Web-othello についてのお話と小噺

KA-FU-　　hinata

目次

第 I 章

まえがき

72 回生 (当時中 1) の KA-FU-です。今回の文化祭ではゲーム展示の一つ、Web-othello を作りました。Web-othello の作成者は二人で、私はオセロ本体と AI [1] を担当しました。もう一人は、同級生の期待の Web マン [2] 、hinata です。Web マンなので、もちろん Web に関する様々なことを手伝ってもらいました。この部誌の内容は題名の通り、Web-othello 作成中のエピソードに関連したお話や、小噺のようなものです。また、この部誌はかなりの部分が小噺、つまりネタになっています。なので、まじめに書かれた部分だけ読みたい方は次のリンクから飛んで読んでください。

また、著者の部分に「KA-FU-」と「hinata」とありますが、一部のセクションのみを hinata が書き、それ以外は私、KA-FU-が書かせて頂くという形式になっております。ご了承ください。

[1] 人工知能
[2] Web 開発をする人

第 II 章

Web-othello 作成の流れ

1　オセロ本体

私は作成当初 C という言語 (通称 C 言語) を使ってプログラミングをしていました。NPCA では、最初に勉強するプログラミング言語として C 言語を多くの人にすすめています。 Wikipedia によると、

> 『UNIX [1] および C コンパイラ [2] の移植性を高めるために開発してきた経緯から、オペレーティングシステム [3] カーネル [4] およびコンパイラ向けの低レベル記述ができる』

だそうです。

とある一般人「オペレーティングシステムとかよくわかんないですけどそれ、簡単に Web で使えるんですか」

そうですよね、そんな気がしますよね。実際難しくて諦めました...。
　ということで、JavaScript というウェブブラウザ [5] 上で動く唯一の言語 (通称 JS) を使うことにしました。

とあるロリコン「えっ JS...？女子小学生のことですか」

違います、日頃 JS を使ってる hinata が怒りだすのでやめてください。 Wikipedia によると

---

[1] 後述するオペレーティングシステムの種類の一つ
[2] プログラミング言語をコンピューターがわかるものに変換してくれるもの
　C コンパイラは C 言語を変換する
[3] Windows や OS X 等のこと
[4] オペレーティングシステムの中でも中心的な機能の部分
[5] Chrome や Safari、Firefox、InternetExplorer 等のこと

> 『実行環境が主にウェブブラウザに実装され、動的なウェブサイト [6] 構築や、リッチインターネットアプリケーションなど高度なユーザインタフェース [7] の開発に用いられる。』

まぁ、Web サイトで使うんですよ、はい。ここらへんは Web マンである hinata に説明してもらったほうがいいのですが、熱弁しだすので遠慮しておきましょう。

JS は C 言語に書き方が似ているので、意外とオセロ本体は簡単に完成しました。そう、オセロ本体は...。

2　AI

オセロ本体と一緒に AI も作ったのですが、考えるのに非常に時間がかかる上に、別にそこまで強くないというお粗末なものでしたので、改良することになりました。

新しい AI は MySQL というツールを使ってデータベースを操作する贅沢なものにしようと考えました。データベースとは、"データ"という単語が入っていることから想像できるように、使いやすく整理してあるたくさんのデータです。Wikipedia によると

> 『特定のテーマに沿ったデータを集めて管理し、容易に検索・抽出などの再利用をできるようにしたもの。』

だそうです。まぁ、データの集まりですよ、はい。このデータベースはサーバーにあって、ブラウザ上で動く JS を使って操作をするのは無理らしいので、仕方なく違う言語を使うことにしました。その言語は、PHP: Hypertext Preprocessor というもの (通称 PHP) です。

とある主婦「えっ DHC？よく健康食品を買わせてもらってるわ。」

聞き間違いも甚だしいです、化学研究部員でもある hinata がドコサヘキサなんとか [8] とか言い出すのでやめてください。 Wikipedia によると、

> 『PHP はサーバーサイド・スクリプト言語として利用されており、Web サーバ上で動作し、Web サーバ上で PHP スクリプトの文書が要求されるたびに、その PHP スクリプトが実行され、結果をウェブブラウザに対して送信する。』

まぁ、サーバーで動くんですよ、はい。で、できた AI がデータベースを存分に使うくせにランダムに打つのとあまり変わらないというもので、更に悪くなってしまました。うひーつらい。ということで、また新しい AI を考える羽目になったのでした、めでたしめでたし。

