東京工業大学情報理工学院総合型選抜　　活動実績報告書

氏名（よみ): **大岡山 花子（おおおかやま はなこ）**
高等学校: **〇〇県立△△高等学校**
(2021年3月　☐卒業・☒卒業予定)

活動実績概要（150 字程度):

JavaScript ではコールバックを多用することがある．これはプログラム全体の見通しを悪くするため，コールバック地獄と呼ばれる．そこで JavaScript でコールバック地獄が起こる原因と既存の解決方法について調査した．主な原因は JavaScript ではプリエンプションが一切ないことにあると分かった．既存の解決策としては，Promise などがあるが，これらも十分ではないと分かった．

活動実績の実施状況:

☒志願者が単独で行った
☐教師などからの指導を受けながら志願者が単独で行った
☐共同で行った
☐その他

報告書本体ページ数(表紙を含まない): 4 ページ

注意:

- 報告書本体を 4 ページ以内で作成し、この表紙と一緒に提出すること。
- 報告書本体の形式は自由とするが、文字の大きさは 10 ポイント以上にすること。また内容として活動実績の背景、具体的な内容、活動実績の実施状況の説明、参考にした資料の一覧などを必ず含むこと。
- 報告書本体に、活動実績を志願者が単独で行ったか否か、共同で行った場合は自身の役割、指導を受けた場合はどの部分に対する指導か等の説明を書くこと。
- 報告書本体には氏名、学校名はどうしても必要な場合を除いて書かないこと。

JavaScript 言語のコールバック地獄の調査

概 要

- **背景**：JavaScript 言語にはコールバック地獄という問題がある．
- **動機**：コールバック地獄は既存回避手法で十分なのか興味を持った．
- **やったこと**：簡単な例題でコールバック地獄を自己体験した．また，コールバック地獄のサンプルコードを 15 個作成し，Promise などの既存手法に適用してみた．
- **アピールポイント**：**for** ループで既存解決手法は不十分など，通常の JavaScript の教科書には載っていない知見を得た．

1 背景

JavaScript は Web プログラミングで重要な地位を占めており，Ajax を用いたサーバ通信，DOM ツリー操作，Greasemonkey や Tampermonkey を用いた Web ブラウザ拡張などで使われている．

文法や予約語は通常の手続き型言語と似ているため，比較的とっつきやすいが，細かいところでプログラマを悩ませている言語でもある．その JavaScript の問題の 1 つが**コールバック地獄**である．

JavaScript では非同期イベントの処理にコールバックを多用する．このコールバックの多用はプログラム全体の見通しを悪くし，この悪い状況がコールバック地獄と呼ばれている．

主な原因は JavaScript ではプリエンプションが一切なく，実行中の関数が終了するまで，他のコールバックに制御が移らないこと（Run-to-Completion と呼ばれる性質）にある．既存の解決策としては，Promise，Generator，Async/Await，RxJS などがある．しかし，これらの使用は必ずしも一般的ではないし，調べた範囲では JavaScript の教科書に分かりやすい記述は載っていない．

2 成果の具体的な内容

2.1 JavaScript の実行モデル

JavaScript ではプリエンプション（preemption，実行の一時中断とスレッドスイッチ）が決して起こらない．このため実行途中で実行をブロックすると（Web ページの更新などを含めて）プログラム全体の実行が止まってしまう．

例えば，C 言語では `sleep` 関数を使って，一時的に実行をブロックするコードを次のように簡単に書ける．

```
printf ("aaa\n");
sleep (10);  // 10秒間，実行を中断
printf ("bbb\n");
```

しかし，JavaScript では `setTimeout` 関数を使って，次のように書く必要がある．

```
console.log ("aaa");
setTimeout (function () {
    console.log ("bbb");
}, 10 * 1000);
// 第2引数にミリ秒単位でスリープ時間を指定
```

`setTimeout` 関数の第 1 引数は**コールバック関数**であり，10 秒後以降に実行される．このコードが示す通り，ブロック後の処理をコールバック関数として与える必要がある．`setTimeout` 関数の呼び出しの後にコードを書いても，そのコードはブロックの**前に**実行されるからである．コールバック関数は 10 秒後以降に実行されるが，`setTimeout` 関数自体はブロックしないことに注意して欲しい．

