

1.Lambda表达式

1.1体验Lambda表达式【理解】

- 案例需求
启动一个线程，在控制台输出一句话：多线程程序启动了
- 实现方式一
 - 实现步骤
 - 定义一个类MyRunnable实现Runnable接口，重写run()方法
 - 创建MyRunnable类的对象
 - 创建Thread类的对象，把MyRunnable的对象作为构造参数传递
 - 启动线程
- 实现方式二
 - 匿名内部类的方式改进
- 实现方式三
 - Lambda表达式的方式改进
- 代码演示

```
//方式一的线程类
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("多线程程序启动了");
    }
}

public class LambdaDemo {
    public static void main(String[] args) {
        //方式一
        // MyRunnable my = new MyRunnable();
        // Thread t = new Thread(my);
        // t.start();

        //方式二
        // new Thread(new Runnable() {
        //     @Override
        //     public void run() {
        //         System.out.println("多线程程序启动了");
        //     }
        // }).start();

        //方式三
        new Thread( () -> {
            System.out.println("多线程程序启动了");
        } ).start();
    }
}
```

```
}  
}
```

- 函数式编程思想概述

函数式思想则尽量忽略面向对象的复杂语法：“强调做什么，而不是以什么形式去做”

而我们要学习的Lambda表达式就是函数式思想的体现

1.2 Lambda表达式的标准格式【理解】

- 格式：

(形式参数) -> {代码块}

- 形式参数：如果有多个参数，参数之间用逗号隔开；如果没有参数，留空即可
- ->：由英文中画线和大于符号组成，固定写法。代表指向动作
- 代码块：是我们具体要做的事情，也就是以前我们写的方法体内容

- 组成Lambda表达式的三要素：

- 形式参数，箭头，代码块

1.3 Lambda表达式练习1【应用】

- Lambda表达式的使用前提

- 有一个接口
- 接口中有且仅有一个抽象方法

- 练习描述

无参无返回值抽象方法的练习

- 操作步骤

- 定义一个接口(Eatable)，里面定义一个抽象方法：void eat();
- 定义一个测试类(EatableDemo)，在测试类中提供两个方法
 - 一个方法是：useEatable(Eatable e)
 - 一个方法是主方法，在主方法中调用useEatable方法

- 示例代码

```
//接口  
public interface Eatable {  
    void eat();  
}  
//实现类  
public class EatableImpl implements Eatable {  
    @Override  
    public void eat() {  
        System.out.println("一天一苹果，医生远离我");  
    }  
}  
//测试类  
public class EatableDemo {  
    public static void main(String[] args) {  
        //在主方法中调用useEatable方法  
        Eatable e = new EatableImpl();  
    }  
}
```

```

        useEatable(e);

        //匿名内部类
        useEatable(new Eatable() {
            @Override
            public void eat() {
                System.out.println("一天一苹果，医生远离我");
            }
        });

        //Lambda表达式
        useEatable(() -> {
            System.out.println("一天一苹果，医生远离我");
        });
    }

    private static void useEatable(Eatable e) {
        e.eat();
    }
}

```

1.4Lambda表达式练习2【应用】

- 练习描述
 - 有参无返回值抽象方法的练习
- 操作步骤
 - 定义一个接口(Flyable)，里面定义一个抽象方法：void fly(String s);
 - 定义一个测试类(FlyableDemo)，在测试类中提供两个方法
 - 一个方法是：useFlyable(Flyable f)
 - 一个方法是主方法，在主方法中调用useFlyable方法
- 示例代码

```

public interface Flyable {
    void fly(String s);
}

public class FlyableDemo {
    public static void main(String[] args) {
        //在主方法中调用useFlyable方法
        //匿名内部类
        useFlyable(new Flyable() {
            @Override
            public void fly(String s) {
                System.out.println(s);
                System.out.println("飞机自驾游");
            }
        });
        System.out.println("-----");

        //Lambda
    }
}

