

# async/await 入門 (JavaScript)

JavaScript 入門 es2017 AsyncAwait

## はじめに

---

今更ですが、JavaScriptの `async / await` に関する備忘録になります。

- 「今まで `$.Deferred()` や `Promise` などで非同期処理は書いたことがあるが、`async / await` はわからない」
- 「`$.Deferred()` や `Promise` などの非同期処理の書き方より、もっと簡潔に書ける書き方があれば知りたい」
- 「今までの非同期処理の書き方と比べて何が良いのかわからない」

といった人達向けの記事です。

`$.Deferred()` や `Promise` などで非同期処理を書いたことがある前提のため、非同期処理自体に関する説明は記載しておりません。

記載している利用例のコードはChrome（最新）のコンソール

上で動きますので、コンソール上で実行して動作を確認してみると理解が深まりやすいと思います。

## 本記事で用いている用語

---

### Promise を返す

Promise オブジェクトを返すこと。

```
// Promiseを返す
return new Promise((resolve, reject) => {

});
```

### Promise の結果を返す

Promise の resolve もしくは reject を実行すること。

```
return new Promise((resolve, reject) => {  
  // Promiseの結果を返す  
  resolve('resolve!!');  
});
```

## resolve する

Promise の resolve を実行すること。

```
return new Promise((resolve, reject) => {  
  // succes!!をresolveする  
  resolve('succes!!');  
});
```

## reject する

Promise の reject を実行すること。

```
return new Promise((resolve, reject) => {  
  // err!!をrejectする
```

```
reject('err!!');  
});
```

## async / await とは

---

async と await を利用した、非同期処理の構文のこと。

## 何故 async / await を利用するのか

---

Promise を利用した構文よりも、簡潔に非同期処理が書けるから。

## async / await の対応状況

---

以下は各ブラウザの async / await の対応状況。

# ECMAScript 2016+ compatibility table

		Compilers/polyfills						Desktop browsers															
		83%	10%	40%	16%	36%	16%	3%	42%	76%	86%	79%	89%	100%	100%	94%	94%	94%	41%	94%	100%	100%	100%
Feature name		Current browser	Traceur	Babel + core-js <sup>[2]</sup>	Closure	TypeScript + core-js	es7-shim	IE 11	Edge 14	Edge 15	Edge 16 Preview	FF 52 ESR	FF 54	FF 55 Beta	FF 56 Nightly	CH 60, OP 47 <sup>[1]</sup>	CH 61, OP 48 <sup>[1]</sup>	CH 62, OP 49 <sup>[1]</sup>	SF 10	SF 10.1	SF 11	SF TP	WK
async functions 		15/15	3/15	3/15	3/15	2/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	0/15	15/15	15/15	15/15	15/15
 jrn		Yes	Yes	Yes <sup>[13]</sup>	Yes	Yes <sup>[14]</sup>	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
throw		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
no line break between async and function		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
no "prototype" property		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
await 		Yes	Yes	Yes <sup>[13]</sup>	Yes	Yes <sup>[14]</sup>	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
await, rejection		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
must await a value		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
can await non-Promise values		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
cannot await in parameters		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
async methods, object literals		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
async methods, classes		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
async arrow functions		Yes	Yes	Yes <sup>[13]</sup>	Yes	No	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
correct prototype chain		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
async function prototype, Symbol.toStringTag		Yes	?	?	?	?	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
async function constructor		Yes	?	?	?	?	No	No	Flag <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
shared memory and atomics		0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	0/17	17/17	0/17	0/17	17/17	17/17	17/17	17/17	17/17	0/17	16/17	17/17	17/17	17/17

全てのブラウザで対応しているわけではないため、利用する  
ならBabel等でトランスパイルする必要がある。

## async とは

非同期関数を定義する関数宣言のこと。

以下のように関数の前に `async` を宣言することにより、非同期関数（`async function`）を定義できる。

```
async function sample() {}
```

# async function（async で宣言した関数）は何をするのか

---

- `async function` は呼び出されると `Promise` を返す。
- `async function` が値を `return` した場合、`Promise` は戻り値を `resolve` する。
- `async function` が例外や何らかの値を `throw` した場合はその値を `reject` する。

言葉だけだとわかりづらいため、利用例を見てみる。

## async function の利用例

---

以下は `async function` が `Promise` を返し、値を `resolve`、もしくは `reject` しているか確認するための利用例。

※このように利用することはほとんどないと思いますが、`async function` がどのような動きをしているのかを確認するために記載しております。

// resolve!!をreturnしているため、この値がresolveされる

```
async function resolveSample() {  
    return 'resolve!!';  
}
```

// resolveSampleがPromiseを返し、resolve!!がresolveされるため

// then()が実行されコンソールにresolve!!が表示される

```
resolveSample().then(value => {  
    console.log(value); // => resolve!!  
});
```

// reject!!をthrowしているため、この値がrejectされる

```
async function rejectSample() {  
    throw new Error('reject!!');  
}
```

