

TypeScript基础笔记(小满版本)

作者：小余同学

如果你觉得笔记不错的话，可以在GitHub帮作者点个Star吗？这对作者是极大的鼓励和动力，每过一段时间可以上去看看有没有新的优质笔记产出，都是原创且最新的哦，PDF版本还有markdown版本都会进行上传，如果PDF的没有上传就是说明内容太大传输不了，但可以自行将现有的markdown文档进行转换

此笔记会**持续更新**(小满出新则动态更新)，平时会根据水友的意见进行修改其中的细节错误，**关注我在GitHub上面这个项目可以随时获取到最新的笔记**

GitHub地址：[2002小余\(github.com\)](https://github.com/2002xys)

小满的QQ群：**855139333** 小满的微信：**a1195566313**(想加入微信群请加小满，让小满拉你)

欢迎你加入小满的群聊和小满一起探讨技术上的问题，一个人只能闭门造车，一起探讨难点才能走得更远(记得视频多刷刷弹幕)

TypeScript基础笔记(小满版本)

基础类型(TS -- 1)

八种内置类型

注意点

null 和 undefined

number 和 bigint

任意类型(TS -- 2)

类型(任意值) -- any

unknown类型

接口和对象类型(TS -- 3)

interface类型

可选属性 -- ? 操作符

任意属性 -- [propName:string]

只读属性 -- readonly

继承属性 -- extends

Object与object{} -- 加餐环节

前置知识点补充

存储位置不同

传值方式不同

Object类型

object类型

{}字面量类型

数组类型(TS -- 4)

泛型 -- Array <类型>

类数组 -- arguments

接口表示数组

函数扩展(TS -- 5)

对象形式的定义

函数重载

联合类型|类型断言|交叉类型(TS -- 6)

联合类型

函数使用联合类型

交叉类型

类型断言

临时断言

as const

内置对象(TS -- 7)

ECMAScript的内置对象

DOM 和 BOM 的内置对象

Class类(TS -- 8)

public

private

protected

static 静态属性 和 静态方法

interface 定义 类

抽象类(TypeScript8)

元组类型(TS -- 9)

越界的元组

枚举类型(TS -- 10)

enum 关键字定义枚举

数字定义枚举

增长枚举

字符串枚举

异构枚举

接口枚举

const枚举

反向映射

类型推论 | 类型别名(TS -- 11)

类型推论

联合类型

函数式的类型别名

定义类型别名

定义函数别名

定义联合类型别名

定义值的别名

never类型(TS -- 12)

Symbol类型

能够读取到Symbol的两种方式

迭代器 | 生成器(TS -- 13)

迭代器

Symbol列表

生成器 (Builder)

跟for in 的区别

泛型(generic) => (TS -- 14上)

无泛型用法

使用泛型优化

定义泛型接口

对象字面量泛型

泛型约束(函数类)

泛型约束 | 泛型类(TS -- 14下)

使用 keyof 约束对象

泛型类

泛型工具类型(大量补充额外内容)

1.typeof

2.keyof

keyof 的作用

3.in

4.infer

5.extends

- 索引类型
- 映射类型
- Partial
 - 定义
 - 举例说明
- DeepPartial

- Required
 - 定义
- Readonly
 - 定义
 - 举例说明

- Pick
 - 定义
 - 举例说明

- Record
 - 定义
 - 举例说明

- ReturnType
 - 定义
 - 举例说明

- Exclude
 - 定义
 - 举例说明

- Extract
 - 定义
 - 举例说明

- Omit
 - 定义
 - 举例说明

- NonNullable
 - 定义
 - 举例说明

- Parameters
 - 定义
 - 举例说明

tsconfig.json配置文件(TS -- 15)

- 生成 tsconfig.json 文件
- 配置详解
 - 配置分类(compilerOptions 选项)
- 常用的配置
 - 严格模式的限制

namespace命名空间(TS -- 16)

- 嵌套命名空间
- 抽离命名空间
- 简化命名空间
- 命名空间的合并

三斜线指令(TS -- 17)

- 引入声明文件

声明文件d.ts(TS -- 18)

Mixins混入(TS -- 19)

- 对象混入
- 类的混入

装饰器Decorator(TS -- 20)

- 装饰器
- 装饰器工厂

- 装饰器组合
- 方法装饰器
- 属性装饰器
- 参数装饰器

Rollup构建TS项目(TS -- 21)

- Rollup 构建 TS 项目
 - 安装依赖
 - 配置 json 文件
 - 配置 rollup 文件
 - 配置 tsconfig.json
- webpack 构建 TS 项目
 - 配置文件

实战TS编写发布订阅模式(TS -- 22)

- 思维导图

TS 进阶用法 proxy & Reflect(TS -- 23)

- Proxy
- Reflect

TS 进阶用法 Partial & Pick(TS -- 24)

- Partial
- Pick

TS 进阶用法 Record & Readonly(TS -- 25)

- Readonly
- Record

TS 进阶用法 infer(TS -- 26)

- infer

infer 类型提取(TS -- 27)

- 提取头部元素
- 提取尾部元素
- 剔除第一个元素 Shift
- 剔除尾部元素 pop

infer 递归(TS -- 28)

基础类型(TS -- 1)

我认为这个TypeScript跟C语言

是很像的，对语言的定义都有严格的规范。

```
let str:string = "这是字符串类型"
//上方我们将str这个变量定义为了string类型，如果对他输入其他类型的内容就会报错，例如：

let str:string = 666
//这个就会报错了，会提示你不能将类型"number"分配给类型"string"

let muban:string = `web${str}`
//我们也可以使用ES6中的模板字符串

let u:void = undefined
let u:void = null
//空值类型能够有这两种内容。void的内容也是不能去赋值给别人的
//某种程度上来说，void 类型像是与 any 类型相反，它表示没有任何类型。 当一个函数没有返回值时，
你通常会见到其返回值类型是 void

function fnvoid():void{
```

```
    return//函数也可以定义为空值，如果定义了void则不能返回内容
}
```

//undefined跟null类型的也可以交换着用的，具体区别放在了下面

八种内置类型

```
let str: string = "jimmy";
let num: number = 24;
let bool: boolean = false;//这里接收的是布尔值，不是布尔值对象(let b:boolean = new Boolean())
let u: undefined = undefined;
let n: null = null;
let obj: object = {x: 1};
let big: bigint = 100n;
let sym: symbol = Symbol("me");
```

注意点

null 和 undefined

默认情况下 `null` 和 `undefined` 是所有类型的子类型。就是说你可以把 `null` 和 `undefined` 赋值给其他类型。

```
// null和undefined赋值给string
let str:string = "666";
str = null
str= undefined

// null和undefined赋值给number
let num:number = 666;
num = null
num= undefined

// null和undefined赋值给object
let obj:object ={};
obj = null
obj= undefined

// null和undefined赋值给Symbol
let sym: symbol = Symbol("me");
sym = null
sym= undefined

// null和undefined赋值给boolean
let isDone: boolean = false;
isDone = null
isDone= undefined

// null和undefined赋值给bigint
let big: bigint = 100n;
big = null
big= undefined
```

如果你在 `tsconfig.json` 指定了 `"strictNullChecks":true` , `null` 和 `undefined` 只能赋值给 `void` 和它们各自的类型。

number 和 bigint

虽然 `number` 和 `bigint` 都表示数字, 但是这两个类型不兼容。

```
let big: bigint = 100n;
let num: number = 6;
big = num;
num = big;
```

会抛出一个类型不兼容的 `ts (2322)` 错误。

任意类型(TS -- 2)

```
npm install @types/node -D
npm install ts-node -g (装全局的)
```

类型(任意值) -- any

```
let anys: any = "小满穿黑丝"
```

```
anys = []
anys = 18
anys = {}
anys = Symbol('666')
```

//`any`类型就跟原生的是一样的, 能够给任意的类型进行定义, 所以在在 `TypeScript` 中, 任何类型都可以被归为 `any` 类型。这让 `any` 类型成为了类型系统的 顶级类型 (也被称作 全局超级类型)。

作用的地方:

1. 有时候, 我们会想要为那些在编程阶段还不清楚类型的变量指定一个类型。这些值可能来自于动态的内容, 比如来自用户输入或第三方代码库。这种情况下, 我们不希望类型检查器对这些值进行检查而是直接让它们通过编译阶段的检查。那么我们可以使用 `any` 类型来标记这些变量
2. 在对现有代码进行改写的时候, `any` 类型是十分有用的, 它允许你在编译时可选择地包含或移除类型检查。你可能认为 `Object` 有相似的作用, 就像它在其它语言中那样。但是 `Object` 类型的变量只是允许你给它赋任意值 - 但是却不能够在它上面调用任意的方法, 即便它真的有这些方法
3. 当你只知道一部分数据的类型时, `any` 类型也是有用的。比如, 你有一个数组, 它包含了不同的类型的数据

unknown类型

`unknown`类型比`any`类型更安全

就像所有类型都可以被归为 `any`, 所有类型也都可以被归为 `unknown`。这使得 `unknown` 成为 `TypeScript` 类型系统的另一种顶级类型 (另一种的 `any`)

```
let unknow:unknown = {a():number =>123}
unknow.a()//报错
//unkonwn类型是不能够去调用属性跟函数的，它是 any 类型对应的安全类型
```

接口和对象类型(TS -- 3)

在 [typescript](#) 中，我们定义对象的方式要用关键字 **interface**（接口），小满的理解是使用 **interface** 来定义一种约束，让数据的结构满足约束的格式。

我的理解是interface是一个国企部门只招一个人的话，他们会针对走后门的那个人量身定制招聘要求，到面试的时候，这些条件少一个都不行，多了也不行，毕竟已经内定了，再叨、这些条件不仅满足了而且还会更多的技能也没用，别人就是不要你。（留下心酸的眼泪）

interface类型

```
interface A{
    readonly name:string//这个readonly是只读属性，意思就是说只能读取，不能将其他值赋值给他
    age?:number//这个问号就是可选的意思，条件稍微宽松了一些，下面引用这个age的话有没有这个属性都可以，不会报错
}

let obj:A = {
    name = "小满嗷嗷叫"//这里如果不写name就会报错，因为我们在上面定义了A类型集合，并且在这个变量中引入了(里面必须要有name属性且类型为字符串)
    age = 18
}
```

注意：这个规定的属性不能多也不能少，参考我上面的案例

可选属性 -- ? 操作符

```
interface A{
    readonly name:string
    age?:number//这个问号就是可选的意思，条件稍微宽松了一些，下面引用这个age的话有没有这个属性都可以，不会报错
}

let obj:A = {
    name = "小满嗷嗷叫"
    age = 18//age写不写无所谓
}
```

任意属性 -- [propName:string]

需要注意的是，一旦定义了任意属性，那么确定属性和可选属性的类型都必须是它的类型的子集

```
interface Person{
  name:string,
  age?:number,
  [propName:string]:string|number//这个属性一旦定义了，引用这个Person的对象就能够写入任意属性，属性的形式主要看冒号后面你定义了什么类型，比如在这里定义的类型就是string和number类型，不是这两者的类型就会报错，包括在Person里面定义除了string跟number之外其他类型也会报错
  //可以理解为这个 [propName:string]任意属性的优先度相当高
```

注意string与number中间的 ``|`` 符号，小飞棍来咯，这是联合类型，后面笔记会写，这里就当作将string和number类型关系到了一块，有点像逻辑或，满足联合起来的其中一个条件都行，两个也可以

```
}
```

只读属性 -- readonly

只读属性必须在声明时或构造函数里被初始化。

```
interface A{
  readonly name:string//这个readonly是只读属性，意思就是说只能读取，不能将其他值赋值给他
}

let obj:A = {
  name = "小满嗷嗷叫"
}

obj.name = "小满芳龄18"//报错
console.log(obj)//能够读取
let name1 = obj.name
console.log(name1)
```

继承属性 -- extends

儿子在前面，父亲在后面。也就是说顺序是 儿子 继承于 父亲

父亲的部分会继承给儿子，父亲的部分如果没有使用?操作符的话，引用儿子的 对象 是必须将父亲的部分都写下去。一说到这个就想到现在有的地方买房子，出政策能够绑定3代人一起还款，父债子还，跑不掉的，连债务都继承了还不能摆脱，这政策太鸡儿黑心了，绝户计

```
interface A{
  name:string
}

interface B extends A{
  age:number
}

let p:B = {
  name:"有看到小满的裤子吗？"
  age:88//两种类型都要写
}
```


Object与object{} -- 加餐环节

前置知识点补充

原始数据类型（基础数据类型）	中文称呼	引用数据类型	中文称呼
Undefined	未定义	{}	对象
Null	空值	function	函数
Boolean	布尔值	[]	数组
Number	数字		
String	字符串		

存储位置不同

原始数据类型：直接存储在栈（stack）中的简单数据段，占据空间小，大小固定，属于被频繁使用的数据，所以存储在栈中；

引用数据类型：存储在堆（heap）中的对象，占据空间大，大小不固定，如果存储在栈中，将会影响程序运行的性能。引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址，当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后，从堆中获得实体。

传值方式不同

- **基本数据类型：按值传递**

不可变 (immutable) 性质：

基本类型是不可变的 (immutable)，只有对象是可变的 (mutable)。有时我们会尝试“改变”字符串的内容，但在 JS 中，任何看似对 string 值的“修改”操作，实际都是创建新的 string 值。任何方法都无法改变一个基本类型的值(在下面的字面量类型中会再次强调)

- **引用类型：按引用传递**

引用类型的值是可变的

引用类型的值是同时保存在栈内存和堆内存中的对象。javascript 和其他语言不同，其不允许直接访问内存中的位置，也就是说不能直接操作对象的内存空间，那我们操作啥呢？实际上，是操作对象的引用，引用类型的值是按引用访问的。

`object`、`Object` 以及 `{}` 这三个类型(第三个类型为空对象字面量模式)大家可能不太理解

这集加餐环节就是进行补充，一个冷门但是不邪门的知识点

Object类型

```
//这个类型是跟原型链有关的原型链顶层就是 Object，所以值类型和引用类型最终都指向 Object，所以在
TypeScript中Object他包含所有类型。就可以等于任何一个值
//1. 数字类型
let a:Object = 123
//字符串类型
let b:Object = "小满今天没穿裤子"
//数组类型
let c:Object = [1314,520]
//对象类型
let d:Object = {name:"草莓",sex:"女",address:"小满微信一群"}
//any或者function
let e:Object = ()=> "学姐贴贴"
```

`Object` 类型是所有 `Object` 类的实例的类型。由以下两个接口来定义：

- `Object` 接口定义了 `Object.prototype` 原型对象上的属性；
- `ObjectConstructor` 接口定义了 `Object` 类的属性，如上面提到的 `Object.create()`。

object类型

`object` 代表所有非值类型(非原始类型)的类型，例如 数组 对象 函数等，常用于泛型约束

所有原始类型都不支持，所有引用类型都支持

```
//错误 原始类型(字符串)
let f:object = '努力会获得回报的'
//错误 原始类型(数字)
let g:object = 123
//错误 原始类型(布尔值类型)
let h:object = true
//正确 引用类型(数组类型)
let i:object = [123,"学姐学习Vue3",true]
//正确 引用类型(对象类型)
let j:object = {name:"小满",identity:['B站UP主','二次元','京东员工'],'全栈开发工程师'],sex:"女"}
//正确 引用类型(函数类型)
let k:object = ()=>"不要对自己pua，相信自己是最棒的，尊重自己，人生更精彩"
```

{ }字面量类型

看起来很别扭的一个东西 你可以把他理解成 `new Object` 就和我们的第一个 `Object` 基本一样 包含所有类型

```
//与Object类型一样
let l:{ } = 123//等等，就不写了，跟Object一样
//补充--字面量模式
//这个虽然可以赋值任意类型，赋值结束后，是没办法进行一个修改和增加的操作的
```

数组类型(TS -- 4)

