TS

介绍

- 6. 其它 | Vue3+TS 快速上手 (gitee.io)
- vscode 中設置添加 "typescript.tsserver.useSyntaxServer": "auto",
- TypeScript(TS)是JavaScript的超集 添加了 类型支持 微软开发的
- TS属于 静态类型 编译期做类型检查 JS属于动态编程 执行期做类型检查 先编译 后执行
- 优点：
  - 减少找bug 改bug 时间
  - 任何位置都有代码提示
  - 重构代码更加容易 支持ES最新语法
  - 类型推断机制
  - 显示标记出代码中的意外行为

安装编译

- 安装：npm i -g typescript 提供tsc命令
- 验证：tsc -v
- 运行：创建ts文件 编译 tsc hello.ts 执行 node hello.js
  - 简化：npm i -g ts-node 安装简化包 ts-node hello.ts

webpack配置TS

-

类型注解(添加类型约束)

- let age：number =18 number为类型注解 约定什么类型 只能给变量赋相同的类型 age='www' 则会报错

常用类型

js原有类型：number、string、boolean、null、undefined、symbol、object(数组、对象、函数)

- 使用 冒号名字 let sum：number =20 let name：string ='小米' ... 小写

```
//js基本
let age: number = 18
let myname: string = '小米'
console.log(`我的年龄${age}我的名字${myname}`);
```

数组

联合类型

- 联合类型： | 符号 （numbe | string)[] 既有数组又用字符串

函数

- 单独指定参数、返回值的类型 function add(参数：参数类型): 返回值类型{}
- 同时：函数名:(参数：参数类型)=>返回值类型=（参数）=>{} 只适用于函数表达式
- 可选参数：？ 只能出现在参数列表 最后
- 剩余参数(rest参数) ..args：类型 放在最后
- 函数重载 函数名字相同，函数参数个数不同

```
// 定义一个函数
// 需求：我们有一个add函数，它可以接收2个string类型的参数进行拼接，也可以接收2个number类型的参数进行相加

// 函数重载声明
function add(x: string, y: string): string
function add(x: number, y: number): number

// 函数声明
function add(x: string | number, y: string | number): string | number {
  if (typeof x === 'string' && typeof y === 'string') {
    return x + y // 字符串拼接
  } else if (typeof x === 'number' && typeof y === 'number') {
    return x + y // 数字相加
  }
}

// 函数调用
// 两个参数都是字符串
console.log(add('诸葛', '孔明'))
// 两个参数都是数字
console.log(add(10, 20))
// 此时如果传入的是非法的数据，ts应该给我提示出错误的信息内容,报红色错误的信息
console.log(add('真香', 10))
console.log(add(100, '真好'))
})()
```

对象

- 对象:{属性：类型;方法名(参数：参数类型)：返回值类型}={属性：属性值} 类型注解一行中需要 加分号
- 方法第二中写法： sayhi:()=>参数类型
- 可选属性？

```
//数组
let arr0: number[] = [1, 2, 3]
let arr2: Array<string> = ['1', '2', '4']
//联合类型
let arr: (number | string)[] = [1, '2', 3]
console.log(arr);
//函数
//单独指定类型
//add(参数：参数类型)：返回值类型
function add(x: number, y: number): number {
```

```
    return x + y
}
const add1 = (x: number, y: number): number => {
    return x + y
}
//同时指定
//函数名:(参数：参数类型)=>返回值类型=（参数）=>{}
const add2: (x: number, y: number) => number = (x, y) => {
    return x + y
}
//可选参数
function myslice(start?: number, end?: number): void {
    console.log('起始索引', start, '结束索引', end);
}
myslice()
myslice(1)
myslice(1, 2)
//对象:{属性：类型 ; 方法名(参数：参数类型)：返回值类型}={属性：属性值}

let person: { name: string; age: number; sayhi(): void; jsage(age: number): number } = {
    name: 'John',
    age: 34,
    sayhi() {
        console.log('我叫' + this.name);
    },
    jsage(age: number) {
        return age
    }
}
//对象可选属性
function axios(config: { url: string; method?: number }) {
    console.log(config);
}
axios({
    url: 'http://localhost',
})
```

