

TypeScript 全面进阶指南

TypeScript 由三个部分组成：类型、语法与工程。



更方便的类型兼容性检查

如果只想要进行类型比较，可以不需要真的去声明两个变量，即设计了值空间的操作。我们完全可以只在类型空间中（用于存放TypeScript 类型信息的内存空间），

```
interface Foo {  
  name: string;  
  age: number;  
}  
  
interface Bar {  
  name: string;  
  job: string;  
}  
  
declare let foo: Foo;  
declare let bar: Bar;  
  
foo = bar;
```

我们使用一个值空间存放这个变量具体的属性，一个类型空间存放这个变量的类型。而通过 `declare` 关键字，我们声明了一个仅在类型空间存在的变量，它在运行时完全不存在，这样就避免了略显繁琐的属性声明。

还可以通过工具类型的形式，如 `tsd` 这个 npm 提供的一系列工具类型

```
import { expectType } from 'tsd';

expectType<string>("hoel"); // ✓
expectType<string>(599); // ✕
```

它的结构大致是这样：`expectType<你预期的类型>(表达式或变量等)`，除了 `expectType`（检查预期类型与表达式或变量的类型是否一致），`tsd` 还提供了 `expectNotType`（检查预期类型与表达式或变量的类型是否不同）、`expectAssignable`（检查表达式或变量的类型是否能赋值给预期类型）等工具类型。

在 JavaScript 中，`null` 与 `undefined` 分别表示“**这里有值，但是个空值**”和“**这里没有值**”。TypeScript 中，`null` 与 `undefined` 类型都是**有具体意义的类型**。也就是说，它们作为类型时，表示的是一个有意义的具体类型值。这两者在没有开启 `strictNullChecks` 检查的情况下，会被**视作其他类型的子类型**，比如 `string` 类型会被认为包含了 `null` 与 `undefined` 类型：

void

`void(0)#` 等价于 `void 0#`，即 `void expression#` 的语法。`void` 操作符会执行后面跟着的表达式并返回一个 `undefined`，如你可以使用它来执行一个立即执行函数（IIFE）

```
void function iife() {
  console.log("Invoked!");
}();
```

能这么做是因为，`void` 操作符强制将后面的函数声明转化为了表达式，因此整体其实相当于：

```
void((function iife(){}))()。
```

在TypeScript的原始类型标注中也有void，但与Js不同，这里的void用于描述一个内部没有return的语句，或者没有显式return一个值的函数的返回值

```
function func1(){}  
function func2(){  
    return;  
}  
function func3(){  
    return undefined;  
}
```

func1 与 func2 的返回值类型都会被隐式推导为 void，只有显式返回了 **undefined** 值的 **func3** 其返回值类型才被推导为了 **undefined**。但在实际的代码执行中，func1 与 func2 的返回值均是 undefined。

虽然 func3 的返回值类型会被推导为 undefined，但是你仍然可以使用 void 类型进行标注，因为在类型层面 func1、func2、func3 都表示“没有返回一个有意义的值”。

void 表示一个空类型，而 null 与 undefined 都是一个具有意义的实际类型

数组是我们日常开发大量使用的数据结构，但在某些情况下，使用元祖来替代数组要更加妥当，比如一个数组中只存放固定长度的变量，但我们进行了超出长度的访问

```
const arr4: [string, string, string] = ['shen', 'jun', 'hong'];  
  
console.log(arr4[599]);
```

元祖内部也可以声明多个与其位置强绑定的，不同类型的元素：

```
const arr5: [string, number, boolean] = ['HOEL', 047, true];
```

元祖也支持了在某一位置上的可选成员：

```
const arr6: [string, number?, boolean?] = ['hoe1'];
```

在TypeScript 4.0中，有了**具名元祖**，使得我们可以为元祖中的元素打上类似属性的标记

```
const arr7: [name: string, age: number, male?: boolean] = ['hoel', 047, true];
```

但使用元组确实能帮助我们进一步提升**数组结构的严谨性**，包括基于**位置的类型标注**、**避免出现越界访问**等等。除了通过数组类型提升数组结构的严谨性，TypeScript 中的对象类型也能帮助我们提升对象结构的严谨性

对象的类型标注

类似于数组类型，在 TypeScript 中我们也需要特殊的类型标注来描述对象类型，即 `interface`，你可以理解为它代表了这个对象对外提供的接口结构。

可选的函数类型属性将会产生一个类型报错：不能调用可能是未定义的方法。

除了标记一个属性为可选以外，你还可以标记这个属性为只读：`readonly`。

因为以往 JavaScript 中并没有这一类概念，它的作用是**防止对象的属性被再次赋值**。

```
interface IDescription {
    readonly name: string;
    age: number;
}

const obj3: IDescription = {
    name: 'hoel',
    age: 599,
};

*// 无法分配到 "name"，因为它是只读属性*
obj3.name = 'Dog';
```

type和interface

type (Type Alias, 类型别名) 来代替接口结构描述对象

- interface 用来描述对象、类的结构
- 类型别名用来将一个函数签名、一组联合类型、一个工具类型等等抽离成一个完整独立的类型。

object、Object 以及 {}

Object的使用，原型链的顶端是Object以及Function，这也就意味着所有的原始类型与对象类型最终都指向Object，在TS中就表现为Object包含了所有的类型。

```
// 对于 undefined、null、void 0，需要关闭 strictNullChecks
const tmp1: Object = undefined;
const tmp2: Object = null;
const tmp3: Object = void 0;

const tmp4: Object = 'hoel';
const tmp5: Object = 047;
const tmp6: Object = { name: 'hoel' };
const tmp7: Object = () => {};
const tmp8: Object = [];
```

