

邂逅TypeScript语法

王红元 coderwhy

目录

content



1 JavaScript类型缺失

2 邂逅TypeScript开发

3 TypeScript运行环境

4 TypeScript变量声明

5 JavaScript数据类型

6 TypeScript数据类型

JavaScript一门优秀的语言

- 我始终相信：任何新技术的出现都是为了解决原有技术的某个痛点。
- JavaScript是一门优秀的编程语言吗？
 - 每个人可能观点并不完全一致，但是从很多角度来看，JavaScript是一门**非常优秀的编程语言**；
 - 而且，可以说在很长一段时间内**这个语言不会被代替**，并且会在**更多的领域**被大家广泛使用；
- 著名的Atwood定律：
 - Stack Overflow的创立者之一的 **Jeff Atwood** 在2007年提出了著名的 **Atwood定律**。
 - any application that can be written in JavaScript, will eventually be written in JavaScript.
 - 任何可以使用JavaScript来实现的应用都最终都会使用JavaScript实现。
- 其实我们已经看到了，这句话正在一步步被应验：
 - **Web端**的开发我们一直都是使用JavaScript；
 - **移动端**开发可以借助于ReactNative、Weex、Uniapp等框架实现跨平台开发；
 - **小程序端**的开发也是离不开JavaScript；
 - **桌面端**应用程序我们可以借助于Electron来开发；
 - **服务器端**开发可以借助于Node环境使用JavaScript来开发。

JavaScript的痛点

- 并且随着近几年前端领域的快速发展，让JavaScript迅速被普及和受广大开发者的喜爱，借助于JavaScript本身的强大，也让使用JavaScript开发的人员越来越多。
- 优秀的JavaScript没有缺点吗？
 - 其实上由于各种历史因素，JavaScript语言本身存在很多的缺点；
 - 比如ES5以及之前的使用的var关键字关于作用域的问题；
 - 比如最初JavaScript设计的数组类型并不是连续的内存空间；
 - 比如直到今天JavaScript也没有加入类型检测这一机制；
- JavaScript正在慢慢变好
 - 不可否认的是，JavaScript正在慢慢变得越来越好，无论是从底层设计还是应用层面。
 - ES6、7、8等的推出，每次都会让这门语言更加现代、更加安全、更加方便。
 - 但是知道今天，JavaScript在类型检测上依然是毫无进展（为什么类型检测如此重要，我后面会聊到）。

类型带来的问题

■ 首先你需要知道，编程开发中我们有一个共识：错误出现的越早越好

- 能在**写代码的时候**发现错误，就不要在**代码编译时**再发现（IDE的优势就是在代码编写过程中帮助我们发现错误）。
- 能在**代码编译期间**发现错误，就不要在**代码运行期间**再发现（类型检测就可以很好的帮助我们做到这一点）。
- 能在开发阶段发现错误，就不要在测试期间发现错误，能在测试期间发现错误，就不要在上线后发现错误。

■ 现在我们想探究的就是如何在 **代码编译期间** 发现代码的错误：

- JavaScript可以做到吗？不可以，我们来看下面这段经常可能出现的代码问题。

```
function getLength(str) {  
  return str.length;  
}  
  
getLength("abc"); // 正确的调用  
getLength(); // 错误的调用(IDE并不会报错)
```

```
TypeError: Cannot read property 'length' of undefined  
    at getLength (/Users/coderwhy/Desktop/Vue3+TS/备课/co  
    at Object.<anonymous> (/Users/coderwhy/Desktop/Vue3+T
```

■ 这是我们一个非常常见的错误：

- 这个错误很大的原因就是JavaScript没有对我们传入的参数进行任何的限制，只能等到运行期间才发现这个错误；
- 并且当这个错误产生时，会影响后续代码的继续执行，也就是整个项目都因为一个小小的错误而深入崩溃；

■ 当然，你可能会想：我怎么可能犯这样低级的错误呢？

- 当我们写像我们上面这样的简单的demo时，这样的错误很容易避免，并且当出现错误时，也很容易检查出来；
- 但是当我们开发一个大型项目时呢？你能保证自己一定不会出现这样的问题吗？而且如果我们是调用别人的类库，又如何知道让我们传入的到底是什么样的参数呢？