Ts新增

ts新增：自定义类型（类型别名）、接口、元组、字面量、枚举、void、any等

自定义类型

自定义类型 type CustomArray=(number | string)[]

void

void类型：无返回值类型函数 function 函数名(参数：参数类型)：void{}

```
//自定义类型
type CustomArray = (number | string)[]
let arr1: CustomArray = [1, '2', 3]
//void
//function 函数名(参数: 参数类型): void{}
function g(name: string): void {
    console.log('hello', name);
}
g('jacl')

```

接口

- 接口
  - 接口来描述对象的类型 达到复用
  - 使用关键字 interface 变量名
  - interface接口与 type 类型别名
    - 相同点：都可以给对象指定类型
    - 不同点：接口只能为对象指定类型 类型别名 可以为任意类型指定别名
  - 继承：两个接口之间都有相同的属性和方法 可以将公共的属性和方法抽离出来 通过extends复用
    - 关键字 继承者(子) extends 被继承者(父)

```
//接口
interface IPerson {
    name: string
    age: number
    sayhi(): void
}
let person2: IPerson = {
    name: 'John',
    age: 34,
    sayhi() { }
}
//类型别名
type IPerson = {
    name: string
    age: number
    sayhi(): void
}
let person2: IPerson = {
    name: 'John',
    age: 34,
    sayhi() { }
}
//继承
interface IPerson2 extends IPerson { sayhello(): void }
let person3: IPerson2 = {

    name: 'John',
```

```
    age: 34,
    sayhi() { },
    sayhello() { }
}
```

元组

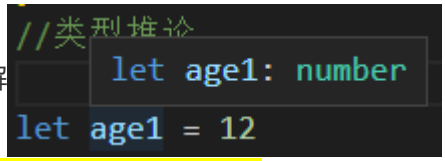
元组

- 场景 地图位置信息(经纬度)
- 确切地知道包含多少个元素
- let position：[number,number]=[39.5427,116.2314]

类型推论

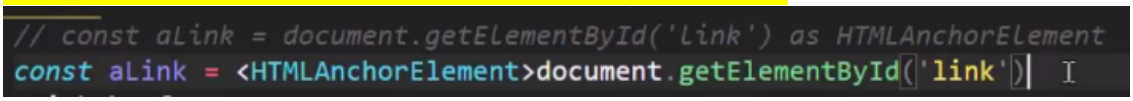
类型推论

- 初始化时 可以省略类型注解



- 如果变量没有立即初始化值时，需要添加类型注解
- 当函数有返回值时 返回值的类型注解也可以省略
- 能省略则省略

类型断言

- 类型断言
  - 当你比TS 知道改元素类型时 将使用类型断言 获取元素后 as 元素类型
    - as HTMLxxxElement （a标签 HTMLAnchorElement ）
    - 其他可通过 浏览器控制台 点击想看的标签 consloe.dir($0) 列表最后
    - 

字面量

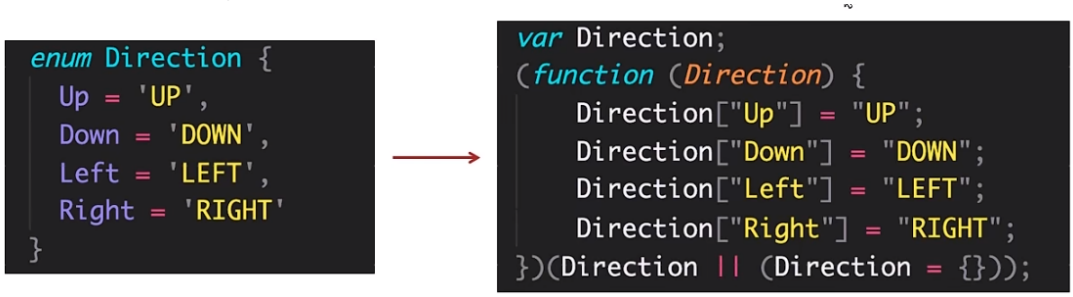
- 变量的值可以为任意 所以类型可以改变
- 常量的值不能改变 则类型注解为值
- 任意js字面量(对象、数字等)都可以作为类型使用
-
```
//字面量
let str = 'Hello Ts'
const str2: 'hello Ts' = 'hello Ts'
let age18: 18 = 18   //如果等于19  则会报错
```

- 作用：表示一组明确的可选值列表= 配合联合类型使用 更加严谨和精确

- ```
  //贪吃蛇  方向只能为上下左右
  function changefangxiang(fangxiang: '上' | '下' | '左' | '右') {
      console.log(fangxiang);
  }
  changefangxiang('上')
  ```

