Git命令下载（直接进入目录）：

git clone <http://gitlab.alibaba-inc.com/damai-mz/mz-common.git>

git bush 创建项目：

winpty vue.cmd create [program name]



**基础类型：**

// 布尔值

let isDone: boolean = false;

// 数字

let decLiteral: number = 6;

// 字符串

let name: string = "bob";

name = "smith";

// 模版字符串，它可以定义多行文本和内嵌表达式。 这种字符串是被反引号包围（ `），并且以${ expr }这种形式嵌入表达式

let name: string = `Gene`;

let age: number = 37;

let sentence: string = `Hello, my name is ${ name }.

I'll be ${ age + 1 } years old next month.`;

// 数组

let list:number[] = [2,3,4];

// 数组泛型，Array<元素类型>

let list1:Array<number> = [1,2,3];

// 元组

let x:[string,number] = ["hello",10];

// enum类型是对JavaScript标准数据类型的一个补充。 像C#等其它语言一样，使用枚举类型可以为一组数值赋予友好的名字。

enum Color1 {Red, Green, Blue}

let c1: Color1 = Color1.Green;

// 默认情况下，从0开始为元素编号。 你也可以手动的指定成员的数值。 例如，我们将上面的例子改成从 1开始编号：

console.log(c1);

enum Color2 {Red = 1, Green, Blue}

let c2: Color2 = Color2.Green;

// 或者，全部都采用手动赋值：

console.log(c2);

enum Color3 {Red = 1, Green = 2, Blue = 4}

let c3: Color3 = Color3.Green;

// 枚举类型提供的一个便利是你可以由枚举的值得到它的名字。 例如，我们知道数值为2，但是不确定它映射到Color里的哪个名字，我们可以查找相应的名字：

console.log(c3);

enum Color4 {Red = 1, Green, Blue}

let colorName: string = Color4[2];

console.log(colorName);  // 显示'Green'因为上面代码里它的值是2

// Any 任意类型

let notSure:any = 4;

notSure = "hello";

let list:any[] = [1,true,"100"];

// void 表示没有任何类型

function warnUser():void {

  console.log("This is my warning message!");

}

let unusable:void = undefined;

// null 和 undefined

let u: undefined = undefined;

let n: null = null;

// never 表示永远不存在的值的类型

function error(message: string): never {

  throw new Error(message);

}

function fail() {

  return error("Something failed");

}

function infinteLoop(): never {

  while(true) {}

}

// object

declare function create(o:object|null):void;

create({prop:0}); //create is not defined

create(null); //create is not defined

// 类型断言

let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;

// as语法

let strLength1: number = (someValue as string).length;

console.log(strLength,strLength1);

// 块级作用域变量需要在明显不同的块里声明

function f(condition:any,x:any){

  if(condition) {

    let x= 100;

    return x;

  }

  return x;

}

console.log(f(false,0));

console.log(f(true,0));

// const 不可重新赋值

const num:number = 9;

const kitty = {

  name:"auto",

  num:9

}

kitty.name='haha';

// 解构作用于函数参数

function f([first,second]:[number,string]){

  console.log(first);

  console.log(second);

}

let ipt:[number,string]=[1,"2"];

f(ipt);

// 数组里使用**...**语法创建剩余变量

let [first,...rest] = [1,2,3,4];

console.log(first);// 1

console.log(rest);// [2,3,4]

let [,second,,fourth] = [1,2,3,4];

console.log(second,fourth);//[2,4]

// 对象解构

// 通过 o.a and o.b 创建了 a 和 b

let o = {

  a:"foo",

  b:12,

  c:"bar"

}

let {a,b}=o;

console.log(a);// 'foo'

console.log(b);// 12

// 没有声明的赋值

({a,b} = {a:"baz",b:101})

console.log(a);

console.log(b);

// ...语法创建剩余变量

let {a,...rest} = o;

let total = rest.b + rest.c.length;

console.log(a,total)

// 属性重命名

let {a:name1,b:name2}= o;// => let name1=o.a;

console.log(name1);

// 指定类型

let {a,b}:{a:string,b:number}= o;

console.log(a);

// 默认值

function obj(wholeObj: {a:string, b?:number}) {

  let {a, b =1001} = wholeObj;

  console.log(a,b);

}

obj({a:'haha',b:100});// 'haha' 100

obj({a:'haha',b:undefined});// 'haha' 1001

// 解构用于函数声明

type c = {a:string,b?:number} //定义一个类型C

function f({a,b}:c):void {

}

f({a:"123",b:45})

// 展开

let first = [1,2];

let second = [3,4];

let both = [0,...first,444,...second];

console.log(both); //展开操作创建了 first和second的一份浅拷贝

// 展开一个对象实例时，你会丢失其方法：

class C {

  p = 12;

  m() {

  }

}

let c = new C();

let clone = { ...c };

clone.p; // ok

clone.m(); // error!

**接口：**

// interface 定义属性用法

interface def\_val{

  label: string;

}

function printVal(valobj:def\_val){

  console.log(valobj.label)

}

let myobj = {size:10,label:'size 10'};

printVal(myobj) // sieze 10

// 可选属性

interface squareConfig{

  color?:string;

  width?:number;

}

function createSquare(config:squareConfig):{color:string;area:number} {

  let newSquare={color:"white",area:100};

  if(config.color){

    newSquare.color=config.color;

  }

  if(config.width){

    newSquare.area=config.width\*config.width;

  }

  return newSquare;

}

let mySquare = createSquare({color:"black"});

console.log(mySquare); //color: "black", area: 100

// 只读属性

interface point{

  readonly x:number;

  readonly y:number;

}

let p1:point = {x: 10, y: 10};

p1.x = 5; //error!!

// 数组只读

let a: number[] = [1,2,3,4];

let ro: ReadonlyArray<number> = a;

ro[0] = 12;// error!

ro.push(5);// error!

ro.length = 100;// error!

a = ro;// error!

a=ro as number[];// 类型断言重写

console.log(a);

// 最简单判断该用readonly还是const的方法是看要把它做为变量使用还是做为一个属性。 做为变量使用的话用 const，若做为属性则使用readonly。

// 绕开额外属性检查,使用类型断言

let mySquare = createSquare({width:100,opacity:0.5} as squaryConfig);

// 字符串索引签名

interface squareConfig{

  color?: string;

  width?: number;

  [propName:string]: any;

}

// 函数类型 参数列表和返回值

interface SearchFunc{

  (source:string, subString:string): boolean;

// 参数列表和返回值

}

let mySearch: SearchFunc;

mySearch=function(source:string, subString:string){

  let result= source.search(subString);

  return result > -1;

}

let a = mySearch('hahahah345','b');

console.log(a); // false

// 可索引类型

interface StringArray {

  [index: number]: string;

  // 这个索引签名表示了当用 number去索引StringArray时会得到string类型的返回值。

}

let myArray: StringArray;

myArray=["bob","fred"];

let myStr:string=myArray[0];

console.log(myStr);