

## 实验一： 使用接口实现静态代理和动态代理

### 实验描述：

在 Java Web 开发中，同样追求高内聚低耦合，通俗来讲就是一个模块的修改不应或应较少地影响其他模块，不能牵一发而动全身，否则代码的可维护性、扩展性就很差。假设 A 调用 B，出于某些原因（解耦、保护 B、扩展 B），B 不想让 A 直接调用，就创建一个中间代理者 C 来代理 B。A 调用 C，**C 中调用 B 的功能和 C 扩展的部分**。

代理分为静态代理和动态代理。静态代理需要为每一个被代理类创建代理类，如果被代理类很多那代理类也会很多；动态代理则是动态地去代理需要被代理的类。

静态代理的实现需要：

1. 一个接口，接口中是之后需要调用的方法
2. 被代理类，被代理类实现这个接口
3. 代理类，代理类也实现这个接口。且代理类中要显式地声明代理的是哪个类，并创建这个被代理类的实例，然后实现接口中方法的时候调用被代理类的方法

JDK 动态代理的实现需要：

1. 一个接口，接口中是之后需要调用的方法
2. 被代理类，被代理类实现这个接口
3. 代理类，要实现 `InvocationHandler` 接口(`java.lang.reflect.InvocationHandler`)。代理类要声明一个被代理类的引用作为成员变量（不初始化），在有参构造方法中初始化这个被代理类的引用
4. 在 `invoke()`中实现对需要调用的方法的调用

以上只是简单列出实现步骤，具体内容可再详细查阅相关资源。

### 实验要求：

1. 创建一个接口 `People`，接口中有一个方法 `speak()`；
2. 实现静态代理：
  - (1) 创建 `Chinese` 类实现接口 `People`，`speak()`方法中输出一句话即可
  - (2) 创建 `ChineseProxy` 类，实现接口 `People`，实现对 `Chinese` 类的代理
  - (3) 思考如果要再创建一个新的 `Russian` 类，如果要对 `Russian` 类也实现静态代理，要怎么做。无需代码实现，文字体现在报告中即可
3. 实现 JDK 动态代理：
  - (1) 创建 `English` 类实现接口 `People`，`speak()`方法中同样输出一句话即可
  - (2) 创建 `PeopleProxy` 类，实现接口 `InvocationHandler`，实现其有参构造方法，并实现 `invoke` 方法
  - (3) 再创建一个 `American` 类实现接口 `People`
4. 创建 `Tester` 类  
在 `Tester` 类中，测试 `ChineseProxy` 类是否能成功实现静态代理 `Chinese`、执行 `speak` 方法；测试 `PeopleProxy` 类是否能成功实现动态代理 `English` 和 `American` 类

## 实验过程

### • People接口

```

1 public interface People {
2     //接口中的方法不能有方法体
3     public void speak();
4
5 }
6

```

## 静态代理

- 被代理的真实对象

```

1 //真实对象
2 public class Chinese implements People{
3     private String name="Chinese";
4     public Chinese(){
5     }
6     public Chinese(String name){
7         this.name=name;
8     }
9     //重写People中的方法
10    public void speak(){
11        System.out.println(name+" speaks "+name);
12        System.out.println("After speaking\n");
13    }
14
15    public String getName() {
16        return name;
17    }
18 }
19

```

- 代理对象（静态）

```

1 //代理角色
2 public class ChineseProxy implements People{
3
4     //需要代理的对象
5     private Chinese people;
6     public ChineseProxy(Chinese people){
7         this.people=people;
8     }
9     //方法重写
10    @Override
11    public void speak(){
12        System.out.println("Using static proxy");
13        System.out.println("In "+people.getName()+" Proxy, before
speaking");
14        people.speak();
15    }
16 }
17

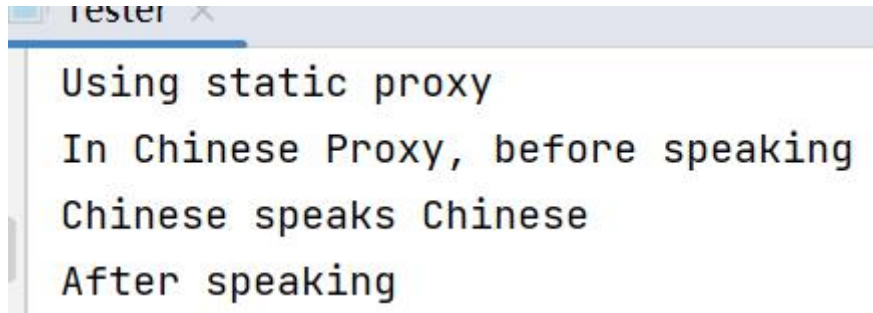
```