枚举

- 枚举类型(字面量类型+联合类型组合的功能) 也可以表示一组明确的可选值 一般推荐使用字面量
- - 枚举:定义一组变量名常量 关键字 enum
  - 枚举中值以大写字母开头 用逗号分隔 直接用枚举名作为类型注解
  - (.)语法 访问枚举成员
  - ```
    //贪吃蛇  枚举
    enum Fangxiang { Up, Down, Left, Right }
    function change(fangxiang: Fangxiang) {
        console.log(fangxiang);
    }
    change(Fangxiang.Left)
    ```
  - 枚举成员的值、
    - 数字枚举
      - 将鼠标移入 Fangxiang.Left 可以看到值为2 从0开始 自增
      - 初始化枚举值 enum Fangxiang { Up=10, Down, Left, Right } 则 left=12
    - 字符串枚举
      - 没有自增行为 字符串枚举的每个成员必须有初始值
  - 特性和原理
    - 枚举不仅用作类型，还提供值
    - 枚举类型会被编译成js代码



说明：枚举与前面讲到的字面量类型+联合类型组合的功能类似，都用来表示一组明确的可选值列表。

一般情况下，推荐使用字面量类型+联合类型组合的方式，因为相比枚举，这种方式更加直观、简洁、高效。

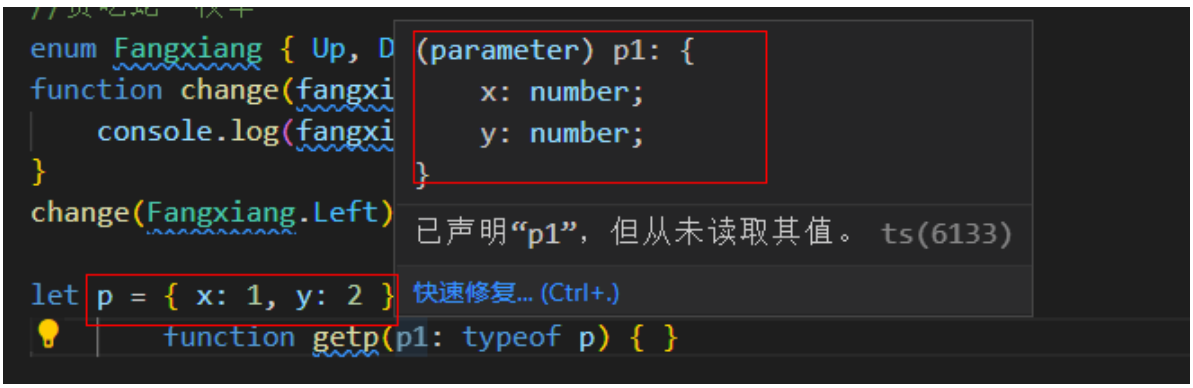
any

- 不推荐使用any 失去ts类型保护优势 可以对值进行任意操作 不会有代码提示
  - let obj：any={x:0}

- 声明的变量 无代码错误提示
- 隐式any 声明变量不提供类型也不提供默认值 函数参数不加类型

typeof

- typeof
  - 新功能 类型上下文 类型查询 typeof出现在 类型注解的位置时
  - typeof 可以获取其他变量的类型 只能用来查询变量或属性的类型 无法查询其他形式的类型如函数调用
    - let num： typeof p.x （V） let num2： typeof add(1,2) （x） 报错无法获取
  -



高级类型

class类

class基本使用

- 语法与js相同 创建类class person{} 创建实例 const p = Person() 类型注解为类
- 实例属性： age：number（无默认值） gender='男' (有默认值可省略类型注解)
-

```
class Person {
    age: number
    gender = '男'
    }
}
const p = new Person()
```

class构造函数

- constructor(参数：参数类型){} 返回值类型不能出现在构造函数里

- ```
  class Person {
      age: number
      gender: string
      //构造函数
      constructor(age: number, gender: string) {
          this.age = age
          this.gender = gender
      }
  }
  const p = new Person(12, '男')
  ```

class实例方法

- 方法的类型注解与函数用法相同

