

# class 类

## Class 类

类是用于 **创建对象** 模板。同时类声明也会引入一个 **新类型**，可定义其 **实例属性**、**方法** 和 **构造函数**。

学习目录：

1. **Class类 实例属性**
2. Class类 构造函数
3. Class类 定义方法
4. 静态属性 和 静态方法
5. 继承 extends 和 super 关键字
6. instanceof 检测是否实例
7. 修饰符 (readonly、private ...)

```
// 类名 首字母大写（规范）
class 类名{
    // 1. 实例属性(字段)

    // 2. 构造函数

    // 3. 方法
}

// 使用类 实例化对象 基于类 创建对象
const p:类名 = new 类名()
```

## Class 类 - 实例属性(字段)

通过实例属性（字段），可以保存各种类型的数据

```
// 类
class 类名{
    // 字段名、类型、初始值
    字段名1:类型='xxx'
    // 可选字段可以不设置初始值
    字段名2?:类型
}

// 可选字段在使用时需要配合 可选链操作符 避免出错
const p: 类名 = new 类名()
p.字段名1
p?.字段名2
```

```
class Person {
    name: string = 'jack'
    food?: string
}

const p = new Person()
p.name = 'tom'

console.log(p.name)
console.log('字符串', p.food?.length)
```

## Class 类

类是用于 **创建对象** 模板。同时类声明也会引入一个 **新类型**，可定义其 **实例属性**、**方法** 和 **构造函数**。

### 学习目录：

1. Class类 实例属性
- 2. Class类 构造函数**
3. Class类 定义方法
4. 静态属性 和 静态方法
5. 继承 extends 和 super 关键字
6. instanceof 检测是否实例
7. 修饰符 (readonly、private ...)

```
// 类名 首字母大写（规范）
class 类名{
    // 1. 实例属性(字段)

    // 2. 构造函数

    // 3. 方法
}

// 使用类 实例化对象 基于类 创建对象
const p:类名 = new 类名()
```

## Class 类 - 构造函数

不同实例，将来需要有不同的字段初始值，就需要通过构造函数实现

```
class 类{
  字段A: 类型
  字段B: 类型
  constructor(参数...) {
    // 通过 new 实例化的时候 会调用 constructor
    // 通过关键字 this 可以获取到实例对象
    this.字段名A = 参数
  }
}

const 实例1 = new 类(参数...)
const 实例2 = new 类(参数...)
```

```
class Food {
  name: string
  price: number
  constructor(name: string, price: number) {
    this.name = name
    this.price = price
  }
}

const f1 = new Food('西红柿鸡蛋', 15)
const f2 = new Food('土豆炖鸡块', 24)
```

## Class 类

类是用于 **创建对象** 模板。同时类声明也会引入一个 **新类型**，可定义其 **实例属性**、**方法** 和 **构造函数**。

### 学习目录：

1. Class类 实例属性
2. Class类 构造函数
- 3. Class类 定义方法**
4. 静态属性 和 静态方法
5. 继承 extends 和 super 关键字
6. instanceof 检测是否实例
7. 修饰符 (readonly、private ...)

```
// 类名 首字母大写（规范）
class 类名{
    // 1. 实例属性(字段)

    // 2. 构造函数

    // 3. 方法
}

// 使用类 实例化对象 基于类 创建对象
const p:类名 = new 类名()
```

## Class 类 - 定义方法

类中可以定义**方法**，并且在内部编写逻辑

```
class 类名{  
    方法名(参数...):返回值类型{  
        // 逻辑...  
        // 可以通过 this 获取实例对象  
    }  
}
```

```
class Person{  
    name:string  
  
    constructor(name:string) {  
        this.name = name  
    }  
  
    sayHi(name:string){  
        console.log(`你好${name}, 我是:${this.name}`)  
    }  
}  
  
const p:Person = new Person('jack')  
p.sayHi('rose')
```