■ 但是，如果我们可以给JavaScript加上很多限制，在开发中就可以很好的避免这样的问题了：

- 比如我们的getLength函数中str是一个必传的类型，没有调用者没有传编译期间就会报错；
- 比如我们要求它的必须是一个String类型，传入其他类型就直接报错；
- 那么就可以知道很多的错误问题在编译期间就被发现，而不是等到运行时再去发现和修改；

类型思维的缺失

- 我们已经简单体会到没有类型检查带来的一些问题，JavaScript因为从设计之初就没有考虑类型的约束问题，所以造成了前端开发人员关于**类型思维的缺失**：
 - 前端开发人员通常不关心变量或者参数是什么类型的，如果在必须确定类型时，我们往往需要使用各种判断验证；
 - 从其他方向转到前端的人员，也会因为没有类型约束，而总是担心自己的**代码不安全，不够健壮**；
- 所以我们经常会说JavaScript**不适合开发大型项目**，因为当项目一旦庞大起来，这种宽松的类型约束会带来非常多的安全隐患，多人员开发它们之间也**没有良好的类型契约**。
 - 比如当我们去实现一个核心类库时，如果没有类型约束，那么需要对**别人传入的参数进行各种验证**来保证我们代码的健壮性；
 - 比如我们去调用别人的函数，对方没有对函数进行任何的注释，我们只能去看里面的逻辑来理解这个函数需要**传入什么参数，返回值是什么类型**；

JavaScript添加类型约束

■ 为了弥补JavaScript类型约束上的缺陷，增加类型约束，很多公司推出了自己的方案：

- 2014年，Facebook推出了flow来对JavaScript进行类型检查；
- 同年，Microsoft微软也推出了TypeScript1.0版本；
- 他们都致力于为JavaScript提供类型检查；

■ 而现在，无疑TypeScript已经完全胜出：

- Vue2.x的时候采用的就是flow来做类型检查；
- Vue3.x已经全线转向TypeScript，98.3%使用TypeScript进行了重构；
- 而Angular在很早期就使用TypeScript进行了项目重构并且需要使用TypeScript来进行开发；
- 而甚至Facebook公司一些自己的产品也在使用TypeScript；

■ 学习TypeScript不仅仅可以为我们的代码增加类型约束，而且可以培养我们前端程序员具备类型思维。

- 如果之后想要学习其他语言，比如Java、Dart等也会是驾轻就熟；

认识TypeScript

- 虽然我们已经知道TypeScript是干什么的，也知道它解决了什么样的问题，但是我们还是需要全面的来认识一下TypeScript到底是什么？
- 我们来看一下TypeScript在GitHub和官方上对自己的定义：
 - GitHub说法：TypeScript is a superset of JavaScript that compiles to clean JavaScript output.
 - TypeScript官网：TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
 - 翻译一下：TypeScript是拥有类型的JavaScript超集，它可以编译成普通、干净、完整的JavaScript代码。
- 怎么理解上面的话呢？
 - 我们可以将TypeScript理解成加强版的JavaScript。
 - JavaScript所拥有的特性，TypeScript全部都是支持的，并且它紧随ECMAScript的标准，所以ES6、ES7、ES8等新语法标准，它都是支持的；
 - TypeScript在实现新特性的同时，总是保持和ES标准的同步甚至是领先；
 - 并且在语言层面上，不仅仅增加了类型约束，而且包括一些语法的扩展，比如枚举类型（Enum）、元组类型（Tuple）等；
 - 并且TypeScript最终会被编译成JavaScript代码，所以你并不需要担心它的兼容性问题，在编译时也可以不借助于Babel这样的工具；
- 所以，我们可以把TypeScript理解成更加强大的JavaScript，不仅让JavaScript更加安全，而且给它带来了诸多好用的好用特性；

