1. TS中的数据类型

- 布尔类型
  - `var flag:boolean = true`
- 数字类型（number）
  - `var num:number = 123`
- 字符串类型（string）
  - `var str:string = 'this is ts'`
- 数组类型（array）
  - 第一种定义数组的方式：`var arr:number[] = [11, 22, 33]`
  - 第二种定义数组的方式：`var arr:Array<number> = [11, 22, 33]`
  - 第二种定义数组的方式：`var arr:any[] = [11, 22, 33]`
- 元组类型（tuple）【属于数组的特殊一种，数组中有number类型和string类型】
  - `let arr:[number, string] = [123, 'this is ts']`
- 枚举类型

```
enum 枚举名 {
  标识符[=整型常数],
  标识符[=整型常数],
  ...
  标识符[=整型常数],
};
```

```
enum Color {
  blue,
  red,
  'orange'
}

let c: Color = Color.red;
console.log(c);       // 1 打印出的是red在Color枚举类型中的索引
```

```
enum Color {
  blue,
  red = 3,
  'orange'
}

let c: Color = Color.orange;
console.log(c);       // 4 打印出的是根据前一个索引+1，red人为的设置=3，所以orange就3+1
```

- 任意类型：any

```
var num:any = 123;
num = 'str';
num = true;
console.log(num);
```

- null 和 undefined 类型【其他(never类型)数据类型的子类型】

```
var num:number | undefined;
num = 123;
console.log(num);
```

```
// 一个元素可能是number类型 可能是null类型 可能是undefined类型
var num: number | null | undefined;
num = 123;
console.log(num);
```

- void类型：typescript中的void表示没有任何类型，一般用于定义方法的时候没有返回值

```
// void表示方法没有返回任何类型
function run(): void {
  console.log(123);
}
```

```
function run(): number {
  return 123;
}

run();
```

- never类型：是其他类型（包括null和undefined）的子类型，代表从不会出现的值。

```
var a: never;

a = (() => {
  throw new Error('错误');
})()
```

2. TS中的函数

1. 函数的定义

- 函数声明法

```
function run(): string {
  return '123';
}
```

- ts中定义方法传参

```
function getInfo(name: string, age: number): string {
  return `${name} --- ${age}`;
}
```

- 函数表达式

```
var fun = function():number {
  return 123;
}
```

- ts中定义方法传参

```
var getInfo = function(name: string, age: number): string {
  return `${name} --- ${age}`;
}
```

2. 方法可选参数【可选参数必须配置到参数的最后面】

> es5里面方法的实参和形参可以不一样，但是ts中必须一样，如果不一样就需要配置可选参数

```
function getInfo(name: string, age?: number): string {
  if (age) {
    return `${name} --- ${age}`;
  } else {
    return `${name} --- 年龄保密`;
  }
}

getInfo('zhangsan');
getInfo('zhangsan', 123);
```

3. 默认参数

> es5里面没法设置默认参数，es6和ts中都可以设置默认参数

```
function getInfo(name: string, age: number = 20): string {
  if (age) {
    return `${name} --- ${age}`;
  } else {
    return `${name} --- 年龄保密`;
  }
}

getInfo('zhangsan');
getInfo('zhangsan', 30);
```

4. 剩余参数

```
function sumAll(...restult: number[]): number {
  let sumData = 0;
  for (let i = 0; i < result.length; i++) {
    sumData += result[i];
  }

  return sumData;
}

sumAll(1, 2, 3, 4, 5);
```

```
function sumAll(a: number, ...restult: number[]): number {
  let sumData = a;
  for (let i = 0; i < result.length; i++) {
    sumData += result[i];
  }

  return sumData;
}

sumAll(1, 2, 3, 4, 5);
```

5. ts函数的重载

> Java中方法的重载：重载指的是两个或者两个以上同名函数，但他们的参数不一样，这时会出现函数重载的情况

> TypeScript中的重载，通过为同一个函数提供多个函数类型定义来试下多种功能的目的

```
function getInfo(name: string): string;
function getInfo(age: number): number;
function getInfo(str: any): any {
  if (typeof str === 'string') {
    return '我叫: ' + str;
  } else {
    return '年龄是: ' + s
  }
}
```

