TypeScript入门

1.什么是TypeScirpt

1.1 TypeScript简介

TypeScript是微软开发的一个开源的编程语言，通过在JavaScript的基础上添加静态类型定义构建而成。TypeScript通过TypeScript编译器或Babel转译为JavaScript代码，可运行在任何浏览器，任何操作系统。

1.2 TypeScript和JavaScript的关系

TypeScript是JavaScript的一个超集，他们之间并不是所属关系，TypeScript扩展了JavaScript弱类型语言的限制，增加了更多的模块解析方式和语法糖。TypeScript并不是一个能独立运行的语言，大多数时候他都被转译成JavaScript运行，所以可以简单的认为TypeScript相当于功能更丰富的编译型的JavaScript。

1.3 为什么要使用TypeScript

传统的JavaScript本身已完全可以满足完整的应用开发需求，但在大型项目协作开发或插件开发的场景中JavaScript弱类型语言的不足便暴漏出来。由于JavaScript并非编译型语言，在代码编写过程中无法轻松的实现良好的类型约束和类型推断。若开发者提供了一个JavaScript依赖包给其他开发者使用，使用依赖的开发者并不能显示观察依赖包内部的类型组成，很容易出现下列场景，代码如下：

```
//若依赖包提供了属性foo并对其期待类型为string
foo = 123 //此时应用foo变量的开发者为其设置数字类型
...
foo = false //后续又存在开发者为其设置boolean类型
//这种应用方式会导致在代码阅读上无法确定该属性的明确类型
//也会导致运行上的一定风险
```

综上所述，JavaScript语言在代码的可维护性上存在一些弱项，所以此时强类型的TypeScript语法正好适用于此类开发场景。TypeScript强类型的约束性以及其面向接口编程的约束性可以让TypeScript语法开发的应用有极强的维护性，代价是更大量的代码篇幅。TypeScript非常适合插件提供者、依赖库提供者、基于JavaScript的服务端项目以及大型项目的工程化开发使用，所以并不意味着其适用于一切JavaScript项目做改造。

1.4 TypeScript的认知误区

1.之所以TypeScript能流行，是因为其的性能优于JavaScript?

这是一个很大的误区，目前TypeScript的主流使用场景都是在Web领域，从上面的介绍得知

任何使用TypeScript脚本开发的应用或游戏，都不是通过直接运行该语言而实现应用运转的，可以认为TypeScript是一种语法，并不是运行语言。直接运行TypeScript的内核暂时尚未普及，所以绝大多数的TypeScript项目都是需要一次编译，最终执行的还是JavaScript，所以并不能代表性能优势。

TypeScript语言之所以流行，是因为其类型化的JavaScript，在上下文阅读时可以提供更好的类型追溯，通过编辑器插件可已实现更有好的提示。其模块和命名空间等能力更加符合工程化的前端思想。

2.用了TypeScript的项目就比使用JavaScript更好

这是很多认知层面的误差导致的错误认识，有一部分公司为了追赶大厂的项目架构，没有目的的在Vue或React项目中植入TypeScript模块，这种植入并没有帮助项目更好的开发，反而会因为开发者对语言的认识偏差使其变成毫无用处的鸡肋。在技术架构并不复杂的应用开发中，TypeScript的使用会增加大量的代码量，所以在简单项目开发中，单纯的使用ES6-ES2022语法足够满足工程化需求。乱用类型也会让TypeScript失去其默认的意义，很多项目中由于开发者并不熟悉TypeScript，会出现除基本类型外全部:any的实现，导致TypeScript变成AnyScript，这也是一个极其不好的现象，因为TypeScript的类型系统主要用于迭代和维护项目，最终运行的JavaScript源代码中并不会出现类型。

