Vol.2018-SE-198 No.22
2018/3/9

実行順序に着目したasync/awaitの実行の可視化

冨永 江奈　荒堀 喜貴　権藤 克彦

概要：JavaScript ではネットワーク通信やノンブロッキング I/O などの非同期処理において，終了後の処理をコールバック関数によって定義する．しかし，連続した非同期処理をコールバック関数を用いて記述すると，複数のコールバック関数が入れ子になり，ネストが深く可読性や保守性の低いコードになる．このような状態をコールバック地獄と呼ぶ．コールバック地獄を解決するため，JavaScript には Promise や async/await といった非同期処理の記述を容易にする機構が実装されてきた．async/await を用いると，コールバック地獄に陥ったコードを，ネストが浅く簡潔なコードに書き換えることが可能である．一方，その動作の複雑さ故に，書き換え後のコードの実行順序を正しく理解するのは容易ではない．本研究では async/await の問題点が実行順序の理解の難しさにあることを明らかにし，トレーサを利用して実行の様子を可視化するツールを作成した．その結果，Promise や async/await の動作の表示によって複雑な実行順序の理解を促すサポート機構が実装可能であることを示した．

1. はじめに

近年，ネットワーク通信を行うインタラクティブな Web ページや Web アプリケーションが広く普及している．そして，これらの記述には Web ブラウザを中心に広く活用されているプログラミング言語である JavaScript がよく用いられる．この JavaScript の特徴の一つに，シングルスレッドであることがあげられる．そのため，ネットワーク通信のような時間のかかる処理を行う場合，完了を待つ間に他の処理を行うことができず，ユーザーがブラウザを操作することができなくなるといった問題が発生する．この問題に対処するため，このような時間のかかる処理は同期処理ではなく，コールバック関数を用いて完了後の処理を記述する非同期処理の形にて記述される．

ここで，次のような連続した非同期なファイル読み込み処理を，コールバック関数を用いて記述する場合を考える．

(1) ファイル a.txt の中身を読み込む
(2) そこからファイル名を取得して，そのファイルの中身を読み込む
(3) さらにそこからファイル名を取得して，そのファイルの中身を出力する

この時，次に読み込むファイルのファイル名は前のファイルの中身に依存しているため，前のファイル読み込みが完了しないと次のファイル読み込みを行うことができない．したがって，次のファイル読み込み処理は前のファイル読み込み処理のコールバック関数内に記述する必要がある．すると，コードは次のようになる．

ソースコード 1: コールバック地獄に陥ったコード例

```
1  readFile("a.txt", (error, response) => {
2      if(error) {
3          console.log(error);
4          return;
5      }
6      var filename = getFilename(response);
7      readFile(filename, (error, response) => {
8          if(error) {
9              console.log(error);
10             return;
11         }
12         var filename = getFilename(response);
13         readFile(filename, (error, response) => {
14             if(error) {
15                 console.log(error);
16                 return;
17             }
18             show(response);
19         });
20     });
21 });
```

ソースコード 1 を見ると，いくつものコールバック関数が入れ子になり，その結果ネストが深くなっていることが確認できる．このように，依存関係のある連続した非同期処理によってコードが読みにくくなった状態を「コールバック地獄」と呼ぶ．コールバック地獄に陥ったコードは可読性，再現性が低い等の多くの問題を抱えている．

JavaScript では，コールバック地獄を解決するために Promise や async/await といった機構が考案されてきた．

async/await を用いるとソースコード 1 は以下のように書き換えられる．

ソースコード 2: ソースコード 1 を書き換えたコード

```
1  (async () => {
2      var response, filename;
3      response = await readFilePromise("a.txt");
4      filename = getFilename(response);
5      response = await readFilePromise(filename);
6      filename = getFilename(response);
7      response = await readFilePromise(filename);
8      show(response);
9  })().catch(error => {
10     console.log(error);
11 })
```

ソースコード 2 では前述の非同期処理を同期処理に近い形で記述することで，コールバック地獄を解決している．

このように，async/await を用いると非同期処理を簡潔に記述できるが，その動作は複雑なため正しく理解するのは難しい．また，async/await は Promise の使用を基本とした機構であり，この Promise の動作の理解も同様に難しいものである．Promise や async/await の動作の理解は，これらを用いたコードの実行順序の理解に大きく影響する．それゆえ，これらを学習中の人が記述したコードはしばしば期待と異なる実行順序になりうる．また，非同期処理にかかる時間は非決定的であり，実行順序が動作環境によって異なる場合もあるため，これらの動作を理解していても実行順序を予想することは容易ではない．本研究では async/await の動作の複雑さに着目し，async/await を用いたコードの実行順序が難解であるという問題を解決することを目的とする．

