例外処理

TypeScriptはJavaと似たような例外処理機構を備えています。ただし、ベースとなっているJavaScriptの言語の制約から、使い勝手などは多少異なります。

TypeScriptの例外処理構文

A → B → Cと順番にタスクをこなすプログラムがあったとします。例えば、データを取得してきて、それを加工して、他のサーバーに送信するバッチ処理のプログラムとかを想像してください。 これが順番通りうまくいけば何も問題がありませんが、例えば、データ取得時や送信時にネットワークにうまく繋がらない、加工しようと思ったが、サーバーから送られてきたデータが想定と違ったなど、うまくいかないこともありえます。その場合に、処理を中断する（例外を投げる）、中断したことを察知して何かしらの対処をする（回復処理）を行います。これらの機構をまとめて例外処理と呼んだりします。

`throw` を使って例外を投げます。 `throw` すると、その行でその関数やメソッド内部の処理は中断し、呼び出し元、そのさらに呼び出し元、と処理がどんどん巻き戻っていきます。最終的に回復処理を行う `try` 節/ `catch` 節のペアにあたるまで巻き戻ります。

```
throw new Error("ネットワークアクセス失敗");

console.log("この行は実行されない");
```

例外を投げうる処理の周りは `try` 節で囲みます。 `catch` 節は例外が飛んできたときに呼ばれるコードブロックです。例外が発生してもしなくても、必ず通るのが `finally` 節です。後片付けの処理を書いたりします。 `finally` は省略できます。

```
try {
  const data = await getData();
  const modified = modify(data);
  await sendData(modified);
} catch (e) {
  console.log(`エラー発生 ${e}`);
} finally {
  // 最後に必ず呼ばれる
}
```

もし、回復処理で回復し切れない場合は、再度例外を投げることもできます。

例外の再送

```
try {
  //
} catch (e) {
  throw e; // 再度投げる
}
```

Javaの例外と似ていると最初に紹介しましたが、Javaと異なるのが、ベースのJavaScriptのコードは型情報をソースコード上に持っていないという点があります。 Javaの場合は、例外の `catch` 節を複数持つことができ、それぞれの節に例外のクラスの種類を書いておくと、飛んできた例外の種類に応じて適切な節が選択されます。 JavaScriptでは1つしか書くことができません。型の種類による分岐というのも、 `catch` 節の中で `if` 文を使って行う必要があります。

例外クラスを自分で作る必要はありますが、Javaと同じことを実現するには、以下のようなコードになります。

Javaと似たような例外の分岐

```
try {
  // 何かしらの処理
} catch (e) {
  // instanceofを使ってエラーの種類を判別していく
  if (e instanceof NoNetworkError) {
    // NoNetworkErrorの場合
  } else if (e instanceof NetworkAccessError) {
    // NetworkAccessErrorの場合
  } else {
    // その他の場合
  }
}
```

`Error` クラス

例外処理で「問題が発生した」ときに情報伝達に使うのが `Error` クラスです。 さきほどの構文は `new` と同時に `throw` していましたが、ふつうのオブジェクトです。

`Error` クラスは作成時にメッセージの文字列を受け取れます。 `name` 属性にはクラス名、 `message` には作成時にコンストラクタに渡した文字列が格納されます。

JavaScriptの言語の標準に含まれていない、処理系独自の機能（といっても、今のところ全ブラウザで使える）のが、 `stack` プロパティに格納されるスタックトレースです。 `throw` した関数が今までどのように呼ばれてきたかの履歴です。ファイル名や行数も書かれていたりします。これがなければTypeScriptのデバッグは数万倍困難だったでしょう。

なお、この行数はTypeScriptをコンパイルした後のJavaScriptのファイル名と行番号だったりしますが、ソースマップというファイルをコンパイル時に出力しておき、実行時にそれがうまく読み込めると（ブラウザなら一緒にアップロード、Node.jsはnpmの `source-map-support` パッケージを利用すれば、もともとのTypeScriptのファイル名と行番号で出力されるようになります。

```
const e = new Error('エラー発生');
console.log(`name: ${e.name}`);
// name: Error
console.log(`message: ${e.message}`);
// message: エラー発生
console.log(`stack: ${e.stack}`);
// stack: Error: test
//    at new MyError (<anonymous>:17:23)
//    at <anonymous>:23:9
```

