认识TypeScript

是什么

TypeScript 是具有类型语法的 JavaScript，是一门强类型的编程语言


```
// JavaScript是弱类型语言
// 变量可以赋不同类型的值
let count = 100
count = '100'
```

```
// TypeScript是强类型语言
// 变量不能做随意类型赋值
let count: number = 100
count = '100'
```

带来的好处（1）

静态类型检查，提前发现代码错误

```
function arrToStr (arr) {
  return arr.join('')
}
arrToStr('123')
```

JavaScript在 运行时 才能发现错误

```
Uncaught TypeError: arr.join is not a function
    at arrToStr (test.html:16:18)
    at test.html:18:5
```

TypeScript 写代码时 就能发现错误

```
function arrToStr(arr: Array<string>) {
  return arr.join('')
}

arrToStr('123')     类型“string”的参数不能赋给类型“string[]”的参数。
```

带来的好处（2）

良好的代码提示，提升开发效率

```
<template>
  <ul>
    <li v-for="item in list" :key="item.id">
      <div>姓名: {{ item.}}</div>
      <div>年龄:</div>
      <div>地址:</div>
    </li>
  </ul>
</template>
```

address
age
id
name

什么时候用

以下是来自社区的一些建议：

1. 你做的是一个大型的应用吗？
2. 是否是团队协作开发模式？
3. 是否在编写通用的代码库？（Vue3 / ElementPlus...）

结论：TypeScript不是万能的，技术的选型不能脱离具体的业务和应用场景，TS更加适合用来开发中大型的项目，或者是通用的JS代码库，再或者是团队协作开发的场景

怎么学

TS核心 + 综合案例

TS业务中常用的核心语法

Vue3 + TS + 实战案例

组合式API如何配合TS使用

搭建TS编译环境

为什么需要编译环境？

TypeScript编写的代码是无法直接在js引擎（浏览器/NodeJs）中运行的，最终还需要经过编译变成js代码才可以正常运行

带来的好处：既可以再开发时使用TS编写代码享受类型带来的好处，同时保证实际运行的还是JS代码

搭建手动编译环境

1. 全局安装 typescript 包（编译引擎）-> 注册 tsc 命令

   npm install -g typescript

2. 新增 hello.ts 文件， 执行 tsc hello.ts 命令生成hello.js文件

3. 执行 node hello.js 运行js文件查看效果

搭建工程化下的自动编译环境

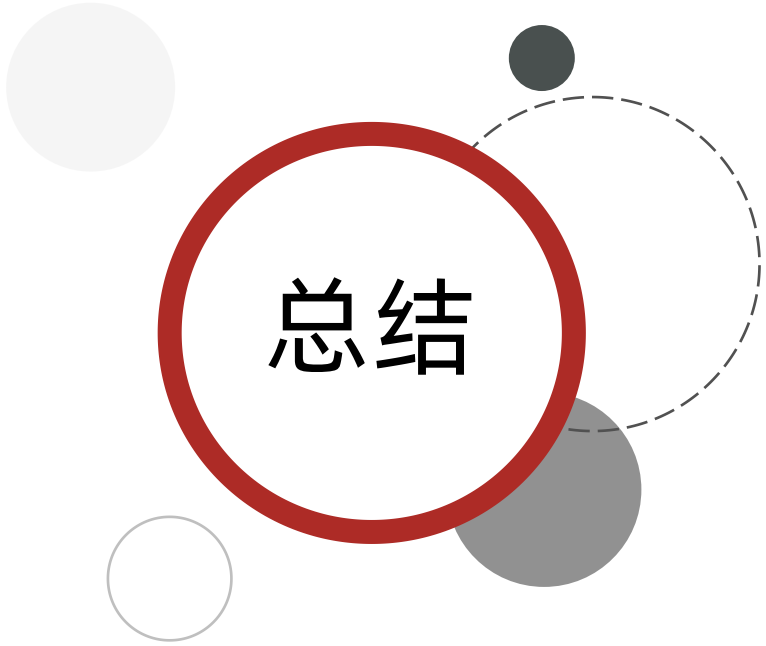
基于工程化的TS开发模式（webpack / vite），TS的编译环境已经内置了，无需手动安装配置，通过以下命令即可创建一个最基础的自动化的TS编译环境

```
npm create vite@latest   ts-pro  -- --template vanilla-ts
```

①　②　③

命令说明：

1. npm create vite@latest　使用最新版本的vite创建项目
2. ts-pro　项目名称
3. -- --template vanilla-ts　　创建项目使用的模板为原生ts模板