和 Object 类似的还有 Boolean、Number、String、Symbol，这几个**装箱类型 (Boxed Types)** 同样包含了一些超出预期的类型。

以 String 为例，它同样包括 undefined、null、void，以及代表的 **拆箱类型 (Unboxed Types)** string，但并不包括其他装箱类型对应的拆箱类型，如 boolean 与 基本对象类型，我们看以下的代码：

```
const tmp9: String = undefined;
const tmp10: String = null;
const tmp11: String = void 0;
const tmp12: String = 'hoel';

// 以下不成立，因为不是字符串类型的拆箱类型
const tmp13: String = 599; // X
const tmp14: String = { name: 'hoel' }; // X
const tmp15: String = () => {}; // X
```

```
const tmp16: String = []; // X
```

在任何情况下，你都不应该使用这些装箱类型。

object 的引入就是为了解决对 Object 类型的错误使用，它代表所有非原始类型的类型，即数组、对象与函数类型这些：

```
const tmp25: {} = undefined; // 仅在关闭 strictNullChecks 时成立，下同
const tmp26: {} = null;
const tmp27: {} = void 0; // void 0 等价于 undefined

const tmp28: {} = 'hoel';
const tmp29: {} = 599;
const tmp30: {} = { name: 'hoel' };
const tmp31: {} = () => {};
const tmp32: {} = [];
```

最后是{}，一个奇奇怪怪的空对象，如果你了解过字面量类型，可以认为{}就是一个对象字面量类型（对应到字符串字面量类型这样）。否则，你可以认为使用{}作为类型签名就是一个合法的，但内部无属性定义的空对象，这类似于 Object（想想 new Object()），它意味着任何非 null / undefined 的值：

```
const tmp25: {} = undefined; // 仅在关闭 strictNullChecks 时成立，下同
const tmp26: {} = null;
const tmp27: {} = void 0; // void 0 等价于 undefined

const tmp28: {} = 'hoel';
const tmp29: {} = 599;
const tmp30: {} = { name: 'hoel' };
const tmp31: {} = () => {};
const tmp32: {} = [];
```

虽然能够将其作为变量的类型，但你实际上无法对这个变量进行任何赋值操作：

```
const tmp30: {} = { name: 'hoel' };

tmp30.age = 18; // X 类型“{}”上不存在属性“age”。
```

最后，为了更好地区分 Object、object 以及 {} 这三个具有迷惑性的类型，我们再做下总结：

- 在任何时候都不要，不要，不要使用 Object 以及类似的装箱类型。
- 当你不确定某个变量的具体类型，但能确定它不是原始类型，可以使用 object。但我更推荐进一步区分，也就是使用 Record<string, unknown> 或 Record<string, any> 表示对象，unknown[] 或 any[] 表示数组，(...args: any[]) => any 表示函数这样。
- 我们同样要避免使用 {}。{} 意味着任何非 null / undefined 的值，从这个层面上看，使用它和使用 any 一样恶劣。

掌握字面量类型与枚举，让你的类型在精确一些

字面量类型与联合类型

```
interface Res {
  code: 10000 | 10001 | 50000;
  status: "success" | "failure";
  data: any;
}
```

对于 **declare var res: Res**，你可以认为它其实就是快速生成一个符合指定类型，但没有实际值的变量，同时它也不存在于运行时中。

字面量类型

“success” 不是一个值吗？为什么它也可以作为类型？在 TypeScript 中，这叫做**字面量类型 (Literal Types)**，它代表着比原始类型更精确的类型，同时也是原始类型的子类型

字面量类型主要包括**字符串字面量类型**、**数字字面量类型**、**布尔字面量类型**和**对象字面量类型**，它们可以直接作为类型标注：

```
const str: 'hoel' = 'hoel';
const num: 599 = 599;
const bool: true = true;
```

为什么说字面量类型比原始类型更精确

```
// 报错! 不能将类型""hoel999""分配给类型""hoel""。
const str1: "hoel999" = "hoel";

const str2: string = "hoel999";
const str3: string = "hoel";
```

上面的代码，原始类型的值可以包括任意的同类型值，而字面量类型要求的是**值级别的字面量一致**。

单独使用字面量类型比较少见，因为单个字面量类型并没有什么实际意义。它通常和联合类型（即这里的 `|`）一起使用，表达一组字面量类型：

联合类型：

```
interface Tmp {
  mixed: true | string | 599 | {} | (() => {}) | (1 | 2)
}
```

- 对于联合类型中的函数类型，需要使用括号`()`包裹起来
- 函数类型并不存在字面量类型，因此这里的 `(() => {})` 就是一个合法的函数类型
- 你可以在联合类型中进一步嵌套联合类型，但这些嵌套的联合类型最终都会被展平到第一级中