標準の例外クラス

それ以外にも、いろいろな例外のためのクラスがあります。TypeScriptを使っているとコンパイル前に多くの問題を潰せるため、遭遇する回数はJavaScriptよりも減ります。

- `EvalError`
- `RangeError`
- `ReferenceError`
- `SyntaxError`
- `TypeError`
- `URIError`

例外を受け取って何もしない（俗称：例外を握りつぶす）は行儀がよくないコードとされますが、JSONパース時には文法がおかしい場合に `SyntaxError` が発生します。 `JSON.parse()` だけは拾って無効値で初期化という処理は頻繁に行うでしょう。

```
let json: any;
try {
  json = JSON.parse(jsonString);
} catch (e) {
  json = null;
}
```

あとはブラウザの `fetch()` 関数でサーバー側のAPIにアクセスするときに、ネットワークエラー（corsでの権限がない場合も）は `TypeError` が発生します。 `fetch` はJSONをパースする場合に `SyntaxError` も発生します。

```
try {
  const res = await fetch("/api/users"); // ここでTypeError発生の可能性
  if (res.ok) {
    const json = await res.json();         // ここでSyntaxError発生の可能性
  }
}
```

よくやりがちなのが、`ok`の確認をしない（ステータスコードが200以外でJSON以外が帰ってきているときに）JSONをパースしようとしてエラーになることです。`404 Not Found`のときは、ボディが`Not Found`というテキストになるので、未知のトークン`N`というエラーになります。あとは403 Forbiddenのときには、未知のトークン`F`のエラーが発生します。

```
SyntaxError: Unexpected token N in JSON at position 0
```

例外処理とコードの読みやすさ

例外処理も、コードを読む人の理解を手助けするための、ちょっとしたコツがあります。

`try`節はなるべく狭くする

「この関数を呼ぶと、AとBの例外が飛んでくる可能性がある」というのはできあがったソースコードを見ても情報はわかりません。次のコード例を見ても、AからEのどこでどんな例外が飛んでくるかわからないでしょう。

広すぎる*try*は例外の出どころをわかりにくくする

```
try {
  logicA();
  logicB();
  logicC();
  logicD();
  logicE();
} catch (e) {
  // エラー処理
}
```

なるべく狭くすることで、どの処理がどの例外を投げるのかが明確になります。

*try*の範囲を狭めると、どこで何がおきるのかがわかりやすくなる

```
logicA();
logicB();
try {
  logicC();
} catch (e) {
  // エラー処理
}
logicD();
logicE();
```

実際に実行時に例外が起きうる（どんなにデバッグしても例外を抑制できない）ポイントは、外部の通信とかごく一部のはずです。あまりたくさん例外処理を書く必要もないと思いますし、書く場合もどこに書いたかがわかりやすくなります。

広くする問題としては、原因の違う例外が混ざってしまう点もあります。例えば、JSONのパースを何箇所かで行なっていると、それぞれの箇所で `SyntaxError` が投げられる可能性が出てきます。原因が違ってリカバリー処理が別の例外が同じ `catch` 節に入ってきてしまうと、正確に例外を仕分けを行って漏れなく対応する必要が出てきます。しかし、どの例外が投げられるかを明示できる文法がなく、実装のコードを見ないとどのような例外が投げられてくるか分かりません。そのため、「間違いなく対処できているか？」を判断するコストが極めて高くなります。

`Error` 以外を `throw` しない

前述の `catch` 文のサンプルでは、 `e` の型が `Error` という前提で書いていました。これにより、 `catch` 節の中でコード補完がきくので、開発はしやすくなります。しかし、実際には、どの型がくるかは実行時の `throw` 次第です。 `throw` には `Error` 関連のクラス以外にも、文字列とか数値とかなんでも投げることができるからです。

基本的には `Error` 関連のオブジェクトだけを `throw` するようにしましょう。

```
try {
  :
} catch (e) {
  // e.とタイプすると、name, messageなどがサジェストされる
  console.log(e.name);
}
```

