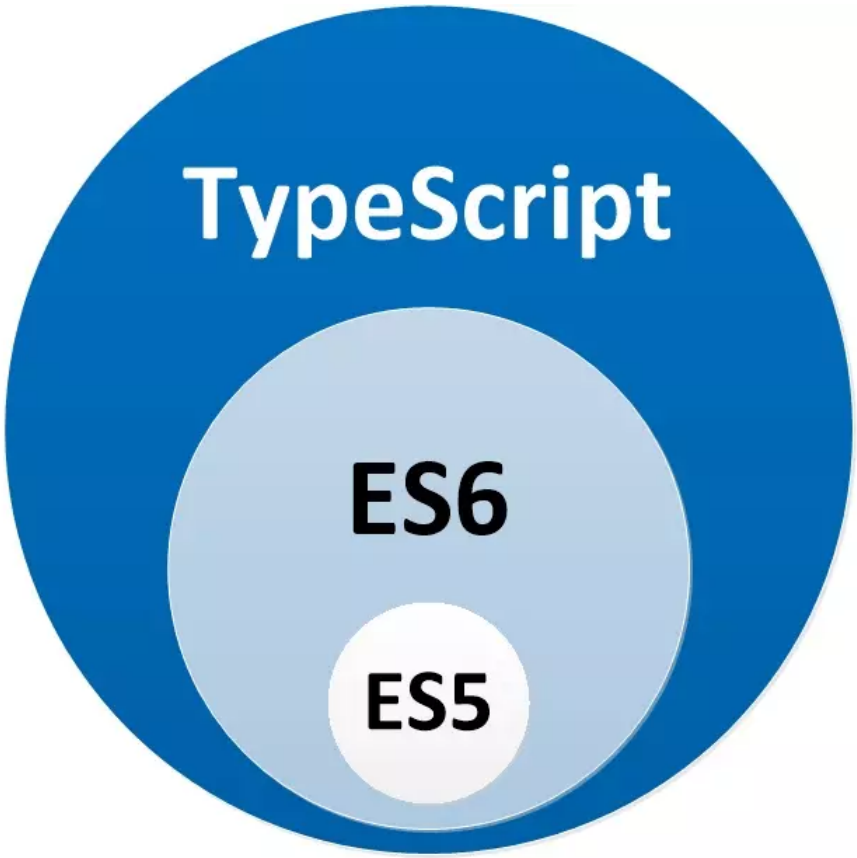
TS1

到底为什么要学习 TypeScript?

TypeScript 在推出之初就既受追捧又受质疑，在社区和各种论坛中总有一些这样的声音：

- 静态语言会丧失 JavaScript 的灵活性
- 静态类型不是银弹，大型项目依然可以用 JavaScript 编写
- TypeScript 必定赴 coffeescript 后尘，会被标准取代

