

- 1. TypeScript
 - 1.1. 什么是 TypeScript 🧬
 - 1.2. 安装 TypeScript 🔧
 - 1.3. 项目中使用 TypeScript 📦
 - 1.3.1. 从零搭建一个 TS 项目
 - 1.3.2. 在 React 项目中使用 TypeScript
 - 1.4. TypeScript 知识点 🤔
 - 1.4.1. 常用类型 🧩
 - 1.4.2. 类型断言 🔍
 - 1.4.3. 接口 🔌
 - 1.4.4. 范型(Generics) (🔥 玩转 ts 必须掌握)
 - 1.4.4.1. 🚀 范型函数
 - 1.4.4.2. 🚀 范型接口
 - 1.4.4.3. 🚀 范型类
 - 1.4.4.4. 🚀 类型约束
 - 1.4.4.5. 🚀 在范型中使用类类型
 - 1.4.4.6. 🚀 范型操作符
 - 1.4.5. 🚀 类 (Class)
 - 1.4.6. 🚀 Utility Types (🔥)

1. TypeScript

1.1. 什么是 TypeScript 🧬

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，扩展了 Javascript 的语法。发展至今，已经成为大型项目的标配，其提供的静态类型系统，大大增强了代码的可读性以及可维护性；同时，它提供最新和不断发展的 JavaScript 特性，能让我们建立更健壮的组件。

npm 下载量 📈

背景

TypeScript 起源于使用 Javascript 开发的大型项目。由于 Javascript 语言本身的局限性，难以胜任和维护大型项目开发。因此微软开发了 TypeScript，使得其能够胜任开发大型项目。2012 年 10 月，微软发布了首个公开版本的 TypeScript，2013 年 6 月 19 日，在经历了一个预览版之后微软正式发布了正式版 TypeScript。当前最新版本为 TypeScript 4.0


TypeScript 和 Javascript 的区别

TypeScript 是 Javascript 的超集，扩展了 Javascript 的语法，因此现有的 Javascript 代码可与 TypeScript 一起工作无需任何修改，TypeScript 通过类型注解提供编译时的静态类型检查。

TypeScript 可处理已有的 Javascript 代码，并只对其中的 TypeScript 代码进行编译。

有什么特性/功能

- 类型注解和编译时类型检查
- 类型推断
- 类型擦除
- 类型组合
- 结构化类型系统
- 泛型编程

等等很多强大的功能....  点我点我

1.2. 安装 TypeScript

安装 TypeScript

```
npm install -g TypeScript
```

安装完成后我们就可以使用 TypeScript 编译器，名称叫 tsc，可将编译结果生成 js 文件

编译代码

```
tsc demo.ts
```

实时编译文件：

```
tsc -w demo.ts
```

编译成功，就会在相同目录下生成一个同名 js 文件

1.3. 项目中使用 TypeScript

工欲善其事，必先利其器。 ---- 《论语·卫灵公》

在项目中使用 typescript 之前，我们得先拥有一个支持 typescript 的编辑器，利其斧，善其事。有了编辑器的支持，才能事半功倍。这里强烈推荐 vscode，本身vscode就是ts写的一个electron应用，对ts的支持度很好。如果用其它编辑器可以安装对应的ts插件。

1.3.1. 从零搭建一个 TS 项目

1. 创建项目目录，并在项目根目录下执行 `npm init` 初始化项目。
2. 安装全局 ts 依赖 typescript 和 tslint

```
npm install typescript tslint -g
```

3. 初始化 tsconfig.json 文件

```
tsc --init #会初始化一个tsconfig.json 的配置文件
```

4. 安装 webpack，用于打包项目

```
npm install webpack webpack-cli webpack-dev-server -D
```

5. 安装 ts-loader,让 webpack 能够处理 ts 文件

```
npm install ts-loader -D
```

6. 安装 cross-env，可以配置在 package.json 的 scripts 配置指令传参

```
npm install cross-env -D
```

7. 在 package.json 中配置打包和运行的 scripts

```
"scripts": {  
  "start": "cross-env NODE_ENV=development webpack-dev-server --  
config ./webpack.config.js",  
  "build": "cross-env NODE_ENV=production webpack --config  
./webpack.config.js"  
}
```

8. 在项目中安装 ts 依赖 typescript 和 tslint

```
npm install typescript tslint
```

9. 配置 ts 检测, 通过 tslint -- init 初始化 tslint.json 配置文件

上面的步骤只是搭建了一个很基础的支持 ts 的项目, 很多细节需要实际操作时遇到的时候逐个针对。

对于 tsconfig 中有一个 lib 配置项, 如果我们的项目有用到 dom 特带的 api, 我们需要配置:

```
"lib":["DOM"]
```

如果我们的 ts 编译的目标是 es5 但是又用到一些新特性, 例如 Promise, 则在 lib 中在添加一个:

```
"lib":["es2015","DOM"]
```

