非同期処理

目次

1.非同期とは

外部のサーバーやデータベース、API へのアクセス、ローカルファイルの読み書きなどは往復で遅延を生じさせる。JavaScript(TypeScript)はそのような時間のかかる処理を基本的に**非同期**という仕組みで処理を行う。(非同期処理)。 **JavaScript(TypeScript)はシングルスレッドだが効率よく処理を行うために重要な仕組である。(時間がかかる処理が原因で全体が止まってしまうことを避ける。)**

```
//処理1
console.log("時間のかかる処理の前");
//処理2
setTimeout(() => {
  console.log("時間のかかる処理");
}, 3000);
//処理3
console.log("時間のかかる処理の後");
```

ログは以下のように出力される

```
時間のかかる処理の前; //処理1
時間のかかる処理の後; //処理3
時間のかかる処理; //処理2
```

3 秒後に結果を返す`setTimeout`の処理(非同期処理)は待たずに、次の処理へ進む。 非同期処理は裏側で動いている。

- 「時間のかかる処理の前」の log を出力
- `setTimeOut`の呼び出し
- 「時間のかかる処理の後」の log を出力
- 3 秒後に`setTimeOut`が実行され、「時間のかかる処理」の log を出力

上から順番に、処理が実行され、前の処理が終わるまで待つ(同期)ではなく、完了後に後から連絡をもらう(非同期)。

2.非同期処理の制御

非同期処理は処理が実行中なのか実行が完了したのかトレースがしずらい。 非同期処理の制御の書き方は大きく 3 種類ある。TypeScript では非同期処理にも型を定義することで、外部のサーバーやデータベースから安全にデータを取得することができる。

> 非同期処理についても制御することで同期的な処理にすることができる

- コールバック
- `Promise`
- `async/await`

3.コールバック

コールバック関数を使い、非同期処理を実装すると何階層ものネスト構造となり、コードの見通しが悪くなり、コールバック地獄といわれていた。(今は使われていない)

```
//ファイル読み込みを1つめから同期的に行う場合。
const btn = document.querySelector(".btn");
btn.addEventListener("click", function () {
  //1つめのファイルの読み込み
  req.addEventListener("load", (e) => {
    //2つめのファイルの読み込み
    req2.addEventListener("load", (e) => {
      //3つめのファイルの読み込み
      req.addEventListener("load", (e) => {});
    });
  });
});
```

4.プロミス

プロミスオブジェクトという状態を管理するオブジェクトの引数に、非同期処理を記述した関数を持たせる。何階層ものネストが 1 段階で書ける。

基本の書き方

- コードの概要

```
//Promiseの作成
new Promise(非同期処理を記述した関数);
```

```
//非同期処理を記述した関数
new Promise(成功時に実行する関数名, 失敗時に実行する関数名) => {
  //非同期の処理
};
```

- コード例

```
const p = new Promise<string>((resolve, reject) => {
  //管理したい非同期処理
  setTimeout(() => {
    console.log("サーバー呼び出し");
    resolve("成功"); //成功時の呼び出し
  }, 3000);
});
console.log(p); //結果：Promise{<pending>}
p.then((msg) => {
  console.log(p); //結果：Promise{'成功'}
  console.log(`${msg}しました。サーバーからデータを受け取った後に実行`);
});
```

- サンプルコード：4_am_samplecode_0-1.ts
- ログ結果

```
Promise { <pending> }
サーバー呼び出し
Promise { '成功' }
成功しました。サーバーからデータを受け取った後に実行
```

Promise型は状態管理なので、状態によって型が変わる。

Promiseの状態	概要
Promise<pending>	初期状態/実行中
Promise<fullfilled>	処理が成功して完了した状態(resolve関数が呼び出される)
Promise<rejected>	処理が失敗して完了した状態(reject関数が呼び出される)

> Promiseの型定義は実行完了後の値を定義する Promiseのthen()の中に、Promiseオブジェクトの処理が終わった時に呼び出してほしいコードを書く

then()で同期的に後続の非同期処理を続けたい

- 失敗のコード例

```
const p: Promise<string> = new Promise<string>((resolve, reject) => {
  setTimeout(() => {
    console.log("サーバー呼び出し");
    resolve("成功");
  }, 3000);
});
p.then((msg) => {
  console.log(`${msg}しました。サーバーからデータを受け取った後に実行`);
  setTimeout(() => {
    console.log("サーバー呼び出し2");
  }, 3000);
}).then(() => {
  console.log(`3番目の処理`);
});
```

- サンプルコード：4_am_samplecode_0-2.ts
- ログ結果

```
サーバー呼び出し
成功しました。サーバーからデータを受け取った後に実行
3番目の処理
サーバー呼び出し2
```

resolve()とthenはメソッドチェーンでつながっているがつないだ処理は非同期(処理を待たない)。同期的な処理になるPromiseをreturnしてつなげる必要がある。

