typescript

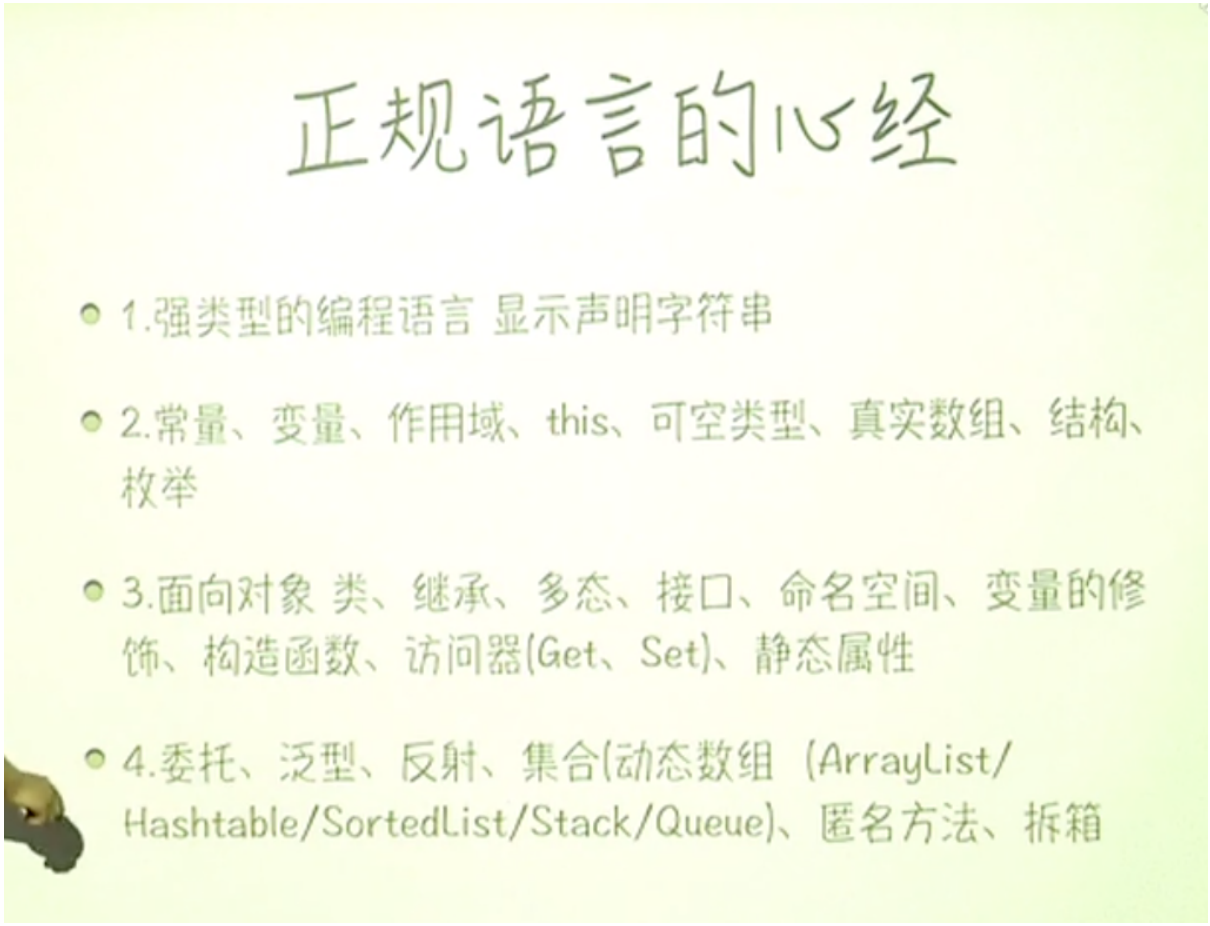
微软开发的

加入注释

可选的静态类型 和 基于 类的面向对象编程

常量 变量 作用域 this 可空类型 真实数组 结构 枚举

接口只负责声明 不实现



5.多线程

高级预处理

绑定事件是异步的。

安装命令

npm isntall -g typescript

编译命令

基础类型

Boolean、Number、Sting、Array、Enum、Any、Void

```
var list:number[] = [1,2,3]

var list2:Array<string> = ['sjh','shj']

function tell(){
    alert(list2[0])
}
tell()
```

enum 枚举类型 给定限定的范围

```
enum Color {Red = 10, Green = 20, Blue = 5};
var colorName:string = Color[10]
function tell(){
    alert(colorName)
    // alert(c)
}

// var c:Color = Color.Green;

tell()
```