通常情况下, 我们大概率会应用到第三方模块, 例如 jquery, jquery 本身不是 ts 编写的模块, 在使用的时候会报类型错误。这种情况下我们可以:

```
npm install @types/jquery
```

通过以上命令安装相关声明，或者自己定义一份 `.d.ts` 文件。`@types` 里定义很多第三方的类型声明文件。

1.3.2. 在 React 项目中使用 TypeScript

在 React 项目中使用 TypeScript，可以通过 React 官方的脚手架 `create-react-app` 创建一个支持 TypeScript 环境的项目,也可以对已有的 React 项目进行改造。

1.项目安装 TypeScript

要创建一个支持 TypeScript 的 Create React App 项目，可以运行：

```
npx create-react-app my-app --template TypeScript
# or
yarn create react-app my-app --template TypeScript
```

这里推荐使用 `npx` ,不推荐在全局安装 `npm install -g create-react-app` 命令，通过 `npx` 可以保证每次用的 `create-react-app` 是最新版本的，能保证拥有最新的特性。

要创建一个已有的项目支持TypeScript，需要执行一下命令：

```
npm install --save TypeScript @types/node @types/react @types/react-dom
@types/jest
# or
yarn add TypeScript @types/node @types/react @types/react-dom
@types/jest
```

2.替换文件后缀

执行完命令后，将文件重命名为 TypeScript 文件（例如 `src/index.js` 改成 `src/index.tsx` 或者 `index.ts`），

然后重启一下服务 `yarn run start` .重启项目后会生成 `tsconfig.json` 配置文件，可以在此基础基础上进行配置。

3.配置 tsconfig.path.json

注意： 下面的配置项是示例， 具体的配置， 根据项目实际情况配置。

tsconfig.path.json 这个文件名是自定义的， 你可以定义成 tsconfig.xxx.json 文件,但一定要是 json 格式， 用于单独设置 TypeScript 的 paths配置， 需要在 tsconfig.json 中 extends 该文件,如果不创建该文件， 则可以在 tsconfig.json 中添加同样的配置， 这里分开是为了更好的维护。

设置 paths 项： 通常我们在 webpack 中用 @ 符号配置了路径别名， 在非 TypeScript 项目中， 可以正常识别， 但是在 TypeScript 项目中， 需要在 paths 配置项添加说明， ts 才能识别@的意思。

设置 baseUrl 来告诉编译器到哪里去查找模块。 所有非相对模块导入都会被当做相对于 baseUrl 。

baseUrl 的值由以下两者之一决定：

- 命令行中 baseUrl 的值（如果给定的路径是相对的， 那么将相对于当前路径进行计算）
- tsconfig.json 里的 baseUrl 属性（如果给定的路径是相对的， 那么将相对于 tsconfig.json 路径进行计算）

注意： 相对模块的导入不会被设置的 baseUrl 所影响， 因为它们总是相对于导入它们的文件。

```
{
  "compilerOptions": {
    "baseUrl": "src",
    "paths": {
      "@/*": ["./*"]
    }
  }
}
```

4.更多配置项 tsconfig.json

默认情况下, 通过 create-react-app 生成的 TypeScript 项目已经有了一份基础的 tsconfig 配置, 对于更多配置项可以通过[点此参考官方文档](#)来查看, 根据项目的需求选择合适的配置。

```
{
  "extends": "../tsconfig.path.json",
  "compilerOptions": {
    "target": "es5",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "noEmit": true,
    "jsx": "react-jsx",
    "isolatedModules": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "downlevelIteration": true,
    "noFallthroughCasesInSwitch": true,
    "suppressImplicitAnyIndexErrors": true
  },
  "include": ["src"]
}
```

1.4. TypeScript 知识点 🤔

1.4.1. 常用类型 🧩

1. 字符串，数字，布尔值

在书写类型名称的时候，类型名称 `String`、`Number` 和 `Boolean`（以大写字母开头）虽然是有效的，但是不推荐用大写开头的，尽量用小写的 `string`、`boolean`、`number`。

```
let username: string = "foxyuan"; //字符串
let id: number = 123456; //数字
let isHappy: boolean = true; //布尔值
```

2. 数组类型

声明一个数组类型时，可以通过两种方式定义：

```
let arr: number[] = [1, 2, 3];
let arr2: Array<number> = [1, 2, 3]; //范型写法
```

3. `any`，`void`，`null`，`undefined`

`any` 是 typescript 的特殊类型，当你不想确切的指定具体的类型，但是也不想编译器报错，那么使用 `any` 则可以解决这个问题。但是虽然在编译阶段不会报错，但是在运行阶段可能导致程序出错。在项目能不用 `any` 则不用。

```
let somewhat: any = 1;
somewhat = "oh no";
somewhat = false;
```