- ```
  //定义一个数学方法类  sum方法 两个数相加
  class myMath {
      x: number
      y: number
      constructor(x?: number, y?: number) {
          this.x = x || 0
          this.y = y || 0
      }
      sum(x: number, y: number): number {
          return x + y
      }
  }
  const math = new myMath()
  console.log(math.sum(2, 6));//8
  ```

class 类的继承

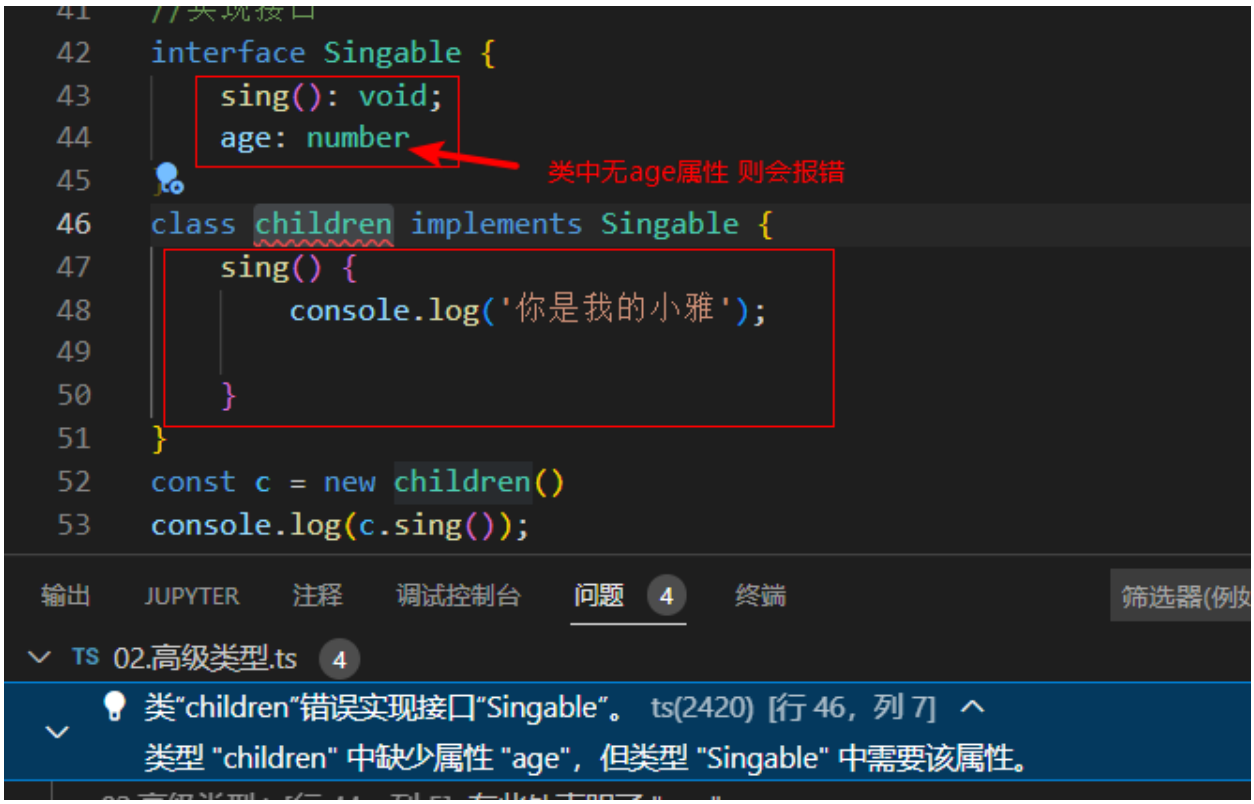
- extends 继承父类 implements 实现接口.
- super 调用父类构造函数 实现子类中属性的初始化操作
- 实现接口意外着，类中必须提供接口中指定的所有方法和属性

- ```
  // extends 继承
  class a {
      saya() {
          console.log('我是 a');
      }
  }
  class b extends a {
      sayb() {
          console.log('我是B');
      }
  }
  const B = new b()
  console.log(B.saya());
  ```
  ```

```ts

//实现接口    Singable接口中的所有方法和属性    在children类中都必须存在
interface Singable {
    sing(): void;
    age: number
}
class children implements Singable {
    sing() {
        console.log('你是我的小雅');

    }
}
```



