

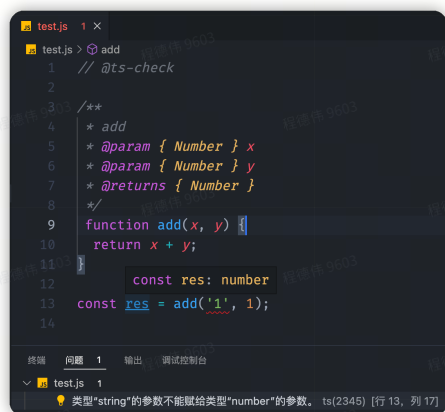
抛砖引玉：TypeScript 从入门到实践

介绍

众所周知 `JavaScript` 是一门弱类型语言，在前端进入工程化后，代码仓库越来越大，`JavaScript` 弱类型的缺点被无限放大，使其难以胜任开发大型项目。在一个多人维护的项目中，往往不知道别人写的函数是什么意思、需要传入什么参数、返回值是什么，一个用法不小心就会导致线上出现 `BUG`，所以除了靠口口相传以外还要维护大量的代码注释或者接口文档来提供其他人了解。但是当我们使用 `TypeScript` 后，除了初期具有一定的学习成本以外，基本上可以很好的解决上述的问题。

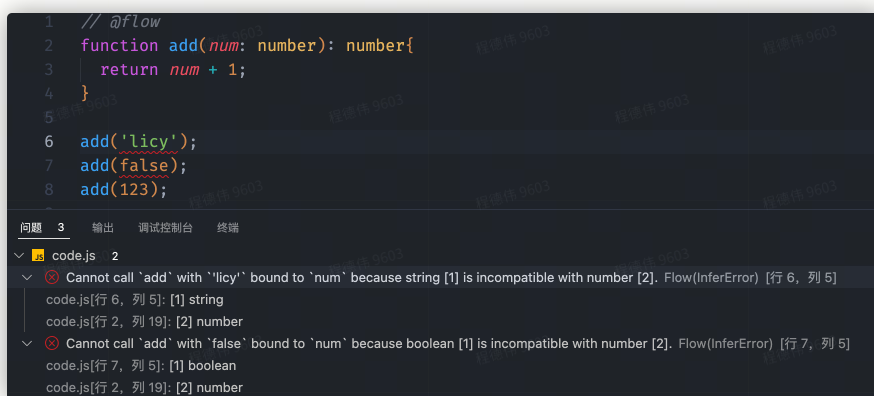
`TypeScript` 是 `JavaScript` 的严格超集，这意味着任何合法的 `JavaScript` 代码在 `TypeScript` 中都是合法的。`TS` 的作者是 [安德斯·海尔斯伯格](#)，2012年10月，微软发布了首个公开版本的 `TypeScript`，2013年6月19日正式发布了正式版 `TypeScript`。根据 [Roadmap](#) 我们可以知道 `TS` 官方每隔三个月会更新一个正式的 `minor` 版本，比如会在今年的 5.27 发布了 `v4.7`。更多关于 `TS` 的故事可以查看 `TypeScript` 团队成员 `orta` 的文章：[Understanding TypeScript's Popularity](#)

当然在 `TS` 彻底大火之前，同期也存在很多相似的工具来辅助开发者做好类型提示，比如 `Flow`、`JSDoc` 等。



```
1 // @ts-check
2
3 /**
4  * add
5  * @param { Number } x
6  * @param { Number } y
7  * @returns { Number }
8  */
9 function add(x, y) {
10   return x + y;
11 }
12
13 const res: number
14 const res = add('1', 1);
```

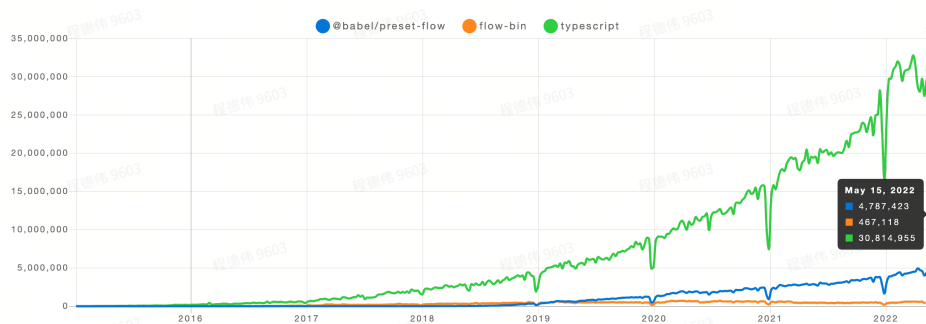
`JSDoc`，通过注释的方式给 `add` 函数添加类参数类型和返回值的类型。通过在编辑器顶部添加 `@ts-check` 的注释开始 `VSCode` 对其的检查。



```
1 // @flow
2 function add(num: number): number{
3   return num + 1;
4 }
5
6 add('licy');
7 add(false);
8 add(123);
```

`Flow`，类似 `TS` 的写法添加类型注解，可以通过安装插件或者命令扫描出当前代码中有问题的地方。

根据 `npm` 的现在趋势，目前 `Flow` 的使用率比 `TS` 低了很多。



经过众多的开发者选择，目前来看 TS 已经是完全胜出了。许多著名的开源库都采用 TS 进行编写。比如 VueJS，其中 v2.x 版本是使用的 Flow 进行的类型编写，但是 v3.x 版本已经全部迁移到了 TS。至于为什么选择 TS 替代 Flow 可以参看 尤玉溪的回答。

所以说 TS 逐渐的得到了社区的认可，就像 HTML、CSS、JS 一样快成为一名前端开发者必备的技能。所以我们不得不去学习它、并将它灵活的运用在项目当中。

基础使用

1. 基本类型的介绍与适用场景
2. 交叉和联合类型
3. 类型检查机制：推断、断言、保护和守卫
4. 全局类型、类型引入、类型重写、类型合并（interface 和 type 的区别）

基础类型

由于 TS 是 JS 的严格超集，所以 JS 中支持的类型在 TS 中肯定支持，所以在 JS 代码中使用什么类型的变量，在 TS 中也使用该类型。

TS 具有类型自动推断的能力，比如当我们使用 const 或者 let 声明一个变量的时，如果直接有赋值，那么就会给当前变量设置成这个赋值的类型。如下：

```
1 let str = 'hello'; // let str: string;
2 let num = 666; // let num: number;
3 let bool = true; // let bool: boolean
4 let undef = undefined; // let undef: undefined
5 let nul = null; // let nul: null
6 let sym = Symbol(123); // let sym: symbol
7 const fn = () => 123; // const fn: () => number
8 let res = fn(); // let res: number
9 let arr = [1, 2, 'abc']; // let a: (string | number)[]
10 const obj = {
11   a: 1,
12   b: true,
13 }
14 /* const obj: {
```

```

15     a: number;
16     b: boolean;
17 }*/
18

```

在有些同学的尝试过程中，当赋值为 `undefined` 和 `null` 时，会自动推断成 `any`。这是由于 `tsconfig.json` 中的配置有问题，把 `compilerOptions.strictNullChecks` 设置为 `true` 即可。目前发现团队中有部分项目都没有开启该配置，那么可能会在运行时出现意料之外的 `BUG`。