---

[6] ページをもう一度読み込まなくても、そのまま表示が変化するウェブサイトなど

[7] そのシステムをユーザーが操作するのに必要なもの

[8] ドコサヘキサエン酸 (DHA)
DHC の健康食品の中ににこれを主成分とするものがある

3　めでたくない

実はこの頃に Lisp という言語を使うようになっていました。Wikipedia によると

『LISP は現在でも広く使われている年代物の言語の一つである。』

だそうです。Lisp を使うようになってからというもの、Lisp 以外の言語を読むことが難しくなってきました。なので、PHP を使って新しい AI を作るのをやめ、PHP 内で Lisp を実行することで Lisp を使った AI を作成しました。~~しかし、やっぱりお粗末なものにしかならず、諦めました。~~ こ、今回は、まだマシなものが出来上がったので、い、一件落着です。

第 III 章

本題

1　Web、襲来

本題に入ってすぐに余談なのですが、Web-othello 作成のいきさつをば。僕は一学期の終わりから二学期の初めの間 (夏休みは何もしてませんでしたが) に、C 言語で CUI [1] のオセロを作りました。文化祭のゲーム展示ではこれを GUI [2] 化して使いまわす気まんまんでした。しかし、二学期に入ってから一ヶ月ほどたったときに hinata が入ってきて、なんと

hinata「プログラミングに興味を持ち、家で JS を勉強していたので少しなら書ける。」

というではありませんか。これは利用するしかないと Web 化を決意した次第であります。まさかあんな事になるとはこの時誰も思いませんでした...

2　動かない、ページ

僕の環境 (Chrome) でオセロ本体が正常に動いたので、他のウェブブラウザでもテストする必要があります。しかし、私は Mac ユーザーなので InternetExplorer(以後、IE) でのテストはちょっとめんどくさいのです。なので、その時 Windows ユーザーだった hinata に IE でのテストを頼みました。(hinata が Mac ユーザーになった今も頼んでいますが。) すると、

hinata「『遊ぶ』ボタンが反応しないです。あと、IE8 以前でレイアウトがおかしいです。それ以外は大丈夫です。」
KA-FU-「ファッ」

[1] キーボードで入力し、文字で出力するユーザインタフェース
[2] キーボードだけでなくマウスなどでも入力でき、また出力も文字だけでなくウィンドウや画像など様々なものを出力するユーザインタフェース

ひーつらいですねぇ。どうにかしなければ。

とある一般人「え、見出しまでつけておいて問題点それだけですか。」

いやー、それだけだったんですよねこれが。まぁ、問題点ですから少ないほうがいいんですよ、はい。で、困ったものです、私には IE のことなんてさっぱりですし一体どうしたものか。(ググればどうにかなるでしょというツッコミは受け付けておりません)

3　決戦、InternetExplorer

IE での問題点が発覚した後しばらくして、

KA-FU-「hinata、IE の件どうなった？」
hinata「『遊ぶ』ボタンの方は type 属性が button の input 要素ではなく、type 属性が button の button 要素を使うことで解決しました。」
KA-FU-「... はい？」
hinata「レイアウトの方は IE8 以前を対応しないことにして解決しました。」
KA-FU-「お、おう。ま、まったく、Web マンは最高だぜ!!(動揺)」

__人人人人人人__
＞　突然の解決　＜
￣ Y^Y^Y^Y^Y ￣

決戦なんてなかった。まぁ、hinata 曰く、根本的な解決にはなってないそうなので実際決戦はなかったらしい。

4　過去の自分が造りしもの

オセロ本体が完成してからしばらくたって、私は hinata に言いました。

ソースコードを読む前に言っておくッ！おれは今ソースコードをほんのちょっぴりだが読んだ
い、いや... 読んだというよりはまったく理解を超えていたのだが.....
あ、ありのまま今起こった事を話すぜ！
「おれはソースコードを読んでいたと思ったら全くもってソースコードを理解できなかった」

な、何を言っているのかわからねーと思うが
おれも何が起こったのかわからなかった...
頭がどうにかなりそうだった... 頭が悪いだとか寝ぼけていただとか
そんなチャチなもんじゃあ断じてねえ
もっと恐ろしいものの片鱗を味わったぜ...