`void` 表示一个空值，常用于函数返回值为空。给变量定一个 `void` 类型，只能赋值 `null` 或 `undefined`

```
let unusable: void = undefined;
```


`null` 和 `undefined` 在 `typescript` 中，这两个都有各自的类型，当 `tsconfig` 中配置项 `strictNullChecks` 为 `true` 时，`null` 和 `undefined` 只能赋值给 `void` 和自身。

```
let u: undefined = undefined;
let n: null = null;
```

5. `never` , `unknown`

`never` 表示永不存在的值的类型。有些场景值会永不存在，比如一个函数执行时抛出了异常，那么这个函数变永远不会有值了（因为抛出异常会直接中断程序运行，这样程序就运行不到返回值那一步了，即具有不可达的终点，也就永不存在返回了）。

`never` 也常用于收窄类型判断。

```
interface Foo {
  type: "foo";
}

interface Bar {
  type: "bar";
}

type All = Foo | Bar;

function handleValue(val: All) {
  switch (val.type) {
    case "foo":
      // 这里 val 被收窄为 Foo
      break;
    case "bar":
      // val 在这里是 Bar
      break;
    default:
      // val 在这里是 never
      const exhaustiveCheck: never = val;
      break;
  }
}
```

```
}
```

如果后续 `type All` 被其它维护的人修改了 `type All = Foo | Bar | Baz`, 忘记了在 `handleValue` 里面加上针对 `Baz` 的处理逻辑, 这个时候在 `default branch` 里面 `val` 会被收窄为 `Baz`, 导致无法赋值给 `never`, 产生一个编译错误。通过这个办法, 你可以确保 `handleValue` 总是穷尽 (exhaust) 了所有 `All` 的可能类型。

`unknown` 写代码的时候还不清楚会得到怎样的数据类型, 如服务器接口返回的数据, `JSON.parse()` 返回的结果等; 该类型相当于 `any`, 可以理解为官网指定的替代 `any` 类型的安全版本 (比 `any` 类型说, 更加安全, 在代码编译期间, 就能帮我们发现由于类型造成的问题, 因此在大多数的场景, 建议使用 `unknown` 类型替代 `any`。);

```
let val: unknown = 22;
val.push(33); //编译器报错: Property 'push' does not exist on type
'unknown'.
```

`never` 是所有类型的子类型 (最底) 以至于自身也不能赋值给自身, `unknown` 是所有类型的基类型 (顶级类型)

6. 枚举类型

枚举是一种类型, 可以作为变量的类型注解, 枚举中的成员只能访问, 不能赋值。枚举的成员是有值的, 默认从 0 开始自增, 也可以是字符串, 当是字符串时不会自增。

```
enum Error{
  serverError:500,
  networkError:404,
  msgError:'MESSAGE'
}
```

如果是数字枚举的时候, 可以反向读取。

更详细的枚举介绍:

1. 点此

2. 一文掌握 TS 枚举

7. 函数类型

函数在平常项目开发中应用的非常频繁。在 typescript 声明一个函数类型：

//1. 通过类型别名：

```
type Func = (arg: string) => string;
```

//2. 函数调用签名声明（使用接口定义），可以给函数声明属性，通过类型别名则不能。

```
interface Func {  
  (arg: string): string;  
  description: string;  
}
```

//3. 构造函数签名

```
type SomeConstructor = {  
  new (s: string): SomeObject;  
};
```

请注意，函数调用声明函数在参数和返回值之间是：。

[更多函数介绍点此](#)

8. 联合类型

联合类型是由两个或多个其他类型组成的类型，表示可以是其中任何一个类型的值。

```
type ValT = string | number | boolean | (arg:string|number|number)=>void
```

1.4.2. 类型断言 🔍

TypeScript 允许你覆盖它的推断，并且能以你任何你想要的方式分析它，这种机制被称为「类型断言」。TypeScript 类型断言用来告诉编译器你比它更了解这个类型，并且它不应该再发出错误。类型断言有两种方式 `as` 和 `<>`

as T 与 <T>

```
let foo: any;
let bar = <string>foo; // 现在 bar 的类型是 'string'
let bbar = foo as string; // 现在 bar 的类型是 'string'
```

当在 JSX 中推荐使用 `as`。

使用类型断言需要很小心，编译器并不会在编译阶段告诉你异常的地方，容易导致遗漏。

双重断言

类型断言，尽管我们已经证明了它并不是那么安全，但它也还是有用武之地。如下一个非常实用的例子所示，当使用者了解传入参数更具体的类型时，类型断言能按预期工作：

```
function handler(event: Event) {
  const mouseEvent = event as MouseEvent;
}
```

