コワクナイヨ TypeScript

- static typing re:loaded -

dojineko / GMO Pepabo, Inc.

2019.05.15 FukuokaJS

TypeScript
やってますか？

最近のTypeScriptダイジェスト

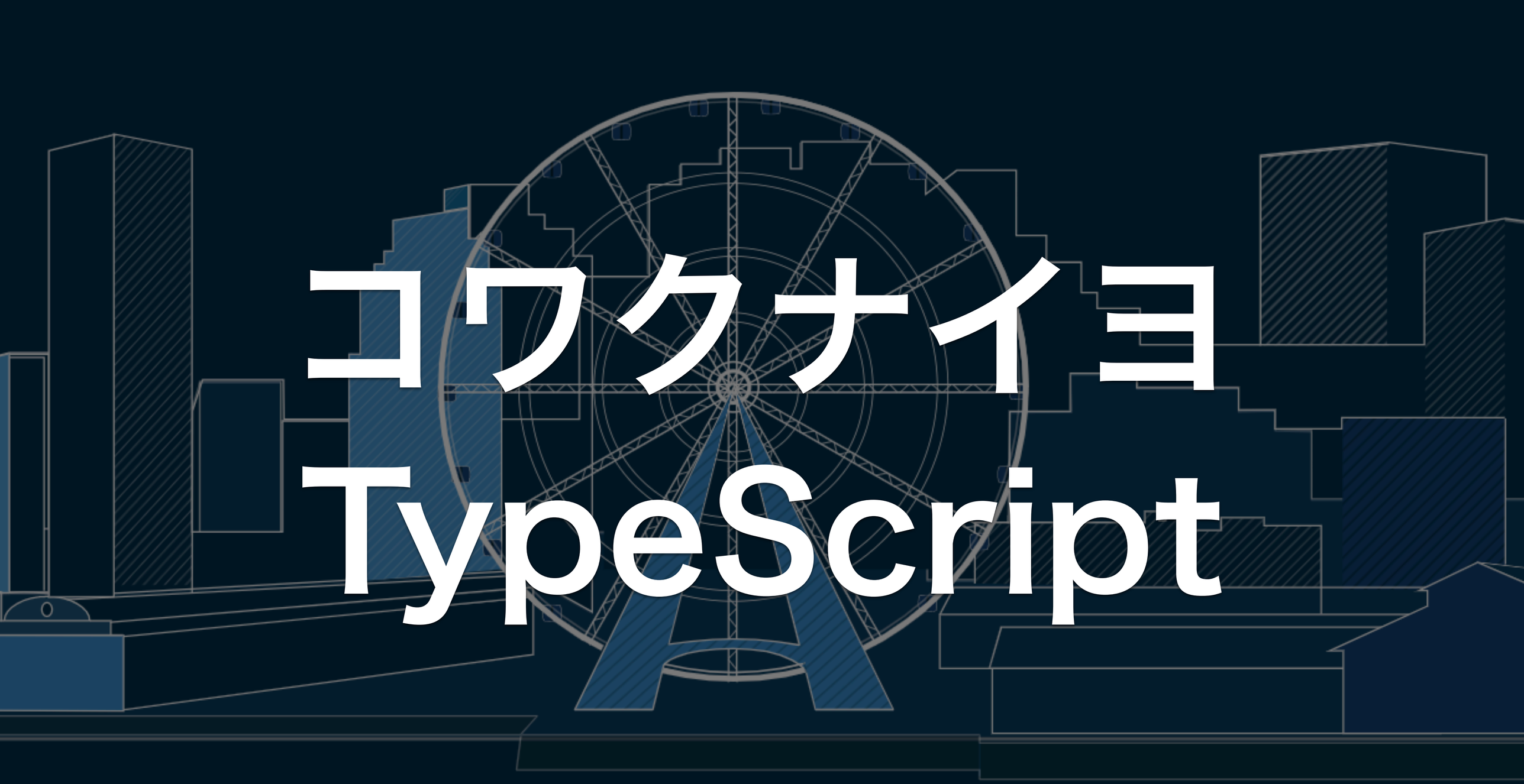
- **TypeScript 3.4**
  - インクリメンタルビルドのサポート、ジェネリクス関数の高階型推論、Readonlyの改良、constの型宣言、globalThisの型チェックサポート
- **TypeScript 3.3**
  - Union型の型推論の改良、増分ファイル監視のサポート
- **TypeScript 3.2**
  - strictBindCallApplyの追加、BigInt のサポート、外部パッケージの tsconfig.json 利用のサポートなど