そう、オセロ本体を少し変えようと思って自分が書いたソースコードを読んでみたら、それが何をしてるのか全然わからないのです。これは困った。読めないソースコードを直すなんてほぼ無理です。

とある一般人「かなりやばそうじゃないですか。いったいどうやって解決したんですか。」

~~結局諦めました~~ わ、私の実力でどうにかしましたよ。
ということで、ここでは『リーダブルコード』[1] を参考に、読みやすいコードについて少しばかり書かせていただきます。

読みやすいコードとは

まず、ここでの読みやすいコードは一体何を指すのかを決めておく必要があります。ここでは、読みやすいコードとは、可読性が高いコードという意味だけでなく、そのソースコードについて何も知らない人が読んでも比較的短い時間で理解できるコードを指します。

とあるプログラマ「なんで他人に分かる必要があるんだよ、このコード見るの俺だけだぜ？」

と思う人もいるかもしれません。しかし、「そのソースコードについて何も知らない人」というのは他人だけでなく、そのソースコードの中身を忘れてしまった未来の自分も含んでいます。そう考えると、後のほうでそのソースコードが原因のバグが見つかったりしたときにも、やはり読みやすいコードである方が良いのです。

名前

変数や関数の名前についてです。ではまず、私が読めなかった (過去の自分の) ソースコードコードで使われている変数や関数の名前を少し見てみましょう。

実際に使われている名前	その変数や関数の中身
pname	プレイヤーの名前
game	決着がついたかどうか

place	AI が駒を置く場所
black	黒の駒を描画して保存する
white	白の駒を描画して保存する

我ながらひどい...。これらの名前の悪い点には次のようなものがあります。

- 名前を見ても中身が何かわからない
- 変に省略している
- 区切られていないので読みにくい

では、“pname” について、悪い点上から変えていきましょう。

1. まず、名前を見たら中身が何かわかるようにします。中身は「プレイヤーの名前」なので、“playername” などが良いと思います。(正しい英語の文法にする必要はありません。)
2. 次に、「変に省略している」ですが、これは 1 で直っているのでよしとします。
3. 最後に、区切ります。この変数が使われているのは JS のソースコードの中なので、“playerName” と区切る場所の直後の文字を大文字にするのがいいかと思います

このように、“pname” という意味不明な名前から、“playerName” というわかり易い名前になりました。(わかりやす差には個人差があります。) また、3 では言語によってよく使われる区切り方が違います。JS では区切る場所の直後を大文字に、Lisp なら区切る場所に “-”、Python [3] なら区切る場所に “_”、という感じだと思います。

では、次に「ホームディレクトリ配下のファイルとサブディレクトリのリスト」が入った変数に名前をつけてみます。そのまま名前に情報を詰め込むと、“list-of-files-and-subdirectories-within-home-directory” のようになるかと思います。(先ほど言ったように、正しい英語の文法にする必要はありません。) これでいいという人もいるかもしれませんが、私は長すぎると思います。このように、ただ情報を詰め込むだけでもダメで、その変数や関数の説明を正確に、そして短く書く必要があります。

では、次のようなコードがあったとします。

```
(defparameter right 1)
(defparameter left 2)
(defparameter tmp right)

(setf right left)
```

[3] 広く使用されているプログラミング言語で、さまざまなものをプログラムできる、非常に便利な言語。**しかし Lisp には劣る。**

```
(setf left tmp)
```

“right” と “left” の中身を入れ替えています。“right” と “left” に入ってる値が何なのかがわかりませんが、今見るべきはそこではありません。“tmp” です。“tmp” というのは “temporary” の略で、“temporary” というのは「一時的」という意味です。では、この “tmp” という名前の変数にはどういう意味があるのでしょう。実は、正解は「何の意味もない」です。“temporary” の意味のとおり、ただ一時的に値を保存しておくための変数で、それ自体に何の意味もありません。こういう時に、“tmp” のような名前をつかいます。変数の一生が短くても、本当に何の意味もない場合しか “tmp” は使ってはいけません。

他にもいろいろ気をつけれる点はあるのですが、~~めんどくさいですし、~~ここまでにしておきます。名前を考えるのも簡単じゃないのです。

コメント

ソースコードに書くコメントについてです。コメントは読み手にコードを読んだだけではわからない書き手の考えを伝えるためのものです。だからもちろん、コメントに書いてはいけないのは、コードを読めばすぐわかることです。例えば

```
;; hoge は要素数 10 の配列
(defparameter hoge (make-array 10))
```

こんなコードがあったとしましょう。CommonLisp が少しでもわかる人だったらきっと全員コメントに対して

とある CommonLisp が少しでもわかる人「んなこたわかっとるわ！！！」

と叫びたくなるのではないでしょうか。(叫びたくなり度には個人差があります。) このコメントはコードを見ればすぐにわかるので全くの無意味です。こういうコメントはしてはいけません。また、名前で補える情報はコメントにせずに名前で補うべきです。では、どのようなコメントをすればいいのかを、具体例を 2 つ挙げて説明します。

