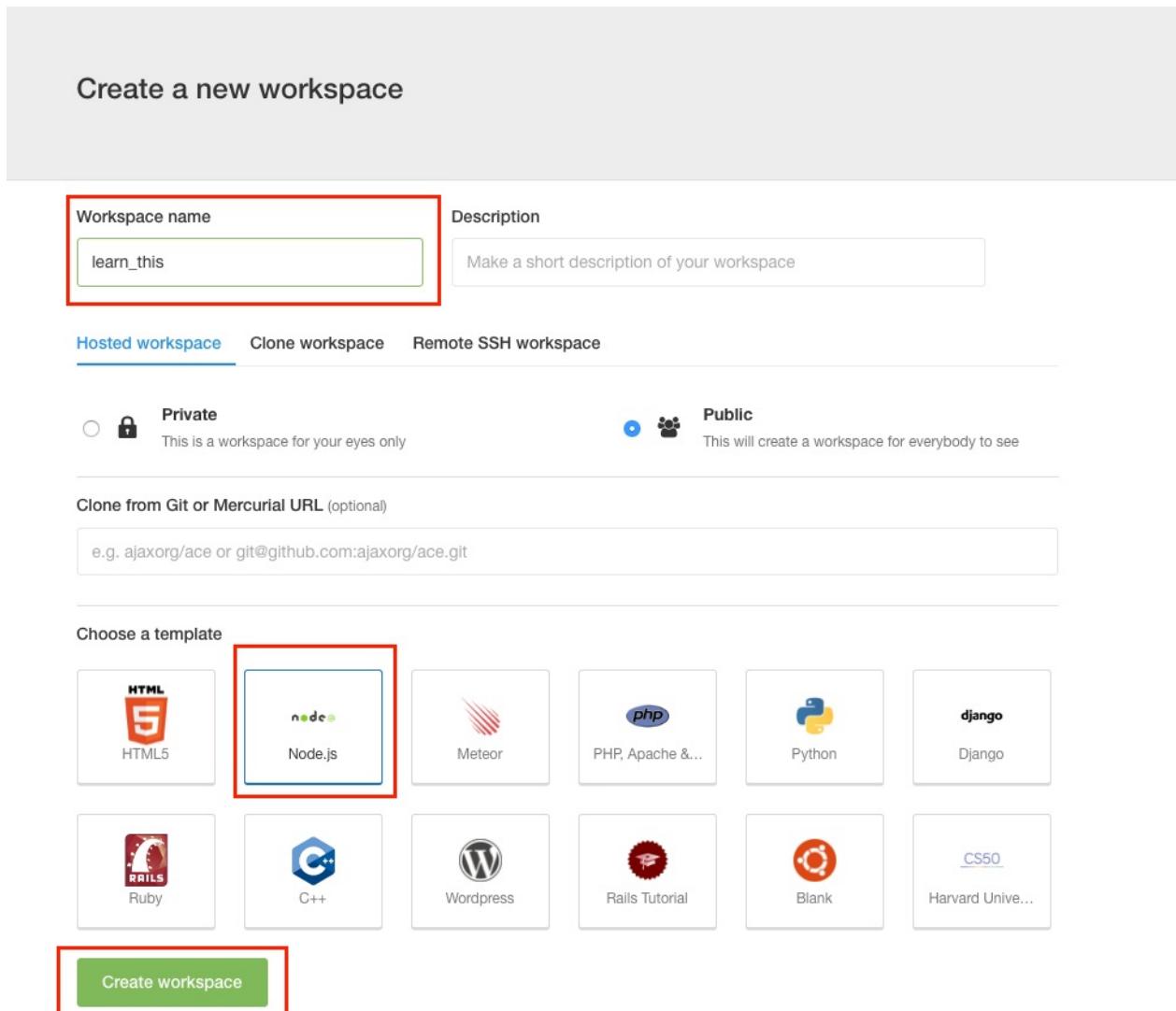
サンプルコードを実行する時に

- ローカルのWebサーバー機能
- jQuery

を念頭に置いて資料を作っていますので各自の環境に応じて以下の設定を行ってください

Cloud9を利用してする場合

- ワークスペースの名前はひとまずlearn_thisという名称で作成してください。



ワークスペースの設定が完了すると初期設定段階でjQueryとローカルのWebサーバー機能がそのまま利用できる状態になってます。

詳しくはCloud9というクラウド上の環境を利用するのページのコマンドを入力してサーバー機能を立ち上げるの項目を参照してください

HTMLを修正する

初期段階のHTMLは不要な記述が多いので、clientフォルダの中のindex.htmlを開いて以下の内容に書き換えてください。

```
<html>
  <head></head>
  <body>

    <script src="/js/jquery.min.js"></script>
    <script type="text/javascript" src='/js/main.js'></script>
  </body>
</html>
```

Mac/Windows上でNode.jsの環境を利用する場合

- ローカルのWebサーバー機能
- jQuery

を利用するためには必要なnpmモジュールをインストールします。

```
{
  "name": "learn_async",
  "version": "1.0.0",
  "description": "",
  "author": "",
  "license": "MIT",
  "dependencies": {
    "gulp": "^3.9.0",
    "gulp-webserver": "*",
    "jquery": "^2.1.3"
  }
}
```

```
npm install
```

HTMLを作成する

index.htmlを作成して以下内容を記述します

```
<html>
  <head></head>
  <body>

    <script type="text/javascript" src='/node_modules/jquery/dist/jquery.min.js'></script>
    <script type="text/javascript" src='/js/main.js'></script>
  </body>
</html>
```

Webサーバー機能を利用するための設定

gulp-webserverを利用してWebサーバーを立ち上げるために**gulpfile.js**を作成して以下内容を記述します

```
var gulp = require('gulp');
var webserver = require('gulp-webserver');

gulp.task('webserver', function() {
  gulp.src('.')
    .pipe(webserver({
      livereload: false,
      port: 9000,
      fallback: 'index.html',
      open: true
    }));
});

gulp.task('default', ['webserver']);
```

gulpfile.jsを作成したら、以下コマンドを実行します

```
./node_modules/gulp/bin/gulp.js
```

Webブラウザを開いて以下にアクセス出来るのを確認してください。

このリポジトリについて

<http://localhost:9000/index.html>

アクティベーションオブジェクトとスコープチェーンについて

thisの理解をする上でアクティベーションオブジェクトとスコープチェーンの概念を抑えておく必要があるかと思うのでご紹介します。

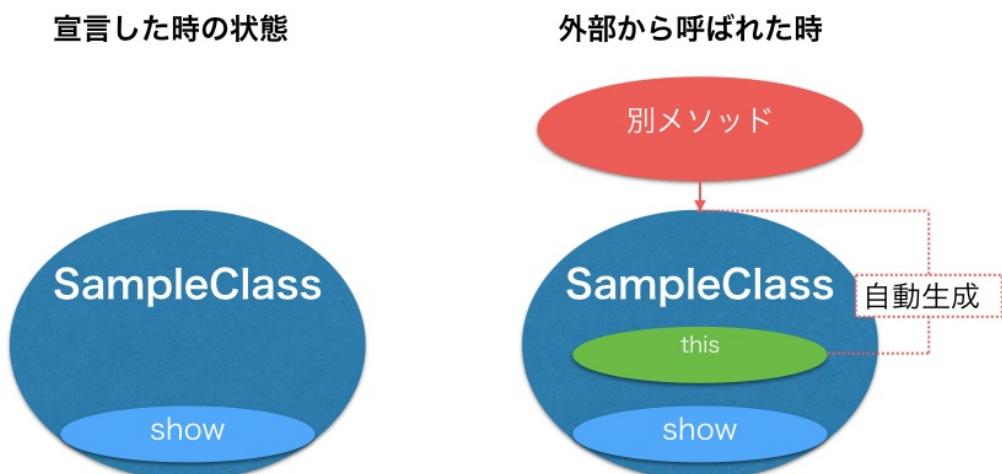
アクティベーションオブジェクトとは？

関数呼び出し時に作成される目に見えない変数オブジェクトというものがあり、その目に見えない変数オブジェクトのことをアクティベーションオブジェクト（もしくはCallオブジェクト）といいます。

この変数オブジェクトは関数の実行時に生成されます。関数宣言しただけではアクセスすることができません。

アクティベーションオブジェクト

関数がコールされた時に自動的に生成されるオブジェクト。thisはアクティベーションオブジェクトの1つ



上記の概念図で記載してますがthisはアクティベーションオブジェクトの一つです。

スコープチェーンについて

以下のコードを例にしながらスコープチェーンについて解説します。

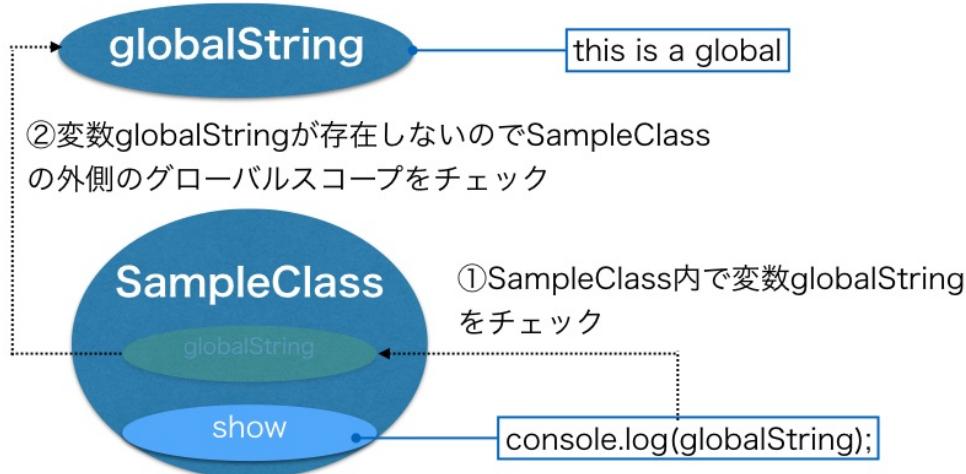
```
var globalString,  
    sample;  
globalString = 'this is a global';  
  
SampleClass = function(){  
    this.show = function(){  
        console.log(globalString); // (1)  
    };  
};  
sample = new SampleClass();  
sample.show();
```

- sample.show()実行時には実際にはコメント(1)の箇所が評価されます
- JavaScriptは関数ごとにスコープがあるのでまずはthis.show()の中で変数globalStringが定義されているかどうか探されます
- このコードの場合にはthis.show内部には変数globalStringは存在しないため、関数のスコープに変数が見つからなかった場合には外側のスコープを探しにいきます
 - このコードの場合には外側はグローバル変数にあたるのでそこで変数globalStringが存在するかどうか探されてglobalString が見つかります
 - this is a globalが代入されてる箇所

ここまで流れを図にすると以下のようになります。

JavaScriptのスコープチェーン

変数の参照を解決する際にアクティベーションオブジェクトを辿る仕組みです



スコープチェーンを解説した理由ですがグローバル変数の領域にも`this`は存在しており、Webブラウザの場合には`window`オブジェクトになります。

またJavaScriptのグローバル変数は`window`オブジェクトのプロパティになるので上記のサンプルコードだと

```
var globalString,  
    sample;  
globalString = 'this is a global';
```

の`globalString`の箇所は

```
window.globalString  
this.globalString
```

と同じ意味になります。

関数呼び出し時にthisが生成される

実際にサンプルコードを書きながら関数呼び出し時にthisが生成されることを確認してみます。

先ほど設定したプロジェクトのjs/main.jsに以下内容を記述します

```
var SampleClass,  
    sample;  
  
SampleClass = function(str){  
    var innerString = 'inner';  
    this.show = function(){  
        console.log(innerString);  
    };  
};  
  
sample = new SampleClass();  
sample.show();
```

記述が完了したら動作を確認します。

Cloud9の環境の方

コマンドを入力してWebサーバー機能を立ち上げて指定のアドレスにアクセスします。

詳しくは[Cloud9というクラウド上の環境を利用する](#)のページのコマンドを入力してサーバー機能を立ち上げるの項目以降を参考に作業してみてください。

ローカルのNode環境を利用してる方

gulp-webserverは9000番ポートを利用してサーバーが起動してますので

このリポジトリについて

<http://localhost:9000>

にアクセスします。

実際に確認してみる

GoogleChromeを使ってサイトにアクセスしてから、デベロッパーツールを表示させます。

デベロッパーツールは以下のようにChromeのメニューの表示→開発・管理→デベロッパーツールと進むことで表示されます。

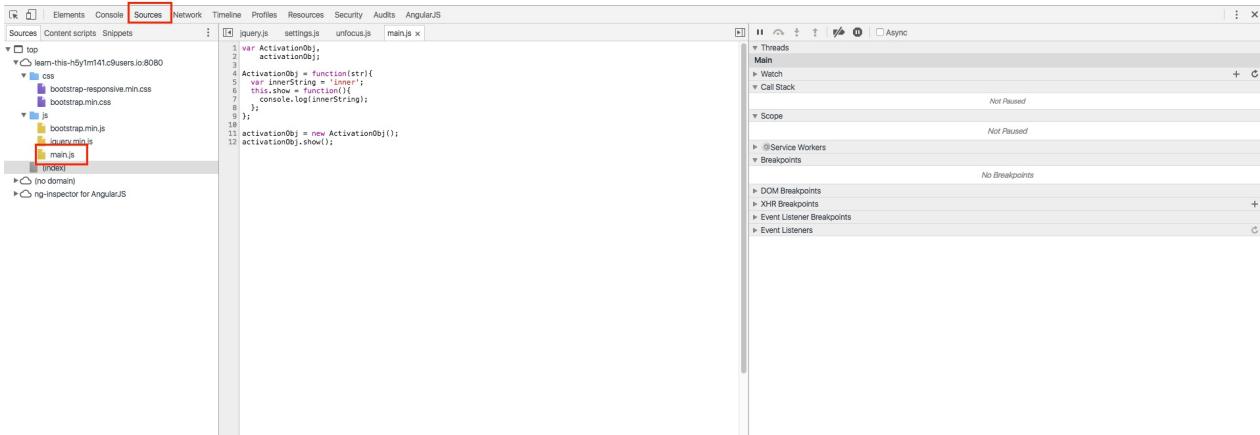


画面が上下に分かれて、下部にデベロッパーツールが表示されるので以下作業をします

1. Sourceタブを選択
2. 左側に読み込まれてるJavaScript、CSSの一覧が表示されます
3. 今回実装したmain.jsを選択すると、デベロッパーツールの真ん中の

このリポジトリについて

ウィンドウに先ほど実装したmain.jsの中身が表示されます。



main.jsが表示されたら処理を一時停止させるためにthis.showの横のあたりをマウスでクリックしてください。

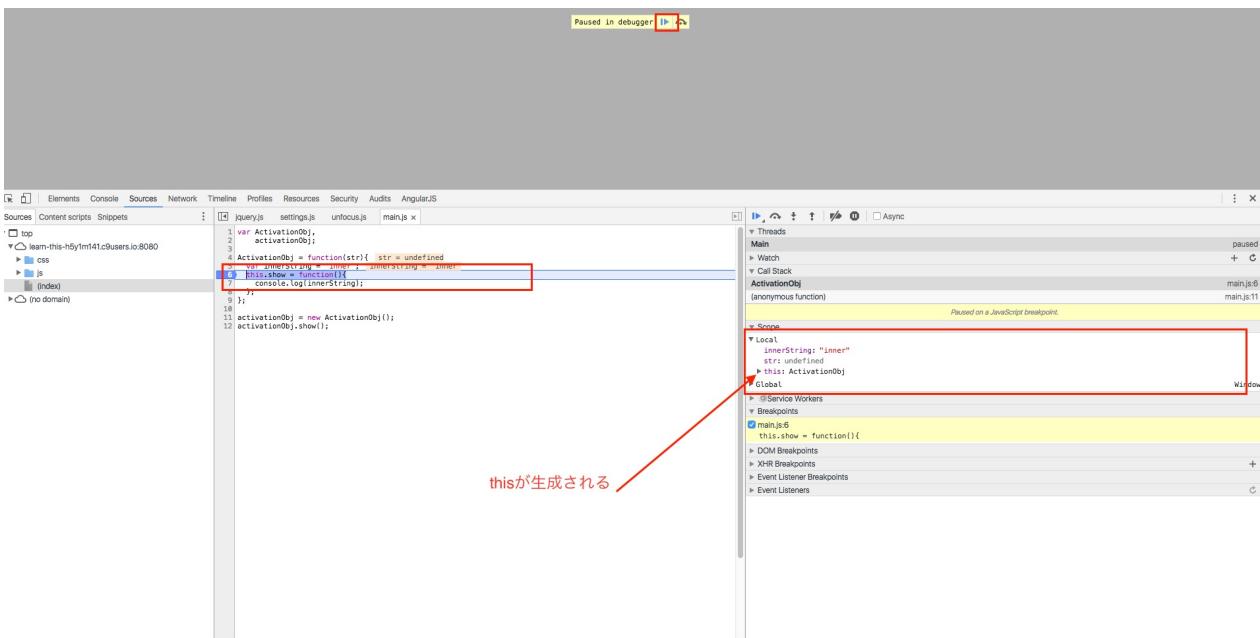
クリックした箇所の左側に青矢印がこのように表示された状態になったことを確認してください

