typescript2-进阶

- 类型别名
  - 用来给一个类型起一个新的名字，常用于联合类型。用type创建类型别名
  - type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
- 字符串字面量类型
  - 字符串字面量类型是用来约束取值只能取某几个字符串中的一个，也是用type来定义的
  - type EventName = ‘click’ | ‘scroll’ | ‘mousemove’
- 元组Tuple
  - 起源于 F# : 数组合并了相同类型的对象，元组合并了不同类型的对象
  - 当直接对元组类型的变量进行初始化或赋值时，需要提供所有元组类型中指定的项
  - eg: let x:[string,number] = [‘wm’,18]
  - 越界行为
    - 当添加越界元素时，只能添加元组中每个类型的联合类型
    - let x: [string, number];
x = ['Xcat Liu', 25];
x.push('http://xcatliu.com/');
x.push(true);//报错只能是字符串或数字
- 枚举Enum(源于C#)
  - 枚举：用于气质被限定在一定范围内的场景，比如 一周七天，颜色限定为红蓝黄等
    - enum Days {Sun,Mon,Tue,Web,Thu,Fri,Sat}
  - 枚举成员会被复制从0开始递增的数字，也会对枚举值到枚举名进行反向映射
    - Days[0]===Sun;
    - Days[‘Sun’]===0
  - 手动赋值
    - enum Days {Sun = 3, Mon = 1, Tue, Wed, Thu, Fri, Sat}
    - 未手动复制的枚举项会接着上一个枚举项递增，若重复也不会报错
      - Days["Sun"] === 3)// true
Days["Wed"] === 3);//true
    - 手动复制的枚举项可以不是数字(需要用类型断言让tsc无视类型检查)
      - enum Days {Sun = 7, Mon, Tue, Wed, Thu, Fri, Sat = <any>"S"};
  - 常数项和计算所得项
    - enum Color {Red, Green, Blue = "blue".length};
    - 计算所得项后面是未手动复制项，则会无法获得初始值而报错
    - 常数枚举
      - 使用const enum定义的枚举类型，和普通枚举的区别是，他会在编译阶段被删除，并且不能包含计算成员
      - const enum Directions { Up, Down, Left, Right}
let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
      - 编译为：var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
    - 外部枚举
      - 使用 declare enum定义的枚举只会用于编译时的检查，编译结果中会被删除。
      - declare enum Directions { Up, Down, Left, Right}
let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
      - 编译结果:var directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
- 泛型Generics
  - 概念：在定义函数，接口，类的时候，不预先指定具体的类型，而是等到使用的时候再指定类型的一种特性
  - 例子
    - 实现一函数：创建一个指定长度的数组同时填充默认值
    - function createArray(length: number, value: any): Array<any> {
  let result = [];
  for (let i = 0; i < length; i++) {
  result[i] = value;
  }
  return result;
}
    - 缺点：没有准确定义返回值类型，我们预期的是每一项都应该是输入的value类型
    - function createArray<T>(length: number, value: T): Array<T> {
  let result: T[] = [];
  for (let i = 0; i < length; i++) {
  result[i] = value;
  }
  return result;
}

createArray<string>(3, 'x'); // ['x', 'x', 'x']
    - 函数后面<T>, T代表任意输入的类型，在后面的输入value:T,Array<T>中即可使用，在调用的时候可以指定为number，也可以不指定，让类型推导自动算出
  - 多个类型参数
    - function swap<T, U>(tuple: [T, U]): [U, T] {
  return [tuple[1], tuple[0]];
}
swap([7, 'seven']); // ['seven', 7]
  - 泛型约束
    - 我们在函数内使用泛型变量时，由于不清楚属于那种类型，so不能随意操作它的属性和方法，这时我们需要对泛型进行约束
    - interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
} 我们使用extends约束泛型T必须符合接口的样子，也就是必须含有length属性
    - function copyFields<T extends U, U>(target: T, source: U): T {
  for (let id in source) {
  target[id] = (<T>source)[id];
  }
  return target;
}
let x = { a: 1, b: 2, c: 3, d: 4 };
copyFields(x, { b: 10, d: 20 });//T继承U，保证了U上不会出现T中不存在的字段
  - 泛型接口
    - interface CreateArrayFunc<T> {
  (length: number, value: T): Array<T>;
}

let createArray: CreateArrayFunc<any>;
  - 泛型类
    - class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}
  - 泛型参数默认类型
    - TS 2.3后我们可以为泛型中的参数类型指定默认值，当使用是未指定类型，也不乏从实际值参数推测时，就会起作用
    - function createArray<T=string>(length:number,value:T):Array<T>{}
- 类 Class
  - TS除了实现所有ES6类中所有的功能之外，还添加了一些新的用法
  - 一些概念
    - 类(Class):定义了一年实物的抽象特点，包含属性和方法
    - 对象(Object)：类的实例，通过new 生成
    - 面向对象(OOP)：三大特性：封装，继承，多态
    - 存取器(getter,setter):用以改变属性的读取和赋值行为
    - 修饰器(Modifiers):修饰符就是一些关键字，用于限定成员或类型的性质 :public 等
    - 抽象类(Abstract Class):抽象类是供其他类继承的基类，1.不允许被实例化，2.抽象类中的抽象方法必须在子类中被实现
    - 接口（Interfaces）:不同类之间共有的属性和方法，可以抽象成一个接口。接口可以被类实现(implements)。一个雷只能继承自另一个类，但是可以实现多个接口
  - ES6+中的类
    - 属性和方法
      - 用class定义类，用constructor定义构造函数，通过new生成新势力的时候，会自动调用构造函数
    - 继承
      - 使用extends实现继承，子类中使用super来调用父类的构造函数和方法
    - 存取器
    - 静态方法static
      - 不需要实例化直接通过类来调用
    - ES7
      - :static静态属性
      - 实例属性可以直接在类中定义
  - TS中新增的
    - public
      - 修饰的属性和方法是公有的，可以在任何地方被访问到。默认都是public
    - private
      - 私有的：不能再声明它的类的外部使用
    - protected
      - 受保护的：和private类似，区别是他在子类中允许被访问
    - 抽象类(abstract)
      - 抽象类用于定义抽象类和其中的抽象方法
      - 不允许被实例化
      - 抽象类中的抽象方法必须被子类实现
- 类与接口
  - 接口可以对对象的’形状‘进行描述，还可以对类的一部分行为进行抽象
  - 一个类智能集成另一个类，有时候不同类之间有一些共有的特性，这个时候可以吧可行提取成接口，用implements实现，大大提高了面向对象的灵活性
  - 类实现接口
  - 接口实现接口
  - 接口实现类
- 声明合并
  - 如果定义了两个相同名字的函数，接口和类就会合并成一个类型
  - 函数的合并
    - 可以使用重载定义多个函数类型
    - function reverse(x: number): number;
function reverse(x: string): string;
function reverse(x: number | string): number | string {
  if (typeof x === 'number') {
  return Number(x.toString().split('').reverse().join(''));
  } else if (typeof x === 'string') {
  return x.split('').reverse().join('');
  }
}
  - 接口/类的合并
    - 合并的属性可以重复，但是类型必须统一