联合类型的常用场景之一是通过多个对象类型的联合，来实现手动的互斥属性，即这一属性如果有字段1，那就没有字段2：

无论是原始类型还是对象类型的字面量类型，它们的本质都是类型而不是值。它们在编译时同样会被擦除，同时也是被存储在内存中的类型空间而非值空间。

枚举


```
enum PageUrl {  
    Home_Page_Url = "url1",  
    Setting_Page_Url = "url2",  
    Share_Page_Url = "url3",  
}  
  
const home = PageUrl.Home_Page_Url;
```

优点：

1. 更好的类型提示。
2. 这些常量被真正地约束在一个命名空间下（上面的对象声明总是差点意思）

如果你只为某一个成员指定了枚举值，那么之前未赋值成员仍然会使用从 0 递增的方式，之后的成员则会开始从枚举值递增。

```
enum Items {  
    // 0  
    Foo,  
    Bar = 599,  
    // 600  
    Baz  
}
```

枚举和对象的重要差异在于，**对象是单向映射的**，我们只能从键映射到键值。而**枚举是双向映射的**，即你可以从枚举成员映射到枚举值，也可以从枚举值映射到枚举成员：

```
enum Items {  
    Foo,  
    Bar,  
    Baz  
}  
  
const fooValue = Items.Foo; // 0  
const fooKey = Items[0]; // "Foo"
```

但需要注意的是，仅有值为数字的枚举成员才能够进行这样的双向枚举，字符串枚举成员仍然只会进行单次映射：

常量枚举

常量枚举和枚举相似，只是其声明多了一个 `const`：

```
**const** **enum** **Items** {  
    **Foo**,  
    **Bar**,  
    **Baz**  
}  
  
**const** fooValue = **Items**.Foo; */ / 0*
```

它和普通枚举的差异主要在访问性与编译产物。

对于常量枚举，你只能通过枚举成员访问枚举值（而不能通过值访问成员）。同时，在编译产物中并不会存在一个额外的辅助对象（如上面的 `Items` 对象），对枚举成员的访问会被直接内联替换为枚举的值。

以上的代码会被编译为如下形式：

```
const fooValue = 0 /* Foo */; // 0
```

使用 `let` 声明的变量是可以再次赋值的，在 TypeScript 中要求赋值类型始终与原类型一致（如果声明了的话）。

因此对于 `let` 声明，只需要推导至这个值从属的类型即可。

而 `const` 声明的原始类型变量将不再可变，因此类型可以直接一步到位收窄到最精确的字面量类型，但对对象类型变量仍可变（但同样会要求其属性值类型保持一致）。

函数与Class中的类型： 详解函数**重载**与**面向对象**

函数部分： 我们主要关注其参数类型、返回值类型以及重载的应用

class部分：除了类型以外，我们还会学习访问性修饰符、继承、抽象类

函数的类型签名：

如果说变量的类型是描述了这个变量的值类型，那么函数的类型就是描述了**函数入参类型与函数返回值类型**，它们同样使用:的语法进行类型标注。

函数表达式中进行类型声明的方式是这样的：

```
const foo = function (name: string): number {  
    return name.length  
}
```

也可以像对变量进行类型标注那样，对 foo 这个变量进行类型声明：

```
const foo: (name: string) => number = function (name) {  
    return name.length  
}
```

这里的 **(name: string) => number** 看起来很眼熟，对吧？它是 ES6 的重要特性之一：箭头函数。但在这里，它其实是 TypeScript 中的**函数类型签名**。而实际的箭头函数，我们的类型标注也是类似的：

```
// 方式一  
const foo = (name: string): number => {  
    return name.length  
}  
  
// 方式二  
const foo: (name: string) => number = (name) => {  
    return name.length  
}
```

void 类型

在 TypeScript 中，一个没有返回值（即没有调用 **return** 语句）的函数，其返回类型应当被标记为 **void** 而不是 **undefined**，即使它实际的值是 `undefined`。

```
// 没有调用 return 语句
function foo(): void { }

// 调用了 return 语句，但没有返回值
function bar(): void {
  return;
}
```

原因和我们在原始类型与对象类型一节中讲到的：

在 **TypeScript** 中，**undefined** 类型是一个实际的、有意义的类型值，而 **void** 才代表着空的、没有意义的类型值。

相比之下，`void` 类型就像是 JavaScript 中的 `null` 一样。因此在我们没有实际返回值时，使用 `void` 类型能更好地说明这个函数没有进行返回操作。

可选参数与 rest 参数

可选参数必须位于必选参数之后。毕竟在 JavaScript 中函数的入参是按照位置（形参），而不是按照参数名（名参）进行传递。

重载

在某些逻辑较复杂的情况下，函数可能有多组入参类型和返回值类型：

```
function func(foo: number, bar?: boolean): string | number {
  if (bar) {
    return String(foo);
  } else {
    return foo * 599;
  }
}
```

```
}  
}
```

在这个实例中，函数的返回类型基于其入参 `bar` 的值，并且从其内部逻辑中我们知道，当 `bar` 为 `true`，返回值为 `string` 类型，否则为 `number` 类型。而这里的类型签名完全没有体现这一点，我们只知道它的返回值是这么个联合类型。

要想实现与入参关联的返回值类型，我们可以使用 TypeScript 提供的**函数重载签名 (Overload Signature)**，将以上的例子使用重载改写：

```
function func(foo: number, bar: true): string;  
function func(foo: number, bar?: false): number;  
function func(foo: number, bar?: boolean): string | number {  
  if (bar) {  
    return String(foo);  
  } else {  
    return foo * 599;  
  }  
}  
  
const res1 = func(599); // number  
const res2 = func(599, true); // string  
const res3 = func(599, false); // number
```