2.TypeScript入门

TypeScript的中文文档地址为：https://www.tslang.cn/index.html后续部分可能需要进行文档查阅。

2.1 环境搭建

打开电脑的命令行工具执行TypeScript引擎安装命令，代码如下：

```
npm install typescript -g
```

安装成功后，在命令行工具中输入引擎版本指令查看版本号码，代码如下：

```
tsc -v
```

出现版本号代表安装成功。

2.2 创建项目

TypeScript引擎安装成功后，就可以在代码编辑器中通过引擎命令实现TypeScript的项目初始化。初始化项目的方式与初始化Node项目类似。在代码编辑器中创建任意文件夹，在命令行工具中打开文件夹，输入初始化TypeScript项目的命令，代码如下：

```
tsc --init
```

命令输入后，文件夹中会自动生成名为tsconfig.json的文件，该文件内容如下：

```
{
  "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": "es2016",                                  /* 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": "commonjs",                                /* 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. */
    // "allowSyntheticDefaultImports": 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. */
    // "preserveSymlinks": 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. */
  }
}
```

本文只介绍涉及到的项目配置，接下来在tsconfig.json文件中将rootDir和outputDir进行配置，代码入如下：

```
{
  "compilerOptions": {
    "rootDirs":["./src"],
    "outDir":"./dist",
    ...
  }
}
```

完整的tsconfig.js配置文件说明详见：https://www.tslang.cn/docs/handbook/tsconfig-json.html

2.3 HelloWorld的实现

在项目中创建src文件夹，在src文件夹中创建index.ts文件，在文件中初始化HelloWorld内容，代码如下：

```
let str:string = 'hello'
console.log(str)
```

完成编码后在命令行工具中输入编译命令，代码如下：

```
tsc
```

命令执行完成后会发现项目文件夹中出现一个dist文件夹，这个就是刚才配置的输出目录，在文件夹中会对应输出index.js文件，文件内容，代码如下：

```
"use strict";
let str = 'hello';
console.log(str);
```

可以命令行工具中打开dist目录，通过NodeJS运行该文件，代码如下：

```
node index
```

控制台中会出现hello字样，此步骤没问题就代表TypeScript已经可以工作在本地计算机上了。根据运行情况得知TypeScript引擎的主要目的是将TypeScript构建成JavaScript代码，最终通过不同的JavaScript引擎来运行最终的代码。

3.TypeScript的详细学习

3.1 类型

TypeScript最具特色的就是其类型系统，类型让TypeScript变成一个更趋向于静态语言的语法，其类型的使用方式如下：

3.1.1 类型声明

通过下面的代码来学习一下TypeScript的类型生命方式吧，代码如下：

```
let str:string = '字符串'
let num:number = 123
let flag:boolean = true
let undef:undefined = undefined
let nul:null = null
console.log(str)
console.log(num)
console.log(flag)
console.log(undef)
console.log(nul)
//这里需要注意的是必须使用模块导出的方式，否则TypeScript会认为该文件的变量为全局变量而导致提示重名错误
export default {}
```

在TypeScript中声明基本类型数据只需要变量名后加入小写的类型名称即可，整体编码除此之外与JavaScript相同，需要注意的是一旦对变量设置具体类型后，该变量就必须使用单一类型，否则会出现类型错误，代码如下：

```
let str:string = '一个字符串'
str = 123
console.log(str)
export default {}
```

当运行此代码片段时，控制台会出现如下错误：

```
zhangyunpeng@zhangyunpengdeMacBook-Pro ts % tsc
src/error1.ts:2:1 - error TS2322: Type 'number' is not assignable to type 'string'.

2 str = 123
  ~~~


Found 1 error.
```

可以通过多类型创建的方式让一个变量支持多个类型，代码如下：

```
let arg:number|string = '哈哈'
console.log(arg)
arg = 123
console.log(arg)
// 报错
// arg = true
```

`never`类型表示的是那些永不存在的值的类型。 例如， `never`类型是那些总是会抛出异常或根本就不会有返回值的函数表达式或箭头函数表达式的返回值类型； 变量也可能是 `never`类型，当它们被永不为真的类型保护所约束时。

`never`类型是任何类型的子类型，也可以赋值给任何类型；然而，*没有*类型是`never`的子类型或可以赋值给`never`类型（除了`never`本身之外）。 即使 `any`也不可以赋值给`never`。

下面是一些返回`never`类型的函数：

