

TypeScript培训内容列表

- 为什么要使用TypeScript;
- TypeScript中的类型、变量、函数;
- 接口、类和继承;
- 装饰器、泛型和异步编程;
- 声明文件的使用和编写;
- 使用第三方JS库;
- 测试驱动开发模式;
- 测试驱动开发框架;
- 模块化;
- 面向对象编程;
- 依赖注入;
- 实战;

TypeScript是什么

- ▶ JS的超集，让JS具有强类型的“语法糖”，其实就是为变量、函数参数、函数返回值增加上类型
- ▶ 带来的好处：
 - 编码时候的类型检查
 - IDE的智能提醒
 - 面向对象开发模式的应用
- ▶ TypeScript编译器判断一个类型的基本规则是在代码中使用TS的基础

基本类型

- ▶ 在了解TS的强类型之前，先看一下JS的动态类型，以及动态类型可能引起的错误

```
function doCalculation(a, b, c) {  
    return (a * b) + c;  
  
}  
  
let result = doCalculation(2, 3, 1);  
console.log('doCalculation(): ' + result);
```

\$ **doCalculation()**: 7

基本类型

- ▶ 如果我们调用函数的时候，是这样调用的：

```
function doCalculation(a, b, c) {  
    return (a * b) + c;  
  
}  
  
let result = doCalculation("2", "3", "1");  
console.log('doCalculation(): ' + result);
```

\$ **doCalculation()**: 61 ?

基本类型

- TypeScript的强类型可以避免我们出现上面那样的问题，那么在TS中该怎么写类型？

```
function doCalculation(a: number, b: number, c: number) {  
    return (a * b) + c;  
  
}  
  
let result = doCalculation("2", "3", "1");  
console.log('doCalculation(): ' + result);
```

\$ error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.

基本类型

- ▶ 举几个例子：

```
let myString: string;  
  
let myNumber: number;  
  
let myBoolean: boolean;  
  
myString = myNumber;  
  
myBoolean = myString;
```

```
$ error TS2322: Build: Type 'number' is not assignable to type 'string'  
$ error TS2322: Build: Type 'string' is not assignable to type 'boolean'
```

基本类型

- ▶ TS的强类型意味着“ = ” 分配操作符左边和右边的类型必须保持一致
- ▶ 修复上面代码的错误，我们需要这样：

```
let myString: string;  
  
let myNumber: number;  
  
let myBoolean: boolean;  
  
myString = myNumber.toString();  
  
myBoolean = (myString === “test”);
```

Ander Heilsberg描述TS是JS的“语法糖”，通过这种语法糖可以使JS成为强类型，一旦我们打破类型规则，编辑器就会生成错误信息，从而保护我们的代码强壮性

类型推断

- ▶ TS也可以进行类型推断来决定一个变量的类型；也就是说，TS可以根据变量第一次被分配值的类型，来决定这个变量以后在代码中的类型。

```
let inferredString = "This is a string";  
let inferredNumber = 1;  
inferredString = inferredNumber;
```

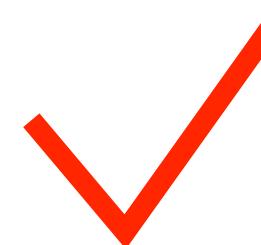
\$ error TS2011: Build: Cannot convert 'string' to 'number'

- 我们在代码中没有用`:`来告诉TS变量的类型。TS自动推断出了变量的类型。

鸭式辨形

- ▶ TS会使用鸭式辨形来推断复杂变量的类型

```
let complexType = { name: "jinbin", id: 1};  
complexType = { id: 2, name: "test"};
```



```
let complexType = { name: "jinbin", id: 1};  
complexType = { id: 2};
```



\$ error TS2322: Type '{ id: number }' is not assignable to type '{ name: string, id: number }'.
Property 'name' is missing in type '{ id: number; }'

模板字符串

- ▶ ES6 模板字符串

```
let myVariable = "test";  
console.log("") + myVariable);
```

```
let myVariable = "test";  
console.log(`myVariable=${myVariable}`);
```

数组和枚举

- ▶ TS中的数组： 和JS中的差不多， 只是增加了强类型， 例如：

```
let arrayofNumbers: number [] = [1, 2, 3];  
  
arrayofNumbers = [3, 4, 5, 6, 7, 8];  
  
console.log(`arrayofNumbers: ${arrayofNumbers}`);  
  
arrayofNumbers = ['1', '2', '3'];
```

\$ error TS2322: Type 'string[]' is not assignable to type 'number[]'.
Type 'string' is not assignable to type 'number'.

for...in 和 for...of

- ▶ TypeScript引入了for...in来简化数组的迭代。

```
let arrayOfStrings: string[] = ["first", "second", "third"];  
for(let itemKey in arrayOfStrings) {  
    let itemValue = arrayOfStrings[itemKey];  
    console.log(`arrayOfStrings[${itemKey}] = ${itemValue}`);  
}
```

- 我们在代码中利用for...in和变量itemKey简化了对数组的迭代。
注意：itemKey是数组中元素的索引，不是元素本身

for...in 和 for...of

- 如果我们不关心数组元素的索引，只关心数组元素的值，那么可以使用for ... of

```
let arrayOfStrings: string[] = ["first", "second", "third"];
for(let arrayItem of arrayOfStrings) {
    console.log(`arrayItem = ${arrayItem}`);
}
```

- 注意：此时的变量arrayItem是数组中元素本身

any 类型

- ▶ TypeScript引入的any类型，本质上是解除了编译器的严格类型检查。

```
let item1: any = { id: 1, name: "item1" };  
item1 = {name: "item2"};
```

- 如果没有在item1上增加any类型，那么就像前面说的一样，TS编译器就会报错。
- 注意：不能滥用any类型，滥用any类型会导致失去使用TS的严格类型检查的意义

显示转换

- ▶ 使用强类型语言的时候，通常都会遇到需要明确指定一个对象的类型的情况。
 - 对象的类型后面会详细讲解，在此之前先了解一下显示转换
 - TS中一个对象可以通过使用<>语法，显示转换这个对象的类型，例如：

```
let item1 = <any>{ id: 1, name: "item1" };  
item1 = {name: "item2"};
```

- 代码中，我们通过<any>显示的告诉TS编译器，{id:1, name: "item1"}这个对象是any类型的。因此，item1变量也就是any类型的。
- 再次提醒any类型是TS为了与JS兼容提供的一种手段，我们应尽量限制any类型的使用。

S.F.I.A.T

- ▶ any类型是TS为了与JS兼容提供的一种手段，我们应尽量限制any类型的使用。
 - 应该尽量找出你正在使用的对象的正确类型，用这个类型来代替使用any的地方

Simply Find an Interface for Any Type

- Interface接口是TS中用来定义自定义类型的一种方法，我们后面会详细讲解

需要时刻记住的是：尽最大努力定义我们使用对象的类型，编写强类型代码，会保护我们免于遭受将来代码的错误和BUG的困扰！

枚举

- ▶ 枚举类型借鉴与其他强类型语言：C#,JAVA，为解决特殊数字识别提供一种方案
 - 枚举把特殊数字与人类容易识别的名称关联起来。

```
enum DoorState {  
    Open,  
    Closed,  
    Ajar  
}
```

- 代码中定义了一个枚举类型对象DoorState代表门的三个状态，Open,Closed,Ajar
- 在底层（生成的JS中），TS会给每个状态分配一个数字：Open 0, Closed 1, Ajar 2

使用枚举

- ◎ 代码验证一下：

```
let openDoor = DoorState.Open;  
console.log(`openDoor is: ${openDoor}`);
```

\$ **openDoor is : 0**

```
let closeDoor = DoorState["Closed"];  
console.log(`closeDoor is ${closeDoor}`);
```

\$ **closeDoor is : 1**

```
let ajarDoor = DoorState[2];  
console.log(`ajarDoor is ${ajarDoor}`);
```

\$ **ajarDoor is : Ajar**

枚举的意义

- 生成的JS文件运行的时候，我们看一下DoorState对象的内容：

```
DoorState
{...}
[prototype]: {...}
[0]:Open
[1]:Closed
[2]:Ajar
[prototype]: []
Ajar:2
Closed:1
Open:0
```

- 通过最后生成的对象，我们可以看到枚举可以让我们方便的把很难记住的数字起了一个容易记住的名字。让我们的代码更加清晰易读。

常量枚举

- 枚举类型的一个变形是常量枚举，看一下代码：

```
const enum DoorStateConst {  
    Open,  
    Closed,  
    Ajar  
}  
  
let constDoorOpen = DoorStateConst.Open;  
console.log(`constDoorOpen is:${constDoorOpen}`);
```

\$ **constDoorOpen** is: 0

- 常量枚举主要是为了提升性能，生成的JS不再包含闭包。而是直接把 DoorStateConst.Open 与 0 对应起来。因此，我们也没办法像下面这样访问：

```
console.log(` ${DoorStateConst[0]} `);
```

常量值

- TS 允许我们用关键字const 把一个变量声明为常量
- 如果一个常量标记为const，那么只能在此变量被定义的时候，才能设置这个变量的值。

```
const constValue = "test";  
  
constValue = "updated";
```

\$ error TS2450: Left-hand side of assignment expression cannot
be a constant or a readonly property.

let关键字

- ▶ JS中通过var声明变量，这种方式允许我们在没有声明一个变量的时候，就使用这个变量，容易引起误解
- ▶ TypeScript引入let关键字来代替var定义变量，好处是：
 - 变量在使用前，必须已经定义好了。
 - let定义的变量具备块级作用域：意思是变量的定义和值限制在定义这个变量的代码块中，举例如下：

let关键字

```
let lValue = 2;
console.log(`lValue = ${lValue}`);

if (lValue === 2) {
  let lValue = 2001;
  console.log(`block scoped lValue: = ${lValue}`);
}

console.log(`lValue = ${lValue}`);
```

\$ lValue = 2

\$ block scoped lValue: = 2001

\$ lValue = 2

函数返回值

- ▶ 目前为止，我们讨论了针对变量的强类型问题。TypeScript针对函数也有一些强类型的规则。
- ▶ 函数返回值的类型：意思是当我们调用一个函数，函数执行后返回一个值，我们也可以对这个返回值进行强类型的约束。

```
function addNumbers(a: number, b: number): string {  
    return a + b;  
}  
let addResult = addNumber(2,3);  
  
console.log(`addNumbers returned: ${ addResult }`);
```

函数返回值

\$ error TS2322: Type 'number' is not assignable to type 'string'.

- 错误提示我们：函数运行后返回的是number类型，而我们在定义函数的时候，期望函数返回的是string，因此TypeScript 编译器给出了错误提示。

```
function addNumbers(a: number, b: number): string {  
    return (a + b).toString();  
}  
let addResult = addNumber(2,3);  
  
console.log(`addNumbers returned: ${ addResult }`);
```

\$ addNumbers returned: 5

匿名函数

- TypeScript和JS一样也支持匿名函数，只是需要为参数和返回值指明类型

```
let addFunction = function(a:number, b:number): number {  
    return a + b;  
}  
  
let addFunctionResult = addFunction(2, 3);  
  
console.log(`addFunctionResult: ${ addFunctionResult }`);
```

\$ **addFunctionResult: 5**

可选参数

- 在JS中，我们调用一个函数的时候，如果函数需要参数，而我们调用的时候，没有提供参数，那么在函数中这个参数的值就是：undefined

```
let concatString = function(a, b, c) {  
    return a + b + c;  
}  
  
let concatAbc = concatString("a", "b", "c");  
console.log("concatAbc:" + concatAbc);  
  
let concatAb = concatString("a", "b");  
console.log("concatAb:" + concatAb);
```

\$ concatAbc: abc

\$ concatAb: abundefined

可选参数

- TypeScript引入?来表示可选参数。注意：可选参数需要在函数参数的最后面的位置

```
function concatStrings(a: string, b:string, c?:string) {  
    return a + b + c;  
}  
  
let concat3strings = concatStrings("a", "b", "c");  
console.log(`concat3strings: ${concat3strings}`);  
let concat2strings = concatStrings("a", "b");  
console.log(`concat2strings: ${concat2strings}`);  
let concat1string = concatStrings("a");
```

\$ error TS2081: Build: Supplied parameters do not match any
signature of call target.

