

Web講習会2021 ワールドワイドウェブ発展

第2回: Node.js



Arthur

この回の目標

- ES2015の便利な構文を知る
- Node.jsというJavaScript実行環境でのプログラミングを試みる
- Yarn (npm)の目的と使い方を知る

今日の内容を1日で理解するのは厳しいと思います。
まずは手を動かして自分で同じようなコードを
動作させるところから始めてみましょう。

Node.js



以降の前提

以下のページを読んで、Node.js 16.4.2とYarnが実行できる環境を用意

自分なりのやり方がある人はその方法でもOK

<https://web-tutorial.tap.buratsuki.page/nodeenv/>

Web講習会2021 Home WWW基礎 WWW応用 Webセキュリティ

Node.js環境構築

以降の講習会は、以下のツールを動かす環境があることが前提になります。好きな方法でこれらを導入してください。

- Node.js (JavaScriptのサーバサイド実行環境) v16.4.2
- Yarn (パッケージ管理ツール)

本ページでは、複数のプロジェクトで異なるバージョンのNode.jsを利用したいときに便利なnodenvの導入について説明します。Dockerなどの仮想環境を導入する必要がなく、カレントディレクトリによって自動的にバージョンを切り替えられます。

anyenvの導入

anyenvは、数ある***envというツールをかんたんに導入できるものです。

Homebrewを導入している人は、以下のコマンドを実行するだけです。

```
$ brew install anyenv
$ anyenv install --init
$ echo "eval \"\${anyenv init -}\"" >> ~/.zshrc # bashの人は~/.bashrc
$ exec $SHELL -i
$ anyenv --version
anyenv 1.1.3
```

Homebrewを使っていない人は、GitHubからコードをダウンロードして導入します。

JavaScript

JavaScript

ブラウザ上で動く唯一のスクリプト言語
≠ Java

近年はC, Rust, Goなどで生成した
バイナリコードも実行できる
→WebAssembly

言語の特徴

- 手続き型・オブジェクト指向・関数型のパラダイムに対応
- オブジェクト指向はプロトタイプベース
- 非同期処理 (≠ 並列処理)
読み込み中などの処理待ちのときに、他の処理を行えるように
- 動的型付け

