第 21 章　接口和面向接口编程

当我们谈到接口的时候，通常会涉及以下几种含义，下面先简单介绍。

我们经常说一个库或者模块对外提供了某某API接口。通过主动暴露的接口来通信，可以隐藏软件系统内部的工作细节。这也是我们最熟悉的第一种接口含义。

第二种接口是一些语言提供的关键字，比如Java的`interface`。`interface` 关键字可以产生一个完全抽象的类。这个完全抽象的类用来表示一种契约，专门负责建立类与类之间的联系。

第三种接口即是我们谈论的“面向接口编程”中的接口，接口的含义在这里体现得更为抽象。用《设计模式》中的话说就是：

> 接口是对象能响应的请求的集合。

本章主要讨论的是第二种和第三种接口。首先要讲清楚的是，本章的前半部分都是针对Java语言的讲解，这是因为JavaScript并没有从语言层面提供对抽象类（`Abstract class`）或者接口（`interface`）的支持，我们有必要从一门提供了抽象类和接口的语言开始，逐步了解“面向接口编程”在面向对象程序设计中的作用。

21.1 回到Java的抽象类

首先让我们来回顾一下1.2节中的动物世界。目前我们有一个鸭子类Duck，还有一个让鸭子发出叫声的AnimalSound 类，该类有一个makeSound 方法，接收Duck 类型的对象作为参数，这几个类一直合作得很愉快，代码如下：

```
public class Duck {       // 鸭子类
    public void makeSound(){
        System.out.println( "嘎嘎嘎" );
    }
}

public class AnimalSound {
    public void makeSound( Duck duck ){     // (1) 只接受Duck类型的参数
        duck.makeSound();
    }
}

public class Test {
    public static void main( String args[] ){
        AnimalSound animalSound = new AnimalSound();
        Duck duck = new Duck();
        animalSound.makeSound( duck );     // 输出：嘎嘎嘎
    }
}
```

目前已经可以顺利地让鸭子发出叫声。后来动物世界里又增加了一些鸡，现在我们想让鸡也叫唤起来，但发现这是一件不可能完成的事情，因为在上面这段代码的(1)处，即AnimalSound 类的sound 方法里，被规定只能接受Duck 类型的对象作为参数：

```
public class Chicken {       // 鸡类
    public void makeSound(){
        System.out.println( "咯咯咯" );
    }
}

public class Test {
```

```
    public static void main( String args[] ){
        AnimalSound animalSound = new AnimalSound();
        Chicken chicken = new Chicken();
        animalSound.makeSound( chicken );
        // 报错，animalSound.makeSound只能接受Duck类型的参数
    }
}
```

在享受静态语言类型检查带来的安全性的同时，我们也失去了一些编写代码的自由。

通过1.3节的讲解，我们已经明白，静态类型语言通常设计为可以“向上转型”。当给一个类变量赋值时，这个变量的类型既可以使用这个类本身，也可以使用这个类的超类。就像看到天上有只麻雀，我们既可以说“一只麻雀在飞”，也可以说“一只鸟在飞”，甚至可以说成“一只动物在飞”。通过向上转型，对象的具体类型被隐藏在“超类型”身后。当对象类型之间的耦合关系被解除之后，这些对象才能在类型检查系统的监视下相互替换使用，这样才能看到对象的多态性。

所以如果想让鸡也叫唤起来，必须先把duck 对象和chicken 对象都向上转型为它们的超类型Animal 类，进行向上转型的工具就是抽象类或者interface 。我们即将使用的是抽象类。

先创建一个Animal 抽象类：

```
public abstract class Animal {
    abstract void makeSound();   // 抽象方法
}
```

然后让Duck 类和Chicken 类都继承自抽象类Animal ：

```
public class Chicken extends Animal{
    public void makeSound(){
        System.out.println( "咯咯咯" );
```

```
    }
}

public class Duck extends Animal{
    public void makeSound(){
        System.out.println( "嘎嘎嘎" );
    }
}
```