1. 浏览器中能直接运行TypeScript代码吗？

   不可以，需要编译为js代码再运行

2. 哪个包可以负责把TS代码编译为JS代码？

   typescript

3. 实际工作中需要我们手动编译代码吗？

   不需要，由工程化工具内置，自动编译

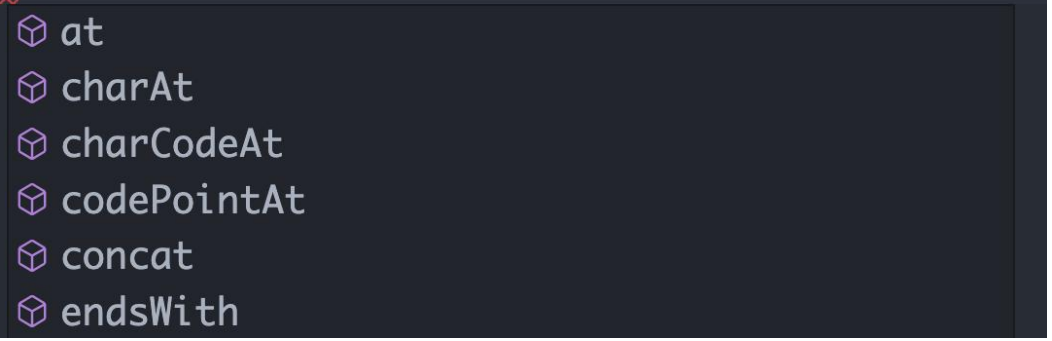
类型注解

TS类型注解是什么

概念：类型注解指的是给变量添加类型约束，使变量只能被赋值为约定好的类型，同时可以有相关的类型提示

```
let msg: string
msg = 'this is message'
msg = 100

msg.
```



说明：:string 就是类型注解， 约束变量 message 只能被赋值为string 类型，同时可以有string类型的相关提示

TS支持的常用类型注解

JS已有类型

1. 简单类型

   number string boolean null undefined

2. 复杂类型

   数组 函数

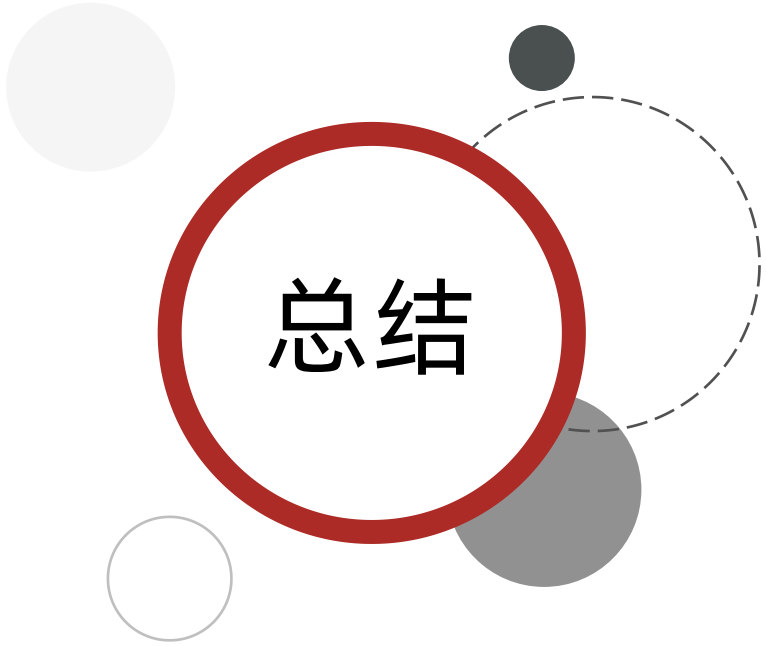
TS新增类型

联合类型、类型别名、接口（interface）、字面量类型、泛型、枚举、void、any等

简单类型如何进行类型注解

简单类型的注解完全按照 JS的类型（全小写的格式）来书写即可

```
let age: number = 18
let name: string = 'jack'
let isLoading: boolean = false
let nullValue: null = null
let undefineValue: undefined = undefined
```



1. 类型注解的作用是什么？

   限制变量能赋值的数据类型并给出提示

2. 类型注解的语法是什么？

   变量 :类型

数组类型注解

注解数组类型有什么用

变量被注解为数组类型之后，有俩点好处：

1. 不仅可以限制变量类型为数组而且可以限制数组成员的类型

```
let arr = [1, 2, 3]
arr.push('4')
```



```
item.toFixed()
```

