# Lambda 表达式

**主要内容**

Lambda 表达式介绍

Lambda 表达式语法

Lambda 表达式入门案例

Lambda 表达式的使用

**学习目标**

| 知识点 | 要求 |
| --- | --- |
| Lambda 表达式介绍 | 了解 |
| Lambda 表达式语法 | 掌握 |
| Lambda 表达式入门案例 | 掌握 |
| Lambda 表达式的使用 | 掌握 |

## 一、 Lambda 表达式介绍

### 1 Lambda 简介

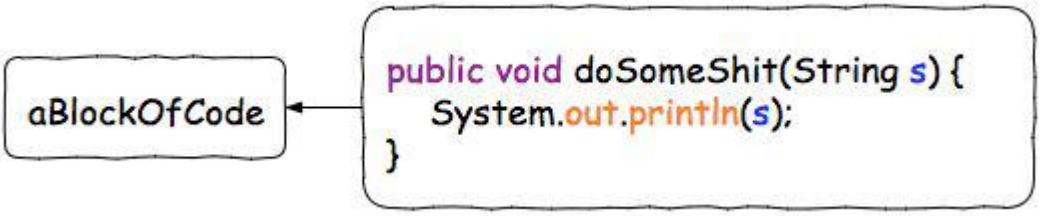

Lambda 表达式是 JDK8 的一个新特性，可以取代大部分的匿名内部类，写出更优雅的

Java 代码，尤其在集合的遍历和其他集合操作中，可以极大地优化代码结构。

在 Java 语言中，可以为变量赋予一个值。

```
int aValue = 129;
String aString = "Hello World!";
Boolean aBoolean = aString.startWith("H");
```

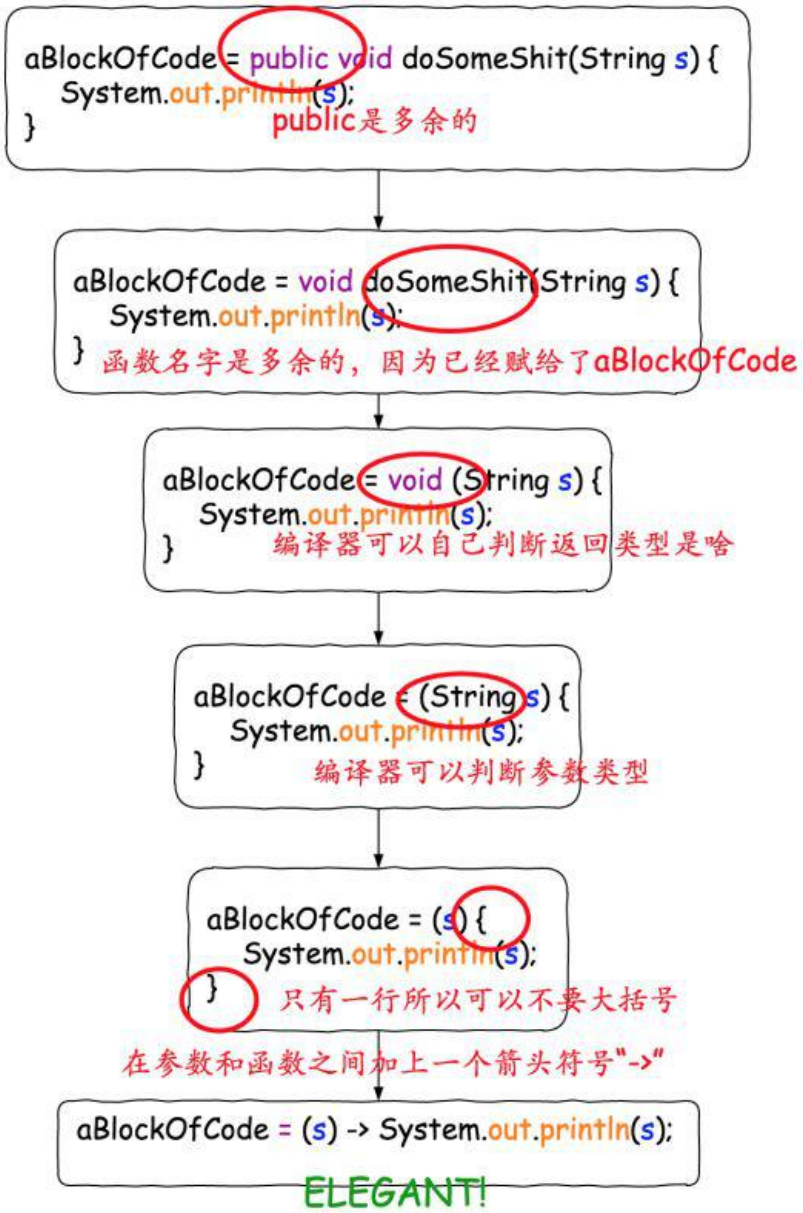
能否把一个代码块赋给一变量吗？




在 Java 8 之前，这个是做不到的。但是 Java 8 问世之后，利用 Lambda 特性，就可以做到了。

```
aBlockOfCode = public void doSomeShit(String s) {
    System.out.println(s);
}
```