普通的声明方式

```
//类型加中括号
let arr:number[] = [123]
//这样会报错定义了数字类型出现字符串是不允许的
let arr:number[] = [1,2,3,'1']
//操作方法添加也是不允许的
let arr:number[] = [1,2,3,]

let arr:number[] = [1,2,3,4]; //数字类型的数组
let arr2:string[] = ["1","2","3","4"]; //字符串类型的数组
let arr3:any[] = [1,"2",true,undefined,[],{}]; //任意类型的数组

let arr4:number[][][] = [[[]],[[]],[[]]]
//这个也能够决定你二维数组还是三维数组想要套几层就写几层
```

泛型 -- Array <类型>

规则 Array <类型>

```
let arr1:Array<number> = [1,2,3,4,5]
let arr2:Array<string> = ["1,2,3,4,5"]
let arr3:Array<boolean> = [true]

//泛型数组套娃写法(还能够决定数组里面数组的类型之类的)
let arr4:Array<Array<number>> = [[123],[456]]
```

类数组 -- arguments

是所有参数的一个集合

```
function Arr(...args:any):void{//...args为ES6的解构方式，任意类型，voidwei不能有返回值
    console.log(arguments)//输出{'0':4,'1':56,'2':789}

    let arr:number[] = arguments//会报错，报缺少类型number[]的以下属性：
    pop,push,concat,join
    let arr:IArguments = arguments//解决方法

    //其中 IArguments 是 TypeScript 中定义好了的类型，它实际上就是：
    interface IArguments {
        [index: number]: any;
        length: number;
        callee: Function;
    }

    Arr(4,56,789)
```

接口表示数组

一般用来描述类数组

```
interface ArrNumber {
    [index: number]: number; //后面的才是定义类型的
    //[index: number]: string; 这个就是定义字符串的了
}
let Arr: ArrNumber = [1, 2, 3, 4, 5];
//let Arr: ArrNumber = ["1, 2, 3, 4, 5"];
//表示：只要索引的类型是数字时，那么值的类型必须是数字。
```

函数扩展(TS -- 5)

函数内参数类型也是可以定义的

```
const fn(name:string,age:number):string{
    return name + age
}
let a = fn('小满',10000)//输入不符合上述参数内定义的类型就会出错
console.log(a)//输出小满10000

-----

const fn(name:string,age:number = 666):string{//如果在下面使用的时候，没有参数传进来就会以你在这里设置的默认参数执行，比如这个666
    return name + age
}
let a = fn('小满')//输入不符合上述参数内定义的类型就会出错
console.log(a)//输出小满666

-----

const fn(name:string,age?:number = 666):string{//也可以使用这个`?`操作符，age传不传就变成可选的了
    return name + age
}
let a = fn('小满穿女仆装')//输入不符合上述参数内定义的类型就会出错
console.log(a)//输出小满穿女仆装
```

对象形式的定义

跟定义对象差不多，但是在针对多个参数的时候会更加的方便，且记得引用的时候要写成({xxxx})形式，不然会报错，输出的是数组形式的

```
interface User{
    name:string;
    age:number
}

const fn(user:User):User{//这里的参数填写方式就变得简单了
    return user
}
let a = fn({
    name:"小满",
    age:18
})//输入不符合上述参数内定义的类型就会出错
console.log(a)//输出{name: '小满', age: 18}
```

函数重载

重载是方法名字相同，而参数不同，返回类型可以相同也可以不同。

如果参数类型不同，则参数类型应设置为 **any**。

参数数量不同你可以将不同的参数设置为可选。

为了让编译器能够选择正确的检查类型，它与 JavaScript 里的处理流程相似。它查找重载列表，尝试使用第一个重载定义。如果匹配的话就使用这个。因此，在定义重载的时候，一定要把最精确的定义放在最前面。

```
function fn(params:number):void//第一套规则
function fn(params:string,params2:number):void//第二套规则
function fn(params:any,params?:any):void{
    console.log(params)
    console.log(params2)
}

let a = fn(1,1)
//输出1跟undefined，因为遵循的是第一套规则
let a = fn("1",1)
//输出"1"跟1，遵循的是第二套规则
```

联合类型 | 类型断言 | 交叉类型(TS -- 6)

联合类型

联合类型能够让我们可选我们自己需要的类型部分，如果需要的类型超过或者达到2个，那就可以使用。

那为什么不使用any呢？那是因为我们需要的并不是所有类型都能通过，我只希望这两个或者3个类型能够通过，如果需要的类型超过或者达到两个都使用any的话，那就和JavaScript原生没有区别了

```
//例如我们的手机号通常是13xxxxxxx 为数字类型 这时候产品说需要支持座机
//所以我们可以使用联合类型支持座机字符串
let myPhone: number | string = '010-820'

//这样写是会报错的应为我们的联合类型只有数字和字符串并没有布尔值
let myPhone: number | string = true//报错
```

函数使用联合类型

这个!!是怎么回事呢？

我们知道一串数字想变成字符串只要加上""就能隐式转换成字符串。

那一个类型只要!就能进行反转，!只有正反，也就是false跟true，这种就有点类似隐式转换了，我们连续转两次就相当于当前形式的布尔值类型了

```
let fn = function(type:number):boolean {
    return !!type//将type强行转化为布尔值类型，如果没用进行转化的话是会报错的
}

-----

let fn = function(type:number|boolean):boolean {
    return !!type//将type强行转化为布尔值类型，如果没用进行转化的话是会报错的
}

let result = fn(1)
console.log(result);//true
```

交叉类型

多种类型的集合，联合对象将具有所有联合类型的所有成员

```
interface Pople{
    name:string
    age:number
}
interface Man{
    sex:number
}

const xiaoman = (man:Pople & Man):void => { //这里通过了&将Pople跟Man交叉在了一起，则
man需要处理Pople也要处理Man。还可以继续跟更多个interface
    console.log(man)
}

xiaoman({
    name:"小满今天坏掉了"
    age:18
    sex:1//如果sex不写是会报错的，会提示你少写了一个sex
})
```

类型断言

语法格式，值 as 类型 或者 <类型>值

需要注意的是，类型断言只能够「欺骗」TypeScript 编译器，无法避免运行时的错误，反而滥用类型断言可能会导致运行时错误

覆盖它的推断，并且能以你任何你想要的方式分析它，这种机制被称为「类型断言」。TypeScript 类型断言用来告诉编译器你比它更了解这个类型，并且它不应该再发出错误

当 **S** 类型是 **T** 类型的子集，或者 **T** 类型是 **S** 类型的子集时，**S** 能被成功断言成 **T**。这是为了在进行类型断言时提供额外的安全性，完全毫无根据的断言是危险的，如果你想这么做，你可以使用 `any`。

2、类型断言的用途

- (1) 将一个联合类型推断为其中一个类型
- (2) 将一个父类断言为更加具体的子类
- (3) 将任何一个类型断言为 `any`

(4) 将 any 断言为一个具体的类型

原型:

```
let fn = function(num:number | string):void{
    console.log(num.length); //这里会报错，因为我们确实没有.length这个内容
}
fn("12345")
```

断言写法

```
let fn = function(num:number | string):void{
    console.log((num as string).length); //用括号括起来，as断言他是string类型
}
fn("12345") //这样会打印出5
fn(12345) //这样会打印出undefined
```

另一个例子

```
interface A{
    run:string
}
interface B{
    build:string
}

let fn(type:A | B) =>{
    console.log(<A>type.run);
}

fn({
    build:"123" //这里是没办法传过去的，断言是不能够滥用的，因为我们确实没有.run这个内容
})
```

临时断言

1.使用any临时断言

```
window.abc = 123
//这样写会报错因为window没有abc这个东西
(window as any).abc = 123
//可以使用any临时断言在 any 类型的变量上，访问任何属性都是允许的。
```

在下面的例子中，将 something 断言为 boolean 虽然可以通过编译，但是并没有什么用 并不会影响结果，因为编译过程中会删除类型断言

```
function toBoolean(something: any): boolean {
    return something as boolean;
}

let bbb = toBoolean(1);
console.log(bbb)
// 返回值为 1
//
```

as const

是对字面值的断言，与 const 直接定义常量是有区别的

如果是普通类型跟直接 const 声明是一样的

```
const names = '小满'
names = 'aa' //无法修改

let names2 = '小满' as const
names2 = 'aa' //无法修改
```

```
// 数组
let a1 = [10, 20] as const;
const a2 = [10, 20];

a1.unshift(30); // 错误，此时已经断言字面量为[10, 20],数据无法做任何修改
a2.unshift(30); // 通过，没有修改指针。之所以没有修改指针是因为const的性质是决定了指针指向的位置是已经固定不会发生改变的了，这个30想要添加进去除非直接修改存储值的地方
```

内置对象(TS -- 7)

ECMAScript的内置对象

JavaScript 中有很多内置对象，它们可以直接在 TypeScript 中当做定义好了的类型。

Boolean、Number、string、RegExp、Date、Error

```
const regexp:RegExp = /\w\d\s///声明正则

const date:Date = new Date()//对象类型
//const date:Date = new Date().getTime() number类型
const error:Error('错误')
```

总结

```
let b: Boolean = new Boolean(1)
console.log(b)
let n: Number = new Number(true)
console.log(n)
let s: String = new String('小满今天穿白丝')
console.log(s)
let d: Date = new Date()
console.log(d)
let r: RegExp = /^1/
console.log(r)
let e: Error = new Error("error!")
console.log(e)
```

DOM 和 BOM 的内置对象

Document、HTMLElement、Event、NodeList 等


```

const list:NodeList = document.querySelectorAll('#list li')
console.log(list)
//NodeList 实例对象是一个类似数组的对象，它的成员是节点对象。Node.childNodes、
document.querySelectorAll () 返回的都是 NodeList 实例对象。 [1] NodeList 对象代表一个
有序的节点列表。

const body:HTMLElement = document.body
console.log(body)

const div:HTMLDivElement = document.querySelector('div')
console.log(div)

document.body.addEventListener('click', (e:MouseEvent)=>{
    console.log(e)
})

//promise
function promise():Promise<number>{//Promise是类型,number是泛型
    return new Promise<number>(resolve,reject)=>{
        resolve(1)//如果不进行断言的话会报错
    }
}

promise().then(res=>{
    console.log(res)//返回1，这里会提示你res应该输入number类型
})

```

```

let body: HTMLElement = document.body;
let allDiv: NodeList = document.querySelectorAll('div');
//读取div 这种需要类型断言 或者加个判断应为读不到返回null
let div:HTMLElement = document.querySelector('div') as HTMLDivElement
document.addEventListener('click', function (e: MouseEvent) {

});
//dom元素的映射表
interface HTMLElementTagNameMap {
    "a": HTMLAnchorElement;
    "abbr": HTMLElement;
    "address": HTMLElement;
    "applet": HTMLAppletElement;
    "area": HTMLAreaElement;
    "article": HTMLElement;
    "aside": HTMLElement;
    "audio": HTMLAudioElement;
    "b": HTMLElement;
    "base": HTMLBaseElement;
    "bdi": HTMLElement;
    "bdo": HTMLElement;
    "blockquote": HTMLQuoteElement;
    "body": HTMLBodyElement;
    "br": HTMLBRElement;
    "button": HTMLButtonElement;

```

```
"canvas": HTMLCanvasElement;
"caption": HTMLTableCaptionElement;
"cite": HTMLCiteElement;
"code": HTMLCodeElement;
"col": HTMLTableColElement;
"colgroup": HTMLTableColElement;
"data": HTMLDataElement;
"datalist": HTMLDataListElement;
"dd": HTMLDataElement;
"del": HTMLModElement;
"details": HTMLDetailsElement;
"dfn": HTMLDataElement;
"dialog": HTMLDialogElement;
"dir": HTMLDirectoryElement;
"div": HTMLDivElement;
"dl": HTMLDLListElement;
"dt": HTMLDataElement;
"em": HTMLDataElement;
"embed": HTMLEmbedElement;
"fieldset": HTMLFieldSetElement;
"figcaption": HTMLCaptionElement;
"figure": HTMLFigureElement;
"font": HTMLFontElement;
"footer": HTMLPageFooterElement;
"form": HTMLFormElement;
"frame": HTMLFrameElement;
"frameset": HTMLFrameSetElement;
"h1": HTMLHeadingElement;
"h2": HTMLHeadingElement;
"h3": HTMLHeadingElement;
"h4": HTMLHeadingElement;
"h5": HTMLHeadingElement;
"h6": HTMLHeadingElement;
"head": HTMLHeadElement;
"header": HTMLPageHeaderElement;
"hgroup": HTMLSectionHeadElement;
"hr": HTMLHRElement;
"html": HTMLHtmlElement;
"i": HTMLDataElement;
"iframe": HTMLIFrameElement;
"img": HTMLImageElement;
: HTMLInputElement;
"ins": HTMLModElement;
"kbd": HTMLDataElement;
"label": HTMLLabelElement;
"legend": HTMLLegendElement;
- li": HTMLLIElement;
"link": HTMLLinkElement;
"main": HTMLMainElement;
"map": HTMLMapElement;
"mark": HTMLMarkElement;
"marquee": HTMLMarqueeElement;
"menu": HTMLMenuElement;
"meta": HTMLMetaElement;
"meter": HTMLMeterElement;

```

```
"nav": HTMLElement;
"noscript": HTMLElement;
"object": HTMLObjectElement;
"ol": HTMLListElement;
"optgroup": HTMLOptGroupElement;
"option": HTMLOptionElement;
"output": HTMLFormElement;
"p": HTMLParagraphElement;
"param": HTMLParamElement;
"picture": HTMLPictureElement;
"pre": HTMLPreElement;
"progress": HTMLProgressElement;
"q": HTMLQuoteElement;
"rp": HTMLElement;
"rt": HTMLElement;
"ruby": HTMLElement;
"s": HTMLElement;
"samp": HTMLElement;
"script": HTMLScriptElement;
"section": HTMLElement;
"select": HTMLSelectElement;
"slot": HTMLSlotElement;
"small": HTMLElement;
"source": HTMLSourceElement;
"span": HTMLSpanElement;
"strong": HTMLElement;
"style": HTMLStyleElement;
"sub": HTMLElement;
"summary": HTMLElement;
"sup": HTMLElement;
"table": HTMLTableElement;

```

Class类(TS -- 8)

ES6 提供了更接近传统语言的写法，引入了 Class（类）这个概念，作为对象的模板。通过 `class` 关键字，可以定义类。基本上，ES6 的 `class` 可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的 `class` 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用 ES6 的“类”改写

JavaScript写法

```
//定义类      JavaScript写法
class Person {
  constructor (name:string,age:number,sub:boolean) {
    this.name = name
    this.age = age
    this.sub = sub
  }
}

new Person("小满",22,false)
```

TypeScript写法

```
//在TypeScript中是需要提前声明类型的
class Person {
  name:string
  age:number
  sub:boolean//没错，没使用标红的是这些
  constructor (name:string,age:number,sub:boolean) {
    this.name = name
    this.age = age
    this.sub = sub//上面定义了变量就需要使用，如果没用使用的话声明的变量就会标红(就算不标红不提示，真运行下去也会报错)，不能就那么放着，要么就用上，要么就给他个默认值0塞着
  }
}

new Person("小满",22,false)
```

public

`public`内部外部都可以访问，如果定义了`public`，像`p`就能够访问`constructor`内部的变量了。当然，默认情况下也是`public`

```
//在TypeScript中是需要提前声明类型的
class Person {
  public name:string
  public age:number
  public sub:boolean//没错，没使用标红的是这些
  constructor (name:string,age:number,sub:boolean) {
    this.name = name
    this.age = age
    this.sub = sub//上面定义了变量就需要使用，如果没用使用的话声明的变量就会标红(就算不标红不提示，真运行下去也会报错)，不能就那么放着，要么就用上，要么就给他个默认值0塞着
  }
}
```

```
let p = new Person("小满",22,false)
p.age p.name p.sub//都可以访问
```

private

private 私有变量只能在内部访问

```
//在TypeScript中是需要提前声明类型的
class Person {
    private name:string
    private age:number
    private sub:boolean//没错，没使用标红的是这些
    constructor (name:string,age:number,sub:boolean) {
        this.name = name
        this.age = age
        this.sub = sub//上面定义了变量就需要使用，如果没用使用的话声明的变量就会标红(就算不标红不提示，真运行下去也会报错)，不能就那么放着，要么就用上，要么就给他个默认值0塞着
    }
}

let p = new Person("小满",22,false)
p.age p.name p.sub//都访问不到了
```

protected

protected内部和子类中访问

private跟protected他们的区别是一个是只能在内部使用，一个是内部与子类访问，例子如下

```
//在TypeScript中是需要提前声明类型的
class Person {
    protected name:string
    private age:number
    public sub:boolean//没错，没使用标红的是这些
    constructor (name:string,age:number,sub:boolean) {
        this.name = name
        this.age = age
        this.sub = sub//上面定义了变量就需要使用，如果没用使用的话声明的变量就会标红(就算不标红不提示，真运行下去也会报错)，不能就那么放着，要么就用上，要么就给他个默认值0塞着
    }
}

class Man extends Person{
    constructor(){
        super("小满",22,false)
        this.name
        this.sub//这两个都可以访问到，this.age访问不到。因为age是private，private只能在内部使用而不能在子类访问，Man是Person的子类
    }
}

let p = new Person("小满",22,false)
p.age p.name p.sub
```

static 静态属性和 静态方法

- **静态属性和非静态属性的区别：**

- 在内存中存放的位置不同：所有 static 修饰的属性和方法都存放在内存的方法区里，而非静态的都存在堆内存中
- 出现的时机不同：静态属性和方法在没创建对象之前就存在，而非静态的需要在创建对象才存在
- 静态属性是整个类都公用的
- 生命周期不一样，静态在类消失后被销毁，非静态在对象销毁后销毁
- 用法：静态的可以直接通过类名访问，非静态只能通过对象进行访问