```
// 返回never的函数必须存在无法达到的终点
function error(message: string): never {
    throw new Error(message);
}

// 推断的返回值类型为never
function fail() {
    return error("Something failed");
}

// 返回never的函数必须存在无法达到的终点
function infiniteLoop(): never {
    while (true) {
    }
```

```
}
```

3.1.2 数组元组和对象

在TypeScript中，数组的定义有很多种方式，代码如下：

```
let arr:number[] = [1,2,3]//纯数字数组
let arr1:Array<number> = [4,5,6]//纯数字数组
// let arr2:Array<string> = ['1',2,3] //该数组会触发编译错误
let tuple:[number,string,boolean] = [12,'ab',true] //该声明方式创建的为元组对象
// let tuple1:[number,string] = [12] //元组声明时即为固定长度不可以违背
console.log(arr)
console.log(arr1)
console.log(tuple)
export default {}
```

该代码案例描述了如何定义TypeScript中的数组对象，在TypeScript的世界中为数组类型的数据增加了元组的实现，但其特性主要体现在编译过程，生成的代码中类型会恢复为数组，代码如下：

```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
let arr = [1, 2, 3]; //纯数字数组
let arr1 = [4, 5, 6]; //纯数字数组
// let arr2:Array<string> = ['1',2,3] //该数组会触发编译错误
let tuple = [12, 'ab', true]; //该声明方式创建的为元组对象
// let tuple1:[number,string] = [12] //元组声明时即为固定长度不可以违背
console.log(arr);
console.log(arr1);
console.log(tuple);
exports.default = {};
```

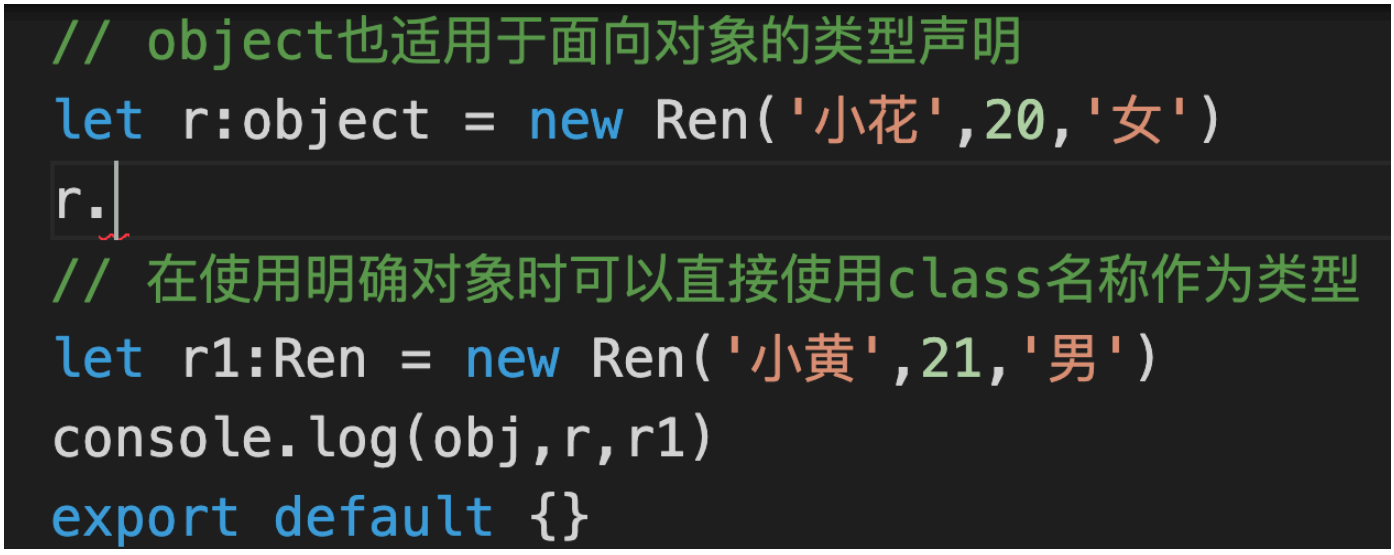
通过两个案例了解后发现在TypeScript的世界中，数组若为动态长度，则类型固定，若支持多类型数组则长度固定，接下来介绍如何声明动态类型数组，这里需要借助any关键字。在TypeScript中提供了any关键字用来表示任意类型。当属性在定义时无法推断其未来的类型或其未来会出现动态类型情况时，就可以使用any类型，代码如下：

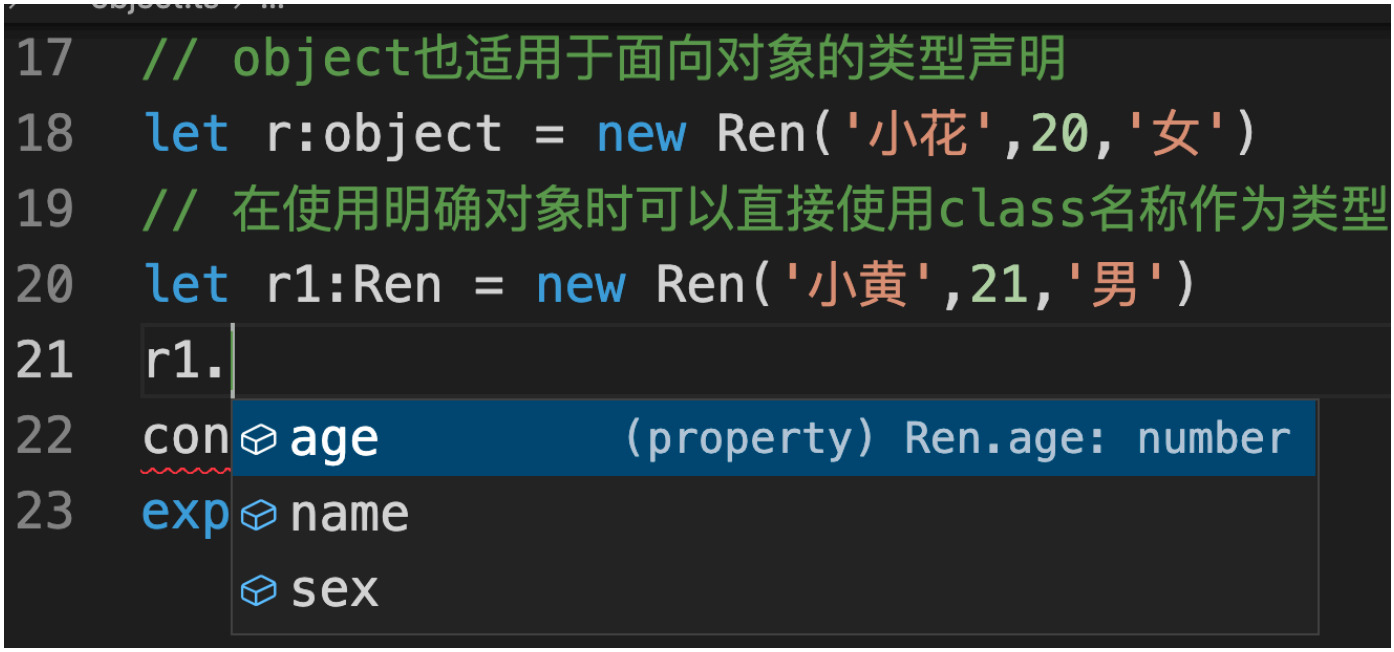
```
// any代表任意类型，一旦对变量定义为any，则其未来可以是任意类型
let arg:any = 123
console.log(arg)
arg = '字符串'
console.log(arg)
arg = [1,'a',true]
console.log(arg)
// 创建一个动态类型的数组
let arr:any[] = [2,3,'a',false]
let arr1:Array<any> = [4,5,7,'d']
console.log(arr)
arr1.push('lalal')
console.log(arr1)
```

数组和元组是编程中常用的引用类型之一，还有一个更加常用的数据类型就是对象类型，当TypeScript中使用对象时，类型定义的方式仍然有很多种个，代码如下：

