TypeScript 类型系统

TypeScript 的学习资料非常多，其中也不乏很多优秀的文章和教程。但是目前为止没有一个我特别满意的。原因有：

* 它们大多数没有一个清晰的主线，而是按照 API 组织章节的，内容在**逻辑上**比较零散。
* 大多是“讲是什么，怎么用“，而不是”讲为什么，讲原理“。
* 大多数内容比较枯燥，趣味性比较低。都是干巴巴的文字，没有图片，缺乏能够引起强烈共鸣的例子。

因此我的想法是做一套不同市面上大多数的 TypeScript 学习教程。以人类认知的角度思考问题，学习 TypeScript，通过通俗易懂的例子和图片来帮助大家建立 TypeScript 世界观。

系列安排：

* [上帝视角看 TypeScript](https://lucifer.ren/blog/2020/08/04/ts-internal/)（已发布）
* TypeScript 类型系统（就是本文）
* types 和 @types 是什么？
* [你不知道的 TypeScript 泛型（万字长文，建议收藏）](https://lucifer.ren/blog/2020/06/16/ts-generics/)（已发布）
* TypeScript 配置文件该怎么写？
* TypeScript 是如何与 React，Vue，Webpack 集成的？
* TypeScript 练习题

目录将来可能会有所调整。

注意，我的系列文章基本不会讲 API，因此需要你有一定的 TypeScript 使用基础，推荐两个学习资料。

* [深入理解 TypeScript](https://jkchao.github.io/typescript-book-chinese/)
* [官方文档](https://www.typescriptlang.org/docs/home)

结合这两个资料和我的系列教程，掌握 TypeScript 指日可待。

接下来，我们通过几个方面来从宏观的角度来看一下 TypeScript。

## 前言

上一节的[上帝视角看 TypeScript](https://lucifer.ren/blog/2020/08/04/ts-internal/)，我们从宏观的角度来对 Typescript 进行了一个展望。之所以把那个放到开头讲是让大家有一个大体的认识，不想让大家一叶障目。当你对整个宏观层面有了一定的了解，那么对 Typescript 的理解就不会错太多。相反，一开始就是具体的概念和 API，则很可能会让你丧失都整体的基本判断。

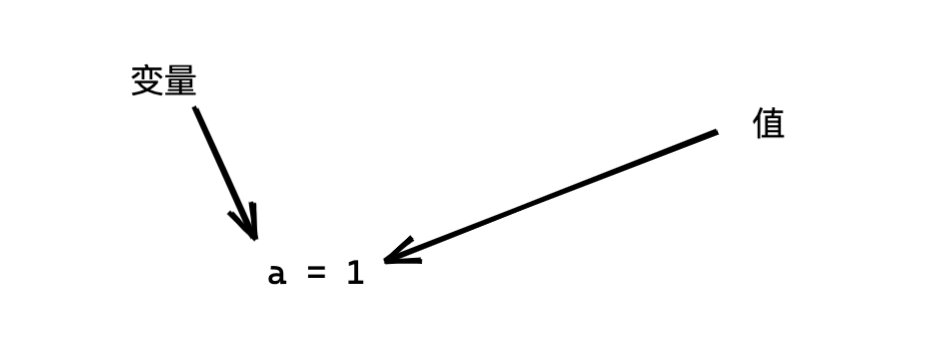
实际上， Typescript 一直在不断更新迭代。一方面是因为当初许下的诺言”Typescript 是 JavaScript 的超集“（JavaScript 的特性你要同步支持，同时也要处理各种新语法带来的不兼容情况）。不单是 ECMA，社区的其他发展可能也会让 Typescript 很难受。 比如 JSX 的广泛使用就给 Typescript 泛型的使用带来了影响。

TypeScript 一直处于高速的迭代。除了修复日常的 bug 之外，TypeScript 也在不断发布新的功能，比如最新 [4.0.0 beta 版本的](https://devblogs.microsoft.com/typescript/announcing-typescript-4-0-rc/#labeled-tuple-elements)**[标签元祖](https://devblogs.microsoft.com/typescript/announcing-typescript-4-0-rc/#labeled-tuple-elements)** 的功能就对智能提示这块很有用。Typescript 在社区发展方面也做的格外好，以至于它的竞争对手 Flow 被 Typescript 完美击败，这在很大程度上就是因为 Typescript 没有烂尾。如今微软在开源方向的发力是越来越显著了，我很期待微软接下来的表现，让我们拭目以待。



