第十九章 Lambda表达式

1. Lambda表达式的概述

1. Lambda表达式：是JDK8新增的一个技术
2. Lambda表达式的作用
   a. 使代码更简洁，更加优雅
   b. 并没有提升程序的执行效率
3. 体验一下 Lambda表达式

代码块

```
1  package com.powernode.lambda03;
2
3  import java.util.Arrays;
4  import java.util.Comparator;
5  import java.util.List;
6
7  public class Test01 {
8      public static void main(String[] args) {
9          //创建一个包含整数元素的list集合，使用Arrays.asList直接创建（不支持添加和删除，
   支持修改）
10         List<Integer> list = Arrays.asList(3, 1, 2, 5, 4);
11         //需求1：想对集合中元素进行升序排序
12        /* list.sort(new Comparator<Integer>() {
13             @Override
14             public int compare(Integer o1, Integer o2) {
15                 return o1 - o2;
16             }
17         });*/
18         list.sort(Integer::compareTo);
19         System.out.println(list);
20     }
21 }
```

4. 什么情况下可以使用 Lambda表达式
   a. 必须是接口
   b. 接口中只有一个抽象方法，可以包含其他非抽象方法

c. 只有一个抽象的方法的接口，称为**函数式接口，函数式接口使用 @FunctionalInterface 进行标识|约束**

代码块

```
package com.powernode.lambda03;

@FunctionalInterface //约束改接口只能有一个抽象方法
interface Flyer{
    void land();
   // void fly();
    private void takeOff(){
        System.out.println("Flyer.takeOff");
    }
}
public class Test02 {
    @Override //约束该方法必须重写父类的
    public String toString() {
        return super.toString();
    }
}
```

2. Lambda表达式深入体验

代码块

```
package com.powernode.lambda03;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Test03 {
    public static void main(String[] args) {
        //创建一个包含整数元素的list集合，使用Arrays.asList直接创建（不支持添加和删除，支持修改）
        List<Integer> list = Arrays.asList(3, 1, 2, 5, 4);
        //需求1：对集合中元素进行排序
        list.sort(new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o1 - o2;
            }
        });
        System.out.println(list);
        /**
```

```
20            * 1.发现Comparator 是一个函数式接口，可以使用Lambda表达式
21            * 2.Lambda表达式语法：
22            *     1.把【函数式接口的参数列表】  和 【方法体】 使用 -> 连接起来
23            *     2.语法：(形参列表) -> {方法体} ：函数式接口的抽象方法
24            *     3.可以把Lambda表达式看成一个匿名方法
25            */
26           list.sort((Integer o1, Integer o2) ->{
27                   return o1 - o2;
28               }
29           );
30           System.out.println(list);
31           /**
32            * 优化1：
33            *     1.参数类型可以省略（会根据上下文推断出参数类型）
34            *     2.当参数列表只有一个参数时（）可以省略
35            */
36           list.sort(( o1,  o2) ->{
37                       return o1 - o2;
38                   }
39           );
40           System.out.println(list);
41           //优化2：当方法体只有一条语句时，return，{}和分号 可以省略
42           list.sort(( o1,  o2) -> o1 - o2);
43           System.out.println(list);
44           //优化3：方法方法引用，后面专题讲，提前体验一下
45           list.sort(Integer::compareTo);
46           System.out.println(list);
47
48       }
49   }
```

3. Lambda表达式语法总结

1. 语法
   a. *(形参列表) -> {方法体}*
   b. 函数式接口抽象方法： *(形参列表) -> {方法体}*
2. 语法规则
   a. *(形参列表)*
      i. *【参数类型】可以省略*
      ii. *【参数类型】只有一个参数， () 可以省略*
   b. *{方法体}：当方法体只有一条语句时，return，{}和分号 可以省略*

c. 方法引用（实例方法）

```
函数式接口中方法        【返回类型】      和【参数】
与
内部方法                【返回类型】      和【参数】
返回类型兼容，参数值一样
```

4. 四个基本的函数式接口

名称	接口名	对应的抽象方法
消费型接口	Consumer<T>	void accept(T t);
转换型接口	Function<T, R>	R apply(T t);
判断型接口	Predicate<T>	boolean test(T t);
生产型接口	Supplier<T>	T get();