ここで次のように最後に無限ループを加えると，10 秒経っても `bbb` が表示されなくなる．コールバック関数は呼び出されなくなるのだ．

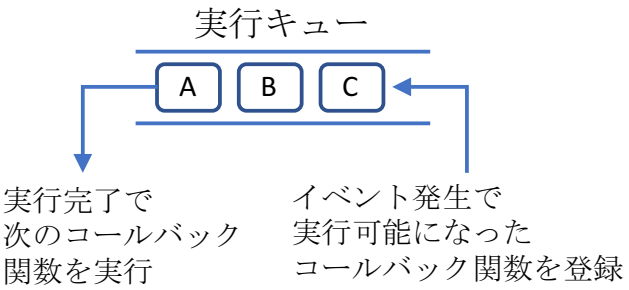

図 1: JavaScript の実行キュー

```
setTimeout (function () {
    console.log ("bbb");
}, 10 * 1000);
while (true); // 無限ループ
```

これは JavaScript の **Run-to-Completion** という性質のために起こる（図 1 も参照）.

- JavaScript のプログラム（関数）は途中でプリエンプションせず，最後まで実行を完了させる（Run-to-Completion）.
- 「10 秒経過」などのイベントが発生すると，そのイベントに関連付けられたコールバック関数は（すぐには実行されず）実行キューに入れられる.
- 実行中のプログラムは常に 1 つだけ．1 つの実行が完了すると，実行キューの先頭から新たに実行可能なコールバック関数を 1 つ取り出して実行する.

この実行キューの動作は実は JavaScript の言語仕様書 [1] では規定されていないが，異なる Web ブラウザや言語処理系で事実上の標準となっている [2].

「プリエンプションが無い」ということは，各コールバック関数（図 1 中の A，B，C）の実行はアトミックであることを意味する．これにより，（イベントの発生順序に依存して実行順序が前後する非決定性はありうるが）マルチスレッドプログラミングに付随する**データ競合**の問題は決して起こらない．このため，複数のコールバック関数が同じデータに安全にアクセスすることができる.（通常のプログラミング言語であれば，慎重に排他制御を行う必要がある.）

その代わり，通常のプログラミングでブロックが必要となる場所では，JavaScript では必ず「それ以降の処理」をコールバック関数として登録しなければならない.

コールバック関数はコールバック関数を連鎖的に呼び出す（もしそうしなければプログラム全体の実行が終了してしまう）ため，関数が論理的な意味の塊でなくなったり，プログラム全体の実行順序の見通しが悪くなる．さらに，コールバック関数は `try-catch` 構文での例外処理ができないため，エラー処理のためのコールバック関数も必要となる．また，コールバック中では実行文脈が異なるため，疑似変数 `this` の意味が異なる．これらの相乗効果で，プログラムの理解性や保守性は著しく悪化する．この状態を**コールバック地獄**という.

2.2 コールバック地獄の例

ここでは次の例題でコールバック地獄を考える.

- **例題**：重い処理（例：サーバへの Ajax リクエスト）を 100 個実行したいが，サーバに負荷をかけないため，1 度に 1 つずつ実行し，1 つの重い処理が完了してから，次の重い処理を実行したい．どうすればいいか.

ブロックが可能な通常のプログラミング言語では `for` などのループ構文で簡単にこの例題を解くことができる．しかし，JavaScript では `for` 文では記述が難しい．`for` ループを完全に抜けないと，Ajax リクエストを処理するコールバック関数が実行されないからである．このため，JavaScript では次のように再帰関数 `do_all_task` を使って解決せざるをえない.

```
function slow_task (name, callback) {
    console.log ("name = " + name);
    setTimeout (callback, 10 * 1000);
}
function do_all_task (array) {
    if (array.length == 0) return;
    var name = array.shift();
    slow_task (name, function () {
        do_all_task (array);
    });
}
do_all_task (["a", "b", "c", "d"]);
```

このプログラムでは重い処理（slow_task）を 1 つずつ処理する．slow_task には処理が終わった後で実行するコールバックを渡せるので，ここに残りの処理を行う do_all_task 関数を渡している．

このコードを拡張し，自分でキューを管理し，例えば同時に 5 個まで重い処理を投げられるコード queue-test.js も実装した [6]．また，これを用いて実際に Ajax リクエストを同時に 5 個まで投げる JavaScript コードを実装した（未公開，約 750 行）が，queue-test.js 中の関数が全体に埋まってしまい,「同時に 5 個までリクエストする」という動作が読み取りにくくなった．コールバック地獄を再確認する結果となった．

