頑なに再代入しない！

abetomo

ClearCode Inc.

はじめに

あくまで「頑なに再代入しない！」についてのお話

内容

- ✓ 頑なに再代入しない！ワケ
- ✓ 再代入しない！コード例
- ✓ 再代入しない！デメリット検証
- ✓ 再代入しない！実践してみて

再代入のコード例

```
let price = 100
if (キャンペーンA期間中) {
  price = price * 0.8
}

// ...
if (キャンペーンF期間中) {
  // キャンペーンF期間中のみ税込価格から2割引
  price = (price * 1.1) * 0.8
} else {
  // ...
}

// ...
// `price` はいくつ？
```

頑なに再代入しない！のワケ

- ✓ 読みやすさ・メンテナンス性の向上
- ✓ バグの発生リスクを低減
- ✓ 保守性・変更への強さ

メンテナンス性の向上

- ✓ 変数の値が途中で変わることがないため、「この時点で何の値か」を一目で把握できる
- ✓ 値の追跡が容易になり、コードの全体像や処理の流れが理解しやすい

バグの発生リスクを低減

- ✓ 意図しない再代入によるバグや予期せぬ挙動が発生しにくく、信頼性が高まる
- ✓ 複数箇所で値を変更する必要がなくなるため、バグを防ぎやすい

保守性・変更への強さ

- ✓ 変数の状態が関数内で常に不変なら、変更やリファクタ時の影響調査が容易になる
- ✓ 意図しない影響範囲が狭くなり、スコープも明確になる

内容

- ~~頑なに再代入しない！ワケ~~
- **再代入しない！コード例**
- 再代入しない！デメリット検証
- 再代入しない！実践してみて

再代入しない！コード例

- ✓ フラグの設定
- ✓ 参考: 配列から抽出
- ✓ 参考: 配列の加工

例: 再代入あり: フラグ設定

```
let flag = false
if (t % 2 === 0) {
  flag = true
} else if (t % 7 === 0) {
  flag = true
}


// ...
```

例: 再代入なし: フラグ設定

```
const flag = (() => {
  if (t % 2 === 0) {
    return true
  }
  if (t % 7 === 0) {
    return true
  }
  return false
})()

// ...
```

参考: 素朴に配列から抽出

（正確には再代入してないけど…）

```
const data = Array.from({ length: 1000 }, (_, i) => i)

const length = data.length
const result = []
for (let i = 0; i < length; i++) {
  if (data[i] % 2 === 0) {
    result.push(data[i])
  }
}
```

補足: Array.prototype.push()

- ✓ constで宣言したresultにpushしてたよ？
- ✓ constは変数への再代入ができなくなるだけ
- ✓ ArrayやObjectの要素は変更できる

補足: pushでエラー

次のコードは再代入なのでエラー

```
const data = []
data = [1]

// data = [1]
//      ^
//
// TypeError: Assignment to constant variable.
```

参考: 配列から抽出（filter）

```
const data = Array.from({ length: 1000 }, (_, i) => i)

const result = data.filter((v) => v % 2 === 0)
```

参考: 素朴に配列の加工

（正確には再代入してないけど…）

```
const data = Array.from({ length: 1000 }, (_, i) => i)

const length = data.length
const result = []
for (let i = 0; i < length; i++) {
  result.push(data[i] * 2)
}
```

参考: 配列の加工（map）

```
const data = Array.from({ length: 1000 }, (_, i) => i)

const result = data.map((v) => v * 2)
```

ここまでまとめ

- ✓ 再代入がないほうがわかりやすい！
- ✓ 途中で値を追加したりしないほうがわかりやすい！

内容

- ~~頑なに再代入しない！ワケ~~
- ~~再代入しない！コード例~~
- **再代入しない！デメリット検証**
- 再代入しない！実践してみて

再代入しないデメリット

やはり気になるのは実行速度。

遅いんだろうな、と思いつつ、どのくらい遅いのか検証したことがなかったので、この機会にざっくり検証してみました。

検証方法

- ✓ Node.js 24で検証
  - ✓ Dockerイメージ `node:24` を活用
- ✓ `console.time()` で測定
  - ✓ 時間が短いほうが速い
- ✓ 5回実行して中央値

