TypeScript 高级类型

目标

1. 熟悉TypeScript中的特殊高级类型使用

大纲

# 泛型

程序设计中，为了向后扩展，我们有时需要组件不仅能支持当前已有的数据类型，同时也能支持未来某种未知的由用户自行决定的数据类型，这在创建大型应用时提供了非常高的灵活性。

在TypeScript中，组件可以不必限制只支持一种数据类型，还可以使用泛型来创建支持多种类型的数据。 如此以提供给用户可以用自己的数据类型来使用组件的入口。

说明：简单的讲，就是程序由“设计者”来“写死”数据类型的模式，变为“预留接口”来由“使用者”来决定数据类型的模式。

泛型使用“<类型标识符[,类型标识符,……]>”来标识“自定义”的数据类型。

## 泛型函数

泛型函数的作用在于，为函数参数、函数体规定用户“自定义的数据类型”。

// 定义  
**function** *echo*<T>( arg:T ):T  
{  
 **return** arg;  
}  
// 用法  
*echo*<**string**>( "good" );  
*echo*<**number**>( 100 );  
  
// 利用类型推断 <string>  
*echo*( "good" );  
*echo*( 100 );

其中“<T>”中的“T”可以是任何合法的标识符。

在使用“T类型”时，应将其视为“any”对象进行使用，若需要使用具体类型的成员属性或者方法，需要进行“类型断言”将其进行标识为指定类型。

**function** *echo*<T>( arg:T ):T  
{

***console***.log( arg.length ); // 错误：T 类型不存在 length 属性

**return** arg;  
}

“.length”是数组的成员属性之一，若明确告诉编译器参数为数组，则可以明确使用该属性。

**function** *echo*<T>( arg:T[] ):T[]  
{  
 ***console***.log( arg.**length** ); // 错误：T 类型不存在 .length 属性  
 **return** arg;  
}

## 泛型接口

泛型接口的作用在于，为接口内部规定用户“自定义的数据类型”。

// 定义  
**function** *echo*<T>( arg:T ):T  
{  
 **return** arg;  
}  
// 用法  
**let *myecho***:{ <T>( arg:T ):T } = *echo*;  
  
// 使用泛型接口来替代 { <T>( arg:T ):T }  
**interface EchoFunc**{  
 <T>( arg:T ):T;  
}  
  
// 则使用时可以替代为  
**let *myecho***:**EchoFunc** = *echo*;

为了使整个接口都统一“自定义类型”，应该将 <T> 提升到整个接口范围，而不仅仅是某个函数范围，修改如下：

**interface EchoFunc**<T>  
{  
 ( arg:T ):T;  
}

## 泛型类

与泛型接口的定义相类似，只是关键字“interface”变成“class”，并且将<T>自定义数据类型作为类的“类型参数”。

泛型类使用<>括起泛型类型，跟在类名后面。

**class** GenericNumber<T>  
{  
 **zeroValue**:T;  
 **add**:( x:T, y:T ) => T;  
}  
  
// 传入 <number>   
**let *myGenericNumber*** = **new** GenericNumber<**number**>();  
***myGenericNumber***.**zeroValue** = 0;  
***myGenericNumber***.**add** = **function**( x, y ){ **return** x + y; };  
  
// 传入 <string>   
**let *stringNumeric*** = **new** GenericNumber<**string**>();  
***stringNumeric***.**zeroValue** = "";  
***stringNumeric***.**add** = **function**( x, y ){ **return** x + y; };  
  
*alert*( ***stringNumeric***.**add**( ***stringNumeric***.**zeroValue**, "test" ) );

## 用接口来约束泛型

在某些特殊情况下，我们应该尽量缩小“自定义类型”的范围，对其进行限定，规定“参数类型”所必须具备的条件，比如让传入的类型必须包含“.length”属性。

当出现类似需求时，我们定义一个接口来描述约束条件。

创建一个包含 .length属性的接口，使用这个接口和extends关键字实现约束：

**interface Lengthwise**{  
 **length**:**number**;  
}  
  
**function** *echo*<T **extends Lengthwise**>( arg:T ):T  
{  
 ***console***.log( arg.**length** );  
 **return** arg;  
}

通过以上代码，我们则可实现对传入“类型”进行限制，要求其必须是具备“.length”属性的类型，比如“字符串、数组”，或者任意包含“.length”属性的对象。

// 错误 number 类型不具备 .length 属性  
*echo*( 3 );

// 只要包含 .length 属性，则被允许  
*echo*( { **length**: 100, **size**: 100 } );

# 交叉类型

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。

例如：

Person & Serializable & Loggable同时是Person和Serializable和Loggable。

就是说这个类型的对象同时拥有了这三种类型的成员。

以下是一个通过复制来进行成员共享的例子：

**class** Person  
{  
 **constructor**(name:**string** ){ }  
}  
**class** ConsoleLogger  
{  
 log(){}  
}  
**let *jim*** = extend( **new** Person( "Jim" ), **new** ConsoleLogger() );  
**let *n*** = ***jim***.name;  
***jim***.log();