JavaScript 的超集

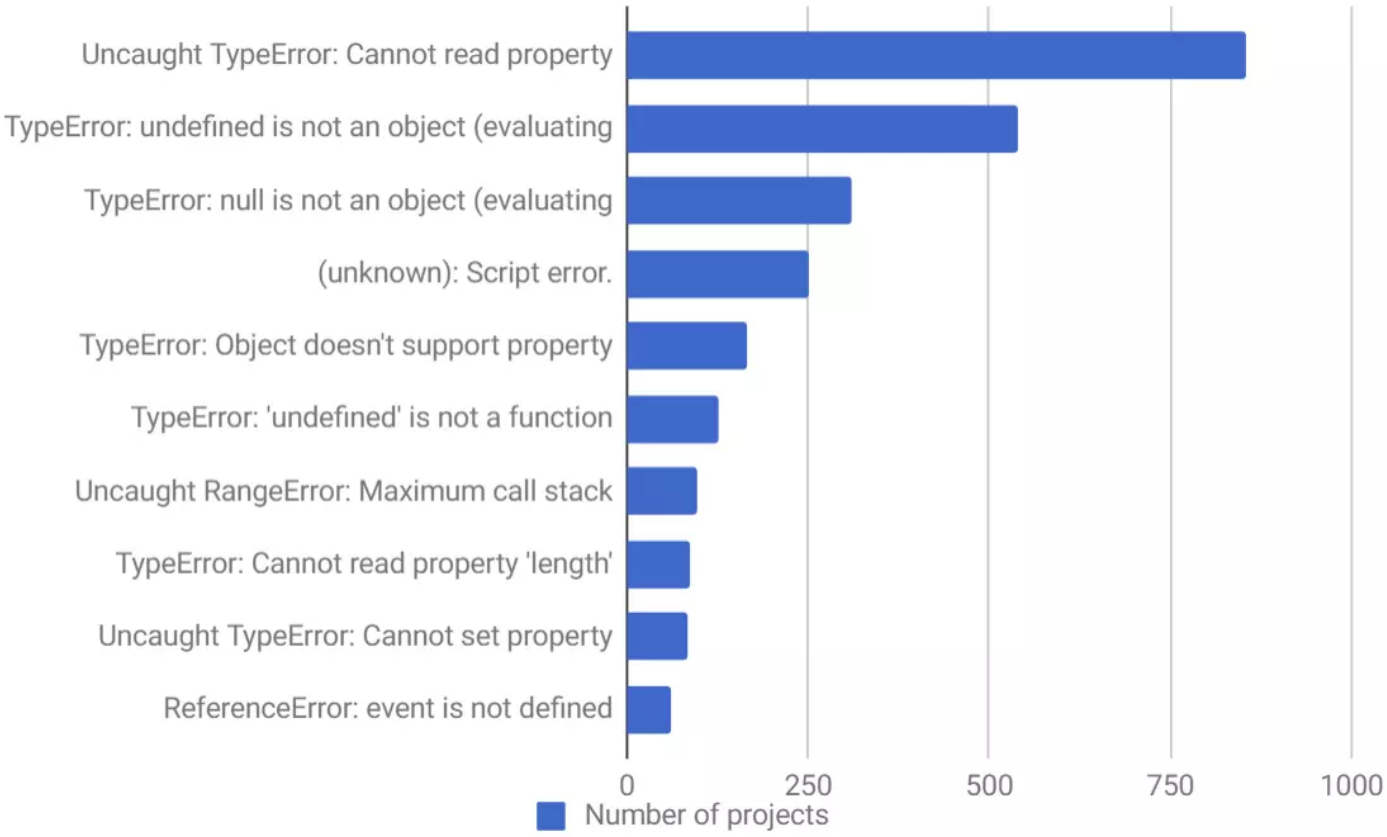


- 在 TypeScript 可以使用一些尚在提案阶段的语法特性，可以有控制访问符，而最主要的区别就是 TypeScript 是一门静态语言。
- 这也是为什么 「TypeScript 必定赴 coffeescript 后尘，会被标准取代」 这个论断几乎不可能成立的原因之一，coffeescript 本质上是 JavaScript 的语法糖化，ES2015 参考了大量 coffeescript 的内容进行了标准化，因此 coffeescript 的优势也就不存在了，被淘汰在所难免。
- 给一门语言加语法糖是相对容易推进到标准的事情，而直接把一门语言从动态改为静态，还要兼容数以亿计的老旧网站，这个在可预见的时间内几乎不可能发生，TypeScript 与 coffeescript 虽然都

是「Compile to JavaScript Language」，但是 TypeScript 的静态性是它立于不败之地的基础

静态类型

简单来说，一门语言在编译时报错，那么是静态语言，如果在运行时报错，那么是动态语言。
JavaScript 项目中最常见的十大错误。



很多项目，尤其是中大型项目，我们是需要团队多人协作的，那么如何保证协作呢？这个时候可能需要大量的文档和注释，显式类型就是最好的注释，而通过 TypeScript 提供的类型提示功能我们可以非常舒服地调用同伴的代码，由于 TypeScript 的存在我们可以节省大量沟通成本、代码阅读成本等等。

严谨不失灵活

很多人以为用了 TypeScript 之后就会丧失 JavaScript 的灵活性，其实并不是。
首先，我们得承认 JavaScript 的灵活性对于中大型项目弊远远大于利，其次，TypeScript 由于兼容 JavaScript 所以其灵活度可以媲美 JavaScript，比如你可以把任何想灵活的地方将类型定义为 any 即可，把 TypeScript 变为 AnyScript 就能保持它的灵活度，毕竟TypeScript 对类型的检查严格程度是可以通过 `tsconfig.json` 来配置的。
即使在开启 `strict` 状态下的 TypeScript 依然是很灵活的，因为为了兼容 JavaScript，TypeScript 采用了Structural Type System。
因此，TypeScript 并不是类型定义本身，而是类型定义的形状（Shape），我们看个例子：

```
class Foo {
  method(input: string): number { ... }
}
class Bar {
  method(input: string): number { ... }
}
const foo: Bar = new Foo(); // Okay.const bar: Bar = new Foo(); // Okay.
```

以上代码是不会报错的，因为他们的「形状」是一样的，而类似的代码在 Java 或者 C# 中是会报错的。
这就是 TypeScript 类型系统设计之初就考虑到了 JavaScript 灵活性，专门选择了 Structural Type System（结构类型系统）。

TypeScript 没有缺点吗?

