Top 10 Des Astuces Rapides Que J'ai Apprises Avec TypeScript

TypeScript offre tout ce que JavaScript fait, plus le typage statique. Outre le système de type de TypeScript, ce qui m'a fait tomber amoureux, c'est qu'il documente votre code. Découvrez ces 10 conseils qui vous aideront à tomber amoureux vous aussi !

En un mot, TypeScript est un langage de programmation qui offre toutes les fonctionnalités JavaScript, mais avec le typage statique activé chaque fois que vous le souhaitez. Le code est compilé en JavaScript brut, et le langage, maintenu par Microsoft, gagne en popularité chaque année, car de plus en plus de frameworks populaires s'en servent (Vue 3, AdonisJS, NestJS…).

Mais, outre le système de type de TypeScript, ce qui m'a fait tomber amoureux de ce langage, c'est qu'il documente votre code. Vous pouvez voir en un coup d'œil de quel type doit être une variable, l'argument d'une fonction ou sa réponse. Cela rend l'expérience du développeur tellement plus agréable. 🧙

Cet article est écrit pour les personnes qui connaissent déjà TypeScript. Je souhaite partager 10 astuces rapides que j'ai apprises tout au long de mon parcours de développeur. 😉

1. Utilisez le type inconnu avant de choisir par défaut Any

Nous savons tous que l'utilisation du **any**mot-clé est en quelque sorte diabolique. 😈

Heureusement, TypeScript 3.0 a introduit un nouveau mot-clé appelé **unknown**.

La principale différence entre **any**et **unknown**est que le **unknown**type ne peut être attribué qu'au **any**type et au **unknown**type lui-même. Ainsi, c'est un type beaucoup moins permissif et un excellent substitut à **any**, car il rappellera constamment à votre éditeur que vous devez le remplacer par quelque chose de plus explicite. En passant de **any**à **unknown**, on n'autorise (presque) rien au lieu de tout autoriser.

Voici un exemple rapide pour illustrer ce que je veux dire :

```
let  unknownValue: unknown;
let  anyValue: any;

let  unknownValue2: unknown = unknownValue; // This is fine
let  anyValue2: any = unknownValue; // This is fine

let  booleanValue: boolean = unknownValue; // Type 'unknown' is not assignable to type
let  booleanValue2: boolean = anyValue; // While this works
```

```
let numberValue: number = unknownValue; // Type 'unknown' is not assignable to type '
let numberValue2: number = anyValue; // While this works

let stringValue: string = unknownValue; // Type 'unknown' is not assignable to type '
let stringValue2: string = anyValue; // While this works

let objectValue: object = unknownValue; // Type 'unknown' is not assignable to type '
let objectValue2: object = anyValue; // While this works

let arrayValue: any[] = unknownValue; // Type 'unknown' is not assignable to type 'ar
let arrayValue2: any[] = anyValue; // While this works

let functionValue: Function = unknownValue; // Type 'unknown' is not assignable to ty
let functionValue2: Function = anyValue; // While this works
```

Une autre chose que j'aime faire dans mes projets est d'activer **noImplicitAny**. Cet indicateur indiquera à TypeScript d'émettre une erreur chaque fois que quelque chose a le type **any**(défini par vous ou déduit par TypeScript). **Plus de détails dans la documentation officielle** .

2. Le type Jamais peut être pratique pour la gestion des erreurs

Il existe également un autre type que nous n'utilisons jamais lorsque nous commençons à jouer avec TypeScript : le mot- **never**clé. En un mot, il représente le type de valeurs qui ne se produisent jamais.

Il y a quelque temps, j'ai trouvé que ce mot-clé peut être pratique lors de l'écriture de fonctions qui déclenchent une ou plusieurs erreurs et ne retournent jamais rien.

```
function throwErrors(statusCode: number): never {
if (statusCode >= 400 && statusCode <= 499) {
throw Error("Request Error");
}

throw Error("Something wrong happened.");
}
```

Nous pouvons également utiliser ce type à notre avantage en nous assurant que chaque situation critique est gérée dans nos fonctions. Voici un exemple rapide de cas où nous avons oublié de tester le cas pour l'espèce humaine.

```
interface Dwarf {
weapon: "axe";
}

interface Elf {
weapon: "bow";
}

interface Man {
weapon: "sword";
}

type Specy = Dwarf | Elf | Man;

function whatIsThatGandalf(specy: Specy) {
let _ensureAllCasesAreHandled: never;

if (specy.weapon === "axe") {
return "This is a dwarf";
} else if (specy.weapon === "bow") {
return "This is an elf";
}

// ERROR: Type 'Man' is not assignable to type 'never'
_ensureAllCasesAreHandled = specy;
}
```