也可以把Animal 定义为一个具体类而不是抽象类，但一般不这么做。Scott Meyers曾指出，只要有可能，不要从具体类继承。

现在剩下的就是让AnimalSound 类的makeSound 方法接收Animal 类型的参数，而不是具体的Duck 类型或者Chicken 类型：

```
public class AnimalSound{
    public void makeSound( Animal animal ){ // 接收Animal类型的参数，而非Du
        animal.makeSound();
    }
}

public class Test {
    public static void main( String args[] ){
        AnimalSound animalSound = new AnimalSound ();
        Animal duck = new Duck();                          // 向上转型
        Animal chicken = new Chicken();                    // 向上转型
        animalSound.makeSound( duck );                     // 输出：嘎嘎嘎
        animalSound.makeSound( chicken );                  // 输出：咯咯咯
    }
}
```

本节通过抽象类完成了一个体现对象多态性的例子。但目前的重点并非讲解多态，而是在于说明抽象类。抽象类在这里主要有以下两个作用。

- 向上转型。让Duck 对象和Chicken 对象的类型都隐藏在Animal 类型身后，隐藏对象的具体类型之后，duck 对象和chicken 对象才能

被交换使用，这是让对象表现出多态性的必经之路。

- 建立一些契约。继承自抽象类的具体类都会继承抽象类里的abstract 方法，并且要求覆写它们。这些契约在实际编程中非常重要，可以帮助我们编写可靠性更高的代码。比如在命令模式中，各个子命令类都必须实现execute 方法，才能保证在调用command.execute 的时候不会抛出异常。如果让子命令类OpenTvCommand 继承自抽象类Command：

```
abstract class Command{
    public abstract void execute();
}

public class OpenTvCommand extends Command{
    public OpenTvCommand (){};
    public void execute(){
    System.out.println( "打开电视机" );
    }
}
```

那么自然有编译器帮助我们检查和保证子命令类OpenTvCommand 覆写了抽象类Command 中的execute 抽象方法。如果没有这样做，编译器会尽可能早地抛出错误来提醒正在编写这段代码的程序员。

总而言之，不关注对象的具体类型，而仅仅针对超类型中的“契约方法”来编写程序，可以产生可靠性高的程序，也可以极大地减少子系统实现之间的相互依赖关系，这就是我们本章要讨论的主题：

面向接口编程，而不是面向实现编程。

奇怪的是，本节我们一直讨论的是抽象类，跟接口又有什么关系呢？实际上这里的接口并不是指interface ，而是一个抽象的概念。

从过程上来看，“面向接口编程”其实是“面向超类型编程”。当对象的具体类型被隐藏在超类型身后时，这些对象就可以相互替换使用，我们的关注点才能从对象的类型上转移到对象的行为上。“面向接口编程”也可以看成面向抽象编程，即针对超类型中的abstract 方法编程，接口在这里被当

成abstract 方法中约定的契约行为。这些契约行为暴露了一个类或者对象能够做什么，但是不关心具体如何去做。

21.2 interface

除了用抽象类来完成面向接口编程之外，使用interface 也可以达到同样的效果。虽然很多人在实际使用中刻意区分抽象类和interface ，但使用interface 实际上也是继承的一种方式，叫作接口继承。

相对于单继承的抽象类，一个类可以实现多个interface 。抽象类中除了abstract 方法之外，还可以有一些供子类公用的具体方法。interface 使抽象的概念更进一步，它产生一个完全抽象的类，不提供任何具体实现和方法体（Java 8已经有了提供实现方法的interface ），但允许该interface 的创建者确定方法名、参数列表和返回类型，这相当于提供一些行为上的约定，但不关心该行为的具体实现过程。

interface 同样可以用于向上转型，这也是让对象表现出多态性的一条途径，实现了同一个接口的两个类就可以被相互替换使用。

再回到用抽象类实现让鸭子和鸡发出叫声的故事。这个故事得以完美收场的关键是让抽象类Animal 给duck 和chicken 进行向上转型。但此时也引入了一个限制，抽象类是基于单继承的，也就是说我们不可能让Duck 和Chicken 再继承自另一个家禽类。如果使用interface ，可以仅仅针对发出叫声这个行为来编写程序，同时一个类也可以实现多个interface 。

