# 1.TypeScript特点:

面向对象 枚举 接口 访问修饰符，可以使用泛型和类型检查机制

# 2.使用ts需要安装哪些包

cnpm install **typescrip**t **ts-loader** -D

cnpm install **@types/react @types/react-dom @types/source-map** -D

执行tsc --init生成 typescript配置文件**tsconfig.json**

# 3. Typescript语法分类

1.基本类型 number、string、boolean、undefined、symbol、function、array、object、null、void、never、any、tuple元组

2.枚举类型: **使用枚举类型可以为一组值赋予一目了然的名字**

3.高级类型

**tuple元祖型:**特殊要求的数组,指定对应位置元素的类型

**symbol型**: symbol  同一个字符串，返回不同的值

// 解决同一个字符串，想当键名，不重复的问题

**any 任意类型**：可以赋值为任何类型都可以，如果不加类型就是any类型

**void型**：void类型与any类型相反，它表示没有任何类型。 代表一个函数没有返回值

**never：**永远没有返回值的类型，用于函数抛出错误和死循环中使用

**undefined 和 null**可以赋值给被人，别人不可以赋值给他，

想赋值给别人需要修改tsconfig，strictNullChecks为false

**数组创建的三种方法：**

let arr1: number[] = [1, 2, 3, 4]

let arr2: Array<number> = [1, 2, 3, 4]

let arr3: Array<number | string | boolean> = [1, "k", true]

**object型:** 无法修改成员，因为没有指定成员的类型，需要给内部属性定义类型

function类型：可以设置函数返回值类型

let fn1 = (): void => {

    console.log("无参数，无返回值函数")

}

# 4. 类中访问修饰符public，protected，private

（1）**public（公共的）**  声明的属性和方法在类的内部和外部均能访问到，也可以被继承。

（2）**protected（受保护的）** 声明的方法和属性只能在类的内部和其子类能访问

（3）**private（私有的）** 声明的方法和属性只能在其类的内部访问，不能被继承。

# 5.接口—interface—implements实现接口

1.对对象的属性进行规范

2.对对象类（class）进行抽象

3.对函数进行规范

**接口定义了某一批类所需要遵循的规范**，接口不关心具体实现，它只规定这批类必须提供的某些方法，同时还增加了更灵活的接口类型，包括**属性接口、函数类型接口、可索引接口、类类型接口**

**通过implements实现接口**。一个类一般只能继承自另一个类，有时候不同类之间可以有一些共有的特性，这时候就可以把**特性提取成接口**（interfaces），用 implements 关键字来实现。

## (1).属性接口

       <1>.属性接口—对批量方法的参数进行约束

               说明:所有参数必须给出，类型也得匹配

         <2>.可选属性接口—对批量方法的参数进行约束

               说明:可选参数不必给出，类型得匹配

**readonly**  sno: string; //!只读属性,不允许修改值

## (2).函数接口:

对方法的参数及返回值类型进行规范

interface jiami{

*// 函数的参数规范成string,返回值规范成string*

     ( *value* :string):string;

  }

## (3).可索引接口:

对数组和对象进行规范

## (4).类类型接口

**可以规范类中必须出现的属性和方法**

**实现方法：**新定义一个类 通过 **implements**(实现)  这个 类

类类型接口中子类继承了父类的属性和方法，接口中的方法必须实现

**一个类可以通过implement实现多个接口**

class Car2 **implements** **Alarm2, Light2**

# 6.静态属性和方法—static

静态属性只能通过类名调用,不能通过实例对象调用

静态方法中只能使用，静态属性，不能使用非静态属性—因为静态方法中的this就是类不是实例对象

 非静态方法中，也可以使用静态属性，但是不能用this，得用类名引用

★静态方法 定义的时候，方法名字前面加static 。使用的时候，直接通过类调用。静态方法可以被继承。  
★直接通过类名定义的属性。属于静态属性，只能通过类调用，可以继承。

# 7.多态

多态:父类定义一个方法不去实现，让继承它的子类去实现，每个子类有不同实现

多态属于继承的一种表现。

说明:子类不是一定非要实现父类的同名方法

# 8.抽象—abstract --派生类

抽象类是不能被实例化的—可以定义类中接口方法

抽象类的方法必须在派生类中实现

## 抽象类:

抽象类提供继承的基类，抽象类只能被继承，不能直接实例化, 并且必须在派生类中实现

## 派生类 ：

继承抽象类的类就叫派生类，抽象类中的方法，必须再继承中实现，而且不能被实例化

# 9.泛型

是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

可以使用泛型来创建可重用的组件，一个组件可以支持多种类型的数据。 这样 用户就可以以自己的数据类型来使用组件。

# 10. 类型检查机制--类型推断、类型兼容性、类型保护

## 类型推断

类型推断或推论:**不需要指定变量的类型**，**typescript可以根据某些规则自动为其推断出一个类型**

* 基础类型推断:TypeScript里，在有些没有明确指出类型的地方，类型推论会帮助提供类型
* 通用类型推断:当需要从几个表达式中推断类型时候，会使用这些表达式的类型来推断出一个最合适的通用类型
* 上下文类型推断： TypeScript类型推论也可能按照相反的方向进行（从左向右）