TypeScript的特点

■ 官方对TypeScript有几段特点的描述，我觉得非常到位（虽然有些官方，了解一下），我们一起来分享一下：

■ 始于JavaScript，归于JavaScript

- TypeScript从今天数以百万计的JavaScript开发者所熟悉的语法和语义开始；
- 使用现有的JavaScript代码，包括流行的JavaScript库，并从JavaScript代码中调用TypeScript代码；
- TypeScript可以编译出纯净、简洁的JavaScript代码，并且可以运行在任何浏览器上、Node.js环境中任何支持ECMAScript 3（或更高版本）的JavaScript引擎中；

■ TypeScript是一个强大的工具，用于构建大型项目

- 类型允许JavaScript开发者在开发JavaScript应用程序时使用高效的开发工具和常用操作比如静态检查和代码重构；
- 类型是可选的，类型推断让一些类型的注释使你的代码的静态验证有很大的不同。类型让你定义软件组件之间的接口和洞察现有JavaScript库的行为；

■ 拥有先进的 JavaScript

- TypeScript提供最新的和不断发展的JavaScript特性，包括那些来自2015年的ECMAScript和未来的提案中的特性，比如异步功能和Decorators，以帮助建立健壮的组件；
- 这些特性为高可信应用程序开发时是可用的，但是会被编译成简洁的ECMAScript3（或更新版本）的JavaScript；

众多项目采用TypeScript

■ 正是因为有这些特性，TypeScript目前已经在很多地方被应用：

- Angular源码在很早就使用TypeScript来进行了重写，并且开发Angular也需要掌握TypeScript；
- Vue3源码也采用了TypeScript进行重写，在阅读源码时你会看到大量TypeScript的语法；
- 包括目前已经变成最流行的编辑器VSCode也是使用TypeScript来完成的；
- 包括在React中已经使用的ant-design的UI库，也大量使用TypeScript来编写；
- 目前公司非常流行Vue3+TypeScript、React+TypeScript的开发模式；
- 包括小程序开发，也是支持TypeScript的；

■ 在之前deno的issue里面出现了一个问题：

denoland / deno

<> Code

Issues 588

Pull requests 37

Discussions

Actions

Wiki

Security 1

Insights

求不要更新了，老子学不动了 #25

Closed

fujiazhang opened this issue on 31 May 2018 · 513 comments



fujiazhang commented on 31 May 2018 • edited

求不要更新了，老子学不动了



740



342



226



97



67



92



fujiazhang changed the title 不要更新子，老子学不动子 求不要更新了，老子学不动了 on 31 May 2018



zjafei commented on 31 May 2018

WTF



26



3



3

大前端的发展趋势

■ 大前端是一群最能或者说最需要折腾的开发者：

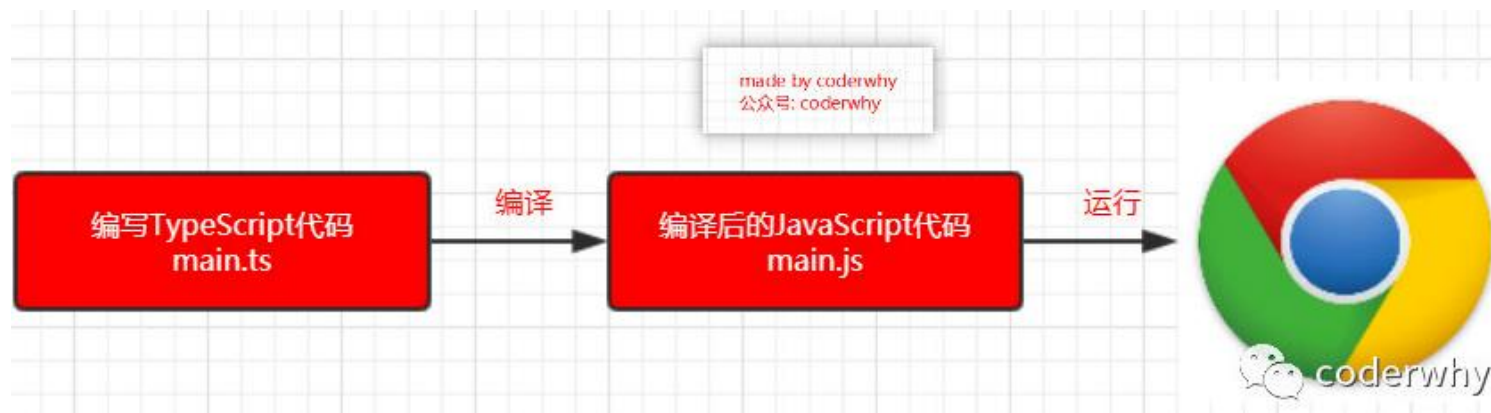
- ❑ **客户端开发者**：从Android到iOS，或者从iOS到Android，到RN，甚至现在越来越多的客户端开发者接触前端相关知识（Vue、React、Angular、小程序）；
- ❑ **前端开发者**：从jQuery到AngularJS，到三大框架并行：Vue、React、Angular，还有小程序，甚至现在也要接触客户端开发（比如RN、Flutter）；
- ❑ 目前又面临着不仅仅学习ES的特性，还要学习TypeScript；
- ❑ 新框架的出现，我们又需要学习新框架的特性，比如vue3.x、react18等等；
- ❑ 新的工具也是层出不穷，比如vite（版本更新也很快）；

