参考自：TS入门教程

基础部分

interface

约束对象的方法

```
Math.pow(10, 2);

interface Math {
    pow(x: number, y: number): number;
}
```

任意属性

```
interface Person {
    name: string;
    age?: number;
    [propName: string]: string | number;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};
```

只读属性

```
interface Person {
    readonly id: number;
    name: string;
    age?: number;
    [propName: string]: any;
}

let tom: Person = {
    id: 89757,
    name: 'Tom',
    gender: 'male'
};

tom.id = 9527;  // error
```

数组

常规表示

```
let fibonacci: number[] = [1, 1, 2, 3, 5];
```

数组泛型

```
let fibonacci: Array<number> = [1, 1, 2, 3, 5];
```

类数组

```
常用的类数组都有自己的接口定义，如 IArguments, NodeList, HTMLCollection 等：
let args: IArguments = arguments;
```

函数

函数表达式

```
注意不要混淆了 TypeScript 中的 => 和 ES6 中的 =>。
在 TypeScript 的类型定义中，=> 用来表示函数的定义，左边是输入类型，需要用括号括起来，右
边是输出类型。

// (x: number, y: number) => number
let mySum: (x: number, y: number) => number = function (x: number, y:
number): number {
    return x + y;
};
```

用接口定义函数的形状

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

let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    return source.search(subString) !== -1;
}
```

可选参数

```
可选参数必须接在必需参数后面

function buildName(firstName: string, lastName?: string) {
    if (lastName) {
        return firstName + ' ' + lastName;
    } else {
        return firstName;
    }
}
let tomcat = buildName('Tom', 'Cat');
let tom = buildName('Tom');
```

参数默认值

```
TypeScript 会将添加了默认值的参数识别为可选参数，此时就不受「可选参数必须接在必需参数后
面」的限制了：

function buildName(firstName: string = 'Tom', lastName: string) {
    return firstName + ' ' + lastName;
}
let tomcat = buildName('Tom', 'Cat');
let cat = buildName(undefined, 'Cat');
```

剩余参数

```
items 是一个数组。所以我们可以用数组的类型来定义它：

function push(array: any[], ...items: any[]) {
    items.forEach(function(item) {
        array.push(item);
    });
}

let a = [];
push(a, 1, 2, 3);
```

类型断言（非常重要且常用）

概念误区

```
类型断言只会影响 TypeScript 编译时的类型，类型断言语句在编译结果中会被删除

类型断言不是类型转换，它不会真的影响到变量的类型

若要进行类型转换，需要直接调用类型转换的方法
```

语法

`值 as 类型`

```
const tom = getCacheData('tom') as Cat;

等价于

const tom: Cat = getCacheData('tom');
```

类型断言的用途

在还不确定类型的时候就访问其中一个类型特有的属性或方法

```
interface Cat {
    name: string;
    run(): void;
}
interface Fish {
    name: string;
    swim(): void;
}

function isFish(animal: Cat | Fish) {
    // (animal as Fish).swim
    if (typeof (animal as Fish).swim === 'function') {
        return true;
    }
    return false;
}
```

将一个父类断言为更加具体的子类

```
当类之间有继承关系时，类型断言也是很常见的：

class ApiError extends Error {
    code: number = 0;
}
class HttpError extends Error {
    statusCode: number = 200;
}

(error as ApiError).code
function isApiError(error: Error) {
    if (typeof (error as ApiError).code === 'number') {
        return true;
    }
    return false;
}


当接口之间有继承关系时

interface ApiError extends Error {
    code: number;
}
interface HttpError extends Error {
    statusCode: number;
}

function isApiError(error: Error) {
    if (typeof (error as ApiError).code === 'number') {
        return true;
    }
    return false;
}
```

将任何一个类型断言为 any

```
window.foo = 1; // error

(window as any).foo = 1;
```

类型断言的限制

```
interface Animal {
    name: string;
}
interface Cat {
    name: string;
    run(): void;
}

let tom: Cat = {
    name: 'Tom',
    run: () => { console.log('run') }
};
```

```
let animal: Animal = tom;