存取器

- 让我可以有效的控制 对 对象中的成员的访问，通过 getters和setters 操作
    - get 和 set

- 

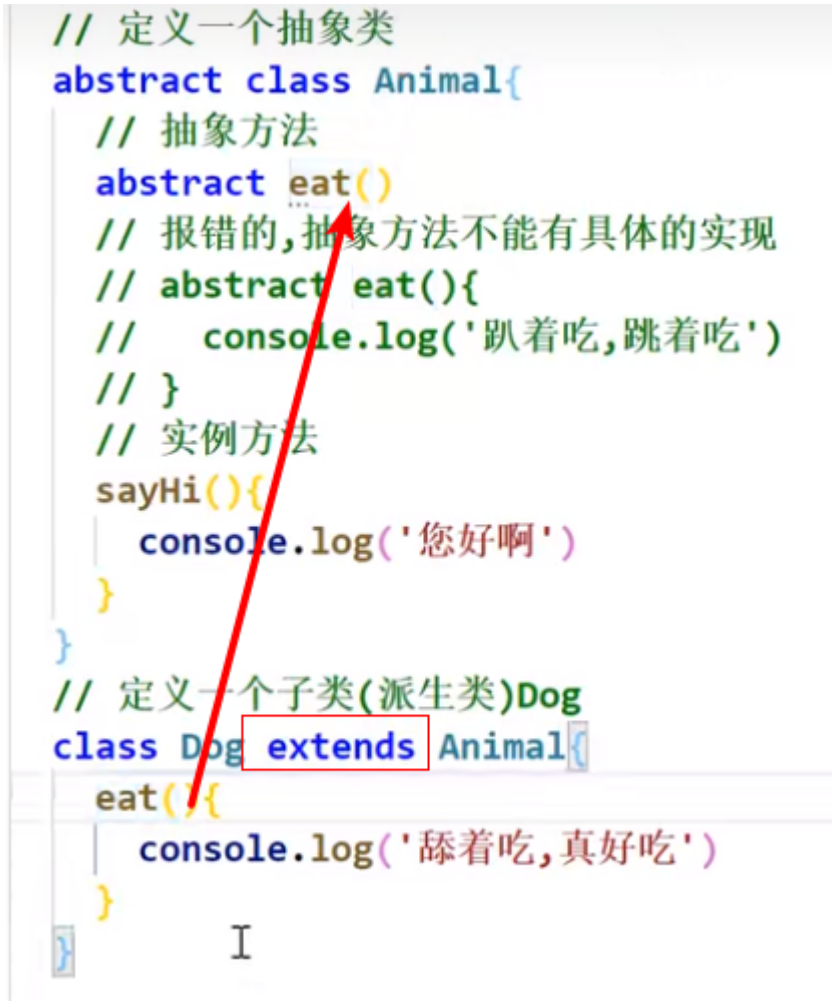
```
class Person {
  firstName: string // 姓氏
  lastName: string // 名字
  constructor(firstName: string, lastName: string) {
    this.firstName = firstName
    this.lastName = lastName
  }
  // 姓名的成员属性(外部可以访问,也可以修改)

  // 读取器----负责读取数据的
  get fullName() {
    console.log('get中...')
    // 姓名====>姓氏和名字的拼接
    return this.firstName + '_' + this.lastName
  }
  // 设置器----负责设置数据的(修改)
  set fullName(val) {
    console.log('set中...')
    // 姓名---->把姓氏和名字获取到重新的赋值给firstName和lastName
    let names = val.split('_')
    this.firstName = names[0]
    this.lastName = names[1]

  }
}
```

抽象类

- 没有任何内容的实现 抽象放法 也可包含实例方法 不能被实例化 为了让子类进行实例化及实例内部的抽象方法
- 抽象类的目的为了子类服务的
  - abstract 关键词

- 

class可见性修饰符

- piblicg共有的(默认) protected 受保护的 private私有
- static静态属性 通过类名.的这种语法调用 无法使用this 调用
- protected 仅对其申声明所在类和子类(非实例对象)中可见 子类通过 this.父类方法名 访问父类中的方法
- readonly 只读 只能在构造函数里对属性赋值 只能修饰属性 接口 或者{} 表示的对象类型 也可以使用
-

```
class a {
    readonly age: number=11  //类型为 number
    readonly age=11  //类型为11
    public saya() {
        console.log('我是 a');
        console.log('我是公有的');
    }
    protected move() {
        console.log('我是受保护的');
    }
    private sing() {
        console.log('我是' + name);
        console.log('我是私有');

    }
```

```
}
//接口
interface Iperson {
    readonly name: string;
}
let ogj: Iperson = {
    name: 'w'
}

//对象
let obj: { readonly name: string } = {
    name: 'www'
}