- 与实际框架结合会有很多坑
- 配置学习成本高
- TypeScript 的类型系统其实比较复杂

小结

TypeScript 的优势所在，总结下来有三点:
1. 规避大量低级错误，避免时间浪费，省时
2. 减少多人协作项目的成本，大型项目友好，省力
3. 良好代码提示，不用反复文件跳转或者翻文档，省心

基础类型

- 布尔类型：`boolean`
- 数字类型：`number`
- 字符串类型：`string`
- 空值：`void`
- Null 和 Undefined：`null` 和 `undefined`
- Symbol 类型：`symbol`
- BigInt 大数整数类型：`bigint`

这里需要提示一下，很多 TypeScript 的原始类型比如 boolean、number、string等等，在 JavaScript中都有类似的关键字 Boolean、Number、String，后者是 JavaScript 的构造函数，比如

我们用 Number 用于数字类型转化或者构造 Number 对象用的，而 TypeScript 中的 number 类型仅仅是表示类型，两者完全不同。
其他：

- `any`类型是多人协作项目的大忌，很可能把Typescript变成AnyScript，通常在不得已的情况下，不应该首先考虑使用此类型。
- `unknown` 是 TypeScript 3.0 引入了新类型,是 `any` 类型对应的安全类型。

`unknown` 与 `any` 的不同之处,虽然它们都可以是任何类型,但是当 `unknown` 类型被确定是某个类型之前,它不能被进行任何操作比如实例化、getter、函数执行等等。

- never 类型表示的是那些永不存在的值的类型，never 类型是任何类型的子类型，也可以赋值给任何类型；然而，没有类型是 never 的子类型或可以赋值给 never 类型（除了never本身之外）。

```
// 抛出异常的函数永远不会有返回值
function error(message: string): never {
    throw new Error(message);
}
// 空数组，而且永远是空的
const empty: never[] = []
```

- 数组

有两种类型定义方式，一种是使用泛型:

```
const list: Array<number> = [1, 2, 3]
```

另一种使用更加广泛那就是在元素类型后面接上 `[]`:

```
const list: number[] = [1, 2, 3]
```

- 元组（Tuple）

元组类型与数组类型非常相似，表示一个已知元素数量和类型的数组，各元素的类型不必相同。

```
let x: [string, number];
x = ['hello', 10, false] // Error
x = ['hello'] // Error
```

元组非常严格，即使类型的顺序不一样也会报错。

```
let x: [string, number];
x = ['hello', 10]; // OK
x = [10, 'hello']; // Error
```

我们可以把元组看成严格版的数组，比如`[string, number]`我们可以看成是:

```
interface Tuple extends Array<string | number> {
  0: string;
```

```
  1: number;
  length: 2;
}
```

元组越界问题

- object 表示非原始类型，也就是除 number，string，boolean，symbol，null 或 undefined 之外的类型。

```
// 这是下一节会提到的枚举类型
enum Direction {
    Center = 1
}
let value: object
value = Direction
value = [1]
value = [1, 'hello']
value = {}
```

普通对象、枚举、数组、元组通通都是 `object` 类型。

深入理解枚举类型

枚举类型是很多语言都拥有的类型,它用于声明一组命名的常数,当一个变量有几种可能的取值时,可以将它定义为枚举类型。

数字枚举

当我们声明一个枚举类型是,虽然没有给它们赋值,但是它们的值其实是默认的数字类型,而且默认从0开始依次累加:

```
enum Direction {
    Up,
    Down,
    Left,
    Right
}

console.log(Direction.Up === 0); // true
console.log(Direction.Down === 1); // true
```

```
10 console.log(Direction.Left === 2); // true
11 console.log(Direction.Right === 3); // true
```

因此当我们把第一个值赋值后,后面也会根据第一个值进行累加:

```
1 enum Direction {
2     Up = 10,
3     Down,
4     Left,
5     Right
6 }
7
8 console.log(Direction.Up, Direction.Down, Direction.Left, Direction.Right); // 10 11 12 13
```

