1.安装TypeScript编译环境

- `npm install typescript -g`
- 查版本 `tsc --version`

2.安装TypeScript运行环境

2.1webpack配置

- 项目阶段使用

2.2使用ts-node

- 安装ts-node，`npm install ts-node -g`
- 安装ts-node依赖的包`npm install tslib @types/node -g`
- 运行`ts-node main.ts`

3.变量的声明

- 声明了类型后TypeScript就会进行**类型检测**，声明的类型可以称之为**类型注解**
  - **var/let/const 标识符: 数据类型 = 赋值**;

```
//这里的string是小写的，和String有区别
//string是TypeScript中定义的字符串类型，
//String是JavaScript中定义的一个字符串包装类，String/Boolean/Number
//如果给message赋值其他类型的值就会报错
let message: string = "Hello TypeScript";
```

- 声明变量的关键字
  - 在TypeScript定义变量（标识符）和ES6之后一致，可以使用var、let、const来定义(var不推荐)
- 类型推导

```
// 默认情况下进行赋值时，会将第一次赋值的值的类型，作为前面标识符的类型
// 这个过程称为类型推导/推断
// foo没有添加类型注解
let foo = "foo"
// foo = 123 //报错
```

4.JavaScript和TypeScript的数据类型

- TS包括JS所有数据类型
- number、boolean、string、array、object、null、undefined、symbol
- object类型使用object类型注解，不能获取数据也不能设置数据

4.1TypeScript类型 - any类型

- 无法确定一个变量的类型，并且可能它会发生一些变化
- 会跳过类型检查器对值的检查，**任何值都可以赋值给`any`类型，`any`类型的值也可以赋值给任何类型**
- 可以对any类型的变量进行任何的操作，包括获取不存在的属性、方法

```
// message()
// message.split(" ")
```

- 可以给一个any类型的变量赋值任何的值，比如数字、字符串的值；

```
const arr: any[] = ["111", 234] //不推荐
```

4.2TypeScript类型 - unknown类型

- 它用于描述类型不确定的变量
- **任何类型的值都可以赋值给unknown类型，但unknow类型只能赋值给any和unknown类型**
- any类型可以赋值给任意类型

```
let notSure: unknown = 4;
let uncertain: any = notSure; // OK

let notSure: any = 4;
let uncertain: unknown = notSure; // OK

let notSure: unknown = 4;
let uncertain: number = notSure; // Error
```

4.3TypeScript类型 - void类型

- 通常用来指定一个函数是没有返回值的，那么它的返回值就是void类型

4.4TypeScript类型 - never类型

- 表示那些永不存在的值的类型
    - 函数中是一个死循环或者抛出一个异常

4.5TypeScript类型 - tuple类型

- 元组类型，可以知道每个元素的类型
- 数组中通常建议存放相同类型的元素，不同类型的元素是不推荐放在数组中
- 元组中每个元素都有自己特性的类型，根据索引值获取到的值可以确定对应的类型

4.6函数的参数和返回值类型

- 给参数加上类型注解: num1: number, num2: number
- 通常情况下可以不写返回值的类型(自动推导)

```
function sum(num1: number, num2: number) {
  return num1 + num2
}
```

4.7匿名函数的参数类型

- 上下文中的函数: 可以不添加类型注解

4.8对象类型

- {x: number, y: number}

```
function printPoint(point: {x: number, y: number}) {
  console.log(point.x);
  console.log(point.y)
}
```

4.9可选类型

- {x: number, y: number, z?: number}

```
function printPoint(point: {x: number, y: number, z?: number}) {
  console.log(point.x)
  console.log(point.y)
  console.log(point.z)
}
```

- 可选链?
  - 当对象的属性不存在时，会短路，直接返回undefined，如果存在，那么才会继续执行

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

const info: Person = {
  name: "why",
  friend: {
    name: "kobe",
    girlFriend: {
      name: "lily"
    }
  }
}


// 另外一个文件中
console.log(info.name)
console.log(info.friend?.name)
console.log(info.friend?.age)
console.log(info.friend?.girlFriend?.name)
```

4.10联合类型

- number|string

4.11可选类型和联合类型的关系

- 一个参数一个可选类型的时候, 它其实类似于是这个参数是 类型|undefined 的联合类型

4.12交叉类型

- 将多个类型合并为一个类型。 这让我们可以把现有的多种类型叠加到一起成为一种类型，它包含了所需的所有类型的特性，使用`&`定义交叉类型

```
interface ISwim {
  swimming: () => void
}

interface IFly {
  flying: () => void
}

type MyType1 = ISwim | IFly
type MyType2 = ISwim & IFly

const obj1: MyType1 = {
  flying() {}
}

const obj2: MyType2 = {
  swimming() {},
  flying() {}
}
```

4.13类型别名

- type用于定义类型别名

```
type IDType = string | number | boolean
type PointType = {
  x: number
  y: number
  z?: number
}

function printId(id: IDType) {}
function printPoint(point: PointType) {}
```

4.14类型断言as

- 类型断言好比其它语言里的类型转换
- 两种方式实现

```
// 尖括号 语法
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;

// as 语法
let someValue: any = "this is a string";
let strLength: number = (someValue as string).length;

```

- 非空断言
- ```
  function printMessageLength(message?: string) {
    // vue3源码
    console.log(message!.length) //message! 非空
  }

  printMessageLength("aaaa")
  ```

4.15运算符

- !!运算符

```
const message = "Hello World"

// !! 将一个其他类型转换成boolean类型
const flag = !!message
console.log(flag)
```

- ??运算符

```
let message: string|null = "Hello World"