这里我们的三个 `function func` 其实具有不同的意义：

- `function func(foo: number, bar: true): string`，重载签名一，传入 `bar` 的值为 `true` 时，函数返回值为 `string` 类型。
- `function func(foo: number, bar?: false): number`，重载签名二，不传入 `bar`，或传入 `bar` 的值为 `false` 时，函数返回值为 `number` 类型。
- `function func(foo: number, bar?: boolean): string | number`，函数的实现签名，会包含重载签名的所有可能情况。

基于重载签名，我们就实现了将入参类型和返回值类型的可能情况进行关联，获得了更精确的类型标注能力。

有多个重载声明的函数在被调用时，是按照重载的声明顺序往下查找的。因此在第一个重载声

明中，为了与逻辑中保持一致，即在 bar 为 true 时返回 string 类型，这里我们需要将第一个重载声明的 bar 声明为必选的字面量类型。

实际上，TypeScript 中的重载更像是伪重载，它只有一个具体实现，其重载体现在方法调用的签名上而非具体实现上。而在如 C++ 等语言中，重载体现在多个名称一致但入参不同的函数实现上，这才是更广义上的函数重载。

异步函数、Generator 函数等类型签名

```
async function asyncFunc(): Promise<void> {}

function* genFunc(): Iterable<void> {}

async function* asyncGenFunc(): AsyncIterable<void> {}
```

Promise 内部包含的类型则通过泛型的形式书写，即 Promise<T>

Class

类与类成员的类型签名

Class 中其实也一样，它的主要结构只有构造函数、属性、方法和访问符（**Accessor**）

属性的类型标注类似于变量，而构造函数、方法、存取器的类型标注类似于函数：

```
class Foo {
  prop: string;

  constructor(inputProp: string) {
    this.prop = inputProp;
  }

  print(addon: string): void {
    console.log(`${this.prop} and ${addon}`)
  }
}
```

```

get propA(): string {
    return `${this.prop}+A`;
}

set propA(value: string) {
    this.prop = `${value}+A`
}
}

```

setter 方法不允许进行返回值的类型标注, 你可以理解为 setter 的返回值并不会被消费, 它是一个只关注过程的函数。类的方法同样可以进行函数那样的重载, 且语法基本一致。

函数可以通过函数声明与函数表达式创建一样, 类也可以通过类声明和类表达式的方式创建。

```

const Foo = class {
    prop: string;

    constructor(inputProp: string) {
        this.prop = inputProp;
    }

    print(addon: string): void {
        console.log(`${this.prop} and ${addon}`)
    }

    // ...
}

```

修饰符

在 TypeScript 中我们能够为 Class 成员添加这些修饰符: **public / private / protected / readonly**。除 readonly 以外, 其他三位都属于访问性修饰符, 而 readonly 属于操作性修饰符 (就和 interface 中的 readonly 意义一致)。

这些修饰符应用的位置在成员命名前：

```
class Foo {  
  private prop: string;  
  
  constructor(inputProp: string) {  
    this.prop = inputProp;  
  }  
  
  protected print(addon: string): void {  
    console.log(`${this.prop} and ${addon}`)  
  }  
  
  public get propA(): string {  
    return `${this.prop}+A`;  
  }  
  
  public set propA(value: string) {  
    this.propA = `${value}+A`  
  }  
}
```

我们通常不会为构造函数添加修饰符，而是让它保持默认的 public。在扩展阅读中我们会讲到 private 修饰构造函数的场景。

- public：此类成员在类、类的实例、子类中都能被访问。
- private：此类成员仅能在类的内部被访问。
- protected：此类成员仅能在类与子类中被访问，你可以将类和类的实例当成两种概念，即一旦实例化完毕（出厂零件），那就和类（工厂）没关系了，即不允许再访问受保护的成员。

静态成员

在 TypeScript 中，可以使用 static 关键字来标识一个成员为静态成员：

```
class Foo {  
  static staticHandler() { }
```



```
public instanceHandler() { }  
}
```

不同于实例成员，在类的内部静态成员无法通过 `this` 来访问，需要通过 `Foo.staticHandler` 这种形式进行访问。

```
var Foo = /** @class */ (function () {  
    function Foo() {  
    }  
    Foo.staticHandler = function () { };  
    Foo.prototype.instanceHandler = function () { };  
    return Foo;  
})();
```

从中我们可以看到，静态成员直接被挂载在函数体上，而实例成员挂载在原型上，这就是二者的最重要差异：

静态成员不会被实例继承，它始终只属于当前定义的这个类（以及其子类）。而原型对象上的实例成员则会沿着原型链进行传递，也就是能够被继承。

****继承、实现、抽象类****

```
class Base { }  
  
class Derived extends Base { }
```

对于这里的两个类，比较严谨的称呼是**基类（Base）**与**派生类（Derived）**

关于基类与派生类，我们需要了解的主要是派生类对基类成员的访问与覆盖操作。

基类中的哪些成员能够被派生类访问，完全是由其访问性修饰符决定的。

派生类中可以访问到使用 **public** 或 **protected** 修饰符的基类成员。

除了访问以外，基类中的方法也可以在派生类中被覆盖，但我们仍然可以通过 `super` 访问到基类中的方法：

```
class Base {  
  print() { }  
}  
  
class Derived extends Base {  
  print() {  
    super.print()  
    // ...  
  }  
}
```