最近のJavaScriptダイジェスト

- **クラス構文の private field が Chrome 74 に実装された**
  - TypeScript 側はまだトランスパイルの実装案はまだ検討中
  - private field の定義を独自で持っているので今後が気になりますね
  - ref: https://github.com/Microsoft/TypeScript/issues/9950
- **Node.js v12 で .mjs じゃなくても ESModule として動作するように**
  - package.json に "type": "module" を追加すると ESModule で動くパッケージに
  - 「--experimental-modules」がやっぱり必要

おおいよ!!

「TypeScriptを始めたいけど
ちょっとわからない…」


コワクナイヨ
TypeScript

@dojineko

よろず屋

fukuoka.ts をよろしくね！

早速ですが...時間の関係上...

- TypeScript の大まかな説明は下記のスライドに移譲します
  - 多分ネタの大部分がかぶってそう・・・いやそうに違いない！(´Д⊂ｸﾞｽﾝ
- 導入向けの内容はこっちを見てくれると嬉しいな (ちょっと古いです)

https://speakerdeck.com/dojineko/typescript-road-to-static-typing-ojisan

今回話すこと

- 移行に際してのtsconfig.json
- tsc と Babel
- any と unknown
- TSLint と ESLint
- strict: true への道

tsconfig.json に何を書けばいいの？

- **TypeScript を JavaScript に変換するときの設定を書きます**
  - Node.js用、ブラウザ用などの出力結果の切り替え
  - どのレベルまでのECMAScriptの機能を使うか
  - 試験的実装(Decoratorなど)の有効化
  - 型チェックのレベル
  - などなど....

Compiler Options

Compiler Options

Option	Type	Default	Description
--allowJs	boolean	false	Allow JavaScript files to be compiled.
--allowSyntheticDefaultImports	boolean	module === "system" or	Allow default imports from modules with no default export. This does not affect
--allowUnreachableCode	boolean	f	
--allowUnusedLabels	boolean	f	
--alwaysStrict	boolean	f	
--baseUrl	string		
--build -b	boolean	f	
--charset	string	"	
--checkJs	boolean	f	
--composite	boolean	t	
--declarationDir	string		Output directory for generated declaration files.
--declarationMap	boolean	false	Generates a sourcemap for each corresponding '.d.ts' file.
--diagnostics	boolean	false	Show diagnostic information.
--disableSizeLimit	boolean	false	Disable size limitation on JavaScript project.
--downlevelIteration	boolean	false	Provide full support for iterables in `for..of`, spread and destructuring when targeting ES5 or ES3.
--emitBOM	boolean	false	Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files.
--emitDeclarationOnly	boolean	false	Only emit '.d.ts' declaration files.
--emitDecoratorMetadata [1]	boolean	false	Emit design-type metadata for decorated declarations in source. See issue #2577 for details.
--esModuleInterop	boolean	false	Emit `__importStar` and `__importDefault` helpers for runtime babel ecosystem compatibility and enable `--allowSyntheticDefaultImports`

設定

大杉

tsconfig.json これだけ書いとけば大丈夫

- **JSからの移行の場合は、とにかくゆるめに設定するのがミソ！**
- いきなり「strict: true」はおすすめしません。
  - ロトの勇者だって「かわのたて」で出発するよね
- module は **"esnext"** がおすすめ「ガンガンいこうぜ」
  - せっかくやるならECMAScriptの新しい機能を使っていこう！(そのほうがロマンがある)
- target 設定はIEは無視できるなら es2015、だめなら es5
- 案件に応じて polyfill.io や babel-polyfill を併用しましょう

設定例 (IE11向けのゆるーい設定)

