TypeScript

声数元枚任空断常解

接可只额函索类扩继

类构三存抽型

函泛推符迭解鸡（j）混 \*

兼对函参重类泛

高交联别字

模作执出入支生他

命外引同别他

合接命 模扩全扩

基础类型：

let 变量名:类型=值

值可以是模板字符串。

类型为数组时可用：基本类型[]或Array<基本类型>

元组Tuple：[类型1,类型2,……]

枚举：enum Color {Red = 6 , Green , Blue};

let c:Color = Color.Green;

都不赋值默认从0开始，赋值后依次加1，可手动赋不一样的值

任意值：让不清楚类型的变量通过编译阶段的检查，用any（Object类型的变量只允许赋任意值，不允许调用任意方法）

空值：void，当变量用void时只能赋予undefined和null

undefined和null，当指定--strictNullChecks时，undefined和nul只能赋给void和它们各自，当未指定时，还可以赋给别的任何变量

never：返回never的函数存在无法达到的终点

类型断言：把any转化为已知<类型>值或值 as 类型（当在TypeScript中使用jsx时，只允许这种）

变量声明

与es6中let 和 const 命令一致

解构数组：

与es6中数组的解构赋值一致

解构对象：

与es6中对象的解构赋值一致

接口：通过结构性子类型检查是否满足其描述，类型检查器不检查属性的顺序，只检查属性是否存在及属性类型是否正确

1 可选属性：在属性名和 : 之间加 ?

2 只读属性：在属性名前加readonly。ReadonlyArray<T>和Array<T>相似，前者把所有可变方法去掉

let a : number[] = [1 , 2 , 3]

let b : ReadonlyArray<number> = a

a = b ; 错

a = b as number[] ; 对

变量用const，属性用readonly

3 额外的属性检查：当对象字面量中属性存在而接口中该属性不存在，且直接赋值会报错。解决（1）用变量中转（2）将对象字面量类型断言（3）接口使用索引签名

4 函数类型：(参数名实际可与它不一样:参数类型):返回类型

构造函数类型：new (参数名实际可与它不一样:参数类型):返回类型

5 索引类型：[参数名无实际意义:索引类型]:返回类型

（1）若接口中既有number索引类型又有string索引类型，则number的返回值应为string返回值的子类型。因为number会转成string去索引

（2）obj.property和obj[”property”]一致

6 类类型：类包含静态部分（中有构造函数）和实例部分，当类implements类类型的接口后只对其实例部分进行类型检查

7 扩展接口：通过extends继承一个或以 , 分割的多个接口

8 接口继承类：继承类的成员但不包括其实现。当接口继承了类中包含private或protect时，这个接口只能被这个类或其子类所实现

类

（1）构造函数：constructor(){}

调用父类构造函数：super();

1. 默认为public
2. private：只能在声明它的类内访问
3. protect：在声明它的类及派生类内访问
4. 当类中含有private或protect时，与它能兼容的类必须也含有这个private或protect，且双方来自同一个private或protect
5. 存取器：set get
6. 抽象类：抽象类和抽象方法前有abstract，抽象方法和接口方法都不包含具体实现，而抽象方法必须包含abstract关键字且可以包含访问修饰符
7. :typeof 类名(是类类型，拥有静态部分)=类名;

: 类名(是实例类型，拥有实例部分)=new 类名();

函数

（1）当没有可选参数时，传入函数的参数个数必须与函数期望的参数个数一致

（2）当有可选参数时，可选参数必须跟在必须参数后面

（3）所有必须参数后的默认参数是可选的，这部分的函数签名用可选参数形式。所有必须参数前的默认参数必须传入undefined来获得默认值

（4）剩余参数：…参数名:参数类型[]

（5）this：

()=>{} 会在函数创建的时候指定this值，而不是在函数调用的时候。

若this在对象字面量里，则类型为any，可用this:对象字面量名为

参数传入对象字面量方法中指出this为对象字面量名的类型

（6）重载：

function 函数名(参数名:参数类型1):返回值

function 同上函数名(同上参数名:参数类型2):返回值

function 同上函数名(同上参数名):any{

//处理所有重载

}

泛型：在声明名称后或签名前使用<T>，可在内部用T代替任何唯一类型。多个泛型用,分离。<>内的泛型可被约束，如使用extends，之后内部使用该泛型时均有相应约束