如下例子中的代码将会报错，尽管使用者已经使用了类型断言：

```
function handler(event: Event) {
  const element = event as HTMLElement; // Error: 'Event' 和
  'HTMLElement' 中的任何一个都不能赋值给另外一个
}
```

此时可以使用双重断言。首先断言成兼容所有类型的 `any`，编译器将不会报错：

```
function handler(event: Event) {
  const element = (event as any) as HTMLElement; // ok
}
```

当 `S` 类型是 `T` 类型的子集，或者 `T` 类型是 `S` 类型的子集时，`S` 能被成功断言成 `T`。这是为了在进行类型断言时提供额外的安全性，完全毫无根据的断言是危险的，如果你想这么做，你可以使用 `any`。

1.4.3. 接口

接口在 typescript 中常用作声明对象类型的另一种方式，很常用。TypeScript 只关心我们传递给对象的值的结构—它只关心它是否具有预期的属性。之所以称 TypeScript 为结构化类型系统，是因为只关心类型的结构和功能。

```
interface Animal {
  name: string;
}
interface Bear extends Animal {
  honey: boolean;
}
const bear = getBear();
bear.name;
bear.honey;
```

类型别名和接口之间的差异

类型别名和接口非常相似，在许多情况下，您可以在它们之间自由选择。接口的几乎所有特性都可以在类型别名中使用，关键的区别是不能给类型别名添加新属性，而接口总是可扩展的。

1.4.4. 泛型(Generics) (🔥 玩转 ts 必须掌握)

泛型 (Generics) 是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

在软件工程中，很重要的一部分就是构建一个不仅具有良好定义和一致性 API，而且还具有可重用性。像 Java 语言一样，Typescript 用于创建可重用组件的主要特性之一也是泛型，也就是说，能够创建可以处理多种类型而不是单个类型的组件。这允许用户使用这些组件并使用自己的类型。

如下面的代码，在没使用泛型特性前，我们通常会给 demo 函数使用具体的 number 类型或 any 类型。

使用 number 可以保证类型信息的存在，但是可复用性将会降低。

使用 any 可以接收更多的类型，但是会因此丢失类型信息，demo 函数始终返回 any 类型，使得程序的健壮性极度降低。

```
function demo(arg: number): number {  
    return arg;  
}
```

```
function demo(arg: any): any {  
    return arg;  
}
```

为了应对以上的情况，TypeScript 里的范型（Generic）便可以很好的解决。在调用的时候提供一个作用于类型的类型变量（例如 number），后续便可以获得该类型信息。使用范型和使用 any 很类似，都可以接受任意的类型，但是，any 会导致类型信息丢失，而范型则会保留类型信息。

```
function demo<Type>(arg: Type): Type {  
    return arg;  
}  
//调用  
let output = demo<string>("myString");  
//OR  
let output2 = demo("myString"); //这里依赖ts自身的类型推断根据上下文推断出类型。
```

如上代码，使用一对尖括号 <> 声明一个范型，在调用的时候，我们可以显示传递一个类型变量，而是交由 typescript 类型参数推断。这将使得代码更简洁和高可读性。

1.4.4.1. 🚀 范型函数

声明一个范型函数和非范型函数区别不大，唯一的就多了范型的定义。

```
//声明函数
function identity<Type>(arg: Type): Type {
    return arg;
}
//函数表达式
let myIdentity: <Type>(arg: Type) => Type = identity;
```

通过对象字面量的形式声明一个范型类型作为函数调用签名

```
{ <Type>(arg: Type): Type } //类似于范型接口
```

1.4.4.2. 🚀 范型接口

在接口中使用范型，在接口名后加上 `<Type>`。接口的所有成员都可以使用传递进来的类型。

```
interface GenericIdentityFn<Type> {
    (arg: Type): Type;
}
function identity<Type>(arg: Type): Type {
    return arg;
}
let myIdentity: GenericIdentityFn<number> = identity;
```

1.4.4.3. 🚀 范型类

跟范型接口的定义很类似，在类名后加上 `<Type>`。类的所有成员除了静态成员都可以使用传递进来的类型。

```
class GenericNumber<NumType> {
  zeroValue: NumType;
  add: (x: NumType, y: NumType) => NumType;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
  return x + y;
};
```

1.4.4.4. 🚀 类型约束

有时候我们不想使用任何类型，而是希望将此函数约束为使用同样具有特定属性的任何类型。只要类型有这个成员，我们就允许它。要做到这一点，我们必须将我们可以对类型进行约束。

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length);
  return arg;
}
```

在泛型约束中使用类型参数


```
function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key)
{
    return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a");
getProperty(x, "m"); //报错, m不在x的key中
```

1.4.4.5. 🚀 在范型中使用类类型

当使用泛型在 TypeScript 中创建工厂时，有必要通过构造函数引用类类型。