- 使用 static 注意事项

- **带静态修饰符的方法只能访问静态属性**
- **非静态方法既能访问静态属性也能访问非静态属性**
- **非静态方法不能定义静态变量**
- **静态方法不能使用 this 关键字**
- **静态方法不能调用非静态方法，反之可以**

- 父子类中静态和非静态的关系

- 对于非静态属性，子类可以继承父类非静态属性，但是当父子类出现相同的非静态属性时，不会发生子类的重写并覆盖父类的非静态属性，而是隐藏父类的非静态属性
- 对于非静态方法，子类可以继承并重写父类的非静态方法
- 对于静态属性，子类可以继承父类的静态属性，但是如何和非静态属性一样时，会被隐藏
- 对于静态方法，子类可以继承父类的静态方法，但是不能重写静态方法，同名时会隐藏父类的

注：静态属性、静态方法、非静态属性都可以被继承和隐藏，但是不可以被重写，非静态方法可以被重写和继承

- **静态代码块的作用：**

一般情况下，有些代码需要在项目启动的时候就执行，这时候就需要静态代码块，比如一个项目启动需要加载配置文件，或初始化内容等。

- **静态代码块不能出现在任何方法体内**

对于普通方法：普通方法是需要加载类 new 出一个实例化对象，通过运行这个对象才能运行代码块，而静态方法随着类加载就运行了。

对于静态方法：在类加载时静态方法也加载了，但是必须需要类名或者对象名才可以访问，相比于静态代码块，静态方法是被动运行，而静态代码块是主动运行

- **静态代码块不能访问普通变量**

普通变量只能通过对象调用的，所以普通变量不能放在静态代码块中。

普通代码块和构造代码块

- **静态代码块和构造代码块在声明上少一个 static 关键字**

- **执行时机：**

构造代码块在创建对象时被调用，每次创建对象都会调用一次，且优先于构造函数执行。

注：不是优先于构造函数执行，而是依托于构造函数，如果不创建对象就不会执行构造代码块

- **普通代码块和构造代码块的区别在于，构造代码块是在类中定于的，而普通代码块是在方法体中定义的，执行顺序和书写顺序一致。**

执行顺序

静态代码块 > 构造代码块 > 构造函数 > 普通代码块

```

class Person {
    protected name:string
    private age:number
    public sub:boolean//没错，没使用标红的是这些
    static aaa:string = '123456'//静态属性

    constructor (name:string,age:number,sub:boolean) {
        this.name = name
        this.age = age
        this.sub = sub//上面定义了变量就需要使用，如果没用使用的话声明的变量就会标红(就算不
        标红不提示，真运行下去也会报错)，不能就那么放着，要么就用上，要么就给他个默认值0塞着
        this.run()//会报错，调用不了。互斥的，不能够通过静态函数去访问内部的变量，或者是在内
        部的变量去调用外部的静态函数
        Person.run()//只能这样去调用
    }

    static run (){
        this.dev()//静态函数之间可以互相调用
        this.aaa//用this的话只能访问上面static类型的，其他的不管是public还是private或者是
        protected都是不能够访问的(会报不存在属性的错误)    因为这里的this指的是当前这个类，而构造函数
        里面的this指的是新的实例对象
        return '789'
    }

    static dev(){
        this.aaa//静态函数之间可以互相调用
        return 'dev'
    }
}

console.log(Person.run())//返回789

Person.aaa//能够直接访问，不需要再new一下
console.log(Person.aaa)
let p = new Person("小满",22,false)

```

interface 定义 类

ts interface 定义类 使用关键字 implements 后面跟 interface 的名字多个用逗号隔开 继承还是用 extends

通过接口去约束类

```

interface Person{
    run(type:boolean):boolean
}

class Man implements Person{//会提示我们Man中缺少属性run，但类型Person中需要该属性
}

```

```

//通过接口去约束类
interface Person{

```

```

    run(type:boolean):boolean
}

interface H{
    set():void
}

class Man implements Person,H{//会报错，提示我们缺少set属性
    run(type:boolean):boolean{
        return type
    }
}

```

```

interface Person{
    run(type:boolean):boolean
}

interface H{
    set():void
}

class A{//也可以使用继承去使用
    params:string
    constructor(params){
        this.params = params
    }
}

class Man extends A implements Person,H{
    run(type:boolean):boolean{
        return type
    }
    set(){
        //啥也没有，这就是用接口去描述类
    }
}

```

抽象类(TypeScript8)

用关键词 `abstract` 修饰的类称为 `abstract 类` (**抽象类**)

应用场景如果你写的类实例化之后毫无用处此时我可以把他定义为抽象类

或者你也可以把他作为一个基类 -> 通过继承一个**派生类**去实现基类的一些方法

对于 `abstract` 方法只允许声明，不允许实现（因为没有方法体）（毕竟叫抽象，当然不能实实在在的让你实现），并且不允许使用 `final` 和 `abstract` 同时修饰一个方法或者类，也不允许使用 `static` 修饰 `abstract` 方法。也就是说，`abstract` 方法只能是实例方法，不能是类方法。

```

abstract class A{
    name:string
    construct(name:string){//construct: 构造器
        this.name = name
    }
}

```



```

    //abstract getName(){//方法getName不能具有实现，因为它标记为抽象。定义抽象类的函数
    //    return 213
    //}
    setName(name:string){
        this.name = name
    }
    abstract getName():string//抽象类

}

class B extends A{//派生类。定义了抽象类必须在派生类里实现
    //B类是继承A类的，此时A类就是一个抽象类
    constructor(){
        super('小满')
    }
    getName():string{
        return this.name
    }
}

//此时A类是无法被创建实例的(new A)，也就是无法创建抽象类的实例
//B类是可以创建实例的(new B)

let b = new B
b.setName("小满2")//通过抽象类的设置，成功修改掉子类的内容
//    setName(name:string){
//        this.name = name
//    }
console.log(b.getName())

```

元组类型(TS -- 9)

数组合并了相同类型的对象，而元组（Tuple）合并了不同类型的对象。

```

let arr:[string,number] = ['小满',22]//这样的方式就叫做元组，定义了每个位置需要满足的不同
类型
arr[0].length//有
arr[1].length//无，因为上面的定义类型会自动帮我们推断是否有该方法
//Number 类型是没有 length 属性的

```

越界的元组

当添加的元组越界的时候，越界的类型会被限制为元组类型中每个类型的联合类型

```

let arr:[string,number] = ['小满',22]//这样的方式就叫做元组，定义了每个位置需要满足的不同
类型
arr.push(true)//会报错，因为类型boolean参数不能赋值给string|number的类型
//这个就是元组对越界元素的处理

arr.push('111',2222)//这种就可以
//也可以对二维数组进行限制规定类型

```

枚举类型(TS -- 10)

在 JavaScript 中是没有 枚举 的概念的 TS 帮我们定义了枚举这个类型

enum 关键字定义枚举

数字定义枚举

默认从0开始的

```
enum color{
    red,
    green,
    blue
}
console.log(Color.red,Color.blue,Color.green)//能够得到他们的顺序数字，这里返回0，2，1
```

增长枚举

能够通过自定义开头决定从哪个数字开始枚举，其他位置的都可以定义，后面的数字就按顺序枚举

```
enum Color{
    red=2,
    green,
    blue
}
console.log(Color.red,Color.blue,Color.green)//能够得到他们的顺序数字，这里返回2，4，3
```

字符串枚举

字符串枚举的概念很简单。在一个字符串枚举里，每个成员都必须用字符串字面量，或另外一个字符串枚举成员进行初始化。

由于字符串枚举没有自增长的行为，字符串枚举可以很好的序列化。换句话说，如果你正在调试并且必须要读一个数字枚举的运行时的值，这个值通常是很难读的 - 它并不能表达有用的信息，字符串枚举允许你提供一个运行时有意义的并且可读的值，独立于枚举成员的名字。

```
enum Types{
    Red = 'red',
    Green = 'green',
    BLue = 'blue'
}
```

异构枚举

枚举可以混合字符串和数字成员

```
enum Types{
  No = "No",
  Yes = 1,
}

console.log(Types.No,Types.Yes)
```

接口枚举

定义一个枚举 Types 定义一个接口 A 他有一个属性 red 值为 Types.yes

声明对象的时候要遵循这个规则

```
enum Color{
  no = "NO",
  yes = 1
}

interface A{
  red:Color.yes
}

let B:A{
  red:Color.yes
  //或者直接red:1, 只能填入这两个内容其中之一, 其他的会报错
}
```

const枚举

let 和 var 都是不允许的声明只能使用 const

大多数情况下, 枚举是十分有效的方案。然而在某些情况下需求很严格。为了避免在额外生成的代码上的开销和额外的非直接的对枚举成员的访问, 我们可以使用 `const` 枚举。常量枚举通过在枚举上使用 `const` 修饰符来定义

const 声明的枚举会被编译成常量

普通声明的枚举编译完后是个对象

```
const enum Types{//有没有const决定是编译成对象还是编译成常量
  sucess,
  fail
}

let code:number = 0
if(code === sucess){//是能执行的
  console.log("我在人民广场吃炸鸡")
}
```

反向映射

它包含了正向映射 (`name -> value`) 和反向映射 (`value -> name`)

要注意的是 不会为字符串枚举成员生成反向映射。

```
enum Types{
    one
}
let success:number = Types.success

console.log(success)//读取得出来为0
```

```
enum Types{
    success
}
let success:number = Types.success

let key = Types[success]

console.log(`value---${success}`, `key---${key}`)//value---0,key---success
```

类型推论 | 类型别名(TS -- 11)

类型推论

我声明了一个变量但是没有定义类型

TypeScript 会在没有明确的指定类型的时候推测出一个类型，这就是类型推论

```
let str = "小满"
str = 123//会报错，虽然我们没用明确限制类型，但是TS编辑器会自动推论为string类型。就不能够在赋值给别的类型
```

如果你声明变量没有定义类型也没有赋值这时候 TS 会推断成 any 类型可以进行任何操作

```
let str//为any类型
str = 123
str = "马杀鸡"
str = false
str = []
```

联合类型

指定多种类型，在前文有提到

```
type s = string|number
let str:s = "永恒的紫罗兰花园"
let num:s = 520//这有这两种类型可以
```

函数式的类型别名

type 关键字（可以给一个类型定义一个名字）多用于符合类型，但也可以要求有固定的东西

定义类型别名

```
type str = string

let s:str = "我是小满"

console.log(s);
```

定义函数别名

```
type str = () => string

let s: str = () => "我是小满"

console.log(s);
```

定义联合类型别名

```
type str = string | number

let s: str = 123

let s2: str = '123'

console.log(s,s2);
```

定义值的别名

```
type value = boolean | 0 | '213'

let s:value = true
//变量s的值 只能是上面value定义的值
```

never类型(TS -- 12)

TypeScript将使用 never 类型来表示不应该存在的状态

返回never的函数必须存在无法达到的终点

```
function error(message:string):never { //因为必定抛出异常，所以 error 将不会有返回值
    throw new Error(message)
}

function loop():never{
    while(true){
        //因为这个是死循环，永远不去返回的
    }
}
```

```
interface A{
    type:"保安"
```

```

}

interface B{
  type:"草莓"
}

interface C{
  type:"卷心菜"
}

type All = A|B
function type(val:All){
  while(val.type){
    case "保安":break
    case "草莓":break
    //case "卷心菜":break
    default://兜底机制，此时C没有用上就会报错提示。这就算never的作用
    const check:never = val
    break
  }
}

```

Symbol类型

symbol 是一种新的原生类型，就像 number 和 string 一样

symbol 类型的值是通过 Symbol构造函数创建的

可以传递参做为唯一标识 只支持 string 和 number 类型的参数

```

let s:symbol = Symbol('小满')
let num:symbol = Symbol('小满')

let obj = {
  [num] : "value",//Symbol
  [s] : "草莓",//Symbol
  name:"小满",
  sex:"男"
}

console.log(obj[num])//取到value
console.log(s,num)//返回Symbol(小满)Symbol(小满)
console.log(s === num)//false
//这个值看似一样，其实因为内存地址指针位置不同，所以是唯一值

for(let key in obj){
  console.log(key)
}//只会打印出name跟sex，[num]与[s]将打印不出来

console.log(Object.keys(obj))//["name","sex"]
console.log(Object.getOwnPropertyNames(obj))//["name","sex"]，跟上面一样，打印不出来
console.log(JSON.stringify(obj));//[{"name":"小满","sex":"男"}]，一样打印不出来

```

能够读取到Symbol的两种方式

静态方法 `Reflect.ownKeys()` 返回一个由目标对象自身的属性键组成的数组(Array)。

语法: `Reflect.ownKeys(target)` => `target` 获取自身属性键的目标对象。

`Reflect.ownKeys` 方法返回一个由目标对象自身的属性键组成的数组。它的返回值等同于 `Object.getOwnPropertyNames(target).concat(Object.getOwnPropertySymbols(target))`。

```
console.log(Object.getOwnPropertySymbols(obj)); //能打印出来两个Symbol，另外两个普通的不会打印出来
```

```
Reflect.ownKeys() //此属性是将所有的属性都列出来  
console.log(Reflect.ownKeys()) //四个全部圆满的打印出来
```

迭代器 | 生成器(TS -- 13)

迭代器: `Symbol.iterator`

迭代器 (Iterator) 是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址

通俗点说，迭代器表现的像指针，读取集合或者数组中的一个值，读完以后又指向下一条数据，一个个数过去。

生成器: `for of`

迭代器

迭代器 `Iterator` 的用法

1. `Iterator` 是 es6 引入的一种新的遍历机制。两个核心：

(1) 迭代器是一个统一的接口，它的作用是使各种 `数据结构` 可被便捷的访问，它是通过一个键为 `Symbol.iterator` 的方法来实现。

(2) 迭代器是用于 `遍历` 数据结构元素的指针（如数据库中的游标）。

使用迭代

1. 使用 `Symbol.iterator` 创建一个迭代器
2. 调用 `next` 方法向下迭代，`next` 方法会返回当前的位置
3. 当 `done` 为 `true` 时则遍历结束

注意点：

1. 在迭代器迭代元素的过程中，不允许使用集合对象改变集合中的元素个数，如果需要添加或者删除只能使用迭代器的方法操作。
2. 如果使用了集合对象改变集合中的元素个数那么就会报错：不改变个数即可，替换也可以的。
3. 迭代器的生存周期为创建到使用结束的时段。
4. `foreach` : `Iterator` 的封装变形，变得比 `Iterator` 更简单。但是他也需要知道数组或集合的类型。并且，`Iterator` 需要注意的，`foreach` 同样需要注意。