2. 编码时不仅可以提示数组的属性和方法而且可以提示成员的属性和方法

```
let arr = [1, 2, 3]
arr.
```

concat
copyWithin
entries
every
fill
filter


```
let arr = ['1', '2', '3']
arr.forEach(item => console.log(item.))
```

charAt
charCodeAt
codePointAt
concat
endsWith
includes

如何注解数组类型

使用数据类型对变量进行类型注解有俩种语法

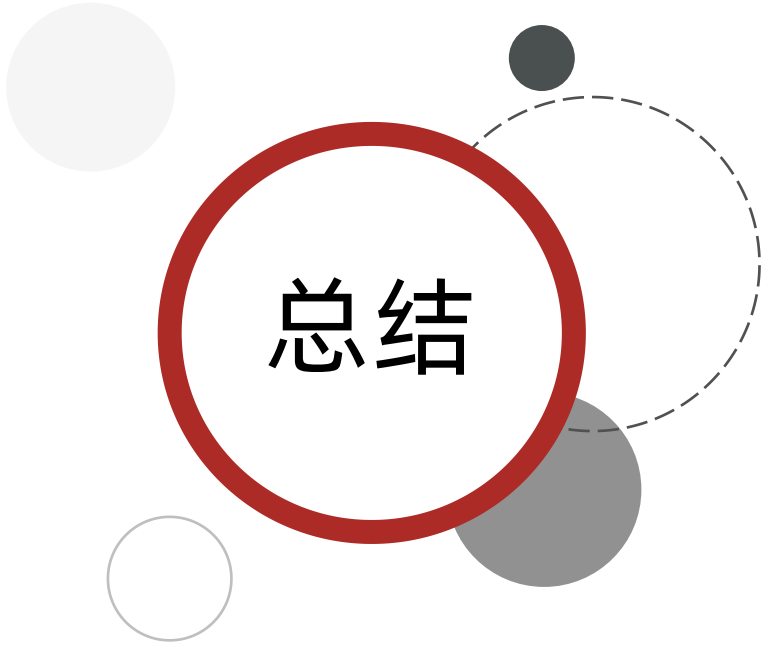
语法一（推荐）：

```
let arr: number[] = [1, 2, 3]
```

说明: 以上代码表示变量arr只能赋值数组类型并且数组的成员必须都是number类型

语法二（泛型写法）：

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



1. 数组类型注解之后除了限制了数组类型还限制了什么？

   还限制了数组中每一个成员的类型

2. 实际开发时常用的是哪种数组注解方式？

   类型[]

有一个变量arr, 要求用俩种方式添加类型注解，使其只能赋值一个成员都是字符串的数组?

联合类型和别名类型

联合类型

概念：将多个类型合并为一个类型对变量进行注解

需求：如何注解数组类型可以让数组中既可以存放string类型的成员也可以存放number类型的成员？

```
let arr3: (string | number)[] = [18, 'jack']
```

说明：string | number 表示arr3中的成员既可以是string类型也可以是number类型

类型别名

概念：通过 type关键词 给写起来较复杂的类型起一个其它的名字，用来简化和复用类型

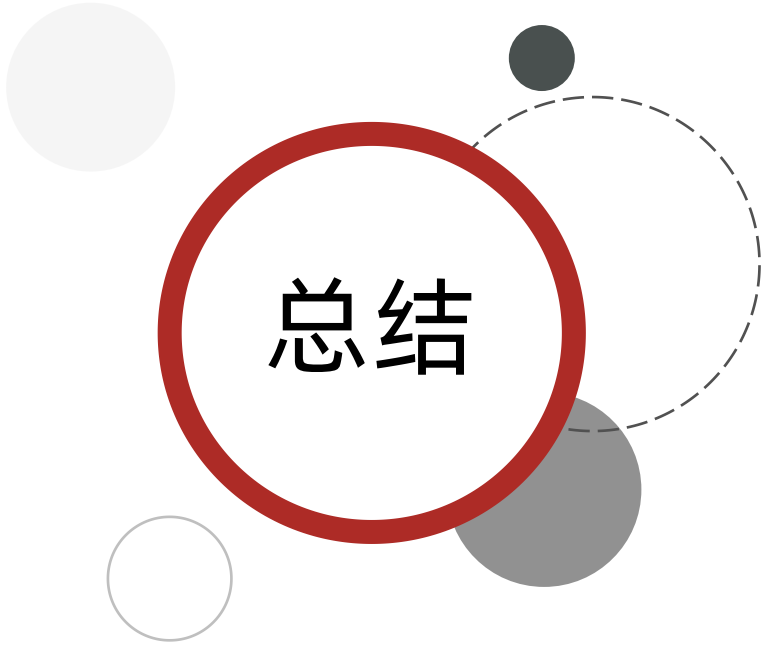
```
let arr3: (string | number)[] = [18, 'jack']
```



```
type ItemType = (string | number)[]

let arr3: ItemType = ['jack', 100]
let arr4: ItemType = ['john', 200]
```