## 变量类型和值类型

有的同学可能有疑问， JavaScript 不是也有类型么？ 它和 Typescript 的类型是一回事么？JavaScript 不是动态语言么，那么经过 Typescript 的限定会不会丧失动态语言的动态性呢？我们继续往下看。



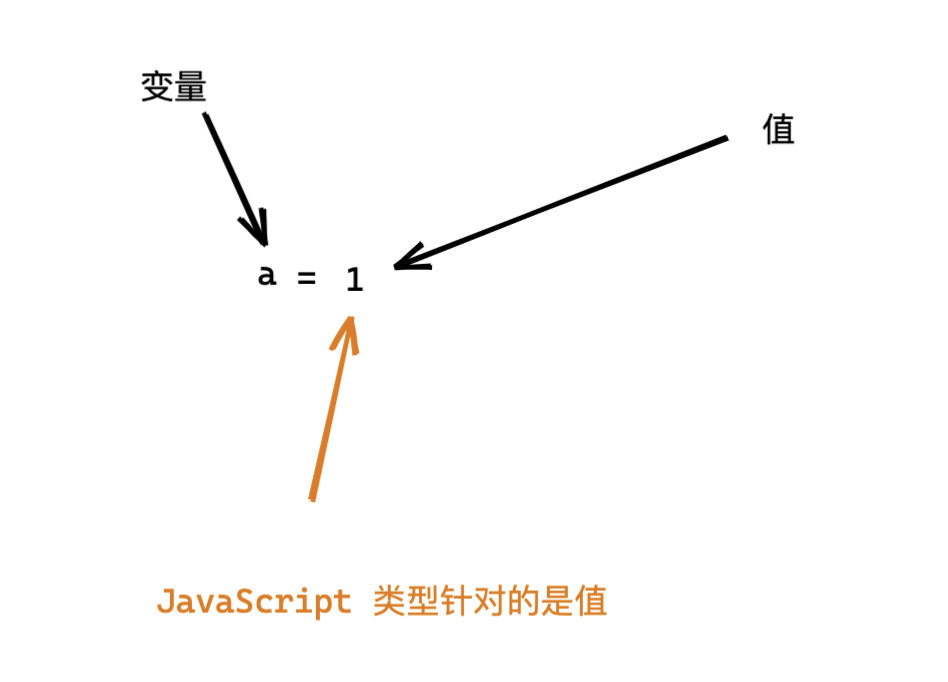
* JavaScript 中的类型其实是值的类型。实际上不仅仅是 JavaScript，任何动态类型语言都是如此，这也是动态类型语言的本质。
* Typescript 中的类型其实是变量的类型。实际上不仅仅是 Typescript，任何静态类型语言都是如此，这也是静态类型语言的本质。

记住这两句话，我们接下来解释一下这两句话。

对于 JavaScript 来说，一个变量可以是任意类型。

var a = 1;  
a = "lucifer";  
a = {};  
a = [];