- 存在 `iterator` 迭代器的有 =>
 - 数组 `[]` 里能够找到 `Symbol(Symbol.iterator)`

- argument内找到 Symbol(Symbol.iterator)
- NodeList内找到Symbol(Symbol.iterator)
- new set()内的Prototype下一层Symbol(Symbol.iterator)
- new Map同理，一样有Symbol(Symbol.iterator)

```
let arr:Array<number> = [1,5,6]
let it:Iterator<number> = arr[Symbol.iterator]()//注意这里的接收类型<number>是固定要写的
```

```
//next一次只遍历一个数，下一次调用将从上一次遍历到的位置开始下一个
console.log(iterator.next()); //{ value: 1, done: false }
console.log(iterator.next()); //{ value: 5, done: false }
console.log(iterator.next()); //{ value: 6, done: false }
console.log(iterator.next()); //{ value: undefined, done: true }
//返回的有两个属性，一个value，一个done。value当读取到值的时候，done为false、读取不到为true
```

type mapKeys = string|number//相当于起别名，在下方使用的时候集合了string与number就会相对方便不少

```
let set:Set<number> = new Set([1,2,3])
let map:Map<mapKeys,mapKeys> = new Map()//这里断言两个mapKeys，一个对应key，一个对应value
map.set('1','小满')
map.set('2','看看腿')

//小迭代器的实现
function gen(erg:any){//这里定义为any类型是因为上面要传到这里的多有不同类型
    let it:iterator<any> = erg[Symbol.iterator]()
    let next:any = {done:false}
    while(!next.done){//判断next，由于next默认为false，while循环只有true会通过，所以需要取反
        next = it.next()//刚开始是声明next给个默认值，等到开始循环的时候再把真正的值赋给他
        if(!next.done){
            console.log(next);
        }
    }
}
```

gen(arr)//调用第一个代码块的arr，输出了与console.log(iterator.next());一样的内容
//对象是不支持迭代器的使用的，其实我们在控制台输出一个对象，查找他内置的属性，也是找不到Symbol.iterator的

Symbol列表

Symbol.hasInstance

方法，会被 instanceof 运算符调用。构造器对象用来识别一个对象是否是其实例。

Symbol.isConcatSpreadable

布尔值，表示当在一个对象上调用 Array.prototype.concat 时，这个对象的数组元素是否可展开。

Symbol.iterator

方法，被 for-of 语句调用。返回对象的默认迭代器。

Symbol.match

方法，被 String.prototype.match 调用。正则表达式用来匹配[字符串](#)。

Symbol.replace

方法，被 String.prototype.replace 调用。正则表达式用来替换字符串中匹配的子串。

Symbol.search

方法，被 String.prototype.search 调用。正则表达式返回被匹配部分在字符串中的索引。

Symbol.species

函数值，为一个构造函数。用来创建派生对象。

Symbol.split

方法，被 String.prototype.split 调用。正则表达式用来分割字符串。

Symbol.toPrimitive

方法，被 ToPrimitive 抽象操作调用。把对象转换为相应的原始值。

Symbol.toStringTag

方法，被内置方法 Object.prototype.toString 调用。返回创建对象时默认的字符串描述。

Symbol.unscopables

对象，它自己拥有的属性会被 with 作用域排除在外。

生成器 (Builder)

又称建造者模式，该模式是一种创建型设计模式，能够分步骤创建复杂对象。该模式允许使用相同的创建代码生成不同类型的对象。

具体内容请跳转链接查看：[设计模式-生成器](#)

for...of 语句

for...of 会遍历可迭代的对象，调用对象上的 `Symbol.iterator` 方法。(此对象非彼对象，这个对象是指你即将下手的目标)

对象也是不支持的，因为对象没用 `Symbol.iterator` 方法。

```
type mapKeys = string|number//相当于起别名，在下方使用的时候集合了string与number就会相对方便不少
```

```
let set:Set<number> = new Set([1,2,3])
let map:Map<mapKeys,mapKeys> = new Map()//这里断言两个mapKeys，一个对应key，一个对应value
map.set('1','小满')
map.set('2','看看腿')
```

```
for (let item of set){
  console.log(item)
}//打印出1 2 3
```

```
for (let item of arr){
  console.log(item)
}//打印出4 5 6
```

```
for (let item of map){
```

```
console.log(item)
} //打印出['1','小满']    ['2','看看腿']
//其实这就是一个语法糖，将of后面的内容遍历存储到of前面的变量中
```

跟for in 的区别

for in循环出来的是索引而不是内容，这个应该是最本质的区别了

因为for of会调用底层iterator里面那个list的.value

泛型(generic) => (TS -- 14上)

泛型简单来说就是类型变量，在 ts 中存在类型，如 number、string、boolean等。泛型就是使用一个类型变量来表示一种类型，类型值通常是在使用的时候才会设置。泛型的使用场景非常多，可以在函数、类、interface 接口中使用

TypeScript中不建议使用 any 类型，不能保证类型安全，调试时缺乏完整的信息。

TypeScript 可以使用泛型来创建可重用的组件。支持当前数据类型，同时也能支持未来的数据类型。扩展灵活，可以在编译时发现类型错误，从而保证了类型安全。

无泛型用法

```
//数字类型
function num(A:number,B:number):Array<number>{//Array<number>为希望返回number类型的数组
    return [a,b]
}
num(6,9)
//字符串类型
function str(A:string,B:string):Array<string>{//Array<number>为希望返回number类型的数组
    return [a,b]
}
str('小满','穿女装')
```

一个笨的方法就像上面那样，也就是说 JS 提供多少种类型，就需要复制多少份代码，然后改下类型签名。这对程序员来说是致命的。这种复制粘贴增加了出错的概率，使得代码难以维护，牵一发而动全身。并且将来 JS 新增新的类型，你仍然需要修改代码，也就是说你的代码**对修改开放**，这样不好。

如果你使用 any 的话，怎么写都是 ok 的，这就丧失了类型检查的效果。实际上我知道我传给你的是 string，返回来的也一定是 string，而 string 上没有 toFixed 方法，因此需要报错才是我想要的。也就是说我真正想要的效果是：当我用到id的时候，你根据我传给你的类型进行推导。比如我传入的是 string，但是使用了 number 上的方法，你就应该报错。

使用泛型优化

为了解决上面的这些问题，我们使用泛型对上面的代码进行重构。和我们的定义不同，这里用了一个类型 T，这个 T 是一个抽象类型，只有在调用的时候才确定它的值，这就不用我们复制粘贴无数份代码了。

其中 T 代表 Type，在定义泛型时通常用作第一个类型变量名称。但实际上 T 可以用任何有效名称代替。除了 T 之外，以下是常见泛型变量代表的意思：

- K (Key) : 表示对象中的键类型;
- V (Value) : 表示对象中的值类型;
- E (Element) : 表示元素类型。

```
function add<T>(a:T,b:T):Array<T>{//通常定义的时候类型是不明确的，所以一般使用T来定义
    return [a,b];
}

add<number>(1,2)//1对应a，2对应b、返回的都是number类型
add<string>('1','2')//这个时候，我们只需要改动这个string，传递到上面的时候就会自动推断为string类型了

//甚至我们可以简写
add(1,2)
add('1','2')//编辑器会自动推断类型，但最好还是写一下，如果你知道你具体需要的是什么的的话

//对泛型进行总结就是：定义前不明确类型，使用的时候再明确类型，能够给我们保留有足够的自由度，又不会像any丧失类型检查的效果
```

我们也可以使用不同的泛型参数名，只要在数量上和使用方式上能对应上就可以。

```
function Sub<T,U>(a:T,b:U):Array<T|U> { //这个T跟U随便起名字都行，没有强制规范
    const params:Array<T|U> = [a,b]
    return params
}

Sub<Boolean,number>(false,1)//我们这里就将其定义为布尔值类型跟数字类型
```

定义泛型接口

声明接口的时候 在名字后面加一个 <参数>

使用的时候传递类型

```
interface MyInter<T> {
    (arg: T): T
}

function fn<T>(arg: T): T {
    return arg
}

let result: MyInter<number> = fn

result(123)
```

对象字面量泛型

```
let foo: { <T>(arg: T): T }

foo = function <T>(arg:T):T {
  return arg
}

foo(123)
```

泛型约束(函数类)

我们期望在一个泛型的变量上面，获取其 `length` 参数，但是，有的数据类型是没有 `length` 属性的

```
function getLegnth<T>(arg:T) {
  return arg.length
}
```

- 这个时候，我们就可以对其进行约束

```
interface Len{
  length:number
}

function getLegnth<T extends Len>(arg:T) { //使用接口让泛型T继承了Len
  return arg.length
}

getLength(1) //这个时候我们这样使用就会提示我们类型"number"的参数不能赋给"Len"的参数
//我们依次对数组、字符串、布尔值都进行尝试，分别为可以、可以、不可以
```

泛型约束 | 泛型类(TS -- 14下)

使用 keyof 约束对象

其中使用了 TS 泛型和泛型约束。首先定义了 T 类型并使用 `extends` 关键字继承 `object` 类型的子类型，然后使用 `keyof` 操作符获取 T 类型的所有键，它的返回 类型是联合 类型，最后利用 `extends` 关键字约束 K 类型必须为 `keyof T` 联合类型的子类型

```
function prop<T, K extends keyof T>(obj: T, key: K) {
  return obj[key]
}

let o = { a: 1, b: 2, c: 3 }

prop(o, 'a')
prop(o, 'd') //，我们需要约束一下这个o里面并没有的东西，此时就会报错发现找不到
//通过提示，我们可以看到类型"d"的参数不能赋给类型"a"|"b"|"c"的参数
```

泛型类

声明方法跟函数类似名称后面定义 `<类型>`

使用的时候确定类型 new Sub()

```
//定义泛型的一个类
class Sub<T>{
    attr:T[] = []//这里的:只是普通的:
    add(a:T):T[]{
        return [a]
    }
}

let s = new Sub<number>()//这里已经使用泛型固定为number了
s.attr = [123]//正常运行
s.attr = ['123']//报错
s.add(123)//也是只能传数字

let str = new Sub<string>()//这里已经使用泛型固定为string了
str.attr = [123]//报错
str.attr = ['123']//正常运行
str.add('123')//也是只能传字符串

console.log(s, str)
```

泛型工具类型(大量补充额外内容)

作者使用了Typora作为写笔记的编辑器，这里可以对目录进行折叠方便我们查阅我们想要的部分



为了方便开发者 TypeScript 内置了一些常用的工具类型，比如 Partial、Required、Readonly、Record 和 ReturnType 等。不过在具体介绍之前，我们得先介绍一些相关的基础知识，方便读者可以更好的学习其它的工具类型。

1.typeof

typeof 的主要用途是在类型上下文中获取变量或者属性的类型，下面我们通过一个具体示例来理解一下。

```
interface Person {
    name: string;
    age: number;
}

const sem: Person = { name: "semlinker", age: 30 };
type Sem = typeof sem; // type Sem = Person
```

在上面代码中，我们通过 `typeof` 操作符获取 `sem` 变量的类型并赋值给 `Sem` 类型变量，之后我们就可以使用 `Sem` 类型：

```
const lol: Sem = { name: "lol", age: 5 }
```

你也可以对嵌套对象执行相同的操作:

```
const Message = {
  name: "jimmy",
  age: 18,
  address: {
    province: '四川',
    city: '成都'
  }
}
type message = typeof Message;
/*
type message = {
  name: string;
  age: number;
  address: {
    province: string;
    city: string;
  };
}
*/
```

此外, `typeof` 操作符除了可以获取对象的结构类型之外, 它也可以用来获取函数对象的类型, 比如:

```
function toArray(x: number): Array<number> {
  return [x];
}
type Func = typeof toArray; // -> (x: number) => number[]
```

2.keyof

`keyof` 操作符是在 TypeScript 2.1 版本引入的, 该操作符可以用于获取某种类型的所有键, 其返回类型是联合类型。

```
interface Person {
  name: string;
  age: number;
}

type K1 = keyof Person; // "name" | "age"
type K2 = keyof Person[]; // "length" | "toString" | "pop" | "push" | "concat" | "join"
type K3 = keyof { [x: string]: Person }; // string | number
```

在 TypeScript 中支持两种索引签名, 数字索引和字符串索引:

```
interface StringArray {
    // 字符串索引 -> keyof StringArray => string | number
    [index: string]: string;
}

interface StringArray1 {
    // 数字索引 -> keyof StringArray1 => number
    [index: number]: string;
}
```

为了同时支持两种索引类型，就得要求数字索引的返回值必须是字符串索引返回值的子类。**其中的原因就是当使用数值索引时，JavaScript 在执行索引操作时，会先把数值索引先转换为字符串索引。**所以 `keyof { [x: string]: Person }` 的结果会返回 `string | number`。

`keyof` 也支持基本数据类型：

```
let K1: keyof boolean; // let K1: "valueOf"
let K2: keyof number; // let K2: "toString" | "toFixed" | "toExponential" | ...
let K3: keyof symbol; // let K1: "valueOf"
```

keyof 的作用

JavaScript 是一种高度动态的语言。有时在静态类型系统中捕获某些操作的语义可能会很棘手。以一个简单的 `prop` 函数为例：

```
function prop(obj, key) {
    return obj[key];
}
```

该函数接收 `obj` 和 `key` 两个参数，并返回对应属性的值。对象上的不同属性，可以具有完全不同的类型，我们甚至不知道 `obj` 对象长什么样。

那么在 TypeScript 中如何定义上面的 `prop` 函数呢？我们来尝试一下：

```
function prop(obj: object, key: string) {
    return obj[key];
}
```

在上面代码中，为了避免调用 `prop` 函数时传入错误的参数类型，我们为 `obj` 和 `key` 参数设置了类型，分别为 `{}` 和 `string` 类型。然而，事情并没有那么简单。针对上述的代码，TypeScript 编译器会输出以下错误信息：

```
Element implicitly has an 'any' type because expression of type 'string' can't
be used to index type '{}'.