使用泛型时给<T>传入希望成为的类型，或没有<T>使用类型推论

泛型只能用在类的实例部分，不能用在类的静态部分，不能用于枚举，不能用于命名空间

类型推论：当类型没有明确指出时，会使用类型推论

1. 最佳通用类型：从所有候选类型中找出能描述所有候选类型的类型作为最佳通用类型
2. 上下文类型

类型兼容性：基于结构性子类型，源=目标

1. 对象字面量：源中除有0个或多个与目标完全相同的属性名和属性类型外无他（源是目标的子类型）
2. 函数：目标的参数除有0个或多个与源完全相同的属性类型外无他，且，源的返回类型是目标的返回类型的子类型。
3. 函数参数双向协变：只有当源函数参数能够赋值给目标函数或者反过来时才能赋值成功
4. 函数重载：源函数的每个重载都要在目标函数上找到对应的函数签名
5. 类：

对于：类名 比较的是类的实例部分

1. 泛型：

比较的是：T赋具体类型后的部分，没赋具体类型T被认为是any

高级类型

1. 交叉类型：用&连接，拥有连接类型的所有特性
2. 联合类型：用|连接，若没有typeof或instanceof类型保护，则只能访问连接类型的公有特性

若if(typeof 参数名 === ”number”或”string”或”boolean”或”symbol”){} 若为真则在该作用域内细化为相应类型

若if(参数名 instanceof 类名){} 若为真则在该作用域内细化为相应类型

1. 类型别名：type 类型别名 = 类型 。类型别名可以是泛型，可以在属性里引用自己，不能出现在声明语句的右侧
2. 接口和类型别名区别：

接口创建了新的名字，错误信息使用的是接口名，类型别名不创建新的名字，错误信息不使用接口名

类型别名不能extends和implements，也不能被extends和implements，接口可以

接口无法描述联合类型和元组类型，类型别名可以

1. 字符串字面量类型 type a = ”b” |”c”|”d”;

e:a

if(e === ”b”){}

字符串字面量可用于区分函数重载：

function A(B:”c”):string

function A(B:”d”):string

function A(B:string):string

Symbol

与es6中Symbol一致

迭代器和生成器

可迭代性：当一个对象实现了Symbol.iterator属性时，认为是可迭代的，如Array。

for…in迭代键 for…of迭代值

for…in可操作任何对象 for…of只操作迭代对象

模块：“内部模块”现在叫“命名空间”。“外部模块”现在叫“模块”

1. 模块作用域外对该模块不可见，要通过export，import访问
2. 执行模块代码前，模块加载器会查找并执行该模块所有依赖。服务于node.js的模块加载器是CommonJS，服务于web应用的模块加载器是AMD
3. TypeScript和ECMAScript2015导出

export 任何声明定义

export {任何声明名称} 后加from ”文件名”，重心导出，

export {任何声明名称 as 别名} 或export \* from ”文件名”，重

新导出

默认导出（只能有一个） export default 任何声明定义或定义

1. TypeScript和ECMAScript2015导入

import {导出的声明名称} from ”文件名”

import {导出的声明名称 as 别名} from ”文件名”

import \* as 别名 from ”文件名”

一些设置全局状态的模块 import ”文件名”

默认导入 import 默认导出的声明名称或除导出的声明名称外的任何名称 from ”文件名”

1. 支持CommonJS和AMD：两者都有exports对象，包含一个模块所有导出内容。TypeScript模块通过export = 定义模块的导出对象来支持CommonJS和AMD

导出 export = 类名/接口名/命名空间名/函数名/枚举名

导入 import 导出名或别名 = require(”文件名”);

1. 生成模块代码：根据编译时指定的模块目标参数，编译器生成相应模块代码

Node.js(commonjs)

Require.js(amd)

使用其他javascript库：描述非TypeScript库，需要声明其API。

通常在.d.ts文件中（模块加载器加载）

declare module ”模块名”{

//导出

}

简写（所有导出类型将是any）

declare module ”模块名”;

