Node.js/JavaScript入門

Node.js/JavaScript入門

Node.jsやJavaScriptの基礎知識を習得し、動作の仕組みや適切な利用方法を習得する

Gitを利用したコード管理の方法を習得する

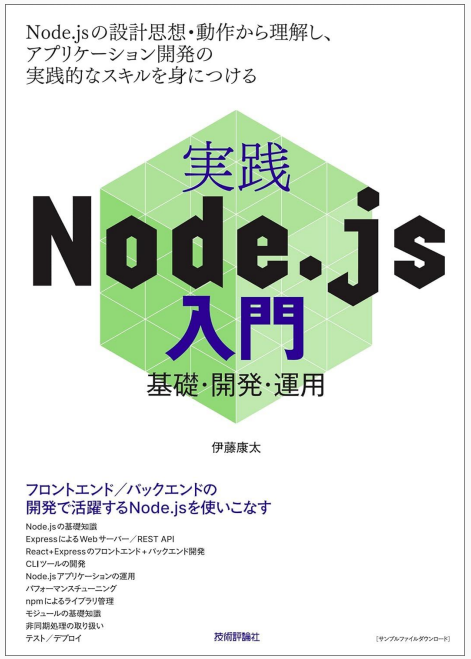
学習方法と進め方

Node.js/JavaScriptについての解説と課題

最終的に簡単なWebサービスを作れるようになることを目指す

実践Node.js入門

https://gihyo.jp/book/2023/978-4-297-12956-9



自己紹介

伊藤 康太（https://koh.dev/）

- 2013/03
法政大学情報科学部卒業

- 2013/04 - 2022/07
ヤフー株式会社
情報システム本部/CTO室
Node.jsサポートチーム立ち上げ/フロントエンド黒帯

2022/07 -
RPGテック合同会社
開発/スタートアップ/技術顧問/アドバイザー

JavaScriptとは

JavaScriptとは

主にブラウザ上で動作するスクリプト言語

動的に変化するコンテンツを作成するために利用される

JavaScriptは「この文法でこのような動作をする」という仕様

実際に動作するためには、ブラウザなどに搭載されている実行エンジンが必要となる

例えばChrome、firefox、safariなどブラウザごとに別々の実行エンジンが搭載されている

同じJavaScriptでもエンジンによって実行結果が異なることがある

JavaScriptとは

JavaScriptの文法はECMAScriptと呼ばれる仕様によって定められる
https://tc39.es/

以前はES4（ECMAScript 4）、ES5とバージョニングがされていたが、現在はES2023、ES2024のように年号でバージョニングされ、毎年リリースされる

ECMAScriptはあくまで仕様なので、動作するかはJavaScriptエンジン次第

課題: JavaScriptを実行してみよう

JavaScriptを実行し、結果を返すものがJavaScriptエンジン

JavaScriptエンジンを実感してみる

- ブラウザの開発者ツールを開く
- 次のコードを実行し結果を確認する

```
> console.log('hello JavaScript')
> 1 + 2
```

Node.jsとは

Node.jsとは

2009年にRyan Dahlによって開発/公開されたJavaScriptランタイム

“Node.js is an open-source, cross-platform JavaScript runtime environment.”

https://github.com/nodejs/node

Node.jsはChromeに内蔵されているV8というJavaScriptエンジンを利用されている

https://v8.dev/

Node.jsのインストール

Node.jsは1年に1度メジャーリリース
https://nodejs.org/en/about/previous-releases

LTS（long-term support）は3年間バグやセキュリティなどのサポートがされる

偶数バージョン: 安定板
奇数バージョン: 開発版

基本的に、アプリケーション等にはLTSのバージョンを利用する

LTSバージョンを取得して実行
https://nodejs.org/en/download/prebuilt-installer

過去のバージョンは次のページから入手可能
https://nodejs.org/dist/

課題: Node.jsをインストールして実行してみよう

Node.jsの実行の仕方を知る

- インストールを行う
- ターミナルからNode.jsが実行できることを確認

```
$ node -v
$ node
> console.log('hello')

$ node
> 1 + 2
```

課題: Node.jsをインストールして実行してみよう（REPL）

Node.jsにもブラウザの開発者ツールのような対話的にコードを実行できるREPLが備わっている

入力した計算式や関数などの結果が次の行に出力され、簡単な挙動の確認や正規表現のチェックなどに便利

```
$ node
> console.log('hello')

$ node
> 1 + 2
```

Node.jsの特徴

Node.jsの特徴

Node.jsを特徴づける大きなポイントは次の2つ

- 非同期のイベント駆動型ランタイム
- Non-Blocking I/Oとシングルスレッド

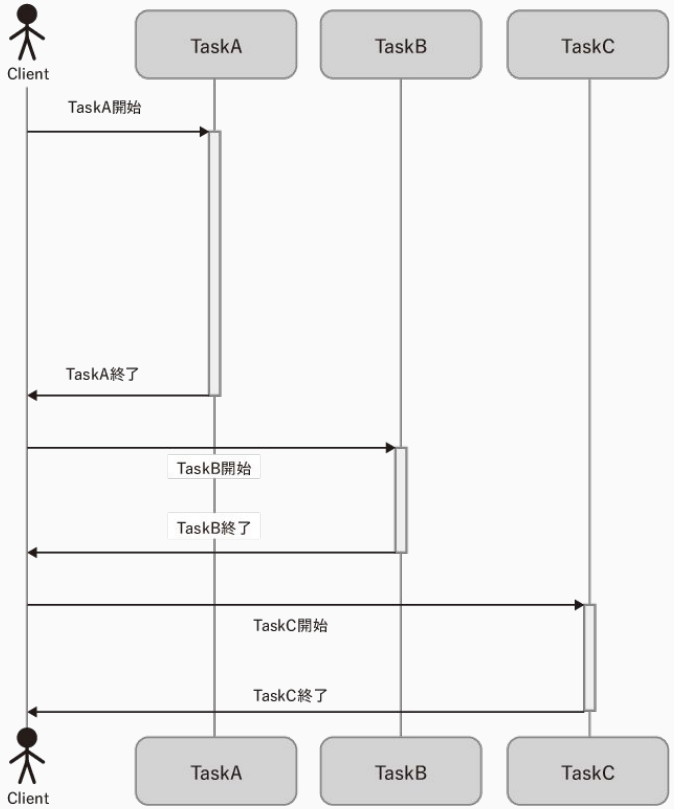
同期処理/非同期処理

通常、プログラムは記述された順番に実行される

また、記述された関数などを実行している間は、その処理が完了するまで他の処理を行わない

これは同期処理と呼ばれる

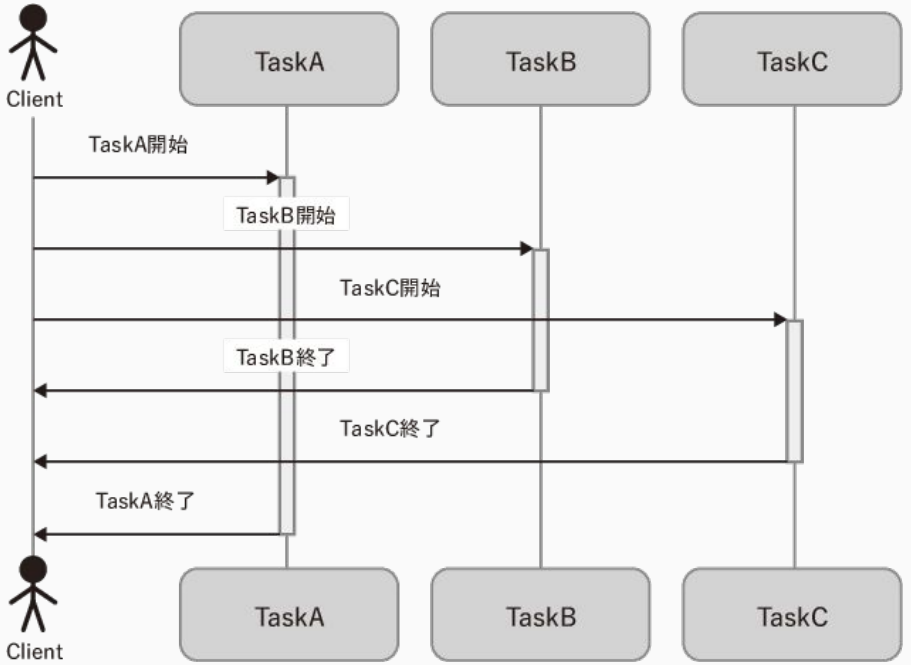
同期処理は同時に1つのタスクしか実行されない



同期処理/非同期処理

非同期処理は記述された順番で実行されるとは限らない

また、タスクの実行が完了する前に別のタスクが動作する可能性がある



JavaScriptと非同期処理

JavaScriptは、もともとブラウザ上で動的なコンテンツを扱うために利用されてきた

ブラウザ上では様々なイベントが発生する

- ユーザーが特定のHTML要素をクリックする
- キーボードで文字入力される

JavaScriptはさまざまなイベントを処理することを得意とする

- 特定のHTML要素をクリックしたら、アラートを表示する
- キーボード入力のたびに、Webページの表示を更新をする

JavaScriptは「いつか特定のイベントが起きたら、特定の処理を実行する」といった、非同期的に起きるイベントの処理が言語自体に深く結び付いている

Node.jsと非同期処理

Webページではブラウザ（の搭載するエンジン）がJavaScriptを動かしている

Node.jsではJavaScriptを動かすために、OSとランタイムが同じ役割を担う

OSもブラウザと同様にさまざまなイベントが非同期的に発生する

- ファイルの読み込み/書き込み
- ネットワークのデータ送受信
- キーボードやマウスの入力など

そのような視点で見るとブラウザとOSは似た性質を持っていると言える

ブラウザで発生していたさまざまなイベントを処理するのと同様に、Node.jsはOSで発生するさまざまなイベントを処理するランタイムであるとも言える

Node.jsと非同期処理

このようにJavaScriptはもともと言語自体に非同期を中心に処理するという性質が深く結び付いている

そのため、JavaScriptを採用するNode.jsはそもそもの文法として、非同期処理を扱いやすい特徴を持つ

既存のJavaScriptの文法を使うとOSなどから発生する非同期なイベントを無理なく表現できた

ブラウザとサーバーというまったく違ったプラットフォームではあるが、非同期という文脈は両者で共通していたため、Node.jsはJavaScriptの表現力を活用できた

Node.jsと非同期処理

Node.jsにはEventEmitterと呼ばれるさまざまイベントを発行し、受け取る汎用的なしくみが存在する

このしくみを介して、OS側のイベントをJavaScriptの世界に持ちこみ、Node.js側で処理を行う

このように発行されるイベントを下敷きにさまざまな処理を行う特徴を「イベント駆動型」と表現する。

そのためNode.jsは「非同期のイベント駆動型ラインタイム」と呼ばれている

シングルスレッドとNon-Blocking I/O

Node.jsとシングルスレッド

Node.jsには、シングルプロセス/シングルスレッドで動作する

プロセスとは実行されているプログラムを管理する単位

ソフトウェアを1つ起動すると1つのプロセスが立ち上がる（イメージ）

プロセスにはどのようなプログラムを実行しているかというような情報を持ったり、プログラムを実行するためのメモリを確保している

プロセスはプログラムを実行するスレッドを1つ以上持つ

スレッドを1つのみ利用する動作をシングルスレッド、複数利用する動作をマルチスレッドと呼ぶ

Node.jsは基本的に1つのプロセスに対し1つのスレッドを生成する、シングルスレッドで動作する（正確には内部的にマルチスレッド動作があるが、挙動の特徴としてシングルスレッドと理解しておくとよい）