6. 箭头函数

this指向问题：箭头函数里面的this指向上下文

```
setTimeout(() => {
  alert('run');
}, 1000)
```

3. ES5中的类、继承

1. ES5中写类

- **构造函数**中定义属性和**原型链**上定义属性区别：原型链上面的属性会被多个实例共享；构造函数不会
- 因此，一般**私有变量**定义在构造函数中，**公共的方法**写在原型链上

```
function Person() {
  this.name = '张三';                // 构造函数中定义属性
  this.age = 20;

  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';      // 原型链上扩展属性
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}

const p = new Person();
p.run();

p.work();
```

2. ES5 类里的静态方法

```
function Person() {
  this.name = '张三';                // 构造函数中定义属性
  this.age = 20;
```

```
  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.getInfo = function() {       // 静态方法
  console.log('我是静态方法');
}

// 调用静态方法
Person.getInfo();
```

3. ES5的继承【用的最多：原型链 + 对象冒充的组合继承模式】

- 继承原理：原型链 + 对象冒充的组合继承模式
- 在子类里实例化一个父类：**冒充是Person类的实例，我们知道实例拥有这个类构造函数里的所有属性和方法**
    - **注意：对象冒充可以继承构造函数里面的属性和方法，但是没法继承原型链上的属性和方法**

(1) 对象冒充实现继承：子类只能继承父类构造函数里的属性和方法；问题：【没法继承父类原型链上的属性和方法】

```
// Person类
function Person() {
  this.name = '张三';                // 构造函数中定义属性
  this.age = 20;

  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';      // 原型链上扩展属性
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}


// Web类 继承Person类：原型链 + 对象冒充的组合继承模式
function Web() {
```

```
  Person.call(this);       // 对象冒充实现继承【冒充是Person类的实例，我们知道实例拥有这个类的所有属性和方法】
}

var w = new Web();
w.run();  正确      // 对象冒充可以继承构造函数里面的属性和方法
w.work(); 错误     // 对象冒充可以继承构造函数里面的属性和方法 但是没法继承原型链上的属性和方法
```

(2) 原型链实现继承：既可以继承父类构造函数里的属性和方法，也可以继承原型链上的属性和方法

```
// Person类
function Person() {
  this.name = '张三';                // 构造函数中定义属性
  this.age = 20;

  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';      // 原型链上扩展属性
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}


// Web类 继承Person类：原型链 + 对象冒充的组合继承模式
function Web() {

}
// 父类的实例挂载到子类的原型上，这样子类就拥有了父类的构造函数和原型链上的所有属性和方法
Web.prototype = new Person();    // 原型链实现继承 可以继承构造函数里的属性和方法，也可以继承原型链上的属性和方法

var w = new Web();
w.run();            // 正确
```

(3) 原型链实现继承的问题【实例化子类的时候没法给父类传参】

```
// Person类
function Person(name, age) {
  this.name = name;                // 构造函数中定义属性
  this.age = age;

  this.run = function() {
    alert(this.name + '在运动');    // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';     // 原型链上扩展属性
Person.prototype.work = function() {     // 原型链上扩展方法
  console.log(this.name + '在工作');
}

var p = new Person('李四', 20);

p.run();         // 李四在运动


// 子类
function Web(name, age) {

}

Web.prototype = new Person();

var w = new Web('赵四', 20);       // 实例化子类的时候没法给父类传参
w.run();         // undefined在运动
```

(4) 原型链 + 对象冒充的组合继承模式

```
// Person类
function Person(name, age) {
  this.name = name;                // 构造函数中定义属性
  this.age = age;

  this.run = function() {
    alert(this.name + '在运动');    // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';     // 原型链上扩展属性
```

```
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}

var p = new Person('李四', 20);

p.run();         // 李四在运动


// 子类
function Web(name, age) {
  Person.call(this, name, age);    // 对象冒充 => 可以继承构造函数里面的属性和方法 => 实例化子类时可以给父类传参
}

Web.prototype = new Person();      // 原型链继承

var w = new Web('赵四', 20);
w.run();         // 赵四在运动
w.work();        // 赵四在工作
```