新增的类型

当然为了代码的灵活编写 `TS` 还具有一些独特的类型：

类型	描述
元组	<p>元组类型允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。</p> <p>比如 <code>useState</code> 的返回值就是一个元组类型。数组的第一项是一个变量的类型，数组的第二项是一个函数。这就是同一个数组各个元素的类型不同。</p> <pre> 1 let tuple: [string, number]; 2 tuple = ['hello', 10]; // OK 3 tuple = [10, 'hello']; // Error </pre>
枚举	<p>和其他语言一样，枚举可以看成是一类相同功能常量的集合，枚举既可以当成类型使用，也可以当成值来使用。在代码中比如星期、月份、HTTP 方法等常量结合就可以用枚举来声明。如果不给枚举不指定值，它会从零开始累加。。</p> <pre> 1 enum Color {Red, Green, Blue} 2 let c: Color = Color.Green; // 1; 3 4 enum HttpMethods { 5 Get = 'get', 6 Post = 'post', 7 } 8 let method = HttpMethods.Get; // get </pre> <p>由于 <code>JS</code> 是不支持枚举，但是由于 <code>TS</code> 中的枚举有可以当成值来使用，则肯定在编译的过程中需要做转换。所以会生成一份比较丑陋的代码，比如上面的代码会生成：</p>

```

1 var Color;
2 (function (Color) {
3     Color[Color["Red"] = 0] = "Red";
4     Color[Color["Green"] = 1] = "Green";
5     Color[Color["Blue"] = 2] = "Blue";
6 })(Color || (Color = {}));
7 var c = Color.Green;
8 /**
9  * COLOR =
10  * {
11  *   0: "Red"
12  *   1: "Green"
13  *   2: "Blue"
14  *   Blue: 2
15  *   Green: 1
16  *   Red: 0
17  * }
18 */
19
20 var HttpMethods;
21 (function (HttpMethods) {
22     HttpMethods["Get"] = "get";
23     HttpMethods["Post"] = "post";
24 })(HttpMethods || (HttpMethods = {}));
25 var method = HttpMethods.Get;

```

注：对于笔者来说，不是很喜欢使用枚举，除了针对上面提到编译出来的代码丑陋以外，还有就是如果枚举的值是使用的数字，会导致出现一些魔法数字，比如：

```

1 const getColor = (c: Color) => c;
2 getColor(Color.Red); // no error
3 getColor(1); // no error
4 getColor(999); // no error

```

由于上述的第四行没有报错，所以可以在编码的过程中出现预期以外的错误。所以笔者喜欢使用 `object as const` 来实现 `enum` 的功能。

```

1 type ValueOf<T> = T[keyof T];
2
3 const Color = {
4     Red: 0,
5     Green: 1,
6     Blue: 2,

```

	<pre> 7 } as const; 8 type TColor = keyof typeof Color; 9 type TColorValue = ValueOf<typeof Color> 10 const c: TColorValue = Color.Green; </pre>
any	<p>可以表示成任何类型，所有类型都可以赋值给 <code>any</code>，也可以把 <code>any</code> 类型赋值给除 <code>never</code> 以外的所有类型。当有些值是来自动态内容时，我们并不知道该值的类型，所以可以使用 <code>any</code> 来描述其类型，比如 <code>JSON.parse</code> 的结果。如果说代码中都使用 <code>any</code> 作为类型，那和写 <code>JS</code> 没有什么区别，我们也戏称这种代码为 <code>AnyScript</code></p>
unknown	<p>表示未知类型，用途和 <code>any</code> 很相似，可以看成一个安全版的 <code>any</code>，所以推荐在编码环境中使用 <code>unknown</code> 替换 <code>any</code>。所有类型都可以赋值给 <code>unknown</code>，但是只有 <code>unknown</code> 和 <code>any</code> 可以赋值给 <code>unknown</code>。一个比较常见的场景是使用 <code>JSON.parse</code> 时，返回值的类型是 <code>any</code>。所以如果不进行类型判断的话很容易出现错误，比如：</p> <pre> 1 const str = ' [{ "name": "lily" }]' 2 const users = JSON.parse(str); // value is any 3 users.forEach(user => console.log(user.name)); // error: user 是 隐式的 “any” 类型 </pre> <p>但是 <code>str</code> 的数据可能不会符合规范，假设 <code>str</code> 中不是一个符合的数组字符串，就会导致运行报错。因为默认返回 <code>any</code> 会导致我们忽略再次做类型判断（比较编写的时候代码中也不会报错）。所以在一个健壮的代码中，我们肯定是需要对于未知的类型加上类型注解的，我们可以使用自定义类型守卫进行判断。</p> <pre> 1 const str = '{ "name": "lily" }' 2 const user: unknown = JSON.parse(str); 3 4 // 提示我们 user 类似是 unknown，所以我们是获取不到 .name 的 5 // console.log(user.name.length); 6 interface User { 7 name: string; 8 } 9 const isUser = (user: any): user is User => { 10 return typeof user.name === 'string'; 11 } 12 if (isUser(user)) { 13 // 当前作用域中 user 的类型已经被 narrow 成 User 14 console.log(user.name.length); 15 } </pre>
void	

表示没有类型，`undefined` 的子类型，主要用在函数的返回值上面。比如下面一个函数就表示不会返回任何内容：

```
1 const noop = (): void => {};
```

很多同学看到这里可能觉得和返回 `undefined` 的区别是什么或者认为 `void` 是 `undefined` 的别名只是为了方便理解。针对疑惑，官方文档指出了区别：[查看详情](#)。简单的来说，函数的返回值是 `void` 的时候，可以在实现的时候返回其他类型，但是实际上获取不会改变，比如：

```
1 type VoidFn = () => void;
2 const fn: VoidFn = () => 1; // no error
3 const res = fn(); // res is void
```

所有我们应用这种特性可以使用在一些需要接收回调函数的地方。

```
1 declare function warpFn(cb: () => undefined): void;
2 const arr: string[] = [];
3 // 因为 arr.push 的返回值是 number 所有不能赋值给 undefined 故这一行会报错
4 // 如果将上面 cb 中的 undefined 改成 void 便不会有问题。
5 warpFn(() => arr.push('lily'));
```

never

表示不存在的值的类型，是任何类型的子类型，除了本身也没有任何子类型，即可以赋值给其他类型，但是其他类型（除了 `never`）均不能赋值给其他类型，包括 `any`）。

在正常的编码环境中很少使用，在一些类型体操中出现的次数比较多。官方的[介绍文档](#)也有标明该类型出现的地方。

比如下图 `fail` 就返回了 `never` 就说明出现这个函数时已经是该处终点了，后续的代码都不会被执行到，所以 `console.log` 会变灰色意味着这些代码是永远都到达不了的，如果使用 `lint` 格式化是时会删除。 `nodeJS` 中的 `process.exit` 就是返回 `never`。

```
function fail(msg: string): never {  
  throw new Error(msg);  
}
```

```
fail('error');  
console.log('hello world');
```



在正常的编码环境中也可以使用 `never` 进行提前的错误规避，可以 [查看尤雨溪的回答](#)。

联合与交叉

我们在实际的开发过程中变量的类型并不是单一的，比如 `array.find` 既可以返回数据的类型也可以返回 `undefined`，或者说我们写一个 `mixin` 的方法需要将2个类型合并。

