導入: callbacks

JavaScript の多くのアクションは 非同期 です。

例えば、次の `loadScript(src)` を見てください:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

この関数の目的は新しいスクリプトを読み込むことです。ドキュメントに `<script src="…">` を追加したとき、ブラウザはそれを読み込み、実行します。

このように使うことができます:

```
// スクリプトを読み込み、実行する
loadScript('/my/script.js');
```

そのアクション(スクリプトの読み込み)は、今ではなく後で終わるため、関数は “非同期” と呼ばれます。

`loadScript` の呼び出しにより、スクリプトの読み込みを開始し、その後実行を続けます。スクリプトが読み込まれている間、それ以降のコードは実行が終わり、もし読み込みに時間がかかる場合は、他のスクリプトも実行される可能性があります。

```
loadScript('/my/script.js');
// loadScript の下のコードはスクリプトの読み込みが終わるのを待ちません
// ...
```

今、新しいスクリプトがロードされたときにそれを使いたいとします。恐らく新しい関数を宣言しているので、それらを実行したいとします。

…しかし、 `loadScript(…)` 呼び出しの直後にそれをしても上手く動作しません:

```
loadScript('/my/script.js'); // このスクリプトは "function newFunction() {…}" を持っています

newFunction(); // そのような関数はありません!
```

もちろん、恐らくブラウザにはスクリプトを読み込む時間がありませんでした。そのため、新しい関数の即時呼び出しは失敗します。今のところ、 `loadScript` 関数は読み込みの完了を追跡する方法を提供していません。スクリプトは読み込まれ、最終的に実行されますが、そのスクリプトの新しい関数や変数を使用するために、それらがいつ起きるのかを知りたいです。

`loadScript` に2つ目の引数に、スクリプトが読み込まれたときに実行する `callback` 関数を追加しましょう:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
```

```
  script.onload = () => callback(script);

  document.head.append(script);
}
```

ロードしたスクリプトにある新しい関数を呼びたい場合は callback に書きます。:

```
loadScript('/my/script.js', function() {
  // コールバックはスクリプトがロード後に実行されます
  newFunction(); // なので、これは動作します
  ...
});
```

That's the idea: 第2引数は、アクションが完了したときに実行される関数（通常は無名）です。

ここで、実際のスクリプトを使った実行可能な例を示します:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
  alert(`Cool, the ${script.src} is loaded`);
  alert( _ ); // ロードされたスクリプトで宣言されている関数
});
```

これは "コールバックベース" と呼ばれる非同期プログラミングのスタイルです。非同期の処理をする関数は、関数が完了した後に実行するための `callback` を提供します。

ここでは `loadScript` でそれを行いましたが、もちろん一般的なアプローチです。

Callback の中の callback

２つのスクリプトを順次読み込む方法: 最初の１つを読み込み、2つ目はその後に読み込む？

自然な解決策は、2つ目の `loadScript` 呼び出しを callback の中に置くことです。次のようになります:

```
loadScript('/my/script.js', function(script) {

  alert(`Cool, the ${script.src} is loaded, let's load one more`);

  loadScript('/my/script2.js', function(script) {
    alert(`Cool, the second script is loaded`);
  });

});
```

外側の `loadScript` の完了後、そのコールバックは内側の `loadScript` を開始します。

…仮にもっとスクリプトを読み込みたい場合はどうなりますか？

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...すべてのスクリプトが読み込まれるまで続きます
    });

  })

});
```

したがって、すべての新しいアクションはコールバックの中です。少ないアクションの場合は問題ありませんが、多い場合には問題です。そのため、この後別の方法を見ていきます。

エラー処理

上の例ではエラーを考慮していませんでした。もしスクリプト読み込みが失敗した場合どうしますか？コールバックはそれに対応する必要があります。

これは、読み込みエラーを追跡する `loadScript` の改良版です:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src}`));

  document.head.append(script);
}
```

これは成功時に `callback(null, script)` を呼び、それ以外の場合には `callback(error)` を呼びます。

使用方法:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // エラー処理
  } else {
    // スクリプトの読み込みが成功
  }
});
```

もう一度言いますが、 `loadScript` で使った方法は、実際に非常に一般的なものです。これは “エラーファーストなコールバック” スタイルと呼ばれます。

慣例は次の通りです:

1. `callback` の最初の引数は、エラーが発生した場合のために予約されています。そして `callback(err)` が呼ばれます。

2. 2つ目の引数(と必要に応じて以降の引数)は正常な結果を得るためのもので、 `callback(null, result1, result2…)` が呼ばれます。

なので、単一の `callback` 関数は、エラー報告と結果渡し両方のために使われます。

破滅のピラミッド

初見では、これは非同期コーディングの実行可能な方法です。確かにその通りです。1つまたは2つ程度のネストされた呼び出しの場合には問題なく見えます。

しかし、次々に続く複数の非同期アクションの場合、次のようなコードを持つことになります:

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...すべてのスクリプトが読み込まれるまで続く (*)
          }
        });

      }
    })
  }
});
```

上記のコードでは:

1. `1.js` をロードし、エラーがなければ
2. `2.js` をロードします。エラーがなければ
3. `3.js` をロードします。エラーがなければ – 他の何か `(*)` を行います。

呼び出しがよりネストされるにつれて、特に `...` ではなく、実際により多くのループや条件式などを含むコードがある場合、コードはより深くなり、益々管理が難しくなります。

これは “コールバック地獄” や “破滅のピラミッド” と呼ばれる場合があります。

ネストされた呼び出しの “ピラミッド” はすべての非同期アクションで右に成長していきます。まもなく、それは制御不能になります。

したがって、このコーディング方法はあまり良くありません。

私たちは、次のようにすべてのアクションをスタンドアロンの関数にすることで、この問題を軽減することができます。:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...すべてのスクリプトが読み込まれた後に続く (*)
  }
};
```

どうでしょう？ 同じことをしますが、今や深いネストはありません。なぜならすべてのアクションをトップレベルの関数に分離したからです。

これは機能しますが、コードはばらばらになったスプレッドシートのように見えます。これは読みにくく、あなたも気づいたでしょう。1つは、読んでいる間に関数間で視点のジャンプが必要になります。これは不便で、読者がコードに精通しておらず、どこに視点を移動させたらよいか分からない場合は特に困ります。

また、 `step*` という名前の関数はすべて1度の使用であり、“破滅のピラミッド” を避けるためだけに作られています。だれもこのアクションの外でそれらを再利用するつもりはありません。そのため、ここには少し散らかっている名前空間があります。

私たちは何かより良いものを望んでいます。

幸いにも、このようなピラミッドを回避するための他のう方法があります。ベストな方法の1つは次のチャプターで説明する “promise” を使うことです。