■ 但是每一样技术的出现都会让惊喜，因为他必然是解决了之前技术的某一个痛点的，而TypeScript真是解决了JavaScript存在的很多设计缺陷，尤其是关于类型检测的。

■ 并且从开发者长远的角度来看，学习TypeScript有助于我们前端程序员培养 类型思维，这种思维方式对于完成大型项目尤为重要。

TypeScript的编译环境

- 在前面我们提到过，TypeScript最终会被编译成JavaScript来运行，所以我们需要搭建对应的环境：
 - 我们需要在电脑上安装TypeScript，这样就可以通过TypeScript的Compiler将其编译成JavaScript；



- 所以，我们需要先可以先进行全局的安装：

```
# 安装命令
npm install typescript -g

# 查看版本
tsc --version
```

TypeScript的运行环境

■ 如果我们每次为了查看TypeScript代码的运行效果，都通过经过两个步骤的话就太繁琐了：

- 第一步：通过tsc编译TypeScript到JavaScript代码；
- 第二步：在浏览器或者Node环境下运行JavaScript代码；

■ 是否可以简化这样的步骤呢？

- 比如编写了TypeScript之后可以直接运行在浏览器上？
- 比如编写了TypeScript之后，直接通过node的命令来执行？

■ 上面我提到的两种方式，可以通过两个解决方案来完成：

- 方式一：通过webpack，配置本地的TypeScript编译环境和开启一个本地服务，可以直接运行在浏览器上；
- 方式二：通过ts-node库，为TypeScript的运行提供执行环境；

■ 方式一：webpack配置

- 方式一在之前的TypeScript文章中我已经有写过，如果需要可以自行查看对应的文章；
- <https://mp.weixin.qq.com/s/wnL1l-ERjTDykWM76l4Ajw>；

■ 方式二：安装ts-node

```
npm install ts-node -g
```

■ 另外ts-node需要依赖 tslib 和 @types/node 两个包：

```
npm install tslib @types/node -g
```

■ 现在，我们可以直接通过 ts-node 来运行TypeScript的代码：

```
ts-node math.ts
```


变量的声明

- 我们已经强调过很多次，在TypeScript中定义变量需要指定 标识符 的类型。
- 所以完整的声明格式如下：
 - 声明了类型后TypeScript就会进行**类型检测**，声明的类型可以称之为**类型注解 (Type Annotation)**；

`var/let/const 标识符: 数据类型 = 赋值;`

- 比如我们声明一个message，完整的写法如下：
 - 注意：这里的string是小写的，和String是有区别的
 - string是TypeScript中定义的字符串类型，String是ECMAScript中定义的一个类

```
let message: string = "Hello World";
```

- 如果我们给message赋值其他类型的值，那么就会报错：

```
Type 'number' is not assignable to type 'string'. ts(2322)
let message1: string
View Problem (\F8) No quick fixes available
message1 = 123
```