给定的保留字或者关键字在ts中不能限定类型 否则会抛错

var list: Array<string> = ['sjh', 'shj']

```
function tell():string{
    return "hello"
}

function say():number{
```

```
    return 10
}

function tell():void{

}
```

函数

```
function add(x:number, y:number):string{
    return "hello sjh"
}

var myAdd = function (x: number, y: number): string {
    return "hello sjh"
}


var myAddts:(name:string, age:number) => number = function(n:string,a:number){
    return a
}
```

可选参数 与默认参数

```
function buildName(firstName: string, lastName?:string):string{
    return 'sjh'
}
var result = buildName('s', 'j'. 'h')
```

可变参数

```
function peopleName(firstName:string, ...resultName:string[]){
    return firstName + " " + resultName.join(" ")

}


var pn = peopleName('sjh' ,'sjj', 'shj', 'shh')
alert(pn)
```

重载

```
function attr(name:string):string;
function attr(age:number):number;
function attr(nameorage:any):any{
    if(nameorage && typeof nameorage === 'string'){
        alert('name')
    }
}
attr('Hello');
```

类的创建 继承（super）

跟es6没什么不同

访问修饰符

默认 public

private

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

new Animal("Cat").name; // Error: 'name' is private;

要注意一点  this.name = theName；跟我们以往理解的不一样
她是从右到左
```

```
class Hello{
    private _name:string ;
    constructor(theName:string){ this._name = theName}
    tell(){
        return this._name
    }
    get name():string{
```

```
        return this._name
    }
    set name(newname:string){
        this._name = newname
    }
}

不加constructor会报错
属性“_name”没有初始化表达式，且未在构造函数中明确赋值。


加了constructor之后
var h = new Hello(11);  //里面必须要放值
```

封装的实现

```
class Greeter{
    greeting:string;
    constructor(message:string){
        this.greeting = message;
    }
    greet(){
        return 'Hello,' + this.greeting;
    }
}

var green:Greeter;
green = new Greeter('iwen');
alert(green.greet())
```

static的使用技巧

```

```

接口

```
interface LabelValue {
    label: string;
}

function printeLabel(labelObj: LabelValue) {
    console.log(labelObj.label)
}

var myObj = { label: 'Hello' };
printeLabel(myObj);
```

可选属性

```
interface USB {
    name?: string;
    age?: number;
}

function printUSB(pu: USB) {
    console.log(pu.name);
    console.log(pu.age);
}

var my = printUSB({ name: "ime", age: 1000})
console.log(my)
```

函数类型

```
interface SearchFunc {
    (source: string, subString: string): boolean;

}

var mySearch: SearchFunc;
mySearch = function (src: string, sub: string) {
    var result = src.search(sub);
    if (result != -1) {
        return true;
    } else {
        return false
    }
}
```

数组类型

```
//数组类型
interface StringArray {
    [index: number]: string
}

var myArray: StringArray;
myArray = ['sjh', 'shh']
alert(myArray[1])
```

class类型

```
class HelloNumber<T>{
    Ten: T;
    add: (x: T, y: T) => T;
}

var myHelloNumber = new HelloNumber<string>();
myHelloNumber.Ten = 'Hello';
myHelloNumber.add = function (x, y) {
    return x + y
}
alert(myHelloNumber.Ten);
alert(myHelloNumber.add('Hello', 'sjh'))
```

继承接口

```
//接口的继承
interface Shape {
    color: string
}
interface PenStorke {
    penWidth: number
}
interface Square extends Shape, PenStorke {
    sideLength: number
}
var s = <Square>{};
s.color = "blue";
s.penWidth = 100;
s.sideLength = 10;
```

混合类型

```
混合类型
interface Counter{
    interval:number;
    reset():void;
    (start:number):string
}

var c:Counter;
c(10);
c.reset();
```

泛型

```
function Hello1<T>(arg: T): T {
    return arg;
}
var output = Hello1<string>("Hello sjh");
alert(output);
```