说明：type 类型别名 = 具体类型 其中类型别名的命名采用规范的大驼峰格式



1. 联合类型的作用是什么？

   把多个类型合并为一个类型

2. 类型别名有什么好处？

   简化和复用类型

有一个变量foo,要求添加类型注解，使其既可以赋值为number类型，也可以赋值为成员都是字符串的数组

函数类型

函数类型的基础使用

概念：函数类型是指给函数添加类型注解，本质上就是给函数的参数和返回值添加类型约束

```
function add(a: number, b: number): number {
  return a + b
}
```

说明：

1. 函数参数注解类型之后不但限制了参数的类型还限制了参数为必填
2. 函数返回值注解类型之后限制了该函数内部return出去的值必须满足类型要求

好处：

1. 避免因为参数不对导致的函数内部逻辑错误　2. 对函数起到说明的作用

函数表达式（箭头函数）

函数表达式的类型注解有俩种方式，参数和返回值分开注解和函数整体注解

1. 参数和返回值分开注解

```
const add1 = (a: number, b: number): number => {
  return a + b
}
```

2. 函数整体注解（只针对于函数表达式）

```
type AddFn = (a: number, b: number) => number

const add1: AddFn = (a, b) => {
  return a + b
}
```

可选参数

概念：可选参数表示当前参数可传可不传，一旦传递实参必须保证参数类型正确

```
function buildName(firstName: string, lastName?: string) {
  if (lastName) {
    return `${firstName}${lastName}`
  } else {
    return firstName
  }
}

console.log(buildName('foo'))
console.log(buildName('foo', 'bar'))
```

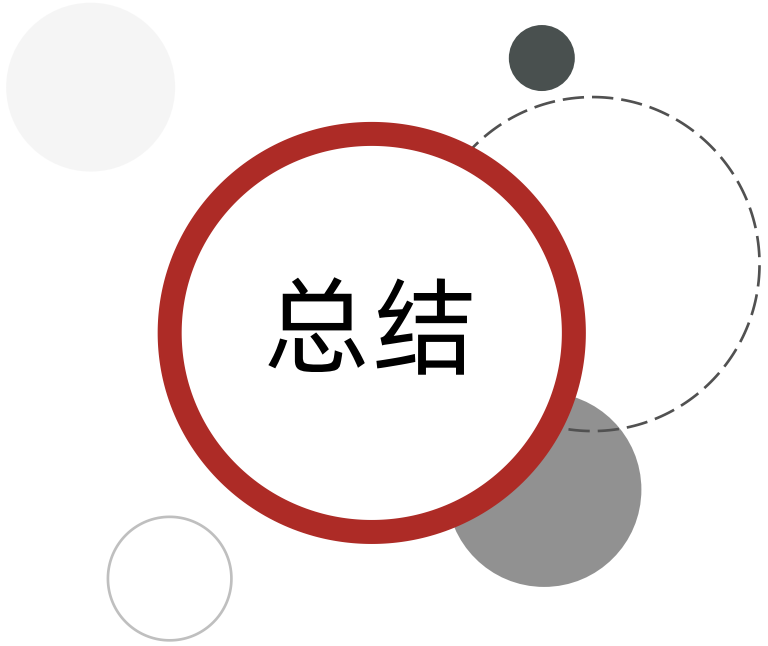
说明：lastName参数表示可选参数，可传可不传，一旦传递实参必须保证类型为string类型

无返回值 - void

概念：JS中的有些函数只有功能没有返回值，此时使用void进行返回值注解，明确表示函数没有函数值

```
function eachArr(arr: number[]): void {
  arr.forEach((item) => {
    console.log(item)
  })
}
```

注意事项：在JS中如何没有返回值，默认返回的是undefined, 在TS中 void和undefined不是一回事，undefined在TS中是一种明确的简单类型，如果指定返回值为undefined，那返回值必须是undefined类型