## Class 类

类是用于 **创建对象** 模板。同时类声明也会引入一个 **新类型**，可定义其 **实例属性**、**方法** 和 **构造函数**。

### 学习目录：

1. Class类 实例属性
2. Class类 构造函数
3. Class类 定义方法
4. **静态属性 和 静态方法**
5. 继承 extends 和 super 关键字
6. instanceof 检测是否实例
7. 修饰符 (readonly、private ...)

```
// 类名 首字母大写（规范）
class 类名{
    // 1. 实例属性(字段)

    // 2. 构造函数

    // 3. 方法
}

// 使用类 实例化对象 基于类 创建对象
const p:类名 = new 类名()
```



## Class 类 - 静态属性和 静态方法

类还可以添加 静态属性、方法，后续访问需要通过 类 来完成

```
// 定义
class 类{
    static 字段:类型
    static 方法(){}
}
```

```
// 使用
类.字段
类.方法()
```

```
// 静态属性和方法
class Robot {
    // 如果不设置值，默认是 undefined
    static version: string = '10.12'

    // 工具方法
    static getRandomNumber () {
        return Math.random()
    }
}
Robot.version
Robot.getRandomNumber()
```

## Class 类

类是用于 **创建对象** 模板。同时类声明也会引入一个 **新类型**，可定义其 **实例属性**、**方法** 和 **构造函数**。

### 学习目录：

1. Class类 实例属性
2. Class类 构造函数
3. Class类 定义方法
4. 静态属性 和 静态方法
5. **继承 extends 和 super 关键字**
6. instanceof 检测是否实例
7. 修饰符 (readonly、private ...)

```
// 类名 首字母大写（规范）
class 类名{
    // 1. 实例属性(字段)

    // 2. 构造函数

    // 3. 方法
}

// 使用类 实例化对象 基于类 创建对象
const p:类名 = new 类名()
```

## Class 类 - 继承 extends 和 super 关键字

类可以通过 **继承** 快速获取另外一个类的 **字段** 和 **方法**

```
class 父类 {  
    // 字段  
    // 方法  
    // 构造函数  
}  
  
class 子类 extends 父类 {  
    // 自己的字段（属性）  
    // 自己的方法  
    // 可以重写父类方法  
}
```

```
class 父类 {  
    func(){  
    }  
}  
  
class 子类 extends 父类 {  
    constructor() {  
        super() // 调用父类构造函数  
    }  
    方法(){  
        super.方法() // 调用父类方法  
    }  
}
```

子类通过 **super** 可以访问父类的实例字段、实例方法和构造函数。

## Class 类

类是用于 **创建对象** 模板。同时类声明也会引入一个 **新类型**，可定义其 **实例属性**、**方法** 和 **构造函数**。

### 学习目录：

1. Class类 实例属性
2. Class类 构造函数
3. Class类 定义方法
4. 静态属性 和 静态方法
5. 继承 extends 和 super 关键字
6. instanceof 检测是否实例
7. 修饰符 (readonly、private ...)

```
// 类名 首字母大写（规范）
class 类名{
    // 1. 实例属性(字段)

    // 2. 构造函数

    // 3. 方法
}

// 使用类 实例化对象 基于类 创建对象
const p:类名 = new 类名()
```

## Class 类 - instanceof

instanceof 运算符可以用来检测某个对象是否是某个类的实例

实例对象 instanceof Class

typeof 表达式

### 试一试:

1. 定义 父类
2. 定义 子类, 继承 父类
3. 实例化子类 并通过 **instanceof** 进行判断
4. 判断 **数组** 是否为 Array 的实例

## Class 类

类是用于 **创建对象** 模板。同时类声明也会引入一个 **新类型**，可定义其 **实例属性**、**方法** 和 **构造函数**。

### 学习目录：

1. Class类 实例属性
2. Class类 构造函数
3. Class类 定义方法
4. 静态属性 和 静态方法
5. 继承 extends 和 super 关键字
6. instanceof 检测是否实例
7. 修饰符 (readonly、private ...)

```
// 类名 首字母大写（规范）
class 类名{
    // 1. 实例属性(字段)

    // 2. 构造函数

    // 3. 方法
}

// 使用类 实例化对象 基于类 创建对象
const p:类名 = new 类名()
```

