非同期処理

JavaScriptのエコシステムは伝統的にはスレッドを使わない計算モデルを使い、それの効率をあげる方向で進化してきました。 例えば、スリープのような、実行を行の途中で止めるような処理は基本的に持っていませんでした。 10秒間待つ、というタスクがあった場合には、10秒後に実行される関数を登録する、といった具合の処理が提供され、その場で「10秒止める」という処理を書く機能は提供されませんでした。

JavaScriptは伝統的に、ホストとなる環境（ブラウザ）の中で実行される、アプリケーション言語として使われることが多く、ホスト側のアプリケーションから見て、長時間ブロックされるなどの行儀の悪い動きをすることが忌避されてきたからではないかと思います。 そのせいかどうかわかりませんが、他の言語とは多少異なる進化を遂げてきました。

ブラウザでは、数々のHTML側のインタラクション、あるいはタイマーなどのイベントに対して、あらかじめ登録しておいたイベントハンドラの関数が呼ばれる、というモデルを採用しています。 JavaScriptがメインの処理系となるNode.jsでも、OSが行う、時間のかかる処理を受けるイベントループがあり、OS側の待ち処理に対してコールバック関数をあらかじめ登録しておきます。そして、結果の準備ができたら、それが呼ばれるというモデルです。

ES2015以降、このコーディングスタイルにも手が入り、土台の仕組みはコールバックではありますが、多数の非同期を効率よく扱う方法が整備されてきました。 現在、見かける非同期処理の書き方は大きく3種類あります。

- コールバック
- `Promise`
- `async` / `await`

本章ではそれらを紹介していきます。なお、非同期処理の例外処理については、例外処理の章で扱います。

非同期とは何か

JavaScriptの処理系には、現在のシステムのUIを担うレイヤーとしてかなりの開発資金が投入されてきました。 ブラウザ戦争と呼ばれる時期には、各ブラウザが競うようにJavaScriptやウェブブラウザの画面描画の速度向上を喧伝し、他社製のブラウザよりも優れていると比較のベンチマークを出していたりしました。 その結果としては、スクリプト言語としてはJavaScriptはトップクラスの速度になりました。Just In Timeコンパイラという実行時の最適化が効くと、コンパイル言語に匹敵する速度を出すことすらあります。

CPU速度が問題になることはあまりないとはいえ、コードで処理するタスクの中には長い時間の待ちを生じさせるものがいくつかあります。 例えば、タイマーなどもそうですし、外部のサーバーやデータベースへのネットワークアクセス、ローカルのファイルの読み書きなどは往復でミリ

秒、場合によっては秒に近い遅延を生じさせます。 JavaScriptは、そのような時間のかかる処理は基本的に「非同期」という仕組みで処理を行います。 タイマー呼び出しをする次のコードを見て見ます。

```
console.log("タイマー呼び出し前");
setTimeout(() => {
  console.log("時間が来た");
}, 1000);
console.log("タイマー呼び出し後");
```

このコードを実行すると次の順序でログが出力されます。

```
タイマー呼び出し前
タイマー呼び出し後 // 上の行と同時に表示
時間が来た         // 1秒後に表示
```

JavaScriptでは時間のかかる処理を実行する場合、完了した後に呼び出す処理を処理系に渡すことはあっても、そこで処理を止めることはありません。 タイマーを設定する `setTimeout()` 関数の実行自体は即座に完了し、その次の行がすぐ呼ばれます。 そして時間のかかるタイマーの待ちが完了すると、渡してあった関数が実行されます。 処理が終わるのをじっくり待つ（同期）のではなく、完了したら後から連絡してもらう（非同期）のがJavaScriptのスタイルです。

昔のJavaScriptのコードでは、時間のかかる処理を行う関数は、かならず引数の最後がコールバック関数でした。 このコールバック関数の中にコードを書くことで、時間のかかる処理が終わったあとに実行する、というのが表現できました。

