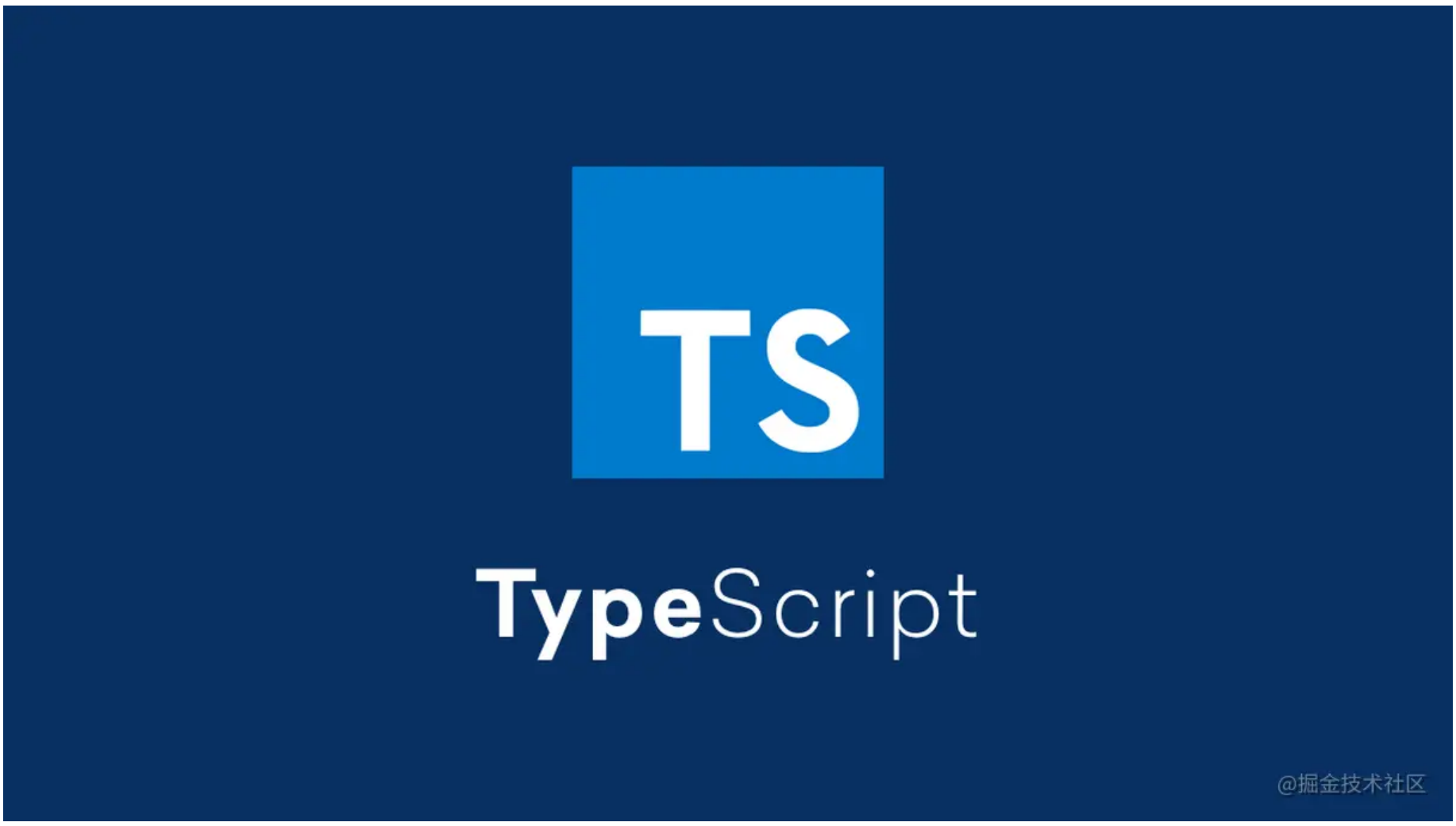
字节前端
2021年02月08日 阅读 34351

+ 关注



TypeScript 高级用法

字节前端 Lv3 北京字节跳动网络技术有限公司

> 本文主要介绍 TypeScript 的高级用法，适用于对 TypeScript 已经有所了解或者已经实际用过一段时间的同学，分别从类型、运算符、操作符、泛型的角度来系统介绍常见的 TypeScript 文章没有好好讲解的功能点，最后再分享一下自己的实践经历。

一、 类型

unknown

unknown 指的是**不可预先定义的类型**，在很多场景下，它可以替代 any 的功能同时保留静态检查的能力。

typescript 复制代码

```typescript
const num: number = 10;
(num as unknown as string).split('');   // 注意，这里和any一样完全可以通过静态检查
```

