TS2

交叉类型(1.ts)

交叉类型是将多个类型合并为一个类型。 这让我们可以把现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性。
在 JavaScript 中，混入是一种非常常见的模式，在这种模式中，你可以从两个对象中创建一个新对象，新对象会拥有着两个对象所有的功能。

联合类型

在 JavaScript 中，你希望属性为多种类型之一，如字符串或者数组。
这就是联合类型所能派上用场的地方（它使用 | 作为标记，如 string | number）。

```
function formatCommandline(command: string[] | string) {
  let line = '';
  if (typeof command === 'string') {
    line = command.trim();
  } else {
    line = command.join(' ').trim();
  }
}
```

类型别名

```
type some = boolean | string

const b: some = true // ok
const c: some = 'hello' // ok
const d: some = 123 // 不能将类型“123”分配给类型“some”

type Tree<T> = {
    value: T;
    left: Tree<T>;
    right: Tree<T>;
}
```

interface 只能用于定义对象类型，而 type 的声明方式除了对象之外还可以定义交叉、联合、原始类型等，类型声明的方式适用范围显然更加广泛。
但是interface也有其特定的用处：

- interface 方式可以实现接口的 extends 和 implements
- interface 可以实现接口合并声明

```
type Alias = { num: number }
interface Interface {
    num: number;
}
declare function aliased(arg: Alias): Alias;
declare function interfaced(arg: Interface): Interface;
```

接口创建了一个新的名字,可以在其它任何地方使用,类型别名并不创建新名字,比如,错误信息就不会使用别名。

可辨识联合类型

字面量类型

字面量（Literal Type）主要分为 真值字面量类型（boolean literal types）,数字字面量类型（numeric literal types）,枚举字面量类型（enum literal types）,大整数字面量类型（bigInt literal types）和字符串字面量类型（string literal types）。

```
const a: 2333 = 2333 // ok
const ab : 0b10 = 2 // ok
const ao : 0o114 = 0b1001100 // ok
const ax : 0x514 = 0x514 // ok
const b : 0x1919n = 6425n // ok
const c : 'xiaomuzhu' = 'xiaomuzhu' // ok
const d : false = false // ok

const g: 'github' = 'pronhub' // 不能将类型“"pronhub"”分配给类型“"github"”
```

当字面量类型与联合类型结合的时候,用处就显现出来了,它可以模拟一个类似于枚举的效果:

```
type Direction = 'North' | 'East' | 'South' | 'West';

function move(distance: number, direction: Direction) {
    // ...
```

```
}
```

类型字面量

类型字面量(Type Literal)不同于字面量类型（Literal Type),它跟 JavaScript 中的对象字面量的语法很相似:

```
type Foo = {
  baz: [
    number,
    'xiaomuzhu'
  ];
  toString(): string;
  readonly [Symbol.iterator]: 'github';
  0x1: 'foo';
  "bar": 12n;
};
```

可辨识联合类型 (2.ts)

我们先假设一个场景,现在又两个功能,一个是创建用户即 `create`,一个是删除用户即 `delete`.
我们先定义一下这个接口,由于创建用户不需要id,是系统随机生成的,而删除用户是必须用到 id 的,那么代码如下:

```
interface Info {
    username: string
}

interface UserAction {
    id?: number
    action: 'create' | 'delete'
    info: Info
}
```

上面的接口是不是有什么问题?
是的,当我们创建用户时是不需要 id 的,但是根据上面接口产生的情况,以下代码是合法的:

```
const action:UserAction = {
    action:'create',
    id: 111,
```

```
    info: {
        username: 'xiaomuzhu'
    }
}
```

但是我们明明不需要 id 这个字段,因此我们得用另外的方法,这就用到了上面提到的「类型字面量」了:

```
type UserAction = {
    id: number
    action: 'delete'
    info: Info
} |
{
    action: 'create'
    info: Info
}
```

类型断言

比如初学者经常会遇到的一类问题:

```
const person = {};
person.name = 'xiaomuzhu'; // Error: 'name' 属性不存在于 '{}'
person.age = 20; // Error: 'age' 属性不存在于 '{}'
```