1. 函数类型实际上是给谁标注类型？

   参数和返回值

2. 可选参数可以不放到末尾吗？

   不可以，必须在所有参数末尾

3. 函数返回值为void和undefined类型是一回事吗？

   不是，void代表没有返回值，undefined在TS中是一种具体的类型

编写一个arr2Str函数，作用为把数组转换为字符串，其中数组中既可以包含字符串和数字，分隔符也可以进行自定义，类型为字符串类型，使用样例：

1. arr2Str([1, 2, 3] , '-') -> ‘1-2-3’

2. arr2Str([‘4’, ’5’] , ’&’) -> ‘4&5’

interface接口类型

interface接口类型的作用

作用：在TS中使用interface接口来描述对象数据的类型（常用于给对象的属性和方法添加类型约束）

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

const p: Person = {
  name: 'jack',
  age: 18,
}
```

说明：一旦注解接口类型之后对象的属性和方法类型都需要满足要求，属性不能多也不能少

interface的典型场景

场景：在常规业务开发中比较典型的就是前后端数据通信的场景

1. 前端向后端发送数据：收集表单对象数据时的类型校验

2. 前端使用后端数据：渲染后端对象数组列表时的智能提示

```
interface LoginForm {
  username: string
  password: string
}

const form: LoginForm = {
  username: 'jack',
  password: '123',
}
```

```
<template>
  <ul>
    <li v-for="item in list" :key="item.id">
      <div>姓名: {{ item.}}</div>
      <div>年龄:</div>
      <div>地址:</div>
    </li>
  </ul>
</template>
```

address
age
id
name

接口的可选设置

概念：通过？对属性进行可选标注，赋值的时候该属性可以缺失，如果有值必须保证类型满足要求

```ts
interface Props {
  type: string
  size?: string
}
```



```ts
// 不传
let props: Props = {
  type: 'success',
}

// 传必须保证类型匹配
props = {
  type: 'sucess',
  size: 'large',
}
```

接口的继承

概念：接口的很多属性是可以进行类型复用的，使用 extends 实现接口继承,实现类型复用

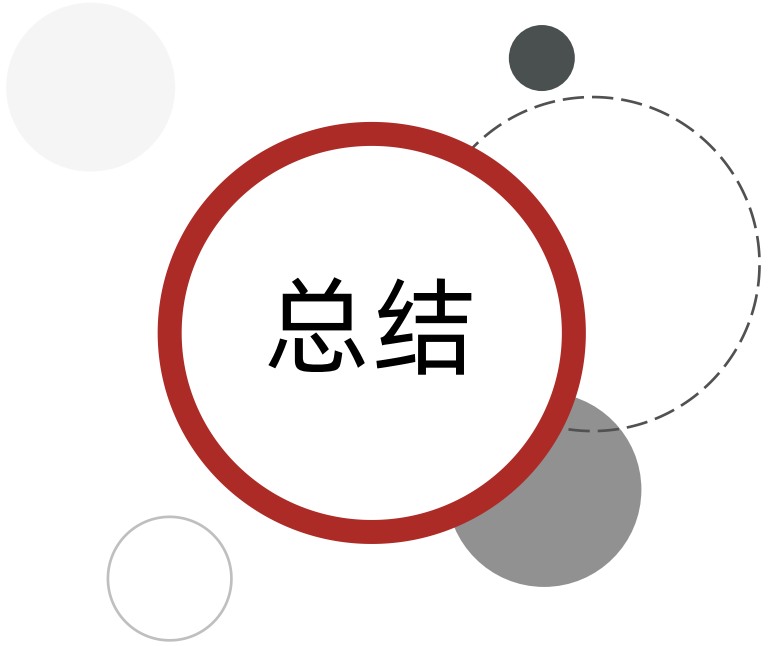
```ts
// 原价商品类型
interface GoodsType {
  id: string
  price: number
}

// 打折商品类型
interface DisGoodsType {
  id: string
  price: number
  disPrice: number
}
```



```ts
// 原价商品类型
interface GoodsType {
  id: string
  price: number
}

// 打折商品类型
interface DisGoodsType extends GoodsType {
  disPrice: number
}
```



1. interface接口类型实际限制的是什么？

   对象的属性（方法）以及其对应的类型

2. interface在业务开发中的典型场景在哪里？

   前后端通信

3. 接口继承解决了什么问题？

   类型的复用问题

通常我们的后端接口返回的数据格式具有一定的规范，比如经常见到的response对象，如下，尝试使用interface接口定义其类型