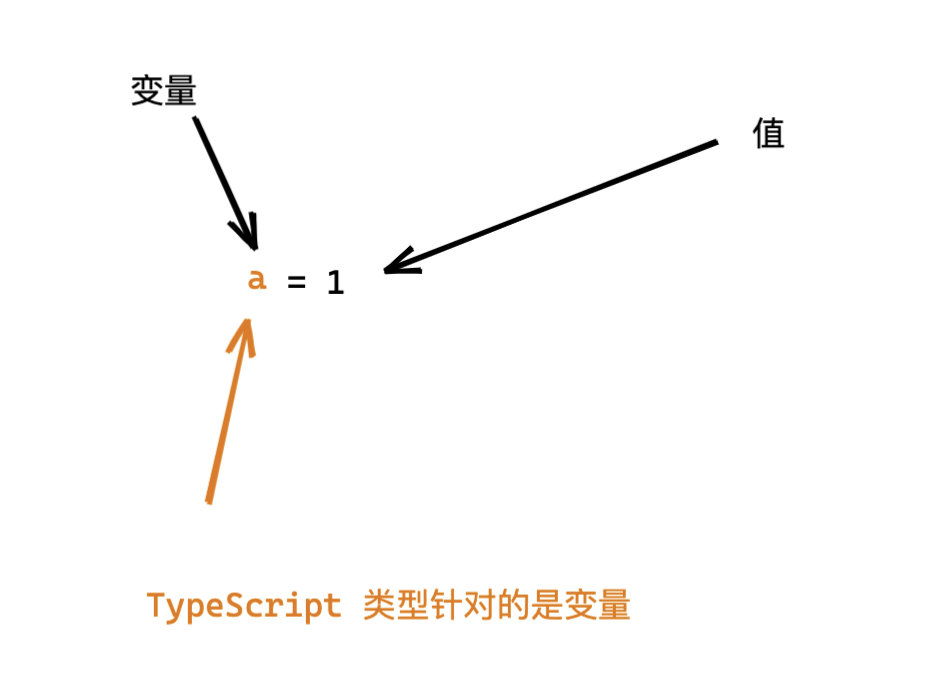
上面的值是有类型的。比如 1 是 number 类型，"lucifer" 是字符串类型， {} 是对象类型， [] 是数组类型。而变量 a 是没有固定类型的。



对于 Typescript 来说， 一个变量只能接受和它类型兼容的类型的值。说起来比较拗口， 看个例子就明白了。

var a: number = 1;  
a = "lucifer"; // error  
var b: any = 1;  
b = "lucifer"; // ok  
b = {}; // ok  
b = []; // ok

我们不能将 string 类型的值赋值给变量 a， 因为 string 和 number 类型不兼容。而我们可以将 string,Object,Array 类型的值赋值给 b，因此 它们和 any 类型兼容。简单来说就是，一旦一个变量被标注了某种类型，那么其就只能接受这个类型以及它的子类型。



## 类型空间和值空间

类型和值居住在不同的空间，一个在阳间一个在阴间。他们之间互相不能访问，甚至不知道彼此的存在。类型不能当做值来用，反之亦然。



### 类型空间

如下代码会报类型找不到的错：

const aa: User = { name: "lucifer", age: 17 };

这个比较好理解，我们只需要使用 interface 声明一下 User 就行。

interface User {  
 name: string;  
 age: number;  
}  
  
const aa: User = { name: "lucifer", age: 17 };

也就是说使用 interface 可以在类型空间声明一个类型，这个是 Typescript 的类型检查的基础之一。

实际上类型空间内部也会有子空间。我们可以用 namespace（老）和 module（新） 来创建新的子空间。子空间之间不能直接接触，需要依赖导入导出来交互。

### 值空间

比如，我用 Typescript 写出如下的代码：

const a = window.lucifer();

Typescript 会报告一个类似Property 'lucifer' does not exist on type 'Window & typeof globalThis'. 的错误。

实际上，这种错误并不是类型错误，而是找不到成员变量的错误。我们可以这样解决：

declare var lucifer: () => any;

也就是说使用 declare 可以在值空间声明一个变量。这个是 Typescript 的变量检查的基础，不是本文要讲的主要内容，大家知道就行。

明白了 JavaScript 和 TypeScript 类型的区别和联系之后，我们就可以来进入我们本文的主题了：**类型系统**。

## 类型系统是 TypeScript 最主要的功能

TypeScript 官方描述中有一句：**TypeScript adds optional types to JavaScript that support tools for large-scale JavaScript applications**。实际上这也正是 Typescript 的主要功能，即给 JavaScript 添加静态类型检查。要想实现静态类型检查，首先就要有类型系统。总之，我们使用 Typescript 的主要目的仍然是要它的静态类型检查，帮助我们提供代码的扩展性和可维护性。因此 Typescript 需要维护一套完整的类型系统。