イベントループとNon-Blocking I/O

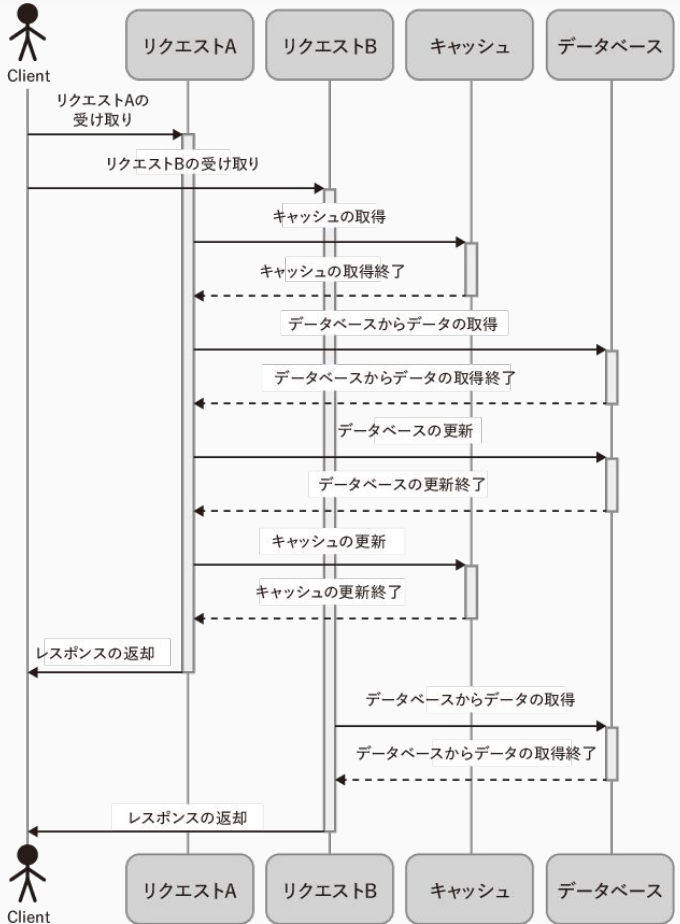
「シングルスレッドは高いパフォーマンスを得られない？」

近年のように発達したハードウェア環境においては、シングルスレッドよりマルチスレッドを利用した方が、一般的に高いパフォーマンスが期待できる

Node.jsはもうひとつの特徴であるNon-Blocking I/Oによって、シングルスレッドでも性能を最大限発揮でき、効率的なタスク処理を可能にしている

イベントループとNon-Blocking I/O

次のようなユーザーが同時に2つのリクエストを同時に送信する場合を考える



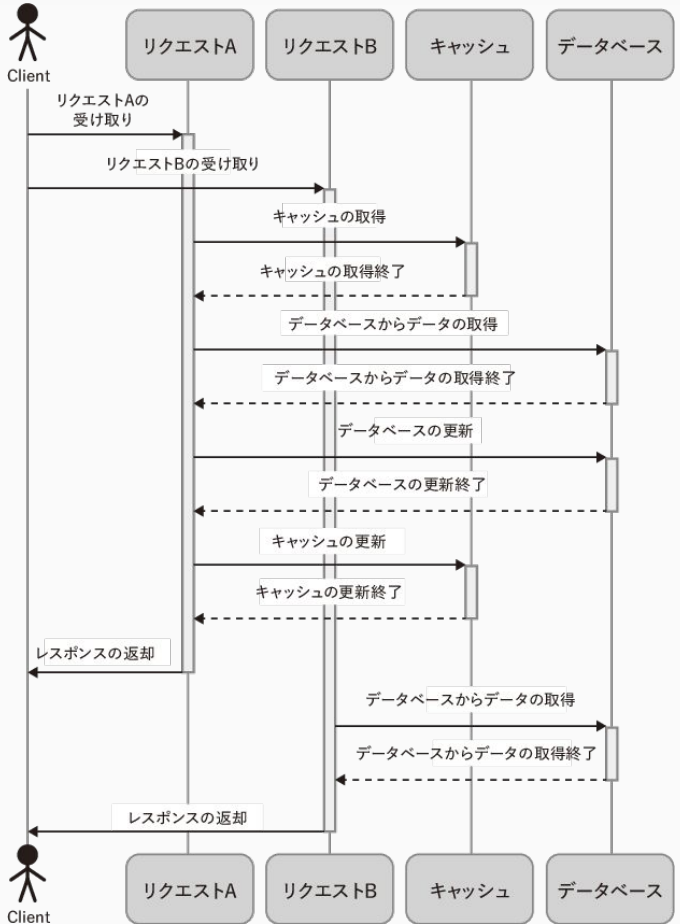
I/Oとは

I/OとはInput/Outputの略。日本語で言えば「入出力」例えばファイルの読み込み/書き込みなど、様々なI/Oが存在する

「ファイルの読み込み」に注目すると入出力は

- 入力: ファイルの内容がほしい
- 出力: ファイルの内容を返す

コンピューターは入力に対して結果が出力される機能の集まり



Blocking I/O / Non-Blocking I/O

Blocking I/O

- I/Oが終了するまで待機（I/O待ち）が発生する

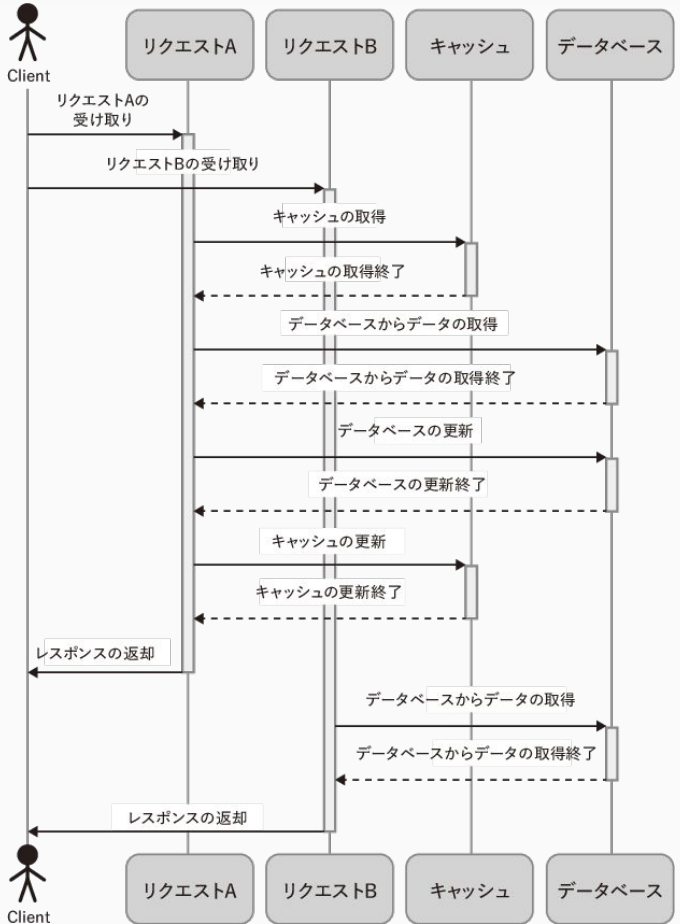
Non-Blocking I/O

- I/O発生中に別のI/Oの処理が継続できる
- Node.jsはこっち

Blocking I/O

シングルスレッドでBlocking I/Oの場合

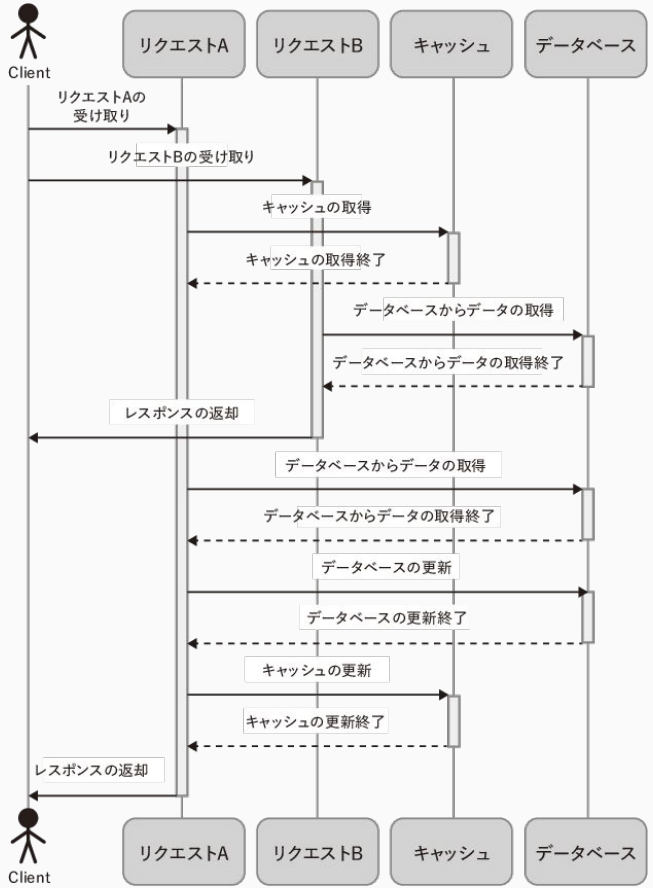
「リクエストAがI/O処理を専有し、終了するまでリクエストBは処理を返せない」



Non-Blocking I/O

シングルスレッドでNon-Blocking I/Oの場合

「リクエストAのI/O処理中にリクエストBの処理を開始できる」



Node.jsとNon-Blocking I/O

Node.jsは一連の処理をI/Oの単位で一定粒度の「イベント」に分割することで、シングルスレッドであっても同時に複数の処理を可能としている

この動作の実現にはイベントループという仕組みが採用されている

Node.jsとNon-Blocking I/O

Node.jsの非同期処理を体験

次のコードは自分自身を読み込み、読み込み完了時に出力するサンプル

次のコードをindex.jsという名前で保存して実行

```
const { readFile } = require('fs');

console.log('A');

// __filname は自分自身のファイルパスが入る
readFile(__filename, (err, data) => {
  // ファイルの読み込みが終わった時に呼び出される
  console.log('B');
});

console.log('C');
```

Node.jsとNon-Blocking I/O

readFileはファイルを読み込むNode.jsの標準機能

コードを実行すると「A -> B -> C」ではなく「A -> C -> B」の順で出力される

```
$ node index.js
A
C
B
```

```
const { readFile } = require('fs');

console.log('A');

// __filname は自分自身のファイルパスが入る
readFile(__filename, (err, data) => {
  // ファイルの読み込みが終わった時に呼び出される
  console.log('B');
});

console.log('C');
```

Node.jsとNon-Blocking I/O

readFileはファイルの読み込みを行う= I/Oを発生させる

Node.jsはI/Oをブロックしないため「ファイルの読み込み完了」を待たずに次の処理を行う

このコードは「ファイルの読み込みがいつか完了したらBと出力する」という意味になる

```
readFile(__filename, (err, data) => {
  console.log('B');
});
```

Node.jsとNon-Blocking I/O

「いつか」はどのように決定されるか

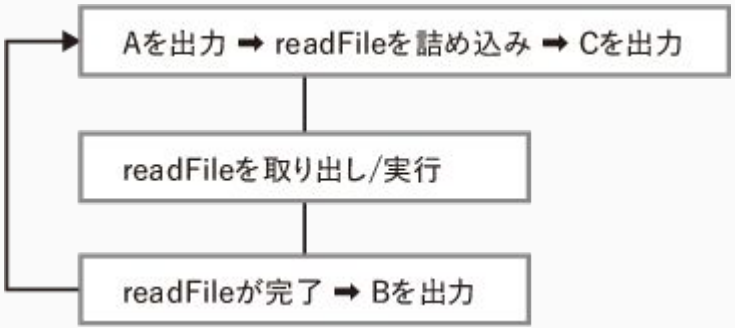
Node.jsではI/Oなどのイベントが発生すると、いったん内部に持つキューのようなものにタスクを詰め込む

このキューからタスクを取り出すために「イベントループ」と呼ばれる無限ループが内部で動いている

readFileを呼び出すといったん処理はされずキューに詰め込まれ、イベントループによって実行されるまで待機する

```
const { readFile } = require('fs');

console.log('A');
// __filname は自分自身のファイルパスが入る
readFile(__filename, (err, data) => {
  // ファイルの読み込みが終わった時に呼び出される
  console.log('B');
});
console.log('C');
```



Node.jsとNon-Blocking I/O

イベントループはキューに待機中の処理がなければアイドル状態になる

イベントループが止まるとアプリケーションは次の処理が行えない

つまりNode.jsではイベントループをいかに停止させないコードをかけるかが重要となる

https://nodejs.org/en/about#about-nodejs

イベントループはNode.jsの生命線

Node.jsとNon-Blocking I/O

プロセスをブロックするアプリケーションでは、スケールさせるためにはブロックしているプロセスと別のプロセスを増やしてタスクの同時実行数を増やす必要がある

スケーラブルなアプリケーションを開発するためには、プロセスを適切に管理する必要がある

Node.jsはシングルスレッドでNon-Blocking I/Oなため、プロセスの管理やI/O待ちを意識せずにすみ、I/Oが中心のアプリケーションをスケーラブルに保ちやすい

Node.jsの歴史

Node.jsの歴史

Node.jsはイベント駆動とNon-Blocking I/Oで、I/Oバウンドな処理の多いアプリケーションでリソースを活かすコードが書きやすい

Node.js登場当時はインターネットや技術の発展に伴い、クライアントからWebサーバーに送信されるリクエスト数は非常に増え、古いアプローチで実装されているWebサーバーではC10K問題の発生などがいわれていた

古いアプローチでは1つのアクセスに対して1つのプロセスを割り当てるものが多かった

このアプローチの場合1万クライアントのアクセスに対し、1万のforkが必要になる

マシンのfork数上限に達してしまうとマシンリソースに余裕があっても応答が悪化する

クライアント数が10K（1万）となった際に応答が悪化する＝C10K問題

Node.jsの歴史

Node.jsはNon-Blocking I/Oなのでプロセスに I/Oが発生している最中にも、別のリクエストに対処できる＝C10K問題が起きにくい

当時の環境ではリクエストの数が多くなってもサーバーのリソースを効率的に活用しやすかった

当時こういった駆動モデルは珍しく、C10K問題への対処なども含めて期待がもたれやすい土壌があった（現在では多くの環境がこれらの問題に対処されている）

I/Oには強いが、逆にCPUを長時間占有するような処理（動画のエンコードなど）には弱いため、特徴を活かしパフォーマンスを維持するためには適した利用法を意識する必要がある

Node.jsの歴史

Node.js登場初期のJavaScriptは複雑化に対応する進化の最中で、様々な新しい仕様が登場していた

- Modules
- const/letなどスコープ単位の変数定義
- Arrow Function
- etc…

策定された仕様の量に対しブラウザの実装がすべてついていけるわけではなく、新しい機能をすべてのブラウザで利用できる状況ではなかった

そういった状況の解決のため、Babelなどのトランスパイル（新しいコードを古い文法へ変換する）ツールが開発された
https://babeljs.io/

Node.jsの歴史

当時のJavaScriptでは標準のモジュール分割はなかった(JavaScriptファイル分割自体は可能可能だったがスコープなど使い勝手に課題があった)

これは複雑化するフロントエンド(ブラウザ側のJS)開発においてはコストが高く、より使いやすいモジュールシステムの導入が求められていた

仕様は議論されていたが、実際に利用できる状況ではなかった

Node.jsが採用していたCommonJS Modulesという仕組みをフロントエンド開発に持ち込み、ファイルを結合し疑似的にモジュール分割の課題を解消するバンドラーが登場

JavaScript開発のためのツールのため、それらのツールはコンテキストスイッチの少ないJavaScriptで記述される合理性が高く、開発にNode.jsが採用された

アプリケーションの開発だけではなく、Node.jsはフロントエンドの開発ツールという立場においても重用されるようになる

Node.jsの歴史

ツールの開発で利用範囲が広がりNode.jsの扱いになれた開発者が増えるにつれて、本来の得意な領域であるサーバーサイドでの採用事例も増えていった

また、1つのサーバーにアプリケーションの機能を詰めこんだモノリシックなスタイルから、機能ごとにアプリケーションを分割するマイクロサービスというスタイルが流行をはじめた頃でもあり、フロントエンドに関係する機能が分割されやすくなった

フロントエンド/バックエンドの両方を同じ言語で記述できるNode.jsは開発のコンテキストスイッチを少なくできる合理性もあり、特にフロントエンドに近い領域でシェアが広がっていった

JavaScript基礎

変数

変数

JavaScriptでは3つの方法で変数を宣言可能

宣言に用いる	内容
var	変数を宣言、初期化
const	スコープ内で有効な再代入不可能な変数を宣言
let	スコープ内で有効な変数を宣言、初期化できる

スコープ: 変数の値や式が参照できる範囲（関数スコープ、if文、for文など）

変数

varはスコープを超えて参照が可能

let/const は宣言したスコープ内でのみ有効

```
if (true) {
  var foo = 5;
}
console.log(foo); // 5
```

```
if (true) {
  const bar = 5;
}
console.log(bar);
// ReferenceError: bar is not defined
```

変数

let/constの違いは「再代入が可能か」

constで宣言した変数に再代入すると実行時にエラーが発生する

```
const foo = 5;
console.log(foo);

foo = 'test';
console.log(foo);
```

```
$ node index.js
foo = 'test';
    ^
TypeError: Assignment to constant variable.
```

変数

JavaScriptに限った話ではないが、変数への再代入が多くなるとバグの原因になりやすい

変数の基本は「スコープは最小限」に「再代入は少なく」書くこと

JavaScriptにおいては「const > let > var」の優先順位で採用するとよい

（現実的に新規のコードでvarを採用することはほぼない）

変数

変数の名前（識別子）は下記のものが利用できる

- 変数名の先頭から利用できる
  - 文字
  - アンダースコア（_）
  - ドル記号（$）
- 先頭以外で利用できる
  - 数字

```
const abc = 'abc'; // OK
const _abc = '_abc'; // OK
const abc123 = 'abc123'; // OK
const 123 = '123'; // NG 数字先頭の定義はできない
```

変数(課題)

- 下記のコードを実行してエラーがでることを確認
- constをletに置き換えてエラーがでないことを確認

```
const foo = 5;
console.log(foo);

foo = 'test';
console.log(foo);
```

変数(課題)

- 下記のコードを実行
  - 文字、アンダースコア(_)、ドル記号($)が利用できることを確認
  - 数字で開始する変数名はSyntaxErrorが発生することを確認

```
const abc = 'abc'; // OK
const _abc = '_abc'; // OK
const abc123 = 'abc123'; // OK
const 123 = '123'; // NG 数字先頭の定義はできない
```

データ型

データ型

JavaScriptは動的な型付けの言語なため、明示的に型を宣言せずに利用できる

実際には加減算や比較など、変数の内部にどのような型のデータが保存されているかを意識するケースはままある

そのため、どのようなデータ型が存在するかを意識しておくことは重要

データ型

7つのプリミティブな型と、それらを複合的に扱うObject型が利用可能

データ型	内容
String	文字列を表す型
Number	整数や浮動小数点数などの数値を扱う型
BigInt	大きな桁を扱う整数値
Boolean	trueかfalseをもつ真偽値
Symbol	一意な値となるシンボル値
undefined	未定義を表す型
null	データがないことを示す型
Object	オブジェクトや配列、正規表現、関数など プリミティブ以外の型

データ型

データ型は typeof 演算子によって確認可能

typeofが返すデータ型は一部先の表とは異なる

```
typeof 'string'     // 文字列は'string'
typeof []           // 配列は'object'
typeof console.log // 関数は'function'
typeof null         // nullは'object'
```

データ型: String

下記でくくられたものを文字列として扱う

- "（二重引用符）
- '（単一引用符）
- `（逆引用符）

```
$ node
> "こんにちは"
'こんにちは'
> 'こんにちは'
'こんにちは'
> `こんにちは`
'こんにちは'
```

"と'の2つはどちらも同じ結果を返す

好きな方を利用してかまわないが、実際のコード中ではどちらかに統一したほうがよい

`（逆引用符）はテンプレートリテラルと呼ばれる記法

変数の展開や複数行にまたがる文字列の定義などが可能

データ型: Number/BigInt

Numberは数値を扱う型

整数または小数表記を用いる

```
const int = -5;
const double = 3.4;
```

BigIntはNumberでは扱えない大きな範囲を扱う型

```
const one = BigInt(1)
// 1n
const bigInt= BigInt(Number.MAX_SAFE_INTEGER);
// 9007199254740991n
```

データ型: Boolean

Booleanは真偽値であるtrueとfalseの2つを扱う

```
const okFlag = true;
const ngFlag = false;

if (okFlag) {
  console.log('ok'); // okが出力される
} else {
  console.log('ng');
}
```

データ型: undefinedとnull

undefinedは変数が未定義であることを表す

```
$ node
> x
undefined
```

nullは「存在しないこと」を表す

```
const data = null
```

データ型: undefinedとnull

undefinedとnullは実行時エラーなど、思わぬエラーを引き起こしがちなので慎重に扱う

```
let obj = { foo: 'hello' };
console.log(obj.foo); // hello

obj = null;
console.log(obj.foo);
// Uncaught TypeError: Cannot read properties of null
(reading 'foo')
```

データ型: Object

ざっくりと言うとプリミティブ（String, Number, Booleanなど）型以外のもの

以下のような、様々なものがObjectを継承して実装されている

- 配列
- Date
- 正規表現
- 関数
- など

データ型: Object

Object初期化は {} でくくった「プロパティ名 : 値」で表現される

プロパティ名には文字列型や数値型、Symbol型などを指定可能

```
const obj = {
  key: 'value',
  key2: 'value2'
};
```

データ型: Object

プロパティの値にはプリミティブ型やObjectの入れ子、Dateオブジェクトや正規表現、関数などさまざまなものが保持可能

```
const obj = {
  foo: {
    bar: 'baz'
  },
  now: new Date(),
  func: function() {
    console.log('function');
  }
}
```

データ型: Object

近年ではいくつかの新記法が登場し、省略やプロパティ名をオブジェクト外で定義したりできる

```
const key = 'value';
const key2 = 'value2';

const obj = {
  key, // keyの値(value)が入る
  key2 // key2の値(value2)が入る
};
```

```
const key = 'keyName';

const obj = {
  [key]: 'value'
};

console.log(obj);
// { keyName: 'value' }
```

データ型: Object

Objectで宣言したプロパティには . もしくは [] でアクセスができる

```
const obj = {
  foo: 'hello',
  bar: {
    baz: 'world'
  }
};

console.log(obj.foo);
console.log(obj['foo']);
console.log(obj.bar.baz);
console.log(obj['bar']['baz']);
```

```
const obj = {
  123: '数値'
  '': '空文字列'
};

console.log(obj.123); // SyntaxError
console.log(obj[123]); // 数値
console.log(obj.''); // SyntaxError
console.log(obj['']); // 空文字列
```

.はプロパティ名が変数の識別子と同じパターンの時にしか使えない

データ型: Object

Objectのプロパティは宣言後に書き換えが可能
constは再代入の禁止なので、プロパティの書き換えは可能

```
const obj = {
  foo: 'hello'
};
console.log(obj.foo); // hello


obj.foo = 'good bye';
console.log(obj.foo); // good bye
```

データ型: 配列

[]でくくられたものをリストとして扱うことができる

区切り文字は , で配列の添字は 0 から始まる

宣言された配列はArrayオブジェクトとして扱われ、いくつかのプロパティや関数を利用できる

```
const arr = ['foo', 'bar', 'baz'];

console.log(arr[0]); // foo
console.log(arr.length); // 3
```

データ型: 配列

配列がもつ関数には以下のようなものがある

- Array.prototype.map: 配列の要素をループしながら新しい配列に変換する
- Array.prototype.filter: 条件に一致したもののみを抽出する

データ型: 配列

```
const students = [
  { name: 'Alice', age: 10 },
  { name: 'Bob', age: 20 },
  { name: 'Catherine' , age: 30 }
];

const nameArray = students.map(function(person) {
  return person.name;
});
console.log(nameArray); // ['Alice', 'Bob', 'Catherine']


const under20 = students.filter(function(person) {
  return person.age <= 20;
});
console.log(under20); // [{ name: 'Alice', age: 10 }, { name: 'Bob', age: 20 }]
```

演算子

演算子

代入、比較、算術、文字列などの演算子が利用できる

```
const a = 2; // 2
const b = a * 2 + 1; // 5
const str = 'hello' + ' world'; // hello world

const less = a < b; // true
const equal = a === b; // false
```

演算子

等価比較は ==（等価）と ===（厳密な等価）の2つがあることに注意

== に対して === はより厳密な比較になる

不等価演算子にも != と !== が存在する

```
// 数字同士の比較
const a = 1;
const b = 1;
const equal = a == b; // true
const equal2 = a === b; // true
```

```
// 数字と文字列の比較
const a = 1;
const b = '1';
const equal = a == b; // true
const equal2 = a === b; // false
```

https://developer.mozilla.org/ja/docs/Web/JavaScript/Reference/Operators/Strict_equality

