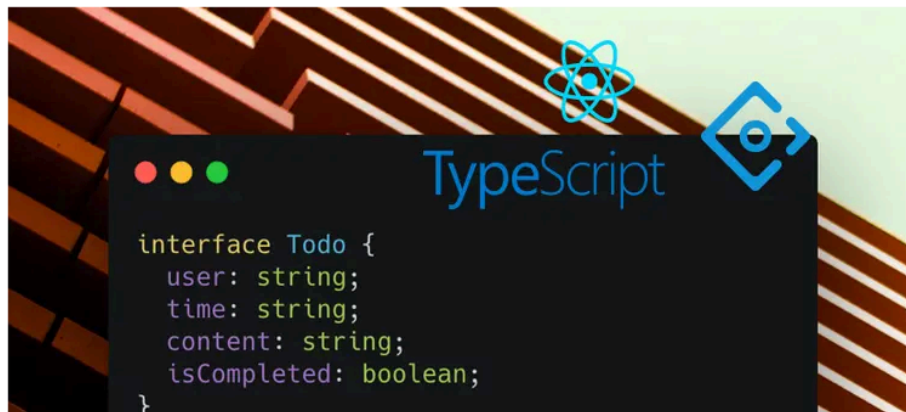


类型即正义：TypeScript 从入门到实践（一）

 一只图雀 2020-04-06 阅读 16 分钟



作者：一只图雀

图雀社区

[注册登录](#)

[主页](#) [关于](#) [RSS](#)

博客：[掘金](#)、[知乎](#)、[慕课](#)

公众号：[图雀社区](#)

联系我：关注公众号后可以加图雀酱微信哦

原创不易，❤️点赞+评论+收藏 ❤️三连，鼓励作者写出更好的教程。

源起

JavaScript 已经占领了世界上的每一个角落，能访问网页的地方，基本上就有 JavaScript 在运作，然而 JavaScript 因为其动态、弱类型、解释型语言的特性、出错的调用栈隐蔽，使得开发者不仅在调试错误上花费大把时间，在团队协作开发时理解队友编写代码也极其困难。TypeScript 的出现极大的解决了上面的问题，TypeScript -- 一个 JavaScript 的超集，它作为

一门编译型语言，提供了对类型系统和最新 ES 语法的支持，使得我们可以在享受使用 ES 最新语法的编写代码的同时，还能在写代码的过程中就规避很多潜在的语法、语义错误；并且其提供的类型系统使得我们可以在团队协作编写代码时可以很容易的了解队友代码的含义：输入和输出，大大提高了团队协作编写大型业务应用的效率。在现代 JavaScript 世界中，已经有[很多大型库在使用 TypeScript 重构](#)，包括前端三大框架：React、Vue、Angular，还有知名的组件库 antd, material，在很多公司内部的大型业务应用也在用 TypeScript 开发甚至重写现有的应用，所以如果你想编写大型业务应用或库，或者想写出更利于团队协作的代码，那么 TypeScript 有十足的理由值得你学习！本文是 TypeScript 系列教程的第一篇，主要通过使用 antd 组件库实战演练一个 TypeScript 版本 React TodoList 应用来讲解 TypeScript 的语法，使得你能在学会语法的同时还能完成一个实际可运行的项目。

本文所涉及的源代码都放在了 [Github](#) 或者 [Gitee](#) 上，如果您觉得我们写得还不错，希望您能给 ❤️ [这篇文章点赞](#) + [Github](#) 或 [Gitee](#) 仓库加星 ❤️ 哦~

此教程属于 [React 前端工程师学习路线](#) 的一部分，欢迎来 Star 一波，鼓励我们继续创作出更好的教程，持续更新中~

代码准备

我们接下来要讲解的整个 **类型即正义：TypeScript 从入门到实践** 系列是基于一个实战项目的，这个实战项目会贯穿整个系列教程的讲解周期，所以我们要尽可能全且精炼的讲解 TypeScript 语法知识的同时，还我们需要一个恰到好处的实战项目，因为准备项目代码的过程不是系列教程讲解的主线，所以如果你有兴趣学习如何搭建 TypeScript React 的开发环境，那么可以学习一下我们的序言教程：

类型即正义：TypeScript 从入门到实践（序章）

**

如果你已经对 TypeScript 如何搭建 React 开发环境，配置 Ant Design 组件库等很熟悉，或者不太感兴趣，那么你也可以直接克隆我们为你准备好的代码：