声明变量的关键字

- 在TypeScript定义变量（标识符）和ES6之后一致，可以使用var、let、const来定义。

```
var myname: string = "why";  
let myage: number = 20;  
const myheight: number = 1.88;
```

- 当然，在tslint中并不推荐使用var来声明变量：

- 可见，在TypeScript中并不建议再使用var关键字了，主要原因和ES6升级后let和var的区别是一样的，var是没有块级作用域的，会引起很多的问题，这里不再展开探讨。

```
var myname: string = "abc";
```

```
Forbidden 'var' keyword, use 'let' or 'const' instead (no-var-keyword) tslint(1)
```

变量的类型推导（推断）

- 在开发中，有时候为了方便起见我们并不会在声明每一个变量时都写上对应的数据类型，我们更希望可以通过TypeScript本身的特性帮助我们推断出对应的变量类型：

```
let message: string
Quick Fix... (⌘.)
let message = "Hello World";
```

- 如果我们给message赋值123：

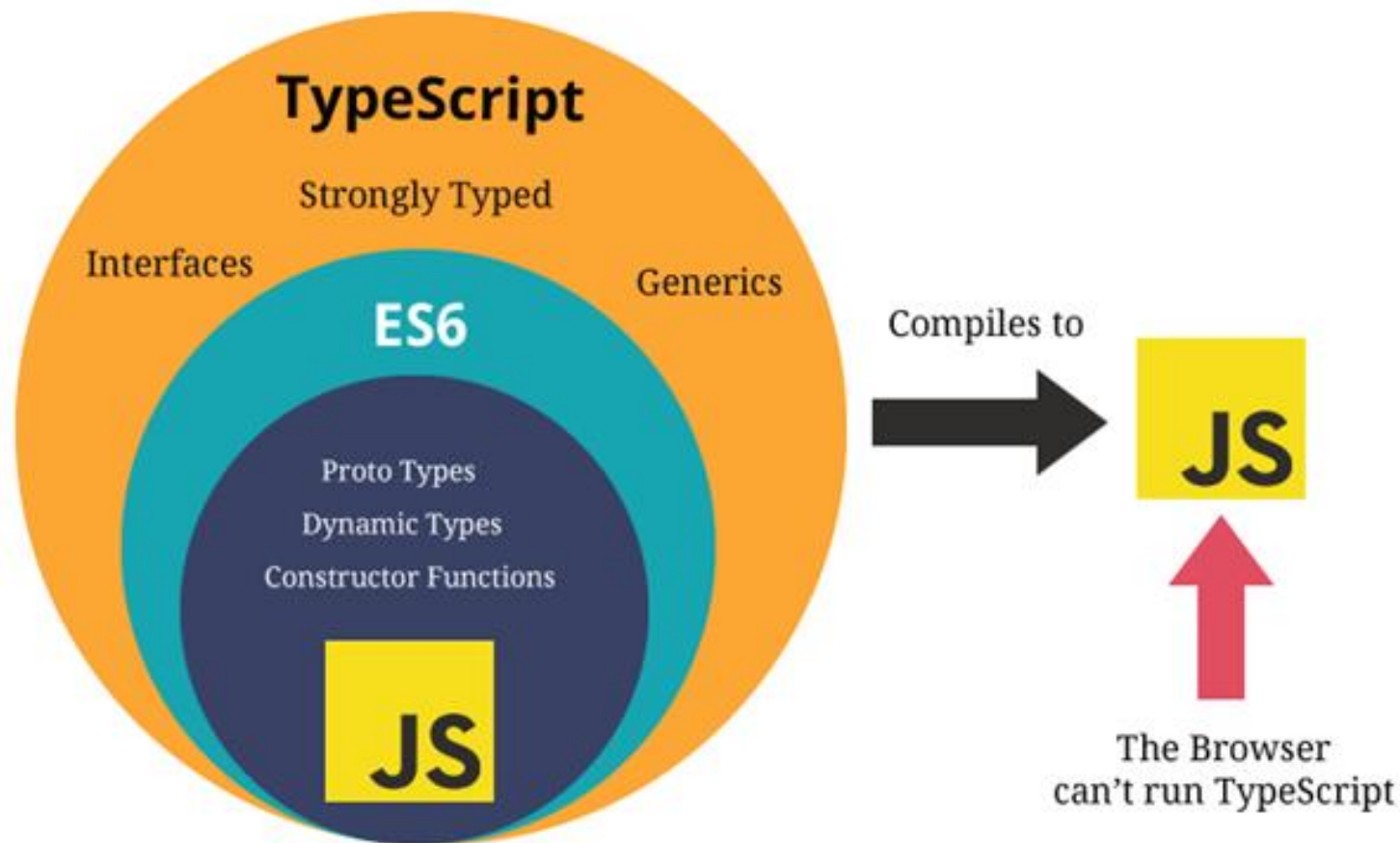
```
Type 'number' is not assignable to type 'string'. ts(2322)
let message: string
View Problem (⌘F8) No quick fixes available
message = 123
```