ループ

ループ

一番基本となるループはC言語ライクな文法のfor文

```
const arr = ['foo', 'bar', 'baz'];

for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
// foo
// bar
// baz
```

ループ

配列オブジェクト（Array）にはforEach関数が存在する

```
const arr = ['foo', 'bar', 'baz'];

arr.forEach((element) => {
  console.log(element);
});
// foo
// bar
// baz
```

ループ

現在の環境では for...of を利用するのがよい

```
const arr = ['foo', 'bar', 'baz'];

for (const element of arr) {
  console.log(element);
}
// foo
// bar
// baz
```

ループ

for...ofは反復処理可能なオブジェクトをループ処理る
for...ofはforEachに比べ配列以外（MapやSetなどiteratorを持つオブジェクト）もループできることや、途中に非同期処理を挟むことができる優位性がある

for...ofか、基本的なfor文の2択で選ぶのがよい

- 添字が必要ない → for...of
- 添字が必要 → for文

JavaScriptとスコープ

関数

関数

関数はfunctionというキーワードから始まる定義

addが関数名、()で囲まれたa,bが引数、{}で囲まれた部分が関数の処理部分

```
function add(a, b) {
  return a + b;
}

const value = add(1, 2);
console.log(value); // 3
```

関数

関数の引数にオブジェクトを渡した場合は参照渡し

```
function setName(obj) {
obj.name = 'Bob';
}

const person = { name: 'Alice' };
console.log(person.name); // Alice

setName(person);
console.log(person.name); // Bob
```

関数

関数は関数式として別名の変数に代入が可能
この時は関数名を省略できる（無名関数）

```
const add = function(a, b) {
  return a + b;
}

// Callbackに無名関数
setTimeout(function() {
  console.log('1s')
}, 1000);
```

関数

近年の関数にはデフォルト引数やArrow Function（アロー関数）という新しい記法が登場する

```
function add(a, b = 2) {
return a + b;
}

const total = add(1);
console.log(total); // 3

const total2 = add(1, 3);
console.log(total2); // 4
```

関数

通常の関数とArrow Functionはスコープなど細かい挙動差が存在する

左のコードはほぼ同じ挙動

Arrow Functionは1つしかない引数の()や、関数内部の処理が1行の場合の{} , returnの省略が可能

```
// Function
function add(a, b) {
  return a + b;
}
// Arrow Function
const add = (a, b) => {
  return a + b;
};
```

```
const double = a => a * 2;
console.log(double(3)); // 6
```

継承

関数

配列にはArray.prototype.mapやArray.prototype.filterといった共通の関数がある

これは配列の元となるArrayオブジェクトに紐づいている関数

JavaScriptはclassではなくprototypeによって継承関係を実現している

JavaScriptではObjectを継承してArrayなど各種の派生型が定義されている

prototypeを拡張することで独自の継承関係を記述できる

関数

Arrayオブジェクトに配列の長さを出力する独自関数を拡張する場合

```
// 配列の長さを標準出力に表示する
Array.prototype.showLength = function() {
  // thisは生成された配列自身をさす
  console.log(this.length)
}

const a = [1, 2, 3];
console.log(a); // [ 1, 2, 3 ]
a.showLength(); // 3
```

JavaScriptとclass

JavaScriptとclass

ES2015以降のJavaScriptではclass構文が導入されている

コンストラクタの引数で与えたnameをthis.nameに保持し、その値をprintName関数で表示するclassの例

JavaScriptのclassはprototypeのシンタックスシュガー

```
class People {
  // コンストラクタ
  constructor(name) {
    this.name = name;
  }
  printName() {
    console.log(this.name);
  }
}

const foo = new People('foo-name');
foo.printName(); // foo-name
```

```
// コンストラクタ
function People(name) {
  this.name = name;
}

People.prototype.printName = function() {
  console.log(this.name);
}

const foo = new People('foo-name');
foo.printName(); // foo-name
```

JavaScriptとthis

JavaScriptとthis

JavaScriptのthisは、実行される場所（コンテキスト）によって値が変わるプロパティ

グローバルなコンテキストで実行された場合のthisは、Node.jsではglobalオブジェクトを指し示す（ブラウザではwindowオブジェクト）

```
$ node
> this === global
true
```

JavaScriptとthis

次のコードを実行するとtrueが出力される

```
function isGlobal() {
console.log(this === global);
}

isGlobal(); // true
```

```
$ node index.js
true
```

つまりこの時のthisは先ほどと同様globalオブジェクト

これはisGlobal関数がグローバルなコンテキストで実行されているため

JavaScriptとthis

別コンテキストで実行する場合を確認する

以下のコードはthis.nameの宛先（global.name）が存在しないため、実行するとundefinedが出力される

```
function printName() {
  console.log(this.name);
}

printName(); // undefined
```

```
$ node index.js
undefined
```

JavaScriptとthis

printNameをオブジェクトに入れて実行する

obj.printNameはオブジェクトの内部というコンテキストで実行される

この時のthisはobj自身になる

```
function printName() {
  console.log(this.name);
}

const obj = {
  name: 'obj-name',
  printName: printName
};
obj.printName(); // obj-name
```

```
$ node index.js
obj-name
```

JavaScriptとthis

1秒後にthis.nameを実行するコードに変更する

この時はobj.printNameはundefinedを出力する

```
function printName() {
 setTimeout(function () {
   console.log(this.name)
 }, 1000);
}

const obj = {
 name: 'obj-name',
 printName: printName
};

obj.printName(); // undefined
```

```
$ node index.js
undefined
```

JavaScriptとthis

setTimeoutは呼び出し元とは別のコンテキストで内部の関数を実行する

そのためthisの向き先が変わりthis.nameがundefinedとなった

```
function printName() {
 // printNameのコンテキスト
 setTimeout(function () {
   // setTimeoutのコンテキスト
   console.log(this.name)
 }, 1000);
 // printNameのコンテキスト
}
```

JavaScriptとthis

コンテキストを一致させるためにはbindによる固定などいくつか方法がある

```
function printName() {
 setTimeout(function () {
   console.log(this.name);
   // setTimeoutのコンテキストをprintNameのthisに固定する
 }.bind(this), 1000);
}
```

JavaScriptとthis

Arrow Functionはbindの機能が含まれている

```
function printName() {
 // Arrow Functionのthisは呼び出し元になる=printNameと一致する
 setTimeout(() => {
   console.log(this.name)
 }, 1000);
}
```

Gitを使った
コード管理

Gitを使ったコード管理

Git: ソースコードの履歴などを保存するバージョン管理システム

GitHub: Gitで管理しているソースコードをクラウド上に保存するサービス

開発プロジェクトに必要な様々な機能が付随する

Node.jsなどのOSSはGitHub上で公開されながら世界中の開発者によって開発されている
https://github.com/nodejs/node

アカウント作成

下記URLからメールアドレス、パスワード等を入力してセットアップ
https://github.com/signup

Two-factor authenticationを設定したほうがよいが、今回の説明ではスキップする
https://github.com/settings/security

鍵の作成

ssh-keygenコマンドを利用してgithubと通信するための鍵を作成

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/username/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/username/.ssh/id_ed25519
Your public key has been saved in /home/username/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:xxxxx your_email@example.com
```

https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

sshの設定（ローカル）

作成した鍵をsshで利用できるように設定する

- /home/username/.ssh/config

```
Host github.com
  AddKeysToAgent yes
  UseKeychain yes
  IdentityFile ~/.ssh/id_ed25519
```

公開鍵の設定（GitHub）

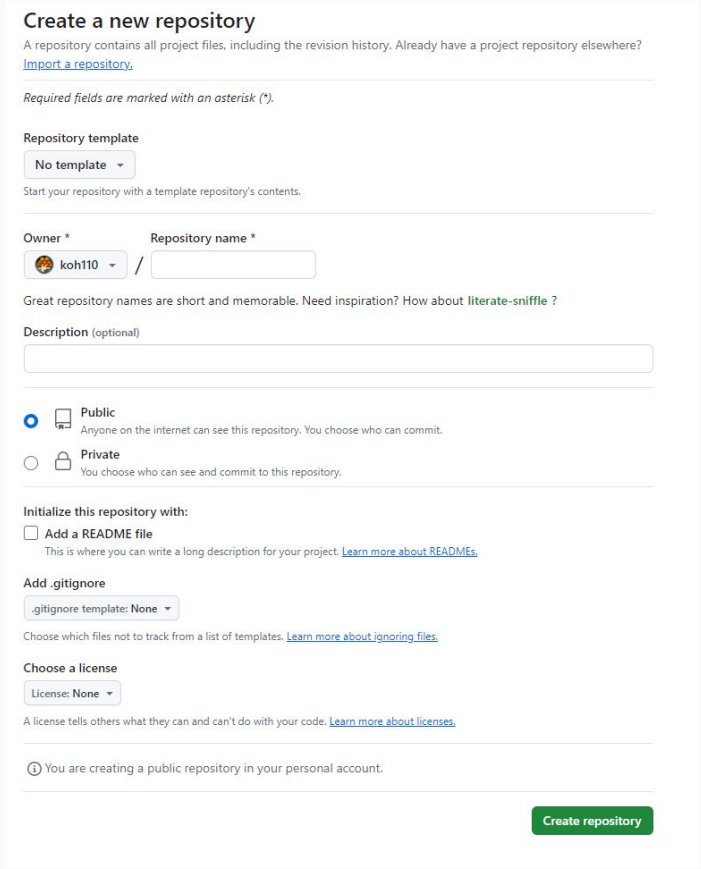
SSH and GPG keysに作成した公開鍵を設定
https://github.com/settings/keys

- 設定画面からNew SSH Keysを選択
- Title: 管理しやすいタイトル
- Key type: Authentication Key
- Key: 作成された公開鍵の値（/home/username/.ssh/id_ed25519.pub）

リポジトリの作成と管理

リポジトリ: コードを管理する箱

- ヘッダー右上の + から New repositoryを選択
  - https://github.com/new
- リポジトリの名前を入力
- Create repositoryを選択



リモートリポジトリへのpush

先ほど作成したリポジトリへファイルをアップロード（push）

```
$ mkdir test-hosei # 作業用ディレクトリの作成
$ cd test-hosei # ディレクトリへ移動
$ echo "# test-hosei" >> README.md # README.mdファイルを作成しつつタイトルを書き込み
$ git init # gitの初期化
$ git add README.md # 管理ファイルにREADME.mdを追加
$ git commit -m "first commit" # コミットメッセージの追加（save）
$ git switch -c main # mainブランチ（デフォルトのブランチ）に変更
$ git remote add origin git@github.com:account/repository-name.git # アップロード先を設定
$ git push -u origin main # GitHubへアップロード
```

リモートリポジトリからpull

GitHubへpushしたファイルは手元から削除しても復旧できる

```
$ rm -rf test-hosei # アップロードしたディレクトリを削除
# 先にアップロードしたファイルをダウンロード
$ git clone git@github.com:account/repository-name.git
$ cd repository-name # ダウンロードしたディレクトリへ移動
$ cat README.md # ダウンロードしたファイルを確認
```

ブランチ

ブランチは履歴のある地点から、別のバージョンや機能を独立して開発/管理する機能

GitHub上ではリポジトリを作成するとmainという名前のブランチがデフォルトブランチとして設定される

特定のブランチから作業用に別のブランチを切り出し、変更をコミット。
元のブランチにマージすることで変更を独立して管理が可能。

図

ブランチ

```
$ git switch -c feature-a # feature-aという名前のブランチを作成してそのブランチに移動する
$ git branch # 現在のブランチを確認
* feature-a
  main

$ vim README.md # エディタでREADME.mdを修正
$ git add README.md # 修正したファイルをコミット対象に追加
$ git commit -m”fix title” # 修正の内容をコミットメッセージに追加
$ git add README.md # 修正したファイルをコミット対象に追加
$ git commit -m”fix title” # 修正の内容をコミットメッセージに追加
```

ブランチ

```
$ git switch main # mainブランチに戻る
$ cat README.md # 先に加えた変更が戻っていることを確認
# test-hosei

$ git diff feature-a
diff --git a/README.md b/README.md
index 48967ab..bf54112 100644
--- a/README.md
+++ b/README.md
@@ -1 +1 @@
-# test hosei fix
+# test-hosei
```

ブランチ

```
$ git merge feature-a # feature-aの内容を取り込み
Updating a21d06d..5c14c90
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git push -u origin main # GitHubに修正をpush
```

.gitignore

Gitで管理しないファイルやディレクトリ（バイナリや鍵、自動生成物など）は .gitignore ファイルを配置することでコミット対象から除くことができる

```
$ touch README.md
$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md
```

.gitignore

.gitignoreファイルにファイルやフォルダを追加すると、Gitのコミット対象に選択できなくなる

```
# README.mdをGitの管理から外す
$ echo "README.md" >> .gitignore
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
```

GitHub: Pull request

ブランチのマージをGitHubのUI上でできるようにしたもの

また、レビューや自動テストなど、複数人での開発に必要な機能も用意されている

```
$ git switch -c feature-b # feature-bを作成
$ echo "console.log(‘hello’)" >> index.js
$ git add index.js
$ git commit -m”add JavaScript”
$ git push -u origin feature-b # GitHubにfeature-bブランチをpush
```

GitHub: Pull request

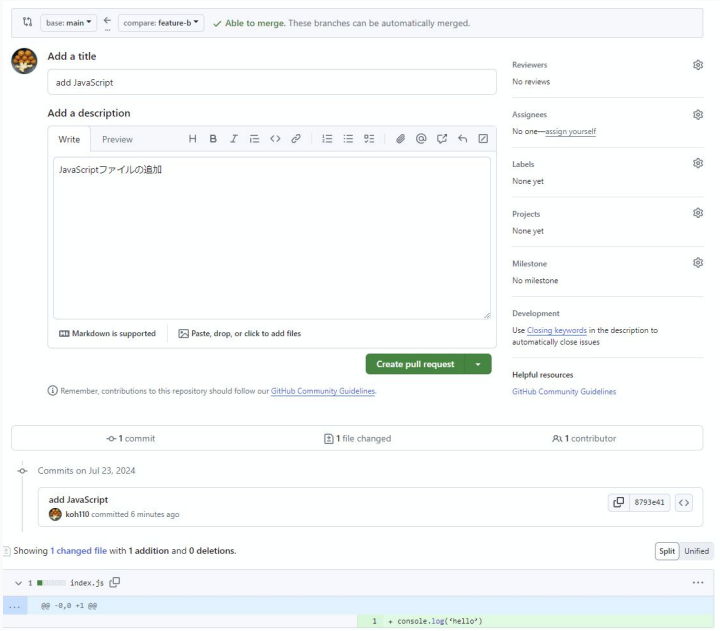
Pull requestsを選択し、New pull requestから新しいPull requestsを作成

base:main ← compare:feature-b となるように選択

変更内容を確認しCreate pull requestを選択

descriptionに変更内容を追加

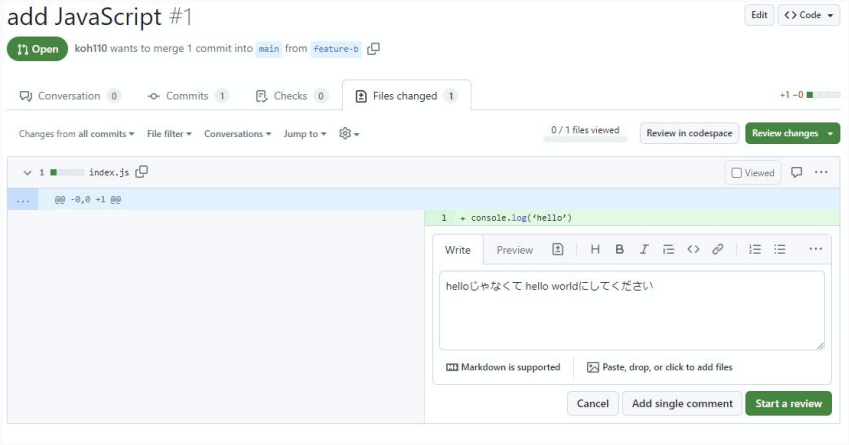
Create pull requestを選択しPull requestを作成

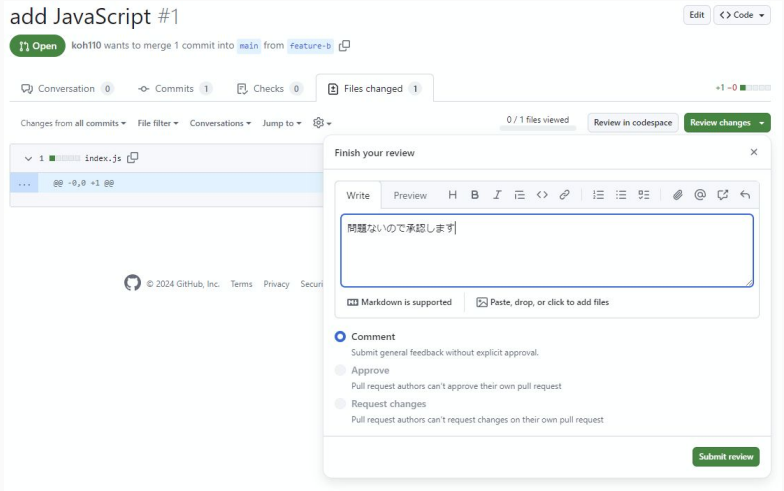


GitHub: Pull request

内容を確認してレビューコメントを追加したり、変更を依頼したり、内容に問題がなければApprove（承認）やマージなどができる

特に複数人で開発を行うOSSなどではレビューは必要になるステップ



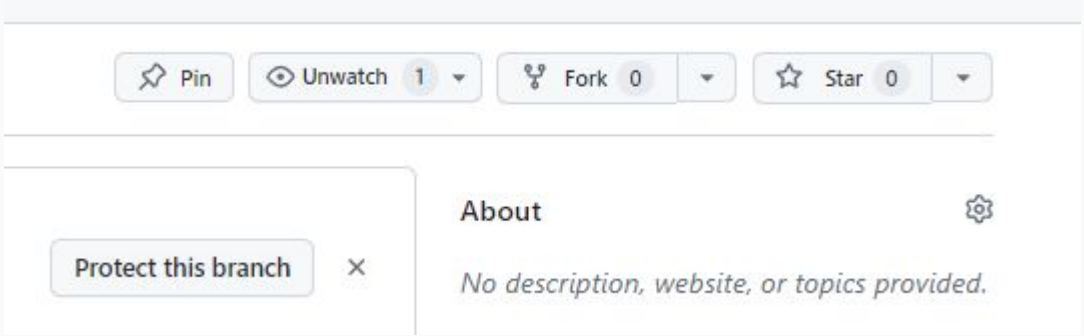


GitHub: Fork

別のリポジトリからコードをコピーする仕組み

https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/fork-a-repo

OSS活動などで自分が権限を所有していないリポジトリはForkすることで、自身が権限を持つリポジトリにコピーを作成し、修正を加えたものを本流のリポジトリにPull Requestとして送信することが多い



課題

- 適当なJavaScriptのファイルを作成してリポジトリにプッシュしてみる
- JavaScriptのファイルを修正し、別ブランチからPull Requestを送る
- Merge pull requestを行い、デフォルトブランチに修正を取り込む
- リポジトリをForkして修正ブランチを作成し、本流のリポジトリにPull Requestを送信する
  - https://github.com/koh110/test-hosei

Node.jsとモジュール

Node.jsとモジュール

モジュール: ファイルを分割する仕組み

それぞれのファイルが独立したスコープを持ち、明示的に外部に公開しない限り別ファイルから参照できない

Node.jsには歴史的な経緯から2種類のモジュール分割方法がある

- CommonJS modules: 標準が定まっていなかったころにNode.jsが採用
- ECMAScript modules: ECMAScriptの標準

Node.jsとモジュール

v20（LTS）ではCommonJS/ECMAScript modulesのどちらも特別なオプションなしに利用できる

2つには多少の機能差が存在し、相互呼び出しも可能な部分はあるがいくつか制約がある

そのため、現時点ではアプリケーションを作成する際には2つの形式を混ぜずに構築するほうがよい

Node.jsとモジュール

ブラウザで動作するJavaScriptは、最初期はとても小さなコードで十分だったため、モジュールの概念が存在しなかった

そのため、別ファイルに存在する関数などを参照するためにはグローバル変数に格納する必要があった

jQueryで言えば$がこれにあたり、scriptタグでjQueryのタグを読み込むと$が定義され、各スクリプトから利用可能になる

Node.jsとモジュール

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
<script>
  // $がグローバル変数として定義される
  $('.foo').html('bar');
</script>
<body>
  <div class="foo">foo</div>
</body>
```

