



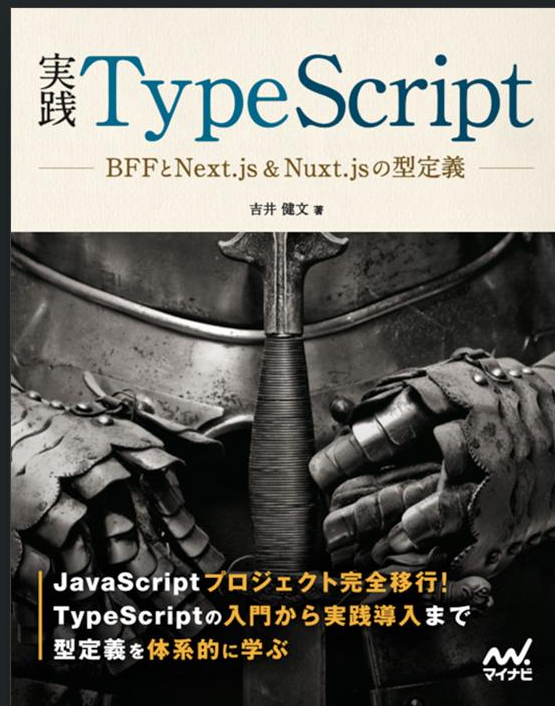
# TypeScript の流儀

---

Bonfire Frontend #4 @Takepepe

# About Me

- Takefumi Yoshii / @Takepepe
- DeNA / DeSC Healthcare
- Frontend Engineer
- 実践TypeScript 著者



# 本日の対象聴講者

- JavaScript でひとつとおりApp開発ができる
- TypeScript を少し触ったことがある
- 型定義について細かい疑問点がある

全体的に入門的な内容になっています。

わかりやすいハンズオンの資料を心がけました。

# TypeScript のよくある疑問

- 型が何を担保し、プログラマはどう扱うべきなのか？
- 巷でタブーとされるあの記法は、なぜ悪いのか？
- 複雑な型定義が、なぜ必要になるのか？

当資料では「TypeScript の流儀」と称し、  
これらの疑問に対する「私的見解」を言語化していきます。

# Agenda

- 1. 型推論いろは
- 2. 攻防一体・型の策略
- 3. コンパイラの合意
- 4. 型の主従関係
- 5. 源流を辿る型定義

# 1. 型推論いろは

---

# 実装推論

TypeScript は実装内容に則り、型推論(実装推論)が得られます。  
ここに、意図的に付与した型情報はありません。

```
function greet() {  
  return 'hello'  
}  
  
const msg = greet() // const msg: string
```

# 宣言 (Assertion)

意図的に Assertion が付与されている場合、  
値は宣言された型であると解釈します。

```
function greet() {  
  return 'hello' as any // any型とする Assertion  
}  
  
const msg = greet() // const msg: any
```



# 注釈(Annotation)

意図的に Annotation が付与されている場合、  
実装はその型に従わなければいけません。

```
function greet(): void { // 戻り値がないことを表す void型
  return 'hello' // Error! 値を返してはいけない
}
const msg = greet() // const msg: void
```

# const / let の推論

変数宣言の `const / let`。

初期代入値から、適切な型が推論されます。

```
let msg1 = 'msg'
```

```
const msg2 = 'msg'
```

## const / let の推論

`let` は再代入可能なため「string 型」に、  
`const` は再代入不可なため「"msg"型 (String Literal 型)」に。

```
let msg1 = 'msg' // let msg1: string  
const msg2 = 'msg' // const msg2: "msg"
```

# Null 許容型

複数の型を表す Union Types。TypeScript では、  
Nullable型 (Null許容型) も、Union Types で表現します。

```
function greet(name: string | null) {  
  let user = 'USER'  
  if (name !== null) {  
    user = name.toUpperCase() // (parameter) name: string  
  }  
  console.log(`HELLO ${user}!`)  
}
```

# Null 許容型

複数の型をパイプ|で連結し、  
引数は「string型 または null型」どちらかの型であることを表します。

```
function greet(name: string | null) {  
  let user = 'USER'  
  if (name !== null) {  
    user = name.toUpperCase() // (parameter) name: string  
  }  
  console.log(`HELLO ${user}!`)  
}
```