```
function create<Type>(c: { new (): Type }): Type {
    return new c();
}
```

```
class ZooKeeper {
    nametag: string;
}
class Animal {
    numLegs: number;
}
class Lion extends Animal {
    keeper: ZooKeeper;
}
function createInstance<A extends Animal>(c: new () => A): A {
    return new c();
}

createInstance(Lion).keeper.nametag;
```

1.4.4.6. 🚀 范型操作符

- keyof

keyof 运算符采用对象类型并生成其 key 的 string 或 number 的字面并集：

```
type Arrayish = { [n: number]: unknown };
type A = keyof Arrayish;
//   ^ = type A = number

type Mapish = { [k: string]: boolean };
type M = keyof Mapish;
//   ^ = type M = string | number 这里是因为[0]与["0"]在对象key中总是相等的, number会被强制转换为string
```

- typeof

TypeScript 添加一个 typeof 运算符，您可以在类型上下文中使用该运算符来引用变量或属性的类型：

```
let s = "hello";
let n: typeof s;
```

- ReturnType<T>

接受函数类型并生成其返回类型。

```
type Predicate = (x: unknown) => boolean;
type K = ReturnType<Predicate>;
//   ^ = type K = boolean

function f() {
    return { x: 10, y: 3 };
}
type P = ReturnType<typeof f>;
```

- 索引访问类型

我们可以使用索引访问类型来查找其他类型的特定属性：

```
type Person = { age: number; name: string; alive: boolean };
type Age = Person["age"];
// ^ = type Age = number
```

索引类型本身就是一种类型，因此我们可以完全使用 unions、keyof 或其他类型：

```
type I1 = Person["age" | "name"];
// ^ = type I1 = string | number

type I2 = Person[keyof Person];
// ^ = type I2 = string | number | boolean

type AliveOrName = "alive" | "name";
type I3 = Person[AliveOrName];
// ^ = type I3 = string | boolean
```

当索引的属性不存在时会报错提示。除了索引声明的类型外，还可以对对象进行类型的获取。

```
const MyArray = [
  { name: "Alice", age: 15 },
  { name: "Bob", age: 23 },
  { name: "Eve", age: 38 },
];

type Person = typeof MyArray[number];
// ^ = type Person = {
//     name: string;
//     age: number;
// }
type Age = typeof MyArray[number]["age"];
// ^ = type Age = number
```

■ 条件类型

条件类型看起来很像 js 中的条件表达式,但是也只是看起来像。

```
SomeType extends OtherType ? TrueType : FalseType;
```

此能力让类型定义变的更加灵活，需要注意： `extends` 运用在 `type` 和 `class` 中时完全是两种作用的效果。当 `extends` 左边的类型可分配给右边的类型时，您将获得第一个分支（“true”分支）中的类型；否则您将获得后一个分支（“false”分支）中的类型。

▪ 优化重载场景

```
interface IdLabel {
  id: number;
}
interface NameLabel {
  name: string;
}
function createLabel(id: number): IdLabel;
function createLabel(name: string): NameLabel;
function createLabel(nameOrId: string | number): IdLabel |
NameLabel;
function createLabel(nameOrId: string | number): IdLabel |
NameLabel {
  throw "unimplemented";
}
```

对上面重载进行优化：

```
type NameOrId<T extends number | string> = T extends number
  ? IdLabel
  : NameLabel;

function createLabel<T extends number | string>(idOrName: T):
NameOrId<T> {
  throw "unimplemented";
}

let a = createLabel("TypeScript");
// ^ = let a: NameLabel
```

▪ 条件类型约束

就像使用类型保护缩小范围可以为我们提供更具体的类型一样，条件类型的 `true` 分支将进一步约束泛型。

```
type MessageOf<T extends { message: unknown }> = T["message"];

interface Email {
  message: string;
}

interface Dog {
  bark(): void;
}

type EmailMessageContents = MessageOf<Email>;
```

上面的代码我们只能拿到满足约束条件的类型。但是如果我们想要 `MessageOf` 可以接收任意类型，在满足约束条件下，返回 `T['message']`，不满足约束条件下，返回 `never` 或其它，我们可以将约束条件移到外面，并使用条件类型。

```
type MessageOf<T> = T extends { message: unknown } ?
  T["message"] : never;
```

▪ 条件类型推断 `infer`

`infer` 相当与一个占位符，常常结合 `extends ? :` 使用，可以取到所占位置的类型。

```

type GetReturnType<Type> = Type extends (...args: never[]) =>
infer Return
  ? Return
  : never;

type Num = GetReturnType<() => number>;
//   ^ = type Num = number

type Str = GetReturnType<(x: string) => string>;
//   ^ = type Str = string

type Bools = GetReturnType<(a: boolean, b: boolean) =>
boolean[]>;
//   ^ = type Bools = boolean[]

```

一道来源于 leetcode 招聘的 infer 应用