A screenshot of the main.js code editor. The line 'this.show = function(){' is highlighted with a blue arrow pointing to the left, indicating it is a breakpoint. The code is as follows:

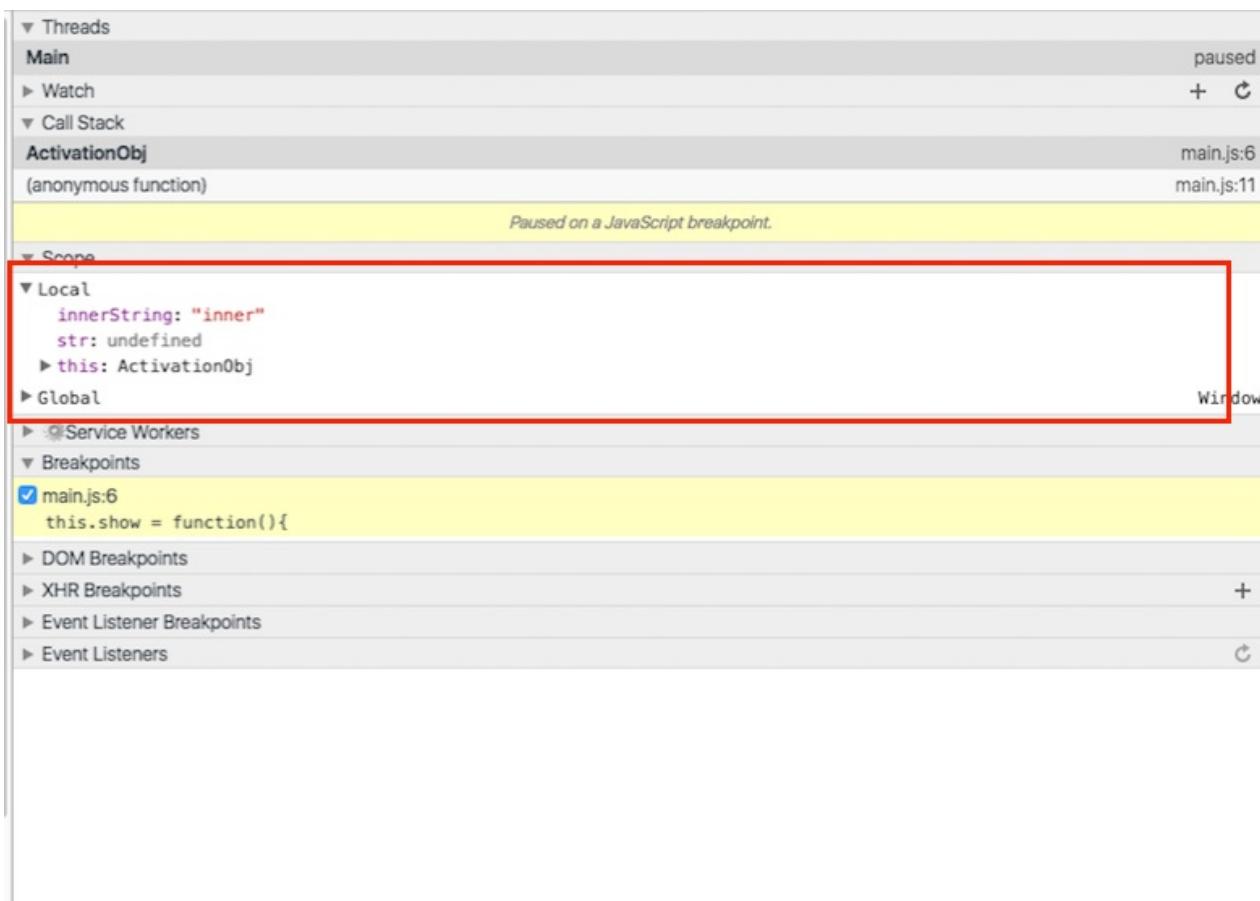
```
1 var ActivationObj,
2     activationObj;
3
4 ActivationObj = function(str){ str = undefined
5   var innerString = inner; innerString = inner;
6   this.show = function(){
7     console.log(innerString);
8   };
9 };
10
11 activationObj = new ActivationObj();
12 activationObj.show();
```

この状態でWebブラウザの再読み込みボタンをクリックすると先ほどクリックした箇所の記述が実行されるタイミングでこのように↓処理が一時停止します。

このリポジトリについて



上記画面の一部を拡大した画像を以下貼りますがこのようにthisが自動的に生成されてることが確認できるかと思います。



まとめ

関数呼び出し時にthisが生成される

このリポジトリについて

GoogleChromeのデベロッパーツールを利用して関数呼び出し時にthisが生成されることが確認できたかと思います。

thisは関数実行時に生成されますが、thisが呼ばれる方法としては全部で4パターンあるのでそのあたりのことを念頭に置いておくともう少し理解が深まると思うので次で順番に解説していきます。

thisの呼ばれ方その1：トップレベルのthis

アクティベーションオブジェクトとスコープチェーンについてのページで紹介しましたが、グローバル変数の領域にもthisは存在しており、Webブラウザの場合にはwindowオブジェクトになります。

実際に動作確認してみる

簡単なサンプルコードを書いて実際にトップレベルのthisについて確認してみます。

設定したプロジェクトのjs/main.jsに以下内容を追記します。

```
function showMessage(){
  console.log('メッセージを表示');
}
showMessage();
```

コードの修正が完了したら、GoogleChromeを使ってサイトにアクセスしてから、デベロッパーツールを表示させます。

1. console.log('メッセージを表示')の横のあたりをマウスでクリックしてブレークポイントを設定
2. この状態でWebブラウザの再読み
3. 設定したブレークポイントの箇所で以下のように処理が止まってその時にthisが生成されてることを確認してみてください。

このリポジトリについて

The screenshot shows the Google Chrome Developer Tools with the "Sources" tab selected. The left pane displays the file structure under "localhost:9000". The right pane shows the code editor with the file "main.js" open. A red box highlights the line of code: "15 console.log('メッセージを表示');". The code editor indicates this is line 15, column 3. The right pane also shows the "Call Stack" and "Local" variables, with "this: Window" highlighted by a red box.

```
Developer Tools - http://localhost:9000/activation_object/index.html
Sources Content... Snippets Sources Network Timeline Profiles Resources Security Audits AngularJS
jquery.js settings.js unfocus.js main.js
Serving from the file system? Add your files int... more never show
top
localhost:9000
activation_object
src
index.html
node_modules/jquery/dist
(no domain)

1 var SampleClass,
2     sample;
3
4 SampleClass = function(str){
5     var innerString = 'inner';
6     this.show = function(){
7         console.log(innerString);
8     };
9 };
10
11 sample = new SampleClass();
12 sample.show();
13
14 function showMessage(){
15     console.log('メッセージを表示');
16 }
showMessage();
17
18

Line 15, Column 3

Call Stack
showMessage
main.js:15
(anonymous function)
main.js:17
Paused on a JavaScript breakpoint.

Scope
Local
this: Window
Global
Breakpoints
main.js:15
console.log('メッセージを表示');

DOM Breakpoints
XHR Breakpoints
Event Listener Breakpoints
Event Listeners
```

thisの呼び方その2：コンストラクタ内のthis

こちらは関数呼び出し時にthisが生成されるのページで紹介しましたが、グローバル変数の領域にもthisは存在しており、Webブラウザの場合にはwindowオブジェクトになります。

実際に動作確認してみる

先ほどと同様に設定したプロジェクトのjs/main.jsに以下内容を追記します。

```
var constructorSample,  
    ConstructorSample;  
ConstructorSample = function(){  
    this.str = 'this is a sample';  
    this.show = function(){  
        console.log(this.str);  
    };  
};  
constructorSample = new ConstructorSample();  
constructorSample.show();
```

コードの修正が完了したら、GoogleChromeを使ってサイトにアクセスしてから、デベロッパーツールを表示させます。

1. constructorSample = new ConstructorSample()の横のあたりをマウスでクリックしてブレークポイントを設定
2. この状態でWebブラウザの再読み
3. 設定したブレークポイントの箇所で以下のように処理が止まります。コンストラクタ内のthisを確認するためには1つづつ処理を進めていく必要がありそのためには、以下画面キャプチャで囲ってるStep intoのボタン箇所をクリックしながら処理を進めていきます

このリポジトリについて

new する箇所にブレークポイントを設定する

Step intoのボタンをクリックしてnewした後の処理を1つづつ順番に進める

```
1 var SampleClass,
2   sample;
3
4 SampleClass = function(str){
5   var innerString = 'inner';
6   this.show = function(){
7     console.log(innerString);
8   };
9 }
10
11 sample = new SampleClass();
12 sample.show();
13
14 function showMessage(){
15   console.log('メッセージを表示');
16 };
17 showMessage();
18
19 var constructorSample,
20   ConstructorSample;
21 ConstructorSample = function(){
22   this.str = 'this is a sample';
23   this.show = function(){
24     console.log(this.str);
25   };
26 }
27 constructorSample = new ConstructorSample();
28 constructorSample.show();
```

4. 順番に処理を進めていくことで以下のようにコンストラクタ内のthisが生成されるタイミングが確認できるかと思います

このリポジトリについて

Developer Tools - http://localhost:9000/activation_object/index.html

Sources Content... Snippets Sources Network Timeline Profiles Resources Security Audits AngularJS

Sources Content... Snippets jquery.js main.js > Async

top localhost:9000 activation_object src index.html node_modules/jquery/dist (no domain)

1 Serving from the file system... more never show

```
1 var SampleClass,
2   sample;
3
4 SampleClass = function(str){
5   var innerString = 'inner';
6   this.show = function(){
7     console.log(innerString);
8   };
9 };
10 sample = new SampleClass();
11 sample.show();
12
13 function showMessage(){
14   console.log('メッセージを表示');
15 };
16 showMessage();
17
18 var constructorSample,
19   ConstructorSample;
20 ConstructorSample = function(){
21   this.str = 'this is a sample';
22   this.show = function(){
23     console.log(this.str);
24   };
25 };
26
27 constructorSample = new ConstructorSample();
28 constructorSample.show();
```

Call Stack

ConstructorSample (anonymous function) main.js:22
main.js:27

Scope

Local

this: ConstructorSample

Global

Breakpoint

main.js:27
constructorSample = new ConstructorSample();

DOM Breakpoints

XHR Breakpoints

Event Listener Breakpoints

Event Listeners

Line 22, Column 3

Step intoの処理で順番に処理を進めていくことでコンストラクタ内のthisの生成が確認できる

thisの呼ばれ方その3：何かに所属している時のthis

例えば以下の様なコードがあったとします。

```
var belongTo;
belongTo = {
  str: '所属する'
};
belongTo.show = function() {
  console.log(this.str);
};

belongTo.show();
```

belongToというオブジェクトに属するstrを参照するshowが呼ばれる時にthisが生成されることを以下の手順で確認してみます。

実際に動作確認

先ほどと同様に設定したプロジェクトのjs/main.jsに上記のサンプルコードを追記します。

コードの修正が完了したら、GoogleChromeを使ってサイトにアクセスしてから、デベロッパーツールを表示させます。

1. belongTo.show()の横のあたりをマウスでクリックしてブレークポイントを設定
2. この状態でWebブラウザの再読み
3. 設定したブレークポイントの箇所で以下のように処理が止まるので、以下画面キャプチャで囲ってるStep intoのボタン箇所をクリックしながら処理を進めていきます

このリポジトリについて

The screenshot shows the Google Chrome Developer Tools interface. The left sidebar lists files: top, localhost:9000 (activation_object, src, index.html), and (no domain). The main area displays the contents of main.js. A red box highlights the call stack icon in the toolbar above the code editor. The right sidebar is the Call Stack panel, which is expanded. It shows a call stack entry: 'belongTo.show' (anonymous function) at main.js:35, which calls 'show' at main.js:38. The Scope section shows the context of the 'show' function. The Local section shows 'this' as an object with a 'show' method and a 'str' property set to '所属してる'. The Global section shows the global window object. The Breakpoints section has a checked checkbox for 'main.js:38 belongTo.show();'. The bottom status bar indicates 'Line 35, Column 3'.

```
var SampleClass,  
    sample;  
SampleClass = function(str){  
    var innerString = 'inner';  
    this.show = function(){  
        console.log(innerString);  
    };  
};  
sample = new SampleClass();  
sample.show();  
function showMessage(){  
    console.log('メッセージを表示');  
};  
showMessage();  
var constructorSample,  
    ConstructorSample;  
ConstructorSample = function(){  
    this.str = 'this is a sample';  
    this.show = function(){  
        console.log(this.str);  
    };  
};  
constructorSample = new Constructors  
constructorSample.show();  
var belongTo;  
belongTo = {  
    str: '所属してる'  
};  
belongTo.show = function() {  
    console.log(this.str);  
};  
belongTo.show();
```

4. `console.log(this.str)`の箇所に処理が移った所で`this`が生成されていることが確認出来るかと思います。

thisの呼び方その4：外部から書き換えられるthis

最後に外部から書き換えられるthisですが、これはcall/applyを

- メソッド名.call(this, strVariable)
- メソッド名.apply(this, arrayVariable)

という形で利用することで、thisが書き換わります。

実際に動作確認してみる

[JavaScript:call と apply の使い方と違いについて](#)という記事で比較的イメージしやすいサンプルコードがあったのでこちらを引用してcallを利用した時の動作について確認してみます

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  return this;
}

function Men(name, age) {
  Person.call(this, name, age);
  this.sex = 'male';
}

function Women(name, age) {
  Person.call(this, name, age);
  this.sex = 'female';
}

var father = new Men('Taro', 45);
var mother = new Women('Hanako', 35);
console.log(father.name);
console.log(father.age);
console.log(father.sex);
```

先ほどと同様に設定したプロジェクトのjs/main.jsに上記のサンプルコードを追記します。

コードの修正が完了したら、GoogleChromeを使ってサイトにアクセスしてから、デベロッパーツールを表示させます。

1. var father = new Men('Taro', 45)の横のあたりをマウスでクリックしてブレークポイントを設定してこの状態でWebブラウザの再読みします
2. 設定したブレークポイントの箇所で以下のように処理が止まるので、以下画面キャプチャで囲ってるStep intoのボタン箇所をクリックしながら処理を進めていきます

このリポジトリについて

```
1 var SampleClass,
2     sample;
3
4 SampleClass = function(str){
5     var innerString = 'inner';
6     this.show = function(){
7         console.log(innerString);
8     };
9 };
10
11 sample = new SampleClass();
12 sample.show();
13
14 function showMessage(){
15     console.log('メッセージを表示');
16 };
17 showMessage();
18
19 var constructorSample,
20     ConstructorSample;
21 ConstructorSample = function(){
22     this.str = 'this is a sample';
23     this.show = function(){
24         console.log(this.str);
25     };
26 };
27 constructorSample = new Constructors
constructorSample.show();
28
29 var belongTo;
30 belongTo = {
31     str: '所属してる'
32 };
33
34 belongTo.show = function() {
35     console.log(this.str);
36 };
37
38 belongTo.show();
39
40 function Person(name,age) {
41     this.name = name;
42     this.age = age;
43     return this;
44 }
45
46 function Men(name, age) {
47     Person.call(this, name, age);
48     this.sex = 'male';
49 }
50
51 function Women(name, age) {
52     Person.call(this, name, age);
53     this.sex = 'female';
54 }
55
56 var father = new Men('Taro', 45);
57 var mother = new Women('Hanako', 35)
58
59 console.log(father.name);
60 console.log(father.age);
61 console.log(father.sex);
62
63 console.log(mother.name);
64 console.log(mother.age);
65 console.log(mother.sex);
66
```

3. 上記ブレークポイントの段階ではまだthisが生成されていませんが、Step intoのボタンをクリックして処理を1つ進めるとthisが生成されていることが確認出来て、この段階ではMenが格納されてるのが確認できます。

このリポジトリについて

Developer Tools - http://localhost:9000/activation_object/index.html

Sources Content... Snippets Sources Network Timeline Profiles Resources Security Audits AngularJS

Serving from the file system... more never show