# ガード節

null や undefined を安全に扱う手法「ガード節」。  
ガード節を通過したブロックでは、型が絞り込まれます。

```
function greet(name: string | null) {  
  let user = 'USER'  
  if (name !== null) { // ガード節  
    user = name.toUpperCase()  
  }  
  console.log(`HELLO ${user}!`)  
}
```

# ガード節

型が絞り込まれると、そのインスタンスに備わった  
プロパティ・メソッドへのアクセスが安全なものであると解釈されます。

```
function greet(name: string | null) {  
  let user = 'USER'  
  if (name !== null) { // ガード節  
    user = name.toUpperCase() // (parameter) name: string  
  }  
  console.log(`HELLO ${user}!`)  
}
```

# ガード節

三項演算子を用いたガード節でも、型が絞り込まれます。  
このように「実装内容で型を絞り込み」安全に値を扱います。

```
function greet(name: string | null) {  
  const user = name !== null ? name.toUpperCase() : 'USER'  
  console.log(`HELLO ${user}!`)  
}
```



# タグ付き Union Types

次の User型 は「UserA型・UserB型・UserC型」からなる Union Types であり、共通のプロパティを保持しています。

```
type UserA = { gender: 'male'; name: string }  
type UserB = { gender: 'female'; age: number }  
type UserC = { gender: 'other'; graduate: string }  
type User = UserA | UserB | UserC
```

# タグ付き Union Types

共通プロパティ「gender」の型は「'male'・'female'・'other'」型  
それぞれ異なる「String Literal 型」です。

```
type UserA = { gender: 'male'; name: string }  
type UserB = { gender: 'female'; age: number }  
type UserC = { gender: 'other'; graduate: string }  
type User = UserA | UserB | UserC
```

# タグ付き Union Types

User 型のように、Literal 型で区別できる Union Types は、「タグ付き Union Types」と呼びます。(別名: Discriminated Unions)

```
type UserA = { gender: 'male'; name: string }
type UserB = { gender: 'female'; age: number }
type UserC = { gender: 'other'; graduate: string }
type User = UserA | UserB | UserC
```

# タグ付き Union Types

タグ付き Union Types が付与された値を分岐にかけると、分岐ブロック内で、型が絞り込まれます。

```
function judgeUserType(user: User) {  
  switch (user.gender) {  
    case 'male':  
      const u0 = user // (parameter) user: UserA  
      break  
    case 'female':  
      const u1 = user // (parameter) user: UserB  
      break  
    case 'other':  
      const u2 = user // (parameter) user: UserC  
      break  
    default:  
      const u3 = user // (parameter) user: never  
  }  
}
```

# タグ付き Union Types

これにより、各型にしか保持していないプロパティであっても、安全なアクセスであると解釈されます。

```
function judgeUserType(user: User) {  
  switch (user.gender) {  
    case 'male':  
      const u0 = user.name // const u0: string  
      break  
    case 'female':  
      const u1 = user.age // const u1: number  
      break  
    case 'other':  
      const u2 = user.graduate // const u2: string  
      break  
    default:  
      const u3 = user // const u3: never  
  }  
}
```

## 1. 型推論いろは

- 型情報がなくても、実装に型はついて回る
- 型推論は JavaScript の構文をなぞらせる

## 2. 攻防一体・型の策略

---

# コンパイラへ策略を通達する

実装推論だけでは、現実のアプリケーションコードで不十分です。

その値がどの様にプログラムで利用されるのか？

という「策略」は、プログラマしか知り得ません。

型の付与は「コンパイラへ策略を通達すること」と言えます。



# 守りの策略・Annotation

Annotation 付与は「守りの策略」と言い換えることができます。  
型をあらかじめ付与し、要件を先に取り決めてしまいます。

```
const str1: any = 'str'  
const str2: string = 'str'  
const str3: 'str' = 'str'  
const str4: 'str' = 'literal' // !Error
```

# 守りの策略・Annotation

コンパイラもプログラマも、これに従います。

策略に基づき、引数・変数・戻り型に付与します。

```
function greet1(message: string) {}  
function greet2(message: 'hello') {}  
greet1('HELLO')  
greet2('HELLO') // !Error
```