- ある機能を実装しようとしていたけど、その日のうちに出来上がらず、途中で一度寝ることにしたとします。もしその人が忘れっぽい人なら次の日の朝には何を実装しようとしていたのかわすれているかもしれません。こういうときは、次にやろうとしていることをコメントに残しておくのが良いです。また、なにかバグを発見したが今すぐにデバッグできないようなときにも、コメントでどういうバグが起きているのかを残しておくことは有効です。

- あるコードで主人公のお小遣いの値を入れる定数を定義したとします。名前からこれにお小遣いの値が入っていたことが分かったとしても、なぜこの値なのかはわかりません。そういう時にはコメントを使いましょう。例えば、「これくらいが年相応です」などなど。

他にも、いろいろなところで、コメントは使えます。でも、どの場合においても、読み手の事を考えて書くことが重要になってきます。なので、コメントも名前と同じで簡潔で、明確、そして正確である必要があります。コメントを書くときにもちゃんと考える必要があるのです。

他にも、読みやすいコードにするために工夫できるところはたくさんあるのですが、私はまだ未熟者なので全てを説明することはできません。なので、まずはこの 2 つに気をつけてみるといいと思います。たぶん、この 2 つに気をつけるだけでも十分読みやすいコードになると思います。他の工夫の仕方を知りたい人は、『リーダブルコード』を読んでください。

5　jQuery、来訪

その後、AI をサーバーで動かすにあたって Ajax という技術を導入しました。データを取得するのにフォームから送信してページを丸ごと読み込むのではなく、必要なデータだけバックグラウンドで読み込み、取得したデータを利用して処理をする、というのが主な流れです。しかし、この Ajax を自前で実装するのは結構面倒です。例えば hoge.php に user が hoge、text がぴょんぴょんとなるデータを送信する例です。

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'hoge.php', true);
xhr.onreadystatechange = function() {
    if (this.readyState !== 4) {
        return;
    }
    console.log('complete!');
    if (this.status === 200) {
        console.log('success');
        console.log(this.responseText);
    } else {
        console.log('error');
    }
};
```

```
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
xhr.send('user=hoge&text=%E3%81%B4%E3%82%87%E3%82%93%E3%81%B4%E3%82%87%E3%82%93');
```

xhr.send のところがわかりにくい (これだけ見てもどんなデータを送ってるのかわかりづらい) のでこうしてもいいんじゃないですかね。(間違ってても責任は取りません。)

```
encodeForm = function(data) {
    var param;
    var paramArray = [];
    for (param in data) {
        part = encodeURIComponent(param) + '=' + encodeURIComponent(data[param]);
        paramArray.push(part);
    }
    return paramArray.join('&').replace(/%20/g, '+');
};

xhr.send(encodeForm({
    user: 'hoge',
    text: 'ぴょんぴょん'
}));
```

どっちにしろ長いですね、はい。とにかく記述が長くなることだけ分かってもらえるといいです。というか書いてることが理解できる人がこの部誌を読むと間違いをたくさん指摘されそう。

ここで jQuery の出番です。jQuery を使っただけで

```
$.post('hoge.php', {
    user: 'hoge',
    text: 'ぴょんぴょん'
}).always(function() {
    console.log('complete');
}).done(function(data) {
    console.log('success');
    console.log(data);
}).fail(function(error) {
    console.log('error');
```

```
});
```

こうなります。(console.log というのはデバッグの時に使う関数で、コンソールに何かを出力します。) 短くなったのはもちろん、always や done、fail といったキーワードが出てきたので「hoge.php に post でデータを渡し、終わったらいつも (成功でも失敗でも)'complete' と出力し、成功したら'complete' と出力したあとに結果を出力する。失敗したら'error' と出力する」という流れが分かりやすくなっています。流れが分かりやすくなるという点は次の章でもう一度触れます (めんどくさくなって部誌を書くのをやめたりしない限り)

jQuery は HTML の要素を操作するのにも長けています。画像や表示されている文字が変わったり、一部が非表示になったりするページはよく見かけますが、そのような処理は jQuery を使うことで簡潔に分かりやすく書けます。