在派生类中覆盖基类方法时，我们并不能确保派生类的这一方法能覆盖基类方法，万一基类中不存在这个方法呢？

所以，TypeScript 4.3 新增了 `override` 关键字，来确保派生类尝试覆盖的方法一定在基类中存在定义：

```
class Base {  
  printWithLove() { }  
}  
  
class Derived extends Base {  
  override print() {  
    // ...  
  }  
}
```

在这里 TS 将会给出错误，因为尝试覆盖的方法并未在基类中声明，过这一关键字我们就能确保首先这个方法在基类中存在，同时标识这个方法在派生类中被覆盖了。

除了基类与派生类以外，还有一个比较重要的概念：**抽象类**。抽象类是对类结构与方法的抽象，简单来说，一个抽象类描述了一个类中应当有哪些成员（属性、方法等），一个抽象方法描述了这一方法在实际实现中的结构

我们知道类的方法和函数非常相似，包括结构，因此抽象方法其实描述的就是这个方法的入参类型与返回值类型。

抽象类使用 `abstract` 关键字声明：

```
abstract class AbsFoo {  
    abstract absProp: string;  
    abstract get absGetter(): string;  
    abstract absMethod(name: string): string  
}
```

注意，抽象类中的成员也需要使用 `abstract` 关键字才能被视为抽象类成员，如这里的抽象方法。我们可以实现（implements）一个抽象类：

```
class Foo implements AbsFoo {  
    absProp: string = "hoel"  
  
    get absGetter() {  
        return "hoel"  
    }  
  
    absMethod(name: string) {  
        return name  
    }  
}
```

此时，我们必须完全实现这个抽象类的每一个抽象成员。需要注意的是，在 TypeScript 中无法声明静态的抽象成员。

此时，我们必须完全实现这个抽象类的每一个抽象成员。需要注意的是，在 TypeScript 中无法声明静态的抽象成员。

对于抽象类，它的本质就是描述类的结构。看到结构，你是否又想到了 interface？是的。interface 不仅可以声明函数结构，也可以声明类的结构：

```
interface FooStruct {
  absProp: string;
  get absGetter(): string;
  absMethod(input: string): string
}

class Foo implements FooStruct {
  absProp: string = "hoel"

  get absGetter() {
    return "hoel"
  }

  absMethod(name: string) {
    return name
  }
}
```

在这里，我们让类去实现了一个接口。这里接口的作用和抽象类一样，都是描述这个类的结构。除此以外，我们还可以使用 **Newable Interface** 来描述一个类的结构（类似于描述函数结构的 **Callable Interface**）：


```
class Foo { }
```

```
interface FooStruct {
  new(): Foo
}
```

```
declare const NewableFoo: FooStruct;

const foo = new NewableFoo();
```

构建私有方法



```
/**
 * @file 单例模式
 * @author yiminfe
 */

// HACK: 定义单例模式
export class User {
  private static readonly instance: User = new User()

  public name = 'yiminfe'

  private constructor() {}

  private static getInstanceFn() {
    return User.instance
  }
  public static readonly getInstance = User.getInstanceFn
}
```

抽象类和接口：

抽象类是只为了类服务的，并且在运行时也会存在，而接口更多是服务对象类型的结构描述，并且在运行时就被擦除了。理论层面的话，抽象类就是为了描述一个类应该符合怎样的一个抽象结构，比如内部抽象方法的入参与返回值类型，它的本质是一个合法的类。从类型层面说明的话，抽象类是存在于值空间的类型描述与约束，而接口则只存在于类型空间。

```

54
55 class Utils {
56     public static identifier = "hoel";    "hoel": Unknown word.
57
58     private constructor(){}
59
60     public static makeUHappy() {
61     }
62 }
63
64 class sub extends Utils{    无法扩展类“Utils”。类构造函数标记为私有。
65
66 }

```

私有构造函数

```

class Utils {
    public static identifier = "hoel";

    private constructor(){}

    public static makeUHappy() {

```

私有构造函数能够阻止它被错误地实例化.或者在一个类希望把实例化逻辑通过方法来实现,而不是通过 new 的形式时,也可以使用私有构造函数来达成目的。

如何使用 TypeScript 提供的内置类型在类型世界里获得更好的编程体验。

any

而 any 类型的主要意义, 其实就是为了表示一个无拘无束的“任意类型”, 它能兼容所有类型, 也能够被所有类型兼容

- 如果是类型不兼容报错导致你使用 any, 考虑用类型断言替代, 我们下面就会开始介绍类型断言的作用。
- 如果是类型太复杂导致你不想全部声明而使用 any, 考虑将这一处的类型去断言为你需要的最简类型。如你需要调用 foo.bar.baz(), 就可以先将 foo 断言为一个具有 bar 方法的类型。
- 如果你是想表达一个未知类型, 更合理的方式是使用 unknown。

