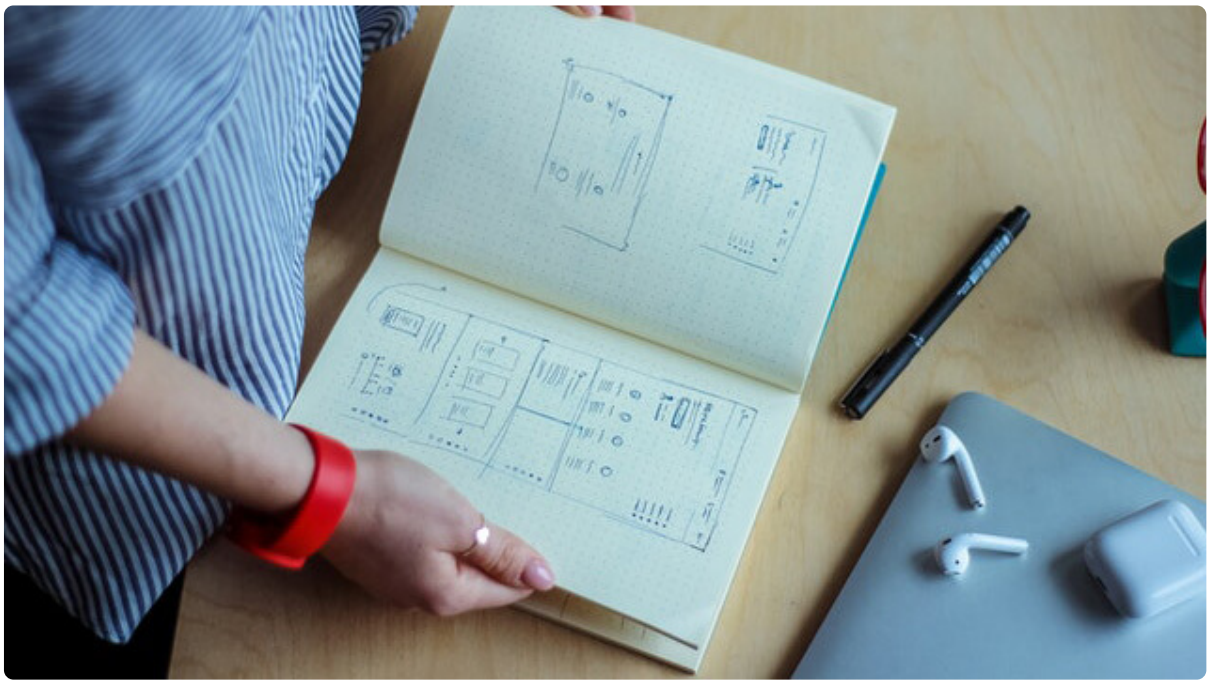
19 使用可辨识联合并保证每个case都被处理

更新时间：2019-06-26 10:13:26



> 我好像是一只牛，吃的是草，挤出的是牛奶。
>
> ——鲁迅

我们可以把单例类型、联合类型、类型保护和类型别名这几种类型进行合并，来创建一个叫做**可辨识联合**的高级类型，它也可称作**标签联合**或**代数数据类型**。

> 所谓单例类型，你可以理解为符合单例模式的数据类型，比如枚举成员类型，字面量类型。

可辨识联合要求具有两个要素：

- 具有普通的单例类型属性（这个要作为辨识的特征，也是重要因素）。
- 一个类型别名，包含了那些类型的联合（即把几个类型封装为联合类型，并起一个别名）。

来看例子：

```
interface Square {
  kind: "square"; // 这个就是具有辨识性的属性
  size: number;
}
interface Rectangle {
  kind: "rectangle"; // 这个就是具有辨识性的属性
  height: number;
  width: number;
}
interface Circle {
  kind: "circle"; // 这个就是具有辨识性的属性
  radius: number;
}
type Shape = Square | Rectangle | Circle; // 这里使用三个接口组成一个联合类型，并赋给一个别名Shape，组成了一个可辨识联合。
function getArea(s: Shape) {
  switch (s.kind) {
    case "square":
      return s.size * s.size;
    case "rectangle":
      return s.height * s.width;
    case "circle":
      return Math.PI * s.radius ** 2;
  }
}
```

上面这个例子中，我们的 Shape 即可辨识联合，它是三个接口的联合，而这三个接口都有一个 kind 属性，且每个接口的 kind 属性值都不相同，能够起到标识作用。