> Promiseのreturnが切れていなければ同期的な処理となる

- 成功のコード例

```
function PromisFunc1() {
  return new Promise<string>((resolve, reject) => {
    //管理したい非同期処理
    setTimeout(() => {
      console.log("サーバー呼び出し");
      resolve("成功"); //成功時の呼び出し
    }, 3000);
  });
}

function PromisFunc2() {
  return new Promise<string>((resolve, reject) => {
    //管理したい非同期処理
    setTimeout(() => {
      console.log("サーバー呼び出し2");
      resolve("サーバー呼び出し2の成功"); //成功時の呼び出し
    }, 3000);
  });
}

function PromisFunc3() {
  return new Promise<string>((resolve, reject) => {
    //管理したい非同期処理
    setTimeout(() => {
      console.log("サーバー呼び出し3");
      resolve("サーバー呼び出し3の成功"); //成功時の呼び出し
    }, 1000);
  });
}

PromisFunc1()
  .then(() => {
    return PromisFunc2();
  })
  .then(() => {
    return PromisFunc3();
  });
```

- サンプルコード：4_am_samplecode_0-3.ts
- ログ結果

```
サーバー呼び出し;
サーバー呼び出し2;
サーバー呼び出し3;
```

- ポイント
  - `then()`のレスポンスも`Promise`なので、連続して後続処理も書くことができる
  - `then()`の中で`return`で返されたものが次の`then()`の入力になる
  - `Promise`の`then()`を見ると上から下へ処理が順序良く流れているように見える

値を受け取り非同期の後続処理へつなげる

- コード例

```
function PromisFunc1(msg: string, time: number) {
  return new Promise<string>((resolve, reject) => {
    //管理したい非同期処理
    setTimeout(() => {
      console.log(msg);
      resolve(`${msg}成功！`); //成功時の呼び出し
    }, time);
  });
}

function PromisFunc2(msg: string, time: number) {
  return new Promise<string>((resolve, reject) => {
    //管理したい非同期処理
    setTimeout(() => {
      console.log(msg);
      resolve(`${msg}→2回目の非同期成功`); //成功時の呼び出し
    }, time);
  });
}

function PromisFunc3(msg: string, time: number) {
  return new Promise<string>((resolve, reject) => {
    //管理したい非同期処理
    setTimeout(() => {
      console.log(msg);
      resolve(`${msg}→3回目の非同期成功`); //成功時の呼び出し
    }, time);
  });
}

PromisFunc1("1回目の非同期", 5000)
  .then((msg) => PromisFunc2(msg, 300))
  .then((msg) => PromisFunc3(msg, 1000));
```

- サンプルコード：4_am_samplecode_0-4.ts
- ログ結果

```
1回目の非同期
1回目の非同期成功！
1回目の非同期成功！→2回目の非同期成功
```

エラーのキャッチ

`Promise` 内での失敗時は`reject()`が呼び出される。**どういうときに成功か、どういうときに失敗かは開発者側で定義する**

- コード例

```
{
  function PromisFunc(msg: string, time: number) {
    return new Promise<void>((resolve, reject) => {
      //管理したい非同期処理
      setTimeout(() => {
        if (time < 6000) {
          console.log(msg);
          resolve(); //成功時の呼び出し
        } else {
          reject(`${msg}の非同期で失敗しました`);
        }
      }, time);
    });
  }

  PromisFunc("1回目の非同期", 2000)
    .then(() => PromisFunc("2回目の非同期", 300))
    .then(() => PromisFunc("3回目の非同期", 3000))
    .then(() => PromisFunc("4回目の非同期", 6000))
    .then(() => PromisFunc("5回目の非同期", 1000))
    .catch((err) => console.log(err));
}
```

- サンプルコード：4_am_samplecode_0-5.ts
- ログ結果

```
1回目の非同期
2回目の非同期
3回目の非同期
4回目の非同期の非同期で失敗しました
```

- ポイント
  - `resolve()`の時は`then`へ処理が進み、`reject()`の時は`catch`に進む。

5.async/await

Promiseもthenの中身が複雑になる可能性もある。さらなる改善のためawaitという新たなキーワードが導入された。これは、Promiseを使ったコードのthen()の中だけをならべたものとほぼ等価(糖衣構文)。よりシンプルに書くことができる。

asyncは非同期の関数という意味

- ポイント
  - async関数はreturn でresolveと同じ意味
  - 例外や値をthrowしたらrejectと同じ意味
  - Promiseをreturnするとそのままを返す

例外をthrowしてエラーハンドリングする場合

- コード例

```
async function asyncFunc(num: number): Promise<void | string> {
  //非同期処理
  if (num % 10 === 0) {
    return "成功";
  } else {
    throw new Error("非同期処理に失敗しました");
  }
}
asyncFunc(80)
  .then((msg) => console.log(msg))
  .catch((err) => console.log(err.message));
```

- サンプルコード：4_am_samplecode_1-1.ts
- ログ結果

```
//asyncFunc(80)
成功;
//asyncFunc(85)
非同期処理に失敗しました;
```

rejectでエラーハンドリングする場合

- コード例

```
function promise(time: number, msg: string) {
  return new Promise<string>((resolve, reject) => {
    setTimeout(() => {
      if (time < 6000) {
        console.log("時間のかかる処理");
        resolve("成功");
      } else {
        reject(`${msg}が失敗しました。`);
      }
    }, 3000);
  });
}
async function async() {
  //そのままPromiseを返す
  return await promise(5000, "1回目の非同期処理");
}

async()
  .then((msg) => console.log(msg))
  .catch((err) => console.log(err));
```