这个时候 unknown 的作用就跟 any 高度类似了，你可以把它转化成任何类型，不同的地方是，在静态编译的时候，unknown 不能调用任何方法，而 any 可以。

typescript 复制代码

```typescript
const foo: unknown = 'string';
foo.substr(1);          // Error: 静态检查不通过报错
const bar: any = 10;
any.substr(1);          // Pass: any类型相当于放弃了静态检查
```

unknown 的一个使用场景是，避免使用 any 作为函数的参数类型而导致的静态类型检查 bug：

稀土掘金 首页 探索稀土掘金 登录

typescript 复制代码

```typescript
function test(input: unknown): number {
  if (Array.isArray(input)) {
    return input.length;    // Pass: 这个代码块中，类型守卫已经将input识别为array类型
  }
  return input.length;      // Error: 这里的input还是unknown类型，静态检查报错。如果入参是any，则会放弃检查直接成功，带来报错风险
}
```

void

在 TS 中，void 和 undefined 功能高度类似，可以在逻辑上避免不小心使用了空指针导致的错误。

typescript 复制代码

```typescript
function foo() {}       // 这个空函数没有返回任何值，返回类型缺省为void
const a = foo();        // 此时a的类型定义为void，你也不能调用a的任何属性方法
```

void 和 undefined 类型最大的区别是，你可以理解为 undefined 是 void 的一个子集，当你对函数返回值并不在意时，使用 void 而不是 undefined。举一个 React 中的实际的例子。

typescript 复制代码

```typescript
// Parent.tsx
function Parent(): JSX.Element {
  const getValue = (): number => { return 2 };          /* 这里函数返回的是number类型 */
  // const getValue = (): string => { return 'str' };   /* 这里函数返回的string类型，同样可以传给子属性 */
  return <Child getValue={getValue} />
}
```

typescript 复制代码

```typescript
// Child.tsx
type Props = {
  getValue: () => void;  // 这里的void表示逻辑上不关注具体的返回值类型，number、string、undefined等都可以
}
function Child({ getValue }: Props) => <div>{getValue()}</div>
```

never

never 是指没法正常结束返回的类型，一个必定会报错或者死循环的函数会返回这样的类型。

typescript 复制代码

```typescript
function foo(): never { throw new Error('error message') }  // throw error 返回值是never
function foo(): never { while(true){} }  // 这个死循环的也会无法正常退出
function foo(): never { let count = 1; while(count){ count ++; } }  // Error: 这个无法将返回值定义为never，因为无法在静态编译阶段直接
```

还有就是永远没有相交的类型：

typescript 复制代码

```typescript
type human = 'boy' & 'girl' // 这两个单独的字符串类型并不可能相交，故human为never类型
```

不过任何类型联合上 never 类型，还是原来的类型：

typescript 复制代码

```typescript
type language = 'ts' | never   // language的类型还是'ts'类型
```

关于 never 有如下特性：

- 在一个函数中调用了返回 never 的函数后，之后的代码都会变成 deadcode

typescript 复制代码

```typescript
function test() {
  foo();                // 这里的foo指上面返回never的函数
  console.log(111);     // Error: 编译器报错，此行代码永远不会执行到
}
```

稀土掘金 首页 探索稀土掘金 登录

- 无法把其他类型赋给 never：

typescript 复制代码

```typescript
let n: never;
let o: any = {};
n = o;  // Error: 不能把一个非never类型赋值给never类型，包括any
```

关于 never 的这个特性有一些很 hack 的用法和讨论，比如这个知乎下的尤雨溪的回答。

二、运算符

非空断言运算符 !

这个运算符可以用在变量名或者函数名之后，用来强调对应的元素是非 null|undefined 的

typescript 复制代码

```typescript
function onClick(callback?: () => void) {
  callback!();          // 参数是可选入参，加了这个感叹号!之后，TS编译不报错
}
```

你可以查看编译后的 ES5 代码，居然没有做任何防空判断。

typescript 复制代码

```typescript
function onClick(callback) {
  callback();
}
```

这个符号的场景，特别适用于我们已经明确知道不会返回空值的场景，从而减少冗余的代码判断，如 React 的 Ref。

typescript 复制代码

```typescript
function Demo(): JSX.Elememt {
  const divRef = useRef<HTMLDivElement>();
  useEffect(() => {
    divRef.current!.scrollIntoView();    // 当组件Mount后才会触发useEffect，故current一定是有值的
  }, []);
  return <div ref={divRef}>Demo</div>
}
```

可选链运算符 ?.