```

类型兼容性

- TS采用的结构化类型系统
  - 如 p：Point =new Point2D()　只检查结构、属性、属性类型都一样时则不会报错

对象

- 对于对象类型来说，y的成员至少与x相同，则 x 兼容y (成员多的可以赋值给少的)
- 只要满足 前面类型要求的成员数量 就可以实现类型兼容
-

```
class Point { x: number; y: number }
class Point3d { x: number; y: number; z: number }
const w: Point = new Point3d();
```

接口

- 接口类型之间兼容： 类似class 并且，class和interface之间也可以兼容
-

```

//接口
interface Point3D { x: number; y: number; z: number }
interface Point2D { x: number; y: number; }
interface Point1D { x: number; y: number; }
class Point3d { x: number; y: number; z: number }
let w1: Point1D = { x: 4, y: 5 }
let w3: Point3D = { x: 4, y: 5, z: 6 }
let p2: Point2D = { x: 4, y: 5 }
w1 = w3
w1 = p2
p2 = w3
w3 = new Point3d()   //class和interface之间也可以兼容
```

函数

- 比较复杂 参数个数 参数类型 返回值类型
- 参数个数
  - 参数少的可以赋值给参数多的
  - ```
    //函数
    type F1 = (a: number) => void
    type F2 = (a: number, b: number) => void
    let f1: F1 = (a = 1) => { }
    let f2: F2 = f1
    ```

- 参数类型
  - 相同位置的参数类型要相同或者兼容
  - 原始类型相同

    ```
    type F1 = (a: number) => void
    type F2 = (a: number) => void
    let f1: F1 = (a = 1) => { }
    let f2: F2 = f1
    ```

  - 对象类型

    ```
    //接口
    interface Point3D { x: number; y: number; z: number }
    interface Point2D { x: number; y: number; }
    type F1 = (p: Point3D) => void  //3个参数
    type F2 = (p: Point2D) => void  //2个参数
    let f1: F1
    let f2: F2
    f1 = f2   //  少赋给多
    ```

- 返回值类型
  - 原始类型本身是否相同 要number 都number
  - 对象类型 则按对象类型兼容性 考虑 此时则多的可以赋给少的

交叉类型

- 类似与接口 用于组合多个类型为一个类型 (常用于对象类型) &符号
    ```

- ```ts
  interface a { name: string }
  interface b { age: number }
  type c = a & b
  //上面代码 相当于   type c= { name: string; age: number }
  let obj: c = {
      name: 'www',
      age: 18
  }
  ```

- 交叉类型（&）和接口继承（extends）的对比：
  - 相同点：都可以实现对象类型的组合。
  - 不同点：两种方式实现类型组合时，对于同名属性之间，处理类型冲突的方式不同。

```ts
interface A {
  fn: (value: number) => string
}
interface B extends A {
  fn: (value: string) => string
}
```

```ts
interface A {
  fn: (value: number) => string
}
interface B {
  fn: (value: string) => string
}
type C = A & B
```

说明：以上代码，接口继承会报错（类型不兼容）；交叉类型没有错误，可以简单的理解为：

```ts
fn: (value: string | number) => string
```

泛型

- 可以在保证类型安全与多种类型一起工作 实现复用.同时保证安全
- 创建泛型

```ts
// Type类型变量 可以替换 名字
function id<Type>(value: Type): Type {
    return value
}
//调用
const num = id<number>(10)
console.log(num);
```

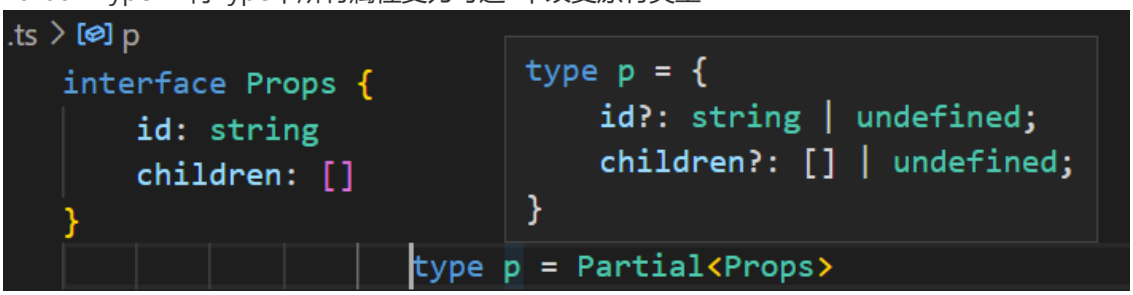
- 简化泛型 能省则省 如果系统推断的类型不明确 时 则需要手动传入
```

- ```
  function id<Type>(value: Type): Type {
      return value
  }
  const num = id(10)
  const a = id(101)
  ```