```
function Hello2<T>(string: T[]): T {
    alert(string.length)
    return string;
}
//报错   不能将泛型T[]分配给 T
//正确
function Hello2<T>(string: T[]): T[] {
    alert(string.length)
    return string;
}
```

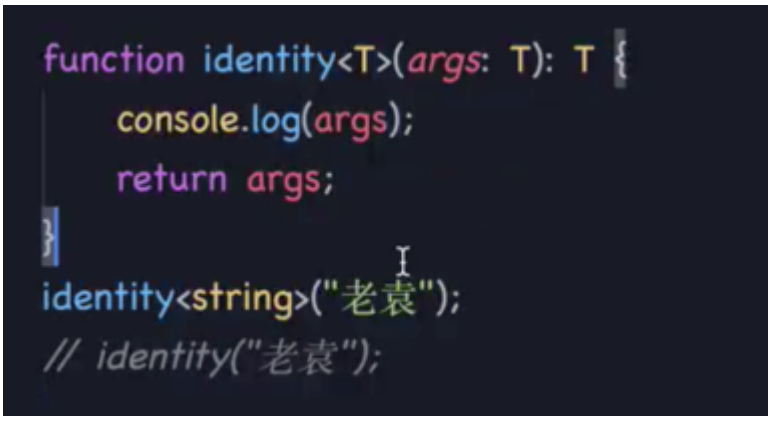
```
function Hello3<T>(arg: T): T {
    return arg;
```

```
}

var myHello: <K>(arg: K) => K = Hello3;
alert(myHello('hello'))


var myHello: { <T>(arg: T): T } = Hello3;
alert(myHello('Hello'))
```

接口与泛型

泛型是跟在后面

```
interface Hello4{
    <T>(arg:T):T
}

function myHello<T>(arg:T):T{
    return arg;
}

var MH:Hello4 = myHello;
alert(MH('Hello'))
```




```
interface Hello5<T>{
    (arg:T):T
}
```

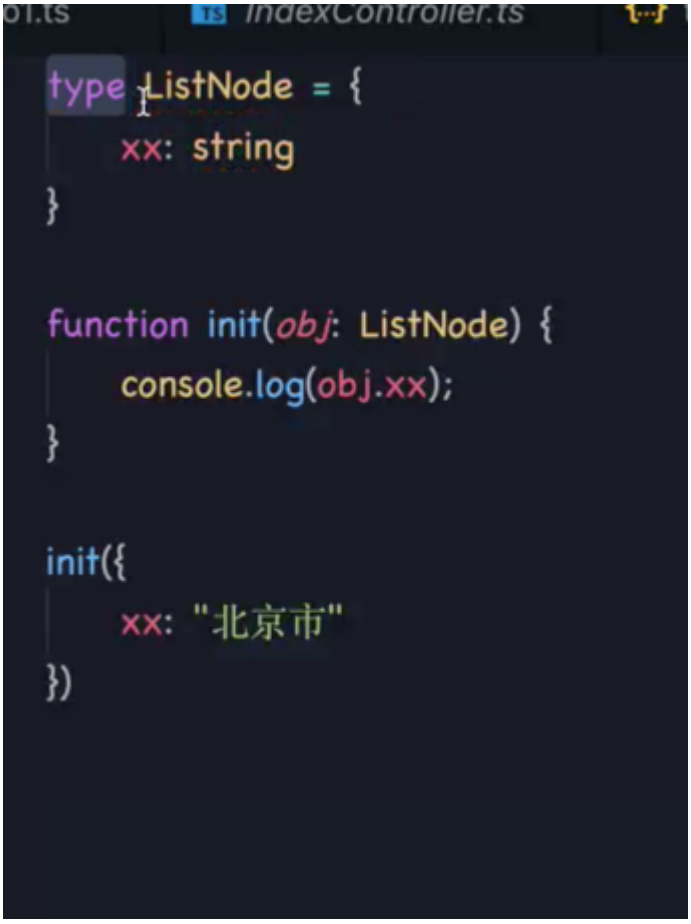
```
function myHello<T>(arg: T): T {
    return arg;
}