リカバリー処理の分岐のためにユーザー定義の例外クラスを作る

例外処理のためにクラスを作ってみましょう。 `Error` を継承することで、例外クラスを作ることができます。ただし、少し `Error` クラスは特殊なので、いくつかの追加処理をコンストラクタで行う必要があります。5個例外クラスを作るとして、全部のクラスで同じ処理を書くこともできま

す。しかし、これが10個とか20個になると大変です。1つのベースのクラスを作り、実際にコード中で扱うクラスはこれから継承して作るようにします。

```
// 共通エラークラス
class BaseError extends Error {
  constructor(e?: string) {
    super(e);
    this.name = new.target.name;
    // 下記の行はTypeScriptの出力ターゲットがES2015より古い場合(ES3, ES5)のみ必要
    Object.setPrototypeOf(this, new.target.prototype);
  }
}

// BaseErrorを継承して、新しいエラーを作る
// statusCode属性にHTTPのステータスコードが格納できるように
class NetworkAccessError extends BaseError {
  constructor(public statusCode: number, e?: string) {
    super(e);
  }
}

// 追加の属性がなければ、コンストラクタも定義不要
class NoNetworkError extends BaseError {}
```

このようにクラスをいくつも作ると、例外を受け取った `catch` 節で、リカバリーの方法を「選ぶ」ことが可能になります。 投げられたクラスごとに `instanceof` と組み合わせて条件分岐に使えます。また、この `instanceof` は型ガードになっていますので、各ブロックの中でコード補完も正しく行われます。 上記のクラスの `statusCode` も正しく補完されます。

```
try {
  await getUser();
} catch (e) {
  if (e instanceof NoNetworkError) {
    alert("ネットワークがありません");
  } else if (e instanceof NetworkAccessError) {
    // この節では、eはNetworkAccessErrorのインスタンスなので、
    // ↓のe.をタイプすると、statusCodeがサジェストされる
    if (e.statusCode < 500) {
      alert("プログラムにバグがあります");
    } else {
      alert("サーバーエラー");
    }
  }
}
```

なお、TypeScriptは、昔のJavaのように継承を前提とした処理を書くことはほとんどありませんので、コードの中で継承を使うことも極めてまれです。Javaの場合は、 `IOException` クラスを継承したクラスがあって、入出力系のエラーなど継承階層を前提としたコードが書かれたりもしました。しかし、これは「AはBの子クラスである」という知識を持っていないと読めないコードにな

ってしまうため、プロジェクトに入ってきたばかりの人には混乱を与えがちです。例外クラスを作る場合も、`BaseClass`からの直系の子供クラスだけで作れば問題ありません。立派な継承ツリーの設計は不要です。あまり例外クラスが多くても使い分けに迷ったりします。

注釈

ターゲットがES3/ES5のときに`Object.setPrototypeOf(this, new.target.prototype);`の行を書き忘れると、`instanceof`が`false`を返してくるようになります。

例外処理を使わないエラー処理

正常に実行できなかったからといって、なんでも例外として処理しなければならないわけではありません。 例えば、ブラウザ標準の`fetch` APIの場合、通信ができたが、正常に終わらなかった場合は`ok`属性を使って判断できます。 例外には深い階層から一発で離脱できる（途中の関数では、エラーがあったかどうかを判定不要）メリットがあります。 しかし、階層が深くなく、呼び出し元と例外処理を行うコードがすごく近い場合には、この`ok`のような属性を用意する方が管理もしやすいでしょう。

```
const res = await fetch("/users");
if (res.ok) {
  // ステータスコードが200/300番台
} else {
  // 400番以降
}
```

非同期と例外処理

非同期処理で難しいのがエラー処理でした。`async`と`await`のおかげで例外処理もだいぶ書きやすくなりました。

`Promise`では`then()`の2つめのコールバック関数でエラー処理が書けるようになりました。 また、エラー処理の節だけを書く`catch()`節もあります。 複数の`then()`節が連なっていても、1箇所だけエラー処理を書けば大丈夫です。 なお、一箇所もエラー処理を書かずにいて、エラーが発生すると`unhandledRejection`というエラーがNode.jsのコンソールに表示されることになります。