コールバックは使わない

以前はJavaScriptで数多くの非同期処理を実装しようとすると、多数のコールバック関数を扱う必要があり、以前はコールバック地獄と揶揄されていました。

非同期の書き方

```
// 旧: Promise以前
func1(引数, function(err, value) {
  if (err) return err;
  func2(引数, function(err, value) {
    if (err) return err;
    func3(引数, function(err, value) {
      // 最後に実行されるコードブロック
    });
  });
});
```

その後、`Promise`が登場し、ネストが1段になり、書きやすく、読みやすくなりました。`Promise`はその名の通り「重たい仕事が終わったら、あとで呼びに来るからね」という約束です。これにより、上記のような、深いネストがされたコードを触れる必要が減ってきました。何階層もの待ちが発生しても、1段階のネストで済むようになりました。

この`Promise`の実装は、文法の進化に頼ることなく、既存のJavaScriptの文法の上で実装されたトリックで実現できました。コミュニティベースで実現されたソリューションです。この`Promise`は現在も生き続けている方法です。直接書く機会は減ると思いますが、`Promise`について学んだことは無駄にはなりません。

非同期の書き方

```
// 中: Promise以後
fetch(url).then(resp => {
  return resp.json();
}).then(json => {
  console.log(json);
}).catch(e => {
  // エラー発生時にここを通過する
}).finally(() => {
  // エラーが発生しても、正常終了時もここを通過する
});
```

`Promise`の`then()`節の中に、前の処理が終わった時に呼び出して欲しいコードを書きます。また、その`then()`のレスポンスもまた`Promise`なので、連続して書けるというわけです。また、この`then()`の中で`return`で返されたものが次の`then()`の入力になります。`then()`の中で`Promise`を返すと、その返された`Promise`が解決すると、その結果が次の`then()`の入力になります。遅い処理を割り込ませるイメージです。`catch()`と`finally()`は通常の例外処理と同じです。`finally()`はES2018で取り込まれた機能です。

コールバック地獄では、コードの呼び出し順が上から下ではなく上→下→中と分断されてしまいますが、`Promise`の`then()`節だけをみれば、上から下に順序良く流れているように見えます。初めて見ると面食らうかもしれませんが、慣れてくるとコールバックよりも流れは追いやすいでしょう。

この`Promise`がJavaScript標準の方法として決定されると、さらなる改善のために`await`という新しいキーワードが導入されました。これは`Promise`を使ったコードの、`then()`節の中だけを並べたのとほぼ等価になります。それにより、さらにフラットに書けるようになりましたし、行数も半分になります。内部的には、`await`はまったく新しい機構というわけではなく、`Promise`を扱いやすくする糖衣構文で、`then()`を呼び出し、その引数で渡される値が関数の返り値となるように動作します。`Promise`対応のコードを書くのと、`await`対応のコードを書くのは差がありません。`Promise`でない返り値の関数の前に`await`を書いても処理が止まることはありません（エラーになることはありません）。

非同期の書き方

```
// 新: 非同期処理をawaitで待つ（ただし、awaitはasync関数の中でのみ有効）
const resp = await fetch(url);
const json = await resp.json();
console.log(json);
```

`await` を扱うには、 `async` をつけて定義された関数でなければなりません。 TypeScriptでは、 `async` を返す関数の返り値は必ず `Promise` になります。 ジェネリクスのパラメータとして、返り値の型を設定します。

```
async function(): Promise<number> {
  await 時間のかかる処理();
  return 10;
}
```

なお、 `Promise` を返す関数は、関数の宣言文を見たときに動作が理解しやすくなるので `async` をつけておく方が良いでしょう。ESLintのTypeScriptプラグインでも、推奨設定でこのように書くことを推奨しています[1]。

TypeScriptの処理系は、この `Promise` の種類と、関数の返り値の型が同一かどうかを判断し、マッチしなければエラーを出してくれます。 非同期処理の場合、実際に動かしてデバッグしようにも、送る側の値と、受ける側に渡ってくる値が期待通りかどうかを確認するのが簡単ではありません。 ログを出して見ても、実際に実行されるタイミングがかなりずれていることがありえます。 TypeScriptを使うメリットには、このように実際に動かすデバッグが難しいケースでも、型情報を使って「失敗するとわかっている実装」を見つけてくれる点にあります。