```
1 var SampleClass,
2     sample;
3
4 SampleClass = function(str){
5     var innerString = 'inner';
6     this.show = function(){
7         console.log(innerString);
8     };
9 }
10
11 sample = new SampleClass();
12 sample.show();
13
14 function showMessage(){
15     console.log('メッセージを表示');
16 }
17 showMessage();
18
19 var constructorSample,
20     ConstructorSample;
21 ConstructorSample = function(){
22     this.str = 'this is a sample';
23     this.show = function(){
24         console.log(this.str);
25     };
26 };
27 constructorSample = new ConstructorSample;
28 constructorSample.show();
29
30 var belongTo;
31 belongTo = {
32     str: '所属してる'
33 };
34 belongTo.show = function() {
35     console.log(this.str);
36 };
37
38 belongTo.show();
39
40 function Person(name, age) {
41     this.name = name;
42     this.age = age;
43     return this;
44 }
45
46 function Men(name, age) {
47     Person.call(this, name, age);
48     this.sex = 'male';
49 }
50
51 function Women(name, age) {
52     Person.call(this, name, age);
53     this.sex = 'female';
54 }
55
56 var father = new Men('Taro', 45);
57 var mother = new Women('Hanako', 35);
58
59 console.log(father.name);
60 console.log(father.age);
61 console.log(father.sex);
62
63 console.log(mother.name);
64 console.log(mother.age);
65 console.log(mother.sex);
66
```

Call Stack

Men (anonymous function)

Scope

Local

this: Men

age: 45
name: "Taro"

proto: Object

Global

Breakpoints

main.js:56
var father = new Men('Taro', 45);

DOM Breakpoints

XHR Breakpoints

Event Listener Breakpoints

Event Listeners

処理を1つ進めるとこのように
this が生成 されてます
この段階でのthisは Men

4. さらにStep intoのボタンをクリックして処理を1つ進めていきnew Womenの箇所が評価されるとthisの値がMenからWomenに変更されるのが以下のように確認できます。

このリポジトリについて

The screenshot shows the Google Chrome Developer Tools with the 'Sources' tab selected. The left pane displays the file structure and the content of `main.js`. The right pane shows the 'Call Stack' and 'Scope' panes. A red box highlights the object `this: Women` in the 'Scope' pane, which has properties `name: "Hanako"` and `__proto__: Object`. A red arrow points from this box to the line `this.name = name;` in the code editor. Another red box highlights the line `var mother = new Women('Hanako', 35);`, with a red arrow pointing from it to the line `new Woman(name, age)` in the code editor.

先程はthis がMenだった
のが今度はWomenに置
き換わっているのが確認
できます

new Womanの箇所が評価
されるところまでStep Into
で1つづつ処理を進める

```
1 var SampleClass,
2     sample;
3
4 SampleClass = function(str){
5     var innerString = 'inner';
6     this.show = function(){
7         console.log(innerString);
8     };
9 };
10
11 sample = new SampleClass();
12 sample.show();
13
14 function showMessage(){
15     console.log('メッセージを表示');
16 };
17 showMessage();
18
19 var constructorSample,
20     ConstructorSample;
21 ConstructorSample = function(){
22     this.str = 'this is a sample';
23     this.show = function(){
24         console.log(this.str);
25     };
26 };
27 constructorSample = new ConstructorSample();
28 constructorSample.show();
29
30 var belongTo;
31 belongTo = {
32     str: '所属してる'
33 };
34 belongTo.show = function() {
35     console.log(this.str);
36 };
37
38 belongTo.show();
39
40 function Person(name,age) { name = "Hanako", age = 35
41     this.name = name;
42     this.age = age; age = 35
43     return this;
44 }
45
46 function Men(name, age) { name = "Hanako", age = 35
47     Person.call(this, name, age);
48     this.sex = 'male';
49 }
50
51 function Women(name, age) { name = "Hanako", age = 35
52     Person.call(this, name, age);
53     this.sex = 'female';
54 }
55
56 var father = new Men('Taro', 45);
57 var mother = new Women('Hanako', 35);
58
59 console.log(father.name);
60 console.log(father.age);
61 console.log(father.sex);
62
63 console.log(mother.name);
64 console.log(mother.age);
65 console.log(mother.sex);
66
```

thisの値が書き換わるタイミングを把握しづらい場合にはStep intoを実行する度に以下赤枠で囲った箇所の値が動的に変化していきますのでそこを注意深く確認しておくことで書き換わるタイミングが確認できるかと思います

```
40 function Person(name, age) { name = "Taro", age = 45
41   this.name = name;
42   this.age = age; age = 45
43   return this;
44 }
45
46 function Men(name, age) { name = "Taro", age = 45
47   Person.call(this, name, age),
48   this.sex = 'male';
49 }
50
51 function Women(name, age) { name = "Hanako", age = 35
52   Person.call(this, name, age);
53   this.sex = 'female';
54 }
55
```

参考情報：call/applyを実際に利用するコードを知る

上記のような書き方を利用してcall/applyを活用するコードというのは慣れないいうちはあまり積極的に使うことは無いかもしれません、JavaScriptでの開発で利用される外部のライブラリ/フレームワークなどではこういう書き方を利用するコードがあるので、簡単に紹介だけしておきます。

Backbone.jsのModelの実装箇所

Backbone.js（バージョン1.3.3）で、Modelの機能を拡張する処理の所で以下コードで、初期化時にapplyを利用している記述があります。

```
var Model = Backbone.Model = function(attributes, options) {
  // ~中略~
  this.initialize.apply(this, arguments);
};
```

Backbone.jsに限らず他のフレームワークやライブラリで、親となる機能を継承する処理を実現したい場合にこういう形でapplyなどを利用するケースというのがありますので興味ある方は一度調べてみると色々勉強になるかと思います。

まとめ

この章では

- 関数呼び出し時にthisが生成される
- thisが呼ばれるパターンは4つある

ということを説明しました。

thisの概念を理解するのは簡単ではないかもしれないで理解が進まない箇所はサンプルコード内で適宜ブレークポイントを設定しつつ、Step intoで処理を1つづつ進めながらその時々の値を確認することで理解が深まるかと思いますのでぜひ、そのような形で復習していただければと思います。

はじめに

PHPやRubyなどの言語を利用してWebアプリケーション開発をしてる方の中でなんとなくJavaScript苦手という方が一定数いるような気がします。

そういう方向けにJavaScriptについて一歩掘り下げるための勉強会を開催していくこうと思ってます。

今回は**WebAPIと連携する処理**に対してどのようにテストを書くかというテーマをとりあげようと思います。

あらかじめJasmine + Karmaを組み合わせたテスト実行環境を事前にGitHub上に準備しておきますので、それを利用しながら以下内容を行おうと思います。

- まずはJasmineを使ってテストを書いてみる
- Jasmineのspyの機能を通じて擬似的にサーバーサイドと通信する状態を作るテストコード+それに対応するコードを書く
- テストやプロトタイプ用のダミーのREST APIを提供してくれる[JSONPlaceholder](#)というクラウドのサービスがあるのでそれを活用してJasmineのテストコードを書きつつ、実装を進める

Jasmineとは？

今回はJasmineとKarmaを組み合わせたテストの実行環境を構築してあるのですが

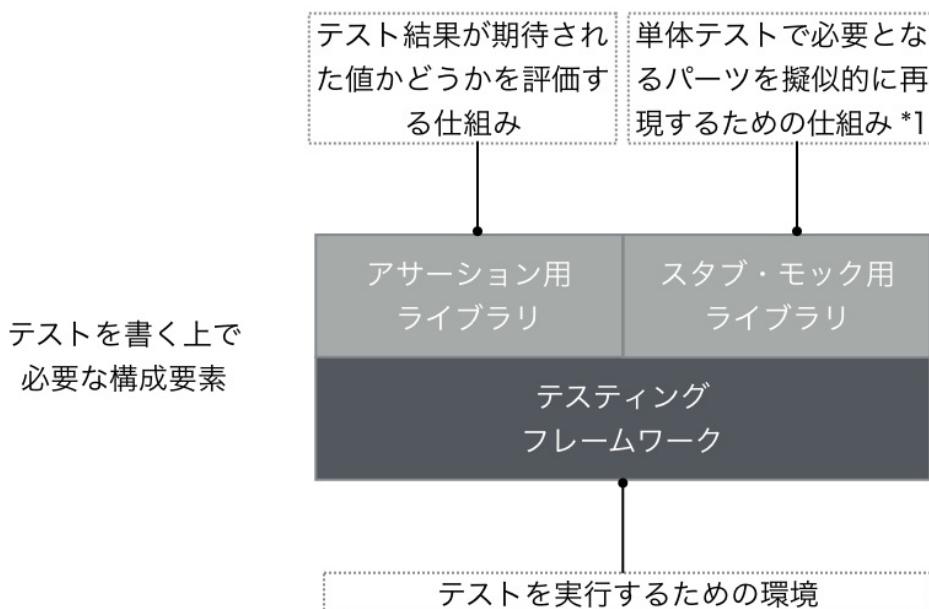
- テストを書く上で必要な構成要素
- Jasmineの特徴
- Jasmineの実行

という3つにわけて簡単にJasmineについて説明しておきます。

テストを書く上で必要な構成要素

テストを書くといっても、色々な構成要素から成り立つてるのでその点について説明しておきます

概念図



(*1) <http://morizyun.github.io/blog/rspec-model-controller-ruby-rails/>
より引用

個々の要素の説明

上記の概念図の個々の要素ですがそれぞれ以下のようにになります

- テスティングフレームワーク
 - テストを書く上での基盤となるもの
- アサーション用のライブラリ
 - テスト結果が期待された値かどうかを評価する仕組み

このリポジトリについて

- モック・スタブ用のライブラリ
 - 単体テストで必要となるパートを擬似的に再現するための仕組み
 - この説明は[RSpec](#)でテストを作るのに役立つ「モック/スタブ」の[シンプルな説明](#)から引用しています

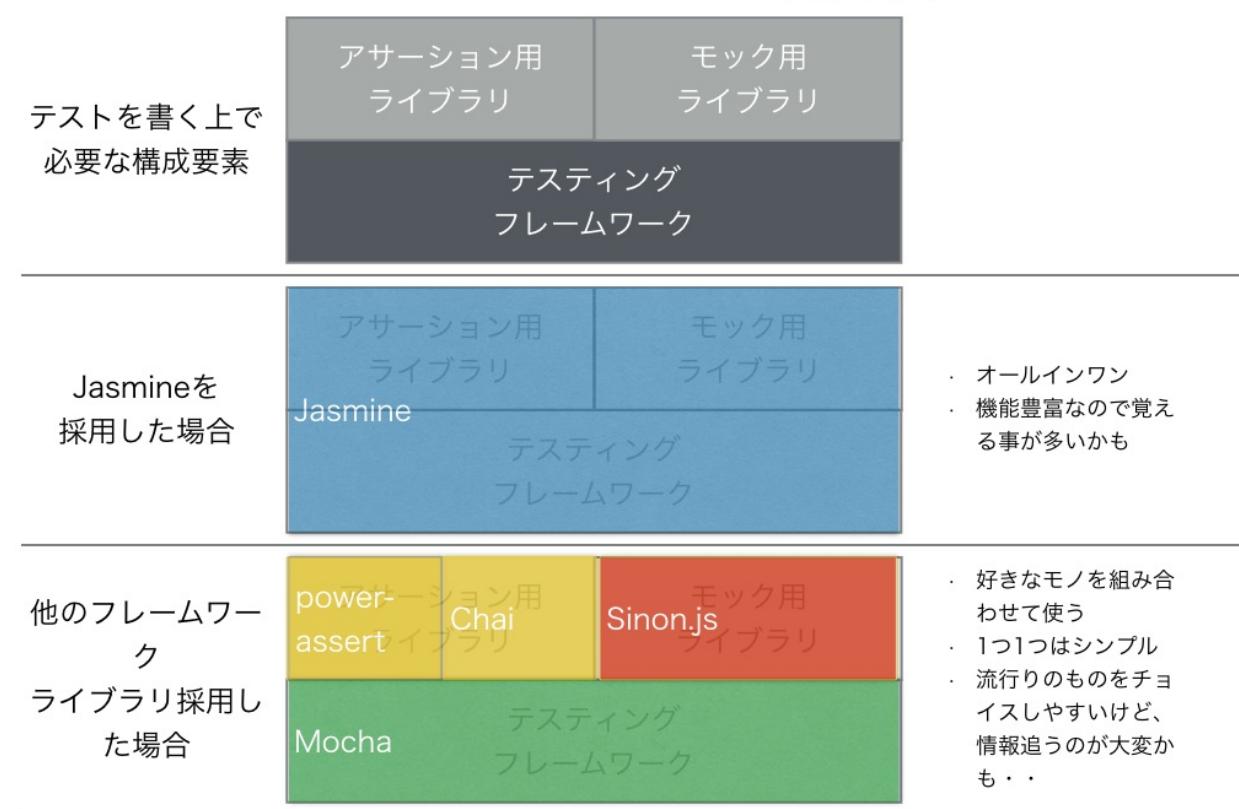
という形になります

Jasmineの特徴

上記で簡単にですが、テストを書いて実行するまでの構成要素について触れましたが、これを踏まえてJasmineの特徴をまとめてみます。

概念図

Jasmineの特徴

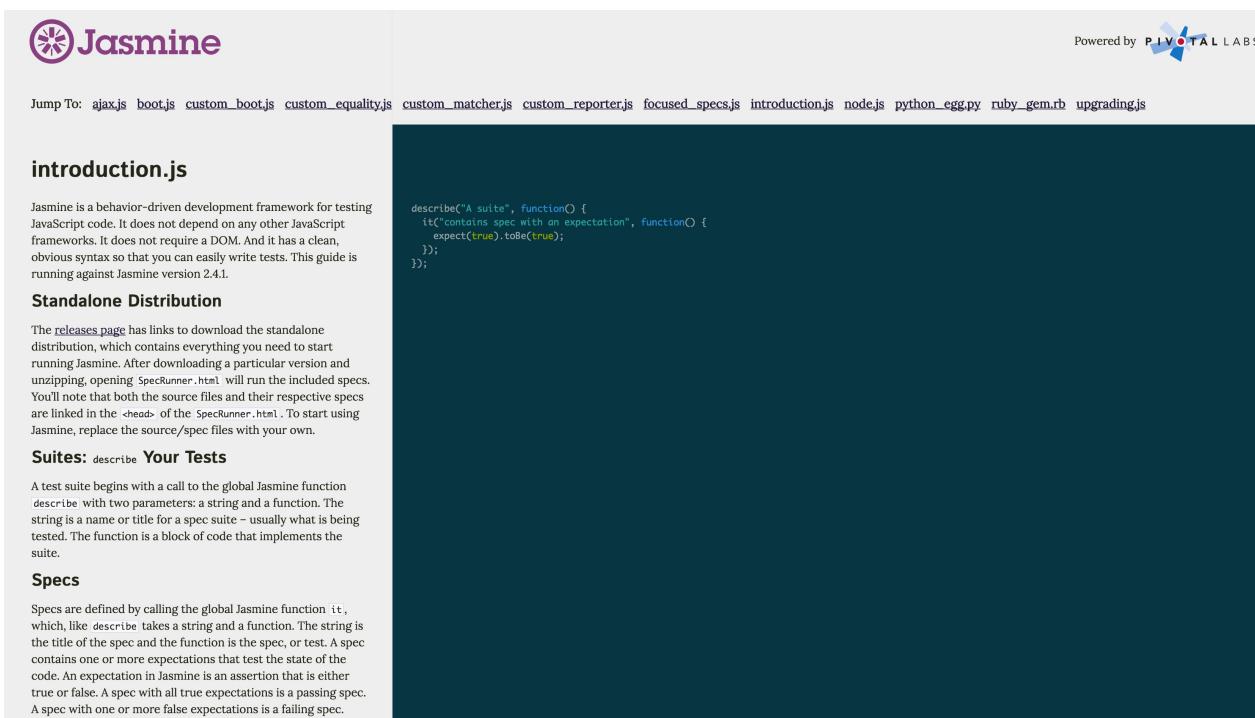


説明

図解しているようにJasmineはオールインワンなテスティングフレームワークなので、基本的にはこれを導入することで一通りのことが行えるようになります。

英語ですが以下のようにドキュメント自体もしっかりと整備されています。

このリポジトリについて



The screenshot shows the Jasmine documentation website. At the top left is the Jasmine logo (a stylized flower icon) and the word "Jasmine". At the top right is a "Powered by PIVOTAL LABS" logo. Below the header, there's a "Jump To:" menu with links to various files like ajax.js, boot.js, etc. The main content area has a dark background with white text. It features sections for "introduction.js", "Standalone Distribution", "Suites: describe Your Tests", and "Specs". Each section contains explanatory text and code snippets. The "introduction.js" section includes a snippet of JavaScript code:

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

上記ドキュメントを見てもらうとわかりますが、テストされた値の評価をする記法が豊富なので、初めてJavaScriptの単体テストを書く時にどこから手を付けて良いのか戸惑いやすいかもしれません。

Jasmineで書かれたコードの構造

Jasmineで書かれたコードは以下の様な構造になります。

Jasmineを使ったコード

```
describe('Gist', function() {
  beforeEach(function() {
    this.gist = new Gist();
  });
  describe('fetchメソッド', function() {
    beforeEach(function () {
      beforeEach①
    });
    it('定義されてる', function(){
    });
    it('APIからJSONが得られる', function(){
    });
  });
  describe('saveメソッド', function() {
    beforeEach(function () {
      beforeEach②
    });
    it('定義されてる', function(){
    });
    it('API経由で値を保存できる', function(){
    });
  });
});
```

コード実行イメージ



ポイントになる所をいくつか説明します。

describe()やit()内のシングルクオート内で説明を書く

通常、Jasmineを使ってテストを書く状況というのは、クラス単位に処理を分割したコードに対して実際にテストを書くケースがほとんどかと思います。

その時に、どのクラスのどのメソッドに対するテストなのかというのをテスト結果で把握しやすくするために、describe()やit()内のシングルクオート内で説明を書いていきます。