本研究と同じく JavaScript の async/await にフォーカスした研究 [4] では，async/await に代わる非同期処理機構 Sync/CC が提案されている．Sync/CC は await キーワードの代わりに継続 [6] とアスペクト指向言語 [7] を用いて非同期な関数を指定する．そのため，各関数について逐一指定が必要であり一概に簡潔な記述になるとは言えない，現在の JavaScript エンジンでは継続が存在しないので実装できない等の問題があり，非同期処理の完全な解決策とは言えない．一方で本研究では，研究 [4] ではフォーカスしていない問題点である，async/await を用いたコードの実行順序の難解さに着目し，この問題を解決するツールを提案する．

一般に，async/await を用いたコードの正しい実行順序を調べる方法としては，随所にログを出力する関数を記述してその出力結果を見る方法があげられる．これは有効なデバッグ方法ではあるが，コードが複雑になればなるほどその手間は大きくなり，かつ出力されたログから実際の実行順序を辿るのが難しくなる．また，出力された実行順序を元に Promise や async/await の動作，およびなぜその実行順序になったのかを開発者自身が熟考する必要がある．この問題を解決するため，本研究ではコードの実行順序および Promise や async/await の動作に関するログを取得するトレーサを計装し，トレーサで取得したログを元にビジュアライザによって実行の様子を可視化するツールを作成した．

提案するツールの適用により，async/await を用いたコードのログを自動的に取得し，Promise や async/await の動作を明示しながら実行順序を表示することに成功した．このツールは実行順序を調べることに加えて，なぜその実行順序になったのかを Promise や async/await の動作から推測することも可能にする．したがって，async/await の実行順序が難解であるという問題を解決することができる．

本研究の貢献は async/await の問題点がその実行順序の理解の難しさにあることを明らかにしたこと，および実行順序の理解に際してのトレーサの有効性を発見し，トレーサを利用して実行順序の理解を補助するツールを作成することで，この問題を解決するサポート機構の実装可能性を示したことにある．

2. 研究背景

2.1 Promise

Promise は ECMAScript2015 から実装された，非同期処理を扱いやすくするための機構である．Promise ではメソッドチェーンを用いることで，複数の連続した非同期処理をネストが深くならずに記述することができる．

Promise は状態，処理の結果の値/エラー，処理の成功/失敗時に呼び出される関数を持つオブジェクトである．Promise オブジェクトの状態は以下の 3 つである．

- pending：非同期処理の結果が返ってきていない状態
- resolved：非同期処理が成功した状態
- rejected：非同期処理が失敗した状態

本論文では非同期処理が成功し resolved 状態に遷移することを Promise が resolve する，非同期処理が失敗し rejected 状態に遷移することを Promise が reject すると呼ぶこととする．

Promise オブジェクトが resolve/reject するとオブジェクトに結果の値/エラーが登録され，これを引数に渡して処理の成功/失敗時に呼び出される関数が実行される．この関数は then メソッド，catch メソッドによって Promise オブジェクトに登録されたものである．then メソッドは引数を 2 つまで指定することができ，第一引数が resolve 時に呼び出される関数，第二引数が reject 時に呼び出される関数となる．catch メソッドでは reject 時に呼び出される関数のみを登録できる．

このため，今までコールバック関数に記述していた非同期処理が完了した後の処理を，then/catch メソッドによっ

て登録することが可能となる．以下に Promise を用いて非同期なファイル読み込み処理を記述したコード例を示す．

ソースコード 3: Promise を用いた非同期処理の例

```
function readFilePromise (filename) {
    return new Promise((resolve, reject) => {
        readFile(filename, (error, result) => {
            if(err) reject(error);
            else resolve(result);
        });
    });
}
readFilePromise(filename).then(result => {
    // 成功時の処理
}, error => {
    // エラー処理
});
```