比較的新しく作られたライブラリなどは最初から `Promise` を返す実装になっていると思いますが、そうでないコールバック関数方式のコードを扱う時は `new Promise` を使って `Promise` 化します。

```
// setTimeoutは最初がコールバックという変態仕様なので仕方なくnew Promise
const sleep = async (time: number): Promise<number> => {
  return new Promise<number>(resolve => {
    setTimeout(()=> {
      resolve(time);
    }, time);
  });
};

await sleep(100);
```

末尾がコールバック、コールバックの先頭の引数はErrorという、2010年代の行儀の良いAPIであれば、`Promise`化してくれるライブラリがあります。 Node.js標準にもありますし、npmで調べてもたくさんあります。

```
// Node.js標準ライブラリのpromisifyを使う

import { promisify } from "util";
import { readFile } from "fs";
const readFileAsync = promisify(readFile);

const content = await readFileAsync("package.json", "utf8");
```

1 `@typescript-eslint/promise-function-async`という設定が該当します。

非同期と制御構文

TypeScriptで提供されている`if`や`for`、`while`などは関数呼び出しを伴わないフラットなコードなので`await`とも一緒に使えます。 `Promise`やコールバックを使ったコードで、条件によって非同期処理を1つ追加する、というコードを書くのは大変です。 試しに、TypeScriptのPlayGroundで下記のコードを変換してみるとどうなるか見て見ると複雑さにひっくり返るでしょう。

```
// たまに実行される
async function randomRun() {
}

// 必ず実行される
async function finallyFunc() {
}

async function main(){
  if (Date.now() % 2 === 1) {
    await randomRun();
  }
  await finallyFunc();
}

main();
```

これを見ると、`await`は条件が複雑なケースでも簡単に非同期を含むコードを扱えるのがメリットであることが理解できるでしょう。

`await`を使うと、ループを一回回るたびに重い処理が完了するのを待つことができます。 同じループでも、配列の`forEach()`を使うと、1要素ごとに`await`で待つことはできませんし、すべてのループの処理が終わったあとに、何かを行わせることもできません。

```
// for of, if, while, switchはawaitとの相性も良い
for (const value of iterable) {
  await doSomething(value);
}
console.log("この行は全部のループが終わったら実行される");
```

```
// このawaitでは待たずにループが終わってしまう
iterable.forEach(async value => {
  await doSomething(value);
});
console.log("この行はループ内の各処理が回る前に即座に実行される");
```

`Promise` の分岐と待ち合わせの制御

`Promise` は「時間がかかる仕事が終わった時に通知するという約束」という説明をしました。みなさんは普段の生活で、時間がかかるタスクというのを行ったことがありますよね？味噌汁をガスレンジあたためつつ、ご飯を電子レンジで温め、両方終わったらいただきます、という具合です。 `Promise` および、その完了を待つ `await` を使えば、そのようなタスクも簡単に実装できます。

```
async function 味噌汁温め(): Promise<味噌汁> {
  await ガスレンジ();
  return new 味噌汁();
}

async function ご飯温め(): Promise<ご飯> {
  await 電子レンジ();
  return new ご飯();
}

const [a味噌汁, aご飯] = await Promise.all([味噌汁温め(), ご飯温め()]);
いただきます(a味噌汁, aご飯);
```

`味噌汁温め()` と `ご飯温め()` は `async` がついた関数です。省略可能ですがあえて返り値に `Promise` をつけています。これまでの例では、 `async` 関数を呼ぶ時には `await` をつけていました。 `await` をつけると、待った後の結果（ここでは味噌汁とご飯のインスタンス）が帰ってきます。 `await` をつけないと、 `Promise` そのものが帰ってきます。

この `Promise` の配列を受け取り、全部の `Promise` が完了するのを待つのが `Promise.all()` です。 `Promise.all()` は、引数のすべての結果が得られると、解決して結果をリストで返す `Promise` を返します。 `Promise.all()` の結果を `await` すると、すべての結果がまとめて得られます。