```

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",
    };
}
}

// 修改 Connect 的类型, 让 connected 的类型变成预期的类型
type Connect = (module: EffectModule) => any;

const connect: Connect = (m) => ({
    delay: (input: number) => ({
        type: "delay",
        payload: `hello 2`,
    }),
    setMessage: (input: Date) => ({
        type: "set-message",
        payload: input.getMilliseconds(),
    }),
});

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

export const connected: Connected = connect(new EffectModule());

```

题解

//利用类型分发和class可以取值来做, 如果是函数, 那就提取, 否则就不提取
 //这里同时利用value如果是never 则keyof就不会返回。
 //这段其实挺有启发性, 因为很多时候, 都想搞个循环判断类型, 然后进行选择, 这就是个很好的范例。

```

type MethodName<T> = {
    [F in keyof T]: T[F] extends Function ? F : never;
}

```

```

}[keyof T];
type EE = MethodName<EffectModule>;

type asyncMethod<T, U> = (input: Promise<T>) =>
Promise<Action<U>>;
type asyncMethodConnect<T, U> = (input: T) => Action<U>;
type syncMethod<T, U> = (action: Action<T>) => Action<U>;
type syncMethodConnect<T, U> = (action: T) => Action<U>;

type EffectMethodAssign<T> = T extends asyncMethod<infer U,
infer V>
  ? asyncMethodConnect<U, V>
  : T extends syncMethod<infer U, infer V>
  ? syncMethodConnect<U, V>
  : never;

type Connect = (
  module: EffectModule
) => {
  [F in MethodName<typeof module>]: EffectMethodAssign<typeof
module[F]>;
};

```

<https://blog.csdn.net/yehuozhili/article/details/108253532>

<https://juejin.cn/post/6844904067420913678>

■ 条件分发类型（可分配条件类型）

当条件类型作用于泛型类型时，当给定一个并集类型时，它们就成为可分配的。

```

type ToArray<Type> = Type extends any ? Type[] : never;

```

如果在 ToArray 中插入一个联合类型，那么条件类型将应用于该联合的每个成员。


```

type ToArray<Type> = Type extends any ? Type[] : never;

type StrArrOrNumArr = ToArray<string | number>;
//   ^ = type StrArrOrNumArr = string[] | number[]

```

很明显此时返回的类型是将 `string | number` 返回成 `string[] | number[]`。如果我们想要返回的结果是 `(string | number)[]`，此时我们可以给 `extends` 两边的类型用 `[]` 括起来。

```

type ToArrayNonDist<Type> = [Type] extends [any] ? Type[] :
never;

// 'StrOrNumArr' is no longer a union.
type StrOrNumArr = ToArrayNonDist<string | number>;
//   ^ = type StrOrNumArr = (string | number)[]

```

■ 映射类型

某些场景下我们可能想要利用已经声明的类型中的部分属性来构建新的类型。

映射类型使用通过 `keyof` 创建的 `union` 来迭代一个类型的 `key` 以创建另一个类型的泛型类型

```

type OptionsFlags<Type> = {
  [Property in keyof Type]: boolean;
};

```

■ 映射修饰符

有两个附加的修饰符可以在映射期间应用：`readonly` 和 `?` 分别影响可变性和可选性。

可以通过在前面加上 `-` 或 `+`，来删除或添加这些修饰符。如果不添加前缀，则默认为 `+`。

```
// Removes 'readonly' attributes from a type's properties
type CreateMutable<Type> = {
  -readonly [Property in keyof Type]: Type[Property];
};

// Removes 'optional' attributes from a type's properties
type Concrete<Type> = {
  [Property in keyof Type]-?: Type[Property];
};
```

在 TypeScript 4.1 及更高版本中，可以使用映射类型中的 `as` 子句重新映射映射映射类型中的键。

```
//使用模版字符
type Getters<Type> = {
  [Property in keyof Type as `get${Capitalize<
    string & Property
  >}`]: () => Type[Property];
};

//删除指定属性
type RemoveKindField<Type> = {
  [Property in keyof Type as Exclude<Property, "kind">]:
    Type[Property];
};
```

1.4.5. 🚀 类 (Class)

ts 全面支持 ES2015 的 `class` 关键字。在面向对象语言如 java 中，我们会经常接触到类，面向对象的三大特性封装、继承、多态。使用类，我们可以写出健壮高可复用性的代码。声明一个类：

```
class Point {}
```

默认情况下，声明的类属性都是可以覆盖的。在属性名称前加上 `readonly` 修饰符，则改属性将不能被覆盖。

构造函数 🐶

类构造函数与函数非常相似。可以添加带有类型注释、默认值和重载的参数。构造函数不能传递类型参数，不能添加返回类型，构造函数返回的永远都是类实例类型