在需要的文件中：

///<reference path = ”文件名.d.ts” />

//导入

模块里不要使用命名空间

命名空间

（1）namespace 命名空间名{

export 任何声明定义;

}

1. 命名空间外只能访问export的内容
2. ///<reference path = ”依赖的命名空间文件” /> 引入依赖的命名空间
3. 引入依赖的命名空间文件后，相同命名空间名内可直接使用
4. 别名：可为任意标识符创建别名，生成不同的引用，即改变别名的值不会影响原始的值

import 别名 =

使用其他javascript库：描述非TypeScript库，需要声明其API。

通常在.d.ts文件中（通过<script>标签加载）

declare namespace 命名空间{

//导出

}

模块解析

1. 相对导入是以/或./或../开头的，此外都是非相对导入
2. 有两种模块解析策略 Node和Classic ，可用--moduleResolution指定使用哪个，默认为Node
3. Classic的相对导入找名.ts 名.d.ts
4. Classic的非相对导入找名.ts 名.d.ts

上个目录的…

1. Node的相对导入找名.ts 名.tsx 名.d.ts

名/package.json（如果指定了”typing”属性）

名/index.ts 名/index.tsx 名/index.d.ts

1. Node的非相对导入找名.ts 名.tsx 名.d.ts

名/package.json（如果指定了”typing”属性）

名/index.ts 名/index.tsx 名/index.d.ts

上个目录的…

1. node.js的模块解析

相对导入找名.js

名/package.json（如果指定了”main”属性）

名/index.js

非相对导入找名.js

名/package.json（如果指定了”main”属性）

名/index.js

上个目录的…

声明合并

1. 合并同名接口：非函数成员必须唯一。函数成员若不唯一则当成是重载，接口内顺序不变，下面的接口靠上。若函数签名有一个参数的类型是单一的字符串字面量，则最靠上
2. 合并同名命名空间

（a）导出的同名接口合并

（b）下面命名空间导出成员添加到上面命名空间内

（c）合并后，从其他命名空间合并进来的成员无法访问非导出成员

1. 模块扩展：扩展文件中有

import {要扩展的类名} from ”要扩展的类名所在文件名”

declare module ”要扩展的类名所在文件名” {

interface 要扩展的类名{

//扩展方法签名

}

}

要扩展的类名.prototype.扩展方法名 = function(){

}

使用文件中有：

import {要扩展的类名} from ”要扩展的类名所在文件名”

import ”扩展文件名”;

1. 全局扩展

declare global{

interface ”要扩展的类名”{

//扩展方法签名

}

}

要扩展的类名.prototype.扩展方法名=function(){

}

jsx在TypeScript中使用

1. 文件扩展名为.tsx
2. TypeScript有两种jsx模式，preserve保留jsx供后续的转换使用（如Babel），输出.jsx文件。react生成React.creatElement，输出.js 文件。可通过--jsx或tsconfig.json指定模式
3. .tsx文件中禁止用<>类型断言，用as类型断言
4. 类型检查

（a）固有元素：若没有以下接口则全部通过。

declare namespace JSX{

interface IntrinsicElement{

存在的标签名:{存在的属性名:存在的属性类型}

}

}

（b）基于值的元素：工厂函数的类类型为这个工厂函数名，实例类型为这

个工厂函数返回值类型

declare namespace JSX{

interface ElementClass{ //默认为{}

//标签名实例类型中0,1或多个属性名和类型

}

interface ElementAttributesProperty{

//标签名实例类型中指定用来作为标签属性的属性名

}

}

1. jsx结果类型：默认是any

Mixins

class A {

a1:boolean;

a2(){this.a1=true;}

}

class B {

b1:boolean;

b2(){this.b1=false;}

}

class C implements A ,B {

c1:boolean;

c2(){this.c1=true}

a1:boolean = false;

a2:()=>void;

b1:boolean=false;

b2:()=>void;

}

function mixins(derived:any,bases:any[]){

bases.forEach(base=>{

object.getOwnPropertyNames(base.prototype).forEach(name=>{

derived.prototype[name]=base.prototype[name];

});

});

}