交叉类型

交叉类型是使用 `&` 符合将多个类型合并成一个类型。如：`type C = A & B` 这样 `C 类型` 就既有 `A` 的类型 也有 `B` 的类型。常见的如 `Object.assign` 方法，可以将对象进行合并，所以需要这样的方法将每个对象的类型进行合并，或者说我们在编写 `React` 高阶组件时，编写的过程中就可以对已有类型进行拓展。

```
1 interface Props {  
2   name: string;  
3   age: number;  
4 }  
5 interface WithHOCProps {  
6   options: {  
7     size: number;  
8   }  
9 }  
10  
11 const App: React.FC<Props & WithHOCProps> = ({  
12   options,  
13 }) => {  
14   options.size // number  
15 }
```


由于是类型合并，所以可能会遇到 2 个类型不兼容的情况，所以如果遇到不兼容的类型就会推导出 `never`。

```
1 interface Item1 {
2   id: string;
3   name: string;
4 }
5 interface Item2 {
6   id: number;
7   age: number;
8 }
9
10 type C = Item1 & Item2;
11 type Id = C['id'] // never
12 type Name = C['name'] // string
13 type Age = C['age'] // number
```

因为上文（类型间的关系）已经展示出了不一样的类型也存在可以相互转换的。所以 `A & B` 的运算关系可以看成：

- A 和 B 可以相互赋值 => 目前只有 `any` 可以满足这种情况
- A 可以赋值给 B，B 不能赋值给 A => A
- B 可以赋值给 A，A 不能赋值给 A => B
- A 和 B 不存在可以赋值的关系 => `never`

```
1 type T1 = number & string; // never
2 type T2 = number & unknown; // number
3 type T3 = number & any; // any
4 type T4 = number & never; // never
```

联合类型

联合类型是使用 `|` 符合将多个类型联系起来，如：`C = A | B` 表明 `C` 要么等于类型 `A` 要么等于类型 `B`。主要用于当我们一个变量的类型不固定时，比如一个函数运行过程中正常的运算结果返回 `A`，运算失败返回 `B`。

```
1 const fn = (num: number): number | string => {
2   if (num >= 0) {
3     return num;
4   } else {
```



```
5         return 'error';
6     }
7 }
8 const res = fn(1);
```

当一个值是联合类型时，只可以调用联合类型的共有属性。如上面的 `res` 类型是 `number | string`，如果不加以判断只能调用共有的 `toString` 和 `valueOf` 等方法。

全局类型

通常情况下，定义的类型需要使用 `export` 进行导出，在使用的地方再使用 `import` 导入。但是有时候在同一个项目中会有一些通用的类型或者类型方法，每次都进行导入是很繁琐的。甚至需要在 `window` 挂载一些新的变量，所以我们需要了解全局类型的概念。声明全局类型的方式有 2 种：

1. 在一个 `.d.ts` 文件中写变量类型，同时不要有 `export` 和 `import` 等导入导出语法。

```
1 // global.d.ts
2 type AnyFunction = (...args: any[]) => any;
```

值得注意的是，你需要在 `tsconfig.json` 的 `include` 选项中包含该文件。另外需要注意的一点是，如果你是一个 `npm` 包的类型中，如果引入 `npm` 包的没有引入你定义的全局类型，则会变成 `any`。

2. 使用 `declare` 定义，比如需要给 `window` 新增类型，给某个包或者某一类文件添加类型说明等。

```
1 // global.d.ts
2 declare module 'react' {
3     export const licy: string;
4 }
5
6 declare module 'npm-package' {
7     export const props: { name: 'licy'; age: number }
8     const App: React.FC<typeof props>;
9     export default App;
10 }
11
12 declare module '*.svg' {
13     const content: {
14         id: string;
15     }
16     export default content;
17 }
```

```

18
19
20 // app.ts
21 import React from 'react';
22 import svg from './log.svg';
23 import { props } from 'npm-package';
24
25 React.lily // string
26 svg.id // string
27 props.name // 'lily'

```

当然很多时候，我们的类型还会引入一些已有类型进行组装，所以就会破坏掉默认 `.d.ts` 是全局类型的约束，所以需要主动的导出。

```

1 // global.d.ts
2 import { ValueOf } from './type';
3
4 declare namespace CommonNS {
5   interface Props {
6     name: 'lily';
7     age: 24
8   }
9   type Value = ValueOf<Props>;
10 }
11
12 // 缺一不可，否则类型使用会加前缀
13 // 将 CommonNS 作为全局类型，类似 UMD
14 export as namespace CommonNS;
15 // 将导出命名修改，否则就会使用 CommonNS.CommonNS.XXX 才可以获取
16 export = CommonNS;
17
18
19 // main.ts
20 const value: CommonNS.Value = 'lily';

```

在全局选项这里需要注意 `skipLibCheck` 的配置，如果该选项配置为 `true` 则会跳过库文件的类型检查，比如 `node_modules` 中其他库的类型检查和当前项目的 `.d.ts` 检查。所以会导致在编写 `.d.ts` 文件的时候不察觉错误，但是有不能保证引入的 `npm` 库的类型文件都是正确的。所以可以在 `tsconfig.json` 将该选项设置为 `false` 然后在编译阶段再将该选项设置为 `true`。

类型间的关系

通过上文我们可以得出类型之间的关系：

	any	unknown	void	null	undefined	never	通用 (string、number、boolean、symbol)
any 赋值给	Y	Y	Y	Y	Y	N	Y
unknown 赋值给	Y	Y	N	N	N	N	N
void 赋值给	Y	Y	Y	N	N	N	N
null 赋值给	Y	Y	N	Y	N	N	N
undefined 赋值给	Y	Y	Y	N	Y	N	N
never 赋值给	Y	Y	Y	Y	Y	Y	Y
通用 (string、number、boolean、symbol) 赋值给	Y	Y	N	N	N	N	同类型是可以相互赋值 如: <code>string</code> 赋值给 <code>string</code> , <code>number[]</code> 赋值给 <code>number[]</code> 。

- 以上表格是在严格模式下, 即 `strictNullChecks` 为 `true` 也是推荐的配置

类型推断与保护

正常情况下类型具有自动推断的能力, 比如我们声明一个变量 `const num = 1`, TS 会自动将变量的类型推断成 `number`, 所以后面我们就可以对 `num` 变量使用 `number` 的一些操作方法。但是当使用联合类型的时候, TS 在编译阶段就无法得知当前的变量类型是什么, 所以只可以使用共有的一些方法, 所以我们需要使用类型保护的能力, 比如可以通过一些判断来缩小当前变量或者断言当前变量的类型。

typeof

`typeof` 是判断变量类型的一个操作符, 我们可以通过 `typeof` 将类型缩小变成一个受保护的类型, 如:

```

1 const fn = (value: number | string): void => {
2   value // value is number | string
3
4   if (typeof value === 'number') {

```

```

5     value // value is number
6     value.toFixed // no error
7 } else {
8     value // value is string
9     value.length // no error
10 }
11 }