相比上面!作用于编译阶段的非空判断， `?.` 这个是开发者最需要的运行时(当然编译时也有效)的非空判断。

typescript 复制代码

```typescript
obj?.prop    obj?.[index]    func?.(args)
```

?.用来判断左侧的表达式是否是 null | undefined，如果是则会停止表达式运行，可以减少我们大量的&&运算。

比如我们写出 `a?.b` 时，编译器会自动生成如下代码

typescript 复制代码

```typescript
a === null || a === void 0 ? void 0 : a.b;
```

这里涉及到一个小知识点: `undefined` 这个值在非严格模式下会被重新赋值，使用 `void 0` 必定返回真正的 undefined。

空值合并运算符 ??

??与||的功能是相似的，区别在于 **??在左侧表达式结果为 null 或者 undefined 时，才会返回右侧表达式** 。

比如我们书写了 `let b = a ?? 10` ，生成的代码如下：

稀土掘金 首页 探索稀土掘金 登录

而 || 表达式，大家知道的，则对 false、''、NaN、0 等逻辑空值也会生效，不适于我们做对参数的合并。

数字分隔符_

typescript 复制代码

```typescript
let num:number = 1_2_345.6_78_9
```

_可以用来对长数字做任意的分隔，主要设计是为了便于数字的阅读，编译出来的代码是没有下划线的，请放心食用。

三、操作符

键值获取 keyof

keyof 可以获取一个类型所有键值，返回一个联合类型，如下：

typescript 复制代码

```typescript
type Person = {
  name: string;
  age: number;
}
type PersonKey = keyof Person;  // PersonKey得到的类型为 'name' | 'age'
```

keyof 的一个典型用途是限制访问对象的 key 合法化，因为 any 做索引是不被接受的。

typescript 复制代码

```typescript
function getValue (p: Person, k: keyof Person) {
  return p[k];  // 如果k不如此定义，则无法以p[k]的代码格式通过编译
}
```

总结起来 keyof 的语法格式如下

复制代码

```
类型 = keyof 类型
```

实例类型获取 typeof

typeof 是获取一个对象/实例的类型，如下：

typescript 复制代码

```typescript
const me: Person = { name: 'gzx', age: 16 };
type P = typeof me;  // { name: string, age: number | undefined }
const you: typeof me = { name: 'mabaoguo', age: 69 }  // 可以通过编译
```

typeof 只能用在具体的对象上，这与 js 中的 typeof 是一致的，并且它会根据左侧值自动决定应该执行哪种行为。

typescript 复制代码

```typescript
const typestr = typeof me;   // typestr的值为"object"
```

typeof 可以和 keyof 一起使用(因为 typeof 是返回一个类型嘛)，如下：

typescript 复制代码

```typescript
type PersonKey = keyof typeof me;   // 'name' | 'age'
```

总结起来 typeof 的语法格式如下：

复制代码

```
类型 = typeof 实例对象
```

稀土掘金 首页 探索稀土掘金 登录

in 只能用在类型的定义中，可以对枚举类型进行遍历，如下：

typescript 复制代码

```typescript
// 这个类型可以将任何类型的键值转化成number类型
type TypeToNumber<T> = {
  [key in keyof T]: number
}
```

keyof 返回泛型 T 的所有键枚举类型， key 是自定义的任何变量名，中间用 in 链接，外围用 [] 包裹起来(这个是固定搭配)，冒号右侧 number 将所有的 key 定义为 number 类型。

于是可以这样使用了：

typescript 复制代码

```typescript
const obj: TypeToNumber<Person> = { name: 10, age: 10 }
```

总结起来 in 的语法格式如下：

复制代码

```
[ 自定义变量名 in 枚举类型 ]: 类型
```

四、泛型

泛型在 TS 中可以说是一个非常重要的属性，它承载了从静态定义到动态调用的桥梁，同时也是 TS 对自己类型定义的元编程。泛型可以说是 TS 类型工具的精髓所在，也是整个 TS 最难学习的部分，这里专门分两章总结一下。

基本使用

泛型可以用在普通类型定义，类定义、函数定义上，如下：

typescript 复制代码

```typescript
// 普通类型定义
type Dog<T> = { name: string, type: T }
// 普通类型使用
const dog: Dog<number> = { name: 'ww', type: 20 }

// 类定义
class Cat<T> {
  private type: T;
  constructor(type: T) { this.type = type; }
}
// 类使用
const cat: Cat<number> = new Cat<number>(20); // 或简写 const cat = new Cat(20)

// 函数定义
function swipe<T, U>(value: [T, U]): [U, T] {
  return [value[1], value[0]];
}
// 函数使用
swipe<Cat<number>, Dog<number>>([cat, dog])  // 或简写 swipe([cat, dog])
```