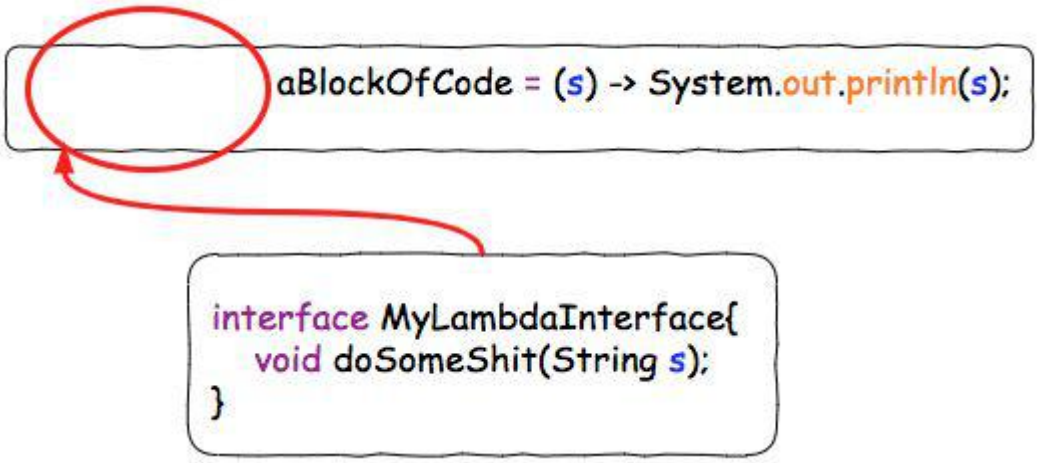
甚至我们可以让语法变得更简洁。



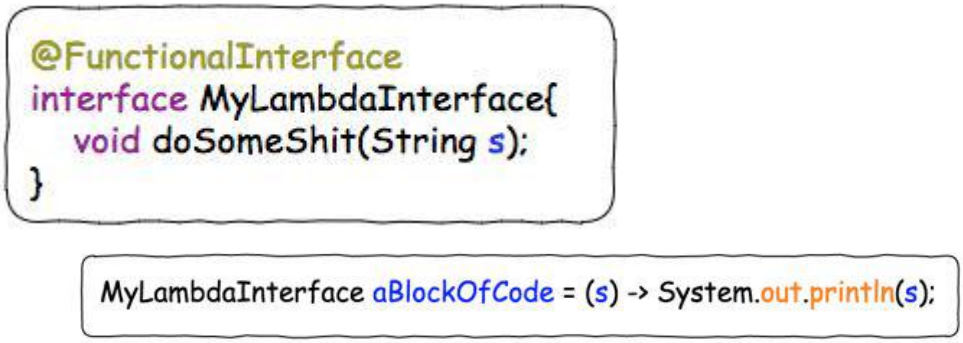

这样，我们就成功的非常优雅的把“一块代码”赋给了一个变量。而“这块代码”，或者说“这个被赋给一个变量的函数”，就是一个 Lambda 表达式。

但是这里仍然有一个问题，就是变量 aBlockOfCode 的类型应该是什么？

在 Java 8 里面，所有的 Lambda 的类型都是一个接口，而 Lambda 表达式本身，也就是”那段代码“，需要是这个接口的实现。这是我认为理解 Lambda 的一个关键所在，简而言之就是，Lambda 表达式本身就是一个接口的实现。直接这样说可能还是有点让人困扰，我们继续看看例子。我们给上面的 aBlockOfCode 加上一个类型：

