TypeScriptの型システム

なぜTypeScriptを使うのか？について説明したとき、TypeScriptの型システムの主な機能を取り上げました。下記は、改めて説明する必要がない、いくつかのキーポイントです：

- TypeScriptの型システムは使うかどうか選べるものとして設計されているので、あなたが書いた*JavaScript*は*TypeScript*でもあります。
- TypeScriptは型エラーがあってもJavaScriptの生成をブロックしないので、徐々に*JS*を*TS*に更新していくことができます。

では、TypeScript型システムの構文から始めましょう。これにより、コード内でこれらのアノテーションをすぐに使用して、その利点を確認することができます。これは後で詳細を掘り下げる準備にもなります。

基本アノテーション

前述のように、 `:TypeAnnotation` 構文を使って型アノテーションを書きます。型宣言空間で使用可能なものは、型アノテーションとして使用できます。

次の例は、変数、関数パラメータ、および関数戻り値の型アノテーションを示しています。

```
var num: number = 123;
function identity(num: number): number {
    return num;
}
```

プリミティブ型(Primitive Types)

JavaScriptプリミティブ型は、TypeScript型システムでカバーしています。これは、以下に示すように `string` 、 `number` 、 `boolean` を意味します：

```
var num: number;
var str: string;
var bool: boolean;

num = 123;
num = 123.456;
num = '123'; // Error

str = '123';
str = 123; // Error

bool = true;
bool = false;
bool = 'false'; // Error
```

配列(Arrays)

TypeScriptは、配列に専用の構文を提供し、コードにアノテーションを付けて文書化するのを容易にします。構文は、基本的に `[]` を有効な型アノテーションに後置します(例えば `:boolean[]`)。これは通常行う配列操作を安全に行うことを可能にし、誤った型のメンバを割り当てるなどのエラーからあなたを守ります。これは以下のとおりです：

```
var boolArray: boolean[];

boolArray = [true, false];
console.log(boolArray[0]); // true
console.log(boolArray.length); // 2
boolArray[1] = true;
boolArray = [false, false];

boolArray[0] = 'false'; // Error!
boolArray = 'false'; // Error!
boolArray = [true, 'false']; // Error!
```

インターフェース(Interfaces)

インターフェースは、複数の型アノテーションを単一の名前付きアノテーションに合成するための、TypeScriptにおける主要な方法です。次の例を考えてみましょう。

```
interface Name {
    first: string;
```

```
    second: string;
}

var name: Name;
name = {
    first: 'John',
    second: 'Doe'
};

name = {            // Error : `second` is missing
    first: 'John'
};
name = {            // Error : `second` is the wrong type
    first: 'John',
    second: 1337
};
```

ここでは、アノテーションを `first: string + second: string` という新しいアノテーション `Name` にまとめて、個々のメンバの型チェックを行っています。インターフェースはTypeScriptで大きなパワーを持っているので、別途専用のセクションでその利点をどのように活かすかを説明します。

インライン型アノテーション(Inline Type Annotation)

新しい `interface` を作成するのではなく、構造 `:{/* Structure */}` を使ってインラインで必要なものにアノテーションを付けることができます。前の例を、インライン型で再掲します：

```
var name: {
    first: string;
    second: string;
};
name = {
    first: 'John',
    second: 'Doe'
};

name = {            // Error : `second` is missing
    first: 'John'
};
name = {            // Error : `second` is the wrong type
    first: 'John',
    second: 1337
};
```

インライン型は、1回だけ使うようなアノテーションを素早く提供するのに最適です。それは(良くないかもしれない)型名を考える手間を省きます。しかし、同じ型アノテーションを複数回インラインで入れている場合は、それをインターフェース(またはこのセクションの後半で説明する `type alias`)にリファクタリングすることを検討することをお勧めします。

特殊な型

上記でカバーしたプリミティブ型以外にも、TypeScriptでは特別な意味を持ついくつかの型があります。これらは `any` 、 `null` 、 `undefined` 、 `void` です。

any

`any` 型は、TypeScript型システムにおいて特別なものです。これは、型システムからの脱出口を与えて、コンパイラに失せるよう指示します。 `any` は型システムのすべての型と互換性があります。つまり、any型の変数には何でも代入できるし、その変数を何にでも代入できるということです。以下に例を示します:

```
var power: any;

// Takes any and all types
power = '123';
power = 123;

// Is compatible with all types
var num: number;
power = num;
num = power;
```

JavaScriptコードをTypeScriptに移植する場合、最初は `any` と友達になります。しかし、型安全性を確保することはあなた次第であるため、この友情を真剣に受け止めてはいけません。これを使うことは、基本的に、コンパイラに意味のある静的解析を行わないように指示することです。

`null` と `undefined`

`strictNullChecks` フラグ(このフラグについては、後で扱います)によって扱いが変わります。 `strictNullChecks` が `false` の場合、 `null` と `undefined` のJavaScriptリテラルは、型システム

においては `any` 型と同じようなものとして扱われます。これらのリテラルは他の型に代入することができます。以下に例を示します：

```
var num: number;
var str: string;

// These literals can be assigned to anything
num = null;
str = undefined;
```

:void

関数に戻り値の型がないことを示すには `:void` を使います：

```
function log(message): void {
    console.log(message);
}
```

ジェネリックス(Generics)