## 类型兼容性

当一个源类型，可以赋值给另一个目标类型的时候，就可以说类型A兼容类型B

## 类型保护

类型保护机制：指的是 TypeScript 能够在特定的区块(类型保护区块)中保证变量属于某种特定的类型。可以在此区块中放心地引用此类型的属性，或者调用此类型的方法。

**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。**

### 为什么要使用类型保护?

答:因为 typescript中联合类型只能访问联合类型中共同拥有的成员。

### 什么是联合类型

就是一个属性可以使用两种及以上的类型就叫联合类型

例如：定义一个对象，可以给他指定多种类型

const sayHello = (name: string | undefined) => { /\* ... \*/ };

这里 name 的类型是 string | undefined 意味着可以将 string 或 undefined 的值传递给sayHello 函数。

### instanceof 类型保护

instanceof类型保护是通过构造函数来细化类型，他的右侧是一个构造函数

### in类型保护

* 可以判断某个属性是不是属于某个对象

### typeof类型保护

说明: typeof类型保护用来判断变量是哪种原始类型。

# Typescript高级类型

## 1.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。 （并集）

## 2.联合类型:

声明的类型并不确定，多个类型选一个类型（交集）

## 3.索引类型:

使用索引类型，编译器就能够检查使用了动态属性名的代码。

例如，常见的JavaScript模式是从对象中选取属性的子集。

比如开发中，有时候需要获取对象属性的值，创建一个集合，不存在的属性需要进行约束

## 4. 映射类型:

TypeScript提供了从旧类型中创建新类型的一种方式，

比如，将一个类的属性，全都变为只读

## 5.条件类型:

是一种由条件表达式决定的类型

# 基本类型声明

*// 1.基本类型  number string  boolean  symbol  object  function  array*

*// tuple 元祖 null undefined  void  never any*

*// 2.枚举类型*

*// 3.高级类型*

let x: number = 1000    *//类型断言 创建变量x并确定变量x的数据类型为number类型--静态类型检查*

console.log(x)

let y: string = "abc"

console.log(y)

let z: boolean = true

console.log(z)

*// 4.数组*

let arr1: number[] = [1, 2, 3, 4]

console.log(arr1)

let arr2: Array<number> = [1, 2, 3, 4]

console.log(arr2)

let arr3: Array<number | string | boolean> = [1, "k", true]

console.log(arr3)

*// 5. tuple  元祖型:特殊要求的数组,指定对应位置元素的类型*

let arr4: [number, string] = [2, "abc"]

*// 6. object型:*

let obj1: object = { name: "tom", age: 19, sex: "男" };

*//但是无法修改成员，因为没有指定成员的类型*

*// obj1.name="jarry";//报错*

console.log("obj1", obj1);

let obj2: { name: string, age: number, sex: string } = { name: "tom", age: 19, sex: "男" };

obj2.name = "jarry";

console.log("obj2", obj2);

*// 7.symbol型:*

*// symbol  同一个字符串，返回不同的值*

*// 解决同一个字符串，想当键名，不重复的问题*

let s1: symbol = Symbol("a1")

let s2 = Symbol("a1")               *//使用类型断言，推断是symbol类型,，成为类型推断*

console.log(s1===s2)        *//false*

*// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!如果不加symbol类型断言，s1是等于s2的*

*// 8. any 任意类型：可以赋值为任何类型都可以，如果不加类型就是any类型*

let x1: any = 109;

console.log(x1);

x1 = "abc";

console.log(x1);

*//存储DOM对象，都是可以的*

*// x1=DOM对象;*

x1 = document.getElementById("app");

x1.style.color = "red";

*// 9.  void void型：void类型像是与any类型相反，它表示没有任何类型。 代表一个函数没有返回值。*

*// 与any相反，一般用于函数没有返回值*

*//!返回值没有任何类型*

function warnUser(): void {

    console.log("This is my warning message");

}

warnUser()

*//   声明一个void类型的变量没有什么大用，因为你只能为它赋予undefined、null*

let unusable: void = undefined;

*//!!!!!!!!!!!!!!!!!!! 说明:赋值为null需要修改tscofnig配置文件 "strictNullChecks": false*

*// 10.never  无返回值*

let x8: never;

*// 抛出错误定义成never类型*

*// x8 = (() => { throw new Error("发送错误报告") })()          //!!不注释下面代码无法运行*

*// 或者是死循环时候使用never*

function test(): never {

    while (true);