- 这是因为在一个变量第一次赋值时，会根据后面的赋值内容的类型，来推断出变量的类型：
 - 上面的message就是因为后面赋值的是一个string类型，所以message虽然没有明确的说明，但是依然是一个string类型；

Keep in mind, we don't always have to write explicit type annotations. In many cases, TypeScript can even just *infer* (or "figure out") the types for us even if we omit them.

JavaScript和TypeScript的数据类型

- 我们经常说TypeScript是JavaScript的一个超级:



JavaScript类型 – number类型

- 数字类型是我们开发中经常使用的类型，TypeScript和JavaScript一样，不区分整数类型（int）和浮点型（double），统一为number类型。

```
let num = 100;  
num = 20;  
num = 6.66;
```

- 学习过ES6应该知道，ES6新增了二进制和八进制的表示方法，而TypeScript也是支持二进制、八进制、十六进制的表示：

```
// 2. 其他进制表示  
num = 100; // 十进制  
num = 0b110; // 二进制  
num = 0o555; // 八进制  
num = 0xf23; // 十六进制
```

JavaScript类型 – boolean类型

- boolean类型只有两个取值：true和false，非常简单

```
// boolean类型的表示  
let flag: boolean = true;  
flag = false;  
flag = 20 > 30;
```

JavaScript类型 – string类型

- string类型是字符串类型，可以使用单引号或者双引号表示：

```
// string类型表示  
let message: string = "Hello World";  
message = 'Hello TypeScript';
```

- 同时也支持ES6的模板字符串来拼接变量和字符串：

```
const name = "why";  
const age = 18;  
const height = 1.88;  
  
const info = `my name is ${name}, age is ${age}, height is ${height}`;  
console.log(info);
```

JavaScript类型 – Array类型

■ 数组类型的定义也非常简单，有两种方式：

□ `Array<string>` 事实上是一种泛型的写法，我们会在后续中学习它的用法；

```
const names: string[] = ["abc", "cba", "cba"]
const names2: Array<string> = ["abc", "cba", "nba"]

names.push("why")
names2.push("why")
```

■ 如果添加其他类型到数组中，那么会报错：

```
names.push(123)
names2.push('string').ts(2345)
```

View Problem (⌘F8) No quick fixes available

```
names.push(123)
names2.push(123)
```


JavaScript类型 – Object类型

- object对象类型可以用于描述一个对象：

```
const myInfo: object = {  
  name: "why",  
  age: 18,  
  height: 1.88  
}
```

- 但是从myinfo中我们不能获取数据，也不能设置数据：

```
myInfo["name"] = "coderwhy"  
console.log(myInfo["age"])
```

```
01_JavaScript数据类型.ts:16:1 - error TS7053: Element implicitly has an 'any' type because expression of type '"name"' can't be  
used to index type '{}'.  
Property 'name' does not exist on type '{}'.  
16 myInfo["name"] = "coderwhy"
```

```
01_JavaScript数据类型.ts:17:13 - error TS7053: Element implicitly has an 'any' type because expression of type '"age"' can't be  
used to index type '{}'.  
Property 'age' does not exist on type '{}'.  
17 console.log(myInfo["age"])
```