**类型系统包括 1. 类型 和 2.对类型的使用和操作**，我们先来看类型。

### 类型

TypeScript 支持 JavaScript 中所有的类型，并且还支持一些 JavaScript 中没有的类型（毕竟是超集嘛）。没有的类型可以直接提供，也可以提供自定义能力让用户来自己创造。 那为什么要增加 JavaScript 中没有的类型呢？我举个例子，比如如下给一个变量声明类型为 Object，Array 的代码。

const a: Object = {};  
const b: Array = [];

其中：

* 第一行代码 Typescript 允许，但是太宽泛了，我们很难得到有用的信息，推荐的做法是使用 interface 来描述，这个后面会讲到。
* 第二行 Typescript 则会直接报错，原因的本质也是太宽泛，我们需要使用泛型来进一步约束。

### 对类型的使用和操作

上面说了**类型和值居住在不同的空间，一个在阳间一个在阴间。他们之间互相不能访问，甚至不知道彼此的存在。**

使用 declare 和 interface or type 就是分别在两个空间编程。比如 Typescript 的泛型就是在类型空间编程，叫做类型编程。除了泛型，还有集合运算，一些操作符比如 keyof 等。值的编程在 Typescript 中更多的体现是在类似 lib.d.ts 这样的库。当然 lib.d.ts 也会在类型空间定义各种内置类型。我们没有必要去死扣这个，只需要了解即可。

lib.d.ts 的内容主要是一些变量声明（如：window、document、math）和一些类似的接口声明（如：Window、Document、Math）。寻找代码类型（如：Math.floor）的最简单方式是使用 IDE 的 F12（跳转到定义）。

## 类型是如何做到静态类型检查的？

TypeScript 要想解决 JavaScript 动态语言类型太宽松的问题，就需要：

1. 提供给**变量**设定类型的能力

注意是变量，不是值。

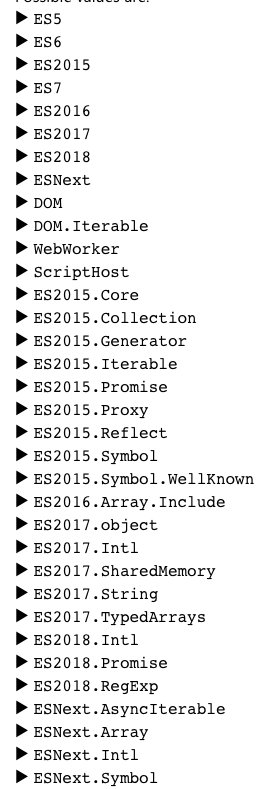
1. 提供常用类型（不必须，但是没有用户体验会极差）并可以扩展出自定义类型（必须）。
2. 根据第一步给变量设定的类型进行类型检查，即不允许类型不兼容的赋值， 不允许使用值空间和类型空间不存在的变量和类型等。

第一个点是通过类型注解的语法来完成。即类似这样：

const a: number = 1;

Typescript 的类型注解是这样， Java 的类型注解是另一个样子，Java 类似 int a = 1。 这个只是语法差异而已，作用是一样的。

第二个问题， Typescript 提供了诸如 lib.d.ts 等类型库文件。随着 ES 的不断更新， JavaScript 类型和全局变量会逐渐变多。Typescript 也是采用这种 lib 的方式来解决的。