(5) 原型链 + 对象冒充组合继承的另一种方式

- 把`Web.prototype = new Person()`改成`Web.prototype = Person.prototype`
- 因为`Person.call(this, name, age)`对象冒充已经把父类构造函数里的属性和方法继承过来了，只需要再继承父类的原型链就行了

```
// Person类
function Person(name, age) {
  this.name = name;                // 构造函数中定义属性
  this.age = age;

  this.run = function() {
    alert(this.name + '在运动');     // 构造函数中定义方法 【实例方法】
  }
}

Person.prototype.gender = '男';      // 原型链上扩展属性
Person.prototype.work = function() {      // 原型链上扩展方法
  console.log(this.name + '在工作');
}

var p = new Person('李四', 20);
```

```
p.run();        // 李四在运动


// 子类
function Web(name, age) {
  Person.call(this, name, age);    // 对象冒充 => 可以继承构造函数里面的属性和方法 => 实例化子类时可以给父类传参
}

Web.prototype = Person.prototype;      // 原型链继承

var w = new Web('赵四', 20);
w.run();        // 赵四在运动
w.work();       // 赵四在工作
```

4. ES6 / TS中类的 定义、继承、修饰符、静态属性、静态方法、继承多态、抽象类、

(1) TS中定义类

- `constructor(){}` 其实就是一个**钩子**，在实例化类的时候触发的方法

```
class Person {
  name: string;    // 属性 前面省略了public关键词
  constructor(n: string) {  // 构造函数 实例化类的时候触发的方法
    this.name = n;
  }

  run(): void {
    alert(this.name);
  }
}

var p = new Person('张三');

p.run();
```

```
class Person {
```

```
  name: string;    // 属性 前面省略了public关键词
  constructor(name: string) {  // 构造函数 实例化类的时候触发的方法
    this.name = name;
  }

  getName(): string {
    return this.name;
  }

  setName(name: string): void {
    this.name = name;
  }
}

var p = new Person('张三');
console.log(p.getName());      // 张三

p.setName('李四');
console.log(p.getName());      // 李四
```

(2) TS中如何实现继承：extends / super

```
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

var p = new Person('王五');
console.log(p.run());          // 王五在运动


// 子类
class Web extends Person {
  constructor(name: string) {     // 子类里不写constructor的话，默认使用父类的
consstructor
    super(name);     // 初始化父类的构造函数
```

```
  }
}

var w = new Web('李四');
console.log(w.run());          // 李四在运动
```

(3) TS中继承的探讨 父类的方法和子类的方法一致

```
class Person {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

var p = new Person('王五');
console.log(p.run());           // 王五在运动


// 子类
class Web extends Person {
  constructor(name: string) {     // 子类里不写constructor的话，默认使用父类的
consstructor
    super(name);      // 初始化父类的构造函数
  }

  run(): string {
    return `${this.name}在运动 - 子类`;
  }

  work() {
    console.log(`${this.name}在工作`);
  }
}

var w = new Web('李四');
w.work();           // 李四在工作
w.run();            // 李四在运动馆 - 子类
```

(4) 类里面的修饰符【TS里面定义属性的时候给我们提供了三种修饰符】

> public, protected, private

- **属性如果不加修饰符 默认就是public**
- public：公有 => 在当前类里面、子类、类外面都可以访问
- protected：保护类型 => 在当前类里面、子类里面可以访问；在类外部没法访问
- private：私有 => 在当前类里面可以访问，子类和类外部都没法访问

```
class Person {
  public name: string;            /* 公有属性 */

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

var p = new Person('王五');
console.log(p.run());          // 王五在运动


// 子类
class Web extends Person {
  constructor(name: string) {     // 子类里不写constructor的话，默认使用父类的
consstructor
    super(name);      // 初始化父类的构造函数
  }

  run(): string {
    return `${this.name}在运动 - 子类`;
  }

  work() {
    console.log(`${this.name}在工作`);
  }
}
```

```
var w = new Web('李四');
w.work();                // 李四在工作
```

- **public 类外部访问公有属性**

```
class Person {
  public name: string;            /* 公有属性 */

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

// 类外部访问公有属性
var p = new Person('哈哈哈');
console.log(p.name);           // 哈哈哈
```

- **protected 类外部没法访问到保护类型的属性 ts编译会报错提示**

```
class Person {
  protected name: string;            /* 保护类型 */

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

var p = new Person('哈哈哈');
console.log(p.name);             // ts编译报错
```

- **private 只能在当前类使用**

```
class Person {
  private name: string;            /* 私有属性 */

  constructor(name: string) {
    this.name = name;
  }

  run(): string {
    return `${this.name}在运动`;
  }
}

class Web extends Person {
  constructor(name: string) {
    super(name);
  }

  work() {
    console.log(`${this.name}在工作`);     // ts编译报错，私有属性只能在定义的类里使用
  }
}
```