这个时候该怎么办？由于类型推断，这个时候 `person` 的类型就是 `{}`，根本不存在后添加的那些属性，虽然这个写法在js中完全没问题，但是开发者知道这个 `person` 实际是有属性的，只是一开始没有声明而已，但是 typescript 不知道啊，所以就需要类型断言了:

```
interface Person {
  name: string;
  age: number;
}
const person = {} as Person;
person.name = 'xiaomuzhu';
person.age = 20;
```

双重断言

```
interface Person {
```

```
        name: string;
        age: number;
}
const person = 'xiaomuzhu' as Person; // Error
const person = 'xiaomuzhu' as any as Person; // ok
```

类型守卫(3.ts)

instanceof、in

类型兼容性

结构类型

TypeScript 里的类型兼容性是基于「结构类型」的，结构类型是一种只使用其成员来描述类型的方式，其基本规则是，如果 x 要兼容 y，那么 y 至少具有与 x 相同的属性。x=y
我们做一个简单的实验，我们构建一个类 `Person`,然后声明一个接口 `Dog`， `Dog` 的属性 `Person` 都拥有，而且还多了其他属性，这种情况下 `Dog` 兼容了 `Person`。

```
class Person {
    constructor(public weight: number, public name: string, public born: string) {
}
interface Dog {
    name: string
    weight: number
}
let x: Dog
x = new Person(120, 'cxk', '1996-12-12') // OK
```

但反过来就不行

函数的类型兼容性(4.ts)

函数类型的兼容性判断，要查看 x 是否能赋值给 y，首先看它们的参数列表。
x 的每个参数必须能在 y 里找到对应类型的参数,注意的是参数的名字相同与否无所谓，只看它们的类型。

类的类型兼容性（5.ts）

泛型的类型兼容性

泛型本身就是不确定的类型,它的表现根据是否被成员使用而不同.

```
interface Person<T> {

}
let x : Person<string>
let y : Person<number>
x = y // ok
y = x // ok
```

由于没有被成员使用泛型,所以这里是没问题的。
那么我们再看下面:

```
interface Person<T> {
    name: T
}
let x : Person<string>
let y : Person<number>
x = y // 不能将类型“Person<number>”分配给类型“Person<string>”。
y = x // 不能将类型“Person<string>”分配给类型“Person<number>”。
```

is 关键字（6.ts）

可调用类型注解(7.ts)

高级类型之索引类型、映射类型、条件类型

索引类型

我们先看一个场景，现在我们需要一个 pick 函数，这个函数可以从对象上取出指定的属性，类似于 `lodash.pick` 方法。
JavaScript :

```
function pick(o, names) {
  return names.map(n => o[n]);
}
const user = {
```

```
    username: 'Jessica Lee',
    id: 460000201904141743,
    token: '460000201904141743',
    avatar: 'http://dummyimage.com/200x200',
    role: 'vip'
}
const res = pick(user, ['id'])
console.log(res) // [ '460000201904141743' ]
```

TypeScript 简陋版

```
interface Obj {
    [key: string]: any
}
function pick(o: Obj, names: string[]) {
    return names.map(n => o[n]);
}
```

高级框架版

```
type key = keyof T ==='username'|'id'...
function pick<T, K extends keyof T>(o: T, names: K[]): T[K][] {
    return names.map(n => o[n]);
}
const res = pick(user, ['token', 'id', ])
```

映射类型(8.ts)

我们有一个User接口，现在有一个需求是把User接口中的成员全部变成可选的，我们应该怎么做？难道要重新一个个`:`前面加上`?`,有没有更便捷的方法?
这个时候映射类型就派上用场了，映射类型的语法是`[K in Keys]`:

- K：类型变量，依次绑定到每个属性上，对应每个属性名的类型
- Keys：字符串字面量构成的联合类型，表示一组属性名（的类型）

```
type partial<T> = { [K in keyof T]?: T[K] }
```

条件类型

条件类型够表示非统一的类型,以一个条件表达式进行类型关系检测，从而在两种类型中选择其一:

```
T extends U ? X : Y
```

上面的代码可以理解为: 若 `T` 能够赋值给 `U`，那么类型是 `X`，否则为 `Y`,有点类似于JavaScript中的三元条件运算符.
比如我们声明一个函数 `f`,它的参数接收一个布尔类型,当布尔类型为 `true` 时返回 `string` 类型,否则返回 `number` 类型:

```
declare function f<T extends boolean>(x: T): T extends true ? string : number;

const x = f(Math.random() < 0.5)
const y = f(false)
const z = f(true)
```

条件类型就是这样,只有类型系统中给出充足的条件之后,它才会根据条件推断出类型结果.