4.1 函数式接口-Consumer

```java
package com.powernode.lambda04;

@FunctionalInterface
public interface Consumer<T> {
    /**
     * 1. T 通过泛型参数来约束参数类型
     * 2. accept(T t)，处理一个T类型的参数，不返回结果
     *    比如：打印t
     */
    void accept(T t);
}


@FunctionalInterface
interface Comparator<T> {
    /**
     * 1.T 通过泛型参数来约束参数类型
     * 2.compare(T o1, T o2)：编写比较规则
```

```
19          *    1. o1 > o2 返回正数
20          *    2. o1 < o2 返回负数
21          *    3. o1 = o2 返回0
22          */
23         int compare(T o1, T o2);
24     }
```

4.2 函数式接口-Function

代码块

```
1
2   @FunctionalInterface
3   public interface Function<T, R> {
4       //接收一个T类型的参数，返回一个R类型的结果
5       R apply(T t);
6   }
```

4.3 函数式接口-Predicate

代码块

```
1   package com.powernode.lambda04;
2
3   @FunctionalInterface
4   public interface Predicate<T> {
5       //判断一个元素是否满足某些条件，满足返回ture,否则返回false
6       boolean test(T t);
7   }
```

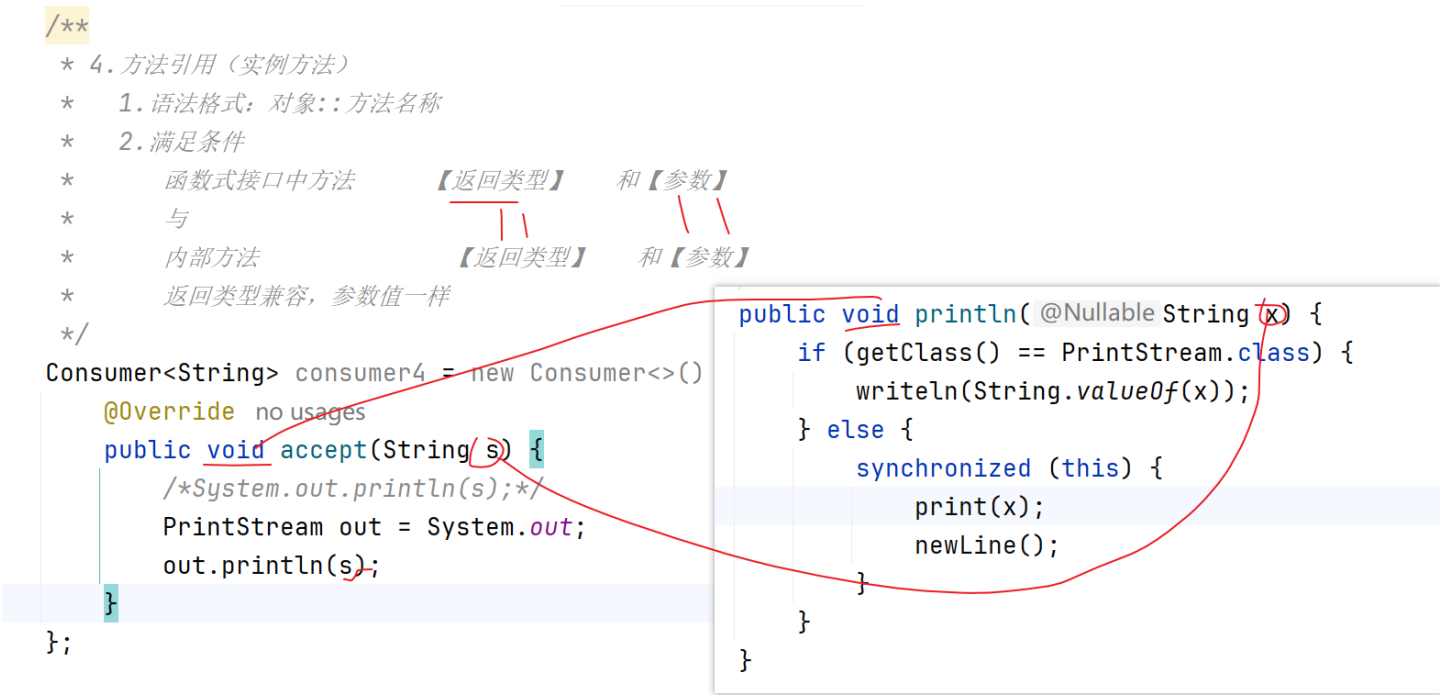
4.4 函数式接口-Supplier

代码块

```
1   @FunctionalInterface
2   public interface Supplier<T> {
3       //返回一个泛型参数类型
4       T get();
5   }
```