Node.jsとモジュール

しかし、JavaScriptの機能は拡張され続け、グローバルな参照だけではアプリケーションの作成難易度が高くなり、JavaScriptの標準としてECMAScript modulesが策定された

現在ではモダンなブラウザーやNode.jsなどのJavaScriptランタイムで実装が完了しECMAScript modulesが利用可能となっている

CommonJS modules

CommonJS modules（CommonJS、CJS）

exportsというキーワードで関数や変数を外部に公開可能
exportsによって公開された変数や関数をequireで読み込む

```
// calc.js
exports.num = 1;

exports.add = (a, b) => {
  return a + b;
};

exports.sub = (a, b) => {
  return a - b;
};
```

```
// index.js
const calc = require('./calc');

console.log(calc.num); // 1

let res = calc.add(3, 1);
console.log(res); // 4

res = calc.sub(3, 1);
console.log(res); // 2
```

CommonJS modules: 省略

パスを指定して読み込む場合.jsを省略可能

./path/to/index.js の場合は ./path/to まで省略可能

具体的なロジックはドキュメントを参照

https://nodejs.org/api/modules.html#modules_all_together

```
const calc = require('./calc');

console.log(calc.num); // 1

let res = calc.add(3, 1);
console.log(res); // 4

res = calc.sub(3, 1);
console.log(res); // 2
```

CommonJS modules: module.exports

exports.xxx 以外にも module.exports という変数が利用できる

module.exportsを利用するとexports.xxxは上書きされる

```
module.exports = {
  num: 1,
  add: (a, b) => {
    return a + b;
  },
  sub: (a, b) => {
    return a - b;
  }
};
```

```
exports.num = 1;

exports.add = (a, b) => {
  return a + b;
};

exports.sub = (a, b) => {
  return a - b;
};
```

課題: CommonJS modules

下記のコードを実行し、同様の結果が出力されることを確認

関数を追加でexportsし、読み込んだ先で利用できることを確認

```
exports.num = 1;

exports.add = (a, b) => {
  return a + b;
};

exports.sub = (a, b) => {
  return a - b;
};
```

```
const calc = require('./calc');

console.log(calc.num); // 1

let res = calc.add(3, 1);
console.log(res); // 4

res = calc.sub(3, 1);
console.log(res); // 2
```

CommonJS modules: シングルトン

モジュールはシングルトン（＝プロセス全体で一意）で読み込まれることに注意する

2つのファイルから同時に読み込みが発生する場合を考える

```
path/to/folder/
├── index.js
├── a.js
├── b.js
└── calc.js
```

```
// a.js
const calc = require('./calc');
// 実行されたらすぐにcalc.numを書き換える
calc.num = 5;
```

```
// b.js
const calc = require('./calc');
// 実行されたらすぐにcalc.numを書き換える
calc.num = 10;
```

課題: シングルトン

下記のコードを実行してみる

aとbの読み込み順を入れ替えて実行してみる

```
// index.js
const a = require('./a'); // a.jsを読み込み（実行）
const b = require('./b'); // b.jsを読み込み（実行）
const calc = require('./calc');

console.log(calc.num);
```

CommonJS modules: シングルトン

- require('./a')
  - a.jsの中身が実行され、calc.numに5が代入される
- require('./b')
  - b.jsの中身が実行され、calc.numに10が代入される
- calcを読み込んでcalc.numを表示する
  - b.jsが後に実行されたのでcalc.numは10が入る

a.jsやb.jsが変更した値がindex.jsにも伝播している
→ calc.jsの参照先は同じ

```
const a = require('./a');
const b = require('./b');
const calc = require('./calc');
console.log(calc.num);
```

ECMAScript modules

ECMAScript modules（ESM）

JavaScriptの標準として策定されたモジュールの仕組み

export/importというキーワードで公開/読み込みが可能

```
// calc.mjs
export const num = 1;

export const add = (a, b) => {
 return a + b;
};

export const sub = (a, b) => {
 return a - b;
};
```

```
// index.mjs
import { num, add, sub } from './calc.mjs';

console.log(num);

let res = add(3, 1);
console.log(res); // 4

res = sub(3, 1);
console.log(res); // 2
```

ECMAScript modules（ESM）

Node.jsの世界では昔からCommonJS modules形式が利用されてきたため.js の拡張子をもつJavaScriptファイルはCommonJS modules形式のファイルが多く存在している

その差異による摩擦を少なく扱うために、Node.jsでは .js はデフォルトではCommonJS modulesとして扱い、ECMAScript modulesを採用するファイルはデフォルトでは .mjs という拡張子が利用される

ECMAScript modules: default export

CommonJS modulesのmodule.exportsと似たdefault exportの機能がある

CommonJSとは違い共存が可能

```
// calc.mjs
export const add = (a, b) => {
 return a + b;
};

export const sub = (a, b) => {
 return a - b;
};

export default function () {
 console.log('calc');
}
```

```
// index.mjs
// defaultCalcは好きな名前でよい
import defaultCalc, { add, sub } from './calc.mjs';

defaultCalc(); // calc
```

ECMAScript modules: 動的読み込み（Dynamic Imports）

ECMAScript modulesの特徴的な機能にモジュールの動的読み込み（Dynamic Imports）がある

import()式にモジュールのパスを指定すると、import()式を呼び出したタイミングで初めてモジュールの読み込みが可能になる

```
import('./calc.mjs')
  .then((module) => {
    console.log(module.add(1, 2))
  })
```

ECMAScript modules: 動的読み込み（Dynamic Imports）

動的読み込みは特にフロントエンド（ブラウザ）のコードでニーズがある

ユーザーがクリックした時にしか使わない機能であれば、クリックしたタイミングで初めてダウンロードすれば、初期描画時にネットワークから取得するコストを下げることが可能

```
document.querySelector('.addButton').addEventListener('click', () => {
 import('./calc.mjs')
   .then((module) => {
     const result = module.add(1, 2);
     document.querySelector('.result').innerText = result;
   });
});
```

ECMAScript modules: 動的読み込み（Dynamic Imports）

フロントエンドではユーザーの端末やネットワークなど、実行環境を特定しきれない。またファイル数が昔に比べファイル数が非常に多いため、初期アクセス時にすべてのファイルをダウンロードすることはコストが高い。そのためパフォーマンスチューニングをするためにも動的読み込みは欠かせない

Node.jsではサーバーの立ち上がりの速度に起因する問題は、設計によって回避がしやすい。そのため初期構築（=サーバーが立ち上がるまで）の速度は、フロントエンドほど重要ではない

そのため、Node.jsでは動的読み込みの登場頻度はあまり高くない

モジュールの使い分け

モジュールの使い分け

Node.jsのモジュールはデフォルトでは、拡張子がjsもしくは.cjsのファイルはCommonJS modules扱い。.mjsのファイルはECMAScript modules扱いとなる

package.jsonというファイルを配置し、typeプロパティを設定することで、配置されたディレクトリ以下の .js ファイルのモジュールタイプを設定できる

type: “module” -> ディレクトリ以下の .js を ESM として扱う
type: “commonjs” -> ディレクトリ以下の .js を CJS として扱う

```
// package.json
{
 "type": "module"
}
```

モジュールの使い分け

ひとつのプロジェクトではどちらかに統一するのがよい

今後はESMで記述されたコードが多くなっていくことが想定される

新規で構築するならばpackage.jsonを利用してESMで記述するのが、現時点では問題が少なく将来的な流れにも沿いやすい

ライブラリの場合は、CJSとESMの両方に対応した公開方法もあり、広く使われているライブラリでは両方に対応しているものも多い

標準モジュール（Core API）

標準モジュール（Core API）

Node.jsは、V8エンジンと標準のJavaScriptにはない各種実装（Core API）を組み合わせた実行環境

モジュール名	利用用途
fs	ファイル作成/削除などの操作
path	ファイルやディレクトリパスなどのユーティリティ機能
http/https	http/httpsサーバーやクライアントの機能
os	CPUの数やホスト名などOS関連情報の取得
child_process	子プロセス関連の機能
cluster,worker_threads	マルチコア、プロセスを利用するため機能
crypto	OpenSSLのハッシュや暗号・署名や検証などの暗号化の機能

標準モジュール（Core API）

Core APIは相対パスなどを指定せずに読み込み可能

```
require('fs');
import * as fs from 'fs'
```

また、近年では外部のモジュールと区別するために“node:”というprefixがついていることも多い
（昔からあるモジュールはprefixなしでも利用できる）

```
require('node:fs');
import * as fs from 'node:fs'
```

標準モジュール（Core API）

fsモジュールを利用して自分自身を読み込んで表示するサンプル

__filenameはCJSでしか利用できない自身のファイルパスが入る独自変数

import.meta.filenameはNode.jsのESMでしか利用できない自身のファイルパスが入る独自変数

```
// index.js
const fs = require('node:fs');

fs.readFile(__filename, (err, data) => {
 console.log(data);
});
```

```
// index.mjs
import { readFile } from 'node:fs'

readFile(import.meta.filename, (err, data) => {
 console.log(data);
});
```

npmと外部モジュール

npmと外部モジュール

Node.jsにバンドルされているnpmを利用することで、https://npmjs.com に公開されている数多くのモジュールを利用できる

また、npmはpackage.jsonやpackage-lock.jsonといったファイルでアプリケーションで利用されている外部モジュールのバージョンを管理し、アプリケーションの再構築を容易にする機能を提供する

package.json/package-lock.json

npmに公開されているモジュールを利用・管理するためにはpackage.jsonというファイルが必要になる

package.jsonの最小構成は private プロパティのみをセットしたjsonファイル

npmコマンドで初期化も可能

```
$ mkdir test_npm
$ cd test_npm
# package.jsonの作成
$ echo '{ "private": true }' >> package.json
$ cat package.json
{ "private": true }
```

```
$ npm init -y
```

package.json/package-lock.json

HTTPリクエストを行うundiciを利用する例
https://www.npmjs.com/package/undici

package.jsonが存在するディレクトリで、installコマンドで利用するモジュールを指定する

インストール後、package.json/package-lock.jsonが更新され、バージョンや依存関係などが記録される

```
# undiciのインストール
$ npm install undici --save
```

```
{
 "private": true,
 "dependencies": {
   "undici": "^6.19.4"
 }
}
```

package.json/package-lock.json

インストールしたnpmモジュールの実体はpackage.jsonがあるディレクトリのnode_modulesディレクトリに保存される

package-lock.jsonはnode_modulesディレクトリを復元するために必要となるファイル

npmに公開されているモジュールは、さらに他のモジュールに依存していることが多い。またOSの種類などによってもnode_modules以下の構造やファイルなどが変化する。
package-lock.jsonはそれらを記録したスナップショット

```
directory/
├── node_modules/
├── package.json
└── package-lock.json
```

package.json/package-lock.json

node_modulesはインストールした環境や状況によって構造が変化する。そのため、別のマシンにフォルダを移動しても動作しない可能性がある。

Gitではpackage-lock.jsonをコミットし、node_modulesフォルダをコミットしてはいけない

アプリケーションを再構築する際はpackage.json/package-lock.jsonが存在するフォルダでinstallコマンドを実行すると、node_modulesフォルダが再構築される

```
# 依存モジュールをすべて再取得
$ npm install
```

セマンティックバージョニング（semver）

npmのパッケージはセマンティックバージョニング（Semantic Versioning、semver）の仕様にしたがってバージョニングすることが推奨されている

セマンティックバージョニングでは、Major.Minor.Patchの規則で、1.0.0や0.13.1のように3つの数値をドットで区切る記法が利用される

名称	ルール
Major	バグ/機能追加を問わず下位互換性を損なうリリース時にインクリメントされ、 MinorとPatchを0にリセットする
Minor	下位互換性のある機能追加のリリース時にインクリメントされ、 Patchを0にリセットする
Patch	下位互換性のあるバグ修正のリリース時にインクリメントされる

セマンティックバージョニング（semver）

semverの差分を見ることで、下位互換性のない破壊的な変更が加えられているかを知る、ある程度の目安となる

npmのパッケージ更新の挙動もsemverを基準に動作する

すべてのモジュールがsemverに従っているとは限らない（バージョンの上げ方は管理者に委ねられている）

モジュールの利用

package.jsonのdependenciesに記録されたモジュールは記録された名前で読み込みできる。

CJSの場合はrequire/ESMの場合はimportで記述する

undiciはどちらにも対応している

```
// index.js
const { request } = require('undici');

request('https://www.yahoo.co.jp')
  .then((res) => {
    return res.body.text()
  })
  .then((body) => {
    console.log(body);
  });
```

モジュールの利用

undiciの実体はnode_modulesディレクトリに存在するので、node_modulesディレクトリを削除するとこのコードは動かなくなる

GitHub等からcloneしてきたばかりのコードなどはnode_modulesディレクトリがないため、インストールコマンド（npm install）を呼ぶ必要がある

```
$ rm -rf node_modules
$ node index.js
node:internal/modules/cjs/loader:936
 throw err;
 ^

Error: Cannot find module 'undici'
```

課題

- npmからundiciをインストールして下記のコードを実行し、結果を確認する
- ESMに書き直して実行する

```
// index.js
const { request } = require('undici');

request('https://www.yahoo.co.jp')
 .then((res) => {
   return res.body.text()
 })
 .then((body) => {
   console.log(body);
 });
```

非同期処理 / フロー制御

同期処理と非同期処理

Node.jsはイベントループがあることで、シングルプロセス/シングルスレッドでも、多数のリクエストを効率的に処理が可能

逆に言うと、イベントループを長時間停止させるコードは、多数のリクエストを受ける際にパフォーマンスを発揮しにくくなる

Node.jsでパフォーマンスの高いアプリケーションを作成する上では、イベントループをいかに長時間停止させないかを意識することが重要

イベントループを停止させる処理は、シンプルに分解すると同期処理（非同期以外の処理）

同期処理と非同期処理

Node.jsにおける非同期処理は、libuvから提供されている
https://github.com/libuv/libuv

libuvはクロスプラットフォーム向けの非同期処理を提供するライブラリ

libuvというクッションを利用することで、Node.jsはLinuxやmacOS、Windowsなどのクロスプラットフォームで非同期な動作の担保が可能になっている

つまり、イベントループを停止させる同期処理とは『libuvで提供されていないもの」が該当する

同期処理と非同期処理

Node.jsはV8エンジンと標準のJavaScriptにはない各種実装（Core API）を組み合わせたランタイム

Core APIは非同期処理がベース。Core APIの非同期処理はlibuvから提供されている

逆にV8が解釈している標準のJavaScript、たとえばループなどの文法や、JSON.parse/Array.forEachなどは基本的に同期処理

同期処理と非同期処理

同期: V8（JavaScriptの標準文法）

非同期: libuv

機能	実装	同期/非同期
JSON operation	V8	同期
Array operation	V8	同期
File I/O	libuv	非同期（Sync関数を除く）
Child process	libuv	非同期（Sync関数を除く）
Timer	libuv	非同期（※setTimeout等はブラウザでも動くが、ECMAScriptには含まれていない）
TCP	libuv	非同期

Node.jsにおける非同期処理

Node.jsで利用される非同期のフロー制御パターンは大きく分けて4つ

- Callback
- Promise
- async/await
- EventEmitter/Stream

Callback

Callback

Callback（コールバック）はJavaScriptの非同期制御で一番古くから利用されている方式

Callbackは非同期なコードを表す優れたインターフェース

```
const { readFile } = require('fs');
console.log('A');

readFile(__filename, (err, data) => {
 console.log('B', data);
});

console.log('C');
```