```

instanceof

和 `typeof` 类似，`instanceof` 也是一个判断变量类型的方式，比如一个函数可以同时接收不同的普通的对象，就会导致 `typeof value === 'object'` 分辨不出来。就可以使用 `instanceof` 来辨别对象是什么。

```

1 class User {
2   say(): void {
3     console.log('hello');
4   }
5 }
6
7 class Stu {
8   read(): void {
9     console.log('read');
10  }
11 }
12
13 const fn = (value: Stu | User): void => {
14   value // value is Stu | User
15
16   if (value instanceof User) {
17     value // value is User
18     value.say() // no error
19   } else {
20     value // value is Stu
21     value.read // no error
22   }
23 }

```

in

`in` 可以检查是否存在某个属性，如：

```

1 type Item1 = {

```

```

2   type: 'item1';
3   name: string;
4   age: number;
5 }
6
7 type Item2 = {
8   type: 'item2',
9   title: string;
10  description: string;
11 }
12
13 const fn = (value: Item1 | Item2): void => {
14   if ('name' in value) {
15     value.age // no error
16   } else {
17     value.description // no error
18   }
19 }

```

字面量判断

如上面的例子，如果后期对 `Item2` 添加 `name` 属性就容易导致这里类型判断失效，导致获取 `value.age` 就会存在问题。所以针对上述这种存在 `type` 区分的情况，可以直接使用字面量判断。

```

1 const fn = (value: Item1 | Item2): void => {
2   if (value.type === 'item1') {
3     value.age // no error
4   } else if (value.type === 'item2') {
5     value.description // no error
6   }
7 }

```

is 关键字自定义

使用上面的方法在简单的场景下是十分有效，但是有时候类型的判断是复杂的，或者这样的判断是通用的，所以为了避免重复的编写我们可能需要对这个类型的保护需要提取成函数，那么就可以使用 `is` 来进行指定。

```

1 type Item1 = {
2   type: 'item1';
3   name: string;
4   age: number;

```

```

5 }
6
7 type Item2 = {
8   type: 'item2',
9   title: string;
10  description: string;
11 }
12
13 const isItem1Arr = (value: any): value is Item1[] => {
14   if (!Array.isArray(value)) {
15     return false;
16   }
17   if (value.length === 0) {
18     return true;
19   }
20   return value.every(item => item.type === 'item1');
21 }
22
23 const fn = (value: Item1[] | Item2[]): void => {
24   if (isItem1Arr(value)) {
25     value.forEach(item => {
26       item.age // no error
27     });
28   }
29 }

```

其中 `isItem1Arr` 返回值是一个 `boolean` 如果返回的是 `true` 则说明当前类型是 `is` 后面的。

类型断言

有了类型断言我们可以轻松的迁移一个项目，但是类型断言是有害的，因为我们主动的给这个变量向 `TS` 类型检查器做了背书而不是通过类型保护的方式。所以假设传入的数据是有误的，就会导致运行异常，所以我们需要谨慎的使用类型断言，除非可以 100% 的保证这里类型。

as 与 <>

有的时候 `TS` 的检验规则是存在缺陷的，不能完美的做好类型保护，比如下面的例子，虽然我们已经提前判断过 `item.parent` 肯定不为空，但是在一个闭包环境中使用，由于 `TS` 的缺陷，类型还是失效了（因为我们是马上运行的）。但是我们可以保证这里的类型肯定是不为 `null` 的，所以我们可以断言它的类型。

```

1 interface Item {
2   parent: Item | null;

```

```

3 }
4
5 const fn = (item: Item) => {
6   if (!item.parent) {
7     return;
8   }
9   const _fn = () => {
10     item.parent // Item | null
11     const parent1 = item.parent as Item;
12     const parent2 = <Item>item.parent;
13   }
14   _fn();
15 }

```

第二种情况是，这个值是在运行过程中产生。所以我们没有办法在定义变量的时候进行初始化，这个时候就需要使用类型断言了。但是这种方式存在弊端，假设后面 `User` 新增了一个属性，就会导致返回的数据有缺省。所以不是很推荐这种方式，而是可以在初始化的时候设置成空值/默认值，这样当新增一个属性后，就会主动报错，提醒我们需要处理额外的属性。

```

1 interface User {
2   type: 'student';
3   name: string;
4 }
5
6 const createUser = (name: string): User => {
7   // const result = {
8   //   type: 'student'
9   // } as User;
10  const result = <User>{
11    type: 'student'
12  };
13
14  if (name) {
15    result.name = parseName(name);
16  }
17
18  return result;
19 }

```

!非空断言

顾名思义，主要是排除变量中 `null` 和 `undefined` 的类型。比如上面提到的 `item.parent` 我们可以很清楚的知道他不是一个 `null` 的，就可以使用这个简单的方式。


```

1 interface Item {
2   parent: Item | null;
3 }
4
5 const fn = (item: Item) => {
6   if (!item.parent) {
7     return;
8   }
9   const _fn = () => {
10    const p1 = item.parent.parent; // error: (item.parent) 对象可能为 null
11    const p2 = (<Item>item.parent).parent // ok, item.parent 整体断言
12    const p3 = item.parent!.parent; // ok, item.parent 非空, 则排除 null
13  }
14 }

```

双重断言

毫无根据的断言是危险的，所以进行类型断言时，TS 会提供额外的安全性，并不是每个变量间都可以断言的，比如 `'licy' as number` 将一个字符串转换成 `number` 肯定就是不行的。

如果 A 和 B 直接存在赋值关系，即 A 是 B 的子类型，或者 B 是 A 的子类型就可以直接使用断言。如果不存在时，可以找一个中间的类型来做桥梁，通过上面【类型间的关系】可以得出 `any`、`unknown`、`never` 三个类型是最少都满足上面 2 个规律之一。

```

1 const n1 = 'licy' as number; // error: string 与 number 不能充分重叠，转换是错误的
2 const n2 = 'aa' as any as number;
3 const n3 = 'aa' as unknown as number; // 推荐
4 const n4 = 'aa' as never as number;

```

因为断言是具有危害性的，所以双重断言也是具有危害性的。我们需要尽量的少用。同时双重断言的使用场景很少很少，笔者只在一次跨 `npm` 包调用的时候，由于底层版本不一致，使用过一次。

高级用法

函数重载

函数重载是静态类型语言当中很重要的一个能力。很多时候编写的函数可能会兼容多种参数类型，可能会根据传入的参数会返回不同的数据。比如：

```

1 const data = { name: 'licy' };
2 const getData = (stringify: boolean = false): string | object => {

```

```

3   if (stringify === true) {
4       return JSON.stringify(data);
5   } else {
6       return data;
7   }
8 }
9
10 const res1 = getData(); // string | object
11 const res2 = getData(true); // string | object

```

在上述的例子中调用 `getData` 方法得到一个联合了联合类型，还需要进行判断将类型缩小或者使用 `as` 进行指定。但是如果作为方法的编写者，当确定传入的参数后就可以很准确的得到返回值的类型，而不是得到这种模棱两可的情况。所以借助函数重载进行改造：

```

1  const data = { name: 'licy' };
2  function getData(stringify: true): string
3  function getData(stringify?: false): object
4  function getData(stringify: boolean = false): unknown {
5      if (stringify === true) {
6          return JSON.stringify(data);
7      } else {
8          return data;
9      }
10 }
11
12 const res1 = getData(); // object
13 const res2 = getData(true); // string