// rejectSampleがPromiseを返し、reject!!がrejectされるため


// catch()が実行されコンソールにreject!!が表示される

```
rejectSample().catch(err => {  
    console.log(err); // => reject!!  
});
```

// resolveErrorはasync functionではないため、Promiseを返さない

```
function resolveError() {  
    return 'resolveError!!';  
}
```

```
// resolveErrorはPromiseを返さないため、エラーが発生して動かない
// Uncaught TypeError: resolveError(...).then is not a function
resolveError().then(value => {
  console.log(value);
});
```



上記の通り、 async function が Promise を返し、値を resolve 、もしくは reject していることがわかった。上記は async function 単体の利用例だが、 await と併用して利用することが多く、「 async を利用するなら await も必ず利用すべき」と書かれている記事もあった。

## await とは

---

async function 内で Promise の結果（ resolve 、 reject ）が返されるまで待機する（処理を一時停止する）演算子のこと。

以下のように、関数の前に await を指定すると、その関数の Promise の結果が返されるまで待機する。



```
async function sample() {  
    const result = await sampleResolve();  
  
    // sampleResolve()のPromiseの結果が返ってくるまで以下は実行され  
    console.log(result);  
}
```

## await は何をするのか

---

- await を指定した関数の Promise の結果が返されるまで、async function 内の処理を一時停止する。
- 結果が返されたら async function 内の処理を再開する。

await は async function 内でないと利用できないため、async / await の利用例を見ていく。

## async / await の利用例

---

以下は単純な `async / await` の利用例。

```
function sampleResolve(value) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(value * 2);  
    }, 2000);  
  })  
}
```

```
/**
```

```
 * sampleResolve()をawaitしているため、Promiseの結果が返されるまで  
 * 今回の場合、2秒後にresolve(10)が返ってきてその後の処理 (return r  
 * resultにはresolveされた10が格納されているため、result + 5 = 15が  
 */
```

```
async function sample() {  
  const result = await sampleResolve(5);  
  return result + 5;  
}
```

```
sample().then(result => {  
  console.log(result); // => 15  
});
```

上記の処理を `Promise` の構文で書くと以下のようなになる。

```
function sampleResolve(value) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(value * 2);  
    }, 2000);  
  })  
}  
  
function sample() {  
  return sampleResolve(5).then(result => {  
    return result + 5;  
  });  
}  
  
sample().then(result => {  
  console.log(result); // => 15  
});
```

2つを見比べてみると、 `async / await` の方が簡潔に書けることがわかる。

より複雑な処理の場合でも `async / await` を利用した方が簡潔に書けるため、いくつか例を紹介していく。

## 連続した非同期処理

---

# 連続した非同期処理 ( Promise 構文)

```
function sampleResolve(value) {  
    return new Promise(resolve => {  
        setTimeout(() => {  
            resolve(value);  
        }, 1000);  
    })  
}
```

```
function sample() {  
    let result = 0;  
  
    return sampleResolve(5)  
        .then(val => {  
            result += val;  
            return sampleResolve(10);  
        })  
        .then(val => {  
            result *= val;  
            return sampleResolve(20);  
        })  
        .then(val => {  
            result += val;  
            return result;  
        });  
}
```

```
sample().then((v) => {  
    console.log(v); // => 70  
});
```

## 連続した非同期処理（ async / await 構文）

await を利用すれば、 then() で処理を繋げなくても連続した非同期処理が書ける。

```
function sampleResolve(value) {  
    return new Promise(resolve => {  
        setTimeout(() => {  
            resolve(value);  
        }, 1000);  
    })  
}
```

```
async function sample() {  
    return await sampleResolve(5) * await sampleResolve(10) + 1;  
}
```

```
async function sample2() {
```

```
const a = await sampleResolve(5);
const b = await sampleResolve(10);
const c = await sampleResolve(20);
return a * b + c;
}
```

```
sample().then((v) => {
  console.log(v); // => 70
});
```

```
sample2().then((v) => {
  console.log(v); // => 70
});
```

for を利用した繰り返しの非同期処理も書ける。

```
function sampleResolve(value) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(value);
    }, 1000);
  })
}
```

```
async function sample() {
  for (let i = 0; i < 5; i += 1) {
    const result = await sampleResolve(i);
  }
}
```

```
        console.log(result);
    }

    return 'ループ終わった。'
}

sample().then((v) => {
    console.log(v); // => 0
                  // => 1
                  // => 2
                  // => 3
                  // => 4
                  // => ループ終わった。
});
```

array.reduce() も利用できる。

```
function sampleResolve(value) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(value);
        }, 1000);
    })
}
```

```
async function sample() {
    const array = [5, 10, 20];
```

```
const sum = await array.reduce(async (sum, value) => {
  return await sum + await sampleResolve(value) * 2;
}, 0);

return sum;
}

sample().then((v) => {
  console.log(v); // => 70
});
```