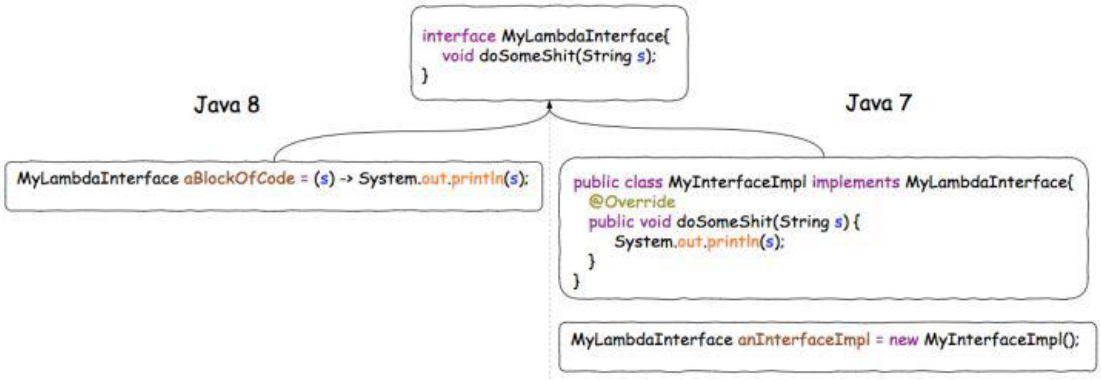



这种只有一个接口函数需要被实现的接口类型，我们叫它"函数式接口"。为了避免后来的人在这个接口中增加接口函数导致其有多个接口函数需要被实现，变成"非函数接口"，我们可以在这个上面加上一个声明@FunctionalInterface，这样别人就无法在里面添加新的接口函数了。




## 2 Lambda 作用

最直观的作用就是使得代码变得异常简洁。



### 3 接口要求

虽然使用 Lambda 表达式可以对某些接口进行简单的实现，但并不是所有的接口都可以使用 Lambda 表达式来实现。Lambda 规定接口中只能有一个需要被实现的方法，不是规定接口中只能有一个方法。

jdk 8 中有另一个新特性：default， 被 default 修饰的方法会有默认实现，不是必须被实现的方法，所以不影响 Lambda 表达式的使用。

### 4 @FunctionalInterface 注解作用

@FunctionalInterface 标记在接口上，“函数式接口”是指仅仅只包含一个抽象方法的接口。

## 二、 Lambda 表达式语法

### 1 语法结构

```
(parameters) -> expression
或
(parameters) ->{ statements; }
```

语法形式为 () -> {}，其中 () 用来描述参数列表，{} 用来描述方法体，-> 为 lambda 运算符 ，读作(goes to)。

### 2 Lambda 表达式的重要特征

可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。

可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。

可选的大括号：如果主体包含了一个语句，就不需要使用大括号。

可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需

要指定明表达式返回了一个数值。

### 3 Lambda 案例

```
// 1. 不需要参数,返回值为 5
() -> 5

// 2. 接收一个参数(数字类型),返回其 2 倍的值
x -> 2 * x

// 3. 接受 2 个参数(数字),并返回他们的差值
(x, y) -> x – y

// 4. 接收 2 个 int 型整数,返回他们的和
(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回 void)
(String s) -> System.out.print(s)
```

## 三、 Lambda 表达式入门案例

### 1 定义函数接口

```
/**
 * 无返回值，无参数
 */
@FunctionalInterface
interface NoReturnNoParam{
    void method();
}

/**
 * 无返回值，有一个参数
 */
```

```
@FunctionalInterface
interface NoReturnOneParam{
    void method(int a);
}

/**
 * 无返回值，有多个参数
 */
@FunctionalInterface
interface NoReturnMultiParam{
    void method(int a,int b);
}

/**
 * 有返回值，无参数
 */
@FunctionalInterface
interface ReturnNoParam{
    int method();
}

/**
 * 有返回值，有一个参数
 */
@FunctionalInterface
interface ReturnOneParam{
    int method(int a);
}

/**
 * 有返回值，有多个参数
 */
@FunctionalInterface
interface ReturnMultiParam{
    int method(int a,int b);
}
```

## 2 实现函数接口

```
public static void main(String[] args) {
    /**
     * 无返回值，无参数
     */
    NoReturnNoParam noReturnNoParam = ()->{
        System.out.println("NoReturnNoParam");
    };
    noReturnNoParam.method();

    /**
     * 无返回值，有一个参数
     */
    NoReturnOneParam noReturnOneParam = (int a)->{
        System.out.println("NoReturnOneParam "+a);
    };
     noReturnOneParam.method(10);

    /**
     * 无返回值，有多个参数
     */
    NoReturnMultiParam noReturnMultiParam = (int a, int b)->{
        System.out.println("NoReturnMultiParam "+a+"\t"+b);
    };
    noReturnMultiParam.method(10,20);

    /**
     * 有返回值，无参数
     */
    ReturnNoParam returnNoParam = ()->{
        System.out.print("ReturnNoParam ");
        return 10;
    };
    System.out.println(returnNoParam.method());

    /**
     * 有返回值，有一个参数
     */
```

```
    ReturnOneParam returnOneParam = (int a)->{
        System.out.print("ReturnOneParam ");
        return a;
    };
    System.out.println(returnOneParam.method(10));

    /**
     * 有返回值，有多个参数
     */
    ReturnMultiParam returnMultiParam = (int a ,int b)->{
        System.out.print("ReturnMultiParam ");
        return a+b;
    };
    System.out.println(returnMultiParam.method(10,20));

}
```