注意，如果对一个类型名定义了泛型，那么使用此类型名的时候一定要把泛型类型也写上去。

而对于变量来说，它的类型可以在调用时推断出来的话，就可以省略泛型书写。

泛型的语法格式简单总结如下：

复制代码

```
类型名<泛型列表> 具体类型定义
```

泛型推导与默认值

稀土掘金 首页 探索稀土掘金 登录

上面提到了，我们可以简化对泛型类型定义的书写，因为TS会自动根据变量定义时的类型推导出变量类型，这一般是发生在函数调用的场合的。

复制代码

```
type Dog<T> = { name: string, type: T }

function adopt<T>(dog: Dog<T>) { return dog };

const dog = { name: 'ww', type: 'hsq' };  // 这里按照Dog类型的定义一个type为string的对象
adopt(dog);  // Pass: 函数会根据入参类型推断出type为string
```

若不适用函数泛型推导，我们若需要定义变量类型则必须指定泛型类型。

复制代码

```
const dog: Dog<string> = { name: 'ww', type: 'hsq' }  // 不可省略<string>这部分
```

如果我们想不指定，可以使用泛型默认值的方案。

复制代码

```
type Dog<T = any> = { name: string, type: T }
const dog: Dog = { name: 'ww', type: 'hsq' }
dog.type = 123;    // 不过这样type类型就是any了，无法自动推导出来，失去了泛型的意义
```

泛型默认值的语法格式简单总结如下：

复制代码

```
泛型名 = 默认类型
```

泛型约束

有的时候，我们可以不用关注泛型具体的类型，如：

typescript 复制代码

```typescript
function fill<T>(length: number, value: T): T[] {
  return new Array(length).fill(value);
}
```

这个函数接受一个长度参数和默认值，结果就是生成使用默认值填充好对应个数的数组。我们不用对传入的参数做判断，直接填充就行了，但是有时候，我们需要限定类型，这时候使用 `extends` 关键字即可。

typescript 复制代码

```typescript
function sum<T extends number>(value: T[]): number {
  let count = 0;
  value.forEach(v => count += v);
  return count;
}
```

这样你就可以以 `sum([1,2,3])` 这种方式调用求和函数，而像 `sum(['1', '2'])` 这种是无法通过编译的。

泛型约束也可以用在多个泛型参数的情况。

typescript 复制代码

```typescript
function pick<T, U extends keyof T>(){};
```

这里的意思是限制了 U 一定是 T 的 key 类型中的子集，这种用法常常出现在一些泛型工具库中。

extends 的语法格式简单总结如下，注意下面的类型既可以是一般意义上的类型也可以是泛型。

复制代码

```
泛型名 extends 类型
```

稀土掘金 首页 探索稀土掘金 登录

上面提到 extends，其实也可以当做一个三元运算符，如下：

```typescript
T extends U? X: Y
```
typescript 复制代码

这里便不限制 T 一定要是 U 的子类型，如果是 U 子类型，则将 T 定义为 X 类型，否则定义为 Y 类型。

注意，生成的结果是**分配式的**。

60

举个例子，如果我们把 X 换成 T，如此形式： `T extends U? T: never` 。

此时返回的 T，是满足原来的 T 中包含 U 的部分，可以理解为 T 和 U 的**交集**。

所以，extends 的语法格式可以扩展为

```
泛型名A extends 类型B ? 类型C: 类型D
```
复制代码

泛型推断 infer

infer 的中文是“推断”的意思，一般是搭配上面的泛型条件语句使用的，所谓推断，就是你不用预先指定在泛型列表中，在运行时会自动判断，不过你得先预定义好整体的结构。举个例子

```typescript
type Foo<T> = T extends {t: infer Test} ? Test: string
```
typescript 复制代码

首选看 extends 后面的内容， `{t: infer Test}` 可以看成是一个包含 `t属性` 的**类型定义**，这个 `t属性` 的 value 类型通过 `infer` 进行推断后会赋值给 `Test` 类型，如果泛型实际参数符合 `{t: infer Test}` 的定义那么返回的就是 `Test` 类型，否则默认给缺省的 `string` 类型。

举个例子加深下理解：

```
type One = Foo<number>  // string，因为number不是一个包含t的对象类型
type Two = Foo<{t: boolean}>  // boolean，因为泛型参数匹配上了，使用了infer对应的type
type Three = Foo<{a: number, t: () => void}> // () => void，泛型定义是参数的子集，同样适配
```
复制代码