默认参数

- 可选参数的一个细微变化是默认参数，如果一个可选参数没有提供，我们可以为参数提供一个默认值。注意：去掉了？

```
function concatStringsDefault(  
    a: string,  
    b: string,  
    c: string = "c"  
) {  
    return a + b + c;  
}  
  
let defaultConcat = concatStringsDefault("a", "b");  
console.log(`defaultConcat: ${defaultConcat}`);
```

\$ defaultConcat: abc

rest剩余参数

- 在JS中，每个函数都有一个特殊的内部属性： arguments。用于？？

```
function testArguments() {  
    if (arguments.length > 0) {  
        for(let i=0; i<arguments.length; i++) {  
            console.log(`argument[${i}] = ${arguments[i]}`);  
        }  
    }  
}  
testArguments(1,2,3);  
testArguments("firstArg");
```

rest剩余参数

```
$ argument[ 0 ] = 1
```

```
$ argument[ 1 ] = 2
```

```
$ argument[ 2 ] = 3
```

```
$ argument[ 0 ] = firstArg
```

- ▶ 为了在TypeScript中也表示类似的功能，在TypeScript引入了(Rest)剩余参数。剩余参数的语法就是三个点 ...
 - ◎ rest语法告诉TypeScript编译器，这个函数可以接受任意数量的参数

rest剩余参数

- 上面的JS函数，用TypeScript的剩余参数语法写出来如下：

```
function testArguments( ... argArray: number [] ) {  
    if (argArray.length > 0) {  
        for(let i=0; i<argArray.length; i++) {  
            console.log(`argArray[${i}]= ${argArray[i]}`);  
            // Use JS argument  
            console.log(`argument[${i}]= ${arguments[i]}`);  
        }  
    }  
}  
testArguments(9);  
testArguments(1,2,3);
```

rest剩余参数

- 使用`... argArray : number []`语法作为我们函数的参数，会告诉TypeScript编译器，函数可以接受任意数量的number类型的参数。输出如下：

```
$ argArray[ 0 ] = 9  
$ argument[ 0 ] = 9  
  
$ argArray[ 0 ] = 1  
$ argument[ 0 ] = 1  
  
$ argArray[ 1 ] = 2  
$ argument[ 1 ] = 2  
  
$ argArray[ 2 ] = 3  
$ argument[ 2 ] = 3
```

注意：使用`... argArray`与`argument`之间的区别是，`argArray`具备类型推导的能力：

TypeScript会把`argArray`看成`number`类型的数组，而`argument`没有类型推导，所以是`any`类型的数组

回调函数

- ▶ JS回调函数功能很强，NODE的本质也是基于回调函数

```
let callbackFunction = function(text) {  
    console.log('inside callbackFunction' + text);  
}  
  
function doSomethingWithCallback(initialText, callback) {  
    console.log('inside doSomethingWithCallback' + initialText);  
    callback(initialText);  
}  
  
doSomethingWithCallback('myText', callbackFunction);
```

\$ **inside doSomethingWithCallback myText**

\$ **inside callbackFunction myText**

回调函数

- ▶ 但是，如果我们传给doSomethingWithCallback函数的参数不是一个函数时，会如何？

```
doSomethingWithCallback('myText', 'anotherText');
```

\$ TypeError: callback is not a Function

- 因此，JS程序员需要在用回调函数的时候需要非常小心。首先，需要保证传入的参数是一个函数，其次，需要阅读API文档弄清楚每个参数都是什么。
- 如果，我们在编码的时候，就可以被告知参数都是什么类型，并且如果没有正确的传入需要函数为参数的情况下，警告编程者，就会提前避免很多错误。

函数签名

- TypeScript的强类型不仅用在变量上，也可以用在回调函数上。使用箭头函数() => {}

```
function callbackFunction(text: string) {  
    console.log(`inside callbackFunction ${text}`);  
}  
  
function doSomethingWithACallback(  
    initialText: string,  
    callback: (initialText: string) => void  
) {  
    console.log(`inside doSomethingWithACallback ${initialText}`);  
    callback(initialText);  
}
```

函数签名

```
callback: (initialText: string) => void
```

- 这里的回调函数callback，通过： 声明了必须是一个函数类型。同时，这个函数的参数也必须是 string
- void 是一个关键字，表明这个函数没有返回值
- 这时，如果我们没有给回调函数传入正确类型：

```
doSomethingWithACallback("myText", "This is not a function")
```

```
$ error TS2345: Argument of type 'string' is not assignable to  
parameter of type '( initialText: string ) => void'
```

函数签名

- 即使我们传递给callback一个函数，如果参数类型不对，也是不行的，例如：

```
function callbackFunctionWithNumber(arg1: number) {  
    console.log(`inside callbackFunctionWithNumber ${arg1}`);  
}  
  
doSomethingWithACallback("myText", callbackFunctionWithNumber)
```

\$ error TS2345: Argument of type '(arg1: number) => void' is not assignable to parameter of type '(initialText: string) => void'

注意：函数签名中的参数名称可以不一样，只要参数的个数，类型以及函数返回值类型一致就可以

函数签名

- ▶ 函数签名是TypeScript的非常强大的特征之一。尤其是当我们需要用到第三方JS代码库的时候，我们需要知道并定义第三方JS代码库提供的函数的签名。这些对第三个JS代码库的函数签名进行描述的文件，我们称之为：类型描述文件(或者声明文件)，以.d.ts扩展名来表示。

函数重载

- ▶ 因为JS是非常动态的语言，所以经常用不同类型的参数调用同一个函数，例如：

```
function add(x, y) {  
    return x + y;  
}  
  
console.log('add(1,1) = ' + add(1,1));  
console.log('add("1", "1") =' + add("1","1"));
```

\$ **add(1,1) = 2**

\$ **add("1", "1") = 11**

函数重载

- 为了保持JS的这种能力，TypeScript引入了函数重载，如果我们想在TypeScript实现上面JS的功能，需要这样：

```
function add(a: string, b: string): string;
function add(a: number, b: number): number;
function add(a: any, b: any): any {
    return a + b;
}
console.log(`add(1,1)=$\{add(1,1)\}`);
console.log(`add("1","1")=$\{add("1","1")\}`);
```

```
$ add(1,1) = 2
$ add("1", "1") = 11
```

函数重载

- ▶ 上面的代码有三点很有意思：
 - 1.前面两行代码中，没有函数体
 - 2.后面的函数声明代码有函数体，但是使用了any。为了实现函数重载，在TypeScript中必须要遵从这种模式。
 - 3.即使使用了any类型，在函数重载的时候，也只能使用前面两个规定好的参数类型，例如： `console.log(`add(true, false)= ${true, false}`);`
\$ error TS2345: Argument of type 'boolean' is not assignable to parameter of type 'number'

高级类型：联合类型

- TypeScript允许我们使用联合类型来表示两个或者多个类型的联合，使用 | 符号

```
let unionType: string | number;  
unionType = 1;  
console.log(`unionType : ${unionType}`);  
unionType = “test”;  
console.log(`unionType: ${unionType}`);
```

\$ unionType: 1

\$ unionType: test

高级类型：类型卫士

- 当使用联合类型的时候，TypeScript编译器依然会保证强类型的检查，例如：

```
function addWithUnion(  
    arg1: string | number,  
    arg2: string | number  
) {  
    return arg1 + arg2  
}
```

\$ error TS2365: Operator '+' cannot be applied to types 'string | number' and 'string | number'

- 这时候就需要有类型卫士出现了！

高级类型：类型卫士

- ▶ 类型卫士就是一个表达式，检查我们的类型，并保证类型在规定范围内。下面代码中的两个if语句就是类型卫士

高级类型：类型卫士

```
function addWithTypeGuard(  
    arg1: string | number,  
    arg2: string | number  
): string | number {  
    if (typeof arg1 === “string”) {  
        console.log(‘first argument is a string’);  
        return arg1 + arg2;  
    }  
    if (typeof arg1 === “number” && typeof arg2 === “number”) {  
        console.log(‘both arguments are numbers’);  
        return arg1 + arg2;  
    }  
    console.log(‘default return’);  
    return arg1.toString() + arg2.toString();  
}
```

高级类型：类型卫士

- › 让我们来看一下，用不同的类型参数来调用函数的结果：

```
console.log(`addWithTypeGuard(1, 2)= ${addWithTypeGuard(1,2)}`);
```

\$ both arguments are numbers

\$ addWithTypeGuard(1, 2) = 3

```
console.log(`addWithTypeGuard("1", "2")= ${addWithTypeGuard("1","2")}`);
```

\$ first argument is a string

\$ addWithTypeGuard("1", "2") = 12

```
console.log(`addWithTypeGuard(1, "2")= ${addWithTypeGuard(1,"2")}`);
```

\$ default return

\$ addWithTypeGuard(1, "2") = 12

高级类型：类型别名

- 当使用联合类型的时候，有时候记住联合类型里包含的类型比较费劲，因此 TypeScript引入类型别名，允许为联合类型起一个好记的名字。使用type关键字

```
type StringOrNumber = string | number;

function addWithAlias(
  arg1: StringOrNumber,
  arg2: StringOrNumber
) {
  return arg1.toString() + arg2.toString();
}
```

- 我们为联合类型定义了一个别名：StringOrNumber，然后在函数中使用了这个类型别名。因此，函数参数 arg1, arg2既可以是字符串类型也可以是number类型

高级类型：类型别名

- ▶ 类型别名也可以用于函数，比如：

```
type CallbackWithString = (string) => void;

function usingCallbackWithString (callback: CallbackWithString) {
  callback("This is a string");
}
```

- 代码中，我们定义了一个类型别名，表示这个类型是一个无返回值，接收一个字符串类型参数的，在下面的函数定义中，直接用类型别名：CallbackWithString来规定，函数定义中参数的类型
- 类型别名让我们在使用联合类型的时候，让我的代码更加清晰、直观

高级类型：Null和undefined

- ▶ 在JS中一个变量被声明，但没有被赋值，这时候访问这个变量的值就是：undefined
- ▶ 在JS中还有一个关键字null
 - null 和 undefined的区别：null表示一个对象不存在，即“没有对象”；undefined表示基本类型或对象的值没被定义，也就“缺少值”！
 - 看一下在JS中的例子：

```
function testUndef(test) {  
    console.log('test parameter:' + test);  
}  
testUndef();  
testUndef(null);
```

\$ test parameter: undefined

\$ test parameter: null

高级类型：Null和undefined

- ▶ 在TS中，也有null 和 undefined关键字
 - 看一下在TS中的例子：

```
function testUndef(test: null | number) {  
    console.log('test parameter:' + test);  
}  
testUndef();
```

- 代码中我们为参数test， 规定了类型 null | number
 - 最后一行调用： testUndef() 没有传入任何参数
- \$ error TS2346: Supplied parameters do not match any signature of call target**
- ▶ TS编译器在编译阶段就保护了我们的代码， 要调用testUndef函数不允许什么参数都不传入。

高级类型：Null和undefined

- 类似的，我们也可以定义一个变量允许具有undefined类型的值
 - 看一下在TS中的例子：

```
let x: number | undefined  
x = 1;  
x = undefined;  
x = null
```

\$ error TS2322: Type 'null' is not assigned to type 'number | undefined'

- 代码中，我们定义x只能接受number类型或者undefined类型的值。因此，最后一行我们赋值null的时候，TS编译器报错保护了我们的代码，告诉我们不能给x赋值为null

高级类型：对象rest和spread

- 当我们处理一个对象的时候，经常会遇到copy一个对象的属性到另外一个对象中的情况，或者从几个对象抽出一部分属性组成新的对象。
 - 为了满足这类需求，TS使用了ES7中增加的一些语法，称为：对象rest和spread

```
let firstObj = {id: 1, name: "firstObj"};  
  
let secondObj = {...firstObj};  
console.log(`secondObj.id: ${secondObj.id}`);  
console.log(`secondObj.name: ${secondObj.name}`);
```

\$ **secondObj.id: 1**

\$ **secondObj.name: firstObj**

高级类型：对象rest和spread

- 我们可以使用rest语法把多个对象组合在一起：

```
let name0bj = {name: 'name0bj'};  
let id0bj = {id: 2};  
  
let obj3 = {...name0bj, ...id0bj};  
console.log(`obj3.id: ${obj3.id}`);  
console.log(`obj3.name: ${obj3.name}`);
```

\$ **obj3.id: 2**

\$ **obj3.name: nameObj**

注意：如果两个对象有相同的属性名，那么后面的对象属性会覆盖前面对象的相同属性

以上都是基础：理解编译器规则

- 基本类型和类型语法
- 类型推断和鸭式辨形
- 模板字符串
- 数组
- 使用for...in 和 forof
- 使用any 和 显示转换
- 枚举
- 常量枚举和值枚举
- let关键字
- 函数和匿名函数
- 可选的函数参数和默认函数参数
- Rest参数
- 函数回调，函数签名，函数重载
- 联合类型，类型卫士，类型别名
- Null 和 undefined

TypeScript培训内容列表

- 为什么要使用TypeScript;
- TypeScript中的类型、变量、函数;
- 接口、类和继承;
- 装饰器、泛型和异步编程;
- 声明文件的使用和编写;
- 使用第三方JS库;
- 测试驱动开发模式;
- 测试驱动开发框架;
- 模块化;
- 面向对象编程;
- 依赖注入;
- 实战;

接口、类和继承

- ▶ 我们已经学习了TS使用基本类型、推断类型和函数签名来为开发带来强类型的体验
- ▶ TS同样带来了面向对象编程的特性：接口、类和继承
- ▶ 这些面向对象的概念在前端开发环境中，都是在ES6及更新的ES标准中提出的
- ▶ TS编译器会把我们写出来的具有这些高级特性的代码编译成为ES3,ES5的代码
- ▶ 因此，使用TS可以让我们提前使用新ES标准中提出的各种高级语言特征
- ▶ 我们这部分讲解分为两部分：
 - 接口、类、继承的概念和基本使用；
 - 如何使用工厂设计模式来创建和使用接口、类、继承

接口概念

- ▶ 接口来定义一个对象必须具有的属性和方法，因此通过使用接口可以让我们实现自定义类型
- ▶ 当一个变量具有接口类型时，意味着变量必须具有接口中描述的属性和方法
- ▶ 一个对象符合一个接口的规定，我们称为对象实现了这个接口。接口使用关键字：interface来定义
- ▶ 看下面代码：

接口概念

```
interface IComplexType {  
    id: number;  
    name: string;  
}  
  
let complexType: IComplexType;  
complexType = {id: 1, name: “test”};  
  
let incompleteType: IComplexType;  
incompleteType = {id: 1};
```

\$ **error TS2322: Type '{id: number}' is not assigned to type 'IComplexType'.
Property 'name' is missing in type '{id: number}'**

接口: 可选属性

- ▶ 接口也可以有可选属性，就像前面讲的函数可选参数一样。

```
interface OptionalProp {  
    id: number;  
    name?: string;  
}
```

```
let idOnly: OptionalProp = { id: 1 };  
let idAndName: OptionalProp = { id: 2, name: "idAndName" };  
  
idAndName = idOnly;
```

接口编译

- ▶ 接口是TS编译时候的语言特征， 编译后的代码中不会包含任何的接口代码
- ▶ 接口仅用来在编译的时候， 对代码进行检查
- ▶ 编写接口的时候， 可以采用一些命名规范： 比如接口前面都用大写I。 也可以不采用， 主要看代码编写规范的要求

