想去力扣当前端，TypeScript 需要掌握到什么程度？

2018 年底的时候，力扣发布了岗位招聘，其中就有前端，仓库地址：https://github.com/LeetCode-OpenSource/hire 。与大多数 JD 不同， 其提供了 5 道题， 并注明了完成一个或多个面试题，获取免第一轮面试的面试机会。完成的题目越多，质量越高，在面试中的加分更多。完成后的代码可以任意形式发送给 jobs@lingkou.com。以上几个问题完成一个或多个都有可能获得面试机会，具体情况取决于提交给我们的代码。



（力扣中国前端工程师 JD）

今天我们就来看下第二题：编写复杂的 TypeScript 类型。通过这道题来看下， TypeScript 究竟要到什么水平才能进力扣当前端？

其它四道题也蛮有意思的，值得一看。

## 问题描述

假设有一个叫 EffectModule 的类

class EffectModule {}

这个对象上的方法**只可能**有两种类型签名:

interface Action<T> {  
 payload?: T  
 type: string  
}  
  
asyncMethod<T, U>(input: Promise<T>): Promise<Action<U>>  
  
syncMethod<T, U>(action: Action<T>): Action<U>

这个对象上还可能有一些任意的**非函数属性**：

interface Action<T> {  
 payload?: T;  
 type: string;  
}  
  
class EffectModule {  
 count = 1;  
 message = "hello!";  
  
 delay(input: Promise<number>) {  
 return input.then((i) => ({  
 payload: `hello ${i}!`,  
 type: "delay",  
 }));  
 }  
  
 setMessage(action: Action<Date>) {  
 return {  
 payload: action.payload!.getMilliseconds(),  
 type: "set-message",  
 };  
 }  
}

现在有一个叫 connect 的函数，它接受 EffectModule 实例，将它变成另一个对象，这个对象上只有**EffectModule 的同名方法**，但是方法的类型签名被改变了:

asyncMethod<T, U>(input: Promise<T>): Promise<Action<U>> 变成了  
asyncMethod<T, U>(input: T): Action<U>

syncMethod<T, U>(action: Action<T>): Action<U> 变成了  
syncMethod<T, U>(action: T): Action<U>

例子:

EffectModule 定义如下:

interface Action<T> {  
 payload?: T;  
 type: string;  
}  
  
class EffectModule {  
 count = 1;  
 message = "hello!";  
  
 delay(input: Promise<number>) {  
 return input.then((i) => ({  
 payload: `hello ${i}!`,  
 type: "delay",  
 }));  
 }  
  
 setMessage(action: Action<Date>) {  
 return {  
 payload: action.payload!.getMilliseconds(),  
 type: "set-message",  
 };  
 }  
}

connect 之后:

type Connected = {  
 delay(input: number): Action<string>;  
 setMessage(action: Date): Action<number>;  
};  
const effectModule = new EffectModule();  
const connected: Connected = connect(effectModule);

要求：

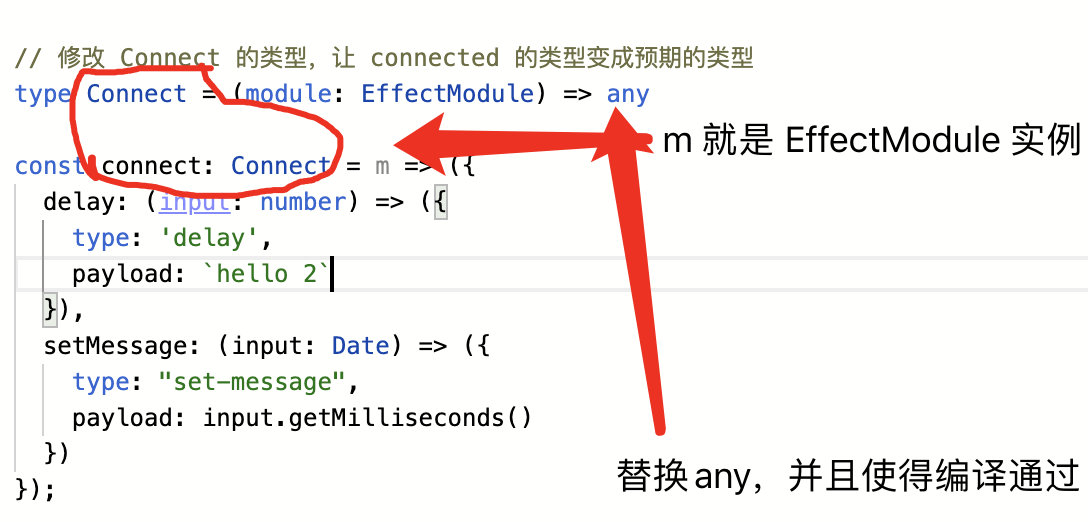
在 [题目链接](https://codesandbox.io/s/4tmtp) 里面的 index.ts 文件中，有一个 type Connect = (module: EffectModule) => any，将 any 替换成题目的解答，让编译能够顺利通过，并且 index.ts 中 connected 的类型与:

type Connected = {  
 delay(input: number): Action<string>;  
 setMessage(action: Date): Action<number>;  
};

**完全匹配**。

以上是官方题目描述，下面我的补充

上文提到的index.ts 比 题目描述多了两个语句，它们分别是：



（题目额外信息）

## 思路

首先来解读下题目。 题目要求我们补充类型 Connect 的定义， 也就是将 any 替换为不报错的其他代码。

回顾一下题目信息：

* 有一个叫 connect 的函数，它接受 EffectModule 实例，将它变成另一个对象，这个对象上只有**EffectModule 的同名方法**，但是方法的类型签名被改变了
* 这个对象上还可能有一些任意的**非函数属性**
* 这个对象（EffectModule 实例）上的方法**只可能**有两种类型签名

根据以上信息，我们能够得到：我们只需要将作为参数传递进来的 EffectModule 实例上的函数类型签名修改一下，非函数属性去掉即可。所以，我们有两件问题要解决：

1. 如何将非函数属性去掉
2. 如何转换函数类型签名

### 如何将非函数属性去掉

我们需要定义一个泛型，功能是接受一个对象，如果对象的 value 是 函数，则保留，否则去掉即可。不懂泛型的朋友可以先看下我之前写的文章： [你不知道的 TypeScript 泛型（万字长文，建议收藏）](https://lucifer.ren/blog/2020/06/16/ts-generics/)

这让我想起了官方提供的 Omit 泛型 Omit<T,K>。举个例子：

interface Todo {  
 title: string;  
 description: string;  
 completed: boolean;  
}  
  
type TodoPreview = Omit<Todo, "description">;  
  
// description 属性没了  
const todo: TodoPreview = {  
 title: "Clean room",  
 completed: false,  
};

官方的 Omit 实现：

type Pick<T, K extends keyof T> = {  
 [P in K]: T[P];  
};  
type Exclude<T, U> = T extends U ? never : T;  
type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>;

实际上我们要做的就是 Omit 的变种，不是 Omit 某些 key，而是 Omit 值为非函数的 key。

由于 Omit 非函数实际就就是 Pick 函数，并且无需显式指定 key，因此我们的泛型只接受一个参数即可。 于是模仿官方的 Pick 写出了如下代码：

// 获取值为函数的 key，形如： 'funcKeyA' | 'funcKeyB'  
type PickFuncKeys<T> = {  
 [K in keyof T]: T[K] extends Function ? K : never;  
}[keyof T];  
  
// 获取值为函数的 key value 对，形如： { 'funcKeyA': ..., 'funKeyB': ...}  
type PickFunc<T> = Pick<T, PickFuncKeys<T>>;

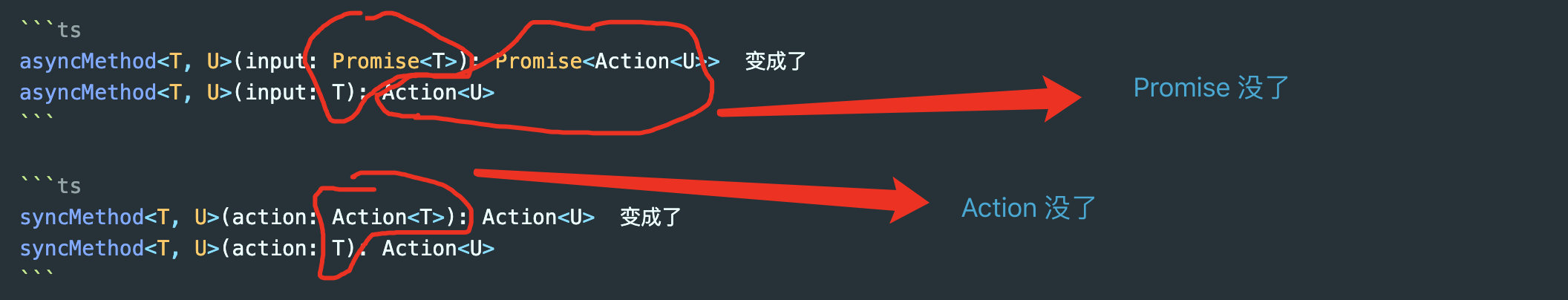
使用效果：

interface Todo {  
 title: string;  
 description: string;  
 addTodo(): string;  
}  
  