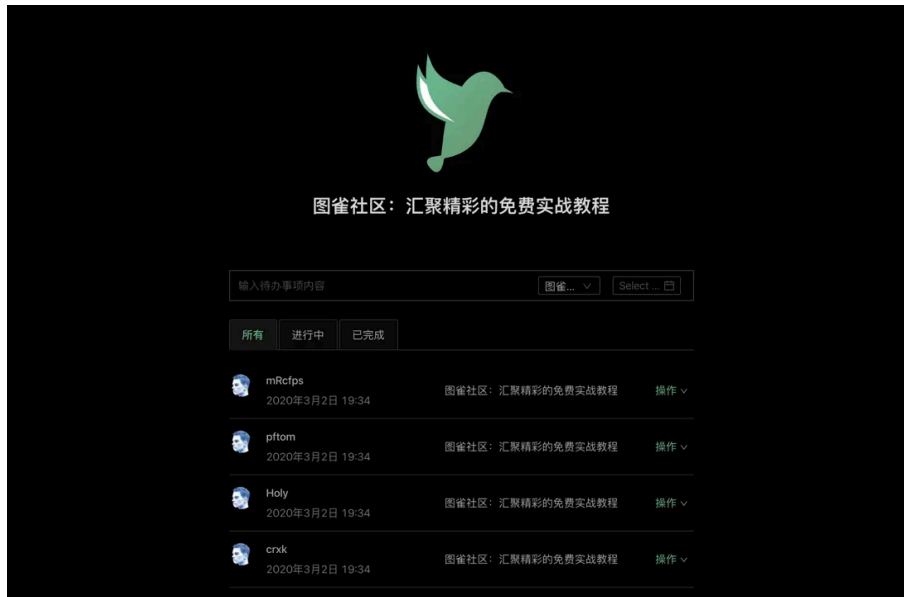
如果你偏爱 码云，那么你可以运行如下命令来获取初始代码：

```
git clone -b initial-code https://gitee.com/tuturu/typescript-tea
cd typescript-tea && npm install && npm start
```

如果你偏爱 Github, 那么你可以运行如下命令来获取初始代码:

```
git clone -b initial-code git@github.com:tutture-dev/typescript-tea
cd typescript-tea && npm install && npm start
```

通过上面的命令克隆初始代码之后, 然后把项目跑起来, 你应该可以看到如下效果:



Boom!!! 一个暗黑模式的 TodoList, 心动了嘛? 不管心不心动, 你都可以愉快的开始接下来的 TypeScript 学习了 🙌。

TypeScript 初探

正式 TS 时间 🍷, TS 是一门静态编程语言, 它是 JavaScript 的超集。首先我们先来解释一下什么是编程语言, 然后我们再来引出 TypeScript 是什么。

编程语言是什么?

那么什么是编程语言了? 编程语言是用来定义计算机程序的形式语言。它是一种被标准化的交流技巧, 用来向计算机发出指令。

我们拿 JS 来举例，一门标准的编程语言一般包含如下几个部分：

- 数据结构：如原始数据类型 string/number/void 等，非原始数据类型 array/object/enum 等
- 控制结构：如 if/else 、 switch 、 while、 for 循环等
- 组织结构：如 函数、类
- 特性：如 JS 的原型链
- 常用的 API：如 isNaN 判断是不是非数字，toFixed 将小数进行四舍五入操作
- 运行环境：如 浏览器端的 JavaScript、服务器端的 Node

其中前五种又称为语言内核，也就是我们常常喊的 ECMAScript 2015，或者 ES6；最后一个运行环境在浏览器端结合 BOM/DOM 即成为 JavaScript，在服务器端结合一些文件/网络的操作即成为 Node。

TypeScript 是什么？

而 TS，作为 JavaScript 的超集，包含着两类属性：

- 属于 JavaScript 端的编程语言特性，使得我们可以执行各种 JavaScript 相关的操作：变量声明、编写 if/else 控制流、使用循环处理重复任务、使用函数执行特定的任务块、使用类来组织和复用代码和模拟真实世界的操作等，还有新特性比如：装饰器、Iterator、Generator 这些。这类特性在此篇文章中，我们默认你已经很清楚了，不会做过多的讲解。
- 属于 TypeScript 端独有的特性：类型，它也具有一套编程语言的特性，比如标志一个变量是 string 类型，一个函数的参数有三个，它们的类型分别是 string/number/boolean，返回类型为 never 等，这是基础类型，我们甚至可以基于类型进行编程，使用类型版本的控制、组织结构来完成高级类型的编写，进而将类型附着在 JavaScript 对应的编程语言特性上，将 JS 静态化，使得我们可以在编译期间就能发现类型上的错误，这一特性是我们本篇文章的重点。

好的，读到这里，相比很多读者已经清楚了，其实 TS 没什么神秘的，主要就是设计了一套类似编程语言的类型语言，然后将这些类型附着在原 JavaScript 的语言之上，给其加上类型限制使得其静态化，进而可以快速的在编写时发现很多潜在的问题，帮助我们编写错误率更低，更适合团队协作的代码，这也是 TypeScript 适合编写大型的业务应用的原因。