```
describe('SomeClassについて', function() {
  describe('xxx()メソッドについて', function() {
    it('定義されてる', function(){
      });
    });
  });
});
```

シングルクオート内の説明を英語で書くと、英語として自然な形で読める構造になるので理想的なのですが、慣れないうちは無理をして英語で書かず日本語でもいいと思います。

上記図解ではテストは4つ行われる

上記図解してのようなコードの場合だと全部で4つのテストが実行されることになり、it()で定義した内容が評価される構造になってます。

個々のテストを行う時に必要となる事前準備は beforeEachなどを行う

また、複数のテストを定義する場合に、個々のテストで共通して行いたい処理（例：クラスの初期化処理や、特定のメソッドで必要となるテストデーターの生成など）が出てくることがほとんどかと思いますが、その場合には、beforeEach()という仕組みを活用します。

上記の場合には

- メソッド単位でクラスの初期化処理をしたい
 - オレンジ色のbeforeEachの箇所
- fetch()、ならびにsave()のメソッド単位でも事前の準備をしたい
 - 黄色のbeforeEachの箇所
 - このようにしておくことでそれぞれのメソッド別にbeforeEachの処理を変更することが出来る

という形で使い分けていきます

beforeEach以外にも似たような仕組みがある

一度だけ実施したい場合にはbeforeAll()というメソッドがあります。またテスト実行後に何か処理を行いたい場合にはafterEach()/afterAll()というメソッドがあります。詳しいことは公式ドキュメントを参照してください

Jasmineを採用する時の参考情報

個人的には、Ruby/RailsでRSpecを使ってテストを書く習慣がベースとしてあって、RSpecと似たような記法のJasmineは違和感がないのでずっと使ってます。

もしも周囲にJavaScriptのテストに詳しい人がいて、かつ、その人がJasmine以外のものを使い慣れてるのならそちらを使っても良いかと思つてます。

※最近はシンプルなpower-assertとmochaを組み合わせて使うのが流行りっぽいです

Jasmineの実行

ターミナル上でコマンドを実行することで、Jasmineを通じてテストが実行されますが、今回のサンプルでは、JasmineとKarmaを組み合わせたやり方を採用してるので、それぞれの位置づけについても簡単にまとめてみました

概念図

Jasmineの実行



Karmaを組み合わせた理由について

Jasmine単体でももちろんテストを実行することは出来ますし、今回のサンプルアプリのようにWebAPIとの通信処理のコードに対するテストを書く場合にはViewのレンダリングをする必要がないので、Karmaを利用する必要は無いかもしれません。

今回は以下の理由によりKarmaを使うことにしてます

- 個人的に最近Karmaを使って作業することが多くすでにKarmaを実行させるための設定ファイルが元々あった
 - それを流用することで環境構築の手間が減らせる
- Jasmine単体でWebブラウザでのJavaScriptを実行させるためのHTMLを準備するのはまた違った準備が必要になるので作業上それなりに手間がかかる

Jasmineを使ったコードを実際に書く

先ほどでJasmineの構造について説明したのでこれを踏まえて実際にコードを書いてみたいと思います

はじめに最初に作るツールの仕様について

簡単に仕様についてまとめておきます

- Gistという名前のクラス
- そのクラスにはfetch()というメソッドが定義されてる
- fetch()メソッドを実行するとあらかじめ設定済のURLからJSONの情報を取得することが出来る

Gistクラスが定義されているファイル名をひとまずgist.jsとしておきます

fetch()について補足

- サーバーとの通信になるので非同期での処理を想定しており
- 通信処理が正常に完了した場合にdone()というコールバック関数が呼ばれる想定しています

最終的に実装されるgist.js

gist.jsを利用する側を仮にmain.jsとした場合には

```
var promise,
    gist = new Gist();
promise = gist.fetch();
promise.done(function(response){
  console.log(response) // gistにアクセスした結果が得られて、 JSON.p
});
```

という感じにすることで、gistの通信処理機能が利用できるようになります

実際にテストを書きながら作業を進めてみる

実際の仕事でメソッドが定義されることについてテストを書くのは冗長なので実際には無いかと思うのですが、まずはテストを書くことになるるためにこのテストを書いてみます。

specディレクトリを開いて、新規に**gist_spec.js**という名前のファイルを作ります。

ファイルを作ったら、以下の内容を記述します

```
describe('Gist', function() {
  describe('fetchメソッド', function() {
    it('定義されてる', function(){
      });
    });
  });
});
```

記述が終わったらターミナルを開いて、以下コマンドを入力して仮想マシンにsshします

```
vagrant ssh
```

sshが終わったら以下の要領で /vagrantディレクトリに移動した上で、Karmaを起動するコマンドを入力します。

```
cd /vagrant/
./node_modules/karma/bin/karma start ./karma.conf.js
```

Karma v0.13.22 server started at <http://localhost:9877/> というメッセージが表示されているのを確認したら、

このリポジトリについて

<http://192.168.33.39:9877/>

にアクセスすると、以下の様な画面が表示されるはずです



まずはJasmine自体に慣れてみる

実際の仕事でメソッドが定義されることについてテストを書くのは冗長なので実際には無いかと思うのですが、まずはテストを書くことになれるためにこのテストを書いてみます。

specディレクトリを開いて、新規に **gist_spec.js** という名前のファイルを作ります。

ファイルを作ったら、以下の内容を記述します

```
describe('Gist', function() {
  describe('fetchメソッド', function() {
    it('定義されてる', function(){
    });
  });
});
```

記述が終わったらターミナルを開いて、以下コマンドを入力して仮想マシンにsshします

```
vagrant ssh
```

sshが終わったら以下の要領で /vagrantディレクトリに移動した上で、Karmaを起動するコマンドを入力します。

```
cd /vagrant/
./node_modules/karma/bin/karma start ./karma.conf.js
```

Karma v0.13.22 server started at <http://localhost:9877/> というメッセージが表示されているのを確認したら、

<http://192.168.33.39:9877/>

このリポジトリについて

にアクセスすると、以下の様な画面が表示されるはずです



この段階ではテスト対象となる記述が全く無い状態なので、本来は意味がないのですが以下の様な内容を書いてみます

```
describe('Gist', function() {
  describe('fetchメソッド', function() {
    it('定義されてる', function(){
      expect(true).toBe(true);
    });
  });
});
```

上記内容を記述して保存した後にターミナルに戻ってみると自動的にテストが実行されて緑色の文字でTOTAL: 1 SUCCESSという文字が表示されてると思います

```
08 04 2016 17:53:27.394:INFO [watcher]: Changed file "/vagrant/spec/gist_spec.js".
✓ 定義されてる

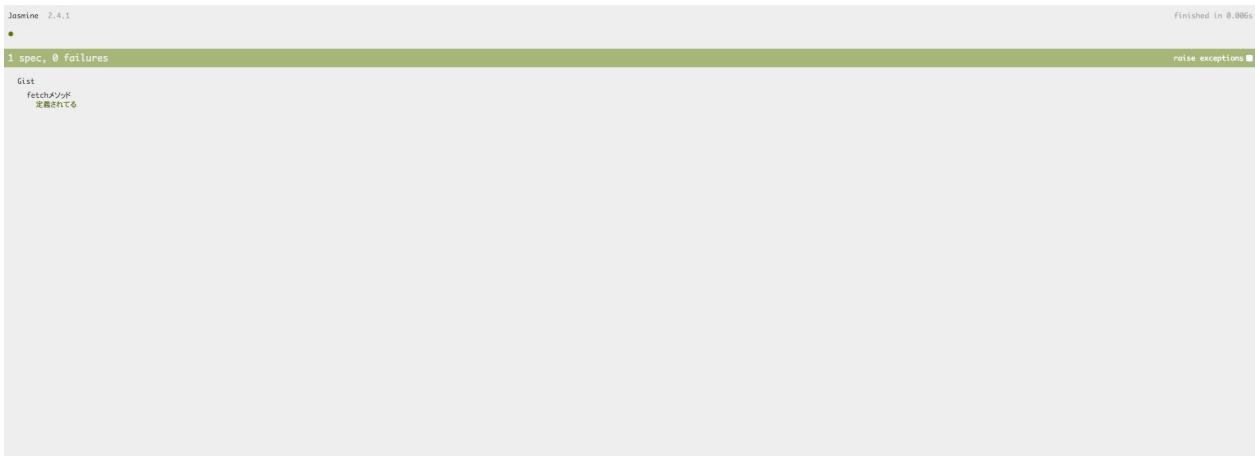
Chrome 49.0.2623 (Mac OS X 10.11.4): Executed 1 of 1 SUCCESS (0.939 secs / 0.002 secs)
TOTAL: 1 SUCCESS
```

また、Webブラウザで

<http://192.168.33.39:9877/debug.html>

このリポジトリについて

にアクセスすると、Webブラウザ上でもJavaScriptが実行されてその結果が表示されてるかと思います



The screenshot shows a browser window running a Jasmine test suite. At the top, there's a header bar with the text "Jasmine 2.4.1" on the left and "Finished in 0.000s" on the right. Below this is a green progress bar indicating "1 spec, 0 failures". Underneath the bar, the test code is visible, starting with "Gist" and "fetchメソッド 定義されている". The main body of the page is mostly blank.

期待される値と実際の値が両方共にtrueなので、当然のことながらこのテストは成功します。

fetch()メソッドが定義されてるというテストを書く

メソッドやプロパティが定義されてるかどうかを評価するために、
JasmineにはtoBeDefined()という記法があるのでそれを使ってテストを以下
の様に書いてみます

```
describe('Gist', function() {
  beforeEach(function() {
    this.gist = new Gist();
  });
  describe('fetchメソッド', function() {
    it('定義されてる', function(){
      expect(this.gist.fetch()).toBeDefined();
    });
  });
});
```

上記内容を記述して保存するとテストが再度自動的に実行されますがGist
クラスが定義されてるファイルをまだ実装していないので

```
this.gist = new Gist();
```

とインスタンス化しようとしても以下のようにエラーになります

このリポジトリについて

```
08 04 2016 17:53:27.394:INFO [watcher]: Changed file "/vagrant/spec/gist_spec.js".
✓ 定義されてる

Chrome 49.0.2623 (Mac OS X 10.11.4): Executed 1 of 1 SUCCESS (0.939 secs / 0.002 secs)
TOTAL: 1 SUCCESS

08 04 2016 18:02:36.067:INFO [watcher]: Changed file "/vagrant/spec/gist_spec.js".
✗ 定義されてる
  ReferenceError: Gist is not defined
    at Object.<anonymous> (/vagrant/spec/gist_spec.js:3:21)

  TypeError: Cannot read property 'fetch' of undefined
    at Object.<anonymous> (/vagrant/spec/gist_spec.js:7:23)

Chrome 49.0.2623 (Mac OS X 10.11.4) Gist fetch✗ ソッド 定義されてる FAILED
  ReferenceError: Gist is not defined
    at Object.<anonymous> (/vagrant/spec/gist_spec.js:3:21)
  TypeError: Cannot read property 'fetch' of undefined
    at Object.<anonymous> (/vagrant/spec/gist_spec.js:7:23)

Chrome 49.0.2623 (Mac OS X 10.11.4): Executed 1 of 1 (1 FAILED) ERROR (0.319 secs / 0.004 secs)
```

この段階ではまだGistクラスを定義したファイル自体がないので、当然のことながらエラーになるので、この段階では気にしないでください。

ひとまず次でGistクラスを定義していくことにします。

Gistクラスを定義したファイルを作成して最初のテストを成功させる

srcディレクトリを開いて、新規にgist.js というファイルを作り以下の内容を記述します

```
var Gist = (function() {
  function Gist(){
  }
  Gist.prototype.fetch = function(){
    return true;
  };
  return Gist;
})();
```

上記内容を記述して保存をすると再度自動的にテストが実行されて今度はテストが成功します。

```
08 04 2016 18:10:09.114:INFO [watcher]: Changed file "/vagrant/src/gist.js".
✓ 定義されてる

Chrome 49.0.2623 (Mac OS X 10.11.4): Executed 1 of 1 SUCCESS (0.3 secs / 0.001 secs)
TOTAL: 1 SUCCESS
```

今回は時間の都合でJavaScriptでのクラス定義の細かい説明は割愛しますが、別に資料としてまとめてるので、詳しいことはそちらを参照してください

JavaScriptとPHPでのクラス定義を対比させてみた #12

<https://github.com/h5y1m141/step-up-javascript/issues/12>

fetch()メソッドを仮実装した理由

先ほどのコードで

```
Gist.prototype.fetch = function(){
  return true;
};
```

とfetch()の実際の処理は単にtrueを返すだけの処理であり意味がない状態なのですが、この仮の段階の実装で成功しないテストは、本実装でも成功しない可能性が高くなり、そもそもどこに原因があるのか調査に手間取る可能性があるためにこのようにしています

テストに書き慣れない最初のうちは面倒かもしれません、

- テストを書く
- 仮実装をしてテストにパスすることを確認する
- 最終的な実装をする

という流れで作業していくことをオススメします。

fetch()メソッドのテストを書く

先ほどの段階ではメソッドが定義されてるというテストにパスした状態で、実際にfetch()自体の中身を実装してなかったのでfetch()メソッドのテストを書きながら作業を進めていきます。

fetch()メソッドを実行するとあらかじめ設定済のURLからJSONの情報を取得することが出来るという仕様を想定していましたが、実際にサーバー上にあらかじめ準備したJSONがあるのでそれを取得できるように実装を進めています

URLについて

Gist上に、JSONデーターを生成しており、そのJSONにアクセスするURLは以下になります。

<https://gist.githubusercontent.com/h5y1m141/f52eece296999105742c/raw/f3291233f69032a0b168192e11958575836833c2/react.json>

上記だと長いので、bit.lyの短縮URLも以下記載しておきます

<http://bit.ly/step-up-javascript-01>

最初に書いたテストを修正する

先ほど書いたテストを以下のように修正します

```
describe('Gist', function() {
  beforeEach(function() {
    this.gist = new Gist();
  });
  describe('fetchメソッド', function() {
    // (1) 以下を追加
    beforeEach(function () {
      spyOn(this.gist, 'fetch').and.callFake(function(){
        var deferred = $.Deferred();
        var dummyResponse = [
          {
            'author': 'おやまだひろし',
            'text': 'はじめてのReact.js'
          },
          {
            'author': 'Hiroshi Oyamada',
            'text': 'First Step ReactJS'
          }
        ];
        deferred.resolve(JSON.stringify(dummyResponse));
        return deferred.promise();
      });
    });
    // ここまでが(1)の追加箇所
    it('定義されてる', function(){
      expect(this.gist.fetch()).toBeDefined();
    });
    // (2) 以下を追加
    it('APIからJSONが得られる', function(){
      var promise,
        result;
      promise = this.gist.fetch();
```

```
promise.done(function(data){  
    result = JSON.parse(data);  
});  
expect(result[0].author).toEqual('おやまだひろし');  
});  
// ここまでが(2) の追加箇所  
});  
});
```

編集箇所のコードの説明について順番に説明します。

1についての説明

WebAPIと連携するテストを書く時に毎回アクセスするのは開発効率が悪かったりするので、単体テストで必要となるパースを擬似的に再現するための仕組みを活用することが多々あり、上記修正した(1)でその処理をしています。

個々の処理でポイントになる箇所を順番に説明していきます。

beforeEachとspyOnについて

beforeEachとspyOnは慣れないいうちは処理内容がイメージしづらいかもしれませんので順番に説明していきます。

beforeEachについて

テストを書く時に以下の様なことを行う必要が比較的多くあるのですが、そのような時にはbeforeEachの処理を利用することで対処できます

- 評価対象のクラスを初期化する
- (今回のように) WebAPIの振る舞いを擬似的に行うようなモックとなるオブジェクトを定義したい

使い方としては以下の様な形になります。

```
describe('対象のクラス名などを記述', function() {
  beforeEach(function() {
    // 事前に行いたい処理を記述
    // 以下の様な場合には評価対象のメソッド1、評価対象のメソッド2
    // それぞれのdescribe句で処理が行われます
  });
  describe('評価対象のメソッド1', function() {
    it('実際の評価内容を記述', function(){
    });
  });
  describe('評価対象のメソッド2', function() {
    it('実際の評価内容を記述', function(){
    });
  });
});
```

参考：beforeEachと似た処理のbeforeAll

このリポジトリについて

beforeEachと似た処理として、beforeAllがあるのですがこちらの場合には一度だけ実施されます。

以下はJasmineの公式ドキュメントから引用します。

```
describe("A spec using beforeAll and afterAll", function() {
  var foo;

  beforeEach(function() {
    foo = 1;
  });

  afterEach(function() {
    foo = 0;
  });

  it("sets the initial value of foo before specs run", function() {
    expect(foo).toEqual(1);
    foo += 1;
  });

  it("does not reset foo between specs", function() {
    expect(foo).toEqual(2);
  });
})
```

spyOnについて

Jasmineでは、Spyという機能で単体テストで必要となるパートを擬似的に再現することができます。

Spyの機能は豊富なので今回使ったspyOn機能についてのみ説明していきます

```
spyOn(this.gist, 'fetch')
```

上記のように記述することで

- beforeEach内で初期化されてたthis.gistオブジェクトが持つfetchメソッドがJasmineのSpy管理下に置かれます
- JasmineのSpy管理下に置かれたメソッドは**本来の実装内容は実施されずspyOnの定義内容に沿った処理**がされます

という形になります。

今回は、this.gistのfetch()が呼ばれた時に、callFake以降で定義されてる以下処理を行うように定義してます。