```

函数重载的使用方法很简单，就是在需要使用函数重载的地方，多声明几个函数的类型。然后在最后一个函数中进行实现，特别要注意的是，最后实现函数中的类型一定要与上面的类型兼容。

值得注意的是由于 `TS` 是在编译后会将类型抹去生成 `JS` 代码，而 `JS` 是没有函数重载这样的能力，所以说这里的函数重载只是类型的重载，方便做类型的提示，实际上还是要在实现函数中进行传入参数的判别，然后返回不同的结果。

分布式条件类型

分布式的联合类型只看extends的左边那一个，其他的联合类型不会进行分布式。

如果是never类型就不会返回(这个仅限于只有条件类型的时候)

```

1  type Exclude<T, P> = T extends P ? never : T;
2  type test = Exclude<'a' | 'b' | 'c', 'c'>

```

泛型

泛型是 TS 一个比较高级的用法，在日常的开发中也是使用比较多的。当你的函数，接口或者类需要支持多种类型的时候就可以使用泛型，比如上面函数重载的例子，也可以使用泛型进行改造。

```
1 // 泛型函数
2 const data = { name: 'licy' };
3 function getData<T extends boolean = false, R = T extends true ? string :
  object>(stringify?: T): R {
4   if (stringify === true) {
5     return JSON.stringify(data) as unknown as R;
6   } else {
7     return data as unknown as R;
8   }
9 }
10
11 const res1 = getData(); // object
12 const res2 = getData(true); // string
13
14 // 泛型类型
15 type ValueOf<T> = T[keyof T];
16
17 interface User {
18   name: 'licy';
19 }
20
21 type A = keyof User; // 'name'
22 type B = ValueOf<User>; // 'licy'
```

亦或者需要对传入进来的参数进行保存时，比如编写 `React` 中的 `HOC`。

```
1 type AnyObject = Record<string, any>;
2 type ExtraProps = {
3   name: 'licy'
4 };
5 const withItem = <
6   T extends AnyObject
7 >(Comp: React.FC<T>): React.FC<T & ExtraProps> => {
8   const NewComp: React.FC<T & ExtraProps> = (props) => {
9     props.name // 'licy'
10     return <Comp {...props} />
11   }
12   NewComp.displayName = 'with-item';
```

```

13   return NewComp;
14 }
15
16 const Demo: React.FC<{ age: number }> = () => null;
17
18 const NewDemo = withItem(Demo);
19
20 const res = (
21   <>
22     <Demo age={24} /> no error
23     <NewDemo age={24} name="lily" /> no error
24     <NewDemo age={24} /> error: 缺少属性 "name"
25   </>
26 )

```

泛型函数

```

1 type funcType = <T>(val: T) => T[];
2 const fn1: funcType = val => [val];
3
4 const fn2 = <T>(val: T): T[] => [val];

```

泛型接口

```

1 interface interfaceType<T> {
2   a: T
3 }
4
5 const obj: interfaceType<string> = {
6   a: '1'
7 }

```

泛型约束

```

1 type funcType<T extends { value: string }> = (val: T) => string[];

```

内置的高级函数

为了方便类型编写，TS 官方内置了许多通用的高级类型方法，这些方法可以帮助我们完成程序中大部分的类型转换。但是如果我们掌握了这些类型方法的实现方式，也可以很轻松的写出符合业务逻辑规范的高级方法。所有的内置方法可以参考：[utility-types](#) 本文只介绍一些典型的。

方法名	作用	实现 & 使用	注意点
Partial	将一个对象中的 key 变成可选的	<pre>1 type Partial<T> = { 2 [P in keyof T]?: T[P]; 3 }; 4 5 interface User { 6 name: 'lily'; 7 } 8 const user: Partial<User> = {} // 没有错误，因为此时 name 是可选的</pre>	首先使用 <code>keyof</code> 得到传入对象的所有 <code>keys</code> 它是一个联合类型。然后在用 <code>in</code> 关键字遍历得到每个 <code>key</code> ，最后再加上 <code>?</code> 表明就是这个类型是可选的。 <code>T[P]</code> 将传入的类型获取出来。
Required	和 <code>Partial</code> 相反，是将传入类型中可选的类型变成必填类型。	<pre>1 type Required<T> = { 2 [P in keyof T]-?: T[P]; 3 };</pre>	比起 <code>Partial</code> 的实现不同的是 <code>?</code> 变成可 <code>-?</code> ，这可以理解为去除了 <code>?</code> 。
Readonly	顾名思义将类型变成可读的，即不可修改。	<pre>1 type Readonly<T> = { 2 readonly [P in keyof T]: T[P]; 3 }; 4 5 interface User { 6 name: 'lily'; 7 } 8 const user: Readonly<User> = { 9 name: 'lily' 10 }; 11 user.name = 'lily'; // error: 无法分配到 "name"，因为它是只读属性。</pre>	和我们声明可读类型类似，就在遍历每个 <code>key</code> 的时候加上 <code>readonly</code> 。
Pick	从一个类型中选出想要的类型，和	<pre>1 type Pick<T, K extends keyof</pre>	注意实现的第二个参数 <code>K extends</code>

	<p><code>lodash</code> 的 <code>pick</code> 方法很像。</p>	<pre> 1 T> = { 2 [P in K]: T[P]; 3 }; 4 5 // 使用 6 interface User { 7 name: string; 8 age: number; 9 addr: string; 10 } 11 const user: Pick<User, 'name' 'age'> = { 12 name: 'licy', 13 age: 24 14 }; // no error, 因为只选择了 name 和 age </pre>	<p><code>keyof T</code>，这就表明当我们填写了第一个参数如 <code>User</code>，<code>K</code> 就会有类型提示，提示可以填写的值。</p>
Omit	<p>和 <code>Pick</code> 想法，排除不想要的值。</p>	<pre> 1 type Exclude<T, U> = T extends U ? never : T; 2 type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>; 3 4 type myOmit<T, K extends keyof T> = { 5 [U in keyof T as (6 U extends K ? never : U 7)]: T[U] 8 } 9 10 interface User { 11 name: string; 12 age: number; 13 addr: string; 14 } 15 const user: Omit<User, 'name' 'addr'> = { 16 name: 'licy', 17 age: 24, 18 }; // error: name 不在类型中 </pre>	<p>从实现来看 <code>Omit</code> 使用了 <code>Pick</code> 和 <code>Exclude</code>。</p> <p>从 <code>Exclude</code> 的实现就可以看出来，如果是从 <code>T</code> 中排除 <code>U</code>。相当于 <code>Omit</code> 的实现就是先将传入需要排除的 <code>Key</code> 找到不排除的，然后使用 <code>Pick</code> 得到。</p> <p>去除接口中某个key的方法，可以在映射类型中将这个key断言为 <code>never</code>，具体例子看 <code>myOmit</code></p>
Parameters	<p>获取函数的参数</p>	<pre> 1 type Parameters<T extends (...args: any) => any> = T </pre>	<p>这里我们需要注意一下使用 <code>infer</code> 这个关键字。可以将这个</p>