JavaScript类型 – Symbol类型

- 在ES5中，如果我们是不可以在对象中添加相同的属性名称的，比如下面的做法：

```
const person = {  
  identity: "程序员",  
  identity: "老师",  
}
```

- 通常我们的做法是定义两个不同的属性名字：比如identity1和identity2。
- 但是我们也可以通过symbol来定义相同的名称，因为Symbol函数返回的是不同的值：

```
const s1: symbol = Symbol("title")  
const s2: symbol = Symbol("title")  
  
const person = {  
  [s1]: "程序员",  
  [s2]: "老师"  
}
```

JavaScript类型 – null和undefined类型

- 在 JavaScript 中，undefined 和 null 是两个基本数据类型。
- 在TypeScript中，它们各自的类型也是undefined和null，也就意味着它们既是实际的值，也是自己的类型：

```
let n: null = null  
let u: undefined = undefined
```

函数的参数类型

- 函数是JavaScript非常重要的组成部分，TypeScript允许我们指定函数的参数和返回值的类型。
- 参数的类型注解
 - 声明函数时，可以在每个参数后添加类型注解，以声明函数接受的参数类型：

```
function greet(name: string) {  
  console.log("Hello " + name.toUpperCase())  
}  
  
// Argument of type 'number' is not assignable to parameter of type 'string'  
greet(123)  
  
// Expected 1 arguments, but got 2.ts(2554)  
greet("abc", "cba")
```

函数的返回值类型

- 我们也可以添加返回值的类型注解，这个注解出现在函数列表的后面：

```
function sum(num1: number, num2: number): number {  
  return num1 + num2  
}
```

- 和变量的类型注解一样，我们通常情况下不需要返回类型注解，因为TypeScript会根据 `return` 返回值推断函数的返回类型：
 - 某些第三方库处于方便理解，会明确指定返回类型，看个人喜好；

匿名函数的参数

■ 匿名函数与函数声明会有一些不同：

- 当一个函数出现在TypeScript可以确定该函数会被如何调用的地方时；
- 该函数的参数会自动指定类型；

```
const names = ["abc", "cba", "nba"]
names.forEach(item => {
  console.log(item.toUpperCase())
})
```

■ 我们并没有指定item的类型，但是item是一个string类型：

- 这是因为TypeScript会根据forEach函数的类型以及数组的类型推断出item的类型；
- 这个过程称之为上下文类型 (*contextual typing*)，因为函数执行的上下文可以帮助确定参数和返回值的类型；

■ 如果我们希望限定一个函数接受的参数是一个对象，这个时候要如何限定呢？

□ 我们可以使用对象类型；

```
function printCoordinate(point: {x: number, y: number}) {  
    console.log("x坐标:", point.x)  
    console.log("y坐标:", point.y)  
}  
  
printCoordinate({x: 10, y: 30})
```

■ 在这里我们使用了一个对象来作为类型：

□ 在对象我们可以添加属性，并且告知TypeScript该属性需要是什么类型；

□ 属性之间可以使用，或者；来分割，最后一个分隔符是可选的；

□ 每个属性的类型部分也是可选的，如果不指定，那么就是any类型；

- 对象类型也可以指定哪些属性是可选的，可以在属性的后面添加一个?：

```
function printCoordinate(point: {x: number, y: number, z?: number}) {  
    console.log("x坐标:", point.x)  
    console.log("y坐标:", point.y)  
    if (point.z) {  
        console.log("z坐标:", point.z)  
    }  
}  
  
printCoordinate({x: 10, y: 30})  
printCoordinate({x: 20, y: 30, z: 40})
```


TypeScript类型 - any类型

- 在某些情况下，我们确实**无法确定一个变量的类型**，并且**可能它会发生一些变化**，这个时候我们可以使用**any类型**（类似于Dart语言中的dynamic类型）。
- any类型有点像一种讨巧的TypeScript手段：
 - 我们可以**对any类型的变量进行任何的操作**，包括**获取不存在的属性、方法**；
 - 我们**给一个any类型的变量赋值任何的值**，比如数字、字符串的值；

```
let a: any = "why";  
a = 123;  
a = true;  
  
const aArray: any[] = ["why", 18, 1.88];
```