类型语言之数据结构

其中 TS 数据结构又包含原始类型、非原始类型、特殊类型和高级类型等几类。我们将结合在 TS 类型侧的定义，以及附着在 JS 上进行实战来讲解。

原始类型

TS 类型侧的定义

和 JS 中的原始数据类型一样，TS 对应着一致的类型定义，包括下面八种：

- number
- string
- boolean
- null
- undefined
- void
- symbol
- bigint

提示

其中前六种是 ES5 中就有的，symbol 从 ES6 开始引入，bigint 是 ES2020 新引进的。

上面是 TS 的原始类型，我们之前提到 TS 就是将类型附着在 JS 上，将其类型化，那么我们来看看上面的原始类型如何附着在 JS 上，将其类型化。

附着在 JS 上的实战

TS 通过独特的**冒号语法**来将其类型侧定义的类型附着在 JS 上，我们来看几个例子：

用 JS 语言来写图雀社区的 Slogan，我们一般会这么写：

```
const tutureSlogan = '图雀社区，汇聚精彩的免费实战教程';
```

我们可以确定, 这句 Slogan 是一个 string 类型的, 所以我们用对应的 TS 类型附着在其变量定义上如下:

```
const tutureSlogan: string = '图雀社区, 汇聚精彩的免费实战教程';
```

这样我们就给原 JS 的 `tutureSlogan` 变量加上了类型定义, 它是一个 `string` 类型的变量, 通过这样的操作, 原 JS 变量的类型就被静态化了, 在初始化时, 就不能再赋值其他的类型给这个 `tutureSlogan` 变量了, 比如我们将 `number` 类型的字面量赋值给 `tutureSlogan`, 就会报错:

```
const tutureSlogan: string = 5201314 // 报错 Type '5201314' is not
```

这就是 TS 的强大之处, 当团队编码时事先约定好数据的类型, 那么后续编写并调用这些设置好类型的变量时就会强制起约束作用, 就像上面的代码一样, 如果给 `tutureSlogan` 赋值 `5201314` 就会报错, 其实你大可克制一点对吧 😊, 给 `5201314` 加个限制, 两边带上引号 `'5201314'` 问题就迎刃而解了, 爱也可以是克制 🤖。

提示

有些细心的同学可能对上面的报错信息有点不能理解, 对于报错信息的后半段类型 `string` 可能理解, 因为我们给 `tutureSlogan` 限制了 `string` 类型, 但是对于我们的赋值 `5201314`, 它原本是一个 JS 的 `number` 类型的字面量, 为什么也成了 `Type` 了?

那是因为, TS 引擎在对语句进行编译的时候, 会对变量赋值两端做一个类型推理, 比如对赋值语句的右侧 `5201314`, 会将其推理成 `5201314` 这个类型, 它是一个属于 `number` 类型的一个特殊的 `number` 类型, 可以被分配 (`assignable`) 给 `number` 类型的变量, 这里的 `assignable` 是可分配的意思, 就是一个子类型可以被分配给一个父类型, 比如数字 `1` 可以被分配给 `number` 数字类型, 但因为 `number` 类型和 `string` 类型是冲突的, 所以这里报错了。

这里读者可能会有感觉了就是, 你写的 JS 语句, 加上类型定义之后, 在 TS 编译器的世界里, 一切皆类型了, 它会以一种类型的视角去看待原 JS 语句, 比如上面的语句, 在 TS 编译器眼里, 就是 `5201314` 类型和 `string` 类型的一个比较过程, 如果比较一致, 那么好的, 我 TS 编译器今天就放你一马, 让你逍遥快活。

小结

我们上面说到了 TS 的原始类型，一共有八个之多，并且通过其中的 `string` 类型来讲解了如何将 TS 类型附着在原 JS 语法上以静态化 JS 语言，剩下的 7 个原始类型的用法和 `string` 类型类似，我们将在之后的讲解中逐渐用到其中的类型。

非原始类型

TS 类型侧的定义

同样的 JS 中的非原始数据类型一样，TS 中也存在非原始类型，表示出了八种原始类型之外的类型，非原始类型也称为是 `object` 类型。

实际上 TS 中还有几个常见的非原始类型，例举如下：

- `array`
- `tuple`
- `enum`

且因为它们属于 `object` 类型，所以 `object` 类型实际上就代表了非原始类型。在上面的三个类型以及其父类型 `object` 中，`array` 和 `object` 其实我们应该有点熟悉，至于 `tuple` 和 `enum` 则是 TS 中新增的类型，JS 中正式提案中目前是没有的。讲完了类型侧定义，我们马上来实践一下上面的 `array` 和 `enum` 非原始类型。

`array` 类型附着实战

其中 `array` 类型我们比较熟悉，但这里有个不同就是之前我们的 JS 因为是动态语言，所以一个数组里面可以有各种不同的数据类型项，比如我们看如下 JS 语句：

```
const arr = ['1', 2, '3'];
```