		<pre>extends (...args: infer P) => any ? P : never; 2 3 // 使用 4 type Fn = (name: string, age: number) => string; 5 type Params = Parameters<Fn>; // [name: string, age: number] 6</pre>	<p>关键理解成报错推断后的类型，并且它只可以用于 extends 的子句当中。</p> <p>比如当前就可以理解成如果传入的 T 是符合 (...args: infer P) => any 的，那么就提取到了传入函数的参数列表。</p>
ConstructorParameters	获取 class 构造方法的参数。	<pre>1 type ConstructorParameters<T extends abstract new (...args: any) => any> = T extends abstract new (...args: infer P) => any ? P : never; 2</pre>	
Uppercase	字符串转大写	内部特殊实现	
Lowercase	字符串转小写	内部特殊实现	
Capitalize	首字母大写	内部特殊实现	
Uncapitalize	首字母小写	内部特殊实现	

通过上面 TS 内部实现的高级类型可以发现，extends 和 infer 是特别重要的。extends 可以实现类似三元表达式的判断，判断传入的泛型是什么类型的，然后返回定义好的类型。infer 可以在判断是什么类型后，可以提取其中的类型并在子句中使用。和我们正常写代码一样，说明我们可以多个 extends 和 infer 进行嵌套使用，这样就可以把一个传入的泛型进行一次次分解。

协变与逆变

在了解协变与逆变之前我们需要知道一个概念——子类型。我们前面提到过 string 可以赋值给 unknown 那么就可以理解为 string 是 unknown 的子类型。正常情况下这个关系即子类型可以

赋值给父类型是不会改变的我们称之为协变，但是在某种情况下两者会出现颠倒我们称这种关系为逆变。如：

```
1 interface Animal {
2   name: string;
3 }
4
5 interface Cat extends Animal {
6   catBark: string;
7 }
8
9 interface OrangeCat extends Cat {
10  color: 'orange'
11 }
12
13 // ts 中不一定要使用继承关系，只要是 A 的类型在 B 中全部都有，且 B 比 A 还要多一些类型
14 // 类似集合 A 属于 B 一样，这样就可以将 B 叫做 A 的子类型。
15
16 // 以上从属关系
17 // OrangeCat 是 Cat 的子类型
18 // Cat 是 Animal 的子类型
19 // 同理 OrangeCat 也是 Animal 的子类型
20
21 const cat: Cat = {
22   name: '猫猫',
23   catBark: '喵~~'
24 }
25 const animal: Animal = cat; // no error
```

假设我有类型 `type FnCat = (value: Cat) => Cat;` 请问下面四个谁是它的子类型，即以下那个类型可以赋值给它。

```
1 type FnAnimal = (value: Animal) => Animal;
2 type FnOrangeCat = (value: OrangeCat) => OrangeCat;
3 type FnAnimalOrangeCat = (value: Animal) => OrangeCat;
4 type FnOrangeCatAnima = (value: OrangeCat) => Animal;
5
6 type RES1 = FnAnimal extends FnCat ? true : false; // false
7 type RES2 = FnOrangeCat extends FnCat ? true : false; // false
8 type RES3 = FnAnimalOrangeCat extends FnCat ? true : false; // true
9 type RES4 = FnOrangeCatAnima extends FnCat ? true : false; // false
```

为什么 `RES3` 是可以的呐？

返回值：假设使用了 `FnCat` 返回值的 `cat.catBark` 属性，如果返回值是 `Animal` 则不会有这个属性，会导致调用出错。估计返回值只能是 `OrangeCat`。

参数：假设传入的函数中使用了 `orangeCat.color` 但是，对外的类型参数还是 `Cat` 没有 `color` 属性，就会导致该函数运行时内部报错。

故可以得出结论：返回值是协变，入参是逆变。

注意如果 `tsconfig.json` 中的 `strictFunctionTypes` 是 `false` 则上述的 `RES2` 也是 `true`，这就表明当前函数是支持双向协变的。当然 `TS` 默认是关闭此选项的，主要是为了方便 `JS` 代码快速迁移到 `TS` 中，详情可以见 [why-are-function-parameters-bivariant](#)，当然如果是一个新项目，建议打开 `strictFunctionTypes` 选项。

允许双向协变是有风险的，可能会在运行时报错。比如在 `ESLint` 中有 `method-signature-style` 规则，简单的来说该规则默认是使用 `property` 来声明方法，比如：

```
1 interface T1<T> {
2   wrapFn: (value: T) => void;
3 }
4
5 interface T2<T> {
6   wrapFn(value: T): void; // eslint error, 声明的方式是 method 形式
7 }
```

假设我们忽略 `eslint` 警告，强制 `T2` 的方法进行声明就会潜在的双向协变的风险，如下列代码：

```
1 declare let animalT1: T1<Animal>;
2 declare let catT1: T1<Cat>
3
4 animalT1 = catT1; // error, Animal 不能分配给 Cat
5 catT1 = animalT1; // no error
6
7 declare let animalT2: T2<Animal>;
8 declare let catT2: T2<Cat>
9
10 animalT2 = catT2; // no error
11 catT2 = animalT2; // no error
```

真实案例

联合类型交叉类型

题目描述

```

1 type Value = { a: string } | { b: number }
2 type Res = UnionToIntersection<Value> // type Res= { a: string } & { b:
  number };

```

思路

1. **Distributive Conditional Types**：当条件类型作用于泛型类型时，它们在给定联合类型时变得可分配。

```

1 // extend 中的分配
2 type ToArray<Type> = Type extends any ? Type[] : never;
3 type StrArrOrNumArr = ToArray<string | number>; // string[] | number[]

```

2. **逆变的特性**：在逆变位置时推断出交叉类型

```

1 // 逆变推断出交叉
2 type Bar<T> = T extends { a: (x: infer U) => void, b: (x: infer U) => void } ?
  U : never;
3 type T20 = Bar<{ a: (x: string) => void, b: (x: string) => void }>; // string
4 type T21 = Bar<{ a: (x: string) => void, b: (x: number) => void }>; // string
  & number
5

```

结合二者

```

1 type ToUnionFunction<T> = T extends unknown ? (x: T) => void : never;
2 type UnionToIntersection<T> = ToUnionFunction<T> extends (x: infer R) =>
  unknown
3     ? R
4     : never

```

测试

```

1 type Res = UnionToIntersection<Value> // type Res= { a: string } & { b:
  number };

```

增强版 Omit

题目描述

假如我们要轻微的修改某个组件亦或者是拓展组件的某个属性，比如下面代码中的 `size`，想给他增加一个 `default` 的配置项，但是其他的 `props` 都不会改变。首先肯定不想进行复制粘贴，其次也为了后续内部的组件 `props` 变更后，外层的逻辑不改。

第一印象就是继承然后修改 `size` 的值，但是很遗憾因为新的类型与已有的类型不兼容，所以不能覆盖成功。所以谋生了第二种思路，想把 `size` 给排除出去，然后重写。但是也不行，因为拓展的组件为了方便添加了 `[key: string]: unknown`，导致 `omit` 有问题。

```
1 interface Props {
2   title: string;
3   size: 'small' | 'large';
4   [key: string]: unknown;
5 }
6
7 interface NewProps1 extends Props {
8   size: 'small' | 'large' | 'default'; // error: 不能将 default 分配给 'small'
    和 'large'
9 }
10
11 interface NewProps2 extends Omit<Props, 'size'> {
12   size: 'small' | 'large' | 'default';
13 }
14
15 const a: NewProps2 = {
16   title: 123, // no error, 但是我们知道 title 类型丢失了
17   size: 'default',
18 }
```

