JavaScriptの非同期処理について

あじぇんだ

- 自己紹介
- そもそも非同期処理って？
- Callbackの問題点
- Promiseを使おう
- co + generatorを使おう
- async / await 使おう
- まとめ

あじぇんだ

- 自己紹介
- そもそも非同期処理って？
- Callbackの問題点
- Promiseを使おう
- co + generatorを使おう
- async / await 使おう
- まとめ

自己紹介

- 名前：沢田晃一（さわだこういち）
- 年齢：３１歳
- 職種：WEBエンジニア
- 好きな言語：Node.js, JavaScript

あじぇんだ

- 自己紹介
- そもそも非同期処理って？
- Callbackの問題点
- Promiseを使おう
- co + generatorを使おう
- async / await 使おう
- まとめ

非同期処理とは

- 同期処理
  あるタスクが実行されている間、
  他のタスクは中断され処理が終わるのを待つ

- 非同期処理
  あるタスクが実行されている間、
  他のタスクが別の処理を実行できる

1秒待つ処理の場合

同期的な処理の場合

```
function wait(ms) {
  let now = Date.now();
  while(Date.now() < now + ms);
}
wait(1000);
console.log('1秒まった！');
```

非同期処理の場合

```
function wait(ms, callback) {
  setTimeout(callback, ms);
}
wait(1000, () => {
  console.log('1秒まった！');
})
```

1秒待つ処理の場合

同期的な処理の場合

```
function wait(ms) {
  let now = Date.now();
  while(Date.now() < now + ms);
}
wait(1000);
console.log('1秒まった！');
```

waitの処理が終わるまで
何もできない！

非同期処理の場合

```
function wait(ms, callback) {
  setTimeout(callback, ms);
}
wait(1000, () => {
  console.log('1秒まった！');
})
```

waitの処理を待たずに
他の処理ができる！

あじぇんだ

- 自己紹介
- そもそも非同期処理って？
- Callbackの問題点
- Promiseを使おう
- co + generatorを使おう
- async / await 使おう
- まとめ

Callbackの問題点

シンプルな場合

```
function wait(ms, callback) {
  setTimeout(callback, ms);
}
wait(1000, () => {
  console.log('1秒まった！');
})
```

わかりやすい

Callbackの問題点

複雑になると、、、

ネストが深くなり
分かりにくくなる！

```
function wait(ms, callback) {
  setTimeout(callback, ms);
}

function exec() {
  wait(1000, ()=> {
    console.log(1);
    wait(1000, () => {
        console.log(2);
        wait(1000, () => {
          console.log(3);
          wait(1000, () => {
            console.log(4);
            wait(1000, () => {
              console.log(5);
            });
          });
        });
    });
  });
}

exec();
```

あじぇんだ

- 自己紹介
- そもそも非同期処理って？
- Callbackの問題点
- Promiseを使おう
- co + generatorを使おう
- async / await 使おう
- まとめ

Promiseとは

- 非同期処理を抽象化したオブジェクトとそれを操作する仕組み

- Promiseを使うことで、複雑だった非同期処理をきれいにまとめることができる

- Promiseの本
  http://azu.github.io/promises-book/

動かしながら説明する

＼(^o^)／

Promiseで書くとこうなる

Callback

```
function wait(ms, callback) {
  setTimeout(callback, ms);
}

function exec() {
  wait(1000, ()=> {
    console.log(1);
    wait(1000, () => {
        console.log(2);
        wait(1000, () => {
          console.log(3);
        });
    });
  });
}

exec();
```

Promise

```
function wait(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

function exec() {
  wait(1000).then(() => {
    console.log(1);
    return wait(1000);
  }).then(() => {
    console.log(2);
    return wait(1000);
  }).then(() => {
    console.log(3);
  });
}

exec();
```

あじぇんだ

- 自己紹介
- そもそも非同期処理って？
- Callbackの問題点
- Promiseを使おう
- co + generatorを使おう
- async / await 使おう
- まとめ

coとgenerator

- generatorとは
  処理の中断と再開ができる関数。
  中断した場所の結果を得ることができる。

- coとは
  npmで公開されているパッケージで、
  generatorと組み合わせることで非同期処理を
  より分かりやすくしてくれる

動かしながら説明する

＼(^o^)／

co+generatorで書くとこうなる

Promise

```
function wait(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

function exec() {
  wait(1000).then(() => {
    console.log(1);
    return wait(1000);
  }).then(() => {
    console.log(2);
    return wait(1000);
  }).then(() => {
    console.log(3);
  });
}

exec();
```

co+generator

```
const co = require('co');

function wait(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

function* exec() {
  yield wait(1000);
  console.log(1);
  yield wait(1000);
  console.log(2);
  yield wait(1000);
  console.log(3);
}

co(exec);
```

coでやってること
(簡略化バージョン)

```
function getPromise(value) {
  return Promise.resolve(value);
}

function coLike(gen, value) {
  if (typeof gen.next !== 'function') {
    gen = gen();
  }

  let next = gen.next(value);
  if (next.done) {
    return;
  }
  getPromise(next.value).then(value => coLike(gen, value))
}
```

あじぇんだ

- 自己紹介
- そもそも非同期処理って？
- Callbackの問題点
- Promiseを使おう
- co + generatorを使おう
- async / await 使おう
- まとめ

async関数とawait

- awaitとは
  Promiseを同期的なように（みせかける）機能。
  Promiseの前にawaitと書くことでPromiseの
  終了を待つことができる

- async関数とは
  awaitの条件として、関数の前にasyncとついた
  関数内でしか利用できない。
  つまり、awaitを利用するよーという宣言。

動かしながら説明する

＼(^o^)／

async/awaitで
書くとこうなる

co+generator

```
const co = require('co');

function wait(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

function* exec() {
  yield wait(1000);
  console.log(1);
  yield wait(1000);
  console.log(2);
  yield wait(1000);
  console.log(3);
}

co(exec);
```

async/await

```
function wait(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function exec() {
  await wait(1000);
  console.log(1);
  await wait(1000);
  console.log(2);
  await wait(1000);
  console.log(3);
}

exec();
```

あじぇんだ

- 自己紹介
- そもそも非同期処理って？
- Callbackの問題点
- Promiseを使おう
- co + generatorを使おう
- async / await 使おう
- まとめ

まとめ

- Promiseを使うことで、コールバック地獄からおさらばできる

- generatorやasync/awaitを使うことで非同期処理をさらに分かりやすくできる

- 使ったことない人は使ってみよう