`infer` 用来对满足的泛型类型进行子类型的抽取，有很多高级的泛型工具也巧妙的使用了这个方法。

五、泛型工具

Partial<T>

此工具的作用就是将泛型中全部属性变为可选的。

```typescript
type Partial<T> = {
        [P in keyof T]?: T[P]
}
```
typescript 复制代码

举个例子，这个类型定义在下面也会用到。

```typescript
type Animal = {
  name: string,
  category: string,
  age: number,
  eat: () => number
}
```
typescript 复制代码

稀土掘金 首页 探索稀土掘金 登录

typescript 复制代码

```typescript
type PartOfAnimal = Partial<Animal>;
const ww: PartOfAnimal = { name: 'ww' }; // 属性全部可选后，可以只赋值部分属性了
```

Record<K, T>

此工具的作用是将 K 中所有属性值转化为 T 类型，我们常用它来申明一个普通 object 对象。

typescript 复制代码

```typescript
type Record<K extends keyof any,T> = {
  [key in K]: T
}
```

这里特别说明一下， `keyof any` 对应的类型为 `number | string | symbol` ，也就是可以做对象键(专业说法叫索引 index)的类型集合。

举个例子：

typescript 复制代码

```typescript
const obj: Record<string, string> = { 'name': 'zhangsan', 'tag': '打工人' }
```

Pick<T, K>

此工具的作用是将 T 类型中的 K 键列表提取出来，生成新的子键值对类型。

typescript 复制代码

```typescript
type Pick<T, K extends keyof T> = {
  [P in K]: T[P]
}
```

我们还是用上面的 `Animal` 定义，看一下 Pick 如何使用。

typescript 复制代码

```typescript
const bird: Pick<Animal, "name" | "age"> = { name: 'bird', age: 1 }
```

Exclude<T, U>

此工具是在 T 类型中，去除 T 类型和 U 类型的交集，返回剩余的部分。

typescript 复制代码

```typescript
type Exclude<T, U> = T extends U ? never : T
```

注意这里的 extends 返回的 T 是原来的 T 中和 U 无交集的属性，而任何属性联合 never 都是自身，具体可在上文查阅。

举个例子

typescript 复制代码

```typescript
type T1 = Exclude<"a" | "b" | "c", "a" | "b">;   // "c"
type T2 = Exclude<string | number | (() => void), Function>; // string | number
```

Omit<T, K>

此工具可认为是适用于键值对对象的 Exclude，它会去除类型 T 中包含 K 的键值对。

typescript 复制代码

```typescript
type Omit = Pick<T, Exclude<keyof T, K>>
```

在定义中，第一步先从 T 的 key 中去掉与 K 重叠的 key，接着使用 Pick 把 T 类型和剩余的 key 组合起来即可。

稀土掘金 首页 探索稀土掘金 登录

```typescript
const OmitAnimal:Omit<Animal, 'name'|'age'> = { category: 'lion', eat: () => { console.log('eat') } }
```

可以发现，Omit 与 Pick 得到的结果完全相反，一个是取非结果，一个取交结果。

ReturnType<T>

此工具就是获取 T 类型(函数)对应的返回值类型：

```typescript
type ReturnType<T extends (...args: any) => any>
  = T extends (...args: any) => infer R ? R : any;
```

看源码其实有点多，其实可以稍微简化成下面的样子：

```typescript
type ReturnType<T extends func> = T extends () => infer R ? R: any;
```

通过使用 infer 推断返回值类型，然后返回此类型，如果你彻底理解了 infer 的含义，那这段就很好理解。

举个例子：

```
function foo(x: string | number): string | number { /*..*/ }
type FooType = ReturnType<foo>;  // string | number
```

Required<T>

此工具可以将类型 T 中所有的属性变为必选项。

```typescript
type Required<T> = {
  [P in keyof T]-?: T[P]
}
```

这里有一个很有意思的语法 `-?` ，你可以理解为就是 TS 中把?可选属性减去的意思。

除了这些以外，还有很多的内置的类型工具，可以参考TypeScript Handbook获得更详细的信息，同时 Github 上也有很多第三方类型辅助工具，如utility-types等。

六、项目实战

这里分享一些我个人的想法，可能也许会比较片面甚至错误，欢迎大家积极留言讨论

Q: 偏好使用 interface 还是 type 来定义类型?