5. 函数式接口的基本应用

5.1 Consumer使用Lambda表达式



代码块

```
package com.powernode.lambda05;

import com.powernode.lambda04.Consumer;

import java.io.PrintStream;

public class Test01 {
    public static void main(String[] args) {
        //-匿名内部类
        Consumer<String> consumer = new Consumer<>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        };
        //1.使用lambda表达式：(形参类别)->{方法体}
        Consumer<String> consumer1 = (String s) ->{
                System.out.println(s);
        };
        /**
         * 2.形参列表
         *    1.参数类型可以省略
         *    2.只有一个参数()可以省略
```

```
24            */
25           Consumer<String> consumer2 =  s ->{
26               System.out.println(s);
27           };
28           //3.方法体：当方法体只有一条语句时，return，{}和分号 可以省略
29           Consumer<String> consumer3 =  s -> System.out.println(s);
30           /**
31            * 4.方法引用（实例方法）
32            *   1.语法格式：对象::方法名称
33            *   2.满足条件
34            *       函数式接口中方法       【返回类型】    和【参数】
35            *       与
36            *       内部方法               【返回类型】    和【参数】
37            *       返回类型兼容，参数值一样
38            */
39           Consumer<String> consumer4 = new Consumer<>() {
40               @Override
41               public void accept(String s) {
42                   /*System.out.println(s);*/
43                   PrintStream out = System.out;
44                   out.println(s);
45               }
46           };
47           Consumer<String> consumer5 = System.out::println;
48       }
49   }
```

5.2 Consumer实际应用（掌握）

代码块

```
1   package com.powernode.lambda05;
2
3   import java.util.Arrays;
4   import java.util.Iterator;
5   import java.util.List;
6   import java.util.function.Consumer;
7
8   public class Test02 {
9       public static void main(String[] args) {
10          //-匿名内部类
11          /* Consumer<String> consumer = new Consumer<>() {
12              @Override
13              public void accept(String s) {
14                  System.out.println(s);
15              }
```

```
        };
        consumer.accept("hello");*/
        //System.out.println("hello");
        List<String> list = Arrays.asList("Hello", "Word", "Java");
        //1.传统写法遍历
        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String next = iterator.next();
            System.out.println(next);
        }
        System.out.println("----------");
        //2.使用forEach
        list.forEach(new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        });
        //3.lambda :(形参列表)->{方法体}
        list.forEach((String s) ->{
                System.out.println(s);
        });
        //4.(形参列表) : 参数类型可以省略，只有一个参数()可以省略
        list.forEach( s ->{
            System.out.println(s);
        });
        //5. {方法体} :只有一条语句时，return ,{}和;可以省略
        list.forEach( s -> System.out.println(s));
        //6.方法引用:
        list.forEach(System.out::println);
    }
}
```

5.3 Predicate实际应用

代码块

```
package com.powernode.lambda05;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

```

```java
public class Test03 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("hello");
        list.add("word");
        list.add("java");
        //1.删除包含o的元素
        list.removeIf(new Predicate<String>() {
            @Override
            public boolean test(String s) {//拿到集合中每个元素给s
                return s.contains("o");//返回true删除，返回false不删除
            }
        });
        //2.lambda
        list.removeIf((String s)-> {
                return s.contains("o");
        });
        //3.形参列表
        list.removeIf( s-> {
            return s.contains("o");
        });
        //4.方法体
        list.removeIf( s-> s.contains("o"));
        System.out.println(list);//[java]
    }
}
```

6. Lambda表达式和匿名内部类

1. 所需类型不同
   a. **Lambda表达式：只能使用函数式接口**
   b. 匿名内部类：可以是接口，抽象类或者具体类都可以
2. 使用现在不同
   a. **Lambda表达式：接口使用一个函数式接口（只有一个抽象方法）**
   b. 如果接口中有多个抽象方法，只能使用匿名内部类

7. Lambda表达式的方法引用（掌握）

7.1 实例方法引用

7.1.1 语法规则和满足条件

1. 语法规则：对象名称::方法名称
2. 满足条件

```
函数式接口中方法      【返回类型】   和  【参数】
与
内部方法              【返回类型】   和  【参数】
    返回类兼容，参数值一样
```