- 如果对于某些情况的处理过于繁琐不希望添加规定的类型注解，或者在引入一些第三方库时，缺失了类型注解，这个时候我们可以使用any：
 - 包括在Vue源码中，也会使用到any来进行某些类型的适配；

TypeScript类型 - unknown类型

■ unknown是TypeScript中比较特殊的一种类型，它用于描述类型不确定的变量。

□ 和any类型有点类似，但是unknown类型的值上做任何事情都是不合法的；

■ 什么意思呢？我们来看下面的场景：

```
function foo(): string {  
  return 'foo'  
}  
  
function bar(): number {  
  return 123  
}
```

```
const flag = true  
let result: unknown  
  
if (flag) {  
  result = foo()  
} else {  
  result = bar()  
}  
  
if (typeof result === 'string') {  
  console.log(result.length)  
}  
  
export {}
```

TypeScript类型 - void类型

- void通常用来指定一个函数是没有返回值的，那么它的返回值就是void类型：

```
function sum(num1: number, num2: number) {  
  console.log(num1 + num2)  
}
```

- 这个函数我们没有写任何类型，那么它默认返回值的类型就是void的，我们也可以显示的来指定返回值是void：

```
function sum(num1: number, num2: number): void {  
  console.log(num1 + num2)  
}
```

- 这里还有一个注意事项：

- 我们可以将undefined赋值给void类型，也就是函数可以返回undefined

- 当基于上下文的类型推导（Contextual Typing）推导出返回类型为 void 的时候，并不会强制函数一定不能返回内容。

```
type FnType = () => void  
  
const foo: FnType = () => {  
  return 123  
}
```

```
const names = ["abc", "cba", "nba"]  
names.forEach(item => item.length)
```

TypeScript类型 - never类型

■ **never** 表示永远不会发生值的类型，比如一个函数：

- 如果一个函数中是一个死循环或者抛出一个异常，那么这个函数会返回东西吗？
- 不会，那么写void类型或者其他类型作为返回值类型都不合适，我们就可以使用never类型；

```
function loopFun(): never {  
  while(true) {  
    console.log("123")  
  }  
}
```

```
function loopErr(): never {  
  throw new Error()  
}
```

```
function handleMessage(message: number|string) {  
  switch (typeof message) {  
    case 'string':  
      console.log('foo')  
      break  
    case 'number':  
      console.log('bar')  
      break  
    default:  
      const check: never = message  
  }  
}
```

■ **never**有什么样的应用场景呢？这里我们举一个例子，但是它用到了联合类型，后面我们会讲到：

TypeScript类型 - tuple类型

- tuple是元组类型，很多语言中也有这种数据类型，比如Python、Swift等。

```
const tInfo: [string, number, number] = ["why", 18, 1.88];  
const item1 = tInfo[0]; // why, 并且知道类型是string类型  
const item2 = tInfo[1]; // 18, 并且知道类型是number类型
```

- 那么tuple和数组有什么区别呢？

- 首先，数组中通常建议存放相同类型的元素，不同类型的元素是不推荐放在数组中。（可以放在对象或者元组中）
- 其次，元组中每个元素都有自己特性的类型，根据索引值获取到的值可以确定对应的类型；

```
const info: (string|number)[] = ["why", 18, 1.88]  
const item1 = info[0] // 不能确定类型  
  
const tInfo: [string, number, number] = ["why", 18, 1.88]  
const item2 = tInfo[0] // 一定是string类型
```

Tuple的应用场景

■ 那么tuple在什么地方使用的是最多的呢?

▣ tuple通常可以作为返回的值，在使用的时候会非常的方便；

```
function useState<T>(state: T): [T, (newState: T) => void] {  
  let currentState = state  
  const changeState = (newState: T) => {  
    currentState = newState  
  }  
  
  return [currentState, changeState]  
}  
  
const [counter, setCounter] = useState(10)
```