类

- ▶ 类是对象的定义，描述了对象有什么数据，可以做什么样的操作。接口和类是面向对象开发的基石

```
class SimpleClass {  
    id: number;  
    print(): void {  
        console.log(`SimpleClass.print() called`);  
    }  
}  
  
let mySimpleClass = new SimpleClass();  
mySimpleClass.print();
```

\$ **SimpleClass.print() called**

类属性

- 为了在类里面访问类的属性，需要使用this关键字

```
class SimpleClass {  
  id: number;  
  print(): void {  
    console.log(`SimpleClass has id: ${this.id}`);  
  }  
}  
  
let mySimpleClass = new SimpleClass();  
mySimpleClass.id = 1001;  
mySimpleClass.print();
```

\$ SimpleClass has id: 1001

实现接口

- ▶ 接口和类之间有什么联系和区别?
 - 类是一个对象的定义。包含了属性和函数
 - 接口是一个自定义类型的定义。也包含了属性和函数
 - 两者的区别是：类必须实现自己的属性和函数，而接口仅仅是描述属性和函数
 - 我们就可以使用接口来描述：一组类的共有行为
 - 看下面代码：

实现接口

```
class ClassA {  
    print() {  
        console.log('ClassA.print()')  
    }  
}  
  
class ClassB {  
    print() {  
        console.log('ClassB.print()')  
    }  
}
```

- 代码中定义了两个类：ClassA、ClassB，两个类都有print()函数。
- 假设：我们接着要写的代码只关注一个类是不是有print()函数，不关心到底是用的那个类，那么就可以非常容易的用接口来描述，看下面代码：

实现接口

```
interface IPrint {  
    print();  
}  
function printClass(a: IPrint) {  
    a.print();  
}
```

- 我们定义了一个IPrint接口，用来描述下面传递给函数printClass的参数的类型
- IPrint接口里面有一个print()函数，就意味着传递给printClass函数的参数变量必须要包含一个print()函数
- 接着，我们修改我们类的定义：

实现接口

```
class ClassA implements IPrint {  
    print() {  
        console.log('ClassA.print()')  
    }  
}  
  
class ClassB implements IPrint {  
    print() {  
        console.log('ClassB.print()')  
    }  
}
```

- 现在我们两个类ClassA, ClassB的定义都使用implements关键字实现了IPrint接口
- 这样我们就可以像下面这样使用这两个类ClassA, ClassB的实例：

实现接口

```
let classA = new ClassA;  
let classB = new ClassB;  
  
printClass(classA);  
printClass(classB);
```

- 代码中我们分别创建了ClassA,ClassB的两个实例
 - 然后我们用这两个实例作为参数， 分别传递给printClass函数
 - 因为printClass函数只要求参数里面包含print(), 因此这两个语句都可以正常执行
-
- ▶ 因此， 接口就是描述类共有行为的一种方法。接口也可以看做是， 类如果想要提供某类行为的时候， 需要必须实现的一种契约。

类的构造函数

- ▶ 类可以在实例化的时候接收参数。这样可以让我们在一行代码中就同时完成创建类实例和设定类的属性

```
class ClassWithConstructor {  
    id: number;  
    name: string;  
    constructor(_id: number, _name: string) {  
        this.id = _id;  
        this.name = _name;  
    }  
}
```

- 代码中，类ClassWithConstructor 包含两个属性id, name，同时还有一个构造函数constructor，这个构造函数接收两个参数，并把传入的参数赋值给两个属性id,name

类的构造函数

```
let classWithConstructor = new ClassWithConstructor(1, "name");

console.log(`classWithConstructor.id = ${classWithConstructor.id}`);
console.log(`classWithConstructor.name = ${classWithConstructor.name}`);
```

\$ **classWithConstructor.id = 1**
\$ **classWithConstructor.name = name**

类的函数

- ▶ 类中的函数都遵从我们之前讲过的关于函数的强类型规则，回忆一下：
 - 强类型
 - 可以使用any释放强类型
 - 可有可选参数
 - 可有默认参数
 - 可使用参数数组，或者rest语法
 - 允许有回调函数，并对指定回调函数的签名
 - 可以有函数重载
- ▶ 看一下代码的例子：

类的函数

```
class ComplexType implements IComplexType {  
    id: number;  
    name: string;  
    constructor(idArg: number, nameArg: string);  
    constructor(idArg: string, nameArg: string);  
    constructor(idArg:any, nameArg:any) {  
        this.id = idArg;  
        this.name = nameArg;  
    }  
    print(): string {  
        return “id:” + this.id + “name:” + this.name;  
    }  
    .....  
}
```

接下页

类的函数

接上页

```
usingTheAnyKeyword(arg1: any): any {  
    this.id = arg1;  
}  
  
usingOptionalParameters(optionalArg1?: number) {  
    if (optionalArg1) {  
        this.id = optionalArg1;  
    }  
}  
  
usingDefaultParameters(defaultArg1: number = 0) {  
    this.id = defaultArg1;  
}  
.....
```

接下页

类的函数

接上页

```
usingRestSyntax(...arrArray: number[]) {  
  if (arrArray.length > 0) {  
    this.id = arrArray[0];  
  }  
}  
  
usingFunctionCallbacks( callback: (id: number) => string) {  
  callback(this.id);  
}  
}
```

- 首先要注意的是构造函数,我们使用了函数重载: 允许我们在使用构造函数的时候, 既可以传入一个数字, 一个字符串参数, 又可以传入两个字符串参数
- 看下面代码:

类的函数

```
let ct_1 = new ComplexType(1, "ct_1");
let ct_2 = new ComplexType("abc", "ct_2");
let ct_3 = new ComplexType(true, "test");
```

- 代码中，ct_1使用数字和字符串实例化ComplexType类，ct_2使用了两个字符串，ct_3使用了boolean和字符串
- TS编译器会对ct_3报错。因为我们使用了构造函数重载，但不允许参数中有boolean类型的值
- 当使用构造函数重载的时候，需要特别留意一下，很容易出错误。我们来看下面代码：

类的函数

```
let ct_2 = new ComplexType("abc", "ct_2");
ct_2.print();
```

- 在使用两个字符串为参数生成实例ct_2的时候，在构造函数中 this.id = idArg

```
class ComplexType implements IComplexType {
  id: number;
  name: string;
  constructor(idArg: number, nameArg: string);
  constructor(idArg: string, nameArg: string);
  constructor(idArg: any, nameArg: any) {
    this.id = idArg;
    this.name = nameArg;
  }
  print(): string {
    return "id:" + this.id + "name:" + this.name;
  }
  ....
```

类的函数

- 很明显，类中id属性被赋值了一个字符串，会引起编译器错误
- 这时候，就需要使用类型卫士来解决这个问题：

```
class ComplexType implements IComplexType {  
    id: number;  
    name: string;  
    constructor(idArg: number, nameArg: string);  
    constructor(idArg: string, nameArg: string);  
    constructor(idArg:any, nameArg:any) {  
        if(typeof idArg === number) {  
            this.id = idArg;  
        }  
        this.name = nameArg;  
    }  
    print(): string {  
        return "id:" + this.id + "name:" + this.name;  
    }  
    ....
```

类的函数

- 继续看类定义里面后面的函数定义：

```
ct_1.usingTheAnyKeyword(true);  
ct_1.usingTheAnyKeyword({ id:1, name: "string"});
```

- 上面代码都是合法的，因为使用了any类型

```
ct_1.usingOptionalParameters(1);  
ct_1.usingOptionalParameters();
```

```
ct_1.usingDefaultParameters(2);  
ct_1.usingDefaultParameters();
```

- 上面代码都是合法的，可选参数的函数既可以传入1也可以什么都不传入
- 默认参数的函数可以传入2，也可以什么都不传入，这时候参数值是默认的0

类的函数

```
ct_1.usingRestSyntax(1,2,3);  
ct_1.usingRestSyntax(1,2,3,4);
```

- rest类型参数的函数中，传入的参数数量不受限制

```
function myCallbackFunction(id: number): string {  
    return id.toString();  
}  
ct_1.usingFunctionCallbacks(myCallbackFunction);
```

- 回调函数作为参数的函数中，我们传入的myCallbackFunction参数符合定义的回调函数格式，因此上面代码都是合法代码

接口的函数定义

- ▶ 接口和类一样，允许同样的规则应用在接口函数的定义上。
 - 我们来更新一下IComplexType接口的定义：

```
interface IComplexType {  
    id: number;  
    name: string;  
    print(): string;  
    usingTheAnyKeyword(arg1: any): any;  
    usingOptionalParameters(optionalArg1?: number);  
    usingDefaultParameters(defaultArg1?: number);  
    usingRestSyntax(...arrArray: number[]);  
    usingFunctionCallbacks( callback: (id: number) => string);  
}
```

接口的函数定义

- 在IComplexType接口的定义中：看起来和我们的类定义中的一样，区别是没有函数体
- 有一个特殊的地方就是默认参数函数，因为接口只是类或者对象的描述，所以接口中不能包含值或者变量；我们就把默认参数函数定义为了可选参数。让将来实现这个接口的类来分配这个默认参数的值。

```
usingDefaultParameters(defaultArg1?: number);
```

- 还有一个特别的地方就是接口里面没有构造函数的定义。因为TS规定接口不能包含构造函数的签名。
- 为什么呢？看下面代码：

接口的函数定义

```
interface IComplexType {  
    constructor(arg1: any, arg2: any);  
}
```

\$ error TS2420: Class ‘ComplexType’ incorrectly implements interface
‘IComplexType’. Types of property ‘constructor’ are incompatible.

- 错误提示告诉我们，当我们使用构造函数时构造函数的返回值类型，由TS编译器默认规定好了
- 因此，IComplexType 如果包含构造函数时，构造函数返回值的类型就是 IComplexType。ComplexType构造函数返回值类型是 ComplexType。
- 即使，ComplexType实现了IComplexType接口，ComplexType与IComplexType也是两种不同类型，所以**接口中不能包含构造函数**

类修饰符

- ▶ public、private、protected; 先讨论public、private, 学习完继承后再讨论protected

```
class ClassWithPublicProperty {  
    public id: number;  
}  
let publicAccess = new ClassWithPublicProperty();  
publicAccess.id = 10;
```

- 代码中定义了public属性id, 因此在类实例化后, 可以在类实例上给这个id属性赋值

类修饰符

```
class ClassWithPrivateProperty {  
    private id: number;  
    constructor(_id: number) {  
        this.id = _id;  
    }  
}  
let privateAccess = new ClassWithPrivateProperty();  
publicAccess.id = 20;
```

\$ **error TS2341: Property 'id' is private and only accessible within the class 'ClassWithPrivateProperty'.**

- 错误信息提示我们：private属性不能在类的实例上访问，只能在类的内部访问。代码中在类的构造函数中访问了private属性
- 类定义中没有加修饰符的，默认是public

构造函数访问修饰符

- ▶ TS提供一种快捷方式，直接在构造函数中指定参数的访问修饰符

```
class classWithAutomaticProperties{  
    constructor(public id: number, private name: string) {  
    }  
}  
  
let myAutoClass = new classWithAutomaticProperties(1, “className”);  
console.log(`myAutoClass id:${ myAutoClass.id }`);  
console.log(`myAutoClass name:${ myAutoClass.name }`);
```

- 代码定义了一个类：classWithAutomaticProperties，并且在构造函数的参数中定义了带访问修饰符的两个参数
- 注意：这种方式只能在构造函数中使用

构造函数访问修饰符

**\$ Property 'name' is private and only accessible within class
'classWithAutomaticProperties'**

- 构造函数参数： name, 声明的是private， 因此，在类的实例上访问name属性的时候， TS编译器就报错
- 使用这种构造函数访问修饰符的方式， 为类初始化属性虽然方便， 但是会降低代码的可读性。
- 使用直接列出类的属性， 还是通过构造函数参数的方式初始化属性都是可以的。只是需要记住： 在代码中保持统一的风格就可以

只读属性

- 类还可以声明一种只读属性。通过`readonly`关键字，一旦一个属性被声明为只读属性，那么意味着这个属性就不可以修改。唯一可以设定`readonly`属性的地方是构造函数中

```
class classWithReadonly {  
    readonly name: string;  
    constructor(_name: string) {  
        this.name = _name;  
    }  
    setReadOnly(_name: string) {  
        this.name = _name;  
    }  
}
```

\$ error TS2540: Cannot assign to 'name' because it is a constant or read-only property.