Node.jsの登場

Node.js

2009年に登場したJavaScript実行環境

大量な同時接続を処理できるWebサーバの開発が目的

JavaScriptをサーバサイド言語として導入するメリット

- シングルスレッドでの非同期処理が容易に実装できる
- サーバサイドとクライアントサイドを同じ言語で書ける



ブラウザの実行環境とNode.jsの違い

ブラウザの実行環境

OSが提供する機能に自由にアクセスできない

Webブラウザとして必要な機能にだけ、APIを通してアクセス可能

Webサイトにアクセスするだけで勝手にファイルを読み書きされたら…

Node.js

OSが提供する機能に自由にアクセスできる

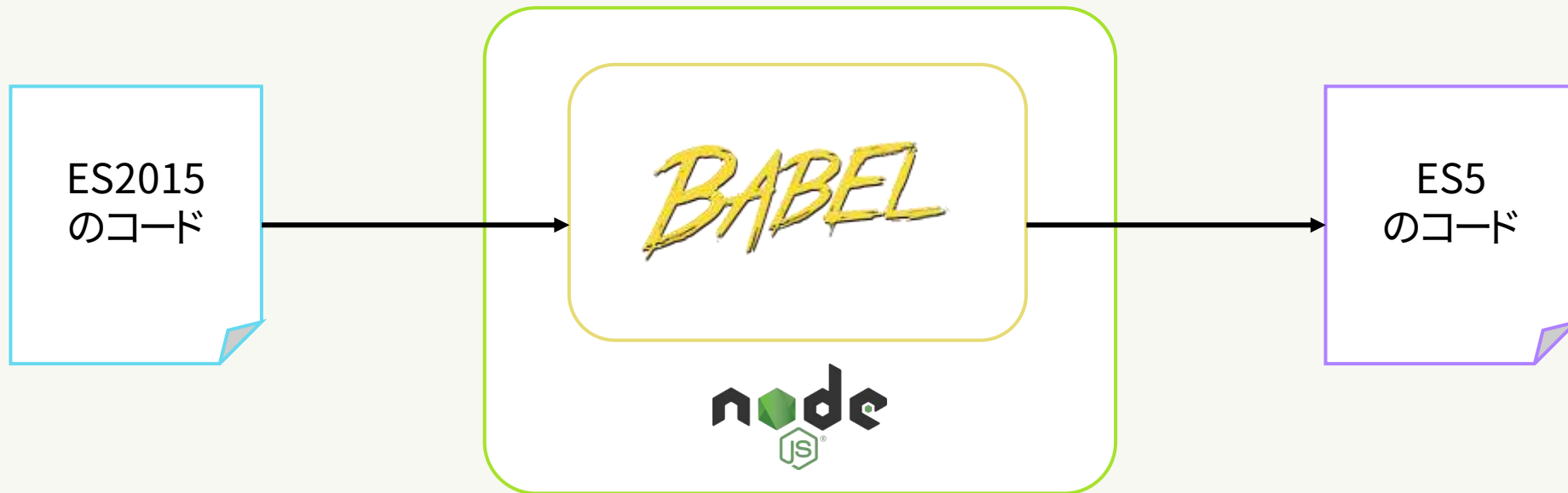
Pythonなどの言語と同等に

- 標準入出力
- ファイルの入出力
- 環境変数の読み取り
- プロセスのforkや終了
- etc

Node.jsの発展

Node.jsの前述の特徴を活かし、クライアントサイドの開発にも
Node.jsを活用するようになる

例) Babel (トランスパイラ)



Node.jsのプログラムの実行

以下のようなコードを用意し、`node` [ソースコードのパス]を実行

```
console.log('Hello, Node.js!')
```

helloworld.js

`console.log`は標準出力に出力する

```
~/Documents/titechapp/pr4 >>> node helloworld.js
Hello, Node.js!
~/Documents/titechapp/pr4 >>> 
```

ES2015以降の便利な構文



const, let

変数の宣言方法に2種類のキーワードが追加

let … 再代入可能

Scalaにおけるvar

const … 再代入不可

Javaでfinalのついた変数、Scalaにおけるval

```
let a = 1
a = 2 // OK
const b = 3
b = 4 // Error
```

【補足】

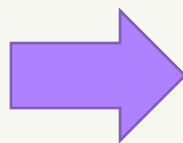
varと異なり、変数の巻き上げ(hoisting)を行わない

ブロックスコープ

letやconstで宣言した変数は、**ブロック{}内のみのスコープ**をもつ

よくあるミス

```
const num = 3
if (num % 3 === 0) {
  const answer = '3の倍数'
} else {
  const answer = '3の倍数ではない'
}
console.log(answer)
```



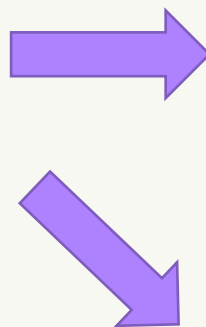
```
const num = 3
let answer
if (num % 3 === 0) {
  answer = '3の倍数'
} else {
  answer = '3の倍数ではない'
}
console.log(answer)
```

アロー関数

アロー関数

PythonのlambdaやJavaのアロー演算子のように、関数を完結に定義できる

```
var multiple = function(num) {  
  return num * 2;  
};  
  
console.log(multiple(3)); // 6
```



```
const multiple = (num) => {  
  return num * 2  
}  
  
console.log(multiple(3)) // 6
```

```
const multiple = (num) => num * 2  
  
console.log(multiple(3)) // 6
```

【補足】

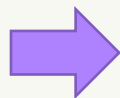
thisがコードに出てくる場合、
通常関数とアロー関数は仕様が異なる

アロー関数と高階関数

```
var words = [  
  'Now', 'I', 'need', 'a', 'drink',  
  'alcoholic', 'of', 'course',  
  'after', 'the', 'heavy', 'lectures',  
  'involving', 'quantum', 'mechanics',  
];
```

```
var count = words.map(function(word) {  
  return words.length;  
}).reduce(function(acc, cur) {  
  return acc + cur;  
});
```

```
console.log(count); // 77
```



```
const words = [  
  'Now', 'I', 'need', 'a', 'drink',  
  'alcoholic', 'of', 'course',  
  'after', 'the', 'heavy', 'lectures',  
  'involving', 'quantum', 'mechanics',  
]
```

```
const count = words.map((word) => words.length)  
  .reduce((acc, cur) => acc + cur)
```

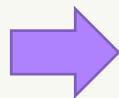
```
console.log(count) // 77
```

テンプレートリテラル

テンプレートリテラル

文字列をいちいち結合せず、C言語のsprintfやPythonのformatのように式の値を埋め込むことができる

```
var age = 18;  
var message = "I'm " + age + ' years old.';  
  
console.log(message); // I'm 18 years old.
```



```
const age = 18  
const message = `I'm ${age} years old.`  
  
console.log(message) // I'm 18 years old.
```