A: 从用法上来说两者本质上没有区别，大家使用 React 项目做业务开发的话，主要就是用来定义 Props 以及接口数据类型。

但是从扩展的角度来说，type 比 interface 更方便拓展一些，假如有以下两个定义：

```
type Name = { name: string };
interface IName { name: string };
```

想要做类型的扩展的话，type 只需要一个 `&` ，而 interface 要多写不少代码。

```
type Person = Name & { age: number };
interface IPerson extends IName { age: number };
```

稀土掘金 首页 探索稀土掘金 登录

另外 type 有一些 interface 做不到的事情，比如使用 `|` 进行枚举类型的组合，使用 `typeof` 获取定义的类型等等。

不过 interface 有一个比较强大的地方就是可以重复定义添加属性，比如我们需要给 `window` 对象添加一个自定义的属性或者方法，那么我们直接基于其 Interface 新增属性就可以了。

复制代码

```
declare global {
    interface Window { MyNamespace: any; }
}
```

60

总体来说，大家知道 TS 是类型兼容而不是类型名称匹配的，所以一般不需用面向对象的场景或者不需要修改全局类型的场合，我一般都是用 type 来定义类型。

Q: 是否允许 any 类型的出现

A: 说实话，刚开始使用 TS 的时候还是挺喜欢用 any 的，毕竟大家都是从 JS 过渡过来的，对这种影响效率的代码开发方式并不能完全接受，因此不管是出于偷懒还是找不到合适定义的情况，使用 any 的情况都比较多。

随着使用时间的增加和对 TS 学习理解的加深，逐步离不开了 TS 带来的类型定义红利，不希望代码中出现 any，所有类型都必须要一个一个找到对应的定义，甚至已经丧失了裸写 JS 的勇气。

这是一个目前没有正确答案的问题，总是要在效率和时间等等因素中找一个最适合自己的平衡。不过我还是推荐使用 TS，随着前端工程化演进和地位的提高，强类型语言一定是多人协作和代码健壮最可靠的保障之一，多用 TS，少用 any，也是前端界的一个普遍共识。

Q: 类型定义文件(.d.ts)如何放置

A: 这个好像业界也没有特别统一的规范，我的想法如下：

- 临时的类型，直接在使用时定义

如自己写了一个组件内部的 Helper，函数的入参和出参只供内部使用也不存在复用的可能，可以直接在定义函数的时候就在后面定义。

typescript 复制代码

```
function format(input: {k: string}[]): number[] { /***/ }
```

- 组件个性化类型，直接定义在 ts(x)文件中

如 AntD 组件设计，每个单独组件的 Props、State 等专门定义了类型并 export 出去。

typescript 复制代码

```
// Table.tsx
export type TableProps = { /***/ }
export type ColumnProps = { /***/ }
export default function Table() { /***/ }
```

这样使用者如果需要这些类型可以通过 import type 的方式引入来使用。

- 范围/全局数据，定义在.d.ts 文件中

全局类型数据，这个大家毫无异议，一般根目录下有个 typings 文件夹，里面会存放一些全局类型定义。

假如我们使用了 css module，那么我们需要让 TS 识别.less 文件(或者.scss)引入后是一个对象，可以如此定义：

typescript 复制代码

```
declare module '*.less' {
  const resource: { [key: string]: string };
  export = resource;
}
```

而对于一些全局的数据类型，如后端返回的通用的数据类型，我也习惯将其放在 typings 文件夹下，使用 Namespace 的方式来避免名字

typescript 复制代码

```typescript
declare namespace EdgeApi {
  interface Department {
    description: string;
    gmt_create: string;
    gmt_modify: string;
    id: number;
    name: string;
  }
}
```

这样，每次使用的时候，只需要 `const department: EdgeApi.Department` 即可，节省了不少导入的精力。开发者只要能约定规范，避免命名冲突即可。

关于 TS 用法的总结就介绍到这里，感谢大家的观看~

欢迎关注「 字节前端 ByteFE 」

简历投递联系邮箱「 tech@bytedance.com 」

文章分类 前端 文章标签 TypeScript 前端

字节前端 Lv5 北京字节跳动网络技术有限公司
发布了 153 篇文章 · 获得点赞 7,315 · 获得阅读 406,954

关注

安装掘金浏览器插件

多内容聚合浏览、多引擎快捷搜索、多工具便捷提效、多模式随心畅享，你想要的，这里都有！

前往安装

输入评论（Enter换行，⌘ + Enter发送）

发表评论

热门评论

毅力的小蚂蚁 Lv1 前端工程师 @ 猫眼电影 9月前

没有入门的可以看这个：hejialianghe.github.io

1 6

DDUDU 9月前

感谢 这个全栈体系有不少好东西哈哈

点赞 回复

毅力的小蚂蚁 Lv1 回复 **DDUDU** 9月前

最新地址：hejialianghe.gitee.io

“感谢 这个全栈体系有不少好东西哈哈”