unknown 类型和 any 类型有些类似，一个 unknown 类型的变量可以再次赋值为任意其它类型，但只能赋值给 any 与 unknown 类型的变量：

```
let unknownVar: unknown = "hoel";

unknownVar = false;
unknownVar = "hoel";
unknownVar = {
  site: "juejin"
};

unknownVar = () => { }

const val1: string = unknownVar; // Error
const val2: number = unknownVar; // Error
const val3: () => {} = unknownVar; // Error
const val4: {} = unknownVar; // Error

const val5: any = unknownVar;
const val6: unknown = unknownVar;
```

unknown 和 any 的一个主要差异体现在赋值给别的变量时，any 就像是“我身化万千无处不在”，所有类型都把它当自己人。而 unknown 就像是“我虽然身化万千，但我坚信我在未来的某一刻会得到一个确定的类型”，只有 any 和 unknown 自己把它当自己人。简单地说，any 放弃了所有的类型检查，而 unknown 并没有。这一点也体现在对 unknown 类型的变量进行属性访问时：

```
let unknownVar: unknown;

unknownVar.foo(); // 报错：对象类型为 unknown
```

****虚无的 never 类型****

```
type UnionWithNever = "hoel" | 047 | true | void | never;
```

将鼠标悬浮在类型别名上，会发现这里显示的类型是 `"hoel" | 047 | true | void`，`never` 类型被直接无视掉了，而 `void` 仍然存在。`void` 作为类型表示一个空类型，就像没有返回值的函数使用 `void` 来作为返回值类型标注一样，`void` 类型就像 JavaScript 中的 `null` 一样代表“这里有类型，但是个空类型”。

never 类型不携带任何的类型信息，因此会在联合类型中被直接移除，比如我们看 `void` 和 `never` 的类型兼容性：

```
declare let v1: never;
declare let v2: void;

v1 = v2; // X 类型 void 不能赋值给类型 never

v2 = v1;
```

`never` 类型被称为 **Bottom Type**，是整个类型系统层级中最底层的类型。和 `null`、`undefined` 一样，它是所有类型的子类型，但只有 `never` 类型的变量能够赋值给另一个 `never` 类型变量。

`any` 的本质是类型系统中的顶级类型，即 `Top Type`

`never` 类型被称为 **Bottom Type**，是整个类型系统层级中最底层的类型。

通常我们不会显式地声明一个 `never` 类型，它主要被类型检查所使用。但在某些情况下使用 `never` 确实是符合逻辑的，比如一个只负责抛出错误的函数：

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

类型断言

类型断言的正确使用方式是，在 TypeScript 类型分析不正确或不符合预期时，将其断言为此处的正确类型：

```
interface IFoo {  
    name: string;  
}  
  
declare const obj: {  
    foo: IFoo  
}  
  
const {  
    foo = {} as IFoo  
} = obj
```

这里从 {} 字面量类型断言为了 IFoo 类型，即为解构赋值默认值进行了预期的类型断言。当然，更严谨的方式应该是定义为 Partial<IFoo> 类型，即 IFoo 的属性均为可选的。

双重断言

```
const str: string = "hoel";  
  
// 从 X 类型 到 Y 类型的断言可能是错误的, blabla  
(str as { handler: () => {} }).handler()
```

此时它会提醒你先断言到 unknown 类型，再断言到预期类型，就像这样：

```
const str: string = "hoel";  
  
(str as unknown as { handler: () => {} }).handler();  
  
// 使用尖括号断言  
(<{ handler: () => {} }>(<unknown>str)).handler();
```

非空断言

非空断言其实是类型断言的简化，它使用 `!` 语法，即 `obj!.func()!.prop` 的形式标记前面的一个声明一定是非空的（实际上就是剔除了 `null` 和 `undefined` 类型），比如这个例子：

```
declare const foo: {  
  func?: () => ({  
    prop?: number | null;  
  })  
};  
  
foo.func().prop.toFixed();
```

此时，`func` 在 `foo` 中不一定存在，`prop` 在 `func` 调用结果中不一定存在，且可能为 `null`，我们就会收获两个类型报错。如果不管三七二十一地坚持调用，想要解决掉类型报错就可以使用非空断言：

```
foo.func!().prop!.toFixed();
```

但不同的是，非空断言的运行时仍然会保持调用链，因此在运行时可能会报错。而可选链则会在某一个部分收到 `undefined` 或 `null` 时直接短路掉，不会再发生后面的调用。

非空断言的常见场景还有 `document.querySelector`、`Array.find` 方法等：

```
const element = document.querySelector("#id")!;  
const target = [1, 2, 3, 599].find(item => item === 599)!;
```

类型层级初探

- 最顶级的类型，`any` 与 `unknown`
- 特殊的 `Object`，它也包含了所有的类型，但和 `Top Type` 比还是差了一层
- `String`、`Boolean`、`Number` 这些装箱类型
- 原始类型与对象类型

- 字面量类型，即更精确的原始类型与对象类型嘛，需要注意的是 `null` 和 `undefined` 并不是字面量类型的子类型
- 最底层的 `never`

类型断言的工作原理也和类型层级有关，在判断断言是否成立，即差异是否能接受时，实际上判断的即是这两个类型是否能够找到一个公共的父类型。比如 `{}` 和 `{ name: string }` 其实可以认为拥有公共的父类型 `{}`（一个新的 `{}`！你可以理解为这是一个基类，参与断言的 `{}` 和 `{ name: string }` 其实是它的派生类）。