```
var deferred = $.Deferred();
var dummyResponse = [
  {
    'author': 'おやまだひろし',
    'text': 'はじめてのReact.js'
  },
  {
    'author': 'Hiroshi Oyamada',
    'text': 'First Step ReactJS'
  }
];
deferred.resolve(JSON.stringify(dummyResponse));
return deferred.promise();
```

上記のようにしたことで、実際のWebAPIにアクセスしたかのように振る舞ってくれます。

なお、上記のDeferredとPromiseについては、爆速でわかるjQuery.Deferred超入門という以下URLの記事がとてもわかりやすいので、詳しいことはこちらを御覧ください

<http://techblog.yahoo.co.jp/programming/jquery-deferred/>

APIからJSONが得られる記述の説明

spyOnの機能を活用してthis.gistのfetch()が呼ばれた時に、callFake以降で定義されてる以下処理を行うように定義してます

```
var deferred = $.Deferred();
var dummyResponse = [
  {
    'author': 'おやまだひろし',
    'text': 'はじめてのReact.js'
  },
  {
    'author': 'Hiroshi Oyamada',
    'text': 'First Step ReactJS'
  }
];
deferred.resolve(JSON.stringify(dummyResponse));
return deferred.promise();
```

また、APIからJSONが得られる記述を以下のようにして

```
// (2) 以下を追加
it('APIからJSONが得られる', function(){
  var promise,
      result;
  promise = this.gist.fetch();
  promise.done(function(data){
    result = JSON.parse(data);
  });
  expect(result[0].author).toEqual('おやまだひろし');
});
```

上記でdeferred.resolve()にてJSONを文字列化した値が返るように定義しているのでこここの流れは以下のようになっていきます

1. `this.gist.fetch()`が実施される
2. `fetch()`は`beforeEach`内の`spyOn`の処理内容が評価されるため、`deferred.promise()`が返る
3. `deferred.promise()`の場合にはそれに紐づくコールバック関数は`promise`オブジェクトの`done()`を通じて実行される
4. `done()`コールバック関数の`data`は、`dummyResponse`の値が返ってくるように`spyOn`にて定義しており、`JSON.parse()`処理でJSON化する
5. JSON化された`result`オブジェクトは配列になっており、`銭湯`の`author`プロパティの値が期待されたものであることを評価します

WebAPI連携のテストを書く

この章では、最近のWebアプリケーションでよくあるRESTの原則に従って実装されているWebAPIと連携する処理に対するJasmineのテストを書いてみます。

JSONPlaceholderとは？

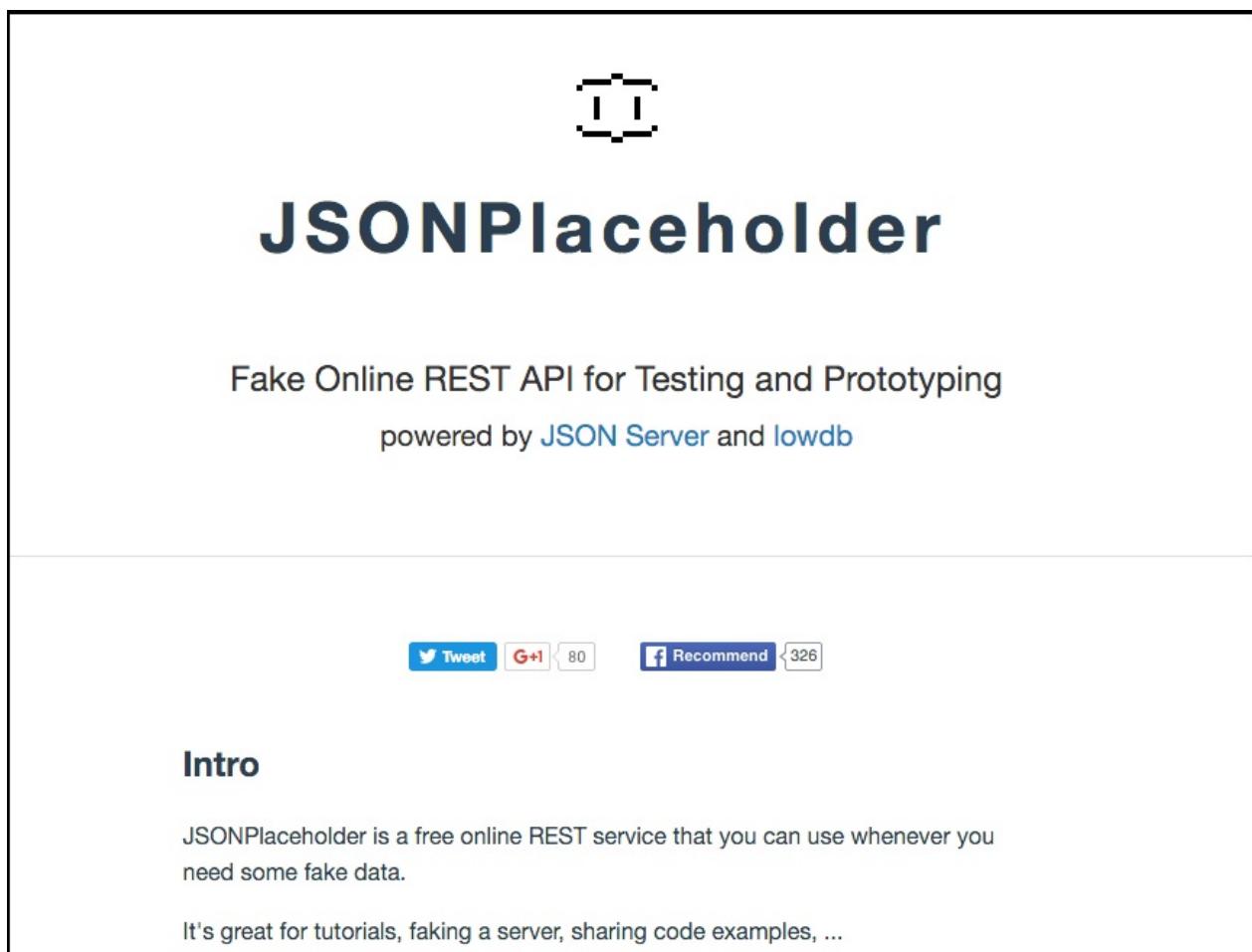
メジャーなWebサービスでは、開発者向けにRESTなWebAPIを提供しているのですが、利用にあたって事前準備が色々必要で煩雑な面があるので今回は

Fake Online REST API for Testing and Prototyping

<http://jsonplaceholder.typicode.com/>より

という特徴をもったJSONPlaceholder というサービスを利用します。

JSONPlaceholderはサインアップ不要で、利用できます。



The screenshot shows the homepage of JSONPlaceholder. At the top center is a small icon of a document with a double-headed arrow. Below it, the word "JSONPlaceholder" is written in a large, bold, dark blue sans-serif font. Underneath the title, the text "Fake Online REST API for Testing and Prototyping" is displayed in a smaller, dark blue font. Below that, it says "powered by [JSON Server](#) and [lowdb](#)". At the bottom of the main content area, there are social sharing buttons for Twitter, Google+, and Facebook, along with their respective counts: 80 and 326. Below these buttons, the word "Intro" is written in a dark blue font. Under "Intro", there is a short paragraph of text: "JSONPlaceholder is a free online REST service that you can use whenever you need some fake data. It's great for tutorials, faking a server, sharing code examples, ...".

JSONPlaceholderのAPIについて

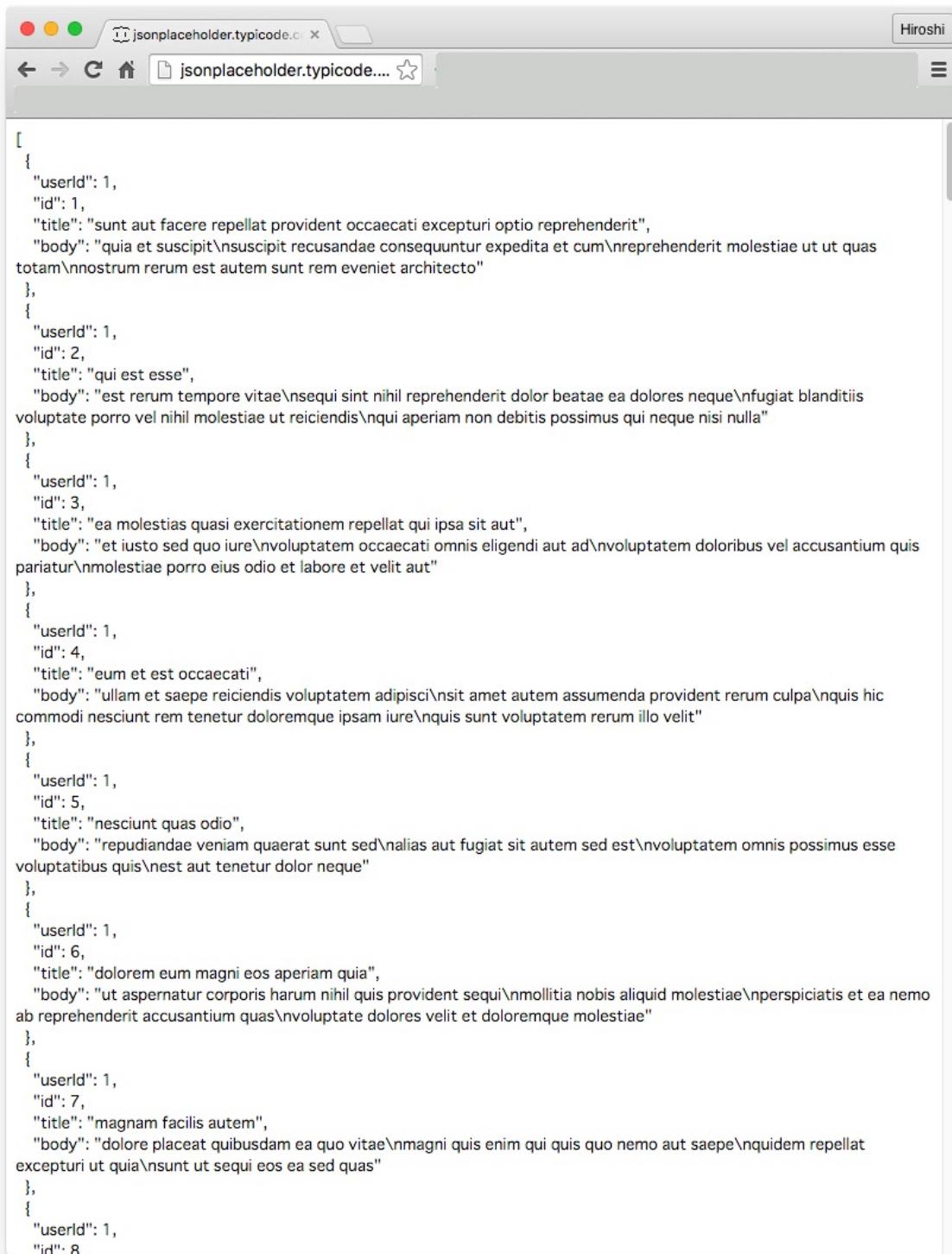
実際にプログラムを書く前にまずはJSONPlaceholderのAPIを利用することでどんな情報が取得できるかブラウザを通じて動作確認してみたいと思います。

投稿（Post）情報一覧を取得する

まずは以下URLにアクセスします。

<http://jsonplaceholder.typicode.com/posts>

画面上には取得されたJSONがこのような形で表示されるかと思います。



```
[  
  {  
    "userId": 1,  
    "id": 1,  
    "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",  
    "body": "quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut quas  
totam\\nnostrum rerum est autem sunt rem eveniet architecto"  
  },  
  {  
    "userId": 1,  
    "id": 2,  
    "title": "qui est esse",  
    "body": "est rerum tempore vitae\\nsequi sint nihil reprehenderit dolor beatae ea dolores neque\\nfugiat blanditiis  
voluptate porro vel nihil molestiae ut reiciendis\\nqui aperiam non debitis possimus qui neque nisi nulla"  
  },  
  {  
    "userId": 1,  
    "id": 3,  
    "title": "ea molestias quasi exercitationem repellat qui ipsa sit aut",  
    "body": "et iusto sed quo iure\\nvolutatem occaecati omnis eligendi aut ad\\nvolutatem doloribus vel accusantium quis  
pariatur\\nmolestiae porro eius odio et labore et velit aut"  
  },  
  {  
    "userId": 1,  
    "id": 4,  
    "title": "eum et est occaecati",  
    "body": "ullam et saepe reiciendis voluptatem adipisci\\nsit amet autem assumenda provident rerum culpa\\nquis hic  
commodi nesciunt rem tenetur doloremque ipsam iure\\nquis sunt voluptatem rerum illo velit"  
  },  
  {  
    "userId": 1,  
    "id": 5,  
    "title": "nesciunt quas odio",  
    "body": "repudiandae veniam quaerat sunt sed\\nalias aut fugiat sit autem sed est\\nvolutatem omnis possimus esse  
voluptatibus quis\\nest aut tenetur dolor neque"  
  },  
  {  
    "userId": 1,  
    "id": 6,  
    "title": "dolorem eum magni eos aperiam quia",  
    "body": "ut aspernatur corporis harum nihil quis provident sequi\\nmollitia nobis aliquid molestiae\\nperspiciatis et ea nemo  
ab reprehenderit accusantium quas\\nvolutate dolores velit et doloremque molestiae"  
  },  
  {  
    "userId": 1,  
    "id": 7,  
    "title": "magnam facilis autem",  
    "body": "dolore placeat quibusdam ea quo vitae\\nmagni quis enim qui quis quo nemo aut saepe\\nquidem repellat  
excepturi ut quia\\nsunt ut sequi eos ea sed quas"  
  },  
  {  
    "userId": 1,  
    "id": 8,
```

コメント (Comment)一覧を取得する

上記のURLでは投稿 (Post) 情報を表現するというリソース一覧を取得するものでしたが、以下URLにアクセスするするとコメント (Comment)一覧を取得することができます。

このリポジトリについて

<http://jsonplaceholder.typicode.com/comments>

特定の投稿情報やコメント情報を取得する

RESTなAPIになっているので、例えば、投稿情報のIDが1を取得したい場合には

<http://jsonplaceholder.typicode.com/posts/1>

にアクセスすることで取得できます。

また、コメントのIDが1を取得したい場合には

<http://jsonplaceholder.typicode.com/comments/1>

にアクセスすることで取得することが出来ます。

他に利用可能なAPIについて

JSONPlaceholderで定義されてるリソース一覧ですが、上記以外に2016年4月15日時点で以下が利用できるようです。

<http://jsonplaceholder.typicode.com/albums>

<http://jsonplaceholder.typicode.com/photos>

<http://jsonplaceholder.typicode.com/todos>

<http://jsonplaceholder.typicode.com/users>

JSONPlaceholderのAPIを簡単に確認しましたので、次からJasmineでテストを書きながらJSONPlaceholderのAPIを使ったJavaScriptのプログラムを実装していきます。

実装するクラスの仕様を考える

実装をする前に、まずはJSONPlaceholderクラスの仕様を考えてみたいと思います。

JSONPlaceholderのWebAPIは

- posts
- comments
- albums
- photos
- todos
- users

というそれぞれのリソースに対して取得、作成、削除・・・といった処理が行えるような構造になってます。

なるべくこの構造を維持したクラスが定義できると、自分自身だけではなく他の人からも利用する時にイメージしやすい構造になるかと思います。

そのため

```
var jsonPlaceHolder = new JsonPlaceHolder();
```

とした後に、例えば投稿情報に対する処理に対しては

```
var id = 1,  
postData = {  
    id: 1,  
    title: 'タイトル',  
    body: '本文'  
};  
jsonPlaceHolder.post.index();           // 一覧を取得  
jsonPlaceHolder.post.show(id);         // 指定したIDの投稿情報を取得  
jsonPlaceHolder.post.create(postData); // 投稿情報を作成  
jsonPlaceHolder.post.update(id, postData); // 指定したIDの投稿情報を更新  
jsonPlaceHolder.post.destroy(id);       // 指定したIDの投稿情報を削除
```

という形で行えて、コメント情報に対する処理に対しては

```
var id = 1,  
postData = {  
    id: 1,  
    title: 'タイトル',  
    body: '本文'  
};  
jsonPlaceHolder.comments.index();        // 一覧を取得  
jsonPlaceHolder.comments.show(id);      // 指定したIDのコメント情報を取得  
jsonPlaceHolder.comments.create(postData); // コメントを作成  
jsonPlaceHolder.comments.update(id, postData); // 指定したIDのコメント情報を更新  
jsonPlaceHolder.comments.destroy(id);     // 指定したIDのコメント情報を削除
```

が行えると理想的かと思うのでこういう処理が行えることを仕様として定義したいと思います。

なお、今回は、時間の都合もあって特定のリソース（投稿情報）の対しての説明だけになります。

クラスを定義して最初のテストを書く

先ほど想定する仕様を考えたので、ここから実際にJasmineでテストを書き上ながら作業を進めていきます。

WebAPIにアクセスする時のURL一覧

JSONPlaceholderのWebAPIを通じて投稿情報のリソースに対して処理をする場合のHTTPメソッドと エンドポイントの対応関係は以下のようになります。

HTTPメソッド	エンドポイント
GET	http://jsonplaceholder.typicode.com/posts
GET	http://jsonplaceholder.typicode.com/posts/1
POST	http://jsonplaceholder.typicode.com/posts
PUT	http://jsonplaceholder.typicode.com/posts/1
PATCH	http://jsonplaceholder.typicode.com/posts/1
DELETE	http://jsonplaceholder.typicode.com/posts/1

エンドポイントのURLの一部を共通化

それぞれのリソースに対する操作を考える時に、エンドポイントのURLの、 <http://jsonplaceholder.typicode.com> の箇所はすべて同じになるのでこの部分は共通化したほうがコードの見通しが良いかと思います。

具体的には