点赞 回复

查看更多回复

稀土掘金 首页 探索稀土掘金 登录

所有类型都必须要一个一个找到对应的定义，甚至已经丧失了裸写 JS 的勇气。 哈哈哈，我也有这种感觉。看到某些 interface 里面用 any 定义的就浑身难受，甚至去找后端要过他们接口返回数据的类型定义来同步到前端。😂

2 3

踏月流星 Lv1 9月前

👍

点赞 回复

迷城梦境 Lv1 9月前

建议直接swagger转.ds

2 回复

查看更多回复

全部评论（45）

最新 最热

幻灵尔依 Lv4 临时工 @ BUG搬运公司 2月前

keyof返回泛型 T 的所有键枚举类型 是联合类型

点赞 回复

山下有风 Lv1 2月前

请教一个问题： type Other = Exclude let a: Other = 'foo' 好像并没有报错，预期Other应该是除了‘foo‘的所有字符串 请问有什么方法可以获取?

点赞 回复

FFF1 Lv2 前端开发工程师 4月前

这是我阅读ts看到最好的文章

点赞 回复

小李小李不讲道理 Lv1 前端开发 @ 无 6月前

ReturnType 的使用好像不对～少个 typeof

点赞 回复

毓逍遥 7月前

?? 检查 undefined | null , 而 ?. 是检查所有空值

点赞 回复

Gavin__ Lv1 前端 7月前

👍看完，点个赞

点赞 回复

EricLiam 前端 @ 字节跳动 8月前

技术学院看一遍这边再看一遍 😘

点赞 回复

Mr_Carl Lv1 首席牧码人🐴 @ 千里码... 8月前

赞一个 👍

点赞 回复

菜鸡不会吃肉 8月前

ReturnType中应该事typeof foo吧

点赞 回复

点赞 回复

许愿瓶 摸鱼 @ 才华有限公司 8月前

强!

点赞 回复

顿顿 学生 8月前

partical? partial?

点赞 回复

lawler61 Lv2 字节跳动 9月前

同理还有UnPromise 跟ReturnType差不多 extends配合infer实现

点赞 回复

commit989264 前端工程师 @ 小米 9月前

下次遇到不想写ts的同事就把这篇文章甩给他

点赞 1

铁皮饭盒 Lv4 9月前

你应该把any甩给他 😄

点赞 回复

czzonet Lv2 js/ts全栈 9月前

太全了 自己总有一些遗漏不使用

点赞 回复

洛亦仙 9月前

能改过，则天地不怒。能安分，则鬼神无权。——《格言联璧·持躬类》

点赞 回复

想改个名字 Lv1 9月前

突然觉得自己TS写的很高级

1 回复

MiyueFE Lv2 公众号：前端小白Miyu... 9月前

unknown第二个代码块例子有笔误

点赞 回复

一起走到天亮 前端研发工程师 9月前

手动点赞

点赞 回复

u96460 Lv2 公众号 @ 前端收藏家 9月前

type Partial = { [key in keyof T]?: T[P] } 这里拼错了，应该是[P in keyof T]?: T[P]

点赞 回复

查看全部 45 条回复

相关推荐

KnowsCount 6月前 TypeScript

为什么我不用 Typescript

说下我个人感觉 Typescript 的缺点、为何它的优点无法打动我用它替代 Javascript，以及跟推荐我使用 Typescript 的大家讲一下我...

7783 22 105

俊劫 4月前 前端 TypeScript JavaScript

越来越多的项目都在支持 TypeScript，越来成为前端必备能力之一。特性虽然很多，但是业务开发用到的并不多，够用就行。这...

2.0w 513 39

Jimmy_kiwi 1月前 TypeScript 前端

2021 typescript史上最强学习入门文章(2w字)

前言 这篇文章前前后后写了几个月,之前总是零零散散的学习,然后断断续续的写文章(懒),自己参考了很多文章以及看了一些ts视频,然后把基础的知识点全部总...

1.2w 480 47

政采云前端团队 3月前 前端 架构 CI/CD

如何搭建适合自己团队的构建部署平台

本文已参与好文召集令活动，点击查看：后端、大前端双赛道投稿，2万元奖池等你挑战！ 前端业界现有的构建部署方案，常用的应该...