> 这里有个 ES7 的新特性：** 运算符，两个 * 符号组成的这个运算符就是求幂运算符，2 ** 3 ==> 8

看了上面的例子，你可以看到我们的函数内应该包含联合类型中每一个接口的 case。但是如果遗漏了，我们希望编译器应该给出提示。所以我们来看下两种**完整性检查**的方法：

3.6.1 利用 strictNullChecks

我们给上面的例子加一种接口：

```
interface Square {
  kind: "square";
  size: number;
}
interface Rectangle {
  kind: "rectangle";
  height: number;
  width: number;
}
interface Circle {
  kind: "circle";
  radius: number;
}
interface Triangle {
  kind: "triangle";
  bottom: number;
  height: number;
}
type Shape = Square | Rectangle | Circle | Triangle; // 这里我们在联合类型中新增了一个接口，但是下面的case却没有处理Triangle的情况
function getArea(s: Shape) {
  switch (s.kind) {
    case "square":
      return s.size * s.size;
    case "rectangle":
      return s.height * s.width;
    case "circle":
      return Math.PI * s.radius ** 2;
  }
}
```

上面例子中，我们的 Shape 联合有四种接口，但函数的 switch 里只包含三个 case，这个时候编译器并没有提示任何错误，因为当传入函数的是类型是 Triangle 时，没有任何一个 case 符合，则不会有 return 语句执行，那么函数是默认返回 undefined。所以我们可以利用这个特点，结合 strictNullChecks(详见3.4小节) 编译选项，我们可以开启 strictNullChecks，然后让函数的返回值类型为 number，那么当返回 undefined 的时候，就会报错：

```
function getArea(s: Shape): number {
  // error Function lacks ending return statement and return type does not include 'undefined'
  switch (s.kind) {
    case "square":
      return s.size * s.size;
    case "rectangle":
      return s.height * s.width;
    case "circle":
      return Math.PI * s.radius ** 2;
  }
}
```

这种方法简单，但是对旧代码支持不好，因为strictNullChecks这个配置项是2.0版本才加入的，如果你使用的是低于这个版本的，这个方法并不会有效。

3.6.2 使用 never 类型

我们在学习基本类型时学习过，当函数返回一个错误或者不可能有返回值的时候，返回值类型为 never。所以我们可以给 switch 添加一个 default 流程，当前面的 case 都不符合的时候，会执行 default 后的逻辑：

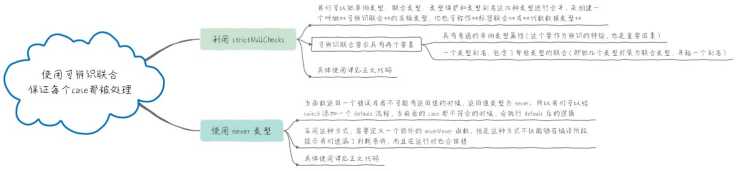
```
function assertNever(value: never): never {
  throw new Error("Unexpected object: " + value);
}
function getArea(s: Shape) {
  switch (s.kind) {
    case "square":
      return s.size * s.size;
    case "rectangle":
      return s.height * s.width;
    case "circle":
      return Math.PI * s.radius ** 2;
    default:
      return assertNever(s); // error 类型“Triangle”的参数不能赋给类型“never”的参数
  }
}
```

采用这种方式，需要定义一个额外的 asserNever 函数，但是这种方式不仅能够在编译阶段提示我们遗漏了判断条件，而且在运行时也会报错。

本节小结

本小节我们学习了可辨识联合类型，定义一个可辨识联合类型有两个要素：具有普通的单例类型属性，和一个类型别名。第一个要素是最重要的一点，因为编译器要根据这个属性来判断当前分支是什么类型，而第二个要素并不影响使用，你完全可以指定上面例子中的s为 Square | Rectangle | Circle 而不使用 Shape 。最后我们讲了两种避免遗忘处理某个case的方法：利用strictNullChecks和使用never类型，都能够帮我们检查遗漏的case，第二种方法的提示更为全面，推荐大家使用。

下个小节我们将学习**this类型**，我们知道this是JavaScript中的关键字，可以用来获取全局对象、类实例对象、构造函数实例等的引用，但是在TypeScript中，它也是一种类型，我们下节课再来细讲。



18 类型别名和字面量类型—单调的类型

20 this，类型？