- 在Tester中调用静态代理

```

1 //静态代理
2 //被代理对象
3 Chinese chinese=new Chinese();
4 //代理对象，将被代理对象传给代理对象
5 ChineseProxy chineseProxy=new ChineseProxy(chinese);
6 //由代理对象调用speak函数
7 chineseProxy.speak();

```



Tester

Using static proxy

In Chinese Proxy, before speaking

Chinese speaks Chinese

After speaking

## 动态代理

- 被代理的真实对象1----English

```

1 public class English implements People{
2     private String name="English";
3     public English(){
4
5     }
6     public English(String name){
7         this.name=name;
8     }
9     @Override
10    public void speak(){
11        System.out.println(name+" speaks "+name);
12        System.out.println("After speaking\n");
13    }
14 }
15

```

- 被代理的真实对象1----American

```

1 public class American implements People{
2     private String name="USA";
3     public American(){
4
5     }
6     public American(String name){
7         this.name=name;
8     }
9     @Override
10    public void speak(){
11        System.out.println(name+" speaks "+name);
12        System.out.println("After speaking\n");
13    }
14 }
15

```

- 代理对象 (动态)

```
1 import java.lang.reflect.InvocationHandler;
2 import java.lang.reflect.Method;
3 import java.lang.reflect.Proxy;
4
5 public class PeopleProxy implements InvocationHandler {
6
7     //要代理的真实对象
8     private Object target;
9     public PeopleProxy(Object target){
10         this.target=target;
11     }
12     //在InvocationHandler接口中的方法，用于处理代理类对象的方法调用
13     public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable{
14         System.out.println("Using dynamic proxy");
15         System.out.println("In proxy,before speaking");
16         method.invoke(target,args);
17         return null;
18     }
19
20 }
21
```

- 在Tester中调用JDK动态代理

```
1 //动态代理
2     //被代理对象
3     People english=new English();
4     //代理对象，将被代理对象传入
5     InvocationHandler handler=new PeopleProxy(english);
6     //获取真实对象的类加载器
7     ClassLoader loader = english.getClass().getClassLoader();
8     //获取真实对象实现的接口列表
9     Class[] interfaces = english.getClass().getInterfaces();
10    //通过Proxy.newProxyInstance()方法生成动态代理类对象
11    People subject = (People) Proxy.newProxyInstance(loader, interfaces,
    handler);
12    //        //打印动态代理对象的类型，即代理类的全限定名
13    //        System.out.println("动态代理对象的类
    型: "+subject.getClass().getName());
14    //调用方法
15    subject.speak();
16
17    //被代理对象
18    People USA=new American();
19    //代理对象，将被代理对象传入
20    InvocationHandler handler2=new PeopleProxy(USA);
21    //获取真实对象的类加载器
22    ClassLoader loader2 = USA.getClass().getClassLoader();
23    //获取真实对象实现的接口列表
24    Class[] interfaces2 = USA.getClass().getInterfaces();
```

```
25 //通过Proxy.newProxyInstance()方法生成动态代理类对象
26 People subject2 = (People) Proxy.newProxyInstance(loader2,
    interfaces2, handler2);
27 // //打印动态代理对象的类型，即代理类的全限定名
28 // System.out.println("动态代理对象的类
    型: "+subject2.getClass().getName());
29 //调用方法
30 subject2.speak();
```

```
Using dynamic proxy
In proxy,before speaking
English speaks English
After speaking
```

```
Using dynamic proxy
In proxy,before speaking
USA speaks USA
After speaking
```