```

元素隐式地拥有 `any` 类型，因为 `string` 类型不能被用于索引 `{}` 类型。要解决这个问题，你可以使用以下非常暴力的方案：

```
function prop(obj: object, key: string) {
  return (obj as any)[key];
}
```

很明显该方案并不是一个好的方案，我们来回顾一下 `prop` 函数的作用，该函数用于获取某个对象中指定属性的属性值。因此我们期望用户输入的属性是对象上已存在的属性，那么如何限制属性名的范围呢？这时我们可以利用本文的主角 `keyof` 操作符：

```
function prop<T extends object, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}
```

在以上代码中，我们使用了 TypeScript 的泛型和泛型约束。首先定义了 `T` 类型并使用 `extends` 关键字约束该类型必须是 `object` 类型的子类型，然后使用 `keyof` 操作符获取 `T` 类型的所有键，其返回类型是联合类型，最后利用 `extends` 关键字约束 `K` 类型必须为 `keyof T` 联合类型的子类型。是骡子是马拉出来遛遛就知道了，我们来实际测试一下：

```
type Todo = {
  id: number;
  text: string;
  done: boolean;
}

const todo: Todo = {
  id: 1,
  text: "Learn TypeScript keyof",
  done: false
}

function prop<T extends object, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}

const id = prop(todo, "id"); // const id: number
const text = prop(todo, "text"); // const text: string
const done = prop(todo, "done"); // const done: boolean
```

很明显使用泛型，重新定义后的 `prop<T extends object, K extends keyof T>(obj: T, key: K)` 函数，已经可以正确地推导出指定键对应的类型。那么当访问 `todo` 对象上不存在的属性时，会出现什么情况？比如：

```
const date = prop(todo, "date");
```

对于上述代码，TypeScript 编译器会提示以下错误：


```
Argument of type '"date"' is not assignable to parameter of type '"id" | "text" | "done"'.
```

这就阻止我们尝试读取不存在的属性。

3.in

`in` 用来遍历枚举类型：

```
type Keys = "a" | "b" | "c"

type Obj = {
  [p in Keys]: any
} // -> { a: any, b: any, c: any }
```

4.infer

在条件类型语句中，可以用 `infer` 声明一个类型变量并且对它进行使用。

```
type ReturnType<T> = T extends (
  ...args: any[]
) => infer R ? R : any;
```

以上代码中 `infer R` 就是声明一个变量来承载传入函数签名的返回值类型，简单说就是用它取到函数返回值的类型方便之后使用。

5.extends

有时候我们定义的泛型不想过于灵活或者说想继承某些类等，可以通过 `extends` 关键字添加泛型约束。

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
```

现在这个泛型函数被定义了约束，因此它不再是适用于任意类型：

```
loggingIdentity(3); // Error, number doesn't have a .length property
```

这时我们需要传入符合约束类型的值，必须包含 `length` 属性：

```
loggingIdentity({length: 10, value: 3});
```

索引类型

在实际开发中，我们经常能遇到这样的场景，在对象中获取一些属性的值，然后建立对应的集合。

```
let person = {
  name: 'musion',
  age: 35
}

function getValues(person: any, keys: string[]) {
  return keys.map(key => person[key])
}

console.log(getValues(person, ['name', 'age'])) // ['musion', 35]
console.log(getValues(person, ['gender'])) // [undefined]
```

在上述例子中，可以看到 `getValues(person, ['gender'])` 打印出来的是 `[undefined]`，但是 `ts` 编译器并没有给出报错信息，那么如何使用 `ts` 对这种模式进行类型约束呢？这里就要用到了索引类型，改造一下 `getValues` 函数，通过 **索引类型查询**和 **索引访问** 操作符：

```
function getValues<T, K extends keyof T>(person: T, keys: K[]): T[K][] {
  return keys.map(key => person[key]);
}

interface Person {
  name: string;
  age: number;
}

const person: Person = {
  name: 'musion',
  age: 35
}

getValues(person, ['name']) // ['musion']
getValues(person, ['gender']) // 报错:
// Argument of Type '"gender"[]' is not assignable to parameter of type '("name" | "age")[]'.
// Type "gender" is not assignable to type "name" | "age".
```

编译器会检查传入的值是否是 `Person` 的一部分。通过下面的概念来理解上面的代码：

`T[K]`表示对象`T`的属性`K`所表示的类型，在上述例子中，`T[K][]` 表示变量`T`取属性`K`的值的数组

```
// 通过[]索引类型访问操作符，我们就能得到某个索引的类型
class Person {
    name:string;
    age:number;
}
type MyType = Person['name']; //Person中name的类型为string type MyType = string
```

介绍完概念之后，应该就可以理解上面的代码了。首先看泛型，这里有 `T` 和 `K` 两种类型，根据类型推断，第一个参数 `person` 就是 `person`，类型会被推断为 `Person`。而第二个数组参数的类型推断 (`K extends keyof T`)，`keyof` 关键字可以获取 `T`，也就是 `Person` 的所有属性名，即 `['name', 'age']`。而 `extends` 关键字让泛型 `K` 继承了 `Person` 的所有属性名，即 `['name', 'age']`。这三个特性组合保证了代码的动态性和准确性，也让代码提示变得更加丰富了

```
getValues(person, ['gender']) // 报错:
// Argument of Type '"gender"[]' is not assignable to parameter of type '("name" | "age")[]'.
// Type "gender" is not assignable to type "name" | "age".
```

映射类型

根据旧的类型创建出新的类型，我们称之为映射类型

比如我们定义一个接口

```
interface TestInterface{
    name:string,
    age:number
}
```

我们把上面定义的接口里面的属性全部变成可选

```
// 我们可以通过+/-来指定添加还是删除

type OptionalTestInterface<T> = {
    [p in keyof T]? :T[p]
}

type newTestInterface = OptionalTestInterface<TestInterface>
// type newTestInterface = {
//     name?:string,
//     age?:number
// }
```

比如我们再加上只读

```

type OptionalTestInterface<T> = {
  +readonly [p in keyof T]: T[p]
}

type newTestInterface = OptionalTestInterface<TestInterface>
// type newTestInterface = {
//   readonly name?: string,
//   readonly age?: number
// }

```

由于生成只读属性和可选属性比较常用，所以 TS 内部已经给我们提供了现成的实现 Readonly / Partial，会面内置的工具类型会介绍。

内置的工具类型

Partial

`Partial<T>` 将类型的属性变成可选

定义

```

type Partial<T> = {
  [P in keyof T]?: T[P];
};

```

在以上代码中，首先通过 `keyof T` 拿到 `T` 的所有属性名，然后使用 `in` 进行遍历，将值赋给 `P`，最后通过 `T[P]` 取得相应的属性值的类。中间的 `?` 号，用于将所有属性变为可选。

举例说明

```

interface UserInfo {
  id: string;
  name: string;
}
// error: Property 'id' is missing in type '{ name: string; }' but required in
type 'UserInfo'
const xiaoming: UserInfo = {
  name: 'xiaoming'
}

```

使用 `Partial<T>`

```

type NewUserInfo = Partial<UserInfo>;
const xiaoming: NewUserInfo = {
  name: 'xiaoming'
}

```

这个 `NewUserInfo` 就相当于

```
interface NewUserInfo {
  id?: string;
  name?: string;
}
```

但是 `Partial<T>` 有个局限性，就是只支持处理第一层的属性，如果我的接口定义是这样的

```
interface UserInfo {
  id: string;
  name: string;
  fruits: {
    appleNumber: number;
    orangeNumber: number;
  }
}

type NewUserInfo = Partial<UserInfo>;

// Property 'appleNumber' is missing in type '{ orangeNumber: number; }' but
// required in type '{ appleNumber: number; orangeNumber: number; }'.
const xiaoming: NewUserInfo = {
  name: 'xiaoming',
  fruits: {
    orangeNumber: 1,
  }
}
```

可以看到，第二层以后就不会处理了，如果要处理多层，就可以自己实现

DeepPartial

```
type DeepPartial<T> = {
  // 如果是 object，则递归类型
  [U in keyof T]?: T[U] extends object
    ? DeepPartial<T[U]>
    : T[U]
};

type PartialWindow = DeepPartial<T>; // 现在T上所有属性都变成了可选啦
```

Required

Required 将类型的属性变成必选

定义

```
type Required<T> = {
  [P in keyof T]-?: T[P]
};
```

其中 `-?` 是代表移除 `?` 这个 modifier 的标识。再拓展一下，除了可以应用于 `?` 这个 modifiers，还有应用在 `readonly`，比如 `Readonly<T>` 这个类型

```
type Readonly<T> = {  
  readonly [p in keyof T]: T[p];  
}
```

Readonly

`Readonly<T>` 的作用是将某个类型所有属性变为只读属性，也就意味着这些属性不能被重新赋值。

定义

```
type Readonly<T> = {  
  readonly [P in keyof T]: T[P];  
};
```

举例说明

```
interface Todo {  
  title: string;  
}  
  
const todo: Readonly<Todo> = {  
  title: "Delete inactive users"  
};  
  
todo.title = "Hello"; // Error: cannot reassign a readonly property
```

Pick

Pick 从某个类型中挑出一些属性出来

定义

```
type Pick<T, K extends keyof T> = {  
  [P in K]: T[P];  
};
```

举例说明

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Pick<Todo, "title" | "completed">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
};
```

可以看到 NewUserInfo 中就只有个 name 的属性了。

Record

`Record<K extends keyof any, T>` 的作用是将 `K` 中所有的属性的值转化为 `T` 类型。

定义

```
type Record<K extends keyof any, T> = {
  [P in K]: T;
};
```

举例说明

```
interface PageInfo {
  title: string;
}

type Page = "home" | "about" | "contact";

const x: Record<Page, PageInfo> = {
  about: { title: "about" },
  contact: { title: "contact" },
  home: { title: "home" },
};
```

ReturnType

用来得到一个函数的返回值类型

定义

```

type ReturnType<T extends (...args: any[]) => any> = T extends (
  ...args: any[]
) => infer R
  ? R
  : any;

```

`infer` 在这里用于提取函数类型的返回值类型。`ReturnType<T>` 只是将 `infer R` 从参数位置移动到返回值位置，因此此时 `R` 即是表示待推断的返回值类型。

举例说明

```

type Func = (value: number) => string;
const foo: ReturnType<Func> = "1";

```

`ReturnType` 获取到 `Func` 的返回值类型为 `string`，所以，`foo` 也就只能被赋值为字符串了。

Exclude

`Exclude<T, U>` 的作用是将某个类型中属于另一个的类型移除掉。

定义

```

type Exclude<T, U> = T extends U ? never : T;

```

如果 `T` 能赋值给 `U` 类型的话，那么就会返回 `never` 类型，否则返回 `T` 类型。最终实现的效果就是将 `T` 中某些属于 `U` 的类型移除掉。

举例说明

```

type T0 = Exclude<"a" | "b" | "c", "a">; // "b" | "c"
type T1 = Exclude<"a" | "b" | "c", "a" | "b">; // "c"
type T2 = Exclude<string | number | (() => void), Function>; // string | number

```

Extract

`Extract<T, U>` 的作用是从 `T` 中提取出 `U`。

定义

```

type Extract<T, U> = T extends U ? T : never;

```

举例说明

```

type T0 = Extract<"a" | "b" | "c", "a" | "f">; // "a"
type T1 = Extract<string | number | (() => void), Function>; // () => void

```


Omit

`Omit<T, K extends keyof any>` 的作用是使用 `T` 类型中除了 `K` 类型的所有属性，来构造一个新的类型。

定义

```
type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>;
```

举例说明

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
};
```

NonNullable

`NonNullable<T>` 的作用是用来过滤类型中的 `null` 及 `undefined` 类型。

定义

```
type NonNullable<T> = T extends null | undefined ? never : T;
```

举例说明

```
type T0 = NonNullable<string | number | undefined>; // string | number
type T1 = NonNullable<string[] | null | undefined>; // string[]
```

Parameters

`Parameters<T>` 的作用是为了获得函数的参数类型组成的元组类型。

定义

```
type Parameters<T extends (...args: any) => any> = T extends (...args: infer P)
=> any
? P : never;
```

举例说明

```
type A = Parameters<() =>void>; // []
type B = Parameters<typeofArray.isArray>; // [any]
type C = Parameters<typeofparseInt>; // [string, (number | undefined)?]
type D = Parameters<typeofMath.max>; // number[]
```

tsconfig.json配置文件(TS -- 15)

生成 tsconfig.json 文件

这个文件是通过 `tsc --init` 命令生成的

tsconfig.json 是 TypeScript 项目的配置文件。如果一个目录下存在一个 tsconfig.json 文件，那么往往意味着这个目录就是 TypeScript 项目的根目录。

tsconfig.json 包含 TypeScript 编译的相关配置，通过更改编译配置项，我们可以让 TypeScript 编译出 ES6、ES5、node 的代码。

配置详解

```
"compilerOptions": {
  "incremental": true, // TS编译器在第一次编译之后会生成一个存储编译信息的文件，第二次编译
  // 会在第一次的基础上进行增量编译，可以提高编译的速度
  "tsBuildInfoFile": "./buildFile", // 增量编译文件的存储位置
  "diagnostics": true, // 打印诊断信息
  "target": "ES5", // 目标语言的版本
  "module": "CommonJS", // 生成代码的模板标准
  "outFile": "./app.js", // 将多个相互依赖的文件生成一个文件，可以用在AMD模块中，即开启时应
  // 设置"module": "AMD",
  "lib": ["DOM", "ES2015", "ScriptHost", "ES2019.Array"], // TS需要引用的库，即声明
  // 文件，es5 默认引用dom、es5、scripthost,如需要使用es的高级版本特性，通常都需要配置，如es8的数
  // 组新特性需要引入"ES2019.Array",
  "allowJS": true, // 允许编译器编译JS，JSX文件
  "checkJs": true, // 允许在JS文件中报错，通常与allowJS一起使用
  "outDir": "./dist", // 指定输出目录
  "rootDir": "./", // 指定输出文件目录(用于输出)，用于控制输出目录结构
  "declaration": true, // 生成声明文件，开启后会自动生成声明文件
  "declarationDir": "./file", // 指定生成声明文件存放目录
  "emitDeclarationOnly": true, // 只生成声明文件，而不会生成js文件
  "sourceMap": true, // 生成目标文件的sourceMap文件
  "inlineSourceMap": true, // 生成目标文件的inline SourceMap, inline SourceMap会包含
  // 在生成的js文件中
  "declarationMap": true, // 为声明文件生成sourceMap
  "typeRoots": [], // 声明文件目录，默认时node_modules/@types
  "types": [], // 加载的声明文件包
  "removeComments": true, // 删除注释
  "noEmit": true, // 不输出文件,即编译后不会生成任何js文件
  "noEmitOnError": true, // 发送错误时不输出任何文件
  "noEmitHelpers": true, // 不生成helper函数，减小体积，需要额外安装，常配合
  // importHelpers一起使用
  "importHelpers": true, // 通过tslib引入helper函数，文件必须是模块
```

```

    "downlevelIteration": true, // 降级遍历器实现，如果目标源是es3/5，那么遍历器会有降级的实现
    "strict": true, // 开启所有严格的类型检查
    "alwaysStrict": true, // 在代码中注入'use strict'
    "noImplicitAny": true, // 不允许隐式的any类型
    "strictNullChecks": true, // 不允许把null、undefined赋值给其他类型的变量
    "strictFunctionTypes": true, // 不允许函数参数双向协变
    "strictPropertyInitialization": true, // 类的实例属性必须初始化
    "strictBindCallApply": true, // 严格的bind/call/apply检查
    "noImplicitThis": true, // 不允许this有隐式的any类型
    "noUnusedLocals": true, // 检查只声明、未使用的局部变量(只提示不报错)
    "noUnusedParameters": true, // 检查未使用的函数参数(只提示不报错)
    "noFallthroughCasesInSwitch": true, // 防止switch语句贯穿(即如果没有break语句后面不会执行)
    "noImplicitReturns": true, //每个分支都会有返回值
    "esModuleInterop": true, // 允许export=导出，由import from 导入
    "allowUmdGlobalAccess": true, // 允许在模块中全局变量的方式访问umd模块
    "moduleResolution": "node", // 模块解析策略，ts默认用node的解析策略，即相对的方式导入
    "baseUrl": "./", // 解析非相对模块的基地址，默认是当前目录
    "paths": { // 路径映射，相对于baseUrl
        // 如使用jq时不想使用默认版本，而需要手动指定版本，可进行如下配置
        "jquery": ["node_modules/jquery/dist/jquery.min.js"]
    },
    "rootDirs": ["src","out"], // 将多个目录放在一个虚拟目录下，用于运行时，即编译后引入文件的位置可能发生变化，这也设置可以虚拟src和out在同一个目录下，不用再去改变路径也不会报错
    "listEmittedFiles": true, // 打印输出文件
    "listFiles": true // 打印编译的文件(包括引用的声明文件)
}

// 指定一个匹配列表（属于自动指定该路径下的所有ts相关文件）
"include": [
    "src/**/*"
],
// 指定一个排除列表（include的反向操作）
"exclude": [
    "demo.ts"
],
// 指定哪些文件使用该配置（属于手动一个个指定文件）
"files": [
    "demo.ts"
]

```

上面的配置详解有点繁杂，我们或许可以将其分类一下

配置分类(compilerOptions 选项)

```

{
    "compilerOptions": {

        /* 基本选项 */
        "target": "es5", // 指定 ECMAScript 目标版本: 'ES3'
        (default), 'ES5', 'ES6'/'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'
        "module": "commonjs", // 指定使用模块: 'commonjs', 'amd',
        'system', 'umd' or 'es2015'
        "lib": [], // 指定要包含在编译中的库文件
    }
}

```

```

"allowJs": true,
"checkJs": true,
"jsx": "preserve",
'react-native', or 'react'
"declaration": true,
"sourceMap": true,
"outFile": "./",
"outDir": "./",
"rootDir": "./",
"removeComments": true,
"noEmit": true,
"importHelpers": true,
"isolatedModules": true,
'ts.transpileModule' 类似)。

/* 严格的类型检查选项 */
"strict": true,
"noImplicitAny": true,
"strictNullChecks": true,
"noImplicitThis": true,
一个错误
"alwaysStrict": true,
'use strict'

