TypeScript

1. 基础类型

1.1. 布尔 boolean

1.2. 数字 number

1.3. 字符串 string

1.4. 数组

两种定义的实现方式

1、在元素类型后面接上 []，表示由此类型元素组成的一个数组

```
let list: number[] = [1, 2, 3];
```

2.使用数组泛型，Array<元素类型>

```
let list: Array<number> = [1, 2, 3];
```

泛型百度百科：

泛型是程序设计语言的一种特性。允许程序员在强类型程序设计语言中编写代码时定义一些可变部分，那些部分在使用前必须作出指明。

泛型类是引用类型，是堆对象，主要是引入了类型参数这个概念。

1.5. 元组

允许表示一个已知元素数量和类型的数组，各元素的类型不必相同。

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ['hello', 10]; // OK
// Initialize it incorrectly
x = [10, 'hello']; // Error
```

1.6. 枚举

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 Color {Red, Green, Blue}
let c: Color = Color.Green;
```

1.7. 任意类型 any

1.8. void null undefined

默认情况下 null 和 undefined 是所有类型的子类型。 就是说你可以把 null 和 undefined 赋值给 number 类型的变量。

指定了 --strictNullChecks 标记，null **和** undefined **只能**赋值给 void **和它**们**各自**

1.9. 永不存在的值 never

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

```
// 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) {
    }
}
```

1.10. 对象类型 object

表示除number，string，boolean，symbol，null或undefined之外的类型

1.11. 类型断言

通过类型断言这种方式可以告诉编译器，“相信我，我知道自己在干什么”。

类型断言好比其它语言里的类型转换，但是不进行特殊的数据检查和解构。
它没有运行时的影响，只是在编译阶段起作用。TypeScript 会假设你，程序员，已经进行了必须的检查。

两种声明方式
1、尖括号方式
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
2、as 方式
let someValue: any = "this is a string";

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

ps:在jsx中时只有as方式有效，建议使用as

2. 接口

2.1. 属性

2.1.1. 可选属性

```
interface SquareConfig {
  color?: string;
  width?: number;
}
```

2.1.2. 只读属性

```
interface Point {
    readonly x: number;
    readonly y: number;
}
```

2.2. 函数类型

为了使用接口表示函数类型，我们需要给接口定义一个调用签名。
它就像是一个只有参数列表和返回值类型的函数定义。参数列表里的每个参数都需要名字和类型。
interface SearchFunc {

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

2.3. 可索引的类型

可索引类型具有一个 索引签名，它描述了对象索引的类型，还有相应的索引返回值类型。

```
interface StringArray {
  [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

2.4. 类类型

用它来明确的强制一个类去符合某种契约。

```
interface ClockInterface {
    currentTime: Date;
}

class Clock implements ClockInterface {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

implements 一个类实现一个接口用的关键字.实现一个接口，必须实现接口中的所有方法：

怎么理解：用构造器签名去定义一个接口并试图定义一个类去实现这个接口时会得到一个错误？

```
interface ClockConstructor {
    new (hour: number, minute: number);
}

class Clock implements ClockConstructor {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

A:对于constructor部分的内容在编译时不会读取，只会读取除其之外的部分，上面的Clock错误之处在于读取签名时没有找到hour与minute变量.

```
interface ClockConstructor {
    new (hour: number, minute: number);
}
```

以”new”关键字为前缀，意味着函数必须作为构造器来调用

名词解释:函数签名(或者类型签名，抑或方法签名)定义了 函数或方法的输入与输出。

2.5. 继承接口　一个接口可以继承多个接口，创建出多个接口的合成接口

```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

2.6. 混合类型

一个对象可以同时做为函数和对象使用，并带有额外的属性

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

```
function getCounter(): Counter {
    let counter = <Counter>function (start: number) { };
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

2.7. 特殊类 接口继承类

当接口继承了一个类类型时，它会继承类的成员但不包括其实现。

当你创建了一个接口继承了一个拥有私有或受保护的成员的类时，这个接口类型只能被这个类或其子类所实现（implement）
(理解：具有私有属性的类被接口继承时意味着想要实现此接口的类必须实现私有属性，所以只有私有属性的子类才能实现此接口，如下所示)

```
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}

class Button extends Control implements SelectableControl {
    select() { }
}

class TextBox extends Control {
    select() { }
}

// 错误：“Image”类型缺少“state”属性。
class Image implements SelectableControl {
    select() { }
```

}

3. 类 ES6概念 待补充语法

4. 泛型

4.1. 定义 类型变量，它是一种特殊的变量，只用于表示类型而不是值

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

我们给identity添加了类型变量T。

T帮助我们捕获用户传入的类型（比如：number），之后我们就可以使用这个类型。 之后我们再次使用了T当做返回值类型。现在我们可以知道参数类型与返回值类型是相同的了。

这允许我们跟踪函数里使用的类型的信息。

我们把这个版本的identity函数叫做泛型，因为它可以适用于多个类型。 不同于使用any，它不会丢失信息，像第一个例子那像保持准确性，传入数值类型并返回数值类型。

两种使用方式

一、传入所有的参数，包含类型参数