(5) 静态属性、静态方法

ES5 中的静态属性、静态方法写法

```
function Person() {
  this.name1 = 'hehe';
  this.run1 = function() {         /* 实例方法 */

  }
}

Person.name2 = "haha";
Person.run2 = function() {         /* 静态方法 */

}

// 两种方法的调用方法不同
var p = new Person();
p.name1;                        // 实例属性的调用方法
```

```
p.run1();                       // 实例方法的调用方法

Person.name2;                   // 静态属性的调用方法
Person.run2();                  // 静态方法的调用方法
```

ES6/TS 中的静态属性、静态方法写法

- 静态方法里面没法直接调用类里面的属性；若想调用的话，把类里的属性写成 `static age` 这样
- 因为静态方法随着类的加载而加载，对象是在类存在后才能创建的，所以静态方法优先于对象存在，所以不能在静态方法里面访问成员变量

```
class Person {
  public name: string;
  static age: number = 20;
  constructor(name) {
    this.name = name;
  }

  run() {                       // 实例方法
    console.log(`${this.name}在运动`);
  }

  work() {                      // 实例方法
    console.log(`${this.work}在工作`);
  }

  static print() {              // 静态方法，里面没法直接调用类里面的属性
    console.log('print方法' + Person.age);
  }
}
```

(6) 多态

> 父类定义一个方法不去实现，让继承它的子类去实现，每一个子类有不同的表现

多态属于继承

```
class Animal {
  name: string;
  constructor(name: string) {
```

```
    this.name = name;
  }

  eat() {             // 具体吃什么 不知道，具体吃什么 让继承它的子类去实现 每个子类的表现不一样
    console.log('吃的方法');
  }
}

class Dog extends Animal {
  constructor(name: string){
    super(name);
  }

  eat() {
    return this.name + '吃肉';
  }
}

class Cat extends Animal {
  constructor(name: string) {
    super(name);
  }

  eat() {
    return this.name + '吃老鼠'
  }
}
```

(7) 抽象方法

> TS中的抽象类：它是提供其他类继承的基类，不能直接被实例化

> 用abstract关键字定义抽象类和抽象方法，抽象类中的抽象方法不包含具体实现并且必须在派生类中实现

> abstract抽象方法只能放在抽象类里面

抽象类和抽象方法用来定义标准；eg: Animal这个类要求它的子类必须包含eat方法

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

```
  abstract eat():any;               // 抽象类中的抽象方法，不包含具体实现且必须在派生类中实现

  run() {
    console.log('其他方法可以不实现');
  }
}

class Dog extends Animal {
  // 抽象类的子类必须实现抽象类里的抽象方法

  constructor(name: any) {
    super(name);
  }

  eat() {
    console.log(this.name + '吃粮食');
  }
}

var d = new Dog('小狗');
d.eat();                          // 小狗吃粮食
```

5. TS中的接口、属性类型接口

> 接口的作用：在面向对象的编程中，接口是一种规范的定义。他定义了行为和动作的规范，在程序设计里面，接口起到了一种限制和规范的作用。接口定义了某一批类所需要遵守的规范，接口不关心这些类的内部状态数据，也不关心这些类里方法的实现细节，它只规定这批类里必须提供某些方法，提供这些方法的类就可以满足实际需要。typescript中的接口类似于java，同时还增加了更灵活的接口类型，包括属性、函数、可索引和类等。

- 属性接口
- 函数类型接口
- 可索引接口
- 类类型接口
- 接口扩展

(1) 属性接口 对json的约束

- **ts中自定义方法传入参数，对json进行约束**

```
function printLabel(labelInfo: {label: string}): void {
  console.log('printLabel');
}

printLabel('haha')               // 错误的写法
printLabel({name: '张三'});      // 错误的写法
printLabel({label: '张三'});     // 正确的写法
```

- **对批量方法传入参数进行约束**
- 接口：行为和动作的规范，对批量方法进行约束

```
// 就是传入对象的约束   属性接口
interface FullName {
  firstName: string;
  secondName: string;
}

function printName(name: FullName) {
  // 必须传入对象  firstName  secondName
  console.log(name.firstName + ' ' + name.secondName)
}

var obj = {
  age: 20,
  firstName: '张',
  secondName: '三'
};
printName(obj);
```