（TypeScript 提供的部分 lib）

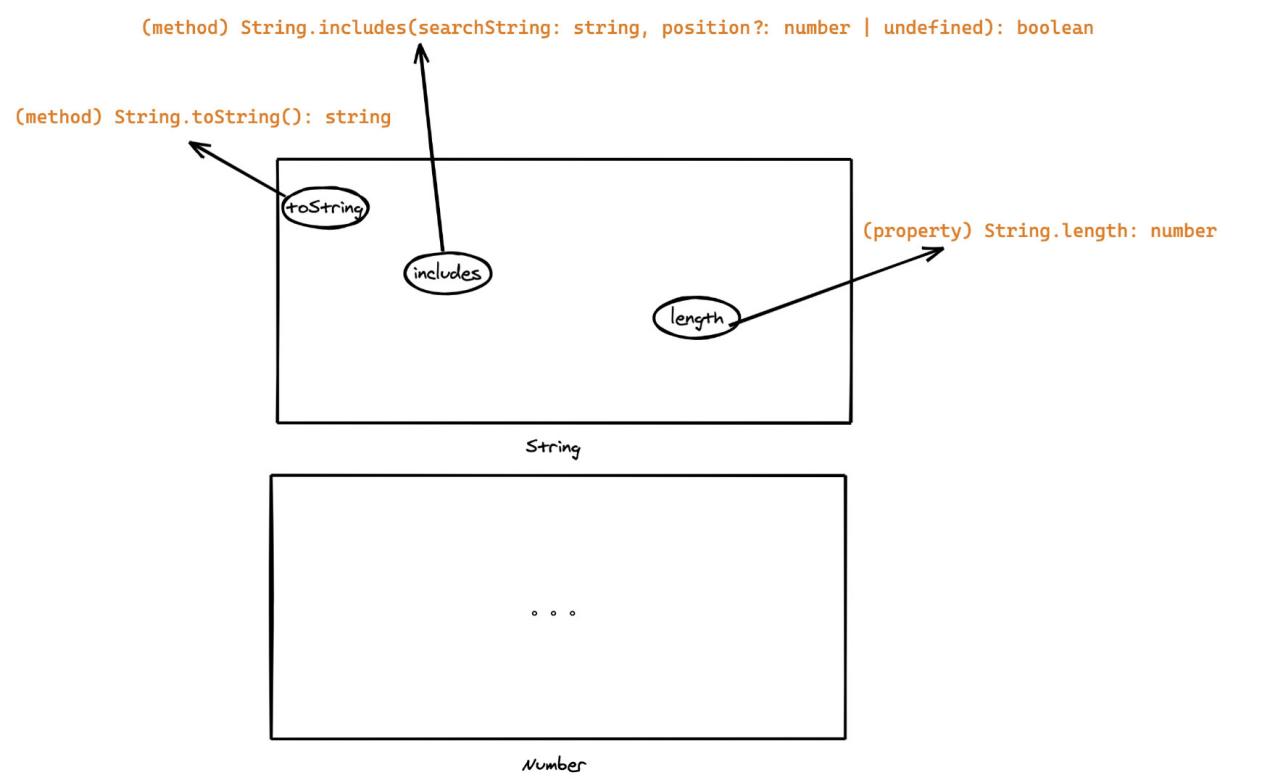
第三个问题，Typescript 主要是通过 interface，type，函数类型等打通**类型空间**，通过 declare 等打通**值空间**，并结合 binder 来进行类型诊断。关于 checker ，binder 是如何运作的，可以参考我第一篇的介绍。

接下来，我们介绍类型系统的功能，即它能为我们带来什么。如果上面的内容你已经懂了，那么接下来的内容会让你感到”你也不过如此嘛“。

## 类型系统的主要功能

1. 定义类型以及其上的属性和方法。

比如定义 String 类型， 以及其原型上的方法和属性。



length， includes 以及 toString 是 String 的**成员变量**， 生活在值空间， 值空间虽然不能直接和类型空间接触，但是类型空间可以作用在值空间，从而给其添加类型（如上图黄色部分）。

1. 提供自定义类型的能力

interface User {  
 name: string;  
 age: number;  
 say(name: string): string;  
}

这个是我自定义的类型 User，这是 Typescript 必须提供的能力。

1. 类型兼容体系。

这个主要是用来判断类型是否正确的，上面我已经提过了，这里就不赘述了。

1. 类型推导

有时候你不需要显式说明类型（类型注解），Typescript 也能知道他的类型，这就是类型推导结果。

const a = 1;

如上代码，编译器会自动推导出 a 的类型 为 number。还可以有连锁推导，泛型的入参（泛型的入参是类型）推导等。类型推导还有一个特别有用的地方，就是用到类型收敛。

接下来我们详细了解下类型推导和类型收敛。

## 类型推导和类型收敛

let a = 1;

如上代码。 Typescript 会推导出 a 的类型为 number。

如果只会你这么写就会报错：

a = "1";