类的属性访问器

- ▶ ES5中已经有属性访问器，TS中类也有属性访问器
 - 属性访问器是指在用户访问或者设定类的一个属性的时候，调用的一个函数。这意味着我们可以监测到用户什么时候读取或者设定了类的属性
 - 属性访问器使用一对get, set关键字修饰一个同名函数来实现，看代码：

```
class ClassWithAccessors {  
    private _id: number;  
    get id() {  
        console.log(`inside get id()`);  
        return this._id;  
    }  
    set id(value: number) {  
        console.log(`inside set id()`);  
        this._id = value;  
    }  
}
```

类的属性访问器

- 定义好访问器属性后，就可以这样使用：

```
let classWithAccessors = new ClassWithAccessors();
classWithAccessors.id = 2;
console.log(`id property is set to ${classWithAccessors.id}`);
```

\$ inside set id()

\$ inside get id()

\$ id property is set to 2

- 读取属性值的时候，运行时会调用get函数；设定属性值的时候，运行时会调用set函数

静态函数

- ▶ 静态函数是指调用这个函数的时候，不需要类的实例。是需要在调用的时候，添加上类名作为前缀

```
class StaticClass {  
    static printTwo() {  
        console.log(`2`);  
    }  
}
```

```
StaticClass.printTwo();
```

静态属性

- 和静态函数类似，类也可以包含静态属性。如果一个属性被标记为静态，那么类的所有实例都会共享这个静态属性。

```
class StaticProperty {  
    static count = 0;  
    updateCount() {  
        StaticProperty.count++;  
    }  
}
```

- 代码中在类中定义了一个静态属性count
- 注意：在类中访问这个静态属性的时候我们没有用this，而是用：StaticProperty.count
- 来看一下静态属性如何使用：

静态属性

```
let firstInstance = new StaticProperty();
console.log(`StaticProperty.count = ${StaticProperty.count}`);
firstInstance.updateCount();
console.log(`StaticProperty.count = ${StaticProperty.count}`);
```

```
let secondInstance = new StaticProperty();
secondInstance.updateCount();
console.log(`StaticProperty.count = ${StaticProperty.count}`);
```

\$ StaticProperty.count = 0

\$ StaticProperty.count = 1

\$ StaticProperty.count = 2

- 结果显示静态属性是在所有的类实例之间共享的

命名空间

- › 当做大型项目，尤其是使用第三方库的时候，经常会遇到类或者接口命名冲突的情况。
- › TS使用命名空间来解决这类问题，看代码：

```
namespace FirstNameSpace {  
    class NotExported {  
    }  
    export class NameSpaceClass {  
        id: number;  
    }  
}
```

命名空间

- 当使用namespace的时候， 命名空间里面定义的类需要使用export关键字修饰， 才可以在命名空间外面使用

```
let firstNameSpace = new FirstNameSpace.NameSpaceClass();

let notExported = new FirstNameSpace.NotExported()
```

\$ error TS2339: Property 'NotExported' does not exist on type 'typeof FirstNameSpace'.

命名空间

- 现在我们增加第二个命名空间

```
namespace secondNameSpace {  
    export class NameSpaceClass {  
        name: string;  
    }  
}  
  
let secondNameSpace = new secondNameSpace.NameSpaceClass();
```

- 可以看到通过使用命名空间作为前缀，两个相同类名(NameSpaceClass)实例化不会产生错误

继承

- ▶ 继承是面向对象编程的另一个基石
- ▶ 继承是指一个对象可以以另一个对象作为基本类型，从而继承这个对象的所有特征，包括这个基本类型的所有属性和方法
- ▶ 接口和类都可以继承；被继承的接口叫基接口，被继承的类叫基类。TS使用extends关键字来表示继承
- ▶ 继承的接口称为派生接口，继承的类称为派生类
- ▶ 先看一下接口继承：

接口继承

```
interface IBase {  
    id: number;  
}  
interface IDerivedFromBase extends IBase {  
    name: string;  
}  
class InterfaceInheritanceClass implements IDerivedFromBase {  
    id: number;  
    name: string;  
}
```

- 代码中定义了基接口IBase, 然后IDerivedFromBase接口继承了IBase, 因此: IDerivedFromBase就包含了 id和name属性。
- 类InterfaceInheritanceClass实现了IDerivedFromBase接口, 因此类 InterfaceInheritanceClass就必须包含id和name属性

类继承

```
class BaseClass implements IBase {  
    id: number;  
}
```

```
class DerivedFromBaseClass extends BaseClass implements IDerivedFromBase {  
    name: string;  
}
```

- 第一个类实现了IBase接口，因此必须包含number类型的id属性
- 第二个类继承了基类BaseClass，同时还实现了IDerivedFromBase接口
- 因为基类已经包含了number类型的id属性，所以第二个类只需要实现string类型的name属性，就符合了IDerivedFromBase接口的规定

类继承

- ▶ TS不支持多重继承的概念。多重继承是指一个类可以继承多个基类。
- ▶ TS中仅支持单继承，因此TS中的类只能有单个基类。
- ▶ 但是，TS支持类实现多个接口：看代码：

类继承

```
interface IFirstInterface {  
    id: number;  
}  
  
interface ISecondInterface {  
    name: string;  
}  
  
class MultipleInterfaces implements IFirstInterface, ISecondInterface {  
    id: number;  
    name: string;  
}
```

- 类MultipleInterfaces实现了两个接口，因此它必须实现id和name属性，来满足这两个接口的要求

super关键字

- 当使用继承的时候，基类和派生类很可能有相同名称的方法，尤其是当类包含构造函数的时候
- 如果一个基类定义了构造函数，那么派生类要调用基类的构造函数就会发生冲突，因为派生类和基类的构造函数都叫： constructor
- TS使用super关键字来解决这个问题
- super关键字让派生类可以调用基类中相同名称的函数
- 看下面代码：

super关键字

```
class BaseClassWithConstructor {  
    private id: number;  
    constructor(_id: number) {  
        this.id = _id;  
    }  
}  
class DerivedClassWithConstructor extends BaseClassWithConstructor {  
    private name: string;  
    constructor(_id:number, _name: string) {  
        super(_id);  
        this.name = _name;  
    }  
}
```

- 代码中使用super调用了基类BaseClassWithConstructor中的构造函数，把_id参数传递给基类的构造函数，来完成id属性的赋值

函数重载

- 类的构造函数本质就是一个函数，因为我们可以用super关键字来在普通的函数中使用，从而实现函数的重载
- 当派生类的函数和基类的函数重名的时候，就可以使用super关键字来重载基类的函数，看代码：

类函数重载

```
class BaseClassWithFunction {  
    public id: number;  
    getProperties(): string {  
        return `id: ${this.id}`;  
    }  
}  
class DerivedClassWithFunction extends BaseClassWithFunction {  
    public name: string;  
    getProperties(): string {  
        return `${super.getProperties()}` + `, name:${this.name}`;  
    }  
}
```

- 代码中基类和派生类有同名函数：getProperties，在派生类中用super调用基类的getProperties

类函数重载

- 看一下如何使用这两个类

```
let derivedClassWithFunction = new DerivedClassWithFunction();
derivedClassWithFunction.id = 1;
derivedClassWithFunction.name = "derivedName";
console.log(derivedClassWithFunction.getProperties());
```

\$ **id: 1 , name: derivedName**

- 实例化了类：DerivedClassWithFunction 的一个实例，然后给实例的id和name属性赋值，并调用derivedClassWithFunction.getProperties()方法打印结果。这个方法内部使用了super关键字调用基类中的getProperties()方法

protected修饰符

- 之前讲到private和public修饰符，现在讲protected修饰符

```
class ClassUsingProtected {  
    protected id: number;  
    public getId(): number {  
        return this.id;  
    }  
}  
  
class DerivedFromProtected extends ClassUsingProtected {  
    constructor(): void {  
        super();  
        this.id = 0;  
    }  
}
```

- 基类ClassUsingProtected中包含了protected修饰符修饰的属性id，在派生类中可以直接this.id进行访问

protected修饰符

- 来看一下在类外面访问由protected修饰符修饰的属性，会怎么样？

```
let derivedFromProtected = new DerivedFromProtected();
derivedFromProtected.id = 1;
console.log(`getId returns: ${derivedFromProtected.getId()}`);
```

\$ error TS2445: Property 'id' is protected and only accessible within class 'ClassUsingProtected' and its subclasses.

- 错误提示告诉我们，被protected修饰符修饰的属性，只允许在类内部或者类的子类中访问，在类外面是无法访问的

抽象类

- ▶ 抽象类是另一个面向对象编程的基本概念
- ▶ 抽象类是一个类的定义，但是不能被实例化。换句话说，就是抽象类是专门为了继承而存在的
- ▶ 抽象类主要用于提供一套基本的函数和属性，提供给一组比较相似的类进行共享
- ▶ 抽象类和接口有点像，但是是有区别的，抽象类不能被实例化，但是可以包含函数的实现，而接口只能包含函数的描述，不能包含函数的实现
- ▶ 抽象类是一种代码复用的有效手段；抽象类使用abstract关键字
- ▶ 看代码：

抽象类

```
class Employee {  
    public id: number;  
    public name: string;  
    printDetail() {  
        console.log(`id: ${this.id} ` + ` , name: ${ this.name }`);  
    }  
}  
class Manager {  
    public id: number;  
    public name: string;  
    public Employees: Employee [];  
    printDetail() {  
        console.log(`id: ${this.id} ` + ` , name: ${ this.name }` +  
            `employeeCount: ${ this.Employees.length }`);  
    }  
}
```

抽象类

- 代码中，我们定义了两个类：Employee和Manager
- 这两个类很相似，都包含id,name属性和printDetail()方法。唯一不同是Manager类多了一个Employees属性
- 为了可以提高代码的复用，我们可以把两个类的公共部分抽提出来，成为一个抽象类
- 看一下代码：

抽象类

```
abstract class AbstractEmployee {  
    public id: number;  
    public name: string;  
    abstract getDetails(): string;  
    public printDetails() {  
        console.log(`this.getDetails()`);  
    }  
}
```

接下页

抽象类

接上页

```
class NewEmployee extends AbstractEmployee {  
    getDetails(): string {  
        return `id: ${this.id}, name: ${this.name}`;  
    }  
}
```

```
class NewManager extends NewEmployee {  
    public Employees: NewEmployee[];  
    getDetails(): string {  
        return super.getDetails() + `, employeeCount: ${this.Employees.length}`  
    }  
}
```

抽象类

- 分析一下代码：我们先定义了一个抽象类：AbstractEmployee。这个抽象类包含了Employee类和Manager类都包含的id和name属性
- 还包括了一个getDetails抽象方法：getDetails，意味着继承这个抽象类的类，必须要实现这个方法
- 最后，还包括了一个printDetails方法，用来打印getDetails方法返回的类的详细信息
- 派生类NewEmployee中，实现了getDetails方法来返回AbstractEmployee类中的id和name属性
- 派生类NewManager又继承了NewEmployee类，我们在NewManager类的getDetails方法中使用了super调用了NewEmployee类中的getDetails方法，并添加了NewManager类自带的信息：\${this.Employees.length}

抽象类

- 看一下我们定义的类如何使用：

```
let employee = new NewEmployee();
employee.id = 1;
employee.name = "Employee Name";
employee.printDetails();
```

```
let manager = new NewManager();
manager.id = 2;
manager.name = "Manager Name";
manager.Employees = new Array();
manager.printDetails();
```

\$ id: 1, name: Employee Name

\$ id: 2, name: Manager Name, employeeCount: 0

- 使用抽象类可以让我们写出逻辑清晰的代码，并且最大限度的复用了代码

组合使用接口、类和继承—工厂设计模式

- ▶ 我们来完成一个假想的功能模块，具体功能如下：
 - 根据给定的出生日期，按照年龄对人群分类，分为：幼儿、少年、成年
 - 标识出哪类人群符合法定年龄可以签订一份合同
 - 小于2岁的划分为幼儿
 - 幼儿不能签订合同
 - 小于18岁大于2岁的划分为少年
 - 少年也不能签订合同
 - 大于18岁的为成年
 - 成年人可以签订合同
 - 为每类人群都生成报告，报告内容包括：出生日期、所属人群、是否可以签订合同

组合使用接口、类和继承—工厂设计模式

▶ 首先看一下什么是工厂设计模式：

- 工厂设计模式使用一个工厂类，为这个工厂类提供信息后，这个工厂类返回某几种类里面其中一个类的实例
- 工厂设计模式的本质是把要创建那个类的实例的决定权放在一个单独的工厂类中
- 因为我们不知道工厂模式会返回那个类型的类的实例，所以我们需要一种办法可以控制所有不同类返回的各种类型的实例
- 接口可以约束和规定类要实现的属性和方法，所以这里很适合使用接口来实现
- 分析了我们的业务需求后，我们可以创建三个类： Infant类， Child类， Adult类
- 当判断能否签合同的时候， Infant和Child类返回false。 Adult类返回true。

IPerson接口

- 按照需求，从工厂中返回的实例，都需要可以完成两件事：打印信息和判断是否可以签合同
- 来定义接口：

```
enum PersonCategory {  
    Infant,  
    Child,  
    Adult  
}  
  
interface IPerson {  
    Category: PersonCategory;  
    canSignContracts(): boolean;  
    printDetails();  
}
```

Person抽象类

- 为了提高代码复用，我们先创建一个抽象类Person：

```
abstract class Person implements IPerson {
    Category: PersonCategory;
    private DateOfBirth: Date;
    constructor(dateOfBirth: Date) {
        this.DateofBirth = dateOfBirth;
    }
    abstract canSignContracts(): boolean;
    printDetails(): void {
        console.log(`Person: `);
        console.log(`Date of Birth: ` + `${this.DateOfBirth.toDateString()}`);
        console.log(`Categroy: ` + `${PersonCategory[this.Category]}`);
        console.log(`Can sign: ` + `${this.canSignContracts()}`);
    }
}
```

Person抽象类

- 抽象类Person中有几点需要注意：
 - DateOfBirth 属性声明为private。因为我们的需求里没有需要在类的外面打印或访问出生日期，所以出生日期设定好后，就不再改动了
 - canSignContracts()方法被设定为abstract,表示所有继承Person抽象类的类，都必须实现这个方法
 - printDetail()方法在抽象类中完全实现了。意味着所有继承Person抽象类的类，都自动拥有了这个方法

特定类

- 现在三个特定类:Infant、Child、Adult都继承自抽象类Person

```
class Infant extends Person {  
    constructor(dateOfBirth: Date) {  
        super(dateOfBirth);  
        this.Category = PersonCategory.Infant;  
    }  
    canSignContracts(): boolean {  
        return false;  
    }  
}
```

特定类

- Child 类

```
class Child extends Person {  
    constructor(dateOfBirth: Date) {  
        super(dateOfBirth);  
        this.Category = PersonCategory.Child;  
    }  
    canSignContracts(): boolean {  
        return false;  
    }  
}
```

特定类

- Adult 类

```
class Adult extends Person {  
    constructor(dateOfBirth: Date) {  
        super(dateOfBirth);  
        this.Category = PersonCategory.Adult;  
    }  
    canSignContracts(): boolean {  
        return true;  
    }  
}
```