可以看到，从 TS 的角度去看这个数组变量 `arr` 所包含的类型，存在字符串类型 `'1'` 和 `'3'`，以及数字类型 `2`。但 TS 总的数组类型要求数组中的元素都是同一个类型，不允许动态变化，比如我们为上面的数组变量 `arr` 声明类型应该如下：


```
const arr: string[] = ['1', '2', '3'];
```

可以看到，我们给变量 `arr` 声明了 `string[]` 类型，即一个 `string` 类型后面跟着一个数组标志，表示是字符串数组类型，当声明了 `string[]` 类型之后，我们需要把之前的数组 `2` 改成字符串 `'2'`。

我们注意到 `array` 类型，它要求数组中每项的类型都一样，一般应用在数组的长度未知的情况，用特定的类型，比如 `string` 类型来约束数组的每一项。

然而从 JS 转过来的同学大多数同学可能对这个 `array` 类型不适应了，我们 JS 的同学经常会遇到编写一个数组，其中的多项的类型不一样，就和我们上面的 JS `arr` 的项一样，既有 `string` 类型又有 `number` 类型，那这该怎么办了？还好！TS 的设计者也为我们考虑到了这一点，那就是我们下面要讲到的 `tuple`（元组）类型。

tuple 类型附着实战

大家可能对 `tuple`（元组）类型很陌生了，其实元组是一种特殊的数组类型，它主要用于这样的场景：“一个数组的项数已知，其中每项的类型也已知”，这句话说起来可能比较绕，我们用上面讲数组的例子来讲元组：

```
const arr = ['1', 2, '3'];
```

我们知道上面的数组第一项和第三项的类型为 `string` 类型，第二项的类型为 `number` 类型，现在我们要给这个 `arr` 附着一个类型，使得其静态化。

这个条件满足我们上面说的元组的适用场景，我们通过给 `arr` 一个对应的元组类型，让我们可以保持上面的写法不变：

```
const arr: [string, number, string] = ['1', 2, '3'];
```

可以看到，元组就是形如 `[type1, type2, type3, ..., typen]` 这样数组

长度已知，且类型已知的情况，其中 `type1` 到 `typen` 中所有的类型都可以不一样。

小结

在这一小结中我们讲解了一下什么是非原始类型，然后说明了在 TS 中有四种非原始类型，其中有一种代表非原始类型 `object`，然后剩下的三种属于 `object` 类型。

接着我们通过实践讲解了 `array` 和 `tuple` 类型，对于 `enum` 类型和 `object` 类型本身，我们将留在之后的章节来讲，敬请期待 🙌。

特殊类型

TS 中还有几个常用的特殊类型，它们是 `any`、`unknown` 和 `never`，其中 `never` 类型一般会伴随着和函数的类型声明一起使用，所以我们将 `never` 类型的时候会提到函数的类型如何进行声明。

接下来我们来讲一讲这三个类型的含义和应用。

any 类型定义与实战

`any` 的字面含义是“任何”，主要用于在编码的时候不知道一个变量的类型，所以先给它加一个 `any` 类型定义，表示它可以是任何类型，一般留待后续确认此变量类型之后再将其 `any` 改为具体的类型。

我们来看一个例子，比如我们有下面一段 TS 变量定义语句：

```
let demand: any;
```

因为有时候产品给一个需求，要我们去开发一个新功能，给了设计稿，但是没交接清楚，对于设计稿有一些内容我们想提前做，但是因为不清楚具体的类型，比如这里的 `demand`，所以我们这里给 `demand` 一个 `any` 类型，然后继续做其他的内容，这样既不会出错，也不会影响其他的开发进度。

等到产品把具体的上下文交代清楚了，诶！我们清楚了知道这个 `demand` 的类型了，我们就可以回过头来给其附着一个严格的类型定义，比如我们知道它是 `string` 类型，那么我们再返回来对其修改如下：

```
let demand: string;
```

就是这样，`any` 的应用场景大多是这样的。但是玩 TS 的朋友要小心哦，不要一碰到不确定的就写个 `any` 类型，然后写了之后还不改，那就把 TS 用成了 AnyScript 了，这就和 JS 一样了 😊。所以你看呀，TS 的优秀之处在于，你完全可以在 TS 的环境中写 JS 还能享受 TS 带来的各种静态语言的优势，所以这么受欢迎也是可以理解滴。

unknown 类型定义与实战

`unknown` 类型和 `any` 都可以表示任何类型，应用场景也和上面类型，但是它更安全。那么具体安全在哪里了？我们通过一个例子来看一看：

```
let demandOne: any;  
let demandTwo: unknown;
```