```

```

        useFlyable((String s) -> {
            System.out.println(s);
            System.out.println("飞机自驾游");
        });

    }

    private static void useFlyable(Flyable f) {
        f.fly("风和日丽，晴空万里");
    }
}

```

1.5 Lambda表达式练习3【应用】

- 练习描述

有参有返回值抽象方法的练习
- 操作步骤
 - 定义一个接口(Addable)，里面定义一个抽象方法：int add(int x,int y);
 - 定义一个测试类(AddableDemo)，在测试类中提供两个方法
 - 一个方法是：useAddable(Addable a)
 - 一个方法是主方法，在主方法中调用useAddable方法
- 示例代码

```

public interface Addable {
    int add(int x,int y);
}

public class AddableDemo {
    public static void main(String[] args) {
        //在主方法中调用useAddable方法
        useAddable((int x,int y) -> {
            return x + y;
        });
    }

    private static void useAddable(Addable a) {
        int sum = a.add(10, 20);
        System.out.println(sum);
    }
}

```

1.6 Lambda表达式的省略模式【应用】

- 省略的规则
 - 参数类型可以省略。但是有多个参数的情况下，不能只省略一个
 - 如果参数有且仅有一个，那么小括号可以省略
 - 如果代码块的语句只有一条，可以省略大括号和分号，和return关键字
- 代码演示

```

public interface Addable {
    int add(int x, int y);
}

public interface Flyable {
    void fly(String s);
}

public class LambdaDemo {
    public static void main(String[] args) {
        //      useAddable((int x,int y) -> {
        //          return x + y;
        //      });
        //参数的类型可以省略
        useAddable((x, y) -> {
            return x + y;
        });

        //      useFlyable((String s) -> {
        //          System.out.println(s);
        //      });
        //如果参数有且仅有一个，那么小括号可以省略
        useFlyable(s -> {
            System.out.println(s);
        });

        //如果代码块的语句只有一条，可以省略大括号和分号
        useFlyable(s -> System.out.println(s));

        //如果代码块的语句只有一条，可以省略大括号和分号，如果有return，return也要省略掉
        useAddable((x, y) -> x + y);
    }

    private static void useFlyable(Flyable f) {
        f.fly("风和日丽，晴空万里");
    }

    private static void useAddable(Addable a) {
        int sum = a.add(10, 20);
        System.out.println(sum);
    }
}

```

1.7 Lambda表达式的注意事项【理解】

- 使用Lambda必须要有接口，并且要求接口中有且仅有一个抽象方法
- 必须有上下文环境，才能推导出Lambda对应的接口
 - 根据局部变量的赋值得知Lambda对应的接口


```
Runnable r = () -> System.out.println("Lambda表达式");
```
 - 根据调用方法的参数得知Lambda对应的接口

```
new Thread(() -> System.out.println("Lambda表达式")).start();
```

1.8 Lambda表达式和匿名内部类的区别【理解】

- 所需类型不同
 - 匿名内部类：可以是接口，也可以是抽象类，还可以是具体类
 - Lambda表达式：只能是接口
- 使用限制不同
 - 如果接口中有且仅有一个抽象方法，可以使用Lambda表达式，也可以使用匿名内部类
 - 如果接口中多于一个抽象方法，只能使用匿名内部类，而不能使用Lambda表达式
- 实现原理不同
 - 匿名内部类：编译之后，产生一个单独的.class字节码文件
 - Lambda表达式：编译之后，没有一个单独的.class字节码文件。对应的字节码会在运行的时候动态生成

2.接口组成更新

2.1接口组成更新概述【理解】

- 常量
`public static final`
- 抽象方法
`public abstract`
- 默认方法(Java 8)
- 静态方法(Java 8)
- 私有方法(Java 9)

2.2接口中默认方法【应用】

- 格式
`public default 返回值类型 方法名(参数列表) { }`
- 范例

```
public default void show3() {  
}
```