```
{
  code: 200,
  msg: 'success',
  data: {
    title: '文章标题',
    content: '文章内容',
  }
}
```

type注解对象类型

type注解对象

概念：在TS中对于对象数据的类型注解，除了使用interface之外还可以使用类型别名来进行注解，作用相似

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

const p: Person = {
  name: 'jack',
  age: 18,
}

```

type + 交叉类型模拟继承

类型别名配合交叉类型（&）可以模拟继承，同样可以实现类型复用

```ts
// 父接口
type GoodsType = {
  id: string
  price: number
}

// 子接口继承
type DisGoodsType = GoodsType & {
  disPrice: number
}
```

interface 对比 type

相同点

1. 都能描述对象类型
2. 都能实现继承，interface使用extends, type配合交叉类型

不同点

1. type除了能描述对象还可以用来自定义其他类型
2. 同名的interface会合并（属性取并集，不能出现类型冲突），同名type会报错

在注解对象类型的场景下非常相似，推荐大家一律使用type, type更加灵活

思考

还是我们熟悉的response对象，如下，尝试使用type定义其类型

```
{
  code: 200,
  msg: 'success',
  data: {
    title: '文章标题',
    content: '文章内容',
  }
}
```

字面量类型

什么是字面量类型

概念：使用 js字面量 作为类型对变量进行类型注解，这种类型就是字面量类型，字面量类型比普通的类型更加精确

```
// 普通number类型 可以赋值任何数值
let count: number
count = 100
count = 200

// 字面量类型100 只能赋值为100
let count1: 100
count1 = 100
count1 = 200     不能将类型“200”分配给类型“100”
```

说明：除了上面的数字字面量，js里常用的字符串字面量，数组字面量，对象字面量等都可以当成类型使用

字面量类型的实际应用

字面量类型在实际应用中通常和联合类型结合起来使用，提供一个精确的可选范围

场景1：性别只能是 '男' 和 '女'，就可以采用联合类型配合字面量的类型定义方案

```
type Gender = '男' | '女'

let gender:Gender = ''     不能将类型“""”分配给类型“Gender”。
                    女
                    男
```

场景2：ElementUI中的el-button组件按钮的type属性

```
type Props = {
  type: 'primary' | 'success' | 'danger' | 'warning'
}

```

字面量类型与const

思考一下下面的 str1 和 str2，TS推断出来的类型分别是什么？

```
let str1 = 'this is str'
const str2 = 'this is const string'
```

说明：const声明的变量称之为常量，常量是不可以进行重新赋值的，所以str2推断出来的是字面量类型而不是string类型

总结

1. 字面量类型通常和哪种类型一起配合使用？

   联合类型

2. 字面量类型与常规类型相比的优势是什么？

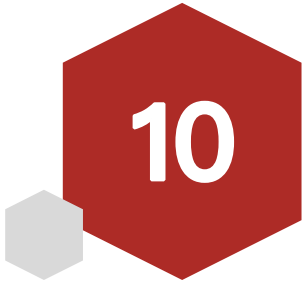
   类型更加精确，提供精确的可选值范围

思考

还是我们熟悉的后端返回数据，这一次业务code码有多种情况

1001、 1002、 1003，尝试改写类型满足要求

```
{
    code: 200,
    msg: '接口成功',
}
```



类型推论和any类型

类型推论

概念：在 TS 中存在类型推断机制，在没有给变量添加类型注解的情况下，TS 也会给变量提供类型，以下是发生类型推断的几个场景：

1. 声明变量并赋值时

```
let age: number
let age = 18
```

2. 决定函数返回值时

```
function add(a: number, b: number): number
function add(a: number, b: number) {
  return a + b
}
```

一些小建议

1. 开发项目的时候，能省略类型注解的地方就省略
2. 刚开始学TS，建议对所有类型都加上，先熟悉
3. 鼠标放至变量上，VsCode 自动提示类型

any类型

作用：变量被注解为any类型之后，TS会忽略类型检查，错误的类型赋值不会报错，也不会有任何提示

```
let obj: any = { age: 18 }
obj.bar = 100
obj()
const n: number = obj
```

注意：any 的使用越多，程序可能出现的漏洞越多，因此不推荐使用 any 类型，尽量避免使用

类型断言

类型断言的基本使用

作用：有些时候开发者比TS本身更清楚当前的类型是什么，可以使用断言（as）让类型更加精确和具体

需求：获取页面中的id为link的a元素，尝试通过点语法访问href属性

```
const aLink: HTMLElement | null
const aLink = document.getElementById('link')
```

```
const aLink1: HTMLAnchorElement
const aLink1 = document.getElementById('link') as HTMLAnchorElement
```