/* 额外的检查 */
"noUnusedLocals": true,
"noUnusedParameters": true,
"noImplicitReturns": true,
误
"noFallthroughCasesInSwitch": true,
（即，不允许 switch 的 case 语句贯穿）

/* 模块解析选项 */
"moduleResolution": "node",
'classic' (TypeScript pre-1.6)
"baseUrl": "./",
"paths": {},
"rootDirs": [],
构内容
"typeRoots": [],
"types": [],
"allowSyntheticDefaultImports": true,

/* Source Map Options */
"sourceRoot": "./",
是源文件的位置
"mapRoot": "./",
位置
"inlineSourceMap": true,
sourcemaps 生成不同的文件
"inlineSources": true,
要求同时设置了 --inlineSourceMap 或 --sourceMap 属性

/* 其他选项 */
"experimentalDecorators": true,

```

// 允许编译 javascript 文件
// 报告 javascript 文件中的错误
// 指定 jsx 代码的生成: 'preserve',

// 生成相应的 '.d.ts' 文件
// 生成相应的 '.map' 文件
// 将输出文件合并为一个文件
// 指定输出目录
// 用来控制输出目录结构 --outDir.
// 删除编译后的所有的注释
// 不生成输出文件
// 从 tslib 导入辅助工具函数
// 将每个文件做为单独的模块（与

// 启用所有严格类型检查选项
// 在表达式和声明上有隐含的 any 类型时报错
// 启用严格的 null 检查
// 当 this 表达式值为 any 类型的时候，生成

// 以严格模式检查每个模块，并在每个文件里加入

// 有未使用的变量时，抛出错误
// 有未使用的参数时，抛出错误
// 并不是所有函数里的代码都有返回值时，抛出错误

// 报告 switch 语句的 fallthrough 错误。

// 选择模块解析策略: 'node' (Node.js) or

// 用于解析非相对模块名称的基目录
// 模块名到基于 baseUrl 的路径映射的列表
// 根文件夹列表，其组合内容表示项目运行时的结

// 包含类型声明的文件列表
// 需要包含的类型声明文件名列表
// 允许从没有设置默认导出的模块中默认导入。

// 指定调试器应该找到 TypeScript 文件而不

// 指定调试器应该找到映射文件而不是生成文件的

// 生成单个 sourcemaps 文件，而不是将

// 将代码与 sourcemaps 生成到一个文件中，

// 启用装饰器

```
"emitDecoratorMetadata": true // 为装饰器提供元数据的支持
}
}
```

常用的配置

- 终端命令
 - echo ">index.ts(创建一个空的名叫index.ts的文件)
 - tsc -init(创建一个tsconfig.json)
 - del index.js(删除名叫index.js文件)
 - mkdir dist(创建一个名叫dist的文件夹)
 - tsc(运行)

1.include

指定编译文件默认是编译当前目录下所有的 ts 文件

- 这个是在中括号中填入路径，路径指向的那个ts文件会被编译出一个js文件出来。这个我们就可以用来编译指定文件

2.exclude

指定排除的文件

- 跟 include 反过来了，除了写入的路径之外，其他全部编译

3.target

指定编译 js 的版本例如 es5 es6

- 有些低配置的浏览器是不兼容es6的，这个时候我们就可以将其编译成es5使其适配

4.allowJS

是否允许编译 js 文件

- 是否允许TypeScript帮你编译js文件，默认是不允许的

5.removeComments

是否在编译过程中删除文件中的注释

6.rootDir

编译文件的目录

7.outDir

- 改变输出的目录，也就是编译后输出到这里设置的文件夹目录中

8.sourceMap

代码源文件

- 这个文件会打包压缩成一行，sourceMap会记录行数，到时候会比较好找

9.strict

严格模式

严格模式的限制

- 严格模式主要有以下限制：
 - 变量必须声明后再使用
 - 函数的参数不能有同名属性，否则报错
 - 不能使用 with 语句
 - 不能对只读属性赋值，否则报错
 - 不能使用前缀 0 表示八进制数，否则报错
 - 不能删除不可删除的属性，否则报错
 - 不能删除变量 delete prop，会报错，只能删除属性 delete global [prop]
 - eval 不会在它的外层作用域引入变量
 - eval 和 arguments 不能被重新赋值
 - arguments 不会自动反映函数参数的变化
 - 不能使用 arguments.callee
 - 不能使用 arguments.caller
 - 禁止 this 指向全局对象
 - 不能使用 fn.caller 和 fn.arguments 获取函数调用的堆栈
 - 增加了保留字（比如 protected、static 和 interface）

要注意 this 的限制。ES6 模块之中，顶层的 this 指向 undefined，即不应该在顶层代码使用 this。

10.module

默认 common.js 可选 es6 模式 amd umd 等

namespace命名空间(TS -- 16)

我们在工作中无法避免 全局变量 造成的污染，TypeScript 提供了 namespace 避免这个问题出现

- 内部模块，主要用于组织代码，避免命名冲突。
- 命名空间内的类默认私有
- 通过 `export` 暴露
- 通过 `namespace` 关键字定义

TypeScript与 ECMAScript 2015 一样，任何包含顶级 `import` 或者 `export` 的文件都被当成一个模块。相反地，如果一个文件不带有顶级的 `import` 或者 `export` 声明，那么它的内容被视为全局可见的（因此对模块也是可见的）

命名空间 中通过 `export` 将想要暴露的部分导出

如果不用 export 导出是无法读取其值的

```
namespace a {
  export const Time: number = 1000
  export const fn = <T>(arg: T): T => {
    return arg
  }
  fn(Time)
}

namespace b {
  export const Time: number = 1000
  export const fn = <T>(arg: T): T => {
    return arg
  }
  fn(Time)
}

a.Time
b.Time
```

案例

```
//文件1与文件2位于同一文件夹下
//文件1 => index.ts
const aa = 23
//文件3 => index2.ts
const aa = 66//此时就会报错，因为我们在文件1已经声明过aa了
```

```
//文件1 => index.ts
namespace A{
  export const aa =23
}
//文件2 => index.ts
namespace B{
  export const aa =66
}//不会报错

//要读取的话怎么做呢？此时在文件1
console.log(A.a);//像使用对象一样。实际上也确实包了一层对象
```

实际编译成js文件的样子

```
"use strict"
var A;
(function (A){
  A.a = 1;
})(A || (A = {}));
console.log(A,a);
//到此js文件所在目录下使用node index.js运行，输出1
```

嵌套命名空间

就是提取内容的时候需要多.几次。比如下方，原本只需要a.b，现在需要a.b.value

编译成js文件的话，就是在function又套了一层

```
namespace a {
  export namespace b {
    export class Vue {
      parameters: string
      constructor(parameters: string) {
        this.parameters = parameters
      }
    }
  }
}

let v = a.b.Vue

new v('1')
```

抽离命名空间

将命名空间的内容抽离出来，通过import引入到其他文件中使用

```
//在index2.ts文件下
export namespace B{
  export const a = 2
}

//在index.ts文件下
import xx from './index2.ts'

namespace A{
  export namespace C{
    export const D = 5
  }
}

console.log(A.C.D,B)//将B抽离成了文件
//将此文件用dsc进行终端编译，然后在tsconfig.json将module修改为CommonJS(node.js不认识
defined, node.js是基于CommonJS的)，进去js文件夹、终端运行node index
```

简化命名空间

可以给命名空间路径起个名字，然后直接使用这个名字就可以代替命名空间路径了

这个是不能够在ts-node的环境下去使用的

```
//在index2.ts文件下
export namespace B{
  export const a = 2
}

//在index.ts文件下
import xx from './index2.ts'

namespace A{
  export namespace C{
```



```

        export const D = 5
    }
}
console.log(A.C.D,B)//将B抽离成了文件

import AAA = A.C.D
console.log(AAA)//起到跟A.C.D一样的作用
import AAA = A.C
console.log(AAA.D)//起到跟A.C.D一样的作用

```

命名空间的合并

如果命名空间的命名一样的话(重名)，会自动合并

```

//案例1
namespace A{
    export const b = 2
}

namespace A{
    export const d = 3
}
//案例2
namespace A{
    export const b = 2
    export const d = 3//案例1跟案例2是一模一样的，会自动合并
}

```

三斜线指令(TS -- 17)

三斜线指令是包含单个 XML 标签的单行注释。注释的内容会做为编译器指令使用。

三斜线指令仅可放在包含它的文件的最顶端。一个三斜线指令的前面只能出现单行或多行注释，这包括其它的三斜线指令。如果它们出现在一个语句或声明之后，那么它们会被当做普通的单行注释，并且不具有特殊的涵义。

`/// <reference path="..." />` 指令是三斜线指令中最常见的一种。它用于声明文件间的依赖。

三斜线引用告诉 编译器 在编译过程中要引入的额外的文件。也可以认为是另一个import

你也可以把它理解成 `import`，它可以告诉编译器在编译过程中要引入的额外的文件

相较于抽离命名空间。范围更广，将整个文件都抽离出来了。

```

//在a.ts
namespace A {
    export const fn = () => 'a'
}
//在b.ts
namespace A {
    export const fn2 = () => 'b'
}
//在index.ts

```

```
///
```

这个时候小满将outFile打开了，这个的作用是：将输出文件合并为一个文件，并编译到指定的路径中

这个时候样式的还在

这个时候，我们再打开removeComments，他的作用是：删除编译后的所有的注释

编译好后，那些样式就不在了

引入声明文件

例如，把 `/// 引入到声明文件，表明这个文件使用了 @types/node/index.d.ts 里面声明的名字；并且，这个包需要在编译阶段与声明文件一起被包含进来。`

仅当在你需要写一个 `d.ts` 文件时才使用这个指令。

使用的话，需要将声明文件装起来(`npm install @types/node -D`)

```
///
```

声明文件d.ts(TS -- 18)

声明文件 declare

当使用 第三方库 时，我们需要引用它的声明文件，才能获得对应的代码补全、接口提示等功能。

```
declare var      声明全局变量
declare function 声明全局方法
declare class    声明全局类
declare enum     声明全局枚举类型
declare namespace 声明（含有子属性的）全局对象
interface 和 type 声明全局类型
///
```

先：npm init -y

npm init -y 在文件夹下生成默认的 package.json 文件，使用命令生成的 package.json 文件内容与不使用命令生产的不一樣

后：npm install express -S 正式线安装

npm install axios

- 从node_modules的axios的package.json可以发现types:"index.d.ts"，可以发现声明文件已经被指定了
 - 我们在去看index.d.ts文件可以看到最后通过declare将其导出了
- 在引入使用的时候，发现express在引入的时候爆红了

- 同样的在node_modules可以找到express中的package.json，发现他根本没有types，也就是说没有指定声明文件。所以才会有爆红的这个问题
- 我们可以在根目录下创建一个express.d.ts文件
- 在文件中写入 `declare var express: () => any`
- 这个时候，我们在使用express的时候，就发现可以使用了 `express()`，不会爆红了
- 另一种方法就是按照提示去装包，然后去tsconfig.json去导出一下 `"allowSyntheticDefaultImports": true` 进行补全

[npm收录包大全](#)

Mixins混入(TS -- 19)

TypeScript 混入 Mixins 其实 vue 也有 mixins 这个东西 你可以把他看作为合并

对象混入

可以使用 ES6 的 `Object.assign` 合并多个对象

此时 `people` 会被推断成一个交差类型 `Name & Age & sex`;

Object.assign()

- `Object.assign()` 这个方法来实现浅复制
- 主要的用途是用来合并多个 JavaScript 的对象
- `Object.assign()` 接口可以接收多个参数，第一个参数是目标对象，后面的都是源对象，`assign` 方法将多个原对象的属性和方法都合并到了目标对象上面，如果在这个过程中出现同名的属性（方法），后合并的属性（方法）会覆盖之前的同名属性（方法）
- `Object.assign` 拷贝的属性是有限制的，只会拷贝对象本身的属性（不会拷贝继承属性），也不会拷贝不可枚举的属性
- `Object.assign` 不会跳过那些值为 `[null]` 或 `[undefined]` 的源对象

```
interface Name {
  name: string
}
interface Age {
  age: number
}
interface Sex {
  sex: number
}

let people1: Name = { name: "小满" }
let people2: Age = { age: 20 }
let people3: Sex = { sex: 1 }

//Object.assign(a,b,c)

const people = Object.assign(people1,people2,people3)
```

类的混入

首先声明两个 mixins 类（严格模式要关闭不然编译不过）

```
//混入类
class A{
  type:boolean
  changeType(){
    this.type = !this.type
  }
}
```

```
class B{
  name:string
  getName(){
    return this.name
  }
}
```

//实现类 首先应该注意到的是，没使用 **extends** 而是使用 **implements**。把类当成了接口。我们可以这么做来达到目的，为将要 **mixin** 进来的属性方法创建出占位属性。这告诉编译器这些成员在运行时是可用的。这样就能使用 **mixin** 带来的便利，虽说需要提前定义一些占位属性

```
class C implements A,B{
  //这个时候编辑器会给出提示类"C"错误实现"A"。你是想扩展"A"并将其成员作为子继承吗？
  //类型"C"缺少类型"A"中的以下属性:type,changeType
  //B类同理
  //这个时候就需要我们提前定义占位符
  type:boolean = false;
  name:string = "小余";
  changeType():=> void
  getName(): => string
}
```

mixins(C,[A,B])//第一个为目标对象，后面为要混入的对象

//最后，创建这个帮助函数，帮我们做混入操作。它会遍历 **mixins** 上的所有属性，并复制到目标上去，把之前的占位属性替换成真正的实现代码

//帮助函数，把我们在实现类中写的去进行一个实现

```
function mixins (curCls:any,itemCls:any[]){
  itemCls.forEach(itemC=>{
    console.log(itemC);//输出[class A][class B]，我们要读取的不是这个，而是他原型上的一些属性
    Object.getOwnPropertyNames(itemC.prototype).forEach(name =>{
      //Object.getOwnPropertyNames () 可以获取对象自身的属性，除去他继承来的属性，对它所有的属性遍历，它是一个数组，遍历一下它所有的属性名
      console.log(name);//打印出来了changeType跟getName
      curCls.prototype[name] = itemC.prototype[name]
    })
  })
}
```

```
let ccc = new C()//实例化一下
console.log(ccc.type);//false
ccc.changeType()//这里切换了布尔值
console.log(ccc.type);//true
```

装饰器Decorator(TS -- 20)

Decorator [装饰器](#)是一项实验性特性，在未来的版本中可能会发生改变

它们不仅增加了代码的可读性，清晰地表达了意图，而且提供一种方便的手段，增加或修改类的功能

若要启用实验性的装饰器特性，你必须在 命令行 或 `tsconfig.json` 里启用 编译器 选项

启用的名字叫 `experimentalDecorators`

装饰器

装饰器是一种特殊类型的声明，它能够被附加到类声明、方法、访问符、属性或者参数上
通过 `@` 语法糖实现

使用方法：对于这个的实现我认为就是定义好了之后直接盖在你想对其使用目标的头顶上

- 以下两个代码块的watcher都是不支持传参的

```
const watcher:ClassDecorator = (target:Function)=>{
  console.log(target)
}
@watcher//通过@去使用，会回传一个构造函数，也就是target
class A{

}
//通过ts-node xxx打印出来的结果为[class A]
```

知识点复习：

prototype 对象是实现面向对象的一个重要机制。每个函数也是一个对象，它们对应的类就是 function，每个函数对象都具有一个子对象 prototype。

Prototype 表示了该函数的原型，prototype 表示了一个类的属性的集合。当通过 new 来生成一个类的对象时，prototype 对象的属性就会成为实例化对象的属性。

[Prototype与prototype的区别](#)

```
const watcher:ClassDecorator = (target:Function)=>{
  target.prototype.getName = <T>(name:T):T =>{
    return name
  }
}
@watcher
class A{

}
let a = new A()

a.getName()//会报类型"A"上不存在属性"getName"
//(<any>a).getName()//我们将其断言成any类型
console.log((<any>a).getName("小满深夜骑单车"))//对其进行使用，输出 小满深夜骑单车

@watcher
class B{

}
let b = new B()
console.log(b.getname('666'))//也是可以的
```

装饰器工厂

- 支持传参的写法

我认为其实就是多了一层壳，这层壳用来接收@watcher的参数。里面return的那一层再用来接收class A这个构造器

其实也就是一个高阶函数 外层的函数接受值 里层的函数最终接受类的构造函数