コンピュータサイエンスの多くのアルゴリズムとデータ構造は、オブジェクトの実際の型に依存しません。しかし、それでも、あなたは、さまざまな変数の間で制約を適用したいと考えているでしょう。単純なおもちゃの例は、項目のリストを取り、逆順にした項目のリストを返す関数です。ここでの制約は、関数に渡されるものと関数によって返されるものの間の制約です。

```
function reverse<T>(items: T[]): T[] {
    var toreturn = [];
    for (let i = items.length - 1; i >= 0; i--) {
        toreturn.push(items[i]);
    }
    return toreturn;
}

var sample = [1, 2, 3];
var reversed = reverse(sample);
console.log(reversed); // 3,2,1

// Safety!
```

```
reversed[0] = '1';      // Error!
reversed = ['1', '2']; // Error!

reversed[0] = 1;        // Okay
reversed = [1, 2];      // Okay
```

ここでは、 `reverse` 関数は何らかの型 `T` の配列(`items: T[]`)を受け取り(`reverse<T>` の型パラメータに注目)、 `T` 型の配列を返します(`: T[]` に注目)。 `reverse` 関数は、同じ型の項目を返すので、 `reversed` 変数も `number[]` 型であることがTypeScriptに分かるため、型の安全性が得られます。同様に `string[]` の配列を `reverse` 関数に渡すと、返される結果も `string[]` の配列になり、以下に示すような型安全性が得られます：

```
var strArr = ['1', '2'];
var reversedStrs = reverse(strArr);

reversedStrs = [1, 2]; // Error!
```

実際、JavaScript配列には既に `.reverse` 関数があり、TypeScriptはジェネリックを使ってその構造を定義しています：

```
interface Array<T> {
 reverse(): T[];
 // ...
}
```

これは、以下のように任意の配列で `.reverse` を呼び出すときに型の安全性を得られることを意味します：

```
var numArr = [1, 2];
var reversedNums = numArr.reverse();

reversedNums = ['1', '2']; // Error!
```

後で、**アンビエント宣言**(Ambient Declarations)の節で `lib.d.ts` を説明するときに、 `Array<T>` インターフェースについてもっと議論します。

ユニオン型(Union Type)

JavaScriptでは、プロパティを複数の型のうちの1つにしたいことがよくあります(例: `string` または `number`)。そういった場合、ユニオン型(型アノテーションの `|` を使い `string|number` のように書く)が便利です。よくある使用例として、単一のオブジェクトまたはオブジェクトの配列をとることができる関数があげられます。

```
function formatCommandline(command: string[]|string) {
    var line = '';
    if (typeof command === 'string') {
        line = command.trim();
    } else {
        line = command.join(' ').trim();
    }

    // Do stuff with line: string
}
```

交差型(Intersection Type)

オブジェクト拡張 (extend)はJavaScriptで非常に一般的なパターンです。ここでは2つのオブジェクトを取得し、これらのオブジェクトの両方の機能を持つ新しいオブジェクトを作成します。**交差型**では、以下に示すようにこのパターンを安全な方法で使用できます。

```
function extend<T, U>(first: T, second: U): T & U {
  return { ...first, ...second };
}

const x = extend({ a: "hello" }, { b: 42 });

// x now has both `a` and `b`
const a = x.a;
const b = x.b;
```

タプル型

JavaScriptには、第一級のタプルのサポートがありません。人々は、一般にタプルとして配列を使用します。これはまさにTypeScriptの型システムがサポートしているものです。タプルは、`: [typeofmember1, typeofmember2]` といったようにアノテーションを付けることができます。タプルには、任意の数のメンバを含めることができます。タプルの例:

```
var nameNumber: [string, number];

// Okay
nameNumber = ['Jenny', 8675309];

// Error!
nameNumber = ['Jenny', '867-5309'];
```

これをTypeScriptの分解(Destructuring)のサポートと組み合わせると、タプルは、その実態は配列であるとはいえ、かなり第一級であるように感じられます：

```
var nameNumber: [string, number];
nameNumber = ['Jenny', 8675309];

var [name, num] = nameNumber;
```

型エイリアス(Type Alias)

TypeScriptは、複数の場所で使用したい型アノテーションの名前を提供するための便利な構文を提供します。エイリアスは `type SomeName = someValidTypeAnnotation` 構文を使用して作成できます。例:

```
type StrOrNum = string|number;

// Usage: just like any other notation
var sample: StrOrNum;
sample = 123;
sample = '123';

// Just checking
sample = true; // Error!
```

`interface` とは違って、型エイリアスは文字通りどんな型アノテーションにも与えることができます(ユニオン型や交差型のようなものに便利です)。構文に慣れ親しむための例をいくつか次に示します。

```
type Text = string | { text: string };
type Coordinates = [number, number];
type Callback = (data: string) => void;
```

> ヒント：型アノテーションの階層を持つ必要がある場合は、 `interface` を使います。インターフェースは `implements` と `extends` で使うことができます
>
> ヒント：型エイリアスは、比較的単純なオブジェクト構造(`Coordinates` (座標)のような)にセマンティックな名前を付けるために使いましょう。また、ユニオン型や交差型にセマンティックな名前を付けたい場合に用いるのもよいでしょう。

まとめ

これで、ほとんどのJavaScriptコードに型アノテーションを付けることができるようになりました。これで、TypeScriptの型システムで使用可能なすべての機能の詳細を説明できます。