この `Promise.all()` は、複数のウェブリクエストを同時に並行で行い、全てが出揃ったら画面を描画する、など多くの場面で使えます。 ループで複数の要素を扱う場合も使えます。

なお、 `Promise.all()` の引数の配列に、 `Promise` 以外の要素があると、即座に完了する `Promise` として扱われます。

類似の関数で `Promise.race()` というものがあります。 これは `all()` と似ていますが、全部で揃うと実行されるわけではなく、どれか一つでも完了すると呼ばれます。 レスポンスの値は、引数のうちのどれか、ということで、結果を受け取る場合は処理が少し複雑になります。 結果を扱わずに、5秒のアニメーションが完了するか、途中でクリックした場合には画面を更新する、みたいな処理には適しているかもしれません。

ループの中の `await` に注意

`for` ループと `await` が併用できることはすでに紹介しました。 しかし、このコード自体は問題があります。

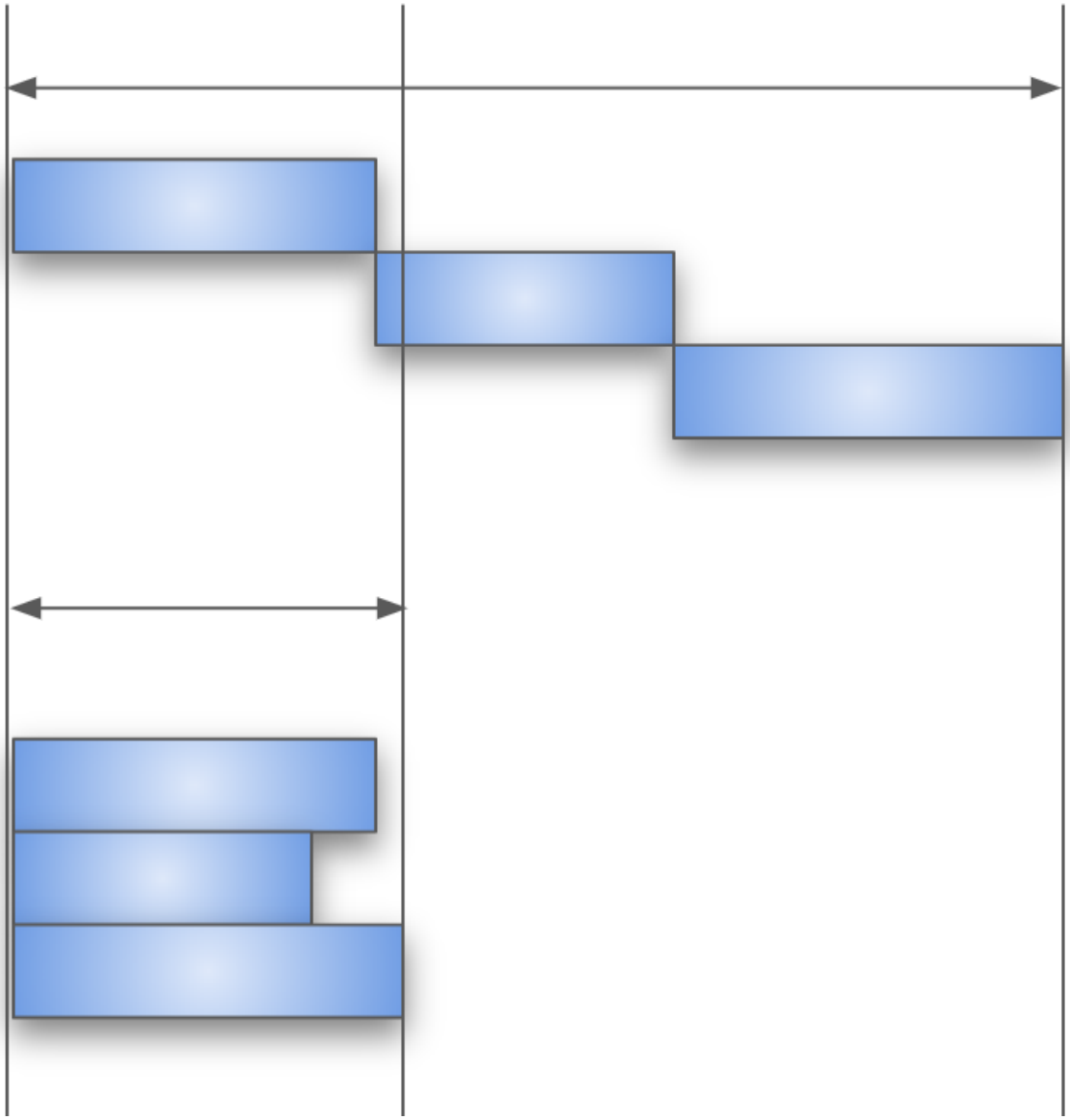
```
for (const value of iterable) {
  await doSomething(value);
}
```