```
const watcher = (name:string):ClassDecorator =>{
  return (target:Function) =>{
    target.prototype.getNames = <T>(name:string):T =>{
      return name
    }
    target.prototype.getOptions = (): string => {
      return name
    }
  }
}
@watcher("小余的笔记")
class A{

}
let a = new A()
console.log((<any>a).getNames())//返回 小余的笔记
```

装饰器组合

就是可以使用多个装饰器

```
//装饰器组合组合
const watcher = (name:string):ClassDecorator =>{
  return (target:Function) =>{
    target.prototype.getName = ()=>{
      return name
    }
  }
}

const log:ClassDecorator = (target:Function) =>{
  target.prototype.a = 213
}

@log
@watcher('小满')
class A{

}
```

```
const watcher = (name: string): ClassDecorator => {
  return (target: Function) => {
    target.prototype.getParams = <T>(params: T): T => {
      return params
    }
  }
}
```

```

        target.prototype.getOptions = (): string => {
            return name
        }
    }
}
const watcher2 = (name: string): ClassDecorator => {
    return (target: Function) => {
        target.prototype.getNames = ():string => {
            return name
        }
    }
}

@watcher2('name2')
@watcher('name')
class A {
    constructor() {

    }
}

const a = new A();
console.log((a as any).getOptions());
console.log((a as any).getNames());

```

方法装饰器

返回三个参数

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 成员的属性描述符。

相当于抽离出来，在类中引用进去，这样可以在不同的类中引用

```

[
    {},
    'setParasm',
    {
        value: [Function: setParasm],
        writable: true,
        enumerable: false,
        configurable: true
    }
]

```

```

const met:MethodDecorator = (...args) => {
    console.log(args);
}

class A {
    constructor() {

```

```

    }
    @met
    getName ():string {
        return '小满'
    }
}

const a = new A();

```

属性装饰器

返回两个参数

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 属性的名字。

[{}, 'name', undefined]

```

const met:PropertyDecorator = (...args) => {
    console.log(args);
}

class A {
    @met
    name:string
    constructor() {

    }

}

const a = new A();

```

参数装饰器

返回三个参数

1. 对于静态成员来说是类的构造函数，对于实例成员是类的原型对象。
2. 成员的名字。
3. 参数在函数参数列表中的索引。

[{}, 'setParasm', 0]

```

const met:ParameterDecorator = (...args) => {
    console.log(args);
}

class A {
    constructor() {

    }

    setParasm (@met name:string = '213') {

    }

}

```



```
}
```

```
const a = new A();
```

Rollup构建TS项目(TS -- 21)

由于我使用的是vite打包工具，那这个视频是属于过一遍的程度，本章节内容将直接把小满在CSDN的文章copy过来

Rollup 构建 TS 项目

安装依赖

1. 全局安装 rollup `npm install rollup-g`
2. 安装 TypeScript `npm install typescript -D`
3. 安装 TypeScript 转换器 `npm install rollup-plugin-typescript2 -D`
4. 安装代码压缩插件 `npm install rollup-plugin-terser -D`
5. 安装 rollupweb 服务 `npm install rollup-plugin-serve -D`
6. 安装[热更新](#) `npm install rollup-plugin-livereload -D`
7. 引入外部依赖 `npm install rollup-plugin-node-resolve -D`
8. 安装配置环境变量用来区分本地和生产 `npm install cross-env -D`
9. 替换环境变量给浏览器使用 `npm install rollup-plugin-replace -D`

配置 json 文件

`npm init -y`

```
{
  "name": "rollupTs",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "cross-env NODE_ENV=development rollup -c -w",
    "build": "cross-env NODE_ENV=production rollup -c"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cross-env": "^7.0.3",
    "rollup-plugin-livereload": "^2.0.5",
    "rollup-plugin-node-resolve": "^5.2.0",
    "rollup-plugin-replace": "^2.2.0",
    "rollup-plugin-serve": "^1.1.0",
    "rollup-plugin-terser": "^7.0.2",
    "rollup-plugin-typescript2": "^0.31.1",
    "typescript": "^4.5.5"
  }
}
```

配置 rollup 文件

```
console.log(process.env);
import ts from 'rollup-plugin-typescript2'
import path from 'path'
import serve from 'rollup-plugin-serve'
import livereload from 'rollup-plugin-livereload'
import { terser } from 'rollup-plugin-terser'
import resolve from 'rollup-plugin-node-resolve'
import replace from 'rollup-plugin-replace'

const isDev = () => {
  return process.env.NODE_ENV === 'development'
}

export default {
  input: './src/main.ts',
  output: {
    file: path.resolve(__dirname, './lib/index.js'),
    format: 'umd',
    sourcemap: true
  },

  plugins: [
    ts(),
    terser({
      compress: {
        drop_console: !isDev()
      }
    }),
    replace({
      'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV)
    }),
    resolve(['.js', '.ts']),
    isDev() && livereload(),
    isDev() && serve({
      open: true,
      openPage: '/public/index.html'
    })
  ]
}
```

配置 tsconfig.json

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig.json to read more about this file */

    /* Projects */
    // "incremental": true,                          /* Enable incremental
    compilation */
    // "composite": true,                             /* Enable constraints
    that allow a TypeScript project to be used with project references. */
    // "tsBuildInfoFile": "./",                       /* Specify the folder
    for .tsbuildinfo incremental compilation files. */
```

```

    // "disableSourceOfProjectReferenceRedirect": true, /* Disable preferring
source files instead of declaration files when referencing composite projects */
    // "disableSolutionSearching": true,                /* Opt a project out of
multi-project reference checking when editing. */
    // "disableReferencedProjectLoad": true,           /* Reduce the number of
projects loaded automatically by TypeScript. */

    /* Language and Environment */
    "target": "es5",                                  /* Set the JavaScript
language version for emitted JavaScript and include compatible library
declarations. */
    // "lib": [],                                     /* Specify a set of
bundled library declaration files that describe the target runtime environment.
*/
    // "jsx": "preserve",                             /* Specify what JSX code
is generated. */
    // "experimentalDecorators": true,                /* Enable experimental
support for TC39 stage 2 draft decorators. */
    // "emitDecoratorMetadata": true,                 /* Emit design-type
metadata for decorated declarations in source files. */
    // "jsxFactory": "",                              /* Specify the JSX
factory function used when targeting React JSX emit, e.g. 'React.createElement'
or 'h' */
    // "jsxFragmentFactory": "",                      /* Specify the JSX
Fragment reference used for fragments when targeting React JSX emit e.g.
'React.Fragment' or 'Fragment'. */
    // "jsxImportSource": "",                         /* Specify module
specifier used to import the JSX factory functions when using `jsx: react-jsx*`.`
*/
    // "reactNamespace": "",                          /* Specify the object
invoked for `createElement`. This only applies when targeting `react` JSX emit.
*/
    // "noLib": true,                                 /* Disable including any
library files, including the default lib.d.ts. */
    // "useDefineForClassFields": true,                /* Emit ECMAScript-
standard-compliant class fields. */

    /* Modules */
    "module": "ES2015",                              /* Specify what module
code is generated. */
    // "rootDir": "./",                               /* Specify the root
folder within your source files. */
    // "moduleResolution": "node",                     /* Specify how
TypeScript looks up a file from a given module specifier. */
    // "baseUrl": "./",                               /* Specify the base
directory to resolve non-relative module names. */
    // "paths": {},                                   /* Specify a set of
entries that re-map imports to additional lookup locations. */
    // "rootDirs": [],                                 /* Allow multiple
folders to be treated as one when resolving modules. */
    // "typeRoots": [],                               /* Specify multiple
folders that act like `./node_modules/@types`. */
    // "types": [],                                   /* Specify type package
names to be included without being referenced in a source file. */

```

```

    // "allowUmdGlobalAccess": true,                /* Allow accessing UMD
globals from modules. */
    // "resolveJsonModule": true,                  /* Enable importing
.json files */
    // "noResolve": true,                          /* Disallow `import`s,
`require`s or `<reference>s` from expanding the number of files TypeScript should
add to a project. */

    /* JavaScript Support */
    // "allowJs": true,                            /* Allow JavaScript
files to be a part of your program. Use the `checkJs` option to get errors from
these files. */
    // "checkJs": true,                            /* Enable error
reporting in type-checked JavaScript files. */
    // "maxNodeModuleJsDepth": 1,                  /* Specify the maximum
folder depth used for checking JavaScript files from `node_modules`. Only
applicable with `allowJs`. */

    /* Emit */
    // "declaration": true,                        /* Generate .d.ts files
from TypeScript and JavaScript files in your project. */
    // "declarationMap": true,                     /* Create sourcemaps for
d.ts files. */
    // "emitDeclarationOnly": true,                 /* Only output d.ts
files and not JavaScript files. */
    "sourceMap": true,                             /* Create source map
files for emitted JavaScript files. */
    // "outFile": "./",                           /* Specify a file that
bundles all outputs into one JavaScript file. If `declaration` is true, also
designates a file that bundles all .d.ts output. */
    // "outDir": "./",                             /* Specify an output
folder for all emitted files. */
    // "removeComments": true,                     /* Disable emitting
comments. */
    // "noEmit": true,                             /* Disable emitting
files from a compilation. */
    // "importHelpers": true,                       /* Allow importing
helper functions from tslib once per project, instead of including them per-file.
*/
    // "importsNotUsedAsValues": "remove",          /* Specify emit/checking
behavior for imports that are only used for types */
    // "downlevelIteration": true,                  /* Emit more compliant,
but verbose and less performant JavaScript for iteration. */
    // "sourceRoot": "",                           /* Specify the root path
for debuggers to find the reference source code. */
    // "mapRoot": "",                              /* Specify the location
where debugger should locate map files instead of generated locations. */
    // "inlineSourceMap": true,                     /* Include sourcemap
files inside the emitted JavaScript. */
    // "inlineSources": true,                       /* Include source code
in the sourcemaps inside the emitted JavaScript. */
    // "emitBOM": true,                            /* Emit a UTF-8 Byte
Order Mark (BOM) in the beginning of output files. */
    // "newline": "crlf",                          /* Set the newline
character for emitting files. */

```

```

    // "stripInternal": true,                /* Disable emitting
declarations that have `@internal` in their JSDoc comments. */
    // "noEmitHelpers": true,                /* Disable generating
custom helper functions like `__extends` in compiled output. */
    // "noEmitOnError": true,                /* Disable emitting
files if any type checking errors are reported. */
    // "preserveConstEnums": true,           /* Disable erasing
`const enum` declarations in generated code. */
    // "declarationDir": ".",                /* Specify the output
directory for generated declaration files. */
    // "preserveValueImports": true,         /* Preserve unused
imported values in the JavaScript output that would otherwise be removed. */

    /* Interop Constraints */
    // "isolatedModules": true,               /* Ensure that each file
can be safely transpiled without relying on other imports. */
    // "allowsSyntheticDefaultImports": true, /* Allow 'import x from
y' when a module doesn't have a default export. */
    "esModuleInterop": true,                 /* Emit additional
JavaScript to ease support for importing CommonJS modules. This enables
`allowSyntheticDefaultImports` for type compatibility. */
    // "preservesSymlinks": true,            /* Disable resolving
symlinks to their realpath. This correlates to the same flag in node. */
    "forceConsistentCasingInFileNames": true, /* Ensure that casing
is correct in imports. */

    /* Type Checking */
    "strict": true,                          /* Enable all strict
type-checking options. */
    // "noImplicitAny": true,                /* Enable error
reporting for expressions and declarations with an implied `any` type.. */
    // "strictNullChecks": true,             /* When type checking,
take into account `null` and `undefined`. */
    // "strictFunctionTypes": true,           /* When assigning
functions, check to ensure parameters and the return values are subtype-
compatible. */
    // "strictBindCallApply": true,          /* Check that the
arguments for `bind`, `call`, and `apply` methods match the original function.
*/
    // "strictPropertyInitialization": true,  /* Check for class
properties that are declared but not set in the constructor. */
    // "noImplicitThis": true,               /* Enable error
reporting when `this` is given the type `any`. */
    // "useUnknownInCatchVariables": true,    /* Type catch clause
variables as 'unknown' instead of 'any'. */
    // "alwaysStrict": true,                 /* Ensure 'use strict'
is always emitted. */
    // "noUnusedLocals": true,               /* Enable error
reporting when a local variables aren't read. */
    // "noUnusedParameters": true,           /* Raise an error when a
function parameter isn't read */
    // "exactOptionalPropertyTypes": true,    /* Interpret optional
property types as written, rather than adding 'undefined'. */
    // "noImplicitReturns": true,            /* Enable error
reporting for codepaths that do not explicitly return in a function. */

```

```

    // "noFallthroughCasesInSwitch": true,          /* Enable error
reporting for fallthrough cases in switch statements. */
    // "noUncheckedIndexedAccess": true,           /* Include 'undefined'
in index signature results */
    // "noImplicitOverride": true,                 /* Ensure overriding
members in derived classes are marked with an override modifier. */
    // "noPropertyAccessFromIndexSignature": true, /* Enforces using
indexed accessors for keys declared using an indexed type */
    // "allowUnusedLabels": true,                  /* Disable error
reporting for unused labels. */
    // "allowUnreachableCode": true,               /* Disable error
reporting for unreachable code. */

    /* Completeness */
    // "skipDefaultLibCheck": true,                /* Skip type checking
.d.ts files that are included with TypeScript. */
    "skipLibCheck": true                          /* Skip type checking
all .d.ts files. */
  }
}

```

npm run dev 启动就可以尽情的玩耍了

webpack 构建 TS 项目

安装依赖

安装 webpack npm install webpack -D

webpack4 以上需要 npm install webpack-cli -D

编译 TS npm install ts-loader -D

TS 环境 npm install [typescript](#) -D

热更新服务 npm install webpack-dev-server -D

HTML 模板 npm install html-webpack-plugin -D

配置文件

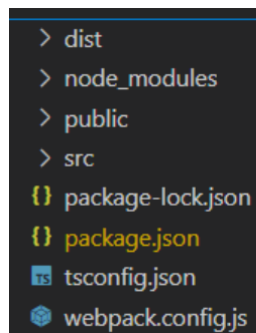
```

const path = require('path')
const htmlWebpackPlugin = require('html-webpack-plugin')
module.exports = {
  entry: './src/index.ts',
  mode: "development",
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: "index.js"
  },
  stats: "none",
  resolve: {
    extensions: ['.ts', '.js'],
    alias: {
      '@': path.resolve(__dirname, './src')
    }
  },
}

```

```
module: {
  rules: [
    {
      test: /\.ts$/,
      use: "ts-loader"
    }
  ]
},
devServer: {
  port: 1988,
  proxy: {}
},
plugins: [
  new HtmlWebpackPlugin({
    template: "./public/index.html"
  })
]
}
```

目录结构



```
> dist
> node_modules
> public
> src
{} package-lock.json
{} package.json
tsconfig.json
webpack.config.js
```

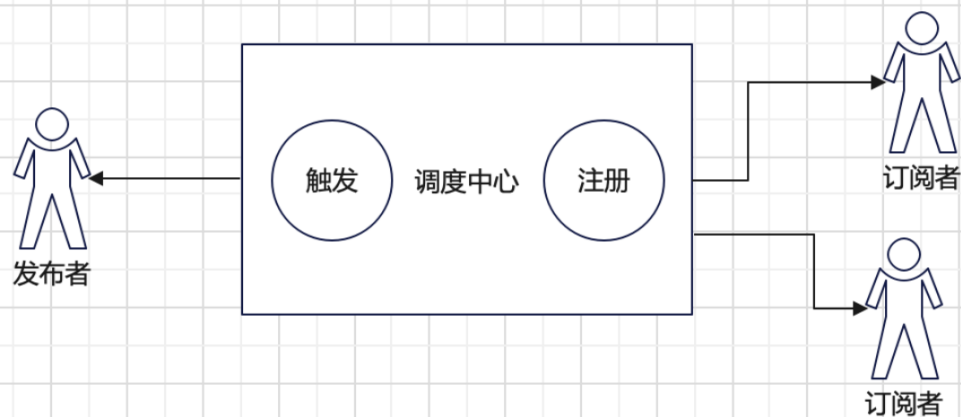
实战TS编写发布订阅模式(TS -- 22)