3. Interfaces vs Types

Nous entendons souvent cette question de la part de personnes qui viennent de commencer leur parcours TypeScript : quelle est la différence entre une interface et un type ? Dois-je utiliser les deux ? 🤔

Il m'a fallu quelques semaines, au début, pour me fixer une règle car on peut souvent faire la même chose avec les deux. Voici comment je résumerais : **Quand je travaille avec des classes ou des objets, j'utilise une interface. Quand je ne le suis pas, j'utilise un type.** Cela aiderait si vous vous souveniez que la chose la plus importante est d'être cohérent avec vos choix dans votre base de code.

Bien sûr, il existe aussi quelques différences subtiles entre les deux, **comme expliqué dans cette superbe vidéo** . Par exemple, une interface peut étendre d'autres interfaces, alors que vous avez

besoin d'une union ou d'une intersection pour fusionner deux types ensemble (car ils sont statiques).

De plus, d'après mon expérience, lorsque je traite des types, le message d'erreur peut généralement être plus brutal pour comprendre quand quelque chose ne va pas.

> ***Une autre chose à garder à l'esprit est que la documentation TypeScript vous encourage à utiliser une interface lorsque cela est possible, surtout si vous écrivez une bibliothèque qui exporte un type. La raison en est qu'une interface peut être étendue pour répondre aux besoins de l'application qui utilise votre code.***

4. Apprenez à utiliser les types génériques

Les types génériques sont pratiques. Plus vous vous familiarisez avec TypeScript, plus vous les utilisez. Ils permettent à votre code d'être plus flexible en vous permettant de définir vous-même le type. Un exemple peindra mille mots. 😃

Disons que nous voulons geler ou transformer toutes les propriétés d'un objet en propriétés en lecture seule. Eh bien… nous pourrions faire quelque chose comme ça.

```
interface  Elf {
name: string;
weapon: "bow";
}

const  myElf: Elf = {
name:  "Legolas",
weapon:  "bow",
};

const  freezedElf = Object.freeze(myElf);

// ERROR: Cannot assign to 'name' because it is a read-only property.
freezedElf.name = "Galadriel";
```

Maintenant, utilisons un type générique pour faire la même chose.

```
interface  Elf {
name: string;
weapon: "bow";
```

```
}

const  myElf: Elf = {
name:  "Legolas",
weapon:  "bow",
};

type  Freeze<T> = {
readonly [P  in  keyof  T]: T[P];
};

const  freezedElf: Freeze<Elf> = myElf;

// ERROR: Cannot assign to 'name' because it is a read-only property.
freezedElf.name = "Galadriel";

// For your information, the generic type Freeze already exists in TypeScript and is c

// https://www.typescriptlang.org/docs/handbook/utility-types.html

// So both are equivalent
const  freezedElf: Freeze<Elf> = myElf;
const  freezedElf: Readonly<Elf> = myElf;
```

> ***Si vous vous posez la question, l' keyofopérateur prend un type d'objet et produit une chaîne ou une union numérique littérale de clés de liste.***

Comme vous pouvez le voir dans l'exemple ci-dessus, ce qui est remarquable, c'est que peu importe la forme de l'objet, nous pouvons créer un autre type (**Freeze<Elf>**) qui inclura toutes les propriétés définies en lecture seule.

Voici un autre type générique qui définira toutes les propriétés de l'objet comme non en lecture seule.

```
type  Writable<T> = {
-readonly [P  in  keyof  T]: T[P];
};
```

Un dernier exemple ici est une fonction utilisant un type générique qui renvoie le dernier élément d'un tableau.

```
const  getLastElement = <T>(array: T[]) => {
return  array[array.length - 1];
};
```

5. Le type d'utilitaire partiel pourrait être votre nouveau meilleur ami

Que se passe-t-il si nous souhaitons créer un nouveau type basé sur un autre type mais avec toutes ses propriétés définies sur facultatives. Comment pourrions-nous faire cela ? 🤔

TypeScript est livré avec un utilitaire que j'utilise chaque semaine appelé **Partial<Type>**. Voici comment cela fonctionne.

```
interface  Elf {
name: string;
weapon: "bow";
lifepoints: number;
}

type  PartialElf = Partial<Elf>;

// We can omit the weapon attribute as all properties are now optional

const  partialElf: PartialElf = {
name:  "Legolas",
lifepoints:  100,
};

// This is how it is coded behind the curtain
type  Partial<T> = {
[P  in  keyof  T]?: T[P];
};
```