速度検証: フラグ

次の3パターンで測定する。

1. IIFE（即時実行関数式）
2. 関数をつくる
3. 再代入でフラグ設定

IIFE検証

```
const n = 100_000

console.time('no reassignment IIFE')
for (let t = 0; t < n; t++) {
  const flag = (() => {
    if (t % 2 === 0) {
      return true
    }
    if (t % 7 === 0) {
      return true
    }
    return false
  })()
}
console.timeEnd('no reassignment IIFE')
```

関数実行で検証

```
console.time('no reassignment func call')
const checkValue = (num) => {
  if (num % 2 === 0) {
    return true
  }
  if (num % 7 === 0) {
    return true
  }
  return false
}
for (let t = 0; t < n; t++) {
  const flag = checkValue(t)
}
console.timeEnd('no reassignment func call')
```

再代入検証

```
console.time('reassignment')
for (let t = 0; t < n; t++) {
  let flag = false
  if (t % 2 === 0) {
    flag = true
  } else if (t % 7 === 0) {
    flag = true
  }
}
console.timeEnd('reassignment')
```

速度検証: フラグの結果

	時間
IIFE	11.51ms
関数実行	1.99ms
再代入	1.929ms

参考: 速度検証: filter

	時間
素朴に抽出	19.696ms
filter	34.01ms

参考: 速度検証: map

	時間
素朴に加工	32.12ms
map	21.378ms

ここまでまとめ

再代入しない、は遅い…？

許容範囲？

他に再代入しないデメリット

オプジェクトのコピーなどの操作が増えるので、メモリの使用量など気になるところ。

~~いい感じの検証ができなかったので~~ 発表時間も限られるので今回の発表では割愛。

内容

- ~~頑なに再代入しない！ワケ~~
- ~~再代入しない！コード例~~
- ~~再代入しない！デメリット検証~~
- **再代入しない！実践してみて**

実践したソフトウェア

node-lambda

AWS Lambdaにコードをdeployするソフトウェア。
Lambdaが始まった頃から開発されていて、私自身も使っていた。
（~~その後はSAMに乗り換え。~~）

再代入しない実践してみて +

- ✓ **やっぱり値が不変なので安心**
  - ✓ **コードレビューもしやすい**
- ✓ 関数をつくることに意識が向きやすい
  - ✓ ユニットテストがしやすくなるメリットも

バグが少ない？

- ✓ 私の全PRは296件
  - ✓ 2017年の春頃からなので8年分
- ✓ PRにbugが含まれるのは7件
  - ✓ 7 / 296 = 2%
- ✓ タイトルに`fix`が含まれるPRは54件
  - ✓ 54 / 296 = 18%

再代入しない実践してみて -

- ✓ 関数即時実行が慣れないと読みにくい、かも
- ✓ ツールの特性上、実行速度にシビアになる必要がなかった
  - ✓ 実行速度が重要な場面では性能検証はしっかりした方が良い
    - ✓ ユニットテストでチェックとか

最後に実際のコード紹介

node-lambdaのコードを紹介します。

実際の例: ファイル名の取得

```
const filename = (() => {
  for (const extension of ['.js', '.mjs']) {
    if (fs.existsSync(splitHandler[0] + extension)) {
      return splitHandler[0] + extension
    }
  }
})()
```

実際の例: map

```
const paramsList = scheduleList.map((schedule) =>
  Object.assign(schedule, { FunctionArn: functionArn }))
```

実際の例: isXXX

```
_isUseS3 (program) {
  if (typeof program.deployUseS3 === 'boolean') {
    return program.deployUseS3
  }
  return program.deployUseS3 === 'true'
}

_useECR (program) {
  return program.imageUri != null && program.imageUri.length > 0
}
```

まとめ

私の頑なに再代入をしない思いのたけを書きました。

保守しやすくて良い点もありますし、書き方によっては遅くなったりしますし、適切な場面で適切に活用すると良いと思います！

おまけ: 再代入しない？

- ✓ `let` はない？
- ✓ 再代入？

let はない？

```
$ grep -r 'let ' node-lambda/{bin,lib} | wc -l
       5
```

let がある…。

再代入？

```
_lambdaFunctionConfiguration (params) {
  const lambdaFunctionConfiguration = {
    Events: params.Events,
    LambdaFunctionArn: params.FunctionArn
  }
  if (params.Filter != null) {
    // 再代入？
    lambdaFunctionConfiguration.Filter = params.Filter
  }

  return lambdaFunctionConfiguration
}
```