### 1.ts中的数据类型

ts必须指定数据类型（给人理解将数据类型分成3种） 1.js有的类型 boolean类型、number类型、string类型、array类型、undefined、null 2.ts多出的类型 tuple类型（元组类型）、enum类型（枚举类型）、any类型（任意类型） 3.特别的类型 void类型（没有任何类型）表示定义方法没有返回值 never类型：是其他类型（包括null和undefined）的子类型，代表从不会出现的值 这意味着声明never变量只能被never类型所赋值

// 第一种定义array类型方法  
 var arr1:number[] = [1,2,3]  
 // 第二种定义array类型方法  
 var arr2:Array<number> = [11,22,33]   
  
 // 定义元组类型的方法  
 let arr3:[number,string] = [111,'111']  
  
 // 定义enum枚举类型方法（在程序中用自然语言和计算机状态联系起来，方便理解）  
 enum Flag {success=1,error=2}  
 let s:Flag = Flag.success  
 console.log(s)  
  
 enum Color {red,blue,orange}  
 let num:Color = Color.red  
 console.log(num)  
  
 // 定义any任意类型方法  
 var num1:any = 123  
 num1 = true  
 var obox:any = document.getElementById('box')  
 obox.style.color = 'red'  
  
 // undefined类型  
 var num2:number | undefined  
 console.log(num2)  
  
 // void类型,函数没有返回值  
 function run():void{  
 console.log('run')  
 }  
  
 function run1():number{  
 return 123  
 }  
  
 // never类型定义方法  
 var a:undefined  
 a = undefined  
  
 var b:null  
 b = null  
  
 // var c:never  
 // c = (()=>{  
 // throw new Error('错误')  
 // })()

### 2. Ts函数

1. ts函数定义  
 // es5函数声明  
 function run3(){  
 return 'run'  
 }  
 // es5匿名函数  
 var run4 = function(){  
 return 'run'  
 }  
  
 // ts函数声明  
 function run5():string{  
 return 'run'  
 }  
 // ts匿名函数  
 var run6 = function():number{  
 return 123  
 }  
  
 // ts中定义方法传参  
 function getInfo(name:string,age:number):string{  
 return 'info'+`$(name)---$(age)`  
 }  
 var getInfo1 = function(name:string,age:number):string{  
 return 'info'+`$(name)---$(age)`  
 }  
 // 没有返回值的方法  
 function getInfo2():void{  
 console.log(123)  
 }  
  
  
2.方法可选参数，在参数后面加？变为可选参数，可选参数必须配置到参数的最后面  
  
 function getInfo3(name:string,age?:number):string{  
 if(age){  
 return 'info'+`$(name)---$(age)`  
 }else{  
 return 'info'+`$(name)---年龄保密`  
 }  
 }  
  
3.默认参数 es6和ts都可以设置默认参数  
  
function getInfo4(name:string,age:number=20):string{  
 if(age){  
 return 'info'+`$(name)---$(age)`  
 }else{  
 return 'info'+`$(name)---年龄保密`  
 }  
 }  
 getInfo4('张三',30)  
  
  
4.剩余参数  
  
function sum(a:number,b:number,c:number):number{  
 return a+b+c  
 }  
 sum(1,2,3)  
 // 三点运算符接收传过来的值  
 function sum1(...result:number[]):number{  
 var sum = 0  
 for(var i=0;i<result.length;i++){  
 sum += result[i]  
 }  
 return sum  
 }  
 sum1(1,2,3)  
  
 function sum2(a:number,...result:number[]):number{  
 var sum = 0  
 for(var i=0;i<result.length;i++){  
 sum += result[i]  
 }  
 return a + sum  
 }  
 sum1(1,2,3)  
  
  
5.函数重载  
  
// java重载是指两个或两个以上同名函数，但是函数参数不同，这时候会出现函数重载的情况  
 // ts重载是指通过一个函数提供多个函数定义来试下多种功能的目的  
 function getInfo5(name:string):string  
 function getInfo5(age:number):number  
 function getInfo5(str:any):any{  
 if(typeof str == 'string'){  
 return str  
 }else{  
 return str  
 }  
 }  
 alert(getInfo5(123))  
 // 方法重载可以和函数选择传参一起用  
  
6.箭头函数  
  
setTimeout(function(){  
 alert('run')  
 },1000)  
 setTimeout(()=>{  
 alert('run')  
 },1000)

### 3. Ts中的类

//1.  
 function Person1(name,age){  
 this.name='zhangsan'  
 this.age=20  
 this.run = function(){  
 alert('yundong')  
 }  
 }  
 Person.prototype.sex = '男'  
 Person.prototype.work = function(){  
 alert('work')  
 }  
 var p = new Person1('zhangsan')  
 //2.  
 class Person3{  
 name:string //属性，前面省略了public关键词  
 constructor(name:string){ //构造函数 实例化类的时候触发的函数  
 this.name = name  
 }  
 getName():string{  
 return this.name  
 }  
 setName(name:string):void{  
 this.name=name  
 }  
 }  
 var p1 = new Person3('zhangsan')  
   
 // ts实现继承  
 class Person4{  
 name:string  
 constructor(name:string){  
 this.name=name  
 }  
 run():string{  
 return `$(this.name)在运动`  
 }   
 }  
 // var p2=new Person4('wangwu')  
 // alert(p2.run())  
 class Web4 extends Person4{  
 constructor(name:string){  
 super(name)  
 }  
 }  
 var w = new Web4()  
 alert(w.run())  
  