スプレッド構文

スプレッド構文

配列やオブジェクトの要素を展開する
オブジェクトのコピーに利用可能

```
const primes = [2, 3, 5, 7]
const exPrimes = [...primes, 11, 13, 17]

console.log(exPrimes) // 2, ..., 7, 11, 13, 17
```

```
const student = {
  id: '15B12345',
  name: 'Arthur',
  department: '情報工学系',
}

const newStudent = {
  ...student,
  id: '19M12345',
  course: '知能情報コース',
}

console.log(newStudent)
// id=19M12345, name=Arthur,
// department=情報工学系, course=知能情報コース
```


分割代入

分割代入 (destructuring assignment)

配列やオブジェクトを個々の変数に分割できる

関数が2つ以上の値を返したいときなどに利用(Pythonのタプルのように)

名前付き引数も擬似的に実現可能

```
const student = {  
  id: '15B12345',  
  name: 'Arthur',  
}  
  
const { id, name } = student  
  
console.log(id) // 15B12345  
console.log(name) // Arthur
```

```
const searchStudent = ({ id, name }) => {  
  console.log(id) // 15B12345  
  console.log(name) // Arthur  
}  
  
searchStudent({  
  id: '15B12345',  
  name: 'Arthur',  
})
```

Promiseとasync/await



コールバック地獄

非同期処理で順番を保証するにはcallbackを入れ子にするしかない
結果を受け取る際も、callback関数内でしか扱えない

```
setTimeout(() => {  
  console.log('A')  
  setTimeout(() => {  
    console.log('B')  
    setTimeout(() => {  
      console.log('C')  
    }, 1000)  
  }, 1000)  
}, 1000)
```

実際の処理例

- WebAPI1にアクセス
- リクエスト結果を使ってWebAPI2にアクセス
- リクエスト結果を使ってWebAPI3にアクセス

Promise

Promiseを使うと非同期処理を良い感じに記述できる

```
const wait1s = () => new Promise((resolve) => {
  setTimeout(() => {
    resolve()
  }, 1000)
})
wait1s().then(() => {
  console.log('A')
  return wait1s()
}).then(() => {
  console.log('B')
  return wait1s()
}).then(() => {
  console.log('C')
})
```

Promiseは3つの状態を持つ

- 待機
- 完了
- 失敗

Promiseオブジェクトはthenメソッドを持つ

resolve()が呼ばれたら待機→完了
完了したらthen()の引数を実行する

async, await

ES2017からはasync, await構文を利用して、同期処理のように書ける

```
const wait1s = () => new Promise((resolve) => {
  setTimeout(() => {
    resolve()
  }, 1000)
})

const main = async () => {
  await wait1s()
  console.log('A')
  await wait1s()
  console.log('B')
  await wait1s()
  console.log('C')
}

main()
```

await

Promiseが完了するまで待つ
互換性の問題から、async修飾子をつけた関数内でのみ使えない

async

関数の返却値をPromiseでラップ

```
const func = async () => 2
```

```
const func = () => new Promise(() => {
  resolve(2)
})
```

2種類のmodule構文

Node.jsとCommonJS

Node.jsはECMAScriptではなく、CommonJSという仕様に基づいて実装されている

ECMAScript … ブラウザ実行環境での標準仕様

CommonJS … ブラウザ外実行環境での標準仕様

主にmodule構文に関する文脈でこれが問題になる

- CommonJS形式
- ES Modules形式

moduleとは

module

プログラムから他のソースコードのファイルを読み込んで利用する仕組み

Node.jsには、様々な機能がコアモジュールとして実装されている

- fs (ファイルの読み書き)
- http (HTTP通信)
- querystring (URLのクエリ文字列を生成・parse)
- etc

CommonJSのmodule

利用する側: `require()` で呼び出す

利用される側: `module.exports` に呼び出されるものを代入

```
const greeting = require('./commonjs_greeting.js')
greeting.hello() // Hello.
```

commonjs_module_test.js

```
~/Documents/titechapp/pr4 >>> node commonjs_module_test.js
Hello.
~/Documents/titechapp/pr4 >>> █
```

```
module.exports = {
  hello() {
    console.log('Hello.')
  },
  goodMorning() {
    console.log('Good morning.')
  },
}
```

commonjs_greeting.js

ES Modules

利用する側: `import ~~ from '~~'` で呼び出す

利用される側: `export default` 文に呼び出されるものを記述

```
import greeting from './esmodules_greeting.mjs'  
greeting.hello() // Hello.
```

esmodules_module_test.mjs

```
~/Documents/titechapp/pr4 >>> node esmodules_module_test.mjs  
Hello.  
~/Documents/titechapp/pr4 >>> █
```