## Class 类 - 修饰符 - readonly

类的 **方法** 和 **属性** 可以通过修饰符来 **限制** 访问

修饰符包括：**readonly**、private、protected 和 public。省略不写默认为 public

```
class 类{  
    readonly 属性: 类型  
}
```

只可以取值，无法修改

## Class 类 - 修饰符 - private

private 修饰的成员不能在声明该成员的类之外访问, 包括子类

```
class 类{  
    private 属性: 类型  
    private 方法(){}  
}
```

```
class Person {  
    private name: string = ''  
    private age: number = 0  
  
    sayHi() {  
        // 内部可以访问  
        console.log(`你好, 我叫:${this.name}`)  
    }  
}  
  
class Student extends Person{  
    sayHello() {  
        // 无法访问 报错  
        console.log(`你好, 我叫:${super.name}`)  
    }  
}  
  
const p = new Person()  
// p.name // 无法访问 报错  
p.sayHi()
```



## Class 类 - 修饰符 - protected

protected修饰符的作用与private修饰符非常相似

不同点是protected修饰的成员允许在 派生类(子类) 中访问

```
class 类{  
    protected 属性:类型  
    protected 方法(){}  
}
```

```
class Person {  
    protected name: string = ''  
    protected age: number = 0  
  
    sayHi() {  
        // 内部可以访问  
        console.log(`你好, 我叫:${this.name}`)  
    }  
}  
  
class Student extends Person{  
    sayHello() {  
        // 可以访问到父类的protected属性  
        console.log(`你好, 我叫:${super.name}`)  
    }  
}  
  
const p = new Person()  
// p.name // 无法访问 报错  
p.sayHi()
```

## Class 类 - 修饰符 - public

public 修饰的类成员（字段、方法、构造函数）

在程序的任何可访问该类的地方都是可见的（默认）

```
class 类{  
    public 属性:类型  
    public 方法(){  
    }  
}
```

```
class Person {  
    public name: string = ''  
    public age: number = 0  
  
    sayHi() {  
        // 内部可以访问  
        console.log(`你好，我叫:${this.name}`)  
    }  
}  
  
class Student extends Person{  
    sayHello() {  
        // 可以访问到父类的protected属性  
        console.log(`你好，我叫:${super.name}`)  
    }  
}  
  
const p = new Person()  
// p.name // 可以访问  
p.sayHi()
```



# 总结

修饰符名	作用	访问限制
readonly	只读	无限制
private	私有	类内部可以访问
protected	保护	类及子类可以访问
public	公共	无限制

# 剩余参数 和 展开运算符



黑马程序员  
[www.itheima.com](http://www.itheima.com)

传智教育旗下  
高端IT教育品牌

## 剩余参数 和 展开运算符 ...

剩余参数的语法，我们可以将 **函数** 或 **方法** 中一个不定数量的参数表示为一个数组

```
// 剩余参数只能写在最后一位
function 函数名(参数1,参数2,...剩余参数数组){
  // 逻辑代码
  // 之前的参数：挨个获取即可
  // 剩余参数：以数组的形式获取
}
```

```
function sum(numA:number,numB:number,...theArgs:number[]) {
  let total = numA+numB;
  for (const arg of theArgs) {
    total += arg;
  }
  return total;
}
console.log(sum(1, 2, 3).toString()) // 6
console.log(sum(1, 2, 3, 4).toString()) // 10
```

## 剩余参数 和 展开运算符 ...

出于程序稳定性，以及运行性能考虑，在 ArkTS 中 ...(展开运算符) 只能用在数组上

```
const numArr1: number[] = [1, 2, 3, 4]
const numArr2: number[] = [5, 6, 7]

// 合并到一起
const totalArr: number[] = [...numArr1, ...numArr2]
```

# 接口补充



黑马程序员  
[www.itheima.com](http://www.itheima.com)