下面用interface 来改写基于抽象类的代码。我们先定义Animal 接口，所有实现了Animal 接口的动物类都将拥有Animal 接口中约定的行为：

```
public interface Animal{
    abstract void makeSound();
}

public class Duck implements Animal{
    public void makeSound() {       // 重写Animal接口的makeSound抽象方法
        System.out.println( "嘎嘎嘎" );
    }
}

public class Chicken implements Animal{
    public void makeSound() {      // 重写Animal接口的makeSound抽象方法
```

```
        System.out.println( "咯咯咯" );
    }
}

public class AnimalSound {
    public void makeSound( Animal animal ){
        animal.makeSound();
    }
}

public class Test {
    public static void main( String args[] ){
        Animal duck = new Duck();
        Animal chicken = new Chicken();

        AnimalSound animalSound = new AnimalSound();
        animalSound.makeSound( duck );      // 输出：嘎嘎嘎
        animalSound.makeSound( chicken );      // 输出：咯咯咯
    }
}
```

21.3 JavaScript语言是否需要抽象类和interface

通过前面的讲解，我们明白了抽象类和interface 的作用主要都是以下两点。

- 通过向上转型来隐藏对象的真正类型，以表现对象的多态性。
- 约定类与类之间的一些契约行为。

对于JavaScript而言，因为JavaScript是一门动态类型语言，类型本身在JavaScript中是一个相对模糊的概念。也就是说，不需要利用抽象类或者interface 给对象进行“向上转型”。除了number 、string 、boolean 等基本数据类型之外，其他的对象都可以被看成“天生”被“向上转型”成了Object 类型：

```
var ary = new Array();
var date = new Date();
```

如果JavaScript是一门静态类型语言，上面的代码也许可以理解为：

```
Array ary = new Array();
Date date = new Date();
```

或者：

```
Object ary = new Array();
Object date = new Date();
```

很少有人在JavaScript开发中去关心对象的真正类型。在动态类型语言中，对象的多态性是与生俱来的，但在另外一些静态类型语言中，对象类型之间的解耦非常重要，甚至有一些设计模式的主要目的就是专门隐藏对象的真正类型。

因为不需要进行向上转型，接口在JavaScript中的最大作用就退化到了检查代码的规范性。比如检查某个对象是否实现了某个方法，或者检查是否给函数传入了预期类型的参数。如果忽略了这两点，有可能会在代码中留下一些隐藏的bug。比如我们尝试执行obj 对象的show 方法，但是obj 对象本身却没有实现这个方法，代码如下：

```
function show( obj ){
    obj.show();         // Uncaught TypeError: undefined is not a functic
}

var myObject = {};     // myObject对象没有show方法
show( myObject );
```

或者：

```
function show( obj ){
    obj.show();         // TypeError: number is not a function
}

var myObject = {     // myObject.show不是Function类型
    show: 1
};
show( myObject );
```

此时，我们不得不加上一些防御性代码：

```
function show( obj ){
    if ( obj && typeof obj.show === 'function' ){
        obj.show();
    }
}
```

或者：

```
function show( obj ){
    try{
        obj.show();
    }catch( e ){

    }
}

var myObject = {};     // myObject对象没有show方法
// var myObject = {     // myObject.show不是Function类型
    // show: 1
// };

show( myObject );
```

如果JavaScript有编译器帮我们检查代码的规范性，那事情要比现在美好得多，我们不用在业务代码中到处插入一些跟业务逻辑无关的防御性代码。作为一门解释执行的动态类型语言，把希望寄托在编译器上是不可能了。如果要处理这类异常情况，我们只有手动编写一些接口检查的代码。

21.4 用鸭子类型进行接口检查

在1.2节中，我们已经了解过鸭子类型的概念：

“如果它走起路来像鸭子，叫起来也是鸭子，那么它就是鸭子。”

鸭子类型是动态类型语言面向对象设计中的一个重要概念。利用鸭子类型的思想，不必借助超类型的帮助，就能在动态类型语言中轻松地实现本章提到的设计原则：面向接口编程，而不是面向实现编程。比如，一个对象如果有`push`和`pop`方法，并且提供了正确的实现，它就能被当作栈来使用；一个对象如果有`length`属性，也可以依照下标来存取属性，这个对象就可以被当作数组来使用。如果两个对象拥有相同的方法，则有很大的可能性它们可以被相互替换使用。