```
class Point {
  x: number;
  constructor(x = 0, y = 0) {
    this.x = x;
  }
}

//重载构造函数
class Point {
  // Overloads
  constructor(x: number, y: string);
  constructor(s: string);
  constructor(xs: any, y?: any) {
    // TBD
  }
}
```

调用父类构造函数 `super()` 🐶

`super` 关键字常常用于调用父类的成员属性或方法。eg: `super.age`。 `super()` 则是调用父类的构造函数，当我们在子类构造函数中使用 `this` 关键字的是否，需要在使用 `this` 之前调用 `super()` 方法。

索引签名 🐶

类可以声明索引签名；它们的工作方式与其他对象类型的索引签名相同，这个特性用的比较少

```
class MyClass {  
  [s: string]: boolean | ((s: string) => boolean);  
  check(s: string) {  
    return this[s] as boolean;  
  }  
}
```

类继承

TypeScript 中的类可以实现多个接口，继承类只能继承一个。继承接口用 `implements` ,继承基类用 `extends` 。派生类可以重写基类字段或属性。用 `super` . 的方式访问基类方法或属性。

```
//继承多接口  
interface A {}  
interface B {}  
class C implements A, B {}
```

在实例化派生类的时候，推荐用基类作为实例的类型。

```
class Base {}  
class Derive extends Base {}  
  
const dins: Base = new Derive();
```

初始化顺序

类初始化顺序是：

```
基类字段已初始化  
基类构造函数运行  
派生类字段已初始化  
派生类构造函数运行
```

继承内置类型

当我们继承内置类型（eg. Error, Array），且编译目标为 ES5 的时候特别需要注意一个地方。在 ES2015 中，在构造函数中调用 `super (...)` 会隐式地将 `this` 值替换。对于

`Error`, `Array` 等原生 `class`，他们的 `constructor` 使用了 `new.target` 来调整原型链。但在 `es5` 中无法保证 `new.target` 一定存在，所以会导致继承出来的 `class` 在原型链上缺失。

`new.target` 属性允许你检测函数或构造方法是否是通过 `new` 运算符被调用的。在通过 `new` 运算符被初始化的函数或构造方法中，`new.target` 返回一个指向构造方法或函数的引用。在普通的函数调用中，`new.target` 的值是 `undefined`。

如果 `ts` 编译目标是 `ES5`，我们需要做一定的处理。

```
class MsgError extends Error {
  constructor(m: string) {
    super(m);
  }

  sayHello() {
    return "hello " + this.message;
  }
}

new MsgError("good") instanceof MsgError; //ES5 target下为false
console.log(new MsgError("good").sayHello()); //ES5 target下为undefined
```

调整原型链,手动调整 `this` 的指向，从原先的指向 `Error` 的，重新指向 `MsgError`。

```
class MsgError extends Error {
  constructor(m: string) {
    super(m);

    // Set the prototype explicitly.
    Object.setPrototypeOf(this, MsgError.prototype);
  }

  sayHello() {
    return "hello " + this.message;
  }
}
```

MsgError 的任何子类也必须手动设置原型。对于不支持 `Object.setPrototypeOf` 对象，则可以使用 `__proto__` 。

公共，私有与受保护的修饰符

修饰符	描述
public	类成员的默认可见性是 public。公共成员可以在任何地方访问
protected	受保护的成员仅对声明它们的类的子类可见。
private	private 类似于 protected，但不允许从子类访问成员

静态成员

这些成员与类的特定实例没有关联。可以通过类构造函数对象本身访问它们。

静态成员还可以使用相同的 public、protected 和 private 可见性修饰符。

```

class MyClass {
    static x = 0;
    static printX() {
        console.log(MyClass.x);
    }
}
console.log(MyClass.x);
MyClass.printX();

```

覆盖函数原型的属性通常是不安全的/不可能的。因为类本身就是可以用 new 调用的函数，所以不能使用某些静态名称。函数属性（如 name 、 length 和 call ）不能定义为静态成员

```

class S {
    static name = "S!";
    //Static property 'name' conflicts with built-in property
    'Function.name' of constructor function 'S'.
}

```

泛型类的静态成员永远不能引用类的类型参数。静态成员是共享的，不同的地方实例化同一个类，传进来的的范型类型可能不一样，这是 很不好的。下面的代码将会编译错误

```

class Box<Type> {
    static defaultValue: Type;
    //Static members cannot reference class type parameters.
}

```

参数属性

TypeScript 提供了特殊的语法，用于将构造函数参数转换为具有相同名称和值的类属性。这些称为参数属性，通过在构造函数参数前面加上可见性修饰

符 public 、 private 、 protected 或 readonly 来创建。结果字段获取这些修饰符：