条件类型与联合类型(9.ts)

条件类型有一个特性,就是「分布式有条件类型」,但是分布式有条件类型是有前提的,条件类型里待检查的类型必须是`naked type parameter`.
`naked type parameter`指的是**裸类型参数**,怎么理解?这个「裸」是指类型参数没有被包装在其他类型里,比如没有被数组、元组、函数、Promise等等包裹.

```
// 裸类型参数,没有被任何其他类型包裹即T
type NakedUsage<T> = T extends boolean ? "YES" : "NO"
// 类型参数被包裹的在元组内即[T]
type WrappedUsage<T> = [T] extends [boolean] ? "YES" : "NO";
```

这一部分比较难以理解,我们可以把「分布式有条件类型」粗略得理解为类型版的`map()方法`,然后我们再看一些实用案例加深理解.

条件类型与映射类型(10.ts)

在一些有要求TS基础的公司,设计工具类型是一个比较大的考点.

强大的infer关键字(11.ts,12.ts)

`infer` 是工具类型和底层库中非常常用的关键字,表示在 extends 条件语句中待推断的类型变量,相对而言也比较难理解,我们不妨从一个 typescript 面试题开始:
我们之前学过 `ReturnType` 用于获取函数的返回类型,那么你会如何设计一个 `ReturnType`?

`infer`非常强大,由于它的存在我们可以做出非常多的骚操作.
tuple转union,比如`[string, number] -> string | number`:

```
type ElementOf<T> = T extends Array<infer E> ? E : never;
```

```
type TTuple = [string, number];
type ToUnion = ElementOf<ATuple>; // string | number
```

常用工具类型解读

用 JavaScript 编写中大型程序是离不开 lodash 这种工具集的，而用 TypeScript 编程同样离不开类型工具的帮助，类型工具就是类型版的 lodash.
我们在本节会介绍一些类型工具的设计与实现，如果你的项目不是非常简单的 demo 级项目，那么在你的开发过程中一定会用到它们。
起初，TypeScript 没有这么多工具类型，很多都是社区创造出来的，然后 TypeScript 陆续将一些常用的工具类型纳入了官方基准库内。
比如 `ReturnType`、`Partial`、`ConstructorParameters`、`Pick` 都是官方的内置工具类型.
其实上述的工具类型都可以被我们开发者自己模拟出来,本节我们学习一下如何设计工具类型.

泛型

我们说过可以把工具类型类比 js 中的工具函数，因此必须有输入和输出，而在TS的类型系统中能担当类型入口的只有泛型.
比如`Partial`,它的作用是将属性全部变为可选.

```
type Partial<T> = { [P in keyof T]?: T[P] };
```

这个类型工具中,我们需要将类型通过泛型`T`传入才能对类型进行处理并返回新类型,可以说,一切类型工具的基础就是泛型.

类型递归(13.ts)

关键字

像`keyof`、`typeof`这种常用关键字我们已经了解过了,当然还有很常用的`Type inference`infer关键字的使用,还有之前的`Conditional Type`条件类型,现在主要谈一下另外一些常用关键字.
`+`、`-`这两个关键字用于映射类型中给属性添加修饰符,比如`-?`就代表将可选属性变为必选,`-readonly`代表将只读属性变为非只读.
比如TS就内置了一个类型工具`Required<T>`,它的作用是将传入的属性变为必选项:

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

常见工具类型

Omit

`Omit`这个工具类型在开发过程中非常常见,以至于官方在3.5版本正式加入了`Omit`类型.

要了解之前我们先看一下另一个内置类型工具的实现`Exclude<T>`:

```
type Exclude<T, U> = T extends U ? never : T;
type T = Exclude<1 | 2, 1 | 3> // -> 2
```

`Exclude` 的作用是从 `T` 中排除出**可分配**给 `U` 的元素.
这里的可分配即`assignable`,指可分配的,`T extends U`指T是否可分配给U
`Omit` = `Exclude` + `Pick`

```
type Omit<T, K> = Pick<T, Exclude<keyof T, K>>
type Foo = Omit<{name: string, age: number}, 'name'> // -> { age: number }
```

`Omit<T, K>`的作用是忽略`T`中的某些属性.