7.1.2 案例应用

```
package com.powernode.lambda06;

import com.powernode.lambda04.Supplier;

class Teacher{
    private String name;

    public Teacher(String name) {
        this.name = name;
    }

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

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "name='" + name + '\'' +
                '}';
    }
}
public class Test01 {
    public static void main(String[] args) {
        Teacher teacher = new Teacher("zs");
        Supplier<String> supplier = new Supplier<>() {
            @Override
            public String get() {
```

```
33                  return teacher.getName();
34              }
35          };
36          /**
37           * 函数式接口中方法      【返回类型】   和  【参数】
38           * 与
39           * 内部方法             【返回类型】   和  【参数】
40           *      返回类兼容，参数值一样
41           */
42          //语法规则：对象名称::方法名称
43          Supplier<String> supplier1 = teacher::getName;
44      }
45  }
```

7.1.3 输出语句的方法引用

代码块

```
1   package com.powernode.lambda06;
2
3   import com.powernode.lambda04.Consumer;
4
5   public class Test02 {
6       public static void main(String[] args) {
7           Consumer<String> consumer = new Consumer<>() {
8               @Override
9               public void accept(String s) {
10                  System.out.println(s);
11              }
12          };
13
14          Consumer<String> consumer1 = System.out::println;
15
16
17      }
18  }
```

7.2 静态方法引用

7.2.1 语法规则和满足条件

1. 语法规则：类名称::方法名称

2. 满足条件

```
函数式接口中方法      【返回类型】   和  【参数】
与
内部方法              【返回类型】   和  【参数】
    返回类兼容，参数值一样
```

7.2.2 案例应用

```java
package com.powernode.lambda06;

import com.powernode.lambda04.Function;

public class Test03 {
    public static void main(String[] args) {
        //蒋一个字符串转换为Integer类型
        Function<String, Integer> function = new Function<>() {
            @Override
            public Integer apply(String s) {
                return Integer.parseInt(s);
            }
        };
        /**
         * 函数式接口中方法      【返回类型】   和  【参数】
         * 与
         * 内部方法              【返回类型】   和  【参数】
         *     返回类兼容，参数值一样
         */
        Function<String, Integer> function1 = Integer::parseInt;
        Integer num = function1.apply("123");
        System.out.println(num);
    }
}
```

7.3 特殊方法引用

7.3.1 语法规则和满足条件

1. 语法规则：类名称::方法名称
2. 满足条件：

代码块

```
package com.powernode.lambda06;

import com.powernode.lambda04.Function;

import java.util.Comparator;

public class Test04 implements Comparator<Integer> {

    /**
     * 1.函数式接口方法中第一个参数，作为内部方法的调用对象
     * 2.函数式接口方法从第二个参数开始，与内部方法的参数值一样
     * 3.返回类型兼容
     *
     */
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1.compareTo(o2);
    }
}
```

7.3.2 特殊方法引用案例应用（一）

代码块

```
package com.powernode.lambda06;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Test05 {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 0, 3, 4, 5);
       /* list.sort(new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o1.compareTo(o2);
            }
        });*/
        //类名称::方法名称
        list.sort(Integer::compareTo);
        System.out.println(list);
    }


}
```

7.3.3 特殊方法引用案例应用（二）

代码块

```
  package com.powernode.lambda06;

import com.powernode.lambda04.Function;

class User{
    private String name;

    public User(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
public class Test06 {
    public static void main(String[] args) {
        Function<User, String> function = new Function<>() {
            @Override
            public String apply(User user) {
                return user.getName();
            }
        };

        Function<User, String> function1 = User::getName;

    }


}
```

7.4 构造器方法引用

7.4.1 语法规则和满足条件

代码块

```
语法规则：类名称::new
满足条件
      1.函数式接口方法的参数      与   构造器方法参数值一样
      2.返回类型兼容
```

7.4.2 构造器方法引用（无参）

代码块

```
package com.powernode.lambda07;

import com.powernode.lambda04.Supplier;

class Student{
    public Student() {
        System.out.println("Student.Student");
    }
}
public class Test01 {
    public static void main(String[] args) {
        Supplier<Student> supplier = new Supplier<>() {
            @Override
            public Student get() {
                return new Student();
            }
        };

        /**
         * 语法规则：类名称::new
         * 满足条件
         *      1.函数式接口方法的参数      与   构造器方法参数值一样
         *      2.返回类型兼容
         */
        Supplier<Student> supplier1 = Student::new;

        Supplier<Object> supplier2 = new Supplier<>() {
            @Override
            public Object get() {
                return new Student();
            }
        };

        Supplier<Object> supplier3 = Student::new;
        Object o = supplier3.get();

    }
```

```
38  }
```