```
var jsonPlaceHolder = new JsonPlaceHolder();
console.log(jsonPlaceHolder.rootURL)
```

このリポジトリについて

というコードがあった場合に、console.logの出力結果として
<http://jsonplaceholder.typicode.com>が返るようなイメージです。

まずはこの部分の振る舞いについて、Jasmineでテストを書きながら作業を進めてみたいと思います。

最初のテストを書く

specディレクトリに、jsonPlaceHolder_spec.jsというファイルを作成して以下内容を書きます

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  describe('rootURLについて', function() {
    it('定義されてる', function(){
      expect(jsonPlaceHolder.rootURL).toBeDefined();
    });
    it('値が確認できる', function(){
      expect(jsonPlaceHolder.rootURL).toBe('http://jsonplaceholder.typicode.com');
    });
  });
});
```

Vagrantの環境にsshしてから

```
cd /vagrant/
./node_modules/karma/bin/karma start ./karma.conf.js
```

として、Karmaを起動させてから

<http://192.168.33.39:9877/>

にアクセスしてみてください。

テストを書いただけで、実際にクラスの実装が終わってないので、当然のことながら、ターミナル上ではテストが失敗してるかと思います。

```
JsonPlaceHolder
rootURLについて
✗ 定義されてる
    ReferenceError: JsonPlaceHolder is not defined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

    TypeError: Cannot read property 'rootURL' of undefined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

Chrome 49.0.2623 (Mac OS X 10.11.4) JsonPlaceHolder rootURLについて
    ReferenceError: JsonPlaceHolder is not defined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder
    TypeError: Cannot read property 'rootURL' of undefined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder
✗ 値が確認できる
    ReferenceError: JsonPlaceHolder is not defined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

    TypeError: Cannot read property 'rootURL' of undefined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

Chrome 49.0.2623 (Mac OS X 10.11.4) JsonPlaceHolder rootURLについて
    ReferenceError: JsonPlaceHolder is not defined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder
    TypeError: Cannot read property 'rootURL' of undefined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder
```

テストが通るようにJsonPlaceHolderクラスを実装する

このリポジトリについて

srcディレクトリ直下にjsonPlaceHolder.jsという名前のファイルを作成して以下内容を記述します。

```
var JsonPlaceHolder = (function(){
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
  };
  return JsonPlaceHolder;
})();
```

上記内容を記述すると、Karmaが自動的に内容を読みこんでテストが実行され以下のように今度はテストがパスするかと思います。

```
JsonPlaceHolder
rootURLについて
✓ 定義されてる
✓ 値が確認できる
```

最初のテストが書けたので、次ではWebAPIに対する処理を充実させていきます。

投稿情報一覧を取得する処理を実装する

テストを書きながら投稿情報一覧を取得する処理を順番に実装します

まずはindexメソッドが定義されてるテストを書く

少し周りくどいかもしれません、まずはindex()メソッドが定義されてるテストを書くことにします。

specディレクトリのjsonPlaceHolder_spec.jsを以下のように修正します

```
describe('JsonPlaceHolder', function() {
  //中略
  describe('rootURLについて', function() {
    //中略
  });
  // 以下追加内容
  describe('Postについて', function() {
    describe('indexについて', function() {
      it('定義されている', function(){
        expect(jsonPlaceHolder.post.index()).toBeDefined();
      });
    });
  });
});
```

これを追加してテスト結果を見ると、indexについては何も実装していないので以下のようにテストは失敗します。

```
JsonPlaceHolder
  rootURLについて
    ✓ 定義されている
    ✓ 値が確認できる
  Postについて
    indexについて
      ✗ 定義されている
      TypeError: Cannot read property 'index' of undefined
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

Chrome 49.0.2623 (Mac OS X 10.10.5) JsonPlaceHolder Postについて
  TypeError: Cannot read property 'index' of undefined
    at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

Chrome 49.0.2623 (Mac OS X 10.10.5): Executed 5 of 5 (1 FAILED)
TOTAL: 1 FAILED, 4 SUCCESS
```

テストに通るようにindex()をまずは仮実装する

ひとまずindex()が定義されているというテストが通ればOKなので、index()のロジックは気にせずひとまず以下のように実装します

```
var JsonPlaceHolder = (function(){
  function Post(){};
  Post.prototype.index = function(){
    return true;
  };
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
    this.post = new Post(this.rootURL);
  };
  return JsonPlaceHolder;
})();
```

indexの実装上のポイント

上記コードのポイントになる箇所を順番に説明します

1. JsonPlaceHolderの子となるオブジェクトがindex()などのメソッドを持つような構造にしたいので、JsonPlaceHolder内部で新たにPostというクラスを定義するためにこのように書きます。
2. ひとまずこの段階ではindex()の細かいロジックは無視して仮にtrueを返すようにします。
3. JsonPlaceHolderのpostに Postクラスをインスタンス化したオブジェクトを代入することで以下のようにした時に、実際にはPostクラスのindex()が呼びだされるようになります

```
var jsonPlaceHolder = new JsonPlaceHolder();
jsonPlaceHolder.post.index(); // 実際にはPost.prototype.indexの箇
```

上記のように実装するとテスト結果は以下のようにパスします

```
JsonPlaceHolder
rootURLについて
✓ 定義されている
✓ 値が確認できる
Postについて
indexについて
✓ 定義されている
```

indexを実装する

先ほどの段階ではメソッドが定義されてるというテストが通る仮の実装だったので、実際のindex()を実装していきます。

specディレクトリのjsonPlaceHolder_spec.jsを以下のように修正します

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  //中略
  describe('Postについて', function() {
    beforeEach(function () {
      spyOn(jsonPlaceHolder.post, 'index').and.callFake(function() {
        var fakeResult = [
          { id: 1, title: 'title1', body: 'body1' },
          { id: 2, title: 'title2', body: 'body2' }
        ];
        var deferred = $.Deferred();
        deferred.resolve(fakeResult);
        return deferred.promise();
      });
    });
  });
  describe('indexについて', function() {
    it('定義されてる', function(){
      expect(jsonPlaceHolder.post.index()).toBeDefined();
    });
    it('一覧を取得できる', function(){
      var result,
        promise;
      promise = jsonPlaceHolder.post.index();
      promise.done(function(response){
        result = response;
      });
      expect(result[0].id).toEqual(1);
    });
  });
});
```

このリポジトリについて

テストを追加した箇所に対してまだ実装が何もされてないので、再びテストに失敗するかと思います。

```
JsonPlaceHolder
  rootURLについて
    ✓ 定義されている
    ✓ 値が確認できる
  Postについて
    indexについて
      ✓ 定義されている
      ✗ 一覧を取得できる
      TypeError: promise.done is not a function
        at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

Chrome 49.0.2623 (Mac OS X 10.10.5) JsonPlaceHolder Postについて
  TypeError: promise.done is not a function
    at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

Chrome 49.0.2623 (Mac OS X 10.10.5): Executed 6 of 6 (1 FAILED)
```

今追加したテストに通るように、src/jsonPlaceHolder.jsを修正します

```
var JsonPlaceHolder = (function(){
  function Post(rootURL){
    this.rootURL = rootURL;
  };
  Post.prototype.index = function(){
    var params = {
      url: this.rootURL + '/posts',
      method: 'GET'
    },
      deferred = $.ajax(params);
    return deferred.promise();
  };
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
    this.post = new Post(this.rootURL);
  };
  return JsonPlaceHolder;
})();
```

上記のように修正してテストを再度実行すると以下のようにパスします。

```
JsonPlaceHolder
rootURLについて
✓ 定義されてる
✓ 値が確認できる
Postについて
indexについて
✓ 定義されてる
✓ 一覧を取得できる
```

この段階でのそれぞれの実装内容

この段階でのそれぞれのソースコードの全体を以下まとめておきます。

spec/jsonPlaceHolder_spec.js

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  describe('rootURLについて', function() {
    it('定義されてる', function(){
      expect(jsonPlaceHolder.rootURL).toBeDefined();
    });
    it('値が確認できる', function(){
      expect(jsonPlaceHolder.rootURL).toBe('http://jsonplaceholder.typicode.com');
    });
  });
  describe('Postについて', function() {
    beforeEach(function () {
      spyOn(jsonPlaceHolder.post, 'index').and.callFake(function() {
        var fakeResult = [
          {id: 1, title: 'title1', body: 'body1' },
          {id: 2, title: 'title2', body: 'body2' }
        ];
        var deferred = $.Deferred();
        deferred.resolve(fakeResult);
        return deferred.promise();
      });
    });
    describe('indexについて', function() {
      it('定義されてる', function(){
        expect(jsonPlaceHolder.post.index()).toBeDefined();
      });
      it('一覧を取得できる', function(){
        var result,
          promise;
        promise = jsonPlaceHolder.post.index();
        promise.done(function(response){
          result = response;
        });
        expect(result[0].id).toEqual(1);
      });
    });
  });
});
```

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

src/jsonPlaceHolder.js

```
var JsonPlaceHolder = (function(){
  function Post(rootURL){
    this.rootURL = rootURL;
  };
  Post.prototype.index = function(){
    var params = {
      url: this.rootURL + '/posts',
      method: 'GET'
    },
      deferred = $.ajax(params);
    return deferred.promise();
  };
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
    this.post = new Post(this.rootURL);
  };
  return JsonPlaceHolder;
})();
```

ここまでで投稿情報一覧を取得する処理は一通り完成したので、次は投稿情報に対する別の処理を追加していきます。

特定の投稿情報を取得する処理を実装する

先程は投稿情報一覧を取得する処理について、仕様をふまえて最初にテストを書き、そのテストが通るように実装するという流れで作業をしていきました。

今回も同様の流れで特定の投稿情報を取得する処理を実装していきます。

まずはshowメソッドが定義されてるテストを書く

specディレクトリのjsonPlaceHolder_spec.jsを以下のように修正します

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  describe('rootURLについて', function() {
    // 省略
  });
  describe('Postについて', function() {
    // 省略
    describe('indexについて', function() {
      // 省略
    });
    // 以下が追加箇所
    describe('showについて', function() {
      it('定義されてる', function(){
        expect(jsonPlaceHolder.post.show()).toBeDefined();
      });
    });
  });
});
```

上記追加したことでの結果、1つテストが失敗しているかと思います。

```
JsonPlaceHolder
rootURLについて
✓ 定義されている
✓ 値が確認できる
Postについて
indexについて
✓ 定義されている
✓ 一覧を取得できる
showについて
✗ 定義されている
TypeError: jsonPlaceHolder.post.show is not a function
at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

Chrome 49.0.2623 (Mac OS X 10.11.4) JsonPlaceHolder Postについて
TypeError: jsonPlaceHolder.post.show is not a function
at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

Chrome 49.0.2623 (Mac OS X 10.11.4): Executed 6 of 6 (1 FAILED)
TOTAL: 1 FAILED, 5 SUCCESS
```

テストに通るようにshowをまずは仮実装する

ひとまずshow()が定義されているというテストが通ればOKなのでロジックは気にせずひとまず以下のように実装します。

```
var JsonPlaceHolder = (function(){
  function Post(rootURL){
    this.rootURL = rootURL;
  };
  Post.prototype.index = function(){
    var params = {
      url: this.rootURL + '/posts',
      method: 'GET'
    },
      deferred = $.ajax(params);
    return deferred.promise();
  };
  Post.prototype.show = function(){
    return true;
  };
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
    this.post = new Post(this.rootURL);
  };
  return JsonPlaceHolder;
})();
```

上記実装したことで、先ほど失敗していたテストが以下のようにパスします。

```
JsonPlaceHolder
rootURLについて
✓ 定義されている
✓ 値が確認できる
Postについて
indexについて
✓ 定義されている
✓ 一覧を取得できる
showについて
✓ 定義されている
```

showについてのテストを1つ追加する

先程の段階ではメソッドが定義されてるというテストのみだったので、showで期待される動作について1つテストを追加します。

先ほどまでは投稿情報の仮のデータをfakeResultという名前で配列で格納していました。その仮のデータをindexだけではなくshowでも利用したいので変数宣言箇所を変更することで対応してます。（以下コメントの*1の箇所です）

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  describe('rootURLについて', function() {
    // 省略
  });
  describe('Postについて', function() {
    // 以下のように変更します
    beforeEach(function () {
      var fakeResult = [
        {id: 1, title: 'title1', body: 'body1' },
        {id: 2, title: 'title2', body: 'body2' }
      ]; // (*1)
      spyOn(jsonPlaceHolder.post, 'index').and.callFake(function()
        var deferred = $.Deferred();
        deferred.resolve(fakeResult);
        return deferred.promise();
      );
      spyOn(jsonPlaceHolder.post, 'show').and.callFake(function()
        var deferred = $.Deferred();
        deferred.resolve(fakeResult[0]);
        return deferred.promise();
      );
    });
    describe('indexについて', function() {

```

```
// 省略
});
describe('showについて', function() {
  it('定義されてる', function(){
    expect(jsonPlaceHolder.post.show()).toBeDefined();
  });
  // 以下が追加箇所
  it('指定された投稿情報が取得できる', function(){
    var result,
      promise;
    promise = jsonPlaceHolder.post.show(1);
    promise.done(function(response){
      result = response;
    });
    expect(result.id).toEqual(1);
  });
});
});
});
```

上記テストにパスするように実装する

showメソッドでは

- 投稿情報を特定するためにIDを引数にとる
 - エンドポイントとなるURLは、投稿情報一覧を取得する時と異なる
- という点を踏まえて、以下のように実装します

このリポジトリについて

```
var JsonPlaceHolder = (function(){
  function Post(rootURL){
    this.rootURL = rootURL;
  };
  Post.prototype.index = function(){
    var params = {
      url: this.rootURL + '/posts',
      method: 'GET'
    },
      deferred = $.ajax(params);
    return deferred.promise();
  };
  Post.prototype.show = function(id){
    var params = {
      url: this.rootURL + '/posts' + id,
      method: 'GET'
    },
      deferred = $.ajax(params);
    return deferred.promise();
  };
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
    this.post = new Post(this.rootURL);
  };
  return JsonPlaceHolder;
})();
```

上記のように実装することで以下のようにテストにパスします

```
JsonPlaceHolder
  rootURLについて
    ✓ 定義されている
    ✓ 値が確認できる
  Postについて
    indexについて
      ✓ 定義されている
      ✓ 一覧を取得できる
    showについて
      ✓ 定義されている
      ✓ 指定された投稿情報が取得できる
```

上記の実装でも問題はないのですが、index,showそれぞれ\$.ajaxの記述が重複している点が少し気になります。

今後、投稿情報を作成したり、更新したりという処理を増やすたびに、\$.ajaxの記述が重複していくので次で少しリファクタリングすることにします。

APIアクセス部分をリファクタリングする

先程まで実装したコードについて

全体像を改めて以下記載します

spec/jsonPlaceHolder_spec.js

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  describe('rootURLについて', function() {
    it('定義されてる', function(){
      expect(jsonPlaceHolder.rootURL).toBeDefined();
    });
    it('値が確認できる', function(){
      expect(jsonPlaceHolder.rootURL).toBe('http://jsonplaceholder.typicode.com');
    });
  });
  describe('Postについて', function() {
    beforeEach(function() {
      var fakeResult = [
        {id: 1, title: 'title1', body: 'body1' },
        {id: 2, title: 'title2', body: 'body2' }
      ];
      spyOn(jsonPlaceHolder.post, 'index').and.callFake(function(index) {
        var deferred = $.Deferred();
        deferred.resolve(fakeResult);
        return deferred.promise();
      });
      spyOn(jsonPlaceHolder.post, 'show').and.callFake(function(id) {
        var deferred = $.Deferred();
        deferred.resolve(fakeResult[0]);
      });
    });
  });
})
```

このリポジトリについて

```
        return deferred.promise();
    });
});

describe('indexについて', function() {
    it('定義されてる', function(){
        expect(jsonPlaceHolder.post.index()).toBeDefined();
    });
    it('一覧を取得できる', function(){
        var result,
            promise;
        promise = jsonPlaceHolder.post.index();
        promise.done(function(response){
            result = response;
        });
        expect(result[0].id).toEqual(1);
    });
});
describe('showについて', function() {
    it('定義されてる', function(){
        expect(jsonPlaceHolder.post.show()).toBeDefined();
    });
    it('指定された投稿情報が取得できる', function(){
        var result,
            promise;
        promise = jsonPlaceHolder.post.show(1);
        promise.done(function(response){
            result = response;
        });
        expect(result.id).toEqual(1);
    });
});
});
```

src/jsonPlaceHolder.js