思路

因为是添加了 `[key: string]: unknown` 后才导致 `omit` 失效的，根据上面 `omit` 的实现我们可以知道实际上是 `keyof` 方法不能处理 `[key: string]` 这样的索引签名，根据常识对象索引签名的类型只支持 `string`、`number`、`symbol` 和字面量。所以只需要保留字面量的 `key` 就行了。

```
1 /**
2  * 我们已知
3  * 1. K 只会为 string | number | symbol | 字面量
4  * 2. string extends K 时，只有 K 为 string 时才是 true，同理这里可以检查出 number
    和 symbol 然后 as 为 never.
5  */
6 type KnownKeys<T> = keyof {
7   [K in keyof T as (
```

```

8      string extends K
9      ? never
10     : number extends K
11     ? never
12     : symbol extends K
13     ? never
14     : K)
15   ]: never;
16 };
17
18 /**
19  * 因为 Pick 的第二个参数需要 K extends keyof T
20  * 所以这里需要判断 KnownKeys<T> extends keyof T,
21  */
22 export type ObtainKeyOmit<T, K extends KnownKeys<T>> = KnownKeys<T> extends
  keyof T ? Pick<T, Exclude<KnownKeys<T>, K>> : never;

```

测试

```

1 interface NewProps3 extends ObtainKeyOmit<Props, 'size'> {
2   size: 'small' | 'large' | 'default';
3   [key: string]: unknown; // 因为 ObtainKeyOmit 去掉了, 所以需要加回来
4 }
5
6 const a: NewProps3 = {
7   title: 123, // error, number 不能赋值给 string
8   size: 'default',
9   name: 'licy',
10 }

```

下划线转驼峰

题目描述

根据目前大多数的规范来说, 服务端开发的同学大多数是使用下划线命名, 而前端的同学是用小驼峰命名。所以大部分同学会在 `axios` 请求数据回来的 `hooks` 中使用 `camelcase-keys` 将数据中的下划线转换成小驼峰。但是大多数接口的类似是使用 `thrift` 进行转换的, 所以生成的类型文件也是下划线构造的。所以可以完成一个方法将下划线转换成小驼峰。

```

1 // thrift 的构造, key 为下划线, 其中值有可能是数组或者对象的嵌套
2 interface User {
3   user_id: string;
4   name: string;

```

```

5   user_status?: number;
6   avatar_url: {
7     default_url: string;
8     small_url?: string;
9     large_url?: string;
10  };
11  colleague: {
12    user_name: string;
13    user_id: string;
14    user_status?: number;
15  }[];
16 };

```

思路

1. 首先需要完成下划线字符串转驼峰的方法 `Under2camel`

- 使用 `T extends `${infer F}_${infer L}`` 的方式可以获取 `_` 前后的值
- 使用 `Capitalize` 可以将字符串的首字母大写
- 注意兼容 `_xx` 开头的数据

2. `Under2camelDeep` 的方法其实只是根据值的类型添加的递归处理

- `key` 进行转换，如果是字符串就使用 `Under2camel` 方法，如果是 `number` 和 `symbol` 就跳过。
- 使用 `T[K] extends (infer O)[]` 判断是数组，并且还可以提取其中的值

```

1  // 只做字符串的转换
2  type Under2camel<T extends string> = T extends `${infer F}_${infer L}`
3    ? F extends ''
4      ? Under2camel<L>
5      : `${F}${Under2camel<Capitalize<L>>}`}
6    : T;
7  type AAA = Under2camel<'_user_id_ddd_aaa_'>; // userIdDddAaa
8
9  // 递归遍历
10 type Under2camelDeep<T> = T extends (infer U)[]
11   ? Under2camelDeep<U>[]
12   : {
13     [K in keyof T as (K extends string ? Under2camel<K> : never)]: T[K] extends
14       Record<string, unknown>
15       ? Under2camelDeep<T[K]>
16       : T[K] extends (infer O)[]
17         ? Under2camelDeep<O>[]
18         : T[K]

```

```
18 }
19
```

测试

```
1 const data: Under2camelDeep<User> = {
2   userId: 'id001',
3   name: 'licy',
4   userStatus: 0,
5   avatarUrl: {
6     smallUrl: 'https://xxx',
7     defaultUrl: 'https://xxx'
8   },
9   colleague: [
10    {
11      userName: 'zhangsan',
12      userId: 'id002',
13      userStatus: 0,
14    }
15  ]
16 };
```

课后作业

编写一个 `Under2camelDeep` 方法的相反方法，`Camel2underDeep` 可以将小驼峰命名转换成 `_` 连接。

加强版 Pick

题目描述

很多时候我们会使用 `lodash` 的 `pick` 方法进行深度的选取，如 `_.pick(o, ['a', 'b.c', 'b.e[0].a'])`。所以也希望能提供一个 `DeepPick` 的方法可以进行深度的选取。

```
1 interface User {
2   name: string;
3   address: {
4     country?: string;
5     city: string;
6   };
7   spend: {
8     price: number;
9     description?: string;
10  }[];
```



```

11 }
12
13 type NewUser = DeepPick<User, 'name' | 'address.city' | 'spend[0].price'>;

```

思路

1. 前文提到过，联合类型具有分配的性质，可以想象成一个数组，每一次只会有一个值代入表达式计算，最后的结果也是一个联合类型。
2. 使用 `infer` 的方式去提取字符串
3. 由于 `[0].` 和 `.` 有共同的部分，需要先判断 `[0].`
4. 递归即可
5. 得到的数据是 `{name: string} | { address: { city: BJ } }` 的形式，并不是一个交叉的类型。
6. 使用前文提到的 `UnionToIntersection` 方法将联合类型转换成交叉类型。

```

1 type _DeepPick<T, U> = U extends `${infer F}[0].${infer Rest}`
2   ? F extends keyof T
3     ? T[F] extends (infer O)[]
4       ? { [P in F]: DeepPick<O, Rest>[] }
5       : never
6     : never
7   : U extends `${infer F}.${infer Rest}`
8     ? F extends keyof T
9       ? { [P in F]: DeepPick<T[F], Rest> }
10      : never
11    : U extends keyof T
12      ? { [P in U]: T[U] }
13      : never;
14
15 type DeepPick<T, U> = UnionToIntersection<_DeepPick<T, U>>;

```

测试

```

1 type NewUser = DeepPick<User, 'name' | 'address.city' | 'spend[0].price'>;
2 // {
3 //   name: string;
4 //   address: {
5 //     city: string;
6 //   };
7 //   spend: {

```

```

8 //    price: number;
9 //    }[]
10 // }
11
12 const a: NewUser = {
13     name: 'licy',
14     address: {
15         city: 'BJ',
16     },
17     spend: [
18         {
19             price: 100,
20         },
21     ],
22 };
23

```

课后作业

1. 如果加入 `address.country` 后，由于该项是一个可选项，所以可以不写，但是目前解析成了 `string | undefined` 所以不写会进行报错，如何修复。

```

1 type NewUser = DeepPick<User, 'name' | 'address.city' | 'spend[0].price'>;
2 // {
3 //   name: string;
4 //   address: {
5 //     country: string | undefined;
6 //     city: string;
7 //   };
8 //   spend: {
9 //     price: number;
10 //   }[]
11 // }
12
13 const a: NewUser = {
14     name: 'licy',
15     address: { // 报错, 缺少 country 类型
16         city: 'BJ',
17     },
18     spend: [
19         {
20             price: 100,
21         },
22     ],
23 };