在`Object.prototype.toString.call( [] ) === '[object Array]'`被发现之前，我们经常用鸭子类型的思想来判断一个对象是否是一个数组，代码如下：

```
var isArray = function( obj ){
    return obj &&
        typeof obj === 'object' &&
        typeof obj.length === 'number' &&
        typeof obj.splice === 'function'
};
```

当然在JavaScript开发中，总是进行接口检查是不明智的，也是没有必要的，毕竟现在还找不到一种好用并且通用的方式来模拟接口检查，跟业务逻辑无关的接口检查也会让很多JavaScript程序员觉得不值得和不习惯。在Ross Harmes和Dustin Diaz合著的***Pro JavaScript Design Pattrns*** 一书中，提供了一种根据鸭子类型思想模拟接口检查的方法，但这种基于双重循环的检查方法并不是很实用，而且只能检查对象的某个属性是否属于`Function`类型。

21.5 用TypeScript编写基于interface的命令模式

虽然在大多数时候interface给JavaScript开发带来的价值并不像在静态类型语言中那么大，但如果我们正在编写一个复杂的应用，还是会经常怀念接口的帮助。

下面我们以基于命令模式的示例来说明interface如何规范程序员的代码编写，这段代码本身并没有什么实用价值，在JavaScript中，我们一般用闭包和高阶函数来实现命令模式。

假设我们正在编写一个用户界面程序，页面中有成百上千个子菜单。因为项目很复杂，我们决定让整个程序都基于命令模式来编写，即编写菜单集合界面的是某个程序员，而负责实现每个子菜单具体功能的工作交给了另外一些程序员。

那些负责实现子菜单功能的程序员，在完成自己的工作之后，会把子菜单封装成一个命令对象，然后把这个命令对象交给编写菜单集合界面的程序员。他们已经约定好，当调用子菜单对象的execute方法时，会执行对应的子菜单命令。

虽然在开发文档中详细注明了每个子菜单对象都必须有自己的execute方法，但还是有一个粗心的JavaScript程序员忘记给他负责的子菜单对象实现execute方法，于是当执行这个命令的时候，便会报出错误，代码如下：

```
<html>
    <body>
        <button id="exeCommand">执行菜单命令</button>
    <script>
    var RefreshMenuBarCommand = function(){};

    RefreshMenuBarCommand.prototype.execute = function(){
        console.log( '刷新菜单界面' );
    };

    var AddSubMenuCommand = function(){};

    AddSubMenuCommand.prototype.execute = function(){
        console.log( '增加子菜单' );
    };
```

```
    var DelSubMenuCommand = function(){};

    /*****没有实现DelSubMenuCommand.prototype.execute *****/
    // DelSubMenuCommand.prototype.execute = function(){

    // };

    var refreshMenuBarCommand = new RefreshMenuBarCommand(),
        addSubMenuCommand = new AddSubMenuCommand(),
        delSubMenuCommand = new DelSubMenuCommand();

    var setCommand = function( command ){
        document.getElementById( 'exeCommand' ).onclick = function(){
            command.execute();
        }
    };

    setCommand( refreshMenuBarCommand );
    // 点击按钮后输出："刷新菜单界面"
    setCommand( addSubMenuCommand );
    // 点击按钮后输出："增加子菜单"
    setCommand( delSubMenuCommand );
    // 点击按钮后报错。Uncaught TypeError: undefined is not a function

     </script>
    </body>
</html>
```

为了防止粗心的程序员忘记给某个子命令对象实现execute 方法，我们只能在高层函数里添加一些防御性的代码，这样当程序在最终被执行的时候，有可能抛出异常来提醒我们，代码如下：

```
var setCommand = function( command ){
    document.getElementById( 'exeCommand' ).onclick = function(){
        if ( typeof command.execute !== 'function' ){
            throw new Error( "command对象必须实现execute方法" );
        }
        command.execute();
    }
};
```