```
var JsonPlaceHolder = (function(){
  function Post(rootURL){
    this.rootURL = rootURL;
  };
  Post.prototype.index = function(){
    var params = {
      url: this.rootURL + '/posts',
      method: 'GET'
    },
      deferred = $.ajax(params); // (*重複)
    return deferred.promise();
  };
  Post.prototype.show = function(id){
    var params = {
      url: this.rootURL + '/posts' + id,
      method: 'GET'
    },
      deferred = $.ajax(params); // (*重複)
    return deferred.promise();
  };
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
    this.post = new Post(this.rootURL);
  };
  return JsonPlaceHolder;
})();
```

この段階で気になる箇所

上記コードのコメントの(*重複)の箇所が気になるのでここをリファクタリングします。

どのようにするかというと

- パラメーターを引数に渡されてajaxを使ったリクエスト処理を行うだけの_requestというメソッドを新規に実装する

- index,showが実行される時には、_requestというメソッドを呼び出す
 - index,showはそれぞれリクエストのパラメーターの生成だけに専念する

ということを考えます。

上記の仕様をふまえてテストを修正

以下のように修正します。

今回一番のポイントは、以下コードの(1)の箇所のtoHaveBeenCalledです。

1. index/showが実行
2. 上記によって_requestが呼ばれる
3. _request内で\$ajaxが実行される

という流れになるので、index,showの実行時には必ず**_requestが呼ばれる**必要があります、その振る舞いについてしっかりとテストをしておく必要があるかと思います。

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  describe('rootURLについて', function() {
    it('定義されてる', function(){
      expect(jsonPlaceHolder.rootURL).toBeDefined();
    });
    it('値が確認できる', function(){
      expect(jsonPlaceHolder.rootURL).toBe('http://jsonplaceholder.typicode.com');
    });
  });
  describe('Postについて', function() {
    // 以下を追加
    describe('_request()メソッドについて', function() {
```

このリポジトリについて

```
beforeEach(function () {
    spyOn(jsonPlaceHolder.post, '_request').and.callThrough();
});
it('WebAPIにアクセスする個々のメソッドが呼ばれる時には通信処理を担当', function() {
    jsonPlaceHolder.post.index();
    expect(jsonPlaceHolder.post._request).toHaveBeenCalled();
    jsonPlaceHolder.post.show();
});
});
// 説明を追加するためにdescribeを挿入します
describe('WebAPIにアクセスするメソッドについて', function() {
    // これ以降は先程の処理をそのまま維持
    beforeEach(function() {
        var fakeResult = [
            {id: 1, title: 'title1', body: 'body1' },
            {id: 2, title: 'title2', body: 'body2' }
        ];
        spyOn(jsonPlaceHolder.post, 'index').and.callFake(function() {
            var deferred = $.Deferred();
            deferred.resolve(fakeResult);
            return deferred.promise();
        });
        spyOn(jsonPlaceHolder.post, 'show').and.callFake(function() {
            var deferred = $.Deferred();
            deferred.resolve(fakeResult[0]);
            return deferred.promise();
        });
    });
    describe('indexについて', function() {
        it('定義されてる', function(){
            expect(jsonPlaceHolder.post.index()).toBeDefined();
        });
        it('一覧を取得できる', function(){
            var result,
                promise;
            promise = jsonPlaceHolder.post.index();
            promise.done(function(response){
                result = response;
            });
        });
    });
});
```

```
    });
    expect(result[0].id).toEqual(1);
  });
});

describe('showについて', function() {
  it('定義されてる', function(){
    expect(jsonPlaceHolder.post.show()).toBeDefined();
  });
  it('指定された投稿情報が取得できる', function(){
    var result,
      promise;
    promise = jsonPlaceHolder.post.show(1);
    promise.done(function(response){
      result = response;
    });
    expect(result.id).toEqual(1);
  });
});
});
});
});
```

参考までにnode-twitterの構造を紹介

今回のようなWebAPIと連携するクラスを作る場合は

- パラメーターを生成する処理
- パラメーターを受け取って実際にWebAPIにリクエストを投げる処理

という設計にすることが多い印象を持っています。

参考までにGitHub上にあるnode-twitterの[コミットID6833c8a](#)時点のソースコードの一部を引用します

<https://github.com/desmondmorris/node-twitter/blob/master/lib/twitter.js>

```
'use strict';

/**
 * Module dependencies
 */

var url = require('url');
var stemparser = require('./parser');
var request = require('request');
var extend = require('deep-extend');

// Package version
var VERSION = require('../package.json').version;

function Twitter (options) {
    // 中略
}

// 中略
Twitter.prototype.__request = function(method, path, params, ca]
    // 中略
    this.request(options, function(error, response, data){

```

TwitterAPIと連携するような処理の場合には、パラメーターのチェックやエラーハンドリングなど細かい所をしっかり行う必要があるのですが、上記のような設計の場合にしておくことで、どこでその処理を行うのかが決めやすくなると思います。

修正したテストにパスするように実装する

以下のように実装します。

```
var JsonPlaceHolder = (function(){
  function Post(rootURL){
    this.rootURL = rootURL;
  };
  Post.prototype._request = function(params){ // (1)
    var deferred = $.ajax(params);
    return deferred.promise();
  };
  Post.prototype.index = function(){
    var params = {
      url: this.rootURL + '/posts',
      method: 'GET'
    };
    return this._request(params); // (2)
  };
  Post.prototype.show = function(id){
    var params = {
      url: this.rootURL + '/posts' + id,
      method: 'GET'
    };
    return this._request(params); // (2)
  };
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
    this.post = new Post(this.rootURL);
  };
  return JsonPlaceHolder;
})();
```

1. 新規に`_request`を実装して、ここで`$.ajax`を利用
2. これまで`index`や`show`で`$.ajax(params)`を呼び出していたが、その代わりに、新規に実装した`_request`を呼び出す

上記の変更によって今回新規に追加したテストだけでなく、以前からのテストもすべてパスします。

JsonPlaceHolder

rootURLについて

- ✓ 定義されている
- ✓ 値が確認できる

Postについて

_request()メソッドについて

- ✓ WebAPIにアクセスする個々のメソッドが呼ばれる時には通信処理を担当する

WebAPIにアクセスするメソッドについて

indexについて

- ✓ 定義されている
- ✓ 一覧を取得できる

showについて

- ✓ 定義されている
- ✓ 指定された投稿情報が取得できる

今回のようなリファクタリングを行う時に、従来から実装してた処理に影響が生じないか気になります。

これまでテストを書きながら実装を進めていたことで、今回のような変更をしても、従来の処理に影響が出ないことが確認でき、かつ、コードの見通しもよくなるのでテストを書きながら作業をすすめてきたことの恩恵を実感できるかと思います。

これまでの作業は、HTTPメソッドでGETを使った処理のみだったので次では、HTTPメソッドでPOSTを使う投稿情報を作成する処理を実装します

投稿を作成する処理を実装する

これまでには、HTTPメソッドのGETのみでしたが、最後にGET以外の処理についても簡単に説明しておきます。

まずはcreateメソッドが定義されてるテストを書く

specディレクトリのjsonPlaceHolder_spec.jsを以下のように修正します

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  describe('rootURLについて', function() {
    // 省略
  });
  describe('Postについて', function() {
    // 省略
    describe('WebAPIにアクセスするメソッドについて', function() {
      describe('indexについて', function() {
        // 省略
      });
      describe('showについて', function() {
        // 省略
      });
      // 追加箇所
      describe('createについて', function() {
        it('定義されてる', function(){
          expect(jsonPlaceHolder.post.create()).toBeDefined();
        });
      });
    });
  });
});
```

これまでの作業と同様に上記追加したことで、1つテストが失敗してるかと思います。

JsonPlaceHolder

rootURLについて

- ✓ 定義されている
- ✓ 値が確認できる

Postについて

_request()メソッドについて

- ✓ WebAPIにアクセスする個々のメソッドが呼ばれる時には通信処理を担当

WebAPIにアクセスするメソッドについて

indexについて

- ✓ 定義されている
- ✓ 一覧を取得できる

showについて

- ✓ 定義されている
- ✓ 指定された投稿情報が取得できる

createについて

- ✗ 定義されてる

TypeError: jsonPlaceHolder.post.create is not a function
at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

chrome 49.0.2623 (Mac OS X 10.10.5) JsonPlaceHolder Postについて

TypeError: jsonPlaceHolder.post.create is not a function
at Object.<anonymous> (/vagrant/spec/jsonPlaceHolder

chrome 49.0.2623 (Mac OS X 10.10.5): Executed 10 of 10 (1 FAILED)

TOTAL: 1 FAILED, 9 SUCCESS

テストに通るようにcreateをまずは仮実装する

ひとまずテストが通ればOKなのでロジックは気にせずひとまず以下のように実装します。

```
var JsonPlaceHolder = (function(){
  // 中略
  Post.prototype.show = function(id){
    // 中略
  };
  // 以下追加箇所
  Post.prototype.create = function(){
    var params;
    return this._request(params); // (2)
  };
  // 以下省略
})();
```

上記の実装により、先ほどまで失敗していたテストが以下のようにパスします。

```
JsonPlaceHolder
rootURLについて
✓ 定義されてる
✓ 値が確認できる
Postについて
_request()メソッドについて
✓ WebAPIにアクセスする個々のメソッドが呼ばれる時には通信処理を担当する
WebAPIにアクセスするメソッドについて
indexについて
✓ 定義されてる
✓ 一覧を取得できる
showについて
✓ 定義されてる
✓ 指定された投稿情報が取得できる
createについて
✓ 定義されてる
```

createのテストを1つ追加してそのテストに通るように実装する

先程はcreateが定義されているというテストのみだったので、実際の振る舞いについてテストを1つ追加して、それに通るように作業を進めていきます。

```
describe('JsonPlaceHolder', function() {
  var jsonPlaceHolder;
  beforeEach(function() {
    jsonPlaceHolder = new JsonPlaceHolder();
  });
  // 中略
  describe('Postについて', function() {
    // 中略
    describe('WebAPIにアクセスするメソッドについて', function() {
      beforeEach(function() {
        var fakeResult = [
          {id: 1, title: 'title1', body: 'body1' },
          {id: 2, title: 'title2', body: 'body2' }
        ];
        spyOn(jsonPlaceHolder.post, 'index').and.callFake(function() {
          var deferred = $.Deferred();
          deferred.resolve(fakeResult);
          return deferred.promise();
        });
        spyOn(jsonPlaceHolder.post, 'show').and.callFake(function() {
          var deferred = $.Deferred();
          deferred.resolve(fakeResult[0]);
          return deferred.promise();
        });
        // 以下を追加
        spyOn(jsonPlaceHolder.post, 'create').and.callFake(function() {
          var deferred = $.Deferred();
          // 基本的には引数に渡したparamオブジェクトがそのまま返ってくる
          // resolveにはparamオブジェクトをひとまずそのまま指定します
          deferred.resolve(param);
        });
      });
    });
  });
});
```

```
        return deferred.promise();
    });
    describe('indexについて', function() {
        // 省略
    });
    describe('showについて', function() {
        // 省略
    });
    describe('createについて', function() {
        it('定義されてる', function(){
            expect(jsonPlaceHolder.post.create()).toBeDefined();
        });
        // 以下を追加
        it('投稿を作成できる', function(){
            var result,
                promise,
                data = {
                    id: 1,
                    title: '投稿情報のタイトル',
                    body: '投稿情報の本文です'
                };
            promise = jsonPlaceHolder.post.create(data);
            promise.done(function(response){
                result = response;
            });
            expect(result.title).toEqual('投稿情報のタイトル');
        });
    });
});
});
```

投稿情報を作成するときには、

- HTTPメソッドをPOSTにする
- これまでの投稿情報を取得する処理では必要としてなかったdataプロ

パーティを準備する

- このプロパティにJsonPlaceHolderのAPIに対して作成する投稿情報の値をセットする

という処理を行い、実装は以下のようになります。

```
var JsonPlaceHolder = (function(){
  function Post(rootURL){
    this.rootURL = rootURL;
  };
  Post.prototype._request = function(params){
    var deferred = $.ajax(params);
    return deferred.promise();
  };
  Post.prototype.index = function(){
    var params = {
      url: this.rootURL + '/posts',
      method: 'GET'
    };
    return this._request(params);
  };
  Post.prototype.show = function(id){
    var params = {
      url: this.rootURL + '/posts' + id,
      method: 'GET'
    };
    return this._request(params);
  };
  Post.prototype.create = function(_params){
    var params = {
      url: this.rootURL + '/posts',
      method: 'POST',
      data: _params
    };
    return this._request(params);
  };
  function JsonPlaceHolder(){
    this.rootURL = 'http://jsonplaceholder.typicode.com';
    this.post = new Post(this.rootURL);
  };
  return JsonPlaceHolder;
})();
```

先ほどリファクタリングしておいたことで、今回のcreateの実装は比較的容易に済むことが実感できたかと思います。

最後に

投稿情報の取得、作成の処理を実装してもらいましたが、残りの処理（特定の投稿情報の更新処理や、特定の更新処理の削除）も

- HTTPメソッドをそれぞれの処理に合わせた形のものを指定
- 必要に応じてパラメータを設定

するだけで処理が行えるかと思いますので残りの処理はご自身で考えながら作業してみてください。

なお、これまでの処理を含めて残りの処理の回答は以下ディレクトリにまとめてあります。

https://github.com/h5y1m141/step-up-javascript/tree/develop/unit_test

付録

時間の都合で詳細に説明できていない情報をこちらにまとめています

この章では、JavaScriptでの変数について抑えておくべき2つのポイントについて解説します

グローバル変数とローカル変数

以下の様なコードを書いたとします。

```
var globalMessage = 'Global Hello';
alert(globalMessage); // (1)
showMessage(globalMessage); // (2)
alert(message); // (3)
function showMessage(text){
    var message = 'Hello';
    alert(message);
    alert(text);
    alert(globalMessage);
}
```

(1)から(3)の処理はそれぞれ以下のようにになります

1. `alert()`の結果として **Global Hello** が表示される
2. 引数に `globalMessage` が渡されて `showMessage()` が実行される。
`showMessage` の中では `alert` が3つあり、最初に関数内で宣言された変数の `message` に代入されてる **Hello** が表示される。その後に引数 `text` で渡された **Global Hello** が表示される。最後にグローバル変数として宣言された `globalMessage` に代入されてる **Global Hello** が表示される。
3. 何も表示されない

JavaScriptでの変数について抑えておくポイント

変数 `globalMessage` は、グローバル変数のためどこからでも参照することができます。

このリポジトリについて

```
var globalMessage = 'Global Hello';
alert(globalMessage);
showMessage(globalMessage);
alert(message);
function showMessage(text){
    var message = 'Hello';
    alert(message);
    alert(text);
    alert(globalMessage);
}
```

これはデキる

理由：変数globalMessage
グローバル変数なので、関
数内であっても参照できる

一方、`showMessage()`という関数内で宣言されてる変数については、変数内では参照できますが、変数の外部からは参照できません。

```
var globalMessage = 'Global Hello';
alert(globalMessage);
showMessage(globalMessage);
alert(message);
function showMessage(text){
    var message = 'Hello';
    alert(message);
    alert(text);
    alert(globalMessage);
}
```

これはデキない

理由：変数messageは
関数内で宣言されてる
ため参照できない

変数を全てグローバル変数で宣言するとどこからでも参照できるため使い勝手が良く、多用してしまいたくなるかと思います。

裏を返すとどこからでも参照出来ることは、裏を返すと自分が意図しない所でグローバル変数が違う値に変更される可能性も秘めているためグローバル変数の利用は必要最低限に抑えたほうが良いかと思います。

JavaScriptの変数のスコープ

JavaScriptの変数のスコープという概念について解説します。

ひとまず以下の様なサンプルコードがあったとします。

```
for(var i = 1; i < 6; i++){
    var message = i + '!!!';
    showMessage(message);
}
function showMessage(text){
    alert(message);
}
```

このforループでのブロック内で、変数messageを宣言しています。

Javaなどの言語では、ifやforなどの{}で囲まれたブロック単位で変数の有効範囲が決まっていますがJavaScriptには基本的にはそのような概念が存在しません。

そのため上記コードについては

```
var message;
for(var i = 1; i < 6; i++){
    message = i + '!!!';
    showMessage(message);
}
function showMessage(text){
    alert(message);
}
```

と書くのと同じ意味になります。

また、JavaScriptでのこういうループ文のサンプルの場合に、変数カウンターとして利用するiの変数宣言をforループ内で行ってますが、これもループ外で宣言してあるの同じになるため、最終的には以下のように書きま

このリポジトリについて

す。

```
var message,i;
for(i = 1; i < 6; i++){
    message = i + '!!!';
    showMessage(message);
}
function showMessage(text){
    alert(message);
}
```

```
var message,i;
var showMessage = function(text){
    alert(message);
}
for(i = 1; i < 6; i++){
    message = i + '!!!';
    showMessage(message);
}
```

if文自体は使いこなすのはそれほど難しい点はないかもしれません、JavaScriptでif文使う上での注意点があるため、簡単なコードを紹介しながらそのその点の解説をしたいと思います。

同じ値かどうか判定しながら処理をする

以下の様なサンプルコードがあった場合に(1)と(2)とでそれぞれどういう結果になるのかわかりますか？