# 守りの策略・Annotation

インデックスシグネチャとよばれる型を付与すると、  
オブジェクトプロパティの型を一律で制約することができます。

```
type Functions = { [k: string]: Function } // インデックスシグネチャ
const funcs: Functions = {
  f1: () => true,
  f2: async () => false,
  s1: 'str' // Error! 関数として評価できない
}
```

## 攻めの戦略・Assertion

次の配列プロパティは「never」配列と推論されてしまい、このままでは何も追加することができません。

```
const state = {  
  count: 0,  
  flag: false,  
  arr: []  
}
```



```
const state: {  
  count: number;  
  flag: boolean;  
  arr: never[]; // 望まない推論結果  
}
```

## 攻めの戦略・Assertion

実装推論では測れない部分的補足として「型解釈のヒント」を付与します。

Assertion 付与は「攻めの戦略」と言い換えることができます。

```
const state = {  
  count: 0,  
  flag: false,  
  arr: [] as string[]  
}
```



```
const state: {  
  count: number;  
  flag: boolean;  
  arr: string[]; // 望みどおりの推論結果  
}
```

# アップキャスト・ダウンキャスト

互換性が成立する場合「広義な型・詳細な型」として、双方解釈することが可能です。これを「アップキャスト・ダウンキャスト」と呼びます。

```
let myName = 'taro'  
const name1 = myName           // const name1: string  
const name2 = myName as 'taro' // const name2: "taro"   ダウンキャスト  
const name3 = myName as any    // const name3: any     アップキャスト
```

# アップキャスト・ダウンキャスト

互換性チェックは構造的部分型に基づくため、次のような誤ったダウンキャストを行っても、コンパイラに責任はありません。

```
let myName = 'taro'  
const name1 = myName           // const name1: string  
const name2 = myName as 'jiro' // const name2: "jiro" 型が誤っている
```

# アップキャスト・ダウンキャスト

広義な型にアップキャストした直後、詳細な型にダウンキャストする

「Double Assertion」という手法があります。

100%プログラマが正しい場合に利用しなければならない苦肉の策です。

```
const store = new Store() as any as StrictStore
```



## 2. 攻防一体・型の策略

- 型は束縛されるものではなく、策略を練るもの
- 策略の通達は、必要に応じて随時行う

### 3. コンパイラの合意

---

# コンパイラの合意とは？

攻めの通達には、様々な方法が用意されています。  
「この型であって欲しい」という通達を行うためには、  
必要最低限の条件を満たす必要があります。

当資料ではこれを「コンパイラの合意」と称します。

# User Defined Type Guard

ランタイム挙動をなぞらえた型推論は、完璧ではありません。

例えば次の変数「`users`」から、男性のみをフィルタリングしてみます。

```
type Male = { id: string; gender: 'male' }
type Female = { id: string; gender: 'female' }
type User = Male | Female

const users: User[] = [
  { id: '1', gender: 'male' },
  { id: '2', gender: 'female' },
  { id: '3', gender: 'male' }
]
```

# User Defined Type Guard

現在の `Array.filter` の推論では、型を絞り込む事はできません。  
ランタイムの挙動と同じように「`const males: Male[]`」が望まれます。

```
const males = users.filter(user => {  
  return user.gender === 'male'  
}) // const males: User[]; 望まない推論結果
```

# User Defined Type Guard

ここに「`: user is Male`」という戻り型 Annotation を付与することで、  
後続の型解釈を操作することができます。

```
const males = users.filter((user): user is Male => {  
  return user.gender === 'male'  
}) // const males: Male[]; 望み通りの推論結果
```

# User Defined Type Guard

注意しなければならないのは「コンパイラに責任はない」ということです。  
boolean型さえ返していればよく、実装内容の正しさに関与しません。

```
const males = users.filter((user): user is Male => {  
  return user.gender === 'female' // oops!  
}) // const males: Male[]; 誤った推論結果
```

# User Defined Type Guard

User Defined Type Guard を利用する場合、  
プログラマが「型安全」を肩代わりしなければいけません。

```
const males = users.filter((user): user is Male => {  
  return user.gender === 'female' // oops!  
}) // const males: Male[]; 誤った推論結果
```

boolean型さえ返却すれば、コンパイラは合意します。



# Non-null assertion

インラインで型を絞りこむ「Non-null assertion」。

「!」を利用することで「null | undefined」が振るい落とされます。

```
const msg = 'hello' as string | null
const nullable = msg // const nullable: string | null
const nonNullable = msg! // const nonNullable: string
```