Callback

Callbackは「処理が終わった時に呼び出される関数を登録する」インターフェース

そのため、Callbackでは処理を直列にする（フロー制御）ためにはCallbackの中でCallbackを呼ぶ必要がある

次の仕様を満たすコードを考える

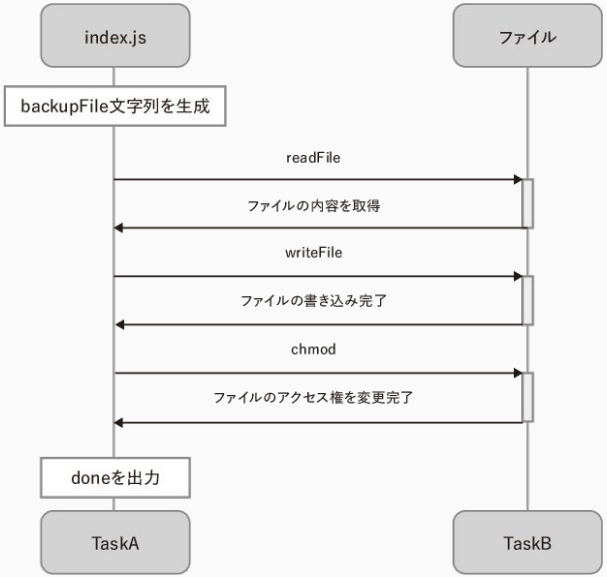
- ファイル自身を読み込み
- ファイル名をフォーマットして別名で書き込み
- バックアップしたファイルをReadOnlyにする

Callback

- ファイル自身を読み込み
  - readFile

- ファイル名をフォーマットして別名で書き込み
  - writeFile

- バックアップしたファイルをReadOnlyにする
  - chmod




```
const { readFile, writeFile, chmod } = require('fs');

const backupFile = `${__filename}-${Date.now()}`;

readFile(__filename, (err, data) => {
 if (err) {
   return console.error(err);
 }
 writeFile(backupFile, data, (err) => {
   if (err) {
     return console.error(err);
   }
   chmod(backupFile, 0o400, (err) => {
     if (err) {
       return console.error(err);
     }
     console.log('done')
   });
 });
});
```

Callback

Callbackは処理が終了したタイミングで実行されるため、直列に処理をつなぐ場合はreadFile → writeFile → chmodとネストが深くなる

ネストがどんどんと深くなるコードはCallback Hellと呼ばれる

Callback Hellは特にGitなどによってコードを管理する現在ではコードレビューのコストを引き上げがち

```
const { readFile, writeFile, chmod } = require('fs');

const backupFile = `${__filename}-${Date.now()}`;

readFile(__filename, (err, data) => {
 if (err) {
   return console.error(err);
 }
 writeFile(backupFile, data, (err) => {
   if (err) {
     return console.error(err);
   }
   chmod(backupFile, 0o400, (err) => {
     if (err) {
       return console.error(err);
     }
     console.log('done')
   });
 });
});
```

課題: Callback + ループ

次のコードを実行する

作成されたdata.txtの中身を確認する

```
const { writeFile } = require('fs');

for (let i = 0; i < 100; i++) {
  const text = `write: ${i}`;
  writeFile('./data.txt', text, (err) => {
    if (err) {
      return console.error(err);
    }
    console.log(text);
  });
}
```

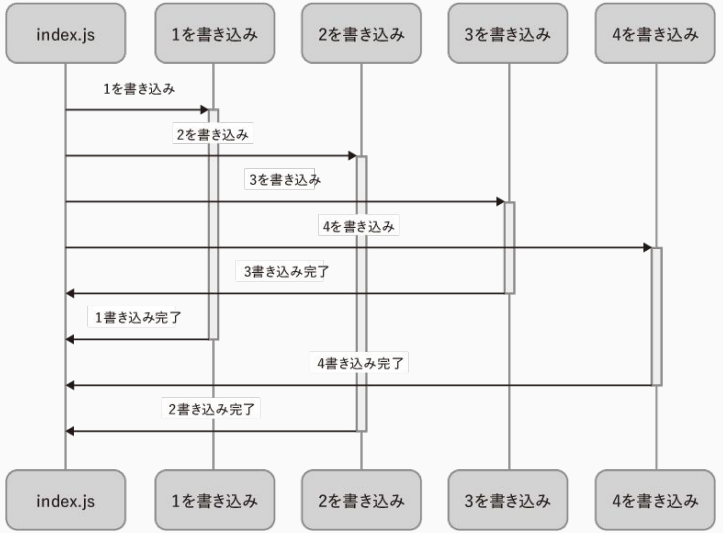
Callback + ループ

fs.writeFileを100回呼ぶことには成功しているが前の結果を待ち受けることができていない

0番目の処理の完了をまたずに1番目の処理が開始するため、書き込みが完了するタイミングの順序は保証されない

Callbackで順序保障するループの実装には再帰処理等が必要になる

```
$ node index.js
write: 1
write: 4
write: 5
...
write: 99
write: 48
write: 50
```



Node.jsにおけるCallback

Core APIに実装されているCallbackのAPIには慣例がある

- APIの最後の引数がCallback
- 処理完了時に一度だけCallback関数が呼び出される
- Callbackの第一引数がエラーオブジェクト

Callbackのエラーハンドリングを行う際には必ず第一引数の空チェックが必要

```
const { readFile } = require('fs');

readFile(__filename, (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

Node.jsにおけるCallback

try-catchでCallback内のエラーを捕捉できない

Callbackの処理をネストしていく場合は、必ずすべてのCallbackごとにエラーのnullチェックをする必要がある

```
const { readFile } = require('fs');

try {
  readFile(__filename, (err, data) => {
    console.log(data);
  });
} catch (err2) {
  // Callbackの引数に入るエラーは補足できない
  console.error(err2);
}
```

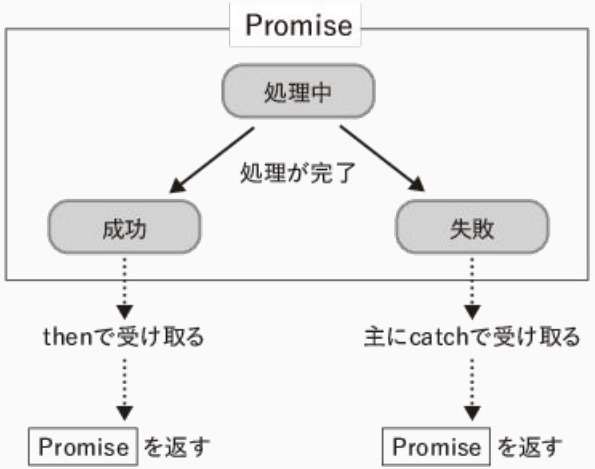
Promise

Promise

Callback自体は優れたインターフェースだが、ネストが深くなりがち、エラーハンドリングなどの弱点もある

それらの弱点を解消した非同期処理を実現したものとして Promiseが登場した

Promiseは成功か失敗を返すオブジェクト
Promiseオブジェクトが生成されたタイミングでは状態が決まらず、非同期処理が完了したタイミングでどちらかの状態に変化する



Promise

Promiseはresolve（成功）とreject（失敗）時に呼び出す関数を引数にもつ関数をコンストラクタとして生成する

```
const promiseFunc = new Promise((resolve, reject) => {
 // 非同期で行う処理を記述する
 // エラーが起きた時はrejectを呼び出す
 if (errorが起きた時) {
   return reject(エラー内容);
 }
 // 成功した時はresolveを呼び出す
 resolve(成功時の内容);
});
promiseFunc.then(成功=resolve時に実行する関数)
promiseFunc.catch(失敗=reject時に実行する関数)
```

Promise

thenやcatchはつなぐ（チェイン）ことが可能
これによりネストを深くせずにフロー制御と包括的なエラーハンドリングが可能になる

ループや条件分岐にはCallback同様課題がある

```
// thenで成功時をつなげる、失敗時はcatchに飛ぶ
promiseA()
  .then((data) =>promiseB(data))
  .then((data) =>promiseC(data))
  .catch((err) => console.log(err))
```

CallbackのPromise化

Callbackによる非同期処理はCallbackをPromiseオブジェクトでラップすることによってPromise化が可能

```
const { readFile } = require('fs');

const readFileAsync = (path) => {
  return new Promise((resolve, reject) => {
    readFile(path, (err, data) => {
      if (err) {
        reject(err);
        return;
      }
      resolve(data);
    });
  });
};
```

CallbackのPromise化

現在のCoreAPIは基本的にPromiseのインターフェースが実装されている

```
const { readFile } = require('fs');
const readFileAsync = (path) => {
  return new Promise((resolve, reject) => {
    readFile(path, (err, data) => {
      if (err) {
        reject(err);
        return;
      }
      resolve(data);
    });
  });
};
```

```
const { readFile } = require('fs/promises');

const readFileAsync = readFile;
```

https://nodejs.org/api/fs.html#promises-api

CallbackのPromise化

```
const { readFile, writeFile, chmod } = require('fs');

const backupFile = `${__filename}-${Date.now()}`;

readFile(__filename, (err, data) => {
  if (err) {
    return console.error(err);
  }
  writeFile(backupFile, data, (err) => {
    if (err) {
      return console.error(err);
    }
    chmod(backupFile, 0o400, (err) => {
      if (err) {
        return console.error(err);
      }
      console.log('done')
    });
  });
});
```

```
const { readFile, writeFile, chmod } = require('fs/promises');

const backupFile = `${__filename}-${Date.now()}`;

readFile(__filename)
  .then((data) => {
    return writeFile(backupFile, data);
  })
  .then(() => {
    return chmod(backupFile, 0o400);
  })
  .catch((err) => {
    console.error(err);
  });
```

Promise

PromiseはCallbackと比較して、ネストが深くなりにくくエラーハンドリングがしやすい
文法的な優位性がある

ループや条件分岐など解消できていない部分も存在する

async/await

async/await

Promiseの弱点をさらに記述しやすくする文法としてasync/awaitが登場

Promiseを利用した非同期処理を同期的な見た目で記述できる

asyncをつけた関数内でPromiseを返す式をawaitで待ち受けることができる

```
async function someFunc() = {
  const foo = await Promiseを返す式;
  const bar = await Promiseを返す式; // 前のawaitが完了するまで実行されない
  await Promiseを返す式;
};

const someFuncArrow = async () =>  {
  await Promiseを返す式;
};
```

async/await

```
const { readFile, writeFile, chmod } =
require('fs/promises');

const backupFile = `${__filename}-${Date.now()}`;

readFile(__filename)
  .then((data) => {
    return writeFile(backupFile, data);
  })
  .then(() => {
    return chmod(backupFile, 0o400);
  })
  .catch((err) => {
    console.error(err);
  });
```

```
const { readFile, writeFile, chmod } =
require('fs/promises');

const backupFile = `${__filename}-${Date.now()}`;

async function main() {
  const data = await readFile(__filename);
  await writeFile(backupFile, data);
  await chmod(backupFile, 0o400);
}

main()
  .catch((err) => {
    console.error(err);
  });
```

async/await

async/awaitはPromiseのシンタックスシュガー

async関数は実行するとPromiseを返す = async/awaitはPromiseと相互に呼び出しが可能

main関数の最終的な実行はPromiseのインターフェースになる

```
const { readFile, writeFile, chmod }
  = require('fs/promises');

const backupFile = `${__filename}-${Date.now()}`;

async function main() {
  const data = await readFile(__filename);
  await writeFile(backupFile, data);
  await chmod(backupFile, 0o400);
}

main()
  .catch((err) => {
    console.error(err);
  });
```

async/await

Promiseに比べてフロー制御が同期的な見た目で記述できる

Promiseで返される値を変数へ格納できる

コード上は同期処理的にわかりやすく、実際には非同期に処理される(I/Oをブロッキングしない)動作が実現できる

ループや条件分岐も同期的に記述可能

```
async function main() {
  const data = await
readFile(__filename);
  await writeFile(backupFile, data);
  await chmod(backupFile, 0o400);
}
```

```
async function main() {
  for (let i=0; i < 10; i++) {
    const flag = await asyncFunction();
    if (flag) {
      break;
    }
  }
}
```

ストリーム処理

ストリーム処理

Node.jsにはasync/awaitのような非同期フロー制御の他に、イベント駆動型の非同期フロー制御（ストリーム処理）が存在する

- 非同期フローの制御
  - Callback
  - Promise
  - async/await
- イベント駆動型の非同期フロー
  - ストリーム処理（EventEmitter/Stream）

ストリーム処理

Callbackでは「処理の完了」というひとつのイベントにのみ処理を行うもの

イベント駆動型のフロー制御では「処理の開始」「処理の途中」「処理の終了」「エラー発生時」といったさまざまなタイミングで処理を行う

図

ストリーム処理

Callbackのような一度きりの処理に比べて、データの逐次処理が可能

メモリの効率的な利用やイベントループの停止時間を分割するといったケースで有効

次のような処理は、ストリーム処理を行う代表的な例

- HTTPリクエスト/レスポンス
- TCP
- 標準入出力

ストリーム処理

イベント駆動型の処理はNode.jsのコアにあるEventEmitter（やそれを継承したStream）と呼ばれる基底クラスを継承し実装されている

```
$ node index.js
on myevent: one
on myevent: two
```

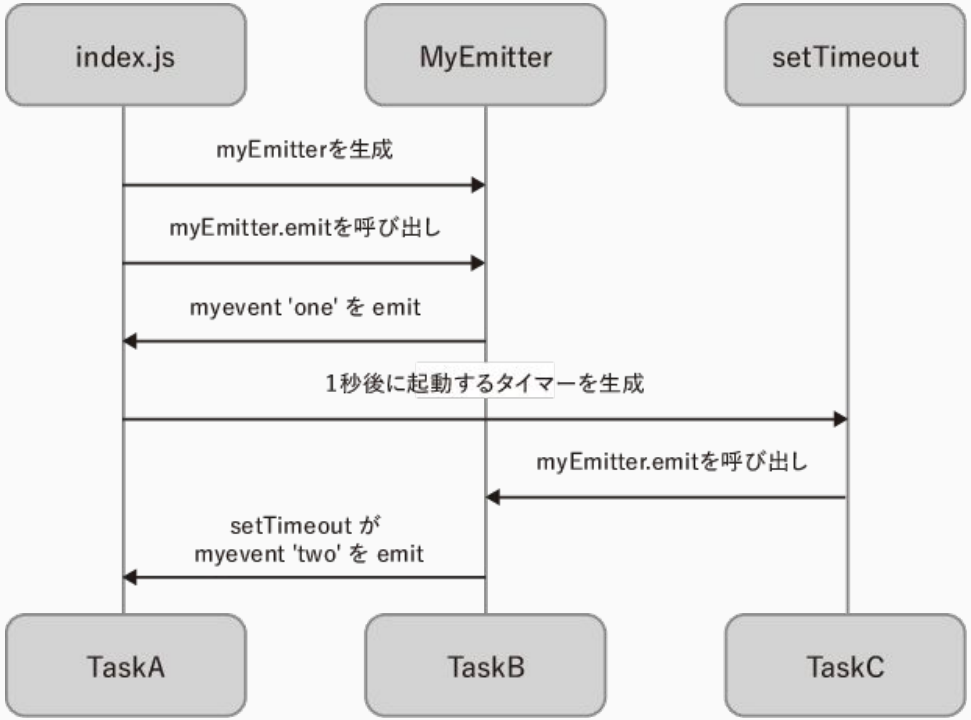
```
const EventEmitter = require('events');

// EventEmitterの基底クラスを継承して独自イベントを扱うEventEmitterを定義
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

// myeventという名前のeventを受け取るリスナーを設定
myEmitter.on('myevent', (data) => {
 console.log('on myevent:', data);
});
// myeventを発行
myEmitter.emit('myevent', 'one');

setTimeout(() => {
 // myeventを発行
 myEmitter.emit('myevent', 'two');
}, 1000);
```

ストリーム処理




```
const EventEmitter = require('events');

// EventEmitterの基底クラスを継承して独自イベントを扱う
EventEmitterを定義
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

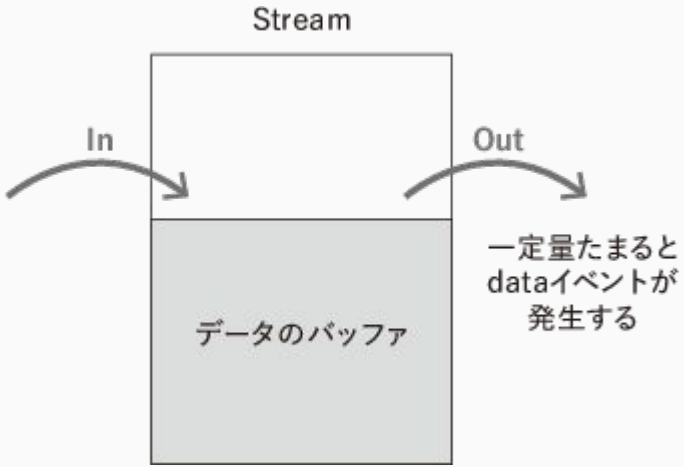
// myeventという名前のeventを受け取るリスナーを設定
myEmitter.on('myevent', (data) => {
 console.log('on myevent:', data);
});
// myeventを発行
myEmitter.emit('myevent', 'one');

setTimeout(() => {
 // myeventを発行
 myEmitter.emit('myevent', 'two');
}, 1000);
```

ストリーム処理

EventEmitterは「何度も」「細切れ」に起きる非同期なイベントを制御するための実装

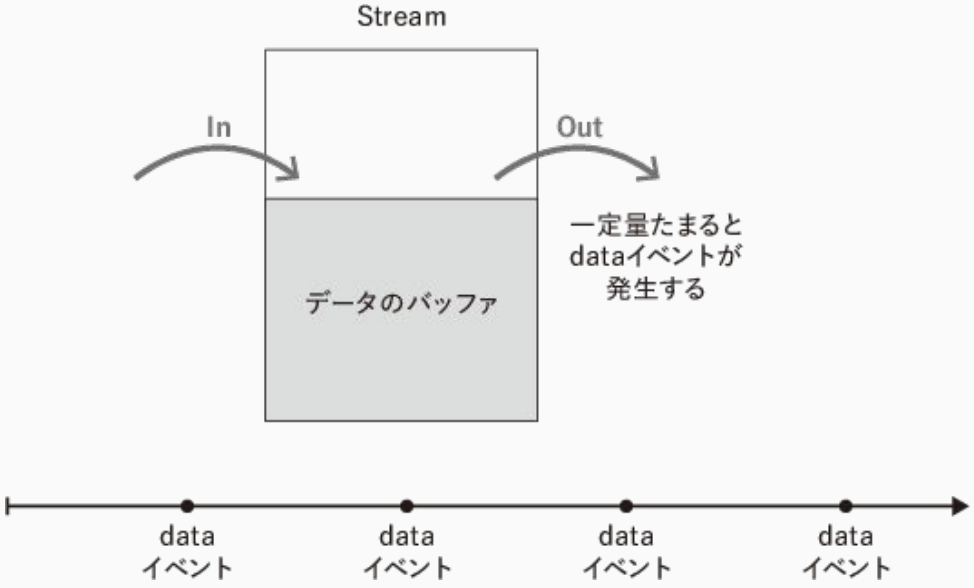
この特性を利用し、断続的なイベントをより扱いやすいようにEventEmitterを継承してつくられたインターフェースがStream



ストリーム処理

StreamはEventEmitterにデータをため込む内部バッファを組み込んだもののようなイメージ

内部バッファに一定量のデータがたまるとイベントが発生する



ストリーム処理

Streamオブジェクトどうしを連結することができる

例) たまったデータを別の形式に変換するStreamを途中に接続する

すべてのデータがたまり切る前に変換可能な状態になったものから処理を始める、といった動作が可能になる

Streamを利用することで、断続的に発生するイベントの制御、データの流量の調整、変換処理など連続するデータの流れを効率的に扱うことが可能

ストリーム処理

Node.jsでは次の4つのStreamが処理のベース

Streamの種別	どういったStreamか
Writable	データの書き込みに利用する（ex: fs.createWriteStream）
Readable	データを読み取りに利用する（ex: fs.createReadStream）
Duplex	書き込み/読み取りの両方に対応（ex: net.Socket）
Transform	Duplexを継承し、読み書きしたデータを変換する （ex: zlib.createDeflate）

ストリーム処理

Node.jsはシングルスレッド/シングルプロセスで動作するモデル

アプリケーションの設計上、I/Oなどの処理はなるべく細切れにすることがパフォーマンス上重要

そのような「処理を細かく、何度も分割する」という点でストリーム処理は優れている

Callbackは完了時に一度呼ばれることに対し、ストリーム処理はひとつの処理に対して何度も処理が発生するという違いがポイント

ストリーム処理

HTTP処理などはストリーム処理の代表例

HTTPリクエストを送信するサンプル

resはレスポンスを表すストリームオブジェクト

‘data’と’end’イベントを受け取るリスナーを設定