```
// object是创建对象属性的最基本方式
let obj:object = {
  name:'小明',
  age:18,
  sex:'男'
}
class Ren{
  name:string
  age:number
  sex:string
  constructor(name:string,age:number,sex:string){
    this.name = name
    this.age = age
    this.sex = sex
  }
}
// object也适用于面向对象的类型声明
let r:object = new Ren('小花',20,'女')
// 在使用明确对象时可以直接使用class名称作为类型
let r1:Ren = new Ren('小黄',21,'男')
console.log(obj,r,r1)
export default {}
```

该案例用以描述如何定义对象的类型，这里的object相当于对象中的any类型，仅仅声明该变量代表一个对象，但是无法具体描述该对象的内部内容，所以用此类型定义的对象为动态对象，未来可以动态为其新增属性，而使用类名创建的对象与元组类型，其类型为固定类型，所以当使用对象类型是，明确对象类型后，编辑器可以对对象内部结构提供完整的提示和说明，如图：





对比上下两张图会发现，明确具体类型后的对象在引用时，编辑器可以为其提供完整的类型提示。

3.1.3 函数、枚举和interface

函数在TypeScript中的使用方式非常简单，参考以下函数示例即可，代码如下：

```
function test(){
  console.log('普通函数')
}
// 带参数无返回值的函数
function test1(name:string,age:number):void{
  console.log(name,age)
}
// 带参数和返回值的函数
function test2(name:string,age:number):string{
  return `name的值为：${name},age的值为${age}`
}
test()
test1('小明',18)
// 已知返回值类型后类型可省略，该变量的类型会被锁定为string不允许设置其他类型
let res = test2('小黄',20)
console.log(res)

export default{}
```

动态类型的JavaScript语言中虽然存在类型的概念，但是其并不具备静态语言的特性，TypeScript中的枚举就是静态语言中非常具有特点的数据结构，其通常用来描述不同常量值，代码如下：

```
enum ErrorCode {
  Success = 200,
  NotFoundError = 404,
  ServerError = 500,
```

```
  UnauthorizedError = 401,
  OtherError = -1
}
function getList(type:string):ErrorCode{
  if(type == 'Success'){
    return ErrorCode.Success
  }else if(type == 'NotFoundError'){
    return ErrorCode.NotFoundError
  }else if(type == 'ServerError'){
    return ErrorCode.ServerError
  }else if(type == 'UnauthorizedError'){
    return ErrorCode.UnauthorizedError
  }else{
    return ErrorCode.OtherError
  }
}
let res1 = getList('Success')
let res2 = getList('NotFoundError')
let res3 = getList('ServerError')
let res4 = getList('UnauthorizedError')
let res5 = getList('Other')
console.log(res1,res2,res3,res4,res5)
```

枚举的使用场景并不多做介绍，接下来介绍一下TypeScript中的interface。学习过面向接口编程的同学对interface的使用方式并不会陌生，一个interface是一个完全抽象的对象，一个interface可以对应多个class对其内部的未实现方法进行实现，在TypeScript中，interface主要用于类型描述，代码如下：

```
let obj = {
  name:'小明',
  age:18,
  sex:'男'
}
// 当使用函数获取该动态类型变量时，getObj的返回结果并不能描述其内部属性
function getObj():object{
  return obj
}
let obj1 = getObj()
console.log(obj1)
interface User{
  name:string,
  age:number,
  sex:string
}
function getObjType():User{
  return obj
}
// 此时返回的数据会带有该数据的类型描述
let obj2 = getObjType()
console.log(obj2)
```