- 注意事项
 - 默认方法不是抽象方法，所以不强制被重写。但是可以被重写，重写的时候去掉default关键字
 - public可以省略，default不能省略

2.3接口中静态方法【应用】

- 格式
`public static 返回值类型 方法名(参数列表) { }`
- 范例

```
public static void show() {  
}
```

- 注意事项
 - 静态方法只能通过接口名调用，不能通过实现类名或者对象名调用
 - public可以省略，static不能省略

2.4接口中私有方法【应用】

- 私有方法产生原因

Java 9中新增了带方法体的私有方法，这其实在Java 8中就埋下了伏笔：Java 8允许在接口中定义带方法体的默认方法和静态方法。这样可能会引发一个问题：当两个默认方法或者静态方法中包含一段相同的代码实现时，程序必然考虑将这段实现代码抽取成一个共性方法，而这个共性方法是不需要让别人使用的，因此用私有给隐藏起来，这就是Java 9增加私有方法的必然性

- 定义格式

- 格式1

private 返回值类型 方法名(参数列表) {}

- 范例1

```
private void show() {  
}
```

- 格式2

private static 返回值类型 方法名(参数列表) {}

- 范例2

```
private static void method() {  
}
```

- 注意事项

- 默认方法可以调用私有的静态方法和非静态方法
- 静态方法只能调用私有的静态方法

3.方法引用

3.1体验方法引用【理解】

- 方法引用的出现原因

在使用Lambda表达式的时候，我们实际上传递进去的代码就是一种解决方案：拿参数做操作

那么考虑一种情况：如果我们在Lambda中所指定的操作方案，已经有地方存在相同方案，那是否还有必要再写重复逻辑呢？答案肯定是不必要

那我们又是如何使用已经存在的方案的呢？

这就是我们要讲解的方法引用，我们是通过方法引用来使用已经存在的方案

- 代码演示

```
public interface Printable {
    void printString(String s);
}

public class PrintableDemo {
    public static void main(String[] args) {
        //在主方法中调用usePrintable方法
        //    usePrintable((String s) -> {
        //        System.out.println(s);
        //    });
        //Lambda简化写法
        usePrintable(s -> System.out.println(s));

        //方法引用
        usePrintable(System.out::println);
    }

    private static void usePrintable(Printable p) {
        p.printString("爱生活爱Java");
    }
}
```

3.2方法引用符【理解】

- 方法引用符
 - :: 该符号为引用运算符，而它所在的表达式被称为方法引用
- 推导与省略
 - 如果使用Lambda，那么根据“可推导就是可省略”的原则，无需指定参数类型，也无需指定的重载形式，它们都将被自动推导
 - 如果使用方法引用，也是同样可以根据上下文进行推导
 - 方法引用是Lambda的孪生兄弟

3.3引用类方法【应用】

引用类方法，其实就是引用类的静态方法

- 格式
类名::静态方法
- 范例
Integer::parseInt
Integer类的方法：public static int parseInt(String s) 将此String转换为int类型数据
- 练习描述
 - 定义一个接口(Converter)，里面定义一个抽象方法 int convert(String s);
 - 定义一个测试类(ConverterDemo)，在测试类中提供两个方法

- 一个方法是：useConverter(Converter c)
- 一个方法是主方法，在主方法中调用useConverter方法
- 代码演示

```
public interface Converter {
    int convert(String s);
}

public class ConverterDemo {
    public static void main(String[] args) {

        //Lambda写法
        useConverter(s -> Integer.parseInt(s));

        //引用类方法
        useConverter(Integer::parseInt);

    }

    private static void useConverter(Converter c) {
        int number = c.convert("666");
        System.out.println(number);
    }
}
```