関数 readFilePromise は，関数 readFile の処理をラップした Promise オブジェクトを返す関数である．Promise オブジェクトは作成されると同時に引数の関数の処理を同期的に実行するため，ここでは関数 readFile の呼び出しが同期的に行われる．関数 readFile のコールバック関数内では処理の成功時に第一引数の関数 resolve を，失敗時に第二引数の関数 reject を呼び出している．これらは Promise オブジェクトの状態を遷移させ，引数の値を Promise オブジェクトに登録する関数である．また，then メソッドによって完了後の処理が登録されており，これらは Promise オブジェクトが resolve/reject すると呼び出される．

さらに，then/catch メソッドの返り値は Promise オブジェクトであるため，以下のように複数の連続した非同期処理をメソッドチェーンの形で記述することができる．

ソースコード 4: then/catch チェーンの例

```
p(1).then(ret => {
    // p(1)のresolve 後に実行
    return p(2);
}).then(ret => {
    // p(2)のresolve 後に実行
    return p(3);
}).then(ret => {
    // p(3)のresolve 後に実行
}).catch(err => {
    // 途中でエラーが発生すると実行
});
```

ソースコード 4 のように複数の then メソッドをチェーンさせると，ある then メソッドで登録された関数が返す Promise オブジェクトに対し，次の then メソッドによって完了後の処理を登録できる．また，途中でエラーが発生した場合，then メソッドで登録した resolve 時の処理を飛ばして catch メソッドで登録した reject 時の処理が実行されるため，最後の catch メソッドにてエラー処理をまとめて記述することができる．このように，Promise は連続した非同期処理をメソッドチェーンを用いて記述することによりコールバック地獄を解決できる．

しかし，すべてのケースが Promise により簡潔に記述できるわけではない．その例として，連続した非同期処理をループによって記述する場合を考える．

ソースコード 5: Promise を用いたループ処理の例

```
function delay (num) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log(num);
            resolve(num);
        }, 1000);
    });
}
var p = delay(0);
for(var i=1;i<5;i++){
    p = p.then(() => delay(i));
}
// 0 4 4 4 4 と表示
```

ループ内では変数 p に p.then() により返される Promise オブジェクトを代入することでメソッドチェーンを作成する．このメソッドチェーンでは，1 秒後に引数を表示して resolve する Promise オブジェクトを返す関数 delay を，引数の値を 1 つずつ増やしながら呼び出す処理を登録しているため，実行すると 0 から 4 までの数が順に 1 秒おきに表示されることが予想される．ところが，実際は 0 以外の数はすべて 4 と表示されてしまう．これは，ループを回している時点では then メソッドでコールバック関数を登録する処理しか行っておらず，実際にコールバック関数が実行されている時には変数 i の値が 4 になっているからである．

この実際の動作と見せかけ上の動作のギャップにより，Promise の then/catch チェーンを用いたコードでは予想と異なる実行結果になる場合が存在する．この問題は，次に説明する async/await によって解決することが可能である．

2.2 async/await

async/await は ECMAScript2017 より実装された機構で，実行の中断，再開が可能な関数を用いて Promise の完了を待機することにより，非同期処理を同期処理のように見せかけた形で記述できる．

async/await では async 関数と await キーワードを用いる．async 関数は async キーワードをつけた function 宣言により定義される．async 関数中では式に対し await キーワードを修飾することができ，その箇所で関数の中断，再開を行うことができる．この await キーワードで修飾される式は，原則として Promise オブジェクトであることが想定されている．

また，async 関数の返り値は Promise オブジェクトであ

る．この Promise オブジェクトは async 関数内で値が返されるとその値で resolve され，エラーが発生するとそのエラーで reject される．

以下に async/await を用いたコード例を示す．

ソースコード 6: async/await を用いたコード例

```
async function foo () {
    var res = await p(); // 関数p は Promise を返す
    console.log(res); // p()のresolve 後に実行
}
foo().catch(() => {
    console.log(e); // p()のreject 後に実行
});
```

async 関数は await キーワードで修飾された Promise オブジェクトが作成されると実行を中断し，この Promise オブジェクトが resolve/reject されると実行を再開する．この際，resolve された値は await 式に渡され，reject されたエラーは await 式の部分で throw される．そのため，async 関数中では await の部分でブロックを起こして Promsie の完了を待機するように見せかけた記述ができる．ただし，本当にブロックするわけではなく，実際は await 以降の実行を暗黙的にその Promise の.then に登録することで完了を待機することに注意が必要である．