类型断言的注意事项

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

```
function log(foo: string | number) {
  console.log((foo as number).toFixed(2))
}

log(100)
log('100')
```

说明：利用断言把foo变量的类型指定为精确的number，但是传参的时候还是可以传递number类型或者string类型均满足类型要求，但是传递string会导致运行时错误

泛型

什么是泛型

概念：泛型（Generics）是指在定义接口、函数等类型的时候，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，使用泛型可以复用类型并且让类型更加灵活

思考：下面的俩种数据结构如何使用interface接口实现类型注解？这样做有何问题？

```
{
  code: 200,
  msg: 'success',
  data: {
    name: 'jack',
    age: 18,
  }
}
```

```
{
  code: 200,
  msg: 'success',
  data: {
    id: 1001,
    goodsName: '衬衫',
  }
}
```

泛型接口

语法：在接口类型的名称后面使用<T>即可声明一个泛型参数，接口里的其他成员都能使用该参数的类型

```
interface ResData<T> {}
```

通用思路：

1. 找到可变的类型部分通过泛型<T>抽象为泛型参数（定义参数）
2. 在使用泛型的时候，把具体类型传入到泛型参数位置 （传参）

```
// 定义泛型
interface ResData<T> {
  msg: string
  code: number
  data: T
}
```

```
// 定义具体类型
interface User {
  name: string
  age: number
}
// 使用泛型并传入具体类型
let userData: ResData<User> = {
  code: 200,
  msg: 'success',
  data: {
    name: 'jack',
    age: 18,
  },
}
```

```
// 定义具体类型
interface Goods {
  id: number
  goodsName: string
}
// 使用泛型并传入具体类型
let goodsData: ResData<Goods> = {
  code: 200,
  msg: 'success',
  data: {
    id: 1001,
    goodsName: '衬衫',
  },
}
```

泛型别名

语法：在类型别名type的后面使用<T>即可声明一个泛型参数，接口里的其他成员都能使用该参数的类型

```
type ResData<T> = {}
```

需求：使用泛型别名重构ResData案例

```
// 定义泛型
type ResData<T> = {
  msg: string
  code: number
  data: T
}
```

```
// 定义具体类型
type User = {
  name: string
  age: number
}
// 使用泛型并传入具体类型
let userData: ResData<User> = {
  code: 200,
  msg: 'success',
  data: {
    name: 'jack',
    age: 18,
  },
}
```

```
// 定义具体类型
type Goods = {
  id: number
  goodsName: string
}
// 使用泛型并传入具体类型
let goodsData: ResData<Goods> = {
  code: 200,
  msg: 'success',
  data: {
    id: 1001,
    goodsName: '衬衫',
  },
}
```

泛型函数

语法：在函数名称的后面使用<T>即可声明一个泛型参数，整个函数中（参数、返回值、函数体）的变量都可以使用该参数的类型

```
function fn<T>(){}
```

需求：设置一个函数 createArray，它可以创建一个指定长度的数组，同时将每一项都填充一个默认值(多种类型)

```
function createArray (length, value) {
  let result = []
  for (let i = 0; i < length; i++) {
    result[i] = value
  }
  return result
}
```

```
function createArray<T>(len: number, value: T) {
  let result = []
  for (let i = 0; i < len; i++) {
    result[i] = value
  }
  return result
}

createArray<number>(4, 100)

createArray<string>(4, '100')
```