```
$('.hoge').removeClass('hoge');
```

これは hoge というクラスが付いた要素のクラスから hoge を取り除くコードです。生の JS で書くととても長くなります。長くなりすぎるし、書いたコードが正しいか自分でも分からなくなるので ~~というよりめんどくさいので~~ 例は省略します。

~~面倒になってきたし、~~ コードとかでちょっと長くなったのでこれくらいにします。

6　瞬間、処理、重ねて

非同期プログラミング

JS においてはネットワークアクセスなどが非同期で動作します。さっきの Ajax もそうでした。非同期の利点は

- 重い処理を止まって待つのではなくほかの作業をしながら待てる
- いくつかの重い処理を並列にできる

などです。(多分)Ajax はユーザーの操作を妨げずにバックグラウンドで通信できるためユーザーが待たされない点が好ましいとされています。サイトの一部を先に読み込むと読み込みが速くなったように感じるのではないでしょうか。

一方、非同期プログラミングは複雑になりがちです。結果が返ってくるのがいつか分からず、複数の非同期プロセスを同時に進行させるとどのプロセスが先に終了するかわからない。さらに、エラーをキャッチするのも難しい。また、後で見ていきますが、非同期プロセスを順番に実行しようとすると関数のネストが深くなる傾向になります。

例えば、setTimeout は非同期に動作するタイマー関数です。setTimeout(func, time) の形で time(単位ミリ秒) 後に func を実行します。ここで大事なのは、タイマーをセットするだ

けで待ちはしないことです。

```
setTimeout(function() {
    console.log('hoge');
}, 100);
console.log('huga');
```

このとき、hoge と fuga はどちらが先に出力されるでしょうか？ setTimeout はタイマーをセットするだけで、そのまま次に進んでしまうので fuga が先に出力され、100 ミリ秒後に hoge と出力されます。

KA-FU-「えっ sleep みたいにその場で待ってくれないんすか。えっえっ意味分かんない。」

というように生粋の JS マンである (Ruby や PHP や C++ もちょっとだけ書けますが) 僕には理解できない言葉を KA-FU-は言いました。setTimeout は非同期なのがあたりまえです、はい。しかし、1 秒後に hoge と出力し、その 1 秒後に fuga と、その 1 秒後に piyo と出力するプログラムを書いてみると

```
setTimeout(function() {
    console.log('hoge');
    setTimeout(function() {
        console.log('fuga');
        setTimeout(function() {
            console.log('piyo');
        }, 1000);
    }, 1000);
}, 1000);
```

一気にネストが増えて見づらくなりました。なお、KA-FU-はネストが増えても別に気にしない人なのでこのコードをみても

KA-FU-「えっ別になんとも思わないしまだ見やすい方なのでは。」

というように生粋の JS マンである僕には (ry)。これを世の人々は「コールバック地獄」(callback hell) と呼んでいます。(コールバックについては後ほど)Ruby の end 地獄と同じくらい読みにくい状態だと思います。これを解消するためにいろいろな偉い人達が頑張っ

てきました。

コールバック

onload や addEventListener などのメソッドででコールバックを設定するタイプと、関数の第二引数で関数を指定するタイプの二種類に別れます。XMLHttpRequest は前者です。

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'url');
xhr.onload = function() {
    if (this.status !== 200) {
        console.log(this.statusText);
    } else {
        var result = this.responseText;
        //result を使っていろいろする
    }
};
xhr.onerror = function() {
    console.log('error');
};
xhr.send();
```

setTimeout は後者です。後者は Node.js の標準ライブラリでよくある形ですね。

```
var fs = require('fs');
fs.readFile('path/to/file', { encoding: 'utf-8' }, function(error, data) {
    if (error) {
        console.log(error);
    } else {
        console.log(data);
    }
});
```

ファイルを読んでエラーが起きなければファイルの内容を、エラーが起きればエラーの内容を表示するだけのサンプルです。fs.readFile はオプションがある場合は第三引数、オプションがない場合は第二引数に関数を取ります。動作が完了した後に呼ばれる関数をコールバックと呼んでいます。どちらにしろ、さっきのコールバック地獄は避けられません。

onload や addEventListener などの引数である関数の中で二回目の非同期プロセスを呼ぶ必要があります。また、複数の非同期プロセスを並列に実行し、全てが終わったら別のことをする、なんて複雑なことをしたらプログラマが死にます。

例えば、ディレクトリのファイルを調べ、それらのファイルを 1 つずつ読んですべて終わったら結果を配列にして出力する、という例を考えてみます。(Node.js 前提です) 正直、こんな処理書きたくないです。fs(File System) というライブラリに readdir という関数がありますが、この関数はディレクトリも含めて返してきます。そこで 1 つずつファイルかどうか調べ (fs.stat という関数で調べられます)、ファイルのものだけ読む (fs.readFile)。全部読み終わったら結果を配列にして出力する。これらをすべてコールバックでやるなんて無茶です。正直やりたくないです。そこで、悩める子羊のために新しい JS(正確に言うと標準の ES6) で Promise というものが正式に導入されました (導入されます)。