(1.1) 接口：可选属性

```
interface FullName {
  firstName: string;
  secondName?: string;
}

function getName(name: FullName) {
  console.log(name);
}

getName({
  firstName: 'haha'
})
```

例子

```
interface Config {
  type: string;
  url: string;
  data?: string;
  dataType: string;
}

function ajax(config: Config) {
  var xht = new XMLHttpRequest();

  xhr.open(config.type, config.url, true);
  xhr.send(config.data);
  xhr.onreadystatechange = function() {
    if(xhr.readyState === 4 && xhr.status === 200) {
      console.log('成功');

      if(config.dataType === 'json') {
        JSON.parse(xhr.responseText);
      } else {
        console.log(xhr.responseText);
      }
    }
  }
}

ajax({
  type: 'get',
  url: 'https://www.google.com',
```

```
  data: 'name: zhangsan',
  dataType: 'json'
})
```

(2) 函数类型接口：对方法传入的参数和返回值进行约束

例子：加密的函数类型接口

```
interface encrypt {
  (key: string, value: string): string;
}

var md5: encrypt = function(key: string, value: string): string {
  // 模拟操作
  return key + value;
}

md5('name', 'zhangsan');
```

(3) 可索引接口：数组、对象的约束【不常用】

- **最前面学过的ts中定义数组的方式**

```
var arr: number[] = [123, 345];
var arr1: Array<string> = ['111', '222'];
```

- **可索引接口：对数组的约束**

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

var arr: UserArr = ['aaa', 'bbb'];        // 正确写法
var arr1: UserArr = [123, 'aaa'];         // 错误写法
```

- **可索引接口：对对象的约束**

```
interface UserObj {
  [index: string]: string
}

var arr: UserObj = {
  name: '20',
}
```

(4) 类类型接口：对类的约束【子类必须实现基类接口的属性和方法】(抽象类有点相似)【常用】

- **implement 这个类类型接口的类，必须实现这个接口里的属性和方法**
  - Animal类定义一个标准，子类实现Animal这个父类需要实现父类里的属性和方法

相当于把属性类型接口和方法类型接口整合了

```
interface Animal {
  name: string;
  eat(str: string): void;
}

class Dog implements Animal {
  name: string;
  constructor(str: string) {
    this.name = name;
  }
  eat() {
    console.log(this.name + '吃粮食');
  }
}

var d = new Dog('小黑');
d.eat();

class Cat implements Animal {
  name: string;
  constructor(str: string) {
    this.name = name;
  }
  eat(food: string) {
    console.log(this.name + '吃' + food);
  }
}
```

```
var c = new Cat('小花');
c.eat('鱼');
```

(5) 接口扩展：接口可以继承接口

- 实现

```
interface Animal {
  eat(): void;
}

interface Person extends Animal {
  work(): void;
}

class Web implements Person {
  public name: string;
  constructor(name: string) {
    this.name = name;
  }
  eat() {
    console.log(this.name + '喜欢吃馒头');
  }
  work() {
    console.log(this.name + '写代码');
  }
}

c                    // 小雷喜欢吃馒头
```

- 继承 + 实现

```
interface Animal {
  eat(): void;
}

interface Person extends Animal {
  work(): void;
}

class Programmer {
```

```
  public name: string;
  constructor(name: string) {
    this.name = name;
  }
  coding(code: string) {
    console.log(this.name + code);
  }
}

class Web extends Programmer implements Person {
  constructor(name: string) {
    super(name);
  }
  eat() {
    console.log(this.name + '喜欢吃馒头');
  }
  work() {
    console.log(this.name + '写代码');
  }
}

var w = new Web('小李');
w.eat();
w.coding('写ts代码');              // 小李写ts代码
```

6. TS中的泛型定义、泛型函数、泛型类、泛型接口

> 泛型：软件工程中，我们不仅要创建一致的定义良好的API，同时也要考虑可重用性。组件不仅能够支持当前的数据类型，同时也能支持未来的数据类型，这在创建大型系统时为你提供了十分灵活的功能。

> 在像C#和Java这样的语言中，可以使用泛型来创建可重用的组件，一个组件可以支持多种类型的数据。这样用户就可以以自己的数据类型来使用组件。

> 通俗理解，泛型就是解决 类 接口 方法的复用性，以及对不特定数据类型的支持（类型校验）

(1) 泛型、泛型函数

- 只能返回string类型的数据**