---

## 实验结果

---

Using static proxy  
In Chinese Proxy, before speaking  
Chinese speaks Chinese  
After speaking

Using dynamic proxy  
In proxy, before speaking  
English speaks English  
After speaking

Using dynamic proxy  
In proxy, before speaking  
USA speaks USA  
After speaking

---

## 实验二：简单的工资系统

---



### 实验要求：

编写一个简单的工资系统，实现不同类型员工工资的查询。如果当月是某个员工的生日月，工资增加 100 元。

具体要求如下：

1. Employee 抽象类
  - (1) 包含成员变量 name, id, birthday, 其中 birthday 是 MyDate 类的对象
  - (2) 抽象方法 getSalary()
  - (3) toString()方法输出对象的信息
2. MyDate 类
  - (1) 包含成员变量 year, month, day
  - (2) toDateString()方法输出日期：xxxx 年 xx 月 xx 日
3. FullTimeEmployee 类继承 Employee 类
  - (1) 该类的有参构造方法直接指定成员的 name, id, birthday, salary 四个信息。salary 就是本月工资
  - (2) 实现 getSalary()方法，返回 salary 即可
  - (3) toString()方法输出员工类型及员工信息
4. PartTimeEmployee 类继承自 Employee 类
  - (1) 该类的有参构造方法指定成员的 name, id, birthday, workHours, hourSalary 五个信息
  - (2) 实现 getSalary()方法， $salary = workHours * hourSalary$
  - (3) toString()方法输出员工类型及员工信息
5. 测试类 Tester 中创建不同类型的员工。获取当前月份（6 月），输出这些员工的信息及工资，如果本月是某些员工的生日月，查询出来的工资多加 100 元

---

## 实验过程

按要求写即可。

### Employee抽象类

```
1 package com.s2;
2 //定义一个抽象类
3 public abstract class Employee {
4     protected String name;
5     protected String id;
6     protected MyDate birthday;
7     public Employee(){
8         name="null";
9         id="null";
10        birthday=new MyDate(0,0,0);
11    }
12    public Employee(String name,String id,int year,int month,int day){
13        this.name=name;
14        this.id=id;
15        birthday=new MyDate(year,month,day);
16    }
17    //抽象方法
18    public abstract int getSalary();
19
20    public String toString(){
```

```
21     return "name: "+name+"\n"+"id:"+id+"\n"+"birthday:"+birthday.toString()+
22         "\n";
23     }
24 }
```

## MyDate类

```
1  package com.s2;
2
3  public class MyDate {
4      private int year;
5      private int month;
6      private int day;
7      public MyDate(){
8          year=0;
9          month=0;
10         day=0;
11     }
12     public MyDate(int year,int month,int day){
13         this.year=year;
14         this.month=month;
15         this.day=day;
16     }
17     public String toString(){
18         return year+"年"+month+"月"+day+"日";
19     }
20
21     public void setDay(int day) {
22         this.day = day;
23     }
24
25     public void setMonth(int month) {
26         this.month = month;
27     }
28
29     public void setYear(int year) {
30         this.year = year;
31     }
32
33     public int getDay() {
34         return day;
35     }
36
37     public int getMonth() {
38         return month;
39     }
40
41     public int getYear() {
42         return year;
43     }
44 }
45
```



## 全职员工类

```
1 package com.s2;
2
3 import javax.print.DocFlavor;
4
5 public class FullTimeEmployee extends Employee{
6     private int salary;
7
8     public FullTimeEmployee(){
9         super();
10        salary=0;
11    }
12    public FullTimeEmployee(String name,String id,int year,int month,int
day,int salary){
13        super(name,id,year,month,day);
14        this.salary=salary;
15    }
16
17    @Override
18    public int getSalary() {
19        //若本月为其生日
20        if (birthday.getMonth()==6)
21            return salary+100;
22        else
23            return salary;
24    }
25    public String toString(){
26        return super.toString()+"salary:"+getSalary()+"\n";
27    }
28 }
29
```