- サンプルコード：4_am_samplecode_1-2.ts
- ログ結果

```
//promise(5000, "1回目の非同期処理")
時間のかかる処理;
成功;
//promise(8000, "1回目の非同期処理")
1回目の非同期処理が失敗しました。
```

- ポイント
  - asyncとawaitはセットで使うこと。
  - awaitをつけることでその処理が終わるまで次の処理は実行されない状態となる

awaitで同期的に処理をおこない、最後にthenとcatchで処理をつなげる

- コード例

```
function promise(time: number, msg: string) {
  return new Promise<string>((resolve, reject) => {
    setTimeout(() => {
      if (time < 6000) {
        console.log("時間のかかる処理");
        resolve("成功");
      } else {
        reject(`${msg}が失敗しました。`);
      }
    }, 3000);
  });
}
async function async() {
  await promise(5000, "1回目の非同期処理");
  await promise(3000, "2回目の非同期処理");
  await promise(2000, "3回目の非同期処理");
}

//thenはawaitの処理が全て終わってから実行されるためresolveでメッセージを渡すことができない
async()
  .then((msg) => console.log(msg))
  .catch((err) => console.log(err));
```

- サンプルコード：4_am_samplecode_2-1.ts
- ログ結果

```
//await promise(5000, "1回目の非同期処理");
//await promise(3000, "2回目の非同期処理");
//await promise(2000, "3回目の非同期処理");
時間のかかる処理
時間のかかる処理
時間のかかる処理
undefined

//await promise(5000, "1回目の非同期処理");
//await promise(8000, "2回目の非同期処理");
//await promise(2000, "3回目の非同期処理");
時間のかかる処理
2回目の非同期処理が失敗しました。
```

- ポイント
  - **thenはawaitの処理が全て終わってから実行**されるためresolveの引数を渡すことができない

await部分をリレー方式で値を渡しながら同期的な処理を行う

- コード例

```
function promise(time: number, msg: string) {
  return new Promise<string>((resolve, reject) => {
    setTimeout(() => {
      if (time < 6000) {
        console.log(msg);
        resolve(`${msg}成功`);
      } else {
        reject(`${msg}が失敗しました。`);
      }
    }, 3000);
  });
}
async function async() {
  const result1: string = await promise(5000, "1回目の非同期処理");
  const result2: string = await promise(4000, `${result1}→2回目の非同期処理`);
  const result3: string = await promise(2000, `${result1}→3回目の非同期処理`);
}

async()
  .then(() => console.log("全ての非同期処理が完了しました"))
  .catch((err) => console.log(err));
```

- サンプルコード：4_am_samplecode_2-2.ts
- ログ結果

```
// const result1: string = await promise(5000, "1回目の非同期処理");
// const result2: string = await promise(4000, `${result1}→2回目の非同期処理
`);
// const result3: string = await promise(2000, `${result1}→3回目の非同期処理
`);
1回目の非同期処理
1回目の非同期処理成功→2回目の非同期処理
1回目の非同期処理成功→3回目の非同期処理
全ての非同期処理が完了しました

// const result1: string = await promise(5000, "1回目の非同期処理");
// const result2: string = await promise(8000, `${result1}→2回目の非同期処理
`);
// const result3: string = await promise(2000, `${result1}→3回目の非同期処理
`);
1回目の非同期処理
1回目の非同期処理成功→2回目の非同期処理が失敗しました。
```

async/awaitでのエラーハンドリング

Promise関数のresolve/reject以外の自分で書いているasync/await部分については自分でエラーハンドリングをする必要がある。

- コードの概要

```
try {
  //例外が発生する可能性のある処理
  if (false) {
    throw new Error(エラーメッセージ);
  }
} catch (エラーを受け取る変数) {
  //例外発生時の処理
}
```

- コード例

```
function promise(time: number, msg: string) {
  return new Promise<string>((resolve, reject) => {
    setTimeout(() => {
      if (time < 6000) {
        console.log(msg);
        resolve(`${msg}成功`);
      } else {
        reject(`${msg}が失敗しました。`);
      }
    }, 3000);
  });
}
async function async(num: number) {
  try {
    if (num % 2 === 0) {
      const result1: string = await promise(5000, "1回目の非同期処理");
      const result2: string = await promise(4000, `${result1}→2回目の非同期処
理`);
      const result3: string = await promise(2000, `${result1}→3回目の非同期処
理`);
    } else {
      throw new Error("渡されてきた引数が2で割り切れません");
    }
  } catch (error: unknown) {
    if (error instanceof Error) {
      console.log(error.message);
    }
  }
}

async(35)
  .then(() => console.log("全ての非同期処理が完了しました"))
  .catch((err) => console.log(err));
```

- サンプルコード：4_am_samplecode_2-3.ts
- ログ結果

```
渡されてきた引数が2で割り切れません;
全ての非同期処理が完了しました;
```