- 使用说明

Lambda表达式被类方法替代的时候，它的形式参数全部传递给静态方法作为参数

3.4引用对象的实例方法【应用】

引用对象的实例方法，其实就引用类中的成员方法

- 格式

对象::成员方法
- 范例

"HelloWorld"::toUpperCase

String类中的方法：public String toUpperCase() 将此String所有字符转换为大写
- 练习描述
 - 定义一个类(PrintString)，里面定义一个方法

public void printUpper(String s)：把字符串参数变成大写的数据，然后在控制台输出
 - 定义一个接口(Printer)，里面定义一个抽象方法

void printUpperCase(String s)
 - 定义一个测试类(PrinterDemo)，在测试类中提供两个方法
 - 一个方法是：usePrinter(Printer p)
 - 一个方法是主方法，在主方法中调用usePrinter方法
- 代码演示

```

public class PrintString {
    //把字符串参数变成大写的数字，然后在控制台输出
    public void printUpper(String s) {
        String result = s.toUpperCase();
        System.out.println(result);
    }
}

public interface Printer {
    void printUpperCase(String s);
}

public class PrinterDemo {
    public static void main(String[] args) {

        //Lambda简化写法
        usePrinter(s -> System.out.println(s.toUpperCase()));

        //引用对象的实例方法
        PrintString ps = new PrintString();
        usePrinter(ps::printUpper);

    }

    private static void usePrinter(Printer p) {
        p.printUpperCase("HelloWorld");
    }
}

```

- 使用说明

Lambda表达式被对象的实例方法替代的时候，它的形式参数全部传递给该方法作为参数

3.5引用类的实例方法【应用】

引用类的实例方法，其实就是引用类中的成员方法

- 格式

类名::成员方法

- 范例

String::substring

```
public String substring(int beginIndex,int endIndex)
```

从beginIndex开始到endIndex结束，截取字符串。返回一个子串，子串的长度为endIndex-beginIndex

- 练习描述

- 定义一个接口(MyString)，里面定义一个抽象方法：

```
String mySubString(String s,int x,int y);
```

- 定义一个测试类(MyStringDemo)，在测试类中提供两个方法

- 一个方法是：useMyString(MyString my)

- 一个方法是主方法，在主方法中调用useMyString方法
- 代码演示

```
public interface MyString {
    String mySubString(String s,int x,int y);
}

public class MyStringDemo {
    public static void main(String[] args) {
        //Lambda简化写法
        useMyString((s,x,y) -> s.substring(x,y));

        //引用类的实例方法
        useMyString(String::substring);
    }

    private static void useMyString(MyString my) {
        String s = my.mySubString("HelloWorld", 2, 5);
        System.out.println(s);
    }
}
```

- 使用说明

Lambda表达式被类的实例方法替代的时候 第一个参数作为调用者 后面的参数全部传递给该方法作为参数

3.6引用构造器【应用】

引用构造器，其实就是引用构造方法

- I格式

类名::new
- 范例

Student::new
- 练习描述
 - 定义一个类(Student)，里面有两个成员变量(name,age)

并提供无参构造方法和带参构造方法，以及成员变量对应的get和set方法
 - 定义一个接口(StudentBuilder)，里面定义一个抽象方法

Student build(String name,int age);
 - 定义一个测试类(StudentDemo)，在测试类中提供两个方法
 - 一个方法是：useStudentBuilder(StudentBuilder s)
 - 一个方法是主方法，在主方法中调用useStudentBuilder方法
- 代码演示

```
public class Student {
    private String name;
    private int age;
```

```

public Student() {
}

public Student(String name, int age) {
    this.name = name;
    this.age = age;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}
}

public interface StudentBuilder {
    Student build(String name,int age);
}

public class StudentDemo {
    public static void main(String[] args) {

        //Lambda简化写法
        useStudentBuilder((name,age) -> new Student(name,age));

        //引用构造器
        useStudentBuilder(Student::new);

    }

    private static void useStudentBuilder(StudentBuilder sb) {
        Student s = sb.build("林青霞", 30);
        System.out.println(s.getName() + "," + s.getAge());
    }
}

```

- 使用说明

Lambda表达式被构造器替代的时候，它的形式参数全部传递给构造器作为参数