## 兼职员工类

```
1 package com.s2;
2
3 public class PartTimeEmployee extends Employee{
4     private int workHours;
5     private int hourSalary;
6     public PartTimeEmployee(){
7         super();
8         workHours=0;
9         hourSalary=0;
10    }
11    public PartTimeEmployee(String name,String id,int year,int month,int
day,int workHours,int hourSalary){
12        super(name,id,year,month,day);
13        this.hourSalary=hourSalary;
14        this.workHours=workHours;
15    }
16    @Override
17    public int getSalary(){
18        //若本月为其生日
19        if (birthday.getMonth()==6)
```

```

20         return workHours*hourSalary+100;
21     else
22         return workHours*hourSalary;
23     }
24
25     @Override
26     public String toString() {
27         return
28         super.toString()+"workHours:"+workHours+"\n"+"hourSalary:"+hourSalary+"\n"+"
29         salary:"+getSalary()+"\n";
30     }
31 }

```

## 测试类

```

1 package com.s2;
2
3 public class Tester {
4     public static void main(String[] args) {
5         FullTimeEmployee p1=new FullTimeEmployee("小明","1",2000,3,4,5000);
6         FullTimeEmployee p2=new FullTimeEmployee("小红","2",2000,6,4,5000);
7
8         PartTimeEmployee p3=new PartTimeEmployee("李华","3",2000,3,4,8,600);
9         PartTimeEmployee p4=new PartTimeEmployee("张伟","3",2000,6,4,8,600);
10
11         System.out.println(p1.toString());
12         System.out.println(p2.toString());
13         System.out.println(p3.toString());
14         System.out.println(p4.toString());
15     }
16 }
17

```

```
Community Edition 2023.1\bin" -Dfile.encoding=UTF-8 -classpath
E:\Javafile\out\production\Project05 com.s2.Tester

name: 小明
id:1
birthday:2000年3月4日
salary:5000

name: 小红
id:2
birthday:2000年6月4日
salary:5100

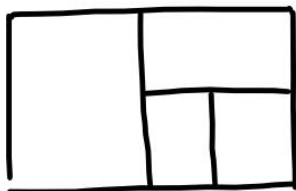
name: 李华
id:3
birthday:2000年3月4日
workHours:8
hourSalary:600
Salary:4800

name: 张伟
id:3
birthday:2000年6月4日
workHours:8
hourSalary:600
Salary:4900
```

## 实验三：Image Gallery

**描述：**请实现一个 JavaFX 程序，满足以下要求：

1. 程序应当从一个指定目录（例如：“images/”）中读取前 4 张图片。
2. 使用 JavaFX 布局管理器（如 `BorderPane`、`VBox` 和 `HBox`），按照以下布局显示



3. 窗口的大小应固定为 1080 像素宽和 720 像素高。
4. 图片应调整大小或进行截取以适应窗口大小。

**提示：**可以使用 JavaFX 的 `ImageView`、`BorderPane`、`VBox` 和 `HBox` 组件来实现图片的显示和布局。

## 实验过程

## 核心代码

- 固定窗口大小

```
1 //固定窗口大小
2 private static final int WINDOW_WIDTH = 1080;
3 private static final int WINDOW_HEIGHT = 720;
4 private static final int MAX_IMAGES = 4;
5 private static final String IMAGE_DIRECTORY = "E:\\桌面\\java实验";
6
7 public static void main(String[] args) {
8     launch(args);
9 }
```