- 由于dateOfBirth在抽象类中声明为private.因此， dateOfBirth属性只能在Person中访问到。而我们三个特定类中， 只在构造函数中能通过super传递过来

特定类

通过使用抽象类，我们在定义特定类的时候，就非常方便。只需要在特定类中指定this.Category，以及定义各自的canSignContracts()方法

工厂类

- 现在开始实现工厂类：PersonFactory
- 这个工厂类功能很明确：就是根据给定的生日日期，判断是属于哪个人群，并返回对应的实例

```
class PersonFactory {  
    getPerson(dateOfBirth: Date): IPerson {  
        let dateNow = new Date();  
        let currentMonth = dateNow.getMonth() + 1;  
        let currentDate = dateNow.getDate();  
  
        let dateTwoYearsAgo = new Date(  
            dateNow.getFullYear() - 2,  
            currentMonth, currentDate  
        );  
    }  
}
```

接下页

工厂类

接上页

```
let date18YearsAgo = new Date(  
    dateNow.getFullYear() - 18,  
    currentMonth, currentDate  
);  
  
if(dateOfBirth >= dateTwoYearsAgo) {  
    return new Infant(dateOfBirth);  
}
```

接下页

工厂类

接上页

```
let date18YearsAgo = new Date(  
    dateNow.getFullYear() - 18,  
    currentMonth, currentDate  
);  
  
if(dateOfBirth >= dateTwoYearsAgo) {  
    return new Infant(dateOfBirth);  
}  
if(dateOfBirth >= date18YearsAgo) {  
    return new Child(dateOfBirth);  
}  
return new Adult(dateOfBirth);  
}  
}
```

工厂类

- 工厂类只包含一个函数：getPerson()
- 在这个函数中，根据当前日期计算出两年前的日期dateTwoYearsAgo，18年前的日期：date18YearsAgo
- 根据传入getPerson()函数中的参数dateOfBirth，判断最后应该返回那个类的实例
- 再看一下，这个工厂类如何使用：

工厂类

```
let factory = new PersonFactory();

let p1 = factory.getPerson(new Date(2018, 11, 12));
p1.printDetails();

let p2 = factory.getPerson(new Date(2014, 11, 12));
p2.printDetails();

let p3 = factory.getPerson(new Date(1989, 11, 12));
p3.printDetails();
```

工厂类总结

- 我们满足了所有的业务要求
- 查看我们的代码，我们的三个类：Infant, Child, Audit只关心自己的分类和是否可以签字
- Person抽象类只关心与接口IPerson相关的逻辑
- 工厂类PersonFactory只关心传入什么样的生日日期，输出对应的对象实例

TypeScript培训内容列表

- 为什么要使用TypeScript;
- TypeScript中的类型、变量、函数;
- 接口、类和继承;
- 装饰器、泛型和异步编程;
- 声明文件的使用和编写;
- 使用第三方JS库;
- 测试驱动开发模式;
- 测试驱动开发框架;
- 模块化;
- 面向对象编程;
- 依赖注入;
- 实战;

装饰器、泛型和异步编程

- ▶ 在接口、类和继承的基础上，TypeScript还引入了一些高级的面向对象编程的特点，包括：装饰器，泛型和用于异步编程的promises 及 async/await 关键字
 - 装饰器可以在对类进行定义的时候注入和查询元数据，同时可以在定义类的时候，提供额外的功能
 - 泛型可以让我们定义一种只有在运行时才知道对象具体类型的代表类型
 - promises可以让我们以流畅的语法来进行异步编程
 - async/await 可以暂停程序的执行，直到异步函数执行完毕
 - 当编写大型应用的时候，这些工具可以帮助我们实现很多种设计模式

装饰器

- 我们已经学过，类的定义就是描述一个类的样子。换句话说，就是描述一个类应该具有什么样的属性和方法
- 只有当类被实例化的时候，也就是说类的实例被创建的时候，这些属性和方法才可用
- 装饰器可以允许我们在进行类定义的时候，注入其他代码块
- 装饰器是一种特殊类型的声明，它可以被附加到其他的声明上
- 比如：装饰器可以被附加到类、类的属性、类的方法、甚至方法的参数上

装饰器

- 装饰器使用@expression 这种语法格式， 其中expression代表一个表达式
- expression这个表达式被求值之后， 结果必须是一个函数
- 这个函数会在运行时的时候， 被调用
- 被装饰器装饰的那些声明信息(类、 类属性、 类方法、 方法的参数)作为参数传入到这个函数中
- 下面看代码来学习装饰器如何定义， 如何使用：

启用装饰器

- 装饰器是属于ES7标准中推荐的语法，TS允许我们提前来使用这个特性
- 在tsconfig.json中配置TS，启用"experimentalDecorators"装饰器的特性

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es3",  
    "sourceMap": true,  
    "experimentalDecorators": true  
},  
  "exclude": [  
    "node_modules"  
]  
}
```

装饰器语法

- 装饰器就是一个带着一系列特殊参数的普通函数
- 这些特殊参数在JS运行时自动填充，它们包含了与这个装饰器对应的类的信息
- 特殊参数的数量和类型，决定了装饰器可以应用在那种场景（1个参数的时候用于类，2个参数的时候用于属性，3个参数的时候用于方法）
- 看代码：

```
function simpleDecorator(constructor: Function) {  
    console.log('simpleDecorator called.');//打印出“simpleDecorator called.”  
}
```

- 上面代码就是一个简单装饰器的定义：一个普通的函数，接收一个类型是Function，名为：constructor的参数；函数体里面仅仅是打印一个字符串；

装饰器语法

- 我们看一下如何应用这个简单的装饰器：

```
@simpleDecorator  
class ClassWithSimpleDecorator() {  
}
```

- 使用@符号来表示装饰器，代码中把我们定义的简单装饰器应用在一个 ClassWithSimpleDecorator类上。运行代码：

\$ simpleDecorator called.

- 代码运行的结果，表明了什么？？

装饰器语法

- 首先：我们仅仅是定义了一个类，并没有对类进行实例化
- 其次：我们只是在这个类的定义前面，增加了一个装饰器
- 结论是：增加了装饰器的类，不需要实例化，就可以在类定义的时候，运行装饰器这个函数
- 我们进一步扩展刚才的代码：

```
let instance_1 = new ClassWithSimpleDecorator();
let instance_2 = new ClassWithSimpleDecorator();

console.log(`instance_1: ${instance_1}`);
console.log(`instance_2: ${instance_2}`);
```

装饰器语法

- 代码中，实例化了类(ClassWithSimpleDecorator)的两个对象：instance_1和instance_2
- 运行结果：

```
$ simpleDecorator called.  
$ instance_1: [object object]  
$ instance_2: [object object]
```

- 代码执行结果为我们得出如下结论：装饰器只调用一次，无论类进行了多少次实例化。装饰器仅在类定义的时候，才执行。

多重装饰器

- 一个类可以应用多个装饰器，看代码：

```
function secondDecorator(constructor: Function) {  
    console.log(`secondDecorator called`);  
}  
  
@simpleDecorator  
@secondDecorator  
class ClassWithMultipleDecorators() {  
}
```

\$ **secondDecorator called.**

\$ **simpleDecorator called.**

- 运行结果告诉我们，可以在类上应用多个装饰器，装饰器执行顺序是：离类近的装饰器先执行

装饰器工厂

- 为了让装饰器函数可以接受参数，我们可以使用装饰器工厂
- 装饰器工厂就是一个简单的包装函数，返回一个表达式，表达式在运行时运算后，结果为装饰器函数

```
function decoratorFactory (name: string) {  
    return function(ctor: Function) {  
        console.log(`decorator function called with: ${name}`);  
    }  
}
```

- 现在我们把装饰器函数包装在一个装饰器工厂函数中，接下来就可以这样使用这个工厂函数：

装饰器工厂

```
function decoratorFactory (name: string) {  
    return function(constructor: Function) {  
        console.log(`decorator function called with: ${name}`);  
    }  
}  
@decoratorFactory('testName')  
class ClassWithDecoratorFactory {  
}
```

\$ decorator function called with: testName

- 装饰器工厂需要注意以下几点，首先：装饰器自身依然是由JS运行时自动填充参数并进行调用
- 其次，装饰器工厂必须返回一个函数定义。
- 最后，装饰器工厂定义的参数，可以在装饰器函数自己内部使用

类装饰器

- 目前我们看到所有的装饰器例子都是类装饰器。记住一点：装饰器函数会在类定义的时候，由JS运行时自动调用。
- 之前的装饰器函数都是接收一个特殊参数：constructor 类型为：Function；JS运行时会为我们自动装填这个参数。
- 我们来仔细看一下装饰器的这个特殊参数
- 类装饰器函数，会被填充一个它装饰的类的构造函数作为参数，进行调用，看代码：

类装饰器

```
function classConstructorDec (constructor: Function) {  
    console.log(`constructor: ${constructor}`);  
}  
@ classConstructorDec  
class ClassWithConstructor {  
}
```

- 这里，我们定义了一个装饰器，装饰器内部仅仅是打印传入的参数：constructor
\$ constructor: function ClassWithConstructor() {}
- 运行结果告诉我们，为装饰器传入的参数是它装饰的类的构造函数
- 我们来增强一下，这个装饰器函数：

类装饰器

```
function classConstructorDec (constructor: Function) {  
    console.log(`constructor: ${constructor}`);  
    console.log(`constructor.name: ${(<any>constructor).name}`);  
    constructor.prototype.testProperty = 'testProperty_name';  
  
}  
@classConstructorDec  
class ClassWithConstructor {  
}
```

- 增加的代码中，第一行我们打印了传给装饰器的参数的name属性。
- 为了可以成功访问参数的name属性，我们使用了<any>显示转换参数类型为any。这很有必要，因为function的name属性只在ES6中有效。如果，不转换为any类型的话，TS的编译器就会产生编译错误

类装饰器修改类的定义

```
function classConstructorDec (constructor: Function) {  
    console.log(`constructor: ${constructor}`);  
    console.log(`constructor.name: ${(<any>constructor).name}`);  
    constructor.prototype.testProperty = 'testProperty_name';  
  
}  
@classConstructorDec  
class ClassWithConstructor {  
}
```

- 装饰器函数的第二行，实际上是在类的构造函数的原型上增加了一个属性：testProperty，并为这个属性赋值，这种操作是一个展示装饰器如何修改类的定义的很好的例子。

装饰器修改类的定义

- 接下来，我们就可以访问这个类的属性：

```
let classConstrInstance = new ClassWithConstructor();
console.log(`classConstrInstance.testProperty: ` +
`$\{ <any>classConstrInstance).testProperty }`);
```

```
$ constructor: function ClassWithConstructor() {}
$ constructor.name: ClassWithConstructor
$ classConstrInstance.testProperty: testProperty_name
```

- 代码中，我们实例化了类ClassWithConstructor的一个实例。并且，打印这个实例的testProperty属性
- 注意：字符串模板中使用了<any>来显示转换实例的类型为any，是因为：要打印的testProperty不是类定义的时候具有的属性，而是通过装饰器动态添加的属性。

属性装饰器

- 属性装饰器是指那些用在类的属性上的装饰器函数
- 属性装饰器带有两个参数：类的原型、类的属性名

```
function propertyDec(target: any, propertyKey: string) {  
    console.log(`target: ${ target }`);  
    console.log(`target constructor: ${ target.constructor }`);  
    console.log(`class name: ${ target.constructor.name }`);  
    console.log(`propertyKey: ${ propertyKey }`);  
}  
  
class ClassWithPropertyDec{  
    @propertyDec name: string;  
}
```

- 代码中定义了一个属性装饰器，接收两个参数：target、propertyKey。其中，target为any类型，propertyKey为string类型

属性装饰器

- 属性装饰器内部分别打印了：target, target.constructor, target.constructor.name, propertyKey
- 接下来，我们定义了一个类：ClassWithPropertyDec，在类中我们使用了属性装饰器，装饰了属性name；结果输出如下：

\$ target: [object object]

\$ target.constructor: function ClassWithPropertyDec() {}

\$ target.constructor.name: ClassWithPropertyDec

\$ propertyKey: name

- 输出的第一行[object object]，表示传入的第一个参数是类的原型对象；第二行打印了类的构造函数；第三行打印类的构造函数的名称，返回的就是类的名称；最后一行，打印了属性装饰器装饰的类的属性的名称

属性装饰器

通过属性装饰器，给我们一种能力：用来监视类中是否声明了某个名字的属性

静态属性装饰器

- 属性装饰器也可以应用在类的静态属性上。语法与普通属性装饰器一样。
- 但是，在运行时传递进来的参数略有不同。
- 我们来看一下前面定义的属性装饰器，如果应用在类的静态属性上会怎么样：

```
function propertyDec(target: any, propertyKey: string) {  
    console.log(`target: ${ target }`);  
    console.log(`target constructor: ${ target.constructor }`);  
    console.log(`class name: ${ target.constructor.name }`);  
    console.log(`propertyKey: ${ propertyKey }`);  
}  
class StaticClassWithPropertyDec{  
    @propertyDec static name: string;  
}
```

静态属性装饰器

- 当属性装饰器应用在静态类属性上，输出的结果如下：

```
$ target: function StaticClassWithPropertyDec() {}  
$ target.constructor: function Function() { [native code] }  
$ target.constructor.name: Function  
$ propertyKey: name
```

- 输出的第一行，表示在对类的静态属性使用装饰器的时候，传入装饰器函数的第一个参数是类的构造函数；第二、三行说明这个构造函数的构造函数一个名为Function的普通函数；第四行依然输入了静态属性的名称；
- 这就给我们提示：在使用属性装饰器的时候，如果用在静态属性上那么传给装饰器函数的第一个参数是构造函数，而用在普通属性上那么传给装饰器函数的第一个参数是类的原型对象

静态属性装饰器

- 我们来修改一下装饰器函数，来适应这两种情况：