等价于

TypeScript 并不关心 Cat 和 Animal 之间定义时是什么关系，而只会看它们最终的结构有什么
关系——所以它与 Cat extends Animal 是等价的
我们把它换成 TypeScript 中更专业的说法，即：Animal 兼容 Cat。

interface Animal {
    name: string;
}
interface Cat extends Animal {
    run(): void;
}

当 Animal 兼容 Cat 时，它们就可以互相进行类型断言了

    name: string;
}
interface Cat {
    name: string;
    run(): void;
}

function testAnimal(animal: Animal) {
    return (animal as Cat);
}
function testCat(cat: Cat) {
    return (cat as Animal);
}

允许 animal as Cat 是因为「父类可以被断言为子类」
允许 cat as Animal 是因为既然子类拥有父类的属性和方法，那么被断言为父类，获取父类的属
性、调用父类的方法，就不会有任何问题，故「子类可以被断言为父类」
```

双重断言

```
interface Cat {
    run(): void;
}
interface Fish {
    swim(): void;
}

function testCat(cat: Cat) {
    // cat as any as Fish
    return (cat as any as Fish);
}
```

类型断言 vs 泛型

```
function getCacheData(key: string): any {
    return (window as any).cache[key];
}

interface Cat {
    name: string;
    run(): void;
}

const tom = getCacheData('tom') as Cat;
tom.run();

泛型改写

function getCacheData<T>(key: string): T {
    return (window as any).cache[key];
}

interface Cat {
    name: string;
    run(): void;
}

const tom = getCacheData<Cat>('tom');
tom.run();
```

进阶部分

类型别名

```
类型别名常用于联合类型

type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === 'string') {
        return n;
    } else {
        return n();
    }
}

--------------

Partial: 定义属性可选

interface People {
  name: string;
  age: number;
}
type TPPeople = Partial<People>
export const partialTest: TPPeople = {}    // 可以不传 name，age

---------------

Omit: 忽略属性

type TOPeople = Omit<People, 'name'>
export const omitTest: TOPeople = { age: 1 }    // 可以不传 name
```

字符串字面量类型

```
type EventNames = 'click' | 'scroll' | 'mousemove';
function handleEvent(ele: Element, event: EventNames) {
    // do something
}

handleEvent(document.getElementById('hello'), 'scroll');     // 没问题
handleEvent(document.getElementById('world'), 'dblclick');   // 报
错，'dblclick' 不在 EventNames 的声明中
```

元组

```
let tom: [string, number] = ['Tom', 25];
```

枚举

类

属性和方法

```
TypeScript 可以使用三种访问修饰符（Access Modifiers），分别是 public、private 和
protected。

public 修饰的属性或方法是公有的，可以在任何地方被访问到，默认所有的属性和方法都是
public 的
private 修饰的属性或方法是私有的，不能在声明它的类的外部访问；当构造函数修饰为 private
时，该类不允许被继承或者实例化
protected 修饰的属性或方法是受保护的，它和 private 类似，区别是它在子类中也是允许被访
问的；当构造函数修饰为 protected 时，该类只允许被继承
```

```
class Animal {
  public name;
  public constructor(name) {
    this.name = name;
  }
}

let a = new Animal('Jack');
console.log(a.name); // Jack
a.name = 'Tom';
console.log(a.name); // Tom

----------

很多时候，我们希望有的属性是无法直接存取的，这时候就可以用 private 了

class Animal {
  private name;
  public constructor(name) {
    this.name = name;
  }
}

let a = new Animal('Jack');
console.log(a.name); // Jack
a.name = 'Tom';

// index.ts(9,13): error TS2341: Property 'name' is private and only
accessible within class 'Animal'.
// index.ts(10,1): error TS2341: Property 'name' is private and only
accessible within class 'Animal'.

--------------

使用 private 修饰的属性或方法，在子类中也是不允许访问的
class Animal {
  private name;
  public constructor(name) {
    this.name = name;
  }
}

class Cat extends Animal {
  constructor(name) {
    super(name);
    console.log(this.name);
  }
}

// index.ts(11,17): error TS2341: Property 'name' is private and only
accessible within class 'Animal'.