Promise

Promise に関する説明は他の本やサイトにゆずります。(そこまで詳しく理解してるわけではないですし) ここでは XMLHttpRequest を Promise を返す関数にしたいと思います。

```
function get(url) {
    return new Promise(function(resolve, reject) {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', url);
        xhr.onload = function() {
            if (this.status !== 200) {
                reject();
            } else {
                resolve(this.responseText);
            }
        };
        xhr.onerror = function() {
            reject();
        };
    });
}
```

使用法

```
get('/hoge/fuga').then(function(data) {
    console.log(data);
}).catch(function() {
    console.log('error');
});
```

とにかく、

```
hoge().then(/* 成功した場合*/).catch(/* 失敗した場合 */);
```

という形です、はい。(then の引数に成功した場合、失敗した場合と 2 つ関数を渡すこともできます)resolve, reject の引数となった値は then や catch の引数の関数に渡されます。resolve(this.responseText) としているので then の引数の関数で data として参照できます。resolve した値が次の関数に渡されています。

ところで、前出の jQuery を使った Ajax のコードを引っ張ってきます。

```
$.post('hoge.php', {
    user: 'hoge',
    text: 'ぴょんぴょん'
}).always(function() {
    console.log('complete');
}).done(function(data) {
    console.log('success');
    console.log(data);
}).fail(function(error) {
    console.log('error');
});
```

似てますね。jQuery でも Promise に似た仕組みがあり、Deferred と呼ばれています。

例:setTimeout を Promise や Deferred を使ってわかりやすくする。

```
function waitByPromise(time) {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            resolve();
        }, time);
```

```
    });
}

function waitByDeferred(time) {
    var deferred = jQuery.Deferred();
    setTimeout(function() {
        deferred.resolve();
    }, time);
    return deferred.promise();
}
```

Deferred については説明が面倒なので省略します。

また、並列実行が簡単になる Promise.all という関数があります。Promise の配列を引数に取り、すべての Promise が resolve した時に resolve する Promise を返します。ちなみに一つでも reject されると reject し、resolve する値はそれぞれの Promise が resolve した値の配列となります。

```
Promise.all([waitByPromise(2000), waitByPromise(7000)]).then(function() {
    console.log('ok');
});
```

この場合、2 秒後に resolve される Promise と 7 秒後に resolve される Promise の配列が Promise.all に渡されるので、7 秒後、全てが resolve された後に ok と出力されます。また、then の関数の中で値を返したときは次の then の引数にその値が渡されますが、その返り値が Promise オブジェクトだった場合はその Promise オブジェクトが resolve するまで待たされます

さて、Promise を使って

> 例えば、ディレクトリのファイルを調べ、それらのファイルを 1 つずつ読んですべて終わったら結果を配列にして出力する

という例を考えてみます。天下の Promise さんならできるはずです。Node の fs には同期の関数もありますがあえて使いません。(非同期縛りです) まず、readdir や readfile を Promise を返す関数にします。また、ファイルかどうか判定する関数も追加します。

```
function readdir(path) {
    return new Promise(function(resolve, reject) {
```

```
            fs.readdir(path, function(error, files) {
                if (error) {
                    reject(error);
                } else {
                    resolve(files);
                }
            });
        });
    }

    function readFile(path, options) {
        return new Promise(function(resolve, reject) {
            fs.readFile(path, options, function(error, data) {
                if (error) {
                    reject(error);
                } else {
                    resolve(data);
                }
            });
        });
    }

    function isFile(path) {
        return new Promise(function(resolve, reject) {
            fs.stat(path, function(error, stats) {
                if (error) {
                    reject(error);
                } else {
                    resolve(stats.isFile());
                }
            });
        });
    }
```

これらは単純に失敗したらエラーを reject、成功したら結果を resolve しています。これらを組み合わせて実装します。

次に、ファイルやディレクトリのの絶対パスの配列を渡すとファイルのパスだけ集めてくれる関数を追加します。

```
function collectFilePath (filepaths) {
    return Promise.all(filepaths.map(isFile)).then(function(result) {
        var ret = [];
        result.forEach(function(fileResult, index) {
            if (fileResult) {
                ret.push(filepaths[index]);
            }
        });
        return ret;
    });
}
```