```
const https = require('node:https');
// サーバーに対してリクエストするオブジェクトを生成
const req = https.request('https://www.yahoo.co.jp', (res) => {
  // 流れてくるデータをutf8で解釈する
  res.setEncoding('utf8');

  // dataイベントを受け取る
  res.on('data', (chunk) => {
    console.log(`body: ${chunk}`);
  });

  // endイベントを受け取る
  res.on('end', () => {
    console.log('end');
  });
});
// リクエスト送信開始
req.end();
```

ストリーム処理

リスナーはイベントが発生するたびに何度も呼び出される

dataはレスポンスから一定量データを受け取るたびに発生する

もし、一度だけデータを取得する場合は、データをすべてメモリ上にのせなければならない

ストリーム処理は逐次的にデータを扱うため、巨大なデータであっても一度に利用するリソースが少なくすむ

また、dataイベントが発生するまでの間は Node.jsは仕事がないため、別の処理を進めることができる

```
// dataイベントを受け取る
res.on('data', (chunk) => {
 console.log(`body: ${chunk}`);
});

// endイベントを受け取る
res.on('end', () => {
 console.log('end');
});
```

ストリーム処理

ストリーム処理は大きなデータを少ないリソースで扱う際にメリットがあるインターフェース

データをイベント単位で細切れに扱うため、イベントの間にほかの処理をはさむことができる パフォーマンスを保ちやすい

ストリーム処理のエラーハンドリング

Promise同様try-catchではエラーを捕捉できない

ストリーム処理はエラーの発生もイベントとして処理されるため、errorイベントのハンドラーをセットする

ストリーム処理のエラーハンドリングは漏れてしまうとエラーをキャッチできず、エラー発生時にプロセスがクラッシュするため注意が必要

```
stream.on('error', (err) => {
  console.error(err);
});
```

エラーハンドリング

エラーハンドリング

Node.jsのエラーハンドリングは他の動作モデルに比較しても特に重要

Node.jsでエラーハンドリングが漏れた場合、プロセスのクラッシュが発生する。プロセスがクラッシュすると、同時に処理しているすべての処理がエラーとなる

1リクエストに対し1プロセスを割り当てるモデルに比べて、シングルスレッド/シングルプロセスで複数のリクエストを受け付けるNode.jsの弱点

エラーハンドリング

	設計	エラーハンドリング
同期処理		try-catch, async/await
非同期処理	Callback	if (err) { }
非同期処理	EventEmitter (Stream)	emitter.on('error', (e) => { })
非同期処理	async/await	try-catch, .catch()

エラーハンドリング（同期）

同期コードのエラーハンドリングはシンプルにtry-catchで囲うことと、async/awaitでラップしてしまうことの2択

async関数はPromiseを返すため、上位関数でcatchによって包括的なエラーハンドリングがされるため、try-catchと同義ととらえることもできる

```
async function main() {
  JSON.parse('error str!');
}

main()
  .catch((e) => console.error(e));
```

エラーハンドリング（非同期）

Callback、EventEmitter（Stream）、async/awaitはそれぞれ下記

- Callback: エラーのnullチェック
- EventEmitter（Stream）: エラーイベントのハンドリング
- async/await: try-catchと上位関数での.catch()

Node.jsのアプリケーションは非同期処理が中心となるため、現在の環境であれば可能な限りasync/awaitに寄せた設計をするとよい

ファイル操作

ファイル操作

基礎となるファイル操作をおさえる

Node.jsにおいて‘node:fs’モジュールでファイル操作を行うAPIが提供される
https://nodejs.org/api/fs.html

‘node:fs’を利用することで、ファイルの読み書き、ディレクトリの作成削除などが可能になる

非同期モジュールは‘node:fs/promises’から提供されるPromiseのインターフェースを利用する

同期処理を行う場合はxxSyncとつくAPIを利用する

ファイル操作: ファイル読み込み

同期/非同期のファイル読み込み

- example.txtがない状態で実行しエラーが起きることを確認

- example.txtを配置した状態で実行し、内容が読み込めることを確認

```
const fs = require('node:fs/promises');

async function main() {
  const data = await fs.readFile('./example.txt', 'utf8');
  console.log(data)
}
main().catch(console.error)
```

```
const fs = require('node:fs');

try {
  const data = fs.readFileSync('./example.txt', 'utf8');
  console.log('ファイルの内容:', data);
} catch (err) {
  console.error(err);
}
```

ファイル操作: ファイル書き込み

同期/非同期のファイル書き込み

- example.txtの内容がHello World!になることを確認

- 書き込み内容を別の内容に変更して実行

```
const fs = require('node:fs/promises');
async function main() {
  await fs.writeFile(
    './example.txt',
    'Hello World!',
    { encoding: 'utf-8' }
  );
}
main().catch(console.error)
```

```
const fs = require('node:fs');
try {
  fs.writeFileSync(
    './example.txt',
    'Hello World!',
    { encoding: 'utf-8' }
  );
} catch (err) {
  console.error(err)
}
```

ファイル操作: ファイル書き込み

基本は上書きになるが、オプションを指定することで追記モードなども選択できる

https://nodejs.org/api/fs.html#file-system-flags

```
const fs = require('node:fs/promises');

async function main() {
 for (let i = 0; i < 10; i++) {
   await fs.writeFile(
       './example.txt',
       `Hello World!: ${i}\n`,
       {
         encoding: 'utf-8',
         flag: 'a' // 追記モードでファイルを開く
       }
     );
 }
}
main().catch(console.error)
```

ファイル操作: ファイル削除

同期/非同期のファイル削除

- example.txtが存在する状態でスクリプトを実行し、ファイルが削除されることを確認

```
const fs = require('node:fs/promises');

async function main() {
  await fs.unlink('./example.txt');
}
main().catch(console.error)
```

```
const fs = require('node:fs');

try {
  fs.unlinkSync('./example.txt');
} catch (err) {
  console.error(err);
}
```

ファイル操作: ファイル情報取得

同期/非同期のファイル情報取得

- example.txtがない状態で実行しエラーが起きることを確認

- example.txtを配置した状態で実行し、情報が取得できることを確認

```
const fs = require('node:fs/promises');

async function main() {
 const data = await fs.stat('./example.txt');
 console.log(data);
}
main().catch(console.error);
```

```
const fs = require('node:fs');

try {
  const data = fs.statSync('./example.txt');
  console.log(data);
} catch (err) {
  console.error(err)
}
```

パス操作

ディレクトリやファイル名を結合したり、絶対パスを取得する

文字列処理なので同期処理

CJSではモジュールにパスの情報が格納される__dirnameや__filenameといったファイル内グローバル変数が自動的に挿入される

ESMではimport.metaを拡張して同様の情報が格納される

```
const path = require('node:path');

const joinPath = path.join('./src', 'example.txt');
// src/example.txt
console.log(joinPath)

const resolvePath = path.resolve('./src', 'example.txt');
// /path/to/src/example.txt
console.log(resolvePath)
```

```
// CommonJS index.js

// /path/to/src
console.log(__dirname)
// /path/to/src/index.js
console.log(__filename)
```

```
// ESM index.mjs

// /path/to/src
console.log(import.meta.dirname)
// /path/to/src/index.mjs
console.log(import.meta.filename)
```

Webアプリケーションにおけるファイル操作の注意点

ファイルの読み書きは同時アクセス（レースコンディション）に注意して実装する必要がある

例) ユーザーA, Bのデータをdata.txtに書き込むケース

- ユーザーAのデータをdata.txtに書き込み
- ユーザーBのデータをdata.txtに書き込み
- ユーザーAがdata.txtのデータを取り出し = ユーザーBのデータが取り出される
- 同時書き込みが起きた場合はデータが壊れるケースもある

ファイルにロックをかけたりファイル名、ディレクトリ名が重複しないような設計をしなければならない（基本的にはデータベースなどのソフトウェアに任せたほうがよい）

CLIツールの作成

CLIツールの作成

CLI: Command Line Interface

CLIツールの開発を通してNode.jsアプリ開発の基礎を学ぶ

- コマンドライン引数
- 環境変数
- ファイル分割
- モジュールの導入

CLIツールの作成

MarkdownファイルからHTMLを作成する簡易的なブログ記事作成ツール

満たす仕様は以下

- オプションに応じてCLIツールの名前を表示する
- file名を指定してMarkdownファイルを読み込む
- MarkdownファイルをHTMLに変換する
- 変換したHTMLをファイルに書き出す
- オプションでアウトプットするHTMLの名前を設定する
- これらの機能をテストするコード

CLIツールの作成

作業ディレクトリとpackage.jsonの用意

npm initによって生成されたpackage.jsonにはprivateプロパティがない

誤ってnpmにコードが公開されてしまうことを防ぐために、npmに公開するモジュールでない場合にはprivateプロパティを追記しておくとよい

```
$ mkdir cli_test
$ cd cli_test
# -yオプションでnpm initを対話なしで実行
$ npm init -y
```

```
{
  "private": true, /* 追記 */
  "name": "cli_test",
  "version": "1.0.0",
  …
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "license": "ISC"
}
```

CLIツールの作成: コマンドライン引数の処理

Node.jsでコマンドライン引数を受け取るにはグローバル変数のprocessオブジェクトを利用する

processオブジェクト内のargvにコマンドライン引数が格納される

process.argvは配列で実行したコードパスや引数などが格納されている

```
// index.js
console.log(process.argv);
```

```
$ node index.js one --two=three --four
[
  '/usr/local/bin/node',
  '/home/xxx/cli_test/argv.js',
  'one',
  'two=three',
  '--four'
]
```

CLIツールの作成: コマンドライン引数の処理

process.argvをそのまま扱ってもよいが煩雑な処理になるため、Core APIを利用してより簡易に利用するとよい

```
const { parseArgs } = require('node:util');

const { values } = parseArgs({
  strict: false
});
console.log(values);
```

```
$ node index.js one --two=three --four
[Object: null prototype] { two: 'three', four: true }
```

CLIツールの作成: 課題

--nameオプションを受け取った時にpackage.jsonのnameを出力するプログラムを作成する

ヒント: fs.readFieSync、JSON.parse

```
$ node index.js --name
cli_test
# オプションが無い時は表示しない
$ node index.js
オプションがありません
```

CLIツールの作成: ひな形作成

fs.readFileSyncは同期APIでファイルを読み込む

同期APIの実行中は処理をブロックしてしまうため、利用の際にはパフォーマンスの問題が起きないよう注意する必要がある

今回のようなCLIツールの場合は実行者以外のタスクを処理しないため『I/Oがブロックされても影響を受けるユーザーが他にいない』=「同期APIを利用しても問題ない」利用ケース

```
const packageStr = fs.readFileSync(
  path.resolve(__dirname, 'package.json'),
  { encoding: 'utf-8' }
);
```

CLIツールの作成: オプションの追加

ファイルのパスを指定して、markdownファイルを読み込む機能を実装

--fileオプションでパスを指定し内容を出力するコードを追加する

```
directory/
├── article.md
├── index.js
├── package.json
└── package-lock.json
```

```
$ node index.js --file=./article.md
# タイトル

hello!

**テスト**
```javascript
const foo = 'bar';
```
```

```
# タイトル

hello!

**テスト**

```javascript
const foo = 'bar';
```
```

CLIツールの作成: オプションの追加（実装例）

```
const path = require('node:path');
const fs = require('node:fs');
const { parseArgs } = require('node:util');

const { values } = parseArgs({
 strict: false
});

if (values.name) {
  const packageStr = fs.readFileSync(
    path.resolve(__dirname, 'package.json'),
    { encoding: 'utf-8' }
  );
  const packageJson = JSON.parse(packageStr);
  console.log(packageJson.name)
  return
}
```

```
if (values.file) {
  // 指定されたMarkdownファイルを読みこむ
  const markdownStr = fs.readFileSync(
    path.resolve(__dirname, values.file),
    { encoding: 'utf-8' }
  );
  console.log(markdownStr);
  return
}

console.log('オプションがありません')
```

CLIツールの作成: ファイル分割

このままだとコードが長くなりすぎるため、ファイル分割を行いコードの見通しをよくする

libディレクトリを作成し、機能単位でファイルを分割する

```
directory/
├── lib/
│   ├── name.js
│   └── file.js
├── article.md
├── index.js
├── package.json
└── package-lock.json
```

```
// lib/file.js
const fs = require('node:fs');

exports.readMarkdownFileSync = (filepath) => {
  const markdownStr = fs.readFileSync(
    filepath,
    { encoding: 'utf-8' }
  );
  return markdownStr
}
```

CLIツールの作成: 課題

オプションが動作するようコードをlib/file.jsとlib/name.jsに分割する

```
$ node index.js --file=./article.md
# タイトル

hello!

**テスト**
```javascript
const foo = 'bar';
```
```

```
const path = require('node:path');
const { parseArgs } = require('node:util');
const { getPackageName } = require('./lib/name');
const { readMarkdownFileSync } = require('./lib/file');

const { values } = parseArgs({ strict: false });

if (values.name) {
  const name = getPackageName();
  console.log(name);
  return
}

if (values.file) {
  const filePath = path.resolve(__dirname, values.file);
  const markdownStr = readMarkdownFileSync(filePath);
  console.log(markdownStr);
  return
}

console.log('オプションがありません')
```

CLIツールの作成: 課題

コード例

```
// lib/name.js
const path = require('node:path');
const fs = require('node:fs');

const packageStr = fs.readFileSync(
  // package.jsonが1階層上になったので相対パスで一つ上に上がる
  path.resolve(__dirname, '../package.json'),
  { encoding: 'utf-8' }
);
const packageJson = JSON.parse(packageStr);

exports.getPackageName = () => {
  return packageJson.name;
};
```

CLIツールの作成: ファイル分割

readMarkdownFileSyncに渡す引数をファイル名ではなく絶対パスとしている

readMarkdownFileSyncはlibディレクトリにあるため、ファイル名だけ渡した場合、どのディレクトリにあるファイルを読み込めばよいのか関数側はわからない

ファイル名だけを渡す場合「暗黙的に実行ディレクトリに依存している」関数となる

暗黙的な依存はテストを書きにくくする

したがって、関数の呼び出し元でパスを指定する方が、より暗黙的な依存が少なく、保守性の高いコードであると言える

CLIツールの作成: モジュールの導入

「Markdownファイルからhtmlを生成する」機能の作成

npmモジュール（ライブラリ）を導入し、これらの機能を実現する

npmに公開されているモジュールの中から、適切なモジュールを選択することもNode.jsでアプリケーション開発をする上で大切なスキル

今回はmarkedを利用して機能を実装する

https://www.npmjs.com/package/marked

```
$ npm install marked
```

CLIツールの作成: モジュールの導入

ファイルの吐き出し先を指定するoutオプションを加える

markdown形式の文字列を受け取りhtmlを作成/書き込みをするwriteHtml関数をlib/file.jsに加える

正しくhtmlファイルが作成されることを確認する

```
$ node index.js --file=./README.md --out=./index.html
```

```
const fs = require('node:fs');
const { marked } = require('marked');

exports.readMarkdownFileSync = (filepath) => {
  const markdownStr = fs.readFileSync(
    filepath,
    { encoding: 'utf-8' }
  );
  return markdownStr
}

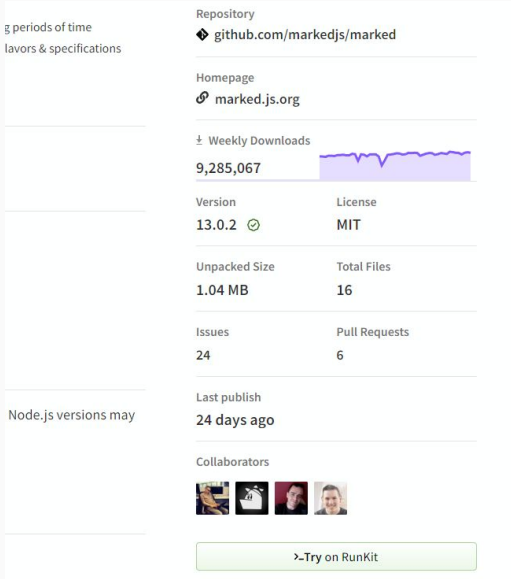
exports.writeHtml = (markdownStr, outPath) => {
  // markdown文字列をhtmlに変換
  const html = marked.parse(markdownStr);
  ……
}
```

CLIツールの作成: モジュールの導入

適切なモジュールの選択はアプリケーション開発にとって非常に大切なポイント

ただ要件を満たすかどうかだけではなく、バグやセキュリティissueなどの適切なアップデートがなされているか等、Node.jsやアプリケーションのアップデート時に足枷にならないモジュールの選定が必要

- npmのLast publishが古すぎないか
- GitHubのissueやPull Requestへの反応
- コミット履歴



Node.jsのテスト

アプリケーションを長期的に保守するためには、仕様や品質などを担保するテストが欠かせない

ドキュメントや手動テストで担保する範囲もあるが、アプリケーションが大きくなるにつれテスト範囲の拡大や、特定条件でのみ起きる異常に対応するケースなど、手動だけではテストが行えないケースが生じる

近年のアプリケーション開発ではテストコードによる自動テストがほぼ必須

テスト: 標準モジュールを利用したテスト

Node.jsのCore APIにはアサーション関数を提供するassertモジュールがある

assertモジュールはその名の通りアサーション（正当性を検証する）関数を提供する標準モジュール

値を比較する際にはassert.strictEqualなどがよく利用される