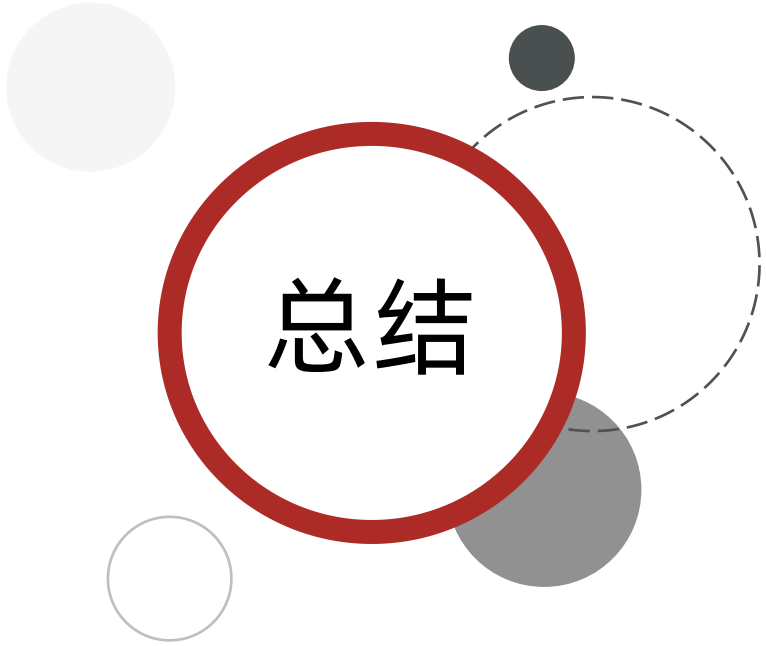
泛型约束

作用：泛型的特点就是灵活不确定，有些时候泛型函数的内部需要访问一些特定类型的数据才有的属性，此时会有类型错误，需要通过泛型约束解决

```
function logLen<T>(obj: T) {
  console.log(obj.length)
}
```

```
// 定义接口
interface LengthObj {
  length: number
}
// 泛型约束
function logLen<T extends LengthObj>(obj: T) {
  console.log(obj.length)
}
```

```
logLen({ length: 8 })
logLen(['1'])
logLen(100)
```



1. 泛型的好处是什么？

   增加类型的复用性和灵活性

2. 泛型实现的基本方法是什么？

   1. 找到类型不确定的类型部分，定义泛型参数

   2. 定义具体类型，传入泛型参数的位置

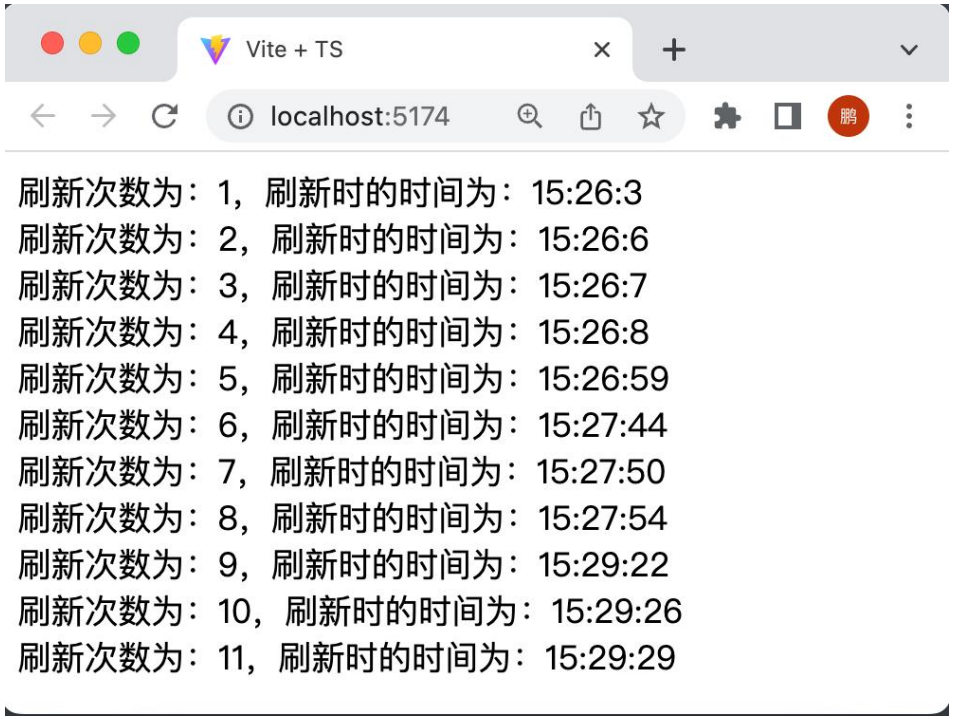
3. 泛型约束的作用是什么？

   既可以保留泛型的灵活性，又做了特定的限制

综合案例

需求描述

记录当前页面的刷新次数和刷新时的时间，每次刷新都自动自增一次，并显示到页面中，要求用TypeScript实现



核心思路

1. 从本地获取到当前最新列表，取出当前列表中的最后一条记录
2. 在最后一条记录的基础上把次数加一,重新把次数和当前时间添加到列表的尾部
3. 把最新列表渲染到页面
4. 把最新列表再次存入本地

传智教育旗下高端IT教育品牌