传智教育旗下  
高端IT教育品牌

## 接口补充 – 接口继承

接口继承使用的关键字是 `extends`

```
interface 接口1{
    属性1:类型
}
interface 接口2 extends 接口1 {
    属性2:类型
}
```

```
interface IAnimal {
    name: string
}
interface ICat extends IAnimal {
    color: string
}

const cat: ICat = {
    name: '加菲猫',
    color: '黑色'
}
```



## 接口补充 – 接口实现

可以通过接口结合 `implements` 来限制 类 必须要有 某些属性 和 方法

```
interface 接口{  
    属性:类型  
    方法:方法类型  
}  
  
class 类 implements 接口{  
    // 必须实现 接口中定义的 属性、方法,  
    // 否则会报错  
}
```

```
interface IDog {  
    name: string  
    bark: () => void  
}  
  
class Dog implements IDog {  
    name: string = ''  
    food: string = ''  
  
    bark() {  
  
    }  
}
```

# 泛型



黑马程序员  
[www.itheima.com](http://www.itheima.com)

传智教育旗下  
高端IT教育品牌



# 目录

Contents

- ◆ 泛型函数
- ◆ 泛型约束
- ◆ 多个泛型参数
- ◆ 泛型接口
- ◆ 泛型类

## 泛型

泛型可以让【函数】等，与多种【不同的类型】一起工作，灵活可复用

通俗一点就是：类型是 可变 的!

```
function fn1 (temp: string): string {  
    return temp  
}  
function fn2 (temp: number): number {  
    return temp  
}  
function fn3 (temp: boolean): boolean {  
    return temp  
}
```

```
function 函数名<Type>(temp:Type):Type{  
    return temp  
}  
  
fn<string>('123')  
fn<number>(1)
```

泛型函数



# 目录

Contents

- ◆ 泛型函数
- ◆ 泛型约束
- ◆ 多个泛型参数
- ◆ 泛型接口
- ◆ 泛型类

## 泛型约束

之前的类型参数，可以传递任何类型，没有限制。

如果希望有限制 → 泛型约束

```
interface 接口{  
    属性: 类型  
}  
function 函数<Type extends 接口>(){}  
  
// 传入的类型必须要有 接口中的属性
```

```
interface ILength {  
    length: number  
}  
  
function fn<T extends ILength>(param: T) {  
    console.log('', param.length)  
}
```



# 目录

Contents

- ◆ 泛型函数
- ◆ 泛型约束
- ◆ 多个泛型参数
- ◆ 泛型接口
- ◆ 泛型类

## 多个泛型参数

日常开发的时候，如果有需要，可以添加多个 类型变量

```
function funcA<T, T2>(param1: T, param2: T2) {  
  console.log('参数 1', param1)  
  console.log('参数 2', param2)  
}  
  
funcA<string, number>('大白菜', 998)  
funcA<string[], boolean[]>(['小老虎'], [false])
```





# 目录

Contents

- ◆ 泛型函数
- ◆ 泛型约束
- ◆ 多个泛型参数
- ◆ 泛型接口
- ◆ 泛型类

## 泛型接口

定义接口的时候，结合泛型定义，就是泛型接口

```
interface 接口<Type>{  
    // 内部使用Type  
}
```

```
interface IdFunc<Type> {  
    id: (value: Type) => Type  
    ids: () => Type[]  
}  
  
let obj: IdFunc<number> = {  
    id(value) { return value },  
    ids() { return [1, 3, 5] }  
}
```



# 目录

Contents

- ◆ 泛型函数
- ◆ 泛型约束
- ◆ 多个泛型参数
- ◆ 泛型接口
- ◆ 泛型类

## 泛型类

定义类的时候，结合泛型定义，就是泛型类。

```
class 类名<Type>{  
    // 内部可以使用 Type  
}
```

```
// 定义  
class Person <T> {  
    id: T  
  
    constructor(id: T) {  
        this.id = id  
    }  
    getId(): T {  
        return this.id  
    }  
}  
// 使用  
let p = new Person<number>(10)
```



传智教育旗下高端IT教育品牌