## 3 Lambda 语法简化

```
 /**
     * 无返回值，无参数
     */
   /* NoReturnNoParam noReturnNoParam = ()->{
        System.out.println("NoReturnNoParam");
    };*/
    /**
     * 简化版
     */
    NoReturnNoParam noReturnNoParam = ()->
System.out.println("NoReturnNoParam");
    noReturnNoParam.method();

    /**
     * 无返回值，有一个参数
     */
   /* NoReturnOneParam noReturnOneParam = (int a)->{
        System.out.println("NoReturnOneParam "+a);
    };*/
    /**
```

```
     * 简化版
     */
    NoReturnOneParam noReturnOneParam = a ->
System.out.println("NoReturnOneParam "+a);
     noReturnOneParam.method(10);

    /**
     * 无返回值，有多个参数
     */
   /* NoReturnMultiParam noReturnMultiParam = (int a, int b)->{
        System.out.println("NoReturnMultiParam "+a+"\t"+b);
    };*/
    NoReturnMultiParam noReturnMultiParam =(a,b)->
System.out.println("NoReturnMultiParam "+a+"\t"+b);
    noReturnMultiParam.method(10,20);

    /**
     * 有返回值，无参数
     */
   /* ReturnNoParam returnNoParam = ()->{
        System.out.print("ReturnNoParam ");
        return 10;
    };*/
    /**
     * 简化版
     */
    ReturnNoParam returnNoParam = ()->10+20;
    System.out.println(returnNoParam.method());

    /**
     * 有返回值，有一个参数
     */
  /*  ReturnOneParam returnOneParam = (int a)->{
        System.out.print("ReturnOneParam ");
        return a;
    };*/
    /***
     * 简化版
```

```
     */
    ReturnOneParam returnOneParam = a->a;
    System.out.println(returnOneParam.method(10));

    /**
     * 有返回值，有多个参数
     */
    /*ReturnMultiParam returnMultiParam = (int a ,int b)->{
       System.out.print("ReturnMultiParam ");
       return a+b;
    };*/
    /**
     * 简化版
     */
    ReturnMultiParam returnMultiParam = (a ,b)->a+b;
    System.out.println(returnMultiParam.method(10,20));

}
```