Promiseのエラー書き方

```
fetch(url).then(resp => {
  return resp.json();
}).then(json => {
  console.log(json);
}).catch(e => {
  console.log("エラー発生!");
  console.log(e);
});
```

`async` 関数の場合はもっとシンプルで、何かしらの非同期処理を実行する場合、 `await` していれば、通常の `try` 文でエラーを捕まえることができます。

*async*関数内部のエラー処理の書き方

```
try {
  const resp = await fetch(url);
  const json = await resp.json();
  console.log(json);
} catch (e) {
  console.log("エラー発生!");
  console.log(e);
}
```

エラーを発生させるには、 `Promise` 作成時のコールバック関数の2つめの引数の `reject()` コールバック関数に `Error` オブジェクトを渡しても良いですし、 `then()` 節の中で例外をスローしても発生させることができます。

```
async function heavyTask() {
  return new Promise<number>((resolve, reject) => {
    // 何かしらの処理
    reject(error);
    // こちらでもPromiseのエラーを発生可能
    throw new Error();
  });
};
```

`Promise` 以前は非同期処理の場合は、コールバック関数の先頭の引数がエラー、という暗黙のルールで実装されていました。 ただし、1つのコールバックでも `return` を忘れると動作しませんし、通常の例外が発生して `return` されなかったりすると、コールバックの伝搬が中断されてしまいます。

原始時代の非同期のエラー処理の書き方

```
// 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
    });
  });
});
```

例外とエラーの違い

この手の話になると、エラーと例外の違いとか、こっちはハンドリングするもの、こっちはOSにそのまま流すものとかいろんな議論が出てきます。例外とエラーの違いについても、コンセンサスは取れておらず、人によって意味が違ったりします。一例としては、回復可能なものがエラーで、そうじゃないものが例外といったことが言われたりします。このエントリーではエラーも例外も差をつけずに、全部例外とひっくるめて説明します。

例外というのはすべて、何かしらのリカバリーを考える必要があります。

- ちょっとしたネットワークのエラーなので、3回ぐらいはリトライしてみる
    - 原因: ネットワークエラー
    - リカバリー: リトライ
- サーバーにリクエストを送ってみたら400エラーが帰ってきた
    - 原因: リクエストが不正
    - リカバリー(開発時): 本来のクライアントのロジックであればバリデーションで弾いていないといけないのでこれは潰さないといけない実装バグ。とりあえずスタックトレースとかありったけの情報をconsole.logに出しておく。
    - リカバリー(本番): ありえないバグが出た、とりあえず中途半端に継続するのではなくて、システムエラー、開発者に連絡してくれ、というメッセージをユーザーに出す（人カリカバリー）
- JSONをパースしたら `SyntaxError`
    - 原因: ユーザーの入力が不正
    - リカバリー: フォームにエラーメッセージを出す

最終的には、実装ミスなのか、ユーザーが間違ってデータ入力したという実行時の値の不正なのか、ネットワークの接続がおかしい、クラウドサービスの秘密鍵が合わないみたいな環境の問題なのか、どれであったとしても、システムが自力でリカバリーする、ユーザーに通知して入力修正やWiFiのある環境で再実行などの人カリカバリーしてもらう、開発者に通知してプログラム修正するといった人カリカバリーなど、何かしらのリカバリーは絶対必要になります。