これらを組み合わせて実装します。

```
readdir(root).then(function(files) {
    return files.map(function(filename) {
        return path.join(root, filename);
    });
}).then(function(filepaths) {
    return collectFilePath(filepaths);
}).then(function(filepaths) {
    return Promise.all(filepaths.map(function(filepath) {
        return readFile(filepath, {
            encoding: 'UTF-8'
        });
    }));
}).then(function(fileData) {
    console.log(fileData);
});
```

return がやけに多い黒魔術みたいなコードになりましたが、コールバックでやるよりはよほどマシでしょう。

(参考にさせていただいたサイト http://www.html5rocks.com/ja/tutorials/es6/promises/)

Generator パターン

ES6(新しい JS などの標準) で Generator というものが追加されました。途中で処理を止められるというのです。そこで、偉い人は考えました。

とある偉い人「非同期のコードを実行してる間は処理止めちゃえばいいじゃん」

実際はこんなに簡単では無いのですが基本的なアイデアはこんな感じです (たぶん)。yield というキーワードで処理を止めることができます。co というライブラリがいろいろ裏でやってくれるので楽できます。

co を使った例です。

```
function readFiles(files, options) {
    return Promise.all(filePaths.map(function(filepath) {
        return readFile(filepath, options);
    }));
}

co(function*() {
    var fileOrDirectory = yield readdir(root);
    var fileOrDirectoryPaths = files.map(function(filename) {
        return path.join(root, filename);
    });
    var filePaths = yield collectFilePath(fileOrDirectoryPaths);
    var fileData = yield readFiles(filePaths);
    console.log(fileData);
});
```

なんということでしょう、return が多用され黒魔術みたいだったコードが、非同期はそのままに同期処理の書きやすさを取り入れたコードになったではありませんか！

ところで、これがブラウザで使えるようになるのは何年後なのでしょうか。10 年後には古いブラウザを気にせず使えるといいですね。でも Microsoft の新しいブラウザが Generator を導入するのが 5 年後くらいかな？でも最近 Microsoft 頑張ってるしもっと速く導入してくれると嬉しいです。むしろ他のブラウザのほうが Microsoft の新しいブラウザ

よりも導入が遅いかもしれません。

ここまで書いた感想

これ、なんの部誌だっけ...？たしかオセロの部誌だった気がするなぁ (遠い目)。半分くらいが Node.js での話になってるんじゃないですかね、どうしてこうなった。まあ、Promise の話はブラウザでもそれ以外でも同じですし、も、問題無いですね (震え)。なんでこんなに非同期について長々と書いてんだよ jQuery の時よりもはるかに長いよ。しかもこないだ勉強したばっかのことだし正確であることを保証できないし信頼性ゼロだよ。

というわけで (なにが"というわけで"だよ全くまとめてねえよ) みなさんはもっと真っ当なサイトや本を読んで勉強しましょう。参考程度に留めていただけると幸いです。

とある一般人「なぜか『jQuery、来訪』と『瞬間、処理、重ねて』だけ気合入った文章じゃないですか？」

そこに気づくとは... やはり NPCA の部誌を読む人は天才か。あ、えっと、それはですね、書いてる人が違うからです。(まぁ編集したのは私 (KA-FU-) ですが。) さて、書いてるのは誰でしょう？というか待って、よく見たらほんとにすごい内容じゃないですか。これは私の立場が危うい...。
(ヒント：この部誌のタイトルの下には 2 つ名前が書いてあります。)

7　SQL インジェクション、そして

オセロはちゃんと遊べるようになりました。やったぜ。しかし、hinata が言うにはセキュリティ的にダメなところがあるらしい。えっまじで、セキュリティとか考えたことなかったよ、Web 怖っ。(※セキュリティ問題は気にするのが普通です。)

hinata「SQL インジェクションによる攻撃に対策しないといけません。」
KA-FU-「は、はい...？」
hinata「prepare という関数を使えば対策できます。」
KA-FU-「へー。って、え、ごめん話の流れつかめてない。」
hinata「やるだけなので頑張ってください。」
KA-FU-「はい！？」

IE 対応はやってくれたのに...。はっこれが時に優しく時に厳しくというやつですねわかります。まぁ、調べたらわかりましたよ、はい。データベースを操作するときには、

```
select * from hoge.huga where foo = 1
```

のような SQL 文というものを実行します。しかし、素直にこれを毎回そのまま書くとプログラミング言語の性質を利用されたりなんやかんやで色々と怖いそうです。なので、prepare という関数などを使って

```
select * from hoge.huga where foo = ?
```

としておいて、実行するときに?の部分にさっきの例でいえば 1 などにして実行するという方法をとればいいそうです。あー Web 怖っ。では、次に行きましょうってぐわっ hinata 何をする、やめろ、くぁ w せ drftgy ふじこ lp

