第二章：面向对象

面向对象是程序中一个非常重要的思想，它被很多同学理解成了一个比较难，比较深奥的问题，其实不然。面向对象很简单，简而言之就是程序之中所有的操作都需要通过对象来完成。

- 举例来说：
  - 操作浏览器要使用window对象
  - 操作网页要使用document对象
  - 操作控制台要使用console对象

一切操作都要通过对象，也就是所谓的面向对象，那么对象到底是什么呢？这就要先说到程序是什么，计算机程序的本质就是对现实事物的抽象，抽象的反义词是具体，比如：照片是对一个具体的人的抽象，汽车模型是对具体汽车的抽象等等。程序也是对事物的抽象，在程序中我们可以表示一个人、一条狗、一把枪、一颗子弹等等所有的事物。一个事物到了程序中就变成了一个对象。

在程序中所有的对象都被分成了两个部分数据和功能，以人为例，人的姓名、性别、年龄、身高、体重等属于数据，人可以说话、走路、吃饭、睡觉这些属于人的功能。数据在对象中被成为属性，而功能就被称为方法。所以简而言之，在程序中一切皆是对象。

1、类（class）

要想面向对象，操作对象，首先便要拥有对象，那么下一个问题就是如何创建对象。要创建对象，必须要先定义类，所谓的类可以理解为对象的模型，程序中可以根据类创建指定类型的对象，举例来说：可以通过Person类来创建人的对象，通过Dog类创建狗的对象，通过Car类来创建汽车的对象，不同的类可以用来创建不同的对象。

- 定义类：
  - ```
    class 类名 {
        属性名: 类型;

        constructor(参数: 类型){
            this.属性名 = 参数;
        }

        方法名(){
            ....
        }

    }
    ```
- 示例：
    ```

```
class Person{
        name: string;
        age: number;

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

        sayHello(){
                console.log(`大家好，我是${this.name}`);
        }
}
```

- 使用类：

```
const p = new Person('孙悟空', 18);
p.sayHello();
```

2、面向对象的特点

- 封装
  - 对象实质上就是属性和方法的容器，它的主要作用就是存储属性和方法，这就是所谓的封装
  - 默认情况下，对象的属性是可以任意的修改的，为了确保数据的安全性，在TS中可以对属性的权限进行设置
  - 只读属性（readonly）：
    - 如果在声明属性时添加一个readonly，则属性便成了只读属性无法修改
  - TS中属性具有三种修饰符：
    - public（默认值），可以在类、子类和对象中修改
    - protected，可以在类、子类中修改
    - private，可以在类中修改
  - 示例：
    - public

```
class Person{
        public name: string; // 写或什么都不写都是public
        public age: number;

        constructor(name: string, age: number){
                this.name = name; // 可以在类中修改
                this.age = age;
        }

        sayHello(){
                console.log(`大家好，我是${this.name}`);
        }
}

class Employee extends Person{
        constructor(name: string, age: number){
                super(name, age);
                this.name = name; //子类中可以修改
```

```
    }
}

const p = new Person('孙悟空', 18);
p.name = '猪八戒';// 可以通过对象修改
```

- protected

```
class Person{
    protected name: string;
    protected age: number;

    constructor(name: string, age: number){
        this.name = name; // 可以修改
        this.age = age;
    }

    sayHello(){
        console.log(`大家好，我是${this.name}`);
    }
}

class Employee extends Person{

    constructor(name: string, age: number){
        super(name, age);
        this.name = name; //子类中可以修改
    }
}

const p = new Person('孙悟空', 18);
p.name = '猪八戒';// 不能修改
```

- private

```
class Person{
    private name: string;
    private age: number;

    constructor(name: string, age: number){
        this.name = name; // 可以修改
        this.age = age;
    }

    sayHello(){
        console.log(`大家好，我是${this.name}`);
    }
}

class Employee extends Person{

    constructor(name: string, age: number){
        super(name, age);
        this.name = name; //子类中不能修改
    }
}

const p = new Person('孙悟空', 18);
```

```
p.name = '猪八戒';// 不能修改
```

- 属性存取器
  - 对于一些不希望被任意修改的属性，可以将其设置为private
  - 直接将其设置为private将导致无法再通过对象修改其中的属性
  - 我们可以在类中定义一组读取、设置属性的方法，这种对属性读取或设置的属性被称为属性的存取器
  - 读取属性的方法叫做setter方法，设置属性的方法叫做getter方法
  - 示例：

```
class Person{
        private _name: string;

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

        get name(){
                return this._name;
        }

        set name(name: string){
                this._name = name;
        }

}

const p1 = new Person('孙悟空');
console.log(p1.name); // 通过getter读取name属性
p1.name = '猪八戒'; // 通过setter修改name属性
```

- 静态属性
  - 静态属性（方法），也称为类属性。使用静态属性无需创建实例，通过类即可直接使用
  - 静态属性（方法）使用static开头
  - 示例：