// 空值合并操作符（??），ES11新增
// 是一个逻辑操作符，当操作符的左侧是 null 或者 undefined 时，返回其右侧操作数，
// 否则返回左侧操作数
const content = message ?? "你好啊，李银河"
// 三目运算符
// const content = message ? message: "你好啊，李银河"
console.log(content)
```

操作符

- keyof

```
//可以获取一个类型所有键值，返回一个联合类型，如下
type Person = {
  name: string;
  age: number;
}
type PersonKey = keyof Person;  // PersonKey得到的类型为 'name' | 'age'
```

- typeof
```

```
//获取实例类型
const me: Person = { name: 'gzx', age: 16 };
type P = typeof me;  // { name: string, age: number | undefined }
const you: typeof me = { name: 'mabaoguo', age: 69 }  // 可以通过编译
```

interface和type的区别

- 如果是定义**非对象**类型，通常推荐使用type
- 如果是定义**对象**类型，那么他们是有区别的
  - interface 可以重复的对某个接口来定义属性和方法，多个可以合并
  - type定义的是别名，别名是不能重复的，type 比 interface 更方便拓展一些

接口

- 定义一些参数，规定变量里面有什么参数，参数是什么类型，使用时就必须有这些对应类型的参数，少或者多参数、参数类型不对都会报错。
- 更简单的，你可以理解为这就是在定义一个较为详细的对象类型

可选属性

- 在可选属性名字定义的后面加一个`?`符号，来证明该属性是可有可无的

```
interface Props {
  name: string;
  age: number;
  money?: number;
}
```

只读属性

- 在属性名前用`readonly`关键字来指定只读属性，该对象属性只能在对象刚刚创建的时候修改其值，与`const`类似

```
interface Point {
  readonly x: number;
  readonly y: number;
}

let p: Point = { x: 10, y: 20 };
p.x = 5; // Error
```

接口继承

- 使用关键字`extends`,继承的本质是复制，抽出共同的代码，所以子接口拥有父接口的类型定义:
- 接口可以多继承

```ts
interface Shape {
  color: string;
}
interface PenStroke {
  penWidth: number;
}
interface Square extends Shape, PenStroke {
  sideLength: number;
}

let square: Square = { sideLength: 1 } // Error
let square1: Square = { sideLength: 1, color: 'red' } // Error
let square2: Square = { sideLength: 1, color: 'red', penWidth: 2 } // OK
```

枚举类型

- 枚举 enum 将一组可能出现的值，一个个列举出来，定义在一个类型中

```ts
enum Direction {
  LEFT = "LEFT",
  RIGHT = "RIGHT",
  TOP = "TOP",
  BOTTOM = "BOTTOM"
}
```

函数类型

类

- class定义

修饰符

- public 默认修饰符，TypeScript中类中的成员默认为public
- private 类的成员不能在类的外部访问，子类也不可以
- protected 类的成员不能在类的外部访问，但是子类中可以访问。如果一个类的构造函数，修饰符为protected，那么此类只能被继承，无法实例化。
- readonly 关键字`readonly`可以将实例的属性，设置为只读

泛型

- 泛型是指在定义函数、接口或类的时候，不预先指定具体的类型，使用时再去指定类型的一种特性。
- 可以把泛型理解为代表类型的参数

```ts
function sum<Type>(num: Type): Type {
  return num
}

// 1.调用方式一：明确的传入类型
sum<number>(20)
sum<{name: string}>({name: "Kylin"})
sum<any[]>(["abc"])

// 2.调用方式二：类型推到
sum(50)
sum("abc")
```

泛型工具

- **Partical**
  - 将泛型中全部属性变为可选的

```ts
type Animal = {
  name: string,
  category: string,
  age: number,
  eat: () => number
}
type PartOfAnimal = Partical<Animal>;
const ww: PartOfAnimal = { name: 'ww' }; // 属性全部可选后，可以只赋值部分属性了
```

- **Record<K, T>**
  - 将 K 中所有属性值转化为 T 类型，我们常用它来申明一个普通 object 对象
- **Pick<T, K>**
  - 将 T 类型中的 K 键列表提取出来，生成新的子键值对类型
- **Exclude<T, U>**
  - 在 T 类型中，去除 T 类型和 U 类型的交集，返回剩余的部分
- **Omit<T, K>**
  - 适用于键值对对象的 Exclude，它会去除类型 T 中包含 K 的键值对
- **ReturnType**
  - 获取 T 类型(函数)对应的返回值类型
- **Required**
  - 将类型 T 中所有的属性变为必选项

模块化开发

- TypeScript支持两种方式来控制我们的作用域：
  - 模块化：每个文件可以是一个独立的模块，支持ES Module，也支持CommonJS
  - 命名空间：通过namespace来声明一个命名空间

类型声明

- 内置类型声明
  - 内置类型声明是typescript自带的、帮助我们内置了JavaScript运行时的一些标准化API的声明文件
- 外部定义类型声明
  - 外部类型声明通常是我们使用一些库（比如第三方库）时，需要的一些类型声明
  - 这些库通常有两种类型声明方式
    - 方式一：在自己库中进行类型声明（编写.d.ts文件），比如axios
    - 方式二：通过社区的一个公有库DefinitelyTyped存放类型声明文件
      - 查找声明安装方式的地址：https://www.typescriptlang.org/dt/search?search=
- 自己定义类型声明
  - 情况一：我们使用的第三方库是一个纯的JavaScript库，没有对应的声明文件；比如lodash
  - 情况二：我们给自己的代码中声明一些类型，方便在其他地方直接进行使用