```
function getData(value: string): string {
  return value;
}
```

- 同时返回string类型和number类型 【代码冗余】

```
function getData1(value: string): string {
  return value;
}

function getData2(value: number): number {
  return value;
}
```

- 同时返回string类型和number类型 - any可以解决这个问题，但是放弃了类型检查

```
function getData(value: any): any {
  return value;
}

getData(123);
getData('123');
```

- 泛型：可以支持不特定的数据类型 要求：传入的参数和返回的参数一致
- T(可以任意字母)表示泛型，具体什么类型是调用这个方法的时候决定的

```
function getData<T>(value: T): T {
  return value;
}

getData<number>(123);            // 正确写法
getData<number>('123');          // 错误写法
```

```
function getData<T>(value: T): any {
  return '123';
}

getData<number>(123);              // 参数必须是number
getData<string>('这是一个泛型');    // 参数必须是string
```

(2) 泛型类

> 泛型类：比如有个最小堆算法，需要同时支持返回数字和字符串两种类型。通过类的泛型来实现

```
class MinClass {
  public list: number[] = [];

  add(num) {
    this.list.push(num);
  }
  min(): number {
    var minNum = this.list[0];

    for(vai i = 0; i < this.list.length; i++) {
      if(minNum > this.list[i]) {
        minNum = this.list[i];
      }
    }

    return minNum;
  }
}

var m = new MinClass();
m.add(2);
m.add(22);
m.add(23);
m.add(1);

console.log(m.min());
```

- **类的泛型**

```
class MinClass<T> {
  public list: T[] = [];

  add(value: T):void {
    this.list.push(value);
  }

  min(): T {
    var minNum = this.list[0];

    for(vai i = 0; i < this.list.length; i++) {
      if(minNum > this.list[i]) {
        minNum = this.list[i];
```

```
      }
    }

    return minNum;
  }
}

var m1 = new MinClass<number>;    // 实例化类 并且指定了类的T代表的类型是number
m1.add(1);
m1.add(22);
m1.add(3);
m1.min();

var m2 = new MinClass<string>;    // 实例化类 并且指定了类的T代表的类型是string
m1.add('a');
m1.add('c');
m1.add('v');
m1.min();
```

(3) 泛型接口

- 函数类型接口

```
interface ConfigFn {
  (value1: string, value2: string): string;
}

var setData: ConfigFn = function(value1: string, value2: string):string {
  return value1 + value2;
}

setData('name', '张三');
```

我们想让这个接口可以返回number类型和string类型

- 第一种定义泛型接口方法

```
interface ConfigFn {
  <T>(value: T): T;
}

var getData: ConfigFn = function<T>(value: T): T {
  return value;
}

getData<string>('张三');           // 正确
getData<string>(1234);            // 错误
```

- **第二种定义泛型接口方法**

```
interface ConfigFn<T> {
  (value: T): T;
}

function getData<T>(value: T): T {
  return value;
}

var myGetData: ConfigFn<string> = getData;

myGetData('20');                  // 正确
myGetData(20);                    // 错误
```

(4) 把类当做参数的泛型类

1. 定义个类
2. 把类作为参数来约束数据传入的类型

- 定义一个User的类，这个类的作用就是映射数据库字段，然后定义一个MysqlDb的类，这个类用于操作数据库，然后把User类作为参数传入到MysqlDb中
- **把类作为参数来约束数据传入的类型**

```
class User {
  username: string | undefined;
  password: string | undefined;
}
```

```
class MysqlDb {
  add(user: User): boolean {
    console.log(user);
    return true;
  };
}

var u = new User();
u.username = '张三';
u.password = '123456';

var Db = new MysqlDb();
Db.add(u);
```

```
class ArticleCate {
  title: string | undefined;
  desc: string | undefined;
  status: number | undefined;
}

class MysqlDb {
  add(info: ArticleCate): boolean {
    console.log(info);
    return true;
  };
}

var a = new ArticleCate();
a.title = '国内';
a.desc = '国内新闻';
a.status = 1;

var Db = new MysqlDb();
Db.add(a);
```

- **使用泛型，避免MysqlDb重复封装**
- 把类当做参数传入泛型类

```
// 操作数据库的泛型类
class MysqlDb<T> {
  add(info: T): boolean {
    console.log(info);
    return true;
  };
```

```
  update(info: T, id: number): boolean {
    console.log(info);
    console.log(id);
    return true;
  }
}