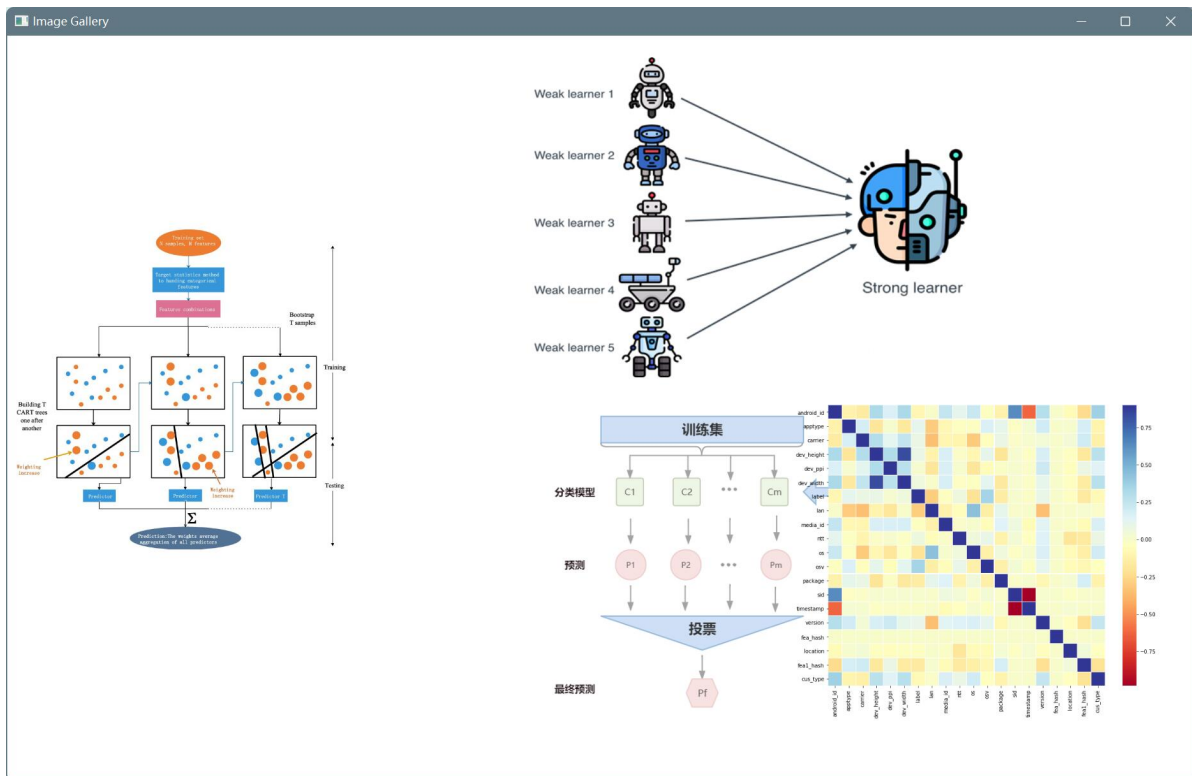
- 使用指定布局

```
1 private GridPane createImageGrid() {
2     GridPane imageGrid = new GridPane();
3
4     List<File> imageFiles = getImageFiles();
5     int numImages = Math.min(imageFiles.size(), 4);
6     int[] col = {0,1,1,2};
7     int[] row = {0,0,1,1};
8
9     //占几列、几行
10    int[] columnSpan={1,2,1,1};
11    int[] rowSpan={2,1,1,1};
12
13    // 自定义图片布局的坐标和尺寸
14    //3x2的列表
15    double[] columnWidths = { 540,270,270 };
16    double[] rowHeights = { 360,360};
17
18    for (int i = 0; i < numImages; i++) {
19        File file = imageFiles.get(i);
20        ImageView imageView = createImageView(file);
21        GridPane.setConstraints(imageView, col[i], row[i],
columnSpan[i], rowSpan[i]);
22
23        // 设置网格列宽和行高
24        ColumnConstraints colConstraints = new
ColumnConstraints(columnWidths[col[i]]);
25        imageGrid.getColumnConstraints().add(colConstraints);
26
27        RowConstraints rowConstraints = new
RowConstraints(rowHeights[row[i]]);
28        imageGrid.getRowConstraints().add(rowConstraints);
29        // 将ImageView添加到网格布局中
30        imageGrid.getChildren().add(imageview);
31    }
32
33    return imageGrid;
34 }
```

- 图片自适应大小

```
1 private ImageView createImageView(File file) {
2     Image image = new Image(file.toURI().toString());
3     ImageView imageView = new ImageView(image);
4     imageView.setPreserveRatio(true);
5
6     // 计算适应宽度和适应高度
7     double windowwidth = WINDOW_WIDTH - 20; // 减去边距的宽度
8     double windowHeight = WINDOW_HEIGHT - 20; // 减去边距的高度
9     double fitwidth = windowwidth / 2 - 15;
10    double fitHeight = windowHeight / 2 - 15;
11
12    imageView.setFitWidth(fitwidth);
13    imageView.setFitHeight(fitHeight);
14
15    return imageView;
16 }
```