如果找不到具有意义的公共父类型呢？这个时候就需要请出 **Top Type** 了，如果我们把它先断言到 **Top Type**，那么就拥有了公共父类型 **Top Type**，再断言到具体的类型也是同理。你可以理解为先向上断言，再向下断言，比如前面的双重断言可以改写成这样：

```
const str: string = "hoel";

(str as (string | { handler: () => {} }) as { handler: () => {} }).handler();
```

类型工具

类型工具可以分成三类：**操作符、关键字与专用语法**。

而按照使用目的来划分，类型工具可以分为 **类型创建** 与 **类型安全保护** 两类

类型创建：基于已有的类型创建新的类型，这些类型工具包括类型别名、交叉类型、索引类型与映射类型。

类型别名

从一个简单的函数类型别名，到让你眼花缭乱的类型体操，都离不开类型别名。

类型别名的作用主要是对一组类型或一个特定类型结构进行封装，以便于在其它地方进行复用。

比如抽离一组联合类型：

```
type StatusCode = 200 | 301 | 400 | 500 | 502;
type PossibleDataTypes = string | number | (() => unknown);

const status: StatusCode = 502;
```

抽离一个函数类型：

```
type Handler = (e: Event) => void;

const clickHandler: Handler = (e) => { };
const moveHandler: Handler = (e) => { };
const dragHandler: Handler = (e) => { };
```

```
type ObjType = {
  name: string;
  age: number;
}
```

类型别名还能作为工具类型。工具类同样基于类型别名，只是多了个泛型。

在类型别名中，类型别名可以这么声明自己能够接受泛型（我称之为泛型坑位）。一旦接受了泛型，我们就叫它工具类型：

```
type Factory<T> = T | number | string;
```

虽然现在类型别名摇身一变成了工具类型，但它的基本功能仍然是创建类型，只不过工具类型能够接受泛型参数，实现更灵活的类型创建功能。

从这个角度看，工具类型就像一个函数一样，泛型是入参，内部逻辑基于入参进行某些操作，再返回一个新的类型。比如在上面这个工具类型中，我们就简单接受了一个泛型，然后把它作为联合类型的一个成员，返回了这个联合类型。

```
const foo: Factory<boolean> = true;
```

当然，我们一般不会直接使用工具类型来做类型标注，而是再度声明一个新的类型别名：

```
type FactoryWithBool = Factory<boolean>;

const foo: FactoryWithBool = true;
```

同时，泛型参数的名称（上面的 T）也不是固定的。通常我们使用大写的 T / K / U / V / M / O ... 这种形式。如果为了可读性考虑，我们也可以写成大驼峰形式（即在驼峰命名的基础上，首字母也大写）的名称，比如：

```
type Factory<NewType> = NewType | number | string;
```

声明一个简单、有实际意义的工具类型：

```
type MaybeNull<T> = T | null;
```

这个工具类型会接受一个类型，并返回一个包括 null 的联合类型。在实际使用时就可以确保你处理了可能为空值的属性读取与方法调用：

```
type MaybeNull<T> = T | null;

function process(input: MaybeNull<{ handler: () => {} }>) {
  input?.handler();
}
```

类似的还有 MaybePromise、MaybeArray

```

type MaybeArray<T> = T | T[];

// 函数泛型我们会在后面了解~
function ensureArray<T>(input: MaybeArray<T>): T[] {
    return Array.isArray(input) ? input : [input];
}

```

联合类型与交叉类型

按位与的 & 则不同，你需要符合这里的所有类型，才可以说实现了这个交叉类型，即 A & B，需要同时满足 **A** 与 **B** 两个类型才行。

```

97  interface NameStruct {
98      name: string;
99      age: string
100 }
101
102 interface AgeStruct {
103     age: number;
104 }
105
106 type ProfileStruct = NameStruct & AgeStruct;
107
108 const profile: ProfileStruct = {
109     name: "hoel",
110     age: 18
111 }
112

```

不能将类型“number”分配给类型“never”。

```

interface NameStruct {
    name: string;
    age: string
}

```

```
interface AgeStruct {
    age: number;
}

type ProfileStruct = NameStruct & AgeStruct;

const profile: ProfileStruct = {
    name: "hoel",
    age: 18
}
```

很明显这里的 profile 对象需要同时符合这两个对象的结构。从另外一个角度来看，ProfileStruct 其实就是一个新的，同时包含 NameStruct 和 AgeStruct 两个接口所有属性的类型。这里是对于对象类型的合并，那对于原始类型呢？

```
type StrAndNum = string & number; // never
```

新的类型会同时符合交叉类型的所有成员，存在既是 string 又是 number 的类型吗？当然不。

这也是 never 这一 BottomType 的实际意义之一，描述根本不存在的类型嘛。

对于对象类型的交叉类型，其内部的同名属性类型同样会按照交叉类型进行合并：

如果是两个联合类型组成的交叉类型呢？其实还是类似的思路，既然只需要实现一个联合类型成员就能认为是实现了这个联合类型，那么各实现两边联合类型中的一个就行了，也就是两边联合类型的交集：

```
type UnionIntersection1 = (1 | 2 | 3) & (1 | 2); // 1 | 2
type UnionIntersection2 = (string | number | symbol) & string; // string
```