ソースコード 5 に例示した Promise ではうまく動作しないループ処理も，async/await を用いると以下のように簡潔に記述することができる．

ソースコード 7: async/await を用いたループ処理の例

```
function delay (num) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log(num);
            resolve(num);
        }, 1000);
    });
}
(async function () {
    for(var i=0;i<5;i++){
        await delay(i);
    }
})();
// 0 1 2 3 4 と表示
```

async 関数内の処理に注目すると，await の部分で Promise の完了を待つことにより，同期処理と同様に for を用いた形で記述できることが確認される．そして，コードを実行した時の表示結果は想定したものと一致する．これは，async 関数内では実行の中断，再開を行うことができ，中断時の変数の値は，中断している時に変更されない限り保存されるためである．このように，async/await を用いると，Promise を用いた場合よりも柔軟なコードを記述することが可能となる．

2.3 async/await の問題点

async/await はコールバック地獄の有力な解決策であるが，async/await を用いたコードは実行順序を理解するのが難しいという問題点が存在する．async 関数内では Promise の完了をブロックして待機するように見せかけた記述が可能であるため，簡単なコードについては直感に即したわかりやすい実行順序になると考えられる．しかし，正しい動作を理解していないと，複雑な記述になった時に正しい実行順序がわからなくなるという問題がある．

1 つ目のわかりにくい点は，await 部分でのブロックは擬似的なものであり，実際は Promise の.then に await 以降の実行を登録することで完了を待機している点である．このことは，以下の例により確認できる．

ソースコード 8: await 部分のブロックの動作を確める例

```
function wait (time) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, time);
    });
}
async function asyncTask(){
    await wait(1000);
    console.log("complete"); // 後に表示
}
asyncTask();
console.log("block?"); // 先に表示
```

関数 asyncTask では 1 秒かかる処理を待って complete を表示する．この asyncTask の呼び出しの後に block?を表示するという処理を実行すると，complete の表示を待たずに block?が表示される．これより，関数 asyncTask は実際にブロックを起こして Promise の完了を待機するわけではなく，残りの処理を Promise の.then に登録して実行を中断しているとわかる．

2 つ目のわかりにくい点は，Promise や async 関数の処理は，呼び出し時に同期的に実行を開始する点である．このことを確認するため，以下のコードを考える．

ソースコード 9: 非同期処理の実行タイミングを確める例

```
function wait (num) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, num*1000);
    });
}
async function foo() {
    var task1 = wait(1);
    var task2 = wait(2);
    await task2;
    console.log("complete task2");
    await task1;
    console.log("complete task1");
}
foo();
```

task1 は完了までに 1 秒かかる処理，task2 は 2 秒かかる処理である．ソースコード 9 のように，変数に格納した Promise の完了を await により待機するとき，await 部分だけを見るとそれぞれの Promise の処理は直列に実行されているように見える．しかし，実際には Promise を作成した時点で処理が開始されるため，task2 の完了以前に task1 は完了しており，complete task1 の表示と本当の task1 の完了のタイミングは異なる．また，task1 の await 部分ではその処理を待機する必要がなくなるが，await 以降の実行はやはり task1 の.then に登録されることにも注意が必要である．そのため，task1 を await する部分で同期的な処理を行う場合とは異なり，complete task2 と complete task1 の表示の間に他の処理が実行される可能性がある．この非同期処理が実行されているタイミングがわかりにくいという問題は，map メソッドを用いる時など明示的に関数呼び出しを行わない場合に，より顕著になると考えられる．

このように，async/await を用いたコードは，これまでに説明した Promise や async/await の動作を理解していないと正しい実行順序を理解することができないという問題がある．しかし，Promise や async/await の動作は複雑なため，これらを正しく理解することは難しい．このことは，Promise や async/await の学習者に向けた Web サイトに正しい動作についての説明をせず，その使い方のみを説明しているものがしばしばあることからも想像がつく．

また，非同期処理にかかる時間は非決定的なため，複数の非同期処理を行う場合それらの実行順序も非決定的である．加えて，実行キューに登録されたコールバック関数が実行される順序は動作環境に依存するという問題も存在する．そのため，これらの動作を正しく理解していても，複雑なコードの実行順序を考察するのは容易ではない．