KA-FU-がログアウトしました。

ちょっと KA-FU-の説明が適当すぎるので僕 (hinata) が少し補足します。

```
select * from users where id ='$id' and password = '$password'
```

このような文字列を直接実行していたとします。このとき、$id = “”, $password = “‘ or ‘a’=‘a” だとすると、文字列は

```
select * from users where id ='' and password = '' or 'a'='a'
```

となります。‘a’ = ‘a’ が常に真なのですべてのデータが合致し、合致するデータがあるためパスワード認証をスルーすることができます。また、複文を利用した SQL インジェクションなど、いろいろな攻撃方法があります。(徳丸本 [2] を参考) 専門の本を読んだほうがわかりやすいと思うのでそちらを参照してください。

なお、他の脆弱性に関して、今回は信頼出来ないデータを元に表示するような部分がなかったため XSS は理論上発生し得ないはず... です。

8　せめて、ログらしく

KA-FU-がログインしました。

はぁ、はぁ、hinata 怖かった...。SQL インジェクションをどうにかしたのですが、実はもうひとつ対策しなきゃいけない大きなセキュリティ問題があります。いやぁ...Web、恐ろしい...。オセロ本体は JS で書かれています、最初のほうで説明したとおりこれはブラウザ上、つまりオセロで遊ぶ人のパソコンやスマホで動くので、JS の部分は自由にいじれちゃ

うわけです。すると、JS がわかる人ならいわゆるチート [4] ができてしまうのです。きゃー怖い。しかも今のままではチートを使われてもそのままデータベースに保存しちゃいます。これだと AI がバグを起こしてエラーが大量発生！？なんてことにもなりかねません。なので、PHP でゲームのログ [5] が普通にプレイされたものかチェックすることにしました。データベースに保存するために JS から PHP にログが送られます。その内容は、どちらかがコマを置いた場所とその後の盤面の情報です。それを元にして PHP 側でログチェックをします。例えば、あるときコマを置いた場所を A、その後の盤面を B、その後コマを置いた場所を C、その後の盤面を D とします。このとき、ログチェックは B の C にコマを置いた時、D と盤面がおなじになるか、という方法で行っています。もちろんオセロのルールで置けないはずの場所においていてもダメです。これで問題点は全部クリアしました。やったぜ、成し遂げたぜ。

[4] いわゆる反則技のこと
[5] 記録のこと

第 IV 章

まとめ

1　世界の中心で Lisp を叫んだもの

まえがきにも書きましたが、Web-othello 作成中のエピソードに関連させてあれこれ書いただけの部誌で、一貫したテーマがなくまとめるようなことは特にありません。なので、私が Web-othello の開発を通して感じたことをここでは書かせていただきます。

第 III 章の 1 にも書いたように、簡単な気持ちでオセロの Web 化を決めました。しかし、実際には Web というのはそんな簡単なものではなく、付け焼き刃の知識では見た感じいいものになるだけでちゃんとしたものは作れませんでした。ちゃっかり Web マンである hinata に Web 方面の仕事を任せたのが幸いして、無事完成しました。餅は餅屋といいますが、まさにそのとおりだと実感しました。

では、この部誌のまとめは私が Web-othello 作成を通じて一番感じたことにさせていただきます。

まとめ
Lisp が一番使いやすい！

第 V 章

あとがき

最後まで読んでいただいた方、そして一緒にこの部誌を書いてくれた hinata、本当にありがとうございました。よく考えてみたら、私はネタ成分しか書いていないような気がします。hinata からは「ネタ部分がいないと部誌は成り立たない」と言ってもらえたのですが、さすがにちょっとという気が否めません。きっとこの部誌は私の黒歴史の一部となるでしょう。でももし、私の部誌をお楽しみいただけたのなら嬉しい限りです。おそらく来年も部誌を書かせていただくことになると思いますので、もしよければまた部誌を手にとっていただけると嬉しいです。そして最後に、まとめに心を打たれた方、「は？何いってんのこいつ」と思った方、その他の方はぜひ Lisp を勉強しましょう。

では、ありがとうございました。

参考文献

[1] Dustin Boswell・Trevor Foucher リーダブルコードーより良いコードを書くためのシンプルで実践的なテクニック 須藤 功平 解説、角 征典 翻訳、オライリージャパン

[2] 徳丸 浩 体系的に学ぶ 安全な Web アプリケーションの作り方ー脆弱性が生まれる原理と対策の実践 ソフトバンククリエイティブ