連続の非同期処理は、**処理を順番に行う必要がない限り利用すべきではない**ため注意。

例えば、画像を非同期読み込みをする場合、上記のような処理だと**1つの画像を読み込みが完了するまで、次の画像の読み込みが始まらない**。

そのため、全ての画像の読み込みにかなりの時間がかかる。画像は連続して読み込む必要はないため、後述する並列の非同期処理で読み込むべき。

## 並列の非同期処理



# 並列の非同期処理 ( Promise 構文)

```
function sampleResolve(value) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(value);  
    }, 2000);  
  })  
}
```

```
function sampleResolve2(value) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(value * 2);  
    }, 1000);  
  })  
}
```

```
function sample() {  
  const promiseA = sampleResolve(5);  
  const promiseB = sampleResolve(10);  
  const promiseC = promiseB.then(value => {  
    return sampleResolve2(value);  
  });  
  
  return Promise.all([promiseA, promiseB, promiseC])  
    .then(([a, b, c]) => {
```

```
        return [a, b, c];
    });
}

sample().then(([a, b, c]) => {
    console.log(a, b, c); // => 5 10 20
});
```

## 並列の非同期処理（ async / await 構文）

Promise.all にも await を利用できるため、以下のように記述できる。

```
function sampleResolve(value) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(value);
        }, 2000);
    })
}
```

```
function sampleResolve2(value) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(value * 2);
        }, 2000);
    })
}
```

```
    }, 1000);  
  })  
}
```

```
async function sample() {  
  const [a, b] = await Promise.all([sampleResolve(5), sampleResolve(10)]);  
  const c = await sampleResolve2(b);  
  
  return [a, b, c];  
}
```

```
sample().then(([a, b, c]) => {  
  console.log(a, b, c); // => 5 10 20  
});
```

array.map() も利用できる。

```
function sampleResolve(value) {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve(value);  
    }, 2000);  
  })  
}
```

```
async function sample() {  
  const array =[5, 10, 15];  
  const results = await Promise.all(array.map(value => sampleResolve(value)));  
  return results;  
}
```

```
const promiseAll = await Promise.all(array.map(async (value) => {
  return await sampleResolve(value) * 2;
}));

return promiseAll;
}

sample().then(([a, b, c]) => {
  console.log(a, b, c); // => 10 20 30
});
```

## 例外処理（エラーハンドリング）

---

### 例外処理（エラーハンドリング） Promise 構文）

```
function throwError() {
  return new Promise((resolve, reject) => {
    setTimeout(function() {
      try {
        throw new Error('エラーあったよ');
      } catch {
        resolve('エラーなかった');
      }
    }, 1000);
  });
}
```

```
        } catch(err) {  
            reject(err);  
        }  
    }, 1000);  
});  
}
```

```
function errorHandler() {  
    return throwError()  
        .then((result) => {  
            return result;  
        })  
        .catch((err) => {  
            throw err;  
        });  
}
```

```
errorHandler().catch((err) => {  
    console.log(err); // => エラーあったよ  
});
```

## 例外処理 (エラーハンドリング) async / await 構文)

await を利用すれば、非同期処理の try catch を書ける。

```
function throwError() {  
  return new Promise((resolve, reject) => {  
    setTimeout(function() {  
      try {  
        throw new Error('エラーあったよ')  
        resolve('エラーなかった');  
      } catch(err) {  
        reject(err);  
      }  
    }, 1000);  
  });  
}
```

```
async function errorHandler() {  
  try {  
    const result = await throwError();  
    return result;  
  } catch (err) {  
    throw err;  
  }  
}
```

```
errorHandler().catch((err) => {  
  console.log(err); // => エラーあったよ  
});
```

実は try catch で囲まなくても上記のコードと同様に動く。

```
function throwError() {  
  return new Promise((resolve, reject) => {  
    setTimeout(function() {  
      try {  
        throw new Error('エラーあったよ')  
        resolve('エラーなかった')  
      } catch(err) {  
        reject(err);  
      }  
    }, 1000);  
  });  
}
```

```
async function errorHandling() {  
  // throwErrorがrejectを返したら処理が中断される  
  const result = await throwError();  
  return result;  
}
```

```
errorHandling().catch((err) => {  
  console.log(err); // => エラーあったよ  
});
```

# 終わり

---

フロントエンド実装で非同期処理は避けられないため、より簡潔に書ける `async / await` を覚えておいて損はないと思います。

とはいえ `async / await` も `Promise` を利用しているため、`Promise` 自体の理解は必須です。

理解を深めるためにも以下の一読をオススメします。

- [JavaScript Promiseの本](#)
- [Promiseとasync-awaitの例外処理を完全に理解しよう](#)

また、連続した非同期処理（`for` 文などの中で `await`）を利用するうえでの注意点に関しては、以下を一読すればさらに理解が深まるかと思います。

- [async/await地獄](#)（単純に「使いどころを気をつけよう」という記事です。）

## お知らせ

---



Udemy で webpack の講座を公開したり、Kindle で技術書を出版しています。

Udemy:

webpack 最速入門 (~~10,800 円~~ -> 2,000 円)

Kindle (Kindle Unlimited だったら無料) :

React 実践入門 (500 円)

React Hooks 入門 (500 円)

興味を持ってくださった方はご購入いただけると大変嬉しいです。よろしくお願いいたします。