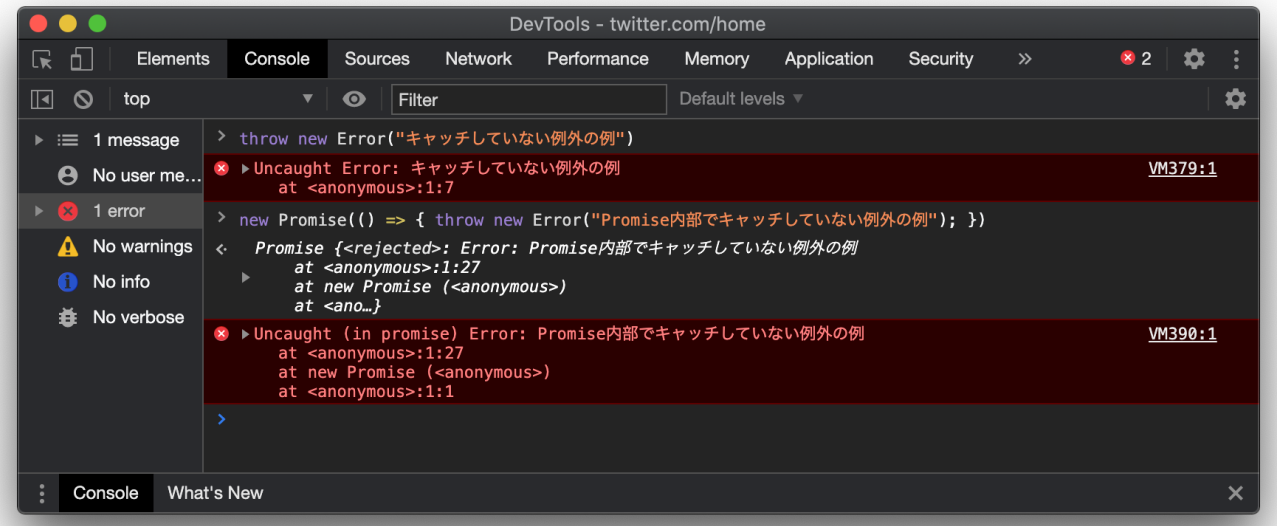
Node.jsで `async` / `await` やら `Promise` を一切使っていないコードの場合、エラーを無視すると、Node.js自体が最後に `catch` して、エラー詳細を表示してプログラムが終了します。これはある意味プログラムとしては作戦放棄ではありますが、「プログラムの進行が不可能なので、OSに処理を返す」というリカバリーと言えなくもないでしょう。開発者にスタックトレースを表示して後を託す、というのも立派なリカバリーの戦術の1つです。

ブラウザの場合、誰もキャッチしないと、開発者ツールのコンソールに表示されますが、開発者ツールを開いていない限りエラーを見ることはできません。普段から開発者コンソールを開いている人はまれだと思いますし、大部分のユーザーには正常に正常に処理が進んだのか、そうでなかったのかわかりませんので、かならずキャッチして画面に表示してあげる必要があるでしょう。

どちらにしても何かしらのリカバリー処理が必要となりますので、本書ではエラーと例外の区別といったことはしません。

例外処理のハンドリングの漏れ

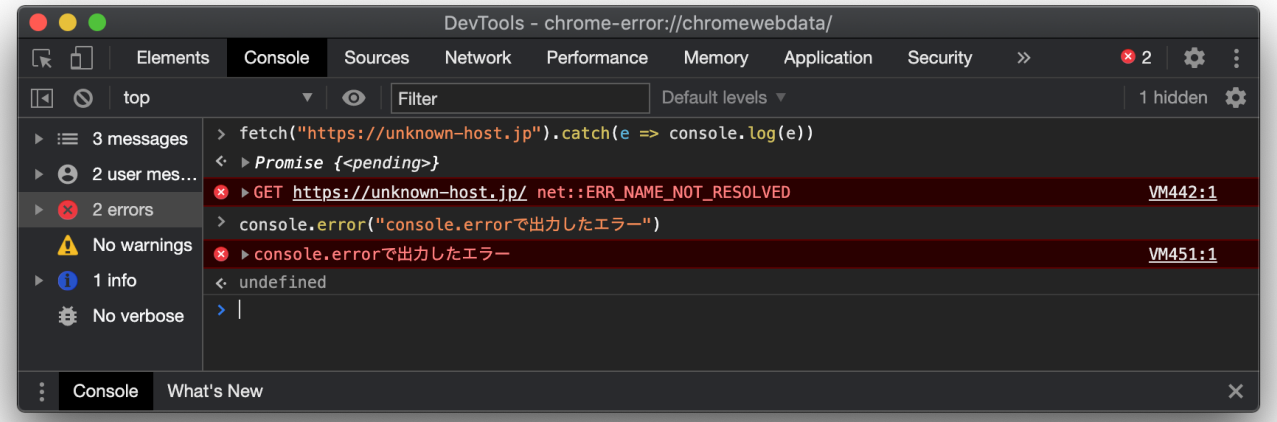
例外処理が漏れた場合、ロジックが中断します。例えば、サーバー通信結果のJSONのパースでエラーが発生すると、`SyntaxError` が発生し、その後の処理が行われなくなります。そのJSONをパースして画面に表示しようとしていた場合は、通信は行われるものの、画面表示が更新されずに何も発生していないように見えます。この場合、開発者ツールをみると、`Uncaught Error` が記録されています。なお、将来のNode.jsではエラーコード0以外で、プログラムが終了することになっています。