2.3 既存のコールバック地獄の解決手法

既存のコールバック地獄の解決手法には例えば以下が存在する．

- Future パターン方式：Promise
- 疑似協調マルチタスキング方式：Generator，Async/Await
- 関数型リアクティブプログラミング方式：RxJS

それぞれの有効性を体感するため，コールバック地獄のサンプルコードを 15 個準備し，上記の解決手法でどれだけ改善できるかを試してみた．全体的には「ある程度は解決できるが，使い方を理解するのが難しいし，十分には解決できない場合がある」という結論を得た．以下では Promise を用いて，これをもう少し詳しく説明する．

Promise は，重い処理を内包するオブジェクト（Promise オブジェクト）を即座に返し，その重い処理が済んだら，暗黙的にコールバック関数を用いて，.then 以下に指定した処理を実行する．例えば，以下のように.then のメソッドチェインを書くと，a，b，c，d の順に重い処理が実行される．

```
function slow_task (name) {
    console.log ("name = " + name);
    return new Promise (
        function (resolve, reject) {
           setTimeout (resolve, 3 * 1000);
    });
}
slow_task ("a")
.then(function(){return slow_task ("b");})
.then(function(){return slow_task ("c");})
.then(function(){return slow_task ("d");});
```

Promise のコンストラクタに渡す関数の第 1 引数 resolve は成功時に実行するコールバック関数，第 2 引数 reject は失敗時に実行するコールバック関数（ここでは使用していない）である．

コールバック関数の深い入れ子が無くなったことや，実行順序がプログラムの記述順序と一致するので，コールバック地獄はある程度，解消されている．

ただし，これを for 文にする際には注意が必要である．素直に以下のように書き換えると，a, b, c, d ではなく，a, d, d, d と出力される．

```
function slow_task (name) {
    console.log ("name = " + name);
    return new Promise (
        function (resolve, reject) {
            setTimeout (resolve, 3 * 1000);
    });
}
var array = ["a", "b", "c", "d"];
var p = slow_task ("a");
array.shift ();
for (name of array) {
    p = p.then (function () {
        return slow_task (name);
    });
}
```

これは，コールバック関数の中で，name が参照されるのが for ループを終了した後になってしまうためである．これを解決するには，以下のように，クロージャを作成して，即座に評価する必要がある．これにより正しい name を渡すことが可能になる．

```
function slow_task (name) {
    console.log ("name = " + name);
    return new Promise (
        function (resolve, reject) {
            setTimeout (resolve, 3 * 1000);
    });
}
var array = ["a", "b", "c", "d"];
var p = slow_task ("a");
array.shift ();
for (var name of array) {
    (function () { // クロージャ
        var name2 = name;
        p = p.then (function () {
            return slow_task (name2);
        });
    })(); // をすぐに評価する
}
```

しかし，プログラムは理解しづらくなる．.then チェインはコールバックの評価順序と，記述順序を一致させる効果があるものの，実際の評価順序はあくまで Run-to-Completion に従うため，このような齟齬が生じてしまう．つまり，for ループに関しては，Promise では十分にはコールバック地獄を解決できない．他の解決方法も同様である．

3 単独の成果か否か

単独の成果である[1]．他の人（教員や友人）の力は借りていない．コールバック地獄とその解決方法については，書籍や Web ページの内容を参考にした [2] [3] [4] [5]．しかし，この文書の内容は参考にした書籍や Web ページそのままではなく，私なりに理解した内容で再構成してある．また，プログラム例も私のオリジナルである．

参考文献

[1] Standard ECMA-262 ECMAScript 2016 Language Specification, https://www.ecma-international.org/publications/standards/Ecma-262.htm

[2] Effective JavaScript，D. Herman 著，吉川邦夫 訳，ISBN-10: 4798131113，翔泳社，2013

[3] http://callbackhell.com/

[4] http://stackoverflow.com/questions/25098066/what-is-callback-hell-and-how-and-why-rx-solves-it

[5] http://postd.cc/node-js-callback-hell-promises-generators/

[6] キュー・テストコード，https://github.com/gondow/js-test/blob/master/queue-test.js

[1] 実際の内容は櫻井直之氏本学院 2016 年度卒業研究の内容に基づいています．