```
class Params {
  constructor(
    public readonly x: number,
    protected y: number,
    private z: number
  ) {
    // No body necessary
  }
}
```

1.4.6. 🚀 Utility Types (🔥)

Partial<Type>

构造一个类型，该类型的所有属性都设置为 `optional` 。返回一个表示给定类型的所有子集的类型。

```
interface Todo {
  title: string;
  description: string;
}
type PTodo = Partial<Todo>;
```

Required<Type>

构造一个类型，该类型包含类型设置为 `required` 的所有属性。与 `partial<T>` 相反。

```
interface Props {
  a?: number;
  b?: string;
}
type RProps = Required<Props>;
```

Readonly<Type>

构造一个类型，该类型的所有属性都设置为 `readonly`，这意味着无法重新修改所构造类型的属性。


```
interface Todo {  
  title: string;  
}  
  
type RTodo: Readonly<Todo>
```

Record<Keys, Type>

构造一个对象类型，其属性键是 `keys`，属性值是 `Type`。可用于将一个类型的属性映射到另一个类型。

```
interface CatInfo {  
  age: number;  
  breed: string;  
}  
  
type CatName = "miffy" | "boris" | "mordred";  
  
const cats: Record<CatName, CatInfo> = {  
  miffy: { age: 10, breed: "Persian" },  
  boris: { age: 5, breed: "Maine Coon" },  
  mordred: { age: 16, breed: "British Shorthair" },  
};
```

Pick<Type, Keys>

通过从 `Type` 中选取一组属性 `keys`（字符串字面量或字符串字面量的并集）来构造类型。

```
interface Todo {  
  title: string;  
  description: string;  
  completed: boolean;  
}  
  
type TodoPreview = Pick<Todo, "title" | "completed">;
```

Omit<Type, Keys>

通过从 Type 中选取所有属性，然后删除 keys （字符串字面量或字符串字面量的并集）来构造类型。

```
interface Todo {  
  title: string;  
  description: string;  
  completed: boolean;  
  createdAt: number;  
}  
  
type TodoPreview = Omit<Todo, "description">;
```

Exclude<Type, ExcludedUnion>

通过从 Type 中排除可分配给 ExcludedUnion 的所有联合成员来构造类型。

```
type T0 = Exclude<"a" | "b" | "c", "a">;  
//    ^ = type T0 = "b" | "c"  
type T1 = Exclude<"a" | "b" | "c", "a" | "b">;  
//    ^ = type T1 = "c"  
type T2 = Exclude<string | number | (() => void), Function>;  
//    ^ = type T2 = string | number
```

Extract<Type, Union>

通过从 Type 中提取可分配给 Union 的所有联合成员来构造类型。

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">;  
//    ^ = type T0 = "a"  
type T1 = Extract<string | number | (() => void), Function>;  
//    ^ = type T1 = () => void
```

NonNullable<Type>

通过从 Type 中排除 null 和 undefined 来构造类型。

```
type T0 = NonNullable<string | number | undefined>;  
//    ^ = type T0 = string | number  
type T1 = NonNullable<string[] | null | undefined>;  
//    ^ = type T1 = string[]
```

Parameters<Type>

从函数类型的参数类型构造一个元祖（tuple）或数组类型

```
declare function f1(arg: { a: number; b: string }): void;  
  
type T0 = Parameters<() => string>;  
//    ^ = type T0 = []  
type T1 = Parameters<(s: string) => void>;  
//    ^ = type T1 = [s: string]  
type T2 = Parameters<<T>(arg: T) => T>;  
//    ^ = type T2 = [arg: unknown]
```

ConstructorParameters<Type>

从构造函数类型的参数类型构造一个元祖（tuple）或数组类型

```
type T0 = ConstructorParameters<ErrorConstructor>;  
//    ^ = type T0 = [message?: string]  
type T1 = ConstructorParameters<FunctionConstructor>;  
//    ^ = type T1 = string[]  
type T2 = ConstructorParameters<RegExpConstructor>;  
//    ^ = type T2 = [pattern: string | RegExp, flags?: string]  
type T3 = ConstructorParameters<any>;  
//    ^ = type T3 = unknown[]
```

ReturnType<Type>

构造由函数类型的返回类型组成的类型。

```
declare function f1(): { a: number; b: string };
```

```
type T0 = ReturnType<() => string>;
```

```
//    ^ = type T0 = string
```

除了以上这些，还有很多官方实现的类型工具[`InstanceType<Type>`],
[`ThisParameterType<Type>`], [`OmitThisParameter<Type>`], [`ThisType<Type>`],
[`Lowercase<StringType>`], [`Uppercase<StringType>`], [`Capitalize<StringType>`],
[`Uncapitalize<StringType>`].这些都可以在[官方文档](#)中可以看到具体的使用案例