- 泛型约束
  - Type可以代表任意类型 无法保证一定存在length属性 如number就没有 length属性
  - 指定更加具体的类型

    ```

    function id<Type>(value: Type[]): Type[] {
        console.log(value.length);       //2
        return value    // 【1, 2】
    }
    const w = id([1, 2])
    console.log(w);

    ```

  - 添加约束 extends 使用接口，并添加约束 表示：传入的类型必须具有length属性

    ```
    interface length { length: number }
    function id<Type extends length>(value: Type): Type {
        console.log(value.length);
        return value
    }
    const w = id([1, 2])
    console.log(w);
    ```

  - 多个变量 用逗号 隔开 keyof 接受一个对象类型，生成其键名称的联合类型
  - keyof 接受一个对象类型

    ```
    //获取对象中存在属性值
    //key extends keyof Type  key 必须满足 Type中所有键中的其中一个  'name'或'age'
    function getprop<Type, key extends keyof Type>(obj: Type, key: key) {
        //obj:{ name: 'John', age: 18 }
        //key:'name
        //obj[name]    John      ;
        return obj[key];
    }
    let person = { name: 'John', age: 18 }
    console.log(getprop(person, 'name')); //John
    //打印传入的两个数
    function log<Type, key extends Type>(x: Type, y: key) {
        return console.log(x, y);
    }
    console.log(log(1, 2));

    ```
    ```

- 泛型接口
  - 接口类型变量Type 对接口中所有其他成员可见 都可以使用
  - ```
    interface a<Type> {
        id: (value: Type) => Type
        ids: () => Type[]
    }
    let obj: a<number> = {
        id: (value) => { return value },
        ids: () => { return [1, 2, 3] }
    }
    ```
  - 通过泛型接口实现具体参数的类型
- 泛型类
  - ```
    class Person<T1, T2> {
        name: T1
        age: T2
        constructor(name: T1, age: T2) {
            this.name = name
            this.age = age
        }
        say(name: T1, age: T2) {
            console.log(`我的名字为${name},年龄为${age}`);
        }
    }
    const p1 = new Person('李慧亮', 21)
    const p1 = new Person<string,number>('李慧亮', 21)
    const p1: Person<string,number> = new Person('李慧亮', 21)
    console.log(p1.age); //21
    console.log(p1.say('李慧亮', 21)); //我的名字为李慧亮,年龄为21
    ```
- 泛型内置方法
  - Partial<Type> 将Type中所有属性变为可选 不改变原有类型



- Readonly<Type> 所有属性变为只读

```ts
interface Props {
  id: string
  children: number[]
}
type ReadonlyProps = Readonly<Props>
```

解释：构造出来的新类型 ReadonlyProps 结构和 Props 相同，但所有属性都变为只读的。

```ts
let props: ReadonlyProps = { id: '1', children: [] }
props.id = '2'
```

- Pick<Type,keys> 选择一组属性来构造新的类型 Type是选择谁的属性 keys 那几个属性

```ts
interface Props {
  id: string
  title: string
  children: number[]
}
type PickProps = Pick<Props, 'id' | 'title'>
```

- Record<keys,Type> 构建一个对象类型 keys对象有那些属性， 属性值类型

```ts
type RecordObj = Record<'a' | 'b' | 'c', string[]>
let obj: RecordObj = {
  a: ['1'],
  b: ['2'],
  c: ['3']
}
```

索引签名类型

- 无法确定对象中有那些属性时 就用索引签名类型
- [key:string]:number key为占位符 可以换成任意名字 key:string约束 键名的类型

使用场景：当无法确定对象中有哪些属性（或者说对象中可以出现任意多个属性），此时，

```ts
interface AnyObject {
  [key: string]: number
}
```

```ts
let obj: AnyObject = {
  a: 1,
  b: 2,
}
```

解释：

映射类型

- 映射类型是基于索引签名类型的 已使用了[]　　减少重复、提升开发效率
- [key in 类型]：类型　in关键字

```
type PropKeys = 'x' | 'y' | 'z'
type Type2 = { [Key in PropKeys]: number }
```

```
type Props = { a: number; b: string; c: boolean }
type Type3 = { [key in keyof Props]: number }
```