```

2. 修改 `DeepPick<T, U>` 中的 `U` 可以让输入的时候和 `Pick` 方法一样，有提示。

Object.assign 类型提示

题目描述

`Object.assign` 是一个常用 API 方法，但是目前来说这个方法自动推断的类型是有问题的，因为他是采用 `type1 & type2` 的方式，所以如果 2 个类型没有共有属性，就会得到 `never`。

```
1  type O1 = {
2    id: string;
3    value: string;
4    age?: number;
5    extra?: {
6      a: number;
7      type: 'a';
8    };
9  };
10
11 type O2 = {
12   name: string;
13   value: number;
14   city?: string;
15   extra?: {
16     type: 'b';
17   };
18 };
19
20 type O3 = {
21   city: string[];
22 };
23
24 const o1: O1 = {
25   id: 'id1',
26   value: 'abc',
27   age: 24,
28   extra: {
29     a: 1,
30     type: 'a',
31   },
32 };
33
34 const o2: O2 = {
35   name: 'licy',
36   value: 0,
```

```

37   city: 'BJ',
38   extra: {
39     type: 'b',
40   },
41 };
42
43 const o3: O3 = {
44   city: ['bj'],
45 };
46
47 const res = Object.assign({}, o1, o2, o3);
48 // 类型丢失了
49 res.value // never
50 res.extra // never
51 res.city // string & string[]

```

思路

因为 `&` 是取交集，而这里 `assign` 的能力是覆盖所以不能直接 `&`。应该判断如果后面的类型中有，则前面的类型就不需要有了。所以在遍历 `T1` 时，如果该 `key` 存在于 `T2` 中，则不应该有此项。

```

1 type Merge<T1 extends Record<string, unknown>, T2 extends Record<string,
  unknown>> = {
2   [K in keyof T1 as K extends keyof T2 ? never : K]: T1[K];
3 } & {
4   [K in keyof T2]: T2[K];
5 };
6
7 type Assign<T extends Record<string, unknown>, U> = U extends [infer F,
  ...infer Rest]
8   ? F extends Record<string, unknown>
9     ? Assign<Merge<T, F>, Rest>
10    : Assign<T, Rest>
11   : T;

```

测试

```

1 const res: Assign<O1, [O2, O3]> = Object.assign({}, o1, o2, o3);
2 // 类型推断正确
3 res.value // number
4 res.extra // { type: 'b' }
5 res.city // string[]

```

课后作业

1. 目前这个方法不是完美的，存在一定的缺陷，假设 `O3` 的类型中有未必填的 `key` 值，然后就会导致类型推断出现问题。如：

```
1 type O3 = {
2   id?: number;
3   city: string[];
4 };
5
6 const res: Assign<O1, [O2, O3]> = Object.assign({}, o1, o2, o3);
7 res.id // id?: number | undefined , 有问题 因为 O1 是肯定有 id, 所以推断出了问题
```

如何完善 `Merge` 方法，使得上面的 `res.id` 自动推断为 `string | number`。

2. 如果是深度的 assign 如何实现？比如我期望上文中的 `res.extra` 推断为 `{ a: number; type: 'b' }`

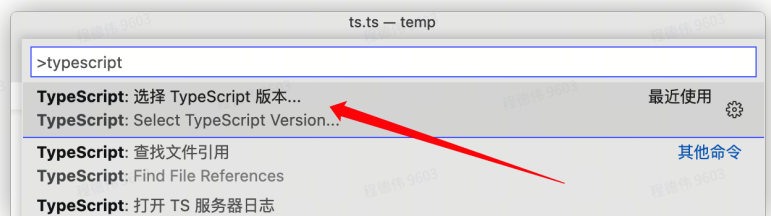
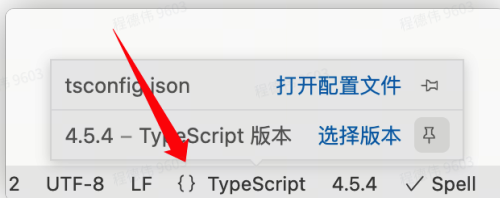
周边工具

推荐工具

1. 在线代码练习：TS 代码演练场

2. 本地练习：VSCode 编辑器

- a. 默认情况下是使用 VSCode 的 TS 版本进行，所以可能造成编写时的提示和编译版本不一致。可以点击图中的 `{}`，然后选择 TS 版本。也可以通过 `command+shift+p` 调出命令界面，输入 `typescript` 进行选择。



3. 很多时候我们在代码中编写工具函数，然后等着页面刷新是很麻烦的，我们可以在一个测试 TS 中间中进行编写，编写完成后直接把代码复制过去。直接运行 TS 的工具 `ts-node`，可以监听变化热更新 `ts-node-dev`。当然也可以用目前最新的工具 `bun`。

```
temp ➤ npx ts-node ./ts.ts
hello licy!!
temp ➤ █
```

```
temp ➤ bun ./ts.ts
hello licy
temp ➤ █
```

4. 很多常用的 TS 类型工具库，可以看成和 `lodash` 类似，如果有不知道如何写的可以参考。TS 类型工具库
5. Lint 检查工具，以前有专门的 TS Lint 但是由于 TS Lint 和 ES Lint 过程高度的相似，所以目前 TSLint 被并入了 ES lint 而 TS Lint 也被官方标记为了放弃维护。所以可以安装：`@typescript-eslint/eslint-plugin` 和 `@typescript-eslint/parser`，使用 ESLint 进行代码风格的检测。
6. 类型覆盖检查工具，可以使用 `type-coverage` 进行项目中类型覆盖度的检测。特别适合从一个 JS 项目迁移到 TS 项目的过程中，得到阶段性的数据，当然也可以作为一个 MR 的准入标准。比如下面就是目前低代码三个核心包的类型覆盖率，还是有一点提升空间的。

```
cs1-framework ➤ feat-dnd type-coverage -p ./packages/low-code/editor-plugins/tsconfig.json
13788 / 14616 94.33%
type-coverage success.
cs1-framework ➤ feat-dnd type-coverage -p ./packages/low-code/editor-schema/tsconfig.json
6068 / 6559 92.51%
type-coverage success.
cs1-framework ➤ feat-dnd type-coverage -p ./packages/uidl/uidl-tools/tsconfig.json
1427 / 1430 99.79%
type-coverage success.
cs1-framework ➤ feat-dnd █
```

推荐学习资料

TypeScript 官方使用手册

深入理解 TypeScript

【强烈推荐】`type-challenges`

类型体操天花板是怎样炼成的 - Web Infra 团队定期技术分享

用 TypeScript 类型运算实现一个中国象棋程序

TypeScript 类型体操天花板，用类型运算写一个 Lisp 解释器

国[★全技巧解析史诗典藏★]用 TypeScript 类型写一个 Lisp 解释器 Pro（尾递归优化版）

公司内部的一个 TS 讨论群，遇到不会体操问题，可以向大佬求救。



无法直接创建“群名片”的副本。请重新插入或打开原文复制粘贴。