```
class Tools{
        static PI = 3.1415926;

        static sum(num1: number, num2: number){
                return num1 + num2
        }
}

console.log(Tools.PI);
console.log(Tools.sum(123, 456));
```

- this
  - 在类中，使用this表示当前对象

- 继承
  - 继承时面向对象中的又一个特性
  - 通过继承可以将其他类中的属性和方法引入到当前类中

- 示例：

```
class Animal{
      name: string;
      age: number;

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

class Dog extends Animal{

      bark(){
            console.log(`${this.name}在汪汪叫！`);
      }
}

const dog = new Dog('旺财', 4);
dog.bark();
```

- 通过继承可以在不修改类的情况下完成对类的扩展
- 重写
  - 发生继承时，如果子类中的方法会替换掉父类中的同名方法，这就称为方法的重写
  - 示例：

```
class Animal{
      name: string;
      age: number;

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

      run(){
            console.log(`父类中的run方法！`);
      }
}

class Dog extends Animal{

      bark(){
            console.log(`${this.name}在汪汪叫！`);
      }

      run(){
            console.log(`子类中的run方法，会重写父类中的run方法！`);
      }
}

const dog = new Dog('旺财', 4);
dog.bark();
```

  - 在子类中可以使用super来完成对父类的引用

- 抽象类（abstract class）
  - 抽象类是专门用来被其他类所继承的类，它只能被其他类所继承不能用来创建实例
  - ```
    abstract class Animal{
        abstract run(): void;
        bark(){
            console.log('动物在叫~');
        }
    }

    class Dog extends Animals{
        run(){
            console.log('狗在跑~');
        }
    }
    ```
  - 使用abstract开头的方法叫做抽象方法，抽象方法没有方法体只能定义在抽象类中，继承抽象类时抽象方法必须要实现

3、接口（Interface）

接口的作用类似于抽象类，不同点在于接口中的所有方法和属性都是没有实值的，换句话说接口中的所有方法都是抽象方法。接口主要负责定义一个类的结构，接口可以去限制一个对象的接口，对象只有包含接口中定义的所有属性和方法时才能匹配接口。同时，可以让一个类去实现接口，实现接口时类中要保护接口中的所有属性。

- 示例（检查对象类型）：
  - ```
    interface Person{
        name: string;
        sayHello():void;
    }

    function fn(per: Person){
        per.sayHello();
    }

    fn({name:'孙悟空', sayHello() {console.log(`Hello, 我是 ${this.name}`)}});

    ```
- 示例（实现）

```
interface Person{
      name: string;
      sayHello():void;
}

class Student implements Person{
      constructor(public name: string) {
      }

      sayHello() {
            console.log('大家好，我是'+this.name);
      }
}
```

4、泛型（Generic）

定义一个函数或类时，有些情况下无法确定其中要使用的具体类型（返回值、参数、属性的类型不能确定），此时泛型便能够发挥作用。

- 举个例子：

```
function test(arg: any): any{
    return arg;
}
```

上例中，test函数有一个参数类型不确定，但是能确定的时其返回值的类型和参数的类型是相同的，由于类型不确定所以参数和返回值均使用了any，但是很明显这样做是不合适的，首先使用any会关闭TS的类型检查，其次这样设置也不能体现出参数和返回值是相同的类型

使用泛型：

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

这里的 `<T>` 就是泛型，T是我们给这个类型起的名字（不一定非叫T），设置泛型后即可在函数中使用T来表示该类型。所以泛型其实很好理解，就表示某个类型。

那么如何使用上边的函数呢？

- 方式一（直接使用）：

```
test(10)
```

使用时可以直接传递参数使用，类型会由TS自动推断出来，但有时编译器无法自动推断时还需要使用下面的方式

- 方式二（指定类型）：

```
test<number>(10)
```

也可以在函数后手动指定泛型

可以同时指定多个泛型，泛型间使用逗号隔开：

```
function test<T, K>(a: T, b: K): K{
      return b;
}

test<number, string>(10, "hello");
```

- 使用泛型时，完全可以将泛型当成是一个普通的类去使用

- 类中同样可以使用泛型：

```
class MyClass<T>{
      prop: T;

      constructor(prop: T){
            this.prop = prop;
      }
}
```

- 除此之外，也可以对泛型的范围进行约束

```
interface MyInter{
      length: number;
}

function test<T extends MyInter>(arg: T): number{
      return arg.length;
}
```

- 使用T extends MyInter表示泛型T必须是MyInter的子类，不一定非要使用接口类和抽象类同样适用。