Merge

`Merge<O1, O2>`的作用是将两个对象的属性合并:

```
type O1 = {
    name: string
    id: number
}
type O2 = {
    id: number
    from: string
}
type R2 = Merge<O1, O2>
```

这个类型工具也非常常用,他主要有两个部分组成:
`Merge<O1, O2>` = `Compute<A>` + `Omit<U, T>`
`Compute`的作用是将交叉类型合并.即:

```
type Compute<A extends any> =
    A extends Function
    ? A
    : { [K in keyof A]: A[K] }
type R1 = Compute<{x: 'x'} & {y: 'y'}>
```

Merge的最终实现如下:

```
type Merge<O1 extends object, O2 extends object> =
    Compute<O1 & Omit<O2, keyof O1>>
```

Intersection

`Intersection<T, U>`的作用是取`T`的属性,此属性同样也存在与`U`.

```ts
type Props = { name: string; age: number; visible: boolean };
type DefaultProps = { age: number };

// Expect: { age: number; }
type DuplicatedProps = Intersection<Props, DefaultProps>;
```

实现

```ts
type Intersection<T extends object, U extends object> = Pick<
  T,
  Extract<keyof T, keyof U> & Extract<keyof U, keyof T>
>;
```

Overwrite

`Overwrite<T, U>`顾名思义,是用`U`的属性覆盖`T`的相同属性.

```ts
type Props = { name: string; age: number; visible: boolean };
type NewProps = { age: string; other: string };

// Expect: { name: string; age: string; visible: boolean; }
type ReplacedProps = Overwrite<Props, NewProps>
```

即:

```ts
type Overwrite<
  T extends object,
  U extends object,
  I = Diff<T, U> & Intersection<U, T>
> = Pick<I, keyof I>;
```

Mutable

将`T`的所有属性的`readonly`移除

```ts
type Mutable<T> = {
  -readonly [P in keyof T]: T[P]
}
```

Record

Record 允许从 Union 类型中创建新类型，Union 类型中的值用作新类型的属性。

```
type Car = 'Audi' | 'BMW' | 'MercedesBenz'
type CarList = Record<Car, {age: number}>

const cars: CarList = {
    Audi: { age: 119 },
    BMW: { age: 113 },
    MercedesBenz: { age: 133 },
}
```

在实战项目中尽量多用 Record，它会帮助你规避很多错误，在 vue 或者 react 中有很多场景选择 Record 是更优解。

巧用类型约束

在 .tsx 文件里，泛型可能会被当做 jsx 标签

```
const toArray = <T>(element: T) => [element]; // Error in .tsx file.
```

加 extends 可破

```
const toArray = <T extends {}>(element: T) => [element]; // No errors.
```

模块与命名空间

模块系统

TypeScript 与 ECMAScript 2015 一样，任何包含顶级 `import` 或者 `export` 的文件都被当成一个模块。
相反地，如果一个文件不带有顶级的`import`或者`export`声明，那么它的内容被视为全局可见的。

模块语法

我们可以用`export`关键字导出变量或者类型，比如:

```
// export.ts
export const a = 1
export type Person = {
    name: String
```

```
}
```

如果你想一次性导出，那么你可以:

```
const a = 1
type Person = {
    name: String
}
export { a, Person }
import { a, Person } from './export';
```

同样的我们也可以重命名导入的模块:

```
import { Person as P } from './export';
```

如果我们不想一个个导入,想把模块整体导入,可以这样:

```
import * as P from './export';
```

我们甚至可以导入后导出模块:

```
export { Person as P } from './export';
```

当然,除了上面的方法之外我们还有默认的导入导出:

```
export default (a = 1)export default () => 'function'
```

命名空间

命名空间一个最明确的目的就是解决重名问题。
TypeScript 中命名空间使用 namespace 来定义，语法格式如下：

```
namespace SomeNameSpaceName {
   export interface ISomeInterfaceName {      }
   export class SomeClassName {      }
}
```