解释：

1. 首先，先执行 keyof Props 获取到对象类型 Props 中所有键的联合类型即，'a'|'b'|'c
2. 然后，Key in ... 就表示 Key 可以是 Props 中所有的键名称中的任意一个。

-

```
type Type3 = {
    a: number;
    b: number;
    c: number;
}
type Type3 = { [key in keyof Props]: number }
```

- 索引查询类型 T[p] p为配查询类型中的属性

  刚刚用到的 T[P] 语法，在 TS 中叫做索引查询（访问）类型。

  作用：用来查询属性的类型。

```
type Props = { a: number; b: string; c: boolean }
```

```
type TypeA = number
type TypeA = Props['a']
```

解释：Props['a'] 表示查询类型 Props 中属性 'a' 对应的类型 number。所以，TypeA 的类型为 number。

注意：[] 中的属性必须存在于被查询类型中，否则就会报错。

- 查询多个是 T[p|g] 用 | 隔开

索引查询类型的其他使用方式：同时查询多个索引的类型

```
type Props = { a: number; b: string; c: boolean }
type TypeA = Props['a' | 'b'] // string | number
```

解释：使用字符串字面量的联合类型，获取属性 a 和 b 对应的类型，结果为： string | number。

```
type TypeA = Props[keyof Props] // string | number | boolean
```

解释：使用 keyof 操作符获取 Props 中所有键对应的类型，结果为： string | number | boolean。

类型声明文件

- 用来为已存在的js库提供类型信息
- .ts文件 包含类型声明、可执行代码 可以编译为.js文件
- .d.ts文件 包含类型声明 不包含可执行代码 不会生成.js文件 仅用于为js提供类型信息
- 使用已有类型声明文件 内置 第三方 CTrl+左键 方法名
- 安装第三方包 npm i -D @types/包名

创建自己的类型声明文件

- 项目内共享项 多个文件 共享一个类型时
  - 创建 xxxx.d.ts 文件 使用 export 导出 导入 import {xxx} from '路径'

index.d.ts

```ts
type Props = { x: number; y: number }

export { Props }
```

- 为已有的js文件提供类型声明
  - js项目迁移到TS项目时(创建库 为他人使用)
    - declare 关键字 ： 类型声明 为其他地方(如.js文件)已存在的变量声明类型
    - 当只能再TS使用 则可以省略declare 如：type、interface
    -

utils.d.ts

```ts
declare let count: number
declare let songName: string
interface Point {
  x: number
  y: number
}
declare let position: Point

declare function add(x: number, y: number): number
declare function changeDirection(
  direction: 'up' | 'down' | 'left' | 'right'
): void
```

utils.js

```js
let count = 10
let songName = '痴心绝对'
let position = {
  x: 0,
  y: 0
}

function add(x, y) {
  return x + y
}

function changeDirection(direction) {
  console.log(direction)
}
```

```ts
declare function changeDirection(
  direction: 'up' | 'down' | 'left' | 'right'
): void

type FomartPoint = (point: Point) => void
declare const fomartPoint: FomartPoint

// 注意：类型提供好以后，需要使用 模块化方案 中提供的
//       模块化语法，来导出声明好的类型。然后，才能在
//       其他的 .ts 文件中使用
export { count, songName, position, add, changeDir
```

```ts
}

function changeDirection(direction) {
  console.log(direction)
}

const fomartPoint = point => {
  console.log('当前坐标: ', point)
}

export { count, songName, position, a
```

内置对象

（比如 DOM）的标准。

1. ECMAScript 的内置对象

> 布尔 数 字符串 日期 正则表达式 错误

```ts
/* 1. ECMAScript 的内置对象 */
let b: Boolean = new Boolean(1)
let n: Number = new Number(true)
let s: String = new String('abc')
let d: Date = new Date()
let r: RegExp = /^1/
let e: Error = new Error('error message')
b = true
// let bb: boolean = new Boolean(2)  // error
```

2. BOM 和 DOM 的内置对象

> 窗 公文 HTMLElement 文档片段 事件 节点列表

```ts
const div: HTMLElement = document.getElementById('test')
const divs: NodeList = document.querySelectorAll('div')
document.addEventListener('click', (event: MouseEvent) => {
  console.dir(event.target)
})
const fragment: DocumentFragment = document.createDocumentFragment()
```