# Non-null assertion

Non-null assertion は「コンパイラを欺く悪い慣習」という印象があります。  
次のコードをみれば、この危険性もうなずけます。

```
const msg1 = 'str' as string | null
const msg2 = 'str' as string | null
const msg3 = null as string | null
msg1.toUpperCase() // コンパイルエラーになるが、ランタイムエラーにならない
msg2!.toUpperCase() // コンパイルエラーにならず、ランタイムエラーにならない
msg3!.toUpperCase() // コンパイルエラーにならず、ランタイムエラーになる
```

# Non-null assertion

しかしながら、Non-null assertion は悪い慣習とは限りません。  
特定のケースにおいて、有効なことがあります。

```
const msg1 = 'str' as string | null
```

```
const msg2 = 'str' as string | null
```

```
const msg3 = null as string | null
```

```
msg1.toUpperCase() // コンパイルエラーになるが、ランタイムエラーにならない
```

```
msg2!.toUpperCase() // コンパイルエラーにならず、ランタイムエラーにならない
```

```
msg3!.toUpperCase() // コンパイルエラーにならず、ランタイムエラーになる
```

# Non-null assertion

それは、コンパイラよりプログラマのほうが型について詳しいケースです。  
例えば、HTML に記述された要素にイベントをバインドするコードです。

```
<html lang="en">  
  <head></head>  
  <body>  
    <button id="btn"><%= count %></button>  
  </body>  
</html>
```

# Non-null assertion

「`document.getElementById`」の戻り型は `Nullable` 型です。  
そのため、次の記述ではコンパイルエラーとなります。

```
// getElementById(elementId: string): HTMLElement | null;  
document.getElementById('btn').addEventListener('click', () => {}) // Error
```

# Non-null assertion

「その要素が存在するか否か」コンパイラは担保できないが、プログラマーが担保している様なケースに限り、Non-null assertion は有効です。

```
// getElementById(elementId: string): HTMLElement | null;  
document.getElementById('btn')!.addEventListener('click', () => {})
```

# Non-null assertion

「Non-null assertion」はコンパイラを欺くためのものではなく、  
「品質担保します」という意思表示に他なりません。

```
// getElementById(elementId: string): HTMLElement | null;  
document.getElementById('btn')!.addEventListener('click', () => {})
```

「プログラマが品質担保します」という署名を信じ、コンパイラは合意します。

## const assertion

const assertion は、初期値を厳格な型として保持する「署名」です。  
変数初期値が「より厳格であってほしい」という意図の通達に利用できます。

```
const user1 = 'taro'           // const user1: "taro"  
let user2 = 'taro'            // let user2: string  
let user3 = 'taro' as const   // let user3: "taro"
```



## const assertion

この署名を行った場合、JavaScript 本来の挙動と異なる「厳格さ」が与えられてしまうことに注意しなければいけません。

```
let user2 = 'taro'           // let user2: string
let user3 = 'taro' as const // let user3: "taro"
user2 = 'TAR0'
user3 = 'TAR0' // Error; JavaScript とは異なる挙動
```

## const assertion

この署名を行った場合、JavaScript 本来の挙動と異なる「厳格さ」が与えられてしまうことに注意しなければいけません。

```
let user2 = 'taro'           // let user2: string
let user3 = 'taro' as const // let user3: "taro"
user2 = 'TARO'
user3 = 'TARO' // Error; JavaScript とは異なる挙動
```

「JSの挙動と異なってもよい」という署名を信じ、コンパイラは合意します。

### 3. コンパイラの合意

- 攻めの策略には、隙が生まれることを心得る
- 合意に基づき、プログラマが品質を担保する

## 4. 型の主従関係

---

## 上流・下流の意識

実装しているコードが「上流工程なのか・下流工程なのか」の意識は型推論に関する重要事項です。

なぜなら、合意を得られた型が、上流から流れてくるからです。  
手短で・厳格な型推論を得るコツを紹介します。

## 「頑張る = 厳格」とは限らない

次の関数定義において与えられた型情報は、  
引数の Annotation「: number」のみです。

```
import { SET_COUNT } from './actionTypes'  
export function setCount(amount: number) {  
  return { type: SET_COUNT, payload: { amount } }  
}
```