```
// sample.test.js
const assert = require('node:assert');

// 第一引数と第二引数が厳密に等しい（===）ことを検証する
assert.strictEqual(1 + 2, 3, '「1 + 2 = 3」である');
```

テスト: 標準モジュールを利用したテスト

テストコードを実行し、結果を確認する

assertの内容を変更し、エラーが出ることを確認する

```
$ node sample.test.js
```

```
assert.strictEqual(1 + 2, 2, '「1 + 2 = 3」である');
```

テスト: 標準モジュールを利用したテスト

assertモジュールには、オブジェクトを比較する
assert.deepStrictEqualなどもある

ここで紹介した2つに共通するキーワードはstrict

strictのない関数もあるが、厳密でない比較(==)になる

テストでは仕様を明確にすべきため、通常は厳密な比較を利用が推奨される

```
assert.equal(1, '1', '数値と文字列の比較'); // テストが通る
assert.strictEqual(1, '1', '数値と文字列の比較（strict）'); // NG
```

```
const assert = require('node:assert');

const obj1 = {
  a: {
    b: 1
  }
};
const obj2 = {
  a: {
    c: 1
  }
};
assert.deepStrictEqual(
  obj1,
  obj2,
  'オブジェクトが等しい'
);
```

テスト: テストランナー

assertモジュールのみでもテストは記述可能

アプリケーションのテストを行う場合、順番にテストを実行したり、テストごとに共通の前処理を行ったりというシーンは多くある

そういったテストでよくあるユースケースをまとめて省力化したAPIを提供しているものがテストランナー

Jestやvitest, mochaなどが利用されてきたが、Node.jsにも組み込みのテストランナーが提供されている

https://nodejs.org/api/test.html

テスト: テストランナー

Node.jsの組み込みテストランナーは--testオプションで起動できる

デフォルトではtestがつくファイル、ディレクトリ名が対象

- **/*.test.?(c|m)js
- **/*-test.?(c|m)js
- **/*_test.?(c|m)js
- **/test-*.?(c|m)js
- **/test.?(c|m)js
- **/test/**/*.?(c|m)js

```
// sample.test.js
const test = require('node:test');
const assert = require('node:assert');

test('1 + 2 = 3', () => {
  assert.strictEqual(1 + 2, 3, '1 + 2が3になる');
});
```

```
$ node --test
✔ 1 + 2 = 3 (1.337192ms)
ℹ tests 1
ℹ suites 0
ℹ pass 1
ℹ fail 0
ℹ cancelled 0
ℹ skipped 0
ℹ todo 0
ℹ duration_ms 61.411508
```

テストランナー

lib/file.jsにテストを追加し、実アプリケーションでのテストの書き方を学ぶ

lib/file.jsと同じ階層にlib/file.test.jsを配置しテストを記述

この状態ではテストを実行すると、test.mdファイルが存在しないため
readMarkdownFileSyncはエラーとなりテストが失敗する

```
const test = require('node:test');
const {
  readMarkdownFileSync
} = require('./file');

test('readMarkdownFileSyncが正しく動作する', () => {
  readMarkdownFileSync('test.md');
});
```

テストランナー

テスト用のシードファイルをtest.mdをfixtures/test.mdに用意

実行結果をassert.strictEqualで比較し、ファイルの中身が読み込めているテストを記述する

```
const test = require('node:test');
const assert = require('node:assert');
const path = require('node:path');
const { readMarkdownFileSync } = require('./file');

test('readMarkdownFileSyncが正しく動作する', () => {
  const str = readMarkdownFileSync(path.resolve(__dirname, '../fixtures/test.md'));
  assert.strictEqual(str, '**bold**', '読み込んだ文字列がfixtureの内容と等しいか比較');
});
```

```
directory/
├── fixtures/
│   └── test.md
├── lib/
│   ├── name.js
│   └── file.js
├── article.md
├── index.js
├── package.json
└── package-lock.json
```

テストのポイント

「テストが書きやすいコードか」という観点は非常に大切

readMarkdownFileSyncの引数がパスではなくファイル名だけだった場合「関数が暗黙的にディレクトリ構成に依存する」

将来的の修正でバグを産む可能性や、今回のようにfixtureを別のディレクトリに配置できない

テストや保守性のために暗黙的な依存をなるべく少なくすることを意識する

- inに対してoutが一意に決まる関数
- 副作用が少ない関数

Webサービス
開発基礎

Webサービス開発基礎

CLIのつくり方を通して、Node.jsプロジェクトの始め方からテストの方法を学んだ

Node.jsが得意とするネットワーク処理を理解するため、Webサーバーの開発に触れる

ライブラリの扱い方を多く扱うが、ライブラリの使い方ではなく、なぜその機能が必要なのかを理解する

Webサービス開発基礎

Webサーバー: HTTPを介してコンテンツをユーザーに配信するアプリケーション

静的: 配信時点で固定化されたファイルをユーザーへ配信する

動的: アクセスに応じてデータベース等から取得した値を利用し変換したコンテンツ(HTMLなど)をユーザーへ配信する

Node.jsを利用した動的Webサーバーの開発を通して、Webサービスの基礎を学習する

Webサービス開発基礎

ブラウザでサーバーから情報を取得するには下記の情報が必要になる

https://www.yahoo.co.jp:443/
- https: プロトコル
- www.yahoo.co.jp: hostname
- 443: port
- / : path

実際にブラウザがHTMLを表示する際の挙動は、宛先（www.yahoo.co.jp）からhttpsプロトコルを利用し443ポート（httpsのデフォルト）にアクセスしルートパス（/）で配信されているHTMLを取得

Webサービス開発基礎

http://localhost:3000

localhostはマシン自身を示すhostname

3000番ポートを利用しhttpプロトコルで配信されているローカルなwebサーバーにGETメソッドでアクセスする

上記のアクセスを受け取り動作するシンプルなサーバー

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

http
  .createServer((req, res) => {
    if (req.url === '/' && req.method === 'GET') {
      res.statusCode = 200;
      res.write('hello world\n');
      res.end();
    } else {
      res.statusCode = 404;
      res.write('Not Found\n');
      res.end();
    }
  })
  .listen(3000);
```

Webサービス開発基礎

Node.jsの標準モジュールだけでもHTTPサーバーの構築は可能

しかし標準モジュールはかなりローレベルなAPI
ルーティングやメソッド（GET/POST）別の実装など、アプリケーション部分以前に実装するものが多くある

HTTPサーバーを作成する用途では、外部モジュールを導入するのがよい

今回は古くから利用されているExpressで説明する

https://expressjs.com/

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

http
  .createServer((req, res) => {
    res.write('hello world\n');
    res.end();
  })
  .listen(3000);
```

Webサーバー基礎

ExpressはNode.js登場初期から広く利用されているWebフレームワーク
非常にミニマムなつくりだがWebサーバーとして必要十分な機能をもつ

Node.jsで動作するフレームワークもExpressを下敷きに実装されたものが多くある

```
$ mkdir server_test
$ cd server_test
$ npm init -y
$ npm install express
```

```
const express = require('express');
const app = express();

// GET '/' （トップ）アクセス時の挙動
app.get('/', (req, res) => {
  res.status(200).send('hello world\n');
});

// ポート: 3000でサーバーを起動
app.listen(3000, () => {
  // サーバー起動後に呼び出されるCallback
  console.log('start listening');
});
```

Webサーバー基礎

Expressが持つ基本機能はルーティングとミドルウェアの2つ

これにエラーハンドリングを加え、下記3つをまずは理解する

- ルーティング
- ミドルウェア
- エラーハンドリング

```
const express = require('express');
const app = express();

// ルーティングとミドルウェア
app.get('/', (req, res) => {
  res.status(200).send('hello world\n');
});

// 包括的エラーハンドリング
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('Internal Server Error');
});

app.listen(3000, () => {
  console.log('start listening');
});
```

Webサーバー基礎: ルーティング

ルーティング: パスに対してミドルウェアを呼び出す機能

ExpressではGET, POST, PUT, DELETEなどのHTTPメソッドに応じた関数にパスの文字列を渡すことで定義する

‘/:id’ のように:から始まる定義は、動的に（リクエスト時に決まる）変数として受け取ることができる

```
// GET '/' 時の挙動
app.get('/', (req, res) => {
  res.status(200).send('hello world\n');
});

// GET '/user/:id' に一致するGETの挙動
app.get('/user/:id', (req, res) => {
  // :idをreq.params.idとして受け取る
  res.status(200).send(req.params.id);
});
```

```
$ curl -X GET http://localhost:3000/user/foo
foo
```

Webサーバー基礎: ミドルウェア

Expressにおけるミドルウェアとは、クライアントのリクエスト・レスポンス発生時にリクエストオブジェクト（req）とレスポンスオブジェクト（res）をハンドリングする関数

req: アクセス時の情報を取得するオブジェクト

res: クライアント（ユーザー）に値を返す操作を行うオブジェクト

```
(req, res) => {
  res.status(200).send(req.headers['foo'])
}
```

Webサーバー基礎: ミドルウェア

ミドルウェアはルーティングの第二引数以降に定義する

ミドルウェアは連鎖的に次のミドルウェアの呼び出しが可能

複数のミドルウェアを連結することでルートを超えた共通処理をまとめる

```
app.get('/foo', middlewareA, middlewareB, middlewareC);
```

```
(req, res, next) => {
  // リクエストの情報をログに出力
  console.log(req.method, req.path);
  // next（次のミドルウェア）を呼びだす
  next();
}
```

Webサーバー基礎: ミドルウェア

ログを出力するミドルウェアを追加してみる

nextを呼ぶことでそのミドルウェアの実行が終わったことをアプリケーションに伝え、次の処理に移る

```
// ルーティングとミドルウェア
app.get(
  '/',
  // 追加したミドルウェア
  (req, res, next) => {
    console.log(req.method, req.path)
    next()
  },
  // 元のミドルウェア
  (req, res) => {
    res.status(200).send('hello world\n')
  }
)
```

Webサーバー基礎: ミドルウェアの切り出し（課題）

共通の処理をミドルウェアとして切り出し再利用可能な状態にする

‘/’ と ‘/user/:id’ それぞれのルートにlogMiddlewareを設定する

```
const logMiddleware = (req, res, next) => {
  console.log(req.method, req.path);
  next();
}
```

Webサーバー基礎: エラーハンドリング

基本的には各ルートでエラーハンドリングを行う必要がある

包括的エラーハンドリング: Expressの各ルートでおきる同期的なエラーを包括的にとらえる機能

引数を4つもつミドルウェアだけが特別な扱いをされ、包括的なエラーハンドリングを行うミドルウェアになる

next関数を引数付きで呼び出すと包括的エラーハンドリングに飛ばすことができる

```
app.get('/', (req, res) => {
  try {
    res.status(200).send('hello world\n');
  } catch (err) {
    res.status(500).send('Internal Server Error' );
  }
});
```

```
const errorMiddleware = (req, res, next) => {
  next(new Error('ミドルウェアからのエラー'));
};

// 包括的エラーハンドリング
// 引数が4つ && 最後に定義されている
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('Internal Server Error');
});

app.listen(3000, () => {
  console.log('start listening');
});
```

Webサーバー基礎: HTML描画

HTMLの描画はresオブジェクトを利用してHTMLの文字列を送信することも可能

自力で文字列としてHTMLを構築はできるが、様々なセキュリティリスクがあるため、入力出力に適切な処理をはさむ必要がある

基本的には文字列で扱わず、そういったリスクに対処しているフレームワークやライブラリを利用すべき

近年はReactやVue, Angularといったフロントエンドのフレームワークがそれらを担保していることが多い

今回はExpressでも古くから利用されているejsで説明を行う

```
$ npm install ejs
```

Webサーバー基礎: HTML描画

app.set('view engine', エンジン名)を用いて設定することで、res.render関数からテンプレートエンジンとしてejsを呼び出しHTMLを返せる

Expressは慣例的にviewsディレクトリにテンプレートファイルを配置することが多い

```
// ejsをビューエンジンに指定
app.set('view engine', 'ejs');

app.get('/', (req, res) => {
  res.render(path.resolve(__dirname,
'views/index.ejs'));
});
```

```
directory/
├── views/
│    └── index.ejs
├── server.js
├── package.json
└── package-lock.json
```

Webサーバー基礎: HTML描画

index.ejsにHTMLを記述し、トップへのアクセス時にHTMLが返却されることをブラウザで確認する

index.ejsの内容を変更しサーバーを再起動後、その内容がブラウザに反映されることを確認する

```
<!-- index.ejs -->
<!DOCTYPE html>
<html lang="ja">

<head>
  <meta charset="UTF-8">
  <meta name="viewport"
content="width=device-width,
initial-scale=1.0">
  <title>top</title>
</head>

<body>
  index.ejs
</body>

</html>
```

Webサーバー基礎: 動的HTML描画

ejsではテンプレートに変数を渡すことで動的にHTMLを生成できる

renderの第3引数に渡した変数をejsファイルで参照できる

```
app.get('/', (req, res) => {
  const users = [
    'alpha', 'bravo', 'charlie', 'delta'
  ];
  res.render(path.join(__dirname, 'views',
'index.ejs'), { users: users });
});
```

```
<!-- index.ejs -->
<!DOCTYPE html>
<html lang="ja">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>top</title>
</head>

<body>
  <ul>
    <% for (const user of users) { %>
      <li><%= user %></li>
    <% } %>
  </ul>
</body>

</html>
```

ブラウザにおける JavaScript

ブラウザにおけるJavaScript

Node.js（サーバー）側とブラウザ側で実行される JavaScriptをそれぞれ区別するため、バックエンド /フロントエンドと呼び分ける事が多い

どちらも同じJavaScriptではあるが、どちらの環境で動くのかによって気を付けるべきポイントが異なる

例: フロントエンドではパフォーマンスの低いコードの影響を受けるのは、そのコードを実行しているユーザーのみだが、バックエンドでは同時にアクセスしているユーザーすべてに影響する

近年フロントエンド開発で利用されることが多いフレームワーク（ React, Vue, Angular）はパフォーマンスなど様々な理由からバックエンドで動作するコードも多く、境界は曖昧になってきている

書いたコードがどの環境で動いているかを意識する事が近年の JavaScript開発においては特に重要

フロントエンド開発の歴史

Node.js登場以前/普及開始時のフロントエンドでは jQueryやprototype.jsなどの薄いライブラリ（フレームワーク）が主流だった

今のようにweb上でリッチな表現はなく、ボタンのクリックなどユーザーのアクションに対して多少のリアクションを行う程度のもの

JavaScriptの仕様も今ほどしっかりしておらず、ライブラリの役割はブラウザ間における挙動差を埋めるために利用されることが多かった。特に jQueryはデファクトスタンダードに近いほど普及し、document.querySelectorなど現在の標準仕様に取り込まれたものもある。

```
document.querySelector('.user-name').addEventListener('click', (e) => { ... })
$('.user-name').on('click', (e) => { ... })
```

フロントエンドとNode.js

jQueryが特によく使われていた理由のひとつとして、機能を拡張するプラグインのしくみがあった

jQueryは多くのプラグインが存在していたが、それぞれが独自にコードの配布を行っていたため、広大なインターネットの中からコードを探し出し、JavaScriptファイルをダウンロードして自分たちのサービス上で配信をしていた

Webサービスを運用していく以上、バグや脆弱性は常に発見されていく。それらは放置できずプラグインもアップデートする必要があるがアップデートがあることに気づく仕組みも弱く、これら各種ライブラリをどう効率的に管理するかという課題があった

ここでNode.jsにバンドルされていたnpmに注目が集まる

npmはもともとNode.jsのコードを再利用可能にするためにパッケージマネージャーだったが、 JavaScriptのコードがホスティングされるため、フロントエンドのパッケージ管理にも利用できた

フロントエンドとNode.js

フロントエンドのパッケージもnpmで管理されることに伴い他のパッケージに依存する新たなパッケージが増えていった

フロントエンドのコードはブラウザで動くという特性上、実行前に JavaScriptファイルがダウンロードされている必要がある

ブラウザが同時にダウンロードできるファイル数は限られ、アプリケーションを構成するファイルが増えるにつれ、アプリケーションの実行時間が通信（ダウンロード）に専有されてしまう

Node.jsは起動時にすべてのファイルをメモリ上に展開するため、ファイル数が増えてもパフォーマンス上の問題は起きにくい

フロントエンドの場合、依存するパッケージの数が増えるほどダウンロードしなければならないファイルが増える＝パフォーマンス上のデメリットとなる

フロントエンドとNode.js

フロントエンドのファイル数問題に対処するため、一定の粒度でファイルを結合（バンドル）するという手段がとられるようになる。また、ファイルの最小化（ minify）などでファイルサイズを下げダウンロードを高速化させるといった処理も行われる

そういったフロントエンドに必要な各タスクを自動化するため、 Node.js製タスクランナー（Grunt, gulp）やバンドラー（webpack）が開発された

Node.jsは元々ネットワーク処理を得意とするバックエンド用の言語として登場したが、フロントエンド開発のツールはJavaScriptを利用するエンジニアが多く参加するため、 Node.jsが自然と採用された

フロントエンドとNode.js

これらと近いタイミングで Webのリッチ化や仕様策定が進み、フロントエンド側の JavaScriptだけでアプリケーションを構築する範囲が広がった

SPA（Single Page Application）: HTMLファイルとJavaScriptで構成されるスマートフォンアプリライクな 画面遷移の実現や、差分描画などで高速なユーザー体験

複雑性の上昇により、React, Vue, Angularといった現代的なフレームワークによる開発が行われるようになる

SPAはブラウザで動作するため、ブラウザが JavaScriptを読み込み実行する初期化ステップが必要になる

そのぶん、サーバーから転送された HTMLを表示するクラシカルな方法に比べて初期表示は遅くなる

また、当時の Googleなどのクローラーはそれほどうまく SPAを処理できなかったため、SEOなどにおいてもSPAはデメリットがあった

フロントエンドとNode.js

SPAの利点を活かしつつ弱点である初期表示の遅さや SEOに対応するためには、アクセス時にサー バー側でHTMLを構築して返す必要があった

このためSSR（Server Side Rendering）と呼ばれる、フロントエンドで JavaScriptによって生成されるページを、サーバーサイドで事前にレンダリングして HTMLとして準備しておく技術が生まれる

JavaScriptで構築されたフレームワークをサーバーで解釈し実行するには、 Node.jsなどのJavaScript実行環境を利用するのが合理的で効率的だった

これらの理由から現代的なフレームワークのバックエンドとしても Node.jsは広く利用されるようになった

Webサーバー基礎: 静的ファイル配信

ブラウザ側で実行されるJavaScriptやCSSなどのファイルは、基本的に配信時点で変更が加わらない静的なファイル

Expressでは配信するディレクトリを設定すると、そのディレクトリ以下を自動的に配信する設定が可能