我们拿到了开发需求，但是不清楚具体类型又打算继续开发时，上面两种情况都可以使用，但是当我们具体使用这两个变量的时候，`any` 类型的变量是可以进行任意进行赋值、实例化、函数执行等操作，但是 `unknown` 只允许赋值，不允许实例化、函数执行等操作，我们来看个例子：

```
demandOne = 'Hello, Tuture'; // 可以的  
demandTwo = 'Hello, Ant Design'; // 可以的  
  
demandOne.foo.bar() // 可以的  
demandTwo.foo.bar() // 报错
```

可以看到，`unknown` 类型只允许赋值操作，不允许对象取值（Getter）、函数执行等操作，所以它更安全。

never / 函数类型定义与实战

`never` 的字面意思是“永不”，在 TS 中代表不存在的值类型，一般用于给函数进行类型声明，函数绝不会有返回值的时候使用，比如函数内抛出错

误, 我们首先看个例子将讲解一下如何给函数进行类型声明, 然后接着我们讲 `never` 类型如何使用:

```
function responseError(message) {  
  // ... 具体操作, 接收信息, 抛出错误  
}
```

对于上面的函数, 我们可以使用箭头函数的形式把它抽象成为形如 `(args1, args2, ... , argsn) => returnValue`, 我们主要关注点在于函数的输入和输出, 所以我们在类型声明的时候把函数的输入参数的类型和输出结果的类型定义好就可以了。

我们注意到上面我们定义的函数有一个参数: `message`, 并且函数体内根据 `message` 抛出对应的错误, 那么我们来给它进行类型声明如下:

```
function responseError(message: string): never {  
  // ... 具体操作, 接收信息, 抛出错误  
}
```

可以看到我们同样使用了 TS 的冒号语法来进行函数参数和返回值的类型定义, 因为 `message` 一般是一个字符串 ID, 所以我们给它 `string` 类型, 而这个函数绝不会有返回值, 只是单纯的抛出错误, 所以我们给返回值一个 `never` 类型。

动手实践

基本了解了类型语言的数据结构之后, 我们马上来写一点 React 代码来实践我们学到的知识。

我们之前准备的代码中可以看到, 有两个假数据数组 `todoListData` 和 `userList`, 我们使用之前学到的知识来给这两个数组进行类型定义, 打开 `src/App.tsx` 对其中的内容作出对应的修改如下:

```
// ...  
interface Todo {  
  user: string;  
  time: string;  
  content: string;
```

```
isCompleted: boolean;
}

interface User {
  id: string;
  name: string;
  avatar: string;
}

const todoListData: Todo[] = [
  {
    content: "图雀社区: 汇聚精彩的免费实战教程",
    user: "mRcfps",
    time: "2020年3月2日 19:34",
    isCompleted: false
  },
  // ...
];

const userList: User[] = [
  ''
```

可以看到, 上面我们定义了两个 `interface` `Todo` 和 `User`, 然后以数组类型的方式对 `todoListData` 和 `userList` 进行注解, 表示 `todoListData` 是 `Todo[]` 类型, `userList` 是 `User` 类型。

这里的 `interface` 我们还没用提到, 我们将马上在后面讲到, 可以理解它类似 JS 中的对象, 用来组织一组类型, 就比如我们这里 `todoList` 中单个元素实际上是包含四个属性的对象, 其中前三个属性为 `string` 原始类型, 最后一个属性为 `boolean` 类型, 所以我们为了给 单个对象元素进行类型注解, 我们使用了 `interface`。

枚举和接口

在上一节中我们提到了 `interface`, 当时没有细讲, 这一节我们就先来细细说一下 `interface` 是什么?

Interface

它相当于类型中的 JS 对象, 用于对函数、类等进行结构类型检查, 所谓的结构类型检查, 就是两个类型的结构一样, 那么它们的类型就是兼容的, 这在计算机科学的世界里也被成为“鸭子类型”。

提示

什么鸭子类型?

当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子, 那么这只鸟就可以被称为鸭子。

我们马上来看一个例子了解一个 `Interface` 是怎么样的, 比如我们之前对象 `Todo`, 一个 `Todo` 对象如下:

```
const todo = {  
  content: '图雀社区, 汇聚精彩的免费技术教程';  
  user: 'mRcfps',  
  time: '2020年3月2日 19:34',  
  isCompleted: false,  
}
```

现在我们要这个 `todo` 做一个类型注解, 根据之前提到的“鸭子类型”的方式, 我们可以定义一个 `Interface` 来为它做注解:

```
interface Todo {  
  content: string;  
  user: string;  
  time: string;  
  isCompleted: boolean;  
}  
  
const todo: Todo = {  
  // ...  
}
```

可以看到我们的接口 `Todo` 内容有四个字段, 并且标注了这四个字段的类型, 比如 `content` 为 `string`, 这个接口的样子和 `todo` 对象是一样的, 所以用 `Interface Todo` 来注解 `todo` 是可行的, 用 VSCode 的同学, 应该可以看到我们这样写之后, 编辑器里面没有抛出异常。