----------------
```

```
当构造函数修饰为 private 时，该类不允许被继承或者实例化

class Animal {
  public name;
  private constructor(name) {
    this.name = name;
  }
}
class Cat extends Animal {
  constructor(name) {
    super(name);
  }
}

let a = new Animal('Jack');

// index.ts(7,19): TS2675: Cannot extend a class 'Animal'. Class
constructor is marked as private.
// index.ts(13,9): TS2673: Constructor of class 'Animal' is private and
only accessible within the class declaration.
```

readonly

```
如果 readonly 和其他访问修饰符同时存在的话，需要写在其后面

class Animal {
  // public readonly name;
  public constructor(public readonly name) {
    // this.name = name;
  }
}
```

存取器

```
使用 getter 和 setter 可以改变属性的赋值和读取行为

class Animal {
  constructor(name) {
    this.name = name;
  }
  get name() {
    return 'Jack';
  }
  set name(value) {
    console.log('setter: ' + value);
  }
}
```

```
let a = new Animal('Kitty'); // setter: Kitty
a.name = 'Tom'; // setter: Tom
console.log(a.name); // Jack
```

抽象类

```
abstract 用于定义抽象类和其中的抽象方法

首先，抽象类是不允许被实例化的

其次，抽象类中的抽象方法必须被子类实现

abstract class Animal {
  public name;
  public constructor(name) {
    this.name = name;
  }
  public abstract sayHi();
}

class Cat extends Animal {
  public sayHi() {
    console.log(`Meow, My name is ${this.name}`);
  }
}

let cat = new Cat('Tom');
```

类与接口

类实现接口

```
接口（Interfaces）可以用于对「对象的形状（Shape）」进行描述

接口的另一个用途，对类的一部分行为进行抽象

多个类，拥有共同的属性或者方法（接口）

一个类可以实现多个接口

interface Alarm {
    alert(): void;
}

interface Light {
    lightOn(): void;
    lightOff(): void;
}

class Car implements Alarm, Light {
    alert() {
        console.log('Car alert');
    }
    lightOn() {
        console.log('Car light on');
    }
    lightOff() {
        console.log('Car light off');
    }
}

------------

TS中的类，不仅可以被new调用创建实例，也可以当作一个类型用作接口约束或接口继承

class Point {
    x: number;
    y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

function printPoint(p: Point) {
    console.log(p.x, p.y);
}

printPoint(new Point(1, 2));

------------

类用作类型，实际上只有实例部分的属性和方法会被当作约束。静态方法会被忽略

class Point {
    /** 静态属性，坐标系原点 */
    static origin = new Point(0, 0);
    /** 静态方法，计算与原点距离 */
    static distanceToOrigin(p: Point) {
        return Math.sqrt(p.x * p.x + p.y * p.y);
    }
    /** 实例属性，x 轴的值 */
    x: number;
    /** 实例属性，y 轴的值 */
    y: number;
    /** 构造函数 */
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
    /** 实例方法，打印此点 */
    printPoint() {
        console.log(this.x, this.y);
    }
}

interface PointInstanceType {
    x: number;
    y: number;
    printPoint(): void;
}

let p1: Point;
let p2: PointInstanceType;

上例中最后的类型 Point 和类型 PointInstanceType 是等价的
```

泛型（非常重要且常用）

泛型约束

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);
    return arg;
}
```

泛型接口

```
可以使用接口的方式来定义一个函数需要符合的形状

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

let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    return source.search(subString) !== -1;
}

--------------

当然也可以使用含有泛型的接口来定义函数的形状
```

```
interface CreateArrayFunc {
    <T>(length: number, value: T): Array<T>;
}

let createArray: CreateArrayFunc;
createArray = function<T>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray(3, 'x'); // ['x', 'x', 'x']

---------------

进一步，我们可以把泛型参数提前到接口名上

interface CreateArrayFunc<T> {
    (length: number, value: T): Array<T>;
}

let createArray: CreateArrayFunc<any>;
createArray = function<T>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray(3, 'x'); // ['x', 'x', 'x']
```

泛型类

```
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; };
```