互換性の問題から、拡張子を.mjsにする必要がある

```
export default {  
  hello() {  
    console.log('Hello.')  
  },  
  goodMorning() {  
    console.log('Good morning.')  
  },  
}
```

esmodules_greeting.mjs

2方式の使い分け

Node.jsでの記述スタイルとしては、CommonJS方式が主流

バージョン14になるまでES Modulesは実験的な機能だった

本日の講習会は以降CommonJSを採用します

しかし、次回以降は、JavaScriptのコードをトランスパイルして利用

この場合はES Modules形式で記述するのが一般的

ひとまず次回までES Modulesのことは忘れてください

Node.jsのビルトインモジュールの利用

Node.jsに実装されているモジュールの読み込み
requireにパスではなくモジュール名を書く
import ~ from ~の場合も同様

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

const server = http.createServer((_req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/html' })
  res.write(`<!DOCTYPE HTML>
<html>
  <head><title>Simple Server</title></head>
  <body><p>Hello World!</p></body>
</html>`)
})

server.listen(8080)
```

← httpモジュールを使って、
シンプルなHTTPサーバを起動する例

Node.jsのパッケージ管理

パッケージ管理とは

パッケージ管理ツール

サードパーティが開発したモジュール群(**パッケージ**)を自分のプログラムで利用できるようにする仕組み

他の言語の例 再帰的頭字語

- Pip: Install Package (Python)
- RubyGems (Ruby)
- Go Modules (Go)

ソースコードを手動でDLして埋め込むのと比較した**メリット**

- 使用しているパッケージにアップデートがあっても簡単に適用可能
- パッケージがさらに他のパッケージに依存してもよい(解決してくれる)
- 多くのパッケージを利用しても、本体のソースコードのサイズは増えない

npmとYarn

Node.jsのパッケージ管理ツールはnpm

Node Package Manager

Node.jsと一緒に導入される

(他の言語のものも含め)世界最大のパッケージレジストリを持つ

ですが、今回はYarnというパッケージ管理ツールを利用

npmの改良版(高速なインストール、セキュリティetc)

今回は事前資料通りに準備すれば導入されているはず

Yarnの準備

`yarn init`で設定ファイルの雛形を設定
生成される`package.json`が、npm/yarnの設定ファイル

```
~/Documents/titechapp/pr4 >>> yarn init
yarn init v1.22.10
question name (pr4):
question version (1.0.0):
question description:
question entry point (index.js):
question repository url:
question author:
question license (MIT):
question private:
success Saved package.json
✨ Done in 18.49s.
```


パッケージの追加

`yarn add [パッケージ名]`でパッケージを追加(&インストール)
package.jsonに依存関係が追記される
yarn.lockにインストールされたパッケージのバージョンを記録

```
~/Documents/titechapp/pr4 >>> yarn add axios
yarn add v1.22.10
info No lockfile found.
[1/4] 🔍 Resolving packages...
[2/4] 📦 Fetching packages...
[3/4] 🔗 Linking dependencies...
[4/4] 🔨 Building fresh packages...
success Saved lockfile.
success Saved 2 new dependencies.
info Direct dependencies
└─ axios@0.21.1
info All dependencies
└─ axios@0.21.1
└─ follow-redirects@1.13.3
✨ Done in 0.86s.
```

axios:
Promiseを利用している
HTTPクライアントモジュール

インストールしたパッケージの利用

npmやYarnでインストールしたパッケージは、ビルトインモジュールと同様に利用可能

```
const axios = require('axios')
const url = 'http://s3-ap-northeast-1.amazonaws.com/titechapp-web-data/news.json'
const main = async () => {
  const response = await axios.get(url)
  console.log(response.data)
}
main()
```

```
~/Documents/titechapp/pr4 >>> node axios_test.js
[
  {
    id: 1,
    type: 2,
    textJa: 'TitechApp 2(iOS)をリリースしました！',
    textEn: 'TitechApp 2 for iOS has been released!',
    linkURL: '/products/titechapp/',
    date: '2018.07.12'
  },
  {
    id: 2,
    type: 1,
    textJa: 'Titech App ProjectのWebサイトを公開しました！',
    textEn: 'The website of Titech App Project has been opened!',
    linkURL: '',
    date: '2018.07.13'
  },
  {
    id: 3,
    type: 2,
    textJa: 'TitechApp 2(Android)をリリースしました！',
    textEn: 'TitechApp 2 for Android has been released!',
    linkURL: '/products/titechapp/',
    date: '2019.01.05'
  }
]
```

次回予告

TypeScriptによる堅牢な開発

- TypeScriptの意義と文法
- 基本の型
- 網羅チェック
- 型定義ファイル