## 「頑張る = 厳格」とは限らない

それでいて、戻り型まで厳格な型 (String Literal 型) が得られています。

"LONG\_PREFIX\_SET\_COUNT"型 は、この定義内のどこにもありません。

```
import { SET_COUNT } from './actionTypes'
export function setCount(amount: number) {
  return { type: SET_COUNT, payload: { amount } }
}
// function setCount(amount: number): {
//   type: "LONG_PREFIX_SET_COUNT"; payload: { amount: number; };
// }
```

# 「頑張る = 厳格」とは限らない

この String Literal 型は、上流工程で既に定められていたものです。

次の様に const assertion が付与されていました。

```
export = {  
  INCREMENT: 'LONG_PREFIX_INCREMENT',  
  DECREMENT: 'LONG_PREFIX_DECREMENT',  
  SET_COUNT: 'LONG_PREFIX_SET_COUNT'  
} as const
```



## 「頑張る = 厳格」とは限らない

下流工程では、そのまま受け流すことで正しく伝搬することができます。

「下流ではむしろ型を付与しない方が良い」ことがわかります。

```
export = {  
  INCREMENT: 'LONG_PREFIX_INCREMENT',  
  DECREMENT: 'LONG_PREFIX_DECREMENT',  
  SET_COUNT: 'LONG_PREFIX_SET_COUNT'  
} as const
```

# 「上流下流 = 依存関係 = 型の主/従」

依存関係は、型の主従関係そのものです。

次の様なヘルパー関数として定義された純関数は、依存がありません。

```
export function isNumberLikeString(value: string) {  
  return !value.match(/[^-^0-9^.] /g)  
} // function isNumberLikeString(value: string): boolean
```

# 「上流下流 = 依存関係 = 型の主/従」

「上流工程なのか・下流工程なのか」は一目瞭然で、  
ファイル上部の `import` を見ればすぐにわかります。

```
export function isNumberLikeString(value: string) {  
  return !value.match(/[^-^0-9^.] /g)  
} // function isNumberLikeString(value: string): boolean
```

## 「上流下流 = 依存関係 = 型の主/従」

何も import していなければ、そこは最上流ということができます。

型定義だけでなく「実装そのもの」が最上流になり得ます。

```
export function isNumberLikeString(value: string) {  
  return !value.match(/[^-^0-9^.] /g)  
} // function isNumberLikeString(value: string): boolean
```

「型定義 > 実装」ではなく「上流 > 下流」である

## 4. 型の主従関係

- 型定義が上流工程とは限らない
- 依存関係が型の主従関係そのものである

## 5. 源流を辿る型定義

---

## その定義は、伝言ゲームになっていないか？

中流工程において「攻めの策略」が誤っていた場合、  
本来の正しい型が覆されるリスクがあることは先に述べたとおりです。  
全工程において「策略のルーツ」を伝搬することが望ましいです。

複雑な型定義の需要は、ここに帰結します。

# 伝家の宝刀、typeof 型クエリー

typeof キーワードを用いることで、  
定義済みの実装から「型を読み取る」ことができる型クエリー。  
Assertion による型の補足もそのまま引き継がれます。

```
type UserState = typeof userState
const userState = {
  user_id: '',
  name: '',
  tasks: [] as Task[]
}
```



# 伝家の宝刀、typeof 型クエリー

「実装推論と同じ」ということができます。  
同等の型定義を付与してまわるよりも、  
正確な型情報を導出することができます。

```
type UserState = typeof userState
const userState = {
  user_id: '',
  name: '',
  tasks: [] as Task[]
}
```



```
type UserState = {
  user_id: string;
  name: string;
  tasks: Task[];
}
```

# 中流構築が扱う Utility Types

この `typeof` キーワードで導出した型を加工してみます。

「`Partial`」は全てのプロパティを「`Optional`」に変換する Utility Types です。

```
type UserState = typeof userState
type Injects = Partial<UserState>
```



```
type Injects = {
  user_id?: string | undefined;
  name?: string | undefined;
  tasks?: Task[] | undefined;
}
```

# 中流構築が扱う Utility Types

TypeScript にあらかじめビルトインされた「Utility Types」を利用すると、既出の型から新しい型定義を創出することができます。

```
type UserState = typeof userState  
type Injects = Partial<UserState>
```



```
type Injects = {  
  user_id?: string | undefined;  
  name?: string | undefined;  
  tasks?: Task[] | undefined;  
}
```