ここで，正しい実行順序を調べるときに考えられる簡単な方法として，随所にログを出力する関数を記述して，その出力順序から実際の実行順序を調べるというものがあげられる．けれども，この方法には大規模なコードになったときに非常に多くの手間がかかってしまうことや，デベロッパ自身が出力された実行順序をから Promise や async/await の動作を考察しなくてはならないといった問題点がある．

そこで，本研究ではトレーサを計装することでログの出力を自動的に行い，さらに出力されたログに基づいて Promise や async/await の動作を可視化することでこれらの動作の理解を補助するツールを作成した．

3. 提案手法

3.1 概要

提案するツールはログ取得用コード（トレーサ）の計装部分と，取得したログをもとに実行の過程を表示するビジュアライザ部分の 2 つから構成される．ログコード計装部分では入力されたコードにログを取得する関数の呼び出しを追加し，書き換えたコードを実行することでログを取得する．ビジュアライザ部分では，トレーサにより取得したログをもとに，実行順序を表示する方法と実行の各ステップの様子を表示する方法の 2 通りで可視化を行う．図 1 にこのツールの概要を示す．

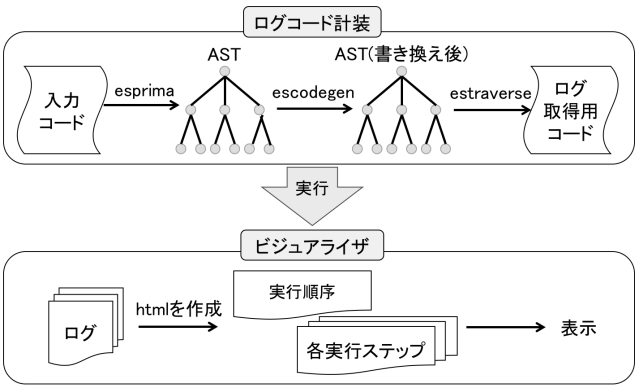

図 1: 提案するツールの概略図

このツールは可視化によって以下の 4 つを明示することを目的とする．

- 入力コードの実行箇所
- 実行箇所の呼び出し元
- 作成された Promise オブジェクトの状態
- async 関数が中断・再開されるタイミングおよびその位置

ログコードの計装やビジュアライザでの表示は，これらの要件を満たすように行う．

なお，このツールは Firefox 上で動作するものである．また，ログコード計装部分ではコードの変更を行うために Node.js の npm パッケージ esprima[12]，estraverse[13]，escodegen[14] を用いた．これらの npm パッケージを使用したコードを Firefox 上で動作させるため，Node.js 用モジュールのブラウザ上での使用を可能にするツール browserify[15] によりコードの書き換えを行なっている．

3.2 ログコード計装

図 1 に示すように，ログコード計装部分ではコードを AST に変換した後，この AST をログを取得するための関数呼び出しが追加されるよう書き換え，これを再びコードに変換することでログ取得用のコードを作成する．

取得するログには以下の 4 つの情報を付与した．

- location：書き換え前のコードにおける行・列ベースでの範囲
- range：書き換え前のコードにおける文字インデックスベースでの範囲
- type：取得したログの種類
- trace：スタックトレース

location，range の情報には esprima で書き換え前のコード

を解析した際に取得できる値を用いた．trace の情報はエラーを発生させ，そのスタックトレースを参照することにより取得した．type の情報はログを取得する状況に合わせて，引数において指定する．

ログを取得するための書き換えは次の箇所について行う．

文の実行
全ての文と文の間にログを取得する関数を挿入することで，各文が実行される直前のログを取得する．

Promise
全ての関数呼び出しを関数でラップして，関数の返り値が Promise オブジェクトであれば Promise の作成を示すログを取得し，resolve/reject 時にログを取得する処理を登録する．

await 式
全ての await 式を関数でラップして，async 関数が中断する直前と再開した直後のログを取得する．

関数呼び出し
関数呼び出しをラップする関数の引数において，関数が呼び出される直前にそのスタックトレースを取得する．

以上より，前述の要件を満たすログを取得することが可能となる．

3.3 ビジュアライザ

図 1 に示すように，ビジュアライザ部分では取得したログを元に，実行の様子を表示するための html を作成する．表示は実行の各ステップの様子を示す方法と，実行順序を示す方法の 2 通りで行っている．どちらの方法においても，出力する html の作成はログの type プロパティから判断したログの種類に応じて，前述の要件を満たすように行う．