什么是发布订阅模式，其实小伙伴已经用到了发布订阅模式例如 `addEventListener`，`Vue` `evnetBus`

都属于发布订阅模式

简单来说就是 你(小满)要和 大满 二满 三满打球，大满带球，二满带水，三满带球衣。全都准备完成后开始打球。

思维导图



```

interface Evenet{
  on(name:string,fn:Function)) => void,
  emit(name:string,..args:Array<any>) => void, //派发
  off(name:string,fn:Function)=> void, //移除
  once(name:string,fn:Function) => void //只执行一次
}

interface List{
  [key:string]:Array<Function>
}

class Dispatch implements Event{//通过implements来约束这个类(Event)
  list:List
  constructor(){
    this.list = {}
  }
  on(name:string,fn:Function){
    const callback = this.list[name] || [] //如果有取到值的话那就是一个数组，没有取到值的话就是一个空数组
    callback.push(fn) //因为不管怎么说，callback都是数组，所以我们后面的数组也可以直接添加上去
    this.list[name] = callback
    console.log(this.list);
  }
  emit(name:string,..args:Array<any>){
    let evnetName = this.list[name]
    //on监听跟emit派发的时候 name 是需要一样的，不然会出错，所以我们这里要进行一个判断
    if(evnetName){
      eventName.forEach(fn=>{
        //内容从下面的o.emit()传送上来
        fn.apply(this,args) //第一个参数this指向，第二个参数为数组，这里也刚好是一个数组就直接传进去。会将on监听的数据直接打印出来，这里打印出来66 99
      })
    }else{
      console.error(`名称错误${name}`)
    }
  }
  off(name:string,fn:Function){
    //off是删除一个函数，所以我们在下面将创建一个fn函数让他来删一下
    let eventName = this.list[name]
  }
}

```



```

//跟emit一样的，需要进行判断有没有值，还有就是函数存不存在，不存在的话就没得删了吧
if(eventName && fn){
    //我们要通过索引来将其删掉
    let index = eventName.findIndex(fns=> fns === fn )
    eventName.splice(index,1)
    console.log(eventName)
}else{
    console.error(`名称错误${name}`)
}
}
once(name:string,fn:Function){
    let de = (...args:Array<any>) =>{
        fn.apply(this,args)//指向到那个只调用一次函数的那里
        this.off(name,de)//调用完就把它删掉，这就是只能调用一次的原因哈哈
    }
    this.on(name,de)//第一个还是名字，第二个临时函数
}
}

const o = new Dispatch()//初始化

o.on('post',()=>{//post作为key
    console.log(66);
})//第一个参数是事件名称，第二个是回调函数

o.on('post',(...args:Array<any>)=>{
    console.log(99,args)
    //这里我们对第二个回调函数传入了...args，也就是收到了o.emit除了第一个参数后面那些乱七八糟的东西(因为我们设定了any，对接收的类型并没有限制，所以收到什么乱七八糟的东西都不奇怪)，并在控制台打印了出来
})

const fn = (...args:Array<any>) => {
    console.log(args,2)
}
o.on('post',fn)//没错，这个就是特地创建出来删掉的
o.off('post',fn)//将fn删掉
//o.on('post2',()=>{
//    //都会在控制台显示出来
//})
o.once('post',(...args:Array<any>)=>{
    console.log(args,'once')
})

o.emit('post',1,false,{name:"小满"})//除了第一个参数一样是事件，后面参数是不限制个数的，而且传什么都行
o.emit('post',2,false,{name:"小满"})//这里如果收到就是有问题，因为我们在上面使用once了，只调用一次

```

上面在off删除中使用到的splice知识点补充

```
splice(index,len,[item])
```

它也可以用来替换 / 删除 / 添加数组内某一个或者几个值（该方法会改变原始数组）

index: 数组开始下标
len: 替换 / 删除的长度
item: 替换的值, 删除操作的话 **item** 为空

删除:

```
// 删除起始下标为 1, 长度为 1 的一个值 (len 设置 1, 如果为 0, 则数组不变)
var arr = ['a','b','c','d'];
arr.splice(1,1);
console.log(arr);    //[ 'a','c','d' ];
// 删除起始下标为 1, 长度为 2 的一个值 (len 设置 2)
var arr2 = ['a','b','c','d'] arr2.splice(1,2);
console.log(arr2);   //[ 'a','d' ]
```

替换:

```
// 替换起始下标为 1, 长度为 1 的一个值为'ttt', len 设置的 1
var arr = ['a','b','c','d'];
arr.splice(1,1,'ttt');
console.log(arr);      //[ 'a','ttt','c','d' ]
// 替换起始下标为 1, 长度为 2 的两个值为'ttt', len 设置的 1
var arr2 = ['a','b','c','d'];
arr2.splice(1,2,'ttt');
console.log(arr2);     //[ 'a','ttt','d' ]
```

添加:

```
// 在下标为 1 处添加一项 'ttt'
var arr = ['a','b','c','d'];
arr.splice(1,0,'ttt');
console.log(arr);      //[ 'a','ttt','b','c','d' ]
```

TS 进阶用法 proxy & Reflect(TS -- 23)

proxy: 对象代理(是ES6新增的对象拦截器, 能够监听到一个对象的变化)

Reflect: 配合proxy来操作对象

Proxy

Proxy 对象用于创建一个对象的代理, 从而实现基本操作的拦截和自定义 (如属性查找、赋值、枚举、函数调用等)

target

要使用 **Proxy** 包装的目标对象 (可以是任何类型的对象, 包括原生数组, 函数, 甚至另一个代理)。

handler

一个通常以函数作为属性的对象, 各属性中的函数分别定义了在执行各种操作时代理 **p** 的行为。

handler.get() 本次使用的 **get**

属性读取操作的捕捉器。

handler.set() 本次使用的 **set**

属性设置操作的捕捉器。

Reflect

与大多数全局对象不同 `Reflect` 并非一个构造函数，所以不能通过 [new 运算符](#) 对其进行调用，或者将 `Reflect` 对象作为一个函数来调用。`Reflect` 的所有属性和方法都是静态的（就像 [Math](#) 对象）

Reflect.get(target, name, receiver)

`Reflect.get` 方法查找并返回 `target` 对象的 `name` 属性，如果没有该属性返回 `undefined`

Reflect.set(target, name,value, receiver)

`Reflect.set` 方法设置 `target` 对象的 `name` 属性等于 `value`。

```
type Person{
  name:string,
  age:number,
  text:string
}

const proxy = (object:any,key:any)=>{//我们要自己实现proxy啦
  return new Proxy(object,{
    get(target,prop,receiver){
      console.log('=====>get',prop);
      //prop就是一个key，target就是地下那个man的对象，receiver是跟target一样的值，
      防止上下文错误的
      return Reflect.get(target,prop,receiver)//这里刚好对应的也是这三个参数
    }
    set(target,prop,value,receiver){//多了一个value，因为我们要设置值
      //日志
      console.log('=====>set',prop);
      return Reflect.set(target,prop,value,receiver)
    }
  })
}

//日志监听函数
//由于我们要监听man里面的内容，所以这里可以使用联合类型
const logAccess = <T>(object: T ,key:"name" | "age" | "text"):T =>{//为了使其灵活度高一点，我们不使其object等于Person，而是为泛型T，使用的时候再去设置
  return proxy(object,key)
}

let man:Person = ({
  name:"小满"
  age:22
  text:"三秒真男人"
},'name')

let man2 = logAccess({
  name:"小余"
},'name')
```

```
man.age = 30//走set
man.age//走get
console.log(man)
```

泛型优化

`const logAccess = <T>(object: T ,key: keyof T):T =>{` //为了使其灵活度高一点，我们不使其 `object` 等于 `Person`，而是为泛型 `T`，使用的时候再去设置。`key` 也不固定死，而是使用 `keyof`，将我们传入的对象推断为联合类型

```
    return proxy(object,key)
}
```

```
let man2 = logAccess({
  name:"小余"
  id:925
},'id')//就可以动态的去约束类型
```

```
let man2 = logAccess({
  name:"小余"
  id:925
},'id2')//报错，因为我们类型里没有id2
```

优化完整版

```
type Person = {
  name: string,
  age: number,
  text: string
}
```

```
const proxy = (object: any, key: any) => {
  return new Proxy(object, {
    get(target, prop, receiver) {
      console.log(`get key=====>${key}`);
      return Reflect.get(target, prop, receiver)
    },

    set(target, prop, value, receiver) {
      console.log(`set key=====>${key}`);

      return Reflect.set(target, prop, value, receiver)
    }
  })
}
```

```
const logAccess = <T>(object: T, key: keyof T): T => {
  return proxy(object, key)
}
```

```
let man: Person = logAccess({
  name: "小满",
  age: 20,
```

```
    text: "我的很小"
  }, 'age')

let man2 = logAccess({
  id:1,
  name:"小满2"
}, 'name')

man.age = 30

console.log(man);
```

TS 进阶用法 Partial & Pick(TS -- 24)

TypeScript内置高级类型Partial Pick

Partial

源码

```
/**
 * Make all properties in T optional
 * 将T中的所有属性设置为可选
 */
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```

手写实现

```
type Person{
  name:string,
  age:number,
  text:string
}
```

//keyof: 将一个接口对象的全部属性取出来变成联合类型

//keyof的作用就是把我们的属性变成联合类型，在底下就相当于"\"name\"|\"age\"|\"text\"\"。而 in 就是为遍历这个联合类型的每一项，然后放到这个P里(所以P里就是name、age、text)，然后使其变成`?`可选的

type Par<T> = { //这个T就是我们传过来的Person，所以T[P]就是Person里面name、age、text的内容(string那些啥的)

//小满对T[P]的形容方式：通过索引取值的方式

```
    [P in keyof T]?:T[P] //所以你看这个肯定能看懂
};
```

type p = Partial<Person> //这个时候，我们会发现p上面的属性，name、age、text都变成可选的了

使用前(范例)

```
type Person = {
  name:string,
  age:number
}

type p = Partial<Person>
```

使用后(范例)

```
type p = {
  name?: string | undefined;
  age?: number | undefined;
}
```

Pick

从类型定义 T 的属性中，选取指定一组属性，返回一个新的类型定义。

源码

```
/**
 * From T, pick a set of properties whose keys are in the union K
 */
type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
};
```

手写实现

```
type Person = {
  name:string,
  age:number,
  text:string,
  address:string
}

type Ex = "text" | "age"

type A = Pick<Person,Ex>
```

分析(需要结合手写的内容看)

```
type Pick<T, K extends keyof T> = { //T跟K都是泛型,T相当于我们的Person, K就相当于我们传的联合类型, 然后同样也经历了keyof的洗礼, 使其变成联合类型, K通过extends被约束了这点, 使其只能为T, 也就是Person内的值
  [P in K]: T[P];
};

type p = Pick<Person, 'age' | 'name'> //这里的Person请参考手写实现的Person
```

TS 进阶用法 Record & Readonly(TS -- 25)

Readonly

和 Partial 很像是吧？只是将Partial替换成了 Readonly

源码

```
type Readonly<T> = {  
  readonly [P in keyof T]: T[P];  
};
```

手写实现

```
type Readonly<T> = {  
  readonly [P in keyof T]: T[P]; //keyof还是那样，转化为联合类型，in去遍历选项。T[P]通过索引取值的方式  
  //然后为里面每个内容都加上只读属性  
};  
  
type Person = {  
  name:string,  
  age:number,  
  text:string  
}  
  
type man = R<Person>
```

Record

- 1 keyof any 返回 string number symbol 的联合类型
 - 2 in 我们可以理解成 for in P 就是 key 遍历 keyof any 就是 string number symbol 类型的每一项
 - 3 extends 来约束我们的类型
 - 4 T 直接返回类型
- 做到了约束 对象的 key 同时约束了 value

源码

```
type Record<K extends keyof any, T> = {  
  [P in K]: T;  
};
```

手写实现

```
type Rec<K extends keyof any, T> = { //T是泛型，传什么在这里并没有限制  
  [P in K]: T;  
};  
  
//keyof返回联合类型  
type key = string | number | symbol  
  
type Person = {  
  name:string,
```

```

    age:number.
    text:string
  }

type K = "A"|"B"|"C"//因为我们在这里定义了K，所以let B才只能使用A、B、C，如果这里换成1、2、3，那底下也只能使用1、2、3而不是A、B、C

type B = Rec<K,Person>//这里会返回成type B = {A:Person;B:Person;C:Person;}的形式

let obj:B = {
  A:{name:"小满",age:3,text:"三秒真男人"}//这里值的类型需要是Person的类型，因为在type B中已经定义了
  B:{name:"小余",age:18,text:"三小时真男人"}
  C:{name:"狗洛",age:1,text:"零点三秒真男人"}
}

```

TS 进阶用法 infer(TS -- 26)

infer 是 TypeScript新增到的关键字 充当占位符

我们来实现一个条件类型推断的例子

定义一个类型 如果是数组类型 就返回 数组元素的类型 否则 就传入什么类型 就返回什么类型

```

type TYPE<T> = T extends Array<any> ? T[number] : T

type A = TYPE<string[]>//会返回type A = string
type B = TYPE<(string|number)[]>//会返回type B = string|number
type C = TYPE<boolean>//返回type C = boolean

```

infer

使用 infer 修改

```

type TYPE<T> = T extends Array<infer U> ? U : T//U不是泛型，而是充当占位符使用，读取Array类型然后进行返回
type TYPE<T> = T extends Array<infer U> ? U : never//限制只能传type T这个元组类型，其他都不能传

type A = TYPE<string[]>//会返回type A = string
type B = TYPE<(string|number)[]>//会返回type B = string|number

type T = [string,number]
//使其变成联合类型(小技巧)
type uni = TYPE<T>//返回联合类型type uni = string|number
type uni = TYPE<T>//返回type uni = never。因为我们进行了限制

```

infer 类型提取(TS -- 27)

提取头部元素

`T extends any[]`：对T进行泛型约束，一个any类型的数组

`type First<T extends any[]> = T extends [infer one,infer two,infer three]? one: []`：对T进行泛型约束为数组类型，用infer提取，提取的变量名对应着Arr的a、b、c。然后决定返回one，也就是第一个元素还是空数组

```
type Arr = ['a','b','c']

type First<T extends any[]> = T extends [infer one,infer two,infer three]? one: []
//ES6进阶版
type First<T extends any[]> = T extends [infer First,...any[]] ? First : []//1
...

type a = First<Arr>
```

提取尾部元素

将头尾反过来了

```
type Arr = ['a', 'b', 'c']

type Last<T extends any[]> = T extends [...any[], infer Last] ? Last : []//... 尾部
type c = Last<Arr>
```

剔除第一个元素 Shift

将除了第一个之外的其他通过ES6语法提取出来

```
type Arr = ['a','b','c']

type First<T extends any[]> = T extends [unknown,...infer Rest] ? Rest : []

type a = First<Arr>
```

剔除尾部元素 pop

将除了第最后一个之外的其他通过ES6语法提取出来

```
type Arr = ['a','b','c']

type First<T extends any[]> = T extends [...infer Rest,unknown] ? Rest : []

type a = First<Arr>
```

infer 递归(TS -- 28)

需求：

有这么一个类型

```
type Arr = [1, 2, 3, 4]
```

希望通过一个 ts 工具变成

```
type Arr = [4, 3, 2, 1]
```

具体思路 首先使用泛型约束 约束只能传入数组类型的东西 然后从数组中提取第一个，放入新数组的末尾，反复此操作，形成递归 满足结束条件返回该类型

```
type Arr = [1, 2, 3, 4]
//先来一个泛型约束(对T)，这是通过不断递归拿到最后一个元素(也就是问好后面的那个First)填到最前面，直到没有元素为止(三元表达式)
//
type ReverArr<T extends any[]> = T extends [infer First, ...infer rest] ?
[...ReverArr<rest>, First] : T

type Arrb = ReverArr<Arr>
```