var MH:Hello5<number> = myHello;
alert(MH(100))
```

namespace

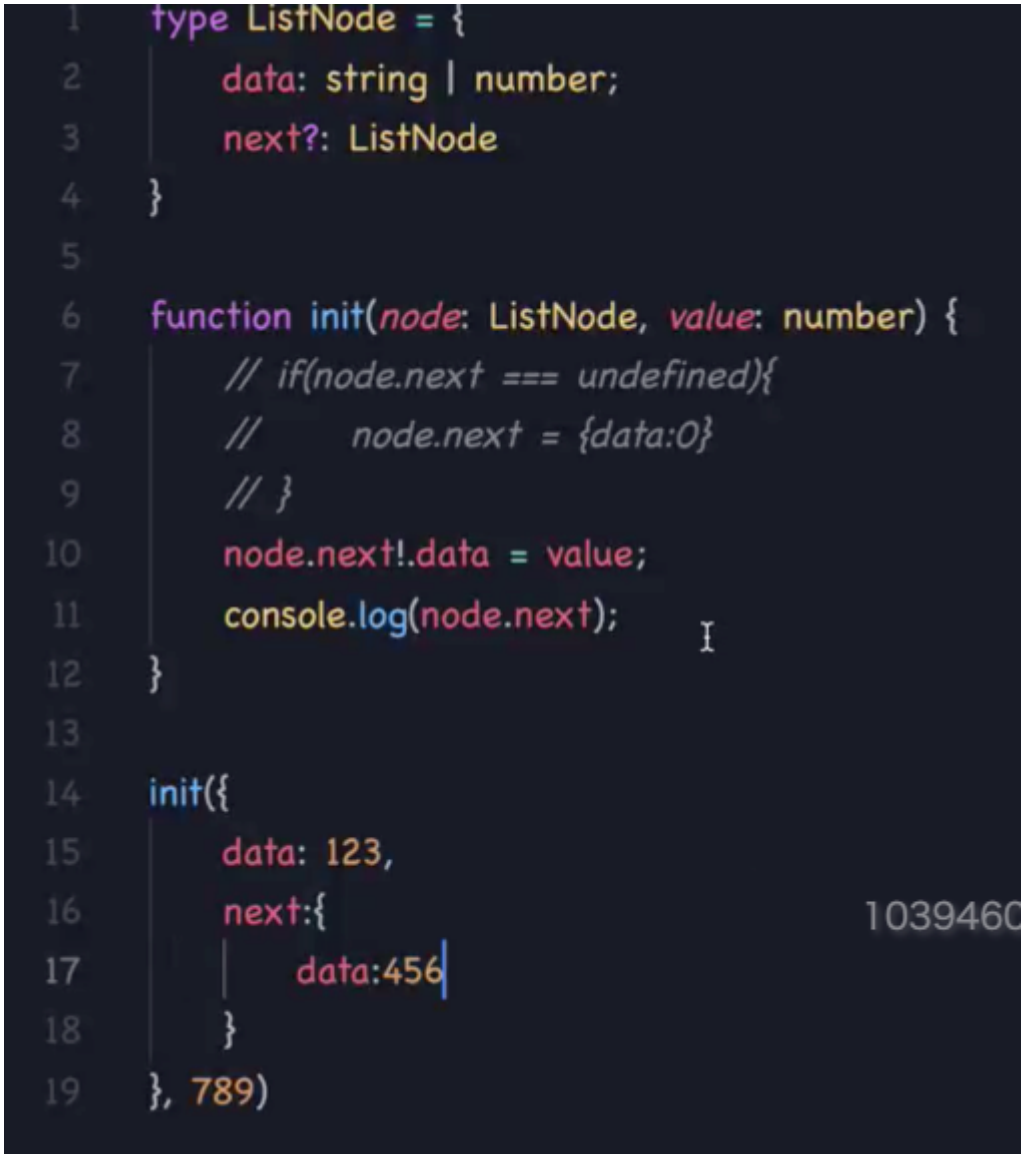
```
```

装饰器

tsc --target ES5 --experimentalDecorators


indexController.ts
type ListNode = {
xx: string
}
function init(obj: ListNode) {
console.log(obj.xx);
}
init({
xx: "北京市"
})

ts小技巧



type-node

枚举： 一组有名字的常量集合

枚举的原理反映射