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": ["esnext", "dom"],
    "module": "esnext",
    "moduleResolution": "node",
    "esModuleInterop": true,
    "allowJs": true
  },
  "include": [
    "./src/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

出力結果はES5の範囲で

使う機能は最新の機能で

CommonJS と ESモジュールの違いを
なるべく意識しなくて良いようにする (Babel的なやつ)

TypeScript から JavaScript の
Import を許可する

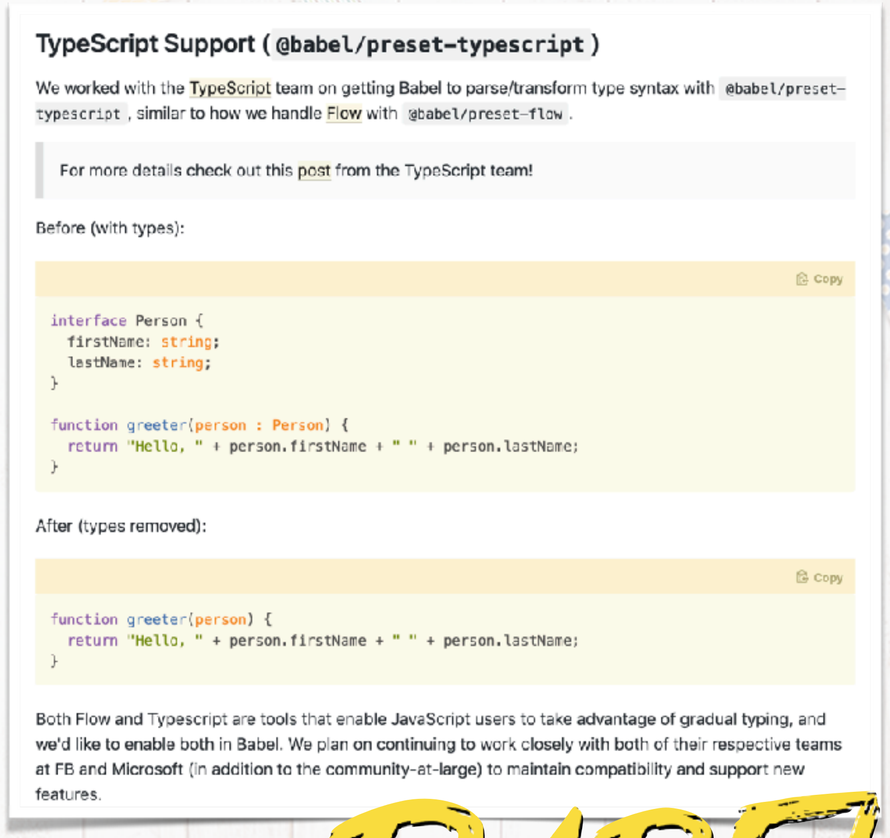
読み込むファイルを指定
ビルドをしたい対象を明示する

今回話すこと

- 移行に際してのtsconfig.json
- **TypeScript と Babel**
- any と unknown
- TSLint と ESLint
- strict: true への道

Babelさんの革命的アップデート

- **Babel7からTypeScriptの**
  トランスパイルが正式サポート
- Babelでトランスパイルする場合は
  一部の機能が使えません
  - namespace や JSX の型アノテーション、Enumの宣言のマージ、トランスパイル時の型チェックなど



TypeScript Support (`@babel/preset-typescript`)

We worked with the TypeScript team on getting Babel to parse/transform type syntax with `@babel/preset-typescript`, similar to how we handle Flow with `@babel/preset-flow`.

> For more details check out this post from the TypeScript team!