  
 // 3.类里面的修饰符，ts三种：public（公类、子类、类外面） protected（类外面不能访问） private（子类、类外面不能访问）  
  
 // 4.静态属性 静态方法  
 class Person{  
 name:string  
 constructor(name:string){  
 this.name = name  
 }  
 run(){  
 alert('这是实例方法')  
 }  
 static print(){  
 alert('这是静态方法')  
 // 静态方法没办法直接调用类里面的属性  
 }  
 }  
 var p = new Person('zhangsan')  
 p.run()  
 Person.print()

### 4. Ts中的多态

// 父类定义一个方法不去实现，让继承它的子类去实现，每一个子类有不同的表现多态属于继承  
class Animal{  
 name:string  
 constructor(name:string){  
 this.name = name  
 }  
 eat(){  
 console.log('吃的方法')  
 }  
}  
class Dog extends Animal{  
 constructor(name:string){  
 super(name)  
 }  
 eat(){  
 return this.name+'吃肉'  
 }  
}  
class Cat extends Animal{  
 constructor(name:string){  
 super(name)  
 }  
 eat(){  
 return this.name+'吃粮食'  
 }  
}

### 6. Ts中的抽象类

// 01.抽象类是提供其他类继承的基类，不能直接被实例化  
// 02.用abstract关键字定义抽象类和抽象方法，抽象类中的抽象方法不包含具体实现并且必须在派生类中实现  
// 03.abstract抽象方法只能在抽象类中  
// 04.抽象类和抽象方法用来定义标准：例如，要求Animal类的子类必须有eat方法  
  
abstract class Animal{  
 name:string  
 constructor(name:string){  
 this.name = name  
 }  
 abstract eat():any  
}  
class Dog extends Animal{  
 constructor(name:any){  
 super(name)  
 }  
 // 抽象类的子类必须实现抽象类里面的方法  
 eat(){  
 console.log(this.name+'吃肉')  
 }  
}  
var d = new Dog('xiaogou')  
d.eat()

### 7. Ts中的接口

接口的作用：在面向对象编程中，接口是一种规范的定义，它定义行为和动作的规范。 在程序设计里面，接口起到一定的限制和规范作用。接口定义某一些类所遵守的规范，接口不关心这些类的内部状态数据，也不关心类里面方法的实现细节 它只规定这批类中必须提供某些方法，提供的这些方法就可以满足某些需求。 ts的接口同时增加更灵活的接口类型，包括属性，函数，可索引和类等。

1.属性类接口  
  
 // 对json的约束  
 // function printLabel(label:string):void{  
 // console.log('printLable')  
 // }  
 // printLabel('hahah')  
 // 自定义方法传入参数对json进行约束  
 function printLable(labelInfo:{lable:string}):void{  
 console.log('printLabel')  
 }  
 printLable({lable:'string'})  
  
2.定义接口对参数进行约束  
  
interface FullName{  
 firstName:string;  
 secondName:string  
 }  
 function printName(name:FullName){  
 console.log(name.firstName+'---'+name.secondName)  
 }  
 var obj = {  
 firstName:'zhang',  
 secondName:'san'  
 }  
 printName(obj)  
  
3.接口：可选属性  
  
interface FullName{  
 firstName:string;  
 secondName?:string  
 // 添加？号之后 secondName可传可不传  
 }  
 function getName(name:FullName){  
 console.log(name.firstName+'---'+name.secondName)  
 }  
 var obj = {  
 firstName:'zhang',  
 secondName:'san'  
 }  
 getName(obj)  
  
ajax接口实践  
interface Config{  
 type:string  
 url:string  
 data?:string  
 dataType:string  
 }  
 function ajax(config:Config){  
 var xhr = new XMLHttpRequest()  
 xhr.open(config.type,config.url,true)  
 xhr.send(config.data)  
 xhr.onreadystatechange=function(){  
 if(xhr.readyState==4&&xhr.status==200){  
 console.log('success')  
 }  
 }  
 }  
 ajax({  
 type:'get'  
 url:'www://http:baidu.com'  
 dataType:'json'  
 })

### 8. TS中的泛型

// 1泛型的定义  
// 泛型：在软件工程中，我们不仅要创建一致的定义良好的api，同时也要考虑可重用性。组件不仅能够支持当前的数据类型，还能支持未来的数据类型  
// 在C#和Java这种语言中，可使用泛型来创建可重用的组件，一个组件支持多种类型的数据  
// 2泛型函数  
// T表示泛型，具体什么类型调用这个方法的时候决定的  
function getData<T>(value:T):T{  
 return value  
}  
getData<number>(123)  
// 3泛型类  
// 比如有个最小堆算法，需要同时支持返回数字和字符串两种类型  
// class Minclass{  
// public list:number[]=[]  
// add(num){  
// this.list.push(num)  
// }  
// min():number{  
// var minNum=this.list[0]  
// for(var i=0;i<this.list.length;i++){  
// if(minNum>this.list[i]){  
// minNum = this.list[i]  
// }  
// }  
// return minNum  
// }  
// }  
// var m = new Minclass()  
// m.add(2)  
  
class Minclass<T>{  
 public list:T[]=[]  
 add(value:T):void{  
 this.list.push(value)  
 }  
 min():T{  
 var minNum=this.list[0]  
 for(var i=0;i<this.list.length;i++){  
 if(minNum>this.list[i]){  
 minNum = this.list[i]  
 }  
 }  
 return minNum  
 }  
}  
var m = new Minclass()  
m.add(2)  
// 4泛型接口  
// 函数类型接口  
// interface Configfn{  
// (value1:string,value2:string):string;  
// }  
// var setData:Configfn=function(value1:string,value2:string):string{  
// return value1+value2  
// }  
// 泛型接口  
interface Configfn{  
 <T>(value:T):T;  
}  
var setData:Configfn=function<T>(value:T):T{  
 return value  
}