```
var list1 = 1.0;
var list2 = "1.0";

// (1) 値の確認が同じかどうか確認する時に == を利用した場合
if(list1 == list2){
    console.log("list1 and list2 is same");
} else {
    console.log("list1 and list2 is not same");
}

// (2) 値の確認が同じかどうか確認する時に === を利用した場合
if(list1 === list2){
    console.log("list1 and list2 is same");
} else {
    console.log("list1 and list2 is not same");
}
```

両方同じ結果になりそうですが、結果はというと

```
[INFO] list1 and list2 is same
[INFO] list1 and list2 is not same
```

という形になります。

「あれ何でだろう？」

と素朴な疑問を持つかもしれませんし、私も昔この違いよく理解せず、ひとまず、全部 `==` としてましたが、ここを理解してないと思いがけないエラーを引き起こす可能性あるため、順を追って説明します。

まず `list1` と `list2` に格納されてる「型」を理解する

`list1` は数字として扱うことを意識してクオーテーション無しで値を代入し、一方、`list2` は文字として扱うことを意識してクオーテーションで囲いました。

(JavaScript含めて) プログラミング言語の変数には型というものがあります。

プログラミング言語によっては、変数を宣言する時に、その変数がどのような型なのかを同時に指定しなければいけないものがあります

JavaScriptは変数名だけを宣言すればOKなのですが、あくまで宣言する時の話であって、実際には、型というものが存在します。

`typeof` という関数を以下のように利用することでその変数の型を調べることが出来ます。

```
console.log("list1 type: " + typeof list1);
console.log("list2 type: " + typeof list2);
```

上記の実行結果はこのようになります

```
[INFO] list1 type: number
[INFO] list2 type: string
```

`list1` は数値、`list2` は文字列として扱われているため、頭のなかで数字同士の足し算をしてるつもりになって

`list1 + list2`

このリポジトリについて

とやっても、このようなケースの場合には、list1が数値でもlist2が文字列のため、前者のlist1が文字列として扱われてしまうため

11.0

という結果になります。

== と === の違い

Qiitaに[厳密等価演算子 javaプログラマのjavascript入門](#)という記事がありここから一部文章を引用します

==を「等価演算子」 ===を「厳密等価演算子」という。「厳密等価演算子」は型が同じかどうかチェックするが、「等価演算子」はチェックしない。「等価演算子」を使うと型が異なる場合↓のように変換して比較する。

- ・数値と文字列を比較するとき、文字列は数値に変換される。

<http://qiita.com/lasaya/items/d7d7a98e089d7fb91b84> より

==を使ったチェックの場合には、型の違いはチェックしないため、値だけのチェックになります。そのためlist1、list2とも、値としては1.0なので、

```
var list1 = 1.0;
var list2 = "1.0";

if(list1 == list2){
    console.log("list1 and list2 is same");
} else {
    console.log("list1 and list2 is not same");
}
```

とあった場合には、コンソール上では

```
console.log("list1 and list2 is same");
```

と表示されます。

一方で、===を利用した場合には、値のチェックのみならず、型のチェックも行い、値は両方共1.0ですが、list1は数値型、list2は文字列型で異なる型になります。そのためif文でチェックした場合には、

```
var list1 = 1.0;
var list2 = "1.0";

if(list1 === list2){
    console.log("list1 and list2 is same");
} else {
    console.log("list1 and list2 is not same");
}
```

というコードはelse句が評価されるため、コンソール上では

```
[INFO] list1 and list2 is not same
```

と表示されます。

まとめ

この章の内容が長くなったので最後簡単にまとめておきます。

if文を使いこなすのは簡単そうに見えるのですが、以下の様なポイントおさえておかないと想いがけないエラーが生じる可能性あります。

- 変数には型があるのをまずは理解しておく必要ある。
- if文を使って値のチェックをする時に以下2つあるのを意識する
 - 等価演算子 (==)
 - 厳密等価演算子 (===)
- list1 = 1.0 と list2 = "1.0" というような変数同士の演算を行う場合には上記2点をしっかりと理解しておく必要ある。

なお今回のようなちょっとしたサンプルアプリ程度なら**型の違い**を深く理解してなくてもエラーになる確率は少ないかもしれません。

ただある程度の規模のアプリケーションの開発をする際には型についてしっかりと抑えておく必要があります。

型についてもう少し踏み込んで勉強したい方は

- JavaScriptの型にはどのようなものがあるのか？
- 上記のような型が異なる場合の処理を行うための型変換

あたりを書籍などで調べて見ると良いかと思います。

このリポジトリについて

- Airbnb JavaScript スタイルガイド
- Truth, Equality and JavaScript

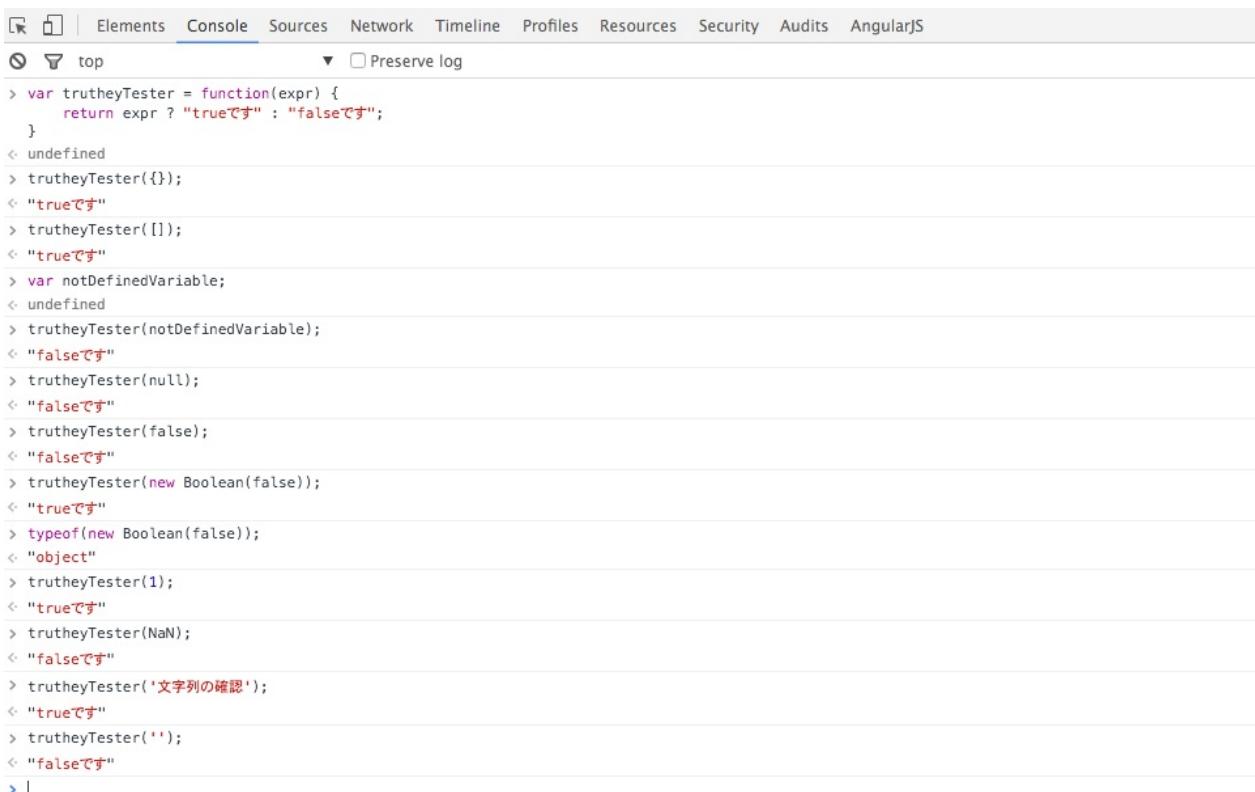
で条件判定について詳しく解説されてるので、そちらを参考にifについてより深く知るための情報をまとめました

true/falseを判定するメソッド

Truth, Equality and JavaScriptの中にあったコードを参考にまずは以下の様なメソッドを定義します

```
var trutheyTester = function(expr) {
    return expr ? "trueです" : "falseです";
}
```

このメソッドにいくつかの引数を設定して実際にどうなるかをChromeのConsoleの機能で確認してみます



The screenshot shows the Chrome DevTools Console tab. The console output displays the execution of the `trutheyTester` function with different inputs, demonstrating its behavior:

```
var trutheyTester = function(expr) {
    return expr ? "trueです" : "falseです";
}
trutheyTester({})
< "trueです"
trutheyTester([])
< "trueです"
var notDefinedVariable;
undefined
trutheyTester(notDefinedVariable);
< "falseです"
trutheyTester(null);
< "falseです"
trutheyTester(false);
< "falseです"
trutheyTester(new Boolean(false));
< "trueです"
typeof(new Boolean(false));
< "object"
trutheyTester(1);
< "trueです"
trutheyTester(NaN);
< "falseです"
trutheyTester('文字列の確認');
< "trueです"
trutheyTester('');
< "falseです"
> |
```

オブジェクトは trueと評価される

```
trutheyTester({});  
// "trueです"
```

配列はオブジェクトなのでtrueと評価される

```
trutheyTester([]);  
// "trueです"
```

undefined は false と評価される

```
var notDefinedVariable;  
// undefined  
trutheyTester(notDefinedVariable);  
// "falseです"
```

null は false と評価されます。

```
trutheyTester(null);  
// "falseです"
```

真偽値 は boolean型の値 として評価されます。

```
trutheyTester(false);  
// "falseです"
```

以下は実際にこういうコードを見たり書いたりすることがないと思うのですが、サンプルに乗っていたので記載しておきます。

```
trutheyTester(new Boolean(false));  
// "trueです"
```

ちなみにtrueになる理由ですが

- new Boolean(false)という記述は以下のようにオブジェクト型になる
- オブジェクトは前述したように常にtrueになる

というためです

```
typeof(new Boolean(false));  
// "object"
```

数値は true と評価されます。

```
trutheyTester(1);  
// "trueです"
```

しかし、+0, -0, or NaN の場合は false です。

```
trutheyTester(NaN);  
// "falseです"
```

文字列は true と評価されます

```
trutheyTester('文字列の確認');  
// "trueです"
```

しかし、空文字" の場合は false です。

```
trutheyTester('');  
// "falseです"
```


はじめに前置き

CakePHPなどでWebアプリケーションを書いてて、JavaScript側のコードが思ったように書けずに苦労するケースが割りとあるのかなと思ってます。

その要因となるものは色々あると思うのですが、考えられるものの1つとして、PHPでのクラス定義とJavaScriptでのクラス定義とがうまく対比できない所にあるのかなと思って、参考になる情報がないかググってみました。

[PHPとJavaScriptにおけるオブジェクト指向を比較する](#)というスライドは情報はとてもまとまっているのですが、自分が探していたそれぞれの言語でのクラス定義の方法を対比させるようなコード例が出ておらず、しかたがないので自分でコードを書いて対比させて折角なのでドキュメントにまとめることにしました。

注意

- JavaScriptでのクラス定義の方法は色々なアプローチがあると思うのですが、自分がなれ親しんでるCoffeeScriptで生成されるJavaScriptのクラスをサンプルとして活用しています。

確認した環境

- Mac OS X 10.9.5
- PHP
 - PHP 5.4.30 (cli) (built: Jul 29 2014 23:43:29)
- JavaScript
 - Node.jsで確認
 - v0.10.13

基本的なクラス定義の方法

メンバー変数に**message**を持った**SimpleClass**クラスというものを考えてみました。

PHPの場合

```
class SimpleClass {  
    public $message = 'Public';  
}
```

となるかと思います

JavaScriptの場合

```
var SimpleClass;  
SimpleClass = (function(){  
    function SimpleClass() {  
        this.message = 'Public';  
    }  
    return SimpleClass;  
})();
```

という形になります。

参考まで上記のJavaScriptを生成したCoffeeScript

もしかしたらPHPを書き慣れてる方にはCoffeeScriptの方が意図が伝わりそうな気がするのでCoffeeScriptでのコードも書いておきます

このリポジトリについて

```
class SimpleClass
  constructor:() ->
    @messsage = 'Public'
```

インスタンス化してパブリックな変数をターミナル上に表示する

PHPの場合

`new`してインスタンス化した上で、メンバー変数を以下のようにして呼び出すことが出来ます

```
class SimpleClass {
    public $message = 'Public';
}
$obj = new SimpleClass();
print $obj->message;
```

JavaScriptの場合

クラス定義はちょっと違和感があるかもしれません、メンバー変数を呼び出す方法はPHPのそれとあまり大差がないかと思います。

```
var SimpleClass;
SimpleClass = (function(){
    function SimpleClass() {
        this.message = 'Public';
    }
    return SimpleClass;
})();
var obj = new SimpleClass();
console.log(obj.message);
```

参考まで上記のJavaScriptを生成したCoffeeScript

CoffeeScriptの場合にも大差がないと思います

このリポジトリについて

```
class SimpleClass
  constructor:() ->
    @messsage = 'Public'

obj = new SimpleClass()
console.log(obj.message)
```

パブリックなメソッドを定義する

さて今度は**printHello**というパブリックなメソッドを定義してみましょう。

PHPの場合

PHPの場合には以下のようになるかと思います。

```
class SimpleClass {  
    public $message = 'Public';  
    public function printHello(){  
        print("newしたオブジェクトからは呼び出せる");  
    }  
}  
$obj = new SimpleClass();  
$obj->printHello();
```

JavaScriptの場合

JavaScriptの場合には、クラス名の**SimpleClass**の後に、ドット(.)と**prototype**を付けて、その後にメソッド名を付けます。prototypeの概念説明はやりだすとキリがない（というかそこまで的確に説明する自信がない）ので[このあたり](#)を読むのが良いのかなと思います。

```
var SimpleClass;
SimpleClass = (function(){
    function SimpleClass() {
        this.message = 'Public';
    }
    SimpleClass.prototype.printHello = function () {
        console.log("newしたオブジェクトからは呼び出せる");
    };
    return SimpleClass;
})();
var obj = new SimpleClass();
obj.printHello();
```

参考まで上記のJavaScriptを生成したCoffeeScript

CoffeeScriptの場合、`function printHello(){}>`に相当する記述が `printHello:()->`という形でメソッドを定義します。

※CoffeeScriptのメソッド定義の記法に最初違和感を感じるかもしれませんね

```
class SimpleClass
  constructor:() ->
    @messsage = 'Public'
  printHello:() ->
    console.log("newしたオブジェクトからは呼び出せる")
obj = new SimpleClass()
console.log(obj.message)
```

プライベートな変数とその変数を得るためにメソッドを定義

最後に**msg**というプライベートな変数を定義しつつ、その変数を得るためにメソッドを定義してみましょう。

PHPの場合

まず、プライベート変数だけ定義してみます

```
class SimpleClass {
    public $message = 'Public';
    private $msg = 'Private';
    public function printHello(){
        print("newしたオブジェクトからは呼び出せる");
    }
}
```

上記定義した後に

```
$obj = new SimpleClass();
print $obj->msg;
```

とすると**Cannot access private property SimpleClass::\$msg** というメッセージが表示されるかと思います。

名前の通りプライベートな変数なので外部からは直接呼び出せません。そこで、このプライベートな変数にアクセスできる**getMsg**というパブリックなメソッドを定義します。

```
class SimpleClass {  
    public $message = 'Public';  
    private $msg = 'Private';  
    function printHello(){  
        print("newしたオブジェクトからは呼び出せる");  
    }  
    public function getMsg(){  
        print($this->msg);  
    }  
}
```

上記のようにした上で、以下のようにgetMsg()を通じて、SimpleClassクラスの中で定義されてるプライベートな変数にアクセスして画面表示されます

```
$obj = new SimpleClass();  
$obj->getMsg();
```

JavaScriptの場合

以下のようにすることで、プライベートなmsg変数を定義できます。パブリックな変数を定義した際には、this.xxxとしましたが、プライベートな変数の場合には、JavaScriptの変数スコープが関数単位である特徴を活かして以下のようにしてあげればプライベートなものとして利用できます。

```
var SimpleClass;
SimpleClass = (function(){
  var msg = 'Private';
  function SimpleClass() {
    this.message = 'Public';
  }
  SimpleClass.prototype.printHello = function () {
    console.log("newしたオブジェクトからは呼び出せる");
  };
  return SimpleClass;
})();
```

上記定義したものに対して

```
var obj = new SimpleClass();
console.log(obj.msg());
```

とすると、以下のようにエラーになります。

```
console.log(obj.msg());
^
TypeError: Object #<SimpleClass> has no method 'msg'
```

プライベートな**msg**にアクセスするために、**getMsg()**メソッドを定義するには以下のようにします。

```
var SimpleClass;
SimpleClass = (function(){
    var msg = 'Private';
    function SimpleClass() {
        this.message = 'Public';
    }
    SimpleClass.prototype.printHello = function () {
        console.log("newしたオブジェクトからは呼び出せる");
    };
    SimpleClass.prototype.getMsg = function () {
        console.log(msg);
    };
    return SimpleClass;
})();
```

上記定義したものに対して以下のようにすると、画面上に**Private**という文字が表示されます

```
var obj = new SimpleClass();
console.log(obj.msg());
```