7.4.3 构造器方法引用（有参）

代码块

```
1   package com.powernode.lambda08;
2
3   import com.powernode.lambda04.Function;
4
5   class Student{
6       private String name;
7
8       public Student(String name) {
9           this.name = name;
10      }
11  }
12  public class Test {
13      public static void main(String[] args) {
14          Function<String, Student> function = new Function<>() {
15              @Override
16              public Student apply(String s) {
17                  return new Student(s);
18              }
19          };
20
21          Function<String, Student> function1 = Student::new;
22          function1.apply("zs");
23
24      }
25  }
```

7.4.4 构造器方法引用（多参）

代码块

```
1   package com.powernode.lambda09;
2
3   import java.util.function.BiFunction;
4
5   class Student{
6       private String name;
7       private int age;
8       private  char sex;
9
```

```
10      public Student(String name, int age) {
11          this.name = name;
12          this.age = age;
13      }
14
15      public Student(String name, int age, char sex) {
16          this.name = name;
17          this.age = age;
18          this.sex = sex;
19      }
20
21      @Override
22      public String toString() {
23          return "Student{" +
24                  "name='" + name + '\'' +
25                  ", age=" + age +
26                  ", sex=" + sex +
27                  '}';
28      }
29  }
30  @FunctionalInterface
31  interface TriFunction<T, U, V,R> {
32      R apply(T t, U u,V v);
33  }
34  public class Test {
35      public static void main(String[] args) {
36          System.out.println("===============两个参数=================");
37          BiFunction<String, Integer, Student> biFunction = new BiFunction<>() {
38              @Override
39              public Student apply(String name, Integer age) {
40                  return new Student(name, age);
41              }
42          };
43          BiFunction<String, Integer, Student> biFunction1 = Student::new;
44          System.out.println(biFunction1.apply("zs", 23));
45          System.out.println("===============三个参数=================");
46          TriFunction<String, Integer, Character, Student> triFunction = new
    TriFunction<>() {
47              @Override
48              public Student apply(String name, Integer age, Character sex) {
49                  return new Student(name,age,sex);
50              }
51          };
52          TriFunction<String, Integer, Character, Student> triFunction1 =
    Student::new;
53          System.out.println(triFunction1.apply("zs", 23, '男'));
54      }
```

```
55  }
```

7.5 数组方法引用

7.5.1 语法规则和满足条件

1. 语法规则：数据类型 [] ::new
2. 满足条件
   a. 函数式接口中的方法只有一个整型参数
   b. 这个整型参数，正好是创建数组时指定的数组长度
   c. 返回类型兼容

7.5.2 案例应用

代码块

```
1   package com.powernode.lambda10;
2
3   import com.powernode.lambda04.Function;
4
5   public class Test {
6       public static void main(String[] args) {
7           Function<Integer, int[]> function = new Function<>() {
8               @Override
9               public int[] apply(Integer length) {
10                  return new int[length];
11              }
12          };
13          /**
14           * 1. 语法规则：数据类型  [] ::new
15           * 2. 满足条件
16           *   1. 函数式接口中的方法只有一个整型参数
17           *   2. 这个整型参数，正好是创建数组时指定的数组长度
18           *   3. 返回类型兼容
19           */
20          Function<Integer, int[]> function1 = int []::new;
21      }
22  }
```

8. 遍历List，Set和Map（掌握）

```java
package com.powernode.lambda11;

import java.util.*;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

public class Test {
    public static void main(String[] args) {
        //1.遍历List
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
        /*list.forEach(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) {
                System.out.println(integer);
            }
        });*/
        list.forEach(System.out::println);//soutc
        //2.遍历Set
        Set<Integer> set = new TreeSet<>();
        set.add(11);
        set.add(22);
        set.add(33);
        set.forEach(System.out::println);
        //3.遍历map
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "zs");
        map.put(2, "ls");
        map.put(3, "ww");
        map.forEach(new BiConsumer<Integer, String>() {
            @Override
            public void accept(Integer key, String value) {
                System.out.println(key + ":" + value);
            }
        });
        // -- 判断是否可以使用方法引用，其次判断是否可以使用lambda表达式
        //1.lambda:(形参列表) ->{方法体}
        map.forEach((Integer key, String value) ->{
                System.out.println(key + ":" + value);
            }
        );
        //2.(形参列表)
        map.forEach(( key,  value) ->{
                    System.out.println(key + ":" + value);
                }
        );
        //3.{方法体}
        map.forEach(( key,  value) -> System.out.println(key + ":" + value));
```

```
48        }
49    }
```