Génial, non ? Plongeons maintenant dans d'autres types d'utilitaires que vous aimerez utiliser dans votre projet. 😃

6. Autres types d'utilitaires à connaître

Vous avez appris **Partial<Type>**à construire un type avec toutes les propriétés de **Type**la valeur facultative. Mais il y en a plus. Voici ceux que j'utilise souvent :

Required<Type>: construit un type composé de toutes les propriétés de **Type**set to required. Comme vous pouvez le deviner, c'est l'opposé de Partial.

```
interface  Properties {
a?: number;
b?: string;
}

const  object: Properties = { a:  5 };

// ERROR: Property 'b' is missing in type '{ a: number; }' but required in type 'Requi
const  object2: Required<Properties> = { a:  5 };
```

Readonly<Type>: construit un type avec toutes les propriétés de **Type**set en lecture seule, ce qui signifie que les propriétés du type construit ne peuvent pas être réaffectées.

```
interface  Todo {
title: string;
}

const  todo: Readonly<Todo> = {
title:  "Learn Kendo UI",
};

// ERROR: Cannot assign to 'title' because it is a read-only property.
todo.title = "Hello";
```

Pick<Type, Keys>: construit un type en sélectionnant l'ensemble de propriétés **Keys**(littéral de chaîne ou union de littéraux de chaîne) dans **Type**.

```
interface  Todo {
title: string;
description: string;
completed: boolean;
}
```

```
type  TodoPreview = Pick<Todo, "title" | "completed">;

const  todo: TodoPreview = {
title:  "Learn DevCraft",
completed:  false,
};
```

Omit<Type, Keys>: construit un type en sélectionnant toutes les propriétés **Type**puis en les supprimant **Keys**(littéral de chaîne ou union de littéraux de chaîne).

```
interface  Todo {
title: string;
description: string;
completed: boolean;
createdAt: number;
}

type  TodoPreview = Omit<Todo, "description">;

const  todo: TodoPreview = {
title:  "Learn Kendo UI",
completed:  true,
createdAt:  1615277055442,
};
```

NonNullable<Type>: Construit un type en excluant **null**et **undefined**à partir de **Type**.

```
// T will be equivalent to string | number
type  T = NonNullable<string | number | undefined | null>;
```

Pour parcourir la liste complète des types d'utilitaires disponibles dans le monde, rendez **-vous sur la documentation officielle** .

7. Utilisez des gardes de type pour accéder à une propriété en toute sécurité

Le code à l'épreuve des balles utilise beaucoup les gardes de type. TypeScript permet de savoir plus facilement quand nous devons en utiliser un. 😍

Pour résumer, les gardes de type permettent de vérifier si un objet appartient au bon type. C'est une protection que nous utilisons dans notre code pour nous assurer que rien de mal ne se

produit. Ce qui est excellent avec TypeScript, c'est que, avec les erreurs affichées directement dans l'éditeur, on peut deviner quand ajouter des gardes de type.

Voici quelques protections de type standard que vous pouvez utiliser.

typeof: L' **typeof**opérateur renvoie une chaîne indiquant le type de l'opérande non évalué.

```
function stringOrNumber(x: number | string) {
if (typeof x === "string") {
return "I am a string";
}

return "I am a number";
}
```

instanceof: L' **instanceof**opérateur teste pour voir si la propriété prototype d'un constructeur apparaît n'importe où dans la chaîne de prototypes d'un objet. La valeur de retour est une valeur booléenne.

```
class Dwarf {
weapon = "axe";
}

class Elf {
weapon = "bow";
}

function dwarfOrElf(specy: Dwarf | Elf) {
if (specy instanceof Dwarf) {
return "I am a dwarf";
}

return "I am an elf";
}
```

in: L' **in**opérateur renvoie true si la propriété spécifiée se trouve dans l'objet spécifié ou sa chaîne de prototypes.

```
interface Dwarf {
weapon: "axe";
lifepoints: number;
}

interface Elf {
weapon: "bow";
}

function dwarfOrElf(specy: Dwarf | Elf) {
if ("lifepoints" in specy) {
return "I am a dwarf";
}

return "I am an elf";
}
```

Protections de type définies par l'utilisateur.

```
interface Dwarf {
weapon: "axe";
lifepoints: number;
}

interface Elf {
weapon: "bow";
}

function isDwarf(specy: any): specy is Dwarf {
return specy.lifepoints !== undefined;
}

function dwarfOrElf(specy: Dwarf | Elf) {
if (isDwarf(specy)) {
return "I am a dwarf";
}

return "I am an elf";
}
```

8. Les décorateurs personnalisés peuvent rendre votre code plus facile à lire (l'exemple de la fonction de synchronisation)

J'aime quand j'utilise un cadre ou une bibliothèque qui fait bon usage des décorateurs. Alors que certaines personnes pensent qu'elles sont un anti-modèle et qu'elles ne devraient pas être utilisées, la réalité est un peu plus complexe. Ils peuvent souvent rendre le code plus facile à lire et plus rapide à écrire. Les frameworks comme AdonisJS, NestJS ou encore Angular font beaucoup appel aux décorateurs.