以上定义了一个命名空间 SomeNameSpaceName，如果我们需要在外部可以调用 SomeNameSpaceName 中的类和接口，则需要在类和接口添加 export 关键字.
其实一个`命名空间`本质上一个`对象`，它的作用是将一系列相关的全局变量组织到一个对象的属性
你在手动构建一个命名空间，但是在`ts`中，`namespace`提供了一颗语法糖。上述可用语法糖改写成：

```
namespace Letter {
  export let a = 1;
  export let b = 2;
```

```
    export let c = 3;
    // ...
    export let z = 26;
}
```

编辑成`js`：

```
var Letter;
(function (Letter) {
    Letter.a = 1;
    Letter.b = 2;
    Letter.c = 3;
    // ...
    Letter.z = 26;
})(Letter || (Letter = {}));
```

命名空间的用处

命名空间在现代TS开发中的重要性并不高,主要原因是ES6引入了模块系统,文件即模块的方式使得开发者能更好的得组织代码,但是命名空间并非一无是处,通常在一些非 TypeScript 原生代码的 `.d.ts` 文件中使用,主要是由于 ES Module 过于静态，对 JavaScript 代码结构的表达能力有限。
因此在正常的TS项目开发过程中并不建议用命名空间。

使用第三方 d.ts

Github 上有一个库 DefinitelyTyped 它定义了市面上主流的JavaScript 库的 d.ts ,而且我们可以很方便地用 npm 引入这些 d.ts。

编写 d.ts 文件

关键字 `declare` 表示声明的意思,我们可以用它来做出各种声明:

- `declare var` 声明全局变量
- `declare function` 声明全局方法
- `declare class` 声明全局类
- `declare enum` 声明全局枚举类型
- `declare namespace` 声明（含有子属性的）全局对象
- `interface` 和 `type` 声明全局类型

TypeScript 的编译原理

编译器的组成

TypeScript有自己的编译器,这个编译器主要有以下部分组成:

- Scanner 扫描器
- Parser 解析器
- Binder 绑定器
- Emitter 发射器
- Checker 检查器

编译器的处理

扫描器通过扫描源代码生成token流:

```
1 SourceCode（源码）+ 扫描器 --> Token 流
```

解析器将token流解析为抽象语法树(AST):

```
1 Token 流 + 解析器 --> AST（抽象语法树）
```

绑定器将AST中的声明节点与相同实体的其他声明相连形成符号(Symbols),符号是语义系统的主要构造块:

```
1 AST + 绑定器 --> Symbols（符号）
```

检查器通过符号和AST来验证源代码语义:

```
1 AST + 符号 + 检查器 --> 类型验证
```

最后我们通过发射器生成JavaScript代码:

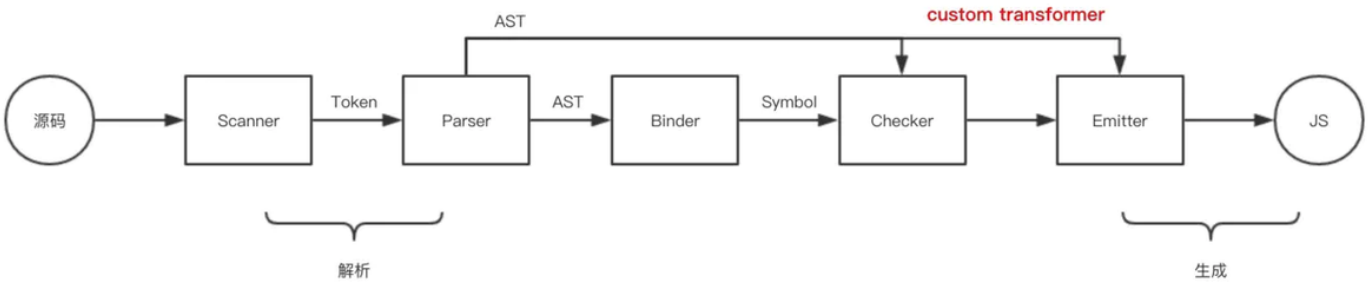
```
1 AST + 检查器 + 发射器 --> JavaScript 代码
```

编译器处理流程

TypeScript 的编译流程也可以粗略得分为三步:

- 解析
- 转换
- 生成

结合上部分的编译器各个组成部分,流程如下图:


AST
custom transformer
源码
Scanner
Token
Parser
AST
Binder
Symbol
Checker
Emitter
JS
解析
生成

一道题

题目地址