```
// 只读属性定义
interface Point{
  readonly x:number,
  readonly y:number
}
let point:Point = {x:10,y:10}
// 此时point的x和y不可以更改
// point.x = 5
// 限制类型后的数据不可以使用规定类型外的属性
// let point1:Point = {x:10,y:10,z:11}
interface Config{
  entry:Array<any>,
  output:string,
  // 该属性可以动态定义非必要内容外的属性类型，使用后可以对原油对象扩展属性
  [str:string]:any
}
let config:Config = {entry:[1,2],output:'./',name:'abc'}
console.log(config)
// 定义函数类型
interface SelectFunc{
  (pno:number,psize:number):Array<User>
}
// 定义类型后该函数可以不需要描述返回值和参数类型
let func:SelectFunc = function(pno,psize){
  return [{name:'小花',age:18,sex:'女'},{name:'小白',age:18,sex:'男'}]
}
let res = func(1,10)
console.log(res)
```

在使用TypeScript进行业务开发时，可能会存在大量的动态对象，为更好的将其类型公开到开发者面前，interface的使用是必要的。

除此之外，interface可以实现多类型合并，代码如下：

```
interface User{
  name:string,
  age:number,
  sex:string
}
interface Admin{
  name:string,
  password:string
}

type Person = User | Admin
//此时的arr数组中既可以包括User的结构又可以包括Admin的结构
let arr:Array<Person> = [
  {
    name:'小明',
```

```
    age:18,
    sex:'男'
  },
  {
    name:'管理员',
    password:'123456'
  }
]
```

3.1.4 泛型

泛型是静态类型语言的另一灵魂工具，这里体现于静态类型的语言在定义类型时必须明确类型而造成的问题，代码如下：

```
function test(arg:number):number{
  return arg
}
function test1(arg:string):string{
  return arg
}
```

当函数的参数和返回类型明确时，相同结构的函数需要根据不同类型定义多个，这种情况就很容易将代码的复杂度提高并降低维护性，通过泛型便可以解决这个问题。

```
// 可以通过T指定泛型，T所代表的类型与any不同，他会识别到函数实际调用时传入的类型
// 并可以根据调用时的类型生成对应的代码提示
function test<T>(arg:T):T{
  return arg
}
// 传入字符串时，该函数的返回值类型默认为string并且无法被更改
let res = test('字符串')
console.log(res)
// 传入数字时 ，该函数的返回值类型被设定为number
let num = test(123)
console.log(num)
export default {}
```

3.2 TypeScript的其他应用

3.2.1 装饰器

随着TypeScript和ES6里引入了类，在一些场景下我们需要额外的特性来支持标注或修改类及其成员。 装饰器（Decorators）为我们在类的声明及成员上通过元编程语法添加标注提供了一种方式。 Javascript里的装饰器目前处在 建议征集的第二阶段，但在TypeScript里已做为一项实验性特性予以支持。

> 注意　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。

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`experimentalDecorators`编译器选项：

命令行:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

*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)，属性或参数上。 装饰器使用`@expression`这种形式，`expression`求值后必须为一个函数，它会在运行时被调用，被装饰的声明信息做为参数传入。

接下来通过一个简单的装饰器示例了解一下装饰器在TypeScript中的使用方式，代码如下：

```
//装饰器
function FormatDate(){
  return function(target:any,key:string,descriptor: PropertyDescriptor){
    //拦截该装饰器所应用属性的set方法
    descriptor.set = function(d:Date){
      let year = d.getFullYear()
      let month = d.getMonth()+1
      let date = d.getDate()
      let _this:any = this
      _this['_'+key] = `${year}-${month}-${date}`
    }
  }
}
interface User{
  name:string,
  age:number,
  [props:string]:any
}
```