Les décorateurs peuvent être appliqués aux définitions de classe, aux propriétés, aux méthodes, aux accesseurs et aux paramètres. Ils représentent des fonctions qui modifieront le comportement de votre code.

Ils sont faciles à écrire. Voici comment nous pouvons créer un décorateur qui calculera le temps d'exécution d'une fonction.

```
function  time(name: string) {
return  function (target, propertyKey: string, descriptor: PropertyDescriptor) {
const  fn = descriptor.value;

descriptor.value = (...args) => {
console.time(name);
const  v = fn(...args);
console.timeEnd(name);
return  v;
};
};
}

class  Specy {
@time("attack")
attack() {
// ...
}
}
```

> ***Vous pouvez avoir un avertissement dans votre éditeur car la prise en charge des décorateurs expérimentaux est une fonctionnalité susceptible de changer dans une future version. Définissez l'option "experimentalDecorators" dans votre tsconfigou jsconfigpour supprimer cet avertissement.***

Si vous voulez en savoir plus sur les décorateurs, vous devriez regarder **cette superbe vidéo de Fireship** (qui est aussi une excellente chaîne de codage à laquelle vous devriez vous abonner 😃).

> ***Un mot d'avertissement : les classes héritées recevront les fonctionnalités du décorateur.***

9. Utilisez strictNullChecks et noUncheckedIndexAccess

Il y a deux drapeaux que j'active habituellement dans mon fichier de configuration TypeScript : **strictNullChecks**et **noUncheckedIndexAccess**.

> **strictNullChecks**: Par défaut, **null**et **undefined**sont assignables à tous les types dans TypeScript. Avec ce drapeau activé, ils ne le seront pas.

```
let  foo = undefined;

foo = null; // Will trigger an error

let  foo2: number = 123;

foo2 = null; // Will trigger an error
foo2 = undefined; // Will trigger an error
```

> **noUncheckedIndexAccess**: TypeScript possède une fonctionnalité appelée signatures d'index. Ces signatures sont un moyen de signaler au système de type que les utilisateurs peuvent accéder à des propriétés nommées arbitrairement. Avec ce drapeau activé, ils ne pourront pas le faire.

```
const  nums = [0, 1, 2];
const  example: number = nums[4]; // WIll trigger an error
```

10. Utilisez noImplicitOverride (en particulier pour les fonctions fictives)

C'est probablement le conseil le plus utile de cette liste pour les personnes qui écrivent beaucoup de tests.

je vais m'expliquer. Lorsque nous créons des fonctions et des classes moqueuses pour vérifier que notre code se comporte correctement, nous ne voulons pas que les gens renomment une fonction à l'intérieur de cette classe initiale sans être avertis qu'ils doivent également la modifier à l'intérieur de la classe moqueuse. Eh bien… TypeScript vous aidera avec cela.

La première chose à faire pour résoudre ce problème est d'activer **noImplicitOverride**dans votre fichier de configuration TypeScript et d'utiliser le mot- **override**clé.

```
class  Specy {
lifepoints = 10;

heal(lifepoints: number) {
this.lifepoints += lifepoints || 10;
}
}

class  MockSpecy  extends  Specy {
override  heal(lifepoints) {
console.log("Let's override the heal method");
}
}
```

Si quelqu'un modifie la **heal**fonction dans **healing**la **Specy**classe, TypeScript affichera une erreur dans la **MockSpecy**classe pour nous dire que nous devons également mettre à jour la méthode ici.

Emballer

Tu l'as fait! Voici dix conseils rapides sur TypeScript que je voulais partager avec vous. 😇

Un dernier conseil est que si vous souhaitez approfondir vos connaissances sur ce magnifique langage, vous devez choisir un framework basé sur celui-ci et plonger dans sa base de code, quelque chose comme Nest, Adonis ou Vue 3. Plus vous lisez le code TS, plus plus vous découvrirez de grandes choses que vous pouvez faire avec.

Lien : https://www.telerik.com/blogs/10-quick-tips-learned-using-typescript

#typescript #javascript