Before (with types):

Copy

```
interface Person {
  firstName: string;
  lastName: string;
}

function greeter(person : Person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}
```

After (types removed):

Copy

```
function greeter(person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}
```

Both Flow and Typescript are tools that enable JavaScript users to take advantage of gradual typing, and we'd like to enable both in Babel. We plan on continuing to work closely with both of their respective teams at FB and Microsoft (in addition to the community-at-large) to maintain compatibility and support new features.

tsc と Babel どっちを使えばいいの？

・tsc がおすすめな場合

- 型チェックを厳密に行いたい場合
- TypeScript の機能をフルに活用していきたい場合

・Babel がおすすめな場合

- すでにBabelのカスタマイズを高度に行っている場合
- 本質的に型チェックを必要としていない場合
- トランスパイル速度を重視する場合
  (型チェックだけ tsc を併用するパターンも有り)

導入するプロジェクトの状態を見て
とにかく無理のない選択を行いましょう
Babel を廃止して tsc を使う、
あるいはその逆も割と簡単に出来ます

今回話すこと

- 移行に際してのtsconfig.json
- tsc と Babel
- **any と unknown**
- TSLint と ESLint
- strict: true への道

unknown型とは

- TypeScript 3.0 で追加された新たな型
- 代入についてはany型と同じ
- unknown型の値を使用する場合は型アサーションが必要になる
- any型にすることで**型チェックが完全にガラ空きになる問題を解決するための手段の１つ**として使える
- TypeScirpt を手堅く運用するなら使っていきたい機能の１つ

使用例 (any型だったら)

```
// any に string[] を代入
const x: any = ['a', 'b', 'c']

// x は any なのでコード的にはOKになる
console.log(x.存在しない関数())
```

「存在しない関数」関数は実行時に
存在しないので undefined (ややこしい)

だけどもコンパイル時にチェックできない

使用例 (unknown型にすると)

```ts
// unknown に string[] を代入
const x: unknown = ['a', 'b', 'c']

// x は unknown なのでコード的にはNGになる
console.log(x.存在しない関数())
```

```
const x: unknown
オブジェクト型は 'unknown' です。 ts(2571)
クイック フィックス...  問題を表示
```

unknown型 をそのまま使っている箇所を
コンパイル時に検知できる
使用時には必ずアサーションが必要！

```ts
// unknown に string[] を代入
const x: unknown = ['a', 'b', 'c']

// x は string[] にアサーションされるのでエラーがわかる
console.log((x as string[]).存在しない関数())
```

結果存在しない関数を呼び出している
ような箇所を検知することができる

今回話すこと

- 移行に際してのtsconfig.json
- tsc と Babel
- any と unknown
- **TSLint と ESLint**
- strict: true への道

ここは大事なとこだけ

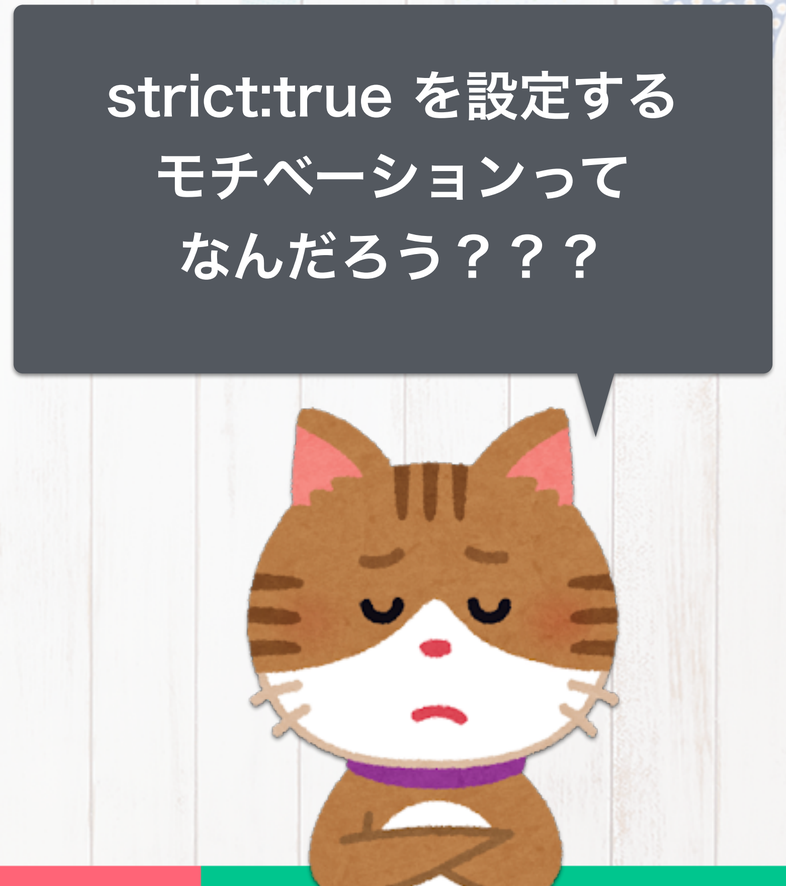
- **TSLint の使用は非推奨になりました**
  - 新規案件はESLintで始めることをおすすめします
- ESLint の TypeScript サポートはまだまだこれから
  - no-floating-promise のような TypeScript の型情報に基づく Linting は完全には移行できないので場合によってはまだ様子見していおいたほうが良いかも？
- とはいえESLintに移行しなければなくなるのは時間の問題
  - 暇があれば移行しておきましょう！
  - 担当プロジェクトではひとまず**移行の準備だけ済ませていますが TSLint がまだ現役**です。