可选属性

上面我们讲到 `Interface` 是用来注解 对象, 函数等, 那么我们就有一个场景, 一个对象里面的某些参数我们可能没有, 比如一个待办事项 `Todo`, 有时候没有设置 `time` 时间属性, 那么修饰这样一个对象我们该怎么办了? 幸好 TS 给我们提供了可选属性这样一个方便的属性, 使得我们可以

方便解决上面的问题，我们来看一下可选属性该怎么写，假如我们上面的那个例子，`time` 是可选的，那么我们可以写出如下这样：

```
interface Todo {  
  content: string;  
  user: string;  
  time?: string;  
  isCompleted: boolean;  
}
```

我们看到，只需要在属性类型修饰冒号左边加一个问号就可以了，这个时候我们就告诉 TS 编译器这个 `time` 属性是可选的一个类型，所以我们用上面的 Interface `Todo` 来注解一下没有 `time` 属性的 `todo` 对象如下：

```
const todo: Todo = {  
  content: '予力内容创作，加速技术传播',  
  user: 'pftom',  
  isCompleted: false,  
}
```

可以看到，使用 VSCode 来跟着教程敲的同学应该发现上面的内容没有错误，类型检查通过了。

只读属性

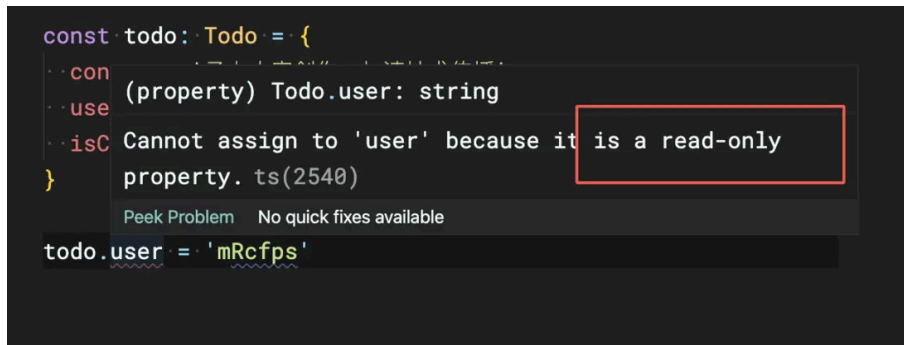
TS 的 Interface 还有一些额外的属性比如只读属性（readonly），表示用相关带有只读属性的接口对某个 JS 元素做类型注解的时候，这个 JS 元素相关的属性被注解为只读属性时，我们之后不可以修改这个属性了，我们来看一个例子：

```
interface Todo {  
  content: string;  
  readonly user: string;  
  time?: string;  
  isCompleted: boolean;  
}
```

可以看到只读属性的添加就是在属性之前加上 `readonly` 关键字，就可以将 Interface 中的属性标志为已读的，我们来试验一下这个只读效果：

```
const todo: Todo = {  
  content: '予力内容创作，加速技术传播',  
  user: 'pftom',  
  isCompleted: false,  
}  
  
todo.user = 'mRcfps'
```

当我们进行上面的修改操作之后，编辑器内会报错：



多余属性检查

我在在 JS 中经常会遇到一个对象，一开始我们知道它有是哪个属性，但是它的属性却可以动态增加，比如我们的 `todo` 可能还存在 `priority` 优先级这样一个属性，那么我们如何定义一个可以注解动态增加属性对象的 Interface 了？

所幸 TS 提供一个多余属性检查的写法，使得上面的问题我们也可以解决，我们来看下一个多余属性教程该怎么定义：

```
interface Todo {  
  isCompleted: boolean;  
  [propName: string]: any;  
}
```

使用类似上面 JS 中的动态属性赋值的方式我们就可为 `Todo` 接口加上多余属性检查，这里我们将其注解为一定拥有 `isCompleted` 属性，其他的属性可以动态添加，因为动态添加的属性的值类型我们不清楚，所以我们用 `any` 来表示值类型，它可以是任意类型。我们马上来试验一下：


```
const todo: Todo = {  
  content: '予力内容创作，加速技术传播',  
  isCompleted: false,  
}  
  
todo.user = 'pftom';  
todo.time = '2020-04-04';
```

可以看到，上面我们我们的 todo 在定义的时候只有两个属性，后面我们额外添加了两个属性，发现编辑器里面也不会报错，这就是多余属性检查的魅力。

Enum

枚举是 TS 中独有的概念，在 JS 中没有，主要用于帮助定义一系列命名常量，常用于给一类变量做类型注解，它们的值是一组值里面的某一个，比如我们应用中参与创建待办事项的用户只有五个人，那么在创建待办事项时，此事项的所属用户是五人中的某一人。

我们马上来看一个例子，我们的将这五个用户放到枚举里面：

```
enum UserId {  
  tuture,  
  mRcfps,  
  crxk,  
  pftom,  
  holy  
}
```

