**协变和逆变**

用合适的术语来描述这个奇怪的表现，可以说我们允许一个函数类型中，返回值类型是协变的，而参数类型是逆变的。返回值类型是协变的，意思是 A ≼ B 就意味着 (T → A) ≼ (T → B) 。参数类型是逆变的，意思是 A ≼ B 就意味着 (B → T) ≼ (A → T) （ A 和 B 的位置颠倒过来了）。

类型兼容性，看的不是多懂

交叉类型（Intersection Types）： < T & U>

联合类型: 如果一个值是联合类型，我们只能访问此联合类型的所有类型里共有的成员。

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。

as foo 与 <foo> 两种类型断言的语法

这里可以注意到我们不得不多次使用类型断言。 假若我们一旦检查过类型，就能在之后的每个分支里清楚地知道 pet的类型的话就好了。

TypeScript里的 类型保护机制让它成为了现实。 类型保护就是一些表达式，它们会在运行时检查以确保在某个作用域里的类型。 要定义一个类型保护，我们只要简单地定义一个函数，它的返回值是一个 类型谓词：

function isFish(pet: Fish | Bird): pet is Fish {

return (<Fish>pet).swim !== undefined;

}

如果编译器不能够去除 null或 undefined，你可以使用类型断言手动去除。 语法是添加 !后缀： identifier!从 identifier的类型里去除了 null和 undefined：

# 可辨识联合（Discriminated Unions）

# 多态的 this类型

多态的 this类型表示的是某个包含类或接口的 子类型。 这被称做 F-bounded多态性。

# 索引类型（Index types）

 首先是 keyof T， **索引类型查询操作符**。 对于任何类型 T， keyof T的结果为 T上已知的公共属性名的联合。

# 映射类型

type Readonly<T> = { readonly [P in keyof T]: T[P]; }

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

Type和interface的区别？type是类型别名

其一，接口创建了一个新的名字，可以在其它任何地方使用。

另一个重要区别是类型别名不能被 extends和 implements（自己也不能 extends和 implements其它类型）。 因为 软件中的对象应该对于扩展是开放的，但是对于修改是封闭的，你应该尽量去使用接口代替类型别名。

另一方面，如果你无法通过接口来描述一个类型并且需要使用联合类型或元组类型，这时通常会使用类型别名。