時間足りてるのかな？

- 移行に際してのtsconfig.json
- tsc と Babel
- any と unknown
- TSLint と ESLint
- **strict: true への道**

より手堅く運用していくポイント

- unknown を使っていく
  - anyを使うより、unknownを使うのがおすすめ
- 型定義を使う、無い定義は書く
  - 新規の定義や、定義の更新はPRするとみんなも幸せになれる
- **tsconfig.json で strict:true を設定する**
  - より強く制約を掛けて運用するお気持ちの表明
  - 必ずしも必要ではないが手堅くはなる



strict: true の効果とは？

- 実は手堅く運用するための複数のルールがまとめて適用される
  - --noImplicitAny: **暗黙の any の使用を禁止する**
  - --noImplicitThis: **thisの型を明示することを要求する**
  - --alwaysStrict: **出力結果に常に "use strict" を指定する**
  - --strictBindCallApply: **bind, call, applyの型推論を厳密に行う**
  - --strictNullChecks: **厳密な null のチェックを行う**
  - --strictFunctionTypes: **厳密な関数の型のチェックを行う**
  - --strictPropertyInitialization: **初期化されていないクラスプロパティを禁止する**

バージョンによって適用される
ルールは若干異なります

strict: true を適用することは良いこと？

- 適用できるならしておいたほうが手堅くはなるし良いこと！
- しかし**必須ではないし、ゴールでもない**
- 適用しないから悪いということもない
- **理解を得ながら、長期的に、無理なく運用できるようにしていくことが大事**
- strict: true にする場合は個別のルールを適用していき制約を徐々に強めていくことをおすすめします

strict: true への道

1. **strict: true の試験適用と現状把握**

   ・問題点を把握するためだけに一時的に適用

2. **noImplicitAny 以外の適用**

   ・暗黙の any 使用箇所が多々あったため

3. **noImplicitAny の適用**

   ・つけては外しを繰り返して問題箇所を駆逐

4. **個別適用を strict: true へ置き換えてまとめる**

5. **以後は strict: true での運用を継続 (現在に至る)**

strict: true を適用したことによる効果

- 制約が強化されるため、より**エラー検知の精度が上がりました**
- ただし、制約が強化されたことによりTypeScript(=静的型付け言語)への「こなれ感」が必要になりました
- JavaScript から移ってきたばかりだと、ストレスに感じる部分が出てくるかもしれません (今は殆どそんなこと無いです)
- もしやるなら「きょうせいギプス」をつけて**練習してる理解で臨みましょう**

まとめ

まとめ

- TypeScriptはコワくない！
  - ゆるーく始めても全然OK！
  - なんなら型定義だけ使ってエディタで補完させるだけでもOK！
- Babel でも TypeScript は使えるよ
- any型よりはunknown型を使うのがおすすめ
- TSLintはESLintに置き換えていく準備をしましょう
- strict:true は便利だけど痛みを伴うので
  用法用量を守って正しくお使いください

ご清聴ありがとうございました