```
function propertyDec(target:any, propertyKey: string) {  
    if(typeof(target) === 'function') {  
        console.log('class name: ${target.name}');  
    } else {  
        console.log('class name: ${target.constructor.name}');  
    }  
    console.log('propertyKey: ${propertyKey}');  
}
```

```
$ class name: ClassWithPropertyDec  
$ propertyKey: name  
$ class name: StaticClassWithPropertyDec  
$ propertyKey: name
```

方法装饰器

- 方法装饰器可以应用在类的方法上，它接收三个参数：类原型、方法名称、方法属性描述符（可选，仅在编译目标为ES5及以上版本时才生效）

```
function methodDec(target:any, methodName:string, descriptor?:  
PropertyDescriptor) {  
    console.log(`target: ${ target }`);  
    console.log(`methodName: ${ methodName }`);  
    console.log(`target[methodName]: ${ target[methodName] }`);  
}
```

- 代码中，装饰器函数体中第三行，答应了target[methodName]，实际上是打印了methodName的函数定义。
- 接下来，看一下方法装饰器如何使用：

方法装饰器

```
function methodDec(target:any, methodName:string, descriptor?:  
PropertyDescriptor) {  
    console.log(`target: ${ target }`);  
    console.log(`methodName: ${ methodName }`);  
    console.log(`target[methodName]: ${ target[methodName] }`);  
}  
class ClassWithMethodDec {  
    @methodDec  
    print(output: string) {  
        console.log(`ClassWithMethodDec.print: (${ output } called.)`);  
    }  
}
```

方法装饰器

- 代码中，我们定义了一个类：ClassWithMethodDec。这个类只有一个print方法，这个方法接收一个string类型的参数output，并且打印这个参数。
- print方法被方法装饰器methodDec装饰，代码输入如下结果：

```
$ target: [object object]
$ methodName: print
$ target[methodName]: function(output) {
  console.log("ClassWithMethodDec.print:" + output + " called." ) }
```

- 从输出结果可以看出，传入方法装饰器的第一个参数是类的原型；第二个参数是方法装饰器装饰的方法的名称；第三个参数可选代码中没有提供第三个参数，而打印的第三行target[methodName]是方法的定义；

方法装饰器的使用

- 因为我们在方法装饰器中定义可以使用的方法，因此，我们可以使用方法装饰器为类注入新的功能。
- 例如：我们想要创建一个审计记录和对类中的某个方法调用的日志，看代码：

```
function auditLogDec(target:any, methodName:string, descriptor?:  
PropertyDescriptor) {  
    let originalFunction = target[methodName];  
    let auditFunction = function() {  
        console.log(`auditLogDec: override of ${methodName} called.`);  
        originalFunction.apply(this, arguments);  
    }  
    target[methodName] = auditFunction;  
}
```

方法装饰器的使用

- 方法装饰器中，实际上是把原始的被装饰的方法包装成为一个新的方法。新的方法增加了一个打印日志的功能，并且使用JS的apply函数调用originalFunction；
- 最后一行，把包装的新函数赋值给原来的函数

复习JS中的： apply, call, bind

- apply() 方法调用一个具有给定this值的函数，以及作为一个数组（或类似数组对象）提供的参数。
- 注意：call()方法的作用和 apply() 方法类似，区别就是call()方法接受的是参数列表，而 apply()方法接受的是一个参数数组。
- bind 返回的是一个新的函数，你必须调用它才会被执行。bind 除了返回是函数以外，它的参数和 call 一样。
- 看示例代码：

复习JS中的： apply, call, bind

```
let obj = {  
    name: '小张',  
    objAge: this.age,  
    myFun: function(fm, t) {  
        console.log(this.name +“年龄”+this.age + “来自” + fm + “去往” + t);  
    }  
};  
let db = {  
    name: '小李',  
    age: 99  
};  
obj.myFun.call(db, '北京', '上海');  
obj.myFun.apply(db, ['北京', '上海']);  
obj.myFun.bind(db, '北京', '上海')();  
obj.myFun.bind(db, ['北京', '上海'])();
```

复习JS中的： apply, call, bind

\$ 小李 年齡 99 来自 北京 去往 上海

\$ 小李 年齡 99 来自 北京 去往 上海

\$ 小李 年齡 99 来自 北京 去往 上海

\$ 小李 年齡 99 来自 北京, 上海 去往 undefined

方法装饰器的使用

- 来看一下方法装饰器的具体应用：

```
class ClassWithAuditDec {  
    @auditLogDec  
    print(output: string) {  
        console.log(`ClassWithMethodDec.print` + `(${output}) called.`);  
    }  
}  
  
let auditClass = new ClassWithAuditDec();  
auditClass.print("test");
```

\$ **auditLogDec: override of print called.**

\$ **ClassWithMethodDec.print (test) called.**

方法装饰器的使用

- 代码的输出展示了，装饰器的auditFunction先被调用，然后再调用了类的print函数；

结论：我们可以通过方法装饰器，可以用非侵入式的方式为类的声明注入额外的功能

- 只要应用了这个audit装饰器的类的方法，都具备了audit装饰器中的打印审计信息的功能

参数装饰器

- 参数装饰器用来装饰一个方法的参数；

```
function parameterDec(target:any, methodName:string,  
parameterIndex:number) {  
    console.log(`target: ${ target }`);  
    console.log(`methodName: ${ methodName }`);  
    console.log(`parameterIndex: ${ parameterIndex }`);  
}  
  
class ClassWithParamDec {  
    print(@parameterDec value: string) {}  
}
```

- 代码中定义了一个参数装饰器，接收三个参数；第一个依然是类的原型，第二个是方法的名称，第三个是方法参数的索引；

参数装饰器

\$ target: [object object]

\$ methodName: print

\$ parameterIndex: 0

- 输出结果展示：第一个参数就是类的原型，第二个参数就是方法的名称；第三个参数就是被修饰方法的参数的索引。
- 我们没有得到被装饰参数的类型、名称这类的信息，因此参数装饰器，仅仅用来表示一个方法是否具有某个参数；比如：@required

元数据反射

- 反射是什么？我们为什么需要JavaScript中的反射？通俗的说，反射就是根据给出的类名（字符串）来生成对象。
- **反射这个词用来描述那些可以检查同一个系统中其它代码(或自己)的代码；反射在一些用例下非常有用(组合/依赖注入，运行时类型检查，测试)**
- 一个强大的反射 API 可以让我们在运行时检测一个未知的对象并且得到它的所有信息。
我们要能通过反射得到以下的信息：
 - 1.这个实例的名字
 - 2.这个实例的类型
 - 3.这个实例实现了哪个接口
 - 4.这个实例的属性的名字和类型
 - 5.这个实例构造函数的参数名和类型

元数据反射

- 原生的ES标准对元数据反射的支持处于早期的开发阶段；Typescript 的编译器已经可以将一些设计时类型元数据序列化给装饰器。
- 我们可以引入 reflect-metadata 库来使用元数据反射 API，将tsconfig.json中的编译参数 emitDecoratorMetadata 设为 true
- 随后我们可以实现我们自己的装饰器并且使用一个可用的元数据设计键。到目前为止，只有三个可用的键：
 - 1.类型元数据使用元数据键"design:type"
 - 2.参数类型元数据使用元数据键"design:paramtypes"
 - 3.返回值类型元数据使用元数据键"design:returntype"

使用元数据反射API获取类型元数据

- 声明如下属性装饰器：

```
function logType(target:any, key: string) {  
  let t = Reflect.getMetadata("design:type", target, key);  
  console.log(`#${key} type:${t.name}`);  
}
```

- 应用这个属性装饰器到一个类的属性上，获取类的属性的类型信息：

```
class Demo {  
  @logType public attr1: string  
}
```

\$ attr1 type: String

使用元数据反射API获取参数类型元数据

- 声明如下参数装饰器：

```
function logParamTypes(target:any, key: string) {  
    let types = Reflect.getMetadata("design:paramtypes", target, key);  
    let s = types.map(a => a.name).join();  
    console.log(`#${key} param types:${s}`);  
}
```

- 应用这个参数装饰器到一个类里面的一个方法上，获取方法的参数的类型信息：

使用元数据反射API获取参数类型元数据

```
class Foo {};
interface IFoo {};
class Demo {
  @logParamTypes // 应用参数装饰器
  doSomething( param1:string, param2:number, param3:Foo, param4: { test:
    string }, param5: IFoo, param6: Function, param7: (a:number) => void,
  ): number {
    return 1;
  }
}
```

\$ **doSomething** param types: **String, Number, Foo, Object, Object, Function, Function**

使用元数据反射API获取返回类型元数据

- 也可以使用元数据键: `design: returntype` 获取返回类型元数据

```
function logParamTypes(target:any, key: string) {  
  let types = Reflect.getMetadata("design:returntype", target, key);  
  let s = types.map(a => a.name).join();  
  console.log(`#${key} return type:${s}`);  
}
```

- 再把这个装饰器应用在类的方法上，返回值就是：

\$ doSomething return type: Number

TS对元数据的基本类型序列化

- 让我们再来看一次上面的 `design:paramtypes` 例子。我们注意到接口 `IFoo` 和字面量对象 `{ test: string}` 都序列化为 `Object`。这是因为 TypeScript 只支持基础类型的序列化。
基础类型的序列化规则是：

1. `number` 序列化为 `Number`
2. `string` 序列化为 `String`
3. `boolean` 序列化为 `Boolean`
4. `any` 序列化为 `Object`
5. `void` 序列化为 `undefined`
6. `Array` 序列化为 `Array`
7. 如果是一个多元组，序列化为 `Array`
8. 如果是一个类，序列化为 `class constructor`
9. 如果是一个枚举，序列化为 `Number`
10. 如果至少有一个调用签名，序列化为 `Function`
11. 其它的序列化为 `Object`（包括接口）

装饰器元数据

- 当我们设定好tsconfig.json中emitDecoratorMetadata: true选项后，TS编译器就会为我们产生一些与类定义相关联的一些额外的信息
- 为了看一些这个编译选项的效果，我们可以看看编译后生成的JS代码；

```
function metadataParameterDec(target:any, methodName:string, parameterIndex:number) {}

class ClassWithMetaData {
    print(@metadataParameterDec id:number, name:string): number {
        return 1000;
    }
}
```

- 代码中定义了一个参数装饰器和一个类，并在类的print方法中为第一个参数使用了这个参数装饰器；

装饰器元数据

- 如果我们没有设定emitDecoratorMetadata选项，那么生成的JS代码为：

```
var ClassWithMetaData = (function(){
    function ClassWithMetaData() {};
    ClassWithMetaData.prototype.print = function(id, name){};
    _decorate(
        [_param(0, metadataParameterDec)],
        ClassWithMetaData.prototype,
        "print");
    return ClassWithMetaData
})();
```

- 生成的JS代码中，为类定义了一个标准的JS闭包。TS还注入了一个_decorate的方法，这个方法包含了print方法的信息：提示方法名称为print并且具有一个参数索引为0

装饰器元数据

- 如果我们设定emitDecoratorMetadata选项为true，那么生成的JS代码：

```
var ClassWithMetaData = (function(){
    function ClassWithMetaData() {};
    ClassWithMetaData.prototype.print = function(id, name){};
    _decorate(
        [_param(0, metadataParameterDec),
         _metadata('design:type', Function),
         _metadata('design:paramtypes', [Number, String]),
         _metadata('design:returntype', Number),],
        ClassWithMetaData.prototype,
        "print");
    return ClassWithMetaData
})();
```

装饰器元数据

- 对比之前生成的JS代码，选项emitDecoratorMetadata为true的时候，生成的JS代码增加了额外的三个_metadata方法的调用；
- 三个_metadata方法分别使用特殊的元数据键：design:type；design:paramtypes；design:returntype
- 这三个_metadata方法实际上就是注册了print方法的额外信息
 - 1.类型元数据使用元数据键"design:type"：注册了print方法的类型是：Function
 - 2.参数类型元数据使用元数据键"design:paramtypes"：注册了print方法的两个参数类型是：Number, String
 - 3.返回值类型元数据使用元数据键"design:returntype"：注册了print方法的返回值类型是Number；

使用装饰器元数据

- 为了可以使用装饰器元数据，我们需要引入 reflect-metadata 库；

```
$ npm install reflect-metadata --save-dev
```

```
$ npm install @types/reflection-metadata --save-dev
```

- 在TS文件中 import ‘reflect-metadata’；
- 然后，就可以在装饰器中使用反射出来的元数据了，看下面代码：

使用装饰器元数据

```
function metadataParameterDec(target:any, methodName:string, parameterIndex:number) {  
    let designType = Reflect.getMetadata("design:type", target, methodName);  
    console.log(`designType: ${designType}`);  
    let designParamTypes = Reflect.getMetadata("design:paramtypes", target, methodName);  
    console.log(`designParamTypes: ${designParamTypes}`);  
    let designReturnType = Reflect.getMetadata("design:returntype", target, methodName);  
    console.log(`designReturnType: ${designReturnType}`);  
}  
  
class ClassWithMetaData {  
    print(@metadataParameterDec id:number, name:string): number {  
        return 1000;  
    }  
}
```

使用装饰器元数据

- 代码运行结果如下：

```
$ designType: function Function() { [native code] }
```

```
$ designParamTypes: function Number() { [native code] }, function String()
{[native code]}
```