```
enum  Role{
  Reporter ,
  Developer,
  Maintainer,
  Owner,
  Guest
}
```

```
var Role;
(function (Role) {
    Role[Role["Reporter"] = 1] = "Reporter";
    Role[Role["Developer"] = 2] = "Developer";
    Role[Role["Maintainer"] = 3] = "Maintainer";
    Role[Role["Owner"] = 4] = "Owner";
    Role[Role["Guest"] = 5] = "Guest";
})(Role || (Role = {}));
```

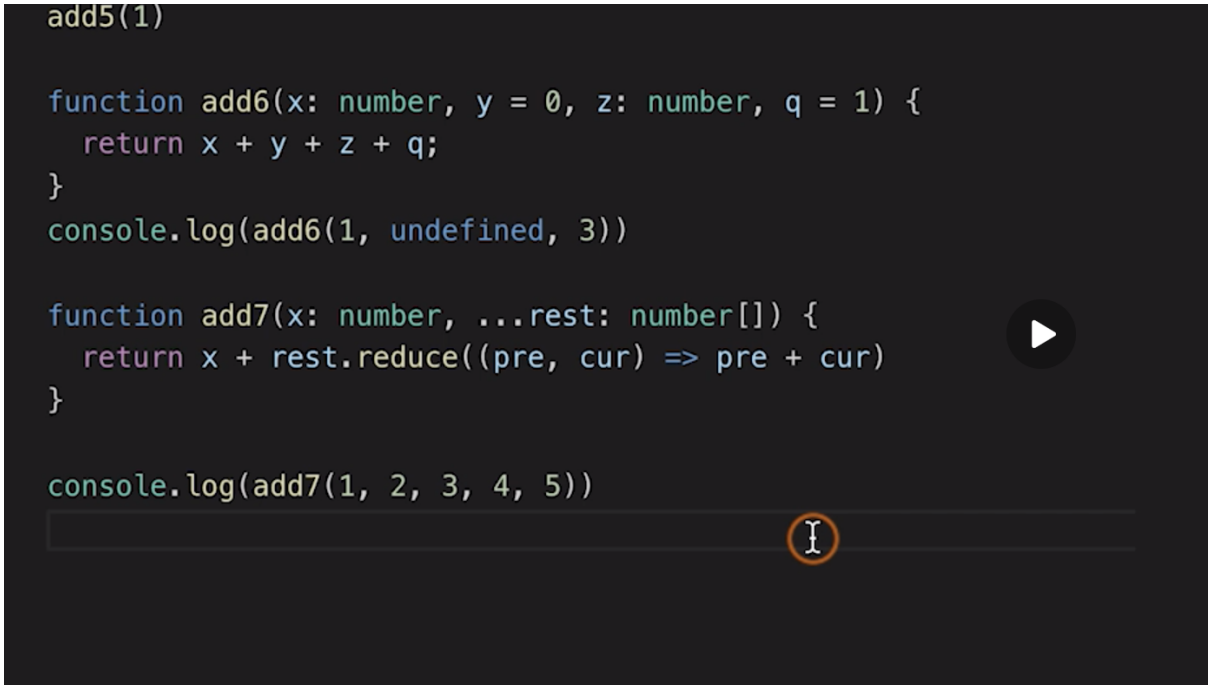
字符串枚举 无反向

```
enum Message{
  success =  'ok',
  fail  = 'no'
}
```

```
var Message;
(function (Message) {
    Message["success"] = "ok";
    Message["fail"] = "no";
})(Message || (Message = {}));
```

接口

函数



可选参数 默认参数 的规则 默认参数前面必须要放在可选参数前面

重载函数

```
//函数重载
function add8(...rest: number[]): number;
function add8(...rest: string[]): string;

function add8(...rest:any):any{
  let first = rest[0];
  if(typeof first ==='string'){
    return rest.join()
  }
  if(typeof first === 'number'){
    return rest.reduce((pre, cur)=>pre+cur)
  }
}
```

类型推断

不需要指定变量的类型（函数的返回值类型）， typescript 可以更具某些规则自动地为其推断出一个类型

1.基础类型推断

2.最佳通用类型推断

3.上下文类型推断

与 ts-loader的主要区别

1） 更适合与babel集成， 使用babel的转义和缓存

2）不需要安装额外的插件， 就可以把类型检查放在独立进程中进行

babel只做语言转换

ts 只做类型检查

React + typeScript 出现的情形

必须为每个函数组件 的属性 指定类型 和 类组件 的属性和状态指定类型 指定类型

组件复用方式	优势	劣势	状态
类组件（class）	发展时间长，接受度广泛	只能继承父类	传统模式，长期存在
Mixin	可以复制任意对象的任意多个方法	组件相互依赖、耦合，可能产生冲突，不利于维护	被抛弃
高阶组件（HOC）	利用装饰器模式，在不改变组件的基础上，动态地为其添加新的能力	嵌套过多调试困难，需要遵循某些约定（不改变原始组件，透传props等）	能力强大，应用广泛
Hooks	代替 class，多个 Hooks 互不影响，避免嵌套地狱，开发效率高	切换到新思维需要成本	React 的未来

mixins 做个占位 和 混合