```
let output = identity<string>("myString");  // type of output will be 'string'
```

二、利用了类型推论 -- 即编译器会根据传入的参数自动地帮助我们确定T的类型

```
let output = identity("myString");  // type of output will be 'string'
```

名词解释：泛型是程序设计语言的一种特性。允许程序员在强类型程序设计语言中编写代码时定义一些可变部分，那些部分在使用前必须作出指明。

ps:个人理解，相当于在实现功能间加了一层过滤器，在不修改功能代码功能的情况下，修改其他的功能。

4.2. 泛型变量

```
function loggingIdentity<T>(arg: T[]): T[] {
    console.log(arg.length);  // Array has a .length, so no more error
    return arg;
}
```

可以这样理解loggingIdentity的类型：

泛型函数loggingIdentity，接收类型参数T和参数arg，它是个元素类型是T的数组，并返回元素类型是T的数组。

如果我们传入数字数组，将返回一个数字数组，因为此时 T的的类型为number。

这可以让我们把泛型变量T当做类型的一部分使用，而不是整个类型，增加了灵活性。

另一种写法：

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
    console.log(arg.length);  // Array has a .length, so no more error
    return arg;
}
```

4.3. 泛型类

看上去与泛型接口差不多。 泛型类使用（ <>）括起泛型类型，跟在类名后面。

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

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

* 类有两部分：静态部分和实例部分。

泛型类指的是实例部分的类型，所以类的静态属性不能使用这个泛型类型。

4.4. 泛型约束

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // Now we know it has a .length property, so no more error
    return arg;
}

loggingIdentity(3);  // Error, number doesn't have a .length property
```

```
loggingIdentity({length: 10, value: 3});
```

4.5. 使用类类型

```
class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}
// A 类型继承于Animal类同时参数c只能用于构造函数且返回类型与A相同，最后返回值类型与A一致
function createInstance<A extends Animal>(c: new () => A): A {
    return new c();
}

createInstance(Lion).keeper.nametag;  // typechecks!
createInstance(Bee).keeper.hasMask;   // typechecks!
```

5. 枚举

5.1. 数字枚举

使用枚举我们可以定义一些带名字的常量。 使用枚举可以清晰地表达意图或创建一组有区别的用例。

```
enum Direction {
```

```
    Up = 1,
    Down,
    Left,
    Right
}
```

定义了一个数字枚举，Up使用初始化为 1。其余的成员会从 1开始自动增长。换句话说，Direction.Up的值为 1，Down为 2，Left为 3，Right为 4。

如果Up不为1，则使用默认的增长规则：Up从0开始，后面相同。

不带初始化器的枚举或者被放在第一的位置，或者被放在使用了数字常量或其它常量初始化了的枚举后面。换句话说，下面的情况是不被允许的。

```
enum E {
    A = getSomeValue(),
    B, // error! 'A' is not constant-initialized, so 'B' needs an initializer
}
```

5.2. 字符串枚举

在一个字符串枚举里，每个成员都必须用字符串字面量

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```

由于字符串枚举没有自增长的行为，字符串枚举可以很好的序列化。

5.3. 异构枚举： 枚举可以混合字符串和数字成员

5.4. 计算的和常量成员

常量枚举表达式:

一个枚举表达式字面量（主要是字符串字面量或数字字面量）

一个对之前定义的常量枚举成员的引用（可以是在不同的枚举类型中定义的）

带括号的常量枚举表达式

一元运算符 +, -, ~其中之一应用在了常量枚举表达式

常量枚举表达式做为二元运算符 +, -, *, /, %, <<, >>, >>>, &, |, ^的操作对象。若常数枚举表达式求值后为 NaN或 Infinity，则会在编译阶段报错。

所有其它情况的枚举成员被当作是需要计算得出的值

6. 类型兼容性

6.1. 基本规则： 如果x要兼容y，那么y至少具有与x相同的属性

```
interface Named {
    name: string;
}

let x: Named;
// y's inferred type is { name: string; location: string; }
let y = { name: 'Alice', location: 'Seattle' };
x = y;
```

6.2. 函数兼容

```
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

要查看x是否能赋值给y，首先看它们的参数列表。 x的每个参数必须能在y里找到对应类型的参数。注意的是参数的名字相同与否无所谓，只看它们的类型