```
$ designReturnType: function Number() { [native code] }
```

- 运行结果显示这个参数装饰器装饰的print方法类型是Function；参数是Number, String类型组成的数组，返回类型是Number
- 元数据由TS编译器自动生成，并且可以在运行时被读取和询问，这是非常有价值的；是实现依赖注入和代码分析工具的理论基础；

泛型

- 泛型是一种描述任意类型的对象的同时还要保持类型完整性的一种代码编写形式
- 我们已经学习了接口、类及基本类型来保证代码中强类型；但是，如果有一段代码，要求使用任意类型的对象，而我们又想保证对象类型的完整性，该如何处理？
- 举例来说：我们想要迭代一个对象组成的数组，并把每个对象的值连接起来返回。
 - 如果这个数组是[1,2,3]，那么应该返回“1,2,3”；如果是[“first”,“second”,“third”]，那么应该返回：“first,second,third”
- 我们可以使用any来表示数组可以包含任意类型的对象，但是使用了any后，就无法保证数组中的每个元素都保持同样的类型，这时候，泛型就可以帮助我们解决这个问题。

泛型语法

- 为了示范泛型的语法，我们来写一个连接数组元素的类：Concatenator，这个类需要有如下功能：
 - 确保数组的每个元素类型是一致的；
 - 数组可以保存任意类型的元素；
 - 这个类可以把数组的每个元素连接在一起，返回连接的字符串
- 为了实现类的各项功能，我们需要依赖于数组元素有可能的所有类型都共有的能力；所有的JS对象都有toString()函数，我们可以利用toString()函数来实现一个通用的类，来处理数组中所有元素的输出，看代码：

泛型语法

```
class Concatenator<T> {  
    concatenatorArray(inputArray: Array<T>): string {  
        let returnString = “”;  
        for(let i=0; i<inputArray.length; i++) {  
            if(i > 0) {  
                returnString += ‘,’;  
                returnString += inputArray[i].toString();  
            }  
        }  
        return returnString;  
    }  
}
```

泛型语法

- 代码中，首先看到特殊的语法：<T>，这个语法用来表示一个泛型，并且告诉我们在下面的代码中泛型的名字是T；注：采用T来作为类型的代表借鉴了JAVA中的泛型命名规范
- 在后面的concatenatorArray函数中，在参数的类型指定的时候，也使用了个泛型语法来指定参数的类型是：Array<T>
- 这个泛型<T>的语法，就保证了concatenatorArray函数的参数数组中的元素类型，必须和类：Concatenator实例化的时候传入的参数类型保持一致；

实例化泛型类

- 要实例化泛型类，我们需要告诉编译器泛型T的真实类型是什么，可以使用TS支持的所有类型来指定泛型T的类型：基本类型，类，接口

```
let stringConcat = new Concatenator<string>();
```

```
let numberConcat = new Concatenator<number>();
```

- 注意代码中，实例化类的时候，我们分别使用了string和number来替换类定义时候的泛型T，这时候类定义中的所有T都会被替换成对应的类型
- 下面举一个反例，来看看没有遵守泛型的规定会如何？

实例化泛型类

```
let stringArray: string[] = ["first", "second", "third"];
let numberArray: number[] = [1, 2, 3];
```

```
let stringResult = stringConcat.concatenatorArray(stringArray);
let numberResult = numberConcat.concatenatorArray(numberArray);
```

```
let stringResult2 = stringConcat.concatenatorArray(numberArray);
let numberResult2 = numberConcat.concatenatorArray(stringArray);
```

\$ **error TS2345: Argument of type 'number[]' is not assignable to parameter of type 'string[]'. Type 'number' is not assignable to 'string'.**

\$ **error TS2345: Argument of type 'string[]' is not assignable to parameter of type 'number[]'. Type 'string' is not assignable to 'number'.**

类型T的使用

- 当使用泛型T的时候，需要注意的一点是，在定义了泛型的类或者函数中，必须遵守一个原则：T的所有属性可以代表所有对象的所有属性。来看代码示例：

```
class Concatenator<T> {  
    concatenateArray(inputArray: Array<T>): string {  
        let returnString = “”;  
        for(let i=0; i<inputArray.length; i++) {  
            if(i > 0) {  
                returnString += ‘,’;  
                returnString += inputArray[i].toString();  
            }  
        }  
        return returnString;  
    }  
}
```

类型T的使用

- 代码中的concatenateArray方法，强类型了它的参数inputArray为Array<T>，也就是说使用这个参数的代码无论数组内部的元素是什么类型，只能使用对数组来说通用的属性和方法。
- 我们在代码中，有两个地方用到了这个参数：inputArray.length 和 inputArray[i].toString，因为length属性和toString方法是数组的通用属性和方法。因此，这个类的语法一切正常。
- 再进一步，我们来自定一个class，并作为T传递给inputArray参数，看看会发生什么：

类型T的使用

```
class MyClass {  
    private _name: string;  
    constructor(arg1: number) {  
        this._name = arg1 + “_MyClass”;  
    }  
}  
let myArray: MyClass[] = [  
    new MyClass(1);  
    new MyClass(2);  
    new MyClass(3);  
]
```

- 我们定义了一个构造函数接收一个数字类型参数的类，并创建了一个包含整个类的三个实例的一个数组myArray。我们想用之前的泛型类打印这个数组的元素的链接字符串，如果，我们把这个数组作为参数传入我们之前定义的泛型类，会怎么样？

类型T的使用

```
let myArrayConcatentator = new Concatenator<MyClass>();  
  
let myResult = myArrayConcatentator.concatenateArray(myArray);  
  
console.log(myResult);
```

\$ [object object],[object object],[object object]

- 很显然这并不是我们想要的结果。原因是数组myArray的元素都是object。而object的toString()返回的结果是[object object]
- 为了得到我们期望的结果，可以在我们定义的类：MyClass中，把toString()方法重载一下，看代码：

类型T的使用

```
class MyClass {  
    private _name: string;  
    constructor(arg1: number) {  
        this._name = arg1 + “_MyClass”;  
    }  
    toString(): string {  
        return this._name;  
    }  
}
```

\$ 1_ MyClass, 2_ MyClass, 3_ MyClass

缩小泛型T的生效范围

- 当使用泛型的时候，有的时候我们期望可以使用某种特殊的类型，或者是某种类型的一个子集作为泛型T，TS使用继承来实现这个需求，看下面的例子：
- 先定义一个枚举类型ClubHomeCountry和一个接口IFootballClub

```
enum ClubHomeCountry {  
    England,  
    Germany  
}  
  
interface IFootballClub {  
    getName(): string;  
    getHomeCountry(): ClubHomeCountry;  
}
```

缩小泛型T的生效范围

- 然后，定义一个抽象类FootballClub

```
abstract class FootballClub implements IFootballClub {  
    protected _name: string;  
    protected _homeCountry: ClubHomeCountry;  
    getName() { return this._name };  
    getHomeCountry() { return this._homeCountry };  
}
```

- 接下来定义两个类：Liverpool, BorussiaDortmund；两个类都实现了抽象类FootballClub，并且在构造函数中设定了保护属性：_name, _homeCountry

缩小泛型T的生效范围

```
class Liverpool extends FootballClub {  
    constructor() {  
        super();  
        this._name = “LiverPool F.C.”;  
        this._homeCountry = ClubHomeCountry.England;  
    }  
}  
  
class BorussiaDortmund extends FootballClub {  
    constructor() {  
        super();  
        this._name = “Borussia Dortmund”;  
        this._homeCountry = ClubHomeCountry.Germany;  
    }  
}
```

缩小泛型T的生效范围

- 然后，创建一个类，这个类的类型是泛型T，而泛型T继承了IFootballClub接口

```
class FootballClubPrinter< T extends IFootballClub > {  
    print(arg: T) {  
        console.log(`${arg.getName()} is` + `${this.IsEnglishTeam(arg)}`  
        + `${an English Football Team}`)  
    }  
    IsEnglishTeam(arg: T): string {  
        if(arg.getHomeCountry() === ClubHomeCountry.England) {  
            return "";  
        } else {  
            return NOT;  
        }  
    }  
}
```

缩小泛型T的生效范围

- 我们定义了一个类：FootballClubPrinter。需要留意的是：这个类的类型是泛型T，而泛型T继承自接口IFootballClub。
- 通过使用继承，实现了代码中使用泛型T的地方，都必须遵从接口IFootballClub的约束。在后面的方法print和IsEnglishTeam中体现了这种约束。
- print的参数为泛型T，因此可以在print方法中使用arg.getName()，而getName()是在接口IFootballClub中规定的必须实现的方法。
- 同样的，IsEnglishTeam的参数为泛型T，因此可以在IsEnglishTeam方法中使用arg.getHomeCountry()，而getHomeCountry()也是在接口IFootballClub中规定的必须实现的方法。
- 为了验证，我们来实例化这个泛型类：

缩小泛型T的生效范围

```
let clubInfo = new FootballClubPrinter();  
  
clubInfo.print(new Liverpool());  
  
clubInfo.print(new BorussiaDortmund());
```

\$ Liverpool F.C. is an English Football Team.

\$ Borussia Dortmund is NOT an English Football Team.

泛型接口

- 接口中也可以使用泛型，可以定义一个泛型接口：IFootballClubPrinter
- 下面是这个IFootballClubPrinter接口的定义，这个接口的类型是泛型T，并且泛型T继承了IFootballClub接口
- IFootballClubPrinter接口规定了，实现这个接口的类必须具备print和IsEnglishTeam方法，并且这两个方法的参数必须是泛型T，而泛型T继承了IFootballClub

```
interface IFootballClubPrinter < T extends IFootballClub > {  
    print(arg : T);  
    IsEnglishTeam(arg : T);  
}
```

泛型接口

- 修改前面的类：FootballClubPrinter来实现这个接口：IFootballClubPrinter

```
class FootballClubPrinter< T extends IFootballClub > implements  
IFootballClubPrinter< T > {  
    print(arg: T) {  
        console.log(`#${arg.getName()} is` + `#${this.IsEnglishTeam(arg)}`  
        + `#${an English Football Team}`)  
    }  
    IsEnglishTeam(arg: T): string {  
        if(arg.getHomeCountry() === ClubHomeCountry.England) {  
            return “”;  
        } else {  
            return NOT;  
        }  
    }  
}
```

泛型接口

- 上面的代码非常直观：
 1. 首先，类FootballClubPrinter 具有的类型是泛型T
 2. 其次，泛型T继承了接口：IFootballClub
 3. 再次，类FootballClubPrinter又实现了接口IFootballClubPrinter
 4. 然后，接口IFootballClubPrinter所需要的类型也是继承了接口IFootballClub的泛型T
 5. 最后，直接给接口IFootballClubPrinter指定了泛型T
- 接口最终的作用就是定义了泛型类的样子，并且保护了我们在使用泛型类的时候，需要满足的条件
- 假设，我们像下面这样修改类FootballClubPrinter的类型：

泛型接口

```
class FootballClubPrinter<T>
    implements IFootballClubPrinter <T> {}
```

- 我们把类FootballClubPrinter的类型修改为T，然后再去实现接口IFootballClubPrinter的时候，TS编译器就会报错：
\$ error TS2344: Type 'T' does not satisfy the constraint 'IFootballClub'
- 错误提示告诉我们：类FootballClubPrinter要想实现接口IFootballClubPrinter，那么它的泛型类型必须满足接口IFootballClubPrinter规定的泛型类型 <T extends IFootballClub>

在泛型类中创建泛型的新对象

- 很多时候，我们需要在泛型类中创建泛型T的实例对象，看下面的代码：

```
class FirstClass {  
    id: number  
}  
class SecondClass {  
    name: string  
}  
class GenericCreator<T> {  
    create(): T {  
        return new T();  
    }  
}
```

- 代码中，定义了三个类：两个普通类 FirstClass, SecondClass 和一个泛型类：GenericCreator

在泛型类中创建泛型的新对象

```
let creator1 = new GenericCreator<FirstClass>()
let firstClass: FirstClass = creator1.create();
```

```
let creator2 = new GenericCreator<SecondClass>()
let secondClass: SecondClass = creator2.create();
```

- 然后，创建泛型类的实例creator1和creator2，然后分别调用这两个实例的create()方法，创建两个属于各自类型的变量：firstClass和secondClass
- 运行结果出错：

\$ error TS2304: Cannot find name 'T'

在泛型类中创建泛型的实例对象

- 根据TypeScript的官方文档中描述的：为了实现在泛型类中，实例化泛型的实例对象，需要引用泛型T的构造函数，同时还需要在实例化泛型类的时候，传递一个类的定义作为参数；来看一下修改后的代码：

```
class GenericCreator<T> {  
    create(arg1: { new(): T }): T {  
        return new arg1();  
    }  
}
```

- 来分解一下代码：在create方法中，我们传递了一个参数arg1；然后，定义这个参数的类型为：**{ new(): T }**（这是一个小技巧，允许我们引用泛型T的构造函数）实际上，我们是定义了一个匿名类型：重载了new()函数，并返回T类型。
- 通过这个小技巧，强类型了参数arg1必须是：包含一个单个构造函数，而且返回类型是T的类型

在泛型类中创建泛型的实例对象

- 经过改造后的类GenericCreator，也意味着传入方法create的参数必须是一个类的定义；

```
let creator1 = new GenericCreator<FirstClass>()
let firstClass: FirstClass = creator1.create(FirstClass);
```

```
let creator2 = new GenericCreator<SecondClass>()
let secondClass: SecondClass = creator2.create(SecondClass);
```

- 可以看到代码中，我们需要传递泛型T的类定义给create方法做为参数；这样修改后的代码，就可以实现我们最初的目标：在泛型类中创建泛型的实例对象；

异步编程

- 本节讲述TS的异步编程特点，包括：promises, async/await

Promises

- 在JS的编程中，经常要调用一个函数，而函数执行的最终结果要过一会才能得到。这种就是异步的情况。传统的JS是采用回调函数来处理异步的情况
- 不幸的是，在用回调函数来处理异步问题的时候，经常会造代码复杂和重复
- promises就是来解决JS处理异步问题的时候，让代码简洁的一种方法
- 先来看一下，在JS中使用回调函数的情况，看下面的代码：

异步编程