}

*// 11. undefined undefined类型*

*//let x2:number;*

*//console.log(x2);//报错            //todo  赋值前不允许打印*

*//------------------------------------------*

let x3: undefined;

console.log(x3);*//正确*

*//let x4:undefined=undefined;//undefined类型只能赋值undefined*

*// 12.null*

*// let y: void = null                           //!!   需要修改tsconfig，strictNullChecks为false*

let x5: null = null;

*// x=10;//报错*

*// TypeScript里，undefined和null两者各自有自己的类型分别叫做undefined和null，*

*// 它们可以赋值给任何类型，可以是任何类型子类型，但是需要配置*

*// todo                      undefined 和 null 可以赋值给被人，别人不可以赋值给他，*

*// !!                       想赋值给别人需要修改tsconfig，strictNullChecks为false*

*// 13.function*

let fn1 = (): void => {

    console.log("无参数，无返回值函数")

}

fn1()

let fn2 = (): string => {

    console.log("无参，有返回值")

    return "abc"

}

console.log(fn2())

let fn3 = (*n1*: number, *n2*: number): number => {

    return *n1* + *n2*

}

console.log(fn3(1, 2))

let fn4 = (*name*: string, *age*?: number): string => {

    if (*age*) {

        return `姓名：${*name*},年龄：${*age*}`

    } else {

        return `姓名：${*name*}`

    }

}

console.log(fn4("tom", 18))

console.log(fn4("lucy"))

let fn5 = (*name*: string, *age*: number = 88): string => {

    return `姓名：${*name*},年龄：${*age*}`

}

console.log(fn5("lucy"))

let add1 = (...*arg*: number[]): number => {

    let sum = 0;

    for (let i = 0; i < *arg*.length; i++) {

        sum = sum + *arg*[i];

    }

    return sum;

}

console.log(add1(1, 2, 3));

console.log(add1(1, 2, 3, 4, 5, 6));

*// 7. 函数重载: 两个或两个以上的同名函数，参数类型不同或个数不同实现不同功能*

*// 1.先申明所有方法重载的定义，不包含方法的实现；*

function add9(*a*: number, *b*: number): number;

function add9(*a*: string, *b*: string): string;

*// 2.再声明一个参数为any类型的重载方法；*

function add9(*a*: any, *b*: any): any {

*// 3.实现any类型的方法并通过参数类型（和返回类型）不同来实现重载；*

    let result = null;

    if (typeof *a* == "number" && typeof *b* == "number") {

        result = *a* + *b*;

    }

    if (typeof *a* == "string" && typeof *b* == "string") {

        result = *a* + *b*;

    }

    return result;

}

console.log(add9(1,2))

console.log(add9("1","2"))

*// 13.function*

let fn1 = (): void => {

    console.log("无参数，无返回值函数")

}

fn1()

let fn2 = (): string => {

    console.log("无参，有返回值")

    return "abc"

}

console.log(fn2())

let fn3 = (*n1*: number, *n2*: number): number => {

    return *n1* + *n2*

}

console.log(fn3(1, 2))

let fn4 = (*name*: string, *age*?: number): string => {

    if (*age*) {

        return `姓名：${*name*},年龄：${*age*}`

    } else {

        return `姓名：${*name*}`

    }

}

console.log(fn4("tom", 18))

console.log(fn4("lucy"))

let fn5 = (*name*: string, *age*: number = 88): string => {

    return `姓名：${*name*},年龄：${*age*}`

}

console.log(fn5("lucy"))

let add1 = (...*arg*: number[]): number => {

    let sum = 0;

    for (let i = 0; i < *arg*.length; i++) {

        sum = sum + *arg*[i];

    }

    return sum;

}

console.log(add1(1, 2, 3));

console.log(add1(1, 2, 3, 4, 5, 6));

*// 7. 函数重载: 两个或两个以上的同名函数，参数类型不同或个数不同实现不同功能*

*// 1.先申明所有方法重载的定义，不包含方法的实现；*

function add9(*a*: number, *b*: number): number;

function add9(*a*: string, *b*: string): string;

*// 2.再声明一个参数为any类型的重载方法；*

function add9(*a*: any, *b*: any): any {

*// 3.实现any类型的方法并通过参数类型（和返回类型）不同来实现重载；*

    let result = null;

    if (typeof *a* == "number" && typeof *b* == "number") {

        result = *a* + *b*;

    }

    if (typeof *a* == "string" && typeof *b* == "string") {

        result = *a* + *b*;

    }

    return result;

}

console.log(add9(1,2))

console.log(add9("1","2"))

# Js与Ts的区别

TypeScript 是 JavaScript 的超集，扩展了 JavaScript 的语法，因此现有的 JavaScript 代码可与 TypeScript 一起工作无需任何修改，TypeScript 通过类型注解提供编译时的静态类型检查。

TypeScript 可处理已有的 JavaScript 代码，并只对其中的 TypeScript 代码进行编译。