## 实现效果



## 实验四：设计一个JavaFX

**描述：** 在本次编程任务中，您需要使用 JavaFX 框架设计一个计算器应用程序的静态界面。请确保计算器界面美观、易用且符合以下要求：

**要求：**

1. 计算器的尺寸应适中，以方便用户在各种屏幕尺寸上使用。
2. 计算器的按钮应呈长方形，以便用户更容易点击。
3. 按钮的布局应清晰易懂，遵循常见的计算器布局。例如，将数字按钮按照 3x3 的网格排列，将加、减、乘、除等运算符按钮放在数字按钮的右侧。
4. 计算器的颜色设置应保持视觉上的区分。例如，可以将数字按钮设置为浅色背景，运算符按钮设置为深色背景，以便用户快速识别。
5. 计算器应具备一个清楚易读的文本框，用于显示用户输入的数字和计算结果。建议将文本框字体设置更大些，以便用户在屏幕上轻松阅读。

**提示：**

1. 通过将组件添加到 GridPane 中的适当行和列，你可以轻松地实现清晰的布局。
2. 使用 JavaFX 的预定义控件，如 Button、TextField 和 Label，以便更好地满足项目需求。

---

## 实验过程

---

**核心代码：**

```
1 public class CalculatorApp extends Application {
2
3     @Override
4     public void start(Stage primaryStage) {
5         primaryStage.setTitle("Calculator");
6
7         // 创建一个 GridPane 用于放置按钮
8         GridPane buttonGrid = createButtonGrid();
9
10        // 创建一个 TextField 用于显示结果
11        TextField displayField = createDisplayField();
12
13        // 创建一个 VBox 用于放置显示结果和按钮
14        VBox vbox = new VBox(10);
15        vbox.setAlignment(Pos.CENTER);
16        vbox.setPadding(new Insets(10));
17        vbox.getChildren().addAll(displayField, buttonGrid);
18
19        // 创建一个场景并将其设置在舞台上
20        Scene scene = new Scene(vbox);
21        primaryStage.setScene(scene);
22
23        primaryStage.show();
```



```

24     }
25
26     private GridPane createButtonGrid() {
27         GridPane gridPane = new GridPane();
28         gridPane.setAlignment(Pos.CENTER);
29         gridPane.setHgap(5);
30         gridPane.setVgap(5);
31
32         // 创建数字按钮
33         for (int i = 0; i < 9; i++) {
34             Button numberButton = createNumberButton(String.valueOf(i + 1));
35             gridPane.add(numberButton, i % 3, i / 3);
36         }
37
38         // 创建运算符按钮
39         Button addButton = createOperatorButton("+");
40         Button subtractButton = createOperatorButton("-");
41         Button multiplyButton = createOperatorButton("*");
42         Button divideButton = createOperatorButton("/");
43
44         gridPane.add(addButton, 3, 0);
45         gridPane.add(subtractButton, 3, 1);
46         gridPane.add(multiplyButton, 3, 2);
47         gridPane.add(divideButton, 3, 3);
48
49         return gridPane;
50     }
51
52     private Button createNumberButton(String number) {
53         Button button = new Button(number);
54         button.setPrefWidth(60);
55         button.setPrefHeight(40);
56         button.setStyle("-fx-background-color: #EFEFEF;");
57         return button;
58     }
59
60     private Button createOperatorButton(String operator) {
61         Button button = new Button(operator);
62         button.setPrefWidth(60);
63         button.setPrefHeight(40);
64         button.setStyle("-fx-background-color: #333333; -fx-text-fill:
white;");
65         return button;
66     }
67
68     private TextField createDisplayField() {
69         TextField textField = new TextField();
70         textField.setEditable(false);
71         textField.setAlignment(Pos.CENTER_RIGHT);
72         textField.setStyle("-fx-font-size: 18;");
73         return textField;
74     }
75
76     public static void main(String[] args) {
77         launch();

```

78	}
79	}

## 实验结果