```
// publicディレクトリ以下のファイルを静的ファイルとして配信
app.use('/static', express.static(path.join(__dirname, 'public')));
```

Webサーバー基礎: 静的ファイル配信

ブラウザ側で動作するJavaScriptに触れるため、publicディレクトリに配信用のJavaScriptファイルを作成する

リストをクリックしたらalertにクリック要素を表示するスクリプトを追加する

クリック対象を明示するため、HTMLにクラス名を付与しスクリプトの読み込みタグを追加する

DOMContentLoadedイベントはブラウザがDOMを構築したタイミングで発行されるイベント。クリックする対象が描画された後に処理を追加するため、DOMContentLoadedイベント発行時のイベントハンドラーに処理を記述する

```
<ul>
  <% for (const user of users) { %>
    <li class="user-name"><%= user %></li>
  <% } %>
</ul>
<script src="/static/index.js" defer></script>
```

```
// public/index.js
window.addEventListener('DOMContentLoaded', (event) => {
  document.querySelectorAll('.user-name').forEach((elem) => {
    elem.addEventListener('click', (event) => {
      alert(event.target.innerHTML);
    });
  });
});
```

script async/defer

scriptタグにつくasync/deferはスクリプト読み込み /実行などのタイミングを仕様レベルで制御するキーワード

async: スクリプトが実行可能になるタイミングまで実行を遅延させる
defer: DOMの構築が完了した後、DOMContentLoadedイベントが発行される直前に実行する

ブラウザはHTMLをパースしてDOMを構築する際にscriptタグを見つけると、そこでDOMの解釈を停止しスクリプトのダウンロードと実行を行う。つまり、途中にダウンロードや実行に時間のかかるスクリプトが存在すると表示までの時間が延びる

現代におけるscriptタグはasyncを基本に、対応できないケースでは deferを利用するとよい

script async/defer

静的なHTMLと多少のスクリプトを含む程度のページの場合「 Webページの表示パフォーマンス」は「HTML表示までの速度」とほぼイコール。 しかし、近年のWebでは動的な挙動を求められることも増え、多くのページでJavaScriptが実行されasync/deferなどの仕様が必要となった

JavaScriptに関わる仕様はほとんどの場合、過去の互換性を壊さず新たな挙動を追加していく。
JavaScriptの進化はWebを壊さないことに腐心し、その互換性は非常に重要視されている

言語としての互換性の強さは Node.jsにとってもバージョンアップによって壊れにくいというメリットになるが、逆にいえば新しく効率のいい書き方でなくても「動かせてしまう」デメリットでもある

全部を知っている必要はないが、キャッチアップを続け自分たちのアプリケーションに必要なアップデートを取り込む意識は重要

ブラウザにおけるJavaScript

フロントエンドのJavaScriptによる入力の扱い方の基礎を理解する

- 入力欄
- ボタン

```
<ul class="user-list">
  <% for (const user of users) { %>
    <li class="user-name"><%= user %></li>
  <% } %>
</ul>
<input type="text" class="input-text" />
<button class="send-button">送信</button>
```

```
// public/index.js
window.addEventListener('DOMContentLoaded', (event) => {
  // ...
  document.querySelector('.send-button').addEventListener('click', (event) =>
{
    const newElement = document.createElement('li');
    const text = document.querySelector('.input-text').value;
    newElement.innerHTML = text;
    document.querySelector('.user-list').appendChild(newElement);
  });
});
```

データベース連携

Webサービス開発基礎

フロントエンドとバックエンドの連携について学ぶ

フロントエンドでの変更を受け取り、バックエンドでデータを永続化する

Webアプリケーションでは一般的にデータを永続化するためにデータベースと呼ばれるソフトウェアが利用される

ファイルシステムを利用した永続化も不可能ではないが、同時書き込み /読み込みの制御や複数サーバー間でのデータ共有を考えると、DBなどのソフトウェアの利用が理想

Webサービス開発基礎

今回はデータベースとしてJavaScriptとも相性のよいNoSQLのMongoDBを利用する
https://www.mongodb.com/

PCのローカルにインストールする方法もあるが、インストール方法はOSやバージョンによって差異が多く構築にコストがかかる

近年のWeb開発ではデータベースなどのソフトウェア（ミドルウェア）はDockerを利用した仮想環境で構築することが多い
https://www.docker.com/

Dockerの導入

Dockerをインストール
https://www.docker.com/

```
# dockerの起動確認
$ docker -v
# mongodbの起動
$ docker run --rm --name my-app-db -p 27017:27017 mongo

# 別ターミナルから
$ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS                      NAMES
7f1ce083e3b0   mongo     "docker-entrypoint.s…"   42 seconds ago   Up 42 seconds   0.0.0.0:27017->27017/tcp   my-app-db


$ docker exec -it my-app-db sh
# mongosh
test> show dbs
admin   40.00 KiB
config  12.00 KiB
local   40.00 KiB
```

Node.jsとDocker

Dockerはコードやコードが依存する実行環境などをまとめてコンテナと呼ばれるものに閉じ込めて仮想化を行えるソフトウェア

開発時に依存するソフトウェアを簡易に構築することに利用されたり、コンテナ実行環境を提供するクラウドサービスへの本番環境構築などにも利用される

コンテナイメージに依存関係を閉じ込めることで、異なるデバイスや OS間での環境の再構築や移行などをより容易にできるメリットがある

Node.jsに関しては互換性が高く、libuvがOS間の差異を吸収するため、開発時に限れば仮想化によるパフォーマンス低下なども考慮すると Dockerにのせる強い動機はあまりない

デプロイ先としてはコンテナは有力な候補になるが Node.js特有の注意点もある

Node.jsとDocker

Dockerが内部で起動するプロセスを pid=1として起動する

pid=1のプロセスはLinux上で特別なプロセスとして扱われ、pid=1として動作するプロセスは送信されたシグナルを自身で適切にハンドリングしなければならない

しかし、Node.jsはデフォルトでは pid=1として動作するように設計されていない

そのため、コンテナ化する場合には自身でシグナルをハンドリングするコードを記述するか、ハンドリングを行うソフトウェアの利用が必要になる

Dockerにはそういったソフトウェアに対応する Tiniがバンドルされ --init オプションで起動ができる

```
$ docker run --rm -p 3000:3000 --init node-simple-server
```

Node.jsからDBへ接続

clientを作成してMongoDBに接続する

サーバーの起動前にDBなどの接続を完了させる

サーバーが起動するとユーザーのアクセスを受け付けてしまうため、DBの接続完了前に起動するとデータが取得可能になる前にアクセスがくる可能性がある

```
$ npm install mongodb
```

```
const { MongoClient } = require('mongodb');
const client = new MongoClient('mongodb://localhost:27017');

async function main() {
  // サーバーのlisten前に接続する
  await client.connect();

  const db = client.db('my-app');

  // ...
  app.get('/', (req, res) => {
    const users = ['alpha', 'bravo', 'charlie', 'delta'];
    res.render(path.join(__dirname, 'views', 'index.ejs'), { users
});
  });

  app.listen(3000, () => {
    console.log('start listening');
  });
}
main()
```

データ取得

userの配列をMongoDBから取得するように変更

DBへデータを挿入し、ブラウザ上でデータが取得できることを確認

```
app.get('/', async (req, res) => {
  try {
    const users = await db.collection('user').find().toArray();
    const names = users.map((user) => { return user.name });
    res.render(
      path.join(__dirname, 'views', 'index.ejs'),
      { users: names }
    );
  } catch (e) {
    console.error(e);
    res.status(500).send('Internal Server Error');
  }
});
```

```
$ docker exec -it my-ap-db mongosh
# my-appというデータベースに接続
test> use my-app
switched to db my-app
# 初期状態ではデータはから
my-app> db.user.find()
[]
my-app> db.user.insertOne({ name: 'alpha' })
{
  acknowledged: true,
  insertedId: ObjectId('66aa5b51c79ec05eb6149f48')
}
# dbにデータが挿入されたことを確認
my-app> db.user.find()
[ { _id: ObjectId('66aa5b51c79ec05eb6149f48'), name: 'alpha' }
]
```

APIの作成

ブラウザからDBにデータを保存できるように WebAPIを作成

POST /api/user にデータを送信して受け取った値を DBに保存する

ブラウザからWebサーバー等にデータを送信するには fetch関数を用いる

DBにデータが挿入されることを確認

```
app.post('/api/user', express.json(), async (req, res) =>
{
  const name = req.body.name;
  if (!name) {
    res.status(400).send('Bad Request');
    return;
  }
  await db.collection('user').insertOne({ name: name });
  res.status(200).send('Created');
});
```

```
// ブラウザのコンソールで確認
fetch('/api/user', { method: 'POST', headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({ name: 'test' }) })
```

APIの作成

本来は送信されたreq.bodyは必ず自分が想定する値（文字列である、最小 /最大数以内である等）であることをチェックしなければならない（バリデーション）

今回行ったようにユーザーはブラウザから自由にリクエストを送信できてしまう

攻撃者は必ずリクエストをカスタマイズして送信してくる

セキュリティのためにもユーザーが送信してきたリクエストを信用してはならない

```
const name = req.body.name;
if (!name) {
  res.status(400).send('Bad Request');
  return;
}
if (typeof name !== 'string') {
  res.status(400).send('Not String');
  return;
}
```

APIのコール

ブラウザから直接送ったリクエストをボタンを押した際に実行するよう修正

データ更新後の反映が自動的に行われるようにする

```
// ブラウザのコンソールで確認
fetch('/api/user', { method: 'POST', headers: { 'Content-Type': 'application/json' }, body: JSON.stringify({ name: 'test' }) })
```

```
// public/index.js
window.addEventListener('DOMContentLoaded', (event) => {
  // ...
  document.querySelector('.send-button').addEventListener('click', (event) => {
    const text = document.querySelector('.input-text').value;
    // ...
  });
});
```

テスト

ここまでで、Webアプリケーションの開発について基礎的な内容を扱った

Webアプリケーションは作って終わりではなく、リリースしてからが本番なため運用や保守について意識する必要がある

特に不特定多数のユーザーがアクセスする Webアプリケーションでは、ユーザーの利用ケースが特定しづらく、テストすべきパターンが膨大になるため自動テストが必須といえる

テストすべきパターンやテストを行いやすい設計とはどういうものかを理解することが重要

テスト

次のコードにテストを書くパターンを考えるために、まず機能（要件）を分解する

- ユーザーのPOSTリクエストを取得
- リクエスト形式のチェック
- DBへデータの保存
- ユーザーへレスポンスを返却

このうち、フレームワーク（ライブラリ）とアプリケーションそれぞれが担保する機能を分類する

下記はExpressが担保している機能

- リクエスト取得
- レスポンス返却

```
app.post('/api/user', express.json(), async (req, res) => {
  const name = req.body.name;
  if (!name) {
    res.status(400).send('Bad Request');
    return;
  }
  if (typeof name !== 'string') {
    res.status(400).send('Bad Request');
    return;
  }
  await db.collection('user').insertOne({ name: name });
  res.status(200).send('Created');
});
```

テスト

フレームワーク（ライブラリ）がもつ機能そのものはフレームワークの側でテストされている

それらの機能そのものをテストするのは過剰なテスト

そのままのコードではフレームワークの機能とアプリケーションの機能を分けたテストが難しい

それぞれを分離し、個別にテスト可能な状態にコードを修正する

例）Expressが担保している機能はどこか

- app.post(‘/api/user’, …)
- req.body.nameに値が入るか
- res.status(...).send(...) でレスポンスを返す

```
app.post('/api/user', express.json(), async (req, res) => {
  const name = req.body.name;
  if (!name) {
    res.status(400).send('Bad Request');
    return;
  }
  if (typeof name !== 'string') {
    res.status(400).send('Bad Request');
    return;
  }
  await db.collection('user').insertOne({ name: name });
  res.status(200).send('Created');
});
```

テスト

次のように分割したケースを考える

insertUserは引数を受け取り、成功 /失敗の状態を返却する関数として分割
insertUserはExpressに対する依存関係がなくなる
= Expressの機能に対してテストを書く必要がない

アプリケーション担保すべき要件は、insertUserに対するパターンと、Expressが適切にinsertUserを呼び出せているかに分割できる

このようにテストすべき箇所を分割していくことがテストを記述する第一歩

```
async function insertUser(name) {
  if (!name) {
    return { status: 400, body: 'Bad Request' };
  }
  if (typeof name !== 'string') {
    return { status: 400, body: 'Bad Request' };
  }

  await db.collection('user').insertOne({ name: name });
  return { status: 200, body: 'Created' };
}

app.post('/api/user', express.json(), async (req, res) => {
  const name = req.body.name;
  const { status, body } = await insertUser(name);
  res.status(status).send(body);
});
```

テスト

さらにテストを記述していくため、機能単位でファイルを分割する

insertUserはdb変数に暗黙的に依存していたため、依存を外に出し引数からうけとる形に修正
insertUserは暗黙的な依存のない関数 = テストがしやすい状態

```
// user.js
async function insertUser(name, db) {
  if (!name) {
    return { status: 400, body: 'Bad Request' };
  }

  if (typeof name !== 'string') {
    return { status: 400, body: 'Bad Request' };
  }

  await db.collection('user').insertOne({ name: name
});

  return { status: 200, body: 'Created' };
}
exports.insertUser = insertUser;
```

```
const { test } = require('node:test');
const assert = require('node:assert');

const { insertUser } = require('./user');

test('insertUser', async (t) => {
  const insertOne = t.mock.fn();
  const db = {
    collection: () => {
      return { insertOne };
    },
  };
  const { status } = await insertUser('test', db)
  assert.strictEqual(status, 200, '正しくデータが挿入された場合、ステータスコード 200
を返す');
  assert.strictEqual(insertOne.mock.callCount(), 1, '1度だけinsertOneが呼ばれる
');
});
```

テスト: 課題

app.get('/', ...) のテストを記述する

- getUsersの内容
- mock, assert

```
async function getUsers(db) {
  // ...
}
exports.getUsers = getUsers;
```

```
app.get('/', async (req, res) => {
  try {
    const users = await getUsers()
    res.render(path.join(__dirname, 'views', 'index.ejs'), {
users: users });
  } catch (e) {
    console.error(e);
    res.status(500).send('Internal Server Error');
  }
});
```

```
const { getUsers } = require('./user');

test('getUsers', async (t) => {
  // mockを作成する
  const db = { ... }
  const users = await getUsers(db)
  // assert
});
```

失敗時のテスト

実際のアプリケーション開発では、失敗時のテストこそ重要

正常に動作しなかったケースでもユーザーへの影響をできる限り少なくする

次のケースはなんらかの問題で DBからデータが取得できなかったケースでエラーが起きるかをテストするコード

一見テストできていそうだが問題がある

```
test('getUsers: error', async (t) => {
  const toArray = t.mock.fn(() => {
    throw new Error('something error');
  });
  const db = {
    collection: () => {
      return {
        find: () => {
          return { toArray };
        }
      };
    },
  };

  try {
    await getUsers(db)
  } catch (e) {
    assert.strictEqual(e.message, 'something error');
  }
});
```

失敗時のテスト

エラーが起きた際にエラーを throwせず空の配列を返す仕様に変更が発生した想定

仕様が変更されたので本来テストは失敗してほしいがテストが通ってしまう＝エラーが起きたことをテストできていない

エラーがthrowされなくなったため、assertがひとつも呼ばれていないためテストが通過する

こういったケースでは assertが1回以上呼ばれることを担保しておく必要がある

```
async function getUsers(db) {
  try {
    const users = await db.collection('user').find().toArray();
    const names = users.map((user) => { return user.name });
    return names
  } catch (e) {
    return []
  }
}
```

```
test('getUsers: error', async (t) => {
  const toArray = t.mock.fn(() => {
    throw new Error('something error');
  });
  const db = { ... }

  try {
    await getUsers(db)
  } catch (e) {
    assert.strictEqual(e.message, 'something error');
  }
});
```

失敗時のテスト

大体のテストランナーで assertの回数を数える機能が提供されている

try-catchを使うケースだけではなく、テスト内に if文やループを利用しているケースでも同様の問題が発生する（修正によってその分岐を通らなくなった、配列が 0件になってしまったなど）

テストを記述する際は記述したアサーションが必ず呼ばれるか、呼ばれなかった場合適切にテストを落とせるかを意識して記述する

```
test('getUsers: error', async (t) => {
  // ...

  // assert の数をカウントする
  t.plan(1);
  try {
    await getUsers(db);
    // 絶対に失敗する assert を書く方法もある
    assert(true, false);
  } catch (e) {
    // テスト内で assert の回数がカウントできるようにする
    t.assert.strictEqual(e.message, 'something error');
  }
});
```

```
test('xxx', async (t) => {
  // ループが0件の場合おかしいので回数を数える
  t.plan(arr.length)

  for (const elem of arr) {
    t.assert.strictEqual(elem, 1);
  }
});
```

自由課題

自由課題

ここまでの知識を利用し、Node.js/JavaScriptを利用したシステムやツールの作成を行う

- Node.jsを利用する

- GitHubにリポジトリを作成しコードを管理する

- README.mdをコミットし、作成したソフトウェアの説明を記述する