## 中流構築が扱う Utility Types

この型の使い所は、例えば次の様なファクトリ関数です。  
デフォルト値と、オプションで注入する値をマージしたうえで、  
「UserState」型と齟齬のない値を返却します。

```
function userStateFactory(injects?: Injects): UserState {  
  return { ...userState, ...injects }  
}
```

# 中流構築が扱う Utility Types

`typeof` キーワードは宣言済み変数のみならず、関数にも適用できます。  
「`ReturnType`」も、ビルトイン Utility Types のひとつです。

```
type StoreState = {  
  user: ReturnType<typeof userStateFactory>  
  app: ReturnType<typeof appStateFactory>  
}
```

## 中流構築が扱う Utility Types

次の例は、先に定義済みの「userStateFactory」関数を参照し、その関数戻り型を抽出する「ReturnType」を併せて構築した型です。

```
type StoreState = {  
  user: ReturnType<typeof userStateFactory>  
  app: ReturnType<typeof appStateFactory>  
}
```

# 中流構築が扱う Utility Types

推論で得られる型は次のとおりです。

```
type StoreState = {  
  user: {  
    user_id: string;  
    name: string;  
    tasks: Task[];  
  };  
  app: {  
    initalized: boolean;  
    isConnected: boolean;  
  };  
}
```

## 中流構築が扱う Utility Types

ここまでで型定義は3つ。「UserState・Injects・StoreState」です。全ての型に含まれる「`user_id: string`」のルーツは「`const userState`」でした。ルーツを参照しているので、次のリファクタは全てに伝搬します。

```
const userState = {  
  user_id: '',  
  name: '',  
  tasks: [] as Task[]  
}
```



```
const userState = {  
  member_id: null as string | null,  
  name: '',  
  tasks: [] as Task[]  
}
```



# Conditional Types が強力なわけ

「Conditional Types」は 型の三項演算子です。

Generics に与えた型「**T**」が、比較対象型「**number**」と互換性がある場合、任意型を導きます。

```
type IsNumber<T> = T extends number ? true : false
type T1 = IsNumber<1>    // type T1 = true
type T2 = IsNumber<'2'> // type T2 = false
```

# Conditional Types が強力なわけ

Conditional Types では、比較対象型の「部分導出」が可能です。  
組み込み Utility Types の「ReturnType」も、これを利用しています。

```
type ReturnType<T> = T extends (...args: any) => infer I ? I : any
```

# Conditional Types が強力なわけ

比較対象型内で「infer I」が指定されている場所が、  
導出対象です。戻り型を指していることが分かります。

```
type ReturnType<T> = T extends (...args: any) => infer I ? I : any
```



比較対象型

# Conditional Types が強力なわけ

Generics に与えられた型「**T**」が、関数として評価できる場合、「infer **I**」に相当する型を導出するということです。

```
type ReturnType<T> = T extends (...args: any) => infer I ? I : any
```

## Conditional Types が強力なわけ

先の例では「戻り型」を導出していましたが、  
例えば「関数第二引数型」の導出などが可能になります。

```
type Arguent2<T> = T extends (a1: any, a2: infer I) => any ? I : never
```

# Conditional Types が強力なわけ

「Conditional Types」を組み合わせれば、  
複雑に入り組んだ源流であっても、辿ることができます。

```
type Arguent2<T> = T extends (a1: any, a2: infer I) => any ? I : never
```

## 5. 源流を辿る型定義

- 下流工程はそのまま受け流すことが最も厳格
- 複雑な型定義は、源流を辿るためにある

# TypeScript の流儀「十訓」

- 型情報がなくても、実装に型はついて回る
- 型推論は JavaScript の構文をなぞらせる
- 型は束縛されるものではなく、策略を練るもの
- 策略の通達は、必要に応じて随時行う
- 攻めの策略には、隙が生まれることを心得る
- 合意に基づき、プログラマが品質を担保する
- 型定義が上流工程とは限らない
- 依存関係が型の主従関係そのものである
- 下流工程はそのまま受け流すことが最も厳格
- 複雑な型定義は、源流を辿るためにある



**ご静聴ありがとうございました**

---