因此 string 类型的值不能赋值给 number 类型的变量。我们可以使用 Typescript 内置的 typeof 关键字来证明一下。

let a = 1;  
type A = typeof a;

此时 A 的类型就是 number，证明了变量 a 的类型确实被隐式推导成了 number 类型。

有意思的是如果 a 使用 const 声明，那么 a 不会被推导为 number，而是推导为类型 1。即**值只能为 1 的类型**，这就是类型收敛。

const a = 1;  
type A = typeof a;

通过 const ，我们将 number 类型收缩到了 **值只能为 1 的类型**。

实际情况的类型推导和类型收敛要远比这个复杂， 但是做的事情都是一致的。

比如这个：

function test(a: number, b: number) {  
 return a + b;  
}  
type A = ReturnType<typeof test>;

A 就是 number 类型。 也就是 Typescript 知道两个 number 相加结果也是一个 number。因此即使你不显示地注明返回值是 number， Typescript 也能猜到。**这也是为什么 JavaScript 项目不接入 Typescript 也可以获得类型提示的原因之一**。

除了 const 可以收缩类型， typeof， instanceof 都也可以。 原因很简单，就是**Typescript 在这个时候可以 100% 确定你的类型了**。 我来解释一下：

比如上面的 const ，由于你是用 const 声明的，因此 100% 不会变，一定永远是 1，因此类型可以收缩为 1。 再比如：

let a: number | string = 1;  
a = "1";  
if (typeof a === "string") {  
 a.includes;  
}

if 语句内 a 100% 是 string ，不能是 number。因此 if 语句内类型会被收缩为 string。instanceof 也是类似，原理一模一样。大家只要记住**Typescript 如果可以 100% 确定你的类型，并且这个类型要比你定义的或者 Typescript 自动推导的范围更小，那么就会发生类型收缩**就行了。

## 总结

本文主要讲了 Typescript 的类型系统。 Typescript 和 JavaScript 的类型是很不一样的。从表面上来看， TypeScript 的类型是 JavaScript 类型的超集。但是从更深层次上来说，两者的本质是不一样的，一个是值的类型，一个是变量的类型。

Typescript 空间分为值空间和类型空间。两个空间不互通，因此值不能当成类型，类型不能当成值，并且值和类型不能做运算等。不过 TypeScript 可以将两者结合起来用，这个能力只有 TypeScript 有， 作为 TypeScript 的开发者的你没有这个能力，这个我在第一节也简单介绍了。

TypeScript 既会对变量存在与否进行检查，也会对变量类型进行兼容检查。因此 TypeScript 就需要定义一系列的类型，以及类型之间的兼容关系。默认情况，TypeScript 是没有任何类型和变量的，因此你使用 String 等都会报错。TypeScript 使用库文件来解决这个问题，最经典的就是 lib.d.ts。

TypeScript 已经做到了足够智能了，以至于你不需要写类型，它也能猜出来，这就是类型推导和类型收缩。当然 TypeScript 也有一些功能，我们觉得应该有，并且也是可以做到的功能空缺。但是我相信随着 TypeScript 的逐步迭代（截止本文发布，TypeScript 刚刚发布了 4.0.0 的 beta 版本），一定会越来越完善，用着越来越舒服的。

我们每个项目的需要是不一样的， 简单的基本类型肯定无法满足多样的项目需求，因此我们必须支持自定义类型，比如 interface， type 以及复杂一点的泛型。当然泛型很大程度上是为了减少样板代码而生的，和 interface ， type 这种刚需不太一样。

有了各种各样的类型以及类型上的成员变量，以及成员变量的类型，再就加上类型的兼容关系，我们就可以做类型检查了，这就是 TypeScript 类型检查的基础。TypeScript 内部需要维护这样的一个关系，并对变量进行类型绑定，从而给开发者提供**类型分析**服务。

## 关注我

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的前端硬核文章，带你认识你不知道的前端。



公众号【 [力扣加加](https://tva1.sinaimg.cn/large/007S8ZIlly1gfcuzagjalj30p00dwabs.jpg)】 知乎专栏【 [Lucifer - 知乎](https://www.zhihu.com/people/lu-xiao-13-70)】

点关注，不迷路！