キャッチしていない例外

一方、やっかいなのが、サーバー通信のエラーです。XMLHttpRequestの場合は`onerror()`コールバックでエラーを取得してハンドルする必要がありますが、うまくハンドルしてもコンソールにはエラーが残ります。fetchの場合は`Uncaught (in promise) Error`が出ますが、これを適切に処理しても、やはり同様のエラーが出力されます。

これはconsole.logによるログ出力の章で紹介した`console.error()`出力と同じです。うまくハンドリングしても、キャッチしなかったときと同じように赤くエラーが表示されます。XHRを直接使う場合は`onerror`が設定されていなくて例外が握り潰されている可能性もゼロではないため注意しなければなりませんが、XHRをラップしたaxiosや、`fetch()`を使った場合には`Uncaught Error`の文字がなければ実装上は気にする必要はありません。



うまく処理できていた場合にも同様に表示されるエラー

例外処理機構以外で例外を扱う

これまでTypeScriptにおける例外処理の方法や作法などを説明してきました。しかし、ベースとなっているJavaScriptの制約により、お世辞にも使いやすい機能とは言えません。理由は以下の3つです。

- Javaの`throws`のように、メソッドがなげうる例外の種類がわからず、ソースの関数の実態やドキュメント（整備されていれば）を確認する必要がある
- JavaやC++のように、`catch`節を複数書いて、型ごとの後処理を書くことができず、`instanceof`を駆使してエラーの種類を見分けるコードを書く必要がある
- `Promise`や`async`関数で、何が`reject`に渡されたり、どんな例外を投げるのかを型定義に書く方法がない

例外に関しては、補完も効かないし、型のチェックも行えません。いっそのこと、例外処理機構を忘れてしまうのも手です。例外処理のないGoでは、関数の返り値の最後がエラーという規約でコードを書きます。 TypeScriptでも、いくつか方法が考えられます。

- タプルを使ってエラーを返す（Go式）
- オブジェクトを使ってエラーを返す
- 合併型を使ってエラーを返す

```
type User = {
  name: string;
  age:  number;
}

// タプル
function create(name: string, age: number): [User?, Error?] {
  if (age < 0) {
    return [undefined, new Error("before born")]
  }
  return [{name, age}];
}

// オブジェクト
function create2(name: string, age: number): {user?: User, error?: Error} {
  if (age < 0) {
    return {error: new Error("before born")}
  }
  return {user: {name, age}};
}

// 合併型
function create3(name: string, age: number): User | Error {
  if (age < 0) {
    return new Error("before born");
  }
  return {name, age};
}
```

この中でどれが良いかは好みの問題ですが、個人的なおすすめはオブジェクトです。タプルよりかは返り値の意味に名前をつけられるのと、合併と異なり、オプショナルチェイニングを使ってエラーチェックを簡単に書ける可能性があります。`instanceof`と毎回タイプする必要性もありません。ただ、どの書き方もマジョリティではなく、好みの問題ではあります。

まとめ

例外についての文法の説明、組み込みのエラー型、エラー型を自作する方法、非同期処理の例外処理などを説明してきました。 例外の設計も、一種のアーキテクチャ設計であるため、ちょっとした経験が必要になるかもしれません。

TypeScript、特にフロントエンドの場合、例外を無視することはユーザーの使い勝手を悪くします。どのようなことが発生し、どのケースではどのようにリカバリーするか、というのをあらかじめ決めておくと実装は楽になるでしょう。