# 四、 Lambda 表达式的使用

## 1 Lambda 表达式引用方法

有时候我们不是必须使用 Lambda 的函数体定义实现，我们可以利用 lambda 表达式指向一个已经被实现的方法。

### 1.1语法

方法归属者::方法名 静态方法的归属者为类名，普通方法归属者为对象。

### 1.2案例

```
/**
 * 无返回值，无参数
 */
@FunctionalInterface
```

```
interface NoReturnNoParam{
    void method();
}

/**
 * 无返回值，有一个参数
 */
@FunctionalInterface
interface NoReturnOneParam{
    void method(int a);
}

/**
 * 无返回值，有多个参数
 */
@FunctionalInterface
interface NoReturnMultiParam{
    void method(int a,int b);
}

/**
 * 有返回值，无参数
 */
@FunctionalInterface
interface ReturnNoParam{
    int method();
}

/**
 * 有返回值，有一个参数
 */
@FunctionalInterface
interface ReturnOneParam{
    int method(int a);
}

/**
 * 有返回值，有多个参数
 */
```

```
@FunctionalInterface
interface ReturnMultiParam{
    int method(int a,int b);
}
public class Test {
    public static void main(String[] args) {
        /**
         * 无返回值，无参数
         */
       /* NoReturnNoParam noReturnNoParam = ()->{
            System.out.println("NoReturnNoParam");
        };*/
        /**
         * 简化版
         */
        NoReturnNoParam noReturnNoParam = ()->
System.out.println("NoReturnNoParam");
        noReturnNoParam.method();

        /**
         * 无返回值，有一个参数
         */
       /* NoReturnOneParam noReturnOneParam = (int a)->{
            System.out.println("NoReturnOneParam "+a);
        };*/
        /**
         * 简化版
         */
        NoReturnOneParam noReturnOneParam = a ->
System.out.println("NoReturnOneParam "+a);
         noReturnOneParam.method(10);

        /**
         * 无返回值，有多个参数
         */
       /* NoReturnMultiParam noReturnMultiParam = (int a, int b)->{
            System.out.println("NoReturnMultiParam "+a+"\t"+b);
        };*/
        NoReturnMultiParam noReturnMultiParam =(a,b)->
```

```
System.out.println("NoReturnMultiParam "+a+"\t"+b);
        noReturnMultiParam.method(10,20);

        /**
         * 有返回值，无参数
         */
       /* ReturnNoParam returnNoParam = ()->{
            System.out.print("ReturnNoParam ");
            return 10;
        };*/
        /**
         * 简化版
         */
        ReturnNoParam returnNoParam = ()->10+20;
        System.out.println(returnNoParam.method());

        /**
         * 有返回值，有一个参数
         */
      /*  ReturnOneParam returnOneParam = (int a)->{
            System.out.print("ReturnOneParam ");
            return a;
        };*/
        /***
         * 简化版
         */
        ReturnOneParam returnOneParam = a->a;
        System.out.println(returnOneParam.method(10));

        /**
         * 有返回值，有多个参数
         */
        /*ReturnMultiParam returnMultiParam = (int a ,int b)->{
            System.out.print("ReturnMultiParam ");
            return a+b;
        };*/
        /**
         * 简化版
```

```
        */
        ReturnMultiParam returnMultiParam = (a ,b)->a+b;
        System.out.println(returnMultiParam.method(10,20));

    }
    /**
     * 要求：
     * 1,参数的个数以及类型需要与函数接口中的抽象方法一致。
     * 2,返回值类型要与函数接口中的抽象方法的返回值类型一致。
     * @param a
     * @return
     */
    public static int doubleNum(int a){
        return 2*a;
    }
    public int addTwo(int a){
        return a+2;
    }
}
```

```
public class Test2 {
    public static void main(String[] args) {
        ReturnOneParam returnOneParam = Test::doubleNum;
        int value = returnOneParam.method(10);
        System.out.println(value);

        Test test = new Test();
        ReturnOneParam returnOneParam1 = test::addTwo;
        int value2 = returnOneParam1.method(10);
        System.out.println(value2);
    }
}
```

## 2 Lambda 表达式创建线程

```
public class Test3 {
    public static void main(String[] args) {

        System.out.println(Thread.currentThread().getName()+" 开始
");
```

```
        new Thread(()->{
            for(int i=0;i<20;i++){
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

System.out.println(Thread.currentThread().getName()+" "+i);
            }
        },"Lambda Thread ").start();

        System.out.println(Thread.currentThread().getName()+" 结束
");
    }
}
```

# 3 操作集合

## 3.1遍历集合

我们可以调用集合的 public void forEach(Consumer<? super E> action) 方法，通过 lambda 表达式的方式遍历集合中的元素。以下是 Consumer 接口的方法以及遍历集合的操作。Consumer 接口是 jdk 为我们提供的一个函数式接口。

```
public class Test4 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        list.add("d");
        list.forEach(System.out::println);
    }
}
```

### 3.2删除集合中的元素

我们通过 public boolean removeIf(Predicate<? super E> filter)方法来删除集合中的某个元素，Predicate 也是 jdk 为我们提供的一个函数式接口，可以简化程序的编写。

```
public class Test5 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        list.add("d");
        list.removeIf(ele->ele.equals("b"));
        list.forEach(System.out::println);
    }
}
```

### 3.3元素排序

之前我们若要为集合内的元素排序，就必须调用 sort 方法，传入比较器重写 compare 方法的比较器对象，现在我们还可以使用 lambda 表达式来简化代码。

```
public class Test6 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("a");
        list.add("d");
        list.add("b");
        list.add("c");
        list.sort((o1,o2)->o1.compareTo(o2));
        list.forEach(System.out::println);
    }
}
```

## 4 Lambda 表达式中的闭包问题

### 4.1什么是闭包

闭包的本质就是代码片断。所以闭包可以理解成一个代码片断的引用。在 Java 中匿名

内部类也是闭包的一种实现方式。

在闭包中访问外部的变量时，外部变量必须是 final 类型，虚拟机会帮我们加上 final 修饰关键字。

```
public class Test7 {
    public static void main(String[] args) {
        int num  =10;
        NoReturnNoParam noReturnNoParam =
()->System.out.println(num);
        noReturnNoParam.method();
    }
}
```