// 想给User表增加数据
// 1. 定义一个User类 和 数据库进行映射
class User {
  username: string | undefined;
  password: string | undefined;
}

var u = new User();

u.username = '阿三';
u.password = '123456';

var Db = new MysqlDb<User>();
Db.add(u);

// 2.定义一个Category类 和 数据库进行映射
class ArticleCate {
  title: string | undefined;
  desc: string | undefined;
  status: number | undefined;
  constructor(params: {
    title: string | undefined,
    desc: string | undefined,
    status: number | undefined;
  }) {
    this.title = params.title;
    this.desc = params.desc;
    this.status = params.status;
  }
}

// 增加操作
var a = new ArticleCate({
  title: '分类',
  desc: '哈哈哈'
});

var Db = new MysqlDb<ArticleCate>();
Db.add(a);
```

```
// 更新操作
var b = new ArticleCate({
  title: '分类111',
  desc: '2222'
});
b.status = 0;
var Db = new MysqlDb<ArticleCate>();
Db.update(b, 12);
```

(5) TS综合使用 - TS封装统一操作Mysql, mongodb, mssql底层库

> 功能：定义一个操作数据库的库 支持Mysql Mssql Mongodb
>
> 要求1：Mysql Mssql MongoDb功能一样 都有 add update delete get 方法
>
> 注意：约束统一的规范，以及代码重用
>
> 解决方案：需要约束规范所以要定义接口，需要代码重用所以用到泛型
>
> 1. 接口：在面向对象的编程中，接口是一种规范的定义，它定义了行为和动作的规范
> 2. 泛型 通俗理解：泛型就是解决 类 接口 方法的复用性

```
interface DBI<T> {
  add(info: T): boolean;
  update(info: T, id: number): boolean;
  delete(id: number): boolean;
  get(id: number): any[];
}

// 定义一个操作mysql数据库的类      注意：要实现泛型接口  这个类也应该是一个泛型类
class MysqlDb implements DBI<T> {
  add(info: T, id: number): boolean {
    console.log(info);
    return true;
  }
  update(info: T, id: number): boolean {
    throw new Error('Method not implemented.');
  }
  delete(id: number): boolean {
    throw new Error('Method not implemented.');
  }
  get(id: number): any[] {
    throw new Error('Method not implemented.');
```

```
  }
}

// 定义一个操作mssql数据库的类
class MsSqlDb implements DBI<T> {
  add(info: T): boolean {
    throw new Error('Method not implemented.');
  }
  update(info: T, id: number): boolean {
    throw new Error('Method not implemented.');
  }
  delete(id: number): boolean {
    throw new Error('Method not implemented.');
  }
  get(id: number): any[] {
    throw new Error('Method not implemented.');
  }
}

// 操作用户表   定义一个User类和数据表做映射
class User {
  username: string | undefined;
  password: string | undefined;
}

var u = new User();
u.username = '张三';
u.password = '123456';

var oMysql = new MysqlDb<User>();       // 类作为参数来约束数据传入的类型
oMysql.add(u);

//
class User {
  username: string | undefined;
  password: string | undefined;
}
var u = new User();
u.username = '张三111';
u.password = '123456';

var oMysql = new MsSqlDb<User>();       // 类作为参数来约束数据传入的类型
oMysql.add(u);
```

7. 模块

> 模块的概念： 我们可以把一些公共的功能单独抽离成一个文件作为模块。
>
> 模块里面的变量 函数 类等默认是私有的，如果我们要在外部访问模块里面的数据（变量、函数、类），我们需要通过export暴露模块里面的数据（变量、函数、类。。。）。暴露后我们通过import引入模块就可以使用模块里面暴露的数据（变量、函数、类）

8. 命名空间

> 命名空间：在代码量较大的情况下，为了避免各种变量命名相冲突，可将相似功能的函数、类、接口等放置到命名空间内。TS的命名空间可以将代码包裹气啦，只对外暴露需要在外部访问的对象。命名空间的对相关通过export暴露

> 命名空间和模块的区别：
>
> 命名空间：内部模块，主要用于组织代码，避免命名冲突
>
> 模块：TS的外部模块的简称，侧重代码的复用，一个模块里可能会有多个命名空间

```
namespace A {
  interface Animal {
    ....
  }

  export class Cat implements Animal {
    ...
  }
}

namespace B {
  interface Animal {
    ...
  }
}
```

9. TS的装饰器