总结一下交叉类型和联合类型的区别就是，联合类型只需要符合成员之一即可（||），而交叉类型需要严格符合每一位成员（&&）

索引类型

索引类型指的不是某一个特定的类型工具，它其实包含三个部分：**索引签名类型**、**索引类型查询**与**索引类型访问**。

它们都通过索引的形式来进行类型操作，但索引签名类型是**声明**，后两者则是**读取**

索引签名类型主要指的是在接口或类型别名中，通过以下语法来**快速声明一个键值类型一致的类型结构**：

```
interface AllStringTypes {  
    [key: string]: string;  
}  
  
type AllStringTypes = {  
    [key: string]: string;  
}
```

在js中，对于obj[prop]形式的访问会将数字索引访问转换为字符串索引访问，# 也就是说，obj[599]# 和obj['599']# 的效果是一致的。在字符串索引签名类型中我们仍然可以声明数字类型的键。类似的，symbol 类型也是如此：

```
const foo: AllStringTypes = {  
    "hoel": "599",  
    599: "hoel",  
    [Symbol("ddd")]: 'symbol',  
}
```

⚠ 索引签名类型也可以和具体的键值对类型声明并存，但这时这些具体的键值类型也需要符合索引签名类型的声明：

```
interface AllStringTypes {
```



```
// 类型“number”的属性“propA”不能赋给“string”索引类型“boolean”。
propA: number;
[key: string]: boolean;
}
```

这里的符合即指子类型，因此自然也包括联合类型：

```
interface StringOrBooleanTypes {
  propA: number;
  propB: boolean;
  [key: string]: number | boolean;
}
```

****索引类型查询****

keyof 操作符： 它可以将对象中的所有键转换为对应字面量类型，然后再组合成联合类型。注意，这里并不会将数字类型的键名转换为字符串类型字面量，而是仍然保持为数字类型字面量。

```
interface Foo {
  hoel: 1,
  599: 2
}

type FooKeys = keyof Foo; // "hoel" | 599
```

keyof 的产物必定是一个联合类型。

****索引类型访问****

`obj[expression]` 的方式来动态访问一个对象属性（即计算属性），`expression` 表达式会先被执行，然后使用返回值来访问属性。

```
interface NumberRecord {  
  [key: string]: number;  
}  
  
type PropType = NumberRecord[string]; // number
```

```
interface Foo {  
  propA: number;  
  propB: boolean;  
}  
  
type PropAType = Foo['propA']; // number  
type PropBType = Foo['propB']; // boolean
```

看起来这里就是普通的值访问，但实际上这里的‘propA’和‘propB’都是字符串字面量类型，而不是一个 **JavaScript** 字符串值。索引类型查询的本质其实就是，通过键的字面量类型（‘propA’）访问这个键对应的键值类型（**number**）。

使用字面量联合类型进行索引类型访问时，其结果就是将联合类型每个分支对应的类型进行访问后的结果，重新组装成联合类型。索引类型查询、索引类型访问通常会和映射类型一起搭配使用，前两者负责访问键，而映射类型在其基础上访问键值类型

```
interface Foo {  
  propA: number;  
  propB: boolean;  
  propC: string;  
}  
  
type PropTypeUnion = Foo[keyof Foo]; // string | number | boolean
```

索引类型的最佳拍档之一就是映射类型，同时映射类型也是类型编程中常用的一个手段。

****映射类型：类型编程的第一步****

不同于索引类型包含好几个部分，映射类型指的就是一个确切的类型工具。映射类型的主要作用即是基于键名映射到键值类型。

```
type Stringify<T> = {  
  [K in keyof T]: string;  
};
```

这个工具类型会接受一个对象类型（假设我们只会这么用），使用 `keyof` 获得这个对象类型的键名组成字面量联合类型，然后通过映射类型（即这里的 `in` 关键字）将这个联合类型的每一个成员映射出来，并将其键值类型设置为 `string`。

```
interface Foo {  
  prop1: string;  
  prop2: number;  
  prop3: boolean;  
  prop4: () => void;  
}  
  
type StringifiedFoo = Stringify<Foo>;  
  
// 等价于  
interface StringifiedFoo {  
  prop1: string;  
  prop2: string;  
  prop3: string;  
  prop4: string;  
}
```

```
type Clone<T> = {  
  [K in keyof T]: T[K];  
};
```

这里的T[K]其实就是上面说到的索引类型访问，我们使用键的字面量类型访问到了键值的类型，这里就相当于克隆了一个接口。需要注意的是，这里其实只有K in 属于映射类型的语法，keyof T 属于 keyof 操作符，[K in keyof T]的[]属于索引签名类型，T[K]属于索引类型访问。

类型工具	创建新类型的方式	常见搭配
类型别名	将一组类型/类型结构封装，作为一个新的类型	联合类型、映射类型
工具类型	在类型别名的基础上，基于泛型去动态创建新类型	基本所有类型工具
联合类型	创建一组类型集合，满足其中一个类型即满足这个联合类型（ ）	类型别名、工具类型
交叉类型	创建一组类型集合，满足其中所有类型才满足映射联合类型（&&）	类型别名、工具类型
索引签名类型	声明一个拥有任意属性，键值类型一致的接口结构	映射类型
索引类型查询	从一个接口结构，创建一个由其键名字符串字面量组成的联合类型	映射类型
索引类型访问	从一个接口结构，使用键名字符串字面量访问到对应的键值类型	类型别名、映射类型
映射类型	从一个联合类型依次映射到其内部的每一个类型	工具类型