実行の各ステップの様子を示す方法では入力コードの一部をハイライトしたり，動作箇所にメッセージを表示することで各ステップの様子を視覚化する．コードのハイライトやメッセージの表示は，入力されたコードに html の span タグによるクラス定義やメッセージを挿入することで行う．なお，実行の各ステップを表示することでステップ実行のような表示を実現しているが，コードの実行によるログの取得やそのログを元にした html の作成は中断することなく行っている．

文の実行を表すログは，図 2 のように実行中の文をハイライトさせることで表示した．加えて，これだけでは同じ関数が複数回呼び出された時，どの関数呼び出しにおいて実行されているのか判断できないため，関数呼び出しのログを利用して呼び出し元の関数のハイライトも行う．

Promise オブジェクトの状態や await 式の中断，再開は，図 3 のように Promise オブジェクトを返す関数呼び出しの背景色や中断，再開を行う await 式の横のメッセージによって表示した．これらの表示には Promise オブジェクト

```
1  function p () {
2    return new Promise(resolve => {
3      setTimeout(resolve, 10);
4    });
5  }
6
7  async function foo () {
8    await p();
9    await p();
10   await p();
11 }
12
13 foo();
```

図 2: 実行中の文とその呼び出し元の様子

```
1  function p () {
2    return new Promise(resolve => {
3      setTimeout(resolve, 10);
4    });
5  }
6
7  async function foo () {
8    await p();
9    await p(); ←await!
10   await p();
11 }
12
13 foo();
```

図 3: Promise オブジェクトの状態と async 関数のメッセージ

の作成，resolve，reject を表すログ，および async 関数の中断，再開を表すログを利用した．これにより，作成された Promise オブジェクトの状態と async 関数が中断，再開されるタイミングやその位置を視覚化している．

実行順序を表示する方法では前述の各ステップの処理の実行順序を一度に示すことで実行の様子を可視化する．実行順序は何が起きたかを示すメッセージ，コード中での該当箇所，該当するコード片の 3 つにより表示する．これにより，Promise オブジェクトの状態や async 関数の中断，再開も含めた実行順序を調べることが可能となる．実際の表示結果は次の 3.4 節にて示す．

3.4 適用結果

2.3 節にて実行順序の理解が難しい例としてあげた，ソースコード 8，ソースコード 9 に本ツールを適用し，実行順序を表示させた結果を図 4，図 5 に示す．

図 4，図 5 では，実際の実行順序に加えて，Promise の状態遷移や async 関数の中断，再開が行われる順序も表示されている．これより，本ツールはただ実行順序を調べるだけでなく，2.3 節で述べた間違って認識されがちな動作も表示できるものであることが確認できる．これらの動作の理解は実行順序を理解するための重要なポイントであるため，正しい実行順序とその実行順序になった理由の両方

```
1: 実行　@10行目1文字目　asyncTask();
2: 実行　@7行目2文字目　await wait(1000);
3: 実行　@2行目3文字目
  return new Promise((resolve, reject) => {
    setTimeout(resolve, time);
  });
4: 実行　@3行目3文字目　setTimeout(resolve, time);
5: Promiseオブジェクトを作成　@7行目8文字目　wait(1000)
6: 関数asyncTaskを中断　@7行目2文字目　await wait(1000)
7: Promiseオブジェクトを作成　@10行目1文字目　asyncTask()
8: 実行　@11行目1文字目　console.log("block?");
9: Promiseオブジェクトがresolve　@7行目8文字目　wait(1000)
10: 関数asyncTaskを再開　@7行目2文字目　await wait(1000)
11: 実行　@8行目2文字目　console.log("complete");
12: Promiseオブジェクトがresolve　@10行目1文字目　asyncTask()
```