进而我们可以改进一下我们在上节 Interface 里面的 Todo 接口，给它的 user 字段一个更精确的类型注解：

```
interface Todo {  
  content: string;  
  user: UserId;  
  time: string;  
  isCompleted: boolean;  
}
```

通过上面的例子我们可以看到, `todo` 里面的 `user` 字段应该是五人之一, 它有可能是 `tuture`, 也有可能是 `mRcfps`, 我们不知道, 所以我们写了一个枚举 `UserId`, 并用它来注解 `Todo` 的 `user` 字段。

数字枚举

上面我们的 `UserId` 中几个枚举值其实都对应着相应的数字, 比如 `UserId.tuture` 它的值是数字 `0`, `UserId.mRcfps` 它的值是数字 `1`, 以此类推, 后面的几个枚举值分别是数字 `2`, `3`, `4`。

当然我们也可以手动给其中某个枚举值赋值一个数字, 这样这个枚举值后面的值会依次在这个赋值的数字上递增, 我们来看个例子:

```
enum UserId {  
  tuture,  
  mRcfps = 6,  
  crxk,  
  pftom,  
  holy,  
}
```

上面我们的每个枚举值对应的数字依次是: `0`, `6`, `7`, `8`, `9`

字符串枚举

枚举的值除了是数字还可以是一系列字符串, 比如:

```
enum UserId {  
  tuture = '66666666',  
  mRcfps = '23410977',  
  crxk = '25455350',  
  pftom = '23410976',  
  holy = '58352313',  
}
```

可以看到, 我们给每个枚举值赋值了对于的字符串。

异构枚举

当然在一个枚举里面既可以有字符串值也可以有数字：

```
enum UserId {  
  tuture = '66666666',  
  mRcfps = 6,  
}
```

动手实践

了解了 `Interface` 和 `Enum` 之后，我们马上运用在我们的项目中来完善我们的待办事项应用。

随着内容越写越多，我们的 `src/App.tsx` 越来越复杂，所以我们打算把 `TodoInput` 组件拆到单独的页面，在 `src` 目录下新建 `TodoInput.tsx`，并在里面编写如下的内容：

```
import React, { useState } from "react";  
import { Input, Select, DatePicker } from "antd";  
import { Moment } from "moment";  
  
import { userList } from "../utils/data";  
  
const { Option } = Select;  
  
enum UserId {  
  tuture = "666666666",  
  mRcfps = "23410977",  
  crxk = "25455350",  
  pftom = "23410976",  
  holy = "58352313"  
}  
  
export interface TodoValue {  
  content?: string;  
  user?: UserId;  
  date?: string;  
}  
  
interface TodoInputProps {  
  value?: TodoValue;  
  onChange?: (value: TodoValue) => void;
```

可以看到上面的内容，主要有如下几个部分的修改：

- 我们定义了新的 `Interface` : `TodoInputProps` , 它主要用来注解 `TodoInput` 这个函数式组件的 `props` 类型, 可看到这个接口主要有两个字段, 一个是 `value` , 它是 `TodoValue` 类型, 还有一个 `onChange` , 它是一个函数类型, 表示父组件将会传递一个 `onChange` 函数, 我们将在之后讲解 TS 怎么注解函数, 。
- 接着我们新增了一个枚举 `UserId` , 用来概括我们应用的五个用户的 ID, 并且人为的为这五个枚举常量赋了对应的值。
- 接着我们改进了定义了一个新 `TodoValue` 接口, 它有三个字段, 主要用于标志 `TodoInputProps` 中上层组件中可能传递下来的值, 所以三个字段都是可选的
- 最后我们定义了三个响应 `Input` 、 `Select` 、 `DatePicker` 的函数, `onContentChange` , `onUserChange` , `onDateOk` , 当上层组件没有传递对应的属性时, 使用 `setXXX` 来更新 React 状态, 否则触发 `triggerChange` , 调用父组件传递下来的 `onChange` 方法来更新对应的状态

提示

上面我们从 `./utils/data` 导入了 `userList` , 以及导入了 `Moment` 用来注解 `moment` 类型的 `date` , 我们将在接下来的来马上来创建对于的 `./utils/data` 文件以及安装对于的 `moment` 。

在 `src/TodoInput.tsx` 中我们导入了 `Moment` 用来注解 `onDateOk` 的函数参数 `date` , 接下来我们来安装它:

```
npm install moment
```

```
// ...  
"customize-cra": "^0.9.1",  
"less": "^3.11.1",  
"less-loader": "^5.0.0",  
"moment": "^2.24.0",  
"react": "^16.13.0",  
"react-app-rewired": "^2.1.5",  
"react-dom": "^16.13.0",  
// ...
```