字符串枚举

```
1 enum Direction {
2     Up = 'Up',
3     Down = 'Down',
4     Left = 'Left',
5     Right = 'Right'
6 }
7
8 console.log(Direction['Right'], Direction.Up); // Right Up
```

异构枚举

```
1 enum BooleanLikeHeterogeneousEnum {
2     No = 0,
3     Yes = "YES",
4 }
```

通常情况下我们很少会这样使用枚举，但是从技术的角度来说，它是可行的。

反向映射

我们可以通过枚举名字获取枚举值，这当然看起来没问题，那么能不能通过枚举值获取枚举名字呢?

```
1 enum Direction {
```

```
    Up,
    Down,
    Left,
    Right
}

console.log(Direction[0]); // Up
```

枚举的本质

以上面的 `Direction` 枚举类型为例,我们不妨看一下枚举类型被编译为 JavaScript 后是什么样子:

```
var Direction;
(function (Direction) {
    Direction[Direction["Up"] = 10] = "Up";
    Direction[Direction["Down"] = 11] = "Down";
    Direction[Direction["Left"] = 12] = "Left";
    Direction[Direction["Right"] = 13] = "Right";
})(Direction || (Direction = {}));
```

常量枚举

性能提升

联合枚举与枚举成员的类型

当所有枚举成员都拥有字面量枚举值时，它就带有了一种特殊的语义，即枚举成员成为了类型。

```
enum Direction {
    Up,
    Down,
    Left,
    Right
}
const a = 0
console.log(a === Direction.Up) // true
```

我们把成员当做值使用，看来是没问题的，因为成员值本身就是0，那么我们再加几行代码：

```
type c = 0
```

```
2
3 declare let b: c
4
5 b = 1 // 不能将类型“1”分配给类型“0”
6 b = Direction.Up // ok
```

联合枚举类型

```
1  enum Direction {
2      Up,
3      Down,
4      Left,
5      Right
6  }
7  declare let a: Direction
8
9  enum Animal {
10     Dog,
11     Cat
12 }
13 a = Direction.Up // ok
14 a = Animal.Dog // 不能将类型“Animal.Dog”分配给类型“Direction”
```

我们把 `a` 声明为 `Direction` 类型，可以看成我们声明了一个联合类型 `Direction.Up | Direction.Down | Direction.Left | Direction.Right`，只有这四个类型其中的成员才符合要求。

枚举合并

为枚举添加静态方法

接口(interface)

TypeScript 的核心原则之一是对值所具有的结构进行类型检查,它有时被称做“鸭式辨型法”或“结构性子类型化”。
在TypeScript里，接口的作用就是为这些类型命名和为你的代码或第三方代码定义契约。

接口的使用

可选属性、只读属性

```
interface User {
    name: string
    age?: number
    readonly isMale: boolean
}
```

函数类型

如果这个 `user` 含有一个函数怎么办?

```
interface User {
    name: string
    age?: number
    readonly isMale: boolean
    say: (words: string) => string
}
```

属性检查

可索引类型

继承接口

```
interface VIPUser extends User {
    broadcast: () => void
}
interface VIPUser extends User, SupperUser {
    broadcast: () => void
}
```

类(Class)

传统的面向对象语言基本都是基于类的，JavaScript 基于原型的方式让开发者多了很多理解成本，在 ES6 之后，JavaScript 拥有了 class 关键字，虽然本质依然是构造函数，但是开发者已经可以比较舒

服地使用 class了。
但是 JavaScript 的 class 依然有一些特性还没有加入，比如修饰符和抽象类等。
之于一些继承、静态属性这些在 JavaScript 本来就存在的特性，我们就不过多讨论了。

抽象类

抽象类做为其它派生类的基类使用,它们一般不会直接被实例化,不同于接口,抽象类可以包含成员的实现细节。
abstract 关键字是用于定义抽象类和在抽象类内部定义抽象方法。
比如我们创建一个 `Animal` 抽象类:

```
abstract class Animal {
    abstract makeSound(): void;
    move(): void {
        console.log('roaming the earch...');
    }
}
```

我在实例化此抽象类会报错
我们不能直接实例化抽象类，通常需要我们创建子类继承基类,然后可以实例化子类。

```
class Cat extends Animal {
    makeSound() {
        console.log('miao miao')
    }
}
const cat = new Cat()
cat.makeSound() // miao miao
cat.move() // roaming the earch...
```