図 4: ソースコード 8 の表示結果

```
1: 実行　@14行目1文字目　foo();
2: 実行　@7行目2文字目　var task1 = wait(1);
3: 実行　@2行目3文字目
  return new Promise((resolve, reject) => {
    setTimeout(resolve, num*1000);
  });
4: 実行　@3行目3文字目　setTimeout(resolve, num*1000);
5: Promiseオブジェクトを作成　@7行目14文字目　wait(1)
6: 実行　@8行目2文字目　var task2 = wait(2);
7: 実行　@2行目3文字目
  return new Promise((resolve, reject) => {
    setTimeout(resolve, num*1000);
  });
8: 実行　@3行目3文字目　setTimeout(resolve, num*1000);
9: Promiseオブジェクトを作成　@8行目14文字目　wait(2)
10: 実行　@9行目2文字目　await task2;
11: 関数fooを中断　@9行目2文字目　await task2
12: Promiseオブジェクトを作成　@14行目1文字目　foo()
13: Promiseオブジェクトがresolve　@7行目14文字目　wait(1)
14: Promiseオブジェクトがresolve　@8行目14文字目　wait(2)
15: 関数fooを再開　@9行目2文字目　await task2
16: 実行　@10行目2文字目　console.log("complete task2");
17: 実行　@11行目2文字目　await task1;
18: 関数fooを中断　@11行目2文字目　await task1
19: 関数fooを再開　@11行目2文字目　await task1
20: 実行　@12行目2文字目　console.log("complete task1");
21: Promiseオブジェクトがresolve　@14行目1文字目　foo()
```

図 5: ソースコード 9 の表示結果

をこの表示結果から知ることが可能となる．

以上より，本ツールは実行順序の考察に際して有用なものであり，本ツールの適用により実行順序の理解が難しいという問題を解決することができると考える．

4. 関連研究

本研究はコールバック地獄の改善に際し，その解決策の一つである async/await の問題が実行順序の理解の難しさにあることに着目した点，および async/await の実行の可視化を行うツールによりこの問題を対処した点において他の研究と異なる．我々の知る限り，async/await に関して同様の着眼点や解決手法を持つ先行研究は存在しない．

コールバック地獄の代表的な解決策，Promise についてはいくつかの研究がなされてきた．研究 [1] はコールバック関数を用いたコードから Promise を用いたコードへの書き換えの自動化に着目した研究である．この研究では Promise の代わりに Due という擬似 Promise オブジェクトを利用し，コールバック関数を用いて記述された非同期処理のコードを Due を用いたコードに自動的に変換するコンパイラを開発した．しかし，コンパイラの使用に際して，対象が Node.js のプログラムであること，with 関数・eval 関数を用いていないことなどの制限がある．加えて，変換のすべての過程が自動化されたわけではなく，それぞれのコールバック関数について，これを変換すべきかどうか手動で判断する必要がある．そのため，コールバック地獄の完全な解決策とはなり得ないと考えられる．

また，研究 [2], [3] は Promise を用いたコードのデバッグの難しさに着目した研究である．研究 [2] では λ p という計算法，および λ p に基づいて構築された promise graph という Promise に関連したエラーの検出に役立つプログラム表現を提案した．研究 [3] では遠隔プロセスにて評価される Promise のデバッグを改善するためのインターフェースを提案した．これらの研究はどちらも Promise のデバッグにフォーカスしており，本研究で対象としたコールバック地獄を解決することそのものには焦点が当てられていない．

ECMAScript2017 から実装された比較的新しい機構である async/await について注目する研究は，研究 [4] を除きほとんど行われていない．研究 [4] では async/await に注目した上でこれはコールバック地獄の十分な解決策ではないとし，async/await に代わるコールバック地獄の解決法として，継続 [6] とアスペクト指向言語 [7] を利用した新しい機構 Sync/CC を提案した．この論文では async/await の問題点は async，await といった特殊なキーワードを随所に記述しなくてはいけない点，使い方をデペロッパがカスタムできない点，実行前に Babel[8] というトランスパイラを通さなければならない点にあるとし，提案手法 Sync/CC においてこれらの問題点を解決した．ただし，現在では多くのブラウザにおいて async/await がサポートされているため，3 つ目のトランスパイラを通す必要がある点は多くの場合問題とならないものである [10]．また，Sync/CC には次の 2 つの問題点が考えられる．1 つ目は Sync/CC で使用している継続とアスペクト指向言語の概念は現在の JavaScript エンジンでは実装されていないため，Sync/CC は Java によって記述された JavaScript の実装である Rhino[9] 上でしか動作しない点である．2 つ目は