```
class User{
  //通过装饰器实现自动格式化时间
  @FormatDate()
  set birthday(v){this._birthday = v}

  get birthday():Date{ return this._birthday }

  constructor({name,age,birthday}:User){
    this.name = name,
    this.age = age

    this.birthday = birthday
  }

}
let u = new User({name:'小明',age:18,birthday:new Date()})
console.log(u)
console.log(u.birthday)
export default {}
```

接下来通过一个服务端案例，更深入的理解一下装饰器的意义，代码如下：

```
interface MappingType{
  [props:string]:any
}
interface GlobalThis{
  requestMappings:MappingType
}
let _globalThis:GlobalThis = {
  requestMappings:{}
}
function Controller(url?:string){
  // console.log(url)
  return function(target:any){
    //存储公共URL
    target.prototype.baseURL = url
  }
}
function Get(url?:string){
  // console.log(url)
  return function(target:any, propertyKey: string, descriptor: PropertyDescriptor){
    // 将函数的访问路径保存到全局对象的requestMappings属性中
    if(_globalThis.requestMappings == undefined){
      _globalThis.requestMappings = {}
    }
    _globalThis.requestMappings[propertyKey] = url
```

```
  }
}
// 定义类型
interface UserController{
  [props:string]:any
}
// 定义控制器
@Controller('/user')
class UserController{
  @Get('/hello')
  hello():string{
    return 'hello'
  }
  @Get('/test')
  test():string{
    return 'test'
  }
}
// 定义控制器执行函数
function runController(controller:any,url:string){
  // 拆分2级的url路径
  let urlArr:Array<string> = url.replace('/','').split('/')
  // 获取当前对象的baseURL
  let baseURL = controller.baseURL.replace('/','')
  // 获取当前url的第二级在全局对象中对应的访问路径
  let methodURL = _globalThis.requestMappings[urlArr[1]]
  // 当两级访问路径同时匹配时
  if(urlArr[0] == baseURL && urlArr[1] == methodURL.replace('/','')){
    // 执行控制器函数并返回结果
    return controller[urlArr[1] ]()
  }
}
// 调用执行函数获取结果
let res = runController(new UserController(),'/user/test')
console.log(res)

export default {}
```

3.2.2 模块和命名空间

TypeScript的模块系统与ESM的模块系统在导入导出上几乎没有任何区别，这里具备特色的内容是命名空间的概念。请务必注意一点，TypeScript 1.5里术语名已经发生了变化。“内部模块”现在称做“命名空间”。“外部模块”现在则简称为“模块”，这是为了与 ECMAScript 2015里的术语保持一致，(也就是说 `module X {` 相当于现在推荐的写法 `namespace X {`)。

命名空间用来将代码内部拆解成单独模块，可以将不同功能的函数和代码块按照命名空间划分，代码如下：

```
namespace StringUtils{
  export const reverse = (str:string):string => {
    let str1 = ''
    for(let i = str.length-1;i>=0;i--){
      str1+=str[i]
    }
    return str1
  }
  export const replaceAll = (str:string,replaceStr:string,targetStr:string):string => {
    let reg = new RegExp(replaceStr,'g')
    return str.replace(reg,targetStr)
  }
}
let str = 'abcdefgabcdefg'
str = StringUtils.reverse(str)
str = StringUtils.replaceAll(str,'a','n')
console.log(str)
```

可以将命名空间分散到多个文件，在代码中合并引用，将上面的案例改造成分散文件模式，代码如下：

```
//在src文件夹下创建string-replace.ts文件并编写如下内容
namespace StringUtils{
  export const replaceAll = (str:string,replaceStr:string,targetStr:string):string => {
    let reg = new RegExp(replaceStr,'g')
    return str.replace(reg,targetStr)
  }
}
//在src文件夹下创建namespace.ts文件并编写如下内容
/// <reference path="string-replace.ts" />
namespace StringUtils{
  export const reverse = (str:string):string => {
    let str1 = ''
    for(let i = str.length-1;i>=0;i--){
      str1+=str[i]
    }
    return str1
  }
}
let str = 'abcdefgabcdefg'

str = StringUtils.reverse(str)
str = StringUtils.replaceAll(str,'a','n')
console.log(str)
```

为确保依赖可以完全加载，在运行时采用指定合并文件的方式，在命令行工具中输入，代码如下：