type AddTodo = PickFunc<Todo>;  
  
const todo: AddTodo = {  
 addTodo() {  
 return "关注脑洞前端~";  
 },  
};  
  
type ADDTodoKey = PickFuncKeys<Todo>; // 'addTodo'

可以看出，PickFunc 只提取了函数属性，忽略了非函数属性。

### 如何转换函数类型签名

我们再来回顾一下题目要求：



也就是我们需要知道**怎么才能提取 Promise 和 Action 泛型中的值**。

实际上这两个几乎一样，会了一个，另外一个也就会了。我们先来看下 Promise。

从：

(arg: Promise<T>) => Promise<U>

变为：

(arg: T) => U;

如果想要完成这个需求，需要借助infer。只需要在类型前加一个关键字前缀 infer，TS 会将推导出的类型自动填充进去。

infer 最早出现在此 [官方 PR](https://github.com/Microsoft/TypeScript/pull/21496) 中，表示在 extends 条件语句中待推断的类型变量。

简单示例如下：

type ParamType<T> = T extends (param: infer P) => any ? P : T;

在这个条件语句 T extends (param: infer P) => any ? P : T 中，infer P 表示待推断的函数参数。

整句表示为：如果 T 能赋值给 (param: infer P) => any，则结果是 (param: infer P) => any 类型中的参数 P，否则返回为 T。

一个更具体的例子：

interface User {  
 name: string;  
 age: number;  
}  
  
type Func = (user: User) => void;  
  
type Param = ParamType<Func>; // Param = User  
type AA = ParamType<string>; // string

这些知识已经够我们用了。 更多用法可以参考 [深入理解 TypeScript - infer](https://jkchao.github.io/typescript-book-chinese/tips/infer.html#%E4%BB%8B%E7%BB%8D) 。

根据上面的知识，不难写出如下代码：

type ExtractPromise<P> = {  
 [K in PickFuncKeys<P>]: P[K] extends (  
 arg: Promise<infer T>  
 ) => Promise<infer U>  
 ? (arg: T) => U  
 : never;  
};

提取 Action 的 代码也是类似：

type ExtractAction<P> = {  
 [K in keyof PickFunc<P>]: P[K] extends (  
 arg: Action<infer T>  
 ) => Action<infer U>  
 ? (arg: T) => Action<U>  
 : never;  
};

至此我们已经解决了全部两个问题，完整代码见下方代码区。

## 关键点

* 泛型
* extends 做类型约束
* infer 做类型提取
* 内置基本范型的使用和实现

## 代码

我们将这几个点串起来，不难写出如下最终代码：

type ExtractContainer<P> = {  
 [K in PickFuncKeys<P>]:  
 P[K] extends (arg: Promise<infer T>) => Promise<infer U> ? (arg: T) => U :  
 P[K] extends (arg: Action<infer T>) => Action<infer U> ? (arg: T) => Action<U> :  
 never  
type Connect = (module: EffectModule) => ExtractContainer<EffectModule>

完整代码在我的 [Gist](https://gist.github.com/azl397985856/5aecb2e221dc1b9b15af34680acb6ccf) 上。

## 总结

我们先对问题进行定义，然后分解问题为：1. 如何将非函数属性去掉, 2. 如何转换函数类型签名。最后从分解的问题，以及基础泛型工具入手，联系到可能用到的语法。

这个题目不算难，最多只是中等。但是你可能也看出来了，其不仅仅是考一个语法和 API 而已，而是考综合实力。这点在其他四道题体现地尤为明显。这种考察方式能真正考察一个人的综合实力，背题是背不来的。我个人在面试别人的时候也非常喜欢问这种问题。

只有**掌握基础 + 解决问题的思维方法**，面对复杂问题才能从容不迫，手到擒来。

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



知乎专栏【 [Lucifer - 知乎](https://www.zhihu.com/people/lu-xiao-13-70 " Lucifer - 知乎")】

点关注，不迷路！