如果确实不喜欢重复编写这些防御性代码，我们还可以尝试使用TypeScript来编写这个程序。

TypeScript是微软开发的一种编程语言，是JavaScript的一个超集。跟CoffeeScript类似，TypeScript代码最终会被编译成原生的JavaScript代码执行。通过TypeScript，我们可以使用静态语言的方式来编写JavaScript程序。用TypeScript来实现一些设计模式，显得更加原汁原味。

TypeScript目前的版本还没有提供对抽象类的支持，但是提供了`interface`。下面我们就来编写一个TypeScript版本的命令模式。

首先定义`Command`接口：

```
interface Command{
    execute: Function;
}
```

接下来定义`RefreshMenuBarCommand`、`AddSubMenuCommand`和`DelSubMenuCommand`这3个类，它们分别都实现了`Command`接口，这可以保证它们都拥有`execute`方法：

```
class RefreshMenuBarCommand implements Command{
    constructor (){
    }
    execute(){
        console.log( '刷新菜单界面' );
    }
}

class AddSubMenuCommand implements Command{
    constructor (){
    }
    execute(){
        console.log( '增加子菜单' );
    }
}

class DelSubMenuCommand implements Command{
```

```
    constructor (){
    }
    // 忘记重写execute方法
}

var refreshMenuBarCommand = new RefreshMenuBarCommand(),
    addSubMenuCommand = new AddSubMenuCommand(),
    delSubMenuCommand = new DelSubMenuCommand();

refreshMenuBarCommand.execute();     // 输出：刷新菜单界面
addSubMenuCommand.execute();         // 输出：增加子菜单
delSubMenuCommand.execute();    // 输出：Uncaught TypeError: undefined is
```

如图21-1所示，当我们忘记在DelSubMenuCommand 类中重写execute方法时，TypeScript提供的编译器及时给出了错误提示。

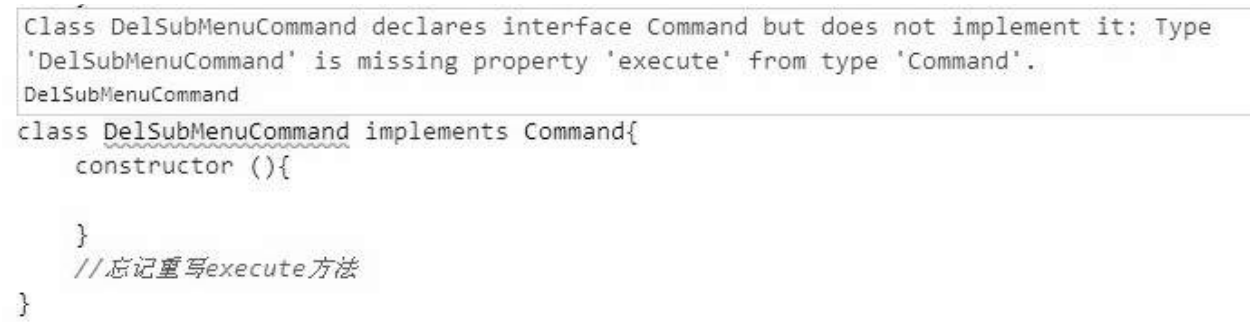

图　21-1

这段TypeScript代码翻译过来的JavaScript代码如下：

```
var RefreshMenuBarCommand = (function () {
    function RefreshMenuBarCommand() {
    }
    RefreshMenuBarCommand.prototype.execute = function () {
        console.log('刷新菜单界面');
    };
    return RefreshMenuBarCommand;
})();

var AddSubMenuCommand = (function () {
    function AddSubMenuCommand() {
    }
    AddSubMenuCommand.prototype.execute = function () {
```

```
        console.log('增加子菜单');
    };
    return AddSubMenuCommand;
})();

var DelSubMenuCommand = (function () {
    function DelSubMenuCommand() {
    }
    return DelSubMenuCommand;
})();

var refreshMenuBarCommand = new RefreshMenuBarCommand(),
    addSubMenuCommand = new AddSubMenuCommand(),
    delSubMenuCommand = new DelSubMenuCommand();

refreshMenuBarCommand.execute();
addSubMenuCommand.execute();
delSubMenuCommand.execute();
```