接着我们来创建对应的 `src/utils/data.ts` 文件, 把之前在 `src/App.tsx` 里面的假数据统一放在这个文件里面, 然后导出:

```
interface Todo {
  user: string;
  time: string;
  content: string;
  isCompleted: boolean;
}

interface User {
  id: string;
  name: string;
  avatar: string;
}

export const todoListData: Todo[] = [
  {
    content: "图雀社区: 汇聚精彩的免费实战教程",
    user: "mRcfps",
    time: "2020年3月2日 19:34",
    isCompleted: false
  },
  {
    content: "图雀社区: 汇聚精彩的免费实战教程",
    user: "pftom",
    time: "2020年3月2日 19:34",
    isCompleted: false
  }
]
```

拆分了 `TodoInput` , 并把假数据移动到单独的文件之后, 我们需要修改 `src/App.tsx` 对应的部分如下:

```
import React, { useRef } from "react";

// ...中间一样

import TodoInput from "../TodoInput";

// ... 中间一样

import { todoListData } from "../utils/data";

const { Title } = Typography;
const { TabPane } = Tabs;

// 中间一样

// ... 删除 TodoInput 部分

// ... TodoList 保持原样

function App() {
```

```
const callback = () => {};  
  
const onFinish = (values: any) => {  
  console.log("Received values from form: ", values);  
};  
const ref = useRef(null);
```

可以看到，上面的内容主要做出了如下的修改：

- 我们删除了对应的假数据 `userList` 和 `todoListData` 及其 `Interface` ；
义 `Todo` 和 `User` ， 转而从我们创建的 `src/utils/data.ts` 里面导入 `todoListData`
- 接着我们删除了 `TodoInput` 组件， 转而导入我们之前创建的 `TodoInput` 组件
- 接着我们给 `Form` 表单部分加上了一个提交按钮， 以及扩展了 `onFinish` 函数
- 最后我们删除了一些不再需要的导包

小结

大功告成，这一节中我们学习了接口（Interface）和枚举（Enum），接口主要是对对象等多属性元素进行类型注解，而枚举是 TS 中独有的一个概念，在 JS 中没有，主要用于帮助定义一系列命名常量，常用于给一类变量做类型注解，它们的值是一组值里面的某一个，最后我们通过改进现有的 Todo 应用来实践了学到的这两个概念。

想要学习更多精彩的实战技术教程？来[图雀社区](#)逛逛吧。

本文所涉及的源代码都放在了 [Github](#) 或者 [Gitee](#) 上，如果您觉得我们写得还不错，希望您能给 ❤️ [这篇文章点赞Github 或 Gitee 仓库加星](#) ❤️ 哦~





typescript

react.js

antd

赞 5

收藏 3

分享

阅读 3.5k • 发布于 2020-04-06



一只图雀

863 声望 • 1.2k 粉丝

我们图雀社区是一个供大家分享用 Tuture 写作工具撰写教程的一个平台。在这里，读者们可以尽情享受高质量的实战教...

[关注作者](#)[« 上一篇](#)[下一篇 »](#)[类型即正义：TypeScript 从入门到... Taro 小程序开发大型实战（七）： ...](#)

引用和评论

被 2 篇内容引用



类型即正义：TypeScript 从入门到实践（三）：类型别名和类

2



类型即正义：TypeScript 从入门到实践（二）

推荐阅读



Taro 小程序开发大型实战（九）：使用 Authing 打造具有微信登录的企业级用户系统

一只图雀 • 赞 1 • 阅读 4.4k



文件导出

热饭班长 • 赞 8 • 阅读 3k



TS-react：react中常用的类型整理

wxp686 • 赞 1 • 阅读 7k



vscode的jsconfig.json和tsconfig.js

热饭班长 • 赞 3 • 阅读 4.5k



【从前端入门到全栈】Node.js之大文件分片上传

野生程序猿江辰 · 赞 3 · 阅读 312 · 评论 1



react组件解耦

热饭班长 · 赞 3 · 阅读 4k



react 踩坑

assassin_cike · 赞 1 · 阅读 3.5k

2 条评论

得票

最新



撰写评论 ...



提交评论

评论支持部分 Markdown 语法: ****粗体**** *_斜体_* [链接]
(<http://example.com>) `代码` - 列表 > 引用。你还可以使用 @
来通知其他用户。



木木: 最近正想看ts, 希望更完啊


👍 · 回复 · 2020-04-06

一只图雀 (作者): @木木 这个您可以放心, 我们会持续输出的, 我们社区的宗旨就是“予力内容创作, 加速技术传播”, 希望您可以持续关注我们, 谢谢?

👍 · 回复 · 2020-04-06

©2024 图雀社区

除特别声明外, 作品采用《署名-非商业性使用-禁止演绎 4.0 国际》进行许可

 使用 SegmentFault 发布

SegmentFault - 凝聚集体智慧, 推动技术进步

服务协议 · 隐私政策 · 浙ICP备15005796号-2 · 浙公网安备33010602002000号