```
tsc --outFile dist/namespace.js src/string-replace.ts src/namespace.ts
```

3.2.3 d.ts的使用方式

在TypeScript的开发中很多场景都需要进行类型定义，不同模块间的代码都存在不同的类型描述。这就意味着需要在.ts文件中使用大量的class或interface来保证类型的支持。这种情况下和容易出现两个模块共用相同类型而导致类型的重复声明，此时d.ts文件便派上了用场。这里通过一个简单的例子来介绍.d.ts文件的使用方式，代码如下：

```
//当src/test.ts中存在如下内容时
interface User{
  name:string,
  age:number,
  birthday:Date,
  [props:string]:any
}
let user:User = {
  name:'小明',
  age:18,
  birthday:new Date(),
  money:100
}
console.log(user)
export default {}
```

当test.ts中包含User类型时，在test1.ts中同样适用User类型时，最简单的方式，代码如下：

```
//src/test1.ts文件中的内容
interface User{
  name:string,
  age:number,
  birthday:Date,
  [props:string]:any
}
let user1:User = {
  name:'小白',
  age:20,
  birthday:new Date(),
  money:100
}
console.log(user1)
export default {}
```

此时会发现存在无用代码的情况，这时很多开发者可能会通过模块加载的方式将类型导出，最终变成如下效果：

```
//src/user.ts
interface User{
  name:string,
  age:number,
  birthday:Date,
```

```
  [props:string]:any
}
export default User

//src/test.ts
import User from './user'
let user:User = {
  name:'小明',
  age:18,
  birthday:new Date(),
  money:100
}
console.log(user)
export default {}

//src/test1.ts
import User from './user'
let user1:User = {
  name:'小白',
  age:20,
  birthday:new Date(),
  money:100
}
console.log(user1)
export default {}
```

完全可以通过文件拆除的形式将类型提取到外部文件，但这种方式在构建JavaScript时会生成无用的空模块，因为在做语言转换时类型会被完全抹去，所以interface中的内容其实最后并不存在。这时就需要使用d.ts文件了。d.ts文件的作用就是用来描述文件类型，类型识别器默认寻找node_modules/@types文件夹来进行全局类型的检索，可以在tsconfig.json文件中配置typeRoots来修改d.ts的存放位置。接下来将typeRoots属性的值设置，代码如下：

```
{
   "typeRoots": ["./types"],
}
```

在项目中创建types文件夹，在其中创建index.d.ts文件，在文件中定义User类型，并将test和test1两个文件的类型引用去掉，代码如下：

```
// types/index.d.ts
interface User{
  name:string,
  age:number,
  birthday:Date,
  [props:string]:any
}

// src/test.ts
```

```
let user:User = {
  name:'小明',
  age:18,
  birthday:new Date(),
  money:100
}
console.log(user)
export default {}

// src/test1.ts
let user1:User = {
  name:'小白',
  age:20,
  birthday:new Date(),
  money:100
}
console.log(user1)
export default {}
```

4.总结

通过对TypeScript语言的简单了解，需要明确TypeScript语言在实际应用开发中的使用意义和优势，TypeScript并不是项目开发中必须使用的语言，他更多的应用于服务端开发、游戏开发、插件开发等场景。在前端应用构建和业务开发场景中，TypeScript往往是与JavaScript语法混合使用，并不是所有部分都需要做类型定义和类型标注，并不是TypeScript中的每一个特性都必须应用在项目开发中，所以在使用这门语言时一定要分析使用场景选择最合适的部分应用，不要盲从。