要实现以上代码，需要合理设计extend函数：

**function** *extend*<T, U>( first:T, second:U ):T & U  
{  
 // 定义交叉类型  
 **let** result = <T & U>{};  
   
 // 复制 first 成员  
 **for**( **let** id **in** first )  
 {  
 (<**any**>result)[ id ] = (<**any**>first)[ id ];  
 }  
   
 // 复制 second 成员  
 **for**( **let** id **in** second )  
 {  
 // 只复制 first 不存在的成员，即过滤重复名称的成员  
 **if**( !result.hasOwnProperty( id ) )  
 {  
 (<**any**>result)[ id ] = (<**any**>second)[ id ];  
 }  
 }  
 **return** result;  
}

# 联合类型

## 概述

联合类型与交叉类型类似，但使用上完全不同，类似于位运算中的“位或”。

联合类型表示一个值可以是几种类型之一。 我们用竖线“|”分隔每个类型。

## 联合类型的来由

例如，我们希望一个函数的参数能接受“string”和“number”两种类型：

**function** *setWidth*( value:**any** )  
{  
 **let** element:**HTMLElement**;  
  
 **if**( **typeof** value === "number" )  
 {  
 element.**style**.**width** = value + "px";  
 }  
 **if**( **typeof** value === "string" )  
 {  
 element.**style**.**width** = value;  
 }  
}

显然，用一个any类型来包含“string”和“number”两种类型会出现问题，因为可以传入这两种类型之外的类型，而仍然可以编译通过，但在运行时却会出现问题甚至错误。

## 使用联合类型

所以我们应该使用联合类型，来在编码阶段就限制其类型，以期“享受”TypeScript的类型检测，将问题杜绝在编译阶段，而不是像JavaScript一样等到运行时才能发现问题。

**function** *setWidth*( value:**string** | **number** );

## 成员访问限制

默认情况下，对于联合类型我们只能访问公共成员（各类型都有的共有成员）

**interface Bird**{  
 fly();  
 layEggs();  
}  
  
**interface Fish**{  
 swim();  
 layEggs();  
}  
  
**function** *getSmallPet*():**Fish** | **Bird**{  
 // ...  
}  
  
**let *pet*** = *getSmallPet*();  
***pet***.layEggs(); // 正确：都存在的方法  
***pet***.swim(); // 错误：只有 Fish 才有的方法

## 区分类型

若需要访问某类型独有的成员 ，必须标识为指定类型，让编译器得知要访问的来源：

**let *pet*** = *getSmallPet*();  
  
**if**( (<**Fish**>***pet***).swim )  
{  
 (<**Fish**>***pet***).swim();  
}  
**else**{  
 (<**Bird**>***pet***).fly();  
}

——————下午内容——————

# 类型别名

TypeScript中，允许使用“type”关键字来给类型起个新名字（别名，简名）。

类型别名和接口很像，但可作用于原始值，联合类型，元组以及其它任何你需要手写的类型。

## 使用方法

**type** Name = **string**;  
**type** NameResolver = () => **string**;  
  
**function** *getName*( n: NameResolver ):Name  
{  
 **return** n();  
}

起别名不会新建一个类型，它创建了一个新名字来引用那个类型。

注意：不要给原始类型起别名，这没任何意义，尽管可以做为文档的一种形式使用。

## 在别名中使用泛型

同接口一样，类型别名也可以是泛型

我们可以添加类型参数并且在别名声明的右侧传入：

**type** Container<T> = { **value**:T };

## 字符串别名

在需要限定属性值的情况下，当使用“静态常量”或者“枚举”的方法来规定选项时，往往无法在编译时由编译器感知问题，而需要代码运行时才能发现错误。

而使用字符串别名，能从“类型”的层级上进行限制，而不是“值”限制。

**type** Color = "red" | "green" | "blue";  
  
**function** *setColor*( color:Color ){}  
*setColor*( "red" );// 正确  
*setColor*( "yellow" );// 错误：不是 "red" | "green" | "blue" 之一

## 在别名中使用交叉类型

别名与交叉类型一起使用，我们可以轻松创建出类似于“链表”的类型。

**type** LinkedList<T> = T & { **next**:LinkedList<T> };  
  
**interface Person**{  
 **name**:**string**;  
}  
  
**let *people***:LinkedList<**Person**>;  
**let *s0*** = ***people***.**name**;  
**let *s1*** = ***people***.**next**.**name**;  
**let *s2*** = ***people***.**next**.**next**.**name**;

## 在别名中使用联合类型

**type** NameOrResolver = Name | NameResolver;

# 练习题