1.2w 217 26

天明夜尽 10月前 TypeScript JavaScript

TypeScript 中提升幸福感的 10 个高级技巧

用了一年时间的 TypeScript 了，项目中用到的技术是 Vue + TypeScript 的，深感中大型项目中 TypeScript 的必要性，特别是生命周...

7615 161 18

前沿技术瞭望官 8月前 前端 TypeScript 资讯

为什么我对 TypeScript 黑转粉？一个 JS 开发者的深情自白

在这篇博客文章中，我将会讲述我是如何从一名 TypeScript 黑粉的开发者转变到如今不想回到原生 JavaScript 世界的开发者的旅程...

4841 40 22

蛙人 2月前 前端 TypeScript JavaScript

连夜爆肝只为将它送到你的面前，写给初级前端快速转TypeScript指南

哈喽，今天给大家带来的是TypeScript教程，希望看完本文你有不一样的收获，谢谢支持，欢迎来踩。

1.6w 587 47

LeonVincent 1年前 TypeScript

TypeScript经常用到的高级类型

ts入门了一段时间，可是碰上一些高级类型总是一头雾水，还需要去看文档，所以总结了一些工作中经常用到的一些高级类型。所以阅...

2844 18 5

前沿技术瞭望官 8月前 前端 TypeScript 资讯

新发布的 TypeScript 手册！先睹为快！

在过去的一年里，团队高度重视提升 TypeScript 文档的规模、时效性和范围。TypeScript 手册是我们的文档中最关键的部分，它是...

4912 71 9

豆皮范儿 5月前 TypeScript

TypeScript 高级用法

hi，豆皮粉儿们，今天又和大家见面了，本期分享的是由bytedancer“米兰的小铁匠”， 带来的TypeScript高级使用， 适用于对TypeScript已经有所了解或者已...

1421 19 评论

DevUI团队 4天前 前端 TypeScript Angular.js

DevUI开源的故事

以下是DevUI开源的故事，这个故事可能会很短，因为DevUI品牌从初创到现在也只有5年，开源的时间只有短短2年，但DevUI与社区...

9276 53 114

阿宝哥 1年前 前端 TypeScript

细数这些年被困扰过的 TS 问题

TypeScript 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，而且本质上向这个语言添加了可选的静态类型...

9621 311 28

稀土掘金 首页 探索稀土掘金 登录

TypeScript体系调研报告

TypeScript = Type + Script（标准JS）。我们从TS的官方网站上就能看到定义：TypeScript is a typed superset of JavaScript that...

4.7w 731 22

暖生 4月前 前端 TypeScript

TypeScript 高级类型及用法

本文详细介绍了 TypeScript 高级类型的使用场景，对日常 TypeScript 的使用可以提供一些帮助。

3761 109 13

前沿技术瞭望官 8月前 前端 掘金翻译计划 deno

Deno 将停用 TypeScript 的五个原因

最近有一份流传的文档，说是 Deno 将停止在其内部代码中使用 TypeScript。文档中提到了当前开发环境的几个问题，包括了...

3.8w 181 157

淘系前端团队 2月前 前端

程序员最重要的能力是什么?

我们邀请了 4 名淘系技术的工程师，给大家分享一些他们认为最重要的能力，希望能够为你提供一份参考。 01 - 淘系技术部 - 繁易 对写代码始终充满兴趣，这...

2472 27 6

---

yuxiaoliang 11月前 JavaScript TypeScript

Typescript代码整洁之道

最近半年陆续交接了几位同事的代码,发现虽然用了严格的eslint来规范代码的书写方式，同时项目也全量使用了Typescript,但是在review代码的过程中,还是有...

8654 253 25

晓前端 12月前 TypeScript

进来看看，TypeScript居然还能这么玩

keyof T 拿到 T 所有属性名, 然后 in 进行遍历, 将值赋给 P, 最后 T[P] 取得相应属性的值. 当然这也只能一层 如上面Partial例子来看jack.person.name 是可以直...

4167 79 1

乘风gg 2年前 React.js TypeScript

可能是你需要的 React + TypeScript 50 条规范和经验

1. 注释 2. 引用组件顺序 3. 引号 4. 缩进 5. 分号 除了代码块的以外的每个表达式后必须加分号。 6. 括号 下列关键字后必须有大括号...

6.6w 1154 90

观察者风过 3年前 JavaScript 前端 开源

开发自己的前端工具库(二)：函数式编程

本系列文章将通过自己的一个开发工具库的实战经验(踩过的坑)教大家如何开发属于自己的一个工具库，在这里你可以学到Git的使用规范，基础项目的搭建，...

3486 124 3