この `doSomething()` の中で外部APIを呼び出しているとすると、要素数×アクセスにかかる時間だけ、処理時間がかかります。 要素数が多い場合、要素数に比例して処理時間が伸びます。 この `await` を内部にもつループがボトルネックとなり、ユーザーレスポンスが遅れることもありえるかもしれません。 上記のような例を紹介はしましたが、基本的にループ内の `await` は警戒すべきコードです。

この場合、 `Promise.all()` を使うと、全部の重い処理を同時に投げ、一番遅い最後の処理が終わるまで待つことができます。 配列の `map()` は、配列の中のすべての要素を、指定の関数に通し、その結果を格納する新しい配列（元の配列と同じ長さ）を作り出して返します。 詳しくは関数型スタイルのコーディングの紹介で触れますが、このメソッドを使うと、上記の例のような、 `Promise` の配列を作ることができます。 `Promise.all()` の引数は、 `Promise` の配列ですので、これをそのまま渡すと、全部の処理が終わるのを待つ、という処理が完成します。

```
await Promise.all(
  iterable.map(
    async (value) => doSomething(value)
  )
);
```

図で見て見ると、この違いは一目瞭然でしょう。



`Promise.all()` が適切ではない場面もいくつかあります。

例えば、外部のAPI呼び出しをする場合、たいてい、秒間あたりのアクセス数が制限されています。 配列に100個の要素があるからといって100並列でリクエストを投げるとエラーが帰って来て正常に処理が終了しないこともありえます。 その場合は、並列数を制御しつつ `map()` と同等のことを実現してくれる `p-map` [2] といったライブラリを活用すると良いでしょう。

2 https://www.npmjs.com/package/p-map

`for` ループ内部の `await` のように、順番に処理をするための専用構文もあります。 `asyncIterator` というプロトコルを実装したオブジェクトでは、 `for await (const element of obj)` というES2018で導入された構文も使えるようになります。 `fetch` のレスポンスのボディがそれにあたります。 普段は `json()` メソッドなどで一括で変換結果を受け取ると思いますが、細切れのブロック単位で受信することもできます。 この構文を使

うと、それぞれのブロックごとにループを回す、という処理が行えます。 ただし、それ以外の用途は今のところ見かけませんし、この用途で使うところも見たことがありませんので、基本的にはループの中の `await` は要注意であることは変わりありません。

非同期で繰り返し呼ばれる処理

`async` / `await` は便利なものですが、ワンショットで終わるイベント向けです。繰り返し行われるイベント（ `addEventListener()` を使うようなスクロールイベントとか、画面のリサイズ、 `setInterval()` の繰り返しタイマー）に対しては引き続きコールバック関数を登録して使います。

そこをモダンにしようという動きにはRxJSがあります。

まとめ

`Promise` と `await` について紹介しました。 非同期は本質的に、難しい処理です。 その難しい処理をなるべく簡単に表現しよう、という試みがむかしから試行錯誤されてきました。 その1つの成果がこのTypeScriptで扱えるこの2つの要素です。

上から順番に実行されるわけではありませんし、なかなかイメージが掴みにくいかもしれません。 最終的には、頭の中で、どの部分が並行で実行されて、どこで待ち合わせをするか、それがイメージができれば、 非同期処理の記述に強いTypeScriptのパフォーマンスを引き出せるでしょう。