Sync/CC では await キーワードを用いて完了を待つべき処理であることを明示する代わりに，アスペクト指向言語を用いてこのことを定義する必要があるため，一概に簡潔な記述になるとは言えない点である．これらの問題点より，Sync/CC も async/await に代わるコールバック地獄の完全な解決策とはなり得ないと考えられる．

コールバックそのものに関する問題に着目した研究としては，研究 [5] があげられる．研究 [5] では非同期コールバックによって発生しうるデータ競合について述べられている．このデータ競合は，その記述法にかかわらず非同期な呼び出しがある限り起こりうるものである．特に async/await を用いて記述した場合，非同期な呼び出しを同期的な呼び出しに近い形で記述することができるため，非同期な呼び出しと同期的な呼び出しが混同されるリスクが高まる．その結果，async/await への書き換えによりプログラマが意図しないデータ競合が発生しうると考えられる．本研究ではデータ競合の検出そのものは研究対象としなかったが，この問題からも実行順序の理解の重要性を窺うことができる．

5. おわりに

本研究では，コールバック地獄の主要な解決策 Promise や async/await を用いたコード例について調査する中で，現状において async/await がコールバック地獄の最も有力な解決策であるが，async/await には実行順序を理解することが難しいという問題点があることを明らかにした．そして，async/await の実行順序の理解にあたってトレーサが有効であることを発見し，トレーサを利用して async/await の実行順序を可視化するツールを作成した．このツールを適用すると，実際の実行順序に加えてその際の Promise や async/await の動作の説明を取得できるため，なぜその実行順序になったのかをこれらを用いて推測することが可能となる．したがって，このツールは async/await の実行順序が難解であるという問題を解決するものであると言える．

残された課題としては，実行の様子のより良い表示方法の検討があげられる．例えば，作成された Promise オブジェクトの状態をコード上の記述位置による表示だけでなく，それぞれのオブジェクトごとに管理，表示することで，より Promise の状態を明確に把握することができるようになると考える．他の課題として，実行時間の変化によって実行順序が変化する可能性がある部分とそうでない部分を識別して表示するなど，実行時間の非決定性による実行順序への影響を調べる機能の追加もあげられる．また，4 章で述べた async/await への書き換えによって生じるデータ競合の問題は，実行順序が非常に大きく関与するものである．そのため，トレーサを拡張することにより，このデータ競合の検出器に応用することが可能であると考える．

謝辞 本研究は JSPS 科研費 26330077 の助成を受けたものです．

参考文献

[1] E. Brodu, S. Frenot and F. Oble: ”Toward automatic update from callbacks to Promises”, AWeS (2015).
[2] M. Madsen, O. Lhotak and F. Tip: ”A Model for Reasoning About JavaScript Promises”, OOPSLA (2017).
[3] M. Leske, A. Chis and O. Nierstrasz: ”A promising approach for debugging remote promises”, IWST (2016).
[4] P. Leger and H. Fukuda: ”Sync/CC: Continuations and Aspects to Tame Callback Dependencies on JavaScript Handlers”, SAC, pp.1245-1250 (2017).
[5] M. Billes: ”Race-Driven UI-Level Test Generation for JavaScript-Based Web Applications”, SPLASH Companion, pp.81-82 (2015).
[6] D. P. Friedman and M. Wand: ”Reification: Reflection without Metaphysics”, LFP, pp.348-335 (1984).
[7] G. Kiczales, J. Irwin, J. Lamping, J.M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar: ”Aspect-Oriented Programming”, Communications of the ACM, pp.29-32 (2001).
[8] Babel: Babel, ”https://babeljs.io/” (2017.11.14).
[9] Mozilla: Rhino, ”https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino” (2017.11.14).
[10] Github: ECMAScript 2016+ compatibility table, ”http://kangax.github.io/compat-table/es2016plus/” (2018.1.22).
[11] ESLint: Disallows unnecessary return await (no-return-await), ”https://eslint.org/docs/rules/no-return-await” (2017.12.12).
[12] npm: esprima, ”https://www.npmjs.com/package/esprima” (2017.12.09).
[13] npm: estraverse, ”https://www.npmjs.com/package/estraverse” (2017.12.09).
[14] npm: escodegen, ”https://www.npmjs.com/package/escodegen” (2017.12.09).
[15] browserify: browserify, ”http://browserify.org/” (2017.12.14).