访问限定符

public

在 TypeScript 的类中，成员都默认为 public, 被此限定符修饰的成员是可以被外部访问。

private

当成员被设置为 private 之后, 被此限定符修饰的成员是只可以被类的内部访问。

protected

当成员被设置为 protected 之后, 被此限定符修饰的成员是只可以被类的内部以及类的子类访问。

class 可以作为接口

上一节我们讲到接口（interface），实际上类（class）也可以作为接口。
而把 class 作为 interface 使用，在 React 工程中是很常用的。
由于组件需要传入 `props` 的类型 `Props` ，同时有需要设置默认 `props` 即 `defaultProps`。
这个时候 class 作为接口的优势就体现出来了。
我们先声明一个类，这个类包含组件 `props` 所需的类型和初始值：

```
// props的类型
export default class Props {
  public children: Array<React.ReactElement<any>> | React.ReactElement<any> | never[] = []
  public speed: number = 500
  public height: number = 160
  public animation: string = 'easeInOutQuad'
  public isAuto: boolean = true
  public autoPlayInterval: number = 4500
  public afterChange: () => {}
  public beforeChange: () => {}
  public selesctedColor: string
  public showDots: boolean = true
}
```

当我们需要传入 `props` 类型的时候直接将 `Props` 作为接口传入，此时 `Props` 的作用就是接口，而当需要我们设置`defaultProps`初始值的时候，我们只需要:

```
public static defaultProps = new Props()
```

`Props` 的实例就是 `defaultProps` 的初始值，这就是 class 作为接口的实际应用，我们用一个 class 起到了接口和设置初始值两个作用，方便统一管理，减少了代码量。

函数(Function)

函数是 JavaScript 应用程序的基础,它帮助你实现抽象层、模拟类、信息隐藏和模块。
在 TypeScript 里,虽然已经支持类、命名空间和模块,但函数仍然是主要的定义行为的地方,TypeScript 为 JavaScript 函数添加了额外的功能，让我们可以更容易地使用。

定义函数类型

```
const add = (a: number, b: number) => a + b
```

函数的参数详解

可选参数

一个函数的参数可能是不存在的，这就需要我们使\用可选参数来定义.
我们只需要在参数后面加上 `?` 即代表参数可能不存在。

```
const add = (a: number, b?: number) => a + (b ? b : 0)
```

参数b有`number`与`undefined`两种可能。

默认参数

```
const add = (a: number, b = 10) => a + b
```

剩余参数

```
const add = (a: number, ...rest: number[]) => rest.reduce(((a, b) => a + b), a)
```

重载（Overload）

泛型（generic）

```
function returnItem<T>(para: T): T {
    return para
}
```

多个类型参数

```
function swap<T, U>(tuple: [T, U]): [U, T] {
    return [tuple[1], tuple[0]];
}

swap([7, 'seven']); // ['seven', 7]
```

泛型变量

```
function getArrayLength<T>(arg: Array<T>) {
```

```
  console.log(arg.length) // ok
  return arg
}
```

泛型接口

```
interface ReturnItemFn<K> {
    (para: K): K
}
const returnItem: ReturnItemFn<number> = para => para
```

泛型类

```
class Stack<T> {
    private arr: T[] = []

    public push(item: T) {
        this.arr.push(item)
    }

    public pop() {
        this.arr.pop()
    }
}
```

泛型约束

我们先看一个常见的需求，我们要设计一个函数，这个函数接受两个参数，一个参数为对象，另一个参数为对象上的属性，我们通过这两个参数返回这个属性的值

```
function getValue<T extends object, U extends keyof T>(obj: T, key: U) {
  return obj[key] // ok
}
```

使用多重类型进行泛型约束

泛型与 new

```
function factory<T>(type: {new(): T}): T {
```

```
  return new type() // ok
}
//new Construct
this = Object.creat(Construct.prototype)
return  typeof Construct.call(this) ==='object'? Construct.call(this):this
```

装饰器

目前装饰器本质上是一个函数,`@expression` 的形式其实是一个语法糖, expression 求值后必须也是一个函数，它会在运行时被调用，被装饰的声明信息做为参数传入.

类装饰器

属性/方法装饰器

参数装饰器

装饰器工厂

装饰器顺序

Reflect Metadata