```
function delayedResponseWithCallback(callback: Function) {  
    function delayedAfterTimeout() {  
        console.log(`delayedAfterTimeout`);  
        callback();  
    }  
    setTimeout(delayedAfterTimeout, 1000);  
}  
function callDelayedAndWait() {  
    function afterWait() {  
        console.log(`afterWait`);  
    }  
    console.log(`calling delayedResponseWithCallback`);  
    delayedResponseWithCallback(afterWait);  
    console.log(`after calling delayedResponseWithCallback`);  
}  
callDelayedAndWait()
```

异步编程

- 上面的回调函数代码，会输入如下结果：

\$ calling delayedResponseWithCallback

\$ after calling delayedResponseWithCallback

\$ delayedAfterTimeout

\$ afterWait

- 这种标准的JS回调函数代码，随着功能的复杂增加，代码也会变的越来越复杂难懂

Promises 语法

- 一个promises是一个对象，通过给new一个Promise类，并且给Promise类的构造函数传入一个函数参数。而这个函数参数接受两个回调函数作为参数。看代码：

异步编程

```
function fnDelayedPromise(  
    resolve: () => void,  
    reject: () => void  
) {  
    function afterTimeout() {  
        resolve();  
    }  
    setTimeout(afterTimeout, 2000);  
}
```

```
function delayedResponsePromise(): Promise<void> {  
    return new Promise<void>(fnDelayedPromise);  
}
```

异步编程

- 上面的代码中，定义了一个函数：fnDelayedPromise，这个函数接收两个回调函数作为参数，这两个回调函数分别是：reslove, reject，返回值都是void。
- 在函数：fnDelayedPromise的函数体里，调用了setTimeout，在延迟2秒之后，执行resolve回调函数；
- 接下来，我们创建了一个函数：delayedResponsePromise，在这个函数中我们返回了一个promise对象。在函数体中，我们只是new了一个Promise类，并且为这个类的构造函数传入了之前定义的fnDelayedPromise函数作为参数；
- 代码看着比较啰嗦，实际使用promise并不这样写，但是上面代码解释了两个重要概念：
 1. 要使用promise必须要返回一个promise对象。
 - 2.一个promise对象是通过一个带有两个参数的函数构造出来的。
- 来看一下实际使用promise的代码：

异步编程

```
function delayedPromise(): Promise<void> {
    return new Promise<void>(
        (
            resolve: () => void,
            reject: () => void
        ) =>
        {
            function afterTimeout() {
                resolve();
            }
            setTimeout(afterTimeout, 1000);
        }
    );
}
```

- 实际使用promise的代码中，直接传递了一个箭头函数给new Promise时，类的构造函数，这样代码看起来比较简洁

promise的使用

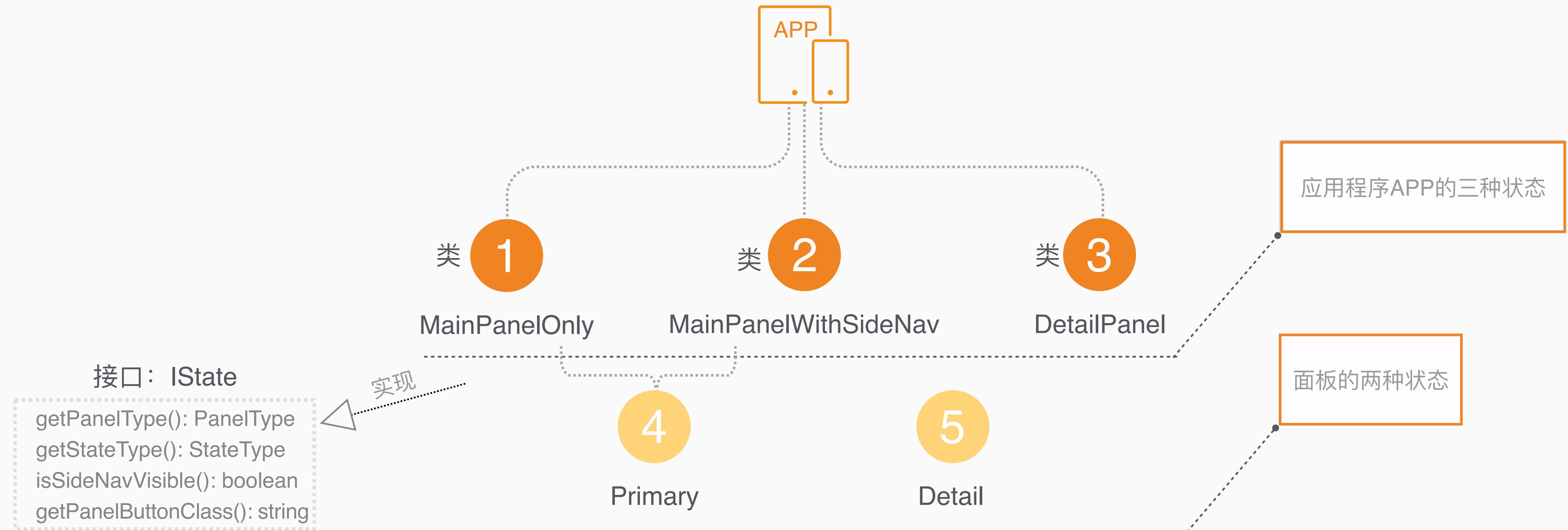
- promise 提供了非常简便的语法来访问resolve, reject函数。我们来看一下promise如何使用：

```
function callDelayedPromise() {  
  console.log(`calling delayedPromise`);  
  delayedPromise().then(  
    () => { console.log(`delayedPromise.then()`) }  
  );  
}  
callDelayedPromise()
```

\$ **calling delayedPromise**

\$ **delayedPromise.then()**

- 代码中定义了



- 面向对象编程：Mediator模式

212

根据APP状态类型，返回对应的State对象 `getStatelmpl()`
返回主面板时，之前的主面板状态 `getCurrentMainPanelState()`
比较当前State对象和下一个State对象，切换状态 `moveToState()`

2



实例化三种State对象

`private _mainPanelState`

`private _sideNavState`

`private _detailPanelState`

1

调用`IMediatorImpl`接口提供的功能，来实现应用状态的改变

3

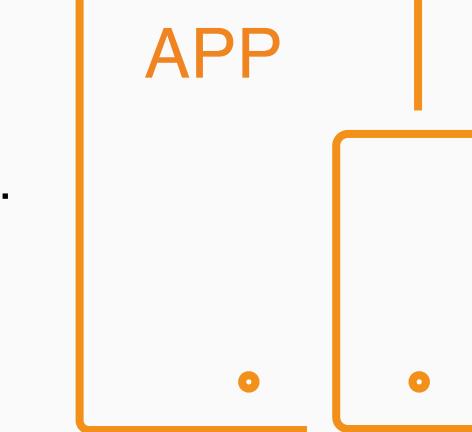
`private _currentState`
存储APP的当前状态 & `private _currentMainPanelState`
当前APP状态下主面板状态
以便保证返回之前状态时，UI的正确性

接口：`IMediatorImpl`

```
showNavPanel();  
hideNavPanel();  
showDetailPanel();  
hideDetailPanel();  
changeShowHideSideButton();
```

实现

APP启动



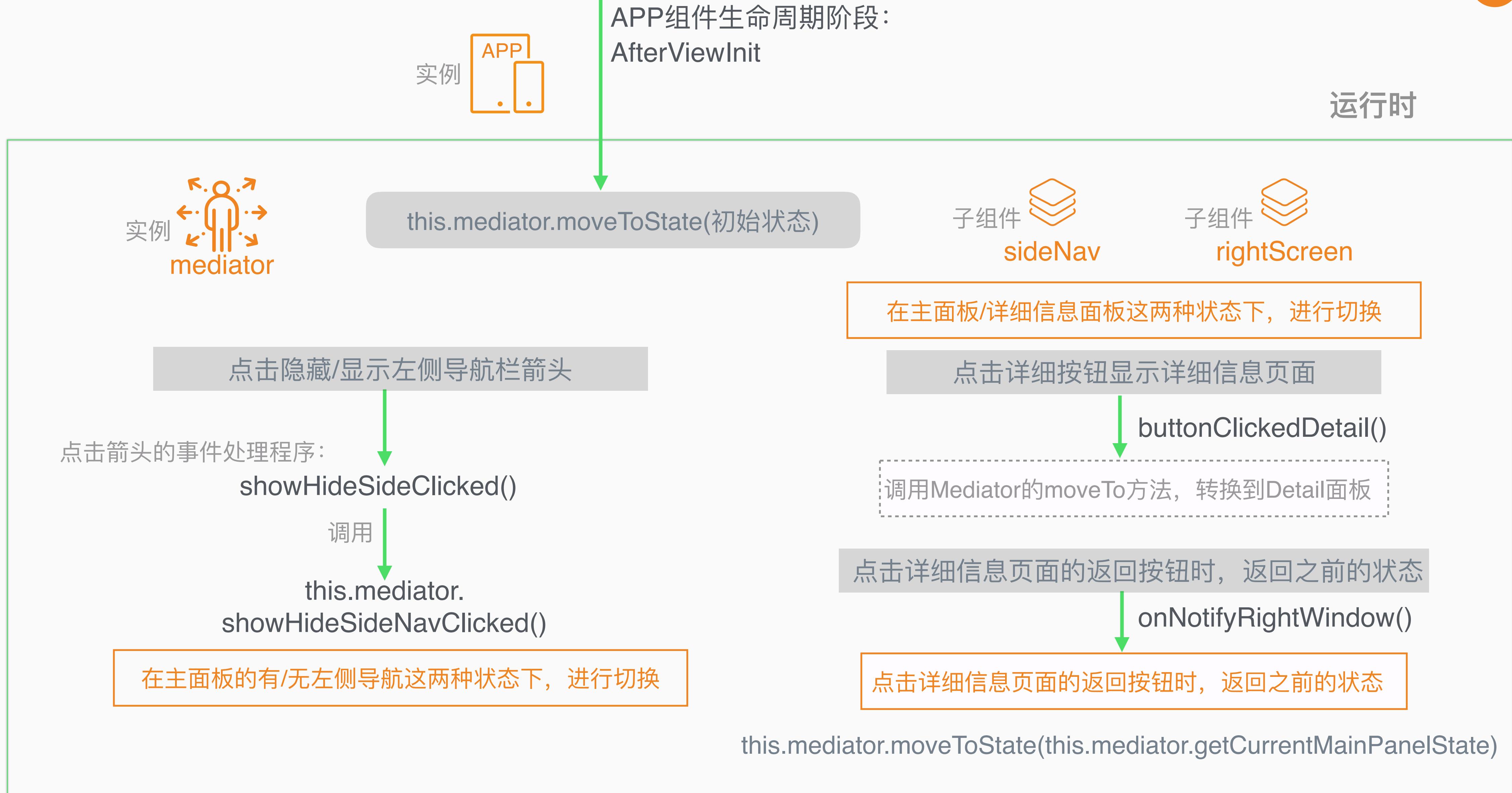
App组件上的notify事件处理函数响应
`onNotifyRightWindow()`

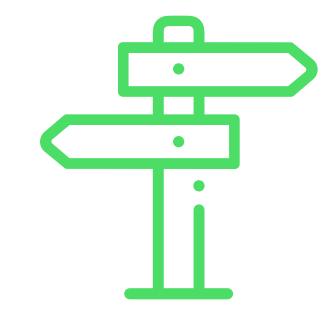
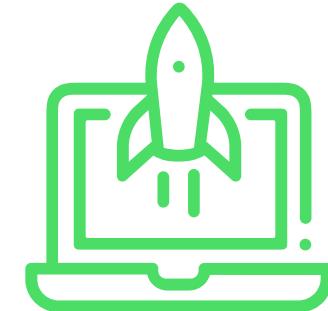
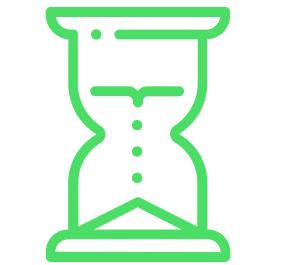
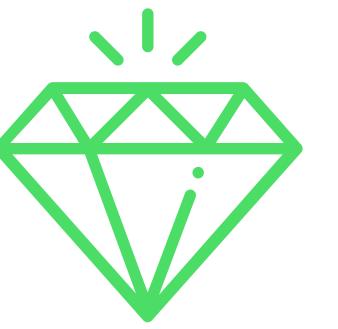
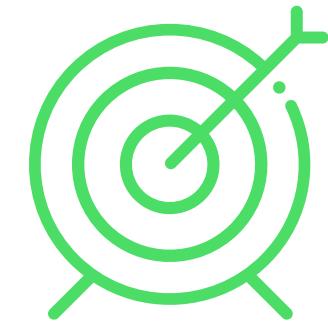
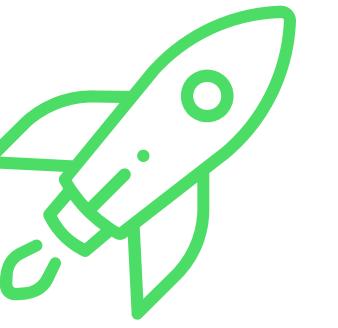
组件类自己的方法

```
CloseNav();  
ShowNav();  
closeRightWindow();  
openRightWindow();  
closeClicked();
```

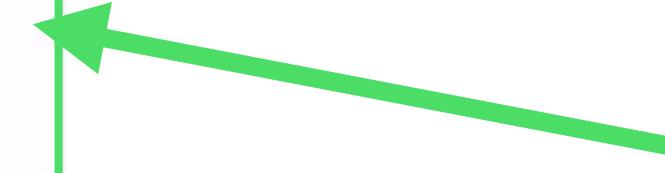
事件发射器，发出：notify事件

- 面向对象编程：Mediator模式





_transporter: nodemailer.Transporter



```
nodemailer
{
  class Transporter{
  }
  方法: createTransport() {
    return new Transporter()
  }
}
```

调用这个方法： nodemailer.createTransport(`smtp://localhost:1025`) 返回了：

nodemailer.Transporter 类型的对象 {}